Оглавление

Глава 1

Элементы теории алгоритмов

1.1 Понятие алгоритма

1.1.1 Свойства алгоритмов

- дискретность — алгоритм представляет собой последовательность изолированных шагов;
- конечность — алгоритм приводит к получению результата за конечное число шагов;
- детерминированность — результат работы алгоритма зависит только от входных данных;
- понятность — шаги алгоритма могут быть выполнены механически и для их выполнение не требуется применения творческих способностей;
- массовость — алгоритм позволяет решить целый класс однотипных задач (например, алгоритм сложения натуральных чисел «столбиком» позволяет вычислить сумму произвольных чисел).

1.1.2 Алгоритмы как словарные операторы

Алфавитом будем называть произвольное конечное множество. Элементы алфавита будем называть *буквами*. Буквами алфавита могут быть произвольные графические символы, например, буквы латинского алфавита, цифры, знаки препинания и другие символы. Для обозначения алфавита будем использовать заглавные греческие буквы Σ, Δ.

Словом в алфавите Σ называется произвольная последовательность букв этого алфавита. Если Σ содержит буквы русского алфавита, то последова-

тельности *азбука* и *ааазззббб* являются словами в этом алфавите. Число элементов последовательности называется длиной слова. Длина слова w обозначается как $|w|$. Запись $w \in \Sigma^*$ означает, что слово w состоит из букв алфавита Σ, а $w = a_1 a_2 \dots a_n$ означает, что слово w состоит из n букв, причём буквы могут повторяться. Например, если слово $w = a_1 a_2 \dots a_6$ равно *азбука*, то значениями a_1 и a_6 является русская буква *а*.

Пустым словом называется слово длины 0. Пустое слово будем обозначать символом ε. Множество всех слов в алфавите Σ обозначается как Σ^*.

Каждая задача может быть представлена в виде слова в некотором алфавите. Например, описанием задачи может быть её описание на русском языке. Результат выполнения алгоритма также может быть представлен в виде слова в некотором подходящем алфавите и алгоритмы можно рассматривать как словарные операторы:

$$\mathcal{A} : \Sigma^* \to \Delta^*.$$

Такая запись означает, что $\mathcal{A}$ принимает на вход слово в алфавите Σ и перерабатывает его в слово в алфавите Δ. Алфавиты Σ и Δ могут совпадать, но в общем случае они различны.

Рассмотрим в качестве примера алгоритм Sum сложения натуральных чисел. Входным алфавитом является $\Sigma = \{0, 1, \dots, 9, +\}$, который содержит арабские цифры и символ $+$. Выходным алфавитом является $\Delta = \{0, 1, \dots, 9\}$. Действие алгоритма может быть задано как

$$Sum(w) = \begin{cases} \text{десятичная запись } n + k, & \text{если входное слово имеет вид } n + k \\ \varepsilon, & \text{иначе.} \end{cases}$$

Например, входное слово $12 + 54$ будет переработано алгоритмом Sum в слово 66, а слово 623 — в пустое слово, так как оно не является корректным описанием задачи сложения натуральных чисел.

1.2 Нормальные алгоритмы Маркова

В терминах алгоритмов Маркова вычисления представляются в виде последовательности преобразования входного слова. Один шаг алгоритма — это замена в слове одной цепочки символов на другую. Правила замены определяются парой слов. Первый элемент пары является последовательностью символов, которую нужно заменить, а второй элемент — на что заменить. Правило подстановки записывается в виде $u \to v$, где $u, v \in \Sigma^*$ — слова в некотором алфавите.

Будем говорить, что правило $u \to v$ *применимо* к слову w, если слово w содержит u в качестве подслова $(\exists \alpha, \beta \in \Sigma^* : w = \alpha u \beta)$. Например, правило *мама* $\to$ *папа* применимо к слову *мамаша* ($\alpha = \varepsilon$ и $\beta =$ *ша*) и неприменимо к слову *бабушка*.

Результатом применения правила $u \to v$ к слову w является слово w' в котором произведена замена первого вхождения левой части правила на его правую часть. В приведённом примере результатом будет слово *папаша*. Если левая часть встречается в слове несколько раз, то заменяется самое левое (первое) вхождение. Например, слово *мама-мама* будет преобразовано в *папа-мама*.

Формально замена первого вхождения определется так. Пусть $w = \alpha v \beta$ для некоторых $\alpha, \beta \in \Sigma^*$, причем $\forall \alpha', \beta' \in \Sigma^* \quad |\alpha'| < |\alpha| \Rightarrow w \neq \alpha' u \beta'$. Тогда результатом применения правила $u \to v$ к слову w является слово $w' = \alpha v \beta$.

Нормальный алгоритм Маркова представляет совой упорядоченную последовательность правил подстановок, некоторые из которых отмечены как заключительные. Графически заключительные правила обозначаются символом точки после стрелки. В приведенном ниже примере второе правило является заключительным.

$$\begin{array}{lll} u_1 & \to & v_1 \\ u_2 & \to . & v_2 \\ \ldots & \ldots & \ldots \\ u_n & \to & v_n \end{array}$$

Такие правила определяют правила преобразования входного слова. Вычисления на входном слове w представляются в виде последовательности слов $w_0, w_1, \ldots$, где $w_0 = w$. Слово w_{i+1} получается из слова w_i выполнением подстановки, заданной применимым правилом с наименьшим индексом. Вычисления заканчиваются, если

- либо ни одно из правил не применимо к текущему слову w_i;
- либо было применено заключительное правило.

Если на входном слове w алгоритм A заканчивает работу за конечное число шагов, то говорят, что значение A на слове w *определено*. Если же ни за какое конечное число шагов выполнение алгоритма не приводит к результату, то говорят, что значение A на слове w *не определено*.

Рассмотрим следующий пример нормального алгоритма, преобразующего слова в алфавите $\Sigma = \{a, b\}$.

1 bbb $\to$. aba
2 bba $\to$ b
3 aba $\to$ ba
4 bab $\to$ ba

Пусть на вход поступает слово $w = babbaa$. Тогда последовательность преобразований, которые выполняет данный алгоритм, можно представить в виде следующей таблицы (подчеркнуты первые вхождения левых частей правил и результат их замены на правые части).

Вход: Упорядоченное набор из n правил $(u_1, v_1), \ldots, (u_n, v_n)$,
$F \subseteq \{1, \ldots, n\}$ — множество индексов заключительных правил,
w — входное слово
Результат: Выходное слово

```
1  repeat
2  |  i ← 0;                    // Номер последнего применённого правила
3  |  Найдём первое применимое правило;
4  |  k ← 1;
5  |  while k ⩽ n и i = 0 do
6  |  |  if Правило u_k → v_k применимо к w then
7  |  |  |  i ← k;
8  |  |  |  w ← Применить(w, u_i → v_i);
9  |  |  end
10 |  |  k ← k+1;
11 |  end
12 until i > 0 и i ∉ F;
13 return w;
```

Рис. 1.1: Выполнение нормального алгоритма

№ правила	исходное слово	результат применения
2	$ba\underline{bba}$	$ba\underline{=}a$
3	$b\underline{aba}$	$b\underline{\underline{=}a}$
2	$\underline{bba}$	$\underline{=}$

Первыми будет применено правило 2, левая часть которого встречается в исходном слове. Левая часть правила 4 также встречается в слове w, но согласно правилу выполнения нормального алгоритма к слову применяется первое по порядку применимое правило. Слово $w_3 = b$ является результатом работы алгоритма, так как ни одно из правил не может быть к нему применено.

Если же на вход приведенного алгоритма подается слово $w' = bbabbaa$, полученное приписыванием в начало слова w буквы b, то цепочка преобразований будет другой: $\underline{bba}bbaa = \underline{bbb}aa = ababaa$. Последнее слово является результатом, так как оно было получено путем применения заключительного правила 1 и его дальнейшие преобразования не производятся.

Словарные операторы могут быть определены различным образом, в том числе, на естественном языке. Нормальные алгоритмы, напротив, определяют *процедуру вычисления* значения словарного оператора, причем эта процедура обладает свойствами алгоритма. Отдельные шаги этой процедуры могут быть выполнены механически (требуется произвести замену перового вхождения левой части правила на его правую часть). В таких случаях говорят, что словарный оператор определен конструктивно. Под алгоритмами при-

нято понимать конструктивно заданную процедуру вычисления словарного оператора.

Для одного словарного оператора существует, вообще говоря, бесконечно много способов его конструктивного задания. Два алгоритма называются *равными*, если они совпадают как системы правил. Алгоритмы называются *эквивалентными*, если они не равны, но определяют один и тот же словарный оператор. Эквивалентные алгоритмы можно представлять себе как разные программы, которые решают одну задачу, — тексты программ различаются, однако если на их вход подаются одинаковые данные, то на выходе получаются одинаковые результаты.

Говорят, что алгоритм A *нормализуем*, если существует эквивалентный ему нормальный алгоритм Маркова. Тезис Маркова состоит в том, что *любой алгоритм нормализуем*, то есть что любой алгоритм может быть представлен в виде нормального алгоритма Маркова. Слово «тезис» говорит о том, что это утверждение нельзя строго доказать (так как нет определения понятия «алгоритм»), однако все алгоритмы, которые известны на сегодня, этому тезису удовлетворяют.

1.2.1 Способы композиции алгоритмов

Методы композиции алгоритмов используются для построения новых алгоритмов на основе уже существующих алгоритмов. Методы композиции включают суперпозицию, ветвление, повторение и объединение.

Алгоритм C называется *суперпозицией* алгоритмов A и B, если для любого входного слова w значение $C(w)$ получается применением алгоритма A к результату вычисления B на слове w, то есть $C(w) = A(B(w))$. Если алгоритм B неприменим к слову w, то значение $C(w)$ не определено.

Алгоритм D называется *ветвлением*, если значение для любого слова w, удовлетворяющего условию $C(w) = \varepsilon$, $D(w) = A(w)$ и для всех остальных слов $D(w) = B(w)$. Алгоритм C является условием ветвления.

Алгоритм D называется *итерацией* (или повторением) алгоритма A с условием C, если для любого входного слова w найдётся такая цепочка слов $w_0, w_1, \ldots, w_k$, что

- $w_0 = w$;
- для любого индекса $i = 1, 2, \ldots, k$ слово w_i равно результату применения алгоритма A к слову w_{i-1};
- для любого индекса $i = 0, 1, 2, \ldots, k-1$ значение слово w_i перерабатывается алгоритмом C в пустое слово, то есть $C(w_i) = \varepsilon$;
- и $C(w_k) \neq \varepsilon$.

Фактически итерация означает «Выполнять алгоритм A до тех пор, пока условие C выполнено».

Алгоритм называют *линейным*, если он не использует ветвления и итерации, *ветвящимся* — если при его определении используется метод композиции «ветвление», и *циклическим*, если в нем используется итерация.

Алгоритм C называют *объединением* алгоритмов A и B, если результат $C(w)$ получается приписыванием слова $B(w)$ к слову $A(w)$.

1.3 Сложность вычислений

Временно́й сложностью алгоритма называется количество шагов, которое необходимо для завершения работы алгоритма на входных данных. *Ёмкостная сложность* — это максимальный объем памяти, который требуется алгоритму в процессе его работы. В терминах алгоритмов Маркова временна́я сложность соответствует числу правил, которые были применены, а ёмкостная — максимальной длине слова, которое возникает в процессе выполнения алгоритма.

Сложность алгоритмов оценивают асимптотически, то есть при неограниченном росте длины входного слова. Обычно оценивается сложность *в худшем случае*. Обозначим через $T(A, w)$ число шагов, которое требуется алгоритму A для обработки входного слова w, а через $T_n(A)$ — максимальное значение $T(A, w)$ для всех слов длины n:

$$T_n(A) = \max_{w \in \Sigma^* : |w| = n} T(A, w).$$

Тогда утверждение вида «Алгоритм A имеет сложность $O(n^2)$ (или имеет сложность порядка n^2)» означает, что найдутся такие константы $C \in \mathbb{R}$ и n_0, что для любого $n > n_0$ выполняется неравенство $T_n(A) \leqslant Cn^2$. Практически можно считать, что для любого достаточно большого n найдется такое входное слово w длины n, что алгоритму A потребуется не более n^2 шагов для его обработки. Более сильное условие «Алгоритм A имеет сложность $\Omega(n^2)$» означает, что $C_1 n^2 \leqslant T_n(A) \leqslant C_2 n^2$ для некоторых констант C_1 и C_2. Алгоритмы, сложность которых ограничена полиномом от n, называются *полиномиальными*.

Например, алгоритм

a $\to$ b
b $\to$ ε

на любом входном слове длины n в алфавите $\Sigma = \{a, b\}$ сделает не более $2n$ шагов (самым «сложным» для него является слово из n букв a) и его сложность составляет $O(n)$.

Сложность задачи, то есть сложность реализации заданного словарного оператора $\mathcal{A} : \Sigma^* \to \Delta^*$, определяется как сложность наилучшего алгоритма,

который решает эту задачу:

$$C(\mathcal{A}) = \min_{A} C(A),$$

где минимум вычисляется по все алгоритмам A, реализующим оператор $\mathcal{A}$. Определить сложность задачи гораздо сложнее, чем сложность конкретного алгоритма, так как существует бесконечно много эквивалентных алгоритмов.

1.4 Универсальный алгоритм

Каждый алгоритм Маркова может быть представлен в виде слова в некотором алфавите. Без ограничения общности предположим, что алфавит в котором задан алгоритм $A : \Sigma^* \to \Delta^*$, не содержит символов «$\to$», «.» и «;», то есть ни одно из правил нормального алгоритма, определяющих алгоритм A, не содержат этих символов. Тогда в качестве такого слова может использоваться запись алгоритма: правила записываются одно за другим и разделяются символом «;». Например, алгоритм

ab $\to$ ba
ba $\to$. ε
aa $\to$ ε

кодируется словом $ab \to ba; ba \to .\varepsilon; aa \to \varepsilon$. Код нормального алгоритма A будем обозначать как A_u, то есть A_u является *словом* в алфавите алгоритма A к которому добавлены символы «$\to$», «.» и «;».

