1 Бинарное дерево поиска

В каждом узле бинарного дерева поиска хранятся *ключ* a и два поддерева, правое и левое. Все ключи в левом поддереве не превосходят a, а в правом — не меньше a. Алгоритм поиска — начиная с корня, сравниваем искомый ключ с ключом в узле, в зависимости от сравнения спускаемся в правое или в левое поддерево.

Вставка в бинарное дерево — поиск + вставляем туда, куда пришёл поиск. Чтобы удалить элемент — ставим на его место самый левый элемент в его правом поддереве.

Проблема такой наивной структуры — может вместо дерева получиться палка (если, например, ключи приходят в порядке по убыванию), и поиск будет занимать $\mathcal{O}(n)$. Красно-чёрные деревья, например, следят за тем, чтобы дерево всегда имело высоту $\mathcal{O}(\log n)$.

Definition 1. Дерево называется идеально сбалансированным (perfectly balanced tree), если размеры детей каждой ее вершины отличаются не больше, чем на 1.

Хотим научиться поддерживать ±баланс, не храня много дополнительной информации (такой, как атрибуты red/black) — в идеале, $\mathcal{O}(1)$ дополнительных данных, какие-нибудь несколько чисел про дерево в целом. Это умеют две структуры.

2 Scapegoat tree

Источники: [GR93; And89]. Мы в основном опираемся на [GR93].

2.1 Структура дерева

Зафиксируем константу $\frac{1}{2} < \alpha < 1$. Будем рассматривать структуру данных, в которой хранится дерево tree. Также будем хранить текущее количество узлов в дереве — size. У каждого узла node есть дети left, right и ключ key.

structure TREE
    root
    size
    maxSize
structure NODE
    left, right
    key

Мы хотим, чтобы глубина дерева была $\mathcal{O}(\log n)$, где n — количество узлов в дереве. Для этого заведем несколько условий

condition α-WEIGHT(node x)
    $\max\{\text{size}(x.\text{left}), \text{size}(x.\text{right})\} \le \alpha \cdot \text{size}(x)$
condition α-HEIGHT(node x)
    $depth(x) \le \lfloor \log \text{size} \rfloor + 1$
condition WEAK α-HEIGHT(node x)
    $depth(x) \le \lfloor \log \text{maxSize} \rfloor + 1$ ▷ maxSize will be defined later

Желаемая максимальная высота дерева (n — количество узлов с ключами) — $\mathcal{O}(\log_{\frac{1}{\alpha}} n)$.

Если $\alpha = \frac{1}{2}$, то результатом будет идеально сбалансированное дерево, то есть α — это, грубо говоря, разрешённое отклонение размера поддеревьев от состояния баланса.

Узел называется *глубоким,* если он нарушает weak α-height condition. Глубокие узлы мы не любим и каждый раз, когда они у нас будут появляться, мы будем переподвешивать часть дерева так, чтобы они переставали быть глубокими.

Заметим, что если дерево α-weight balanced, то оно и α-height balanced. Обратного следствия нет, потому что может быть «один сын справа, а слева сбалансированное поддерево».

Иногда мы будем перестраивать все дерево. Чтобы реализовать вставку и удаление, нам также потребуется хранить величину maxSize для всего дерева tree. maxSize — штука, отвечающая какой максимальный размер был у дерева с момента последней его полной перестройки. (То есть, кроме собственно дерева с ключами, мы храним дополнительно только size и maxSize — два числа.) Также, нам понадобится еще один инвариант для нашего дерева

Инвариант: $\alpha \cdot \text{maxSize} \le \text{size} \le \text{maxSize}$

Заметим, что из этого инварианта следует, что глубина дерева без глубоких вершин не превосходит $\mathcal{O}(\log n)$.

Удаление: просто удаляем. Проверяем, не нарушился ли инвариант. Если нарушился — просто перестроим всё дерево с нуля, сделав массив с ключами за линию и соорудив из него идеально сбалансированное дерево. size при этом уменьшается на 1, а maxSize = size.

Вставка: сначала стандартная вставка, добавляем ключ в лист. При этом size увеличивается на 1,

$$\text{maxSize} \coloneqq \max\{\text{maxSize}, \text{size}\}.$$

Может, однако, оказаться так, что новый узел x оказался глубоким. Тогда рассмотрим путь от x до корня $a_0 \dots a_H$ и найдём среди этих узлов (просто за линию, посчитав количество) самый нижний, не сбалансированный по весу (такой найдётся, докажем) и перестраиваем (глупо, за линию) дерево под ним.

Theorem 1. *Среди $a_0 \dots a_H$ всегда найдётся узел, не сбалансированный по весу (козёл отпущения).*

Доказательство. Пусть нет, тогда $\text{size}(a_i) \le \alpha \cdot \text{size}(a_{i+1})$. Тогда $\text{size}(x) \le \alpha^H \cdot \text{size}(T)$. Прологарифмируем это неравенство по основанию $\frac{1}{\alpha}$:

$$0 \le -H + \log_{\frac{1}{\alpha}} n$$

□

Theorem 2. *При вставке элемента сохраняется сбалансированность по высоте.*

Доказательство. Интересен только случай, когда вставленный элемент глубокий. Достаточно показать, что при перестройке глубина перестроенного поддерева уменьшится. Заметим, что у нас в каждый момент времени бывает не более одного глубокого элемента (при вставке может появится только один, вот-вот вставленный, а при удалении maxSize меняется только если все дерево было перестроено), значит, глубина поддерева может остаться прежней тогда и только тогда, когда выбранное поддерево состояло из полного поддерева с добавленным к нему одним глубоким элементом. Но такое поддерево удовлетворяет условию сбалансированности по весу, а значит, мы его не могли выбрать. □

Корректность мы показали, но у нас остались операции перестройки, которые работают в худшем случае за линию. Покажем, что они хорошо амортизируются.

2.2 Время работы

Сначала разберемся с перестройкой дерева при удалении. Эта операция линейна и происходит не чаще, чем раз в $\alpha \cdot \text{size}(T)$ операций удаления, а значит, имеет ее амортизированная сложность $\mathcal{O}(1)$.

Осталась операция перестройки нижнего несбалансированного поддерева при вставке. Пусть корень этого дерева — x. У этого поддерева есть больший ребенок (не умаляя общности будем считать, что он левый) и меньший (соответственно, правый). Рассмотрим все операции вставки в левое поддерево и удаления из правого поддерева с момента последней перестройки какого-либо родителя x. Для того, чтобы x перестал быть сбалансированный по высоте, их количество должно быть хотя бы линейно от $\text{size}(x)$. Сопоставим все эти операции перестройке дерева. Заметим, что каждая вставка и удаление была сопоставлена не более чем $\mathcal{O}(\log n)$ перестройкам, значит, амортизированная сложность этих операций не увеличилась. При этом каждой перестройке мы сопоставили линейное количество вставок и удалений, значит, амортизированная сложность всех перестроек не превосходит $\mathcal{O}(1)$.

