

## ГЛАВА 3

# Деревья

Деревья представляют собой иерархическую структуру некой совокупности элементов. Знакомыми вам примерами деревьев могут служить генеалогические и организационные диаграммы. Деревья используются при анализе электрических цепей, при представлении структур математических формул. Они также естественным путем возникают во многих областях компьютерных наук. Например, деревья используются для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ. В главе 5 описано применение деревьев для представления данных. В этой книге вы встретитесь со многими примерами деревьев. В этой главе мы введем основные определения (терминологию) и представим основные операторы, выполняемые над деревьями. Затем опишем некоторые наиболее часто используемые структуры данных, представляющих деревья, и покажем, как наиболее эффективно реализовать операторы деревьев.

### 3.1. Основная терминология

Дерево — это совокупность элементов, называемых *узлами* (один из которых определен как *корень*), и отношений (“родительских”), образующих иерархическую структуру узлов. Узлы, так же, как и элементы списков, могут быть элементами любого типа. Мы часто будем изображать узлы буквами, строками или числами. Формально *дерево* можно рекуррентно определить следующим образом.

1. Один узел является деревом. Этот же узел также является корнем этого дерева.
2. Пусть  $n$  — это узел, а  $T_1, T_2, \dots, T_k$  — деревья с корнями  $n_1, n_2, \dots, n_k$  соответственно. Можно построить новое дерево, сделав  $n$  родителем узлов  $n_1, n_2, \dots, n_k$ . В этом дереве  $n$  будет корнем, а  $T_1, T_2, \dots, T_k$  — *поддеревьями* этого корня. Узлы  $n_1, n_2, \dots, n_k$  называются *сыновьями* узла  $n$ .

Часто в это определение включают понятие *нулевого дерева*, т.е. “дерева” без узлов, такое дерево мы будем обозначать символом  $\Lambda$ .

**Пример 3.1.** Рассмотрим оглавление книги, схематически представленное на рис. 3.1, а. Это оглавление является деревом, которое в другой форме показано на рис. 3.1, б. Отношение родитель-сын отображается в виде линии. Деревья обычно рисуются сверху вниз, как на рис. 3.1, б, так, что родители располагаются выше “детей”.

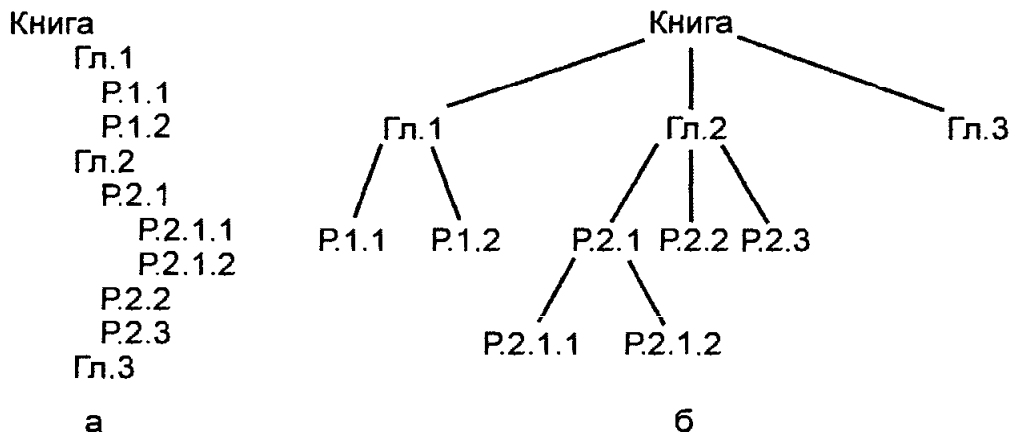


Рис. 3.1. Оглавление книги и его представление в виде дерева

Корнем этого дерева является узел *Книга*, который имеет три поддерева соответственно с корнями *Гл.1*, *Гл.2* и *Гл.3*. Эти отношения показаны линиями, идущими из корня *Книга* к узлам *Гл.1*, *Гл.2* и *Гл.3*. Узел *Книга* является родителем узлов *Гл.1*, *Гл.2* и *Гл.3*, а эти три узла — сыновьями узла *Книга*.

Третье поддерево, с корнем *Гл.3*, состоит из одного узла, остальные два поддерева имеют нетривиальную структуру. Например, поддерево с корнем *Гл.2* в свою очередь имеет три поддерева, которые соответствуют разделам книги *P.2.1*, *P.2.2* и *P.2.3*; последние два поддерева имеют по одному узлу, в то время как первое имеет два поддерева, соответствующие подразделам книги *P.2.1.1* и *P.2.1.2*. □

Пример 3.1 показывает типичные данные, для которых наилучшим представлением будут деревья. В этом примере родительские отношения устанавливают подчиненность глав и разделов: родительский узел связан с узлами сыновей (и указывает на них), как узел *Книга* подчиняет себе узлы *Гл.1*, *Гл.2* и *Гл.3*. В этой книге вы встретите много различных отношений, которые можно представить с помощью родительских отношений и подчиненности в виде деревьев.

Путем из узла  $n_1$  в узел  $n_k$  называется последовательность узлов  $n_1, n_2, \dots, n_k$ , где для всех  $i, 1 \leq i \leq k$ , узел  $n_i$  является родителем узла  $n_{i+1}$ . Длиной пути называется число, на единицу меньшее числа узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе. На рис. 3.1 путем длины 2 будет, например, путь от узла *Гл.2* к узлу *P.2.1.2*.

Если существует путь из узла  $a$  в  $b$ , то в этом случае узел  $a$  называется *предком* узла  $b$ , а узел  $b$  — *потомком* узла  $a$ . Например, на рис. 3.1 предками узла *P.2.1* будут следующие узлы: сам узел *P.2.1* и узлы *Гл.2* и *Книга*, тогда как потомками этого узла являются опять сам узел *P.2.1* и узлы *P.2.1.1* и *P.2.1.2*. Отметим, что любой узел одновременно является и предком, и потомком самого себя.

Предок или потомок узла, не являющийся таковым самого себя, называется *истинным предком* или *истинным потомком* соответственно. В дереве только корень не имеет истинного предка. Узел, не имеющий истинных потомков, называется *листом*. Теперь поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

*Высотой узла* дерева называется длина самого длинного пути из этого узла до какого-либо листа. На рис.3.1 высота узла *Гл.1* равна 1, узла *Гл.2* — 2, а узла *Гл.3* — 0. *Высота дерева* совпадает с высотой корня. *Глубина узла* определяется как длина пути (он единственный) от корня до этого узла.

## Порядок узлов

Сыновья узла обычно упорядочиваются слева направо. Поэтому два дерева на рис. 3.2 различны, так как порядок сыновей узла  $a$  различен. Если порядок сыновей игнорируется, то такое дерево называется *неупорядоченным*, в противном случае дерево называется *упорядоченным*.

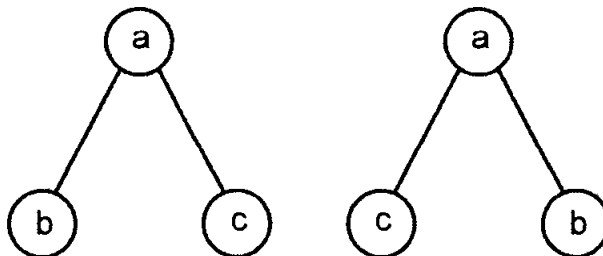


Рис. 3.2. Два различных упорядоченных дерева

Упорядочивание слева направо *сыновей* (“родных детей” одного узла) можно использовать для сопоставления узлов, которые не связаны отношениями предки-потомки. Соответствующее правило звучит следующим образом: если узлы  $a$  и  $b$  яв-

ляются сыновьями одного родителя и узел  $a$  лежит слева от узла  $b$ , то все потомки узла  $a$  будут находиться слева от любых потомков узла  $b$ .

**Пример 3.2.** Рассмотрим дерево на рис. 3.3. Узел 8 расположен справа от узла 2, слева от узлов 9, 6, 10, 4 и 7, и не имеет отношений справа-слева относительно предков 1, 3 и 5.

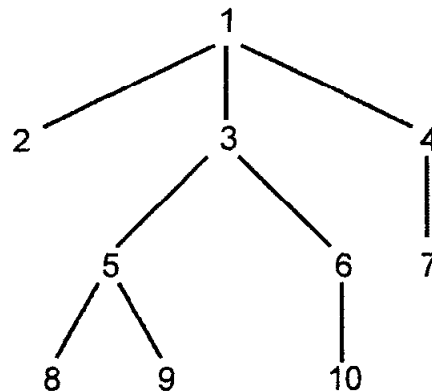


Рис. 3.3. Дерево

Существует простое правило, позволяющее определить, какие узлы расположены слева от данного узла  $n$ , а какие — справа. Для этого надо прочертить путь от корня дерева до узла  $n$ . Тогда все узлы и их потомки, расположенные слева от этого пути, будут находиться слева от узла  $n$ , и, аналогично, все узлы и их потомки, расположенные справа от этого пути, будут находиться справа от узла  $n$ .  $\square$

## Прямой, обратный и симметричный обходы дерева

Существует несколько способов обхода (прохождения) всех узлов дерева<sup>1</sup>. Три наиболее часто используемых способа обхода дерева называются *обход в прямом порядке*, *обход в обратном порядке* и *обход во внутреннем порядке* (последний вид обхода также часто называют *симметричным обходом*, мы будем использовать оба этих названия как синонимы). Все три способа обхода рекурсивно можно определить следующим образом.

- Если дерево  $T$  является нулевым деревом, то в список обхода заносится пустая запись.
- Если дерево  $T$  состоит из одного узла, то в список обхода записывается этот узел.
- Далее, пусть  $T$  — дерево с корнем  $n$  и поддеревьями  $T_1, T_2, \dots, T_k$ , как показано на рис. 3.4. Тогда для различных способов обхода имеем следующее.

