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 *аааззббб* 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).

Алгоритм 1.1: Выполнение нормального алгоритма

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

Выход: Выходное слово

```
repeat
    i ← 0;                    // Номер последнего применённого правила
    Найдём первое применимое правило;
    k ← 1;
    while k ⩽ n и i = 0 do
        if Правило u_k → v_k применимо к w then
            i ← k;
            w ← Применить(w, u_i → v_i);
        end if
        k ← k+1;
    end while
until i > 0 и i ∉ F;
return w;
```

№ правила	исходное слово	результат применения
2	$ba\underline{bba}a$	$ba\underline{=}a$
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 $A(w)$.

1.2.2 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.

1.3 Сложность вычислений

Временно́й сложностью алгоритма называется количество шагов, которое необходимо для завершения работы алгоритма на входных данных. *Пространственная сложность* (или *ёмкостная сложность*) алгоритма — это максимальный объем памяти, который требуется алгоритму в процессе его работы. В терминах алгоритмов Маркова временна́я сложность соответствует числу правил, которые были применены, а пространственная — 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 время работы алгоритма на входных данных размера $2n$ будет в четыре раза больше, чем при обработке входноого слова длины n. 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.

Сложность задачи можно оценивать аналогично алгоритмам. Поскольку сложность задачи связана с рассмотрением бесконечного множества алгоритмов, то рассматривают верхние и нижние оценки. *Верхняя оценка* определяет равенство $C(\mathcal{A}) = O(f(n))$ (задача имеет сложность $O(f(n))$). Обычно это означает, что известен алгоритм решения задачи, который имеет сложность $O(f(n))$. *Нижняя оценка* сложности утверждает, что не существует алгоритма решения задачи, который имеет меньшую сложность. При этом алгоритм, который имеет такую оценку может быть неизвестен. Если же такой алгоритм существует, то говорят, что нижняя оценка является *точной* и задача имеет сложность $\Omega(n^2)$, то есть $C_1 n^2 \leqslant T_n(A) \leqslant C_2 n^2$ для некоторых констант C_1 и C_2.

1.4 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.

Неразрешимость какой-либо массовой задачи как правило доказывается путем ее сведения к другой задаче, про которую уже известно, что она является неразрешимой. В этом разделе мы докажем алгоритмическую неразрешимость проблемы определения свойства самоприменимости алгоритма (определение будет дано ниже), и покажем, как на основе этого результата можно доказать неразрешимость проблемы остановки алгоритма.

1.4.1 Неразрешимость проблемы определения самоприменимости

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

[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).

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. □

1.4.2 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).

Теорема 2. *Не существует алгоритма, который для заданного алгоритма A и слова w проверяет, что значение $A(w)$ определено.*

Доказательство. Предположим, что такой алгоритм существует. Обозначим его C. Пусть A_u — код алгоритма A в терминах универсального алгоритма Маркова. По предположению, алгоритм C проверяет определённость A для любого входного слова. Тогда мы можем вычислить $C(A, A_u)$. Это означает, что мы можем проверить *самоприменимость* алгоритма A. Но это противоречит Теореме 1. Значит, алгоритма C не существует. □

1.4.3 Другие неразрешимые задачи

В предыдущем разделе мы доказали неразрешимость одной задачи путем её сведения к другой задаче. Ниже приводятся некоторые другие неразрешимые задачи, которые часто используются в доказательствах неразрешимости (вместо задачи определения самоприменимости).

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).

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. (см. Упражнение 1.14)

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
$u_1 u_1 u_2 = aab\ aab\ b = a\ a\ baabb = v_1 v_1 v_2$.

1.5 Другие модели вычислений

Алгоритмы Маркова являются не единственной и, возможно, не самой известной формализацией понятия алгоритма. В этом разделе кратко рассматриваются другие модели.

1.5.1 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 соответству-

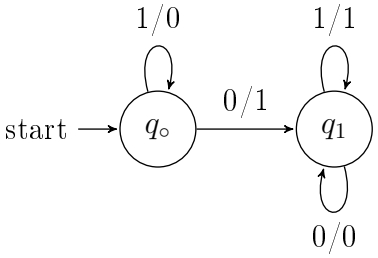


Рис. 1.1: Пример конечного автомата, который прибавляет единицу к двоичному представлению числа, записанному в обратном порядке.

ющее 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 1.1 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 $w = a_1 a_2 \ldots a_k$ 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 1.1 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.

1.6 Задачи и упражнения

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

1.2 Составьте нормальный алгоритм Маркова, который входное слово $x_1x_2\ldots x_n$ в алфавите $\Sigma = \{a, b\}$ преобразует в слово x_1x_n, то есть удаляет из слова все буквы, кроме первой и последней.

1.3 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.

1.4 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, $ababba \mapsto babaab$.

1.5 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. $aaabaaaaaaaabaa \mapsto aaababaa$.

1.6 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, $ababba \mapsto aabbaabbbbaa$.

1.7 Составьте нормальный алгоритм Маркова, который любое слово $w = x_1x_2\ldots x_n$ а алфавите $\Sigma = \{a, b\}$ переводит в слово $x_nx_1x_2\ldots x_{n-1}$. Например, $ababb \mapsto babab$.

1.8* Пусть задан алгоритм Маркова A, который содержит пять правил и обрабатывает слова в алфавите Σ. Если для некоторого входного слова $w \in \Sigma^*$ значение $A(w)$ определено, то слову w можно сопоставить номер $k_A(w) \in \{1, 2, 3, 4, 5\}$ последнего примененного правила алгоритма A. Покажите, как по алгоритму A можно составить алгоритм A', который вычисляет $k_A(w)$.

1.9 Пусть задан алгоритм Маркова A, который «разворачивает» слова в алфавите Σ, то есть преобразует любое слово $w = x_1x_2\ldots x_n$ к виду $x_nx_{n-1}\ldots x_1$. Докажите, что система правил этого алгоритма обязательно использует вспомогательные символы, которые не входят валфавит Σ.

1.10 Докажите, что не существует алгоритма, который для заданного алгоритма Маркова $\mathcal{A}$ и слова w определяет, что алгоритм $\mathcal{A}$ определен на слове w.

1.11 Составьте нормальный алгоритм Маркова, который для слова w в $\Sigma = \{a, b\}$ вычисляет его длину в двоичной записи. Пример: $abbaba \mapsto 110$.

1.12 Пусть $f : \mathbb{N} \to \mathbb{N}$. Рассмотрим множество $\mathcal{A}_f$ алгоритмов Маркова в алфавите Σ, у которых размер промежуточных слов ограничен величиной $f(n)$, где n — длина входного слова. Верно ли, что существует алгоритм, который для любой пары $(A, w) \in \mathcal{A}_f \times \Sigma^*$ проверяет, что $A(w)$ определено.

1.13 Составьте нормальный алгоритм Маркова, который слова вида $a^n \# a^m$ переводит в $a^{n \bmod m}$ (остаток от деления).

1.14 Докажите, что не существует алгоритма, который для заданного алгоритма Маркова A и номера правила проверяет, что существует входное слово w, такое что при вычислении $A(w)$ будет применено правило с этим номером.

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

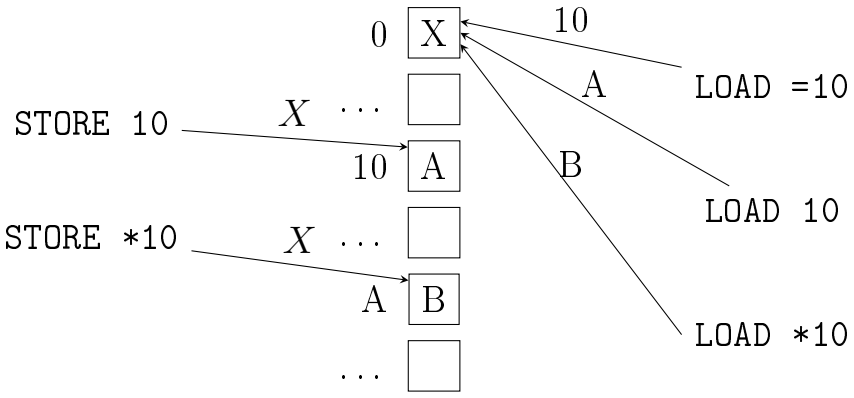


Рис. 2.1: Действие команд LOAD и STORE для различных типов аргументов.

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,

Алгоритм 2.1: Нахождение максимального элемента

Вход: Элементы последовательности считываются из входного потока

Выход: 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 if
7 |  n ← Прочитать;
8 end while
9 Напечатать max;
```

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. 2.1. 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 записано 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 Вызов подпрограмм*

При решении практических задач некоторые действия приходится неоднократно выполнять для разных входных данных. Например, очень часто тре-

буется расположить элементы некоторой последовательности в порядке возрастания. Подобные задачи мы будем подробно обсуждать в главе 4. Сейчас рассмотрим достаточно простой алгоритм решения этой задачи. Предположим для определенности, что адрес начала последовательности A записан в ячейке 1, а число элементов последовательности n — в ячейке 2. Если мы найдем максимальный элемент последовательности и поменяем его с последним элементом, то для решения всей задачи нам понадобится выполнить аналогичную процедуру еще раз, но только уже для последовательности из $n-1$ элемента. Действительно, сначала по адресу $A+n-1$ (это адрес, где расположен последний элемент последовательности из n элементов, которая начинается с адреса A) будет записано самое большое значение. Потом по адресу $A+n-2$ будет записано самое большое значение из оставшихся. И так далее. Псевдокод этого алгоритма приведен на рис. 4.13 (стр. 69).

Ясно, что такой алгоритм может быть реализован на ассемблере (попробуйте сделать это в качестве упражнения!). Можно ли сделать его универсальным в том смысле, что если нам нужно отсортировать несколько последовательностей, то мы не должны копировать текст программы несколько раз, заменяя при этом адреса некоторых ячеек или команды? Части программы, которые можно использовать в программе несколько раз без копирования кода, называются *подпрограммами*, процедурами или функциями.

Для создания подпрограммы нужно решить три проблемы. Во-первых, нужно передавать в подпрограмму входные данные. В нашем примере такими данными будет адрес A и число n. Во-вторых, подпрограмма должна иметь возможность вернуть какие-то данные. В нашем примере подпрограмма сортировки ничего не возвращает, а функция вычисления максимального значения должна возвращать одно число. Наконец, нужно знать, какую команду следует выполнять после окончания работы подпрограммы.

Первые два вопроса могут быть решены *соглашением* об адресах входных и выходных параметров. Например, наша подпрограмма может всегда считывать входные данные из ячеек с адресами 100 и 101. Проблема возврата из подпрограммы немного сложнее. В нашем упрощённом ассемблере она не может быть решена, так как единственная возможность изменить порядок выполнения команд — это выполнение оператора условного или безусловного перехода, а для этого нужно указать метку перехода. Рассмотрим возможную реализацию подпрограммы.

```
1           LOAD =500
2           STORE 100       ; адрес A
3           LOAD =10
4           STORE 101       ; n
5           JUMP sort_subprogram ; вызов подпрограммы
6   return_point:
7           ...             ; следующие команды
```

```
...          ...