Таким образом, операции вставки у удаления работают за амортизированное время $\mathcal{O}(\log n)$.

3 Splay tree

Оригинальная статья: [ST85]

3.1 Общая структура дерева

В этом дереве мы каждый раз, когда захотим что-то сделать с вершиной, будем поднимать ее до корня (операция splay). В самом дереве в этот раз мы можем не хранить ничего, кроме корня root. Но часто хочется уметь быстро считать размер дерева, для этого можно хранить отдельную переменную size для всего дерева.

structure TREE
    root
    size ▷ optional
structure NODE
    left, right
    key

Выразим сначала операции insert и erase через операцию splay, а потом будем разбираться со splay. Для erase нам понадобится операция splay_front(node). Эта операция делает splay для наименьшего ключа в поддереве.

```
1: procedure INSERT(x)
2:     standard_insert(x)
3:     splay(x)
4: procedure GET(x)
5:     splay(x)
6: procedure ERASE(x)
7:     splay(x)
8:     splay_front(root.right)
9:     standard_erase(x)
```

Два вызова функции splay при удалении нужны для того, чтобы правый сын корневой вершины не имел левого сына (потому что он содержит наименьший ключ в своем поддереве) и операция standard_erase(x) работала за $\mathcal{O}(1)$ (потому что она просто возьмет этого правого сына и поставит на место удаленного корня). Еще стоит отметить, что даже при простом доступе к вершине мы вызываем операцию splay, это нужно потому что наше дерево может иметь довольно большую глубину во время работы, а оценка у нас будет только на амортизированную сложность операции splay.

Ниже мы будем оценивать сложность splay при фиксированном множестве ключей в дереве, покажем, что этого достаточно. Удаление вершины из дерева испортить время работы очевидно не сможет, а при добавлении мы спускаемся на полную глубину дерева и можно считать, что искомая вершина была в дереве всегда, просто мы ее не трогали до момента добавления. Тут стоит обратить внимание на то, что с таким подходим, если у нас был какой-то ключ, мы его удалили, а потом добавили обратно, то в оценке времени работы их надо рассматривать как два различных ключа.

3.2 Splay

Итак, нам надо научиться понимать вершину в корень. Это делается при помощи нескольких видов вращений дерева. Все вращения в дальнейшем будем рассматривать с точностью до симметрии. Простейшее вращение называется zig (см. рис. 1). Легко видеть, что это вращение поднимает вершину x на один уровень выше. При помощи одного этого вращения можно поднять вершину в корень, но для амортизационного анализа нам этого не хватит, поэтому мы будем делать сразу двойные вращения.

Двойные вращения бывают двух видов: zig-zig (рис. 2) и zig-zag (рис. 3). Оба эти вращения реализуются при помощи пары вращений zig, но для того, чтобы выразить

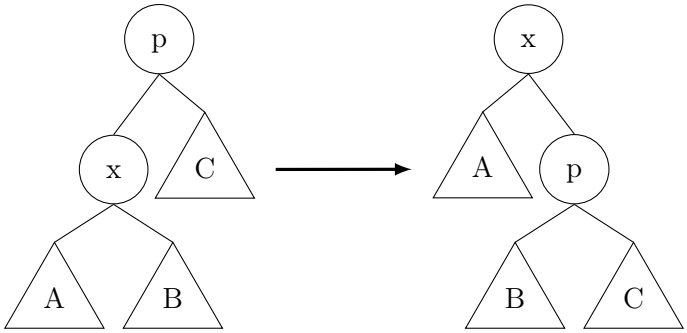


Рис. 1: Zig

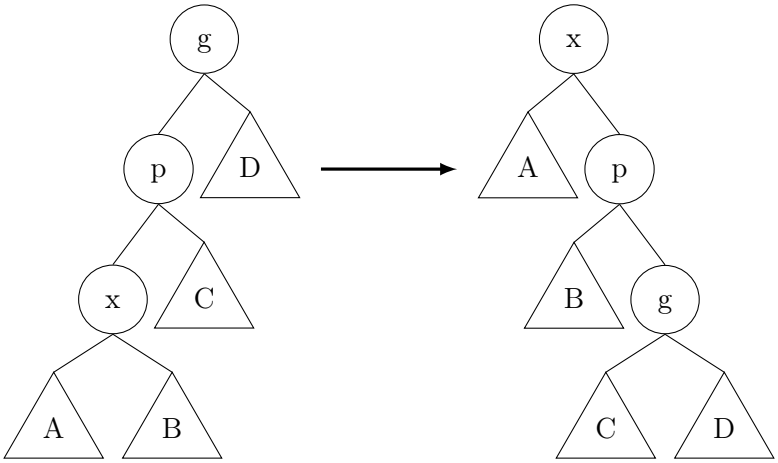


Рис. 2: Zig-zig

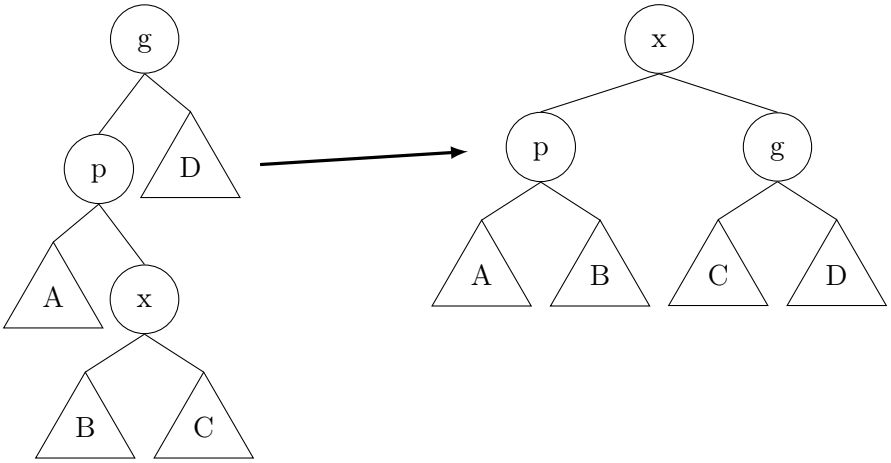


Рис. 3: Zig-zag

zig-zig, надо сначала выполнить zig от вершины p, и только потом от x. Zig-zag при этом выражается как два вызова zig от x. Стоит отметить, что при splay мы не сможем выполнить двойное вращение, если интересующая нас вершина непосредственный сын корня, тогда мы должны сделать zig и не забыть его посчитать при анализе (но он может быть только один).