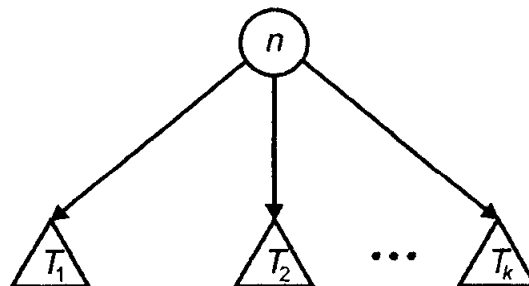


Рис. 3.4. Дерево  $T$

<sup>1</sup> Обход узлов дерева равнозначен упорядочиванию по какому-либо правилу этих узлов. Поэтому в данном разделе мы будем использовать слова “обход узлов” и “упорядочивание узлов” как синонимы. — *Прим. ред.*

1. При *прохождении в прямом порядке* (т.е. при *прямом упорядочивании*) узлов дерева  $T$  сначала посещается корень  $n$ , затем узлы поддерева  $T_1$ , далее все узлы поддерева  $T_2$ , и т.д. Последними посещаются узлы поддерева  $T_k$ .
2. При *симметричном обходе* узлов дерева  $T$  сначала посещаются в симметричном порядке все узлы поддерева  $T_1$ , далее корень  $n$ , затем последовательно в симметричном порядке все узлы поддеревьев  $T_2, \dots, T_k$ .
3. Во время *обхода в обратном порядке* сначала посещаются в обратном порядке все узлы поддерева  $T_1$ , затем последовательно посещаются все узлы поддеревьев  $T_2, \dots, T_k$ , также в обратном порядке, последним посещается корень  $n$ .

В листинге 3.1,а показан набросок процедуры PREORDER (Прямое упорядочивание), составляющей список узлов дерева при обходе его в прямом порядке. Чтобы из этой процедуры сделать процедуру, выполняющую обход дерева в обратном порядке, надо просто поменять местами строки (1) и (2). В листинге 3.1,б представлен набросок процедуры INORDER (Внутреннее упорядочивание). В этих процедурах производится соответствующее упорядочивание деревьев путем вызова соответствующих процедур к корню дерева.

### Листинг 3.1. Рекурсивные процедуры обхода деревьев

#### а. Процедура PREORDER

```

procedure PREORDER (  $n$ : узел );
  begin
    (1)      занести в список обхода узел  $n$ ;
    (2)      for для каждого сына  $c$  узла  $n$  в порядке слева направо do
              PREORDER( $c$ )
  end; { PREORDER }

```

#### б. Процедура INORDER

```

procedure INORDER (  $n$ : узел );
  begin
    if  $n$  — лист then
      занести в список обхода узел  $n$ ;
    else begin
      INORDER(самый левый сын узла  $n$ );
      занести в список обхода узел  $n$ ;
      for для каждого сына  $c$  узла  $n$ , исключая самый левый,
        в порядке слева направо do
        INORDER( $c$ )
    end
  end; { INORDER }

```

**Пример 3.3.** Сделаем обход дерева, показанного на рис. 3.3, в прямом порядке. Сначала запишем (посетим) узел 1, затем вызовем процедуру PREORDER для обхода первого поддерева с корнем 2. Это поддерево состоит из одного узла, которое мы и записываем. Далее обходим второе поддерево, выходящее из корня 1, это поддерево имеет корень 3. Записываем узел 3 и вызываем процедуру PREORDER для обхода первого поддерева, выходящего из узла 3. В результате получим список узлов 5, 8 и 9 (именно в таком порядке). Продолжая этот процесс, в конце мы получим полный список узлов, посещаемых при прохождении в прямом порядке исходного дерева: 1, 2, 3, 5, 8, 9, 6, 10, 4 и 7.

Подобным образом, предварительно преобразовав процедуру PREORDER в процедуру, выполняющую обход в обратном порядке (как указано выше), можно получить обратное упорядочивание узлов дерева из рис. 3.3 в следующем виде: 2, 8, 9, 5, 10,

6, 3, 7, 4 и 1. Применяя процедуру INORDER, получим список симметрично упорядоченных узлов этого же дерева: 2, 1, 8, 5, 9, 3, 10, 6, 7 и 4.

При обходе деревьев можно применить следующий полезный прием. Надо нарисовать непрерывный контур вокруг дерева, начиная от корня дерева, рисуя контур против часовой стрелки и поочередно обходя все наружные части дерева. Такой контур вокруг дерева из рис. 3.3 показан на рис. 3.5.

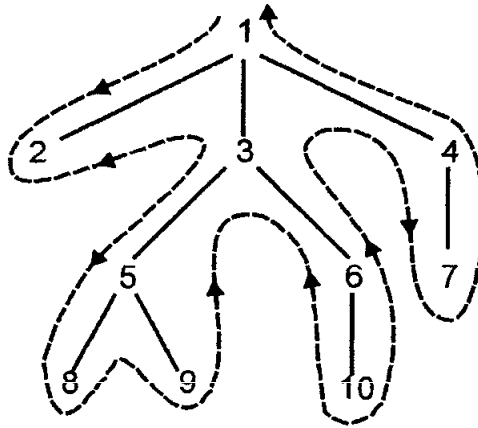


Рис. 3.5. Контур дерева

При прямом упорядочивании узлов надо просто записать их в соответствии с нарисованным контуром. При обратном упорядочивании после записи всех сыновей переходим к их родителю. При симметричном (внутреннем) упорядочивании после записи самого правого листа мы переходим не по ветви в направлении к корню дерева, а к следующему “внутреннему” узлу, который еще не записан. Например, если на рис. 3.5 узлы 2 и 1 уже записаны, то мы как бы перескакиваем “залив” между узлами 2 и 3 и переходим к узлу 8. Отметим, что при любом упорядочивании листьев всегда записываются в порядке слева направо, при этом в случае симметричного упорядочивания между сыновьями может быть записан родитель. □

## Помеченные деревья и деревья выражений

Часто бывает полезным сопоставить каждому узлу дерева *метку* (label) или значение, точно так же, как мы в предыдущей главе сопоставляли элементам списков определенные значения. Дерево, у которого узлам сопоставлены метки, называется *помеченным деревом*. Метка узла — это не имя узла, а значение, которое “хранится” в узле. В некоторых приложениях мы даже будем изменять значение метки, поскольку имя узла сохраняется постоянным. Полезна следующая аналогия: дерево — список, узел — позиция, метка — элемент.

**Пример 3.4.** На рис. 3.6 показано дерево с метками, представляющее арифметическое выражение  $(a + b) * (a + c)$ , где  $n_1, \dots, n_7$  — имена узлов (метки на рисунке представлены рядом с соответствующими узлами). Правила соответствия меток деревьев элементам выражений следующие.

1. Метка каждого листа соответствует операнду и содержит его значение, например узел  $n_4$  представляет операнд  $a$ .
2. Метка каждого внутреннего (родительского) узла соответствует оператору. Предположим, что узел  $n$  помечен бинарным оператором  $\theta$  (например,  $+$  или  $*$ ) и левый сын этого узла соответствует выражению  $E_1$ , а правый — выражению  $E_2$ . Тогда узел  $n$  и его сыновья представляют выражение  $(E_1) \theta (E_2)$ . Можно удалять родителей, если это необходимо.

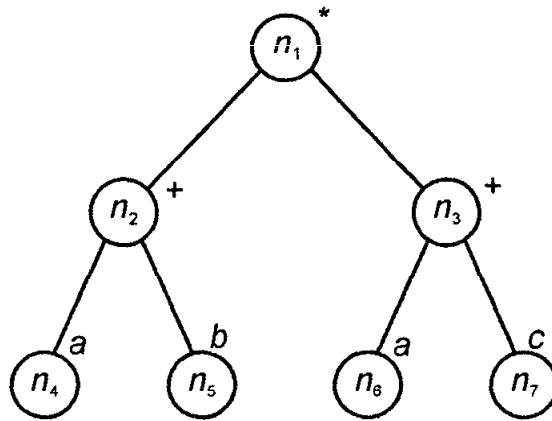


Рис. 3.6. Дерево выражения с метками

Например, узел  $n_2$  имеет оператор  $+$ , а левый и правый сыновья представляют выражения (операнды)  $a$  и  $b$  соответственно. Поэтому узел  $n_2$  представляет  $(a) + (b)$ , т.е.  $a + b$ . Узел  $n_1$  представляет выражение  $(a + b) * (a + c)$ , поскольку оператор  $*$  является меткой узла  $n_1$ , выражения  $a + b$  и  $a + c$  представляются узлами  $n_2$  и  $n_3$  соответственно.  $\square$

Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом упорядочивании получаем известную *префиксную форму* выражений, где оператор предшествует и левому, и правому операндам. Для точного описания префиксной формы выражений сначала положим, что префиксным выражением одиночного операнда  $a$  является сам этот операнд. Далее, префиксная форма для выражения  $(E_1) \theta (E_2)$ , где  $\theta$  — бинарный оператор, имеет вид  $\theta P_1 P_2$ , здесь  $P_1$  и  $P_2$  — префиксные формы для выражений  $E_1$  и  $E_2$ . Отметим, что в префиксных формах нет необходимости отделять или выделять отдельные префиксные выражения скобками, так как всегда можно просмотреть префиксное выражение  $\theta P_1 P_2$  и определить единственным образом  $P_1$  как самый короткий префикс выражения  $P_1 P_2$ .

Например, при прямом упорядочивании узлов (точнее, меток) дерева, показанного на рис. 3.6, получаем префиксное выражение  $*+ab+ac$ . Самым коротким корректным префиксом для выражения  $+ab+ac$  будет префиксное выражение узла  $n_2$ :  $+ab$ .

Обратное упорядочивание меток дерева выражений дает так называемое *постфиксное* (или *польское*) *представление* выражений. Выражение  $\theta P_1 P_2$  в постфиксной форме имеет вид  $P_1 P_2 \theta$ , где  $P_1$  и  $P_2$  — постфиксные формы для выражений  $E_1$  и  $E_2$  соответственно. При использовании постфиксной формы также нет необходимости в применении скобок, поскольку для любого постфиксного выражения  $P_1 P_2$  легко проследить самый короткий суффикс  $P_2$ , что и будет корректным составляющим постфиксным выражением. Например, постфиксная форма выражения для дерева на рис. 3.5 имеет вид  $ab+ac+*$ . Если записать это выражение как  $P_1 P_2 *$ , то  $P_2$  (т.е. выражение  $ac+$ ) будет самым коротким суффиксом для  $ab+ac+$  и, следовательно, корректным составляющим постфиксным выражением.

