13 Двоичные деревья поиска

Деревья поиска (search trees) позволяют выполнять следующие операции с динамическими множествами: SEARCH (поиск), MINIMUM (минимум), MAXIMUM (максимум), PREDECESSOR (предыдущий), SUCCESSOR (следующий), INSERT (вставить) и DELETE (удалить). Таким образом, дерево поиска может быть использовано и как словарь, и как очередь с приоритетами.

Время выполнения основных операций пропорционально высоте дерева. Если двоичное дерево «плотно заполнено» (все его уровни имеют максимально возможное число вершин), то его высота (и тем самым время выполнения операций) пропорциональна логарифму числа вершин. Напротив, если дерево представляет собой линейную цепочку из n вершин, это время вырастает до $\Theta(n)$. В разделе 13.4 мы увидим, что высота случайного двоичного дерева поиска есть $O(\log n)$, так что в этом случае время выполнения основных операций есть $\Theta(\log n)$.

Конечно, возникающие на практике двоичные деревья поиска могут быть далеки от случайных. Однако, приняв специальные меры по балансировке деревьев, мы можем гарантировать, что высота дерева с n вершинами будет $O(\log n)$. В главе 14 рассмотрен один из подходов такого рода (красно-чёрные деревья). В главе 19 рассматриваются Б-деревья, которые особенно удобны для данных, хранящихся во вторичной памяти с произвольным доступом (на диске).

В этой главе мы рассмотрим основные операции с двоичными деревьями поиска и покажем, как напечатать элементы дерева в неубывающем порядке, как искать заданный элемент, как найти максимальный или минимальный элемент, как найти элемент, следующий за данным, элемент, предшествующий данному, и, наконец, как добавить или удалить элемент. Напомним, что определение дерева и основные свойства деревьев приводятся в главе 5.

13.1. Что такое двоичное дерево поиска?

В **двоичном дереве поиска** (binary search tree; пример приведён на рис. 13.1) каждая вершина может иметь (или не иметь) левого и правого ребёнка; каждая вершина, кроме корня, имеет родителя. При представлении с использованием указателей мы храним для каждой вершины дерева, помимо значения ключа *key*

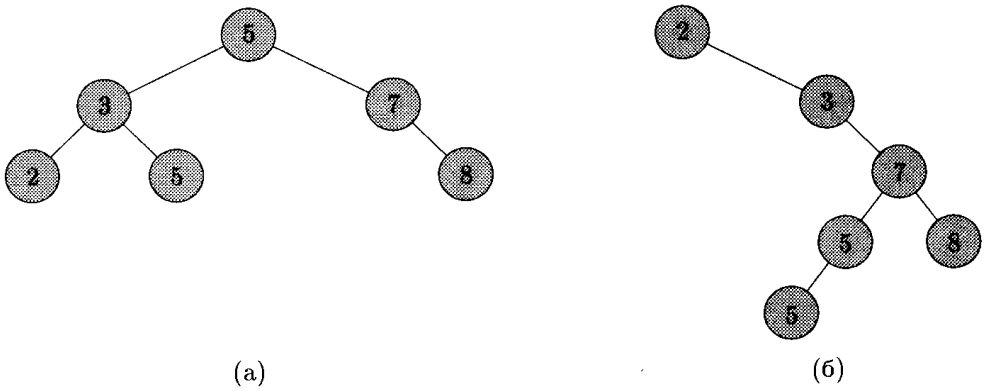

Рис. 13.1. Двоичные деревья поиска. Левое поддерево произвольной вершины x содержит ключи, не превосходящие $key[x]$, правое — не меньшие $key[x]$. Разные двоичные деревья поиска могут представлять одно и то же множество. Время выполнения (в худшем случае) большинства операций пропорционально высоте дерева. (а) Двоичное дерево поиска высоты 2 с 6 вершинами. (б) Менее эффективное дерево высоты 4, содержащее те же ключи.

и дополнительных данных, также и указатели *left*, *right* и p (левый ребёнок, правый ребёнок, родитель). Если ребёнка (или родителя — для корня) нет, соответствующее поле содержит NIL.