Универсальным алгоритмом называется такой алгоритм, который для любого входного слова вида $A_u; ; w$ вычисляет значение алгоритма A на входном слове w. На вход универсального алгоритма фактически поступает описание произвольного алгоритма A и некоторое слово w. Два символа «;» обозначают конец A_u. Универсальный алгоритм вычисляет результат применения алгоритма A к слову w. Другими словами универсальный алгоритм может выполнить *любой алгоритм*.

Теорема об универсальном алгоритме утверждает, что универсальный алгоритм нормализуем.

1.5 Алгоритмически неразрешимые задачи

Свойство массовости алгоритма означает, что алгоритм решения задачи позволяет применим к целому классу однотипных задач. Задача называется *алгоритмически неразрешимой*, если не существует алгоритма, который позволяет единым методом решить любую задачу из класса. При этом могут существовать конкретные задачи данного типа решение которых возможно.

1.5.1 Неразрешимость проблемы определения самопроименимости

Обозначим через A_u слово, которое является кодом алгоритма A для универсального алгоритма Маркова. Назовём алгоритм A *самоприменимым*, если значение $A(A_u)$ определено[1] и *несамоприменимым*, если значение $A(A_u)$ не определено. Задача распознавания свойства самоприменимости состоит в нахождении алгоритма, который по заданному слову A_u определяет, является ли алгоритм A самоприменимым или нет.

Теорема 1. *Не существует нормального алгоритма Маркова, который распознает свойство самоприменимости.*

Доказательство. Предположим, что алгоритм проверки самоприменимости существует. Обозначим его через C. Его действие определяется, например, как

$$C(w) = \begin{cases} \varepsilon, & \text{если } w \text{ является кодом самоприменимого алгоритма} \\ w, & \text{иначе.} \end{cases}$$

Построим алгоритм D который определён на всех словах, которые задают несамоприменимые алгоритмы, и только на них: на любом другом слове алгоритм D не определён, то есть не выдаёт никакого результата. Алгоритм D легко может быть построен на основе алгоритма C, алгоритма I, определённого на всех словах, и алгоритма L, не определённого ни на одном слове[2]. Сначала ко входному слову w применяется алгоритм C. Если на выходе получается слово w, то слово w не является кодом самоприменимого алгоритма и результатом алгоритма D является результат применения I. Если же C возвращает пустое слово, то применяем алгоритм L.

Теперь рассмотрим действие D на слове D_u, то есть применим его к своему коду. Возможны два варианта: либо значение $D(D_u)$ определено, либо не определено. Если $D(D_u)$ определено, то алгоритм D является самоприменимым (по определению понятия самоприменимости, так как он выдал результат на своем коде). Однако по построению алгоритм D определён только на словах, задающих несамоприменимые алгоритмы, а значит он не должен выдать никакого результата на своём коде. Противоречие. Если же значение $D(D_u)$ не определено, то он является несамоприменимым (по определению) и должен быть применим к своему коду (по построению). Ни один из вариантов не может иметь места, а значит предположение о существовании алгоритма C ошибочно. □

[1] A_u является некоторым словом. Если A самоприменим, то он перерабатывает A_u в какое-то слово за конечное число шагов.

[2] Алгоритм I может быть задан с помощью правила $\varepsilon \to .\varepsilon$, а алгоритм L — с помощью правила $\varepsilon \to \varepsilon$ (без точки).

1.5.2 Другие неразрешимые задачи

Проблема равенства слов. Пусть задан конечный набор пар слов (u_1, v_1), $\ldots$, (u_n, v_n) в некотором алфавите Σ, которые называются соотношениями. Будем говорить, что слово слово w' *непосредственно выводится* из слова w, если найдётся такой номер $i \in \{1, 2, \ldots, n\}$ и слова $\alpha, \beta \in \Sigma^*$, что либо $w = \alpha u_i \beta$ и $w' = \alpha v_i \beta$, либо $w = \alpha v_i \beta$ и $w' = \alpha u_i \beta$. Другими словами слово w' непосредственно выводится из w, если оно получается выполнением одной операцией замены u_i на v_i или v_i на u_i (замена может быть произведена в любом месте слова w). Слова w и w' называются равными, если существует такая цепочка слов $w_0, w_1, \ldots, w_k$, что $w_0 = w$, $w_k = w'$ и для любого $i \in \{1, 2, \ldots, k\}$ слово w_i непосредственно выводится из w_{i-1}. **Пример**: если $\Sigma = \{a, b\}$ и задано единственное соотношение $ab = ba$, то любые два слова с одинаковым количеством букв a и b являются равными.

Не существует алгоритма, который для произвольной системы соотношений $(u_1, v_1), \ldots, (u_n, v_n)$ и двух слов w и w' проверяет их равенство. (Такой алгоритм должен всегда заканчивать свою работу и возвращать два различных значения. Например, пустое слово, если слова w и w' равны, и не пустое слово, если слова не равны.) Заметим, что в приведённом выше примере задача равенства слов разрешима (достаточно посчитать количество букв a и b).

Проблема остановки. Не существует алгоритма, который по описанию произвольного алгоритма A и входному слову w проверит, что значение A на слове w определено (что алгоритм A закончит свою работу, если ему на вход подаётся слово w).

Выполнимость оператора. Пусть задана программа на языке ассемблера (см. следующую главу) и указана конкретная строчка в этой программе. В общем случае невозможно определить, существуют ли такие входные данные, при которых указанный оператор будет выполнен.

Проблема соответствий Пóста. Пусть задан конечный набор пар слов $(u_1, v_1), \ldots, (u_n, v_n)$. Сопоставим каждой последовательности $i_1, i_2, \ldots, i_k$ элементов множества $\{1, \ldots, n\}$ пару слов (u, v) следующим образом. Слово u получается выписыванием левых частей элементов набора, то есть $u = u_{i_1} u_{i_2} \ldots u_{i_k}$, а слово v — последовательным выписыванием соответствующих правых частей, $v = v_{i_1} v_{i_2} \ldots v_{i_k}$. Проблема соответствий Пóста — найти алгоритм, который для любого набора пар слов проверяет, существует ли такая последовательность $i_1, i_2, \ldots, i_k$, что соответствующие ей слова совпадают. **Пример**: рассмотрим набор из двух эелементов $(aab, a), (b, baabb)$. Решением является последовательнсоть $1, 1, 2$, так как

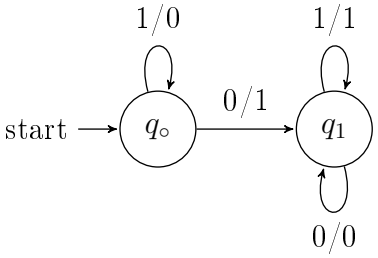


Рис. 1.2: Пример конечного автомата, который прибавляет единицу к двоичному представлению числа, записанному в обратном порядке.

$u_1u_1u_2 = aab\ aab\ b = a\ a\ baabb = v_1v_1v_2$.

1.6 Машина Тьюринга

Сначала рассмотрим более простую модель вычислительного устройства, известную как *конечный автомат*. Конечный автомат — это устройство, которое считывает входное слово по одному символу и печатает соответствующее выходное слово. В каждый момент времени автомат может находтся в одном из состояний некоторого конечного множестве состяний (поэтому он называется конечным автоматом). Текущее состояние автомата изменятся при обработке входного слова по чётко определеннму правилу. В зависимости от своего текущего состояния и буквы входного слова автомат выводит очередную букву выходного слова.

Формально конечный автомат определяется следующеим набором:

- конечными алфавитами Σ и Δ (входной и выходной алфавит автомата);
- конечным множеством $Q = \{q_1, \ldots, q_n\}$, элементы которого называются *состояниями* автомата;
- выделенным *начальным состоянием* $q_\circ \in Q$;
- *функцией переходов* $\delta : Q \times \Sigma \to Q \times \Delta$, которая определяет следуещее состояние автомата и символ, который необходимо вывести.

Визуально конечный автомат может быть представлен в виде ориентированного помеченного графа. На рисунке 1.2 представлен пример конечного автомата. Здесь $\Sigma = \Delta = \{0, 1\}$, множество допустимых состояний автомата содержит два элемента $Q = \{q_\circ, q_1\}$, начальное состояние, отмеченное стрелкой start, есть $q_\circ$, а функция переходов задается как $\{(q_\circ, 0, q_1, 1), (q_\circ, 1, q_\circ, 1), (q_1, 0, q_1, 0), (q_1, 1, q_1, 1)\}$. Метка ребра, например, 0/1 между вершинами $q_\circ$ и q_1 означает, что $(q_1, 1) = \delta(q_\circ, 0)$.

Действие автомата при поступлении входного слова $w = a_1a_2 \ldots a_k$ в алфавите Σ определяется следующим образом. Сначала автомат находится в

своем начальном состоянии $q_\circ$. Далее он считывает одну букву входного слова, то есть a_1, и определяет свое следующее состояние q' и букву выходного слова b на основании функции перехода: $(q', b) = \delta(q, a)$. Автомат переходит в состояние q', печатает букву b и повторяет аналогичную процедуру для следующей буквы входного слова. Таким образом, конечный автомат определяет некоторый словарный оператор $A : \Sigma^* \to \Delta^*$.

Рассмотрим в качестве примера действие автомата, изображенного на рисунке 1.2 при поступлении входного слова 111001 в алфавите $\{0, 1\}$. Это слово является обратной записью двоичного представления числа $39 = 1+2+4+32$ (слева записан *младший* разряд двоичного представления числа). При поступлении первого символа 1 автомат находится в начальном состоянии и переходит по стрелке с меткой 1/0, то есть печатает букву 0 и остается в состоянии $q_\circ$. Вторая и третья единицы входного слова аналогичным образом преобразуются в 0, а автомат остается в состоянии $q_\circ$. После прочтения первого символа 0 (четвертой буквы входного слова) автомат печатает 1 и переходит в состояние q_1. В этом состоянии автомат копирует остающуюся часть входного слова без изменения. Протокол работы автомата можно представить в виде следующей таблицы:

Текущее состояние	$q_\circ$	$q_\circ$	$q_\circ$	q_1	q_1	q_1
Входная буква	1	1	1	0	0	1
Выходная буква	0	0	0	1	0	1

Заметим, что выходное слово является обратной двоичной записью числа $2^3 + 2^5 = 8 + 32 = 40$. Этот автомат прибавляет единицу.

Второй распространенной формализацией понятия алгоритма является *машина Тьюринга*. Машина Тьюринга состоит из:

- бесконечной ленты, разделенной *ячейки*, каждая из которых может хранить одну букву некоторого конечного алфавита;
- управляющего устройства (или *головки* чтения и записи машины Тьюринга), которое может перемещаться по ленте, находится в конечном числе *состояний* и может считывать и изменять содержимое той ячейки, на которой оно находится.

Управляющее устройство по сути представляет собой конечный автомат.

Программа для машины Тьюринга состоит из последовательности команд следующего вида. **Если** управляющее устройство находится в состоянии q и текущая ячейка ленты содержит символ x, **то** необходимо записать в текущую ячейку символ y, перевести управляющее устройство в состояние q' и сдвинуть считывающую головку на одну позицию вправо, влево или оставить ее на месте. Каждая команда может кодироваться выражением вида $(q, x) \mapsto (q', y, s)$, где $q, q' \in Q$, $x, y \in \Sigma$, $s \in \{-1, 0, +1\}$. Например, команда $(3, a) \mapsto (4, b, +1)$ означает, что если машина находится в состоянии 3,

текущий символ ленты равен a, то необходимо записать в ячейку символ b, перейти в состояние 4 и сдвинуть головку вправо.

Машины Тьюринга эквиваленты алгоритм Маркова. Например, можно составить алгоритм Маркова, который выполняет произвольную машину Тьюринга (аналогично универсальному алгоритму Маркова). Верно и обратное утверждение. Можно построить машину Тьюринга, которая будет выполнять произвольный алгоритм Маркова.

Упражнения

1. Составьте нормальный алгоритм Маркова, который любое слово нечетной длины переводит в среднюю букву, а слово четной длины — в пустое слово. Например, $A(abb) = b$, $A(bbaa) = \varepsilon$.

2. Составьте нормальный алгоритм Маркова, который любое слово $w = a_1a_2\ldots a_n$ в алфавите $\Sigma = \{a, b\}$ длины $n > 1$ преобразует в букву a_2.

3. Составьте нормальный алгоритм Маркова, который в любом слове $w = a_1a_2\ldots a_n$ а алфавите $\Sigma = \{a, b\}$ заменяет буквы a на b и b на a. Например, $ababba \mapsto babaab$.

4. Составьте нормальный алгоритм Маркова, который в любом слове $w = a_1a_2\ldots a_n$ а алфавите $\Sigma = \{a, b\}$ удаляет последовательности подряд идущих букв a на одну букву a, но только если по краям стоят буквы b. $aaabaaaaaaaabaa \mapsto aaababaa$.

5. Составьте нормальный алгоритм Маркова, который в любом слове $w = a_1a_2\ldots a_n$ а алфавите $\Sigma = \{a, b\}$ «удваивает» буквы a и b. Например, $ababba \mapsto aabbaabbbbaa$.

6. Составьте нормальный алгоритм Маркова, который любое слово $w = a_1a_2\ldots a_n$ а алфавите $\Sigma = \{a, b\}$ переводит в слово $a_na_1a_2\ldots a_{n-1}$. Например, $ababb \mapsto babab$.

Глава 2

Упрощённый ассемблер

2.1 Архитектура современных ЭВМ

Современные компьютеры построены на основе принципа произвольного доступа к памяти и принципа хранимой программы, которые описываются ниже.

К числу основных компонент современных вычислительных систем относятся *процессор* (арифметическо-логическое устройство, АЛУ) и *оперативная память*. Процессор выполняет операции с данными, такие как умножение или сложение, в соответствии с инструкциями программы. Оперативная память представляет собой бесконечную последовательность однотипных ячеек, каждая из которых может хранить целое число. Каждая ячейка памяти имеет уникальный *адрес* (порядковый номер) и процессор может прочитать или изменить содержимое любой ячейки, указав её адрес. Такая модель называется *моделью с произвольным доступом к памяти (random access memory, RAM)*, что подчёркивает её отличие от алгоритмов Маркова и других абстрактных моделей, которые не допускают изменения произвольных ячеек[1].

