ГЛАВА 12

Бинарные деревья поиска

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

Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время $\Theta(\lg n)$ в наихудшем случае. Как будет показано в разделе 12.4, математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\lg n)$, так что все основные операции над динамическим множеством в таком дереве выполняются в среднем за время $\Theta(\lg n)$.

На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. В главе 13 будет представлена одна из таких версий, а именно — красно-черные деревья, высота которых $O(\lg n)$. В главе 18 вы познакомитесь с B-деревьями, которые особенно хорошо подходят для баз данных, хранящихся во вторичной памяти с произвольным доступом (на дисках).

После знакомства с основными свойствами деревьев поиска, в последующих разделах главы будет показано, как осуществляется обход дерева поиска для вывода его элементов в отсортированном порядке, как выполняется поиск минимального и максимального элементов, а также предшествующего данному элементу и следующего за ним, как вставлять элементы в дерево поиска и удалять их оттуда. Основные математические свойства деревьев описаны в приложении Б.

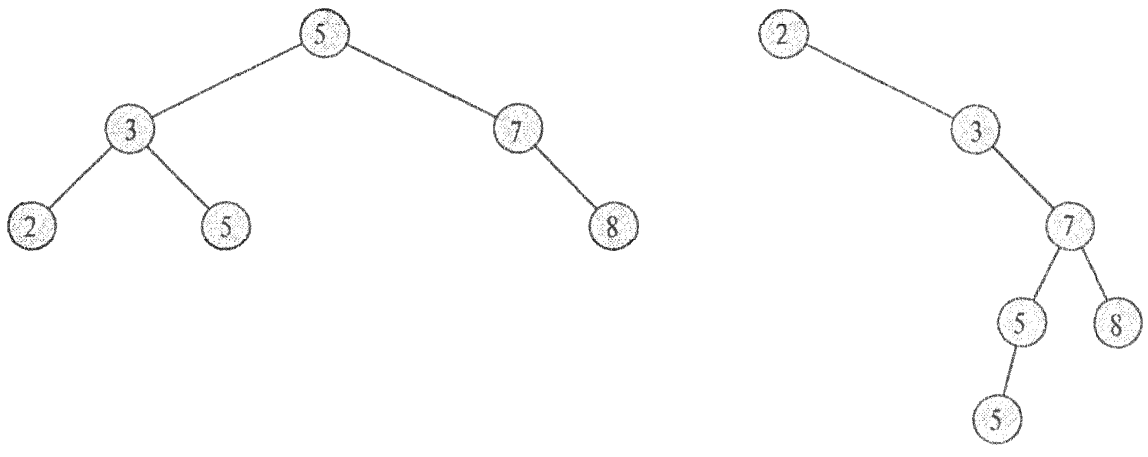
12.1 Что такое бинарное дерево поиска

Как следует из названия, бинарное дерево поиска в первую очередь является бинарным деревом, как показано на рис. 12.1. Такое дерево может быть представлено при помощи связанной структуры данных, в которой каждый узел является объектом. В дополнение к полям ключа *key* и сопутствующих данных, каждый узел содержит поля *left*, *right* и *p*, которые указывают на левый и правый дочерние узлы и на родительский узел соответственно. Если дочерний или родительский узел отсутствуют, соответствующее поле содержит значение NIL. Единственный узел, указатель p которого равен NIL, — это корневой узел дерева. Ключи в бинарном дереве поиска хранятся таким образом, чтобы в любой момент удовлетворять следующему ***свойству бинарного дерева поиска***.

> Если x — узел бинарного дерева поиска, а узел y находится в левом поддереве x, то $key[y] \leqslant key[x]$. Если узел y находится в правом поддереве x, то $key[x] \leqslant key[y]$.

Так, на рис. 12.1*а* ключ корня равен 5, ключи 2, 3 и 5, которые не превышают значение ключа в корне, находятся в его левом поддереве, а ключи 7 и 8, которые не меньше, чем ключ 5, — в его правом поддереве. То же свойство, как легко убедиться, выполняется для каждого другого узла дерева. На рис. 12.1*б* показано дерево с теми же узлами и обладающее тем же свойством, однако менее эффективное в работе, поскольку его высота равна 4, в отличие от дерева на рис. 12.1*а*, высота которого равна 2.

