

М. Г. Курносов, Д. М. Берлизов

АЛГОРИТМЫ И СТРУКТУРЫ ОБРАБОТКИ ИНФОРМАЦИИ



Новосибирск

2019

УДК 510.5
ББК 22.18я73
К93

Курносов М.Г., Берлизов Д.М.

К93 Алгоритмы и структуры обработки информации. – Новосибирск:
Параллель, 2019. – 211 с.

ISBN 978-5-98901-230-5

В книге рассмотрены фундаментальные структуры и алгоритмы обработки данных. Изложены основы асимптотического анализа вычислительной сложности итеративных и рекурсивных алгоритмов. Приведены классические методы сортировки и поиска информации. Значительная часть материала посвящена подходам к реализации абстрактных типов данных: списков, множеств, ассоциативных массивов и очередей с приоритетами. В конце каждого раздела приведены упражнения.

Книга ориентирована в первую очередь на студентов младших курсов соответствующих специальностей, что нашло отражение в стиле и глубине изложения материала.

ББК 22.18я73

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Утверждено к печати редакционно-издательским советом Сибирского государственного университета телекоммуникаций и информатики.

© Курносов М.Г., 2019.
© Берлизов Д.М., 2019.
© СибГУТИ, 2019.

ISBN 978-5-98901-230-5

 Creative Commons Attribution 4.0.

Содержание

Предисловие	7
1. Алгоритмы и их эффективность	9
1.1. Задача, алгоритм, программа	9
1.2. Показатели эффективности алгоритмов	11
1.3. Подсчет числа операций алгоритма	12
1.4. Скорость роста функций	16
1.5. Асимптотические обозначения	19
1.6. Этапы асимптотического анализа	27
1.7. Упражнения	32
2. Анализ рекурсивных алгоритмов	35
2.1. Рекурсивные алгоритмы	35
2.2. Сортировка слиянием	37
2.3. Решение рекуррентных уравнений	42
2.4. Упражнения	43
3. Сортировка	45
3.1. Задача сортировки	45
3.2. Свойства и виды алгоритмов сортировки	45
3.3. Сортировка вставкой	47
3.4. Сортировка выбором	50
3.5. Нижняя граница времени сортировки сравнением	51
3.6. Быстрая сортировка	53
3.7. Сортировка слиянием	57
3.8. Пирамидальная сортировка	58
3.9. Сортировка подсчетом	58
3.10. Выбор алгоритма сортировки	60
3.11. Упражнения	61
4. Поиск	63
4.1. Задача поиска	63
4.2. Линейный поиск	63
4.3. Бинарный поиск	63
4.4. Упражнения	65

5. Абстрактные типы данных	67
5.1. Базовые типы данных	67
5.2. Структуры данных	68
5.3. Абстрактные типы данных	69
5.4. Выбор абстрактного типа данных	75
5.5. Упражнения	80
6. Списки	83
6.1. АТД список	83
6.2. Реализация списков на базе массивов	84
6.3. Связные списки	89
6.4. Односвязные списки	90
6.5. Двусвязные списки	95
6.6. Реализация АТД список	98
6.7. Упражнения	101
7. Стеки	103
7.1. АТД стек	103
7.2. Реализация стека на базе массива	104
7.3. Реализация стека на базе односвязного списка	106
7.4. Сравнение реализаций	107
7.5. Упражнения	108
8. Очереди	109
8.1. АТД очередь	109
8.2. Реализация очереди на базе связного списка	110
8.3. Реализация очереди на базе кольцевого буфера	112
8.4. Сравнение реализаций	116
8.5. Упражнения	116
9. Бинарные деревья	119
9.1. Корневые деревья	119
9.2. Бинарные деревья	120
9.3. Представление деревьев в памяти	121
9.4. Обходы бинарных деревьев	123
9.5. Упражнения	124
10. Бинарные деревья поиска	125
10.1. Структура бинарного дерева поиска	125
10.2. Создание узла	126
10.3. Добавление узла	126
10.4. Поиск узла по ключу	128
10.5. Поиск минимального и максимального узлов	128
10.6. Поиск следующего и предыдущего узлов	129

10.7. Обход дерева в упорядоченной последовательности	131
10.8. Удаление узла	132
10.9. Удаление дерева	136
10.10. Высота бинарного дерева поиска	137
10.11. Упражнения	138
11. Красно-черные деревья	139
11.1. Структура узла красно-черного дерева	140
11.2. Высота красно-черного дерева	141
11.3. Добавление узла	142
11.4. Удаление узла	148
11.5. Упражнения	152
12. Префиксные деревья	153
12.1. Структура префиксного дерева	154
12.2. Вставка элемента	154
12.3. Поиск элемента	155
12.4. Удаление элемента	156
12.5. Узел префиксного дерева	157
12.6. Представление дерева на основе связных списков	157
12.7. Трудоемкость операций префиксного дерева	158
12.8. Упражнения	159
13. Хеш-таблицы	161
13.1. Таблицы с прямой адресацией	161
13.2. Хеш-таблицы	162
13.3. Метод цепочек	163
13.4. Открытая адресация	167
13.5. Хеш-функции	172
13.6. Преобразование ключей в целые числа	175
13.7. Упражнения	177
14. Бинарные кучи	179
14.1. Представление в памяти	179
14.2. Операции бинарной кучи	180
14.3. Пирамидальная сортировка	187
14.4. Упражнения	188
15. Биномиальные кучи	189
15.1. Структура биномиальной кучи	189
15.2. Поиск минимального элемента	191
15.3. Слияние биномиальных куч	191
15.4. Вставка узла	193
15.5. Удаление минимального узла	195

15.6. Понижение приоритета узла	195
15.7. Упражнения	197
16. Приложения	199
16.1. Функции округления	199
16.2. Прогрессии	200
16.3. Некоторые числовые ряды	200
16.4. Логарифмы	201
16.5. Факториалы	202
16.6. Числа Фибоначчи	202
Предметный указатель	205
Литература	209

Предисловие

Эта книга содержит часть теоретического материала по курсу «Структуры и алгоритмы обработки данных», который авторы читали в разные годы студентам 1 и 2 годов обучения в Сибирском государственном университете телекоммуникаций и информатики (г. Новосибирск). При изложении материала принято во внимание то обстоятельство, что на первом курсе студенты только начинают изучать методы математического анализа и основы теории вероятностей. При доказательстве оценок вычислительной сложности алгоритмов сведено к минимуму использование аппарата теории вероятностей, а там, где он применен, даны комментарии. Основные математические формулы, используемые в книге, вынесены в приложение.

Первые две главы посвящены классическим аспектам анализа временной и пространственной эффективности алгоритмов. В них изложены основные этапы асимптотического анализа вычислительной сложности алгоритмов: от подсчета числа операций до применения асимптотических обозначений. Рассмотрены методы оценки эффективности рекурсивных алгоритмов (анализ дерева рекурсивных вызовов и применение основной теоремы).

В третьей и четвертой главах рассмотрены задачи сортировки и поиска. В них приведены описания тривиальных алгоритмов сортировки с квадратичной сложностью и основные асимптотически оптимальные алгоритмы: сортировка слиянием и пирамидальная сортировка. Разобраны линейный и бинарный алгоритмы решения задачи поиска.

Значительная часть книги (главы 5–9) охватывает вопросы реализации важнейших структур данных. Здесь изложение материала начинается с введения абстрактных типов данных и их классификации. На примере реализации линейных типов данных (список, стек и очередь) рассмотрены основные операции над связными списками, статическими массивами и кольцевыми буферами. В главах 10–13 приведено описание бинарных деревьев поиска и хеш-таблиц как основных средств реализации абстрактных типов данных множество и ассоциативный массив (словарь).

В главе 14 уделено внимание бинарным кучам и реализации на их основе очередей с приоритетами. В главе 15 изложены теоретические основы эффективно сливаемых биномиальных куч.

Описание алгоритмов приводится на естественном языке и дополнено псевдокодом, по синтаксису и семантике близким к императивным языкам Pascal и Algol.

Последняя версия этой книги доступна на сайте

<http://dsabook.mkurnosov.net>

и распространяется по лицензии Creative Commons Attribution 4.0.

Авторы будут благодарны за все конструктивные замечания и отзывы относительно содержания книги, а также найденные ошибки и опечатки. Связаться с ними можно по адресу mkurnosov@gmail.com.

*М.Г. Курносов, Д.М. Берлизов
Новосибирск, март 2019*

1. Алгоритмы и их эффективность

1.1. Задача, алгоритм, программа

Процесс *решения задачи* (problem solving) на компьютере включает в себя нижеследующие этапы.

1. Постановка задачи.
2. Разработка алгоритма решения задачи.
3. Доказательство корректности алгоритма и анализ его эффективности.
4. Реализация алгоритма на языке программирования.
5. Выполнение программы для получения требуемого результата.

В *постановке задачи* (problem statement) приводится точная формулировка условий задачи с описанием ее *входных* (input) и *выходных* (output) данных.

В процессе *разработки алгоритма* (algorithm design) формулируется пошаговая процедура получения из входных данных выходных. Дадим определение понятию алгоритм.

Алгоритм (algorithm) – это конечная последовательность инструкций *исполнителю*, в результате выполнения которых обеспечивается получение из входных данных требуемого выходного результата (решение задачи).

Алгоритм записывается на формальном языке исполнителя, что исключает неоднозначность толкования предназначенных ему предписаний. Запись алгоритма на формальном языке исполнителя называется *программой* (program, рис. 1.1).

Говорят, что алгоритм *корректен* (correct), если он для любых корректных значений входных данных выдает корректные выходные данные. Здесь под корректностью данных понимается их соответствие условиям решаемой задачи.

После того как разработан алгоритм решения задачи осуществляется его *реализация* (implementation) на одном из языков программирования, например: C, C++, Java, C#, Go, Python, JavaScript и др. Язык программирования выступает в роли языка исполнителя, а исполнителем является интерпретатор или процессор, в набор инструкций которого компилятор

транслирует программу.

В ходе разработки программы алгоритм может претерпевать изменения, связанные с учетом архитектуры целевой системы. Например, в алгоритм могут вноситься изменения, обеспечивающие эффективное использование кеш-памяти процессора, и др.

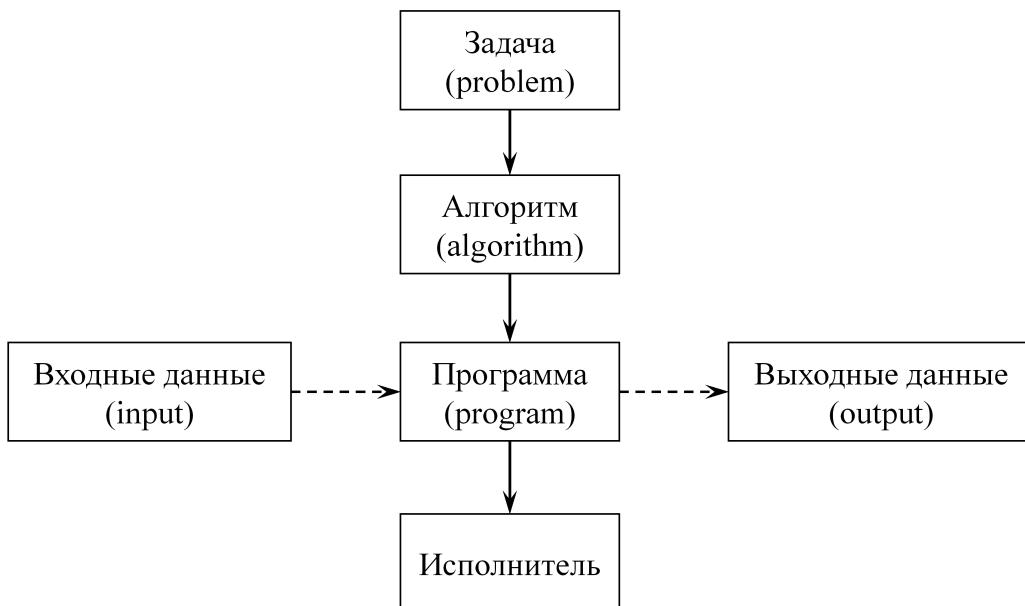


Рис. 1.1. Процесс решения задачи: задача, алгоритм, программа.

Рассмотрим основные свойства, которыми должен обладать алгоритм.

1. *Дискретность* – алгоритм представляется как последовательность инструкций исполнителя. Каждая инструкция выполняется только после того, как закончилось выполнение предыдущей команды.
2. *Конечность* (результативность, финитность) – алгоритм должен заканчиваться после выполнения конечного числа инструкций.
3. *Детерминированность* – каждый шаг алгоритма должен быть однозначно определен – записан на формальном языке исполнителя. Детерминированность обеспечивает совпадение результатов, получаемых при многократном выполнении алгоритма, на одном и том же наборе входных данных.
4. *Массовость* – алгоритм решения задачи должен быть применим для некоторого класса задач, различающихся лишь значениями входных данных.

Пример. Поиск максимального элемента массива. Рассмотрим задачу поиска в массиве номера элемента с максимальным значением. Формальная постановка задачи выглядит следующим образом:

- **вход:** последовательность из n чисел (a_1, a_2, \dots, a_n) ;
- **выход:** номер i элемента a_i , имеющего наибольшее значение:

$$a_i \geq a_j, \quad \forall j \in \{1, 2, \dots, n\}. \quad (1.1)$$

Если на вход подается последовательность (14, 20, 34, 6, 85, 232, 177), то решением задачи является номер 6 (элемент со значением 232). Такой конкретный набор входных данных называется *экземпляром задачи* (problem instance).

Ниже приведен псевдокод линейного алгоритма Max решения рассматриваемой задачи о поиске номера максимального элемента. На вход алгоритма поступает массив $a[1..n]$ из n элементов.

Алгоритм 1.1. Поиск максимального элемента массива

```

1 function Max( $a[1..n]$ )
2      $maxi = 1$ 
3     for  $i = 2$  to  $n$  do
4         if  $a[i] > a[maxi]$  then
5              $maxi = i$ 
6         end if
7     end for
8     return  $maxi$ 
9 end function
```

Алгоритм Max является *дискретным*, так как записан на диалекте императивного языка программирования Алгол-60. *Конечность* алгоритма обеспечивается, тем, что он всегда заканчивает свою работу после проштотра $n - 1$ элементов массива a . В процессе своей работы алгоритм не использует датчик псевдослучайных чисел для принятия решений о ходе дальнейшего выполнения, это обеспечивает его *детерминированность*. *Массовость* алгоритма обусловлена тем, что он решает задачу поиска номера максимального элемента для произвольного массива $a[1..n]$.

1.2. Показатели эффективности алгоритмов

Алгоритмы, разработанные для решения одной и той же задачи, могут значительно различаться по эффективности. Поэтому для характеристики их качества вводят показатели эффективности. Рассмотрим наиболее распространенные из них.

1. Количество операций – *временная эффективность* (time efficiency), показывает насколько быстро работает алгоритм.
2. Объем потребляемой памяти – *пространственная эффективность* (space efficiency), отражает максимальное количество памяти, требуемой для выполнения алгоритма.

Существуют и другие показатели, которые имеет смысл рассматривать, если они в значительной степени влияют на процесс решения задачи. Например, для алгоритмов, работающих с данными на внешних носителях информации (жесткие диски, сетевые хранилища), целесообразно учитывать количество обращений к внешней памяти, а в алгоритмах, использующих сетевые каналы связи, важно принимать во внимание количество переданных сообщений (сетевых пакетов).

Введение показателей эффективности позволяет проводить анализ алгоритмов с целью сравнения их между собой и оценивания потребности того или иного алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети.

1.3. Подсчет числа операций алгоритма

Для большинства алгоритмов количество выполняемых ими операций напрямую зависит от размера входных данных. Чем больше входных данных, тем дольше работает алгоритм. Так, время выполнения алгоритма Мах поиска максимального элемента (алгоритм 1.1) зависит от длины n массива.

Количество операций алгоритма можно выразить как функцию от одного или нескольких параметров, связанных с размером входных данных. Рассмотрим несколько примеров:

- алгоритм сортировки выбором – количество операций алгоритма (время его работы) зависит от числа n элементов в массиве;
- алгоритм умножения двух матриц – время выполнения зависит от количества строк m и столбцов n в матрицах;
- алгоритм Дейкстры поиска кратчайшего пути в графе – время выполнения зависит от числа n вершин и m ребер в графе.

RAM-машина. Для подсчета числа операций, выполняемых алгоритмом, необходимо формально описать систему команд некоторого исполнителя. В качестве такого исполнителя будем использовать модель однопроцессорной вычислительной машины с произвольным доступом к памяти (Random Access Machine — RAM) [1, 2]. Условимся, что машина обладает неограниченной памятью и функционирует по следующим правилам:

- для выполнения арифметических и логических операций ($+, -, *, /, \%$) требуется один временной шаг (такт процессора);
- каждое обращение к ячейке в оперативной памяти для чтения или записи занимает один временной шаг;
- выполнение условного перехода (*if-then-else*) требует вычисления логического выражения и выполнения одной из ветвей *if-then-else*;
- выполнение цикла (*for*, *while*, *do*) подразумевает выполнение всех его

итераций, в свою очередь, выполнение каждой итерации требует вычисления условия завершения цикла и выполнение его тела.

Пример. Суммирование элементов массива. Вычислим количество операций алгоритма SUMARRAY, реализующего вычисление суммы n элементов массива.

Алгоритм 1.2. Суммирование элементов массива

```

1 function SUMARRAY( $a[1..n]$ )
2    $sum = 0$ 
3   for  $i = 1$  to  $n$  do
4      $sum = sum + a[i]$ 
5   end for
6   return  $sum$ 
7 end function
```

Время работы алгоритма SUMARRAY зависит только от размера n массива. В строке 2 выполняется одна операция записи в память. Далее, перед выполнением каждой из n итераций цикла происходят проверка условия его окончания $i = n$ и переход на строку 4 или 6. На каждой итерации в строке 4 выполняется четыре операции: чтение из памяти значений sum и $a[i]$, их сложение и запись результата в память. В конце алгоритма выполняется возврат результирующего значения – одна операция. Таким образом, количество операций $T(n)$, выполняемых алгоритмом SUMARRAY, есть

$$T(n) = 4n + 2.$$

Далее мы убедимся, что такой точный анализ числа операций алгоритма во многих случаях не требуется. Достаточно ограничиться подсчетом лишь тех операций, суммарное количество которых зависит от размера входных данных. Так, в алгоритме SUMARRAY строки 2 и 6 не имеют значимого влияния на итоговое время выполнения, которое фактически определяется только операциями в строке 4.

При анализе вычислительной сложности алгоритмов мы будем игнорировать операции, связанные с проверкой условия окончания цикла *for* и автоматическим увеличением его счетчика.

Пример. Линейный поиск. Существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но и от их значений. В качестве примера рассмотрим алгоритм LINEARSEARCH линейного поиска заданного значения x в массиве из n элементов.

Количество операций алгоритма LINEARSEARCH может существенно различаться для одного и того же размера n входных данных. Рассмотрим три возможных случая.

Алгоритм 1.3. Линейный поиск

```

1 function LINEARSEARCH( $a[1..n]$ ,  $x$ )
2   for  $i = 1$  to  $n$  do
3     if  $a[i] = x$  then
4       return  $i$ 
5     end if
6   end for
7   return  $-1$ 
8 end function

```

Лучший случай (best case) – экземпляр задачи (набор входных данных), на котором алгоритм выполняет *наименьшее* число операций. В нашем примере – входной массив, первый элемент которого содержит искомое значение x . В этой ситуации требуется выполнить

$$T_{Best}(n) = 3$$

операции: проверка условия окончания цикла, условие в цикле (строка 3) и возврат найденного значения (строка 4). Таким образом, время работы алгоритма в лучшем случае – теоретическая *нижняя граница* времени его работы.

Худший случай (worst case) – экземпляр задачи, на котором алгоритм выполняет наибольшее число операций. Для рассматриваемого алгоритма – массив, в котором отсутствует искомый элемент или он расположен в последней ячейке. В этой ситуации требуется выполнить

$$T_{Worst}(n) = 2n + 1$$

операций: n раз проверить условие окончания цикла и условие в нем, затем вернуть значение -1 . Время работы алгоритма в худшем случае – теоретическая *верхняя граница* времени его работы.

Средний случай (average case) – «средний» экземпляр задачи, набор «усредненных» входных данных. В среднем случае оценивается математическое ожидание количества операций, выполняемых алгоритмом. Стоит заметить, что не всегда очевидно, какие входные данные считать «усредненными» для задачи. Часто делается предположение, что все наборы данных поступают на вход алгоритма с одинаковой вероятностью.

Вернемся к нашему примеру и проведем анализ его эффективности для среднего случая. Обозначим вероятность успешного поиска элемента в массиве через $p \in [0, 1]$. Тогда вероятность отсутствия значения x в массиве равна $1 - p$. Будем считать, что искомый элемент с одинаковой вероятностью p/n может находиться в любой из n ячеек массива.

Если искомый элемент x находится в ячейке 1, то для его поиска тре-

буется выполнить 3 операции (проверить условие окончания цикла, условие в цикле и вернуть значение 1); если элемент находится в ячейке 2, то требуется 5 операций, и т.д. В общем случае, если искомый элемент x расположен в ячейке i , то это требует выполнения $2i + 1$ операций.

Запишем математическое ожидание (среднее значение) числа операций, выполняемых алгоритмом. По определению, математическое ожидание есть сумма произведений значения дискретной случайной величины на вероятность принятия случайной величиной этого значения. Наша случайная величина – число операций, выполняемых алгоритмом. Как мы условились выше, она может принимать с одинаковой вероятностью p/n следующие значения: 3, 5, …, $(2i + 1)$, …, $2n + 1$. Поэтому

$$T_{Average}(n) = 3\frac{p}{n} + 5\frac{p}{n} + \cdots + (2i + 1)\frac{p}{n} + \cdots + (2n + 1)\frac{p}{n}.$$

В нашей оценке мы должны учесть тот факт, что искомое значение x с вероятностью $1 - p$ может отсутствовать в массиве. Тогда выражение примет следующий вид:

$$T_{Average}(n) = \frac{p}{n} [3 + 5 + \cdots + (2i + 1) + \cdots + (2n + 1)] + (1 - p)(2n + 1).$$

Нетрудно заметить, что в квадратных скобках записана сумма членов арифметической прогрессии (см. приложение). Вычислим ее:

$$\begin{aligned} T_{Average}(n) &= \frac{p}{n} [n^2 + 2n] + (1 - p)(2n + 1) = \\ &= p(n + 2) + (1 - p)(2n + 1). \end{aligned}$$

Из полученной формулы можно сделать следующий вывод: если искомый элемент присутствует в массиве ($p = 1$), то в среднем требуется выполнить $n + 2$ операции для его нахождения.

Анализ эффективности алгоритмов для среднего случая – более трудная задача, чем анализ их поведения в худшем и лучшем случаях. Однако для многих алгоритмов именно оценки эффективности для среднего случая дают реальную картину относительно их практической применимости.

Далее при анализе алгоритмов мы будем уделять основное внимание времени работы алгоритмов в худшем случае – максимальному времени работы на всех наборах входных данных и, по возможности, строить оценки эффективности для среднего случая.

В качестве синонимов понятию *количество операций* алгоритма мы будем использовать термины *время выполнения* алгоритма (execution time) и *вычислительная сложность* алгоритма (computational complexity).

1.4. Скорость роста функций

Пусть, мы имеем два алгоритма решения одной и той же задачи. В соответствии с описанной выше схемой построены функции $T_1(n)$ и $T_2(n)$ зависимости числа операций алгоритмов от размера их входных данных для лучшего, среднего либо худшего случая. Определимся, что

$$T_1(n) = 90n^2 + 201n + 2000,$$

$$T_2(n) = 2n^3 + 3.$$

Возникает вопрос: какой из алгоритмов предпочтительнее использовать на практике?

Время выполнения обоих алгоритмов возрастет с увеличением размера n входных данных. На рис. 1.2 приведены графики функций $T_1(n)$ и $T_2(n)$, а в табл. 1.1 – их значения. На данных небольшого размера второй алгоритм выполняет меньше операций чем первый, следовательно, в этом случае предпочтение надо отдать второму алгоритму. На больших значениях n второй алгоритм начинает заметно уступать первому, здесь стоит выбрать первый алгоритм.

В общем случае мы можем найти такое значение n_0 , при котором происходит пересечение функций $T_1(n)$ и $T_2(n)$, и, зная конкретный размер n входных данных и n_0 , отдавать предпочтение тому или иному алгоритму.

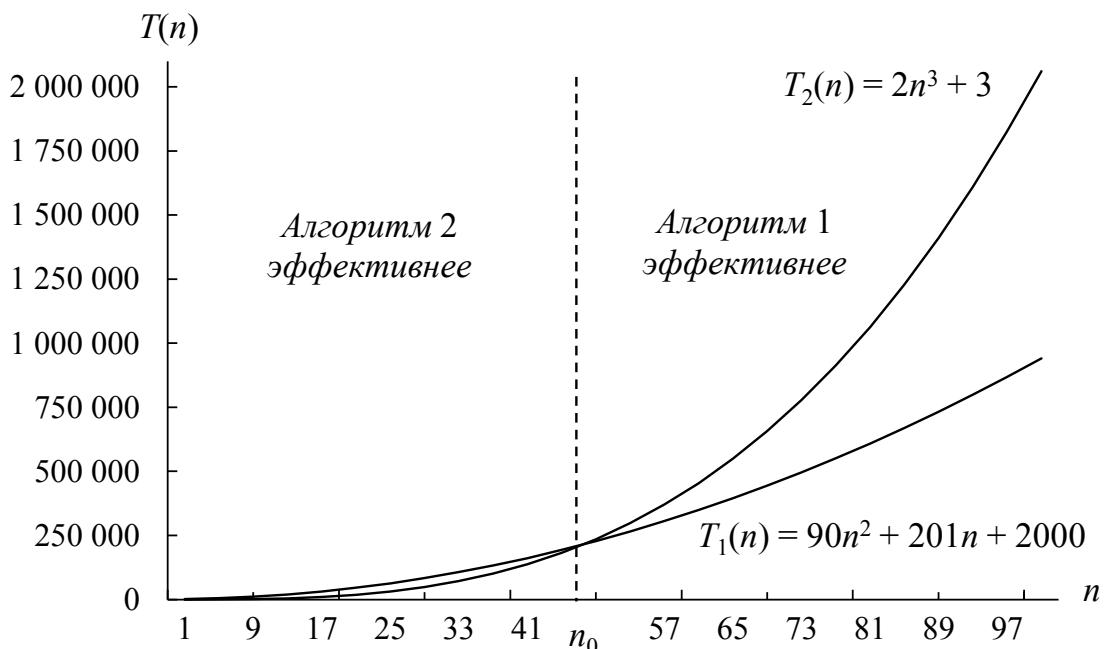


Рис. 1.2. Зависимость числа операций, выполняемых алгоритмами, от размера n входных данных.

В большинстве случаев при малых размерах n входных данных разница во времени выполнения алгоритмов, решающих одну и ту же задачу, как правило, незначительна. Если априори известно, что на вход будут

Таблица 1.1. Количество операций, выполняемых алгоритмами

n	$T_1(n)$	$T_2(n)$	Сравнение алгоритмов
10	13 010	2 003	Первый алгоритм в 6.5 раза медленнее второго: $T_1(n)/T_2(n) = 6.5$
20	42 020	16 003	Первый в 2.6 раза медленнее второго
30	89 030	54 003	Первый в 1.6 раза медленнее второго
40	154 040	128 003	Первый в 1.2 раза медленнее второго
50	237 050	250 003	Первый в 1.1 раза быстрее второго
60	338 060	432 003	Первый в 1.3 раза быстрее второго
70	457 070	686 003	Первый в 1.5 раза быстрее второго
80	594 080	1 024 003	Первый в 1.7 раза быстрее второго
90	749 090	1 458 003	Первый в 1.9 раза быстрее второго
100	922 100	2 000 003	Первый в 2.2 раза быстрее второго
1 000	9 0203 000	2 000 000 003	Первый в 22 раза быстрее второго
10 000	9 002 012 000	2 000 000 000 003	Первый в 222 раз быстрее второго

поступать данные небольших размеров, то вопрос о выборе эффективного алгоритма не является первостепенным: можно использовать самый «простой» алгоритм – понятный, легко реализуемый на языке программирования или доступный в стандартной библиотеке языка. *Даже неэффективный алгоритм при работе с входными данными небольшого размера завершается за допустимое время.*

Вопросы, связанные с пространственной и времененной эффективностью алгоритмов, приобретают смысл при больших размерах входных данных. В том числе, когда заранее не известно, какого размера экземпляры задач нам предстоит решать. Следовательно, в первую очередь нас будет интересовать вопрос о том, как быстро растет число операций $T(n)$, выполняемых алгоритмом, при увеличении размера n входных данных.

Для сравнения функций по тому, насколько быстро они изменяют свои значения с увеличением значений их аргументов, вводят понятие *скорости роста функций*.

Скорость роста (rate of growth) или *порядок роста* (order of growth) функции $T(n)$ определяются ее старшим, доминирующим членом. При больших значениях n членами меньшего порядка можно пренебречь. В нашем примере функция $T_1(n) = 90n^2 + 201n + 2000$ растет как n^2 , а функ-

ция $T_2(n) = 2n^3 + 3$ как n^3 . Порядок роста функции n^3 больше порядка роста функции n^2 , так как n^3 принимает большие значения, чем n^2 .

Один алгоритм рассматривается как более эффективный по сравнению с другим (в худшем, среднем либо лучшем случае), если число его операций имеет более низкий порядок роста.

В табл. 1.2 показана скорость роста некоторых функций, которые часто встречаются в оценках времени работы алгоритмов. Медленнее всего растут функции $\lg n$ и $\log_2 n$. Алгоритмы с подобной логарифмической зависимостью числа операций от размера входных данных работают практически мгновенно. Например, бинарный поиск элемента в упорядоченном массиве или поиск узла в АВЛ-дереве или красно-черном дереве. Быстрее всех (из приведенных в табл. 1.2) растут функции 2^n и $n!$. Алгоритмы с такой сложностью на практике мало применимы, так как даже на небольших размерах входных данных требуют выполнения колоссального числа операций. Примерами могут служить некоторые оптимизационные алгоритмы, реализующие полный перебор множества допустимых решений задачи. Если решение представляется в виде перестановки из n чисел, то количество перебираемых алгоритмом вариантов равно числу перестановок порядка n , а именно $n!$.

Таблица 1.2. Скорость роста функций

$\lg n$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	$n!$
0	0	1	0	1	2	1
0.3	1	2	2	4	4	2
0.5	1.6	3	5	9	8	6
0.6	2.0	4	8	16	16	24
0.7	2.3	5	12	25	32	120
0.78	2.6	6	16	36	64	720
0.85	2.8	7	20	49	128	5 040
0.90	3	8	24	64	256	40 320
0.95	3.2	9	29	81	512	362 880
1	3.3	10	33	100	1024	3 628 800
3	10	1 000	9 966	1 000 000		
4	13.3	10 000	132 877	100 000 000		
5	16.6	100 000	1 660 964	10 000 000 000		
6	19.9	1 000 000	19 931 569	1 000 000 000 000		

Для сравнения и классификации скоростей роста функций, выраженных время работы алгоритмов, нам необходим специальный математиче-

ский аппарат. Он должен давать возможность делать выводы о том, что два алгоритма при больших n работают с одинаковым временем или один алгоритм эффективнее другого. Ответы на эти вопросы дает *асимптотический анализ* (asymptotic analysis), который позволяет оценивать скорость роста функций $T(n)$ при стремлении размера входных данных к бесконечности (при $n \rightarrow \infty$).

1.5. Асимптотические обозначения

Как было показано ранее, время выполнения алгоритма в худшем, среднем и лучшем случаях можно представить, как функцию $T(n)$ от размера его входных данных. Однако анализировать эффективность алгоритмов по таким функциям достаточно трудно по ряду причин. Как правило, функция $T(n)$ времени выполнения алгоритма имеет большое количество локальных экстремумов – неровный график с выпукенностями и впадинами (рис. 1.3, а). Например, возможна ситуация, когда время выполнения алгоритма на входных массивах с нечетной длиной будет больше времени выполнения на массивах с четной длиной. В этом случае график $T(n)$ будет иметь пилообразный вид, как на рис. 1.3, б. Поэтому намного проще работать с верхней и нижней границами (оценками) времени выполнения алгоритма. Так для примера на рис. 1.3, б вместо пилообразной функции $T(n)$ можно использовать ее верхнюю границу $5n + 15$ или другую.

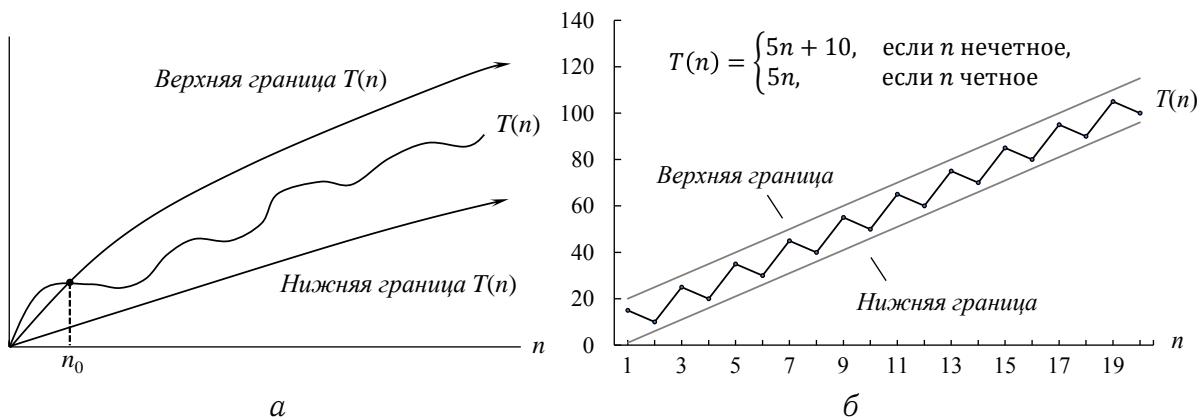


Рис. 1.3. Пример возможных верхней и нижней границ времени $T(n)$ выполнения алгоритмов.

Для указания границ функций $T(n)$ в теории *вычислительной сложности алгоритмов* (computational complexity theory) используют асимптотические обозначения: O (о большое), Ω (омега большое), Θ (тета большое), а также o (о малое) и ω (омега малое). Далее будем считать, что областью определения функций $f(n)$ и $g(n)$, которые выражают число операций алгоритма, является множество неотрицательных целых чисел

($n \in \{0, 1, 2, \dots\}$). Функции $f(n)$ и $g(n)$ являются *асимптотически неотрицательными* – при больших значениях n они принимают значения, большие или равные нулю. Применительно к анализу сложности алгоритмов последнее требование выполняется всегда, так функции выражают число операций алгоритма или объем потребляемой им памяти, которые не могут быть отрицательными.

1.5.1. O -обозначение

O -обозначение используют, если необходимо указать *асимптотическую верхнюю границу* (asymptotic upper bound) для функции $f(n)$, числа операций алгоритма. Говорят, что функция $f(n)$ принадлежит множеству функций $O(g(n))$, что записывается как $f(n) \in O(g(n))$, если существуют положительная константа $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq f(n) \leq cg(n).$$

Формально множество $O(g(n))$ определяется следующим образом:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.2)$$

Иначе говоря, $O(g(n))$ – это множество всех функций, значения которых при больших n *не превышают* значение $cg(n)$. Обычно факт принадлежности функции $f(n)$ множеству $O(g(n))$ записывают как $f(n) = O(g(n))$. Читается как « f от n есть о большое от g от n ».

Для доказательства, $f(n) = O(g(n))$, требуется найти константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.2). Если доказано, что $f(n) = O(g(n))$, то говорят: «функция f асимптотически ограничена сверху функцией g с точностью до постоянного множителя». Таким образом, произведение $cg(n)$ является *асимптотической оценкой сверху* времени $f(n)$ работы алгоритма.

Пример. Докажем, что $2n - 10 = O(n)$. Для этого требуется найти соответствующие константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$. Доказательство: возьмем $c = 2$, $n_0 = 5$. Эти значения обеспечивают выполнение неравенства $0 \leq 2n - 10 \leq 2n$ для любых $n \geq 5$. На рис. 1.4 видно, что прямая $2n$ проходит выше прямой $2n - 10$. Можно было взять и другие значения c и n_0 , главное – это то, что мы показали их существование.

Аналогично можно доказать корректность следующих утверждений.

1. $3n^2 + 100n + 8 = O(n^2)$. Для доказательства возьмем $c = 4$, $n_0 = 101$, при любых $n \geq 101$ справедливо неравенство $0 \leq 3n^2 + 100n + 8 \leq 4n^2$.
2. $3n^2 + 100n + 8 = O(n^3)$. Возьмем $c = 1$, $n_0 = 12$, для любых $n \geq 12$ справедливо $0 \leq 3n^2 + 100n + 8 \leq n^3$.

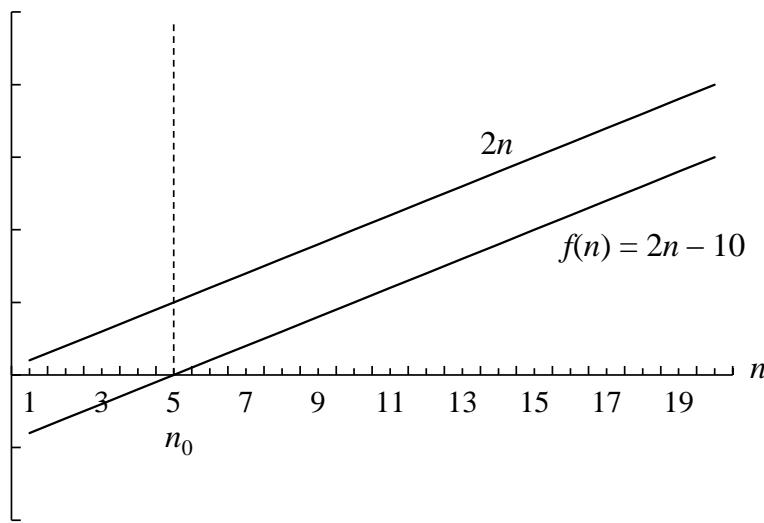


Рис. 1.4. Иллюстрация принадлежности функции $2n - 10$ множеству $O(n)$.

3. $0.000001n^3 \neq O(n^2)$, так как не существует констант $c > 0$ и n_0 , которые обеспечивают выполнение неравенств (1.2). Для любых $c > 0$ и $n \geq c/0.000001$ имеет место неравенство $0.000001n^3 > cn^2$.

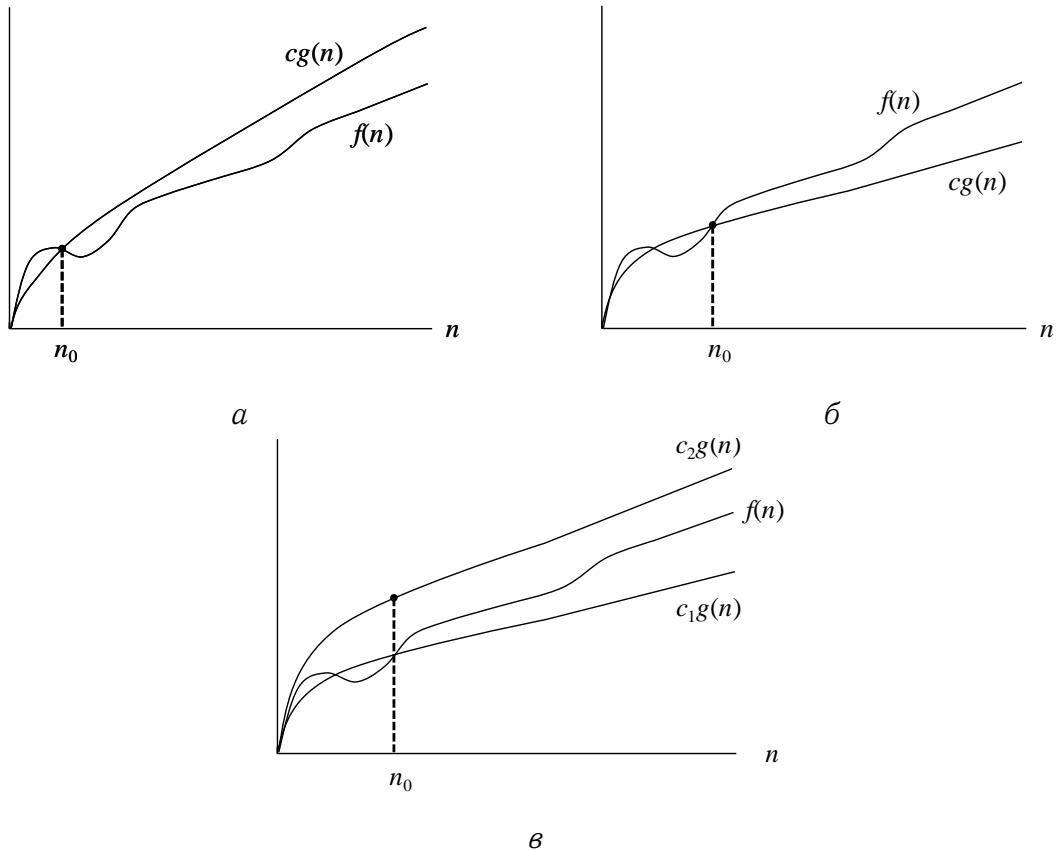


Рис. 1.5. Иллюстрация асимптотических обозначений:
 $\alpha - f(n) = O(g(n)); \beta - f(n) = \Omega(g(n)); \gamma - f(n) = \Theta(g(n)).$

На рис. 1.5 приведены иллюстрации основных асимптотических обозначений.

1.5.2. Ω -обозначение

Ω -обозначение используется для записи *асимптотической нижней границы* (asymptotic lower bound) для функции $f(n)$. Говорят, функция $f(n)$ принадлежит множеству $\Omega(g(n))$ (записывается как $f(n) \in \Omega(g(n))$), если существуют константа $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq cg(n) \leq f(n).$$

Читается как « f от n есть омега большое от g от n ». Формально множество $\Omega(g(n))$ определяется следующим образом

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.3)$$

Другими словами, $\Omega(g(n))$ – это множество всех функций, значения которых при больших n не меньше значения $cg(n)$. Для доказательства принадлежности функции $f(n)$ множеству $\Omega(g(n))$ требуется найти константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.3). Если доказано, что $f(n) = \Omega(g(n))$, то говорят: «функция f асимптотически ограничена снизу функцией g с точностью до постоянного множителя». Таким образом, произведение $cg(n)$ является *асимптотической оценкой снизу* времени $f(n)$ работы алгоритма.

Пример. Докажем справедливость отношения $n^3 + 5 = \Omega(n^2)$. Следуя определению, нам необходимо показать существование констант $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, при которых $n^2 \leq 4n^3 + 5$ для любых $n \geq n_0$. Для доказательства достаточно взять $c = 1$ и $n_0 = 0$.

1.5.3. Θ -обозначение

Θ -обозначение позволяет записать *асимптотически точную оценку* (asymptotic tight bound) для функции $f(n)$. Функция $f(n)$ принадлежит множеству функций $\Theta(g(n))$, записывается как $f(n) \in \Theta(g(n))$, если существуют положительные константы $c_1 > 0, c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Читается как « f от n есть тета большое от g от n ». Множество $\Theta(g(n))$ определяется следующим образом

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c_1 > 0, c_2 > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.4)$$

Для доказательства того, что $f(n) = \Theta(g(n))$, требуется найти константы $c_1 > 0$, $c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.4). Если доказано, что $f(n) = \Theta(g(n))$, то говорят: «функция f асимптотически ограничена снизу и сверху функцией g с точностью до постоянного множителя».

Обозначение Θ более сильное, чем O и Ω . Используя обозначения теории множеств, можно записать $\Theta(g(n)) \subseteq O(g(n))$.

Утверждение. Для любых двух функций $f(n)$ и $g(n)$ отношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Из этого утверждения следует: если известно, что $f(n) = \Theta(g(n))$, то это гарантирует $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. С другой стороны, если мы покажем, что для некоторой функции $f(n)$ одновременно выполняются $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, то тем самым мы докажем справедливость $f(n) = \Theta(g(n))$. Рассмотренное утверждение дает нам двухшаговую процедуру для построения асимптотически точных оценок для функций.

Пример. Докажем корректность утверждения $2n^2 + 3n = \Theta(n^2)$. По определению требуется найти константы $c_1 > 0$, $c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств

$$0 \leq c_1 n^2 \leq 2n^2 + 3n \leq c_2 n^2.$$

Сперва покажем, что наша функция ограничена сверху. Пусть $c_2 = 3$. Неравенство $2n^2 + 3n \leq 3n^2$ выполняется для любых $n \geq 3$ – это точка пересечения функций $2n^2 + 3n$ и $3n^2$. Таким образом $n_0 = 3$ и $2n^2 + 3n = O(n^2)$. Осталось показать выполнение неравенств $0 \leq c_1 n^2 \leq 2n^2 + 3n$. Они справедливы при $c_1 = 2$ и $n_0 = 0$, следовательно, $2n^2 + 3n = \Omega(n^2)$. Далее, нам нужно выбрать n_0 , при котором выполняются как условия ограничения функции снизу, так и ограничения сверху. Поэтому выберем n_0 как максимальное значение из найденных n_0 , это обеспечит выполнение обоих условий: $n_0 = \max(0, 3) = 3$. Мы нашли константы $c_1 = 2$, $c_2 = 3$ и $n_0 = 3$, тем самым доказали справедливость исходного утверждения $2n^2 + 3n = \Theta(n^2)$.

1.5.4. o -обозначение

Обозначение o используют для указания того, что верхняя граница функции не является асимптотически точной. Например, запись

$$100n = O(n^2)$$

содержит в себе корректную асимптотическую верхнюю границу функции $100n$, но не обеспечивает нас верной информацией об асимптотически точ-

ной верхней границе функции. Дадим определение o -обозначения:

$$o(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c > 0, \exists n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n) < cg(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.5)$$

Таким образом $o(g(n))$ – это множество всех функций, значения которых при больших n *меньше* значения $c g(n)$ для *любых* c . Факт принадлежности функции $f(n)$ множеству $o(g(n))$ записывают как $f(n) = o(g(n))$. Читается как « f от n есть о малое от g от n ». Важным отличием от O -обозначения является ограничение сверху функции $f(n)$ значением $c g(n)$ для *любых* (*всех*) положительных констант c . Например, $100n = o(n^2)$, в то же время $4n^2 + 2n \neq o(n^2)$.

1.5.5. ω -обозначение

Обозначение ω подобно обозначению Ω . Дадим ему формальное определение:

$$\omega(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c > 0, \exists n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq cg(n) < f(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.6)$$

Иначе говоря, $\omega(g(n))$ – множество всех функций, значения которых при больших n *больше* значения $c g(n)$ для *любых* c . Факт принадлежности функции $f(n)$ множеству $\omega(g(n))$ записывают как $f(n) = \omega(g(n))$. Читается как « f от n есть омега малое от g от n ». Важным отличием от Ω -обозначения является ограничение снизу функции $f(n)$ значением $c g(n)$ для любых положительных констант c . Например, справедливо отношение $n^2 = \omega(n)$, но $24n^2 \neq \omega(n^2)$.

1.5.6. Случай функции нескольких переменных

Асимптотические обозначения можно обобщить и на случай функций нескольких переменных. Например, для функции двух переменных будем иметь следующее определение O -обозначения:

$$O(g(n, m)) = \left\{ f(n, m) : \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, m_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n, m) \leq cg(n, m), \quad \forall n \geq n_0, \forall m \geq m_0. \end{array} \right\}$$

Аналогично можно ввести определения остальных асимптотических обозначений для функций нескольких переменных.

1.5.7. Сравнение порядка роста функций через предел

Пусть, как и ранее, мы имеем две функции $f(n)$ и $g(n)$. Для ответа на вопрос, какая из функций имеет более высокий порядок роста, достаточно вычислить передел их отношения при $n \rightarrow \infty$ [2, 3]

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & \text{если } f(n) = o(g(n)), \\ c, & \text{если } f(n) = \Theta(g(n)), \\ \infty, & \text{если } f(n) = \omega(g(n)). \end{cases} \quad (1.7)$$

Если предел равен 0, то $f(n) = o(g(n))$ и $f(n) = O(g(n))$. Второй случай: $f(n) = \Theta(g(n))$, следовательно, $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. Из третьего случая следует $f(n) = \omega(g(n))$ и $f(n) = \Omega(g(n))$.

Пример. Сравним порядки (скорости) роста функций $f(n) = n(n-1)/2$ и $g(n) = n^2$.

$$\lim_{n \rightarrow \infty} \frac{n(n-1)/2}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Предел равен положительной константе, тогда по (1.7) обе функции имеют одинаковый порядок роста $n(n-1)/2 = \Theta(n^2)$.

Пример. Сравним порядки роста функции $f(n) = 2n + 1$ и функции $g(n) = \log_a n$ при $a > 1$.

$$\lim_{n \rightarrow \infty} \frac{2n + 1}{\log_a n} = \lim_{n \rightarrow \infty} \frac{2}{\frac{1}{n \ln a}} = \lim_{n \rightarrow \infty} 2n \ln a = \infty.$$

Для вычисления последнего предела использовано *правило Лопиталя* – перешли к рассмотрению предела отношения производных функций $f(n)$ и $g(n)$. Предел равен бесконечности. Это означает, что функция $f(n)$ имеет больший порядок роста, чем $g(n)$. Иначе говоря, $2n+1 = \omega(\log n)$. Обратите внимание, что здесь намеренно опущено основание a логарифма, так как при использовании совместно с асимптотическими обозначениями оно не имеет смысла. По свойствам логарифмов изменение основания приводит к умножению логарифма на константу, а по свойствам асимптотических обозначений константы могут быть отброшены:

$$\log_a n = \frac{1}{\log_c a} \log_c n.$$

1.5.8. Свойства асимптотических отношений

Факт принадлежности функции $f(n)$ некоторому множеству мы записывали через знак равенства, например, $f(n) = O(g(n))$. Это обозначение удобно на практике, но нужно помнить, что смысл его в том, что функция,

стоящая слева от знака равенства, принадлежит множеству, указанному справа.

$$f(n) \in O(g(n)).$$

Говоря более точно, это не равенство в обычном понимании, а несимметричное *бинарное отношение* между функциями $f(n)$ и $g(n)$. Например, запись $f(n) = O(g(n))$ является корректной, в то же время выражение $O(g(n)) = f(n)$ лишено смысла.

Если асимптотическое обозначение O , Ω или Θ присутствует в формуле, то вместо него можно мысленно подставить любую функцию из этого множества. Например, в выражении $3n^2 + 5n + 34 = 3n^2 + \Theta(n)$, вместо $\Theta(n)$ можно подставить любую функцию с линейным порядком роста.

Асимптотические бинарные отношения O , Ω и Θ обладают многими свойствами бинарных отношений между действительными числами (для определенности обозначим числа через f и g):

- $f(n) = O(g(n))$ соответствует $f \leq g$;
- $f(n) = \Omega(g(n))$ соответствует $f \geq g$;
- $f(n) = \Theta(g(n))$ соответствует $f = g$;
- $f(n) = o(g(n))$ соответствует $f < g$;
- $f(n) = \omega(g(n))$ соответствует $f > g$.

Транзитивность

- Если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, то $f(n) = O(h(n))$;
- Если $f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$, то $f(n) = \Omega(h(n))$;
- Если $f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$, то $f(n) = \Theta(h(n))$;
- Если $f(n) = o(g(n))$ и $g(n) = o(h(n))$, то $f(n) = o(h(n))$;
- Если $f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$, то $f(n) = \omega(h(n))$.

Рефлексивность

- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$;
- $f(n) = \Theta(f(n))$.

Симметричность

- $f(n) = \Theta(g(n))$ тогда и только тогда, когда $g(n) = \Theta(f(n))$.

Перестановочная симметрия

- $f(n) = O(g(n))$ тогда и только тогда, когда $g(n) = \Omega(f(n))$;
- $f(n) = o(g(n))$ тогда и только тогда, когда $g(n) = \omega(f(n))$.

Рассмотрим свойства O -обозначения, которые удобно использовать на практике:

- **произведение** – если $f_1 = O(g_1)$ и $f_2 = O(g_2)$, то $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$;

- **сложение** – если $f_1 = O(g_1)$ и $f_2 = O(g_2)$, то $f_1 + f_2 \in O(|g_1| + |g_2|)$;
- **умножение на константу** – если $k > 0$ и $f = O(g)$, то $kf \in O(g)$.

При использовании асимптотических обозначений константы в функции $T(n)$ игнорируются.

Из свойств следует: для любого полинома $p(n)$ степени $k \in \{0, 1, 2, \dots\}$ при $a_k > 0$ справедливо

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = \Theta(n^k).$$

1.6. Этапы асимптотического анализа

Асимптотический анализ сложности алгоритма заключается в построении асимптотических оценок его вычислительной сложности (computational complexity) или объема требуемой ему памяти (space complexity) в худшем, среднем или лучшем случае. Рассмотрим основные шаги асимптотического анализа.

1. Определяются параметры, от которых зависит время выполнения алгоритма или объем требуемой ему памяти: устанавливается, сколько аргументов будет у функции $T(n)$. Например, $T(n)$, $T(n, m)$, $T(n, m, q)$.

2. Для худшего, среднего или лучшего случая вычисляется количество операций алгоритма или объем требуемой ему памяти, что записывается как функция от параметров, установленных на предыдущем шаге. Например, $T(n) = 3n^2 + 32n + 5$. В первую очередь вычисляют число операций алгоритма для худшего случая. Если на практике возникновение этого случая маловероятно, то подсчитывается число операций алгоритма для среднего случая. На данном этапе можно ограничиться подсчетом числа только *базовых операций* (basic operations) – наиболее важных операций, от которых зависит время выполнения алгоритма. Например, можно не учитывать константное число операций чтения значений переменных из памяти и операций присваивания.

3. К построенной функции $T(n)$ применяются асимптотические обозначения для классификации порядка ее роста. При помощи Θ -обозначения строится асимптотически точная оценка для функции $T(n)$. В случае затруднения построения такой оценки, ограничиваются O -обозначением – асимптотической верхней границей функции $T(n)$.

На практике в процессе асимптотического анализа алгоритмов встречается небольшой набор классов сложности, которые приведены в табл. 1.3 (в порядке возрастания скорости роста).