При симметричном обходе дерева выражений получим так называемую *инфиксную форму* выражения, которая совпадает с привычной “стандартной” формой записи выражений, но также не использует скобок. Для дерева на рис. 3.6 инфиксное выражение запишется как  $a + b * a + c$ . Читателю предлагается разработать алгоритм обхода дерева выражений, который бы выдавал инфиксную форму выражения со всеми необходимыми парами скобок.

## Вычисление „наследственных“ данных

Обход дерева в прямом или обратном порядке позволяет получить данные об отношениях предок–потомок узлов дерева. Пусть функция  $postorder(n)$  вычисляет позицию узла  $n$  в списке узлов, упорядоченных в обратном порядке. Например, для уз-

лов  $n_2$ ,  $n_4$  и  $n_5$  дерева, представленного на рис. 3.6, значения этой функции будут 3, 1 и 2 соответственно. Определим также функцию  $desc(n)$ , значение которой равно числу истинных потомков узла  $n$ .

Эти функции позволяют выполнить ряд полезных вычислений. Например, все узлы поддерева с корнем  $n$  будут последовательно занимать позиции от  $postorder(n) - desc(n)$  до  $postorder(n)$  в списке узлов, упорядоченных в обратном порядке. Для того чтобы узел  $x$  был потомком узла  $y$ , надо, чтобы выполнялись следующие неравенства:

$$postorder(y) - desc(y) \leq postorder(x) \leq postorder(y).$$

Подобные функции можно определить и для списков узлов, упорядоченных в прямом порядке.

## 3.2. Абстрактный тип данных TREE

В главе 2 списки, стеки, очереди и отображения получили трактовку как абстрактные типы данных (АТД). В этой главе рассмотрим деревья как АТД и как структуры данных. Одно из наиболее важных применений деревьев — это использование их при разработке реализаций различных АТД. Например, в главе 5 мы покажем, как можно использовать “дерево двоичного поиска” при реализации АТД, основанных на математической модели множеств. В следующих двух главах будут представлены многочисленные примеры применения деревьев при реализации различных абстрактных типов данных.

В этом разделе мы представим несколько полезных операторов, выполняемых над деревьями, и покажем, как использовать эти операторы в различных алгоритмах. Так же, как и в случае списков, можно предложить большой набор операторов, выполняемых над деревьями. Здесь мы рассмотрим следующие операторы.

1. **PARENT( $n, T$ )**. Эта функция возвращает родителя (parent) узла  $n$  в дереве  $T$ . Если  $n$  является корнем, который не имеет родителя, то в этом случае возвращается  $\Lambda$ . Здесь  $\Lambda$  обозначает “нулевой узел” и указывает на то, что мы выходим за пределы дерева.
2. **LEFTMOST\_CHILD( $n, T$ )**. Данная функция возвращает самого левого сына узла  $n$  в дереве  $T$ . Если  $n$  является листом (и поэтому не имеет сына), то возвращается  $\Lambda$ .
3. **RIGHT\_SIBLING( $n, T$ )**. Эта функция возвращает правого брата узла  $n$  в дереве  $T$  и значение  $\Lambda$ , если такового не существует. Для этого находится родитель  $p$  узла  $n$  и все сыновья узла  $p$ , затем среди этих сыновей находится узел, расположенный непосредственно справа от узла  $n$ . Например, для дерева на рис. 3.6 **LEFTMOST\_CHILD( $n_2$ )** =  $n_4$ , **RIGHT\_SIBLING( $n_4$ )** =  $n_5$  и **RIGHT\_SIBLING( $n_5$ )** =  $\Lambda$ .
4. **LABEL( $n, T$ )**. Возвращает метку узла  $n$  дерева  $T$ . Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.
5. **CREATE( $v, T_1, T_2, \dots, T_i$ )** — это обширное семейство “созидающих” функций, которые для каждого  $i = 0, 1, 2, \dots$  создают новый корень  $r$  с меткой  $v$  и далее для этого корня создают  $i$  сыновей, которые становятся корнями поддеревьев  $T_1, T_2, \dots, T_i$ . Эти функции возвращают дерево с корнем  $r$ . Отметим, что если  $i = 0$ , то возвращается один узел  $r$ , который одновременно является и корнем, и листом.
6. **ROOT( $T$ )** возвращает узел, являющимся корнем дерева  $T$ . Если  $T$  — пустое дерево, то возвращается  $\Lambda$ .
7. **MAKENULL( $T$ )**. Этот оператор делает дерево  $T$  пустым деревом.

**Пример 3.5.** Напишем рекурсивную **PREORDER** и нерекурсивную **NPREORDER** процедуры обхода дерева в прямом порядке и составления соответствующего списка его меток. Предположим, что для узлов определен тип данных **node** (узел), так же, как и для типа данных **TREE** (Дерево), причем АТД **TREE** определен для деревьев с

метками, которые имеют тип данных `labeltype` (тип метки). В листинге 3.2 приведена рекурсивная процедура, которая по заданному узлу  $n$  создает список в прямом порядке меток поддерева, корнем которого является узел  $n$ . Для составления списка всех узлов дерева  $T$  надо выполнить вызов `PREORDER(ROOT(T))`.

### Листинг 3.2. Рекурсивная процедура обхода дерева в прямом порядке

```

procedure PREORDER (  $n$ : node );
  var
     $c$ : node;
  begin
    print(LABEL( $n$ ,  $T$ ));
     $c$ := LEFTMOST_CHILD( $n$ ,  $T$ );
    while  $c$  <>  $\Lambda$  do begin
      PREORDER( $c$ );
       $c$ := RIGHT_SIBLING( $c$ ,  $T$ )
    end
  end; { PREORDER }

```

Теперь напишем нерекурсивную процедуру для печати узлов дерева в прямом порядке. Чтобы совершить обход дерева, используем стек  $S$ , чей тип данных `STACK` уже объявлен как “стек для узлов”. Основная идея разрабатываемого алгоритма заключается в том, что, когда мы дошли до узла  $n$ , стек хранит путь от корня до этого узла, причем корень находится на “дне” стека, а узел  $n$  — в вершине стека.<sup>1</sup>

Один из подходов к реализации обхода дерева в прямом порядке показан на примере программы `NPREORDER` в листинге 3.3. Эта программа выполняет два вида операций, т.е. может находиться как бы в одном из двух режимов. Операции первого вида (первый режим) осуществляют обход по направлению к потомкам самого левого еще не проверенного пути дерева до тех пор, пока не встретится лист, при этом выполняется печать узлов этого пути и занесение их в стек.

Во втором режиме выполнения программы осуществляется возврат по пройденному пути с поочередным извлечением узлов из стека до тех пор, пока не встретится узел, имеющий еще “не описанного” правого брата. Тогда программа опять переходит в первый режим и исследует новый путь, начиная с этого правого брата.

Программа начинается в первом режиме с нахождения корня дерева и определения, является ли стек пустым. В листинге 3.3 показан полный код этой программы.  $\square$

### Листинг 3.3. Нерекурсивная процедура обхода дерева в прямом порядке

```

procedure NPREORDER (  $T$ : TREE );
  var
     $m$ : node; { переменная для временного хранения узлов }
     $S$ : STACK; { стек узлов, хранящий путь от корня до
      родителя TOP( $S$ ) текущего узла  $m$  }
  begin { инициализация }
    MAKENULL( $S$ );
     $m$ := ROOT( $T$ );
    while true do
      if  $m$  <>  $\Lambda$  then begin
        print(LABEL( $m$ ,  $T$ ));
        PUSH( $m$ ,  $S$ );

```

<sup>1</sup> Можно вернуться к разделу 2.6, где обсуждалась реализация рекурсивных процедур с помощью стека активационных записей. При рассмотрении листинга 3.2 нетрудно заметить, что активационную запись можно заносить в стек при каждом вызове процедуры `PREORDER( $n$ )` и, это главное, стек будет содержать активационные записи для всех предков узла  $n$ .



```

        { исследование самого левого сына узла  $m$  }
         $m := \text{LEFTMOST\_CHILD}(m, T)$ 
    end
else begin
    { завершена проверка пути, содержащегося в стеке }
    if EMPTY( $S$ ) then
        return;
    { исследование правого брата узла,
      находящегося в вершине стека }
     $m := \text{RIGHT\_SIBLING}(\text{TOP}(S), T);$ 
    POP( $S$ )
end
end; { NPREORDER }

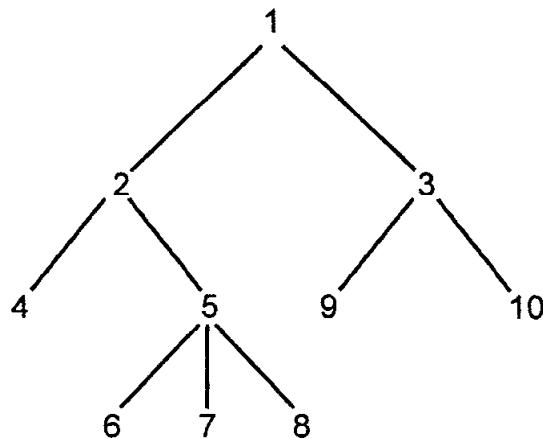
```

### 3.3. Реализация деревьев

В этом разделе мы представим несколько основных реализаций деревьев и обсудим их возможности для поддержки операторов, введенных в разделе 3.2.

#### Представление деревьев с помощью массивов

Пусть  $T$  — дерево с узлами  $1, 2, \dots, n$ . Возможно, самым простым представлением дерева  $T$ , поддерживающим оператор PARENT (Родитель), будет линейный массив  $A$ , где каждый элемент  $A[i]$  является указателем или курсором на родителя узла  $i$ . Корень дерева  $T$  отличается от других узлов тем, что имеет нулевой указатель или указатель на самого себя как на родителя. В языке Pascal указатели на элементы массива недопустимы, поэтому мы будем использовать схему с курсорами, тогда  $A[i] = j$ , если узел  $j$  является родителем узла  $i$ , и  $A[i] = 0$ , если узел  $i$  является корнем.