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



а) б)

Рис. 12.1. Бинарные деревья поиска

и правого поддерева. Имеются и другие способы обхода, а именно — ***обход в прямом порядке*** (preorder tree walk), при котором сначала выводится корень, а потом — значения левого и правого поддерева, и ***обход в обратном порядке*** (postorder tree walk), когда первыми выводятся значения левого и правого поддерева, а уже затем — корня. Центрированный обход дерева T реализуется процедурой INORDER_TREE_WALK($root\,[T]$):

```
INORDER_TREE_WALK(x)
1  if x ≠ NIL
2     then INORDER_TREE_WALK(left[x])
3          print key[x]
4          INORDER_TREE_WALK(right[x])
```

В качестве примера рассмотрите центрированный обход деревьев, показанных на рис. 12.1, — вы получите в обоих случаях один и тот же порядок ключей, а именно 2, 3, 5, 5, 7, 8. Корректность описанного алгоритма следует непосредственно из свойства бинарного дерева поиска.

Для обхода дерева требуется время $\Theta(n)$, поскольку после начального вызова процедура вызывается ровно два раза для каждого узла дерева: один раз для его левого дочернего узла, и один раз — для правого. Приведенная далее теорема дает нам более формальное доказательство линейности времени центрированного обхода дерева.

Теорема 12.1. Если x — корень поддерева, в котором имеется n узлов, то процедура INORDER_TREE_WALK(x) выполняется за время $\Theta(n)$.

Доказательство. Обозначим через $T(n)$ время, необходимое процедуре INORDER_TREE_WALK в случае вызова с параметром, представляющим собой корень дерева с n узлами. При получении в качестве параметра пустого поддерева, процедуре требуется небольшое постоянное время для выполнения проверки $x \neq$ NIL, так что $T(0) = c$, где c — некоторая положительная константа.

В случае $n > 0$ будем считать, что процедура INORDER_TREE_WALK вызывается один раз для поддерева с k узлами, а второй — для поддерева с $n - k - 1$ узлами. Таким образом, время работы процедуры составляет $T(n) = T(k) + T(n - k - 1) + d$, где d — некоторая положительная константа, в которой отражается время, необходимое для выполнения процедуры без учета рекурсивных вызовов.

Воспользуемся методом подстановки, чтобы показать, что $T(n) = \Theta(n)$, путем доказательства того, что $T(n) = (c + d)n + c$. При $n = 0$ получаем

$T(0) = (c+d) \cdot 0 + c = c$. Если $n > 0$, то

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d = \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d = \\ &= (c+d)n + c - (c+d) + c + d = (c+d)n + c, \end{aligned}$$

что и завершает доказательство. ■

Упражнения

12.1-1. Начертите бинарные деревья поиска высотой 2, 3, 4, 5 и 6 для множества ключей $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1-2. В чем заключается отличие свойства бинарного дерева поиска от свойства неубывающей пирамиды (раздел 6.1)? Можно ли использовать свойство неубывающей пирамиды для вывода ключей дерева с n узлами в отсортированном порядке за время $O(n)$? Поясните свой ответ.

12.1-3. Разработайте нерекурсивный алгоритм, осуществляющий обход дерева в симметричном порядке. (*Указание*: имеется простое решение, которое использует вспомогательный стек, и более сложное (и более элегантное) решение, которое обходится без стека, но предполагает возможность проверки равенства двух указателей).

12.1-4. Разработайте рекурсивный алгоритм, который осуществляет прямой и обратный обход дерева с n узлами за время $\Theta(n)$.

12.1-5. Покажите, что, поскольку сортировка n элементов требует в модели сортировки сравнением в худшем случае $\Omega(n \lg n)$ времени, любой алгоритм построения бинарного дерева поиска из произвольного списка, содержащего n элементов, также требует в худшем случае $\Omega(n \lg n)$ времени.