11   sort_subprogram:
12           ...            ; реализация алгоритма сортировки
13           JUMP  return_point
```

В строках 1–4 производится запись входных параметров по тем адресам, которые «знает» подпрограмма. В строке 5 происходит передача управления в подпрограмму. Содержательная часть сортировки опущена (это же упражнение!). Предположим, что мы успешно переставили элементы в соответствии с алгоритмом 4.13. После окончания подпрограммы нужно вернуть управление основной программе, то есть продолжить её выполнение начиная со строки 7. Если пометить эту строку меткой (в нашем примере — return_point), то подпрограмма всегда будет возвращаться к строке 7, независимо от того, из какой части программы она была «вызвана».

Поэтому в реальных процессорах есть специальные регистры, содержащие адрес следующей команды (напомним, что сама программа хранится в оперативной памяти как последовательность машинных кодов) и адрес специальной области памяти, так называемый *стек вызовов*, которая используется для хранения значения этого регистра перед вызовом подпрограммы. Назовем первый регистр CA (command address), а второй — SP (stack pointer). Начальным значением SP является какое-то большое число, а CA — адрес, где в памяти хранится первая команда программы. При выполнении каждой команды процессор автоматически изменяет значение CA таким образом, что он указывает на следующую команду. Если выполняется команда перехода (JUMP, JZERO или аналогичная), то в CA записывается адрес команды, помеченной соответствующей меткой.

Для вызова подпрограмм используются команды типа `CALL метка` и `RETURN`, которые делают следующее. Команда CALL записывает по адресу SP значение регистра CA, уменьшает значение SP и записывает в CA адрес, соответствующий метке `метка`. Команда `RETURN` сначала увеличивает значение SP, а потом загружает в регистр CA содержимое ячейки по адресу SP. Можно заметить, что такая схема позволяет вызывать из одной подпрограммы другую подпрограмму. Точки возврата (адреса, где находятся команды `CALL`) будут записаны один за другим.

В реальности через стек (память, на которую указывает SP) передаются и входные параметры вызова подпрограммы. Перед вызовом `CALL` в стек добавляют значения входных параметров (уменьшая значение SP после записи каждого значения), а в начале подпрограммы эти данные «извлекают» из стека. Подпрограмма знает сколько аргументов должно быть передано и может прибавить к текущему значению SP нужное смещение. Например, при вызове нашей подпрограммы сортировки в стек может попасть три значения: адрес возврата (значение регистра CA перед вызовом), адрес начала последовательности A и число элементов n. Передача параметров через стек

позволяет избежать ситуации, когда две подпрограммы требуют передачи параметров через одну ячейку (выше мы говорил, что наша подпрограмма считывает значения ячеек 100 и 101; эти же адреса могут использоваться и в другой подпрограмме).

2.4 Задачи и упражнения

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

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

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

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

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

2.5 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.

2.6 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.

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

2.8 Пусть в ячейках с адресами $A, A{+}1, \ldots, A{+}n{-}1$ записаны числа из диапазона $[0, 10]$, причем n значительно больше 10. Адрес A и n записаны в ячейках 1 и 2, соответственно. Напишите программу на ассемблере, которая подсчитывает количество вхождений чисел $[0, 10]$.

2.9* Решите задачу 2.8 за один проход по последовательности A.

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.

Существуют *явные* и *неявные* структуры данных. В явных структурах используются явные указатели на физическое расположение частей структуры в памяти. Примером явных структур данных являются список и бинарное дерево. В неявных структурах связь между частями определяются за счет вычисления арифметических операций. Примером неявной структуры является куча.

При выборе или разработке конкретной струкутры данных необходимо учитывать возможный сценарий работы с этими даннми. Для представления одних и тех же данных можно использовать различные структуры, которые будут отличаться по скорости выполнения операций и объему памяти, необходимой для хранения данных. Как правило, компактное представление данных приводит к более медленным алгоритмам их обраьотки.

3.1 Кортеж

Модель памяти предполагает, что каждая ячейка содержит одно целое число. В некоторых случах требуется хранить в памяти элементы, которые на логи-

ческом 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).

В общем случае *кортежем* (или *записью*, *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 кортежа с несколькими полями используют фиксированный порядок полей. Например, поле age будет первым полем структуры Patient, room – вторым, а weight – третьим. Каждый кортеж занимает несколько подряд идущих ячеек, причем сначала хранится первое поле, потом второе, и так далее. Если известен адрес памяти K, где рассположен первый элемент кортежа, то для получения доступа к i-ому по порядку полю достаточно вычислить *смещение* этого поля относительно начала кортежа. Если все поля занимают по одной ячейке памяти, то i-ое поле структуры находится по адресу $K + i$.

Количество ячеек памяти, которые отводятся под хранение кортежа, называется его *размером*.

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

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.

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 a, $a+1$, $a+2$. 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 логически первым 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

1	2	3	$\cdots$	21	22	23	24	25	26	27	28	29	30	$\cdots$
21	10	k		a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	

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

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. 3.1. Очевидно, что при вставке нового элемента в

Алгоритм 3.1: Добавление элемента в массив по заданному индексу.

Вход: Массив A из n элементов, индекс k, новый элемент x.
Выход: 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 while
5 Сейчас позиция A[k] «освободилась»;
6 A[k] ← x;
```

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

Массивы записей. Элементами массива могут быть не только целые числа, но и записи (структуры). Если запись состоит из нескольких полей, то для ее хранения необходимо использовать несколько ячеек памяти. Пусть для одной записи требуется R ячеек. Тогда массив из N элементов будет занимать $N \cdot R$ ячеек, причем первые R ячеек соответствуют первой записи (элементу массива с индексом 0), следущие R ячеек — второму элементу и так далее.

Все записи имеют одинаковый порядок хранения полей в памяти. Поэтому адрес какого-либо поля k-ого элемента массива A может быть вычислен по формуле $A + k * R + f$, где f — смещение поля относительно адреса начала хранения записи в памяти.

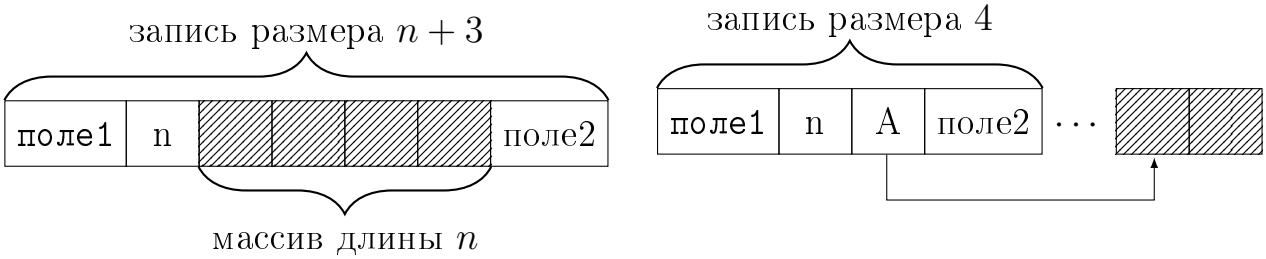
Рассмотрим пример работы с массивом записей на ассемблере. Пусть массив A, адрес начала которого записан в ячейке 1, содержит записи о пациентах. Каждая такая запись имеет четыре поля: height (рост), weight (вес), age (возраст) и temperature (температура). Предположим, нам нужно узнать температуру k-ого пациента, где значение k записано в ячеке с адресом 2. Это можно сделать следующей последовательностью команд:

```
LOAD 2   ; загрузили k
MUL =4   ; умножили на размер одной записи
ADD 1    ; прибавили адрес начала массива
;; сейчас в регистре находится адрес начала структуры k-ого пациента
;; по этому адресу записан его (или её) рост - первое поле структуры
ADD =3   ; прибавили "сдвиг" внутри структуры
STORE 3  ; куда-то сохранили, нужно для выполнения следующей команды
LOAD *3  ; загружаем A + (k*4) + 3 - искомое значение
```

Видно, что как и в случае обычного массива, время загрузки данных не зависит ни от числа элементов в массиве A, ни от размера записи.

Записи, содержащие массив. Полем записи может быть массив (вообще говоря произвольных элементов, в том числе, и записей!). Возможно как минимум два способа включения массива в запись. В первом случае массив размещается непосредственно в области памяти, которая отводится под запись. Схематично это изображено на рис. 3.2a. Такой способ характерен для статических массивов, размер которых известен в момент написания программы, Во втором варианте запись содержит одно поле, значением которого является адрес размещения массива (рис. 3.2b). В этом случае память под массив, как правило, выделяется в момент выполнения программы. Заметим, что при втором способе размеры массивов, связанных с разными экземплярами структуры, могут различаться. Поясним это на примере.

Продолжим рассматривать пример из предыдущего параграфа с массивом пациентов. Пусть теперь мы хотим определить запись, описывающую



(a) Статический массив «внутри» структуры.

(b) Указатель на массив, расположенный в другой области памяти.

Рис. 3.2: Представление структур, содержащих массив.

комнату санатория. Мы хотим знать номер комнаты, этаж и всех пациентов, которые живут в этой комнате. Такая структура, назовем её room, может иметь следующий набор полей:

- number (номер);
- floor (этаж);
- npatients (число пациентов);
- patients (адрес начала массива пациентов).

Отметим, что для задания массива пациентов мы используем два поля — длину массива (npatients) и указатель на начало массива (patients).

Если теперь мы должны задать описание нескольких комнат, например, все комнаты на этаже, то можно создать массив комнат. Некоторые комнаты могут быть одноместными, другие — двух или трехместными. Если мы используем второй способ включения массива в структуру, то размеры всех описаний комнат становятся одинаковыми, независимо от того, сколько человек в них проживает. При этом избыточная память не выделяется. При «встроенных» массивах (рис. 3.2a) для того, чтобы можно было бы сделать массив из структур room, описания одноместных комнат придется хранить в структуре, включающей трехэлементный массив (если максимальное число пациентов в одной комнате равно трем). Две ячейки памяти расходуются зря.

На практике «встроенные» массивы используются в тех случаях, когда есть естественное ограничение на максимальный размер массива, максимальная длина не очень большая (единицы или десятки элементов) и/или все экземпляры структуры используют приблизительно равное число элементов.

Пусть теперь известно, что в ячейке 1 записан адрес структуры room, и нам нужно опять получить значение температуры k-ого пациента, проживающего в этой комнате. Как и раньше, значение k записано в ячейке с адресом 2.

```
LOAD 1   ; загрузили адрес структуры room
ADD =2   ; добавили смещение; получили адрес room.patients
STORE 3  ; сохранили адрес room.patients во временную ячейку
LOAD 2   ; загрузили k
MUL =4   ; умножили на размер одной записи о пациенте
ADD *3   ; получили patients + (k*4)
ADD =3   ; прибавили "сдвиг" внутри структуры
STORE 3  ; временная ячейка
LOAD *3  ; искомое значение
```

Массивы записей, содержащих массивы в качестве полей. Предположим, что в нашей задаче помимо данных о весе и возрасте пациента дополнительно теребуется знать и перечень всех его контактов в социальной сети. Пациенты объединяются в массивы (по комнатам), а все элементы массива должны иметь одинаковый размер. С другой стороны, количество контактов в социальной сети может сильно отличаться для разных пациентов. У одного пациента контактов мало, а у другого — очень много. Если установить ограничение на максимальное число контактов, то, во-первых, есть опасность, что когда-нибудь к нам попадет пациент, число контактов которого превысит это ограничение, а, во-воторых, большинство пациентов будет иметь значительно меньшее число контактов, и память будет использоваться неэффективно. В таких ситуациях используется представление записей, при котором массив хранится отдельно от самой записи (рис. 3.2b). В нашем примере это позволяет для каждого пациента выделить массив для хранения контактов, длина которого *в точности равна* количеству реальных контактов этого пациента. Представление такой структуры в памяти приведено на рис. 3.3.

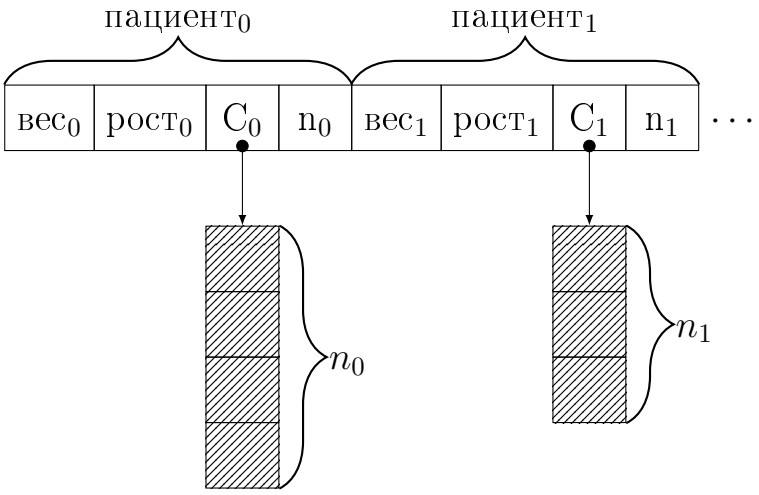


Рис. 3.3: Схематичное представление массива записей, содержащих массивы различной длины. Каждая запись о пациенте имеет поля weight, height, C (от слова Contacts) и n. Значением поля C является адрес начала массива контактов данного пациента. Поле n содержит длину этого массива.

Предположим, что нам требуется получить значение k-ого контакта p-ого пациента. Для этого достаточно знать только одно значение — адрес начала массива пациентов. По адресу массива пациентов можно вычислить адрес ячейки памяти, в которой записано искомое значение. Покажем, как это можно сделать на упрощенном ассемблере. Программа в основном совпадает с приведенной выше програмой работы с массивом записей. Отличие состоит в том, что после загрузки требуеого поля структуры дополнительно производится загрузка элемента из массива контактов.

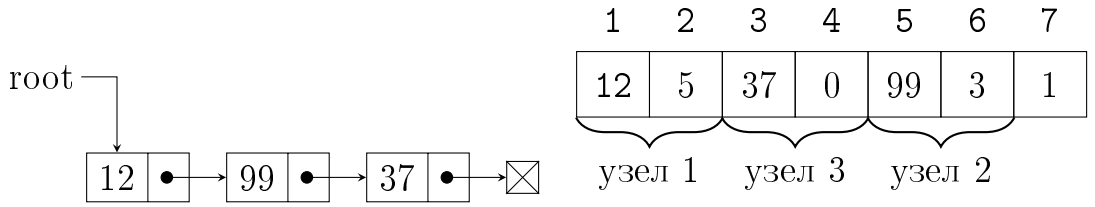
```
;; В ячейке 1 записан адрес массива пациентов;
;; в ячейке 2 -- индекс p пациента
;; в ячейке 3 -- индекс k его контакта
;; Программа загружает значение k-ого контакта пациента с индексом p
LOAD 2   ; загрузили p -- индекс пациента
MUL =4   ; умножили на размер записи об одном пациенте
ADD 1    ; прибавили адрес начала массива пациентов
         ; получили адрес, где начинается запись о p-ом пациенте
ADD =2   ; добавили смещение внутри записи
         ; получили адрес, где записано значение поля C пациента p
STORE 4  ; сохранили этот адрес во временную ячейку
LOAD *4  ; сейчас в регистр попало значение поля C пациента p
ADD 3    ; прибавили смещение в массиве контактов
STORE 4  ; еще раз сохраняем во временную ячейку
LOAD *4  ; загрузили требуемый контакт
```

В случае, когда каждый контакт в свою очередь является кортежем, доступ к его полям можно получить прибавив необходимое смещение к результату последней команды. Аналогично, если одно из полей контакта окажется массивом, можно добавлением смещения получить его элемент. В дальнейшем мы не будем рассматривать программы на ассемблере, котороые извлекают поля структур данных. По сути, обработка остальных структур данных сводится к комбинации описанных приемов для работы с записями и массивами.

