Министерство образования и науки
Российской Федерации

Федеральное агентство по образованию

Нижегородский государственный университет
им. Н.И. Лобачевского

В.Е. АЛЕКСЕЕВ, В.А. ТАЛАНОВ

ГРАФЫ.
МОДЕЛИ ВЫЧИСЛЕНИЙ.
СТРУКТУРЫ ДАННЫХ

Учебник

Рекомендовано Научно-методическим советом по прикладной математике
и информатике УМО университетов РФ в качестве учебника
для студентов, обучающихся по специальности
010200 – Прикладная математика и информатика
и по направлению 510200 – Прикладная математика и информатика

Нижний Новгород
Издательство Нижегородского госуниверситета
2005

УДК 519.17+510.58+681.142
ББК В12
А 47

Р е ц е н з е н т ы :
д.ф.-м.н., проф. НГТУ В.М. Галкин
д.ф.-м.н., проф. НГПУ М.И. Иорданский

Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород: Изд-во ННГУ, 2005. 307 с.

ISBN 5–85747–810–8

Учебник состоит из трех частей, посвященных вопросам анализа и разработки алгоритмов: графы и алгоритмы, модели вычислений, структуры данных. Для понимания материала достаточно математической подготовки в объеме первого курса университета или технического вуза.

Предназначен для студентов, обучающихся по направлению 510200 – Прикладная математика и информатика и по специальности 010200 – Прикладная математика и информатика.

Работа выполнена в учебно-исследовательской лаборатории «Математические и программные технологии для современных компьютерных систем (Информационные технологии)» при поддержке корпорации «Интел».

ББК В12

ISBN 5–85747–810–8

Предисловие

В этой книге под одной обложкой собраны учебные тексты, на первый взгляд разнородные, но относящиеся к одной сравнительно молодой области человеческой деятельности. Это деятельность по созданию и исследованию алгоритмов, для которой пока не придумано общеупотребительного объединяющего названия (она является частью того, что охватывается терминами «computer science» и «информатика»). Работа в этой области требует определенных математических знаний и представления о проблемах, связанных с разработкой компьютерных программ, но она не сводится к математике или программированию. Ее роль можно сравнить с ролью технологии по отношению к науке и производству.

Материал настоящего учебника в целом соответствует программе курса «Анализ и разработка алгоритмов», читавшегося в течение ряда лет для магистрантов факультета ВМК ННГУ, а отдельные его фрагменты используются в различных спецкурсах. Книга состоит из трех частей, которые могут изучаться независимо друг от друга и в произвольном порядке.

Первая часть посвящена алгоритмам на графах. Значительный объем в ней занимает глава 1, в которой приводятся базовые понятия и факты из теории графов. В других двух главах излагаются некоторые алгоритмы для решения задач на графах. Основной принцип отбора и организации материала этих глав состоял в том, что каждый рассматриваемый пример должен нести определенную идейную нагрузку, знакомить слушателя с одним из важных изобретений или открытий в алгоритмической области. При этом предпочтение отдавалось не самым последним или рекордным алгоритмам, а самым простым для понимания и убедительно демонстрирующим ту или иную идею. Для большинства рассматриваемых алгоритмов даются доказательства их правильности (т.е. того, что алгоритм действительно решает поставленную задачу) и оценок трудоемкости. Умение достаточно строго обосновывать алгоритмы и оценивать их трудоемкость является существенной частью квалификации алгоритмиста. Материал первой части может быть использован и в общем курсе дискретной математики. Каждая из глав этой части содержит задания для самостоятельной работы – математические задачи или упражнения, в которых предлагается разработать какой-нибудь алгоритм.

Во второй части книги рассматриваются классические модели вычислений, которые сыграли основную роль в формировании математического понятия алгоритма. Дается описание машин Тьюринга, алгорифмов Маркова, «машины абак» и как наиболее реалистичной модели вычислительного автомата – модели с адресуемой памятью RAM. Приводятся основные сведения о формальных языках и способах их конструктивного задания, а также теоретические основы логического программирования. Важность этих вопросов вытекает не только из общенаучных проблем развития математики, но также из практических задач общества, использующего вычислительную технику в производстве, экономике, инженерных расчетах и нуждающегося в адекватном представлении о возможностях вычислительных автоматов.

В третьей части рассматриваются способы структурирования информации в моделях с адресуемой памятью. Одной из основных целей при разработке структур данных является формирование математических понятий, которые пока не входят в классическую математику, но требуют формального описания и математического анализа их свойств. Основной интерес здесь представляют сложностные аспекты выполнения типичных операций. Возникновение наиболее удачных структур, использующихся в различных алгоритмах, приводит к формированию так называемых абстрактных типов данных, которые позволяют вести проектирование нетривиальных алгоритмов на более высоком уровне, не упуская из виду конкретных реализаций. Методы реализации абстрактных типов данных можно рассматривать как переход от описания алгоритма с использованием прикладных или математических понятий к описанию в конкретной системе вычислений. В нашей книге рассматриваются методы реализации приоритетных очередей, динамически меняющихся отношений эквивалентности, а также некоторые способы организации словарей, основывающиеся на использовании так называемых поисковых деревьев, приводятся примеры использования рассматриваемых структур в алгоритмах решения некоторых задач из теории графов.

В этой книге не принят какой-либо стандартный способ для описания алгоритмов. Каждое такое описание имеет целью продемонстрировать алгоритм в целом и взаимодействие его частей, не вдаваясь в излишнюю детализацию, но и не теряя ничего существенного. Иногда даются пошаговые описания разной степени подробности, иногда тексты на «псевдоязыке», использующем (не слишком последовательно) элементы синтаксиса языка Pascal и математическую символику.

Часть 1 написана В.Е. Алексеевым, части 2 и 3 – В.А. Талановым.

Часть 1. ГРАФЫ И АЛГОРИТМЫ

Глава 1. Элементы теории графов

Графы являются существенным элементом математических моделей в самых разнообразных областях науки и практики. Они помогают наглядно представить взаимоотношения между объектами или событиями в сложных системах. Многие алгоритмические задачи дискретной математики могут быть сформулированы как задачи, так или иначе связанные с графами, например задачи, в которых требуется выяснить какие-либо особенности устройства графа или найти в графе часть, удовлетворяющую некоторым требованиям, или построить граф с заданными свойствами.

Настоящая глава является кратким введением в теорию графов. В ней приводится минимум понятий, необходимый для того, чтобы можно было начать какую-либо содержательную работу с графами или приступить к более глубокому изучению теории графов. Доказательства даются только в тех случаях, когда их сложность не превышает некоторого интуитивно ощущаемого порога. Поэтому, например, такие важные факты, как теорема Кирхгофа или теорема Понтрягина – Куратовского, сообщаются без доказательств.

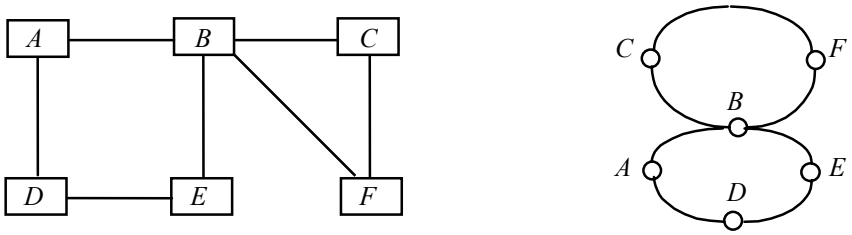
1.1. Начальные понятия

1.1.1. Определение графа

Для описания строения различных систем, состоящих из связанных между собой элементов, часто используют графические схемы, изображая элементы точками (кружками, прямоугольниками и т.д.), а связи между ними – линиями или стрелками, соединяющими элементы. При этом получаются диаграммы вроде тех, что показаны на рис. 1.1.

На таких диаграммах часто ни способ изображения элементов, ни форма или длина линий не имеют значения – важно лишь, какие именно пары элементов соединены линиями. Если посмотреть внимательно, то можно заметить, что рис. 1.1,*а* и 1.1,*б* изображают одну и ту же структуру связей между элементами A, B, C, D, E, F. Эту же структуру можно описать, не прибегая к графическому изображению, а просто перечислив пары связанных между собой элементов: (A, B), (A, D), (B, C), (B, E), (B, F),

(*C*, *F*), (*D*, *E*). Таким образом, когда мы отвлекаемся от всех несущественных подробностей, у нас остаются два списка: список элементов и список пар элементов. Вместе они составляют то, что математики называют графом. Из этого примера видно, что понятие графа само по себе не связано прямо с геометрией или графикой. Тем не менее возможность нарисовать граф – одна из привлекательных черт этого математического объекта.



а) *б*)

Рис. 1.1

Термин «граф» неоднозначен, это легко обнаружить, сравнивая приводимые в разных книгах определения графа. Однако во всех этих определениях есть и общее. В любом случае *граф* состоит из двух множеств – множества *вершин* и множества *ребер*, причем для каждого ребра указана пара вершин, которые это ребро *соединяет*. Вершины и ребра называются *элементами* графа. Здесь будут рассматриваться только *конечные* графы, то есть такие, у которых оба множества конечны. Чтобы получить законченное определение графа того или иного типа, необходимо уточнить еще следующие три момента.

1. Ориентированный или неориентированный? Прежде всего, нужно договориться, считаем ли мы пары (*a*, *b*) и (*b*, *a*) различными. Если да, то говорят, что рассматриваются упорядоченные пары (порядок элементов в паре важен), если нет – неупорядоченные. Если ребро *e* соединяет вершину *a* с вершиной *b* и пара (*a*, *b*) считается упорядоченной, то это ребро называется *ориентированным*, вершина *a* – его *началом*, вершина *b* – *концом*. Если же эта пара считается неупорядоченной, то ребро называется *неориентированным*, а обе вершины – его *концами*. Чаще всего рассматривают графы, в которых все ребра имеют один тип – либо ориентированные, либо неориентированные. В соответствии с этим и весь граф называют ориентированным или неориентированным. На рисунках ориентацию ребра (направление от начала к концу) указывают стрелкой. На рис.1.1 показаны неориентированные графы, а на рис. 1.2 – ориентированные.

2. Кратные ребра. Следующий пункт, требующий уточнения, – могут ли разные ребра иметь одинаковые начала и концы? Если да, то говорят, что в графе допускаются *кратные ребра*. Граф с кратными ребрами называют также *мультиграфом*. На рис. 1.2 изображены два графа, левый является ориентированным мультиграфом, а правый – ориентированным графом без кратных ребер.

3. Петли. Ребро, которому поставлена в соответствие пара вида (a, a), то есть ребро, соединяющее вершину a с нею же самой, называется *петлей*. Если такие ребра не допускаются, то говорят, что рассматриваются *графы без петель*.

Рис. 1.2

Комбинируя эти три признака, можно получить разные варианты определения понятия графа. Особенно часто встречаются неориентированные графы без петель и кратных ребер. Такие графы называют *обыкновенными*. Если в графе нет кратных ребер, то можно просто отождествить ребра с соответствующими парами вершин – считать, что ребро это и есть пара вершин. Чтобы исключить петли, достаточно оговорить, что вершины, образующие ребро, должны быть различны. Это приводит к следующему определению обыкновенного графа.

Определение. *Обыкновенным графом* называется пара $G = (V, E)$, где V – конечное множество, E – множество неупорядоченных пар различных элементов из V. Элементы множества V называются *вершинами* графа, элементы множества E – его *ребрами*.

Слегка модифицируя это определение, можно получить определения других типов графов без кратных ребер: если заменить слово «неупорядоченных» словом «упорядоченных», получится определение ориентированного графа без петель, если убрать слово «различных», получится определение графа с петлями. Ориентированный граф часто называют *орграфом*.

В дальнейшем термин «граф» будем употреблять в смысле «обыкновенный граф», а рассматривая другие типы графов, будем специально это оговаривать.

Множество вершин графа G будем обозначать через VG, множество ребер – EG, число вершин – $n(G)$, число ребер – $m(G)$.

Из определения видно, что для задания обыкновенного графа достаточно перечислить его вершины и ребра, причем каждое ребро должно быть парой вершин. Положим, например, $VG = \{a, b, c, d, e, f\}$, $EG = \{(a,c), (a,f), (b,c), (c,d), (d,f)\}$. Тем самым задан граф G с $n(G) = 6$, $m(G) = 5$. Если граф не слишком велик, то более наглядным способом представить его является рисунок, на котором вершины изображаются кружками или иными значками, а ребра – линиями, соединяющими вершины. Заданный выше граф G показан на рис. 1.3. Мы будем часто пользоваться именно этим способом представления графа, при этом обозначения вершин иногда будут помещаться внутри кружков, изображающих вершины, иногда рядом с ними, а иногда, когда имена вершин несущественны, и вовсе опускаться.

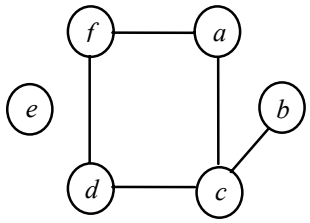


Рис. 1.3

1.1.2. Графы и бинарные отношения

Напомним, что *бинарным отношением* на множестве A называется любое подмножество R множества A^2, состоящего из всевозможных упорядоченных пар элементов множества A. Каждому такому отношению можно поставить в соответствие *граф отношения* $G = (A, R)$. Сравнивая с тем, что говорилось выше об определениях различных типов графов, видим, что понятие бинарного отношения эквивалентно понятию ориентированного графа с петлями. Другие типы графов без кратных ребер – это частные виды бинарных отношений. Отношение R называется *рефлексивным*, если для любого $x \in A$ пара (x, x) принадлежит R, и *антирефлексивным*, если ни одна такая пара не принадлежит R. Отношение называется *симметричным*, если из $(x, y) \in R$ следует, что $(y, x) \in R$. В графе антирефлексивного и симметричного отношения нет петель и для каждой пары вершин либо нет ни одного, либо есть два ребра, соединяющих эти вершины. Если в таком графе каждую пару ориентированных ребер, со-

единяющих одни и те же две вершины, заменить одним неориентированным ребром, то получится обыкновенный граф.

1.1.3. Откуда берутся графы

Легко найти примеры графов в самых разных областях науки и практики. Сеть дорог, трубопроводов, электрическая цепь, структурная формула химического соединения, блок-схема программы – в этих случаях графы возникают очень естественно и видны «невооруженным глазом». При желании графы можно обнаружить практически где угодно. Яркая демонстрация этого содержится в книге Д. Кнута [D.E. Knuth «The Stanford GraphBase»] – графы извлекаются из романа «Анна Каренина», из картины Леонардо да Винчи, из материалов Бюро экономического анализа США и из других источников.

Немало поводов для появления графов и в самой математике. Наиболее очевидный пример – любой многогранник в трехмерном пространстве. Вершины и ребра многогранника можно рассматривать как вершины и ребра графа. При этом мы отвлекаемся от того, как расположены элементы многогранника в пространстве, оставляя лишь информацию о том, какие вершины соединены ребрами. На рис. 1.4 показаны три способа изобразить один и тот же граф трехмерного куба.

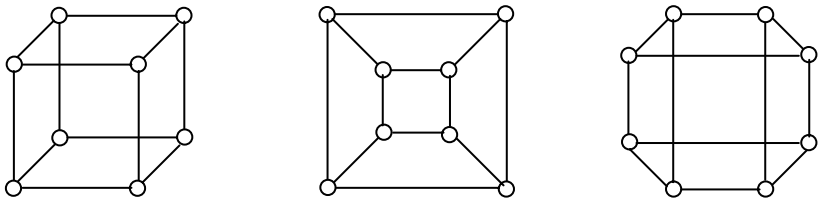


Рис. 1.4

Еще один способ образования графов из геометрических объектов иллюстрирует рис. 1.5.

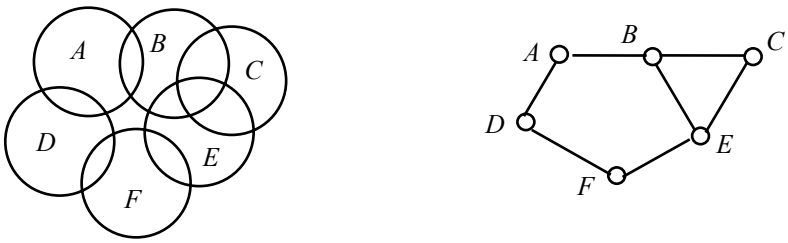


Рис. 1.5

Слева показаны шесть кругов на плоскости, а справа – граф, в котором каждая вершина соответствует одному из этих кругов и две вершины соединены ребром в том и только том случае, когда соответствующие круги пересекаются. Такие графы называют графами пересечений. Можно построить граф пересечений семейства интервалов на прямой или дуг окружности, или параллелепипедов. Вообще, для любого семейства множеств $\{S_1, \ldots, S_n\}$ можно построить граф пересечений с множеством вершин $\{1, \ldots, n\}$, в котором ребро (i, j) имеется тогда и только тогда, когда $i \neq j$ и $S_i \cap S_j \neq \varnothing$. Известно, что любой граф можно представить как граф пересечений некоторого семейства множеств.

1.1.4. Число графов

Возьмем какое-нибудь множество V, состоящее из n элементов, и будем рассматривать всевозможные (обыкновенные!) графы с множеством вершин V. Обозначим число таких графов через g_n. Эти графы различаются только множествами ребер, а каждое ребро – это неупорядоченная пара различных элементов из V. В комбинаторике такие пары называются сочетаниями из n по 2, их число равно

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Каждая пара может быть включена или не включена в множество ребер графа. Применяя правило произведения, приходим к следующему результату.

Теорема 1.1. $g_n = 2^{n(n-1)/2}$.

1.1.5. Смежность, инцидентность, степени

Если в графе имеется ребро $e = (a, b)$, то говорят, что вершины a и b *смежны* в этом графе, ребро e *инцидентно* каждой из вершин a, b, а каждая из них *инцидентна* этому ребру.

Множество всех вершин графа, смежных с данной вершиной a, называется *окрестностью* этой вершины и обозначается через $V(a)$.

На практике удобным и эффективным при решении многих задач способом задания графа являются так называемые списки смежности. Эти списки могут быть реализованы различными способами в виде конкретных структур данных, но в любом случае речь идет о том, что для каждой вершины a перечисляются все смежные с ней вершины, то есть элементы множества $V(a)$. Такой способ задания дает возможность быстрого просмотра окрестности вершины.

Число вершин, смежных с вершиной a, называется *степенью* вершины a и обозначается через $\deg(a)$.

Если сложить степени всех вершин некоторого графа, то каждое ребро внесет в эту сумму вклад, равный 2, поэтому справедливо следующее утверждение.

Теорема 1.2. $\sum_{a \in VG} \deg(a) = 2m(G)$.

Это равенство известно как «лемма о рукопожатиях». Из него следует, что число вершин нечетной степени в любом графе четно.

Вершину степени 0 называют *изолированной*.

Граф называют *регулярным* степени d, если степень каждой его вершины равна d.

Набор степеней графа – это последовательность степеней его вершин, выписанных в неубывающем порядке.

1.1.6. Некоторые специальные графы

Рассмотрим некоторые особенно часто встречающиеся графы.

Пустой граф – граф, не содержащий ни одного ребра. Пустой граф с множеством вершин $\{1, 2, ..., n\}$ обозначается через O_n.

Полный граф – граф, в котором каждые две вершины смежны. Полный граф с множеством вершин $\{1, 2, ..., n\}$ обозначается через K_n. Граф K_1, в частности, имеет одну вершину и ни одного ребра. Очевидно, $K_1 = O_1$. Будем считать также, что существует граф K_0, у которого $VG = EG = \varnothing$.

Цепь (*путь*) P_n – граф с множеством вершин $\{1, 2, ..., n\}$ и множеством ребер $\{(1, 2), (2, 3), \ldots, (n-1, n)\}$.

Цикл C_n – граф, который получается из графа P_n добавлением ребра $(1, n)$.

Все эти графы при $n = 4$ показаны на рис. 1.6.

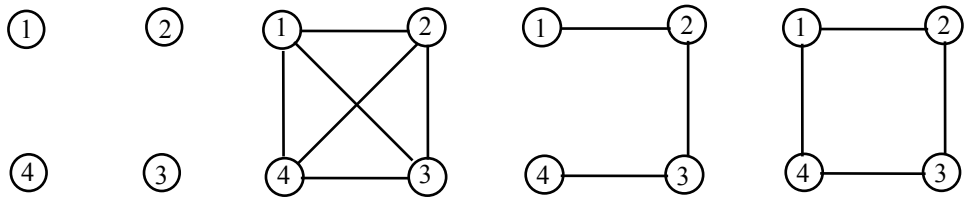


Рис. 1.6

1.1.7. Графы и матрицы

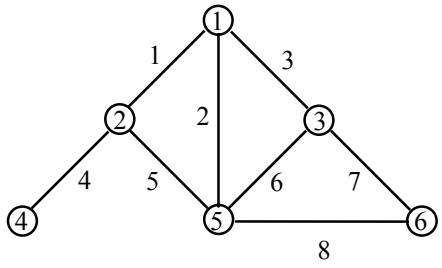
Пусть G – граф с n вершинами, причем $VG = \{1, 2, \ldots, n\}$. Построим квадратную матрицу A порядка n, в которой элемент A_{ij}, стоящий на пере-

сечении строки с номером i и столбца с номером j, определяется следующим образом:

$$A_{ij} = \begin{cases} 1, \text{ если } (i, j) \in EG, \\ 0, \text{ если } (i, j) \notin EG. \end{cases}$$

Она называется *матрицей смежности* графа. Матрицу смежности можно построить и для ориентированного графа, и для неориентированного, и для графа с петлями. Для обыкновенного графа она обладает двумя особенностями: из-за отсутствия петель на главной диагонали стоят нули, а так как граф неориентированный, то матрица симметрична относительно главной диагонали. Обратно, каждой квадратной матрице порядка n, составленной из нулей и единиц и обладающей двумя указанными свойствами, соответствует обыкновенный граф с множеством вершин $\{1, 2, \ldots, n\}$.

Другая матрица, ассоциированная с графом – это *матрица инцидентности*. Для ее построения пронумеруем вершины графа числами от 1 до n, а ребра – числами от 1 до m. Матрица инцидентности I имеет n строк и m столбцов, а ее элемент I_{ij} равен 1, если вершина с номером i инцидентна ребру с номером j, в противном случае он равен нулю. На рис. 1.7 показан граф с пронумерованными вершинами и ребрами и его матрицы смежности и инцидентности.



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}, \quad I = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

Рис. 1.7

Для ориентированного графа матрица инцидентности определяется несколько иначе: ее элемент I_{ij} равен 1, если вершина i является началом ребра j, он равен –1, если она является концом этого ребра, и он равен 0, если эта вершина и это ребро не инцидентны друг другу.

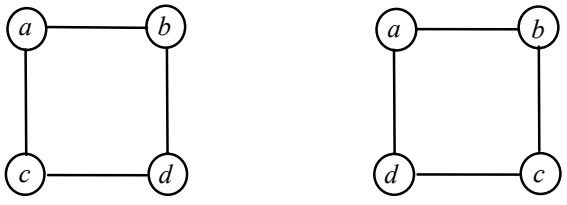
1.1.8. Взвешенные графы

Часто, особенно когда графы используются для моделирования реальных систем, их вершинам или ребрам, или и тем и другим приписываются некоторые числа. Природа этих чисел может быть самая разнообразная. Например, если граф представляет собой модель железнодорожной сети, то число, приписанное ребру, может указывать длину перегона между двумя станциями или наибольший вес состава, который допустим для этого участка пути, или среднее число поездов, проходящих через этот участок в течение суток и т.п. Что бы ни означали эти числа, сложилась традиция называть их *весами*, а граф с заданными весами вершин и/или ребер – *взвешенным графом*.

1.2. Изоморфизм

1.2.1. Определение изоморфизма

На рис. 1.8 изображены два графа с одним и тем же множеством вершин $\{a, b, c, d\}$. При внимательном рассмотрении можно обнаружить, что это разные графы – в левом имеется ребро (a, c), в правом же такого нет. В то же время, если не обращать внимания на наименования вершин, то эти графы явно одинаково устроены: каждый из них – цикл из четырех вершин. Во многих случаях при исследовании строения графов имена или номера вершин не играют роли, и такие графы, один из которых получается из другого переименованием вершин, удобнее было бы считать одинаковыми. Для того чтобы это можно было делать «на законном основании», вводится понятие изоморфизма графов.



G_1 G_2

Рис. 1.8

Определение. Графы G_1 и G_2 называются изоморфными, если суще-

ствует такая биекция f множества VG_1 на множество VG_2, что $(a, b) \in EG_1$ тогда и только тогда, когда $(f(a), f(b)) \in EG_2$. Отображение f в этом случае называется изоморфизмом графа G_1 на граф G_2.

Тот факт, что графы G_1 и G_2 изоморфны, записывается так: $G_1 \cong G_2$. Для графов, изображенных на рис. 1.8, изоморфизмом является, например, отображение, задаваемое таблицей:

x (вершина графа G_1)	a	b	c	d
$f(x)$ (вершина графа G_2)	a	b	d	c

Заметим, что в этом примере есть и другие изоморфизмы первого графа на второй.

Сформулированное определение изоморфизма годится и для ориентированных графов, нужно только обе упоминаемые в нем пары вершин считать упорядоченными.

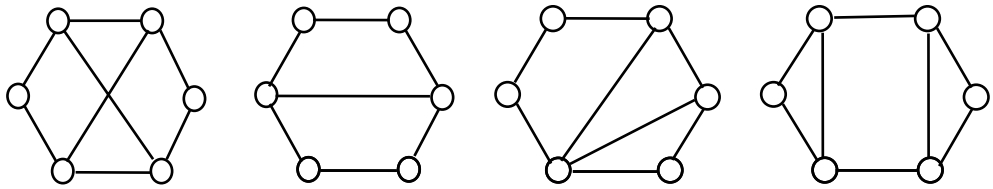
Изоморфизм – бинарное отношение на множестве графов. Очевидно, это отношение рефлексивно, симметрично и транзитивно, то есть является отношением эквивалентности. Классы эквивалентности называются абстрактными графами. Когда говорят, что рассматриваются абстрактные графы, это означает, что изоморфные графы считаются одинаковыми. Абстрактный граф можно представлять себе как граф, у которого стерты имена (пометки) вершин, поэтому абстрактные графы иногда называют также непомеченными.

1.2.2. Инварианты

В общем случае узнать, изоморфны ли два графа, достаточно сложно. Если буквально следовать определению, то нужно перебрать все биекции множества вершин одного из них на множество вершин другого и для каждой из этих биекций проверить, является ли она изоморфизмом. Для n вершин имеется $n!$ биекций и эта работа становится практически невыполнимой уже для не очень больших n (например, $20! > 2 \cdot 10^{18}$). Однако во многих случаях бывает довольно легко установить, что два данных графа неизоморфны. Рассмотрим, например, графы, изображенные на рис. 1.9.

Так как при изоморфизме пара смежных вершин переходит в пару смежных, а пара несмежных – в пару несмежных, то ясно, что число ребер у двух изоморфных графов должно быть одинаковым. Поэтому сразу можно сказать, что графы G_1 и G_2, у которых разное количество ребер, неизоморфны. У графов G_1 и G_3 одинаковое число ребер, но они тоже неизоморфны. Это можно установить, сравнивая степени вершин. Очевидно,

при изоморфизме каждая вершина переходит в вершину той же степени. Следовательно, изоморфные графы должны иметь одинаковые наборы степеней, а у графов G_1 и G_3 эти наборы различны. С графами G_1 и G_4 дело обстоит немного сложнее – у них и наборы степеней одинаковы. Все же и эти графы неизоморфны: можно заметить, что в графе G_4 есть цикл длины 3, а в графе G_1 таких циклов нет. Ясно, что при изоморфизме каждый цикл длины 3 переходит в цикл длины 3.



G_1 G_2 G_3 G_4

Рис. 1.9

Характеристики графов, которые сохраняются при изоморфизме, называются инвариантами. В этом примере мы видели некоторые простые инварианты – число ребер, набор степеней, число циклов заданной длины, в дальнейшем будут введены еще многие другие. Если удается установить, что для двух исследуемых графов некоторый инвариант принимает разные значения, то эти графы неизоморфны. Для того чтобы доказать, что два графа изоморфны, необходимо предъявить соответствующую биекцию.

1.3. Операции над графами

Для получения новых графов можно использовать разнообразные операции. Здесь мы рассмотрим два вида операций – *локальные*, при которых заменяются, удаляются или добавляются отдельные элементы графа, и *алгебраические*, когда новый граф строится по определенным правилам из нескольких имеющихся.

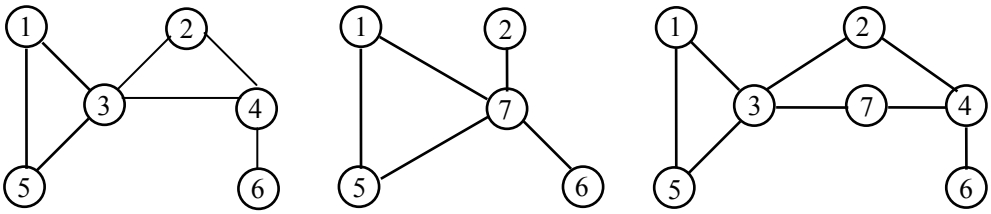
1.3.1. Локальные операции

Простейшая операция – *удаление ребра*. При удалении ребра сохраняются все вершины графа и все его ребра, кроме удаляемого. Обратная операция – *добавление ребра*.

При *удалении вершины* вместе с вершиной удаляются и все инцидентные ей ребра. Граф, получаемый из графа G удалением вершины a, обозначают $G - a$. При *добавлении вершины* к графу добавляется новая изолированная вершина. С помощью операций добавления вершин и ребер можно «из ничего», то есть из графа K_0, построить любой граф.

Операция *стягивания ребра* (a, b) определяется следующим образом. Вершины a и b удаляются из графа, к нему добавляется новая вершина c и она соединяется ребром с каждой вершиной, с которой была смежна хотя бы одна из вершин a, b.

Операция *подразбиения ребра* (a, b) действует следующим образом. Из графа удаляется это ребро, к нему добавляется новая вершина c и два новых ребра (a, c) и (b, c). На рис. 1.10 изображены исходный граф G, граф G', полученный из него стягиванием ребра (3, 4) и G'', полученный подразбиением того же ребра. В обоих случаях вновь добавленная вершина обозначена цифрой 7.

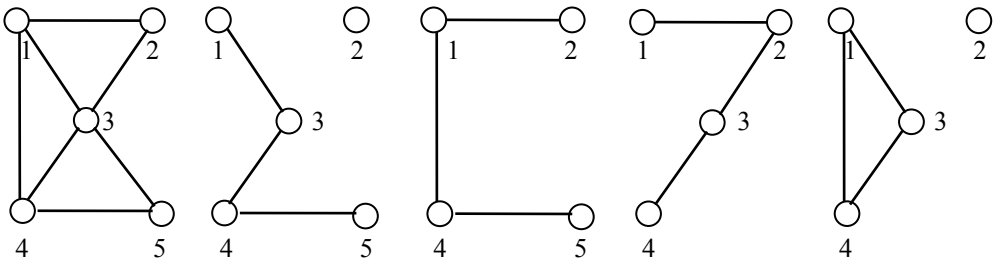


G G' G''

Рис. 1.10

1.3.2. Подграфы

Граф G' называется *подграфом* графа G, если $VG' \subseteq VG$, $EG' \subseteq EG$. Всякий подграф может быть получен из графа удалением некоторых вершин и ребер. На рис. 1.11 изображены граф G и его подграфы G_1, G_2, G_3, G_4.



G G_1 G_2 G_3 G_4

Рис. 1.11

Подграф G' графа G называется *остовным*, если $VG' = VG$. Остовный подграф может быть получен из графа удалением некоторых ребер, вершины же остаются в неприкосновенности. На рис. 1.11 G_1 – остовный подграф графа G, а G_2, G_3 и G_4 не являются остовными подграфами.

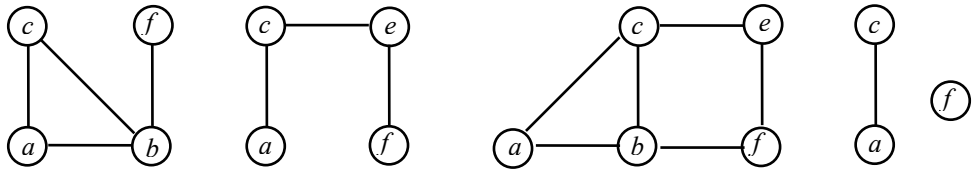
Другая важная разновидность подграфов – *порожденные* подграфы. Пусть задан граф $G = (V, E)$ и в нем выбрано множество вершин $U \subseteq V$. Рассмотрим подграф $G' = (U, E')$, где E' состоит из всех тех ребер графа G, у которых оба конца принадлежат U. Говорят, что этот подграф *порожден множеством вершин* U. Он обозначается через $G\langle U \rangle$. Порожденный подграф может быть получен из графа удалением «лишних» вершин, то есть вершин, не принадлежащих U.

Можно определить также подграф, *порожденный множеством ребер* $F \subseteq E$. Это подграф $G' = (V', F)$, где V' состоит из всех вершин, инцидентных ребрам из F.

На рис. 1.11 G_2 – подграф графа G, порожденный множеством вершин $\{1, 2, 4, 5\}$, то есть $G_2 = G\langle\{1, 2, 4, 5\}\rangle$, он же порождается множеством ребер $\{(1, 2), (1, 4), (4, 5)\}$; подграф G_3 не порождается множеством вершин, но порождается множеством ребер $\{(1, 2), (2, 3), (3, 4)\}$; подграф G_4 не является ни остовным, ни порожденным в каком-либо смысле.

1.3.3. Алгебраические операции

Поскольку граф состоит из двух множеств (вершины и ребра), то различные операции над множествами естественным образом порождают соответствующие операции над графами. Например, *объединение* двух графов G_1 и G_2 определяется как граф $G = G_1 \cup G_2$, у которого $VG = VG_1 \cup VG_2$, $EG = EG_1 \cup EG_2$, а *пересечение* – как граф $G = G_1 \cap G_2$, у которого $VG = VG_1 \cap VG_2$, $EG = EG_1 \cap EG_2$. Обе операции иллюстрирует рис. 1.12.



G_1 G_2 $G_1 \cup G_2$ $G_1 \cap G_2$

Рис. 1.12

Дополнением (*дополнительным графом*) к графу $G = (V, E)$ называется граф $\overline{G}$, у которого множество вершин то же, что у G, а множество ребер является дополнением множества E до множества всех неупорядоченных пар вершин. Иначе говоря, две различные вершины смежны в графе $\overline{G}$ тогда и только тогда, когда они несмежны в графе G. Например, $\overline{O}_n = K_n$.

Другой пример показан на рис. 1.13. Очевидно, всегда $\overline{\overline{G}} = G$.

G $\qquad\qquad$ $\overline{G}$

Рис. 1.13

Под *суммой* $G_1 + G_2$ двух абстрактных графов понимают объединение графов с непересекающимися множествами вершин. Точнее говоря, имеется в виду следующее. Сначала вершинам графов-слагаемых присваиваются имена (пометки, номера) так, чтобы множества вершин не пересекались, затем полученные графы объединяются. Операция сложения ассоциативна, то есть $(G_1 + G_2) + G_3 = G_1 + (G_2 + G_3)$ для любых трех графов. Поэтому можно образовывать сумму любого числа графов, не указывая порядка действий с помощью скобок. Если складываются k экземпляров одного и того же графа G, то полученный граф обозначается через kG. Например, $O_n \cong nK_1$. На рис. 1.14 изображен граф $C_4 + 2K_2 + 4K_1$.

Соединением двух графов G_1 и G_2 называется граф, получаемый из их суммы добавлением всех ребер, соединяющих вершины первого слагаемого с вершинами второго. Будем записывать эту операцию как $G_1 \circ G_2$. На рис. 1.15 представлен граф $P_3 \circ O_2$.

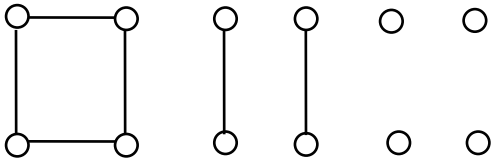


Рис. 1.14

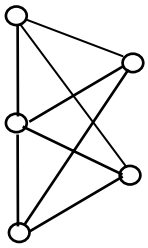
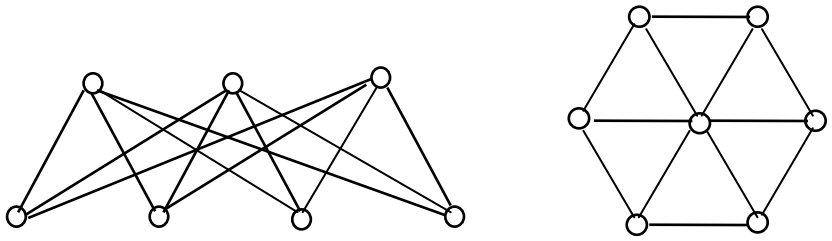


Рис. 1.15

Легко видеть, что операции сложения и соединения графов связаны друг с другом следующими простыми соотношениями:

$$\overline{G_1 + G_2} = \overline{G_1} \circ \overline{G_2}, \quad \overline{G_1 \circ G_2} = \overline{G_1} + \overline{G_2}.$$

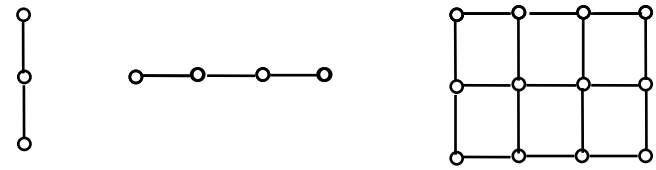
Введем еще два типа специальных графов, которые легко описываются с помощью операции соединения. Первый – *полный двудольный граф* $K_{p,q} = O_p \circ O_q$. В этом графе множество вершин разбито на два подмножества (доли), в одном из которых p вершин, в другом q, и две вершины в нем смежны тогда и только тогда, когда они принадлежат разным подмножествам. Второй – колесо $W_n = C_n \circ K_1$. На рис. 1.16 показаны графы $K_{3,4}$ и W_6.



$K_{3,4}$ W_6

Рис. 1.16

Произведение $G = G_1 \times G_2$ графов G_1 и G_2 определяется следующим образом. Множеством вершин графа G является декартово произведение множеств VG_1 и VG_2, то есть вершины этого графа – упорядоченные пары (x, y), где x – вершина первого сомножителя, y – вершина второго. Вершины (x_1, y_1) и (x_2, y_2) в G смежны тогда и только тогда, когда $x_1 = y_1$ и y_1 смежна с y_2 в графе G_2 или $y_1 = y_2$ и x_1 смежна с x_2 в графе G_1. С помощью операции произведения можно выразить некоторые важные графы через простейшие. Например, произведение двух цепей дает *прямоугольную решетку* – см. рис. 1.17. Если один из сомножителей превратить в цикл, добавив одно ребро, то прямоугольная решетка превратится в *цилиндрическую*, а если и второй сомножитель превратить в цикл, то получится *тороидальная решетка*.



P_3 P_4 $P_3 \times P_4$

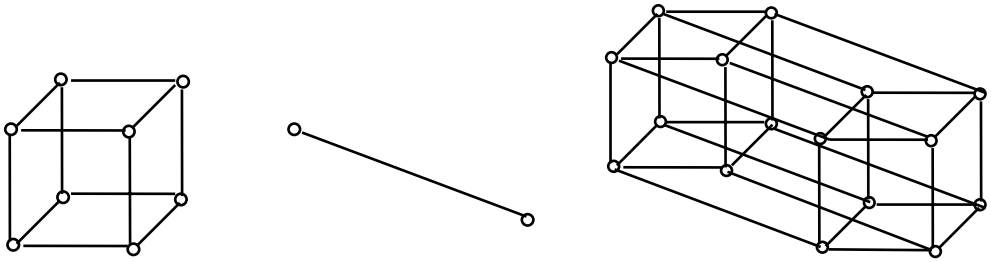
Рис. 1.17

Другой пример – k-мерный куб Q_k, описываемый в задаче 3. С помо-

щью операции произведения его (точнее, изоморфный ему граф) можно определить рекурсивно:

$$Q_1 = K_2, \quad Q_k = Q_{k-1} \times K_2.$$

На рис. 1.18 показано, как получается Q_4 из Q_3.



Q_3 K_2 $Q_4 = Q_3 \times K_2$

Рис. 1.18

1.4. Маршруты, связность, расстояния

1.4.1. Маршруты, пути, циклы

Маршрут в графе – это последовательность вершин x_1, x_2, …, x_n, такая, что для каждого $i = 1, 2, .., n - 1$ вершины x_i и x_{i+1} соединены ребром. Эти $n - 1$ ребер называются *ребрами маршрута*, говорят, что маршрут *проходит* через них, а число $n - 1$ называют *длиной* маршрута. Говорят, что маршрут *соединяет* вершины x_1 и x_n, они называются соответственно *началом* и *концом* маршрута, вершины x_2, …, x_{n-1} называются *промежуточными*. Маршрут называется *замкнутым*, если $x_1 = x_n$.

Путь – это маршрут, в котором все ребра различны. Путь называется *простым*, если и все вершины в нем различны.

Цикл – это замкнутый путь. Цикл x_1, x_2, …, x_{n-1}, x_1 называется *простым*, если вершины x_1, x_2, …, x_{n-1} все попарно различны.

В графе на рис. 1.19 последовательность вершин

2, 3, 5, 4 – не маршрут;

2, 3, 4, 5, 1, 4, 3 – маршрут, но не путь;

3, 1, 4, 5, 1, 2 – путь, но не простой;

2, 3, 1, 4, 3, 1, 2 – замкнутый маршрут, но не цикл;

2, 3, 1, 4, 5, 1, 2 – цикл, но не простой;

2, 3, 4, 5, 1, 2 – простой цикл.

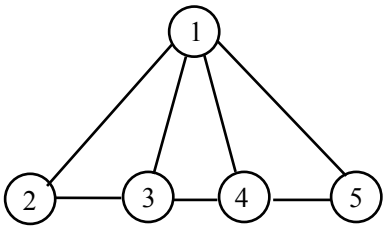


Рис. 1.19

Установим некоторые простые свойства маршрутов.

Лемма 1.3. *В любом маршруте, соединяющем две различные вершины, содержится простой путь, соединяющий те же вершины. В любом цикле, проходящем через некоторое ребро, содержится простой цикл, проходящий через это ребро.*

Доказательство. Пусть $x_1, x_2, \ldots, x_n$ – маршрут. Если все его вершины различны, то это уже простой путь. В противном случае, пусть $x_i = x_j$, $i < j$. Тогда последовательность $x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, x_n$, полученная из этого маршрута удалением отрезка последовательности от x_{i+1} до x_j, тоже является маршрутом. Новый маршрут соединяет те же вершины и имеет меньшую длину. Продолжая действовать таким образом, после конечного числа «спрямлений» получим простой путь, соединяющий x_1 и x_n. Второе утверждение доказывается аналогично. □

Отметим, что в формулировке леммы 1.3 нельзя заменить слово «цикл» словами «замкнутый маршрут». Действительно, если (a, b) – ребро графа, то последовательность a, b, a – замкнутый маршрут, проходящий через это ребро, но никакого цикла в нем нет.

Лемма 1.4. *Если в графе степень каждой вершины не меньше* 2, *то в нем есть цикл.*

Доказательство. Найдем в графе простой путь наибольшей длины. Пусть это $x_1, x_2, \ldots, x_n$. Вершина x_n смежна с x_{n-1}, а так как ее степень не меньше 2, то она смежна еще хотя бы с одной вершиной, скажем, y. Если y была бы отлична от всех вершин пути, то последовательность $x_1, x_2, \ldots, x_n, y$ была бы простым путем большей длины. Следовательно, y – это одна из вершин пути, $y = x_i$, причем $i < n - 1$. Но тогда $x_i, x_{i+1}, \ldots, x_n, x_i$ – цикл. □

1.4.2. Связность и компоненты

Граф называется *связным*, если в нем для любых двух вершин имеется маршрут, соединяющий эти вершины. Заметим, что ввиду леммы 1.3 можно в этом определении заменить слово «маршрут» словами «простой путь».

Для произвольного графа определим на множестве вершин *отношение соединимости*: вершина *a* соединима с вершиной *b*, если существует соединяющий их маршрут. Легко видеть, что это отношение рефлексивно, симметрично и транзитивно, то есть является отношением эквивалентности. Классы эквивалентности называются *областями связности*, а порождаемые ими подграфы – *компонентами связности* графа. В связном графе имеется только одна компонента связности – весь граф. Компоненты связности можно определить также как максимальные по включению связные подграфы данного графа.

У графа на рис. 1.20 имеются четыре области связности – {1, 2, 9}, {3, 10, 11}, {4}, {5, 6, 7, 8, 12, 13, 14, 15}.

Рис. 1.20

Вершина называется *шарниром* (или *точкой сочленения*), если при ее удалении число компонент связности увеличивается. У графа на рис. 1.20 имеется четыре шарнира – это вершины 3, 6, 7, 8.

Ребро, при удалении которого увеличивается число компонент связности, называется *перешейком*. Перешейками графа, изображенного на рис. 1.20, являются ребра (3, 10), (3, 11), (6, 7), (7, 8), (7, 13).

Легко доказываются следующие свойства шарниров и перешейков.

Лемма 1.5. *Вершина a является шарниром тогда и только тогда, когда в графе имеются такие отличные от a вершины b и c, что любой путь, соединяющий b и c, проходит через a.*

Лемма 1.6. *Ребро является перешейком в том и только том случае, если в графе нет простого цикла, содержащего это ребро.*

1.4.3. Метрические характеристики графов

Расстоянием между двумя вершинами графа называется длина кратчайшего пути, соединяющего эти вершины. Расстояние между вершинами *a* и *b* обозначается через $d(a, b)$. Если в графе нет пути, соединяющего *a* и *b*, то есть эти вершины принадлежат разным компонентам связности, то расстояние между ними считается бесконечным.

Легко видеть, что функция $d(x, y)$ обладает свойствами:

1) $d(x, y) \geq 0$, причем $d(x, y) = 0$ тогда и только тогда, когда $x = y$;

2) $d(x, y) = d(y, x)$;

3) $d(x, y) + d(y, z) \geq d(x, z)$ (неравенство треугольника).

В математике функцию двух переменных, определенную на некотором множестве и удовлетворяющую условиям 1) – 3), называют *метрикой*, а множество, на котором задана метрика, – *метрическим пространством*. Таким образом, множество вершин любого графа можно рассматривать как метрическое пространство.

Расстояние от данной вершины a до наиболее удаленной от нее вершины называется *эксцентриситетом* вершины a и обозначается через $ecc(a)$. Таким образом,

$$ecc(a) = \max_{x \in VG} d(a, x) .$$

Вершину с наименьшим эксцентриситетом называют *центральной*, а вершину с наибольшим эксцентриситетом – *периферийной*. Множество всех центральных вершин называется *центром* графа. Сама величина наименьшего эксцентриситета называется *радиусом* графа и обозначается через $\mathrm{rad}(G)$, а величина наибольшего – *диаметром* и обозначается $\mathrm{diam}(G)$. Иначе говоря,

$$\mathrm{rad}(G) = \min_{x \in VG} \max_{y \in VG} d(x, y),$$

$$\mathrm{diam}(G) = \max_{x \in VG} \max_{y \in VG} d(x, y).$$

Наименьший диаметр имеет полный граф – его диаметр равен 1. Среди связных графов с n вершинами наибольший диаметр, равный $n - 1$, имеет цепь P_n.

Если расстояние между двумя вершинами равно диаметру графа, то кратчайший путь, соединяющий эти вершины, называется *диаметральным путем*, а подграф, образованный вершинами и ребрами этого пути, – *диаметральной цепью*.

Для графа, изображенного на рис. 1.21, эксцентриситеты вершин приведены в таблице:

x	1	2	3	4	5	6	7	8	9
$ecc(x)$	5	4	4	3	5	3	3	4	5

Центр этого графа составляют вершины 4, 6, 7; периферийные вершины – 1, 5 и 9; радиус его равен 3, а диаметр 5. Одна из диаметральных цепей порождается множеством вершин {1, 3, 6, 7, 8, 9}.

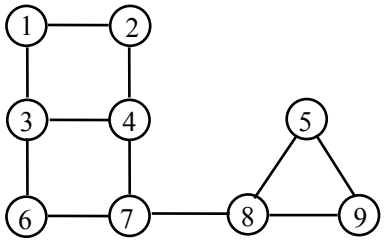


Рис. 1.21

1.4.4. Маршруты и связность в орграфах

Для ориентированного графа можно определить два типа маршрутов. *Неориентированный маршрут* (или просто *маршрут*) – это последовательность вершин $x_1, x_2, \ldots, x_n$, такая, что для каждого $i = 1, 2, \ldots, n - 1$ хотя бы одно из ребер (x_i, x_{i+1}), (x_{i+1}, x_i) принадлежит графу. Маршрут называется *ориентированным* (или *ормаршрутом*), если для каждого i пара (x_i, x_{i+1}) является ребром графа. Таким образом, при движении вдоль маршрута в орграфе ребра могут проходиться как в направлении ориентации, так и в обратном направлении, а при движении вдоль ормаршрута – только в направлении ориентации. Это различие очевидным образом распространяется на пути и циклы, так что в орграфе можно рассматривать пути и орпути, циклы и орциклы. Будем говорить, что маршрут $x_1, x_2, \ldots, x_n$ *соединяет* вершины x_1 и x_n, а ормаршрут $x_1, x_2, \ldots, x_n$ *ведет из* x_1 *в* x_n.

Соответственно двум типам маршрутов определяются и два типа связности орграфов. Орграф называется *связным* (или *слабо связным*), если для каждой пары вершин в нем имеется соединяющий их маршрут; он называется *сильно связным*, если для каждой упорядоченной пары вершин (a, b) в нем имеется ормаршрут, ведущий из a в b. Максимальные по включению подмножества вершин орграфа, порождающие сильно связные подграфы, называются его *областями сильной связности*, а порождаемые ими подграфы – *компонентами сильной связности*. Очевидно, разные области сильной связности не могут иметь общих вершин, так что множество вершин каждого орграфа разбивается на области сильной связности. Областями сильной связности орграфа на рис. 1.22 являются множества {1, 2, 5}, {3, 4, 6, 7, 8}, {9}.

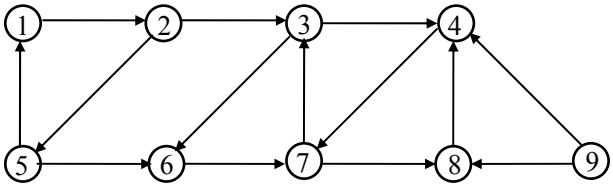


Рис. 1.22

1.5. Деревья

1.5.1. Определение и элементарные свойства

Деревом называется связный граф, не имеющий циклов. В графе без циклов, таким образом, каждая компонента связности является деревом. Такой граф называют *лесом*.

Из леммы 1.4 следует, что во всяком дереве, в котором не меньше двух вершин, имеется вершина степени 1. Такие вершины называют *висячими вершинами*, или *листьями*. В действительности легко доказать, что в каждом дереве не меньше двух листьев, а цепь P_n – пример дерева, в котором точно два листа.

В следующих двух теоремах устанавливаются некоторые свойства деревьев.

Теорема 1.7. *Граф с n вершинами и m ребрами является деревом тогда и только тогда, когда он удовлетворяет любым двум из следующих трех условий*:

(1) *связен*;

(2) *не имеет циклов*;

(3) $m = n - 1$.

Доказательство. Первые два условия вместе составляют определение дерева. Покажем, что выполнение любых двух из условий (1)–(3) влечет выполнение третьего.

(1) и (2) $\Rightarrow$ (3). Индукция по числу вершин. При $n = 1$ утверждение очевидно. При $n \geq 1$ в дереве имеется хотя бы один лист. Если из дерева удалить лист, то снова получится дерево, так как циклов не появится, а связность, очевидно, сохранится. В этом новом дереве $n - 1$ вершина и, по предположению индукции, $n - 2$ ребра. Следовательно, в исходном дереве было $n - 1$ ребро.

(2) и (3) $\Rightarrow$ (1). Пусть в графе, не имеющем циклов, $n - 1$ ребро, а его компонентами связности являются G_1, G_2, …, G_k, причем G_i состоит из n_i вершин, $i = 1, \ldots, k$. Каждая компонента является деревом, поэтому, как доказано выше, число ребер в G_i равно $n_i - 1$, а всего ребер в графе $\sum_{i=1}^{k}(n_i - 1) = n - k = n - 1$. Значит, $k = 1$ и граф связен.

(1) и (3) $\Rightarrow$ (2). Рассмотрим связный граф с $n - 1$ ребром. Если бы в нем был цикл, то, удалив любое цикловое ребро, получили бы связный граф с меньшим числом ребер. Мы можем продолжать такое удаление ребер до тех пор, пока не останется связный граф без циклов, то есть дерево. Но ребер в этом дереве было бы меньше, чем $n - 1$, а это противоречит

доказанному выше. □

Теорема 1.8. *Если G – дерево, то*

1) *в G любая пара вершин соединена единственным путем*;

2) *при добавлении к G любого нового ребра образуется цикл*;

3) *при удалении из G любого ребра он превращается в несвязный граф*.

Доказательство. Существование пути между любыми двумя вершинами следует из связности дерева. Допустим, что в некотором дереве существуют два различных пути, соединяющих вершины a и b. Начальные отрезки этих путей совпадают (оба пути начинаются в одной и той же вершине a). Пусть x – последняя вершина этого совпадающего начала, а после x в одном пути следует вершина y_1, а в другом – вершина y_2. Рассмотрим ребро (x, y_1). Если его удалить из графа, то в оставшемся подграфе вершины y_1 и x будут соединимыми – соединяющий их маршрут можно построить так: взять отрезок первого пути от y_1 до b и к нему присоединить отрезок второго от x до b, взятый в обратном порядке. Но это означает, что ребро (x, y_1) не является перешейком. Однако из леммы 1.6 следует, что в дереве каждое ребро является перешейком. Этим доказано утверждение 1). Если к дереву добавить новое ребро, то, поскольку вершины, соединяемые этим ребром, уже были соединены путем, образуется цикл. Утверждение 3) следует из леммы 1.6. □

Отметим, что единственный путь, соединяющий две вершины дерева, всегда простой (если путь не является простым, в нем обязательно содержится цикл).

1.5.2. Центр дерева

Центр графа может состоять из одной вершины (как, например, в графе $K_{1,q}$), а может включать все его вершины (полный граф). Для дерева, как мы увидим, имеется гораздо более ограниченный диапазон возможностей.

Теорема 1.9. *Центр дерева состоит из одной вершины или из двух смежных вершин.*

Доказательство. Допустим, что в некотором дереве имеются две несмежные центральные вершины c_1 и c_2. На пути, соединяющем эти вершины, найдем промежуточную вершину a с максимальным эксцентриситетом, и пусть b_1 и b_2 – вершины, соседние с a на этом пути (см. рис. 1.23).

Пусть x – вершина, наиболее удаленная от a в дереве, то есть $d(a, x) = ecc(a)$. Путь, соединяющий a с x, не может проходить через обе вершины b_1 и b_2. Допустим, он не проходит через b_1. Тогда единственный путь

из b_1 в x проходит через a и $d(b_1, x) > d(a, x)$. Отсюда следует, что $ecc(b_1) > ecc(a)$, а это противоречит выбору вершины a.

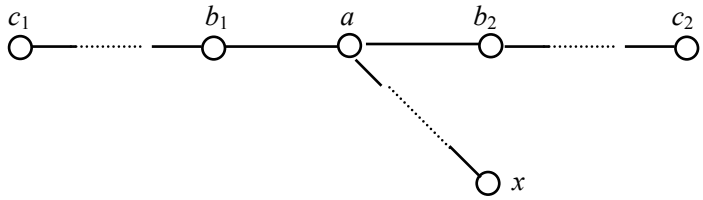


Рис. 1.23

Следовательно, любые две центральные вершины смежны, а так как в дереве не может быть трех попарно смежных вершин, то в нем не больше двух центральных вершин. □

1.5.3. Корневые деревья

Часто в дереве особо выделяется одна вершина, играющая роль своего рода «начала отсчета». Дерево с выделенной вершиной называют *корневым деревом*, а саму эту вершину – *корнем*. Из дерева с n вершинами можно, таким образом, образовать n различных корневых деревьев.

При графическом изображении корневого дерева обычно придерживаются какого-нибудь стандарта. Один из наиболее распространенных состоит в следующем. Возьмем на плоскости семейство параллельных прямых с равными расстояниями между соседними прямыми. Изобразим корень точкой на одной из этих прямых, смежные с корнем вершины – точками на соседней прямой, вершины, находящиеся на расстоянии 2 от корня, – на следующей, и т.д. Ребра изобразим отрезками прямых. Ясно, что вершины на каждой прямой можно разместить так, чтобы ребра не пересекались. Пример нарисованного таким образом корневого дерева показан на рис. 1.24 (корень обведен кружком). Чаще, впрочем, дерево рисуют корнем вверх, а не вниз.

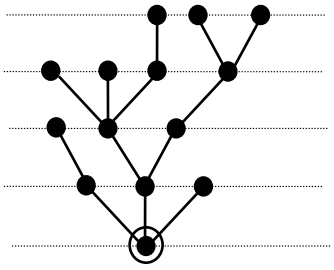


Рис. 1.24

Иногда бывает полезно ребра корневого дерева ориентировать так, чтобы в каждую вершину вел ориентированный путь из корня (для дерева на рис. 1.24 это означает, что каждое ребро ориентируется снизу вверх). Такое ориентированное корневое дерево будем называть *исходящим* деревом. В исходящем дереве каждая вершина, кроме корня, является концом единственного ребра. Если в исходящем дереве имеется ребро xy, то вершину x называют *отцом* вершины y, а вершину y – *сыном* вершины x. Естественный и для многих целей удобный способ задания корневого дерева состоит в указании для каждой вершины ее отца. При этом иногда считают, что корень приходится отцом самому себе – это равносильно добавлению петли при корне.

Если в исходящем дереве T имеется ориентированный путь из вершины x в вершину y, то говорят, что x – *предок* y, а y – *потомок* x. В частности, каждая вершина является предком и потомком самой себя. Множество всех предков вершины x порождает ориентированный путь из корня в x. Множество всех потомков вершины x порождает исходящее дерево с корнем в x, оно называется *ветвью* дерева T в вершине x.

Высотой корневого дерева называется эксцентриситет его корня. Если мы хотим превратить некоторое дерево в корневое и притом минимальной высоты, то в качестве корня следует взять центральную вершину.

1.5.4. Каркасы

Пусть G – обыкновенный граф. Его *каркасом* называется остовный подграф, в котором нет циклов, а области связности совпадают с областями связности графа G. Таким образом, каркас связного графа – дерево, а в общем случае – лес.

У любого графа есть хотя бы один каркас. Действительно, если в G нет циклов, то он сам является собственным каркасом. Если же циклы есть, то можно удалить из графа любое ребро, принадлежащее какому-нибудь циклу. Такое ребро не является перешейком, поэтому при его удалении области связности не изменятся. Продолжая действовать таким образом, после удаления некоторого количества ребер получим остовный подграф, в котором циклов уже нет, а области связности – те же, что у исходного графа, то есть этот подграф и будет каркасом. Можно даже точно сказать, сколько ребер необходимо удалить для получения каркаса. Если в графе n вершин, m ребер и k компонент связности, то в каркасе будет тоже n вершин и k компонент связности. Но в любом лесе с n вершинами и k компонентами связности имеется ровно $n - k$ ребер. Значит, удалено будет $m - n + k$ ребер. Это число называется *цикломатическим числом*

графа и обозначается через $\nu(G)$.

Если в графе есть циклы, то у него больше одного каркаса. Определить точное число каркасов связного графа позволяет так называемая матричная теорема Кирхгофа. Приведем ее без доказательства. Для графа G определим матрицу $K(G)$ – квадратную матрицу порядка n с элементами

$$K_{ij} = \begin{cases} -1, & \text{если } (i,j) \in EG, \\ 0, & \text{если } (i,j) \notin EG \text{ и } i \neq j, \\ \deg(i), & \text{если } i = j. \end{cases}$$

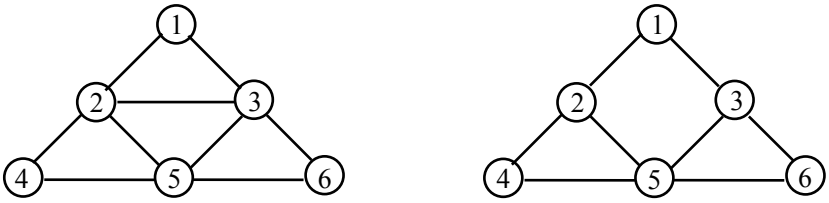
Иначе говоря, $K(G)$ получается из матрицы смежности, если заменить все 1 на –1, а вместо нулей на главной диагонали поставить степени вершин. Заметим, что матрица $K(G)$ – вырожденная, так как сумма элементов каждой строки равна 0, то есть столбцы линейно зависимы.

Теорема 1.10 (матричная теорема Кирхгофа). *Если G – связный граф с не менее чем двумя вершинами, то алгебраические дополнения всех элементов матрицы $K(G)$ равны между собой и равны числу каркасов графа G.*

1.6. Эйлеровы графы

Первая теорема теории графов была доказана задолго до того, как стало употребляться словосочетание «теория графов». В 1736 г. появилась работа Эйлера, в которой не только была решена предложенная ему задача о кенигсбергских мостах, но и сформулировано общее правило, позволяющее решить любую задачу такого рода. Интересно, что в одном из писем Эйлер писал по этому поводу: «...это решение по своему характеру, по-видимому, имеет мало отношения к математике, и мне непонятно, почему следует скорее от математика ожидать этого решения, нежели от какого-нибудь другого человека...».

На языке теории графов задача состоит в том, чтобы определить, имеется ли в графе путь, проходящий через все его ребра (напомним, что путь, по определению, не может дважды проходить по одному ребру). Такой путь называется *эйлеровым путем*, а если он замкнут, то *эйлеровым циклом*. Граф, в котором есть эйлеров цикл, называют *эйлеровым графом*. В графе, изображенном на рис. 1.25,*а*, эйлеров цикл существует – например, последовательность вершин 1, 2, 4, 5, 2, 3, 5, 6, 3, 1 образует такой цикл. В графе же на рис. 1.25,*б* эйлерова цикла нет, но есть эйлеровы пути, например, 2, 4, 5, 2, 1, 3, 5, 6, 3.



а) *б*)

Рис. 1.25

Рассмотрим сначала условия существования эйлерова цикла в обыкновенном графе. Ясно, что в несвязном графе эйлеров цикл может существовать только в том случае, когда все его ребра принадлежат одной компоненте связности, а все остальные компоненты – просто изолированные вершины. Поэтому достаточно рассматривать связные графы.

Теорема 1.11. *Связный граф эйлеров тогда и только тогда, когда в нем степени всех вершин четны.*

Доказательство. Необходимость условия очевидна, так как при каждом прохождении цикла через какую-либо вершину используются два ребра – по одному из них маршрут входит в вершину, по другому выходит из нее (это относится и к стартовой вершине – в ней ведь маршрут должен закончиться). Докажем его достаточность.

Пусть G – связный граф, в котором больше одной вершины и степени всех вершин четны. Значит, степень каждой вершины не меньше 2, поэтому по лемме 1.4 в графе G имеется цикл Z_1. Если удалить все ребра этого цикла из графа G, то получится граф G_1, в котором степени вершин также четны. Если в G_1 нет ни одного ребра, то Z_1 – эйлеров цикл. В противном случае, применяя ту же лемму 1.4 к графу, полученному из G_1 удалением всех изолированных вершин, заключаем, что в G_1 имеется цикл Z_2. Удалив из G_1 все ребра цикла Z_2, получим граф G_2. Продолжая действовать таким образом, пока не придем к пустому графу, получим в итоге систему циклов Z_1, …, Z_k, причем каждое ребро графа принадлежит в точности одному из них. Покажем теперь, что из этих циклов можно составить один цикл. Действительно, из того, что исходный граф связен, следует, что хотя бы один из циклов Z_1, …, Z_{k-1} имеет общую вершину с Z_k. Допустим, для определенности, что таков цикл Z_{k-1}. Пусть $Z_k = x_1, x_2, \ldots, x_p$, $Z_{k-1} = y_1, y_2, \ldots, y_q$, и $x_i = y_j$ для некоторых i и j. Тогда последовательность вершин

$$Z'_{k-1} = x_1, x_2, \ldots, x_i, y_{j+1}, y_{j+2}, \ldots, y_q, y_2, \ldots, y_j, x_{i+1}, \ldots, x_p$$

очевидно, является циклом, а множество ребер этого цикла есть объединение множеств ребер циклов Z_{k-1} и Z_k. Таким образом, получаем систему из меньшего числа циклов, по-прежнему обладающую тем свойством, что каждое ребро графа принадлежит в точности одному из них. Действуя далее таким же образом, в конце концов получим один цикл, который и будет эйлеровым. □

Теорема 1.11 верна и для мультиграфов (кстати, в задаче о кенигсбергских мостах ситуация моделируется именно мультиграфом). Она остается верной и при наличии петель, если при подсчете степеней вершин каждую петлю считать дважды.

Теперь нетрудно получить и критерий существования эйлерова пути.

Теорема 1.12. *Эйлеров путь в связном графе существует тогда и только тогда, когда в нем имеется не более двух вершин с нечетными степенями.*

Доказательство. Если в графе нет вершин с нечетными степенями, то, по предыдущей теореме, в нем имеется эйлеров цикл, он является и эйлеровым путем. Не может быть точно одной вершины с нечетной степенью – это следует из теоремы 1.2. Если же имеются точно две вершины с нечетными степенями, то построим новый граф, добавив ребро, соединяющее эти вершины. В новом графе степени всех вершин четны и, следовательно, существует эйлеров цикл (возможно, что при добавлении нового ребра получатся кратные ребра, но, как отмечалось выше, теорема об эйлеровом цикле верна и для мультиграфов). Так как циклический сдвиг цикла – тоже цикл, то существует и такой эйлеров цикл, в котором добавленное ребро – последнее. Удалив из этого цикла последнюю вершину, получим эйлеров путь в исходном графе. □

В ориентированном графе под эйлеровым путем (циклом) понимают ориентированный путь (цикл), проходящий через все ребра графа. Ориентированный вариант критерия существования эйлерова цикла формулируется следующим образом.

Теорема 1.13. *Эйлеров цикл в связном орграфе существует тогда и только тогда, когда у каждой его вершины число входящих в нее ребер равно числу выходящих.*

1.7. Двудольные графы

Граф называется *двудольным*, если множество его вершин можно так разбить на два подмножества, чтобы концы каждого ребра принадлежали разным подмножествам. Эти подмножества называются *долями*. Таким образом, каждая из долей порождает пустой подграф. Примером двудоль-

ного графа является простая цепь P_n при любом n: одна доля порождается вершинами с четными номерами, другая – с нечетными. Граф K_3 – пример графа, не являющегося двудольным: при любом разбиении множества его вершин на два подмножества в одном из этих подмножеств окажутся две смежных вершины.

Прикладное значение понятия двудольного графа связано с тем, что с помощью таких графов моделируются отношения между объектами двух типов, а такие отношения часто встречаются на практике (например, отношение «продукт x используется в производстве изделия y» между исходными продуктами и готовыми изделиями или «работник x владеет профессией y» между работниками и профессиями). В математике такие отношения тоже нередки, один из наиболее распространенных их видов – отношения инцидентности. Пусть A – множество, а B – семейство его подмножеств. Элемент $x \in A$ и множество $X \in B$ *инцидентны* друг другу, если $x \in X$. Отношение инцидентности можно описать с помощью двудольного графа G, в котором $VG = A \cup B$, $EG = \{(x,X) | \, x \in A, X \in B, x \in X\}$. На рис. 1.26 показан граф отношения инцидентности для $A = \{a, b, c\}$, $B = \{B_1, B_2, B_3, B_4\}$, где $B_1 = \{a\}$, $B_2 = \{a, b, c\}$, $B_3 = \{b, c\}$, $B_4 = \varnothing$.

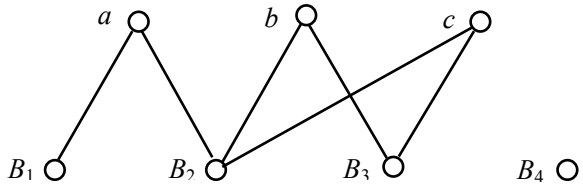


Рис. 1.26

Вообще говоря, разбиение множества вершин двудольного графа на доли можно осуществить не единственным способом. Так, в графе из только что приведенного примера можно взять в качестве долей множества $\{a, b, c, B_4\}$ и $\{B_1, B_2, B_3\}$. В то же время в самом определении этого графа уже заложено «естественное» разбиение на доли A и B. Двудольные графы, возникающие в приложениях, нередко бывают заданы именно так – с множеством вершин, изначально состоящим из двух частей, и с множеством ребер, каждое из которых соединяет вершины из разных частей.

Если разбиение на доли не задано, то может возникнуть вопрос, существует ли оно вообще, то есть является ли данный граф двудольным? Если в графе n вершин, то имеется 2^{n-1} разбиений множества вершин на два подмножества и непосредственная проверка всех этих разбиений будет очень трудоемким делом. Следующая теорема дает критерий двудольно-

сти, а из ее доказательства можно извлечь и эффективный алгоритм проверки двудольности.

Теорема 1.14. *Следующие утверждения для графа G равносильны*:
(1) *G – двудольный граф*;
(2) *в G нет циклов нечетной длины*;
(3) *в G нет простых циклов нечетной длины*.

Доказательство. Докажем, что из (1) следует (2). Пусть G – двудольный граф, в котором выбрано некоторое разбиение на доли, $C = x_1, x_2, \ldots, x_k, x_1$ – цикл длины k в графе G. При любом $i = 1, \ldots, k-1$ вершины x_i и x_{i+1} смежны и, следовательно, принадлежат разным долям. Таким образом, одна доля состоит из всех вершин с нечетными индексами, то есть $x_1, x_3, \ldots$, другая – из всех вершин с четными индексами. Но вершины x_k и x_1 тоже смежны и должны принадлежать разным долям. Следовательно, k – четное число.

Очевидно, что из (2) следует (3); остается доказать, что из (3) следует (1). Рассмотрим граф G, в котором нет простых циклов нечетной длины. Ясно, что граф, в котором каждая компонента связности – двудольный граф, сам двудольный. Поэтому можно считать, что граф G связен. Зафиксируем в нем некоторую вершину a и докажем, что для любых двух смежных между собой вершин x и y имеет место равенство

$$|d(a, x) - d(a, y)| = 1.$$

Действительно, допустим сначала, что $d(a, x) = d(a, y) = t$. Пусть $x_1, x_2, \ldots, x_t$ – кратчайший путь из a в x, $y_1, y_2, \ldots, y_t$ – кратчайший путь из a в y. Эти пути начинаются в одной вершине: $x_1 = y_1 = a$, а оканчиваются в разных: $x_t = x$, $y_t = y$. Поэтому найдется такое k, что $x_k = y_k$ и $x_i \neq y_i$ при всех $i > k$. Но тогда последовательность $x_k, x_{k+1}, \ldots, x_t, y_t, \ldots, y_{k+1}, y_k$ является простым циклом длины $2(t - k) + 1$. Следовательно, $d(a, x) \neq d(a, y)$. Предположим, что $d(a, x) < d(a, y)$. Если $x_1, x_2, \ldots, x_t$ – кратчайший путь из a в x, то, очевидно, $x_1, x_2, \ldots, x_t, y$ – кратчайший путь из a в y, следовательно, $d(a, y) = d(a, x) + 1$. Итак, расстояния от двух смежных вершин до вершины a различаются ровно на единицу. Поэтому, если обозначить через A множество всех вершин графа, расстояние от которых до вершины a четно, а через B множество всех вершин с нечетными расстояниями до a, то для каждого ребра графа один из его концов принадлежит множеству A, другой – множеству B. Следовательно, граф G –двудольный. □

Пусть C – цикл в графе G. Множество вершин цикла C порождает в G подграф, который содержит все ребра этого цикла, но может содержать и

ребра, ему не принадлежащие. Такие ребра называют *хордами* цикла C. Простой цикл, не имеющий хорд, – это порожденный простой цикл. В графе, изображенном на рис. 1.27, хордами цикла 4, 1, 2, 6, 5, 4 являются ребра (1, 5), (1, 6) и (2, 5), а цикл 2, 3, 7, 6, 2 – порожденный простой цикл.

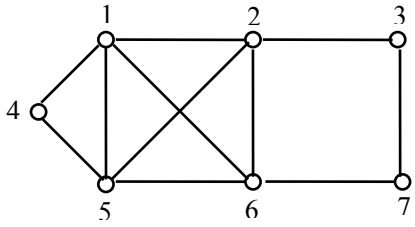


Рис. 1.27

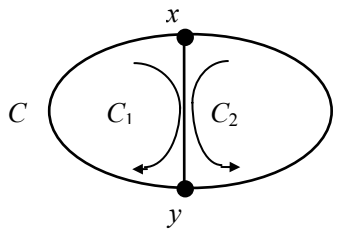


Рис. 1.28

Заметим, что любой цикл длины 3 является порожденным простым циклом.

Пусть C – простой цикл длины k в некотором графе, (x, y) – хорда этого цикла. Ребро (x, y) вместе с ребрами цикла C образует два цикла меньшей длины, C_1 и C_2 (см. рис. 1.28), сумма длин которых равна $k + 2$.

Значит, если C – цикл нечетной длины, то один из циклов C_1, C_2 тоже имеет нечетную длину. Отсюда следует, что в графе, в котором есть цикл нечетной длины, имеется и порожденный простой цикл нечетной длины. Поэтому критерий двудольности справедлив и в следующей формулировке.

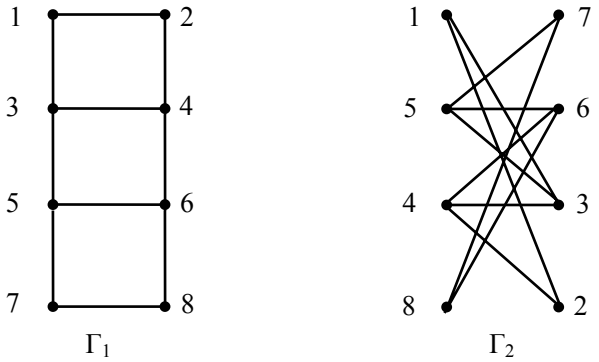
Следствие. *Граф является двудольным тогда и только тогда, когда в нем нет порожденных простых циклов нечетной длины.*

1.8. Планарные графы

Геометрический граф – это плоская фигура, состоящая из вершин – точек плоскости и ребер – линий, соединяющих некоторые пары вершин. Всякий граф можно многими способами представить геометрическим графом, и мы уже не раз пользовались этой возможностью. На рис. 1.29 показаны два геометрических графа Γ_1 и Γ_2, представляющих, как нетрудно проверить, один и тот же обыкновенный граф. Простое устройство этого графа, очевидное на левом изображении, не так легко обнаружить, рассматривая правое. Главная причина этого – в том, что в Γ_1 ребра не имеют «лишних» пересечений.

Геометрический граф, в котором никакие два ребра не имеют общих точек, кроме инцидентной им обоим вершины, называют *плоским графом*,

а по отношению к представляемому им обыкновенному графу – его *плоской укладкой*. Не каждый граф допускает плоскую укладку. Граф, для которого существует плоская укладка, называется *планарным графом*. Кроме удобства визуального анализа, есть немало поводов, в том числе и сугубо практических, для интереса к планарным графам и их плоским укладкам.



Γ_1 Γ_2

Рис. 1.29

Если плоскость разрезать по ребрам плоского графа, она распадется на связные части, которые называют *гранями*. Всегда имеется одна неограниченная *внешняя* грань, все остальные грани называются *внутренними*. Если в плоском графе нет циклов, то у него имеется только одна грань. Если же циклы есть, то граница каждой грани содержит цикл, но не обязательно является циклом. На рис. 1.30 показан плоский граф с пятью пронумерованными гранями. Граница грани с номером 3 состоит из двух циклов, а граница грани с номером 2 кроме цикла длины 5 включает еще дерево из трех ребер. Множества ребер, образующие границы граней, могут быть разными для разных плоских укладок одного и того же графа. На рис. 1.31 показаны две плоские укладки одного графа. В левой укладке есть две грани, границы которых являются простыми циклами длины 5. В правой укладке таких граней нет, но есть грани, ограниченные циклами длины 4 и 6. Однако число граней, как показывает следующая теорема, не зависит от укладки, то есть является инвариантом планарного графа.

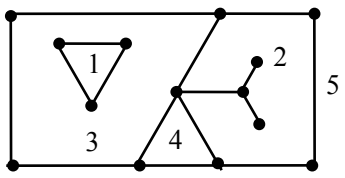


Рис. 1.30

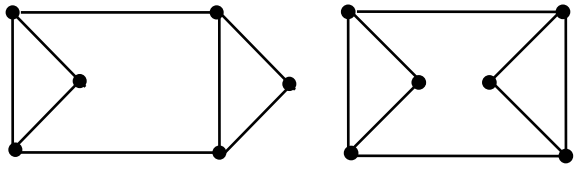


Рис. 1.31

Теорема 1.15 (формула Эйлера). *Количество граней в любой плоской укладке планарного графа, имеющего n вершин, m ребер и k компонент связности, равно* $m - n + k + 1$.

Доказательство. Докажем сначала утверждение теоремы при $k = 1$. Рассмотрим связный плоский граф G . Если в нем нет циклов, то имеется единственная грань, а $m = n - 1$, и формула верна. Если же есть хотя бы один цикл, то возьмем какое-нибудь ребро e, принадлежащее простому циклу C. Это ребро принадлежит границе двух граней, одна из которых целиком лежит внутри цикла C, другая – снаружи. Если удалить ребро e из графа, эти две грани сольются в одну. Граф G_1, полученный из графа G удалением ребра e, очевидно, будет плоским и связным, в нем на одно ребро и на одну грань меньше, чем в G, а число вершин осталось прежним. Если в G_1 еще есть циклы, то, удалив еще одно цикловое ребро, получим граф G_2. Будем продолжать удаление цикловых ребер до тех пор, пока не получится связный плоский граф G_r без циклов, то есть дерево. У него $n - 1$ ребро и единственная грань. Значит, всего было удалено $r = m - n + 1$ ребер, а так как при удалении каждого ребра число граней уменьшалось на единицу, то в исходном графе было $m - n + 2$ грани. Таким образом, формула верна для любого связного плоского графа. Если граф несвязен, то в компоненте связности, имеющей n_i вершин и m_i ребер, как доказано выше, будет $m_i - n_i + 1$ внутренняя грань. Суммируя по всем компонентам и прибавляя 1 для учета внешней грани, убеждаемся в справедливости формулы в общем случае. □

Следствие 1. *Если в планарном графе n вершин,* $n \geq 3$*, и m ребер, то* $m \leq 3(n - 2)$.

Доказательство. Если в графе нет циклов, то $m = n - k$ и неравенство выполняется при $n \geq 3$. Рассмотрим плоский граф G с r гранями, в котором имеются циклы. Пронумеруем грани числами от 1 до r и обозначим через a_i количество ребер, принадлежащих грани с номером i. Так как граница каждой грани содержит цикл, то $a_i \geq 3$ для каждого i, следовательно, $\sum_{i=1}^{r} a_i \geq 3r$. С другой стороны, каждое ребро принадлежит границе не более чем двух граней, поэтому $\sum_{i=1}^{r} a_i \leq 2m$. Из этих двух неравенств следует, что $3r \leq 2m$. Применяя формулу Эйлера, получаем $m \leq 3n - 3k - -3 \leq 3n - 6$ □.

Следствие 1 дает необходимое условие планарности, которое в неко-

торых случаях позволяет установить, что граф не является планарным. Рассмотрим, например, полный граф K_5. У него $n = 5$, $m = 10$ и мы видим, что неравенство из следствия 1 не выполняется. Значит, этот граф непланарен. В то же время существуют графы, не являющиеся планарными, для которых неравенство следствия 1 выполняется. Пример – полный двудольный граф $K_{3,3}$. У него 6 вершин и 9 ребер. Неравенство выполняется, но мы сейчас установим, что он непланарен. Заметим, что в этом графе нет циклов длины 3 (так как он двудольный, в нем вообще нет циклов нечетной длины). Поэтому граница каждой грани содержит не менее четырех ребер. Повторяя рассуждения из доказательства следствия 1, но используя неравенство $a_i \geq 4$ вместо $a_i \geq 3$, получаем следующий результат.

Следствие 2. Если в планарном графе n вершин, $n \geq 3$, m ребер и нет циклов длины 3, то $m \leq 2(n - 2)$.

Для графа $K_{3,3}$ неравенство следствия 2 не выполняется, это доказывает, что он непланарен.

Известно несколько критериев планарности, сформулируем без доказательства два из них. Два графа называют *гомеоморфными,* если из них с помощью подразбиения ребер можно получить изоморфные графы. На рис. 1.32 изображены гомеоморфные графы.

