4. Поиск

Рассмотрим множества объектов A и B. *Задача поиска* заключается в том, чтобы для заданной функции $f: A \to B$ определить все элементы $a \in A$ такие, что $f(a) = b$ для заданного элемента b из B. Элементы множества B будем называть *ключами*. Процедуру поиска элементов a будем называть *алгоритмом поиска*. В ряде случаев алгоритмы поиска требуют, чтобы на множестве элементов B было задано отношение линейного порядка $\preccurlyeq$.

Результатом поиска может быть один из двух исходов: либо поиск завершился *успешно*, и искомые объекты были найдены; либо поиск оказался *неудачным*, и искомые объекты не были обнаружены.

Методы поиска могут быть классифицированы несколькими способами. Различают алгоритмы *внешнего* поиска и *внутреннего*. Возможно деление на *статические* и *динамические* методы поиска, где термин статический означает, что содержимое множества A остается неизменным. Термин динамический означает, что множество A изменяется путем вставки (либо удаления) в него элементов. Третья возможная схема классификации методов поиска – их разделение в зависимости от того, на чем они основаны: на сравнении ключей или на некоторых их свойствах. Существуют и другие классификации, рассмотрение которых выходит за рамки данного пособия [9].

Ниже будут рассмотрены методы поиска, которые являются базовыми и могут быть применены к широкому классу математических объектов. Описываемые алгоритмы претендуют на некую универсальность, т. к. они не учитывают внутренней структуры объектов, к которым алгоритм применяется. Следует акцентировать внимание на том, что приведенные алгоритмы не имеют ничего общего с алгоритмами поиска на графах либо строковыми алгоритмами, идеи которых изначально базируются на особенностях объектов, их определяющих.

4.1. Элементарные методы поиска

Будем говорить, что метод поиска является *элементарным*, если для успешного завершения алгоритма, лежащего в его основе, достаточно, чтобы элементы множества A хранились в элементарной структуре данных. В этом разделе будем считать, что элементы из множества A записаны в массив и занумерованы натуральными числами от 1 до $|A|$, где $|A|$ – число элементов в A. В дальнейшем будем предполагать, что $|A| = n, n \in \mathbb{N}$. Если же алгоритм поиска работает не с дискретными объектами, а, например, с непрерывной функцией на заданном отрезке, то это будет специально оговорено.

Ниже будут рассмотрены базовые алгоритмы поиска – линейный, двоичный, тернарный с его модификацией, базирующейся на использовании «золотого сечения».

4.1.1. Линейный поиск

Линейный поиск является простейшим методом поиска, идея которого заключается в последовательном просмотре элементов множества A. Условно алгоритм линейного поиска может быть описан при помощи процедуры 4.1.1:

```
Процедура 4.1.1. Linear Search
01:  Linear Search Impl(A[1, n], b)
02:        for index = 1 to n do
03:              if f(A[index]) = b then
04:                    return index
```

Проанализируем время работы приведенного алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1) = 1, \\ T(n) = T(n-1) + O(1),\ n > 1. \end{cases} \tag{4.1}$$

Решением данного соотношения является функция $T(n) = O(n)$, поэтому сложность алгоритма линейного поиска есть $O(n)$, где n – число элементов в A. Основным недостатком метода является просмотр всех элементов множества A, число которых может быть достаточно велико. Следовательно, данный алгоритм не используется в ситуациях, когда процедуру поиска требуется повторять неоднократно. Линейный поиск применим как к упорядоченным множествам, так и к множествам, на которых не задано отношение порядка, а только ключ поиска. Делаем вывод, что метод линейного поиска является наиболее универсальной процедурой по сравнению с методами, рассмотреными ниже.

4.1.2. Двоичный поиск

Двоичный поиск (или *бинарный*, *дихотомия*) – метод поиска, базирующийся на том, что на множестве B задано отношение линейного порядка $\preccurlyeq$, в соответствие с которым элементы множества могут быть упорядочены.

Идея алгоритма двоичного поиска заключается в следующем. Пусть элемент с искомым ключом находится в позиции i. Тогда:

1) для всех j таких, что $i < j$ выполняется $f(A_i) \preccurlyeq f(A_j)$;

2) для всех j таких, что $j < i$ выполняется $f(A_j) \preccurlyeq f(A_i)$.

Следовательно, выбирая заведомо хорошую позицию i, которая будет проверяться на совпадение с ключом, двоичный поиск позволяет отбросить от дальнейшего анализа значительную часть элементов множества A. Хорошей позицией в случае двоичного поиска является средний элемент.

Предположив, что на множестве ключей B задано линейное отношение порядка «меньше либо равно», алгоритм двоичного поиска можно описать следующим образом (процедура 4.1.2).

Зачастую данные, которые поступают на вход двоичному поиску, не являются упорядоченными. В этом случае непосредственное применение двоичного поиска может привести к некорректному результату. Поэтому для достижения гарантированного успеха первое, что необходимо сделать, это проверить, являются ли элементы множества A отсортированными по ключу. Такую проверку можно осуществить, используя один проход по массиву, последовательно сравнивая соседние элементы (процедура 4.1.3).

Процедура 4.1.2. Binary Search

```
01:  Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) = b then
06:                    return middle
07:              if f(A[middle]) < b then lo = middle + 1
08:              if f(A[middle]) > b then hi = middle - 1
09:        return Linear Search(A[lo, hi], b)
```

Процедура 4.1.3. Check Binary Search Prerequisites

```
01:  for index = 1 to n - 1 do
02:        if f(A[index]) > f(A[index - 1]) then
03:              return false
04:  return true
```

Проанализируем время работы двоичного поиска. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1) = 1, \\ T(n) = T(n/2) + O(1),\ n > 1. \end{cases} \tag{4.2}$$

Решением данного соотношения является функция $T(n) = O(\log n)$. Поэтому сложность алгоритма двоичного поиска есть $O(\log n)$, где n – число элементов в A. Основным достоинством метода является его время выполнения в худшем случае, которое составляет $O(\log n)$ операций. Исходя из асимптотики времени выполнения алгоритма двоичного поиска, можно сделать вывод о том, что он очень хорошо подходит для задач, в которых процедуру поиска требуется повторять многократно для одного и того же множества ключей. Существуют ситуации, в которых линейный поиск будет приносить ощутимый выигрыш во времени выполнения. Одной из них является такая, при которой процедуру поиска требуется осуществлять не более чем $\lfloor \log_2 n \rfloor$ раз, при условии, что массив данных не является упорядоченным и ключи не могут быть отсортированы за линейное время. Для применения алгоритма двоичного поиска необходимо применение алгоритма сортировки, трудоемкость которого, по крайней мере, составит $O(n \log n)$.

Левосторонний / правосторонний двоичный поиск

В ряде случаев требуется определить не только вхождение элемента с заданным ключом в искомый массив данных, но и левую и правую границы, между которыми находятся все элементы из A, удовлетворяющие заданному критерию. В этом случае принято говорить о *левостороннем* и *правостороннем* двоичном поиске. Целью левостороннего является нахождение первого вхождения элемента с заданным ключом. Соответственно, целью правостороннего – нахождение последнего вхождения искомого ключа в заданный массив. Если элемент с заданным ключом не обнаружен, то левосторонний / правосторонний

поиск должен вернуть такой индекс i, для которого справедливо следующее соотношение:

$$f(A[i-1]) < b < f(A[i]). \quad (4.3)$$

Данное соотношение задает такую позицию i, в которую элемент с ключом b может быть вставлен без нарушения заданного отношения порядка.

С точки зрения практики наиболее полезными являются следующие соотношения:

для левостороннего поиска:

$$f(A[i-1]) < b \leq f(A[i]);$$

для правостороннего поиска:

$$f(A[i-1]) \leq b < f(A[i]).$$

Следует сказать, что возвращаемый индекс может варьироваться в зависимости от требований задачи. Несмотря на выбор критериев, оба метода выполняются за $O(\log n)$ шагов в худшем случае. Процедуры левостороннего и правостороннего поиска приведены ниже (процедура 4.1.4):

Процедура 4.1.4. Leftmost/Rightmost Binary Search

```
01:  Leftmost Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) < b then
06:                    lo = middle + 1
07:              else
08:                    hi = middle
09:        return Linear Search(A[lo, hi], b)
10:
11:  Rightmost Binary Search Impl(A[1, n], b)
12:        lo = 1, hi = n
13:        while (hi – lo > 2) do
14:              middle = lo + ⌊(hi - lo)/2⌋
15:              if f(A[middle]) > b then
16:                    hi = middle
17:              else
18:                    lo = middle + 1
19:        return Linear Search(A[lo, hi], b)
```

Вещественный двоичный поиск

Вещественный двоичный поиск используется для решения уравнений вида $f(x) = y$ на отрезке $[l, u]$ с заданной точность ε, где $f(x)$ – монотонная функция, множество A –область определения функции f, множество B – область значений функции f.

Будем предполагать, что $f(x)$ – неубывающая на заданном отрезке функция. Применив идеи для дискретного двоичного поиска, рассмотренные выше,

процедуру вещественного двоичного поиска можно описать следующим образом (процедура 4.1.5):

Процедура 4.1.5. Real Binary Search

```
01:  Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

Время работы приведенной процедуры есть $O\left(\log\frac{u-l}{\varepsilon}\right)$ в худшем случае. Отметим, что применение вещественного двоичного поиска в том виде, в котором он приведен в процедуре (4.1.5) может привести к зацикливанию по причине плохого выбора точности ε. Во избежание этого используется итеративная версия процедуры (4.1.5), приведенная ниже (процедура 4.1.6):

Процедура 4.1.6. Iterative Real Binary Search

```
01:  Iterative Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        for iteration = 1 to number of iterations do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

Выбор числа итераций зависит от специфики задачи и от точности, которой должно удовлетворять найденное значение. На практике же обычно достаточно 40–50 прогонов.

