

## 979 Глава 5

# 980 Задачи RMQ и LCA

981 В данном разделе мы изучим две задачи, которые неожиданным образом ока-  
982 зываются связанными: *задачу о минимуме на отрезке* и *задача о наименьшем об-  
983 щем предке*.

### 984 5.1. Задача о минимуме на отрезке

985 **Определение 5.1.1.** *Задача о минимуме на отрезке (range minimum query, RMQ):*  
986 по последовательности  $A = (a_1, \dots, a_n)$  и паре индексов  $(i, j)$ ,  $i \leq j$ , найти такой  
987 индекс  $k \in [i, j]$ , что  $a_k = \min\{a_i, \dots, a_j\}$ .

988 В такой постановке эта задача имеет сложность  $\Theta(n)$ : её можно решить за  $O(n)$   
989 *прямым вычислением*, т.е. перебрав элементы  $a_i, \dots, a_j$ , и при этом нельзя решить  
990 быстрее, т.к. для пары индексов  $(1, n)$  нам потребуется перебрать все элементы  
991 последовательности. Задача становится более интересной, если к одной и той же  
992 последовательности совершаются множество запросов. В этом случае мы можем  
993 вычислить какую-то вспомогательную информацию или использовать структу-  
994 ры данных, которые позволяют отвечать на такие запросы значительно быстрее.

#### 995 5.1.1. Динамическая задача RMQ

996 Разрешим дополнительно кроме запроса минимума на отрезке также изме-  
997 нять элементы последовательности  $A = (a_1, \dots, a_n)$  (но не их количество) при  
998 помощи запроса  $\text{change}(i, x)$ , который заменяет элемент  $a_i$  на  $x$ . В таких случа-  
999 ях принято говорить о *динамической постановке задачи RMQ*. Давайте подумаем,  
1000 какой сложности для таких запросов следует ожидать в лучшем случае. Предпо-  
1001 ложим, что на вычисление вспомогательной информации и на построение струк-  
1002 тур данных мы хотим потратить не более  $O(n)$  (на меньшее надеяться не стоит,  
1003 т.к. нам требуется прочитать все элементы последовательности). Можно доказать  
1004 следующую оценку.

1005 **Теорема 5.1.1.** *Если сложность вычисления вспомогательной информации и по-  
1006 строения структур данных не более  $O(n)$ , то сложность запросов в динамической  
1007 постановке задачи RMQ не меньше  $\Omega(\log n)$ .*

1008 **Доказательство.** Будем доказывать от обратного. Предположим, что существует  
1009 алгоритм, позволяющий выполнять запросы  $\text{minimum}(i, j)$  и  $\text{change}(i, x)$  на  
1010 массиве длины  $n$  за  $o(\log n)$  после предобработки массива функцией  $\text{rmq-preprocess}$ ,  
1011 сложность которой не более  $O(n)$ . Покажем, что в таком случае мы можем отсор-  
1012 тировать произвольных массив длины  $n$  сравнениями за  $o(n \log n)$ , что проти-  
1013 воречит нижней оценке  $\Omega(n \log n)$  на сортировку сравнениями. Пусть нам дан

1014 массив  $A$  длины  $n$ , и пусть  $M$  — это некоторое значение, которое больше любого  
 1015 элемента массива  $A$ . Рассмотрим следующую функцию, которая упорядочивает  
 1016 массив  $A$  используя алгоритм для RMQ.

```

1017 # сортирует массив используя алгоритм для RMQ
1018 def sort-using-rmq(A):
1019     n = len(A)
1020
1021     # предобработка для RMQ, не более O(n)
1022     rmq-preprocess(A)
1023
1024     # массив для результата
1025     result = []
1026
1027     for i in range(n):
1028         # вычисляем минимум для всего массива, O(log n)
1029         k = minimum(0, n - 1)
1030         # сохраняем минимум
1031         result.append(A[k])
1032         # заменяем элемент k на значение M, O(log n)
1033         change(k, M)
1034
1035     return result

```

1036 Сложность этой функции складывается из сложности предобработки  $O(n)$ , слож-  
 1037 ности  $n$  запросов `minimum` и сложности  $n$  запросов `change`. По нашему предпо-  
 1038ложению сложность любого запроса не более  $O(\log n)$ , следовательно суммарная  
 1039 сложность ограничена  $O(n \log n)$ .  $\square$

1040 Таким образом, самая оптимистичная оценка — это  $O(\log n)$  на запрос, при  
 1041 условии, что на вычисление вспомогательной информации и на построение струк-  
 1042 тур данных тратится не более  $O(n)$  операций. Такую оценку позволяет достичь  
 1043 специальная структура данных, называемая *деревом отрезков*.

1044 **Определение 5.1.2.** Дерево отрезков для массива длины  $n$  — это двоичное дере-  
 1045 во, с каждой вершиной которого ассоциирован отрезок массива по следующему  
 1046 правилу:

- 1047 • с корнем  $r$  ассоциирован отрезок  $S(r) = [1, n]$ ,
- 1048 • для каждой вершины  $v$  с отрезком  $S(v) = [i, j]$ ,
  - 1049 – если  $i = j$ , то  $v$  — лист,
  - 1050 – в противном случае  $v$  имеет два потомка  $u$  и  $w$ , причём

$$1051 \quad S(u) = [i, m] \quad \text{и} \quad S(w) = [m + 1, j], \quad \text{где} \quad m = \lfloor (i + j)/2 \rfloor.$$

1052 Легко проверить, что у дерева отрезков для массива длины  $n$  ровно  $n$  листьев —  
 1053 по одному на каждый элемент массива. Кроме того, так как отрезки во внутрен-  
 1054 них вершинах при переходе к потомкам уменьшаются вдвое, то высота такого  
 1055 дерева отрезков ограничена  $\lceil \log n \rceil$ . Будем называть отрезки, которые соотв-  
 1056 ствуют вершинам дерева, *каноническими* и докажем следующую лемму.

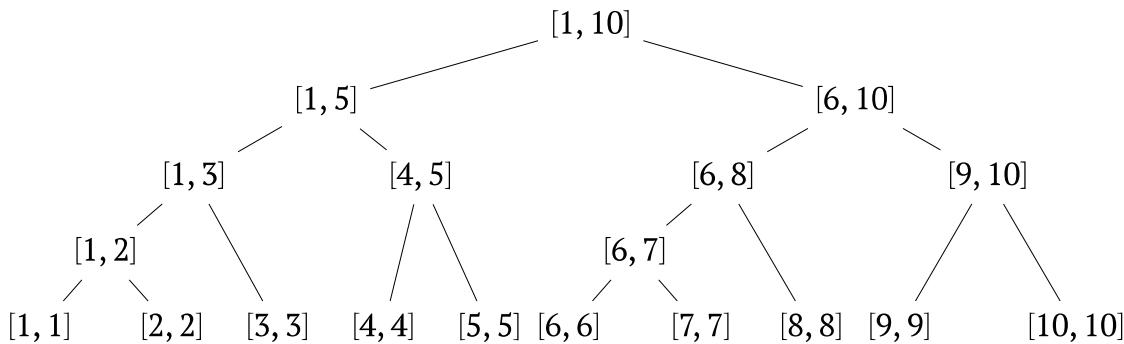


Рис. 5.1.1. Дерево отрезков для массива длины 10.

1057 **Лемма 5.1.1.** Любой отрезок  $s \subseteq [1, n]$  может быть представлен в виде объединения непересекающихся канонических отрезков  $s_1, \dots, s_k$ , где  $k = O(\log n)$ .

1059 **Доказательство.** Будем задавать отрезки парами целых чисел и в каждой вершине двоичного дерева дополнительно хранить поле `segment` — ассоциированный отрезок. Нам будет удобно воспользоваться следующей вспомогательной функцией для пересечения двух отрезков.

```

1063 # вычисляет отрезок — пересечение двух отрезков
1064 def segment_intersect((l1,r1), (l2, r2)):
1065     # левая граница — это максимум из левых границ
1066     l = max(l1, l2)
1067
1068     # правая граница — это минимум из правых границ
1069     r = min(r1, r2)
1070
1071     # если окажется, что l > r, то пересечение пустое
1072     return (l, r)
  
```

1073 Рассмотрим следующую рекурсивную процедуру, которая по отрезку  $[l, r]$  возвращает список канонических отрезков, задающий разбиение  $[l, r]$ .

```

1075 # разбивает отрезок s на множество канонических отрезков
1076 def decompose((l,r), v = root):
1077     # обрабатываем пустой отрезок
1078     if r < l:
1079         return []
1080
1081     # возвращаем канонический отрезок
1082     if (l,r) == v.segment:
1083         return [v.segment]
1084
1085     # разбиваем s
1086     s_left = segment_intersect((l,r), v.left.segment)
1087     s_right = segment_intersect((l,r), v.right.segment)
1088
1089     # выполняем рекурсивные вызовы и объединяем результаты
1090     return decompose(s_left, v.left) + decompose(s_right, v.right)
  
```

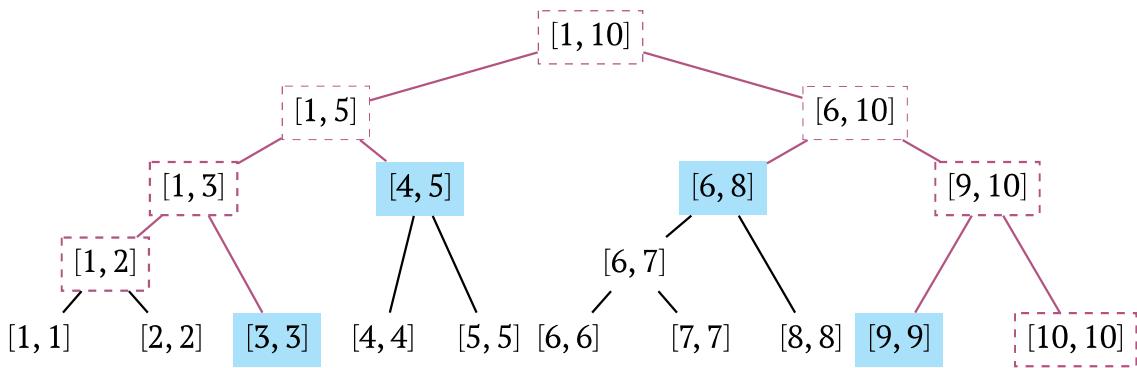


Рис. 5.1.2. Разбиение отрезка  $[3, 9]$  функцией `decompose` для массива длины 10. В результате отрезок разбивается на канонические отрезки  $[3, 3]$ ,  $[4, 5]$ ,  $[6, 8]$ ,  $[9, 9]$ . Остальные просмотренные вершины выделены пунктиром.

Сначала докажем корректность, а именно покажем, что данная процедура действительно задаёт разбиение отрезка на канонические. Заметим, что при каждом рекурсивном вызове `decompose(s, v)` поддерживается следующий инвариант: отрезок  $s$  всегда является подотрезком  $v.segment$ . Будем доказывать корректность этой процедуры по индукции по длине отрезка, ассоциированного с вершиной  $v$ . База индукции: процедура `decompose` корректна для листьев. Теперь предположим, что процедура корректна для всех вершин с отрезками длины менее  $k$ , и покажем корректность для вершины с отрезком длины  $k$ . Если мы попали в один из `if`-ов, то процедура корректна. Если же произошли рекурсивные вызовы, то по предположению индукции оба вызова вернут корректное разбиение для левой и правой частей отрезка  $s$ , что в сумме даст корректное разбиение для  $s$ .

Осталось доказать, что в разбиении получится не более  $O(\log n)$  отрезков. Сначала рассмотрим частный случай, когда один из концов отрезка  $s$  совпадает с соответствующим концом отрезка  $v.segment$ . Для определённости давайте предположим, что у  $s$  и  $v.segment$  общее начало. Тогда могут быть три варианта выполнения `decompose(s, v)`:

- если  $s == v.segment$ , то вызов завершится без рекурсивных вызовов,
- если  $s_left == v.left.segment$ , то внутри `decompose(s_left, v.left)` не будет рекурсивных вызовов, а у  $s_right$  и  $v.right$  будет общее начало,
- если  $s_left != v.left.segment$ , то отрезок  $s_right$  — пустой, следовательно внутри `decompose(s_right, v.right)` не будет рекурсивных вызовов, а у  $s_left$  и  $v.left.segment$  будет общее начало.

Другими словами, при каждом вызове `decompose` рекурсия будет продолжаться не более, чем в одной ветке. Поэтому всего будет не более  $2 \lceil \log n \rceil$  вызовов `decompose`, и следовательно, `decompose(s, v)` вернёт не более  $O(\log n)$  отрезков.

Теперь предположим, что у  $s$  и  $v.segment$  нет общих концов. В этом случае есть два варианта выполнения `decompose(s, v)`:

- если один из отрезков  $s_left$  и  $s_right$  пустой, то рекурсия продолжится только в одном из рекурсивных вызовов `decompose`,
- если оба отрезка  $s_left$  и  $s_right$  непустые, т.е. в этой вершине произошло нетривиальное разбиение отрезка  $s$  на две части, то теперь в обоих рекурсивных вызовах `decompose` один из концов отрезка совпадает с концом отрезка, ассоциированного с вершиной.

Пока мы находимся в условиях первого варианта, рекурсия продолжается только в одном из рекурсивных вызовов. Как только происходит нетривиальное разбиение отрезка, то в обоих рекурсивных вызовах мы попадаем в условия частного случая, который мы рассмотрели выше, и про который мы доказали оценку  $O(\log n)$  на количество отрезков. Поэтому в сумме в общем случае `decompose` вернёт не более  $O(\log n) + O(\log n) = O(\log n)$  канонический отрезков. □

*Замечание 5.1.1.* Может показаться, что и время работы функции `decompose` ограничено  $O(\log n)$ . Действительно, мы доказали, что функция `decompose` посещает  $O(\log n)$  вершин дерева. Однако, эта функция возвращает список, и на объединение списков этих списков в Python дополнительно потратится  $O(\log^2 n)$ .

1134 Упражнение 5.1.1. Перепишите функцию `decompose` так, чтобы её сложность была  
1135 не более  $O(\log n)$ .

1136 Применим дерево отрезков для решения поставленной задачи. Пусть дан неко-  
1137 торый массив  $A$  длины  $n$ . Построим дерево отрезков для этого массива и в каждой  
1138 вершине запишем индекс минимального элемента на соответствующем отрезке.  
1139 Отметим, что построение дерева и заполнение индексов можно организовать за  
1140  $O(n)$ : будем заполнять индексы в вершинах от листьев к корню, индекс в листе —  
1141 это номер соответствующего элемента массива, а индексом во внутренней вер-  
1142шине выбирается один из индексов её сыновей — тот, что соответствует меньше-  
му элементу массива.

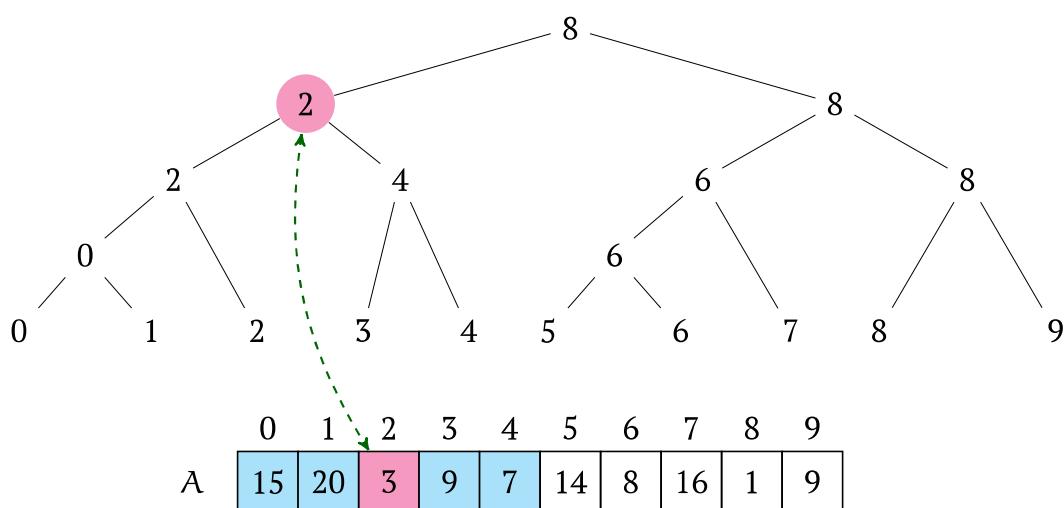


Рис. 5.1.3. Дерево отрезков с индексами минимумов для массива длины 10. Выделенный элемент хранит индекс минимального элемента среди первых пяти элементов массива

Пришло время реализовать процедуры `minimum` и `change`. Будем предполагать, что в каждой вершине хранится значение `value`, которое равно минимальному элементу на соответствующем отрезке. Процедура `minimum` реализуется аналогично процедуре `decompose`, посещает те же узлы дерева отрезков, и, следовательно, имеет сложность  $O(\log n)$ .

```
1149 # действуем аналогично процедуре decompose
1150 def minimum((l,r), v = root):
1151     # обрабатываем пустой отрезок
1152     if r < l:
1153         return None
```

```

1154
1155     # если отрезок совпал с отрезком вершины
1156     if (l,r) == v.segment:
1157         # возвращаем минимум на этом отрезке
1158         return A[v.index]
1159
1160     # разбиваем s
1161     s_left = segment_intersect((l,r), v.left.segment)
1162     s_right = segment_intersect((l,r), v.right.segment)
1163
1164     # выполняем рекурсивные вызовы
1165     left = minimum(s_left)
1166     right = minimum(s_right)
1167
1168     # объединяем результаты
1169     if left == None:
1170         return right
1171     if right == None:
1172         return left
1173
1174     # если оба вызова вернули не None
1175     if A[left] < A[right]:
1176         return left
1177     else:
1178         return right

```

Теперь опишем процедуру изменения элемента массива. При изменении элемента нам нужно обновить минимумы во всех вершинах-предках соответствующего листа, количество таких вершин не больше высоты дерева, т.е.  $O(\log n)$ .

```

1182     # изменяю элемент массива
1183     def change(i, x, v = root):
1184         # если мы пришли в соответствующий лист
1185         if v.segment == (i,i):
1186             v.value = x
1187             return
1188
1189         # рекурсивно спускаемся до соответствующего листа
1190         if i <= v.left.segment[1]:
1191             change(i, x, v.left)
1192         else:
1193             change(i, x, v.right)
1194
1195         # обновляем значение в вершине
1196         v.value = min(v.left.value, v.right.value)

```

Таким образом мы научились решать задачу RMQ в динамической постановке за  $O(n)$  на построение дерева отрезков и  $O(\log n)$  на каждый запрос. Заметим, что наша конструкция не связана на какие-либо свойства операции минимума, кроме ассоциативности. Поэтому аналогичный подход может быть использован,

1201 например, для максимума, суммы, произведения и других ассоциативных опе-  
1202 раций.

1203 **5.1.2. Статическая задача RMQ**

1204 Теперь предположим, что массив для задачи RMQ задан и никогда не меня-  
1205 ется. В этом случае принято говорить о *статической постановке задачи RMQ*. Мы  
1206 уже умеем отвечать на запрос RMQ за  $O(\log n)$  при помощи дерева отрезков. В  
1207 статической постановке мы можем получить дополнительное ускорение за счёт  
1208 *предобработки* — можно вычислить какие-то дополнительные данные о масси-  
1209 ве, когда уже известно, что дальше он изменяться не будет, и использовать их  
1210 для более быстрого ответа на запрос. В дальнейшем, говоря о сложности реше-  
1211 ния задачи RMQ, мы будем оперировать парами оценок  $(f(n), g(n))$ , где  $f(n)$  —  
1212 это сложность предобработки, а  $g(n)$  — сложность ответа на запрос RMQ. В этих  
1213 обозначениях сложность прямого вычисления  $(O(1), O(n))$ .

1214 Перед тем как перейти к решению статической задачи RMQ, рассмотрим более  
1215 простую задачу RSQ.

1216 **Определение 5.1.3.** *Задача о сумме на отрезке (range sum query, RSQ):* по после-  
1217 довательности  $A = (a_1, \dots, a_n)$  и паре индексов  $(i, j)$ ,  $i \leq j$ , вычислить сумму  
1218  $S_{i,j} = a_i + \dots + a_j$ .

1219 В статической постановке эту задачу легко решить за  $(O(n), O(1))$ . Давайте вы-  
1220 числим вспомогательные *частичные суммы*

$$1221 S_0 = 0, \quad \forall k \in [n], S_k = a_1 + \dots + a_k.$$

1222 Это легко сделать за  $O(n)$ . Теперь для ответа на запрос  $RSQ(i, j)$  нам достаточно  
1223 вычислить разность соответствующих частичных сумм:

$$1224 RSQ(i, j) = S_j - S_{i-1}.$$

1225 Для RMQ это не сработает, т.к. операция минимума не имеет обратной.