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. 3.4а. Этот список содержит три числа, причем первым числом является 12, вторым – 99 и третьим – 37.



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

Рис. 3.4: Пример списка из трех элементов.

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 3.4b. Здесь первый элемент списка занимает ячейки с адресами 1 и 2. Второй элемент – ячейки с адресами 5 и 6, а третий – 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 третий 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. 3.2). Список задается своим корнем. Добавляемый элемент задан адресом node своей первой ячейки. Поскольку корень списка указывает на его

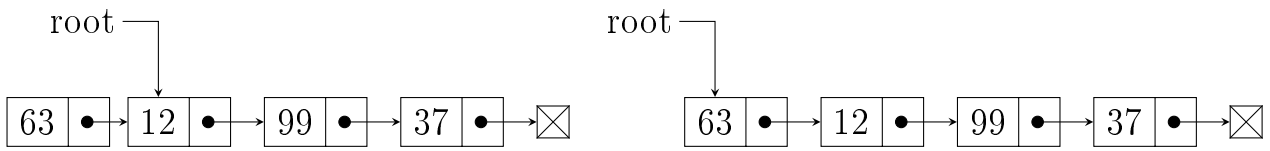
Алгоритм 3.2: Добавление элемента в начало списка.

Вход: Корень списка root, адрес нового элемента node

Выход: Список изменяется таким образом, что его первым элементом становится node

1 node.link ← root;
2 root ← node;

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,



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

Рис. 3.5: Пример добавления элемента в список по алгоритму 3.2.

вообще говоря может быть любым, то есть в памяти первый элемент может быть размещен после второго (рис. 3.4b).

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. 3.3.

Алгоритм 3.3: Добавление элемента в конец списка.

Вход: Корень списка root, новый элемент node
Выход: 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 while
5 Сейчас elem указывает на последний элемент;
6 elem.link ← node;
7 node.link ← ∅;
```

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. 3.4. 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?
  ; не нашли...
  ; загрузим адрес следующего элемента списка - он записан в ячейке,
```

Алгоритм 3.4: Поиск элемента списка по значению.

Вход: Корень списка *root*, значение x

Выход: 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 if
6 |   elem ← elem.link; // Перейдем к следующему элементу
7 end while
8 return ∅; // Просмотрели все элементы
```

```
  ; адрес которой на единицу больше содержимого ячейки 3
  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. 3.6.

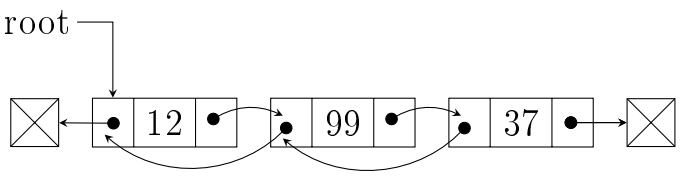


Рис. 3.6: Пример двусвязного списка из трех элементов.

Добавление элементов в двусвязный список прозводится путем изменения ссылок соседних элементов. Алгоритм добавления нового элемента после заданного элемента приведен на рис. 3.5. Предполагается, что каждый элемент

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

Алгоритм 3.5: Вставка элемента в двусвязный список.

Вход: Текущий элемент списка $current \neq \varnothing$, новый узел $node \neq \varnothing$.

Выход: 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 if
```

Строчка 4 алгоритма 3.5 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.

3.2.3 Очередь, стек и дек

Очередью (англоязычное название queue) называют линейную структуру у которой добавление новых элементов производится в конец очереди, а извлечение производится из её 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.

Очередью с приоритетом называется очередь, для каждого элемента которой задано числовое значение приоритета[2]. Операция извлечения элемента из очереди выбирает элемент с максимальным (или минимальным) значением приоритета. Таким образом, обычная очередь является очередью с приоритетом, у которой выбирается элемент с минимальным приоритетом, а в качестве значения приоритета используется время добавления элемента.

Стеком (stack) называют очередь, для которой последний добавленный элемент является первым извлекаемым. То есть добавление и извлечение элементов производится с одной стороны. Это называется очередью с дисциплиной FILO (First In Last Out) или магазинной памятью по аналогии с магазином автоматической винтовки: патрон, который вставили в магазин первым, будет стрелять последним. *Дек* (deque) — это «очередь с двумя концами». Добавлять у извлекать элементы можно с любой стороны.

[2]В общем случае значеним приоритета является элемент линейно упорядоченного множества.

Сначала мы достаточно подробно рассмотрим несколько вариантов представления «обычных» очередей, так как стек и дек по сути являются модификациями простой очереди.

3.2.4 Представление очереди

Очередь может быть реализована различными способами. В этом разделе мы рассмотрим реализации очереди на базе массива и списка (эти структуры данных будут использоваться для хранения элементов очереди). Позднее мы покажем, как можно эффективно реализовать очередь с приоритетом (когда у каждого элемента очереди есть вес «чем больше, тем ближе к началу очереди») на базе деревьев. Но независимо от выбранной реализации можно говорить, что с объектом *очередь* можно выполнять следующие операции:

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

Набор операций, которые можно выполнять с каким-либо объектом называется его *интерфейсом*.

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[3] 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 — число m, а по адресу X+2 — адрес первого элемента массива с элементами очереди.

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

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. 3.6. Соответствующая реализация на упрощенном ассемблере приведен на рис. 3.7 на странице 40.

Алгоритм 3.6: Извлечение первого элемента очереди.

Вход: Очередь X: структура с полями N, m и T.

Выход: Значение первого элемента, если очередь не пуста, или -1.

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

Реализация очереди на основе кольцевого массива. Реализация на основе массива, которая предполагает смещение всех элементов при извлечение первого элемента очереди, не самая удачная, так как для извлечения элемента из N-элементной очереди требуется выполнить $O(N)$ действий. Копирования данных можно избежать, если рассматривать *кольцевой массив*. В памяти такой массив представляется как обычный массив, но логически мы считаем, что за «последним» элементом $T[N-1]$ следует $T[0]$.

Очередь определяется:

- адресом T первого элемента массива, содержащего элементы очереди.
- максимальным размером очереди N;
- индексом $first \in \{0, \ldots, N-1\}$ первого элемента очереди;
- индексом $empty \in \{0, \ldots, N-1\}$ первого свободного элемента массива;

При создании очереди выделяется массив T на N элементов, а индексам $first$ и $empty$ присваиваются нулевые значения. Равенство этих индексов является условием пустой очереди: первый элемент очереди также является и пустым. Если в очередь добавляется новый элемент X, то он записывается в $T[empty]$ (это первая свободная ячейка в массиве T), после

```
  ;; 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. 3.7: Программа извлечения первого элемента очереди на упрощенном ассемблере.

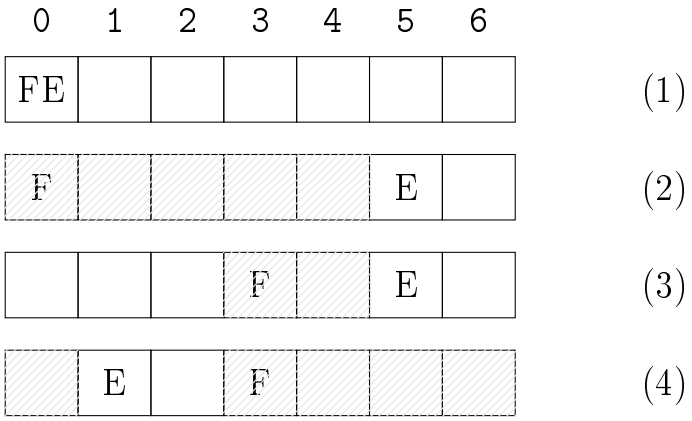


Рис. 3.8: Добавление элементов в очередь на базе кольцевого массива длины $N = 7$. Занятые элементы заштрихованы, буквами F и E обозначены «позиции» индексов first и empty. (1) пустая очередь; (2) после добавления пяти элементов; (3) после извлечения первых трех элементов; (4) после добавления еще трех элементов.

чего индекс *empty* увеличивается на единицу. Если новое значение *empty* стало равно N (элемента с индексом N в массиве T не существует, последний допустимый индекс равен $N-1$), то присваиваем индексу *empty* значение 0 (кольцевой массив). Корректное новое значение индекса *empty* можно вычислить без условного оператора «если $empty = N$, то $empty \leftarrow 0$» по формуле $empty \leftarrow (empty + 1) \bmod N$. Очередь считается полной, если $(empty + 1) = first \bmod N$, то есть после увеличения индекса *empty* на единицу (по модулю N) значения индексов совпадут.

Схематично процесс добавления и извлечения элементов из очереди представлен на рис. 3.8. Заметим, что на диаграмме (4) очередь состоит из значений элементов массива T[3], T[4], T[5], T[6], T[0] именно в таком порядке.

Алгоритм 3.7: Добавление элемента в очередь на базе кольцевого массива.

Вход: Очередь q, значение x добавляемого элемента.
Выход: Изменение очереди, если очередь не полна.

```
1 if (q.empty + 1) = q.first  mod q.N then
2 |   return;
3 end if
4 q.T[q.empty] ← x;
5 q.empty ← q.empty + 1  mod q.N;
6 return;
```

Реализация очереди на основе списка. Еще одна возможность для реализации очереди связана с использованием для хранения элементов очереди

односвязного списка. Элементы очереди записываются в список. Первый элемент очереди, который будет извлечен первым, записывается в корень списка, второй — в следующий элемент списка. И так далее. Мы видели, что удаление элементов из головы списка эффективно реализуется. Для этого достаточно просто изменить указатель на первый элемент списка. Но интерфейс очереди предполагает добавление элементов в конец очереди, что означает добавление в конец списка. Алгоритм 3.3 требует просмотра всего списка.

Очередь с приоритетом на основе массива. В разделе 3.3.3 мы рассмотрим эффективную реализацию очереди с приоритетом на основе специальной структуры данных. Сейчас покажем, как очередь с приоритетом может быть реализована на основе массива структур. В отличие от простой очереди, массив состоит из записей, содержащих два поля: value и priority. При добавление нового элемента в очередь передается значение элемента и его приоритет. Основная идея реализация состоит в том, что массив поддерживается в порядке убывания приоритета (или возрастания, если при извлечении нужно выбирать элемент с минимальным приоритетом).

Сравнение вариантов реализации очереди. Описанные способы представления очереди в памяти отличаются как по сложности алгоритмов добавления и удаления элементов в/из очереди, так и по объему памяти, который требуется при представлении очереди с заданным числом элементов. Данные приведены в таблице 3.1.

способ	память		сложность	
представления	максимум	при $K \leqslant N$	добавления	извлечения
массив	N	$O(N)$	$O(K)$	$O(K)$
циклический массив	N	$O(N)$	$O(1)$	$O(1)$
список	–	$O(K)$	$O(1)$	$O(1)$

Таблица 3.1: Сложность и требования по памяти для представления очереди, содержащей K элементов и максимум (если он задан) N элементов.

Нужно понимать, что лучшего решения не существует. Вариант с массивом однозначно хуже варианта на основе циклического массива. Но могут быть задачи, где реализация на основе списка окажется предпочтительнее реализации на основе циклического массива.

Данные таблицы 3.1 могут быть несколько обманчивы. Кажется, что списочная реализации выигрывает по сравнению с реализацией на основе циклического массива: ограничений по максимальному размеру очереди нет, а скорость работы одинаковая. Но в таблице 3.1 приводятся асимптотические оценки, которые не учитывают констант. Оба алгоритма при добавлении элемента в очередь выполняют константное число действий (не зависящее от

длины очереди), но реализация на основе списков *сложнее*. Для добавления элемента нужно выделять динамическую память, а это «дорогая» операция.

3.2.5 Представление стека и дека

Стек — это очередь для которой последний добавленный элемент будет извлекаться первым. Реализация стека по сути аналогична реализации очереди, но несколько проще, так как элементы добавляются и удалаются с одной стороны.

Реализация стека на основе массива Естественная реализация стека использует массив для хранения элементов. Сам стек определяется кортежем из трех полей: адресом массива элементов (data), его длины (n) и степени заполненности (filled). Схематичное представление такой структуры приведено на рис. 3.9.

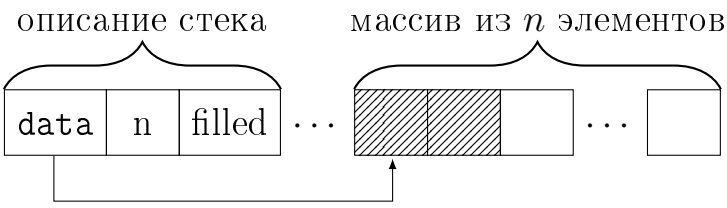


Рис. 3.9: Представление стека на основе массива. Заполненная часть массива заштрихована. Рисунок соответствует состоянию, когда поле filled имеет значение 2.

При создании стека выделяется память под массив data, а поле filled получает значение 0 (в стеке нет ни одного элемента). Стек полностью заполнен, если значения полей filled и n совпадают. Добавление элемента в стек (при условии, что он не полностью заполнен) сводится к записи нового элемента в позицию filled массива data и увеличению filled на единицу. Извлечение элемента сначала уменьшает значение filled, а потом возвращает значение data[filled].