Рис. 1.32

Сформулируем без доказательства два критерия планарности.

Теорема 1.16 (критерий Понтрягина – Куратовского). *Граф планарен тогда и только тогда, когда у него нет подграфов, гомеоморфных K_5 или $K_{3,3}$.*

Граф G называется *стягиваемым* к графу H, если H можно получить из G последовательностью операций стягивания ребер.

Теорема 1.17 (критерий Вагнера). *Граф планарен тогда и только тогда, когда у него нет подграфов, стягиваемых к K_5 или $K_{3,3}$.*

Отметим, что, несмотря на внешнее сходство двух теорем, фигурирующие в них понятия гомеоморфизма и стягиваемости существенно различны. Так, граф Петерсена стягивается к графу K_5, но в нем нет подграфа, гомеоморфного K_5 (см. задачу 11).

Задачи

1. Определите число неориентированных графов с n вершинами, в которых нет кратных ребер, но могут быть петли.

2. Определите число ориентированных графов с n вершинами без петель, в которых каждая пара различных вершин соединена

а) не более чем одним ребром;

б) точно одним ребром.

3. Для любого натурального числа k определим граф Q_k следующим образом. Вершинами его являются всевозможные упорядоченные двоичные наборы длины k. Всего, таким образом, в этом графе 2^k вершин. Вершины $x = (x_1, \ldots, x_k)$ и $y = (y_1, \ldots, y_k)$ смежны в нем тогда и только тогда, когда наборы x и y различаются точно в одной координате. Этот граф называется *k-мерным кубом*. Определите число ребер в графе Q_k.

4. *Граф перестановок порядка k* строится следующим образом. Его вершины соответствуют всевозможным перестановкам элементов 1, 2, ..., k. В этом графе, следовательно, $k!$ вершин. Две вершины смежны тогда и только тогда, когда одна из соответствующих перестановок может быть получена из другой одной транспозицией (перестановкой двух элементов). При $k = 3$ этот граф показан на рис. 1.33. Определите число ребер в графе перестановок порядка k.

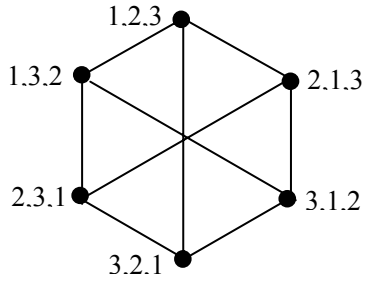


Рис. 1.33

5. Перечислите все попарно неизоморфные графы

а) с 4 вершинами;

б) с 6 вершинами и 3 ребрами;

в) с 6 вершинами и 13 ребрами.

6. Найдите все (с точностью до изоморфизма) графы с 6 вершинами, у которых степень каждой вершины равна 3.

7. Выясните, при каких значениях n существуют регулярные графы степени а) 3; б) 4 с n вершинами.

8. Сколько имеется различных изоморфизмов G_1 в G_2 для графов, изображенных на рис. 1.8?

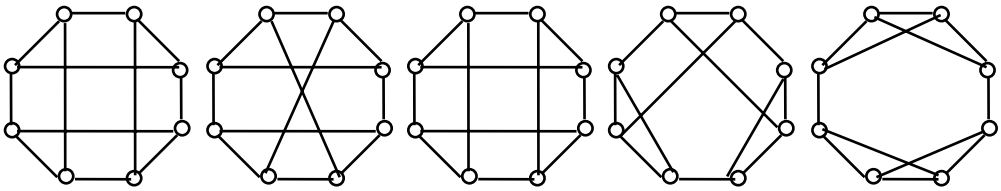
9. Граф, изоморфный своему дополнению, называется самодополнительным.

а) Докажите, что граф C_5 – самодополнительный.

б) Найдите самодополнительный граф с наименьшим числом вершин $n > 1$.

в) Существуют ли самодополнительные графы с 6 вершинами?

10. Выясните, какие из графов, изображенных на рис. 1.34, изоморфны друг другу.



G_1 G_2 G_3 G_4 G_5

Рис. 1.34

11. На рис. 1.35 изображен *граф Петерсена*. Выясните, можно ли из него получить граф K_5 с помощью операций

а) удаления вершин и ребер и подразбиения ребер;

б) стягивания ребер.

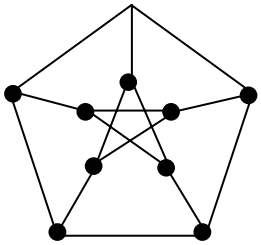


Рис. 1.35

12. Проверьте, что каждый граф с 3 вершинами является либо суммой, либо соединением меньших графов. Верно ли это для графов с 4 вершинами?

13. В графе G_1 имеется n_1 вершин и m_1 ребер, а в графе G_2 – n_2 вершин и m_2 ребер. Сколько ребер будет в графе $G_1 \circ G_2$? В графе $G_1 \times G_2$?

14. Верен ли для произвольных графов G_1, G_2, G_3 «дистрибутивный закон» $(G_1 + G_2) \circ G_3 = (G_1 \circ G_3) + (G_2 \circ G_3)$?

15. Найдите радиус и диаметр каждого из графов C_n, Q_k, $K_{p,q}$, W_n.

16. Сколько имеется в графе Q_n путей длины n, соединяющих вершину (0, 0, ..., 0) с вершиной (1, 1, ..., 1)?

17. Какое наибольшее число шарниров может быть в графе с n вершинами?

18. Докажите, что для любого графа G справедливы неравенства

$$\mathrm{rad}(G) \le \mathrm{diam}(G) \le 2\mathrm{rad}(G).$$

19. Перечислите все (с точностью до изоморфизма) деревья с числом вершин, не превосходящим 6.

20. Пусть в дереве с n вершинами каждая вершина, не являющаяся листом, имеет степень k. Сколько в нем листьев?

21. Сколько ребер в лесе с n вершинами и k компонентами связности?

22. Какой может быть наименьшая высота корневого дерева, у которого каждая вершина имеет не более двух сыновей, если

а) дерево имеет n листьев?

б) дерево имеет n вершин?

23. Выясните, какие из следующих утверждений верны для любого графа G и любого его ребра e:

а) в G существует каркас, содержащий e;

б) в G существует каркас, не содержащий e;

в) если e – не перешеек, то в G существует каркас, не содержащий e.

24. Каждое дерево с множеством вершин $\{1, 2, \ldots, n\}$ является каркасом полного графа K_n. Применяя теорему Кирхгофа, найдите число различных деревьев с n вершинами.

25. При каких n существует эйлеров цикл в графе Q_n?

26. Докажите, что если в связном графе имеется ровно $2k$ вершин с нечетными степенями, то множество его ребер можно разбить на k путей.

27. Верно ли, что для любых двудольных графов G_1 и G_2 граф а) $G_1 \cup G_2$, б) $G_1 \cap G_2$, в) $G_1 \times G_2$ будет двудольным?

26. Докажите, что граф Q_k при любом k является двудольным.

28. Выясните, какие из графов, изображенных на рис. 1.36 , планарны.

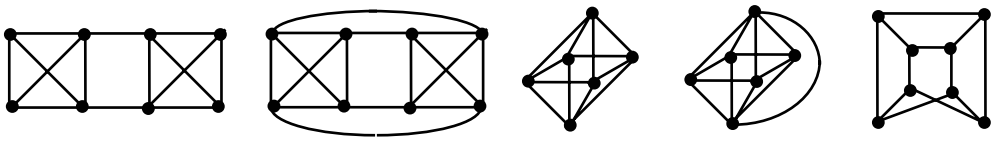


Рис. 1.36

29. Найдите в графе Петерсена подграф, гомеоморфный графу $K_{3,3}$.

30. Какое наименьшее количество ребер нужно удалить из графа K_6, чтобы получить планарный граф?

31. Обобщите необходимое условие планарности из следствия 2 на графы, в которых наименьшая длина цикла равна c.

Глава 2. Анализ графов

В этой главе рассматриваются некоторые задачи исследования структурных свойств и метрических характеристик графов. Нельзя сказать, что это четко определенный круг задач, но, в общем, речь идет об инструментах, с помощью которых мы можем что-то узнать об устройстве графа. Эти знания могут помочь в решении более сложных задач.

Многие задачи анализа графов могут быть решены путем обхода графа с посещением всех его вершин и исследованием всех его ребер. Такой обход можно выполнить многими способами, в действительности же широкое распространение благодаря своей простоте, а в большей степени благодаря своей полезности, получили две стратегии – поиск в глубину и поиск в ширину. Рассмотрение этих алгоритмов и их применений составляет содержание первых трех разделов настоящей главы. В остальных трех разделах рассматриваются задачи анализа циклов в графах.

2.1. Поиск в ширину

2.1.1. Метод поиска в ширину

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер. Какие именно действия выполняются при посещении вершины и исследовании ребра – это зависит от конкретной задачи, для решения которой производится обход. В любом случае, однако, факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной. Вершину, которая еще не посещена, будем называть *новой*. В результате посещения вершина становится *открытой* и остается такой, пока не будут исследованы все инцидентные ей ребра. После этого она превращается в *закрытую*.

Идея поиска в ширину состоит в том, чтобы посещать вершины в порядке их удаленности от некоторой заранее выбранной или указанной стартовой вершины a. Иначе говоря, сначала посещается сама вершина a, затем все вершины, смежные с a, то есть находящиеся от нее на расстоянии 1, затем вершины, находящиеся от a на расстоянии 2, и т.д.

Рассмотрим алгоритм поиска в ширину с заданной стартовой вершиной a. Вначале все вершины помечаются как новые. Первой посещается вершина a, она становится единственной открытой вершиной. В даль-

нейшем каждый очередной шаг начинается с выбора некоторой открытой вершины x. Эта вершина становится *активной*. Далее исследуются ребра, инцидентные активной вершине. Если такое ребро соединяет вершину x с новой вершиной y, то вершина y посещается и превращается в открытую. Когда все ребра, инцидентные активной вершине, исследованы, она перестает быть активной и становится закрытой. После этого выбирается новая активная вершина, и описанные действия повторяются. Процесс заканчивается, когда множество открытых вершин становится пустым.

Основная особенность поиска в ширину, отличающая его от других способов обхода графов, состоит в том, что в качестве активной вершины выбирается та из открытых, которая была посещена раньше других. Именно этим обеспечивается главное свойство поиска в ширину: чем ближе вершина к старту, тем раньше она будет посещена. Для реализации такого правила выбора активной вершины удобно использовать для хранения множества открытых вершин очередь – когда новая вершина становится открытой, она добавляется в конец очереди, а активная выбирается в ее начале. Схематически процесс изменения статуса вершин изображен на рис. 2.1. Черным кружком показана активная вершина.

Рис. 2.1

Опишем процедуру поиска в ширину (BFS – от английского названия этого алгоритма – Breadth First Search) из заданной стартовой вершины a. В этом описании $V(x)$ обозначает множество всех вершин, смежных с вершиной x, Q – очередь открытых вершин. Предполагается, что при посещении вершины она помечается как посещенная и эта пометка означает, что вершина уже не является новой.

Procedure BFS(a)
1 посетить вершину a
2 $a \Rightarrow Q$
3 **while** $Q \neq \varnothing$ **do**
4 $x \Leftarrow Q$
5 **for** $y \in V(x)$ **do**
6 исследовать ребро (x, y)

7 **if** вершина y новая
8 **then** посетить вершину y
9 $y \Rightarrow Q$

Отметим некоторые свойства процедуры BFS.

1. *Процедура BFS заканчивает работу после конечного числа шагов.*

Действительно, при каждом повторении цикла **while** из очереди удаляется одна вершина. Вершина добавляется к очереди только тогда, когда она посещается. Каждая вершина может быть посещена не более одного раза, так как посещаются только новые вершины, а в результате посещения вершина перестает быть новой. Таким образом, число повторений цикла **while** не превосходит числа вершин.

2. *В результате выполнения процедуры BFS будет посещены все вершины из компоненты связности, содержащей вершину a, и только они.*

Очевидно, что вершина может быть посещена только в том случае, когда существует путь, соединяющий ее с вершиной a (так как посещается всегда вершина, смежная с уже посещенной). То, что каждая такая вершина будет посещена, легко доказывается индукцией по расстоянию от данной вершины до вершины a.

3. *Время работы процедуры BFS есть $O(m)$, где m – число ребер в компоненте связности, содержащей вершину a.*

Из предыдущих рассуждений видно, что каждая вершина из этой компоненты становится активной точно один раз. Внутренний цикл **for** для активной вершины x выполняется $\deg(x)$ раз. Следовательно, общее число повторений внутреннего цикла будет равно $\sum_{x \in VG} \deg(x) = 2m$.

Итак, процедура BFS(a) производит обход компоненты связности, содержащей вершину a. Чтобы перейти к другой компоненте, достаточно выбрать какую-нибудь новую вершину, если такие вершины еще имеются, в качестве стартовой. Пусть V – множество вершин графа. Следующий алгоритм осуществляет полный обход графа методом поиска в ширину.

Алгоритм 1. *Поиск в ширину.*

1 пометить все вершины как новые
2 создать пустую очередь Q
3 **for** $a \in V$ **do** **if** a новая **then** BFS(a)

Учитывая, что цикл **for** в строке 3 повторяется n раз, где n – число вершин графа, получаем общую оценку трудоемкости $O(m + n)$. Необходимо отметить, что эта оценка справедлива в предположении, что время, требуемое для просмотра окрестности вершины, пропорционально степе-

ни этой вершины. Это имеет место, например, если граф задан списками смежности. Если же граф задан матрицей смежности, то для просмотра окрестности любой вершины будет затрачиваться время, пропорциональное n. В этом случае общее время работы алгоритма будет оцениваться как $O(n^2)$. Наибольшее значение величины m при данном n равно $n(n-1)/2$, то есть имеет порядок n^2. Таким образом, трудоемкость алгоритма поиска в ширину при задании графа списками смежности не выше, чем при задании матрицей смежности. В целом же первый способ задания предпочтительнее, так как дает выигрыш для графов с небольшим числом ребер.

В качестве простейшего примера применения поиска в ширину рассмотрим задачу выявления компонент связности графа. Допустим, мы хотим получить ответ в виде таблицы, в которой для каждой вершины x указан номер $\mathrm{comp}(x)$ компоненты, которой принадлежит эта вершина. Компоненты будут получать номера в процессе обхода. Для решения этой задачи достаточно ввести переменную c со значением, равным текущему номеру компоненты, и каждый раз при посещении новой вершины x полагать $\mathrm{comp}(x) = c$. Значение c первоначально устанавливается равным 0 и модифицируется при каждом вызове процедуры BFS.

2.1.2. BFS-дерево и вычисление расстояний

Другая простая задача, для решения которой можно применить поиск в ширину, – построение каркаса. Напомним, что каркасом графа называется остовный лес, у которого области связности совпадают с областями связности графа. Каркас связного графа – остовное дерево.

Ребра, исследуемые в процессе обхода графа, можно разделить на две категории: если ребро соединяет активную вершину x с новой вершиной y, то оно классифицируется как *прямое*, в противном случае – как *обратное*. В зависимости от решаемой задачи прямые и обратные ребра могут подвергаться различной обработке.

Предположим, что алгоритм поиска в ширину применяется к связному графу. Покажем, что в этом случае по окончании обхода множество всех прямых ребер образует дерево. Действительно, допустим, что на некотором шаге работы алгоритма обнаруживается новое прямое ребро (x, y), а множество прямых ребер, накопленных к этому шагу, образует дерево F. Тогда вершина x принадлежит дереву F, а вершина y не принадлежит ему. Поэтому при добавлении к дереву F ребра (x, y) связность сохранится, а циклов не появится.

Итак, если применить поиск в ширину к связному графу и запомнить

все прямые ребра, то получим каркас графа. Для произвольного графа будет получен лес, также, очевидно, являющийся каркасом.

Каркас, который будет построен описанным образом в результате поиска в ширину в связном графе, называется BFS-*деревом*. Его можно рассматривать как корневое дерево с корнем в стартовой вершине a. BFS-дерево с заданным корнем a, вообще говоря, не единственно – зависит от того, в каком порядке просматриваются окрестности вершин. Однако всякое BFS-дерево обладает свойством, на котором и основаны наиболее важные применения поиска в ширину. Каркас T связного графа G с корнем a назовем *геодезическим деревом*, если для любой вершины x путь из x в a в дереве T является кратчайшим путем между x и a в графе G.

Теорема 2.1. *Любое BFS-дерево является геодезическим деревом.*

Доказательство. Обозначим через $D(i)$ множество всех вершин графа, находящихся на расстоянии i от стартовой вершины a. Работа алгоритма начинается с посещения стартовой вершины, то есть единственной вершины, составляющей множество $D(0)$. При первом выполнении цикла **while** будут посещены и помещены в очередь все вершины из множества $D(1)$. Затем эти вершины будут одна за другой извлекаться из очереди, становиться активными, и для каждой из них будут исследоваться все смежные с ней вершины. Те из них, которые еще не посещались, будут посещены и помещены в очередь. Но это как раз все вершины из множества $D(2)$ (когда начинается исследование окрестностей вершин из $D(1)$, ни одна вершина из $D(2)$ еще не посещалась и каждая из них смежна хотя бы с одной вершиной из $D(1)$). Следовательно, каждая вершина из $D(2)$ будет посещена после всех вершин из $D(1)$. Рассуждая далее таким образом, приходим к следующему выводу.

(А) Все вершины из $D(i + 1)$ будут посещены после всех вершин из $D(i)$, $i = 0, 1, \ldots$

Строгое доказательство легко провести индукцией по i. Отметим еще следующий факт.

(Б) Если активной является вершина из $D(i)$, то в этот момент все вершины из $D(i)$ уже посещены.

В самом деле, из (А) следует, что вершины из $D(i)$ попадут в очередь после вершин из $D(i - 1)$. Поэтому, когда первая вершина из $D(i)$ становится активной, все вершины из $D(i - 1)$ уже закрыты. Значит, к этому моменту окрестности всех вершин из $D(i - 1)$ полностью исследованы, и, следовательно, все вершины из $D(i)$ посещены.

Рассмотрим теперь момент работы алгоритма, когда активной являет-

ся вершина $x \in D(i)$ и обнаруживается смежная с ней новая вершина y. В BFS-дереве расстояние между y и a на 1 больше, чем расстояние между x и a. В графе расстояние между y и a не больше, чем $i + 1$, так как x и y смежны. Ввиду (А) это расстояние не может быть меньше i, а ввиду (Б) оно не может быть равно i. Значит, $y \in D(i + 1)$, то есть в графе расстояние между y и a тоже на 1 больше, чем расстояние между x и a. Следовательно, если до какого-то момента работы алгоритма расстояния от каждой из посещенных вершин до стартовой вершины в графе и в дереве были равны, то это будет верно и для вновь посещаемой вершины. Поскольку это верно вначале, когда имеется единственная посещенная вершина a (оба расстояния равны 0), то это останется верным и тогда, когда будут посещены все вершины. □

Итак, мы можем применить поиск в ширину для вычисления расстояний от стартовой вершины a до всех остальных вершин графа – нужно только в процессе обхода для каждой посещаемой вершины y определять расстояние от y до a в BFS-дереве. Это сделать легко: $d(a, y) = d(a, x) + 1$, где x – активная вершина. Вначале устанавливаем $d(a, a) = 0$.

Если граф несвязен, некоторые расстояния будут бесконечными. Чтобы учесть эту возможность, положим вначале $d(a, x) = \infty$ для всех $x \neq a$. Пока вершина x остается новой, для нее сохраняется значение $d(a, x) = \infty$, когда же она посещается, $d(a, x)$ становится равным расстоянию между a и x и больше не меняется. Таким образом, бесконечность расстояния можно использовать как признак того, что вершина новая. Если по окончании работы $d(a, x) = \infty$ для некоторой вершины x, это означает, что x не достижима из a, то есть принадлежит другой компоненте связности.

Для того чтобы не только определять расстояния, но и находить кратчайшие пути от a до остальных вершин, достаточно для каждой вершины y знать ее отца $F(y)$ в BFS-дереве. Очевидно, $F(y) = x$, где x – вершина, активная в момент посещения вершины y. Заполнение таблицы F фактически означает построение BFS-дерева.

Модифицируя процедуру BFS с учетом сделанных замечаний, получаем следующий алгоритм.

Алгоритм 2. *Построение BFS-дерева и вычисление расстояний от вершины a до всех остальных вершин*

1 **for** $x \in V$ **do** $d(a, x) := \infty$
2 $d(a, a) := 0$
3 $a \Rightarrow Q$
4 **while** $Q \neq \varnothing$ **do**

```
5          x ⇐ Q
6          for y ∈ V(x) do
7                  if  d(a, y) = ∞
8                          then  d(a, y) = d(a, x) + 1
9                                  F(y) := x
10                                 y ⇒ Q
```

2.2. Поиск в глубину

2.2.1. Метод поиска в глубину

Поиск в глубину – вероятно, наиболее важная ввиду многочисленности приложений стратегия обхода графа. Идея этого метода – идти вперед в неисследованную область, пока это возможно, если же вокруг все исследовано, отступить на шаг назад и искать новые возможности для продвижения вперед. Метод поиска в глубину известен под разными названиями, например «бэктрекинг», «поиск с возвращением».

Понятия новой, открытой, закрытой и активной вершин для поиска в глубину имеют такой же смысл, как и для поиска в ширину. Отметим, что всегда имеется не более чем одна активная вершина.

Обход начинается с посещения заданной стартовой вершины a, которая становится активной и единственной открытой вершиной. Затем выбирается инцидентное вершине a ребро (a, y) и посещается вершина y. Она становится открытой и активной. Заметим, что при поиске в ширину вершина a оставалась активной до тех пор, пока не были исследованы все инцидентные ей ребра. В дальнейшем, как и при поиске в ширину, каждый очередной шаг начинается с выбора активной вершины из множества открытых вершин. Если все ребра, инцидентные активной вершине x, уже исследованы, она превращается в закрытую. В противном случае выбирается одно из неисследованных ребер (x, y), это ребро исследуется. Если вершина y новая, то она посещается и превращается в открытую.

Главное отличие от поиска в ширину состоит в том, что при поиске в глубину в качестве активной выбирается та из открытых вершин, которая была посещена последней. Для реализации такого правила выбора наиболее удобной структурой хранения множества открытых вершин является стек: открываемые вершины складываются в стек в том порядке, в каком они открываются, а в качестве активной выбирается последняя вершина. Схематически это показано на рис. 2.2.

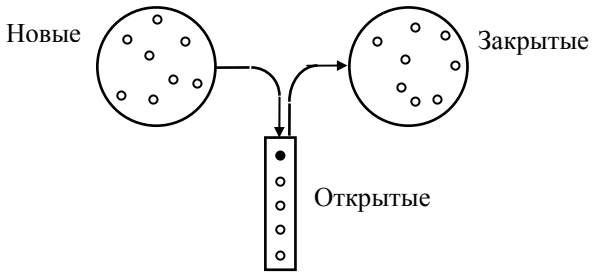


Рис. 2.2

Обозначим стек для открытых вершин через S, остальные обозначения сохраняют тот же смысл, что и в предыдущем разделе. Через top(S) обозначается верхний элемент стека (то есть последний элемент, добавленный к стеку). Процедура обхода одной компоненты связности методом поиска в глубину со стартовой вершиной a тогда может быть записана следующим образом (DFS – от Depth First Search).

Procedure DFS(a)

```
1   посетить вершину a
2   a ⇒ S
3   while S ≠ ∅ do
4           x = top(S)
5           if имеется неисследованное ребро (x, y)
6                   then исследовать ребро (x, y)
7                           if вершина y новая
8                                   then посетить вершину y
9                                           y ⇒ S
10                  else удалить x из S
```

Еще раз обратим внимание на основное отличие этой процедуры от аналогичной процедуры поиска в ширину. При поиске в ширину вершина, став активной, остается ею, пока не будет полностью исследована ее окрестность, после чего она становится закрытой. При поиске в глубину, если в окрестности активной вершины x обнаруживается новая вершина y, то y помещается в стек и при следующем повторении цикла **while** станет активной. При этом x остается в стеке и через какое-то время снова станет активной. Иначе говоря, ребра, инцидентные вершине x, будет исследованы не подряд, а с перерывами.

Алгоритм обхода всего графа – тот же, что и в случае поиска в ширину (алгоритм 1), только нужно очередь заменить стеком, а процедуру BFS – процедурой DFS.

Свойства 1 и 2 поиска в ширину, отмеченные в предыдущем разделе, сохраняются и для поиска в глубину. Остается верной и оценка трудоемкости $O(m + n)$, но ее доказательство требует несколько иных рассуждений, так как каждая вершина теперь может становиться активной несколько раз. Однако каждое ребро рассматривается только два раза (один раз для каждой инцидентной ему вершины), поэтому в операторе **if** в строке 5 ветвь **then** (строки 6-9) повторяется $O(m)$ раз. В этом же операторе ветвь **else** (строка 10) повторяется $O(n)$ раз, так как каждая вершина может быть удалена из стека только один раз. В целом получается $O(m + n)$, причем остаются справедливыми сделанные в предыдущем разделе замечания об условиях, при которых имеет место эта оценка.

2.2.2. DFS-дерево

Поиск в глубину можно применить для нахождения компонент связности графа или для построения каркаса точно таким же образом, как поиск в ширину. Понятия прямого и обратного ребра определяются так же, как в предыдущем разделе, и так же доказывается, что прямые ребра при поиске в глубину образуют каркас графа. Для связного графа каркас, получаемый поиском в глубину, называется *DFS-деревом*. DFS-дерево рассматривается как корневое дерево с корнем в стартовой вершине *a*. Это дерево обладает особыми свойствами, на использовании которых основаны многочисленные применения метода поиска в глубину. Рассмотрим наиболее важное из этих свойств.

Относительно любого корневого остовного дерева все ребра графа, не принадлежащие дереву, можно разделить на две категории. Ребро назовем *продольным*, если одна из его вершин является предком другой, в противном случае ребро назовем *поперечным*. В примере на рис. 2.3 ребра каркаса выделены жирными линиями, корень – черным кружком.

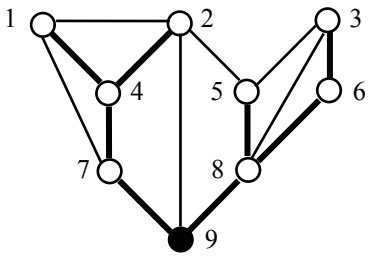


Рис. 2.3

Обратные ребра показаны тонкими линиями, из них продольными являются ребра (1, 7), (2, 9), (3, 8), а поперечными – ребра (1, 2), (2, 5), (3, 5).

Теорема 2.2. *Пусть G – связный граф, T – DFS-дерево графа G. Тогда относительно T все обратные ребра являются продольными.*

Доказательство. Убедимся сначала, что, после того, как стартовая вершина *a* помещена в стек, на каждом последующем шаге работы алгоритма последовательность вершин, хранящаяся в стеке, образует путь с началом в вершине *a*, а все ребра этого пути принадлежат дереву. Вначале это, очевидно, так. В дальнейшем всякий раз, когда новая вершина *y* помещается в стек, к дереву добавляется прямое ребро (*x*, *y*), причем вершина *x* находится в стеке перед вершиной *y*. Значит, если указанное свойство имело место до добавления вершины в стек, то оно сохранится и после добавления. Удаление же вершины из стека, конечно, не может нарушить этого свойства.

Пусть теперь (*x*, *y*) – обратное ребро. Каждая из вершин *x* и *y* в ходе работы алгоритма когда-либо окажется в стеке. Допустим, *x* окажется там раньше, чем *y*. Рассмотрим шаг алгоритма, на котором *y* помещается в стек. В этот момент *a* еще находится в стеке. Действительно, вершина исключается из стека только тогда, когда в ее окрестности нет непосещенных вершин. Но непосредственно перед помещением в стек вершина *y* является новой и принадлежит окрестности вершины *x*. Таким образом, вершина *x* лежит на пути, принадлежащем дереву и соединяющем вершины *a* и *y*. Но это означает, что вершина *x* является предком вершины *y* в дереве *T* и, следовательно, ребро (*x*, *y*) – продольное. □

Таким образом, каркас, изображенный на рис. 2.3, не мог быть построен методом поиска в глубину. Кстати, он не мог быть построен и с помощью поиска в ширину (почему?).

2.2.3. Другие варианты алгоритма поиска в глубину

Ввиду важности этого метода опишем еще два варианта алгоритма поиска в глубину. Первый из них – рекурсивный, и, как обычно, рекурсия дает возможность представить алгоритм в наиболее компактной форме. Для того чтобы алгоритм выполнял какую-то полезную работу, будем нумеровать вершины в том порядке, в каком они встречаются при обходе. Номер, получаемый вершиной *x*, обозначается через *Dnum*(*x*) и называется ее *глубинным номером*. Вначале полагаем *Dnum*(*x*) = 0 для всех *x*. Это нулевое значение сохраняется до тех пор, пока вершина не становится открытой, в этот момент ей присваивается ее настоящий глубинный номер. Таким образом, нет необходимости в какой-либо специальной структуре для запоминания новых вершин – они отличаются от всех других нулевым значением *Dnum*. Переменная *c* хранит текущий номер. Рекурсивная про-

цедура DFSR обходит одну компоненту связности, а алгоритм 3 обходит весь граф и присваивает вершинам глубинные номера.

Алгоритм 3. *Поиск в глубину с вычислением глубинных номеров – рекурсивный вариант*

```
1   for x ∈ V do Dnum(x) := 0
2   c := 0
3   for x ∈ V  do
4           if Dnum(x) = 0 then DFSR(x)
```

Procedure DFSR(x)

```
1   c := c + 1
2   Dnum(x) := c
3   for y ∈ V  do
4   if Dnum(y) = 0 then DFSR(y)
```

Следующий вариант алгоритма поиска в глубину отличается тем, что не использует стека для хранения открытых вершин. Стек нужен для того, чтобы в момент, когда окрестность активной вершины x исследована и необходимо сделать «шаг назад», можно было определить вершину, в которую нужно вернуться. Но это та вершина, которая является отцом вершины x в DFS-дереве. Поэтому, если решение задачи предусматривает построение DFS-дерева, то это дерево можно использовать и для организации «возвратных движений» в процессе обхода. Описываемый ниже алгоритм строит каркас произвольного графа, каждая компонента связности этого каркаса является DFS-деревом соответствующей компоненты связности графа. Через $F(x)$ обозначается отец вершины x в этом DFS-дереве, при этом для корня дерева (стартовой вершины) a полагаем $F(a) = a$. Здесь и далее в описаниях алгоритмов инструкция «открыть (закрыть) вершину» означает, что вершина каким-то образом помечается как открытая (закрытая).

Алгоритм 4. *Поиск в глубину с построением каркаса*

```
1   пометить все вершины как новые
2   for a ∈ V do
            if вершина a новая then DFST(a)
```

Procedure DFST(a)

```
1   F(a) := a
2   открыть вершину a
3   x := a
4   while x открытая do
```

```
5          if имеется неисследованное ребро (x, y)
6               then исследовать ребро (x, y)
7                    if вершина y новая
8                         then F(y) := x
9                              открыть вершину y
10                             x := y
11                   else закрыть вершину x
12                        x := F(x)
```

2.2.4. Шарниры

В качестве примера задачи, для эффективного решения которой можно использовать основное свойство DFS-дерева, выражаемое теоремой 2.2, рассмотрим задачу выявления шарниров в графе. Напомним, что шарниром называется вершина, при удалении которой увеличивается число компонент связности. Для простоты будем сейчас считать, что рассматриваемый граф связен, так что шарнир – это вершина, при удалении которой нарушается связность.

Отсутствие поперечных ребер относительно DFS-дерева позволяет очень просто узнать, является ли стартовая вершина a (корень этого дерева) шарниром.

Лемма 2.3. *Стартовая вершина a является шарниром графа тогда и только тогда, когда ее степень в DFS-дереве больше 1.*

Доказательство. Если вершину a удалить из дерева, то оно распадется на поддеревья, называемые ветвями. Число ветвей равно степени вершины a в дереве. Так как поперечных ребер нет, то вершины из разных ветвей не могут быть смежными в графе и каждый путь из одной ветви в другую обязательно проходит через вершину a. Следовательно, если степень вершины a в DFS-дереве больше 1, то эта вершина – шарнир. Если же степень вершины a в DFS-дереве равна 1, то в дереве имеется единственная вершина b, смежная с a, и каждая из остальных вершин графа соединена с вершиной b путем, не проходящим через a. Поэтому в этом случае удаление вершины a не нарушает связности графа и эта вершина не является шарниром. □

Это свойство корня DFS-дерева можно было бы использовать для выявления всех шарниров, просто выполнив n раз поиск в глубину, стартуя поочередно в каждой вершине. Оказывается, все шарниры можно выявить однократным поиском в глубину. Следующая теорема характеризует все шарниры, отличные от корня DFS-дерева. Напомним, что каждая вершина дерева является и предком, и потомком самой себя. Предок (потомок)

вершины, отличный от самой этой вершины, называется собственным предком (потомком).

Лемма 2.4. *Пусть T – DFS-дерево графа G с корнем a. Вершина $x \neq a$ является шарниром графа тогда и только тогда, когда у нее в дереве T имеется такой сын y, что ни один потомок вершины y не соединен ребром ни с одним собственным предком вершины x.*

Доказательство. Если y – сын вершины x и ни один потомок вершины y не соединен ребром ни с одним собственным предком вершины x, то, ввиду отсутствия поперечных ребер, любой путь, соединяющий вершину y с корнем, проходит через x. Следовательно, в этом случае вершина x – шарнир. Если же для каждого сына y вершины x имеется ребро, соединяющее вершину y с каким-либо собственным предком вершины x, то каждый сын вершины x соединен с корнем дерева путем, не проходящим через x. Поэтому при удалении вершины x граф останется связным и x в этом случае не является шарниром □.

Для применения этого критерия к отысканию шарниров введем на множестве вершин функцию *Low*, связанную с DFS-деревом: значением $Low(x)$ является наименьший из глубинных номеров вершин, смежных с потомками вершины x. Если вершина y является сыном вершины x, то $Low(y) \leq Dnum(x)$ (так как вершина y является потомком самой себя и смежна с вершиной x). Из леммы 2.4 следует, что вершина x, отличная от a, является шарниром тогда и только тогда, когда у нее имеется сын y такой, что $Low(y) = Dnum(x)$.

Функцию *Low* можно определить рекурсивно – если мы знаем ее значения для всех сыновей вершины x и глубинные номера всех вершин, смежных с x и не являющихся ее сыновьями, то $Low(x)$ есть минимум из всех этих величин, то есть

$$Low(x) = \min\left(\min_{y \in A} Low(y), \min_{y \in B} Dnum(y) \right),$$

где A обозначает множество всех сыновей вершины x, а B – множество всех остальных вершин, смежных с x. Нетрудно видеть, что это определение эквивалентно первоначальному. Исходя из него, можно вычислять значения функции *Low* в процессе поиска в глубину с помощью следующей рекурсивной процедуры. Предполагается, что вначале всем элементам массива *Dnum* присвоены нулевые значения.

Procedure ComputeLow(x)

1 $c := c + 1$

2 $Dnum(x) := c$

```
3   Low(x) := c
4   for y ∈ V(x) do
5           if  Dnum(y) = 0
6                   then ComputeLow(y)
7                           Low(x) := min(Low(x), Low(y))
8           else Low(x) := min(Low(x), Dnum(y))
```

2.3. Блоки

Если граф состоит из нескольких компонент связности, то его можно изучать «по частям», и это может упростить описание графа и облегчить решение многих задач. Однако и связный граф иногда можно представить как состоящий из частей и такое представление также может быть полезным. После компонент связности простейшими частями такого рода являются блоки (называемые также компонентами двусвязности). Блок – это максимальный подграф графа, не имеющий собственных шарниров (то есть некоторые шарниры графа могут принадлежать блоку, но своих шарниров у блока нет). На рис. 2.4 изображены граф G и его блоки $B_1 - B_5$. Ниже будет дано другое определение блока, из которого видно, почему блоки называют компонентами двусвязности. Затем в этом разделе будут рассмотрены некоторые свойства блоков и описан алгоритм выявления блоков, основанный на поиске в глубину.

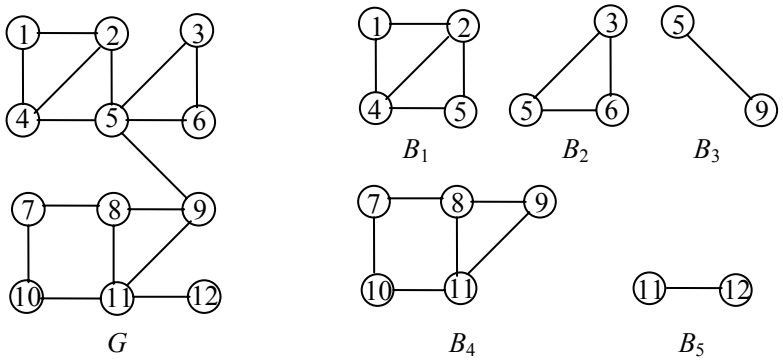


Рис. 2.4

2.3.1. Двусвязность

Связный граф с не менее чем тремя вершинами, в котором нет шарниров, называется *двусвязным*. Примеры двусвязных графов – цикл C_n и полный граф K_n, $n \geq 3$, цепь же P_n не является двусвязным графом ни при каком n.

Будем говорить, что два элемента графа (напомним, что элементы графа – это вершины и ребра) *циклически связаны*, если в графе имеется простой цикл, содержащий оба эти элемента.

Теорема 2.5. *В двусвязном графе любые два различных элемента циклически связаны. Если в графе любые два ребра циклически связаны, то он двусвязен*.

Доказательство. Докажем сначала, что в двусвязном графе G для любых двух различных вершин a и b имеется простой цикл, проходящий через обе эти вершины. Доказательство проводим индукцией по расстоянию между a и b. Если $d(a, b) = 1$, то a и b смежны. Ребро (a, b) не является перешейком (иначе хотя бы одна из вершин a, b была бы шарниром). Но тогда из леммы 1.6 следует, что в графе имеется простой цикл, проходящий через это ребро. Пусть $d(a, b) > 1$. Рассмотрим кратчайший путь из a в b, и пусть x – предпоследняя вершина этого пути. Тогда $d(a, x) = d(a, b) - 1$ и, по предположению индукции, существует простой цикл C, содержащий вершины a и x. Так как вершина x – не шарнир, то существует простой путь P из b в a, не проходящий через x. Пусть y – первая вершина этого пути, принадлежащая C (такая существует, так как $a \in C$). Тогда отрезок пути P от b до y вместе с отрезком цикла от y до x, содержащим вершину a, и с ребром (x, b) образует простой цикл, содержащий обе вершины a и b (показан стрелками на рис. 2.5).

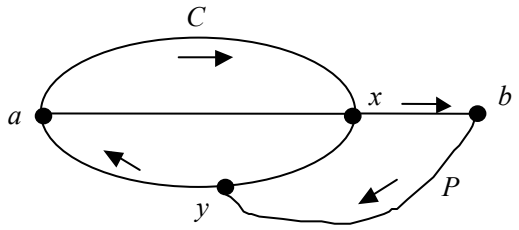


Рис. 2.5

Теперь покажем, что для любой вершины a и любого ребра (x, y) двусвязного графа G в нем имеется цикл, содержащий эту вершину и это ребро. Как доказано выше, существует простой цикл C_1, содержащий вершины a и x. Если этот цикл проходит и через y, то, заменив в нем отрезок от x до y, не содержащий a, ребром (x, y), получим простой цикл, проходящий через вершину a и ребро (x, y). В противном случае возьмем цикл C_2, содержащий вершины a и y. Кратчайший отрезок этого цикла, соединяющий y с какой-либо вершиной z на C_1, вместе с отрезком цикла C_1 от z до x, содержащим вершину a, и с ребром (x, y) образует простой

цикл, содержащий это ребро и вершину a.

Доказательство того, что в двусвязном графе через любые два ребра проходит простой цикл, почти в точности повторяет предыдущее, только вместо вершины a нужно рассматривать ребро (a, b).

Остается доказать, что если в графе G через любые два различных элемента проходит простой цикл, то этот граф – двусвязный. Действительно, допустим, что вершина a – шарнир графа G. Возьмем вершины x и y, смежные с a и принадлежащие разным компонентам связности графа, получающегося при удалении вершины a. Тогда в G не существует цикла, содержащего оба ребра (a, x) и (a, y). □

Из этой теоремы следует, что свойство двусвязности можно охарактеризовать следующим образом.

Следствие. Граф с не менее чем двумя ребрами двусвязен тогда и только тогда, когда в нем любые два различных ребра циклически связаны.

Рассмотрим подробнее отношение циклической связанности ребер. Оно, очевидно, симметрично. Будем считать, что каждое ребро циклически связано с самим собой, тогда это отношение будет и рефлексивным. Докажем, что оно на самом деле является отношением эквивалентности.

Теорема 2.6. *Для любого графа отношение циклической связанности ребер является отношением эквивалентности.*

Доказательство. Остается доказать транзитивность этого отношения. Пусть C_1 – простой цикл, содержащий ребра e_1 и e_2, а C_2 – простой цикл, содержащий ребра e_2 и e_3; покажем, что существует простой цикл, содержащий ребра e_1 и e_3. Если e_1 принадлежит C_2, то последний и является этим циклом. Если же e_1 не принадлежит C_2, то в C_1 есть отрезок P_1, включающий e_1, у которого концевые вершины a и b принадлежат C_2, а все внутренние вершины не принадлежат C_2. Пусть P_2 – отрезок цикла C_2, концами которого являются a и b и который включает ребро e_3. Соединение P_1 и P_2 дает простой цикл, содержащий e_1 и e_3. □

Итак, множество ребер любого графа разбивается на классы эквивалентности по отношению циклической связанности. Каждый перешеек образует отдельный класс эквивалентности. Если граф двусвязен, то имеется единственный класс циклической связанности.

2.3.2. Блоки и BC-деревья

Блоком графа G называется подграф B, удовлетворяющий одному из трех условий:

а) B состоит из одной изолированной вершины графа G (такой блок

называется тривиальным);

б) B порождается единственным ребром, которое является перешейком в G;

в) B является максимальным двусвязным подграфом графа G.

Из последних двух теорем следует, что ребра нетривиального блока образуют класс циклической связанности. Следовательно, различные блоки не имеют общих ребер. Однако, в отличие от компонент связности, блоки могут иметь общие вершины. Таким общими вершинами, как показывает следующая теорема, могут быть только шарниры графа.

Теорема 2.7. *Два различных блока одного графа могут иметь не более одной общей вершины. Вершина принадлежит более чем одному блоку тогда и только тогда, когда она является шарниром графа.*

Доказательство. Пусть B_1 и B_2 – различные блоки графа G. Рассмотрим подграф $B = B_1 \cup B_2$. Он не является блоком, следовательно, или несвязен, или имеет шарнир. Если B несвязен, то B_1 и B_2 – его компоненты связности и, следовательно, не имеют общих вершин. Если же B связен и a – шарнир в B, то после удаления вершины a граф B распадается на компоненты связности. При этом все вершины подграфа B_1, отличные от a, принадлежат одной компоненте, иначе a была бы шарниром в B_1. То же верно для вершин подграфа B_2. Значит, имеется всего две компоненты, одна из которых состоит полностью из вершин графа B_1, другая – из вершин графа B_2. Следовательно, a – единственная общая вершина B_1 и B_2.

Если вершина x принадлежит более чем одному блоку, то она инцидентна двум ребрам, (x, y_1) и (x, y_2), принадлежащим разным блокам, то есть не являющимся циклически связанными. Но тогда всякий путь, соединяющий y_1 и y_2, проходит через x, следовательно, по лемме 1.5, x – шарнир. Обратно, если x – шарнир, то найдутся две смежные с x вершины y_1 и y_1, принадлежащие разным компонентам связности графа, получаемого удалением вершины x. Но тогда ребра (x, y_1) и (x, y_2) не являются циклически связанными, следовательно, принадлежат разным блокам. □

Строение связного графа, состоящего из нескольких блоков, может быть схематически описано с помощью так называемого *дерева блоков и шарниров*, кратко именуемого *BC-деревом*. В этом дереве имеются две категории вершин – одни поставлены в соответствие блокам графа, другие – его шарнирам. Вершина-блок в дереве соединяется ребром с вершиной-шарниром, если в графе соответствующий шарнир принадлежит соответствующему блоку. На рис. 2.6 изображено BC-дерево для графа с рис. 2.4. Блоки изображены белыми, а шарниры черными кружками.

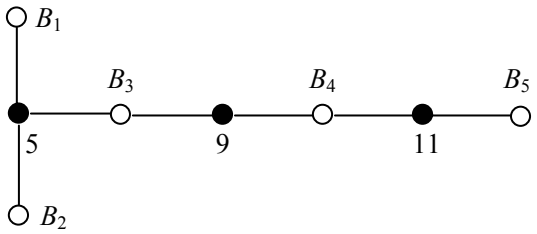


Рис. 2.6

2.3.3. Выявление блоков

Рассмотрим связный граф G и в нем DFS-дерево T, построенное поиском в глубину из стартовой вершины a. Через $F(x)$ будем обозначать отца вершины x в этом дереве, при этом считаем, что $F(a) = a$. Будем также считать, что в процессе обхода графа вычисляются значения функций *Dnum* и *Low*, определенных в предыдущем разделе.

Пусть B – блок графа, а x – вершина этого блока с наименьшим значением $Dnum(x)$. Иначе говоря, x – вершина блока, посещаемая при обходе первой. Среди сыновей вершины x имеется единственная вершина y, принадлежащая блоку B. Вершину x будем называть начальной вершиной, а ребро (x, y) – начальным ребром блока B.

Лемма 2.8. *Пусть $x = F(y)$ в DFS-дереве T. Ребро (x, y) является начальным ребром некоторого блока тогда и только тогда, когда $Low(y) = Dnum(x)$.*

Доказательство. Если $x = a$, то для каждого сына y вершины x имеет место равенство $Low(y) = Dnum(x)$ и ребро (x, y) является начальным ребром некоторого блока.

Пусть $x \neq a$ и (x, y) – начальное ребро блока B. Предположим, что $Low(y) \neq Dnum(x)$. Это означает, что имеется ребро, соединяющее некоторого потомка вершины y с собственным предком вершины x. Но тогда ребра (x, y) и $(x, F(x))$ оказываются циклически связанными, а отсюда следует, что вершина $F(x)$ принадлежит блоку B. Это противоречит тому, что x – начальная вершина блока, так как $Dnum\ F(x) < Dnum(x)$.

Обратно, пусть $Low(y) = Dnum(x)$. Тогда вершина x является шарниром графа. Рассмотрим поддерево, состоящее из всех потомков вершины y. Ни одна из вершин этого поддерева не смежна ни с одной отличной от x вершиной вне поддерева. Значит, все вершины блока, содержащего ребро (x, y), принадлежат этому поддереву, и (x, y) – начальное ребро этого блока.

В основе описываемого ниже алгоритма выявления блоков лежит ре-

курсивная процедура вычисления функции *Low* из предыдущего раздела. Напомним, что *Low*(x) есть наименьший из глубинных номеров вершин, смежных с потомками вершины x. Переменная k – счетчик блоков, $B(k)$ – множество вершин блока с номером k. В стеке S накапливаются вершины графа, впервые встречающиеся в процессе обхода (то есть превращающиеся из новых в открытые).

Множества вершин блоков строит процедура NewBlock. Она вызывается всякий раз, когда обнаруживается начальное ребро (x, y) некоторого блока (выполняется равенство *Low*(y) = *Dnum*(x)). Эта процедура включает в новое множество $B(k)$ вершины x, y и все вершины, находящиеся в стеке выше вершины y. Эти вершины удаляются из стека (кроме вершины x, которая является начальной вершиной блока и может принадлежать еще и другим блокам). Для обоснования алгоритма остается убедиться в том, что блок состоит именно из этих вершин. Доказательство можно провести индукцией по номеру блока k. Вершина y помещается в стек S, когда она становится открытой, а условие *Low*(y) = *Dnum*(x) проверяется для вершины y тогда, когда она превращается в закрытую. Все вершины, помещаемые в стек между этими двумя событиями, будут потомками вершины y в DFS-дереве, каждый потомок вершины y будет помещен в стек после y, и когда y становится закрытой, все эти вершины уже закрыты. Если $k = 1$, то среди потомков вершины y нет начальных вершин блоков (иначе номер этого блока был бы больше 1), следовательно, блок с начальным ребром (x, y) состоит из всех этих вершин и вершины x. Если же $k > 1$, то, по предположению индукции, все вершины других блоков, состоящих из потомков вершины y, не принадлежащие блоку $B(k)$, к моменту обнаружения начального ребра (x, y) уже удалены из стека, следовательно, $B(k)$ состоит в точности из x, y и вершин, находящихся в стеке выше вершины y.

Алгоритм 5. *Выявление блоков*

1 **for** $x \in V$ **do** *Dnum*(x) := 0
2 $c := 0$
3 $k := 0$
4 **for** $x \in V$ **do if** *Dnum*(x) = 0 **then** Blocks(x)

Procedure Blocks(x)

5 $c := c + 1$
6 *Dnum*(x) := c
7 *Low*(x) := c
8 $x \Rightarrow S$

```
9   for y ∈ V (x) do
10          if Dnum(y) = 0
11                  then Blocks(y)
12                          Low(x) := min(Low (x), Low(y))
13                                  if Low(y) = Dnum(x) then NewBlock
14          else Low(x) := min(Low (x), Dnum(y))
Procedure NewBlock
1   k := k + 1
2   B(k) := x
3   repeat
4           z ⇐ S
5           B(k) := B(k) ∪ {z}
5   until z = y
```

2.4. База циклов

2.4.1. Пространство подграфов

Зафиксируем некоторое множество V и рассмотрим множество Γ_V всех графов с множеством вершин V. Буквой O будем обозначать пустой граф из этого множества: $O = (V, \varnothing)$.

Для графов $G_1 = (V, E_1)$ и $G_2 = (V, E_2)$ из Γ_V определим их *сумму по модулю 2* (в дальнейшем в этом разделе будем называть ее просто суммой) как граф $G_1 \oplus G_2 = (V, E_1 \oplus E_2)$, где $E_1 \oplus E_2$ обозначает симметрическую разность множеств E_1 и E_2. Иначе говоря, ребро принадлежит графу $G_1 \oplus G_2$ тогда и только тогда, когда оно принадлежит в точности одному из графов G_1 и G_2. Пример показан на рис. 2.7.

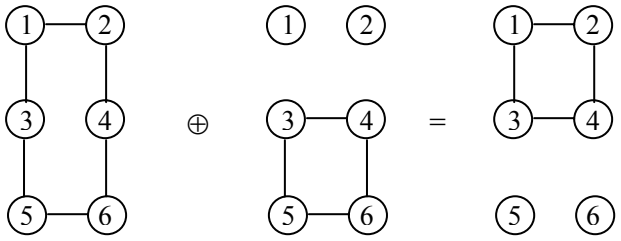


Рис. 2.7

Следующие свойства введенной операции очевидны или легко проверяются.

1) Коммутативность: $G_1 \oplus G_2 = G_2 \oplus G_1$ для любых G_1 и G_2.

2) Ассоциативность: $G_1 \oplus (G_2 \oplus G_3) = (G_1 \oplus G_2) \oplus G_3$ для любых G_1, G_2, G_3.

3) $G \oplus O = G$ для любого G.

4) $G \oplus G = O$ для любого G.

Отсюда следует, что множество Γ_V относительно операции $\oplus$ образует абелеву группу. Нейтральным элементом («нулем») этой группы служит граф O, а противоположным к каждому графу является сам этот граф. Уравнение $G \oplus X = H$ с неизвестным X и заданными графами G и H имеет единственное решение $X = G \oplus H$. Благодаря свойству ассоциативности мы можем образовывать выражения вида $G_1 \oplus G_2 \oplus \ldots \oplus G_k$, не используя скобок для указания порядка действий. Легко понять, что ребро принадлежит графу $G_1 \oplus G_2 \oplus \ldots \oplus G_k$ тогда и только тогда, когда оно принадлежит нечетному количеству из графов G_1, G_2, …, G_k.

Рассмотрим множество из двух элементов $\{0, 1\}$. Оно является полем относительно операций умножения и сложения по модулю 2. Определим операцию умножения элементов этого поля на графы: $0 \cdot G = O$, $1 \cdot G = G$ для любого графа G. Множество Γ_V с введенными операциями сложения графов и умножения на элементы поля является линейным векторным пространством.

Зафиксируем некоторый граф $G \in \Gamma_V$ и рассмотрим множество всех его остовных подграфов, которое будем обозначать $S[G]$. Это множество состоит из $2^{m(G)}$ элементов, среди них сам граф G и граф O. Оно замкнуто относительно сложения графов и умножения на элементы поля, следовательно, является подпространством пространства Γ_V. Его называют *пространством подграфов* графа G.

Любой граф из $S[G]$ может быть выражен как сумма однореберных подграфов. Всего у графа G имеется $m(G)$ однореберных подграфов и они, очевидно, линейно независимы. Следовательно, однореберные подграфы образуют базис пространства $S[G]$, а размерность этого пространства равна $m(G)$.

В пространстве $S[G]$ можно очень естественным способом ввести координаты. Пронумеруем ребра графа G: $EG = \{e_1, e_2, \ldots, e_m\}$. Теперь остовному подграфу H можно поставить в соответствие характеристический вектор $\alpha(H) = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ его множества ребер:

$$\alpha_i = \begin{cases} 1, \text{ если ребро } e_i \text{ принадлежит } H, \\ 0, \text{ если ребро } e_i \text{ не принадлежит } H. \end{cases}$$

Получаем взаимно однозначное соответствие между множеством $S[G]$ и множеством всех двоичных векторов с m координатами. Сумме графов соответствует векторная (покоординатная) сумма по модулю 2 их характеристических векторов.

2.4.2. Квазициклы

В этом разделе слово «цикл» будем понимать несколько иначе, чем до сих пор. Именно, циклом будем называть граф, у которого одна компонента связности является простым циклом, а остальные – изолированными вершинами. Рисунок 2.7 показывает, что в результате сложения двух циклов иногда получается цикл. Это не всегда так (например, когда складываемые циклы не имеют общих ребер), но все-таки графы, которые можно получить, складывая циклы, обладают определенными особенностями. На этом основан алгебраический подход к изучению устройства множества циклов графа.

Рассмотрим некоторый граф $G \in \Gamma_V$. Среди его остовных подграфов, возможно, имеется некоторое количество циклов. Обозначим через $C[G]$ подпространство пространства подграфов, порождаемое всеми этими циклами. $C[G]$ называется *пространством циклов* графа G. Оно содержит граф O (если в G нет циклов, то O является единственным элементом пространства циклов), а все остальные его элементы – это всевозможные линейные комбинации циклов графа G. Заметим, что коэффициентами в линейных комбинациях являются элементы множества $\{0, 1\}$, поэтому речь идет на самом деле просто о всевозможных суммах циклов.

Остовный подграф, у которого степени всех вершин четны, называется *квазициклом*. Оказывается, множество $C[G]$ состоит в точности из всех квазициклов графа G. Прежде чем доказать это, покажем сначала, что множество всех квазициклов замкнуто относительно сложения.

Лемма 2.9. *Сумма двух квазициклов есть квазицикл.*

Доказательство. Пусть H_1 и H_2 – квазициклы. Рассмотрим произвольную вершину $a \in V$, и пусть ее степени в H_1 и H_2 равны соответственно d_1 и d_2. Тогда степень вершины a в графе $H_1 \oplus H_2$ будет равна $d = d_1 + d_2 - 2d_{1,2}$, где $d_{1,2}$ – число вершин, с которыми a смежна в обоих графах H_1 и H_2. Отсюда видно, что число d четно, если четны оба числа d_1 и d_2. □

Следующая лемма объясняет строение квазициклов.

Лемма 2.10. *Любой квазицикл с непустым множеством ребер является объединением простых циклов, не имеющих общих ребер.*

Доказательство. В квазицикле H в любой компоненте связности, состоящей не менее чем из двух вершин, степени всех вершин не меньше 2,

и, по лемме 1.4, в нем есть цикл, а, значит, и простой цикл. Взяв какой-нибудь простой цикл в H и удалив его ребра из H, снова получим квазицикл. Если в этом новом квазицикле есть хотя бы одно ребро, то в нем также имеется простой цикл, и т.д. В конце концов, когда останется пустой граф, будет построено семейство простых циклов, не имеющих общих ребер и в совокупности содержащих все ребра графа H. □

Теорема 2.11. *Граф принадлежит множеству $C[G]$ тогда и только тогда, когда он является квазициклом графа G.*

Доказательство. Всякий цикл является квазициклом. Так как элементы $C[G]$ – это суммы циклов, то, по лемме 2.9, все они – квазициклы. Обратное утверждение (каждый квазицикл принадлежит $C[G]$) следует из леммы 2.10, так как объединение циклов, не имеющих общих ребер, совпадает с их суммой. □

2.4.3. Фундаментальные циклы

Компактное представление пространства дает его базис. Если выписать все простые циклы графа G, то это в большинстве случаев не будет его базисом, так как некоторые из этих циклов могут быть суммами других (см. пример на рис. 2.7). Построить базис пространства $C[G]$, состоящий из простых циклов, можно следующим образом. Выберем в графе G какой-нибудь каркас T. Пусть $e_1, \ldots, e_S$ – все ребра графа G, не принадлежащие T. Если добавить к T ребро e_i, то в полученном графе образуется единственный (простой) цикл Z_i. Таким образом, получаем семейство из s циклов, они называются *фундаментальными циклами* относительно каркаса T.

Теорема 2.12. *Множество всех фундаментальных циклов относительно любого каркаса T графа G образует базис пространства циклов этого графа.*

Доказательство. Зафиксируем некоторый каркас T и рассмотрим фундаментальные циклы $Z_1, Z_2, \ldots, Z_S$ относительно этого каркаса. В каждом из этих циклов имеется ребро e_i, принадлежащее этому циклу и не принадлежащее никакому из остальных. Поэтому при сложении этого цикла с другими фундаментальными циклами это ребро не «уничтожится» – оно будет присутствовать в суммарном графе. Следовательно, сумма различных фундаментальных циклов никогда не будет пустым графом, то есть фундаментальные циклы линейно независимы.

Покажем теперь, что любой квазицикл графа G является суммой фундаментальных циклов. Действительно, пусть H – такой квазицикл. Пусть $e_{i_1}, e_{i_2}, \ldots e_{i_t}$ – все ребра H, не принадлежащие T. Рассмотрим граф

$F = H \oplus Z_{i_1} \oplus Z_{i_2} \oplus \ldots \oplus Z_{i_t}$. Каждое из ребер e_{i_j}, $j = 1, \ldots, t$, входит ровно в два слагаемых этой суммы – в H и в Z_{i_j}. Следовательно, при сложении все эти ребра уничтожатся. Все остальные ребра, присутствующие в графах-слагаемых, принадлежат T. Значит, F – подграф графа T. Так как все слагаемые являются квазициклами, значит, F – тоже квазицикл. Но в T нет циклов, поэтому имеется единственная возможность: $F = O$, откуда получаем $H = Z_{i_1} \oplus Z_{i_2} \oplus \ldots \oplus Z_{i_t}$. □

Из этой теоремы следует, что размерность пространства циклов графа равна числу ребер, не входящих в его каркас. Так как каркас содержит $n - k$ ребер, где k – число компонент связности графа, то эта размерность равна $\nu(G) = m - n + k$. Это число называют *цикломатическим числом* графа.

2.4.4. Построение базы циклов

Базис пространства циклов графа коротко называют *базой циклов*. На основании теоремы 2.12 можно предложить достаточно простой способ построения базы циклов графа. Сначала находится какой-нибудь каркас, затем для каждого ребра, не принадлежащего каркасу, отыскивается тот единственный цикл, который это ребро образует с ребрами каркаса. Таким образом, любой алгоритм построения каркаса может быть использован для нахождения базы циклов.

Поиск в глубину особенно удобен благодаря основному свойству DFS-дерева (теорема 2.2) – каждое обратное ребро относительно этого дерева является продольным. Это означает, что из двух вершин такого ребра одна является предком другой в DFS-дереве. Каждое такое ребро в процессе поиска в глубину встретится дважды – один раз, когда активной вершиной будет предок, другой раз, когда ею будет потомок. В этом последнем случае искомый фундаментальный цикл состоит из рассматриваемого обратного ребра и участка пути в DFS-дереве, соединяющего эти две вершины. Но этот путь так или иначе запоминается в процессе обхода в глубину, так как он необходим для последующего возвращения. Если, например, для хранения открытых вершин используется стек, то вершины этого пути находятся в верхней части стека. В любом случае этот путь легко доступен и цикл находится без труда. Запишем процедуру построения фундаментальных циклов на базе алгоритма поиска в глубину с построением DFS-дерева (алгоритм 4). Переменная k – счетчик циклов, $C(k)$ – последовательность (список) вершин, составляющих цикл с номером k.

Алгоритм 6. *Построение базы циклов*

```
1   пометить все вершины как новые
2   k := 1
3   for x ∈ V do  if x новая  then  CycleBase(x)
```

Procedure CycleBase(a)

```
1   открыть вершину a
2   F(a) := a
3   x := a
4   while x открытая do
5           if имеется неисследованное ребро (x, y)
6                   then пометить ребро (x, y) как исследованное
7                           if вершина y новая
8                                   then открыть вершину y
9                                           F(y) := x
10                                          x := y
11                                  else NewCycle
12                  else закрыть вершину x
13                          x := F(x)
```

Procedure NewCycle

```
1   k := k + 1
2   Создать список C(k) из одного элемента x
3   z := x
4   repeat z := F(z)
5           добавить z к списку C(k)
6   until z = y
```

Хотя сам поиск в глубину выполняется за линейное от числа вершин и ребер время, решающее влияние на трудоемкость этого алгоритма оказывает необходимость запоминать встречающиеся циклы. Подсчитаем суммарную длину этих циклов для полного графа с n вершинами. DFS-дерево в этом случае является простым путем, относительно него будет $n - 2$ цикла длины 3, $n - 3$ цикла длины 4, ..., 1 цикл длины n. Сумма длин всех фундаментальных циклов будет равна

$$\sum_{i=1}^{n-2} i(n+1-i) = \frac{n^3 + 3n^2 - 16n + 12}{6}.$$

Таким образом, на некоторых графах число операций этого алгоритма будет величиной порядка n^3.

2.4.5. Рационализация

Приведенный алгоритм нетрудно модифицировать так, что он будет строить базу циклов с суммарной длиной, ограниченной сверху величиной порядка n^2 (и такой же будет оценка трудоемкости алгоритма). Рассмотрим в графе произвольную вершину x и пусть $y_1, y_2, \ldots, y_k$ – все ее предки в DFS-дереве, соединенные с x обратными ребрами. Положим также $y_{k+1} = x$. Обозначим через P_i для $i = 1, \ldots, k$ путь в DFS-дереве, соединяющий y_i и y_{i+1}. Описанный выше алгоритм выдает циклы вида $C_i = aP_iP_{i+1}\ldots P_ka$, $i = 1, \ldots, k$. Рассмотрим циклы $C'_i = aP_ia$, $i = 1, \ldots, k$. Так как $C_i = C'_i \oplus C'_{i+1} \oplus \ldots \oplus C'_k$, то совокупность всех таких циклов также образует базу циклов графа. Назовем эту систему циклов сокращенной. Алгоритм легко модифицировать так, чтобы вместо циклов C_i выдавались циклы C'_i – нужно только после обнаружения обратного ребра, ведущего от предка x к потомку y (строка 11), выписать вершины, содержащиеся в стеке, начиная с y и заканчивая следующей вершиной, смежной с x. Для эффективной проверки этой смежности удобно использовать матрицу смежности.

Оценим суммарную длину S циклов сокращенной системы. Предположим, что граф имеет n вершин и m ребер. Каждое обратное ребро принадлежит не более чем двум циклам сокращенной системы. Значит, суммарный вклад обратных ребер в S не превосходит $2m$.

Для каждого цикла из сокращенной системы назовем верхушкой этого цикла вершину цикла с наибольшим глубинным номером (это та вершина x, при исследовании окрестности которой был найден этот цикл). Очевидно, для каждого прямого ребра в сокращенной системе имеется не более одного цикла с данной верхушкой. Значит, число циклов, в которые входит данное прямое ребро, не превосходит числа вершин, лежащих в дереве выше этого ребра (то есть являющихся потомками вершин этого ребра). Тем более это число не превосходит числа всех вершин графа. Так как имеется не более чем $n - 1$ прямое ребро, то для суммарного вклада всех прямых ребер в S получаем верхнюю оценку n^2. Таким образом, $S < 2m + n^2 = O(n^2)$, то есть на порядок меньше максимальной суммарной длины системы фундаментальных циклов.

2.5. Эйлеровы циклы

Напомним, что эйлеровым циклом называется замкнутый маршрут, в котором каждое ребро графа встречается точно один раз. Согласно теореме 1.11 для существования такого маршрута в связном графе необходимо

и достаточно, чтобы степени всех вершин были четными. В этом разделе описывается алгоритм, который находит эйлеров цикл в заданном графе при условии, что это условия связности и четности степеней выполнены.

Этот алгоритм похож на алгоритм поиска в глубину: начиная с произвольно выбранной стартовой вершины a, строим путь, выбирая каждый раз для дальнейшего продвижения еще не пройденное ребро. Главное отличие от поиска в глубину состоит в том, что как пройденные помечаются именно ребра, а не вершины. Поэтому одна и та же вершина может посещаться несколько раз, но каждое ребро проходится не более одного раза, так что в полученном маршруте ребра не будут повторяться. Вершины пути накапливаются в стеке S. Через некоторое количество шагов неизбежно наступит тупик – все ребра, инцидентные активной (последней посещенной) вершине x, уже пройдены. Так как степени всех вершин графа четны, то в этот момент $x = a$ и пройденные ребра образуют цикл, но он может включать не все ребра графа. Для обнаружения еще не пройденных ребер возвращаемся по пройденному пути, перекладывая вершины из стека S в другой стек C, пока не встретим вершину x, которой инцидентно непройденное ребро. Так как граф связен, то такая вершина обязательно встретится. Тогда возобновляем движение вперед по непройденным ребрам, пока не дойдем до нового тупика и т.д. Процесс заканчивается, когда в очередном тупике обнаруживается, что S пуст. В этот момент в стеке C находится последовательность вершин эйлерова цикла.

Алгоритм 7. *Построение эйлерова цикла*

```
1  выбрать произвольно вершину  a
2  a ⇒ S
3  while S ≠ ∅ do
4          x := top(S)
5          if имеется непройденное ребро (x, y)
6                  then пометить ребро (x, y) как пройденное
7                          y ⇒ S
8                  else переместить вершину x из S в C
```

Для обоснования алгоритма заметим сначала, что первой в стек S помещается вершина a, и она будет последней перемещена из S в C. Следовательно, она будет последней вершиной в стеке C. Далее, как было отмечено выше, первый раз, когда обнаружится, что все инцидентные активной вершине ребра пройдены (то есть будет выполняться ветвь **else** в строке 8), активной будет стартовая вершина a. Значит, эта вершина будет первой перемещена из S в C. Итак, по окончании работы алгоритма в на-

чале и в конце последовательности вершин, содержащейся в стеке C, находится вершина a. Иначе говоря, если эта последовательность представляет маршрут (а далее будет показано, что это так и есть), то он замкнут.

Далее отметим, что в конечном итоге каждое ребро будет пройдено. Действительно, допустим, что в момент окончания работы алгоритма имеются еще не пройденные ребра. Так как граф связен, то должно существовать хотя бы одно непройденное ребро, инцидентное посещенной вершине. Но тогда эта вершина не могла быть удалена из стека S и S не мог стать пустым.

Будем говорить, что ребро (x, y) представлено в стеке (S или C), если в какой-то момент работы алгоритма в стеке рядом находятся вершины x и y. Ясно, что каждое ребро графа будет представлено в стеке S и что каждые две вершины, расположенные рядом в этом стеке, образуют ребро. Допустим, в какой-то момент из стека S в стек C перемещается вершина x, а непосредственно под ней в стеке S находится вершина y. Возможно, что вершина y будет перемещена из S в C при следующем повторении цикла **while**, тогда ребро (x, y) будет представлено в стеке C. Другая возможность – между перемещением вершины x и следующим перемещением, то есть следующим выполнением ветви **else**, будет несколько раз выполнена ветвь **then** (строки 6, 7). Это означает, что будет пройдена некоторая последовательность ребер, начинающаяся в вершине y. Ввиду четности степеней эта последовательность может закончиться только в вершине y. Значит, и в этом случае следующей за вершиной x будет перемещена из S в C вершина y. В любом случае ребро (x, y) будет представлено в стеке C. Из этого рассуждения видно, что последовательность вершин в стеке C является маршрутом и что каждое ребро графа в конечном итоге будет содержаться в этом маршруте, причем один раз.

При каждом повторении цикла **while** в рассмотренном алгоритме либо проходится одно ребро, либо одна вершина перемещается из S в C. Последнее можно трактовать как прохождение уже пройденного однажды ребра в обратном направлении. Каждое ребро в каждом направлении будет пройдено один раз, поэтому общая трудоемкость этого алгоритма оценивается как $O(m)$. Необходимо только оговориться, что этот вывод, как и аналогичные заключения об алгоритмах обхода в первых разделах этой главы, справедлив лишь при определенных предположениях о том, как задан граф. Способ задания должен обеспечить возможность быстрого просмотра множества ребер, инцидентных данной вершине. Подходящим является, например, задание графа списками инцидентности, в которых для каждой вершины перечисляются инцидентные ей ребра. Необходимо

также иметь возможность быстро пометить ребро как пройденное или проверить, пройдено ли данное ребро. Для этого подходящей структурой может служить характеристический массив на множестве ребер.

2.6. Гамильтоновы циклы

Гамильтоновым циклом (*путем*) называют простой цикл (путь), содержащий все вершины графа. В графе, изображенном на рис. 2.8 слева, гамильтоновым циклом является, например, последовательность 1, 2, 3, 5, 4, 1. В графе, изображенном в центре, нет гамильтоновых циклов, но есть гамильтоновы пути, например, 2, 1, 3, 5, 4. В правом графе нет и гамильтоновых путей.

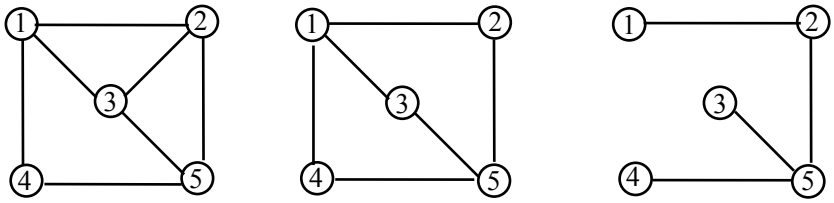


Рис. 2.8

Внешне определение гамильтонова цикла похоже на определение эйлерова цикла. Однако имеется кардинальное различие в сложности решения соответствующих задач распознавания и построения. Мы видели, что имеется достаточно простой критерий существования эйлерова цикла и эффективный алгоритм его построения. Для гамильтоновых же циклов (и путей) неизвестно никаких просто проверяемых необходимых и достаточных условий их существования, а все известные алгоритмы требуют для некоторых графов перебора большого числа вариантов.

Гамильтонов цикл представляет собой, с комбинаторной точки зрения, просто перестановку вершин графа. При этом в качестве начальной вершины цикла можно выбрать любую вершину, так что можно рассматривать перестановки с фиксированным первым элементом. Самый бесхитростный план поиска гамильтонова цикла состоит в последовательном рассмотрении всех этих перестановок и проверке для каждой из них, представляет ли она цикл в данном графе. Такой способ действий уже при не очень большом числе вершин становится практически неосуществимым ввиду быстрого роста числа перестановок – имеется $(n-1)!$ перестановок из n элементов с фиксированным первым элементом.

Более рациональный подход состоит в рассмотрении всевозможных простых путей, начинающихся в произвольно выбранной стартовой вершине a, до тех пор, пока не будет обнаружен гамильтонов цикл или все

возможные пути не будут исследованы. По сути дела, речь тоже идет о переборе перестановок, но значительно сокращенном – если, например, вершина b не смежна с вершиной a, то все $(n-2)!$ перестановок, у которых на первом месте стоит a, а на втором b, не рассматриваются.

Рассмотрим этот алгоритм подробнее. Будем считать, что граф задан окрестностями вершин: для каждой вершины x задано множество вершин, смежных с x. На каждом шаге алгоритма имеется уже построенный отрезок пути, он хранится в стеке *PATH*. Для каждой вершины x, входящей в *PATH*, хранится множество $N(x)$ всех вершин, смежных с x, которые еще не рассматривались в качестве возможных продолжений пути из вершины x. Когда вершина x добавляется к пути, множество $N(x)$ полагается равным $V(x)$. В дальнейшем рассмотренные вершины удаляются из этого множества. Очередной шаг состоит в исследовании окрестности последней вершины x пути *PATH*. Если $N(x) \neq \varnothing$ и в $N(x)$ имеются вершины, не принадлежащие пути, то одна из таких вершин добавляется к пути. В противном случае вершина x исключается из стека. Когда после добавления к пути очередной вершины оказывается, что путь содержит все вершины графа, остается проверить, смежны ли первая и последняя вершины пути, и при утвердительном ответе выдать очередной гамильтонов цикл.

Алгоритм 8. *Поиск гамильтоновых циклов*

```
1   выбрать произвольно вершину a
2   a ⇒ PATH
3   N(a) := V(a)
4   while PATH ≠ ∅ do
5           x := top(PATH)
6           if N(x) ≠ ∅
7                   then взять y ∈ N(x)
8                           N(x) := N(x) – y
9                           if вершина y не находится в PATH
10                                  then  y ⇒ PATH
11                                        N(y) := V(y)
12                                        if PATH содержит все вершины
13                                                then  if x смежна с a
14                                                        then выдать цикл
15                  else удалить вершину x из PATH
```

Этот алгоритм очень похож на алгоритм поиска в глубину и отличается от него по существу только тем, что открытая вершина, когда вся ее окрестность исследована, не закрывается, а опять становится новой (ис-

ключается из стека). В начале все вершины новые. Процесс заканчивается, когда все вершины опять станут новыми. На самом деле это и есть поиск в глубину, только не в самом графе, а в дереве путей. Вершинами этого дерева являются всевозможные простые пути, начинающиеся в вершине a, а ребро дерева соединяет два пути, один из которых получается из другого добавлением одной вершины в конце. На рис. 2.9 показаны граф и его дерево путей из вершины 1.

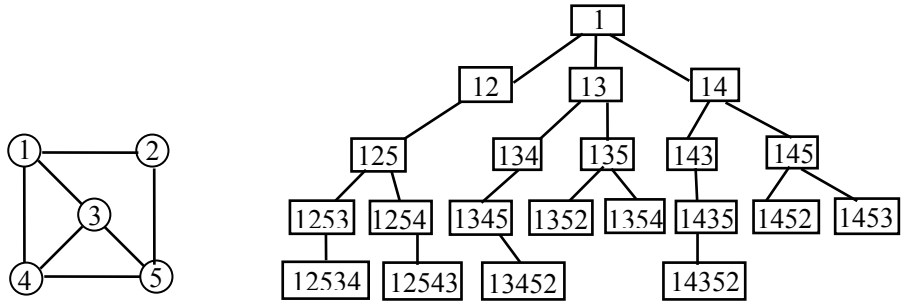


Рис. 2.9

В худшем случае время работы этого алгоритма тоже растет с факториальной скоростью. Например, для графа $K_{n-1} + K_1$ (граф с двумя компонентами связности, одна из которых – полный граф с $n - 1$ вершиной, другая – изолированная вершина), если в качестве стартовой выбрана не изолированная вершина, то будут рассмотрены все $(n - 1)!$ простых путей длины $n - 2$ в большой компоненте. Вместе с тем, если перед поиском гамильтонова цикла исходный граф проверить на связность, то ответ будет получен быстро. Можно пойти дальше и при обходе дерева путей проверять на связность каждый встречающийся «остаточный граф», то есть граф, получающийся из исходного удалением всех вершин рассматриваемого пути. Если этот граф несвязен, то этот путь не может быть продолжен до гамильтонова пути. Поэтому можно не исследовать соответствующую ветвь дерева, а вернуться к рассмотрению более короткого пути, удалив последнюю вершину (то есть сделать «шаг назад» в поиске в глубину). Можно пойти еще дальше и заметить, что если некоторая вершина x DFS-дерева с корнем a является развилкой, то есть имеет не менее двух сыновей, то в подграфе исходного графа, полученном удалением всех предков этой вершины, кроме нее самой, она будет шарниром. Поэтому путь от a до x в DFS-дереве не может быть продолжен до гамильтонова пути. Эти соображения приводят к такой модификации алгоритма: обходим граф поиском в глубину с построением DFS-дерева, затем находим в

этом дереве самую нижнюю развилку (развилку с наименьшим глубинным номером). Если ни одной развилки нет, то само DFS-дерево представляет собой гамильтонов путь и остается проверить наличие ребра, соединяющего начало и конец пути. Если же x – развилка, то возвращаемся из x в предшествующую вершину пути, помечаем все вершины, кроме собственных предков вершины x, как непосещенные и возобновляем поиск в глубину с этого места.

Рассмотрим другой алгоритм, выясняющий существование гамильтонова цикла, идейно близкий к поиску в ширину и имеющий не столь быстро (хотя все же быстро) растущую оценку трудоемкости.

Пусть граф G задан матрицей смежности $A = \|A(i, j)\|$. Выберем произвольно стартовую вершину a и определим для каждого $k = 0, 1, \ldots, n - 2$ функцию $H_k(x, X)$, где значениями переменной x являются вершины, отличные от a, а значениями переменной X – k-элементные подмножества множества $VG - \{a\}$, причем вершина x не должна принадлежать множеству X. Эти функции определяются так: полагаем $H_k(x, X) = 1$, если существует простой путь длины $k + 1$ из вершины a в вершину x, проходящий только через вершины из множества X, и $H_k(x, X) = 0$, если такого пути не существует. Тогда

$$H_0(x, \varnothing) = A(a, x) \text{ для всех } x,$$

а для $k > 0$

$$H_k(x, X) = \bigvee_{y \in X} H_{k-1}(y, X - y) A(y, x).$$

Таким образом, зная все значения функции H_{k-1}, мы можем вычислить все значения функции H_k, причем для вычисления одного значения требуется выполнить $2k - 1$ логических операций. Общее время на вычисление всех этих функций, как легко подсчитать, составит $O(n^2 2^n)$. Остается только для всех x, для которых $H_{n-2}(x, X) = 1$ (X в этом случае определяется однозначно), выяснить, чему равно $A(x, a)$ – если хотя бы в одном случае это равно 1, то гамильтонов цикл существует. Очевидный недостаток этого алгоритма – необходимость хранения большого количества промежуточной информации.

Задачи и упражнения

1. Сколько различных DFS-деревьев можно построить для полного графа K_n?

2. Докажите, что в двусвязном графе для любых трех вершин a, b, c существует путь, соединяющий a и b, не проходящий через c.

3. Сколько различных квазициклов имеется в графе с n вершинами, m ребрами и k компонентами связности?

4. Докажите, что в графе Q_n при любом $n \geq 2$ существует гамильтонов цикл.

5. Разработайте алгоритм с трудоемкостью $O(m + n)$ для нахождения всех перешейков графа.

6. Разработайте алгоритм, проверяющий, является ли данный граф деревом.

7. Разработайте алгоритм с трудоемкостью $O(m + n)$, проверяющий, является ли данный граф двудольным, а при отрицательном ответе находящий в нем нечетный цикл.

8. Задача об одностороннем движении: требуется каждому ребру заданного неориентированного графа присвоить ориентацию таким образом, чтобы полученный ориентированный граф был сильно связным. Докажите, что это возможно тогда и только тогда, когда в графе нет перешейков. Постройте алгоритм для нахождения такой ориентации.

9. Докажите, что в каждом связном неориентированном графе существует замкнутый маршрут, проходящий по каждому ребру точно один раз в каждом направлении. Разработайте алгоритм, который находит такой маршрут за время $O(m)$.

10. Разработайте алгоритм с трудоемкостью $O(m)$, находящий в данном неориентированном графе кратчайший цикл, проходящий через заданную вершину.

11. Разработайте алгоритм с трудоемкостью $O(m)$, находящий в данном неориентированном графе все циклы заданной длины k, проходящие через заданную вершину.

Глава 3. Экстремальные задачи на графах

Во многих задачах на графах требуется найти какой-нибудь максимум или минимум, например наибольший подграф с заданным свойством, или разбиение графа на наименьшее число частей, удовлетворяющих каким-то условиям, и т.д. В этой главе рассматриваются несколько задач такого рода, считающихся классическими в теории графов. Некоторые из них известны как NP-трудные, для них рассматриваются переборные алгоритмы, приемы рационализации, эвристики и приближенные алгоритмы. Для других задач излагаются известные эффективные алгоритмы. В последнем разделе затрагиваются вопросы теории т.н. жадных (градиентных) алгоритмов.