Среди задач, которые можно эффективно решать при помощи вещественного двоичного поиска следует отметить задачу нахождения вещественного корня n-ой степени из числа x. В этом случае нижней границей поиска будет число 1, а верхней – число x.

4.1.3. Тернарный поиск

Тернарный поиск (или *троичный*) – метод, который используется для поиска максимума / минимума выпуклой / вогнутой на заданном отрезке $[l,u]$ функции $f(x)$.

Пусть требуется найти максимум функции $f(x)$. Алгоритм базируется на том факте, что должно существовать некоторое значение x, чтобы:

1) для всех x_1, x_2, таких, что $l \le x_1 < x_2 \le x$, выполнялось $f(x_1) < f(x_2)$;
2) для всех x_1, x_2, таких, что $x \le x_1 < x_2 \le u$, выполнялось $f(x_1) > f(x_2)$.

С учетом вышеприведенного, алгоритм тернарного поиска можно описать следующим образом (процедура 4.1.7):

Процедура 4.1.7. Real Ternary Search

```
01:  Real Ternary Search Impl([l, u])
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(loThird) + ε < f(hiThird) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return (lo + hi) / 2
```

Если же алгоритм тернарного поиска применяется к дискретным множествам, то соответствующая подпрограмма будет претерпевать незначительные изменения (процедура 4.1.8)

Процедура 4.1.8. Ternary Search

```
01:  Ternary Search Impl(A[1, n])
02:        lo = 1, hi = n
03:        while (hi – lo > 3) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(A[loThird]) < f(A[hiThird]) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return Linear Search(A[lo, hi])
```

Проанализируем время работы тернарного поиска. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(n) = O(1), & n \le 3, \\ T(n) = T(2/3 \cdot n) + O(1), & n > 3. \end{cases} \tag{4.4}$$

Решением данного соотношения является функция $T(n) = O(\log_{1.5} n)$. Если алгоритм тернарного поиска применяется к функциям, работающим с непрерывными объектами, то сложность составит $O\left(\log_{1.5} \frac{u-l}{\varepsilon}\right)$.

Отметим, что недостатком тернарного поиска является сравнение значений функции в двух точках, определяющих границы каждой из третьих. Однако существуют техники, которые позволяют получить значительный прирост в скорости, путем «обсчета» каждой из рассматриваемых точек только один раз. Одна из таких техник будет рассмотрена ниже.

Поиск с помощью «золотого сечения»

Скрытую константу в оценке времени работы тернарного поиска можно значительно уменьшить, если вместо деления отрезка на три равные части, делить его в отношении «золотого сечения».

Пусть требуется найти, как и в предыдущем разделе, максимум функции $f(x)$, выпуклой на заданном отрезке $[l,u]$. Будем делить отрезок $[l,u]$ точками x_1 и x_2 $(x_1 < x_2)$ так, чтобы выполнялись следующие соотношения

$$\begin{cases} x_1 = l + \varphi \cdot (u - l), & x_1 = x_2 - \varphi \cdot (x_2 - l), \\ x_2 = u - \varphi \cdot (u - l), & x_2 = x_1 + \varphi \cdot (u - x_1). \end{cases} \quad (4.5)$$

Учитывая, что неизвестный параметр $\varphi \in (0,1)$, получим единственное решение системы уравнений (4.5) в виде $\varphi = 2 - \frac{1+\sqrt{5}}{2}$ * Соответствующая подпрограмма приведена ниже (процедура 4.1.9):

Процедура 4.1.9. Gold Section Ternary Search

```
01:  Gold Section Ternary Search Impl([l, u])
02:        φ = 2 − (1+√5)/2
03:        lo = l, hi = u
04:        if (lo + ε < hi) do
05:              loThird = lo + φ·(hi – lo)
06:              hiThird = hi - φ·(hi – lo)
07:              fLoThird = f(loThird)
08:              fHiThird = f(hiThird)
09:              do
10:                    if fLoTthird + ε < fHiThird then
11:                          lo = loThird
12:                          loThird = hiThird
13:                          hiThird = hi - φ·(hi – lo)
14:                          fLoThird = fHiThird
15:                          fHiThird = f(hiThird)
16:                    else
17:                          hi = hiThird
18:                          hiThird = loThird
19:                          loThird = lo + φ·(hi – lo)
20:                          fHiThird = fLoThird
21:                          fLoThird = f(loThird)
22:              while (lo + ε < hi)
23:        return (lo + hi) / 2
```

4.1.4. Интерполяционный поиск

Интерполяционный поиск, как и двоичный, используется для поиска данных в упорядоченных массивах. Их отличие состоит в том, что вместо деления области поиска на две примерно равные части, интерполяционный поиск производит оценку новой области поиска по расстоянию между ключом и текущими значениями граничных элементов (процедура 4.1.10):

* Отсюда название метода «золотого сечения».

Процедура 4.1.10. Interpolation Search

```
01:  Interpolation Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > threshold) do
04:              x = lo + (b – A[lo]) * (hi – lo) / (A[hi] – A[lo])
05:              if A[x] = b then
06:                    return x
07:              if A[x] < b then lo = x + 1
08:              if A[x] > b then hi = x - 1
09:        return Binary Search(A[lo, hi], b)
```

Использование интерполяционного поиска оправдано в том случае, когда данные в массиве, в котором происходит поиск, распределены достаточно равномерно. Тогда трудоемкость приведенной процедуры может составить $O(\log\log n)$, что асимптотически лучше трудоемкости двоичного поиска $O(\log n)$. Если данные распределены не достаточно равномерно, а, например, экспоненциально, то трудоемкость интерполяционного поиска может составить $O(n)$ в худшем случае. На практике интерполяционный поиск обычно используется для поиска значений в больших файлах данных, причем поиск продолжается до тех пор, пока не будет достигнут некий пороговый диапазон, в котором находится искомое значение. После этого обычно применяется двоичный либо линейный поиск.

В качестве примера использования интерполяционного поиска рассмотрим следующую задачу. Пусть есть словарь, состоящий из всех слов русского языка, упорядоченных по алфавиту. Задача состоит в том, чтобы достаточно быстро найти некоторое слово $\alpha_1\alpha_2 \ldots \alpha_n$. Очевиден тот факт, что человек не будет искать требуемое слово, используя идеи двоичного поиска, а начнет со страницы, которая содержит слова с префиксом искомого слова. В зависимости от места остановки поиска дальнейшие действия могут пропустить либо небольшое количество страниц, либо количество страниц, существенно большее половины допустимого интервала. Таким образом, основное отличие интерполяционного поиска от рассмотренных выше методов состоит в том, что в случае равномерно распределенных данных интерполяционный поиск учитывает разницу между «немного больше» и «существенно больше», тем самым позволяя сократить время поиска искомых данных до минимума.

4.2. Деревья поиска

Рассмотрим корневое бинарное дерево B, с каждым узлом которого ассоциирован некоторый объект, называющийся *ключом узла*. Предположим, что на множестве ключей задано отношение линейного порядка и все ключи являются различными. Дерево B является *бинарным деревом поиска*, если на нем определены операции интерфейса АТД «Бинарное дерево» и для каждого его узла выполнено следующее условие: ключ узла больше, чем любой из ключей его левого поддерева, и меньше, чем любой из ключей его правого поддерева.[*] Усло-

[*] В англоязычно литературе используется термин *BST – binary search tree.*

вие, накладываемое на порядок ключей в дереве, в дальнейшем будем называть *основным свойством* бинарного дерева поиска.

Для хранения BST в памяти ЭВМ воспользуемся подходом, описанным в разд. 2.2.5, процедура (2.2.20). В дальнейшем значение узла будем отождествлять с ключом, которому соответствует данное значение. Тогда процедура создания дерева может выглядеть следующим образом (процедура 4.2.1):

Процедура 4.2.1. BST: Make BST

```
01:  Make BST Impl(key, leftTree, rightTree)
02:        return binarySearchTree {
03:              root = binarySearchTreeNode {
04:                    key = key
05:                    left  = leftTree
06:                    right = rightTree
07:              }
```

Отметим, что требование к уникальности ключей не является строгим, поскольку узлы, ключи которых совпадают с ключом корня дерева, могут быть размещены либо слева, либо справа.

В соответствие с основным свойством BST процедура поиска может выглядеть следующим образом (процедура 4.2.2):

Процедура 4.2.2. BST: Contains

```
01:  Contains Impl(B, key)
02:        if not Is Empty(B) then
03:              R = Root(B)
04:              with R do
05:                    if key < Key(R) then return Contains(left , key)
06:                    if key > Key(R) then return Contains(right, key)
07:              return true
08:        return false
```

Операция вставки в дерево также не представляет сложностей. В этом случае достаточно спуститься до некоторого листа дерева и создать у него левого либо правого потомка, в зависимости от значения ключа, который вставляется в дерево (процедура 4.2.3):

Процедура 4.2.3. BST: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              if Is Empty(B) then
08:                    return Make BST(key, ∅, ∅)
09:              R = Root(B)
10:              with R do
11:                    if key < Key(R) then
12:                          return Make BST(key, IRec(left), right)
13:                    if key > Key(R) then
14:                          return Make BST(key, left, IRec(right))
```

Трудоемкость приведенного алгоритма есть $O(h)$, где h – высота дерева, так как в худшем случае вставляемое значение будет «прицеплено» к самому глубокому листу.

Перед рассмотрением процедуры удаления узла дерева с заданным ключом, введем ряд определений. *Преемником* узла n в заданном бинарном дереве поиска назовем либо пустой узел, если ключ узла n является наибольшим, либо узел, ключ которого непосредственно следует после ключа узла n в отсортированной последовательности ключей. *Предшественником* узла n в заданном бинарном дереве поиска назовем либо пустой узел, если ключ узла n является наименьшим, либо узел, ключ которого непосредственно следует перед ключом узла n в отсортированной последовательности ключей.[*] Для того чтобы найти преемника либо предшественника заданного узла, можно воспользоваться процедурой симметричного обхода, которая применительно к бинарным деревьям поиска будет иметь вид (процедура 4.2.4):

Процедура 4.2.4. BST: InOrder