Реализация стека на основе списка

3.3 Деревья и графы

Графом называется пара $G = \langle V, E \rangle$, где V — конечное множество *вершин* графа, а $E \subseteq V \times V$ — множество *ребер* графа. Граф называется *неориентированным*, если множество ребер состоит из неупорядоченных пар вершин,

то есть из $(u, v) \in E$ следует $(v, u) \in E$. Вершины v_1 и v_2 графа G называются *смежными*, если $(v_1, v_2) \in E$. *Путь* в графе G — это последовательность попарно смежных вершин $v_1, v_2, \ldots v_n$, то есть для всех $1 \leqslant i \leqslant n-1$ выполняется $(v_i, v_{i+1}) \in E$. *Циклом* называется путь, первая и последняя вершина которого совпадают. Граф называется *связным*, если любая пара его вершин соединена по крайней мере одним путем. Граф называется *ациклличным*, если он не содержит циклов.

Пусть X — некоторое множество, например, целые числа $\mathbb{Z}$ или множество Σ^+ всех слов в алфавите Σ. Граф называется X-помеченным (или взвешенным), если для него задана функция $val : V \to X$, то есть каждой вершине графа сопоставлено некоторое значение. Значение вершин не обязательно должны быть числовыми и, вообще говоря, могут быть равными для нескольких вершин графа. Часто рассматривают графы с помеченными ребрами. Например, весом ребра может быть «расстояние» между вершинами.

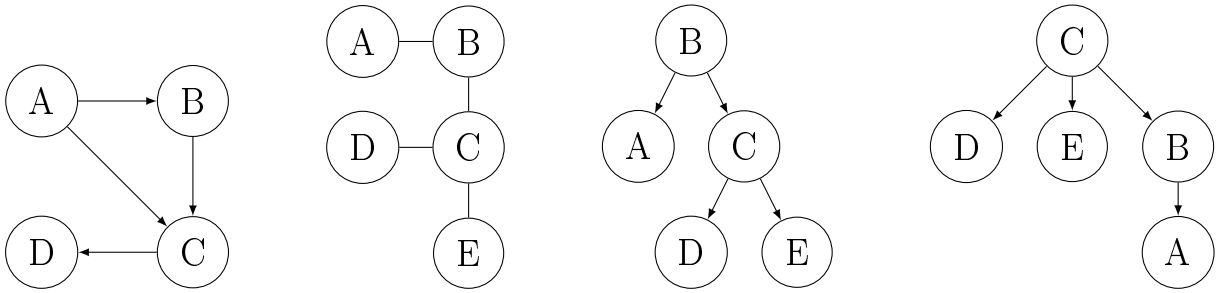
Вершины графа принято изображать в виде окружностей (или просто точек), а ребра — в виде дуг, соединяющих пары смежных вершин. Если граф ориентированный, то на конце дуги ставится стрелка (или на обоих концах, если обе пары (u, v) и (v, u) принадлежат множеству ребер E). Стрелка указывает «направление» пары вершин: от первого элемента ко второму. При изображении неориентированных графов стрелки не ставятся. Пример графа представлен на рис. 3.10.

Деревом называется связный неориентированный граф без циклов. Отметим, что список является частным случаем дерева — это дерево с одним листом. В практических приложениях удобно выделять в дереве *корневую вершину*. Выбор корневой вершины позволяет говорить об отношении «родитель-потомок» между вершинами дерева и расположить вершины дерева по *уровням* (или *ярусам*). Из определения дерева следует, что для каждой вершины v существует ровно один путь, соединяющий корень r и v (связный граф без циклов). Длину этого пути назовем уровнем вершины v. Корневая вершина имеет уровень 0. Пусть вершина v расположена в дереве T на уровне $n > 0$. Тогда смежные с ней вершины будут на уровнях $n-1$ и $n+1$, причем на уровне $n-1$ будет ровно одна вершина p. Эту вершину будем называть родительской для n, а все остальные — дочерними. Вершины дерева, которые не имеют дочерних, называются *листовыми*. Вершины, которые не являются листовыми или корневой, называются внутренними вершинами деева.

Далее выражения *вершина* дерева и *узел*дерева будут использоваться как синонимы.

3.3.1 Бинарное дерево

Дерево называется *бинарным* (или двоичным), если для любого его элемента существует не более двух дочерних элементов, причем правый и левый



(a) Ориентированный граф.

(b) Нериентированный граф без циклов.

(c) Дерево с корнем B.

(d) Дерево с корнем C.

Рис. 3.10: Примеры графов и деревьев. Деревья на рис. (c) и (d) получены из графа (a) выбором в качестве корня вершин B и C, соответственно.

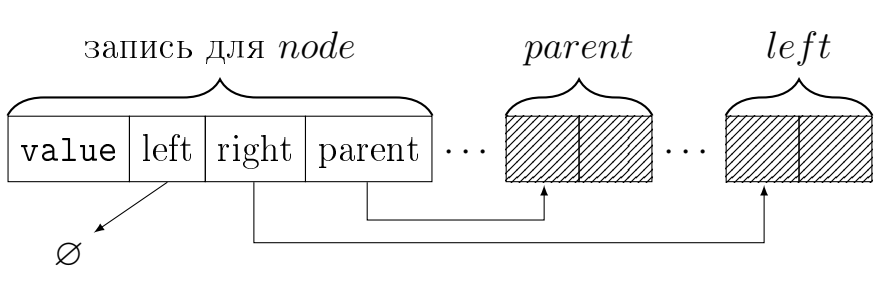
дочерний элемент различаются (если у элемента есть только один дочерний элемент, то он либо правый, либо левый). Дерево на рис. 3.10c — бинарное, а на рис. 3.10d — нет, так как у корня C три дочерних вершины. Если у вершины бинарного дерева поменять местами правую и левую дочерние вершины, то получится другое дерево.

Для представления бинарных деревьев в памяти ЭВМ вершины удобно описывать в виде записи, включающей поля:

- value — значение, приписанное вершине помеченного дерева;
- left — указатель на область памяти, содержащую аналогичную запись для левого дочернего узла;
- right — указатель на запись для правого дочернего узла;
- parent — указатель на запись для родительского узла.

Если вершина дерева не имеет левого, правого или родительского узла (последняя ситуация возможна только для корневого узла дерева), то соответствующее поле записи имеет нулевое значение.

Пусть значение узла node равно X и этот узел не имеет левого дочернего узла. Тогда запись, представляющая этот узел будет иметь нулевое значение поля left и ненулевые значения полей right и parent. Схематично представление такого узла дерева изображено на рис. 3.11. Заметим, что по значению полей записи node нельзя сказать, является этот узел правым или левым дочерним для своего родителя. Поэтому родительский узел на рис. 3.11b изображен строго над узлом node. Также отметим, что без обращения к полям записей из заштрихованных областей памяти невозможно определить значения родительской и правой дочерней вершины *node*. Ненулевые значения полей *node.right* и *node.parent* позволяют сказать, что у вершины есть родительская и дочерняя вершина. Как и в случае списка, адреса памяти, где расположены данные записей, не имеют принципиального значения.



(a) Представление в памяти. Заштрихованные ячейки означают области памяти, занятые записями аналогичной структуры.

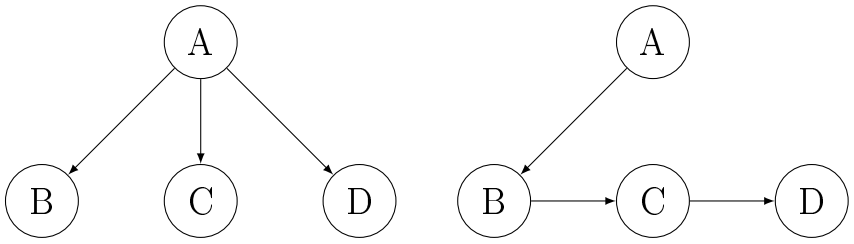
(b) Логическое представление.

Рис. 3.11: Представление узла бинарного дерева в памяти.

Добавление и удаление узлов бинарного дерева. ...

3.3.2 Дерево произвольного вида

Структура данных для хранения произвольного дерева сложнее, чем для бинарного, так как количество дочерних узлов не является постоянным. Некоторые узлы дерева могут иметь небольшое дочерних число вершин, в то время как другие — очень много.



(a) Вершина произвольного дерева.

(b) Измененная структура: горизонтальные ссылки указывают на следующий соседний узел.

Рис. 3.12: Преобразование узла произвольного дерева. На рис (a) изображен узел A и его дочерние узлы.

Унификация достигается за счет добавления ссылок на соседний узел («брата»). Назовем два узла дерева *братьями* (или *соседями*), если у них совпадает родительский узел. Дочерние вершины *node* можно связать в список. В этом случае для описания вершины дерева достаточно знать адрес первого элемента такого списка. Поскольку любая вершина дерева, отличная от его корня, попадает ровно в один такой список (список дочерних узлов ее родителя), то ссылку на следующий элемент можно разместить непосредственно в записи, представляющей узел дерева. В вершине дерева достаточно хранить одну ссылку на первый дочерний узел и одну ссылку на соседний узел. Это преобразование представлено на рис. 3.12.

Таким образом, вершина произвольного (помеченного) дерева описывается следующими полями:

- value — значение вершины;
- parent — ссылка на родительский узел;
- child — ссылка на первый дочерний узел;
- sibling — ссылка на следующего брата.

Узлы дерева являются элементами односвязного списка по полю sibling.

3.3.3 Пирамида

Любому массиву можно сопоставить «почти полное» бинарное помеченное дерево, если заполнять узлы дерева элементами массива сверху вниз и слева направо. Нулевой элемент массива соответствует корню дерева. Элементы с индексами 1 и 2 соответствуют левому и правому дочернему узлу корня дерева. Узлам следующего уровня дерева соответствуют элементы с индексами 3,4,5 и 6. И так далее. Если массив закончился, то последний уровень оказывается не полностью заполненным.

Кучей или *пирамидой* (по-английски heap) называется массив A из n элементов, такой что для любого индекса $0 \leqslant k < n$ выполняются следующие условия:

$$\begin{array}{l} \text{если } 2k+1 < n, \text{ то } A[k] \geqslant A[2k+1]; \\ \text{если } 2k+2 < n, \text{ то } A[k] \geqslant A[2k+2]. \end{array} \tag{3.1}$$

В отличие от других структур данных, которые мы рассматривали ранее, определение кучи учитывает значения элементов массива.

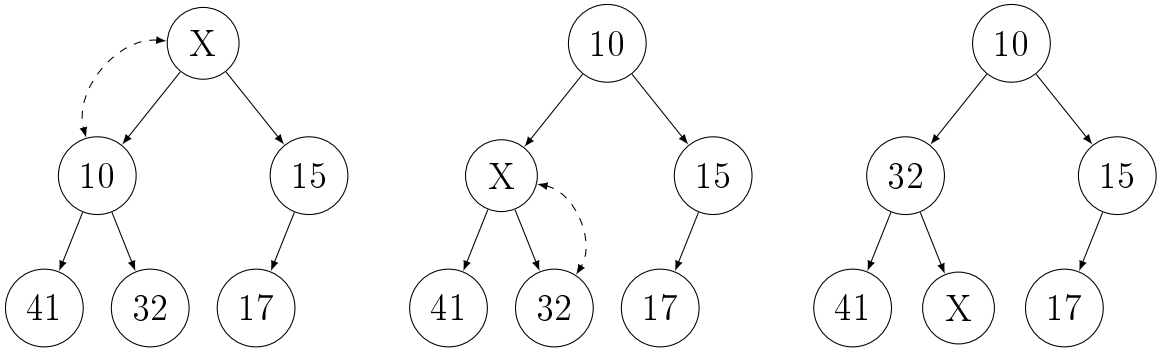
Пусть по массиву A было построено «почти полное» бинарное дерево $T = T(A)$. Определение кучи в терминах помеченного бинарного дерева T означает, что для каждого узла $node$ дерева T значение узла $node$ не меньше значений его левого и правого дочерних узлов.

Соответствие между массивами и бинарными деревьями позволяет задать структуру дерева *без использования дополнительной памяти* для хранения указателей на левое и правое поддеревья. Действительно, если задан индекс $k < n$ некоторого элемента массива, то ему соответствует узел $node$ дерева T. Индексы элементов массива, соответствующих левому дочернего узлу $left(node)$, правому дочернему узлу $right(node)$ и родительскому узлу $parent(node)$, могут быть получены по формулам:

$$\begin{array}{rcl} left & = & 2k+1 \\ right & = & 2k+2 \\ parent & = & \lfloor (k+1)/2 \rfloor - 1 \end{array} \tag{3.2}$$

Таким образом, зная индекс k можно перейти к дочернему или родительскому элементу. Естественно, необходимо проверять, что эти элементы существуют в массиве, то есть вычисленное по такой формуле значение принадлежит множеству $\{0, 1, \ldots, N-1\}$.

Построение пирамиды. Произвольный массив чисел не обязательно является кучей. Задача построения кучи по массиву состоит в выполнении такой последовательности обменов элементов массива, после которой условия 3.1 будут выполняются для всех элементов массива. Построение пирамиды (будем использовать эту ассоциацию) происходит снизу-вверх. Идея алгоритма состоит в следующем. Пусть у нас уже построена пирамида (с условием, что элементы массива с меньшими индексами должны иметь меньшее значение), но в ее вершине находится «большое» число, которое нарушает условие 3.1. Тогда это значение можно «спустить» на нужный уровень, проведя последовательность обменов. Алгоритм приведен на рис. 3.8.



(a) Пирамида для массива $X, 10, 15, 41, 32, 17$.

(b) После одного обмена.

(c) Конечный результат после второго обмена.