Ключи в двоичном дереве поиска хранятся с соблюдением **свойства упорядоченности** (binary-search-tree property):

> Пусть x — произвольная вершина двоичного дерева поиска. Если вершина y находится в левом поддереве вершины x, то $key[y] \leqslant key[x]$. Если y находится в правом поддереве x, то $key[y] \geqslant key[x]$.

Так, на рис. 13.1(а) в корне дерева хранится ключ 5, ключи 2, 3 и 5 в левом поддереве корня не превосходят 5, а ключи 7 и 8 в правом — не меньше 5. То же самое верно для всех вершин дерева. Например, ключ 3 на рис. 13.1(а) не меньше ключа 2 в левом поддереве и не больше ключа 5 в правом.

Свойство упорядоченности позволяет напечатать все ключи в неубывающем порядке с помощью простого рекурсивного алгоритма (называемого по-английски inorder tree walk). Этот алгоритм печатает ключ корня поддерева после всех ключей его левого поддерева, но перед ключами правого поддерева. (Заметим в скобках, что порядок, при котором корень предшествует обоим поддеревьям, называется **preorder**; порядок, в котором корень следует за ними, называется **postorder**.)

Вызов INORDER-TREE-WALK($root[T]$) печатает (в указанном порядке) все ключи, входящие в дерево T с корнем $root[T]$.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          напечатать key[x]
4          INORDER-TREE-WALK(right[x])
```

К примеру, для обоих деревьев рис. 13.1 будет напечатано $2, 3, 5, 5, 7, 8$. Свойство упорядоченности гарантирует правильность алгоритма (индукция по высоте поддерева). Время работы на дереве с n вершинами есть $\Theta(n)$: на каждую вершину тратится ограниченное время (помимо рекурсивных вызовов) и каждая вершина обрабатывается один раз.

Упражнения

13.1-1. Нарисуйте двоичные деревья поиска высоты 2, 3, 4, 5 и 6 для одного и того же множества ключей $\{1, 4, 5, 10, 16, 17, 21\}$.

13.1-2. Кучи из раздела 7.1 также были двоичными деревьями, и требование упорядоченности там тоже было. В чём разница между тем требованием и теперешним? Как вы думаете, можно ли напечатать элементы двоичной кучи в неубывающем порядке за время $O(n)$? Объясните ваш ответ.

13.1-3. Напишите нерекурсивный алгоритм, печатающий ключи в двоичном дереве поиска в неубывающем порядке. (Указание: простое решение использует в качестве дополнительной структуры стек; более изящное решение не требует стека, но предполагает, что можно проверять равенство указателей.)

13.1-4. Напишите рекурсивные алгоритмы для обхода деревьев в различных порядках (preorder, postorder). Как и раньше, время работы должно быть $O(n)$ (где n — число вершин).

13.1-5. Покажите, что любой алгоритм построения двоичного дерева поиска, содержащего заданные n элементов, требует (в худшем случае) времени $\Omega(n \log n)$. Воспользуйтесь тем, что сортировка n чисел требует $\Omega(n \log n)$ действий.

13.2. Поиск в двоичном дереве

В этом разделе мы покажем, что двоичные деревья поиска позволяют выполнять операции SEARCH, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR за время $O(h)$, где h — высота дерева.

Поиск

Процедура поиска получает на вход искомый ключ k и указатель x на корень поддерева, в котором производится поиск. Она возвращает указатель на вершину с ключом k (если такая есть) или специальное значение NIL (если такой вершины нет).

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$(right[x], k)$

В процессе поиска мы двигаемся от корня, сравнивая ключ k с ключом, хранящимся в текущей вершине x. Если они равны, поиск завершается. Если $k < key[x]$, то поиск продолжается в левом поддереве x (ключ k может быть только там, согласно свойству упорядоченности). Если $k > key[x]$, то поиск продолжается в правом поддереве. Длина пути поиска не превосходит высоты дерева, поэтому время поиска есть $O(h)$ (где h — высота дерева).

Вот итеративная версия той же процедуры (которая, как правило, более эффективна):

ITERATIVE-TREE-SEARCH(x, k)
1 **while** $x \neq$ NIL и $k \neq key[x]$
2 **do if** $k < key[x]$
3 **then** $x \leftarrow left[x]$
4 **else** $x \leftarrow right[x]$
5 **return** x

Минимум и максимум

Минимальный ключ в дереве поиска можно найти, пройдя по указателям *left* от корня (пока не упрёмся в NIL), см. рис. 13.2. Процедура возвращает указатель на минимальный элемент поддерева с корнем x.

TREE-MINIMUM(x)
1 **while** $left[x] \neq$ NIL
2 **do** $x \leftarrow left[x]$
3 **return** x

Свойство упорядоченности гарантирует правильность процедуры TREE-MINIMUM. Если у вершины x нет левого ребёнка, то минимальный элемент поддерева с корнем x есть x, так как любой ключ в правом поддереве не меньше $key[x]$. Если же левое поддерево вершины x не пусто, то минимальный элемент поддерева с корнем x находится в этом левом поддереве (поскольку сам x и все элементы правого поддерева больше).