Для анализа, мы воспользуемся методом потенциалов. Для начала заведем функцию $w : \text{keys} \to \mathbb{R}_+$. На нее тоже будут какие-то условия. Про то, какой она может быть, поймем позже, пока можно считать, что она всегда возвращает 1, реально менять ее придется только для следствий. Определим функцию «размера» поддерева $s(x) = \sum_{v \in \text{subtree of } x} w(v)$ и функцию «ранга» $r(x) = \log_2 s(x)$ (логарифм двоичный, это неожиданно важно, но дальше основание писать не будем), а функцией потенциала всего дерева T будет $\Phi(T) = \sum_{x \in T} r(x)$. Для того, чтобы метод потенциалов работал, нужно чтобы Φ всегда было неотрицательно (ну или придется оценить оценить насколько сильно оно бывает отрицательным и прибавить к асимптотике). При $w \equiv 1$ это очевидно, а вообще это надо запомнить как первое ограничение на w. Амортизированная стоимость операции splay am.cost $= \Delta\Phi +$ #rotations (да, это просто определение). Пусть мы выполнили один splay. Теперь $r(x)$ и $s(x)$ будут обозначать значения до вызова операции, а $r'(x)$ и $s'(x)$ — после. Тогда на самом деле мы хотим доказать следующую теорему:

Theorem 3. *am.cost* $\leq 3(r'(x) - r(x)) + \mathcal{O}(1)$

Доказательство. Надо оценить $\Delta\Phi$ для каждого из вращений.

Zig:

$$\begin{aligned}
\Delta\Phi &= r'(p) - r(p) + r'(x) - r(x) \\
&= r'(p) - r(x) && \text{since } r'(x) = r(p) \\
&\leq r'(x) - r(x) && \text{since } p \text{ is lower than } x \text{ after zig}
\end{aligned}$$

Дополнительно стоит отметить, что $r'(x) \geq r(x)$ поскольку слева написана сумма по большему множеству, поэтому если мы вдруг захотим это умножить на какую-нибудь произвольно взятую константу 3, ничего не испортится.

Zig-zig:

$$\begin{aligned}
\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\
&= r'(g) + r'(p) - r(p) - r(x) \\
&\leq r'(g) + r'(x) - 2r(x) && \text{due to the tree structure} \\
&\leq 3(r'(x) - r(x)) - 2 && \text{since } r'(g) + r(x) \leq 2(r'(x) - 1)
\end{aligned}$$

Осталось показать почему $r'(g) + r(x) \leq 2(r'(x) - 1)$.

$$\begin{aligned}\frac{r'(g)+r(x)}{2} &= \log s'(g) + \log s(x) \\ &\le \log\left(\frac{s'(x)-w(p)}{2}\right) && \text{Jensen's inequality} \\ &= \log\bigl(s'(x)-w(p)\bigr) - 1 \\ &\le r'(x) - 1\end{aligned}$$

Zig-zag:

$$\begin{aligned}\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\ &= r'(g) + r'(p) - r(p) - r(x) \\ &\le r'(g) + r'(p) - 2r(x) && \text{due to the tree structure} \\ &\le 3(r'(x) - r(x)) - 2 && \text{since } r'(g)+r'(p) \le 2(r'(x)-1)\end{aligned}$$

Доказательство неравенства $r'(g) + r'(p) \le 2(r'(x) - 1)$ в точности повторяет доказательство аналогичного неравенства выше.

Изменения потенциала от каждого двойного вращения мы оценили как $3(r'(x) - r(x)) - 2$. Все наши страдания были на самом деле направлены на то, чтобы получить двойку в конце. Теперь, когда мы просуммируем по всем вращениям при операции splay, мы получим оценку $\Delta\Phi \le 3(r'(x) - r(x)) - \#\text{rotations} + \mathcal{O}(1)$, поскольку все промежуточные $r(x)$ скомпенсируются, zig будет вызван не более одного раза, а в оценке двойных вращений есть слагаемое -2, которые просуммируются в количество вращений. Таким образом, am.cost $= \Delta\Phi + \#\text{rotations} \le 3(r'(x) - r(x)) + \mathcal{O}(1)$, что нам и надо. □

Ниже мы будем считать, что наше дерево работает с ключами $1 \dots n$, выполняет m операций, а $W \coloneqq \sum_i w(i)$. Теперь нам надо выбрать w. Надо вспомнить какие условия ограничения мы насобирали на w. Ограничения у нас появлялись в двух местах: из определения $w > 0$ (потому что мы потом хотим логарифмировать) и из метода потенциалов $\Phi \ge 0$. При $w \ge 1$ потенциал неотрицателен автоматически, поскольку все слагаемые неотрицательны.

Corollary 4 (Balance Theorem)**.** *Амортизированное время работы на любой последовательности из m запросов $\mathcal{O}(m \log n + n \log n)$.*

Доказательство. Берем $w(x) = 1$. □

Corollary 5 (Static Optimality Theorem)**.** *Пусть q_x — количество доступов к элементу x. Тогда амортизированное время работы $\mathcal{O}\Bigl(m + \sum_x q_x \log\bigl(\frac{m}{q_x}\bigr)\Bigr)$.*

Доказательство. Берем $w(x) = q_x$. □

Из этой теоремы следует, что splay деревья работают не хуже (с точностью до константного множителя, конечно), чем оптимальное статическое дерево поиска. Аналогичное утверждение про динамические деревья остается открытой проблемой.

Conjecture 6 (Dynamic Optimality Conjecture)**.** *Пусть A — произвольное двоичное дерево поиска, которое может делать некоторые вращения (Zig, рис. 1), и обрабатывать запрос на доступ к вершине за ее глубину. Обозначим $A(S)$ — время работы A на последовательности запросов S. Тогда время работы splay дерева на последовательности S не превосходит $\mathcal{O}(n + A(S))$.*

Для следующего следствия стоит вспомнить, что мы считаем, что элементы $1 \dots n$.

Corollary 7 (Static Finger Theorem)**.** *Пусть f — некоторый фиксированный элемент, «finger». Тогда время работы $\mathcal{O}\Big(m + n \log n + \sum_{x\ -\ запрос} \log(|x - f| + 1)\Big)$.*

Доказательство. Берем $w(x) = \frac{1}{(|x-f|+1)^2}$. Тогда $W = \mathcal{O}(1)$, а потенциал может быть отрицательным, но не больше, чем на $\mathcal{O}(n \log n)$, поскольку $w \geq \frac{1}{n^2}$, это слагаемое мы можем просто искусственно добавить к потенциалу и, следовательно, асимптотике. □

Corollary 8 (Dynamic Finger Theorem)**.** *Аналогично, но теперь f, «finger» — элемент, к которому обращались предыдущим запросом (и, следовательно, находящийся в корне). Тогда время работы $\mathcal{O}\Big(m + n + \sum_{x\ -\ запрос} \log(|x - f| + 1)\Big)$.*

Доказательство. Надо бы как-нибудь доказать. □ todo 1

Theorem 9 (Scanning Theorem or Sequential Access Theorem or Queue theorem)**.** *Доступ к элементам в порядке возрастания работает за амортизированную единицу на запрос.*

Доказательство. Следует из Dynamic Finger Theorem (Corollary 8). □

4 Об оффлайн деревьях поиска: нижняя граница времени работы, геометрическое представление

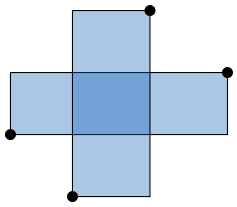
4.1 Основные определения и предваряющие результаты

Пусть дано бинарное дерево поиска с n ключами. Мы знаем последовательность запросов, которые зададим этому дереву: $P = \{s_1, s_2, \dots, s_m\}$. В поисках ключей s_i мы будем бегать по дереву туда-сюда и в процессе спуска/подъёма пройдём через некоторые вершины, которые нам не нужны.