Принцип хранимой программы означает, что инструкции, которые должен выполнять процессор, также хранятся в оперативной памяти. Каждая инструкция и необходимые для её выполнения данные кодируются некоторыми числами. Например, команда сложения содержимого ячеек с адресами 804 и 312 и записи результата в ячейку 911 может кодироваться четырьмя числами $61, 804, 312, 911$, где число 61 является кодом операции сложения. Вся программа (последовательность действий, которую должен выполнить процессор) размещается в подряд идущих ячейках оперативной памяти. Процессор «знает» в какой ячейке памяти находится команда, которую он должен выполнить следующей и команды выполняются последовательно. Если в этой и последующих ячейках расположена последовательность

[1] В модели Маркова аналогом ячейки памяти можно считать букву слова. Правило подстановки $u \to v$ не может изменить произвольную букву текущего слова — его действие ограничено длиной слов u и v.

61, 804, 312, 911, 61, 911, 805, 312, то сначала сумма значений ячеек 804 и 312 будет записана в ячейку 911 (первая команда 61), а потом сумма значений ячеек с адресами 911 и 805 будет записана в ячейку 312.

2.2 Язык ассемблера

Рассмотрим упрощённый язык ассемблера с одним регистром. Будем считать, что процессор содержит одну ячейку памяти, которая называется *регистр* и используется для выполнения арифметических операций. Команды данного языка разделяются на следующие классы:

- команды ввода-вывода, которые позволяют считывать из входного потока (с клавиатуры) или выводить в выходной поток (на экран) числа;
- команды работы с памятью, которые позволяют загружать в регистр содержимое ячеек памяти и записывать в ячейки памяти значение регистра;
- комады выполнения арифметических операций;
- команды управлением последовательностью выполнения команд.

Описание команд приведено в таблице 2.1.

Программа на языке ассемблера представляется в виде последовательности команд. Команды выполняются последовательно. Например, программа

```
READ
MUL =2
WRITE
```

считывает из входного потока число и помещает его в регистр (READ), после чего умножает содержимое регистра на два и помещает результат обратно в регистр (MUL =2) и выводит результат в выходной поток (WRITE). Таким образом, приведённая выше программа является программой умножения на 2.

Некоторые команды в программе могут быть помечены *метками* — произвольными последовательностями символов. Метки, в сочетании с командами условного или безусловного перехода, используются для изменения порядка выполнения команд программы. Приведённая ниже программа считывает число и, если введённое значение отлично от нуля, выводит в выходной поток константу 1.

```
       READ
       JZERO ноль
       LOAD =1
       WRITE
ноль:  HALT
```

Вторая команда в данной программе называется *условным переходом*. Если в момент выполнения этой команды значением регистра является 0, то JZERO изменяет естественный порядок выполнения команд. Вместо команды LOAD =1 будет выполнена команда, помеченная меткой *ноль*. Если же в момент выполнения команды JZERO значением регистра является отличное от нуля число, то порядок выполнения не изменяется. Существует команда переход `JUMP`, которая меняет порядок выполнения программы независимо от текущего значения регистра. Такая команда называется безусловным переходом.

Программа на ассемблере, которая не использует команд перехода, всегда заканчивает свое выполнение. Если в программе есть условные или безусловные переходы, то вычисления могут продолжаться бесконечно долго. Например, программа, состоящая из команд

```
цикл: JUMP цикл
      HALT
```

никогда не закончит работу, так как команда HALT не выполняется.

команда	аргумент	описание
READ	нет	прочитать из входного потока число и записать его в регистр
WRITE	нет	записать в выходной поток (напечатать) содержимое регистра
LOAD	=n	записать в регистр число n
	n	записать в регистр значение ячейки с адресом n
	*n	записать в регистр значение ячейки, номер которой записан в ячейке с адресом n
STORE	n	записать в ячейку с адресом n содержимое регистра
	*n	записать содержимое регистра в ячейку, адрес которой находится в ячейке n
ADD SUB MUL DIV	=n n *n	в регистр записывается результат выполнения операции сложения (ADD), вычитания (SUB), умножения (MUL) или деления (DIV) содержимого регистра и значения аргумента; значением =n является число n, значением n является содержимое ячейки n, значением *n является содержимое ячейки, адрес которой записан в ячейке n

JUMP	метка	безусловный переход; управление передаётся на команду с меткой *метка*
JZERO	метка	если в регистре 0, то управление передаётся на команду с меткой *метка*
JGTZERO	метка	если значение регистра строго больше 0, то управление передаётся на команду с меткой *метка*
JLTZERO	метка	если значение регистра строго меньше 0, то управление передаётся на команду с меткой *метка*
HALT	нет	остановка программы

Таблица 2.1: Система команд упрощённого ассемблера с одним регистром

Операции с памятью. Модель произвольного доступа к памяти предполагает возможность непосредственного обращения к любой ячейке оперативной памяти. В нашем языке для загрузки данных из памяти в регистр используется команда LOAD. Её действие состоит в записи значения аргумента в регистр (изменение значения регистра). Команда LOAD имеет три варианта, отличающиеся формой своего аргумента. Самым простой формой аргумента команды является запись =n, где n — произвольное число. Значением «=n» является число z. Например, если процессор выполняет команду LOAD =123, то в регистр записывается число 123, а при выполнении команды LOAD =-123 в регистр попадет отрицательное число -123. Значения ячеек памяти при выполнении такой команды не изменяется.

Второй формой аргумента является «n» — положительное число, записанное без знака равенства, например LOAD 16. Значением такого аргумента является содержимое ячейки памяти с данным адресом. Если в ячейке с адресом 16 будет записано число 100, то выполнение команды LOAD 16 приведёт к тому, что регистр будет иметь значение 100. Такая операция называется *прямой адресацией* (мы явно указали из какой ячейки оперативной памяти нужно загрузить значение в регистр). Значение ячейки 16 *не изменяется*.

Последняя форма задания аргумента — «*n». Значением такого аргумента является содержимое ячейки, адрес которой записан в ячейке n. Такая операция называется *косвенной адресацией* и более подробно рассматривается позже. Схематично три типа аргументов команд представлены на рис. 2.1. В зависимости от выполняемой команды в регистр попадает либо число 10, либо содержимое ячейки с адресом 10, либо число B, которое находится в ячейке A.

Опреация записи в память выполнется аналогично. Команда STORE n записывает содержимое регистра в ячейку с адресом n, а команда STORE *n записывает содеримое регистра в ячейку, адрес которой находится в ячейке

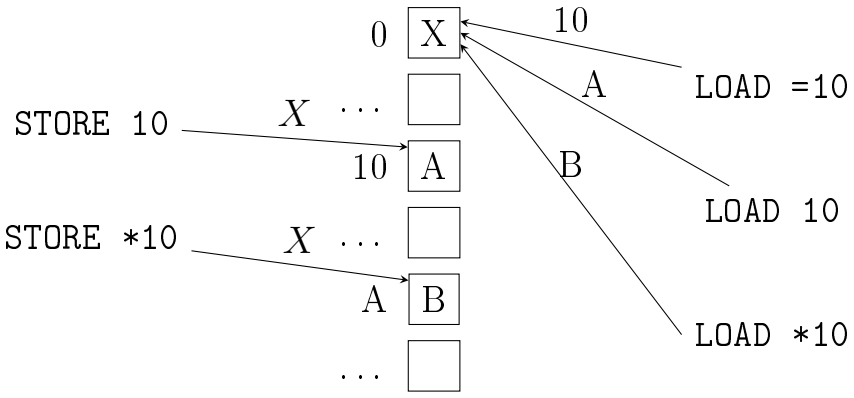


Рис. 2.1: Действие команд LOAD и STORE для различных типов аргументов.

n. Команда STORE =n недопустима, так как константам нельзя присваивать значения.

Вызов команд LOAD и STORE с нулевым или отрицательным значением аргумента недопустим и приводит к ошибке. Также ошибочным является использование значений ячеек, которые не были записаны в ходе выполнени программы. При запуске программы каждая ячейка памяти имеет некоторое значение, но оно может изменяться от запуска к запуску. Программа, состоящая из двух команд

```
LOAD 10
WRITE
```

выводит, вообще говоря, произвольное число.

Арифметические операции. Все арифметические операции выполняются с регистром и единственным аргументом операции. Регистр является левым опрандом выполняемого действия, а значение аргумента — правым. Результат выполнения операции помещается в регистр. Значение аргумента вычисляется аналогично правилам команды LOAD.

Предположим, что в регистре находится число 4, в ячейке 3 находится число 6, а в ячейке 6 — число -2. Тогда после выполнения команды ADD =3 в регистре будет число 7, а после выполнения MUL 3 (в регистре 4) в регистр записыватся число $4 * 6$. Если же выполняется команда DIV *3, то в регистр будет записано число $4/(-2) = -2$. При выполнении арифметических операций изменяется только содержимое регистра.

Циклические конструкции. Во многих случаях возникает необходимость выполнения однотипных операций для входных данных, объём которых неизвестен в момент написания программы. Рассмотрим в качестве примера задачу нахождения максимального элемента последовательности,

Вход: Элементы последовательности считываются из входного потока
Результат: Максимальный элемент

```
1 n ← Прочитать;
2 max ← n;
3 while n ≠ 0 do          // Пока последовательность не закончилась
4 |  if n > max then
5 |  |  max ← n;                        // Встретили новый максимум
6 |  end
7 |  n ← Прочитать;
8 end
9 Напечатать max;
```

Рис. 2.2: Нахождение максимального элемента

которая считывается из входного потока и заканчивается числом 0 (при считывании числа 0). Алгоритм решения такой задачи может быть описан на псевдокоде как показано на рис. 2.2. На языке ассемблера этот алгоритм может быть реализован следующим образом.

```
 1           READ
 2           STORE 100       ; в ячейке 100 храним максимум
 3    loop:  STORE 200       ; в ячейке 200 -- текущий элемент
 4           JZERO print     ; проверим, не кончилась ли последовательность
 5           SUB 100         ; разность текущего (в регистре) и максимума
 6           JLEZERO r
 7           LOAD 200        ; нашли новый максимум
 8           STORE 100
 9    r:     READ            ; считываем следующий элемент
10           JUMP loop
11    print: LOAD 100        ; загружаем и печатаем максимум
12           WRITE
13           HALT
```

Команды, расположенные в строчках 3–10 выполняются циклически до тех пор, пока на вход не поступит число 0. После выполнения команды в строке 3 справедливы следующие утверждения: значение последнего прочитанного элемента последовательности находится в ячейке 2 и в регистре, а ячейка 1 содержит максимальное значение той части последовательности, которая была обработана на текущий момент.

Косвенная адресация. Использование в программе только команд с указанием абсолютных адресов аргументов (например, LOAD 60 или ADD 14) ограничивает объём доступной программе памяти. Все адреса должны быть известны в момент написания программы и количество используемых ячеек не зависит от размера входных данных. Для преодоления такого ограничения

применяется метод *косвенной адресации* памяти.

В отличие от прямой адресации, когда аргументом команды является адрес ячейки памяти, из которой нужно загрузить или записать данные, при косвенной адресации аргументом команды является адрес ячейки, которая содержит *адрес* требуемой ячейки памяти. Для обозначения косвенной адресации в рассматриваемом языке ассемблера используется символ *, который ставится перед номером ячейки. Если значением ячейки с адресом 12 является число 82, то команда LOAD 12 (прямая адресация) загружает в регистр число 82, а команда LOAD *12 (косвенная адресация) загружает в регистр значение ячейки 82, так как в ячейке 12 записано число 82. Аналогично, команда ADD *12 прибавляет к текущему значению регистра содержимое ячейки 82.

Рассмотрим в качестве примера программу, которая проверяет, что заданная ячейка имеет указанное значение. Такая программа считывает два числа, — номер ячейки n и значение k, и выводит 1, если ячейка с адресом n имеет значение k.

```
 1           READ
 2           STORE 100       ; в ячейке 100 храним n
 3           READ
 4           STORE 200       ; в ячейке 200 храним число k

 5           LOAD *100       ; загружаем содержимое ячейки n
 6           SUB   200       ; сравниваем с k
 7           JZERO print     ; если в регистре 0, то в ячейке n записано k
 8           LOAD  =0        ; значение ячейки n не равно k
 9           WRITE
10           HALT            ; необходимо, так как иначе будет напечатано 01
11    print: LOAD  =1        ; печатаем положительный ответ
12           WRITE
```

Основная команда в данной программе – команда LOAD *100 в строке 5. После выполения команд 1 и 2 в ячейке с номером 100 будет записно значеие n. Команда LOAD *100 загружает в регистр содержимое ячейки с номером n. Заметим, что данную задачу невозможно решить без использования косвенной адресации, поскольку адрес ячейки, значение которой необходимо проверить, **неизвестен в момент написания программы**, а во всех командах должны использоваться числовые константы (в языке нет команды LOAD n).

2.3 Задачи и упражнения

В задачах на последовательности предполагается, что последовательность не содержит нуля и маркером конца последовательности является число 0.

1. Написать программу, которая проверяет, что заданная последовательность целых чисел является возрастающей.

2. Написать программу, которая проверяет, что последовательность a_1, $a_2, \ldots, a_n$ является палиндромом, то есть совпадает с $a_n, a_{n-1}, \ldots, a_1$.

3. Показать, что «двойная косвенная адресация», например, LOAD **10, может быть реализована в упрощенном ассемблере.

4. Написать программу, которая проверяет, что последовательность является арифметической прогрессией.

5. Написать программу, которая считывает число b и последовательность $a_0, a_1, \ldots, a_n$ и выводит значение полинома $a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ в точке $x = b$. Записывать последвательность в память не разрешается.

6. Написать программу, которая считывает число b и последовательность $a_0, a_1, \ldots, a_n$ и выводит значение полинома $a_n + a_{n-1} x + a_{n-2} x^2 + \ldots + a_0 x^n$ в точке $x = b$. Коэффициенты полинома вводтся в порядке *убывания* степеней. Записывать последвательность в память не разрешается.

