

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

О.Б. Фофанов

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

*Рекомендовано в качестве учебного пособия
Редакционно-издательским советом
Томского политехнического университета*

Издательство
Томского политехнического университета
2014

УДК 004.421(075.8)

ББК 32.97я73

Ф81

Фофанов О.Б.

Ф81

Алгоритмы и структуры данных: учебное пособие / О.Б. Фофанов; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2014. – 126 с.

В пособии кратко изложены базовые понятия теории алгоритмов, методы оценки временной и пространственной сложности. Подробно рассмотрена концепция абстрактных типов данных, основные структуры, реализующие абстрактные типы данных, их линейные и динамические представления. Описаны алгоритмы сортировки, поиска и модификации данных в различных структурах данных.

Предназначено для студентов, обучающихся по направлениям «Программная инженерия», «Информатика и вычислительная техника», «Прикладная информатика».

УДК 004.421(075.8)

ББК 32.97я73

Рецензенты

Доктор технических наук, доцент
заведующий кафедрой БИС ТУСУРа

Р.В. Мещеряков

Кандидат технических наук, доцент ТУСУРа

В.П. Коцубинский

© ФГАОУ ВО НИ ТПУ, 2014

© Фофанов О.Б., 2014

© Оформление. Издательство Томского
политехнического университета, 2014

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ	5
1. ВВЕДЕНИЕ В АЛГОРИТМЫ	6
1.1. Свойства алгоритмов	6
1.2. Анализ алгоритмов	10
1.2.1. Псевдокод.....	12
1.2.2. Простейшие операции.....	15
1.2.3. Анализ средних и худших показателей.....	17
1.2.4. Асимптотическая нотация	18
2. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ (АТД)	23
2.1. АТД векторы, списки и последовательности	26
2.1.1. АТД вектор	26
2.1.2. АТД список	30
2.1.3. АТД стек.....	35
2.1.4. АТД очередь	39
2.1.5. АТД дек.....	43
3. ПОИСК	46
3.1. Наивный метод.....	47
3.2. Алгоритм Кнута-Морриса-Прата	49
3.3. Алгоритм Бойера-Мура	51
3.4. Алгоритм Рабина-Карпа.....	55
4. СОРТИРОВКА	60
4.1. Сортировка подсчетом	62
4.2. Сортировка включением	64
4.3. Сортировка Шелла	64
4.4. Сортировка извлечением	66
4.5. Пирамидальная сортировка.....	67
4.6. Обменные сортировки.....	71
4.7. Быстрая сортировка	73
4.8. Сортировка слиянием	75
5. ДЕРЕВЬЯ	78
5.1. Прохождение бинарных деревьев	79
5.2. Бинарные деревья поиска	82
5.2.1. Поиск заданного ключа	84
5.2.2. Поиск минимума и максимума.....	85
5.2.3. Предшествующий и последующий элементы	86
5.2.4. Вставка и удаление	87
5.3. Сбалансированные деревья.....	90

5.4. Сильноветвящиеся деревья	97
5.4.1. Определение В-деревьев	98
5.4.2. Поиск в В-дереве	101
5.4.3. Создание пустого В-дерева	102
5.4.4. Вставка ключа в В-дерево	102
5.4.5. Удаление ключа из В-дерева	107
6. ХЕШИРОВАНИЕ ДАННЫХ	109
6.1. Таблицы с прямой адресацией	110
6.2. Хеш-таблицы	111
6.3. Хеш-функции	112
6.4. Методы разрешения коллизий	116
СПИСОК ЛИТЕРАТУРЫ	121

ПРЕДИСЛОВИЕ

В пособии кратко изложены базовые понятия теории алгоритмов, методы оценки временной и пространственной сложности. Подробно рассмотрена концепция абстрактных типов данных, основные структуры, реализующие абстрактные типы данных, их линейные и динамические представления. Описаны базовые алгоритмы сортировки, классические алгоритмы поиска контекста в объемных текстах. В разделе, посвященном деревьям рассмотрены в первую очередь различные типы сбалансированных структур и алгоритмы поиска, вставки, удаления и модификации данных в древовидных структурах. В последнем разделе рассматриваются принципы хеширования данных и способы разрешения коллизий, возникающих в хеш-таблицах. Данное пособие предназначено для студентов бакалавриата, обучающихся по направлениям «Программная инженерия», «Информатика и вычислительная техника» и «Прикладная информатика».

1. ВВЕДЕНИЕ В АЛГОРИТМЫ

Понятие *алгоритм* [1] является основным для всей области компьютерного программирования, поэтому начать мы должны с тщательного анализа этого термина. Слово "алгоритм" (algorithm) уже само по себе представляет большой интерес. На первый взгляд может показаться, будто кто-то собирался написать слово "логарифм" (logarithm), но случайно переставил первые четыре буквы. Этого слова еще не было в издании словаря *Webster's New World Dictionary*, вышедшем в 1957 году. Мы находим там только устаревшую форму "algorism" – старинное слово, которое означает "выполнение арифметических действий с помощью арабских цифр". В средние века абакисты считали на абаках (счетных досках), а алгоритмики использовали "algorism". Наконец историки математики обнаружили истинное происхождение слова "algorism": оно берет начало от имени автора знаменитого персидского учебника по математике – Абу Абд Аллаха Мухаммеда ибн Муса аль-Хорезми (ок. 825 г.), означающего буквально "Отец Абдуллы, Мухаммед, сын Мусы, уроженец Хорезма". Аральское море в Центральной Азии когда-то называлось озером Хорезм, и район Хорезма (Khwarizm) расположен в бассейне реки Амударья южнее этого моря. Аль-Хорезми написал знаменитую книгу *Китаб аль-джебр валь-мукабала* – "Книга о восстановлении и противопоставлении". От названия этой книги, которая была посвящена решению линейных и квадратных уравнений, произошло еще одно слово – "алгебра".

К 1950 году слово "алгоритм" чаще всего ассоциировалось с алгоритмом Евклида, который представляет собой процесс нахождения наибольшего общего делителя двух чисел. Данный алгоритм был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.) и является наиболее цитируемым при рассмотрении введения в алгоритмизацию и программирование.

Современное значение слова "алгоритм" во многом аналогично таким понятиям, как рецепт, процесс, метод, способ, процедура, программа, но все-таки слово "algorithm" имеет дополнительный смысловой оттенок.

1.1. Свойства алгоритмов

Алгоритм – это не просто набор конечного числа правил, задающих последовательность выполнения операций для решения задачи определенного типа. Помимо этого, он имеет пять важных особенностей [1].

1) **Конечность.** Алгоритм всегда должен заканчиваться после выполнения конечного числа шагов. Количество шагов может быть сколь угодно большим; выбор слишком больших значений m и n в алгоритме Евклида приведет к тому, что некоторые шаги будут выполняться более миллиона раз (неважно какая модификация алгоритма будет использована, циклическая или рекурсивная).

Процедура, обладающая всеми характеристиками алгоритма, за исключением, возможно, конечности, называется методом вычислений. Евклид предложил не только алгоритм нахождения наибольшего общего делителя, но и аналогичное ему геометрическое построение "наибольшей общей меры" длин двух отрезков прямой; это уже метод вычислений, выполнение которого не заканчивается, если заданные длины оказываются несоизмеримыми

2) **Определенность.** Каждый шаг алгоритма должен быть точно определен. Действия, которые нужно выполнить, должны быть строго и недвусмысленно определены для каждого возможного случая. На практике алгоритмы могут описываться и на обычном языке, и на формализованных псевдоязыках так и на языках программирования. Метод вычислений, выраженный на языке программирования, называется программой.

3) **Ввод.** Алгоритм имеет некоторое (возможно, равное нулю) число входных данных, т. е. величин, которые задаются до начала его работы или определяются динамически во время его работы. Эти входные данные берутся из определенного набора объектов. Например, в алгоритме есть два входных значения, а именно m и n , которые принадлежат множеству целых положительных чисел.

4) **Вывод.** У алгоритма есть одно или несколько выходных данных, т. е. величин, имеющих вполне определенную связь с входными данными. У алгоритма Евклида имеется только одно выходное значение, а именно – наибольший общий делитель двух входных значений.

5) **Эффективность.** Алгоритм обычно считается эффективным, если все его операторы достаточно просты для того, чтобы их можно было точно выполнить в течение конечного промежутка времени с помощью карандаша и бумаги. В алгоритме Евклида используются только следующие операции: деление одного целого положительного числа на другое, сравнение с нулем и присвоение одной переменной значения другой. Эти операции являются эффективными, так как целые числа можно представить на бумаге с помощью конечного числа знаков и так как существует, по меньшей мере, один способ ("алгоритм деления") деления одного целого числа на другое. Но те же самые операции были

бы неэффективными, если бы они выполнялись над действительными числами, представляющими собой бесконечные десятичные дроби, либо над величинами, выражающими длины физических отрезков прямой, которые нельзя измерить абсолютно точно.

На практике нам нужны не просто алгоритмы, а хорошие алгоритмы в широком смысле этого слова. Одним из критериев качества алгоритма является время, необходимое для его выполнения; данную характеристику можно оценить по тому, сколько раз выполняется каждый шаг. Другими критериями являются адаптируемость алгоритма к различным компьютерам, его простота, изящество и т. д.

Часто решить одну и ту же проблему можно с помощью нескольких алгоритмов и нужно выбрать наилучший из них. Таким образом, возникает чрезвычайно интересная и крайне важная область анализа алгоритмов. Предмет этой области состоит в том, чтобы для заданного алгоритма определить рабочие характеристики. Как правило, это среднее число операций, необходимых для выполнения алгоритма – T_n , где n – параметр, характеризующий каким-то образом исходные данные, например число входных данных.

Для обозначения области подобных исследований используется термин анализ алгоритмов. Основная идея заключается в том, чтобы взять конкретный алгоритм и определить его количественные характеристики. Можно выяснять, является ли алгоритм оптимальным в некотором смысле. Теория алгоритмов – это совершенно другая область, в которой, в первую очередь, рассматриваются вопросы существования или не существования эффективных алгоритмов вычисления определенных величин.

Алгоритмы, как и аппаратное обеспечение компьютера, представляют собой технологию. Общая производительность системы настолько же зависит от эффективности алгоритма, как и от мощности применяющегося аппаратного обеспечения. В области разработки алгоритмов происходит такое же быстрое развитие, как и в других компьютерных технологиях.

Возникает вопрос, действительно ли так важны алгоритмы, работающие на современных компьютерах, если и так достигнуты выдающиеся успехи в других областях высоких технологий, таких как:

- аппаратное обеспечение с высокими тактовыми частотами, конвейерной обработкой и суперскалярной архитектурой;
- легкодоступные, интуитивно понятные графические интерфейсы (GUI);

- объектно-ориентированные системы;
- локальные и глобальные сети.

Ответ – да, безусловно. Несмотря на то, что иногда встречаются приложения, – которые не требуют алгоритмического наполнения (например, некоторые простые Web-приложения), для большинства приложений определенное алгоритмическое наполнение необходимо. Например, рассмотрим Web-службу, определяющую, как добраться из одного места в другое. В основе ее реализации лежит высокопроизводительное аппаратное обеспечение, графический интерфейс пользователя, глобальная сеть и, возможно, объектно-ориентированный подход. Кроме того, для определенных операций, выполняемых данной Web-службой, необходимо использование алгоритмов – например, таких как вычисление квадратных корней (что может потребоваться для определения кратчайшего пути), визуализации карт и интерполяции адресов.

Более того, даже приложение, не требующее алгоритмического наполнения на высоком уровне, сильно зависит от алгоритмов. Известно, что работа приложения зависит от производительности аппаратного обеспечения, а при его разработке применяются разнообразные алгоритмы. Все мы также знаем, что приложение тесно связано с графическим интерфейсом пользователя, а для разработки любого графического интерфейса пользователя требуются алгоритмы.

Вспомним приложения, работающие в сети. Чтобы они могли функционировать, необходимо осуществлять маршрутизацию, которая, как уже говорилось, основана на ряде алгоритмов. Чаще всего приложения составляются на языке, отличном от машинного. Их код обрабатывается компилятором или интерпретатором, которые интенсивно используют различные алгоритмы. И таким примерам нет числа. Кроме того, ввиду постоянного роста вычислительных возможностей компьютеров, они применяются для решения все более сложных задач. Как мы уже убедились на примере сравнительного анализа двух методов сортировки, с ростом сложности решаемой задачи различия в эффективности алгоритмов проявляются все значительнее. Знание основных алгоритмов и методов их разработки – одна из характеристик, отличающих умелого программиста от новичка. Располагая современными компьютерными технологиями, некоторые задачи можно решить и без основательного знания алгоритмов, однако знания в этой области позволяют достичь намного большего.

1.2. Анализ алгоритмов

Время выполнения алгоритма или операции над структурой данных зависит, как правило, от целого ряда факторов, вследствие чего возникает вопрос – как следует проводить его измерение. При реализации алгоритма можно определить затраты времени, регистрируя действительное время, затраченное на выполнение алгоритма в каждом отдельном случае запуска с различными исходными данными [8]. Подобные измерения должны проводиться с достаточной точностью с помощью системных вызовов, встроенных в язык или операционную систему, для которой написан данный алгоритм (например, метод `System.currentTimeMillis()` в Java или вызовом исполняющей среды с возможностью профилирования). В общем, требуется определить, каким образом время выполнения программы зависит от количества исходных данных. Для решения этой задачи можно провести ряд экспериментов, в которых будет использовано различное количество исходных данных. Далее полученные результаты наглядно представляются с помощью графика, где каждый случай выполнения алгоритма обозначается с помощью точки, координата x которой равна размеру исходных данных n , а координата y – времени выполнения алгоритма t (рис. 1.1). Чтобы сделать определенные выводы на основе полученных экспериментов, необходимо использовать качественные образцы исходных данных и провести достаточно большое число экспериментов, что позволит определить некоторые статистические характеристики в отношении времени выполнения алгоритма.

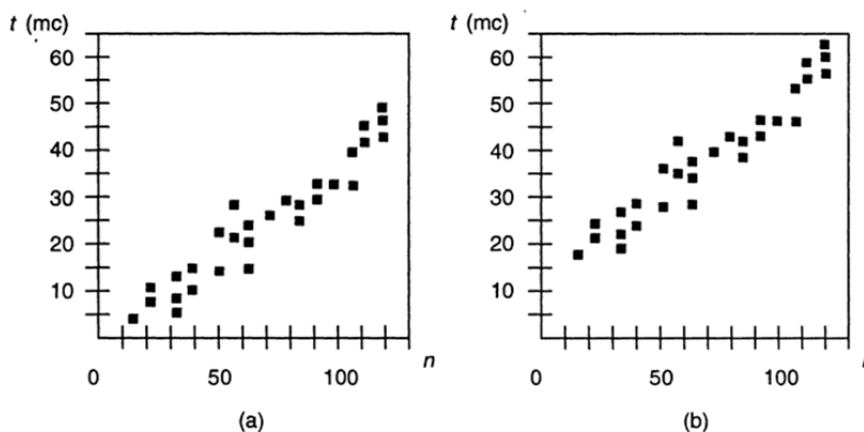


Рис.1.1

Результаты экспериментального исследования времени выполнения алгоритма. Точка с координатами (n, t) обозначает, что при размере

исходных данных n время выполнения алгоритма составило t миллисекунд (мс).

На рис. 1.1, а представлены результаты выполнения алгоритма на компьютере с быстрым процессором, на рис. 1.1, б представлены результаты выполнения алгоритма на компьютере с медленным процессором. В целом можно сказать, что время выполнения алгоритма или метода доступа к полям структуры данных возрастает по мере увеличения размера исходных данных, хотя оно зависит и от типа данных, даже при равном размере. Кроме того, время выполнения зависит от аппаратного обеспечения (процессора, тактовой частоты, размера памяти, места на диске и др.) и программ программного обеспечения (операционной среды, языка программирования, компилятора, интерпретатора и др.), с помощью которых осуществляется реализация, компиляция и выполнение алгоритма. Например, при всех прочих равных условиях время выполнения алгоритма определенного количества исходных данных будет меньше при использовании более мощного компьютера или при записи алгоритма в виде программы на машинном коде по сравнению с его исполнением виртуальной машиной, проводящей интерпретацию в байт-коды.

Экспериментальные исследования, безусловно, очень полезны, однако при их проведении существуют три основных ограничения:

- эксперименты могут проводиться лишь с использованием ограниченного набора исходных данных; результаты, полученные с использованием другого набора, не учитываются;
- для сравнения эффективности двух алгоритмов необходимо, чтобы эксперименты по определению времени их выполнения проводились на одинаковом аппаратном и программном обеспечении;
- для экспериментального изучения времени выполнения алгоритма необходимо провести его реализацию и выполнение.

Обычно для сравнительного анализа рассматривается, так называемая общая методология анализа времени выполнения алгоритмов, которая:

- учитывает различные типы входных данных;
- позволяет производить оценку относительной эффективности любых двух алгоритмов независимо от аппаратного и программного обеспечения;
- может проводиться по описанию алгоритма без его – непосредственной реализации или экспериментов.

Сущность такой методологии состоит в том, что каждому алгоритму соответствует функция, которая представляет время выполнения алгоритма как функцию размера исходных данных n . Наиболее распро-

страненными являются функции n и n^2 . Например, можно записать следующее утверждение: «Время выполнения алгоритма A пропорционально n ». В этом случае, в результате проведения экспериментов, окажется, что время выполнения алгоритма A при любом размере входных данных n не превышает значения cn , где c является константой, определяемой условиями используемого аппаратного и программного обеспечения. Если имеются два алгоритма A и B , причем время выполнения алгоритма A пропорционально n , а время выполнения B пропорционально n^2 , то предпочтительнее использовать алгоритм A , так как функция n возрастает медленнее, чем функция n^2 . Выводы о скорости возрастания этих функций являются, безусловно, очевидными, однако предлагаемая методология содержит точные определения данных понятий. Прежде чем перейти к описанию общей методики, определим понятие псевдокод и области его применения.

1.2.1. Псевдокод

Зачастую программистам требуется создать описание алгоритма, предназначенное только для человека. Подобные описания не являются программами, но вместе с тем они более структурированы, чем обычный текст. В частности, «высокоуровневые» описания сочетают естественный язык и распространенные структуры языка программирования, что делает их доступными и вместе с тем информативными. Такие описания способствуют проведению высокоуровневого анализа структуры данных или алгоритма. Подобные описания принято называть псевдокодом.

Проблема «максимума в массиве» является простой задачей поиска элемента с максимальным значением в массиве A , содержащем n целых чисел. Для решения этой задачи можно использовать алгоритм `аггауМах`, который осуществляет просмотр массива A с использованием цикла `for`.

Псевдокод алгоритма `аггауМах` представлен во фрагменте кода, а полная реализация программы Java представлена в следующем фрагменте кода.

Input: массив A , содержащий n целых чисел ($n > 1$).

Output: элемент с максимальным значением в массиве A .

1 `currentMax` $\leftarrow A[0]$ выполняется один раз

2 `for` $i \leftarrow 1$ to $n - 1$ `do` выполняется от 1 до n , $n-1$ раз соответственно

3 `if` `currentMax` $< A[i]$ `then` выполняется $n-1$ раз

4 `currentMax` $\leftarrow A[i]$ выполняется максимально $n-1$ раз

5 `return` `currentMax` выполняется один раз

Фрагмент кода Алгоритм arrayMax

```
// Тестируем программу алгоритма поиска максимального элемента массива.
public class ArrayMaxProgram {
    // находит элемент с максимальным значением в массиве A, содержащем n целых //чисел.
    static int arrayMax(int[] A, int n) {
        int currentMax = A[0]; // выполняется один раз
        for (int i = 1; i < N; i++) /* выполняется от 1 до n, n-1 раз соответственно. */
            if (currentMax < A[i]) // выполняется n-1 раз
                currentMax = A[i]; // выполняется максимально n-1 раз
        return currentMax; // выполняется один раз
    }
    // Тестирующий метод, вызываемый после выполнения программы.
    public static void main(String args[] ) {
        int[] num = { 10, 15, 3, 5, 56, 107, 22, 16, 85 };
        int n = num.length;
        System.out.print("Array:");
        for (int j = 0; j < n; j++)
            System.out.print(" " + num[j]);
        System.out.println(".");
        System.out.println("The maximum element is" + arrayMax(num,n) + ".");
    }
}
```

Фрагмент кода Алгоритм arrayMax внутри законченной Java-программы.

Как можно заметить, псевдокод выглядит компактнее Java-кода, и его легче читать и понимать. При анализе псевдокода можно поспорить о правильности алгоритма arrayMax с простым аргументом. Переменная currentMax первоначально принимает значение первого элемента массива A. Можно утверждать, что перед началом итерации по номеру i значение currentMax равно максимальному значению среди первых i элементов массива A. Так как при повторении i значение currentMax сравнивается с $A[i]$, то, если это утверждение верно перед данной итерацией, оно будет верно и после нее для $i + 1$ (которое является следующим значением счетчика i). Таким образом, после количества итераций $n - 1$ значение currentMax будет равно элементу массива A, содержащему максимальное значение.

Псевдокод является сочетанием естественного языка и конструкций языка программирования, которые используются для описания основных идей, заложенных в реализацию структуры данных или алгоритма. Точного определения языка псевдокода не существует, так как в нем все же преобладает естественный язык. В то же время для обеспечения четкости и ясности в псевдокоде наряду с естественным языком используются стандартные конструкции языка программирования, такие как [8]:

- **Выражения.** Для написания числовых и логических выражений используются стандартные математические символы. Знак стрелки применяется в качестве оператора присваивания в командах присваивания (равнозначен оператору «= \Rightarrow » языка Java). Знак «= \Rightarrow » используется для передачи отношения равенства в логических выражениях (что соответствует оператору «= $=$ » языка Java).

- **Объявление метода.** Имя алгоритма (*param1, param2,...*) объявляет «имя» нового метода и его параметры.

- **Структуры принятия решений.** Условие *if, then* – действия, если условие верно [*else* – если условие не верно]. Отступы используются для обозначения выполняемых в том или другом случае действий.

- **Цикл *while, while*** – условие, *do* - действия. Отступ обозначает действия, выполняемые внутри цикла.

- **Цикл *repeat, repeat*** – действия, которые выполняются, пока выполняется условие *until*. Отступ обозначает действия, выполняемые внутри цикла.

- **Цикл *for, for*** – описание переменной и инкремента, *do* – действия. Отступ обозначает действия, выполняемые внутри цикла

- **Индексирование массива.** $A[i]$ обозначает *i*-ую ячейку массива *A*. Ячейки массива *A* с количеством ячеек *n* индексируются от $A[0]$ до $A[n - 1]$ (как в Java).

- **Обращения к методам, *object.method (args)*** (часть *object* необязательна, если она очевидна).

- **Возвращаемое методом значение.** Значение *return*. Данный оператор возвращает значение, указанное в методе, вызывающим данный метод.

При записи псевдокода следует помнить, что он записывается для анализа человеком, а не компьютером, и необходимо выразить основные глобальные идеи, а не базовые детали реализации структуры. В то же время не следует упускать из виду важные этапы. Таким образом, как при любом типе человеческого общения, написание псевдокода состоит в поиске баланса между общим и частным; такое умение вырабатывается в ходе практической деятельности. Дополнительно, для луч-

шей передачи сущности алгоритма или структуры данных, описание алгоритма должно начинаться с краткого абстрактного объяснения характера входного и выходного потока данных, а также основных действий и используемых идей алгоритма.

1.2.2. Простейшие операции

После изложения высокоуровневого способа описания алгоритмов приступим к рассмотрению различных способов анализа алгоритмов.

Как уже отмечалось, экспериментальный анализ может быть очень полезен, но обладает рядом ограничений. Чтобы провести анализ алгоритма без экспериментов, связанных с изучением времени его выполнения, используем аналитический подход, который состоит из следующих операций:

1. Записать алгоритм в виде кода одного из развитых языков программирования (например, Java).
2. Перевести программу в последовательность машинных команд (например, байт-коды, используемые в виртуальной машине Java).
3. Определить для каждой машинной команды i время t_i , необходимое для ее выполнения.
4. Определить для каждой машинной команды i количество повторений команды n_i за время выполнения алгоритма.
5. Определить произведение $t_i * n_i$ всех машинных команд, что и будет составлять время выполнения алгоритма.

Данный подход позволяет точно определить время выполнения алгоритма, однако он очень сложен и трудоемок, поскольку требуется доскональное знание машинных команд, создаваемых компилятором и средой, в которой выполняется алгоритм.

В силу этого удобнее проводить анализ непосредственно программы, написанной на языке высокого уровня, или псевдокода. Выделим ряд простейших операций высокого уровня, которые в целом не зависят от используемого языка программирования и могут использоваться в псевдокоде:

- присваивание переменной значения,
- вызов метода,
- выполнение арифметической операции (например, сложение двух чисел),
- сравнение двух чисел,
- индексация массива,
- переход по ссылке на объект,
- возвращение из метода.

Следует отметить, что простейшие операции соответствуют машинным командам, время выполнения которых зависит от аппаратного и программного обеспечения, но тем не менее является постоянной известной величиной. Вместо попыток определения времени выполнения отдельной простейшей операции достаточно просто подсчитать - количество таких операций и использовать полученное значение в качестве критерия оценки времени выполнения алгоритма. Такой подсчет количества операций можно соотнести со временем выполнения алгоритма в условиях определенного аппаратного и программного обеспечения, так как каждая простейшая операция соответствует машинной команде, время выполнения которой есть величина постоянная, а число простейших операций известно и неизменно. Данный метод основан на неявном предположении о том, что время выполнения различных простейших операций приблизительно одинаково. Таким образом, число t простейших операций, выполняемых внутри алгоритма, пропорционально действительному времени выполнения данного алгоритма.

Рассмотрим процесс подсчета числа простейших операций, выполняемых внутри алгоритма, на примере алгоритма `aggauMax`. Псевдокод и Java-реализация этого алгоритма представлены во фрагментах кодов 3.1 и 3.2 соответственно. Данный анализ может проводиться как на основании псевдокода, так и Java-реализации.

- На этапе инициализации переменной `currentMax` и присваивания ей значения $A[0]$ выполняются две простейшие операции (индексация массива и присваивание переменной значения), которые однократно выполняются в начале алгоритма. Таким образом, счетчик операций равен 2.

- В начале выполнения цикла `for` счетчик i получает значение 1. Это соответствует одной простейшей операции (присваивание значения переменной).

- Перед выполнением тела цикла проверяется условие $i < n$. В данном случае выполняется простейшая операция сравнения чисел. Так как первоначальное значение счетчика i равно 0, а затем его значение увеличивается на 1 в конце каждой итерации цикла, сравнение $i < n$ проверяется n раз. Таким образом, в счетчик простейших операций добавляется еще n единиц.

- Тело цикла `for` выполняется $n \geq 1$ раз (для значений счетчика 1, 2, ..., $n-1$). При каждой итерации $A[i]$ сравнивается с `currentMax` (две простейшие операции – индексирование и сравнение), значение $A[i]$, возможно, присваивается `currentMax` (две простейшие операции – индексирование и присваивание значения), а счетчик i увеличивается на 1 (две простейшие операции – сложение и присваивание значения). Таким

образом, при каждой итерации цикла выполняется 4 или 6 простейших операций, в зависимости от того $A[i] \leq \text{currentMax}$ или $A[i] > \text{currentMax}$. Таким образом, при выполнении тела цикла в счетчик простейших операций добавляется

$4(n - 1)$ или $6(n - 1)$ единиц.

- При возвращении значения переменной `currentMax` однократно выполняется одна простейшая операция.

Итак, число простейших операций $t(n)$, выполняемых алгоритмом `aggaMax`, минимально равно

$$2 + 1 + n + 4(n - 1) + 1 = 5n,$$

а максимально

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2.$$

Число выполняемых операций равно минимально ($t(n) = 5n$) в том случае, если $A[0]$ является максимальным элементом массива, то есть переменной `currentMax` не присваивается нового значения. Число выполняемых операций максимально равно ($t(n) = 7n - 2$) в том случае, если элементы массива отсортированы по возрастанию, и переменной `currentMax` присваивается новое значение при каждой очередной итерации цикла.

1.2.3. Анализ средних и худших показателей

На примере метода `aggaMax` можно убедиться, что при определенном типе исходных данных алгоритм выполняется быстрее, чем при других. В данном случае можно попытаться выразить время выполнения алгоритма как среднее, взятое на основе результатов, полученных при всех возможных исходных данных. К сожалению, проведение анализа с точки зрения средних показателей является весьма проблематичным, так как в этом случае требуется определить вероятностное распределение входящего потока. На рис. 3.3 схематично представлена зависимость времени выполнения алгоритма от распределения входного потока данных. Например, если исходные данные только типа «А» или «D».

При анализе средних показателей необходимо определить предположительное время выполнения алгоритма при некотором распределении входного потока данных. Для проведения подобных вычислений зачастую требуется применение понятий высшей математики и теории вероятности.

В связи с этим в дальнейшем будем по умолчанию указывать худший показатель времени выполнения алгоритма (если не будет оговорено другое условие). Можно сказать, что в худшем случае алгоритм

аггауМах выполняет $t(n) = 7n - 2$ простейших операций, то есть максимальное число простейших операций, выполняемых алгоритмом при использовании всех исходных данных размера n , составляет $7n - 2$.

Подобный тип анализа намного проще анализа средних показателей, так как не требует использования теории вероятности; для него просто необходимо определить, при каком типе исходных данных время выполнения алгоритма будет максимальным, что зачастую является вполне очевидным. Кроме того, использование такого подхода может способствовать совершенствованию алгоритмов. Другими словами, если создаваемый алгоритм должен успешно работать при худших исходных данных, предполагается, что он будет работать при любом типе исходных данных.

Таким образом, проектирование алгоритма на основании худших показателей приводит к созданию более устойчивой «сущности» алгоритма, аналогично ситуации, когда чемпион по бегу во время тренировок бежит только в гору.

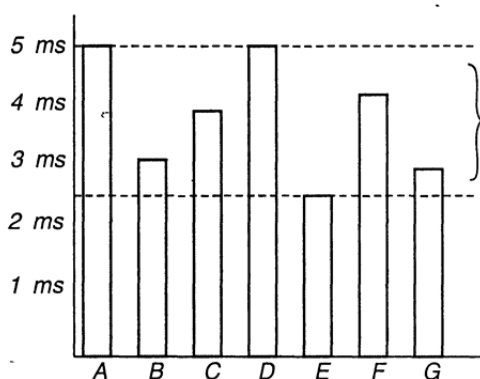


Рис. 1.2. Различие между наилучшим и наихудшим показателями времени выполнения алгоритма

Каждый прямоугольник соответствует времени выполнения алгоритма при различных типах исходных данных

1.2.4. Асимптотическая нотация

Очевидно, что анализ времени выполнения такого простого алгоритма, как аггауМах, проведен слишком глубоко. В ходе анализа возникает несколько вопросов:

- Действительно ли необходим такой уровень детализации?
- Действительно ли так важно установить точное число простейших операций, выполняемых алгоритмом?
- Насколько досконально следует определять количество простейших операций? Например, сколько простейших операций выполняется в

команде $y = a * x + b$? (Можно определить, что выполняются две арифметические операции и одна операция присваивания, но, с другой стороны, в этом случае не учитывается еще одна «скрытая» операция присваивания результата произведения $a * x$ временной переменной перед выполнением операции сложения.)

В целом каждый этап псевдокода или команда программы на языке программирования содержит небольшое число простейших операций, которое не зависит от размера исходных данных. Таким образом, можно проводить упрощенный анализ, при котором число простейших операций определяется применением некоторой константы, путем подсчета выполненных шагов псевдокода или команд программы. Возвращаясь к алгоритму `arrayMax`, упрощенный анализ даст следующий результат: алгоритм выполняет от $5n$ до $7n - 2$ шагов при размере исходных данных n .

При анализе алгоритмов следует рассматривать увеличение времени его выполнения как функцию исходных данных n , уделяя основное внимание глобальным аспектам и не вдаваясь в отдельные мелкие детали. Зачастую достаточно просто знать, что время выполнения алгоритма, например, `arrayMax`, увеличивается пропорционально n . Это означает, что действительное время выполнения алгоритма является произведением n на некий постоянный множитель, который определяется условиями аппаратного и программного обеспечения, и может колебаться в определенных пределах в зависимости от особенностей исходных данных.

Формализуем приведенный метод анализа структур данных и алгоритмов, используя математическую систему функций, в которой не учитываются постоянные факторы. В частности, будем описывать время выполнения и требования к памяти с помощью функций, которые преобразуют целые числа в действительные, обращая внимание на глобальные характеристики функции времени выполнения алгоритма или ограничения дискового пространства.

Рассмотрим основные понятия так называемой нотация большого O [6].

Пусть $f(n)$ и $g(n)$ являются функциями, которые преобразуют неотрицательные целые числа в действительные. Докажем, что $f(n)$ есть $O(g(n))$, если существует действительная константа $c > 0$ и целочисленная константа $n_0 > 1$, такие, что $f(n) < cg(n)$ для любого целого числа $n > n_0$. Такое определение функции зачастую называется нотацией большого O , так как иногда говорят, что $f(n)$ является большим O функции $g(n)$. Другими словами, можно сказать, что $f(n)$ есть порядковая функция $g(n)$ (определение показано на рис. 3.4).

Пример: $7n - 2$ есть $O(n)$.

Доказательство: по определению нотации большого O необходимо найти действительную константу $c > 0$, а также целочисленную константу $n_0 \geq 1$, так, что $7n - 2n_0 \leq cn$ для любого целого числа, $n \geq n_0$. Одним из очевидных вариантов является $c = 7$, а $n_0 = 1$. Безусловно, это один из бесконечного множества вариантов; c в данном случае может принимать значение любого действительного числа, равного или большего 7, а n_0 - любое целочисленное значение, большее или равное 1.

Нотация большого O позволяет выразить, что функция n «меньше или равна» другой функции (в определении это выражается знаком « \leq ») до определенного постоянного значения (константа c в определении) и по мере того, как n стремится к бесконечности (условие « $n \geq n_0$ » в определении).