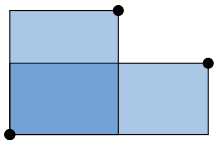
Definition 2. $E(P)$ — множество всех вершин, которые мы посетим в процессе поиска вершин с ключами из P. $E = P \cup X$, X — множество «лишних» вершин.

Definition 3. OPT — минимальный размер $E(P)$ (обозначение множества P будем опускать, и так по контексту ясно).

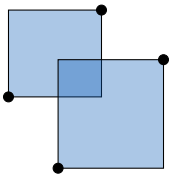
Definition 4. Пусть $a, b \in \mathbb{R}^2$. Тогда $\mathrm{rect}(a, b)$ — прямоугольник, стороны которого параллельны осям координат, а противоположные вершины — точки a и b. Его же будем называть *прямоугольник, определённый точками a, b.*



(a) Эти прямоугольники независимы



(b) Эти прямоугольники независимы



(c) Эти прямоугольники **не** независимы

Рис. 4: Примеры прямоугольников, независимых и не очень

Definition 5. Конечное множество $G \subset \mathbb{R}^2$ называется *arborally satisfiable,* если

$$\forall\, a, b \in G \quad x(a) = x(b), \ \text{либо} \ \ y(a) = y(b), \ \text{либо}$$
$$\exists\, c \in \operatorname{rect}(a, b) \ \ \text{(внутри или на границе)}.$$

Theorem 10 (Доказана ранее)**.** *Рассмотрим последовательность запросов*

$$\{(s_1, 1), (s_2, 2), \ldots, (s_m, m)\} \subset \mathbb{Z}^2.$$

Надмножество этой последовательности может представлять из себя последовательность узлов, которые были посещены при поиске $s_1, \ldots, s_m$, в том и только том случае, если оно arborally satisfiable.

Далее мы будем рассматривать изображение последовательности запросов на плоскости, соответственно под множеством P будем понимать $\{(s_1, 1), (s_2, 2), \ldots, (s_m, m)\}$, аналогично вторую координату приделаем к ключам вершин из множества E.

Definition 6. Пусть дано множество P и его надмножество E. Два прямоугольника, определённых каждый двумя вершинами множества P, будем называть *независимыми* (смотреть Рисунок 4), если

1) они оба не arborally satisfiable, то есть им не принадлежит ни одна точка из E,

2) ни одна из вершин одного из этих прямоугольников не лежит во внутренности другого.

4.2 Оценка снизу числа OPT

Definition 7. Будем говорить, что прямоугольник, определённый точками (x_1, y_1), (x_2, y_2), *имеет тип «+»,* если $(x_1 - x_2) \cdot (y_1 - y_2) \geq 0$, иначе прямоугольник *имеет тип «−»* (смотреть Рисунок 5).

Рис. 5: Прямоугольники типа «+» и типа «−».

Definition 8. $\mathrm{MAX\,IND}$ — наибольшее число попарно независимых прямоугольников, определённых точками из P. Соответственно, $\mathrm{MAX\,IND}_+$, $\mathrm{MAX\,IND}_-$ — наибольшие количества попарно независимых прямоугольников фиксированного типа.

Theorem 11.

$$\mathrm{OPT} \;\geq\; |P| + \frac{1}{2}\,\mathrm{MAX\,IND}\,. \tag{1}$$

Прежде чем приступить к доказательству Теоремы 11, докажем следующую лемму:

Lemma 12.

$$\mathrm{OPT}_+(P) \;\geq\; |P| + \frac{1}{2}\,\mathrm{MAX\,IND}_+(P). \tag{2}$$

Здесь OPT_+ — количество точек в множестве $E(P)$, нужное для того, чтобы множество всех прямоугольников типа «+» было arborally satisfiable. Это более слабое условие.

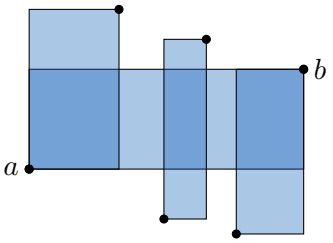
Далее мы забываем о том, что множества точек, с которыми мы работаем, — это вообще говоря выходы какой-то процедуры поиска, и рассматриваем произвольные конечные множества точек на плоскости.

Доказательство Леммы 12. Пусть все координаты точек из P различны (точки можно чуть-чуть пошевелить, чтобы это стало так и ничего больше не нарушилось). Рассмотрим максимальный набор попарно независимых «+»-прямоугольников и самый широкий из них — пусть он определён точками a, b. Некоторые прямоугольники будут пересекать наш самый широкий прямоугольник, одной из их вершин может быть a или b, либо их определяющие вершины будут лежать за границами самого широкого прямоугольника, одна выше, одна ниже, смотреть Рисунок 6a.

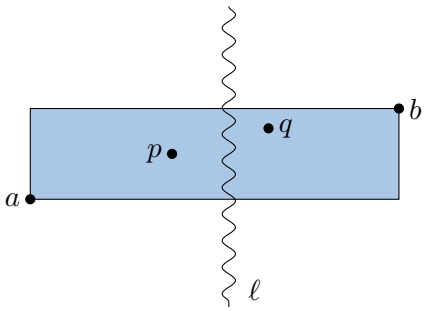
Прямоугольники, имеющие своей вершиной a, не пересекаются с прямоугольниками, имеющими своей вершиной b, потому что иначе получается случай прямо как на Рисунке 4c. Более того, оставшиеся прямоугольники тоже не могут никак налезать друг на друга, потому что опять же получится случай с Рисунка 4c. Поэтому существует вертикальная линия, пересекающая *только* выбранный нами широкий прямоугольник $\mathrm{rect}(a, b)$, обозначим её через ℓ, смотреть Рисунок 6b.

Рассмотрим самую верхнюю, самую правую точку из $E(P)$, которая левее ℓ и принадлежит $\mathrm{rect}(a, b)$, обозначим её через p. Такая точка точно существует, потому что как минимум a подойдёт, мы выбираем из непустого множества. Рассмотрим самую нижнюю, самую левую точку из $E(P)$, которая правее ℓ, принадлежит $\mathrm{rect}(a, b)$ и не ниже p, обозначим её через q. Опять же такая найдётся, потому что есть b.