7. Написать программу, которая считывает k и выводит k-ое число Фибоначчи f_k. Поледовательность чисел Фибоначчи задаётся правилами $f_1 = 1, f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$ $(n \geqslant 3)$.

Глава 3

Структуры данных и их представление в памяти ЭВМ

Многие программные системы предназначены для обраотки данных реального мира. Например, система учета книг в библиотеке должна работать с электронным каталогом библиотеки — множеством «карточек», которые описывают имеющиеся в библиотеке книги. Информация об отдельной книге может включать ее название, авторов, количество страниц. В данном разделе рассматриваются основные структуры данных, которые используются при решении практических задачах, и описываются методы представления этих структур в памяти (последовательности ячеек) вычислительной машины.

3.1 Запись

Модель памяти предполагает, что каждая ячейка содержит одно целое число. В некоторых случах требуется хранить в памяти элементы, которые на логическом уровне содержат несколько чисел. Например, информация о пациенте в санатории может содержать номер комнаты в которой он проживает, его возраст, рост и вес. Эти четыре числа образуют логически единую запись.

В реальных языках программирования одна ячейка может содержать значение определенного типа (целое число, действительное число, буква, дата и т.д.). Множество возможных типов фиксировано, но может отличаться в различных языках программирования[1]. Наличие различных типов данных удобно на практике, но теоретически все можно представить в виде числа. Например, все буквы можно занумеровать (буква a представляется числом 1, b – числом 2, и так далее).

[1]Например, языки, предназначенные для обработки финансовых данных, могут иметь специальный тип для денежных единиц

В общем случае *записью* (или *структурой*) называют объединение элементов различных типов. В приведенном выше примере рост, возраст и номер комнаты могут быть целыми числами, а вес — действительным числом. Составные части записи называются *полями*.

В языке программирования Си запись определяется с использованием ключевого слова struct. Например,

```
struct Patient {
  int age;
  int room;
  double weight;
} ivanov, petrov;
```

объявляет две переменные (ivanov и petrov) типа структура. Для доступа к отделным полям переменных типа структура используется символ точки. Например, условие if (ivanov.room == 12) позволяет проверить, что ivanov живет в комнате номер 12.

При хранении в памяти ЭВМ записи с несколькими полями используют фиксированный порядок полей. Например, поле age будет первым полем структуры Patient, room – вторым, а weight – третьим. Каждая запись занимает несколько подряд идущих ячеек, причем сначала хранится первое поле, потом второе, и так далее. Если известен адрес памяти K, где рассположен первый элемент записи, то для получения доступа к i-ому по порядку полю структуры достаточно вычислить *смещенеие* этого поля относительно начала записи. Если все поля занимают по одной ячейке памяти, то i-ое поле структуры находится по адресу $K + i$.

3.2 Линейные структуры

Линейной структурой данных называется совокупность однотипных элементов на которой определено отношение порядка (предыдущий/следующий), причём:

- для каждого элемента E существует не более чем один элемент, предшествующий E, и не более чем один элемент, следующий за E;
- существует единственный элемент для которого нет предшествующего (первый);
- существует единственный элемент у которого нет следующего (последний);
- каждый элемент достижим из первого.

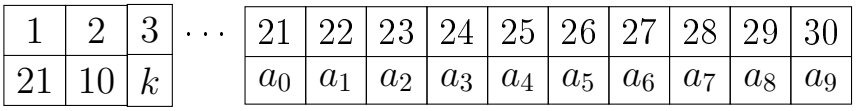


Рис. 3.1: Пример размещения массива в памяти. В ячейках 1 и 2 записаны адрес первого элемента и длина массива.

Если линейная структура содержит n элементов, то они могут быть занумерованы числами от 1 до n с сохранением отношения порядка, то есть элементы с номерами 1 и n являются первым и последним, а элементы с номерами $i-1$ и $i+1$ являются предыдущим и следующим для элемента с номером i (естественно, если они существуют, то есть $i-1 \geqslant 1$ и $i+1 \leqslant n$). Для представления линейной структуры в ячейках памяти могут использоваться либо массивы, либо списки.

3.2.1 Массивы

Массивом будем называть упорядоченный набор однотипных элементов, расположенных в подряд идущих ячейках памяти. Например, массив целых чисел из трех элементов занимает ячейки с адресами x,$x+1$,$x+2$. Если упорядоченный набор целых чисел занимает ячейки 10, 20 и 30, то такое представление не является массивом, так как между ячейками, содержащими элементы набора, есть ячейки, которые в него не входят. Массив однозначно определяется двумя значениями: адресом первого элемента (в какой ячейке размещен первый элемент линейной структуры) и числом элементов. Будем считать, что имя массива обозначает и адрес его первого элемента (такое соглашение используется, например, и в языке C++). Порядковый номер элемента массива называется *индексом элемента*. Для обозначения элемента массива A с индексом i будем использовать запись $A[i]$. Далее будем предполагать, что массивы индексируются начиная с нуля, то есть первый элементом массива A из n элементов является $A[0]$, а последним — $A[n-1]$.

Пусть первый элемент массива из десяти элементов расположен в ячейке с адресом 21 (рис. 3.1) и этот адрес записан в ячейке 1. Также допустим, что в ячейке с адресом 3 записано число k, причем $0 \leqslant k < 10$. Тогда элемент $A[k]$ находится в ячейке $21+k$. На языке ассемблера загрузка этого элемента в регистр может быть выполнена командами

```
LOAD 1   ; загрузили адрес первого элемента
ADD 3    ; прибавили k
STORE 4  ; готовимся к выполнению следующей команды
LOAD *4  ; загружаем содержимое ячейки 21+k
```

Очевидно, что время, необходимое для загрузки в регистр k-ого элемента массива, не зависит от значения k. Для этого необходимо выполнить приведенные выше четыре команды.

Второй важной особенностью массивов является отсутствие дополнительной памяти при хранении элементов. Если один элемент массива может быть записан в одну ячейку массива, то весь массив из n элементов будет занимать ровно n ячеек. В общем случае для хранения массива потребуется $n * s$ ячеек памяти, где s — это число ячеек необходимое для хранения одного элемента.

Если в массиве A нужно изменить значение элемента $A[k]$, то для этого нужно вычислить адрес $A + k$ (напомним, что имя массива — это адрес его первого элемента) и изменить значение ячейки с этим адресом. Если же требуется *вставить* новый элемент по указанному индексу, то есть от массива $A[0], \ldots, A[k-1], A[k], A[k+1], \ldots, A[n-1]$ перейти к массиву $A[0], \ldots, A[k-1], \mathbf{x}, A[k], A[k+1], \ldots, A[n-2]$, то необходимо *сдвинуть* все элементы с индексами большим или равным k на одну позицию. Такой алгоритм приведен на рис. 3.2. Очевидно, что при вставке нового элемента в

Вход: Массив A из n элементов, индекс k, новый элемент x.
Результат: Изменяется массив. Старое значение A[n-1] удаляется.

```
1 p ← n − 1;
2 while p > k do                    // Сдвинем элементы n-2, n-3,..., k
3 |  A[p] ← A[p-1];
4 end
5 Сейчас позиция A[k] «освободилась»;
6 A[k] ← x;
```

Рис. 3.2: Добавление элемента в массив по заданному индексу.

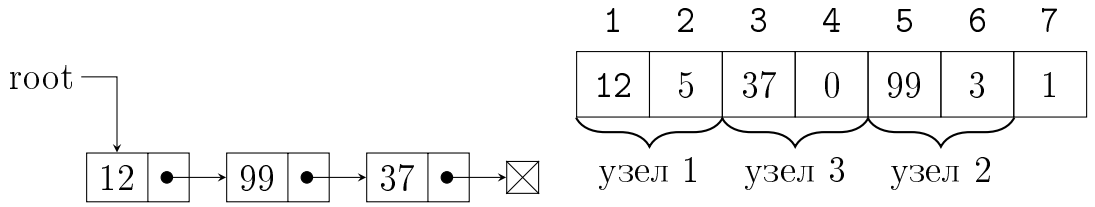
первую позицию нужно передвинуть все элементы массива, поэтому сложность данного алгоритма $O(n)$.

3.2.2 Списки

Вторым методом представления линейных структур в памяти ЭВМ являются *списки*. Списки ориентированы на быстрое выполнение операции вставки новых элементов.

Каждый элемент списка представляет собой запись (структуру) с двумя полями. Первое поле содержит «полезную информацию», а второе поле носит технический характер — это указатель на следующий элемент списка. Будем обозначать эти поля как *data* (данные) и *link* (связь). Пример списка из трех элементов приведен на рис. 3.3a. Этот список содержит три числа, причем первым числом является 12, вторым – 99 и третьим – 37.

Элементы списка занимают две ячейки памяти (если считать, что полезная информация всегда может быть записана в одну ячейку). Пример представления данного списка в памяти ЭВМ приведен на рисунке 3.3b. Здесь



(a) Логическое представление. (b) Представление в памяти.

Рис. 3.3: Пример списка из трех элементов.

первый элемент списка занимает ячейки с адресами 1 и 2. Второй элемент – ячейки с адресами 5 и 6, а третьий – ячейки с адресами 3 и 4. Содержимое ячеек 1 и 2 означает, что элемент списка содержит число 12, и что *следующий за ним* элемент списка размещен в ячейке 5 (и 6, так как каждый элемент занимает две ячейки). Ноль в ячейке с адресом 4 означает, что третьий элемент списка является последним.

В какой-то выделенной ячейке сохранен адрес первого элемента списка, так называемый *корень списка*. В нашем примере корень списка (root) записан в ячейке 7. Зная корень списка можно найти адреса все элементов списка, последовательно переходя по ссылкам. Если корень списка равен нулю (в нашем примере – в ячейке 7 записан 0), то список не содержит ни одного элемента. Такой список называется *пустым*.

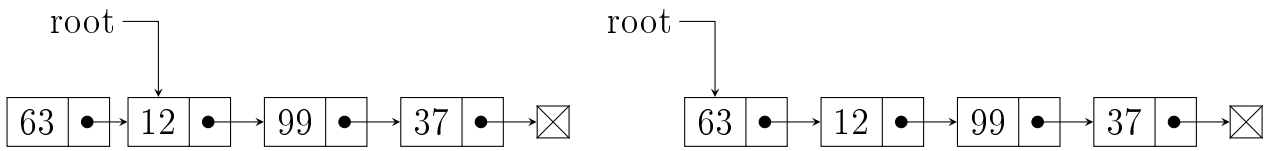
Рассмотрим теперь алгоритм вставки нового элемента в начало списка (рис. 3.4). Список задается своим корнем. Добавляемый элемент задан адресом node своей первой ячейки. Поскольку корень списка указывает на его

Вход: Корень списка root, адрес нового элемента node
Результат: Список изменяется таким образом, что его первым элементом становится node

```
1 node.link ← root;
2 root ← node;
```

Рис. 3.4: Добавление элемента в начало списка.

первый элемент первая строчка алгоритма по сути означает следующее: следующим после элемента node должен быть первый элемент списка. Вторая строчка делает элемент node новым первым элементом. Схематично действие этого алгоритма приведено на рис. 3.5. Видно, что после выполнения шага 1 (рис. 3.5a) список все еще содержит три элемента, так как новый элемент node недостижим из корня. Адрес, по которому расположен новый элемент, вообще говоря может быть любым, то есть в памяти первый элемент может быть размещен после второго (рис. 3.3b).



(a) После выполнения строки 1. (b) После выполнения строки 2.

Рис. 3.5: Пример добавления элемента в список по алгоритму 3.4.

Добавление нового элемента в конец списка требует последовательного просмотра всех его элементов, так как в отличие от массива, зная корень списка невозможно сказать по какому адресу будет расположен его последний элемент. Алгоритм добавления последнего элемента приведен на рис. 3.6.

Вход: Корень списка root, новый элемент node
Результат: Список изменяется таким образом, что его последним элементом становится node

```
1 elem ← root;
2 while elem.link ≠ ∅ do                    // Найдем последний элемент
3 |   elem ← elem.link;
4 end
5 Сейчас elem указывает на последний элемент;
6 elem.link ← node;
7 node.link ← ∅;
```

Рис. 3.6: Добавление элемента в конец списка.

Просмотр списка требуется и для нахождения k-ого элемента. В отличие от массива, когда по индексу элемента можно вычислить адрес памяти, по которому записан этот элемент, элементы списка могут быть засположены в памяти хаотично.

Рассмотрим в качестве примера программу на языке ассемблера, которая находит в заданном списке число x. Псевдокод данного алгорита приведен на рис. 3.7. Будем считать, что корень списка записан в ячейке с адресом 1, а число x, которое необходимо найти, находится в ячейке 2.

```
  LOAD 1        ; загружаем корень списка
  STORE 3       ; в ячейке 3 находится указатель на текущий элемент elem
next_item:
  LOAD *3       ; в регистр попадает поле data текущего элемента
  SUB 2         ; вычислим разность elem.data-x
  JZERO found   ; нашли то, что искали?
  ; на нашли...
  ; загрузим адрес следующего элемента списка - он записан в ячейке,
  ; адрес которой не единицу больше содержимого ячейки 3
```

Вход: Корень списка *root*, значение x
Результат: Возвращает адрес элемента, значение которого равно x

```
1 elem ← root;
2 while elem ≠ ∅ do                    // Просмотрим элементы по очереди
3 |   if elem.data = x then
4 |   |   return elem;
5 |   end
6 |   elem ← elem.link; // Перейдем к следующему элементу
7 end
8 return ∅; // Просмотрели все элементы
```

Рис. 3.7: Поиск элемента списка по значению.

```
  LOAD 3
  ADD =1
  STORE 3
  LOAD *3       ; сейчас в регистре значение elem.next
  STORE 3       ; перешли на следующий элемент списка
  JGTZERO next_item  ; если в регистре не 0, то список не закончился
  ;; Список закончился, но мы не встретили элемент со значением x
  ;; В ячейке 3 сейчас находится 0 (адрес текущего элемента)
found:
  LOAD 3        ; если поиск удачный, то выводим адрес найденного элемента
  WRITE         ; если значение x не нашли, то выводим 0
  HALT
```

Двусвязный список. Списки, которые были определены выше, позволяют переходить от текущего элемента к следующему. Переход к предыдущему элементу возможен только в результате выполнения поиска, сложность которого линейно зависит от числа элементов. *Двусвязый список* (или *двунаправленный список*) — это список, каждый элемент которого хранит ссылку на своего левого и правого соседа. Пример двусвязного списка приведен на рис. 3.8.

