Глава 12. Бинарные деревья поиска

Деревья поиска представляют собой структуры данных, которые поддерживают многие операции с динамическими множествами, включая SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT и DELETE. 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. Однако, если дерево представляет собой линейную цепочку из n узлов, те же операции выполняются в наихудшем случае за время $\Theta(n)$. 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 составляет $O(\lg n)$. 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 с помощью связанной структуры данных, в которой каждый узел является

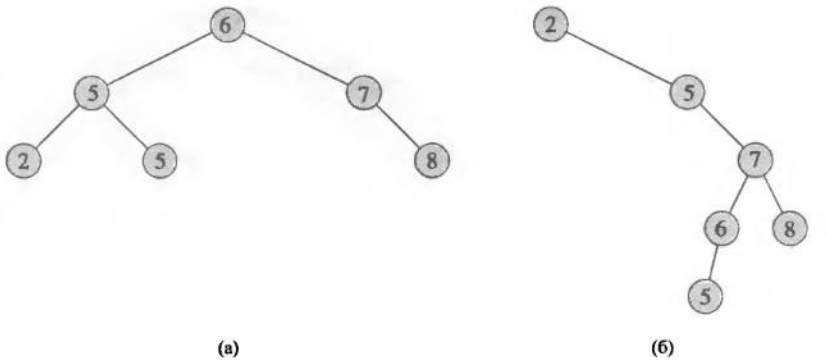


Рис. 12.1. Бинарные деревья поиска. Для любого узла x ключи в левом поддереве x не превышают $x.key$, а ключи в правом поддереве x — не меньше $x.key$. Одно и то же множество значений могут представлять различные бинарные деревья поиска. Время работы в наихудшем случае для большинства операций в дереве поиска пропорционально его высоте. **(а)** Бинарное дерево поиска с 6 узлами высотой 2. **(б)** Менее эффективное бинарное дерево поиска с высотой 4, содержащее те же узлы.

объектом. В дополнение к атрибуту ключа key и сопутствующим данным каждый узел содержит атрибуты $left$, $right$ 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***.

> Пусть x представляет собой узел бинарного дерева поиска. Если y является узлом в левом поддереве x, то $y.key \leq x.key$. Если y является узлом в правом поддереве x, то $y.key \geq x.key$.

Таким образом, на рис. 12.1, (а) ключом корня является значение 6, ключи 2, 5 и 5 в его левом поддереве не превосходят значение 6, а ключи 7 и 8 в его правом поддереве не меньше 6. То же свойство выполняется для каждого узла дерева. Например, ключ 5 в левом дочернем по отношению к корню узле не меньше ключа 2 в его левом поддереве и не больше ключа 5 в его правом поддереве.

Свойство бинарного дерева поиска позволяет 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 и правого поддерева. Имеются и другие способы обхода, а именно — ***обход в прямом порядке*** (preorder tree walk), при котором сначала выводится корень, а затем — значения левого и правого поддеревьев, и ***обход в обратном порядке*** (postorder tree walk). когда первыми выводятся значения левого и правого поддеревьев, а уже затем — корня. Центрированный обход бинарного дерева поиска T реализуется вызовом процедуры INORDER-TREE-WALK$(T.root)$.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

В качестве примера центрированный обход деревьев, показанных на рис. 12.1, выводит ключи в порядке $2, 5, 5, 6, 7, 8$. Корректность описанного алгоритма следует по индукции непосредственно из свойства бинарного дерева поиска.

Для обхода дерева с n узлами 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 с n узлами, то время работы вызова INORDER-TREE-WALK(x) составляет $\Theta(n)$.

Доказательство. Обозначим через $T(n)$ время, необходимое процедуре INORDER-TREE-WALK в случае вызова с параметром, представляющим собой корень дерева с n узлами. Поскольку процедура INORDER-TREE-WALK посещает все n узлов поддерева, мы имеем $T(n) = \Omega(n)$. Остается показать, что $T(n) = O(n)$.

Поскольку INORDER-TREE-WALK требует маленького, фиксированного количества времени для работы с пустым деревом (для выполнения проверки $x \neq$ NIL), мы имеем $T(0) = c$ для некоторой константы $c > 0$.

В случае $n > 0$ будем считать, что процедура INORDER-TREE-WALK вызывается в узле x один раз для левого поддерева с k узлами, а второй — для правого поддерева с $n - k - 1$ узлами. Таким образом, время работы процедуры INORDER-TREE-WALK(x) ограничено $T(n) \leq T(k) + T(n-k-1) + d$ для некоторой константы $d > 0$, которая отражает время, необходимое для выполнения тела процедуры без учета рекурсивных вызовов.

Воспользуемся методом подстановки, чтобы показать, что $T(n) = O(n)$, путем доказательства того, что $T(n) \leq (c+d)n + c$. Для $n = 0$ мы имеем $(c+d) \cdot 0 + c = c = T(0)$. Для $n > 0$ мы имеем

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

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 (с. 181)? 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, но 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 обходы 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 времени $\Omega(n \lg n)$, любой основанный на сравнениях алгоритм построения бинарного дерева поиска из произвольного списка, содержащего n элементов, также требует в худшем случае времени $\Omega(n \lg n)$.

12.2. Работа с бинарным деревом поиска

Наиболее частой операцией, выполняемой с бинарным деревом поиска, является поиск в нем определенного ключа. Помимо операции SEARCH, бинарные деревья поиска поддерживают такие запросы, как MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR. 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 приведенная ниже 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).

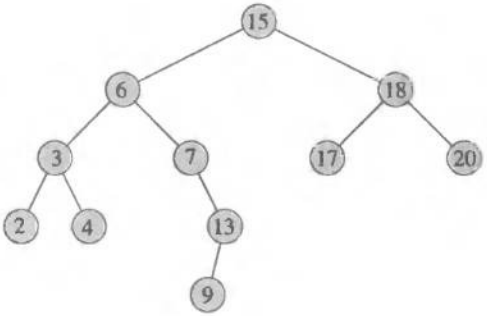


Рис. 12.2. Запросы к бинарному дереву поиска. Для поиска ключа 13 необходимо пройти путь $15 \to 6 \to 7 \to 13$ от корня. Минимальным в дереве является ключ 2, который находится при следовании по указателям *left* от корня. Максимальный ключ 20 достигается при следовании от корня по указателям *right*. Последующим узлом после узла с ключом 15 является узел с ключом 17, поскольку это минимальный ключ в правом поддереве узла 15. Узел с ключом 13 не имеет правого поддерева, так что следующим за ним является наименьший предок, левый наследник которого также является предком данного узла. В нашем случае это узел с ключом 15.

TREE-SEARCH(x, k)

```
1  if x == NIL или k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Процедура поиска начинается с корня дерева и проходит вниз по простому пути по дереву, как показано на рис. 12.2. Для каждого встреченного на пути вниз узла x его ключ $x.key$ сравнивается с переданным в качестве параметра ключом k. Если ключи одинаковы, поиск завершается. Если k меньше $x.key$, поиск продолжается в левом поддереве x, так как из свойства бинарного дерева поиска понятно, что искомый ключ не может находиться в правом поддереве. Аналогично, если k больше $x.key$, поиск продолжается в правом поддереве x. Узлы, которые мы посещаем при рекурсивном поиске, образуют простой 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, "развернув" оконечную рекурсию в цикл **while**. На большинстве компьютеров такая версия оказывается более эффективной.

ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL и k ≠ x.key
2       if k < x.key
3            x = x.left
4       else x = x.right
5  return x
```

Минимум и максимум

Элемент с минимальным значением ключа всегда можно найти, следуя по дочерним указателям *left* от корневого узла до тех пор, пока не встретится значение NIL, как показано на рис. 12.2. Приведенная ниже процедура возвращает указатель на минимальный элемент поддерева с корнем в данном узле x, который предполагается не равным NIL.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
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 $x.key$, минимальный ключ поддерева с корнем в узле x находится в узле $x.key$. Если же у узла x есть левое поддерево, то, поскольку в правом поддереве не может быть узла с ключом, меньшим $x.key$, а все ключи в узлах левого поддерева не превышают $x.key$, узел с минимальным значением ключа в поддереве с корнем x находится в поддереве, корнем которого является узел $x.left$.

Псевдокод процедуры TREE-MAXIMUM симметричен.

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Обе эти 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 простой нисходящий путь от корня дерева.