3.1. Независимые множества, клики, вершинные покрытия

3.1.1. Три задачи

Независимым множеством вершин графа называется любое множество попарно не смежных вершин, то есть множество вершин, порождающее пустой подграф. Независимое множество называется *максимальным*, если оно не является собственным подмножеством другого независимого множества, и *наибольшим*, если оно содержит наибольшее количество вершин. Число вершин в наибольшем независимом множестве графа G обозначается через $\alpha(G)$ и называется *числом независимости* графа. Задача о независимом множестве состоит в нахождении наибольшего независимого множества.

Кликой графа называется множество вершин, порождающее полный подграф, то есть множество вершин, каждые две из которых смежны. Число вершин в клике наибольшего размера называется *кликовым числом* графа и обозначается через $\omega(G)$. Очевидно, задача о независимом множестве преобразуется в задачу о клике и наоборот простым переходом от данного графа G к дополнительному графу $\overline{G}$, так что $\alpha(G) = \omega(\overline{G})$.

Вершинное покрытие графа – это такое множество вершин, что каждое ребро графа инцидентно хотя бы одной из этих вершин. Наименьшее число вершин в вершинном покрытии графа G обозначается через $\beta(G)$ и называется *числом вершинного покрытия* графа. В графе на рис. 3.1 наибольшим независимым множеством является множество {1, 3, 4, 7}, наи-

большей кликой – множество {2, 3, 5, 6}, наименьшим вершинным покрытием – множество {2, 5, 6}.

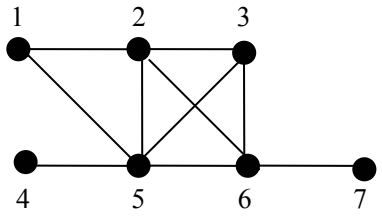


Рис. 3.1

Между задачами о независимом множестве и о вершинном покрытии тоже имеется простая связь благодаря следующему факту.

Теорема 3.1. *Подмножество U множества вершин графа G является вершинным покрытием тогда и только тогда, когда* $\overline{U} = VG - U$ *– независимое множество.*

Доказательство. Если U – вершинное покрытие, то всякое ребро содержит хотя бы одну вершину из множества U и, значит, нет ни одного ребра, соединяющего две вершины из множества $\overline{U}$. Следовательно, $\overline{U}$ – независимое множество. Обратно, если $\overline{U}$ – независимое множество, то нет ребер, соединяющих вершины из $\overline{U}$ и, значит, у каждого ребра одна или обе вершины принадлежат множеству U. Следовательно, U – вершинное покрытие. □

Из этой теоремы следует, что $\alpha(G) + \beta(G) = n$ для любого графа G с n вершинами.

Таким образом, все три задачи тесно связаны друг с другом, так что достаточно научиться решать одну из них, и мы будем уметь решать остальные две. Вместе с тем известно, что эти задачи NP-полны. Для таких задач не известно эффективных алгоритмов, а накопленный к настоящему времени опыт делает правдоподобным предположение о том, что таких алгоритмов и не существует. Тем не менее, алгоритмы для подобных задач разрабатывались и продолжают разрабатываться, и в некоторых случаях они могут быть полезны. Все эти алгоритмы в той или иной форме осуществляют перебор вариантов (число которых может быть очень большим). Далее рассмотрим один из способов такого перебора для задачи о независимом множестве.

3.1.2. Стратегия перебора для задачи о независимом множестве

Пусть G – граф, в котором требуется найти наибольшее независимое множество. Выберем в нем произвольную вершину a. Обозначим через G_1 подграф, получающийся удалением из графа G вершины a, то есть $G_1 = G - a$, а через G_2 подграф, получающийся удалением из G всех вершин, смежных с a. На рис. 3.2 показаны графы G_1 и G_2, получающиеся из графа G, изображенного на рис. 3.1, при $a = 1$.

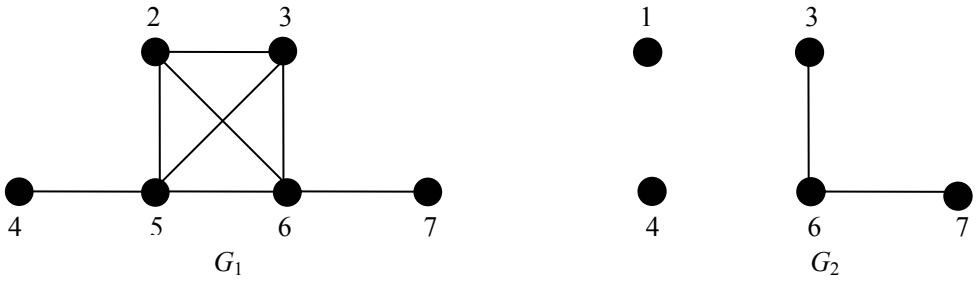


Рис. 3.2

Пусть X – какое-нибудь независимое множество графа G. Если оно не содержит вершины a, то оно является независимым множеством графа G_1. Если же $a \in X$, то никакая вершина, смежная с a, не принадлежит X. В этом случае множество X является независимым множеством графа G_2. Заметим, что в графе G_1 на одну вершину меньше, чем в исходном графе G. Если вершина a не является изолированной, то и в графе G_2 вершин меньше, чем в графе G. Таким образом, задача о независимом множестве для графа G свелась к решению той же задачи для двух графов меньшего размера. Это приводит к рекуррентному соотношению для числа независимости:

$$\alpha(G) = \max \{ \alpha(G_1), \alpha(G_2)\}$$

и к рекурсивному алгоритму для нахождения наибольшего независимого множества графа G: найдем наибольшее независимое множество X_1 графа G_1, затем наибольшее независимое множество X_2 графа G_2 и выберем большее из этих двух множеств. В целом процесс решения задачи при этом можно рассматривать как исчерпывающий поиск в возникающем дереве подзадач. Чтобы не путать вершины дерева и вершины графа, вершины дерева будем называть узлами. Узел, не являющийся листом, называется внутренним узлом. Каждому внутреннему узлу дерева соответствует некоторый граф H и некоторая вершина этого графа x. Вершину x можно выбирать произвольно, но она не должна быть изолированной

вершиной графа H. Внутренний узел имеет двух сыновей – левого и правого. Левому сыну соответствует подграф графа H, получаемый удалением вершины x, а правому – подграф, получаемый удалением всех вершин, смежных с x. Корню дерева соответствует исходный граф. Листьям соответствуют подграфы, не имеющие ребер, то есть подграфы, у которых все вершины изолированные. Множества вершин этих подграфов – это независимые множества исходного графа.

Для нахождения наибольшего независимого множества не обязательно строить все дерево полностью, а достаточно обойти его в том или ином порядке, запоминая на каждом шаге только небольшую часть информации об устройстве этого дерева. Можно, например, применить поиск в глубину для обхода дерева: сначала пройти от корня до некоторого листа, затем вернуться к предку этого листа и искать следующий лист, и т.д.

Для одного и того же графа могут получиться разные деревья в зависимости от того, как выбирается активная вершина x в каждом узле дерева. Может быть различным и число листьев в этих деревьях, а значит, и трудоемкость алгоритма, основанного на обходе дерева. Однако в любом случае листьев в дереве будет не меньше, чем число максимальных независимых множеств у графа, так как каждое из этих множеств будет соответствовать некоторому листу. Так, для графа pK_2, то есть графа, состоящего из p компонент связности, каждая из которых изоморфна графу K_2, в дереве подзадач будет в лучшем случае 2^p листьев.

3.1.3. Рационализация

Известны различные приемы сокращения перебора при использовании описанной стратегии исчерпывающего поиска. Один из них основан на следующем наблюдении. Допустим, в графе G, для которого нужно найти наибольшее независимое множество, имеются две вершины a и b такие, что каждая вершина, отличная от b и смежная с вершиной a, смежна и с вершиной b. Иначе говоря, $V(a) - \{b\} \subseteq V(b)$. Будем говорить в этом случае, что вершина b поглощает вершину a. Если при этом вершины a и b смежны, то скажем, что вершина b смежно поглощает вершину a. Вершину b в этом случае назовем смежно поглощающей. Например, в графе, изображенном на рис. 3.1, вершина 2 смежно поглощает вершины 1 и 3. Вершины 5 и 6 в этом графе тоже являются смежно поглощающими.

Лемма 3.2. *Если вершина b является смежно поглощающей в графе G, то* $\alpha(G - b) = \alpha(G)$.

Доказательство. Допустим, вершина b смежно поглощает вершину a в графе G. Пусть X – наибольшее независимое множество графа G. Если X

не содержит вершину b, то оно является наибольшим независимым множеством и в графе $G - b$, так что в этом случае $\alpha(G - b) = \alpha(G)$. Предположим, что множество X содержит вершину b. Тогда ни одна вершина из множества $V(b)$ не принадлежит X. Значит, X не содержит вершину a и ни одну вершину из множества $V(a)$. Но тогда множество $(X - \{b\}) \cup \{a\}$ тоже будет независимым, причем оно целиком содержится в графе $G - b$, а число элементов в нем такое же, как в множестве X. Значит, и в этом случае $\alpha(G - b) = \alpha(G)$. □

Итак, если мы удалим из графа смежно поглощающую вершину b, то получим граф с тем же числом независимости. Так как новый граф является порожденным подграфом исходного графа G, то каждое наибольшее независимое множество нового графа будет наибольшим независимым множеством исходного. Этот прием называется «сжатием по включению». Исследование применимости и применение операции сжатия по включению к каждому встречающемуся подграфу требуют, конечно, дополнительных расходов времени, но могут привести к существенному сокращению дерева подзадач. Для некоторых графов задача о независимом множестве может быть решена с помощью одних только сжатий по включению. Таков, например, граф pK_2, и вообще любой лес. Действительно, любая вершина, смежная с листом, поглощает этот лист. Рассмотрим более широкий класс графов, для которых этот прием эффективен.

3.1.4. Хордальные графы

Граф называется *хордальным* (или *триангулированным*), если в нем нет порожденных простых циклов длины ≥ 4. Иначе говоря, в хордальном графе для каждого простого цикла длины 4 или больше имеется хотя бы одна хорда – ребро, не принадлежащее циклу, но соединяющее две вершины цикла.

Теорема 3.3. *В любом непустом хордальном графе имеется смежно поглощающая вершина.*

Доказательство. Пусть G – непустой граф, в котором нет смежно поглощающих вершин. Докажем, что G не хордальный. Рассмотрим в нем простой путь $P = x_1, x_2, \ldots, x_k$ наибольшей длины, не имеющий хорд, то есть ребер, соединяющих две вершины пути и не принадлежащих пути. Так как граф непустой, то $k \geq 2$. Рассмотрим вершину x_k. Так как она не поглощает вершину x_{k-1}, то существует вершина $y \neq x_{k-1}$, смежная с вершиной x_k, но не смежная с x_{k-1}. Вершина y не принадлежит пути P, так как иначе ребро (x_k, y) было бы хордой этого пути. Следовательно, последова-

тельность $P' = x_1, x_2, \ldots, x_k, y$ является простым путем. Но длина этого пути больше, чем длина пути P, поэтому, в силу выбора пути P, у пути P' должна существовать хорда. Такой хордой может быть только ребро вида (y, x_i), где $i \le k - 2$. Пусть i – наибольшее, при котором ребро (y, x_i) является хордой пути P'. Тогда последовательность $y, x_i, x_{i+1}, \ldots, x_k, y$ является циклом без хорд длины не менее 4. □

Итак, для хордального графа наибольшее независимое множество можно найти с помощью одних только сжатий по включению. Нужно только находить смежно поглощающие вершины и удалять их из графа до тех пор, пока оставшийся граф не станет пустым. Множество оставшихся вершин и является наибольшим независимым множеством.

3.1.5. Эвристики для задачи о независимом множестве

Поиск в дереве вариантов неэффективен в общем случае, а приемы сокращения перебора, подобные описанному выше сжатию по включению, применимы далеко не ко всем графам. Одним из выходов из этого положения является применение так называемых эвристических алгоритмов, или эвристик. Так называются алгоритмы, основанные на каких-нибудь интуитивных соображениях, которые, как кажется, ведут к получению хорошего решения. Такие алгоритмы могут иногда не давать вообще никакого решения или давать решение, далекое от оптимального. Но они, как правило, очень быстро работают и иногда (а может быть, и очень часто) дают решение, близкое к оптимальному или приемлемое для практики. Рассмотрим две простые эвристики для задачи о независимом множестве.

Одна из эвристических идей состоит в том, чтобы рассмотреть только один путь от корня до листа в дереве вариантов в надежде, что этому листу соответствует достаточно большое независимое множество. Для выбора этого единственного пути могут применяться разные соображения. В дереве вариантов, описанном выше, у каждого внутреннего узла имеются два сына. Одному из них соответствует подграф, получающийся удалением некоторой произвольно выбранной вершины a, а другому – подграф, получающийся удалением окрестности этой вершины. Чтобы вместо дерева получился один путь, достаточно каждый раз выполнять какую-нибудь одну из этих двух операций. Рассмотрим оба варианта.

Допустим, мы решили каждый раз удалять выбранную вершину. Эти удаления производятся до тех пор, пока не останется граф без ребер, то есть независимое множество. Оно и принимается в качестве решения задачи. Для полного описания алгоритма необходимо еще сформулировать

правило выбора активной вершины a. Мы хотим получить граф без ребер, в котором было бы как можно больше вершин. Чем меньше вершин будет удалено, тем больше их останется. Значит, цель – как можно быстрее удалить все ребра. Кажется, мы будем двигаться в нужном направлении, если на каждом шаге будем удалять наибольшее возможное на этом шаге число ребер. Это означает, что в качестве активной вершины всегда нужно выбирать вершину наибольшей степени. Алгоритмы такого типа называются жадными или градиентными. К сожалению, как будет показано дальше, оптимальный выбор на каждом шаге не гарантирует получения оптимального решения в конечном итоге.

Другой вариант – каждый раз удалять окрестность активной вершины a. Это опять повторяется до тех пор, пока оставшиеся вершины не будут образовывать независимого множества. Удаление окрестности вершины a равносильно тому, что сама эта вершина включается в независимое множество, которое будет получено в качестве ответа. Так как мы хотим получить в итоге как можно большее независимое множество, естественно постараться удалять на каждом шаге как можно меньше вершин. Это означает, что в качестве активной вершины всегда нужно выбирать вершину наименьшей степени. Получается еще один вариант жадного алгоритма.

Имеется немало графов, для которых каждая из этих эвристик дает близкое к оптимальному, а иногда и оптимальное решение. Но, как это обычно бывает с эвристическими алгоритмами, можно найти примеры графов, для которых найденные решения будут весьма далеки от оптимальных. Рассмотрим граф G_k, у которого множество вершин V состоит из трех частей: $V = A \cup B_1 \cup B_2$, $|A| = |B_1| = |B_2| = k$, причем A является независимым множеством, каждое из множеств B_1, B_2 – кликой и каждая вершина из множества A смежна с каждой вершиной из множества $B_1 \cup B_2$. С помощью операций суммы и соединения графов (раздел 1.3) этот граф можно представить формулой $G_k = (K_k + K_k) \circ O_k$. Степень каждой вершины из множества A в этом графе равна $2k$, а степень каждой вершины из множества $B_1 \cup B_2$ равна $2k - 1$. Первый алгоритм, выбирающий вершину наибольшей степени, будет удалять вершины из множества A до тех пор, пока не удалит их все. После этого останется граф, состоящий из двух клик, и в конечном итоге будет получено независимое множество из двух вершин. Второй алгоритм на первом шаге возьмет в качестве активной одну из вершин множества $B_1 \cup B_2$ и удалит всю ее окрестность. В результате получится граф, состоящих из этой вершины и клики, а после второго шага получится независимое множество, состоящее опять из двух вершин. Итак, при применении к этому графу любой из двух эвристик по-

лучается независимое множество из двух вершин. В то же время в графе имеется независимое множество A мощности k.

3.1.6. Приближенный алгоритм для задачи о вершинном покрытии

Иногда для алгоритма, не гарантирующего точного решения, удается получить оценку степени приближения, то есть отклонения получаемого решения от точного. В таком случае говорят о приближенном алгоритме. Рассмотрим один простой приближенный алгоритм для задачи о вершинном покрытии.

Работа алгоритма начинается с создания пустого множества X и состоит в выполнении однотипных шагов, в результате каждого из которых к множеству X добавляются некоторые вершины. Допустим, перед очередным шагом имеется некоторое множество вершин X. Если оно покрывает все ребра (то есть каждое ребро инцидентно одной из этих вершин), то процесс заканчивается и множество X принимается в качестве искомого вершинного покрытия. В противном случае выбирается какое-нибудь непокрытое ребро (a, b) и вершины a и b добавляются к множеству X.

Для полного описания алгоритма нужно бы еще сформулировать правило выбора ребра (a, b). Однако для оценки степени приближения, которая будет сейчас получена, это не имеет значения. Можно считать, что какое-то правило выбрано.

Обозначим через $\beta'(G)$ мощность вершинного покрытия, которое получится при применении этого алгоритма к графу G, и докажем, что $\beta'(G) \le 2\beta(G)$. Иначе говоря, полученное с помощью этого алгоритма решение не более чем в два раза отличается от оптимального.

Действительно, допустим, что до окончания работы алгоритм выполняет k шагов, добавляя к множеству X вершины ребер (a_1, b_1), …, (a_k, b_k). Тогда $\beta'(G) = 2k$. Никакие два из этих k ребер не имеют общей вершины. Значит, чтобы покрыть все эти ребра, нужно не меньше k вершин. Следовательно, $\beta(G) \ge k$ и $\beta'(G) \le 2\beta(G)$.

3.1.7. Перебор максимальных независимых множеств

Основной недостаток изложенного выше способа организации перебора состоит в том, что при нем часто рассматриваются не только максимальные независимые множества. Это означает, что делается заведомо лишняя работа, так как наибольшее независимое множество находится, конечно, среди максимальных. Кроме того в некоторых случаях бывает необходимо знать все максимальные независимые множества. Рассмотрим алгоритм, который строит все максимальные и только максимальные не-

зависимые множества графа.

Предположим, что вершинами заданного графа G являются числа 1, 2, ..., n. Рассматривая любое подмножество множества вершин, будем выписывать его элементы в порядке возрастания. Лексикографический порядок на множестве получающихся таким образом кортежей порождает линейный порядок на множестве всех подмножеств множества вершин, который тоже будем называть лексикографическим. Например, множество {2, 5, 7, 9} предшествует в этом порядке множеству {2, 5, 8}, а множество {2, 5, 7, 10} занимает промежуточное положение между этими двумя.

Нетрудно найти лексикографически первое максимальное независимое множество: нужно на каждом шаге брать наименьшую из оставшихся вершин, добавлять ее к построенному независимому множеству, а все смежные с ней вершины удалять из графа. Если граф задан списками смежности, то это построение выполняется за время $O(m)$.

Допустим теперь, что $U \subseteq VG$, G' – подграф графа G, порожденный множеством U, и пусть имеется список L всех максимальных независимых множеств графа G. Тогда однократным просмотром списка L можно получить список L' всех максимальных независимых множеств графа G'. Это основано на следующих очевидных утверждениях:

1) каждое максимальное независимое множество графа G' содержится в некотором максимальном независимом множестве графа G;

2) для каждого максимального независимого множества N графа G' имеется точно одно максимальное независимое множество M графа G такое, что $N \subseteq M$ и $M - N$ – лексикографически первое среди максимальных независимых множеств подграфа, порожденного множеством $VG - (U \cup V(N))$ (последнее утверждение верно и в том случае, если $VG - (U \cup V(N)) = \varnothing$, если считать, что пустое множество является максимальным независимым множеством в графе с пустым множеством вершин).

Будем теперь рассматривать множества из L одно за другим и пусть M – очередное такое множество. Положим $N = M \cap U$. Если N не является максимальным независимым множеством графа G', то переходим к следующему элементу списка L. Если же N – максимальное независимое множество в G', то рассматриваем множество $M - N$. Если оно является лексикографически первым среди максимальных независимых множеств подграфа, порожденного множеством $VG - (U \cup V(N))$, то включаем N в список L'.

Выберем в графе G произвольную вершину a и пусть A – множество всех вершин графа, смежных с a (окрестность вершины a), B – множество всех вершин, не смежных с a и отличных от a. Обозначим через G_1 подграф, получающийся удалением из графа G вершины a, а через G_2 подграф, получающийся удалением из G всех вершин множества $A \cup \{a\}$. Иначе говоря, G_1 – подграф графа G, порожденный множеством $A \cup B$, а G_2 – подграф, порожденный множеством B.

Допустим, что имеется список L_1 всех максимальных независимых множеств графа G_1. На основании вышеизложенного можно предложить следующую процедуру получения списка L всех максимальных независимых множеств графа G.

1. Взять очередной элемент M списка L_1.

2. Если $M \subseteq B$, то добавить к списку L множество $M \cup \{a\}$ и перейти к 1, иначе добавить к списку L множество M.

3. Если множество $M \cap B$ не является максимальным независимым множеством в графе G_2, то перейти к 1.

4. Если множество $N = M \cap A$ является лексикографически первым максимальным независимым множеством подграфа, порожденного множеством $A - V(M \cap B)$, то добавить к списку L множество $M \cup \{a\}$.

5. Если список L_1 не исчерпан, перейти к 1.

Начиная с одновершинного графа (у которого список максимальных независимых множеств состоит из одного элемента), добавляя последовательно по одной вершине, получаем последовательность графов G_1, G_2, …, $G_n = G$. Применяя для каждого $i = 1, \ldots, n - 1$ описанный алгоритм для построения списка всех максимальных независимых множеств графа G_{i+1} по такому списку для графа G_i, в конце концов получим список всех максимальных независимых множеств графа G. По сути дела, этот алгоритм представляет собой поиск в ширину в дереве вариантов. Для того, чтобы не хранить всех получающихся списков, его можно преобразовать в поиск в глубину. Заметим, что приведенная процедура для каждого максимального независимого множества графа G_i находит одно или два максимальных независимых множества графа G_{i+1}. Одно из этих новых множеств рассматривается на следующем шаге, другое, если оно есть, запоминается в стеке.

Изложенный алгоритм можно применить для отыскания наибольших независимых множеств в графах, про которые известно, что в них мало максимальных независимых множеств. Одним из классов графов с таким свойством является класс всех графов, не содержащих $2K_2$ в качестве по-

рожденного подграфа. Известно, что в графе с m ребрами из этого класса число максимальных независимых множеств не превосходит $m + 1$.

3.2. Раскраски

3.2.1. Раскраска вершин

Раскраской вершин графа называется назначение цветов его вершинам. Обычно цвета – это числа 1, 2, ..., k. Тогда раскраска является функцией, определенной на множестве вершин графа и принимающей значения во множестве $\{1, 2, \ldots, k\}$. Раскраску можно также рассматривать как разбиение множества вершин $V = V_1 \cup V_2 \cup \ldots \cup V_k$, где V_i – множество вершин цвета i. Множества V_i называют цветными классами. Раскраска называется *правильной*, если каждый цветной класс является независимым множеством. Иначе говоря, в правильной раскраске любые две смежные вершины должны иметь разные цвета. Задача о раскраске состоит в нахождении правильной раскраски данного графа G в наименьшее число цветов. Это число называется *хроматическим числом* графа и обозначается $\chi(G)$.

В правильной раскраске полного графа K_n все вершины должны иметь разные цвета, поэтому $\chi(K_n) = n$. Если в каком-нибудь графе имеется полный подграф с k вершинами, то для раскраски этого подграфа необходимо k цветов. Отсюда следует, что для любого графа выполняется неравенство

$$\chi(G) \geq \omega(G). \tag{3.1}$$

Однако хроматическое число может быть и строго больше кликового числа. Например, для цикла длины 5 $\omega(C_5) = 2$, а $\chi(C_5) = 3$. Другой пример показан на рис. 3.3. На нем изображен граф, вершины которого раскрашены в 4 цвета (цвета вершин показаны в скобках). Нетрудно проверить, что трех цветов для правильной раскраски этого графа недостаточно. Следовательно, его хроматическое число равно 4. Очевидно также, что кликовое число этого графа равно 3.

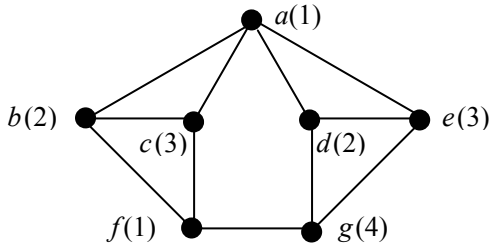


Рис. 3.3

Очевидно, $\chi(G) = 1$ тогда и только тогда, когда G – пустой граф. Нетрудно охарактеризовать и графы с хроматическим числом 2 (точнее, не больше 2). По определению, это такие графы, у которых множество вершин можно разбить на два независимых множества. Но это совпадает с определением двудольного графа. Поэтому двудольные графы называют еще *бихроматическими*. Согласно теореме 1.14, граф является бихроматическим тогда и только тогда, когда в нем нет циклов нечетной длины.

Для графов с хроматическим числом 3 такого простого описания неизвестно. Неизвестно и простых алгоритмов, проверяющих, можно ли данный граф раскрасить в 3 цвета. Более того, задача такой проверки (вообще, задача проверки возможности раскрасить граф в k цветов при любом фиксированном $k \geq 3$) является NP-полной.

3.2.2. Переборный алгоритм для раскраски

Рассмотрим алгоритм решения задачи о раскраске, похожий на описанный выше алгоритм для задачи о независимом множестве. Сходство заключается в том, что задача для данного графа сводится к той же задаче для двух других графов. Поэтому снова возникает дерево вариантов, обход которого позволяет найти решение. Но есть и одно существенное различие, состоящее в том, что теперь два новых графа не будут подграфами исходного графа.

Выберем в данном графе G две несмежные вершины x и y и построим два новых графа: G_1, получающийся добавлением ребра (x, y) к графу G, и G_2, получающийся из G слиянием вершин x и y. Операция слияния состоит в удалении вершин x и y и добавлении новой вершины z и ребер, соединяющих ее с каждой вершиной, с которой была смежна хотя бы одна из вершин x, y. На рис. 3.4 показаны графы G_1 и G_2, получающиеся из графа G, изображенного на рис. 3.3, с помощью этих операций, если в качестве x и y взять вершины a и f.

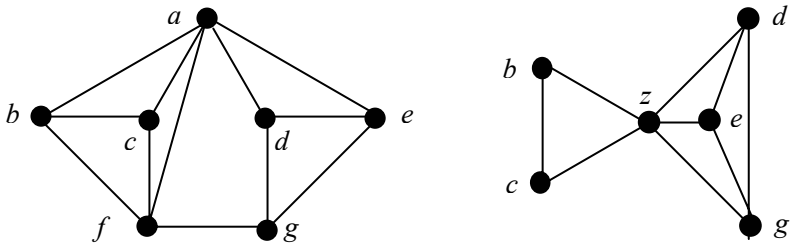


Рис. 3.4

Если в правильной раскраске графа G вершины x и y имеют разные цвета, то она будет правильной и для графа G_1. Если же цвета вершин x и y в раскраске графа G одинаковы, то граф G_2 можно раскрасить в то же число цветов: новая вершина z окрашивается в тот цвет, в который окрашены вершины x и y, а все остальные вершины сохраняют те цвета, которые они имели в графе G. Обратно, раскраска каждого из графов G_1, G_2, очевидно, дает раскраску графа G в то же число цветов. Поэтому

$$\chi(G) = \min \{ \chi(G_1), \chi(G_2)\},$$

что дает возможность рекурсивного нахождения раскраски графа в минимальное число цветов. Заметим, что граф G_1 имеет столько же вершин, сколько исходный граф, но у него больше ребер. Поэтому рекурсия в конечном счете приводит к полным графам, для которых задача о раскраске решается тривиально.

3.2.3. Рационализация

В описанную схему решения задачи о раскраске можно включить тот же прием сжатия по включению, что и для задачи о независимом множестве. Небольшое отличие состоит в том, что теперь вершины a и b должны быть несмежны. Итак, пусть в графе G имеются две несмежные вершины a и b такие, что $V(a) \subseteq V(b)$. Будем говорить, что вершина b несмежно поглощает вершину a, а вершину a называть несмежно поглощаемой. В графе на рис. 3.3 нет несмежно поглощаемых вершин (но вершины b и c смежно поглощают друг друга). В графе G_1 на рис. 3.4 вершина a несмежно поглощает вершину g.

Лемма 3.4. *Если вершина a является несмежно поглощаемой в графе G, то* $\chi(G - a) = \chi(G)$.

Доказательство. Допустим, вершина a несмежно поглощается вершиной b. Рассмотрим правильную раскраску графа $G - a$ в наименьшее число цветов. Применим эту же раскраску к графу G, окрасим a в тот цвет, который имеет вершина b. Так как вершина a смежна только с такими вершинами, с которыми смежна b, то получится правильная раскраска графа G в то же самое число цветов. Следовательно, $\chi(G) = \chi(G - a)$. □

Как и для задачи о независимом множестве, для некоторых графов этот прием позволяет находить решение, совсем не прибегая к перебору. Допустим, вершина b смежно поглощает вершину a в графе G. Тогда в дополнительном графе $\overline{G}$, очевидно, вершина a будет несмежно поглощать вершину b. Верно и обратное утверждение. Поэтому из теоремы 3.3 следует

Теорема 3.5. *В любом графе, дополнительном к хордальному и не являющемся полным, имеется несмежно поглощаемая вершина.*

Таким образом, для графов, дополнительных к хордальным, раскраска в минимальное число цветов может быть найдена с помощью одних только сжатий по включению. Оказывается, и для хордальных графов существует эффективное решение задачи о раскраске.

3.2.4. Хордальные графы

Установим сначала некоторые свойства хордальных графов. Подмножество множества вершин графа называется *разделяющим множеством*, если удаление всех этих вершин приводит к увеличению числа компонент связности. Таким образом, понятие разделяющего множества является обобщением понятия шарнира. Разделяющее множество называется минимальным, если оно не содержится в большем разделяющем множестве.

Лемма 3.6. *В хордальном графе всякое минимальное разделяющее множество является кликой.*

Доказательство. Допустим, что в некотором графе G есть минимальное разделяющее множество X, не являющееся кликой. Это означает, что в X имеются несмежные вершины a и b. При удалении множества X образуется не менее двух новых компонент связности. Пусть C_1 и C_1 – такие компоненты. Вершина a смежна по крайней мере с одной вершиной в каждой из этих компонент. Действительно, если a была бы не смежна, скажем, ни с одной из вершин компоненты C_1, то множество $X - \{a\}$ тоже было бы разделяющим, а это противоречит минимальности разделяющего множества X. То же относится к вершине b. Выберем в компоненте C_1 такие вершины x_1 и y_1, чтобы x_1 была смежна с вершиной a, y_1 – с вершиной b и при этом расстояние между x_1 и y_1 в C_1 было минимальным (возможно $x_1 = y_1$). Аналогично выберем x_2 и y_2 в компоненте C_2. Пусть P_1 – кратчайший путь из x_1 в y_1 в компоненте C_1, а P_2 – кратчайший путь из y_2 в x_2 в компоненте C_2 (каждый из этих путей может состоять из одной вершины). Тогда последовательность a, P_1, b, P_2, a является простым циклом без хорд длины не менее 4. Следовательно, граф G не хордальный. □

Вершина графа называется *симплициальной*, если множество всех смежных с ней вершин является кликой или пустым множеством.

Лемма 3.7. *В любом хордальном графе имеется симплициальная вершина.*

Доказательство. В полном графе любая вершина является симплициальной. Докажем индукцией по числу вершин n, что в любом неполном

хордальном графе есть две несмежные симплициальные вершины. При $n = 2$ это, очевидно, так. Пусть G – хордальный граф с n вершинами, $n > 2$, не являющийся полным. Если G несвязен, то, по предположению индукции, во всех компонентах связности есть симплициальные вершины. Допустим, что граф G связен. Так как он неполный, то в нем есть разделяющее множество, а по лемме 3.6 есть разделяющая клика. Пусть C – такая клика, A и B – две новые компоненты связности, появляющиеся при удалении из графа всех вершин клики C. Рассмотрим подграф G_A, порожденный множеством $A \cup C$. Если он полный, то в нем любая вершина симплициальна. Если же он неполный, то по предположению индукции в нем есть две несмежные симплициальные вершины. Хотя бы одна из этих двух вершин принадлежит множеству A. Итак, в любом случае в множестве A имеется вершина a, являющаяся симплициальной в графе G_A. Окрестность вершины a во всем графе G совпадает с ее окрестностью в подграфе G_A. Следовательно, a – симплициальная вершина графа G. Аналогично, в множестве B имеется симплициальная вершина графа G и она не смежна с вершиной a. □

Существование симплициальных вершин можно использовать для создания эффективного алгоритма раскрашивания хордального графа в наименьшее число цветов. План такого алгоритма содержится в доказательстве следующей теоремы.

Теорема 3.8. *Для любого хордального графа* $\chi(G) = \omega(G)$.

Доказательство. Пусть G – хордальный граф с n вершинами и $\omega(G) = k$. Покажем, что граф G можно правильно раскрасить в k цветов. Найдем в нем симплициальную вершину и обозначим ее через x_n, а граф, полученный удалением этой вершины, через G_{n-1}. Этот граф тоже хордальный, значит, в нем тоже есть симплициальная вершина. Пусть x_{n-1} – симплициальная вершина в графе G_{n-1}, а G_{n-2} – граф, получаемый из него удалением этой вершины. Продолжая действовать таким образом, получим последовательность вершин $x_n, x_{n-1}, \ldots, x_1$ и последовательность графов $G_n, G_{n-1}, \ldots, G_1$ (здесь $G_n = G$), причем при каждом i вершина x_i является симплициальной в графе G_i, а граф G_{i-1} получается из G_i удалением этой вершины.

Допустим, что граф G_{i-1} правильно раскрашен в k цветов. Покажем, что вершину x_i можно покрасить в один из этих цветов, сохраняя правильность раскраски. Действительно, x_i – симплициальная вершина графа G_i, значит, множество C всех смежных с ней в этом графе вершин является кликой. Так как при добавлении к множеству C вершины x_i тоже полу-

чается клика, а мощность наибольшей клики в графе G равна k, то $|C| \leq$ $\leq k - 1$. Значит, для окрашивания вершин множества C использовано не более $k - 1$ цвета. Поэтому для вершины x_i можно использовать один из оставшихся цветов.

Итак, каждый из графов G_i, а значит, и исходный граф G, можно правильно раскрасить в k цветов. Отсюда следует, что $\chi(G) \leq \omega(G)$, а вместе с неравенством (3.1) это дает утверждение теоремы. □

3.2.5. Раскраска ребер

Наряду с задачей о раскраске вершин имеется задача о *раскраске ребер* графа, когда цвета назначаются ребрам. Раскраска ребер (или реберная раскраска) называется правильной, если любые два ребра, имеющие общую вершину, окрашены в разные цвета. Минимальное число цветов, необходимое для правильной раскраски ребер графа G, называется *хроматическим индексом* графа и обозначается через $\chi'(G)$.

Обозначим через $\Delta(G)$ максимальную степень вершины в графе. При правильной реберной раскраске все ребра, инцидентные одной вершине, должны иметь разные цвета. Отсюда следует, что для любого графа выполняется неравенство $\chi'(G) \geq \Delta(G)$. Для некоторых графов имеет место строгое неравенство, например, $\Delta(C_3) = 2$, а $\chi'(C_3) = 3$. Следующая теорема, доказанная В.Г. Визингом в 1964 г., показывает, что $\chi'(G)$ может отличаться от $\Delta(G)$ не более чем на 1.

Теорема 3.9. *Для любого графа G справедливы неравенства*

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1.$$

Доказательство. Приводимое ниже доказательство дает и план алгоритма для раскрашивания ребер графа не более чем в $\Delta(G) + 1$ цветов. Оно основано на двух операциях перекрашивания, с описания которых и начнем. Далее будут рассматриваться частичные реберные раскраски, то есть правильные раскраски, при которых некоторые ребра остаются неокрашенными.

Допустим, ребра графа G правильно (может быть, частично) раскрашены. Пусть α и β – два из использованных в этой раскраске цветов. Рассмотрим подграф H, образованный всеми ребрами, имеющими цвета α или β. В этом подграфе степень каждой вершины не превосходит 2, следовательно, каждая компонента связности в нем является цепью или циклом. Такую компоненту будем называть (α, β)-*компонентой*. Если в какой-нибудь (α, β)-компоненте поменять местами цвета α и β (то есть все

ребра, окрашенные в цвет α, перекрасить в цвет β и наоборот), то полученная раскраска тоже будет правильной. Эту операцию назовем перекраской (α, β)-компоненты.

Другая операция применяется к частично раскрашенному подграфу, называемому *веером*. Будем говорить, что при данной раскраске цвет α отсутствует в вершине x, если ни одно из ребер, инцидентных вершине x, не окрашено в этот цвет. Веером называется подграф $F(x, y_1, \ldots, y_k, \alpha_1, \ldots, \alpha_k)$, состоящий из вершин $x, y_1, \ldots, y_k$ и ребер $(x, y_1), \ldots, (x, y_k)$, в котором:

ребро (x, y_1) не окрашено;

ребро (x, y_i) окрашено в цвет α_{i-1}, $i = 2, \ldots, k$;

в вершине y_i отсутствует цвет α_i, $i = 1, \ldots, k$;

$\alpha_1, \ldots, \alpha_{k-1}$ все попарно различны.

Перекраска веера состоит в том, что ребра $(x, y_1), \ldots, (x, y_{k-1})$ окрашиваются соответственно в цвета $\alpha_1, \ldots, \alpha_{k-1}$, а ребро (x, y_k) становится неокрашенным. Очевидно, новая частичная раскраска тоже будет правильной. На рис. 3.5 слева показан веер, а справа – результат его перекраски. Цвета ребер представлены числами, а отсутствующие цвета в вершинах – числами со знаком минус. Неокрашенное ребро изображено пунктиром.

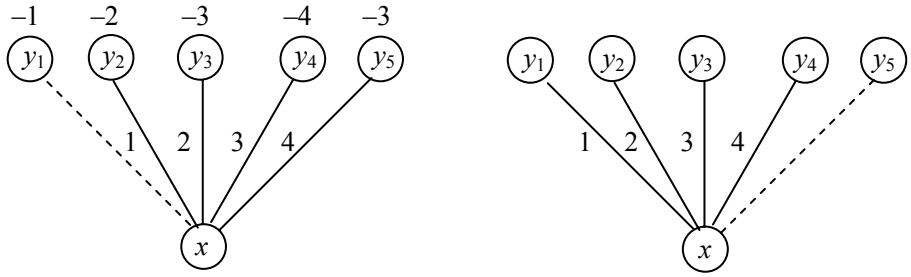


Рис. 3.5

Покажем, что с помощью этих двух процедур перекрашивания можно ребра любого графа G окрасить в не более чем $\Delta(G) + 1$ цветов. Допустим, что уже построена частичная правильная раскраска, использующая не более чем $\Delta(G) + 1$ цветов, и имеется неокрашенное ребро (x, y). Так как число разрешенных цветов больше, чем максимальная степень вершины, то в каждой вершине какой-нибудь цвет отсутствует. Допустим, в вершине x отсутствует цвет β.

Будем строить веер следующим образом. Положим $y_1 = y$ и пусть α_1 – цвет, отсутствующий в вершине y. Получаем веер $F(x, y_1, \alpha_1)$. Допустим,

веер $F(x, y_1, \ldots, y_k, \alpha_1, \ldots, \alpha_k)$ уже построен. Если цвет α_k отличен от $\alpha_1, \ldots, \alpha_{k-1}$ и имеется инцидентное вершине x ребро (x, z) этого цвета, то увеличиваем k на 1 и полагаем $y_k = z$, α_k – цвет, отсутствующий в вершине z. Этот процесс построения веера продолжается до тех пор, пока не наступит одно из следующих событий.

(А) Нет ребра цвета α_k, инцидентного вершине x. Перекрашиваем веер, в результате ребро (x, y) становится окрашенным, а ребро (x, y_k) – неокрашенным, причем цвет α_k отсутствует и в вершине y_k, и в вершине x. Но тогда можно это ребро окрасить в цвет α_k, получим правильную раскраску, в которой на одно окрашенное ребро больше.

(Б) Цвет α_k совпадает с одним из цветов $\alpha_1, \ldots, \alpha_{k-1}$ (именно этот случай изображен на рисунке 3.5). Пусть $\alpha_k = \alpha_i$. Рассмотрим вершины x, y_i, y_k. В каждой из них отсутствует какой-нибудь из цветов β или α_k. Значит, в подграфе, образованном ребрами этих двух цветов, степень каждой из этих вершин не превосходит 1. Следовательно, все три вершины не могут принадлежать одной (α_k, β)-компоненте. Рассмотрим две возможности.

(Б1) Вершины x и y_i принадлежат разным (α_k, β)-компонентам. Перекрасим веер $F(x, y_1, \ldots, y_i, \alpha_1, .., \alpha_i)$. Ребро (x, y_i) станет неокрашенным. Теперь перекрасим (α_k, β)-компоненту, содержащую вершину y_i. После этого цвет β будет отсутствовать в вершине y_i и ребро (x, y_i) можно окрасить в этот цвет.

(Б2) Вершины x и y_k принадлежат разным (α_k, β)-компонентам. Перекрасим веер $F(x, y_1, \ldots, y_i, \alpha_1, .., \alpha_i)$. Ребро (x, y_i) станет неокрашенным. Теперь перекрасим (α_k, β)-компоненту, содержащую вершину y_k. После этого цвет β будет отсутствовать в вершине y_k и ребро (x, y_k) можно окрасить в этот цвет.

Итак, в любом случае получаем правильную раскраску, в которой добавилось еще одно раскрашенное ребро (x, y). □

На рис. 3.6 иллюстрируются случаи (Б1) и (Б2) на примере веера с рис. 3.5. Здесь $k = 5$, $i = 3$. Левое изображение соответствует случаю (Б1): вершины x и y_3 принадлежат разным (3, 5)-компонентам. После перекраски веера $F(x, y_1, y_2, y_3, 1, 2, 3)$ и (3, 5)-компоненты, содержащей вершину y_3, появляется возможность окрасить ребро (x, y_3) в цвет 5. Случай (Б2) показан справа: здесь вершины x и y_5 принадлежат разным (3, 5)-компонентам, поэтому после перекраски веера $F(x, y_1, y_2, y_3, y_4, y_5, 1, 2, 3, 4, 3)$ и (3,5)-компоненты, содержащей вершину y_5, появляется возможность окрасить ребро (x, y_5) в цвет 5.

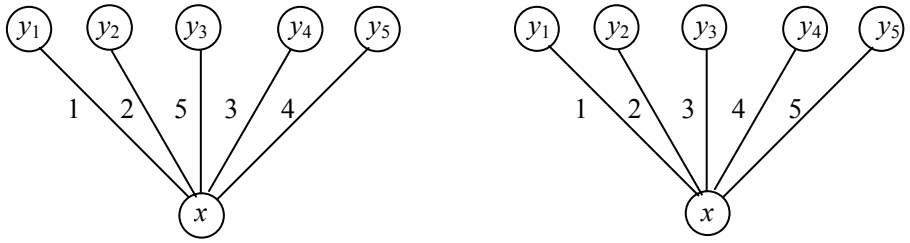


Рис. 3.6

Итак, все графы делятся на два класса: у одних хроматический индекс равен максимальной степени вершины, у других он на единицу больше. Оказывается, определение принадлежности графа к тому или иному классу является NP-трудной задачей. Алгоритм, который можно извлечь из доказательства теоремы 3.9, за полиномиальное время находит раскраску в не более чем $\Delta(G) + 1$ цветов. Его можно назвать «идеальным» приближенным алгоритмом – более высокую точность имеет только точный алгоритм.

3.3. Паросочетания

3.3.1. Паросочетания и реберные покрытия

Паросочетанием в графе называется множество ребер, попарно не имеющих общих вершин. Задача о паросочетании состоит в том, чтобы в данном графе найти паросочетание с наибольшим числом ребер. Это число для графа G будем обозначать через $\pi(G)$. *Реберным покрытием* графа называется такое множество ребер, что всякая вершина графа инцидентна хотя бы одному из этих ребер. Наименьшее число ребер в реберном покрытии графа G обозначим через $\rho(G)$. Заметим, что реберное покрытие существует только для графов без изолированных вершин.

Определение паросочетания похоже на определение независимого множества вершин, паросочетание иногда так и называют – независимое множество ребер. Эта аналогия усиливается еще тесной связью между реберными покрытиями и паросочетаниями, подобно тому, как связаны между собой вершинные покрытия и независимые множества. Даже равенство, количественно выражающее эту связь, имеет точно такой же вид (напомним, что числа независимости $\alpha(G)$ и вершинного покрытия $\beta(G)$ связаны равенством $\alpha(G) + \beta(G) = n$). Приводимое ниже доказательство этого факта имеет алгоритмическое значение, так как показывает, каким образом каждая из двух задач может быть сведена к другой.

Теорема 3.10. *Для любого графа G с n вершинами, не имеющего изолированных вершин, справедливо равенство* $\pi(G) + \rho(G) = n$.

Доказательство. Пусть M – наибольшее паросочетание в графе G. Обозначим через W множество всех вершин графа, не покрытых ребрами этого паросочетания. Тогда $|W| = n - 2\pi(G)$. Очевидно, W – независимое множество (иначе M не было бы наибольшим). Выберем для каждой вершины из M какое-нибудь инцидентное ей ребро. Пусть F – множество всех выбранных ребер. Тогда $M \cup F$ – реберное покрытие и $|M \cup F| = n - \pi(G)$, следовательно, $\rho(G) = n - \pi(G)$.

Обратно, пусть C – наименьшее реберное покрытие графа G. Рассмотрим подграф H графа G, образованный ребрами этого покрытия. В графе H один из концов каждого ребра является вершиной степени 1 (ребро, каждая вершина которого инцидентна по крайней мере еще одному ребру, можно было бы удалить из C, оставшиеся ребра по-прежнему покрывали бы все вершины). Отсюда следует, что каждая компонента связности графа H является звездой (звезда – это дерево, у которого не более одной вершины степени больше 1). Так как в любом лесе сумма количеств ребер и компонент связности равна числу вершин, то число компонент связности в графе H равно $n - \rho(G)$. Выбрав по одному ребру из каждой компоненты, получим паросочетание. Отсюда следует, что $\pi(G) \geq n - \rho(G)$. □

Несмотря на такое сходство между «вершинными» и «реберными» вариантами независимых множеств и покрытий, имеется кардинальное различие в сложности соответствующих экстремальных задач. «Вершинные» задачи, как уже отмечалось, являются NP-полными. Для реберных же известны полиномиальные алгоритмы. Они основаны на методе чередующихся цепей, к рассмотрению которого мы теперь переходим. Отметим только еще, что ситуация похожа на то, что наблюдается для задач об эйлеровом и гамильтоновом циклах – реберный вариант эффективно решается, а вершинный является NP-полным.

3.3.2. Метод увеличивающих цепей

Пусть G – граф, M – некоторое паросочетание в нем. Ребра паросочетания будем называть *сильными,* остальные ребра графа – *слабыми*. Вершину назовем *свободной,* если она не принадлежит ребру паросочетания. На рис. 3.7 слева показан граф и в нем выделены ребра паросочетания $M = \{(1, 2), (6, 8), (9, 10), (3, 7)\}$. Вершины 4 и 5 – свободные. Заметим, что к этому паросочетанию нельзя добавить ни одного ребра, то есть оно максимальное. Однако оно не является наибольшим. В этом легко убедиться, если рассмотреть путь 5, 6, 8, 9, 10, 7, 3, 4 (показан пунктиром).

Он начинается и заканчивается в свободных вершинах, а вдоль пути чередуются сильные и слабые ребра. Если на этом пути превратить каждое сильное ребро в слабое, а каждое слабое – в сильное, то получится новое паросочетание, показанное на рисунке справа, в котором на одно ребро больше. Увеличение паросочетания с помощью подобных преобразований – в этом и состоит суть метода увеличивающих цепей.

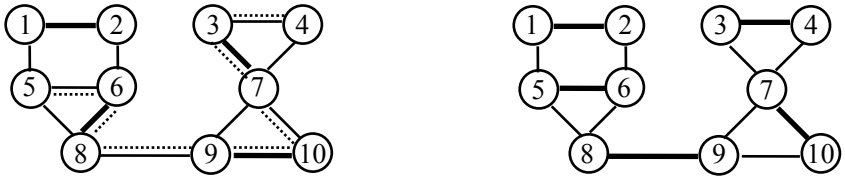


Рис. 3.7

Сформулируем необходимые понятия и докажем теорему, лежащую в основе этого метода. *Чередующейся цепью* относительно данного паросочетания называется простой путь, в котором чередуются сильные и слабые ребра (то есть за сильным ребром следует слабое, за слабым – сильное). Чередующаяся цепь называется *увеличивающей*, если она соединяет две свободные вершины. Если M – паросочетание, P – увеличивающая цепь относительно M, то легко видеть, что $M \otimes P$ – тоже паросочетание и $|M \otimes P| = |M| + 1$.

Теорема 3.11. *Паросочетание является наибольшим тогда и только тогда, когда относительно него нет увеличивающих цепей.*

Доказательство. Если есть увеличивающая цепь, то, поступая так, как в рассмотренном примере, то есть, заменяя вдоль этой цепи сильные ребра на слабые и наоборот, мы, очевидно, получим большее паросочетание. Для доказательства обратного утверждения рассмотрим паросочетание M в графе G и предположим, что M не наибольшее. Покажем, что тогда имеется увеличивающая цепь относительно M. Пусть M' – другое паросочетание и $|M'| > |M|$. Рассмотрим подграф H графа G, образованный теми ребрами, которые входят в одно и только в одно из паросочетаний M, M'. Иначе говоря, множеством ребер графа H является симметрическая разность $M \otimes M'$. В графе H каждая вершина инцидентна не более чем двум ребрам (одному из M и одному из M'), то есть имеет степень не более двух. В таком графе каждая компонента связности – путь или цикл. В каждом из этих путей и циклов чередуются ребра из M и M'. Так как $|M'| > |M|$, то имеется компонента, в которой ребер из M' содержится больше, чем ребер из M. Это может быть только путь, у которого оба кон-

цевых ребра принадлежат M'. Легко видеть, что относительно M этот путь будет увеличивающей цепью. □

Для решения задачи о паросочетании остается научиться находить увеличивающие цепи или убеждаться, что таких цепей нет. Тогда, начиная с любого паросочетания (можно и с пустого множества ребер), можем строить паросочетания со все увеличивающимся количеством ребер до тех пор, пока не получим такое, относительно которого нет увеличивающих цепей. Оно и будет наибольшим. Известны эффективные алгоритмы, которые ищут увеличивающие цепи для произвольных графов. Рассмотрим сначала более простой алгоритм, решающий эту задачу для двудольных графов.

3.3.3. Паросочетания в двудольных графах

Пусть $G = (A, B, E)$ – двудольный граф с долями A и B, M – паросочетание в G. Всякая увеличивающая цепь, если такая имеется, соединяет вершину из множества A с вершиной из множества B.

Зафиксируем некоторую свободную вершину $a \in A$. Мы хотим найти увеличивающий путь, начинающийся в a, либо убедиться в том, что таких путей нет. Оказывается, нет необходимости рассматривать все чередующиеся пути, начинающиеся в вершине a, для того, чтобы установить, какие вершины достижимы из вершины a чередующимися путями.

Вершину x назовем *четной* или *нечетной* в зависимости от того, четно или нечетно расстояние между нею и вершиной a. Так как граф двудольный, то любой путь, соединяющий вершину a с четной (нечетной) вершиной, имеет четную (нечетную) длину. Поэтому в чередующемся пути, ведущем из вершины a в четную (нечетную) вершину, последнее ребро обязательно сильное (слабое).

Определим *дерево достижимости* как максимальное дерево с корнем a, в котором каждый путь, начинающийся в корне, является чередующимся. Дерево достижимости определено не однозначно, но любое такое дерево в двудольном графе обладает следующим свойством.

Лемма 3.12. *Вершина x принадлежит дереву достижимости тогда и только тогда, когда существует чередующийся путь, соединяющий вершины a и x.*

Доказательство. Рассмотрим некоторое дерево достижимости T и докажем, что всякая вершина x, достижимая из вершины a чередующимся путем, принадлежит этому дереву. Проведем индукцию по длине кратчайшего чередующегося пути из a в x. Пусть y – предпоследняя (то есть предшествующая x) вершина такого пути. По предположению индукции,

вершина y принадлежит дереву T. Если она четная, то любой чередующийся путь из вершины a в вершину y заканчивается сильным ребром. Следовательно, в дереве T вершину y с ее отцом соединяет сильное ребро, а ребро (x, y) – слабое. Поэтому, если добавить к дереву вершины x и ребро (x, y), то путь в дереве, соединяющий a с x, будет чередующимся. Значит, если предположить, что дерево T не содержит вершины x, то окажется, что оно не максимально, а это противоречит определению. Аналогично рассматривается случай, когда вершина y нечетная. □

Итак, для решения задачи остается научиться строить дерево достижимости. Для этого можно использовать слегка модифицированный поиск в ширину из вершины a. Отличие от стандартного поиска в ширину состоит в том, что открываемые вершины классифицируются на четные и нечетные. Для четных вершин исследуются инцидентные им слабые ребра, а для нечетных – сильные. Через $V(x)$, как обычно, обозначается множество вершин, смежных с вершиной x, Q – очередь, используемая при поиске в ширину. Если вершина x не является свободной, то есть инцидентна некоторому сильному ребру, то другая вершина этого ребра обозначается через $p(x)$.

Алгоритм 9. *Построение дерева достижимости*

```
1   объявить все вершины новыми
2   объявить вершину a четной
3   a ⇒ Q
4   создать дерево T из одной вершины a
5   while  Q ≠ ∅ do
6          x ⇐ Q
7          if вершина x нечетная
8             then if вершина x несвободная
9                     then y := p(x)
10                         y ⇒ Q
11                         объявить вершину y четной
12                         добавить к дереву T вершину y и ребро (x, y)
13            else for y ∈ V(x) do
14                 if вершина y новая
15                    then y ⇒ Q
16                         объявить вершину y нечетной
17                         добавить к T вершину y и ребро (x, y)
```

Если очередная рассматриваемая вершина x оказывается свободной (это выясняется при проверке в строке 8), нет необходимости доводить

построение дерева до конца. В этом случае путь между вершинами a и x в дереве является увеличивающим путем и можно его использовать для построения большего паросочетания. После этого снова выбирается свободная вершина (если такая еще есть) и строится дерево достижимости. В приведенном тексте алгоритма соответствующий выход отсутствует, но его легко предусмотреть, добавив ветвь **else** к оператору **if** в строке 8.

Если дерево построено и в нем нет других свободных вершин, кроме корня, то нужно выбрать другую свободную вершину и построить дерево достижимости для нее (конечно, если в графе больше двух свободных вершин). При этом, как показывает следующая лемма, вершины первого дерева можно удалить из графа.

Лемма 3.13. *Если дерево достижимости содержит хотя бы одну вершину увеличивающего пути, то оно содержит увеличивающий путь.*

Доказательство. Пусть дерево достижимости T с корнем a имеет общие вершины с увеличивающим путем P, соединяющим свободные вершины b и c. Покажем, что в T есть увеличивающий путь (это не обязательно путь P). Достаточно доказать, что хотя бы одна из вершин b, c принадлежит дереву T. Если одна из них совпадает с вершиной a, то из леммы 3.12 следует, что и другая принадлежит дереву, так что в этом случае в дереве имеется увеличивающий путь между вершинами b и c. Допустим, обе вершины b и c отличны от a. Пусть R – простой путь, начинающийся в вершине a, заканчивающийся в вершине x, принадлежащей пути P, и не содержащий других вершин пути P. Очевидно, последнее ребро пути R слабое. Вершина x делит путь P на два отрезка, P_b и P_c, содержащие соответственно вершины b и c. В одном из этих отрезков, скажем, в P_b, ребро, инцидентное вершине x, сильное. Тогда объединение путей R и P_b образует чередующийся путь, соединяющий вершины a и b. По лемме 3.12, вершина b принадлежит дереву T. □

Из этой леммы следует, что если полностью построенное дерево достижимости не содержит других свободных вершин, кроме корня, то ни одна вершина этого дерева не принадлежит никакому увеличивающему пути. Поэтому, приступая к построению следующего дерева достижимости, вершины первого дерева можно удалить из графа (временно, конечно). Этот процесс – построение деревьев достижимости и удаление их вершин из графа – продолжается до тех пор, пока либо будет найден увеличивающий путь, либо останется граф с не более чем одной свободной вершиной. В первом случае паросочетание увеличивается, граф восстанавливается, и вновь начинается поиск увеличивающего пути. Во втором случае имеющееся паросочетание является наибольшим.

Так как при поиске в ширину каждое ребро исследуется не более чем дважды, то общее время поиска увеличивающего пути для данного паросочетания есть $O(m)$. Число ребер в паросочетании не может превышать $n/2$, поэтому общая сложность алгоритма будет $O(mn)$.

3.3.4. Паросочетания в произвольных графах (алгоритм Эдмондса)

Для графа, не являющегося двудольным, утверждение леммы 3.12 может быть неверным. Пример этого показан на рис. 3.8. В графе, изображенном слева, с паросочетанием из двух ребер, имеется увеличивающий путь 5, 3, 1, 2, 4, 6. Справа показано дерево достижимости, построенное для вершины 5. Вершина 6 не вошла в это дерево, хотя имеется чередующийся путь, соединяющий ее с вершиной 5. В результате увеличивающий путь не будет найден. Причина этого – наличие в графе ребра $(1, 2)$, соединяющего вершины, находящиеся на одинаковом расстоянии от корня дерева. В тот момент, когда исследуется это ребро, обе вершины 1 и 2 уже присутствуют в дереве, поэтому построение дерева заканчивается. Наличие такого ребра означает, что в графе есть нечетный цикл, и это является настоящей причиной неудачи.

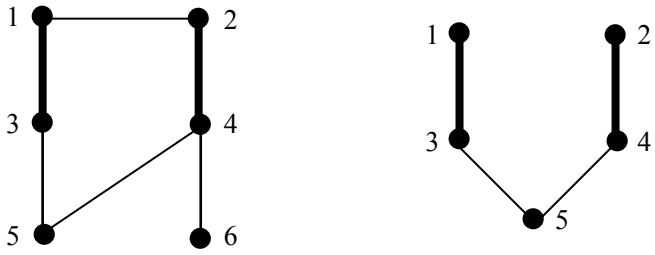


Рис. 3.8

Тем не менее и для графов с нечетными циклами задачу о наибольшем паросочетании можно решать эффективно. Рассмотрим алгоритм построения наибольшего паросочетания в произвольном графе, предложенный Эдмондсом.

Сначала, как и в случае двудольного графа, методом поиска в ширину строится дерево достижимости для некоторой свободной вершины a. Построение дерева для двудольного графа прекращалось, если к дереву нельзя было добавить ни одной вершины либо если к нему добавлялась свободная вершина, то есть обнаруживалось наличие увеличивающего пути. Для произвольного графа будем прекращать построение дерева еще в том случае, когда исследуемое ребро соединяет две четные вершины дерева. При поиске в ширину это может быть только тогда, когда эти две

вершины находятся на одинаковом расстоянии от корня. Обнаружение такого ребра означает, что найден подграф, называемый *цветком* (рис. 3.9). Он состоит из чередующегося пути P, соединяющего корень дерева a с некоторой вершиной b, и нечетного цикла C. При этом b является единственной общей вершиной пути P и цикла C, а C можно рассматривать как замкнутый чередующийся путь, начинающийся и заканчивающийся в вершине b. На рисунке показаны также смежные четные вершины x и y, находящиеся на одинаковом расстоянии от вершины a.

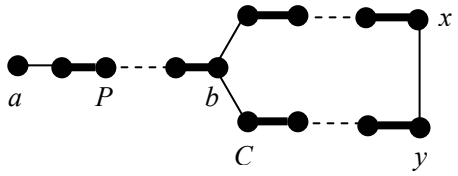


Рис. 3.9

Выявление цветков не представляет трудности – нужно только добавить ветвь **else** к оператору **if** в строке 14 алгоритма 9. Первое, что мы сделаем, обнаружив цветок, – превратим все сильные ребра пути P в слабые, а слабые – в сильные. После этого преобразования множество сильных ребер является паросочетанием той же мощности, но вместо вершины a свободной вершиной станет вершина b. Таким образом, на цикле C будет одна свободная вершина и этот цикл является чередующимся путем, начинающимся и заканчивающимся в этой вершине. Покажем, что такой цикл можно стянуть в одну вершину, не теряя информации о существовании увеличивающих путей.

Операция стягивания подграфа H в графе G состоит в следующем. Все вершины этого подграфа удаляются из графа, а вместо них добавляется новая вершина. Эта новая вершина соединяется ребрами с теми и только теми из оставшихся вершин графа, которые были смежны хотя бы с одной вершиной подграфа H. Граф, получаемый в результате такой операции, будем обозначать через G/H.

Теорема 3.14. *Пусть M – паросочетание в графе G, C – цикл длины $2k + 1$ в этом графе, причем на цикле имеется k сильных ребер и одна свободная вершина. Пусть M' – паросочетание в графе $G' = G/C$, составленное из всех ребер паросочетания M, не принадлежащих циклу C. Паросочетание M является наибольшим в графе G тогда и только тогда, когда M' – наибольшее паросочетание в графе G'.*

Доказательство. Докажем, что из существования увеличивающего пути относительно паросочетания M в графе G следует, что существует

увеличивающий путь относительно паросочетания M' в графе G' и обратно.

Пусть b – свободная вершина цикла C. Новую вершину, образованную в графе G' при стягивании цикла C, обозначим через c. Отметим, что она является свободной вершиной относительно паросочетания M'.

Пусть P – увеличивающий путь в графе G. Если он не содержит вершин цикла C, то он будет увеличивающим путем и в графе G'. В противном случае рассмотрим отрезок P' пути P, начинающийся в свободной вершине, отличной от b, заканчивающийся в вершине x, лежащей на цикле C, и не содержащий других вершин цикла C. Если в пути P' заменить вершину x вершиной c, то, очевидно, получится увеличивающий путь в графе G'.

Обратно, пусть P' – увеличивающий путь в графе G'. Если P' не проходит через вершину c, то он будет увеличивающим путем и в графе G. В противном случае рассмотрим путь P'', получающийся удалением вершины c из пути P'. Можно считать, что вершина c была последней вершиной пути P', а путь P'' заканчивается в предпоследней вершине x. Так как вершина x смежна с вершиной c в графе G', то в графе G на цикле C имеется вершина y, смежная с x. Добавим к пути P'' тот из отрезков цикла C, соединяющих вершину y с вершиной b, который начинается сильным ребром. В результате получится увеличивающий путь в графе G. □

Из доказательства видно, что увеличивающий путь в графе G при известном увеличивающем пути в графе G' находится за линейное время. Для получения оценок времени работы алгоритма в целом требуется еще проработка ряда деталей, например подробностей выполнения операции стягивания и т.д. Однако ясно, что это время ограничено полиномом.

3.4. Оптимальные каркасы

3.4.1. Задача об оптимальном каркасе и алгоритм Прима

Задача об оптимальном каркасе (стягивающем дереве) состоит в следующем. Дан обыкновенный граф $G = (V, E)$ и весовая функция на множестве ребер w: $V \to \mathbf{R}$. Вес множества $X \subseteq E$ определяется как сумма весов составляющих его ребер. Требуется в графе G найти каркас максимального веса. Обычно рассматривают задачу на минимум, но это не существенно – она преобразуется в задачу на максимум, если заменить функцию w на $-w$. В этом разделе будем предполагать, что граф G связен, так что решением задачи всегда будет дерево. Для решения задачи об оп-

тимальном каркасе известно несколько алгоритмов. Рассмотрим два из них.

В *алгоритме Прима* на каждом шаге рассматривается частичное решение задачи, представляющее собой дерево. Вначале это дерево состоит из единственной вершины, в качестве которой может быть выбрана любая вершина графа. Затем к дереву последовательно добавляются ребра и вершины, пока не получится остовное дерево, то есть каркас. Для того чтобы из текущего дерева при добавлении нового ребра опять получилось дерево, это новое ребро должно соединять вершину дерева с вершиной, еще не принадлежащей дереву. Такие ребра будем называть *подходящими* относительно рассматриваемого дерева. В алгоритме Прима используется следующее правило выбора: на каждом шаге из всех подходящих ребер выбирается ребро наибольшего веса. Это ребро вместе с одно новой вершиной добавляется к дереву. Если обозначить через U и F множества вершин и ребер строящегося дерева, а через W множество вершин, еще не вошедших в это дерево, то алгоритм Прима можно представить следующим образом.

Алгоритм 10. *Построение оптимального каркаса методом Прима*

1 $U := \{a\}$, где a – произвольная вершина графа
2 $F := \varnothing$
3 $W := V - \{a\}$
4 **while** $W \neq \varnothing$ **do**
5 найти ребро наибольшего веса $e = (x, y)$ среди всех таких ребер, у которых $x \in U, y \in W$
6 $F := F \cup \{e\}$
7 $U := U \cup \{y\}$
8 $W := W - \{y\}$

Докажем, что алгоритм Прима действительно находит оптимальный каркас. Дерево F назовем *фрагментом*, если существует такой оптимальный каркас T_0 графа G, что F является подграфом дерева T_0. Иначе говоря, фрагмент – это дерево, которое можно достроить до оптимального каркаса.

Теорема 3.15. *Если F – фрагмент, e – подходящее ребро наибольшего веса относительно F, то* $F \cup \{e\}$ *– фрагмент.*

Доказательство. Пусть T_0 – оптимальный каркас, содержащий F в качестве подграфа. Если ребро e принадлежит T_0, то $F \cup \{e\}$ – подграф T_0 и, следовательно, фрагмент. Допустим, e не принадлежит T_0. Если добавить ребро e к дереву T_0, то образуется цикл. В этом цикле есть еще хотя

бы одно подходящее ребро относительно F (никакой цикл, очевидно, не может содержать единственное подходящее ребро). Пусть e' – такое ребро. Тогда подграф $T_0' = T_0 - e' + e$, получающийся из T_0 удалением ребра e' и добавлением ребра e, тоже будет деревом. Так как $w(e') \le w(e)$, то $w(T_0') \ge w(T_0)$. Но T_0 – оптимальный каркас, следовательно, $w(T_0') = w(T_0)$ и T_0' – тоже оптимальный каркас. Но $F \cup \{e\}$ является подграфом графа T_0' и, следовательно, фрагментом. □

Дерево, состоящее из единственной вершины, очевидно, является фрагментом. Из теоремы 3.15 следует, что если после некоторого количества шагов алгоритма Прима дерево F является фрагментом, то оно будет фрагментом и после следующего шага. Следовательно, и окончательное решение, полученное алгоритмом, будет фрагментом, то есть оптимальным каркасом.

Оценим время работы алгоритма Прима. Цикл **while** в строке 4 повторяется один раз для каждой вершины графа, кроме стартовой. Внутри этого цикла есть еще скрытый цикл в строке 5, где ищется ребро наибольшего веса среди всех ребер, соединяющих вершины из множества U с вершинами из W. Допустим, что этот поиск производится самым бесхитростным образом, то есть просматриваются все пары вершин (x, y) с $x \in U$, $y \in W$. Если $|U| = k$, то имеется $k(n - k)$ таких пар. Так как k меняется от 1 до $n - 1$, то всего получаем

$$\sum_{k=1}^{n-1} k(n-k) = n\sum_{k=1}^{n-1} k - \sum_{k=1}^{n-1} k^2 = \frac{n^2(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} = \frac{n^3 - n}{6}$$

пар, которые нужно рассмотреть. Таким образом, трудоемкость алгоритма будет $O(n^3)$.

Небольшое усовершенствование позволяет на порядок ускорить этот алгоритм. Допустим, что для каждой вершины y из множества W известна такая вершина $b(y) \in U$, что $w(b(y), y) = \max\limits_{x \in U} w(x, y)$. Тогда при $|U| = k$ необходимо будет выбрать ребро наибольшего веса среди $n - k$ ребер, а общее число анализируемых ребер будет равно

$$\sum_{k=1}^{n-1} (n-k) = \frac{n(n-1)}{2}.$$

В этом случае, однако, необходимы дополнительные действия для обновления таблицы значений функции b при добавлении одной вершины к дереву, то есть при переносе одной вершины из множества W в множество

U. Сначала, когда множество U состоит из единственной вершины a, полагаем $b(x) = a$ для всех $x \in W$. В дальнейшем эти значения могут меняться. Допустим, на некотором шаге к дереву присоединяется вершина y. Тогда для каждой вершины $z \in W$ либо сохраняется старое значение $b(z)$, либо устанавливается новое $b(z) = y$, в зависимости от того, какое из ребер $(b(z), z)$ и (y, z) имеет больший вес. Иначе говоря, для модификации функции b достаточно в алгоритме 10 после строки 8 (и внутри цикла **while**) добавить следующее:

```
9          for z ∈ W do
10                 if w(b(z), z) < w(y, z) then b(z) := y
```

При $|U| = k$ цикл в строке 9 повторяется $n - k$ раз. Таким образом, дополнительное время, необходимое для обслуживания таблицы b, тоже оценивается сверху квадратичной функцией от n и общая оценка трудоемкости усовершенствованного алгоритма Прима будет $O(n^2)$.

Другой путь к усовершенствованию алгоритма Прима подсказывает следующее замечание. При выборе подходящего ребра (в строке 5 алгоритма 10) можно рассматривать не все пары $(x, y) \in U \times W$, а только те, которые являются ребрами графа. Если граф разреженный, то есть содержит много меньше ребер, чем полный граф, то это может значительно ускорить решение задачи. Дополнительный выигрыш можно получить, если использовать приоритетную очередь для хранения множества ребер, подлежащих исследованию (упражнение 5).

3.4.2. Алгоритм Краскала

Другой жадный алгоритм для задачи об оптимальном каркасе известен как *алгоритм Краскала*. В нем тоже на каждом шаге рассматривается частичное решение. Отличие от алгоритма Прима состоит в том, что в алгоритме Краскала частичное решение всегда представляет собой остовный лес F графа G, то есть лес, состоящий из всех вершин графа G и некоторых его ребер. Вначале F не содержит ни одного ребра, то есть состоит из изолированных вершин. Затем к нему последовательно добавляются ребра, пока не будет построен каркас графа G. Пусть F – лес, построенный к очередному шагу. Ребро графа, не принадлежащее F, назовем красным, если вершины этого ребра принадлежат одной компоненте связности леса F, и зеленым, если они принадлежат разным компонентам. Если к F добавить красное ребро, то образуется цикл. Если же к F добавить зеленое ребро, то получится новый лес, в котором будет на одну компоненту связности меньше, чем в F, так как в результате добавления ребра две компо-

ненты сольются в одну. Таким образом, к F нельзя добавить никакое красное ребро и можно добавить любое зеленое. Для выбора добавляемого ребра применяется тот же жадный принцип, что и в алгоритме Прима – из всех зеленых ребер выбирается ребро наибольшего веса. Для того чтобы облегчить поиск этого ребра, вначале все ребра графа упорядочиваются по убыванию весов: $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_m)$. Теперь последовательность ребер $e_1, e_2, \ldots, e_m$ достаточно просмотреть один раз и для очередного рассматриваемого ребра нужно только уметь определять, является ли оно красным или зеленым относительно построенного к этому моменту леса F. Красные ребра просто пропускаются, а зеленые добавляются к F.

Для более формального описания алгоритма заметим, что текущий лес F определяет разбиение множества вершин графа на области связности этого леса: $V = P_1 \cup P_2 \cup \ldots \cup P_k$ и что красное ребро – это такое, у которого обе вершины принадлежат одной части разбиения. Пусть $Part(x)$ – функция, возвращающая для каждой вершины x имя той части разбиения, которой принадлежит x, а $Unite(x, y)$ – процедура, которая по именам x и y двух частей разбиения строит новое разбиение, заменяя эти две части их объединением. Пусть $e_i = (a_i, b_i)$, $i = 1, \ldots, m$. Тогда алгоритм Краскала (после упомянутого упорядочения ребер) можно записать следующим образом.

Алгоритм 11. *Построение оптимального каркаса методом Краскала*

```
1  for i := 1 to m do
2      x := Part(a_i)
3  y := Part(b_i)
4          if x ≠ y then {F := F ∪ {e_i}, Unite(x, y)}
```

Более подробно алгоритм Краскала рассматривается в главе «Разделенные множества» части 3 настоящей работы. Корректность этого алгоритма следует из общей теоремы Радо – Эдмондса, которая будет рассмотрена в следующем разделе.

3.5. Жадные алгоритмы и матроиды

Жадными (градиентными) называют алгоритмы, действующие по принципу: максимальный выигрыш на каждом шаге. Такая стратегия не всегда ведет к конечному успеху – иногда выгоднее сделать не наилучший, казалось бы, выбор на очередном шаге, с тем чтобы в итоге получить оптимальное решение. Но для некоторых задач применение жадных алгоритмов оказывается оправданным. Одной из самых известных задач такого рода является задача об оптимальном каркасе. В этом разделе приводится теорема, обосновывающая применимость жадных алгоритмов к

задачам определенного типа. Затем будет рассмотрено применение жадного алгоритма в сочетании с методом увеличивающих цепей для решения задачи о паросочетании наибольшего веса в двудольном графе со взвешенными вершинами.

3.5.1. Матроиды

Когда возник вопрос о том, каковы обстоятельства, при которых жадный алгоритм приводит к успеху, или иначе – что особенного в тех задачах, для которых он дает точное решение, оказалось, что математическая теория, с помощью которой что-то в этом можно прояснить, уже существует. Это теория матроидов, основы которой были заложены Уитни в работе 1935 г. Целью создания теории матроидов было изучение комбинаторного аспекта линейной независимости, в дальнейшем обнаружились разнообразные применения понятия матроида, первоначально совсем не имевшиеся в виду.

Определение. *Матроидом* называется пара $M = (E, \Phi)$, где E – конечное непустое множество, Φ – семейство подмножеств множества E, удовлетворяющее условиям:

(1) если $X \in \Phi$ и $Y \subseteq X$, то $Y \in \Phi$;

(2) если $X \in \Phi$, $Y \in \Phi$ и $|X| < |Y|$, то существует такой элемент $a \in Y - X$, что $X \cup (a) \in \Phi$.

Элементы множества E называются *элементами матроида*, а множества из семейства Φ – *независимыми множествами* матроида. Максимальное по включению независимое множество называют *базой* матроида. Из аксиомы (2) следует, что все базы матроида состоят из одинакового количества элементов.

Если E – множество строк некоторой матрицы, а Φ состоит из всех линейно независимых множеств строк этой матрицы, то пара (E, Φ) образует матроид, называемый *матричным матроидом*.

Другим важным типом матроидов являются *графовые матроиды*. Пусть $G = (V, E)$ – обыкновенный граф. Подмножество множества E назовем *ациклическим*, если подграф, образованный ребрами этого подмножества, не содержит циклов, то есть является лесом.

Теорема 3.16. *Если $G = (V, E)$ – обыкновенный граф, Φ – семейство всех ациклических подмножеств множества E, то пара $M_G = (E, \Phi)$ является матроидом.*

Доказательство. Аксиома (1) выполняется, так как всякое подмножество ациклического множества, очевидно, является ациклическим. Докажем, что выполняется и (2). Пусть X и Y – два ациклических множества и

$|X| < |Y|$. Допустим, что не существует такого ребра $e \in Y$, что множество $X \cup \{e\}$ является ациклическим. Тогда при добавлении любого ребра из множества Y к множеству X образуется цикл. Это означает, что концы каждого такого ребра принадлежат одной компоненте связности остовного подграфа, образованного ребрами множества Y. Тогда каждая область связности подграфа X содержится в какой-нибудь области связности подграфа Y. Но компоненты связности каждого из этих подграфов – деревья, а дерево с k вершинами содержит ровно $k - 1$ ребер. Следовательно, в этом случае число ребер в Y не превосходило бы числа ребер в X, что противоречит условию $|X| < |Y|$. □

Базами графового матроида являются все каркасы графа. Для связного графа это будут все его остовные деревья.

3.5.2. Теорема Радо – Эдмондса

Рассмотрим общий тип оптимизационных задач, формулируемых следующим образом. Дано произвольное конечное множество E и некоторое семейство Ф его подмножеств. Для каждого элемента $x \in E$ задан его вес – положительное число $w(x)$. Вес множества $x \subseteq E$ определяется как сумма весов его элементов. Требуется найти множество наибольшего веса, принадлежащее Ф. В эту схему укладываются многие известные задачи, например задача о независимом множестве графа (E – множество вершин графа, вес каждой вершины равен 1, а Ф состоит из всех независимых множеств) или задача об оптимальном каркасе (E – множество ребер графа, Ф состоит из всех ациклических множеств).

Сформулируем теперь жадный алгоритм для решения этой общей задачи. Чтобы отличать этот алгоритм от других жадных алгоритмов, назовем его СПО (сортировка и последовательный отбор).

Алгоритм 12. *Алгоритм СПО*

1 Упорядочить элементы множества E по убыванию весов: $E = \{e_1, e_2, \ldots, e_m\}$, $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_m)$.
2 $A := \varnothing$
3 **for** $i := 1$ **to** m **do**
4 **if** $A \cup \{e_i\} \in \Phi$ **then** $A := A \cup \{e_i\}$

Алгоритм Краскала является примером алгоритма этого типа. Уместен вопрос: каким условиям должно удовлетворять семейство Ф для того, чтобы при любой весовой функции w алгоритм СПО находил оптимальное решение? Исчерпывающий ответ дает следующая теорема Радо – Эдмондса.

Теорема 3.17. *Если* $M = (E, \Phi)$ *– матроид, то для любой весовой функции* $w: \to \mathbf{R}^+$ *множество* A*, найденное алгоритмом СПО, будет множеством наибольшего веса из* Φ*. Если же* $M = (E, \Phi)$ *не является матроидом, то найдется такая функция* $w: \to \mathbf{R}^+$*, что* A *не будет множеством наибольшего веса из* Φ*.*

Доказательство. Предположим, что $M = (E, \Phi)$ является матроидом и $A = \{a_1, a_2, \ldots, a_n\}$ – множество, построенное алгоритмом СПО, причем $w(a_1) \geq w(a_2) \geq \ldots \geq w(a_n)$. Очевидно, A является базой матроида. Пусть $B = \{b_1, b_2, \ldots, b_k\}$ – любое другое независимое множество и $w(b_1) \geq \geq w(b_2) \geq \ldots \geq w(b_k)$. Так как A – база, то $k \leq n$. Покажем, что $w(a_i) \geq w(b_i)$ для каждого $i \in \{1, \ldots, k\}$. Действительно, положим $X = \{a_1, \ldots, a_{i-1}\}$, $Y = \{b_1, \ldots, b_{i-1}, b_i\}$ для некоторого i. Согласно условию (2) определения матроида, в множестве Y имеется такой элемент b_j, что $b_j \notin X$ и множество $X \cup \{b_j\}$ – независимое. В соответствии с алгоритмом, элементом наибольшего веса, который может быть добавлен к X так, чтобы получилось независимое множество, является a_j. Следовательно, $w(a_i) \geq w(b_j) \geq w(b_i)$.

Теперь предположим, что $M = (E, \Phi)$ не является матроидом. Допустим сначала, что нарушается условие (1), то есть существуют такие подмножества X и Y множества E, что $X \in \Phi$, $Y \subset X$ и $Y \notin \Phi$. Определим функцию w следующим образом:

$$w(x) = \begin{cases} 1, & \text{если } x \in Y, \\ 0, & \text{если } x \notin Y. \end{cases}$$

Алгоритм СПО сначала будет рассматривать все элементы множества Y. Так как $Y \notin \Phi$, то не все они войдут в построенное алгоритмом множество A. Следовательно, $w(A) < |Y|$. В то же время имеется множество $X \in \Phi$ такое, что $w(A) < |Y|$. Таким образом, в этом случае алгоритм СПО строит не оптимальное множество. Если же условие (1) выполнено, а не выполняется условие (2), то существуют такие подмножества X и Y множества E, что $X \in \Phi$, $Y \in \Phi$, $|X| < |Y|$ и $X \cup \{x\} \notin \Phi$ для каждого $x \in Y$. Выберем такое ε, что $0 < \varepsilon < |Y|/|X| - 1$, и определим функцию w следующим образом:

$$w(x) = \begin{cases} 1 + \varepsilon, & \text{если } x \in X, \\ 1, & \text{если } x \in Y - X, \\ 0, & \text{если } x \notin X \cup Y. \end{cases}$$

Алгоритм СПО сначала выберет все элементы множества X, а затем отвергнет все элементы из $Y - X$. В результате будет построено множество A с весом $w(A) = (1 + \varepsilon)|X| < |Y|$, которое не является оптимальным, так как $W(Y) = |Y|$. □

Алгоритм Краскала – это алгоритм СПО, применяемый к семейству ациклических множеств ребер графа. Из теорем 3.16 и 3.17 следует, что он действительно решает задачу об оптимальном каркасе. В то же время существует много жадных алгоритмов, не являющихся алгоритмами типа СПО. Примером может служить алгоритм Прима. Эти алгоритмы не попадают под действие теоремы Радо – Эдмондса, для их обоснования нужна иная аргументация.