1226 **Полное предвычисление.** Начнём с наиболее простого подхода: раз уж мы от-  
1227 дельно учтываем сложность предобработки и сложность запроса, то давайте сде-  
1228 лаем все вычисления на этапе предобработки. Для этого вычислим RMQ для всех  
1229 возможных пар индексов  $(i, j)$  и запишем ответы в таблицу размера  $n \times n$ . Такое  
1230 решение будет иметь сложность  $(O(n^2), O(1))$ . (Если действовать „в лоб“ и запол-  
1231 нять эту таблицу прямым вычислением для каждого запроса, то получится и во-  
1232 все  $(O(n^3), O(1))$ , но нетрудно заметить, что такую таблицу можно заполнить за  
1233  $O(n^2)$ , если действовать немного умнее.)

1234 **Метод разреженной таблицы.** Назовём отрезок *стандартным*, если его дли-  
1235 на равна некоторой степени двойки. На этапе предобработки вычислим ответы  
1236 на запросы RMQ только для стандартных отрезков. Это потребует  $O(n \log n)$  вре-  
1237 мени и памяти (опять же такая оценка достигается, только если заполнять соот-  
1238 ветствующую табличку от меньших отрезков к большим). Как воспользоваться  
1239 этими данными для ответа на запрос RMQ для произвольного отрезка? Восполь-  
1240 зумемся следующим наблюдением.

1241 **Лемма 5.1.2.** Любой отрезок является либо стандартным, либо может быть пред-  
1242 ставлен как объединение двух пересекающихся стандартных отрезков.

1243 **Доказательство.** Пусть отрезок  $[l, r]$  не является стандартным. Выберем  $k$  такое,  
1244 что

$$2^{k+1} > r - l + 1 > 2^k.$$

1246 Такое  $k$  обязательно существует, иначе отрезок был бы стандартным. Тогда  $[l, r]$   
1247 можно представить как объединение двух стандартных отрезков длины  $2^k$ :

$$\text{1248} \quad [l, r] = [l, l + 2^k - 1] \cup [r - 2^k - 1, r].$$

1249 Можно проверить, что  $l + 2^k - 1 \geq r - 2^k - 1$ , т.е.  $l - r + 1 \geq 2^{k+1} - 1$ .  $\square$

1250 Теперь для ответа на запрос нам нужно либо вернуть значение из таблицы, ес-  
1251 ли отрезок оказался стандартным, либо представить отрезок в виде объединения  
1252 двух стандартных и взять минимум из минимумов на этих отрезках.

1253 **Замечание 5.1.2.** В методе разреженной таблицы мы пользуемся свойством *идем-*  
1254 *потентности* операции минимума, т.е. тем, что  $\min\{a, a\} = a$ . Это позволяет по-  
1255лучать ответ как минимум из минимумов на пересекающихся отрезках. Поэтому  
1256 этот подход не будет работать для операций, которые этим свойством не обла-  
1257дают. Например, метод разреженной таблицы не подходит для задачи RSQ, т.к.  
1258 сумма на отрезке не равна сумме на двух пересекающихся отрезках, которые его  
1259 покрывают. К счастью, для RSQ у нас есть более эффективное решение.

1260 **Замечание 5.1.3.** При оценке сложности этого метода предполагалось, что мы уме-  
1261ем находить ближайшую степень двойки (подходящее число  $k$  из леммы 5.1.2)  
1262 за  $O(1)$ . Этого можно достичь, заполнив таблицу размера  $n$ , но на практике это  
1263обычно реализуется при помощи битовых операций, время работы которых в  
1264этом случае (число  $n$  ограничено размером памяти) можно считать константным.

## 1265 5.2. Задача о наименьшем общем предке

1266 Теперь пришло время поговорить о второй задаче этого раздела.

1267 **Определение 5.2.1.** Задача о наименьшем/нижайшем общем предке (*least/lowest common ancestor, LCA*): по паре вершин  $(u, v)$  корневого дерева  $T$  найти их самого  
1268 нижнего общего предка. Другими словами, требуется найти наиболее удалённую  
1269 от корня вершину, лежащую одновременно и на пути от  $u$  к корню и на пути от  $v$   
1270 к корню. Поэтому, если  $v$  является предком  $u$ , то  $LCA(u, v) = v$ .

1272 Для решения этой задачи мы применим *сведение* задачи LCA к задаче RMQ, т.е.  
1273 научимся по условию задачи LCA строить условие для задачи RMQ, а потом из от-  
1274вета восстанавливать ответ для исходной задачи. Существует множество различ-  
1275ных видов сведений, некоторые из которых мы изучим, когда будем говорить о  
1276NP-трудных задачах. Следующее определение довольно неформально, но вполне  
1277достаточно для наших целей.

1278 **Определение 5.2.2.**  $(O(n), O(1))$ -сведением задачи A к задаче B будем называть  
1279тройку алгоритмов  $(F, G, H)$  таких, что

- 1280 • алгоритм  $F$  по условию для задачи A размера  $n$  строит условие для задачи B  
1281размера  $O(n)$  за время  $O(n)$ ,

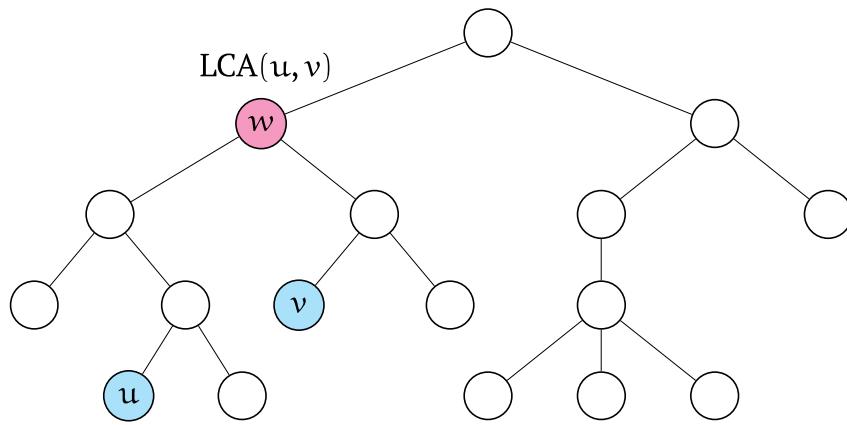


Рис. 5.2.1. Вершина  $w$  является ближайшим общим предком  $u$  и  $v$ .

- алгоритм  $G$  по запросу для задачи А строит соответствующий запрос для задачи В за  $O(1)$ ,
- алгоритм  $H$  по ответу для запроса к задаче В восстанавливает ответ к исходному запросу для задачи А за время  $O(1)$ .

**Утверждение 5.2.1.** Если существует  $(O(n), O(1))$ -сведение задачи LCA к задаче RMQ, то задача LCA решается за время  $(O(n \log n), O(1))$ .

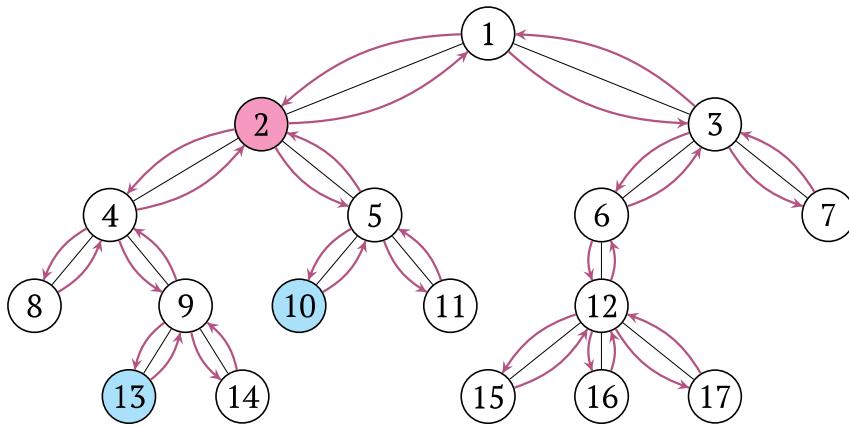
**Доказательство.** Для решения задачи LCA на дереве  $T$  с  $n$  вершинами на стадии предобработки применим алгоритм  $F$ , который по  $T$  построит массив  $A$  размера  $O(n)$  за линейное время. Для массива  $A$  построим разреженную таблицу за время  $O(n \log n)$ . На стадии запросов для каждого запроса LCA будем применять алгоритм  $G$ , вычислим полученный RMQ-запрос методом разреженной таблицы, и в завершение применим алгоритм  $H$ , который вернёт нам ответ для исходного LCA-запроса. В результате на предобработку мы потратим  $O(n \log n)$ , а каждый запрос будет обрабатываться за  $O(1)$ .  $\square$

**Сведение LCA к RMQ.** Построим сведение LCA к RMQ. Пусть нам дано корневое дерево  $T$  с  $n$  вершинами. Выполним эйлеров обход дерева  $T$ , т.е. рекурсивно обойдём дерево, проходя по каждому ребру ровно два раза (см. рис. 5.2.2), и выпишем для вершины на пути её номер и глубину в дереве.

```

# построение эйлерова обхода
def euler-traversal(v = root, depth = 0, result = None):
    # создаём список для хранения результата
    if result == None:
        result = []
    # добавим в обход текущую вершину
    result.append((v, depth))
    # рекурсивно вызовем для каждого ребёнка и объединим результаты
    for c in v.children:
        euler-traversal(c, depth + 1, result)
    result.append((v, depth))

    return result
  
```



вершина	1	2	4	8	4	9	13	9	14	9	4	2	5	10	5	11	...
глубина	0	1	2	3	2	3	4	3	4	3	2	1	2	3	2	3	...

Рис. 5.2.2. Эйлеров обход дерева  $T$  и пример вычисления  $\text{LCA}(13, 10)$  через RMQ.

1315 В результате обхода мы получим массив пар длины  $2n - 1$ . Заполним вспомогательный массив  $\text{first}$  длины  $n$ , где для каждой вершины  $v$  в ячейке с номером  $v$  запомним индекс её первого вхождения в эйлерову обход. Теперь по эйлерову обходу можно построить задачу RMQ, в которой глубина вершины будет ключом, а номер вершины — дополнительной информацией. Докажем следующую теорему.

1321 **Теорема 5.2.1.**  $\text{LCA}(v, u)$  равно вершине, на которой достигается  $\text{RMQ}(\text{first}[v], \text{first}[u])$ .

1322 Давайте для начала проверим это утверждение на примере. Сначала найдём на рис. 5.2.2 первое вхождение вершин 13 и 5. Потом на отрезке между этими двумя ячейками найдём вершину с минимальной глубиной — это будет вершина 2, и действительно она является ближайшим общим предком вершин 13 и 5. Теперь перейдём к доказательству теоремы.

1327 **Доказательство.** Давайте переформулируем условие теоремы в терминах дерева: ближайший общий предок  $u$  и  $v$  — это вершина с минимальной глубиной, которая встречается в эйлеровом обходе между первым вхождением  $u$  и первым вхождением  $v$ . Покажем сначала, что ближайший общий предок обязательно встретится между первым вхождением  $u$  и первым вхождением  $v$ . Действительно, в дереве между парой вершин есть только один простой путь, и ближайший общий предок всегда лежит на этом пути, поэтому по пути из  $u$  в  $v$  мы обязательно посетим ближайшего общего предка. Теперь покажем, что на этом пути не будет других вершин с меньшей глубиной. Предположим, что такая вершина есть. Следовательно, по пути из  $u$  в  $v$  эйлеров обход прошёл через ближайшего общего предка и поднялся выше, т.е. поднялся по ребру, по которому до этого когда-то спустился. Но ведь для того, чтобы дойти до  $v$ , необходимо будет снова спуститься по этому ребру, а такого не бывает, т.к. каждое ребро проходится ровно два раза.  $\square$

1341 Для завершения построения сведения нам осталось описать, как будут устроены алгоритмы  $F$ ,  $G$ ,  $H$ . Алгоритм  $F$  по дереву  $T$  строит его эйлеров обход и заполняет массив  $\text{first}$ . Алгоритм  $G$  преобразует запрос  $\text{LCA}(v, u)$  в запрос  $\text{RMQ}(\text{first}[v], \text{first}[u])$ .

1344 Алгоритм  $\mathsf{H}$  возвращает номер вершины, записанный в ячейке массива, на ко-  
1345 торой достигается минимум на соответствующем отрезке. Нетрудно проверить,  
1346 что эти алгоритмы имеют требуемую сложность.

1347 *Замечание 5.2.1.* Получившаяся в результате сведения задача RMQ обладает важ-  
1348 ными дополнительными свойствами, а именно любые два соседних значения от-  
1349 личаются на единицу. Задачи RMQ с таким свойством называют  $\pm 1$ -RMQ. Для  
1350 статической задачи  $\pm 1$ -RMQ существует алгоритм Фарака-Колтона — Бендера [1],  
1351 который решает её за  $(O(n), O(1))$ . Таким образом и задачу LCA можно решить за  
1352  $(O(n), O(1))$  при помощи построенного нами сведения. Но оказывается, что мож-  
1353 но пойти дальше, и построить сведение задачи RMQ к LCA (по массиву для RMQ  
1354 можно построить декартово дерево и на нём запускать алгоритм для LCA). В ре-  
1355 зультате композиции этих двух сведений можно построить сведение задачи RMQ  
1356 к  $\pm 1$ -RMQ, а следовательно решить задачу RMQ в общем случае за  $(O(n), O(1))$ .  
1357 Однако на практике этот алгоритм не используется, т.к. асимптотическое уско-  
1358 рение нивелируется большими константами и сложностью реализации.

1359 *Упражнение 5.2.1.* Постройте  $(O(n), O(1))$ -сведение задачи RMQ к задаче LCA.



1360 **Глава 6**

1361 **Деревья поиска**

1362 **6.1. АТД с быстрым поиском**

1363 **Определение 6.1.1** (множество, set). Абстрактный тип данных *множество* позволяет хранить набор однотипных элементов и реализует следующие запросы.

- 1365 • `insert(k)` — добавить  $k$  в множество.
- 1366 • `delete(k)` — удалить  $k$  из множества.
- 1367 • `find(k)` — найти  $k$  в множестве.

1368 Если разрешается хранить несколько одинаковых значений, то такой АТД называется *мультимножеством*.

1370 **Определение 6.1.2** (отображение, словарь, map, dictionary). Абстрактный тип данных *отображение* позволяет хранить набор пар “ключ-значение” и реализует следующие запросы.

- 1373 • `insert(k, v)` — добавить значение  $v$  с ключом  $k$ .
- 1374 • `delete(k)` — удалить запись с ключом  $k$ .
- 1375 • `find(k)` — найти запись с ключом  $k$ .

1376 Если разрешается хранить несколько пар с одинаковыми ключами, то такой АТД называется *мультиотображение*.

1378 Заметим, что имея реализацию множества, мы можем реализовать и отображение: будем хранить в множестве пары и сравнивать их только по первому полю. Поэтому в дальнейшем мы будем говорить только о реализации множества, понимая при этом, что все полученные результаты справедливы и для отображения.

1383 Нас будет интересовать эффективная реализация множества в случае, если на элементах множества задан порядок (т.е. элементы множества можно сравнивать). Для эффективной реализации множества применяются деревья поиска.

1386 **Замечание 6.1.1.** Если множество или отображение не изменяется после начального заполнения (т.е. сначала выполняются все запросы `insert()`, а потом поступают только запросы `find()`), то можно реализовать эти структуры на упорядоченном массиве: отсортируем массив после заполнения и будем искать элементы двоичным поиском.

## 1391 6.2. Представление корневых деревьев в памяти

1392 Перед тем, как говорить о деревьях поиска, полезно обсудить, как можно хра-  
 1393 нить корневые деревья в памяти компьютера. Существует два основных подхода.

- 1394 • **Хранение в массиве.** Такой способ мы применяли для хранения двоичной  
 1395 кучи. Этот способ хорошо подходит для хранения двоичных деревьев, кото-  
 1396 рые являются полными или почти полными. В этом случае корень хранится  
 1397 в ячейке с номером 1, его сыновья в ячейках 2 и 3, и в общем случае для вер-  
 1398 шины в ячейке  $i$  номера ячеек сыновей и предка вычисляются по формулам:

$$1399 \quad \text{left}(i) = 2i, \quad \text{right}(i) = 2i + 1, \quad \text{parent}(i) = \lfloor i/2 \rfloor.$$

1400 Такой подход можно обобщить на деревья любой ограниченной арности.  
 1401 Преимущество данного подхода в отсутствии необходимости хранить ссыл-  
 1402 ки и отсутствии фрагментации памяти. Однако, если дерево сильно отлича-  
 1403 ется от полного, то в данном представлении будет слишком много пустых  
 1404 ячеек, т.е. память будет использовать неэкономно.

- 1405 • **Хранение в виде набора записей.** При таком способе хранения каждая  
 1406 вершина задаётся записью со ссылками на потомков, родителя и другими  
 1407 данными. Само дерево задаётся ссылкой на корень. При таком подходе не  
 1408 важно, является ли дерево полным, — количество используемой памяти про-  
 1409 порционально количеству вершин в дереве. Более того, данный подход поз-  
 1410 воляет хранить деревья произвольной арности — для этого в вершинах мож-  
 1411 но хранить массив или список потомков. К минусам данного подхода сто-  
 1412 ит отнести возможную фрагментацию памяти (зависит от способа хране-  
 1413 ния записей). Кроме того, могут потребоваться дополнительные затраты на  
 1414 управление памятью.

1415 **Представление «левый ребёнок — правый сосед».** Как мы уже отметили, пред-  
 1416 ставление корневого дерева в виде набора записей позволяет хранить деревья  
 1417 произвольной арности. Однако, в некоторых случаях желательно, чтобы каждая  
 1418 вершина имела фиксированный размер. В таких случаях можно использовать пред-  
 1419 ставление «левый ребёнок — правый сосед». В этом случае каждая вершина хра-  
 1420 нит две ссылки: на левого ребёнка и на правого соседа. Такое представление, с  
 1421 одной стороны, сохраняет фиксированным размер вершины, а с другой стороны  
 1422 позволяет эффективно перемещаться по дереву, как если бы мы хранили потом-  
 1423 ков вершины в виде списка.

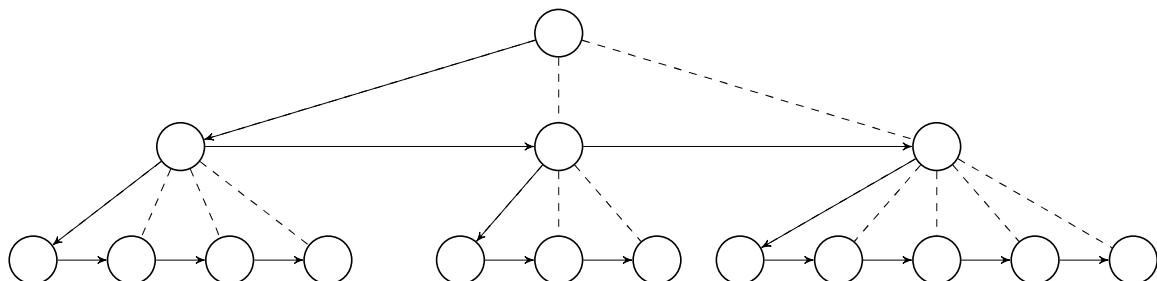


Рис. 6.2.1. Представление «левый ребёнок — правый сосед».

<sub>1424</sub> **6.3. Двоичные деревья поиска**

<sub>1425</sub> **Определение 6.3.1.** Двоичное дерево поиска — это структура данных, которая пред-  
<sub>1426</sub>ставляет корневое двоичное дерево, состоящее из однотипных узлов. Для значе-  
<sub>1427</sub>ний, которые хранятся в узлах, поддерживается следующий инвариант: элемент  
<sub>1428</sub>в узле не меньше всех элементов в его левом поддереве и не больше всех элемен-  
<sub>1429</sub>тов в его правом поддереве.

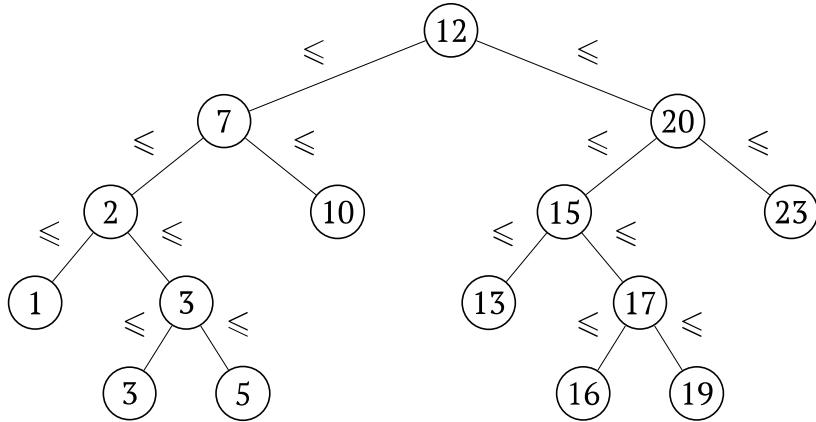


Рис. 6.3.1. Двоичное дерево поиска.

