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]

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.