```
01:  InOrder Impl(B)
02:        if not Is Empty(B) then
03:              R = Root(B)
04:              with R do
05:                    InOrder(left)
06:                    process(R)
07:                    InOrder(right)
```

Несложно видеть, что узлы дерева B в процессе симметричного обхода будут обработаны в порядке возрастания ключей. Непосредственное применение процедуры (4.2.4) к нахождению преемника и предшественника может привести к временным затратам $O(n)$, где n – число узлов дерева. Для того чтобы добиться трудоемкости $O(h)$, преемника и предшественника нужно находить непосредственно. Детали, касающиеся процедуры нахождения преемника приведены ниже (процедура 4.2.5). Процедура нахождения предшественника не приводится по причине сходства с (4.2.5).

Процедура 4.2.5. BST: Successor

```
01:  Successor Impl(R)
02:        if not Is Empty(Right(R)) then
03:              successor = Root(Right(R))
04:              while not Is Empty(Left(successor)) do
05:                    successor = Root(Left(successor))
06:              return successor
07:        else
08:              successor = Parent(R)
09:              while successor ≠ ∅ do
10:                    if Root(Left(successor)) = R then
11:                          break
12:                    R = successor, successor = Parent(R)
13:              return successor
```

[*] В англоязычной литературе соответственно используются термины *successor* и *predecessor*.

Для того чтобы удалить узел из бинарного дерева поиска, соответствующий заданному ключу, требуется найти необходимый узел и заменить его предшественником (в этом случае говорят о *левом удаление*) либо преемником (в этом случае говорят о *правом удалении*). Если у удаляемого узла только один потомок или их нет вовсе, то узел заменяется соответствующим потомком. Процедура правого удаления приведена ниже (процедура 4.2.6):

Процедура 4.2.6. BST: Delete

```
01:  Delete Impl(B, key)
02:        if Contains(B, key) then
03:              return DRec(B)
04:        return B
05:
06:        DRec Impl(B)
07:              R = Root(B)
08:              if key = Key(R) then
09:                    if Is Leaf(R) then
10:                          return ∅
11:                    with R do
12:                        if Is Empty(left)  then return right
13:                        if Is Empty(right) then return left
14:                        key = Key(Successor(R))
15:                        return Make BST(key, left, Delete(right, key))
16:              with R do
17:                  if key < Key(R) then
18:                        return Make BST(Key(R), DRec(left), right)
19:                  if key > Key(R) then
20:                        return Make BST(Key(R), left, DRec(right))
```

Несложно видеть, что процедура удаления узла дерева будет иметь трудоемкость $O(h)$, поскольку трудоемкость всех этапов (поиска узла, нахождения и удаления преемника) не превосходит $O(h)$.

Следует отметить, что для бинарных деревьев поиска, как и для списков, определяются операции *соединения* и *разделения*. Операция соединения соединяет два дерева в одно, предполагая, что ключи одного дерева строго меньше, чем ключи другого. Операция разделения разделяет заданное дерево на два поддерева по заданному ключу таким образом, чтобы ключи одного были строго меньше, чем ключи другого. Обе операции имеют трудоемкость $O(h)$.

Так же, как и для линейных списков, операции вставки и удаления в бинарное дерево поиска могут быть реализованы путем применения конечного числа операций разделения и соединения. Так как каждая из них имеет трудоемкость $O(h)$, то и трудоемкость операций, реализованных посредством соединения и разделения составит $O(h)$. Отметим, что на практике непосредственная реализация операций вставки и удаления является наиболее предпочтительной, так как скрытая константа в асимптотике O в этом случае будет значительно меньше.

Если применять бинарные деревья поиска в том виде, в каком они описаны выше, то трудоемкость всех операций может составить $O(n)$ в случае, если дерево выродиться в цепь (например, при последовательной вставке возрастающей последовательности ключей). Для того чтобы дерево оставалось сбалансированным, т. е. его высота удовлетворяла бы оценке $O(\log n)$, используются специальные виды бинарных деревьев поиска, каждое из которых придерживается фиксированного набора инвариантов, сохранение которых позволяет добиться трудоемкости $O(\log n)$ для всех операций.

Под операциями *балансировки* будем понимать такие операции над бинарным деревом поиска, которые позволяют сохранить инварианты, определенные на дереве, с целью поддержания его в сбалансированном состоянии.

К основным операциям балансировки относятся операции *вращения* (или *поворота*), среди которых выделяют *малые вращения* (рис. 4.1) и *большие вращения* (или *двойные вращения*) (рис. 4.2, 4.3). Идея их заключается в том, чтобы изменить некоторые связи «предок – потомок» в дереве, при этом сохранив основное свойство бинарных деревьев поиска. Как среди малых, так и среди больших вращений выделяют *правые* и *левые*. Процедуры малых вращений приведены ниже (4.2.7).

Процедура 4.2.7. BST: Rotate Left, Rotate Right

```
01:  Rotate Left Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(right) then
05:                    return B
06:              X = Root(right)
07:              leftTree  = Make BST(Key(R), left, Left(X))
08:              return Make BST(Key(X), leftTree, Right(X))
09:
10:  Rotate Right Impl(B)
11:        R = Root(B)
12:        with R do
13:              if Is Empty(left) then
14:                    return B
15:              X = Root(left)
16:              rightTree = Make BST(Key(R), Right(X), right)
17:              return Make BST(Key(X), Left(X), rightTree)
```

Несложно видеть, что обе операции имеют трудоемкость $O(1)$, так как они затрагивают фиксированное число узлов дерева.

Большие вращения базируются на малых вращения. Большое левое вращение вначале совершает правый поворот, а затем левый (процедура 4.2.8).

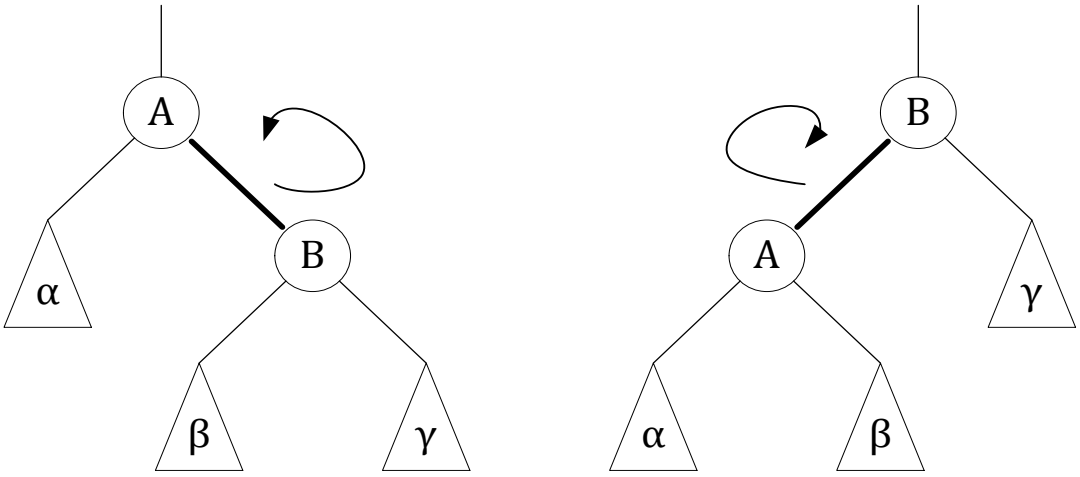

Рис. 4.1. Малые левое и правое вращения

Процедура 4.2.8. BST: Double Rotate Left

```
01:  Double Rotate Left Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(right) then
05:                    return B
06:              rotated = Make BST(Key(R), left, Rotate Right(right))
07:              return Rotate Left(rotated)
```

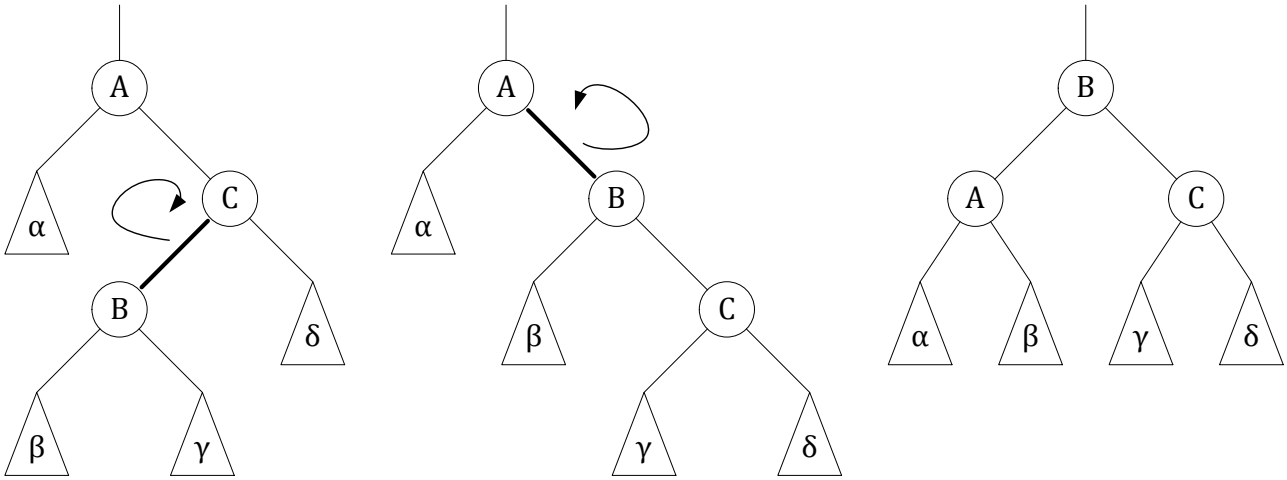

Рис. 4.2. Большое левое вращения

Большое правое вращение вначале совершает левый поворот, а затем правый (процедура 4.2.9).

Процедура 4.2.9. BST: Double Rotate Right