Предшествующий и последующий элементы

Иногда для заданного узла 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 $x.key$. Структура бинарного дерева поиска позволяет 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 x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL и x == y.right
5      x = y
6      y = y.p
7  return y
```

Код процедуры TREE-SUCCESSOR разбивается на две части. Если правое поддерево узла x непустое, то следующий за x элемент является крайним слева узлом в правом поддереве x, который выявляется в строке 2 вызовом процедуры TREE-MINIMUM$(x.right)$. Например, на рис. 12.2 следующим за узлом с ключом 15 является узел с ключом 17.

С другой стороны, как требуется показать в упр. 12.2.6, если правое поддерево узла x пустое и у x имеется следующий за ним элемент y, то y является наименьшим предком x, левый наследник которого 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 от x до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя. Это действие выполняется в строках 3–7 процедуры TREE-SUCCESSOR.

Время работы алгоритма TREE-SUCCESSOR в дереве высотой h составляет $O(h)$, поскольку мы либо движемся по простому пути вниз от исходного узла, либо по простому пути вверх. Процедура поиска предшествующего узла в дереве TREE-PREDECESSOR симметрична процедуре TREE-SUCCESSOR и также имеет время работы $O(h)$.

Даже если в дереве имеются узлы с одинаковыми ключами, мы можем просто определить последующий и предшествующий узлы как такие, которые возвращаются процедурами TREE-SUCCESSOR(x) и TREE-PREDECESSOR(x) соответственно.

Таким образом, мы доказали следующую теорему.

Теорема 12.2

Операции SEARCH, MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR над динамическим множеством могут быть реализованы таким образом, что их время выполнения равно $O(h)$ в бинарном дереве поиска высотой h. ■

Упражнения

12.2.1

Предположим, что 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 $a \leq b \leq c$. 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, левый дочерний узел которого также является предком x. (Вспомните, что каждый узел является собственным предком.)

12.2.7

Альтернативный центрированный обход бинарного дерева поиска с n узлами можно осуществить путем поиска минимального элемента дерева с помощью 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, для k последовательных вызовов процедуры TREE-SUCCESSOR потребуется время $O(k + h)$.

12.2.9

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

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, атрибуты которого $z.key = v$, $z.left = \text{NIL}$ и $z.right = \text{NIL}$. Процедура таким образом изменяет T и некоторые поля z, что z оказывается вставленным в корректную позицию дерева.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // Дерево T было пустым
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

На рис. 12.3 показано, как работает процедура TREE-INSERT. Подобно процедурам TREE-SEARCH и ITERATIVE-TREE-SEARCH, процедура TREE-INSERT начинает работу с корневого узла дерева и проходит по простому нисходящему пути. Указатель x отмечает путь, проходимый в поисках NIL, который должен быть заменен входным элементом z. Процедура поддерживает также ***замыкающий указатель*** (trailing pointer) y, который представляет собой указатель на родительский по отношению к x 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 $z.key$ и $x.key$, до тех пор, пока x не станет равным NIL. Это значение находится именно в той позиции, в которую следует поместить элемент z. Замыкающий указатель y нужен для того, чтобы знать, какой узел должен быть изменен. В строках 8–13 выполняется установка значений указателей, обеспечивающая вставку элемента z.

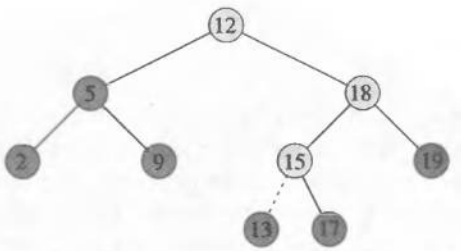


Рис. 12.3. Вставка элемента с ключом 13 в бинарное дерево поиска. Светлые узлы указывают простой путь от корня вниз по дереву в позицию, в которую должен быть вставлен новый элемент. Пунктирная линия указывает добавляемую в дерево связь, обеспечивающую вставку элемента.

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

Удаление

Стратегия удаления узла z из бинарного дерева поиска T имеет три основные ситуации; как мы увидим, одна из ситуаций оказывается достаточно сложной.

- Если у z нет дочерних узлов, то мы просто удаляем его, внося изменения в его родительский узел, а именно — заменяя дочерний узел z на NIL.
- Если у z только один дочерний узел, то мы удаляем узел z, создавая новую связь между родительским и дочерним узлами узла z.
- Если у узла z два дочерних узла, то мы находим следующий за ним узел y, который должен находиться в правом поддереве z и который занимает в дереве место z. Остаток исходного правого поддерева z становится новым поддеревом y, а левое поддерево z становится новым левым поддеревом y. Это самый сложный случай, поскольку, как мы увидим, здесь играет роль, является ли y правым дочерним узлом z.

Процедура удаления данного узла z из бинарного дерева поиска T получает в качестве аргумента указатели на T и на z. Она организована несколько иначе, чем описано ранее, и рассматривает не три, а четыре случая, показанные на рис. 12.4.

- Если z не имеет левого дочернего узла (см. рис. 12.4, (а)), то мы заменяем z его правым дочерним узлом, который может быть (или не быть) NIL. Если правый дочерний узел z представляет собой NIL, то мы оказываемся в ситуации, когда у узла z нет дочерних узлов. Если правый дочерний узел z отличен от NIL, то мы оказываемся в ситуации, когда у узла z имеется единственный дочерний узел, являющийся правым дочерним узлом.
- Если z имеет только один дочерний узел, являющийся его левым дочерним узлом (см. рис. 12.4, (б)), то мы заменяем z его левым дочерним узлом.
- В противном случае z имеет и левый, и правый дочерние узлы. Мы находим узел y, следующий за z. Этот узел располагается в правом поддереве z и не

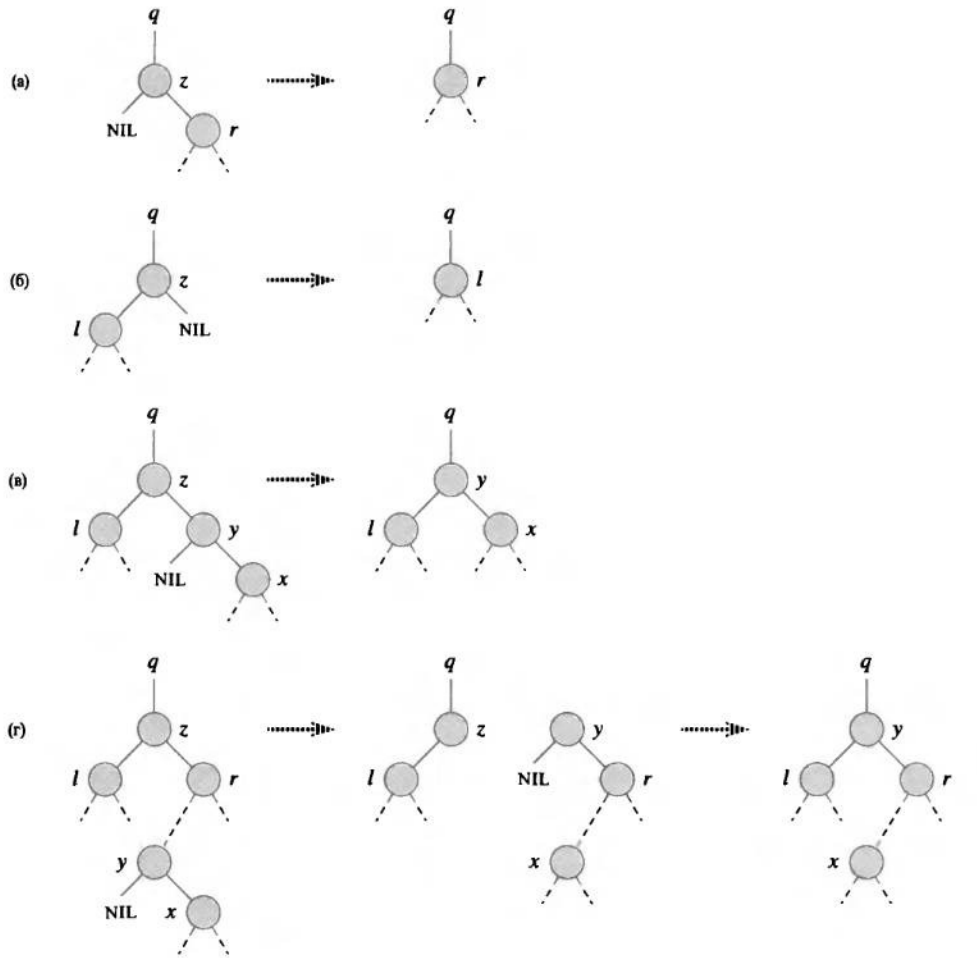


Рис. 12.4. Удаление узла z из бинарного дерева поиска. Узел z может быть корнем, левым дочерним узлом узла q или правым дочерним узлом q. **(а)** Узел z не имеет левого дочернего узла. Мы заменяем z его правым дочерним узлом r, который может быть (а может и не быть) значением NIL. **(б)** Узел z имеет левый дочерний узел l, но не имеет правого дочернего узла. Мы заменяем z на l. **(в)** Узел z имеет два дочерних узла; его левый дочерний узел — l, а правый дочерний узел y является следующим за z узлом дерева; правый дочерний по отношению к y узел — x. Мы заменяем z на y, обновляем левый дочерний узел y (им становится l), но оставляем x правым дочерним узлом y. **(г)** Узел z имеет два дочерних узла (левый дочерний узел l и правый дочерний узел r), а следующий за z узел $y \neq r$ находится в поддереве, корнем которого является r. Мы заменяем y его собственным правым дочерним узлом x и устанавливаем y в качестве родительского по отношению к r узла. Затем мы устанавливаем y в качестве дочернего узла q и родительского узла l.

имеет левого дочернего узла (см. упр. 12.2.5). Мы хотим вырезать y из его текущего положения и заменить им в дереве узел z.

- Если y является правым дочерним узлом z (см. рис. 12.4, (в)), то мы заменяем z на y, оставляя нетронутым правый дочерний по отношению к y узел.
- В противном случае y находится в правом поддереве узла z, но не является правым дочерним узлом z (см. рис. 12.4, (г)). В этом случае мы сначала заменяем y его собственным правым дочерним узлом, а затем заменяем z на y.

Для перемещения поддеревьев в бинарном дереве поиска мы определяем подпрограмму TRANSPLANT, которая заменяет одно поддерево, являющееся дочерним по отношению к своему родителю, другим поддеревом. Когда TRANSPLANT заменяет поддерево с корнем в узле u поддеревом с корнем в узле v, родитель узла u становится родителем узла v, который становится соответствующим дочерним узлом родительского по отношению к u узла.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

В строках 1 и 2 обрабатывается случай, когда u является корнем T. В противном случае u является либо левым, либо правым дочерним узлом своего родителя. В строках 3 и 4 выполняется обновление $u.p.left$, если u является левым дочерним узлом, в строке 5 обновляется $u.p.right$, если u является правым дочерним узлом. Мы допускаем значение NIL для узла v, и строки 6 и 7 обновляют $v.p$, если v не NIL. Заметим, что TRANSPLANT не пытается обновлять $v.left$ и $v.right$; выполнение этих действий (или их не выполнение) находится в зоне ответственности вызывающей подпрограмму TRANSPLANT процедуры.

Имея подпрограмму TRANSPLANT, мы можем реализовать процедуру удаления узла z из бинарного дерева поиска T следующим образом.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Процедура TREE-DELETE работает следующим образом. Строки 1 и 2 обрабатывают случай, когда у узла z нет левого дочернего узла, а строки 3 и 4 — когда у z есть левый дочерний узел, но нет правого. Строки 5–12 работают с оставшимися двумя случаями, когда у z есть два дочерних узла. Строка 5 находит узел y, следующий за z. Поскольку z имеет непустое правое поддерево, следующий за z узел должен быть узлом этого поддерева с наименьшим ключом; поэтому поиск узла y осуществляется вызовом TREE-MINIMUM$(z.right)$. Как отмечалось ранее, у y нет левого дочернего узла. Мы хотим вырезать y из его текущего положения, после чего этот узел должен заменить в дереве узел z. Если y является правым дочерним узлом z, то строки 10–12 заменяют z как дочерний узел его родителя на y и заменяют левый дочерний узел y левым дочерним узлом z. Если y не является правым дочерним узлом z, в строках 7–9 выполняется замена y как дочернего узла своего родителя правым дочерним по отношению к y узлом, и преобразование правого дочернего узла z в правый дочерний узел y, после чего строки 10–12 заменяют z как дочерний узел его родителя узлом y, а левым дочерним узлом y становится левый дочерний узел z.

Каждая строка TREE-DELETE, включая вызовы TRANSPLANT, выполняется за константное время, за исключением вызова TREE-MINIMUM в строке 5. Таким образом, TREE-DELETE выполняется за время $O(h)$ в дереве высотой h.

Таким образом, доказана следующая теорема.

Теорема 12.3

Операции над динамическим множеством INSERT и DELETE можно реализовать таким образом, что каждая из них выполняется в бинарном дереве поиска высотой h 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 заданное множество из n чисел можно следующим образом: сначала построить бинарное дерево поиска, содержащее эти числа (многократно вызывая процедуру TREE-INSERT для вставки чисел в дерево одно за другим), а затем выполнить центрированный обход полученного дерева. Чему равно время работы такого алгоритма в наилучшем и наихудшем случаях?

12.3.4

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? Поясните, почему это так, или приведите контрпример.

12.3.5

Предположим, что вместо поддержки в каждом узле x атрибута $x.p$, указывающего на родительский узел x, поддерживается атрибут $x.succ$, указывающий на узел, следующий за x. Запишите псевдокод процедур SEARCH, INSERT и DELETE для бинарного дерева поиска, использующего такое представление. Эти процедуры должны выполняться за время $O(h)$, где h — высота дерева T. (*Указание:* можно реализовать подпрограмму для поиска узла, родительского по отношению к данному.)

12.3.6

Если узел z в процедуре TREE-DELETE имеет два дочерних узла, можно выбрать не следующий за ним узел y, а предшествующий ему. Какие при этом следует внести изменения в процедуру TREE-DELETE? Есть также мнение, что стратегия, которая состоит в равновероятном выборе предшествующего или последующего узла, приводит к большей производительности. Каким образом следует изменить процедуру TREE-DELETE для реализации указанной стратегии?

⋆ 12.4. Случайное построение бинарных деревьев поиска

Мы показали, что все базовые операции с бинарными деревьями поиска имеют время выполнения $O(h)$, где h — высота дерева. Однако при вставке и удалении элементов высота дерева меняется. Если, например, все элементы вставляются в дерево в строго возрастающей последовательности, то такое дерево вырождается в цепочку высотой $n - 1$. С другой стороны, как показано в упр. Б.5.4. $h \geq \lfloor \lg n \rfloor$. Как и в случае быстрой сортировки, можно показать, что поведение

алгоритма в среднем случае гораздо ближе к наилучшему случаю, чем к наихудшему.

К сожалению, в ситуации, когда при формировании бинарного дерева поиска используются и вставки, и удаления, о средней высоте образующихся деревьев известно мало, так что мы ограничимся анализом ситуации, когда дерево строится только с использованием вставок, без удалений. Определим ***случайно построенное бинарное дерево поиска*** (randomly built binary search tree) с n ключами как дерево, которое возникает при вставке ключей в изначально пустое дерево в случайном порядке, когда все $n!$ перестановок входных ключей равновероятны (в упр. 12.4.3 требуется показать, что это условие отличается от условия равновероятности всех возможных бинарных деревьев поиска с n узлами). В данном разделе мы докажем следующую теорему.

Теорема 12.4

Математическое ожидание высоты случайно построенного бинарного дерева поиска с n различными ключами равно $O(\lg n)$.

Доказательство. Начнем с определения трех случайных величин, которые помогут определить высоту случайного бинарного дерева поиска. Обозначая высоту случайного бинарного дерева поиска с n ключами как X_n, определим ***экспоненциальную высоту*** $Y_n = 2^{X_n}$. При построении бинарного дерева поиска с n ключами мы выбираем один из них в качестве корня. Обозначим через R_n случайную величину, равную ***рангу*** корневого ключа в множестве из всех n ключей, т.е. R_n содержит позицию, которую бы занимал ключ, если бы множество было отсортировано. Значение R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$. Если $R_n = i$, то левое поддерево корня представляет собой случайно построенное бинарное дерево поиска с $i - 1$ ключами, а правое — с $n - i$ ключами. Поскольку высота бинарного дерева на единицу больше наибольшей из высот поддеревьев корневого узла, экспоненциальная высота бинарного дерева в два раза больше экспоненциальной высоты наивысшего из поддеревьев корневого узла. Если мы знаем, что $R_n = i$, то

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) \ .$$

В качестве базового случая мы имеем $Y_1 = 1$, поскольку экспоненциальная высота дерева с одним узлом составляет $2^0 = 1$, и для удобства мы определим $Y_0 = 0$.

Далее мы определим индикаторные случайные величины $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, где

$$Z_{n,i} = \mathrm{I}\{R_n = i\} \ .$$

Поскольку R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$, $\Pr\{R_n = i\} = 1/n$ для $i = 1, 2, \ldots, n$, а следовательно, согласно лемме 5.1 мы имеем

$$\mathrm{E}[Z_{n,i}] = 1/n \tag{12.1}$$

для $i = 1, 2, \ldots, n$. Поскольку ровно одно значение $Z_{n,i}$ равно 1, а все прочие равны 0, мы также имеем

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right) .$$

Мы покажем, что $\mathrm{E}[Y_n]$ полиномиально зависит от n, что неизбежно приводит к выводу, что $\mathrm{E}[X_n] = O(\lg n)$.

Мы утверждаем, что индикаторная случайная переменная $Z_{n,i} = \mathrm{I}\{R_n = i\}$ не зависит от значений Y_{i-1} и Y_{n-i}. Если $R_n = i$, то левое поддерево, экспоненциальная высота которого равна Y_{i-1}, случайным образом строится из $i - 1$ ключей, ранги которых меньше i. Это поддерево ничем не отличается от любого другого случайного бинарного дерева поиска из $i - 1$ ключей. Выбор $R_n = i$ никак не влияет на структуру этого дерева, а влияет только на количество содержащихся в нем узлов. Следовательно, случайные величины Y_{i-1} и $Z_{n,i}$ независимы. Аналогично правое поддерево, экспоненциальная высота которого равна Y_{n-i}, строится случайным образом из $n - i$ ключей, ранги которых больше i. Структура этого дерева не зависит от R_n, так что случайные величины Y_{n-i} и $Z_{n,i}$ независимы. Следовательно, мы имеем

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] && \text{(из линейности математического ожидания)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}] \, \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(из независимости)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(согласно (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}\left[\max(Y_{i-1}, Y_{n-i})\right] && \text{(согласно (B.22))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(из упр. B.3.4)} .
\end{aligned}$$

Поскольку каждый член $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ появляется в последней сумме дважды, как $\mathrm{E}[Y_{i-1}]$ и как $\mathrm{E}[Y_{n-i}]$, мы получаем следующее рекуррентное соотношение:

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] . \tag{12.2}$$

Используя метод подстановок, покажем, что для всех натуральных n рекуррентное соотношение (12.2) имеет следующее решение:

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3} .$$

При этом мы воспользуемся тождеством

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4} . \tag{12.3}$$

(В упр. 12.4.1 данное тождество предлагается доказать самостоятельно.)

Заметим, что для базовых случаев границы $0 = Y_0 = \mathrm{E}[Y_0] \le (1/4)\binom{3}{3} = 1/4$ и $1 = Y_1 = \mathrm{E}[Y_1] \le (1/4)\binom{1+3}{3} = 1$ справедливы. По индукции имеем

$$\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&\le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(согласно гипотезе индукции)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(согласно (12.3))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3} .
\end{aligned}$$

Мы получили границу для $\mathrm{E}[Y_n]$, но наша конечная цель — найти границу $\mathrm{E}[X_n]$. В упр. 12.4.4 требуется показать, что функция $f(x) = 2^x$ выпуклая вниз (см. с. 1251). Таким образом, мы можем применить неравенство Йенсена (B.26), которое гласит, что

$$\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \mathrm{E}\left[2^{X_n}\right] \\
&= \mathrm{E}[Y_n] ,
\end{aligned}$$

и получить

$$\begin{aligned} 2^{\mathrm{E}[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\ &= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\ &= \frac{n^3+6n^2+11n+6}{24} . \end{aligned}$$

Взятие логарифма от обеих частей дает нам $\mathrm{E}[X_n] = O(\lg n)$. ■

Упражнения

12.4.1

Докажите уравнение (12.3).

12.4.2

Приведите пример бинарного дерева поиска с n узлами, такого, что средняя глубина узла в дереве равна $\Theta(\lg n)$, в то время как высота дерева — $\omega(\lg n)$. Найдите асимптотическую верхнюю границу высоты бинарного дерева поиска с n узлами, средняя глубина узла в котором составляет $\Theta(\lg n)$.

12.4.3

Покажите, что понятие случайного бинарного дерева поиска с n ключами, когда выбор каждого дерева равновероятен, отличается от понятия случайно построенного бинарного дерева поиска, приведенного в этом разделе. (*Указание:* рассмотрите все возможные деревья при $n = 3$.)

12.4.4

Покажите, что функция $f(x) = 2^x$ является выпуклой вниз.

12.4.5 ⋆