Таблица 1.3. Классы сложности алгоритмов

Обозначение класса сложности	Название класса	Пример
$O(1)$	Константная сложность	Алгоритм определения четности целого числа. Время выполнения таких алгоритмов (или объем потребляемой памяти) не зависит от размера входных данных
$O(\log n)$	Логарифмическая сложность	Алгоритм бинарного поиска в упорядоченном массиве. Такие алгоритмы на каждом шаге обрабатывают лишь часть входного набора данных
$O(n)$	Линейная сложность	Алгоритм поиска минимального элемента в неупорядоченном массиве. Алгоритмы с такой сложностью выполняют просмотр всего набора входных данных
$O(n \log n)$	Линейно-логарифмическая сложность	Алгоритм сортировки слиянием. Такая сложность характерна для алгоритмов, разработанных с использованием метода декомпозиции («разделяй и властвуй»)
$O(n^2)$	Квадратичная сложность	Алгоритм сортировка выбором
$O(n^3)$	Кубическая сложность	Алгоритм умножения квадратных матриц по определению
$O(2^n)$	Экспоненциальная сложность	Алгоритмы, обрабатывающие все подмножества некоторого множества из n элементов
$O(n!)$	Факториальная сложность	Оптимизационные алгоритмы, реализующие полный перебор множества допустимых решений задачи

Пример. Факториал числа. Проведем асимптотический анализ вычислительной сложности алгоритма вычисления факториала целого числа n . По определению факториал числа n есть

$$n! = 1 \cdot 2 \cdot \dots \cdot n.$$

Алгоритм 1.4. Вычисление факториала числа

```

1 function FACTORIAL( $n$ )
2    $fact = 1$ 
3   for  $i = 2$  to  $n$  do
4      $fact = fact \cdot i$ 
5   end for
6   return  $fact$ 
7 end function
```

Шаг 1. Время выполнения алгоритма FACTORIAL зависит только от значения n , следовательно, функция $T(n)$ будет иметь один аргумент.

Шаг 2. Вычислим количество $T(n)$ операций алгоритма. Худший, средний и лучший случаи для этого алгоритма совпадают. Одна операция уходит на инициализацию переменной $fact$, далее на каждой из $n-1$ итерации цикла выполняется две операции (умножение и присваивание), после чего происходит возврат значения из функции – еще одна операция. Следовательно,

$$T(n) = 1 + 2(n - 1) + 1 = 2n.$$

Шаг 3. Получим асимптотически точную оценку вычислительной сложности алгоритма. Докажем, что $2n = \Theta(n)$. Возьмем $c_1 = 1$, $c_2 = 2$ и $n_0 = 0$, эти константы обеспечивает выполнение неравенства

$$0 \leq n \leq 2n \leq 2n, \quad \text{при } n \geq 0.$$

Результатом асимптотического анализа вычислительной сложности алгоритма FACTORIAL является асимптотически точная оценка $\Theta(n)$, которая говорит о том, что алгоритм имеет линейную вычислительную сложность.

Теперь выполним асимптотический анализ сложности по памяти алгоритма. Вычислим количество $M(n)$ ячеек памяти RAM-машины, которое требуется алгоритму. В строке 1 инициализируется переменная $fact$ – это требует одной ячейки памяти, для переменной i также требуется одна ячейка памяти. Окончательно получаем

$$M(n) = 2 = \Theta(1).$$

Алгоритм FACTORIAL имеет константную сложность по памяти – он не создает в памяти структур данных, размер которых зависит от n .

Пример. Сортировка выбором. Проведем асимптотический анализ вычислительной сложности алгоритма сортировки выбором (selection sort). Подробное описание алгоритма приведено в разделе 3.4.

Алгоритм 1.5. Сортировка выбором

```

1 function SELECTIONSORT( $A[1..n]$ )
2   for  $i = 1$  to  $n - 1$  do
3      $min = i$ 
4     for  $j = i + 1$  to  $n$  do
5       if  $a[j] < a[min]$  then
6          $min = j$ 
7       end if
8     end for
9     if  $min \neq i$  then
10       $temp = a[i]$ 
11       $a[i] = a[min]$ 
12       $a[min] = temp$ 
13    end if
14  end for
15 end function
```

Шаг 1. Время выполнения алгоритма SELECTIONSORT зависит от размера n массива и значений элементов в нем.

Шаг 2. Вычислим количество $T(n)$ операций алгоритма, выполняемых им в худшем случае. Это подразумевает выполнение максимального числа операций, что достигается при выполнении на всех итерациях циклов условий в строках 5 и 9. Худший случай для алгоритма SELECTIONSORT – это массив, элементы которого упорядочены по убыванию – в порядке, обратном направлению сортировки. В таком случае внешний цикл *for* выполняется $n - 1$ раз. На каждой итерации этого цикла одна операция приходится на запись значения i в переменную min (строка 3), что в сумме дает $n - 1$ операцию. На строки 9–13 приходится четыре операции (проверка условия и три присваивания), что дает еще $4(n - 1)$ операций. Тогда

$$T(n) = (n - 1) + 4(n - 1) + T_{InnerLoop}(n),$$

где функция $T_{InnerLoop}(n)$ отражает количество операций, приходящихся на внутренний цикл, за все время выполнения алгоритма.

Вычислим, сколько операций приходится на внутренний цикл в строках 4–8. Рассмотрим процесс выполнение внешнего цикла:

- при $i = 1$ внутренний цикл выполняется $n - 1$ раз (переменная j принимает значения от 2 до n);
- при $i = 2$ внутренний цикл выполняется $n - 2$ раза (переменная j

принимает значения от 3 до n);

- ...
- при $i = n - 2$ внутренний цикл выполняется 2 раза (переменная j принимает значения от $n - 1$ до n);
- при $i = n - 1$ внутренний цикл выполняется 1 раз (переменная j принимает значения от n до n).

Учитывая, что на каждой итерации внутреннего цикла выполняется две операции (проверка условия и присваивание (строки 4, 5)), получаем

$$\begin{aligned} T_{InnerLoop}(n) &= 2(n - 1 + n - 2 + \dots + 1), \\ T(n) &= (n - 1) + 4(n - 1) + 2[n - 1 + n - 2 + \dots + 1]. \end{aligned}$$

В квадратных скобках записана сумма членов арифметической прогрессии с разностью 1. Вычислим ее

$$T(n) = (n - 1) + 4(n - 1) + 2 \left[\frac{n^2 - n}{2} \right] = n^2 + 4n - 5.$$

Шаг 3. Получим асимптотически точную оценку вычислительной сложности алгоритма `SELECTIONSORT`. Докажем справедливость

$$T(n) = n^2 + 4n - 5 = \Theta(n^2).$$

Следуя определению асимптотических обозначений, возьмем константы $c_1 = 1$, $c_2 = 2$ и $n_0 = 2$. Они обеспечивают выполнение неравенств

$$0 \leq n^2 \leq n^2 + 4n - 5 \leq 2n^2, \quad \text{для любых } n \geq 2.$$

Таким образом, мы показали, что алгоритм сортировки выбором в худшем случае имеет *квадратичную вычислительную сложность*, и эта оценка является асимптотически точной.

Действуя аналогично, можно оценить вычислительную сложность алгоритма сортировки выбором для лучшего случая, когда входной массив уже упорядочен в направлении сортировки. Условия внутри циклов выполнятся не будут, и число операций алгоритма будет равно

$$T_{Best}(n) = (n - 1) + (n - 1) + [n - 1 + n - 2 + \dots + 1] = \Theta(n^2).$$

Сложность по памяти алгоритма сортировки выбором является константной, так как в процессе своей работы алгоритму требуется лишь четыре дополнительные ячейки под переменные i , j , min и $temp$ (входной массив $a[1..n]$ не учитывается).

1.7. Упражнения

1. Определите параметры, от которых зависит время работы перечисленных ниже алгоритмов:

- алгоритм последовательного суммирования элементов массива;
- алгоритм сложения по определению двух квадратных матриц;
- алгоритм Евклида нахождения наибольшего общего делителя двух целых чисел.

2. Разместите следующие функции по неубыванию скорости их роста

$$n, \quad \log_2 n, \quad \sqrt{n}, \quad \log_{10} n, \quad n^{1/3}, \quad \log_2(\log_2 n), \quad \frac{1}{2^n}, \quad (\log_2 n)^2, \quad 2^n.$$

3. Имеются вычислитель (процессор), который выполняет $3 \cdot 10^9$ операций в секунду, и три алгоритма с числом операций

$$T_1(n) = 400n \log_2 n + 1024, \quad T_2(n) = 2n^2 + 8n + 4, \quad T_3(n) = n! + 4.$$

Определите сколько времени (дней, часов, минут, секунд) будет выполняться каждый алгоритм при $n = 100$.

4. Используя определения асимптотических обозначений O и Θ , докажите справедливость следующих отношений:

- $n^3 - 2n^2 - 100n + 1 = O(n^3)$;
- $n^2 = O(2^n)$;
- $2^{n+1} = \Theta(2^n)$.

5. Сравните порядки роста функций через предел (1.7)

- $f(n) = 2^{2n}$, $g(n) = 2^n$;
- $f(n) = \ln n$, $g(n) = n$;
- $f(n) = 2^n$, $g(n) = n^{100}$ *;
- $f(n) = \log_2 n$, $g(n) = \sqrt{n}$;
- $f(n) = n!$, $g(n) = 2^n$ †.

6. Ниже приведен псевдокод функции LOOPS_A. Вычислите количество $T(n)$ операций в ней.

7. Ниже приведен псевдокод функции LOOPS_B. Вычислите количество $T(n)$ операций в ней.

*Подсказка: примените правило Лопиталя 100 раз.

†Подсказка: используйте формулу Стирлинга.

```
1 function LOOPSA(n)
2     x = 1
3     for i = 1 to n do
4         for j = i + 1 to n do
5             for k = 1 to n do
6                 x = x + 1
7             end for
8         end for
9     end for
10    return x
11 end function
```

```
1 function LOOPSB(n)
2     x = 1
3     for i = 1 to n do
4         for j = i + 1 to n do
5             for k = 1 to j do
6                 x = x + 1
7             end for
8         end for
9     end for
10    return x
11 end function
```

2. Анализ рекурсивных алгоритмов

2.1. Рекурсивные алгоритмы

При проектировании и реализации различных структур данных и алгоритмов достаточно часто приходится иметь дело с рекурсивными функциями.

Рекурсивная функция (recursive function) – это функция, в теле которой присутствует вызов самой себя. Соответственно, алгоритм, основанный на таких функциях, называется *рекурсивным алгоритмом* (recursive algorithm). В практике программирования встречаются различные формы рекурсии. Ниже приведен пример *прямой рекурсии* (direct recursion) для вычисления факториала заданного числа n . Такая форма рекурсии подразумевает присутствие вызова функции непосредственно в ее теле. Если же некоторая функция A вызывает функцию B , а функция B реализует рекурсивный вызов A , то мы имеем дело с *косвенной рекурсией* (indirect recursion).

Алгоритм 2.1. Рекурсивное вычисление факториала числа

```
1 function FACTORIAL( $n$ )
2     if  $n = 1$  then
3         return 1
4     end if
5     return  $n \cdot \text{FACTORIAL}(n - 1)$ 
6 end function
```

Рекурсивный алгоритм можно охарактеризовать *деревом рекурсивных вызовов* (recursion tree). В таком дереве каждый узел соответствует вызову функции. Время выполнения рекурсивного алгоритма определяется числом узлов в его дереве рекурсивных вызовов. Точнее говоря, суммарным временем выполнения всех узлов – временем реализации всех рекурсивных вызовов.

На рис. 2.1, а показано дерево рекурсивных вызовов для алгоритма вычисления факториала числа. Этот алгоритм является примером *линейной рекурсии* (linear recursion) – в функции FACTORIAL присутствует единственный рекурсивный вызов (дерево вырождается в список). Если в функции

присутствует более одного рекурсивного вызова, то имеет место *нелинейная рекурсия* (nonlinear recursion), которую также называют *древовидной рекурсией*. Примером может служить алгоритм FIB вычисления n -го числа последовательности Фибоначчи (рис. 2.1, б).

Алгоритм 2.2. Рекурсивное вычисление n -го числа Фибоначчи

```

1 function FIB( $n$ )
2   if  $n < 2$  then
3     return  $n$ 
4   end if
5   return FIB( $n - 1$ ) + FIB( $n - 2$ )
6 end function
```

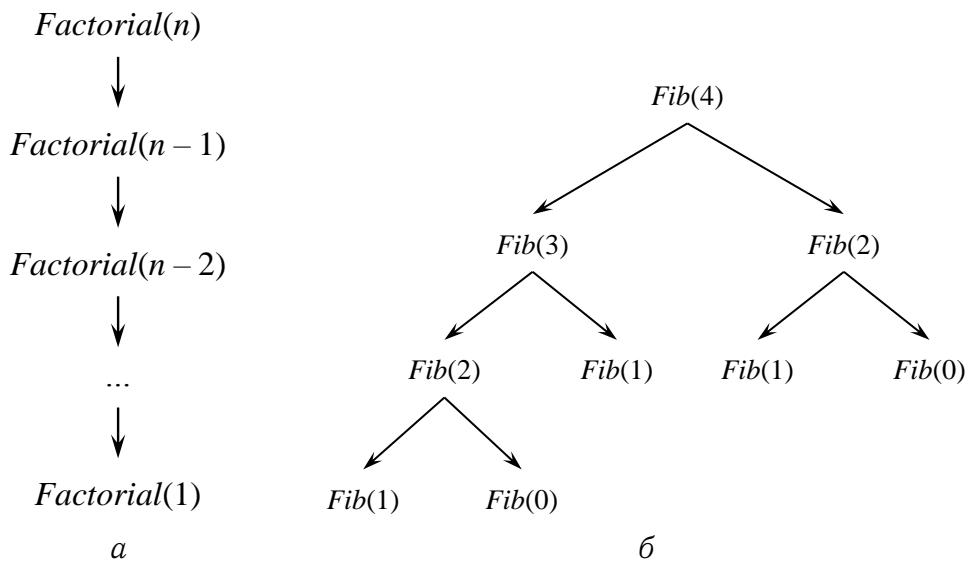


Рис. 2.1. Дерево рекурсии:

- a – алгоритм вычисления факториала числа n ;
- b – вычисление 4-го члена последовательности Фибоначчи.

При вызове функции значения переданных ей параметров и адрес возврата сохраняются в аппаратном стеке и регистрах процессора и удаляются из стека при выходе из нее. Аппаратный стек имеет ограниченный размер, что важно учитывать при использовании рекурсивных алгоритмов. Если глубина рекурсивных вызовов будет слишком большой – возможно возникновение системной ошибки: переполнение стека (stack overflow).

Для выполнения алгоритма FACTORIAL (см. рис. 2.1, а) требуется $\Theta(n)$ ячеек памяти аппаратного стека. При каждом вызове функции FACTORIAL в стеке сохраняется адрес возврата из функции. Например, если аппаратный стек имеет размер 8 Мбайт, а каждый вызов функции требует сохранения в стеке адреса возврата длиной 8 байт, то через $8 \cdot 1024 \cdot 1024 / 8 = 1\,048\,576$ рекурсивных вызовов стек будет переполнен и программа аварийно завершится. На самом деле аварийное завершение наступит раньше, так как

в нашей оптимистичной оценке мы не учли константное число ячеек, которые сохраняются в стеке при каждом вызове функции (например, ее аргументы). Предварительный анализ глубины дерева рекурсивных вызовов алгоритма и объема требуемой стековой памяти позволяет запускать программы с необходимым размером стека и избегать подобных ошибок.

Анализ эффективности рекурсивных алгоритмов сложнее анализа итеративных алгоритмов (ему была посвящена глава 1). Трудности здесь связаны с необходимостью оценивания высоты дерева рекурсивных вызовов. Далее мы рассмотрим основные этапы анализа эффективности рекурсивных алгоритмов на примере сортировки слиянием.

2.2. Сортировка слиянием

Задаче сортировки посвящена глава 3. Здесь мы остановимся лишь на основных определениях, необходимых для рассмотрения алгоритма сортировки слиянием.

Задан массив $A[1..n]$, содержащий n элементов (ключей). Будем считать, что нам известна процедура сравнения ключей. Требуется упорядочить массив *по неубыванию* (non-decreasing order) так, чтобы

$$A[1] \leq A[2] \leq \dots \leq A[n].$$

Сортировка слиянием (merge sort) – это рекурсивный алгоритм сортировки, основанный на методе декомпозиции (decomposition) или, как его еще называют, «разделяй и властвуй» (divide and conquer). Ниже приведен псевдокод функции $\text{MERGESORT}(A, low, high)$, которая реализует сортировку слиянием подмассива $A[low..high]$. Подразумевается, что первоначальный вызов этой функции имеет вид $\text{MERGESORT}(A[1..n], 1, n)$.

2.2.1. Фаза разделения

При каждом вызове функции MERGESORT массив $A[low..high]$, содержащий $high - low + 1$ элементов, *разделяется* (partition) на две максимально равные по длине части. Первая часть содержит $\lceil n/2 \rceil$ элементов, а вторая $\lfloor n/2 \rfloor$. Выражение $\lceil x \rceil$ обозначает наименьшее целое число, которое больше или равно x (ближайшее целое сверху, ceil), а $\lfloor x \rfloor$ – наибольшее целое число, которое меньше или равно x (ближайшее целое снизу, floor). Границей разбиения служит переменная mid , в которую записывается номер центрального элемента отрезка $[low..high]$.

Полученные подмассивы $A[low..mid]$ и $A[mid+1..high]$ рекурсивно сортируются с использованием сортировки слиянием. Разбиение и рекурсивные вызовы осуществляются до тех пор, пока в подмассивах не останется

Алгоритм 2.3. Сортировка слиянием

```

1 function MERGESORT( $A[1..n]$ ,  $low$ ,  $high$ )
2   if  $low < high$  then
3      $mid = \text{FLOOR}((low + high)/2)$ 
4     MERGESORT( $A$ ,  $low$ ,  $mid$ )
5     MERGESORT( $A$ ,  $mid + 1$ ,  $high$ )
6     MERGE( $A$ ,  $low$ ,  $mid$ ,  $high$ )
7   end if
8 end function

9 function MERGE( $A[1..n]$ ,  $low$ ,  $mid$ ,  $high$ )
10  for  $i = low$  to  $high$  do
11     $B[i] = A[i]$                                 /* Создаем копию массива  $A$  */
12  end for
13   $l = low$           /* Номер первого элемента левого подмассива */
14   $r = mid + 1$     /* Номер первого элемента правого подмассива */
15   $i = low$ 
16  while  $l \leq mid$  and  $r \leq high$  do
17    if  $B[l] \leq B[r]$  then
18       $A[i] = B[l]$ 
19       $l = l + 1$ 
20    else
21       $A[i] = B[r]$ 
22       $r = r + 1$ 
23    end if
24     $i = i + 1$ 
25  end while
26  while  $l \leq mid$  do
27     $A[i] = B[l]$       /* Копируем элементы из левого подмассива */
28     $l = l + 1$ 
29     $i = i + 1$ 
30  end while
31  while  $r \leq high$  do
32     $A[i] = B[r]$     /* Копируем элементы из правого подмассива */
33     $r = r + 1$ 
34     $i = i + 1$ 
35  end while
36 end function
```

по одному элементу. После чего осуществляются подъем по дереву рекурсивных вызовов и *слияние* (merge) отсортированных подмассивов. На рис. 2.2 показано дерево рекурсивных вызовов функции MERGESORT для

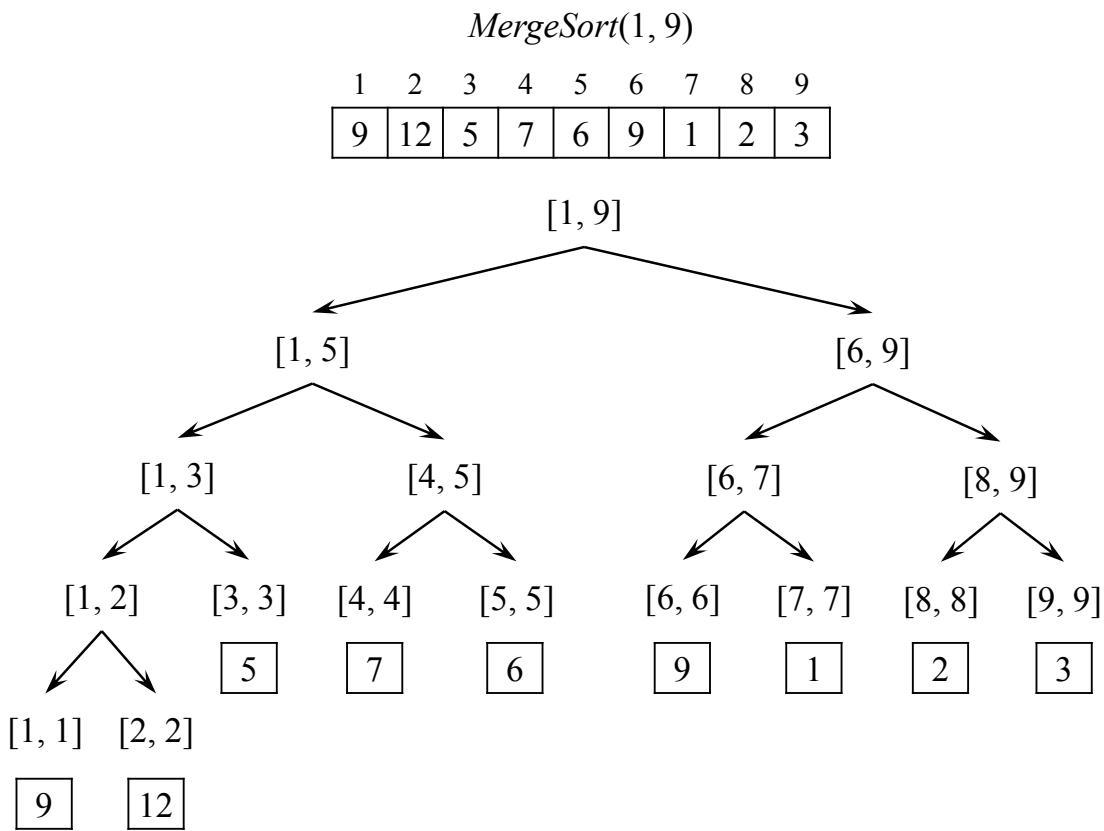


Рис. 2.2. Сортировка слиянием массива из 9 элементов: фаза разделения (в квадратных скобках показаны границы сортируемых подмассивов).

массива из 9 элементов, а на рис. 2.3 приведена фаза слияния подмассивов.

2.2.2. Фаза слияния

Слияние реализуется функцией `MERGE`. Она сливает упорядоченные подмассивы $A[low..mid]$ и $A[mid+1..high]$ в один отсортированный массив, элементы которого занимают позиции $A[low..high]$.

Перед слиянием создается копия B массива $A[low..high]$. Это делается для того, чтобы не затереть во время слияния элементы в A . Далее в цикле сравниваются первые элементы левого и правого подмассивов: $B[l]$ и $B[r]$. Меньшее значение заносится в ячейку $A[i]$. В подмассиве, из которого скопировали значение в $A[i]$, текущим становится следующий элемент – переменная l или r увеличивается на единицу. Сравнение и копирование элементов в A происходит до тех пор, пока не будет достигнут конец одного из подмассивов. После этого оставшиеся элементы из массива с большей длиной дописываются в A .

Функция `MERGE` требует порядка $\Theta(n)$ ячеек памяти для хранения копии B сортируемого массива.

Вычислительная сложность слияния определяется длинами подмассивов и равна $\Theta(m)$, где $m = high - low + 1$. Для создания копии массива A требуется выполнить порядка $\Theta(m)$ операций присваивания. Сравнение и

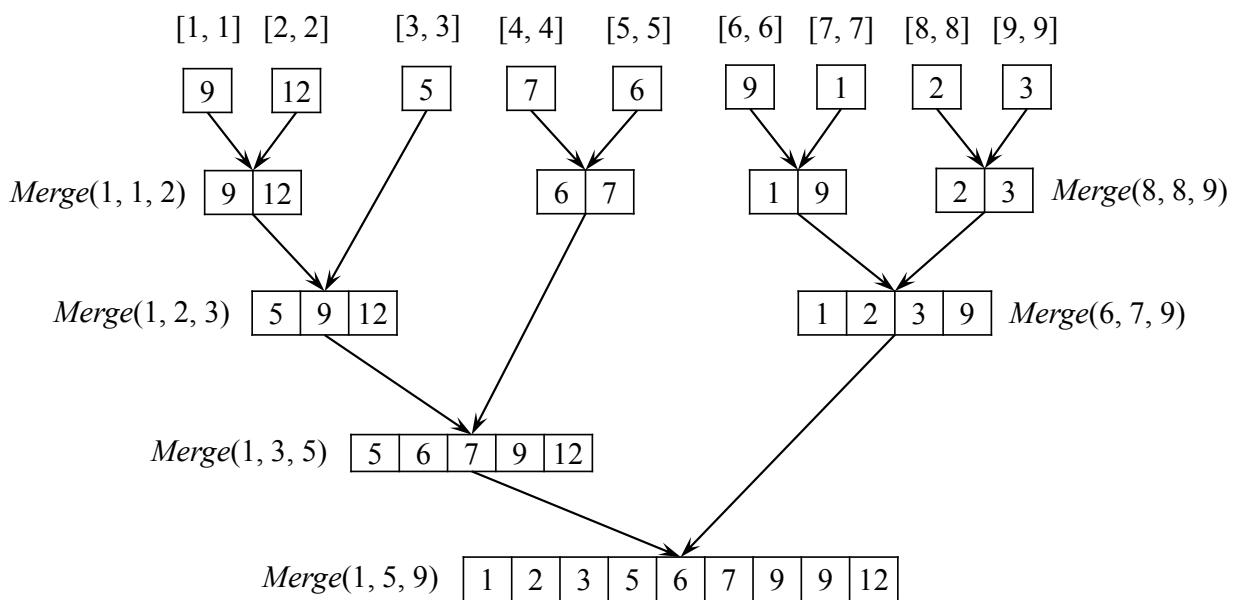


Рис. 2.3. Сортировка слиянием массива из 9 элементов: фаза слияния (в квадратных скобках показаны границы сливаемых подмассивов).

перенос элементов из массива B в массив A требует $\Theta(\max\{L, R\})$ операций, где L и R – длины левого и правого подмассивов: $L = mid - low + 1$, $R = high - mid$.

2.2.3. Анализ эффективности сортировки слиянием

Обозначим время выполнения сортировки слиянием массива из n элементов через $T(n)$. Оно включает в себя время сортировки левого подмассива длины $\lceil n/2 \rceil$ и правого – с числом элементов $\lfloor n/2 \rfloor$, а также время $\Theta(n)$ слияния подмассивов после их рекурсивного упорядочивания

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n). \quad (2.1)$$

Приведенная запись времени выполнения алгоритма называется *рекуррентным уравнением*, так как время $T(n)$ сортировки массива из n элементов выражено через время $T(\lceil n/2 \rceil)$ и $T(\lfloor n/2 \rfloor)$ сортировки массивов меньшей длины. Нам необходимо решить это уравнение и получить выражение для $T(n)$ без рекуррентности. В следующем разделе мы рассмотрим общий подход к решению рекуррентных уравнений. Сейчас же проведем анализ вычислительной сложности сортировки слиянием *методом анализа дерева рекурсивных вызовов*.

Не теряя общности, будем считать, что число n элементов в массиве равно степени двойки. Это допущение не влияет на итоговую асимптотическую оценку вычислительной сложности алгоритма.

На рис. 2.4 приведено дерево рекурсивных вызовов сортировки слиянием массива из n элементов. В каждом узле дерева записано число

операций, требуемых для слияния двух подмассивов.

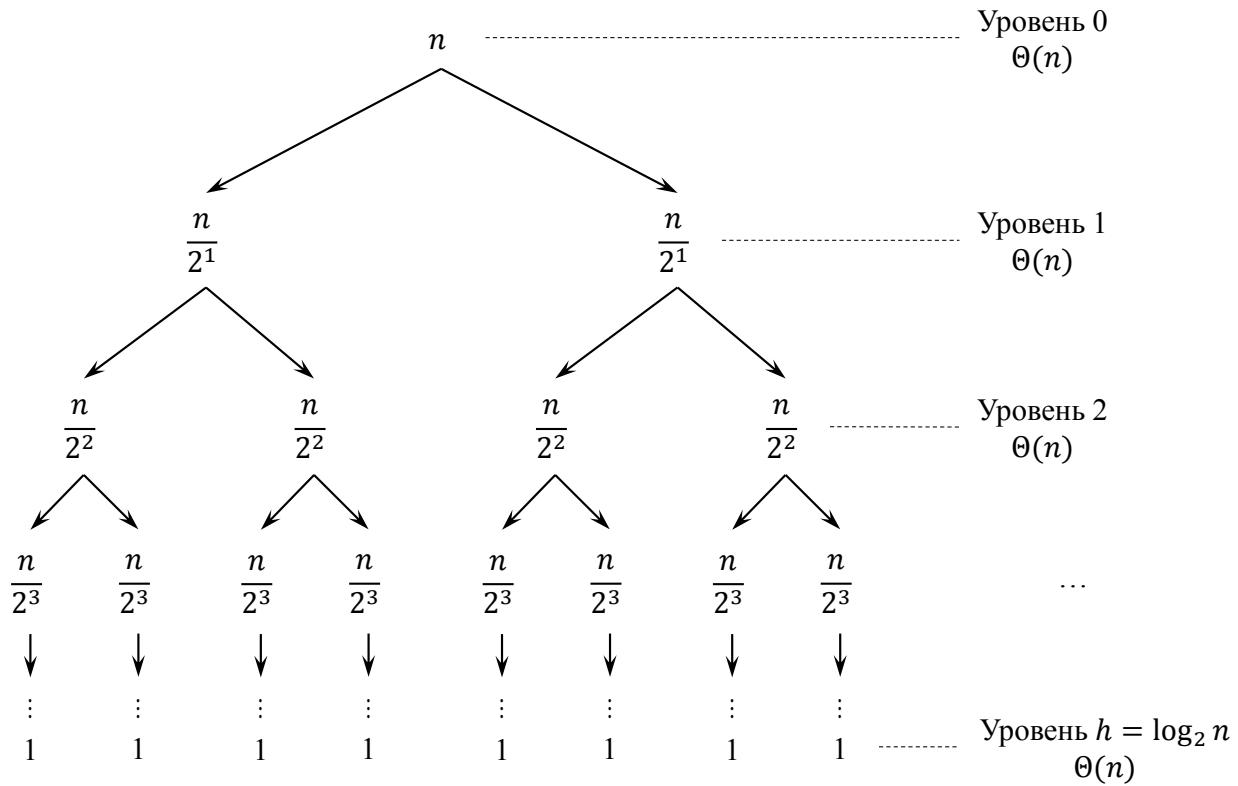


Рис. 2.4. Дерево рекурсивных вызовов алгоритма сортировки слиянием массива из n элементов (n равно степени двойки).

Число $T(n)$ операций алгоритма MERGESORT равно сумме операций, записанных во всех узлах дерева. Другими словами, равно количеству операций, требуемых для слияния всех подмассивов.

Оценим высоту h дерева рекурсивных вызовов алгоритма MERGESORT. На уровне 0 осуществляется разбиение исходного массива длины n , на уровне 1 – подмассивов длины $n/2$, на уровне 2 – подмассивов длины $n/4$ и т.д. Разбиение заканчивается на некотором уровне h , когда длина подмассива становится равной 1. Следовательно

$$\frac{n}{2^h} = 1 \Rightarrow h = \log_2 n.$$

Таким образом, мы показали, что высота h дерева рекурсивных вызовов равна $\Theta(\log n)$.

В общем случае на каждом уровне $i \in \{0, 1, \dots, \log_2 n\}$ находится 2^i узлов. Каждый узел требует выполнения $n/2^i$ операций. Вычислим сумму операций всех узлов на всех уровнях

$$T(n) = \sum_{i=0}^h 2^i \frac{n}{2^i} = \sum_{i=0}^h n = (h+1)n = n \log_2 n + n = \Theta(n \log n).$$

Вычислительная сложность сортировки слиянием в худшем случае равна $\Theta(n \log n)$. Сложность по памяти алгоритма есть $\Theta(n)$, так как слияние требует создания копии сортируемого массива.

2.3. Решение рекуррентных уравнений

Для решения рекуррентных уравнений разработаны различные методы [1, 4]. При анализе вычислительной сложности сортировки слиянием мы использовали метод деревьев рекурсии. Теперь мы рассмотрим один из общих подходов к решению рекуррентных уравнений – *основной метод* (master method).

Основной метод широко применяется для анализа алгоритмов, основанных на методе декомпозиции. Рассмотрим решение рекуррентных уравнений, когда исходную задачу размера n можно разделить на $a \geq 1$ подзадач размера n/b . Будем считать, что для решения задачи размера 1 требуется время $O(1)$, а для декомпозиции задачи размера n и комбинирования (слияния) решений подзадач требуется $f(n)$ единиц времени. Тогда время $T(n)$ решения задачи размера n можно записать как

$$T(n) = aT(n/b) + f(n),$$

где $a \geq 1$, $b > 1$.

Записанное уравнение называется *обобщенным рекуррентным уравнением декомпозиции* (general divide-and-conquer recurrence). Решением этого уравнения является порядок роста функции $T(n)$, который определяется из следующей теоремы [1, 4].

Теорема 1. *Если в обобщенном рекуррентном уравнении декомпозиции $f(n) = \Theta(n^d)$, где $d \geq 0$, то*

$$T(n) = \begin{cases} \Theta(n^d), & \text{если } a < b^d, \\ \Theta(n^d \log n), & \text{если } a = b^d, \\ \Theta(n^{\log_b a}), & \text{если } a > b^d. \end{cases} \quad (2.2)$$

Например, в рекуррентном уравнении (2.1) алгоритма сортировки слиянием $a = 2$, $b = 2$, $f(n) = \Theta(n)$ и $d = 1$. Следовательно, имеем случай $a = b^d$. Тогда, следуя теореме, вычислительная сложность сортировки слиянием в худшем случае равна

$$T(n) = \Theta(n^d \log n) = \Theta(n \log n).$$

Рассмотрим другой пример. Пусть для некоторого алгоритма получен-

ное рекуррентное уравнение

$$T(n) = 2T(n/2) + 1.$$

Необходимо найти асимптотически точную оценку для $T(n)$. Нетрудно заметить, что в данном случае $a = 2$, $b = 2$, $f(n) = \Theta(1)$ и $d = 0$. Следовательно, поскольку $a > b^d$:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Более подробное описание основного метода и границ его применимости можно найти в [1].

2.4. Упражнения

1. Используя дерево рекурсивных вызовов на рис. 2.1, а постройте асимптотическую оценку вычислительной сложности алгоритма FACTORIAL.

2. С помощью основного метода постройте асимптотически точные оценки для следующих $T(n)$:

- $T(n) = 4T(n/2) + 1;$
- $T(n) = 4T(n/2) + \sqrt{n};$
- $T(n) = 4T(n/2) + n;$
- $T(n) = 4T(n/2) + n^2;$
- $T(n) = 2T(n/4) + 1;$
- $T(n) = 2T(n/4) + \sqrt{n};$
- $T(n) = 2T(n/4) + n;$
- $T(n) = 2T(n/4) + n^2.$

3. Постройте дерево рекурсивных вызовов для приведенного ниже алгоритма SUM вычисления суммы элементов заданного массива. Используя основной метод, получите асимптотически точную оценку вычислительной сложности алгоритма.

```

1 function SUM( $A[1..n]$ ,  $low$ ,  $high$ )
2   if  $low = high$  then
3     return  $A[low]$ 
4   end if
5    $mid = \text{FLOOR}((low + high)/2)$ 
6   return SUM( $A$ ,  $low$ ,  $mid$ ) + SUM( $A$ ,  $mid + 1$ ,  $high$ )
7 end function

```

4. Запишите рекуррентное уравнение для алгоритма вычисления n -го числа последовательности Фибоначчи (см. рис. 2.4, б). Используя формулу

Бине, покажите, что асимптотическая оценка вычислительной сложности алгоритма FIB равна $\Theta(\varphi^n)$, где $\varphi \approx 1.62$ – отношение золотого сечения (см. приложение).

5. Выясните, какой размер аппаратного стека по умолчанию выделяется процессам в вашей операционной системе. Оцените, при каком значении n произойдет переполнение стека в алгоритме FACTORIAL.

6. Подумайте, насколько реально на практике возникновение переполнения аппаратного стека в рекурсивном алгоритме MERGESORT.

7. Выполните асимптотический анализ вычислительной сложности рекурсивного алгоритма ISPRIME проверки заданного числа n на простоту.

```

1 function HASDIVISORS(number, divisor)
2     if divisor = 1 then
3         return False
4     end if
5     if number % divisor = 0 then
6         return True
7     end if
8     return HASDIVISORS(number, divisor - 1)
9 end function

10 function ISPRIME(n)
11     if n = 1 then
12         return True
13     end if
14     if HASDIVISORS(n, n/2) = False then
15         return True
16     end if
17     return False
18 end function
```

3. Сортировка

3.1. Задача сортировки

Задача сортировки (sorting problem) заключается в упорядочивании заданной последовательности из n ключей (a_1, a_2, \dots, a_n) по неубыванию или по невозрастанию. Под упорядочиванием заданной последовательности *по неубыванию* (non-decreasing order) понимается нахождение такой перестановки (i_1, i_2, \dots, i_n) ключей, при которой

$$a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}.$$

Аналогично, упорядочивание *по невозрастанию* (non-increasing order) – это нахождение такой перестановки (i_1, i_2, \dots, i_n) ключей, при которой

$$a_{i_1} \geq a_{i_2} \geq \dots \geq a_{i_n}.$$

Отметим, что элементы входной последовательности представляют собой объекты одного типа данных и могут иметь одинаковые значения.

Алгоритму сортировки должен быть известен способ сравнения ключей. Для этого ему сообщается *функция сравнения* (comparison function), которая для заданной пары ключей (x, y) возвращает значение *True*, если $x < y$, и *False* в противном случае.

Для базовых типов данных (целый, вещественный, логический тип) в языках программирования уже реализована операция « $<$ ». Однако для составных/пользовательских типов такую функцию необходимо явно сообщить алгоритму. Например, для сравнения строк с учетом регистра символов используется специальная функция, которая реализует посимвольное сравнение двух строк (устанавливает *лексикографический порядок* на множестве строк).

3.2. Свойства и виды алгоритмов сортировки

Рассмотрим основные свойства и виды алгоритмов сортировки. Алгоритм сортировки, который явно или неявно использует функцию сравнения ключей, называется *сортировкой, основанной на сравнениях* (comparison sort), или просто *сортировкой сравнением*. Вычислительная сложность

таких алгоритмов определяется числом выполняемых ими операций сравнения ключей. К этому классу алгоритмов сортировки относятся: *сортировка слиянием* (merge sort), *быстрая сортировка* (quick sort), *пирамидальная сортировка* (heap sort), *сортировка выбором* (selection sort), *сортировка вставкой* (insertion sort) и др.

Существуют алгоритмы сортировки, которые не используют операцию сравнения ключей (non-comparison sort). Такие алгоритмы опираются на информацию о типе данных ключа и возможных значениях элементов сортируемой последовательности. Как правило, они способны упорядочивать ключи лишь фиксированного типа данных. За счет этого некоторые из них позволяют выполнять сортировку за линейное время. Представителями этого семейства являются алгоритмы *целочисленной сортировки* (integer sort): *сортировка подсчетом* (counting sort) и *поразрядная сортировка* (radix sort). Они предназначены для упорядочивания последовательностей неотрицательных целых чисел.

Алгоритм сортировки называется *устойчивым* (stable), если он сохраняет относительный порядок следования одинаковых ключей. Иначе говоря, если во входной последовательности на позициях i и j имеется два одинаковых ключа, причем $i < j$, то в отсортированной последовательности их номера будут i' и j' и будет справедливо неравенство $i' < j'$. Рассмотрим это свойство на примере. На рис. 3.1, а показана исходная последовательность пар «(Фамилия, Имя)», которую требуется упорядочить по ключу «Фамилия». На рис. 3.1, б показан результат применения устойчивой сортировки. Как и во входной последовательности пара (Иванов, Иван), предшествует паре (Иванов, Николай), которая в свою очередь стоит перед парой (Иванов, Никодим). На рис. 3.1, в показан результат применения неустойчивой сортировки. Пара (Иванов, Никодим) перенесена на первое место, это нарушило исходный порядок следования элементов с одинаковым значением ключа сортировки.

(Иванов, Иван), (Сидоров, Сидор), (Иванов, Николай),
(Петров, Пётр), (Иванов, Никодим)

а
(Иванов, Иван), (Иванов, Николай), (Иванов, Никодим),
(Петров, Пётр), (Сидоров, Сидор)

б
(Иванов, Никодим), (Иванов, Николай), (Иванов, Иван),
(Петров, Пётр), (Сидоров, Сидор)

в

Рис. 3.1. Сортировка последовательности по ключу «Фамилия»:
а – исходная последовательность; *б* – применение устойчивой сортировки;
в – применение неустойчивой сортировки.

Алгоритм сортировки, который в процессе своей работы помимо заданной совокупности ключей (a_1, a_2, \dots, a_n) использует лишь константное число ячеек памяти, называется *сортировкой на месте* (in-place sort). Другими словами, алгоритм не использует дополнительных структур данных с размером, зависящим от объема и значений входных данных.

Если упорядочиваемая последовательность ключей (a_1, a_2, \dots, a_n) целиком помещается в оперативной памяти, то говорят о *внутренней сортировке* (internal sort). Если же последовательность ключей очень большая и не помещается в оперативную память, то она хранится на относительно медленном внешнем устройстве памяти (жестком диске, сетевом хранилище и др.). В этом случае используют алгоритмы *внешней сортировки* (external sort). Они многократно используют оперативную память для сортировки частей исходной последовательности ключей. Примером может служить *внешняя сортировка слиянием* (external merge sort).

Мы начнем рассмотрение методов сортировки с простейших алгоритмов с квадратичной вычислительной сложностью в худшем случае. К ним относятся: *сортировка вставкой* (insertion sort), *сортировка выбором* (selection sort) и *пузырьковая сортировка* (bubble sort). На практике их используют только для сортировки небольших массивов либо в комбинации с более эффективными алгоритмами.

Подробно мы рассмотрим *сортировку слиянием* (merge sort), *быструю сортировку* (quick sort), *пирамидальную сортировку* (heap sort) и *сортировку подсчетом* (counting sort). В реальных приложениях в среднем случае быстрая сортировка опережает сортировку слиянием и пирамидальную сортировку.

Ниже приведена таблица с информацией о вычислительной сложности популярных алгоритмов сортировки.

3.3. Сортировка вставкой

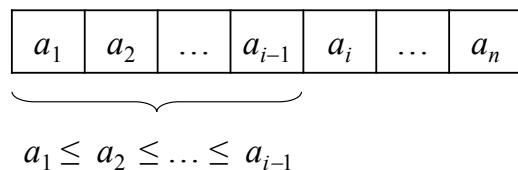
Сортировка вставкой (insertion sort) относится к простейшим алгоритмам сортировки на основе сравнений. Алгоритм имеет квадратичную вычислительную сложность и основан на *методе уменьшения задачи* (decrease and conquer). Суть его заключается в следующем. Имеется массив $A[1..n]$, состоящий из n элементов (ключей). Будем двигаться по нему слева направо и располагать элементы по неубыванию.

Подмассив из одного элемента $A[1..1]$ уже отсортирован. Переходим к элементу 2. Слева от элемента $A[2]$ находится упорядоченный подмассив $A[1..1]$ из одного ключа. Сравниваем $A[1]$ и $A[2]$, большее значение переносим в позицию 2. Получили упорядоченный подмассив $A[1..2]$. Переходим к элементу 3. В упорядоченном подмассиве $A[1..2]$ находим позицию для $A[3]$. Получили отсортированный подмассив $A[1..3]$ и т. д. В общем слу-

Таблица 3.1. Алгоритмы сортировки

Алгоритм	Вычислительная сложность			Сложность по памяти	Свойства
	Худший случай	Средний случай	Лучший случай		
Сортировка выбором (selection sort)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Устойчивость зависит от реализации
Сортировка вставкой (insertion sort)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$O(1)$	Устойчивая, online
Сортировка слиянием (merge sort)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Устойчивая
Пирамидальная сортировка (heap sort)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$	Неустойчивая
Быстрая сортировка (quick sort)	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(\log n)$	Неустойчивая
Сортировка подсчетом (counting sort)	$\Theta(k + n)$	$\Theta(k + n)$	$\Theta(k + n)$	$\Theta(k + n)$	Устойчивая, целочисленная

чае на шаге i мы имеем упорядоченный подмассив $A[1..i - 1]$ и элемент $A[i]$, который необходимо вставить в этот подмассив (рис. 3.2). Для этого проходим по подмассиву справа налево, пока не найдем первый ключ, меньший или равный $A[i]$, вставляем элемент $A[i]$ непосредственно за ним. В процессе движения влево (к началу массива) мы переносим элементы в соседнюю ячейку справа, тем самым освобождая позицию для $A[i]$. Таким образом мы получим упорядоченный подмассив $A[1..i]$. Ниже приведен псевдокод алгоритма INSERTION SORT.

Рис. 3.2. Сортировка вставкой: вставка элемента a_i в упорядоченный подмассив $A[1..i - 1]$.

Алгоритм 3.1. Сортировка вставкой

```

1 function INSERTIONSORT( $A[1..n]$ )
2   for  $i = 2$  to  $n$  do
3      $key = A[i]$ 
4      $j = i - 1$            /* Вставляем  $A[i]$  в подмассив  $A[1..i - 1]$  */
5     while  $j > 0$  and  $A[j] > key$  do
6        $A[j + 1] = A[j]$ 
7        $j = j - 1$ 
8     end while
9      $A[j + 1] = key$ 
10   end for
11 end function

```

Оценим время работы алгоритма INSERTIONSORT для худшего случая, при котором цикл *while* всегда доходит до первого элемента массива. Такая ситуация возникает, если на вход поступил массив, упорядоченный по убыванию: $A[1] > A[2] > \dots > A[n]$. Для вставки элемента $A[i]$ на свое место требуется $i - 1$ итерация цикла *while*. На каждой итерации этого цикла выполняется константное число c действий: перенос элемента в соседнюю ячейку и уменьшение переменной j . Тогда, учитывая, что нам необходимо найти позиции для $n - 1$ элемента, время $T(n)$ выполнения алгоритма в худшем случае будет равно

$$T(n) = \sum_{i=2}^n c(i-1) = c + 2c + \dots + (i-1)c + \dots + (n-1)c = \frac{cn(n-1)}{2} = \Theta(n^2).$$

На уже упорядоченных по неубыванию массивах сортировка вставкой работает за *линейное время* (лучший случай, см. упражнения). В некоторых приложениях это может оказаться полезным. Например, сперва мы можем начать упорядочивать массив алгоритмом быстрой сортировки, а при малых значениях n , когда массив уже почти упорядочен, переключиться на использование сортировки вставкой. Пороговое значение n для переключения на INSERTIONSORT выбирается опытным путем (например, $n = 10$).

В среднем случае время работы алгоритма сортировки вставкой равно

$$T(n) = \frac{cn(n-1)}{4} = \Theta(n^2).$$

Это означает, что в среднем сортировка вставкой работает в два раза быстрее по сравнению с худшим случаем. Анализ среднего случая основан на введении понятия инверсии и индикаторной случайной величины [1, 2].

Алгоритм сортировки вставкой является *устойчивым*, так как не меняет относительный порядок следования одинаковых ключей. В процессе

своей работой алгоритм использует константное число дополнительных ячеек памяти (переменные i , key и j), что относит его к классу алгоритмов сортировки *на месте* (in-place sort). Кроме того, алгоритм относится к классу *online-алгоритмов* – обеспечивает возможность упорядочивания массивов при динамическом поступлении новых элементов.

3.4. Сортировка выбором

Сортировка выбором (selection sort) – это алгоритм сортировки с квадратичной вычислительной сложностью, основанный на сравнениях. Ниже приведен его псевдокод. На первой итерации внешнего цикла *for* среди элементов $A[2..n]$ отыскивается минимальный ключ $A[minindex]$. Элементы $A[1]$ и $A[minindex]$ меняются местами. На второй итерации цикла минимальный ключ ищется среди элементов $A[3..n]$. Элементы $A[2]$ и найденный минимальный $A[minindex]$ меняются местами. В общем случае на итерации i цикла *for* часть массива $[1..i - 1]$ уже упорядочена. Среди элементов $A[i + 1..n]$ отыскивается минимальный ключ для помещения в ячейку $A[i]$. Таким образом, двигаясь слева направо, мы упорядочиваем массив, находя очередной минимальный элемент в оставшейся правой части массива.

Алгоритм 3.2. Сортировка выбором

```

1 function SELECTIONSORT( $A[1..n]$ ,  $n$ )
2   for  $i = 1$  to  $n - 1$  do
3      $minindex = i$ 
4     for  $j = i + 1$  to  $n$  do
5       if  $A[j] < A[minindex]$  then
6          $minindex = j$ 
7       end if
8     end for
9     if  $minindex \neq i$  then
10       $temp = A[i]$ 
11       $A[i] = A[minindex]$ 
12       $A[minindex] = temp$ 
13    end if
14  end for
15 end function
```

Вычислительные сложности алгоритма SELECTIONSORT для худшего и среднего случая совпадают. Дело в том, что на любом наборе входных данных число сравнений во внутреннем цикле остается неизменным. Вычислим его. На первой итерации внешнего цикла ($i = 1$) условие в строке 5

выполняется $n - 1$ раз, на второй итерации – $n - 2$ раза и т. д. На итерации $n - 1$ условие выполняется один раз. После выполнения внутреннего цикла происходит обмен значений элементов $A[i]$ и $A[minindex]$. На это требуется константное число операций. Обозначим его через c . Теперь запишем суммарное время $T(n)$ выполнения алгоритма

$$T(n) = c(n - 1) + c(n - 2) + \dots + c = \sum_{i=1}^{n-1} c(n - i) = \frac{cn(n - 1)}{2} = \Theta(n^2).$$

Рассмотренная реализация сортировки выбором является *неустойчивой* (non-stable sort). Найденный минимальный элемент $A[minindex]$ меняется местами с элементом $A[i]$. Следовательно, элемент $A[i]$ может переместиться на другую позицию относительно равного ему ключа. Рассмотрим пример. Пусть дана совокупность пар, которую надо отсортировать по первому полю:

$$(55, A), (55, B), (10, C).$$

После выполнения нашего алгоритмы получим следующую упорядоченную последовательность

$$(10, C), (55, B), (55, A).$$

Видно, что порядок пар $(55, A)$ и $(55, B)$ нарушен.

В процессе своей работы алгоритм сортировки выбором использует константное число дополнительных ячеек памяти (переменные i , j , $minindex$ и $temp$), следовательно, он является сортировкой на месте.

3.5. Нижняя граница времени сортировки сравнением

Любой алгоритм сортировки сравнением можно представить в виде *дерева решений* (decision tree) – полного бинарного дерева (full binary tree), в котором каждой вершине сопоставлена пара (i, j) , что означает сравнение i -го и j -го элементов сортируемой последовательности. Любой внутренний узел имеет две дочерних вершины. Левый сын соответствует ситуации $a_i < a_j$, правый: $a_i \geq a_j$. Каждый лист дерева задает некоторую перестановку $\pi(i_1, i_2, \dots, i_n)$ элементов, которая приводит последовательность к упорядоченному виду. Процесс работы алгоритма соответствует движению по данному дереву от корня к листу. При достижении листа получаем упорядоченную последовательность. Дерево решения отражает все возможные варианты сравнений, которые возникают при упорядочивании

произвольной последовательности из n элементов заданным алгоритмом. На рис. 3.3 показано дерево решения для алгоритма сортировки вставкой массива из трех элементов [1].

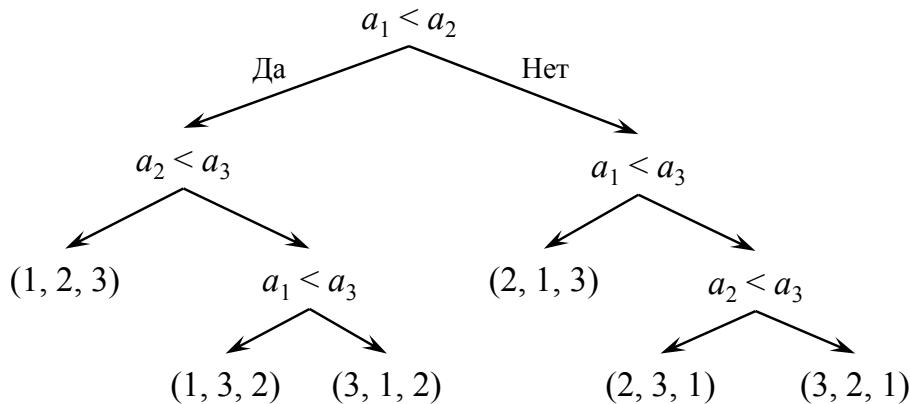


Рис. 3.3. Дерево решения алгоритма сортировки вставкой (для $n = 3$).

Следующее утверждение дает оценку снизу времени работы любого алгоритма сортировки, основанного на сравнениях.

Утверждение 1. Любой алгоритм сортировки сравнением в худшем случае требует выполнения $\Theta(n \log n)$ сравнений.

Доказательство. Процесс работы любого алгоритма сортировки сравнением соответствует движению по его дереву решений от корня к листу. При достижении листа получаем упорядоченную последовательность. Высота h дерева решений – это максимальная длина пути от корня до листа. Она соответствует максимальному числу сравнений, которые выполняет алгоритм. Нас интересует оценка снизу для h . Каждому листу дерева соответствует некоторая перестановка n элементов исходной последовательности. Известно, что число перестановок порядка n равно $n!$. С другой стороны, в полном бинарном дереве число листьев не превышает значения 2^h . Следовательно

$$n! \leq 2^h,$$

$$h \geq \log_2(n!).$$

Используя формулу Стирлинга для факториала (см. приложение), получаем

$$h \geq \log_2 \left(\sqrt{2\pi n} \left(\frac{n}{e} \right)^n \right) = \log_2 \left(\sqrt{2\pi n} \right) + n \log_2 \left(\frac{n}{e} \right) = \Theta(n \log n).$$

Из этого утверждения следует, что не существует алгоритма сортировки сравнением, который в худшем случае будет работать быстрее, чем за время $\Theta(n \log n)$. ■

Пирамидальная сортировка и сортировка слиянием являются *асимптотически оптимальными сортировками сравнением*. Так, время их работы $O(n \log n)$ в худшем случае совпадает с теоретической нижней границей времени работы алгоритмов сортировки такого типа.

3.6. Быстрая сортировка

Быстрая сортировка (quicksort) является одним из самых распространенных на практике алгоритмов внутренней сортировки. В среднем случае он работает за время $\Theta(n \log n)$ и требует порядка $\Theta(\log n)$ ячеек памяти на поддержание рекурсивных вызовов. Однако алгоритм не обладает свойством устойчивости и в худшем случае требует выполнения порядка $\Theta(n^2)$ операций. Быстрая сортировка относится к рекурсивным алгоритмам сортировки сравнением и основана на методе декомпозиции (decomposition). Ниже приведен псевдокод функции `QUICKSORT`, которая упорядочивает подмассив $A[low..high]$. Первоначальный вызов этой функции имеет вид `QUICKSORT(A[1..n], 1, n)`.