Зачастую нотация большого O используется для характеристики времени выполнения и использования памяти на основании некоего параметра n , который может различаться в конкретных ситуациях, однако, как правило, зависит от целей анализа.

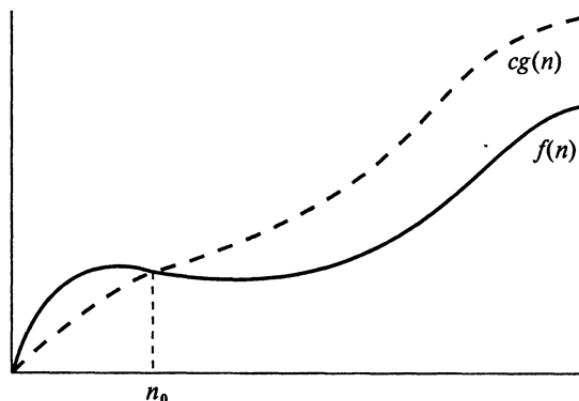


Рис. 1.3. Наглядное изображение нотации большого O .
Функция $f(n)$ есть $O(g(n))$, так как $f(n) \leq cg(n)$ при $n \geq n_0$

Например, если требуется найти наибольший элемент массива целых чисел (см. `аггауМах`, представленный во фрагментах кодов), то вполне естественно использовать n для обозначения числа элементов массива. Нотация большого O позволяет не учитывать постоянные множители и частные детали, а сосредоточиться только на основных компонентах функции, определяющих ее возрастание.

Используя нотацию большого O , можно представить математически точное определение времени выполнения алгоритма `аггауМах` независимо от применяемых аппаратного и программного обеспечения.

Утверждение. Время выполнения алгоритма `аггауМах`, определяющего максимальный элемент массива целых чисел, есть функция $O(n)$.

Доказательство. Как было отмечено, максимальное число простейших операций, выполняемых алгоритмом `аггауМах`, равно $7n - 2$. Таким образом, существует положительная константа a , которая определяется единицами измерения времени, а также применяемым при реализации, компиляции и исполнении аппаратным и программным обеспечениями, при которой время выполнения алгоритма `аггауМах` для размера

исходных данных n равно максимально $a(7n - 2)$. Используя нотацию большого O для $c = 7a, a_{n_0} = 1$, можно сделать вывод, что время выполнения алгоритма `аггауМах` является $O(n)$.

Рассмотрим еще несколько примеров, демонстрирующих использование нотации большого O .

Рассмотрим еще несколько примеров, демонстрирующих использование нотации большого O .

Пример: $20n^3 + 10n \log n + 5$ есть $O(n^3)$.

Доказательство: $20n^3 + 10n \log n + 5 < 35n^3$ для $n \geq 1$.

В сущности, любой многочлен (полином) $a^k n^k + a^{k-1} n^{k-1} + \dots + 0$ является $O(n^k)$.

Пример: 2^{100} есть $O(1)$

Доказательство: $2^{100} \leq 2^{100} \cdot 1$ для $n \geq 1$. Заметьте, что переменная n не присутствует в неравенстве, так как в данном случае рассматриваются постоянные функции.

Пример: $5 / n$ есть $O(1/n)$

Доказательство: $5 / n \leq 5(1/n)$ для $n \geq 1$ (на самом деле это убывающая функция).

В целом можно сказать, что нотация большого O используется для максимально точного описания функции. Если верно, что функция $F(n) = 4n^3 + 3n^{4/3}$ есть $O(n^5)$ или даже $O(n^3 \log n)$, то более точно будет сказать, что $f(n)$ есть $O(n^3)$. Рассмотрим следующую аналогию. Голодный путешественник долго едет по пустынной проселочной дороге и встречает местного фермера, который возвращается домой с рынка. Если путешественник спросит фермера, сколько ему нужно ехать, чтобы добраться до ближайшего места, где он может поесть, фермер может вполне правдиво ответить: «Не более 12 часов», но более точным (и полезным) будет, если он скажет: «Всего в нескольких минутах езды отсюда находится рынок».

Таким образом, даже используя нотацию большого O , следует стремиться сообщать все возможные подробности.

При анализе алгоритмов и структур данных часто применяются некоторые функции, имеющие особые названия. Например, термин «линейная функция» обозначает функции вида $O(n)$. В табл. 3.1 представлены функции, часто используемые при анализе алгоритмов.

Логарифмическая	Линейная	Квадратичная	Полиномиальная	Показательная
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k)(k \geq 1)$	$O(a^n)(a > 1)$

Классы функций

Нотация большого O позволяет проводить асимптотический анализ, то есть определять, что функция «меньше или равна» другой функции. Существуют типы нотаций, которые позволяют проводить асимптотические сравнения других типов: Ω - и Θ - нотации, которые асимптотически задают ограничения на функцию снизу или сверху одновременно (рис 1.4)

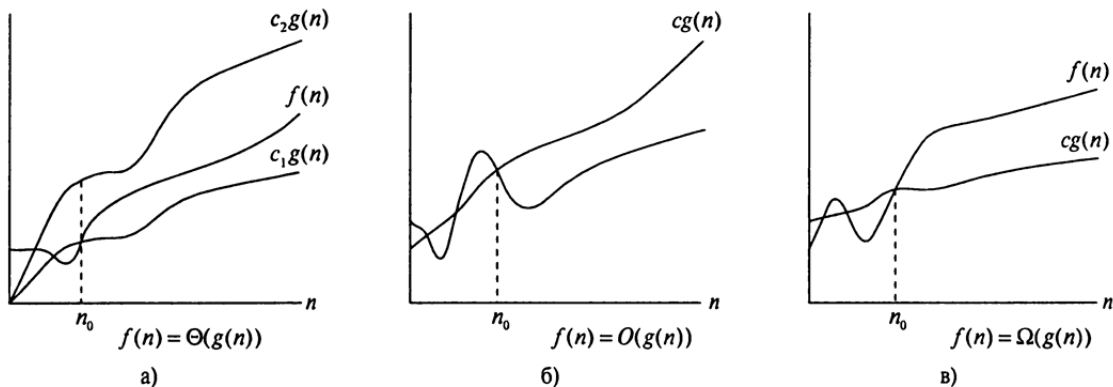


Рис. 1.4. Графические примеры Θ , O и Ω , обозначений; в каждой части рисунка в качестве n_0 используется минимально возможное значение, т.е. любое большее значение также сможет выполнить роль n_0

Выскажем некоторые предостережения в отношении использования асимптотических нотаций. Во-первых, следует отметить, что нотация большого O и аналогичные ей могут привести к ошибочным результатам, если «скрываемые» ими постоянные множители достаточно велики. Например, хотя функция $10^{100}n$ есть $\Theta(n)$, если она обозначает время выполнения алгоритма, сравниваемого с алгоритмом, время выполнения которого равно $10n \log n$, следует выбрать алгоритм со временем $\Theta(n \log n)$, даже несмотря на то, что асимптотически время выполнения, выраженное линейной функцией, меньше. Данный выбор обусловлен тем, что постоянный множитель 10^{100} , называемый «hiding», считается верхним пределом числа атомов в наблюдаемой части вселенной. Таким образом, маловероятно, что алгоритму придется решать задачу реального мира с таким размером исходных данных. Таким образом, при использовании нотации большого O следует обращать внима-

ние на постоянные множители и элементы низшего порядка, которые могут «скрываться» за ними.

В целом нотации большого O , большой омеги и большой теты являются удобным средством анализа структур данных и алгоритмов. Как упоминалось ранее, удобство данных нотаций состоит в том, что они позволяют сосредоточиться на основных составляющих, влияющих на время выполнения алгоритмов без учета частных деталей.

2. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ (АТД)

Со времени формулирования этой концепции в 1974 году (Liskov B., Zilles S. Programming with abstract data types) абстрактные типы данных играют важную роль в теории программирования. Концепция АТД является, наряду с объектно-ориентированным подходом, наиболее популярной в настоящее время методологией для создания программ. В процессе декомпозиции программы на составляющие компоненты доступ к ним организуется посредством так называемого кластера операций, который представляет собой конечный список операций, которые могут быть использованы для модификации данных, предоставляемых данным компонентом.

Отличительной особенностью АТД как механизма абстракции является тот факт, что функциональность компонента программы, описываемая кластером операций, может быть реализована различными способами. Различные реализации абстрактных типов данных взаимозаменяемы благодаря механизму абстракции АТД, позволяющему скрыть детали реализации с помощью набора predetermined операций.

Концепция абстрактных типов данных хорошо описывается с помощью математической теории алгебраических систем. Алгебраическую систему или, проще говоря, алгебру (абстрактную алгебру), неформально можно определить как множество с набором операций, действующих на элементах данного множества. Операции реализуются как функции от одного или более параметров, действующие на элементах данного множества (для операции с одним аргументом) или на декартовых произведениях множества (для операций с несколькими аргументами). Описание операций, включающее в себя описание типов аргументов и возвращаемых значений, называется сигнатурой алгебраической системы. Сигнатуры, очевидно, представляют математическую модель абстрактного типа данных. Это обстоятельство дает возможность опи-

сывать программные сущности, заданные посредством АДД, как алгебраические системы.

Типы данных впервые были описаны Д. Кнутом в его книге «Искусство программирования» [1]. В главе 2, «Информационные структуры», Кнут описывает так называемые структуры данных, определяемые как способы организации данных внутри программы. Кнут описывает такие типы данных, как списки, деревья, стеки, очереди, деки и т.д. Рассмотрим, например, как Кнут описывает тип данных «стек». Кроме собственно описания самой структуры данных, Кнут описывает «алгоритмы обработки» этой структуры с помощью словаря специальных терминов [1, стр. 281]. Для стека этот словарь содержит термины: push (втолкнуть), pop (вытолкнуть) и top (верхний элемент стека).

Таким образом, типы данных описываются Кнутом с помощью специального языка, задающего определенную терминологию и толкование этой терминологии. Эта особенность описания была замечена Стивеном Жиллем в работе [4] и, таким образом, явилась одним из побудительных мотивов для осознания важности концепции АДД.

В 1972 году была напечатана работа Дэвида Парнаса (David Parnas), в которой впервые был сформулирован принцип разделения программы на модули. Модули – это компоненты программы, которые имеют два важных свойства:

- модули скрывают в себе детали реализации;
- модули могут быть повторно использованы в различных частях программы.

Парнас представлял модули как абстрактные машины, хранящие внутри себя состояния и позволяющие изменять это состояние посредством определенного набора операций. Эта концепция является базовой, как для концепции абстрактных типов данных, так и для объектно-ориентированного программирования.

Понятие абстрактного типа данных впервые в явном виде было сформулировано в совместной работе Стивена Жилия и Барбары Лисков (1974). В разделе «Смысл понятия абстракции» авторы обсуждают, каким образом понятие абстракции может быть применимо к программному коду. Абстракция – это способ отвлечься от неважных деталей и, таким образом, выбрать наиболее важные признаки для рассмотрения. В процессе создания программы разработчик строит программную модель решаемой задачи. В процессе построения программной модели разработчик оперирует элементами этой модели. Программный код структурируется соответствующим образом. Для выделения программных сущностей в коде программы естественно использовать механизм абстракции. В работе Жилия и Лисков рассматривался механизм т.н. поведенче-

ской абстракции или, в терминологии авторов, функциональной абстракции.

Функциональная абстракция подразумевает выделение набора операций (функций) для работы с элементами программной модели. Таким образом, сущности программной модели представляются с помощью набора операций. Так осуществляется поведенческая абстракция сущности в программном коде. Сами авторы использовали термин «operational cluster», т.е. набор операций, и назвали такой набор операций абстрактным типом данных.

Подход, основанный на АД, говорит нам, что серьезное интеллектуальное исследование должно отвергать всякую попытку понять суть вещей изнутри как бесполезную, и вместо этого должно сосредотачиваться на понимании используемых свойств этих вещей. Не объясняйте мне, что вы собой представляете, скажите мне, что у вас есть – что я могу от вас получить, т.е. принцип разумного эгоизма.

Конструкторы программ брались – с различной степенью успеха – за решение ряда самых сложных из когда-либо рассматриваемых интеллектуальных задач. Немногие из инженерных проектов могут сравниться по сложности с программными проектами, содержащими много миллионов строк, которые регулярно производятся в наши дни. Приложив немало амбициозных усилий, программистское сообщество достигло точного понимания таких предметов и понятий, как размер, сложность, структура, абстракция, таксономия, параллельность, рекурсивный вывод, различие между описанием и предписанием, язык, изменение и инварианты. Все это произошло так недавно и настолько интуитивно, что сама эта профессиональная среда еще не осознала последствий собственной деятельности. В конце концов, появится кто-нибудь, кто объяснит, какие уроки весь интеллектуальный мир может извлечь из опыта конструирования ПО. Нет сомнений в том, что абстрактные типы данных будут играть в них выдающуюся роль.

Подытоживая, перечислим ключевые моменты концепции АД.

1. Теория абстрактных типов данных (АТД) примиряет необходимость в точности и полноте спецификаций с желанием избежать лишних деталей в спецификации.

2. Спецификация абстрактного типа данных является формальным математическим описанием, а не текстом программы. Она аппликативна, т.е. не включает в явном виде изменений.

3. АД может быть родовым, и он задается функциями, аксиомами и условиями. Аксиомы и условия выражают семантику данного типа и важны для полного и однозначного его описания.

4. Частичные функции образуют удобную математическую модель для описания не всюду определенных операций. У каждой частичной функции имеется условие, задающее условие, при котором она будет выдавать результат для заданного конкретного аргумента.

5. Объектно-ориентированная система – это совокупность классов. Каждый класс основан на некотором абстрактном типе данных и задает частичную или полную реализацию этого АД.

6. Класс является эффективным, если он полностью реализован, в противном случае он называется отложенным.

7. Классы должны разрабатываться в наиболее общем виде, допускающем повторное использование; процесс их объединения в систему часто идет снизу-вверх.

Абстрактные типы данных являются скорее неявными, чем явными описаниями. Эта неявность, которая также означает открытость, переносится на весь объектно-ориентированный подход.

2.1. АД векторы, списки и последовательности

Концепция последовательности [3,5,9], в которой элементы следуют друг за другом, в программировании является фундаментальной. Последовательность встречается в строковом выводе кода компьютерной программы, где порядок команд определяет выполняемые программой действия. Последовательность сетевых пакетов составляет сообщение электронной почты, поскольку сообщение будет иметь смысл только при приеме пакетов в той же последовательности, в которой они отправлены. Последовательности представляют такие важные отношения между объектами, как «следующий» и «предыдущий». Кроме того, последовательности зачастую используются для реализации других структур данных, то есть они представляют собой блоки, на которых базируется проектирование таких структур.

2.1.1. АД вектор

АД «вектор» расширяет конкретную структуру данных массива. Он содержит методы доступа к обычным элементам последователь-

ности, а также методы обновления, обеспечивающие удаление и добавление элементов с определенным индексом. Для отличия от процесса доступа к элементам конкретной структуры данных массива при обозначении индекса элемента вектора используется термин «разряд».

Пусть S – линейная последовательность из n элементов. Каждый элемент e последовательности S имеет уникальный индекс, выраженный целым числом в интервале $[0, n-1]$, равный числу элементов, предшествующих e . Таким образом, определим, что разряд элемента e последовательности S равен количеству элементов, находящихся в S перед e , то есть разряд первого элемента последовательности равен 0, а последнего – $n-1$. Данный метод соответствует принципу индексирования массивов в Java и других языках программирования (в том числе C и C++).

Обратите внимание, что в определении не требуется, чтобы реализованная в массиве последовательность сохраняла элемент с разрядом 0 в ячейке с индексом 0, хотя это один из возможных вариантов. Понятие разряда позволяет обращаться к «индексу» элемента последовательности без учета конкретной реализации списка. Итак, если элемент является r -ным элементом последовательности, его разряд равен $r-1$. Таким образом, если разряд элемента равен r , то разряд предыдущего элемента (если он существует) равен $r-1$, а следующего элемента (если он существует) $r+1$. Следует, однако, отметить, что разряд элемента может изменяться при обновлении последовательности. Например, при добавлении нового элемента в начале последовательности разряд каждого элемента увеличится на 1.

Линейная последовательность элементов, в которой доступ элементам осуществляется по их разряду, называется вектором. Разряд является простым и в то же время удобным средством, так как с его помощью определяется место добавления нового элемента или удаления элемента вектора. Например, можно указать разряд, который получит новый элемент после его добавления (например, insert at rank 2). Разряд также указывает элемент, который должен быть удален из вектора (например, remove the element at rank 2).

Вектор S является абстрактным типом данных, который поддерживает следующие основные методы:

elemAtRank (r): возвращает элемент S с разрядом r ; если $r < 0$ или $r > n-1$, где n - текущее число элементов, выдается сообщение об ошибке.

Input: целое число; **Output:** объект.

replaceAtRank (r, e): замещает объектом e элемент с разрядом r и возвращает замещаемый объект. Если $r < 0$ или $r > n-1$, выдается сообщение об ошибке.

Input: целое число r и объект e ; **Output:** объект.

insertAtRank (r, e): добавляет в S новый элемент e , если $r < 0$ или $r > n$, выдается сообщение об ошибке.

Input: целое число r и объект e ; **Output:** нет.

removeAtRank (r): удаляет из S элемент с разрядом r ; если $r < 0$ или

$r > n - 1$, выдается сообщение об ошибке.

Input: целое число; **Output:** объект.

Кроме того, АД вектор обычно поддерживает стандартные методы **size()** и **isEmpty()**.

Реализация с помощью массива

Самым очевидным способом реализации векторного АД является его реализация на основе массива A , где $A[i]$ содержит ссылку на элемент разряда i . Считаем длину N массива A достаточно большой, а для обозначения количества элементов вектора используем переменную $n < N$. Реализация методов векторного АД достаточно проста. При выполнении операции **elemAtRank**(r) программа возвращает $A[r]$. Реализация методов **insertAtRank**(r, e) и **removeAtRank**(r) показана во фрагменте кода 5.1. Одна из операций (занимающая значительную часть времени) заключается в перемещении элементов таким образом, чтобы занятые ячейки массива образовывали неразрывную последовательность. Такое перемещение необходимо для соблюдения установленного нами правила о хранении элемента с разрядом i в A с индексом i (рис. 5.1)

Алгоритм **insertAtRank**(r, e)

1 for $i = n - 1, n - 2, \dots, r$ do

2 $A[i + 1] \leftarrow A[i]$ (создает место для нового элемента)

3 $A[r] \leftarrow e$

4 $n \leftarrow n + 1$

Алгоритм **removeAtRank**(r)

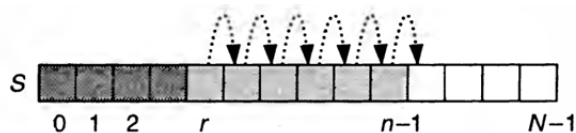
$e \leftarrow A[r]$ (e – временная переменная)

for $i = r, r + 1, n - 2$ do

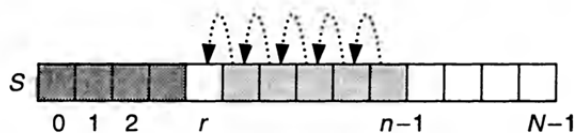
$A[i] \leftarrow A[i + 1]$ (замещает удаленный элемент)

$n \leftarrow n - 1$

return e



(a)



(b)

Рис. 2.1. Реализация вектора S ,

содержащего n элементов, на основе массива:

- a – перемещение элементов вперед для вставки нового элемента с разрядом r ;
- b – перемещение элементов назад для заполнения разряда r удаленного элемента

В табл. 5.2 представлено время выполнения методов реализации вектора на основе массива. Методы `isEmpty`, `size` и `elemAtRank` выполняются за время $O(1)$ однако для выполнения методов вставки и удаления элементов может потребоваться гораздо больше времени. В частности, худшем случае метод `insertAtRank(r, e)` будет выполняться за время $\Theta(n)$. Худшими условиями для данного метода является ситуация, когда $r = 0$, так как все существующие n элементов должны быть перемещены вперед. То же можно сказать и о методе `removeAtRank(r)`, который выполняется за время $O(n)$, так как в худшем случае $r=0$, и ему необходимо переместить $n - 1$ элементов. В действительности, если предположить, что каждый разряд равновероятно может быть аргументом данных операций, среднее время выполнения есть $\Theta(n)$, так как в среднем необходимо будет переместить $n/2$ элементов.

Таблица 2.2

Выполнение вектора, реализованного на основе массива
(размер используемого пространства $O(N)$, где N – размер массива)

Метод	Время
<code>size</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>elemAtRank(r)</code>	$O(1)$
<code>replaceAtrank(r, e)</code>	$O(1)$
<code>insertAtRank(r, e)</code>	$O(n)$
<code>removeAtRank(r)</code>	$O(n)$

При более детальном рассмотрении методов `insertAtRank(r, e)` и `removeAtRank(r)` можно определить, что время выполнения каждого из

них равно $O(n - r + 1)$, так как в ходе исполнения программы перемещаются только элементы с разрядом r и выше. Таким образом, для добавления или удаления элемента в конце вектора с помощью методов `insertAtRank(n, e)` и `removeAtRank(n-1)` необходимо время $O(1)$.

Основной недостаток реализации вектора на основе простого массива, состоит в необходимости предварительного указания размера массива N , то есть максимального числа элементов вектора. Если же действительное число элементов значительно меньше N , то при такой реализации бесполезно занимает место в памяти. Хуже того, если n окажется больше значения N , то реализация приведет к сбою программы. К счастью, существует простой способ разрешения этой проблемы.

Предположим, имеются средства увеличения размера массива A , содержащего элементы вектора S . Безусловно, в Java (и других языках программирования) в действительности невозможно увеличить размер массива, его длина ограничена заданным значением N , как отмечалось ранее. Вместо этого при возникновении переполнения, то есть при $n = N$, и вызове метода `insertAtRank` выполняются следующие операции:

- 1) создается новый массив B длиной $2N$;
- 2) копируется $A[i]$ в $B[i]$, где $i = 0, \dots, N-1$;
- 3) присваивается $A = B$, то есть используется B как массив, содержащий S .

Данная стратегия замещения массивов известна как расширяемый массив

2.1.2. АД «Список»

Для доступа к месту расположения элемента в списке может применяться не только понятие разряда. Если имеется список S , реализованный на основе однонаправленного или двусвязного списка, более удобным и эффективным является использование для определения места доступа и обновления списка узлов вместо разрядов. Рассмотрим абстрактное понятие узла, описывающего определенное «место» в списке, не вдаваясь в детали реализации списка.

Пусть S – линейный список, реализованный с помощью двусвязного списка. Необходимо определить методы для S , которые принимают в качестве параметров узлы списка и возвращают значения узлов. Данные методы значительно быстрее методов, основанных на использовании разрядов, так как для поиска разряда в связном списке необходим последовательный просмотр всех элементов от начала до конца, с подсчетом пройденных элементов.

Например, определим гипотетический метод `removeAtNode(v)`, который удаляет элемент S , хранящийся в узле списка v . Использование узла в качестве параметра позволяет удалить элемент за время $O(1)$ путем перехода к данному узлу, удаления его и перенаправления соответствующим образом ссылки `next` и `prev` соседних узлов. Аналогично можно добавить новый элемент e за время $O(1)$ с помощью операции `insertAfterNode(v,e)`, определяющий узел v , после которого необходимо вставить узел нового элемента. В данном случае просто «вплетается» новый узел.

Добавив к методам АД «список» упомянутые операции с узлами, определим количество информации, которое необходимо задать для непосредственной реализации списка. Безусловно, в лучшем случае можно применить однонаправленный или двусвязный список, не сообщая пользователю деталей. Аналогично пользователю не следует модифицировать внутреннюю структуру списка без особой надобности. Такая модификация, однако, может быть возможна, если предоставленная пользователю ссылка на узел списка разрешает доступ к внутренним данным этого узла (то есть доступ к полям `next` и `prev`).

Для абстрагирования и унификации разных способов хранения элементов в различных реализациях списка введем понятие позиции в списке, которое формализует традиционное понятие «места» элемента, связанного с другими элементами в списке.

Таким образом, для изучения операций над списками введено абстрактное понятие «позиции», которое позволяет использовать реализации с помощью однонаправленного и двусвязного списка, не нарушая в то же время принципов объектно-ориентированного программирования. В данной схеме список рассматривается как контейнер элементов, хранящихся в определенных позициях, и эти позиции линейно упорядочены.

Позиция сама по себе является абстрактным типом данных, который поддерживает следующий простой метод:

element(): возвращает элемент, хранящийся в данной позиции.

Input: нет; **Output:** объект.

Позиции всегда определяются относительно своих соседей. В списке позиция p всегда будет следовать «за» некоторой позицией q и «перед» некоторой позицией s (при условии, что p не является первой или последней позицией). Позиция p , связанная с некоторым элементом e в списке S , остается неизменной даже при изменении разряда e , если e явно не удаляется (и таким образом не уничтожается позиция p). Более того, позиция p не изменится, даже если заменить элемент e , хранящийся в p , на другой элемент. Такие особенности позволяют определить

большое число методов, использующих позиции в качестве параметров и возвращающих объекты, хранящиеся в позициях.

Используя концепцию позиции для инкапсуляции «узла» списка, опишем еще один тип последовательностей, называемый АДД «список». Этот АДД поддерживает следующие методы, выполняемые над списком S :

first(): возвращает позицию первого элемента списка S ; если список пуст, выдается сообщение об ошибке.

Input: нет; **Output**: позиция.

last(): возвращает позицию последнего элемента списка S ; если список пуст, выдается сообщение об ошибке.

Input: нет; **Output**: позиция.

isFirst (p): возвращает логическое значение, показывающее, является ли данная позиция первой в списке.

Input: позиция p ; **Output**: логическое значение.

isLast (p): возвращает логическое значение, показывающее, является ли данная позиция последней в списке.

Input: позиция p ; **Output**: логическое значение.

before (p): возвращает позицию элемента S , который предшествует элементу позиции P ; если p является первой позицией, выдается сообщение об ошибке.

Input: позиция; **Output**: позиция.

after (p): возвращает позицию элемента S , который следует за элементом позиции p ; если p является последней позицией, выдается сообщение об ошибке.

Input: позиция; **Output**: позиция.

Данные методы позволяют обращаться к смежным позициям списка, начиная с конца или с начала, а также постепенно перемещаться по всему списку. Позиции можно считать узлами списка, однако обратите внимание, что в этих методах нет особых ссылок к объектам узлов и ссылок `next` и `prev` этих методов. Кроме указанных методов, а также общих методов **size** и **isEmpty** АДД «список» описывает следующие методы обновления списка:

replaceElement (p, e): замещает элемент в позиции p на e и возвращает элемент, который до этого был в позиции p .

Input: позиция p и объект e ; **Output**: объект.

swapElements (p, q): меняет местами элементы в позициях p и q таким образом, что элемент в позиции p перемещается в позицию q , а элемент, бывший в позиции q , перемещается в позицию p .

Input: две позиции; **Output**: нет.

insertFirst(e): вставляет новый элемент e в S в качестве первого элемента списка.

Input: объект e ; **Output:** позиция вставленного элемента e .

insertLast(e): вставляет новый элемент e в S в качестве последнего элемента списка.

Input: объект e ; **Output:** позиция вставленного элемента e .

insertBefore (p, e): вставляет новый элемент e в S перед позицией p ; если p является первой позицией, выдается сообщение об ошибке.

Input: позиция p и объект e ; **Output:** позиция вставленного элемента e .

insertAfter (p, e): вставляет новый элемент e в S после позиции p ; если p является последней позицией, выдается сообщение об ошибке.

Input: позиция p и объект e ; **Output:** позиция вставленного элемента e .

remove (p): удаляет из S элемент в позиции p .

Input: позиция; **Output:** удаленный элемент.

АТД «список» позволяет рассматривать упорядоченные с точки зрения местоположения коллекции объектов, не учитывая, каким именно образом представлены эти места.

Реализация АТД «список»

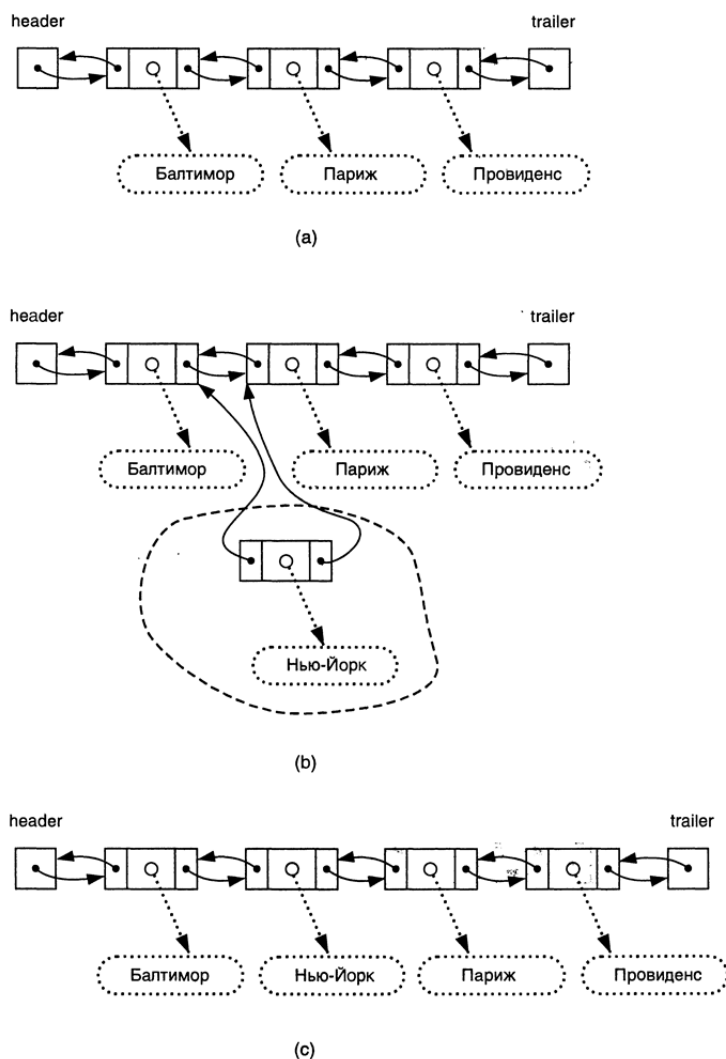
Для реализации АТД «список» обычно используются односвязные и двусвязные списки. Двусвязные списки предпочтительнее, поскольку позволяют осуществлять прохождение элементов в двух направлениях.

Предположим, необходимо создать АТД «список» с помощью двусвязного списка. В самом простом случае узлы двусвязного списка будут соответствовать позиции АТД. Другими словами, каждый узел реализует интерфейс `Position` и, следовательно, содержит метод `element()`, который возвращает элемент, хранящийся в данном узле. Таким образом, сами узлы выступают в качестве позиций, имея двойственный характер: с одной стороны, это внутренние узлы связного списка, а с другой – традиционные позиции. Таким образом, для каждого узла v можно задать переменные `next` и `prev`, осуществляющие обращения соответственно к последующему и предыдущему узлам (которые могут быть просто сигнальными узлами, обозначающими начало или конец списка). Далее с точки зрения позиции в S нужно «открыть» позицию p , чтобы найти узел v . Эта операция выполняется с помощью приведения типа позиции к узлу. Поскольку действия осуществляются с узлами, можно выполнить метод `before (p)`, возвращая `v.prev` (при условии, что `v.prev` не является головным сигнальным узлом, иначе это приведет к ошибке). Таким образом, позиции списка при реализации двусвязным списком используют преимущества объектно-ориентированного программиро-

вания, причем для этого не требуются дополнительные затраты места и времени. Кроме того, данный позволяет скрыть детали реализации, что обеспечивает возможность неоднократного использования кода, так как пользователь не будет знать, что позиционные объекты, передаваемые и возвращаемые как параметры, в действительности являются объектами узлов.

На рис. 2.2–2.3 показаны реализация АД «список» с помощью двусвязных списков и схемы вставки и удаления элементов списков.

Класс `java.util.LinkedList` является встроенным классом Java, которые обладают аналогичными вышеописанным функциями АД «список».