Рассмотрим процедуру RANDOMIZED-QUICKSORT, работающую с входной последовательностью из n различных чисел. Докажите, что для любой константы $k > 0$ время работы алгоритма превышает $O(n \lg n)$ только для $O(1/n^k)$-й части всех возможных $n!$ перестановок.

Задачи

12.1. Бинарные деревья поиска с одинаковыми ключами

Одинаковые ключи приводят к проблеме при реализации бинарных деревьев поиска.

a. Чему равно асимптотическое время работы процедуры TREE-INSERT при вставке n одинаковых ключей в изначально пустое дерево?

Мы предлагаем улучшить алгоритм TREE-INSERT, добавив в него перед строкой 5 проверку равенства $z.key = x.key$, а перед строкой 11 — проверку равенства $z.key = y.key$. Если равенства выполняются, мы реализуем одну из описанных далее стратегий. Для каждой из них найдите асимптотическое время работы при вставке n одинаковых ключей в изначально пустое дерево. (Описания приведены для строки 5, в которой сравниваются ключи z и x. Для строки 11 замените в описании x на y.)

б. Храним в узле x булев флаг $x.b$ и устанавливаем x равным либо $x.left$, либо $x.right$, в зависимости от значения $x.b$, которое поочередно принимает значения FALSE и TRUE при каждом посещении x в процессе вставки узла с тем же ключом, что и у x.

в. Храним все элементы с одинаковыми ключами в одном узле x с помощью списка и при вставке просто добавляем элемент z в этот список.

г. Случайным образом присваиваем x значение $x.left$ или $x.right$. (Каково будет время работы такой стратегии в наихудшем случае? Оцените ожидаемое время работы данной стратегии.)

12.2. Цифровые деревья

Пусть имеются две строки $a = a_0 a_1 \ldots a_p$ и $b = b_0 b_1 \ldots b_q$, в которых все символы a_i и b_j принадлежат некоторому упорядоченному множеству. Мы говорим, что строка a ***лексикографически меньше*** строки b, если выполняется одно из двух условий:

1. существует целое $0 \le j \le \min(p, q)$, такое, что $a_i = b_i$ для всех $i = 0, 1, \ldots, j - 1$ и $a_j < b_j$, или
2. $p < q$ и $a_i = b_i$ для всех $i = 0, 1, \ldots, p$.

Например, если a и b представляют собой битовые строки, то $10100 < 10110$ согласно правилу 1 (полагая $j = 3$), а $10100 < 101000$ согласно правилу 2. Это упорядочение подобно упорядочению слов в словаре естественного языка по алфавиту.

Структура данных ***цифрового дерева*** (radix tree), показанная на рис. 12.5, хранит битовые строки 1011, 10, 011, 100 и 0. При поиске ключа на глубине i мы переходим к левому узлу, если $a_i = 0$, и к правому, если $a_i = 1$. Пусть S — множество различных битовых строк с суммарной длиной n. Покажите, как использовать цифровое дерево для лексикографической сортировки за время $\Theta(n)$. В примере, приведенном на рис. 12.5, отсортированная последовательность должна выглядеть следующим образом: 0, 011, 10, 100, 1011.

12.3. Средняя глубина вершины в случайно построенном бинарном дереве поиска

В данной задаче мы докажем, что средняя глубина узла в случайно построенном бинарном дереве поиска с n узлами равна $O(\lg n)$. Хотя этот результат и является более слабым, чем в теореме 12.4, способ доказательства демонстрирует

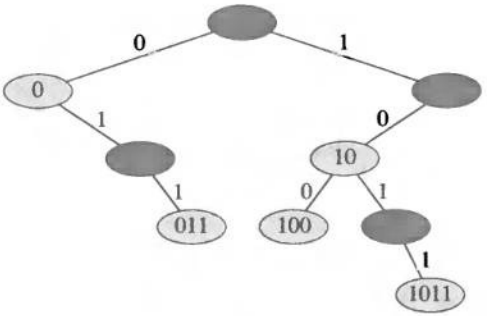


Рис. 12.5. Цифровое дерево с битовыми строками 1011, 10, 011, 100 и 0. Ключ каждого узла можно определить путем прохода по простому пути от корня к этому узлу. Следовательно, нет необходимости хранить ключи в узлах; здесь значения ключей приведены только в иллюстративных целях. Узлы заштрихованы темным цветом, если соответствующие им ключи отсутствуют в дереве; такие узлы существуют только для установления путей к другим узлам.

интересные аналогии между построением бинарного дерева поиска и работой алгоритма RANDOMIZED-QUICKSORT из раздела 7.3.

Определим ***общую длину путей*** $P(T)$ бинарного дерева T как сумму глубин всех узлов $x \in T$, которую мы будем обозначать как $d(x, T)$.

а. Покажите, что средняя глубина узла в дереве T равна

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n}P(T) \ .$$

Таким образом, нужно показать, что математическое ожидание $P(T)$ равно $O(n \lg n)$.

б. Обозначим через T_L и T_R соответственно левое и правое поддеревья дерева T. Покажите, что если дерево T имеет n узлов, то

$$P(T) = P(T_L) + P(T_R) + n - 1 \ .$$

в. Обозначим через $P(n)$ среднюю общую длину путей случайно построенного бинарного дерева поиска с n узлами. Покажите, что

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n - i - 1) + n - 1) \ .$$

г. Покажите, как переписать $P(n)$ как

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) \ .$$

д. Вспоминая альтернативный анализ рандомизированной версии алгоритма быстрой сортировки из задачи 7.3, сделайте вывод о том, что $P(n) = O(n \lg n)$.

В каждом рекурсивном вызове быстрой сортировки мы выбираем случайный опорный элемент для разделения множества сортируемых элементов. Каждый узел в бинарном дереве поиска также отделяет часть элементов, попадающих в поддерево, для которого данный узел является корневым.

е. Опишите реализацию алгоритма быстрой сортировки, в которой в процессе сортировки множества элементов выполняются те же сравнения, что и для вставки элементов в бинарное дерево поиска. (Порядок, в котором выполняются сравнения, может быть иным, однако сами множества сравнений должны совпадать.)

12.4. Количество различных бинарных деревьев

Обозначим через b_n количество различных бинарных деревьев с n узлами. В этой задаче будет выведена формула для b_n и найдена ее асимптотическая оценка.

а. Покажите, что $b_0 = 1$ и что для $n \geq 1$

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} \ .$$

б. Пусть $B(x)$ — производящая функция (определение производящей функции можно найти в задаче 4.4)

$$B(x) = \sum_{n=0}^{\infty} b_n x^n \ .$$

Покажите, что $B(x) = xB(x)^2 + 1$, и, следовательно, $B(x)$ можно записать в аналитическом виде как

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1 - 4x}\right) \ .$$

Разложение в ряд Тейлора функции $f(x)$ вблизи точки $x = a$ определяется как

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x - a)^k \ ,$$

где $f^{(k)}(x)$ — k-я производная f в точке x.

в. Покажите, что

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(n-е ***число Каталана***), используя разложение в ряд Тейлора функции $\sqrt{1-4x}$ вблизи точки $x = 0$. (Если хотите, вместо разложения в ряд Тейлора используйте обобщение биномиального разложения (B.4) для нецелого показателя степени n, когда для любого действительного числа n и целого k выражение $\binom{n}{k}$ интерпретируется как $n(n-1)\cdots(n-k+1)/k!$ при $k \geq 0$ и как 0 — в противном случае.)

г. Покажите, что

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right) .$$

Заключительные замечания

Подробное рассмотрение простых бинарных деревьев поиска (которые были независимо открыты рядом исследователей в конце 1950-х годов) и их различных вариаций имеется в книге Кнута (Knuth) [210][1]. Там же рассматриваются и цифровые деревья. Цифровые деревья часто называют "лучами" ("tries"), термином, образованным из средних букв слова "по***луч***ение" ("re***trie***val").

Во многих книгах, включая два предыдущих издания данной книги, приводится несколько более простой метод удаления узла из бинарного дерева поиска при наличии обоих дочерних узлов. Вместо замены узла z следующим за ним узлом y мы удаляем узел y, но копируем его ключ и сопутствующие данные в узел z. Недостатком такого подхода является то, что в действительности удаляемый узел может не совпадать с узлом, передаваемым процедуре удаления. Если другие компоненты программы работают с указателями на узлы дерева, в результате они могут оказаться с устаревшими указателями на реально удаленные узлы. Хотя метод удаления, представленный в этом издании книги, немного сложнее использовавшегося ранее, он гарантирует, что вызов для удаления узла z в действительности удалит узел z и только его.

В разделе 15.5 будет показано, каким образом можно построить оптимальное бинарное дерево поиска, если частоты поисков известны заранее, до начала построения дерева. В этом случае дерево строится таким образом, чтобы при наиболее частых поисках просматривалось минимальное количество узлов дерева.

[1] Имеется русский перевод: Д. Кнут. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: И.Д. "Вильямс", 2000.

Глава 13. Красно-черные деревья

В главе 12 было показано, что бинарные деревья поиска высоты h реализуют все базовые операции над динамическими множествами, такие как SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT и DELETE, со временем работы $O(h)$. Таким образом, операции выполняются тем быстрее, чем меньше высота дерева. Однако в наихудшем случае производительность бинарного дерева поиска оказывается ничуть не лучшей, чем производительность связанного списка. Красно-черные деревья представляют собой одну из множества "сбалансированных" схем деревьев поиска, которые гарантируют время выполнения операций над динамическим множеством $O(\lg n)$ даже в наихудшем случае.

13.1. Свойства красно-черных деревьев

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом ***цвета*** в каждом узле. Цвет узла может быть либо красным (RED), либо черным (BLACK). В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно ***сбалансированными***.

Каждый узел дерева содержит атрибуты *color*, *key*, *left*, *right* и *p*. Если не существует дочернего или родительского узла по отношению к данному, соответствующий указатель принимает значение NIL. Мы будем рассматривать эти значения NIL как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все "нормальные" узлы, содержащие поле ключа, становятся внутренними узлами дерева.

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим ***красно-черным свойствам***.

1. Каждый узел является либо красным, либо черным.
2. Корень дерева является черным узлом.
3. Каждый лист дерева (NIL) является черным узлом.
4. Если узел красный, то оба его дочерних узла черные.

5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

На рис. 13.1, (а) приведен пример красно-черного дерева.

Для удобства работы с граничными условиями в красно-черных деревьях мы заменим все листья одним ограничивающим узлом, представляющим значение NIL (с этим приемом мы уже встречались — см. с. 270). В красно-черном дереве T ограничитель $T.nil$ представляет собой объект с теми же атрибутами, что и обычный узел дерева. Значение *color* этого узла равно BLACK (черный), а все остальные атрибуты — p, *left*, *right* и *key* — могут иметь произвольные значения. Как показано на рис. 13.1, (б), все указатели на NIL заменяются указателями на ограничитель $T.nil$.

Использование ограничителя позволяет нам рассматривать дочерний по отношению к узлу x NIL как обычный узел, родителем которого является узел x. Хотя можно было бы использовать различные ограничители для каждого значения NIL (что позволило бы точно определять их родительские узлы), этот подход привел бы к неоправданному перерасходу памяти. Вместо этого мы используем единственный ограничитель $T.nil$ для представления всех NIL — как листьев, так и родительского узла корня. Значения атрибутов p, *left*, *right* и *key* ограничителя не играют никакой роли, хотя для удобства мы можем присвоить им те или иные значения.

В целом мы ограничим наш интерес к красно-черным деревьям только их внутренними узлами, поскольку лишь они хранят значения ключей. В оставшейся части данной главы при изображении красно-черных деревьев все листья опускаются, как это сделано на рис. 13.1, (в).

Количество черных узлов на любом простом пути от узла x (не считая сам узел) к листу будем называть ***черной высотой*** (black-height) узла и обозначать как $\mathrm{bh}(x)$. В соответствии со свойством 5 красно-черных деревьев черная высота узла — точно определяемое значение, поскольку все нисходящие простые пути из узла содержат одно и то же количество черных узлов. Черной высотой дерева будем считать черную высоту его корня.

Следующая лемма показывает, почему красно-черные деревья хорошо использовать в качестве деревьев поиска.

Лемма 13.1

Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\lg(n+1)$.

Доказательство. Начнем с того, что покажем, что поддерево любого узла x содержит как минимум $2^{\mathrm{bh}(x)} - 1$ внутренних узлов. Докажем это по индукции по высоте x. Если высота x равна 0, то узел x должен быть листом ($T.nil$), а поддерево узла x содержит не менее $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов. Теперь для выполнения шага индукции рассмотрим узел x, который имеет положительную высоту и представляет собой внутренний узел с двумя потомками. Каждый дочерний узел имеет черную высоту либо $\mathrm{bh}(x)$, либо $\mathrm{bh}(x) - 1$ в зависимости от того, является ли его цвет соответственно красным или черным. Поскольку

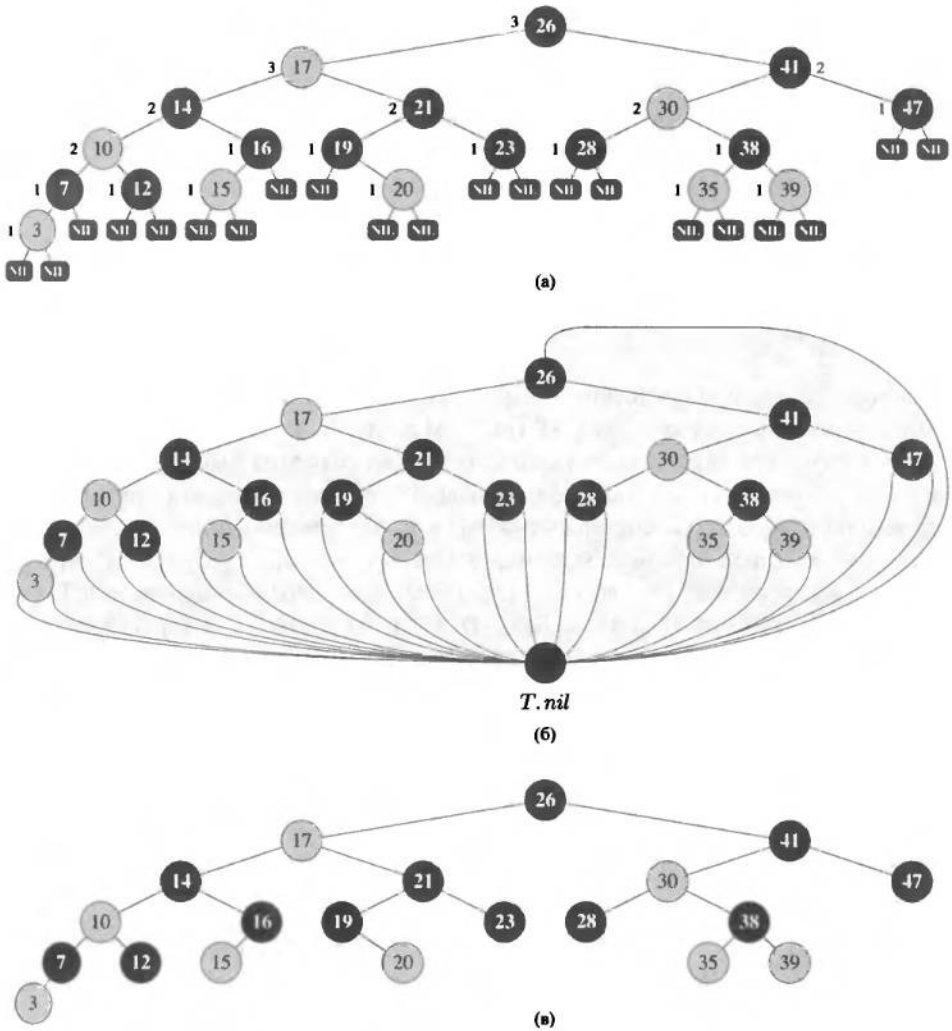


Рис. 13.1. Красно-черное дерево с более темными черными узлами и более светлыми красными. Каждый узел красно-черного дерева либо красный, либо черный; оба дочерних узла красного узла – черные, и количество черных узлов одинаково на каждом простом пути от узла до его наследника-листа. **(а)** Каждый лист, показанный как NIL, черный. Каждый не-NIL узел помечен его черной высотой; листья NIL имеют черную высоту 0. **(б)** То же красно-черное дерево, но все NIL в нем заменены единственным ограничителем $T.nil$, который всегда черный, а черные высоты узлов опущены. Родителем корневого узла также является ограничитель. **(в)** То же красно-черное дерево, но с опущенными листьями и родителем корневого узла. Именно этот стиль изображения красно-черных деревьев будет использоваться далее в этой главе.

высота потомка x меньше, чем высота самого узла x, мы можем использовать предположение индукции и сделать вывод о том, что каждый из потомков x имеет как минимум $2^{\mathrm{bh}(x)-1} - 1$ внутренних узлов. Таким образом, дерево с корнем в вершине x содержит как минимум $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ внутренних узлов, что и доказывает наше утверждение.

Для того чтобы завершить доказательство леммы, обозначим высоту дерева через h. Согласно свойству 4 по крайней мере половина узлов на любом простом пути от корня к листу, не считая сам корень, должны быть черными. Следовательно, черная высота корня должна составлять как минимум $h/2$; значит,

$$n \geq 2^{h/2} - 1 \, .$$

Перенося 1 в левую часть и логарифмируя, получим, что $\lg(n+1) \geq h/2$, или $h \leq 2\lg(n+1)$. ■

Непосредственным следствием леммы является то, что такие операции над динамическими множествами, как SEARCH, MINIMUM, MAXIMUM, PREDECESSOR и SUCCESSOR, при использовании красно-черных деревьев выполняются за время $O(\lg h)$, поскольку, как показано в главе 12, время работы этих операций на дереве поиска высотой h составляет $O(h)$, а любое красно-черное дерево с n узлами является деревом поиска высотой $O(\lg n)$. (Само собой разумеется, обращения к NIL в алгоритмах в главе 12 должны быть заменены обращениями к $T.nil$.) Хотя алгоритмы TREE-INSERT и TREE-DELETE из главы 12 и характеризуются временем работы $O(\lg n)$, если использовать их для вставки и удаления из красно-черного дерева, непосредственно использовать их для выполнения операций INSERT и DELETE нельзя, поскольку они не гарантируют сохранение красно-черных свойств после внесения изменений в дерево. Однако в разделах 13.3 и 13.4 вы увидите, что эти операции также могут быть выполнены за время $O(\lg n)$.

Упражнения

13.1.1

Начертите в стиле рис. 13.1, (а) полное бинарное дерево поиска высотой 3 с ключами $\{1, 2, \ldots, 15\}$. Добавьте к нему листья NIL и раскрасьте полученное дерево разными способами, так, чтобы в результате получились красно-черные деревья с черной высотой 2, 3 и 4.

13.1.2

Начертите красно-черное дерево, которое получится в результате вставки с помощью алгоритма TREE-INSERT ключа 36 в дерево, изображенное на рис. 13.1. Если вставленный узел закрасить красным, будет ли полученное в результате дерево красно-черным? А если закрасить этот узел черным?

13.1.3

Определим ***ослабленное красно-черное дерево*** как бинарное дерево поиска, которое удовлетворяет красно-черным свойствам 1, 3, 4 и 5. Другими словами, корень

может быть как черным, так и красным. Рассмотрите ослабленное красно-черное дерево T, корень которого красный. Если мы перекрасим корень дерева T из красного цвета в черный, будет ли полученное в результате дерево красно-черным?

13.1.4

Предположим, что каждый красный узел в красно-черном дереве "поглощается" его черным родительским узлом, так что дочерний узел красного узла становится дочерним узлом его черного родителя (мы не обращаем внимания на то, что при этом происходит с ключами в узлах). Чему равен возможный порядок черного узла после того, как будут поглощены все его красные потомки? Что вы можете сказать о глубине листьев полученного дерева?

13.1.5

Покажите, что самый длинный простой нисходящий путь от вершины x к листу красно-черного дерева имеет длину, не более чем в два раза превышающую кратчайший нисходящий путь от x к листу-потомку.

13.1.6

Чему равно наибольшее возможное количество внутренних узлов в красно-черном дереве с черной высотой k? А наименьшее возможное количество?

13.1.7

Опишите красно-черное дерево с n ключами с наибольшим возможным отношением количества красных внутренних узлов к количеству черных внутренних узлов. Чему равно это отношение? Какое дерево имеет наименьшее указанное отношение, и чему равна его величина?

13.2. Повороты

Операции над деревом поиска TREE-INSERT и TREE-DELETE, будучи применены к красно-черному дереву с n ключами, выполняются за время $O(\lg n)$. Поскольку они изменяют дерево, в результате их работы могут нарушаться красно-черные свойства, перечисленные в разделе 13.1. Для восстановления этих свойств необходимо изменить цвета некоторых узлов дерева, а также структуру его указателей.

Изменения в структуре указателей будут выполняться с помощью ***поворотов*** (rotations), которые представляют собой локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска. На рис. 13.2 показаны два типа поворотов — левый и правый. При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом $T.nil$; x может быть любым узлом дерева, правый дочерний узел которого — не $T.nil$. Левый поворот выполняется "вокруг" связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

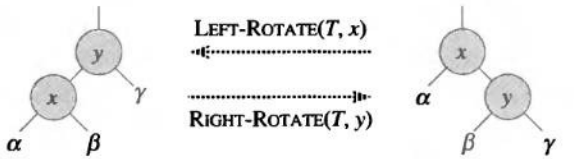


Рис. 13.2. Операция поворота в бинарном дереве поиска. Операция LEFT-ROTATE(T, x) преобразует конфигурацию из двух узлов справа в конфигурацию, показанную слева, путем изменения константного количества указателей. Обратная операция RIGHT-ROTATE(T, y) преобразует конфигурацию, показанную слева, в конфигурацию в правой части рисунка. Буквы α, β и γ представляют произвольные поддеревья. Операция поворота сохраняет свойство бинарного дерева поиска: ключи в α предшествуют ключу $x.key$, который предшествует ключам в β, которые предшествуют ключу $y.key$, который предшествует ключам в γ.

В псевдокоде процедуры LEFT-ROTATE предполагается, что $x.right \neq T.nil$ и что родитель корневого узла — $T.nil$.