При вызове функции `QUICKSORT` сортируемый массив $A[low..high]$ из $high - low + 1$ элементов *разбивается* (*partition*) на две части. Логической границей разбиения служит *опорный элемент* (*pivot*) с некоторым номером p . Первая часть (левая) содержит элементы $A[low..p - 1]$, вторая часть (правая): $A[p + 1..high]$. Причем, каждый элемент первой части $A[low..p - 1]$ должен быть меньше или равен $A[p]$, а каждый элемент из $A[p + 1..high]$ должен быть больше или равен $A[p]$. Выполнение последнего условия достигается путем перестановки элементов $A[low..high]$. Заметим, что число элементов в левом и правом подмассивах заранее неизвестно и зависит от выбора опорного элемента (рис. 3.4). Выбранный опорный элемент $A[p]$ далее в процессе сортировки не участвует и не меняет своего положения в массиве.

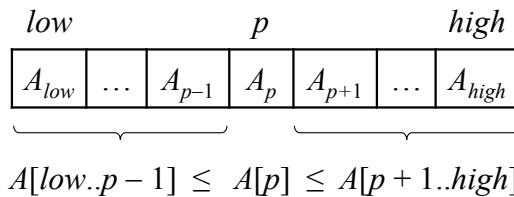


Рис. 3.4. Алгоритм `QUICKSORT`: разбиение на две части массива $A[low..high]$ опорным элементом с номером p .

Полученные подмассивы $A[low..p - 1]$ и $A[p + 1..high]$ рекурсивно упорядочиваются. Рекурсивный спуск и разбиение частей завершается, если в текущем подмассиве $A[low..high]$ осталось менее двух элементов.

Разбиение массива $A[low..high]$ реализуется функцией `PARTITION`, которая должна выбрать опорный элемент и путем перестановки элементов

получить левую и правую части: $A[low..p - 1]$ и $A[p + 1..high]$ такие, что

$$A[low..p - 1] \leq A[p] \leq A[p + 1..high].$$

Алгоритм 3.3. Быстрая сортировка

```

1 function QUICKSORT( $A[1..n]$ ,  $low$ ,  $high$ )
2   if  $low < high$  then
3      $p = \text{PARTITION}(A, low, high)$ 
4     QUICKSORT( $A, low, p - 1$ )
5     QUICKSORT( $A, p + 1, high$ )
6   end if
7 end function

8 function PARTITION( $A[1..n]$ ,  $low$ ,  $high$ )
9    $pivot = A[high]$                                 /* Выбрали опорный элемент */
10   $j = low$                                      /* Правая граница левой части */
11  for  $i = low$  to  $high - 1$  do
12    if  $A[i] \leq pivot$  then
13      SWAP( $A[i], A[j]$ )                         /* Меняем местами */
14       $j = j + 1$                                 /* Перенесли ключ в левую часть */
15    end if
16  end for
17  SWAP( $A[j], A[high]$ )                         /* Номер опорного элемента */
18  return  $j$ 
19 end function
```

В нашей реализации в качестве опорного элемента выбирается $A[high]$. Другие стратегии выбора опорного элемента мы обсудим ниже. Далее, в цикле начинаем двигаться по массиву $A[low..high]$ слева направо. Переменная j служит нам правой границей левого подмассива. Каждый ключ $A[i]$, который меньше или равен опорному элементу $pivot$, перемещается в левый подмассив на место ключа $A[j]$. Переменную j увеличиваем на единицу – смещаем границу подмассива вправо. Таким образом после выполнения каждой итерации цикла имеем

$$A[low..j - 1] < pivot.$$

Остальные элементы $A[j..i]$, которые мы проверили, больше или равны опорному элементу:

$$A[j..i] \geq pivot.$$

Элементы $A[i + 1..high - 1]$ мы еще не проверили и ничего о них не знаем.

После окончания цикла для элементов массив будет справедливо

$$A[low..j - 1] \leq pivot \leq A[j..high].$$

После завершения прохода по массиву меняем местами опорный элемент $A[high]$ и $A[j]$. Это не нарушит приведенных выше неравенств. Возвращаем в функцию `QUICKSORT` значение j как номер опорного элемента (граница разбиения).

На рис. 3.4 показан пример разбиения функцией `PARTITION`($A, 1, 7$) массива из семи элементов. В процессе прохода по массиву выполнены следующие перестановки элементов: $A[2]$ и $A[1]$, $A[3]$ и $A[2]$, $A[4]$ и $A[3]$. После окончания цикла опорный элемент $A[7]$ и ключ $A[4]$ поменяны местами. Результат работы функции – это номер 4 опорного элемента.

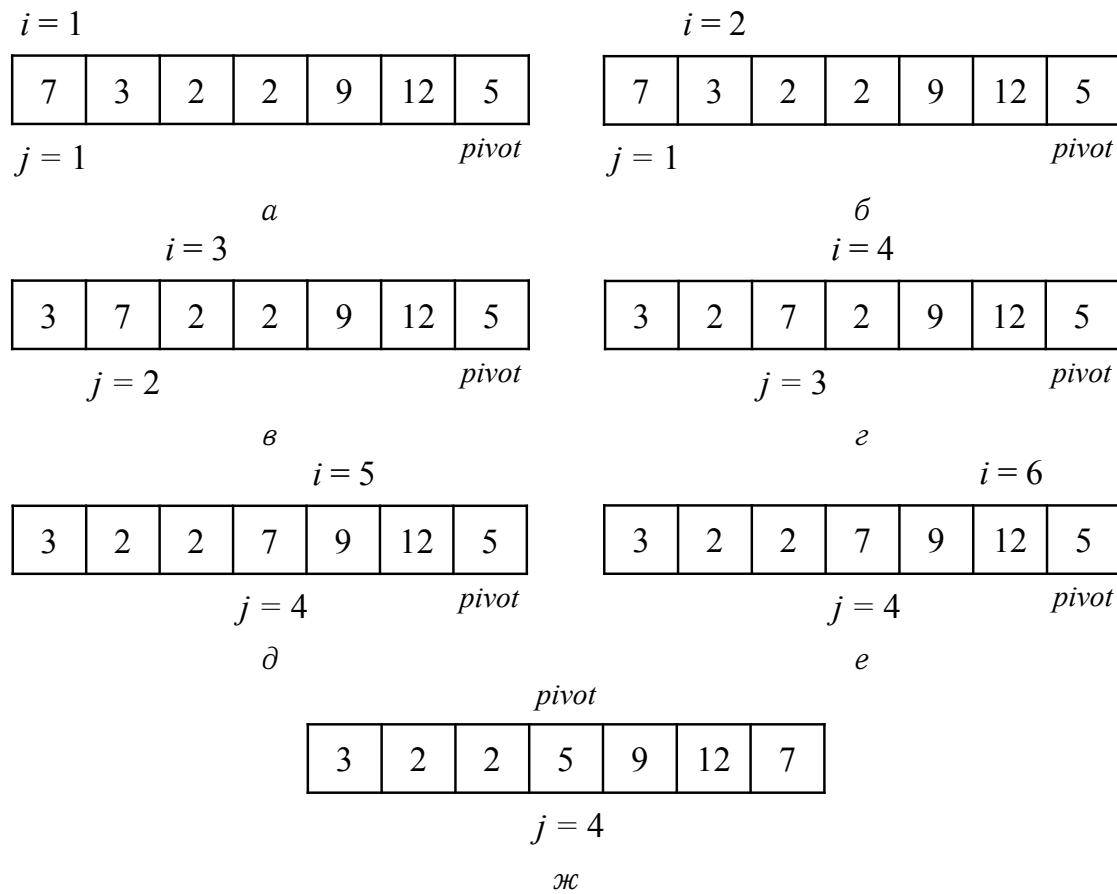


Рис. 3.5. Процесс разбиения массива $A[1..7]$ на две части относительно опорного элемента $A[7] = 5$:

$a - e$ – состояние массива перед выполнением итераций цикла с первой по шестую; ж – опорный элемент перенесен в ячейку $A[4]$ (итоговое разбиение).

Количество операций функции `PARTITION` линейно зависит от длины массива $A[low..high]$. В цикле проверяются все элементы последовательности, за исключением последнего ключа. Это требует порядка $\Theta(n)$ сравнений и перестановок ключей, где n – длина массива.

3.6.1. Эффективность быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от того, насколько сбалансированные разбиения (близкие по размеру подмассивы) формирует процедура PARTITION. Определяющим фактором здесь является стратегия выбора опорного элемента.

Выбор в качестве опорного элемента первого $A[low]$ или последнего $A[high]$ элементов массива может привести на почти упорядоченных последовательностях к возникновению худшего случая. Этой ситуации можно избежать, если взять в качестве опорного элемента случайный ключ $A[RANDOM(low, high)]$ или центральный элемент $A[(high+low)/2]$. На практике хорошо зарекомендовал себя подход, когда в качестве опорного элемента берется медиана из трех значений ключей: первого $A[low]$, центрального $A[(high + low)/2]$ и последнего $A[high]$.

Лучший случай. В лучшем случае опорный элемент разбивает массив $A[low..high]$ на две максимально равные части по $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil - 1$ элементов в каждой. В этой ситуации время выполнения алгоритма описывается следующим рекуррентным уравнением декомпозиции

$$T(n) = 2T(n/2) + \Theta(n).$$

Тогда по основной теореме и учитывая, что $a = 2$, $b = 2$, $f(n) = \Theta(n)$ и $d = 1$, имеем

$$T(n) = \Theta(n \log n).$$

Худший случай. При каждом вызове функции PARTITION в качестве опорного элемента выбирается наибольший или наименьший элемент массива $A[low..high]$. Это приводит к несбалансированному разбиению: одна часть будет пустой, а вторая будет содержать $high - low$ элементов. Тогда время выполнения алгоритма записывается в следующем виде

$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n).$$

Проведем анализ дерева рекурсивных вызовов алгоритма быстрой сортировки для худшего случая (рис. 3.6).

В общем случае на уровне i имеется массив из $n - i$ ключей. Для его разбиения требуется выполнить $n - i$ операций перестановки ключей. Разбиения происходят вплоть до массивов длины 2. Учитывая последнее получаем сумму операций по всем узлам дерева

$$T(n) = n + (n - 1) + \dots + 2 = \Theta(n^2).$$

Средний случай. В среднем случае время работы алгоритма быстрой сортировки равно $\Theta(n \log n)$. Доказательство этого можно найти в [1].

Алгоритм быстрой сортировки был предложен в 1959 г. Ч. Хоаром

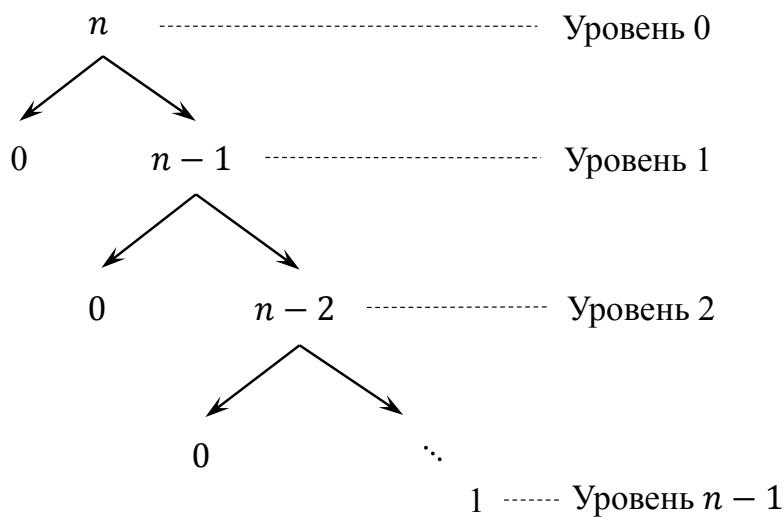


Рис. 3.6. Дерево рекурсивных вызовов быстрой сортировки (худший случай).

(С. А. R. Hoare) в ходе его стажировки в Московском государственном университете. В то время Ч. Хоар работал над проблемой машинного перевода. В его распоряжении был Русско-английский словарь, который хранился на магнитной ленте. Быстрая сортировка была создана им для упорядочивания слов, которые требуют своего перевода, путем поиска в имеющемся словаре.

3.7. Сортировка слиянием

Сортировка слиянием (merge sort) – это асимптотически оптимальный алгоритм сортировки сравнением. В худшем случае он требует выполнения $\Theta(n \log n)$ операций. Сложность алгоритма по памяти равна $\Theta(n)$. Устойчивость сортировки зависит от конкретной реализации алгоритма.

Подробное описание и анализ эффективности сортировки слиянием приведены в разделе 2.2. Отметим лишь, что алгоритм может быть использован для упорядочивания линейных структур данных с последовательным доступом, например, связных списков. К его недостаткам можно отнести то, что он требует дополнительной памяти для слияния массивов и для сортировки почти упорядоченных последовательностей требует столько же времени, сколько для случайных массивов.

Алгоритм был предложен в 1945 г. Джоном фон Нейманом (John von Neumann) [5].

3.8. Пирамидальная сортировка

Пирамидальная сортировка (heapsort) – это асимптотически оптимальный алгоритм сортировки сравнением. В худшем случае он требует выполнения $\Theta(n \log n)$ операций и не требует дополнительной памяти, но не обеспечивает устойчивости (non-stable sort).

Описание пирамидальной сортировки приведено в разделе 14.3. Заметим, что алгоритм неприменим для упорядочивания линейных структур данных с последовательным доступом, например, связных списков. К недостаткам также можно отнести и то, что алгоритм требует для сортировки почти упорядоченных последовательностей столько же времени, сколько для случайных массивов.

Впервые алгоритм был опубликован в 1964 г. Джоном Уильямсом (J.W.J. Williams).

3.9. Сортировка подсчетом

Сортировка подсчетом (counting sort) относится к семейству алгоритмов целочисленной сортировки (integer sort). В процессе своей работы алгоритм не использует операцию сравнения ключей. Сфера его применения – это упорядочивание последовательностей неотрицательных целых чисел, значения которых достаточно малы по сравнению с размером n массива. Например, сортировка 100 000 000 целых чисел, значения которых не превышают 50 000.

Известно несколько модификаций алгоритма сортировки подсчетом. Мы рассмотрим его устойчивую версию. Будем считать, что входной массив состоит из n целых чисел из множества $\{0, 1, \dots, k\}$, где k – максимальное значение во входном массиве (или оценка сверху для него). Оно может быть найдено перед выполнением сортировки за время $\Theta(n)$.

Ниже приведен псевдокод алгоритма COUNTINGSORT, в котором используется два вспомогательных массива: $C[0..k]$ и $B[0..n - 1]$. Массив $C[0..k]$ используется для подсчета числа появлений каждого значения в исходном массиве $A[0..n - 1]$, а массив $B[0..n - 1]$ – это результат сортировки.

В двух первых циклах (строки 2–7) вычисляется, сколько раз каждый элемент множества $\{0, 1, \dots, k\}$ присутствует в сортируемом массиве $A[0..n - 1]$. В результате элемент $C[i]$ содержит число появлений значения i в исходном массиве. На рис. 3.7, б приведен пример состояния массива $C[0..k]$ после выполнения первых двух циклов.

Если начать просматривать значения полученного массива $C[0..k]$ слева направо и записывать индексы i ненулевых элементов $C[i]$, то получим упорядоченную последовательность элементов исходного массива. Например, для массива на рис. 3.7, а получим: 1, 1, 2, 2, 3, 3, 4, 5. Однако такой

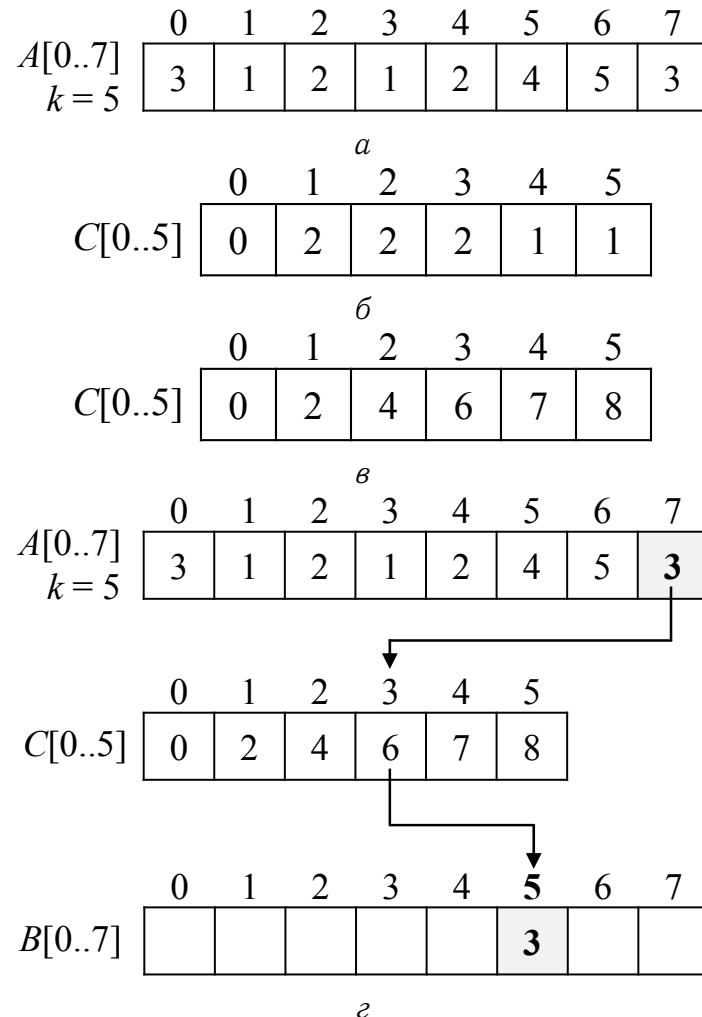


Рис. 3.7. Сортировка подсчетом массива из 8 элементов ($n = 8$, $k = 5$):
а – состояние массива $A[0..7]$ перед сортировкой; *б* – состояние массива $C[0..5]$ после выполнения первых двух циклов; *в* – состояние массива $C[0..5]$ после выполнения третьего цикла; *г* – запись элемента $A[7]$ на свое место $B[5]$ в отсортированном массиве.

подход не обеспечивает устойчивости, поскольку элементы с равными значениями могут быть переставлены местами. Для обеспечения устойчивости в нашем алгоритме присутствуют следующие два цикла.

Третий цикл (строки 8–10) записывает в каждую ячейку $C[i]$ число элементов, не превышающих значение i . Далее в последнем цикле используем этот массив для помещения элементов $A[i]$ в свои позиции упорядоченного массива $B[0..n - 1]$. Например, на рис. 3.7, г $A[7]$ помещается в ячейку $B[5]$, $A[6]$ в $B[7]$, $A[5]$ в $B[6]$, ..., $A[0]$ в $B[4]$.

Время выполнения четырех циклов равно, соответственно, $\Theta(k)$, $\Theta(n)$, $\Theta(k)$ и $\Theta(n)$. Следовательно, суммарное время выполнения алгоритма равно $\Theta(k + n)$. Алгоритм требует использования дополнительных массивов C и B , отсюда сложность алгоритма по памяти $\Theta(k + n)$.

Если минимальное значение min в массиве больше нуля, то при работе

Алгоритм 3.4. Сортировка подсчетом

```

1 function COUNTINGSORT( $A[0..n - 1]$ ,  $B[0..n - 1]$ ,  $k$ )
2   for  $i = 0$  to  $k$  do
3      $C[i] = 0$ 
4   end for
5   for  $i = 0$  to  $n - 1$  do
6      $C[A[i]] = C[A[i]] + 1$ 
7   end for
8   for  $i = 1$  to  $k$  do
9      $C[i] = C[i] + C[i - 1]$ 
10  end for
11  for  $i = n - 1$  to 0 do
12     $C[A[i]] = C[A[i]] - 1$ 
13     $B[C[A[i]]] = A[i]$ 
14  end for
15 end function

```

с массивом $C[0..k]$ из каждого значения $A[i]$ следует вычитать \min , а при обратной записи прибавлять. При наличии отрицательных чисел нужно при работе с массивом $C[0..k]$ к $A[i]$ прибавлять $|\min|$, а при обратной записи вычесть.

3.10. Выбор алгоритма сортировки

Выбор алгоритма сортировки обусловлен рядом ограничений, которые накладывает целевая вычислительная система, тип данных ключей и размер сортируемого массива. На практике можно руководствоваться следующей пошаговой процедурой выбора алгоритма сортировки.

Шаг 1. Если сортируемый массив помещаются в оперативную память целевой системы, то переходим к рассмотрению алгоритмов *внутренней сортировки* (internal sorting). В противном случае выбор падает на методы *внешней сортировки* (external sorting).

Шаг 2. Если в ходе сортировки нельзя использовать дополнительный объем памяти, зависящий от длины массива, то ограничиваемся рассмотрением лишь алгоритмов сортировки *на месте* (in-place sorting).

Шаг 3. Если необходимо выполнять устойчивую сортировку, то рассматриваем только алгоритмы *устойчивой сортировки* (stable sorting).

Шаг 4. Если тип данных ключа – целое число из ограниченного диапазона, то включаем в рассмотрение алгоритмы *целочисленной сортировки* (integer sort). В противном случае принимаем во внимание только алгоритмы *сортировки сравнением* (comparision sort).

Шаг 5. Из полученного множества выбираем алгоритм с наименьшей

вычислительной сложностью (см. табл. 3.1) для среднего случая или худшего случая (если вероятность его возникновения высока).

На последнем этапе выбора алгоритма важно учесть и то, насколько эффективно реализуются алгоритмы на имеющейся вычислительной системе (эффективно используют процессор, память, дисковую подсистему). Здесь выбор в пользу одного или другого алгоритма может быть сделан путем серии экспериментов по сортировке реальных наборов данных.

Рассмотрим пример. Пусть нам требуется разработать программу, в которой необходимо периодически выполнять сортировку таблицы, содержащей фамилии и другие данные участников зимних Олимпийских игр. Будем считать, что фамилия спортсмена не превышает 128 символов английского алфавита, а данные о нем занимают 1024 байт. В среднем число участников Олимпийских игр не превышает 15 000 человек. Тогда вся таблица потребует примерно $15\,000 \cdot (128 + 1024) < 17$ Мбайт памяти и легко поместится в оперативную память обычного компьютера. Следовательно, остановим наш выбор на алгоритмах внутренней сортировки. Так как сортируемая таблица занимает относительно немного места в оперативной памяти, то будем рассматривать и алгоритмы, использующие дополнительную память (например, MERGESORT и COUNTINGSORT). В нашем примере требование устойчивости сортировки не является обязательным, поэтому будем рассматривать и алгоритмы неустойчивой сортировки (QUICKSORT и HEAPSORT). Тип данных ключа сортировки – это строка (фамилия). Наш выбор ограничивается только алгоритмами сортировки, основанными на сравнениях: сортировка слиянием, быстрая сортировка, пирамидальная сортировка, сортировка вставкой и др. Из полученного множества выбираем алгоритм с наименьшей вычислительной сложностью в среднем случае – быстрая сортировка, которая в среднем работает за время $\Theta(n \log n)$.

3.11. Упражнения

1. Проиллюстрируйте процесс упорядочивания последовательности чисел (20, 4, 45, 1, 2, 6, 88) алгоритмом сортировки вставкой.
2. Модифицируйте функцию INSERTIONSORT для сортировки ключей по невозрастанию.
3. Проведите анализ эффективности алгоритма сортировки вставкой для лучшего случая и покажите, что в этом случае вычислительная сложность алгоритма равна $\Theta(n)$.
4. Выполните анализ вычислительной сложности алгоритма сортировки вставкой для лучшего случая при условии, что элементы массива в цикле *while* перебираются не справа налево, а слева направо.
5. Оцените вычислительную сложность алгоритма сортировки выбором для лучшего случая.

6. Выполните анализ вычислительной сложности алгоритма сортировки вставкой для среднего случая.

7. По аналогии с примером на рис. 3.3 постройте дерево принятия решения алгоритма сортировки выбором для $n = 3$.

8. Разработайте функцию, выбирающую в качестве опорного элемента быстрой сортировки медиану трех ключей: первого $A[low]$, центрального $A[(high + low)/2]$ и последнего $A[high]$.

9. Разработайте алгоритм поиска медианы для произвольного массива $A[1..n]$. Оцените его вычислительную сложность.

10. Проанализируйте поведение алгоритма QUICKSORT в случае, если на его вход поступил массив из n одинаковых ключей.

4. Поиск

4.1. Задача поиска

Имеется последовательность (a_1, a_2, \dots, a_n) из n элементов и задан способ проверки пары элементов на равенство $(a_i = a_j)$. Требуется определить номер элемента последовательности, который имеет заданное значение key .

4.2. Линейный поиск

Алгоритм *линейного поиска* (linear search) последовательно проверяет все элементы массива $a[1..n]$ на равенство с заданным значением. Работа алгоритма прерывается при обнаружении первого совпадения. Ниже приведен псевдокод алгоритма LINEARSEARCH.

Алгоритм 4.1. Линейный поиск

```
1 function LINEARSEARCH( $a[1..n]$ ,  $key$ )
2     for  $i = 1$  to  $n$  do
3         if  $a[i] = key$  then
4             return  $i$ 
5         end if
6     end for
7     return  $-1$ 
8 end function
```

В худшем случае искомое значение отсутствует в массиве или расположено в последней ячейке. В этом случае алгоритм выполняет порядка $\Theta(n)$ операций. Подробный анализ эффективности алгоритма приведен в разделе 1.3.

4.3. Бинарный поиск

Если последовательность упорядочена по неубыванию или по невозрастанию, то задачу поиска можно решить за время $O(\log n)$ алгоритмом *бинарного поиска* (двоичного поиска, binary search).

Ниже приведен псевдокод алгоритма BINARYSEARCH. Предполагается, что входной массив $a[1..n]$ упорядочен по неубыванию

$$a[1] \leq a[2] \leq \cdots \leq a[n].$$

Алгоритм 4.2. Бинарный поиск

```

1 function BINARYSEARCH( $a[1..n]$ ,  $key$ )
2    $low = 1$ 
3    $high = n$ 
4   while  $low \leq high$  do
5      $mid = (low + high)/2$ 
6     if  $a[mid] = key$  then
7       return  $mid$ 
8     else if  $key > a[mid]$  then
9        $low = mid + 1$ 
10    else
11       $high = mid - 1$ 
12    end if
13  end while
14  return  $-1$ 
15 end function
```

В ходе своей работы алгоритм BINARYSEARCH поддерживает две переменные low и $high$ – это границы подмассива, в котором происходит поиск элемента key . На каждой итерации цикла осуществляется проверка, находится ли искомый элемент в центре текущего подмассива. Учитывая, что массив упорядочен, мы знаем, с какой стороны от центрального элемента $a[mid]$ может находиться ключ (или отсутствовать в массиве). Если ключ меньше центрального элемента, то переходим к поиску в левом подмассиве (сдвигаем границу $high$); если в правом – переходим к правому подмассиву (сдвигаем границу low). Итерации повторяются до тех пор, пока мы не закончим обработку подмассива из одного элемента.

Оценим вычислительную сложность алгоритма. В худшем случае на каждой итерации осуществляется вычисление индекса mid центрального элемента. Это требует трех операций. Далее производится вычисление условий в строках 5 и 6, которые в худшем случае не выполняются, и происходит смещение правой границы $high$ подмассива (строка 10). Таким образом на каждой итерации цикла в худшем случае выполняется константное число с операций. Осталось вычислить количество итераций цикла *while*. На первой итерации длина подмассива равна n , на второй – $n/2$, на третьей – $n/4$ и т. д., пока на последней итерации k не будет обработан подмассив длины 1:

$$n, \quad \frac{n}{2^1}, \quad \frac{n}{2^2}, \quad \dots \quad \frac{n}{2^k} = 1,$$

из последнего равенства находим k :

$$k = \log_2 n.$$

Учитывая константное число операций на инициализацию и корректировку переменных *low* и *high*, получаем оценку вычислительной сложности алгоритма BINARYSEARCH для худшем случая $O(\log n)$.

4.4. Упражнения

1. Оцените вычислительную сложность алгоритма линейного поиск для среднего случая.
2. Оцените вычислительную сложность алгоритма бинарного поиска для лучшего случая.
3. Разработайте версию алгоритма бинарного поиска, который не запускает цикл *while*, если искомый элемент отсутствует в массиве.
4. Проверьте корректность работы рассмотренного алгоритма бинарного поиска в случае: пустого входного массива ($n = 0$), отсутствия искомого ключа в массиве, если искомый элемент находится в ячейке $a[1]$ или $a[n]$.
5. При вычислении центрального элемента $mid = (low + high)/2$ возможно переполнение. Например, значения *low* и *high* помещаются в тип *int*, а их сумма *low + high* – нет. Разработайте версию алгоритма бинарного поиска, устойчивого к переполнению при вычислении значения переменной *mid*.
6. Разработайте версию алгоритма бинарного поиска для случая, когда входной массив упорядочен по неубыванию.

5. Абстрактные типы данных

5.1. Базовые типы данных

Для обработки информации различного вида, например целых и вещественных чисел, логических значений и последовательностей символов, в языках программирования вводят понятие *типа данных* (data type). Тип данных – это неотъемлемый атрибут любого значения, которое встречается в программе. Каждый тип данных определяет две характеристики:

- множество допустимых значений, которые могут принимать данные, принадлежащие к этому типу;
- набор операций, которые можно выполнять над данными.

Все типизированные языки программирования (C, C++, C#, Go, Java, Python, PHP и др.) поддерживают набор *базовых (примитивных) типов данных*. Например, тип целых чисел (*integer*), тип вещественных чисел (*real*, *float*), логический тип (*boolean*), символьный тип (*char*) и ссылочный тип (*pointer*).

Система типов языка программирования относит каждое вычисленное в программе значение к определенному типу данных и отслеживает корректность выполнения операций над ними. Например, на этапе компиляции программы или ее выполнения могут быть обнаружены ошибки такого рода, как деление строкового значения на целое число и др.

Помимо базовых типов язык программирования может поддерживать *составные типы данных* (агрегатные, структурные, композитные типы) – это типы данных, которые формируются на основе базовых. Например, массивы, структуры и классы в языках С и С++ относят к составным типам данных.

Примером широко используемого на практике составного типа данных может служить *одномерный массив* (array) – совокупность элементов одного типа, размещенных в непрерывном участке памяти. Непрерывное размещение позволяет за константное время отыскивать по номеру элемента его расположение в памяти. Номер элемента массива также называют его *индексом* (index, subscript). Как правило, нумерация элементов начинается с 0, однако в таких языках программирования, как Pascal, можно в явном виде задавать начальный и конечный индексы элементов массива. Пусть b – адрес первого элемента одномерного массива, а s – размер типа данных элементов в байтах. Тогда адрес элемента с индексом $i \in \{0, 1, 2, \dots\}$

можно вычислить по следующей формуле:

$$b + s \cdot i.$$

Ниже приведен фрагмент программы на языке С, в которой создается массив из четырех элементов:

```
int a[4] = {50, 60, 70, 80};
```

В табл. 5.1 показано непрерывное размещение массива *a* в памяти, начиная с адреса 100. Элементы массива имеют тип `int`, размер которого полагается равным 4 байтам. Адрес элемента с индексом 0 равен 100, адрес элемента с индексом 1 равен 104, адрес элемента с индексом 2 равен 108 и т. д. Для получения адреса элемента с индексом *i* достаточно вычислить $100 + 4i$. Последнее требует выполнения всего двух операций, что и обеспечивает константное время доступа к элементу массива по его индексу.

Таблица 5.1. Массив из четырех элементов типа `int`

Индекс:	0	1	2	3
Значение:	50	60	70	80
Адрес:	100	104	108	112

На практике для доступа к определенному элементу массива программисту не требуется в явном виде вычислять его адрес в памяти. Эту задачу берет на себя компилятор или интерпретатор языка программирования, который поддерживает различные способы доступа к элементам массива по индексу, например, через индексные скобки `a[i]`.

5.2. Структуры данных

Структуры и классы в языках С, С++, С#, Java позволяют определять *пользовательские типы данных* (user-defined data type). Операции над такими типами реализуются программистом через набор функций или методов класса. Такая программная единица, реализующая хранение и выполнение операций над совокупностью однотипных элементов, называется *структурой данных* (data structure). Набор функций (методов) для выполнения операций над структурой данных называется ее *интерфейсом* (interface).

Примером структуры данных служит *односвязный список*. В каждом узле списка хранятся данные и указатель на следующий элемент. Ниже приведен пример интерфейса связного списка на языке С. Интерфейс

включает описание типа данных каждого узла – структура `listnode`, а также прототипы функций добавления элемента в начало списка, поиска элемента по ключу и удаление узла. В зависимости от решаемой задачи в интерфейс могут входить и другие функции, например, добавление элемента в конец списка, сортировка списка и т. д. Подробное рассмотрение связных списков приводится в разделе 6.3.

```
/* Узел односвязного списка */
struct listnode {
    int value;                      /* Данные узла */
    struct listnode *next;           /* Указатель на следующий узел */
};

/* list_addfront: Добавляет узел в начало списка list */
struct listnode *list_addfront(struct listnode *list, int value);

/*
 * list_lookup: Находит в списке list узел с заданным
 *               значением value
 */
struct listnode *list_lookup(struct listnode *list, int value);

/*
 * list_delete: Удаляет из списка list узел с заданным
 *               значением value
 */
struct listnode *list_delete(struct listnode *list, int value);
```

5.3. Абстрактные типы данных

Наряду с понятием структуры данных мы будем рассматривать абстрактные типы данных. *Абстрактный тип данных* (АТД, abstract data type) – это тип данных, который задан только описанием своего интерфейса. Способ хранения данных в памяти компьютера и алгоритмы работы с ними скрыты внутри функций интерфейса. Именно в этом и заключается суть абстракции. Если детально описать реализацию функций АТД, то мы получим конкретную *структурку данных*.

Поясним на примере различие между АТД и структурой данных. Абстрактный тип данных *список* (`list`) может быть реализован с помощью двух разных структур данных: статического массива или связного списка. Каждая реализация определяет один и тот же набор функций: добавить элемент, найти элемент, удалить и т. д. Результат работы функций в обеих реализациях совпадает, но не эффективность – вычислительная сложность и сложность по памяти функций разных реализаций могут быть различными.

На практике при разработке алгоритмов широко используется ограниченный набор абстрактных типов данных:

- список (list);
- стек (stack);
- очередь (queue);
- дек (dequeue);
- множество (set);
- ассоциативный массив (associative array, map);
- очередь с приоритетом (priority queue);
- граф (graph).

Абстрактные типы данных удобно использовать при высокоуровневом описании алгоритмов, не вдаваясь в подробности реализации операций АТД. Заметим, что стандартные библиотеки высокоуровневых языков программирования включают реализацию основных АТД. Например, в стандартные библиотеки языков C++, C#, Java входят классы для работы со списками, множествами, ассоциативными массивами и очередями с приоритетом.

Можно провести классификацию абстрактных типов данных по различным критериям. Все АТД можно разделить на линейные и нелинейные типы. Подробное описание рассматриваемых далее АТД будет дано в последующих главах. Здесь же мы проведем экспресс-обзор: рассмотрим определения и базовые операции основных абстрактных типов данных.

5.3.1. Линейные типы данных

В *линейных типах данных* (linear data types) между элементами присутствуют линейные логические связи – они связаны отношением «следует за» или «предшествует». Линейные типы данных можно разделить на типы с прямым доступом и последовательным.

Типы данных с *прямым доступом* (direct access) позволяют за константное время обращаться к элементу по его номеру (индексу). Основным представителем класса линейных АТД с прямым доступом является *массив* (array). В типах с *последовательным доступом* (sequential access) для поиска элемента по его индексу необходимо последовательно пройти все элементы начиная с первого. К этому классу абстрактных типов данных относят списки, стеки, очереди и деки (рис. 5.1). Они отличаются дисциплиной добавления и удаления элементов из них.

Операции INSERT, PUSH, ENQUEUE, PUSHBACK и PUSHFRONT осуществляют добавление элементов в соответствующие АТД. Функции DELETE, POP, DEQUEUE, POPBACK, POPFRONT реализуют удаление элемента из АТД. Доступ к элементу без его удаления выполняется операциями LOOKUP,

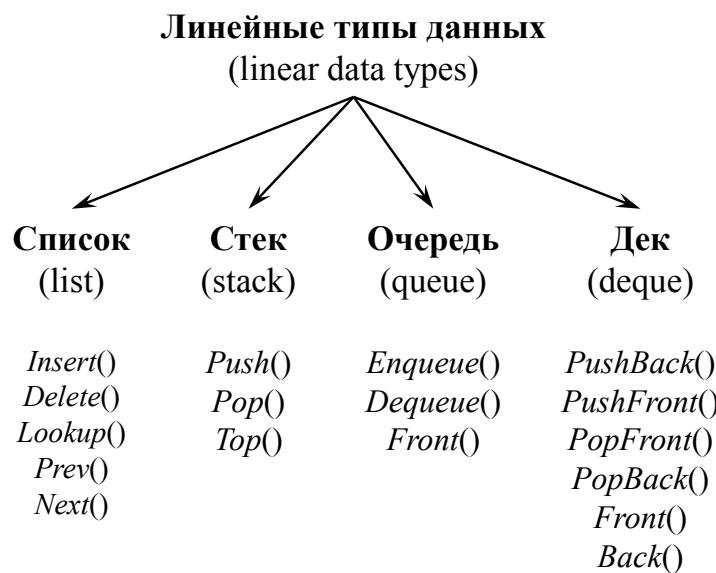


Рис. 5.1. Линейные типы данных с последовательным доступом (курсивом показаны основные операции указанных типов).

TOP, FRONT и BACK.

Список. Абстрактный тип данных *список* (list) реализует хранение набора однотипных элементов (рис. 5.2). С каждым элементом ассоциирован его номер в списке – *индекс* (index), а также некоторое *значение* (value). Между элементами установлено отношение «следует за», которое позволяет для любого узла определить элемент, следующий за ним и предшествующий ему (операции NEXT и PREV, от англ. previous – предыдущий). Допускается вставка и удаление элемента в/из любой позиции списка (операции INSERT и DELETE). Поиск элемента (LOOKUP) осуществляется по его индексу или значению.

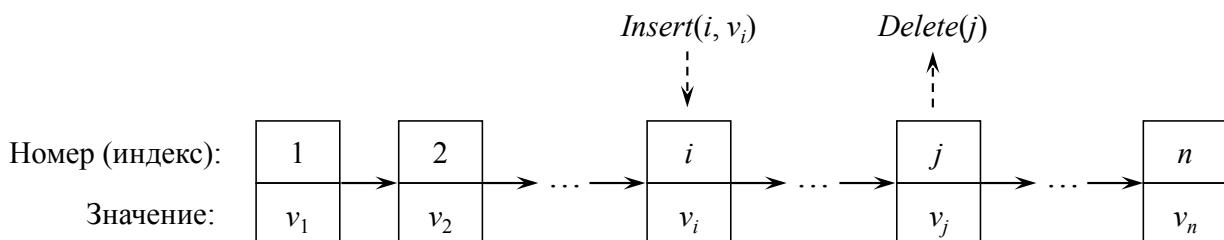


Рис. 5.2. АТД список (стрелками показано отношение «следует за»).

Стек. *Стек* (stack) – это абстрактный тип данных, представляющий собой список, в котором вставка (PUSH) и удаление элементов (POP) выполняется с одного конца – *вершины стека* (top). Стек функционирует по принципу «последним пришел – первым вышел» (Last In – First Out, LIFO). В качестве примера можно привести стопку тарелок – очередная тарелка всегда кладется сверху, взять тарелку из стопки можно только сверху (рис. 5.3).

Очередь. *Очередь* (queue) – это абстрактный тип данных, представляющий собой список, в котором вставка (ENQUEUE) и удаление элементов

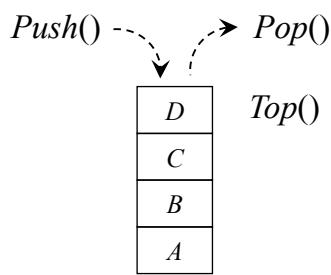


Рис. 5.3. АТД стек.

(DEQUEUE) выполняется с разных концов. Добавление элементов выполняется с *заднего конца* (tail, rear, back), а удаление – с *переднего* (front, head). Очередь функционирует по принципу «первым пришел – первым вышел» (First In – First Out, FIFO). Доступ к первому элементу очереди выполняется операцией FRONT (рис. 5.4). В качестве примера можно привести бесприоритетную очередь из покупателей в магазине – кто первым встал в очередь, тот первым будет обслужен.

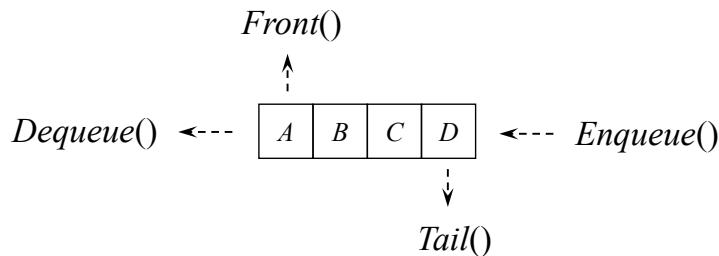


Рис. 5.4. АТД очередь.

Дек. Дек (deque, от англ. double-ended queue) – это абстрактный тип данных, представляющий двустороннюю очередь. Вставка (PUSHBACK, PUSHFRONT) и удаление элементов (POPBACK, POPFRONT) может выполняться с *заднего конца* или *переднего*.

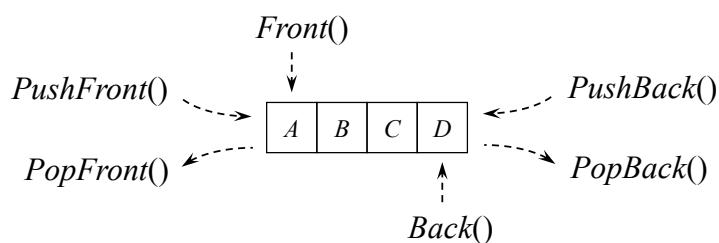


Рис. 5.5. АТД дек.

5.3.2. Нелинейные типы данных

В *нелинейных типах* (non-linear data types), связи между элементами имеют более сложную форму, например, в бинарных деревьях каждый элемент может иметь ноль, один или два дочерних узла. На рис. 5.6 приведены основные нелинейные типы данных.

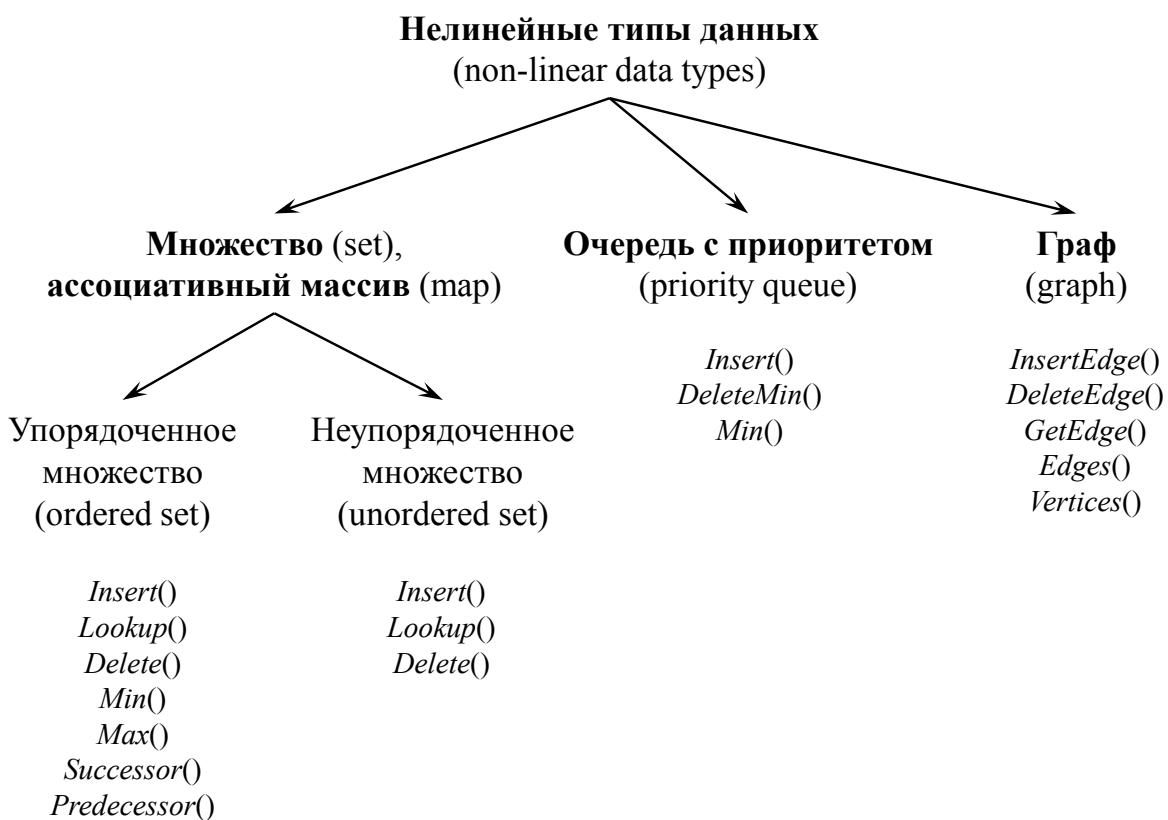


Рис. 5.6. Нелинейные типы данных (курсивом показаны основные операции).

Множество. Абстрактный тип данных *множество* (set) представляет собой совокупность элементов одного типа. Все элементы множества различны. Множество поддерживает операции добавления, удаления и поиска элементов. Множество является важнейшим АТД, на базе которого реализуются другие фундаментальные типы данных, такие как ассоциативные массивы (словари) и очереди с приоритетом.

Ассоциативный массив (associative array, map) – это вид множества, элементами которого являются пары вида «(ключ, значение)». Ассоциативные массивы также называют *словарями* (dictionary). Поиск элемента в ассоциативном массиве осуществляется по *ключу* (key). *Значение* (value), ассоциированное с каждым ключом, может иметь тип данных, отличный от типа данных ключа.

Единственное отличие ассоциативных массивов от множеств в том, что множества хранят только ключи, а ассоциативные массивы с каждым ключом хранят и некоторое значение (данные из предметной области решаемой задачи).

Если между ключами множества или ассоциативного массива задано бинарное отношение *строгого порядка* «<» («меньше чем» или «предшествует»), то множество называется *линейно упорядоченным множеством* (ordered set). По определению бинарное отношение строгого порядка является антирефлексивным, антисимметричным и транзитивным. Упорядочен-

ные множества поддерживают операции поиска экстремальных элементов – элемента с наименьшим значением ключа (операция *MIN*) и наибольшим значением ключа (*MAX*). Кроме того, поддерживаются операции поиска следующего элемента (*SUCCESSOR*) – узла с ключом, ближайшим к ключу заданного узла и превышающим его, а также нахождение предыдущего элемента (*PREDECESSOR*) – узла с ключом, ближайшим к ключу заданного узла и меньшим его. Эти операции позволяют осуществлять упорядоченный перебор элементов множества – от наименьшего ключа к наибольшему и наоборот.

Множество, допускающее присутствие в нем нескольких элементов с одинаковым значением, называется *мультимножеством* (multiset). Ассоциативный массив, допускающий присутствие в нем нескольких пар с одинаковыми ключами, называется *ассоциативным массивом с повторениями* (multimap).

Очередь с приоритетом. *Очередь с приоритетом* (priority queue) – это абстрактный тип данных, представляющий собой упорядоченное множество, элементами которого являются пары вида «(приоритет, значение)». Все элементы очереди упорядочены по *приоритету* (priority). Такая очередь поддерживает две базовые операции – вставка элемента с заданным приоритетом (операция *INSERT*) и удаление элемента с экстремальным значением приоритета: минимальным (*DELETEMIN*) либо максимальным (*DELETEMAX*).

АТД очередь с приоритетом можно реализовать на базе операций упорядоченного ассоциативного массива. В частности, *DELETEMIN* можно реализовать через обращение к операции *MIN* и последующего удаления найденного элемента функцией *DELETE*. Однако такая реализация может быть не всегда эффективной и избыточной (нам не требуются операции *LOOKUP*, *SUCCESSOR*, *PREDECESSOR* упорядоченного множества). Для очередей с приоритетом созданы специальные структуры данных, такие как бинарные и биномиальные кучи.

Граф. *Граф* (graph) – это абстрактный тип данных, представляющий собой совокупность *вершин* (vertices) и *ребер* (edges), связывающих их (рис. 5.7). Граф поддерживает операции добавления нового ребра, заданного парой вершин (*INSERTEDGE*), удаления ребра (*DELETEEDGE*), поиска ребра (*GETEDGE*), а также операции получения информации о числе ребер и вершин в графе (операции *EDGES* и *VERTICES*). Заметим, что это неполный перечень возможных операций. В зависимости от решаемой задачи могут понадобиться операции добавления/удаления вершин, определения связности графа, установления присутствия в графе циклов и др.

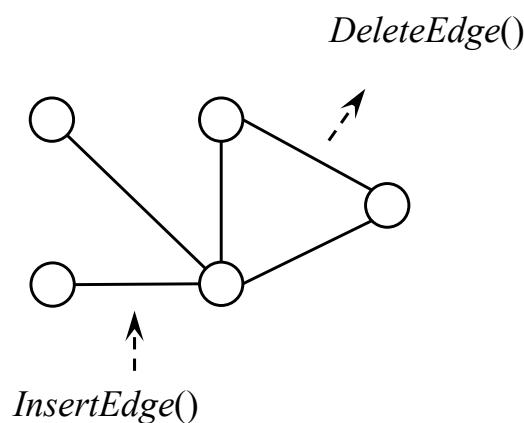


Рис. 5.7. АТД граф.

5.4. Выбор абстрактного типа данных

Одним из важнейших этапов в разработке алгоритма решения задачи является выбор способа организации и хранения данных в памяти. Общий план выбора АТД включает следующие шаги.

1. Выделяется набор операций, которые планируется выполнять над входными данными. Например, вставка, поиск, удаление элементов, поиск элемента с минимальным или максимальным значением ключа.
2. Определяются абстрактные типы данных, поддерживающие требуемые операции. Для этого можно использовать табл. 5.2.

Таблица 5.2. Операции абстрактных типов данных

Операция	Абстрактный тип данных
INSERT()	Список, множество, очередь с приоритетом
DELETE()	Список, множество
LOOKUP()	Список, множество
MIN()	Упорядоченное множество, очередь с приоритетом
MAX()	Упорядоченное множество, очередь с приоритетом
SUCCESSOR()	Упорядоченное множество
PREDECESSOR()	Упорядоченное множество
DELETEMIN()/DELETEMAX()	Очередь с приоритетом

Например, если требуется один раз записать в память большой набор данных, а затем в течение длительного времени многократно осуществлять в нем поиск, то здесь целесообразно использовать АТД, обеспечивающий быстрый поиск элементов. Эффективность других операций нас может не интересовать, так как мы их не используем или обращаемся к ним крайне редко.

3. Анализируется информация о входных данных. Устанавливается тип решаемой задачи и разрабатываемого алгоритма – *offline* или *online*. Алгоритмы типа *offline*, или просто *offline-алгоритмы*, подразумевают, что заранее известен весь набор входных данных. Алгоритмы типа *online*, напротив, допускают динамическую обработку поступающих данных. Примером *online-алгоритма* может служить алгоритм сортировки вставками (*insertion sort*), который для каждого поступающего значения находит его позицию в ранее упорядоченной последовательности элементов.

Если установлено, что заранее известны все входные данные, то некоторые операции АТД могут быть эффективно реализованы с учетом этого. Например, заранее могут быть определены минимальный и максимальные значения или выполнена сортировка набора данных для эффективной реализации операций упорядоченного множества и т. д.

Далее необходимо определяют тип ключ (приоритета) элементов – целое число, строка и т. д. Знание информации о ключе, о его типе и возможных значениях позволяет использовать специализированные структуры данных для реализации АТД. Например, если установлено, что ключом является строка, то одним из возможных вариантов реализации упорядоченного множества будет префиксное дерево (*prefix tree*).

4. Задействуется существующая или разрабатывается новая структура данных для реализации выбранного АТД. На этом этапе учитывают имеющиеся ограничения на объем доступной памяти и требования к вычислительной сложности операций. На рис. 5.8 приведены основные структуры данных для реализации нелинейных типов.

Пример. Процессы операционной системы. Необходимо принять решение о выборе АТД для хранения информации о процессах (программах), выполняющихся операционной системой. Каждый процесс характеризуется приоритетом из множества $\{0, 1, \dots, 139\}$, целочисленным неотрицательным идентификатором из множества $\{0, 1, \dots, 65535\}$ и полями с информацией о ресурсах процесса.

Осуществим выбор АТД, в соответствии с описанной выше схемой.

1. Выделим набор операций, которые планируется выполнять над совокупностью процессов. Нам необходимо добавлять новые процессы в АТД при их запуске, удалять завершившие свое выполнение, отыскивать процесс по его идентификатору, а также перебирать процессы в порядке невозрастания их приоритетов – операции *Max*, *PREDECESSOR*.