Если для некоторой конкретной задачи удалось установить применимость к ней алгоритма СПО, это не значит, что все проблемы позади. Этот алгоритм внешне очень прост, но он включает операцию проверки принадлежности множества семейству Ф, эффективное выполнение которой может потребовать серьезных дополнительных усилий. В алгоритме Краскала, например, для этого применяются специальные структуры данных. Ниже рассмотрим еще один пример, когда для успешного решения задачи алгоритм СПО комбинируется с методом увеличивающих цепей для задачи о паросочетании, рассмотренным в разделе 3.3.

3.5.3. Взвешенные паросочетания

Рассмотрим следующую задачу. Дан двудольный граф $G = (A, B, E)$ и для каждой вершины $x \in A$ задан положительный вес $w(x)$. Требуется найти такое паросочетание в этом графе, чтобы сумма весов вершин из доли A, инцидентных ребрам паросочетания, была максимальной. Эту задачу иногда интерпретируют следующим образом. A – это множество работ, а B – множество работников. Ребро в графе G соединяет вершину $a \in A$ с вершиной $b \in B$, если квалификация работника b позволяет ему выполнить работу a. Каждая работа выполняется одним работником. Выполнение работы a принесет прибыль $w(a)$. Требуется так назначить работников на работы, чтобы максимизировать общую прибыль. Покажем, что эта задача может быть решена алгоритмом СПО в сочетании с методом чередующихся цепей.

Множество $X \subseteq A$ назовем *отображаемым*, если в графе G существует такое паросочетание M, что каждая вершина из X инцидентна ребру этого паросочетания. M в этом случае будем называть *отображением* для X. Пусть Ф – семейство всех отображаемых множеств.

Теорема 3.18. *Пара* (A, Φ) *является матроидом.*

Доказательство. Условие (1) определения матроида, очевидно, выполняется. Докажем, что выполняется и условие (2). Пусть $X \in \Phi$, $Y \in \Phi$, $|X| < |Y|$. Рассмотрим подграф H графа G, порожденный всеми вершинами из $X \cup Y$ и всеми смежными с ними вершинами из доли B. Пусть M_X – отображение для X, M_Y – для Y. Так как M_X не является наибольшим паросочетанием в графе H, то по теореме 3.11 относительно него в этом графе существует увеличивающая цепь. Одним из концов этой цепи является свободная относительно M_X вершина $a \in Y$. После увеличения паросочетания M_X с использованием этой цепи, как было описано выше, получим паросочетание M', отображающее множество $X \cup \{a\}$. Следовательно, $X \cup \{a\} \in \Phi$. □

Даже если бы в задаче требовалось только найти только отображаемое множество наибольшего веса, проверка принадлежности множества семейству Ф требовала бы и нахождения соответствующего отображения, то есть паросочетания. На самом же деле построение паросочетания входит в условие задачи. Комбинируя СПО с алгоритмом поиска увеличивающих цепей, получаем следующий алгоритм.

Алгоритм 13. *Построение паросочетания наибольшего веса в двудольном графе* $G = (A, B, E)$ *с заданными весами вершин доли* A

1 Упорядочить элементы множества A по убыванию весов:
$A = \{a_1, a_2, \ldots, a_k\}$, $w(a_1) \geq w(a_2) \geq \ldots \geq w(a_k)$
2 $X := \varnothing$
3 $M := \varnothing$
4 **for** $i = 1$ **to** k **do**
if в G существует увеличивающая цепь P относительно M, начинающаяся в вершине a_i
then $\{X := X \cup \{a_i\}; M := M \otimes P\}$

Если для поиска увеличивающей цепи применить метод поиска в ширину, как описано выше, то время поиска будет пропорционально числу ребер. Общая трудоемкость алгоритма будет $O(mk)$, где k – число ребер в доле A.

Упражнения

1. Реализуйте описанный выше алгоритм нахождения наибольшего независимого множества на основе поиска в глубину в дереве подзадач и модифицированный алгоритм с применением сжатия по включению к ка-

ждой подзадаче. Сравните экспериментально время работы двух алгоритмов на случайных графах.

2. Реализуйте алгоритм нахождения всех максимальных независимых множеств на основе поиска в глубину в дереве вариантов.

3. Разработайте алгоритм поиска поглощающих вершин и нахождения наибольшего независимого множества в хордальном графе.

4. Задан неориентированный граф со взвешенными ребрами и множество выделенных вершин в нем. Требуется построить лес минимального веса, в котором каждая компонента связности содержала бы точно одну из выделенных вершин. Покажите, что эта задача сводится к построению оптимального каркаса. Постройте алгоритм для ее решения (на основе какого-либо известного алгоритма для задачи об оптимальном каркасе).

5. Разработайте вариант алгоритма Прима с использованием приоритетной очереди, как описано в разделе 3.4. Как оценивается время работы этого алгоритма, если для реализации приоритетной очереди используется бинарная куча?

Часть 2. МОДЕЛИ ВЫЧИСЛЕНИЙ

Исторические сведения

К концу XIX – началу XX века в математике накопилось некоторое количество вычислительных задач, для которых математики, несмотря на упорные попытки, не могли предложить методов решения. Одной из таких задач является задача о разрешимости диофантова уравнения. В докладе Д. Гильберта, прочитанном на II Международном конгрессе математиков в августе 1900 года, она звучит следующим образом:

«Пусть задано диофантово уравнение с произвольными неизвестными и целыми рациональными числовыми коэффициентами. Указать способ, при помощи которого возможно после конечного числа операций установить, разрешимо ли это уравнение в целых рациональных числах».

Неудачные попытки математиков решить эту и другие подобные задачи привели к мысли о том, что может и не существовать метода их решения. Но, чтобы доказывать подобного рода утверждения, необходимо иметь математическое определение метода, то есть для интуитивных понятий разрешимости и вычислимости необходимо иметь их формальные эквиваленты. Одним из важнейших достижений математики XX века является формирование математических понятий, которые раскрывают сущность интуитивных представлений о том, что такое метод (алгоритм) решения той или иной задачи, что такое вычислимая функция.

Важность этих понятий вытекает не только из общенаучных проблем развития математики, но также из практических задач общества, использующего вычислительную технику в производстве, экономике, инженерных расчетах и нуждающегося в адекватном представлении о возможностях вычислительных автоматов. Приведем краткие исторические сведения о возникновении теории алгоритмов.

В 1932–1935 годах А. Черч и С.К. Клини ввели понятие λ-определимой функции, которое сыграло важную роль в определении объема интуитивного понятия вычислимой функции.

В 1934 году К. Гедель, на основе идей Дж. Эрбрана, рассмотрел класс функций, названных общерекурсивными, а в 1936 году А. Черч и С.К. Кли- ни доказали, что этот класс совпадает с классом λ-определимых функций.

В 1936 году А. Тьюринг ввел свое понятие вычислимой функции, а в 1937 году доказал, что оно совпадает с понятием λ-определимой функции.

В 1943 году Э. Пост, основываясь на своей неопубликованной работе 1920–1922 годов, выдвинул еще один формальный эквивалент понятия вычислимой функции.

Еще одну формулировку дает теория алгоритмов Маркова (1951 г.).

Любое из упомянутых здесь уточнений интуитивного понятия вычислимой функции можно принять за математическое определение класса вычислимых функций. Основными доводами для такого принятия являются эквивалентность различных формулировок и естественно-исторический опыт, показывающий, что все изучавшиеся до сих пор функции, которые принято считать вычислимыми, являются таковыми в смысле любого из упомянутых выше определений.

Еще один подход к моделированию вычислений развивается в рамках так называемого логического программирования. При таком подходе условия решаемой задачи и ее цель формулируются с помощью предложений формального логического языка. Затем осуществляется автоматический поиск доказательства цели, сопровождаемый вычислением неизвестных в задаче величин.

Заметим, что разрабатываемые в настоящее время алгоритмические языки, составляющие математическое обеспечение современных вычислительных устройств, также можно использовать для определения понятия вычислимости. Более того, можно проследить тесную связь между упомянутыми здесь теоретическими моделями вычислений и реальным программированием. Так, λ-исчисление Черча является прообразом функционального программирования, реализованного в известном программистам языке ЛИСП, разработанном в 1961 году Дж. Маккарти, а модель Поста содержит идеи, реализованные в операторных языках типа Фортран, Алгол. Методы логического программирования реализованы в настоящее время в нескольких версиях языка Пролог.

Однако реальные языки программирования из-за своей громоздкости и избыточности выразительных средств мало пригодны для теоретического анализа понятия вычислимости. Отметим также модели вычислительных устройств, получившие название РАМ (равнодоступная адресная машина) и РАСП (равнодоступная адресная машина с хранимой программой). Эти модели в большей степени, чем, например, модель Тьюринга, отражают структуру современных вычислительных устройств.

Глава 1. ТЬЮРИНГОВА МОДЕЛЬ ПЕРЕРАБОТКИ ИНФОРМАЦИИ

Описанная ниже модель несущественно отличается от модели, предложенной Тьюрингом.

Представление информации (модель памяти). Считаем, что информация представляется словами, то есть конечными последовательностями, составленными из букв конечного алфавита, и записывается на неограниченной в обе стороны ленте, разделенной на ячейки. Слово записывается в идущих подряд ячейках по одной букве в ячейке.

В ячейку может быть ничего не записано, в этом случае говорим, что ячейка содержит пробел. Для обозначения пробела используем символ *. Конечную последовательность, составленную из символов алфавита $A = \{a_1, a_2, \ldots, a_t\}$ и символа пробела, называем псевдословом. Считаем, что слева от первой буквы псевдослова и справа от последней записаны пробелы, кроме того, один из символов псевдослова будем помечать стрелкой. Множество всех конечных последовательностей символов из алфавита A обозначается через A^*.

Если псевдослово имеет вид $X*u_t*u_{t-1}*\ldots*\overset{\downarrow}{u_1}*$, где $u_i \in A^*$, $X \in (A \cup \{*\})^*$, то u_1 называем его первым словом, u_2 – вторым и т.д. Слова u_i могут быть и пустыми. Пустые слова не занимают места на ленте. При необходимости будем считать, что между двумя идущими подряд пробелами записано пустое слово.

Поскольку на ленте в каждый момент времени будет находиться не более чем конечное число символов, отличных от пробела, постольку для любого n в псевдослове будет определено его n-е слово, возможно пустое.

Преобразователь информации. Преобразователь информации можно представить как некоторое устройство, снабженное головкой, обозревающей в каждый момент времени одну из ячеек ленты, которое по заранее намеченному плану (программе) может выполнять операции следующего вида:

- напечатать один из символов алфавита в обозреваемой ячейке;
- сдвинуть головку по ленте на одну ячейку влево;
- сдвинуть головку по ленте на одну ячейку вправо;
- ничего не делать до следующего такта времени.

Определение программы. Программу преобразования информации будем представлять в виде ориентированного графа, вершины которого помечены символами из множества $A \cup \{*, r, l, s\}$, а дуги – символами из множества A так, что разным дугам, выходящим из одной вершины, приписаны разные символы. Одна вершина графа выделена в качестве входной, на рисунках будем отмечать ее входящей стрелкой. Предполагаем, что $A \cap \{*, r, l, s\} = \varnothing$.

Действие программы осуществляется следующим образом. В начальный момент головка вычислительного устройства обозревает одну из ячеек ленты. Просматривается входная вершина программы. Если ей приписан символ r, l или s, то головка вычислителя сдвигается по ленте на одну ячейку соответственно вправо, влево или остается на месте; если же ей приписан символ из алфавита A или $*$, то этот символ печатается в обозреваемой ячейке, старое содержимое ячейки при этом стирается. После того как выполнено действие, соответствующее вершине q, в графе отыскивается выходящая из q дуга, помеченная той буквой, которая находится в данный момент в обозреваемой ячейке. Следующим выполняется действие, соответствующее вершине, в которую ведет найденная дуга. Процесс продолжается до тех пор, пока не будет достигнута вершина, из которой не выходит дуга, помеченная буквой, обозреваемой в данный момент. Если такой момент никогда не наступает, то программа работает бесконечно долго.

Вершину v, для которой найдется хотя бы одна буква из A, не используемая в качестве метки на дугах, выходящих из v, будем называть выходной. Выходные вершины будем помечать выходящими в «никуда» стрелками, помеченными буквами, не использованными на других дугах.

Согласно данному описанию, программу можно задать как набор:

$$P = (Q, A, q_0, \varphi, \psi),$$

в котором Q – множество вершин графа; A – алфавит символов, печатающихся на ленте; q_0 – входная вершина ($q_0 \in Q$); φ – отображение Q в $A \cup \{*, r, l, s\}$; ψ – частичное отображение $A \times Q$ в Q.

Чтобы не загромождать чертежи большим количеством стрелок и надписей при изображении программ, мы используем следующие соглашения. Если из вершины q в вершину q' ведет несколько дуг, будем заменять их одной дугой с надписанными над ней буквами, соответствующими заменяемым дугам; одну из дуг, выходящих из данной вершины, будем ос-

тавлять неподписанной, считая при этом, что она помечена всеми буквами алфавита A, которые не использованы на других дугах, выходящих из вершины q. Такая дуга может оказаться единственной, выходящей из вершины q. Ввиду большой близости введенного нами понятия программы с понятием машины Тьюринга будем называть наши программы тьюринговыми. Множество выходных вершин программы P обозначим через V.

Пример. Пусть $A = \{0, 1\}$ и на ленте записано псевдослово $*\overset{\downarrow}{a_1}a_2...a_k*$, где $a_i \in A$, $k \geq 1$, а стрелка над символом $*$ показывает положение головки в начальный момент. Рассматривая слово $a_1a_2...a_k$ как двоичную запись натурального числа n, составить программу, которая на ленте оставляет псевдослово $*\overset{\downarrow}{b_1}b_2...b_s*$ являющееся двоичной записью числа $n + 1$. Нетрудно увидеть, что поставленную задачу решает программа, представленная на рис. 1.

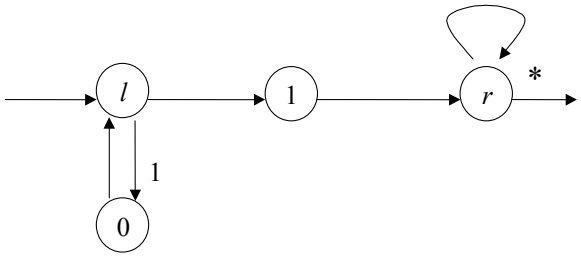


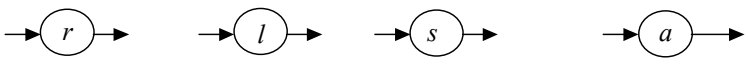
Рис. 1

Здесь входная и выходная вершины помечены соответственно входящей и выходящей стрелками.

1.1. Алгебра тьюринговых программ

Для записи программ в виде формульных выражений введем обозначения для некоторых элементарных программ и операций, позволяющие строить из уже построенных программ более сложные.

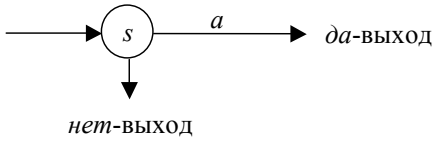
Элементарными вычисляющими программами будем называть программы вида



обозначать их будем соответственно символами *r, l, s, a,* где $a \in A$.

Программы, у которых множество выходов разбито на два непустых подмножества: подмножество *да*-выходов и подмножество *нет*-выходов, назовем бинарными распознающими программами.

Элементарными распознающими программами будем считать программы вида



обозначать такую программу будем через $< a >$.

Правила композиции. Введем несколько правил, которые позволят нам из уже построенных программ строить более сложные.

1. Если $T_1, T_2, ..., T_k$ – программы, то выражение $[T_1, T_2, ..., T_k]$ обозначает программу, которая получена следующим образом. Все выходы программы T_i соединены дугой с входом программы T_{i+1} ($i = 1, 2, ..., k - 1$). Каждая такая дуга помечена буквами из A, которые не использованы на других дугах, выходящих из рассматриваемой выходной вершины (в дальнейшем при соединении выходов одной программы с входом другой будем пользоваться этим правилом). Входом в полученную программу является вход программы T_1, а выходами – выходы программы T_k. Таким образом, программа $[T_1, T_2, ..., T_k]$ предписывает последовательное выполнение программ $T_1, T_2, ..., T_k$.

2. Если P – бинарная распознающая программа, а T – произвольная, то выражение

$$(\text{если } P)\ T$$

означает программу, полученную следующим образом. Все *да*-выходы программы P соединяются с входом программы T. Входом в полученную программу является вход в программу P, а выходом – выходы программы T и *нет*-выходы программы P. Программы такого вида называются охраняемыми, и в таких случаях говорят, что программа T охраняется программой P.

3. Если дан набор охраняемых программ вида: (если P_i) T_i ($i = 1, 2, ..., k$), то выражение вида

$$(\text{если } P_1)T_1 \vee (\text{если } P_2)T_2 \vee ... \vee (\text{если } P_k)T_k$$

обозначает программу, полученную следующим образом. *Да*-выходы программы P_i соединяются с входом T_i ($i = 1, 2, ..., k$); *нет*-выходы программы P_i соединяются с входом T_{i+1} ($i = 1, 2, ..., k - 1$). Входом в полученную программу является вход в программу P_1, а выходом – выходы

программ T_1, T_2, ..., T_k и *нет*-выходы программы P_k.

4. Если P – бинарная распознающая программа, а T – любая, то выражение

$$(\text{пока } P)T$$

обозначает программу, полученную следующим образом. *Да*-выходы программы P соединяются с входом программы T. Все выходы программы T соединены с входом в P. Входом в полученную программу является вход в P, а выходом – *нет*-выходы программы P.

5. Если P – бинарная распознающая программа, а T – любая, то выражение

$$T(\text{до } P)$$

обозначает программу, полученную соединением *нет*-выходов программы P с входом в T, а выходов T – с входом в P. Входом в полученную программу является вход в T, а выходами – *да*-выходы программы P.

Сокращения. Программы вида

$$T_1 \vee T_2 \vee \ldots \vee T_k$$

будем сокращенно записывать в виде

$$\bigcup_{i=1}^{k} T_i \ ,$$

а программы вида

$$[T_1, T_2, \ldots, T_k]$$

в случае, когда $T_1 = T_2 = \ldots = T_k = T$ – в виде T^k.

В контексте со словами «если», «пока», «до» угловые скобки в записи элементарного распознающего оператора $<a>$ будем опускать.

1.2. Начальное математическое обеспечение

Приведем несколько программ, для которых введем обозначения и которые в дальнейшем используем для построения более сложных программ. Они будут составлять начальное математическое обеспечение программирования.

В таблице приведены их схемы в предположении, что алфавит A состоит из символов a_1, a_2, ..., a_t; а символ * обозначен через a_0.

Кроме того, считаем, что X и Y – произвольные псевдослова над алфавитом A; u_1, u_2, ..., u_n, u – слова в алфавите A; a – произвольный символ из

$A \cup \{*\}$; u^{-1} – слово, полученное из слова u изменением порядка символов на противоположный; $n = 1, 2, \ldots$.

Программы R и L, описанные в начале таблицы, используются в последующих программах.

Таблица

Сдвиг головки влево до ближайшего пробела. Обозначение L

Вход	$X*u\overset{\downarrow}{a}Y$
Выход	$X\overset{\downarrow}{*}ua\ Y$
Программа	l (до $*$)

Сдвиг головки вправо до ближайшего пробела. Обозначение R

Вход	$\overset{\downarrow}{X}au*Y$
Выход	$Xua\overset{\downarrow}{*}Y$
Программа	r (до $*$)

Копирование n-го слова. Обозначение K_n

Вход	$X*u_n*u_{n-1}\ldots*\overset{\downarrow}{u_1}*$
Выход	$X*u_n*u_{n-1}\ldots*u_1*\overset{\downarrow}{u_n}*$
Программа	L^n, r, [$\cup_i$ (если a_i) [$*$, R^{n+1}, a_i, L^{n+1}, a_i, r]] (до $*$), R^n

Удаление буквы со сдвигом. Обозначение S

Вход	$\overset{\downarrow}{X}au*Y$
Выход	$Xu\overset{\downarrow}{*}*Y$
Программа	[r, $\cup_i$ (если a_i) [l, a_i], r] (до $*$)

Циклический сдвиг n слов. Обозначение Z_n

Вход	$X*u_n*u_{n-1}*\ldots*\overset{\downarrow}{u_1}*$
Выход	$X*u_{n-1}*\ldots*u_1*\overset{\downarrow}{u_n}*$
Программа	R, [L^{n+1}, r, $\cup_i$ (если a_i) [S^n, a_i]] (до $*$)

Удаление n-го слова. Обозначение Λ_n

Вход	$X*u_n*u_{n-1}*\ldots*\overset{\downarrow}{u_1}*$
Выход	$X*u_{n-1}*\ldots*\overset{\downarrow}{u_1}*$
Программа	Z_n, [$*$, l] (до $*$)

1.3. Методика доказательства правильности программ

Рассматриваемая методика предназначена для доказательства правильности алгоритмов, представленных в виде графов, вершинам которых поставлены в соответствие операторы над памятью, а дугам – переходы от оператора к оператору. Одну из вершин назовем входной, ей соответствует оператор, с которого начинается выполнение алгоритма, а выходных вершин может быть несколько. Считаем, что входная и выходная вершины помечены, соответственно, входящей и выходящей стрелками. Такие представления алгоритмов называют блок-схемами. Доказать правильность алгоритма – это значит доказать утверждение вида:

> «Если входные данные удовлетворяют входному условию, то алгоритм через конечное число шагов завершает работу и выходные данные удовлетворяют требуемому выходному условию».

На практике такое утверждение часто разбивают на два.

(1) «Если входные данные удовлетворяют входному условию и алгоритм через конечное число шагов завершает работу, то выходные данные удовлетворяют требуемому выходному условию».

(2) «Если входные данные удовлетворяют входному условию, то алгоритм через конечное число шагов завершает работу».

Алгоритм, для которого доказано утверждение (1), называется частично правильным или частично корректным. Если же доказаны утверждения (1) и (2), то алгоритм называется правильным или корректным.

Заметим, что когда доказательство утверждения (2) представляет непреодолимые трудности, то ограничиваются доказательством утверждения (1). Таковы, например, итерационные алгоритмы, для которых неизвестна область сходимости. В таком случае, если алгоритм в приемлемое время завершает свою работу, то правильность ответа гарантируется.

Остановимся на доказательстве частичной корректности.

Методика заключается в следующем.

1. Для контроля над ходом вычислений выбираются так называемые контрольные дуги. К числу контрольных обязательно относят входную и все выходные дуги, а также некоторое количество других дуг так, чтобы в граф-схеме алгоритма оказались «разрезанными» все циклы.
2. Для каждой контрольной дуги формулируется индуктивное условие, которому предположительно должно удовлетворять содержимое памяти алгоритма при каждом его прохождении через рассматриваемую дугу. Считаем, что все контрольные дуги (в дальнейшем будем называть их контрольными точками) и соответст-

вующие им индуктивные утверждения пронумерованы.

3. Для каждой пары i, j контрольных точек, для которых в блок-схеме имеется путь из i в j, минующий другие контрольные точки, выбираются все такие пути и для каждого выбранного пути доказывается утверждение (индуктивный шаг): «Если при очередном проходе через точку i выполнялось индуктивное предположение P_i и если реализуется рассматриваемый путь, то при достижении точки j будет выполняться условие P_j».

Если все индуктивные шаги доказаны, то, используя принцип математической индукции, можно утверждать частичную корректность алгоритма. Для доказательства полной корректности остается доказать завершаемость программы через конечное число шагов.

1.4. Вычислимость и разрешимость

Упорядоченный набор из n слов в алфавите A называется n-местным набором над A. Множество всех n-местных наборов над A обозначим через $(A^*)^n$.

Любое подмножество R множества $(A^*)^n$ называется n-местным словарным отношением.

Любое, возможно частичное, отображение f: $(A^*)^n \to A^*$ называется n-местной словарной функцией. Область определения функции f обозначается через $\mathrm{Def}(f)$.

Результатом работы программы T на входном псевдослове x называется псевдослово $T(x)$, которое появляется на ленте в момент остановки программы; если программа работает бесконечно, то результат неопределен.

Программу, которая в процессе работы над любым псевдословом x не сдвигает головку левее пробела, расположенного слева от n-го слова псевдослова x, будем называть n-программой.

Словарное n-местное отношение R называется *полуразрешимым*, если существует n-программа T, которая останавливается в точности на всех псевдословах, имеющих вид

$$X \# u_n \# u_{n-1} \# \ldots \# u_1 \overset{\downarrow}{\#},$$

где $(u_1, u_2, \ldots, u_n) \in R$.

Словарное n-местное отношение R называется *разрешимым*, если R и $\overline{R}$ полуразрешимы (под $\overline{R}$ здесь понимается множество $(A^*)^n \setminus R$).

Словарная n-местная функция $f\colon (A^*)^n \to A^*$ называется вычислимой по Тьюрингу, если существует программа T такая, что

$$T(*u_1*u_2*\ldots*u_n\overset{\downarrow}{*}) = *u_1*u_2*\ldots*u_n*\overset{\downarrow}{v}*,$$

где $u_1, u_2, \ldots, u_n \in \mathrm{Def}\,(f)$ и $v = f(u_1, u_2, \ldots, u_n)$, в противном случае результат не определен.

Вычислимые по Тьюрингу функции уместно было бы назвать полувычислимыми, а полувычислимые с разрешимой областью определения – вычислимыми, но это противоречит установившимся традициям.

1.5. Вычисление числовых функций

Чтобы вычислять значения числовых функций с помощью тьюринговых программ, необходимо выбрать способ кодирования на ленте аргументов и значений функции. Мы рассматриваем функции из N^n в N, где N – множество натуральных чисел, включая 0, а $n \geq 1$.

Значения функции и ее аргументов будем записывать в бинарном, унарном или каком-либо ином коде, для этого нам потребуется соответственно алфавит $A = \{1\}$, $A = \{0, 1\}$ и т.д. Значения аргументов перед вычислением должны быть представлены на ленте в виде псевдослова

$$*x_n*x_{n-1}*\ldots*\overset{\downarrow}{x_1}*,$$

где x_i – код i-го аргумента ($i = 1, 2, \ldots, n$).

После вычисления содержимое ленты должно иметь вид:

$$*x_n*x_{n-1}*\ldots*x_1*\overset{\downarrow}{y}*,$$

где y – код значения функции при заданных значениях аргумента.

Упражнения

1. Составить программу, перерабатывающую псевдослово $*\overset{\downarrow}{u}*$ в псевдослово $*u*\overset{\downarrow}{v}*$, где u – бинарный, а v – унарный код некоторого числа изN.

2. Составить программу сложения и умножения чисел в унарном и бинарном кодах.

3. Составить программу для удвоения числа в бинарном и унарном кодах.

4. Составить программу деления нацело натуральных чисел в унарном коде.

1.6. Частично-рекурсивные функции

Пытаясь выяснить содержание интуитивного понятия вычислимой

функции, А. Черч в 1936 году рассмотрел класс так называемых рекурсивных функций, а Клини расширил его до класса частично-рекурсивных функций. В то же время впервые была высказана естественно-научная гипотеза о том, что интуитивное понятие вычислимой частичной функции совпадает с понятием частично рекурсивной функции. Эту гипотезу называют тезисом Черча. Здесь мы напомним понятие частично-рекурсивной функции и покажем, что любая частично-рекурсивная функция вычислима по Тьюрингу. Набор аргументов $x_1, x_2, ..., x_m$ обозначим через **x**.

Функция $f(\mathbf{x})$ называется *суперпозицией* n-местных функций $g_1, g_2, ..., g_m$ и m-местной функции h, если

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), g_2(\mathbf{x}), ..., g_m(\mathbf{x})).$$

Говорят, что $(n + 1)$-местная функция $f(\mathbf{x}, y)$ получена *примитивной рекурсией* из $(n + 2)$-местной функции g и n-местной функции h, если

$$f(\mathbf{x}, y) = \begin{cases} h(\mathbf{x}) & \text{при} \quad y = 0; \\ g(f(\mathbf{x}, y-1), \mathbf{x}, y-1), & \text{при} \quad y > 0. \end{cases}$$

Говорят, что n-местная функция $f(\mathbf{x})$ получена *минимизацией* из $(n + 1)$-местной функции g, если

$$f(\mathbf{x}) = \begin{cases} k, \text{если } g(k, \mathbf{x}) = 0 \text{ и при всех } k' < k \; g(k', \mathbf{x}) \text{ определена и не равна 0,} \\ \text{не определена в противном случае.} \end{cases}$$

Часто обозначают $f(\mathbf{x})$ через $\mu y(g(y, \mathbf{x}) = 0)$.

Заметим, что суперпозиция и примитивная рекурсия, примененные к всюду определенным функциям, дают всюду определенные функции, тогда как минимизация, примененная к всюду определенной функции, может дать частичную функцию.

Числовая функция f: $N^n \to N$ называется частично рекурсивной, если она является одной из базисных функций:

а) $O(y) = 0$ (при всех $y \in N$),

б) $S(y) = y + 1$ (при всех $y \in N$),

в) $I_m^n(x_1, x_2, ..., x_n) = x_m$ $(n = 1, 2,...; \; 1 \le m \le n)$

или получена из них с помощью конечного числа применений суперпозиции, примитивной рекурсии и минимизации.

Теорема. *Любая частично-рекурсивная функция вычислима по Тьюрингу.*

Доказательство теоремы заключено в следующих четырех леммах, с использованием унарного кодирования чисел.

Лемма 1 *(о базисных функциях). Базисные функции* $O(y)$, $S(y)$, $I_m^n(x_1, x_2, ..., x_n)$ *вычислимы по Тьюрингу.*

Действительно, функцию $S(y)$ вычисляет программа $[K_m, 1, r]$, функцию $O(y)$ – программа $[r]$, функцию $I_m^n(x_1, x_2, ..., x_n)$ – программа K_m.

Лемма 2 *(о суперпозиции). Если функции* $g_1(\mathbf{x})$, $g_2(\mathbf{x})$, ..., $g_m(\mathbf{x})$ *и* $h(y_1, y_2, ..., y_m)$ *вычислимы, соответственно, программами* G_1, G_2 ,..., G_m *и* H, *то функцию*

$$f(\mathbf{x})=h(g_1(\mathbf{x}), g_2(\mathbf{x}),..., g_m(\mathbf{x}))$$

вычисляет программа:

$$[G_m, (Z_{n+1})^n, G_{m-1}, (Z_{n+1})^n, ..., G_1, (Z_{n+1})^n, (Z_{n+m})^n, H, (\Lambda_2)^m].$$

Чтобы убедиться в справедливости этого утверждения, достаточно выписать псевдослова, которые появляются на ленте после выполнения отдельных частей программы. Сначала на ленте находится псевдослово

$$*x_n*x_{n-1}*...*\overset{\downarrow}{x_1}*,$$

после выполнения G_m на ленте будет

$$*x_n*x_{n-1}*...*x_1*\overset{\downarrow}{g_m}*,$$

после $[G_m, (Z_{n+1})^n]$ –

$$*g_m*x_n*x_{n-1}*...*\overset{\downarrow}{x_1}*,$$

после $[G_m, (Z_{n+1})^n, G_{m-1}]$ –

$$*g_m*x_n*x_{n-1}*...*x_1*\overset{\downarrow}{g_{m-1}}*,$$

и т.д.

после $[G_m, (Z_{n+1})^n, G_{m-1}, (Z_{n+1})^n, ..., G_1, (Z_{n+1})^n]$ –

$$*g_m*g_{m-1}*...*g_1*x_n*x_{n-1}*...*\overset{\downarrow}{x_1}*,$$

после $[G_m, (Z_{n+1})^n, G_{m-1}, (Z_{n+1})^n, ..., G_1, (Z_{n+1})^n, (Z_{n+m})^n, H]$ –

$$*x_n*x_{n-1}*...*x_1*g_m*g_{m-1}*...*g_1*\overset{\downarrow}{f}*,$$

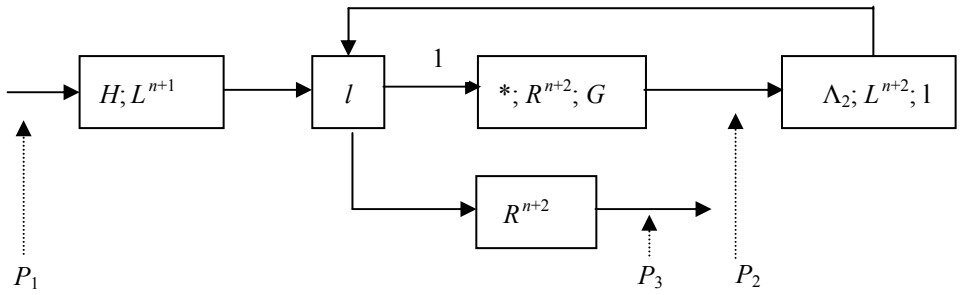
и, наконец, после выполнения всей программы получим

$$*x_n*x_{n-1}*...*x_1*\overset{\downarrow}{f}*.$$

Лемма 3 *(о примитивной рекурсии). Если функции* $h(x_1, x_2, ..., x_n)$ *и* $g(y_1, y_2, ..., y_{n+2})$ *вычислимы соответственно с помощью программ H и G, то функция* $f(x_1, x_2, ..., x_n, y)$, *полученная по схеме примитивной рекурсии, вычислима программой*

$$[H, L^{n+1}, l, (\text{пока } 1)[*, R^{n+2}, G, \Lambda_2, L^{n+2}, 1, l], R^{n+2}].$$

Доказательство. Представим программу, предлагаемую для вычисления функции $f(x_1, x_2, \ldots, x_n, y)$, блок-схемой, изображенной на рисунке.



Пунктирными стрелками показаны контрольные дуги, для которых будут сформированы соответствующие индуктивные утверждения P_1, P_2, P_3.

Утверждение P_1 соответствует входной дуге и поэтому должно описывать содержимое ленты в начальный момент. Утверждение P_3 соответствует выходной дуге и должно описывать содержимое ленты в момент завершения работы программы. Утверждение P_2 относится к дуге, разрезающей единственный имеющейся в блок-схеме цикл, поэтому должно быть сформулировано так, чтобы ему удовлетворяло содержимое ленты каждый раз, когда в программе реализуются переход по рассматриваемой дуге.

Напомним, что основное требование, предъявляемое к утверждениям P_1, P_2, P_3, заключается в том, чтобы была возможность доказательства индуктивных шагов:

$P_1 \to P_2$, если реализуется путь $[H; L^{n+1}; l; *; R^{n+2}; G]$;
$P_1 \to P_3$, если реализуется путь $[H; L^{n+1}; l; R^{n+2}]$;
$P_2 \to P_2$, если реализуется путь $[\Lambda_2; L^{n+2}; 1; l; *; R^{n+2}; G]$;
$P_2 \to P_3$, если реализуется путь $[\Lambda_2; L^{n+2}; 1; l; R^{n+2}]$.

Пусть $x_1, x_2, \ldots, x_n, y$ – исходные значения аргументов из множества N, тогда требуемые утверждения можно сформулировать следующим обра-

зом:

P_1 – содержимое ленты равно $*1^{y}*1^{x_n}*1^{x_{n-1}}*\ldots*1^{x_1}\overset{\downarrow}{*}$,

P_2 – существует $y_1 \geq 1$, y_2, g_1, $g_2 \geq 0$, такие, что содержимое ленты равно $*1^{y_2}*1^{y_2}*1^{x_n}1^{x_{n-1}}*\ldots*1^{x_1}1^{g_1}1^{g_2}\overset{\downarrow}{*}$ и $y_1+y_2+1 = y$, $g_1 = g(f(x_1, x_2, \ldots, x_n, y_1-1), x_1, x_2, \ldots x_n, y_1-1)$, $g_2 = g(f(x_1, x_2, \ldots, x_n, y_1), x_1, x_2, \ldots, x_n, y_1)$,

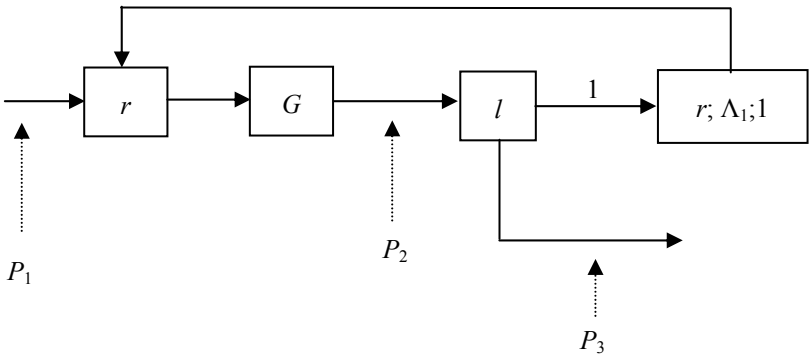
P_3 – содержимое ленты равно $*1^{y}*1^{x_n}*1^{x_{n-1}}*\ldots*1^{x_1}*1^{z}\overset{\downarrow}{*}$ и $z = f(x_1, x_2, \ldots, x_n, y)$.

Доказательство индуктивных шагов легко получить, выписывая содержимое ленты после каждого оператора в соответствующем пути. Читателю предоставляется возможность это проделать и убедиться в правильности программы, предлагаемой в формулировке леммы 3. Не забудьте доказать завершаемость исследуемой программы.

Лемма 4 (*о минимизации*). *Если функция* $g(y, x_1, x_2, \ldots, x_n)$ *вычислима программой* G, *то функция* $f(x_1, x_2, \ldots, x_n)$, *полученная из нее по схеме минимизации*, *вычисляется программой*

$$[r, G, l, (\text{пока } 1)\ [r, \Lambda_1, 1, r, G, l]].$$

Доказательство. Представим программу, предлагаемую для вычисления функции $f(x_1, x_2, \ldots, x_n)$, блок-схемой с указанными контрольными точками P_1, P_2, P_3.



Пусть $x_1, x_2, \ldots, x_n$ – исходные значения аргументов, тогда для доказательства частичной корректности предлагаемой программы можно воспользоваться следующими индуктивными утверждениями:

P_1 – содержимое ленты равно $*1^{x_n}*1^{x_{n-1}}*\ldots*1^{x_1}\overset{\downarrow}{*}$,

P_2 – существуют k, $z \geq 0$, такие, что содержимое ленты равно $*1^{x_n} * *1^{x_{n-1}} *...*1^{x_1} *1^k *1^z \overset{\downarrow}{*}$, где $z = g(k, x_1, x_2, \ldots, x_n)$,

P_3 – содержимое ленты равно $*1^{x_n} *1^{x_{n-1}} *...*1^{x_1} *1^z \overset{\downarrow}{*}$, где $z = f(x_1, x_2, \ldots, x_n)$.

Требуется доказать следующие индуктивные шаги.

$P_1 \to P_2$, если реализуется путь $[r, G]$;

$P_2 \to P_2$, если реализуется путь $[l; r; \Lambda_1; 1; r; G]$;

$P_2 \to P_3$, если реализуется путь $[l]$, и головка остановится на символе *.

Доказательство индуктивных утверждений и завершаемости программы предоставляется читателю в качестве упражнений.

Заметим, что в доказательствах лемм 3 и 4 при рисовании блок-схемы мы несущественно отступили от текстов программ, данных в их формулировках, а при доказательстве леммы 2 не выписывали индуктивных утверждений, так как в представлении программы блок-схемой нет циклов. Обращаем внимание на то, что правильность блоков, из которых составлены программы, мы не подвергаем сомнению.

1.7. Универсальная тьюрингова программа и пример невычислимой функции

В этом параграфе мы объявляем о существовании универсальной тьюринговой программы U, которая может имитировать любую программу, работающую над фиксированным алфавитом $A = \{a_1, a_2, ..., a_n\}$. Эта программа U, получив на входе псевдослово, содержащее в определенном виде код произвольной программы T и псевдослово X, должна оставить на ленте код программы T и псевдослово $T(X)$ – результат работы программы T на псевдослове X.

Читатель, построивший универсальную программу, может считать себя изобретателем компьютера.

Пусть Ψ_n – множество всех функций из N в N таких, что каждая из них определена в точке 0 и вычислима программой, состоящей не более чем из n тьюринговых команд. Рассмотрим функцию $s(n) = \max f(0)$, где максимум берется по всем функциям из Ψ_n. Очевидно, $s(n)$ всюду определена и монотонна. Более того, справедлива

Теорема. *Для любой всюду определенной вычислимой функции $f: N \to N$ существует $k \in N$ такое, что при любом $m \geq k$ выполняется неравенство $f(m) < s(m)$.*

Действительно, пусть $f(n)$ – произвольная всюду определенная функция из N в N, тогда, очевидно, функция

$$F(n) = \max(f(3n), f(3n + 1), f(3n + 2)) + 1$$

будет также всюду определенной и вычислимой. Пусть в унарном коде ее вычисляет программа T_F, состоящая из k команд. Рассмотрим программу $T = [[1, r]^n, T_F]$, которая сначала записывает на ленте число n, а затем работает как T_F. Очевидно, она состоит из $2n + k$ команд и вычисляет некоторую функцию $F' \in \Psi_{2n+k}$.

По определению F' и s имеем $F(n) = F'(0) \le s(2n + k)$. Используя монотонность функции s, получим $F(n) \le s(3n)$ при всех $n \ge k$, следовательно, $f(3n + i) < s(3n + i)$, $(i = 0, 1, 2)$. Отсюда следует, что при $m \ge 3k$ выполняется неравенство $f(m) < s(m)$, что и требовалось доказать.

Следствие. Функция $s(n)$ невычислима, так как она растет быстрее, чем любая вычислимая функция.

Заметим, что при любом фиксированном n значение $s(n)$ можно попытаться вычислить путем перебора всех программ длины $\le n$ и определения для каждой из них времени работы до момента остановки или доказательства ее незавершаемости. Но вопрос о завершаемости программ в общем виде алгоритмически неразрешим. Уточним основные моменты этого утверждения.

Пусть T – тьюрингова программа, работающая в алфавите A, и пусть $\mathrm{kod}(T)$ – слово в алфавите A, кодирующее программу T (на деталях кодирования не останавливаемся).

Программа T называется самоприменимой, если при подаче ей на вход ее собственного кода она через конечное число шагов остановится, в противном случае программа называется несамоприменимой. Пусть M – множество кодов самоприменимых программ.

Теорема. *Множество M алгоритмически неразрешимо*.

Доказательство. Пусть M – алгоритмически разрешимо. Тогда существуют две программы T_1 и T_2 такие, что T_1 останавливается только на словах из M, а T_2 – только на словах из $A^* \setminus M$.

Тогда, если T_2 на своем собственном коде остановится, то $\mathrm{kod}(T_2) \in M$ по определению M, но $\mathrm{kod}(T_2) \notin M$ по определению T_2.

Если же T_2 на своем собственном коде не остановится, то по определению M $\mathrm{kod}(T_2) \notin M$, а по определению T_2 $\mathrm{kod}(T_2) \in M$. Итак, в любом случае имеем противоречие.

Еще одним примером алгоритмически неразрешимого множества является множество M_1 кодов программ, которые останавливаются при пустом входе. Легко показать, что если бы M_1 было разрешимым, то множество M тоже было бы разрешимым.

1.8. Об измерении алгоритмической сложности задач

При практическом решении многих интересных задач с помощью вычислительных автоматов существенную роль играет время работы и объем памяти, которые требуются для их решения соответствующими алгоритмами. Упомянутые характеристики называются, соответственно, временной и пространственной сложностью алгоритма.

Временной и пространственной сложностью задачи в классе алгоритмов (или автоматов) называется время и объем памяти, требующиеся «лучшему» в данном классе алгоритму (автомату) для решения рассматриваемой задачи.

Вопрос о нахождении сложностных характеристик задач весьма труден. Трудности, как правило, связаны с логической сложностью рассматриваемых задач и обилием алгоритмов в любом универсальном классе.

Рассмотрим некоторые подробности на примере тьюринговых программ. Считаем, что задача представлена двухместным словарным предикатом $R(u, v)$ над некоторым алфавитом A и заключается в нахождении по заданному слову $u \in A^*$ слова $v \in A^*$ такого, что $R(u, v)$ истинно.

Слово v будем считать решением задачи R при входном слове u. Чтобы пустоту слова v трактовать как отсутствие решения, наложим на R следующие ограничения

$$R(\lambda, \lambda),$$

$$\forall u \, \exists v \, R(u, v),$$

$$\forall u \, [R(u, \lambda) \to \forall v \, [v \neq \lambda \to \neg R(u, v)]].$$

Результат работы тьюринговой программы T на входном слове u обозначим $T(u)$, считая $T(u)$ равным выходному слову.

Будем говорить, что тьюрингова программа T решает задачу R, если на любом входном слове u она останавливается через конечное число шагов и $\forall u \, R(u, T(u))$.

Через $\mathrm{time}(T, u)$ обозначим число элементарных тьюринговых команд, которые будут выполнены программой T от начального момента до момента остановки при работе на входном слове u. Если при входном слове u программа T выполняет бесконечное число шагов, то считаем

$\text{time}(T, u) = \infty$.

Величину $\text{time}(T, u)$ будем называть временем работы программы T на слове u. В большинстве случаев эта величина существенно зависит от длины слова u, поэтому представляет интерес функция $t(T, n) = \max \text{time}(T, u)$, где максимум вычисляется по всем словам длины n.

Заметим, что эта ситуация не является общей; в некоторых случаях величина $\text{time}(T, u)$ не зависит от длины слова u. Например, пусть требуется определить четность числа, представленного в двоичном коде. Для этого достаточно посмотреть на его младший разряд.

Временной сложностью задачи R можно было бы попытаться назвать функцию $f(n) = \min t(T, n)$, где минимум вычисляется по всем программам T, решающим задачу R. Однако существование такой функции не гарантировано. Можно надеяться на существование таких функций при ограничениях на класс вычислительных алгоритмов. На практике ограничиваются нахождением верхней и нижней оценочных функций для времени работы конкретных алгоритмов.

Заметим, что получение нижних нетривиальных оценок каждый раз представляет собой сложную математическую задачу. Известно, например, что временная сложность задачи распознавания симметрии слова тьюринговыми программами оценивается снизу функцией cn^2, где n – длина слова, а c – некоторая константа.

Для многих интересных с практической точки зрения задач известные верхние оценки носят экспоненциальный характер, что часто свидетельствует о больших затратах времени при выполнении соответствующих алгоритмов на вычислительных установках.

Если для задачи R имеется полиномиальная от n верхняя оценочная функция, то говорят, что R разрешима в полиномиальное время.

Для многих задач не удается установить существования верхних полиномиальных оценочных функций. Однако распознавание на паре слов u, v, является ли слово v решением задачи R на входе u, решается за полиномиальное от длины слова u время и, кроме того, для каждого u существует ответ v, длина которого ограничена некоторым полиномом от длины u. Это так называемые задачи с проверяемым за полиномиальное время ответом.

Такой задачей является, например, задача о выполнимости булевых формул, которая заключается в следующем. По заданной булевой формуле найти набор значений переменных, при которых соответствующая формуле булева функция принимает значение 1. Имея минимальный про-

граммистский опыт, легко убедиться в возможности проверки ответа к этой задаче за полиномиальное время.

Задачи с проверяемым за полиномиальное время ответом называются переборными с гарантированным экспоненциальным перебором. Действительно, пусть $R(u, v)$ такая задача и длина возможного ответа v ограничена полиномом p от длины входа u, то есть $|v| \le p(|u|)$. Пусть, далее q – полином, являющийся верхней оценкой времени работы программы, проверяющей ответ, тогда, перебирая всевозможные $2^{p(|u|)}$ слов длины $p(|u|)$ и затрачивая на проверку каждого не более $q(|u|)$ тактов времени, получим алгоритм с верхней оценкой $q(u)\cdot 2^{p(|u|)}$.

Задача R полиномиально сводится к задаче R', если существуют работающие полиномиальное время тьюринговы программы T_1 и T_2 такие, что

$$\forall u\ \forall v'\ [R'(T_1(u), v') \to R(u, T_2(v'))].$$

Из этого определения следует, что для решения задачи R на входе u достаточно

- вычислить $u' = T_1(u)$,
- затем найти ответ v' задачи R' на входе u',
- и, наконец, применить к v' программу T_2, получив ответ $v = T_2(v')$ задачи R на входе u.

Таким образом, имеем следующую схему получения ответа v:

$$u \to u' \to v' \to v.$$

Оказывается, что среди задач с полиномиально проверяемым ответом существует задача, к которой полиномиально сводится любая другая задача с полиномиально проверяемым ответом. Такие задачи получили название универсальных переборных задач.

Исторически существование универсальных переборных задач обнаружено в 1971 году американским математиком С.А. Куком, когда он доказал, что задача выполнимости булевой формулы является универсальной переборной задачей.

Тогда же было доказано, что и многие другие широко известные задачи являются универсальными переборными задачами.

Круг таких задач в настоящее время постоянно расширяется. По данному вопросу имеется обширная литература [3].

Глава 2. АБАК

2.1. Основные определения

Абак наряду с машинами Тьюринга является одной из простейших универсальных моделей вычислений. Это числовая модель; элементами информации являются целые неотрицательные числа.

Память представляет собой потенциально бесконечный набор ячеек, каждая ячейка может содержать любое целое неотрицательное число. Считается, что ячейки пронумерованы числами 1, 2, ...

Исполняющее устройство способно выполнять всего две операции (элементарные команды) над числами, это прибавление (increment) и вычитание (decrement) единицы из указанного в команде числа. Команды имеют вид $\mathrm{inc}\,(x)$ и $\mathrm{dec}\,(x)$, где x – номер ячейки. Поскольку в каждом конкретном алгоритме может быть использовано лишь конечное число ячеек и с номерами ячеек никаких операций не производится, мы будем обозначать их в примерах для наглядности отдельными буквами, возможно с использованием индексов.

Программа (алгоритм) – это ориентированный граф, вершинам которого приписаны элементарные команды указанного выше вида. Из каждой вершины, помеченной командой вида $\mathrm{inc}(x)$, выходит одна дуга в вершину со следующей командой. Из каждой вершины, помеченной командой вида $\mathrm{dec}(x)$, выходят две дуги. Одна из них помечается знаком «+» и ведет в вершину, помеченную командой, которая должна выполняться следующей в случае, если перед ее выполнением в ячейке x находилось число, отличное от нуля. Вторая дуга помечается знаком «–» и ведет в вершину, помеченную командой, которая должна выполняться следующей в случае, если перед ее выполнением в ячейке x находилось число нуль. Одна из вершин графа помечается как входная, в нее ведет дуга из «ниоткуда», выходных вершин может быть несколько.

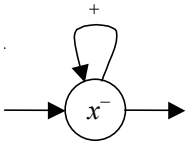
Несмотря на простоту этой модели вычислений, она «эквивалентна по силе» любой из изучавшихся универсальных моделей вычислений и относительно нее можно сформулировать тезис, аналогичный тезису Черча, который может выглядеть следующим образом. Любая интуитивно вычислимая функция может быть вычислена на абаке. При этом, конечно, надо условиться о том, где располагаются входные аргументы и куда следует поместить результат (значение вычисляемой функции).

Примеры программ. В рассмотренных ниже примерах операция $\mathrm{inc}(x)$ обозначается для краткости через x^+, а операция $\mathrm{dec}(x)$ – через x^-. Основные операции изображены кружками. Прямоугольниками изображены операции, для которых в предыдущих примерах построены алгоритмы.

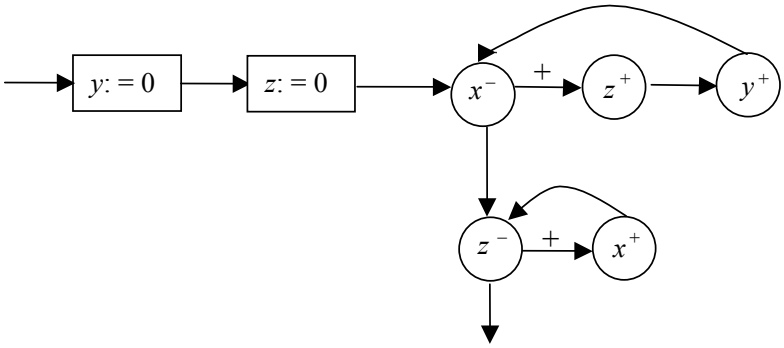
Знак «–» на соответствующих дугах опущен. Сформулируйте инварианты циклов во всех рассмотренных ниже примерах.

1. Программа $x := 0$

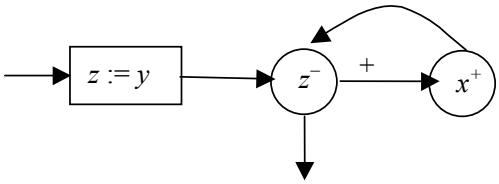
В языках программирования эта операция обычно является элементарной. На абаке это действие можно выполнить следующей программой



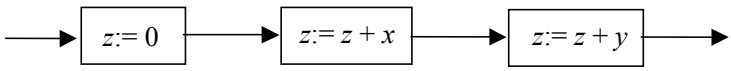
2. Программа $y := x$



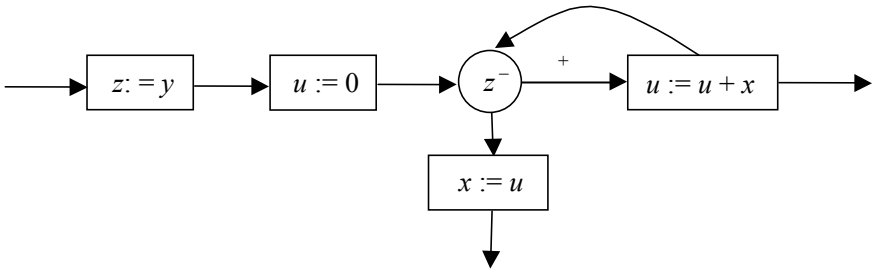
3. Программа $x := x + y$



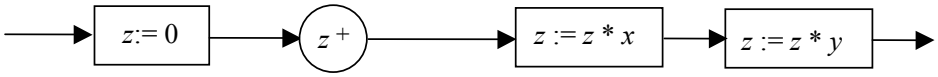
4. Программа $z := x + y$



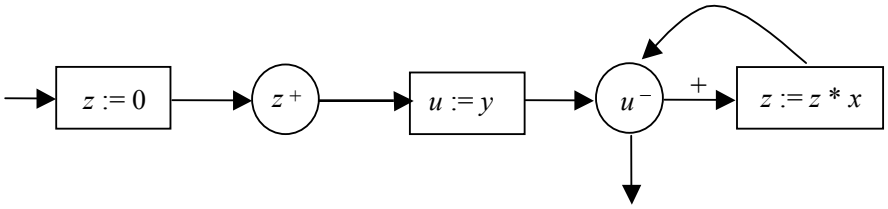
5. Программа $x := x * y$



6. Программа $z := x * y$



7. Программа $z := x^y$



2.2. Примеры неразрешимости

Функцию $p: N \to N$ назовем *вычислимой* на абаке, если существует программа P, которая, получив в некоторой, заранее обусловленной ячейке x значение аргумента n, а в остальных ячейках нули, через конечное число шагов остановится, и в ячейке y будет находиться $p(n)$.

Проблема построения невычислимой функции известна как проблема усердного бобра. Пусть A – абак-программа и y – номер некоторой ячейки. Определим величину $f(A, y)$ следующим образом. Если в начальный момент все ячейки содержат число 0 и программа A через конечное число шагов останавливается, то $f(A, y)$ равно числу $[y]$ в момент остановки. Если же программа работает бесконечно, то считаем $f(A, y) = 0$. Величину $f(A, y)$ назовем y-продуктивностью программы A.

Обозначим через $\tilde{A}(n)$ множество всех абак-программ, состоящих из n команд. Определим функцию $p(n)$ как максимум $f(A, y)$ по всем про-

граммам A из $\tilde{A}$(n) и ячейкам y. Очевидно, эта функция определена при всех натуральных значениях аргумента и строго монотонна.

Лемма. *Для любого натурального числа n выполняется неравенство*

$$p(n + 17) \geq 2n.$$

Для доказательства достаточно рассмотреть программу, которая сначала запишет 0 в ячейку x, затем прибавит к ней n раз 1, скопирует содержимое ячейки x в ячейку y и добавит к ней содержимое ячейки x. Очевидно, после выполнения такой программы в ячейке y будет число $2n$, а подсчет числа команд показывает, что их будет $n + 17$. Наличие такой программы (рис. 1) доказывает требуемое неравенство.

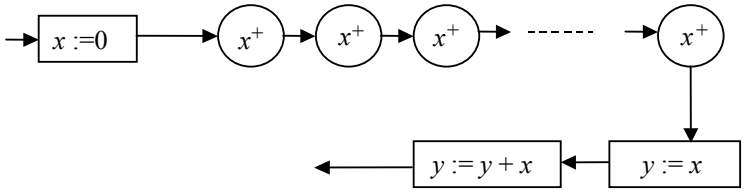


Рис. 1

Предположим теперь, что $p(n)$ вычислима некоторой программой P, состоящей из k команд, которая, получив n в ячейке x, поместит ответ в ячейку y.

Тогда для каждого натурального n можно построить программу, вычисляющую $p(p(n + 17))$, состоящую из $n + 2k + 25$ команд. Эта программа сначала запишет 0 в ячейку x, затем прибавит к ней $n + 17$ раз единицу, затем с помощью программы P в ячейке y вычислит $p(n + 17)$, скопирует y в x и, наконец, опять с помощью программы P вычислит $p(p(n + 17))$.

Наличие такой программы (рис. 2) означало бы, что при любом натуральном n выполняется неравенство

$$p(n + 2k + 25) \geq p(p(n + 17)).$$

Поскольку функция $p(n)$ монотонна, то получаем отсюда

$$n + 2k + 25 \geq p(n + 17).$$

Сопоставляя это неравенство с неравенством в утверждении леммы, получим

$$n + 2k + 25 \geq p(n + 17) \geq 2n,$$

что приводит к противоречию, например, при $n = 2k + 26$.

Итак, предположение о вычислимости функции $p(n)$ привело к противоречию.

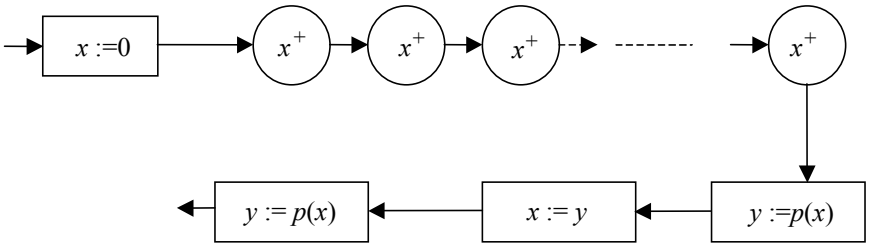


Рис. 2

На рис. 2 команда x^+ повторяется $n + 17$ раз, общее количество команд в программе $n + 2k + 25$, в ячейке y вычисляется $p(p(n + 17))$.

Рассмотрим произвольную инъективную нумерацию g программ, то есть нумерацию, ставящую в соответствие каждой программе P ее номер $g(P)$, причем разным программам ставятся разные номера. Программу назовем самоприменимой относительно ячейки x, если она, получив в ячейке x свой номер, а в остальных ячейках нули, через конечное число шагов завершает вычисления.

Рассмотрим функцию $p: N \to N$, определяемую следующим образом:

$p(n) = 1$, если n является номером некоторой самоприменимой относительно ячейки x программы,

$p(n) = 0$, в противном случае.

Докажем, что функция p не вычислима никакой программой. Предположим, что p вычисляется программой P, которая, получив в ячейке x число n, остановится через конечное число шагов и в ячейке y оставит $p(n)$. Рассмотрим программу P', изображенную на рис. 3

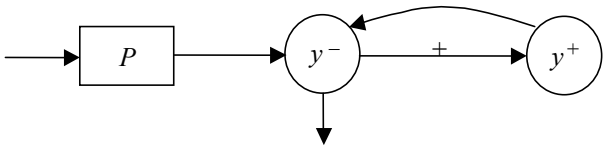


Рис. 3

Если P' самоприменима относительно ячейки x, то $p(g(P')) = 1$, поэтому, когда проработает программа P, в ячейке y будет записана 1 и остальная часть программы P' будет работать без остановки. Следовательно, P' не самоприменима относительно x.

Если же P' не самоприменима относительно ячейки x, то $p(g(P')) = 0$,

следовательно, когда проработает программа P, в ячейке y будет записан 0 и тогда остальная часть программы P' сразу же завершит работу. Следовательно, P' самоприменима относительно x.

Итак, предположение о вычислимости функции $p(n)$ в любом случае приводит к противоречию.

Глава 3. АЛГОРИФМЫ МАРКОВА

Термин «алгорифм» является устаревшим вариантом современного термина «алгоритм», однако по отношению к алгоритмам Маркова принято использовать авторский вариант.

***Информация**,* обрабатываемая алгорифмом Маркова, представляется словом в некотором фиксированном алфавите A.

Алгорифм (программа) представляется последовательностью пар слов в алфавите A. Пары, составляющие алгорифм, называются также подстановками и записываются в виде

$$\alpha \to \beta,$$

где α, β – слова в алфавите A, причем β может быть пустым (обозначаем λ).

Программа имеет вид

$$\alpha_1 \to \beta_1$$

$$\alpha_2 \to \beta_2$$

.....

$$\alpha_i \to \beta_i!$$

.....

$$\alpha_n \to \beta_n.$$

Некоторые подстановки помечаются восклицательным знаком и называются заключительными.

Функционирование. Во входном слове ищется фрагмент, совпадающий с левой частью первой подстановки. Если фрагмент находится, то самый левый такой фрагмент во входном слове заменяется на ее правую часть, в противном случае рассматривается вторая подстановка из алгорифма и так далее. Вычисления заканчиваются, когда ни одна из левых частей подстановок не является фрагментом обработанного к данному моменту слова или когда выполнена заключительная подстановка. Заметим, что описанный таким образом процесс может оказаться и бесконечным.

Замечание. Алгорифмы Маркова составляют теоретическую основу системы программирования, использующую язык РЕФАЛ.

Пример 1. Алфавит $A = \{1, +\}$. Здесь запятая не является символом алфавита.

Рассмотрим программу

1. $1+ \to +1$,
2. $++ \to +$,
3. $+ \to \lambda$!

Убедитесь в том, что она входное слово вида 11...1+11...11 перерабатает в слово 11...1 в котором число символов «1» такое же, как во входном слове. Можно считать, что программа выполняет сложение натуральных чисел, представленных в унарной системе счисления.

Пример 2. Алфавит $A = \{1, *, v, z\}$.

Программа

1. $*11 \to v*1$
2. $*1 \to v$
3. $1v \to v1z$
4. $zv \to vz$
5. $z1 \to 1z$
6. $v1 \to v$
7. $vz \to z$
8. $z \to 1$
9. $1 \to 1$!

Рассмотрим протокол вычислений на входном слове 11*111. Справа указаны применяемые подстановки.

$11*111$	$*11 \to v*1$
$11v*11$	$*11 \to v*1$
$11vv*1$	$*1 \to v$
$11vvv$	$1v \to v1z$
$1v1zvv$	$1v \to v1z$
$v1z1zvv$	$zv \to vz$
$v1z1vzv$	$1v \to v1z$
$v1zv1zzv$	$zv \to vz$
$v1vz1zzv$	$1v \to v1z$

$vv1zz1zzv \quad zv \to vz$

$vv1zz1zvz \quad zv \to vz$

$vv1zz1vzz \quad 1v \to v1z$

$vv1zzv1zzz \quad zv \to vz$

$vv1zvz1zzz \quad zv \to vz$

$vv1vzz1zzz \quad 1v \to v1z$

$vvv1zzz1zzz \quad z1 \to 1z$

$vvv1zz1zzzz \quad z1 \to 1z$

$vvv1z1zzzzz \quad z1 \to 1z$

$vvv11zzzzzz \quad v1 \to v$

$vvv1zzzzzz \quad v1 \to v$

$vvvzzzzzz \quad vz \to z$

$vvzzzzzz \quad vz \to z$

$vzzzzzz \quad vz \to z$

$zzzzzz \quad z \to 1$

$1zzzzz \quad z \to 1$

$11zzzz \quad z \to 1$

$111zzz \quad z \to 1$

$1111zz \quad z \to 1$

$11111z \quad z \to 1$

$111111 \quad 1 \to 1!$

Если считать, что во входном слове закодирована задача умножения 2*3 в унарной системе счисления, то в выходном слове получен ответ 6.

Докажите, что программа дает верный ответ при любом корректном входном слове.

Глава 4. РАВНОДОСТУПНАЯ АДРЕСНАЯ МАШИНА

Равнодоступная адресная машина (РАМ) – это числовая модель вычислительного устройства. Эта модель является наиболее близкой из рассмотренных к реальным вычислительным машинам и позволяет наиболее реалистично применять теоретические оценки сложности алгоритмов к реальным вычислениям.

Память машины состоит из регистров (ячеек). Каждый регистр имеет адрес и может содержать произвольное число. Регистр с номером 0 называется сумматором.

Программа – последовательность пронумерованных команд. Команда имеет вид

<код операции> <операнд>

Коды операций:

Load, Store, Add, Sub, Mult, Div, Read, Write, Jump, JgtZ, Jzero, Halt.

Операнд может быть одного из трех видов

$$= i,\ i,\ *i,$$

где i – натуральное число.

Содержимое регистра с номером i обозначим через $c(i)$. Значение $v(a)$ операнда a определяется в зависимости от его вида следующим образом

$v(a)$ – число i, если a имеет вид $= i$,

$v(a)$ – число $c(i)$, если a имеет вид i,

$v(a)$ – число $c(c(i))$, если a имеет вид $*i$.

При оценке сложности алгоритмов используют в зависимости от обстоятельств разные веса команд. При работе с малыми числами и малым числом регистров реалистичным оказывается так называемый равномерный вес, при котором каждая команда требует единицу времени. При работе с большими числами и большим количеством регистров используется так называемый логарифмический вес команды, при котором время выполнения команды зависит как от значения операнда, так и от значения его адреса.

При определении веса команды используется функция $L: Z \to Z$, выражающая длину записи числа

$$L(i) = \begin{cases} \log(i) + 1 & \text{при} \quad i \neq 0, \\ 1 & \text{при} \quad i = 0. \end{cases}$$

Основание логарифма при получении асимптотических оценок не имеет существенного значения.

Вес $t(a)$ операнда a определяется в зависимости от его вида следующим образом

$t(a) = L(i)$, если a имеет вид $= i$,

$t(a) = L(i) + c(i)$, если a имеет вид i,

$t(a) = L(i) + L(c(i)) + L(c(c(i)))$, если a имеет вид $*i$.

Команда	**Действие**	**Логарифмический вес**
Load(a)	$c(0) := v(a)$	$t(a)$
Store(i)	$c(i) := c(0)$	$L(c(0)) + L(i)$
Store(*i)	$c(c(i)):=c(0)$	$L(c(0)) + L(i) + L(c(i))$
Add(a)	$c(0) := c(0) + v(a)$	$L(c(0)) + t(a)$
Sub(a)	$c(0) := c(0) - v(a)$	$L(c(0)) + t(a)$
Mult(a)	$c(0) := c(0) \cdot v(a)$	$L(c(0)) + t(a)$
Div(a)	$c(0) := c(0)$ div $v(a)$	$L(c(0)) + t(a)$
Read(i)	$c(i)$:= очередное число	$L(i) + L(c(i))$
Read(*i)	$c(c(i))$:= очередное число	$L(i)+L(c(i))+L(c(c(i)))$
Write(a)	печать $v(a)$	$t(a)$
Jump(a)	переход на команду с номером a	1
JgtZ(a)	переход на команду с номером a, если $c(0)>0$	$L(c(0))$
Jzero(a)	переход на команду с номером a, если $c(0)=0$	$L(c(0))$
Halt(a)	конец вычислений	1

Например, команда Add *i имеет логарифмический вес

$$L(c(0)) + L(i) + L(c(i)) + L(c(c(i))).$$

Временная сложность программы определяется как сумма весов всех выполненных команд с учетом их многократного выполнения.

Емкостная сложность программы определяется как сумма по всем регистрам длин максимальных чисел, побывавших в этих регистрах.

Пример. Рассмотрим вычисление n^n. На псевдокоде оно может выглядеть следующим образом.

```
Read(r1);
if r1 < 0 then write (0) else
    {r2:= r1; r3:= r1–1; while r3<0 do {r2:= r2*r1; r3:= r3-1}; write (r2)}.
```

Этот псевдокод легко может быть представлен в виде следующей РАМ-программы.

1. Read 1
2. Load 1
3. JgtZ 6
4. Write = 0
5. Jump 22
6. Load 1
7. Store 2
8. Load 1
9. Sub =1
10. Store 3
11. Load 3
12. JgtZ = 14
13. Jump 21
14. Load 2
15. Mult 1
16. Store 2
17. Load 3
18. Sub = 1
19. Store 3
20. Jump 11
21. Write 2
22. Halt

Когда команда 15 выполняется i-й раз, сумматор содержит n^i, а $r2$ содержит n. Эта команда выполняется $(n - 1)$ раз.

При равномерном весовом критерии суммарное время – $O(n)$.

При логарифмическом весовом критерии суммарное время равно $\Sigma(L(n^i) + L(n))$, где суммирование ведется по $i = 1, 2, \ldots, n$. Поскольку $(L(n^i) + L(n)) \sim (i + 1)\,\mathrm{Log}\,(n)$, получаем

$$\Sigma(L(n^i)+L(n))= O\,(n^2 \log n).$$

Емкостная сложность программы при равномерном критерии равна $O(1)$, при логарифмическом – $O(n \log n)$.

Глава 5. ФОРМАЛЬНЫЕ ЯЗЫКИ

5.1. Основные понятия и обозначения

Алфавит – конечное множество абстрактных символов, как правило, упорядочено, в так называемом алфавитном порядке.

Слово (в алфавите A) – конечная последовательность символов (алфавита A).

Длина слова – количество вхождений символов в слово. Длина слова u, обычно обозначается $|u|$.

Пустое слово – пустая последовательность, то есть последовательность, не содержащая ни одного символа. Пустое слово, соблюдая традиции, часто обозначают греческой буквой λ, полагая при этом, что она не является символом рассматриваемого алфавита.

Слово длины k ($k > 0$) можно отождествить с элементом декартова произведения ($A \times A \times \ldots \times A$), в котором k сомножителей, обозначаемого A^k. При $k = 0$ имеем A^0, состоящее из одного пустого слова; не путать с пустым множеством, обозначаемым знаком ∅.

Замечание. При отождествлении элемента декартова произведения со словом полагаем, что слово составлено из входящих в него символов в соответствующем порядке.

Множество всех слов в алфавите A обозначают

$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots = \bigcup_{i=0}^{\infty} A^i.$$

Множество всех непустых слов в алфавите A обозначают

$$A^+ = A^1 \cup A^2 \cup \ldots = \bigcup_{i=1}^{\infty} A^i.$$

Сверхслово (в алфавите A) – бесконечная последовательность символов (алфавита A).

Формальный язык (в алфавите A) – множество слов (в алфавите A).

Конкатенация слов – двухместная операция над словами, заключающаяся в приписывании второго слова к первому. Результат конкатенации слов u и v обозначается uv.

Начальный фрагмент слова u, имеющий длину $k \le |u|$, называется

префиксом длины k слова u, обозначается $\mathrm{pref}_k\, u$.

Конечный фрагмент слова u, имеющий длину $k \le |u|$, называется *суффиксом* длины k слова u, обозначается $\mathrm{suff}_k\, u$.

Если $k < |u|$, то префикс и суффикс называются собственными. Заметим, что при нашем определении пустое слово λ будет и собственным префиксом и собственным суффиксом любого слова u.

Операции над формальными языками. Поскольку формальные языки являются множествами, то к ним применяются обычные теоретико-множественные операции: объединение, пересечение, дополнение (до множества всех слов в рассматриваемом алфавите). Кроме перечисленных применяются специфические операции – это конкатенация двух языков и итерация языка.

Результатом конкатенации языков L_1 и L_2 является язык $L = \{uv \mid u \in \in L_1,\ v \in L_2\}$, обозначаемый также $L_1 \cdot L_2$. Результат конкатенации k экземпляров языка L обозначим через L^k.

Результатом итерации языка L является язык $L^* = \{u \mid (\exists\, k \ge 0)\ u \in L^k\}$. Заметим, что $L^0 = \{\lambda\}$ и поэтому $\lambda \in L^*$ при любом L.

Замечание. При работе с формальными языками операцию объединения часто обозначают знаком “+”. В следующих тождествах используется именно это соглашение.

Основные тождества. Пусть α, β, γ – произвольные формальные языки над некоторым фиксированным алфавитом, тогда справедливы следующие тождества:

$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma),$$
$$(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma),$$
$$\alpha + \beta = \beta + \alpha,$$
$$\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma,$$
$$(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma,$$
$$\alpha \cdot \varnothing^* = \alpha,$$
$$\alpha \cdot \varnothing = \varnothing,$$
$$\alpha^* = \alpha \cdot \alpha^* + \varnothing^*,$$
$$\alpha^* = (\alpha + \varnothing^*)^*.$$

5.2. Способы задания формальных языков

Прежде всего, для задания формального языка может подойти любое математически корректное определение множества слов в заданном алфавите. Однако если иметь в виду задание, при котором возможно алгоритмическое решение вопроса о принадлежности слова языку, то нужны средства более ограниченные.

Наиболее общим из конструктивных способов задания языков является способ, использующий так называемые формальные грамматики.

Формальной грамматикой для порождения формального языка в алфавите A называется набор

$$G = (A, B, S, P),$$

где A – алфавит терминальных (основных) символов; B – алфавит нетерминальных (вспомогательных) символов, $A \cap B = \varnothing$; S – стартовый символ, $S \in B$; P – конечный набор правил вывода. Каждое правило вывода имеет вид $\varphi \to \psi$, где φ, ψ – слова в объединенном алфавите $A \cup B$, причем φ содержит хотя бы один символ из алфавита B.

Правило $\varphi \to \psi$ применимо к слову u, если φ является фрагментом слова u. Результатом применения этого правила к слову u называется слово v, полученное заменой любого фрагмента φ в слове u на слово ψ.

Если v – результат применения некоторого правила к слову u, то пишем $u \Rightarrow v$.

Если $u \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_n \Rightarrow v$, то пишем $u \Rightarrow\Rightarrow v$.

Язык $L(G)$, порождаемый грамматикой G, определяется следующим образом

$$L(G) = \{v \mid v \in A^*, S \Rightarrow\Rightarrow v\}.$$

Другими словами, $L(G)$ – множество слов в основном алфавите, которые могут быть получены из стартового символа S путем конечного числа применений правил грамматики.

Классификация Хомского

- грамматики типа 0 – это грамматики, не имеющие ограничений на вид правил;
- грамматики типа 1 – это грамматики, в которых правила имеют вид: $\varphi_1 X \varphi_2 \to \varphi_1 v \varphi_2$, где X – нетерминальный символ, а φ_1, φ_2, v – слова в объединенном алфавите. Слова φ_1, φ_2 называются контекстом правила. Эти грамматики (и языки, порождаемые ими) называются контекстными, так как в описанном правиле символ X заменяется словом v, если находится в контексте φ_1, φ_2;

- грамматики типа 2 – это грамматики, в которых правила имеют вид: $X \to v$, где X – нетерминальный символ, а v – непустое слово в объединенном алфавите. Эти грамматики (и языки, порождаемые ими) называются контекстно-свободными;
- грамматики типа 3 – это грамматики, в которых правила имеют вид: $X \to v$, где X – нетерминальный символ, а v может иметь вид либо a, либо aY, где a – символ основного алфавита, а Y – вспомогательного. Языки, порождаемые грамматиками типа 3, называются регулярными.

Известно, что класс языков, задаваемых грамматиками типа 0, является классом рекурсивно перечислимых языков, не совпадающим с классом рекурсивных языков. На языке теории алгоритмов это означает, что не существует алгоритма, который по любой грамматике G типа 0 и любому слову u отвечает на вопрос «$u \in L(G)$?». С другой стороны, существует алгоритм, который, получив на входе грамматику G и слово u, ответит «да», если $u \in L(G)$, в противном случае он либо ответит «нет», либо будет работать бесконечно.

Классы языков, задаваемых грамматиками типа 1, 2, 3, являются классами рекурсивных языков.

Альтернативный способ задания формальных языков – их описание с помощью различных видов автоматов.

Одним из простейших классов языков, имеющих большое прикладное значение, является класс регулярных языков, допускающих описание и с помощью конечных автоматов, и с помощью аналитических выражений.

5.3. Регулярные выражения

Регулярные выражения – это аналитический (формульный) способ задания регулярных языков.

Определение. *Регулярным выражением над алфавитом A называется выражение, построенное по следующим правилам:*

1. $\varnothing$ *– регулярное выражение;*
2. λ *– регулярное выражение;*
3. a *– регулярное выражение, если* $a \in A$*;*
4. $(P \vee S)$ *– регулярное выражение, если P и S – регулярные выражения;*
5. $(P \cdot S)$ *– регулярное выражение, если P и S – регулярные выражения;*
6. P^* *– регулярное выражение, если P – регулярное выражение.*

Регулярное выражение R задает язык $L(R)$ в соответствии со следующими правилами:

- $L(\varnothing)$ – пустой язык;
- $L(\lambda)$ – язык, состоящий из одного пустого слова;
- $L(a)$ – язык, состоящий из одного однобуквенного слова a;
- $L((P \vee S)) = L(P) \cup L(S)$;
- $L((P \cdot S)) = L(P)L(S)$;
- $L(P^*) = (L(P))^*$.

Пример 1. Рассмотрим регулярное выражение $R = (ab \vee ac)^*(a \vee \lambda)$ над алфавитом $A = \{a, b, c\}$. Язык $L(R)$ состоит из слов, в которых на нечетных местах стоит символ a, а на четных b или c.

Замечание 1. В регулярных выражениях вместо знака «$\vee$» часто используют знак «+».

Замечание 2. Если дополнить правила построения регулярных выражений еще двумя правилами

7. $(P \cap S)$ *– регулярное выражение, если* P *и* S *– регулярные выражения;*
8. $\overline{P}$ *– регулярное выражение, если* P *– регулярное выражение, и считать* $L((P \cap S)) = L(P) \cap L(S)$, *а* $L(\overline{P}) = \overline{L(P)}$,

где дополнение берется до множества всех слов в алфавите A, то получим так называемые расширенные регулярные выражения. Если не использовать дополнение, то получим полурасширенное регулярное выражение.

Как увидим в дальнейшем, использование расширенных регулярных выражений не расширяет класса регулярных языков.

Замечание 3. Используя описанную интерпретацию регулярных выражений, мы будем вместо соотношения $u \in L(R)$ писать $u \in R$.

5.4. Решение уравнений в словах

Рассмотрим уравнение вида $X = \alpha \cdot X + \beta$, где α и β – формальные языки над некоторым алфавитом A.

Теорема. *Если* $\lambda \notin \alpha$, *то уравнение* $X = \alpha X + \beta$ *имеет единственное решение* $X = \alpha^*\beta$. *Если* $\lambda \in \alpha$, *то* $X = \alpha^*(\beta + \gamma)$ *будет решением уравнения* $X = \alpha X + \beta$ *при любом* $\gamma \in A^*$.

Доказательство. Пусть $\lambda \notin \alpha$ и X_0 – решение, тогда, подставляя его в уравнение, получим

$$X_0 = \alpha X_0 + \beta = \alpha(\alpha X_0 + \beta) + \beta = \alpha(\alpha(\alpha X_0 + \beta) + \beta) + \beta = \alpha^3 X_0 + \alpha^2\beta + \alpha\beta + \beta.$$

Продолжая выполнять подстановки, видим, что при любом $k = 0, 1, 2, \ldots$ выполняется равенство

$$X_0 = \alpha^{k+1} X_0 + (\alpha^k\beta + \alpha^{k-1}\cdot\beta + \ldots + \alpha\beta + \beta). \qquad (1)$$

Покажем сначала, что $\alpha^*\cdot\beta \subseteq X_0$. Действительно, пусть $u \in \alpha^*\cdot\beta$, тогда при некотором значении k получим $u \in (\alpha^k\beta + \alpha^{k-1}\beta + \ldots + \alpha\beta + \beta)$ и из (1) при таком значении k получаем $u \in X_0$.