*Рис.2.2. Добавление нового узла после позиции «Балтимор»:
a – до вставки; b – создание нового узла и его связывание; c – после вставки*

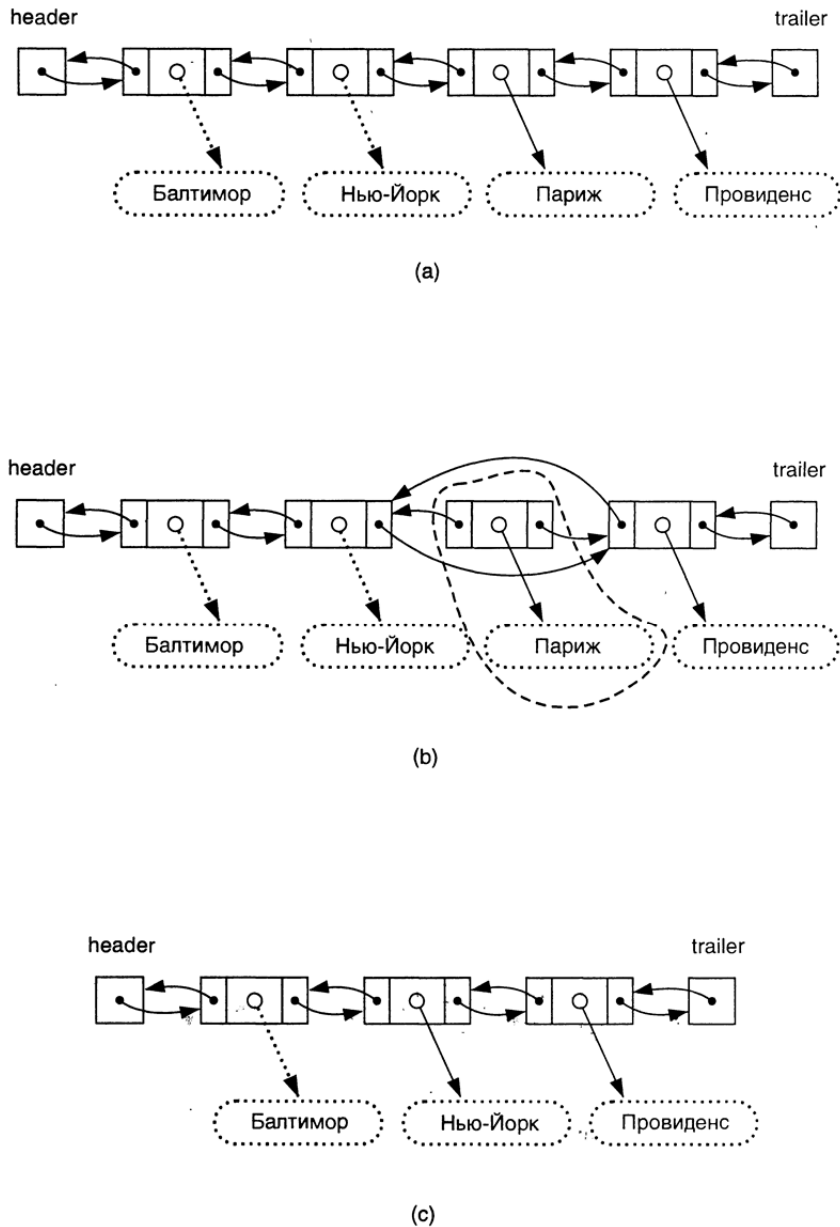


Рис 2.3. Удаление объекта, хранящегося в позиции «Париж»:
a – до удаления; *b* – удаление старого узла; *c* – после удаления (и сбора мусора)

2.1.3. Абстрактный тип данных «стек»

Стек (англ. stack – стопка) – структура данных, в которой доступ к элементам организован по принципу LIFO (last in - first out, «последним пришёл – первым вышел»). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

Добавление элемента, называемое также проталкиванием (push), возможно только в вершину стека (добавленный элемент становится

первым сверху). Удаление элемента, называемое также выталкиванием (pop), тоже возможно только из вершины стека, при этом второй сверху элемент становится верхним.

Стеки широко применяются в вычислительной технике. Например, для отслеживания точек возврата из подпрограмм используется стек вызовов, который является неотъемлемой частью архитектуры большинства современных процессоров. Языки программирования высокого уровня также используют стек вызовов для передачи параметров при вызове процедур.

Арифметические сопроцессоры, программируемые микрокалькуляторы и язык Forth используют стековую модель вычислений.

В интернет-браузерах адреса посещаемых сайтов хранятся в виде стеков. При открытии пользователем нового Web-сайта его адрес добавляется, «вдавливается» в стек адресов. После этого пользователь может вернуться к открытым ранее сайтам с помощью кнопки «Назад».

В большинстве текстовых редакторов существует механизм отмены действий «Undo», с помощью которого можно отменить последние операции редактирования и вернуться к предыдущему виду документа. Такой механизм отмены действий реализуется благодаря хранению проводимых изменений в виде стека.

Механизм работы стека можно представить, если воспользоваться аналогией с железнодорожным разъездом, которая предложена Э. Дейкстрой (рис. 2.4)

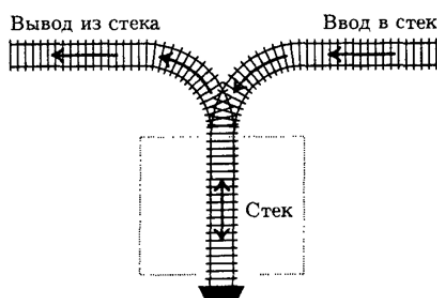


Рис. 2.4.

Многие исследователи независимо пришли к выводу о важности стеков и очередей, а потому присвоили им иные собственные имена. Так, стеки часто называют [1] магазинными списками (push-down lists), реверсивными хранилищами (reversion storages), магазинами (cellars), вложенными хранилищами (nesting stores), кучами (piles), дисциплинами обслуживания в обратном порядке (last-in-first-out lists-LIFO lists) и даже флюгерными списками (yo-yo lists).

Стек S является абстрактным типом данных, который поддерживает следующие основные методы:

push (o): помещает объект o в вершину стека.

Input: объект; **Output:** нет.

pop (): удаляет объект из стека и возвращает новый верхний объект стека; если стек пуст, выдается сообщение об ошибке.

Input: нет; **Output:** объект.

Кроме того, АД стек выполняет следующие дополнительные методы:

size (): возвращает число объектов в стеке.

Input: нет; **Output:** целое число.

isEmpty (): возвращает логическое значение, подтверждающее, что стек пуст.

Input: нет; **Output:** логическое значение.

top (): возвращает верхний объект в стеке, не удаляя его; если стек пуст, выдается сообщение об ошибке.

Input: нет; **Output:** объект.

Реализация с помощью массива

Самым простым способом реализации стека является хранение его элементов в виде массива. Так как длина массива должна быть задана при его создании, то одним из важных аспектов рассматриваемой реализации стека является необходимость указания некоторого максимального размера M стека, например, $N = 1000$ элементов. Таким образом, полученный стек состоит из массива S , содержащего N элементов, плюс целочисленная переменная t , которая обозначает индекс последнего элемента стека S (рис. 2.5).



Рис. 2.5. Реализация стека в виде массива S .
Последний элемент массива хранится в ячейке $S[t]$

Поскольку индекс массива в Java начинается с 0, переменной t первоначально присваивается значение, равное -1, которое показывает, что стек пуст. Эта же переменная может использоваться для определения числа элементов в стеке ($t + 1$).

Algorithm size():

1 return $t + 1$

Algorithm isEmpty():

1 return ($t < 0$)

Algorithm top():

1 if isEmpty() then
2 вызов StackEmptyException
3 return $S[t]$

Algorithm push(o):

1 if size() = N then
2 вызов StackFullException
3 $t \leftarrow t + 1$
4 $S[t] \leftarrow o$

Algorithm pop():

1 if isEmpty() then
2 вызов StackEmptyException
3 $e \leftarrow S[t]$
4 $S[t] \leftarrow \text{null}$
5 $t \leftarrow t - 1$
6 return e

Фрагмент кода: Реализация стека с помощью массива

При данной реализации АД «стек» время выполнения каждого метода постоянно и равно $O(1)$.

Вторым возможным способом реализации является представление стека в виде связанного списка (рис. 2.6)

В принципе, вершина стека может находиться как в начале, так и в конце списка. С другой стороны, так как добавление и удаление элементов за время $O(1)$ возможно только в начале списка, именно здесь более эффективно определить вершину стека. Кроме того, для выполнения операции size за время $O(1)$ контролировать текущее число элементов будем с помощью переменной экземпляра. Все методы интерфейса Stack выполняются за время $O(1)$. Кроме экономии времени, данная реализация определяет занимаемое стеком пространство, равное $O(n)$, где n – текущее число элементов стека. Таким образом, размер используемого места зависит от числа элементов в стеке, а не от задаваемого предела. Кроме того, данная реализация не требует создания нового исключения при превышении задаваемого предела. Вместо этого исполняющая среда будет рассматривать в целом, превысит ли реализация выделенный размер памяти, что приведет к возникновению ошибки OutOfMemoryError. Таким образом, реализация стека на основе однона-

правленного связанного списка имеет существенное преимущество по сравнению с реализацией на основе массива – требуется явно устанавливать максимальный размер стека. Используем переменную ссылочного типа `top` для обращения к началу списка (имеет нулевое значение при пустом списке). При добавлении в стек нового элемента e создается новый узел v , из v устанавливается ссылка на e , и v добавляется в начало списка. Аналогично при извлечении элемента из стека узел удаляется из начала списка и возвращается его элемент. Таким образом выполняются все добавления и удаления элементов в начале списка.

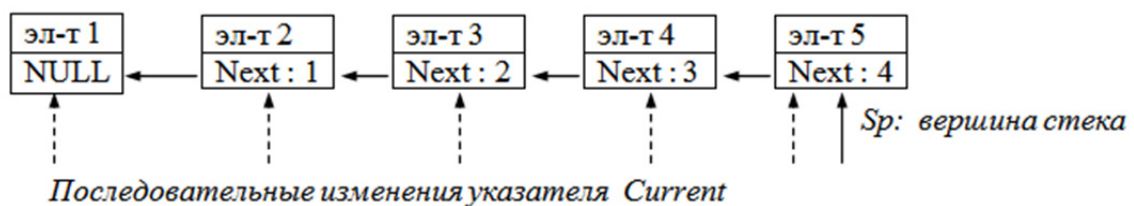


Рис.2.6.

В качестве встроенного класса реализующего АД стек можно процитировать класс [14,18] **java.util. Stack (public class Stack extends Vector implements Cloneable, Collection, List, Serializable)**. Методы этого класса:

boolean empty() проверка стека на наличие элементов – он возвращает true, если стек пуст.

Object peek() возвращает верхний элемент, не удаляя его из стека.

Object pop() извлекает верхний элемент, удаляя его из стека.

Object push(Object item) помещает элемент в вершину стека.

int search(Object o) ищет заданный элемент в стеке, возвращая количество операций `pop`, которые требуются для того чтобы перевести искомый элемент в вершину стека. Если заданный элемент в стеке отсутствует, этот метод возвращает -1.

2.1.4. Абстрактный тип данных «очередь»

Еще одной базовой структурой данных является очередь. Эта структура данных аналогична стеку, так как очередь объединяет, объекты, работа с которыми – добавление и удаление – осуществляется по принципу FIFO (first-in first-out) («первым пришел – первым ушел»). Добавление объектов в очередь может осуществляться в любое время, однако удаленным может быть лишь объект, который был добавлен первым. Говорят, что

элементы добавляются в очередь с конца, а удаляются с начала. Можно сравнить очередь данных с очередью людей в парке аттракционов. Люди встают в очередь с конца, а те, кто был первым в очереди, катаются на качелях, например.

Формально абстрактный тип данных «очередь» представляет собой последовательность объектов, в которой доступ и удаление разрешены только для первого элемента очереди, называемого началом очереди, а добавление элементов возможно только в конец очереди.

Очередь S является абстрактным типом данных, который поддерживает два следующих основных метода:

enqueue (o): помещает объект o в конец очереди.

Input: объект; **Output:** нет.

dequeue (): производит удаление и возвращает объект из начала очереди; если очередь пуста, выдается сообщение об ошибке.

Input: нет; **Output:** объект.

Кроме того, как и в случае с АД «стек», АД «очередь» выполняет следующие дополнительные методы:

size (): возвращает число объектов в очереди.

Input: нет; **Output:** целое число.

isEmpty (): возвращает логическое значение, подтверждающее, что очередь пуста.

Input: нет; **Output:** логическое значение.

front (): возвращает первый объект в очереди, не удаляя его; если очередь пуста, выдается сообщение об ошибке.

Input: нет; **Output:** объект.

Реализация на основе массива

Рассмотрим, каким образом можно осуществить реализацию очереди с помощью массива Q , длина которого для хранения элементов очереди, например,

$N=1000$. Так как основной характеристикой АД «очередь» является добавление и удаление объектов по принципу FIFO, необходимо определить способ фиксации начала и конца очереди.

Одним из возможных решений является подход, примененный при реализации стека, а именно – допущение, что $Q[0]$ является первым элементом очереди и с него начинается рост очереди. Однако такое решение не является эффективным, так как в этом случае придется перемещать все элементы массива на одну ячейку вперед при выполнении операции dequeue. В силу этого для выполнения метода dequeue данная реализация потребует $O(n)$ времени, где n – текущее значение количества объектов в очереди. Поэтому, если необходимо установить некоторое постоянное

время выполнения каждого метода очереди, требуется найти другое решение.

Для того чтобы избежать перемещения объектов в Q , введем две переменные f и r :

- f является индексом ячейки Q , в которой хранится первый элемент очереди (кандидат на удаление при выполнении метода dequeue) при условии, что очередь не является пустой (в этом случае $f=r$).

- r является индексом следующей доступной ячейки массива Q . Изначально присваиваем $f=r=0$, что означает, пустую очередь. Теперь, после удаления элемента из начала очереди, просто увеличиваем значение f на 1 для получения индекса следующей ячейки массива. Если добавляем новый элемент, то увеличиваем значение r на 1 для получения индекса следующей доступной ячейки массива Q . При таком подходе можно выполнить методы enqueue и dequeue за фиксированный промежуток времени, равный $O(n)$. Тем не менее при этом возникают определенные проблемы.

Рассмотрим пример, когда многократно удаляется и добавляется единственный элемент N . В данном случае $f=r=N$. Попытка добавления этого элемента еще раз приведет к ошибке превышения пределов массива (так как допустимые значения N в Q находятся в интервале от $Q[0]$ до $Q[N-1]$), даже, несмотря на наличие в очереди свободного места. Во избежание подобных проблем, а также для обеспечения возможности использования всего массива Q предположим, что индексы f и r «охватывают» конечные точки массива Q . В этом случае Q рассматривается как «циклический» массив, элементы которого изменяются от $Q[0]$ до $Q[N-1]$, а затем снова возвращаемся к $Q[0]$ (см. рис 2.7).

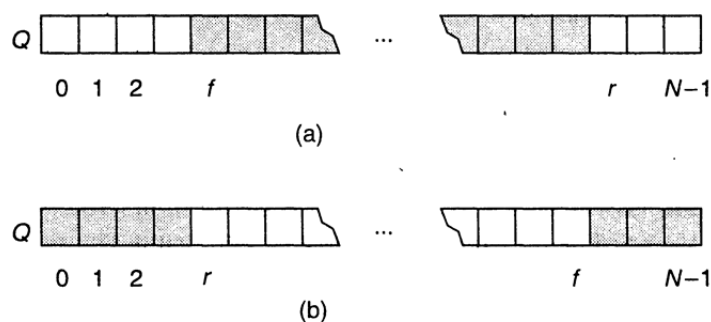


Рис. 2.7. Представление массива в виде «окружности»:

a – обычная конфигурация, где $f \leq r$; b – «циклическая» конфигурация, где $r < f$. Ячейки, в которых хранятся элементы очереди, заштрихованы

Реализация подобного представления Q довольно проста. При каждом увеличении значения f и r на 1 необходимо записать эту операцию в виде « $(f + 1) \bmod N$ » или « $(r + 1) \bmod N$ » соответственно, где \bmod обозна-

чает операцию деления по модулю. времени. Таким образом, с помощью операции деления по модулю разрешается проблема превышения пределов массива, однако еще одна небольшая проблема остается.

Рассмотрим ситуацию добавления N новых объектов в Q , не удаляя ни один из них. В этом случае $f = r$, что является условием пустой очереди, то есть в данном случае невозможно определить различие между пустой и полной очередью. Существует целый ряд способов решения данной задачи. В предлагаемом решении необходимо установить, что Q не может содержать более $N - 1$ объектов.

Алгоритм `size()`:

```
1 return  $(N - f + r) \bmod N$ 
```

Алгоритм `isEmpty()`:

```
1 return  $(f = r)$ 
```

Алгоритм `front()`:

```
1 if isEmpty() then  
2 вызов QueueEmptyException  
3 return  $Q[f]$ 
```

Алгоритм `dequeue()`:

```
1 if isEmpty() then  
2 вызов QueueEmptyException  
3  $temp < Q[f]$   
4  $Q[f] \leftarrow null$   
5  $f \leftarrow (f + 1) \bmod N$   
6 return temp
```

Алгоритм `enqueue(o)`:

```
1 if size() =  $N - 1$  then  
2 вызов QueueFullException  
3  $O[r] \leftarrow o$   
4  $r \leftarrow (r + 1) \bmod N$ 
```

Фрагмент кода: Реализация очереди с помощью циклического массива. При этом применяется операция деления по модулю, что позволяет индексировать границы массива, а также используются две переменные экземпляра класса (f и r), которые являются соответственно индексом начала очереди и индексом первой свободной ячейки за концом очереди.

Таким образом, при данной реализации время выполнения каждого метода есть $O(1)$. Как и в случае реализации стека на основе массива,

единственным недостатком подобной же реализации очереди является необходимость искусственно устанавливать ее размер N . В ходе выполнения программы может потребоваться разный объем памяти. Однако при возможности точно определить число элементов, которые одновременно находятся в очереди, реализация очереди на основе массива вполне эффективна.

АТД «очередь» можно реализовать на основе однонаправленного связного списка. Для наибольшей эффективности поместим начало очереди, где можно только удалять элементы, в начало списка, а конец очереди, где можно добавлять элементы, – в конец списка. Необходимы две ссылки на начальный и конечный узлы списка. Каждый метод при реализации очереди на основе однонаправленного связного списка выполняется за время $O(1)$. Кроме того, не нужно указывать максимальный размер очереди, как в случае реализации на основе массива, и это достигается за счет уменьшения пространства, выделяемого для каждого элемента. В то же время методы, используемые при реализации очереди на основе однонаправленного связного списка, являются более сложными, так как необходимо следить за обработкой особых случаев, например, пустая очередь перед операцией enqueue или очередь становится пустой после операции dequeue.

2.1.5. Абстрактный тип данных «дек»

Рассмотрим структуры данных, напоминающие очереди, но в которых добавление и удаление элементов может осуществляться как в начале, так и в конец. Подобный усовершенствованный вариант очереди называется двунаправленной очередью, или деком (deque - double-ended queue).

Абстрактный тип данных «дек» обладает большими возможностями, чем стек или очередь. Этот АТД поддерживает следующие базовые методы (D обозначает дек):

insertFirst (e): помещает новый элемент e в начало D .

Input: объект; **Output:** нет.

insertLast (e): помещает новый элемент e в конец D .

Input: объект; **Output:** нет.

removeFirst (): удаляет и возвращает первый элемент D , если D пуст, выдается сообщение об ошибке.

Input: нет; **Output:** объект.

removeLast (): удаляет и возвращает последний элемент D , если D пуст,

выдается сообщение об ошибке.

Input: нет; **Output:** объект.

Кроме того, дек выполняет следующие дополнительные методы:

first(): возвращает первый элемент D , если D пуст, выдается сообщение об ошибке.

Input: нет; **Output**: объект.

last(): возвращает последний элемент D , если D пуст, выдается сообщение об ошибке.

Input: нет; **Output**: объект,

size (): возвращает число элементов D .

Input: нет; **Output**: целое число.

isEmpty (): определяет, является ли D пустым.

Input: нет; **Output**: логическое значение.

Реализация дека на основе двусвязного списка

Так как дек предусматривает возможность добавления и удаления объектов на обоих концах списка, его реализация на основе однонаправленного связного списка не может быть эффективной. Однако существует тип связного списка, который обладает большими возможностями, в том числе добавлением и удалением элементов на обоих концах списка, которые выполняются за время $O(1)$. Узел двусвязного списка содержит две ссылки – ссылку к следующему узлу списка `next` и ссылку к предыдущему узлу списка `prev`.

Для упрощения программирования рекомендуется добавлять специальные узлы на обоих концах списка: узел `header` перед первым узлом списка и узел `trailer` непосредственно после последнего узла списка. Данные «холостые», или сигнальные узлы не содержат элементов. Узел `header` содержит действительную ссылку `next` и нулевую ссылку `prev`, в то время как узел `trailer` – действительную ссылку `prev` и нулевую ссылку `next`.

Пример такого двусвязного списка вместе с сигнальными узлами представлен на рис. 2.8. Обратите внимание, что объекты связного списка хранят эти сигнальные узлы, а также счетчик `size`, содержащий количество элементов в списке (не считая сигнальных узлов). Добавление или удаление элементов на одном из концов двусвязного списка выполняется за время $O(1)$ (рис. 2.8). Ссылка `prev` устраняет необходимость просмотра всего списка для доступа к предпоследнему узлу.

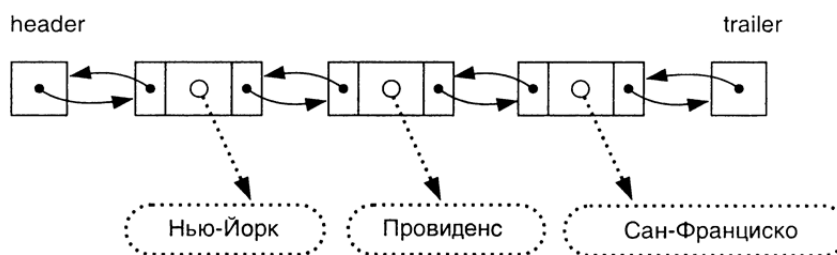


Рис. 2.8. Пример двусвязного списка с сигнальными узлами `header` и `trailer` на концах списка

В пустом списке эти сигнальные узлы будут ссылаться друг к другу. Операции над двусвязным списком с использованием сигнальных узлов header и trailer: (a) добавление элемента в начале списка; состояние после добавления и перед удалением элемента из конца списка; (c) удаление элемента из конца списка; (d) состояние после удаления приведены на рис. 2.9.

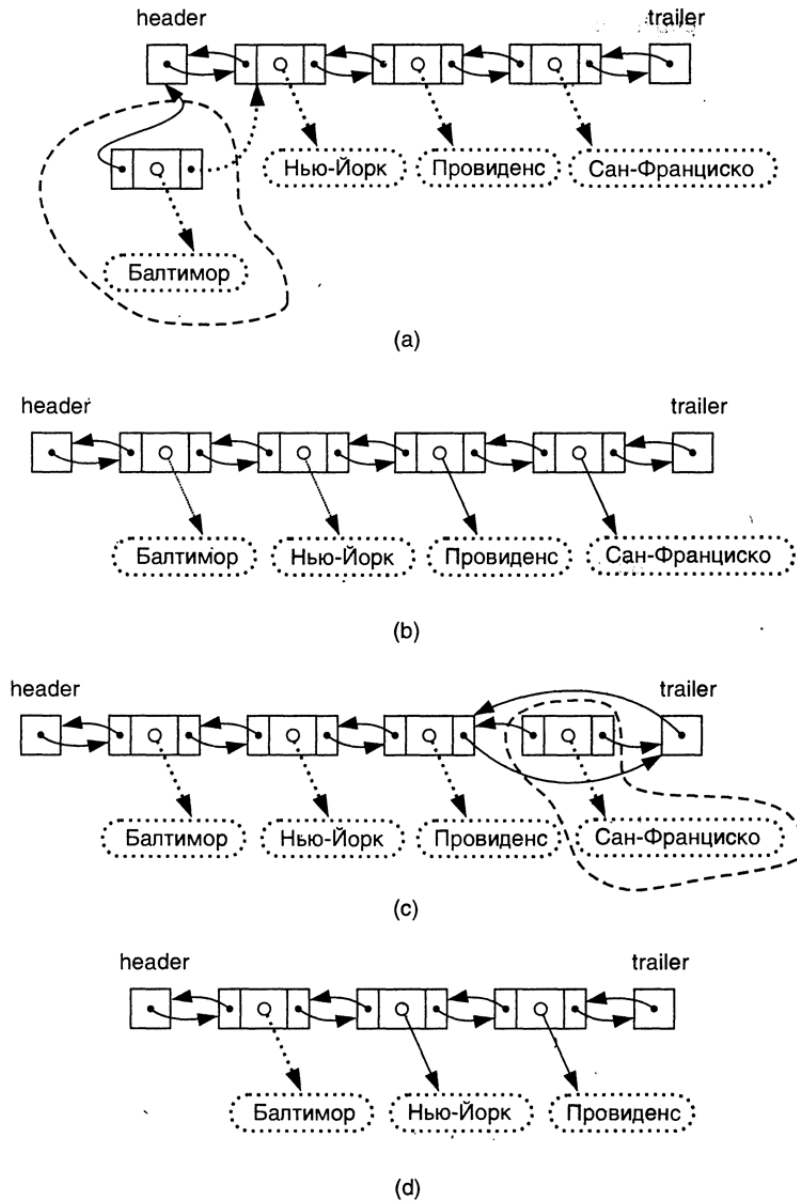


Рис. 2.9.

Двусвязный список позволяет реализовывать все методы АД «дек» за время $O(1)$.

3. ПОИСК

Получение конкретного фрагмента или фрагментов информации из больших томов ранее сохраненных данных – основополагающая операция, называемая поиском, характерная для многих вычислительных задач. Цель поиска[2,3,5] – отыскание элементов с ключами, которые соответствуют заданному ключу поиска. Обычно, назначением поиска является получение доступа к информации внутри элемента (а не просто к ключу) с целью ее обработки, но из полей элемента играет роль ключа

Поиск используется повсеместно и связан с выполнением множества различных операций. Например, в банке требуется отслеживать информацию о счетах всех клиентов и выполнять поиск в этих записях для подведения баланса и выполнения банковских операций. На авиалинии необходимо отслеживать количество мест на каждом рейсе и выполнять поиск свободных мест, отказа в продаже билетов или внесения каких-либо изменений в списки пассажиров. Еще один пример – средство поиска в сетевом интерфейсе программы, которое отыскивает в сети все документы, содержащие заданное ключевое слово.

Так, в словаре английского (или любого другого) языка "ключи" – это слова, а "элементы" – связанные со словами записи, которые содержат определение, правила произношения и другую информацию. Алгоритмы поиска, используемые для отыскания информации в словаре, обычно основываются на алфавитном расположении записей. Телефонные книги, энциклопедии и другие справочники, в основном, организованы таким же образом, и некоторые из рассматриваемых методов поиска также основываются на том, что записи упорядочены.

Мы начнем рассмотрение поисковых задач с алгоритмов нахождения образца в больших массивах информации. Ее можно сформулировать следующим образом: пусть задана строка P из m символов, именуемая образцом или паттерном (pattern), и более длинная строка T из N символов, именуемая текстом (text). Задача о точном совпадении заключается в отыскании всех вхождений образца P в тексте T . Например, если $P = aba$ и $T = bbabaxababay$, то P входит в T , начиная с позиций 3, 7 и 9. Заметим, что два вхождения P могут перекрываться, как это видно по вхождениям P в позициях 7 и 9.

Эта задача возникает в широком спектре приложений, настолько обширном, что всего даже не перечислить. Некоторые из наиболее общих приложений встречаются в текстовых редакторах; в таких утилитах, как `grep` в UNIX; в информационно-поисковых текстовых системах, таких как Яндекс, Google; в поисковых системах библиотечных каталогов, которые в большинстве больших библиотек заменили обычные карточные

каталоги; и интернетовских браузеров и краулерах (поисковый робот), которые просеивают огромные количества текстов в поисках материалов, содержащих данные ключевые слова; в программах, которые читают интернетовские новости и могут отыскивать статьи на требуемую тему; в гигантских цифровых библиотеках, в электронных журналах, которые уже "публикуются" в обслуживании телефонных справочников; в интерактивных энциклопедиях и других средствах обучения на компакт-дисках; в интерактивных словарях и тезаурусах, особенно таких, где организованы перекрестные ссылки (проект Oxford English Dictionary создал электронную интерактивную версию этого словаря, содержащую 50 миллионов слов), и в различных специализированных базах данных[9]. В молекулярной биологии есть несколько сотен специализированных баз данных, содержащих исходные строки ДНК, РНК и аминокислот, а также обработанные образцы (называемые motifs - мотивами), полученные из исходных строк.

3.1. Наивный метод

Почти все обсуждения поиска точного совпадения начинаются с наивного метода, и мы следуем этой традиции. В наивном методе левый конец образца P ставится вровень с левым концом текста T и все символы P сравниваются с соответствующими символами T слева направо, пока либо не встретится несовпадение, либо не исчерпается P . В последнем случае констатируется, что найдено вхождение P в текст. При любом исходе P сдвигается на одну позицию вправо, и сравнения возобновляются, начиная с левого конца P . Процесс продолжается, пока правый конец P не зайдет за правый конец T .

Понятие «сравнения» состоит в том, что существует подстрока строки T , начинающаяся с индекса i и совпадающая с P посимвольно, так что $T[i] = P[0]$, $T[i+1] = P[1]$, ..., $T[i+m-1] = P[m-1]$. То есть $P = T[i...i+m-1]$. Следовательно, результатом алгоритма будет либо указание на то, что в T нет P , либо целое число, обозначающее индекс начала подстроки P в T . Можно также определить все индексы, с которых начинается совпадение подстроки из T со строкой P .

Алгоритм NativeMatch(T, P):

Input: строка T (текст) из n символов и P (образец) из m символов.

Output: индекс начала подстроки, соответствующей P , в строке T или признак

того, что P не является подстрокой T .

1 for $i \leftarrow 0$ to $n - m$ { для каждого подходящего индекса в T } do

2 $j \leftarrow 0$

```

3 while ( $j < m$  and  $T[i + j] = P[j]$ ) do
4  $j \leftarrow j + 1$ 
5 if  $j = m$  then
6 return  $i$ 
7 return "В тексте  $T$  нет образца  $P$ "

```

Упростить алгоритм сопоставления просто невозможно. Он состоит из двух вложенных циклов. Внешний цикл проходит все возможные индексы начала шаблона в тексте, внутренний цикл – все символы шаблона, сравнивая их с потенциальными символами текста.

Нельзя сказать, однако, что время выполнения силового шаблонного сопоставления в наихудших случаях будет очень хорошим, так как для каждого символа T следует выполнить до m сравнений, чтобы выяснить, что P не соответствует T для этого индекса. Из вышеприведенного фрагмента кода видно, что внешний цикл максимально выполняется за $n - m + 1$ времени, а внутренний цикл – за m времени. Следовательно, время выполнения наивного метода составит $O((n - m + 1) m)$, или просто $O(nm)$. Таким образом, в наихудшем случае, когда n и m приблизительно равны, этот алгоритм имеет квадратичное время выполнения. На рис. 3.1 показано выполнение алгоритма.

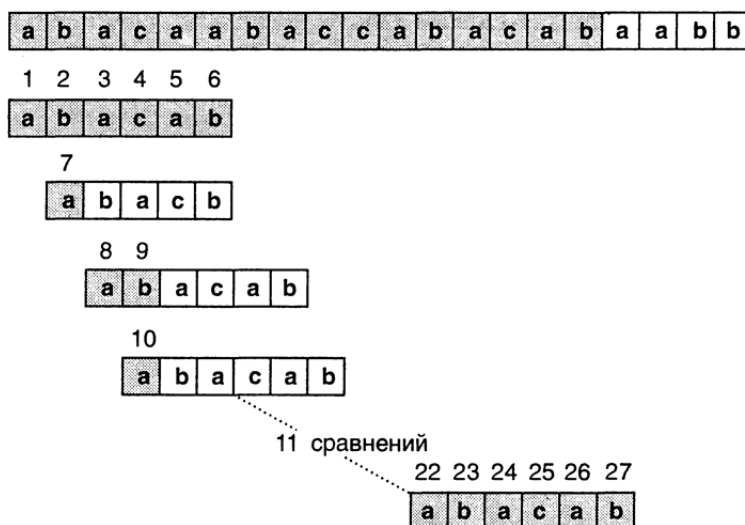


Рис. 3.1. Пример выполнения алгоритма сопоставления. Алгоритм выполняет 27 сравнений символов, отмеченных номерами

Первые идеи ускорения заключались в попытках сдвинуть P при несовпадении больше чем на один символ, но так, чтобы ни в коем случае не пропустить вхождения P в T . Увеличение сдвига экономит сравнения, так как P продвигается вдоль T быстрее. Кроме того, некоторые методы экономят сравнения, пропуская после сдвига какую-то часть образца.

3.2. Алгоритм Кнута-Морриса-Пратта

Самый известный алгоритм с линейным временем для задачи точного совпадения предложен Кнутом, Моррисом и Праттом [6,5,9]. Хотя этот метод редко используется и часто на практике уступает методу Бойера-Мура (и другим), он может быть просто объяснен, и его линейная оценка времени легко обосновывается. Кроме того, он создает основу для известного алгоритма Ахо-Корасика, который эффективно находит все вхождения в текст любого образца из заданного набора образцов.

Предположим, что при некотором перемещении P по тексту T наивный алгоритм обнаружил совпадение первых i символов из P с их парами из T , а при следующем сравнении было несовпадение. В этом случае наивный алгоритм сдвинет P на одно место и начнет сравнение заново с левого конца P . Но часто можно сдвинуть образец на большее число позиций. Например, если $P = abcxabcde$ и при текущем расположении P и T несовпадение нашлось в позиции 8 строки P , то есть возможность (и мы докажем это ниже) сдвинуть P на четыре позиции без пропуска вхождений P в T . Отметим, что это можно увидеть, ничего не зная о тексте T и расположении P относительно T . Требуется только место несовпадения в P . Алгоритм Кнута-Морриса-Пратта, основываясь на таком способе рассуждений, и делает сдвиг больше, чем наивный алгоритм. Теперь формализуем эту идею.

Определение. Для каждой позиции i образца P определим $sp_i(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P .

Иначе говоря, $sp_i(P)$ – это длина наибольшей собственной подстроки $P[1..i]$, которая кончается в i и совпадает с префиксом P . Когда аргумент P ясен из контекста, мы будем вместо полного обозначения использовать sp_i .

Например, если $P = abcaeabcabd$, то $sp_2 = sp_3 = 0$, $sp_4 = 1$, $sp_8 = 3$ и $sp_{10} = 2$. Заметим, что по определению для любой строки $sp_1 = 0$.

Оптимизированный вариант алгоритма Кнута-Морриса-Пратта использует такие значения.

Определение. Для каждой позиции i образца P определим $sp'_i(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P , с дополнительным условием, что символы $P(i + 1)$ и $P(sp'_i - 1)$ не равны.

Алгоритм Кнута-Морриса-Пратта совмещает P с T и затем сравнивает соответствующие пары символов слева направо, как в наивном алгоритме.

Если для любого расположения P и T первое несовпадение (при ходе слева направо) отмечается в позиции $i + 1$ образца P и позиции k текста T , то сдвиг P вправо (относительно T) таков, что $P[1..sp'_i]$ совместится с $T[k - sp'_i..k - 1]$. Другими словами, P сдвинется вправо на $i + 1 - (sp'_i + 1) = i - sp'_i$ позиций, так что символ $sp'_i + 1$ из P совпадет с символом k из T . В случае, если вхождение P найдено (несовпадений нет), P сдвигается на $n - sp'_n$ позиций.