а. Дерево

	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

б. Курсоры на родителей

Рис. 3.7. Дерево и курсоры на родителей

Данное представление использует то свойство деревьев, что каждый узел, отличный от корня, имеет только одного родителя. Используя это представление, родителя любого узла можно найти за фиксированное время. Прохождение по любому пути, т.е. переход по узлам от родителя к родителю, можно выполнить за время, пропорциональное количеству узлов пути. Для реализации оператора LABEL можно использовать другой массив  $L$ , в котором элемент  $L[i]$  будет хранить метку узла  $i$ , либо объявить элементы массива  $A$  записями, состоящими из целых чисел (курсоров) и меток.

**Пример 3.6.** На рис. 3.7 показаны дерево и массив  $A$  курсоров на родителей этого дерева. □

Использование указателей или курсоров на родителей не помогает в реализации операторов, требующих информацию о сыновьях. Используя описанное представление, крайне тяжело для данного узла  $n$  найти его сыновей или определить его высоту. Кроме того, в этом случае невозможно определить порядок сыновей узла (т.е. какой сын находится правее или левее другого сына). Поэтому нельзя реализовать операторы, подобные LEFTMOST\_CHILD и RIGHT\_SIBLING. Можно ввести искусственный порядок нумерации узлов, например нумерацию сыновей в возрастающем порядке слева направо. Используя такую нумерацию, можно реализовать оператор RIGHT\_SIBLING, код для этого оператора приведен в листинге 3.4. Для задания типов данных node (узел) и TREE (Дерево) используется следующее объявление:

```
type
  node = integer;
  TREE = array [1..maxnodes] of node;
```

В этой реализации мы предполагаем, что нулевой узел  $\Lambda$  представлен 0.

#### Листинг 3.4. Оператор определения правого брата

```
procedure RIGHT_SIBLING ( n: node; T: TREE ) : node;
var
  i, parent: node;
begin
  parent := T[n];
  for i := n + 1 to maxnodes do
    if T[i] = parent then
      return(i);
  return(0) { правый брат не найден }
end; { RIGHT_SIBLING }
```

### Представление деревьев с использованием списков сыновей

Важный и полезный способ представления деревьев состоит в формировании для каждого узла списка его сыновей. Эти списки можно представить любым методом, описанным в главе 2, но, так как число сыновей у разных узлов может быть разное, чаще всего для этих целей применяются связанные списки.

На рис. 3.8 показано, как таким способом представить дерево, изображенное на рис. 3.7, а. Здесь есть массив ячеек заголовков, индексированный номерами (они же имена) узлов. Каждый заголовок (*header*) указывает на связанный список, состоящий из “элементов”-узлов. Элементы списка  $header[i]$  являются сыновьями узла  $i$ , например узлы 9 и 10 — сыновья узла 3.

Прежде чем разрабатывать необходимую структуру данных, нам надо в терминах абстрактного типа данных LIST (список узлов) сделать отдельную реализацию списков сыновей и посмотреть, как эти абстракции согласуются между собой. Позднее мы увидим, какие упрощения можно сделать в этих реализациях. Начнем со следующих объявлений типов:

```

type
  node = integer;
  LIST = { соответствующее определение для списка узлов };
  position = { соответствующее определение позиций в списках };
  TREE = record
    header: array[1..maxnodes] of LIST;
    labels: array[1..maxnodes] of labeltype;
    root: node
  end;
end;

```

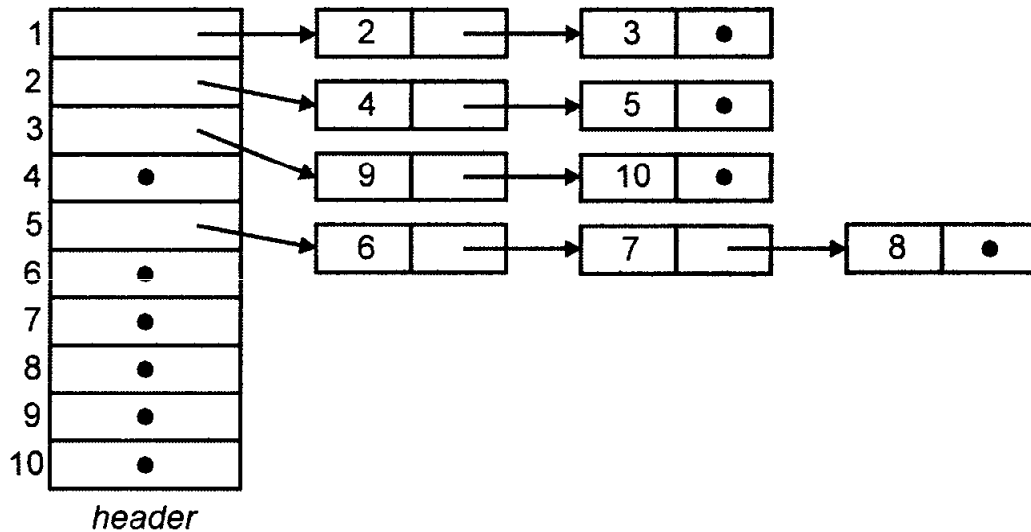


Рис. 3.8. Представление дерева с помощью связанных списков

Мы предполагаем, что корень каждого дерева хранится отдельно в поле *root* (корень). Для обозначения нулевого узла используется 0.

В листинге 3.5 представлен код функции **LEFTMOST\_CHILD**. Читатель в качестве упражнения может написать коды других операторов.

### Листинг 3.5. Функция нахождения самого левого сына

```

function LEFTMOST_CHILD ( n: node; T: TREE ): node;
{ возвращает самого левого сына узла n дерева T }
var
  L: LIST; { скоропись для списка сыновей узла n }
begin
  L := T.header[n];
  if EMPTY(L) then { n является листом }
    return(0)
  else
    return(RETRIEVE(FIRST(L), L))
end; { LEFTMOST_CHILD }

```

Теперь представим конкретную реализацию списков, где типы **LIST** и **position** имеют тип целых чисел, последние используются как курсоры в массиве записей *cellspace* (область ячеек):

```

var
  cellspace: array[1..maxnodes] of record
    node: integer;
    next: integer
  end;
end;

```

Для упрощения реализации можно положить, что списки сыновей не имеют ячеек заголовков. Точнее, мы поместим  $T.header[n]$  непосредственно в первую ячейку списка, как показано на рис. 3.8. В листинге 3.6 представлен переписанный (с учетом этого упрощения) код функции `LEFTMOST_CHILD`, а также показан код функции `PARENT`, использующий данное представление списков. Эта функция более трудна для реализации, так как определение списка, в котором находится заданный узел, требует просмотра всех списков сыновей.

### Листинг 3.6. Функции, использующие представление деревьев посредством связанных списков

```
function LEFTMOST_CHILD ( n: node; T: TREE ): node;
{ возвращает самого левого сына узла n дерева T }
var
  L: integer; { курсор на начало списка сыновей узла n }
begin
  L:= T.header[n];
  if L = 0 then { n является листом }
    return(0)
  else
    return(cellspace[L].node)
end; { LEFTMOST_CHILD }

function PARENT ( n: node; T: TREE ): node;
{ возвращает родителя узла n дерева T }
var
  p: node; { пробегает возможных родителей узла n }
  i: position; { пробегает список сыновей p }
begin
  for p:= 1 to maxnodes do begin
    i:= T.header[p];
    while i <> 0 do { проверка на наличие сыновей узла p }
      if cellspace[i].node = n then
        return(p)
      else
        i:= cellspace[i].next
    end;
    return(0) { родитель не найден }
  end; { PARENT }
```

## Представление левых сыновей и правых братьев

Среди прочих недостатков описанная выше структура данных не позволяет также с помощью операторов `CREATEi` создавать большие деревья из малых. Это является следствием того, что все деревья совместно используют массив *cellspace* для представления связанных списков сыновей; по сути, каждое дерево имеет собственный массив заголовков для своих узлов. А при реализации, например, оператора `CREATE2( $v, T_1, T_2$ )` надо скопировать деревья  $T_1$  и  $T_2$  в третье дерево и добавить новый узел с меткой  $v$  и двух его сыновей — корни деревьев  $T_1$  и  $T_2$ .

Если мы хотим построить большое дерево на основе нескольких малых, то желательно, чтобы все узлы всех деревьев располагались в одной общей области. Логическим продолжением представления дерева, показанного на рис. 3.8, будет замена массива заголовков на отдельный массив *nodespace* (область узлов), содержащий записи с произвольным местоположением в этом массиве. Содержимое поля *header* этих записей соответствует “номеру” узла, т.е. номеру записи в массиве *cellspace*, в

свою очередь поле *node* массива *cellspace* теперь является курсором для массива *nodespace*, указывающим позицию узла. Тип TREE в этой ситуации просто курсор в массиве *nodespace*, указывающий позицию корня.