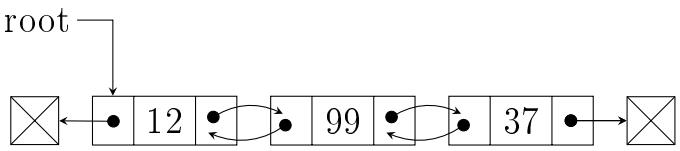


Рис. 3.8: Пример двусвязного списка из трех элементов.

Добавление элементов в двусвязный список прозводится путем изменения ссылок соседних элементов. Алгоритм добавления нового элемента после заданного элемента приведен на рис. 3.9. Предполагается, что каждый элемент

списка содержит поля next и prev, которые хранят ссылки на правого и левого соседа, соответственно.

Вход: Текущий элемент списка $current \neq \varnothing$, новый узел $node \neq \varnothing$.
Результат: Узел *node* добавлятеся в список после *current*.

```
1 node.prev ← current;
2 node.next ← current.next;
3 current.next ← node;
4 if node.next ≠ ∅ then
5 |   node.next.prev ← node;
6 end
```

Рис. 3.9: Вставка элемента в двусвязный список.

Строчка 4 алгоритма 3.9 проверяет, что в исходном списке был по крайней мере один элемент после *current* (в момент проверки этого условия node.next содержит старое значение current.next). Если current был последним элементом, то значение node.next.prev не определено, так как адрес node.next равен нулю.

3.3 Очередь, стек и дек

Очередью называют линейную структуру у которой добавление новых элементов производится в конец очереди, а извлечение производится из начала. Такие очереди называют FIFO, от английского First in First Out (первым пришел, первым ушел). Элементами очереди могут быть, вообще говоря, произвольные объекты: числа, адреса, структуры и т.д. Для простоты изложения далее будем предполагать, что в очередь помещаются целые числа.

Интерфейс очереди может включать следующие функции:

- создать очередь с заданным максимальным размером;
- добавить новый элемент в очередь;
- проверить, что очередь пуста (не содержит ни одного элемента);
- проверить, что очередь полна;
- извлечь первый элемент;
- удалить очередь.

Реализация очереди может использовать массив или список в качестве структуры данных лля хранения элементов очереди.

Реализация очереди на основе массива. При данном методе хранения элементов очередь определяется следующими параметрами:

- максимальным размером очереди N;
- текущим числом элементов в очереди m;
- адресом T первого элемента массива, содержащего элементы очереди.

Тройка N,m,T может быть объединена в одну структуру Queue[2] и вся очередь будет представляться адресом, где эта структура расположена в памяти. Зная один адрес X (фактически, одно целое число) можно получить всю информацию об очереди: по адресу X записано число N (первое поле структуры, описывающей очередь), по адресу X+1 — m, а по адресу X+2 — адрес первого элемента массива с элементами очереди.

Выделение элементов очереди в отдельный массив позволяет унифицировать описание очереди. Размер структуры Queue не зависит от максимального размера очереди, что позволяет, например, сделать массив из очередей. У разных очередей могут быть массивы различной длины, но в самой структуре Queue хранится только адрес начала массива.

Добавление и удаление элементов из очереди требует изменения данных в массиве. Алгоритм извлечения первого элемента из очереди приведен на рис. 3.10. Соответствующая реализация на упрощенном ассемблере приведен на рис. 3.11 на странице 31.

Вход: Очередь X: структура с полями N, m и T.
Результат: Значение первого элемента, если очередь не пуста, или -1.

```
1 if X.m = 0 then
2 |   return -1;
3 end
4 first ← X.T[X.m];
5 сдвинуть_массив(X.T, X.m);
6 X.m ← X.m-1;
7 return first;
```

Рис. 3.10: Извлечение первого элемента очереди.

Реализация очереди на основе кольцевого списка. Реализация стека на основе массива.

[2] В переводе с английского Queue означает «очередь».

```
  ;; Пусть адрес X, по которому записана струкутра очереди, находится в ячейке 1
  ;; То есть по адресу X, X+1, X+2 записаны N, m и T, соответственно
  ;; Для удобства запишем адреса N, m и T в ячейки 2, 3 и 4
  LOAD 1        ; загружаем X
  STORE 2       ; В 2 находится адрес N
  ADD =1
  STORE 3       ; В 3 находится адрес m
  ADD =1
  STORE 4
  LOAD *4
  STORE 4       ; В 4 находится T - адрес первого элемента очереди

  ;; Сохраним первый элемент очереди во временной ячейке
  LOAD *4
  STORE 5

  ;; Сдвинем все элементы массива T на одну ячейку вверх ("удалим нулевой")
  LOAD =0
  STORE 6       ; счетчик цикла i; Для всех 0 <= i < m-1 будем делать T[i]=T[i+1]
next_item:
  LOAD *3
  SUB =1
  SUB 6         ; m-1-i
  JLEZERO end_shift    ; переходим, если m-1 <= i

  ;; Копируем один элемент массива
  LOAD 4
  ADD 6
  STORE 7       ; получили адрес T[i]
  ADD =1
  STORE 8       ; получили адрес T[i+1]
  LOAD *8
  STORE *7      ; выполнили копирование T[i]=T[i+1]

  LOAD 6
  ADD =1
  STORE 6       ; увеличили i на единицу
  JUMP next_item

end_shift:
  LOAD *3
  SUB =1
  STORE *3      ; m=m-1 (очередь стала на один элемент короче)

  LOAD 5        ; значение извлеченного элемента
  WRITE
  HALT
```

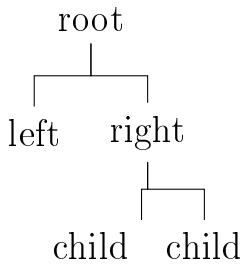
Рис. 3.11: Программа извлечения первого элемента очереди на упрощенном ассемблере.

3.4 Деревья

Деревом называется набор $T = \langle N, E \rangle$, где N — множество вершин дерева, а $E \subseteq N \times N$ — множество ребер, который удовлетворяет следующим условиям:

- каждая вершина имеет не более одного элемента p, который ему предшествует (его родитель);
- существует ровно один элемент, который называется *корнем* дерева, у которого нет предшественника.

Элементы дерева, которые не имеют дочерних элементов, называются *листовыми*. Если элемент p является родителем элемента c, то c называется *дочерним элементом* для p (буквы p и c обозначают parent и child).



Отметим, что список является деревом с одним листовым элементом.

Дерево называется *бинарным* (или двоичным), если для любого его элемента существует не более двух дочерних элементов, причем правый и левый дочерний элемент различаются (если у элемента есть только один дочерний элемент, то он либо правый, либо левый).

Глава 4

Алгоритмы

4.1 Алгоритмы поиска

4.1.1 Последовательный поиск

Вход: Массив $A[0], \ldots, A[n-1]$, key – ключ поиска
Результат: idx – индекс массива, такой что $A[idx] = key$ или -1

```
1 i ← 0;
2 while i < n do
3 |   if A[i] = key then
4 |   |   return i;
5 |   end
6 |   i ← i + 1;
7 end
8 return -1;
```

Рис. 4.1: Последовательный поиск в массиве.

Простейший алгоритм поиска элемента с заданным значением ключа состоит в последовательном просмотре всех элементов массива (или списка) до тех пор, пока не будет найден искомый элемент. Такой алгоритм приведен на рис. 4.1.

Если входной массив содержит n элементов, то для выполнения поиска потребуется $O(n)$ операций (нужный нам элемент может оказаться последним).

4.1.2 Поиск в упорядоченном массиве

Если известно, что входной массив упорядочен по возрастанию, то есть $A[i] \leqslant A[i+1]$ для всех $i = 1, \ldots, n-2$, то время поиска можно существенно сократить.

Бинарный поиск (рис 4.2).

Вход: Упорядоченный массив $A[0], \ldots, A[n-1]$, $A[i] \leqslant A[i+1]$ для всех $0 < i < n-1$; K – значение
Результат: индекс idx массива, такой что $A[idx] = K$, или -1

```
1  l ← 0, r ← n − 1; // Левая и правая граница поиска
   // Значение K всегда лежит между A[l] и A[r]
2  while l ⩽ r do          // Пока не все элементы массива проверены
3  |  mid ← ⌊(l + r)/2⌋; // Целочисленное деление
4  |  if A[mid] = K then                    // Нашли нужное значение
5  |  |  return mid;
6  |  end
7  |  if A[mid] > K then
8  |  |  r ← mid − 1;
9  |  else
10 |  |  l ← mid + 1;
11 |  end
12 end
13 return −1
```

Рис. 4.2: Двоичный поиск.

Поскольку каждая итерация цикла, который начинается в строке 2 алгоритма 4.2, сокращает область поиска в два раза (либо левая, либо права граница поиска передвигается в середину), то время поиска заданного значения составляет $O(\log_2 n)$.

4.1.3 Таблицы расстановки

Самым быстрым методом поиска заданного значения в массиве является метод *таблиц расстановки* (хеш-таблиц, hash-table).

Пусть задана функция $h : \mathbb{N} \to \{0, \ldots, n-1\}$, которая произвольное число отображает в элемент конечного множества. Будем называть такую функцию *функцией расстановки* или *хеш-функцией*. Простейшим примером хеш-функции является функция, которая вычисляет остаток от деления на n, то есть $h(x) = x \bmod n$.

Хеш-таблица представляет собой массив T с заданной функцией расстановки h. Размер массива должен быть не меньше, чем множество возможных значений функции h.

При добавлении нового элемента x в хеш-таблицу значение x присваивается элементу массива $T[h(x)]$, то есть функция расстановки определяет позицию в массиве T, в которую нужно записать число x. Если элемент массива с индексом $h(x)$ занят, то необходимо определить другую позицию для

записи элемента x. Совпадение значений хеш-функции для различных значений аргумента называется *коллизией*. Процедура выбора новой позиции для записи значения x в случае возникновения коллизии называется *разрешением коллизий*. Будем считать, что существует специальное значение, например -1, которое означает, что данная ячейка массива пуста[1].

Вход: Массив T из n элементов, хеш-функция h, новое значение x
Результат: Изменяется массив T, возвращает индекс добавленного элемента

```
1  k ← h(x); // Определим индекс для x
2  i ← k;
3  repeat                                  // Пока не все ячейки проверили
4  |  if T[i] = −1 then
5  |  |  T[i] ← x;
6  |  |  return i;
7  |  end
8  |  i ← i + 1 mod n;
9  until i ≠ k;
10 return −1; // Таблица полностью заполнена
```

Рис. 4.3: Метод линейных проб. Вставка элемента.

Метод линейных проб. Вставка нового элемента x в хеш-таблицу при разрешении коллизий методом линейных проб производится следующим образом. Если ячейка $T[h(x)]$ пуста, то x вставляется «на свое законное место» $h(x)$. Если же там уже расположен другой элемент, то x вставляется в первую свободную ячейку, которая циклически находится справа от $h(x)$ (то есть ячейка с индексом 0 является следующей после ячейки $n-1$, так как в массиве из n элементов индекс n не существует). В алгоритме 4.3 такой циклический порядок реализован в строке 8: переменная i увеличивается на единицу по модулю n.

Поиск элемента со значением x в таблице осуществляется аналогично. Проверяется ячейка $h(x)$ и все последующие, вплоть до первой свободной (рис. 4.4).

Если из хеш-таблицы нужно удалить значение x, то для этого может потребоваться перемещение некоторых других элементов. Это необходимо для

[1]Это означает, что в нашей хеш-таблице не может храниться значение -1. Если решаемая задача не позволяет выбрать такого «пустого» значения (все числа допустимы), то для обозначения пустой ячейки может использоваться какой-то другой метод кодирования. Например, элементами хеш-таблицы будут не числа, а структуры с полями isEmpty («свободно?») и value. Или информация о занятости ячеек может хранится в отдельном битовом множестве.

Вход: Массив T из n элементов, хеш-функция h, значение x
Результат: Индекс элемента, значение которого равно x, или -1

```
1  k ← h(x); // Определим возможную позицию x
2  next ← k;
3  repeat                    // Проверяем все ячейки до первой пустой
4  |  i ← next;
5  |  if T[i] = x then
6  |  |  return i;
7  |  end
8  |  next ← i + 1 mod n;
9  until T[i] ≠ −1 и next ≠ k;
10 return −1; // Встретили пустую или просмотрели все элементы
```

Рис. 4.4: Метод линейных проб. Поиск элемента.

того, чтобы алгоритм поиска смог найти такие элементы y, которые занимают позиции, отличные от $h(y)$. Алгоритм удаления приведен на рис. 4.5.

Вход: Массив T из n элементов, хеш-функция h, значение x
Результат: Изменение массива T

```
1  empty ← поиск(T, h, x); // Определим индекс x (алгоритм 4.4)
2  T[empty] ← −1; // empty всегда указывает на пустую ячейку
3  i ← empty + 1 mod n;
4  while T[i] ≠ −1 do  // Обрабатываем один блок заполненных ячеек
   |  // Проверим, что элемент T[i] можно перенести в empty
   |  // Условие: h(T[i]) находится не между empty и i
   |     (циклически)
5  |  if не (empty < h(T[i]) ⩽ i или h(T[i]) ⩽ i < empty или
   |  i < empty < h(T[i])) then
6  |  |  T[empty] ← T[i];
7  |  |  empty ← i;
8  |  end
9  |  i ← i + 1 mod n;
10 end
```

Рис. 4.5: Метод линейных проб. Удаление элемента.