Это правило сдвига гарантирует, что после сдвига P префикс $P[1..sp'_i]$ совпадет с прилегающей подстрокой T . В следующем сравнении будут участвовать символы $T(k)$ и $P[sp'_i + 1]$. Сильное правило сдвига, использующее sp'_i гарантирует, что то же несовпадение не встретится при новом расположении, но не гарантирует, что $T(k) = P[sp'_i + 1]$.

Обсуждаемое правило сдвига имеет два преимущества. Во-первых, P часто сдвигается больше чем на один символ. Во-вторых, после сдвига гарантировано, что левые sp'_i символов P совпадают со своими парами в T . Таким образом, для проверки полного совпадения T с приложенным текстом T надо начинать сравнивать P и T с позиции $sp'_i + 1$ в P (и позиции k в T).

Мы описали алгоритм Кнута-Морриса-Пратта в терминах сдвигов P , но время, требуемое на сами сдвиги, никогда не учитывали. Дело в том, что сдвиг – это принцип, а явным образом P никуда не сдвигается. Просто увеличиваются текущие индексы P и T . Мы используем индекс p позиции в P и один индекс c текущей позиции в T .

Алгоритм Кнута-Морриса-Пратта

1 Обработать (препроцессинг) P , определив $F'(k) = sp'_{k-1} + 1$ для k изменяющимся

2 от 1 до $m + 1$

3 $c \leftarrow 1$

4 $p \leftarrow 1$

5 while $c + (n - p) \leq n$ do

6 while $P(p) = T(c)$ и $p \leq m$ do

7 $p \leftarrow p + 1$

8 $c \leftarrow c + 1$

9 if $p = m + 1$ then

10 зафиксировать вхождение P в T , начиная с позиции $c - m$.

11 if $p = 1$ then $c \leftarrow c + 1$

12 $p \leftarrow F'(P)$

Алгоритм Кнута-Морриса-Пратта (КМП) позволяет добиться времени выполнения, которое пропорционально $O(n + m)$ и является оптимальным для самых сложных ситуаций. Это означает, что в наихудшем случае алгоритм исследует все символы текста и все символы шаблона всего лишь один раз.

3.3. Алгоритм Бойера-Мура

Как и наивный алгоритм, алгоритм Бойера-Мура последовательно передвигает образец P по тексту T и проверяет совпадение символов P с соответствующими символами T . Когда проверка завершается, P сдвигается вправо по T точно так же, в наивном алгоритме. Однако алгоритм Бойера-Мура использует три здравые идеи, которых нет в наивном алгоритме: сравнение символов справа налево, правило сдвига по плохому символу и правило сдвига по хорошему суффиксу. Совместный эффект этих трех идей позволяет методу обычно проверять меньше чем $m + n$ символов (метод с ожидаемым сублинейным временем) и (после некоторого улучшения) работать за линейное время в худшем случае.

Правило плохого символа

Чтобы воспринять саму идею правила плохого символа, предположим, что последний (крайний правый) символ P есть y , а символ в T , с которым он сопоставляется, – это $x \neq y$. Когда случается это начальное несовпадение, то, зная в P крайнюю правую позицию символа x , мы можем смело сдвигать P направо до совмещения крайнего правого x в P с найденным x в T , потому что более короткий сдвиг немедленно даст несовпадение. Так что этот большой сдвиг корректен (в том смысле, что при нем не теряется вхождения P в T). Если x вообще не встречается в P , то мы можем сдвинуть P полностью за точку несовпадения в T . В этих случаях некоторые символы из T вообще не будут проверяться, и метод будет работать за "сублинейное" время. Это наблюдение формализуется ниже.

Определение. Для каждого символа алфавита x пусть $R(x)$ – позиция крайнего правого вхождения x в P . Если x в P не входит, $R(x)$ считается нулем.

Можно просмотреть P за время $O(m)$ и подготовить значения $R(x)$. Мы используем значения R в следующем правиле сдвига по плохому символу.

Предположим, что при некотором сопоставлении P с T крайние правые $m - i$ символов P совпадают со своими парами в T , но следующий слева символ $P(i)$ не совпадает со своей парой, скажем, в позиции k строки T . Правило плохого символа гласит, что P следует сдвинуть

вправо на $\max(1, i - R(T(k)))$ позиций. Таким образом, если крайнее правое вхождение в P символа $T(k)$ занимает позицию $j < i$ (включая возможность $j = 0$), то P сдвигается так, чтобы символ j в P поравнялся с символом k в T . В противном случае, P сдвигается на одну позицию.

Цель этого правила – сдвиг P , когда это возможно, больше чем на одну позицию. После сдвига сравнение P и T начинается заново с правого конца P .

(Сильное) правило хорошего суффикса

Само по себе правило плохого символа имеет репутацию высокоэффективного в практических условиях, в частности для английского текста, но оно оказалось менее эффективным для маленьких алфавитов и не дает линейного времени в худшем случае. Поэтому мы введем еще одно правило, называемое сильным правилом хорошего суффикса.

Наше сильное правило хорошего суффикса таково:

Пусть строка P приложена к T и подстрока t из T совпадает с суффиксом P , но следующий левый символ уже не совпадает. Найдем, если она существует, крайнюю правую копию t' строки t в P , такую что t' не является суффиксом P и символ слева от t' в P отличается от символа слева от t в P . Сдвинем P вправо, приложив подстроку t' в P к подстроке t в T (рис. 2.1). Если t' не существует, то сдвинем левый конец P за левый конец t в T на наименьший сдвиг, при котором префикс сдвинутого образца совпал бы с суффиксом t в T . Если такого сдвига не существует, то сдвинем P на m позиций вправо. Если найдено вхождение P , то сдвинем P на наименьший сдвиг, при котором собственный префикс сдвинутого P совпадает с суффиксом вхождения P в T . Если такой сдвиг невозможен, нужно сдвинуть P на m позиций, т.е. сдвинуть P за t в T .

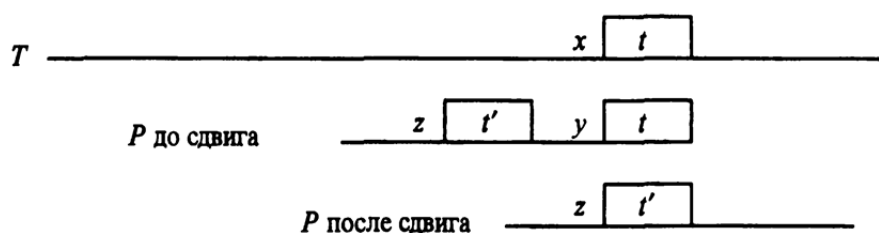


Рис. 3.2. Правило сдвига по хорошему суффиксу, где символ x из T не совпал с символом y из P . Символы y и z из P гарантированно различны по правилу хорошего суффикса, поэтому y и z есть шанс совпасть с x .

Опубликованный первоначально алгоритм Бойера-Мура [75] использует простой, слабый вариант правила хорошего суффикса. Этот вариант требует только, чтобы смещенный образец P прилегал к тексту по строке t , и не требует, чтобы следующий символ влево от вхождения

t был другим, мы будем называть более простое правило слабым правилом хорошего суффикса, а правило, сформулированное выше, сильным правилом хорошего суффикса.

Формализуем теперь препроцессную обработку, требуемую для алгоритма Бойера - Мура. Определение. Для каждого i пусть $L(i)$ – наибольшая позиция, меньшая m и такая, что строка $P[i..m]$ совпадает с суффиксом строки $P[1..L(i)]$. Если такой позиции нет, $L(i)$ считается равным 0. Для каждого пусть $L'(i)$ – наибольшая позиция, меньшая чем m и такая, что $P[i..m]$ совпадает с суффиксом $P[1..L'(i)]$, а символ, предшествующий этому суффиксу, не равен $P(i - 1)$. Если такой позиции нет, $L'(i)$ считается равным 0.

Например, если $P = \text{cabdabdab}$, то $L(8) = 6$ и $L'(8) = 3$.

$L(i)$ определяет позицию правого конца крайней правой копии $P[i..m]$, которая сама не является суффиксом P , а $L'(i)$ – такую же позицию с дополнительным усиливающим условием, что предшествующий символ не равен $P(i - 1)$. Поэтому в варианте сильного сдвига алгоритма Бойера-Мура если символ $i - 1$ образца P участвовал в обнаружении несовпадения и $L'(i) > 0$, то P сдвигается вправо на $m - L'(i)$ позиций. В результате если правый конец P до сдвига стоял на уровне позиции k в T , то на уровне k теперь нужно выровнять позицию $L'(i)$.

Препроцессинг для алгоритма Бойера-Мура вычисляет $L'(i)$ (и $L(i)$, если нужно) для каждой позиции i в P . Это делается за время $O(m)$. Пусть $N_j(P)$ - длина наибольшего суффикса подстроки $P[1..j]$, который является также суффиксом полной строки P .

Например, если $P = \text{cabdabdab}$, то $N_3(P) = 2$ и $N_6(P) = 5$.

Теорема. $L(i)$ – наибольший индекс j , меньший m и такой, что $N_j(P) \geq |P[i..m]| = m - i + 1$. $L'(i)$ – наибольший индекс j , меньший n и такой, что $N_j(P) = |P[i..m]| = (m - i + 1)$.

Из этой теоремы прямо следует, что все значения $L'(i)$ могут быть получены за

линейное время из значений N использованием следующего алгоритма:

Метод Бойера-Мура, основанный на Z -алгоритме (Напомним, что $Z_i(S)$ – длина наибольшей подстроки S , начинающейся в i и совпадающей с префиксом S . Ясно, что функция N является обращенной к Z).

```
for  $i \leftarrow 1$  to  $n$  do  $L'(i) \leftarrow 0$ 
```

```
for  $j \leftarrow 1$  to  $n - 1$  do
```

```
   $i \leftarrow m - N_j(P) + 1$ 
```

```
   $L'(i) \leftarrow j$ 
```

Значения $L(i)$ (если потребуется) могут быть получены добавлением к этому псевдокоду следующих строк:

```
 $L(2) \leftarrow L'(2)$   
for  $i \leftarrow 3$  to  $n$  do  $L(i) \leftarrow \max(L(i - 1), L'(i))$ 
```

Предварительный этап должен подготовить данные и к случаю, когда $L'(i) = 0$ или когда найдено вхождение P . Эту обработку обеспечивают следующие определение и теорема.

Определение. Пусть $l'(i)$ обозначает длину наибольшего суффикса $P[i..m]$, который является префиксом P , если такой существует. Если же не существует, то $l'(i)$ равно нулю.

Теорема. $l'(i)$ равно наибольшему $j \leq |P[i..m]| = m - i + 1$, для которого $N_j(p) = j$.

Можно сформулировать правило хорошего суффикса на стадии поиска в алгоритме Бойера-Мура. Вычисленные заранее значения $L'(i)$ и $l'(i)$ для каждой позиции i в P используются на стадии поиска для увеличения сдвигов. Если при поиске обнаружилось несовпадение в позиции $i - 1$ строки P и $L'(i) > 0$, то правило хорошего суффикса сдвигает P на $n - L'(i)$ мест вправо, так что префикс сдвинутого P длины $L'(i)$ пристраивается к суффиксу длины $L'(i)$ несдвинутого P . В случае $L'(i) = 0$ правило хорошего суффикса смещает P на $m - l'(i)$ позиций. Когда обнаруживается вхождение P , это правило сдвигает P на $m - l'(2)$ мест. Заметьте, что правила работают корректно и при $l'(i) = 0$.

Остался один специальный случай. Когда первое сравнение дает несовпадение (т.е. не совпал символ $P(m)$), то P надо сдвинуть на одно место вправо.

Полный алгоритм Бойера-Мура

Мы увидели, что ни правило хорошего суффикса, ни правило плохого символа не пропускают при сдвиге P его вхождений. Поэтому алгоритм Бойера-Мура сдвигает образец на наибольшее из расстояний, предлагаемых обоими правилами. Мы можем теперь представить весь алгоритм.

```
Алгоритм Бойера-Мура  
//Препроцессинг  
//Задан образец  $P$ .  
//Вычислить  $L'(i)$  и  $l'(i)$  для каждой позиции  $i$  из  $P$ ,  
//а также  $R(x)$  для каждого символа  $x$  из  $\Sigma$ .  
//Стадия поиска  
1  $k \leftarrow m$   
2 while  $k \leq n$  do  
3  $i \leftarrow m$   
4  $h \leftarrow k$ 
```

```

5 while  $i > 0$  и  $P(i) = T(h)$  do
6  $i \leftarrow i - 1$ 
7  $h \leftarrow h - 1$ 
8 if  $i = 0$  then
9 зафиксировать вхождение  $P$  в  $T$  с последней позицией  $k$ .
10  $k \leftarrow k + m - l'(2)$ 
11 else
12 сдвинуть  $P$  (увеличить  $k$ ) на максимальную из величин, задаваемых (расширенным) правилом плохого символа и правилом хорошего суффикса,

```

Заметим, что, хотя мы всегда говорим о "сдвиге P " и даем правила для определения того, насколько P должно быть "сдвинуто", физически ничто не сдвигается. Просто индекс k увеличивается до значения, которое соответствует правому концу "сдвинутого" P . Следовательно, каждое действие сдвига P выполняется за константное время.

Можно показать, что использование только сильного правила хорошего суффикса в методе Бойера-Мура дает в наихудшем случае время счета $O(n)$ в предположении, что образец в тексте не встречается. Это было впервые доказано Кнутом, Моррисом и Праттом.

Когда образец в тексте не встречается, исходный метод Бойера-Мура работает в худшем случае за время $O(nm)$.

3.4. Алгоритм Рабина-Карпа

Рабин (Rabin) и Карп (Karp) [6] предложили алгоритм поиска подстроки, показывающий на практике хорошую производительность, а также допускающий обобщения на другие родственные задачи, такие как задача о сопоставлении двумерного образца. В алгоритме Рабина-Карпа время $O(m)$ затрачивается на предварительную обработку, а время его работы в наихудшем случае равно $O((n-m+1)m)$.

Однако с учетом определенных предположений удастся показать, что среднее время работы этого алгоритма оказывается существенно лучше.

В этом алгоритме используются обозначения из элементарной теории чисел, такие как эквивалентность двух чисел по модулю третьего числа.

Для простоты предположим, что $\Sigma = \{0, 1, \dots, 9\}$, т.е. каждый символ – это десятичная цифра. (В общем случае можно предположить, что каждый символ – это цифра в системе счисления с основанием d , где $d = |\Sigma|$). После этого строку из k последовательных символов можно рассматривать как число длиной k .

Таким образом, символьная строка '31415' соответствует числу 31415. При такой двойной интерпретации входных символов и как графических знаков, и как десятичных чисел, в этом разделе удобнее подразумевать под ними цифры, входящие в стандартный текстовый шрифт.

Для заданного образца $P [1..m]$ обозначим через p соответствующее ему десятичное значение. Аналогично, для заданного текста $T [1..n]$ обозначим через t_s десятичное значение подстроки $T [s + 1..s + m]$ длиной m при $s = 0, 1, \dots, n - m$. Очевидно, что $t_s = p$ тогда и только тогда, когда $T [s + 1..s + m] = P [1..m]$; таким образом, s – допустимый сдвиг тогда и только тогда, когда $t_s = p$. Если бы значение p можно было вычислить за время $O(m)$, а все значения t_s – за суммарное время $O(n - m + 1)$, то значения всех допустимых сдвигов можно было бы определить за время $O(m) + O(n - m + 1) = O(n)$ путем сравнения значения p с каждым из значений t_s . (Отметим, что величины p и t_s могут оказаться очень большими числами.)

С помощью правила Горнера величину p можно вычислить за время $O(m)$:

$$p = P [m] + 10 (P [m - 1] + 10 (P [m - 2] + \dots + 10 (P [2] + 10P [1]) \dots)).$$

Значение t_0 можно вычислить из массива $T [1..m]$ за время $O(m)$ аналогичным способом. Чтобы вычислить остальные значения t_1, t_2, \dots, t_{n-m} за время $O(n - m)$, достаточно заметить, что величину t_{s+1} можно вычислить из величины t_s за фиксированное время, так как

$$t_{s+1} = 10 (t_s - 10^{m-1} T [s + 1]) + T [s + m + 1].$$

Например, если $m = 5$ и $t_s = 31415$, то нужно удалить цифру в старшем разряде $T [s + 1] = 3$ и добавить новую цифру в младший разряд (предположим, это цифра $T [s + 5 + 1] = 2$). В результате мы получаем $t_{s+1} = 10 (31415 - 10000 \cdot 3) + 2 = 14152$.

Чтобы удалить из числа t_s цифру старшего разряда, из него вычитается значение $10^{m-1} T [s + 1]$; путем умножения на 10 число сдвигается на одну позицию влево, а в результате добавления элемента $T [s + m + 1]$ в его младшем разряде появляется нужная цифра. Если предварительно вычислить константу 10^{m-1} . Таким образом, число p можно вычислить за время $O(m)$, величины t_1, t_2, \dots, t_{n-m} за время $O(n - m + 1)$, а все вхождения образца $P [1..m]$ в текст $T [1..n]$ можно найти, затратив на фазу предварительной обработки время $O(m)$, а на фазу сравнения – время $O(n - m + 1)$.

Единственная сложность, возникающая в этой процедуре, может быть связана с тем, что значения p и t_s могут оказаться слишком большими и с ними будет неудобно работать. Если образец P содержит m

символов, то предположение о том, что каждая арифметическая операция с числом p (в которое входит m цифр) занимает "фиксированное время", не отвечает действительности. К счастью, эта проблема имеет простое решение вычислять значения p и t_s по модулю некоторого числа q . Поскольку вычисление величин p , t_0 и рекуррентного соотношения можно производить по модулю q , получается, что величина p по модулю q вычисляется за время $O(m)$, а вычисление всех величин t_s по модулю q – за время $O(n - m + 1)$. В общем случае, если имеется d -символьный алфавит $\{0, 1, \dots, d - 1\}$, значение q выбирается таким образом, чтобы длина величины dq не превышала длины компьютерного слова, и чтобы с рекуррентным соотношением было удобно работать по модулю q . После этого рассматриваемое соотношение принимает вид

$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q,$$

где $h \equiv d^{m-1} \pmod{q}$ – значение, которое приобретает цифра «1», помещенная в старший разряд m -цифрового текстового окна. Работа алгоритма Рабина-Карпа проиллюстрирована на рис. 3.4. Каждый символ здесь представлен десятичной цифрой, а вычисления производятся по модулю 13. В части а приведена строка текста. Подстрока длиной 5 символов выделена серым цветом. Находится численное значение выделенной подстроки по модулю 13, в результате чего получается значение 7. В части б изображена та же текстовая строка, в которой для всех возможных 5-символьных подстрок вычислены соответствующие им значения по модулю 13. Если предположить, что образец имеет вид $P = 31415$, то нужно найти подстроки, которым соответствуют значения 7 по модулю 13, поскольку $31415 \equiv 7 \pmod{13}$. Найдены две такие подстроки; они выделены серым цветом. Первая подстрока, которая начинается в тексте на позиции 7, действительно совпадает с образцом, в то время как вторая, которая начинается в позиции 13, представляет собой т.н. ложное совпадение. В части в показано, как в течение фиксированного времени вычислить значение, соответствующее данной подстроке, по значению предыдущей подстроки. Значение, соответствующее первой подстроке, равно 31415. Если отбросить цифру 3 в старшем разряде, произвести сдвиг числа влево (умножение на 10), а затем добавить к полученному результату цифру 2 в младшем разряде, то получим новое значение 14152. Однако все вычисления производятся по модулю 13, поэтому для первой подстроки получится значение 7, а для второй – значение 8.

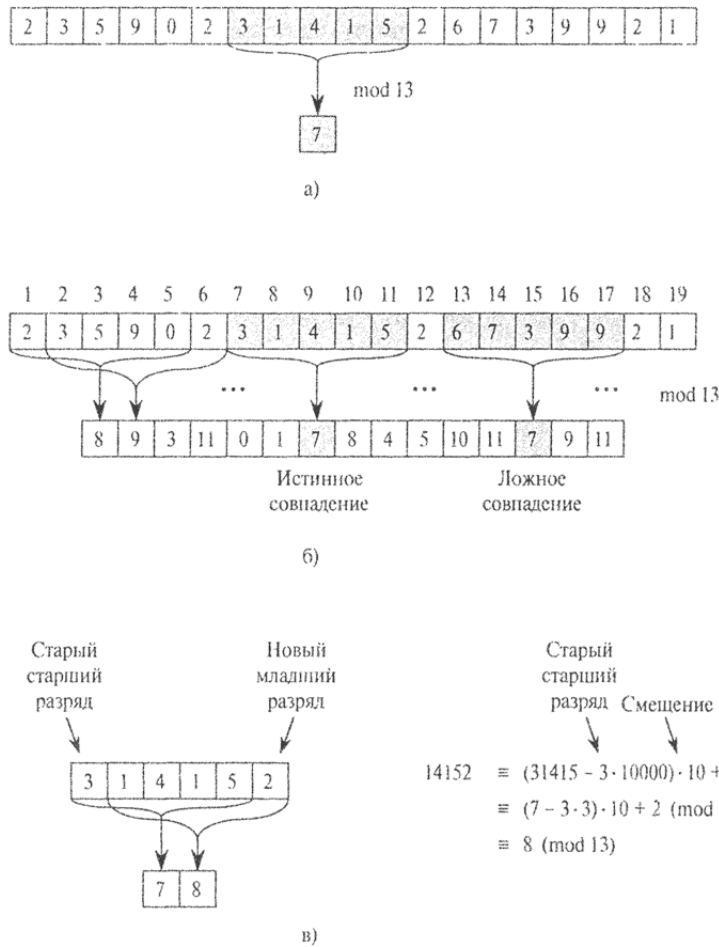


Рис. 3.4. Алгоритм Рабина-Карна

Итак, как видим, идея работы по модулю q не лишена недостатков, поскольку из $t_s \equiv p \pmod{q}$ не следует, что $t_s = p$. С другой стороны, если $t_s \not\equiv p \pmod{q}$, то обязательно выполняется соотношение $t_s \neq p$ можно сделать вывод, что сдвиг s недопустимый. Таким образом, соотношение $t_s \equiv p \pmod{q}$ можно использовать в качестве быстрого эвристического теста, позволяющего исключить недопустимые сдвиги s . Все сдвиги, для которых справедливо соотношение $t_s \equiv p \pmod{q}$, необходимо подвергнуть дополнительному тестированию, чтобы проверить, что действительно ли сдвиг s является допустимым, или это просто ложное совпадение. Такое тестирование можно осуществить путем явной проверки условия $P[l..m] = T[s + l..s + m]$. Если значение q достаточно большое, то можно надеяться, что ложные совпадения встречаются довольно редко и стоимость дополнительной проверки окажется низкой.

Сформулированная ниже процедура поясняет описанные выше идеи. В роли входных значений для нее выступает текст T , образец P ,

разряд d (в качестве значения которого обычно выбирается $|\Sigma|$) и простое число q .

```

RABIN_KARP_MATCHER( $T, P, d, q$ )
1   $n \leftarrow \text{length}[T]$ 
2   $m \leftarrow \text{length}[P]$ 
3   $h \leftarrow d^{m-1} \bmod q$ 
4   $p \leftarrow 0$ 
5   $t_0 \leftarrow 0$ 
6  for  $i \leftarrow 1$  to  $m$            ▷ Предварительная обработка
7      do  $p \leftarrow (dp + P[i]) \bmod q$ 
8          $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ 
9  for  $s \leftarrow 0$  to  $n - m$      ▷ Проверка
10     do if  $p = t_s$ 
11         then if  $P[1..m] = T[s + 1..s + m]$ 
12             then print "Образец обнаружен при сдвиге"  $s$ 
13     if  $s < n - m$ 
14         then  $t_{s+1} \leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$ 

```

Опишем работу процедуры Rabin_Karp_Matcher. Все символы интерпретируются как цифры в системе счисления по основанию d . Индексы переменной t приведены для ясности; программа будет правильно работать и без них. В строке 3 переменной h присваивается начальное значение, равное цифре, расположенной в старшем разряде m -цифрового текстового окна. В строках 4-8 вычисляется значение p , равное $P[1..m] \bmod q$, и значение t_0 , равное $T[1..m] \bmod q$. В цикле *for* в строках 9-14 производятся итерации по всем возможным сдвигам s . При этом сохраняется сформулированный ниже инвариант.

При каждом выполнении строки 10 справедливо соотношение

$$t_s = T[s + 1..s + m].\bmod q.$$

Если в строке 10 выполняется условие $p = t_s$ ("совпадение"), то в строке 11 проверяется справедливость равенства $P[1..m] = T[s + 1..s + m]$, чтобы исключить ложные совпадения. Все обнаруженные допустимые сдвиги выводятся в строке 12. Если $s < n - m$ (это неравенство проверяется в строке 13), то цикл *for* нужно будет выполнить хотя бы еще один раз, поэтому сначала выполняется строка 14, чтобы гарантировать соблюдение инварианта цикла, когда мы снова перейдем к строке 10. В строке 14 на основании значения $t_s \bmod q$ с использованием уравнения C2.2) в течение фиксированного интервала времени вычисляется величина $t_{s+1} \bmod q$.

В процедуре Rabin_Karp_Matcher на предварительную обработку затрачивается время $O(m)$, а время сравнения в нем в наихудшем случае равно $O((n - m + 1)m)$, поскольку в алгоритме Рабина-Карпа (как и в

простейшем алгоритме поиска подстрок) явно проверяется допустимость каждого сдвига. Если $P = a^m$ и $T = a^n$, то проверка займет время $O((n - m + 1) m)$, поскольку все $n - m + 1$ возможных сдвигов являются допустимыми.

Во многих приложениях ожидается небольшое количество допустимых сдвигов (возможно, выражающееся некоторой константой c); в таких приложениях математическое ожидание времени работы алгоритма равно сумме $O((n - m + 1) + cm) = O(n + m)$ и времени, необходимого для обработки ложных совпадений. В основу эвристического анализа можно ожидать, что число ложных совпадений равно $O(n/q)$, потому что вероятность того, что произвольное число t_s будет эквивалентно p по модулю q , можно оценить как $1/q$. Поскольку имеется всего $O(n)$ позиций, в которых проверка в строке 10 дает отрицательный результат, а на обработку каждого совпадения затрачивается время $O(m)$, математическое ожидание времени сравнения в алгоритме Рабина-Карпа равно

$$O(n) + O(m(v + n/q))$$

где v – количество допустимых сдвигов. Если $v = O(1)$, а q выбрано так, что $q \geq m$ приведенное выше время выполнения равно $O(n)$. Другими словами, если математическое ожидание количества допустимых сдвигов мало ($O(1)$), а выбранное простое число q превышает длину образца, то можно ожидать, что для выполнения фазы сравнения процедуре Рабина-Карпа потребуется время $O(n + m)$. Поскольку $m < n$, то математическое ожидание времени сравнения равно $O(n)$.

4. СОРТИРОВКА

На практике сортируемые числа редко являются изолированными значениями. Обычно каждое из них входит в состав набора данных, который называется записью (record). В каждой записи содержится ключ (key), представляющий собой сортируемое значение, в то время как остальная часть записи нередко состоит из сопутствующих данных, дополняющих ключ. Алгоритм сортировки на практике должен быть реализован так, чтобы он вместе с ключами переставлял и сопутствующие данные. Если каждая запись включает в себя сопутствующие данные большого объема, то с целью свести к минимуму перемещение данных сортировка часто производится не в исходном массиве, а в массиве указателей на записи.

В определенном смысле сделанное выше замечание относится к особенностям реализации, отличающим алгоритм от конечной про-

граммы. Метод, с помощью которого процедура сортировки размещает сортируемые величины в нужном порядке, не зависит от того, сортируются ли отдельные числа или большие записи.

Таким образом, если речь идет о задаче сортировки, обычно предполагается, что входные данные состоят только из чисел. Преобразование алгоритма, предназначенного для сортировки чисел, в программу для сортировки записей представляет концептуальных трудностей, хотя в конкретной практической ситуации иногда возникают различные тонкие нюансы, усложняющие задачу программиста.

Многие исследователи [2,7,10.11], работающие в области вычислительной техники, считают сортировку наиболее фундаментальной задачей при изучении алгоритмов. Иногда в приложении не обойтись без сортировки информации. Например, чтобы подготовить отчет о состоянии счетов клиентов, банку необходимо выполнить сортировку чеков по их номерам. Часто в алгоритмах сортировка используется в качестве основной подпрограммы. Например, программе, выполняющей визуализацию перекрывающихся графических объектов, которые находятся на разных уровнях, сначала может понадобиться отсортировать эти объекты по уровням снизу вверх, чтобы установить порядок их вывода.

Имеется большой выбор алгоритмов сортировки, в которых применяются самые разные технологии. Фактически в алгоритмах сортировки используются многие важные методы (зачастую разработанные еще на заре компьютерной эры), применяемые при разработке различных классов алгоритмов.

В этом отношении задача сортировки представляет также исторический интерес. В процессе реализации алгоритмов сортировки на передний план выходят многие прикладные проблемы. Выбор наиболее производительной программы сортировки в той или иной ситуации может зависеть от многих факторов, таких как предварительные знания о ключах и сопутствующих данных, об иерархической организации памяти компьютера (наличии кэша и виртуальной памяти) и программной среды. Многие из этих вопросов можно рассматривать на уровне алгоритмов, а не кода.

В общем виде задачу сортировки можно сформулировать следующим образом [3,4,13]: имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (ключ сортировки). Тип данных ключа должен включать операции сравнения ("=", ">", "<", ">=" и "<="). Требуется преобразовать исходную

последовательность в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа.

Мы будем различать два типа сортировки:

- внутренняя сортировка, в которой предполагается, что данные находятся в оперативной памяти, и важно оптимизировать число действий программы (для методов, основанных на сравнении, число сравнений, обменов элементов и пр.)

- внешняя, в которой данные хранятся на внешнем устройстве с медленным доступом (магнитные лента, барабан, диск) и прежде всего надо снизить число обращений к этому устройству.

На рис. приведена классификация методов сортировки.

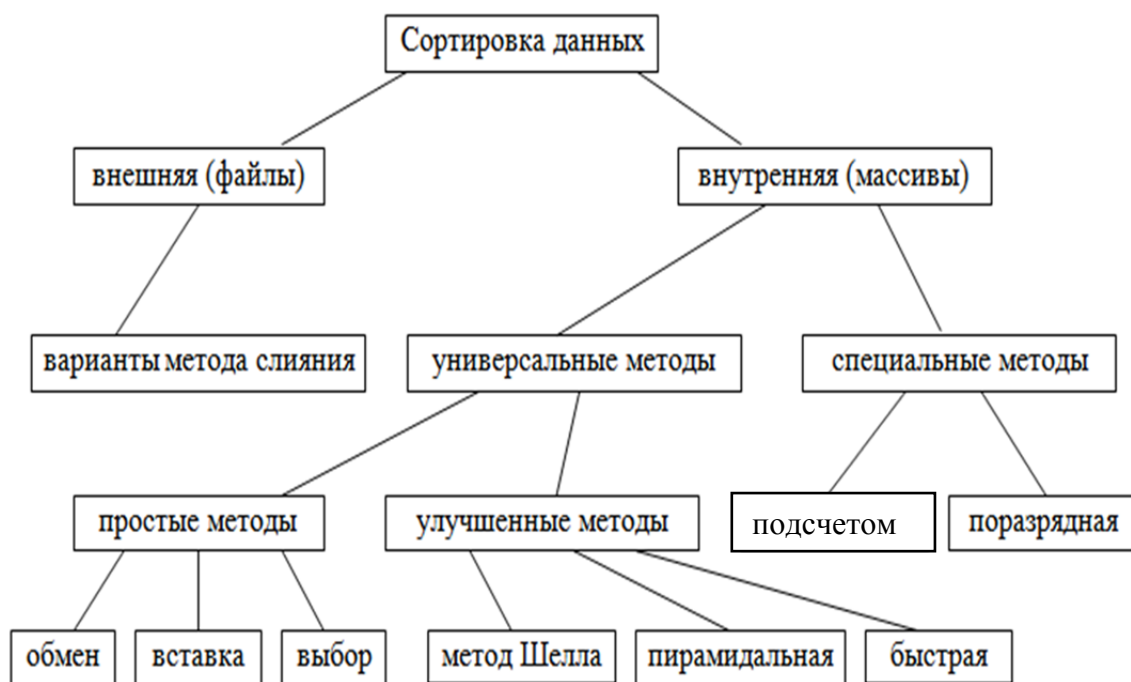


Рис.4.1

4.1. Сортировка подсчетом

Начнем рассмотрение с алгоритмов внутренней сортировки с сортировки подсчетом, которая требует некоторую специфику представления исходных данных. В сортировке подсчетом (counting sort) предполагается, что все n входных элементов – целые числа, принадлежащие интервалу от 0 до k , где k – некоторая целая константа. Основная идея сортировки подсчетом заключается в том, чтобы для каждого рассматриваемого элемента x определить количество элементов, которые меньше x . С помощью этой информации элемент x можно разместить на

той позиции выходного массива, где он должен находиться. Исходный массив – $A[1..n]$, результат – $B[1..n]$, вспомогательный массив – $C[0..k]$.

COUNTING_SORT(A, B, k)

1 for $i \leftarrow 0$ to k

2 do $C[i] \leftarrow 0$

3 for $j \leftarrow 1$ to $\text{length}[A]$

4 do $C[A[j]] \leftarrow C[A[j]] + 1$

5 // В $C[i]$ хранится количество элементов, равных i .

6 for $i \leftarrow 1$ to k

7 do $C[i] \leftarrow C[i] + C[i - 1]$

8 // В $C[i]$ – количество элементов, не превышающих i .