**Пример 3.7.** На рис. 3.9, а показано дерево, а на рис. 3.9, б — структура данных, где узлы этого дерева, помеченные как A, B, C и D, размещены в произвольных позициях массива *nodespace*. В массиве *cellspace* также в произвольном порядке размещены списки сыновей. □

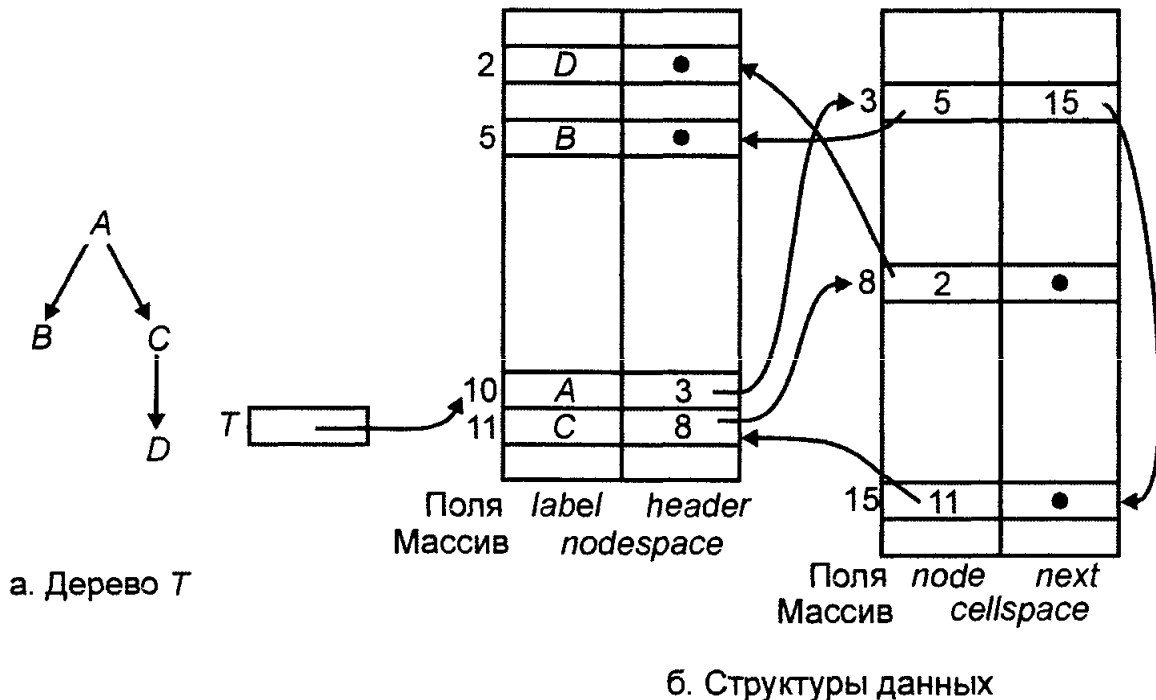


Рис. 3.9. Структура данных для дерева, использующая связанные списки

Структура данных, показанная на рис. 3.9, б, уже подходит для того, чтобы организовать слияние деревьев с помощью операторов CREATEi. Но и эту структуру можно значительно упростить. Для этого заметим, что цепочка указателей поля *next* массива *cellspace* перечисляет всех правых братьев.

Используя эти указатели, можно найти самого левого сына следующим образом. Предположим, что *cellspace[i].node* = *n*. (Повторим, что “имя” узла, в отличие от его метки, является индексом в массиве *nodespace* и этот индекс записан в поле *cellspace[i].node*.) Тогда указатель *nodespace[n].header* указывает на ячейку самого левого сына узла *n* в массиве *cellspace*, поскольку поле *node* этой ячейки является именем этого узла в массиве *nodespace*.

Можно упростить структуру, если идентифицировать узел не с помощью индекса в массиве *nodespace*, а с помощью индекса ячейки в массиве *cellspace*, который соответствует данному узлу как сыну. Тогда указатель *next* (переименуем это поле в *right\_sibling* — правый брат) массива *cellspace* будет точно указывать на правого брата, а информацию, содержащуюся в массиве *nodespace*, можно перенести в новое поле *leftmost\_child* (самый левый сын) массива *cellspace*. Здесь тип TREE является целочисленным типом и используется как курсор в массиве *cellspace*, указывающий на корень дерева. Массив *cellspace* можно описать как следующую структуру:

```
var
  cellspace: array[1..maxnodes] of record
    label: labeltype;
    leftmost_child: integer;
    right_sibling: integer
  end;
```

**Пример 3.8.** Новое представление для дерева, показанного на рис. 3.9, а, схематически изображено на рис. 3.10. Узлы дерева расположены в тех же ячейках массива, что и на рис. 3.9, б.  $\square$

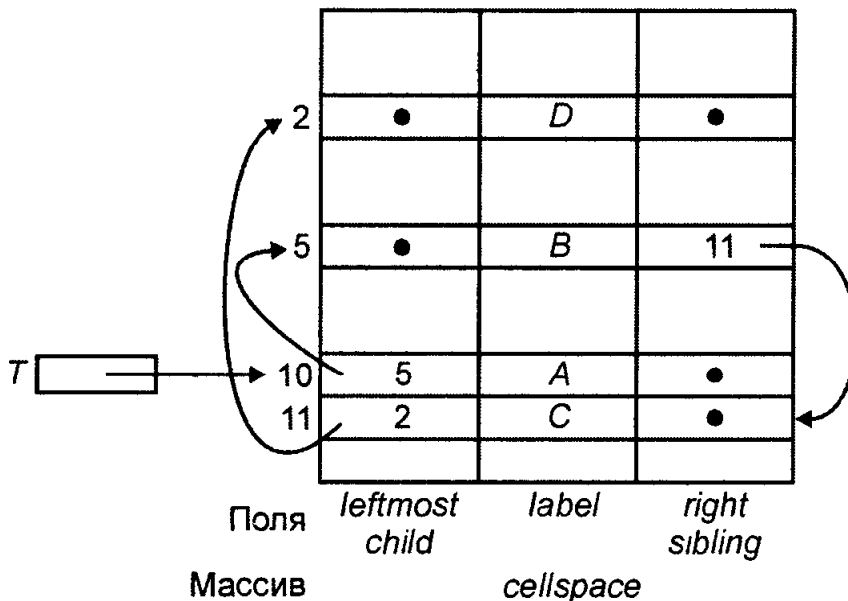


Рис. 3.10. Представление дерева посредством левых сыновей и правых братьев

Используя описанное представление, все операторы, за исключением PARENT, можно реализовать путем прямых вычислений. Оператор PARENT требует просмотра всего массива *cellspace*. Если необходимо эффективное выполнение оператора PARENT, то можно добавить четвертое поле в массив *cellspace* для непосредственного указания на родителей.

В качестве примера операторов, использующих структуры данных рис. 3.10, напомним код функции CREATE2, показанный в листинге 3.7. Здесь мы предполагаем, что неиспользуемые ячейки массива *cellspace* связаны в один свободный список, который в листинге назван как *avail*, и ячейки этого списка связаны посредством поля *right\_sibling*. На рис. 3.11 показаны старые (сплошные линии) и новые (пунктирные линии) указатели в процессе создания нового дерева.

### Листинг 3.7. Функция CREATE2

```

function CREATE2 ( v: labeltype; T1, T2: integer ): integer;
{ возвращает новое дерево с корнем v и поддеревьями T1 и T2 }
var
    temp: integer; { хранит индекс первой свободной ячейки
                     для корня нового дерева }
begin
    temp:= avail;
    avail:= cellspace[avail].right_sibling;
    cellspace[temp].leftmost_child:= T1;
    cellspace[temp].label:= v;
    cellspace[temp].right_sibling:= 0;
    cellspace[T1].right_sibling:= T2;
    cellspace[T2].right_sibling:= 0; {необязательный оператор}
    return(temp)
end; { CREATE2 }

```

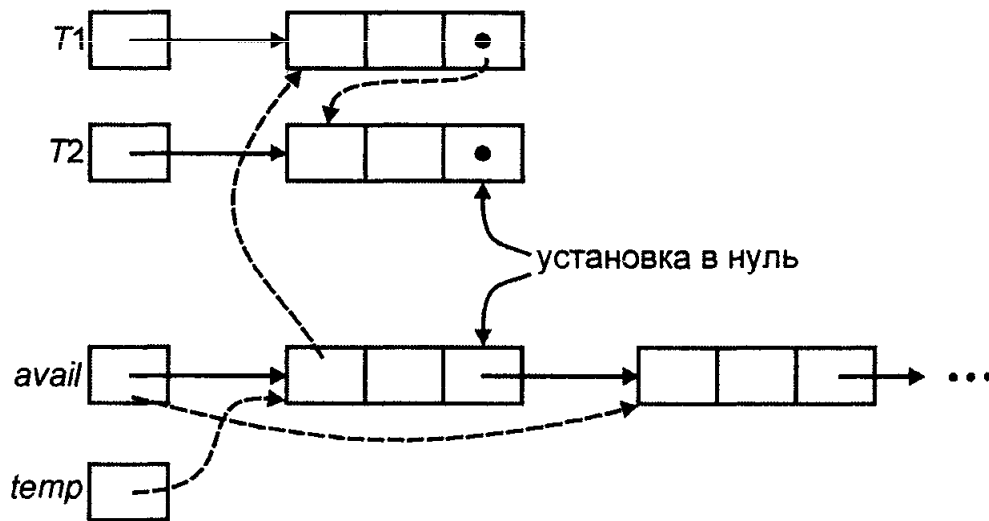


Рис. 3.11. Изменение указателей при выполнении функции CREATE2

Можно уменьшить область памяти, занимаемую узлами дерева (но при этом увеличится время выполнения операторов), если в поле *right\_sibling* самого правого сына вместо нулевого указателя поместить указатель на родителя. Но в этом случае, чтобы избежать двусмысленности, необходимо в каждую ячейку поместить еще двоичную (логическую) переменную, которая будет показывать, что содержится в поле *right\_sibling*: указатель на правого брата или указатель на родителя.

При такой реализации можно найти для заданного узла его родителя, следуя за указателями поля *right\_sibling*, пока не встретится указатель на родителя. В этом случае время, необходимое для поиска родителя, пропорционально количеству сыновей у родителя.

### 3.4. Двоичные деревья

Деревья, которые мы определили в разделе 3.1, называются *упорядоченными ориентированными деревьями*, поскольку сыновья любого узла упорядочены слева направо, а пути по дереву ориентированы от начального узла пути к его потомкам. *Двоичное* (или *бинарное*) *дерево* — совершенно другой вид. Двоичное дерево может быть или пустым деревом, или деревом, у которого любой узел или не имеет сыновей, или имеет либо *левого сына*, либо *правого сына*, либо *обоих*. Тот факт, что каждый сын любого узла определен как левый или как правый сын, существенно отличает двоичное дерево от упорядоченного ориентированного дерева.

**Пример 3.9.** Если мы примем соглашение, что на схемах двоичных деревьев левый сын всегда соединяется с родителем линией, направленной влево и вниз от родителя, а правый сын — линией, направленной вправо и вниз, тогда на рис. 3.12,а, б представлены два различных дерева, хотя они оба похожи на обычное (упорядоченное ориентированное) дерево, показанное на рис. 3.13. Пусть вас не смущает тот факт, что деревья на рис. 3.12,а, б различны и не эквивалентны дереву на рис. 3.13. Дело в том, что двоичные деревья нельзя непосредственно сопоставить обычному дереву. Например, на рис. 3.12,а узел 2 является левым сыном узла 1 и узел 1 не имеет правого сына, тогда как на рис. 3.12,б узел 1 не имеет левого сына, а имеет правого (узел 2). В тоже время в обоих двоичных деревьях узел 3 является левым сыном узла 2, а узел 4 — правым сыном того же узла 2. □

Обход двоичных деревьев в прямом и обратном порядке в точности соответствует таким же обходам обычных деревьев. При симметричном обходе двоичного дерева с

корнем  $n$  левым поддеревом  $T_1$  и правым поддеревом  $T_2$  сначала проходится поддерево  $T_1$ , затем корень  $n$  и далее поддерево  $T_2$ . Например, симметричный обход дерева на рис. 3.12,а даст последовательность узлов 3, 5, 2, 4, 1.