```
LEFT-ROTATE(T, x)
 1  y = x.right                  // Установка y
 2  x.right = y.left             // Превращение левого поддерева y
                                 // в правое поддерево x
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                    // Передача родителя x узлу y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                   // Размещение x в качестве левого
                                 // дочернего узла y
12  x.p = y
```

На рис. 13.3 показан конкретный пример изменения бинарного дерева поиска процедурой LEFT-ROTATE. Код процедуры RIGHT-ROTATE симметричен коду LEFT-ROTATE. Обе эти процедуры выполняются за время $O(1)$. При повороте изменяются только указатели; все прочие атрибуты сохраняют свое значение.

Упражнения

13.2.1

Напишите псевдокод процедуры RIGHT-ROTATE.

13.2.2

Покажите, что в каждом бинарном дереве поиска с n узлами имеется ровно $n - 1$ возможных поворотов.

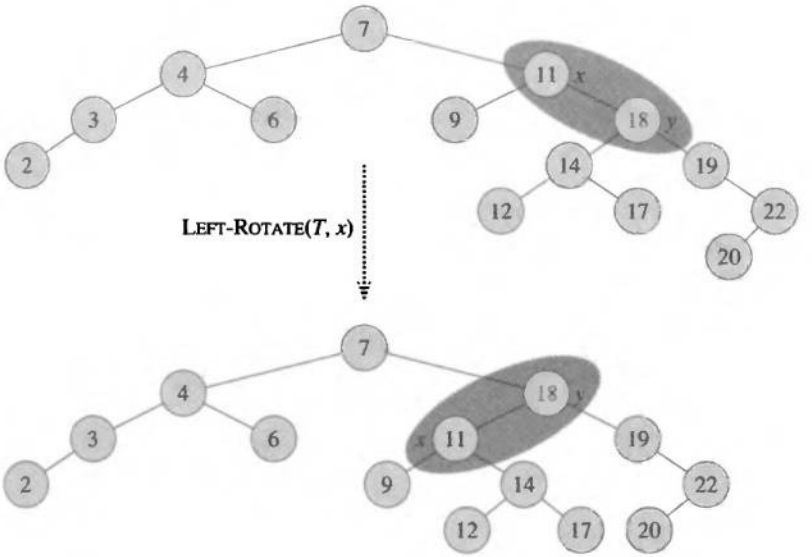


Рис. 13.3. Пример изменения бинарного дерева поиска процедурой LEFT-ROTATE(T, x). Центрированные обходы исходного и модифицированного деревьев дают одинаковые списки значений ключей.

13.2.3

Пусть a, b и c представляют собой произвольные узлы в поддеревьях α, β и γ соответственно в правом дереве на рис. 13.2. Как изменятся глубины узлов a, b и c при левом повороте в узле x?

13.2.4

Покажите, что произвольное бинарное дерево поиска с n узлами может быть преобразовано в любое другое бинарное дерево поиска с n узлами с использованием $O(n)$ поворотов. (*Указание:* сначала покажите, что $n - 1$ правых поворотов достаточно для преобразования дерева в правую цепочку.)

13.2.5 ⋆

Назовем бинарное дерево поиска T_1 ***правопреобразуемым*** в бинарное дерево поиска T_2, если можно получить T_2 из T_1 путем выполнения последовательности вызовов процедуры RIGHT-ROTATE. Приведите пример двух деревьев T_1 и T_2, таких, что T_1 не является правопреобразуемым в T_2. Затем покажите, что если дерево T_1 является правопреобразуемым в T_2, то это преобразование можно выполнить с помощью $O(n^2)$ вызовов процедуры RIGHT-ROTATE.

13.3. Вставка

Вставка узла в красно-черное дерево с n узлами может быть выполнена за время $O(\lg n)$. Для вставки узла z в дерево T мы используем немного модифицированную версию процедуры TREE-INSERT (см. раздел 12.3), которая вставляет узел в дерево, как если бы это было обычное бинарное дерево поиска, а затем окрашивает его в красный цвет. (В упр. 13.3.1 нужно ответить, почему выбран именно красный, а не черный цвет.) Для того чтобы вставка сохраняла красно-черные свойства дерева, после нее вызывается вспомогательная процедура RB-INSERT-FIXUP, которая перекрашивает узлы и выполняет повороты. Вызов RB-INSERT(T, z) вставляет в красно-черное дерево T узел z с уже заполненным атрибутом key.

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4       y = x
 5       if z.key < x.key
 6            x = x.left
 7       else x = x.right
 8  z.p = y
 9  if y == T.nil
10       T.root = z
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

Есть четыре отличия между процедурами TREE-INSERT и RB-INSERT. Во-первых, все NIL в TREE-INSERT заменены на $T.nil$. Во-вторых, для поддержки корректности структуры дерева в строках 14 и 15 процедуры RB-INSERT выполняется присвоение $T.nil$ атрибутам $z.left$ и $z.right$. В третьих, в строке 16 мы назначаем узлу z красный цвет. И наконец, в-четвертых, поскольку красный цвет z может вызвать нарушение одного из красно-черных свойств, в строке 17 вызывается вспомогательная процедура RB-INSERT-FIXUP(T, z), назначение которой — восстановить красно-черные свойства дерева.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // Случай 1
 6              y.color = BLACK                      // Случай 1
 7              z.p.p.color = RED                    // Случай 1
 8              z = z.p.p                            // Случай 1
 9          else if z == z.p.right
10                  z = z.p                          // Случай 2
11                  LEFT-ROTATE(T, z)                // Случай 2
12              z.p.color = BLACK                    // Случай 3
13              z.p.p.color = RED                    // Случай 3
14              RIGHT-ROTATE(T, z.p.p)               // Случай 3
15      else (то же, что и в части then, но с заменой
              "правого" (right) "левым" (left) и наоборот)
16  T.root.color = BLACK
```