Условие в строке 5 алгоритма 4.5 рассматривает три случая. Первая часть соответствует ситуации, когда отрезок от позиции *empty* и до текущего элемента i непрерывен. Во втором и третьем случае указатель i уже перешел через значение n, а *empty* еще нет. Элементы, которые находятся между *empty* и i разделены на две части. Во втором условии $h(T[i])$ попадает в

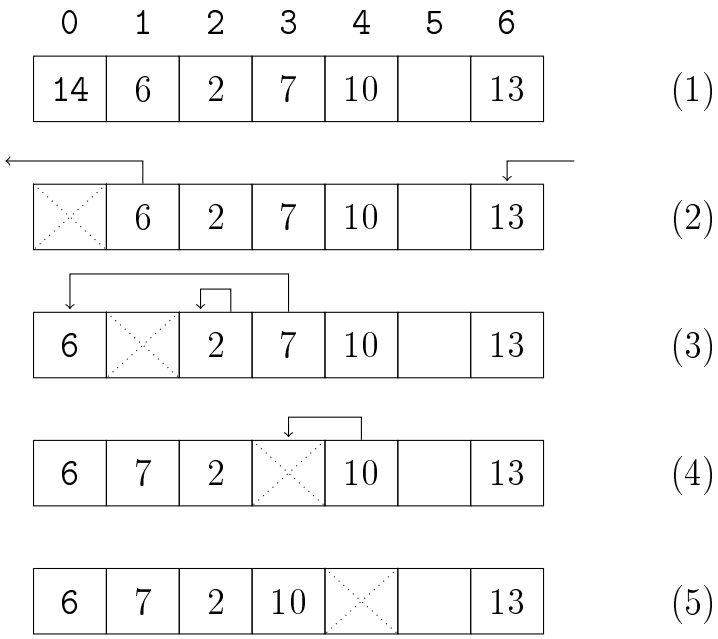


Рис. 4.6: Пример удаления из хеш-таблицы значения 14, $h(x) = x \bmod 7$.

левую часть, а в третьем – в правую.

Рассмотрим работу этого алгоритма на следующем примере. Пусть в хеш-таблицу, построенную с использованием функции $h(x) = x \bmod 7$, добавляются числа $14, 2, 13, 6, 7, 10$. В результате получится таблица, изображенная в первой строке на рис. 4.6. При удалении значения 14 ячейка с индексом 0 освобождается. На рисунке освободившаяся ячейка перечеркнута (строка 2). При этом нарушается достижимость значения 6, так как $h(6) = 6$ (это отражено стрелкой) и алгоритм поиска не сможет найти значение 6 в случае, если ячейка 0 останется пустой. Поэтому значение 6 необходимо перенести в только что освободившуюся позицию, а свободной становится ячейка с индексом 1. Поскольку $h(2) = 2$, то это число остается неподвижным. В ячейку 1 должно быть скопировано значение 7. При этом новая свободная ячейка появляется по индексу 3. Далее, поскольку $h(10) = 3$, необходимо передвинуть значение 10.

Оценка ожидаемого числа сравнений при поиске по методу линейных проб. При поиске заданного значения в хеш-таблице алгоритм 4.4 производит последовательные сравнения ключа поиска с элементами таблицы. Чем меньше производится сравнений, тем быстрее будет работать поиск. Очевидно, что при полностью заполненной таблице и поиске ключа, которого нет в хеш-таблице, будет произведено n сравнений. Если в таблице находится всего один элемент, то потребуется одно сравнение. Оценим ожидаемое число сравнений при условии, что известен коэффициент заполненности таблицы m/n, где m — число ненулевых элементов таблицы.

Будем предполагать, что хеш-функция h равномерно распределяет эле-

менты по множеству значений $\{0, \ldots, n-1\}$. Пусть коэффициент заполненности равен $m/n = \lambda < 1$ и требуется найти значение key. Наихудший с точки зрения числа сравнений случай возникает, когда key не встречается в хеш-таблице. При неудачном поиске каждая последовательность проверок закончится пустой ячейкой. Пусть X обозначает число сравнений, которое потребуется алгоритму поиска при отсутствии ключа. Вероятность того, что $T[h(key)] \neq -1$ есть λ. Если уже было проверено k подряд идущих ячеек и все они оказались заполнены, то вероятность того, что и $(k+1)$-я ячейка будет заполнена есть $\frac{m-k}{n-k}$. (Поскольку k ячеек были проверены и все они оказались заполнены, то всего осталось $m-k$ заполненных ячеек. Хеширование предполагается равномерным, значит эти $m-k$ заполненных ячеек равномерно распределены по оставшимся непроверенным $n-k$ ячейкам хеш-таблицы, что и дает нужную оценку.) Поскольку $m < n$, то $\frac{m-k}{n-k} \leqslant \frac{m}{n}$. Вероятность того, что при поиске потребуется сделать более i сравнений есть

$$P\{X \geqslant i\} = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \frac{m-(i-2)}{n-(i-2)} \leqslant \left(\frac{m}{n}\right)^{i-1} = \lambda^{i-1}.$$

Ожидаемое значение $M[X]$ есть

$$M[X] = \sum_{i=1}^{\infty} P\{X \geqslant i\} \leqslant \sum_{i=1}^{\infty} \lambda^{i-1} = \sum_{i=0}^{\infty} \lambda^i = \frac{1}{1-\lambda}.$$

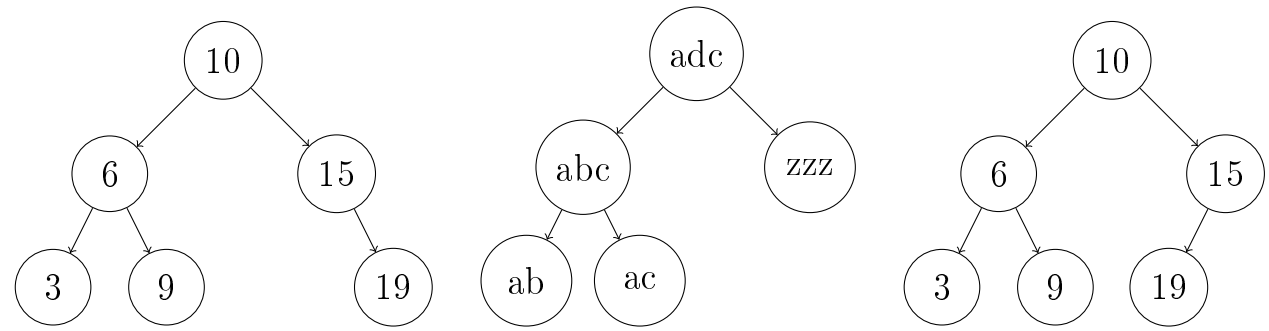
Таким образом справедлива следующая теорема.

Теорема 2. *Математическое ожидание числа сравнений при неудачном поиске в хеш-таблице при равномерном хешировании и коэффициенте заполненности таблицы $\lambda < 1$ не превышает $\frac{1}{1-\lambda}$.*

4.1.4 Деревья поиска

Деревом поиска называется помеченное бинарное дерево, которое удовлетворяет следующему условию. Для любого узла n значения узлов в его левом поддереве меньше значения n, а значения узлов в правом поддереве больше, чем значение в n. Значениями узлов дерева могут быть любые объекты для которых определена операция сравнения. Это могут быть числа с обычным отношением $<$, или строки, сравнение которых производится в соответствии с *лексикографическим порядком*[2], который обычно используется при составлении словарей.

[2]Строка A длины n *лексикографически меньше* строки B длины m тогда и только тогда, когда либо найдется такой индекс $k < \min(n, m)$, что $A[k] < B[k]$ и для любого индекса $k' < k$ выполняется $A[k'] = B[k']$, либо такого k не существует и $n < m$. Например, "ab" < "acd" (k=1), "ab" < "abc" (первая строка совпадает с началом второй). Элементы строк, то есть буквы $A[k]$, сравниваются по значению их ASCII кодов, что для обычных символов соответствует порядку их появления в латинском алфавите.



(a) Дерево поиска, вершины которого помечены числами.

(b) Дерево поиска, вершины которого помечены строками.

(c) Бинарное дерево, которое **не является** деревом поиска.

Рис. 4.7: Примеры деревьев поиска.

Рассмотрим в качестве примера деревья, изображенные на рис. 4.7. На первом рисунке (рис. 4.7a) изображено дерево поиска, узлы которого содержат целые числа. Можно проверить, что для любого узла выполняются условия определения дерева поиска. Отметим, что узел со значением 15 не имеет левого дочернего элемента. Дерево на рис. 4.7b также является деревом поиска, если его узлы сравниваются в лексикографическом порядке. На рис. 4.7c приведен пример дерева, которое не удовлетворяет условию определения: левое поддерево узла со значением 15 содержит узел с большим значением.

Далее будем считать, что каждая вершина дерева описывается структурой с полями val, left и right. Поле val содержит значение, которое приписано этому узлу дерева, а поля left и right являются ссылками на корневой узел левого и правого поддерева, соответственно. Иногда бывает удобно добавлять в структуру узла дерева ссылку на родительский узел parent. Это позволяет легко подниматься по дереву от листьев к корню, но немного усложняет процедуру добавления и изменения узлов, так как значения поля parent должны быть согласованы с другими полями. В частности, должно выполняться равенство $node.left.parent = node.right.parent = node$ для любого узла, у которого есть оба поддерева. Если у узла $node$ изменяется значение $node.left$, то соответствующее изменение поля $parent$ нужно произвести и в узле $node.left$. Мы не будем включать поле $parent$ в описание узла дерева, а в тех случаях, когда это необходимо, ссылка на родительский узел будет вычисляться.

Поиск. Деревья поиска предназначены для поиска элементов по значению. Предположим, что у нас есть дерево поиска и в нем требуется найти узел с заданным значением x. Сначала нужно проверить значение в корне и, если оно не равно x, то перейти к левому или правому поддереву. Если x больше значения в корне, то поиск следует продолжать в правом поддереве, так как все элементы в левом поддереве меньше корня. Если же x меньше, чем значение в корне, то поиск следует продолжить в левом поддереве.

```
   Вход: Корневой узел root дерева поиска T, значение x
   Результат: Адрес узла дерева, значение которого равно x, или ∅
 1 current ← root; // Начинаем поиск с корня
 2 while current ≠ ∅ do              // Пока текущий узел существует
 3 |   if current.val = x then
 4 |   |   return current; // Нашли узел с нужным значением.
 5 |   end
 6 |   if x < current.val then
 7 |   |   current ← current.left; // Спускаемся в левое поддерево
 8 |   else
 9 |   |   current ← current.right; // Спускаемся в правое поддерево
10 |   end
11 end
12 return ∅; // Поиск закончился неудачей
```

Рис. 4.8: Поиск по значению узла в дереве поиска.

Текст алгоритма поиска в дереве поиска приведен на рис. 4.8. Рассмотрим работу этого алгоритма в случае, когда дерево поиска соответсвует рис. 4.7a и требуется найти значение $x = 19$. Сначала *current* указывает на корень дерева. Так как значение в корне меньше x, то *current* изменяет свое знаение в строке 9 и указывает на вершину со значением 15. Далее происходит сравнение значения x и 15, что приводит к очередному изменению переменной *current* в строке 9. Поскольку теперь *current* указывает на узел с нужным значеним, то алгорит заканчивает свою работу в строке 4. Заметим, что если бы дерево поиска соответсвовало бы рис. 4.7c, то алгоритм не смог бы найти нужный узел.

Если дерево поиска содержит N узлов, то сложность поиска составляет $O(N)$. Примером наихудшего случая является дерево с одим листом. Каждый узел такого дерева (кроме листа) имеет только один дочерний узел. Если искомое значение x совпадает со значениемм в листовом узле такого дерева, то поиск сводится к полседовательному поиску в линейном списке из N элементов.

Специальным случаем является поиск минимального (или максимального) значения в дереве. Алгоритм поиска минимуа приведен на рис. 4.9.

Добавление элементов. Теперь рассмотрим алгоритм добавления элементов в дерево поиска. Добавление в некотором смысле аналогично алгоритму поиска — для нового значения требуется найти такую позицию в дереве, чтобы оно не нарушало определение дерева поиска. Алгоритм приведен на рис. 4.10. Здесь предполагается, что новый элемент добавляется в дерево, которое содержит по крайней мере один узел — корень дерева. Если же де-

Вход: Корневой узел $root \neq \varnothing$ дерева поиска T
Результат: Адрес корня узла, имеющего минимальное значение

```
1 current ← root;
2 while current.left ≠ ∅ do
3 |   current ← current.left;
4 end
5 return current
```

Рис. 4.9: Поиск минимального значения.

рево пусто, то добавление элемента сводится к созданию корневого элемента (строки 14–16 алгоритма 4.10).

Вход: Корневой узел $root \neq \varnothing$ дерева поиска T, значение x
Результат: Адрес узла дерева, значение которого равно x

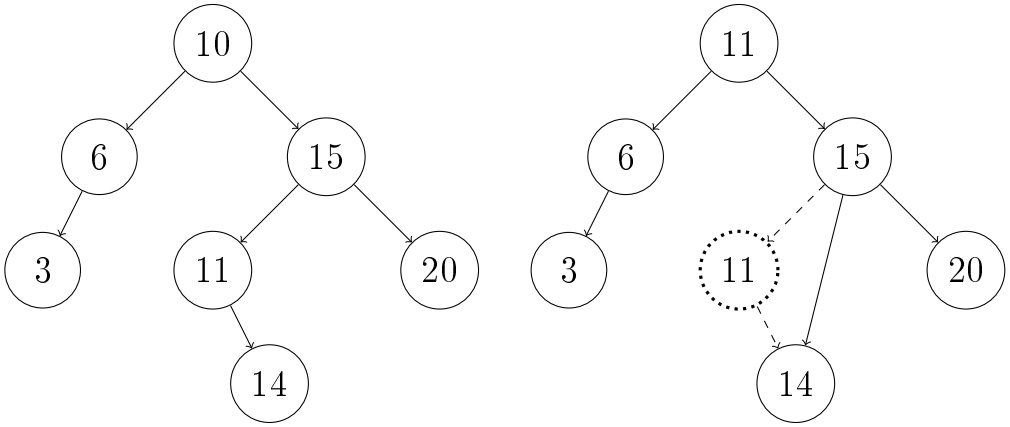
```
1  current ← root; // Начинаем поиск с корня
2  parent ← ∅; // Родитель current. У корня его нет.
3  repeat
4  |   parent ← current;
5  |   if current.val = x then
6  |   |   return current; // Значение x уже есть в дереве.
7  |   end
8  |   if x < current.val then
9  |   |   current ← current.left; // Спускаемся в левое поддерево
10 |   else
11 |   |   current ← current.right; // Спускаемся в правое поддерево
12 |   end
13 until current ≠ ∅;
14 node ← выделитьПамять(3); // Создаем узел со значением x
15 node.val ← x;
16 node.left ← ∅, node.right ← ∅;
17 if x < parent.val then
18 |   parent.left ← node;
19 else
20 |   parent.right ← node;
21 end
22 return node
```

Рис. 4.10: Добавление в непустое дерево поиска.