9 for $j \leftarrow \text{length}[A]$ down to 1

10 do $B[C[A[j]]] \leftarrow A[j]$

11 $C[A[j]] \leftarrow C[A[j]] - 1$

После инициализации в цикле for в строках 1-2, в цикле for в строках 3-4 выполняется проверка каждого входного элемента. Если его значение равно i , то к величине $C[i]$ прибавляется единица. Таким образом, после выполнения строки 4 для каждого $i = 0, 1, \dots, k$ в переменной $C[i]$ хранится количество входных элементов, равных i . В строках 6-7 для каждого $i = 0, 1, \dots, k$ определяется число входных элементов, не превышающих i .

Наконец, в цикле for в строках 9-11 каждый элемент $A[j]$ помещается в надлежащую позицию выходного массива B . Если все n элементов различны, то при первом переходе к строке 9 для каждого элемента $A[j]$ в переменной $C[A[j]]$ хранится корректный индекс конечного положения этого элемента в выходном массиве, поскольку имеется $C[A[j]]$ элементов, меньших или равных $A[j]$. Поскольку разные элементы могут иметь одни и те же значения, помещая значение $A[j]$ в массив B , мы каждый раз уменьшаем $C[A[j]]$ на единицу. Благодаря этому следующий входной элемент, значение которого равно $A[j]$ (если таковой имеется), в выходном массиве размещается непосредственно перед элементом $A[j]$.

Сколько времени требуется для сортировки методом подсчета? На выполнение цикла for в строках 1-2 затрачивается время $O(k)$, на выполнение цикла for в строках 3-4 – время $O(n)$, цикл в строках 6-7 требует $O(k)$ времени, а цикл в строках 9-11 – $O(n)$. Таким образом, полное время можно записать как $O(k + n)$ отсюда временная сложность работы алгоритма равна $O(n)$.

4.2. Сортировка включением (вставками)

Массив делится на 2 части: отсортированную и неотсортированную. На каждом шаге берется очередной элемент из неотсортированной части и включается в отсортированную. Простое включение предполагает, что отсортировано начало массива A_1, A_2, \dots, A_{i-1} , остаток массива A_i, \dots, A_n – неотсортирован. На очередном шаге A_i включается в отсортированную часть на соответствующее место. Рис. иллюстрирует пошаговое выполнение алгоритма:

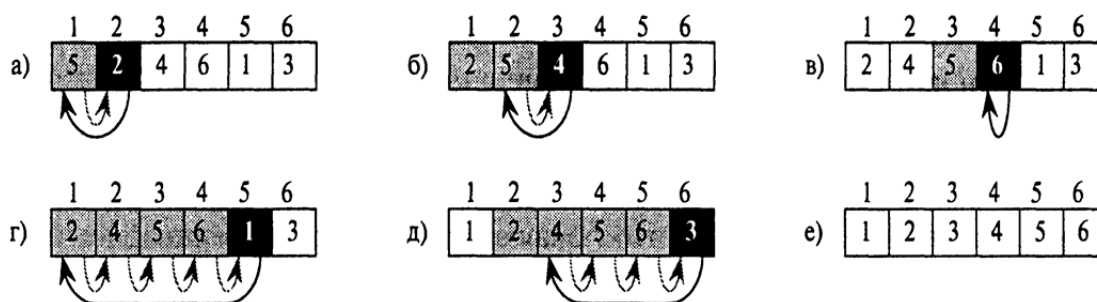


Рис. 4.2.

Описание алгоритма:

INSERTION_SORT (A)

1 for $j \leftarrow 2$ to length (A)

2 do $key \leftarrow A[j]$

3 // Вставка элемента $A[j]$ в отсортированную последовательность $A[1..j-1]$

4 $i \leftarrow j-1$

5 while $i > 0$ и $A[i] > key$

6 do $A[i+1] \leftarrow A[i]$

7 $i \leftarrow i-1$

8 $A[i+1] \leftarrow key$

Алгоритм имеет сложность $O(n^2)$, но в случае исходно отсортированного массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет в этом случае временную сложность $O(n)$.

4.3. Сортировка Шелла (Donald Shell, 1959г.)

Сортировка Шелла является модификацией алгоритма сортировки включением, которая состоит в следующем: вместо включения $A[i]$ в подмассив предшествующих ему элементов, его включают в подмассив, содержащий элементы $A[i-h]$, $A[i-2h]$, $A[i-3h]$ и тд, где h – по-

ложительная константа. Таким образом, формируется массив, в котором « h - серии» элементов, отстоящих друг от друга на h , сортируются отдельно.

Процесс возобновляется с новым значением h , меньшим предыдущего. И так до тех пор, пока не будет достигнуто значение $h=1$.

Для достаточно больших массивов рекомендуемой считается такая последовательность, что

$$h_{i+1}=3h_i, \text{ а } h_1=1.$$

Начинается процесс с h_{m-2} , h_{m-2} – первый такой член последовательности, что

$$h_{m-2} \geq [n/9].$$

Ниже приведены две последовательности для определения числа серий:

$$1, 4, 13, 40, 121 \dots (h_{i+1}=3h_i)$$

$$1, 3, 7, 15, 31 (h_{i+1}=2h_i).$$

Опишем алгоритм метода Шелла

SHELL_SORT (A)

```

1  $h \leftarrow 1$ 
2 while  $h < N \text{ div } 9$  do  $h \leftarrow h*3 + 1$ 
3 repeat // цикл по сериям
4 for  $k \leftarrow 1 \dots h$  do {сортировка  $k$ -ой серии}
5  $i \leftarrow h + k$ 
6 while  $i \leq N$  do //включение  $A[i]$  на свое место в серии
7  $x \leftarrow A[i]$ 
8  $j \leftarrow i - h$ ;
9 while  $(j \geq 1)$  and  $(A[j] > x)$  do //сдвиг
10  $A[j + h] \leftarrow A[j]$ 
11  $j \leftarrow j - h$ 
12  $A[j + h] \leftarrow x$ 
13  $i \leftarrow i + h$ 
14  $h \leftarrow h \text{ div } 3$  {переход к новой серии}
15 while  $h > 0$ 
```

Р. Сэдзвик [10] предложил другой способ определения последовательности числа серий:

$$h_s = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{если } s \text{ четно;} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{если } s \text{ нечетно.} \end{cases}$$

Отсюда получаем последовательность $(h_0, h_1, h_2, \dots) = (1, 5, 19, 41, 109, 209, \dots)$, конец последовательности – h_{t-1} если $3h_t \geq n$.

Временная сложность для алгоритма Шелла – $O(n^{4/3})$ и $\Theta(n^{7/6})$, среднее число перемещений $\sim 1,66n^{1,25}$.

Количество перестановок элементов по результатам экспериментов со случайным массивом иллюстрируется следующей таблицей.

Размерность	n = 25	n = 1000	n = 100000
Сортировка Шелла	50	7700	2 100 000
Сортировка простыми вставками	150	240 000	2.5 млрд.

4.4. Сортировка извлечением (выбором)

Как и в предыдущем алгоритме массив делится на уже отсортированную часть

$$A_{i+1}, A_{i+2} \dots A_n$$

и неотсортированную

$$A_1, A_2, \dots, A_i$$

На каждом шаге извлекается максимальный элемент из неотсортированной части и ставится в начало отсортированной части.

Опишем алгоритм сортировки извлечением.

EXTRACTION_SORT (A)

1 for $i \leftarrow n$ downto 2 do

2 $MaxIndex \leftarrow 1$

3 for $j \leftarrow 2$ to i do

4 if $A[j] > A[MaxIndex]$ then $MaxIndex \leftarrow j$

5 $Tmp \leftarrow A[i]$

6 $A[i] \leftarrow A[MaxIndex]$

7 $A[MaxIndex] \leftarrow Tmp$

Очевидный второй вариант реализации этого метода – последовательность создается, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная с первого и заканчивая $(n-1)$. На i -м шаге выбираем наименьший из элементов $A[i] \dots A[n]$ и меняем его местами с $A[i]$. На рисунке показаны последовательные шаги алгоритма по второму варианту.

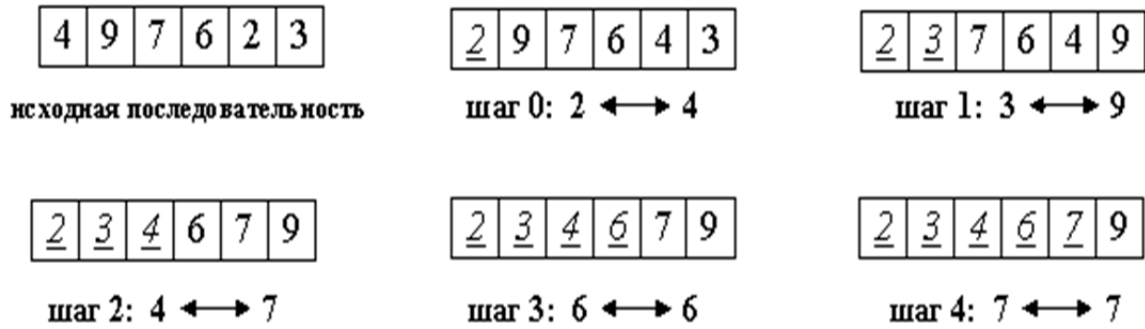


Рис.4.3.

С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

$$n + (n-1) + (n-2) + (n-3) + \dots + 1 = 1/2 * (n^2 - n) = O(n^2).$$

Естественной идеей улучшения алгоритма выбором является идея использования информации, полученной при сравнении элементов при поиске максимального (минимального) элемента на предыдущих шагах.

В общем случае, если n – точный квадрат, можно разделить массив на \sqrt{n} групп по \sqrt{n} элементов и находить максимальный элемент в каждой подгруппе.

Любой выбор, кроме первого, требует не более чем $\sqrt{n} - 2$ сравнений внутри группы ранее выбранного элемента плюс $\sqrt{n} - 1$ сравнений среди "лидеров групп".

Этот метод получил название квадратичный выбор общее время его работы составляет порядка $O(n \sqrt{n})$ что существенно лучше, чем $O(n^2)$.

Пример выполнения первых шагов алгоритма приведен на рис. 4.4.

503 087 512 061 | 908 170 897 275 | 653 426 154 509 | 612 677 765 703

512, 908, 653, 765.

512, 897, 653, 765

512, 275, 653, 765

Рис.4.4.

4.5. Пирамидальная сортировка

Пирамида (binary heap) – это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное

бинарное дерево[5,7]. Каждый узел этого дерева – соответствует определенному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненный уровень – это такой, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы. Представляющий пирамиду массив A является объектом с двумя атрибутами: $\text{length}(A)$, т.е. количество элементов массива, и $\text{heap_size}(A)$, т.е. количество элементов пирамиды, содержащихся в массиве A . Другими словами, несмотря на то, что в массиве A $[1.. \text{length}(A)]$ все элементы могут быть корректными числами, ни один из элементов, следующих после элемента A $[\text{heap_size}(A)]$, где $\text{heap_size}(A) \leq \text{length}(A)$, не является элементом пирамиды. В корне дерева находится элемент A $[1]$, а дальше оно строится по следующему принципу: если какому-то узлу соответствует индекс i , то индекс его родительского узла вычисляется с помощью представленной ниже процедуры $\text{Parent}(i)$, индекс левого дочернего узла – с помощью процедуры $\text{Left}(i)$, а индекс правого дочернего узла – с помощью процедуры $\text{Right}(i)$:

```

Parent(i)
return [i/2]
Left(i)
return 2i
Right(i)
return 2i + 1

```

На большинстве компьютеров операция $2i$ в процедуре Left выполняется при помощи одной команды процессора путем побитового сдвига числа i на один бит влево. Операция $2i + 1$ в процедуре Right тоже выполняется очень быстро – достаточно биты двоичного представления числа i сдвинуть на одну позицию влево, а затем младший бит установить равным 1. Процедура Parent выполняется путем сдвига числа i на один бит вправо. При реализации пирамидальной сортировки эти функции часто представляются в виде макросов или встраиваемых процедур.

Определим пирамиду следующим образом:

Пирамида (двоичное дерево) определяется как последовательность (часть массива)

$$H_1, H_{1+1}, \dots, H_r \text{ такая, что } H_i \leq H_{2i} \text{ и } H_i \leq H_{2i+1} \text{ для } i=1 \dots r/2$$

$$H_1 = \min(H_1, H_2, \dots, H_n)$$

Пирамиде соответствует бинарное дерево попарного сравнения элементов (Рис.4.5), построенное на основе алгоритма выбора в результате $n-1$ сравнения при нахождении минимального элемента, который будет находиться в корне дерева.

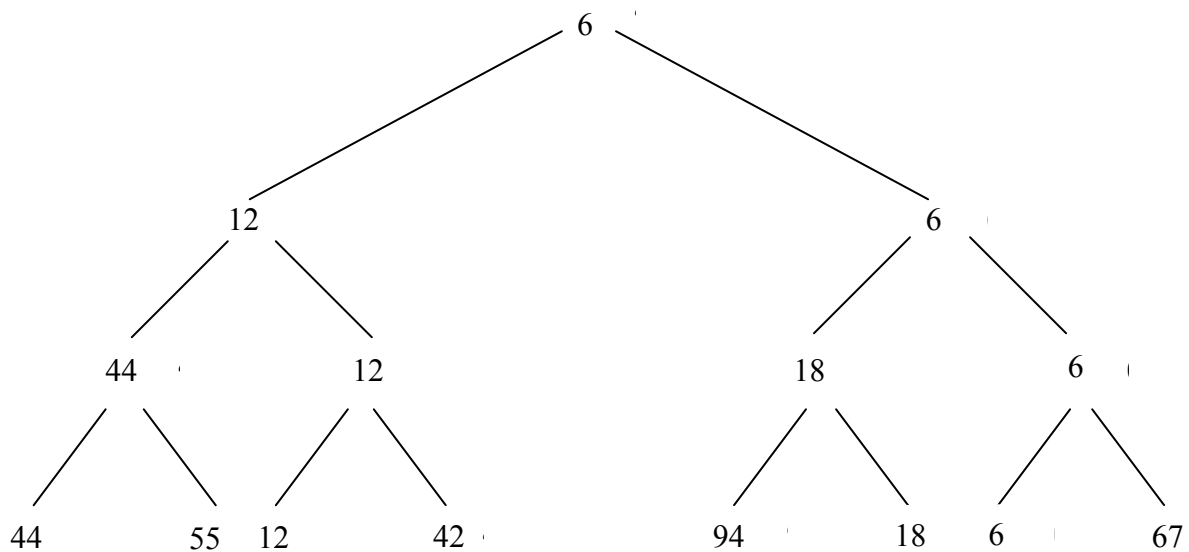


Рис 4.5. Бинарное дерево попарных сравнений
в алгоритме сортировки выбором

Второй этап сортировки – спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева. Элемент продвинувшийся в корень опять будет следующим минимальным. После n шагов дерево становится пустым и сортировка заканчивается.

Флойд предложил алгоритм построение пирамиды на «том же месте»:

пусть H_1, H_2, \dots, H_n есть сортируемый массив, причем H_m, \dots, H_n , где $m = (n \div 2) + 1$ образует нижний слой пирамиды, поскольку нет индексов i, j таких что $j=2i$ или $j=2i+1$, т.е. для этого слоя упорядоченности не требуется.

Пирамида расширяется влево – каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент, пока элементы стоящие слева от H_m не будут образовывать пирамиду. Такая процедура называется – Sift (просеивание):

Sift(L, R : index)

1 // i, j : index x : item; x – элемент вставляемый в пирамиду

2// i, j – пара индексов, фиксирующих элементы, меняющиеся на каждом шаге местами.

3 $i \leftarrow L$

4 $j \leftarrow 2 * L$

5 $X \leftarrow a[L]$

6 if ($j < R$) & ($a[j+1] < a[j]$) then $j \leftarrow j+1$

```

7 while (j <= R) & (a[j] < X) do
8 a[i] ← a[j]
9 a[j] ← X
10 i ← j
11 j ← 2*j
12 if (j < R) & (a[j+1] < a[j]) then j ← j+1

```

Процесс формирования пирамиды из n элементов $H_1... H_n$ на том же самом месте описывается следующим образом:

```

1 L ← (n div 2) + 1;
2 while L > 1 do
3 L ← L - 1
4 sift(L, n)

```

И наконец, n сдвигающих шагов выталкивания наименьших элементов на вершину дерева с последующим смещением в конец массива (рис.):

```

1 R ← n
2 while R > 1 do
3 x ← a[1]
4 a[1] ← a[R]
5 a[R] ← x
6 R ← R-1
7 sift(1, R)

```

На рис. 4.6. показан процесс просеивания минимального элемента.

1	2	3	4	5	6	7	8
4	5	1	4	9	1	0	6
4	5	2	2	4	8	6	7
4	5	2	1	4	9	1	0
4	5	2	2	4	8	6	7
4	5	6	0	4	9	1	1
4	5	6	2	4	8	2	7
4	4	0	5	9	1	1	6
4	2	6	5	4	8	2	7
0	4	1	5	9	1	4	6
6	2	2	5	4	8	4	7

Рис. 4.6. Построение пирамиды на том же самом месте

	1	2	3	4	5	6	7	8
0	4	1	5	9	1	4	6	
6	2	2	5	4	8	4	7	
1	4	1	5	9	6	4	0	
2	2	8	5	4	7	4	6	
1	4	4	5	9	6	1	0	
8	2	4	5	4	7	2	6	
4	5	4	6	9	1	1	0	
2	5	4	7	4	8	2	6	
4	5	9	6	4	1	1	0	
4	5	4	7	2	8	2	6	
5	6	9	4	4	1	3	0	
5	7	4	4	2	8	2	6	
6	9	5	4	4	1	1	0	
7	4	5	4	2	8	2	6	
9	6	5	4	4	1	1	0	
4	7	5	4	2	8	2	6	

Рис. 4.7. Пример процесса сортировки с помощью Heapsort

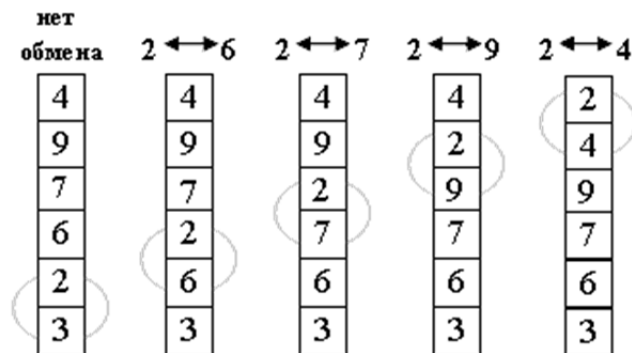
В самом плохом из возможных случаев Heapsort потребует $n \log n$ шагов.

$$T(n) = O(n \log n)$$

Heapsort «любит» начальные последовательности, в которых элементы более или менее отсортированы, то фаза порождения пирамиды потребует мало перемещений. Среднее число перемещений приблизительно равно $n/2 \log(n)$, причем отклонения от этого значения незначительны.

4.6. Обменные сортировки

Алгоритм прямого обмена основывается на сравнении и смене позиций пары соседних элементов (рис. 4.8). Процесс продолжается до тех пор, пока не будут упорядочены все элементы (рис. 4.9).



Нулевой проход, сравниваемые пары выделены

Рис. 4.8.

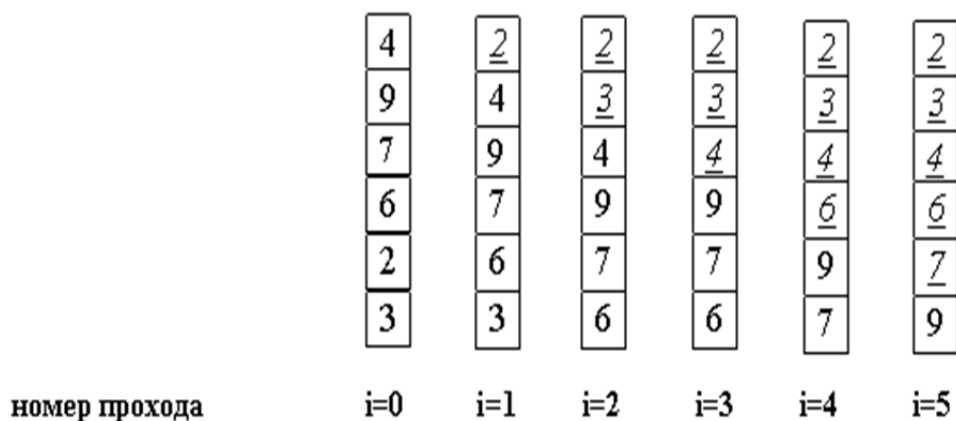


Рис. 4.9.

Опишем алгоритм прямого обмена

```

BUBBLESORT(a, n)
1 for i ← n-1 downto 1 do
2   Flag ← false
3   for j ← 1 to i do
4     if a[j] > a[j+1] then
5       Tmp ← a[j]
6       a[j] ← a[j+1]
7       a[j+1] ← Tmp
8     Flag ← true
9   if not Flag then return a

```

Алгоритм имеет среднюю и максимальную временные сложности $O(n^2)$ (два вложенных цикла, зависящих от n линейно). Введение переменной Flag и прерывание работы в случае отсортированного массива позволяет свести минимальную временную сложность к $O(n)$.

Отметим одну особенность приведенного алгоритма: легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются с минимальной скоростью: один шаг за итерацию. Массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов. Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Данный алгоритм иногда называют "шейкер-сортировкой".

```

SHAKERSORT (a,n);
1 L ← 2
2 R ← n

```



```

3  $k \leftarrow n$ 
4 repeat
5 for  $j \leftarrow R$  downto  $L$  do {справа налево}
6 if  $a[j-1] > a[j]$  then
7  $x \leftarrow a[j-1]$ 
8  $a[j-1] \leftarrow a[j]$ 
9  $a[j] \leftarrow x$ 
10  $k \leftarrow j$ 
11  $L \leftarrow k+1$ 
12 for  $j \leftarrow L$  to  $R$  do {слева направо}
13 if  $a[j-1] > a[j]$  then
14  $x \leftarrow a[j-1]$ 
15  $a[j-1] \leftarrow a[j]$ 
16  $a[j] \leftarrow x$ 
17  $k \leftarrow j$ 
18  $R \leftarrow k-1$ 
19 while  $L < R$ 

```

Число сравнений строго обменном алгоритме $(n^2-n)/2$ среднее число перемещений:

$3(n^2-n)/2$. Минимальное число сравнений $C_{min} = n-1$, среднее число сравнений пропорционально $\frac{1}{2}(n^2-n(k_2+\ln(n)))$.

4.7. Быстрая сортировка

Быстрая сортировка – это алгоритм сортировки, время работы которого для входного массива из n чисел в наихудшем случае равно $O(n^2)$. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм на практике зачастую оказывается оптимальным благодаря тому, что в среднем время его работы намного лучше: $O(n \lg n)$. Кроме того, постоянные множители, не учтенные в выражении $O(n \lg n)$, достаточно малы по величине. Алгоритм обладает также тем преимуществом, что сортировка в нем выполняется без использования дополнительной памяти, поэтому он хорошо работает даже в средах с виртуальной памятью.

Алгоритм быстрой сортировки является реализацией парадигмы «разделяй и властвуй». Разделение исходного массива осуществляется по следующему принципу:

1. Выбрать наугад какой-либо элемент массива – x
2. Просмотреть массив слева направо, пока не обнаружим элемент $A_i > x$
3. Просмотреть массив справа налево, пока не встретим $A_i < x$

4. Поменять местами эти два элемента
5. Процесс просмотра и обмена продолжается, пока указатели обоих просмотров не встретятся

Рис. 4.10 иллюстрирует процесс разделения



Рис. 4.10.

Опишем алгоритм разделения массива на две части относительно элемента со значением x (на рис. 3 $x=6$).

```

partition (a, l, n, x)
1 i ← l
2 j ← n
3 выбрать x
4 repeat
5 while a[i] < x do i ← i+1
6 while a[j] > x do j ← j-1
7 if i ≤ j then w ← a[i]
8 a[i] ← a[j]
9 a[j] ← w
10 i++
11 j--
12 while i < j

```

После первичного разбиения остается рекурсивно применить алгоритм partition () отдельно для левой и правой части и т.д.

```

SORT (L,R:index);
1 i ← L
2 j ← R
3 x ← a[(L+R) div 2]
4 partition (a,L,R,x)
5 if j > L then Sort (L,j)

```

6 if $i < R$ then Sort (i, R)

Начало рекурсии – Sort ($1, n$).

Ожидаемое число обменов в быстром алгоритме – $(n-1)/6$, общее число сравнений $n \log n$. Наихудший случай – в качестве элемента для разбиения x выбирается наибольшее из всех значений в указанной области, т.е. левая часть состоит из $n-1$ элементов, а правая из 1, тогда временная сложность становится пропорциональна n^2 .

4.8. Сортировка слиянием

Многие полезные алгоритмы имеют рекурсивную структуру: для решения данной задачи они рекурсивно вызывают сами себя один или несколько раз, чтобы решить вспомогательную задачу, имеющую непосредственное отношение к поставленной задаче. Такие алгоритмы зачастую разрабатываются с помощью метода декомпозиции, или разбиения: сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем; далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получить решение исходной задачи. Парадигма, лежащая в основе метода декомпозиции «разделяй и властвуй», на каждом уровне рекурсии включает в себя три этапа [6].

1. Разделение задачи на несколько подзадач.

2. Покорение – рекурсивное решение этих подзадач. Когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно.

3. Комбинирование решения исходной задачи из решений вспомогательных задач. Алгоритм сортировки слиянием (merge sort) в большой степени соответствует парадигме метода разбиения.

На интуитивном уровне его работу можно описать таким образом.

Разделение: сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит $n/2$ элементов

Покорение: сортировка обеих вспомогательных последовательностей методом слияния.

Комбинирование: слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной.

Основная операция, которая производится в процессе сортировки по методу слияний, – это объединение двух отсортированных последователь-

ностей в ходе комбинирования (последний этап). Это делается с помощью вспомогательной процедуры $\text{Merge}(A, p, q, r)$, где A – массив, а p , q и r – индексы, нумерующие элементы массива, такие, что $p < q < r$. В этой процедуре предполагается, что элементы подмассивов $A[p..q]$ и $A[q+1..r]$ упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива $A[p..r]$.

Для выполнения процедуры MERGE требуется время в $O(n)$, где $n = r - p + 1$ – количество подлежащих слиянию элементов.

```

MERGE( $A, p, q, r$ )
1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  Создаем массивы  $L[1..n_1 + 1]$  и  $R[1..n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5      do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7      do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13     do if  $L[i] \leq R[j]$ 
14         then  $A[k] \leftarrow L[i]$ 
15              $i \leftarrow i + 1$ 
16     else  $A[k] \leftarrow R[j]$ 
17          $j \leftarrow j + 1$ 

```

Опишем работу процедуры Merge. В строке 1 вычисляется длина n_1 подмассива $A[p..q]$, а в строке 2 – длина n_2 подмассива $A[q + 1..r]$. Далее в строке 3 создаются массивы L ("левый" – "left") и R ("правый" – "right"), длины которых равны $n_1 + 1$ и $n_2 + 1$ соответственно. В цикле for в строках 4 и 5 подмассив $A[p..q]$ копируется в массив $L[1..n_1]$, а в цикле for в строках 6 и 7 подмассив $A[q + 1..r]$ копируется в массив $R[1..n_2]$. В строках 8 и 9 последним элементам массивов L и R присваиваются пороговые значения.

Перед каждой итерацией цикла for в строках 12-17, подмассив $A[p..k - 1]$ содержит $k - p$ наименьших элементов массивов $L[1..n_1 + 1]$ и $R[1..n_2 + 1]$ в отсортированном порядке. Кроме того, элементы $L[i]$ и $R[j]$ являются наименьшими элементами массивов L и R , которые еще не скопированы в массив A .

Необходимо показать, что этот инвариант цикла соблюдается перед первой итерацией рассматриваемого цикла for, что каждая итерация цикла

не нарушает его, и что с его помощью удастся продемонстрировать корректность алгоритма, когда цикл заканчивает свою работу.

Инициализация. Перед первой итерацией цикла $k = p$, поэтому подмассив $A [p..k - 1]$ пуст. Он содержит $k - p = 0$ наименьших элементов массивов L и R . Поскольку $r = j = 1$, элементы $L[i]$ и $R [j]$ – наименьшие элементы массивов L и R , не скопированные обратно в массив A .

Сохранение. Чтобы убедиться, что инвариант цикла сохраняется после каждой итерации, сначала предположим, что $L [r] \leq R [j]$. Тогда $L [r]$ – наименьший элемент, не скопированный в массив A . Поскольку в подмассиве $A [p..k - 1]$ содержится $k - p$ наименьших элементов, после выполнения строки 14, в которой значение элемента $L [r]$ присваивается элементу $A [k]$, в подмассиве $A [p..k]$ будет содержаться $k - p + 1$ наименьший элемент. В результате увеличения параметра k цикла `for` и значения переменной r (строка 15), инвариант цикла восстанавливается перед следующей итерацией. Если же выполняется неравенство $L[i] > R[j]$, то в строках 16 и 17 выполняются соответствующие действия, в ходе которых также сохраняется инвариант цикла.

Завершение. Алгоритм завершается, когда $k = r + 1$. В соответствии с инвариантом цикла, подмассив $A [p..k - 1]$ (т.е. подмассив $A [p..r]$) содержит $k - p = r - p + 1$ наименьших элементов массивов $L [1..n_1 + 1]$ и $R [1..n_2 + 1]$ в отсортированном порядке. Суммарное количество элементов в массивах L и R равно $n_1 + n_2 + 2 = r - p + 3$. Все они, кроме двух самых больших, скопированы обратно в массив A , а два оставшихся элемента являются сигнальными (пороговыми).

Чтобы показать, что время работы процедуры Merge равно $O(n)$, где $n = r - p + 1$, заметим, что каждая из строк 1-3 и 8-11 выполняется в течение фиксированного времени; длительность циклов `for` в строках 4-7 равна $O(n_1 + n_2) = O(n)$, а в цикле `for` в строках 12-17 выполняется n итераций, на каждую из которых затрачивается фиксированное время.

Теперь процедуру Merge можно использовать в качестве подпрограммы в алгоритме сортировки слиянием. Процедура Merge_Sort(A, p, r) выполняет сортировку элементов в подмассиве $A [p..r]$. Если справедливо неравенство $p \geq r$, то в этом подмассиве содержится не более одного элемента, и, таким образом, он отсортирован. В противном случае производится разбиение, в ходе которого вычисляется индекс q , разделяющий массив $A[p..r]$ на два подмассива: $s [n/2]$ элементами и $A [q..r]$ с $[n/2]$ элементами.

```

MERGE_SORT( $A, p, r$ )
1 if  $p < r$ 
2 then  $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
3 MERGE_SORT( $A, p, q$ )
4 MERGE_SORT( $A, q + 1, r$ )

```

5 MERGE(A, p, q, r)

Чтобы отсортировать последовательность $A = (A [1], A [2], \dots, A [n])$, вызывается процедура Merge_Sort(A, l, n). Временная сложность алгоритма сортировки слиянием можно определить как $O(n \log n)$.

5. ДЕРЕВЬЯ

Формально дерево (tree) определяется как конечное множество T одного или более узлов со следующими свойствами[1,3,4]:

- существует один выделенный узел – корень (root) данного дерева T ;

- остальные узлы (за исключением корня) распределены среди $m \geq 0$ непересекающихся множеств T_1, \dots, T_m , и каждое из этих множеств, в свою очередь, является деревом; деревья T_1, \dots, T_m называются поддеревьями (subtrees) данного корня.

На рис. 5.1 приведены различные интерпретации определения дерева.

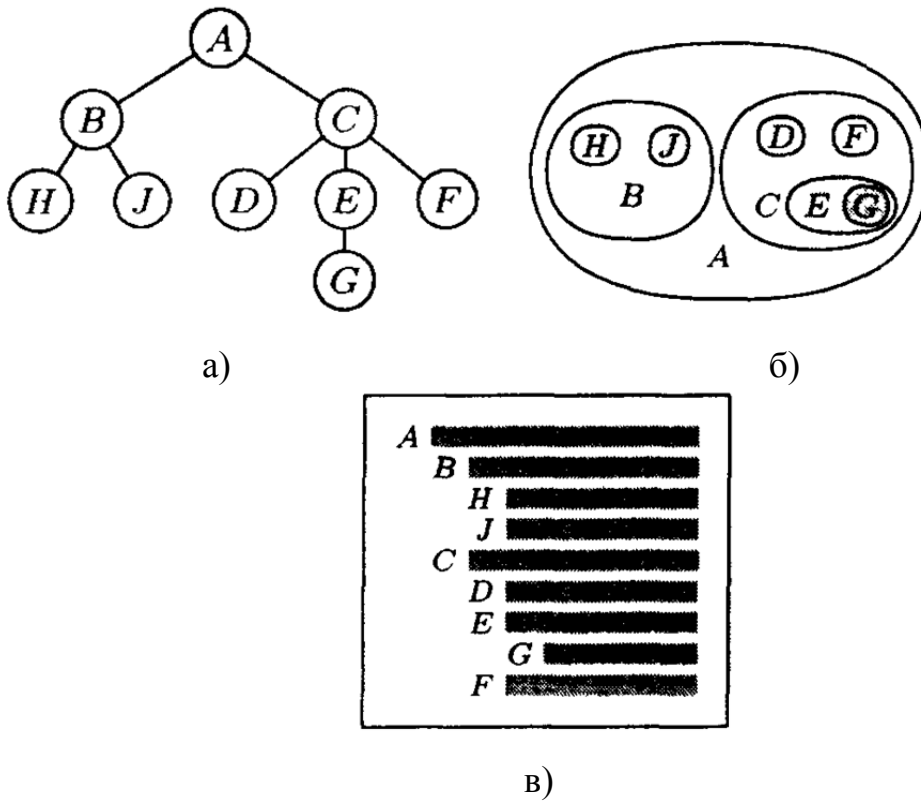


Рис. 5.1.