```
01:  Double Rotate Right Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(left) then
05:                    return B
06:              rotated = Make BST(Key(R), Rotate Left(left), right)
07:              return Rotate Right(rotated)
```

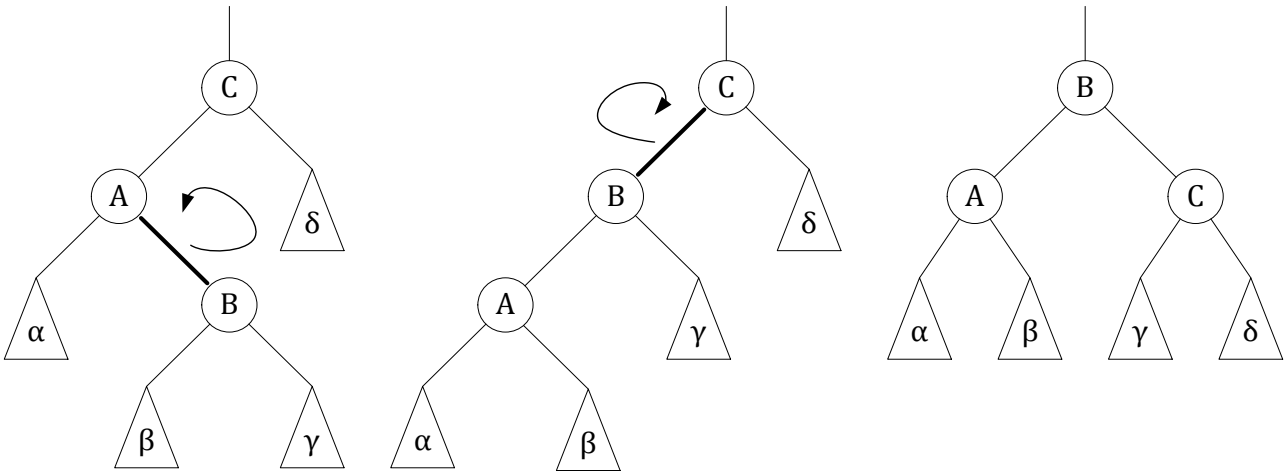

Рис. 4.3. Большое правое вращения

Так как малые вращения имеют трудоемкость $O(1)$, то и трудоемкость больших вращений составит $O(1)$.

В зависимости от свойств, которыми обладает бинарное дерево поиска, операции вращения могут претерпевать те либо иные изменения (например, вращения на AVL-дереве, на красно-черном дереве и т. д.). Общей идеей, объединяющей все типы вращений, является подъем (или вытягивание) поддеревьев с низких уровней на верхние, что позволяет достичь сравнительной сбалансированности всего дерева в целом.

Помимо операций вращения существует ряд других операций и техник, которые позволяют поддерживать дерево в сбалансированном состоянии. Отметим лишь, что большинство из них базируется на специфических свойствах бинарных деревьев, на которых эти операции определены.

4.2.1. AVL-дерево

AVL-дерево – сбалансированное бинарное дерево поиска, у которого высота поддеревьев каждого из узлов отличается не более, чем на 1.*

Несложно показать, что высота сбалансированного AVL-дерева с n узлами будет удовлетворять следующей оценке:

$$\log_2(n+1) - 1 \le h < 1{,}4404 \cdot \log_2(n+1) - 1{,}3277. \qquad (4.6)$$

Поскольку AVL-дерево является бинарным, то количество узлов не должно превышать величины $2^{h+1} - 1$. Следовательно, $h \ge \log_2(n+1) - 1$. Для доказательства левой части неравенства составим рекуррентное соотношение для числа узлов дерева высотой h:

$$\begin{cases} T(0) = 1,\ T(1) = 2, \\ T(h) = T(h-1) + T(h-2) + 1,\ n > 1. \end{cases} \qquad (4.7)$$

Для того чтобы найти замкнутый вид данного уравнения, сделаем замену $T(h) + 1 = F(h)$. Получим уравнение

* Аббревиатура *AVL* происходит от фамилий советских ученых Г. М. Адельсон-Вельский и Е. М. Ландис, которые впервые предложили данный тип деревьев в 1962 г.

$$\begin{cases} F(0) = 1,\ F(1) = 3, \\ F(h) = F(h-1) + F(h-2),\ n > 1, \end{cases} \tag{4.8}$$

решив которое найдем $F(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h$.

Следовательно, для числа узлов AVL-дерева высотой h справедлива следующая оценка:

$$n \geq T(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1. \tag{4.9}$$

Если учесть, что $\left|\frac{1-\sqrt{5}}{2}\right| < 1$, то полученное неравенство можно переписать в виде

$$n \geq \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h - 1,$$

откуда $h < \log_{\varphi} \frac{n+1}{1+\frac{2}{\sqrt{5}}} \approx 1{,}4404 \cdot \log_2(n+1) - 1{,}3277$.

Видим, что высота AVL-дерева есть $O(\log n)$, где n – число узлов дерева. Полученная оценка утверждает тот факт, что высота AVL-дерева никогда не превысит высоту идеально сбалансированного дерева более чем на 45 %.

Используя операции вращения, достаточно просто поддерживать инвариант, определенный для AVL-дерева.

Рассмотрим некоторое дерево B. Будем говорить, что левое его поддерево *тяжелее* правого или наоборот, если разность между их высотой больше единицы. Процедуры определения тяжести поддеревьев заданного дерева приведены ниже (процедура 4.2.10):

Процедура 4.2.10. AVL-tree: Invariant Ops

```
01:  Is Le Heavy Impl(B) return Score(B) > 1
02:  Is Ri Heavy Impl(B) return Score(B) < 1
03:
04:  Is Balanced Impl(B)
05:        return |Score(B)| ≤ 1
06:
07:  Score Impl(B)
08:        R = Root(B)
09:        with R do
10:              return Height(left) - Height(right)
```

Будем предполагать, что трудоемкость процедур 4.2.10 есть $O(1)$. Этого можно добиться, если в каждом узле дерева хранить его высоту. Операция балансировки будет заключаться в том, чтобы определить, нуждается ли дерево в балансировке вообще и, если это необходимо, осуществить некоторый поворот. Левое или правое вращение выбирается исходя из того, какое из поддеревьев тяжелее. Так, в случае левого поддерева требуется применить правый поворот, в случае правого поддерева – левый поворот. В зависимости от того, как сба-

лансированы поддеревья более тяжелого дерева, выбирается большой либо малый поворот (процедура 4.2.11):

Процедура 4.2.11. AVL-tree: Make Balanced

```
01:  Make Balanced Impl(B)
02:        if not Is Balanced(B) then
03:              R = Root(B)
04:              with R do
05:                    if Is Le Heavy(B) then
06:                          if Is Ri Heavy(left) then
07:                                return Double Rotate Right(B)
08:                          return Rotate Right(B)
09:                    if Is Ri Heavy(B) then
10:                          if Is Le Heavy(right) then
11:                                return Double Rotate Left(B)
12:                          return Rotate Left(B)
13:        return B
```

Операцию балансировки требуется выполнять всякий раз, когда в дереве происходят изменения. В операциях вставки и удаления перед непосредственным возвратом из процедуры необходимо проверить, является ли дерево сбалансированным, и, если это не так, то осуществить балансировку. В качестве примера приведем процедуру, описывающую операцию вставки в AVL-дерево (процедура 4.2.12).

Процедура 4.2.12. AVL-tree: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              if Is Empty(B) then
08:                    return Make BST(key, ∅, ∅)
09:              R = Root(B)
10:              with R do
11:                    if key < Key(R) then
12:                          result = Make BST(key, IRec(left), right)
13:                    if key > Key(R) then
14:                          result = Make BST(key, left, IRec(right))
15:              return Make Balanced(result)
```

Операция удаления по сравнению с приведенной для общего бинарного дерева поиска, будет претерпевать изменения, аналогичные операции вставки. Операция поиска в AVL-дереве будет схожа с операцией поиска для BST (процедура 4.2.2).

Недостатком рассмотренных AVL-деревьев является многократное выполнение операции балансировки для операций вставки и удаления элемента, что может привести к росту скрытой константы в оценке $O(\log n)$. Таким образом, несмотря на сравнительно высокую степень сбалансированности,

AVL-дерево не получило достаточно широкого распространения на практике. Экспериментальным путем было выяснено, что в среднем одна операция балансировки приходится на каждые две операции вставки и на каждые пять операций удаления.

4.2.2. (2-4)-дерево

(2-4)-дерево – сбалансированное дерево поиска, у которого каждый из узлов может содержать не более четырех дочерних элементов и высота поддеревьев одного и того же уровня является одинаковой (рис. 4.4).

