МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Брянский государственный технический университет

Утверждаю
Ректор университета

_____О.Н. Федонин

«____»_____2017 г.

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

СБАЛАНСИРОВАННЫЕ БИНАРНЫЕ ДЕРЕВЬЯ

Методические указания
к выполнению лабораторной работы №5
для студентов очной, очно-заочной и заочной форм обучения
по направлениям подготовки
230100 «Информатика и вычислительная техника»,
010500 «Математическое обеспечение и администрирование информационных систем»,
231000 «Программная инженерия»

Брянск 2017

СОДЕРЖАНИЕ

Визуализаторы https://people.ksp.sk/~kuko/gnarley-trees/

УДК 006.91

Структуры и алгоритмы обработки данных. Сбалансированные деревья [Текст] + [Электронный ресурс]: методические указания к выполнению лабораторной работы №5 для студентов очной, очно-заочной и заочной форм обучения по направлениям подготовки 230100 «Информатика и вычислительная техника», 010500 «Математическое обеспечение и администрирование информационных систем», 231000 «Программная инженерия». – Брянск: БГТУ, 2014. –23 с.

Разработал:
канд. техн. наук, проф.
В.К. Гулаков

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №1 от 13.09.13)

1. ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

Целью лабораторной работы является приобретение навыков построения сбалансированных деревьев.

Продолжительность лабораторной работы – 4 часа.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Общие положения по сбалансированным бинарным деревьям

В худшем случае бинарное дерево может выродиться в линейный список. В этом случае хранение данных в упорядоченном бинарном дереве никакого преимущества в сложности операций по сравнению с массивом или линейным списком не дает.

В лучшем случае, мы имеем идеально сбалансированное дерево, когда для всех операций получается логарифмическая сложность, что значительно лучше.

Однако, при выполнении различных операций редактирования такого дерева показатели балансировки нарушаются и требуется восстановление баланса. Это не только усложняет обработку данных, но и требует дополнительную память и время. Чтобы уйти от этих недостатков и иметь дерево почти минимальной высоты, были предложены почти идеально сбалансированные деревья. В настоящее время наиболее известны следующие разновидности сбалансированных деревьев:

АВЛ-деревья;

сбалансированные по весу деревья (ВВ-деревья);

Красно-черные деревья. Деревья промежутков;

Расширяющиеся деревья – самонастраивающиеся структуры данных;

Выровненные деревья.

К сбалансированным можно также отнести большую группу сильноветвящихся деревьев (2-3 деревья, 2-3-4 деревья, B- дерево и его разновидности: B+ дерево, B++ дерево, B* дерево, UB – дерево,

(a, b) дерево, и др.), но учитывая, что здесь рассматриваются только бинарные сбалансированные деревья, они здесь не приводятся.

2.2. Идеально сбалансированные и оптимальные бинарные деревья

В общем случае бинарное дерево называется идеально сбалансированным деревом, если все листья находятся на уровне h, либо на уровне h-1, где h – минимальная высота дерева и равна $\lceil \log_2 n+1 \rceil$, т.е. почти полное бинарное дерево.

Обозначение $\lceil x \rceil$ соответствует минимальному целому числу, большему x («потолок» x), а $\lfloor x \rfloor$ — максимальному целому числу, меньшему x («пол» x) (по К. Е. Иверсону).

Полностью сбалансированное бинарное дерево имеет минимальную высоту. Когда все ключи (аргументы) поиска равновероятны (например, цифры в различных числах), полностью сбалансированное бинарное дерево представляет собой *оптимальное бинарное дерево поиска*.

Однако нередки случаи, когда частоты появления ключей не равны между собой (например, символы алфавита), и при этом заранее известны. В этом случае полностью сбалансированное бинарное дерево не оптимально

Как правило, идеально сбалансированные деревья применяются, когда множество данных практически не меняется.