<sub>1430</sub> Для реализации в каждом узле будем хранить следующие поля:

- <sub>1431</sub> • key — элемент множества,
- <sub>1432</sub> • left — левый потомок,
- <sub>1433</sub> • right — правый потомок,
- <sub>1434</sub> • parent — родитель.

<sub>1435</sub> Само же дерево будет задаваться указателем на корень.

<sub>1436</sub> **6.3.1. Операции поиска**

<sub>1437</sub> Начнём с рекурсивного обхода дерева, при котором элементы будут переби-  
<sub>1438</sub>раться в порядке возрастания (это т.н. *inorder обхода дерева*).

```

# inorder обход дерева
def inorder-traverse(v = root):
    if v == None:
        return
    inorder-traverse(v.left)
    print v.key
    inorder-traverse(v.right)
  
```

<sub>1446</sub> **Утверждение 6.3.1.** Дерево  $T$  является деревом поиска тогда и только тогда, когда  
<sub>1447</sub>его *inorder* обход печатает ключи в неубывающем порядке.

<sub>1448</sub> Тогда поиск элемента можно описать следующим псевдокодом.

```

1449 # Поиск элемента в дереве поиска
1450 def find(k, v = root):
1451     if v == None or k == v.key:
1452         return v
1453     if k < v.key:
1454         return find(k, v.left)
1455     else:
1456         return find(k, v.right)

```

1457 Структура дерева поиска позволяет так же легко найти, например, минимальный  
1458 или максимальный элемент.

```

1459 # Поиск минимального элемента
1460 def min_element(v = root):
1461     if v.left == None:
1462         return v
1463     return min_element(v.left)

1464 # Поиск максимального элемента
1465 def max_element(v = root):
1466     if v.right == None:
1467         return v
1468     return max_element(v.right)

```

1470 Заметим, что предыдущие функции возвращают не элемент множества, а узел  
1471 дерева. Имея ссылку на элемент дерева, можно перейти к следующему или преды-  
1472 дущему элементу.

```

1473 # Поиск следующего элемента
1474 def succ(v):
1475     if v.right != None:
1476         return min_element(v.right)
1477     while v.parent != None and v.parent.left != v:
1478         v = v.parent
1479     return v.parent

1480 # Поиск предыдущего элемента
1481 def pred(v):
1482     if v.left != None:
1483         return max_element(v.right)
1484     while v.parent != None and v.parent.right != v:
1485         v = v.parent
1486     return v.parent

```

### 1488 6.3.2. Добавление элемента

1489 Будем всегда добавлять новый элемент в качестве листа дерева. Поэтому нам  
1490 нужно запустить поиск по дереву и найти подходящего родителя, т.е. такой узел,  
1491 у которого нет одного из сыновей и его значение находится в правильном соот-  
1492 ношении с добавляемым элементом.

```

1493 # Добавление нового элемента
1494 # NB: если одинаковые элементы запрещены,
1495 # то наличие элемента нужно проверить отдельно вызовом find
1496 def insert(k):
1497     newnode = node(k)
1498     if root != None:
1499         p = root
1500         while True:
1501             if k < p.key:
1502                 if p.left == None:
1503                     p.left = newnode
1504                     newnode.parent = p
1505                     return newnode
1506                 p = p.left
1507             else # k >= p.key
1508                 if p.right == None:
1509                     p.right = newnode
1510                     newnode.parent = p
1511                     return newnode
1512                 p = p.right
1513     else:
1514         root = newnode
1515     return newnode

```

### 1516 6.3.3. Удаление элемента

1517 Заметим, что никаких проблем с удалением листа нет — у листа нет потомков, которые нужно куда-то перевешивать. Теперь предположим, что у удаляемой вершины нет одного из потомков, например левого. Тогда после удаления этой вершины её место займёт её правый потомок.

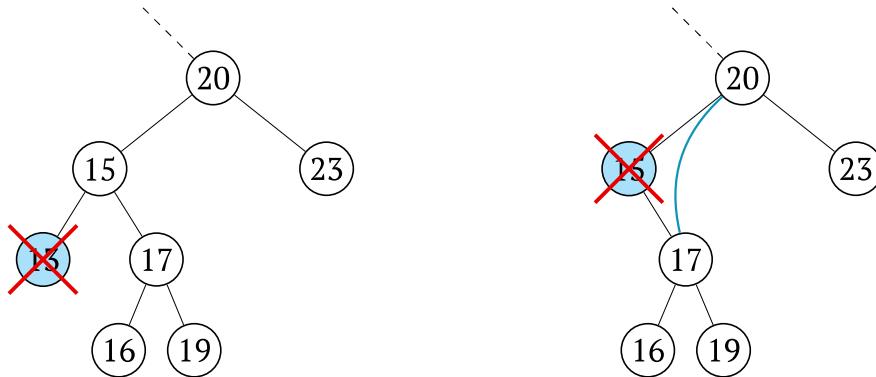


Рис. 6.3.2. Удаление листа и удаление вершины с одним потомком.

1520 Осталось понять, что делать в случае удаления вершины, у которой есть оба
1521 потомка. Пусть мы хотим удалить некоторую вершину  $v$ , у которой есть оба по-
1522 томка. Тогда в поддереве правого потомка обязательно будет вершина  $u = \text{succ}(v)$ ,
1523 причём у вершины  $u$  гарантированно не будет левого потомка (следующая за  $v$ 
1524 вершина — это минимальная вершина в поддереве её правого потомка). Обме-
1525 няя вершины  $u$  и  $v$  местами и удалим  $v$  одним из описанных выше способов

<sup>1527</sup> (теперь у вершины  $v$  нет левого потомка). Нетрудно заметить, что в результате дерево останется деревом поиска.

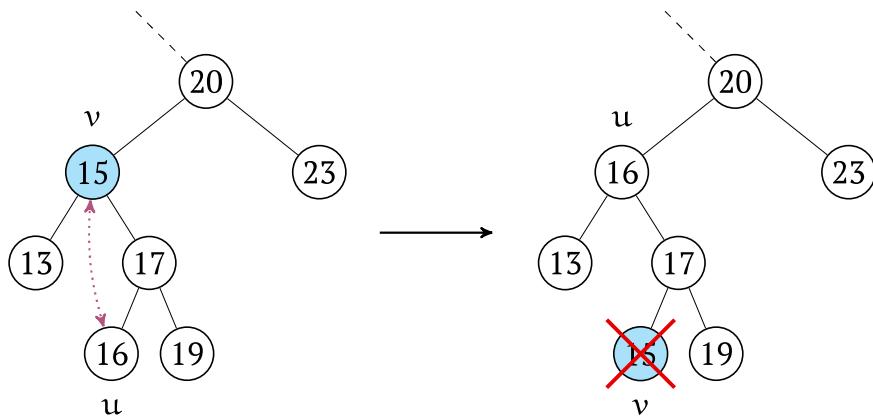


Рис. 6.3.3. Удаление вершины с двумя потомками.

1528

### Упражнение 6.3.1. Написать соответствующий код для удаление вершины.

**Утверждение 6.3.2.** Все рассмотренные операции с деревом работают за  $O(h)$ , где  $h$  – высота дерева.

Действительно, все рассмотренные выше операции делают не более одного спуска и не более одного подъёма по дереву. При этом в худшем случае высота дерева с  $n$  элементами может достигать  $n - 1$  (например, если последовательно добавлять в дерево поиска целые числа от 1 до  $n$ , то мы получим „бамбук“ длины  $n - 1$ ), т.е. операции с таким деревом будут работать за  $O(n)$ .

Для того, чтобы использование дерева поиска было разумным, нам нужно по-  
заботиться о том, чтобы дерево имело высоту поменьше, тогда и операции будут  
выполняться быстрее. Минимальная высота двоичного дерева достигается, если  
оно *полное*, то есть все уровни дерева, кроме последнего, заполнены. Высота пол-  
ного двоичного дерева с  $n$  вершинами равна  $\lceil \log(n + 1) \rceil - 1$ , т.е. время работы  
всех рассмотренных операций будет не более  $O(\log n)$ .

Однако, поддерживать дерево в таком состоянии очень затратно (можно придумать примеры, когда при удалении или добавлении вершины нам придётся перевесить множество других вершин, чтобы снова сделать дерево полным). Вместо полноты двоичного дерева можно накладывать более слабые требования — требования *сбалансированности*, которые позволяют получить такую же асимптотическую оценку  $O(\log n)$  на время работы операций, но при этом можно поддерживать дерево в сбалансированном состоянии, тратя на это не более  $O(\log n)$  на каждую операцию. Существует несколько реализаций *сбалансированных деревьев поиска*, наиболее известные из которых — *красно-чёрные деревья* и *AVL-дерево*. Описание красно-чёрных деревьев можно посмотреть в [2] — на практике их использую чаще, но эта структура данных значительно более сложна в описании, нежели AVL-дерево.

Можно заметить, что оценка  $O(\log n)$  на стоимость *всех* операции для дерева поиска — это лучшее, чего можно добиться. Действительно, пусть мы изобрели какое-то особенное дерево, для которого все операции работают за  $O(\log n)$ . Тогда мы можем построить дерево для произвольного набора элементов  $a_1, \dots, a_n$  за  $O(n \log n)$  ( $n$  раз добавить вершину за  $O(\log n)$ ). Если мы теперь обойдём это дерево (это можно сделать за  $O(n)$  операций), то таким образом получим элементы

1561  $a_1, \dots, a_n$  в порядке возрастания, т.е. мы научились сортировать произвольные  
 1562 элементы за  $O(n \log n)$ , что противоречит нижней оценке на сортировку сравне-  
 1563 ниями.

## 1564 6.4. АВЛ-дерево

### 1565 6.4.1. Сбалансированность

1566 АВЛ-дерево [3] придумано советскими учёными Г.М. Адельсоном-Вельским и  
 1567 Е.М. Ландисом. Будем называть двоичное дерево *сбалансированным*, если у лю-  
 1568 бой его вершины высота её левого поддерева отличается от высоты её правого  
 1569 поддерева не более чем на 1.

1570 **Утверждение 6.4.1.** В сбалансированном дереве высоты  $h$  не более  $2^h - 1$  вершины.

1571 *Доказательство.* Равенство достигается в случае полного двоичного дерева.  $\square$

1572 **Утверждение 6.4.2.** В сбалансированном дереве высоты  $h$  не менее  $1.6^h$  вершин.

1573 *Доказательство.* Пусть  $M_h$  — это минимальное количество вершин в сбаланси-  
 1574 рованном дереве высоты  $h$ . Тогда  $M_h = 1 + M_{h-1} + M_{h-2}$ . Заметим, что это ре-  
 1575 куррентное соотношение очень похоже на соотношения для чисел Фибоначчи:  
 1576  $F_n = F_{n-1} + F_{n-2}$ . Легко проверить, что  $M_h = F_{h+3} - 1$ . Используя оценку для чи-  
 1577 сел Фибоначчи  $F_n \geq ((\sqrt{5} + 1)/2)^n > 1.6^n$ , получаем  $M_h \geq F_{h+3} - 1 \geq 1.6^{h+3} - 1 >$   
 1578  $1.6^h$ .  $\square$

1579 **Следствие 6.4.1.** Сбалансированное дерево с  $n$  вершинами имеет высоту  $\Theta(\log n)$ .

1580 *Доказательство.* Прологарифмируем  $2^h > n > 1.6^h$ .  $\square$

1581 Мы показали, что свойства сбалансированности достаточно для того, чтобы  
 1582 высота дерева была логарифмической, а следовательно, операции поиска, добав-  
 1583 ления и удаления вершины работали бы за  $O(\log n)$ . Осталось разобраться, как  
 1584 поддерживать дерево сбалансированным после добавления или удаления верши-  
 1585 ны.

### 1586 6.4.2. Вращения

1587 Для поддержания дерева в сбалансированном состоянии применяются локаль-  
 1588 ные коррекции дерева, называемые *вращениями*. Вращения применяются, если  
 1589 после добавления или удаления вершины, дерево перестало быть сбаланси-  
 1590 рованным. Вращение позволяет восстановить нарушенный баланс и при этом де-  
 1591 рево остаётся деревом поиска.

1592 **Утверждение 6.4.3.** При добавлении (удалении) вершины в сбалансированное дере-  
 1593 во, баланс мог нарушиться только у тех вершин, которые лежат на пути от корня  
 1594 к добавленной (удалённой) вершине. При этом разница высот поддеревьев в таких  
 1595 вершинах не может быть больше 2.

1596 Пусть  $x$  — первая вершина на пути от добавленной/удалённой вершины к кор-  
 1597 ню, в которой нарушился баланс. Для восстановления баланса к ней нужно при-  
 1598 менить одно из двух вращений.

1599 **Малые вращения.** Применяются в случае, когда левое поддерево левого по-  
 1600 томка имеет высоту на 2 больше, чем правое поддерево  $x$ , и в симметричном слу-  
 1601 чае. Пусть  $y$  — левый потомок  $x$ . При *малом правом вращении*  $y$  занимает место  $x$ ,  
 1602 а  $x$  в свою очередь становится её правым потомком и забирает одно из поддер-  
 1603 евьев, см. рис. 6.4.1 (высота поддерева  $\beta$  может совпадать с высотой  $\alpha$  или быть на  
 1604 1 меньше). При этом дерево остаётся деревом поиска. Для того, чтобы проверить  
 1605 это, выпишем отношения вершин этого дерева до и после вращения и в обоих  
 случаях мы получим одинаковый порядок  $\alpha \leq y \leq \beta \leq x \leq \gamma$ .

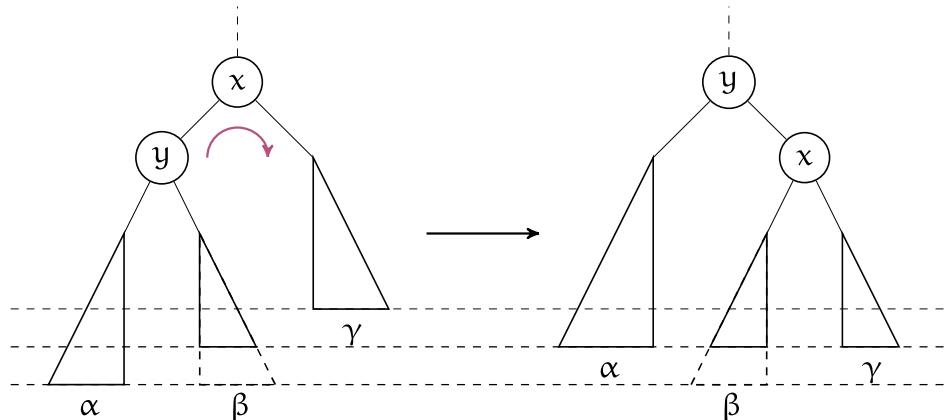


Рис. 6.4.1. Малое правое вращение АВЛ-дерева.

1606

1607 **Большое вращение.** Большие вращения применяются в случаях, когда малые  
 1608 вращение не могут исправить баланс, т.е. когда правое поддерево левого потомка  
 1609  $x$  имеет высоту на 1 больше, чем высота правого поддерева  $x$ , и в симметричном  
 1610 случае. Большое вращение складывается из двух малых вращений, см. рис. 6.4.2  
 1611 (высота одного из поддеревьев  $\beta$  и  $\gamma$  должна совпадать с высотой  $\alpha$ , а у второго  
 1612 может быть такой же или на 1 меньше). Можно проверить, что в этом случае од-  
 1613 ним малым вращением не исправить баланс в вершине  $x$ . Большое вращение —  
 1614 это композиция двух малых вращений, каждое из которых сохраняет дерево де-  
 1615 ревом поиска, поэтому в результате большого вращения дерево так же остается  
 деревом поиска.

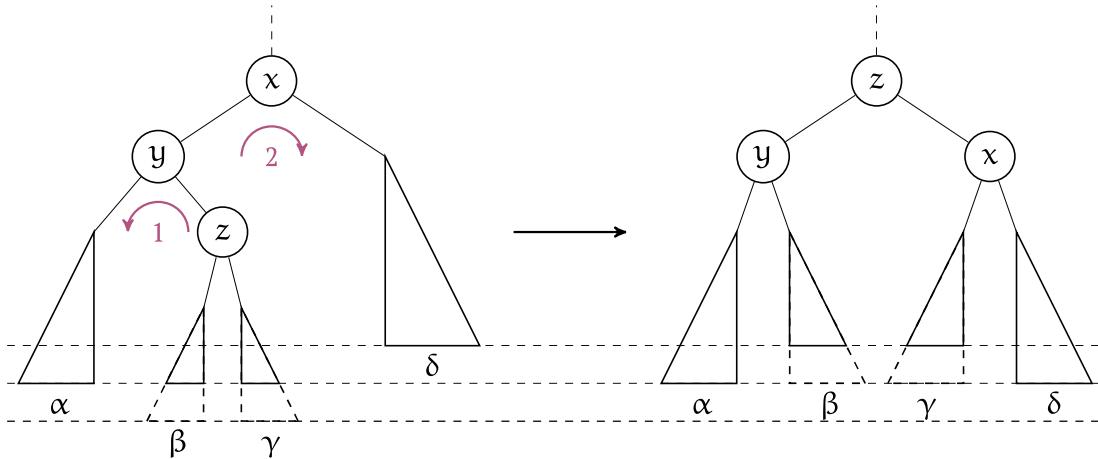


Рис. 6.4.2. Большое правое вращение АВЛ-дерева.

1616

1617 **Утверждение 6.4.4.** Малое и большое вращения покрывают все возможные случаи  
 1618 нарушения баланса, когда высота поддеревьев отличается на 2.

1619 **Восстановление сбалансированности.** Теперь опишем как происходит вос-  
 1620 становление сбалансированности при помощи вращений. Предположим, что мы  
 1621 добавили или удалили какую-то вершину из дерева. Будем подниматься от этой  
 1622 вершины к корню, пока не встретим какую-нибудь вершину, в которой баланс на-  
 1623 рушен. Пусть  $x$  — такая первая вершина. Так как мы добавили или удалили только  
 1624 одну вершину, то разница высот поддеревьев  $x$  не может быть больше 2, а значит,  
 1625 мы можем применить вращение, чтобы восстановить баланс. При этом вращения  
 1626 не могут нарушить баланс ниже по дереву, поэтому если после восстановления  
 1627 баланса в  $x$  в дереве и остались несбалансированные вершины, то они находятся  
 1628 выше по дереву на пути к корню. Будем продолжать подниматься и исправлять  
 1629 баланс во всех встречающихся несбалансированных вершинах.

1630 **Замечание 6.4.1.** В данном рассуждении пропущен важный момент: мы не объ-  
 1631 яснили, почему не может случиться так, что после нескольких вращений баланс  
 1632 в одной из вершин-предков нарушится более чем на 2. Для того, чтобы это объ-  
 1633 яснить, нужно сделать следующие наблюдения.

- 1634 • Вращения могут только уменьшить высоту какого-то под дерева. Поэтому  
 1635 интересующая нас ситуация, когда у какой-то вершины выше баланс нару-  
 1636 шился более чем на 2, могла произойти только при удалении.
- 1637 • При удалении вершины баланс нарушается только в одной вершине (при  
 1638 добавлении может нарушиться в нескольких). И в дальнейшем, каждое вра-  
 1639 щение, которое исправляет это дисбаланс может нарушить баланс только в  
 1640 одной вершине выше, т.к. может только уменьшить высоту.

1641 Из этого можно сделать вывод, что такая ситуация невозможна.

1642 **Оценка времени работы.** Количество вращений не больше, чем вершин на пу-  
 1643 ти от листа к корню, т.е.  $O(\log n)$ . Каждое вращение можно выполнить за  $O(1)$ ,  
 1644 т.к. оно затрагивает константное количество вершин. В сумме получается, что  
 1645 на восстановление баланса после изменяющей дерево операции нам потребует-  
 1646 ся  $O(\log n)$ .