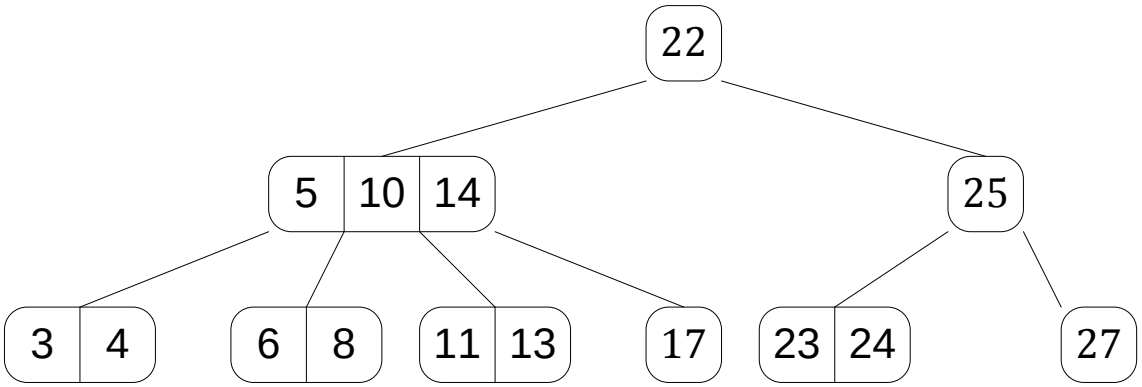

Рис. 4.4. Пример (2-4)-дерева

Согласно определению (2-4)-дерева, каждый из его узлов можно отнести к одному из трех типов: *2-узел*, *3-узел* либо *4-узел*. Отдельно выделим тип *листового узла*, который также относится к одному из трех типов, перечисленных выше, но у которого нет дочерних элементов (рис. 4.5).

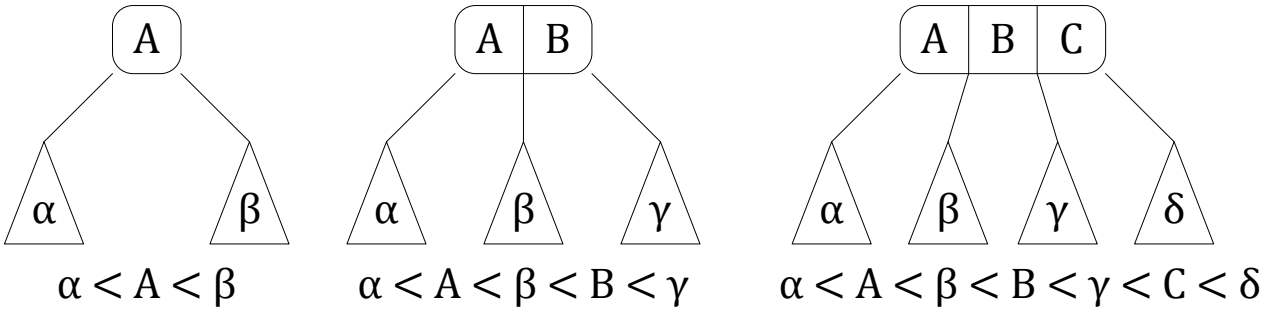

Рис. 4.5. Узлы (2-4)-дерева

Отдельно выделяются 1-узел и 5-узел, имеющие в точности один и пять потомков соответственно. Узлы данного типа могут быть использованы в целях упрощения реализации некоторых из операций, определенных над (2-4)-деревом. Наличие же 1- и 5-узлов в дереве не допускается по определению.

К основным операциям над (2-4)-деревьями относятся операции поиска элемента с заданным ключом, вставки и удаления элемента. В силу свойства сбалансированности (2-4)-дерева, все операции будут иметь трудоемкость $O(h)$, где высота дерева h удовлетворяет следующей оценке:

$$\frac{1}{2}\log_2(n+1) - 1 \le h \le \log_2(n+1) - 1 \Rightarrow h = O(\log n). \qquad (4.10)$$

Для доказательства неравенства (4.10) достаточно рассмотреть дерево, у которого все узлы являются либо 4-узлами, либо 2-узлами.

К вспомогательным операциям на (2-4)-дереве относятся операции *разделения* узла, *слияния* двух узлов и операция *перемещения*.

Операция *разделения* может применяться либо к 4-узлу (рис. 4.6), либо к 5-узлу (рис. 4.7). Если вставка элемента в (2-4)-дерево осуществляется «сверху-вниз», то обычно используют разделение 4-узлов. Для вставки элемента «снизу-вверх» используется как разделение 4-узлов, так и 5-узлов. Процедура разделения 4-узла приведена ниже (процедура 4.2.13):

Процедура 4.2.13. (2-4)–Tree: Split (4–Node Split Impl)

```
01:  Split Impl(B)
02:        R = Root(B)
03:        if Is 4–Node(R) then
04:              with R do
05:                    left  = Make 2–Node(key1, child1, child2)
06:                    right = Make 2–Node(key3, child3, child4)
07:                    return Make (2-4)-Tree (
08:                          Make 2–Node (
09:                                key2,
10:                                Make (2-4)-Tree(left),
11:                                Make (2-4)-Tree(right)))
12:        return B
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

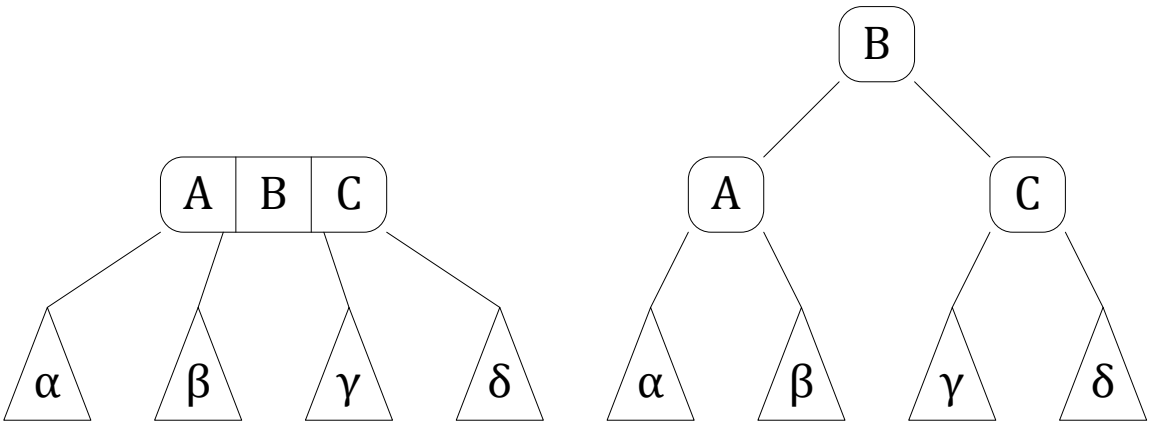

Рис. 4.6. Разделение 4-узла

Разработка процедуры разделения 5-узла не приводится по причине сходства с процедурой разделения 4-узла.

Операция *слияния*[*] применяется к двум узлам одного и того же родительского узла при условии, что один из узлов является 1-узлом (рис. 4.8). Таким образом, операция слияния может быть применена в том случае, если удаление элемента происходит из 2-листа и у удаляемого элемента брат также является

[*] В англоязычной литературе применительно к (2-4)-деревьям используется термин *fusion*.

2-листом. Следует отметить, что если родителем сливаемых узлов является 2-узел, то операцию слияния требуется *каскадировать*[*] наверх и применить к родительскому узлу (процедура 4.2.14).

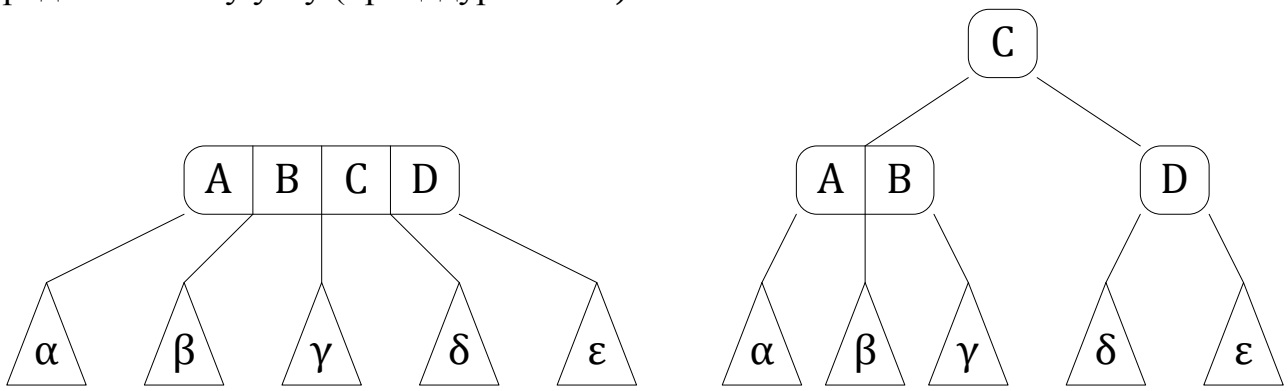

Рис. 4.7. Разделение 5-узла

Процедура 4.2.14. (2-4)-Tree: Fuse

```
01:  Fuse Impl(B)
02:        R = Root(B)
03:        if Is 1–Node(R) then
04:              if Exist Sibling(R) then
05:                    if Is 2–Node(S) then
06:                        return Complete Fuse(R, Sibling(R), Parent(R))
07:                    return Transfer(B)
08:              return Child(R)
09:        return B
10:
11:        Complete Fuse Impl(R, S, P)
12:            key1 = Get Fusing Key(S) {* Min or Max Key *}
13:            key2 = Get Fusing Key(P) {* Min or Max Key *}
14:            F = Make (2-4)-Tree(
15:                  Make 3-Node(key1, key2, Children(S) ∪ Children(R)))
16:            if Is 2-Node(P) then
17:                  return Fuse(Make (2-4)-Tree(Make 1-Node(∅, F))
18:            return Make (2-4)-Tree(
19:                       Make Node(
20:                           Keys(P) \ {key2},
21:                               Children(P) \ {T(R), T(S)} ∪ {F})
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

Отметим, что в результате выполнения операции слияния высота дерева может уменьшиться на единицу.

[*] В англоязычной литературе используется термин *cascading*.