2. Выберем АТД для решения нашей задачи. Во-первых, нам требуется иметь возможность быстро находить процесс по его идентификатору (операция *LOOKUP*). Для этого подходит АТД неупорядоченный ассоциативный массив, в котором ключом является идентификатор процесса. Во-вторых, процессы должны быть упорядочены по их приоритетам. Для этого можно использовать АТД упорядоченный ассоциативный массив, в котором ключ – это приоритет процесса. Таким образом, для решения нашей задачи по-

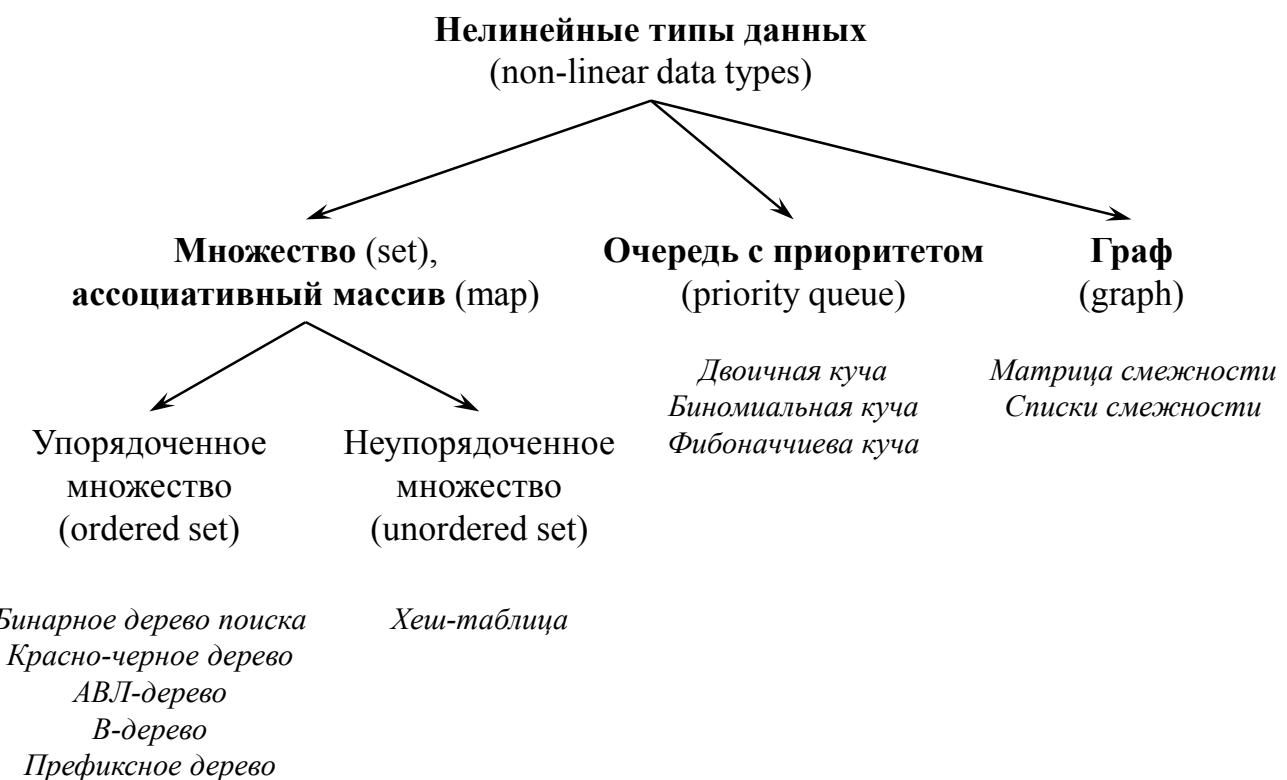


Рис. 5.8. Основные структуры данных для реализации нелинейных типов.

требуется использовать два АТД – неупорядоченный ассоциативный массив по идентификатору процесса и упорядоченный ассоциативный массив по приоритету процесса.

3. Алгоритмы обработки процессов относятся к типу *online* – процессы динамически создаются и удаляются. Оба ключа являются целыми неотрицательными числами из ограниченного диапазона.

4. Выберем структуры данных для реализации АТД (рис. 5.8). Неупорядоченный ассоциативный массив с целочисленным ключом можно эффективно реализовать на базе хеш-таблицы, а для упорядоченного ассоциативного массива с целым ключом можно использовать красно-черное дерево.

Пример. Задача сортировки. Задана последовательность ключей

$$a_1, a_2, \dots, a_n.$$

Требуется упорядочить ее по неубыванию (см. раздел 3) – найти такую перестановку (i_1, i_2, \dots, i_n) ключей, что

$$a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}.$$

Решение 1. Рассмотрим приведенное ниже решение SETSORT на базе упорядоченного мульти множества S (ordered multiset). Мы используем мульти множество, так как некоторые элементы массива могут иметь одинаковые значения.

```

1 function SETSORT( $a[1..n]$ )
2     /* Вставляем элементы в упорядоченное мульти множество */
3     for  $i = 1$  to  $n$  do
4         SETINSERT( $S, a[i]$ )
5     end for
6     /* Вставляем элементы в массив – от меньших к большим */
7     for  $i = 1$  to  $n$  do
8          $min = \text{SETMIN}(S)$ 
9          $a[i] = min$ 
10        SETDELETE( $S, min$ )
11    end for
12 end function

```

Первый цикл осуществляет вставку всех элементов массива во множество. На это требуется порядка $O(nT_{Insert})$ операций, где T_{Insert} – вычислительная сложность добавления элемента во множество. Далее заполняем массив, начиная с первого элемента. На каждой итерации i цикла отыскиваем текущий минимальный элемент, вставляем его в i -ю ячейку массива и удаляем элемент из множества. Таким образом, после удаления минимальным станет другой элемент множества, он будет выбран на следующей итерации цикла. Время работы второго цикла есть $O(nT_{Min} + nT_{Delete})$ – мы n раз выполняем поиск минимального элемента и n раз выполняем удаление. Итоговая вычислительная сложность будет равна

$$T_{SetSort}(n) = O(nT_{Insert} + nT_{Min} + nT_{Delete}).$$

Сложность по памяти рассмотренного решения имеет порядок не менее $O(n)$, потому что мы вынуждены хранить копии всех элементов массива в упорядоченном множестве.

Если конкретизировать способ реализации упорядоченного множества, мы получим более определенные оценки вычислительной сложности и сложности по памяти нашего решения. Например, можно реализовать упорядоченное множество на базе красно-черного дерева (red-black tree), которое гарантирует выполнение использованных нами операций за время $O(\log n)$, даже в худшем случае. Тогда вычислительная сложность сортировки на базе красно-черного дерева будет

$$T_{RBTreeSort}(n) = O(n \log n + n \log n + n \log n) = O(n \log n).$$

Сложность по памяти при использовании красно-черного дерева есть $O(n)$ – каждый элемент массива хранится в отдельном узле дерева.

Решение 2. Рассмотрим другой способ SETSORTITERATED решения на базе упорядоченного мульти множества. Реализуем упорядоченный перебор

всех элементов множества при помощи операций MIN и SUCCESSOR . Вставим все элементы массива во множество. Далее отыщем наименьший элемент и поставим его на первое место в массиве. Затем найдем следующий элемент за минимальным (операция SUCCESSOR) и поставим его на второе место в массиве, перейдем к следующему элементу и поставим его на третье место массиве и т. д. Таким образом мы выполним упорядоченный перебор элементов множества. Для чего нам потребуется выполнить одну операцию MIN и $n - 1$ операцию SUCCESSOR .

```

1 function SETSORTITERATED( $a[1..n]$ )
2     /* Вставляем элементы в упорядоченное мульти множество */
3     for  $i = 1$  to  $n$  do
4         SETINSERT( $S, a[i]$ )
5     end for
6      $min = \text{SETMIN}(S)$ 
7      $i = 1$ 
8     while  $i \leq n$  do
9          $a[i] = min$ 
10         $i = i + 1$ 
11         $min = \text{SETSUCCESSOR}(S, min)$ 
12    end while
13 end function
```

Вычислительная сложность алгоритма SETSORTITERATED равна

$$T_{\text{SetSortIterated}}(n) = O(nT_{\text{Insert}} + T_{\text{Min}} + (n - 1)T_{\text{Successor}}).$$

Как правило, в упорядоченных множествах операции SUCCESSOR и PREDECESSOR реализуются эффективно. Например, в красно-черном дереве эти операции выполняются за время $O(\log n)$. В этом случае вычислительная сложность SETSORTITERATED будет

$$T_{\text{RBTreeSetSortIterated}}(n) = O(n \log n + \log n + (n - 1) \log n) = O(n \log n).$$

Рассмотренные решения SETSORT и SETSORTITERATED имеют одинаковые асимптотические оценки вычислительной сложности и сложности по памяти. Однако второе решение SETSORTITERATED на практике может быть предпочтительнее, так как требует выполнения меньшего числа операций.

Решение 3. Рассмотрим третий вариант PRIORITYQUEUESORT , основанный на использовании АТД очередь с приоритетом. Вставим все элементы массива в очередь. В качестве приоритета используется само значение элемента массива. Затем n раз выполним операцию DELETEMIN , которая возвращает и удаляет из очереди элемент с минимальным приоритетом. На каждом шаге i элемент с наименьшим приоритетом помещается в ячейку

массива с номером i .

```

1 function PRIORITYQUEUESORT( $a[1..n]$ )
2   for  $i = 1$  to  $n$  do
3     PRIORITYQUEUEINSERT( $Q, a[i]$ )
4   end for
5   for  $i = 1$  to  $n$  do
6      $min = \text{PRIORITYQUEUEDELETEMIN}(Q)$ 
7      $a[i] = min$ 
8   end for
9 end function

```

Вычислительная сложность алгоритма PRIORITYQUEUESORT равна

$$T_{\text{PriorityQueueSort}} = O(nT_{\text{Insert}} + nT_{\text{DeleteMin}}).$$

При реализации очереди с приоритетом на базе двоичной кучи (binary heap), которая обеспечивает выполнение операций INSERT и DELETEMIN за время $O(\log n)$, вычислительная сложность алгоритма будет

$$T_{\text{BinaryHeapSort}} = O(n \log n + n \log n) = O(n \log n).$$

Недостатком этого решения также является линейная сложность по памяти. Каждый элемент массива хранится в отдельном узле двоичного дерева.

Отметим, что идея последнего решения лежит в основе алгоритма пирамидальной сортировки (heapsort), в которой элементы входного массива переупорядочиваются в виде двоичной кучи (пирамиды, см. раздел 14.3).

Заметим, что рассмотренные нами решения не является наиболее эффективными методами решения задачи сортировки, так как имеют линейную сложность по памяти и большие константы при слагаемых в оценках вычислительной сложности.

5.5. Упражнения

1. Выберите АТД для хранения информации о файлах на жестком диске. Каждый файл характеризуется именем и метаданными (размером, временем последнего изменения и др.). Предполагается, что востребованными будут операции поиска файла по имени, сортировка файлов по размеру и времени последней модификации.

2. У разработчиков программы межсетевого экрана (файервола, брандмауэра) имеется необходимость поддерживать реестр IP-адресов запрещенных сайтов. При поступлении запроса на установление соединения с за-

данным IP-адресом необходимо оперативно определять, к какому типу он относится – запрещенному или разрешенному. Заранее все запрещенные адреса не известны, они могут поступать в систему динамически.

3. Вы разрабатываете компилятор для языка программирования C++. В процессе выполнения лексического и синтаксического анализа из программы выделяются идентификаторы (названия переменных, функций и др.) и помещаются в *таблицу символов* (symbol table). С каждым идентификатором связана информация о его типе, области видимости и пр. Необходимо выбрать АТД для реализации таблицы символов, при условии, что идентификаторы динамически добавляются в таблицу и периодически осуществляется их поиск в ней.

4. Некоторая компания ведет разработку программы для загрузки файлов из сети Интернет. Каждый загружаемый файл («закачка») характеризуется адресом в сети (URL) и приоритетом доступа к каналу связи по отношению к другим файлам. Пользователь вправе динамически добавлять новые файлы и изменять их приоритеты. Программа периодически обращается к коллекции файлов и с учетом их приоритетов загружает их части из сети. Необходимо выбрать АТД для поддержания информации о загружаемых файлах.

5. Компания Интернет-гигант ведет разработку своего поискового робота (crawler), который перебирает страницы в сети Интернет и сохраняет информацию о них в базу данных. Функционирование робота начинается с заданного начального адреса. На каждой странице робот отыскивает ссылки на другие ресурсы и запоминает их для дальнейшей обработки. Выберите АТД для хранения коллекции ссылок на интернет-страницы.

6. Списки

6.1. АТД список

Список (list) – это абстрактный тип данных, представляющий собой набор однотипных элементов (item, entry, node), упорядоченных в соответствии с их *позицией* в списке – *индексом* (index). Каждый элемент списка помимо позиции также хранит некоторое *значение* (value). Нумерация элементов начинается с единицы или нуля. Последнее оговаривается при реализации АТД в виде конкретной структуры данных. Между элементами списка задано бинарное отношение «следует за» или «предшествует». Элементы в списке могут иметь одинаковые значения. На рис. 6.1 показан пример списка из шести элементов, каждый из которых характеризуется положительным номером (индексом) и целым значением.

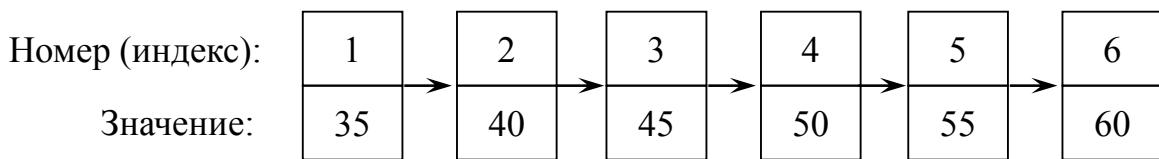


Рис. 6.1. Список из шести элементов
(стрелками показано отношение «следует за»).

Обозначим через L список элементов со значениями типа T , i – позиция элемента в списке, x – значение элемента.

Вставка. Функция $\text{INSERT}(L, i, x)$ добавляет элемент со значением x в позицию i списка L . При этом все элементы от позиции i и далее перемещаются в следующую позицию в направлении конца списка. Например, если список состоял из элементов a_1, a_2, \dots, a_n , то после выполнения вставки список примет следующий вид:

$$a_1, a_2, \dots, a_{i-1}, x, a_i, a_{i+1}, \dots, a_n.$$

Если $i = \text{SIZE}(L) + 1$, то вставка элемента осуществляется в конец списка:

$$a_1, a_2, \dots, a_n, x.$$

Если позиции i нет в списке, то функция должна вернуть специальное значение.

Поиск по значению. Функция $\text{LOOKUP}(L, x)$ возвращает позицию элемента со значением x . Если в списке несколько таких элементов, то возвращается номер позиции первого из них. Если элемента нет в списке, то функция должна вернуть специальное значение, сигнализирующее об этом. Например, для списка на рис. 6.1 операция $\text{LOOKUP}(L, 55)$ вернет значение 5.

Поиск по индексу. Операция $\text{GETITEM}(L, i)$ возвращает значение элемента, который находится в позиции i . Если в списке нет позиции i , то функция должна вернуть специальное значение. Для списка на рис. 6.1 результатом выполнения операции $\text{GETITEM}(L, 3)$ будет значение 45.

Удаление. Функция $\text{DELETE}(L, i)$ удаляет из списка элемент, который находится в позиции i . Все элементы от позиции $i+1$ и далее перемещаются в предыдущую позицию. Если список состоял из элементов a_1, a_2, \dots, a_n , то после удаления элемента с номером i , список примет вид

$$a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n.$$

Получение следующего узла. Операция $\text{NEXT}(L, i)$ возвращает позицию, следующую за i . Если $i = \text{SIZE}(L)$, то функция должна вернуть значение $\text{SIZE}(L) + 1$.

Получение предшествующего узла. Операция $\text{PREV}(L, i)$ возвращает позицию, предшествующую i . Функция должна вернуть специальное значение, если позиции i нет в списке или $i = 1$.

Длина списка. Функция $\text{SIZE}(L)$ возвращает количество элементов в списке.

При необходимости интерфейс списка может быть расширен такими операциями, как добавление элемента в начало или конец списка, удаление всех элементов и др.

6.2. Реализация списков на базе массивов

Одной из возможных реализаций АТД список является использование одномерного массива фиксированного размера t . Номер элемента списка используется как индекс в массиве. Такая реализация позволяет за константное время отыскивать элементы по их позиции (функция GETITEM). Однако операция LOOKUP поиска элемента по значению реализуется за линейное время, так как в худшем случае требуется просмотреть все ячейки массива для поиска элемента с заданным значением. Вставка нового элемента в середину списка также имеет линейную вычислительную сложность от размера массива и требует перемещения всех последующих элементов на одну позицию к концу массива. Кроме того, необходимо контролировать, чтобы количество элементов в списке не превысило размер

массива. С удалением элемента из середины списка возникает аналогичная ситуация – при удалении необходимо перемещать элементы на одну позицию к началу массива.

Сложность по памяти реализации списка на базе массива равна $\Theta(m)$. Независимо от числа n элементов в списке мы всегда занимаем память под m ячеек массива. Это один из недостатков реализации списков на базе массивов.

Ниже приведен псевдокод функций, реализующих операции АТД список на базе массива. Через n обозначено количество элементов в списке, m – максимальный размер массива (рис. 6.2). Условимся, что нумерация элементов массива начинается с 1.

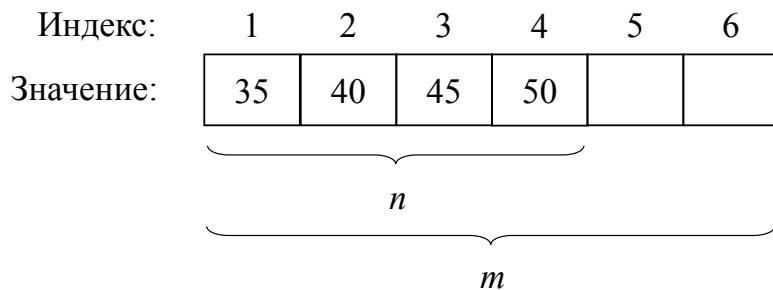


Рис. 6.2. Реализация списка на базе массива.

Поиск по значению. Функция LOOKUP реализует линейный поиск позиции заданного значения x в массиве. Если элемент не найден, функция возвращает специальное значение *NotFound* (элемент не найден).

```

1 function LOOKUP( $L[1..m]$ ,  $x$ )
2     for  $i = 1$  to  $n$  do
3         if  $L[i] = x$  then
4             return  $i$ 
5         end if
6     end for
7     return NotFound
8 end function

```

Худший случай для функции LOOKUP – это отсутствие искомого элемента в массиве или его нахождение в последней ячейке. В этой ситуации алгоритму требуется пройти по всем ячейкам массива, что требует порядка $\Theta(n)$ операций.

Проведем анализ эффективности алгоритма LOOKUP для среднего случая, когда искомый элемент может быть найден в любой из позиций списка с одинаковой вероятностью $1/n$. Обозначим вероятность успешного поиска элемента в массиве через $p \in [0, 1]$. Тогда вероятность отсутствия значения x в массиве равна $1 - p$.

Рассмотрим возможные варианты выполнения алгоритма при условии, что искомый элемент присутствует в массиве. Если искомый элемент x находится в ячейке 1, то для его поиска требуется выполнить 3 операции: проверить условие окончания цикла, условие в теле цикла и вернуть значение 1. Если элемент находится в ячейке 2, то это требует выполнения 5 операций и т. д. В общем случае, если элемент x расположен в ячейке i , то потребуется выполнить $2i + 1$ операции. Таким образом, число операций, выполняемых алгоритмом, есть дискретная случайная величина, которая может принимать значения $3, 5, \dots, 2i + 1, \dots, 2n + 1$ с одинаковой вероятностью, равной p/n .

Если искомый элемент отсутствует в массиве, то число операций равно $2n + 1$. Это обусловлено тем, что мы вынуждены просмотреть все элементы массива и вернуть значение *NotFound*.

Запишем математическое ожидание числа операций, выполняемых алгоритмом. По определению, математическое ожидание есть сумма произведений значения дискретной случайной величины на вероятность принятия случайной величиной этого значения:

$$T_{Average}(n) = 3\frac{p}{n} + 5\frac{p}{n} + \dots + (2i + 1)\frac{p}{n} + \dots + (2n + 1)\frac{p}{n}.$$

В нашей оценке мы должны учесть и тот факт, что искомое значение x с вероятностью $1 - p$ может отсутствовать в массиве. Тогда формула примет следующий вид:

$$T_{Average}(n) = \frac{p}{n}[3 + 5 + \dots + (2n + 1)] + (1 - p)(2n + 1).$$

В квадратных скобках записана сумма членов арифметической прогрессии. Вычислим ее

$$T_{Average}(n) = \frac{p}{n}[n^2 + 2n] + (1 - p)(2n + 1) = p(n + 2) + (1 - p)(2n + 1).$$

Таким образом, вычислительная сложность алгоритма `LOOKUP` в среднем случае равна $\Theta(n)$.

Поиск по индексу. Функция `GETITEM` осуществляет поиск элемента по его позиции i . Для выполнения этой операции требуется константное время, так как позиция i элемента используется как индекс в одномерном массиве.

Вставка. Вставка элемента в список реализуется функцией `INSERT`. Все элементы, начиная с позиции i , перемещаются на одну ячейку к концу списка. Это реализуется циклом, который перемещает значение из ячейки n в ячейку $n + 1$, значение из ячейки $n - 1$ в ячейку n и т. д., пока не переместит значение из i в $i + 1$. После чего значение x записывается в ячейку i .

```

1 function GETITEM( $L[1..m]$ ,  $i$ )
2     if  $i < 1$  or  $i > n$  then
3         return InvalidIndex
4     end if
5     return  $L[i]$ 
6 end function

```

Худший случай для INSERT – это вставка элемента в начало списка. Это требует перемещения всех n элементов на одну позицию к концу списка. Следовательно, в худшем случае вычислительная сложность вставки элемента в список равна $\Theta(n)$.

```

1 function INSERT( $L[1..m]$ ,  $i, x$ )
2     if  $n \geq m$  then
3         return ListOverflow          /* Список заполнен */
4     end if
5     if  $i < 1$  or  $i > n + 1$  then
6         return InvalidIndex
7     end if
8      $j = n$ 
9     while  $j \geq i$  do
10         $L[j + 1] = L[j]$            /* Перенос на одну позицию к концу */
11         $j = j - 1$ 
12    end while
13     $L[i] = x$ 
14     $n = n + 1$ 
15 end function

```

Удаление. Удаление элемента из заданной позиции реализуется функцией DELETE. В этой функции осуществляется перенос элементов на одну позицию к началу списка. Здесь возникает ситуация, аналогичная той, что мы рассмотрели при анализе функции INSERT, – в худшем случае осуществляется попытка удаления первого элемента. Это приводит к n перемещениям элементов. Вычислительная сложность удаления в худшем случае равна $\Theta(n)$.

Поиск соседних элементов. Поиск следующей и предыдущий позиций для заданного номера i элемента осуществляется функциями NEXT и PREV. Вычислительная сложность этих функций является константной, так как они реализуются двумя арифметическими операциями — увеличением и уменьшением на единицу заданной позиции i .

Положительными сторонами реализации списка на основе массива является константная сложность поиска элемента по его индексу, а также непрерывное размещение элементов в памяти компьютера, что повышает

```

1 function DELETE( $L[1..m]$ ,  $i$ )
2     if  $i < 1$  or  $i > n$  then
3         return InvalidIndex
4     end if
5     if  $i < n$  then
6         for  $j = i$  to  $n$  do
7              $L[j] = L[j + 1]$       /* Перенос на одну позицию к началу */
8         end for
9     end if
10     $n = n - 1$ 
11 end function

```

вероятность попадания смежных элементов массива в кеш-память процессора. Кроме этого, непрерывное размещение массива в памяти делает возможным использование векторных инструкций процессора (Intel SSE/AVX, IBM AltiVec, ARM NEON SIMD) для оптимизации перемещения подмассивов в операциях INSERT и DELETE.

К недостаткам можно отнести линейную вычислительную сложность вставки и удаления элементов, а также фиксированность размера массива и возможный нерациональный расход памяти, обусловленный хранением пустых ячеек массива (как правило, $n < m$).

Заметим, что в случае переполнения массива можно создать новый массив большего размера и перенести в него элементы списка. Размер нового массива можно выбирать разными способами, в частности, используя *мультипликативную схему* – увеличивать размер в несколько раз, например, в два. Такие расширяющиеся массивы называют *динамическими* (dynamic array, growable array) [1].

```

1 function NEXT( $i$ )
2     if  $i < 1$  or  $i > n$  then
3         return InvalidIndex
4     end if
5     return  $i + 1$ 
6 end function

7 function PREV( $i$ )
8     if  $i < 2$  or  $i > n$  then
9         return InvalidIndex
10    end if
11    return  $i - 1$ 
12 end function

```

6.3. Связные списки

Связный список (linked list) – это динамическая структура данных, представляющая собой совокупность логически связанных узлов. Каждый узел (node) содержит информационные поля с данными и указатели на *следующий узел* (next) и при необходимости на *предыдущий* (prev, от англ. previous – предыдущий).

Если каждый узел списка содержит указатель только на следующий элемент (next), то список называется *односвязным*, или *однонаправленным* (singly linked list). На рис. 6.3 приведен пример односвязного списка из четырех элементов.

Указатель на первый элемент списка принято называть *головой* (head). Элемент списка, указатель next которого равен специальному значению null, называется *хвостовым* узлом (tail), или *последним* (last). Константа null обозначает нулевой указатель. Если указатель на первый элемент списка равен null, то список считается пустым.

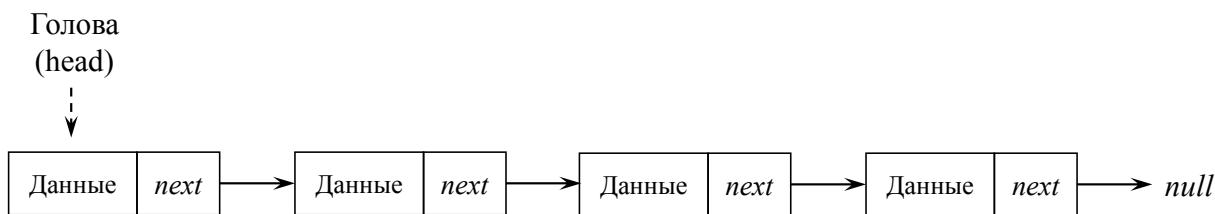


Рис. 6.3. Односвязный список.

Связные списки относят к динамическим структурам данных, так как узлы в них можно добавлять по мере необходимости. Для каждого элемента списка выделяется отдельный участок памяти. Порядок следования элементов связного списка может не совпадать с порядком размещения узлов в памяти компьютера. Это принципиально отличает связные списки от массивов. Например, на рис. 6.4 показан односвязный список, в котором логический порядок следования узлов отличается от порядка их размещения в памяти – адреса узлов могут быть произвольными. Чтобы добраться до определенного элемента в списке, необходимо пройти, начиная с головы, по указателям next всех узлов. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

Если каждый узел списка содержит указатель на следующий узел и на предыдущий, то список называется *двусвязным*, или *двунаправленным* (doubly linked list). В таких списках можно передвигаться как к началу списка, так и к его концу (рис. 6.5). Это упрощает процедуру удаления, поскольку известны адреса предшествующего и следующего элементов.

Связные списки могут быть *кольцевыми* (circular), или, как их еще называют, *циклическими, замкнутыми*. В кольцевом связном списке поле next последнего элемента указывает на первый узел списка, а поле prev первого узла содержит адрес последнего элемента.

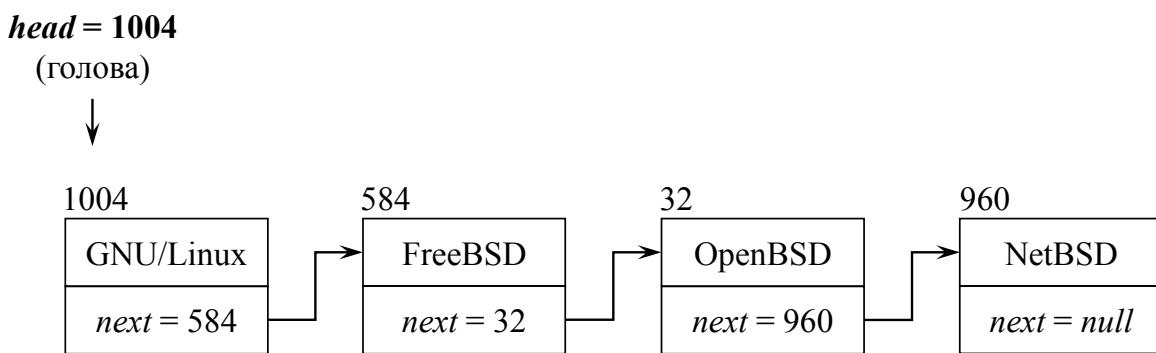


Рис. 6.4. Односвязный список из четырех элементов (над узлами показаны их адреса в памяти, а также значения поля *next*).



Рис. 6.5. Двусвязный список.

6.4. Односвязные списки

Каждый узел односвязного списка имеет два поля *value* и *next*. Поле *value* – это некоторые данные, ассоциированные с узлом. В поле *next* хранится адрес узла, следующего за текущим. На рис. 6.6 показан пример односвязного списка с адресами узлов в памяти.

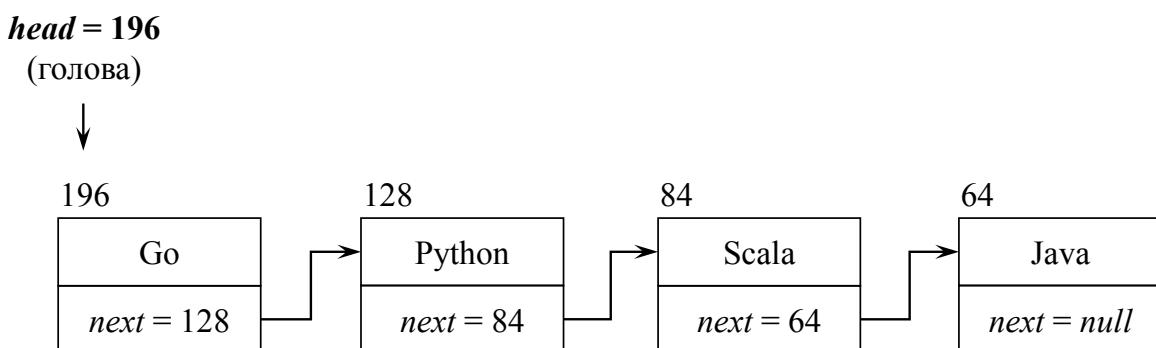


Рис. 6.6. Пример односвязного списка (над узлами показаны их адреса в памяти, а также значения поля *next*).

Создание нового узла. Создание нового элемента выполняется функцией `LINKEDLISTCREATENODE`. Она выделяет память под узел и заполняет его поля значениями по умолчанию. Фактически эта функция создает список из одного узла. Вычислительная сложность этой функции определяется сложностью операции `ALLOCATEMEMORY` выделения памяти под узел. Далее будем считать, что `ALLOCATEMEMORY` имеет константную вычислительную сложность.

```

1 function LINKEDLISTCREATENODE(value)
2     node = ALLOCATEMEMORY()          /* Выделяем память под узел */
3     if node = null then
4         return null                  /* Ошибка выделения памяти */
5     end if
6     node.value = value
7     node.next = null
8     return node
9 end function

```

Добавление в начало. Добавление узла в начало списка реализуется операцией LINKEDLISTADDFRONT. Функция создает в памяти новый узел с заданным значением поля *value*. Затем в поле *next* нового узла заносится адрес головы списка. Таким образом новый узел становится первым в списке. Из функции возвращается адрес нового узла. Далее этот адрес должен использоваться как адрес головы списка. Функция LINKEDLISTADDFRONT характеризуется константной вычислительной сложностью. На рис. 6.7 показан пример добавления узла «JavaScript» в начало списка с рис. 6.6.

```

1 function LINKEDLISTADDFRONT(list, value)
2     node = LINKEDLISTCREATENODE(value)
3     node.next = list              /* Делаем головой списка новый узел */
4     return node
5 end function

```

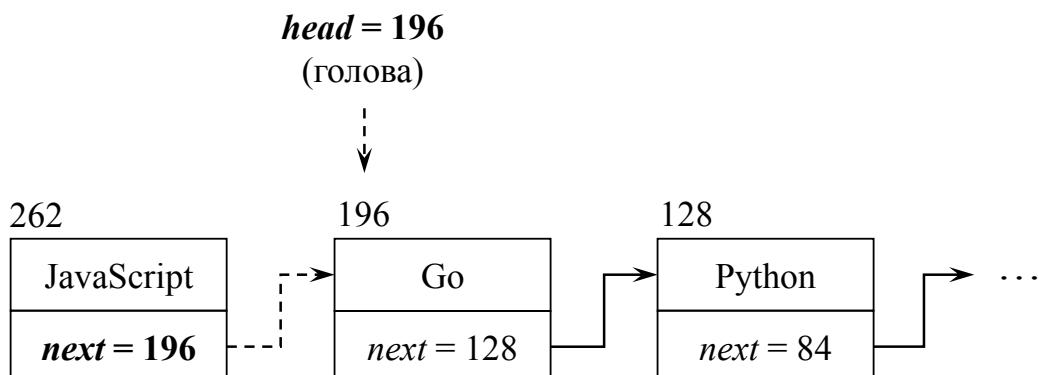


Рис. 6.7. Добавление узла «JavaScript» в начало односвязного списка (штриховыми линиями показаны указатели, требующие корректировки).

Ниже приведен пример использования функции LINKEDLISTADDFRONT для построения списка на рис. 6.6.

При первом вызове функции LINKEDLISTADDFRONT ей передается адрес *head* головы списка, равный *null*. Функция создает узел «Java» с начальным адресом 64 (см. рис. 6.6) и возвращает этот адрес в качестве адреса

```

1 head = null
2 head = LINKEDLISTADDFRONT(head, “Java”)
3 head = LINKEDLISTADDFRONT(head, “Scala”)
4 head = LINKEDLISTADDFRONT(head, “Python”)
5 head = LINKEDLISTADDFRONT(head, “Go”)

```

головы списка. Таким образом переменная *head* принимает значение 64. При втором обращении к функции LINKEDLISTADDFRONT ей передается адрес *head*, равный 64. Она создает узел «Scala» с адресом 84. В поле *next* этого узла записывается адрес головы списка, равный 64. Функция возвращает адрес 84, который записывается в переменную *head*, и т. д. После выполнения четвертого вызова функции LINKEDLISTADDFRONT переменная *head* будет содержать адрес 196.

Добавление в конец. Для добавления нового узла в конец связного списка необходимо пройти все элементы, начиная с головы, и найти последний узел – узел, поле *next* которого равно *null*. Далее необходимо связать последний узел с новым узлом. В поле *next* последнего узла записывается адрес нового элемента. Ниже приведен псевдокод функции LINKEDLISTADDEND, которая реализует описанную операцию. Иллюстрация работы этой функции приведена на рис. 6.8.

```

1 function LINKEDLISTADDEND(list, value)
2     newnode = LINKEDLISTCREATENODE(value)
3     if list = null then
4         return newnode
5     end if
6     node = list                                /* Находим последний узел */
7     while node.next ≠ null do
8         node = node.next                      /* Переходим к следующему узлу */
9     end while
10    node.next = newnode                     /* Связываем с новым узлом */
11    return list
12 end function

```

Вычислительная сложность функции LINKEDLISTADDEND в худшем и среднем случаях равна $\Theta(n)$, где n – количество узлов в списке. Это действительно так, поскольку в любом случае нам необходимо пройти по всем элементам списка для нахождения его последнего узла.

Поиск узла. Поиск узла по заданному значению поля *value* реализуется функцией LINKEDLISTLOOKUP. В цикле проверяем все узлы, начиная с головы списка. Если узел найден, возвращаем из функции указатель на него. Если узла с заданным значением поля *value* нет в списке, то функция возвращает *null*.

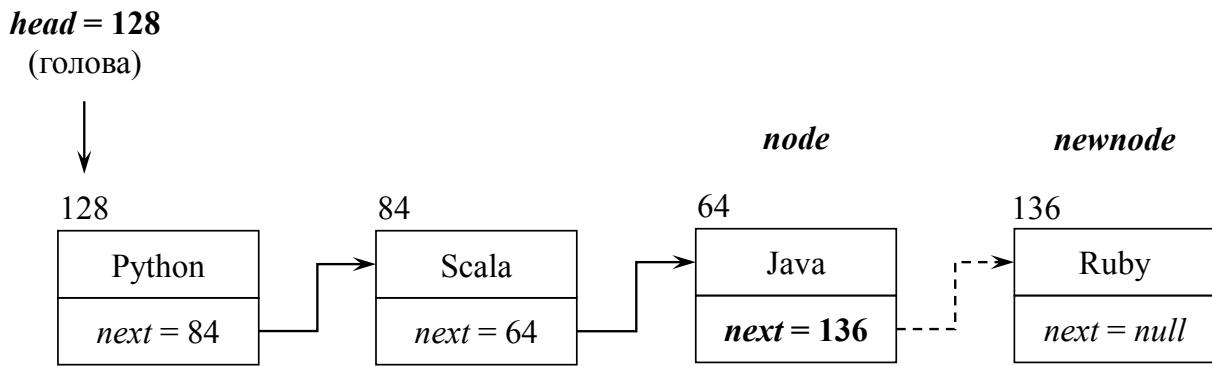


Рис. 6.8. Добавление узла со значением «Ruby» в конец списка (штриховыми линиями показан указатель, требующий корректировки).

```

1 function LINKEDLISTLOOKUP(list, value)
2   while list ≠ null do
3     if list.value = value then
4       return list
5     end if
6     list = list.next           /* Переходим к следующему узлу */
7   end while
8   return null                /* Узел не найден */
9 end function

```

В худшем случае искомое значение может отсутствовать в списке или находится в последнем узле. Следовательно, вычислительная сложность функции LINKEDLISTLOOKUP в худшем случае равна $\Theta(n)$.

Удаление узла. Удаление из связного списка элемента с заданным значением поля *value* реализуется функцией LINKEDLISTDELETE.

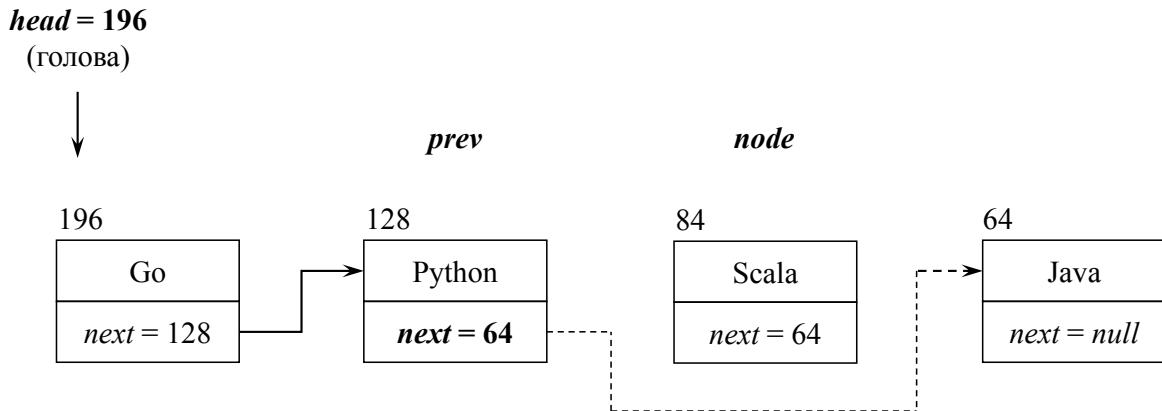


Рис. 6.9. Пример удаление узла со значением «Scala» из связного списка (штриховыми линиями указатель, требующий корректировки).

Первым делом функция отыскивает удаляемый элемент. В процессе своей работы она поддерживает указатель *prev* на предыдущий узел по отношению к текущему элементу *node*. После нахождения узла возмож-

ны два случая. Случай 1: удаляемый элемент является первым элементом списка, у него нет предшествующего узла ($prev = null$). В этом случае головой списка становится элемент, следующий за первым. Случай 2: удаляемый узел имеет предшествующий элемент (рис. 6.9). Здесь необходимо перенаправить указатель $next$ предшествующего узла на узел, следующий за удаляемым ($prev.next = node.next$). После корректировки указателей происходят освобождение памяти из-под удаляемого узла и возврат указателя на голову модифицированного списка.

```

1 function LINKEDLISTDELETE(list, value)
2     node = list
3     prev = null
4     while node ≠ null do          /* Отыскиваем удаляемый узел */
5         if node.value = value then
6             if prev = null then
7                 list = node.next    /* Удаляемый узел – голова списка */
8             else
9                 prev.next = node.next
10            end if
11            FREEMEMORY(node)           /* Освобождаем память */
12            return list
13        end if
14        prev = node
15        node = node.next
16    end while
17    return null                  /* Узел не найден */
18 end function
```

Ниже приведен пример использования рассмотренных нами функций. После добавления узлов в связный список они будут расположены в следующем порядке: «PHP» → «Basic» → «JavaScript». После удаления узла «Basic» список примет вид: «PHP» → «JavaScript».

```

1 head = null
2 head = LINKEDLISTADDFRONT(head, “Basic”)
3 head = LINKEDLISTADDFRONT(head, “PHP”)
4 head = LINKEDLISTADDEND(head, “JavaScript”)
5 head = LINKEDLISTDELETE(head, “Basic”)
6 node = LINKEDLISTLOOKUP(head, “JavaScript”)
7 if node ≠ null then
8     print(“JavaScript”)
9 end if
```

6.5. Двусвязные списки

Каждый узел двусвязного списка имеет три поля: *value*, *next* и *prev*. Поле *value* – это некоторые данные, ассоциированные с узлом. В поле *next* хранится адрес узла, следующего за текущим, а в поле *prev* – адрес предшествующего узла. На рис. 6.10 показан пример двусвязного списка с их адресами в памяти.

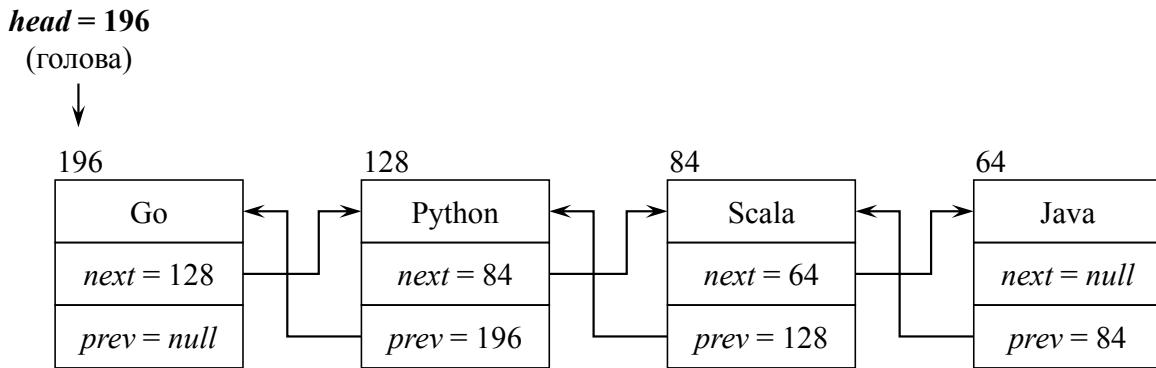


Рис. 6.10. Пример двусвязного списка.

Операции над двусвязными списками подобны операциям над односвязными. Рассмотрим некоторые из них.

Создание нового узла. Создание нового элемента выполняется функцией DOUBLYLINKEDLISTCREATE NODE. Она выделяет память под узел и заполняет его поля значениями по умолчанию.

```

1 function DOUBLYLINKEDLISTCREATE NODE(value)
2   node = ALLOCATEMEMORY()      /* Выделяем память под узел */
3   node.value = value
4   node.next = null
5   node.prev = null
6   return node
7 end function
  
```

Добавление в начало. Добавление узла в начало списка реализуется операцией DOUBLYLINKEDLISTADDFRONT. Функция создает в памяти новый узел с заданным значением поля *value*. Затем в поле *next* нового узла записывается адрес головы списка. Если список не пуст, то необходимо записать в указатель *prev* первого узла адрес нового элемента. Вычислительная сложность этой функции $\Theta(1)$. На рис. 6.11 показан пример добавления узла «JavaScript» в начало списка с рис. 6.10.

Добавление в конец. Для добавления нового узла в конец двусвязного списка необходимо пройти все элементы, начиная с головы, и найти последний узел – узел, поле *next* которого равно *null*. Далее необходимо связать последний узел с новым узлом: скорректировать их указатели *next*

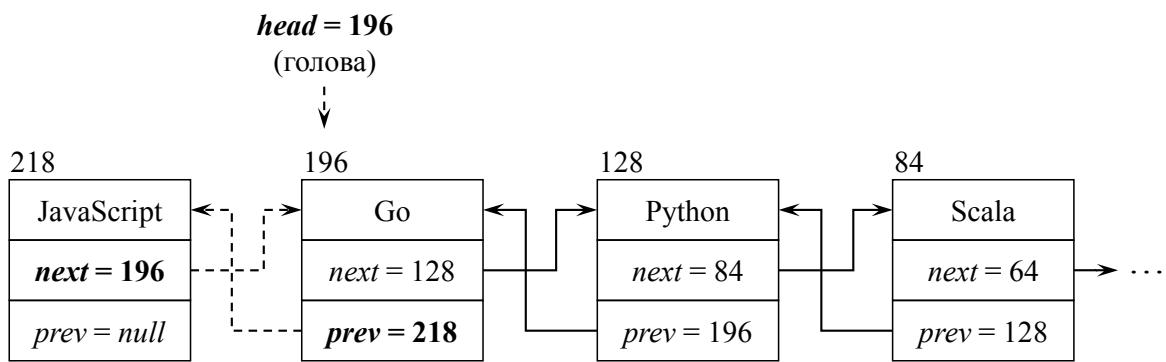


Рис. 6.11. Добавление узла «JavaScript» в начало двусвязного списка (штриховыми линиями показаны указатели, требующие корректировки).

```

1 function DOUBLYLINKEDLISTADDFRONT(list, value)
2   node = LINKEDLISTCREATENODE(value)
3   node.next = list           /* Делаем головой списка новый узел */
4   if list ≠ null then
5     list.prev = node
6   end if
7   return node
8 end function
```

и *prev*. Ниже приведен псевдокод функции DOUBLYLINKEDLISTADDEND, которая реализует описанную операцию. Иллюстрация работы этой функции приведена рис. 6.12.

```

1 function DOUBLYLINKEDLISTADDEND(list, value)
2   newnode = LINKEDLISTCREATENODE(value)
3   if list = null then
4     return newnode
5   end if
6   node = list           /* Поиск последнего узла */
7   while node.next ≠ null do
8     node = node.next
9   end while
10  node.next = newnode      /* Связываем с новым узлом */
11  newnode.prev = node
12  return list
13 end function
```

В худшем и среднем случаях для вставки элемента в конец списка необходимо пройти по всем его элементам. Поэтому вычислительная сложность функции DOUBLYLINKEDLISTADDEND в худшем и среднем случаях равна $\Theta(n)$, где n – это количество узлов в списке.

Поиск узла. Поиск узла в двусвязном списке реализуется точно так

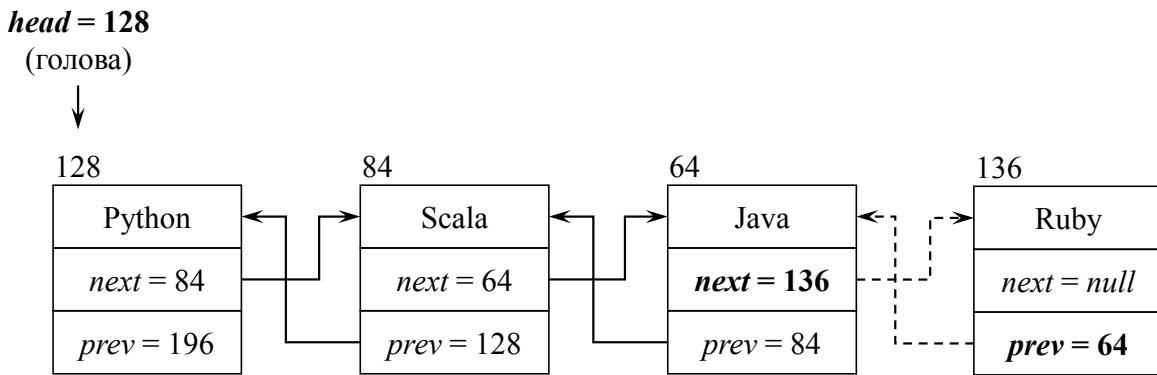


Рис. 6.12. Добавление узла со значением «Ruby» в конец списка (штриховыми линиями показаны указатели, требующие корректировки).

же, как и в односвязном.

Удаление узла. Удаление из двусвязного списка элемента с заданным значением реализуется функцией DOUBLYLINKEDLISTDELETE. Алгоритм работы функции аналогичен тому, что мы рассмотрели применительно к односвязным спискам. Функция отыскивает удаляемый элемент и корректирует указатели *prev* и *next* узлов (рис. 6.13). Вычислительная сложность удаления элемента в худшем случае равна $\Theta(n)$.

```

1 function DOUBLYLINKEDLISTDELETE(list, value)
2     node = list
3     while node ≠ null do          /* Отыскиваем удаляемый узел */
4         if node.value = value then
5             if node.prev = null then
6                 list = node.next /* Удаляемый узел – голова списка */
7             else
8                 node.prev.next = node.next
9             end if
10            if node.next ≠ null then
11                node.next.prev = node.prev
12            end if
13            FREEMEMORY(node)           /* Освобождаем память */
14            return list
15        end if
16        node = node.next
17    end while
18    return null                    /* Узел не найден */
19 end function

```

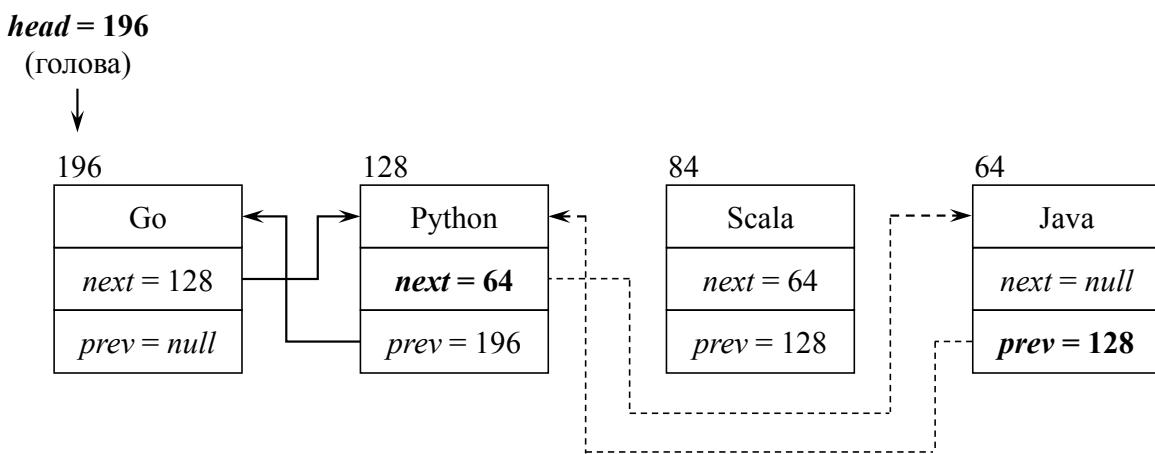


Рис. 6.13. Пример удаление узла со значением «Scala» из связного списка (штриховыми линиями показаны указатели, требующие корректировки).

6.6. Реализация АТД список

Рассмотрим реализацию АТД список на базе односвязного списка. В функциях, приведенных ниже, через *list* обозначен указатель на голову односвязного списка, *x* – значение элемента, *n* – количество элементов в списке.

Поиск по значению. Функция LOOKUP реализует поиск позиции заданного значения *x* в списке. Если элемент не найден, функция возвращает специальное значение *NotFound* (элемент не найден). Вычислительная сложность операции равна $\Theta(n)$.

```

1 function LOOKUP(list, x)
2     index = 1
3     while list ≠ null do
4         if list.value = x then
5             return index
6         end if
7         list = list.next          /* Переходим к следующему узлу */
8         index = index + 1
9     end while
10    return NotFound
11 end function
```

Поиск по индексу. Функция GETITEM осуществляет поиск элемента по его позиции *i*. Для этого в худшем случае требуется просмотреть все элементы связного списка. Следовательно, вычислительная сложность функции равна $\Theta(n)$.

Вставка. Вставка элемента в список реализуется функцией INSERT. Эта функция отыскивает узел с номером *i*, создает новый узел и вставляет его перед найденным элементом. Для реализации возможности добавления

```

1 function GETITEM(list, i)
2     index = 1
3     while list ≠ null do
4         if index = i then
5             return list.value
6         end if
7         list = list.next           /* Переходим к следующему узлу */
8         index = index + 1
9     end while
10    return NotFound
11 end function

```

узла в конец списка ($i = \text{SIZE}(\text{list}) + 1$) после цикла стоят соответствующая проверка и обращение к функции `LINKEDLISTADDEND`. Худший случай для этой операции – это вставка узла в конец списка. Таким образом, вычислительная сложность вставки элемента в список равна $\Theta(n)$.

```

1 function INSERT(list, i, x)
2     node = list
3     prev = null
4     index = 1
5     while node ≠ null do          /* Отыскиваем узел с индексом i */
6         if index = i then
7             newnode = LINKEDLISTCREATE NODE(x)
8             if prev = null then
9                 newnode.next = list
10                list = newnode           /* Вставка в начало списка */
11            else
12                newnode.next = node
13                prev.next = newnode
14            end if
15            return list
16        end if
17        prev = node
18        node = node.next
19        index = index + 1
20    end while
21    if index = i then           /* Вставка в конец */
22        return LINKEDLISTADDEND(list, x)
23    end if
24    return InvalidIndex
25 end function

```

Удаление. Удаление элемента из заданной позиции реализуется функцией `DELETE`. В этой функции осуществляется поиск и удаления узла с номером i . Вычислительная сложность функции $\Theta(n)$.

```

1 function DELETE(list, i)
2     node = list
3     prev = null
4     index = 1
5     while node ≠ null do
6         if index = i then
7             if prev = null then
8                 list = node.next
9             else
10                prev.next = node.next
11            end if
12            FREEMEMORY(node)          /* Освобождаем память */
13            return list
14        end if
15        prev = node
16        node = node.next
17        index = index + 1
18    end while
19    return InvalidIndex
20 end function
```

Таблица 6.1. Вычислительная сложность операций АТД список
(худший случай)

Операция	Реализация списка	
	Массив	Односвязный список
<code>INSERT(<i>L</i>, <i>i</i>, <i>x</i>)</code>	$\Theta(n)$	$\Theta(n)$
<code>LOOKUP(<i>L</i>, <i>x</i>)</code>	$\Theta(n)$	$\Theta(n)$
<code>GETITEM(<i>L</i>, <i>i</i>)</code>	$O(1)$	$\Theta(n)$
<code>DELETE(<i>L</i>, <i>i</i>)</code>	$\Theta(n)$	$\Theta(n)$
<code>NEXT(<i>L</i>, <i>i</i>)</code>	$O(1)$	$\Theta(n)$
<code>PREV(<i>L</i>, <i>i</i>)</code>	$O(1)$	$\Theta(n)$
<code>SIZE(<i>L</i>)</code>	$O(1)$	$\Theta(n)$

Положительными сторонами рассмотренной реализации является отсутствие жестких ограничений на размер списка. К недостаткам можно

отнести линейную вычислительную сложность поиска элемента по его номеру (индексу).

В табл. 6.1 приведены вычислительные сложности операций АТД списка для худшего случая.

6.7. Упражнения

1. Разработайте алгоритмы удаления всех элементов из односвязного и двусвязного списков.
2. Создайте функцию, реализующую подсчет числа элементов в односвязном списке (операция $\text{SIZE}(L)$). Подумайте, как реализовать операцию вычисления длины связного списка за константное время.
3. Разработайте функцию изменения порядка следования узлов в связном списке на обратный – последний узел становится первым, предпоследний вторым и т. д. Продумайте реализацию этой функции для двусвязных списков.
4. Реализуйте процедуру поиска элемента в кольцевом односвязном списке. Продумайте способы определения того, что при проходе по кольцевому списку достигнут его конец (голова).
5. Разработайте функцию сортировки элементов односвязного списка.
6. Предложите реализацию операций неупорядоченного множества на базе односвязного списка.
7. Разработайте алгоритм объединения двух односвязных списков.
8. Предложите реализацию операций очереди с приоритетом на базе односвязного списка.