Рис. 3.13: Пример добавления корня со значением $X = 40$ к «готовой» пирамиде. Пунктирная стрелка показывает вершины для обмена.

Рассмотрим сначала работу этого алгоритма на примере массива, изображенного на рис. 3.13. Если значение X больше одного из своих дочерних узлов, то оно не может находится в вершине пирамиды по определению. Тогда в вершину «выводится» наименьшее значение из $\{X, 10, 15\}$. Этот обмен отмечен пунктирной стрелкой на рис. 3.13a. Заметим, что для выявления необходимости обмена и его выполнения достаточно обратится к узлу дерева и его дочерним узлам. Далее аналогичную операцию необходимо произвести в «новой позиции» элемента X. Теперь выбор минимума производится из множества $\{X, 41, 32\}$. Необходимый обмен отмечен пунктиром.

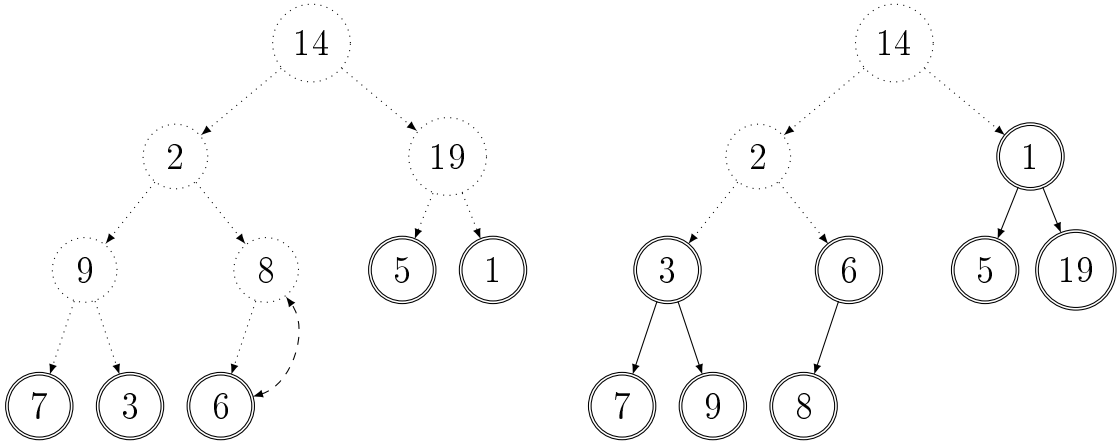
Отметим, что элемент X может «остановиться» раньше. Например, если X будет равно 20, то дерево на рис. 3.13b будет итоговым. В худшем случае элемент X станет листовым. Тогда будет совершено $O(\log_2 n)$ обменов, где n — общее число вершин (куча — почти идеальное дерево).

Алгоритм 3.8: HeapSink: «спуск» вершины пирамиды.

Вход: Массив A, n — длина массива, k — индекс вершины.
Результат: Узел с индексом k является вершиной пирамиды.

```
1  i ← k ; // Позиция A[k] при «спуске»
2  repeat                                          // Пока были обмены
       // Вычислим индексы поддеревьев i по формуле (3.2)
3      left ← 2i + 1;
4      right ← 2i + 2;
5      if right < n u A[left] > A[right] then least_at ← right ;
6      else least_at ← left ;
7      if A[i] > A[least_at] then
8          Поменять(A[i], A[least_at]);
9          i ← least_at;
10     end if
11 until i = least_at;
```

Для построения пирамиды по исходному массиву будем его просматривать справа-налево, добавляя каждый раз по одному элементу. Заметим, что каждый элемент массива, не обязательно с индексом 0, является вершиной (самым верхним элементом) некоторой пирамиды. Например, узел со значением X на рис. 3.13b, который в массиве имеет индекс 1, является вершиной пирамиды с узлами $X, 41, 32$. Индексы правого и левого узлов вычисляются по формулам (3.2). Всё это дает возможность свести задачу построения пирамиды к только что описанной процедуре спуска вершины пирамиды.



(a) Представление массива в виде дерева.

(b) После включения $8, 9, 19$.

Рис. 3.14: Построение пирамиды по массиву $14, 2, 19, 9, 8, 5, 1, 7, 3, 6$. Двойные окружности означают, что элемент включен в пирамиду.

Сначала в пирамиду включатся все листовые элементы массива и только они (рис. 3.14a). Элемент с индексом k листовой, если $2k + 1 \geqslant n$. Каждый лист образует пирамиду с одним узлом. Далее к пирамиде (пока у нас

есть несколько не связанных друг с другом пирамид) присоединяется первый нелистовой элемент массива. В нашем примере это элемент со значением 8 (и индексом 4). К этому элементу применяется процедура HeapSink «спуска» вершины пирамиды. Пунктирной стрелкой показан следующий обмен значений. После этого к пирамиде аналогичным образом добавляются другие элементы. На рис. 3.14b показывается состояние недостроенной пирамиды после обработки элементов массива со значениями 8, 9 и 19. Добавление каждого нелистового элемента объединяет две пирамиды в одну. Процесс добавления элементов продолжается до тех пор, пока в пирамиду не будет добавлен элемент массива с индексом 0. Формальное описание алгоритма приведено на рис. 3.9.

Алгоритм 3.9: MakeHeap: построение пирамиды по массиву.

Вход: Массив A, n — длина массива.

Выход: Массив A образует пирамиду.

```
1 k ← ⌊(n−1)/2⌋ − 1          // Индекс последнего нелистового элемента
2 while k ⩾ 0 do                              // Пока не дошли до вершины
3 |   HeapSink(A, n, k);
4 |   k ← k − 1;
5 end while
```

Реализация очереди с приоритетом. *Очередью с приоритетом* называется очередь, для каждого элемента которой задано числовое значение — *приоритет*[4]. Операция извлечения элемента из очереди выбирает элемент с максимальным значением приоритета. Очередь с приоритетом может быть эффективно реализована на основе пирамиды (точнее — антипирамиды, в корене которой стоит максимальный элемент). Добавление элемента в очередь с приоритетом сводится к добавлеюнию нового элемента в пирамиду. Извлечение элемента соответствует удалению корневого элемента пирамиды. Обе операции могут быть выполнены за $O(\log_2 N)$, где N — текущая длина очереди.

[4]В общем случае приоритет является элементом линейно упорядоченного множества.

3.3.4 Граф

3.4 Матрицы и разреженные матрицы

3.5 Задачи и упражнения

3.1 Пусть задан адрес некоторого узла двоичного дерева. Как можно проверить, что этот узел является правым дочерним узлом для своего родителя?

3.2 Предположим, что мы должны реализовать систему обработки электронной очереди в банке. Какую реализацию очереди вы выберете, если в зале не может одновременно находиться более N человек?

3.3 Покажите, как на основе одного массива из N элементов можно реализовать два стека, каждый из которых может содержать до N элементов при условии, что в сумме они содержат не более N элементов.

3.4* Напишите программу на ассемблере, которая вычисляет высоту бинарного дерева. Дерево задается адресом памяти, по которому расположена запись с двумя полями — адресом левого и правого поддеревьев (указатели на аналогичные записи). Программа может использовать $O(n)$ ячеек памяти, где n — число вершин исходного дерева.

3.5 Алгоритм 3.9 выстраивает пирамиду снизу-вверх. Напишите алгоритм, который выстраивает прирамиду обрабокой массива слева-направо, то есть добавлением новых элементов к существующей пирамиде снизу.

3.6 Как определить, что односвязный список, заданный адресом корневого элемента, не содержит циклов? Копировать и изменять список не разрешается. Алгоритм должен использовать $O(1)$ ячеек дополнительной памяти.

Глава 4

Алгоритмы

4.1 Рекурсивные и итеративные алгоритмы

Функция называется *рекурсивной*, если ее значение зависит от значений этой же функции при других значениях аргумента. Классическим примером рекурсивной функции является функция вычисления числа Фибоначчи:

$$fib(n) = \begin{cases} 1, & n < 2 \\ fib(n-1) + fib(n-2) & n \geqslant 2. \end{cases} \tag{4.1}$$

Для вычисления n-ого числа необходимо знать два предыдущих числа Фибоначчи. Нерекурсивное задание первых двух значений функции fib позволяет вычислить любое число Фибоначчи.

Многие задачи, в том числе, связанные с обработкой структур данных, естественном образом описываются в виде рекурсивных функций. Например, определение числа элементов списка можно задать в следующей форма:

$$len(root) = \begin{cases} 0, & root = \varnothing \\ 1 + len(root.next), & \text{иначе.} \end{cases} \tag{4.2}$$

В данном случае используется рекурсивный характер структуры данных. Каждый элемент списка является корневым элементом «хвоста» исходного списка.

Алгоритмами называют конструктивно заданные функции. Если алгоритм «вызывает сам себя», то он называется рекурсивным. Выражение (4.2) может быть реализовано в виде рекурсивного алгоритма (алг. 4.1). Если поданный на вход адрес не нулевой, то список содержит по крайней мере один элемент, а для вычисления длины списка нужно вычислить длину его «хвоста».

Итеративным алгоритмом называется алгоритм, который использует циклические конструкции и не является рекурсивным. Вычисление длины списка можно свести к циклу по всем элементам списка (алг. 4.2).

Алгоритм 4.1: Рекурсивная версия вычисления длины списка.

```
1 Function ListLenRec(elem) is
      Вход: elem – адрес корневого элемента списка.
      Результат: Длина списка, который начинается в elem.
2     if elem = ∅ then
3         return 0;
4     end if
5     return 1 + ListLenRec (elem.next);
6 end
```

Алгоритм 4.2: Итеративная версия вычисления длины списка.

```
1 Function ListLenIter(elem) is
      Вход: elem – адрес корневого элемента списка.
      Результат: Длина списка, который начинается в elem.
2     len ← 0;
3     while elem ≠ ∅ do
4         elem ← elem.next;
5         len ← len + 1;
6     end while
7     return len;
8 end
```

4.1.1 Сведение рекурсивных алгоритмов к итеративным

Хвостовая рекурсия.

Преобразование произвольных рекурсивных функций.

4.2 Алгоритмы поиска

Песть задано некоторый набор записей, содержащих поле key. Задача поиска состоит в нахождении такой записи, у которой значение поля key имеет заданное значение K. Поле key называется *ключом поиска*, а сам процесс поиска — поиском по ключу.

Задачи поиска встречаются повсеместно. Например, толковый словарь является набором записей, описывающих значения слов естественного языка. Ключом поиска в данном случае может выступать слово. В системе учета успеваемости студентов каждая запись может включать данные о конкретном студенте, фамилия, номер группы, полученные оценки и т.п., а ключем

поиска может быть номер зачетной книжки.

В этом разделе мы будем рассматривать возможные реализации абстрактного типа данных, соответсвующий понятиям множества или мультимножества, если мы будем допускать повторяющиеся значения. Интерфейс этого типа данных состоит из трех функций:

- добавление нового элемента;
- поиск элемента по ключу;
- удаление элемента.

В практических приложениях результатом поиска является запись. Например, по номеру зачетной книжки можно получить всю информацию о студенте. Для описания алгоритмов поиска нам потребуются только значения поля key, поэтому мы будем считать. что рассматривамый абстрактный тип данных является множеством ключей, без какой-либо дополнительной инофрмации. Более того, мы будем считать, что ключи поиска являются целыми числами. Таким образом, мы будем рассматривать реализации *множеств (или мультимножеств) целых чисел.*

4.2.1 Последовательный поиск

Алгоритм 4.3: Последовательный поиск в массиве.

Вход: Массив $A[0], \ldots, A[n-1]$, key – ключ поиска
Выход: 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 if
6 |   i ← i + 1;
7 end while
8 return -1;
```

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. 4.3.

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

4.2.2 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 применив алгоритм *двоичного (бинарного) поиска*. Предположим, что требуется найти в массиве A позицию элемента со значением K. Алгоритм двоичного поиска основан на следующей идее. Сравним значение среднего элемента массива, то есть элемента с индексом $m = \lfloor N/2 \rfloor$, с искомым значением K. Возможны три варианта: $A[m] = K$, $A[m] < K$ и $A[m] > K$. В первом случае поиска завершился успешно: искомое значение встречается в массиве A по индексу m. Во втором случае мы можем воспользоваться тем, что массив A не убывает. Если $A[m] < K$, то для любого индекса $m' < m$ справедливо неравенство $A[m'] < K$. Из этого следует, что искомое значение K может встретиться только в *половине* массива. Аналогичное рассуждение позволяет сократить в два раза область поиска в случае, когда $A[m] > K$.

Алгоритм 4.4: Двоичный поиск.

Вход: Упорядоченный массив $A[0], \ldots, A[n-1]$, $A[i] \leqslant A[i+1]$ для всех $0 < i < n-1$; K – значение

Выход: 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 if
7  |  if A[mid] > K then
8  |  |  r ← mid − 1;
9  |  else
10 |  |  l ← mid + 1;
11 |  end if
12 end while
13 return −1
```

Псевдокод алгоритма двоичного поиска приведен на рис. 4.4. Поскольку каждая итерация цикла, который начинается в строке 2 алгоритма 4.4, сокращает область поиска в два раза (либо левая, либо права граница поиска передвигается в середину), то время поиска заданного значения составляет $O(\log_2 n)$.

4.2.3 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].

Алгоритм 4.5: Метод линейных проб. Вставка элемента.

Вход: Массив T из n элементов, хеш-функция h, новое значение x.
Выход: Индекс добавленного элемента или -1.
Результат: Изменяется массив T.

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

[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.

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 4.5 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. 4.6).