Алгоритм TREE-MAXIMUM симметричен:

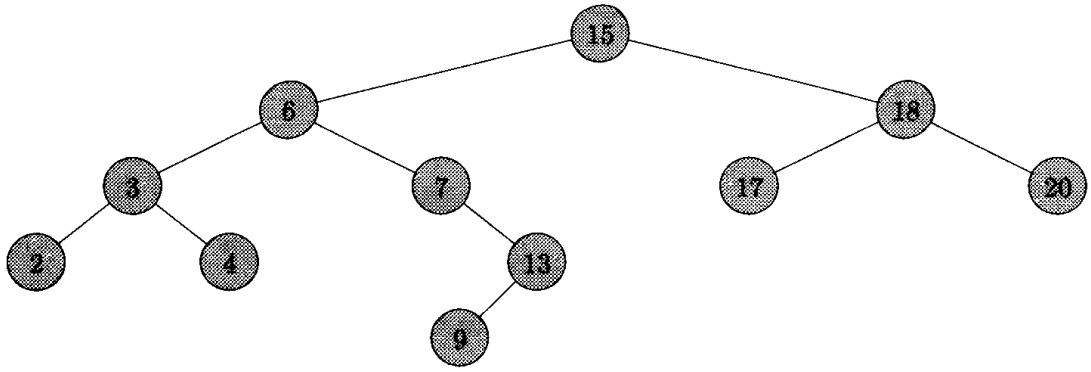

Рис. 13.2. Поиск в двоичном дереве. При поиске ключа 13 мы идём от корня по пути $15 \to 6 \to 7 \to 13$. Чтобы найти минимальный ключ 2, мы всё время идём налево; чтобы найти максимальный ключ 20 — направо. Для вершины с ключом 15 следующей будет вершина с ключом 17 (это минимальный ключ в правом поддереве вершины с ключом 15). У вершины с ключом 13 нет правого поддерева; поэтому, чтобы найти следующую за ней вершину, мы поднимаемся вверх, пока не пройдём по ребру, ведущему вправо-вверх; в данном случае у следующей вершины — ключ 15.

TREE-MAXIMUM(x)
1 **while** $right[x] \neq$ NIL
2 **do** $x \leftarrow right[x]$
3 **return** x

Оба алгоритма требуют времени $O(h)$, где h — высота дерева (поскольку двигаются по дереву только вниз).

Следующий и предыдущий элементы

Как найти в двоичном дереве элемент, следующий за данным? Свойство упорядоченности позволяет сделать это, двигаясь по дереву. Вот процедура, которая возвращает указатель на следующий за x элемент (если все ключи различны, он содержит следующий по величине ключ) или NIL, если элемент x — последний в дереве:

TREE-SUCCESSOR(x)
1 **if** $right[x] \neq$ NIL
2 **then return** TREE-MINIMUM($right[x]$)
3 $y \leftarrow p[x]$
4 **while** $y \neq$ NIL and $x = right[y]$
5 **do** $x \leftarrow y$
6 $y \leftarrow p[y]$
7 **return** y

Процедура TREE-SUCCESSOR отдельно рассматривает два случая. Если правое поддерево вершины x непусто, то следующий за x элемент — минималь-

ный элемент в этом поддереве и равен TREE-MINIMUM($right[x]$). Например, на рис. 13.2 за вершиной с ключом 15 следует вершина с ключом 17.

Пусть теперь правое поддерево вершины x пусто. Тогда мы идём от x вверх, пока не найдём вершину, являющуюся левым сыном своего родителя (строки 3–7). Этот родитель (если он есть) и будет искомым элементом. Формально говоря, цикл в строках 4–6 сохраняет такое свойство: $y = p[x]$; искомый элемент непосредственно следует за элементами поддерева с корнем в x.

Время работы процедуры TREE-SUCCESSOR на дереве высоты h есть $O(h)$, так как мы двигаемся либо только вверх, либо только вниз.

Процедура TREE-PREDECESSOR симметрична.

Таким образом, мы доказали следующую теорему.

Теорема 13.1. *Операции* SEARCH, MINIMUM, MAXIMUM, SUCCESSOR *и* PREDECESSOR *на дереве высоты h выполняются за время $O(h)$.*

Упражнения

13.2-1. Предположим, что в двоичном дереве поиска хранятся числа от 1 до 1000 и мы хотим найти число 363. Какие из следующих последовательностей *не* могут быть последовательностями просматриваемых при этом ключей:

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) $935, 278, 347, 621, 299, 392, 358, 363$?

13.2-2. Пусть поиск ключа в двоичном дереве завершается в листе. Рассмотрим три множества: A (элементы слева от пути поиска), B (элементы на пути) и C (справа от пути). Профессор утверждает, что для любых трёх ключей $a \in A$, $b \in B$ и $c \in C$ верно $a \leqslant b \leqslant c$. Покажите, что он неправ, и приведите контрпример минимально возможного размера.

13.2-3. Докажите формально правильность процедуры TREE-SUCCESSOR.

13.2-4. В разделе 13.1 был построен алгоритм, печатающий все ключи в неубывающем порядке. Теперь это можно сделать иначе: найти минимальный элемент, а потом $n - 1$ раз искать следующий элемент. Докажите, что время работы такого алгоритма есть $O(n)$.

13.2-5. Докажите, что k последовательных вызовов TREE-SUCCESSOR выполняются за $O(k + h)$ шагов (h — высота дерева) независимо от того, с какой вершины мы начинаем.

13.2-6. Пусть T — двоичное дерево поиска, все ключи в котором различны, x — его лист, а y — родитель x. Покажите, что $key[y]$ является соседним с $key[x]$ ключом (следующим или предыдущим в смысле порядка на ключах).

13.3. Добавление и удаление элемента

Эти операции меняют дерево, сохраняя свойство упорядоченности. Как мы увидим, добавление сравнительно просто, а удаление чуть сложнее.

Добавление

Процедура TREE-INSERT добавляет заданный элемент в подходящее место дерева T (сохраняя свойство упорядоченности). Параметром процедуры является указатель z на новую вершину, в которую помещены значения $key[z]$ (добавляемое значение ключа), $left[z] = \text{NIL}$ и $right[z] = \text{NIL}$. В ходе работы процедура меняет дерево T и (возможно) некоторые поля вершины z, после чего новая вершина с данным значением ключа оказывается вставленной в подходящее место дерева.