12.2 Работа с бинарным деревом поиска

Наиболее распространенной операцией, выполняемой с бинарным деревом поиска, является поиск в нем определенного ключа. Кроме того, бинарные деревья поиска поддерживают такие запросы, как поиск минимального и максимального элемента, а также предшествующего и последующего. В данном разделе мы рассмотрим все эти операции и покажем, что все они могут быть выполнены в бинарном дереве поиска высотой h за время $O(h)$.

Поиск

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

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

Процедура поиска начинается с корня дерева и проходит вниз по дереву. Для каждого узла x на пути вниз его ключ $key\,[x]$ сравнивается с переданным в качестве параметра ключом k. Если ключи одинаковы, поиск завершается. Если k меньше $key\,[x]$, поиск продолжается в левом поддереве x; если больше — то поиск переходит в правое поддерево. Так, на рис. 12.2 для поиска ключа 13 мы должны пройти следующий путь от корня: 15 → 6 → 7 → 13. Узлы, которые мы посещаем при рекурсивном поиске, образуют нисходящий путь от корня дерева, так что время работы процедуры TREE_SEARCH равно $O\,(h)$, где h — высота дерева.

Ту же процедуру можно записать итеративно, "разворачивая" оконечную рекурсию в цикл **while**. На большинстве компьютеров такая версия оказывается более эффективной.

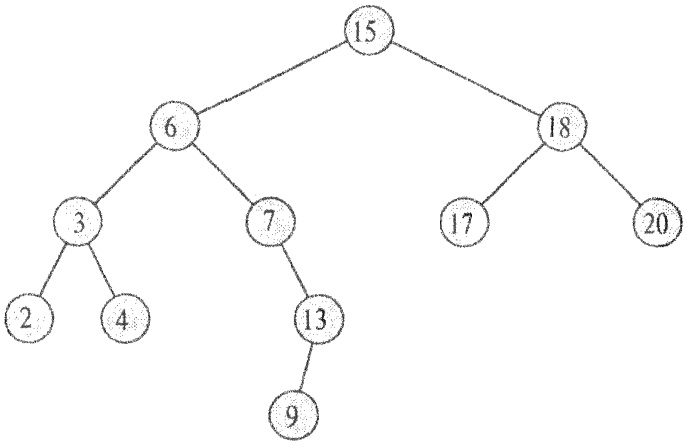


Рис. 12.2. Запросы в бинарном дереве поиска (пояснения в тексте)

```
ITERATIVE_TREE_SEARCH(x, k)
1  while x ≠ NIL и k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Поиск минимума и максимума

Элемент с минимальным значением ключа легко найти, следуя по указателям *left* от корневого узла до тех пор, пока не встретится значение NIL. Так, на рис. 12.2, следуя по указателям *left*, мы пройдем путь 15 → 6 → 3 → 2 до минимального ключа в дереве, равного 2. Вот как выглядит реализация описанного алгоритма:

```
TREE_MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

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

Алгоритм поиска максимального элемента дерева симметричен алгоритму поиска минимального элемента:

```
TREE_MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

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

Предшествующий и последующий элементы

Иногда, имея узел в бинарном дереве поиска, требуется определить, какой узел следует за ним в отсортированной последовательности, определяемой порядком

центрированного обхода бинарного дерева, и какой узел предшествует данному. Если все ключи различны, последующим по отношению к узлу x является узел с наименьшим ключом, большим $key[x]$. Структура бинарного дерева поиска позволяет нам найти этот узел даже не выполняя сравнение ключей. Приведенная далее процедура возвращает узел, следующий за узлом x в бинарном дереве поиска (если таковой существует) и NIL, если x обладает наибольшим ключом в бинарном дереве.

```
TREE_SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE_MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL и x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

Код процедуры TREE_SUCCESSOR разбивается на две части. Если правое поддерево узла x непустое, то следующий за x элемент является крайним левым узлом в правом поддереве, который обнаруживается в строке 2 вызовом процедуры TREE_MINIMUM($right[x]$). Например, на рис. 12.2 следующим за узлом с ключом 15 является узел с ключом 17.