Алгоритм 4.6: Метод линейных проб. Поиск элемента.

Вход: Массив T из n элементов, хеш-функция h, значение x
Выход: 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 if
8  |  next ← i + 1 mod n;
9  until T[i] ≠ −1 и next ≠ k;
10 return −1; // Встретили пустую или просмотрели все элементы
```

Если из хеш-таблицы нужно удалить значение x, то для этого может потребоваться перемещение некоторых других элементов. Это необходимо для того, чтобы алгоритм поиска смог найти такие элементы y, которые занимают позиции, отличные от $h(y)$. Алгоритм удаления приведен на рис. 4.7.

Условие в строке 5 алгоритма 4.7 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 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. 4.1. При удалении значения 14 ячейка с индексом 0 освобождается. На рисунке освободившаяся ячейка перечеркнута (строка

Алгоритм 4.7: Метод линейных проб. Удаление элемента.

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 4.6)
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 if
9      i ← i + 1 mod n;
10 end while
```

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 4.6 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 элементы 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

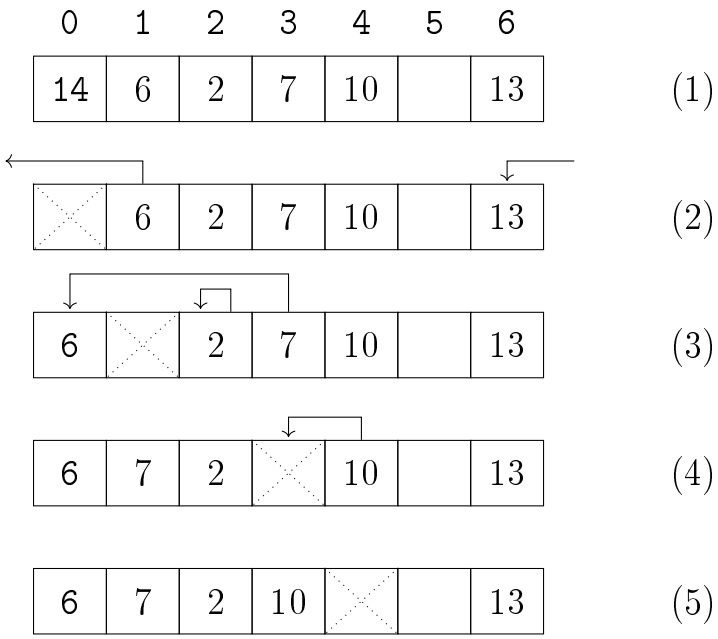


Рис. 4.1: Пример удаления из хеш-таблицы значения 14, $h(x) = x \bmod 7$.

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

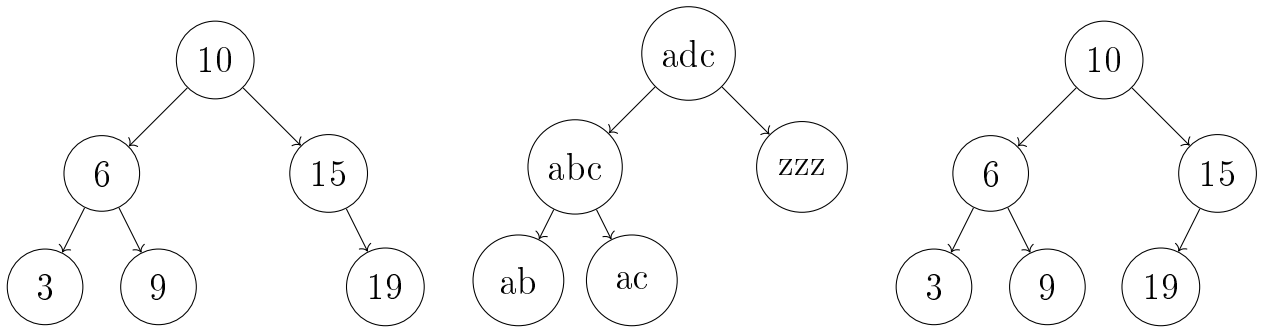
Идеальные таблицы расстановки. Интересная задача возникает в случае, когда множество ключей поиска *фиксировано*. Например, в толковом

словаре новые записи не появляются, а поиск словарных статей по запросу-слову требуется выполнять часто. При фиксированном множестве ключей можно подобрать такую хеш-функцию, которая на этом множестве ключей не будет давать коллизий. Если дополнительно потребовать, что размер таблицы должен совпадать с количеством обрабатываемых записей (словарных статей), то такую таблицу можно назвать идеальной: поиск осуществляется простым обращением по индексу (отсутствие коллизий), а дополнительная память, размер которой в общем случе выбирается на основании Теоремы 3, не требуется.

Хеш-функция называется *совершенной*, если областью её значений является начальный отрезок натурального ряда и она не имеет коллизий на заданном множестве ключей. Совершенная хеш-функция назвается *минимальной*, если число ее возможных значений совпадает с числом элементов множества возможных ключей.

4.2.4 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.



(a) Дерево поиска, вершины которого помечены числами.

(b) Дерево поиска, вершины которого помечены строками.

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

Рис. 4.2: Примеры деревьев поиска.

Рассмотрим в качестве примера деревья, изображенные на рис. 4.2. На

[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.

первом рисунке (рис. 4.2a) 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. 4.2b также является деревом поиска, если его узлы сравниваются в лексикографическом порядке. На рис. 4.2c 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.

Текст алгоритма поиска в дереве поиска приведен на рис. 4.8. Рассмотрим работу этого алгоритма в случае, когда дерево поиска соответствует рис. 4.2a 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 алгоритм заканчивает свою работу в строке 4. Заметим, что если бы дерево поиска соответсвовало бы рис. 4.2c, 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

Алгоритм 4.8: Поиск по значению узла в дереве поиска.

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

```
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 if
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 if
11 end while
12 return ∅; // Поиск закончился неудачей
```

Алгоритм 4.9: Поиск минимального значения.

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

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

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

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

Алгоритм 4.10: Добавление в непустое дерево поиска.

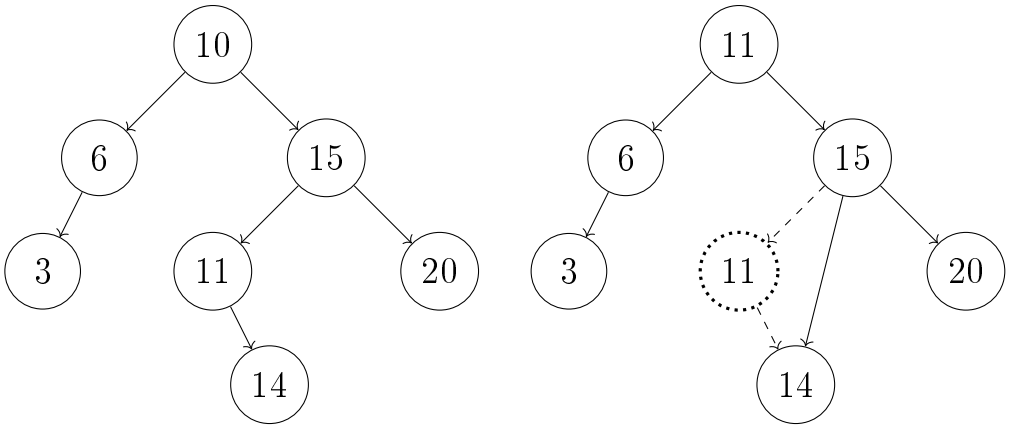
Вход: Корневой узел $root \neq \varnothing$ дерева поиска T, значение x
Выход: Адрес узла дерева, значение которого равно x

```
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 if
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 if
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 if
22 return node
```

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.4.3а 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. 4.3b. 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. 4.3: Удаление элемента из дерева.

Алгоритм удаления значения из дерева поиска приведен на рис. 4.11. 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. 4.3a переменная $minR$ будет указывать узлу со значением 11, а *parent* — на узел со значением 15. Далее производится изменение ссылок в соответсвии с рис. 4.3b. 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.

4.2.5 Сбалансированные деревья

АВЛ-дерево это дерево поиска для которого выполняется дополнительное условие: **для любого узла дерева** глубина его левого поддерева отличается

Алгоритм 4.11: Удаление корня дерева поиска.

Вход: Корневой узел $root \neq \varnothing$ дерева поиска T
Выход: 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 while
7  if minR = ∅ then                    // У root нет правого поддерева
8  |   newRoot ← root.left;
9  |   освободитьПамять(root);
10 |   return newRoot
11 end if
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 if
18 освободитьПамять(minR);
19 return root
```

от глубины правого поддерева не более чем на единицу.

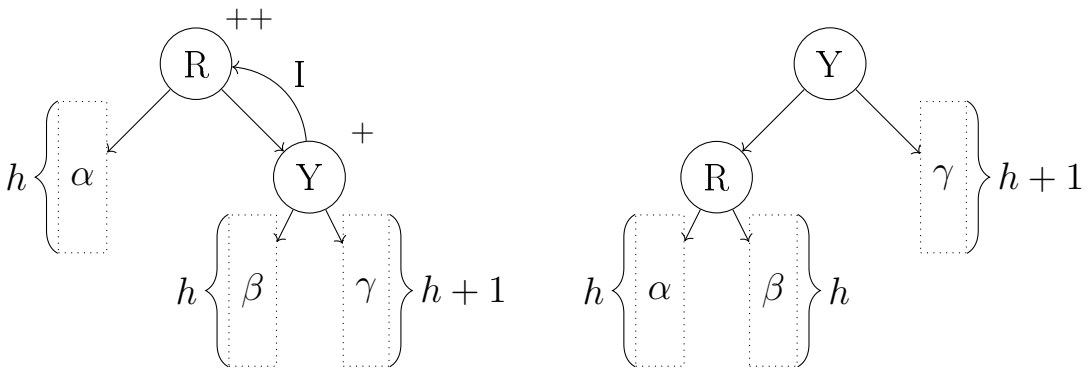
Основная идея применения сбалансированных деревьев состоит в том, что они обспечивают гарантированно высокую скорость работы алгоритмов поиска, добавления и удаления элементов. Если обычное дерево поиска может в вырожденном случае представлять собой список, то АВЛ-дерево имеет «примерно равные длины ветвей». Для АВЛ-деревьев известна верхняя оценка высоты.

Теорема 4. *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. 4.4 и рис. 4.5. 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.

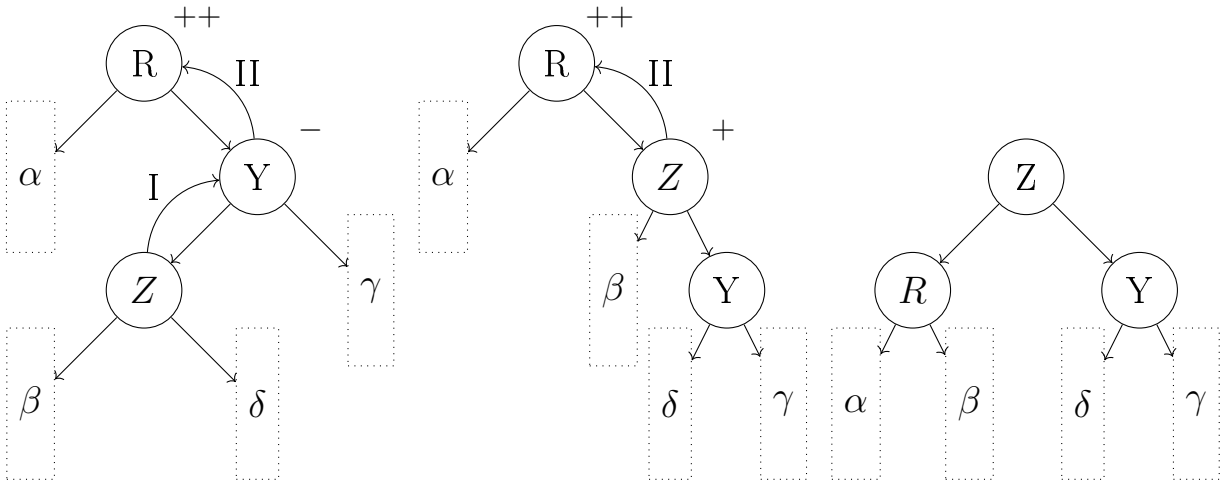
Рис. 4.4: Одинарный поворот в корне АВЛ-дерева.

4.2.6 Сильно ветвящиеся деревья

4.2.7 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	$O(1)$	$O(n)$	$O(n)$	$O(1)$
$\leqslant$-список	$O(n)$	$O(n)$	$O(n)$	$O(1)$
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 3). В худшем случае, когда таблица заполнена полностью, для каждой из задач может потребоваться $O(n)$ операций.



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

Рис. 4.5: Двойной поворот при добавлении в АВЛ-дерево. Поддеревья α, β, δ и γ имеют равную высоту h.

4.3 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 ключом 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. *Сортировка подсчетом* вычисляет для каждого элеента массива его позицию в отсортированном массиве, то есть число элементов, значение которых не превосходят текущий. Существуют специализированные алгоритмы сортировки, которые разработаны при условии, что

исходный массив удовлетворяет некоторым ограничениям. Одним из таких примеров является сортировка массива заранее известной (небольшой) длины или сортировка подсчетом при условии, 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.

4.3.1 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. 4.12.

Алгоритм 4.12: Сортировка простыми вставками.

Вход: Массив A из n элементов
Выход: 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 while
10 |  ready ← ready + 1;
11 end while
```

Внешний цикл выполняется n раз, для всех значений $ready$ в диапазоне 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 $O(n^2)$.