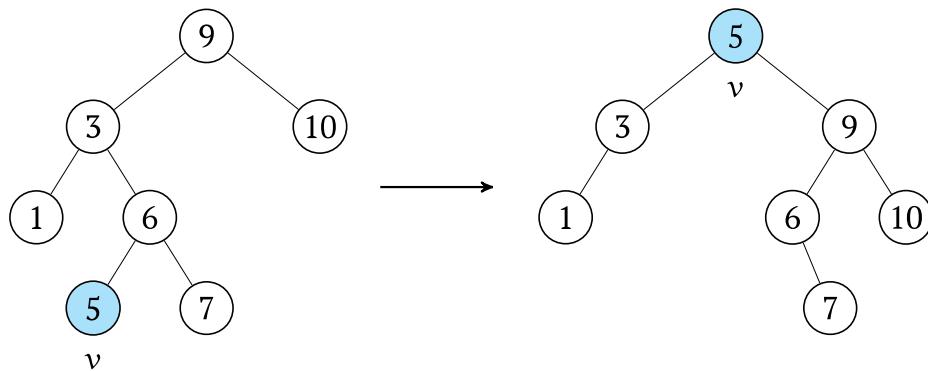
1647 **Замечание 6.4.2.** Для эффективной реализации АВЛ-дерева в каждой вершине  
 1648 следует хранить и поддерживать высоту её под дерева.

## 1649 6.5. Splay-дерево

1650 *Splay-дерево* — это самобалансирующееся двоичное дерево поиска, изобре-  
 1651 тёйное Слеатором и Тарьяном [4, 1] в 1983. В отличие от АВЛ-дерева, которое  
 1652 обеспечивает оценки в худшем, splay-дерево имеет амортизированную оценку  $O(\log n)$   
 1653 на время работы основных операций. При этом реализовать splay-дерево немно-  
 1654 го проще, чем АВЛ-дерево, т.к. не нужно хранить и поддерживать высоту под-  
 1655 дерева в каждой вершине, а также не нужно рассматривать разные случаи при  
 1656 удалении вершины.

### 1657 6.5.1. Запросы к splay-дереву

1658 Ключевую роль в splay-дереве играет операция  $splay(v)$ , которая перестра-  
 1659 ивает дерево при помощи вращений так, чтобы вершина  $v$  оказалась в корне.

Рис. 6.5.1. Работа операции `splay(v)`.

Более подробно устройство этой операции мы рассмотрим позже, а пока с её помощью научимся реализовывать другие операции.

Операция `splay` вызывается для каждой вершины, которая была найдена, добавлена или удалена.

```

# Поиск элемента в splay-дереве
def splay-find(k, v = root):
    if k == v.key:
        splay(v)
        return v
    if k < v.key:
        if v.left == None:
            splay(v)
            return None
        return splay-find(k, v.left)
    else:
        if v.right == None:
            splay(v)
            return None
        return splay-find(k, v.right)

# Добавление вершины в splay-дерево
def splay-insert(k):
    # вызов insert для обычного дерева поиска
    v = insert(k)
    splay(v)
    return v

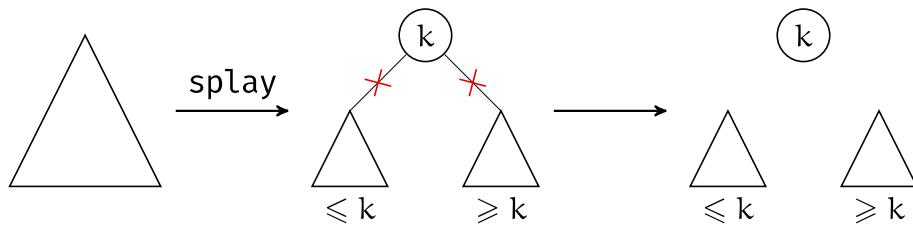
```

Для реализации удаления давайте введём две вспомогательные операции — `split` и `merge`. Операция `split(k)` дерево на три части: дерево с элементами  $\leq k$ , вершина с ключом  $k$ , дерево с элементами  $\geq k$ .

```

# Разбиение дерева относительно ключа k
# предполагается, что ключ k в дереве есть
def splay-split(k):
    v = splay-find(k)
    T1 = v.left
    T2 = v.right

```

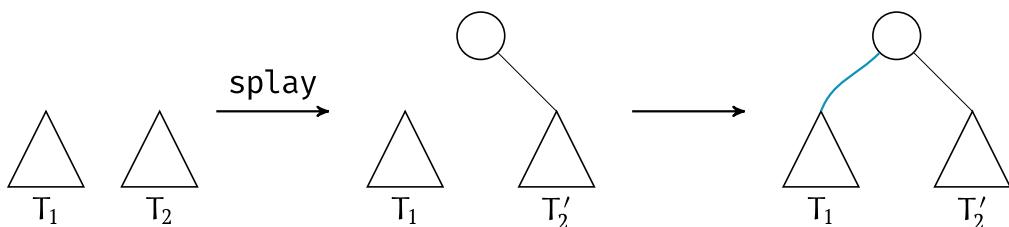
Рис. 6.5.2. Работа операции splay-split( $k$ ).

```

1695     v.left = None
1696     v.right = None
1697     return (T1, v, T2)

```

1698 Операция merge объединяет два дерева в одно (требуется, чтобы все ключи первого дерева были меньше ключей второго дерева).

Рис. 6.5.3. Работа операции splay-merge( $T_1, T_2$ ).

```

1699
1700 # Слияние двух деревьев
1701 # ключи T1 меньше ключей T2
1702 def splay-merge(T1, T2):
1703     v = min_element(T2)
1704     splay(v)
1705     v.left = T1
1706     return v

```

1707 Теперь можно описать операцию удаления вершины.

```

1708 # Удаление вершины с ключом k
1709 def splay-remove(k):
1710     (T1, v, T2) = splay-split(k)
1711     return splay-merge(T1, T2)

```

1712 Как мы увидим далее, сложность операции splay ограничена  $O(h)$ , где  $h$  —  
1713 высота дерева, соответственно, верно следующее утверждение.

1714 **Утверждение 6.5.1.** Все рассмотренные операции имеют сложность не более  $O(h)$ .

## 1715 6.5.2. Операция splay

1716 Мы научились реализовывать все необходимые операции со splay-деревом,  
1717 кроме самой операции splay, которая используется в каждой из рассмотренных  
1718 операций как подпроцедура. Теперь мы рассмотрим устройство операции splay  
1719 и докажем амортизированную оценку  $O(\log n)$  на её сложность. Начнём со сле-  
1720 дующих двух утверждений.

1721 **Лемма 6.5.1.** *Истинная стоимость  $splay(v)$  пропорциональна глубине вершины  
1722  $v$ .*

1723 Эта лемма сама по себе не является особенно примечательной, т.к. реализация  
1724 операции  $splay$  „в лоб“ при помощи малых вращений для АВЛ дерева имела бы  
1725 такую же оценку (каждое малое вращение уменьшает глубину вершины на 1).

1726 Вторая лемма значительно более интересная.

1727 **Лемма 6.5.2.** *Существует целочисленная функция потенциала  $\Phi$ , определённая на  
1728 двоичных деревьях и принимающая неотрицательные значения, такая, что*

- 1729 (a) *для любого дерева с  $n$  вершинами  $\Phi(T) = O(n \log n)$ ,*
- 1730 (b) *учётная стоимость любой операции  $splay$  относительно  $\Phi$  не более  $O(\log n)$ ,*
- 1731 (c) *добавление листа увеличивает потенциал не более чем на  $\log n$ ,*
- 1732 (d) *сумма потенциалов деревьев, получившихся из  $T$  удалением корня, меньше ис-  
1733 ходного потенциала  $T$  не более чем на  $\log n$ ,*
- 1734 (e) *при подсоединении одного дерева к корню другого потенциал получившегося де-  
1735 рева не более чем на  $\log n$  превосходит сумму потенциалов исходных деревьев.*

1736 Вместе лемма 6.5.1 и пункты (a) и (b) леммы 6.5.2 позволяют сделать следую-  
1737 щее неожиданное заключение. Предположим, что мы начали с некоторого  $splay$ -  
1738 дерева с  $n$  вершинами и  $m \geq n$  раз выполнили следующую операцию: нашли  
1739 самую глубокую вершину и вызвали для неё  $splay$ . По лемме 6.5.2 суммарная (ис-  
1740 тинная) стоимость этих операций не будет превосходить  $O(m \log n) + O(n \log n) =$   
1741  $O(m \log n)$ . С другой стороны мы ведь каждый раз вызывали  $splay$  для самой глу-  
1742 бокой вершины, а значит, по лемме 6.5.1 количество операций было пропорцио-  
1743 нально высоте дерева. Получается, что и высота дерева в среднем была  $O(\log n)$ ,  
1744 т.е. за счёт вызовов  $splay$  дерево самобалансируется и „в среднем“ имеет неболь-  
1745 шую высоту.

1746 Теперь покажем, как можно обобщить это умозаключение на  $m$  произволь-  
1747 ных (из рассмотренных выше) операций со  $splay$ -деревом. Все рассмотренные  
1748 операции устроены более-менее одинаково: мы сначала спускаемся по дереву  
1749 до некоторой вершины, а потом вызываем для неё  $splay$ . Рассмотрим подроб-  
1750 ннее операцию  $splay\text{-}find$ : в поиске ключа мы спускаемся от корня вниз пока  
1751 не найдём подходящую вершину или не убедимся, что её нет, а потом вызываем  
1752  $splay$  от вершины, на которой поиск завершился. Заметим что если мы спусти-  
1753 лись на глубину  $d$ , то  $splay$  совершил  $\Theta(d)$  операций, чтобы поднять вершину в  
1754 корень (по лемме 6.5.1). Таким образом, сколько бы операций спуска по дереву  
1755 мы не сделали, примерно столько же операций подъёма по дереву произойдёт  
1756 внутри  $splay$  (тут важно, что если мы не нашли вершину, то мы всё-равно вызы-  
1757 ваем  $splay$  последней вершиной в поиске). Получается, что стоимость  $splay\text{-}find$   
1758 лишь в константу раз больше стоимости вызова  $splay$  внутри неё. Соответствен-  
1759 но, если над деревом размера  $n$  выполнить  $m \geq n$  операций, некоторые из ко-  
1760 торых  $splay$ , а некоторые —  $splay\text{-}find$ , то суммарная сложность получится не  
1761 более  $O(m \log n)$ , т.е. в среднем  $O(\log n)$  на каждую операцию.

1762 Можно ли перенести это рассуждение и на другие операции? Рассмотрим, на-  
1763 пример,  $splay\text{-}insert$ . Аналогичные рассуждения показывают, что истинная сто-  
1764 имость  $splay\text{-}insert$  лишь в константу раз больше, чем истинная стоимость  $splay$   
1765 внутри неё. Однако тут мы не учли, что  $splay\text{-}insert$ , в отличие от  $splay\text{-}find$ ,

изменяет дерево, а значит, изменяет и его потенциал, и таким образом учётная стоимость `splay-insert` может оказаться значительно больше, чем учётная стоимость `splay`. Тут нам нужно воспользоваться пунктом (c) леммы 6.5.2 о том, что добавление листа изменяет потенциал не более чем на  $\log n$ , а значит, учётная стоимость `splay-insert` тоже ограничена  $O(\log n)$ .

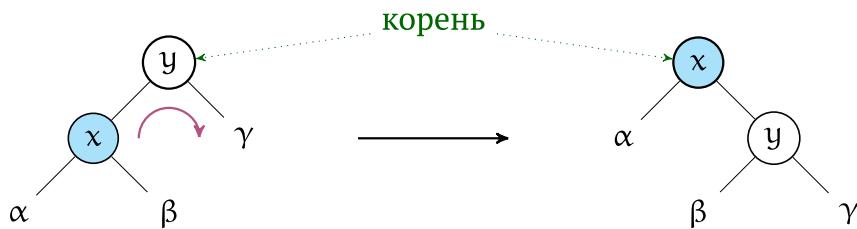
Аналогичные рассуждения позволяют показать, что учётная стоимость операции `splay-remove` не превышает  $O(\log n)$ . Удаление вершины состоит из двух спусков по дереву, из двух вызовов `splay` для соответствующих вершин, удалении корня и подсоединении одного дерева к корню другого. Осталось сказать, что сложность спусков не более чем в константу раз превышает сложность соответствующих операций `splay`, а операции удаления корня (пункт (d) леммы 6.5.2) и подсоединения одного дерева к корню другого (пункт (e) леммы 6.5.2) суммарно изменяют потенциал не более чем на  $\log n$ .

Таким образом, если над `splay`-деревом выполнить  $m \geq n$  произвольных операций, в процессе которых в дереве никогда не будет более  $n$  элементов, то суммарная сложность будет не более  $O(m \log n)$ , т.е. в среднем  $O(\log n)$  на каждую операцию.

Осталось разобраться в устройстве функции и доказать леммы 6.5.1 и 6.5.2.

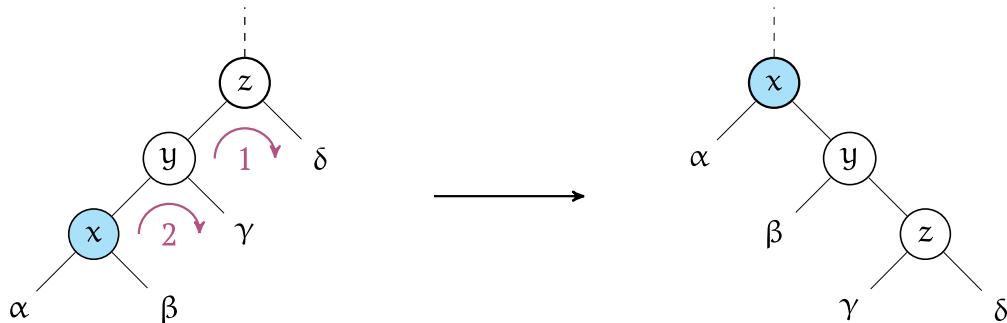
**Вращения.** Как уже было сказано, операция `splay` основана на вращениях. Опишем её работу для вершины  $x$ . Если  $x$  является корнем, то ничего делать не нужно. В противном случае применяется один из трёх шагов-вращений (или один из симметричных им).

- **zig-шаг.** Применяется только в том случае, если предок вершины  $x$  является корнем дерева.



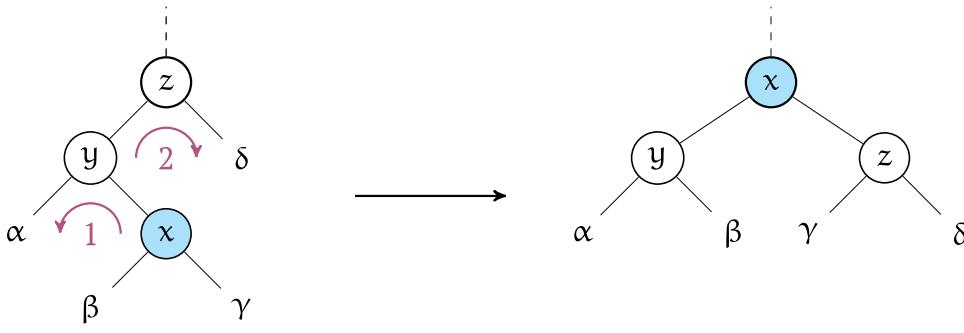
1790

- **zigzag-шаг.** Применяется, если  $x$  является правым сыном и левым внуком одновременно.



1793

- **zigzig-шаг.** Применяется, если  $x$  является левым сыном и левым внуком одновременно.



1796

1797 **Замечание 6.5.1.** При реализации zigzag и zigzag-шагов **важен порядок вращений**.  
 1798 Если бы в zigzag-шаге можно было бы поменять вращения местами, то не  
 1799 было бы необходимости рассматривать zigzag и zigzag-шаги отдельно, ведь они  
 1800 соответствовали бы двум zig-шагам. Но в этом случае получится другая структу-  
 1801 ра данных, про которую лемму 6.5.2 уже доказать не получается.

### 1802 6.5.3. Анализ сложности

1803 Из описания операции splay следует лемма 6.5.1 (количество вращений про-  
 1804 порционально глубине, а каждое вращение требует не более  $O(1)$  операций). До-  
 1805 кажем лемму 6.5.2.

1806 **Доказательство леммы 6.5.2.** Для каждой вершины  $x$  определим *вес*  $w(x)$  равным  
 1807 количеству вершин в поддереве с корнем  $x$ . Теперь мы можем определить функ-  
 1808 цию *потенциала для вершины*  $\Phi(x)$  равной  $\lfloor \log w(x) \rfloor$ . И наконец, определим по-  
 1809 тенциал для всего дерева как сумму потенциалов его вершин.

1810 Потенциал каждой вершины не превосходит  $O(\log n)$ , соответственно, потен-  
 1811 циал дерева не превосходит  $O(n \log n)$ . Таким образом  $\Phi$  удовлетворяет требова-  
 1812 нию пункта (а).

1813 Рассмотрим один вызов функции  $splay(x)$  для дерева с корнем  $r$ . После вы-  
 1814 зова функции  $splay(x)$  потенциал вершины  $x$  изменится на  $\Phi_0(r) - \Phi_0(x)$ , где  
 1815  $\Phi_0$  обозначает функцию потенциала до вызова  $splay$ . Мы покажем, что учётная  
 1816 стоимость этой операции относительно  $\Phi$  не превосходит  $3(\Phi_0(r) - \Phi_0(x)) + 1$ . По-  
 1817 скольку потенциал любой вершины не превосходит  $\log n$ , то и учётная стоимость  
 1818  $splay$  будет ограничена  $3 \log n + 1 = O(\log n)$ .

1819 Разобьём операцию  $splay$  на шаги в соответствии с её определением и оценим  
 1820 учётную стоимость каждого шага по-отдельности. Пусть  $\Phi$  обозначает функцию  
 1821 потенциала до совершения шага, а  $\Phi'$  — после. Тогда утверждается, что

- 1822 1. учётная стоимость zig-шага не превосходит  $3(\Phi'(x) - \Phi(x)) + 1$ .
- 1823 2. учётная стоимость zigzag-шага не превосходит  $3(\Phi'(x) - \Phi(x))$ .
- 1824 3. учётная стоимость zigzag-шага не превосходит  $3(\Phi'(x) - \Phi(x))$ .

1825 После вызова  $splay$  вершина  $x$  станет корнем дерева, поэтому суммируя все эти  
 1826 оценки по всем шагам мы получаем, что учётная стоимость  $splay(x)$  не превосхо-  
 1827 дит  $3(\Phi_0(r) - \Phi_0(x)) + 1$  (мы пользуемся тем, что в  $splay$  не более одного zig шага  
 1828 и тем, что потенциал корня не изменяется).

1829 Начнём с оценки zig-шага. Истинная стоимость любого вращения  $O(1)$ . Для  
 1830 простоты будем считать, что она равна 1. Тогда учётная стоимость zig-шага скла-  
 1831 дывается из его истинной стоимости и приращения потенциала дерева, т.е. равна

$$1832 \tilde{c} = 1 + \Phi'(x) + \Phi'(y) - \Phi(x) - \Phi(y).$$

1833 Здесь  $\Phi'(x) = \Phi(y)$  — это потенциал корня, поэтому  $\Phi'(y) \leq \Phi'(x)$ . Подставляем  
1834 и получаем требуемую оценку

$$1835 \quad \tilde{c} = 1 + \Phi'(y) - \Phi(x) \leq 1 + \underbrace{\Phi'(x) - \Phi(x)}_{\geq 0} \leq 1 + 3(\Phi'(x) - \Phi(x)).$$

1836 Теперь докажем оценку для zigzag-шага — для zigzag-шага оценка доказыва-  
1837 ется аналогично. Выпишем учётную стоимость шага:  $\tilde{c} = 1 + \Delta\Phi$ , где

$$1838 \quad \Delta\Phi = \Phi'(x) + \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) - \Phi(z).$$

1839 Воспользуемся тем, что потенциал корня не изменяется, т.е.  $\Phi'(x) = \Phi(z)$ , а также  
1840 тем, что

$$1841 \quad \Phi'(z) \leq \Phi'(y) \leq \Phi'(x) \quad \text{и} \quad \Phi(y) \geq \Phi(x). \quad (6.5.1)$$

1842 Получаем, что

$$1843 \quad \Delta\Phi = \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) \leq 2(\Phi'(x) - \Phi(x)).$$

1844 Рассмотрим два случая. Пусть сначала  $\Phi'(x) > \Phi(x)$ , тогда

$$1845 \quad \tilde{c} \leq 1 + 2(\Phi'(x) - \Phi(x)) \leq 3(\Phi'(x) - \Phi(x)) \quad (6.5.2)$$

1846 (здесь мы пользуемся целочисленностью потенциала), и утверждение доказано.