С другой стороны, как требуется показать в упражнении 12.2-6, если правое поддерево узла x пустое, и у x имеется следующий за ним элемент y, то y является наименьшим предком x, чей левый наследник также является предком x. На рис. 12.2 следующим за узлом с ключом 13 является узел с ключом 15. Для того чтобы найти y, мы просто поднимаемся вверх по дереву до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя. Это действие выполняется в строках 3–7 алгоритма.

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

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

Таким образом, в этом разделе мы доказали следующую теорему.

Теорема 12.2. Операции поиска, определения минимального и максимального элемента, а также предшествующего и последующего, в бинарном дереве поиска высоты h могут быть выполнены за время $O(h)$. ■

Упражнения

12.2-1. Пусть у нас имеется ряд чисел от 1 до 1000, организованных в виде бинарного дерева поиска, и мы выполняем поиск числа 363. Какая из следующих последовательностей *не* может быть последовательностью проверяемых узлов?

а) 2, 252, 401, 398, 330, 344, 397, 363.

б) 924, 220, 911, 244, 898, 258, 362, 363.

в) 925, 202, 911, 240, 912, 245, 363.

г) 2, 399, 387, 219, 266, 382, 381, 278, 363.

д) 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2. Разработайте рекурсивные версии процедур TREE_MINIMUM и TREE_MAXIMUM.

12.2-3. Разработайте процедуру TREE_PREDECESSOR.

12.2-4. Разбираясь с бинарными деревьями поиска, студент решил, что обнаружил их новое замечательное свойство. Предположим, что поиск ключа k в бинарном дереве поиска завершается в листе. Рассмотрим три множества: множество ключей слева от пути поиска A, множество ключей на пути поиска B и множество ключей справа от пути поиска C. Студент считает, что любые три ключа $a \in A$, $b \in B$ и $c \in C$ должны удовлетворять неравенству $a \leqslant b \leqslant c$. Приведите наименьший возможный контрпример, опровергающий предположение студента.

12.2-5. Покажите, что если узел в бинарном дереве поиска имеет два дочерних узла, то последующий за ним узел не имеет левого дочернего узла, а предшествующий ему — правого.

12.2-6. Рассмотрим бинарное дерево поиска T, все ключи которого различны. Покажите, что если правое поддерево узла x в бинарном дереве поиска T пустое и у x есть следующий за ним элемент y, то y является самым нижним предком x, чей левый дочерний узел также является предком x. (Вспомните, что каждый узел является своим собственным предком.)

12.2-7. Центрированный обход бинарного дерева поиска с n узлами можно осуществить путем поиска минимального элемента дерева при помощи процедуры TREE_MINIMUM с последующим $n - 1$ вызовом процедуры TREE_SUCCESSOR. Докажите, что время работы такого алгоритма равно $\Theta(n)$.

12.2-8. Докажите, что какой бы узел ни был взят в качестве исходного в бинарном дереве поиска высотой h, на k последовательных вызовов процедуры TREE_SUCCESSOR потребуется время $O(k + h)$.

12.2-9. Пусть T — бинарное дерево поиска с различными ключами, x — лист этого дерева, а y — его родительский узел. Покажите, что $key\,[y]$ либо является наименьшим ключом в дереве T, превышающим ключ $key\,[x]$, либо наибольшим ключом в T, меньшим ключа $key\,[x]$.