2.3. Сбалансированные по высоте АВЛ-деревья

Однако идеальную сбалансированность трудно поддерживать. В некоторых случаях при добавлении или удалении элементов может потребоваться значительная перестройка дерева, не гарантирующая логарифмической сложности операций над ним. В 1962 году два советских математика: Г.М. Адельсон-Вельский и Е.М. Ландис – ввели менее строгое определение сбалансированности и доказали, что при таком определении можно написать программы добавления и/или удаления узлов, имеющие логарифмическую сложность и сохраняющие дерево сбалансированным.

Дерево считается ***сбалансированным по АВЛ*** (сокращения от фамилий Г.М. Адельсон-Вельский и Е.М. Ландис), если для каждой вершины выполняется требование: высота левого и правого

поддеревьев различаются не более, чем на 1, *т*.е. используется менее строгий критерий сбалансированности.

(показатель сбалансированности узла) = (высота правого поддерева этого узла)—(высота левого поддерева этого узла). В АВЛ-дереве показатель сбалансированности всех узлов равен или -1, или 0, или +1.

Если для всех узлов, за исключением листьев, показатель сбалансированности равен +1 или –1, то такое дерево называется *деревом Фибоначчи*.

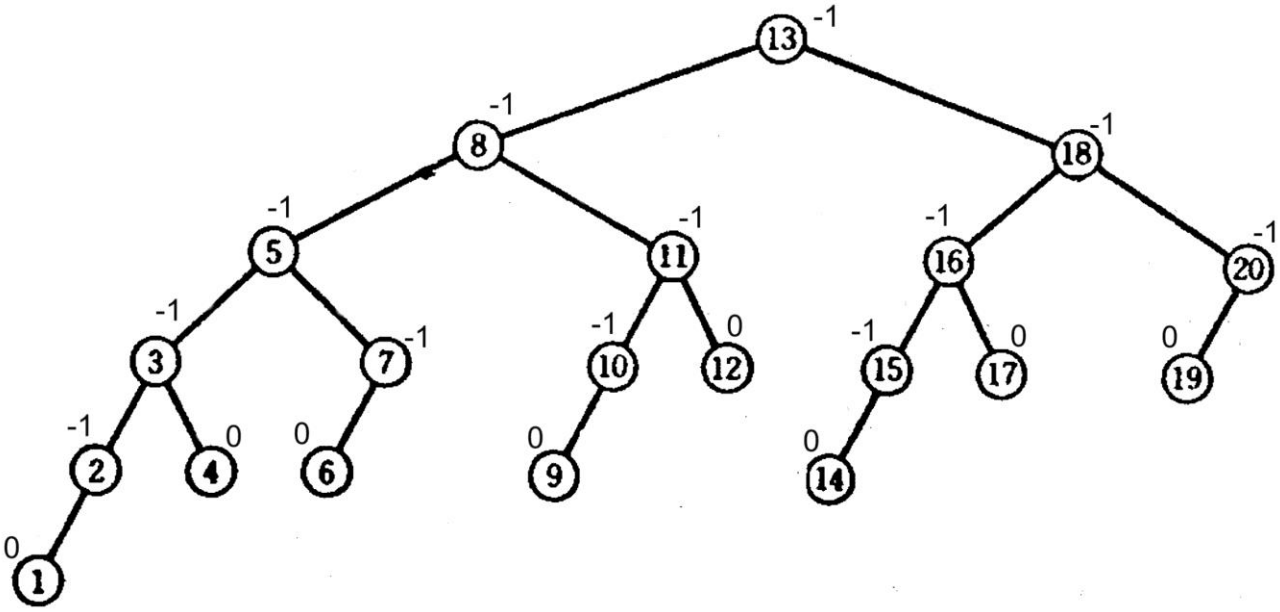


Рис. 1. *Дерево Фибоначчи*

Между полностью сбалансированными деревьями и деревьями Фибоначчи располагаются самые разные сбалансированные АВЛ-деревья высотой $1{,}4404 \log_2(n+2)-0{,}328 > h > \log_2(n+1)$.

АВЛ-дерево представляет собой структуру, для которой любая операция поиск, вставка и удаление ключа имеет временную сложность $O(\log_2 n)$.