Claim 13. *Точки p и q лежат на одной горизонтали.*



(a) Прямоугольники, независимые с $\mathrm{rect}(a, b)$



(b) Вертикальная линия, не пересекающая ни один из прямоугольников набора. Две точки, соответствующие прямоугольнику $\mathrm{rect}(a, b)$

Рис. 6: Доказательство Леммы 12

Иначе образованный ими прямоугольник должен быть arborally satisfiable, и это бы значило, что мы неправильно выбрали p, q (найдётся точка из $E(P)$, принадлежащая прямоугольнику $\mathrm{rect}(p, q)$ и лежащая ближе к ℓ). Сопоставим прямоугольнику $\mathrm{rect}(a, b)$ горизонтальный отрезок pq, удалим этот прямоугольник из набора и продолжим сопоставление.

Claim 14. *Каждый отрезок pq сопоставлен не более чем одному $\mathrm{rect}(a, b)$ из набора независимых прямоугольников.*

Потому что pq лежит внутри $\mathrm{rect}(a, b)$ и пересекает линию, которую не пересекает больше никто из прямоугольников набора, имеющих общие точки с $\mathrm{rect}(a, b)$. Остальные прямоугольники из выбранных нами независимых просто не пересекаются с $\mathrm{rect}(a, b)$.

Рассмотрим точки $p_1 \dots p_t$, $q_1 \dots q_t$, отрезки с концами в которых были сопоставлены некоторым прямоугольникам и которые все оказались на одной горизонтальной прямой.

Claim 15. *Точки p_i, q_i (соответствующие одному прямоугольнику) — соседние из отмеченных точек на этой горизонтальной прямой.*

В противном случае отрезок $p_i q_i$ будет пересекать какой-то другой отрезок $p_j q_j$. И в процессе сопоставления точек соответствующим прямоугольникам мы бы взяли какие-то другие точки.

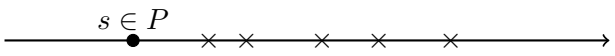


Рис. 7: Точки, добавленные в данную строку

Рассмотрим некоторую строку, в ней находится одна точка из исходного множества запросов P, смотреть Рисунок 7. Пусть мы добавили в эту строку ещё n точек, сопоставленных различным независимым прямоугольникам. Тогда точек стало $n + 1$, и наибольшее число прямоугольников, которое может им соответствовать, — n, потому что Утверждение 15. То есть на одну точку из $E(P)$ добавляется не более одного

прямоугольника. Лемма доказана. □

Доказательство теоремы 11.

$$\mathrm{OPT} \;\geq\; \max(\mathrm{OPT}_+, \mathrm{OPT}_-).$$

Теперь воспользуемся тем, что максимум не меньше среднего, а также леммой 12.

$$\begin{aligned}\max(\mathrm{OPT}_+, \mathrm{OPT}_-) \;&\geq\; |P| + \frac{1}{2}(\mathrm{MAX\,IND}_+ + \mathrm{MAX\,IND}_-) \;\geq \\ &\geq\; |P| + \frac{1}{2}\cdot \mathrm{MAX\,IND}\,.\end{aligned}$$

□

4.3 Более практичная оценка снизу

Рассмотрим пару (s_i, i) из набора поисковых запросов. Упорядочим все остальные точки (s_j, j), $j < i$ по второй координате и соединим их y-монотонной ломаной сверху вниз, смотреть Рисунок 8.

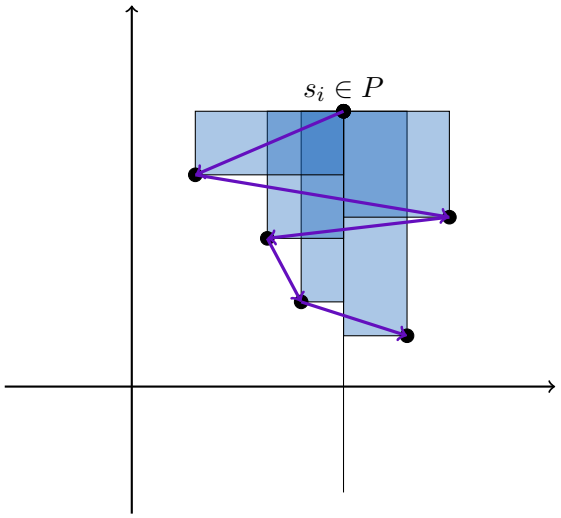


Рис. 8: Подсчёт числа пересечений с вертикальной прямой

Обозначим через $J(s_i)$ количество пересечений этой ломаной с вертикальным лучом, идущим из s_i вниз. Понятно, что такое число можно посчитать для любого элемента последовательности запросов.

Theorem 16.

$$\mathrm{OPT}(P) \;\geq\; |P| + \sum_{s_i} \frac{J(s_i)}{2} \tag{3}$$

Доказательство. На каждом ребре ломаной, пересекающем вертикальный луч, построим как на диагонали прямоугольник, стороны которого параллельны осям координат. Так у каждого пересечения появится свой прямоугольник. Объединим получившиеся наборы прямоугольников, смотреть Рисунок 9.

Все прямоугольники в объединении, легко видеть, будут попарно независимы. Осталось лишь применить теорему 11. □

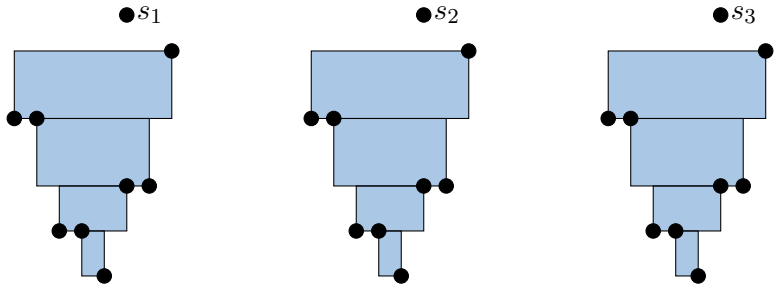


Рис. 9: Набор попарно независимых прямоугольников

4.4 Оценка снизу через число перебежек

Рассмотрим вершину q бинарного дерева поиска T. Обозначим через $R(q)$ количество чередований между спусками в левое поддерево q и правое поддерево q. Спуски в сам узел q и всё, что происходит вне поддерева q, при этом игнорируется.

Theorem 17.

$$\mathrm{OPT}(P) \;\geq\; \sum_{q \in T} R(q). \tag{4}$$

Доказательство. Следует из Теоремы 16. □

0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Рис. 10: Bit-reversal sequence делает нижнюю оценку бессмысленно большой

5 Tango деревья

Дерево, где у каждой вершины есть «любимый потомок» — тот, в которого происходил спуск при предыдущем запросе. Отметим у каждой вершины её любимого потомка — дерево окажется представленным виде объединения путей, смотреть Рисунок 11.

Каждому такому пути сопоставим дерево поиска (чтобы за $\log \log$ отправляться в нужное место пути). При смене любимого потомка у вершины нам придётся перестраивать такие деревья. Это мы умеем.