1847 Теперь предположим, что  $\Phi'(x) = \Phi(x)$  (меньше быть не может, т.к. после это-  
1848 го шага  $x$  стал корнем поддерева). В этом случае неравенство (6.5.2) уже невер-  
1849 но. Однако это не значит, что доказываемое утверждение неверно, т.к. неравен-  
1850 ство (6.5.2) получилось применением неравенств (6.5.1), и в этот момент мы мог-  
1851 ли что-то потерять. Действительно, покажем, что в этом случае

$$1852 \quad \Delta\Phi < 0 = 2(\Phi'(x) - \Phi(x)) = 3(\Phi'(x) - \Phi(x)),$$

1853 а следовательно,

$$1854 \quad \tilde{c} = 1 + \Delta\Phi < 1 + 3(\Phi'(x) - \Phi(x)) \leq 3(\Phi'(x) - \Phi(x)).$$

1855 Проведём доказательство от обратного. Предположим, что  $\Phi'(x) = \Phi(x)$  и  $\Delta\Phi =$   
1856 0, т.е. что все неравенства (6.5.1) на самом деле являются равенствами. Тогда по-  
1857 лучается, что все шесть потенциалов  $\Phi'(x), \Phi'(y), \Phi'(z), \Phi(x), \Phi(y), \Phi(z)$  равны меж-  
1858 ду собой и равны какому-то числу  $k$ . Покажем, что так не бывает. Пусть  $w$  и  $w'$   
1859 обозначают веса вершин до и после шага. Заметим, что после шага веса подде-  
1860 ревьев  $\alpha, \beta, \gamma, \delta$  не изменились — могли измениться только веса вершин  $x, y, z$ .  
1861 Тогда

$$1862 \quad \begin{aligned} w'(x) &= w'(\alpha) + w'(\beta) + w'(\gamma) + w'(\delta) + 3 \\ &= (1 + w(\alpha) + w(\beta)) + (1 + w'(\gamma) + w'(\delta)) + 1 \\ &= w(x) + w'(z) + 1 \geq 2^k + 2^k + 1 \geq 2^{k+1}, \end{aligned} \quad (6.5.3)$$

1863 а значит,  $\Phi'(x) \geq k + 1$ , что противоречит нашему предположению.

1864 Осталось доказать пункты (c)–(e).

1865 (c) Добавление листа увеличивает вес всех вершин на пути от корня к листу. У  
1866 скольких из них при этом может увеличиться потенциал? Только у тех, вес  
1867 которых стал равен  $2^k$  для некоторого целого  $k$ . Так как вес вершин на пути  
1868 от листа к корню строго возрастает в промежутке 0 до  $n$ , то таких вершин  
1869 не может быть более  $\log n$ .

1870 (d) Удаление корня уменьшает суммарный потенциал деревьев на потенциал  
 1871 корня, т.е. не более чем на  $\log n$ .

1872 (e) Подсоединение одного дерева к корню другого увеличивает потенциал кор-  
 1873 ня, а он не может превышать  $\log n$ .  $\square$

1874 Упражнение 6.5.1. Докажите оценку для zigzag-шага.

1875 Упражнение 6.5.2. Проверьте, что если изменить порядок вращений в zigzag-шаге,  
 1876 то желаемое утверждение не получится доказать.

1877 Если немного обобщить функцию потенциала в этой лемме, то можно дока-  
 1878 зать интересное свойство splay-деревьев.

1879 **Лемма 6.5.3.** Пусть в splay-дереве хранятся  $n$  ключей  $k_1, k_2, \dots, k_n$ . Рассмотрим  
 1880 набор из  $m \geq n$  запросов splay-find таких, что каждый ключ запрашивается хо-  
 1881 тя бы один раз. Для каждого  $i \in [n]$  обозначим через  $q_i \geq 1$  количество запросов  
 1882 ключа  $k_i$ . Тогда суммарное количество операций на выполнение всех запросов будет  
 1883 ограничено  $O(m + \sum_i q_i \log(m/q_i))$ .

*Доказательство.* Давайте посмотрим, как изменится доказательство леммы 6.5.2, если каждой вершине  $x$  назначить вес  $\rho(x) \geq 1$  и определить  $w(x)$  как сумму весов всех вершин в поддереве с корнем  $x$ . Тогда потенциал  $\Phi(x)$  вер-  
 шины  $x$  будет ограничен  $O(\log(\sum_{i=1}^n \rho(i)))$ . При таком определении  $w$  неравен-  
 ство (6.5.3), остаётся верным:

$$\begin{aligned} w'(x) &= w'(\alpha) + w'(\beta) + w'(\gamma) + w'(\delta) + \rho(x) + \rho(y) + \rho(z) \\ &= (\rho(x) + w(\alpha) + w(\beta)) + (\rho(z) + w'(\gamma) + w'(\delta)) + \rho(y) \\ &\geq w(x) + w'(z) + 1 \geq 2^k + 2^k + 1 \geq 2^{k+1}, \end{aligned}$$

1884 Таким образом учётная стоимость операции splay для вершины  $x$  будет не пре-  
 1885 восходить

$$1886 3(\Phi_0(r) - \Phi_0(x)) + 1 \leq 3 \left( \log \sum_{i=1}^n \rho(i) - \log \rho(x) \right) + 1.$$

1887 Отсюда суммарное количество операций на  $m$  запросов не будет превосходить  
 1888  $3 \sum_{i=1}^n q_i \cdot (\Phi_0(r) - \Phi_0(i)) + m$ . Положим теперь  $\rho(i) = q_i$  для всех  $i \in [n]$ . При таком  
 1889 выборе весов получаем финальную оценку

$$1890 O\left(m + \sum_{i=1}^n q_i \cdot \log \frac{m}{q_i}\right) = O\left(m + m \cdot H\left(\frac{q_1}{m}, \dots, \frac{q_n}{m}\right)\right).$$

1891  $\square$

1892 **Следствие 6.5.1** (Статическая оптимальность splay-дерева). Суммарное количе-  
 1893 ство операций на выполнение  $m$  запросов поиска в splay-дереве не больше, чем в лю-  
 1894 бом статическом двоичном дереве поиска.

1895 *Доказательство.* Заметим, что оптимальное количество операций достигается  
 1896 на соответствующем дереве кода Хаффмана с частотами  $\frac{q_1}{m}, \frac{q_2}{m}, \dots, \frac{q_n}{m}$ .  $\square$

1897 **6.6. Декартово дерево**

1898 **6.6.1. Определение и теорема существования**

1899 **Определение 6.6.1.** *Декартово дерево (cartesian tree) — это бинарное дерево, в*  
 1900 *каждой вершине которого хранится пара значений (ключ, приоритет), и которое*  
 1901 *является одновременно деревом поиска по ключу и кучей по приоритету. Впер-*  
 1902 *вые рассмотрены Вулемином [5] в 1980.*

1903 Совершенно не очевидно, что декартово дерево существует для любого набора  
 1904 пар ключей и приоритетов. Поэтому мы начнём со следующей теоремы.

1905 **Теорема 6.6.1.** *Для любого набора входных пар  $(k_1, p_1), (k_2, p_2), \dots, (k_n, p_n)$  суще-*  
 1906 *стует декартово дерево.*

1907 **Доказательство.** Мы предложим алгоритм построения декартова дерева. Пред-  
 1908 положим, что пары уже отсортированы по ключу, т.е.  $k_i < k_{i+1}, \forall i \in [n - 1]$  (в  
 1909 противном случае отсортируем их). Теперь найдём пару с максимальным прио-  
 1910 ритетом, пусть это пара  $(k_i, p_i)$ . Назначим вершину  $(k_i, p_i)$  корнем дерева и ре-  
 1911 курсивно вызовем алгоритм на  $(k_1, p_1), \dots, (k_{i-1}, p_{i-1})$  и  $(k_{i+1}, p_{i+1}), \dots, (k_n, p_n)$ ,  
 1912 получая таким образом левое и правое поддерево соответственно. Заметим те-  
 1913 перь, что в корне выполняются одновременно свойство дерева поиска и свойство  
 кучи. Следовательно, в результате мы получим декартово дерево (см. 6.6.1).  $\square$

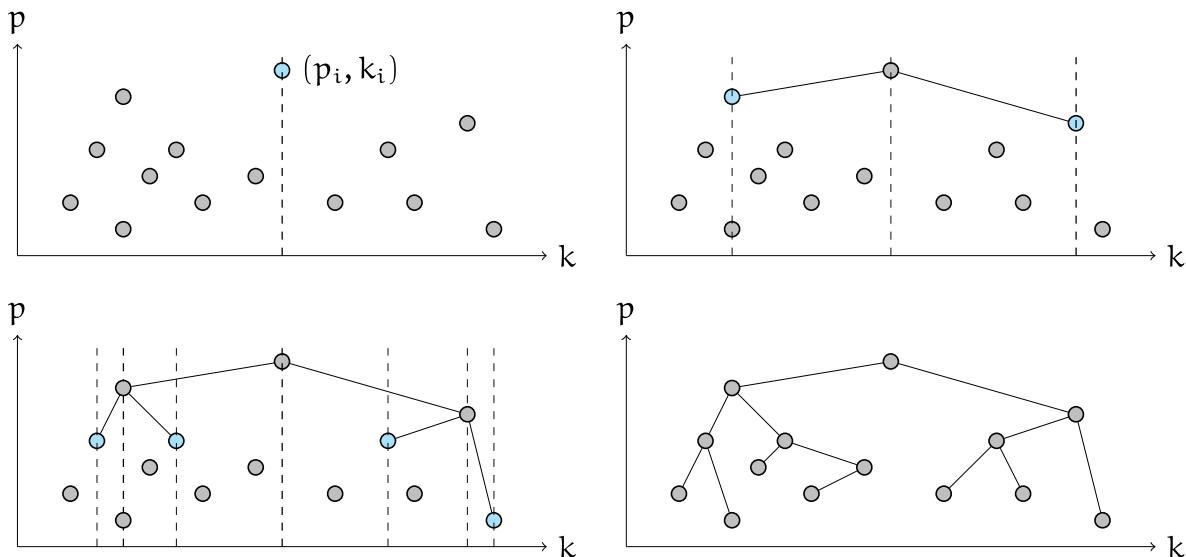


Рис. 6.6.1. Геометрическая интерпретация построения декартова дерева.

1914

1915 Давайте теперь оценим время полученного алгоритма: мы  $n$  раз ищем мак-  
 1916 симум, т.е. получается  $O(n^2)$ . Легко заметить, что этот алгоритм действительно  
 1917 будет работать за квадратичное время, если, например, приоритеты тоже отсор-  
 1918 тированы по возрастанию. Кроме того, этот алгоритм построения позволяет до-  
 1919 казать следующее утверждение.

1920 **Утверждение 6.6.1.** *Если все ключи и приоритеты различны, то декартово дерево*  
 1921 *определенено однозначно.*

Действительно, в этом случае на каждой итерации корнем будет выбираться вершина с наибольшим приоритетом, и только эта вершина может быть корнем, иначе нарушится свойство кучи. Аналогично, уникальность ключей гарантирует, что после выбора корня каждая вершина однозначно будет отнесена либо к левой, либо к правой части.

### 6.6.2. Эффективное построение

Нам уже известно, что в общем случае построить дерево поиска быстрее чем за  $\Theta(n \log n)$  не получится. Кроме того, мы умеем строить АВЛ-дерево и сплайдерево за  $O(n \log n)$ . Для декартона же дерева у нас пока есть только квадратичный в худшем случае алгоритм. В этом разделе мы разработаем алгоритм, который будет работать за линейное время при условии, что входные данные уже отсортированы по ключу (в противном случае нам придётся предварительно отсортировать входные данные, и алгоритм построения будет работать за  $O(n \log n)$ ).

Будем добавлять вершины в дерево в порядке возрастания ключа. Каждая следующая вершина имеет ключ не меньше предыдущих, а следовательно, её можно будет добавить самой последней вершиной в самой правой ветке. Однако при этом мы не должны нарушить свойство кучи. Пусть  $v_1, v_2, \dots, v_t$  — это вершины правой ветви в порядке от корня вниз (т.е.  $v_1$  — это корень). Предположим, что мы добавляем вершину  $v = (k_i, p_i)$ . Имеет место один из трёх случаев (см. рис. 6.6.2). Если  $p_i$  меньше приоритета  $v_t$ , то добавим  $v$  правым потомком вершины  $v_t$ . Если, напротив,  $p_i$  больше приоритета корня, то сделаем вершину  $v$  новым корнем, а  $v_1$  — её левым потомком. Остался случай, когда  $p_i$  меньше приоритета корня, но больше приоритета  $v_t$ . Найдём пару таких вершин  $v_j$  и  $v_{j+1}$ , что  $p_i$  меньше приоритета  $v_j$ , но больше  $v_{j+1}$ . Тогда  $v$  становится правым потомком  $v_j$ , а  $v_{j+1}$  — левым потомком  $v$ .

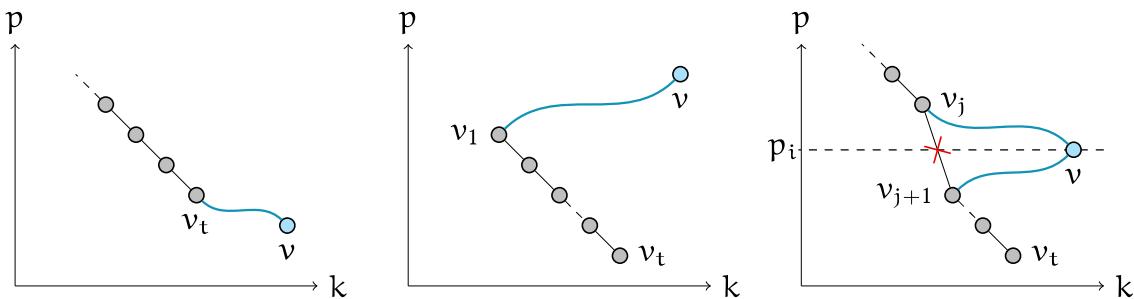


Рис. 6.6.2. Три случая добавления следующей вершины.

Оценим время работы этого алгоритма. Если при добавлении новой вершины мы будем каждый раз перебирать все вершины  $v_1, \dots, v_t$ , то в худшем случае алгоритм получится квадратичным (действительно, квадратичное время достигается, например, если на вход алгоритму подать следующий набор пар  $(1, n), (2, n - 1), \dots, (n, 1)$ ). Кажется, что мы ничего не выиграли по сравнению с первым рекурсивным алгоритмом построения. Однако, если изменить порядок и перебирать вершины  $v_1, \dots, v_t$  с конца (т.е. дополнительно хранить указатель на последнюю вершину в правой ветви и перебирать вершины от последней к корню), то алгоритм будет иметь линейное(!) время работы.

Давайте разберёмся, почему так происходит. Когда мы добавляем новую вершину  $v$  между вершинами  $v_j$  и  $v_{j+1}$  (третий случай на рис. 6.6.2), то вершины

1958  $v_{j+1}, \dots, v_t$  попадают в левое поддерево  $v$  и перестают принадлежать правой вет-  
 1959 ке. Поэтому вершины  $v_{j+1}, \dots, v_t$  никогда больше не будут встречаться при поис-  
 1960 ке места для новой вершины. Другими словами, если мы ищем подходящую пару  
 1961 вершин в правой ветке с конца, то каждое просмотренное на этой итерации реб-  
 1962 ро  $(v_k, v_{k+1})$  больше никогда не будет просмотрено — оно либо будет удалено при  
 1963 добавлении новой вершины, либо перестанет принадлежать правой ветке. При  
 1964 этом на каждой итерации мы добавляем не более одного ребра в правую ветвь.  
 1965 Значит в сумме при поиске подходящей пары вершин мы рассмотрим не более  $n$   
 1966 рёбер, а значит сложность полученного алгоритма не будет превышать  $O(n)$ .

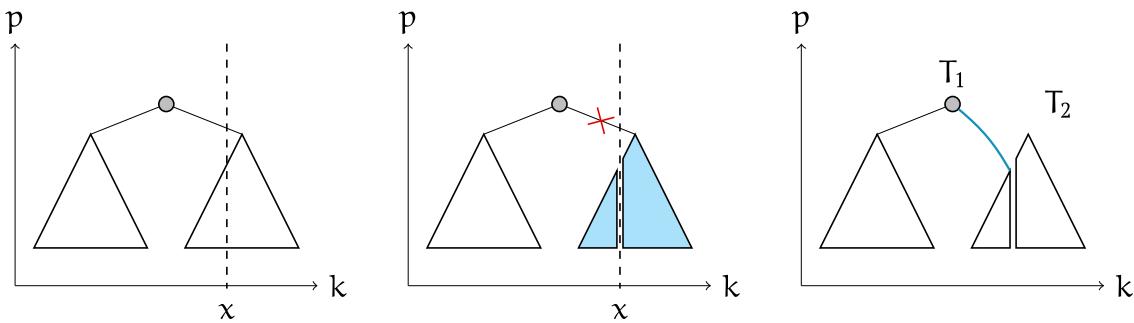
1967 *Замечание 6.6.1.* В предыдущем параграфе на самом деле описана амортизаци-  
 1968 онная оценка для операции добавления новой вершины с ключом больше, чем  
 1969 все ключи в дереве. Подобные рассуждения будут ещё неоднократно встречаться  
 1970 в курсе. Неформально можно сформулировать следующий принцип: „если на  
 1971 каждой итерации некоторого алгоритма мы увеличиваем некоторую величину  $t$  не  
 1972 более чем на единицу, и совершаём  $k \leq t$  операций, попутно уменьшая  $t$  на  $k$ , то сум-  
 1973 марное количество операций не превосходит количества итераций“. Этот нефор-  
 1974 мальный принцип можно доказать «методом ростовщика»: на каждой итерации  
 1975 мы откладываем 1 рубль, а каждая операция тратит 1 рубль, причём нельзя по-  
 1976 тратить больше, чем к этому моменту отложено. Так как каждая операция в ре-  
 1977 зультате оплачена, то количество операций не превосходит количества отложен-  
 1978 ных рублей, т.е. количества итераций.

1979 *Упражнение 6.6.1.* Докажите этот принцип при помощи метода потенциалов.

### 1980 6.6.3. Операции

1981 Осталось научиться добавлять и удалять вершины. Нам потребуются две вспо-  
 1982 могательные операции — `cartesian-split` и `cartesian-merge`.

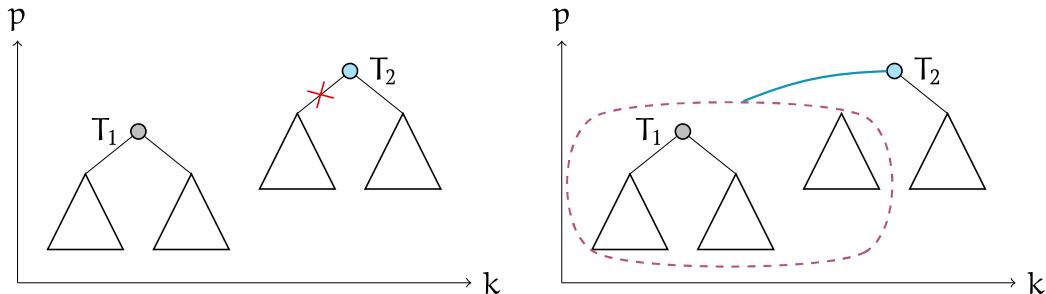
```
1983 # Разделяет дерево на два по ключу x
1984 def cartesian-split(x, v = root):
1985     if v == None:
1986         return (None, None)
1987
1988     if key(v) < x:
1989         (T1, T2) = cartesian-split(x, v.right)
1990         v.right = T1
1991         return (v, T2)
1992     else:
1993         (T1, T2) = cartesian-split(x, v.left)
1994         v.left = T2
1995         return (T1, v)
1996
1997 # Слияние двух деревьев
1998 # ключи T1 меньше ключей T2
1999 def cartesian-merge(T1, T2):
2000     if T1 == None:
2001         return T2
2002     if T2 == None:
2003         return T1
```

Рис. 6.6.3. Работа операции `cartesian-split(k)`.

```

2004 # корень правого дерева имеет больший приоритет
2005 if pri(T1) < pri(T2):
2006     T = cartesian-merge(T1, T2.left)
2007     T2.left = T
2008     return T2
2009 else
2010     T = cartesian-merge(T1.right, T2)
2011     T1.right = T
2012     return T1

```

Рис. 6.6.4. Работа операции `cartesian-merge(T1, T2)`.