Осталось показать, что $X_0 \subseteq \alpha^*\beta$. Действительно, пусть $u \in X_0$, тогда при любом k

$$u \in \alpha^{k+1}\cdot X_0 + (\alpha^k\beta + \alpha^{k-1}\beta + \ldots + \alpha\beta + \beta).$$

Но так как $\lambda \notin \alpha$, то при достаточно больших значениях k каждое слово в множестве $\alpha^{k+1}\cdot X_0$ будет длиннее слова u и, следовательно, $u \notin \alpha^{k+1}\cdot X_0$, но тогда при таких k

$$u \in (\alpha^k\beta + \alpha^{k-1}\beta + \ldots + \alpha\beta + \beta) \subseteq \alpha^*\beta.$$

Следовательно, $u \in \alpha^*\beta$. Итак, мы показали, что если X_0 – решение, то оно задается формулой $X_0 = \alpha^*\beta$, то есть является единственным. Тот факт, что $\alpha^*\beta$ на самом деле решение, проверяется простой подстановкой.

Второе утверждение теоремы предоставляем доказать читателю.

Замечание. Если в уравнении $X = \alpha X + \beta$ под α и β понимать регулярные выражения, то в случае $\lambda \notin L(\alpha)$ его единственным решением будет регулярное выражение $\alpha^*\beta$.

В случае когда $L(\alpha)$ содержит λ, уравнение имеет бесконечно много решений вида $X = \alpha^*(\beta + \gamma)$, но здесь под γ можно понимать не только регулярные выражения, но и выражения в каком-либо формализме, задающие произвольный язык. Часто в таком случае интересуются наименьшим по включению решением, так называемой «наименьшей неподвижной точкой».

Системы линейных уравнений с регулярными коэффициентами. Под стандартной системой понимают систему вида

$$\begin{cases} X_1 = \alpha_{11}X_1 + \alpha_{12}X_2 + ... + \alpha_{1n}X_n + \beta_1, \\ X_2 = \alpha_{21}X_1 + \alpha_{22}X_2 + ... + \alpha_{2n}X_n + \beta_2, \\ . \ . \ . \\ X_n = \alpha_{1n}X_1 + \alpha_{1n}X_2 + ... + \alpha_{nn}X_n + \beta_n, \end{cases}$$

где α_{ij}, β_i – регулярные выражения, X_i – переменные ($i, j = 1, 2, \ldots, n$).

Решением системы называется набор ($L(X_1)$, $L(X_2)$, …, $L(X_n)$) формальных языков, которые при подстановке вместо соответствующих переменных в уравнения обращают их в равенства. Удобно на решение смотреть как на отображение L, которое каждой переменной X_i ставит в соответствие язык $L(X_i)$. Решение L^1 называется наименьшей неподвижной точкой системы, если для любого другого решения L выполняются соотношения $L^1(X_i) \subseteq L(X_i)$ при $i = 1, 2, \ldots, n$.

Теорема. *Каждая стандартная система уравнений имеет единственную неподвижную точку.*

Доказательство. Действительно, нетрудно видеть, что отображение L^1, определяемое по формулам $L^1(X_i) = \bigcap_L L(X_i)$, где пересечение берется по всем решениям L ($i = 1, 2, \ldots, n$), является искомой неподвижной точкой системы.

Решаются такие системы уравнений методом исключения неизвестных. Если, например, $\alpha_{11} \neq \varnothing$, то первое уравнение можно представить в виде $X_1 = \alpha_{11}X_1 + \beta$, где $\beta = \alpha_{12}X_2 + \ldots + \alpha_{1n}X_n + \beta_1$, записать его решение описанным выше способом в виде $(\alpha_{11})^*\beta$ и подставить в остальные уравнения. Получим систему с меньшим числом неизвестных и так далее.

5.5. Автоматное задание языков

Недетерминированные конечные автоматы с ε-переходами. Недетерминированным конечным автоматом с ε-переходами над алфавитом A называется набор

$$\mathfrak{R} = (Q, A, q_0, F, \varphi),$$

где Q – множество состояний, A – алфавит, q_0 – начальное состояние ($q_0 \in Q$), F – множество финальных состояний ($F \subseteq Q$), φ: $Q \times (A \cup \{\varepsilon\}) \to 2^Q$ – переходная функция.

Такой автомат можно представить нагруженным ориентированным мультиграфом (диаграммой) следующим образом. Вершинами графа объявить состояния, то есть элементы множества Q, и если $q' \in \varphi(q, x)$, то из

состояния q в состояние q' провести дугу, помеченную символом $x \in \in (A \cup \{\varepsilon\})$.

Язык $L(\mathfrak{R})$, порождаемый автоматом $\mathfrak{R}$, состоит из всех слов, которые можно прочитать, двигаясь, начиная со стартового состояния q_0, по ребрам и читая приписанные им символы. Чтение заканчивается в любом из финальных состояний множества F не обязательно при первом туда попадании. При чтении символов воспринимать ε как пустое слово.

Пример 2. Пусть алфавит $A = \{a, b, c\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_1\}$ и переходная функция φ задана таблицей

	ε	a	b	c
q_0	$\{q_1\}$	$\{q_1, q_2\}$	∅	∅
q_1	∅	∅	$\{q_0\}$	$\{q_0\}$
q_2	$\{q_1\}$	∅	∅	∅

Диаграмма автомата изображена на рис. 1.

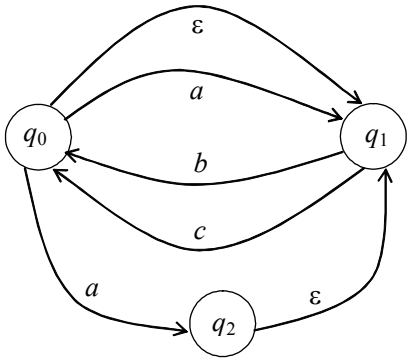


Рис. 1

Недетерминированные конечные автоматы без ε-переходов. Недетерминированным конечным автоматом без ε-переходов над алфавитом A называется набор

$$\mathfrak{R} = (Q, A, q_0, F, \varphi),$$

где Q – множество состояний, A – алфавит, q_0 – начальное состояние ($q_0 \in Q$), F – множество финальных состояний ($F \subseteq Q$) и φ: $Q \times A \to 2^Q$ – переходная функция. Такой автомат также можно представить нагруженным ориентированным мультиграфом (диаграммой). Отличие в том, что дуги могут быть помечены только символами алфавита A.

Язык $L(\mathfrak{R})$, порождаемый таким автоматом $\mathfrak{R}$, состоит из всех слов, которые можно прочитать, двигаясь, начиная со стартового состояния q_0, по ребрам и читая приписанные им символы. Чтение заканчивается в любом из финальных состояний множества F не обязательно при первом туда попадании.

Детерминированные конечные автоматы. Детерминированным конечным автоматом над алфавитом A называется набор

$$\mathfrak{R} = (Q, A, q_0, F, \varphi),$$

где Q – множество состояний, A – алфавит, q_0 – начальное состояние ($q_0 \in Q$), F – множество финальных состояний ($F \subseteq Q$) и φ: $Q \times A \to Q$ – переходная функция. Такой автомат также можно представить нагруженным ориентированным мультиграфом (диаграммой). Отличие от недетерминированного автомата в том, что из каждого состояния выходит ровно одна дуга, помеченная конкретной буквой алфавита A, причем для каждой буквы алфавита такая дуга существует.

Язык $L(\mathfrak{R})$, порождаемый таким автоматом $\mathfrak{R}$, определяется аналогично тому, как это было для недетерминированных автоматов.

Теорема. *Классы языков, задаваемые регулярными выражениями, недетерминированными конечными автоматами с* ε*-переходами, недетерминированными конечными автоматами без* ε*-переходов, детерминированными конечными автоматами, совпадают.*

Доказательство. Для доказательства достаточно по каждому регулярному выражению научиться строить равносильный недетерминированный конечный автомат с ε-переходами (синтез), затем избавляться от ε-переходов, затем детерминизировать и, наконец, по детерминированному автомату строить регулярное выражение (анализ).

Синтез. Регулярное выражение $\varnothing$ представляется автоматом $\mathfrak{R} = (Q, A, q_0, F, \varphi)$, где $Q = \{q_0, q_1\}$, алфавит A – произволен, q_0 – начальное состояние, $F = \{q_1\}$ – множество финальных состояний и переходная функция φ задается соотношениями $(\forall x \in A \cup \{\varepsilon\})\ \varphi(q_0, x) = \varnothing$, $\varphi(q_1, x) = \varnothing$.

Регулярное выражение λ представляется автоматом $\mathfrak{R} = (Q, A, q_0, F, \varphi)$, где $Q = \{q_0, q_1\}$, алфавит A – произволен, q_0 – начальное состояние, $F = \{q_1\}$ – множество финальных состояний и переходная функция φ задается соотношениями $\varphi(q_0, \varepsilon) = \{q_1\}$, $\varphi(q_1, \varepsilon) = \varnothing$ и $(\forall x \in A)\ \varphi(q_0, x) = \varnothing$, $\varphi(q_1, x) = \varnothing$.

Регулярное выражение a ($a \in A$) представляется автоматом $\mathfrak{R} = (Q, A,$

q_0, F, φ), где $Q = \{q_0, q_1\}$, q_0 – начальное состояние, $F = \{q_1\}$ – множество финальных состояний и переходная функция φ задается соотношениями $\varphi(q_0, a) = \{q_1\}$, $(\forall x \in (A \cup \{\varepsilon\}) \setminus \{a\})\ \varphi(q_0, x) = \varnothing$ и $(\forall x \in (A \cup \{\varepsilon\}))$ $\varphi(q_1, x) = \varnothing$.

Для регулярного выражения $(P \vee S)$, где P и S – регулярные выражения, можно построить задающий автомат $\mathfrak{R} = (Q, A, q_0, F, \varphi)$ следующим образом. Пусть автомат $\mathfrak{R}_1 = (Q_1, A, q_1, F_1, \varphi_1)$ задает $L(P)$, а автомат $\mathfrak{R}_2 = (Q_2, A, q_2, F_2, \varphi_2)$ задает $L(S)$. Не уменьшая общности, можно считать, что $F_1 = \{f_1\}$ и $F_2 = \{f_2\}$ – одноэлементные и что $q_1 \neq f_1$, $q_2 \neq f_2$. Положим $Q = Q_1 \cup Q_2 \cup \{q_0, f\}$, где q_0, f – новые состояния, и поясним построение автомата $\mathfrak{R}$ на языке диаграмм. Состояние q_0 соединим дугами со стартовыми состояниями q_1, q_2 автоматов $\mathfrak{R}_1$, $\mathfrak{R}_2$ и пометим их символом ε. Состояния f_1 и f_2 автоматов $\mathfrak{R}_1$, $\mathfrak{R}_2$ соединим дугами с новым состоянием f и также пометим их символом ε. Начальным состоянием построенного автомата объявим q_0, а финальным – f.

Для регулярного выражения $P{\cdot}S$, где P и S – регулярные выражения, можно построить задающий автомат $\mathfrak{R} = (Q, A, q_0, F, \varphi)$ следующим образом. Пусть автомат $\mathfrak{R}_1 = (Q_1, A, q_1, F_1, \varphi_1)$ задает $L(P)$, а автомат $\mathfrak{R}_2 = (Q_2, A, q_2, F_2, \varphi_2)$ задает $L(S)$. Не уменьшая общности, опять считаем, что $F_1 = \{f_1\}$ и $F_2 = \{f_2\}$ – одноэлементные и что $q_1 \neq f_1$, $q_2 \neq f_2$. Положим $Q = Q_1 \cup Q_2$ и поясним построение автомата $\mathfrak{R}$ на языке диаграмм. Финальное состояние автомата $\mathfrak{R}_1$ соединим дугой со стартовым состоянием автомата $\mathfrak{R}_2$ и пометим ее символом ε. В качестве q_0 возьмем стартовое состояние автомата $\mathfrak{R}_1$, а в качестве финального состояния f возьмем финальное состояние f_2 автомата $\mathfrak{R}_2$.

Для регулярного выражения P^*, где P – регулярное выражение, можно построить задающий автомат $\mathfrak{R}_{new} = (Q, A, q_0, \{f_0\}, \varphi)$ следующим образом. Пусть автомат $\mathfrak{R}_1 = (Q_1, A, q_1, \{f_1\}, \varphi_1)$ задает $L(P)$. Опять не уменьшая общности, считаем, что $F_1 = \{f_1\}$ – одноэлементное и что $q_1 \neq f_1$. Добавляем к множеству Q_1 два новых состояния – q_0 и f_0. Соединяем ε-переходами пары состояний (q_0, q_1), (f_1, f_0), (q_0, f_0) и (f_1, q_1).

В завершение заметим, что изложенные приемы очевидно позволяют по любому регулярному выражению R построить недетерминированный автомат $\mathfrak{R}$ с ε-переходами, с одним стартовым и одним финальным состоянием, причем стартовое состояние отлично от финального, и при этом такой, что $L(R) = L(\mathfrak{R})$.

Таким образом, задача синтеза решена.

Избавление от ε-переходов. Покажем, как по недетерминированному автомату $\mathfrak{R} = (Q, A, q_0, F, \varphi)$ с ε-переходами построить недетерминированный автомат $\mathfrak{R}_1 = (Q, A, q_0, F_1, \varphi_1)$ без ε-переходов, такой, что $L(\mathfrak{R}_1) = L(\mathfrak{R})$. Назовем ε-путем путь в диаграмме автомата $\mathfrak{R}$, возможно пустой, порождающий пустое слово. Обозначим через $\lambda(q)$ множество состояний, достижимых из q с помощью некоторого ε-пути.

Положим $F_1 = \{q|$ существует ε-путь из q в $F\}$. Переходную функцию φ_1 построим следующим образом

$$\varphi_1(q,a) = \bigcup_{q^1 \in \lambda(q)} \varphi(q^1, a),$$

Заметим, что в полученном автомате множество финальных состояний может быть не одноэлементным.

Детерминизация. Покажем, как по недетерминированному автомату $\mathfrak{R} = (Q, A, q_0, F, \varphi)$ без ε-переходов построить детерминированный автомат $\mathfrak{R}_1 = (Q_1, A, q_1, F_1, \varphi_1)$, такой, что $L(\mathfrak{R}_1) = L(\mathfrak{R})$. Положим $Q_1 = 2^Q$, $q_1 = \{q_0\}$, $F_1 = \{q|\ (q \subseteq Q)\ \&\ (q \cap F \neq \varnothing)\}$, а φ_1: $Q_1 \times A \to Q_1$ определим следующим образом

$$(\forall q \in Q_1)(\forall a \in A)\varphi_1(q,a) = \bigcup_{s \in q} \varphi(s,a).$$

Нетрудно видеть, что построенный таким образом автомат $\mathfrak{R}_1$ удовлетворяет условию $L(\mathfrak{R}_1) = L(\mathfrak{R})$.

Анализ. Для завершения доказательства теоремы покажем, как по заданному детерминированному конечному автомату $\mathfrak{R}$ построить регулярное выражение R, такое, что $L(R) = L(\mathfrak{R})$. Именно это и называют задачей анализа. Правда, метод, который мы используем, можно применить и к недетерминированным автоматам. Метод заключается в том, что мы сводим задачу к решению стандартной системы уравнений. Итак, рассмотрим автомат $\mathfrak{R} = (Q, A, q_0, F, \varphi)$.

Пусть $Q = \{q_0, q_1, \ldots, q_n\}$, $A = \{a_1, a_2, \ldots, a_m\}$. Введем переменные $X(q_0), X(q_1), \ldots, X(q_n)$. Переменную $X(q_i)$ для каждого $i = 0, 1, \ldots, n$ будем интерпретировать как множество слов, которые можно прочитать, начиная от состояния q_i и заканчивая в финальном состоянии, тогда $X(q_i)$ должна удовлетворять уравнению

$$X(q_i) = a_1 \cdot X(\varphi(q_i, a_1)) + a_2 \cdot X(\varphi(q_i, a_2)) + \ldots + a_m \cdot X(\varphi(q_i, a_m)) + \beta_i,$$

где $\beta_i = \lambda$, если $q_i \in F$, $\beta_i = \varnothing$, если $q_i \notin F$. Решив систему, берем в качестве ответа значение переменной $X(q_0)$.

Задача. Построить регулярное выражение, задающее язык, порождаемый автоматом $\mathfrak{R} = (Q, A, q_0, F, \varphi)$, где $Q = \{q_0, q_1, q_2, q_3\}$, $A = \{a, b\}$, $F = \{q_3\}$ и функция φ задана таблицей

	a	b
q_0	q_3	q_1
q_1	q_3	q_2
q_2	q_2	q_3
q_3	q_1	q_3

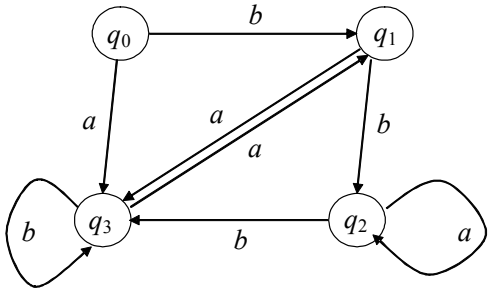


Рис. 2

Решение. Запишем систему уравнений

$$X_0 = aX_3 + bX_1,$$
$$X_1 = a \cdot X_3 + bX_2,$$
$$X_2 = aX_2 + bX_3,$$
$$X_3 = aX_1 + bX_3 + \lambda.$$

Заметим, что при записи системы мы упростили обозначения переменных используя индексы.

Из четвертого уравнения получаем $X_3 = b^*(aX_1+\lambda)$. Подставляя полученное выражение во все остальные уравнения, получим систему из трех уравнений

$$X_0 = a\,b^*(aX_1 + \lambda) + bX_1,$$
$$X_1 = ab^*(aX_1 + \lambda) + bX_2,$$
$$X_2 = aX_2 + bb^*(aX_1 + \lambda).$$

Перепишем ее в стандартном виде

$$X_0 = (ab^*a + b)X_1 + ab^*,$$
$$X_1 = ab^*aX_1 + bX_2 + ab^*,$$
$$X_2 = bb^*aX_1 + aX_2 + bb^*.$$

Из третьего уравнения получаем $X_2 = a^*(bb^*aX_1 + bb^*)$ и подставляем в остальные уравнения

$$X_0 = (ab^*a + b)X_1 + ab^*,$$
$$X_1 = ab^*aX_1 + ba^*(bb^*aX_1 + bb^*) + ab^*.$$

Преобразуем второе уравнение к стандартному виду

$$X_1 = (ab^*a + ba^*bb^*a)X_1 + ba^*bb^* + ab^*$$

и получаем из него

$$X_1 = (ab^*a + ba^*bb^*a)^*(ba^*bb^* + ab^*).$$

Наконец, получаем ответ

$$X_0 = (ab^*a + b)(ab^*a + ba^*bb^*a)^*(ba^*bb^* + ab^*) + ab^*.$$

5.6. Применение конечных автоматов в программировании

Задача. По заданному регулярному выражению α над алфавитом $A = \{a_1, a_2, \ldots, a_n\}$ найти в тексте x наименьший префикс, содержащий слово из $L(\alpha)$.

Решение. Строится регулярное выражение $\beta = (a_1 \vee a_2 \vee \ldots \vee a_n)^*\alpha$ и для него – недетерминированный конечный автомат с ε-переходами. Пусть это будет автомат $\mathfrak{R} = (Q, A, q_0, F, \varphi)$. Если при чтении текста x построенным автоматом мы приходим в финальное состояние, то это означает, что мы прочитали префикс текста x, содержащий слово из языка $L(\alpha)$.

Алгоритм, моделирующий работу недетерминированного конечного автомата $\mathfrak{R}$ с ε-переходами на входном слове $x = x_1x_2\ldots x_n \in A^*$

Q_0:= $\{q_0\}$;
for i := 1 **to** n **do** Q_i:= $\bigcup_{q \in Q_{i-1}} \varphi(q, x_i)$;
Пометить все состояния из Q_i как рассмотренные;
Пометить все состояния из $Q \setminus Q_i$ как нерассмотренные;
Все состояния из Q_i поместить в очередь;
While Очередь не пуста **do**
    {t := головной элемент из очереди (с удалением);
    For $u \in \varphi(t, \varepsilon)$ & u – не рассмотрен **do**
        {Пометить u как рассмотренное;
        Поместить u в хвост очереди и в Q_i}}

Оценим трудоемкость приведенного алгоритма. Пусть $|Q| = m$, $|\varphi(q, a)| \le e$, тогда тело цикла «while» оценивается как $O(e)$, а тело цикла «**for** i := 1 **to** n **do**» как $O(em)$ и весь алгоритм имеет трудоемкость $O(emn)$.

Анализируя алгоритм построения автомата $\Re = (Q, A, q_0, \{f\}, \varphi)$ с ε-переходами по регулярному выражению β, легко установить следующие свойства:

- $|Q|<2\cdot|\beta|$, где $|\beta|$ – длина выражения β с учетом скобок и символов операций;
- $q_0 \neq f$;
- $(\forall x \in (A \cup \{\varepsilon\}))\ \varphi(f, x) = \varnothing$;
- $(\forall q \in Q) \sum_{a \in (A \cup \{\varepsilon\})} |\varphi(q, x)| \le 2$.

Учитывая приведенные свойства, можем теперь оценить алгоритм, моделирующий работу автомата $\Re$, величиной $O(n\cdot|\beta|)$.

Рассмотрим теперь задачу частную по отношению к рассмотренной выше, полагая, что вместо регулярного выражения α, задано одно слово-образец y.

Задача. Требуется найти вхождение заданного слова-образца $y = y_1 y_2 \ldots y_n$ в слово-текст $x = x_1 x_2 \ldots x_m$ или установить, что такого вхождения нет.

Определение. По данному образцу y определим функцию S_y: $A^* \to A^*$ следующим образом: $(\forall x \in A^*)$ $S_y(x)$ – наибольший префикс слова y, являющийся суффиксом слова x.

Очевидно, $(\forall x \in A^*)\ |S_y(x)| = \max\{k \mid \mathrm{pref}_k\, y = \mathrm{suff}_k\, x\}$.

Утверждение 1. *Для любой строки x и любого символа a* $|S_y(xa)| \le$

$\le |S_y(x)| + 1$.

Действительно, пусть $|S_y(xa)| > |S_y(x)| + 1$ и $S_y(xa) = ua$, тогда $|ua| > |S_y(x)| + 1$, а u будет префиксом и суффиксом строки x, причем $|u| > |S_y(x)|$, что противоречит определению $|S_y(x)|$.

Утверждение 2. *Пусть* $q = |S_y(x)|$, *тогда для любого символа* a $|S_y(xa)| = |S_y(y_1 y_2 \ldots y_q a)|$.

Действительно, по предыдущему утверждению, $|S_y(xa)| \le q + 1$, поэтому значение $|S_y(xa)|$ не изменится, если от строки xa оставить последние $q + 1$ символов, а именно $y_1 y_2 \ldots y_q a$.

Построим по слову-образцу $y = y_1 y_2 \ldots y_n$ конечный автомат $\Re = (Q, A, q_0, \{f\}, \varphi)$, где $Q = \{0, 1, \ldots, n\}$, $q_0 = 0$, $f = n$, а переходную функцию φ определим следующим образом $(\forall q \in Q)(\forall a \in A)$ $\varphi(q, a) = |S_y(y_1 y_2 \ldots y_q a)|$. Для построенного автомата, очевидно, будет справедливо следующее утверждение.

Утверждение 3. *Прочитав текст* x, *автомат* $\Re = (Q, A, q_0, \{f\}, \varphi)$ *будет находиться в состоянии* $|S_y(x)|$.

Алгоритм вычисления функции переходов:

```
n := length(y);
for q := 0 to n do for a ∈ A do k := min{n + 1, q + 2};
repeat k := k – 1 until y1y2…yq = suff (y1 y2 … yq a);
φ (q, a) := k
```

Время работы этого алгоритма $O(n^3 |A|)$.

Пример. Пусть алфавит $A = \{a, b\}$ и $Y = aabbaab$. Допустим, что, читая текст x, мы обнаружили некоторый префикс $x_1 x_2 \ldots x_i$ слова x, заканчивающийся фрагментом $aabbaa$, являющимся префиксом слова Y, а следующий символ x_{i+1} в тексте x не равен b, то есть не совпадает с очередным символом слова Y. Считаем, что потерпели неудачу, но при этом заметим, что суффикс aa этого фрагмента является его префиксом и, возможно, он является префиксом некоторого вхождения слова Y в x. Делая такое предположение, продолжаем читать x, сравнивая очередные символы слова x с соответствующими, начиная с третьего символами, слова Y в надежде на этот раз обнаружить его вхождение в x.

Таким образом, читая x, будем считать, что мы в каждый момент находимся в некотором состоянии j, если только что прочитан префикс Y' слова Y длины j. Если при чтении следующего символа терпим неудачу,

то переходим в новое состояние j', такое, что j' – максимальный префикс слова Y', являющийся его суффиксом. Функцию, которая состоянию j ставит в соответствие j', называют функцией откатов. В нашем примере ее можно изобразить следующей диаграммой.

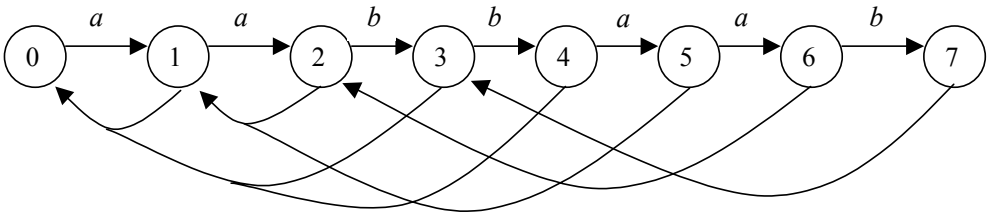


Рис. 3

Введем необходимые обозначения. Пусть Y – непустое слово в некотором алфавите, а $L(Y)$ – наибольший собственный префикс слова Y, являющийся его суффиксом. Тогда справедливы следующие утверждения:

1. Слова $L^2(Y)$, $L^3(Y)$, … являются собственными префиксами и суффиксами слова Y.

2. Последовательность $L(Y)$, $L^2(Y)$, $L^3(Y)$, … обрывается на пустом слове.

3. Любое префикс слова Y, являющийся его суффиксом, находится в последовательности $L(Y)$, $L^2(Y)$, $L^3(Y)$, …

Пример. Пусть $Y = abbabbabbacabbab$. Тогда

$$L(Y) = abbab,$$

$$L^2(Y) = ab,$$

$$L^3(Y) = \lambda.$$

Определение. *Функцией откатов для слова* $Y = Y_1 Y_2 \ldots Y_n$ *называют функцию* $f: \{1, 2, \ldots, n\} \to \{0, 1, 2, \ldots, n-1\}$, *определяемую соотношением* $f(i) = |L(\text{pref}_i Y)|$, *где* $\text{pref}_i Y$ *– префикс длины i слова Y.*

В нашем примере функция $f(i)$ задается следующей таблицей

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f(i)$	0	0	0	1	2	3	4	5	6	7	0	1	2	3	4	5

Алгоритм Кнута – Морриса – Пратта построения функции откатов для слова $Y = Y_1 Y_2 \ldots Y_n$:

$f(1) := 0;$

```
for i :=1 to n – 1 do
   begin
      j := f (i);
       while (Y [j + 1] ≠ Y [i + 1]) &(j > 0) do j := f [j];
       if Y [j + 1] = Y [i + 1] then f [i + 1] := j + 1 else f [i +1] := 0;
   end
```

Для разъяснения работы алгоритма рассмотрим ситуацию, возникшую при обработке слова Y на шаге $i = 9$. К этому моменту вычислены значения $f(i)$ при $i = 1, 2, \ldots, 9$

									i							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$Y =$	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*c*	*a*	*b*	*b*	*a*	*b*
$f(i) =$	0	0	0	1	2	3	4	5	6							

Выполняем $j := f[i]$ (= 6). Видим, что условие во внутреннем цикле не выполняется из-за первого сомножителя, так как $Y[i + 1] = Y[j + 1]$, поэтому тело внутреннего цикла не выполняется, и, далее, в соответствии с алгоритмом вычисляем $f[i + 1] := j + 1$ (= 7) и $i := i + 1$ (= 10).

Пришли к следующей ситуации $i = 10$

										i						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$Y =$	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*c*	*a*	*b*	*b*	*a*	*b*
$f(i) =$	0	0	0	1	2	3	4	5	6	7						

Вычисляем $j := f[i]$ (= 7).

Видим, что условие ($Y[i + 1] \neq Y[j + 1]$ & $j > 0$) во внутреннем цикле выполняется. Следовательно, вычисляется новое значение $j := f[j]$ (= 4); условие опять выполнено, вычисляем новое $j := f[j] = 1$ и на этот раз условие выполняется, снова вычисляем $j := f[j]$ (= 0). Наконец внутренний цикл завершается, причиной завершения является невыполнение условия ($j > 0$) и поэтому $f[i + 1] := 0$. Итак, вычислено $f[11] = 0$.

Оценим трудоемкость алгоритма. Обработка очередной буквы $Y[i + 1]$ может потребовать многих итераций во внутреннем цикле. Обозначим их число через N_i. Заметим, что каждая итерация внутреннего цикла уменьшает j по крайней мере на 1. С другой стороны, переход к следующему значению i увеличивает j не более чем на 1. Таким образом, имеем неравенства

$$f[i+1] \le f[i] - N_i + 1$$

или

$$N_i \le f[i] - f[i+1] + 1.$$

Суммируя последнее неравенство по i от 1 до $n-1$, получим

$$\sum_{i=1}^{n-1} N_i = f[1] - f[n] + n - 1 \le n\,.$$

Отсюда трудоемкость оценивается сверху величиной $O(n)$.

Построение детерминированного конечного автомата по функции откатов. Задача заключается в том, чтобы построить конечный автомат, который, читая произвольный текст, приходил бы в финальное состояние, обнаружив фрагмент, совпадающий с заданным словом $Y = Y_1\, Y_2 \ldots\, Y_n$.

Изложенный ниже алгоритм строит переходную функцию φ автомата

$$\mathfrak{R} = (Q, A, q_0, \{f\}, \varphi),$$

где $Q = \{0, 1, 2, \ldots, n\}$, $q_0 = 0$, $f = n$.

Предполагаем, что функция откатов f уже построена.

for $j := 1$ **to** n **do** $\varphi\,[j-1, Y_j] := j$;
for $a \in A$, $a \ne Y_1$ **do** $\varphi\,[0, a] := 0$;
for $j := 1$ **to** n **do for** $a \in A$, $a \ne Y_{j+1}$ **do** $\varphi\,[j, a] := \varphi\,[f[j], a]$;

Для слова $Y = aabbaab$ получим автомат, заданный диаграммой, изображенной на рис. 4.

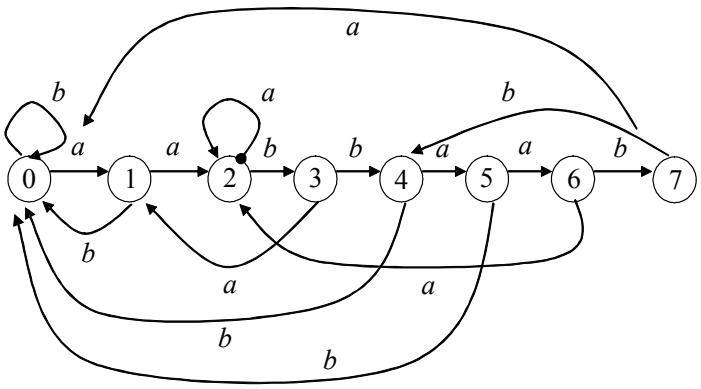


Рис. 4

Глава 6. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Теоретические исследования математических моделей вычислений играют важную роль в формировании технологии использования компьютерной техники. Каждая такая модель вносит свой вклад в реальную технологию. Развитие технологии в обход теоретических исследований часто приводит к неуклюжим труднопонимаемым и, в конечном счете, малопроизводительным видам деятельности.

Исследования универсальных методов доказательства в рамках логики предикатов дают повод для разработки на их основе новых языков программирования. Опыты, проведенные в 70-х годах прошлого века, показали перспективность использования этих методов в реальных технологиях автоматической переработки информации.

В настоящее время в рамках так называемого логического программирования ведутся исследования по использованию различных стратегий поиска доказательств утверждений, сформулированных в языке предикатов, и в частности известного в математической логике метода резолюций. Эти стратегии реализованы в настоящее время в нескольких версиях языка Пролог (Эрити Пролог, Турбо Пролог и др.).

6.1. Язык предикатов

Формулы языка предикатов строятся из предикатных и функциональных символов с помощью логических связок, кванторов и некоторых вспомогательных символов. Набор предикатных и функциональных символов заранее не фиксируется, а выбирается из содержательных соображений, связанных с желанием строить высказывания о тех или иных объектах или их свойствах и отношениях между ними. С каждым предикатным символом связывается натуральное число, его арность (число аргументов).

«Строительные материалы», из которых конструируются формулы и предложения языка предикатов:

1. Логические связки и кванторы:

$$\& \vee \neg \ \rightarrow \forall \ \exists.$$

2. Символы для конструирования переменных: (по традиции) латинская буква x и $'$ (штрих). Отдельную букву x или x с несколькими штрихами будем считать переменной. Делая такой выбор, мы подчеркиваем то обстоятельство, что используем всего два символа для образования любо-

го конечного множества переменных. На практике, конечно, это неудобно, поэтому используются и другие символы, возможно с индексами. Контекст не позволит нам «заблудиться».

3. Вспомогательные символы: прямые и круглые скобки и запятая.
4. Предикатные и функциональные символы.

Замечания

- Предикатные символы используются для обозначения предложений, в которых некоторые слова заменены переменными, так что при замене переменных именами конкретных объектов получаются высказывания об этих объектах, которые можно оценить при определенных обстоятельствах как истинные или ложные. Каждое такое предложение называется высказывательной формой, а количество k различных переменных, входящих в такое предложение, – ее арностью или местностью. Например, предложение: «Река x является притоком реки y» – двухместная форма, которая при замене переменной y на собственное имя Волга превращается в одноместную форму. «Река x является притоком реки Волга». А если еще и переменную x заменить именем Ока, то получим истинное высказывание. Если же переменную x заменить именем Енисей, то – ложное.
- При $k = 0$ имеем дело с конкретным высказыванием.
- Функциональные символы используются для обозначения отображений (функций).
- Нульместные функции называются также константами.

Основными конструкциями языка предикатов являются термы и формулы.

Правила образования термов

1. Любая переменная или константа является термом.
2. Если f – функциональный k-местный символ, а $t_1, t_2, \ldots, t_k$ – термы, то выражение $f(t_1, t_2, \ldots, t_k)$ является термом.

Замечания

- Многоточие, используемое в определении терма, не следует понимать буквально, поскольку таких символов в нашем распоряжении нет. При любом конкретном значении k мы обходимся без многоточий.
- Если в терме нет переменных, то он интерпретируется как имя некоторого объекта, если же переменные есть, то терм удобно рассматривать как схему для образования имени. Например, $\mathrm{Sin}(x)$ –

терм, который при замене переменной x константой 1 превращается в терм $\mathrm{Sin}(1)$, являющийся именем вполне конкретного числа, хотя и нетрадиционным. Под выражением Sin мы понимаем здесь функциональный символ, хотя и состоящий из трех латинских букв.

- В термах, построенных с помощью функциональных двухместных символов, традиционно используется инфиксная форма записи, при которой знак функции помещается между аргументами, например пишется $x + y$ вместо $+(x, y)$. Аналогичное замечание справедливо также и для двухместных предикатов.

Правила образования формул

1. Если p – k-местный предикатный символ, а $t_1, t_2, \ldots, t_k$ – термы, то выражение $p(t_1, t_2, \ldots t_k)$ является формулой (атомарной).
2. Если A и B – формулы, то выражения $[\mathrm{A} \,\&\, B]$, $[A \vee B]$, $[A \to B]$ и $\neg A$ являются формулами.
3. Если A – формула, а y – переменная, то выражения $\forall y\, A$, $\exists y\, A$ являются формулами.

Замечания

- В правиле 3 формула A называется областью действия соответствующего квантора, а все вхождения переменной y в атомарные подформулы формулы A называются связанными.
- Переменная, имеющая вхождение в атомарную подформулу формулы A, не находящуюся в области действия соответствующего квантора, называется свободной переменной формулы A. Конечно, одна и та же переменная может иметь как связанные, так и свободные вхождения в формулу.
- Формула, не имеющая переменных со свободными вхождениями, называется предложением.
- Формулы со свободными переменными трактуются как высказывательные формы, а предложения – как высказывания, истинностная оценка которых зависит от интерпретации входящих в них предикатных и функциональных символов в соответствии со смыслом логических связок и кванторов.

Пример. Пусть нелогическая сигнатура состоит из трех символов $\{E, M, S\}$, где E – двухместный предикат, S – одноместная функция, M – двухместная функция, тогда выражение

$$\forall z\ \forall y\, E(M(S(z), S(y)), S(M(z, y))),$$

очевидно, будет формулой. Поскольку в этой формуле нет свободных переменных, то она является предложением.

Рассмотрим следующую интерпретацию нашей сигнатуры. Пусть универсом рассуждения будет множество точек плоскости, это означает, что значениями переменных являются точки. Далее, пусть

$M(z, y)$ – точка, являющаяся серединой отрезка (z, y),

$S(z)$ – точка, симметричная точке z относительно некоторой заранее выбранной точки,

$E(z, y)$ – предикат, означающий равенство точек z и y.

При такой интерпретации нелогических символов E, M, S приведенная выше формула есть утверждение о том, что середина отрезка (z, y) симметрична середине отрезка с концами, симметричными точкам z, y.

Очевидно, это утверждение истинно.

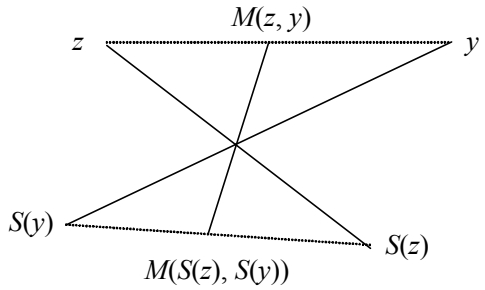


Рис. 1

Рассмотрим еще одну интерпретацию нашей сигнатуры. Пусть на этот раз универсом рассуждения будет множество действительных чисел, исключая число 0,

$M(z, y)$ – произведение чисел z, y,

$S(z)$ – число, обратное числу z,

$E(z, y)$ – «$z = y$».

Рассматриваемая нами формула является теперь утверждением о том, что для любых двух чисел из нашего универса выполняется равенство $(z\,y)^{-1} = z^{-1}\,y^{-1}$. Очевидно, это утверждение истинно.

Нетрудно привести примеры интерпретаций, при которых наша формула ложна. Следующие примеры показывают, что существуют формулы, тождественно истинные, то есть истинные при любой интерпретации, а также тождественно ложные.

Примеры. Пусть P и Q – одноместные предикатные символы.

1. $\forall x\,[\neg P(x) \vee P(x)]$ – тождественно истинная формула.

2. $\forall x\,[\neg P(x)\ \&\ P(x)]$ – тождественно ложная формула.
3. $\forall x\,[P(x) \vee Q(x)]$ – истинность этой формулы зависит от интерпретации предикатных символов P и Q.

Если в формуле есть свободные переменные, то она получает конкретное истинностное значение при означивании этих переменных.

Формула называется выполнимой, если она истинна хотя бы при одной интерпретации.

Две формулы называются логически равносильными, если при любой интерпретации предикатных и функциональных символов и при любом означивании свободных переменных они имеют одно и то же истинностное значение.

Например, формулы

$$\forall x\,[P(x) \vee Q(x)],$$
$$[\forall x\,P(x) \vee \forall x\,Q(x)]$$

не являются логически равносильными, а формулы

$$\forall x\,[P\,(x)\ \&\ Q(x)],$$
$$[\forall x\,P(x)\ \&\ \forall x\,Q(x)]$$

логически равносильны.

Логическую равносильность формул будем обозначать знаком ≡, например,

$$\forall x\,[P(x)\ \&\ Q(x)] \equiv [\forall x\,P(x)\ \&\ \forall x\,Q(x)].$$

Заметим, что в этом выражении фигурирует не одна формула, а две, соединенные знаком логической равносильности.

6.2. Некоторые сведения из математической логики

Напомним кратко основную цепочку построений математической логики, используемых в логическом программировании.

Известно, что любое предложение в логике предикатов логически равносильно предложению в предваренной нормальной форме, то есть в такой форме, когда в начале расположены все ее кванторы, за которыми расположена бескванторная ее часть. Рассмотрим пример такой формулы

$$\forall x\,\exists z\,\forall y\,\exists u\,\forall v\,[P(x,y)\ \&\ R(y,z) \to S(x,u,v)\ \&\ [S(y,u,v) \vee S(z,y,v)]], \quad (1)$$

где P, R, S – предикатные символы соответствующей арности; x, y, z, u, v – индивидные переменные.

Известно, что по любому предложению A в предваренной нормаль-

ной форме можно построить так называемое сколемовское предложение B. Для этого избавляются от кванторов существования следующим образом. Пусть $\exists z$ – самое левое вхождение квантора существования в рассматриваемую формулу и перед ним расположены k кванторов общности с переменными $x_1, x_2, ..., x_k$. Выбираем новый k-местный функциональный символ f, вхождение $\exists z$ удаляем из формулы, а каждое вхождение переменной z заменяем термом $f(x_1, x_2, ..., x_k)$. Аналогичным образом избавляемся и от других кванторов существования. В результате получим сколемовскую формулу B для исходной формулы A.

Так, для формулы (1) соответствующая сколемовская формула будет иметь вид

$$\forall x\, \forall y\, \forall v\, [[P(x,y)\, \&\, R(y,f(x)) \rightarrow$$

$$\rightarrow S(x, g(x,y), v)]\, \&\, [S(y, g(x,y), v) \vee S(g(x,y), y, v)]], \qquad (2)$$

полученный из (1) заменой z на $f(x)$, а u на $g(x, y)$. Заметим, что если бы было $k = 0$, то переменная z заменялась бы на новую константу. Процесс получения сколемовской формулы по заданной формуле A называется сколемизацией.

Известно, что сколемовская формула B, соответствующая формуле A, может быть логически неравносильна формуле A, однако они либо обе выполнимы, либо обе невыполнимы (равносильность по выполнимости).

Для иллюстрации этого факта рассмотрим простую формулу

$$\forall x\, \exists y\, R(x, y),$$

которая интуитивно выражает существование функции f такой, что для любого элемента x выполняется $R(x,f(x))$. При сколемизации она превращается в формулу

$$\forall x\, R(x, f(x)).$$

Равносильность по выполнимости формулы A и соответствующей ей сколемовской формулы B может быть использована следующим образом. Предположим, мы хотим доказать, что формула C является логическим следствием формул A и B. Это сводится к доказательству невыполнимости формулы

$$[A\, \&\, B\, \&\, \neg C]$$

или соответствующей ей сколемовской формулы, что осуществить технически оказывается проще.

Поскольку в сколемовской формуле используются только кванторы общности и все они расположены в начале формулы, то их обычно опускают, подразумевая по умолчанию их наличие, а бескванторную часть представляют в нормальной конъюнктивной форме. Полученная таким образом формула называется клаузальной. В нашем случае формула (2) превращается в клаузальную формулу

$$[\neg P(x,y) \vee \neg R(y,f(x)) \vee S(x,g(x,y),v)] \,\&\, [S(y,g(x,y),v) \vee S(g(x,y),y,v)] \quad (3)$$

Упомянутый выше метод резолюций основывается на единственном правиле вывода, называемом правилом резолюции, которое заключается в следующем.

Из двух формул вида

$$[\neg A \vee B_1 \vee B_2 \vee \ldots \vee B_k]$$

и

$$[A \vee D_1 \vee D_2 \vee \ldots \vee D_s]$$

в соответствии с правилом резолюции выводится формула

$$[B_1 \vee B_2 \vee \ldots \vee B_k \vee D_1 \vee D_2 \vee \ldots \vee D_s].$$

Видно, что клаузальная форма хорошо приспособлена для применения правила резолюции. Детали этого применения в логике предикатов будут рассмотрены ниже.

Из математической логики известно, что не существует алгоритма, который по любому множеству H формул-гипотез логики предикатов и еще одной формуле A отвечал бы на вопрос, является ли A логическим следствием множества H. Однако существует алгоритм, который в случае, когда A логически следует из H, строит доказательство этого факта с использованием правила резолюции, в противном случае алгоритм может работать бесконечно.

Различные версии языка Пролог базируются на использовании так называемых хорновских клаузальных формул. Хорновскими называются формулы, являющиеся дизъюнкциями атомарных формул и/или их отрицаний, причем атомарная часть без отрицания может быть в такой формуле не более чем одна. Рассмотрим пример такой формулы

$$R(x, y) \vee \neg P(x) \vee \neg Q(x, z) \vee \neg S(f(y)). \quad (4)$$

Ее можно представить в виде

$$P(x) \,\&\, Q(x, z) \,\&\, S(f(y)) \rightarrow R(x, y). \quad (5)$$

Эта формула воспринимается Прологом так, как если бы все ее переменные

были связаны квантором общности. Восстанавливая кванторы, имеем

$$\forall x\ \forall y\ \forall z\ [P(x)\ \&\ Q(x, z)\ \&\ S(f(y)) \to R(x, y)]. \qquad (6)$$

Учитывая, что z не входит в правую часть импликации, формулу (6) можно переписать в виде

$$\forall x\ \forall y\ [\exists z\ [P(x)\ \&\ Q(x, z)\ \&\ S(f(y))] \to R(x, y)],$$

изменив область действия квантора $\exists z$.

В Прологе принято формулы, аналогичные формуле (5), записывать в виде

$$R(x, y) :- P(x), Q(x, z), S(f(y)), \qquad (7)$$

меняя местами левую и правую части импликации и вместо знака конъюнкции ставя запятую.

Формулу (7) Пролог воспримет как указание на то, что для доказательства истинности $R(x, y)$ надо найти некоторое значение z и доказать, что истинны $P(x)$, $Q(x, z)$, $S(f(y))$. Такие формулы принято называть правилами.

Если в хорновской клаузальной формуле отсутствуют атомарные части с отрицанием, то такая формула называется фактом. Если в хорновской клаузальной формуле отсутствует атомарная часть без отрицания, то такая формула называется запросом. Программой в Прологе называется набор фактов и правил.

По заданной программе и запросу система Пролог определяет, является ли запрос логическим следствием фактов и правил программы. При этом если в запросе имеются свободные переменные, то в процессе поиска доказательства эти переменные конкретизируются, то есть принимают конкретные значения, и при успешном его завершении эти конкретизированные значения являются ответом к поставленной задаче. Если же доказательство не будет найдено, то система ответит «no».

6.3. Примеры формальных доказательств

Пример 1. Вывести из гипотез H_1, H_2, H_3 заключение C, где

$$H_1: \forall x\ [E(x)\ \&\ \neg P(x) \to \exists y\ [R(x, y)\ \&\ D(y)]],$$
$$H_2: \exists x\ [E(x)\ \&\ M(x)\ \&\ \forall y\ [R(x, y) \to M(y)]],$$
$$H_3: \forall x\ [M(x) \to \neg P(x)],$$
$$C: \exists x\ [M(x)\ \&\ D(x)].$$

Префиксная форма:

$$\text{Pref}\ (H_1)\colon \forall x\ \exists y\ [E(x)\ \&\ \neg P(x) \to [R(x, y)\ \&\ D(y)]],$$

$$\text{Pref}\ (H_2)\colon \exists x\ \forall y\ [E(x)\ \&\ M(x)\ \&\ [R(x, y) \to M(y)]],$$

$$\text{Pref}\ (H_3)\colon \forall x\ [M(x) \to \neg P(x)],$$

$$\text{Pref}\ (C)\colon \exists x\ [M(x)\ \&\ D(x)].$$

Сколемовская форма:

$$\text{Sk}\ (H_1)\colon \forall x\ [E(x)\ \&\ \neg P(x) \to [R(x, f(x))\ \&\ D(f(x))]],$$

$$\text{Sk}\ (H_2)\colon \forall y\ [E(a)\ \&\ M(a)\ \&\ [R(a, y) \to M(y)]],$$

$$\text{Sk}\ (H_3)\colon \forall x\ [M(x) \to \neg P(x)],$$

$$\text{Sk}\ (\neg C)\colon \forall x\ [\neg M(x) \vee \neg D(x)].$$

Клаузальная форма (опускаем кванторы общности, а бесквантворные части приводим к КНФ и из каждого сомножителя получаем клаузу):

$$\text{Cla}\ (H_1)\colon [\neg E(x) \vee P(x) \vee R(x, f(x)] \ \&\ [\neg E(x) \vee P(x) \vee D(f(x))\ ],$$

$$\text{Cla}\ (H_2)\colon M(a)\ \&\ E(a)\ \&\ [\neg R(a, y) \vee M(y)],$$

$$\text{Cla}\ (H_3)\colon \neg P(x) \vee \neg M(x),$$

$$\text{Cla}\ (C)\colon \neg M(x) \vee \neg D(x).$$

Доказательство с использованием правила резолюции

1. $\neg E(x) \vee P(x) \vee R(x, f(x)$ – из гипотезы H_1,
2. $\neg$E $(x) \vee P(x) \vee D(f(x))$ – из гипотезы H_1,
3. $M(a)$, – из гипотезы H_2,
4. $E(a)$, – из гипотезы H_2,
5. $\neg R(a, y) \vee M(y)$, – из гипотезы H_2,
6. $\neg P(x) \vee \neg M(x)$, – из гипотезы H_3,
7. $\neg M(x) \vee \neg D(x)$, – из заключения C,
8. $P(a) \vee R(a, f(a)$, – из 1, 4 с помощью подстановки (x/a),
9. $P(a) \vee D(f(a))$, – из 2, 4 с помощью подстановки (x/a),
10. $\neg P(a)$, – из 3, 6 с помощью подстановки (x/a),

11. $D(f(a))$, – из 9, 10,

12. $\neg M(f(a))$ – из 7, 11 с помощью подстановки $(x/f(a))$,

13. $R(a, f(a))$, – из 8, 10,

14. $M(f(a))$, – из 5, 13 с помощью подстановки $(y/f(a))$,

15. $\square$ – из 12, 14

Пример 2. Рассмотрим предикаты с интерпретацией:

$F(x, y) \Leftrightarrow x$ является отцом для y,

$S(x, y) \Leftrightarrow x, y$ – дети одного отца,

$M(x) \Leftrightarrow x$ – мужчина,

$B(x, y) \Leftrightarrow x$ брат для y,

В качестве аксиом рассмотрим формулы

A_1: $\forall x\ \forall y\ [F(x, y) \to M(x)]$,

A_2: $\forall x\ \forall y\ \forall w\ [F(x, y)\ \&\ F(x, w) \to S(y, w)]$,

A_3: $\forall x\ \forall y\ [S(x, y)\ \&\ M(x) \to B(x, y)]$.

Пусть из интерпретации известны факты

A_4: F(‘Иван’, ‘Харитон’),

A_5: F(‘Иван’, ‘Василий’),

A_6: F(‘Василий’, ‘Елена’).

Вопрос: «Есть ли брат у Харитона?» – на языке предикатов записывается как

A_7: $\exists z\ B(z,$ ‘Харитон’)?

Доказательство

1. $\neg F(x, y) \vee M(x)$ – из формулы A_1,

2. $\neg F(x, y) \vee \neg F(x, w) \vee S(y, w)$ – из формулы A_2,

3. $\neg S(x, y) \vee \neg M(x) \vee B(x, y)$ – из формулы A_3,

4. F(‘Иван’, ‘Харитон’) – формула A_4,

5. F(‘Иван’, ‘Василий’) – формула A_5,

6. F(‘Василий’, ‘Елена’) – формула A_6,

7. $\neg B(z,$ ‘Харитон’) – отрицание запроса A_7,

8. $\neg F($‘Иван’$, w) \vee S($‘Василий’$, w)$ – из 2, 5, подстановка (x/‘Иван’, y/‘Василий’),

9. S(‘Василий’, ‘Харитон’) – из 4, 8, подстановка (w/‘Харитон’),

10. M(‘Василий’) – из 6, 1, подстановка (x/‘Василий’, y/‘Елена’),
11. $\neg S$(‘Василий’,y) $\vee$ B(‘Василий’,y) – из 10, 3, подстановка (x/‘Василий’),
12. B(‘Василий’, ‘Харитон’) – из 9, 11, подстановка (y/‘Харитон’),
13. □ – из 12, 7, подстановка (z/‘Василий’)

Фактически мы не только получили ответ на наш запрос, но и подтвердили его конкретным значением переменной z. Приведенный вывод можно модифицировать, если ввести предикат answer(z) и вместо цели «7. $\neg B$(z, ‘Харитон’)» поставить новую цель

7′. $\neg B$(z, ‘Харитон’) $\vee$ answer(z). Тогда шаг 13 превратится в 13′.

13′. answer(‘Василий’) – из 12, 7′, подстановка (z/‘Василий’).

Упражнение

Рассмотрите вывод, в котором первые 7 формул являются посылками. Для остальных формул выпишите пояснения к применению правила резолюции.

1. $\neg A(z)$,
2. $A(x) \vee \neg P(x) \vee \neg Q(x, y)$,
3. $A(x) \vee \neg R(y) \vee \neg Q(y, x)$,
4. $P(a)$,
5. $Q(b, c)$,
6. $R(a)$,
7. $R(b)$,
8. $\neg P(z) \vee \neg Q(z, y)$,
9. $\neg Q(a, y)$,
10. $\neg R(y) \vee \neg Q(y, z)$,
11. $\neg Q(a, z)$,
12. $\neg Q(b, z)$,
13. □

6.4. Элементы языка Пролог

Основным элементом языка Пролог является ***терм.*** Термы строятся из переменных, атомов, чисел и функторов с использованием круглых скобок.

Переменная **–** это цепочка (слово), составленная из букв, цифр и символа подчеркивания, начинающаяся с большой буквы или символа подчеркивания. Если переменная используется однажды, то вместо нее можно использовать так называемую анонимную переменную, состоящую из одного символа подчеркивания.

Атом **–** это цепочка, составленная из букв, цифр и символа подчеркивания, начинающаяся с маленькой буквы или с большой буквы, но тогда в одинарных кавычках. Последний способ удобен, если атом является собственным именем. Иногда атомы строятся и из специальных знаков, но мы не будем их использовать при первоначальном знакомстве.

Числа записываются традиционным образом. Числа с плавающей запятой в обычных применениях Пролога используются редко из-за ошибок округления.

Функтор синтаксически совпадает с атомом.

Терм – это либо переменная, либо атом, либо число, либо выражение вида

$$f(t_1, t_2, \ldots, t_k),$$

где f – функтор, а $t_1, t_2, \ldots, t_k$ – термы.

Для некоторых специальных функторов, например знаков арифметических операций, отношений сравнения и других, в Прологе, как в традиционной математике, используется инфиксная форма записи. Например, выражение $X + 1$ рассматривается как терм с функтором + и двумя аргументами X и 1.

Среди термов ввиду особой важности выделяются термы для представления списков. Канонически список представляется двухместным термом, первым аргументом которого является головной элемент списка, а вторым – его хвост, то есть список, полученный из исходного удалением головного элемента. Функтором в такой записи часто используется символ точка. Альтернативным представлением списка является выражение вида $[t_1, t_2, \ldots, t_k]$ или $[t \mid L]$, где t – головной элемент, а L – хвост списка. Допустимо также выражение вида $[t_1, t_2, \ldots, t_k \mid L]$.

Важным инструментом в языке Пролог является унификация термов с помощью подстановок. Такую унификацию мы применяли выше в примерах на доказательство методом резолюций. Сейчас более подробно рассмотрим понятие унификации.

Подстановкой называется набор пар $\theta = (x_1/t_1, x_2/t_2, \ldots, x_n/t_n)$, где $x_1, x_2, \ldots, x_n$ – переменные, а $t_1, t_2, \ldots, t_n$ – термы.

Через $E\theta$ обозначим результат подстановки термов $t_1, t_2, \ldots, t_n$ в вы-

ражение E вместо переменных $x_1, x_2, \ldots, x_n$.

Пусть $\pi = (y_1/u_1, y_2/u_2, \ldots, y_m/u_m)$ – еще одна подстановка. Композиция $\theta\pi$ двух подстановок θ и π определяется следующим образом

$$E(\theta\pi) = (E\theta)\pi.$$

Подстановка $\theta\pi$ может быть вычислена следующим образом. Составим из подстановок θ и π последовательность

$$(x_1/t_1\pi, x_2/t_2\pi, \ldots, x_n/t_n\pi, y_1/u_1, y_2/u_2, \ldots, y_m/u_m)$$

и проведем следующие две операции:

1. Если некоторое y_i совпадает с некоторым x_j, то вычеркиваем пару y_i/u_i.
2. Если $t_i\pi = x_i$, то вычеркиваем пару $x_i/t_i\pi$.

Пример. Пусть $\theta = (x/f(y), y/z)$, $\pi = (x/a, y/b, z/y)$. Составим последовательность

$$(x/f(y)\pi, y/z\pi, x/a, y/b, z/y) = (x/f(b), y/y, x/a, y/b, z/y)$$

и по первому правилу вычеркиваем пары x/a и y/b, затем по второму правилу – пару y/y. В результате получим

$$\theta\pi = (x/f(b), z/y).$$

Подстановка θ называется ***унификатором*** термов E_1, E_2, если $E_1\theta = E_2\theta$. ***Наиболее общим унификатором*** термов E_1, E_2 называется подстановка σ, такая, что любой другой их унификатор θ представляется в виде $\theta = \sigma\pi$.

Пример. Для термов $P(a, y)$, $P(x, f(b))$ унификатором будет подстановка

$$(x/a, y/f(b)).$$

Будет ли она наиболее общим унификатором?

Пример. Для термов $P(a, x, f(g(y)))$ и $P(z, f(z), f(u))$ наиболее общим унификатором будет подстановка $(z/a, x/f(a), u/g(y))$. Результатом унификации будет терм $P(a, f(a), f(g(y)))$.

Часть 3. СТРУКТУРЫ ДАННЫХ

Введение

При разработке алгоритма для компьютерного решения той или иной задачи необходимая информация формализуется в виде набора элементов различных типов. В каждой системе программирования предусмотрено использование некоторых примитивных типов данных и средств, с помощью которых можно группировать их в более сложные структуры. Будем считать примитивными следующие типы: булевский, целый, вещественный и символьный. Многие системы программирования позволяют группировать данные примитивных типов в массивы однотипных элементов, записи из фиксированного числа элементов разных типов и в некоторые другие структуры. Вообще, под типом данных понимают произвольное множество, называемое множеством значений, и набор операций над значениями. Так, значениями целого типа являются целые числа из некоторого диапазона, зависящего от системы программирования, а операциями – обычные арифметические операции.

При алгоритмизации задач, решение которых опирается на использование математических знаний и требует математических доказательств, разработка алгоритма часто проводится также в математических или формализованных прикладных терминах. При этом в достаточно большой степени происходит отвлечение от технических возможностей исполнителя алгоритмов (человека или технического устройства). Так, если при описании алгоритма используется понятие множества, то достаточно уметь выполнять некоторый набор операций с множествами и отвечать на некоторые вопросы относительно множеств. Перечень таких операций может быть следующим:

- ввести в рассмотрение пустое множество;
- включить элемент в множество;
- исключить элемент из множества;
- проверить, пусто ли рассматриваемое множество:
- узнать, сколько элементов в рассматриваемом множестве.

Эти операции с некоторой точки зрения можно считать элементарными и до известной поры не думать о способе их реализации исполнителем. В таких случаях говорят, что мы имеем дело с абстрактным типом данных.

Широко используемыми абстрактными типами данных наряду с множествами являются мультимножества, взвешенные множества, кортежи, отображения, приоритетные очереди, графы и так далее. Когда дело доходит до разработки программ в конкретной системе программирования, приходится принимать решения о представлении таких данных в памяти и о реализации операций над ними доступными средствами.

В настоящее время большинство алгоритмов проектируется для использования в устройствах, обладающих адресуемой памятью. Каждый элемент информации, размещенный в такой памяти, занимает определенную позицию. По известной позиции элемента в такой памяти доступ к нему осуществляется за некоторую условную единицу времени, зависящую только от типа получаемой информации, фактически – от физического размера ячейки памяти или от количества таких ячеек, предназначенных для ее хранения, но не от ее конкретного содержания. Более того, позиции элементов сами могут быть элементами информации, с которыми могут производиться некоторые операции, что позволяет использовать так называемую косвенную адресацию. Наличие косвенной адресации позволяет поручить программной системе или разрабатываемой прикладной программе поиск свободных участков памяти для размещения новых элементов информации и запоминание их адресов с последующим их использованием для доступа к информации.

Информация, размещенная в адресуемой памяти, приобретает новые свойства. Элемент информации характеризуется не только своим содержанием, но и адресом, то есть местом расположения в памяти. Два элемента данных, соседних в некотором содержательном смысле, не обязательно будут располагаться в соседних ячейках памяти. Они могут оказаться в «непредвиденных» местах. Проектируя программную реализацию алгоритма, необходимо проектировать и способ расположения в памяти обрабатываемой информации.

Существуют типы данных, которые естественным образом вкладываются в адресуемую структуру технической памяти, причем легко выполняются все операции, предусмотренные для такого типа данных. Примером может служить вектор фиксированной размерности, задаваемый упорядоченным набором своих компонент. Наиболее естественным является его хранение в виде массива, при котором соседние компоненты располагаются в ячейках с подряд идущими номерами. Этот способ позволяет легко выполнять покомпонентные операции (сложение векторов, вычисление скалярного произведения и другие). Однако если размерность вектора в процессе работы изменяется, например путем удаления компонент

или вставки новых, то представление в виде массива оказывается неудобным, так как операции удаления и вставки при условии сохранения порядка следования элементов требуют перезаписи, возможно, достаточно большого числа компонент, что может неблагоприятно сказаться на эффективности алгоритма. Чтобы избежать неэффективности такого рода, одновременно с алгоритмом проектируется структура представления данных, позволяющая реализовать выполнение всех необходимых операций в приемлемое время.

В практике программирования накоплен большой опыт структурирования информации. К счастью, способы структурирования, изобретенные при решении одной задачи, часто находят применение и во многих других. Например, для представления кортежей и множеств в памяти компьютера могут использоваться такие структуры данных, как линейные и циклические списки.

Во многих задачах исходные данные представляют собой так называемые взвешенные множества. Взвешенным называется множество, каждому элементу которого поставлено в соответствие в качестве веса некоторое число. Часто используемыми операциями с такими множествами являются поиск элемента с минимальным весом, вставка нового элемента со своим весом, удаление элемента и некоторые другие. Для быстрого выполнения таких операций разработаны так называемые кучеобразные структуры данных.

Классы функций, используемые для оценки сложности алгоритмов. Все функции, используемые ниже для оценки сложности алгоритмов, считаются асимптотически неотрицательными функциями натурального аргумента, то есть неотрицательными начиная с некоторого значения аргумента n.

Для асимптотических оценок сверху используется класс функций

$$\mathrm{O}(g(n)) = \{f(n) : \exists c > 0, \exists n_0 \forall n > n_0 [0 \le f(n) \le c \cdot g(n)]\}.$$

Для асимптотических оценок снизу используется класс функций

$$\Omega(g(n)) = \{f(n) : \exists c > 0, \exists n_0 \forall n > n_0 [0 \le c \cdot g(n) \le f(n)]\}.$$

Для асимптотически точных оценок используется класс функций

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0, \exists n_0 \forall n > n_0 [0 \le c_1 g(n) \le f(n) \le c_2 g(n)]\}.$$

Очевидно, справедливы следующие соотношения

$$\Theta(g(n)) = \mathrm{O}(g(n)) \bigcap \Omega(g(n)),$$

$$f(n) \in \mathrm{O}(g(n)) \Leftrightarrow g(n) \in \Omega(f(n)).$$

Далее, предполагается знание стандартных функций, используемых при оценках сложности, таких, как полиномы, экспоненты, суперэкспоненты, логарифмы, суперлогарифмы, факториалы, числа Фибоначчи. Предполагается умение их сравнивать по скоростям роста.

Амортизационный анализ. Наряду с получением верхних и нижних оценок и оценок в среднем часто используются так называемые амортизационные оценки.

Амортизационный анализ применяется при оценке времени выполнения корректной последовательности, состоящей из n однотипных или разнотипных операций с некоторой структурой данных. Если верхнюю оценку времени выполнения одной операции умножить на n, получим верхнюю оценку выполнения всех n операций. Часто такая оценка бывает сильно завышенной. Иногда большое время выполнения очередной операции влечет малое время выполнения следующих операций. Более того, такая ситуация может создаваться искусственно, то есть при выполнении очередной операции мы можем готовить почву для более эффективного выполнения следующей. Поэтому возникает задача изучения асимптотического поведения гарантированной оценки для среднего времени выполнения одной операции.

При амортизационном анализе определяется некоторая так называемая учетная (амортизационная) стоимость одной операции, которая может быть как больше, так и меньше реальной стоимости конкретной операции. Но при этом для любой корректной последовательности операций фактическая суммарная длительность всех операций не должна превосходить суммы их учетных стоимостей. Зная учетную стоимость одной операции, верхнюю оценку времени выполнения последовательности из n операций можно получить, умножив ее на n.

Ниже рассмотрим три часто используемых метода амортизационного анализа: метод группировки, метод предоплаты и метод потенциалов.

Метод группировки. Предположим, что мы оценили сверху время выполнения последовательности из n операций, установив, что она не превосходит $T(n)$, тогда величину $T(n)/n$ объявим учетной стоимостью любой операции из рассматриваемой последовательности, независимо от ее длительности.

Метод предоплаты. В этом методе операции разных типов получают разные учетные стоимости, причем эти стоимости могут быть как больше, так и меньше фактических. Если учетная стоимость превосходит факти-

ческую, то разность между ними рассматривается как резерв на оплату в будущем тех операций, у которых учетная стоимость ниже реальной. Учетные стоимости должны выбираться так, чтобы в любой момент времени фактическая стоимость не превосходила суммы учетных стоимостей, то есть чтобы резерв оставался неотрицательным.

Метод потенциалов. Этот метод является обобщением метода предоплаты. Здесь резерв определяется функцией состояния структуры данных в целом. Эта функция называется потенциалом.

Общая схема метода такова. Пусть над структурой данных предстоит произвести n операций, и пусть D_i – состояние структуры данных после i-й операции (D_0 – исходное состояние). Потенциал представляет собой функцию ϕ из множества возможных состояний структуры данных в множество действительных чисел.

Пусть c_i – реальная стоимость i-й операции. Учетной стоимостью i-й операции объявим число C_i, определяемое формулой

$$C_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

как сумма реальной стоимости операции плюс приращение потенциала в результате выполнения этой операции. Тогда суммарная учетная стоимость всех операций равна

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0).$$

Если нам удалось придумать функцию ϕ, для которой

$$\phi(D_n) \geq \phi(D_0),$$

то суммарная учетная стоимость даст верхнюю оценку для реальной стоимости последовательности из n операций. Не ограничивая общности, можно считать, что

$$\phi(D_0) = 0.$$

Говоря неформально, если разность потенциалов

$$\phi(D_i) - \phi(D_{i-1})$$

положительна, то учетная стоимость i-й операции включает в себя резерв (предоплату за будущие операции); если же эта разность отрицательна, то учетная стоимость i-й операции меньше реальной и разница покрывается за счет накопленного к этому моменту потенциала.

Учетные стоимости и оценки реальной стоимости, рассчитанные с

помощью метода потенциалов, зависят от выбора потенциальной функции, а сам этот выбор является делом творческим.

Ниже эти три метода будут проиллюстрированы на примере анализа работы двоичного счетчика с единственной операцией Increment (прибавление единицы).

Амортизационные оценки работы двоичного счетчика. Рассмотрим работу k-разрядного двоичного сбрасываемого счетчика, реализованного как массив битов $A[0 .. k–1]$, хранящий двоичную запись числа x. Считаем, что $A[0]$ – младший разряд. Пусть первоначально $x = 0$. Единственной операцией в нашем примере будет операция Increment, увеличивающая x на 1 по модулю 2^k.

Увеличение счетчика на единицу происходит следующим образом: все начальные единичные биты в массиве A, если они есть, становятся нулями, а следующий непосредственно за ними нулевой бит, если таковой есть, устанавливается в единицу. Стоимость операции Increment линейно зависит от общего количества битов, подвергшихся изменению. Каждое такое изменение будем считать элементарной операцией.

Анализ работы двоичного счетчика методом группировки. Применим метод группировки для анализа сложности n-кратного выполнения операции Increment. Поскольку в худшем случае, когда массив A состоит из одних единиц, меняются все k битов, то n-кратное выполнение операции Increment может быть оценено как $O(nk)$ элементарных операций. Но эта оценка слишком груба.

Чтобы получить более точную оценку, учтем, что не каждый раз значения всех k битов меняются. В самом деле, младший бит $A[0]$ меняется при каждом исполнении операции Increment. Следующий по старшинству бит $A[1]$ меняется только через раз. При счете от нуля до n этот бит меняется $[n/2]$ раз. Бит $A[2]$ меняется только каждый четвертый раз, и так далее. Заметим, что если $0 \le i \le \log_2 n$, то в процессе счета от 0 до n разряд $A[i]$ меняется $[n/2^i]$ раз, а если $i > [\log_2 n]$, то он вообще не меняется. Следовательно, общее количество операций зануления и записи 1 равно

$$n + [n/2] + [n/4] + \ldots + [n/2]^{[\log n]} < n(1 + 1/2 + 1/4 + \ldots) = 2n.$$

Тем самым увеличение двоичного счетчика от 0 до n требует не более $O(n)$ операций, причем константа не зависит от k и равна 2. Учетную стоимость операции Increment можно считать равной $O(n)/n = O(1)$, константа не зависит от k.

Анализ работы двоичного счетчика методом предоплаты. Приме-

ним метод предоплаты для анализа сложности n-кратного выполнения операции Increment. Будем считать, что реальная стоимость изменения бита составляет 1 рубль. Установим такие учетные стоимости: 2 рубля за запись единицы, 0 за очистку. При каждой установке бита в единицу одним из двух рублей учетной стоимости будем расплачиваться за реальные затраты на эту установку, а второй рубль, остающийся в резерве, будем «прикреплять» к рассматриваемому биту. Поскольку первоначально все биты были нулевыми, в каждый момент к каждому ненулевому биту будет прикреплен резервный рубль. Стало быть, за очистку любого бита дополнительно платить нам не придется: мы расплатимся за нее рублем, прикрепленным к этому биту в момент его установки.

Теперь легко определить учетную стоимость операции Increment. Поскольку каждая такая операция требует не более одной установки бита, ее учетную стоимость можно считать равной 2 рублям. Следовательно, фактическая стоимость n последовательных операций Increment, начинающихся с нуля, есть $O(n)$, поскольку она не превосходит суммы учетных стоимостей $2n$.

Анализ работы двоичного счетчика методом потенциалов. Проанализируем теперь трудоемкость n-кратного выполнения операции Increment с помощью метода потенциалов.

Пусть D_0 – содержимое счетчика в начальный момент, D_i – содержимое счетчика после выполнения i-й операции, $\phi(D_i)$ – число единиц в записи D_i, t_i – число единиц, превращенных в нули при i-й операции.

Очевидно, $\phi(D_i) \le \phi(D_{i-1}) - t_i + 1$.

Пусть далее c_i – реальная стоимость i-й операции Increment, C_i – ее учетная стоимость.

Очевидно $c_i \le t_i + 1$. Тогда

$$C_i = c_i + \phi(D_i) - \phi(D_{i-1}) \le t_i + 1 + \phi(D_i) - \phi(D_{i-1}) \le$$

$$\le t_i + 1 + \phi(D_{i-1}) - t_i + 1 - \phi(D_{i-1}) = 2.$$

Если счет начинается с нуля, то

$$\phi(D_0) = 0$$

и

$$\phi(D_i) \ge \phi(D_0)$$

для всех i. Поскольку сумма учетных стоимостей оценивает сверху сумму реальных стоимостей, имеем

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} C_i \le 2n,$$

то есть получаем, что суммарная стоимость n операций есть $O(n)$ с константой (двойкой), не зависящей от k.

Метод потенциалов позволяет разобраться и со случаем, когда счет начинается не с нуля. В этом случае имеем

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0),$$

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} C_i - \phi(D_n) + \phi(D_0) \le 2n + \phi(D_0) \le 2n + k,$$

откуда при достаточно больших значениях n ($n = \Omega(k)$) получаем, что реальная стоимость оценивается как $O(n)$, причем константа в O-записи не зависит ни от k, ни от начального значения счетчика.

Глава 1. СПИСКИ

1.1. Общие сведения о списках

Остановимся на наиболее часто используемых структурах данных, называемых списками. Списки лежат в основе многих более сложных структур данных. В простейшем случае списки используются для представления кортежей.

Кортеж – это конечная последовательность, возможно с повторениями, элементов некоторого множества E. Элементами кортежа могут быть числа, символы некоторого алфавита, точки плоскости и т. д. В более сложных случаях элементами кортежа, в свою очередь, могут быть также кортежи. Элементы, не являющиеся кортежами, называются атомами. Кортеж характеризуется своей длиной. Удобно рассматривать кортежи, не содержащие ни одного элемента. Такие кортежи называются пустыми. Длина пустого кортежа считается равной 0.

Элемент кортежа характеризуется своим номером в последовательности (кортежным номером) и своим содержанием, то есть элементом множества E. Если длина кортежа равна n, $n > 0$, то кортеж S удобно рассматривать как отображение s множества $N = \{1, 2, ..., n\}$ в множество E. Таким образом, $s(i)$ – это i-й элемент кортежа S.

Термин «список» используется как обобщающее название различных структур данных, используемых для представления кортежей в памяти компьютера. При представлении кортежа в памяти появляется еще одна характеристика элемента кортежа – его позиция в памяти. В некоторых случаях номер элемента в кортеже и его позиция в памяти связаны друг с другом арифметическими соотношениями таким образом, что по номеру легко вычисляется позиция и, наоборот, по позиции вычисляется номер. В других случаях связь между номерами и позициями задается «таблично» или осуществляется с помощью алгоритмических процедур. Множество позиций обозначим через P. Иногда удобно считать, что в множестве P имеется специальный элемент nil, указывающий на несуществующую область памяти. Таким образом, при рассмотрении того или иного списка имеем дело с тремя множествами E, N, P и с отображениями на этих множествах.

Типичными при работе со списками являются следующие операции:

- Нахождение позиции элемента в памяти по его номеру в кортеже.
- Нахождение позиции элемента, следующего в кортеже за элементом из заданной позиции.
- Нахождение позиции элемента, предшествующего в кортеже элементу из заданной позиции.
- Удаление элемента, находящегося в заданной позиции.
- Вставка в кортеж нового элемента перед элементом, расположенным в заданной позиции.
- Определение длины кортежа.

При описании этих и других операций со списками будем использовать следующие отображения и константы, заданные на множествах E, N, P:

- Info: $P \to E$, где Info (pos) – элемент списка, находящийся в позиции pos памяти.
- Next: $P \to P$, где Next (pos) – позиция элемента, следующего за элементом из позиции pos.
- Precede: $P \to P$, где Precede (pos) – позиция элемента, находящегося перед элементом из позиции pos.
- Number: $P \to N$, где Number (pos) – кортежный номер элемента, находящегося в позиции pos.
- Position: $N \to P$, где Position (k) – позиция элемента, имеющего кортежный номер k.
- Length – длина списка.
- First – позиция первого элемента списка.
- Last – позиция последнего элемента списка.

Иногда для распознавания концевых элементов списка пользуются следующим соглашением: если pos является позицией последнего элемента списка, то полагают Next (pos) = pos, а если началом, то – Preced (pos) = pos. В случаях когда позицией элемента, следующего за последним или предшествующего первому, является несуществующая позиция nil, будем считать, что nil принадлежит множеству P. При изменении содержимого списка все введенные множества, отображения и константы могут изменяться.

Различные представления кортежей с помощью списковых структур характеризуются степенью эффективности выполнения тех или иных операций. Так, при одном представлении эффективно вычисляется порядко-

вый номер элемента по его позиции, но менее эффективно осуществляется вставка нового элемента. При другом представлении легко осуществляется вставка, а определение порядкового номера элемента по его позиции требует заметного времени, например пропорционального длине списка. К сожалению, не всегда удается найти способ представления кортежей, для которого все используемые в конкретном алгоритме операции одинаково эффективны. Поэтому выбор структуры для представления кортежа сопряжен с анализом алгоритма, а именно, с выяснением того, какие операции, в конечном счете, определяют его суммарную трудоемкость. В зависимости от набора операций, которые предполагается выполнять со списками, выделяют некоторые их разновидности.

Список, в который предполагается добавлять и удалять элементы лишь с одного его конца, называется стеком. Если добавление элементов должно происходить с одного конца, а удаление с другого, то такой список называется очередью. Если удалять и добавлять элементы можно с любого конца списка, то такой список называется деком. Если добавлять можно с любого конца, а удалять только с одного, то список называется деком с ограниченным выходом. Списки более общего вида позволяют включать и удалять элементы из любой позиции. Списки классифицируются также по возможностям их сканирования (просмотра) в одном направлении (от начала к концу или от конца к началу) или в обоих направлениях. В первом случае список называется односторонним, во втором двусторонним.

Списки, для которых не планируется выполнять операции вставки/удаления из произвольной позиции, удобно представлять массивами. В этом случае кортежный номер элемента можно либо отождествить с номером элемента в массиве, либо сделать легко вычисляемым по этому номеру. Такие списки называют списками с прямым доступом к элементам. Другими словами, под прямым доступом мы понимаем возможность по номеру элемента в кортеже за единицу времени определять как сам элемент, так и его позицию в памяти.

При представлении кортежей, для которых планируется выполнение операций вставки/удаления элемента из произвольной позиции, используется возможность нахождения программным путем свободного пространства в памяти для размещения вставляемого элемента. При использовании языков программирования высокого уровня эти обязанности обычно берет на себя система программирования (оператор new – в языках PASCAL и C). При вставлении нового элемента в список место, куда он вставляется, указывается с помощью косвенной адресации. Это может быть адрес элемента, перед которым либо после которого, вставляется новый элемент,

либо и тот, и другой. Такой способ дает возможность лишь последовательного доступа к элементам. Другими словами, при последовательном доступе гарантируется определение за единицу времени позиции очередного элемента лишь по позиции предыдущего или следующего за ним элемента, но не по его номеру в кортеже.

Отметим еще, что при конструировании списков иногда удобно элементами списка считать не сами элементы множества *E*, а их позиции в памяти. В этом случае списки по терминологии Р. Тарьяна называются экзогенными (внешними), в противном случае – эндогенными (внутренними). Эндогенный способ используют в тех случаях, когда элементы множества *E* для своего представления требуют большого пространства и переписывание элемента из одного участка памяти в другой сопряжено с большими затратами времени.

Преимущество использования эндогенных списков вместо экзогенных может ярко проявиться в такой задаче, как сортировка (упорядочение) элементов списка в порядке возрастания или убывания некоторой числовой характеристики, называемой ключом элемента. Основными операциями при сортировке, определяющими ее трудоемкость, являются операция сравнения ключей и операция перемещения элементов. Очевидно, что при эндогенном способе представления списка суммарные затраты времени на перемещения элементов могут оказаться намного меньше, чем при экзогенном способе.

Возможность более быстрой перестановки коротких адресов по сравнению с перестановкой длинных записей не исчерпывает преимуществ эндогенного представления списков. Предположим, что нам одновременно понадобилось не одно, а несколько разных упорядочений одного и того же списка по разным критериям. Для этого достаточно ввести несколько адресных массивов, при упорядочении по каждому критерию переставляются элементы соответствующего адресного массива.

При описании операций над списками будем считать, что каждый из них описан с помощью дескриптора, имеющего форму записи, состоящей из нескольких полей. Дескриптор предназначен для того, чтобы явно указать составные части, из которых формируется список. Как правило, в дескриптор будет входить поле first, содержащее позицию первого элемента. Если в конструкции списка используется некоторый массив, то в дескрипторе указывается имя этого массива.

Поля, относящиеся к конкретному списку *L*, будем записывать в форме

L.<имя_поля>.

В полях дескриптора будем указывать также имена процедур и функций, которые реализуют примитивные операции над списками при рассматриваемой их реализации.

Так, например, дескриптор списка L может иметь форму

[first, length].

При таком дескрипторе

- L.first означает позицию первого элемента списка L,
- L.length – его длину.

1.2. Списки с прямым доступом

Прямой доступ, как правило, реализуется при представлении списка массивом. Элементы кортежа размещаются в идущих подряд ячейках некоторого массива. Для локализации списка в массиве введем целочисленную переменную first для хранения номера позиции массива, в которой расположен его первый элемент, и целочисленную переменную length, означающую длину списка. Равенство length = 0 является признаком того, что массив содержит пустой список. Иногда для переменных, хранящих позицию элемента массива, удобно иметь какое-либо условное значение, выходящее за рамки индексации массива. Будем обозначать его beyond.

Рассмотрим подробнее реализацию списка с прямым доступом, дающую возможность добавлять элементы к списку с любого его конца. Воспользуемся циклической «нумерацией» элементов массива, при которой следующим за последним элементом массива считается его первый элемент, а предыдущим для первого – последний (речь идет об элементах массива, а не об элементах списка). Если элементы массива пронумеровать числами от 0 до $n - 1$, то переход к следующему (предыдущему) элементу списка осуществляется с помощью прибавления (вычитания) единицы по модулю n, где n – длина массива.