```
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
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

На рис. 13.3 показано, как работает процедура TREE-INSERT. Подобно процедурам TREE-SEARCH и ITERATIVE-TREE-SEARCH, она двигается вниз по дереву, начав с его корня. При этом в вершине y сохраняется указатель на родителя вершины x (цикл в строках 3–7). Сравнивая $key[z]$ с $key[x]$, процедура решает, куда идти — налево или направо. Процесс завершается, когда x становится равным NIL. Этот NIL стоит как раз там, куда надо поместить z, что и делается в строках 8–13.

Как и остальные операции, добавление требует времени $O(h)$ для дерева высоты h.

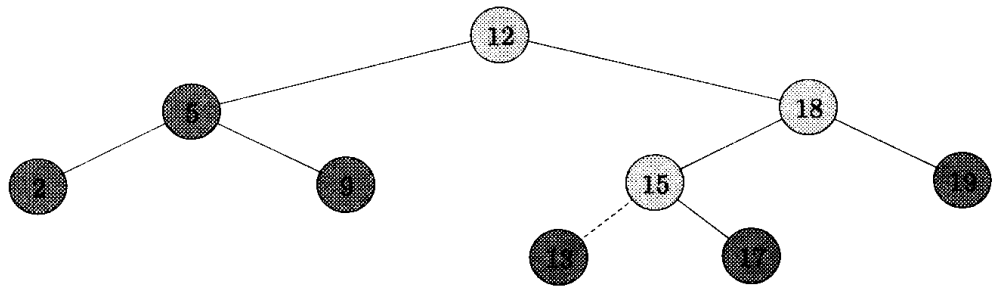

Рис. 13.3. Добавление элемента с ключом 13. Светло-серые вершины находятся на пути от корня до позиции нового элемента. Пунктир связывает новый элемент со старыми.