а – иерархическая структура, б – множества, в – линейное представление

5.1. Прохождение /обход (traversing/walk) бинарных деревьев

Пошаговый перебор элементов дерева по связям между предками-узлами и потомками-узлами называется обходом дерева. Обход, при котором каждый узел-предок просматривается прежде его потомков называется предупорядоченным обходом или обходом в прямом порядке (pre-order walk), а когда просматриваются сначала потомки, а потом предки, то обход называется поступорядоченным обходом или обходом в обратном порядке (post-order walk). Существует также симметричный обход, при котором посещается сначала левое поддерево, затем узел(текущий корень), затем – правое поддерево, и обход в ширину, при котором узлы просматриваются уровень за уровнем. Каждый уровень обходится слева направо.

Первые три способа обхода рекурсивно можно определить следующим образом:

если дерево *Tree* является пустым деревом, то в список обхода заносится пустая запись;

если дерево *Tree* состоит из одной вершины, то в список обхода записывается эта вершина;

если *Tree* – дерево с корнем *n* поддеревьями *Tree₁*, *Tree₂*, ..., *Tree_k*,... то:

при прохождении в прямом порядке сначала посещается корень *n*, затем в прямом порядке вершины поддерева *Tree₁*, далее в прямом порядке вершины поддерева *Tree₂* и т.д. последними в прямом порядке посещаются вершины поддерева *Tree_k*, формализуем описание алгоритма

```
procedure PREORDER(n: вершина);  
  //Обход дерева в прямом порядке  
  1 Занести в список обхода вершину n;  
  2 for для каждого потомка s вершины n в порядке слева направо  
do  
  3 PREORDER(s);
```

Рис.5.2 иллюстрирует обход в прямом порядке.

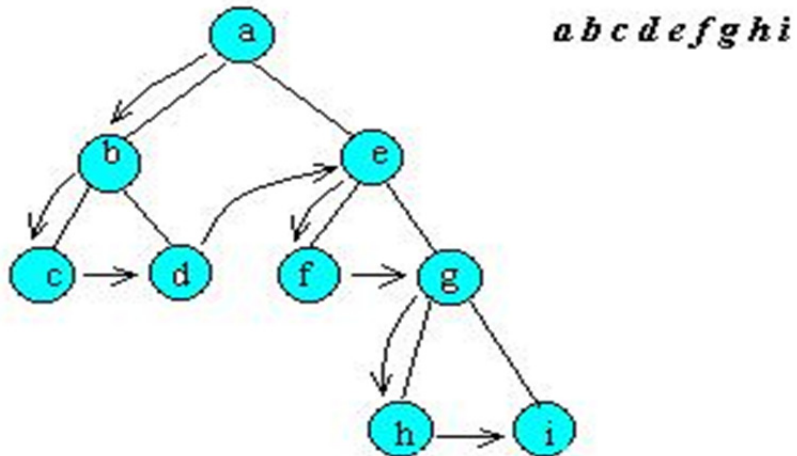


Рис. 5.2.

при прохождении в обратном порядке сначала посещаются в обратном порядке вершины поддерева $Tree_1$

далее последовательно в обратном порядке посещаются вершины поддеревьев $Tree_2 \dots, Tree_k$

последним посещается корень n ;

procedure LASTORDER(n : вершина);

//Обход дерева в обратном порядке

1 for для каждого потомка s вершины n в порядке слева направо do
LASTORDER(s)

2 Занести в список обхода вершину n

Рис.5.3 иллюстрирует обход в обратном порядке.

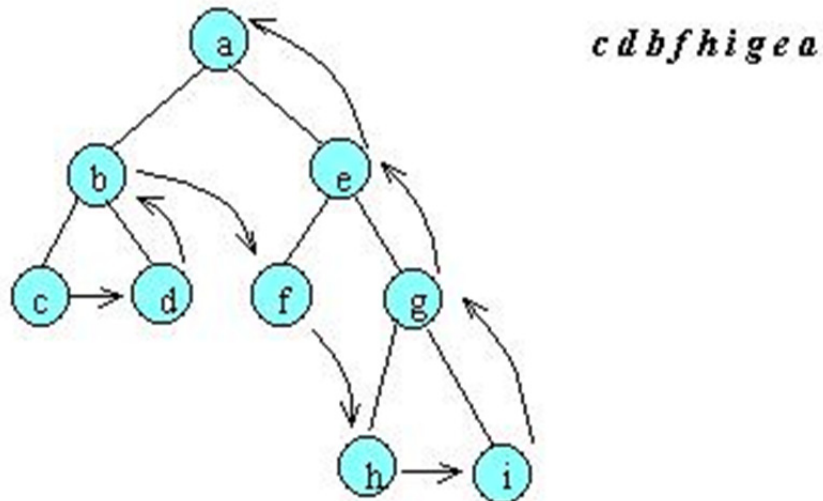


Рис. 5.3.

Прямой и обратный порядок прохождения деревьев используется в компиляторах для преобразования арифметических и логических выражений в бесскобочную запись (прямая и обратная польская запись)

Например, выражение $Y=3\ln(x+1) - a/x^2$ может быть представлено в виде следующего дерева рис.

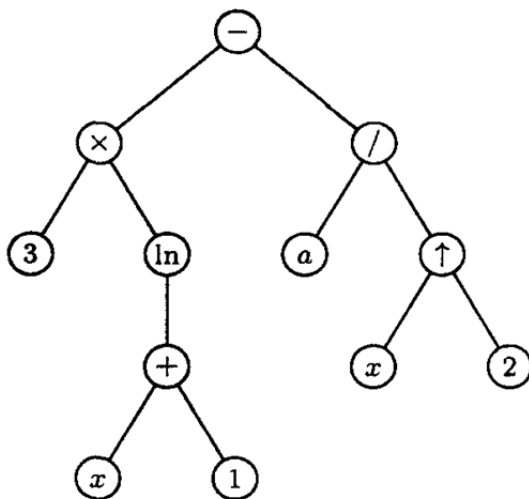


Рис.5.4.

Прямой обход дает прямую польскую запись:

$- \times 3 \ln + x 1 / a 1 a \uparrow x 2$

Обратный обход дает обратную польскую запись:

$3 x 1 + \ln \times a x 2 \uparrow / -$

При прохождении в симметричном порядке сначала посещаются в симметричном порядке вершины поддерева $Tree_1$, далее корень n , затем последовательно в симметричном порядке вершины поддеревьев $Tree_2, \dots, Tree_k$:

```

procedure INORDER( $n$ : вершина)
// Обход дерева в симметричном порядке
1 if  $n$  – лист then
2   занести в список обхода узел  $n$ 
3 else
4   INORDER(самый левый потомок вершины  $n$ )
5   Занести в список обхода вершину  $n$ 
6   for для каждого потомка  $s$  вершины  $n$ ,
7     исключая самый левый, в порядке слева
8     направо do INORDER( $s$ );
  
```

Как уже отмечалось с помощью симметричного обхода можно получить отсортированный массив. Дерево строится по следующим правилам: в качестве корня создается узел, в который записывается первый

элемент массива. Для каждого очередного элемента создается новый лист. Если элемент меньше значения в текущем узле, то для него выбирается левое поддерево, если больше или равен правое:



Рис. 5.5.

5.2. Бинарные деревья поиска

Деревья поиска представляют собой структуры данных, которые поддерживают многие операции с динамическими множествами, включая поиск элемента, минимального и максимального значения, предшествующего и последующего элемента, вставку и удаление. Таким образом, дерево поиска может использоваться и как словарь, и как очередь с приоритетами.

Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте[6]. Для полного бинарного дерева с n узлами эти операции выполняются за время $O(\lg n)$ в наихудшем случае. Математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\lg n)$, так что все основные операции динамическим множеством в таком дереве выполняются в среднем за время $\Theta(\lg n)$.

На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. Речь идет о деревьях сбалансированных по высоте по определенным

критериям, это AVL –деревья, 2-3, 3-4 деревья, красно-черные деревья, высота которых определяется как $O(\lg n)$.

Определим понятие бинарное дерево поиска. объектом. В дополнение к полям ключа *key* и сопутствующих данных, каждый узел содержит поля *left*, *right* и *p*, которые указывают на левый и правый дочерние узлы и на родительский узел соответственно. Если дочерний или родительский узел отсутствуют, соответствующее поле содержит значение NIL. Единственный узел, указатель *p* которого равен nil, – это корневой узел дерева. Ключи в бинарном дереве поиска хранятся таким образом, чтобы в любой момент удовлетворять следующему свойству бинарного дерева поиска. Если *x* – узел бинарного дерева поиска, а узел *y* находится в левом поддереве *x*, то $key[y] \leq key[x]$. Если узел *y* находится в правом поддереве *x*, то $key[x] \leq key[y]$.

Так, на рис. 5.6а ключ корня равен 5, ключи 2, 3 и 5, которые не превышают значение ключа в корне, находятся в его левом поддереве, а ключи 7 и 8, которые не меньше, чем ключ 5, – в его правом поддереве. То же свойство, как легко убедиться, выполняется для каждого другого узла дерева. На рис. 5.6б показано дерево с теми же узлами и обладающее тем же свойством, однако менее эффективное в работе, поскольку его высота равна 4, в отличие от дерева на рис. 5. 6а, высота которого равна 2.

Свойство бинарного дерева поиска позволяет нам вывести все ключи, находящиеся в дереве, в отсортированном порядке с помощью простого рекурсивного алгоритма, называемого центрированным (симметричным) обходом дерева (*inorder tree walk*). Этот алгоритм получил данное название в связи с тем, что ключ в корне поддерева выводится между значениями ключей левого поддерева.

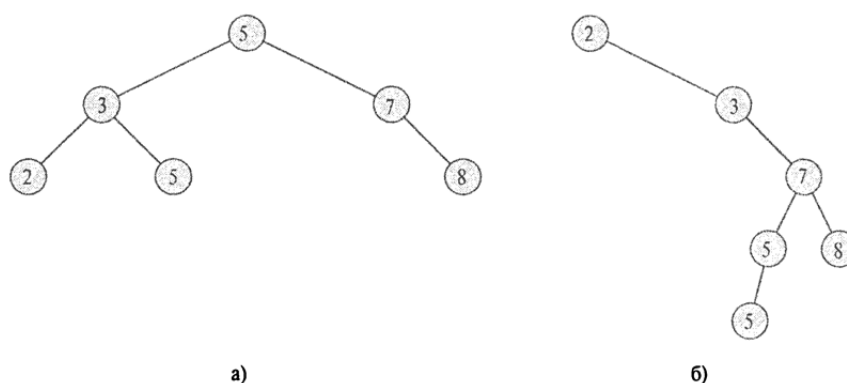


Рис. 5.6.

Наиболее распространенной операцией, выполняемой с бинарным деревом поиска, является поиск в нем определенного ключа. Кроме то-

го, бинарные деревья поиска поддерживают такие запросы, как поиск минимального и максимального элемента, а также предшествующего и последующего. В данном разделе мы рассмотрим все эти операции и покажем, что все они могут быть выполнены в бинарном дереве поиска высотой h за время $O(h)$.

5.2.1. Поиск заданного ключа

Для поиска узла с заданным ключом в бинарном дереве поиска используется следующая процедура `Tree_Search`, которая получает в качестве параметров указатель на корень бинарного дерева и ключ k , а возвращает указатель на узел с этим ключом (если таковой существует; в противном случае возвращается значение `NIL`).

```
TREE_SEARCH( $x, k$ )
1 if  $x = \text{NIL}$  или  $k = \text{key}[x]$ 
2   then return  $x$ 
3 if  $k < \text{key}[x]$ 
4   then return TREE_SEARCH(left[ $x$ ],  $k$ )
5   else return TREE_SEARCH(right[ $x$ ],  $k$ )
```

Процедура поиска начинается с корня дерева и проходит вниз по дереву. Для каждого узла x на пути вниз его ключ `key[x]` сравнивается с переданным в качестве параметра ключом k . Если ключи одинаковы, поиск завершается. Если k меньше `key[x]`, поиск продолжается в левом поддереве x ; если больше – то поиск переходит в правое поддерево. Так, (рис. 5.7) для поиска ключа 13 мы должны пройти следующий путь от корня: $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$. Узлы, которые мы посещаем при рекурсивном поиске, образуют нисходящий путь от корня дерева, так что время работы процедуры `Tree_Search` равно $O(h)$, где h – высота дерева. Ту же процедуру можно записать итеративно, "разворачивая" окончательную рекурсию в цикл `while`. На большинстве компьютеров такая версия оказывается более эффективной.

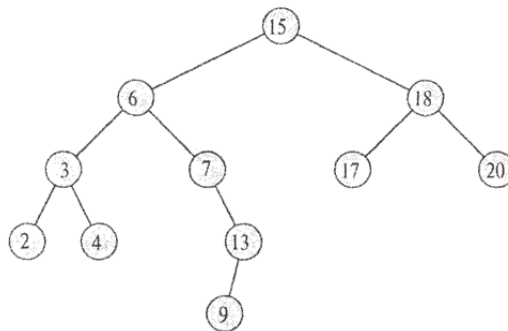


Рис. 5.7. Запросы в бинарном дереве поиска

ITERATIVE_TREE_SEARCH(x, k)

```
1 while  $x \neq \text{NIL}$  и  $\text{key}[x]$ 
2   do if  $k < \text{key}[x]$ 
3     then  $x \leftarrow \text{left}[x]$ 
4     else  $x \leftarrow \text{right}[x]$ 
5 return  $x$ 
```

5.2.2. Поиск минимума и максимума

Элемент с минимальным значением ключа легко найти, следуя по указателям `left` от корневого узла до тех пор, пока не встретится значение `NIL`. Так, на рис. 5.7, следуя по указателям `left`, мы пройдем путь $15 \rightarrow 6 \rightarrow 3 \rightarrow 2$ до минимального ключа в дереве, равного 2. Вот как выглядит реализация описанного алгоритма:

TREE_MINIMUM(z)

```
1 while  $\text{left}[x] \neq \text{NIL}$ 
2   do  $x \leftarrow \text{left}[x]$ 
3 return  $x$ 
```

Свойство бинарного дерева поиска гарантирует корректность процедуры `TREE_MINIMUM`. Если у узла x нет левого поддерева, то поскольку все ключи в правом поддереве x не меньше ключа $\text{key}[x]$, минимальный ключ поддерева с корнем в узле x находится в этом узле. Если же у узла есть левое поддерево, то, поскольку в правом поддереве не может быть узла с ключом, меньшим $\text{key}[x]$, а все ключи в узлах левого поддерева не превышают $\text{key}[x]$, узел с минимальным значением ключа находится в поддереве, корнем которого является узел `left[x]`.

Алгоритм поиска максимального элемента дерева симметричен алгоритму поиска минимального элемента:

TREE_MAXIMUM(x)

```
1 while  $\text{right}[x] \neq \text{NIL}$ 
2   do  $x \leftarrow \text{right}[x]$ 
3 return  $x$ 
```

Обе представленные процедуры находят минимальный (максимальный) элемент дерева за время $O(h)$, где h – высота дерева, поскольку, как и в процедуре `TREE_SEARCH`, последовательность проверяемых узлов образует нисходящий путь от корня дерева.

5.2.3. Предшествующий и последующий элементы

Иногда, имея узел в бинарном дереве поиска, требуется определить, какой узел следует за ним в отсортированной последовательности, определяемой порядком централизованного обхода бинарного дерева, и какой узел предшествует данному. Если все ключи различны, последующим по отношению к узлу x является узел с наименьшим ключом, большим $\text{key}[x]$. Структура бинарного дерева поиска позволяет нам найти этот узел даже не выполняя сравнение ключей. Приведенная далее процедура возвращает узел, следующий за узлом x в бинарном дереве поиска (если таковой существует) и NIL, если x обладает наибольшим ключом в бинарном дереве.

```
TREE_SUCCESSOR( $x$ )
1 if  $\text{right}[x] \neq \text{NIL}$ 
2   then return TREE_MINIMUM( $\text{right}[x]$ )
3  $y \leftarrow p[x]$ 
4 while  $y \neq \text{NIL}$  и  $x = \text{right}[y]$ 
5   do  $x \leftarrow y$ 
6    $y \leftarrow p[y]$ 
7 return  $y$ 
```

Код процедуры TREE_SUCCESSOR разбивается на две части. Если правое поддерево узла x непустое, то следующий за x элемент является крайним левым узлом в правом поддереве, который обнаруживается в строке 2 вызовом процедуры TREE_MINIMUM(right[x]). Например, на рис. 5.7 следующим за узлом с ключом 15 является узел с ключом 17.

С другой стороны, если правое поддерево узла x пустое, и у x имеется следующий за ним элемент y , то y является наименьшим предком x , чей левый наследник также является предком x . На рис. 5.7 следующим за узлом с ключом 13 является узел с ключом 15. Для того чтобы найти y , мы просто поднимаемся вверх по дереву до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя. Это действие выполняется в строках 3-7 алгоритма.

Время работы алгоритма TREE_SUCCESSOR в дереве высотой h составляет $O(h)$, поскольку мы либо движемся по пути вниз от исходного узла, либо пути вверх. Процедура поиска последующего узла в дереве TREE_PREDECESSOR симметрична процедуре TREE_SUCCESSOR и также имеет время работы $O(h)$.

Если в дереве имеются узлы с одинаковыми ключами, мы можем просто определить последующий и предшествующий узлы как те, что возвращаются процедурами TREE_SUCCESSOR и TREE_PREDECESSOR соответственно.

Очевидно, операции поиска, определения минимального и максимального элемента, а также предшествующего и последующего, в бинарном дереве поиска высоты h могут быть выполнены за время $O(h)$. \dagger

5.2.4. Вставка и удаление

Операции вставки и удаления приводят к внесению изменений в динамическое множество, представленное бинарным деревом поиска. Структура данных должна быть изменена таким образом, чтобы отражать эти изменения, но при этом сохранить свойство бинарных деревьев поиска. Как мы увидим в этом разделе, вставка нового элемента в бинарное дерево поиска выполняется относительно просто, а удалением значительно сложнее.

Вставка

Для вставки нового значения v в бинарное дерево поиска T мы воспользуемся процедурой TREE_INSERT. Процедура получает в качестве параметра узел z , у которого $\text{key}[z] = v$, $\text{left}[z] = \text{NIL}$ и $\text{right}[z] = \text{NIL}$, после чего она, таким образом, изменяет T и некоторые поля z , что z оказывается вставленным в соответствующую позицию в дереве.

```
TREE_INSERT( $T, z$ )
1  $y \leftarrow \text{NIL}$ 
2  $x \leftarrow \text{root}[T]$ 
3 while  $x \neq \text{NIL}$ 
4   do  $y \leftarrow x$ 
5     if  $\text{key}[z] < \text{key}[x]$ 
6       then  $x \leftarrow \text{left}[x]$ 
7       else  $x \leftarrow \text{right}[x]$ 
8  $p[z] \leftarrow y$ 
9 if  $y = \text{NIL}$ 
10 then  $\text{root}[T] \leftarrow z$  // Дерево  $T$  – пустое
11 else if  $\text{key}[z] < \text{key}[y]$ 
12   then  $\text{left}[y] \leftarrow z$ 
13   else  $\text{right}[y] \leftarrow z$ 
```

На рис. 5.8 показана работа процедуры Tree_Insert. Подобно процедурам TREE_SEARCH и ITERATIVE_TREE_SEARCH, процедура TREE_INSERT начинает работу с корневого узла дерева и проходит по нисходящему пути. Указатель x отмечает проходимый путь, а указатель y указывает на родительский по отношению к x узел. После инициализации цикл while в строках 3-7 перемещает эти указатели вниз по дереву, перемещаясь влево или вправо в зависимости от результата сравнения ключей $\text{key}[x]$ и $\text{key}[z]$ до тех пор пока x не станет равным NIL. Это значение находится именно в той позиции, куда следует поместить элемент z . В

строках 8-13 выполняется установка значений указателей для вставки z . Так же, как и другие примитивные операции над бинарным деревом поиска, процедура TREE_INSERT выполняется за время $O(h)$ в дереве высотой h .

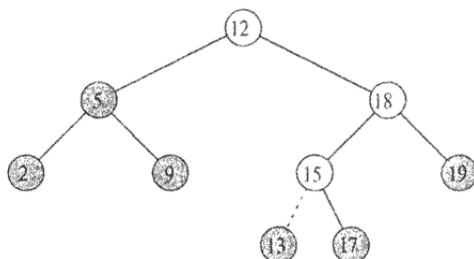


Рис. 5.8. Вставка элемента с ключом 13 в бинарное дерево поиска (светлые узлы указывают путь от корня к позиции вставки; пунктиром указана связь, добавляемая при вставке нового элемента)

Удаление

Процедура удаления данного узла z из бинарного дерева поиска получает в качестве аргумента указатель на z . Процедура рассматривает три возможные ситуации, показанные на рис. 5.9. Если у узла z нет дочерних узлов (рис. 5.9а), то мы просто изменяем его родительский узел $p[z]$, заменяя в нем указатель на z значением NIL. Если у узла z только один дочерний узел (рис. 5.9б), то мы удаляем узел z , создавая новую связь между родительским и дочерним узлом узла z . И наконец, если у узла z два дочерних узла (рис. 5.9в), то мы находим следующий за ним узел y , у которого нет левого дочернего узла и убираем его из позиции, где он находился ранее, путем создания новой связи между его родителем и потомком, и заменяем им узел z .

```

TREE_DELETE( $T, z$ )
1 if left[ $z$ ] = NIL или right[ $z$ ] = NIL
2 then  $y \leftarrow z$ 
3 else  $y \leftarrow$  TREE_SUCCESOR( $z$ )
4 if left[ $y$ ]  $\neq$  NIL
5   then  $x \leftarrow$  left[ $y$ ]
6   else  $x \leftarrow$  right[ $y$ ]
7 if  $x \neq$  NIL
8 then  $p[x] \leftarrow p[y]$ 
9 if  $p[y] =$  NIL
10 then root[ $T$ ]  $\leftarrow x$ 
11 else if  $y =$  left[ $p[y]$ ]
12   then left[ $p[y]$ ]  $\leftarrow x$ 
13   else right[ $p[y]$ ]  $\leftarrow x$ 
14 if  $y \neq z$ 

```



```

15 then key[z] ← key[y]
16 // Копирование сопутствующих данных в z
17 return y

```

В строках 1-3 алгоритм определяет убираемый путем "склейки" родителя и потомка узел y . Этот узел представляет собой либо узел z (если у узла z не более одного дочернего узла), либо узел, следующий за узлом z (если у z два дочерних узла). Затем в строках 4-6 x присваивается указатель на дочерний узел узла y либо значение NIL, если у y нет дочерних узлов. Затем узел y убирается из дерева в строках 7-13 путем изменения указателей в $p[y]$ и x . Это удаление усложняется необходимостью корректной отработки граничных условий (когда x равно NIL или когда y – корневой узел). И наконец, в строках 14-16, если удаленный узел y был следующим за z , мы перезаписываем ключ z и сопутствующие данные ключом и сопутствующими данными y . Удаленный узел y возвращается в строке 17, с тем чтобы вызывающая процедура могла при необходимости освободить или использовать занимаемую им память. Время работы описанной процедуры с деревом высотой h составляет $O(h)$.

Таким образом, операции вставки и удаления в бинарном дереве поиска высоты h могут быть выполнены за время $O(h)$.

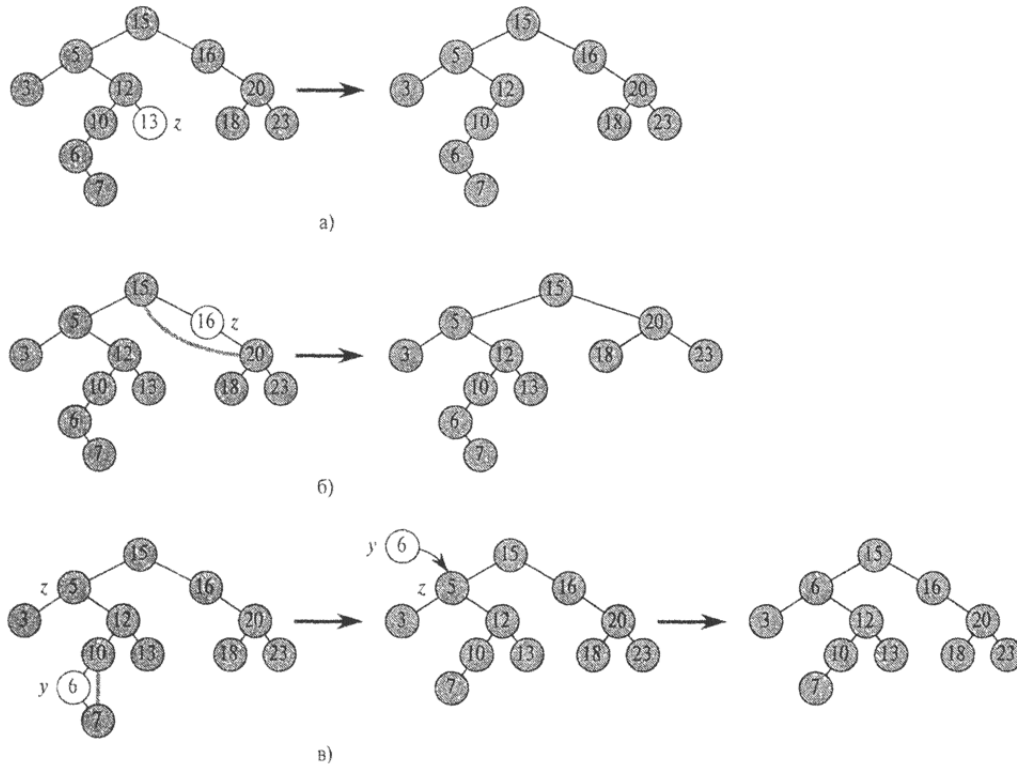


Рис. 5.9. Удаление узла z из бинарного дерева поиска

5.3. Сбалансированные деревья

Алгоритм вставки в бинарное дерево, который мы только что рассмотрели, дает хорошие результаты при использовании случайных входных данных, но все же существует неприятная возможность того, что при этом будет построено вырожденное дерево. Можно было бы разработать алгоритм, поддерживающий дерево в оптимальном состоянии все время, где под оптимальностью мы в данном случае понимаем сбалансированность дерева. Идеально сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1. поддержка идеальной сбалансированности, к сожалению, очень сложная задача. Другая идея заключается в том, чтобы ввести менее жесткие критерии сбалансированности и на их основе предложить достаточно простые алгоритмы обеспечения этих критериев.

Очень красивое решение проблемы поддержания дерева поиска в хорошем состоянии было предложено в 1962 году двумя советскими математиками – Г.М. Адельсон-Вельским и Е.М. Ландисом [ДАН СССР 146 (1962), 263-266]. Их метод требует всего лишь двух дополнительных битов на узел и никогда не использует более $O(\log n)$ операций для поиска по дереву или вставки элемента

Метод, предоставляющий все эти преимущества, будем называть сбалансированными деревьями (многие авторы используют термин AVL(АВЛ)-деревья – по первым буквам фамилий авторов).

Высота дерева определяется как его максимальный уровень, длина самого длинного пути от корня к внешнему узлу. Бинарное дерево поиска назовем сбалансированным, если высота левого поддерева любого узла отличается не более чем на ± 1 от высоты правого поддерева. На рис. показано сбалансированное дерево высотой 5 с 17 внутренними узлами; фактор сбалансированности обозначен внутри каждого узла знаками "+", "." и "-" в соответствии с величиной разности между высотами правого и левого поддерева (+1, 0 или -1).

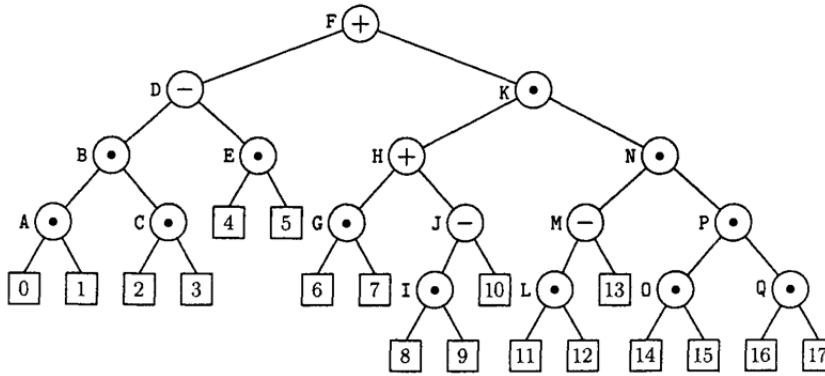


Рис. 5.10. Сбалансированное по AVL бинарное дерево

Адельсон-Вельский и Ландис доказали следующую теорему: Высота сбалансированного дерева с n внутренними узлами ограничена значениями

$$\lg(n+1) \text{ и } 1.4405 \lg(n+2) - 0.3277.$$

Отсюда можно сделать вывод, что путь поиска по сбалансированному по AVL дереву не превышает пути поиска по идеально сбалансированному дереву более чем 45%.

Рассмотрим варианты возникающие при вставке нового узла в AVL-дерево.

- $hL = hR$. После включения L и R станут разной высоты, но критерий сбалансированности не будет нарушен;
- $hL < hR$. После включения L и R станут равной высоты, то есть критерий сбалансированности даже улучшится;
- $hL > hR$. После включения критерий сбалансированности нарушится и дерево необходимо перестраивать.

Отсюда сформулируем общий алгоритм вставки нового узла в AVL-дерево:

1. проход по дереву, чтобы убедиться, что включаемого значения в дереве нет;
2. включение новой вершины и определение результирующего показателя сбалансированности;
3. "отступление" по пути поиска и проверка в каждой вершине показателя сбалансированности. При необходимости – балансировка.

На практике используются 4 алгоритма восстановления AVL-сбалансированности:

малый и большой левый поворот, малый и большой правый поворот, которые выбираются в зависимости от значений показателей разбалансированности (рис)

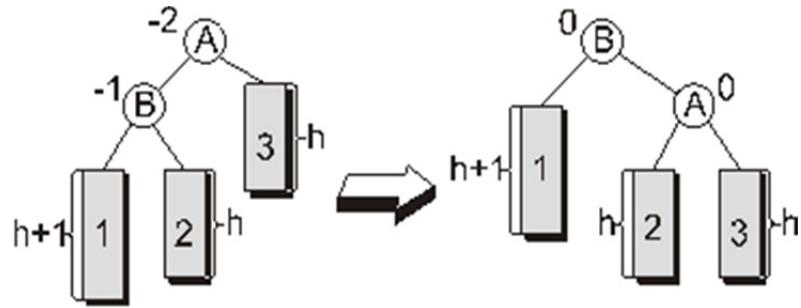


Рис. 5.11. Малое левое вращение

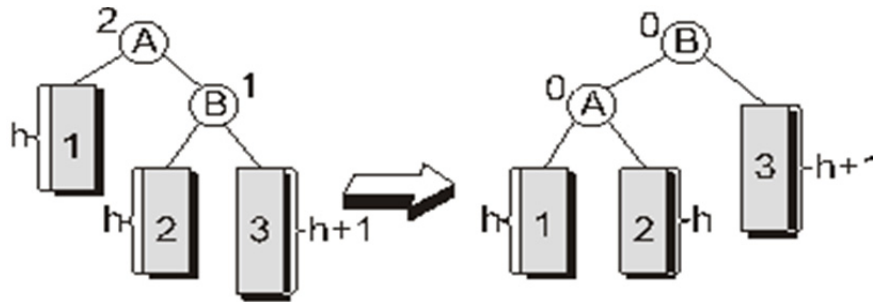


Рис. 5.12. Малое правое вращение

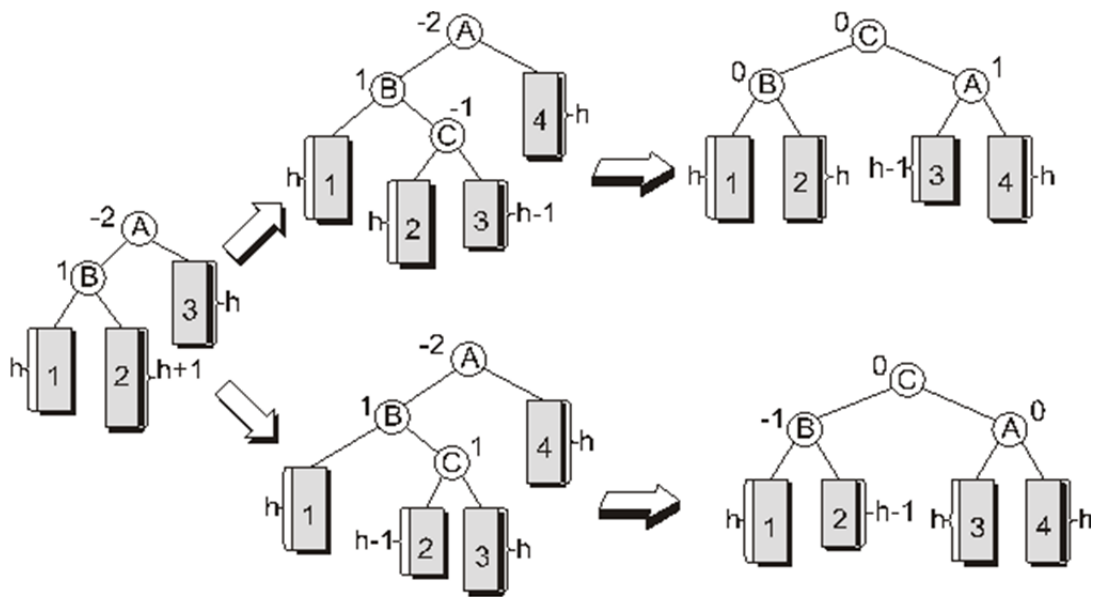


Рис. 5.13. Большое левое вращение

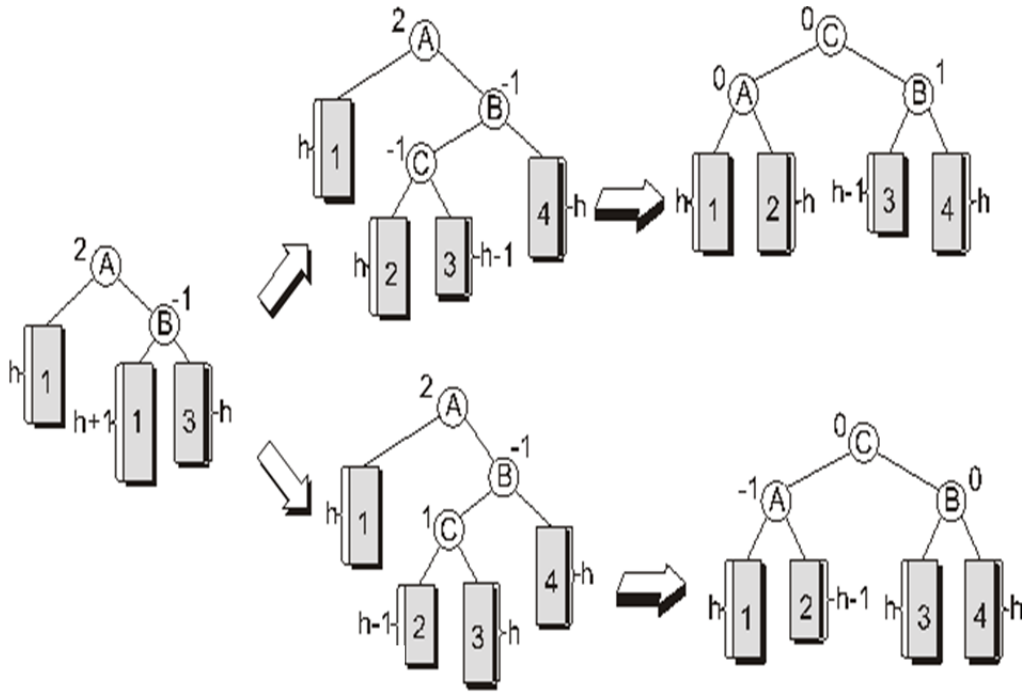


Рис. 5.14. Большое правое вращение

На рисунках прямоугольниками обозначены поддеревья, цифры внутри – номера поддеревьев, цифры рядом с узлами – показатели сбалансированности. Алгоритм балансировки показан на следующем примере малого левого поворота.