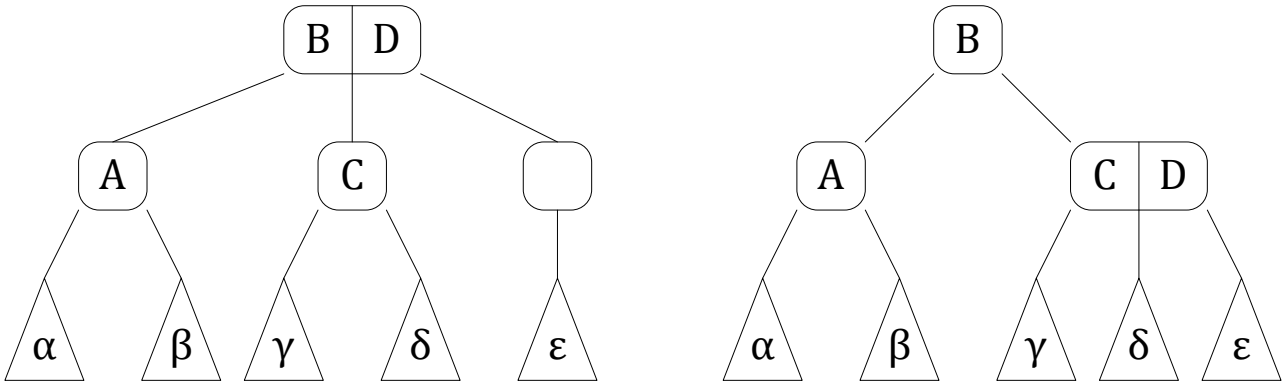

Рис. 4.8. Слияние двух узлов

Операция *перемещения*[*] (рис. 4.9), как и слияния, используется для того, чтобы удалить 1-узлы в (2-4)-дереве. Но в отличие от слияния перемещение необходимо тогда, когда братом 1-узла является 3-узел либо 4-узел. Если братом узла является 2-узел, то вызывается соответственно операция слияния узлов. Описание операции перемещения приведено в процедуре 4.2.15:

Процедура 4.2.15. (2-4)-Tree: Transfer

```
01:  Transfer Impl(B)
02:        R = Root(B)
03:        if Is 1-Node(R) then
04:              if Exist Sibling(R) then
05:                    if Is 2-Node(S) then
06:                          return Fuse(B)
07:                    return Complete Transfer(R, Sibling(R), Parent(R))
08:              return Child(R)
09:        return B
10:
11:      Complete Transfer Impl(R, S, P)
12:            key1 = Get Transferring Key(S) {* Min or Max Key *}
13:            key2 = Get Transferring Key(P) {* Min or Max Key *}
14:            FR = Make (2-4)-Tree(
15:                  Make Node(Keys(R) ∪ {key2},
16:                      Children(R) ∪ Get Appropriate Child(S, key1)))
17:            FS = Make (2-4)-Tree(
18:                  Make Node(Keys(S) \ {key1},
19:                      Children(S) \ Get Appropriate Child(S, key1)))
20:            return Make (2-4)-Tree(
21:                       Make Node(
22:                           Keys(P) \ {key2},
23:                               Children(P) \ {T(R), T(S)} ∪ {FR, FS})
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

[*] В англоязычной литературе применительно к (2-4)-деревьям используется термин *transfer*.

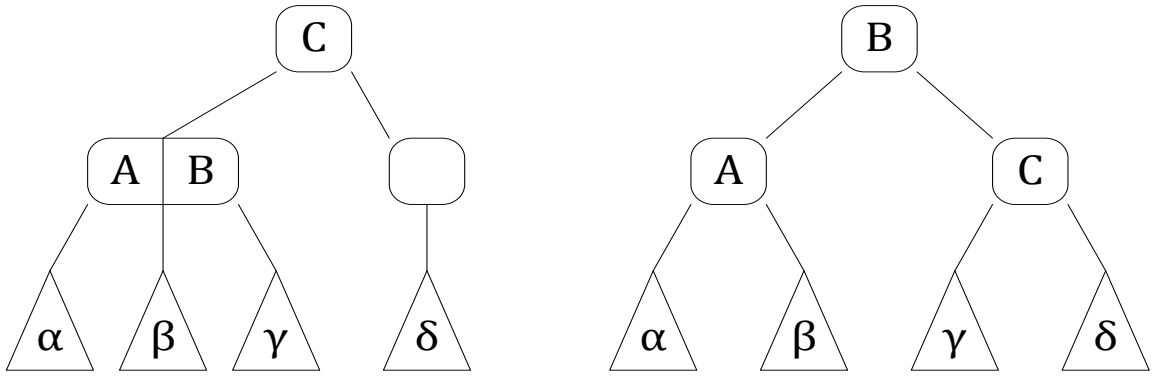

Рис. 4.9. Операция перемещения

Вставка элемента в (2-4)-дерево осуществляется только в листовой узел и может быть реализована по принципу «сверху-вниз» либо по принципу «снизу-вверх» [*]. В случае если операция вставки производится «сверху-вниз», требуется разделить каждый 4-узел, который встретится на пути от корня дерева до листа. Если операция вставки производится «снизу-вверх», необходимо осуществить «проталкивание» элементов наверх (если произошла вставка в 4-лист), причем «проталкивание» может потребоваться вплоть до корня дерева. Здесь можно воспользоваться как разделением 4-узла, так и 5-узла. Процедура вставки в (2-4)-дерево, реализованная по принципу «сверху-вниз» может выглядеть следующим образом (процедура 4.2.16):

Процедура 4.2.16. (2-4)-Tree: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              R = Root(B)
08:              if Is 4-Node(R) then
09:                    return IRec(Split(B))
10:              if Is Leaf(R) then
11:                    return Make (2-4)-Tree(
12:                                Make Node(Keys(R) ∪ {key}, ∅))
13:              return IRec(Get Child(R, key), B)
```

Трудоемкость приведенной процедуры есть $O(\log n)$. Реализация операции вставки «снизу-вверх» также будет иметь трудоемкость $O(\log n)$. Детали реализации оставляются читателю в качестве упражнения.

Для того чтобы удалить элемент с заданным ключом из (2-4)-дерева, требуется: найти необходимый узел, в котором храниться искомый элемент; заменить элемент его преемником; удалить преемника. Таким образом, операция удаления произвольного элемента из дерева сводится к удалению элемента, расположенного в одном из листьев того же дерева. Если удаление элемента

[*] В англоязычной литературе используются соответственно термины *top-down* и *bottom-up*.

приводит к образованию 1-узла, то требуется выполнить операцию перемещения либо слияния, чтобы сохранить инварианты, определенные на (2-4)-дереве. Детали реализации процедуры удаления приведены ниже (процедура 4.2.17):

Процедура 4.2.17. (2-4)-Tree: Delete

```
01:  Delete Impl(B, key)
02:        if not Contains(B, key) then
03:              return DRec Impl(B)
04:        return B
05:
06:        DRec Impl(B)
07:              R = Root(B)
08:              if Contains Key(R, key) then
09:                    if Is Leaf(R) then
10:                          return Delete From Leaf(B)
11:                    Replace Key(R, key, Key1(Successor(R, key)))
12:                    return DRec(Get GT Child(R, key), B)
13:              return DRec(Get Child(R, key))
14:
15:        Delete From Leaf Impl(L)
16:              R = Root(L)
17:              if Contains Key(R, key) then
18:                    D = Remove Key(R, Key)
19:                    if Is 1-Node(D) then
20:                          return Fuse(T(D))
21:                    return D
22:              return L
```

Трудоемкость приведенной процедуры есть $O(\log n)$. Так как процедура удаления может привести к образованию 1-узла, то после удаления искомого элемента высота дерева может уменьшиться на единицу.

В заключение отметим, что существуют так называемые (2-3)-деревья [4] и d-деревья, у которых максимум может быть d детей. Особенностью (2-3)-деревьев, как и (2-4)-деревьев, является гарантированная высота дерева $O(\log n)$, и, следовательно, гарантированная трудоемкость операций вставки и удаления $O(\log n)$. Рассмотрим полное d-дерево D, у каждого узла которого ровно d детей. Тогда высота дерева D есть $O(\frac{\log n}{\log d})$. Операция поиска элемента с заданным ключом в d-дереве в худшем случае будет иметь трудоемкость $O(\log n)$, так как помимо перемещения с более высоких уровней дерева на более низкие требуется осуществлять проверку наличия элемента внутри узла с использованием двоичного поиска, трудоемкость которого составит для d-узла $O(\log d)$. Таким образом, использование d-деревьев не позволяет улучшить асимптотику выполнения операции поиска в сравнение с (2-4)-деревьями. В то же самое время, реализация (2-4)- и (2-3)-деревьев представляется значительно проще реализации сбалансированных d-деревьев.

Также следует отметить, что, несмотря на асимптотику выполнения $O(\log n)$ всех операций, определенных на (2-4)-дереве, их использование на

практике вызвано рядом трудностей. К основным из них относится поддержка различных типов узлов, а также анализ ряда частных случаев с целью поддержания дерева в сбалансированном состоянии. По этой причине (2-4)-деревья обычно заменяют их бинарными эквивалентами (например, красно-черным деревом), трудоемкость выполнения операций на которых также есть $O(\log n)$.

4.2.3. Красно-черное дерево

Красно-черное дерево – сбалансированное бинарное дерево поиска, на котором определен следующий набор инвариантов:

1) каждый узел дерева раскрашен в красный либо в черный цвет;

2) корень и листья дерева окрашены в черный цвет;

3) у красного узла оба дочерних узла окрашены в черный цвет;

4) все пути от узла до листьев, родителем которых является этот узел, содержат одинаковое число черных узлов.

Красно-черное дерево было предложено Р. Байером в 1972 г. Название же «красно-черное» дерево было введено Л. Гибасом и Р. Седжвиком в 1978 г.