12.3 Вставка и удаление

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

Вставка

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

```
TREE_INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Дерево T — пустое
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

На рис. 12.3 показана работа процедуры TREE_INSERT. Подобно процедурам TREE_SEARCH и ITERATIVE_TREE_SEARCH, процедура TREE_INSERT начинает работу с корневого узла дерева и проходит по нисходящему пути. Указатель x отмечает проходимый путь, а указатель y указывает на родительский по отношению к x

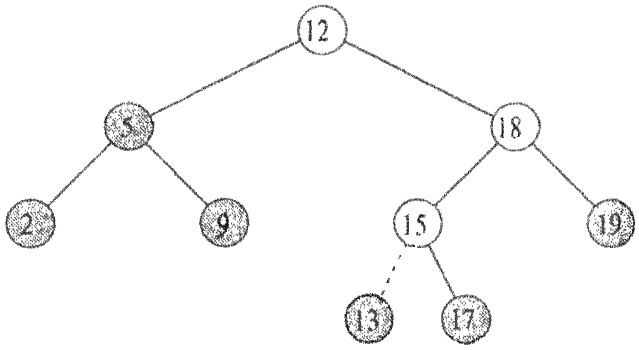


Рис. 12.3. Вставка элемента с ключом 13 в бинарное дерево поиска. Светлые узлы указывают путь от корня к позиции вставки; пунктиром указана связь, добавляемая при вставке нового элемента

узел. После инициализации цикл **while** в строках 3–7 перемещает эти указатели вниз по дереву, перемещаясь влево или вправо в зависимости от результата сравнения ключей $key[x]$ и $key[z]$, до тех пор пока x не станет равным NIL. Это значение находится именно в той позиции, куда следует поместить элемент z. В строках 8–13 выполняется установка значений указателей для вставки z.

Так же, как и другие примитивные операции над бинарным деревом поиска, процедура TREE_INSERT выполняется за время $O(h)$ в дереве высотой h.

Удаление

Процедура удаления данного узла z из бинарного дерева поиска получает в качестве аргумента указатель на z. Процедура рассматривает три возможные ситуации, показанные на рис. 12.4. Если у узла z нет дочерних узлов (рис. 12.4*а*), то мы просто изменяем его родительский узел $p[z]$, заменяя в нем указатель на z значением NIL. Если у узла z только один дочерний узел (рис. 12.4*б*), то мы удаляем узел z, создавая новую связь между родительским и дочерним узлом узла z. И наконец, если у узла z два дочерних узла (рис. 12.4*в*), то мы находим следующий за ним узел y, у которого нет левого дочернего узла (см. упражнение 12.2-5), убираем его из позиции, где он находился ранее, путем создания новой связи между его родителем и потомком, и заменяем им узел z.

Код процедуры TREE_DELETE реализует эти действия немного не так, как они описаны.

```
TREE_DELETE(T, z)
 1  if left[z] = NIL или right[z] = NIL
 2     then y ← z
 3     else y ← TREE_SUCCESSOR(z)
 4  if left[y] ≠ NIL
```

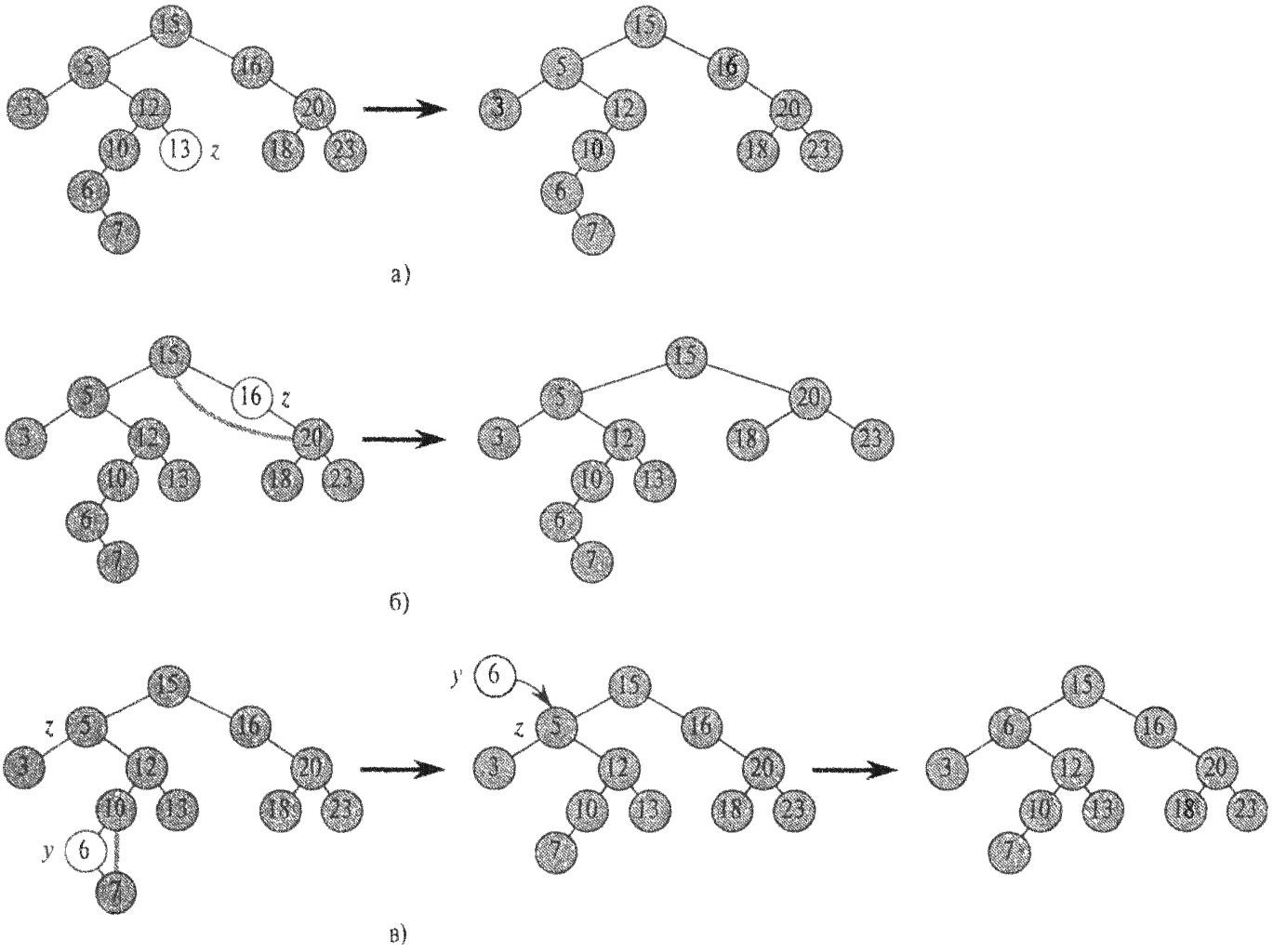


Рис. 12.4. Удаление узла z из бинарного дерева поиска