При операциях добавления и удаления может произойти нарушение сбалансированности дерева. В этом случае потребуются некоторые преобразования, не нарушающие упорядоченности дерева и способствующие лучшей сбалансированности.

Рассмотрим такие преобразования.

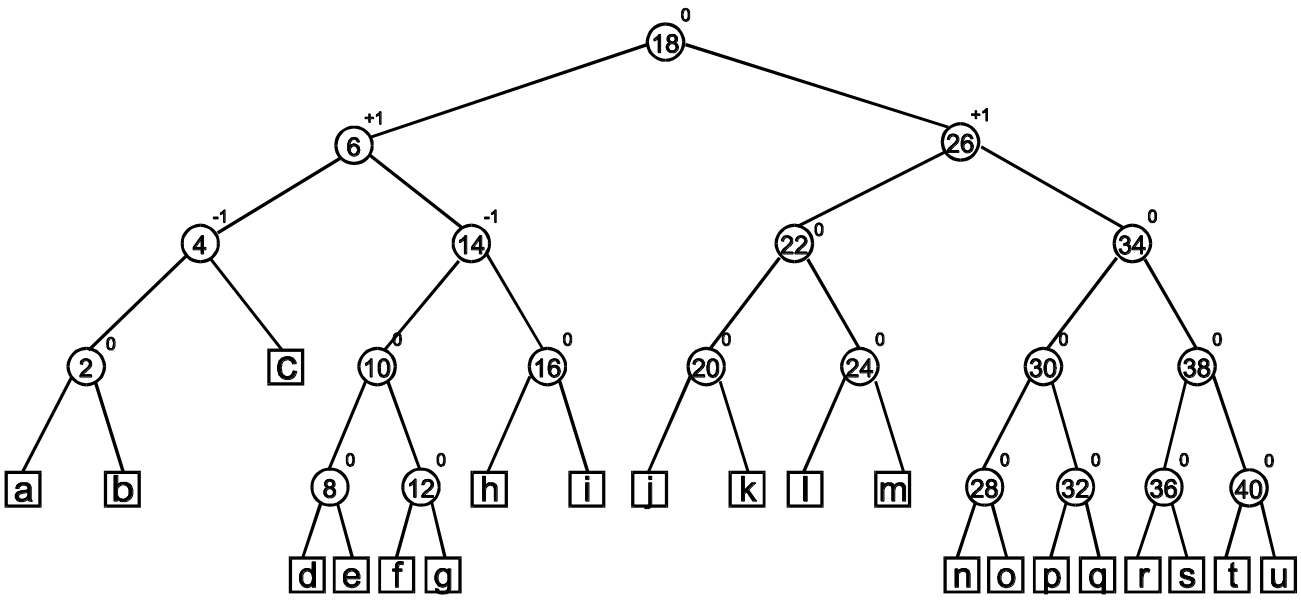


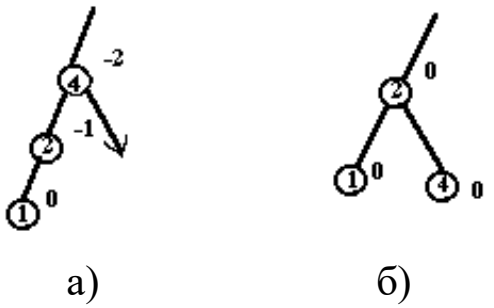
Рис. 2. Пример АВЛ-дерева.

Вставка ключей в АВЛ-дерево

Вставляемый ключ займет место одного из внешних узлов дерева a, b, ..., u,. При этом могут наблюдаться следующие случаи:

– Случай c. Показатель сбалансированности улучшается.
– Случаи h, i, j, k, l, m.. Показатель сбалансированности ухудшается, но свойства АВЛ-дерева сохраняются.
– Все остальные случаи, за исключением указанных в 1 и 2, нарушают свойства АВЛ-дерева. Появляются узлы с показателем сбалансированности +2 или -2, и для сохранения свойства сбалансированности АВЛ-дерева потребуется некоторая корректировка структуры дерева.