Между красно-черными деревьями и (2-4)-деревьями существует тесная связь, которая заключается в том, что одно дерево может быть получено из другого путем применения ряда несложных операций. Для того чтобы преобразовать красно-черное дерево в эквивалентное ему (2-4)-дерево, достаточно поднять красные узлы на уровень их предков – черных узлов (рис. 4.10). Для преобразования (2-4)-дерева в красно-черное дерево можно воспользоваться набором правил, приведенным на рис. 4.11. Видим, что одному и тому же (2-4)-дереву может соответствовать несколько красно-черных деревьев, поскольку 3-узел можно представить двумя способами в красно-черном дереве.

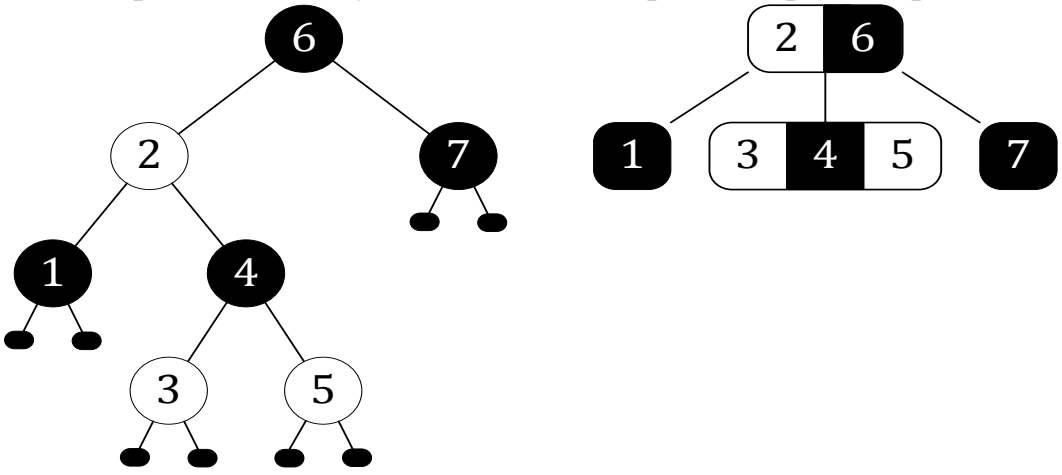

Рис. 4.10. Красно-черное дерево и эквивалентное ему (2-4)-дерево

Для того чтобы показать, что красно-черное дерево есть бинарное дерево поиска высоты $O(\log n)$, введем понятие *черной высоты* дерева. Под *черной высотой* будем понимать количество черных узлов на пути от корня дерева до любого из его листьев. Тогда число узлов, которое может содержать красно-черное дерево с черной высотой b, должно быть не менее $2^b - 1$. Так как высота дерева h не может превышать величины $2b$, то

$$h \le 2b \le 2\log_2(n+1) \Rightarrow h = O(\log n). \qquad (4.11)$$

Из соотношения (4.11) следует, что красно-черное дерево есть сбалансированное бинарное дерево поиска.

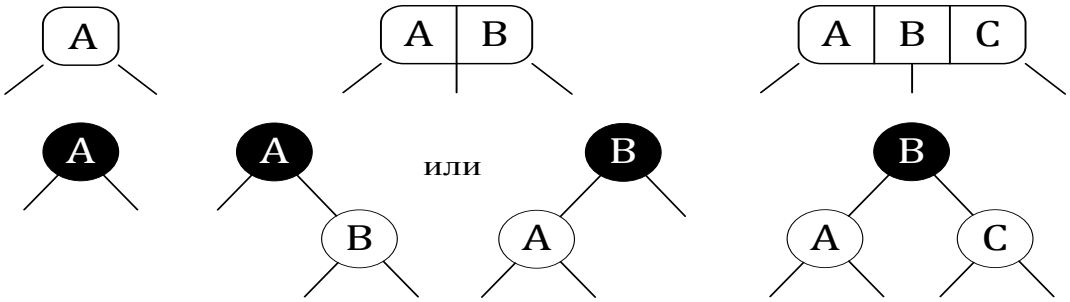

Рис. 4.11. Преобразование узлов (2-4)-дерева в узлы красно-черного дерева

Операции вставки и удаления в красно-черное дерево осуществляются, как и в обычное бинарное дерево поиска с тем лишь отличием, что после добавления либо удаления узла требуется совершить операцию балансировки. Особенностью красно-черных деревьев является то, что операция балансировки для операций добавления и удаления будет различной. За подробностями операции балансировки отправляем читателя к таким источникам как [7, 11]. Отметим лишь, что балансировка красно-черных деревьев, как и AVL-деревьев, базируется на операциях поворота.

В заключение следует отметить, что красно-черные деревья – наиболее распространенный тип BST, который используется для реализации словарных структур данных. Несмотря на то, что высота красно-черного дерева может быть выше высоты AVL-дерева, построенного для тех же элементов, использование красно-черного дерева является более предпочтительным, поскольку число поворотов, которое требуется сделать после вставки либо удаления элемента, есть $O(1)$, в то время как для AVL-дерева число поворотов может достигать величины $O(\log n)$.*

4.2.4. B-деревья

B-дерево порядка t – сбалансированное дерево поиска, на котором определен следующий набор инвариантов [13]:

1) каждый узел, кроме корня, содержит не менее $t-1$ ключей, и каждый внутренний узел имеет, по меньшей мере, t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ;

2) каждый узел, кроме корня, содержит не более $2t-1$ ключей;

3) корень содержит от 1 до $2t-1$ ключей, если дерево не пусто;

4) каждый из узлов дерева хранит ключи в отсортированном порядке;

5) все поддеревья одного и того же уровня имеют одинаковую высоту.

* Можно показать, что высота красно-черного дерева будет превышать высоту AVL-дерева не более чем на 39 %.

Из определения B-дерева следует, что рассмотренное выше (2-4)-дерево есть B-дерево с показателем 2. Несложно показать, что высота B-дерева с показателем большим единицы будет удовлетворять следующей оценке:

$$\log_{2t}(n+1) - 1 \leq h \leq \log_t \frac{(n+1)}{2} \Rightarrow h = O(\log n). \qquad (4.12)$$

При $t = 2$ соотношение (4.12) принимает вид (4.10).

Как и для (2-4)-деревьев, для B-деревьев определены операции поиска, вставки и удаления элемента, причем операции вставки и удаления базируются на операциях разделения, слияния и перемещения. Так, если требуется разделить полный узел B-дерева, то достаточно взять его средний элемент и переместить наверх. В случае операций слияния и перемещения требуется проверить, является ли братом узел, который содержит $t - 1$ ключей (или более), и в зависимости от результата проверки осуществить слияние узлов либо перемещение. Как и для (2-4)-деревьев, операция слияния узлов B-дерева может быть каскадирована наверх, если родитель сливаемых узлов «потерял» ключ.

Трудоемкость операции поиска в B-дереве в худшем случае есть $O(\log n)$ и она сопоставима со временем поиска в (2-4)-дереве либо в красно-черном дереве, поскольку проверка принадлежности искомого элемента заданному узлу в худшем случае имеет трудоемкость $O(\log_2 t)$. Несмотря на то, что использование B-деревьев не позволяет улучшить время поиска элемента с заданным ключом, они получили широкое распространение на практике для хранения больших объемов данных. Здесь следует отметить, что под большим объемом данных подразумевается такой, который не может быть размещен в оперативной памяти ЭВМ.

Наиболее широкое распространение B-деревья получили при проектировании систем управления базами данных (СУБД) и файловых систем. Так, в контексте СУБД B-деревья и его модификации используются для хранения индексов, позволяющих получить быстрый доступ к элементам с искомым ключом. Основной же целью использования B-деревьев при работе с внешней памятью является минимизация числа операций чтения / записи на диск. Таким образом, высота B-дерева как раз и определяет, сколько обращений придется сделать к диску в худшем случае для той, либо иной операции.

Следует отметить, что у B-дерева существуют модификации, которые получили названия *B*-дерева* и *B+-дерева*. В B-дереве вместе с ключом элемента может храниться только указатель на некий блок памяти, содержащий сопутствующую информацию для данного ключа. В *B+-дереве* вся информация хранится в листьях, а во внутренних узлах хранятся только ключи и указатели на дочерние узлы. Таким образом можно добиться максимально возможной степени ветвления во внутренних узлах. *B*-дерево* – модификация B-дерева, в которой каждый внутренний узел должен быть заполнен как минимум на две трети, а не наполовину, как в стандартном B-дереве. В отличие от B+-деревьев, узел в B*-дереве не разбивается на два узла, если полностью заполнен. Вместо этого

необходимо найти место в уже существующем соседнем узле, и только после заполнения обоих узлов они разделяются на три узла.

4.3. Задачи для самостоятельного решения

1. На плоскости находится выпуклый многоугольник P и набор точек S. Разработайте алгоритм, который для каждой точки p из S определит, принадлежит точка p многоугольнику P или нет. Точка p принадлежит многоугольнику P, если она лежит либо на границе, либо внутри многоугольника P.

2. Рассмотрим множество точек S на плоскости. Будем говорить, что они образуют шеренгу, если расположены на горизонтальной либо вертикальной прямой и расстояние между соседними точками равняется 1. Требуется заданное множество точек преобразовать в шеренгу, используя минимальное количество шагов. За один шаг разрешается передвинуть одну из точек на единицу по вертикали либо по горизонтали, но не в позиции, которые уже заняты.

3. Рассмотрим m отрезков. Длина отрезка i есть L_i. Требуется из заданных m отрезков сформировать n отрезков одинаковой длины, причем длина полученных отрезков должна быть максимально возможной. Отрезки допускается разрезать. Операция соединения отрезков запрещена.