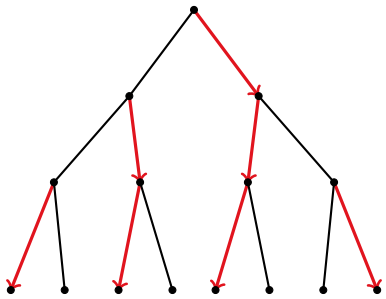


Рис. 11: Tango-дерево представлено в виде объединения путей

6 Link-Cut trees

6.1 Описание структуры, план действий

Наша цель — поддерживать структуру данных, которая умеет хранить лес подвешенных бинарных деревьев и производить с ними следующие операции:

- makeTree(v) — создать дерево из одной вершины v.
- link(v, w) — подвесить u к w (при этом u является корнем одного из деревьев леса, а у w не более одного ребёнка).
- cut(v) — удалить ребро между v и её родителем.
- findRoot(u) — найти корень дерева вершины u.
- findCost(u) — возвращает ближайшее к корню ребро минимального веса на пути от u до корня.
- addCost(u, x) — добавить x к весам всех рёбер на пути от u до корня.

При этом findCost можно адаптировать, чтобы искать не минимум на пути, а, например, сумму и т.д.

В [**sleator1983linkcut**] описано, как добиться асимптотики $\mathcal{O}(\log n)$ на операцию в худшем случае. Мы же изучим link-cut trees, работающие за амортизированное $\mathcal{O}(\log n)$ ([**tarjan1984linkcut**]).

В описании структуры данных и доказательстве времени работы будет две смысловых части.

1) Научиться реализовывать структуру для частного случая дерева — пути. А именно, нам потребуются следующие операции:
   - makePath(v) — создать путь из одной вершины.
   - findPath(v) — вернуть путь, в котором лежит вершина v.
   - findTail(p) — найти верхний конец пути p.
   - join(p, v, q) — объединить пути p и q в один через вершину v, т.е., верхний конец пути p и нижний конец пути q соединить с v.

- split(v) — отрезать ребро, ведущее из v в предка в пути.
- findPathCost(u), addPathCost(u, x).

2) Выразить операции на лесе через операции на путях. Т.е., разобьём вершины дерева на пути. После этого некоторые рёбра лежат на путях (сплошные рёбра), а некоторые соединяют разные пути (пунктирные рёбра). Для операций на дереве нам понадобится также дополнительная функция expose(v), которая превращает путь от v до корня дерева в один из путей разбиения (при этом рёбра, идущие из v вниз, не входят в этот путь).

6.2 Выражение операций на дереве через операции на путях

Мы начнём с того, что выразим операции на дереве (разбитом на пути) через операции на путях и expose(v).

```
1: procedure makeTree(u)
2:     makePath(u)
3: procedure findRoot(u)
4:     findTail(expose(u))
5: procedure findCost(u)
6:     expose(u)
7:     findPathCost(u)
8: procedure addCost(u, x)
9:     expose(u)
10:    addPathCost(u, x)
11: procedure link(u, w)
12:    join(∅, expose(u), expose(w))
13: procedure cut(v)
14:    expose(v)
15:    split(v)
```

Таким образом, expose помогает нам свести задачу на дереве к задаче на пути. Мы считаем, что функция expose возвращает указатель на путь, получившийся в результате её исполнения. Некоторых пояснений требует функция link: здесь мы отождествляем вершину и путь, состоящий только из этой вершины.

Итак, теперь нужно научиться делать expose.

```
1: procedure expose(u)
2:     p := ∅                          ▷ Здесь будем накапливать наш текущий путь
3:     while u ≠ ∅ do
4:         w := successor(findPath(u))  ▷ Запомним следующий сверху путь в дереве
5:         (q, r) := split(u)           ▷ Отрежем у u сплошное ребро вниз
6:         if q ≠ ∅ then                ▷ q — часть пути, проходящего через u, ниже u
7:             successor(q) := u        ▷ Теперь ребро из q в u — пунктирное
8:         p := join(p, u, r)           ▷ А ребро из u в наш текущий путь — сплошное
9:         u := w                       ▷ Перейдём к вершине следующего пути
```

10: successor(p) := ∅

Операцию, которая происходит в теле while, назовём splice.

Theorem 18. *Пусть выполнено m операций с деревом, из них n операций makeTree (т.е., в дереве не более n вершин). Тогда верно следующее:*

1) *Мы произвели $\mathcal{O}(m)$ операций с путями дерева.*

2) *expose был вызван $\mathcal{O}(m)$ раз.*

3) *За все вызовы expose было выполнено $\mathcal{O}(m \log n)$ операций splice.*

Доказательство. Первые два пункта очевидно следуют из того факта, что во всех операциях на дереве expose вызывается константное количество раз. Докажем оценку на количество splice.

Назовём ребро (v, w) тяжёлым, если $2 \cdot \text{size}(v) > \text{size}(w)$, и лёгким, если это неравенство не выполняется. Таким образом, на пути от любой вершины до корня дерева не более логарифма лёгких рёбер.

Мы будем рассматривать следующие величины:

- HS — количество тяжёлых сплошных рёбер в текущий момент времени;
- HSC — сколько раз мы создавали тяжёлые сплошные рёбра к текущему моменту времени.

Каждый splice превращает некоторое пунктирное ребро в сплошное. Будем рассматривать отдельно лёгкие и тяжёлые рёбра. Так как на пути от u до корня не более логарифма лёгких рёбер, то и превратить лёгкое пунктирное в лёгкое сплошное мы могли не более логарифма раз.

Тогда $\#\text{splice} \le m(\log n + 1) + \text{HSC}$.

В конце $\text{HS} \le n - 1$. Значит, почти все создания тяжёлых сплошных рёбер были «отменены», т.е., если мы создавали HSC тяжёлых сплошных рёбер, то по крайней мере $\text{HSC} - n + 1$ раз мы превратили тяжёлое сплошное в тяжёлое пунктирное.

Это могло произойти во время splice, тогда одновременно с этим мы превратили лёгкое пунктирное в лёгкое сплошное. Из этого следует, что $\text{HSC} \le n - 1 + \frac{m}{2}(\log n + 1)$

Итак, мы получили нужную оценку на количество splice. По модулю одной маленькой детали: операции link и cut тоже влияют на наш потенциал HSC.

Во время этих операций лёгкое сплошное ребро могло превратиться в тяжёлое сплошное — такие тяжёлые рёбра можно просто не учитывать в значении HSC.

Также тяжёлое сплошное ребро могло превратиться в лёгкое сплошное. Это соответствует уменьшению потенциала, которое при этом не «уравновешивает» создание этого тяжёлого ребра в какой-то предыдущий момент времени. Однако, так как на любом пути лёгких рёбер не больше логарифма, то на каждую из m операций может произойти не более $\log n$ «незарегистрированных» изменений потенциала.

Суммарно это внесёт в HSC (и нашу итоговую оценку) ещё $\mathcal{O}(m \log n)$ операций.