Для того чтобы понять, как работает процедура RB-INSERT-FIXUP, разобьем изучение кода на три части. Сначала определим, какие из красно-черных свойств нарушаются при вставке узла z и его окраске в красный цвет. Затем рассмотрим назначение цикла **while** в строках 1–15. После этого изучим каждый из трех случаев[1], которые встречаются в этом цикле, и посмотрим, каким образом достигается цель в каждом из них. На рис. 13.4 показан пример работы процедуры RB-INSERT-FIXUP.

Какие из красно-черных свойств могут быть нарушены перед вызовом RB-INSERT-FIXUP? Свойство 1, определенно, выполняется (как и свойство 3), так как оба дочерних узла вставляемого узла являются ограничителями $T.nil$. Свойство 5, согласно которому для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов, также остается в силе, поскольку узел z замещает (черный) ограничитель, будучи при этом красным и имея черные дочерние узлы. Таким образом, может нарушаться только свойство 2, которое требует, чтобы корень красно-черного дерева был черным, и свойство 4, согласно которому красный узел не может иметь красного потомка. Оба нарушения возможны в силу того, что узел z после вставки окрашивается в красный цвет. Свойство 2 оказывается нарушенным, если узел z становится корнем, а свойство 4 — если родительский по отношению к z узел является красным. На рис. 13.4, (а) показано нарушение свойства 4 после вставки узла z.

Цикл **while** в строках 1–15 сохраняет в начале каждой итерации цикла следующий инвариант, состоящий из трех частей.

---

[1]Случаи 2 и 3 не являются взаимоисключающими.

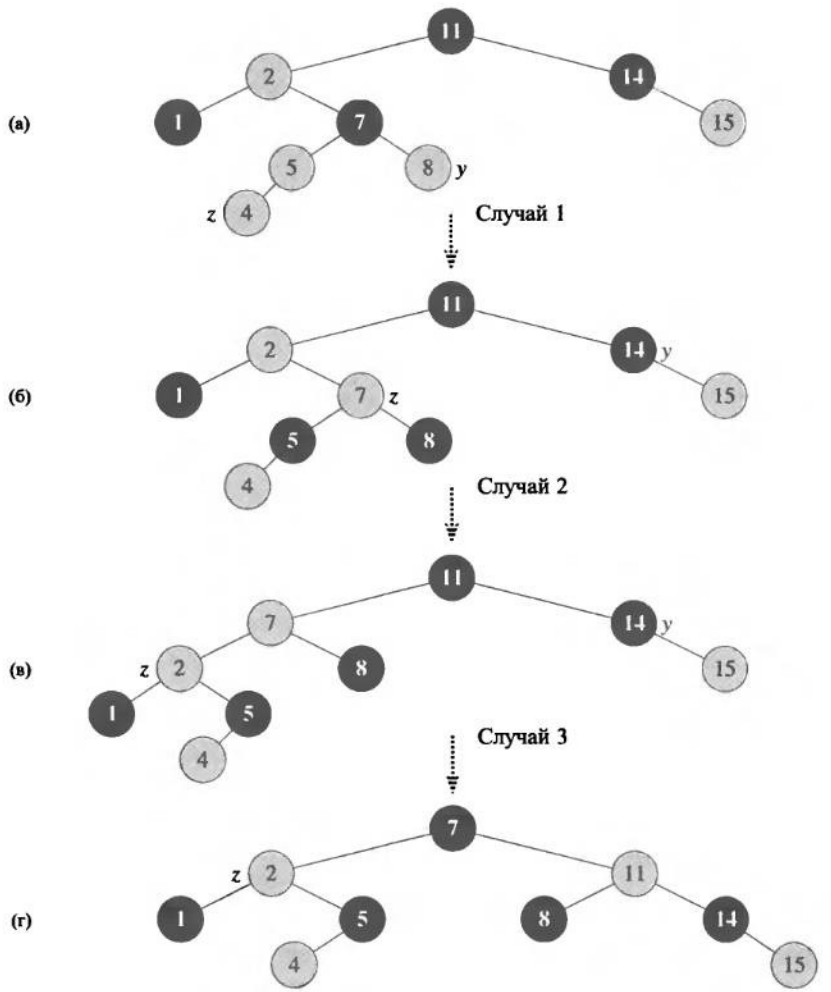


Рис. 13.4. Пример работы процедуры RB-INSERT-FIXUP. **(а)** Узел z после вставки. Поскольку и z. и его родитель $z.p$ красные, наблюдается нарушение свойства 4. Поскольку "дядя" z, узел y, красный, в коде процедуры срабатывает случай 1. Мы перекрашиваем узлы и перемещаем указатель z вверх по дереву, получая результат, показанный в части **(б)**. Вновь z и его родитель красные, но теперь "дядя" z, узел y, черный. Поскольку z является правым дочерним узлом по отношению к $z.p$. применим случай 2. Мы выполняем левый поворот, и полученное в результате дерево показано в части **(в)**. Теперь z является левым дочерним узлом своего родителя, так что применим случай 3. Перекрашивание и правый поворот дают показанное в части **(г)** корректное красно-черное дерево.