На основе этих двух операций легко реализовать операции добавления и удаления вершин.

```

2015 # Добавление вершины
2016 def cartesian-insert((k,p), T):
2017     v = node(k, p)
2018     (T1, T2) = cartesian-split(k, T)
2019     T1 = cartesian-merge(T1, v)
2020     return cartesian-merge(T1, T2)
2021
2022 # Удаление всех вершин с ключом k
2023 def cartesian-remove(k, T):
2024     (T1, T2) = cartesian-split(k, T)
2025     (T2l, T2r) = cartesian-split(k+1, T2)
2026     return cartesian-merge(T1, T2r)

```

Как и операции для других деревьев, все рассмотренные операции с декартовым деревом работают за время  $O(h)$ , где  $h$  — высота дерева. Однако, как мы уже

2029 знаем, у декартова дерева с  $n$  вершинами высота может достигать  $n$ . Кроме того,  
 2030 нет никакой возможности балансировать декартово дерево, т.к. легко построить  
 2031 пример входных данных, на которых декартово дерево определено однозначно  
 2032 и имеет линейную глубину. Что же делать?

2033 **6.6.4. Дуча**

2034 **Определение 6.6.2.** *Дуча (treap, курево, дерамида)* — это дерево поиска постро-  
 2035 енное на декартовом дереве, в котором приоритеты выбираются независимо и  
 2036 случайно.

2037 Идея использовать случайные значения для приоритетов в декартовом дереве  
 2038 появилась только в 1996 году и принадлежит Сиделю и Арагону [6].

2039 **Теорема 6.6.2.** *Математическое ожидание высоты дучи ограничено  $O(\log n)$ .*

2040 **Доказательство.** Мы здесь воспользуемся теоремой о математическом ожида-  
 2041 нии глубины рекурсии для быстрой сортировки. Какая связь у декартова дерева  
 2042 с быстрой сортировкой? Давайте ещё раз посмотрим на квадратичный алгоритм  
 2043 построения декартова дерева, при помощи которого мы доказали теорему суще-  
 2044 ствования. Предположим, что на вход нам дали только ключи, а приоритеты мы  
 2045 выбираем независимо и случайно. С какой вероятностью каждая вершина будет  
 2046 выбрана корнем? Каждая вершина имеет одинаковую вероятность стать корнем,  
 2047 и это свойство сохраняется для рекурсивных вызовов: на каждом подотрезке, со-  
 2048 ответствующем рекурсивному вызову алгоритма, все вершины имеют одинако-  
 2049 вую вероятность стать корнем соответствующего поддерева. Этот процесс очень  
 2050 похож на то, что происходит в быстрой сортировке: каждая вершина имеет оди-  
 2051 наковую вероятность стать опорным элементом при первом вызове, и это свой-  
 2052 ство сохраняется для рекурсивных вызовов. Эти процессы не просто похожи, а  
 2053 верно и более сильное свойство: для любого фиксированного набора ключей су-  
 2054 ществует взаимнооднозначное соответствие между всеми возможными дучами  
 2055 и деревьями рекурсии быстрой сортировки на этом наборе. Более того, это соот-  
 2056 ветствие сохраняет вероятности (т.е. некоторая конкретная дуча будет иметь ту  
 2057 же вероятность, что и соответствующее ей дерево рекурсии быстрой сортировки).  
 2058 Таким образом оценка на математическое ожидание глубины рекурсии быстрой  
 2059 сортировки влечёт за собой такую же оценку на математическое ожидание высо-  
 2060 ты декартова дерева.  $\square$

2061 **Следствие 6.6.1.** *Операции `find`, `cartesian-insert` и `cartesian-remove` для дучи*  
 2062 *работают за  $O(\log n)$  в среднем.*

2063 **6.7. В-деревья**

2064 Другой подход к реализации деревьев поиска — это так называемые *В-деревья*,  
 2065 у которых вершины могут иметь произвольное количество потомков. В-деревья  
 2066 часто используются для поиска по данным, хранящихся во внешней памяти (на-  
 2067 пример, в файловых системах или базах данных): за счёт увеличения арности  
 2068 внутренних вершин, происходит уменьшение глубины дерева, что позволяет умень-  
 2069 шить количество запросов на чтение из внешней памяти (стоимость которых обыч-  
 2070 но значительно больше, чем стоимость операций с внутренней памятью). Хоро-  
 2071 шо известный всем пример В-дерева — это система каталогов файловой системы:

- 2072 В каждом каталоге может быть произвольное число файлов (листьев) и подкаталогов (внутренних вершин). В различных реализациях B-деревьев, элементы мо-

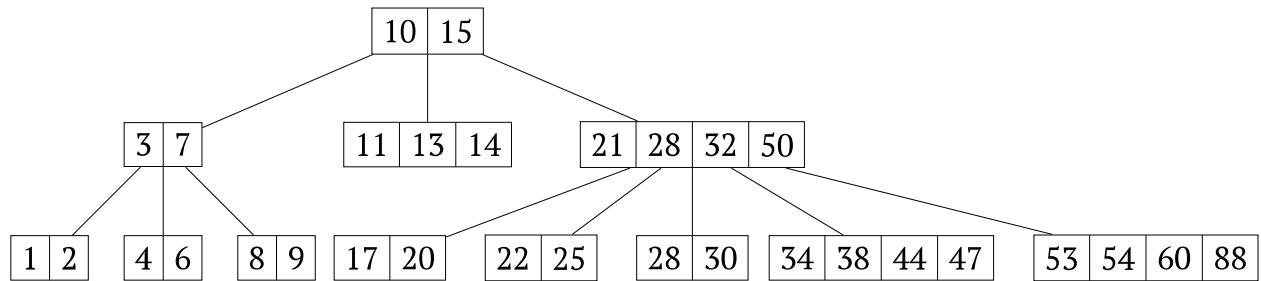


Рис. 6.7.1. B-дерево.

- 2073  
 2074 Гут храниться как во всех вершинах, так и только в листьях (тогда в вершинах хра-  
 2075 нятся ссылки на максимальные или минимальные элементы в соответствующих  
 2076 поддеревьях). «Компромиссным» вариантом между двоичными деревьями и B-  
 2077 деревьями является 2-3 дерево, реализующее идею сбалансированного B-дерева,  
 2078 у которого все внутренние вершины имеют два или три потомка.

2079 **Глава 7**

2080 **Хеширование**

2081 В этом разделе мы снова поговорим про реализации абстрактных типов дан-  
2082 ных: множество и словарь (стр. 47). Ранее мы предполагали, что на ключах за-  
2083 дан линейный порядок, и этого было достаточно, чтобы выполнять сортировку и  
2084 применять двоичный поиск или реализовывать дерево поиска. Для разговора о  
2085 хешировании нам потребуется другое предположение — мы будем предполагать,  
2086 что существует функция, сопоставляющая каждому ключу некоторое неотрица-  
2087 тельное целое число.

2088 **7.1. Прямая адресация**

2089 Начнём с ситуации, когда ключи сами по себе являются целыми числами из  
2090 отрезка  $[0, m - 1]$ . Если число  $m$  не очень большое, то мы можем хранить элемен-  
2091 ты в массиве длины  $m$ , или, другими словами, применить *таблицу с прямой ад-  
2092 дресацией* (*direct-address table*). Для реализации множества в каждой ячейке будем  
2093 хранить булево значение, которое определяет, присутствует элемент в множес-  
2094 стве или нет. Для реализации словаря в каждой ячейке будем хранить ссылку на  
2095 соответствующее значение или пустую ссылку, обозначающую отсутствие клю-  
2096 ча. При такой реализации используемая память будет пропорциональна  $m$ , а все  
2097 операции поиска, добавления и удаления будут работать за  $O(1)$ . Подобную кон-  
2098 струкцию мы уже встречали в алгоритме сортировки подсчётом — там в каждой  
2099 ячейке хранился счётчик, обозначающий сколько раз элемент встречается в мас-  
2100 сиве. Узким местом данного подхода является избыточное использование памя-  
2101 ти: например, при хранении множества 32-битных ключей, нам в любом случае  
2102 потребуется зарезервировать  $m = 2^{32}$  ячеек памяти вне зависимости от размера  
2103 множества, а для хранения 64-битных ключей данный метод и вовсе неприме-  
2104 ним.

2105 **7.2. Хеш-таблица**

2106 Основной проблемой прямой адресации было то, что общее количество ячеек  
2107  $m$  в таблице никак не зависело от количества элементов, которые в этой таблице  
2108 хранятся. Для эффективного использования памяти мы хотели бы, чтобы размер  
2109 таблицы был не более  $O(n)$ , где  $n$  — количество элементов, которые хранятся в  
2110 таблице. Но как этого достичь, если диапазон ключей очень большой или ключи  
2111 и вовсе не являются целыми числами?

### 7.2.1. Хеш-функция

Мы будем предполагать, что на множестве ключей  $K$  задана некоторая хеш-функция (*hash function*)

$$h : K \rightarrow \{0, \dots, m - 1\}.$$

Значение  $h(k)$  будем называть хеш-кодом (*hash code*) ключа  $k$ . Более того, мы будем предполагать, что мы умеем определять хеш-функции для разных значений  $m$  и таким образом влиять на количество пустых ячеек в таблице. В дальнейшем при анализе сложности мы потребуем, чтобы хеш-функции обладали некоторыми дополнительными свойствами, которые позволят получить хорошие оценки в среднем. Пока же мы будем считать, что „хорошая“ хеш-функция должна хорошо „перемешивать“ результаты, т.е., например, значение хеш-функции от двух близких по значению ключей могут быть произвольно далеки друг от друга.

### 7.2.2. Методы разрешения коллизий

Наличие хеш-функции  $h$  позволяет нам определить хеш-таблицу для ключей из множества  $K$ . Заведём массив  $H$  размера  $m$ . В этом массиве каждому ключу  $k \in K$  будет соответствовать ячейка с номером  $h(k)$ . По принципу Дирихле, если размер множества ключей больше, чем  $m$ , то обязательно найдутся два ключа, которым будет соответствовать одна и та же ячейка. Каким образом хранить два элемента в одной ячейке? Такая ситуация, при которой  $h(k_1) = h(k_2)$  при  $k_1 \neq k_2$ , называется **коллизией**. Коллизии неизбежно случаются, и поэтому хеш-таблицы обязательно имеют **метод разрешения коллизий**. Существуют множество подходов к этой проблеме, мы рассмотрим только два наиболее популярных из них.

**Метод цепочек.** *Метод цепочек (chaining)* предлагает в каждой ячейке таблицы  $H$  хранить список и записывать в него все ключи, которые попали в эту ячейку. При такой реализации метода разрешения коллизий время поиска в ячейке

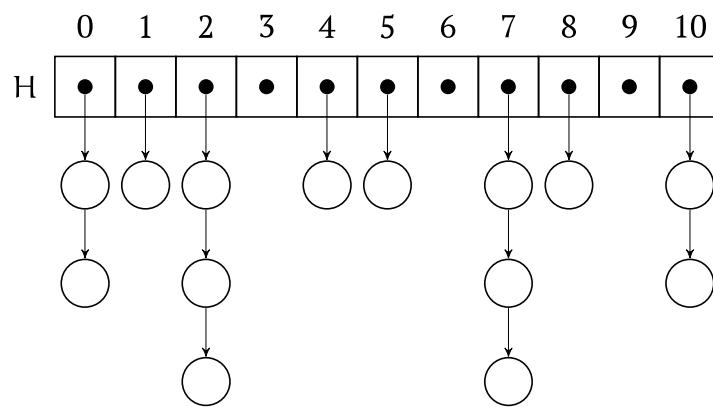


Рис. 7.2.1. Хеш-таблица с разрешения коллизий при помощи метода цепочек,  $m = 11$ .

составляется  $O(l)$ , где  $l$  — длина цепочки. Операция добавления элемента так же занимает время  $O(l)$ , если нам нужно проверить, нет ли уже этого ключа в списке, и  $O(1)$ , если в хеш-таблице разрешаются повторяющиеся ключи. Удаление занимает  $O(l)$ . Становится ясно, что в дальнейшем нас будут интересовать хеш-функции, которые имеют небольшое количество коллизий, и тем самым в

хеш-таблице будут не очень длинные цепочки. Как мы увидим дальше, этот метод асимптотически оптimalен при правильном выборе хеш-функции. Однако у него есть небольшой недостаток — для хранения списка требуется дополнительное место для хранения ссылок.

**Открытая адресация.** Для полноты картины рассмотрим общую схему методов, альтернативных методу цепочек. При *открытой адресации* в каждой ячейке может храниться только один ключ, поэтому требуется  $n \leq m$ , но каждому ключу соответствует не одна ячейка, а последовательность ячеек. Для этого выбирается хеш-функция от двух аргументов:

$$h : K \times [0, m - 1] \rightarrow [0, m - 1],$$

где второй аргумент задаёт номер в последовательности. Таким образом ключу  $k$  соответствуют ячейки с номерами

$$h(k, 0), h(k, 1), \dots, h(k, m - 1).$$

Разумно потребовать, чтобы при фиксированном  $k$  эта последовательность была бы некоторой перестановкой чисел  $\{0, \dots, m - 1\}$ . Для того, чтобы найти ключ  $k$  в хеш-таблице с открытой адресацией, нужно последовательно проверять ячейки  $h(k, 0), h(k, 1), \dots$  до тех пор, пока не встретится либо ключ  $k$ , либо пустая ячейка. Добавление ключа реализуется так же: ячейки  $h(k, 0), h(k, 1), \dots$  последовательно проверяются, пока не встретится пустая ячейка. При удалении элемента возникают трудности, т.к. удаление может нарушить цепочку, соответствующую некоторому ключу. Это можно обойти, храня в каждой ячейке дополнительный флаг, обозначающий, что значение из ячейки было удалено — таким образом можно различать пустые ячейки, и ячейки, значение из которых было удалено. Однако при такой реализации длина цепочек не уменьшается даже при удалении элементов, что отрицательно сказывается на производительности.

Такую хеш-функцию с двумя аргументами можно сконструировать из „обычной“ хеш-функции. Самая простая реализация, задающая *линейный поиск* в таблице, строит  $h$  по хеш-функции  $h_1$  следующим образом:

$$h(k, i) = (h_1(k) + i) \bmod m.$$

Однако этот способ обладает серьёзным недостатком — цепочки для близких ячеек таблицы могут „склеиваться“. В более сложном методе *двойного хеширования* функция  $h$  строится на основе двух „обычных“ хеш-функций  $h_1, h_2$  следующим образом:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

Для этого метода цепочки различных ключей будут „разбросаны“ по таблице с разным шагом, и поэтому не будут склеиваться. Нужно только как-то обеспечить, чтобы для фиксированного ключа элементы последовательности не повторялись. Для этого удобно выбрать  $m$  простым, и тогда нам подойдёт любая хеш-функция  $h_2(k) : K \rightarrow [1, m - 1]$ .

### 7.2.3. Примеры хеш-функций

В общем случае для того, чтобы выбрать хорошую хеш-функцию, нам нужно что-то знать про распределение ключей. В этом разделе мы рассмотрим некоторые конструкции, которые могут давать неплохие результаты.

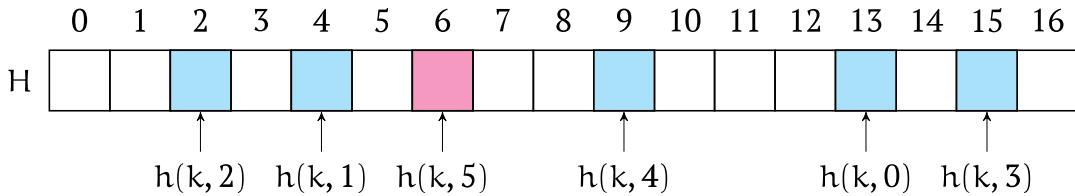


Рис. 7.2.2. Просмотр ячеек при открытой адресации,  $m = 17$ .

- **Равномерно распределённые вещественные числа.** Будем считать, что вещественные числа лежат в полуинтервале  $[0, 1)$ . Если это не так, то воспользуемся масштабированием. Тогда в качестве хеш-функции можно использовать

$$h(k) = \lfloor mk \rfloor.$$

Такая хеш-функция просто разбивает  $[0, 1)$  на  $m$  полуинтервалов одинаковой длины и сопоставляет каждому числу номер интервала, в который оно попадает.

- **Целые числа.** Самый простой вид хеш-функция для целых чисел даёт метод *деления с остатком*:

$$h(k) = k \bmod m.$$

Более сложный *мультипликативный метод* обладает значительно лучшими перемешивающими свойствами:

$$h(k) = \lfloor m \cdot \{ck\} \rfloor,$$

где  $\{\cdot\}$  обозначает дробную часть числа, а  $c$  — некоторая вещественная константа. Для лучшего результата константу  $c$  нужно выбирать иррациональной, например,  $c = \sqrt{2}$ , тогда дробная часть  $ck$  не равна нулю при любом  $k$  (в памяти компьютера можно задать только некоторое рациональное приближение иррационального числа, но и этого будет достаточно). Этот метод можно обобщить и на пару значений, выбрав две различные константы  $c_1$  и  $c_2$  и определив

$$h(k_1, k_2) = \lfloor m \cdot \{c_1k_1 + c_2k_2\} \rfloor.$$

(константы  $c_1$  и  $c_2$  лучше выбирать линейно независимыми над  $\mathbb{Q}$  иррациональными числами, например,  $\sqrt{2}$  и  $\sqrt{3}$ .)

- **Последовательности целых чисел, строки.** Для вычисления хеш-кода последовательности целых чисел  $a = (a_0, a_1, \dots, a_{n-1})$  хорошие результаты может дать *полиномиальный метод*: выберем некоторое взаимнопростое с  $m$  число  $x$ , называемое *основанием*, и определим

$$h(a) = (a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{n-1} \cdot x^{n-1}) \bmod m.$$

- **Кортежи разнотипных значений.** Довольно частая задача, которая возникает при написании программ, — это придумать хеш-функцию для некоторой структуры, поля которой имеют разные типы. В этом случае разумно сначала вычислить хеш-коды для каждого из полей, а потом скомбинировать их при помощи мультипликативного или полиномиального методов.

2219 Замечание 7.2.1. Вы могли также встречать или даже применять для каких-то за-  
 2220 дач *криптографические хеш-функции*, например, такие как MD5 или SHA-1. Как  
 2221 следует из названия, такие хеш-функции предназначены для криптографических  
 2222 протоколов и не используются в хеш-таблицах. Во-первых потому, что они име-  
 2223 ют очень большое пространство хеш-кодов, значительно больше, чем нужно для  
 2224 хеш-таблиц (128 битов для MD5 и 160 битов для SHA-1). А во-вторых, так как на  
 2225 криптографические хеш-функции накладываются более сложные требования, то  
 2226 они довольно сложно устроены, и как следствие этого, долго вычисляются (зна-  
 2227 чительно дольше, чем рассмотренные выше конструкции). С другой стороны, в  
 2228 некотором приближении можно считать, что схема хранения файлов в репозито-  
 2229 рии системы контроля версий Git — это хеш-таблица, основанная на криптогра-  
 2230 фической хеш-функции SHA-256 (если говорить точнее, то там вместо таблицы  
 2231 используется дерево каталогов файловой системы, т.е. В-дерево).

2232 В дальнейшем в этом разделе мы для простоты будем предполагать, что слож-  
 2233 ность вычисления хеш-функции равна  $O(1)$ . Это верно для чисел, но неверно, на-  
 2234 пример, для строк и последовательностей. Поэтому в случаях, когда вычисление  
 2235 хеш-функции требует неконстантного времени, в оценке нужно будет дополни-  
 2236 тельно учесть, сколько раз тот или иной алгоритм вычисляет хеш-функцию.

#### 2237 7.2.4. Детали реализации

2238 При реализации хеш-таблицы часто заранее неизвестно, сколько ключей в  
 2239 ней будет храниться. Поэтому разумно начинать с некоторой небольшой кон-  
 2240 стантны, а потом расширять таблицу аналогично тому, как мы делали это для рас-  
 2241 ширяющегося массива: если коэффициент заполнения  $\alpha = n/m$  стал больше неко-  
 2242 торого порога, то нужно создать новую таблицу размера  $\gamma m$  для некоторой кон-  
 2243 стантны  $\gamma > 1$ , выбрать новую хеш-функцию  $K \rightarrow \{0, \dots, \gamma m - 1\}$  и добавить в  
 2244 неё все ключи из старой. При таком подходе будут верны оценки, полученные  
 2245 нами для расширяющегося массива, т.е. на расширение таблицы за всё время ра-  
 2246 боты мы дополнительно потратим не более  $O(n)$  копирований и  $O(n)$  вычисле-  
 2247 ний хеш-функции. Увеличивать таблицу можно не только при слишком большом  
 2248 значении  $\alpha$ , но и если длина самой длинной цепочки стала больше некоторого  
 2249 порога.