7. Стеки

7.1. АТД стек

Стек (stack) – это абстрактный тип данных, представляющий собой список, в котором *вставка* (PUSH) и удаление элементов (POP) выполняются с одного конца – *вершины стека* (top). Стек функционирует по принципу «последним пришел – первым вышел» (Last In – First Out, LIFO). В качестве примера можно привести стопку тарелок – очередная тарелка всегда кладется сверху, взять тарелку из стопки можно также только сверху. На рис. 7.1 приведен пример стека, содержащего четыре элемента, помещенные в него в порядке A, B, C, D . В отечественной литературе можно встретить альтернативное название стека – *магазин*.

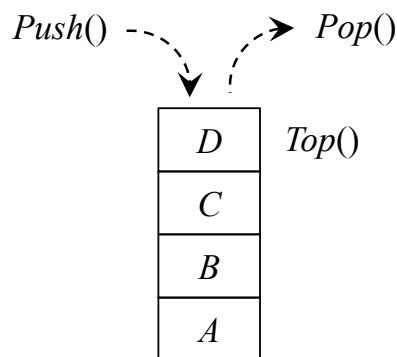


Рис. 7.1. Стек с четырьмя элементами, добавленными в порядке A, B, C, D .

Обозначим через S стек со значениями типа T , x – значение элемента.

Помещение элемента в стек. Операция $\text{PUSH}(S, x)$ помещает элемент со значением x в вершину стека S (от англ. push – проталкивать). Элемент, который находился в вершине стека, становится элементом, следующим за вершиной. Если рассматривать стек как список, то помещение элемента в стек эквивалентно добавлению элемента в начало списка.

Удаление элемента из стека. Функция $\text{POP}(S)$ удаляет элемент из вершины стека (от англ. pop – выталкивать). Если стек пуст, то функция должна вернуть специальное значение. Удаление элемента из вершины стека эквивалентно удалению первого элемента списка.

Доступ к вершине стека. Операция $\text{TOP}(S)$ возвращает значение элемента из вершины стека. Если стек пуст, то функция должна вернуть специальное значение.

Проверка стека на пустоту. Функция $\text{ЕМРТУ}(S)$ возвращает значение *True* если стек пуст и значение *False* в противном случае.

На рис. 7.2 показаны состояния стека в результате выполнения последовательности операций PUSH и POP.

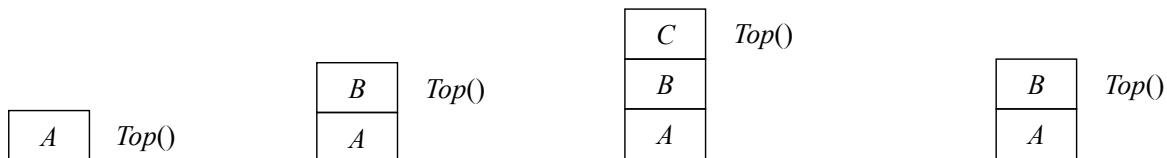


Рис. 7.2. Последовательность операций над стеком:

- a* – помещение в стек элемента *A*;
- b* – помещение в стек элемента *B*;
- c* – извлечение из стека элемента *C*.

Стеки активно используются в алгоритмах, которые в процессе своей работы осуществляют возврат к ранее частично обработанным данным. Например, в алгоритмах обхода деревьев и графов стек используется для хранения посещенных узлов и последующего возврата к ним.

Заметим, что процессоры поддерживают аппаратный стек, который хранится в оперативной памяти, а в качестве указателя на вершину стека используется специальный регистр. Аппаратный стек используется для организации вызова подпрограмм и возврата из них. Кроме этого, стек предназначен для реализации локальной области видимости переменных и передачи параметров в функции.

Понятие стека было предложено в 1946 г. Аланом Тьюрингом (Alan Turing, 1912–1954) при разработке автоматической вычислительной машины ACE (Automatic Computing Engine). Он описал использование LIFO-очереди для реализации вызова подпрограмм и возврата из них. Операции PUSH и POP имели названия BURY (закопать) и UNBURY (откопать) соответственно.

7.2. Реализация стека на базе массива

Рассмотрим реализацию стека на базе одномерного массива $S[1..n]$ фиксированной длины n . Переменная *top* будет содержать номер ячейки массива, являющейся вершиной стека. Таким образом, элемент $S[\text{top}]$ хранит последнее добавленное в стек значение. Начальное значение переменной *top* равно 0 (стек пуст). При хранении стека в виде массива требуется порядка $\Theta(n)$ ячеек памяти.

Помещение элемента в стек. Помещение элемента в стек реализуется операцией PUSH. Для этого значение переменной *top* увеличивается на 1 и в соответствующую ячейку массива записывается заданное значение. Перед записью значения в стек выполняется проверка на переполнение

массива. Если количество добавленных в стек элементов совпадает с размером n массива, пользователю сообщается о переполнении стека (stack overflow). Вычислительная сложность этой операции равна $\Theta(1)$.

```

1 function PUSH( $S[1..n]$ ,  $x$ )
2   if  $top \geq n$  then
3     return StackOverflow
4   end if
5    $top = top + 1$ 
6    $S[top] = x$ 
7 end function
```

Проверка стека на пустоту. Функция EMPTY возвращает значение *True*, если стек пуст, и значение *False* в противном случае. Вычислительная сложность операции равна $\Theta(1)$.

```

1 function EMPTY( $S[1..n]$ )
2   if  $top = 0$  then
3     return True
4   end if
5   return False
6 end function
```

Извлечение элемента из стека. Извлечение элемента из стека выполняется функцией POP. Она уменьшает значение переменной top на единицу и возвращает значение, находившееся в вершине стека $S[top+1]$. При попытке извлечения элемента из пустого стека возвращается специальное значение (stack underflow). Вычислительная сложность этой операции равна $\Theta(1)$.

```

1 function POP( $S[1..n]$ )
2   if EMPTY( $S$ ) then
3     return StackUnderflow
4   end if
5    $top = top - 1$ 
6   return  $S[top + 1]$ 
7 end function
```

Получение элемента из вершины стека. Чтение значения элемента, находящегося в вершине стека, без его удаления выполняется функцией TOP. Вычислительная сложность этой операции равна $\Theta(1)$.

```

1 function TOP( $S[1..n]$ )
2   if EMPTY( $S$ ) then
3     return StackUnderflow          /* Стек пуст */
4   end if
5   return  $S[top]$ 
6 end function

```

7.3. Реализация стека на базе односвязного списка

Если ограничения на максимальный размер стека заранее неизвестны, то подходящей структурой данных для реализации стека является односвязный список. При такой реализации все элементы стека хранятся в списке S . Помещение элемента в стек реализуется путем добавления нового узла в начало односвязного списка. Удаление элемента из стека равносильно удалению первого узла списка. Начальное значение указателя S на голову списка равно $null$ — стек пуст. Для хранения стека, содержащего k элементов, требуется $\Theta(k)$ ячеек памяти.

Помещение элемента в стек. Помещение элемента в стек реализуется операцией PUSH, которая добавляет новый узел в начало односвязного списка S . Вычислительная сложность этой операции равна $\Theta(1)$.

```

1 function PUSH( $S, x$ )
2   node = LINKEDLISTCREATENODE( $x$ )
3   if node = null then
4     return StackOverflow
5   end if
6   node.next =  $S$            /* Делаем головой списка новый узел */
7   return node
8 end function

```

Извлечение элемента из стека. Извлечение элемента из стека выполняется функцией POP. Она извлекает значение из первого узла и удаляет элемент. Головой списка (вершиной стека) становится второй узел. Вычислительная сложность операции равна $\Theta(1)$.

Проверка стека на пустоту. Функция EMPTY возвращает значение *True*, если стек пуст, и значение *False* в противном случае. Вычислительная сложность операции равна $\Theta(1)$.

Получение элемента из вершины стека. Чтение значения элемента, находящегося в вершине стека, без его удаления выполняется функцией TOP. Эта функция возвращает значение, хранящееся в первом узле списка. Вычислительная сложность операции равна $\Theta(1)$.

```

1 function POP(S)
2     if S = null then
3         return StackUnderflow
4     end if
5     value = S.value
6     node = S
7     S = S.next           /* Делаем второй элемент головой списка */
8     FREEMEMORY(node)
9     return value
10 end function

```

```

1 function EMPTY(S)
2     if S = null then
3         return True
4     end if
5     return False
6 end function

```

```

1 function TOP(S)
2     if EMPTY(S) then
3         return StackUnderflow           /* Стек пуст */
4     end if
5     return S.value
6 end function

```

7.4. Сравнение реализаций

Положительными сторонами реализации стека на основе массива является его непрерывное размещение в памяти, что повышает вероятность попадания смежных элементов массива в кеш-память процессора. Кроме того, при помещении элементов в стек не требуется обращаться к процедурам динамического выделения памяти, на выполнение которых требуется дополнительное время.

К недостаткам можно отнести фиксированность размера массива и возможный нерациональный расход памяти, обусловленный хранением *n* ячеек массива даже для пустого стека. Стек на базе связных списков занимает ровно столько памяти, сколько требуется под элементы, добавленные в него. В табл. 7.1 приведены вычислительные сложности операций АТД стек для худшего случая.

Таблица 7.1. Вычислительная сложность операций АТД стек

Операция	Реализация стека	
	Массив	Односвязный список
PUSH(S, x)	$\Theta(1)$	$\Theta(1)$
POP(S)	$\Theta(1)$	$\Theta(1)$
EMPTY(S)	$\Theta(1)$	$\Theta(1)$

7.5. Упражнения

1. Реализуйте стек на базе двусвязного списка.
2. Предложите реализацию операции ТОР на базе операций PUSH и POP.
3. На базе операций АТД стек разработайте алгоритм проверки правильности расстановок скобок в выражении. Например, запись ‘((())’ является корректной, а выражение ‘))()’ – таковым не является.
4. Разработайте алгоритм, который будет менять порядок элементов в стеке на обратный: первый элемент становится последним, второй – предпоследним и т. д.
5. Разработайте алгоритм операции MULTI PUSH(S, x_1, x_2, \dots, x_k), которая помещает в стек k заданных значений. Оцените ее вычислительную сложность. Продумайте обработку ошибок в случае нехватки памяти при помещении одного из k значений в стек. Как вернуть стек в исходное состояние?

8. Очереди

8.1. АТД очередь

Очередь (queue) – это абстрактный тип данных, представляющий собой список, в котором вставка (enqueue) и удаление элементов (dequeue) выполняется с разных концов. Добавление элементов выполняется с *заднего конца* (tail, rear, back), а удаление с *переднего* (front, head). Очередь функционирует по принципу «первым пришел – первым вышел» (First In – First Out, FIFO). Доступ к первому элементу очереди выполняется операцией FRONT (рис. 8.1). В качестве примера можно привести бесприоритетную очередь из покупателей в магазине: кто первым встал в очередь, тот первым будет обслужен.

Очереди широко используются в различных алгоритмах. Например, в алгоритме обхода графа в ширину (breadth first search) посещаемые вершины хранятся в очереди и извлекаются из нее по мере необходимости. Во многих многопоточных программах необходимо поддерживать очередь, в которую потоки-производители помещают данные, а потоки-потребители извлекают их из очереди и выполняют над ними заданные преобразования. Другим примером использования очередей, может служить программа, поддерживающая очередь печати принтера (спулер печати, printer spooler).

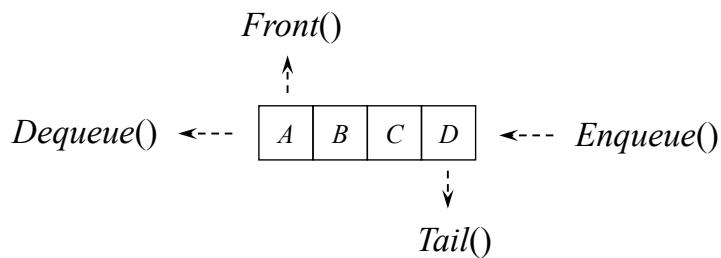


Рис. 8.1. Очередь из четырех элементов A, B, C, D .

Обозначим через Q очередь со значениями типа T , x – значение элемента.

Помещение элемента в очередь. Операция $\text{ENQUEUE}(Q, x)$ помещает элемент со значением x в конец очереди Q . Если очередь заполнена, функция возвращает специальное значение.

Извлечение первого элемента. Функция $\text{DEQUEUE}(Q, x)$ удаляет первый элемент очереди (с переднего конца) и возвращает его значение.

Получение первого элемента очереди. Операция $\text{FRONT}(Q)$ возвращает значение первого элемента в очереди. Если очередь пуста, то функция должна вернуть специальное значение.

Проверка очереди на отсутствие элементов. Функция $\text{EMPTY}(Q)$ возвращает значение True , если очередь пуста, и значение False в противном случае.

Рассмотрим пример выполнения описанных операций применительно к очереди на рис. 8.1. Результатом выполнения операции $\text{DEQUEUE}(Q)$ является значение A . Если еще раз обратиться к этой операции, то из очереди будет извлечен элемент B . Теперь, если выполнить операцию $\text{ENQUEUE}(Q, E)$, очередь примет состояние C, D, E .

Рассмотрим подходы к реализации очередей на базе массивов и связных списков.

8.2. Реализация очереди на базе связного списка

Значения элементов очереди хранятся в односвязном списке, каждый узел которого содержит элемент очереди (*value*) и указатель на следующий узел (*next*). Очередь Q характеризуется двумя атрибутами: $Q.front$ – указатель на первый элемент списка и $Q.tail$ – указатель на последний узел списка (рис. 8.2). Начальное значение полей $Q.front$ и $Q.tail$ очереди равно *null* (очередь пуста).

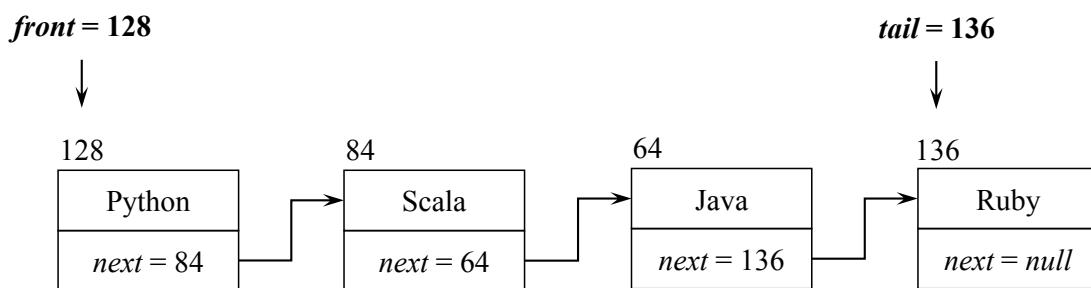


Рис. 8.2. Пример очереди на базе односвязного списка (над верхним левым углом каждого элемента показан его адрес в памяти).

Все операции очереди на базе односвязного списка имеют константную вычислительную сложность, так как реализуются путем проверки нескольких условных выражений и преобразований указателей.

Проверка очереди на отсутствие в ней элементов. Функция $\text{EMPTY}(Q)$ возвращает значение True , если в очереди отсутствуют элементы, и False в противном случае. Для реализации этой функции достаточно проверить, имеются ли в связном списке элементы – содержит ли $Q.front$ действительный адрес в памяти, отличный от *null*.

```

1 function EMPTY(Q)
2     if Q.front = null then
3         return True
4     end if
5     return False
6 end function

```

Получение первого элемента очереди. Функция FRONT возвращает значение первого элемента очереди. Если очередь пуста, функция возвращает специальное значение *QueueEmpty*.

```

1 function FRONT(Q)
2     if EMPTY(Q) then
3         return QueueEmpty
4     end if
5     return Q.front.value
6 end function

```

Помещение элемента в очередь. Добавление элемента в конец очереди реализуется функцией ENQUEUE. В памяти создается новый узел с заданным значением, после чего он добавляется в конец связного списка. Эффективный доступ к концу списка обеспечивается благодаря наличию указателя *Q.tail* на последний узел. Если список был пуст, то требуется скорректировать и указатель *Q.front* (рис. 8.3).

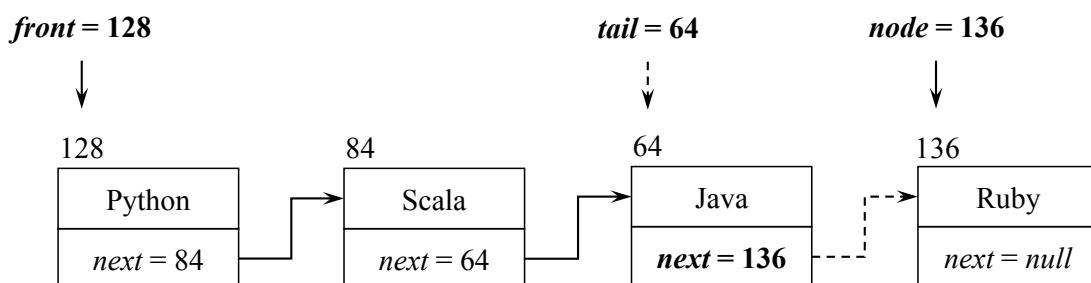


Рис. 8.3. Добавления элемента со значением «Ruby» в очередь на базе односвязного списка (штриховой линией показаны указатели, требующие корректировки).

Извлечение первого элемента очереди. Функция DEQUEUE удаляет из очереди первый узел и возвращает его значение. Если очередь содержала один элемент, то оба указателя *Q.front* и *Q.tail* принимают значение *null*. В случае попытки извлечения элемента из пустой очереди функция возвращает специальное значение *QueueEmpty*.

```

1 function ENQUEUE( $Q, x$ )
2      $node = \text{LINKEDLISTCREATENODE}(x)$ 
3     if EMPTY( $Q$ ) then
4          $Q.front = node$                                 /* Делаем новый узел головой */
5     else
6          $Q.tail.next = node$                           /* Добавляем узел в конец */
7     end if
8      $Q.tail = node$ 
9 end function

```

```

1 function DEQUEUE( $Q$ )
2     if EMPTY( $Q$ ) then
3         return QueueEmpty
4     end if
5      $value = Q.front.value$ 
6      $front = Q.front$ 
7      $Q.front = Q.front.next$                          /* Делаем головой второй узел */
8     FREEMEMORY( $front$ )
9     if  $Q.front = \text{null}$  then
10         $Q.tail = \text{null}$                             /* Список пуст */
11    end if
12    return  $value$ 
13 end function

```

8.3. Реализация очереди на базе кольцевого буфера

Кольцевой буфер (circular buffer, cyclic buffer) – это одномерный массив фиксированной длины n , в котором элементы логически организованы в виде кольца – за последним элементом идет первый.

Для работы с очередью используются две переменные $front$ – это индекс первого элемента очереди и $tail$ – индекс элемента, следующего за последним добавленным в очередь. Элементы очереди хранятся в ячейках массива с номерами $front, front + 1, \dots, tail - 1$. Новый элемент всегда записывается в позицию $tail$ массива – его хвост (конец очереди). Изначально $tail$ и $front$ имеют значение 1 (либо 0 при индексации массива с нуля). На рис. 8.4 показана последовательность добавления и извлечения элементов из очереди на базе кольцевого буфера.

При каждом добавлении элемента в конец очереди значение $tail$ увеличивается на 1. Аналогично поступаем при извлечении элемента из головы очереди – увеличиваем $front$ на 1. При этом надо контролировать значения переменных $front$ и $tail$: если они достигли конца массива, на-

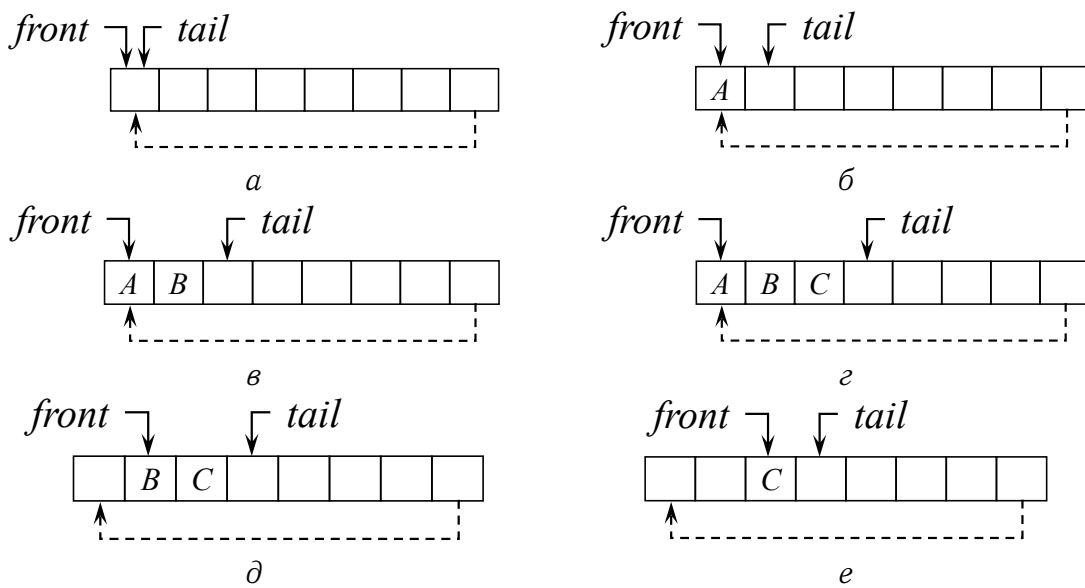
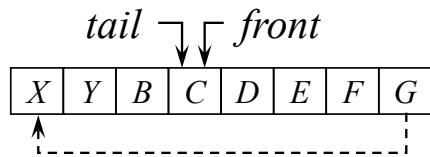


Рис. 8.4. Очередь на базе кольцевого буфера из 8 элементов:

a – начальное состояние; *б* – добавлен элемент *A*;*в* – добавлен элемент *B*; *г* – добавлен элемент *C*;*д* – извлечен элемент *A*; *е* – извлечен элемент *B*.

до записать в них номер первого элемента (соблюдаем условие кольцевой замкнутости).

Если $tail = front$, то очередь считается пустой. Здесь возникает проблема – как отличить пустую очередь от заполненной? На рис. 8.5 приведен пример пустой очереди: $front = tail$. В то же время, такая конфигурация возможна, если очередь полностью заполнена, например, элементами *C, D, E, F, G, X, Y, B* (хвост очереди «догнал» голову). Нам необходим способ определения, является ли очередь пустой или полностью заполненной.

Рис. 8.5. Пример неопределенного состояния очереди:
очередь пуста либо заполнена.

Один из возможных вариантов решения этой проблемы – использовать под очередь только $n - 1$ элементов массива. Тогда если $front = tail$, то очередь считается *пустой*. Если $front = 1$ и $tail = n$ или $front = tail + 1$, очередь считается *заполненной*. Условие заполненности очереди можно записать, используя операцию взятия остатка от деления:

$$tail \% n + 1 = front.$$

Обозначим через $Q.front$ – индекс первого элемента очереди, $Q.tail$ –

позицию последнего элемента очереди, $Q.size$ – размер массива (предельное количество элементов в очереди равно $Q.size - 1$), $Q.queue[1..Q.size]$ – массив элементов очереди.

Создание пустой очереди. Функция CREATE создает и возвращает пустую очередь для хранения $size$ элементов.

```

1 function CREATE(size)
2   Q.size = size + 1           /* Один элемент всегда свободен */
3   Q.queue = ALLOCATEMEMORY(Q.size)
4   Q.front = 1
5   Q.tail = 1
6   return Q
7 end function
```

Проверка очереди на отсутствие в ней элементов. Функция EMPTY возвращает значение *True*, если в очереди отсутствуют элементы, и *False* в противном случае. Для реализации этой функции достаточно проверить, содержат ли *front* и *tail* одинаковые значения.

```

1 function EMPTY(Q)
2   if Q.front = Q.tail then
3     return True
4   end if
5   return False
6 end function
```

Проверка очереди на переполнение. Функция FULL возвращает значение *True*, если очередь заполнена – *tail* содержит номер ячейки, предшествующей *front*. Мы используем операцию взятия остатка от деления, так как *tail* может указывать на последнюю ячейку массива.

```

1 function FULL(Q)
2   if Q.tail % Q.size + 1 = Q.front then
3     return True
4   end if
5   return False
6 end function
```

Получение первого элемента очереди. Функция FRONT возвращает значение первого элемента очереди. Если очередь пуста, функция возвращает значение *QueueEmpty*.

Помещение элемента в очередь. Добавление элемента в конец очереди реализуется функцией ENQUEUE. Значение элемента записывается в

```

1 function FRONT(Q)
2   if EMPTY(Q) then
3     return QueueEmpty
4   end if
5   return Q.queue[front]
6 end function

```

ячейку с номером $Q.tail$. После чего $Q.tail$ сдвигается вправо на одну позицию. Если мы записали элемент в последнюю свободную ячейку, то $tail$ будет равен $front$ (хвост догнал голову) и нам необходимо сдвинуть $front$ на одну позицию вправо – затереть первый элемент (рис. 8.6).

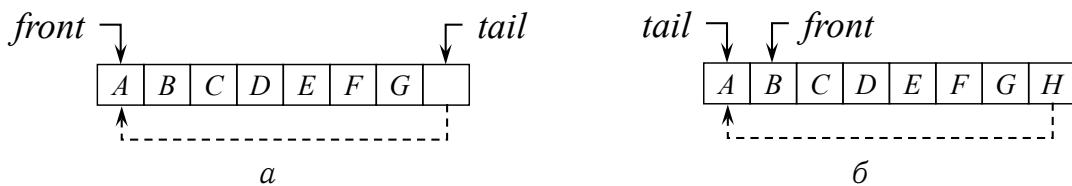


Рис. 8.6. Перезапись элемента при переполнении очереди:
a – очередь заполнена; *б* – добавлен элемент *H*, первый элемент затерт.

```

1 function ENQUEUE(Q, x)
2   Q.queue[Q.tail] = x
3   Q.tail = (Q.tail + 1) % Q.size
4   if FULL(Q) then
5     Q.front = (Q.front + 1) % Q.size
6   end if
7 end function

```

Извлечение первого элемента очереди. Функция DEQUEUE удаляет из очереди первый узел и возвращает его значение. Значение $Q.front$ циклически сдвигается вправо на одну позицию. В случае попытки извлечения элемента из пустой очереди функция возвращает значение *QueueEmpty*.

```

1 function DEQUEUE(Q)
2   if EMPTY(Q) then
3     return QueueEmpty
4   end if
5   value = Q.queue[Q.front]          /* Голова списка */
6   Q.front = (Q.front + 1) % Q.size
7   return value
8 end function

```

8.4. Сравнение реализаций

Положительной стороной реализации очереди на базе связного списка является отсутствие ограничения на размер очереди, отрицательной стороной – необходимость обращение к функциям динамического выделения памяти, что требует дополнительного времени. Также последнее приводит к тому, что элементы очереди располагаются не в непрерывном участке памяти, а по произвольным адресам. Это влечет за собой неэффективное использование кеш-памяти процессора. В языках программирования с автоматическим управлением динамической памятью (Java, C#) удаление неиспользуемых объектов возложено на *сборщик мусора* (garbage collector) – специальный поток (процесс), который автоматически запускается и удаляет неиспользуемые объекты. При активном использовании функций управления памятью (помещение и удаление элементов из очереди) вероятность запуска сборщика мусора достаточно высока. Это может привести к тому, что он займет часть ресурсов процессора и окажет негативное влияние на время выполнения программы.

Реализация очереди на базе кольцевого буфера лишена описанных выше недостатков. Однако, она требует предварительных оценок максимального размера очереди для минимизации вероятности ее переполнения. В табл. 9.1 приведены вычислительные сложности операций очереди для худшего случая.

Таблица 8.1. Вычислительная сложность операций АТД очередь

Операция	Реализация очереди	
	Односвязный список	Кольцевой буфер
ENQUEUE(Q, x)	$\Theta(1)$	$\Theta(1)$
DEQUEUE(Q)	$\Theta(1)$	$\Theta(1)$
FRONT(Q)	$\Theta(1)$	$\Theta(1)$
EMPTY(Q)	$\Theta(1)$	$\Theta(1)$

8.5. Упражнения

- Предложите реализацию очереди на базе односвязного списка, при условии, что известен указатель только на его голову. Оцените вычислительную сложность операций.
- Реализуйте на базе двусвязного списка операции для работы с двухсторонней очередью – деком. Оцените их вычислительную сложность.
- Подумайте, как реализовать очередь на базе кольцевого буфера, в

котором вместо значения *tail* используется поле *count* – текущее количество элементов в очереди.

4. Предложите реализацию АТД очередь с приоритетом на базе связного списка. Оцените время выполнения операций `INSERT`, `DELETEMIN` и `MIN`.

5. Предложите реализацию АТД очередь с приоритетом на базе кольцевого буфера. Оцените время выполнения операций `INSERT`, `DELETEMIN` и `MIN`. Сравните эффективность этой реализации с реализацией на базе связного списка.

6. Для очереди на базе односвязного списка реализуйте операцию `SIZE`, возвращающую число элементов в очереди.

7. Аналогично, для очереди на базе кольцевого буфера реализуйте операцию `SIZE`, возвращающую число элементов в очереди.

9. Бинарные деревья

9.1. Корневые деревья

Корневое дерево (rooted tree) – это структура данных, которая представляет собой совокупность связанных *узлов* (nodes), один из которых является *корнем дерева* (root).

Ребра (edges) связывают узлы дерева и устанавливают между ними отношение «родитель-дочерний» (parent-child). На рис. 9.1 приведен пример корневого дерева.

Узел, который не имеет дочерних вершин, называется *внешним узлом* (external node), или *листом* (leaf node). На рис. 9.1 это узлы: 4, 8, 9, 6 и 7. Аналогично, узел дерева, имеющий дочерние вершины называется *внутренним узлом* (internal node).

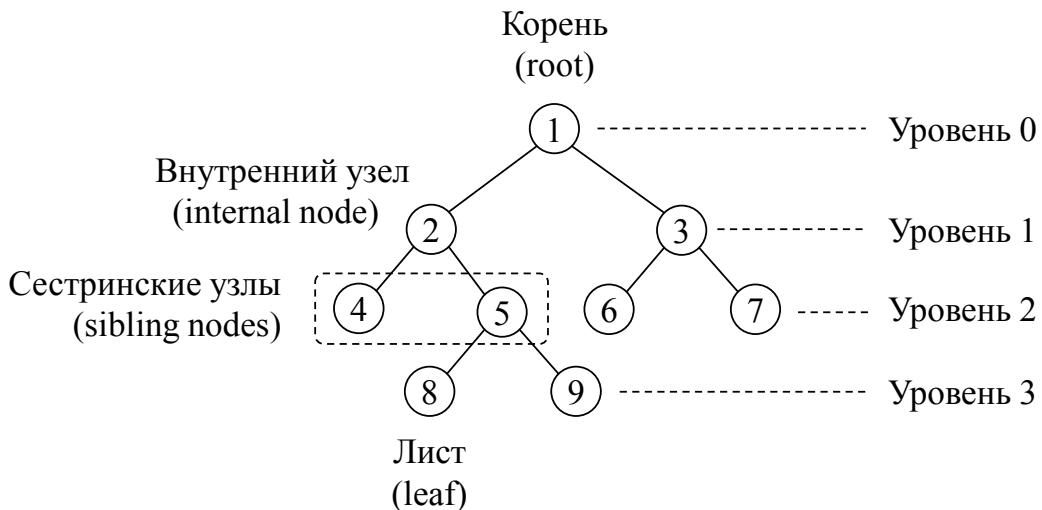


Рис. 9.1. Корневое дерево из 9 элементов (высота дерева 3).

Любой внутренний узел по отношению к своим дочерним вершинам (child nodes) называется *родительским* (parent node). Узлы, имеющие общего родителя называются *родственными*, или *сестринскими узлами* (sibling nodes). Любой узел в поддереве некоторой вершины называется ее *потомком* (descendant), а вершина для этих узлов является *предком* (ancestor).

Степень узла (node degree) – это количество его дочерних узлов.

Высота узла (node height) – количество ребер в длиннейшем пути от узла до листа.

Высота дерева (tree height) – количество ребер в длиннейшем пути от корня до листа.

Глубина узла (node depth) или *уровень узла* (node level) – количество ребер в пути от узла до корня.

Рассмотрим подробнее введенные определения. На рис. 9.1 приведено дерево из 9 узлов. Вершина 2 является родителем для вершин 4 и 5, а также предком для узлов 4, 5, 8 и 9. Узлы 4 и 5 являются сестринскими. Высота дерева равна 3, а количество уровней – 4. Степень узла 5 равна 2, а его высота 1.

9.2. Бинарные деревья

Бинарное дерево (двоичное дерево, binary tree) – это корневое дерево, в котором каждый узел имеет не более двух дочерних вершин (0, 1 или 2 вершины).

Полное бинарное дерево (full binary tree) – это бинарное дерево, в котором каждый узел имеет 0 или 2 дочерних узла. Пример такого дерева приведен на рис. 9.1.

Завершенное бинарное дерево (complete binary tree) – это бинарное дерево, в котором каждый уровень, возможно за исключением последнего, полностью заполнен узлами, а заполнение последнего уровня осуществляется слева направо (рис. 9.2).

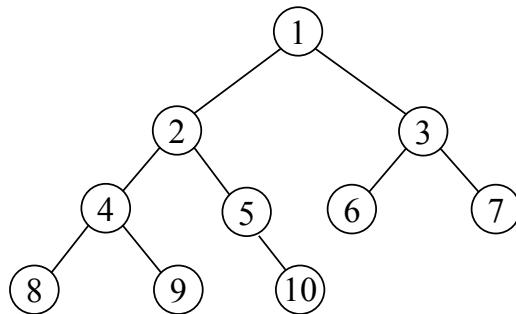


Рис. 9.2. Завершенное бинарное дерево из 10 элементов.

Высота завершенного бинарного дерева из n узлов равна $\lfloor \log_2 n \rfloor$, а число вершин с высотой $h \in \{0, 1, \dots, \lfloor \log_2 n \rfloor\}$ есть $\lceil n/2^{h+1} \rceil$.

Совершенное бинарное дерево (perfect binary tree) – это бинарное дерево, в котором все листья имеют одинаковую глубину (рис. 9.3).

При числе узлов n минимальной высотой $h = \log_2(n + 1) - 1$ обладает совершенное бинарное дерево, а максимальной высотой – бинарное дерево, в котором каждый внутренний узел имеет единственную дочернюю вершину (дерево вырождается в цепочку высоты $n - 1$). Следовательно, высота h любого бинарного дерева из n узлов ограничена снизу и сверху

$$\lceil \log_2(n + 1) \rceil - 1 \leq h \leq n - 1.$$

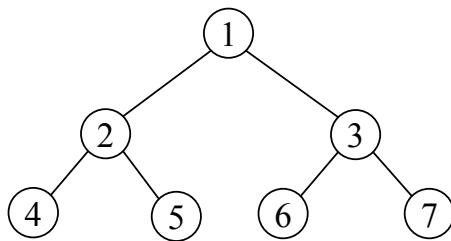


Рис. 9.3. Совершенное бинарное дерево высоты 2.

9.3. Представление деревьев в памяти

9.3.1. Представление на базе структур и указателей

При таком представлении память под узлы дерева выделяется динамически и они располагаются по произвольным адресам. Каждый узел содержит следующие поля:

- *parent* – указатель на родительский узел;
- *left* – указатель на левый дочерний узел;
- *right* – указатель на правый дочерний узел.

Указатель *parent* на родительский узел не является обязательным. Он необходим, если требуется продвигаться по дереву от дочерних узлов к родительским. Если узел x дерева является листом, то поле $x.left = null$ и $x.right = null$.

9.3.2. Представление на базе массива

Завершенное бинарное дерево из n узлов можно разместить в одномерном массиве $A[1..n]$ длины n . Для определенности будем полагать, что индексация элементов в массиве начинается с 1. Все узлы дерева размещаются в ячейках массива в порядке их обхода в ширину. Корень дерева хранится в первой ячейке массива. Для любой вершины, размещенной в ячейки i , левый дочерний узел расположен в позиции $2i$, а правый дочерний узел в позиции $2i+1$. Родительский узел вершины i находится в ячейке $\lfloor i/2 \rfloor$. Если значения $2i$ или $2i + 1$ больше n , то у вершины i отсутствует соответствующий дочерний узел. На рис. 9.4 показано представление в виде массива дерева с рис. 9.2.

Хранение дерева в виде массива положительно сказывается на работе кеш-памяти процессора, так как узлы дерева размещены в непрерывном участке памяти.

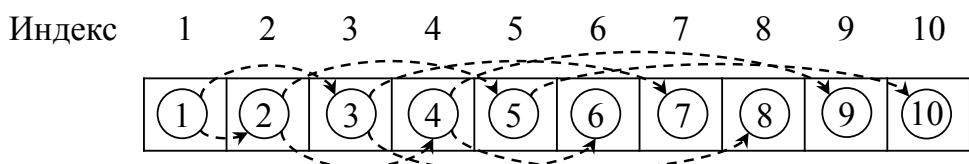


Рис. 9.4. Представление завершенного бинарного дерева с рис. 9.2 в виде одномерного массива $A[1..10]$.

9.3.3. Представление деревьев с произвольным ветвлением

Для хранения деревьев, в которых число дочерних узлов у каждой вершины не превышает заданной константы k , можно обобщить подход на базе указателей. Каждый узел дерева будет содержать k указателей: $child_1, child_2, \dots, child_k$.

Если значение k заранее неизвестно, то такая схема работать не будет. Также возможна ситуация, когда большая часть узлов не будет иметь k дочерних узлов и часть памяти под указатели будет расходоваться напрасно.

Для случая произвольного числа дочерних узлов можно использовать подход «левый дочерний и правый сестринский» (left-child, right-sibling). При таком представлении каждый узел дерева содержит следующие поля:

- *parent* – указатель на родительский узел;
- *child* – указатель на крайний левый дочерний узел;
- *sibling* – указатель на ближайший справа сестринский узел.

Если узел x не имеет дочерних узлов, то $x.child = null$, а если он является крайним справа дочерним узлом своего родителя, то $x.sibling = null$.

На рис. 9.6 показано представление «левый дочерний и правый сестринский» для дерева с рис. 9.5.

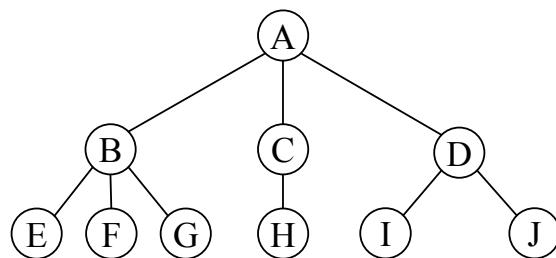


Рис. 9.5. Дерево степени 3.

Представление «левый дочерний и правый сестринский» позволяет экономно расходовать память для хранения узлов дерева с произвольным ветвлением.

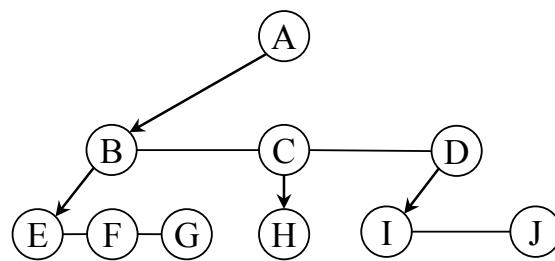


Рис. 9.6. Представление дерева в виде «левый дочерний и правый сестринский» (стрелками показаны указатели *child*, горизонтальные ребра – указатели *sibling*).

9.4. Обходы бинарных деревьев

Во многих алгоритмах, использующих деревья, возникает необходимость перебрать все узлы дерева начиная с корня. Такая процедура называется *обходом дерева* (tree traversal). Каждая вершина должна быть посещена один раз. Посещение вершины может быть связано с выполнением некоторых вычислений и/или обработкой данных, хранящихся в ней.

Обход бинарного дерева можно выполнять в ширину и в глубину. Все способы обходов дерева начинают свою работу с корня.

При *обходе в ширину* (breadth first search, BFS) узлы дерева посещаются слева направо, уровень за уровнем. Например, при обходе в ширину дерева на рис. 9.2 вершины будут посещены в следующем порядке:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

При *обходе в глубину* (depth first search, DFS) сперва посещаются все дочерние вершины левого поддерева и уже потом происходит переход к правому поддереву текущего узла.

Существует три способа обхода дерева в глубину, различающихся порядком посещения вершин и их дочерних узлов.

1. *Обход в прямом порядке* (префиксный обход, pre-order traversal) – каждый узел посещается до того, как рекурсивно посещены его потомки. Для каждого узла x рекурсивно выполняется последовательность действий: посетить узел x ; обойти левое поддерево; обойти правое поддерево. При таком обходе узлы дерева на рис. 9.2 будут посещены в следующем порядке:

1, 2, 4, 8, 9, 5, 10, 3, 6, 7.

2. *Симметричный обход* (инфиксный обход, in-order traversal) – сперва посещаются потомки левого поддерева узла, затем сам узел и затем посещаются потомки правого поддерева. Для каждого узла x рекурсивно выполняется последовательность действий: обойти левое поддерево, посетить узел x , обойти правое поддерево. Результат симметричного обхода

дерева на рис. 9.2:

8, 4, 9, 2, 5, 10, 1, 6, 3, 7.

3. *Обход в обратном порядке* (постфиксный обход, post-order traversal) – каждый узел посещается после того, как рекурсивно посещены его потомки. Для каждого узла x рекурсивно выполняется последовательность действий: обойти левое поддерево, обойти правое поддерево, посетить узел x . Результат обхода в обратном порядке дерева на рис. 9.2:

8, 9, 4, 10, 5, 2, 6, 7, 3, 1.

9.5. Упражнения

1. Оцените количество листьев в совершенном бинарном дереве из n узлов.
2. Оцените количество внутренних узлов в совершенном бинарном дереве из n вершин.
3. Разработайте процедуру перебора всех узлов бинарного дерева поиска в порядке убывания ключей.
4. Запишите результат обхода в ширину дерева с рис. 9.1.
5. Выполните прямой (префиксный) обход дерева на рис. 9.1.
6. Предложите способ представления бинарного дерева с использованием подхода «левый дочерний и правый сестринский».
7. Разработайте алгоритм посещения всех узлов дерева, представленного с использованием подхода «левый дочерний и правый сестринский».

10. Бинарные деревья поиска

Бинарные деревья поиска предназначены для реализации *множеств* (set) и *ассоциативных массивов* (associative array), или, как их еще называют, *словарей* (dictionaries). Ассоциативные массивы отличаются от множеств тем, что последние хранят только ключи, а ассоциативные массивы с каждым ключом хранят и некоторое значение (см. раздел 5.3.2).

Структура бинарного дерева поиска позволяет эффективно реализовать упорядоченный перебор всех ключей, находящихся в узлах дерева. Поэтому такие деревья широко используют для реализации АТД упорядоченное множество (ordered set).

10.1. Структура бинарного дерева поиска

Бинарное дерево поиска (binary search tree) – это бинарное дерево, в котором каждый узел содержит *ключ* (key), причем ключи в узлах левого поддерева меньше ключа родительского узла, а ключи в узлах правого поддерева больше ключа родительского узла.

Подразумевается, что известна (задана) процедура сравнения ключей бинарного дерева поиска. Как правило, для базовых типов данных (целые и вещественные числа, символы и логические значения) операция сравнения выполняется за время $O(1)$. Если же ключ относится к составным типам (например, строка), то вычислительная сложность операции сравнения уже будет зависеть от длины ключа.

На рис. 10.1 показан пример бинарного дерева поиска, в узлах которого записаны значения ключей.

Из определения следует, что в бинарном дереве поиска не может присутствовать двух узлов с одинаковыми ключами.

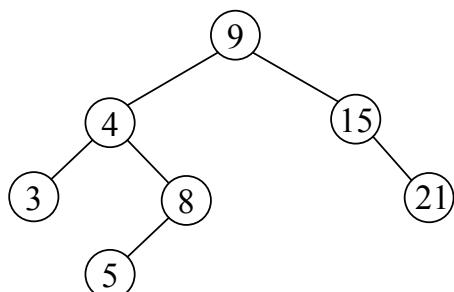


Рис. 10.1. Бинарное дерево поиска из семи узлов.

Рассмотрим основные операции, которые поддерживает бинарное дерево поиска. Будем считать, что каждый узел дерева содержит поля:

- *key* – ключ узла;
- *left* – указатель на левый дочерний узел;
- *right* – указатель на правый дочерний узел;
- *value* – данные (необходимо только при реализации АТД ассоциативный массив).

10.2. Создание узла

Создание нового узла дерева осуществляется функцией CREATE NODE. Она выделяет для него память и инициализирует поля начальными значениями. Созданный узел – это вырожденное бинарное дерево поиска, состоящее из одной вершины. Вычислительная сложность этой операции $O(1)$. Здесь, как и ранее, мы считаем, что вычислительная сложность операций выделения и освобождения памяти равна $O(1)$.

```

1 function CREATE NODE(key, value)
2     node = ALLOCATE MEMORY()
3     if node ≠ null then
4         node.key = key
5         node.value = value
6         node.left = null
7         node.right = null
8     end if
9     return node
10 end function
```

10.3. Добавление узла

Добавление узла в дерево выполняется функцией INSERT. Для этого необходимо найти лист, который станет родителем для новой вершины. Поиск листа начинается с корня дерева. Если добавляемый ключ *key* меньше ключа текущего узла *tree*, то делаем текущим левый дочерний узел, в противном случае текущим становится правый дочерний элемент. Спуск по дереву продолжается до тех пор, пока текущий узел не примет значение *null*. Последнее означает, что мы достигли листа дерева. Далее создается новый узел, который становится дочерним по отношению к найденному листу.

На рис. 10.2 показан процесс добавления в дерево поиска ключей 9, 15, 7, 8 и 3.

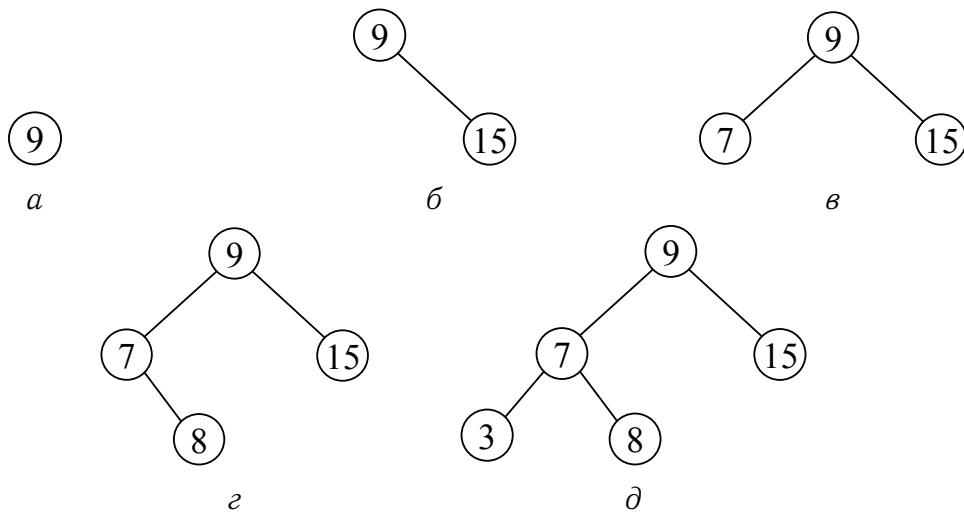


Рис. 10.2. Процесс добавления в бинарное дерево поиска ключей:

а – добавление ключа 9; б – добавление ключа 15;
в – добавление ключа 7; г – добавление ключа 8; д – добавление ключа 3.

```

1 function INSERT(tree, key, value)
2   parent = null
3   while tree ≠ null do                                /* Отыскиваем листовой узел */
4     parent = tree
5     if key < tree.key then
6       tree = tree.left
7     else if key > tree.key then
8       tree = tree.right
9     else
10      return                                         /* Заданный ключ уже в дереве */
11    end if
12  end while
13  node = CREATE NODE(key, value)
14  if key < parent.key then    /* Связываем новый узел с листом */
15    parent.left = node
16  else
17    parent.right = node
18  end if
19 end function

```

Вычислительная сложность операции добавления определяется числом сравнений ключей, которые выполняет функция INSERT. При поиске листового узла для новой вершины нам требуется выполнить порядка $O(h)$

сравнений ключей, где h – это высота бинарного дерева поиска. Таким образом вычислительная сложность операции добавления ключа равна $O(h)$. Далее мы рассмотрим оценку высоты h бинарного дерева поиска для среднего и худшего случаев.

10.4. Поиск узла по ключу

Поиск узла по ключу реализует функция LOOKUP. Поиск начинается с корня дерева. Если искомый ключ key меньше ключа текущего узла, то делаем текущим левый дочерний узел, в противном случае текущим становится правый дочерний элемент. Спуск по дереву продолжается до тех пор, пока мы не найдем требуем узел или текущий узел не примет значение $null$. Последнее означает, что мы достигли листа дерева и ключ не найден.

В лучшем случае искомый ключ находится в корне дерева, что требует $O(1)$ операций сравнения. В остальных случаях (в среднем и худшем) искомый ключ может находиться как во внутреннем узле, так и в листе дерева. В любом случая число сравнений, выполняемых при поиске ключа, имеет порядок $O(h)$.

```

1 function LOOKUP(tree, key)
2   while tree ≠ null do
3     if key < tree.key then
4       tree = tree.left
5     else if key > tree.key then
6       tree = tree.right
7     else
8       return tree                                /* Узел найден */
9     end if
10   end while
11   return null
12 end function
```

10.5. Поиск минимального и максимального узлов

Структура бинарного дерева поиск однозначно определяет положение узлов, которые содержат ключ с наименьшим и наибольшим значениями ключа (рис. 10.3).

Узлом с наименьшим значением ключа является самый левый потомок корня. Для его нахождения следует двигаться от корня по левым указателям узлов.

Узлом с наибольшим значением ключа является самый правый потомок корня. Для его нахождения следует двигаться от корня по правым указателям узлов.

Вычислительная сложность операций MIN и MAX зависит от высоты дерева и распределения ключей в нем. В общем случае число операций перехода по указателям *left* или *right* имеет порядок $O(h)$.

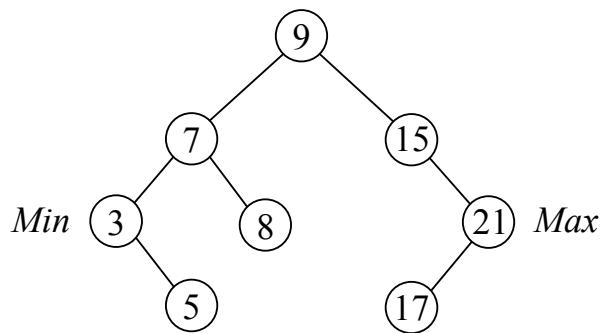


Рис. 10.3. Узлы с минимальным (min) и максимальным (max) значениями ключей.

```

1 function MIN(tree)
2   while tree.left ≠ null do
3     tree = tree.left
4   end while
5   return tree
6 end function

7 function MAX(tree)
8   while tree.right ≠ null do
9     tree = tree.right
10  end while
11  return tree
12 end function

```

10.6. Поиск следующего и предыдущего узлов

Поиск узла, который содержит ключ, следующий за ключом *key*, выполняется функцией *SUCCESSOR*. Если узел с ключом *key* в дереве отсутствует, функция возвращает специальное значение *KeyNotFound*.

```

1 function SUCCESSOR(tree, key)
2     successor = null
3     while tree ≠ null do
4         if key < tree.key then
5             successor = tree
6                 /* Большой предок */
7             tree = tree.left
8         else if key > tree.key then
9             tree = tree.right
10        else
11            if tree.right ≠ null then          /* Ключ key найден */
12                return MIN(tree.right)
13            end if
14            return successor
15        end if
16    end while
17    return KeyNotFound
18 end function

18 function PREDECESSOR(tree, key)
19     predecessor = null
20     while tree ≠ null do
21         if key < tree.key then
22             tree = tree.left
23         else if key > tree.key then
24             predecessor = tree
25             tree = tree.right
26         else
27             if tree.left ≠ null then          /* Ключ key найден */
28                 return MAX(tree.left)
29             end if
30             return predecessor
31         end if
32     end while
33     return KeyNotFound
34 end function

```

Сперва выполняется поиск узла *node* с заданным ключом *key*. Если он имеет правое поддерево, то *следующий узел* (*successor*) – это минимальная вершина в его правом поддереве. Например, на рис. 10.3 следующей для ключа 15 является вершина 17 – минимальный ключ в правом поддереве узла 15.

Если же у узла *node* нет правого поддерева, то мы отыскиваем ближайшую вершину-предок, которая содержит ключ больше *key*. Поиск такой вершины-предка реализуется одновременно с поиском узла *node* при спуске от корня дерева по левым указателям вершин. На рис. 10.3 следующим ключом для узла 5 является вершина 7 (указатель на узел 7 сохраняется в переменной *successor*).

Функция PREDECESSOR находит узел, который содержит ключ, предшествующий заданному ключу *key*. Если узел с ключом *key* в дереве отсутствует, функция возвращает специальное значение *KeyNotFound*.

На рис. 10.3 предшествующим узлом для ключа 7 является вершина 5 – максимальный ключ в левом поддереве 7. Для ключа 15 предшествующим является узел 9, указатель на него сохраняется в переменной *predecessor* при переходах по указателям *right*.

Вычислительная сложность этих операций равна $O(h)$, так как нам необходимо либо отыскивать потомка узла *node*, либо находить узел с экстремальным значением ключа.

10.7. Обход дерева в упорядоченной последовательности

Структура бинарного дерева поиска позволяет эффективно реализовать упорядоченный обход его узлов. Для перебора всех вершин в порядке возрастания ключей необходимо выполнить симметричный обход дерева начиная с корня (in-order traversal). Сперва посещаются потомки левого под дерева узла, затем сам узел и затем потомки правого под дерева. Ниже приведен псевдокод функции TRAVERSEINORDER. Предполагается, что в функции PROCESSNODE реализована обработка узла (например, его вывод на экран или сохранение в другой структуре данных). Вычислительная сложность симметричного обхода дерева равна $\Theta(n)$, так как нам требуется посетить все узлы.

Для перебора всех вершин в порядке убывания ключей необходимо также выполнить симметричный обход дерева, но с другим порядком посещения узлов – сперва посещаем узлы правого под дерева, а затем – левого.

```

1 function TRAVERSEINORDER(tree)
2   if tree ≠ null then
3     TRAVERSEINORDER(tree.left)
4     PROCESSNODE(tree)
5     TRAVERSEINORDER(tree.right)
6   end if
7 end function

```

10.8. Удаление узла

Функция DELETE удаляет из дерева узел с заданным ключом *key*. Заметим, что удаляемой вершиной может оказаться корень дерева, поэтому функция DELETE возвращает в качестве результата указатель на новый корень дерева.