4.3.2 Сортировка выбором

Еще один тип алгоритмов сортировки основан на следующем наблюдении. В отсортированном массиве на первом месте стоит наименьший элемент (при сортировке по возрастанию). На втором месте — наименьший *из оставшихся*. И так далее. Алгоритм сортировки можно построить на выборе наименьшего значения из неотсортированной части массива.

Пусть есть функция Swap, которая меняет местами два значения, и функция MinPos, которая по заданному адресу массива и его длине вычисляет позицию максимального элемента. Для заданных A и n функция MinPos возвращает индекс $m \in \{0, \ldots, n-1\}$ такой, что $A[m]$ не меньше любого другого элемента массива. Заметим, что вызов MinPos$(A+k,\, n-k)$ позволяет получить позицию максимального элемента в массиве $A[k], A[k+1], \ldots A[n-1]$. Возвращаемое значение при этом будет принадлежать множеству $\{0, \ldots, n-k\}$.

Алгоритм 4.13: ChoiceSort: Сортировка выбором.

Вход: Адрес A начала массива, число элементов n.
Выход: Изменение порядка элементов массива A.

```
1 k ← 0;                                  // Первый не обработанный элемент
2 while k < n − 1 do
3 |   m ← k + MinPos(A + k, n − k);        // Позиция минимума
4 |   Swap(A + m, A + k);
5 |   k ← k + 1;
6 end while
```

4.3.3 Сортировка пузырьком

Сортировка пузырьком является одним из самых простых, известных 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. 4.14, 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, если в процессе очередного прохода не было выполнено ни одного обмена.

Алгоритм 4.14: BubbleSort: Сортировка пузырьком.

Вход: Массив A из n элементов
Выход: 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 if
10 |  |  k ← k − 1;
11 |  end while
12 |  processed ← swapAt + 1;
13 until swapAt ⩾ 0;
```

4.3.4 Сортировка слиянием

Сортировка слиянием является рекурсивным алгоритмом, который основан на том, что два отсортированных массива могут быть эффективно объединены в один упорядоченный массив. Если входной массив достаточно короткий, то он сортируется каким-либо из описанных ранее алгоритмов. Если же входной массив «большой», то он разбивается на части, каждая из которых сортируется отдельно. Схематично процесс сортировки приедставлен на рис. 4.6.

Опишем процесс слияния упорядоченных массивов.

4.3.5 Пирамидальная сортировка

Эффективный алгоритм сортировки может быть построен с использованием кучи 3.3.3. Его называют *пирамидальной сортировкой*, сортировкой кучей. Распространенное название реализующей этот алгоритм функции — HeapSort. Предположим, что по исходному массиву была построена куча. Из определения следует, что нулевой элемент массива является наименьшим из всех (или наибольшим, если пирамида была построена с условием $\geqslant$) . Если мы удалим этот элемент, а из оставшихся элементов сформируем кучу, то это приведет к алгоритму сортировки, так как следующим значеним по индексу 0 будет максимальный из оставшихся элементов массива.

Код этого алгоритма приведен на рис. 4.15.

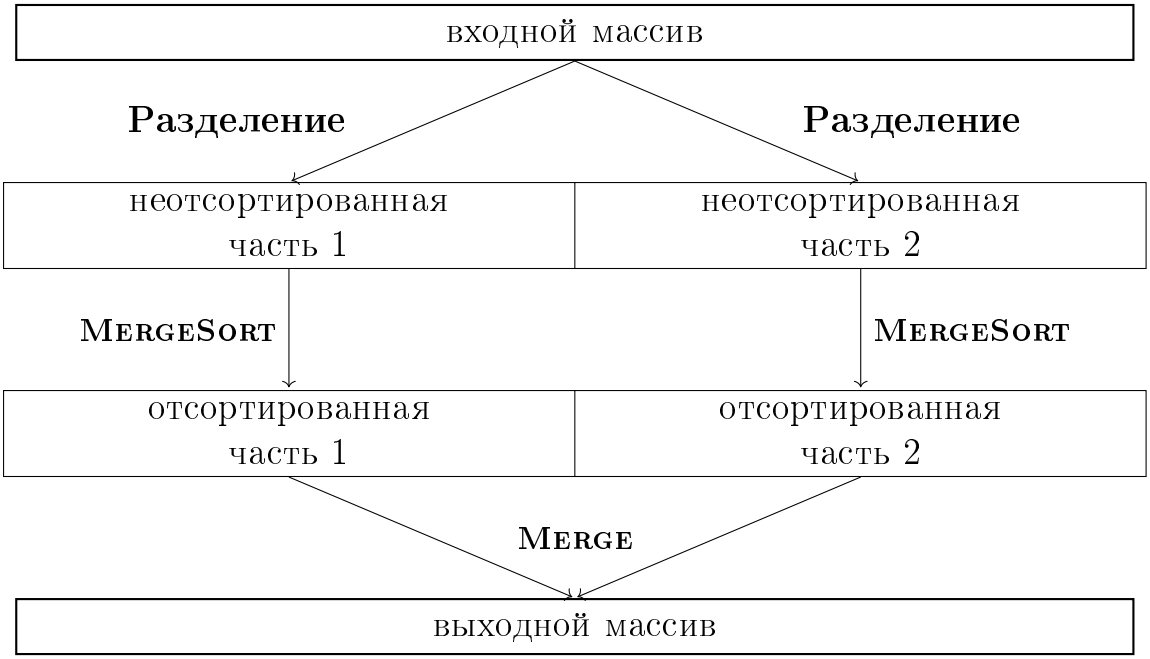


Рис. 4.6: Схема алгоритма сортировки слиянием.

Алгоритм 4.15: HeapSort: пирамидальная сортировки.

Вход: Массив A из n элементов
Результат: Изменение порядка элементов массива A.

```
// Первый этап: построить кучу по массиву A (алгоритм 3.9)
1 MakeHeap(A, n);
// Второй этап: формирование результата
2 k ← n − 1;
3 while k ⩾ 0 do
4 |   Поменять A[0] и A[i];
5 |   HeapSink(A, n, 0);
6 |   k ← k − 1;
7 end while
```

4.3.6 Быстрая сортировка

Алгоритм быстрой сортировки является одним из распространенных вариантов сортировки. Именно этот алгоритм реализован в функции qsort стандартной библиотеки языка Си.

Идея алгоритма состоит в следующем[4]. Предположим, что мы выбрали значение $\alpha \in \mathbb{R}$ таким образом, что в массиве есть по крайней мере один элемент со значением не превосходящим α. Тогда массив можно разделить на две части: в левой части будут элементы с «маленькими» значениями ($\leqslant \alpha$), в

[4]Для определенности будем предполагать, что мы сортируем массив действительных чисел, но алгоритм без изменений может быть применен для сортировки элементов любого линейно упорядоченного множества.

Алгоритм 4.16: QSSplit: разделение массива по сечению α.

Вход: Массив A, индексы $left$ и $right$ ($left \leqslant right$), α.
Выход: Индекс k, такой что $A[i] \leqslant \alpha$ при $i \leqslant k$ и $A[i] > \alpha$ при $i > k$.
Результат: Перестановка элементов $A[left], A[left+1], \ldots, A[right]$.

```
1  i ← left;
2  j ← right;
3  while i < j do
       // Найдем первый слева элемент ⩾ α
4      while A[i] < α и i < j do
5          i ← i + 1;
6      end while
       // Найдем первый справа элемент ⩽ α
7      while A[j] > α и i < j do
8          j ← j − 1;
9      end while
10     if i < j then
11         Меняем местами A[i] и A[j];
12         j ← j − 1;
13     end if
14 end while
15 return i;
```

правой — с большими ($> \alpha$). *Границей* назовем такой индекс b, что $A[i] \leqslant \alpha$ при всех $i \leqslant b$ и $A[i] > \alpha$ при $i > b$. Теперь заметим, что части массива можно сортировать независимо друг от друга: если после разделения массива по значению α какой-то элемент имеет индекс $k \leqslant b$, то после сортировки всего массива этот не может оказаться в позиции $k' > b$. Верно и обратное утверждение. Если индекс элемента больше b, то и после сортировки он будет больше b. Это приводит к естественному рекурсивному алгоритму сортировки (рис. 4.17).

Разделение массива по значению α выполняется обменами некоторых элементов. Алгоритм находит самое левое значение, которое не меньше α, и самое правое значение, которое не больше α. Эти элементы обмениваются и производится поиск следующей пары значений для обмена. Текст алгоритма разделения массива приведен на рис. 4.16.

Выбор порогового значения должен проводится таким образом, чтобы разделять массив на две приблизительно равные по длине части. Такое условие нельзя гарантировать. Обычно используются следующие варианты α: значение первого элемента сортируемой части массива ($A[left]$), значение среднего элемента, полусумма первого и последнего, медианное значение из трех (первого, последнего и среднего), значение случайно выбранного эле-

Алгоритм 4.17: QuickSort: быстрая сортировка.

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 if
5  |  return;
6  end if
7  α ← (A[left] + A[right])/2; // Выберем сечение α
8  border ← QSSplit(A, left, right, α);
   // Рекурсивные вызовы: сортируем части массива
9  QuickSort(A, left, border); // левую
10 QuickSort(A, border + 1, right); // правую
```

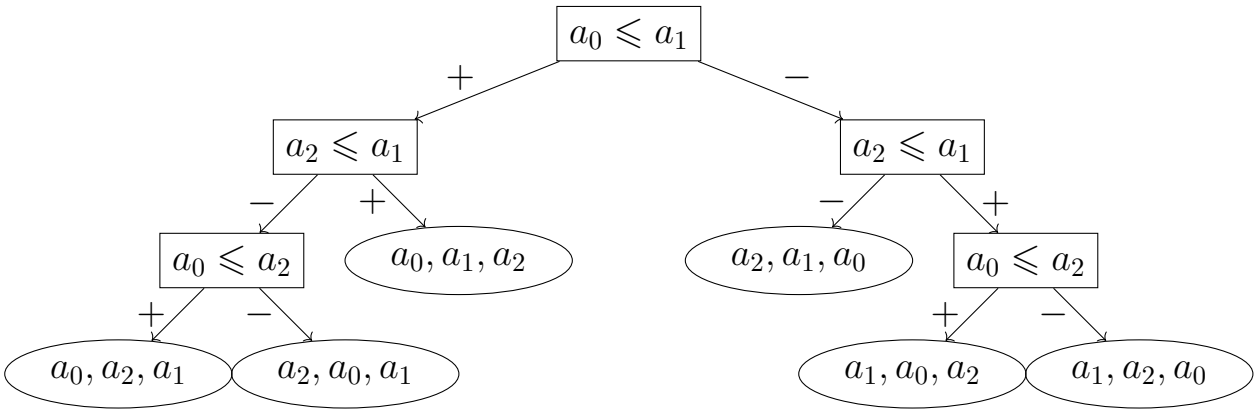


Рис. 4.7: Пример дерева решений для сортировки массива из трех элементов.

мента.

4.3.7 Нижняя оценка сложности сортировки на основе сравнения ключей

В предыдущих разделах ма рассмотрели несколько алгоритмов сортировки. Простые алгоритмы (пузырьковая сортировка, сортировка вставками) имеют сложность $O(n^2)$. Пирамидальная сортировка и сортировка слиянием работают быстрее — при сортировке массива из n элементов они гарантируют получение ответа за $O(n \log_2 n)$. Можно ли отсортировать массив еще быстрее? Возможно, что мы пока просто не нашли лучшего алгоритма. Следующая теорема говорит, что таких алгоритмов не существует.

Теорема 5. *Не существует алгоритма сортировки на основе сравнения 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 массива из трех элементов изображен на рис. 4.7. 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.

4.3.8 Сортировка за линейное время

Теорема 5 утверждает, что не может быть алгоритмов сортировки, работающих быстрее, чем за $O(n \log_2 n)$. Но это верно для произвольных массивов, и алгоритмов, которые значения ключей. Если же 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).

Поразрядная сортировка.

4.4 Алгоритмы обработки строк

4.4.1 Строки и базовые операции с ними

4.4.2 Поиск подслова

Задача поиска подслова.

Наивный алгоритм поиска.

Конечные автоматы.

4.4.3 Суффиксные деревья

4.4.4 Нечеткое сопоставление с образцом

4.4.5 Алгоритмы архивирования

4.5 Задачи и упражнения

4.1 Модифицируйте алгоритм QSSplit (алг. 4.16 на стр. 72) разделения массива относительно заданного значения таким образом, чтобы в нем вместо обменов (в строке 11) использовались пересылки.

Глава 5

Алгоритмические языки

Алгоритмическим языком называется формальный язык, который может использоваться для определения алгоритмов. Формальность языка означает, что для него строго определен синтаксис (всегда можно сказать, образует ли последовательность символов допустимую программу) и семантика (недвусмысленное описание того, как следует понимать ту или иную запись). *Язык программирования* — это формальный язык, предназначенный для реализации алгоритмов. Как правило, языки программирования являются алгоритмическими языками. В англоязычной литературе понятия алгоритмический язык и язык программирования часто используются как синонимы. В этой главе рассматривается несколько известных языков программирования, на примере которых показываются различные подходы к определению вычислительных процессов (алгоритмов).