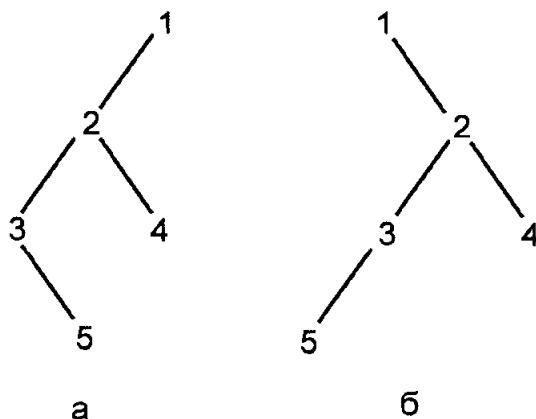


Рис. 3.12. Два двоичных дерева

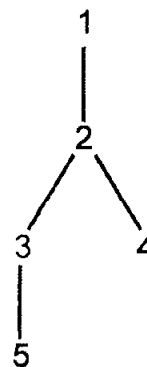


Рис. 3.13. “Обычное” дерево

## Представление двоичных деревьев

Если именами узлов двоичного дерева являются их номера 1, 2, ...,  $n$ , то подходящей структурой для представления этого дерева может служить массив *cellspace* записей с полями *leftchild* (левый сын) и *rightchild* (правый сын), объявленный следующим образом:

```
var
    cellspace: array[1..maxnodes] of record
        leftchild: integer;
        rightchild: integer
    end;
```

В этом представлении *cellspace[i].leftchild* является левым сыном узла  $i$ , а *cellspace[i].rightchild* — правым сыном. Значение 0 в обоих полях указывает на то, что узел  $i$  не имеет сыновей.

Пример 3.10. Двоичное дерево на рис. 3.12,а можно представить в виде табл. 3.1. □

Таблица 3.1. Представление двоичного дерева

	Значение поля <i>leftchild</i>	Значение поля <i>rightchild</i>
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

## Пример: коды Хаффмана

Приведем пример применения двоичных деревьев в качестве структур данных. Для этого рассмотрим задачу конструирования *кодов Хаффмана*. Предположим, мы имеем сообщения, состоящие из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении. Например, мы имеем сообщения, состоящие из пяти символов  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$ , которые появляются в сообщениях с вероятностями 0.12, 0.4, 0.15, 0.08 и 0.25 соответственно.



Мы хотим закодировать каждый символ последовательностью из нулей и единиц так, чтобы код любого символа являлся префиксом кода сообщения, состоящего из последующих символов. Это *префиксное свойство* позволяет декодировать строку из нулей и единиц последовательным удалением префиксов (т.е. кодов символов) из этой строки.

**Пример 3.11.** В табл. 3.2 показаны две возможные кодировки для наших пяти символов. Ясно, что первый код обладает префиксным свойством, поскольку любая последовательность из трех битов будет префиксом для другой последовательности из трех битов; другими словами, любая префиксная последовательность однозначно идентифицируется символом. Алгоритм декодирования для этого кода очень прост: надо поочередно брать по три бита и преобразовать каждую группу битов в соответствующие символы. Например, последовательность 001010011 соответствует исходному сообщению *bcd*.

Таблица 3.2. Два двоичных кода

Символ	Вероятность	Код 1	Код 2
<i>a</i>	0.12	000	000
<i>b</i>	0.40	001	11
<i>c</i>	0.15	010	01
<i>d</i>	0.08	011	001
<i>e</i>	0.25	100	10

Легко проверить, что второй код также обладает префиксным свойством. Процесс декодирования здесь не отличается от аналогичного процесса для первого кода. Единственная сложность для второго кода заключается в том, что нельзя сразу всю последовательность битов разбить на отдельные сегменты, соответствующие символам, так как символы могут кодироваться и двумя и тремя битами. Для примера рассмотрим двоичную последовательность 1101001, которая опять представляет символы *bcd*. Первые два бита 11 однозначно соответствуют символу *b*, поэтому их можно удалить, тогда получится 01001. Здесь 01 также однозначно определяет символ *c* и т.д.  $\square$

Задача конструирования кодов Хаффмана заключается в следующем: имея множество символов и значения вероятностей их появления в сообщениях, построить такой код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальной. Мы хотим минимизировать среднюю длину кода для того, чтобы уменьшить длину вероятного сообщения (т.е. чтобы сжать сообщение). Чем короче среднее значение длины кода символов, тем короче закодированное сообщение. В частности, первый код из примера 3.11 имеет среднюю длину кода 3. Это число получается в результате умножения длины кода каждого символа на вероятность появления этого символа. Второй код имеет среднюю длину 2.2, поскольку символы *a* и *d* имеют суммарную вероятность появления 0.20 и длина их кода составляет три бита, тогда как другие символы имеют код длиной 2<sup>1</sup>.

Можно ли придумать код, который был бы лучше второго кода? Ответ положительный: существует код с префиксным свойством, средняя длина которого равна 2.15. Это наилучший возможный код с теми же вероятностями появления символов. Способ нахождения оптимального префиксного кода называется *алгоритмом Хаффмана*. В этом алгоритме находятся два символа *a* и *b* с наименьшими вероятностями появления и заменяются одним фиктивным символом, например *x*, который имеет вероятность появления, равную сумме вероятностей появления символов *a* и *b*. Затем, используя эту процедуру рекурсивно, находим оптимальный префиксный код для

<sup>1</sup> Отсюда следует очевидный вывод, что символы с большими вероятностями появления должны иметь самые короткне коды. — *Прим. ред.*

меньшего множества символов (где символы  $a$  и  $b$  заменены одним символом  $x$ ). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа  $a$  будет соответствовать коду символа  $x$  с добавленным нулем перед этим кодом, а для кода символа  $b$  перед кодом символа  $x$  будет добавлена единица.

Можно рассматривать префиксные коды как пути на двоичном дереве: прохождение от узла к его левому сыну соответствует 0 в коде, а к правому сыну — 1. Если мы пометим листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева. Префиксное свойство гарантирует, что нет символов, которые были бы метками внутренних узлов дерева (не листьев), и наоборот, помечая кодируемыми символами только листья дерева, мы обеспечиваем префиксное свойство кода этих символов.

**Пример 3.12.** Двоичные деревья для кодов 1 и 2 из табл. 3.2 показаны на рис. 3.14 (дерево слева соответствует коду 1, а дерево справа — коду 2).  $\square$

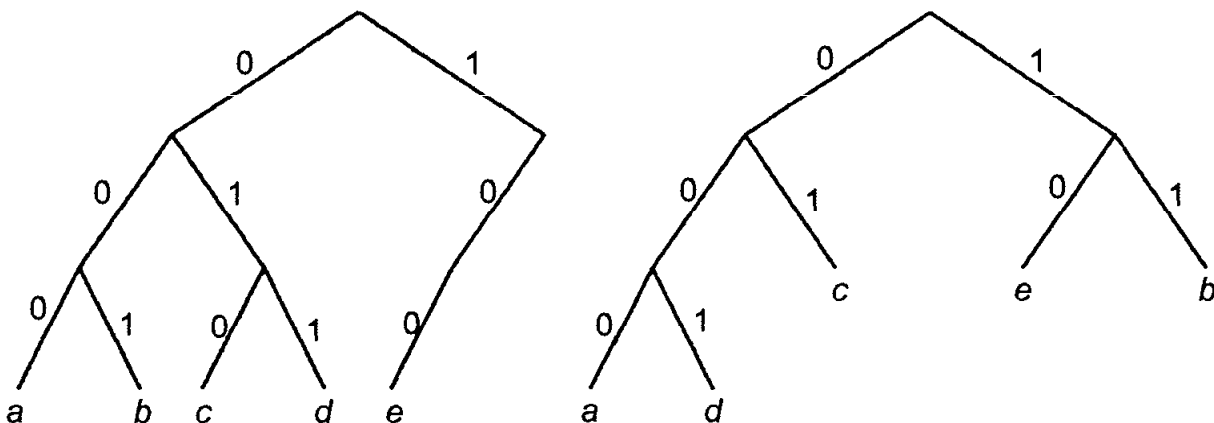


Рис. 3.14. Двоичные деревья, представляющие коды с префиксным свойством

Для реализации алгоритма Хаффмана мы используем *лес*, т.е. совокупность деревьев, чьи листья будут помечены символами, для которых разрабатывается кодировка, а корни помечены суммой вероятностей всех символов, соответствующих листьям дерева. Мы будем называть эти суммарные вероятности *весом* дерева. Вначале каждому символу соответствует дерево, состоящее из одного узла, в конце работы алгоритма мы получим одно дерево, все листья которого будут помечены кодируемыми символами. В этом дереве путь от корня к любому листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует 0, а правый — 1 (как на рис. 3.14).