□

6.3 Операции на путях

Для реализации операций на путях мы будем использовать Splay-дерево. Будем хранить путь в дереве таким образом, чтобы при обходе дерева dfs-ом мы выписывали путь слева направо, заканчивая вершиной tail (таким образом, findTail будет просто возвращать самую правую вершину дерева). В узле дерева будем хранить также следующие величины:

- $\Delta\text{cost}(x) = \text{cost}(x) - \text{mincost}(x)$, где $\text{mincost}(x)$ — это минимальная стоимость вершины в поддереве x.
- $\Delta\text{min}(x) = \text{mincost}(x) - \text{mincost}(p(x))$, а если x — корень дерева, то $\Delta\text{min}(x) = \text{mincost}(x)$

```
 1: procedure makePath(u)
 2:     makeSplayTree(u)
 3: procedure findPath(v)
 4:     splay(v)
 5:     return(v)
 6: procedure findPathCost(v)
 7:     while right(v) ≠ 0 and min(right(v)) = 0 or left(v) ≠ 0 and min(left(v)) = 0 do
 8:         if right(v) ≠ 0 and min(right(v)) = 0 then
 9:             v := right(v)
10:         else
11:             v := left(v)
12:     splay(v)
13:     return(v, Δmin(v))
14: procedure addPathCost(v, x)
15:     Δ(min)(v) = Δ(min)(v) + x
16: procedure join(p, v, q)
17:     v.left = p
18:     v.right = q
19: procedure split(v)
20:     splay(v)
21:     cut(v, v.left)
22:     cut(v, v.right)
```

Для анализа мы воспользуемся уже доказанной асимптотикой splay-дерева. Мы рассмотрим «виртуальное» splay-дерево, которое будет состоять из всех splay-деревьев путей, а также проведённых между путями пунктирными рёбрами.

Потенциалы будут такими же:

$$iw(v) = \begin{cases} \text{size}(v), & \text{если у v два пунктирных ребра} \\ \text{size}(v) - \text{size}(u), & \text{если (u, v) — сплошное ребро} \end{cases}$$

$$tw(v) = \sum_{\text{u — из поддерева v в виртуальном дереве}} iw(u)$$

$$r(v) = \log tw(v)$$

$$\Phi = \sum_{v} r(v)$$

Тогда за одну операцию splay на одном splay-дереве мы платим $3(r(u) - r(v)) + 1$, что даёт амортизированный логарифм, как в анализе асимптотики splay-дерева. Но нам нужно сказать, что на все операции splay во время выполнения одного expose мы суммарно заплатим не более логарифма. Легко видеть, что операция splay не меняет структуры виртуального дерева, а значит, не меняет потенциалы. Таким образом, во время переходов от одного пути к другому во время операции expose все слагаемые $r(v)$, кроме двух, взаимно уничтожатся. Тогда:

$$\text{expose}(v) = 3(r(root) - r(v)) + 2\#\text{splice},$$

что есть $\mathcal{O}(m \log n)$.

7 Динамизация структур данных

Пусть у нас есть задача поиска:

$$Q : X \times 2^D \to A,$$

где A – ответы, D – объекты, X – запросы.

7.1

Definition 9. Говорим, что задача поиска является decomposable, если функция Q обладает следующим свойством:

$$Q(x, D \cup D') = Q(x, D) \blacklozenge Q(x, D'),$$

где $\blacklozenge$ означает, операцию, которая быстро считается.

Example 1. *Пусть мы хотим в каком-то множестве точек узнавать ближайшего соседа от точки запроса. Ясно, что мы можем узнать ближайшего соседа в множестве D, ближайшего соседа в множестве D' и взять из них того, что ближе.*

Предположим, что у нас есть такая функция Q. Давайте сформулируем задачу:

Итак, пусть существует структура данных, которая умеет только хранить и выдавать ответ на запрос, со следующими свойствами:

- В ней n объектов;

- Она занимает $S(n)$ памяти;
- Её можно построить за $P(n)$;
- Она отвечает на вопрос за $Q(n)$.

Problem 1. Мы хотим построить структуру данных, которая обладает следующими свойствами:

- Она занимает $S'(n) = O(S(n))$ памяти;
- Она строится за $P'(n) = O(P(n))$;
- Она отвечает на вопрос за $Q'(n) = O(\log n \cdot Q(n))$;
- Вставка занимает $I'(n) = O\left(\log n \frac{P(n)}{n}\right)$.

В данной задаче имеется ввиду амортизированное время для запроса и вставки.

Итак, давайте строить. Разбиваем наши элементы на на логарифмическое количество уровней. Теперь у нас имеется $\log n$ уровней, которые называются $L_0, \ldots, L_l$:

- L_0: $\varnothing$ или структура с 1 элементом
- L_1: $\varnothing$ или структура с 2 элементами
- ...
- L_i: $\varnothing$ или структура с 2^{i-1} элементами
- ...
- L_l: $\varnothing$ или структура с 2^{l-1} элементами

Запрос делаем следующим образом:

```
Query(x):
  a := E (ответ на ∅)
  for i = 0 to l
    if L_i ≠ ∅
      a := a♦Q(x, L_i)
  return a
```

Ясно, что на запрос мы потратим времени

$$] \sum_{i=0}^{l-1} Q(2^i) < l \cdot Q(n) = O(\log n) Q(n).$$

Remark. Если $Q(n)$ – большое, то есть $Q(n) > n^{\varepsilon}$ при каком-то $\varepsilon > 0$, то время на запросе – это $O(Q(n))$.

Вставку делаем следующим образом:

```
Insert(x):
  Find min k : L_k = ∅
  build L_k := {x} ∪ ⋃_{i<k} L_i
  for i = 0 to k − 1
    destroy L_i
```

Ясно, что build происходит за $P\left(2^k\right)$.

Мы хотим, чтоб цена за одну вставку была

$$I'(n) = O(\log n)\frac{P(n)}{n}.$$

$$I'(n) = O(\log n)\frac{P(n)}{n} = \sum_{i=0}^{\log n} \frac{P(2^i)}{2^i}.$$

Ясно, что за n вставок у нас получится

$$\sum_{i=0}^{\log n} P\left(2^i\right)\frac{n}{2^i} = n\sum_{i=0}^{\log n}\frac{P(2^i)}{2^i} = O(\log n)P(n).$$

Что и требовалось.

Remark. Если $P(n) > n^{1+\varepsilon}$, где $\varepsilon > 0$, то $I'(n) = O\left(\frac{P(n)}{n}\right)$.

Problem 2. Мы хотим получить тот же результат, что и в Задаче 1, только время для запроса и вставку теперь не амортизированное, а в худшем случае.

Теперь на каждом уровне у нас будут старые структуры и новые. То есть, на уровне i будут находиться структуры O_i^1, O_i^2, O_i^3, N_i, снова в каждой из них $\varnothing$ или 2^i объектов. Также ещё мы хотим, чтоб выполнялось:

- если $O_i^1 = \varnothing$, то $O_i^2 = \varnothing$
- если $O_i^2 = \varnothing$, то $O_i^3 = \varnothing$

От N_i хотим, чтоб $N_i = \varnothing$ или N_i было частичной (то есть той, которая находится в процессе построения) структура для 2^i объектов. Также нам надо, чтоб каждый объект был ровно в одной структуре O и, может быть, в структуре N.