Удаление

Параметром процедуры удаления является указатель на удаляемую вершину. При удалении возможны три случая, показанные на рис. 13.4. Если у z нет детей, для удаления z достаточно поместить NIL в соответствующее поле его родителя (вместо z). Если у z есть один ребёнок, можно «вырезать» z, соединив его родителя напрямую с его ребёнком. Если же детей двое, требуются некоторые приготовления: мы находим следующий (в смысле порядка на ключах) за z элемент y; у него нет левого ребёнка (упр. 13.3-4). Теперь можно скопировать ключ и дополнительные данные из вершины y в вершину z, а саму вершину y удалить описанным выше способом.

Примерно так и действует процедура TREE-DELETE (хотя рассматривает эти три случая в несколько другом порядке).

```
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
 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          ▷ Копируем дополнительные данные, связанные с y.
17  return y
```

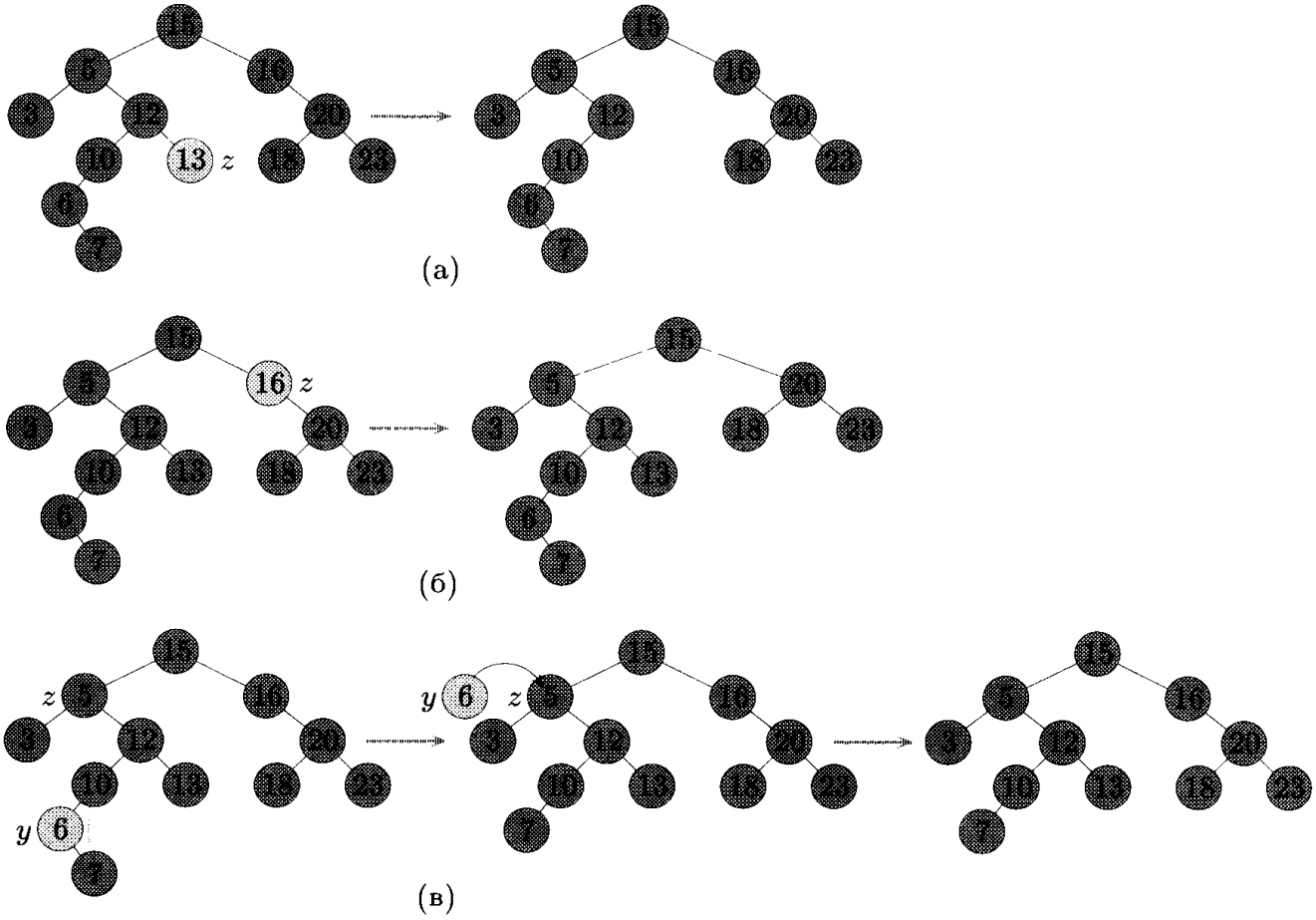

Рис. 13.4. Удаление вершины z из двоичного дерева поиска. (а) Если вершина z не имеет детей, её можно удалить без проблем. (б) Если вершина z имеет одного ребёнка, помещаем его на место вершины z. (в) Если у вершины z двое детей, мы сводим дело к предыдущему случаю, удаляя вместо неё вершину y с непосредственно следующим значением ключа (у этой вершины ребёнок один) и помещая ключ $key[y]$ (и связанные с ним дополнительные данные) на место вершины z.

В строках 1–3 определяется вершина y, которую мы потом вырежем из дерева. Это либо сама вершина z (если у z не более одного ребёнка), либо следующий за z элемент (если у z двое детей). Затем в строках 4–6 переменная x становится указателем на существующего ребёнка вершины y, или равной NIL, если у y нет детей. Вершина y вырезается из дерева в строках 7–13 (меняются указатели в вершинах $p[y]$ и x). При этом отдельно рассматриваются граничные случаи, когда $x = \text{NIL}$ или когда y является корнем дерева. Наконец, в строках 14–16, если вырезанная вершина y отлична от z, ключ (и дополнительные данные) вершины y перемещаются в z (ведь нам надо было удалить z, а не y). Наконец, процедура возвращает указатель y (это позволит вызывающей процедуре впоследствии освободить память, занятую вершиной y). Время выполнения есть $O(h)$ на дереве высоты h.