Важным этапом в работе алгоритма является выбор из леса двух деревьев с наименьшими весами. Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих деревьев. При слиянии деревьев создается новый узел, который становится корнем объединенного дерева и который имеет в качестве левого и правого сыновей корни старых деревьев. Этот процесс продолжается до тех пор, пока не получится только одно дерево. Это дерево соответствует коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

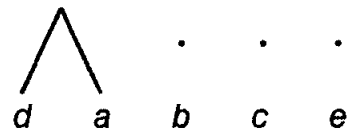
**Пример 3.13.** Последовательные шаги выполнения алгоритма Хаффмана для кодируемых символов и их вероятностей, заданных в табл. 3.2, представлены на рис. 3.15. Здесь видно (рис. 3.15,д), что символы  $a$ ,  $b$ ,  $c$ ,  $d$  и  $e$  получили соответственно коды 1111, 0, 110, 1110 и 10. В этом примере существует только одно нетривиальное дерево, соответствующее оптимальному коду, но в общем случае их может быть несколько. Например, если бы символы  $b$  и  $e$  имели вероятности соответственно 0.33 и 0.32, то после шага алгоритма, показанного на рис. 3.15,в, можно было бы комбинировать  $b$  и  $e$ , а не присоединять  $e$  к большому дереву, как это сделано на рис. 3.15,г.  $\square$

0.12 0.40 0.15 0.08 0.25

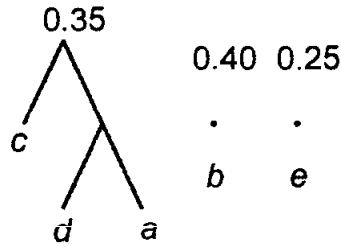
. . . . .  
a b c d e

а. Исходная ситуация

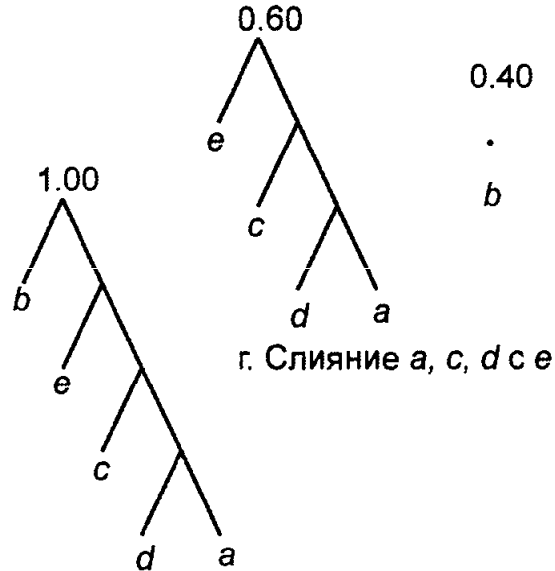
0.20 0.40 0.15 0.25



б. Слияние a с d



в. Слияние a, d с c



г. Слияние a, c, d с e

д. Законченное дерево

Рис. 3.15. Этапы создания дерева Хаффмана

Теперь опишем необходимые структуры данных. Во-первых, для представления двоичных деревьев мы будем использовать массив *TREE* (Дерево), состоящий из записей следующего типа:

```
record
    leftchild: integer;
    rightchild: integer;
    parent: integer
end
```

Указатели в поле *parent* (родитель) облегчают поиск путей от листа к корню при записи кода символов. Во-вторых, мы используем массив *ALPHABET* (Алфавит), также состоящий из записей, которые имеют следующий тип:

```
record
    symbol: char;
    probability: real;
    leaf: integer { курсор }
end
```

В этом массиве каждому символу (поле *symbol*), подлежащему кодированию, ставится в соответствие вероятность его появления (поле *probability*) и лист, меткой которого он является (поле *leaf*). В-третьих, для представления непосредственно деревьев необходим массив *FOREST* (Лес). Этот массив будет состоять из записей с полями *weight* (вес) и *root* (корень) следующего типа:

```

record
    weight: real;
    root: integer
end

```

Начальные значения всех трех массивов, соответствующих данным на рис. 3.15,а, показаны на рис. 3.16. Эскиз программы (псевдопрограмма, т.е. программа на псевдоязыке, как описано в главе 1) построения дерева Хаффмана представлен в листинге 3.8.

1	0.12	1	1	a	0.12	1	1	0	0	0
2	0.40	2		b	0.40	2		0	0	0
3	0.15	3		c	0.15	3		0	0	0
4	0.08	4		d	0.08	4		0	0	0
5	0.25	5		e	0.25	5		0	0	0

Поля *weight* *root*

*symbol* *proba-* *leaf*  
*bility*

*left-* *right-* *parent*  
*child* *child*

Массивы *FOREST*

*ALPHABET*

*TREE*

Рис. 3.16. Исходное состояние массивов

### Листинг 3.8. Программа построения дерева Хаффмана

```

(1) while существует более одного дерева в лесу do
    begin
(2)     i:= индекс дерева в FOREST с наименьшим весом;
(3)     j:= индекс дерева в FOREST со вторым наименьшим весом;
(4)     Создание нового узла с левым сыном FOREST[i].root и
        правым сыном FOREST[j].root;
(5)     Замена в FOREST дерева i деревом, чьим корнем является
        новый узел и чей вес равен
        FOREST[i].weight + FOREST[j].weight;
(6)     Удаление дерева j из массива FOREST
    end;

```

Для реализации строки (4) листинга 3.8, где увеличивается количество используемых ячеек массива *TREE*, и строк (5) и (6), где уменьшается количество ячеек массива *FOREST*, мы будем использовать курсоры *lasttree* (последнее дерево) и *lastnode* (последний узел), указывающие соответственно на массив *FOREST* и массив *TREE*. Предполагается, что эти курсоры располагаются в первых ячейках соответствующих массивов<sup>1</sup>. Мы также предполагаем, что все массивы имеют определенную объявленную длину, но здесь мы не будем проводить сравнение этих ограничивающих значений со значениями курсоров.

В листинге 3.9 приведены коды двух полезных процедур. Первая из них, *lightones*, выполняет реализацию строк (2) и (3) листинга 3.8 по выбору индексов двух деревьев с наименьшими весами. Вторая процедура, функция *create*( $n_1$ ,  $n_2$ ), создает новый узел и делает заданные узлы  $n_1$  и  $n_2$  левым и правым сыновьями этого узла.

<sup>1</sup> Для этапа чтения данных, который мы опускаем, необходим также курсор для массива *ALPHABET*, который бы "следил" за заполнением данным этого массива.

### Листинг 3.9. Две процедуры

```
procedure lightones ( var least, second: integer );
{ присваивает переменным least и second индексы массива
  FOREST, соответствующие деревьям с наименьшими весами.
  Предполагается, что lasttree ≥ 2. }
var
  i: integer;
begin { инициализация least и second,
      рассматриваются первые два дерева }
  if FOREST[1].weight <= FOREST[2].weight then
    begin least:= 1; second:= 2 end
  else
    begin least:= 2; second:= 1 end
  for i:= 3 to lasttree do
    if FOREST[i].weight < FOREST[least].weight then
      begin second:= least; least:= i end
    else if FOREST[i].weight < FOREST[second].weight then
      second:= i
  end; { lightones }

function create ( lefttree, righttree: integer ): integer;
{ возвращает новый узел, у которого левым и правым сыновьями
  становятся FOREST[lefttree].root и FOREST[righttree].root }
begin
  lastnode:= lastnode + 1;
  { ячейка TREE[lastnode] для нового узла }
  TREE[lastnode].leftchild:= FOREST[lefttree].root;
  TREE[lastnode].rightchild:= FOREST[righttree].root;
  { теперь введем указатели для нового узла и его сыновей }
  TREE[lastnode].parent:= 0;
  TREE[FOREST[lefttree].root].parent:= lastnode;
  TREE[FOREST[righttree].root].parent:= lastnode;
  return(lastnode)
end; { create }
```

Теперь все неформальные операторы листинга 3.8 можно описать подробнее. В листинге 3.10 приведен код процедуры *Huffman*, которая не осуществляет ввод и вывод, а работает со структурами, показанными на рис. 3.16, которые объявлены как глобальные.

### Листинг 3.10. Реализация алгоритма Хаффмана

```
procedure Huffman;
var
  i, j: integer; {два дерева с наименьшими весами из FOREST}
  newroot: integer;
begin
  while lasttree > 1 do begin
    lightones(i, j);
    newroot:= create(i, j);
    { далее дерево i заменяется деревом с корнем newroot }
    FOREST[i].weight:=FOREST[i].weight + FOREST[j].weight;
    FOREST[i].root:= newroot;
    { далее дерево j заменяется на дерево lasttree,
      массив FOREST уменьшается на одну запись }
```

```

FOREST[j] := FOREST[lasttree];
lasttree := lasttree - 1
end
end; { Huffman }

```

На рис. 3.17 показана структура данных из рис. 3.16 после того, как значение переменной *lasttree* уменьшено до 3, т.е. лес имеет вид, показанный на рис. 3.15,в.

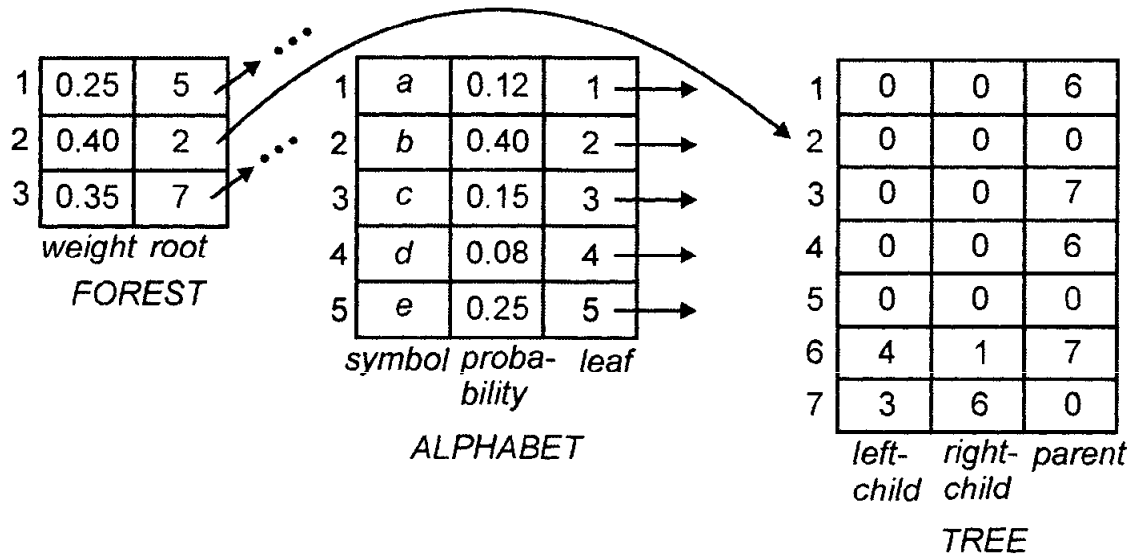


Рис. 3.17. Структура данных после двух итераций

После завершения работы алгоритма код каждого символа можно определить следующим образом. Найдем в массиве *ALPHABET* запись с нужным символом в поле *symbol*. Затем по значению поля *leaf* этой же записи определим местоположение записи в массиве *TREE*, которая соответствует листу, помеченному рассматриваемым символом. Далее последовательно переходим по указателю *parent* от текущей записи, например соответствующей узлу *n*, к записи в массиве *TREE*, соответствующей его родителю *p*. По родителю *p* определяем, в каком его поле, *leftchild* или *rightchild*, находится указатель на узел *n*, т.е. является ли узел *n* левым или правым сыном, и в соответствии с этим печатаем 0 (для левого сына) или 1 (для правого сына). Затем переходим к родителю узла *p* и определяем, является ли его сын *p* правым или левым, и в соответствии с этим печатаем следующую 1 или 0, и т. д. до самого корня дерева. Таким образом, код символа будет напечатан в виде последовательности битов, но в обратном порядке. Чтобы распечатать эту последовательность в прямом порядке, надо каждый очередной бит помещать в стек, а затем распечатать содержимое стека в обычном порядке.