а. Узел z красный.

б. Если $z.p$ является корнем, то $z.p$ черный.

в. Если имеется нарушение красно-черных свойств, то это нарушение только одно — нарушение либо свойства 2, либо свойства 4. Если нарушено свойство 2, то это вызвано тем, что корнем дерева является красный узел z; если нарушено свойство 4, то в этом случае красными являются узлы z и $z.p$.

Часть (в), в которой говорится о возможных нарушениях красно-черных свойств, наиболее важна для того, чтобы показать, что процедура RB-INSERT-FIXUP восстанавливает красно-черные свойства. Части (а) и (б) просто поясняют ситуацию. Поскольку мы сосредоточиваем свое рассмотрение только на узле z и узлах, находящихся в дереве вблизи него, полезно знать, что узел z — красный (часть (а)). Часть (б) используется для того, чтобы показать, что узел $z.p.p$, к которому мы обращаемся в строках 2, 3, 7, 8, 13 и 14, существует.

Вспомним, что необходимо показать, что инвариант цикла выполняется перед первой итерацией цикла, что любая итерация цикла сохраняет инвариант и что инвариант цикла обеспечивает выполнение требуемого свойства по окончании работы цикла.

Начнем с рассмотрения инициализации и завершения работы цикла, а затем, подробнее рассмотрев работу цикла, докажем, что он сохраняет инвариант цикла. Попутно покажем, что есть только два возможных варианта действий в каждой итерации цикла — указатель z перемещается вверх по дереву или выполняются некоторые повороты и цикл завершается.

Инициализация. Перед выполнением первой итерации цикла имеется красно-черное дерево без каких-либо нарушений красно-черных свойств, к которому мы добавляем красный узел z. Покажем, что все части инварианта цикла выполняются к моменту вызова процедуры RB-INSERT-FIXUP.

а. В момент вызова процедуры RB-INSERT-FIXUP узел z — вставленный в дерево красный узел.

б. Если узел $z.p$ является корнем, то он черный и не изменяется до вызова процедуры RB-INSERT-FIXUP.

в. Мы уже убедились в том, что красно-черные свойства 1, 3 и 5 сохраняются к моменту вызова процедуры RB-INSERT-FIXUP.

Если нарушается свойство 2, то красный корень должен быть добавленным в дерево узлом z, который при этом является единственным внутренним узлом дерева. Поскольку и родитель, и оба потомка z являются ограничителями, свойство 4 не нарушается. Таким образом, нарушение свойства 2 — единственное нарушение красно-черных свойств во всем дереве.

Если же нарушено свойство 4, то, поскольку дочерние по отношению к z узлы являются черными ограничителями, а до вставки z никаких нарушений красно-черных свойств в дереве не было, нарушение заключается в том, что и z, и $z.p$ — красные. Кроме этого, других нарушений красно-черных свойств не имеется.

Завершение. Цикл завершает свою работу, когда $z.p$ становится черным (если z — корневой узел, то $z.p$ представляет собой черный ограничитель $T.nil$). Таким образом, свойство 4 при завершении цикла не нарушается. В соответствии с инвариантом цикла единственным нарушением красно-черных свойств может быть нарушение свойства 2. В строке 16 это свойство восстанавливается, так что по завершении работы процедуры RB-INSERT-FIXUP все красно-черные свойства дерева выполняются.

Сохранение. В действительности во время работы цикла **while** следует рассмотреть шесть разных случаев, однако три из них симметричны трем другим; разница лишь в том, является ли родитель $z.p$ левым или правым дочерним узлом по отношению к своему родителю $z.p.p$, что и выясняется в строке 2 (мы привели код только для ситуации, когда $z.p$ является левым потомком). Узел $z.p.p$ существует, поскольку, в соответствии с частью (б) инварианта цикла, если $z.p$ — корень дерева, то он черный. Поскольку цикл начинает работу, только если $z.p$ — красный, то $z.p$ не может быть корнем. Следовательно, $z.p.p$ существует.

Случай 1 отличается от случаев 2 и 3 цветом "брата" родительского по отношению к z узла, т.е. "дяди" узла z. После выполнения строки 3 указатель y указывает на дядю узла z — узел $z.p.p.right$, а в строке 4 проверяется его цвет. Если y — красный, выполняется код для случая 1; в противном случае выполняется код для случаев 2 и 3. В любом случае узел $z.p.p$ — черный, поскольку узел $z.p$ — красный, а свойство 4 нарушается только между z и $z.p$.

Случай 1. "Дядя" y узла z — красный

На рис. 13.5 показана ситуация, возникающая в случае 1 (строки 5–8), когда и $z.p$, и y — красные узлы. Поскольку узел $z.p.p$ — черный, мы можем исправить ситуацию, когда и z, и $z.p$ оба красные, покрасив и $z.p$, и y в черный цвет. Мы можем также окрасить $z.p.p$ в красный цвет, тем самым поддержав свойство 5. После этого мы повторяем цикл **while** с узлом $z.p.p$ в качестве нового узла z. Указатель z, таким образом, перемещается на два уровня вверх по дереву.

Теперь покажем, что в случае 1 инвариант цикла сохраняется. Обозначим через z узел z в текущей итерации, а через $z' = z.p.p$ — узел, который будет называться z в проверке в строке 1 в следующей итерации.

а. Поскольку в данной итерации цвет узла $z.p.p$ становится красным, в начале следующей итерации узел z' — красный.

б. Узел $z'.p$ в текущей итерации — $z.p.p.p$, и цвет данного узла в пределах данной итерации не изменяется. Если это корневой узел, то его цвет до начала данной итерации был черным и остается таковым в начале следующей итерации.

в. Мы уже доказали, что в случае 1 свойство 5 сохраняется; кроме того, понятно, что при выполнении итерации не возникает нарушения свойства 1 или 3.

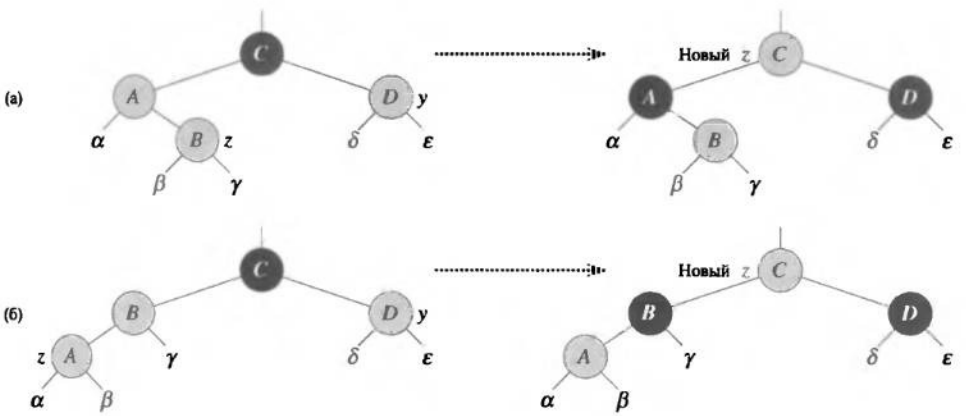


Рис. 13.5. Случай 1 процедуры RB-INSERT-FIXUP. Свойство 4 нарушено, поскольку и z, и его родительский узел $z.p$ красные. Мы предпринимаем одни и те же действия, когда **(а)** z является правым дочерним узлом и когда **(б)** z является левым дочерним узлом. Каждое из поддеревьев α, β, γ, δ и ε имеет черный корень, и у каждого одна и та же черная высота. Код для случая 1 изменяет цвета некоторых узлов, сохраняя свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Цикл **while** продолжается с "дедом" $z.p.p$ узла z в качестве нового узла z. Любое нарушение свойства 4 может теперь произойти только между новым z (окрашенным в красный цвет) и его родителем, если он также красный.

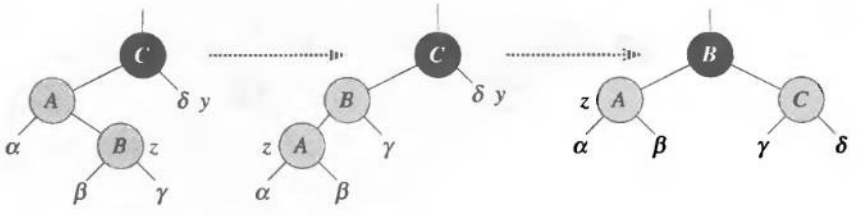
Если узел z' в начале очередной итерации является корнем, то код, соответствующий случаю 1, корректирует единственное нарушение свойства 4. Поскольку узел z' — красный и корневой, единственным нарушенным становится свойство 2, причем это нарушение связано с узлом z'.

Если узел z' в начале следующей итерации является корнем, то код, соответствующий случаю 1, корректирует единственное нарушение свойства 4, имеющееся перед выполнением итерации. Поскольку z' — узел красный и корневой, свойство 2 становится единственным нарушенным, и это нарушение вызвано узлом z'.

Если узел z' в начале следующей итерации корнем не является, то код, соответствующий случаю 1, не вызывает нарушения свойства 2. Этот код корректирует единственное нарушение свойства 4, имеющееся перед выполнением итерации. Коррекция выражается в том, что узел z' становится красным и оставляет узел $z'.p$ нетронутым. Если узел $z'.p$ был черным, то свойство 4 не нарушается; если же этот узел был красным, то окрашивание узла z' в красный цвет приводит к нарушению свойства 4 между узлами z' и $z'.p$.

Случай 2. "Дядя" y узла z черный, и z — правый потомок
Случай 3. "Дядя" y узла z черный, и z — левый потомок

В случаях 2 и 3 цвет узла y, являющегося "дядей" узла z, черный. Эти два случая отличаются один от другого тем, что z является левым или правым дочерним узлом по отношению к родительскому узлу $z.p$. Строки 10 и 11



Случай 2 Случай 3

Рис. 13.6. Случаи 2 и 3 процедуры RB-INSERT-FIXUP. Как и в случае 1, свойство 4 нарушается либо случаем 2, либо случаем 3, поскольку z и его родительский узел $z.p$ красные. Каждое из поддеревьев α, β, γ и δ имеет черный корень (α, β и γ согласно свойству 4, а δ — поскольку в противном случае мы получили бы случай 1), и каждое из них имеет одну и ту же черную высоту. Мы преобразуем случай 2 в случай 3 левым поворотом, который сохраняет свойство 5: все нисходящие простые пути от узла к листу содержат одно и то же количество черных узлов. Случай 3 приводит к определенному перекрашиванию и правому повороту, что также сохраняет свойство 5. Затем цикл **while** завершается, поскольку свойство 4 удовлетворено: в одной строке больше нет двух красных узлов.