Первая часть алгоритма практически совпадает с алгоритмом 4.8 поиска в дереве. Отличие состоит в добавлении строки 4, которая гарантирует,

что переменная *parent* всегда указывает на родительский узел для *current*. Если в процессе поиска места для добавления нового элемента выясняется, что узел с таким значением уже существует, то алгоритм добавления просто возвращает этот узел без изменения дерева. Если же цикл закончился, то новый узел необходимо сделать дочерним по отношению к узлу *parent*: алгоритм поиска перешел от *parent* к одному из его поддеревьев, но оно оказалось пустым (переменная *current* стала равна $\varnothing$).

Удаление элементов. Пусть требуется удалить элемент со значением x. При удалении узла из дерева поиска необходимо сохранить свойство, что для любого узла значения в его левом поддереве меньше, а значения в правом поддереве больше значения в узле. Рассмотрим в качестве примера дерево на рис.4.11a и предположим, что требуется удалить значение $x = 10$. Одним из возможных способов является «сдвиг» какой-либо цепочки узлов вверх. Например, значение 15 копируется в 10, 11 — в 15, а значение 14 — в узел, который в исходном дереве содержал значение 11. После такого сдвига листовой узел 14 можно удалить. Такое изменение дерева сохраняет свойство дерева поиска, но трубует изменения значительного числа узлов. Желательно, чтобы при удалении элементов сущетвующие узлы дерева по возможности не модифицировались. Для этого можно скопировать в удаляемый элемент значение минимального элемента из его правого поддерева. Реально из дерева будет удален именно этот узел. Схематично эта операция изображена на рис. 4.11b. Если в позицию удаляемого узла (в нашем примере это корень дерева) копируется минимальный элемент из его правого поддерева, то получается дерево поиска.



(a) Дерево поиска до удаления узла со значением 10.

(b) После удаления корня. Отмеченные пунктиром ребра и узлы удалены.

Рис. 4.11: Удаление элемента из дерева.

Алгоритм удаления значения из дерева поиска приведен на рис. 4.12. Первая часть (строки с 1 по 6) находит $minR$ — минимальный элемент в правом поддереве корня, и устанавливает *parrent* на родительский узел для $minR$.

В дереве на рис. 4.11a переменная $minR$ будет указывать узлу со значением 11, а $parent$ — на узел со значением 15. Далее производится изменение ссылок в соответсвии с рис. 4.11b. Существует два особых случая. Во-первых, у корня может не быть правого поддерева. В этом случае из дерева удаляется корень, а новым корнем становится его левый дочерний элемент. Во-вторых, у $root.right$ может не быть левого поддерева. Тогда новым правым дочерним узлом корня будет $minR.right$. В нашем примере в дереве это соответствует ситуации, когда в дереве нет элементов 11 и 14. Удалит нужно узел 15 (в корень будет скопировано значение 15, а не 11), а новое ребро провести от корня к узлу 20.

Вход: Корневой узел $root \neq \varnothing$ дерева поиска T
Результат: Адрес корня дерева или $\varnothing$

```
1  minR ← root.right; // Этот узел будет удален
2  parent ← root; // Родитель minR
3  while minR ≠ ∅ и minR.left ≠ ∅ do
4  |   parent ← minR;
5  |   minR ← minR.left;
6  end
7  if minR = ∅ then                    // У root нет правого поддерева
8  |   newRoot ← root.left;
9  |   освободитьПамять(root);
10 |   return newRoot
11 end
12 root.val ← minR.val;
13 if parent = root then          // У root.right нет левого поддерева
14 |   root.right ← mirR.right;
15 else
16 |   parent.left ← minR.right;
17 end
18 освободитьПамять(minR);
19 return root
```

Рис. 4.12: Удаление корня дерева поиска.

4.1.5 Сбалансированные деревья

АВЛ-дерево это дерево поиска для которого выполняется дополнительное условие: **для любого узла дерева** глубина его левого поддерева отличается от глубины правого поддерева не более чем на единицу.

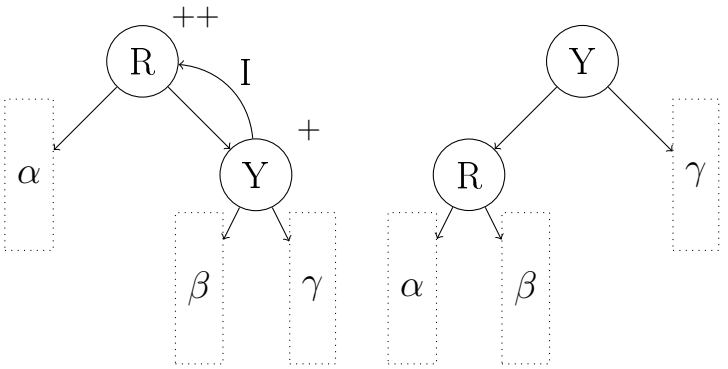
Основная идея применения сбалансированных деревьев состоит в том, что они обспечивают гарантированно высокую скорость работы алгоритмов по-

иска, добавления и удаления элементов. Если обычное дерево поиска может в вырожденном случае представлять собой список, то АВЛ-дерево имеет «примерно равные длины ветвей». Для АВЛ-деревьев известна верхняя оценка высоты.

Теорема 3. *Высота АВЛ-дерева, содержащего n листовых узлов, не превосходит* $1.5 \log_2 n$.

Добавление элемента в сбалансированное дерево может потребовать выполнения дополнительных действий по сравнению с алгоритмом 4.10. Вершину сбалансированного дерева удобно описывать пятеркой значений: value, left, right, parent, balance. Первые четыре поля соответствуют описанию узла бинарного дерева, а значения balance принадлежат множеству $\{-1, 0, 1\}$ и соответствуют разности высоты правого и левого поддеревьев данного узла (значение -1 означает, что левое поддерево на единицу выше правого).

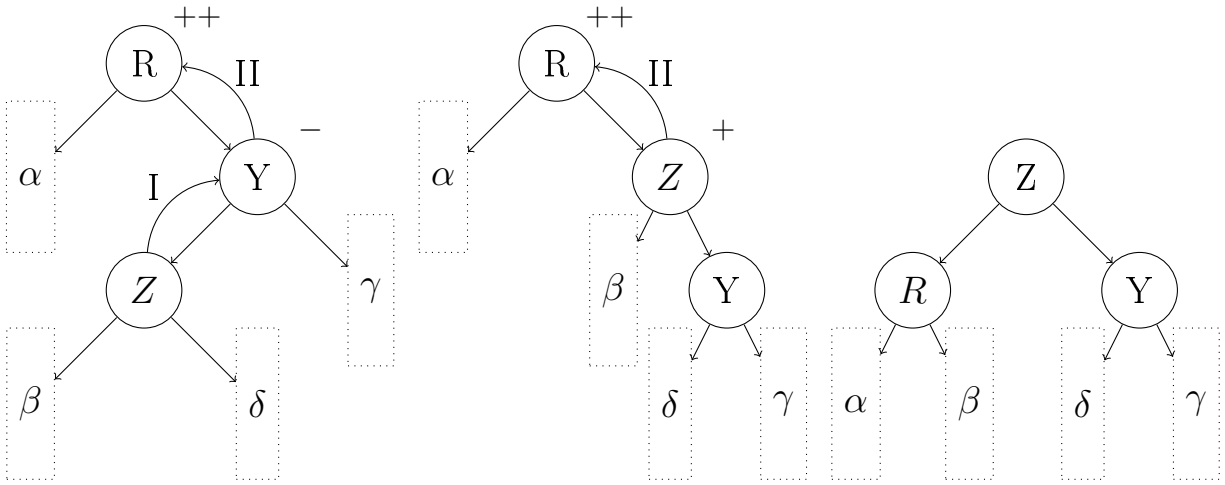
Добавление нового элемента в АВЛ-дерево начинается с добавления элемента как в обычное дерево поиска. Если после добавления нового узла в соответствующую позицию в дереве появляется узел, баланс которого больше единицы, то дерево становится несбалансированным и производится один или два *поворота*. Схематично эти повороты представлены на рис. 4.13 и рис. 4.14. Существует еще два поворота, которые симметричны представленным на этих рисунках. Повороты осуществляются в узле, баланс которого нарушается и который ближе всего расположен к вновь созданному узлу, то есть расположен ниже всего в цепочке от корня до нового узла. Позицию узла для поворота можно вычислить в процессе спуска по дереву при поиске позиции для вставки нового узла.



(a) До поворота.

(b) После поворота Y в корень.

Рис. 4.13: Одинарный поворот в корне АВЛ-дерева.



(a) До поворота. (b) После первого поворота. (c) После второго поворота.

Рис. 4.14: Двойной поворот при добавлении в АВЛ-дерево.

4.1.6 Сложность операций

		поиск		
	вставка	по значению	по индексу	удаление
массив	$O(n)$	$O(n)$	$O(1)$	$O(n)$
$\leqslant$-массив	$O(n)$	$O(\log_2 n)$	$O(1)$	$O(n)$
список	$O(1)$	$O(n)$	$O(n)$	$O(1)$
хеш-таблица[3]	$O(1)$	$O(1)$	-	$O(1)$
дерево поиска	$O(n)$	$O(n)$	-	$O(n)$
АВЛ-дерево	$O(\log_2 n)$	$O(\log_2 n)$	-	$O(\log_2 n)$

[3]Для хеш-таблицы сложность не зависит от n, но зависит от степени заполнения таблицы (см. Теорему 2). В худшем случае, когда таблица заполнена полностью, для каждой из задач может потребоваться $O(n)$ операций.

4.2 Сортировка

В задаче сортировки предполагается, что есть *массив записей* (структур), который нужно упорядочить в порядке возрастания значений *ключа* сортировки. Например, если одна запись описывает библиотечную карточку книги, то она может включать такие поля как авторы, название, год издания, количество страниц и другие. Описание содержимого библиотеки может представляться массивом таких записей. Если требуется вывести все книги в алфавитном порядке по названию, то мы должны решить задачу сортировки массива по ключу *название*. В общем случае для корректной постановки задачи сортировки должны быть определены правила получения по записи ее ключа и сравнения этих ключей. Для упрощения описания алгоритмов мы будем предполагать, что сортируются числовые массивы относительно обычной функции сравнения $\leqslant$, то есть ключем поиска является само значение элемента массива.

Основные универсальные алгоритмы сортировки используют сравнение ключей. Существует несколько классов алгоритмов сортировки. *Обменные сортировки* переставляют элементы в массиве в соответствии с результатом сравнения ключей до тех пор, пока в массиве не останется элементов, нарушающих порядок. *Сортировки выбором* выбирают из исходного массива элементы в требуемом порядке изменения ключей (например, в порядке возрастания). *Сортировки вставками*, напротив, по очереди добавляют элементы в отсортированный массив. Существуют специализированные алгоритмы сортировки, которые разработаны при условии, что исходный массив удовлетворяет некоторым ограничениям. Одним из таких примеров является сортировка подсчетом, которая основана на предположении, что исходный массив содержит малое число различных значений ключей. Такие ограничения могут возникать из требований прикладных задач.

Далее в этом разделе рассматриваются наиболее известные методы сортировки и оценивается сложность их работы.

4.2.1 Сортировка вставками

Предположим, что k начальных элементов массива A из n элементов (то есть $A[0], A[1], \ldots, A[k-1]$) уже были отсортированы и нам нужно добавить еще один элемент со значением X. Для этого потребуется найти такой индекс j, что $A[j-1] \leqslant X \leqslant A[j]$, где $0 < j < k$. Новое значение нужно добавить в позицию j, но для того, чтобы освободить ячейку с индексом j, нужно сдвинуть элементы массива, начиная с индекса j, вправо на одну позицию. Если добавляемый элемент находится в позиции k, то поиск позиции для вставки можно совместить со сдвигом элементов массива A. Это приводит к алгоритму, представленному на рис. 4.15.

Внешний цикл выполняется n раз, для всех значений $ready$ в диапазоне

```
   Вход: Массив A из n элементов
   Результат: Изменение массива A
 1 ready ← 0; // Элементы от A[0] до A[ready] отсортированы
 2 while ready < n − 1 do                  // Не все элементы обработаны
 3 |   k ← ready + 1; // Поставим A[k] на нужное место
 4 |   while k > 0 и A[k] < A[k − 1] do
   |   |   // Меняем местами A[k] и A[k − 1]
 5 |   |   temp ← A[k];
 6 |   |   A[k] ← A[k − 1];
 7 |   |   A[k − 1] ← temp;
 8 |   |   k ← k − 1;
 9 |   end
10 |   ready ← ready + 1;
11 end
```

Рис. 4.15: Сортировка простыми вставками.

от 0 до $n-1$. При каждом значении $ready$ внутренний цикл выполняется не более $ready+1$ раз: в худшем случае добавляемый элемент окажется меньше, чем все предыдущие. В среднем на каждой итерации внешнего цикла будет выполняться $n/2$ действий, что дает общую оценку сложности данного алгоритма $O(n^2)$.

4.2.2 Сортировка пузырьком

Сортировка пузырьком является одним из самых простых и медленных способов обменной сортировки. Некоторые авторы даже считают, что этот алгоритм должен быть исключен из учебников, так как существуют гораздо более эффективные алгоритмы. Этот алгоритм, текст которого приведен на рис. 4.16, сортирует массив в несколько проходов. На каждом проходе элементы массива просматриваются справа налево и все пары соседних элементов, которые нарушают порядок сортировки, обмениваются местами. Ясно, что после одного прохода минимальное значение массива окажется по нулевому индексу. На следующем проходе процедура повторяется, только теперь массив рассматривается начиная не с нулевого элемента, а с позиции последнего обмена. Алгоритм заканчивается, в процессе очередного прохода не было выполнено ни одного обмена.