Удаление начинается с поиска узла *node*, который содержит заданный ключ *key*. Если узел не найден (*node = null*), функция возвращает исходный корень в качестве результата своей работы. Если же искомый узел найден, то возможны три случая его размещения в дереве. Анализ этих случаев реализован в функции DELETENODE, которая удаляет найденный узел *node* и возвращает указатель на корень дерева.

```

1 function DELETE(tree, key)
2   parent = null
3   node = tree
4   while node ≠ null and node.key ≠ key do          /* Поиск узла */
5     parent = node                                /* Родитель узла */
6     if key < node.key then
7       node = node.left
8     else
9       node = node.right
10    end if
11  end while
12  if node = null then                           /* Узел не найден */
13    return tree
14  end if
15  return DELETENODE(tree, node, parent)
16 end function

```

Случай 1. Удаляемый узел *node* не имеет дочерних вершин (является листом). В этом случае достаточно соответствующему указателю *left* либо *right* родительского узла *parent* присвоить значение *null*. Выбор указателя *left* или *right* зависит от того, в каком поддереве родителя находится узел *node*. После корректировки указателя функцией REPLACENODE выпол-

```

1 function DELETENODE(tree, node, parent)
2   if node.left = null then                                /* Случай 1 или 2 */
3     REPLACENODE(parent, node, node.right)
4     if parent = null then
5       tree = node.right                                     /* Удалили корень дерева */
6     end if
7   else if node.right = null then                            /* Случай 2 */
8     REPLACENODE(parent, node, node.left)
9     if parent = null then
10      tree = node.left                                    /* Удалили корень дерева */
11    end if
12   else                                                 /* Случай 3 – два дочерних узла */
13     min = node.right        /* Поиск узла с минимальным ключом */
14     minparent = min      /* Родитель минимального узла */
15     while min.left ≠ null do
16       minparent = min
17       min = min.left
18     end while
19     REPLACENODE(parent, node, min)
20     if parent = null then
21       tree = min                                         /* Удалили корень дерева */
22     end if
23     if node.right ≠ min then                            /* Случай 3.1 */
24       minparent.left = min.right
25       min.left = node.left
26       min.right = node.right
27     else                                              /* Случай 3.2 */
28       min.left = node.left
29     end if
30   end if
31   FREEMEMORY(node)
32   return tree
33 end function

34 function REPLACENODE(parent, node, child)
35   if parent ≠ null then          /* Связываем parent с узлом child */
36     if node.key < parent.key then
37       parent.left = child
38     else
39       parent.right = child
40     end if
41   end if
42 end function

```

няется освобождение памяти из-под вершины *node*. На рис. 10.4 показан пример удаления листового узла 25 из бинарного дерева поиска: указатель *left* узла 30 установлен в значение *null*.

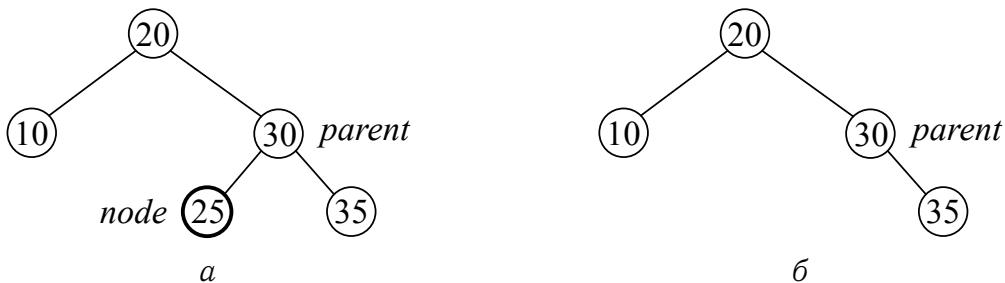


Рис. 10.4. Удаление листового узла 25 из бинарного дерева поиска:
а – состояние дерева до удаления узла 25;
б – состояние дерева после корректировки указателя *left* узла 30.

Случай 2. Удаляемый узел *node* имеет одну дочернюю вершину. Если отсутствует левый дочерний узел (*node.left = null*), заменяем удаляемую вершину *node* ее правым дочерним узлом *node.right*. Для этого достаточно связать родительский узел и узел *node.right*: присвоить указателю *left* либо *right* родителя значения *node.right*. Последнее выполняется функцией *REPLACENODE*, которая связывает родительский узел *parent* вершины *node* с его потомком *child*.

Аналогично, при отсутствии правого дочернего узла (*node.right = null*), заменяем удаляемую вершину *node* ее левым дочерним узлом.

Если удаляемый узел является корнем дерева (*parent = null*), новым корнем становится левый или правый дочерний узел вершины *node*.

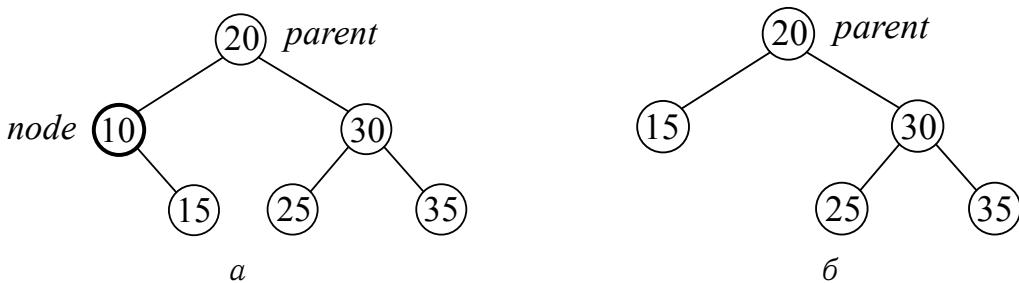


Рис. 10.5. Удаление узла 10 из бинарного дерева поиска (случай 2):
а) состояние дерева до удаления узла;
б) состояние дерева после корректировки указателя *left* узла 20.

На рис. 10.5 приведен пример удаления узла 10 с одной правой дочерней вершиной. Указателю *left* вершины 20 присваивается адрес вершины 15 (вершина 10 заменяется вершиной 15).

Случай 3. Удаляемый узел *node* имеет две дочерние вершины. Необходимо заменить узел его другим узлом, чтобы сохранить свойства

бинарного дерева поиска. Для этого мы можем использовать предшествующий (predecessor) либо следующий (successor) за *node* узел. Для определенности будем заменять узел *node* вершиной со следующим по порядку ключом (successor) – это узел с минимальным ключом в правом поддереве *node*.

Работа функции `DELETENODE` начинается с поиска в правом поддереве вершины *node* минимального узла *min* и его родителя *minparent*. Затем удаляемый узел *node* заменяется вершиной *min*. Это реализуется функцией `REPLACENODE`, которая связывает узлы *parent* и *min*.

Если узел *node* не является родителем узла *min* (*node.right* ≠ *node*) и *minparent*, обозначим этот случай как 3.1), то мы выполняем следующие шаги:

- делаем родителем правого дочернего узла вершины *min* его родительский узел: *minparent.left* = *min.right*;
- устанавливаем в качестве левого и правого поддеревьев узла *min* соответственно левое и правое поддеревья узла *node*.

Если же узел *node* является родителем узла *min* (*node.right* = *min* = *minparent*, случай 3.2), то достаточно сделать левым поддеревом узла *min* левое поддерево узла *node*.

Если удаляемая вершина является корнем, то мы соответствующим образом корректируем значение переменной *root* и возвращаем ее значение из функции `DELETENODE`.

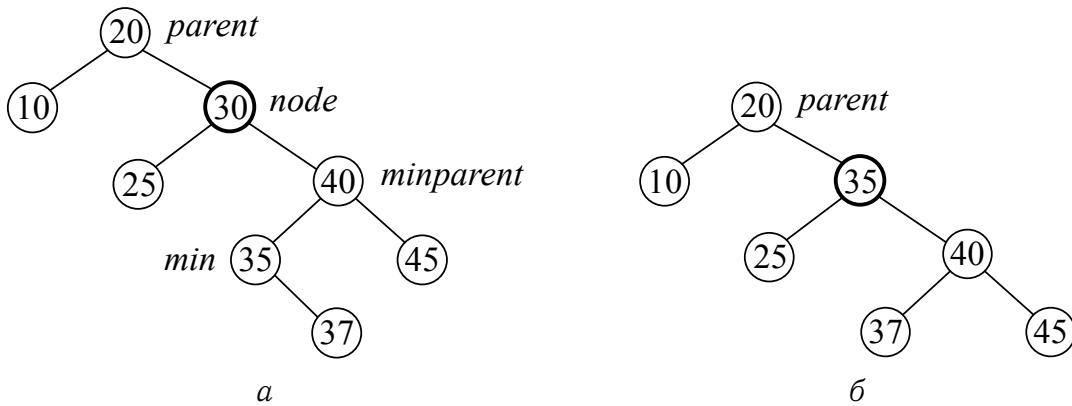


Рис. 10.6. Удаление узла 30 из бинарного дерева поиска (случай 3.1):

а – состояние дерева до удаления узла; *б* – состояние дерева после замены узла 30 вершиной 35 и корректировки указателя *left* узла 40.

На рис. 10.6 приведен пример удаления вершины 30, которая имеет два дочерних узла. Элементом с минимальным ключом в правом поддереве вершины 30 является узел 35, который не является дочерней вершиной узла 30, следовательно, имеет место случай 3.1. Вершина 30 заменяется узлом 35: в указатель *right* узла 20 записывается адрес вершины 35. Далее новым родителем для узла 37 становится вершина 40, а левым и правым

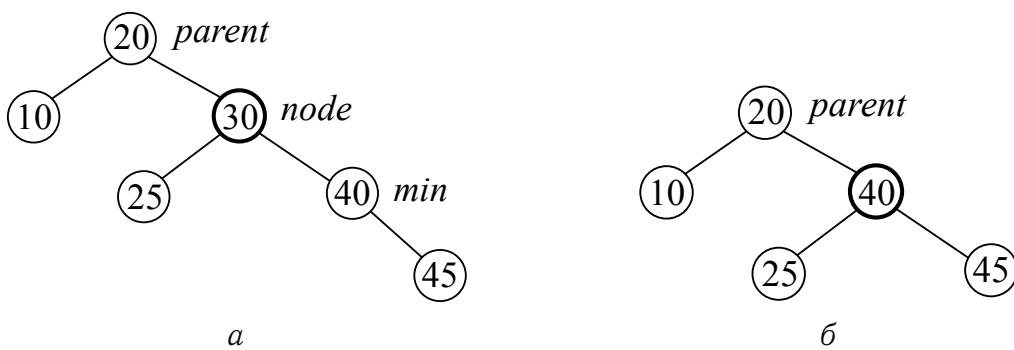


Рис. 10.7. Удаление узла 30 из бинарного дерева поиска (случай 3.2):

а – состояние дерева до удаления узла;

б – состояние дерева после замены узла 30 вершиной 40.

поддеревьями для узла 35 становятся вершины 25 и 40 соответственно.

На рис. 10.7 приведен пример удаления узла 30, что соответствует случаю 3.2. Минимальный узел 40 является дочерним по отношению к вершине 30. Узел 30 заменяется узлом 40, а вершина 25 становится левым дочерним узлом для элемента 40.

Вычислительная сложность удаления узла из бинарного дерева поиска имеет порядок $O(h)$. За $O(h)$ операций мы отыскиваем удаляемый узел и за $O(h)$ действий разрешаем один из трех случаев (случай 3 требует поиска узла с минимальным ключом).

10.9. Удаление дерева

Освобождение памяти из-под всех узлов дерева выполняется путем его обхода в обратном порядке (post-order traversal). Память из под узла освобождается только после того, как рекурсивно освободили память из-под его потомков. Вычислительная сложность процедуры `DELETETREE` равна $\Theta(n)$.

```

1 function DELETETREE(tree)
2   if tree ≠ null then
3     TRAVERSEINORDER(tree.left)
4     TRAVERSEINORDER(tree.right)
5     FREEMEMORY(tree)           /* Освобождаем память */
6   end if
7 end function
```

10.10. Высота бинарного дерева поиска

Вычислительная сложность операций бинарного дерева поиска зависит от его высоты h , которая динамически изменяется при добавлении и удалении узлов.



Рис. 10.8. Худший случай добавления ключей в бинарное дерево поиска:
а – ключи добавлены в порядке убывания (7, 3, 2);
б – ключи добавлены в порядке возрастания (7, 11, 15).

Таблица 10.1. Вычислительная сложность операций бинарного дерева поиска

Операция	Средний случай	Худший случай
$\text{INSERT}(tree, key, value)$	$O(\log n)$	$O(n)$
$\text{LOOKUP}(tree, key)$	$O(\log n)$	$O(n)$
$\text{DELETE}(tree, key)$	$O(\log n)$	$O(n)$
$\text{MIN}(tree)$	$O(\log n)$	$\Theta(n)$
$\text{MAX}(tree)$	$O(\log n)$	$\Theta(n)$
$\text{PREDECESSOR}(tree, key)$	$O(\log n)$	$O(n)$
$\text{SUCCESSOR}(tree, key)$	$O(\log n)$	$O(n)$
$\text{DELETETREE}(tree)$	$\Theta(n)$	$\Theta(n)$

В худшем случае все n ключей добавляются в дерево в упорядоченной последовательности (рис. 10.8). Дерево вырождается в цепочку из n узлов, связанных указателями *left* (если ключи добавляются от больших к меньшим) или указателями *right* (если ключи добавляются по возрастанию). Высота дерева в этом случае равна $n - 1$, и, как следствие, вычислительная сложность операций INSERT , LOOKUP , DELETE , MIN , MAX , PREDECESSOR и SUCCESSOR в худшем случае равна $O(n)$.

В [1] показано, что математическое ожидание высоты h случайно построенного бинарного дерева поиска с n ключами равно $O(\log n)$. Это означает, что в среднем случае вычислительная сложность операций бинарного дерева поиска равна $O(\log n)$.

В табл. 10.1 приведены вычислительные сложности основных операций бинарного дерева поиска.

В приведенных выше оценках вычислительной сложности операций предполагается, что операция сравнения ключей выполняется за время $O(1)$. Если же ключ имеет составной тип данных, то вычислительная сложность операции сравнения зависит от длин сравниваемых ключей. Например, если ключом является строка длины t , то вычислительная сложность операций INSERT, LOOKUP, DELETE, PREDECESSOR и SUCCESSOR в среднем случае будет $O(t \log n)$ и $O(tn)$ в худшем случае. Операции MIN и MAX не требуют выполнения операции сравнения, поэтому их вычислительная сложность не зависит от длины ключа.

10.11. Упражнения

1. Нарисуйте бинарное дерево поиска минимальной высоты, в котором присутствует пять узлов с ключами 10, 20, 30, 40, 50.
2. Предложите рекурсивную версию операции добавления узла в бинарное дерево поиска.
3. Модифицируйте код операции INSERT для корректной обработки случая, когда переданное значение $tree = null$ (вставка в пустое дерево).
4. Какое значение вернут функции SUCCESSOR и PREDECESSOR, если в них передать соответственно наибольший и наименьший ключи дерева?
5. Разработайте процедуру перебора всех узлов бинарного дерева поиска в порядке убывания ключей.
6. Предложите алгоритм удаления узла из бинарного дерева поиска, если каждая вершина дополнительно содержит указатель $node.parent$ на родительский узел.
7. Модифицируйте процедуру удаления узла из бинарного дерева поиска так, чтобы в качестве замены для узла $node$ выбиралась предшествующая ему вершина (predecessor).
8. Разработайте рекурсивную версию функции поиска узла.
9. Предложите алгоритм вычисления высоты бинарного дерева поиска, если известен указатель на его корень.

11. Красно-черные деревья

Трудоемкость основных операций над бинарным деревом поиска линейно зависит от его высоты. Как известно из раздела 10.10, в среднем случае высота бинарного дерева поиска составляет $O(\log n)$. Однако в худшем случае элементы могут добавляться в упорядоченной последовательности, что приводит к вырождению дерева в цепочку, аналогичную связному списку длины n , и основные операции над бинарным деревом поиска выполняются за время $O(n)$.

Такого недостатка лишены *сбалансированные деревья поиска* (self-balancing search trees), которые динамически корректируют свою структуру для обеспечения высоты не более $O(\log n)$.

Существует множество сбалансированных деревьев поиска, например:

- АВЛ-дерево (AVL tree);
- красно-черное дерево (red-black tree);
- В-дерево (B-tree);
- АА-дерево (AA tree).

Первой известной реализацией сбалансированного дерева поиска является *АВЛ-дерево*, описанное в 1962 г. советскими учеными Г.М. Адельсон-Вельским и Е.М. Ландисом [6]. В АВЛ-дереве у любой вершины высота левого и правого поддеревьев различаются не более, чем на 1.

Красно-черным деревом называется бинарное дерево поиска, структура узла которого содержит дополнительное поле, *цвет* (color), который может быть либо *красным* (red), либо *черным* (black). Эта структура данных была представлена в 1972 г. Рудольфом Байером (Rudolf Bayer) под названием «симметричное бинарное В-дерево» (symmetric binary B-tree) [7]. Современное название красно-черного дерева было дано в работе Роберта Седжвика в 1978 г. [8].

В-дерево (B-дерево) – это сбалансированное дерево поиска, узлы которого хранятся во внешней памяти. В-дерево было разработано в 1972 г. Рудольфом Байером и Эдвардом Маккрейтом (Edward M. McCreight) [9]. Размер В-дерева может значительно превышать доступный объем оперативной памяти, поэтому в любой момент времени в памяти находится лишь часть дерева. В-дерево и его вариации (B^+ -дерево, B^* -дерево) применяются в файловых системах и системах управления базами данных (СУБД).

АА-дерево, названное по имени своего автора, Арне Андерссона (Arne Andersson) [10], является разновидностью красно-черного дерева. Вместо

цвета в этом дереве используется понятие *уровня* (level) вершины, а операция балансировки значительно упрощена по сравнению с красно-черным деревом.

Из описанных выше реализаций сбалансированных деревьев поиска наиболее широко на практике применяется красно-черное дерево.

11.1. Структура узла красно-черного дерева

Красно-черное дерево (red-black tree) – это бинарное дерево поиска, для которого выполняются красно-черные свойства (red-black properties):

- 1) Каждый узел дерева является либо красным, либо черным.
- 2) Корневой узел дерева является черным.
- 3) Каждый листовой узел дерева является черным.
- 4) Оба дочерних узла красной вершины являются черными.
- 5) У любого узла все пути от него до листовых узлов, являющихся его потомками, содержат одинаковое количество черных узлов.

Черной высотой $bh(x)$ узла x (black height) называется количество черных узлов на пути от вершины x (но не считая ее) до листового узла дерева. Черная высота красно-черного дерева равна черной высоте его корневого узла.

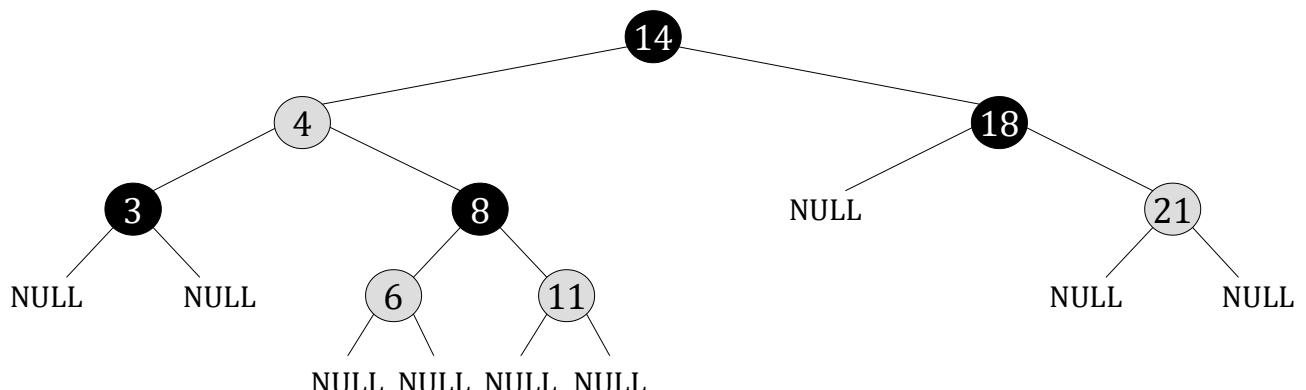


Рис. 11.1. Красно-черное дерево из восьми узлов.
Красные узлы закрашены серым цветом.

На рис. 11.1 показан пример красно-черного дерева, в узлах которого записаны значения ключей. Черная высота приведенного дерева равна 2: каждый путь от корневого узла с ключом 14 содержит один внутренний черный узел и один листовой узел, который по свойству 3 также является черным.

В целях сокращения используемой памяти все листовые узлы (*NULL*) красно-черного дерева заменяются указателями на один *ограничивающий узел* (sentinel node) черного цвета.

11.2. Высота красно-черного дерева

Как и в обычном бинарном дереве поиска, вычислительная сложность операций красно-черного дерева линейно зависит от его высоты.

Лемма 1. *Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2 \log(n + 1)$.*

Доказательство. Покажем по индукции, что любое поддерево с вершиной в узле x содержит не менее $2^{bh(x)} - 1$ внутренних узлов. Если высота h узла x равна 0, то узел x является листовым (*NULL*), а его поддерево содержит не менее $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов.

В качестве шага индукции рассмотрим узел x , который имеет положительную высоту $h(x)$ и является внутренним узлом с двумя дочерними узлами. Каждый дочерний узел имеет черную высоту либо $bh(x)$, либо $bh(x) - 1$, в зависимости от его цвета, соответственно красного или черного. Поскольку высота дочернего узла x меньше высоты узла x , мы можем использовать индуктивное предположение и сделать вывод о том, что каждый потомок x имеет как минимум $2^{bh(x)-1} - 1$ внутренних узлов. Тогда все дерево с корнем в узле x содержит не менее

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

внутренних узлов.

Пусть h – высота дерева. По свойству 4 красно-черного дерева как минимум половина узлов на пути от корня к листу – черные, следовательно, черная высота корневого узла будет не меньше, чем $h/2$, тогда

$$n \geq 2^{h/2} - 1.$$

Путем логарифмирования и элементарных преобразований в неравенстве получаем

$$\log(n + 1) \geq h/2,$$

$$h \leq 2 \log(n + 1).$$

□

В бинарном дереве поиска вычислительная сложность операций *INSERT*, *LOOKUP*, *DELETE*, *MIN*, *Max*, *PREDECESSOR* и *SUCCESSOR* линейно зависит от его высоты h . Таким образом, вычислительная сложность операций красно-черного дерева в худшем случае равна $O(\log n)$.

В табл. 11.1 приведены вычислительные сложности основных операций красно-черного дерева.

Таблица 11.1. Вычислительная сложность операций красно-черного дерева

Операция	Средний случай	Худший случай
$\text{INSERT}(tree, key, value)$	$O(\log n)$	$O(\log n)$
$\text{LOOKUP}(tree, key)$	$O(\log n)$	$O(\log n)$
$\text{DELETE}(tree, key)$	$O(\log n)$	$O(\log n)$
$\text{MIN}(tree)$	$O(\log n)$	$O(\log n)$
$\text{MAX}(tree)$	$O(\log n)$	$O(\log n)$
$\text{PREDECESSOR}(tree, key)$	$O(\log n)$	$O(\log n)$
$\text{SUCCESSOR}(tree, key)$	$O(\log n)$	$O(\log n)$
$\text{DELETETREE}(tree)$	$\Theta(n)$	$\Theta(n)$

11.3. Добавление узла

Для добавления узла в красно-черное дерево сперва отыскивается элемент, который становится родителем для новой вершины, аналогично функции INSERT в обычном бинарном дереве поиска. Новый узел окрашивается в красный цвет и становится дочерним по отношению к найденному элементу. Затем выполняется восстановление красно-черных свойств дерева.

Зададимся вопросом: какие свойства красно-черного дерева могут быть нарушены при добавлении в него нового элемента красного цвета? Свойство 1 выполняется: все узлы в дереве по-прежнему будут иметь либо красный, либо черный цвет. Также не могут нарушиться свойства 3 и 5: новый красный элемент замещает черный листовой узел, но сам имеет два черных дочерних листа, поэтому его черная высота не изменяется.

После вставки в дерево нового узла красного цвета могут быть нарушены красно-черные свойства 2 и 4. Например, при добавлении первого элемента корень дерева окажется красным, что противоречит свойству 2. Другой пример: если родительский узел нового элемента имеет красный цвет, это нарушает свойство 4: у красной вершины оба дочерних узла должны являться черными. Таким образом, после добавления нового элемента необходимо выполнить операцию восстановления свойств красно-черного дерева.

Восстановление свойств красно-черного дерева после добавления элемента выполняется функцией RBTreeInsertFixup . Обход узлов начинаем с последнего добавленного элемента, далее продвигаемся вверх к корневому узлу дерева. Возможны 6 случаев, нарушающих красно-черные свойства, 3 из них симметричны другим трем.

Обозначим через z текущий узел, p – его родительский узел $z.parent$, g – родительский узел p , u – «дядя» узла z , родственный узел p .

```

1 function RBTREEINSERT(T, key, value)
2     tree = T.root
3     while tree ≠ null do
4         if key < tree.key then
5             tree = tree.left
6         else if key > tree.key then
7             tree = tree.right
8         else
9             return                  /* Ключ уже присутствует в дереве */
10        end if
11    end while
12    node = RBTREECREATENODE(key, value)
13    if T.root = null then           /* Пустое дерево */
14        T.root = node
15    else
16        if key < tree.parent.key then
17            tree.parent.left = node
18        else
19            tree.parent.right = node
20        end if
21    end if
22    node.color = RED
23    /* Восстановление красно-черных свойств */
24    RBTREEINSERTFIXUP(T, node)
25 end function

```

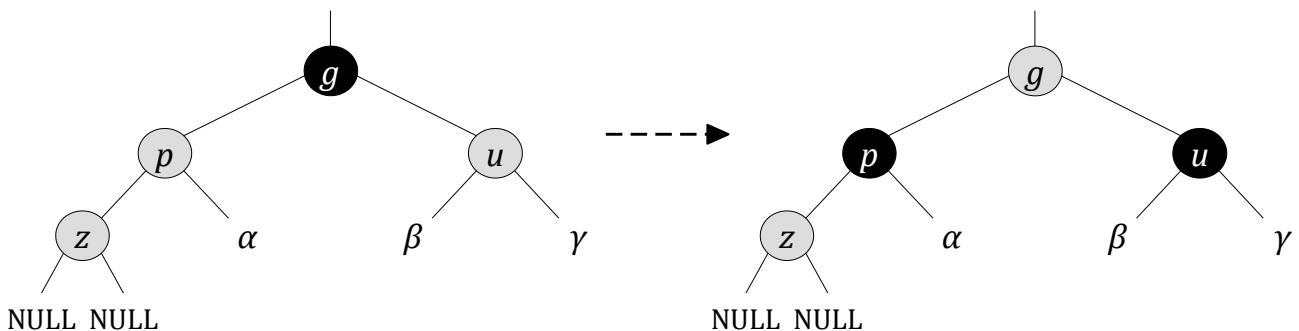


Рис. 11.2. Восстановление красно-черных свойств
после добавления узла, случай 1.

Случай 1. Вершина *z* имеет родительский узел *p*, также красного цвета. Узел *p* является корнем левого поддерева своего черного родителя *g* и имеет красный родственный узел *u* – «дядю» узла *z*. В узле *p* нарушается свойство 4: красный узел не может иметь красных потомков.

Этот случай разрешается путем перекрашивания узлов следующим способом: узлы *p* и *u* перекрашиваются в черный цвет, а узел *g* – в крас-

ный. Так как красный узел g может иметь красного родителя, делаем его текущим узлом и продвигаемся вверх по дереву (рис. 11.2).

Случай 2. Имеем красный узел z , который является правым потомком своего родительского узла p , также красного цвета. Вершина p в свою очередь является корнем левого поддерева своего родителя g . «Дядя» узла z – черный. В узле p нарушаются свойство 4.

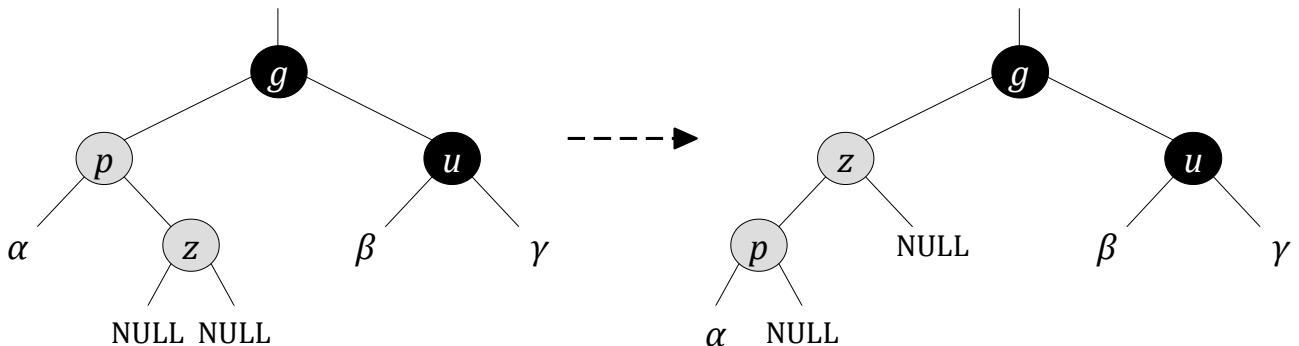


Рис. 11.3. Восстановление красно-черных свойств после добавления узла, случай 2.

Такой случай разрешается путем перехода к случаю 3. Для этого необходимо выполнить поворот дерева с корнем в узле p влево (рис. 11.3).

Для восстановления красно-черных свойств применяются вспомогательные функции LEFTROTATE и RIGHTROTATE, выполняющие соответственно левый и правый *повороты* (rotation) дерева. Функции поворотов преобразуют указатели, сохраняя при этом свойства бинарного дерева поиска.

На рис. 11.4 показаны процессы левого и правого поворотов бинарного дерева поиска.

```

1 function LEFTROTATE(x)
2     y = x.right
3     x.right = y.left
4     if y.left ≠ null then
5         y.left.parent = x
6     end if
7     y.parent = x.parent
8     if x = x.parent.left then
9         x.parent.left = y
10    else
11        x.parent.right = y
12    end if
13    y.left = x
14    x.parent = y
15 end function

```

```

1 function RIGHTROTATE( $x$ )
2      $y = x.left$ 
3      $x.left = y.right$ 
4     if  $y.right \neq null$  then
5          $y.right.parent = x$ 
6     end if
7      $y.parent = x.parent$ 
8     if  $x = x.parent.left$  then
9          $x.parent.left = y$ 
10    else
11         $x.parent.right = y$ 
12    end if
13     $y.right = x$ 
14     $x.parent = y$ 
15 end function

```

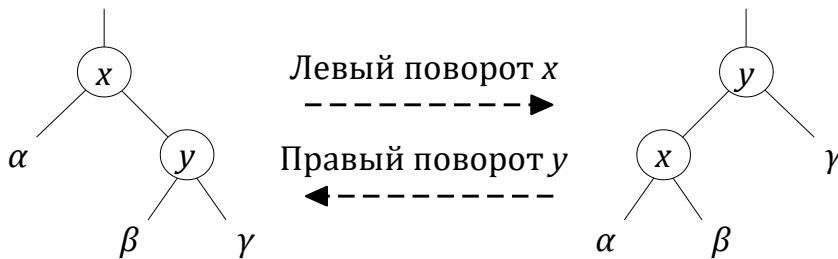


Рис. 11.4. Повороты бинарного дерева поиска.

Левый поворот может быть выполнен для любого бинарного дерева с корнем в узле x , правый потомок y которого не является листовым узлом. Выполняется поворот ребра, связывающего узлы x и y , в результате чего вершина y становится корнем дерева, а узел x – ее правым потомком.

Правый поворот дерева является симметричным левому, число операций при каждом повороте имеет порядок $O(1)$ – в функциях производится только корректировка фиксированного числа указателей.

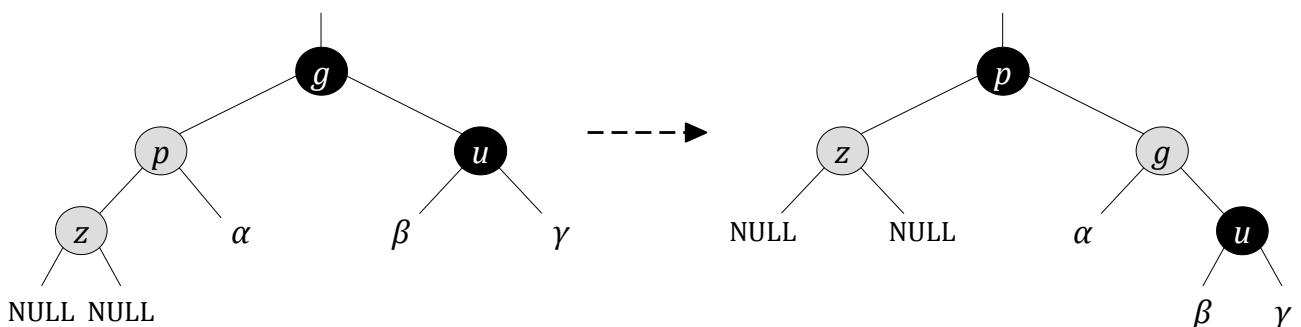


Рис. 11.5. Восстановление красно-черных свойств
после добавления элемента, случай 3.

Случай 3. Этот случай аналогичен случаю 2, с одним отличием: текущий красный узел z является левым потомком своего родителя r . Узел r по-прежнему нарушает свойство 4, так как имеет красного потомка z .

Разрешение случая 3 проходит в два этапа. Сперва необходимо выполнить перекрашивание вершин: узел r становится черным, его родитель g – красным. Затем выполняется правый поворот дерева с корнем в узле g (рис. 11.5).

Случаи 4–6, как уже было отмечено, симметричны случаям 1–3. В них узел r является корнем правого поддерева своего родительского узла g . Разрешение этих случаев выполняется аналогично разрешению трех случаев, описанных выше.

На рис. 11.6 показан пример добавления в красно-черное дерево элемента 5 с последующим восстановлением структуры дерева в три шага (случай 1, случай 2, случай 3).

```

1 function RBTREEINSERTFIXUP( $T, z$ )
2   while  $z.parent.color = RED$  do
3     if  $z.parent = z.parent.parent.left$  then
4       /* Узел  $z$  находится в левом поддереве  $g$  */
5        $u = z.parent.parent.right$ 
6       if  $u.color = RED$  then                                /* Случай 1 */
7          $z.parent.color = BLACK$ 
8          $u.color = BLACK$ 
9          $z.parent.parent.color = RED$ 
10         $z = z.parent.parent$ 
11      else
12        if  $z = z.parent.right$  then                      /* Случай 2 */
13           $z = z.parent$ 
14          RBTREERotateLEFT( $T, z$ )
15        end if
16        /* Случай 3 */
17         $z.parent.color = BLACK$ 
18         $z.parent.parent.color = RED$ 
19        RBTREERotateRIGHT( $T, z.parent.parent$ )
20      end if
21    else
22      /* Случаи 4–6:  $z$  находится в правом поддереве  $g$  */
23      /* ... */
24    end if
25  end while
26   $T.root.color = BLACK$ 
27 end function
```

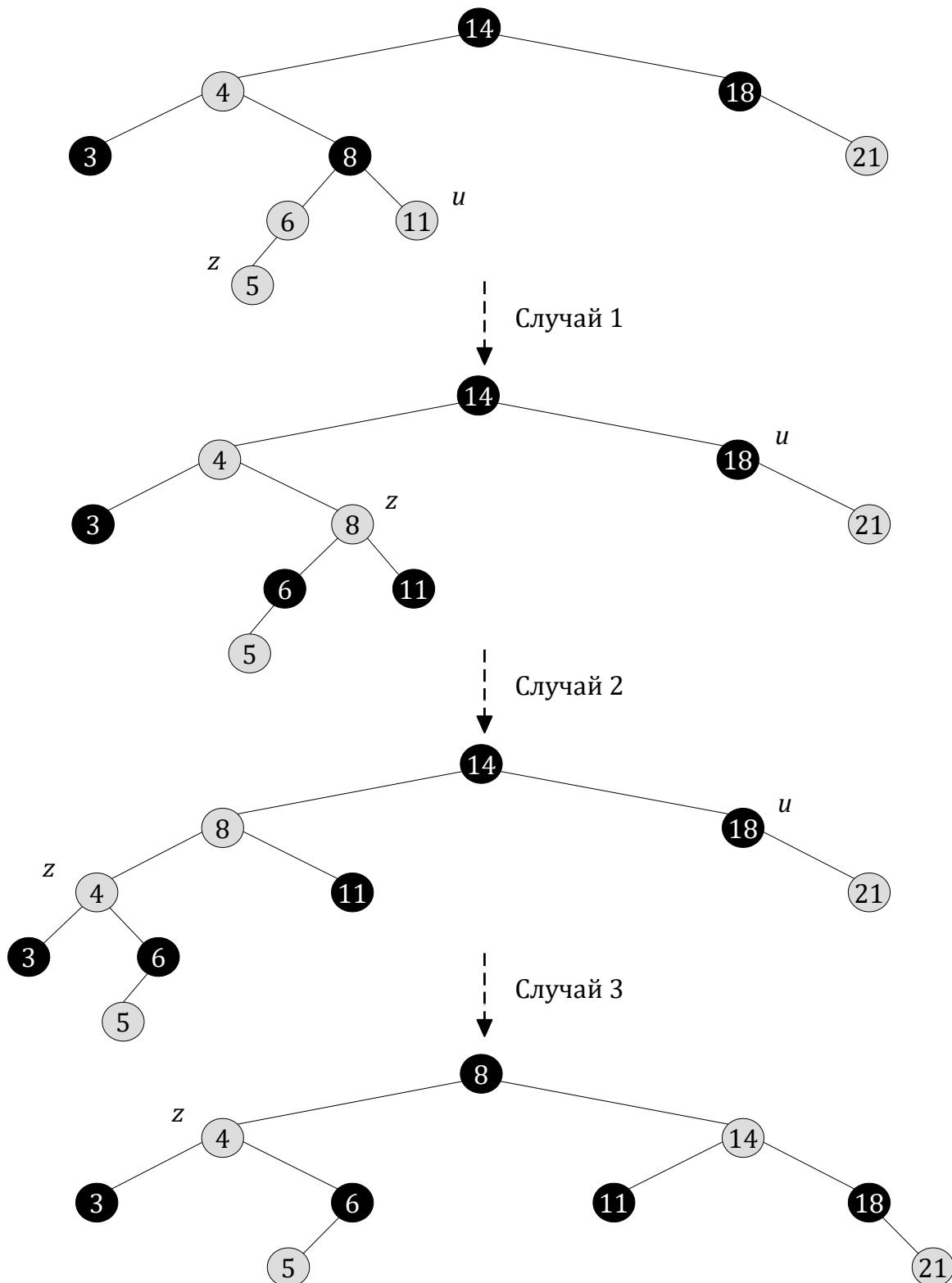


Рис. 11.6. Восстановление свойств красно-черного дерева после добавления ключа 5.

Отметим, что внешний цикл в операции RBTreeInsertFixup выполняется повторно только в случае 1, после чего указатель *z* перемещается вверх по дереву на два уровня. В случаях 2 и 3 цикл завершает работу, поэтому при восстановлении красно-черных свойств никогда не выполняется

больше двух поворотов. Количество итераций в худшем случае – $O(\log n)$.

11.4. Удаление узла

Для удаления элемента из красно-черного дерева необходимо отыскать его по заданному ключу и удалить, как в случае обычного дерева поиска. Далее, в случае удаления черного узла, путем перекрашивания узлов и выполнения поворотов восстанавливаются красно-черные свойства.

```

1 function RBTREEDELETE( $T, key$ )
2    $z = \text{RBTREELOOKUP}(T, key)$ 
3    $y = z$ 
4    $y.color = y.color$ 
5   if  $z.left = null$  then           /*  $z$  не имеет левого поддерева */
6      $x = z.right$ 
7     RBTREETRANSPLANT( $T, z, z.right$ )
8   else if  $z.right = null$  then    /*  $z$  не имеет правого поддерева */
9      $x = z.left$ 
10    RBTREETRANSPLANT( $T, z, z.left$ )
11  else                                /*  $z$  имеет оба поддерева */
12     $y = \text{RBTREEMIN}(z.right)$ 
13     $y.color = y.color$ 
14     $x = y.right$ 
15    if  $y.parent = z$  then
16       $x.parent = y$ 
17    else
18      RBTREETRANSPLANT( $T, y, y.right$ )
19       $y.right = z.right$ 
20       $y.right.parent = y$ 
21    end if
22    RBTREETRANSPLANT( $T, z, y$ )
23     $y.left = z.left$ 
24     $y.left.parent = y$ 
25     $y.color = z.color$ 
26  end if
27  if  $y.color = BLACK$  then
28    /* Восстановление красно-черных свойств */
29    RBTREEDELETEFIXUP( $T, x$ )
30  end if
31 end function
```

Операция RBTREEMIN поиска минимального элемента дерева выполняется точно так же, как и в обычном бинарном дереве поиска. Вспомо-

гательная функция RBTREETRANSPLANT заменяет узел u вершиной v , ее псевдокод приведен ниже.

```

1 function RBTREETRANSPLANT( $T, u, v$ )
2   if  $u.parent = null$  then
3      $T.root = v$ 
4   else if  $u = u.parent.left$  then
5      $u.parent.left = v$ 
6   else
7      $u.parent.right = v$ 
8   end if
9    $v.parent = u.parent$ 
10 end function
```

При удалении узла из красно-черного дерева могут быть нарушены свойства 2, 4 и 5. Свойство 2 нарушается, если мы удаляем черный корень дерева и заменяем его красным потомком. В случае, когда и узел x , который заменяет удаленный из дерева элемент, и его родитель $x.parent$ имеют красный цвет, нарушается свойство 4. Перемещение черного узла u в дереве приводит к тому, что путь, содержащий u ранее, теперь имеет на один черный узел меньше, что нарушает свойство 5.

Нарушения этих свойств возможно избежать путем присваивания вершине x дополнительного черного цвета. Таким образом, красный узел x становится черным, что исключает возможность нарушения свойств 2 и 4. Если же узел x имеет черный цвет, он становится «дважды черным» (doubly black). Это позволяет сохранить свойство 5, но приводит к нарушению свойства 1: вершина красно-черного дерева не может иметь более одного цвета.

Функция RBTREEDELETEFIXUP применяется для восстановления свойства 1 в дважды черном узле x . Это происходит либо при его перекрашивании в красный цвет (тогда x становится просто черным), либо при перемещении x в корень дерева (в этом случае дополнительный черный цвет может быть удален). Очевидно, что преобразования дерева не должны нарушать остальные красно-черные свойства.

В операции RBTREEDELETEFIXUP рассматриваются 8 случаев, 4 из которых симметричны другим четырем. Переменной w обозначим родственный узел вершины x .

Случай 1. Дважды черная вершина x является левым потомком своего родителя $x.parent$ черного цвета и имеет красный родственный узел w . Такой случай разрешается путем перехода к одному из случаев 2–4. Для этого необходимо сделать узел w черным, а $x.parent$ – красным, после чего выполняется левый поворот дерева с корнем в узле $x.parent$ (рис. 11.7, а).

```

1 function RBTREEDELETEFIXUP( $T, x$ )
2   while  $x \neq T.\text{root}$  and  $x.\text{color} = \text{BLACK}$  do
3     if  $x = x.\text{parent}.left$  then      /*  $x$  – левый потомок  $x.\text{parent}$  */
4        $w = x.\text{parent}.right$ 
5       if  $w.\text{color} = \text{RED}$  then                      /* Случай 1 */
6          $w.\text{color} = \text{BLACK}$ 
7          $x.\text{parent}.\text{color} = \text{RED}$ 
8         RBTREERotateLeft( $T, x.\text{parent}$ )
9          $w = x.\text{parent}.right$ 
10        end if
11        if  $w.\text{left}.\text{color} = \text{BLACK}$  and  $w.\text{right}.\text{color} = \text{BLACK}$  then
12          /* Случай 2 */
13           $w.\text{color} = \text{RED}$ 
14           $x = x.\text{parent}$ 
15        else
16          if  $w.\text{right}.\text{color} = \text{BLACK}$  then                  /* Случай 3 */
17             $w.\text{left}.\text{color} = \text{BLACK}$ 
18             $w.\text{color} = \text{RED}$ 
19            RBTREERotateRight( $T, w$ )
20             $w = x.\text{parent}.right$ 
21          end if                                         /* Случай 4 */
22           $w.\text{color} = x.\text{parent}.\text{color}$ 
23           $x.\text{parent}.\text{color} = \text{BLACK}$ 
24           $w.\text{right}.\text{color} = \text{BLACK}$ 
25          RBTREERotateLeft( $T, x.\text{parent}$ )
26           $x = T.\text{root}$ 
27        end if
28      else
29        /* Случай 5–8:  $x$  – правый потомок  $x.\text{parent}$  */
30        /* ... */
31      end if
32    end while
33     $x.\text{color} = \text{BLACK}$ 
34 end function

```

Случай 2. Узел w имеет черный цвет, оба его потомка также черные. В этом случае узел w становится красным, а дополнительный черный цвет узла x перемещается на $x.\text{parent}$, родителя x и w (рис. 11.7, б).

Если узел $x.\text{parent}$ изначально был черным, цикл переходит к следующей итерации с перемещением указателя x на уровень выше, в противном случае выполнение функции завершается: для восстановления свойства 1 остается сделать $x.\text{parent}$ просто черным.

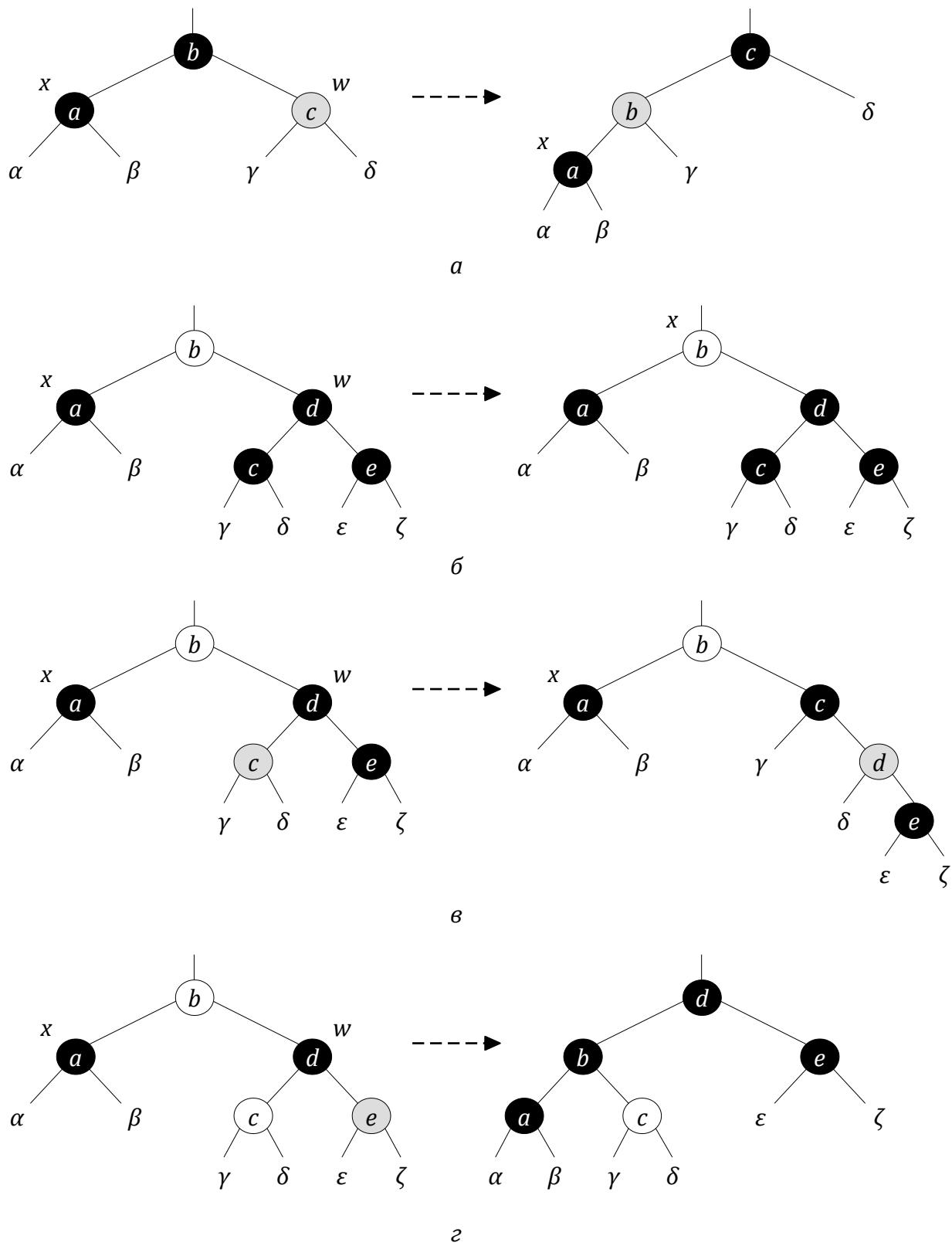


Рис. 11.7. Восстановление красно-черных свойств после удаления элемента: a – случай 1; $б$ – случай 2; $в$ – случай 3; $г$ – случай 4. Узел x имеет дополнительный черный цвет. Незакрашенные узлы могут быть как красного, так и черного цвета.

Случай 3. Узел w имеет черный цвет, его левый потомок – красный, а правый – черный. Обмениваем цвета w и его левого потомка, после чего

выполняем правый поворот дерева с корнем в вершине w (рис. 11.7, в). Таким образом, новым родственным узлом вершины x становится черный узел с красным правым потомком, что сводит случай 3 к случаю 4.

Случай 4. Узел w имеет черный цвет, его правый потомок – красный. Перекрашивание родителя и правого потомка w в черный цвет с последующим левым поворотом дерева $x.parent$ делает возможным удаление дополнительного черного цвета в узле x без нарушения красно-черных свойств (рис. 11.7, г). Указатель x перемещается в корень дерева, цикл завершается.

Следует отметить, что возникновение случаев 1, 3 и 4 приводит к завершению внешнего цикла функции, при этом выполняется не более трех поворотов дерева. Случай 2 может привести к следующей итерации цикла, но в нем выполняется только перемещение указателя x вверх по дереву. Таким образом, количество итераций внешнего цикла RBTREEDELETEFIXUP составляет $O(\log n)$.

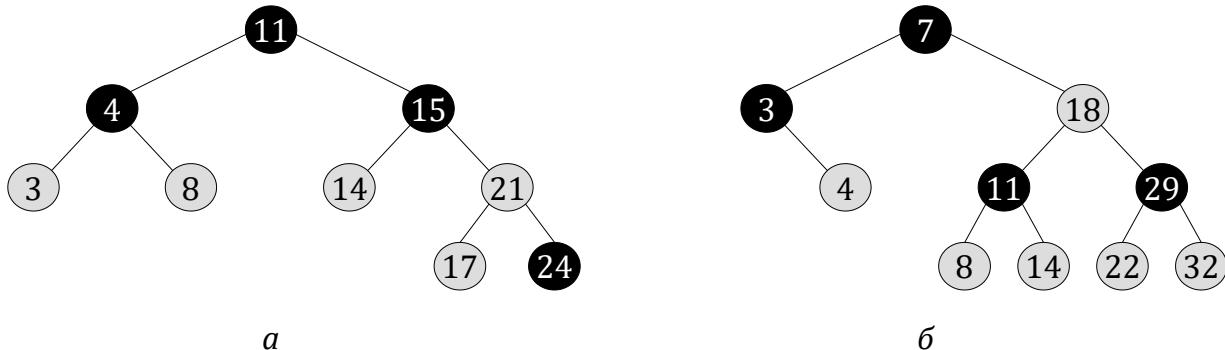


Рис. 11.8. Иллюстрация к заданию 1.

11.5. Упражнения

- Являются ли деревья на рис. 11.8 (а, б) красно-черными? Если нет, какие свойства нарушены?
- Нарисуйте для набора ключей 12, 15, 24, 32, 17, 19, 21 красно-черное дерево с пошаговым добавлением узлов в указанной последовательности и балансировкой структуры дерева.
- Найдите обычную и черную высоту красно-черного дерева, полученного в задании 2.
- Добавьте в красно-черное дерево, полученное в задании 2, ключ 22.
- Допишите псевдокод функции RBTREEINSERTFIXUP восстановления красно-черных свойств для случаев 4–6.
- Объясните, почему после удаления узла красного цвета не требуется вызов функции RBTREEDELETEFIXUP.
- Подумайте, не могут ли нарушить операции, выполняемые в функции RBTREEDELETEFIXUP, свойство дерева о черной высоте?

12. Префиксные деревья

При анализе вычислительной сложности операций рассмотренных ранее реализаций ассоциативных массивов (бинарных деревьев поиска, красно-черных деревьев) мы полагали, что время выполнения операции сравнения двух ключей составляет $O(1)$. Если же ключи представлены строковым типом данных, время их сравнения линейно зависит от длины заданных строк, и его следует учитывать при анализе трудоемкости операций словаря.

Одной из наиболее широко известных реализаций словаря со строковыми ключами является *префиксное дерево* (prefix tree, trie). Оригинальное название этой структуры данных, *trie*, происходит от английского слова *retrieval* (поиск, извлечение, выборка). В различных источниках также встречаются названия «*бор*», «*луч*» и «*нагруженное дерево*».

Префиксные деревья нашли широкое применение в алгоритмах предиктивного ввода текста, автозавершения (autocomplete) и проверки правописания (spellcheck) в различных текстовых редакторах, а также при поиске по IP-адресам в кэширующих прокси-серверах (Squid Caching Proxy).

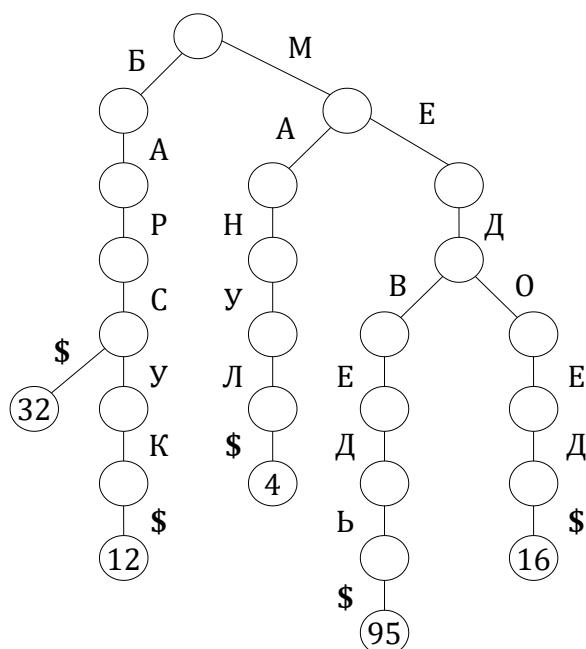


Рис. 12.1. Префиксное дерево из 5 элементов.