Языки программирования можно разделить на два крупных класса: машинные и высокоуровневые. Машинные языки связаны с системой команд конкретного процессора. Рассмотренный в главе 2 язык ассемблера — пример машинного языка (для вымышленной вычислительной машины). Языки высокого уровня, напротив, разрабатываются с учетом возможности реализации на различных вычислительных устройствах. Как правило, машинные языки не считают алгоритмическими, а подавляющее число языков программирования является языками высокого или сверхвысокого уровня.

5.1 Классификация языков программирования

Довольно сложно предложить однозначную классификацию языков программирования, так как реальные языки включают элементы из разных групп. К числу основных характеристик язков программирования можно отнести следующие.

- Низкоурвневые, высокоуровневые и сверхвысокоуровневые
- Универсальные или специализированные

- Статическая или динамическая типизация
- Компилируемые или интерпретируемые
- Декларативные или императивные
- Процедурные, функциональные, логические, объектно-ориентированные
- С автоматическим или ручным управлением памятью

5.2 Процедурное программирование: язык C

В качестве нашего первого примера рассмотрим язык C. Он является основой для целого семейства C-подобных языков, которое включает такие распространенные языки программирования как C, C++, C#, Objective C, Java.

Язык программирования C — это язык высокого уровня, но по возможности управления аппаратной частью компьютера мало в чем уступает ассемблеру.

5.2.1 Простейшая програма на C

Рассмотрим в качестве примера достаточно простую программу на языке C, которая вычисляет факториал заданного числа. Текст программы приведен на рис. 5.1. Если эту программу *откомпилиовать* и *запустить* (о том, что это означают эти слова будет рассказано чуть позже), то она напечатает строчку `10! = 3628800`. Рассмотрим на примере этой программы такие элементы языка C как обявление перемнных, объявление функций, циклы, функция печати.

```
1  #include <stdio.h>
2
3  int main(void) {
4    int n = 10; /* вычисляем n! */
5    int factorial = 1; // здесь будет записан ответ
6
7    for(int k = 1; k <= n; k++) {
8      factorial = factorial * k;
9    }
10   printf("%d! = %d\n", n, factorial);
11
12   return 0;
13 }
```

Рис. 5.1: Программа вычисления факториала на языке C.

Сначала отметим, что каждая программа на C обязательно должна реализовать функцию с именем main. При запуске программы её выполнение начинается с вызова функции main.

```
#include <stdio.h>

int factorial(int n); /* compute factroial of n */

int main(void) {
  int n = 10; /* compute n! */
  printf("%d! = %d\n", n, factorial(n));

  return 0;
}

int factorial(int n) {
  if(n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

Рис. 5.2: Программа рекурсивного вычисления факториала на языке C.

5.2.2 Объявления переменных

5.2.3 Управляющие конструкции цикла

В языке C есть три оператора цикла: while, do/while и for.

```
for(int k = 0; k < n; k++) {
  printf("k = %d\n", k);
}
```

5.3 Функциональное программирование: LISP

Язык программирования Лисп является компилируемым функциональным языком программирования с автоматическим управлением памятью и динамической типизацией с автоматическим выводом типов. Строго говоря, следует говорить о семействе языков типа Лисп. В данном разделе описываются основные конструкции диалекта, который называется Common Lisp. Изложение не претендует на полноту.

5.3.1 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			

5.3.2 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

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

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.

```
(dotimes (k 10)
  (print k))
```

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)`.

5.3.3 Объявление функций

5.3.4 Работа со списками

5.4 Объектно-ориентированное программирование: C++

5.5 Почему языки такие разные

Глава 6

Подсказки и решения

Для того, чтобы усовершенствовать ум,
надо больше размышлять, чем заучивать.

Кто берется давать наставления, должен
считать себя искуснее тех, кому он их дает:
малейшая его погрешность заслуживает
порицания.

Рене Декарт (1596 –1650)
Французский математик

Если вы не смогли решить задачу, то перед тем как просто посмотреть ее решение прочитайте подсказку и еще раз пострайтесь решить задачу. Подсказки приведены в разделе 6.1, решения — в разделе 6.2. Если ваше решение не совпадает с приведенным в этой главе, то это еще не означает, что оно неправильное. Существует множество вариантов решения одной задачи.

6.1 Подсказки

6.1.1 Элементы теории алгоритмов

Задача 1.8. Рассмотрите случай, когда A не содержит заключительных правил. К каждому правилу припишите его номер.

Задача 1.12. Оцените число шагов, после которого можно с уверенностью сказать, что значение $A(w)$ не определено.

Задача 1.13. Сведите задачу к многократному вычитанию.

6.1.2 Программы на ассемблере

6.1.3 Структуры данных

6.1.4 Алгоритмы

6.2 Решения задач

6.2.1 Элементы теории алгоритмов

Задача 1.8.

Пусть алгоритм A не содержит заключительных правил. Обозначим эти правила $u_i \to v_i$, где $i \leqslant 5$. Введем новый алфавит $D = \{1, 2, 3, 4, 5\}$, такой что $D \cap \Sigma = \varnothing$, и составим алгоритм A' в алфавите $\Sigma' = \Sigma \cup D$ следующим образом. Добавим в правую часть каждого правила алгоритма A его порядковый номер. Тогда применение правила приведет к появлению его номера в слове. Добавим правила, которые выполняются до правил алгоритма A и переводят номер правила в конец слова. Теперь, если слово заканчивается на два элемента D, то удалим правую цифру — это номер предыдущего примененного правила. Таким образом, в слове остается только один номер — номер последнего примененного правила. После окончания работы A, когда ни одно из правил A не может быть применено, применяются правила, записанные последними. Они просто удаляют все буквы алфавита Σ (выходное слово алгоритма A) и завершают работу алгоритма A'.

Приведенный ниже алгоритм содержит несколько *метаправил*, которые раскрываются в большое число «обычных» правил. Например, $\mathbf{an} \to \mathbf{n}$ определяет $|\Sigma| \times |D|$ правил, где $\mathbf{a}$ заменяется на букву алфавита Σ, а $\mathbf{n}$ — на элемент D. Метасимволы выделены жирным шрифтом.

$\mathbf{na}$	$\to$	$\mathbf{an}$	*переводим цифры вправо:* $\boldsymbol{n} \in D$, $\boldsymbol{a} \in \Sigma$
$\mathbf{nk}$	$\to$	$\mathbf{n}$	*удаляем правую цифру из двух:* $\boldsymbol{n}, \boldsymbol{k} \in D$
u_1	$\to$	$v_1 1$	
u_2	$\to$	$v_2 2$	
u_3	$\to$	$v_3 3$	
u_4	$\to$	$v_4 4$	
u_5	$\to$	$v_5 5$	
$\mathbf{an}$	$\to$	$\mathbf{n}$	*удаляем буквы левее цифры:* $\boldsymbol{n} \in D$, $\boldsymbol{a} \in \Sigma$
$\mathbf{n}$	$\to.$	$\mathbf{n}$	*окончание работы (нужно, если* $u_i = \varepsilon$ *для некоторого* i*)*

Корректность этого преобразования следует из того, что если в A было правило, применимое к некоторому слову w, то это правило применимо и к слову с приписанными справа цифрами. Причем позиции первого вхождения левой части правила совпадают.

Если же алгоритм A содержит заключительные правила, то приведенный выше алгоритм A' нужно модифицировать. Правила $u_i \to v_i$i остаются в таком же виде. Если в алгоритме A правило с индексом i было заключительным, то точка удаляется, а правила из последних двух строк переносятся выше $u_1 \to v_1 1$ (если сработало заключительное правило, то нужно закончить работу алгоритма A').

Заметим, что если первое метоправило записать в виде **an** $\to$ **na** и соответсвующим образом изменить остальные метаправила, чтобы номер последнего правила сохранялся в начале, а не в конце слова, то A' будет корректно работать только в том случае, когда ни одно из слов u_i не совпадает с пустым словом.

Задача 1.11. Решение разбивается на следующие этапы.

- Сначала (для удобства) заменяем все буквы b на a. Длина слова при этом не изменится.
- Полученное слово $aa\ldots a$, состоящее теперь из одних букв a, приведем к виду $0\#aa\ldots a$. Слева от решетки будет строиться запись двоичного представления длины слова. Старшие разряды будут ближе к началу слова.
- Удаляем по одной букве справа от решетки и одновременно добавляем единицу к представлению числа. За это отвечают правила с символом $+$.
- Когда все буквы обработаны, остается только удалить символ решетки.

b	$\to$	a	*приведем слово к однобуквенному алфавиту*
$1+$	$\to$	$+0$	*сохраняем перенос разряда*
$0+$	$\to$	1	*увеличили разряд без переноса; +1 закончено*
$+$	$\to$	1	*число кончилось; нужно увеличить разрядность числа*
#a	$\to$	$+\#$	*убрали одну букву* a, *пометили, что нужно прибавить* 1
#	$\to$	. ε	
ε	$\to$	$0\#$	*записали нулевую длину, начинаем считать*

Задача 1.12. Такой алгоритм существует. При обработке алгоритмом A входного слова w «возникают» слова $w_0 = w, w_1, w_2, \ldots$. Если $A(w)$ определено, то последовательность $\{w_i\}_{i\geqslant 0}$ конечна. По условию задачи $|w_i| \leqslant f(|w_0|)$ для любого $i > 0$. Это означает, что искомый алгоритм проверки может запомнить все различные промежуточные слова, которые возникают при работе алгоритма A на входном слове w. Если какое-либо слово встречается в последовательности $\{w_i\}_{i\geqslant 0}$ более одного раза и при этом не является последним, то есть $w_p = w_q$ для некоторых различных p и q и слово w_q было получено применением нетерминального правила, то алгоритм A не закончит работу. Цепочка слов $w_p, w_{p+1}, \ldots, w_{q-1}$ образует цикл. Если алгоритм закончил работу, то число шагов не превосходит $f(|w_0|) + 1$.

Эти рассуждения можно сформулировать в виде следующего (конструктивного) утверждения. Если $A(w)$ определено, то ответ получается не более чем за $f(|w_0|) + 1$ шагов.

Задача 1.13. Идея решения построена на многократном вычитании правой части слова из левой. Весь процесс можно разбить на *этапы*, которые заканчиваются словами вида $a^n\#a^m$, $a^{n-m}\#a^m$, $a^{n-2m}\#a^m$ и так далее, пока $n-km \geqslant m$. Основная проблема в том, что вычитание нужно делать по одной букве. Для этого в правую часть слова поставим маркер $\|$, отмечающий число единиц, которое на текущем этапе уже было удалено из левой части слова. Синхронное удаление букв слева и справа достигается сдвигом маркера $\|$ на одну позицию вправо («виртуальное» удаление буквы) и вставку маркера $\langle$, который должен удалить букву из левой части слова. Когда справа от $\|$ нет больше букв, этап алгоритма заканчивается и $\|$ удаляется из слова.

Если слева от символа # не осталось букв, то ответ можно получить, приписав одну букву a к тому слову, которое осталось справа от $\|$. Действительно, в этот момент исходное слово $a^n\#a^m$ будет преобразовано к виду $\#a^p\|a^q$, то есть левее решетки букв a нет и на последнем этапе было удалено $p-1 \geqslant 0$ букв a слева. Последняя попытка удаления буквы a не была доведена до конца: маркер $\|$ сдвинулся на одну позицию вправо, а $\langle$ уже не смог удалить соответствующую букву слева.

$a\langle$	$\to$	$\langle a$	
$a\#\langle$	$\to$	$\#$	*маркер $\langle$ дошел до # и удалил одину букву слева*
Ra	$\to$	R	*удаляем лишнее*
$R\|$	$\to.$	a	*«прибавили» единицу к числу справа от $\|$*
$\#\langle$	$\to$	R	*маркер $\langle$ дошел до #, но букв нет; завершающая часть*
$\|a$	$\to$	$\langle a\|$	*«вычитаем» единицу из m (сдвиг $\|$) и вставляем $\langle$*
$\|$	$\to$	ε	*удалим $\|$, если он дошел до конца*
$\#$	$\to$	$\#\|$	*добавили маркер количества вычитаний*

Рассмотрите работу этого алгоритма на входном слове *aaa#aa*.

6.2.2 Программы на ассемблере

6.2.3 Структуры данных

Задача 3.5.

6.2.4 Алгоритмы

Задача 4.1.

Алгоритм добавления элемента к куче.

Вход: Массив A, число m. Первые m элементов A образуют пирамиду.

Выход: Изменение массива A: первые $m+1$ элементов образуют пирамиду.

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

Быстрое разделение массива по сечению α.

Вход: Массив A, индексы $left$ и $right$ ($left \leqslant right$), α.
Выход: Индекс k, такой что $A[i] \leqslant \alpha$ при $i \leqslant k$ и $A[i] > \alpha$ при $i > k$.
Результат: Перестановка элементов $A[left], A[left+1], \ldots, A[right]$.

```
1  i ← left;
2  j ← right;
3  while A[i] ⩽ α u i < j do      // Найдем первый слева элемент > α
4  |  i ← i + 1;
5  end while
6  t ← A[i];
7  empty ← i;
8  while i < j do
   |  // Найдем первый справа элемент ⩽ α
9  |  while A[j] > α u i ⩽ j do
10 |  |  j ← j − 1;
11 |  end while
12 |  A[empty] ← A[j];
13 |  empty ← j;
14 |  j ← j − 1;
   |  // Найдем следующий слева элемент > α
15 |  while A[i] ⩽ α u i < j do
16 |  |  i ← i + 1;
17 |  end while
18 |  A[empty] ← A[i];
19 |  empty ← i;
20 end while
21 A[empty] ← t;
22 return i;
```

Предметный указатель