псевдокода соответствуют случаю 2, который показан на рис. 13.6 вместе со случаем 3. В случае 2 узел z является правым потомком своего родительского узла. Мы используем левый поворот для преобразования сложившейся ситуации в случай 3 (строки 12–14), когда z является левым потомком. Поскольку и z, и $z.p$ — красные узлы, поворот не влияет ни на черную высоту узлов, ни на выполнение свойства 5. Когда мы приходим к случаю 3 (либо непосредственно, либо поворотом из случая 2), узел y, дядя узла z, имеет черный цвет (поскольку иначе мы бы получили случай 1). Кроме того, обязательно существует узел $z.p.p$, так как мы доказали, что этот узел существовал при выполнении строк 2 и 3, а также что после перемещения узла z на один узел вверх в строке 10 с последующим опусканием на один уровень в строке 11 узел $z.p.p$ остается неизменным. В случае 3 мы выполняем ряд изменений цвета и правый поворот, которые сохраняют свойство 5. После этого, так как у нас нет двух идущих подряд красных узлов, работа процедуры завершается. Больше тело цикла **while** не выполняется, так как узел $z.p$ теперь черный.

Покажем, что случаи 2 и 3 сохраняют инвариант цикла. (Как мы только что доказали, перед следующей проверкой в строке 1 узел $z.p$ будет черным и тело цикла больше выполняться не будет.)

а. В случае 2 выполняется присвоение, после которого z указывает на красный узел $z.p$. Никаких других изменений z или его цвета в случаях 2 и 3 не выполняется.

б. В случае 3 узел $z.p$ делается черным, так что если $z.p$ в начале следующей итерации является корнем, то этот корень — черный.

в. Как и в случае 1, в случаях 2 и 3 свойства 1, 3 и 5 сохраняются.

Поскольку узел z в случаях 2 и 3 не является корнем, нарушение свойства 2 невозможно. Случаи 2 и 3 не могут приводить к нарушению свойства 2.

поскольку при повороте в случае 3 сделанный красным узел становится дочерним по отношению к черному.

Таким образом, случаи 2 и 3 приводят к коррекции нарушения свойства 4, при этом не внося никаких новых нарушений красно-черных свойств.

Показав, что при любой итерации инвариант цикла сохраняется, мы тем самым показали, что процедура RB-INSERT-FIXUP корректно восстанавливает красно-черные свойства дерева.

Анализ

Чему равно время работы процедуры RB-INSERT? Поскольку высота красно-черного дерева с n узлами равна $O(\lg n)$, выполнение строк 1–16 процедуры RB-INSERT требует времени $O(\lg n)$. В процедуре RB-INSERT-FIXUP цикл **while** повторно выполняется только в случае 1, и указатель z при этом перемещается вверх по дереву на два уровня. Таким образом, общее количество возможных выполнений тела цикла **while** равно $O(\lg n)$. Следовательно, общее время работы процедуры RB-INSERT равно $O(\lg n)$. Интересно, что в ней никогда не выполняется больше двух поворотов, поскольку цикл **while** в случаях 2 и 3 завершает работу.

Упражнения

13.3.1

В строке 16 процедуры RB-INSERT мы окрашиваем вновь вставленный узел в красный цвет. Заметим, что если бы мы окрашивали его в черный цвет, то свойство 4 красно-черного дерева не было бы нарушено. Так почему же мы не делаем этого?

13.3.2

Изобразите красно-черные деревья, которые образуются при последовательной вставке ключей $41, 38, 31, 12, 19, 8$ в изначально пустое красно-черное дерево.

13.3.3

Предположим, что черная высота каждого из поддеревьев α, β, γ, δ и ε на рис. 13.5 и 13.6 равна k. Найдите черные высоты каждого узла на этих рисунках, чтобы убедиться, что свойство 5 сохраняется при указанных преобразованиях.

13.3.4

Профессор озабочен вопросом, не может ли в процедуре RB-INSERT-FIXUP произойти присвоение значения RED узлу $T.nil.color$, ведь в этом случае проверка в строке 1 не вызовет окончания работы цикла, если z — корень дерева. Покажите, что страхи профессора безосновательны, доказав, что процедура RB-INSERT-FIXUP никогда не окрашивает $T.nil.color$ в красный цвет.

13.3.5

Рассмотрим красно-черное дерево, образованное вставкой n узлов с помощью процедуры RB-INSERT. Докажите, что если $n > 1$, то в дереве имеется как минимум один красный узел.

13.3.6

Предложите эффективную реализацию процедуры RB-INSERT в случае, когда представление красно-черных деревьев не включает указатель на родительский узел.

13.4. Удаление

Как и остальные базовые операции над красно-черными деревьями с n узлами, удаление узла выполняется за время $O(\lg n)$. Удаление оказывается несколько более сложной задачей, чем вставка.

Процедура для удаления узла из красно-черного дерева основана на процедуре TREE-DELETE (раздел 12.3). Сначала нужно внести изменения в подпрограмму TRANSPLANT, которую процедура TREE-DELETE вызывает в процессе работы с красно-черным деревом.

RB-TRANSPLANT(T, u, v)
1 **if** $u.p == T.nil$
2 $\quad T.root = v$
3 **elseif** $u == u.p.left$
4 $\quad u.p.left = v$
5 **else** $u.p.right = v$
6 $v.p = u.p$

Процедура RB-TRANSPLANT имеет два отличия от процедуры TRANSPLANT. Во-первых, строка 1 обращается к ограничителю $T.nil$, а не к NIL. Во-вторых, присваивание атрибуту $v.p$ в строке 6 выполняется безусловно: возможно выполнение присваивания, даже если v указывает на ограничитель. В действительности мы будем использовать возможность присваивания атрибуту $v.p$ при $v = T.nil$.

Процедура RB-DELETE подобна процедуре TREE-DELETE, но имеет дополнительные строки псевдокода. Некоторые из них отслеживают узел y, который может вызвать нарушения красно-черных свойств. Если нужно удалить узел z и z имеет меньше двух дочерних узлов, то z удаляется из дерева, и мы делаем y совпадающим с z. Если у z два дочерних узла, то узел y должен быть преемником z в дереве, и y перемещается в дереве в позицию узла z. Мы также запоминаем цвет y перед его удалением или перемещением и отслеживаем узел x, который перемещается в исходную позицию узла y в дереве, поскольку узел x также может привести к нарушению красно-черных свойств. После удаления узла z процедура RB-DELETE вызывает вспомогательную процедуру RB-DELETE-FIXUP, которая

изменяет цвета и выполняет повороты для восстановления свойств красно-черного дерева.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Хотя процедура RB-DELETE содержит почти в два раза больше строк, чем псевдокод TREE-DELETE, обе эти процедуры имеют одинаковую базовую структуру. Каждую строку TREE-DELETE можно найти в RB-DELETE (с тем отличием, что NIL заменено $T.nil$, а вызов TRANSPLANT — вызовом RB-TRANSPLANT), и выполняются эти строки при одних и тех же условиях.

А вот отличия между этими двумя процедурами.

- Мы поддерживаем узел y в качестве узла, либо удаленного из дерева, либо перемещенного в пределах последнего. В строке 1 y становится указывающим на узел z, если z имеет меньше двух дочерних узлов и, таким образом, оказывается удаленным. Когда z имеет два дочерних узла, в строке 9 y становится указывающим на узел, следующий в дереве за z, так же, как в процедуре TREE-DELETE, и y перемещается в дереве в позицию узла z.
- Поскольку цвет узла y может измениться, переменная $y\text{-}original\text{-}color$ хранит цвет узла y до любых изменений цвета. В строках 2 и 10 выполняется установка этой переменной немедленно после присваивания значения переменной y. Когда z имеет два дочерних узла, $y \neq z$ и узел y перемещается в исходную позицию узла z в красно-черном дереве; строка 20 назначает y тот же цвет, что

и цвет узла z. Необходимо хранить исходный цвет y для его проверки в конце процедуры RB-DELETE; если он был черным, то удаление или перемещение y может привести к нарушениям свойств красно-черного дерева.

- Как уже говорилось, мы отслеживаем узел x, который перемещается в исходную позицию узла y. Присваивания в строках 4, 7 и 11 делают x указывающим либо на единственный дочерний узел узла y, либо, если y не имеет дочерних узлов, на ограничитель $T.nil$. (Вспомним из раздела 12.3, что у узла y нет левого дочернего узла.)
- Поскольку узел x перемещается в исходную позицию узла y, атрибут $x.p$ всегда устанавливается указывающим на исходную позицию родительского по отношению к y узла в дереве, даже если x в действительности является ограничителем $T.nil$. Присваивание атрибуту $x.p$ в строке 6 процедуры RB-TRANSPLANT имеет место во всех случаях, кроме ситуации, когда z исходно является родителем y (что осуществляется, только когда z имеет два дочерних узла и следующий за z элемент y представляет собой правый дочерний узел z). (Заметим, что когда RB-TRANSPLANT вызывается в строке 5, 8 или 14, третий передаваемый параметр совпадает с x.)

  Однако если исходным родительским узлом узла y является узел z, нам не нужно, чтобы атрибут $x.p$ указывал на исходный родитель y, поскольку мы удаляем этот узел из дерева. Поскольку узел y передвинется вверх и займет в дереве позицию узла z, установка $x.p$ равным y в строке 13 приведет к тому, что $x.p$ будет указывать на исходную позицию родителя y, даже если $x = T.nil$.
- Наконец, если узел y был черным, в свойства красно-черного дерева может быть внесено одно или несколько нарушений, так что для восстановления свойств красно-черного дерева в строке 22 выполняется вызов RB-DELETE-FIXUP. Если узел y был красным, при переименовании или удалении узла y красно-черные свойства сохраняются по следующим причинам.

  1. Ни одна черная высота в дереве не меняется.
  2. Никакие красные узлы не делаются смежными. Поскольку y занимает в дереве место z вместе с цветом узла z, мы не можем получить два смежных красных узла в новой позиции узла y в дереве. Кроме того, если y не был правым дочерним узлом z, исходный правый дочерний узел x узла y заменяет последний в дереве. Если y красный, то x должен быть черным, так что замена y на x не может привести к тому, что два красных узла станут смежными.
  3. Поскольку узел y не может быть корнем, если он был красным, корень остается черным.

Если узел y был черным, то могут возникнуть три проблемы, которые исправит вызов RB-DELETE-FIXUP. Во-первых, если y был корнем, а теперь новым корнем стал красный потомок y, нарушается свойство 2. Во-вторых, если и x, и $x.p$ красные, то нарушается свойство 4. И в-третьих, перемещение y в дереве

приводит к тому, что любой простой путь, ранее содержавший y, теперь имеет на один черный узел меньше. Таким образом, для всех предков y оказывается нарушенным свойство 5. Мы можем исправить ситуацию, утверждая, что узел x, ныне занимающий исходную позицию y, — "сверхчерный", т.е. при рассмотрении любого простого пути, проходящего через x, следует добавлять дополнительную единицу к количеству черных узлов. При такой интерпретации свойство 5 остается выполняющимся. При удалении или перемещении черного узла y мы передаем его "черноту" узлу x. Проблема заключается в том, что теперь узел x не является ни черным, ни красным, что нарушает свойство 1. Вместо этого узел x окрашен либо "дважды черным", либо "красно-черным" цветом, что дает при подсчете черных узлов на простых путях, содержащих x, вклад, равный соответственно 2 или 1. Атрибут $color$ узла x при этом остается равным либо RED (если узел красно-черный), либо BLACK (если узел дважды черный). Другими словами, цвет узла x не соответствует его атрибуту $color$.

Теперь рассмотрим процедуру RB-DELETE-FIXUP и то, как она восстанавливает красно-черные свойства дерева поиска.

RB-DELETE-FIXUP(T, x)

```
1  while x ≠ T.root и x.color == BLACK
2      if x == x.p.left
3          w = x.p.right
4          if w.color == RED
5              w.color = BLACK                          // Случай 1
6              x.p.color = RED                          // Случай 1
7              LEFT-ROTATE(T, x.p)                      // Случай 1
8              w = x.p.right                            // Случай 1
9          if w.left.color == BLACK и w.right.color == BLACK
10             w.color = RED                            // Случай 2
11             x = x.p                                  // Случай 2
12         else if w.right.color == BLACK
13                 w.left.color = BLACK n               // Случай 3
14                 nw.color = RED                       // Случай 3
15                 RIGHT-ROTATE(T, w)                   // Случай 3
16                 w = x.p.right                        // Случай 3
17             w.color = x.p.color                      // Случай 4
18             x.p.color = BLACK                        // Случай 4
19             w.right.color = BLACK                    // Случай 4
20             LEFT-ROTATE(T, x.p)                      // Случай 4
21             x = T.root                               // Случай 4
22     else (то же, что и в части then, но с заменой
              "правого" (right) "левым" (left) и наоборот)
23 x.color = BLACK
```

Процедура RB-DELETE-FIXUP восстанавливает свойства 1, 2 и 4. В упр. 13.4.1 и 13.4.2 требуется показать, что эта процедура восстанавливает свойства 2 и 4,

так что в оставшейся части раздела мы обратим свое внимание на свойство 1. Цель цикла **while** в строках 1–22 заключается в перенесении дополнительной "черноты" вверх по дереву до тех пор, пока не выполнится одно из следующих условий.