12.1. Структура префиксного дерева

Префиксное дерево содержит n пар строковых ключей и ассоциированных с ними значений. Ключом в префиксном дереве является набор символов (c_1, c_2, \dots, c_m) из заранее определенного алфавита $A = \{a_1, a_2, \dots, a_d\}$. Здесь и далее m – длина ключа, d – размер (мощность) алфавита.

Каждый внутренний узел префиксного дерева может содержать от 1 до d дочерних узлов. Ключи не хранятся в узлах деревьев: символы алфавита закреплены за ребрами дерева. В листовых узлах дерева содержатся значения, ассоциированные с ключами словаря.

На рис. 12.1 показан пример префиксного дерева, содержащего 5 пар «ключ – значение»:

(Барс, 32), (Барсук, 12), (Манул, 4), (Медведь, 95), (Медоед, 16).

Символом «\$» обозначается маркер конца строки.

Высота префиксного дерева определяется размером самой длинной строки, хранящейся в нем.

12.2. Вставка элемента

Вставка элемента в префиксное дерево производится посимвольно, начиная с корня структуры.

```

1 function TRIEINSERT(tree, key[1..l], value)
2   node = root
3   for i = 1 to l do
4     child = GETCHILD(node, key[i])
5     if child = null then
6       child = TRIECREATENODE()
7       SETCHILD(node, key[i], child)
8     end if
9   end for
10  node.value = value
11 end function

```

На итерации i в текущем узле отыскивается указатель на дочерний узел, соответствующий символу c_i ключа. Найденный дочерний узел делается текущим, после чего описанные действия выполняются для следующего символа ключа. Если искомый указатель отсутствует в узле, создается новый дочерний узел и процедура продолжается аналогично.

При достижении конца строки (\$) значение вставляется в текущий

узел. Ключи, содержащиеся в префиксном дереве, как и в других реализациях ассоциативного массива, должны быть уникальными.

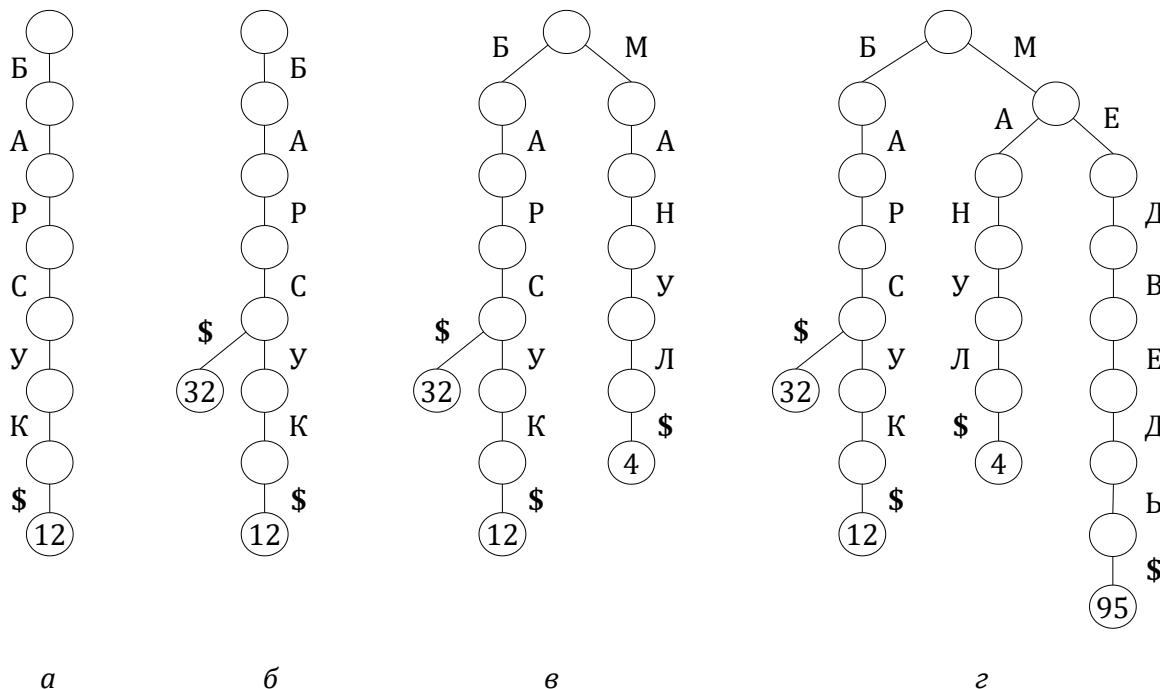


Рис. 12.2. Добавление в префиксное дерево 4 элементов:

- a* – добавлен элемент (Барсук, 12); *б* – добавлен элемент (Барс, 32);
в – добавлен элемент (Манул, 4); *г* – добавлен элемент (Медведь, 95).

Ниже приведен псевдокод функции `TRIEINSERT` добавления элемента в префиксное дерево. Функция `GETCHILD` возвращает указатель на дочерний узел *child*, соответствующий символу *c_i* добавляемого ключа. Функция `SETCHILD` устанавливает указатель символа *c_i* на новый дочерний узел *child*.

На рис. 12.2 показан пример добавления в префиксное дерево четырех пар «ключ – значение».

12.3. Поиск элемента

Для поиска элемента в префиксном дереве необходимо в цикле просмотреть все символы заданного ключа. Для каждого из них в текущем узле, начиная с корневого, отыскивается соответствующий дочерний узел, который делается текущим для следующей итерации. Если на каком-либо шаге искомый дочерний узел отсутствует, поиск прекращается: ключ отсутствует в дереве.

После достижения конца строки необходимо проверить, что с ключом в данном дереве действительно ассоциировано значение – найденный ключ может оказаться префиксом другого ключа.

```

1 function TRIELOOKUP(tree, key[1..l])
2     node = root
3     for i = 1 to l do
4         child = GETCHILD(node, key[i])
5         if child = null then
6             return null
7         end if
8         node = child
9     end for
10    if node.value = null then
11        return null
12    end if
13    return node
14 end function

```

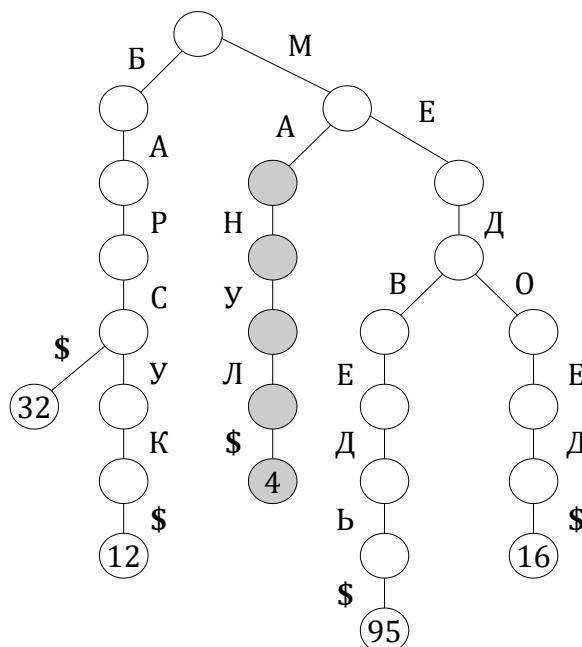


Рис. 12.3. Удаление элемента (Манул, 4) из префиксного дерева. Серым цветом закрашены узлы, которые будут удалены из памяти.

12.4. Удаление элемента

Первым шагом операции удаления элемента из префиксного дерева является поиск листового узла, содержащего искомый ключ. Далее производится подъем по дереву: если текущий узел не имеет дочерних узлов, он удаляется из памяти, после чего текущим делается его родительский узел. В противном случае подъем по дереву заканчивается.

На рис. 12.3 показана операция удаления из префиксного дерева элемента с ключом «Манул».

12.5. Узел префиксного дерева

Как было отмечено выше, каждый внутренний узел префиксного дерева может содержать от 1 до d указателей на дочерние узлы. Возникает вопрос: как хранить эти указатели?

От способа организации внутренних узлов префиксного дерева зависит сложность операций `GETCHILD` и `SETCHILD`, а следовательно, и всех основных операций дерева. Рассмотрим некоторые возможные варианты.

Массив указателей. Индексом в таком массиве является порядковый номер символа в алфавите. Сложность операций `GETCHILD/SETCHILD` в этом случае будет составлять $O(1)$, но такой подход может привести к нерациональному потреблению памяти: даже для узлов с одним потомком будет создаваться массив указателей на дочерние узлы размерности d .

Связный список. Указатели на дочерние узлы можно объединять в односвязный список. Новые узлы будут создаваться по мере необходимости, а сложность операций `GETCHILD/SETCHILD` составит $O(d)$.

Префиксное дерево, в узлах которого указатели на дочерние элементы хранятся в связном списке, является *неупорядоченным деревом поиска* (*unordered search tree*).

Сбалансированное дерево поиска. Использование для хранения указателей одного из сбалансированных деревьев поиска (красно-черное дерево, АВЛ-дерево и др.) позволяет сократить сложность выполнения операций `GETCHILD/SETCHILD` по сравнению с подходом на основе связного списка: трудоемкость добавления и поиска нужных указателей в этом случае составит $O(\log d)$.

12.6. Представление дерева на основе связных списков

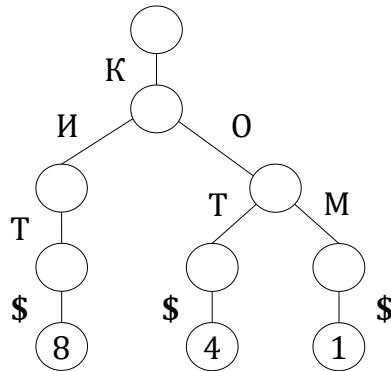


Рис. 12.4. Префиксное дерево из 3 элементов.

Для хранения префиксного дерева, указатели на дочерние узлы которого организованы в связный список, используется представление «левый

дочерний и правый сестринский» (раздел 9.3.3).

На рис. 12.5 показано представление «левый дочерний и правый сестринский» неупорядоченного префиксного дерева из 3 элементов с рис. 12.4. Поле *child* элемента c_i указывает на голову списка (c_1, c_2, \dots, c_d) его дочернего узла, а поле *sibling* – на следующий элемент списка указателей в текущем узле.

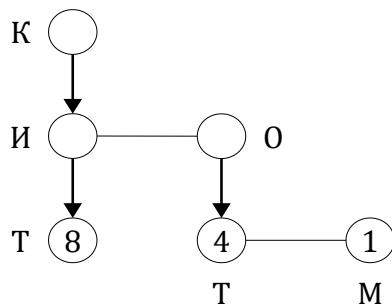


Рис. 12.5. Реализация префиксного дерева в виде «левый дочерний и правый сестринский» (стрелками показаны указатели *child*, горизонтальными ребрами – указатели *sibling*).

12.7. Трудоемкость операций префиксного дерева

Вычислительная сложность основных операций префиксного дерева (добавление, поиск, удаление) не зависит от количества n элементов в словаре: на время выполнения операций влияют только длина m ключа и мощность d используемого алфавита.

Таблица 12.1. Вычислительная сложность операций префиксного дерева

Операция	Способ работы с указателями на дочерние узлы		
	Связный список	Массив	Сбалансированное дерево поиска
TRIEINSERT($tree, key, value$)	$O(md)$	$O(m)$	$O(m \log d)$
TRIELOOKUP($tree, key$)	$O(md)$	$O(m)$	$O(m \log d)$
TRIEDELETE($tree$)	$O(md)$	$O(m)$	$O(m \log d)$
TRIEMIN($tree$)	$O(hd)$	$O(hd)$	$O(h \log d)$
TRIEMAX($tree$)	$O(hd)$	$O(hd)$	$O(h \log d)$

В отличие от хеш-таблиц, в префиксном дереве не возникают коллизии, а также возможен обход элементов в упорядоченной последовательности – реализация *упорядоченного множества* (*ordered set*), в зависимости от способа реализации функций *GETCHILD* и *SETCHILD*.

Ключи префиксного дерева не хранятся в узлах, поэтому для их хранения не используется дополнительная память.

В табл. 12.1 приведена вычислительная сложность основных операций префиксного дерева. Через h обозначена высота дерева – количество символов в самом длинном ключе. В случае, если префиксное дерево упорядочено, операции поиска экстремальных элементов (TRIEMIN/TRIEMAX) реализуются за время $O(h)$.

12.8. Упражнения

1. Сравните трудоемкость основных операций префиксного дерева и аналогичных операций красно-черного дерева, ключами которого являются строки.
2. Оцените сложность по памяти реализации префиксного дерева, указатели на дочерние узлы которого хранятся в массиве.
3. Изобразите дерево с рис. 12.1 в представлении «левый дочерний и правый сестринский». Является ли это дерево упорядоченным?
4. Реализуйте функцию SETCHILD префиксного дерева, указатели на дочерние узлы которого хранятся в связном списке. Почему ее трудоемкость составляет $O(d)$?
5. Разработайте алгоритмы поиска минимального (TRIEMIN) и максимального (TRIEMAX) ключа в префиксном дереве.
6. Реализуйте алгоритм обхода префиксного дерева в упорядоченной последовательности (TRIETRAVERSEINORDER).
7. Запишите псевдокод функции TRIEDELETE.

13. Хеш-таблицы

Хеш-таблицы (hash table) относятся к динамическим структурам данных, предназначенным для реализации АТД *неупорядоченное множество* (unordered set) и *ассоциативный массив* (associative array, map). Они поддерживают операции добавления, поиска и удаления элементов, но не обеспечивают эффективной реализации операций упорядоченного перебора ключей и поиска минимального и максимального среди них. Последнее является причиной того, что хеш-таблицы реализуют АТД неупорядоченное множество.

Мы начнем этот раздел с рассмотрения *таблиц с прямой адресацией* (direct-address tables).

13.1. Таблицы с прямой адресацией

Пусть нам требуется реализовать АТД *неупорядоченное множество* для хранения *целочисленных ключей* из совокупности $U = \{0, 1, \dots, u - 1\}$, где u – это мощность множества ключей.

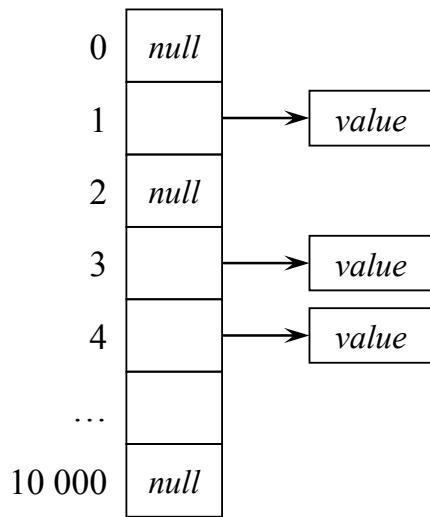


Рис. 13.1. Таблица $T[0..10000]$ с прямой адресацией, содержащая ключи 1, 3 и 4 ($U = \{0, 1, \dots, 10000\}$, $u = 10001$).

Таблицей с прямой адресацией (direct-address table) называется массив $T[0..u-1]$, в котором каждая ячейка взаимно-однозначно соответствует некоторому ключу из множества U . Ячейка с номером key содержит указатель на данные, ассоциированные с таким ключом. Если ключ key от-

существует в таблице, то соответствующая ячейка $T[key]$ содержит значение *null*. На рис. 13.1 показан пример таблицы, в которой содержатся ключи 1, 3 и 4 из множества $U = \{0, 1, \dots, 10000\}$, $u = 10001$.

Перед использованием таблицы с прямой адресацией необходимо выделить под нее память и инициализировать все ячейки значением *null*. На это требуется порядка $O(u)$ операций.

Ниже приведен псевдокод основных операций. Все они выполняются за время $O(1)$. На практике в приведенные функции следует добавить проверку корректности переданного значения *key*: $0 \leq key \leq u - 1$.

```

1 function INITIALIZE( $T[0..u - 1]$ )
2   for  $i = 0$  to  $u - 1$  do
3      $T[i] = null$ 
4   end for
5 end function

6 function INSERT( $T[0..m - 1]$ , key, value)
7    $T[key] = value$ 
8 end function

9 function LOOKUP( $T[0..m - 1]$ , key)
10   return  $T[key]$ 
11 end function

12 function DELETE( $T[0..m - 1]$ , key)
13    $T[key] = null$ 
14 end function
```

Положительными сторонами реализации неупорядоченного множества на базе таблицы с прямой адресацией является константная $O(1)$ вычислительная сложность основных операций. Главным недостатком является потребление порядка $\Theta(u)$ ячеек памяти, даже если используется лишь небольшое подмножество ключей исходной совокупности U .

13.2. Хеш-таблицы

Если совокупность ключей $U = \{0, 1, \dots, u - 1\}$ превосходит по размеру множество M реально используемых ключей, то применение таблиц с прямой адресацией становится неэффективным с точки зрения расходования оперативной памяти.

Хеш-таблицы позволяют снизить требования к оперативной памяти до $\Theta(|M|)$ для хранения ключей из совокупности U .

По сравнению с таблицами с прямой адресацией, хеш-таблицы реализуют основные операции за время $O(1)$ только в среднем случае. В худшем случае вычислительная сложность операций равна $O(n)$, где n – количество ключей, хранящихся в таблице. Это пример *пространственно-временного компромисса* – жертвуем временем ради сокращения использования памяти.

Хеш-таблица (hash table) – это массив $T[0..m - 1]$, в котором номер ячейки для ключа $key \in U$ вычисляется с использованием *хеш-функции* $hash(key)$. Причем размер m хеш-таблицы значительно меньше размера исходной совокупности ключей U . Ячейки хеш-таблицы также называют *слотами* (slot, bucket).

Хеш-функция (hash function) принимает в качестве аргумента целочисленный ключ из множества U и возвращает соответствующий ему номер ячейки таблицы $T[0..m - 1]$. Другими словами, хеш-функция отображает совокупность ключей U на множество ячеек хеш-таблицы $T[0..m - 1]$:

$$hash(key): U \rightarrow \{0, 1, \dots, m - 1\}.$$

Значение хеш-функции для ключа называют его *хеш-кодом* (hash code), *хеш-значением* (hash value) или просто *хешем* (hash).

Заметим, что в определении хеш-функции мы ограничились лишь ключами целочисленного типа. Далее мы рассмотрим способы преобразования ключей других типов данных (строки и вещественные числа) в целочисленные ключи из фиксированного диапазона.

Из-за того, что $|U| > m$, некоторым ключам из U неизбежно будет соответствовать одна и та же ячейка таблицы $T[0..m - 1]$. Такая ситуация, при которой для двух и более разных ключей хеш-функция возвращает одинаковый хеш-код, называется *коллизией* (collision). Формально

$$\forall key_1, key_2 \in U, \quad key_1 \neq key_2 : \quad hash(key_1) = hash(key_2).$$

Широкое распространение получили два основных *метода разрешения коллизий* (collision resolution): *метод цепочек* (chaining) и *открытая адресация* (open addressing). Задачей любого метода разрешения коллизий является формирование правил хранения в хеш-таблице ключей с одинаковыми хеш-кодами.

13.3. Метод цепочек

В методе цепочек каждая ячейка h хеш-таблицы $T[0..m - 1]$ содержит указатель на голову связного списка, в который помещаются все ключи имеющие одинаковый хеш-код h . В каждом узле списка содержится ключ и некоторое значение. Заметим, что хранение значений в узлах списков

не является обязательным и необходимо только при реализации АТД ассоциативный массив. При реализации неупорядоченных множеств в узлах списков достаточно хранить только ключи.

13.3.1. Реализация операций

Вставка. Для добавления элемента в хеш-таблицу вычисляется хеш-код h заданного ключа key . Найденный хеш-код позволяет получить доступ к связному списку $T[h]$ и добавить новый узел в его начало.

```

1 function INSERT( $T[0..m - 1]$ ,  $key$ ,  $value$ )
2    $h = \text{hash}(key)$ 
3    $T[h] = \text{LINKEDLISTADDFRONT}(T[h], key, value)$ 
4 end function
```

Поиск. Для заданного ключа key вычисляется его хеш-код h . После чего в связном списке $T[h]$ осуществляется поиск узла, содержащего заданный ключ key .

```

1 function LOOKUP( $T[0..m - 1]$ ,  $key$ )
2    $h = \text{hash}(key)$ 
3   return LINKEDLISTLOOKUP( $T[h]$ ,  $key$ )
4 end function
```

Удаление. Для того чтобы удалить из хеш-таблицы узел с ключом key , необходимо вычислить его хеш-код h и удалить из списка $T[h]$ соответствующий узел.

```

1 function DELETE( $T[0..m - 1]$ ,  $key$ )
2    $h = \text{hash}(key)$ 
3    $T[h] = \text{LINKEDLISTDELETE}(T[h], key)$ 
4 end function
```

Как и таблицы с прямой адресацией, хеш-таблицы требуют предварительной инициализации: выделения памяти под массив $T[0..m - 1]$ и записи в каждую ячейку значения $null$. Эта операция выполняется за время $\Theta(m)$.

Для освобождения памяти из-под всей хеш-таблицы требуется пройти по всем ячейкам хеш-таблицы и удалить узлы всех непустых связных списков. На это требуется $\Theta(m + n)$ операций.

Хеш-таблицы не позволяют *эффективно* реализовать операции MIN , MAX , SUCCESSOR и PREDECESSOR упорядоченного множества. Тем не менее реализовать их можно путем просмотра всех связных списков хеш-таблицы. Последнее требует порядка $O(m + n)$ операций сравнения ключей.

13.3.2. Сложность по памяти

Хеш-таблица, основанная на методе цепочек, занимает $\Theta(m+n)$ ячеек памяти. Здесь важно отметить, что помимо памяти для хранения ключей и значений мы также вынуждены хранить вспомогательные указатели на узлы списков. На хранение указателей на головы связных списков требуется $\Theta(m)$ ячеек. Каждый узел списка, как минимум, содержит указатель *next* на следующий элемент. Следовательно, требуется еще $\Theta(n)$ ячеек памяти. Таким образом, хеш-таблица на базе метода цепочек для хранения вспомогательных указателей требует порядка $\Theta(m+n)$ ячеек памяти.

13.3.3. Вычислительная сложность операций

Оценим вычислительную сложность рассмотренных выше операций. Вычислительная сложность вставки элемента в хеш-таблицу равна $O(1)$. Это обусловлено тем, что время вычисления хеш-функции не зависит от числа ключей в хеш-таблице и равно $O(1)$ (в этом мы убедимся далее при рассмотрении требований к хеш-функциям). Кроме этого, добавление узла в начало связного списка также требует порядка $O(1)$ операций.

Время выполнения поиска и удаления элемента из хеш-таблицы определяется длиной соответствующего связного списка. В худшем случае все ключи могут попасть в один связный список длины $\Theta(n)$ – случай неэффективной хеш-функции. Следовательно, вычислительная сложность операций поиска и удаления элемента из хеш-таблицы в худшем случае равна $\Theta(n)$.

Зная число n ключей в хеш-таблице и ее размер m , можно вычислить *коэффициент α заполненности* хеш-таблицы (load factor) – среднее число узлов, приходящееся на одну ячейку хеш-таблицы:

$$\alpha = \frac{n}{m}.$$

На рис. 13.2 показан пример хеш-таблицы на базе метода цепочек, содержащей семь ключей. Некоторые ключи имеют одинаковые хеш-коды (21, 12, 30, 39) и сгруппированы в связные списки. Коэффициент заполненности для этой таблицы равен $\alpha = n/m = 7/9 = 0.78$.

Будем считать, что в *среднем случае хеш-функция равномерно распределает ключи* по m ячейкам хеш-таблицы так, что математическое ожидание длины связных списков равно n/m . Анализ этого случая требует рассмотрения двух ситуаций: искомый ключ *key* отсутствует в хеш-таблице и ключ находится в одном из связных списков.

Если искомый ключ *key* отсутствует в хеш-таблице, то поиск и удаление элемента требует перебора всех $\alpha = n/m$ узлов связного списка $T[hash(key)]$. Следовательно, вычислительная сложность этих операций в

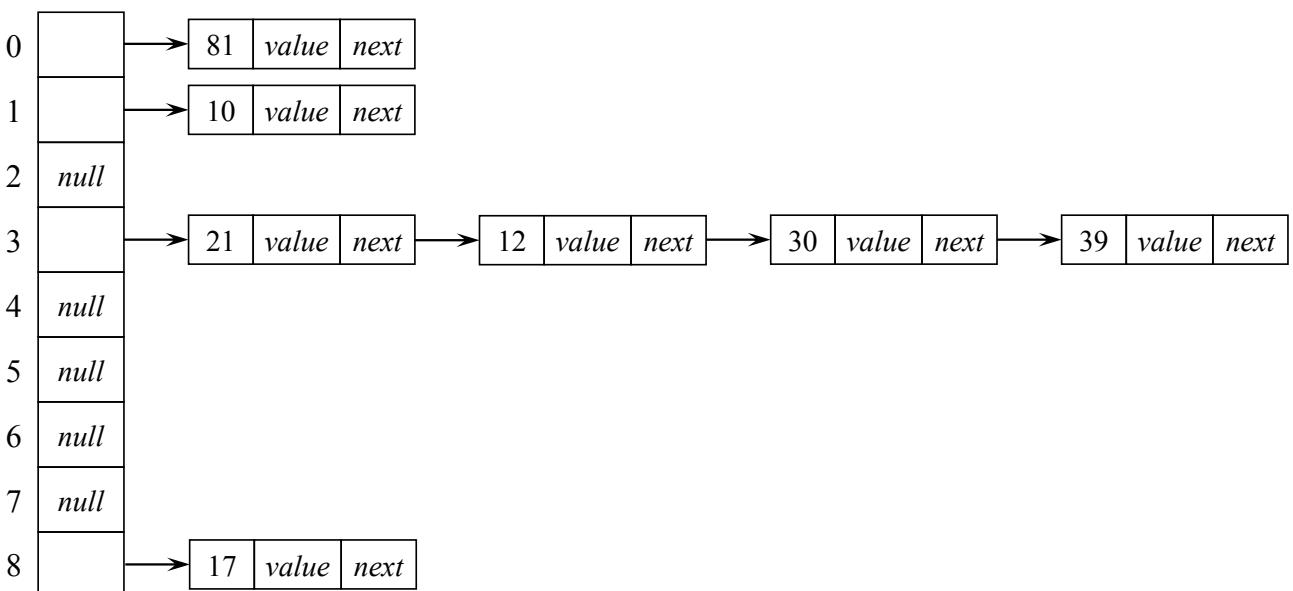


Рис. 13.2. Хеш-таблица $T[0..8]$ на базе метода цепочек, содержащая семь ключей: $n = 7$, $m = 9$, $\alpha = 0.78$.

среднем случае при отсутствии ключа в хеш-таблице будет $O(1 + \alpha)$.

Если ключ key присутствует в хеш-таблице, то он может с одинаковой вероятностью $1/\alpha$ находится в любом из α узлов связного списка $T[hash(key)]$. Если ключ находится в первом узле, то для его поиска требуется одно сравнение, если во втором – выполняется два сравнения и т. д. Математическое ожидание числа операций сравнения ключей при поиске узла равно

$$\frac{1}{\alpha} \cdot 1 + \frac{1}{\alpha} \cdot 2 + \dots + \frac{1}{\alpha} \cdot \alpha \approx \frac{1 + \alpha}{2}.$$

Следовательно, вычислительная сложность успешного поиска и удаления элемента из хеш-таблицы в среднем случае также равна $O(1 + \alpha)$.

Анализ среднего случая можно провести и другим методами. Например, в [1] путем введения индикаторных случайных величин показано, что в среднем случае вычислительная сложность поиска и удаления элемента из хеш-таблицы на базе метода цепочек равна $\Theta(1 + n/m)$. В [11] приведено доказательство леммы о том, что с вероятностью, близкой к 1, длина каждого связного списка незначительно отличается от n/m .

В табл. 13.1 приведены вычислительные сложности основных операций хеш-таблицы на базе метода цепочек.

Важно заметить, что производительность в среднем здесь зависит от того является ли α константой ($\alpha < 1$). Если значение α становится больше 1, то имеет смысл перестроить хеш-таблицу – создать новую большего размера такую, что $\alpha < 1$, и перенести в нее все ключи.

Таблица 13.1. Вычислительная сложность операций хеш-таблицы на базе метода цепочек

Операция	Средний случай	Худший случай
INITIALIZE($T[0..m - 1]$)	$\Theta(m)$	$\Theta(m)$
INSERT($T[0..m - 1], key, value$)	$O(1)$	$O(1)$
LOOKUP($T[0..m - 1], key$)	$O(1 + n/m)$	$O(n)$
DELETE($T[0..m - 1], key$)	$O(1 + n/m)$	$O(n)$

Дополнительные операции		
$\text{MIN}(T[0..m - 1])$	$\Theta(m + n)$	$\Theta(m + n)$
$\text{MAX}(T[0..m - 1])$	$\Theta(m + n)$	$\Theta(m + n)$
$\text{PREDECESSOR}(T[0..m - 1], key)$	$\Theta(m + n)$	$\Theta(m + n)$
$\text{SUCCESSOR}(T[0..m - 1], key)$	$\Theta(m + n)$	$\Theta(m + n)$
$\text{DELETEHASHTABLE}(T[0..m - 1], key)$	$\Theta(m + n)$	$\Theta(m + n)$

13.4. Открытая адресация

Основной недостаток метода цепочек – использование порядка $\Theta(m + n)$ ячеек памяти для хранения вспомогательных указателей связных списков. Основная идея открытой адресации – хранение ключей и ассоциированных с ними значений непосредственно в ячейках таблицы $T[0..m - 1]$. Такой подход позволяет более экономно использовать память.

Нетрудно заметить, что при использовании такой стратегии в хеш-таблице может храниться не более m ключей и, как следствие, значение коэффициента $\alpha = n/m$ не может превышать 1.

	<i>key</i>	<i>value</i>	<i>state</i>
0	81		<i>Filled</i>
1	10		<i>Filled</i>
2			<i>Empty</i>
3	21		<i>Filled</i>
4	12		<i>Filled</i>
5	30		<i>Filled</i>
6	39		<i>Filled</i>
7			<i>Empty</i>
8	17		<i>Filled</i>

Рис. 13.3. Хеш-таблица $T[0..8]$ на базе открытой адресации, содержащая семь ключей: $n = 7$, $m = 9$, $\alpha = 0.78$.

13.4.1. Реализация операций

Для выполнения вставки, поиска и удаления элементов необходимо каким-то образом определять, является ли ячейка таблицы свободной. Это может быть реализовано несколькими способами:

- в каждой ячейке хранить дополнительное поле, информирующее о ее состоянии (свободна, занята, удалена и др.);
- использовать специальное значение ключа для обозначения незанятой ячейки;
- хранить в ячейке указатель на запись с ключом и данными (*null* сигнализирует о том, что ячейка свободна).

Далее будем считать, что вместе с ключом и значением в ячейке содержится дополнительное поле, отражающее ее состояние (*state*).

Каждая ячейка хеш-таблицы может находиться в любом из трех состояний: *свободна* (*Empty*), *заполнена* (*Filled*) и *удалена* (*Deleted*). Изначально все ячейки находятся в состоянии *Empty*. Ячейка может находиться только в одном из этих состояний. Смысл состояния *Deleted* будет раскрыт в процессе рассмотрения операции удаления ключа из хеш-таблицы.

На рис. 13.3 приведен пример хеш-таблицы $T[0..8]$ на базе открытой адресации. Она содержит семь ключей, причем ключи 21, 12, 30 и 39 имеют одинаковый хеш-код 3 и размещены в смежных ячейках.

Для определения состояния ячеек доступны функции $\text{EMPTY}(T, h)$ и $\text{DELETED}(T, h)$, которые возвращают значение *True*, если ячейка h находится в соответствующем состоянии, и *False* – в противном случае.

При возникновении коллизии (ячейка $T[h]$ занята) требуется каким-то образом перебрать оставшиеся $m - 1$ ячеек с целью поиска свободной позиции для вставки нового элемента или для удаления заданного ключа. Процедура поиска ячейки в хеш-таблице называется *исследованием* (*probing*). Для заданного ключа она задает последовательность перебора m ячеек хеш-таблицы. Всего существует $m!$ вариантов исследований m ячеек таблицы (по числу перестановок множества $\{0, 1, \dots, m - 1\}$).

Для того чтобы задать порядок исследования, хеш-функцию расширяют добавлением в нее второго аргумента $i \in \{0, 1, \dots, m - 1\}$ – номера исследования:

$$\text{hash}(\text{key}, i) : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

Таким образом, если ячейка $\text{hash}(\text{key}, 0)$ занята, выполняем проверку ячейки $\text{hash}(\text{key}, 1)$, затем ячейки $\text{hash}(\text{key}, 2)$ и т. д., пока не найдем свободную или не проверим все m ячеек.

Ниже мы рассмотрим несколько подходов к заданию хеш-функций $\text{hash}(\text{key}, i)$, реализующих различные варианты исследований ячеек хеш-таблицы.

Вставка. Для добавления элемента в хеш-таблицу вычисляется хеш-код $hash(key, 0)$ заданного ключа key . Если ячейка $T[hash(key, 0)]$ свободна или помечена как удаленная, то ключ и значение записываются в нее. При занятой ячейке среди оставшихся $m - 1$ ячеек отыскивается доступная для использования.

В случае успешного добавления элемента функция `INSERT` возвращает номер найденной ячейки. Если таблица заполнена, функция возвращает специальное значение `HashTableOverflow`.

Очевидно, что при отсутствии свободной ячейки мы будем вынуждены выполнить порядка $O(m)$ исследований.

```

1 function INSERT( $T[0..m - 1]$ ,  $key$ ,  $value$ )
2      $i = 0$ 
3     while  $i < m$  do
4          $h = hash(key, i)$ 
5         if EMPTY( $T, h$ ) or DELETED( $T, h$ ) then
6              $T[h].key = key$ 
7              $T[h].value = value$ 
8             return  $h$ 
9         end if
10         $i = i + 1$ 
11    end while
12    return HashTableOverflow
13 end function
```

Поиск. При поиске элемента мы последовательно проверяем ячейки с номерами $hash(key, 0), hash(key, 1), \dots, hash(key, m - 1)$, пока не найдем требуемую. В процессе исследований мы пропускаем ячейки, находящиеся в состоянии *Deleted*, и останавливаем поиск при обнаружении пустой ячейки. Приведенная ниже функция `LOOKUP` в случае успеха возвращает номер ячейки с искомым ключом и -1 в случае ошибки.

Удаление. Процедура удаления элемента является самой сложной в реализации. Первым делом мы отыскиваем ячейку $T[h]$, содержащую заданный ключ key . Мы не можем использовать наивный подход, основанный на пометке удаляемой ячейки $T[h]$ как пустой. Это нарушит структуру хеш-таблицы и не позволит находить ключи, при вставке которых ячейка h была занятой. На рис. 13.3 приведен пример хеш-таблицы, в которойключи 21, 12, 30 и 39 имеют одинаковый хеш-код 3 и размещены в смежных ячейках. Если при удалении ключа 12 мы просто изменим его состояние на *Empty*, то потеряем возможность находить ключи 30 и 39. Для решения этой проблемы мы будем переводить удаляемые ячейки в состояние *Deleted*. При вставке новых элементов такие ячейки будут считаться свободными, а при поиске – игнорироваться.

```

1 function LOOKUP( $T[0..m - 1]$ ,  $key$ )
2      $i = 0$ 
3     while  $i < m$  do
4          $h = hash(key, i)$ 
5         if EMPTY( $T, h$ ) then
6             return  $-1$ 
7         else if  $T[h].key = key$  then
8             return  $h$ 
9         end if
10         $i = i + 1$ 
11    end while
12    return  $-1$ 
13 end function

```

В случае успешного удаления ключа из хеш-таблицы функция DELETE возвращает его позицию в массиве и -1 при отсутствии ключа в таблице.

```

1 function DELETE( $T[0..m - 1]$ ,  $key$ )
2      $i = 0$ 
3     while  $i < m$  do
4          $h = hash(key, i)$ 
5         if EMPTY( $T, h$ ) then
6             return  $-1$ 
7         else if  $T[h].key = key$  then
8              $T[h].state = Deleted$ 
9             return  $h$ 
10        end if
11         $i = i + 1$ 
12    end while
13    return  $-1$ 
14 end function

```

13.4.2. Стратегии исследований

Одной из самых простых стратегий является *линейное исследование* (linear probing), при котором ячейки проверяются последовательно одна за другой.

Хеш-функция в этом случае имеет следующий вид:

$$hash(key, i) = (hash(key) + i) \% m,$$

где $i \in \{0, 1, \dots, m-1\}$. Если при проверке достигнут конец таблицы, то по-

иск продолжается с ячейки 0. Таким образом, при линейном исследовании проверяются ячейки

$$T[\text{hash}(\text{key})], T[\text{hash}(\text{key}) + 1], \dots, T[m - 1], T[0], \dots, T[\text{hash}(\text{key}) - 1].$$

При линейном исследовании в процессе добавления ключей возникает эффект кластеризации. *Кластер* (cluster) – это последовательность смежных заполненных ячеек (с учетом того, что таблица трактуется как циклический массив). Образование кластеров, их рост и слияние негативно сказываются на времени выполнения всех операций хеш-таблицы. В примере на рис. 13.3 присутствует два кластера: кластер из ключей 21, 12, 30, 39 и кластер 17, 81, 10. Если добавить в указанную хеш-таблицу ключ 48 с хеш-кодом 3, то он займет позицию 7. Это приведен к слиянию двух кластеров в один, что влечет увеличение времени исследования ячеек.

Для снижения эффекта кластеризации разработаны другие стратегии исследований. Среди них *квадратичное исследование* (quadratic probing) и *двойное хеширование* (double hashing).

Основная причина кластеризации при линейном исследовании – последовательный просмотр смежных ячеек. Вместо этого в методе квадратичного исследования проверяются ячейки, находящиеся на расстоянии, квадратично зависящем от номера $i \in \{0, 1, \dots, m - 1\}$ исследования. Здесь хеш-функция имеет следующий вид:

$$\text{hash}(\text{key}, i) = (\text{hash}(\text{key}) + c_1 i + c_2 i^2) \% m,$$

где $c_2 > 0$. Константы c_1 и c_2 подбираются при проектировании хеш-таблицы. Например, при $c_1 = 0$ и $c_2 = 1$, для ключа key проверяются следующие ячейки хеш-таблицы:

$$T[\text{hash}(\text{key}) + 0^2], T[\text{hash}(\text{key}) + 1^2], \dots, T[(\text{hash}(\text{key}) + i^2) \% m], \dots$$

Такая процедура исследования снижает вероятность образования кластеров и тем самым положительно влияет на время выполнения основных операций хеш-таблицы.

Однако по сравнению с линейным исследованием квадратичная схема исследования обладает меньшей эффективностью использования кеш-памяти процессора, так как приводит к обращениям по не смежным адресам памяти.

Наиболее эффективной стратегией исследования является *двойное хеширование* (double hashing), при котором хеш-функция использует вторую (вспомогательную) хеш-функцию $\text{hash}'(\text{key})$

$$\text{hash}(\text{key}, i) = (\text{hash}(\text{key}) + i \cdot \text{hash}'(\text{key})) \% m,$$

В отличие от линейного и квадратичного исследований, при двойном хешировании номер первой исследуемой ячейки и расстояние между проверяемыми ячейками зависят от значения ключа. Это приводит к тому, что получаемые последовательности перебора ячеек обладают многими характеристиками случайно выбираемых перестановок [1, 2].

Для того чтобы значения хеш-функции $hash(key, i)$ охватывали номера всех ячеек таблицы, значения функции $hash'(key)$ должны быть взаимно простыми с размером t хеш-таблицы. Этого можно добиться, если взять в качестве t число, равное степени 2, а из функции $hash'(key)$ возвращать нечетные значения. Другой вариант – взять в качестве t простое число, а $hash'(key)$ должна возвращать натуральные числа, меньшие t [1].

13.4.3. Эффективности открытой адресации

Как и в случае метода цепочек производительность операций для открытой адресации зависит от значения коэффициента α заполненности хеш-таблицы. Чем ближе его значение к 1, тем медленнее работают операции вставки, поиска и удаления ключей. В худшем случае каждая операция выполняется за время $O(n)$.

Если предположить, что хеш-функция равномерно распределяет ключи по ячейкам хеш-таблицы и $\alpha < 1$, то можно получить следующие оценки числа исследований для основных операций [1, 2, 4]:

- среднее число исследований при поиске ключа, отсутствующего в хеш-таблице, равно $1/(1 - \alpha)$;
- среднее число исследований при поиске ключа, присутствующего в хеш-таблице, равно $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$;
- среднее число исследований при вставке ключа в хеш-таблице равно $1/(1 - \alpha)$.

13.5. Хеш-функции

Время выполнения операций хеш-таблицы во многом определяется эффективностью используемой хеш-функции. Рассмотрим основные требования, предъявляемые к ним.

1. *Равномерное хеширование* (uniform hashing). Хеш-функция должна как можно равномернее распределять входные ключи по t ячейкам хеш-таблицы (обеспечивать минимум коллизий).

2. *Детерминированность* (determinism). Для заданного ключа key хеш-функция должна всегда возвращать один и тот же хеш-код $hash(key)$.

3. Время вычисления хеш-функции не должно зависеть от количества n ключей, находящихся в хеш-таблице. Число операций, выполняе-

мых хеш-функцией, должно зависеть только от длины ключа key . Поэтому считается, что вычислительная сложность хеш-функции равна $O(1)$ или $O(|key|)$.

Рассмотрим два основных подхода к построению хеш-функций: *метод деления* (division method) и *метод умножения* (multiplication method).

13.5.1. Метод деления

Хеш-функция, основанная на *методе деления*, отображает целочисленный ключ $key \in U$ во множество $\{0, 1, \dots, m - 1\}$ путем взятия остатка от деления key на m

$$\text{hash}(key) = key \% m.$$

Такие хеш-функции также называют *модульными* (modular hashing). Например, если $m = 111$, а $key = 3400$, то $\text{hash}(3400) = 70$.

При использовании этого метода важным вопросом является выбор размера m хеш-таблицы. От этого зависит эффективность хеш-функции. Значение m не должно быть степенью 2. Если поступить иначе и положить $m = 2^p$, то функция $\text{hash}(key)$ будет возвращать в качестве значения p младших битов ключа key . Поэтому еще одним крайне желательным требованием, предъявляемым к хеш-функциям, является зависимость их результата от всех битов ключа.

Рекомендуется в качестве m брать простое число, далекое от степени 2. Например, можно использовать следующие значения m :

$$97, 997, 9973, 99991, 999983, \dots$$

$$101, 1009, 10007, 100003, 1000003, \dots$$

Допустим, в хеш-таблице требуется хранить не более 5000 ключей и нас устраивает порядка 2 сравнений при неудачном поиске, тогда в качестве размера хеш-таблицы можно взять значение $m = 2503$ как простое число, близкое к $5000/2 = 2500$. Хеш-функция примет следующий вид

$$\text{hash}(key) = key \% 2503.$$

13.5.2. Метод умножения

Исходя из практических соображений или технических требований размер хеш-таблицы может быть выбран как степень 2 (например, по размеру страницы виртуальной памяти и пр.), тогда использовать метод деления становится затруднительно. В этой ситуации можно применить *метод умножения*, который работает для любых значений m .

В методе умножения ключ $key \in U = \{0, 1, \dots, u - 1\}$ умножается

на заранее выбранную константу $0 < A < 1$. От полученного значения берется дробная часть и умножается на m . В качестве итогового результата возвращается ближайшее целое число снизу

$$\text{hash}(\text{key}) = \lfloor m \cdot (\text{key} \cdot A \% 1) \rfloor,$$

где через $\text{key} \cdot A \% 1$ обозначена дробная часть выражения $\text{key} \cdot A$. В процессе вычисления хеш-функции промежуточные значения находятся в следующих границах: $0 \leq \text{key} \cdot A < \text{key}$, $0 \leq \text{key} \cdot A \% 1 < 1$.

На практике неплохие результаты дает следующий выбор значения константы A [5, 11]:

$$A = (\sqrt{5} - 1)/2 = 0.618033\dots$$

Вернемся к примеру, рассмотренному выше, – требуется хранить не более 5000 ключей и нас устраивает порядка 2 сравнений при неудачном поиске. В качестве размера хеш-таблицы выбрано значение $m = 1024$, $A = 0.618$. Тогда, используя метод умножения, хеш-функция примет следующий вид

$$\text{hash}(\text{key}) = \lfloor 1024 \cdot (\text{key} \cdot 0.618 \% 1) \rfloor,$$

Вычислим значение полученной хеш-функции для ключа $\text{key} = 453$

$$\text{hash}(453) = \lfloor 1024 \cdot (453 \cdot 0.618 \% 1) \rfloor = \lfloor 1024 \cdot 0.954 \rfloor = 976.$$

Операции умножения и деления целых чисел на степень 2 можно эффективно реализовать при помощи операций поразрядных сдвигов влево и вправо.

13.5.3. Другие методы построения хеш-функций

Для хеш-функций, построенных на базе рассмотренных методов всегда можно подобрать «плохую» последовательность из n ключей, которые будут иметь одинаковый хеш-код. Это приведет к тому, что поиск будет выполняться за время $\Theta(n)$. Любая фиксированная хеш-функция уязвима к подобного рода атакам. Решением рассмотренной проблемы является случайный выбор хеш-функции из некоторого наперед заданного семейства хеш-функций. Такой выбор осуществляется каждый раз перед началом использования хеш-таблицы. Подход получил название *универсальное хеширование* (universal hashing).

Встречаются ситуации, когда множество ключей известно заранее или начиная с некоторого момента времени новые ключи не добавляются в хеш-таблицу. Например, множество имен файлов на DVD-диске или множество названий улиц города. Для *статических множеств* ключей суще-

стует метод построения хеш-таблиц, который обеспечивает выполнение поиска за время $O(1)$ даже в худшем случае. Такой метод называется *идеальное*, или *совершенное хеширование* (perfect hashing). Основная идея метода заключается в том, что зная информацию о множестве ключей строится двухуровневая хеш-таблица, на каждом уровне которой используется универсальное хеширование.

Подробнее об универсальном и идеальном хешировании можно прочитать в [1].

13.6. Преобразование ключей в целые числа

Мы рассмотрели основные операции хеш-таблицы в предположении, что ключи – это целые числа из совокупности $U = \{0, 1, \dots, u - 1\}$. На практике приходится иметь дело с ключами и других типов данных. Например, ключи могут быть иметь строковый или вещественный тип данных. Рассмотрим способы преобразования таких типов данных в целые числа.

13.6.1. Строковый тип данных

Строка символов может рассматриваться как последовательность чисел, где каждое число – это код соответствующего символа в кодировке ASCII, Unicode и др. Например, строке «color» в кодировке ASCII соответствует последовательность чисел (99, 111, 108, 111, 114). Поэтому строку символов можно рассматривать как число, записанное в системе исчисления с основанием 128 (мы считаем, что для кодирования символов используется 7 бит). Переведем число $(99, 111, 108, 111, 114)_{128}$ в десятичную систему исчисления

$$99 \cdot 128^4 + 111 \cdot 128^3 + 108 \cdot 128^2 + 111 \cdot 128 + 114 \cdot 128^0 = 26809677810.$$

Обозначим через s_i целочисленный код символа i строки s , l – длина строки. Тогда общая формула преобразования строки s , интерпретируемой как число в системе исчисления с основанием b , в целое значение $h(s)$ имеет следующий вид:

$$h(s) = \sum_{i=0}^{l-1} b^{l-i-1} \cdot s_i.$$

Для вычисления этого выражения можно использовать схему Горнера.

Существуют и другие способы преобразования строк в целые числа. Однако важно учесть, что используемое преобразование должно разным ключам сопоставлять разные целочисленные значения (не вносить дополнительных ограничений на количество символов в строке).

нительных коллизий). Например, если мы для преобразования строки в число будем просто суммировать коды символов строки, то обнаружим, что ключам «color», «goloc», «lorco» и др. сопоставлено одно и то же целое число 543.

Рассмотренная схема преобразования строки в число применима и для произвольных последовательностей чисел (например, одномерных массивов).

Ниже приведен пример модульной хеш-функции HASHSTRING для строк в кодировке ASCII [11]. За основание системы исчисления взято простое число 127. Это позволяет сократить число коллизий, в случае если размер m таблицы заранее неизвестен и является степенью 2 или кратен 2. Вычисления выполняются по схеме Горнера и требуют порядка $O(l)$ операций.

```

1 function HASHSTRING( $s[0..l - 1]$ )
2      $h = 0$ 
3     for  $i = 0$  to  $l - 1$  do
4          $h = (h \cdot 127 + s_i) \% m$ 
5     end for
6     return  $h$ 
7 end function
```

В [12] предлагается использовать для строк аналогичную модульную хеш-функцию, но в качестве основания системы исчисления предлагается брать простые значения $b = 31$ или $b = 37$.

13.6.2. Вещественный тип данных

Если ключ – это вещественное число из диапазона $(0, 1)$, то его достаточно умножить на m и взять ближайшее целое число снизу. Если же ключом является вещественное число из ограниченного диапазона $\min < key < \max$, то для его преобразования в целое число из множества $\{0, 1, \dots, m - 1\}$ можно использовать функцию HASHFLOAT.

```

1 function HASHFLOAT( $key$ )
2     return  $\lfloor m \cdot (key - \min) / (\max - \min) \rfloor$ 
3 end function
```

13.6.3. Выбор размера хеш-таблицы

Если заранее известна оценка числа ключей, которые будут храниться в хеш-таблице, то можно выбрать ее размер для обеспечения приемлемого значения коэффициента α . Например, если в хеш-таблице на базе метода

цепочек будет храниться порядка $n = 10000$ ключей и мы хотим, чтобы в среднем выполнялось не более четырех проверок при неудачном поиске, то размер m хеш-таблицы следует выбрать как

$$m = \frac{n}{\alpha} = \frac{10000}{4} = 2500.$$

В процессе выполнения операций над хеш-таблицей мы можем контролировать показатель α и при достижении порогового значения (например, $2/3$ или $3/4$) *динамически увеличивать размер хеш-таблицы*.

Размер таблицы можно увеличить на заданную константу (*аддитивная схема*) или в несколько раз (*мультипликативная схема*). После чего все пары $(key, value)$ исходной хеш-таблицы вставляются в новую. Например, в хеш-таблицу размера $m = 1024$ вставлено $n = 768$ ключей. Коэффициент $\alpha = n/m$ достиг порогового значения 0.75. Тогда если следовать мультипликативной схеме и увеличить размер хеш-таблицы в 2 раза ($m = 2048$), мы сократим значение α новой таблицы до 0.38.

13.7. Упражнения

- Почему в методе цепочек добавление нового ключа осуществляется в начало связного списка, а не в конец?
- Как в методе цепочек реализовать добавление ключа в конец соответствующего связного списка за время $O(1)$?
- Предложите алгоритм поиска в хеш-таблице ключа с минимальным/максимальным значением.
- Разработайте алгоритм подсчета числа коллизий в хеш-таблице на базе метода цепочек.
- Модифицируйте процедуру вставки ключа в хеш-таблицу так, чтобы она гарантировала уникальность всех ключей в таблице. Как изменится вычислительная сложность операции?
- В некоторой программе требуется поддерживать ассоциативный массив, в котором ключом являются вещественные координаты (x, y, z) точки в трехмерном пространстве. Предложите хеш-функцию для таких ключей.
- Оцените вычислительную сложность операции увеличения размера имеющейся хеш-таблицы в два раза (мультипликативная схема).

14. Бинарные кучи

Бинарные кучи (binary heap) предназначены для реализации АТД *очередь с приоритетом*. Они поддерживают операции поиска и удаления элемента с экстремальным значением ключа (либо минимальным, либо максимальным), но не предоставляют возможности поиска элемента по ключу.

Бинарные кучи также называют *пирамидами* и *сортирующими деревьями*. Алгоритмы этой структуры данных лежат в основе *пирамидальной сортировки* (heapsort).

14.1. Представление в памяти

Бинарной кучей (binary heap) называется завершенное бинарное дерево (complete binary tree), в котором каждый узел содержит ключ и значение. Ключ выступает в роли *приоритета узла*, а значение – это некоторые данные, ассоциированные с ключом. Структура дерева и распределение ключей по нему удовлетворяют следующим требованиям:

- каждый уровень дерева, возможно за исключением последнего, полностью заполнен узлами, а заполнение последнего уровня осуществляется слева направо;
- ключ любого узла не меньше (не больше) ключей его потомков.

Как мы убедились ранее (см. 9.3.2), завершенные бинарные деревья можно хранить в одномерном массиве. Все узлы дерева размещаются в ячейках массива $A[1..n]$ в порядке их обхода в ширину. Корень дерева хранится в первой ячейке массива. Для любой вершины, размещенной в ячейке $i \in \{1, 2, \dots, n\}$, левый дочерний узел $\text{LEFT}(i)$ расположен в позиции $2i$, а правый дочерний узел $\text{RIGHT}(i)$ в позиции $2i + 1$. Родительский узел $\text{PARENT}(i)$ вершины i находится в ячейке $\lfloor i/2 \rfloor$. Если значения $2i$ или $2i + 1$ больше n , то у вершины i отсутствует соответствующий дочерний узел.

Известно, что число уровней в завершенном бинарном дереве (куче) из n ключей равно $\lfloor \log_2 n \rfloor + 1$.

На рис. 14.1 приведен пример двух бинарных куч из шести элементов, а на рис. 14.2 представление бинарной кучи из рис. 14.1, а в виде одномерного массива.

В зависимости от правила распределения ключей по узлам дерева выделяют два типа бинарных куч.

```

1 function LEFT(i)
2     return 2i
3 end function

4 function RIGHT(i)
5     return 2i + 1
6 end function

7 function PARENT(i)
8     return  $\lfloor i/2 \rfloor$ 
9 end function

```

Бинарная куча, в которой ключ любого узла не меньше ключей его потомков, называется *невозрастающей бинарной кучей*, или *невозрастающей пирамидой* (max-heap). В таком дереве ключ с максимальным значением расположен в его корне:

$$A[\text{PARENT}(i)] \geq A[i], \quad \forall i \in \{2, 3, \dots, n\}.$$

Аналогично, бинарная куча, в которой ключ любого узла не больше ключей его потомков, называется *неубывающей бинарной кучей*, или *неубывающей пирамидой* (min-heap). В корне дерева находится ключ с минимальным значением.

$$A[\text{PARENT}(i)] \leq A[i], \quad \forall i \in \{2, 3, \dots, n\}.$$

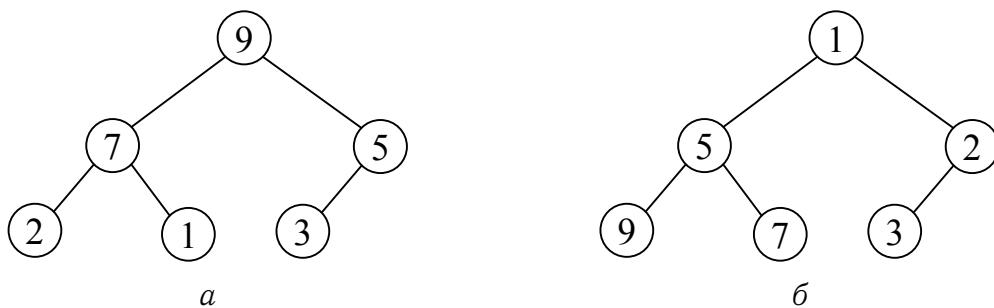


Рис. 14.1. Бинарная куча из шести элементов:
a – невозрастающая куча (max-heap); *б* – неубывающая куча (min-heap).