Итак, мы доказали следующую теорему.

Теорема 13.2. *Операции* INSERT *и* DELETE *могут быть выполнены за время* $O(h)$, *где* h — *высота дерева.*

Упражнения

13.3-1. Напишите рекурсивный вариант процедуры TREE-INSERT.

13.3-2. Начиная с пустого дерева, будем добавлять элементы с различными ключами один за другим. Если после этого мы проводим поиск элемента с ключом x, то число сравнений на единицу больше числа сравнений, выполненных при добавлении этого элемента. Почему?

13.3-3. Набор из n чисел можно отсортировать, сначала добавив их один за другим в двоичное дерево поиска (с помощью процедуры TREE-INSERT), а потом обойти дерево с помощью процедуры INORDER-TREE-WALK. Найдите время работы такого алгоритма в худшем и в лучшем случае.

13.3-4. Покажите, что если вершина двоичного дерева поиска имеет двоих детей, то следующая за ней вершина не имеет левого ребёнка, а предшествующая ей вершина — правого.

13.3-5. Предположим, что указатель на вершину y хранится в какой-то внешней структуре данных и что предшествующая y вершина дерева удаляется с помощью процедуры TREE-DELETE. Какие при этом могут возникнуть проблемы? Как можно изменить TREE-DELETE, чтобы этих проблем избежать?

13.3-6. Коммутируют ли операции удаления двух вершин? Другими словами, получим ли мы одинаковые деревья, если в одном случае удалим сначала x, а потом y, а в другом — наоборот? Объясните свой ответ.

13.3-7. Если у z двое детей, мы можем использовать в TREE-DELETE не следующий за z элемент, а предыдущий. Можно надеяться, что справедливый подход, который в половине случаев выбирает предыдущий, а в половине — следующий элемент, будет приводить к лучше сбалансированному дереву. Как изменить текст процедуры, чтобы реализовать такой подход?