4.2.3 Сортировка кучей

Кучей (или *пирамидой*) называется массив A из n элементов, удовлетворяющий следующим условиям:

```
   Вход: Массив A из n элементов
   Результат: Изменение массива A
 1 processed ← 0; // Количество отсортированных элементов
 2 repeat
 3 |   k ← N − 1;
 4 |   swapAt ← −1; // Позиция последнего обмена
 5 |   while k > processed do
 6 |   |   if A[k] < A[k − 1] then
 7 |   |   |   Поменять местами A[k] и A[k − 1];
 8 |   |   |   swapAt ← k − 1;
 9 |   |   end
10 |   |   k ← k − 1;
11 |   end
12 |   processed ← swapAt + 1;
13 until swapAt ⩾ 0;
```

Рис. 4.16: Сортировка пузырьком.

1. для любого индекса $0 \leqslant k < n$ такого, что $2(k+1)-1 < n$ значение $A[k] \geqslant A[2(k+1)-1]$;

2. для любого индекса $0 \leqslant k < n$ такого, что $2(k+1) < n$ значение $A[k] \geqslant A[2(k+1)]$.

4.2.4 Быстрая сортировка

Алгоритм быстрой сортировки является одним из распространенных вариантов сортировки. Именно этот алгоритм реализован в функции qsort стандартной библиотеки языка Си. Текст этого алгоритма приведен на рис. 4.20.

4.2.5 Нижняя оценка сложности сортировки на основе сравнения ключей

Теорема 4. *Не существует алгоритма сортировки на основе вравнения ключей, который на всех входных массивах требует менее, чем* $O(n \log_2 n)$ *операций.*

Доказательство. Рассмотрим *дерево решений*, которое представляет собой бинаное дерево, внутренние узлы которого определяют операцию сравнение двух элементов исходного массива, а листовые — указывают перестановку элементов, которая делает массив упорядоченным. Пример такого дерева для

Вход: Массив A из n элементов
Результат: Изменение порядка эементов массива A; заполнение массива B

```
// Первый этап: построить кучу по массиву A
k ← 1; // Массив из одного элемента является кучей
while k < n do
    Добавить(A, k); // Выполнить алгоритм 4.18 добавления элемента к куче
end
// Сейчас массив A является кучей
//
// Второй этап: вывести результат
k ← 0;
while k < n do
    C[k] ← A[0];
    Извлечь(A, n − k); // Выполнить алгоритм 4.19 удаления корня
end
```

Рис. 4.17: Алгоритм сортировки кучей.

Вход: Массив A, число m. Первые m элементов A образуют кучей
Результат: Изменение массива A: первые $m + 1$ элементов образуют кучу

```
k ← m; // Элементы A[0], ..., A[m − 1] образуют кучу. Добавим A[k].
while k > 0 do                                  // Пока не дошли до корня
    // Вычислим индекс родителя k
    parent ← ⌊(k + 1)/2⌋ − 1; // Целочисленное деление
    if A[parent] < A[k] then                    // A[k] нарушает условие кучи
        // Меняем местами A[k] и A[parent] -- поднимаем A[k] на один уровень
        temp ← A[k];
        A[k] ← A[parent];
        A[parent] ← temp;
        k ← parent; // Сейчас новый элемент в позиции parent
    else
        return ; // Новый элемент занял нужную позицию
    end
end
```

Рис. 4.18: Алгоритм добавления элемента к куче.

Вход: Массив A из n элементов, элементы которого являются кучей
Результат: Изменение массива A: корень удаляется

```
1  k ← 0; // k указывает на пустой узел
2  repeat                                  // Пока не дошли до листа дерева
       // Вычислим индексы дочерних для k элементов
3      left ← 2(k + 1) − 1;
4      right ← 2(k + 1);
       // Найдем индекс максимума двух чисел A[left], A[right]
5      max ← left;
6      if right < n и A[left] < A[right] then      // right < n ⇒ left < n
7          max ← right;
8      end
9      if max < n then                          // Дочерний элемент существует
10         A[k] ← A[max];
11     end
12     k ← max;
13 until k < n;
```

Рис. 4.19: Алгоритм удаления корня кучи.

массива из трех элементов изображен на рис. 4.21. Для каждого значения n алгоритму сортировки соответствует некоторое дерево решений. Для разных алгоритмов деревья решений могут отличаться. Высота дерева решений определяет сложность алгоритма: в худшем случае для сортировки массива из n элементов алгоритму потребуется выполнить все сравнения на самой длинной ветке дерева.

Оценим высоту дерева решений. Это дерево содержит $n!$ листовых узлов, так как любая перестановка элементов может быть результатом сортировки. Высота h идеально сбалансированного бинарного дерева с $n!$ листами, то есть дерева, у которого все его листья находиться на одном уровне, равна $\lfloor \log_2 n! \rfloor + 1$. Это означает, что высота произвольного дерева решений для сортировки массива из n элементов не может быть меньше $\log_2 n!$. Теперь заметим, что $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \geqslant \frac{n}{2}^{\frac{n}{2}}$. В произведении $n!$ есть $n/2$ сомножителей, которые не меньше, чем $n/2$. Это означает, что высота дерева удовлетворяет неравенству $h \geqslant \log_2 n! \geqslant \frac{n}{2} \log_2 \frac{n}{2} = O(n \log_2 n)$. □

Следствием этой теоремы является, в частности, оптимальность алгоритма пирамидальной сортировки. На практике следует учитывать, что теоретическая оценка сложности алгоритмов может проявляться только при достаточно больших значениях n. Например, при сортировке «коротких» массивов, содержащих не более десяти элементов, алгоритм сортировки пузырьком может оказаться более эффективным, чем пирамидальная сортировка или сортировка слиянием в силу более простой реализации.

Вход: Массив A, начальный индекс $left$, конечный индекс $right$
Результат: Элементы $A[left], A[left+1], \ldots, A[right]$ отсортированы

```
1  if right − left ⩽ 1 then                          // Это короткий массив
2  |  if A[left] > A[right] then
3  |  |  Поменять местами A[left] и A[right]
4  |  end
5  |  return
6  end
7  α ← (A[left] + A[right])/2; // Выберем сечение α
   // Переставим элементы так, что сначала идут значения ⩽ α, а
      потом > α
8  i ← left;
9  j ← right;
10 while i < j do
11 |  while A[i] ⩽ α u i < j do   // Найдем первый слева элемент > α
12 |  |  i ← i + 1;
13 |  end
14 |  while A[j] > α u i < j do  // Найдем первый справа элемент ⩽ α
15 |  |  j ← j − 1;
16 |  end
17 |  if i < j then
18 |  |  Меняем местами A[i] и A[j].
19 |  end
20 end
21 border ← j; // Место разделения массива на две части
22 Быстрая сортировка(A, left, border); // Сортируем левую часть
23 Быстрая сортировка(A, border+1, right); // Сортируем правую часть
```

Рис. 4.20: Алгоритм быстрой сортировки.

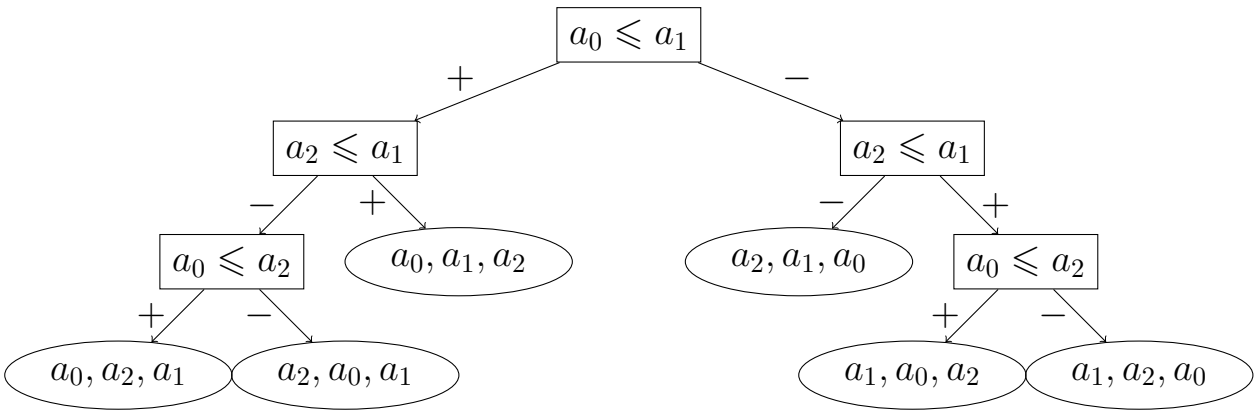


Рис. 4.21: Пример дерева решений для сортировки массива из трех элементов.

4.3 Сортировка за линейное время

Теорема 4 утверждает, что не может быть алгоритмов сортировки, работающих быстрее, чем за $O(n \log_2 n)$. Но это верно для произвольных массивов. Если эе про исходные данные известна некоторая дополнительная информация, то можно реализовать алгоритмы сортировки, работающие за линейне время. Очевидно, что более быстрых алгоритмов быть не может, так как только для просмотра исходного массива требуется время $O(n)$.

Сортировка подсчетом. Если известно, что исходный массив содержит не более K различных элементов, и K мало, то для его сортировки достаточно посчитать количество вхождений каждого значения. После этого можно заполнить отсортированный массив (нужное число раз вывести минимальное значение, потом — следующее по порядку, и так далее).

Глава 5

Алгоритмические языки

5.1 Классификация языков программирования

Низкоурвневые или высокоуровневые
- Универсальные или специализированные
- Статическая или динамическая типизация
- Компилируемые или интерпретируемые
- Декларативные или императивные
- Процедурные, функциональные, логические
- С автоматическим или ручным управлением памятью

5.2 Язык Лисп

Язык программирования Лисп является компилируемым функциональным языком программирования с автоматическим управлением памятью и динамической типизацией с автоматическим выводом типов. В данном разделе описываются основные конструкции языка. Изложение не претендует на полноту.

5.2.1 S-выражения

Как данные, так и сами программы на языке Лисп представляются в виде S-выражений. Определение S-выражения имеет рекурсивный характер и основывается на понятии атома. Атомы – это либо числовые или строковые константы, либо символы. Например, `12`, `imja-peremennoj`, `"это строка"` являются атомами. Специальной формой атома можно считать символ, который начинается с двоеточия, например `:test`. Такие атомы называются ключевыми словами.

S-выражением является либо атом, либо последовательность S-выражений, заключенная в круглые скобки. Например, `51`, `(cos 2.0)` или `(search (list 2) (list 1 2 2) :from-end t)` являются S-выражениями.

Каждое S-выражение вычисляется в некоторое значение[1]. Значением константы является сама константа. С символами могут быть связаны некоторые значения (значения переменных). Если S-выражение имеет вид `(f a b c)`, то символ `f` рассматривается как имя функции, а последующие элементы – как аргументы функции. Значением является результат применения функции к её аргументам. Некоторые S-выражения, которые называются специальными формами, нарушают данное правило. Специальные формы — это элементы языка. Наиболее часто используемые специальные формы включают:

Объявления	**Условные операторы**	**Циклы**	**Переменные**	**Прочее**
defun	and	do	let	progn
defmacro	or	do*	let*	prog1
defvar	if	dolist	setf	quote
defparameter	case	dotimes	incf	function
labels	cond	loop	decf	
	when			

5.2.2 Основные конструкции языка

Управление порядком выполнения if, when, cond, case

Локальные переменные.

Группировка операторов.

Циклические конструкции. В языке Лисп имеется несколько вариантов определения повторяющихся действий: dotimes, dolist, do, do*, loop. Do-циклы позволяют выполнять тело цикла, которое является неявным progn, до тех пор, пока не выполниться условие выхода из цикла. Цикл loop обладает более гибким и более сложным языком описания циклических конструкций и в данном разделе не рассматривается.

Самым простым циклом является **цикл dotimes**. Он позволяет ввести переменную цикла, которая последовательно будет принимать значения от 0 до заданного максимального числа, и выполнить тело цикла, которое является неявным progn. Например, следующий цикл выводит значения от 0 до 9 включительно.

```
1 (dotimes (k 10)
2   (print k))
```

[1] Существует единственное исключение — форма `(values)` не возвращает никакого значения.

Значение всего выражения dotimes будет NIL. Если после верхней границы числа итераций указана еще одна форма, то ее значение будет вычислено по окончании цикла. Например, цикл `(dotimes (k 10 (* 2 k)))`, который имеет пустое тело, вернет значение 20, так как форма `(* 2 k)` будет вычислена после последней итерации, когда k примет значение 10.

Цикл dolist аналогичен dotimes, но вместо верхней границы числа итераций указывается список значений. Переменная цикла последовательно указывает на элементы этого списка. Например, в выражении

```
(dolist (item '(1 (2 3) 4))
  (print item))
```

переменная цикла item будет последовательно принимать значения 1, '(2 3) и 3. Тело цикла будет выполнено три раза и на второй итерации значением item будет список из двух элементов.

Последним циклом из семейства do-циклов является **цикл do**. Описание цикла состоит из списка переменных цикла, условия выхода и возвращаемого значения. Каждая переменная описывается именем, начальным значением и действием, которое выполняется для получения следующего значения переменной. Например, в цикле

```
(do ((k 0 (1+ p))
     (p 0 k))
    ((< 4 k) (+ k p))
  (format t "k=~D, p=~D~%" k p))
```

вводятся две переменные, k и p, которые принимают начальные значения 0. После окончания каждой итерации текущие значения переменных используются при параллельном вычислении форм `(1+ p)` и `k`. Таким образом, на второй итерации k будет равно значению 1, а p – нулю (значению переменной k на предыдущей итерации). Выход из цикла происходит при выполнении условия `(< 4 k)`. Значением формы do будет результат вычисления `(+ k p)`.

5.2.3 Объявление функций

5.2.4 Работа со списками

Предметный указатель