14.2. Операции бинарной кучи

Для хранения элементов бинарной кучи используется массив $A[1..m]$ из m ячеек. Каждая ячейка i хранит ключ $A[i].key$ и некоторое значение $A[i].value$. Очевидно, что при таком представлении максимальное число

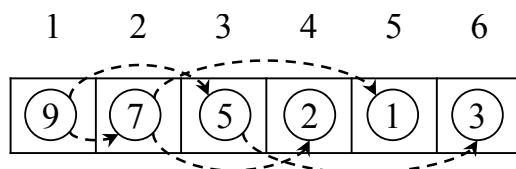


Рис. 14.2. Представление бинарной кучи из 6 элементов (max-heap) в виде массива $A[1..6]$

узлов в куче не превышает m . Для определенности обозначим через n количество элементов, реально хранящихся в бинарной куче ($n \leq m$).

Мы рассмотрим реализацию операций *невозрастающей бинарной кучи* (max-heap). Операции неубывающей бинарной кучи реализуются аналогично.

Поиск максимального элемента. По определению элемент с максимальным ключом (приоритетом) хранится в первой ячейке массиве. Если куча не содержит элементов, то функция *Max* возвращает специальное значение *HeapEmpty*. Вычислительная сложность операции равна $O(1)$.

```

1 function Max( $A[1..m]$ )
2   if  $n < 1$  then
3     return HeapEmpty
4   end if
5   return  $A[1]$ 
6 end function

```

Вставка элемента. Новый элемент добавляется в конец массива. В дереве этой позиции соответствует самый правый лист на последнем уровне. Если свободных ячеек в массиве не осталось, функция *INSERT* возвращает специальное значение *HeapOverflow*.

После вставки нового элемента могут быть нарушены свойства невозрастающей бинарной кучи – добавленный ключ может быть больше ключа родительского узла. Поэтому нам необходимо восстановить свойства бинарной кучи. Для этого мы проходим в функции *HEAPIFYUP* по дереву от листа $A[n]$ до корня $A[1]$ и выполняем обмены узлов, если ключ родительского узла меньше ключа текущего элемента. Перебор узлов прерывается, если ключ родителя стал больше ключа текущего узла.

В худшем случае элементы добавляются в порядке возрастания ключей. В такой ситуации после каждого добавления мы вынуждены пройти все $O(\log n)$ уровней бинарной кучи и выполнить соответствующее число обменов значений узлов. Следовательно, вычислительная сложность вставки в худшем случае равна $O(\log n)$.

На рис. 14.3 показан пример добавления ключа 11 в кучу из шести элементов. Видно, что после вставки нового ключа 11 в конец массива нарушены свойства невозрастающей бинарной кучи (max-heap): ключ 5

меньше ключа 11. Узлы 5 и 11 меняются местами. Далее выполняются сравнение ключей 9 и 11 и перестановка соответствующих узлов. Таким образом ключ 11 оказался в корне дерева. Свойства бинарной кучи восстановлены.

```

1 function INSERT( $A[1..m]$ ,  $key$ ,  $value$ )
2   if  $n = m$  then
3     return HeapOverflow
4   end if
5    $n = n + 1$ 
6    $A[n].key = key$ 
7    $A[n].value = value$ 
8   HEAPIFYUP( $A$ ,  $n$ )
9 end function

10 function HEAPIFYUP( $A[1..m]$ ,  $i$ )
11   while  $i > 1$  and  $A[\text{PARENT}(i)].key < A[i].key$  do
12     SWAP( $A[i]$ ,  $A[\text{PARENT}(i)]$ )           /* Обмен значений узлов */
13      $i = \text{PARENT}(i)$ 
14   end while
15 end function

```

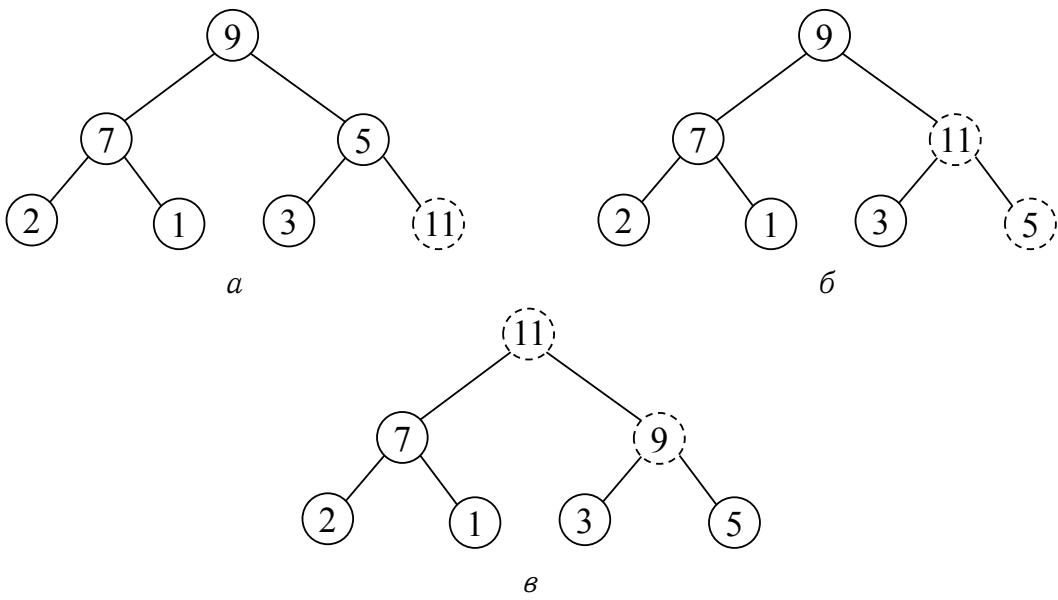


Рис. 14.3. Добавление ключа 11 в невозрастающую бинарную кучу:

- а* – добавление ключа 11 в конец массива – ячейку $A[7]$;
- б* – результат обмена значениями узлов 11 и 5;
- в* – результат обмена значениями узлов 11 и 9.

Удаление максимального элемента. Для удаления максимального элемента $A[1]$ мы запоминаем его значение для последующего возврата из

функции `DELETEMAX`. Далее последний элемент $A[n]$ переносится в ячейку $A[1]$, число n элементов в куче уменьшается на 1. После этого необходимо восстановить свойства невозрастающей кучи, так как в корне дерева теперь может находиться элемент не с максимальным значением.

Восстановление свойств кучи выполняется функцией `HEAPIFYDOWN`, которая в цикле движется от корня к листьям. На каждой итерации цикла среди текущего узла i и двух его дочерних элементов $left$ и $right$ выбирается наибольший $largest$ – тот, который содержит наибольший ключ. Если наибольшим оказался один из дочерних узлов, то выполняется обмен текущего узла i и найденного наибольшего. Проход по дереву останавливается, если наибольшее значение ключа оказалось у текущего элемента i , а не у его дочерних. Это означает, что дальше свойства кучи восстанавливать не требуется.

Время выполнения функции `HEAPIFYDOWN` зависит от высоты h узла i , для которого она вызвана. Таким образом, время выполнения этой функции есть $O(h)$. В худшем случае `HEAPIFYDOWN` осуществляет проход по всем уровням завершенного бинарного дерева (от корня до листа), что требует $O(\log n)$ операций обменов.

На рис. 14.4 показан пример удаления ключа 11 с максимальным значением. Последний элемент 5 перемещен в корень дерева – ячейку $A[1]$. После чего запущена процедура `HEAPIFYDOWN(A, 1)` восстановления свойств невозрастающей кучи: узел 5 и 9 переставлены местами, после чего цикл завершают свою работу, так как $5 > 3$.

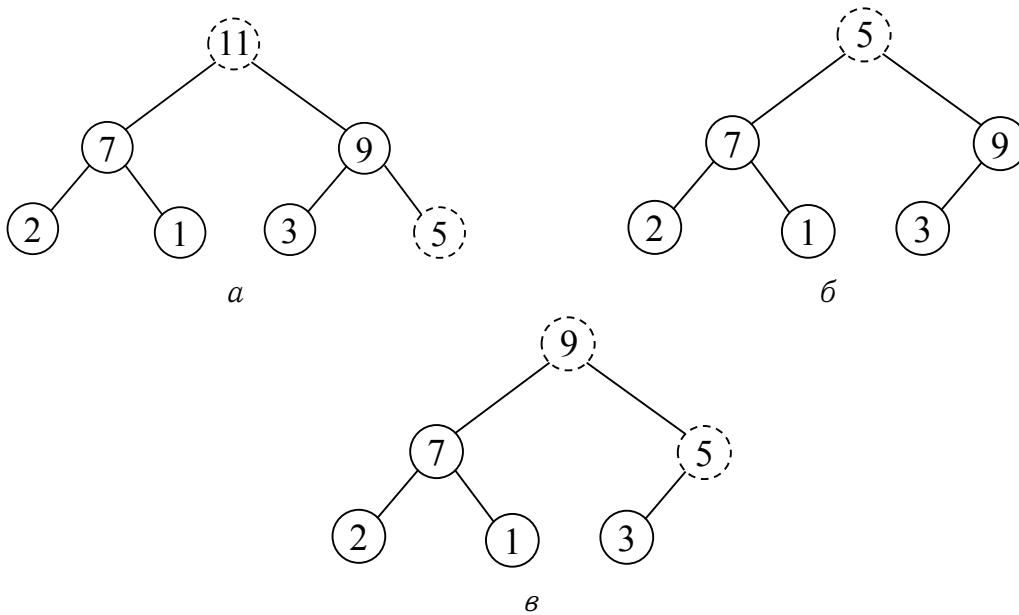


Рис. 14.4. Удаление ключа 11 с максимальным значением (max-heap):
 а – исходное состояние кучи;
 б – последний элемент 5 массива перемещен в корень дерева;
 в – результат обмена значениями узлов 5 и 9.

```

1 function DELETEMAX( $A[1..m]$ )
2   if  $n < 1$  then
3     return HeapEmpty
4   end if
5    $max = A[1]$ 
6    $A[1] = A[n]$ 
7    $n = n - 1$ 
8   HEAPIFYDOWN( $A, 1$ )
9   return  $max$ 
10 end function

11 function HEAPIFYDOWN( $A[1..m], i$ )
12   while  $i \leq n$  do
13      $left = \text{LEFT}(i)$ 
14      $right = \text{RIGHT}(i)$ 
15      $largest = i$ 
16     if  $left \leq n$  and  $A[left].key > A[largest].key$  then
17        $largest = left$ 
18     else if  $right \leq n$  and  $A[right].key > A[largest].key$  then
19        $largest = right$ 
20     end if
21     if  $largest \neq i$  then
22       SWAP( $A[i], A[largest]$ )
23        $i = largest$ 
24     else
25       break                                /* Завершаем проход по дереву */
26     end if
27   end while
28 end function

```

Увеличение ключа. Операция INCREASEKEY заменяет у узла, хранящегося в ячейке i , значение ключа на большее. Если переданное значение нового ключа key меньше текущего $A[i].key$, то функция возвращает специальное значение *HeapInvalidKey*.

После установки нового значения ключа следует восстановить свойства невозрастающей кучи. Новый ключ может быть больше ключа родительского элемента, поэтому мы вызываем функцию HEAPIFYUP для ячейки с номером i .

В худшем случае выполняется изменение приоритета листового узла и новый ключ превосходит по значению все имеющиеся ключи. В такой ситуации функция HEAPIFYUP выполняет порядка $O(\log n)$ обменов узлов.

Построение кучи. Если нам заранее задан массив $A[1..m]$, содержа-

```

1 function INCREASEKEY( $A[1..m]$ ,  $i$ ,  $key$ )
2     if  $A[i].key > key$  then
3         return  $HeapInvalidKey$  /* Новый ключ меньше текущего */
4     end if
5      $A[i].key = key$ 
6     HEAPIFYUP( $A, i$ )           /* Восстанавливаем свойства кучи */
7 end function

```

щий n ключей, то мы можем преобразовать его в невозрастающую бинарную кучу за время $O(n)$.

Любой массив чисел можно интерпретировать как завершенное бинарное дерево. Чтобы сформировать из него невозрастающую бинарную кучу требуется восстановить свойство распределения ключей. Для этого можно использовать процедуру HEAPIFYDOWN. При помощи нее мы расставляем ключи на корректные позиции.

В завершенном бинарном дереве элементы, хранящиеся в ячейках $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$, не имеют дочерних узлов (являются листьями). Поэтому восстановление свойств кучи целесообразно начинать с ячейки $\lfloor n/2 \rfloor$ и двигаться в направлении начала массива (корня дерева).

Ниже приведен псевдокод функции BUILDMAXHEAP, которая строит невозрастающую бинарную кучу (max-heap) из элементов заданного массива $A[1..m]$.

```

1 function BUILDMAXHEAP( $A[1..m]$ ,  $n$ )
2      $i = \lfloor n/2 \rfloor$ 
3     while  $i \geq 1$  do
4         HEAPIFYDOWN( $A, i$ )
5          $i = i - 1$ 
6     end while
7 end function

```

На рис. 14.5 приведен пример построения невозрастающей бинарной кучи для массива $(3, 7, 9, 6, 8, 4)$ из шести ключей. Восстановление свойств кучи начинается с ячейки $\lfloor 6/2 \rfloor = 3$. При вызове HEAPIFYDOWN($A, 3$) обмены не выполняются (рис. 14.5, б), так как ключ 9 больше 4. Вызов HEAPIFYDOWN($A, 2$) влечет за собой обмен узлов с ключами 7 и 8 (рис. 14.5, в). При вызове HEAPIFYDOWN($A, 1$) выполняется два обмена: сперва меняются местами узлы с ключами 3 и 9, а затем 3 и 4 (рис. 14.5, г).

Оценим вычислительную сложность функции BUILDMAXHEAP. В цикле осуществляется вызов процедуры HEAPIFYDOWN для узлов кучи, имеющих различные высоты $h \in \{1, 2, \dots, \lfloor \log_2 n \rfloor\}$. Известно, что в завершенном бинарном дереве присутствует не более $\lceil n/2^{h+1} \rceil$ узлов с высотой h . Для любого узла с высотой h функция HEAPIFYDOWN выполняет порядка

$O(h)$ операций обменов. Запишем суммарное число операций, выполняемых функцией `BUILDMAXHEAP`:

$$T(n) = \sum_{h=1}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right).$$

Последнюю сумму можно найти, используя формулу (16.6), в которой следует положить $a = 0, r = 1, q = 1/2$:

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} h \left(\frac{1}{2}\right)^h < \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1 - 1/2)^2} = 2.$$

Откуда следует

$$T(n) = O \left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = O(n).$$

Таким образом, мы показали, что время $T(n)$ построения бинарной кучи из заданных n ключей есть $O(n)$.

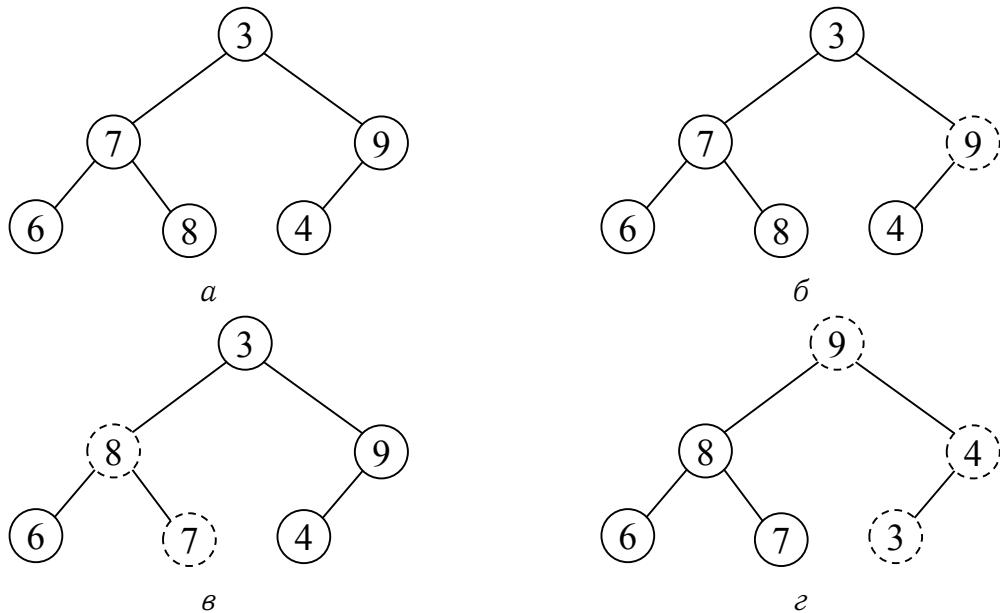


Рис. 14.5. Построение невозрастающей бинарной кучи (max-heap) из массива ключей (3, 7, 9, 6, 8, 4):

- α – завершенное бинарное дерево, соответствующее исходному массиву;
- β – результат вызова `HEAPIFYDOWN(A, 3)`;
- γ – результат вызова `HEAPIFYDOWN(A, 2)`;
- δ – результат вызова `HEAPIFYDOWN(A, 1)`.

14.3. Пирамидальная сортировка

Пирамидальная сортировка (*heapsort*) – это асимптотически оптимальная сортировка сравнением, которая использует бинарную кучу в процессе упорядочивания последовательности из n ключей. Алгоритм требует порядка $O(n \log n)$ операций сравнения ключей и не использует помимо сортируемого массива дополнительной памяти. Ниже приведен псевдокод алгоритма *HEAPSORT* для упорядочивания по неубыванию элементов заданного массива $A[1..n]$.

Работа алгоритма начинается с упорядочивания элементов массива $A[1..n]$ в виде невозрастающей бинарной кучи (пирамиды). Далее в цикле осуществляется перемещение максимального ключа из ячейки $A[1]$ в ячейку $A[i]$ – текущий конец кучи, $i = n, n-1, \dots, 2$. После чего свойства невозрастающей кучи восстанавливаются функцией *HEAPIFYDOWN*. Это необходимо сделать, так как в ячейку $A[1]$ записан элемент $A[i]$, который может не иметь максимального значения. Таким образом, с каждой итерацией цикла размер кучи уменьшается на 1, а очередной максимальный элемент занимает свое окончательное место в массиве.

Оценим вычислительную сложность алгоритма *HEAPSORT*. Построение невозрастающей бинарной кучи из элементов массива $A[1..n]$ выполняется за время $O(n)$. Функция *HEAPIFYDOWN* вызывается $n - 1$ раз и в худшем случае выполняет обмены всех узлов на пути от корня для листа. Число обменов в этом случае имеет порядок $O(\log n)$. Следовательно, вычислительная сложность алгоритма пирамидальной сортировки равна

$$T_{\text{HeapSort}} = O(n) + O((n - 1) \log n) = O(n \log n).$$

Алгоритм не обеспечивает устойчивости (non-stable sort), так как в процессе своей работы меняет относительный порядок следования ключей с одинаковыми значениями. Использует константное число дополнительных ячеек памяти, поэтому относится к классу алгоритмов сортировки на месте (*in-place*). На почти упорядоченных входных данных пирамидальная сортировка работает столько же времени, как и на случайных данных. В процессе выполнения операций над кучей осуществляются обращения к не смежным ячейкам массива, что негативно сказывается на работе кеш-памяти процессора.

Во многих случаях алгоритм является конкурентом сортировки слиянием и используется вместо нее, если имеются ограничения на объем используемой памяти (напомним, что сортировка слиянием требует $O(n)$ ячеек дополнительной памяти).

Алгоритм 14.1. Пирамидальная сортировка

```

1 function HEAP SORT( $A[1..n]$ ,  $n$ )
2     BUILDMAXHEAP( $A$ ,  $n$ )
3      $i = n$ 
4     while  $i \geq 2$  do
5         SWAP( $A[1]$ ,  $A[i]$ )
6          $i = i - 1$ 
7         HEAPIFYDOWN( $A$ , 1)
8     end while
9 end function

```

14.4. Упражнения

1. Реализуйте операции для неубывающей бинарной кучи (min-heap).
2. Предложите рекурсивную версию операции INSERT.
3. Можно ли динамически увеличить размер кучи – скопировать все ячейки в новый массив большего размера? Надо ли будет после этого обращаться к функциям восстановления свойств кучи?
4. Приведите пример невозрастающей бинарной кучи (max-heap), иллюстрирующий худший случай для операции DELETEMAX.
5. Оцените время поиска элемента по заданному ключу (приоритету) в бинарной куче.
6. Постройте пример худшего случая для пирамидальной сортировки.
7. Модифицируйте рассмотренный алгоритм пирамидальной сортировки для упорядочивания массива по невозрастанию.

15. Биномиальные кучи

Биномиальные кучи (binomial heap) реализуют АТД *очередь с приоритетом*, аналогично бинарным кучам (раздел 14), но в отличие от последних биномиальные кучи относятся к *эффективно сливаляемым кучам* (mergeable heap). Операция слияния в них выполняется за время $\Omega(\log n)$ в худшем случае. К основным операциям, определенным для биномиальных куч, относятся:

- добавление элемента;
- поиск и удаление узла с минимальным/максимальным приоритетом;
- понижение приоритета заданного элемента;
- слияние двух очередей в одну.

15.1. Структура биномиальной кучи

Биномиальная куча формируется на основе *биномиальных деревьев* (binomial tree) – рекурсивно определяемых деревьев высоты k , в которых:

- общее количество узлов равно 2^k ;
- количество узлов на уровне i определяется количеством сочетаний из k по i :

$$\binom{k}{i} = \frac{k!}{i!(k-i)!};$$

- корень имеет k дочерних узлов, каждый из которых является биномиальным деревом меньшей степени.

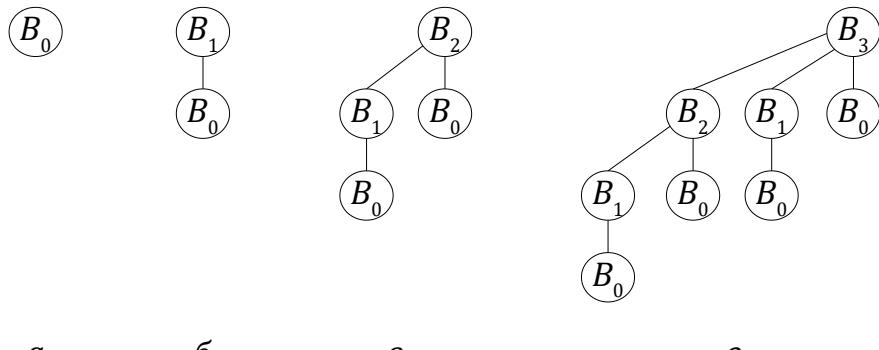


Рис. 15.1. Биномиальные деревья различных степеней:
α – дерево B_0 ; β – дерево B_1 ; γ – дерево B_2 ; δ – дерево B_3 .

На рис. 15.1 представлены биномиальные деревья различных степеней. Нетрудно заметить, что первым (крайним слева) дочерним узлом корня дерева B_k является дерево B_{k-1} , вторым – дерево B_{k-2} , и так далее, до крайнего правого потомка – дерева B_0 .

В биномиальном дереве с n вершинами максимальная степень узла равна $O(\log n)$.

Биномиальной кучей называется упорядоченное по возрастанию степеней множество биномиальных деревьев, каждое из которых удовлетворяет свойствам неубывающей или невозрастающей кучи (min-heap/max-heap). Корни деревьев, составляющих биномиальную кучу, объединены в односвязный список – *список корней* (root list). В куче имеется не более одного экземпляра биномиального дерева каждой степени.

Биномиальная куча, содержащая n узлов, состоит не более, чем из $\lfloor \log n \rfloor + 1$ деревьев. Порядок и количество биномиальных деревьев, составляющих кучу, однозначно определяется количеством n узлов в ней: каждый разряд в двоичной записи числа n показывает наличие или отсутствие в куче дерева соответствующей степени.

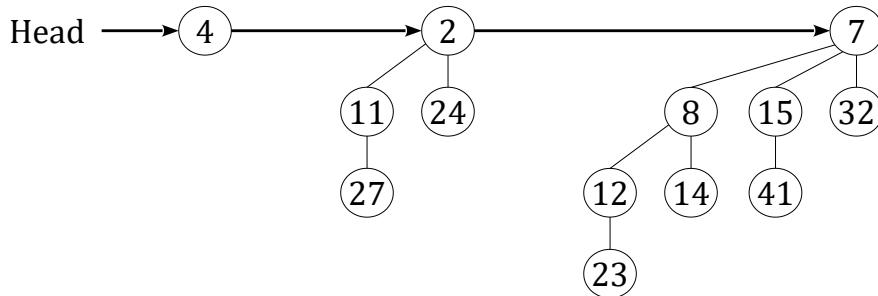


Рис. 15.2. Биномиальная куча из 13 узлов
(список корней состоит из деревьев B_0 , B_2 и B_3).

На рис. 15.2 представлена биномиальная куча из $13_{10} = 1101_2$ узлов. Двоичная запись числа 13 содержит единицы в разрядах 0, 2 и 3, следовательно, куча состоит из деревьев B_0 , B_2 и B_3 .

Рассмотрим реализацию основных операций биномиальной кучи, деревья которой являются неубывающими кучами (min-heap). Операции кучи из невозрастающих биномиальных деревьев реализуются аналогично. Будем считать, что каждый из узлов биномиальной кучи содержит следующие поля:

- *key* – приоритет узла (также вес, ключ);
- *value* – данные;
- *degree* – количество дочерних узлов (степень узла);
- *parent* – указатель на родительский узел;
- *child* – указатель на крайний левый дочерний узел;
- *sibling* – указатель на правый сестринский узел.

15.2. Поиск минимального элемента

Так как в корневом узле каждого биномиального дерева, составляющего кучу, хранится его минимальный ключ, для поиска минимального элемента в биномиальной куче достаточно пройти по списку из $\lfloor \log n \rfloor + 1$ корней.

```

1 function BINOMIALHEAPMIN(heap)
2     x = heap
3     min.key =  $\infty$ 
4     while x ≠ null do
5         if x.key < min.key then
6             min = x
7         end if
8         x = x.sibling
9     end while
10    return min
11 end function
```

В биномиальной куче возможна реализация поиска минимального ключа за время $O(1)$, для этого необходимо поддерживать указатель на корень дерева, в котором находится экстремальный ключ.

15.3. Слияние биномиальных куч

Для слияния (*merge*, *union*) двух биномиальных куч H_1 и H_2 в новую кучу H необходимо слить списки корней H_1 и H_2 в один упорядоченный список, а затем выполнить восстановление свойств биномиальной кучи H .

Так как списки корней H_1 и H_2 упорядочены по возрастанию степеней узлов, их объединение возможно выполнить аналогично слиянию упорядоченных массивов в алгоритме сортировки слиянием (*MergeSort*). После слияния списков корней известно, что в куче H имеется не более двух корней с одинаковой степенью и они соседствуют.

После слияния списков корней в результирующей куче могут находиться два дерева одной степени, поэтому необходимо обойти новый список корней, выполняя восстановление свойств биномиальной кучи. Во время обхода списка корней поддерживается указатель x на текущий узел, а также указатели *prevx* и *nextx* на предыдущий и следующий за x элементы списка. Рассмотрим 4 возможных случая, возникающих при слиянии биномиальных куч.

Случай 1. Узел x является корнем дерева B_k , узел *nextx* – корнем дерева B_l , где $l > k$. Здесь свойства биномиальной кучи не нарушаются, поэтому перемещаем указатели далее по списку корней.

```

1 function BINOMIALHEAPUNION(heap1, heap2)
2   heap = BINOMIALHEAPLISTMERGE(heap1, heap2)
3   prevx = null
4   x = heap
5   nextx = x.sibling
6   while nextx ≠ x.sibling do
7     if x.degree ≠ nextx.degree or nextx.sibling ≠ null and
8       nextx.sibling.degree = x.degree then
9         prevx = x                                /* Случай 1 и 2 */
10        x = nextx
11     else if x.key ≤ nextx.key then
12       x.sibling = nextx.sibling                /* Случай 3 */
13       BINOMIALTREELINK(nextx, x)
14     else                                         /* Случай 4 */
15       if prevx = null then
16         heap = nextx
17       else
18         prevx.sibling = nextx
19       end if
20       BINOMIALTREELINK(x, nextx)
21       x = nextx
22     end if
23     nextx = x.sibling
24   end while
25   return heap
26 end function

```

Случай 2. Узлы x , $nextx$ и следующий за ним $nextx.sibling$ имеют одинаковую степень k . Такой случай сводится к одному из случаев 3–4, где деревья с корнями в узлах $nextx$ и $nextx.sibling$ объединяются в дерево B_{k+1} . Для перехода к одному из следующих случаев указатели перемещаются далее по списку корней.

Случай 3. Узлы x и $nextx$ имеют одинаковую степень k , отличную от степени следующего за ними узла $nextx.sibling$. Приоритет вершины x не выше приоритета вершины $nextx$ ($x.key \leq nextx.key$). Для разрешения случая 3 необходимо связать деревья x и $nextx$ – сделать $nextx$ левым дочерним узлом x .

Случай 4. Узлы x и $nextx$ имеют одинаковую степень k , отличную от степени узла $nextx.sibling$. Приоритет вершины x выше приоритета вершины $nextx$ ($x.key > nextx.key$). В этом случае также образуется дерево B_{k+1} : элемент x становится левым дочерним узлом $nextx$.

Так как длина списка корней не превышает $\log n_1 + \log n_2 + 2 = O(\log n)$,

внешний цикл функции BINOMIALHEAPUNION выполняется не более $O(\log n)$ раз. На каждой итерации цикла происходит либо перемещение указателя по списку корней вправо на одну позицию, либо удаление одного узла, что требует времени $O(1)$. Таким образом, вычислительная сложность операции слияния двух биномиальных куч составляет $O(\log n)$.

На рис. 15.3 приведен пример слияния двух биномиальных куч.

```

1 function BINOMIALHEAPLISTMERGE(heap1, heap2)
2     heap = null
3     while heap1 ≠ null and heap2 ≠ null do
4         if heap1.degreee ≤ heap2.degreee then
5             LINKEDLISTADDEND(heap, heap1)
6             heap1 = heap1.next
7         else
8             LINKEDLISTADDEND(heap, heap2)
9             heap2 = heap2.next
10        end if
11    end while
12    while heap1 ≠ null do
13        LINKEDLISTADDEND(heap, heap1)
14        heap1 = heap1.next
15    end while
16    while heap2 ≠ null do
17        LINKEDLISTADDEND(heap, heap2)
18        heap2 = heap2.next
19    end while
20    return heap
21 end function

22 function BINOMIALHEAPLINK(child, parent)
23     child.parent = parent
24     child.sibling = parent.child
25     parent.child = child
26     parent.degreee = parent.degreee + 1
27 end function
```

15.4. Вставка узла

Для добавления в биномиальную кучу нового элемента создается куча из одного узла x – биномиального дерева B_0 , после чего происходит слияние исходной кучи с кучей из узла x .

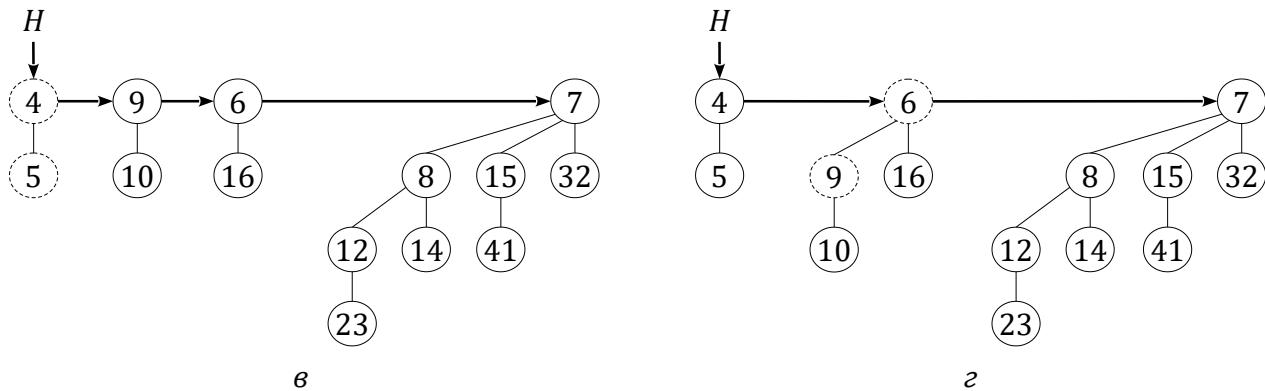
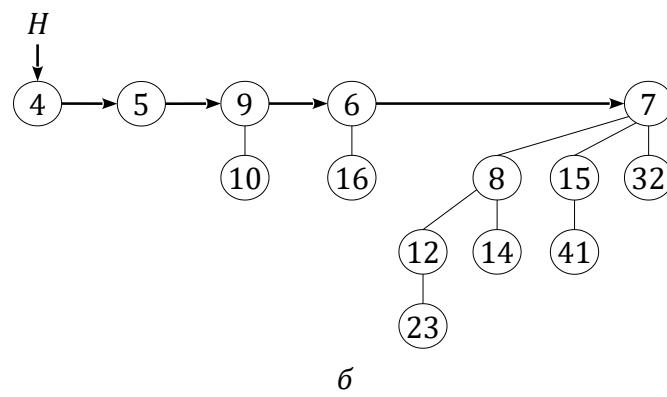
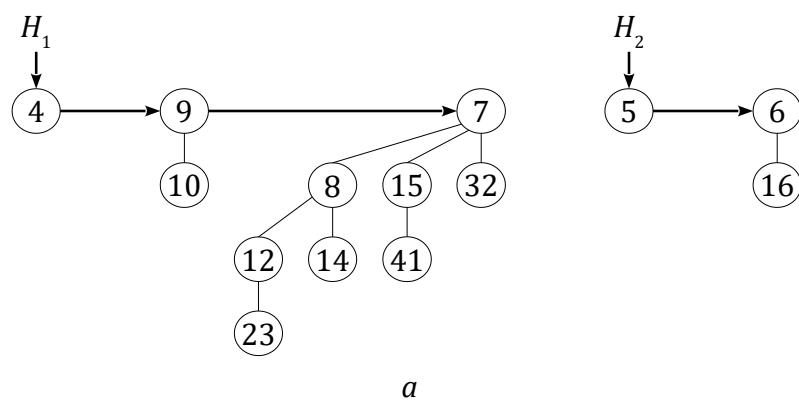


Рис. 15.3. Слияние биномиальных куч: *a* – исходное состояние куч; *б* – слияние списков корней; *в* – слияние двух деревьев B_0 в дерево B_1 ; *г* – слияние двух деревьев B_1 в дерево B_2 .

```

1 function BINOMIALHEAPINSERT(heap, key, value)
2   x.key = key
3   x.value = value
4   x.degreee = 0
5   x.parent = null
6   x.child = null
7   x.sibling = null
8   BINOMIALHEAPUNION(heap, x)
9 end function

```

15.5. Удаление минимального узла

Для удаления минимального узла необходимо отыскать корень x с минимальным ключом и удалить его из списка корней кучи. Порядок следования дочерних узлов корня x меняется на обратный, поле $parent$ каждого из них устанавливается в $NULL$. Дочерние узлы удаленного корня x образуют новую биномиальную кучу, которая сливается с исходной.

На рис. 15.4 показан пример удаления из биномиальной кучи ключа 1 с минимальным приоритетом.

```

1 function BINOMIALHEAPDELETEMIN(heap)
2      $x = heap$ 
3      $xmin.key = \infty$ 
4      $prev = null$ 
5     while  $x \neq null$  do
6         if  $x.key < xmin.key$  then
7              $xmin = x$ 
8              $prevmin = prev$ 
9         end if
10         $prev = x$ 
11         $x = x.sibling$ 
12    end while
13    if  $prevmin \neq null$  then
14         $prevmin.sibling = xmin.sibling$ 
15    else
16         $heap = xmin.sibling$ 
17    end if
18     $child = xmin.child$ 
19     $prev = null$ 
20    while  $child \neq null$  do
21         $sibling = child.sibling$ 
22         $child.sibling = prev$ 
23         $prev = child$ 
24         $child = sibling$ 
25    end while
26    BINOMIALHEAPUNION(heap, prev)
27 end function
```

15.6. Понижение приоритета узла

Для понижения приоритета узла необходимо получить указатель на заданный узел x и изменить его ключ, при этом новое значение приори-

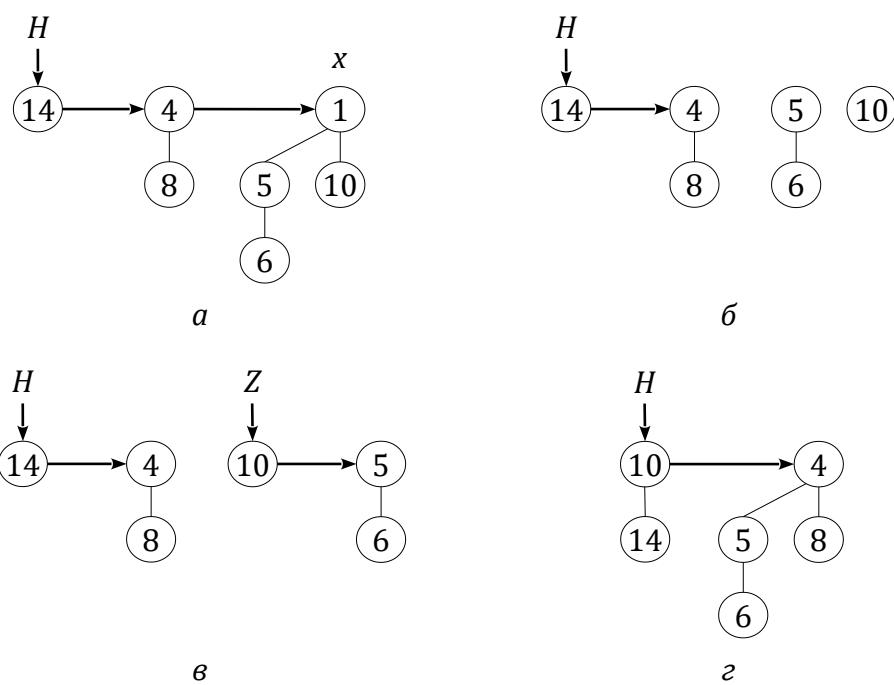


Рис. 15.4. Удаление ключа 1 с минимальным приоритетом: *а* – исходное состояние кучи; *б* – разрыв связи с минимальным корнем *x* и его удаление; *в* – формирование кучи *Z* из дочерних элементов *x*; *г* – слияние куч *H* и *Z*.

тета не должно быть выше прежнего ($newkey \leq x.key$). Затем, если узел *x* с новым приоритетом нарушает свойства неубывающей кучи, в цикле происходит его подъем по текущему биномиальному дереву. Выполнение цикла завершается, либо когда узел *x* достигает корня дерева, либо когда приоритет *x* не ниже приоритета его родителя.

```

1 function BINOMIALHEAPDECREASEKEY(heap, x, newkey)
2   if x.key < newkey then
3     return
4   end if
5   x.key = newkey
6   y = x
7   z = y.parent
8   while z ≠ null and y.key < z.key do
9     temp = y.key
10    y.key = z.key
11    z.key = temp
12    y = z
13    z = y.parent
14  end while
15 end function
```

```
1 function BINOMIALHEAPDELETE(heap, x)
2     BINOMIALHEAPDECREASEKEY(heap, x,  $-\infty$ )
3     BINOMIALHEAPDELETEMIN(heap)
4 end function
```

На основе операций удаления минимального узла и понижения приоритета возможна реализация операции BINOMIALHEAPDELETE удаления произвольного узла из биномиальной кучи. Приоритет заданного узла понижается до минимально возможного, после чего он удаляется из кучи как минимальный элемент.

15.7. Упражнения

1. Реализуйте операции для биномиальной кучи, деревья которой являются невозрастающими кучами (max-heap).
2. Определите количество и степени биномиальных деревьев в куче, состоящей из 87 узлов.
3. Добавьте в биномиальную кучу с рис. 15.2 ключи 1, 5, 6.
4. Из кучи, полученной в задании 3, удалите элемент с минимальным приоритетом.
5. Оцените время выполнения операции BINOMIALHEAPDELETE.
6. Реализуйте алгоритм вставки узла в биномиальную кучу без использования операции BINOMIALHEAPUNION.
7. Предложите алгоритм поиска элемента по заданному ключу в биномиальной куче.

16. Приложения

16.1. Функции округления

Округление вещественного числа x до меньшего ближайшего целого числа обозначается как $\lfloor x \rfloor$, или $\text{floor}(x)$ – *пол*. Также эту функцию называют *антье* (от фр. entier) – целая часть вещественного числа

$$x - 1 < \lfloor x \rfloor \leq x.$$

Примеры использования функции пол:

- $\lfloor 1.3 \rfloor = 1;$
- $\lfloor 1.7 \rfloor = 1;$
- $\lfloor -3.7 \rfloor = -4;$
- $\lfloor -4 \rfloor = -4.$

Округление вещественного числа x до большего ближайшего целого числа обозначается как $\lceil x \rceil$, или $\text{ceil}(x)$ – *потолок*

$$x \leq \lceil x \rceil < x + 1.$$

Примеры использования функции потолок:

- $\lceil 1.3 \rceil = 2;$
- $\lceil 1.7 \rceil = 2;$
- $\lceil -3.7 \rceil = -3;$
- $\lceil -4 \rceil = -4.$

Функции *пол* и *потолок* введены в 1962 г. К. Айверсоном [3]. Для них справедливы следующие соотношения:

$$\lfloor x \rfloor \leq x \leq \lceil x \rceil.$$

Целочисленное слагаемое a можно вносить и выносить за скобки функций пол/потолок

$$\lfloor x + a \rfloor = \lfloor x \rfloor + a, \quad \lceil x + a \rceil = \lceil x \rceil + a.$$

16.2. Прогрессии

Арифметическая прогрессия – это бесконечная последовательность чисел вида

$$a_1, a_1 + d, a_1 + 2d, \dots, a_1 + (n - 1)d, \dots,$$

где d – разность (шаг) прогрессии; при $d > 0$ прогрессия является возрастающей, а при $d < 0$ – убывающей. Справедливы следующие равенства:

$$a_n = a_{n-1} + d,$$

$$a_n = a_1 + d(n - 1).$$

Сумма S_n первых n членов арифметической прогрессии может быть найдена по следующей формуле:

$$S_n = \frac{a_1 + a_n}{2}n = \frac{2a_1 + d(n - 1)}{2}n. \quad (16.1)$$

Геометрическая прогрессия – это бесконечная последовательность чисел вида

$$b_1, b_1q, b_1q^2, \dots, b_1q^{n-1}, \dots,$$

где $q \neq 0$ – знаменатель прогрессии, $b_1 \neq 0$. Если $|q| > 1$, то прогрессия является возрастающей последовательностью; при $|q| < 1$ прогрессия называется убывающей. Справедливо следующее равенство:

$$b_n = b_1q^{n-1}.$$

Сумма S_n первых n членов геометрической прогрессии ($|q| \neq 1$)

$$S_n = \frac{b_1 - b_nq}{1 - q} = \frac{b_1(1 - q^n)}{1 - q}. \quad (16.2)$$

Сумма S членов бесконечной убывающей геометрической прогрессии при $|q| < 1$, $n \rightarrow \infty$ равна

$$S \rightarrow \frac{b_1}{1 - q}.$$

16.3. Некоторые числовые ряды

1. Сумма n первых натуральных чисел

$$\sum_{k=1}^n k = 1 + 2 + 3 + \dots + n = \frac{n(n + 1)}{2}. \quad (16.3)$$

2. Сумма n первых четных натуральных чисел

$$\sum_{k=1}^n 2k = 2 + 4 + 6 + \dots + 2n = n(n+1). \quad (16.4)$$

3. Сумма n первых нечетных натуральных чисел

$$\sum_{k=1}^n 2k - 1 = 1 + 3 + 5 + \dots + (2n-1) = n^2. \quad (16.5)$$

4. При $|x| < 1$

$$\sum_{k=0}^{\infty} (a + kr)q^k = \frac{a}{1-q} + \frac{rq}{(1-q)^2}. \quad (16.6)$$

16.4. Логарифмы

Логарифм числа b по основанию a – это показатель степени, в которую надо возвести основание a , чтобы получить число b . Вещественный логарифм $\log_a b$ имеет смысл при $a > 0$, $a \neq 1$ и $b > 0$.

Натуральный логарифм (основание – число Эйлера $e \approx 2.72$)

$$\ln n = \log_e n.$$

Десятичный логарифм (основание – число 10)

$$\lg n = \log_{10} n.$$

Основное логарифмическое тождество

$$a^{\log_a b} = b.$$

Свойства логарифмов

$$\log_a(xy) = \log_a x + \log_a y,$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y,$$

$$\log_a x^p = p \log_a x,$$

$$a^{\log_b c} = c^{\log_b a}.$$

Замена основания логарифма – логарифм $\log_a b$, по основанию a можно

преобразовать в логарифм по другому основанию c

$$\log_a b = \frac{1}{\log_c a} \log_c b.$$

16.5. Факториалы

Факториал (factorial) целого числа n – это произведение натуральных чисел от 1 до n включительно

$$n! = 1 \cdot 2 \cdot \dots \cdot n,$$

$$n! = n \cdot (n - 1)!,$$

$$0! = 1.$$

Асимптотическая формула Стирлинга для приближенного вычисления факториала целого числа n

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + O \left(\frac{1}{n} \right) \right),$$

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n.$$

16.6. Числа Фибоначчи

Числа Фибоначчи (Fibonacci numbers) – это элементы последовательности целых чисел, в которой каждое следующее число равно сумме двух предыдущих чисел

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots$$

Формально последовательность Фибоначчи задается линейным рекуррентным соотношением

$$F_n = \begin{cases} 0, & \text{если } n = 0, \\ 1, & \text{если } n = 1, \\ F_{n-1} + F_{n-2}, & \text{если } n \geq 2. \end{cases}$$

Для нахождения общего члена F_n последовательности можно использовать формулу Бине. Она выражает в явном виде значение F_n как функ-

цию от n

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}, \quad \varphi = \frac{1 + \sqrt{5}}{2} \approx 1.61803,$$

где φ – это *отношение золотого сечения* (golden ratio).

Из формулы Бине следует [3]

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor.$$

В частности, при $n \rightarrow \infty$ справедливо

$$F_n \approx \frac{\varphi^n}{\sqrt{5}}.$$

Предметный указатель

- Алгоритм, 9
 - детерминированный, 10
 - дискретный, 10
 - худший случай, 14
 - количество операций, 12
 - конечный, 10
 - корректный, 9
 - лучший случай, 14
 - массовый, 10
 - показатели эффективности, 11
 - рекурсивный, 35
 - средний случай, 14
 - свойства, 10
 - вычисления факториала, 29
 - offline, 76
 - online, 50, 76
- Асимптотические обозначения, 19
 - O -обозначение, 20
 - Ω -обозначение, 22
 - Θ -обозначение, 22
 - ω -обозначение, 24
 - o -обозначение, 23
- Асимптотический анализ, 19
 - основные этапы, 27
- Ассоциативный массив, 73
- Бинарная куча, 179
- Бинарное дерево поиска, 125
- Биномиальная куча, 189
- Дек, 72
- Дерево
 - AVL, 139
 - бинарное, 120
 - биномиальное, 189
 - корневое, 119
 - красно-черное, 140
- обход, 123
 - обход в глубину, 123
 - обход в ширину, 123
 - поиска, 125
 - полное бинарное, 120
 - префиксное, 153
 - рекурсивных вызовов, 35, 40
 - решений, 51
 - сбалансированное, 139
 - совершенное бинарное, 120
 - завершенное бинарное, 120
 - AA-дерево, 139
 - В-дерево, 139
- Двойное хеширование, 171
- Эффективность
 - пространственная, 11
 - временная, 11
- Глубина узла, 120
- Граф, 74
- Хеш-функция
 - метод деления, 173
 - метод умножения, 173
 - требования, 172
- Хеш-таблица, 163
 - коэффициент заполненности, 165
- Хеширование
 - двойное, 171
 - совершенное, 175
 - универсальное, 174
- Исполнитель, 9
- Исследование, 168
 - квадратичное, 171
 - линейное, 170
- Коллизия, 163
- Кольцевой буфер, 112

- Красно-черное дерево, 140
- Куча
- бинарная, 179
 - биномиальная, 189
 - эффективно сливаемая, 189
 - неубывающая, 180
 - невозрастающая, 180
- Лист, 119
- Массив
- динамический, 88
- Метод
- «разделяй и властвуй», 37
 - декомпозиции, 37
 - основной, 42
 - разрешения коллизий, 163
- Метод цепочек, 163
- Множество, 73
- упорядоченное, 73
- Мульти множество, 74
- Обход дерева
- инфиксный, 123
 - обратный, 124
 - постфиксный, 124
 - префиксный, 123
 - прямой, 123
 - симметричный, 123
- Очередь, 71, 109
- Очередь с приоритетом, 74
- Опорный элемент, 53
- Открытая адресация, 167
- Пирамидальная сортировка, 187
- Поиск, 63
- бинарный, 63
 - линейный, 63
- Порядок роста функции, 17
- сравнение через предел, 25
- Поворот дерева, 144
- Префиксное дерево, 153
- Программа, 9
- Рекурсивная функция, 35
- Рекурсия, 35
- древовидная, 36
 - косвенная, 35
- линейная, 35
- нелинейная, 36
- прямая, 35
- Сборщик мусора, 116
- Скорость роста функции, 17
- Словарь, 73
- со строковым ключом, 153
- Сортировка, 45
- асимптотически оптимальная, 53
 - быстрая, 53
 - целочисленная, 46
 - на месте, 47
 - пирамидальная, 58, 187
 - подсчетом, 58
 - слиянием, 37, 57
 - сравнением, 45
 - устойчивая, 46
 - выбором, 30, 50
 - внешняя, 47
 - внутренняя, 47
 - вставкой, 47
- Список, 71
- двусвязный, 89, 95
 - кольцевой, 89
 - односвязный, 89, 90
 - связный, 89
- Стек, 71, 103
- Степень узла, 119
- Структура данных, 68
- интерфейс, 68
- Таблица с прямой адресацией, 161
- Теорема
- основная, 42
- Тип данных, 67
- абстрактный, 69
 - ассоциативный массив, 73
 - базовый, 67
 - дек, 72
 - граф, 74
 - линейный, 70
 - множество, 73
 - нелинейный, 72
 - очередь, 71, 109

очередь с приоритетом, 74
пользовательский, 68
примитивный, 67
с последовательным доступом, 70
с прямым доступом, 70
составной, 67
список, 71, 83
стек, 71, 103

Уравнение

декомпозиции, 42
рекуррентное, 40

Узел

потомок, 119
предок, 119
родительский, 119
сестринский, 119
внешний, 119
внутренний, 119

Высота дерева, 120

Высота узла, 119

Задача

этапы решения, 9
постановка, 9

Master method, 42

RAM-машина, 12

Литература

- [1] Алгоритмы: построение и анализ. 3-е изд. / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — М. : Вильямс, 2013.
- [2] Левитин А.В. Алгоритмы: введение в разработку и анализ. — М. : Вильямс, 2006.
- [3] Грэхем Р., Кнут Д.Э., Паташник О. Конкретная математика. Математические основы информатики. — М. : Вильямс, 2010.
- [4] Ахо А.В., Хопкрофт Д., Ульман Д.Д. Структуры данных и алгоритмы. — М. : Вильямс, 2001.
- [5] Кнут Д.Э. Искусство программирования. Том 3. Сортировка и поиск. — М. : Вильямс, 2012.
- [6] Адельсон-Вельский Г.М., Ландис Е.М. Один алгоритм организации информации // Доклады Академии наук СССР. — 1962. — Т. 146, № 2. — С. 263–266.
- [7] Bayer R. Symmetric binary B-Trees: Data structure and maintenance algorithms // Acta Informatica. — 1972. — Vol. 1, no. 4. — P. 290–306.
- [8] Guibas L., Sedgewick R. A dichromatic framework for balanced trees // Proc. of the 19th Annual Symposium on Foundations of Computer Science. — 1978. — P. 8–21.
- [9] Bayer R., McCreight E. Organization and maintenance of large ordered indices // Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories. — 1970.
- [10] Andersson A. Balanced search trees made simple // Workshop on Algorithms and Data Structures. — 1993. — P. 60–71.
- [11] Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. — К. : Диа Софт, 2001.
- [12] Керниган Б.В., Р. Пайк. Практика программирования. — СПб. : Невский Диалект, 2001.

- [13] Кнут Д.Э. Искусство программирования. Том 1. Основные алгоритмы. — М. : Вильямс, 2010.
- [14] Кнут Д.Э. Искусство программирования. Том 2. Получисленные алгоритмы. — М. : Вильямс, 2011.
- [15] Вирт Н. Алгоритмы и структуры данных. — М. : ДМК, 2010.
- [16] Скиена С.С. Алгоритмы. Руководство по разработке. 2-е изд. — СПб. : БХВ, 2011.
- [17] Макконнелл Дж. Основы современных алгоритмов. 2-е изд. — М. : Техносфера, 2004.
- [18] Миллер Р., Боксер Л. Последовательные и параллельные алгоритмы: общий подход. — М. : БИНОМ, 2006.
- [19] Бентли Дж. Жемчужины программирования. — СПб. : Питер, 2002.
- [20] Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы. — М. : МЦНМО, 2014.
- [21] Кормен Т.Х. Алгоритмы: вводный курс. — М. : Вильямс, 2014.

Учебное пособие

Курносов Михаил Георгиевич

Берлизов Даниил Михайлович

АЛГОРИТМЫ И СТРУКТУРЫ ОБРАБОТКИ ИНФОРМАЦИИ

Подписано в печать 05.04.2019. Формат $70 \times 100^1/16$.

Бумага офсетная. Печать цифровая.

Усл. печ. л. 17,14.

Тираж 300. Заказ 0327.

Отпечатано в ООО «Параллель».
630090, Новосибирск, Весенний пр., 4. Тел. (383) 330-26-98.