## Реализация двоичных деревьев с помощью указателей

Для указания на правых и левых сыновей (и родителей, если необходимо) вместо курсоров можно использовать настоящие указатели языка Pascal. Например, можно сделать объявление

```

type
  node = record
    leftchild: ↑ node;
    rightchild: ↑ node;
    parent: ↑ node
  end

```

Используя этот тип данных узлов двоичного дерева, функцию *create* (листинг 3.9) можно переписать так, как показано в следующем листинге.

**Листинг 3.11. Код функции *create* при реализации двоичного дерева с помощью указателей**

```
function create ( lefttree, righttree: ↑node): ↑node;
var
    root: ↑node;
begin
    new(root);
    root↑.leftchild:= lefttree;
    root↑.rightchild:= righttree;
    root↑.parent:= 0;
    lefttree↑.parent:= root;
    righttree↑.parent:= root;
    return(root)
end; { create }
```

## Упражнения

3.1. Ответьте на следующие вопросы о дереве, показанном на рис. 3.18:

- а) какие узлы этого дерева являются листьями?
- б) какой узел является корнем дерева?
- в) какой узел является родителем узла *C*?
- г) назовите сыновей узла *C*;
- д) назовите предков узла *E*;
- е) назовите потомков узла *E*;
- ж) какие узлы являются правыми братьями узлов *D* и *E*?
- з) какие узлы лежат слева и справа от узла *G*?
- и) какова глубина узла *C*?
- к) какова высота узла *C*?

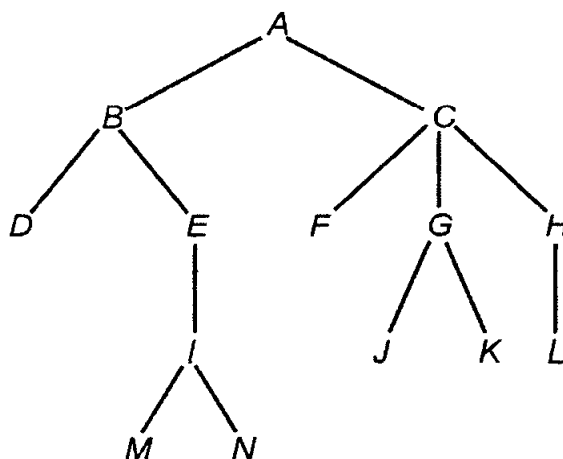


Рис. 3.18. Дерево

- 3.2. Сколько путей длины 3 существует на дереве, показанном на рис. 3.18?
- 3.3. Напишите программы вычисления высоты дерева с использованием трех представлений деревьев, описанных в разделе 3.3.

- 3.4. Составьте списки узлов дерева, представленного на рис. 3.18, при обходе этого дерева
- а) в прямом порядке;
  - б) в обратном порядке;
  - в) во внутреннем порядке.
- 3.5. Пусть два различных узла  $m$  и  $n$  принадлежат одному и тому же дереву. Покажите, что только одно из следующих утверждений может быть истинным:
- а) узел  $m$  расположен слева от узла  $n$ ;
  - б) узел  $m$  расположен справа от узла  $n$ ;
  - в) узел  $m$  — истинный предок узла  $n$ ;
  - г) узел  $m$  — истинный потомок узла  $n$ .
- 3.6. Поставьте галочку в ячейку на пересечении строки  $i$  и столбца  $j$ , если одновременно выполняются условия, представленные в заголовках строки  $i$  и столбца  $j$ .

	Узел $n$ предшествует узлу $m$ при обходе дерева в прямом порядке	Узел $n$ предшествует узлу $m$ при обходе дерева в обратном порядке	Узел $n$ предшествует узлу $m$ при симметричном обходе дерева
Узел $n$ расположен слева от узла $m$			
Узел $n$ расположен справа от узла $m$			
Узел $n$ — истинный предок узла $m$			
Узел $n$ — истинный потомок узла $m$			

Например, поставьте галочку в ячейку на пересечение третьей строки и второго столбца, если уверены, что узел  $n$  может быть истинным предком узла  $m$  и в тоже время предшествовать узлу  $m$  при обходе дерева в обратном порядке.

- 3.7. Предположим, что есть массивы  $PREORDER[n]$ ,  $INORDER[n]$  и  $POSTORDER[n]$ , содержащие списки узлов дерева, полученные при его обходе в прямом, внутреннем и обратном порядке соответственно. Используя эти массивы, опишите алгоритм, который для любой пары узлов  $i$  и  $j$  определяет, является ли узел  $i$  предком узла  $j$ .
- \*3.8. Существует способ проверить, является ли один узел истинным предком другого узла, который основан на следующем правиле: узел  $m$  является истинным предком узла  $n$ , если узел  $m$  предшествует узлу  $n$  при обходе дерева в  $X$  порядке, но следует за узлом  $n$  при обходе в  $Y$  порядке, где  $X$  и  $Y$  выбираются из множества {прямом, обратном, внутреннем}. Определите все возможные пары  $X$  и  $Y$ , когда это правило справедливо.
- 3.9. Напишите программы обхода двоичных деревьев
- а) в прямом порядке;
  - б) в обратном порядке;
  - в) во внутреннем порядке.



- 3.10. При прохождении дерева *в порядке уровней* в список узлов сначала заносится корень дерева, затем все узлы глубины 1, далее все узлы глубины 2 и т.д. Узлы одной глубины заносятся в список узлов в порядке слева направо. Напишите программу обхода деревьев в порядке уровней.
- 3.11. Преобразуйте выражение  $((a + b) + c * (d + e) + f) * (g + h)$
- а) в префиксную форму;
  - б) в постфиксную форму.
- 3.12. Нарисуйте дерево, соответствующее префиксным выражениям
- а)  $*a + b*c + de$ ;
  - б)  $*a + *b + cde$ .
- 3.13. Пусть  $T$  — дерево, в котором каждый узел, не являющийся листом, имеет ровно двух сыновей. Напишите программы преобразования
- а) списка узлов дерева  $T$ , составленного при обходе дерева в прямом порядке, в список, составленный при обходе в обратном порядке;
  - б) списка узлов дерева  $T$ , составленного при обходе дерева в обратном порядке, в список, составленный при обходе в прямом порядке;
  - в) списка узлов дерева  $T$ , составленного при обходе дерева в прямом порядке, в список, составленный при симметричном обходе.
- 3.14. Напишите программу вычисления арифметических выражений при обходе дерева
- а) в прямом порядке;
  - б) в обратном порядке.
- 3.15. Двоичное дерево можно определить как АТД со структурой двоичного дерева и операторами LEFTCHILD( $n$ ), RIGHTCHILD( $n$ ), PARENT( $n$ ) и NULL( $n$ ). Первые три оператора возвращают соответственно левого сына, правого сына и родителя узла  $n$  (если такового нет, то возвращается  $\Lambda$ ). Последний оператор возвращает значение true тогда и только тогда, когда дерево с корнем  $n$  является нулевым. Реализуйте эти операторы для двоичного дерева, представленного в табл. 3.1.
- 3.16. Напишите процедуры для семи операторов деревьев из раздела 3.2, используя следующие реализации деревьев:
- а) указатели на родителей;
  - б) список сыновей;
  - в) указатели на самого левого сына и на правого брата.
- 3.17. *Степенью узла* называется количество его сыновей. Покажите, что в произвольном двоичном дереве количество листьев на единицу больше числа узлов со степенью 2.
- 3.18. Докажите, что в любом двоичном дереве высотой  $h$  количество узлов не превышает  $2^{h+1} - 1$ . Двоичное дерево высотой  $h$  с максимально возможным количеством узлов называется *полным* двоичным деревом.
- \*3.19. Предположим, что дерево реализуется с помощью указателей на самого левого сына, на правых братьев и на родителя. Разработайте алгоритмы обхода деревьев в прямом, обратном и внутреннем порядке, не используя при этом стека, как сделано в листинге 3.3.
- 3.20. Пусть символы  $a, b, c, d, e$  и  $f$  имеют вероятности появления соответственно 0.07, 0.09, 0.12, 0.22, 0.23, 0.27. Найдите оптимальный код Хаффмана и нарисуйте соответствующее ему дерево. Какова средняя длина кода?

- \*3.21. Пусть  $T$  — дерево Хаффмана и пусть лист, помеченный символом  $a$ , имеет большую глубину, чем лист, помеченный символом  $b$ . Докажите, что вероятность символа  $b$  не меньше, чем вероятность символа  $a$ .
- \*3.22. Докажите, что в результате выполнения алгоритма Хаффмана для заданных вероятностей получается оптимальный код. *Совет:* используйте упражнение 3.21.

### Библиографические замечания

В работах [11] и [47] рассматриваются математические свойства деревьев. В [65] и [81] приведена дополнительная информация о деревьях двоичного поиска. Многие работы, приведенные в главе 6 и относящиеся к графам и их применению, содержат также материал о деревьях.

Алгоритм поиска оптимального кода, описанный в разделе 3.4, взят из работы [54]. В [83] дано современное исследование этого алгоритма.