1. x указывает на красно-черный узел — в этом случае мы просто делаем узел x "единожды черным" в строке 23.
2. x указывает на корень — в этом случае мы просто убираем излишнюю черноту.
3. Выполнив некоторые повороты и перекраску, мы выходим из цикла.

Внутри цикла **while** x всегда указывает на дважды черный узел, не являющийся корнем. В строке 2 мы определяем, является ли x левым или правым дочерним узлом своего родителя $x.p$. (Приведен подробный код для ситуации, когда x — левый потомок. Для правого потомка код аналогичен, симметричен и скрыт за описанием в строке 22.) Поддерживается указатель w, который указывает на второй потомок родителя x. Поскольку узел x дважды черный, узел w не может быть $T.nil$; в противном случае количество черных узлов на простом пути от $x.p$ к (единожды черному) листу w было бы меньше, чем количество черных узлов на простом пути от $x.p$ к x.

Четыре разных возможных случая[2] показаны на рис. 13.7. Перед тем как приступить к детальному рассмотрению каждого случая, убедимся, что в каждом из случаев преобразования сохраняется свойство 5. Ключевая идея заключается в необходимости убедиться, что применяемые преобразования в каждом случае сохраняют количество черных узлов (включая дополнительную черноту в x) на пути от корня включительно до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$. Таким образом, если свойство 5 выполнялось до преобразования, оно выполняется и после него. Например, на рис. 13.7, (а), который иллюстрирует случай 1, количество черных узлов на пути от корня до поддеревьев α и β равно 3 как до, так и после преобразования (не забудьте о том, что узел x — дважды черный). Аналогично количество черных узлов на пути от корня до любого из поддеревьев γ, δ, ε и ζ равно 2 как до, так и после преобразования. На рис. 13.7, (б) подсчет должен включать значение c, равное значению атрибута *color* корня показанного поддерева, которое может быть либо RED, либо BLACK. Если определить $\mathrm{count}(\mathrm{RED}) = 0$ и $\mathrm{count}(\mathrm{BLACK}) = 1$, то на пути от корня до поддерева α количество черных узлов равно $2 + \mathrm{count}(c)$; эта величина одинакова до и после выполнения преобразований. В такой ситуации после преобразования новый узел x имеет атрибут *color*, равный c, но реально это либо красно-черный узел (если $c = \mathrm{RED}$), либо дважды черный (если $c = \mathrm{BLACK}$). Прочие случаи могут быть проверены аналогично (см. упр. 13.4.5).

---

[2]Как и в процедуре RB-INSERT-FIXUP, случаи в процедуре RB-DELETE-FIXUP не являются взаимоисключающими.

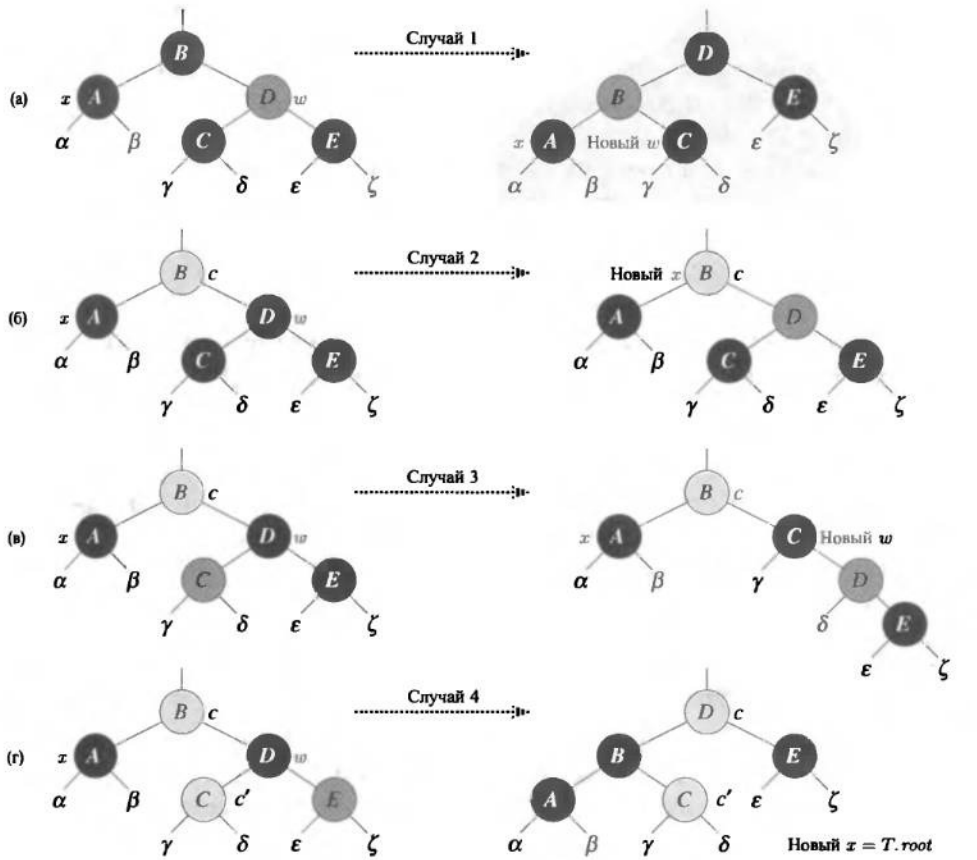


Рис. 13.7. Случаи цикла **while** процедуры RB-DELETE-FIXUP. У черных узлов атрибут *color* равен BLACK, у темно-серых — атрибут *color* равен RED, а у светлых узлов атрибут *color* представлен значениями c и c', которые могут быть либо RED, либо BLACK. Буквы $\alpha, \beta, \ldots, \zeta$ представляют произвольные поддеревья. Каждый случай преобразует конфигурацию, показанную слева, в конфигурацию, показанную справа, путем изменения некоторых цветов и/или поворота. Любой узел, на который указывает x, имеет дополнительный черный цвет и является либо дважды черным, либо красно-черным. Цикл повторяется только в случае 2. **(а)** Случай 1 преобразуется в случай 2, 3 или 4 путем изменения цвета узлов B и D и выполнения левого поворота. **(б)** В случае 2 дополнительная чернота, представленная указателем x, перемещается вверх по дереву путем окраски узла D в красный цвет и установки x указывающим на узел B. Если мы попадаем в случай 2 из случая 1, цикл **while** завершается, поскольку новый узел x — черно-красный, и, следовательно, значение c его атрибута *color* равно RED. **(в)** Случай 3 преобразуется в случай 4 путем обмена цветов узлов C и D и выполнения правого поворота. **(г)** Случай 4 убирает дополнительную черноту, представленную x, путем изменения некоторых цветов и выполнения левого поворота (без нарушения красно-черных свойств), после чего цикл завершается.

Случай 1. Брат w узла x — красный

Случай 1 (строки 5–8 процедуры RB-DELETE-FIXUP и рис. 13.7, (а)) возникает, когда узел w ("брат" узла x) — красный. Поскольку w должен иметь черные потомки, можно обменять цвета w и $x.p$, а затем выполнить левый поворот вокруг $x.p$ без нарушения каких-либо красно-черных свойств. Новый "брат" x, до поворота бывший одним из дочерних узлов w, теперь черный. Таким путем случай 1 приводится к случаю 2, 3 или 4.

Случаи 2, 3 и 4 возникают при черном узле w и отличаются один от другого цветами дочерних по отношению к w узлов.

Случай 2. Узел w — черный, оба его дочерних узла — черные

В этом случае (строки 10 и 11 процедуры RB-DELETE-FIXUP и рис. 13.7, (б)) оба дочерних узла w — черные. Поскольку узел w также черный, мы можем забрать черную окраску у x и w, сделав x единожды черным, а w — красным. Для того чтобы компенсировать удаление черной окраски x и w, мы можем добавить дополнительный черный цвет узлу $x.p$, который до этого мог быть как красным, так и черным. После этого будет выполнена следующая итерация цикла, в которой роль x будет играть текущий узел $x.p$. Заметим, что если мы переходим к случаю 2 от случая 1, новый узел x — красно-черный, поскольку исходный узел $x.p$ был красным. Следовательно, значение c атрибута $color$ нового узла x равно RED и цикл завершается при проверке условия цикла. После этого новый узел x окрашивается в обычный черный цвет в строке 23.

Случай 3. Брат w узла x — черный, левый дочерний узел узла w — красный, а правый — черный

В этом случае (строки 13–16 процедуры RB-DELETE-FIXUP и рис. 13.7, (в)) узел w — черный, его левый дочерний узел — красный, а правый — черный. Мы можем обменять цвета w и его левого дочернего узла $w.left$, а затем выполнить правый поворот вокруг w без нарушения каких-либо красно-черных свойств. Новым "братом" узла x после этого будет черный узел с красным правым дочерним узлом, и, таким образом, случай 3 приводится к случаю 4.

Случай 4. Брат w узла x черный, а правый дочерний узел узла w красный

В этом случае (строки 17–21 процедуры RB-DELETE-FIXUP и рис. 13.7, (г)) узел w — черный, а его правый дочерний узел — красный. Выполняя обмен цветов и левый поворот вокруг $x.p$, мы можем устранить излишнюю черноту в x, делая его просто черным, без нарушения каких-либо красно-черных свойств. Присвоение x указателя на корень дерева приводит к завершению работы цикла при проверке условия при следующей итерации.

Анализ

Чему равно время работы процедуры RB-DELETE? Поскольку высота дерева с n узлами равна $O(\lg n)$, общая стоимость процедуры без вызова вспомога-

тельной процедуры RB-DELETE-FIXUP равна $O(\lg n)$. В процедуре RB-DELETE-FIXUP в случаях 1, 3 и 4 завершение работы происходит после выполнения фиксированного числа изменений цвета и не более трех поворотов. Случай 2 — единственный, после которого возможно выполнение очередной итерации цикла **while**, причем указатель x перемещается вверх по дереву не более чем $O(\lg n)$ раз и никакие повороты при этом не выполняются. Таким образом, время работы процедуры RB-DELETE-FIXUP составляет $O(\lg n)$, причем она выполняет не более трех поворотов. Общее время работы процедуры RB-DELETE, само собой разумеется, также равно $O(\lg n)$.

Упражнения

13.4.1

Покажите, что после выполнения процедуры RB-DELETE-FIXUP корень дерева должен быть черным.

13.4.2

Покажите, что если в процедуре RB-DELETE и x, и $x.p$ красные, то при вызове RB-DELETE-FIXUP(T, x) свойство 4 будет восстановлено.

13.4.3

В упр. 13.3.2 вы построили красно-черное дерево, которое является результатом последовательной вставки ключей $41, 38, 31, 12, 19, 8$ в изначально пустое дерево. Покажите теперь красно-черные деревья, которые будут получаться в результате последовательного удаления ключей в порядке $8, 12, 19, 31, 38, 41$.

13.4.4

В каких строках кода процедуры RB-DELETE-FIXUP мы можем обращаться к ограничителю $T.nil$ или изменять его?

13.4.5

Для каждого из случаев, показанных на рис. 13.7, подсчитайте количество черных узлов на пути от корня показанного поддерева до каждого из поддеревьев $\alpha, \beta, \ldots, \zeta$ и убедитесь, что оно не меняется при выполнении преобразований. Если узел имеет атрибут $color$, равный c или c', воспользуйтесь символьными обозначениями $\mathrm{count}(c)$ или $\mathrm{count}(c')$.

13.4.6

Профессор озабочен вопросом, не может ли узел $x.p$ не быть черным в начале случая 1 в процедуре RB-DELETE-FIXUP. Если профессор прав, то строки 5 и 6 процедуры ошибочны. Покажите, что в начале случая 1 узел $x.p$ не может не быть черным, так что профессору нечего волноваться.

13.4.7

Предположим, что узел x вставлен в красно-черное дерево с помощью процедуры RB-INSERT, после чего немедленно удален из этого дерева процедурой RB-