4. Рассмотрим n объектов, занумерованных натуральными числами от 1 до n. Известно, что масса первого объекта есть m_1, масса n-го объекта – m_n, а масса i-го объекта $(1 < i < n) - m_i = d + (m_{i-1} + m_{i+1})/2$, где $d -$ некоторое заданное число. Разработайте алгоритм, который позволит эффективно определять массу j-го объекта для заданного j.

5. При наборе текста довольно часто возникают опечатки из-за неправильного нажатия на клавиши, например, замена букв, лишние буквы в словах либо их нехватка. В большинстве случаев эти опечатки можно исправить автоматически. В частности, существует метод проверки орфографии, основанный на поиске в словаре слов, *похожих* на проверяемые. Два слова называются *похожими*, если можно удалить из каждого слова не более одной буквы так, чтобы они стали одинаковыми, возможно пустыми. Требуется разработать алгоритм, который для каждого слова проверяемого текста определит количество похожих на него слов из заданного словаря.

6. Рассмотрим внешней файл записей, каждая из которых представляет ребро некоторого графа G и стоимость этого ребра. Разработайте алгоритм, который построит минимальное остовное дерево графа G, полагая, что объем основной памяти достаточен для хранения всех вершин графа, но не достаточен для хранения всех его ребер.

7. Хранилище данных на некотором сервере организовано как таблица, у которой две строки и n столбцов. В каждой клетке таблицы хранится информация о некотором объекте. Каждый объект, хранимый в таблице, характеризуется своим уникальным идентификатором, который является целым числом. Пронумеруем верхнюю строку таблицы числом 0, а нижнюю – числом 1. Также пронумеруем столбцы таблицы слева направо от 0 до $n - 1$ и обозначим через

$id(i,j)$ идентификатор объекта, хранимого в клетке на пересечении строки i столбца j. Известно, что данные хранятся в таблице в определенном порядке, для которого выполнены следующие условия:

1) $id(0,j) < id(1,j)$ для всех j от 0 до $n-1$ включительно;

2) $id(0,j) < id(0,j+1)$ для всех j от 0 до $n-2$ включительно;

3) $id(1,j) < id(1,j+1)$ для всех j от 0 до $n-2$ включительно.

Для того чтобы найти некоторый объект в хранилище, требуется отправить на сервер запрос, содержащий числа i,j и x. В ответ сервер посылает число -1, если $id(i,j) < x$; число 0, если $id(i,j) = x$; число 1, если $id(i,j) > x$.

Требуется разработать алгоритм, который по числу n определит количество запросов, затрачиваемых в худшем случае на поиск одного объекта при оптимальной организации процесса поиска.

8. Рассмотрим забор, которым огорожено некоторое квадратное поле размером $n \times n$. Один угол забора расположен в точке $(0,0)$, противоположный ему угол находится в точке (n,n). Стороны забора параллельны осям X и Y. Столбы, к которым крепится забор, стоят не только по углам, но и через метр вдоль каждой стороны поля, всего в заборе $4 \times n$ столбов. Столбы стоят вертикально и не имеют толщины (радиус равен нулю). Задача состоит в определении количества столбов, которые можно увидеть, находясь в определенной точке поля. Проблема заключается в том, что на поле расположено R огромных камней, и из-за них не видны некоторые из столбов. Основание каждого камня представляет собой выпуклый многоугольник ненулевой площади, вершины которого имеют целочисленные координаты. Камни стоят на поле вертикально и у них нет общих точек между собой, а также с забором. Точка, где стоит наблюдатель, лежит внутри, но не на границе поля, а также не на границе и не внутри камней.

По размеру поля, положению и форме камней на нем и месту, где стоит наблюдатель, требуется вычислить количество столбов, которые может видеть наблюдатель. Если вершина основания камня находится на одной линии с местом расположения наблюдателя и некоторым столбом, то наблюдатель не видит этот столб.

9. В городе H планируется создать сеть автобусных маршрутов с n автобусными остановками. Каждая остановка находится на некотором перекрестке. Поскольку H – современный город, на карте он представлен улицами с квадратными кварталами одинакового размера. Две из n остановок выбираются пересадочными станциями h_1 и h_2, которые будут соединены друг с другом прямым автобусным маршрутом, а каждая из оставшихся $n-2$ остановок будет непосредственно соединена маршрутом с одной из пересадочных станций h_1 или h_2 (но не с обеими), и не соединена ни с одной из оставшихся остановок.

Расстояние между любыми двумя остановками определяется как длина кратчайшего пути по улицам города. Это означает, что если остановка представлена парой координат (x,y), то расстояние между двумя остановками $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ будет равно $d(p_1,p_2) = |x_1 - x_2| + |y_1 - y_2|$. Если

остановки A и B соединены с одной и той же пересадочной станцией, например h_1, то длина пути из A в B есть $d(A,h_1)+d(B,h_1)$. Если они соединены с разными пересадочными станциями, например A с h_1 и B с h_2, то длина пути из A в B есть $d(A,h_1)+d(h_1,h_2)+d(B,h_2)$.

Проектировщики города H хотят быть уверены, что любой его житель сможет добраться до какой-либо точки города достаточно быстро. Поэтому проектировщики хотят сделать пересадочными станциями такие две остановки, чтобы в полученной сети автобусных маршрутов максимальная длина пути между любыми двумя остановками была минимальной.

Вариант P выбора пересадочных станций и соединения остановок с ними будет лучше варианта Q, если максимальная длина пути между любыми двумя остановками в варианте P будет меньше, чем в варианте Q. Требуется разработать алгоритм вычисления максимальной длины пути между любыми двумя остановками для наилучшего варианта P выбора пересадочных станций и соединения остановок с ними.

10. Рассмотрим битовую последовательность S достаточно большой длины, записанную в файл. Требуется найти битовые подпоследовательности последовательности S длиной от A до B включительно $(1 \le A, B \le 15)$, которые встречаются в S наиболее часто. Подпоследовательности могут перекрываться между собой. Предполагается, что последовательность S настолько длинна, что она не может поместиться полностью в оперативной памяти ЭВМ.

11. Рассмотрим прямоугольную металлическую платформу P размерами $n \times m$. Платформа P разделена на k прямоугольных секций. Стороны каждой из секций параллельны сторонам платформы P. Разработайте алгоритм, который позволит найти наибольший прямоугольник, полученный путем объединений не более чем s соседних секций платформы P $(1 \le s \le k)$.

12. Рассмотрим турнир по троеборью, в котором участвует n спортсменов. Сила каждого из спортсменов характеризуется тройкой чисел (a_i, b_i, c_i), где a_i – уровень силы спортсмена i в спорте a, b_i – в спорте b, c_i – в спорте c. Известно, что уровни силы всех спортсменов в рамках одного и того же спорта различны. Будем говорить, что спортсмен i *прямо побеждает* спортсмена j, если у спортсмена i по крайней мере два уровня силы больше, чем у спортсмена j. Разработайте алгоритм, который для каждого из n спортсменов определит, сможет он выиграть турнир или нет.

13. На плоскости построен план слаломной трассы так, что точка старта имеет координаты (0,0), точка финиша – координаты $(x, y), y > 0$. На трассе находятся n ворот, через которые обязательно должен проехать слаломист (пусть даже проезжая по одной из их границ). Порядок прохождения ворот соответствует порядку их описания.

Каждые ворота расположены на плане параллельно оси абсцисс и описываются точками (a_i, y_i) и (b_i, y_i), где $0 < y_1 < \cdots < y_n < y$, $a_i < b_i$. Требуется определить, какое минимальное расстояние должен преодолеть слаломист от старта до финиша, чтобы не нарушить правила прохождения ворот.

Литература

1. Okasaki, C. Purely Functional Data Structures / C. Okasaki – Cambridge University Press, New York, NY, USA, 1999. – 230 p.
2. Kaplan, H. Purely Functional, Real–Time Deques with Catenation / H. Kaplan, R. E. Tarjan – JACM, Vol. 46, Issue 5, Sept. 1999. – P. 577 – 603.
3. Driscoll, J. R. Making Data Structures Persistent / J. R. Driscoll [and others] – STOC '86, ACM New York, NY, USA, 1986. – P. 109 – 121.
4. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Ульман, Дж. Хопкрофт; пер. с англ. – М. : Вильямс, 2003. – 384 с.
5. Bender, M. A. The LCA Problem Revisited / M. A. Bender, M. Farach-Colton – LATIN'00, Springer-Verlag London, UK, 2000. – P. 88 – 94.
6. Уоррен, Г. С., мл. Алгоритмические трюки для программистов / Г. С. Уоррен, мл.; пер. с англ. – М. : Вильямс, 2007. – 288 с.
7. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен [и др.]; пер. с англ. – М. : Вильямс, 2007. – 1296 с.
8. Tarjan, R. E. Worst-case Analysis of Set Union Algorithms / R. E. Tarjan, J. van Leeuwen – JACM, Vol. 31, Issue 2, April. 1984. – P. 245 – 281.
9. Кнут, Д. Э. Искусство программирования. Том 3. Сортировка и поиск / Д. Э. Кнут; пер. с англ. – М. : Вильямс, 2011. – 824 с.
10. Shell, D. L. A High-Speed Sorting Procedure / D. L. Shell – CACM, Vol. 2, Issue 7, July 1959. – P. 30 – 32.
11. Седжвик, Р. Алгоритмы на C++ / Р. Седжвик; пер. с англ. – М. : Вильямс, 2011. – 1056 с.
12. Blum, M. Time bounds for selection / M. Blum [and others]. – JCSS Vol. 7, 1973. – P. 448 – 461.
13. Bayer, R. Organization and Maintenance of Large Ordered Indexes / R. Bayer, E. McCreight – Acta Informatica, Vol. 1, Fasc. 3, 1972. – P. 173 – 189.