Запрос делаем следующим образом:

```
Query(x):
  a_i = E
  for i = 0 to l
    if Q_i^1 ≠ ∅
      a_i = a♦Query(x, Q_i^1)
    if Q_i^2 ≠ ∅
      a_i = a♦Query(x, Q_i^2)
    if Q_i^3 ≠ ∅
      a_i = a♦Query(x, Q_i^3)
  return a
```

Очевидно, что время запроса у нас такое же.

Вставку делаем следующим образом:

```
Insert(x):
  for i = l ... 1
    if O^1_{i-1} ≠ ∅ and O^2_{i-1} ≠ ∅
      do P(2^i)/2^i шагов построения N_i(:= O^1_{i-1} ∪ O^2_{i-1})
      if N_i is complete
        destroy O^1_{i-1}, O^2_{i-1}
        O^1_{i-1} := O^3_{i-1}
        destroy O^3_{i-1}
        Update(i)
      N_0 := x
      Update(0)
```

Что такое Update(i)?

```
Update(i):
  if O^1_i = ∅
    O^1_i := N_i
  else if O^2_i = ∅
    O^2_i := N_i
  else O^3_i := N_i
  destroy N_i
```

Все ошки не могут быть заняты, это доказывается по индукции, но это было оставлено в качестве упражнения.

Идея в том, что мы не делаем построение сразу, а делаем столько его шагов, сколько можем себе позволить. А значит, время вставки даже в худшем случае будет $O\left(\frac{P(n)}{n}\log n\right)$.

7.2

Definition 10. Говорим, что задача поиска является invertible, если функция Q обладает следующим свойством: Если $D = D_1 \cup D_2$, то

$$Q(x, D_1) = Q(x, D) - \blacklozenge Q(x, D_2),$$

где $-\blacklozenge$ быстро считается.

Предположим, что у нас есть такая функция Q. Давайте сформулируем задачу:

Problem 3. Пусть у нас есть структура данных M, которая умеет вставлять и ещё другая структура данных G, в которую мы будем вставлять элемент, который хотим удалить из M. Хотим, чтоб амортизированное время удаления из M равнялось $O\left(P(n)\frac{\log n}{n}\right)$.

```
Query(x):
  Return Q(x, M) − ♦Q(x, G)
```

Подвох заключается в том, что у нас может быть много удалённых элементов, поэтому весь наш анализ будет от количества элементов, которые когда-либо вставлялись. А мы хотим не этого.

Мы будем ждать момента, когда $|G| > \frac{1}{2}|M|$, и в это время перестраивать структуру полностью. Идея в том, что между двумя такими моментами пройдёт минимум $\frac{n}{2}$ удалений, а значит, когда мы считаем амортизированную стоимость удаления, мы можем заключить, что цена каждого удаления – это $O\left(\frac{P(n)}{n}\right) + O\left(P(n)\frac{logn}{n}\right)$ (второе слагаемое – это вставки в структуру G).

When $|G| > \frac{1}{2}|M|$
  Build $M := M \backslash G$
    $G := \varnothing$

Также есть проблема, что глобальные перестроения могут помешать локальным, то есть для работы вставок мы тоже перестраиваем систему, и надо, чтоб нам хватило ресурсов для вставки, несмотря на то, что мы потратили что-то на удаление. Для борьбы с этим достаточно просто амортизированную стоимость удаления умножить на достаточно большую константу.

Problem 4. Мы хотим получить тот же результат, что и в задаче 3, только время теперь не амортизированное, а в худшем случае.

Для решения Задачи 4 мы поддерживаем три структуры: M, I, G:

Query(x):
  Return $Q(x) = Q(x, M)\blacklozenge Q(x, I) - \blacklozenge Q(x, G)$

В какой-то момент снова перестраиваем структуру, а именно в случае, если $|G| > \frac{1}{2}(|M| + |I|)$. Чтоб удовлетворять запросам, придётся заморозить наши структуры. Поэтому также поддерживаем ещё три структуры: M', I', G'.

Во время работы по перестроению структуры, мы будем временно объекты, которые мы хотим удалить, вставлять в G', которые хотим вставить, вставляем в I'. M' – это, собственно, структура, которую мы строим. Пока мы строим:

Query(x):
  Return $Q(x) = Q(x, M)\blacklozenge Q(x, I)\blacklozenge Q(x, I') - \blacklozenge Q(x, G) - \blacklozenge Q(x, G')$

Когда мы делаем каждое удаления, мы будем делать $c\frac{P(n)}{n}$ (для какой-то константы c) шагов построения структуры M'.

Через $\frac{n}{c}$ шагов
$M := M', I := I', G := G'$

Ясно, что время в худшем случае будет каким надо, а именно,

$$O\left(P(n)\frac{\log n}{n}\right).$$

7.3

Наши запросы не всегда бывают invertible. Далее будет идея того, что нужно делать с теми запросами, которые не invertible.

Нам всё равно потребуется какое-нибудь условие, а именно, weak deletion in time $D(n)$. Это какая-то операция, которая даст структуре данных понять, что этого элемента не должно в ней быть, и при этом не увеличит время на запрос.

Делаем то же самое для вставок – поддерживаем уровни.

Когда нам нужно удалить x: сделаем weak deletion x в каждом уровне, содержащем x (+ чтоб найти эти уровни, нам нужен какой-то словарик, который по элементу называет уровни, где он содержится).

Делаем те же глобальные перестройки, то есть когда не удаленных объектов $> \frac{1}{2}$ удалённых – делаем global rebuild.

Из прошлого рассуждения знаем, что

- амортизированное время вставки – это $O\left(P(n)\frac{\log n}{n} + D(n)\right)$;
- амортизированное время удаления – это $O\left(\frac{P(n)}{n}\right)$.

Это также можно несколькими структурами сделать в худшем случае. А именно, структурами структурами: M, S, M', S', u. M – главная, S её дублирует. Когда у нас удалений становится больше, чем половина тех элементов, которые лежат в M, снова начинаем глобальное перестроение. Для этого мы заморозим структуру S, для новых запросов заведём u – очередь запросов, которую будем применять к M'. M' – это рабочая копия M на время перестроения. Когда мы закончили, у нас M' будет на правах M и S' на правах S. После этого надо сделать все обновления в очереди u. Нужно просто подобрать константы, сколько шагов построения S' и M' мы будем выполнять каждый раз, когда делаем удаление.

Список литературы

[And89] Arne Andersson. «Improving partial rebuilding by using simple balance criteria». В: *Workshop on Algorithms and Data Structures*. Springer. 1989, с. 393—402.

[GR93] Igal Galperin и Ronald L Rivest. «Scapegoat Trees.» В: *SODA*. Т. 93. 1993, с. 165—174.

[ST85] Daniel Dominic Sleator и Robert Endre Tarjan. «Self-Adjusting Binary Search Trees». В: *J. ACM* 32.3 (июль 1985), с. 652—686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.