

## 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  $\pm$ 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**  $\alpha$ -WEIGHT(node  $x$ )

$\max\{\text{size}(x.\text{left}), \text{size}(x.\text{right})\} \leq \alpha \cdot \text{size}(x)$

**condition**  $\alpha$ -HEIGHT(node  $x$ )

$\text{depth}(x) \leq \lfloor \log \text{size} \rfloor + 1$

**condition** WEAK  $\alpha$ -HEIGHT(node  $x$ )

$\text{depth}(x) \leq \lfloor \log \text{maxSize} \rfloor + 1$

▷ 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  $\alpha$  — это, грубо говоря, разрешённое отклонение размера поддеревьев от состояния баланса.

Узел называется *глубоким*, если он нарушает weak  $\alpha$ -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  $\alpha$ -weight balanced, то оно и  $\alpha$ -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:*  $\alpha \cdot \text{maxSize} \leq \text{size} \leq \text{maxSize}$

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,

$$\text{maxSize} := \max\{\text{maxSize}, \text{size}\}.$$

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

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

□

**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 появиться 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.  $\square$

Корректность мы показали, но у нас остались операции перестройки, которые работают в худшем случае за линейно. 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 сбалансированным 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 и 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

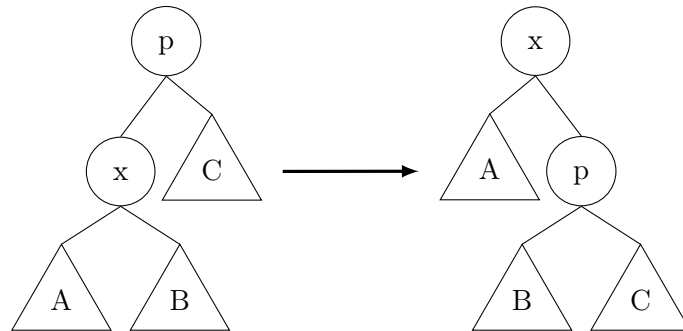
▷ optional

**structure** NODE

left, right

key

Рис. 1: Zig



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.

Объяснение, почему можно забить на то, что множество ключей в дереве меняется, я допишу позже.

todo 1

## 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 zig-zig, надо сначала выполнить zig от вершины  $p$ , и только потом от  $x$ . Zig-zag при

Рис. 2: Zig-zig

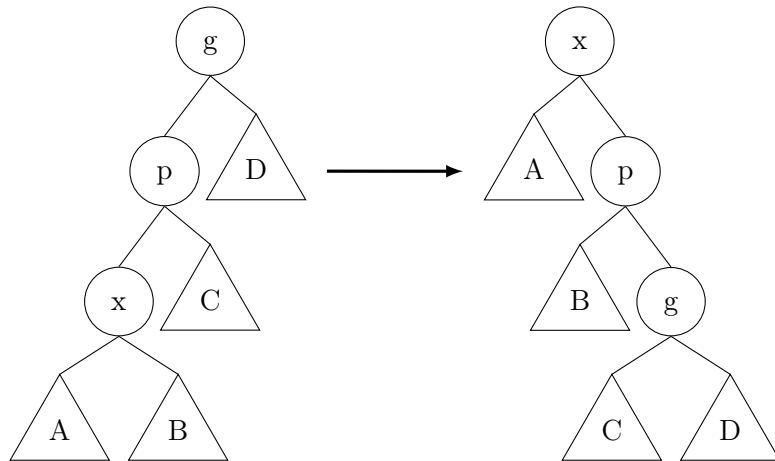
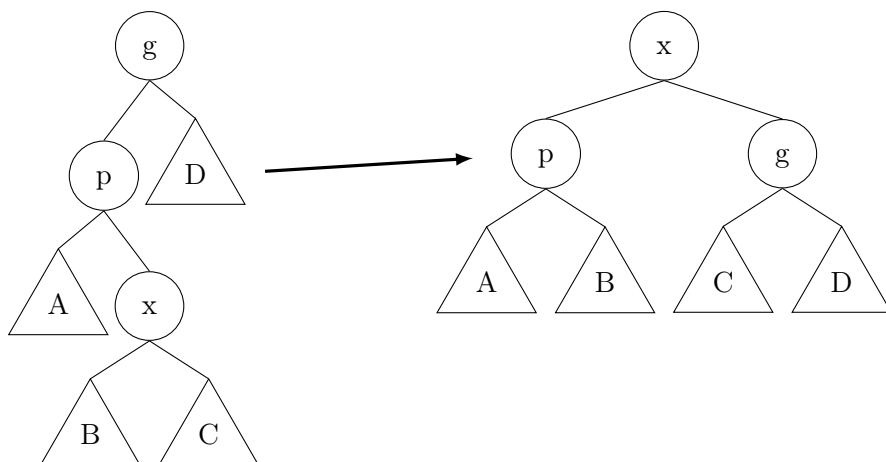


Рис. 3: Zig-zag



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  $w : \text{keys} \rightarrow \mathbb{R}_+$ . 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  $\Phi$  всегда было неотрицательно (ну или придется 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`  $\text{am.cost} = \Delta\Phi + \#\text{rotations}$  (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.**  $\text{am.cost} \leq 3(r'(x) - r(x)) + \mathcal{O}(1)$

*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  $r'(g) + r(x) \leq 2(r'(x) - 1)$ .

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

*Zig-zag:*

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

Доказательство неравенства  $r'(g) + r'(p) \leq 2(r'(x) - 1)$  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  $\Delta\Phi \leq 3(r'(x) - r(x)) - \#rotations + \mathcal{O}(1)$ , 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,  $\text{am.cost} = \Delta\Phi + \#rotations \leq 3(r'(x) - r(x)) + \mathcal{O}(1)$ , что нам и надо.  $\square$

Ниже мы будем считать, что наше дерево работает с ключами  $1 \dots n$ , выполняет  $m$  операций, а  $W := \sum_i w(i)$ . 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  $\Phi \geq 0$ . При  $w \geq 1$  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  $w(x) = 1$ .  $\square$

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

*Доказательство.* Берем  $w(x) = q_x$ .  $\square$

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  $\mathcal{O}\left(m + \sum_x \text{запрос} \log(|x - f| + 1)\right)$ .*

*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.  $\square$

**Theorem 8** (Scanning Theorem or Sequential Access Theorem or Queue theorem). *Доступ к элементам в порядке возрастания работает за амортизированную единицу на запрос.*

Не помню доказывали ли мы теорему выше, но точно упомянули.

## 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>.