Дескриптор такого списка будет иметь форму

S = [n, info, first, length].

Добавление элемента к началу списка осуществляется его записью в позицию newfirst = (first – 1) mod n ($0 \le$ newfirst $< n$) с последующим присваиванием first := newfirst, а добавление в конец – записью элемента в позицию (first + length) mod n с последующим выполнением оператора length := (length + 1). Заметим, что при таком способе начальный фрагмент кортежа может оказаться в конце массива, а конечный фрагмент – в

начале. Заметим также, что добавление нового элемента возможно только при условии length < *n*. Следует заметить также, что в системах программирования со статическим распределением памяти под массивы, которое происходит во время компиляции, длину массива следует выбирать достаточной для размещения списков, порождаемых разрабатываемым алгоритмом. Следует иметь в виду, что максимальная длина списка зависит не только от алгоритма, но и от входных данных.

Основные отображения для списка с прямым доступом, имеющего дескриптор S = [*n*, info, first, length], определяются следующим образом:

Info (pos) = S.info [pos],

Next (pos) = **if** (pos = S.first+S.length−1) **then** pos
else if (S.length < S.n) **then** (pos + 1) mod S.n **else** beyond,

Preced (pos) = **if** (pos = S.first) **then** pos
else if (S.length < S.n) **then** (pos − 1) mod S.n **else** beyond,

Last = (S.first + S.length − 1) mod S.n,

Number (pos) = **if** (S.first < pos) **then** (pos − S.first + 1)
else (S.pos − S.first + S.n − 1).

Приведем несколько процедур для работы со списками с прямым доступом.

Создать пустой список S

```
procedure SetEmpty (S);
begin S.first:=0; S.length:=0 end;
```

Добавить элемент e к концу списка S

```
procedure AddToEnd (e, S);
begin
  if S.length < S.n
  then {S.info[(S.first + S.length) mod S.n] := e; S.length := S.length + 1}
    else 'массив переполнен'
end;
```

Добавить элемент e к началу списка S

```
procedure AddToBegin (e, S);
```

```
begin
    if S.length < S.n
    then {S.first := S.first–1; S.info[S.first] := e; S.length := S.length + 1}
    else 'массив переполнен'
end;
```

Заменить элемент с кортежным номером k на элемент e

```
procedure Set (k, e, S);
begin
S.info [S.first + k – 1] := e
end;
```

Удалить последний элемент списка S

```
procedure DelLast (S);
begin
    if S.length > 0 then S.length:= S.length – 1
    else 'список пуст'
end;
```

Удалить первый элемент списка S

```
procedure DelFirst (S);
begin
    if S.length > 0
    then {S.first := (S.first + 1) mod S.n; S.length := S.length – 1}
    else 'список пуст'
end;
```

1.3. Списки с последовательным доступом

Последовательный доступ к элементам списка, как правило, реализуется с использованием динамического выделения памяти во время исполнения программы. Поиск свободных участков памяти обычно возлагается на систему программирования. Мы будем называть такие списки связными. Преимущества связных списков перед списками с прямым доступом проявляются в тех случаях, когда часто используются вставки в списки и удаления элементов из списков. Еще одно преимущество динамического выделения памяти может проявиться, когда в алгоритме одновременно используется большое количество списков, каждый из которых в процессе работы может потребовать большой объем памяти, однако в совокупности эта память может быть ограничена приемлемой величиной.

Элементы связного списка, следующие друг за другом, не обязательно размещаются в последовательных ячейках памяти, доступ к следующему и предыдущему элементам осуществляется при помощи специальных ссылок (указателей). Чтобы обеспечить запоминание указателей на следующий и предыдущий элементы, каждый элемент списка «погружается» в узел, для которого в памяти компьютера формируется запись, состоящая из нескольких полей. В простейшем случае эта запись может состоять из двух полей. Одно из них – Info предназначено для запоминания самого элемента, а другое – Next для запоминания позиции следующего. Для обозначения такого узла будем использовать следующую форму

t: [Info, Next],

где *t* – позиция (адрес) узла в памяти. Поскольку у последнего элемента нет следующего, его поле Next заполняют значением nil. Иногда вместо nil используют ссылку на самого себя, что также может являться признаком конца списка. Мы часто будем пользоваться именно этим способом распознавания конца списка. Представление списка с помощью таких узлов обеспечивает сканирование списка от начала к его концу. Доступ к самому списку осуществляется через его голову с помощью переменной first, содержащей позицию первого элемента. Такие списки называются односторонними.

При описании операций со списками через *t*^ будем обозначать узел, расположенный в позиции *t*. Для доступа к полям узла *t*^ используем форму *t*^.Info, *t*^.Next и т. д.

Оператор для создания нового узла будем записывать в виде

Create (*t*: [Info, Next]).

Для обеспечения сканирования как от начала к концу, так и от конца к началу используют узлы следующего вида

t: [Info, Next, Preced].

Поле *t*^.Preced используется для запоминания позиции элемента, предшествующего элементу, находящемуся в позиции *t*. Доступ к такому списку может осуществляться как через его начало с помощью переменной first, так и через конец с помощью переменной last. Такие списки называются двусторонними.

На рис. 1 – 6 представлено несколько разновидностей списков. Узлы списков изображены прямоугольниками, разделенными на части по числу полей. Стрелки проведены в соответствии со значениями полей Next и

Preced.

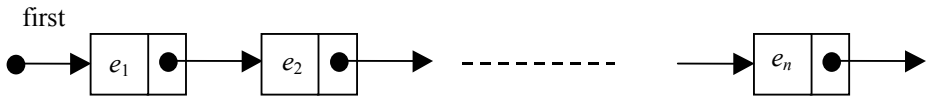


Рис. 1. Односторонний список: вход через первый элемент,
сканирование от начала к концу, признак конца – Next(pos) = nil

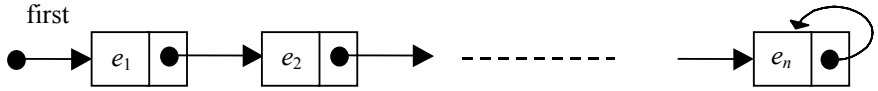


Рис. 2. Односторонний список: вход через первый элемент; сканирование
от начала к концу, признак конца – Next(pos) = pos

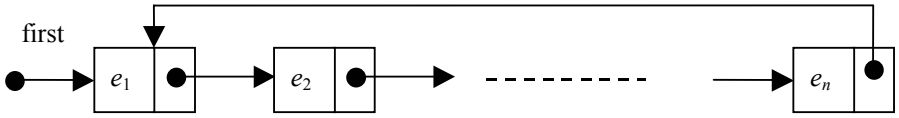


Рис. 3. Односторонний *циклический* список: вход через первый элемент;
сканирование от начала к концу, признак конца – Next(pos) = first

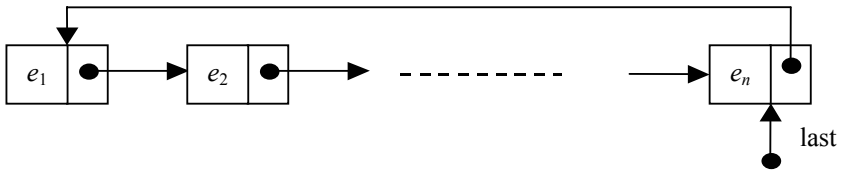


Рис. 4. Односторонний циклический список: вход через последний элемент с помощью ссылки Last^.next; сканирование от начала к концу, признак конца – pos = last

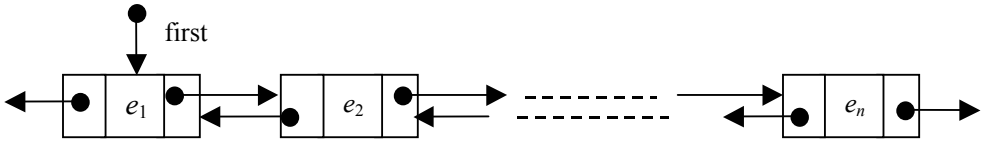


Рис. 5. Двусторонний список: вход через первый элемент, сканирование от начала
к концу и от конца к началу; признак начала – Proced(pos) = nil;
признак конца – Next(pos) = nil

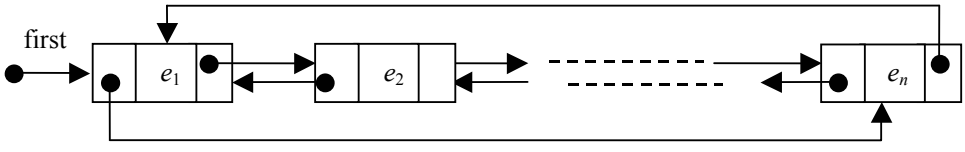


Рис. 6. Двусторонний циклический список: вход через первый элемент; сканирование
от начала к концу и от конца к началу, признак конца – Next(pos) = first

Рассмотрим основные отображения и операции на примере двустороннего списка, сформированного из узлов вида *t*: [Info, Next, Preced].

Считаем, что дескриптор списка S имеет вид: [first, last].

Ниже условимся считать, что признак конца – Next (pos) = nil, а признак начала – Preced (pos) = nil.

Основные отображения определяются следующим образом.

Info (pos) = pos^. info,

Next (pos) = pos^. next,

Preced (pos) = pos^. preced,

Создать пустой список *S*

```
SetEmpty (S);
begin S. first := nil end;
```

Удалить из списка S элемент, находящийся в позиции pos памяти

```
procedure Del(S, pos);
begin
    t := pos^.preced; u := pos^.next;
    t^.next := u; u^.preced := t
end;
```

Замечание. В теле процедуры Del отсутствует параметр S. При ее использовании могут возникнуть проблемы, связанные с некорректным обращением, так как в процедуре не производится проверка того, является ли позиция pos позицией какого-либо элемента списка S. Ответственность за некорректное обращение несет вызывающая программа. Проверка этого условия с помощью сканирования списка могла бы оказаться слишком дорогой и свела бы на нет преимущества использования связных списков. Еще одна проблема, связанная с использованием этой операции, заключается в том, что узел pos^ может оказаться недоступным при потере значения переменной pos, но память будет оставаться занятой. Если это нежелательно, следует, воспользовавшись системными средствами, освободить занимаемую узлом память. Замечания по поводу некорректного обращения будут справедливы и для некоторых следующих процедур, однако, мы не будем каждый раз напоминать об этом.

Вставить в список S элемент e после элемента, находящегося в позиции pos

```
procedure InsertAfter(S, pos, e);
begin
    create (t: [e, pos^.next, pos]);
    pos^.next^.preced := t;
    pos^.next := t
end;
```

Следующие две операции рассмотрим на примере одностороннего циклического списка (см. рис. 4).

Добавить элемент e к концу списка S

```
procedure AddToEnd(e, S);
begin
    create (t: [e, S.last^.next]);
    S.last:= t;
end;
```

Добавить элемент e к началу списка S

```
procedure AddToBegin (e, S);
begin
    create(t: [e, S^.last^.next]);
    S.last^.next := t
end;
```

Следующие три процедуры рассмотрим на примере двустороннего циклического списка (см. рис. 6).

Удалить последний элемент списка S

```
procedure DelLast (S);
begin
    t := S.first^.preced^.preced;
    t^.next := S.first;
    S.first^.preced := t
end;
```

Удалить первый элемент списка S

```
procedure DelFirst(S);
begin
    t := S.first^.next;
```

```
    S.first^.preced^.next := t;
    S.first := t
end;
```

Удалить из списка S элемент, находящийся в позиции pos

```
procedure DelPosition (S, pos);
begin
    if pos = S.first
    then DelFirst (S)
    else
    if pos = S.last
    then DelLast (S)
    else {t^.preced^.next := t^.next; t^.next^.preced := t^.preced}
end;
```

1.4. Некоторые дополнительные операции со связными списками

Конкатенация. Эта операция предназначена для соединения двух списков в один результирующий. Она выполняется эффективно в тех случаях, когда обеспечен доступ к последнему элементу списка с трудоемкостью $O(1)$. При соединении двух списков $S1$ и $S2$ первый элемент списка $S2$ становится преемником последнего элемента списка $S1$. При этом возникают вопросы – должен ли получившийся список иметь какое-то новое имя и должны ли сохраниться как таковые исходные списки $S1$ и $S2$?

В рассмотренной ниже процедуре Concat, реализующей операцию «Соединить два списка», принято следующее решение. К списку $S1$ присоединяется список $S2$, список $S2$ сохраняется, а результирующим является список $S1$. Следует, однако, понимать, что если в список $S2$ будут внесены изменения, они автоматически произойдут в новом списке. Трудоемкость этой операции – $O(1)$.

```
begin S1. last^. next := S2.first end;
```

Из списка S удалить элементы, удовлетворяющие некоторому условию. Предположим, что требуемое условие на элемент e проверяется предикатом condition(e).

```
begin
    t := S.first;
```

```
    while t ≠ nil do
    {if condition (t^.info) then DelPosition (S, pos); t := t^.next}
end;
```

Построить список S1, состоящий из элементов данного списка S, удовлетворяющих некоторому условию. Предположим, что требуемое условие на элемент *e* проверяется предикатом condition (*e*).

```
begin
    t := S.first; SetEmpty(S1);
    while t ≠ nil do
    {if condition(t^.info) then AddToEnd (e, S1); t := t^.next}
end;
```

Получить список *S*1 **реверсированием списка S**

```
begin
     SetEmpty(S1);
    t := S.first;
    while  t ≠ nil do {AddToBegin (t^.inf, S1);  t := t^.next}
end;
```

1.5. Моделирование списков с последовательным доступом при помощи массивов

Если использование динамических ссылок невозможно или нежелательно (тому могут быть свои причины), список со связями можно смоделировать при помощи массивов. В массиве Inf хранятся элементы списка, то есть значения соответствующих полей узлов списка со связями. Позицией элемента является значение целочисленного индекса массива. Кроме того, вводится целочисленный массив Next, в котором для каждого узла списка указана позиция, где расположен его преемник. В качестве индексного пространства используем отрезок [1..*n*] целочисленного типа.

В одних и тех же массивах Inf и Next могут размещаться сразу несколько списков, состоящих из узлов одного типа. С учетом такого возможного сосуществования различных списков их элементы могут размещаться в этих массивах хаотично, подобно тому, как узлы списков, представленных с помощью ссылок, могут хаотично располагаться в памяти компьютера.

На рис. 7 показано возможное заполнение массивов Inf и Next для одностороннего списка, представляющего кортеж (*a*, *b*, *c*, *d*, *e*) (пустые клет-

клетки не имеют отношения к этому списку).

Адрес	1	2	3	4	5	6	7	8	9	10	11	12
Inf	*e*		*b*			*c*			*d*		*a*	
Next	0		6			9			1		3	

Рис. 7. Моделирование одностороннего списка при помощи массивов

Доступ к списку можно осуществить через его первый элемент, позиция которого в массиве задается значением переменной first = 11. Значение Next[1] = 0 говорит о том, что в позиции 1 расположен элемент, у которого нет преемника, то есть последний элемент кортежа.

На рис. 8 показано возможное заполнение массивов Inf, Next и Preced для представления кортежа (*a*, *b*, *c*, *d*, *e*) двусторонним списком.

Pos	1	2	3	4	5	6	7	8	9	10	11	12
Inf	*e*		*b*			*c*			*d*		*a*	
Next	0		6			9			1		3	
Preced	9		11			3			6		0	

Рис. 8. Моделирование двустороннего списка при помощи массивов

Основные отображения Info(pos), Next(pos), Preced(pos), First, Last, Length задаются очевидным образом. Если какие-либо из них не заданы явно, то их можно вычислять через другие сканированием списка.

Чтобы одни и те же массивы Info, Next, Preced использовать для одновременного хранения нескольких однотипных списков, позиции этих массивов объединяют в один так называемый свободный список Avail. Это можно сделать, например, с помощью операторов

```
begin
    Avail.first:=1;
    Next[n] := 0;
    for i := 1 to n – 1 do Next[i] := i + 1;
    Preced[1] := 0;
    for i := 2 to n do Preced[i] := i – 1;
end;
```

Массив Info при этом не заполняется. При создании новых списков используются элементы массивов Info, Next, Preced, предварительно удаляемые из списка Avail. В момент создания нового узла из списка Avail удаляется головной элемент, который и используется для добавления в новый список. С другой стороны, при удалении элемента из какого-либо списка освобождаемая позиция добавляется к свободному списку для последующего использования. Такая техника применялась, когда системы

программирования не имели стандартных средств динамического выделения памяти. Однако в условиях ограниченной памяти этот прием можно использовать и сейчас. Дело в том, что при достаточно большом объеме оперативной памяти стандартные системы вынуждены использовать многоразрядную адресацию, в то время как для позиционирования в массивах Info, Next, Preced можно использовать малоразрядные представления чисел.

1.6. Деревья и графы

Деревья находят широкое применение при проектировании алгоритмов и, в частности, структур данных. Отсылая читателя к литературе по теории графов, будем пользоваться такими понятиями, как узел, ребро, лист, потомок, сын, левый потомок, правый потомок, предок, отец, корень, ветвь и другие. Регулярным деревом назовем дерево, в котором фиксировано максимально возможное (как правило, небольшое) число потомков для каждого из его узлов. В частности, если число потомков для каждого узла не превосходит двух, то дерево называется бинарным, если не более трех – тернарным. Если это число может равняться только двум или трем, то дерево называется (2–3)-деревом.

Достаточно универсальным является способ представления регулярных деревьев, при котором каждый узел представляется записью, содержащей кроме прикладной информации также позиции смежных с ним элементов, например позиции потомков или наряду с потомками позицию предка или еще каких-либо узлов, в зависимости от потребностей. Регулярность дерева позволяет фиксировать число полей, достаточное для представления любого узла.

Так, узлы бинарного корневого дерева можно представлять записями вида

[Element, Left, Right],

где Element представляет связанную с узлом прикладную информацию, Left – позицию его левого потомка, а Right – позицию правого потомка. Само дерево в таком случае можно представить позицией его корня. Если в алгоритме необходимо продвижение от узла к предку, то узлы бинарного корневого дерева можно представлять записями вида

[Element, Left, Right, Father],

где Father – позиция предка рассматриваемого узла.

Для представления нерегулярных деревьев (то есть деревьев, узлы которых могут иметь произвольное число потомков) применяют следующий

способ: потомки каждого узла нумеруются и каждый узел представляется записью, включающей в себя позицию его первого (левого) потомка и позицию его «правого брата».

Для регулярных деревьев более экономным по памяти может оказаться представление с помощью массива. Рассмотрим этот прием на примере бинарного дерева. Значения индексов массива отождествляются с узлами дерева, пронумерованными так, что корень получает номер 1, а потомки узла с номером i получают номера $2i$ и $2i + 1$. При таком представлении предок узла с номером i будет иметь номер i div 2 (частное от деления i на 2). Аналогично можно представить тернарное и другие регулярные деревья.

Остановимся вкратце на представлении графов общего вида.

Обыкновенный граф с n вершинами часто представляют матрицей смежности, то есть матрицей размера $n \times n$, в которой элемент, расположенный в i-й строке и j-м столбце, равен 1, если вершины графа с номерами i, j соединены ребром, и равен 0, если такого ребра нет. Если граф не ориентирован, то его матрица смежности симметрична и можно ограничиться хранением ее треугольной части.

Матричный способ представления может оказаться неэкономным с точки зрения использования памяти, если граф разрежен. Так, например, известно, что число ребер связного планарного графа с n вершинами не превосходит величины $(3n - 6)$ при $n \geq 3$, то есть оценивается величиной $O(n)$, а не как в общем случае $O(n^2)$. Представлять такие графы матрицей смежности, как правило, нецелесообразно.

Другой способ представления графа – это список или массив пар вершин, соответствующих ребрам. При таком способе, если граф не ориентирован, то из двух возможных пар (i, j) и (j, i) целесообразно хранить только одну, например ту, у которой первая компонента меньше второй.

Еще один способ, часто имеющий преимущества перед указанными выше, – это представление графа массивом или списком списков (см. рис. 9).

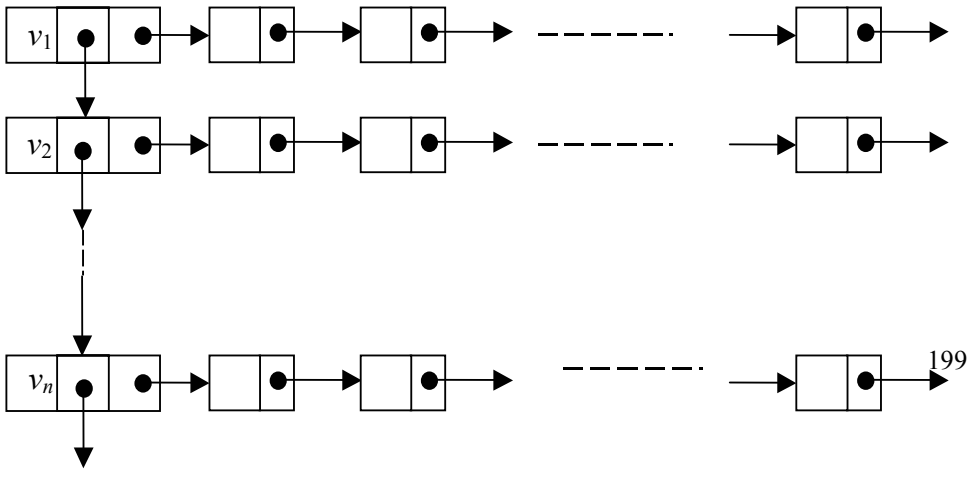


Рис. 9. Представление графа комбинацией списков: множество вершин представлено списком (v_1, v_2, …, v_n) узлов, к каждому из которых справа подцеплен список смежных с ним вершин

А именно, для каждой вершины организуется список смежных с ней вершин. В этом случае легко осуществляется доступ к окрестностям вершин. Примерно такого же эффекта можно достичь, представляя граф с помощью двух массивов: inf [1..m] и adr [1..(n + 1)], где m – число ребер графа. Массив adr назовем адресным, а inf – информационным. В информационном массиве вначале перечисляются номера вершин, смежных с первой вершиной, затем – со второй и так далее. В адресном массиве указываются номера позиций информационного массива так, чтобы для каждой вершины i по ним можно было находить фрагменты массива inf, в которых записаны номера вершин, смежных с этой вершиной. Например, adr[i] может хранить позицию, с которой начинаются в массиве inf вершины, смежные с i-й, при этом adr[n + 1] – первая позиция за пределами массива inf. В таком случае если нам требуется с каждой вершиной j из окрестности вершины i выполнить оператор $S(j)$, то можно сделать это с помощью оператора цикла

for k := adr[i] **to** adr [i + 1]–1 **do** S(inf[k]).

Заметим, что если граф не ориентирован, то каждое ребро (i, j) будет представлено дважды: один раз в последовательности вершин, смежных с вершиной i, а второй раз в последовательности вершин, смежных с вершиной j. Но эта избыточность часто бывает полезна с точки зрения времени выполнения операций над окрестностями вершин графа. Как недостаток такого представления графа можно отметить неудобство при динамической модификации графа, например добавление к графу ребра может потребовать большого количества пересылок в массиве inf. Этого недостатка лишен способ представления графа, показанный на рис. 9.

Глава 2. РАЗДЕЛЕННЫЕ МНОЖЕСТВА

Разделенные множества – это абстрактный тип данных, предназначенный для представления коллекции, состоящей из некоторого числа k попарно непересекающихся подмножеств U_1, U_2, ..., U_k заданного множества U. Для простоты в качестве U будем рассматривать множество $\{1, 2, \ldots, n\}$.

Этот тип данных применяется в таких задачах, как отыскание минимального остовного дерева для заданного взвешенного неориентированного графа, построение компонент связности графа, минимизация конечного автомата, и многих других, требующих динамического поддержания некоторого отношения эквивалентности. Примеры таких задач будут рассмотрены ниже.

Как правило, в таких задачах вычисления начинаются с пустой коллекции подмножеств ($k = 0$). Затем по мере вычислений формируются новые подмножества, включаемые в коллекцию. Формирование новых подмножеств происходит либо путем создания одноэлементного подмножества, либо путем объединения уже существующих в коллекции подмножеств. Для осуществления таких действий используются имена включенных в коллекцию подмножеств. В качестве имени подмножества будем использовать один из его элементов (главный элемент), выбираемый по определенному правилу. Поскольку в коллекции всегда будут находиться попарно непересекающиеся подмножества множества U, такое имя будет однозначно определять требуемое подмножество.

2.1. Операции над разделенными множествами

СОЗДАТЬ (x). Эта операция предназначена для введения в коллекцию нового подмножества, состоящего из одного элемента x, при этом предполагается, что x не входит ни в одно из подмножеств коллекции, созданной к моменту выполнения этой операции. Элемент x указывается в качестве параметра. Именем созданного подмножества будет считаться сам элемент x.

ОБЪЕДИНИТЬ (x, y). С помощью этой операции можно объединить два подмножества коллекции, имеющие, соответственно, имена x и y, в одно новое подмножество, при этом оба объединяемые подмножества

удаляются из коллекции, а вновь построенное подмножество получает некоторое имя. Во всех рассматриваемых нами случаях именем нового полученного в результате этой операции подмножества будет одно из имен x или y. Имена объединяемых подмножеств указываются в качестве параметров.

НАЙТИ (x, y). Эта операция позволяет определить имя y того подмножества коллекции, которому принадлежит элемент x. Если элемент x до выполнения операции не входил ни в одно из подмножеств коллекции, то в качестве y берется 0.

Последовательность σ, составленную из операций типа СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ, назовем *корректной*, если перед выполнением каждой операции из последовательности σ выполнены условия ее применения. Например, перед выполнением очередной операции вида ОБЪЕДИНИТЬ (x, y) подмножества с именами x и y должны быть уже созданы. Перед выполнением операции СОЗДАТЬ (x) элемент x не должен принадлежать ни одному из подмножеств коллекции. Операция НАЙТИ (x, y) применима при любом значении аргумента $x \in U$. Следует только помнить, что если x не принадлежит ни одному из подмножеств коллекции, то получим $y = 0$.

Мы рассмотрим несколько способов представления коллекции разделенных множеств в памяти компьютера и алгоритмической реализации перечисленных операций. А именно, будут рассмотрены представления

- с помощью массива;
- с помощью древовидной структуры;
- с помощью древовидной структуры с использованием рангов вершин;
- с помощью древовидной структуры с использованием рангов вершин и сжатия путей.

Последний из перечисленных способов является наиболее эффективным по времени выполнения произвольных корректных последовательностей операций типа СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ. Строго говоря, во всех перечисленных случаях будут использоваться массивы, но интерпретации их содержимого будут различными. Каждый раз при описании очередной реализации мы будем обсуждать оценки трудоемкости рассматриваемых операций.

2.2. Примеры использования разделенных множеств

Пример 1. Рассмотрим задачу выделения компонент связности неориентированного графа. Напомним, что компонентой связности называется максимальное по включению подмножество вершин графа такое, что любые две его вершины связаны цепью. Полагаем, что вершины графа пронумерованы числами 1, 2, ..., n и каждое ребро представлено парой (i, j) номеров вершин. Предполагаем также, что множество ребер непусто.

Алгоритм выделения компонент связности неориентированного графа

1. Создать коллекцию из n одноэлементных подмножеств множества $\{1, 2, ..., n\}$;
2. Прочитать очередное ребро (i, j);
3. Найти имя a подмножества коллекции, содержащего элемент i;
4. Найти имя b подмножества коллекции, содержащего элемент j;
5. Если $a \neq b$, то объединить подмножества с именами a и b;
6. Если есть еще непрочитанные ребра, перейти к п. 2, в противном случае закончить вычисления.

Очевидно, построенные подмножества коллекции будут представлять искомые компоненты связности. Используя названия основных операций над коллекцией разделенных множеств, представленный выше алгоритм можно записать в следующем виде

1. **For** i:= 1 **to** n **do** СОЗДАТЬ (i);
2. Прочитать очередное ребро (i, j);
3. НАЙТИ (i, a);
4. НАЙТИ (j, b);
5. **if** $a \neq b$ **then** ОБЪЕДИНИТЬ (a, b);
6. Если есть еще непрочитанные ребра, перейти к п. 2, в противном случае закончить вычисления.

Пример 2. Рассмотрим неориентированный связный граф без петель, ребрам которого приписаны в качестве весов вещественные числа. Требуется построить остовное дерево, накрывающее все вершины графа и имеющее минимальный суммарный вес входящих в него ребер. Итак, пусть заданный граф G имеет множество V вершин, пронумерованных числами 1, 2, ..., n, и множество E ребер. Каждому ребру e из множества E поставлена в соответствие пара $(N(e), K(e))$ его концевых вершин и число $C(e)$ – его вес. Для решения этой задачи были предложены различные алгоритмы. Мы рассмотрим алгоритм, который разработал Краскал.

Алгоритм Краскала

1. Создать коллекцию из n одноэлементных подмножеств множества $\{1, 2, ..., n\}$;
2. Создать пустое множество T;
3. В множестве E найти ребро e с минимальным весом и удалить его из множества E;
4. Найти имя a подмножества коллекции, содержащего элемент $N(e)$;
5. Найти имя b подмножества коллекции, содержащего элемент $K(e)$;
6. Если $a \neq b$, то объединить подмножества с именами a и b, а ребро e добавить к множеству T;
7. Если множество E не пусто и $|T| < n - 1$, перейти к п. 3, в противном случае закончить вычисления.

Заметим, что в процессе работы алгоритма в множестве T будут находиться ребра, составляющие ациклический подграф исходного графа, являющийся лесом, состоящим из некоторого числа деревьев. Отсутствие циклов гарантируется проверкой «Если $a \neq b$» в пункте 6 описанного алгоритма. Фактически при $a \neq b$ происходит объединение двух поддеревьев в одно дерево с помощью ребра e, найденного на шаге 3.

Если исходный граф связен, как это сказано в постановке задачи, то построенное с помощью такого алгоритма множество T будет, очевидно, представлять дерево, накрывающее все вершины исходного графа. Доказательство того факта, что суммарный вес входящих в него ребер будет минимальным, можно найти в разделе «Графы».

В алгоритме естественным образом используется структура разделенных множеств. Обратим внимание на операцию поиска в множестве E ребра e с минимальным весом. Эффективность этой операции существенно зависит от выбора структуры данных для хранения множества E. Приемы эффективного выполнения этой операции рассмотрены в разделе «Приоритетные очереди».

2.3. Представление разделенных множеств с помощью массива

Пусть $U = \{1, 2, \ldots, n\}$ – множество, из элементов которого будет

строиться коллекция разделенных подмножеств.

Одним из очевидных способов представления коллекции является представление ее с помощью массива. При таком способе для каждого элемента i в соответствующей (i-й) ячейке массива помещаем имя (канонический элемент) того подмножества, которому принадлежит элемент i. Если элемент i не принадлежит ни одному из подмножеств коллекции, то в i-ю ячейку записываем 0.

Реализация операций с помощью массива. Обозначим через f массив длины n, с помощью которого будем представлять коллекцию. Пустая коллекция представляется массивом, заполненным нулями.

Операция СОЗДАТЬ (x) осуществляется записью элемента x в ячейку с номером x. Время выполнения операции – $O(1)$.

Операция ОБЪЕДИНИТЬ (x, y) осуществляется следующим образом. Просматриваются элементы массива f, и в те ячейки, в которых было записано имя x, заносится новое имя – y. Следовательно, именем вновь образованного подмножества будет y, а x перестанет быть именем какого-либо подмножества. Очевидно, время выполнения этой операции – $O(n)$.

Операция НАЙТИ (x, y) выдает в качестве y содержимое элемента с номером x в массиве f. Время выполнения операции – $O(1)$.

При такой реализации разделенных множеств, очевидно, время выполнения m произвольных операций, среди которых $O(n)$ операций ОБЪЕДИНИТЬ, есть величина $O(m \cdot n)$.

2.4. Представление разделенных множеств с помощью древовидной структуры

Пусть, по-прежнему, $U = \{1, 2, \ldots, n\}$ – множество, из элементов которого будет строиться коллекция.

Каждое подмножество коллекции представляется корневым деревом, узлы которого являются элементами этого подмножества, то есть отождествляются с номерами из множества $\{1, 2, \ldots, n\}$. Корень дерева используется в качестве имени соответствующего подмножества (канонический элемент). Для каждого узла дерева определяется узел $p(x)$, являющийся его родителем в дереве; если x – корень, то полагаем $p(x) = x$.

Фактически в памяти компьютера это дерево будем представлять массивом $p[1..n]$ так, что $p(x)$ будет предком узла x, если x не является корнем, и $p(x) = x$, если x – корень. Если же x не входит ни в одно из подмножеств коллекции, то $p(x) = 0$.

Рассмотрим пример. Пусть $U = \{1, 2, \ldots, 7\}$ и коллекция состоит из

двух подмножеств {1, 2, 3, 7} и {4, 6}. Деревья, представляющие эти подмножества, могут быть такими, как на рис. 1. Кружочки обозначают узлы дерева; указатели на родителей представлены при помощи стрелок. Именем одного из этих подмножеств является 3, другого – 6:

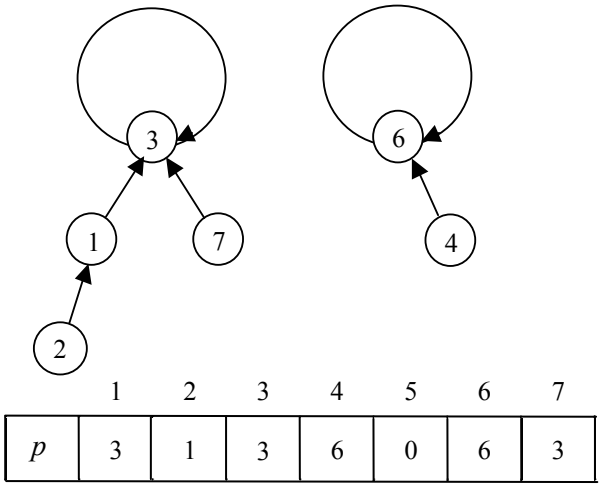


Рис. 1

Реализация операций с помощью древовидной структуры

Операция СОЗДАТЬ(*x*) назначает в качестве родителя узла *x* сам узел *x* с помощью присваивания *p*[*x*] := *x*. Таким образом, время выполнения операции есть *O*(1). В результате выполнения операции СОЗДАТЬ(*x*) образуется новое одновершинное дерево с петлей в корне, изображенное на рис. 2.

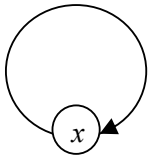
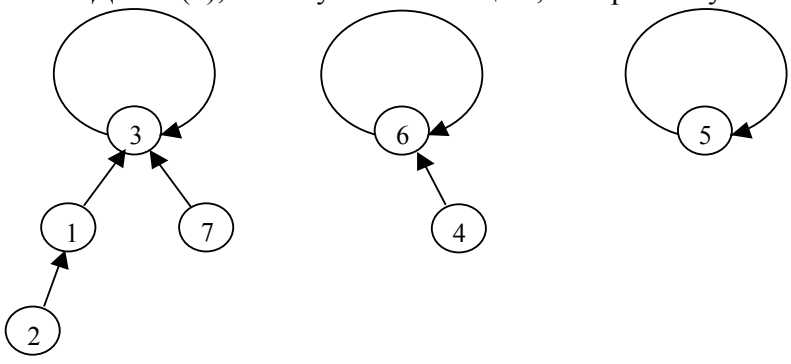


Рис. 2

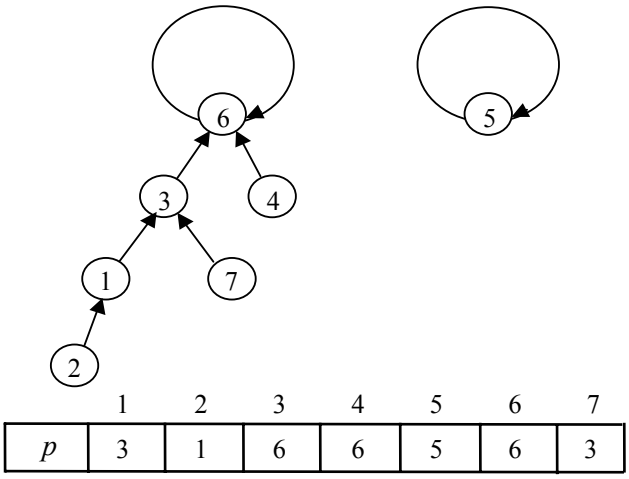
Если к коллекции подмножеств, изображенных на рис. 1, применить операцию СОЗДАТЬ (5), то получим коллекцию, изображенную на рис. 3.



	1	2	3	4	5	6	7
p	3	1	3	6	5	6	3

Рис. 3

Операция ОБЪЕДИНИТЬ (x, y) назначает узел y родителем узла x с помощью присваивания $p[x] := y$. Заметим, что x и y должны быть до выполнения рассматриваемой операции корнями соответствующих деревьев. Именем вновь образованного подмножества будет y, а x перестанет быть именем какого-либо множества. Время выполнения этой операции есть $O(1)$.



	1	2	3	4	5	6	7
p	3	1	6	6	5	6	3

Рис. 4

Если применить операцию **ОБЪЕДИНИТЬ** $(3, 6)$ к коллекции, представленной на рис. 3, то получим коллекцию, состоящую из двух подмножеств $\{1, 2, 3, 4, 6, 7\}$ и $\{5\}$, изображенную на рис. 4. Именем первого из этих подмножеств будет 6, второго – 5.

Операция НАЙТИ (x, y) осуществляется продвижением по указателям на родителей от узла x до корня дерева. В качестве y берется этот ко-

рень. Описанные действия можно реализовать с помощью операторов

```
while p[x] ≠ x do x := p[x]; y := x;
```

Очевидно, время выполнения данной операции есть $O(h)$, где h – длина пути из узла x в корень соответствующего дерева. Но заметим, что при выполнении операций СОЗДАТЬ и ОБЪЕДИНИТЬ возможно образование дерева в виде линейной цепочки из n узлов, изображенной на рис. 5.

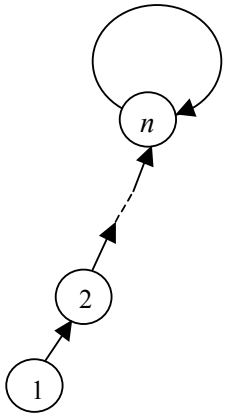


Рис. 5

К такой цепочке может привести, например, последовательность операций

```
СОЗДАТЬ (1);
СОЗДАТЬ (2);
……..
СОЗДАТЬ (n);
ОБЪЕДИНИТЬ (1, 2);
ОБЪЕДИНИТЬ (2, 3);
…….
ОБЪЕДИНИТЬ (n – 1, n);
```

Как видим, h может достигать величины n, поэтому трудоемкость операции НАЙТИ является величиной $O(n)$.

Худший случай применения операции НАЙТИ в данной ситуации – это НАЙТИ $(1, y)$. В этом случае необходимо сделать $n - 1$ переход по ссылкам на родителей, чтобы дойти от узла 1 к корню дерева n, и один переход, чтобы узнать, что родитель узла n есть сам узел n.

Если количество выполнений операции СОЗДАТЬ равно n, то время выполнения последовательности, составленной из m операций

ОБЪЕДИНИТЬ и/или НАЙТИ, при рассматриваемой реализации разделенных множеств есть величина $O(m \cdot n)$. Действительно, время выполнения m операций ОБЪЕДИНИТЬ, очевидно, есть $O(m)$, так как время выполнения одной такой операции есть константа. Время выполнения m операций НАЙТИ есть $O(m \cdot n)$, так как время выполнения одной такой операции есть $O(n)$. Итак, время выполнения m произвольных операций есть $O(m \cdot n)$.

2.5. Представление разделенных множеств с использованием рангов вершин

Предыдущую реализацию разделенных множеств можно усовершенствовать следующим образом. Операцию ОБЪЕДИНИТЬ можно выполнить так, чтобы высота дерева, соответствующего объединению двух множеств, была как можно меньше. А именно, корень большего по высоте дерева сделать родителем корня другого дерева. Назовем такую реализацию операции ОБЪЕДИНИТЬ объединением по рангу. В качестве ранга в данном случае берется высота соответствующего дерева.

Реализация операций с использованием рангов вершин. Для такой реализации разделенных множеств необходимо хранить с каждым узлом x дополнительно еще одну величину – высоту поддерева, корнем которого является узел x. Будем называть ее высотой, или рангом, узла x. Остальные операции нужно настроить на корректную работу с этим полем. Будем хранить высоту каждого узла x в ячейке $h[x]$ массива h.

Операция СОЗДАТЬ (x) назначает в качестве родителя узла x тот же самый x, а высотой узла x считает 0. Таким образом, время выполнения данной операции есть $O(1)$. В результате выполнения операции СОЗДАТЬ (x) образуется новое дерево, изображенное на рис. 6. Число, расположенное рядом с узлом, обозначает его высоту. Описанные действия реализуются с помощью операторов

$p[x] := x;\ h[x] := 0;$

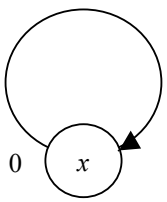


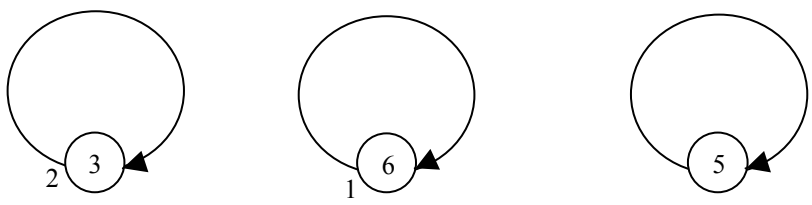
Рис. 6

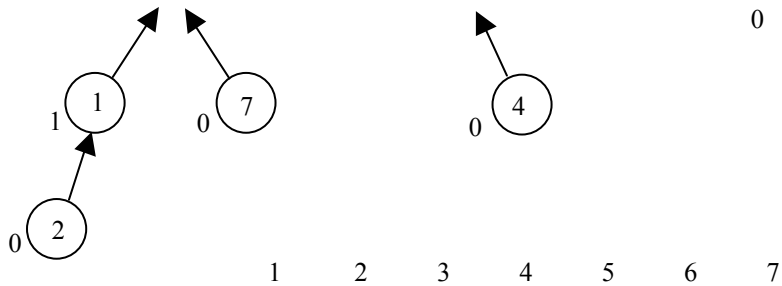
Операция ОБЪЕДИНИТЬ (x, y) назначает корень большего по высоте дерева родителем корня другого дерева. Если деревья имеют одинаковую высоту, то узел y назначается родителем узла x, после чего значение высоты узла y увеличивается на единицу. Заметим, что x и y должны быть до выполнения операции корнями соответствующих деревьев. Именем вновь образованного подмножества будет имя того из объединяемых подмножеств, у которого корень имел большую высоту, а имя другого из объединяемых подмножеств перестанет быть именем какого-либо из подмножеств. Очевидно, время выполнения этой операции есть константа. Выполнить описанные действия можно с помощью следующей процедуры.

```
Procedure ОБЪЕДИНИТЬ (x, y);
begin
   if (h[x] < h[y]) then p[x] := y
   else if (h[x] > h[y]) then p[y] := x
   else {p[x] := y; h[y] := h[y] + 1}
end;
```

На рис. 7 и 8 показано применение операции ОБЪЕДИНИТЬ (3, 6) к коллекции, изображенной на рис. 3, с учетом высот объединяемых поддеревьев. Рядом с кружочками, изображающими узлы, показаны их высоты. Так как $h(3) = 2 > h(6) = 1$, то родителем узла 6 становится узел 3.

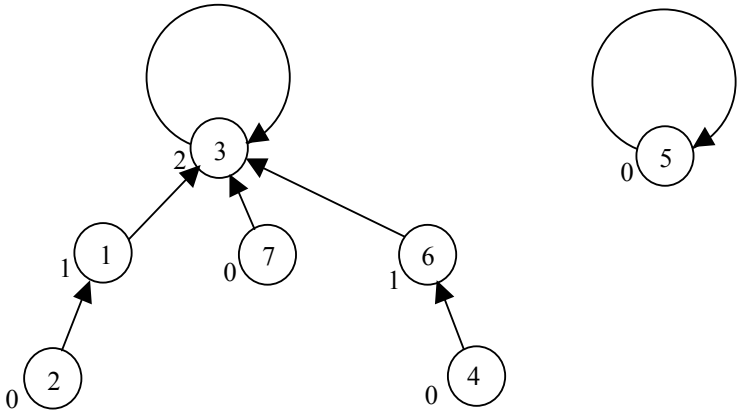
Операция НАЙТИ (x, y) осуществляется, так же как и в предыдущей реализации, продвижением по указателям на родителей от узла x до корня дерева. В качестве y берется найденный корень. Например, результатом применения операции НАЙТИ (4, y) к коллекции, изображенной на рис. 8, будет $y = 3$.





	1	2	3	4	5	6	7
p	3	1	3	6	5	6	3
h	1	0	2	0	0	1	0

Рис. 7



	1	2	3	4	5	6	7
p	3	1	3	6	5	3	3
h	1	0	2	0	0	1	0

Рис. 8

Очевидно, время выполнения данной операции, как и ранее, пропорционально длине пути из узла x в корень соответствующего дерева. Однако длина такого пути в данном случае может быть оценена иначе. Для оценки длины этого пути докажем следующие леммы.

Лемма 1. В результате выполнения любой последовательности операций из набора {СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ} над пустой коллекцией разделенных множеств для любого узла x выполняется неравенство $n[x] \geq 2^{h(x)}$, где $n[x]$ – количество узлов в поддереве с корнем x, $h[x]$ – высота узла x.

Доказательство. Очевидно, перед первым применением операции

ОБЪЕДИНИТЬ для любого узла x имели $n[x] = 1$, $h[x] = 0$ и, следовательно, $n[x] \geq 2^{h(x)}$. Операции СОЗДАТЬ и НАЙТИ не могут нарушить доказываемого неравенства, поэтому доказательство можно провести индукцией по количеству применений операции ОБЪЕДИНИТЬ.

Предположим, что перед очередным применением операции ОБЪЕДИНИТЬ (x, y) доказываемое неравенство все еще остается верным, тогда если высота узла x меньше высоты узла y, то дерево, полученное с помощью ОБЪЕДИНИТЬ (x,y), имеет корень y, а высоты узлов x и y не изменились. Количество узлов в дереве с корнем x не изменилось, а количество узлов в дереве с корнем y увеличилось. Таким образом, как для узлов x, y, так и для всех остальных неравенство сохраняется. Случай, когда высота узла x больше высоты узла y, аналогичен рассмотренному.

Если же высоты деревьев с корнями x и y до выполнения операции были одинаковы ($h[x] = h[y] = h$), то узел y становится родителем узла x, высота узла y увеличивается на 1, а высота узла x не изменяется. Пусть после выполнения операции величины $h[x]$, $h[y]$, $n[x]$, $n[y]$ становятся равными соответственно $h'[x]$, $h'[y]$, $n'[x]$, $n'[y]$, тогда имеем $h'[y] = h[y] + 1$, $h'[x] = h[x]$, $n'[x] = n[x]$, $n'[y] = n[y] + n[x]$. По предположению индукции, имеем $n[y] \geq 2^{h[y]}$ и $n[x] \geq 2^{h[x]}$. Следовательно, после выполнения рассматриваемой операции для узлов x и y имеем соотношения $n'[x] = n[x] \geq 2^{h[x]} = 2^{h'[x]}$ и $n'[y] = n[y] + n[x] \geq 2^{h[y]} + 2^{h[x]} = 2^{h+1} = 2^{h'[y]}$. Таким образом, утверждение леммы остается верным и в этом случае.

Лемма 2. Если за время работы, начавшейся с пустой коллекции, операция СОЗДАТЬ применялась n раз, то для любого $h \geq 0$ число k узлов высоты h удовлетворяет неравенству $k \leq n/2^h$.

Доказательство. Пусть $x_1, x_2, ..., x_k$ – все узлы высоты h, тогда по лемме 1 при $i = 1, 2, \ldots, k$ справедливы неравенства $n[x_i] \geq 2^h$. Следовательно,

$$n \geq n[x_1] + n[x_2] + ...+ n[x_k] \geq k2^h,$$

откуда и следует требуемое неравенство $k \leq n/2^h$.

Следствие 1. В результате выполнения любой последовательности операций из набора {СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ} над пустой коллекцией разделенных множеств для любого узла x имеет место неравенство $h[x] \leq \log n$.

Доказательство. Дерево максимальной высоты образуется, очевидно, лишь тогда, когда все n элементов объединяются в одно множество. Для

такого дерева количество k узлов максимальной высоты h равно 1, по лемме 2 имеем $1 = k \le n / 2^h$, откуда $2^h \le n$ и, следовательно, $h \le \log n$.

Следствие 2. Время выполнения операции НАЙТИ есть $O(\log n)$.

Следствие 3. При реализации разделенных множеств с использованием рангов время выполнения m операций ОБЪЕДИНИТЬ и/или НАЙТИ есть величина $O(m \cdot \log n)$.

Замечание. При реализации операции объединения подмножеств в качестве ранга узла можно использовать количество узлов в поддереве с корнем в данном узле. Утверждение леммы 1 будет справедливым и в этом случае, следовательно, сохранятся и оценки времени выполнения операций.

2.6. Представление разделенных множеств с использованием рангов вершин и сжатия путей

Предыдущий способ реализации разделенных множеств можно еще улучшить за счет усовершенствования реализации операции НАЙТИ (x, y). Она теперь будет выполняться в два прохода. При первом проходе находится корень у того дерева, которому принадлежит x. При втором проходе из x в y все встреченные узлы делаются непосредственными потомками узла y. Этот прием, как увидим ниже, намного уменьшает время выполнения последующих операций НАЙТИ.

Реализация операций с использованием рангов вершин и сжатия путей. Рассматриваемая реализация не требует новых полей данных по сравнению с предыдущим случаем. Как и прежде, для каждого узла i будем хранить указатель $p[i]$ на его родителя и ранг $r[i]$, который теперь не обязательно будет равен высоте дерева с корнем i. Он будет равен этой высоте, если из последовательности применяемых операций удалить все операции НАЙТИ.

Операция СОЗДАТЬ (x) выполняется с помощью операторов

begin $p[x] := x$; $r[x] := 0$ **end**;

В качестве родителя узла x берется тот же самый x, а его рангом считаем 0. Таким образом, время выполнения операции есть константа.

Операция ОБЪЕДИНИТЬ (x, y) выполняется как и прежде, разница лишь в том, что вместо массива h используется массив r. Время выполнения операции – константа.

```
procedure ОБЪЕДИНИТЬ (x, y);
begin
   if (r [x] < r [y]) then p [x] := y
   else if (r [x] > r [y]) then p [y] := x
   else {p [x] := y; r [y] := r [y] + 1}
end;
```

Операция НАЙТИ (x, y), как уже говорилось, выполняется в два прохода. При первом проходе мы идем от узла x к его родителю, потом к родителю его родителя и так далее, пока не достигнем корня у дерева, содержащего узел x. При втором проходе из x в y все встреченные на этом пути узлы делаются непосредственными потомками узла y. Будем называть это «сжатием путей». Очевидно, как и раньше, время выполнения одной такой операции есть $O(\log n)$. Но ниже будет доказано, что время выполнения m таких операций на самом деле меньше, чем $O(m \cdot \log n)$. Заметим, что при выполнении этой операции ранги узлов не изменяются.

```
procedure НАЙТИ (x, y);
begin
   z := x; while (p [x] ≠ x) do x := p [x];
   y := x; while (p [z] ≠ z) do {z1 := z; z := p [z]; p [z1]:= y}
end;
```

2.7. Анализ трудоемкости

Для анализа трудоемкости выполнения операций нам потребуются две функции. Одна из них, $b(n)$, является суперэкспонентой и определяется следующим образом:

$$b(0) = 0, \qquad b(n) = 2^{b(n-1)} \text{ при } n > 0.$$

Вторая – суперлогарифм $\log^* n$, по основанию 2, определяемая соотношением:

$$\log^* n = \max\{k: b(k) \le n\}.$$

Суперлогарифм является в некотором смысле обратной функцией к суперэкспоненте, $\log^*(b(n)) = n$. Значения функций $\log^* n$ и $b(n)$ при нескольких значениях аргументов приведены в следующих таблицах.

n	0	1	2	3	4	5	6
$b(n)$	0	1	2	4	16	65536	2^{65536}

n	0	1	2	3	4	5	…	15	16	…	65535	65536	…	$2^{65536}-1$
$\log^* n$	0	1	2	2	3	3	…	3	4	…	4	5	…	5

Ребро $(x, p(x))$ при текущем состоянии коллекции назовем корневым, если $p(x)$ – корень и $p(x) = x$ (петля); назовем его прикорневым, если $p(x)$ – корень и $p(x) \neq x$, в противном случае – внутренним.

Отметим следующие свойства коллекции на множестве из n элементов. Прикорневое ребро может превратиться во внутреннее, а корневое – в прикорневое только при выполнении операции ОБЪЕДИНИТЬ.

Внутреннее ребро (x, y) при первом же выполнении операции НАЙТИ, «проходящей через него», исчезает, но вместо него появляется прикорневое ребро (x, y'), при этом $r(y') > r(y)$, следовательно, внутреннее ребро «участвует в поиске» не более одного раза.

Если при выполнении очередной операции ОБЪЕДИНИТЬ (x, y) узел y становится родителем узла x, то после ее выполнения справедливо неравенство $r(y) > r(x)$.

При выполнении операции НАЙТИ ранги узлов не изменяются, но узлы могут менять своих родителей, то есть меняется структура леса.

Если перед выполнением операции НАЙТИ узел x был родителем узла y, а после выполнения этой операции родителем узла y стал узел $x' \neq x$, то выполняется неравенство $r(x) < r(x')$. Следовательно, даже после изменения леса в результате выполнения операции НАЙТИ ранги вдоль любого пути от листа к корню будут строго возрастать.

При выполнении операции ОБЪЕДИНИТЬ ранг любого некорневого элемента не изменяется, а ранг корня либо сохраняется, либо увеличивается на 1.

Теорема. Время выполнения последовательности операций, состоящей из n операций СОЗДАТЬ, $u \leq n - 1$ операций ОБЪЕДИНИТЬ и f операций НАЙТИ, при использовании рангов и сжатия путей является величиной $O((f + n) \log^* u)$.

Доказательство. Пусть $s_1, s_2, \ldots, s_m$ – все операции вида СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ, объявленные в формулировке теоремы и выписанные в порядке их следования, $m = n + u + f$. Очевидно, суммарная трудоемкость всех операций СОЗДАТЬ есть $O(n)$, суммарная трудоемкость всех операций ОБЪЕДИНИТЬ есть $O(u)$. Остается оценить суммарную трудоемкость операций НАЙТИ.

Через $r_t(x)$ обозначим ранг узла x, который получится после выполнения операции s_t, а $p_t(x)$ – родитель узла x, получающийся после выполне-

ния этой операции.

Определим множество $G_k(t) = \{x\colon \log^* r_t(x) = k\} = \{x\colon b(k) \le r_t(x) < b(k+1)\}$. Для краткости будем обозначать $\log^* r_t(x) = i_t(x)$.

Поскольку ранг узла может увеличиваться лишь при выполнении операции ОБЪЕДИНИТЬ, причем не более чем на 1, то после u таких операций ранг никакого узла не может стать больше u, следовательно, максимальный индекс k, при котором $G_k(t)$ может быть непустым, равен $\log^* u$.

Оценим теперь суммарное время, требуемое для выполнения f операций НАЙТИ; очевидно, оно пропорционально числу ребер, ведущих от сыновей к отцам и встречающихся при выполнении всех таких операций. Для оценки времени, затрачиваемого на реализацию этих операций, применим бухгалтерский прием. Отнесем расход времени на прохождение очередного ребра (x, y) от узла x к его родителю y при выполнении операции s_{t+1} типа НАЙТИ на одну из трех разных статей расходов: «корневую», «транзитную» и «местную» в зависимости от следующих условий.

Если $i_t(x) \ne i_t(y)$ и y в рассматриваемый момент не является корнем, то расходы относим на статью T транзитных расходов. Если $i_t(x) = i_t(y) = k$ и y не является корнем, то на статью M_k местных расходов в k-м диапазоне, если же y – корень, то на статью K корневых расходов.

Сумму местных расходов во всех диапазонах обозначим через

$$M = \sum_{k=0}^{\log^* u} M_k .$$

Тогда имеем $K = O(f)$, так как при каждом выполнении операции НАЙТИ проходится одно корневое и, возможно, одно прикорневое ребро.

Для транзитных переходов имеем $T = O(f \cdot \log^* u)$, так как при каждом выполнении операции НАЙТИ происходит не более $\log^* u$ переходов из одного диапазона в другой.

Для оценки величины M введем потенциал $c_t(x) = r_t(p_t(x))$ узла x после выполнения операции s_t. Если к узлу x еще не применялась операция СОЗДАТЬ, то $c_t(x) = 0$.

Потенциалом группы $G_k(t)$ при текущем состоянии коллекции назовем величину

$$C_k(t) = \sum_{x \in G_k(t)} c_t(x).$$

Очевидно, в любой момент времени справедливы неравенства

$$C_k(t) \le |G_k(t)|\, b(k+1). \qquad (1)$$

Покажем, что для любого узла x при любом $t = 1, 2, 3, \ldots, m$ выполняется неравенство $c_t(x) \geq c_{t-1}(x)$.

Действительно, если s_t – операция СОЗДАТЬ (x) то $c_{t-1}(x) = c_t(x) = 0$, то есть потенциал узла x, так же как, очевидно, и всех остальных, не изменяется.

Пусть s_t – операция ОБЪЕДИНИТЬ (x, y). В случае $r_{t-1}(x) < r_{t-1}(y)$ имеем $c_t(x) = r_t(p(x)) = r_t(y) = r_{t-1}(y) > r_{t-1}(x) = r_{t-1}(p(x)) = c_{t-1}(x)$ и $c_t(y) = c_{t-1}(y)$. Случай $r_{t-1}(x) > r_{t-1}(y)$ аналогичен. В случае $r_{t-1}(x) = r_{t-1}(y)$ имеем $c_t(y) = r_t(p(y)) = r_t(y) = r_{t-1}(y) + 1 > r_{t-1}(y) = r_{t-1}(p(y)) = c_{t-1}(y)$, $c_t(x) = r_t(p(x)) = r_t(y) = r_{t-1}(y)+1 > r_{t-1}(y) = r_{t-1}(x) = r_{t-1}(p(x)) = c_{t-1}(x)$.

Пусть теперь s_t является операцией НАЙТИ, проходящей через узел x, и при этом $p_{t-1}(x)$ не корень (x получает нового родителя $p_t(x)$). Тогда имеем $c_t(x) = r_t(p_t(x)) = r_{t-1}(p_t(x)) > r_{t-1}(p_{t-1}(x)) = c_{t-1}(x)$. В этом случае совершен переход по внутреннему ребру (местный или транзитный).

Итак, для любого узла x величина $c(x)$ при выполнении операции вида СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ не может уменьшиться и, следовательно,

$$C_k(t) \geq C_k(t-1).$$

При этом, если s_t – операция НАЙТИ, то у $M_k(t)$ вершин при ее выполнении потенциал увеличится, по крайней мере, на 1, следовательно, число $M_k(t)$ местных переходов в группе $G_k(t)$ при ее выполнении удовлетворяет неравенству

$$M_k(t) \leq C_k(t) - C_k(t-1).$$

Суммируя это неравенство по всем t и учитывая неравенства (1), получаем, что число M_k местных переходов при всех операциях НАЙТИ удовлетворяет неравенству

$$M_k \leq C_k(m) - C_k(0) \leq C_k(m) \leq |G_k|\, b(k+1).$$

Заметим далее, что утверждение леммы 2, которая гарантирует, что количество узлов ранга r не более $n/2^r$, остается верным и при использовании сжатия путей, так как при выполнении операции НАЙТИ ранги элементов не меняются. Следовательно, справедливы соотношения

$$|G_k| = \sum_{r=b(k)}^{b(k+1)-1} \frac{n}{2^r} < \frac{n}{2^{b(k)}}\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right) = \frac{2n}{2^{b(k)}} = \frac{2n}{b(k+1)}.$$

Отсюда

$$M_k \le \frac{2n}{b(k+1)} b(k+1) = 2n,$$

$$M = \sum_{k=0}^{\log^* u} M_k = O(n \cdot \log^* u).$$

Итак, суммарная трудоемкость выполнения f операций НАЙТИ равна

$$K + T + M = O(f + f \cdot \log^* u + n \log^* u) = O((f + n) \log^* u).$$

Учитывая теперь оценки трудоемкости операций СОЗДАТЬ и ОБЪЕДИНИТЬ получаем утверждение теоремы.

Замечание. Р.Е. Тарьян доказал, что время выполнения последовательности, состоящей из u операций ОБЪЕДИНИТЬ с перемешанными с ними f операциями НАЙТИ, где $u \le n - 1$, $u + f = m$, является величиной $O(m\,\alpha(m, n))$. Также он показал, что эта оценка не может быть улучшена, а значит, алгоритм может потребовать для своего выполнения $\Omega(m\alpha(m,n))$ времени.

Здесь $\alpha(f, n) = \min\{i \ge 1: A(i, \lfloor f/n \rfloor) > \log n\}$, где $A(i, j)$ – функция Аккермана, задаваемая равенствами:

$$A(1, j) = 2^j \text{ при } j \ge 1,$$

$$A(i, 1) = A(i - 1, 2) \text{ при } i \ge 2,$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \text{ при } i, j \ge 2.$$

Сводные данные о сложности операций с разделенными множествами

Реализация с помощью массива

СОЗДАТЬ (x)	$O(1)$
ОБЪЕДИНИТЬ (x, y)	$O(n)$
НАЙТИ (x, y)	$O(1)$
m операций	$O(mn)$

Реализация с помощью древовидной структуры

СОЗДАТЬ (x)	$O(1)$
ОБЪЕДИНИТЬ (x, y)	$O(1)$
НАЙТИ (x, y)	$O(n)$
m операций	$O(mn)$

Реализация с помощью древовидной структуры
с использованием рангов вершин

СОЗДАТЬ (x)	O (1)
ОБЪЕДИНИТЬ (x, y)	O(1)
НАЙТИ (x, y)	O(log n)
m операций	O(m log n)

Реализация с помощью древовидной структуры
с использованием рангов и сжатия путей

СОЗДАТЬ (x)	O(1)
ОБЪЕДИНИТЬ (x, y)	O(1)
НАЙТИ (x, y)	O(log n)
m операций	O(n log*(u + 1)) O(m α (m, n))

Глава 3. ПРИОРИТЕТНЫЕ ОЧЕРЕДИ

3.1. Основные определения

Приоритетная очередь – это абстрактный тип данных, предназначенный для представления взвешенных множеств. Множество называется взвешенным, если каждому его элементу однозначно соответствует число, называемое ключом или весом. Основными операциями над приоритетной очередью являются следующие операции:

- **ВСТАВИТЬ** в множество новый элемент со своим ключом.
- **НАЙТИ** в множестве элемент с минимальным ключом. Если элементов с минимальным ключом несколько, то находится один из них. Найденный элемент не удаляется из множества.
- **УДАЛИТЬ** из множества элемент с минимальным ключом. Если элементов с минимальным ключом несколько, то удаляется один из них.

Дополнительные операции над приоритетными очередями:

- **ОБЪЕДИНИТЬ** два множества в одно.
- **УМЕНЬШИТЬ** ключ указанного элемента множества на заданное положительное число.

Приоритетная очередь естественным образом используется в таких задачах, как сортировка элементов массива, отыскание во взвешенном неориентированном графе минимального остовного дерева, отыскание кратчайших путей от заданной вершины взвешенного графа до его остальных вершин, и многих других.

Приоритетную очередь можно представить с помощью массива или списка элементов, но такие реализации неэффективны по времени выполнения основных операций. Так, например, поиск элемента с минимальным ключом в неупорядоченном массиве или списке требует последовательногого просмотра всех его элементов. Если поддерживать упорядоченность массива или списка по ключу, то «неудобной» окажется операция вставки нового элемента.

Чаще всего приоритетная очередь представляется с помощью корневого дерева или набора корневых деревьев с определенными свойствами. При этом узлам дерева ставятся во взаимно однозначное соответствие элементы рассматриваемого множества.

Соответствие между узлами дерева и элементами множества называется кучеобразным, если для каждого узла i соблюдается условие:

> Ключ элемента, приписанного узлу i, не превосходит ключей элементов, приписанных его потомкам.

Такие представления взвешенных множеств называются *кучами*. Вид дерева и способ его представления в памяти компьютера подбирается в зависимости от тех операций, которые предполагается выполнять над множеством, и от того, насколько эти операции сказываются на суммарной трудоемкости алгоритма.

3.2. Представление приоритетной очереди с помощью *d*-кучи

Представление приоритетной очереди с помощью d-кучи основано на использовании так называемых завершенных d-арных деревьев ($d \geq 2$).

Завершенное d-арное дерево – это корневое дерево со следующими свойствами:

1. Каждый внутренний узел (то есть узел, не являющийся листом дерева), за исключением, быть может, только одного, имеет ровно d потомков. Один узел-исключение может иметь от 1 до $d - 1$ потомков.

2. Если k – глубина дерева, то для любого $i = 1, ..., k - 1$ такое дерево имеет ровно d^i узлов глубины i.

3. Количество узлов глубины k в дереве глубины k может варьироваться от 1 до d^k. Это свойство является следствием первых двух.

Узлы завершенного d-арного дерева принято нумеровать следующим образом: корень получает номер 0, потомки узла с номером i получают номера: $i \cdot d + 1$, $i \cdot d + 2$, ..., $i \cdot d + d$. Такая нумерация удобна тем, что позволяет разместить узлы дерева в массиве в порядке возрастания их номеров, при этом позиции потомков любого узла в массиве легко вычисляются по позиции самого узла. Так же легко по позиции узла вычислить позицию его родителя. Так, для узла, расположенного в позиции i, родительский узел располагается в позиции $(i - 1)$ div d, где div – операция деления нацело.

В изображении завершенного d-арного дерева узлы одинаковой глубины удобно располагать на одном уровне, при этом потомки одного узла располагаются слева направо в порядке объявленных номеров. При таком рисовании нижний уровень заполняется, возможно, не полностью.

Отметим некоторые простые утверждения о завершенных d-арных де-

ревьях, которые будут полезны при анализе трудоемкости основных операций.

Утверждение 1. *Длина h пути из корня завершенного d-арного дерева с* $n > 1$ *узлами в любой лист удовлетворяет неравенствам:* $\log_d n - 1 < h < \log_d n + 1$.

Доказательство. Минимальное количество узлов в d-куче высоты h ($h > 0$), по свойствам 2 и 3 d-арного дерева, очевидно, равно $1 + d + d^2 + \ldots + d^{h-1} + 1$ (последний уровень содержит лишь один узел).

Максимальное количество узлов в такой d-куче равно $1 + d + d^2 + \ldots + d^h$ (последний уровень содержит d^h узлов). Отсюда имеем неравенства:

$$(1 + d + d^2 + \cdots + d^{h-1} + 1) \le n \le (1 + d + d^2 + \cdots + d^h).$$

Суммируя левую и правую части как геометрические прогрессии, получим

$$(d^h - 1) / (d - 1) + 1 \le n \le (d^{h+1} - 1) / (d - 1),$$

и после некоторых очевидных оценок с помощью логарифмирования получаем требуемые неравенства:

$$\log_d n - 1 < h < \log_d n + 1.$$

Утверждение 2. *Количество узлов высоты h не превосходит* n/d^h. (Под высотой узла понимается расстояние от него до наиболее далекого потомка.)

Кучу, содержащую n элементов, будем представлять двумя массивами $a[0 .. n - 1]$ и $key[0 .. n - 1]$, полагая $a[i]$ – имя элемента, приписанного узлу i; $key[i]$ – его ключ. Иногда под $a[i]$ удобно понимать сам элемент исходного множества или ссылку на него. В некоторых прикладных задачах нет необходимости помещать в приоритетную очередь ни сами элементы, ни их имена, в таких случаях при организации кучи используется лишь массив $key[0 .. n - 1]$.

На рис. 1 приведен пример кучи для $d = 3$, $n = 18$. Кружочками изображены узлы дерева, в них записаны элементы массива, представляющие имена элементов кучи.

Пример кучи при $d = 3$, $n = 7$ для приоритетной очереди, содержащей элементы с ключами 1, 2, 2, 2, 3, 4, 5, изображен на рис. 2, где пара чисел в каждом кружочке обозначает номер узла и ключ соответствующего элемента.

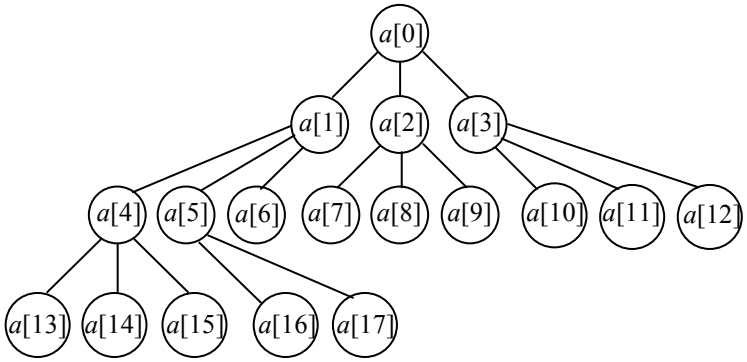


Рис. 1

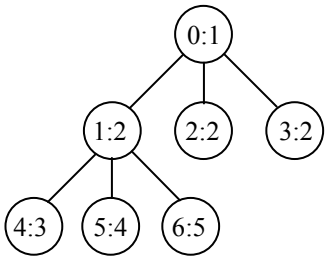


Рис. 2

При реализации основных операций над кучами используются две вспомогательные операции – ВСПЛЫТИЕ и ПОГРУЖЕНИЕ. При реализации этих операций введем еще одну вспомогательную операцию – транспонирование, с помощью которой будем менять местами элементы, расположенные в двух разных узлах дерева. Ее реализация может быть представлена следующим образом:

```
procedure tr(i, j);
begin
        temp0 := a[i]; a[i] := a[j]; a[j] := temp0;
        temp1 := key[i]; key[i] := key[j]; key[j] := temp1;
end;
```

Замечание. Если в кучу помещаются только ключи элементов, то процедура транспонирования модифицируется соответствующим образом.

Операция ВСПЛЫТИЕ. Эта операция может быть применена в тех случаях, когда в некотором узле кучи, например в *i*-м, расположен эле-

мент *x*, нарушающий кучеобразный порядок, то есть ключ элемента в узле *i* меньше ключа элемента *y*, приписанного узлу, являющемуся родителем узла *i*.

Данная операция меняет местами *x* и *y*. Если после этого элемент *x* снова не удовлетворяет условиям кучи, то еще раз проводим аналогичную перестановку. И так до тех пор, пока *x* не встанет на свое место.

Рассмотрим 3-дерево, представленное на рис. 3. В этом дереве кучеобразный порядок нарушает элемент с ключом 14, приписанный узлу с номером 17, так как его родительскому узлу приписан элемент с ключом 31 > 14.

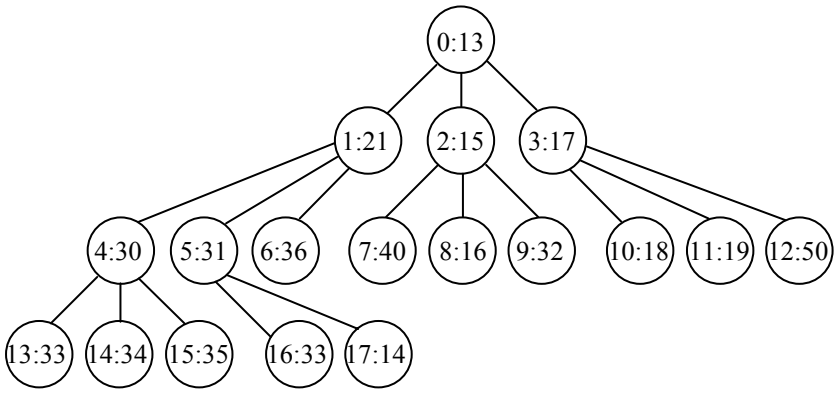


Рис. 3

Применим к узлу 17 операцию ВСПЛЫТИЕ. Элементы с ключами 31 и 14 меняются местами. В результате получается дерево, представленное на рис. 4.

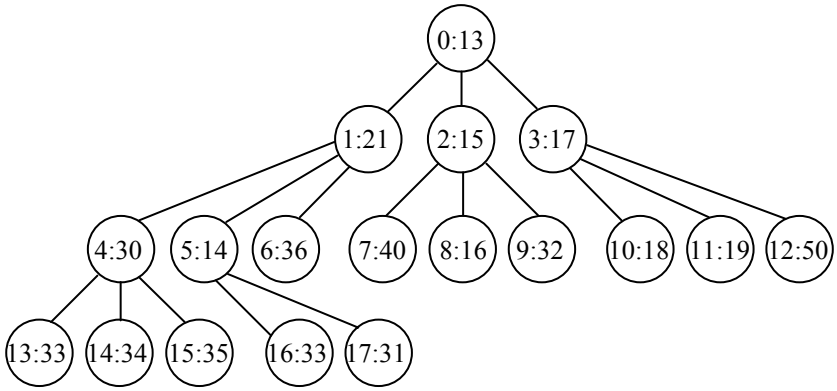


Рис. 4

Теперь нарушен кучеобразный порядок в узле 5 (21 > 14), меняем местами элементы с ключами 21 и 14. В результате получаем кучу, изображенную на рис. 5. Кучеобразный порядок восстановлен, операция ВСПЛЫТИЕ завершена.

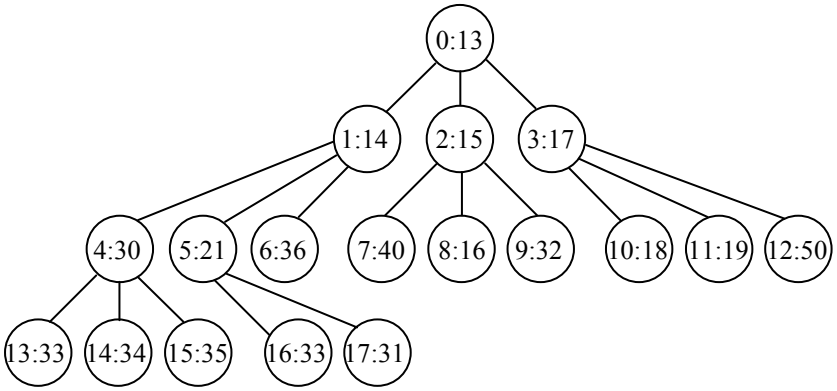


Рис. 5

Вычислительная сложность этой операции пропорциональна числу сравнений элементов и их обменов. Это число, очевидно, не более чем удвоенное число узлов в пути от узла x до корня дерева. Длина такого пути в d-куче с n узлами не превосходит ее высоты, а именно $\log_d n + 1$, по доказанному выше утверждению 1. Значит, время выполнения данной операции – $O(\log_d n)$.

Реализация операции ВСПЛЫТИЕ. Входным параметром этой операции является номер узла, в котором нарушен порядок.

```
procedure ВСПЛЫТИЕ(i);
begin
p := (i – 1) div d;
while (i ≠ 0) and (key[p] > key[i]) do {tr(i, p); i := p; p := (i – 1) div d};
end;
```

Замечания

1. Операцию ВСПЛЫТИЕ можно применять не только к d-куче, но и к другим видам куч.

2. Для более эффективного выполнения операции ВСПЛЫТИЕ можно поступить следующим образом. Запомнить элемент, находящийся в узле i, переместить элемент из его родительского узла $p = (i – 1)$ div d в узел i, затем из узла $(p – 1)$ div d в узел p и так до тех пор, пока не освободится

узел для запомненного элемента. После этого поместить запомненный элемент на освободившееся место. Более точно это можно выразить с помощью операторов:

```
begin
key0:= key[i]; a0:= a[i]; p := (i – 1) div d;
while (i ≠ 0) and (key[p] > key0 do
{a[i]:=a[p]; key[i]:= key[p]; i:= p; p:= (i – 1) div d};
a[i]:= a0; key[i]:= key0
end;
```

Операция ПОГРУЖЕНИЕ. Эта операция также применяется для восстановления свойства кучеобразности. Пусть, например, в *i*-м узле расположен элемент *x*, нарушающий кучеобразный порядок таким образом, что ключ элемента *x* больше ключа элемента *y*, приписанного потомку узла *i*. В этом случае среди непосредственных потомков узла *i* выбирается тот, которому приписан элемент *y* с наименьшим ключом, и элементы *x* и *y* меняются местами. Если после этого элемент *x* снова не удовлетворяет условиям кучи, то еще раз проводим аналогичную перестановку. И так до тех пор, пока *x* не встанет на свое место.

Рассмотрим 3-дерево, представленное на рис. 6. В узле 1 расположен элемент *x* с ключом 31, и этот узел имеет двух потомков с меньшими ключами, а именно 30 и 14. Применим к элементу *x* операцию ПОГРУЖЕНИЕ.

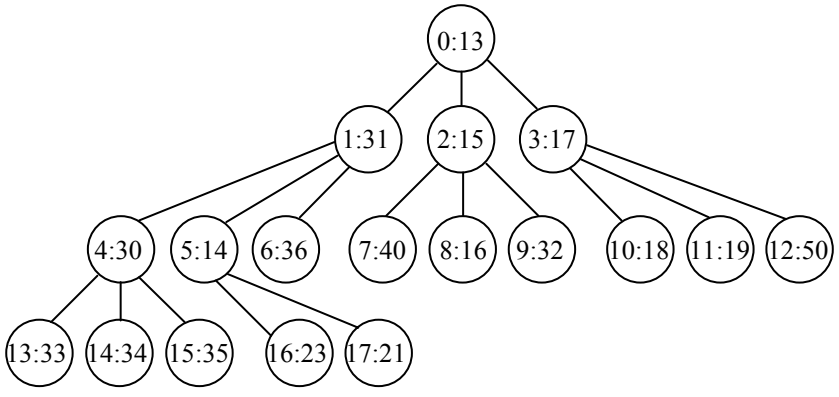


Рис. 6

Среди непосредственных потомков узла 1 находим узел, которому приписан элемент *y* с наименьшим ключом, в нашем случае это узел 5

с ключом 14. Меняем местами элементы *x* и *y*. В результате получается дерево, изображенное на рис. 7.

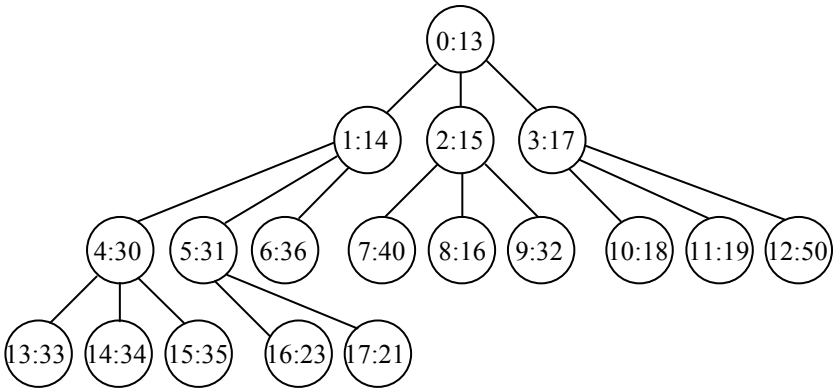


Рис. 7

Теперь элемент *x* снова имеет потомка с меньшим, чем у него, ключом (а точнее, оба его потомка имеют меньшие ключи). Снова находим непосредственного потомка узла, содержащего элемент *x*, в котором находится элемент с наименьшим ключом, и меняем его и *x* местами. Получается дерево, изображенное на рис. 8.

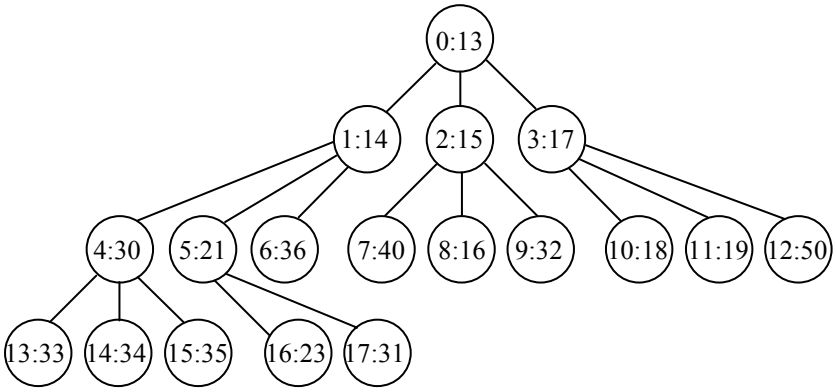


Рис. 8

Теперь *x* находится в узле 17 и не имеет потомков с меньшим, чем у него, ключом (точнее, у него вообще нет потомков). Операция ПОГРУЖЕНИЕ завершена.