### 2250 7.3. Гипотеза равномерного хеширования

2251 Для удобства изложения введём обозначение для коэффициента заполнения  
 2252 хеш-таблицы  $\alpha = \frac{n}{m}$ . Как мы увидели ранее, сложность поиска ключа  $k$  в хеш-  
 2253 таблице пропорциональна длине цепочки, соответствующей  $k$ . Поэтому желательным  
 2254 свойством хеш-функции является „равномерность“ распределения хеш-  
 2255 кодов — тогда в среднем длина цепочки будет равна коэффициенту заполнения,  
 2256 и, соответственно, сложность поиска в среднем будет ограничена  $O(1 + \alpha)$ . Од-  
 2257 нако на самом деле равномерность распределения хеш-кодов зависит не только  
 2258 от функции, но и от распределения ключей. Для любой хеш-функции можно по-  
 2259 добрать такое распределение на ключах, для которого распределение хеш-кодов  
 2260 будет очень далеко от равномерного: выберем некоторое  $j \in \{0, \dots, m - 1\}$  и рас-  
 2261 смотрим равномерное распределение на ключах с хеш-кодом  $j$ . Рассуждая в тер-  
 2262 минах злонамеренного противника мы могли бы сказать, что противник всегда  
 2263 может заставить хеш-таблицу работать медленно — для этого ему нужно записы-

2264 вать в хеш-таблицу только ключи с хеш-кодом  $j$  для некоторого фиксированного  
 2265  $j$ .

2266 На практике же иногда у нас могут быть причины предполагать, что на име-  
 2267 ющемся распределении ключей выбранная нами хеш-функция задаёт распреде-  
 2268 ление хеш-кодов, близкое к равномерному. Давайте оценим сложность операций  
 2269 с хеш-таблицей в этом предположении. Для его формализации мы будем поль-  
 2270 зоваться следующей гипотезой, которая избавит нас от необходимости делать  
 2271 какие-то предположения про распределение ключей.

2272 *Предположение 7.3.1* (Гипотеза равномерного хеширования). Значение хеш-фун-  
 2273 кции  $h : K \rightarrow \{0, \dots, m - 1\}$  от ключа  $k$  является случайной величиной, равномерно  
 2274 распределённой на множестве  $\{0, \dots, m - 1\}$ . При этом значения хеш-функции на  
 2275 различных ключах  $k_1$  и  $k_2$  независимы.

2276 *Замечание 7.3.1.* На практике значение хеш-функции не является случайной ве-  
 2277 личиной, т.к. хеш-функции не используют случайных битов. Тем не менее, мы  
 2278 могли бы попробовать запрограммировать хеш-функцию, удовлетворяющую этой  
 2279 гипотезе. Такая функция могла бы быть устроена следующим образом: каждый  
 2280 раз, когда хеш-функция вычисляется от нового ключа  $k$ , значением  $h(k)$  выби-  
 2281 рается случайный элемент множества  $\{0, \dots, m - 1\}$ , и эта информация запоми-  
 2282 нается. Если такую функцию реализовать на практике, то возникнет проблема с  
 2283 тем, что для эффективного хранения информации о выбранных значениях нам  
 2284 потребуется хеш-таблица.

2285 **Теорема 7.3.1.** В предположении гипотезы равномерного хеширования среднее вре-  
 2286 мя безуспешного поиска при использовании метода цепочек равно  $\Theta(1 + \alpha)$ .

2287 *Доказательство.* В процессе безуспешного поиска ключа  $k$  мы должны полно-  
 2288 стью просмотреть цепочку в ячейке  $h(k)$ . Для каждого  $i \in \{0, \dots, m - 1\}$  обозна-  
 2289 чим длину цепочки в ячейке  $i$  как  $l_i$ . По гипотезе равномерного хеширования  
 2290 значение  $h(k)$  с равной вероятностью равно каждому из индексов  $\{0, \dots, m - 1\}$ .  
 2291 Следовательно, математическое ожидание длины цепочки в ячейке  $h(k)$  равно

$$2292 \mathbb{E}[l_{h(k)}] = \sum_{i=0}^{m-1} \frac{1}{m} \cdot l_i = \frac{n}{m} = \alpha.$$

2293 Отсюда среднее время поиска равно  $\Theta(1 + \alpha)$  (единица возникает, т.к. мы всегда  
 2294 сделаем хотя бы одну операцию, даже если цепочка пустая.)  $\square$

2295 В предыдущей теореме нам достаточно было применить гипотезу равномер-  
 2296 ного хеширования только для ключа, который мы ищем. В следующей теореме  
 2297 нам нужно будет применить эту гипотезу ко всем ключам в таблице.

2298 **Теорема 7.3.2.** В предположении гипотезы равномерного хеширования среднее вре-  
 2299 мя успешного поиска при использовании метода цепочек равно  $\Theta(1 + \alpha)$ .

2300 *Замечание 7.3.2.* В данной теореме идёт речь о поиске ключа, который хранит-  
 2301 ся в таблице, поэтому среднее время вычисляется не только по вероятностному  
 2302 пространству, соответствующему хеш-функции, но и по множеству ключей, на-  
 2303 ходящихся в таблице.

2304 *Доказательство.* Пусть в хеш-таблице хранятся ключи  $k_1, k_2, \dots, k_n$ , и пусть ну-  
 2305 мерация ключей соответствует порядку, в котором ключи добавлялись в хеш-  
 2306 таблицу. Для каждой пары  $i, j \in [n]$  определим случайную величину  $X_{i,j}$ , равную 1,

2307 если случилась коллизия  $h(k_i) = h(k_j)$ , и равную 0 в противном случае. Заметим,  
 2308 что  $E[X_{i,i}] = 1$ , и по гипотезе равномерного хеширования  $E[X_{i,j}] = \frac{1}{m}$  для  $i \neq j$   
 2309 (здесь мы пользуемся независимостью  $h(k_i)$  и  $h(k_j)$ ).

2310 Предположим теперь, что мы ищем ключ  $k_i$ . Сколько элементов нам нужно  
 2311 просмотреть для поиска  $k_i$  в среднем? Будем считать, что при добавлении эле-  
 2312 мента в ячейку он добавляется в начало списка. Тогда количество элементов, ко-  
 2313 торые мы просмотрим при поиске  $k_i$ , равно  $X_{i,i} + \dots + X_{i,n}$ . Тогда среднее время  
 2314 поиска ключа в таблице (среднее по всем ключам присутствующим в таблице и  
 2315 по распределению случайных величин  $\{X_{i,j}\}$ ) равно

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n \mathbb{E} \left[ \sum_{j=i}^n X_{i,j} \right] &= \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \mathbb{E}[X_{i,j}] \right) \\ &= \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\ &= 1 + \frac{1}{mn} \sum_{i=1}^n (n - i) \\ &= 1 + \frac{1}{mn} \left( n^2 - \frac{n(n+1)}{2} \right) \\ &= \Theta(1 + \alpha). \end{aligned}$$

2316

□

## 2318 7.4. Универсальное хеширование

2319 Гипотеза равномерного хеширования позволяет объяснить, почему при «хо-  
 2320 рошем» (т.е., равномерном) распределении хеш-кодов поиск в хеш-таблице бу-  
 2321 дет работать быстро  $O(1+\alpha)$ . Однако в общем случае не понятно, будет ли распре-  
 2322 деление хеш-кодов для конкретной хеш-функции действительно равномерным,  
 2323 т.к. не известно распределение на ключах. Эту проблему можно решить, если рас-  
 2324 сматривать не одну хеш-функцию, а целое семейство хеш-функций с дополни-  
 2325 тельным свойством.

2326 **Определение 7.4.1.** Множество функций  $\mathcal{H}$  из множества ключей  $K$  в  $\{0, \dots, m-1\}$   
 2327 называется *универсальным семейством хеш-функций (УСХФ)*, если для любой пары  
 2328 различных ключей  $k_1$  и  $k_2$

$$2329 \quad \Pr_{h \leftarrow \mathcal{H}} [h(k_1) = h(k_2)] \leq \frac{1}{m}.$$

2330 *Замечание 7.4.1.* Отметим, что аналогичный факт верен в предположении гипо-  
 2331 тезы равномерного хеширования, но вероятность определяется по другому ве-  
 2332 роятностному пространству. Если в случае гипотезы равномерного хеширования  
 2333 значение хеш-функции на разных ключах выбираются случайно и независимо,  
 2334 то в случае универсального семейства хеш-функций случайным является только  
 2335 выбор хеш-функции, и она в свою очередь однозначно определяет хеш-коды для  
 2336 всех ключей.

2337 При реализации хеш-таблицы, основанной на универсальном семействе хеш-  
 2338 функций, перед началом заполнения таблицы мы выбираем случайную функ-  
 2339 цию из универсального семейства и используем её для вычисления хеш-кодов.

2340 За счёт этого обеспечивается хорошая в среднем производительность вне зависи-  
 2341 мости от распределения ключей, т.к. теперь усреднение происходит не только  
 2342 по распределению на ключах, но и по выбору хеш-функции из семейства. Дру-  
 2343 гими словами, теперь противнику сложно подобрать распределение, на котором  
 2344 хеш-функция будет иметь много коллизий — он может это сделать для каждой  
 2345 хеш-функции из семейства, но он не знает заранее, какую функцию мы выберем.

2346 **Теорема 7.4.1.** При использовании универсального хеширования среднее время без-  
 2347 успешного поиска ключа равно  $\Theta(1 + \alpha)$ .

2348 **Доказательство.** Пусть в хеш-таблице хранятся ключи  $k_1, k_2, \dots, k_n$ . В процес-  
 2349 се безуспешного поиска ключа  $k$  мы должны полностью просмотреть цепочку в  
 2350 ячейке  $h(k)$ . Для каждого  $i \in [n]$  определим случайную величину  $X_i$ , равную 1,  
 2351 если случилась коллизия  $h(k_i) = h(k)$ , и равную 0 в противном случае. Из опреде-  
 2352 ления универсального семейства следует, что  $\mathbb{E}[X_i] \leq \frac{1}{m}$ . Тогда математическое  
 2353 ожидание длины цепочки в ячейке  $h(k)$  равно

$$2354 \quad \mathbb{E} \left[ \sum_{i=1}^n X_i \right] = \sum_{i=1}^n \mathbb{E}[X_i] \leq \frac{n}{m} = \alpha.$$

2355 Поэтому среднее время поиска равно  $\Theta(1 + \alpha)$  (единица возникает, т.к. мы всегда  
 2356 сделаем хотя бы одну операцию.)  $\square$

2357 **Теорема 7.4.2.** При использовании универсального хеширования среднее среднее вре-  
 2358 мя успешного поиска ключа равно  $\Theta(1 + \alpha)$ .

2359 **Доказательство.** Доказательство полностью повторяет доказательство предыду-  
 2360 щей теоремы за исключением того, что  $k$  теперь присутствует в таблице. Поэтому  
 2361 один из членов в получившейся сумме будет равен 1, и мы получим

$$2362 \quad \sum_{i=1}^n \mathbb{E}[X_i] \leq 1 + \frac{n-1}{m} \leq 1 + \alpha.$$

2363  $\square$

2364 Получается, что используя универсальное семейство хеш-функций мы с хоро-  
 2365 шей вероятностью можем обеспечить эффективность хеш-таблицы. Однако пока  
 2366 не ясно, существуют ли такие семейства и как легко их построить. Оказывается,  
 2367 что такие семейства есть и построить их не сложно. Мы покажем, как построить  
 2368 такое семейство для целых чисел.

**Теорема 7.4.3.** Пусть  $K = \{0, \dots, n\}$ . Выберем простое  $p > n$ . Тогда семейство хеш-  
 функций  $\mathcal{H}$ , состоящее из функций

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

2369 для всех  $a \in \{1, \dots, p-1\}$  и  $b \in \{0, \dots, p-1\}$ , является универсальным.

2370 **Доказательство.** Рассмотрим пару ключей  $k_1 \neq k_2$  и посмотрим, как на них дей-  
 2371 ствует «внутреннее» линейное преобразование по модулю  $p$ . Положим

$$2372 \quad t_1 = (ak_1 + b) \bmod p, \quad \text{и} \quad t_2 = (ak_2 + b) \bmod p.$$

2373 Т.к. ключи  $k_1$  и  $k_2$  не превосходят  $p$ , то  $t_1 \neq t_2$ . Действительно, если предположить,  
2374 что  $t_1 = t_2$ , то из соотношения

$$2375 \quad ak_1 + b = ak_2 + b \pmod{p}$$

2376 получается, что

$$2377 \quad a(k_1 - k_2) = 0 \pmod{p}.$$

2378 Произведение двух чисел меньших  $p$  (тут важно, что  $p$  простое) делится на  $p$  тогда  
2379 и только тогда, когда одно из них равно нулю. Мы выбираем  $a \neq 0$ , следовательно  
2380  $k_1 - k_2 = 0$ , а такого быть не может, т.к. мы рассматриваем различные  $k_1$  и  $k_2$ .

2381 Теперь покажем, что при фиксированных  $k_1$  и  $k_2$  различные пары  $(a, b)$  приво-  
2382 дят к разным парам  $(t_1, t_2)$ . Действительно, при заданных  $k_1$  и  $k_2$  по  $(a, b)$  можно  
2383 вычислить  $(t_1, t_2)$ , а при заданных  $(t_1, t_2)$  можно однозначно восстановить  $(a, b)$ :  
2384 из соотношений

$$2385 \quad \begin{cases} t_1 = (ak_1 + b) \pmod{p}, \\ t_2 = (ak_2 + b) \pmod{p}, \end{cases}$$

2386 получаем  $a = (t_1 - t_2) \cdot (k_1 - k_2)^{-1} \pmod{p}$  (тут обратное берётся по модулю  $p$ ), а  
2387 затем, уже зная  $a$ , вычисляем  $b = (t_1 - ak_1) \pmod{p}$ . Получается, что между парами  
2388  $(a, b)$  и  $(t_1, t_2)$ ,  $t_1 \neq t_2$  есть взаимнооднозначное соответствие. При всех возмож-  
2389 ных  $p(p-1)$  парах  $(a, b)$  каждая такая пара  $(t_1, t_2)$  встречается равно один раз.  
2390 Т.е. если выбирать функцию  $h_{a,b}$  случайно и равномерно из  $\mathcal{H}$ , то распределе-  
2391 ние соответствующих пар  $(t_1, t_2)$  тоже будет случайно и равномерно на всех парах  
2392  $t_1, t_2 \in \{0, p-1\}$ ,  $t_1 \neq t_2$ .

2393 Таким образом вероятность того, что  $h_{a,b}(k_1) = h_{a,b}(k_2)$  равна вероятности  
2394 того, что случайные различные  $t_1, t_2 \in \{0, p-1\}$  окажутся равными по модулю  
2395  $m$ . Для фиксированного  $t_1$  количество  $t_2 \neq t_1$ , которые дают тот же остаток по  
2396 модулю  $m$ , не превосходит

$$2397 \quad \left\lceil \frac{p}{m} \right\rceil - 1 \leqslant \frac{p+m-1}{m} - 1 = \frac{p-1}{m}.$$

2398 Каждое  $t_2$  встречается с вероятностью  $1/p$ , следовательно, вероятность получить  
2399 коллизию равна

$$2400 \quad \frac{1}{p} \cdot \frac{p-1}{m} \leqslant \frac{1}{m},$$

2401 что и требовалось доказать. □

2402 Используя универсальное хеширование для целых чисел можно построить уни-  
2403 версальное семейство хеш-функций для целочисленных последовательностей и  
2404 строк, основанное на идее полиномиального хеширования. Пусть все элементы  
2405 последовательности являются целыми числами из  $\{0, \dots, d-1\}$ . Выберем простое  
2406  $p \geqslant \max(d, m)$  и построим универсальное семейство хеш-функций вида  $\{0, \dots, p-1\} \rightarrow \{0, \dots, m-1\}$ , обозначив его  $\mathcal{H}_p$ . Тогда семейство хеш-функций, состоящее  
2407 из всех функций вида:

$$2409 \quad h_x((a_0, a_1, \dots, a_{n-1})) = h((a_0x_0 + a_1x_1 + \dots + a_{n-1}x^{n-1}) \pmod{p}),$$

2410 где  $x \in \{1, \dots, p-1\}$  и  $h \in \mathcal{H}_p$ , является универсальным.

## 7.5. Совершенное хеширование

В этом разделе мы покажем, что если в хеш-таблице хранится неизменяемое множество элементов (т.е. мы имеем дело со статической задачей поиска), то можно эффективным образом добиться отсутствия коллизий, и тогда сложность доступа будет  $O(1)$  в худшем. Такая конструкция называется *совершенным хешированием*. Эту конструкцию можно применять даже в том случае, если множество ключей в таблице изменяется, но множество всех возможных ключей достаточно мало (тогда размер таблицы будет пропорционален мощности множества всех возможных ключей).

Пусть размер множества ключей  $K$ , которые мы хотим хранить в таблице, равен  $n$ . Структура совершенной хеш-таблицы состоит из двух уровней (см. рис. 7.5.1). Первый уровень образован хеш-таблицей  $H$  размера  $n$  с хеш-функцией  $h$ . В каждой ячейке этой таблицы хранятся ссылка на хеш-таблицу второго уровня и описание хеш-функции для неё, для ячейки с номером  $i$  будем обозначать их  $T_i$  и  $g_i$  соответственно. Таким образом, если  $h(k) = i$ , то ключ  $k$  хранится в хеш-таблице  $T_i$  в ячейке с номером  $g_i(k)$ . Осталось определить размер хеш-таблиц второго уровня. Для каждого  $i \in \{0, \dots, n-1\}$  через  $n_i$  обозначим количество ключей, для которых  $h(k) = i$ , тогда размер  $T_i$  равен  $n_i^2$ .

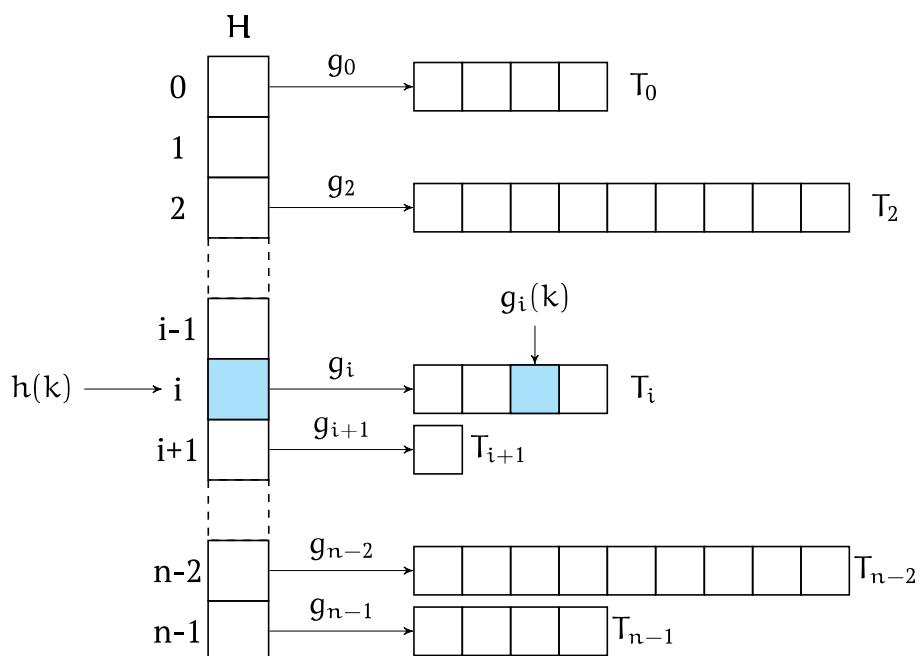


Рис. 7.5.1. Схема совершенного хеширования. Некоторые ячейки хеш-таблицы  $H$  могут оказаться пустыми.

Утверждается, что для фиксированного множества ключей при использовании универсального хеширования можно выбрать функции  $h, g_0, g_1, \dots, g_{n-1}$  таким образом, чтобы не было коллизий (т.е. в каждой ячейке хеш-таблиц второго уровня хранилось не более одного ключа) и при этом суммарный размер таблиц не превышал  $O(n)$ . Начнём с того, что покажем, как выбрать хеш-функции без коллизий хеш-таблиц второго уровня.

**Теорема 7.5.1.** При использовании универсального хеширования для хеш-таблицы размера  $m = n^2$  вероятность возникновения коллизий не превышает  $1/2$ .

2437 *Доказательство.* Если всего  $n$  ключей, то различных неупорядоченных пар  $\binom{n}{2}$ .  
 2438 Вероятность коллизии для каждой пары по определению универсального семейства хеш-функций не превосходит  $1/m$ . Пусть случайная величина  $X$  равна количеству коллизий. Тогда её математическое ожидание

$$2441 \quad \mathbb{E}[X] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

2442 Теперь нужно воспользоваться неравенством Маркова (теорема ??):

$$2443 \quad \Pr[X \geq 1] \leq \frac{\mathbb{E}[X]}{1} < \frac{1}{2}.$$

□

2445 Эта теорема показывает, что как минимум половина хеш-функций из универсального семейства не будут иметь коллизий для заданного множества ключей, если размер таблицы равен квадрату количества элементов. Поэтому в среднем для каждой таблицы второго уровня нам нужно проверить две случайные хеш-функции из универсального семейства, чтобы найти такую, которая не имеет коллизий. Остаётся показать, что можно выбрать такую  $h$ , что суммарный размер таблиц второго уровня будет ограничен  $O(n)$ . Вспомним, что  $n_i$  — это количество ключей, которые попадают в ячейку  $i$  таблицы  $H$ . Тогда суммарный размер хеш-таблиц второго уровня равен  $\sum_{i=0}^{n-1} n_i^2$ .