```
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          Копирование сопутствующих данных в z
17  return y
```

В строках 1–3 алгоритм определяет убираемый путем "склейки" родителя и потомка узел y. Этот узел представляет собой либо узел z (если у узла z не более одного дочернего узла), либо узел, следующий за узлом z (если у z два дочерних

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

Таким образом, в этом разделе мы доказали следующую теорему.

Теорема 12.3. Операции вставки и удаления в бинарном дереве поиска высоты h могут быть выполнены за время $O(h)$. ■

Упражнения

12.3-1. Приведите рекурсивную версию процедуры TREE_INSERT.

12.3-2. Предположим, что бинарное дерево поиска строится путем многократной вставки в дерево различных значений. Покажите, что количество узлов, проверяемых при поиске некоторого значения в дереве, на один больше, чем количество узлов, проверявшихся при вставке этого значения в дерево.

12.3-3. Отсортировать множество из n чисел можно следующим образом: сначала построить бинарное дерево поиска, содержащее эти числа (вызывая процедуру TREE_INSERT для вставки чисел в дерево одно за другим), а затем выполнить центрированный обход получившегося дерева. Чему равно время работы такого алгоритма в наилучшем и наихудшем случае?

12.3-4. Предположим, что указатель на узел y бинарного дерева поиска хранится в некоторой внешней структуре данных и что предшествующий ему узел z удаляется из дерева с помощью процедуры TREE_DELETE. Какая проблема при этом может возникнуть? Каким образом следует переписать процедуру TREE_DELETE, чтобы ее избежать?

12.3-5. Является ли операция удаления "коммутативной" в том смысле, что удаление x с последующим удалением y из бинарного дерева поиска приводит к тому же результирующему дереву, что и удаление y с последующим удалением x? Обоснуйте свой ответ.

12.3-6. Если узел z в процедуре TREE_DELETE имеет два дочерних узла, то можно воспользоваться не последующим за ним узлом, а предшествующим. Утверждается, что стратегия, которая состоит в равновероятном выборе