Вычислительная сложность этой операции пропорциональна числу сравнений элементов и их обменов. Для каждого узла в пути следования данной операции производится *d* сравнений (при поиске потомка с мини-

мальным ключом) и один обмен. Длина этого пути в d-куче с n узлами не превосходит ее высоты, а именно $\log_d n$, по доказанному выше утверждению 1. Значит время выполнения данной операции – $O(d \cdot \log_d n)$.

Для реализации операции погружения воспользуемся функцией minchild (i), позволяющей для любого узла i находить его непосредственного потомка с минимальным ключом. Если у узла i нет потомков, то minchild (i) = 0.

Реализация операции ПОГРУЖЕНИЕ

```
procedure ПОГРУЖЕНИЕ(i);
begin
c := minchild(i);
while (c ≠ 0) and (key[c] < key[i]) do {tr(i, c); i:= c;
c := minchild (c)}
end;
```

```
function minchild (i);
begin
if i·d + 1 > n then minchild := 0
else
begin first_child := i · d + 1;
last_child := min ((i + 1)·d – 1, n);
min_key := key[first_child];
for i := first_child to last_child do
if key[i] < min_key then {min_key := key[i]; minchild := i}
end
end;
```

Операция ВСТАВКА. Если перед выполнением этой операции куча содержала n узлов (напомним, что они пронумерованы числами от 0 до $n – 1$), то добавляем к дереву $(n + 1)$-й узел (его номер будет n) и приписываем ему элемент с именем *nameX* и ключом *keyX*. Вставка нового элемента производится посредством отведения для него места в n-ых позициях массивов a и *key* соответственно, после чего к добавленному узлу применяется операция ВСПЛЫТИЕ для восстановления кучеобразного порядка.

Вставим в d-кучу, изображенную на рис. 9, новый элемент с ключом 14.

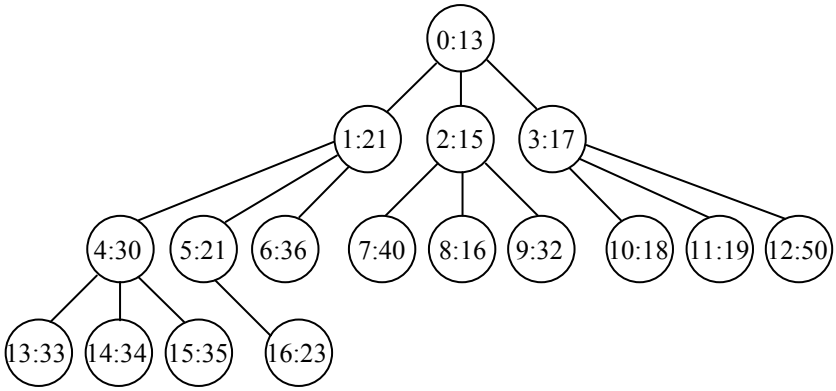


Рис. 9

Сначала добавляем к дереву новый узел с номером 17 и приписываем ему элемент с ключом 14. Получим дерево, представленное на рис. 10.

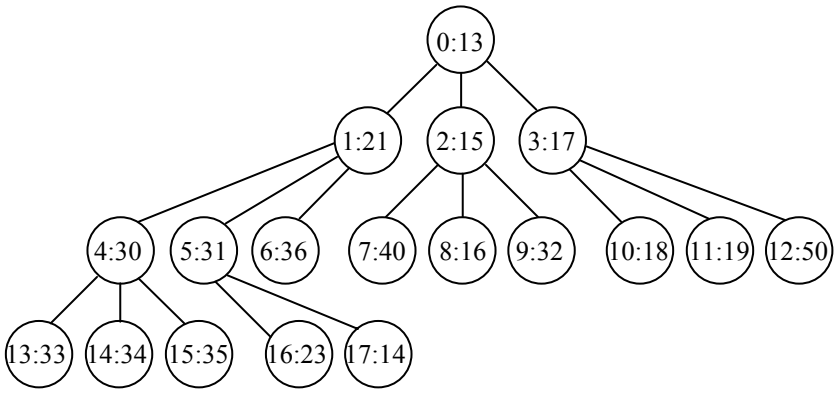


Рис. 10

Затем применяем к узлу 17 операцию ВСПЛЫТИЕ. При описании этой операции использовался именно этот пример (см. рис. 3, 4, 5).

Вычислительная сложность данной операции равна константе плюс вычислительная сложность операции ВСПЛЫТИЕ, то есть $O(\log_d n)$.

Реализация операции ВСТАВКА

```
procedure ВСТАВКА (nameX, keyX);
begin a[n]:= nameX; key[n] := keyX; ВСПЛЫТИЕ (n); n := n + 1 end;
```

Операция УДАЛЕНИЕ. Эта операция используется для удаления элемента, приписанного узлу с заданным номером *i*. Сначала элемент,

приписанный последнему узлу дерева, переносится на место удаляемого элемента, последний узел при этом становится ненужным и поэтому удаляется из дерева. Далее, если узел *i*, в который помещен новый элемент, имеет родителя с большим ключом, то к узлу *i* применяется операция ВСПЛЫТИЕ, в противном случае – ПОГРУЖЕНИЕ.

Таким образом, ориентируясь на худший случай, вычислительную сложность операции УДАЛЕНИЕ оцениваем величиной $O(d \cdot \log_d n)$.

Реализация операции УДАЛЕНИЕ

```
procedure УДАЛЕНИЕ(i);
begin
a[i]:= a[n – 1]; key[i]:= key[n – 1]; n:= n – 1;
if i ≠ 0 and key[i] < key[(i – 1) div d] then ВСПЛЫТИЕ (i)
else ПОГРУЖЕНИЕ (i)
end;
```

Операция УДАЛЕНИЕ_МИНИМУМА. Эта операция предназначена для взятия из кучи элемента с минимальным ключом (он находится в корне дерева) и удаления его из кучи с помощью операции УДАЛЕНИЕ.

Реализация операции УДАЛЕНИЕ_МИНИМУМА

```
procedure УДАЛЕНИЕ_МИНИМУМА (nameX, keyX);
begin nameX:= a[0]; keyX:= key[0]; УДАЛЕНИЕ (0) end;
```

Функция MINKEY. Эта функция предназначена для определения минимального ключа без удаления соответствующего элемента.

Реализация функции MINKEY

```
function MINKEY;
begin MINKEY: = key[0] end;
```

Трудоемкость операции MINKEY, очевидно, равна $O(1)$.

Операция УМЕНЬШЕНИЕ_КЛЮЧА. Предназначена для уменьшения ключа у элемента, приписанного узлу с заданным номером *i*, на заданную величину Δ. Это действие может нарушить кучеобразный порядок лишь таким образом, что уменьшенный ключ элемента в узле *i* станет меньше ключа элемента в родительском узле. Для восстановления порядка в куче используется операция ВСПЛЫТИЕ.

Вычислительная сложность данной операции определяется временем, затрачиваемым на уменьшение ключа (то есть константой), и временем

выполнения операции ВСПЛЫТИЕ (то есть $O(\log_d n)$). В итоге вычислительная сложность операции УМЕНЬШИТЬ_КЛЮЧ равна $O(\log_d n)$.

Реализация операции УМЕНЬШЕНИЕ_КЛЮЧА

```
procedure УМЕНЬШИТЬ_КЛЮЧ (i, delta);
begin key[i] := key[i] – delta; ВСПЛЫТИЕ (i); end;
```

Операция ОКУЧИВАНИЕ. Заметим, что если d-куча создается путем n-кратного применения операции ВСТАВКА, то суммарная трудоемкость ее создания будет равна $O(n \cdot \log_d n)$. Если же все n элементов сначала занимают в произвольном порядке массив $a[0 .. (n-1)]$ и, соответственно, массив $key[0 .. (n-1)]$, то можно превратить их в d-кучу, применяя операцию ПОГРУЖЕНИЕ по очереди к узлам $(n-1)$, $(n-2)$, … , 0.

Такой процесс будем называть окучиванием массива. Для доказательства того, что в результате действительно устанавливается кучеобразный порядок, достаточно заметить, что если поддеревья с корнями в узлах $n-1$, $n-2$, …, $i+1$ упорядочены по правилу кучи, то после применения процедуры ПОГРУЖЕНИЕ к узлу i поддерево с корнем в этом узле также станет упорядоченным по правилу кучи.

Итак, остановимся на следующей реализации.

Реализация операции ОКУЧИВАНИЕ

```
procedure ОКУЧИВАНИЕ;
begin for i := n – 1 downto 0 do ПОГРУЖЕНИЕ (i) end;
```

Утверждение 3. *Вычислительная сложность операции ОКУЧИВАНИЕ равна* $O(n)$.

Доказательство. Заметим, что трудоемкость погружения с высоты h равна $O(h)$, а количество узлов высоты h не превосходит n/d^h. Осталось оценить сумму

$$\sum_{h=1}^{H} h \frac{n}{d^h},$$

где $H = \lceil \log_d n \rceil$, и убедиться, что полученная сумма есть $O(n)$.

Для суммирования можно воспользоваться формулой

$$\sum_{i=1}^{k} \frac{i}{x^i} = \frac{x^{k+1} - (k+1)x + k}{x^k (x-1)^2}.$$

Предоставляем читателю возможность завершить доказательство.

Операция СОЗДАТЬ_СПИСОК_МИНИМАЛЬНЫХ. Эта операция применяется для получения списка элементов, которые имеют ключи, меньшие заданного значения *key*0, и реализуется следующим образом. Если ключ элемента, находящегося в корне, больше, чем *key*0, то это дерево не имеет искомых элементов. В противном случае включаем его в выходной список *S*, а затем применяем ту же процедуру ко всем потомкам узла, включенного в список.

Пусть куча содержит k элементов с ключами, меньшими, чем *key*0. По свойству кучи, они все расположены на ее «верхушке». Данная процедура обходит эту верхушку за время, пропорциональное k, и для каждого из этих k элементов просматривает все его d (или меньше) непосредственных потомков. Получаем, что время выполнения данной процедуры является величиной $O(d \cdot k)$.

Реализация операции СОЗДАТЬ_СПИСОК_МИНИМАЛЬНЫХ

```
procedure СОЗДАТЬ_СПИСОК_МИНИМАЛЬНЫХ(S, key0);
begin
        Инициализируем список S пустым списком;
        Инициализируем стек;
        0 ⇒ стек;
        while стек не пуст do
        begin
        стек ⇒ i;
        if (key[i] < key0) then Добавить a[i] к списку S;
        for j:=d*i + 1 to d*(i + 1) do if j ≤ (n – 1) then j ⇒ стек;
        end
end;
```

Сводные данные о трудоемкости операций с *d*-кучами

ВСПЛЫТИЕ (i)	$O(\log_d n)$
ПОГРУЖЕНИЕ (i)	$O(d \log_d n)$
ВСТАВКА (*nameX*, *keyX*)	$O(\log_d n)$
УДАЛЕНИЕ (i)	$O(d \log_d n)$
УДАЛЕНИЕ_МИН (*nameX*, *keyX*)	$O(d \log_d n)$
MINKEY	$O(1)$
УМЕНЬШЕНИЕ_КЛЮЧА (i, Δ)	$O(\log_d n)$
ОБРАЗОВАТЬ_ОЧЕРЕДЬ	$O(n)$
СПИСОК_МИН (x, h)	$O(d\,k)$

Замечание. Для d-куч «неудобной» является операция слияния куч.

3.3. Применение приоритетных очередей в задаче сортировки

Под задачей сортировки в простейшем случае понимают следующее: дана последовательность ($key[1]$, $key[2]$, ... , $key[n]$) из n элементов некоторого линейно упорядоченного множества, например целых или вещественных чисел, записанных в массив key. Требуется переставить элементы массива так, чтобы после перестановки выполнялись неравенства:

$$key\ [1] \le key\ [2] \le ... \le\ key\ [n].$$

Уточнения этой задачи связаны с теми средствами, с помощью которых предполагается ее решение. Нас интересуют алгоритмы с точки зрения их компьютерной реализации. Оценивая качество различных алгоритмов, обычно интересуются тем, как зависит время счета от длины n сортируемой последовательности и требуется ли для этого дополнительная память, размер которой определяется параметром n. Существенную роль при этом играет метод доступа к элементам памяти. При сортировке во внутренней (оперативной) памяти обычно используется прямой доступ, а во внешней – последовательный.

Бесхитростная сортировка в памяти с прямым доступом. Бесхитростный алгоритм сортировки может заключаться в выполнении следующих операторов

```
for k := 1 to n – 1 do for i := k + 1 to n do
if key[i] < key[k] then tr(i, k);
```

Здесь $tr(i, k)$ – процедура, транспонирующая элементы $key[i]$, $key[k]$. Заметим, что число сравнений

$$"key[i] < key[k]"$$

при реализации такого алгоритма равно $n\ (n – 1) / 2$. В частности, это означает, что время работы алгоритма равно $O(n^2)$.

Сортировка методом «разделяй и властвуй». Предположим, что в нашем распоряжении имеется процедура РАЗДЕЛЯЙ (i, j, k), которая по заданным значениям индексов i, j находит некоторое промежуточное значение k и переставляет элементы сегмента $key[i..j]$ так, чтобы для $s = i$, $i + 1$, ..., $k – 1$ выполнялось неравенство $key[s] \le key[k]$, а для $s = k + 1$, $k + 2$, ..., j – неравенство $key[k] \le key[s]$.

Тогда для сортировки сегмента *key*[*i* .. *j*] может быть использована рекурсивная процедура СОРТИРУЙ.

```
procedure СОРТИРУЙ (i, j);
begin if i = j then exit
                else {РАЗДЕЛЯЙ (i, j, k); СОРТИРУЙ (i, k – 1);
                      СОРТИРУЙ (k + 1, j)}
end;
```

Для сортировки всего исходного массива достаточно выполнить оператор СОРТИРУЙ (1, *n*).

Заметим, что если бы процедура РАЗДЕЛЯЙ работала линейное от длины сегмента время и давала значение *k*, близкое к середине между *i* и *j*, то число обращений к ней приблизительно равнялось бы $\log n$ и сортировка всего массива проходила бы за время порядка $O(n \cdot \log n)$. Однако можно доказать, что при естественной реализации эта оценка справедлива лишь в среднем.

Упражнения

1. Разработайте вариант процедуры СОРТИРУЙ без использования рекурсии. Сколько дополнительной памяти требуется для запоминания границ еще не отсортированных сегментов?
2. Охарактеризуйте работу процедуры СОРТИРУЙ на заранее отсортированном массиве.
3. Напишите на известном вам алгоритмическом языке программу сортировки числового массива с помощью процедуры СОРТИРУЙ и испытайте ее на массивах, сгенерированных с помощью датчика случайных чисел. Составьте таблицу, отражающую время работы вашей программы на массивах разной длины. Каков максимальный размер массива, который можно отсортировать составленной программой на вашем компьютере?

Сортировка «слиянием». Этот метод является разновидностью метода «разделяй и властвуй», впрочем, уместнее было бы назвать его «властвуй и объединяй».

Предположим, что у нас есть процедура СЛИВАЙ (*i*, *j*, *k*), которая два уже отсортированных сегмента *key*[*i*..(*j* – 1)] и *key*[*j*..*k*] преобразует (сливает) в один сегмент *key*[*i*..*k*], делая его полностью отсортированным. Тогда рекурсивная процедура

```
procedure СОРТИРУЙ (i, j);
begin if i = j then exit
            else {m:= (i + j) div 2; СОРТИРУЙ (i, m);
                  СОРТИРУЙ (m + 1, j); СЛИВАЙ (i, m, j)};
end;
```

очевидно, сортирует сегмент *key*[*i* .. *j*], а для сортировки всего исходного массива достаточно выполнить оператор СОРТИРУЙ (1, *n*). Как видим, вопрос балансировки размера сегментов решается здесь просто. Число обращений к процедуре СЛИВАЙ (*i*, *m*, *j*) равно log *n*, а время ее выполнения легко сделать линейным от суммарной длины сливаемых сегментов.

Упражнения

1. Разработайте процедуру СЛИВАЙ и вариант процедуры СОРТИРУЙ без использования рекурсии. Сколько дополнительной памяти требуется для ее реализации?
2. Оцените теоретически время работы алгоритма по методу слияния.
3. Напишите на известном вам алгоритмическом языке программу сортировки числового массива методом слияния и испытайте ее на массивах, сгенерированных с помощью датчика случайных чисел. Составьте таблицу, отражающую время работы вашей программы на массивах разной длины. Каков максимальный размер массива, который можно отсортировать составленной программой на вашем компьютере?

Сортировка с помощью *d*-кучи. Для представления сортируемой последовательности используем структуру *d*-кучи. Сортировку можно провести в два этапа.

1. Окучить сортируемый массив, применяя последовательно операцию ПОГРУЖЕНИЕ по очереди к узлам ($n-1$), ($n-2$), … , 0 в предположении, что сначала все n ключей занимают в произвольном порядке массив *key*[0 .. $n-1$].
2. Осуществить окончательную сортировку следующим образом. Первый (минимальный) элемент кучи меняем местами с последним, уменьшаем размер кучи на 1 (минимальный элемент остается в последней позиции массива key, не являясь уже элементом кучи) и применяем операцию ПОГРУЖЕНИЕ к корню, затем повторяем аналогичные действия, пока размер кучи не станет равным 1.

   Эти два этапа реализуются с помощью процедуры SORT, которая сор-

тирует массив по убыванию ключей.

```
procedure SORT(n);
begin
for i := n – 1 downto 0 do ПОГРУЖЕНИЕ (i);
while n > 1 do {tr (1, n); n:= n – 1; ПОГРУЖЕНИЕ (1)}
end;
```

Заметим, что процедура SORT не требует дополнительной памяти, размер которой зависел бы от длины массива *key*.

Упражнения

Докажите, что оператор

```
for i := n – 1 downto 0 do ПОГРУЖЕНИЕ (i);
```

в процедуре SORT можно заменить на оператор

```
for i:= n div 2 downto 0 do ПОГРУЖЕНИЕ (i).
```

Напишите на известном вам алгоритмическом языке программу сортировки числового массива с помощью процедуры SORT(n) и испытайте ее на массивах, сгенерированных с помощью датчика случайных чисел. Составьте таблицу, отражающую время работы вашей программы на массивах разной длины. Каков максимальный размер массива, который можно отсортировать составленной программой на вашем компьютере?

3.4. Нахождения кратчайших путей в графе

Входные данные:

- Граф G со взвешенными ребрами (под весами можно понимать длины ребер, если речь идет о геометрическом графе, или любые другие числовые характеристики ребер). Пусть $L(i, j)$ – вес ребра (i, j).
- Стартовая вершина s (вершина, от которой вычисляются расстояния до всех остальных вершин).

Выходные данные:

- Массив $dist[1..n]$, ($dist[i]$ – кратчайшее расстояние от вершины s до вершины i).
- Массив $up[1..n]$, ($up[i]$ – предпоследняя вершина в кратчайшем пути из s в i).

Замечание. Приводимый ниже алгоритм Дейкстры корректно решает задачу для графов с неотрицательными весами вершин. Если же в графе

есть ребра с отрицательными весами, но нет циклов с отрицательным суммарным весом, то для решения задачи можно использовать алгоритм Форда, Беллмана.

Алгоритм Дейкстры

1. Заполнить массив *up* [1..*n*] нулями.
2. Каждой вершине *i* приписать в качестве ключа *dist* [*i*] – максимально возможное число (оно должно быть больше, чем длина наибольшего из кратчайших путей в графе; в процессе вычислений это число будет уменьшаться и в итоге заменится на длину кратчайшего пути из вершины *s* в вершину *i*).
3. Организовать приоритетную очередь из вершин графа, взяв в качестве ключей величины *dist* [*i*], *i*= 1, 2, …, *n*.
4. Заменить ключ вершины *s* на 0.
5. Пока очередь не пуста, выполнять операции 6, 7.
6. Выбрать (с удалением) из приоритетной очереди элемент r_0 с минимальным ключом.
7. Для каждой вершины *r*, смежной с r_0, выполнить операции 8, 9.
8. Вычислить величину $delta = dist[r] - (dist[r_0] + L(r_0, r))$.
9. Если $delta > 0$, то уменьшить ключ *dist* [*r*] элемента *r* на величину *delta* и заменить старое значение величины *up* [*r*] на r_0.

Упражнение

Напишите на известном вам алгоритмическом языке реализацию алгоритма Дейкстры с использованием *d*-кучи при различных значениях *d* и испытайте ее на тестовых примерах.

Глава 4. ОБЪЕДИНЯЕМЫЕ ПРИОРИТЕТНЫЕ ОЧЕРЕДИ

4.1. Левосторонние кучи

Левосторонняя куча – это представление приоритетной очереди с помощью так называемого *левостороннего бинарного дерева*. При реализации приоритетных очередей левосторонними кучами предусматривается возможность их объединения.

Бинарным деревом называется корневое дерево, у которого каждый узел имеет не более двух непосредственных потомков. Один из потомков называется левым, другой, если он есть, – правым. Узел называем неполным, если он имеет менее двух непосредственных потомков. В частности, листья дерева являются неполными узлами.

Рангом узла называем увеличенное на 1 расстояние (число ребер) от него до ближайшего неполного потомка.

Ранг узла можно определить также следующим образом. Расширить данное дерево до полного бинарного дерева, добавляя к каждому узлу, имеющему менее двух потомков, в том числе и к листьям исходного дерева, недостающее количество потомков. Затем приписать каждому из листьев полученного расширенного дерева ранг 0, а ранг каждого из остальных узлов определить как минимум из рангов его непосредственных потомков плюс 1. Очевидно, ранги вершин исходного дерева совпадут с рангами соответствующих вершин расширенного дерева.

Левостороннее дерево – это бинарное дерево, для каждого узла которого ранг его левого непосредственного потомка в расширенном дереве не меньше ранга его правого потомка.

Ветвью бинарного дерева называем последовательность его узлов, начинающуюся с корня, заканчивающуюся листом, такую, что каждый следующий узел является непосредственным потомком предыдущего.

Правой ветвью дерева называем ветвь, заканчивающуюся в узле, не имеющем правого потомка, такую, что каждый следующий узел является непосредственным правым потомком предыдущего.

Пример левостороннего дерева (и его расширения) приведен на рис. 1. Ребра исходного дерева изображены жирными линиями, а ребра, добавленные при расширении, – пунктиром. Числа рядом с узлами – их ранги.

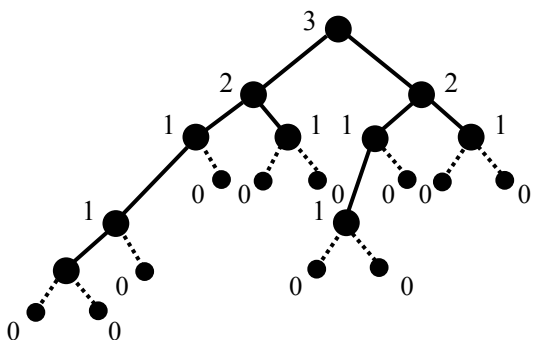


Рис. 1

Свойства левостороннего дерева

1. Правая ветвь из любого узла дерева имеет минимальную длину среди всех ветвей, исходящих из этого узла.

2. Длина правой ветви левостороннего дерева, имеющего n узлов, ограничена величиной $c\lfloor \log_2 n \rfloor$, $c = \text{const}$.

Первое свойство непосредственно следует из определения левостороннего дерева. Для доказательства второго свойства рассмотрим левостороннее дерево T, у которого длина правой ветви равна h. Индукцией по числу h докажем, что число n узлов в таком дереве удовлетворяет неравенству $n \geq 2^h - 1$. Действительно, при $h = 1$ утверждение очевидно. При $h > 1$ левое и правое поддеревья дерева T будут левосторонними, а ранги их корней больше или равны $h - 1$. Следовательно, по предположению индукции число узлов в каждом из них больше или равно $2^{h-1} - 1$, а в дереве T – больше или равно

$$(2^{h-1} - 1) + (2^{h-1} - 1) + 1 = 2^h - 1.$$

Для реализации приоритетной очереди с помощью левосторонней кучи будем использовать узлы вида

$$Node = (element, key, rank, left, right, parent),$$

содержащие следующую информацию:

- *element* – элемент приоритетной очереди или ссылка на него (используется прикладной программой);
- *key* – его ключ (вес);
- *rank* – ранг узла, которому приписан рассматриваемый элемент;
- *left, right* – указатели на левое и правое поддеревья;
- *parent* – указатель на родителя.

Куча представляется указателем на ее корень. Если h − указатель на

корень кучи, то через h будем обозначать и саму кучу.

Заметим, что указатель на родителя используется лишь в операциях *УДАЛИТЬ* и *УМЕНЬШИТЬ_КЛЮЧ* (см. ниже).

Операция СЛИЯНИЕ. Эта операция позволяет слить две левосторонние кучи h_1 и h_2 в одну кучу h. Реализуется она посредством слияния правых путей двух исходных куч в один правый путь, упорядоченный по правилам кучи, а левые поддеревья узлов сливаемых правых путей остаются левыми поддеревьями соответствующих узлов в результирующем пути. В полученной куче необходимо восстановить свойство левизны каждого узла. Это свойство может быть нарушено только у узлов правого пути полученной кучи, так как левые поддеревья с корнями в узлах правых путей исходных куч не изменились. Восстанавливается свойство левизны при помощи прохода правого пути снизу вверх (от листа к корню) с попутным транспонированием в случае необходимости левых и правых поддеревьев и вычислением новых рангов проходимых узлов.

Рассмотрим процесс слияния двух левосторонних куч h_1 и h_2, изображенных на рис. 2.

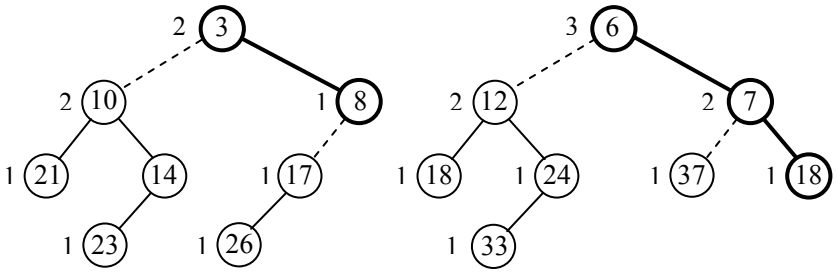


Рис. 2

Числа внутри кружочков являются ключами элементов, приписанных к соответствующим узлам. Правые ветви куч показаны жирными линиями. Числа рядом с узлами – их ранги.

После объединения правых путей получим дерево, изображенное на рис. 3. Оно не является левосторонним. В скобках указаны ранги узлов, какими они были в исходных кучах до слияния.

Восстановление свойства левизны кучи начинаем с последнего узла правой ветви. Это узел с ключом 18. Очевидно, он должен иметь ранг 1, совпадающий с его старым значением и поэтому не требующий обновления.

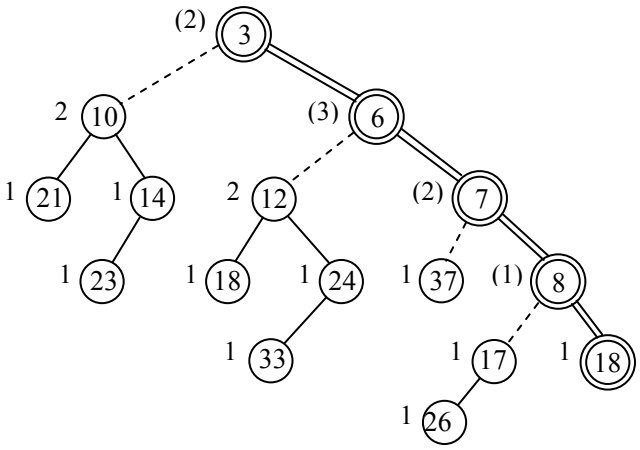


Рис. 3

Следующий по направлению к корню узел правой ветви имеет ключ 8, ранг его левого сына не меньше ранга правого сына, следовательно, условие левизны выполняется и поэтому транспонирования его левого и правого поддеревьев не требуется. Однако ранг этого узла необходимо обновить, так как его старое значение 1 не совпадает с увеличенным на 1 минимальным из рангов его потомков, то есть с числом 2. Обновив ранг, получим кучу, изображенную на рис. 4.

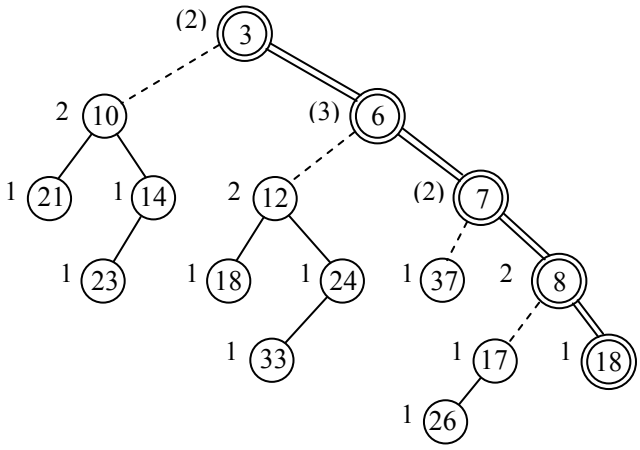


Рис. 4

Теперь рассмотрим узел с ключом 7. Он имеет левого сына с ключом 37 и рангом 1 и правого сына с ключом 8 и рангом 2. Для восстановления свойства левизны в этом узле необходимо поменять местами его левое и правое поддеревья и обновить ранг. Его новым значением будет минимум

из рангов его потомков (это ранг нового правого сына) плюс 1, то есть 2. В результате получаем дерево, изображенное на рис. 5.

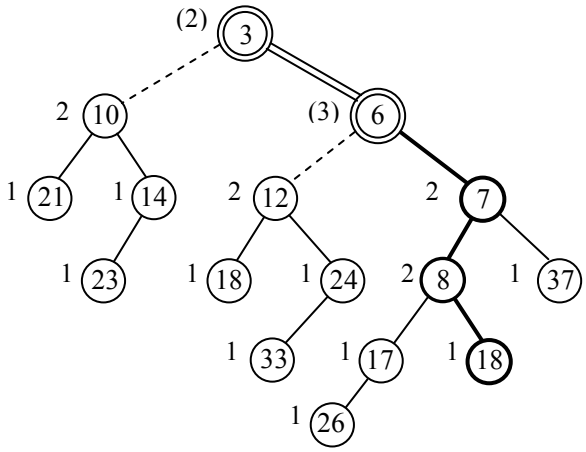


Рис. 5

Далее рассматриваем узел с ключом 6. Оба его сына имеют одинаковый ранг 2, следовательно, менять их местами не требуется. Вычислим лишь новое значение ранга: оно равно минимальному из рангов его детей (рангу правого сына) плюс 1, то есть 3. Получаем дерево, изображенное на рис. 6.

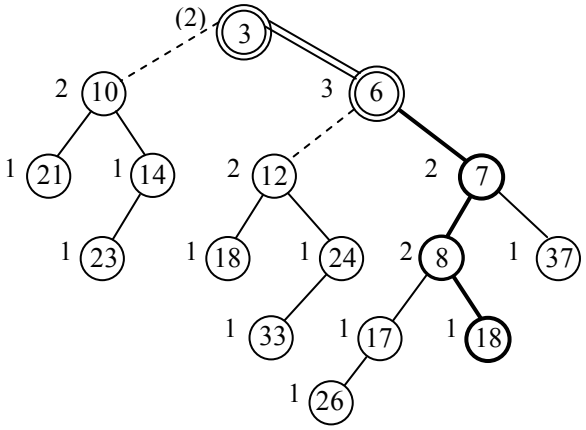


Рис. 6

Наконец, рассматриваем узел с ключом 3, который является последним в правой ветви, полученной слиянием правых ветвей исходных куч. Его потомков (узлы с ключами 10 и 6) необходимо поменять местами для

восстановления свойства левизны и обновить ранг, который будет теперь равен 3. После выполнения этих операций получим левостороннюю кучу, изображенную на рис. 7. На этом выполнение операции СЛИЯНИЕ заканчивается.

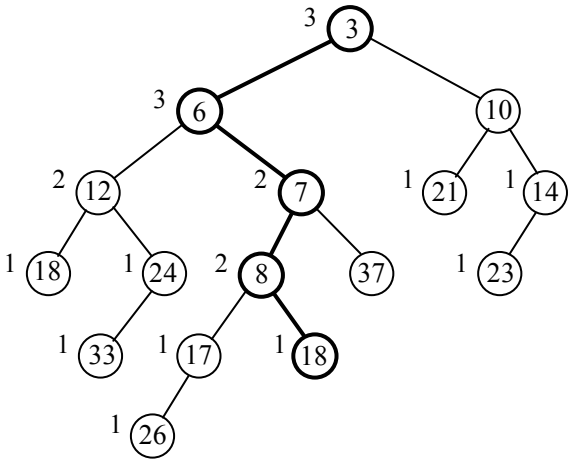


Рис. 7

Очевидно, время выполнения операции СЛИЯНИЕ пропорционально сумме длин правых путей сливаемых куч. По свойству левосторонней кучи оно не превосходит величины $\log n_1 + \log n_2 < \log n + \log n$, где n_1, n_2 – количества узлов в исходных кучах, а $n = n_1 + n_2$ – количество узлов в результирующей куче. Следовательно, вычислительная сложность операции СЛИЯНИЕ равна $O(\log n)$.

Реализация операции СЛИЯНИЕ

```
procedure СЛИЯНИЕ(h1, h2, h);
begin
  if h1 = nil then {h := h2; exit};
  if h2 = nil then {h := h1; exit};
  if h1^.key > h2^.key then {h3 := h1; h1 := h2; h2 := h3;}
  h := h1; СЛИЯНИЕ(h1^. right, h2, h3); h^. right := h3;
  if h^.left^.rank < h^.right^.rank then
   {h3:= h^.left; h^.left := h^.right; h^.right := h3};
  h^.rank := min(h^.right^.rank, h^.left^.rank) + 1;
end;
```

Операция ВСТАВКА. Эта операция позволяет осуществить вставку в кучу h нового элемента x с ключом k. Она производится посредством об-

разования левосторонней кучи, содержащей единственный элемент *x* с ключом *k*, и слияния ее с кучей *h*. Вычислительную сложность данной операции можно оценить так же, как вычислительную сложность операции СЛИЯНИЕ, то есть величиной *O* (log *n*).

Реализация операции ВСТАВКА

```
procedure ВСТАВКА(x, k, h);
begin
  CREATE node h1: [element, key, rank, left, right, parent] = [x, k, 1,
  nil, nil, nil];
  СЛИЯНИЕ(h, h1, h2);  h := h2
end;
```

Операция УДАЛЕНИЕ_МИНИМУМА. Эта операция позволяет из кучи *h* удалить элемент *xMin* с минимальным ключом. Она производится посредством удаления корня кучи *h* (трудоемкость *O* (1)), а затем слияния его левой и правой подкуч (трудоемкость *O* (log *n*)). Таким образом, вычислительная сложность данной операции является величиной *O* (log *n*).

Реализация операции УДАЛЕНИЕ_МИНИНИМУМА

```
procedure УДАЛЕНИЕ_МИНИМУМА (h, xMin);
begin
  xMin := h^.element; СЛИЯНИЕ (h^.left, h^.right, h3);  h := h3
end;
```

Операция МИНИМУМ. Эта операция позволяет взять из кучи *h* элемент с минимальным ключом, не удаляя его из кучи. Поскольку элемент с минимальным ключом находится в корне кучи, то требуется лишь скопировать его в нужное место. Вычислительная сложность данной операции *O* (1).

Реализация операции МИНИМУМ

```
function МИНИМУМ (h);
begin
  МИНИМУМ := h^.element;
end;
```

Операция УДАЛЕНИЕ. Эта операция позволяет удалить из кучи *h* элемент *x*, расположенный в узле, заданном позицией *pos*. Удаление может быть проведено в несколько этапов.

1. Если узел *x* является корнем кучи *h*, то применяется операция

УДАЛЕНИЕ_МИНИМУМА из кучи h. Иначе выполняются следующие действия.

2. От исходной кучи h отрывается подкуча h_2 с корнем в удаляемом узле x. Оставшаяся куча, для которой сохраняем обозначение h, не обязательно является левосторонней.

3. Затем узел x удаляется из кучи h_2, а его левая и правая подкучи сливаются в одну кучу h_2' (время выполнения – $O(\log n)$, как доказано выше).

4. Куча h_2' делается таким же сыном узла p (p – родитель узла x), каким являлся для нее узел x (левым или правым).

5. Наконец, в куче h восстанавливается свойство левизны. Фактически свойству левизны могут не удовлетворять только узлы, находящиеся на пути от p к корню кучи h. Длина этого пути в худшем случае может линейно зависеть от n. Но на самом деле нам нужно проверить только первые не более чем $\lfloor \log (n + 1) \rfloor$ узлов на этом пути (потому что максимальный по длине правый путь имеет максимум $\lfloor \log (n + 1) \rfloor$ узлов).
Таким образом, время выполнения данной операции – $O(\log n)$.

Рассмотрим пример выполнения данной операции. Пусть из кучи h, изображенной на рис. 8, необходимо удалить элемент x с ключом 9.

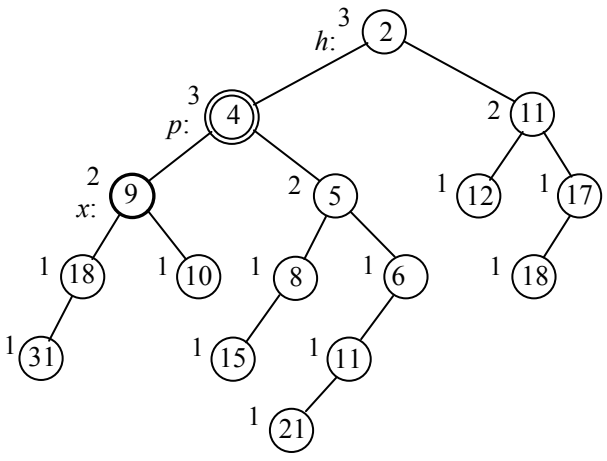


Рис. 8

Сначала отрывается подкуча h_2 с корнем x. От h остаются куча h_1 (нелевосторонняя, так как свойству левизны не удовлетворяет узел p) и левосторонняя куча h_2, см. рис. 9.

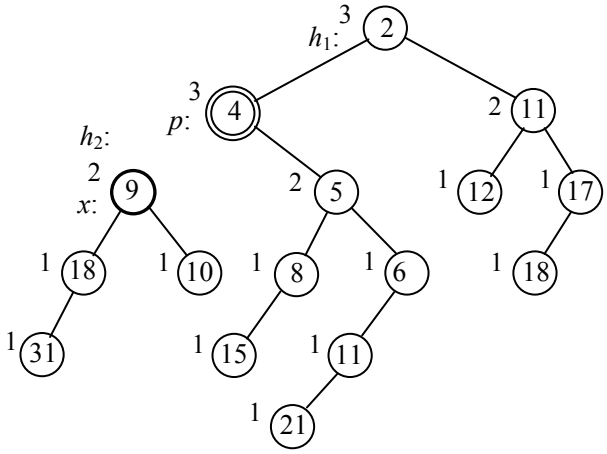


Рис. 9

Затем удаляется узел x, а его левая и правая подкучи h_{2L} и h_{2R} сливаются в одну кучу h'_2 при помощи вышеописанной операции СЛИЯНИЕ; см. рис. 10.

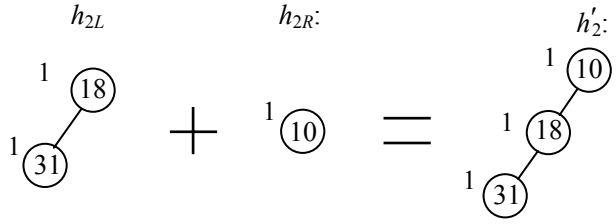


Рис. 10

Поскольку узел x не являлся корнем кучи h, операция еще не завершена. Куча h'_2 становится левым поддеревом узла p, так как узел x был его левым сыном, см. рис. 11.

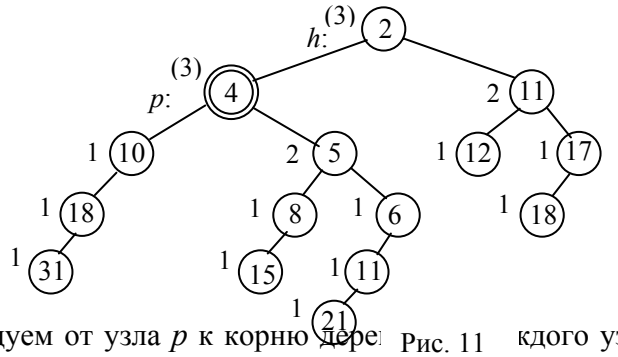


Рис. 11

Следуем от узла p к корню дере кдого узла этого пути вос-

станавливаем свойство левизны и ранг. Сначала проверяем узел *p*: его детей надо поменять местами, так как ранг узла с ключом 10 (он равен 1) меньше ранга узла с ключом 5 (он равен 2). После этого обновляется ранг узла *p*: он равен рангу правого сына плюс 1, то есть 2. Получилось дерево, изображенное на рис. 12.

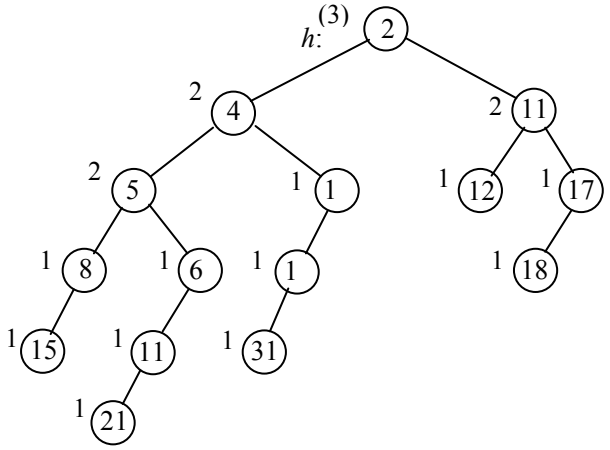


Рис. 12

Следующий узел на пути к корню – это родитель узла *p* с ключом равным 2. Ранги его сыновей равны, значит менять их местами не нужно. Однако его собственный ранг возможно требует обновления, новое значение равно рангу его правого сына плюс 1, т.е. старому: 3. В результате получается дерево, изображенное на рис. 13.

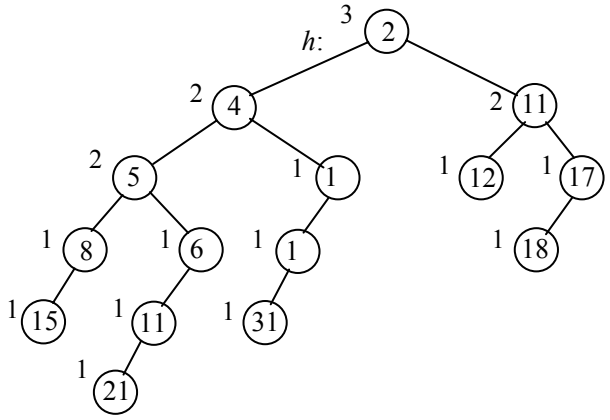


Рис. 13

Поскольку узел с ключом 2 является корнем дерева, операция

УДАЛЕНИЕ завершена.

Реализация операции УДАЛЕНИЕ

```
procedure УДАЛЕНИЕ (h, pos);
begin
 if pos = h then {УДАЛЕНИЕ_МИНИМУМА (h, xMin); exit};
 p := pos^.parent; h2 := pos; УДАЛЕНИЕ_МИНИМУМА (h2, xMin);
 if p^.left = pos then p^.left := h2 else p^.right := h2;
 while p ≠ nil do
 begin
 if p^.left ≠ nil then r1:= p^.left^.rank else r1:= 0;
 if p^.right ≠ nil then r2:= p^.right^.rank else r2:= 0;
 newrank := min (r1, r2) + 1;
 if r1 < r2 then tr (p^.left, p^.right);
 if newrank ≠ p^.parent^.rank then p^.parent^.rank:= newrank
 else exit;
 p:= p^.parent;
 end
end
```

Операция УМЕНЬШИТЬ_КЛЮЧ. Ключ узла x, находящегося в дереве в позиции *pos*, уменьшается на положительное число Δ. Это действие может нарушить кучеобразный порядок лишь таким образом, что уменьшенный ключ узла x будет меньше ключа его родителя. Уменьшение ключа может быть проведено в несколько этапов:

1. От исходной кучи h отрывается подкуча h_2 с корнем в узле x. Оставшаяся куча h не обязательно будет левосторонней.

2. Затем ключ узла x уменьшается на заданное число Δ. Куча h_2 при этом все еще остается левосторонней.

3. В куче h следующим образом восстанавливается свойство левизны. Фактически свойству левизны могут не удовлетворять только узлы, находящиеся на пути от p (p – родитель узла x) до корня h. Длина этого пути в худшем случае линейно зависит от n. Но на самом деле нам нужно проверить только первые не более чем $\lfloor \log (n + 1) \rfloor$ узлов на этом пути.

Наконец, куча h сливается с h_2 за время $O (\log n)$.

Таким образом, время выполнения данной операции – $O (\log n)$.

Рассмотрим пример выполнения рассматриваемой операции. Пусть в куче, изображенной на рис. 14, необходимо уменьшить ключ узла x от

9 до 0.

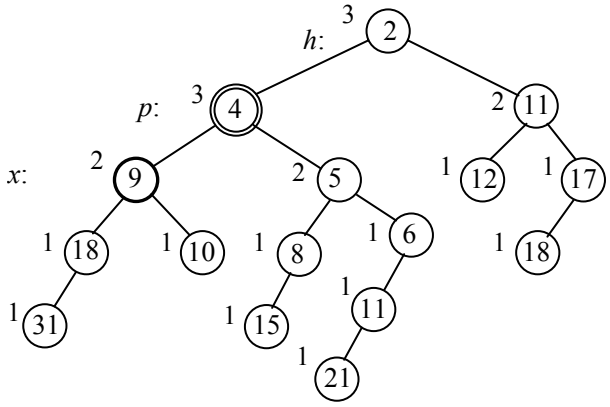


Рис. 14

Делается это следующим образом. От исходной кучи h отрывается подкуча h_2 с корнем в удаляемом узле x. Теперь куча h не является левосторонней, так как в узле p нарушено свойство левизны; см. рис. 15.

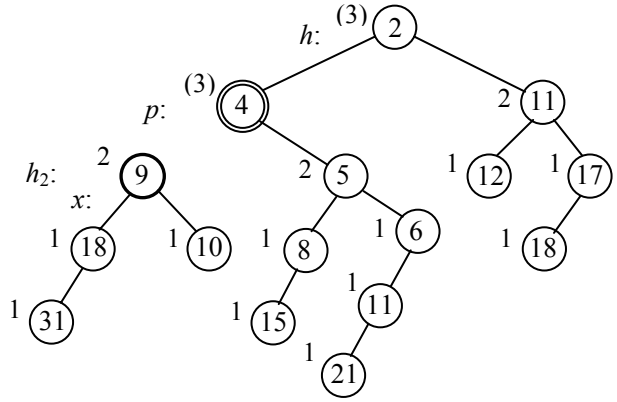


Рис. 15

Ключ узла x уменьшается до 0, куча h_2 при этом все еще остается левосторонней; см. рис. 16.

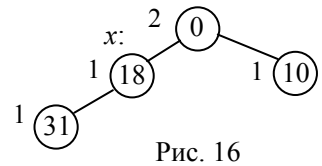


Рис. 16

Следуем от узла p до корня дерева h, для каждого узла этого пути вос-

станавливаем свойство левизны и ранг. Сначала проверяем узел p: его детей надо поменять местами (а фактически – только одного-единственного правого сына сделать левым). После этого вычисляется новый ранг узла p: он равен рангу правого сына плюс 1, то есть 1 (так как правого сына нет). Получилось дерево, изображенное на рис. 17.

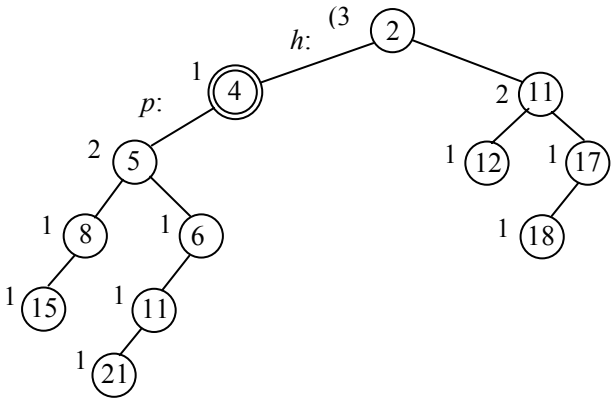


Рис. 17

Следующий узел – это родитель узла p с ключом 2. Его потомков тоже необходимо поменять местами (так как ранг левого сына меньше ранга правого). После этого ранг узла с ключом 2 вычисляется как ранг правого сына плюс 1, то есть 2. Получается куча, представленная на рис. 18:

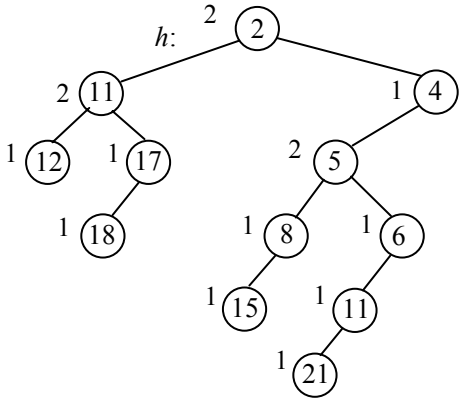


Рис. 18

Наконец, сливаем кучи h и h_2, получая в результате левостороннюю кучу (см. рис. 19).

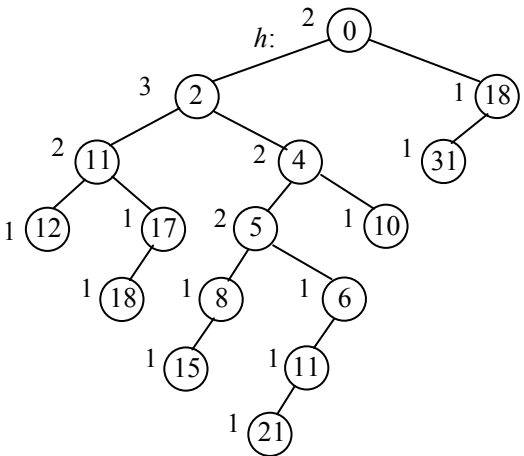


Рис. 19

Реализация операции УМЕНЬШИТЬ_КЛЮЧ

```
procedure УМЕНЬШИТЬ_КЛЮЧ (h, pos, delta);
begin
  pos^.key := pos^.key – delta; if pos = h then  exit;
  p := pos^.parent; h2 := pos;
  if p^.left = pos then p^.left := nil else
  if p^.right = pos then p^.right := nil;
  while p ≠ nil do
  begin
    if p^.left  ≠ nil then r1 := p^.left^.rank else r1 := 0;
    if p^.right  ≠ nil then r2 := p^.right^.rank else r2 := 0;
    newrank := min ( r1, r2 ) + 1;
    if r1 < r2 then tr ( p^.left, p^.right );
    if newrank ≠ p^.parent^.rank then p^.parent^.rank := newrank
       else exit;
    p := p^.parent
  end;
  СЛИЯНИЕ (h, h2, h);
end;
```

Операция ОБРАЗОВАТЬ_ОЧЕРЕДЬ. Из элементов списка S ($|S| = n$) образуется левосторонняя куча h. Способ формирования такой кучи посредством n применений операции ВСТАВИТЬ неэффективен. Читате-

лю предоставляется возможность доказать, что в худшем случае формирование кучи таким способом может потребовать $c \cdot n \cdot \log n$ операций, где $c = \text{const}$.

Более эффективным является следующий способ образования n-элементной левосторонней кучи. Заводится список Q, в который помещаются n одноэлементных куч. Пока длина списка Q больше 1, из его начала извлекаются две кучи, производится их слияние, а полученная куча вставляется в конец списка Q.

Читателю предоставляется возможность доказать, что время выполнения операции ОБРАЗОВАТЬ_ОЧЕРЕДЬ таким способом – $O(n)$.

Реализация операции ОБРАЗОВАТЬ_ОЧЕРЕДЬ

```
procedure ОБРАЗОВАТЬ_ОЧЕРЕДЬ (S, h);
begin
  Создать список Q из одноэлементных куч, содержащих элементы
  списка S;
  while |Q|>1 do
  begin
     Из начала списка Q изъять две кучи h1, h2;
     Создать кучу h, объединяя кучи h1, h2;
     Поместить кучу h в конец списка Q
  end
end;
```

Сводные данные о трудоемкости операций с левосторонними кучами

СЛИТЬ (h_1, h_2, h)	O (log n)
ВСТАВИТЬ (x, h)	O (log n)
УДАЛИТЬ_МИН (h, x)	O (log n)
МИН (x, h)	O (1)
УДАЛИТЬ (x, h)	O (log n)
УМЕНЬШИТЬ_КЛЮЧ (x, Δ, h)	O (log n)
ОБРАЗОВАТЬ_ОЧЕРЕДЬ (q, h)	O (n)

4.2 Ленивая левосторонняя и самоорганизующиеся кучи

Ленивая левосторонняя куча – это представление приоритетной очереди левосторонним деревом, но при этом, в отличие от обычной левосторонней кучи, каждый узел может содержать, а может и не содержать в

себе (быть пустым) элемент приоритетной очереди. Для реализации ленивой левосторонней кучи к каждому узлу добавляется еще одно поле, для хранения признака того, содержит ли данный узел элемент или является пустым. Такие кучи носят название «ленивых» из-за способа выполнения операций **УДАЛИТЬ** и **СЛИТЬ**.

При выполнении операции **УДАЛИТЬ** узел не удаляется, а лишь помечается как пустой. Время «ленивого» выполнения этой операции равно $O(1)$.

Операция **СЛИТЬ** осуществляется следующим образом. Заводится пустой корневой узел, сыновьями которого становятся корневые узлы объединяемых куч. Время «ленивого» выполнения этой операций равно $O(1)$.

При операции **НАЙТИ_ЭЛЕМЕНТ_С_МИНИМАЛЬНЫМ_КЛЮЧОМ** происходит расплата за ленивость, так как эта операция выполняется следующим образом. Сначала делается обход дерева сверху для составления списка, содержащего верхние непустые узлы, чьи родители помечены как пустые. Затем из построенного списка образуется приоритетная очередь с непустыми узлами, после чего берется элемент, содержащийся в корне дерева. Справедливо следующее

Утверждение. Время выполнения операции **НАЙТИ_ЭЛЕМЕНТ_С МИНИМАЛЬНЫМ_КЛЮЧОМ** является величиной

$$O\,(k \max\{1, \log n / (k + 1\}),$$

где k – количество верхних пустых элементов.

При операции **УДАЛИТЬ_ЭЛЕМЕНТ_С_МИНИМАЛЬНЫМ_КЛЮЧОМ** также происходит расплата за ленивость. Она выполняется следующим образом. Сначала, как описано выше, делается обход дерева сверху для нахождения узла с минимальным ключом; найденный узел помечается как пустой. После этого, снова путем обхода дерева сверху, составляется список верхних непустых узлов. И, наконец, поддеревья с корнями в этих узлах сливаются в одну кучу h.

Операция **ВСТАВИТЬ** новый элемент x в кучу h производится посредством слияния кучи h с кучей, содержащей единственный элемент x.

Операция **ОБРАЗОВАТЬ_ОЧЕРЕДЬ** в форме ленивой левосторонней кучи из элементов списка производится как в обычных левосторонних кучах, то есть с неленивыми слияниями.

Операция **УМЕНЬШИТЬ_КЛЮЧ** элемента x на величину Δ выполняется следующим образом. Ключ элемента x уменьшается на Δ, узел, его

содержащий, помечается как пустой, из этого элемента x образуется новая одноэлементная куча, которая сливается с исходной кучей.

Сводные данные о трудоемкости операций с ленивыми левосторонними кучами

УДАЛИТЬ УЗЕЛ	$O(1)$
НАЙТИ ЭЛЕМЕНТ С МИНИМАЛЬНЫМ КЛЮЧОМ	$O(k \max\{1, \log n / (k+1\})$
УДАЛИТЬ ЭЛЕМЕНТ С МИНИМАЛЬНЫМ КЛЮЧОМ	$O(k \max\{1, \log n / (k+1\})$
СЛИТЬ	$O(1)$
ВСТАВИТЬ	$O(1)$
ОБРАЗОВАТЬ ОЧЕРЕДЬ	$O(n)$
УМЕНЬШИТЬ КЛЮЧ	$O(1)$

Самоорганизующаяся куча – это представление приоритетной очереди корневым деревом, операции над которым производятся аналогично операциям над левосторонней кучей, но без использования рангов. Длина правого пути из корня такого дерева в лист может быть произвольной, поэтому время выполнения всех операций в худшем случае есть $O(n)$, где n – число элементов в очереди. Однако среднее время выполнения m произвольных операций есть $O(m \log n)$, то есть время, приходящееся на одну операцию, как не удивительно, является величиной $O(\log n)$. Для их реализации необходимо с каждым узлом дерева хранить элемент, его ключ, указатели на левое и правое поддеревья, то есть узлы представлять записями вида

$$Node = (element, key, left, right).$$

Операция СЛИТЬ кучи h_1 и h_2 в одну кучу h выполняется следующим образом. Правые пути двух исходных куч h_1 и h_2 сливаются в один путь, упорядоченный по правилам кучи, и этот путь становится левым путем результирующей кучи h. Левые поддеревья узлов, попавших в результирующий левый путь, становятся правыми.

Операция ВСТАВИТЬ в кучу h новый элемент x производится посредством слияния кучи h с кучей, содержащей единственный элемент x. Таким образом, время выполнения этой операции равно времени выполнения операции СЛИТЬ.

Операция УДАЛИТЬ_ЭЛЕМЕНТ_С_МИНИМАЛЬНЫМ_КЛЮЧОМ производится посредством удаления корня кучи h и слияния его

левой и правой подкуч. Таким образом, вычислительная сложность этой операции равна вычислительной сложности операции СЛИТЬ.

Операция НАЙТИ_ЭЛЕМЕНТ_С_МИНИМАЛЬНЫМ_КЛЮЧОМ выполняется, очевидно, за время $O(1)$, так как этот элемент находится в корне.

Анализ времени выполнения операции СЛИТЬ. Поскольку время выполнения всех трудоемких операций определяется временем выполнения операции СЛИТЬ, остается проанализировать именно эту операцию. Очевидно, время ее выполнения пропорционально количеству узлов в пра-вых путях исходных куч h_1 и h_2. Длина такого пути в худшем случае мо-жет зависеть линейно от количества узлов в соответствующей куче. Таким образом, время выполнения операции СЛИТЬ есть величина $O(n_1+n_2) = = O(n)$, где n_1, n_2, n – количества узлов в кучах h_1, h_2, h, соответственно.

Нахождение суммарной оценки времени выполнения m операций СЛИТЬ. Введем определение. Узел назовем *тяжелым***,** если количество узлов в его правом поддереве строго больше, чем в левом. Остальные узлы назовем *легкими*.

Определим потенциал коллекции куч как общее количество содержащихся в ней тяжелых узлов. Пусть P_j – потенциал коллекции после выполнения j-й операции.

Утверждение. Время T выполнения m операций СЛИТЬ, примененных к коллекции, состоящей из $(m + 1)$ куч с нулевым потенциалом, яв-ляется величиной $O(m \log n)$, где n – общее количество узлов в коллекции.

Доказательство. Пусть i-я операция заключается в слиянии куч h_1 и h_2 в результирующую кучу h. Пусть перед ее выполнением H_1 и H_2 – количества тяжелых узлов в правых путях куч h_1 и h_2 соответственно, L_1 и L_2 – количества легких узлов в этих путях, Q_1, Q_2 – количества тяжелых узлов в остальных частях куч.

Время выполнения этой операции с точностью до постоянного множителя оценивается сверху величиной $C_i = (H_1 + L_1) + (H_2 + L_2)$.

Подсчитаем изменение ΔP_i потенциала при ее выполнении. Имеем

$$P_{i-1} = H_1 + Q_1 + H_2 + Q_2.$$

По завершении этой операции тяжелые узлы правых путей становятся лег-кими, их количество равно $H_1 + H_2$. Легкие узлы правых путей могут как стать тяжелыми, так и остаться легкими, их будет не более $L_1 + L_2$

штук, а количества тяжелых узлов в остальной части обоих деревьев $Q_1 + Q_2$ не изменились. Следовательно, количество P_i тяжелых узлов после выполнения операции удовлетворяет неравенству

$$P_i \le L_1 + Q_1 + L_2 + Q_2.$$

Таким образом, получаем изменение потенциала

$$\Delta P_i = P_i - P_{i-1} \le (L_1 + Q_1 + L_2 + Q_2) - (H_1 + Q_1 + H_2 + Q_2) =$$

$$= L_1 + L_2 - H_1 - H_2$$

и, следовательно,

$$C_i + \Delta P_i \le (H_1 + L_1 + H_2 + L_2) + (L_1 + L_2 - H_1 - H_2) = 2(L_1 + L_2).$$

Из определения легкого узла следует, что количество L_i легких узлов в куче h_i ($i = 1, 2$) не превосходит логарифма количества n_i узлов в этой куче.

Следовательно,

$$C_i + \Delta P_i \le 2(L_1 + L_2) \le 2(\log n_1 + \log n_2) \le 2\log n_{12} \le 2\log n,$$

где $n_{12} = n_1 + n_2$, а n – общее количество узлов в исходных m кучах.

Суммируя левую и правую части последнего неравенства по $i = 1, 2, \ldots, m$, получаем, что величина T с точностью до постоянного множителя оценивается сверху величиной, пропорциональной $2m \log n$, то есть принадлежит $O(m \log n)$.

Величина $2\log n$ является амортизационной оценкой времени выполнения операции СЛИТЬ, то есть является величиной $O(\log n)$.

Замечание. Вначале коллекция, состоящая из $m + 1$ куч, к которым применяются m операций СЛИТЬ, может иметь произвольное количество узлов, в сумме равное n. Важно, чтобы потенциал каждой из них, следовательно, и суммарный потенциал был равен нулю, то есть кучи не должны первоначально иметь тяжелых узлов.

Это могут быть, например, кучи, целиком являющиеся левыми путями. Крайний случай – это куча высоты h с минимальным количеством узлов, не имеющая тяжелых узлов, а также это могут быть кучи с заполненным последним уровнем узлов. Другой крайний случай – это куча высоты h с максимальным количеством узлов, не имеющая тяжелых узлов. Остальные варианты являются промежуточными для этих двух.

Важным является случай, когда каждая из $m + 1$ начальных куч состоит из единственного узла.

Итак, для всех коллекций таких куч амортизационное время выполнения одной операции СЛИТЬ является величиной $O(\log n)$, где n – общее количество их узлов.

Сводные данные о трудоемкости операций с самоорганизующимися кучами

Операция	Верхняя оценка	Амортизационная оценка
СЛИТЬ	$O(n)$	$O(\log n)$
ВСТАВИТЬ	$O(n)$	$O(\log n)$
УДАЛИТЬ_МИНИМУМ	$O(n)$	$O(\log n)$
НАЙТИ_МИНИМУМ	$O(1)$	$O(1)$

4.3. Биномиальные и фибоначчиевы кучи

Биномиальные кучи. Для каждого $k = 0, 1, 2, \ldots$ ***биномиальное дерево*** B_k определяется следующим образом: B_0 – дерево, состоящее из одного узла высоты 0; далее при $k = 1, 2, \ldots$ дерево B_k высоты k формируется из двух деревьев B_{k-1}, при этом корень одного из них становится потомком корня другого. На рис. 20 изображены биномиальные деревья B_0, B_1, B_2, B_3, B_4.

Биномиальный лес – это набор биномиальных деревьев, в котором любые два дерева имеют разные высоты.

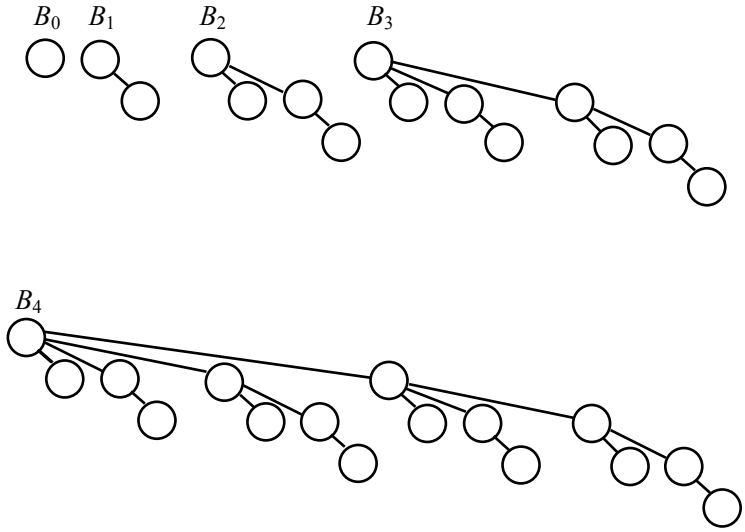


Рис. 20

Свойства биномиальных деревьев

1. Дерево B_k состоит из корня с присоединенными к нему корнями

поддеревьев B_{k-1}, ... , B_1, B_0 в указанном порядке.

2. Дерево B_k имеет высоту k.
3. Дерево B_k имеет ровно 2^k узлов.
4. В дереве B_k на глубине i имеется ровно C_k^i узлов.
5. В дереве B_k корень имеет степень k, остальные узлы имеют меньшую степень.
6. Для каждого натурального числа n существует биномиальный лес, в котором количество узлов равно n.
7. Максимальная степень вершины в биномиальном лесе с n узлами равна $\log_2 n$.
8. Биномиальный лес содержит не более $\lfloor \log_2 n \rfloor$ биномиальных поддеревьев.

Чтобы убедиться в существовании биномиального леса из n узлов, представим n в двоичной системе счисления (разложим по степеням двойки) $n = a_0 2^0 + a_1 2^1 + ... + a_s 2^s$, где $a_k \in \{0, 1\}$. Для каждого $k = 0, 1, 2, \ldots, s$, такого, что $a_k = 1$, в искомый лес включаем дерево B_k.

Биномиальная куча – это набор биномиальных деревьев, узлам которых приписаны элементы взвешенного множества в соответствии с кучеобразным порядком, при котором вес элемента, приписанного узлу, не превосходит весов элементов, приписанных его потомкам.

Поскольку количество детей у узлов варьируется в широких пределах, ссылка на детей осуществляется через левого ребенка, а остальные дети образуют односвязный список. Каждый узел в биномиальной куче представляется набором полей

$$[key, parent, child, sibling, degree],$$

где *key* – ключ (вес) элемента, приписанного узлу, *parent* – родитель узла, *child* – левый ребенок узла, *sibling* – правый брат узла, *degree* – степень узла.

Доступ к куче осуществляется ссылкой на самое левое поддерево. Корни деревьев, из которых составлена куча, оказываются организованными с помощью поля *sibling* в так называемый корневой односвязный список.

Поиск элемента с минимальным ключом. Поскольку искомый элемент находится в корне одного из деревьев кучи, то элемент с минимальным ключом находится просмотром корневого списка за время $O(\log n)$.

Слияние двух очередей. Две очереди H_1 и H_2 объединяются в одну очередь H следующим образом. Последовательно выбираются деревья из

исходных очередей в порядке возрастания их высот и вставляются в результирующую очередь H, вначале пустую.

Если дерево B_i очередной высоты i присутствует лишь в одной из исходных очередей, то перемещаем его в результирующую очередь. Если оно присутствует в одной из исходных очередей и уже есть в результирующей очереди, то объединяем эти деревья в одно B_{i+1}, которое вставляем в H. Если B_i присутствует во всех трех очередях, то сливаем два из них в B_{i+1} и вставляем в H, а третье дерево B_i просто перемещаем в H. Трудоемкость – $O(\log n)$.

Вставка нового элемента. Создается одноэлементная очередь из вставляемого элемента, которая объединяется с исходной очередью. Трудоемкость – $O(\log n)$.

Удаление минимального элемента. Сначала в исходной куче H производится поиск дерева B_k, имеющего корень с минимальным ключом. Найденное дерево удаляется из H, его прикорневые поддеревья B_{k-1}, ... , B_1, B_0 включаются в новую очередь H_1, которая объединяется с исходной очередью H. Трудоемкость – $O(\log n)$.

Уменьшение ключа. Осуществляется с помощью всплытия. Трудоемкость – $O(\log n)$.

Удаление элемента. Уменьшается ключ удаляемого элемента до $-\infty$, применяется всплытие, всплывший элемент удаляется как минимальный. Трудоемкость – $O(\log n)$.

Фибоначчиевы кучи. Название рассматриваемых куч связано с использованием чисел Фибоначчи при анализе трудоемкости выполнения операций. В отличие от биномиальных куч, в которых операции вставки, поиска элемента с минимальным ключом, удаления, уменьшения ключа и слияния выполняются за время $O(\log n)$, в фибоначчиевых кучах они выполняются более эффективно. Операции, не требующие удаления элементов, в этих кучах имеют учетную стоимость $O(1)$. Теоретически фибоначчиевы кучи особенно полезны, если число операций удаления мало по сравнению с остальными операциями. Такая ситуация возникает во многих приложениях.

Например, алгоритм, обрабатывающий граф, может вызывать процедуру уменьшения ключа для каждого ребра графа. Для плотных графов, имеющих много ребер, переход от $O(\log n)$ к $O(1)$ в оценке времени работы этой операции может привести к заметному уменьшению общего времени работы. Наиболее быстрые известные алгоритмы для задач построения минимального остовного дерева или поиска кратчайших путей

из одной вершины используют фибоначчиевы кучи.

К сожалению, скрытые константы в асимптотических оценках трудоемкости велики и использование фибоначчиевых куч редко оказывается целесообразным: обычные двоичные (d-ичные) кучи на практике эффективнее. С практической точки зрения желательно придумать структуру данных с теми же асимптотическими оценками, но с меньшими константами. Такие кучи будут рассмотрены в следующих разделах.

При отсутствии операций уменьшения ключа и удаления элемента фибоначчиевы кучи имели бы ту же структуру, что и биномиальные. Но в общем случае фибоначчиевы деревья обладают большей гибкостью, чем биномиальные. Из них можно удалять некоторые узлы, откладывая перестройку дерева до удобного случая.

Строение фибоначчиевой кучи. Каждая фибоначчиева куча состоит из нескольких деревьев. В отличие от биномиальных деревьев здесь дети любого узла могут записываться в любом порядке. Они связываются в двусторонний циклический список. Каждый узел x этого списка имеет поля $left[x]$ и $right[x]$, указывающие на его соседей в списке. На рис. 21 показано схематическое строение фибоначчиевой кучи.

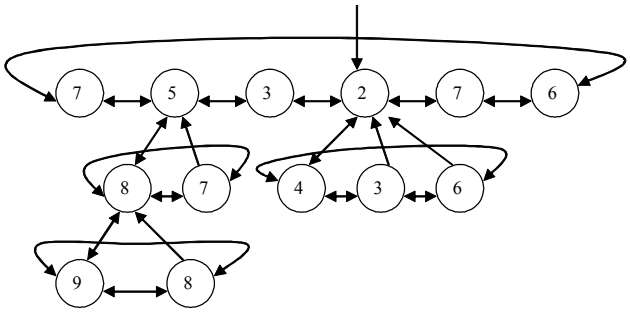


Рис. 21

Двусторонние циклические списки удобны по двум причинам. Во-первых, из такого списка можно удалить любой узел за время $O(1)$. Во-вторых, два таких списка можно соединить в один за время $O(1)$.

Помимо указанной информации, каждый узел имеет поле $degree[x]$, где хранится его степень (число детей), а также поле $mark[x]$. В этом поле хранится булевское значение. Смысл его таков: $mark[x]$ истинно, если узел x потерял ребенка после того, как он в последний раз сделался чьим-либо потомком. Позже будет ясно, как и когда это поле используется.

Корни деревьев, составляющих фибоначчиеву кучу, так же связаны с помощью указателей *left* и *right* в двусторонний циклический список, называемый корневым списком. Таким образом, каждый узел фибоначчиевой кучи представляется записью вида

$$Node = [key, left, right, parent, child, degree, mark].$$

Доступ к куче H производится ссылкой *minH* на узел с минимальным ключом. Кроме того, общее число узлов задается атрибутом $n[H]$.

Потенциал. При анализе учетной стоимости операций используют метод потенциала. Пусть $t(H)$ – число деревьев в корневом списке кучи H, а $m(H)$ – количество помеченных узлов. Потенциал определяется формулой

$$\phi(H) = t(H) + 2m(H).$$

В каждый момент времени в памяти может хранится несколько куч; общий потенциал по определению равен сумме потенциалов всех этих куч. В дальнейшем мы выберем единицу измерения потенциала так, чтобы единичного изменения потенциала хватало для оплаты $O(1)$ операций (формально говоря, мы умножим потенциал на подходящую константу). В начальном состоянии нет ни одной кучи и потенциал равен 0. Как и положено, потенциал всегда неотрицателен.

Максимальная степень. Через $D(n)$ обозначим верхнюю границу для степеней узлов в кучах, которые могут появиться при выполнении операций. Аргументом функции D является общее число всех узлов в куче, обозначаемое через n.

Мы не будем углубляться в анализ трудоемкости операций с фибоначчиевыми кучами, отсылая читателя к соответствующей литературе [7, 19], скажем только, что $D(n) = O(\log n)$ и все операции кроме операции удаления элемента имеют амортизационную трудоемкость $O(1)$, а операция удаления – $O(\log n)$.

Фибоначчиевы кучи ввел М. Фредман и Р. Тарьян [17]. В их статье описаны также приложения фибоначчиевых куч к задачам о кратчайших путях из одной вершины, о кратчайших путях для всех пар вершин, о паросочетаниях с весами и о минимальном покрывающем дереве.

Впоследствии Д. Дрисколл и Р. Тарьян [16] разработали структуру данных, называемую *relaxed heaps*, как замену для фибоначчиевых куч. Есть две разновидности такой структуры данных. Одна из них дает те же оценки учетной стоимости, что и фибоначчиевы кучи. Другая – позволяет

выполнять операцию *DecreaseKey* за время $O(1)$ в худшем случае, а операции *ExtractMin* и *Delete* – за время $O(\log n)$ в худшем случае. Эта структура данных имеет также некоторые преимущества по сравнению с фибоначчиевыми кучами при использовании в параллельных алгоритмах.

4.4. Тонкие кучи

Рассматриваемые здесь тонкие и в следующем разделе толстые кучи предложены М. Фредманом и Х. Капланом как альтернатива фибоначчиевым кучам. Долгое время фибоначчиевы кучи считались рекордными по производительности. Оценки операций над фибоначчиевыми кучами имеют амортизационный характер, а скрытые в них константы велики настолько, что реальный выигрыш во времени работы с ними достигался только на данных «астрономических» размеров. Рассматриваемые здесь тонкие кучи имеют те же асимптотические оценки, что и фибоначчиевы, но гораздо практичнее их. Оценки для толстых куч «хуже» по операции слияния, выполняемой за $O(\log n)$ времени. Достоинством этой структуры является то, что ее оценки рассчитаны на худший случай. Заметим, что на данный момент ни фибоначчиевы, ни толстые, ни тонкие кучи не являются рекордными, так как Г. Бродал предложил новую структуру, которую будем называть кучей Бродала. Кучи Бродала характеризуется такими же, как и фибоначчиевы кучи, оценками операций, но все оценки справедливы для худшего случая. К сожалению, структура, предложенная Г. Бродалом, сложна для реализации. Рассмотрим реализацию приоритетной очереди с помощью тонкой кучи.

Основные определения. Тонкие кучи, как и многие другие кучеобразные структуры, аналогичны биномиальным кучам.

Тонкое дерево T_k ранга k – это дерево, которое может быть получено из биномиального дерева B_k удалением у нескольких внутренних, то есть не являющихся корнем или листом, узлов самого левого сына. Заметим, что у листьев детей нет, а если у корня B_k удалить самого левого сына, то B_k превратится в B_{k-1}. Ранг тонкого дерева равен количеству детей корня.

Для любого узла x в дереве T_k обозначим: $Degree(x)$ – количество детей узла x; $Rank(x)$ – ранг соответствующего узла в биномиальном дереве B_k.

Тонкое дерево T_k удовлетворяет следующим условиям:

1. Для любого узла x либо $Degree(x) = Rank(x)$, в этом случае говорим, что узел x не помечен (полный); либо $Degree(x) = Rank(x) - 1$, в этом случае говорим, что узел x помечен (неполный).
2. Корень не помечен (полный).

3. Для любого узла x ранги его детей от самого правого к самому левому равны соответственно 0, 1, 2, … , $Degree(x) - 1$.
4. Узел x помечен тогда и только тогда, когда его ранг на 2 больше, чем ранг его самого левого сына, или его ранг равен 1 и он не имеет детей.

На рис. 22 приведены примеры тонких деревьев; числа рядом с узлами обозначают их ранги. Вверху изображено биномиальное дерево B_3, внизу – два полученных из B_3 тонких дерева ранга три. Стрелки указывают на помеченные узлы.

Заметим, что биномиальное дерево является тонким деревом, у которого все узлы непомечены.

Тонкий лес – это набор тонких деревьев, ранги которых не обязательно попарно различны.

Нагруженный лес – это лес, узлам которого взаимно однозначно поставлены в соответствие элементы взвешенного множества.

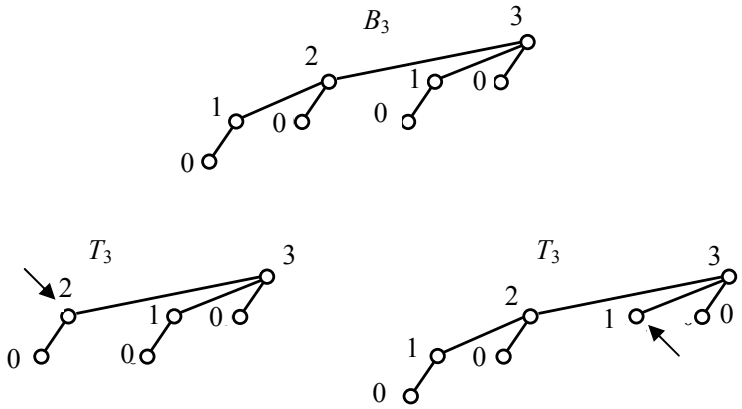


Рис. 22

Тонкая куча – это кучеобразно нагруженный тонкий лес.

Заметим, что в тонкой куче могут встречаться тонкие деревья одинакового ранга, в то время как в биномиальной куче все деревья должны иметь попарно различные ранги.

Утверждение. Для любого натурального числа n существует тонкий лес, который содержит ровно n элементов и состоит из тонких деревьев попарно различных рангов.

Действительно, любой биномиальный лес является тонким, а для биномиального леса рассматриваемое утверждение справедливо.

Пусть $D(n)$ – максимально возможный ранг узла в тонкой куче, содержащей n элементов.

Теорема. В тонкой куче из n элементов $D(n) \le \log_{\Phi}(n)$, где $\Phi = (1+\sqrt{5})/2$ – золотое сечение.

Доказательство. Сначала покажем, что узел ранга k в тонком дереве имеет не менее $F_k \ge \Phi^{k-1}$ потомков, включая самого себя, где F_k – k-е число Фибоначчи, определяемое соотношениями $F_0 = 1$, $F_1 = 1$, $F_k = F_{k-2} + F_{k+1}$ для $k \ge 2$.

Действительно, пусть T_k – минимально возможное число узлов, включая самого себя, в тонком дереве ранга k. По свойствам 1 и 3 тонкого дерева получаем следующие соотношения:

$$T_0 = 1,\ T_1 = 1,\ T_k \ge 1 + \sum_{i=0}^{k-2} T_i \ \text{ для } k \ge 2.$$

Числа Фибоначчи удовлетворяют этому же рекуррентному соотношению, причем неравенство можно заменить равенством. Отсюда по индукции следует, что $T_k \ge F_k$ для любых k. Неравенство $F_k \ge \Phi^{k-1}$ хорошо известно.

Теперь убедимся в том, что максимально возможный ранг $D(n)$ тонкого дерева в тонкой куче, содержащей n элементов, не превосходит числа $\log_{\Phi}(n) + 1$. Действительно, выберем в тонкой куче дерево максимального ранга. Пусть n^* – количество вершин в этом дереве, тогда $n \ge n^* \ge \Phi^{D(n)-1}$.

Отсюда следует, что $D(n) \le \log_{\Phi}(n) + 1$.

Представление тонкой кучи в памяти компьютера. Тонкие кучи формируют из узлов, представленных записями следующего вида:

$$Node = (Key, Left, Right, LChild, Rank),$$

где *Key* – ключ элемента, приписанного узлу; *Left* – указатель на ближайшего левого брата, если такового нет, то на родителя, а если нет и родителя, то указатель заземлен; *Right* – указатель на ближайшего правого брата, если такового нет, то указатель заземлен; *LChild* – указатель на самого левого сына, если такового нет, то указатель заземлен; *Rank* – ранг узла.