2454 **Теорема 7.5.2.** *При использовании универсального хеширования для хеш-таблицы размера  $m = n$*

$$2456 \quad \Pr\left[\sum_{i=0}^{n-1} n_i^2 \geq 4n\right] \leq 1/2.$$

2457 *Доказательство.* Сначала покажем, что

$$2458 \quad \mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] < 2n.$$

2459 Для этого воспользуемся соотношением  $n_i^2 = n_i + 2\binom{n_i}{2}$ :

$$2460 \quad \mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] = \mathbb{E}\left[\sum_{i=0}^{n-1} n_i + 2 \sum_{i=0}^{n-1} \binom{n_i}{2}\right] = n + 2 \mathbb{E}\left[\sum_{i=0}^{n-1} \binom{n_i}{2}\right].$$

2461 Заметим, что  $\sum_{i=0}^{n-1} \binom{n_i}{2}$  — это общее число пар ключей, которые дают коллизию. Поэтому математическое ожидание этой величины можно оценить аналогично предыдущей теореме: для каждой пары ключей вероятность коллизии не превосходит  $1/m = 1/n$ , следовательно математическое ожидание общего числа коллизий не превосходит  $\binom{n}{2} \cdot \frac{1}{n}$ . Таким образом получаем, что

$$2466 \quad \mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] \leq n + 2\binom{n}{2} \frac{1}{n} = n + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n} = n + n - 1 < 2n.$$

2467 Остаётся применить неравенство Маркова (теорема ??):

$$2468 \quad \Pr\left[\sum_{i=0}^{n-1} n_i^2 \geq 4n\right] \leq \frac{\mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}.$$

□

2470       Теперь можно описать (вероятностный) алгоритм построения таблицы для со-  
2471       вершенного хеширования. Пусть нам дано множество  $n$  ключей. Сначала построим  
2472       хеш-таблицу  $H$  и выберем хеш-функцию  $h : K \rightarrow \{0, \dots, n - 1\}$  так, чтобы  
2473       сумма квадратов количества элементов в ячейках  $H$  не превосходила  $4n$ . По тео-  
2474       реме 7.5.2 для этого нам в среднем потребуется проверить две случайные хеш-  
2475       функции из универсального семейства. Когда такая  $h$  найдена, то для каждой  
2476       ячейки  $i$  хеш-таблицы  $H$  мы заводим хеш-таблицу второго уровня  $T_i$  размера  $n_i^2$   
2477       и выбираем для неё хеш-функцию  $g_i : K \rightarrow \{0, \dots, n_i^2 - 1\}$  так, чтобы она не име-  
2478       ла коллизий на множестве ключей, для которых  $h(k) = i$ . По теореме 7.5.1 для  
2479       этого нам тоже в среднем потребуется проверить две случайные хеш-функции  
2480       из универсального семейства. В результате, в среднем предложенный алгоритм  
2481       построения работает за  $O(n)$ .

## 2482 Глава 8

# 2483 Числовые алгоритмы

### 2484 8.1. Арифметические операции с $n$ -битовыми числами

2485 В данном разделе мы обсудим алгоритмы для выполнения арифметических  
2486 операций с целыми числами произвольной длины. Числа будут задаваться их  
2487 представлением в двоичной системе счисления, т.е. можно считать, что число  
2488 задаётся битовым массивом его двоичных цифр. Числа в таком представлении  
2489 легко умножать и делить на два, для этого нужно, соответственно, добавить ноль  
2490 в конец массива или удалить последний элемент массива. Ещё проще вычислить  
2491 остаток при делении на два — остаток записан в последнем элементе массива.  
2492 Для сравнения двух таких чисел нужно сначала сравнить их длины (игнорируя  
2493 ведущие нули, если по какой-то причине они присутствуют в массиве), а если  
2494 длины оказались равны, то просто сравнить эти два массива как строки.

2495 Сложение и вычитание  $n$ -битовых целых легко реализовать за  $O(n)$  используя  
2496 школьный метод сложения в столбик. Умножение мы тоже можем реализовать  
2497 в столбик за  $O(n^2)$ , но значительно для битового представления проще реали-  
2498 зовать умножение в виде следующего рекурсивного алгоритма. В качестве идеи  
2499 алгоритма возьмём следующее соотношение:

$$2500 a \cdot b = \begin{cases} 2(a \cdot \lfloor b/2 \rfloor), & b \text{ — чётное,} \\ 2(a \cdot \lfloor b/2 \rfloor) + a, & b \text{ — нечётное.} \end{cases}$$

```
2501 # Умножение двух битовых чисел
2502 def multiply(a, b):
2503     if b == 0:
2504         return 0
2505
2506     # рекурсивно вычисляем  $2(a \cdot \lfloor b/2 \rfloor)$ 
2507     c = 2 * multiply(a, b // 2)
2508
2509     # обработка нечётного b
2510     if b % 2 == 1:
2511         c = c + a
2512
2513 return c
```

2514 Проанализируем сложность этого алгоритма, если длина  $a$  составляет  $n$  битов,  
2515 длина  $b = m$  битов, и  $n \geq m$ . В этом коде проверка на равенство нулю, умноже-  
2516 ние и деление на два, а также взятие остатка по модулю два, можно реализовать за

2517  $O(1)$ , пользуясь тем, что числа заданы битовыми массивами. Остается только сложение  $c$  и  $a$ , которое занимает время пропорциональное сумме длин  $|c| + |a| \leq 3n$ .  
 2519 Таким образом каждый вызов `multiply` без учёта вложенных рекурсивных вызовов выполняется за  $O(n)$ . Глубина рекурсии равна количеству бит в записи  $b$ , поэтому в сумме получается  $O(n \cdot m)$ .

2522 Теперь поговорим о делении. Запрограммировать «школьный» алгоритм деления в столбик довольно сложно (попробуйте сами!). Значительно проще реализовать следующий рекурсивный для деления битовых чисел с остатком (идея алгоритма аналогична идеей алгоритма умножения, рассмотренного выше).

```

2526 # Деление двух битовых чисел с остатком
2527 # Возвращает пару (частное, остаток)
2528 def divide(a, b):
2529     # база рекурсивного алгоритма
2530     if a < b:
2531         return (0, a)
2532
2533     # рекурсивное вычисляем  $\lfloor a/2 \rfloor / b$  и  $\lfloor a/2 \rfloor \bmod 2$ 
2534     (q, r) = divide(a // 2, b)
2535
2536     # умножаем результаты на 2
2537     q = q * 2
2538     r = r * 2
2539
2540     # если a был нечётным, то к остатку ещё нужно добавить 1
2541     if a % 2 == 1:
2542         r = r + 1
2543
2544     # исправляем результат, если остаток стал больше делителя
2545     if r > b:
2546         q = q + 1
2547         r = r - b
2548
2549     return (q, r)

```

2550 Оценим сложность этого алгоритма для  $n$ -битового  $a$  и  $m$ -битового  $b$ , где  $n \geq m$ .  
 2551 Умножение на два, деление на два, а также взятие остатка по модулю два, реализуются за  $O(1)$ . Сравнение  $a$  и  $b$ , сложение и вычитание можно реализовать за  $O(n + m) = O(n)$ , т.е. каждый вызов `divide` без учёта рекурсивных вызовов выполняется за  $O(n)$ . Общее количество рекурсивных не более в  $n - m$ , в результате получается  $O(n \cdot (n - m + 1))$  (если  $n = m$ , то алгоритм выполнится за  $O(n)$ ).

2556 Для наших целей будет полезно научиться также вычислять *наибольший общий делитель* (*НОД*, *greatest common divisor*, *GCD*) двух чисел. Для этой задачи существует широко известный алгоритм Евклида, который появился задолго по появления вычислительных машин. Идея алгоритма Евклида содержится в следующем соотношении:

$$2561 \quad \gcd(a, b) = \gcd(b, a \bmod b).$$

2562 Действительно, пусть  $a = bq + r$ , где  $r = a \bmod b$ . Если какое-то число делит  $b$  и  $r$ ,  
 2563 то оно обязательно делит  $a$ . Если какое-то число делит  $a$  и  $b$ , то оно также делит  
 2564 и  $r = a - bq$ . Таким образом множество общих делителей пары  $(a, b)$  совпадает с

2565 множеством общих делителей ( $b, r$ ), а следовательно наибольший делитель у этих  
 2566 пар тоже общий. Таким образом можно использовать следующий алгоритм для  
 2567 вычисления наибольшего общего делителя двух чисел:

```
2568 # алгоритм Евклида
2569 # вычисление наибольшего общего делителя
2570 def gcd(a,b):
2571     if b == 0:
2572         return a
2573
2574     return gcd(b, a % b)
```

2575 Однако для дальнейших целей на будет полезно научиться также находить так  
 2576 называемые *коэффициенты Безу*.

2577 **Теорема 8.1.1** (Теорема Безу). Для любых целых  $a$  и  $b$  из того, что  $\gcd(a, b) = d$ ,  
 2578 следует, что существуют такие целые  $x$  и  $y$ , что

$$2579 \quad ax + by = d.$$

2580 Такие коэффициенты  $x$  и  $y$  называются *коэффициентами Безу*. Для нахожде-  
 2581 ния коэффициентов Безу используется *расширенный алгоритм Евклида*.

2582 *Замечание 8.1.1.* Уравнения в целых числах называются *Диофантовыми уравне-  
 2583 ниями*. В теореме Безу мы имеем дело с линейным Диофантовым уравнением от  
 2584 двух переменных  $ax + by = d$ . Можно показать, что у такого уравнения будет бес-  
 2585 конечное количество решений (тут важно, что  $d = \gcd(a, b)$ ). Алгоритм Евклида  
 2586 находит в некотором смысле наименьшее решение, а именно такое, что  $|x| \leq b$  и  
 2587  $|y| \leq a$ .

2588 Пусть пусть  $a = bq + r$ , где  $r = a \bmod b$ , и  $\gcd(a, b) = d$ . Предположим, что мы  
 2589 нашли некоторые  $x'$  и  $y'$  такие, что

$$2590 \quad bx' + ry' = d.$$

2591 Покажем, как вычислить  $x$  и  $y$ . Подставим  $r = a - bq$ .

$$2592 \quad bx' + (a - bq)y' = d.$$

2593 Перегруппируем члены этого выражения:

$$2594 \quad ay' + b(x' - qy') = d.$$

2595 Получается, что  $x = y'$ , а  $y = x' - qy'$ . Что приводит нас к следующему алгоритму.

```
2596 # расширенный алгоритм Евклида
2597 # вычисление наибольшего общего делителя и коэффициентов Безу
2598 # возвращает тройку из НОД и коэффициентов Безу
2599 def egcd(a, b):
2600     if b == 0:
2601         return (a, 1, 0)
2602
2603     # вычисляем q и r
2604     (q, r) = divide(a, b)
```

```

2606     # рекурсивно вычисляем d, x' и y'
2607     (d, x_, y_) = egcd(b, r)
2608
2609     # вычисляем x и y
2610     x = y_
2611     y = x_ - q * y_
2612
2613     return (d, x, y)

```

Для оценки сложности этого алгоритма заметим, если вызвать алгоритм для  $a < b$ , то первый же рекурсивный вызов «переставит» аргументы местами, и во всех следующих вызовах  $a \geq b$ . Если  $a \geq b$ , то  $r < a/2$ , т.е один из аргументов на каждой итерации уменьшается как минимум вдвое. Действительно, если  $a \geq 2b$ , то  $r < b \leq a/2$ , если же  $a < 2b$ , то  $r = a - b < a/2$ . Поэтому можно оценить время работы этого алгоритма для двух для  $n$ -битовых как  $O(n^3)$ , т.к. глубина рекурсии ограничена  $O(n)$  и на каждой итерации требуется вычислить  $\text{divide}(a, b)$ , который мы умеем вычислять за  $O(n^2)$ .

Более аккуратный анализ позволяет получить оценку  $O(n^2)$ . Для этого сначала докажем, следующую лемму.

**Лемма 8.1.1.** Для любых целых  $a$  и  $b$ ,  $a \geq b > 0$ , функция  $\text{egcd}(a, b)$  вернёт тройку  $(d, x, y)$ , где  $|x| \leq b$  и  $|y| \leq a$ .

**Доказательство.** Будем доказывать по индукции. В качестве базы рассмотрим случай, когда  $a$  делится на  $b$ . Тогда  $r$  будет равен нулю, а следовательно, рекурсивный вызов  $\text{egcd}(b, r)$  вернёт  $(b, 1, 0)$ . Получаем, что  $|x| = 0 \leq b$  и  $|y| = 1 \leq a$ , т.е. для этого базового случая лемма верна.

Теперь рассмотрим случай, когда  $a > b > 0$ , и  $a$  не делится на  $b$ . Предположим, что для всех  $a' < a$  и  $b' < b$  лемма верна. Положим, как в алгоритме,  $q = \lfloor a/b \rfloor$  и  $r = a \bmod b$ . Тогда вложенный в  $\text{egcd}(a, b)$  вызов  $\text{egcd}(b, r)$  вернёт тройку  $(d, x', y')$ , для которой по предположению индукции выполняется  $|x'| \leq r$  и  $|y'| \leq b$ . Тогда

$$\begin{aligned} |x| &= |y'| \leq r = a \bmod b < b, \\ |y| &= |x' - qy'| \leq |x'| + |qy'| \leq r + qb = a. \quad \square \end{aligned}$$

Теперь используя эту лемму покажем, что  $n$ -битового  $a$  и  $m$ -битового  $b$ , где  $n \geq m$ , алгоритм выполняется за  $O(n^2)$  операций. Будем тоже доказывать это по индукции. База индукции: для однобитных чисел достаточно константного количества операций. Предположим, что для любых пар чисел, длины которых не превосходят  $m$ , соответственно, алгоритм выполняется за  $O(m^2)$ . Тогда сложность вычисления  $\text{egcd}(a, b)$  складывается из сложности вызова  $\text{divide}$ , рекурсивного вызова  $\text{egcd}$  и сложности вычисления  $y$ , т.е.  $O(n(n-m+1))$ ,  $O(m^2)$  и  $O(|x'| + |q| \cdot |y'|)$ . Последнее слагаемое по лемме 8.1.1 можно ограничить  $O(n + (n-m) \cdot m)$ . Все три слагаемые не превосходят  $O(n^2)$ , так мы и получаем желаемую оценку.

**Замечание 8.1.2.** В данном алгоритме нам потребуется работа с отрицательными числами. Для представления отрицательных чисел потребуется ещё один дополнительный бит, отвечающий за знак числа, т.е. для хранения  $n$ -битового числа нужен  $n + 1$  бит. Кроме того, в реализацию операций

Ещё один полезный алгоритм — это алгоритм для возведения в степень. Найвный алгоритм, который для вычисления числа  $a^b$  в цикле  $b$  умножает на  $a$  (и,

следовательно, требует  $b$  умножений), нельзя назвать эффективным. Дело в том, что значение  $n$ -битового числа может достигать  $2^n$ , а значит время предложенного алгоритма экспоненциально от длины числа  $b$ . Более эффективный алгоритм можно построить, если действовать аналогично алгоритму умножения, рассмотренному выше.

```

# Быстрое возведение в степень
def power(a, b):
    if b == 0:
        return 1

    # рекурсивно вычисляем  $a^{\lfloor b/2 \rfloor}$ 
    res = power(a, b // 2)

    # возводим результат в квадрат
    res = res * res

    # обработка нечётного b
    if b % 2 == 1:
        res = res * a

    return res

```

Давайте проанализируем время работы этого алгоритма для пары  $n$ -битовых чисел  $a$  и  $b$ . Глубина рекурсии не превосходит  $n$ , а на каждом уровне рекурсии вычисляется три умножения. В результате получаем оценку  $O(n^3)$ , что значительно лучше экспоненциальной оценки наивного алгоритма.

К сожалению, в предыдущем рассуждении мы совершили очень грубую ошибку. Мы предположили, что все умножения будут стоить не более  $O(n^2)$ , но эта оценка верна только для  $n$ -битовых чисел. Давайте проанализируем, какой длины будут числа, которые получаются в данном алгоритме. Для простоты будем считать, что  $a = b = 2^n$ . Сколько бит в числе  $a^b$ ? Количество битов в двоичной записи числа с точностью до константы равно двоичному логарифму этого числа. В нашем случае  $\log_2 a^b = \log_2 2^{n2^n} = n2^n$ . Таким образом на верхнем уровне рекурсии будут умножаться два  $n2^{n-1}$ -битных числа. Соответственно, сложность этого алгоритма уж точно не меньше  $\Omega(n^2 2^{2n})$ , т.е. алгоритм имеет экспоненциальную сложность. Эта ошибка показывает, насколько внимательным нужно быть при оценке сложности алгоритмов.

## 8.2. Модульная арифметика

*Модульная арифметика* — это работа с целыми числами в кольце остатков по некоторому модулю  $m$ , т.е. все числа и все результаты операций берутся по модулю  $m$ . Реализовать сложение двух чисел по модулю очень просто — нужно вычислить результат как с обычными целыми числами, а от результата взять остаток по модулю  $m$ .

```

# Сложение двух чисел по модулю
def plus_mod(a, b, m):
    return (a + b) % m

```

```

2696
2697 # Умножение двух чисел по модулю
2698 def multiply_mod(a, b, m):
2699     return (a * b) % m
```

Пусть  $m$  —  $n$ -битное число. Тогда оба алгоритма имеют сложность  $O(n^2)$ .  
 Осталось разобраться с делением по модулю. Деление по модулю — это деление в кольце остатков, оно работает не так, как деление обычных целых чисел.  
 Для того, чтобы поделить  $a$  на  $b$  по модулю  $m$ , нужно найти такое число  $q \in [m-1]$ , что

$$bq = 1 \pmod{m}.$$

Такое число  $q$  называется *обратным* к  $b$  и обозначается  $b^{-1}$ . Теперь можно определить деление на  $b$  как умножение на  $b^{-1}$ :

$$a/b = ab^{-1} \pmod{m}.$$

Нетрудно проверить, что обратные могут быть только у чисел взаимно простых с  $m$ . Пусть  $\gcd(b, m) = d > 1$ . Тогда  $bq$  делится на  $d$  для любого  $q$ , а следовательно остаток от  $bq$  при делении на  $m$  тоже делится на  $d$ , т.е.  $(bq \bmod m) \neq 1$ .

Пусть  $b$  и  $m$  взаимно просты. Как найти обратное к  $b$ ? Тут нам поможет расширенный алгоритм Евклида. Давайте найдём коэффициенты Безу для  $b$  и  $m$ . Это будет пара  $x$  и  $y$  удовлетворяющие следующему соотношению  $bx + my = 1$ . Рассмотрим это соотношение по модулю  $m$ . Тогда в левой части второе слагаемое обнуляется, и мы получаем

$$bx = 1 \pmod{m},$$

т.е.  $x$  — обратное к  $b$ . Осталось заметить, что расширенный алгоритм Евклида гарантирует, что  $|x| \leq m$ . Это даёт следующий алгоритм деления.

```

2720 # Деление на a на b по модулю m
2721 # предполагается, что gcd(b, m) = 1
2722 def divide_mod(a, b, m):
2723     # вычисляем коэффициенты Безу
2724     (d, x, y) = egcd(b, m)
2725
2726     # обработка отрицательного x
2727     if x < 0:
2728         x = x + m
2729
2730     return (a * x) % m
```

Сложность полученного алгоритма  $O(n^2)$ .

Давайте также проанализируем следующий код, вычисляющий  $a^b \bmod m$ .

```

2733 # Быстрое возведение в степень по модулю
2734 def power_mod(a, b, m):
2735     if b == 0:
2736         return 1
2737
2738     # рекурсивно вычисляем  $a^{\lfloor b/2 \rfloor} \bmod m$ 
2739     res = power_mod(a, b // 2, m)
2740
```

```
2741     # возводим результат в квадрат
2742     res = res * res
2743
2744     # обработка нечётного b
2745     if b % 2 == 1:
2746         res = res * a
2747
2748     return res % m
```

2749 Поскольку все числа в этом алгоритме не превосходят  $m^3$ , то алгоритм имеет  
2750 сложность  $O(n^5)$  (теперь без подвоха).