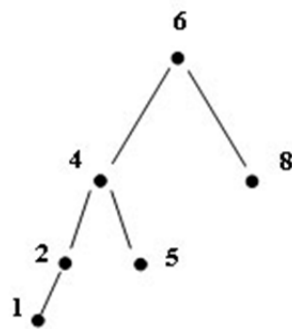


Рис. 5.15. Исходное дерево

1. Определяем адрес той вершины, которая станет корнем дерева:
2. $p1 = (*p).Left;$

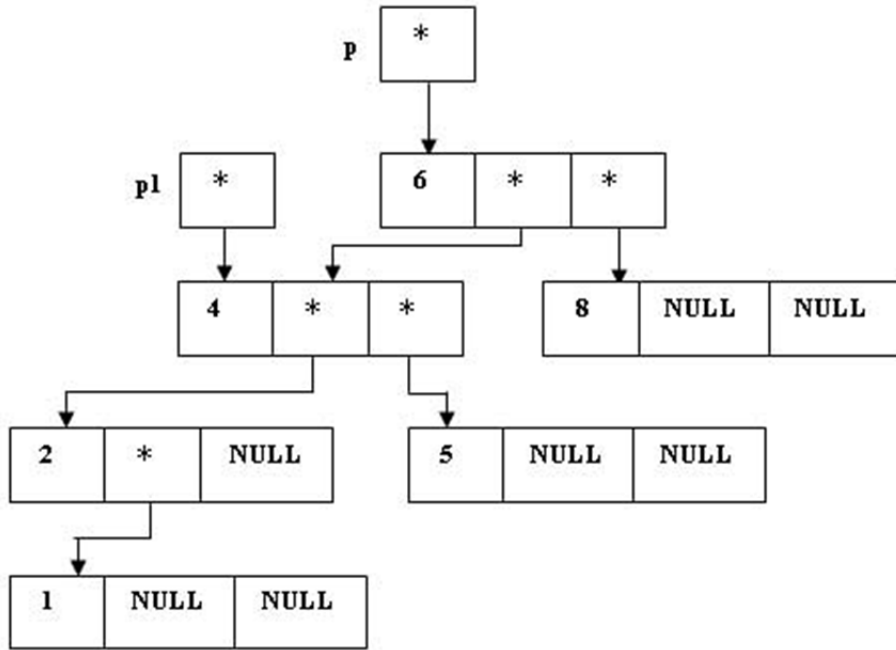


Рис. 5.16. Сохранение адреса нового корня дерева

3. Переприкрепляем правое поддереву от "нового" корня, делая это поддереву левым поддеревом "старого" корня:

4. $(*p).Left = (*p1).Right;$

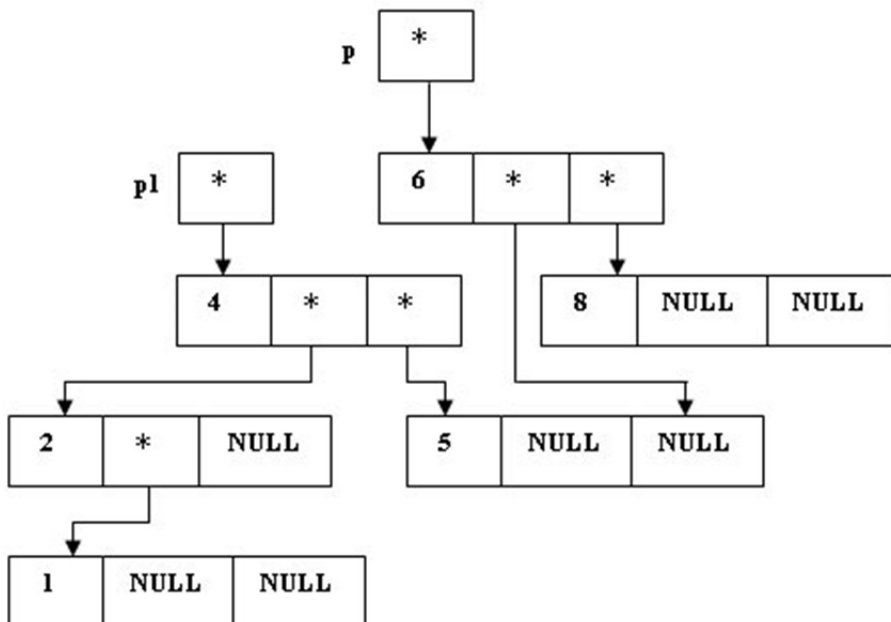


Рис. 5.17. Переприкрепление

5. Определяем правое поддереву "нового" корня, как начинающееся со "старого" корня:

6. $(*p1).Right = p;$

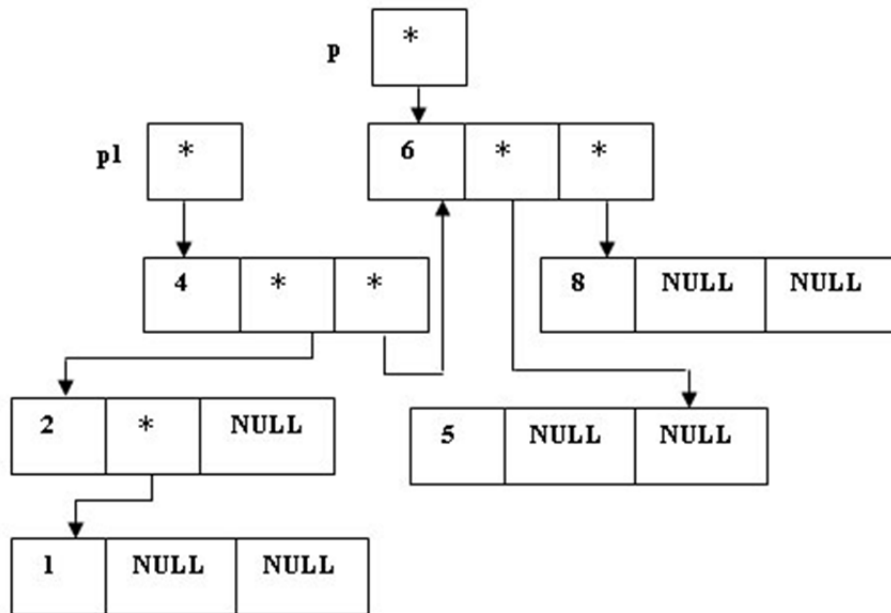


Рис. 5.18. Определение правого поддерева "нового" корня

7. Изменяем значение указателя на корень дерева (p) и обнуляем значение сбалансированности:

8. $(*p).bal = 0; p = p1;$

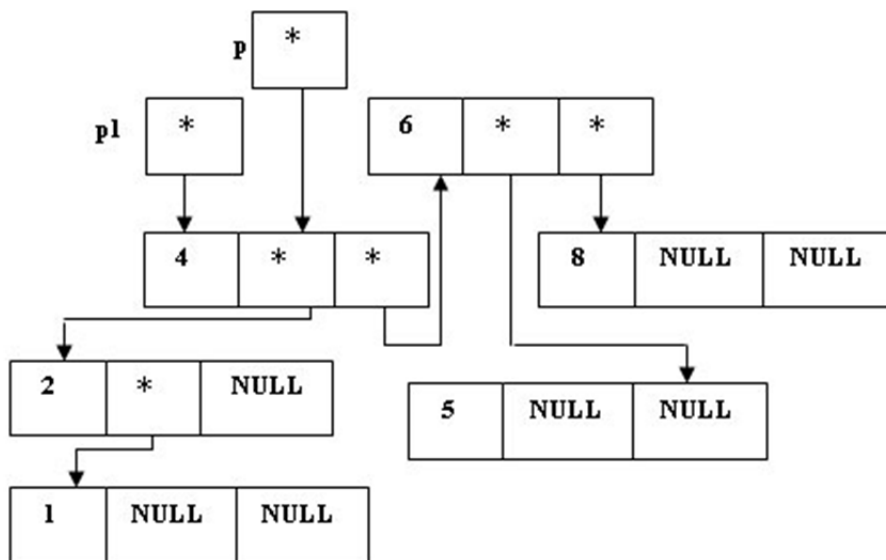


Рис. 5.19. Установка начальных значений

После балансировки получилось следующее сбалансированное по АВЛ дерево:

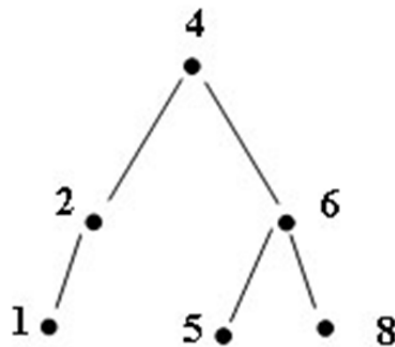


Рис. 5.20. Результат балансировки

Весь алгоритма поворота описывается 5 операторами (C++).

```

p1 = (*p).Left;
(*p).Left = (*p1).Right; (*p1).Right = p;
(*p).bal = 0; ;

```

Идеи балансировки деревьев получили дальнейшее развитие, например в красно-черных деревьях (red-black tree). Свойства КЧД (Р. Байер, 1972 г). Определение КЧД:

1. Корень черный;
2. Каждая вершина может быть либо красной, либо чёрной. Бесцветных вершин, или вершин другого цвета быть не может.
3. Каждый лист (NIL) имеет чёрный цвет.
4. Если вершина красная, то оба её потомка – чёрные.
5. Все пути от корня к листьям содержат одинаковое число чёрных вершин.

Для КЧД (Рис.5.21) ни один путь от узла к корню не превышает другого пути более чем вдвое. Операции вставки и удаления вершин в КЧД могут нарушать свойства КЧД. Чтобы восстановить эти свойства, надо будет перекрашивать некоторые вершины и менять структуру дерева – осуществлять повороты аналогичные поворотам AVL-деревьев.

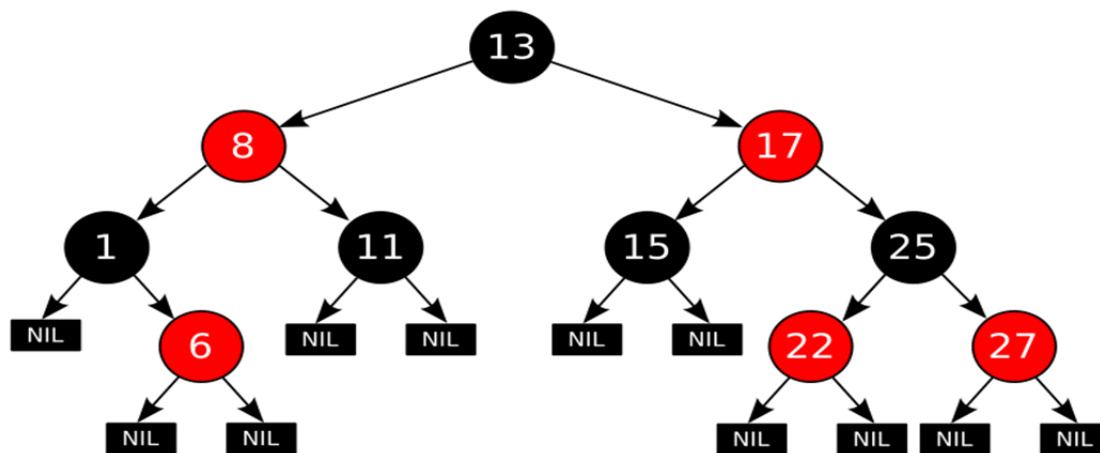


Рис. 5.21. Красно-черное дерево

Добавление объекта «ключ-элемент» в красно-черное дерево, хранящее n элементов, выполняется за $O(\log n)$ операций и требует максимум $O(\log n)$ перекрашиваний

5.4. Сильноветвящиеся деревья (B-деревья)

B-деревья представляют собой сбалансированные деревья поиска, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом). B-деревья похожи на красно-черные деревья, но отличаются более высокой оптимизацией дисковых операций ввода-вывода. Многие СУБД используют для хранения информации именно B-деревья (или их разновидности). B-деревья отличаются от красно-черных деревьев тем, что узлы B-дерева могут иметь много дочерних узлов – до тысяч, так что степень ветвления B-дерева может быть очень большой (хотя обычно она определяется характеристиками используемых дисков). B-деревья схожи с красно-черными деревьями в том, что все B-деревья с n узлами имеют высоту $O(\lg n)$, хотя само значение высоты B-дерева существенно меньше, чем у красно-черного дерева за счет более сильного ветвления. Таким образом, B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время $O(\lg n)$.

B-деревья представляют собой естественное обобщение бинарных деревьев поиска.

5.4.1. Определение B-деревьев

Для простоты предположим, как и в случае бинарных деревьев поиска и красно-черных деревьев, что сопутствующая информация, связанная с ключом, хранится в узле вместе с ключом. На практике вместе с ключом может храниться только указатель на другую дисковую страницу, содержащую сопутствующую информацию для данного ключа. Псевдокод в данной главе неявно подразумевает, что при перемещении ключа от узла к узлу вместе с ним перемещается и сопутствующая информация или указатель на нее. В распространенном варианте B-дерева, который называется B+-деревом, вся сопутствующая информация хранится в листьях, а во внутренних узлах хранятся только ключи и указатели на дочерние узлы. Таким образом, удастся получить максимально возможную степень ветвления во внутренних узлах.

B-дерево T представляет собой корневое дерево (корень которого $\text{root}[T]$), обладающее следующими свойствами[6].

1. Каждый узел x содержит следующие поля:

а) $n[x]$, количество ключей, хранящихся в настоящий момент в узле x .

б) Собственно ключи, количество которых равно $n[x]$ и которые хранятся в невозрастающем порядке, так что $\text{key}_1[x] \leq \text{key}_2[x] \leq \dots < \text{key}_{n[x]}[x]$

в) Логическое значение $\text{leaf}[x]$ равное TRUE, если x является листом,

и false, если x – внутренний узел.

2. Кроме того, каждый внутренний узел x содержит $n[x] + 1$ указателей $c_1[x]$,

$c_2[x], \dots, c_{n[x]+1}[x]$ на дочерние узлы. Листья не имеют дочерних узлов,

так что их поля c_i не определены.

3. Ключи $\text{key}_i[x]$ разделяют поддиапазоны ключей, хранящихся в поддеревьях: если k_i – произвольный ключ, хранящийся в поддереве с корнем $c_i[x]$, то $k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}[x] \leq k_{n[x]+1}$

4. Все листья расположены на одной и той же глубине, которая равна высоте дерева h .

5. Имеются нижняя и верхняя границы количества ключей, которые могут содержаться в узле. Эти границы могут быть выражены с помощью одного фиксированного целого числа $t \geq 2$, называемого минимальной степенью (minimum degree) B-дерева:

а) Каждый узел, кроме корневого, должен содержать как минимум $t - 1$ ключей. Каждый внутренний узел, не являющийся корневым, имеет,

таким образом, как минимум t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.

б) Каждый узел содержит не более $2t - 1$ ключей. Таким образом, внутренний узел имеет не более $2t$ дочерних узлов. Мы говорим, что узел заполнен (full), если он содержит ровно $2t - 1$ ключей.

Простейшее B-дерево получается при $t = 2$. При этом каждый внутренний узел может иметь 2, 3 или 4 дочерних узла, и мы получаем так называемое 2-3-4-дерево (2-3-4 tree). Однако обычно на практике используются гораздо большие значения t .

В типичном приложении, использующем B-дерева, количество обрабатываемых данных достаточно велико, и все они не могут одновременно разместиться в оперативной памяти. Алгоритмы работы с B-деревами копируют в оперативную память с диска только некоторые выбранные страницы, необходимые для работы, и вновь записывают на диск те из них, которые были изменены в процессе работы. Алгоритмы работы с B-деревами сконструированы таким образом, чтобы в любой момент времени обходиться только некоторым постоянным количеством страниц в основной памяти, так что ее объем не ограничивает размер B-деревьев, с которыми могут работать алгоритмы.

В нашем псевдокоде мы моделируем дисковые операции следующим образом. Пусть x – указатель на объект. Если объект находится в оперативной памяти компьютера, то мы обращаемся к его полям обычным способом – например, `key[x]`. Если же объект, к которому мы обращаемся посредством x , находится на диске, то мы должны выполнить операцию `Disk_Read(x)` для чтения объекта x в оперативную память перед тем, как будем обращаться к его полям. (Считаем, что если объект x уже находится в оперативной памяти, то операция `Disk_Read(x)` не требует обращения к диску.) Аналогично, для сохранения любых изменений в полях объекта x выполняется операция `Disk_Write(x)`. Таким образом, типичный шаблон работы с объектом x имеет следующий вид:

$x \leftarrow$ указатель на некоторый объект

`Disk_Read(x)`

Операции, обращающиеся и/или изменяющие поля x

`Disk_Write(x)` // Не требуется, если поля x не изменялись

Прочие операции, не изменяющие поля x

Система в состоянии поддерживать в процессе работы в оперативной памяти только ограниченное количество страниц. Мы будем считать, что страницы, которые более не используются, удаляются из оперативной памяти системой; наши алгоритмы работы с B-деревами не будут заниматься этим самостоятельно.

Поскольку в большинстве систем время выполнения алгоритма, работающего с B -деревьями, зависит в первую очередь от количества выполняемых операций с диском `Disk_Read` и `Disk_Write`, желательно минимизировать их количество и за один раз считывать и записывать как можно больше информации. Таким образом, размер узла B -дерева обычно соответствует дисковой странице. Количество потомков узла B -дерева, таким образом, ограничивается размером дисковой страницы.

Для больших B -деревьев, хранящихся на диске, степень ветвления обычно находится между 50 и 2000, в зависимости от размера ключа относительно размера страницы. Большая степень ветвления резко снижает как высоту дерева, так и количество обращений к диску для поиска ключа. На рис. 5.22 показано B -дерево высота которого равна 2, а степень ветвления – 1001; такое дерево может хранить более миллиарда ключей. При этом, поскольку корневой узел может храниться в оперативной памяти постоянно, для поиска ключа в этом дереве требуется максимум два обращения к диску.

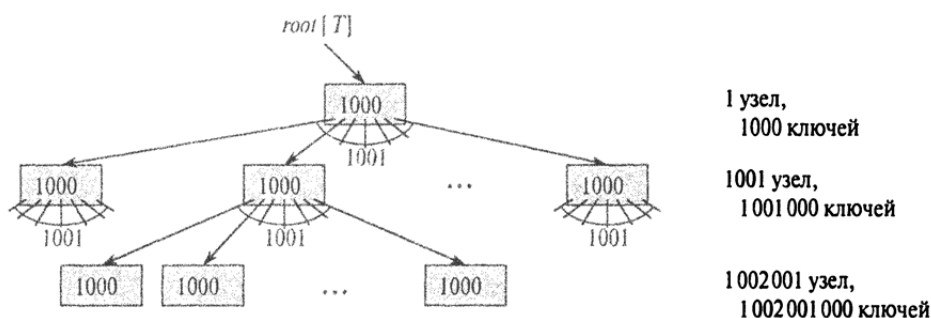


Рис. 5.22. B -дерево, хранящее более миллиарда ключей

Количество обращений к диску, необходимое для выполнения большинства операций с B -деревом, пропорционально его высоте. Нетрудно доказать, что высота B -дерева T с $n \geq 1$ узлами и минимальной степенью $t \geq 2$ не превышает $\log_t (n + 1)/2$. Таким образом, B -деревья требуют исследования примерно в $\lg t$ раз меньшего количества узлов по сравнению с красно-черными деревьями. Поскольку исследование узла дерева обычно требует обращения к диску, количество дисковых операций при работе с B -деревьями оказывается существенно сниженным.

Далее мы более подробно рассмотрим алгоритмы `B_TREE_SEARCH`, `B_TREE_CREATE` и `B_TREE_INSERT`. При рассмотрении этих алгоритмов мы примем следующие соглашения:

- корень B -дерева всегда находится в оперативной памяти, так что операция `Disk_Read` для чтения корневого узла не нужна; однако при

изменении корневого узла требуется выполнение операции Disk__Write, сохраняющей внесенные изменения на диске;

- все узлы, передаваемые в качестве параметров, уже считаны с диска.

Все представленные здесь процедуры выполняются за один нисходящий проход по дереву.

5.4.2. Поиск в B-дереве

Поиск в B-дереве очень похож на поиск в бинарном дереве поиска, но с тем отличием, что если в бинарном дереве поиска мы выбирали один из двух путей, то здесь предстоит сделать выбор из большего количества альтернатив, зависящего от того, сколько дочерних узлов имеется у текущего узла. Точнее, в каждом внутреннем узле x нам предстоит выбрать один из $n[x] + 1$ дочерних узлов.

Операция B_TREE_SEARCH представляет собой естественное обобщение процедуры TREE_SEARCH, определенной для бинарных деревьев поиска. В качестве параметров процедура B_TREE_SEARCH получает указатель на корневой узел x поддерева и ключ k , который следует найти в этом поддереве. Таким образом, вызов верхнего уровня для поиска во всем дереве имеет вид B_TREE_SEARCH(root [T], k). Если ключ k имеется в B-дереве, процедура B_TREE_SEARCH вернет упорядоченную пару (y, i) , состоящую из узла y и индекса i , такого что $key_i[y] = k$. В противном случае процедура вернет значение NIL.

```
B_TREE_SEARCH( $x, k$ )
1  $i \leftarrow 1$ 
2 while  $i \leq n[x]$  и  $k > key_i[x]$ 
3   do  $i \leftarrow i + 1$ 
4 if  $i \leq n[x]$  и  $k = key_i[x]$ 
5 then return  $(x, i)$ 
6 if leaf[ $x$ ]
7 then return NIL
8 else Disk_Read( $c_i[x]$ )
9 return B_TREE_SEARCH( $c_i[x], k$ )
```

В строках 1-3 выполняется линейный поиск наименьшего индекса i , такого что $k \leq key_i[x]$ (иначе i присваивается значение $n[x] + 1$). В строках 4-5 проверяется, не найден ли ключ в текущем узле, и если он найден, то выполняется его возврат. В строках 6-9 процедура либо завершает свою работу неудачей (если x является листом), либо рекурсивно вызывает себя для поиска в соответствующем поддереве x (после

выполнения чтения с диска необходимого дочернего узла, являющегося корнем исследуемого поддерева).

Процедура `B_TREE_SEARCH`, как и процедура `TREE_SEARCH` при поиске в бинарном дереве, проходит в процессе рекурсии узлы от корня в нисходящем порядке. Количество дисковых страниц, к которым выполняется обращение процедурой `B_TREE_SEARCH`, равно $O(h) = O(\log_t n)$, где h – высота B -дерева, а n – количество содержащихся в нем узлов. Поскольку $n[x] < 2t$, количество итераций цикла `while` в строках 2-3 в каждом узле равно $O(t)$, а общее время вычислений $-O(th) = O(t \log_t n)$.

5.4.3. Создание пустого B -дерева

Для построения B -дерева T мы сначала должны воспользоваться процедурой `B_TREE_CREATE` для создания пустого корневого узла, а затем вносить в него новые ключи при помощи процедуры `B_TREE_INSERT`. В обеих этих процедурах используется вспомогательная процедура `Allocate_Node`, которая выделяет дисковую страницу для нового узла за время $O(1)$. Мы можем считать, что эта процедура не требует вызова `Disk_Read`, поскольку никакой полезной информации о новом узле на диске нет.

```

B_TREE_CREATE( $T$ )
1  $x \leftarrow$  Allocate_Node()
2 leaf[ $x$ ]  $\leftarrow$  true
3  $n[x] \leftarrow 0$ 
4 Disk_Write( $x$ )
5 root[ $T$ ]  $\leftarrow x$ 

```

Процедура `B_TREE_CREATE` требует $O(1)$ дисковых операций и выполняется за время $O(1)$.

5.4.4. Вставка ключа в B -дерево

Вставка ключа в B -дерево существенно сложнее вставки в бинарное дерево поиска. Как и в случае бинарных деревьев поиска, мы ищем позицию листа, в который будет вставлен новый ключ. Однако при работе с B -деревом мы не можем просто создать новый лист и вставить в него ключ, поскольку такое дерево не будет являться корректным B -деревом. Вместо этого мы вставляем новый ключ в существующий лист. Поскольку вставить новый ключ в заполненный лист невозможно, мы вводим новую операцию – разбиение (`splitting`) заполненного (т.е. содержащего $2t - 1$ ключей) узла на два, каждый из которых содержит

по $t - 1$ ключей. Медиана, или средний ключ, – $\text{key}_t [y]$ (median key) – при этом перемещается в родительский узел, где становится разделительной точкой для двух вновь образовавшихся поддеревьев. Однако если родительский узел тоже заполнен, перед вставкой нового ключа его также следует разбить, и такой процесс разбиения может идти по восходящей до самого корня.

Как и в случае бинарного дерева поиска, в B -дереве мы вполне можем осуществить вставку за один нисходящий проход от корня к листу. Для этого нам не надо выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. Вместо этого при проходе от корня к листьям в поисках позиции для нового ключа мы разбиваем все заполненные узлы, через которые проходим (включая лист). Тем самым гарантируется, что если нам надо разбить какой-то узел, то его родительский узел не будет заполнен.

Разбиение узла B -дерева

Процедура $B_TREE_SPLIT_CHILD$ получает в качестве входного параметра незаполненный внутренний узел x (находящийся в оперативной памяти), индекс i и узел y (также находящийся в оперативной памяти), такой что $y = c_i [x]$ является заполненным дочерним узлом x . Процедура разбивает дочерний узел на два и соответствующим образом обновляет поля x , внося в него информацию о новом дочернем узле. дочернем узле. Для разбиения заполненного корневого узла мы сначала делаем корень дочерним узлом нового пустого корневого узла, после чего можем использовать вызов $B_TREE_SPLIT_CHILD$. При этом высота дерева увеличивается на 1. Разбиение – единственное средство увеличения высоты B -дерева.

На рис. 18.5 показан этот процесс. Заполненный узел y разбивается медианой 5, которая перемещается в родительский узел. Те ключи из y , которые больше медианы, помещаются в новый узел z , который становится новым дочерним

```

B_TREE_SPLIT_CHILD( $x, i, y$ )
1  $z \leftarrow \text{Allocate\_Node}()$ 
2  $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3  $n[z] \leftarrow t - 1$ 
4 for  $j \leftarrow 1$  to  $t - 1$ 
5   do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6 if not  $\text{leaf}[y]$ 
7 then for  $j \leftarrow 1$  to  $t$ 
8   do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9  $c_{i+1}[x] \leftarrow t-1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 

```

```

11 do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12  $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14   do  $key_{j+1}[x] \leftarrow key_j[x]$ 
15    $key_i[x] \leftarrow key_t[y]$ 
16    $n[x] \leftarrow n[x] + 1$ 
17   Disk_Write( $y$ )
18   Disk_Write( $z$ )
19   Disk_Write( $x$ )

```

Процедура B_TREE_SPLIT_CHILD использует простой способ "вырезать и вставить". Здесь y является i -м дочерним узлом x и представляет собой именно тот узел, который будет разбит на два. Изначально узел y имеет $2t$ дочерних узлов (содержит $2t - 1$ ключей); после разбиения количество его дочерних узлов снизится до t ($t - 1$ ключей). Узел z получает t больших дочерних узлов ($t - 1$ ключей) y и становится новым дочерним узлом x , располагаясь непосредственно после y в таблице дочерних узлов x . Медиана y перемещается в узел x и разделяет в нем y и z .

В строках 1-8 создается узел z и в него переносятся большие $t - 1$ ключей и соответствующие t дочерних узлов y . В строке 9 обновляется поле количества ключей в y . И наконец, строки 10-16 делают z дочерним узлом x , перенося медиану из y в x для разделения y и z , и обновляют поле количества ключей в x . В строках 17-19 выполняется запись на диск всех модифицированных данных. Время работы процедуры равно $\Theta(t)$ из-за циклов в строках 4-5 и 7-9 (прочие циклы выполняют $O(t)$ итераций). Процедура выполняет $O(1)$ дисковых операций.

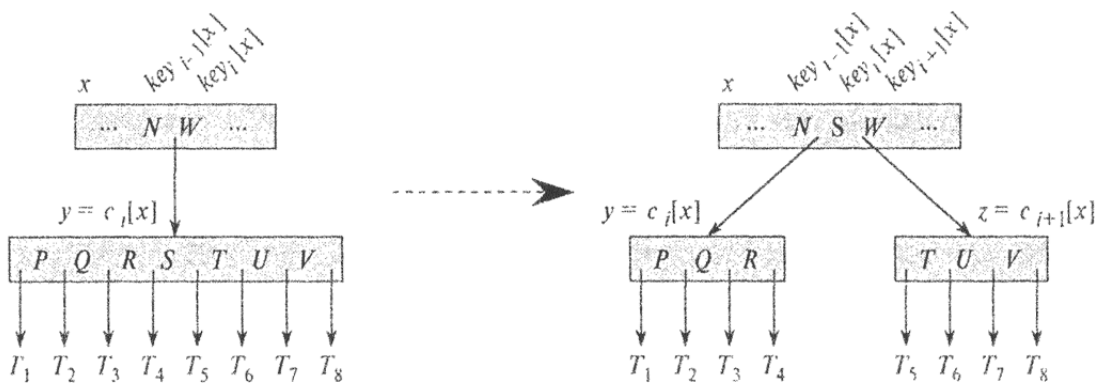


Рис. 5.23. Разбиение узла с $t = 4$

Вставка ключа в B-дерево за один проход

Вставка ключа k в B-дерево T высоты h выполняется за один нисходящий проход по дереву, требующий $O(h)$ обращений к диску. Необходимое процессорное время составляет $O(th) = O(t \log n)$. Процедура `B_TREE_INSERT` использует процедуру `B_TREE_SPLIT_CHILD` для гарантии того, что рекурсия никогда не столкнется с заполненным узлом.

```

B_TREE_INSERT( $T, k$ )
1  $r \leftarrow \text{root}[T]$ 
2 if  $n[r] = 2t-1$ 
3   then  $s \leftarrow \text{Allocate\_Node}()$ 
4      $\text{root}[T] \leftarrow s$ 
5      $\text{leaf}[s] \leftarrow \text{FALSE}$ 
6      $n[s] \leftarrow 0$ 
7      $c_1[s] \leftarrow r$ 
8     B_TREE_SPLIT_CHILD( $s, 1, r$ )
9     B_TREE_INSERT_NONFULL( $s, k$ )
10  else B_TREE_INSERT_NONFULL( $r, k$ )
  
```

Строки 3-9 предназначены для случая, когда заполнен корень дерева: при этом корень разбивается и новый узел s (у которого два дочерних узла) становится новым корнем B-дерева. Разбиение корня – единственный путь увеличения высоты B-дерева. становится новым корнем B-дерева.

На рис. 5.24 проиллюстрирован такой процесс разбиения корневого узла. В отличие от бинарных деревьев поиска, у B-деревьев высота увеличивается "сверху", а не "снизу". Завершается процедура вызовом другой процедуры – `B_TREE_INSERT_NONFULL`, которая выполняет вставку ключа k в дерево с незаполненным корнем. Данная процедура при необходимости рекурсивно спускается вниз по дереву, причем каждый узел, в который она входит, является незаполненным, что обеспечивается (при необходимости) вызовом процедуры `B_TREE_SPLIT_CHILD`.