Таким образом, узлы-братья связаны в двусвязный список при помощи указателей *Left* и *Right*. У самого левого брата в этом списке указатель *Left* указывает на общего родителя всех узлов в списке. У самого правого брата из списка указатель *Right* заземлен. Корни деревьев в тонкой куче связаны в односвязный циклический список. Этот список будем называть

корневым списком. Корневой список реализуется при помощи поля *Right*. Поле *Left* у каждого узла корневого списка заземлено.

В случае необходимости в описании узла может присутствовать и другая прикладная информация. На рис. 23 приведен пример тонкой кучи.

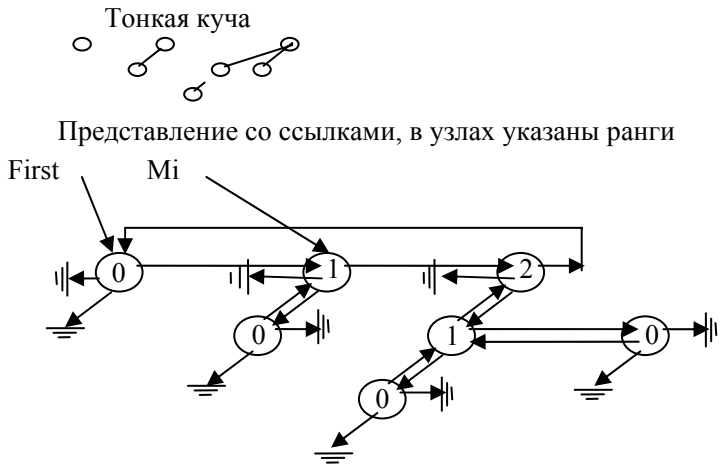


Рис. 23

Заметим, что принадлежность заданного узла корневому списку кучи осуществляется проверкой указателя *Left* на заземленность.

Введем еще одну запись *Heap*, которая будет соответствовать отдельной куче и иметь вид:

$$Heap = (First, Min),$$

где *First* – указатель на начальный элемент корневого списка; *Min* – указатель на элемент корневого списка с минимальным ключом.

Очевидно, узел с минимальным ключом обязательно находится в корневом списке.

Реализация основных операций и оценки трудоемкости. Сосредоточим внимание на амортизационных оценках трудоемкости. Будем получать их методом потенциалов. Потенциалом тонкой кучи будем считать величину $\Phi = n + 2 \cdot m$, где n – количество деревьев в куче, а m – число помеченных вершин. Заметим, что потенциал кучи неотрицателен и в начальный момент равен 0.

Операция MakeHeap. Эта операция создает указатель на новую пустую кучу. Очевидно, фактическая стоимость операции есть $O(1)$, а потенциал созданной кучи равен 0.

Операция FindMin(H). Указатель на узел с минимальным ключом в куче H определяется с помощью указателя Min. Если куча пуста, то результирующий указатель нулевой. Амортизационная оценка совпадает с фактической и равна $O(1)$, потенциал не изменяется.

Операция Insert(i, H). С помощью этой операции осуществляется вставка в кучу H нового элемента с ключом i. При ее реализации создается новое тонкое дерево ранга 0, которое вставляется в корневой список кучи H, разрывая его в произвольном месте. При необходимости перевычисляется ссылка на минимальный элемент.

Операция увеличивает потенциал на 1, так как добавляется одно дерево в корневой список кучи, но это не влияет на амортизационную оценку, которая равна фактической $O(1)$.

Операция Meld(H1, H2). Результатом этой операции является указатель на кучу, полученную слиянием двух куч $H1$ и $H2$. Она осуществляется соединением корневых списков сливаемых куч. При таком способе выполнения операции, как и при реализации вставки элемента в кучу, можем получить в корневом списке результирующей кучи несколько деревьев одинакового ранга. При удобном случае, а именно при удалении минимального элемента, мы освободим корневой список от этой неоднозначности. Оценка совпадает с оценками для всех предыдущих операций. Суммарный потенциал не изменяется.

Операция DeleteMin(H). Эта операция предназначена для удаления узла с минимальным ключом из непустой кучи H. Для ее реализации удаляем минимальный узел из корневого списка кучи H, добавляем список детей удаленного узла в корневой список и повторяем следующий «связывающий шаг».

Находим любые два дерева, корни которых имеют одинаковые ранги, и связываем их, делая корень с большим ключом новым левым потомком корня с меньшим ключом, увеличивая ранг нового полученного тонкого дерева на единицу. При этом следует удалить из корневого списка кучи H корень с большим ключом. Как только не останется деревьев с корнями одинакового ранга, в полученном корневом списке необходимо найти элемент с минимальным ключом.

Рассмотрим теперь, с помощью каких средств реализуется связывающий шаг. Для хранения ссылок на корни деревьев используем временный массив $RankT$, размера $D(n)$. Величина $RankT[i]$ будет указателем на тонкое дерево ранга i. Если найдется еще одно дерево ранга i, то свяжем два дерева ранга i в одно дерево ранга $i + 1$, в i-й ячейке массива $RankT$ уста-

новим нулевой указатель и продолжим связывающую процедуру с вновь полученным деревом ранга $i + 1$.

При включении списка детей необходимо учесть возможность помеченности детей минимального узла. То есть уменьшить их ранг там, где это необходимо. Это требование вытекает из свойства 2 определения тонкого дерева. Очевидно, для проверки помеченности узла требуется $O(1)$ операций.

В результате выполнения связывающих шагов получаем заполненный массив *RankT*. Теперь остается только связать все деревья, находящиеся в этом массиве, в корневой список и найти в этом списке новый минимальный элемент. Очевидно, все это можно выполнить с трудоемкостью $O(D(n))$.

Чтобы оценить амортизационную стоимость операции *DeleteMin*, подсчитаем фактическую стоимость операции и изменение потенциала. Фактическая стоимость складывается из $O(1)$ операций на проверку кучи на пустоту, $O(D(n))$ действий при добавлении детей минимального узла в корневой список и O(количество связывающих шагов) + $O(D(n))$ при выполнении связывающих шагов.

В итоге фактическая стоимость операции удаления минимального элемента есть O(количество связывающих шагов + $D(n)$).

Очевидно, потенциал уменьшился, как минимум, на число связывающих шагов, так как при каждом связывающем шаге количество деревьев в корневом списке уменьшается на единицу. Поскольку амортизационная стоимость равна фактической стоимости плюс изменение потенциала, то амортизационная стоимость равна равна $O(D(n))$.

Операция DecreaseKey (Δ, i, H). При уменьшении ключа у некорневого элемента i в куче H на величину Δ может быть нарушено свойство кучеобразности. Для восстановления этого свойства перемещаем поддерево с корнем в изменяемом элементе в корневой список, но при этом, возможно, оставшееся после переноса дерево может не оказаться тонким.

Покажем как, затратив $O(1)$ амортизированного времени, исправлять его структуру. В процедуре *DecreaseKey* после уменьшения ключа корректируется, если это необходимо, указатель на минимальный элемент кучи. После этого проверяется, не является ли измененный узел x корнем дерева T. Если это действительно так, то процедура завершается, в противном случае переносим поддерево с корнем в узле x в корневой список кучи и запускаем процедуру коррекции оставшегося дерева T'.

Будем различать два вида нарушений свойств тонкого дерева:

1. Братские нарушения – это нарушения третьего правила из определения тонкого дерева.
2. Родительские нарушения – это нарушения первого или второго правила.

Рассмотрим подробнее каждое из двух видов нарушений.

Назовем узел *y* узлом локализации братского нарушения среди детей узла *z,* если ранг узла *y* отличается от ранга его ближайшего правого брата на 2, либо он не имеет правого брата и его ранг равен 1. Пример – на рис. 24.

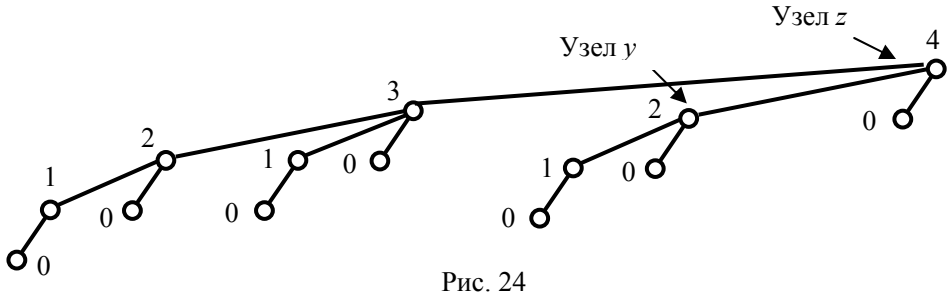


Рис. 24

Назовем узел *y* узлом локализации родительского нарушения, если выполнено одно из трех условий:

1. Ранг узла *y* на три больше, чем ранг его самого левого сына.
2. Ранг узла *y* равен двум, и он не имеет детей.
3. Узел *y* есть помеченный корень дерева.

Пример приведен на рис. 25.

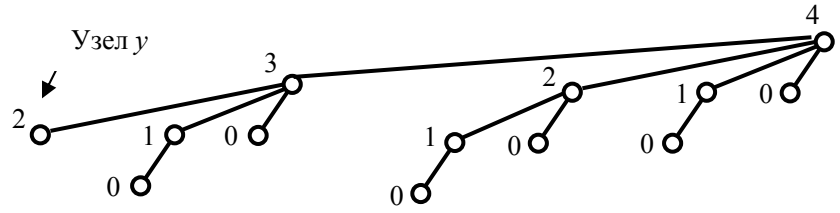


Рис. 25

Рассмотрим теперь, как можно перестроить дерево, чтобы избавиться от братского нарушения либо свести его к родительскому. Пусть узел *y* – это узел локализации братского нарушения. Рассмотрим два возможных варианта.

Узел *y* непомечен, то есть ранг его самого левого сына на единицу

меньше ранга самого узла *y*. Пример – на рис. 26.

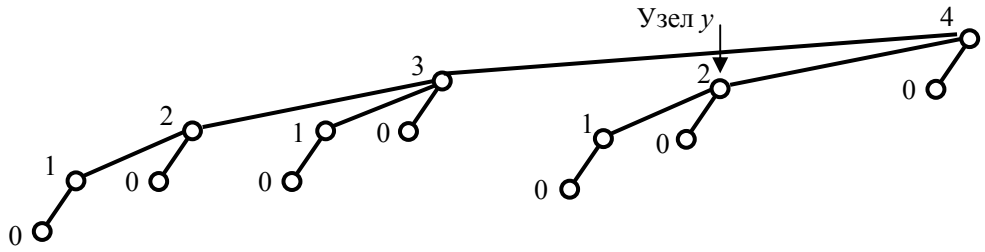


Рис. 26

В данном случае, чтобы исправить братское нарушение, помещаем на место пропущенного в братском списке поддерева поддерево с корнем в самом левом сыне узла *y*. Узел *y* при такой операции становится помеченным, но зато дерево теперь удовлетворяет всем трем свойствам определения тонкого дерева. Очевидно, что это операция заканчивает процедуру исправления дерева.

Узел *y* помечен, тогда уменьшаем ранг узла *y* на единицу. Это не исправит дерева, но зато теперь узлом локализации нарушения будет левый брат узла *y* либо его родитель. В последней ситуации нарушение становится родительским. Пример приведен на рис. 27.

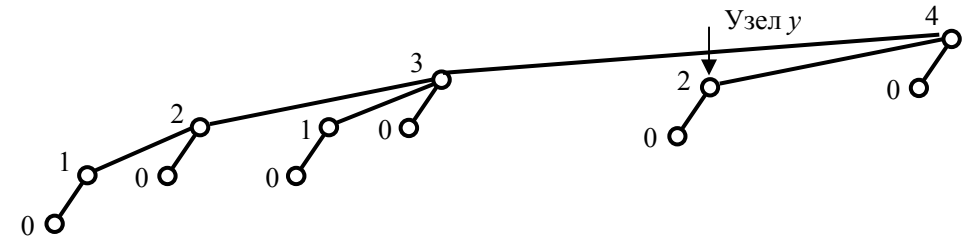


Рис. 27

Таким образом, мы либо исправим структуру дерева, либо рекурсивно придем к узлу локализации родительского нарушения.

Выясним, чтó же делать с родительскими нарушениями. Пусть узел *y* – это узел локализации родительского нарушения, а узел *z* – родитель узла *y*. Тогда предлагается переместить поддерево с корнем в узле *y* в корневой список кучи, делая при этом узел *y* непомеченным. Считаем, что *z* – это не корень дерева. Если узел *z* не был помечен, то, очевидно, процедура исправления дерева закончена. Если он был помечен, то считаем его узлом локализации нового родительского нарушения. При этом, очевид-

но, количество помеченных узлов уменьшится на единицу. Продолжая такого вида рекурсивные шаги, мы либо дойдем до корня дерева, либо исправим его структуру раньше. Если узел z стал корнем, то для того чтобы исправить структуру дерева, необходимо лишь сделать ранг корня на единицу большим ранга его самого левого сына. На этом процедура исправления дерева будет закончена.

Заметим, что каждый промежуточный шаг рекурсии уменьшает число помеченных узлов на единицу и добавляет в корневой список не более одного дерева. Тогда потенциал при каждом шаге рекурсии уменьшается как минимум на единицу. Отсюда и следует обещанная оценка $O(1)$ времени выполнения операции *DecreaseKey*.

Операция Delete(i, H) удаляет элемент i из кучи H следующим образом. Ключ удаляемого элемента i уменьшается до некоторого значения, меньшего минимального, и элемент удаляется как минимальный. Очевидно, трудоемкость этой операции есть $O(D(n))$.

Итак, амортизационная трудоемкость выполнения операций *DeleteMin* и *Delete* на тонкой куче из n элементов равна $O(\log n)$, а для остальных операций, как видели ранее, – $O(1)$.

4.5. Толстые кучи

Рассматриваемое в этом разделе представление приоритетной очереди основано на использовании так называемых избыточных счетчиков, позволяющих за время $O(1)$ инкрементировать любой разряд. Заметим, что использованные здесь счетчики – лишь один из способов реализации толстых куч. На самом деле для их реализации подойдет произвольный d-арный счетчик, при условии, что трудоемкость инкрементирования любого его разряда является константной.

Избыточное представление чисел. Основные определения. Избыточным b-арным представлением неотрицательного целого числа x считаем последовательность $d = d_n, d_{n-1}, \ldots, d_0$, такую, что

$$x = \sum_{i=0}^{n} d_i b^i,$$

где $d_i \in \{0, 1, .., b\}$, $i \in \{0, 1, \ldots, n\}$. Будем называть d_i цифрой, стоящей в i-м разряде. В примерах запятые между цифрами опускаем.

Заметим, что избыточное представление отличается от обычного b-арного представления использованием «лишней» цифры b, что приводит к неоднозначности представления чисел. Например, при $b = 3$ число 3

может быть представлено как 3 и как 10.

В примерах, в которых $b = 10$, «цифру» 10 будем обозначать символом b.

Назовем b-арное избыточное представление числа ***регулярным***, если в нем между любыми двумя цифрами, равными b, найдется цифра, отличная от $b - 1$.

Пример. Пусть $b = 10$, а число x представляется в обычной десятичной системе последовательностью 1100, тогда представления $b9b$ и $bb0$ не являются регулярными b-арными избыточными представлениями числа x, а представления 1100 и $10b0$ регулярны.

Пусть $L(i)$ – номер разряда, отличного от $b - 1$ и ближайшего слева от i-го разряда в регулярном b-арном избыточном представлении d.

Определим $L'(i)$ следующим образом: $L'(i) = L(i)$, если $d_i \in \{b - 1, b - 2\}$ и $d(L(i)) = b$; $L'(i)$ – произвольное число $> i$, если $d_i \in \{b - 1, b - 2\}$ и $d(L(i)) < b - 1$; $L'(i)$ – не определено, если $d_i \notin \{b - 1, b - 2\}$.

Величину $L'(i)$ будем называть прямым указателем.

Пусть $d = d_n, \ldots, d_0$ – b-арное регулярное представление некоторого числа.

Фиксацией цифры b, стоящей в i-м разряде представления d, **(*Fix* (*i*))** назовем операцию, заключающуюся в обнулении цифры d_i и инкрементировании цифры d_{i+1}, при этом если $i = n$, то полагаем $d_{n+1} = 1$. При каждом выполнении операции фиксации будем обновлять значение $L'(i)$. Очевидно, при $b > 2$ операцию Fix (i) можно выполнить с помощью следующих операторов.

```
if d_i = b then {d_i := 0; d_{i+1} := d_{i+1}+1}; if d_{i+1} = b–1 then L′(i) := L′(i+1)
else L′(i) := i+1;
```

Инкрементирование i-й цифры избыточного представления d **(*Inc* (*i*))** можно выполнить с помощью операторов

```
Fix (i); if (d_i = b -1) or (d_i = b–2) then Fix (L′(i)); d_i := d_i + 1; Fix (i);
```

Очевидно, инкрементирование i-го разряда регулярного b-арного избыточного представления числа x производит представление числа $x' = x + b^i$.

Нетрудно доказать, что операции фиксации и инкрементирования, примененные к регулярному избыточному представлению, не нарушают регулярности и корректно вычисляют указатели L с трудоемкостью $O(1)$.

Эта схема может быть расширена для выполнения за константное время декрементирования произвольной цифры добавлением дополнительного цифрового значения $b + 1$. Оставляем детали в качестве упражнения.

Представление толстой кучи. Основные определения. Определяем ***толстое дерево*** F_k ранга k ($k = 0, 1, 2,\ldots$) следующим образом:

- Толстое дерево F_0 ранга ноль состоит из единственного узла.
- Толстое дерево F_k ранга k, для $k \geq 1$, состоит из трех деревьев F_{k-1} ранга $k - 1$, связанных так, что корни двух из них являются самыми левыми потомками корня третьего.

Ранг узла x в толстом дереве определяется как ранг толстого поддерева с корнем в узле x.

На рис. 28 приведены примеры толстых деревьев.

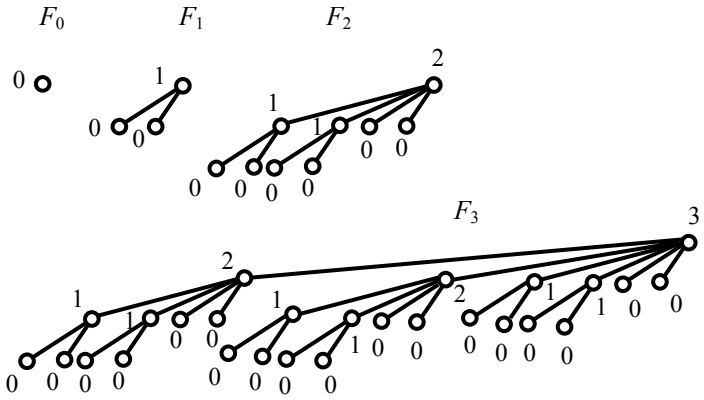


Рис. 28

Свойства толстых деревьев:

1. В толстом дереве ранга k ровно 3^k узлов.
2. Для любого натурального числа n существует лес из толстых деревьев, в котором ровно n узлов. Такой лес можно построить, включив в него столько деревьев ранга i, каково значение i-го разряда представления числа n в троичной системе счисления. Заметим, что для построения такого леса можно использовать и избыточные троичные представления.
3. Толстый лес из n узлов содержит $O(\log n)$ деревьев.

Доказательства этих свойств оставляются читателю в качестве упражнения.

Рассмотрим лес из нескольких толстых деревьев, ранги которых не

обязательно попарно различны и узлам которых взаимно однозначно поставлены в соответствие элементы взвешенного множества. Такой лес будем называть нагруженным. Узел в нагруженном лесе назовем неправильным, если его ключ меньше ключа его родителя. Нагруженный лес назовем почти кучеобразным, если для каждого значения k в нем имеется не более двух неправильных узлов ранга k.

Толстая куча – это почти кучеобразный нагруженный лес.

Представление толстой кучи. Каждый узел толстой кучи будем представлять записью следующего вида:

$$FatNode = (Key, Parent, Left, Right, LChild, Rank),$$

где *Key* – ключ элемента, приписанного узлу дерева; *Parent* – указатель на родителя; *Left* – указатель на ближайшего левого брата; *Right* – указатель на ближайшего правого брата; *LChild* – указатель на самого левого сына; *Rank* – ранг узла. Таким образом, «братья» связаны в двусвязный список при помощи указателей *Left* и *Right*. У самого левого (правого) «брата» в этом списке указатель *Left* (*Right*) заземлен.

На рис. 29 представлено толстое дерево F_2 (внутри узлов указаны их ранги).

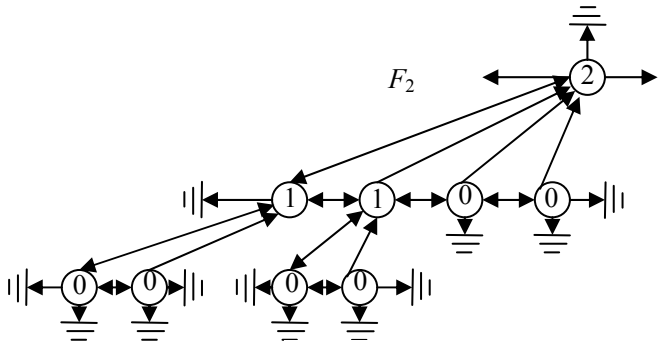


Рис. 29

Для представления толстой кучи введем новую структуру, которую назовем корневым счетчиком, а для того чтобы быстро находить неправильные узлы, введем еще один избыточный счетчик, который назовем счетчиком нарушений. Таким образом, толстую кучу можно представить записью следующего вида

$$FatHeap = (RootCount, CountViolation, MinPointer, MaxRank),$$

где *RootCount* – массив, соответствующий корневому счетчику; *CountViolation* – массив, соответствующий счетчику нарушений; *MinPointer* –

указатель на элемент кучи, имеющий минимальный ключ; *MaxRank* – наибольший ранг среди рангов деревьев, присутствующих в куче.

Корневой счетчик. Корневой счетчик состоит из избыточного троичного представления числа элементов в куче и набора списочных элементов.

Значение *i*-го разряда избыточного корневого представления равно количеству деревьев ранга *i*, присутствующих в куче. При таком определении, избыточного корневого представления число, которое оно представляет равно числу узлов в куче, так как толстое дерево ранга *i* содержит ровно 3^i узлов. Заметим, что состояние избыточного корневого представления определяется неоднозначно. Отсюда следует, что толстая куча с одним и тем же набором элементов может быть представлена различными наборами толстых деревьев. Очевидно, для любой толстой кучи, состоящей из *n* элементов, существует регулярное избыточное представление корневого счетчика.

Списочный элемент, приписанный *i*-му разряду избыточного корневого представления, – это указатель на список деревьев ранга *i*, присутствующих в куче, образованный посредством указателей *Right* корневых узлов связываемых деревьев.

Определение корневого счетчика дает возможность сделать несколько утверждений.

1. Корневой счетчик позволяет иметь доступ к корню любого дерева ранга *i* за время $O(1)$.
2. Вставка толстого дерева ранга *i* соответствует операции инкрементирования *i*-го разряда корневого счетчика.
3. Удаление толстого дерева ранга *i* соответствует операции декрементирования *i*-го разряда корневого счетчика.
4. Операции инкрементирования и декрементирования *i*-го разряда корневого счетчика осуществляются за время $O(1)$.

Представление корневого счетчика. Корневой счетчик представляем расширяющимся массивом RootCount, каждый его элемент – это запись с тремя полями:

$$(Value, ForwardPointer, ListPointer),$$

которые интерпретируем следующим образом:

- *RootCount*[*i*].*Value* – *i*-й разряд, равный количеству деревьев ранга *i*;
- *RootCount*[*i*].*ForwardPointer* – прямой указатель *i*-го разряда;

- *RootCount* [*i*].*ListPointer* – указатель на список деревьев ранга *i*, присутствующих в толстой куче. Деревья в этом списке связаны при помощи указателя *Right* корневых узлов связываемых деревьев. Если в куче нет деревьев ранга *i*, то указатель *ListPointer* заземлен. Заметим, что если значение *RootCount* [*i*].*Value* равно нулю, то нам неважно, каково значение указателя *RootCount* [*i*].*ListPointer*.

Инициализация корневого счетчика (InitRootCount). Поскольку корневой счетчик реализован как массив записей, то возникает вопрос о величине данного массива и о том, что делать, когда весь этот массив заполнен. Чтобы была возможность оценить время инициализации счетчиков величиной $O(1)$, используем поразрядную их инициализацию. То есть будем добавлять новые разряды только тогда, когда возникает такая необходимость, и при этом инициализировать новый разряд сразу в обоих счетчиках. Для этого мы вводим переменную *MaxRank*, которая показывает нам, какая часть массивов счетчиков используется в данный момент.

При начальной инициализации необходимо установить счетчики в состояние, которое отвечает пустой куче. Очевидно, что в пустой куче не может быть никаких нарушений. Операция инициализации выглядит следующим образом.

Обновление прямого указателя i-го разряда корневого счетчика UpdateForwardPointer(i) заключается в выполнении операторов

```
If (RootCount[i+1].Value = 3-1)
  then RootCount[i].ForwardPointer:= RootCount[i+1].ForwardPointer
  else RootCount[i].ForwardPointer:= i+1;
```

Корректировка списочной части i-го разряда корневого счетчика при вставке в кучу нового дерева ранга i (InsertTree(i, p)). Эта процедура вставляет новое дерево ранга *i* (на него указывает указатель *p*) в списочную часть *i*-го разряда корневого счетчика *RootCount* и заключается в выполнении операторов

```
p1 := RootCount[i].ListPointer;
if (RootCount[i].Value ≠ 0) then p^.Right := p1 else p^.Right := nil;
p^.Left:= nil; RootCount[i].ListPointer:= p;
```

Корректировка списочной части i-го разряда корневого счетчика при удалении из кучи дерева ранга i (DeleteTree(i; p)). Эта процедура удаляет дерево ранга *i* (на него указывает указатель *p*) из списочной части *i*-го разряда корневого счетчика *RootCount*. Считаем, что указанное дере-

во присутствует в куче. Процедура заключается в выполнении операторов

```
p1:= RootCount[i].ListPointer;
If (p1 = p) then RootCount[i].ListPointer:= p^.Right;
j:= 1;
while (j ≤ RootCount[i].Value) and (p1^.Right ≠ p) do
begin j := j+1; p1 := p1^.Right End;
p1^.Right := p^.Right;
```

***Связывание* (*Fastening*(*p1, p2, p3*)) *трех толстых деревьев ранга i в одно толстое дерево ранга i* +1.** Эта функция принимает три указателя (*p*1, *p*2, *p*3) на три разных толстых дерева одного и того же ранга *i* и возвращает указатель на вновь сформированное дерево ранга *i* + 1. Процедура заключается в выполнении операторов

```
if (p1^.key ≤ p2^.Key) and (p1^.key ≤ p3^.Key) then
{MinP:=p1; p1:=p2; p2:=p3};
if (p2^.key ≤ p1^.Key) and (p2^.key ≤ p3^.Key) then
{MinP:=p2; p1:= p1; p2:= p3};
if (p3^.key ≤ p1^.Key) and (p3^.key ≤ p2^.Key) then
   {MinP:= p3; p1:= p1; p2:= p2};
   p1^.Right := p2; p1^.Left := nil; p1^.Parent := MinP;
   p2^.Right := MinP^.LChaild; p2^.Left := p1; p2^.Parent := MipP;
   if (PMin^.LChild ≠ NiL) then PMin^.LChild^.Left :=p2;
   MinP^.LChaild := p1; MinP^.Rank := MinP^.Rank +1;
   PMin^.Right := NiL; PMin^.Left :=NiL; Fastening:= MinP;
```

Функция GetKey* (*p*) *по указателю p на элемент определяет значение его ключа и реализуется оператором

```
if (p = nil) then Min := ∞ else Min := p ^.Key; GetKey := Min;
```

Функция MinKeyNodeRoot(p), которая по указателю *p* на списочную часть разряда корневого счетчика возвращает указатель на корневой узел с минимальным ключом, реализуется операторами

```
p1:= p; MinP:= p1;
while (p1 ≠ nil) do
begin if (p1^.Key < MinP^.Key) then MinP := p1; p1:= p1^.Right End
MinKeyNodeRoot := MinP;
```

Очевидно, что трудоемкость всех приведенных выше операций оце-

нивается величиной $O(1)$.

Операция фиксации (FixRootCount(i)). Операция фиксации i-го разряда корневого счетчика подразумевает, что его значение равно трем, а списочная часть содержит указатель на список деревьев ранга i, состоящий ровно из трех деревьев. При выполнении этой операции значение в i-м разряде должно стать равным нулю, а значение в $(i + 1)$-м разряде увеличиться на единицу. То есть в куче не должно остаться деревьев ранга i, а количество деревьев ранга $i + 1$ должно увеличиться на единицу. Для этого следует удалить из кучи три присутствующих в ней дерева ранга i, связать их в дерево ранга $i + 1$ и вставить вновь полученное дерево в кучу.

Следует учесть, что ранг нового дерева может стать больше, чем MaxRank, что потребует инициализации нового разряда. Для этого необходимо увеличить значение MaxRank на единицу и заполнить новое поле, а также провести инициализацию нового разряда

Операция фиксации осуществляется с помощью операторов

```
if (MaxRank = i) then {MaxRank:= i+1; RootCount[i+1]^.Value:= 0;
CountViolation[i+1].Value:= 0}
    else { UpdateForwardPointer(i+1)};
RootCount[i].Value:= 0;
p1:= RootCount[i].ListPointer; p2:= p1^.Right; p3:= p2^.Right;
p:= Fastening(p1, p2, p3); RootCount[i]^.ListPointer:= nil;
InsertTree(i+1, p);
RootCount[i+1].Value:= RootCount [i+1].Value + 1;
```

Очевидно, если списочная часть корневого счетчика до операции соответствовала избыточному корневому представлению, то и после операции фиксации это соответствие сохранится. Сохраняется также и регулярность представления. Трудоемкость данной операции $O(1)$.

Инкрементирование i-го разряда корневого счетчика (IncRootCount (i, p)). По сравнению с описанным алгоритмом инкрементирования i-го разряда избыточного представления здесь мы должны учесть работу со списочной частью и обновить прямые указатели. Процедура реализуется операторами

```
if (RootCount[i].Value = 1) or (RootCount[i].Value = 2)
    then if (RootCount [ RootCount[i].ForwardPointer ].Value = 3)
        then FixRootCount(RootCount[i].ForwardPointer);
if (RootCount[i].Value = 3) then FixRootCount(i);
```

```
InsertTree(i, p);
RootCount[i].Value:= RootCount[i].Value + 1;
UpdateForwardPointer(I);
if (RootCount[i].Value = 3) then FixRootCount(i);
```

Очевидно, если корневой счетчик находится в корректном состоянии и $i \le MaxRank$, то операция инкрементирования i-го разряда корневого счетчика переводит корневой счетчик в новое корректное состояние. Трудоемкость этой операции равна $O(1)$.

Процедура удаления дерева из кучи подразумевает наличие в куче этого дерева. Пусть удаляемое дерево имеет ранг i. Тогда значение i-го разряда избыточного корневого представления не равно нулю. То есть уменьшение этого значения на единицу не испортит регулярности представления и не потребует обновления каких-либо указателей. Необходимо лишь соответствующим образом обработать списочную часть. Процедура реализуется операторами

```
DeleteTree(i, p); RootCount[i].Value:= RootCount[i].Value –1;
```

Трудоемкость операции $O(1)$.

Нахождение дерева с минимальным ключом в корне (MinKey) реализуется операторами

```
MinP:= nil;
for i:= 0 to MaxRank do
begin
p1 := MinKeyNodeRoot (RootCount[i].ListPointer);
if (GetKey(p1) < GetKey(MinP)) then MinP:= p1;
end;
MinKey:= MinP;
```

Трудоемкость данной операции также $O(1)$.

Счетчик нарушений. К сожалению, здесь не удается разделить работу с избыточным представлением и списочной частью, как в корневом счетчике. Поэтому рассмотрим работу со счетчиком нарушений более подробно. Счетчик нарушений состоит из расширенного избыточного двоичного представления и набора списочных элементов.

Отличие заключается в том, что:

1. Нас теперь интересует не само число, а только значения разрядов.
2. Операция фиксации тесно связана с толстой кучей.

Значение i-го разряда для счетчика нарушений интерпретируется как

количество неправильных узлов ранга i, а его списочная часть – это указатели на неправильные узлы ранга i.

Такое определение счетчика нарушений дает возможность сделать несколько утверждений:

- наличие счетчика нарушений позволяет иметь доступ к любому неправильному узлу ранга i за время $O(1)$;
- уменьшение ключа у элемента ранга i соответствует операции инкрементирования i-го разряда счетчика нарушений (естественно, лишь в случае, когда новое значение ключа у изменяемого узла становится меньше значения ключа его родителя);
- операции инкрементирования и декрементирования i-го разряда осуществляются за время $O(1)$.

Представление счетчика нарушений. Счетчик нарушений – это расширяющийся массив, элементы которого являются записями из четырех полей

(*Value*, *ForwardPointer*, *FirstViolation*, *SecondViolation*)

со следующей интерпретацией: *CountViolation* [*i*].*Value* – количество неправильных узлов ранга i в куче, *CountViolation* [i].*ForwardPointer* – прямой указатель i-го разряда, *CountViolation* [*i*].*FirstViolation* и *CountViolation* [*i*].*SecondViolation* – указатели на неправильные узлы ранга i.

Заметим, что если значение *CountViolation* [*i*].*Value* равно единице, то важно лишь значение первого указателя *FirstViolation* и неважно значение второго *SecondViolation*. Если *CountViolation* [*i*].*Value* равно нулю, то неинтересны оба указателя.

Далее ограничимся рассмотрением только наиболее важных операций. Так как счетчик нарушений похож на описанный выше корневой счетчик, акцентируем внимание лишь на различиях. Реализация всех необходимых процедур оставляется читателю в качестве упражнения.

Инициализация нового звена. Для инициализации нового звена счетчика нарушений необходимо лишь занулить его значение в новом разряде. Делается это только тогда, когда мы вводим в кучу новое дерево ранга *MaxRank* + 1. Это первый момент появления в куче узла ранга *MaxRank* + + 1. Для тех нарушений, которые могут возникнуть в узлах ранга меньше либо равного *MaxRank* + 1, соответствующие разряды счетчика нарушений уже инициализированы, а узлов большего ранга в куче пока нет.

Вспомогательные процедуры

1. Процедура обновления прямого указателя i-го разряда счетчика

нарушений аналогична процедуре ***UpdateForwardPointer*(*i*)** для корневого счетчика. Необходимо лишь учесть, что счетчик нарушений двоичный.

2. Процедура корректировки списочной части *i*-го разряда счетчика нарушений при появлении в куче нового *i*-рангового нарушения – назовем ее ***InsertViolation* (*i*; *pNode*)** – вставляет новый нарушенный узел, обновляя, в зависимости от значения *CountViolation* [*i*].*Value*, либо первый (*FirstViolation*), либо второй (*SecondViolation*) указатель. Причем перед тем как вставлять в счетчик новое нарушение, необходимо проверить, не присутствует ли оно там.
3. Процедура взаимной замены поддеревьев кучи с корнями в узлах *p*1 и *p*2 – назовем ее ***InterChange* (*p*1, *p*2)** – подразумевает, что ранги обмениваемых деревьев одинаковы.
4. Также нам необходима функция ***SearchBrother*(*p*)**, которая возвращает указатель на брата того же ранга, что и передаваемый ей узел. Она проверяет ранги своего правого и левого братьев (если такие существуют) и возвращает указатель на брата того же ранга (такой существует обязательно).
5. Функция, которая связывает три толстых дерева ранга *i* в одно толстое дерево ранга *i* + 1, аналогична соответствующей функции для корневого счетчика.
6. Функция, которая возвращает указатель на минимальный нарушенный узел ранга *i* среди элементов *i*-го разряда счетчика нарушений. Если *i*-й разряд счетчика нарушений пуст, то возвращается nil.

Как и в случае корневого счетчика, все операции выполняются за константное время.

Свойство регулярности. Определим свойство регулярности для счетчика нарушений. Назовем состояние счетчика нарушений регулярным, если между любыми двумя цифрами, равными двум, существует цифра, отличная от единицы. Неправильный узел ранга *i* в дальнейшем будем называть *i*-ранговым нарушением.

Операция фиксации. Фиксация *i*-й цифры $d_i = 2$ соответствует либо преобразованию двух *i*-ранговых нарушений в одно $(i + 1)$-ранговое нарушение, либо устранению обоих *i*-ранговых нарушений. Проводить эту операцию предлагается следующим образом.

Упорядочиваем два *i*-ранговых нарушения так, чтобы они имели одного родителя (очевидно, что в общем случае *i*-ранговые нарушения мо-

гут иметь разных родителей). Сделать это предлагается заменой поддерева с корнем в нарушенном узле, чей родитель имеет меньший ключ, на поддерево с корнем в i-ранговом брате нарушаемого узла, чей родитель имеет больший ключ. Легко проверить, что такая замена не приводит к созданию новых нарушений. Пусть узел y – общий родитель двух нарушаемых узлов после замены принадлежит дереву F.

Разобьем дальнейшее рассмотрение на два случая.

1. Ранг y равен $i + 1$. Пусть F^1 и F^2 – это толстые деревья ранга i с корнями в двух нарушаемых узлах, а дерево F^y – толстое дерево ранга i, полученное из поддерева с корнем в узле y удалением поддеревьев F^1 и F^2.
   а) Если узел y не является корнем дерева F, то удаляем из дерева F поддерево F^y. Из трех толстых деревьев (F, F^1, F^2) ранга i образуем одно дерево ранга $i + 1$, чей корень z является узлом с наименьшим ключом среди корней деревьев F, F^1, F^2. Вставляем в дерево F вновь полученное толстое дерево с корнем в узле z вместо поддерева с корнем в узле y. Если узел z оказывается нарушенным, инкрементируем d_{i+1}. Значение i-го разряда делаем нулевым.
   б) Если узел y – корень дерева F, то удаляем дерево F из кучи. Из трех толстых деревьев (F, F^1, F^2) ранга i образуем одно дерево ранга i, чей корень z является узлом с наименьшим ключом среди ключей корней деревьев F, F^1, F^2. Вставляем вновь полученное толстое дерево с корнем в узле z в кучу. Значение i-го разряда делаем нулевым.
2. Если ранг y больше, чем $i + 1$, то, по условию регулярности счетчика нарушений, узел y должен иметь хотя бы одного сына w ранга $i + 1$, который не является $(i + 1)$-ранговым нарушением, и два i-ранговых сына w должны быть также ненарушенными. Тогда заменяем два нарушенных i-ранговых сына узла y на два хороших i-ранговых сына узла w. Тем самым мы свели задачу к случаю 1.

Можно доказать, что рассматриваемая операция не испортит регулярности счетчика.

***Инкрементирование i-го разряда счетчика нарушений* (*IncCount Violation* (*i, p*)).** Используя описанную выше операцию фиксации, можно осуществить инкрементирование i-го разряда счетчика нарушений следующими операторами.

```
FixCountViolation (i);
FixCountViolation (CountViolation [i]^.ForwardPointer);
  InsertViolation(i, pNode);
  CountViolation[i].Value:= CountViolation[i].Value + 1;
  FixCountViolation (i);
  FixCountViolation (CountViolation [i]^.ForwardPointer);
```

Трудоемкость операции $O(1)$.

Удаление нарушения из кучи. Заметим, что удаление нарушения из кучи подразумевает наличие в куче этого нарушения; пусть это нарушение ранга i. Тогда значение i-го разряда для счетчика нарушений не равно нулю. Следовательно, уменьшение этого значения на единицу не испортит регулярности и не потребует обновления каких-либо указателей. Необходимо лишь уменьшить на единицу значение переменной *CountViolation* [i].Value и обработать указатели *FirstViolation* и *SecondViolation*. Очевидно, что трудоемкость этой операции $O(1)$.

Нахождение узла с минимальным значением ключа среди всех нарушений. Для реализации этой функции предлагается перебрать все нарушения до максимального ранга и найти среди них узел с минимальным весом. Трудоемкость данной операции $O(\log n)$.

Основные операции:

Операция make-heap заключается в инициализации счетчиков; трудоемкость $O(1)$.

Операция FindMin возвращает указатель на минимальный элемент. Трудоемкость $O(1)$.

Операция Insert(key). Чтобы выполнить эту операцию, делаем новый элемент отдельным деревом и выполняем процедуру вставки нового элемента ранга 0 в корневой счетчик. После этого, если необходимо, корректируем значение указателя на минимальный элемент.

Операция уменьшения ключа DecreaseKey(Δ, p). Чтобы выполнить эту операцию, поступим следующим образом. Пусть x – узел, на который указывает указатель p. Вычитаем Δ из ключа узла x. Если новый ключ x меньше минимального ключа кучи H, обмениваем ключ элемента p с ключом минимального элемента. Новых нарушений операция не создаст. Пусть r – ранг x. Если x – нарушаемый узел, добавляем x как новое r-ранговое нарушение инкрементированием r-й цифры d_r счетчика нарушений. Трудоемкость $O(1)$.

Операция DeleteMin выполняется следующим образом. Удаляем под-

дерево с корнем в минимальном узле из леса. Минимальность этого элемента гарантирует нам, что среди его детей нарушений порядка кучи не было. То есть нет необходимости работать со счетчиком нарушений. Затем вставляем в кучу все деревья с корнями, расположенными в детях удаляемого узла. Очевидно, что новый минимальный ключ – либо в корне дерева леса, либо в нарушенном узле. Выполняем поиск нового минимального элемента среди корней деревьев и нарушенных узлов.

Если минимальный элемент оказался в нарушенном узле, то обмениваем его с элементом, хранимым в корне этого дерева, корректируя корневой счетчик, если это необходимо. После замены новый минимум – в корне дерева леса. Этот корень будет новым минимальным узлом. Трудоемкость операции равна $O(\log n)$.

Операция удаления элемента. Выполняется с помощью *DecreaseKey* и затем *DeleteMin*. Трудоемкость операции $O(\log n)$.

Операция Meld(h1, h2). Выполняется следующим образом. Первый шаг – фиксируются все нарушения в куче с меньшим максимальным рангом (разрывая связь произвольно). Не уменьшая общности, считаем, что эта куча – $h2$. Пройти по счетчику нарушений $h2$ от младшей цифры к старшей, пропуская цифры со значением 0. Для i-й цифры $d_i \neq 0$ делаем операцию фиксирования на каждой цифре, показываемой прямым указателем d_i, если эта цифра имеет значение 2. Затем, если $d_i = 2$, фиксируем d_i. Если $d_i = 1$, преобразуем это i-ранговое нарушение в $(i + 1)$-ранговое нарушение, как при фиксировании, используя i-рангового брата нарушенного узла вместо (несуществующего) другого i-рангового нарушения.

Как только $h2$ не будет содержать каких-либо нарушений, вставить корни из корневого счетчика $h2$ в корневой счетчик $h1$ инкрементированием соответствующих цифр. Если минимальный узел $h2$ содержит меньший ключ, чем минимальный узел $h1$, установить новым минимальным узлом $h1$ минимальный узел $h2$. Вернуть модифицированную кучу $h1$ в качестве результата *Meld*. Трудоемкость операции равна $O(\log n)$.

Операция DeleteViolation. Для освобождения кучи от нарушений достаточно выполнить операторы

```
for i:= 0 to h2^.MaxRank do
if (CountViolation[i].Value = 2) then FixCountViolation( i);
for i:= 0 to h2^.MaxRank do if (CountViolation[i].Value = 1) then
{IncCountViolation(i, SearchBrother (CountViolation[i].FirstViolation));
FixCountViolation(i)};
```

Основываясь на описанной выше реализации толстой кучи получаем следующий результат. В толстых кучах операции *FindMin*, *Insert* и *DecreaseKey* выполняются за время $O(1)$, а *Delete*, *DeleteMin* и *Meld* – за время $O(\log n)$.

Замечания. Существует альтернативное представление избыточных счетчиков. Вместо одной записи на цифру можно использовать одну запись на блок одинаковых цифр. Инкрементирование любой цифры можно выполнить за время $O(1)$, используя это альтернативное представление. Преимущество такого представления – возможность расширить счетчик на произвольное число одинаковых цифр за постоянное время.

Г. Бродал описывает кучевидную структуру, которая теоретически лучше, чем толстые кучи, так как их временная оценка для *Meld* – $O(1)$ в худшем случае. Структура Бродала, однако, намного сложнее толстых куч.

Сводные сведения о трудоемкости операций с приоритетными очередями

1. Трудоемкость операций над различными реализациями приоритетной очереди в худшем случае

Операции	*d*-куча	Левосторонняя куча	Ленивая левосторонняя куча	Самоорганизующаяся куча	Биномиальная очередь	Ленивая биномиальная очередь	Фибоначчиева куча	Слаборастущая куча (run-relaxed)	Куча Бродала
ВСТАВИТЬ	$O(\log_d n)$	$O(\log n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
МИН	$O(1)$	$O(1)$	$O(F)$*	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
УДАЛИТЬ МИН	$O(d \log_d n)$	$O(\log n)$	$O(F)$*	$O(n)$	$O(\log n)$	$O(n)$	. . .	$O(\log n)$	$O(\log n)$
УДАЛИТЬ	$O(d \log_d n)$	$O(\log n)$	$O(1)$	. . .	$O(\log n)$	. . .	. . .	$O(\log n)$	$O(\log n)$
УМЕНЬШИТЬ КЛЮЧ	$O(\log_d n)$	$O(\log n)$	$O(1)$	. . .	$O(\log n)$	. . .	. . .	$O(1)$	$O(1)$
СЛИТЬ	—	$O(\log n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$		$O(\log n)$	$O(1)$
ОБРАЗОВАТЬ ОЧЕРЕДЬ	$O(n)$	$O(n)$	$O(n)$	. . .	$O(n)$			$O(1)$	$O(1)$

* $F = k \max \{1, \log (n/(k + 1))\}$

2. Амортизационная трудоемкость выполнения операций

Операции	d-куча	Левосторонняя куча	Ленивая левосторонняя куча	Самоорганизующаяся куча	Биномиальная очередь	Ленивая биномиальная очередь	Фибоначчиева куча	Тонкая куча (thin heap)	Толстая куча (fat heap)
ВСТАВИТЬ	$O(\log_d n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
МИН	$O(1)$	$O(1)$	. . .	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
УДАЛИТЬ МИН	$O(d \log_d n)$	$O(\log n)$	. . .	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
УДАЛИТЬ	$O(d \log_d n)$	$O(\log n)$	$O(1)$	. . .	$O(\log n)$	. . .	$O(\log n)$	$O(\log n)$	$O(\log n)$
УМЕНЬШИТЬ КЛЮЧ	$O(\log_d n)$	$O(\log n)$	$O(1)$	. . .	$O(\log n)$	. . .	$O(1)$	$O(1)$	$O(1)$
СЛИТЬ	—	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
ОБРАЗОВАТЬ ОЧЕРЕДЬ	$O(n)$	$O(n)$	$O(n)$	. . .	$O(n)$		$O(1)$	$O(1)$	$O(1)$

Глава 5. ПОИСКОВЫЕ ДЕРЕВЬЯ

5.1. Двоичные деревья поиска

Общие сведения. Деревья поиска предназначены для представления словарей как абстрактного типа данных. Так же как и приоритетные очереди, они представляют взвешенные множества, но с другим набором операций, а именно:

- ***Search*** – поиск элемента с заданным ключом;
- ***Minimum*** – поиск элемента с минимальным ключом;
- ***Maximum*** – поиск элемента с максимальным ключом;
- ***Predecessor*** – поиск элемента с предыдущим ключом;
- ***Successor*** – поиск элемента со следующим ключом;
- ***Insert*** – вставка элемента со своим ключом;
- ***Delete*** – удаление указанного элемента.

Считается, что каждый элемент словаря имеет ключ (вес), принимающий значение из какого-либо линейно упорядоченного множества. Таким множеством может быть, например, числовое множество или множество слов в некотором алфавите. В последнем случае в качестве линейного порядка можно рассматривать лексикографический порядок. Таким образом, дерево поиска может быть использовано и как словарь, и как приоритетная очередь.

Время выполнения основных операций пропорционально высоте дерева. Если каждый внутренний узел двоичного дерева имеет ровно двух потомков, то его высота и время выполнения основных операций пропорциональны логарифму числа узлов. Напротив, если дерево представляет собой линейную цепочку из n узлов, это время вырастает до $\Theta(n)$. Известно, что высота случайного двоичного дерева поиска есть $O(\log n)$, так что в этом случае время выполнения основных операций есть $\Theta(\log n)$.

Конечно, возникающие на практике двоичные деревья поиска могут быть далеки от случайных. Однако, приняв специальные меры по балансировке деревьев, мы можем гарантировать, что высота деревьев с n узлами будет $O(\log n)$. Ниже рассмотрим один из подходов такого рода (красно-черные деревья и как частный случай АВЛ-деревья). Будут рассмотрены также Б-деревья, которые особенно удобны для данных, хранящихся во вторичной памяти с произвольным доступом (на диске).

Представление двоичных деревьев поиска. Двоичным деревом поиска называется корневое двоичное дерево, каждому узлу которого поставлен в соответствие взвешенный элемент. При этом для каждого узла x выполняется следующее условие:

Веса всех узлов левого поддерева в дереве с корнем x меньше, а веса узлов его правого поддерева больше веса узла x или равны ему.

Представляется такое дерево узлами следующего вида

$$Node = (element, key, left, right, parent).$$

Доступ к дереву T осуществляется с помощью ссылки *root*.

Процедура Walk (x) обходит все узлы поддерева с корнем в узле x и печатает их ключи в неубывающем порядке.

```
pocedure Walk (x);
begin
  if (x ≠ nil) then {Walk (left[x]); write (key[x]); Walk (right[x])}
end;
```

Свойство упорядоченности гарантирует правильность алгоритма. Время работы на дереве с n вершинами есть $\Theta(n)$, каждая вершина обрабатывается один раз. Оператор *Walk*(*root*) напечатает ключи всех элементов в неубывающем порядке.

Заметим, что порядок, при котором корень предшествует узлам обоих поддеревьев, называется ***preorder***; порядок, в котором корень следует за ними, называется ***postorder***.

Упражнения

1. Нарисуйте двоичные деревья поиска высоты 2, 3, 4, 5 и 6 для одного и того же множества ключей 1, 4, 5, 10, 16, 17, 21.

2. Напишите нерекурсивный алгоритм, печатающий ключи в двоичном дереве поиска в неубывающем порядке.

3. Напишите рекурсивные алгоритмы для обхода деревьев в различных порядках (*preorder*, *postorder*). Как и раньше, время работы должно быть $O(n)$ (где n – число вершин).

4. Покажите, что любой алгоритм построения двоичного дерева поиска, содержащего заданные n элементов, требует времени $\Omega(n \cdot \log n)$. Воспользуйтесь тем, что сортировка n чисел требует $\Omega(n \cdot \log n)$ действий.

Операции с двоичным поисковым деревом. Покажем, что двоичные поисковые деревья позволяют выполнять операции *Search*, *Minimum*,

Maximum, *Successor* и *Predecessor* за время $O(h)$, где h – высота дерева.

Поиск (*Search*). Процедура поиска получает на вход искомый ключ k и указатель x на корень поддерева, в котором производится поиск. Она возвращает указатель на вершину с ключом k (если такая есть) или nil (если такой вершины нет).

```
procedure Search (x, k);
  begin
    if (x = nil) or (k = key [x]) then return x;
    if (k < key [x]) then return Search (left[x], k) else
        return Search (right[x], k)
  end;
```

В процессе поиска мы двигаемся от корня, сравнивая ключ k с ключом, хранящимся в текущей вершине x. Если они равны, поиск завершается. Если $k < key[x]$, то поиск продолжается в левом поддереве x. Если $k > key[x]$, то поиск продолжается в правом поддереве. Длина пути поиска не превосходит высоты дерева, поэтому время поиска есть $O(h)$ (где h – высота дерева).

Итеративная версия процедуры Поиск

```
procedure IterativeSearch (x, k);
  begin
    while (x ≠ nil) and (k ≠ key [x]) do
      if k < key [x] then x:= left[x] else x:= right[x];
    return (x)
  end;
```

Минимум и Максимум. Элемент с минимальным ключом в дереве поиска можно найти, пройдя от корня по указателям *left* пока не упремся в nil. Процедура *Minimum*(x) возвращает указатель на найденный элемент поддерева с корнем x.

```
procedure Minimum(x);
  begin While left [x] ≠ nil do x:= left[x]; Return (x) end;
```

Алгоритм *Maximum* симметричен:

```
procedure Maximum(x);
  begin while (right [x] ≠ nil) do x:= right[x]; return (x) end;
```

Оба алгоритма требуют времени $O(h)$, где h – высота дерева (поскольку-

ку двигаются по дереву только вниз).

Следующий и предыдущий элементы. Если x – указатель на некоторый узел дерева, то процедура *Successor*(x) возвращает указатель на узел со следующим за x элементом или nil, если указанный элемент последний в дереве.

```
procedure Successor(x);
  begin
    If (right[x] ≠ nil) then Return Minimum (right[x]);
    y:= p[x];
    while (y ≠ nil) and (x=right [y]) do {x:= y; y:= parent[y]};
    Return y
  end;
```

Приведенная процедура отдельно рассматривает два случая. Если правое поддерево вершины x не пусто, то следующий за x элемент – минимальный элемент в этом поддереве и он равен *Minimum*(*right*[x]). Если правое поддерево вершины x пусто, то идем от x вверх, пока не найдем вершину, являющуюся левым сыном своего родителя. Этот родитель (если он есть) и будет искомым элементом. Время работы процедуры *Successor* на дереве высоты h есть $O(h)$, так как мы двигаемся либо только вверх, либо только вниз. Процедура *Predecessor* симметрична.

Упражнения

1. Пусть поиск ключа в двоичном дереве завершается в листе. Рассмотрим три множества: A – элементы слева от пути поиска, B – элементы на пути и C – справа от пути. Верно ли, что для любых трех ключей $a \in A$, $b \in B$ и $c \in C$ выполняются неравенства $a \le b \le c$?

2. Докажите, что k последовательных вызовов процедуры *Successor* выполняются за $O(k + h)$ шагов (h – высота дерева) независимо от того, с какой вершины мы начинаем.

3. Пусть T – двоичное дерево поиска, все ключи в котором различны, x – его лист, а y – родитель узла x. Покажите, что $key[y]$ является соседним с $key[x]$ ключом (следующим или предыдущим).

Добавление элемента. Процедура *Insert* (T, z) добавляет заданный элемент в подходящее место дерева T. Параметром процедуры является указатель z на новую вершину, в которую помещены значения $key[z]$, $left[z] = nil$ и $right[z] = nil$. В ходе работы процедура изменяет дерево T и (возможно) некоторые поля вершины z, после чего новая вершина с дан-

ным значением ключа оказывается вставленной в подходящее место дерева.

```
procedure Insert(T, z);
begin
  y := nil; x := root;
  while (x ≠ nil) do
    {y := x; if key[z] < key[x] then x := left[x] else x := right[x]};
  p [z] := y;
  if y = nil then root := z else if key[z] < key[y] then left[y] := z else
    right [y]:= z
end;
```

Подобно процедурам *Search* и *IterativeSearch*, процедура *Insert* двигается вниз по дереву, начав с его корня. При этом в вершине y сохраняется указатель на родителя вершины x. Сравнивая $key[z]$ с $key[x]$, процедура решает куда идти – налево или направо. Процесс завершается, когда x становится равным *nil*. Этот *nil* стоит как раз там, куда надо поместить z, что и делается. Очевидно, добавление требует времени $O(h)$ для дерева высоты h.

Удаление элемента. Параметром процедуры удаления является указатель z на удаляемую вершину. При удалении возможны три случая. Если у z нет детей, для удаления z достаточно поместить nil в соответствующее поле его родителя вместо z. Если у z есть один ребенок, можно вырезать z, соединив его родителя напрямую с его ребенком. Если же детей двое, находим следующий за z элемент y; у него нет левого ребенка. Теперь можно скопировать ключ и дополнительные данные из вершины y в вершину z, а саму вершину y удалить описанным выше способом.

Упражнения

1. Напишите рекурсивный вариант процедуры *Insert*.

2. Напишите процедуру *Delete*, удаляющую элемент z из дерева T.

3. Набор из n чисел можно отсортировать, сначала добавив их один за другим в двоичное дерево поиска с помощью процедуры *Insert*, а потом обойти дерево с помощью процедуры *Walk*. Оцените время работы такого алгоритма.

4. Покажите, что если вершина двоичного дерева поиска имеет двоих детей, то следующая за ней вершина не имеет левого ребенка, а предшествующая – правого.

Случайные двоичные деревья поиска. Поскольку основные операции с двоичными деревьями поиска требуют времени $O(h)$, где h – высота дерева, важно знать, какова высота «типичного» дерева. Для этого принимают какие-то статистические предположения о распределении ключей и последовательности выполняемых операций. К сожалению, в общем случае ситуация трудна для анализа. Если определить случайное двоичное дерево из n различных ключей как дерево, получающееся из пустого дерева добавлением этих ключей в случайном порядке, считая все $n!$ перестановок равновероятными, то можно доказать, что средняя высота случайного двоичного дерева поиска, построенного по n различным ключам, равна $O(\log n)$.

5.2. Красно-черные деревья

Мы видели, что основные операции с двоичным поисковым деревом высоты h могут быть выполнены за $O(h)$ действий. Деревья эффективны, если их высота мала, но если не принимать специальные меры при выполнении операций, малая высота не гарантируется, и в этом случае деревья не более эффективны, чем списки.

Для повышения эффективности операций используют различные приемы перестройки деревьев, чтобы высота дерева была величиной $O(\log n)$. Такие приемы называются балансировкой деревьев. При этом используются разные критерии качества балансировки. Одним из видов сбалансированных деревьев поиска являются так называемые красно-черные деревья, для которых предусмотрены операции балансировки, гарантирующие оценку высоты величиной $O(\log n)$.

Частным случаем такой балансировки является АВЛ-балансировка, при которой у каждого узла высота его левого поддерева отличается от высоты правого не более чем на единицу. Заметим, что наихудшими в некотором смысле АВЛ-деревьями являются деревья Фибоначчи T_h ($h = 0$, $1, 2, \ldots$), определяемые следующим образом: T_0 – пустое дерево, T_1 – дерево, состоящее из одного узла. При $h > 1$ дерево T_h состоит из корня с левым поддеревом T_{h-1} и правым – T_{h-2}. Нетрудно видеть, что при заданной величине h дерево T_h имеет наименьшее число узлов среди всех АВЛ-деревьев высоты h.

Для удобства поисковые деревья будем расширять, вводя дополнительный фиктивный узел (nil-узел) и считая его потомком каждого узла исходного дерева, у которого нет правого или левого или обоих потомков, его же считаем родителем корня.

Красно-черное дерево – это расширенное двоичное дерево поиска, вершины которого разделены на красные (red) и черные (black) так, что

1. Каждый узел либо красный, либо черный.
2. Каждый лист (nil-узел) – черный.
3. Если узел красный, то оба его ребенка черные.
4. Все пути, идущие вниз от корня к листьям, содержат одинаковое количество черных узлов.

Свойства 1–4 называют RB-свойствами. Узлы красно-черного дерева будем представлять записями вида

$$Node = (color, key, left, right, parent).$$

Комбинаторные свойства красно-черных деревьев. Для произвольного узла x определим черную высоту $bh(x)$ как количество черных узлов на пути из x в некоторый лист, не считая сам узел x. По свойству 4 эта сумма не зависит от выбранного листа. Черной высотой дерева будем считать черную высоту его корня.

Пусть $size[x]$ – количество внутренних узлов в поддереве с корнем x (nil-узлы не считаются).

Лемма 1. *Для произвольного узла x красно-черного дерева выполняется неравенство*

$$size[x] \geq 2^{bh(x)} - 1.$$

Доказательство. Если x – лист, то $bh(x) = 0$ и $size[x] = 0$, следовательно, утверждение леммы выполнено. Далее, пусть для узлов $left[x]$ и $right[x]$ утверждение леммы справедливо, то есть

$$size[left[x]] \geq 2^{bh(left[x])} - 1 \text{ и } size[right[x]] \geq 2^{bh(right[x])} - 1,$$

тогда

$$size[x] = size[left[x]] + size[right[x]] + 1 \geq$$
$$\geq (2^{bh(left[x])} - 1) + (2^{bh(right[x])} - 1) + 1 =$$
$$= 2^{bh(left[x])} + 2^{bh(right[x])} - 1 \geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \geq 2^{bh(x)} - 1.$$

Предпоследнее неравенство справедливо в силу соотношения

$$bh(left[x]) \geq (bh(x) - 1) \text{ и } bh(right[x]) \geq (bh(x) - 1).$$

Лемма 2*. Красно-черное дерево с n внутренними узлами* (*nil-листья не считаются*) *имеет высоту не больше* $2log(n + 1)$.

Доказательство. Обозначим высоту дерева через h. Согласно свой-

ству 3, по меньшей мере половину всех вершин на пути от корня к листу, не считая корень, составляют черные вершины. Следовательно, черная высота дерева не меньше $h/2$. Тогда $n \geq 2^{h/2} - 1$ и, переходя к логарифмам, получаем $\log(n+1) \geq h/2$ или $h \leq 2\log(n+1)$. Лемма доказана.

Полученная оценка высоты красно-черных деревьев гарантирует выполнение операций *Search*, *Minimum*, *Maximum*, *Successor* и *Predecessor* с красно-черными деревьями за время $O(\log n)$. Сложнее обстоит дело с процедурами *Insert* и *Delete*: проблема в том, что они могут испортить структуру красно-черного дерева, нарушив RB-свойства. Поэтому эти процедуры придется модифицировать. Ниже увидим, как можно реализовать их за время $O(\log n)$ с сохранением RB-свойств.

Упражнения

1. Предположим, что корень красно-черного дерева красный. Если мы покрасим его в черный цвет, останется ли дерево красно-черным?

2. Покажите, что самый длинный путь вниз от вершины x к листу не более чем вдвое длиннее самого короткого такого пути.

3. Какое наибольшее и наименьшее количество внутренних узлов может быть в красно-черном дереве черной высоты k?

Вращения – это манипуляции с красно-черными деревьями с целью восстановления RB-свойств в случае их нарушения. Их используют при реализации операций *Insert* и *Delete*. Вращение представляет собой локальную операцию, при которой меняется несколько указателей, но свойство упорядоченности сохраняется.

На рис. 1 показаны два взаимно обратных вращения: левое и правое.

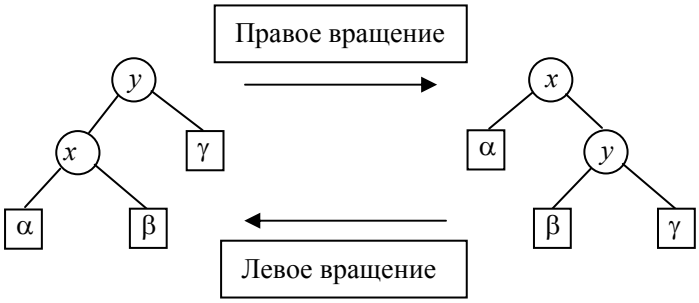


Рис. 1

Левое вращение возможно в любом узле x, правый ребенок которого (назовем его y) не является листом (nil). После вращения y оказывается корнем поддерева, x – левым ребенком узла y, а бывший левый ребенок y

– правым ребенком узла x.

Упражнения

1. Покажите, что левое и правое вращения можно осуществить за время $O(1)$.

2. Напишите процедуры *LeftRotate*(T,x) и *RightRotate*(T,x), реализующие левое и правое вращение в дереве T относительно узла x.

3. Пусть a, b и c – произвольные узлы в поддеревьях α, β и γ на рис. 1 (справа). Как изменится глубина a, b и c при выполнении левого вращения?

4. Покажите, что произвольное двоичное дерево поиска с n узлами может быть преобразовано в любое другое дерево с тем же числом узлов (и теми же ключами) с помощью $O(n)$ вращений. (Указание: сначала покажите, что $n-1$ правых вращений достаточно, чтобы преобразовать любое дерево в идущую вправо цепочку.)

5. Напишите процедуры *Insert*(T,x) и *Delete*(T,x), которые добавляют и удаляют элемент x из дерева T за время $O(\log n)$.

6. Разработайте алгоритм объединения двух красно-черных деревьев в одно красно-черное дерево за время $O(\log n)$.

Комбинаторные свойства АВЛ-деревьев. АВЛ-балансировка по определению требует, чтобы для каждого узла высота его правого поддерева отличалась от высоты левого не более чем на единицу.

Пусть n_k – минимальное число узлов в АВЛ-дереве высоты k. Тогда $n_0 = 1$, $n_1 = 2$, $n_2 = 4$, $n_k = n_{k-1} + n_{k-2} + 1$ при $k \ge 2$.

Пусть $\alpha = (1+\sqrt{5}) / 2$ (положительный корень уравнения $x^2 - x - 1$).

Теорема. *Для любого $k \ge 3$ выполняется неравенство $n_k \ge \alpha^{k+1}$.*

Доказательство. Непосредственно проверяется базис индукции $n_3 \ge \alpha^4$, $n_4 \ge \alpha^5$.

Предположим, что при $k = l$ выполняется $n_k \ge \alpha^{k+1}$, и докажем при $k = l + 1$. Действительно, $n_{l+1} = n + n_{l-1} + 1 > \alpha^{l+1} + \alpha^l + 1 > \alpha^{l+2}$. Докажем последнее в цепочке неравенство. Пусть $\alpha^{l+1} + \alpha^l + 1 \le \alpha^{l+2}$, тогда $\alpha^{l+2} - \alpha^{l+1} - \alpha^l - 1 \ge 0$ или $\alpha^l(\alpha^2 - \alpha - 1) - 1 \ge 0$ и $-1 \ge 0$, противоречие.

Следствие. Для любого АВЛ-дерева высоты k с n узлами выполняется неравенство $k + 1 < \log_\alpha n = \log_\alpha 2 \log_2 n \approx 1{,}44 \cdot \log_2 n$, что обеспечивает «логарифмическую трудоемкость» выполнения основных операций с АВЛ-деревом.

Замечания. Идея балансировки двоичных деревьев поиска принадлежит Г.М. Адельсону-Вельскому и Е.М. Ландису, предложившим в 1962 г.

класс сбалансированных деревьев, называемых теперь АВЛ-деревьями. Баланс поддерживается с помощью процедуры вращения. Для его восстановления в дереве с n узлами после добавления или удаления узла может потребоваться $\Theta(\log n)$ вращений.

Еще один класс деревьев поиска, называемых 2-3-деревьями, был предложен Дж. Хопкрофтом в 1970 г. Здесь баланс поддерживается за счет изменения степеней узлов. Обобщение 2-3-деревьев предложили Д. Байер и Е. Мак-Крейт. Их деревья называются Б-деревьями, которые мы рассмотрим в следующем разделе.

Красно-черные деревья предложил Д. Байер, назвав их симметричными двоичными Б-деревьями. Л. Гибас подробно изучил их свойства и предложил использовать для наглядности красный и черный цвета [7].

Из многих других вариаций на тему сбалансированных деревьев наиболее любопытны, видимо, расширяющиеся деревья, которые придумали Д. Слеатор и Р. Тарьян. Эти деревья являются саморегулирующимися. Хорошее описание расширяющихся деревьев дал Тарьян. Расширяющиеся деревья поддерживают баланс без использования дополнительных полей (типа цвета). Вместо этого расширяющие операции, включающие вращения, выполняются при каждом обращении к дереву. Учетная стоимость в расчете на одну операцию с деревом для расширяющихся деревьев составляет $O(\log n)$.

Упражнения

1. Напишите процедуру *Insert*(T, z) для вставки элемента z в АВЛ-дерево T.

2. Напишите процедуру *Delete*(T, z) для удаления элемента z из АВЛ-дерева Т.

5.3. Б-деревья

Б-деревья –это один из видов сбалансированных деревьев, при котором обеспечивается эффективное хранение информации на магнитных дисках и других устройствах с прямым доступом. Б-деревья похожи на красно-черные деревья. Разница в том, что в Б-дереве узел может иметь много детей, на практике до тысячи, в зависимости от характеристик используемого диска. Благодаря этому константа в оценке $O(\log n)$ для высоты дерева существенно *меньше*, чем для черно-красных деревьев. Как и черно-красные деревья, Б-деревья позволяют реализовать многие операции с множествами размера n за время $O(\log n)$.

Узел x, хранящий $n[x]$ ключей, имеет $n[x] + 1$ детей. Хранящиеся в x

ключи служат границами, разделяющими всех его потомков на $n[x] + 1$ групп; за каждую группу отвечает один из детей x. При поиске в Б-дереве мы сравниваем искомый ключ с $n[x]$ ключами, хранящимися в x, и по результатам сравнения выбираем одного из $n[x] + 1$ потомков.

Особенности работы со структурами данных, размещаемых на диске. Алгоритмы, работающие с Б-деревьями, хранят в оперативной памяти лишь небольшую часть всей информации (фиксированное число секторов).

Диск рассматривается как большой участок памяти, работа с которым происходит следующим образом: перед тем как работать с объектом x, мы должны выполнить специальную операцию *Disk-Read*(x) (чтение с диска). После внесения изменений в наш объект x мы выполняем операцию *Disk-Write* (x) (запись на диск).

Время работы программы в основном определяется количеством этих операций, так что имеет смысл читать/записывать возможно больше информации за один раз и сделать так, чтобы узел Б-дерева заполнял полностью один сектор диска. Таким образом, степень ветвления (число детей узла) определяется размером сектора.

Типичная степень ветвления Б-деревьев находится между 50 и 2000 в зависимости от размера элемента. Увеличение степени ветвления резко сокращает высоту дерева, и тем самым число обращений к диску, при поиске. Например, Б-дерево степени 1001 и высоты 2 может хранить более миллиарда ключей. Учитывая, что корень можно постоянно хранить в оперативной памяти, достаточно двух обращений к диску при поиске нужного ключа.

Считаем, что дополнительная прикладная информация, связанная с ключом, хранится в том же узле дерева. На практике это не всегда удобно, и в реальном алгоритме узел может содержать лишь ссылку на сектор, где хранится эта дополнительная информация. Считаем, что при перемещениях ключа дополнительная информация (или ссылка на нее) перемещается вместе с ним. Тем самым элементом Б-дерева будет ключ вместе со связанной с ним информацией.

Замечание. Часто используется другая организация Б-деревьев, при которой сопутствующая информация помещается в листьях, где больше места, так как не надо хранить ключи, а во внутренних узлах хранятся только ключи и указатели на детей.

Определение Б-дерева. Б-деревом называют корневое дерево, устроенное следующим образом. Каждый узел x содержит поля:

- $n[x]$ – количество ключей, хранящихся в узле x;
- $key_1[x]$, $key_2[x]$, … , $key_{n[x]}[x]$ – сами ключи в неубывающем порядке;
- $leaf\,[x]$ – булевское значение, истинное, когда узел x является листом.

Если x – внутренний узел, то он содержит указатели $c_1[x]$, $c_2[x]$, …, $c_{n[x]+1}[x]$ на его детей в количестве $n[x] + 1$.

- У листьев детей нет, и эти поля для них не определены.
- Все листья находятся на одной и той же глубине, равной высоте дерева.
- Возможное число ключей, хранящихся в одном узле, определяется параметром $t \geq 2$, которое называется минимальной степенью Б-дерева.
- Для каждого некорневого узла x выполняются неравенства $(t - 1) \leq \leq n[x] \leq (2t - 1)$. Таким образом, число детей у любого внутреннего узла (кроме корня) находится в пределах от t до $2t$.
- Если дерево не пусто, то в корне должен храниться хотя бы один ключ. Узел, хранящий ровно $2t - 1$ ключей, назовется полным.

Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях. Точнее,

- $c_1[x]$ ссылается на поддерево, ключи в котором меньше, чем $key_1[x]$;
- $c_i[x]$ при $i = 2, 3, \ldots, n$ ссылается на поддерево, ключи в котором находятся в пределах от $key_{i-1}[x]$ до $key_i[x]$;
- $c_{n[x]+1}[x]$ ссылается на поддерево, ключи в котором больше, чем $key_{n[x]}[x]$.

В простейшем случае, когда $t = 2$, у каждого внутреннего узла 2, 3 или 4 ребенка, и мы получаем так называемое 2-3-4-дерево. Для эффективной работы с диском на практике t надо брать достаточно большим. Число обращений к диску для большинства операций пропорционально высоте Б-дерева. Оценим сверху эту высоту.

Теорема. *Для всякого Б-дерева T высоты h и минимальной степени t, хранящего n ≥ 1 ключей, выполнено неравенство* $h \leq log_t\,(n+1/2)$.

Доказательство. Наименьшее число узлов в дереве высоты h будет в случае, если степень каждого узла минимальна, то есть у корня 2 ребенка, а у внутренних узлов по t детей. В этом случае на глубине 1 мы имеем 2 узла, на глубине 2 имеем $2t$ узлов, на глубине 3 имеем $2t^2$ узлов, на глубине h имеем $2t^{h-1}$ узлов. При этом в корне хранится один ключ, а во всех остальных узлах по $t - 1$ ключей. Таким образом, получаем неравенство

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2 \cdot t^{i-1} = 1 + 2\,(t-1)\,((t^h - 1)/(t-1)) = 2t^h - 1,$$

откуда следует утверждение теоремы.

Как и для красно-черных деревьев, высота Б-дерева с n узлами есть $O(\log n)$, но основание логарифма для Б-деревьев гораздо больше, что примерно в $\log t$ раз сокращает количество обращений к диску.

Основные операции с Б-деревьями. Можем считать, что корень Б-дерева всегда находится в оперативной памяти, то есть операция чтения с диска для корня никогда не требуется; однако всякий раз, когда мы изменяем корень, мы должны его сохранять на диске. Все узлы, передаваемые как параметры, уже считаны с диска. Все процедуры обрабатывают дерево за один проход от корня к листьям.

Поиск в Б-дереве. Поиск в Б-дереве похож на поиск в двоичном дереве. Разница в том, что в каждом узле x мы выбираем один вариант из $(n[x] + 1)$, а не из двух. При поиске просматриваются узлы дерева от корня к листу. Поэтому число обращений к диску есть $\theta(h) = \theta(\log_t n)$, где h – высота дерева, а n – количество ключей. Так как $n[x] \leq 2t$, то время вычислений равно $O(th) = O(t \cdot \log_t n)$.

Создание пустого Б-дерева. Пустое дерево создается с помощью процедуры, которая находит место на диске для нового узла и размещает его. Считаем, что это можно реализовать за время $O(1)$ и не использовать операцию чтения с диска.

Добавление элемента в Б-дерево. При выполнении этой операции используется процедура разбиения полного (с $2t - 1$ ключами) узла y на два узла, имеющие по $t - 1$ элементов в каждом. При этом ключ-медиана $key_t[y]$ отправляется к родителю x узла y и становится разделителем двух полученных узлов. Это возможно, если узел x неполон. Если y – корень, процедура работает аналогично. В этом случае высота дерева увеличивается на единицу.

Процедура *Insert* добавляет элемент k в Б-дерево T, пройдя один раз от корня к листу. На это требуется время $O(th) = O(t \cdot \log_t n)$ и $O(h)$ обращений к диску, если высота дерева h. По ходу дела с помощью процедуры разбиения разделяются встречающиеся полные узлы. Заметим, что если полный узел имеет неполного родителя, то его можно разделить, так как в родителе есть место для дополнительного ключа, поэтому, поднимаясь вверх, доходим до неполного листа, куда и добавляем новый элемент.

Удаление элемента из Б-дерева. Удаление элемента из Б-дерева про-

исходит аналогично добавлению, хотя немного сложнее. Читателю предоставляется возможность разработать процедуру удаления, которая требует $O(h)$ обращений к диску для Б-дерева высоты h, при этом вся процедура требует $O(t \cdot h) = O(t \cdot \log_t n)$ времени.

В заключение заметим, что сбалансированные деревья и Б-деревья обсуждаются в книгах Д. Кнута, А. Ахо, Дж. Хопкрофта и Дж. Ульмана. Подробный обзор Б-деревьев дан в книге Т. Кормена и др. Л. Гибас и Р. Седжвик рассмотрели взаимосвязи между разными видами сбалансированных деревьев, включая красно-черные и 2-3-4-деревья.

В 1970 г. Дж. Хопкрофт предложил понятие 2-3-деревьев, которые явились предшественниками Б-деревьев и 2-3-4-деревьев. В этих деревьях каждая внутренняя вершина имеет 2 или 3 детей. Б-деревья были определены Д. Байером и Е. Мак-Крейтом в 1972 г.

Список литературы

1. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Мир, 1979.

2. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Вильямс, 2000.

3. Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М.: Мир, 1982.

4. Емеличев, В.А. Лекции по теории графов / В.А. Емеличев и др. – М.: Наука, 1990.

5. Кормен, Т. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзер, Р. Ривест. МЦНМО, Москва, 1999.

6. Кристофидес, Н. Теория графов. Алгоритмический подход / Н. Кристофидес. – М.: Мир, 1978.

7. Ловас, Л. Прикладные задачи теории графов / Л. Ловас, М. Пламмер. – М.: Мир, 1998.

8. Липский, В. Комбинаторика для программистов / В. Липский. – М.: Мир, 1988.

9. Новиков, Ф.А. Дискретная математика для программистов / Ф. А. Новиков. – СПб.: Питер, 2001.

10. Рейнгольд, Э. Комбинаторные алгоритмы / Э. Рейнгольд, Ю. Нивергельт, Н. Део. – М.: Мир, 1980.

11. Brodal, G.S. Fast meldable priority queues / G.S. Brodal // Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95), Springer, 1995. – P. 282–290.

12. Brodal, G.S. Optimal purely functional priority queues / G.S. Brodal, C. Okasaki // Journal of Functional Programming. – 1996. – 6(6). – P. 839–858.

13. Brodal, G.S. Worst case priority queues / G.S. Brodal // Proceeding 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA 96). ACM Press, 1996. – P. 52–58.

14. Brown, M.R. Implementation and analysis of binomial queue algorithms / M.R. Brown // SIAM J. Computing. – 1978. – 7(3). – P. 298–319.

15. Clancy, M.J. A programming and problem-solving seminar / M.J. Clancy, D.E. Knuth // Technical Report STAN-CS-77-606 / Department of Computer Science, Stanford University, Palo Alto. – 1977.

16. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation / J.R. Driscoll, H. N. Gabow, R. Shrairman, R.E. Tarjan // Communication of the ACM. – 1988. – 31(11). P. 1343–1354.

17. Fredman, M.L. Fibonacci heaps and their uses in improved network optimization algorithms / M.L. Fredman, R.E. Tarjan // Journal of the ACM. – 1987. –

34(3). – P. 596–615.

18. A new representation for linear lists / L.J. Guibas, E.M. McCreight, M.F. Plass, and J.R. Roberts // Proc. of the 9th Annual ACM Symposium on Theory of computting. ACM Press, 1977. – P. 49–60.

19. Kaplan, H. Persistent lists with catenation via recursive slow-down / H. Kaplan and R.E. Tarjan // Proceedings of the 27th Annual ACM Symposium on Theory of Computing. ACM Press, 1995. – P. 93–102.

20. Kaplan, H. Purely functional representations of catenable sorted lists / H. Kaplan and R.E. Tarjan // Proceedings of the 28th Annual ACM Symposium on Theory of Computing. ACM Press, 1996. – P. 202–211.

21. Knuth, D.E. The Art of Computer Programming. V. 1. Fundamental Algorithms: Reading / D.E. Knuth. – Second edition. – Addison-Wesley, Massachusetts, 1973.

22. Tarjan, R. E. Amortized computational complexity / R.E. Tarjan // SIAM J. on Algebraic and Discrete Methods. – 1985. – 6(2). – P. 306–318.

23. Vuillemin, J. A data structure for manipulating priority queues / J. Vuillemin // Communications of the ACM. – 1978. – 21. – P. 309–314.

24. Kaplan, H. New Heap Data Structures / H. Kaplan, R.E. Tarjan. New Heap Data Structures, Technical Report TR 597-99. Priston University, 1999.

Оглавление

Владимир Евгеньевич Алексеев
Владимир Александрович Таланов

ГРАФЫ. МОДЕЛИ ВЫЧИСЛЕНИЙ. СТРУКТУРЫ ДАННЫХ

Учебник

Формат 70×108 1/16. Бумага офсетная. Гарнитура Таймс.
Печать офсетная. Уч.-изд. 21,7. Усл.печ.л. 26,8.
Тираж 300 экз. Заказ

Издательство Нижегородского госуниверситета.
603950, Н. Новгород, пр. Гагарина, 23.

Типография ННГУ. 603000, Н. Новгород, ул. Б. Покровская, 37.
Лицензия ПД №18-0099 от 04.05.2001