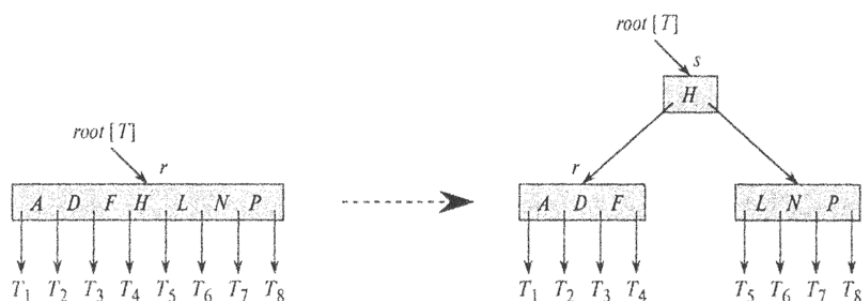


Рис. 5.24. Разбиение корня с $t = 4$

Вспомогательная процедура `B_TREE_INSERT_NONFULL` вставляет ключ k в узел x , который должен быть незаполненным при вызове процедуры. Операции `B_TREE_INSERT` и `B_TREE_INSERT_NONFULL` гарантируют, что это условие незаполненности будет выполнено.

```

B_TREE_INSERT_NONFULL( $x, \kappa$ )
1  $i \leftarrow n[x]$ 
2 if leaf[ $x$ ]
3   then while  $i \geq 1$  и  $k < \text{key}_i[x]$ 
4     do  $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$ 
5        $i \leftarrow i - 1$ 
6      $\text{key}_{i+1}[x] \leftarrow k$ 
7      $n[x] \leftarrow n[x] + 1$ 
8     Disk_Write( $x$ )
9   else while  $i \geq 1$  и  $k < \text{key}_i[x]$ 
10    do  $i \leftarrow i - 1$ 
11     $i \leftarrow i + 1$ 
12    Disk_Read( $c_i[x]$ )
13    if  $n[c_i[x]] = 2t-1$ 
14      then B_TREE_SPLIT_CHILD( $x, i, c_i[x]$ )
15        if  $k > \text{key}_i[x]$ 
16          then  $i \leftarrow i + 1$ 
17    B_TREE_INSERT_NONFULL( $c_i[x], \kappa$ )

```

Процедура `B_TREE_INSERT_NONFULL` работает следующим образом. Строки 3-8 обрабатывают случай, когда x является листом; при этом ключ k просто вставляется в данный лист. Если же x не является листом, то мы должны вставить k в подходящий лист в поддереве, корнем которого является внутренний узел x . В этом случае строки 9-11 определяют дочерний узел x , в который спустится рекурсия. В строке 13 проверяется, не заполнен ли этот дочерний узел, и если он заполнен, то вызывается процедура `B_TREE_SPLIT_CHILD`, которая разбивает его на два незаполненных узла, а строки 15-16 определяют, в какой из двух получившихся в результате разбиения узлов должна спуститься рекурсия. (Обратите внимание, что в строке 16 после увеличения i операция чтения с диска не нужна, поскольку процедура рекурсивно спускается в узел, только что созданный процедурой `B_TREE_SPLIT_CHILD`.) Строки 13-16 гарантируют, что процедура никогда не столкнется с заполненным узлом. Строка 17 рекурсивно вызывает процедуру `B_TREE_INSERT_NONFULL` для вставки k в соответствующее поддерево.

Количество обращений к диску, выполняемых процедурой `B_TREE_INSERT` для B -дерева высотой h , составляет $O(h)$, поскольку между вызовами `B_TREEINSERT_NONFULL` выполняется только $O(1)$ операций `Disk_Read` и `Disk_Write`.

Необходимое процессорное время равно $O(th) = O(t \log_t n)$. Поскольку в `B_TREE_INSERT_NONFULL` использована оконечная рекурсия, ее можно реализовать итеративно с помощью цикла `while`, наглядно показывающего, что количество страниц, которые должны находиться в оперативной памяти, в любой момент времени равно $O(1)$.

5.4.5. Удаление ключа из B -дерева

Удаление ключа из B -дерева, хотя и аналогично вставке, представляет собой более сложную задачу. Это связано с тем, что ключ может быть удален из любого узла, а не только из листа, а удаление из внутреннего узла требует определенной перестройки дочерних узлов. Как и в случае вставки, мы должны обеспечить, чтобы при выполнении операции удаления не были нарушены свойства B -дерева. Аналогично тому, как мы имели возможность убедиться, что узлы не слишком сильно заполнены для вставки нового ключа, нам предстоит убедиться, что узел не становится слишком мало заполнен в процессе удаления ключа (за исключением корневого узла, который может иметь менее $t - 1$ ключей, хотя и не может иметь более $2t - 1$ ключей). Итак, пусть процедура `B_TREE_DELETE` должна удалить ключ k из поддерева, корнем которого является узел x . Эта процедура разработана таким образом, что при ее рекурсивном вызове для узла x гарантировано наличие в этом узле, по крайней мере, t ключей. Это условие требует наличия в узле большего количества ключей, чем минимальное в обычном B -дереве, так что иногда ключ может быть перемещен в дочерний узел перед тем, как рекурсия обратится к этому дочернему узлу. Такое ужесточение свойства B -дерева (наличие "запасного" ключа) дает нам возможность выполнить удаление ключа за один нисходящий проход по дереву (с единственным исключением, которое будет пояснено позже). Следует также учесть, что если корень дерева x становится внутренним узлом, не содержащим ни одного ключа, то узел x удаляется, а его единственный дочерний узел $c_1[x]$ становится новым корнем дерева (при этом уменьшается высота B -дерева и сохраняется его свойство, требующее, чтобы корневой узел непустого дерева содержал как минимум один ключ).

Вместо того чтобы представить вам полный псевдокод процедуры удаления узла из B -дерева, мы просто набросаем последовательность выполняемых действий.

1. Если узел k находится в узле x и x является листом – удаляем k из x .

2. Если узел k находится в узле x и x является внутренним узлом, выполняем

следующие действия.

а) Если дочерний по отношению к x узел y , предшествующий ключу k в узле x , содержит не менее t ключей, то находим k_1 – предшественника k в поддереве, корнем которого является y . Рекурсивно удаляем k_1 и заменяем k в x ключом k' . (Поиск ключа k_1 и удаление его можно выполнить в процессе одного нисходящего прохода.)

б) Ситуация, симметричная ситуации а: если дочерний по отношению к x узел z , следующий за ключом k в узле x , содержит не менее t ключей, то находим k' – следующий за k ключ в поддереве, корнем которого является z . Рекурсивно удаляем k_1 и заменяем k в x ключом k_1 . (Поиск ключа k_1 и удаление его можно выполнить в процессе одного нисходящего прохода.)

в) В противном случае, если и y , и z содержат по $t - 1$ ключей, вносим k и все ключи z в y (при этом из x удаляется k и указатель на z , а узел y после этого содержит $2t - 1$ ключей), а затем освобождаем z и рекурсивно удаляем k ; из y .

3. Если ключ k отсутствует во внутреннем узле x , находим корень $c_i[x]$ поддерева, которое должно содержать k (если таковой ключ имеется в данном B -дереве). Если $c_i[x]$ содержит только $t - 1$ ключей, выполняем шаг 3а или 3б для того, чтобы гарантировать, что далее мы переходим в узел, содержащий как минимум t ключей. Затем мы рекурсивно удаляем k из поддерева с корнем $c_i[x]$.

а) Если $c_i[x]$ содержит только $t - 1$ ключей, но при этом один из ее непосредственных соседей (под которым мы понимаем дочерний по отношению к x узел, отделенный от рассматриваемого ровно одним ключом-разделителем) содержит как минимум t ключей, передадим в $c_i[x]$ ключ-разделитель между данным узлом и его непосредственным соседом из x , на его место поместим крайний ключ из соседнего узла и перенесем соответствующий указатель из соседнего узла в $c_i[x]$.

б) Если и $c_i[x]$ и оба его непосредственных соседа содержат по $t - 1$ ключей, объединим $c_i[x]$ с одним из его соседей (при этом бывший ключ-разделитель из x станет медианой нового узла).

Поскольку большинство ключей в B -дереве находится в листьях, можно ожидать, что на практике чаще всего удаления будут выполняться из листьев. Процедура `B_TREE_DELETE` в этом случае выполняется за один нисходящий проход по дереву, без возвратов. Однако при удалении ключа из внутреннего узла процедуре может потребоваться воз-

врат к узлу, ключ из которого был удален и замещен его предшественником или последующим за ним ключом (случаи 2а и 2б).

Хотя описание процедуры выглядит достаточно запутанным, она требует всего лишь $O(h)$ дисковых операций для дерева высотой h , поскольку между рекурсивными вызовами процедуры выполняется только $O(1)$ вызовов процедур `Disk_Read` или `Disk_Write`. Необходимое процессорное время составляет $O(th) = O((t \log_t n))$.

6. ХЕШИРОВАНИЕ ДАННЫХ

Для многих приложений достаточны динамические множества, поддерживающие только стандартные словарные операции вставки, поиска и удаления. Например, компилятор языка программирования поддерживает таблицу символов, в которой ключами элементов являются произвольные символьные строки, соответствующие идентификаторам в языке. Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, что и в связанном списке, а именно $O(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях математическое ожидание времени поиска элемента в хеш-таблице составляет $O(1)$.

Термин `hashing` или `scatter storage` (рассеянная память) – крошить, размалывать, рубить и тд (рассеяние, расстановка, рандомизация). Идея хеширования состоит в использовании некоторой частичной информации, полученной из ключа, в качестве основы поиска, т.е. вычисляется хеш-адрес $h(\text{key})$, который используется для поиска в хеш-таблице [1,5,10]

Хеш-таблица (`hash table`) представляет собой обобщение обычного массива. Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Прямая индексация рассматривается в применении, если мы в состоянии выделить массив размера, достаточного для того, чтобы для каждого возможного значения ключа имелась своя ячейка.

Если количество реально хранящихся в массиве ключей мало по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного

использования ключа в качестве индекса массива, индекс вычисляется по значению ключа.

6.1. Таблицы с прямой адресацией

Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества $U = \{0, 1, \dots, m - 1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или таблицу с прямой адресацией, который обозначим как $T[0..m - 1]$, каждая позиция, или ячейка (position, slot), которого соответствует ключу из пространства ключей U . На рис. 11.1 представлен данный подход. Ячейка k указывает на элемент множества с ключом k . Если множество не содержит элемента с ключом k , то $T[k] = \text{NIL}$. На рисунке каждый ключ из пространства $U = \{0, 1, \dots, 9\}$ соответствует индексу таблицы. Множество реальных ключей $K = \{2, 3, 5, 8\}$ определяет ячейки таблицы, которые содержат указатели на элементы. Остальные ячейки (закрашенные темным цветом) содержат значение NIL.

Реализация словарных операций тривиальна:

```

DIRECT_ADDRESS_SEARCH( $T, k$ )
return  $T[k]$ 
DIRECT_ADDRESS_INSERT( $T, x$ )
 $T[\text{key}[x]] \leftarrow x$ 
DIRECT_ADDRESS_DELETE( $T, x$ )
 $T[\text{key}[x]] \leftarrow \text{NIL}$ 
    
```

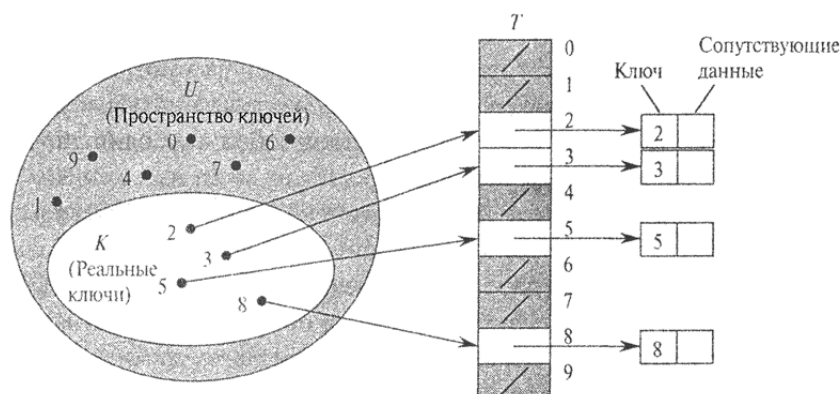


Рис. 6.1. Реализация динамического множества с использованием таблицы с прямой адресацией

Каждая из приведенных операций очень быстрая: время их работы равно $O(1)$. В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. То есть вместо хранения ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице – указателей на эти объекты, эти объекты можно хранить непосредственно в ячейках таблицы (что тем самым приводит к экономии используемой памяти). Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если мы знаем индекс объекта в таблице, мы тем самым знаем и его ключ. Однако если ключ не хранится в ячейке таблицы, то нам нужен какой-то иной механизм для того, чтобы пометить пустые ячейки.

6.2. Хеш-таблицы

Недостаток прямой адресации очевиден: если пространство ключей U велико, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно – зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество K реально сохраненных ключей может быть мало по сравнению с пространством ключей U , а в этом случае память, выделенная для таблицы T , в основном расходуется напрасно.

Когда множество K хранящихся в словаре ключей гораздо меньше пространства возможных ключей U , хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Надо только заметить, что это граница среднего времени поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для наихудшего случая.

В случае прямой адресации элемент с ключом k хранится в ячейке k . При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем хеш-функцию h для вычисления ячейки для данного ключа k . Функция h отображает пространство ключей U на ячейки хеш-таблицы $T[0..m-1]$:

$$h:U \rightarrow \{0,1,\dots,m-1\}.$$

Мы говорим, что элемент с ключом k хешируется в ячейку $h(k)$; величина $h(k)$ называется хеш-значением ключа k . На рис. 11.2 представлена основная идея хеширования. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо $|U|$ зна-

чений мы можем обойтись всего лишь m значениями. Соответственно снижаются и требования к количеству памяти.

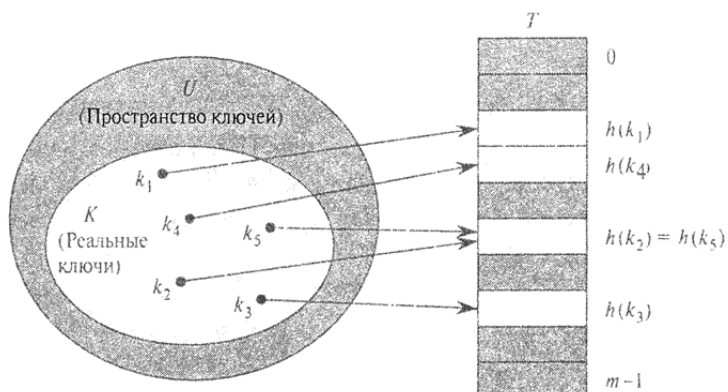


Рис. 6.2. Использование хеш-функции h для отображения ключей в ячейки хеш-таблицы. Ключи k_2 и k_5 отображаются в одну ячейку, вызывая коллизию

Однако здесь есть одна проблема: два ключа могут быть хешированы одну и ту же ячейку. Такая ситуация называется коллизией. К счастью, имеются эффективные технологии для разрешения конфликтов, вызываемых коллизиями.

Конечно, идеальным решением было бы полное устранение коллизий. Мы можем попытаться добиться этого путем выбора подходящей хеш-функции h . Одна из идей заключается в том, чтобы сделать h "случайной", что позволило бы избежать коллизий или хотя бы минимизировать их количество (этот характер функции хеширования отображается в самом глаголе "to hash", который означает "мелко порубить, перемешать"). Само собой разумеется, функция h должна быть детерминистической и для одного и того же значения k всегда давать одно и то же хеш-значение $h(k)$. Однако поскольку $|U| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью избежать коллизий невозможно в принципе, и хорошая хеш-функция в состоянии только минимизировать количество коллизий. Таким образом, нам крайне необходим метод разрешения возникающих коллизий.

6.3. Хеш-функции

В этом разделе мы рассмотрим некоторые вопросы, связанные с разработкой качественных хеш-функций, и познакомимся с тремя схемами их построения. Две из них, хеширование делением и хеширование умножением, эвристичны по своей природе, в то время как третья схема

– универсальное хеширование – использует рандомизацию для обеспечения доказуемого качества.

Качественная хеш-функция удовлетворяет (приближенно) предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из m ячеек, независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно; кроме того, вставляемые ключи могут не быть независимыми.

Иногда распределение вероятностей оказывается известным. Например, если известно, что ключи представляют собой случайные действительные числа, равномерно распределенные в диапазоне $0 \leq k \leq 1$, то хеш-функция $h(k) = [k]$ удовлетворяет условию простого равномерного хеширования.

На практике при построении качественных хеш-функций зачастую используются различные эвристические методики. В процессе построения большую помощь оказывает информация о распределении ключей. Рассмотрим, например, таблицу символов компилятора, в которой ключами служат символьные строки, представляющие идентификаторы в программе. Зачастую в одной программе встречаются похожие идентификаторы, например, *pt* и *pts*. Хорошая хеш – функция должна минимизировать шансы попадания этих идентификаторов в одну ячейку хеш-таблицы.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак не коррелировала с закономерностями, которым могут подчиняться существующие данные. Например, метод деления, который рассматривается в разделе 11.3.1, вычисляет хеш – значение как остаток от деления ключа на некоторое простое число. Если это простое число никак не связано с распределением исходных данных, метод часто дает хорошие результаты.

В заключение заметим, что некоторые приложения хеш-функций могут накладывать дополнительные требования, помимо требований простого равномерного хеширования. Например, мы можем потребовать, чтобы "близкие" в некотором смысле ключи давали далекие хеш-значения (это свойство особенно желательно при использовании линейного исследования). Универсальное хеширование, описанное ниже, зачастую приводит к желаемым результатам.

Для большинства хеш-функций пространство ключей представляется множеством целых неотрицательных чисел $N = \{0, 1, 2, \dots\}$. Если же ключи не являются целыми неотрицательными числами, то можно найти способ их интерпретации как таковых. Например, строка симво-

лов может рассматриваться как целое число, записанное в соответствующей системе счисления. Так, идентификатор pt можно рассматривать как пару десятичных чисел (112,116), поскольку в ASCII-наборе символов $p = 112$ и $t = 116$. Рассматривая pt как число в системе счисления с основанием 128, мы находим, что оно соответствует значению $112 \cdot 128 + 116 = 14452$. В конкретных приложениях обычно не представляет особого труда разработать метод для представления ключей в виде (возможно, больших) целых чисел. Далее при изложении материала мы будем считать, что все ключи представляют целые неотрицательные числа. Рассмотрим альтернативные подходы к конструированию хеш-функций.

Построение хеш-функции методом деления состоит в отображении ключа k в одну из ячеек путем получения остатка от деления k на m , т.е. хеш-функция имеет вид $h(k) = k \bmod m$.

Например, если хеш-таблица имеет размер $m = 12$, а значение ключа $k = 100$, то $h(k) = 4$. Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления считается достаточно быстрым. При использовании данного метода мы обычно стараемся избегать некоторых значений m . Например, m не должно быть степенью 2, поскольку если $m = 2^p$, то $h(k)$ представляет собой просто p младших битов числа k . Если только заранее неизвестно, что все наборы младших p битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа.

Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки. Предположим, например, что мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения $n = 2000$ символьных строк, размер символов в которых равен 8 битам. Нас устраивает проверка в среднем трех элементов при неудачном поиске, так что мы выбираем размер таблицы равным $m = 701$. Число 701 выбрано как простое число, близкое к величине $2000/3$ и не являющееся степенью 2. Рассматривая каждый ключ k как целое число, мы получаем искомую хеш-функцию:

$$h(k) = k \bmod 701.$$

Построение хеш-функции методом умножения выполняется в два этапа. Сначала мы умножаем ключ k на константу $0 < A < 1$ и получаем дробную часть полученного произведения. Затем мы умножаем полученное значение на m применяем к нему функцию "floor" т.е.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

где выражение " $kA \bmod 1$ " означает получение дробной части произведения kA , т.е. величину $kA - \lfloor kA \rfloor$.

Достоинство метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2. Предположим, что разрядность компьютера определяется размером слова w битов и k помещается в одно слово. Ограничим возможные значения константы A значением $s/2^w$, где s – целое число из диапазона $0 < s < 2^w$. Тогда мы сначала умножаем k на w -битовое целое число $s = A \cdot 2^w$. Результат представляет собой $2w$ -битовое число $r_1 2^w + r_0$, где r_1 – старшее слово произведения, а r_0 – младшее. Старшие p битов числа r_0 представляют собой искомое p -битовое хеш-значение (рис. 11.4).

Хотя описанный метод работает с любыми значениями константы A , некоторые значения дают лучшие результаты по сравнению с другими. Оптимальный выбор зависит от характеристик хешируемых данных. В [3] Кнут предложил использовать дающее неплохие результаты значение

$$A \approx (\sqrt{5}-1)/2 \approx 0.6180339887$$

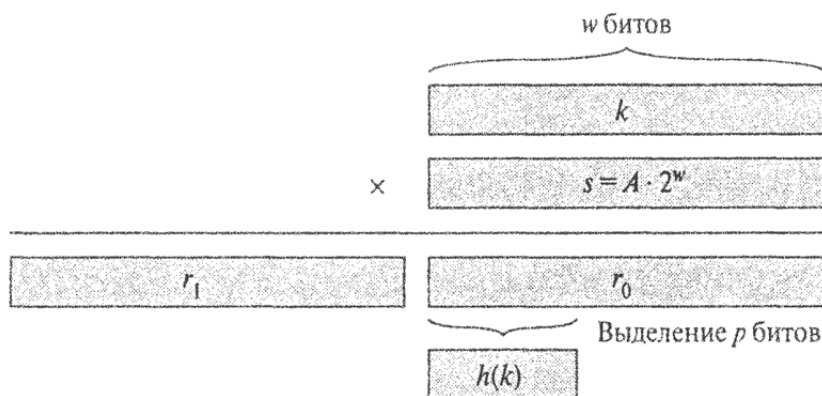


Рис. 5.3. Хеширование методом умножения

Универсальное хеширование. Если недоброжелатель будет умышленно выбирать ключи для хеширования при помощи конкретной хеш-функции, то он сможет подобрать n значений, которые будут хешироваться в одну и ту же ячейку таблицы, приводя к среднему времени выборки $\Theta(n)$. Таким образом, любая фиксированная хеш-функция становится уязвимой, и единственный эффективный выход из ситуации – случайный выбор хеш-функции, не зависящий от того, с какими именно ключами ей предстоит работать. Такой подход, который называется универсальным хешированием, гарантирует хорошую производительность в среднем, независимо от того, какие данные будут выбраны злоумышленником.

Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Как и в случае быстрой сортировки, рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз по-разному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных. Возвращаясь к примеру с таблицей символов компилятора, мы обнаружим, что никакой выбор программистом имен идентификаторов не может привести к постоянному снижению производительности хеширования. Такое снижение возможно только тогда, когда компилятором выбрана случайная хеш-функция, которая приводит к плохому хешированию конкретных входных данных; однако вероятность такой ситуации очень мала и одинакова для любого множества идентификаторов одного и того же размера.

В качестве примера такого семейства можно привести хеш-функцию $h_{a,b}$, где $a \in \{0, 1, \dots, p-1\}$ и $b \in \{1, 2, \dots, p-1\}$, p – большое простое число:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

Этот класс хеш-функций обладает тем свойством, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Поскольку число a можно выбрать $p - 1$ способом, и p способами – число b , всего в данном семействе будет содержаться $p(p - 1)$ хеш-функций.

6.4. Методы разрешения коллизий

Разрешение коллизий при помощи цепочек. При использовании данного метода мы объединяем все элементы, хешированные в одну и ту же ячейку, в связанный список, как показано на рис. 11.3. Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j ; если таких элементов нет, ячейка содержит значение NIL. На рис. 6.4 показано разрешение коллизий, возникающих из-за того, что $h(k_1) = h(k_4)$, $h(k_5) = h(k_2) = h(k_7)$ и $h(k_8) = h(k_6)$.

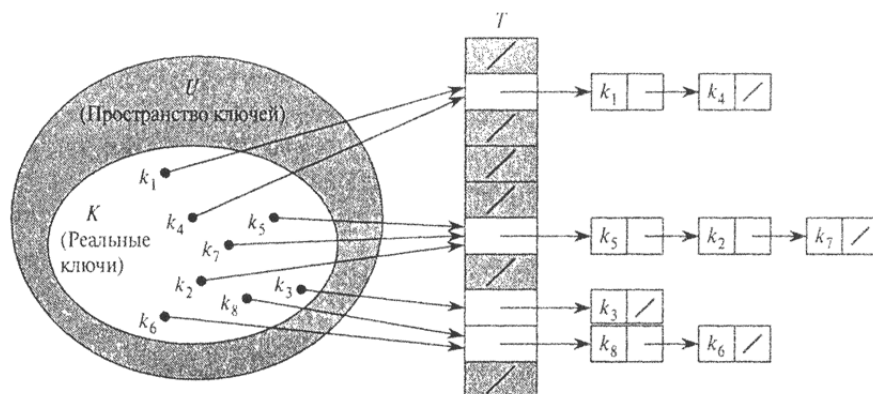


Рис. 6.4. Разрешение коллизий при помощи цепочек

Время, необходимое для вставки в наихудшем случае, равно $O(1)$. Процедура вставки выполняется очень быстро, поскольку предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено путем выполнения поиска перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка. Удаление элемента может быть выполнено за время $O(1)$ при использовании двусвязных списков.

Открытая адресация. При использовании метода открытой адресации все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо значение NIL. При поиске элемента мы систематически проверяем ячейки таблицы до тех пор, пока не найдем искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, в методе открытой адресации хеш-таблица может оказаться заполненной, делая невозможной вставку новых элементов; коэффициент заполнения α не может превышать 1.

Конечно, при хешировании с разрешением коллизий методом цепочек можно использовать свободные места в хеш-таблице для хранения связанных списков, но преимущество открытой адресации заключается в том, что она позволяет полностью отказаться от указателей. Вместо того чтобы следовать по указателям, мы вычисляем последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

Для выполнения вставки при открытой адресации мы последовательно проверяем, или исследуем (probe), ячейки хеш-таблицы до тех пор, пока не находим пустую ячейку, в которую помещаем вставляемый

ключ. Вместо фиксированного порядка исследования ячеек $0, 1, \dots, m - 1$ (для чего требуется $\Theta(n)$ времени), последовательность исследуемых ячеек зависит от вставляемого в таблицу ключа. Для определения исследуемых ячеек мы расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0). В методе открытой адресации требуется, чтобы для каждого ключа k последовательность исследований

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

представляла собой перестановку множества $\langle 0, 1, \dots, m - 1 \rangle$, чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы. В приведенном далее псевдокоде предполагается, что элементы в таблице T представляют собой ключи без сопутствующей информации; ключ k тождественен элементу, содержащему ключ k . Каждая ячейка содержит либо ключ, либо значение NIL (если она не заполнена):

```
HASH_INSERT( $T, k$ )
```

```
1  $i \leftarrow 0$ 
```

```
2 repeat  $j \leftarrow h(k, i)$ 
```

```
3   if  $T[j] = \text{NIL}$ 
```

```
4     then  $T[j] \leftarrow k$ 
```

```
5       return  $j$ 
```

```
6     else  $i \leftarrow i + 1$ 
```

```
7 until  $i = m$ 
```

```
8 error "Хеш-таблица переполнена"
```

Алгоритм поиска ключа k исследует ту же последовательность ячеек, что и алгоритм вставки ключа k . Таким образом, если при поиске встречается пустая ячейка, поиск завершается неуспешно, поскольку ключ k должен был бы быть вставлен в эту ячейку в последовательности исследований, и никак не позже нее. (Мы предполагаем, что удалений из хеш-таблицы не было.) Процедура HASH_SEARCH получает в качестве входных параметров хеш-таблицу T и ключ k и возвращает номер ячейки, которая содержит ключ k (или значение NIL, если ключ в хеш-таблице не обнаружен):

```
HASH_SEARCH( $T, k$ )
```

```
1  $i \leftarrow 0$ 
```

```
2 repeat  $j \leftarrow h(k, i)$ 
```

```
3   if  $T[j] = k$ 
```

```
4     then return  $j$ 
```

```
5      $i \leftarrow i + 1$ 
```

```
6 until  $T[j] = \text{NIL}$  или  $i = m$ 
```

```
7 return NIL
```

Процедура удаления из хеш-таблицы с открытой адресацией достаточно сложна. При удалении ключа из ячейки i мы не можем просто пометить ее значением NIL. Поступив так, мы можем сделать невозможным выборку ключа k , в процессе вставки которого исследовалась и оказалась занятой ячейка i . Одно из решений состоит в том, чтобы пометить такие ячейки специальным значением DELETED вместо NIL. При этом мы должны слегка изменить процедуру HASH_INSERT, с тем, чтобы она рассматривала такую ячейку, как пустую и могла вставить в нее новый ключ. В процедуре HASH_SEARCH никакие изменения не требуются, поскольку мы просто пропускаем такие ячейки при поиске и исследуем следующие ячейки в последовательности. Однако при использовании специального значения DELETED время поиска перестает зависеть от коэффициента заполнения α , и по этой причине, как правило, при необходимости удалений из хеш-таблицы в качестве метода разрешения коллизий выбирается метод цепочек.

Линейное исследование m . Пусть задана обычная хеш-функция $h' : U \leftarrow \{0, 1, \dots, m - 1\}$, которую мы будем в дальнейшем именовать вспомогательной хеш-функцией (auxiliary hash function). Метод линейного исследования для вычисления последовательности исследований использует хеш-функцию

$$h(k, i) = (h'(k) + i) \bmod m,$$

где i принимает значения от 0 до $m - 1$ включительно. Для данного ключа k первой исследуемой ячейкой является $T[h'(k)]$, т.е. ячейка, которую дает вспомогательная хеш-функция. Далее мы исследуем ячейку $T[h'(k) + 1]$ и далее последовательно все до ячейки $T[m - 1]$, после чего переходим в начало таблицы и последовательно исследуем ячейки $T[0]$, $T[1]$, и так до ячейки $T[h'(k) - 1]$. Поскольку начальная исследуемая ячейка однозначно определяет всю последовательность исследований целиком, всего имеется m различных последовательностей.

Линейное исследование легко реализуется, однако с ним связана проблема первичной кластеризации, связанной с созданием длинных последовательностей занятых ячеек, что, само собой, разумеется, увеличивает среднее время поиска. Кластеры возникают в связи с тем, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i + 1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию к все большему удлинению, что приводит к увеличению среднего времени поиска.

Квадратичное исследование. Квадратичное исследование использует хеш-функцию вида

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m,$$

где h' – вспомогательная хеш-функция, c_1 и $c_2 \neq 0$ – вспомогательные константы, а i принимает значения от 0 до $m - 1$ включительно. Начальная исследуемая ячейка – $T[h'(k)]$; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i . Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1 , c_2 и m (в задаче 11-3 показан один из путей выбора этих параметров). Кроме того, если два ключа имеют одну и то же начальную позицию исследования, то одинаковы и последовательности исследования в целом, так как

из $h_1(k, 0) = h_2(k, 0)$ вытекает $h_1(k, i) = h_2(k, i)$. Это свойство приводит к более мягкой вторичной кластеризации. Как и в случае линейного исследования, начальная ячейка определяет всю последовательность, поэтому всего используется m различных последовательностей исследования.

Двойное хеширование. Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок.

Двойное хеширование использует хеш-функцию вида

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

где h_1 и h_2 – вспомогательные хеш-функции. Начальное исследование выполняется в позиции $T[h_1(k)]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю m . Следовательно, в отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа k по двум параметрам – в плане выбора начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа. Последовательность исследований. Производительность двойного хеширования достаточно близка к производительности "идеальной" схемы равномерного хеширования.

СПИСОК ЛИТЕРАТУРЫ

1. Кнут Д. Искусство программирования для ЭВМ. Т.1. Основные алгоритмы: пер. с англ. – М.: Вильямс, 2000.
2. Кнут Д. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск: пер. с англ. – М.: Вильямс, 2000.
3. Вирт Н. Алгоритмы + структуры данных = программы: пер. с англ. – М.: Мир, 1985.
4. Гудман С., Хидетниemi С. Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.
5. Вирт Н. Алгоритмы и структуры данных: пер. с англ. – М.: Мир, 1998.
6. Кормен Т. и др. Алгоритмы. Построение и анализ: пер. с англ. -М.: Вильямс, 2007.
7. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы: пер. с англ. – М.: Вильямс, 2000.
8. Гудрич М.Т., Тамассия Р. Структуры данных и алгоритмы в Java: пер. с англ. – Мн.: Новое знание, 2003.
9. Гасфилд Д. Строки, деревья и последовательности в алгоритмах: пер. с англ. – СПб.: Невский Диалект; БХВ-Петербург, 2003.
10. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: пер. с англ. – К.: ДиаСофт, 2001.
11. Топп У., Форд У. Структуры данных в C++. М.: Бином", 2000.
12. Ключарев А. А., Матяш В. А., Щекин С. В. Структуры и алгоритмы обработки данных: учеб. пособие/СПбГУАП. СПб., 2003.
13. Кузнецов С.Д. Методы сортировки и поиска [Электронный ресурс]. – Режим доступа: <http://citforum.ru/programming/theory/sorting/sorting1.shtml>
14. Brass P. Advanced data structures. CAMBRIDGE UNIVERSITY PRESS, 2008.
15. Mehlhorn K., Sanders P. Algorithms and Data Structures. The Basic Toolbox. – Springer, 2008.
16. J. Edmonds. How to think about algorithms. CAMBRIDGE UNIVERSITY PRESS, 2008.
17. Siena S. The Algorithm Design Manual, 2nd Edition [Электронный ресурс]. – Режим доступа: <http://www.algorist.com/>

18. Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java [Электронный ресурс]. – Режим доступа: <http://www.brpreiss.com/books/opus5/>

Учебное издание

ФОФАНОВ Олег Борисович

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие


Компьютерная верстка *В.В. Михалев*

**Зарегистрировано в Издательстве ТПУ
Размещено на корпоративном портале ТПУ**



Национальный исследовательский Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета
сертифицирована в соответствии с требованиями ISO 9001:2008



ИЗДАТЕЛЬСТВО  **ТПУ**. 634050, г. Томск, пр. Ленина, 30
Тел./факс: 8(3822)56-35-35, www.tpu.ru