Рассмотрим несколько примеров. При вставке узла с ключом 1 (Рис. 3а) показатель сбалансированности узла “4” окажется равным -2, и условия АВЛ-дерева перестанут удовлетворяться.

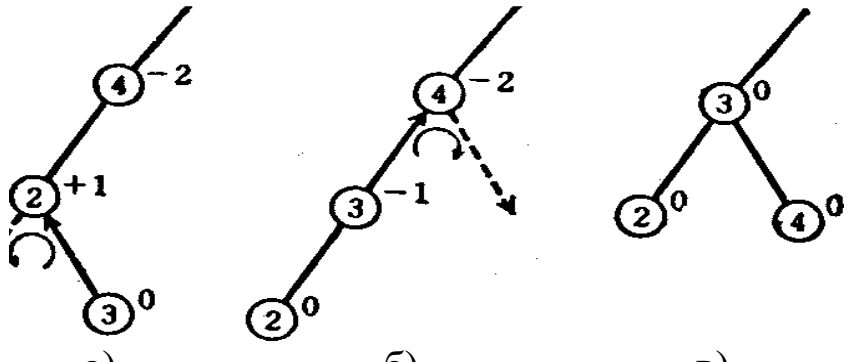


а) б)

Рис. 3. Вставка ключ со значением “1”

Вместо узла “4” корнем этого поддерева можно сделать узел “2”, совершив поворот узлов “2” и “4” направо (Рис. 3б).

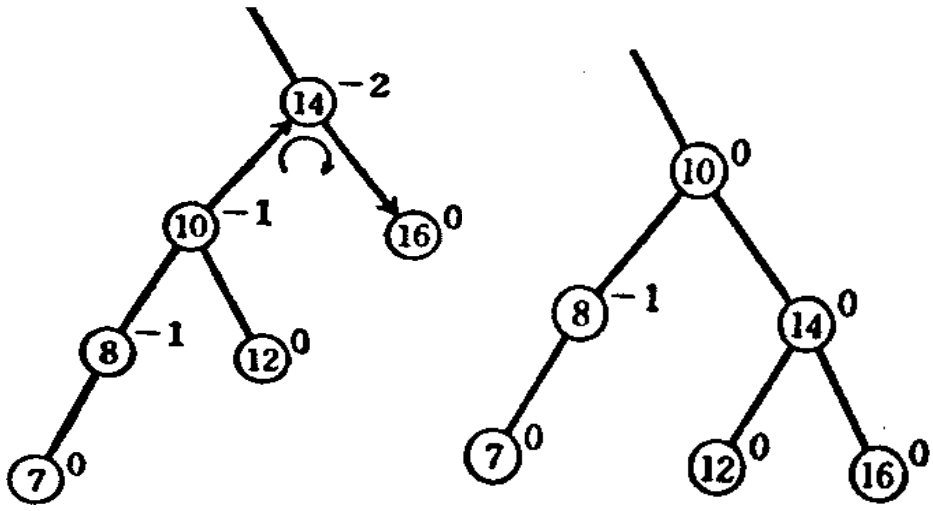
При вставке узла с ключом 3 (Рис. 4а) восстановить баланс будет несколько сложнее. Вначале повернем налево узлы “2” и “3”, как показано на рис. 4(а) и получим структуру, показанную на рис. 4(б), которая аналогична структуре на рис. 3(а). Теперь, если узлы “3” и “4” повернуть направо, то участок дерева окажется сбалансированным, как показано на рис. 4(в)



а) б) в)

Рис. 4. Вставка ключа со значением “3”

В случае вставке узла с ключом 7 (Рис. 5а) вблизи вставленного узла сбалансированность сохраняется. Однако результат вставки сказывается выше—для узла “14”, показатель сбалансированности для которого становится равным -2. Поскольку при этом узел “14” находится точно в таком же положении по отношению к узлу “10”, как и узел “4” по отношению к узлу “2” на предыдущем рис. 4а, то, если совершить аналогичный поворот направо, получим сбалансированный участок, как показано на рис. 5б).



а) б)

Рис. 5. Пример вставки ключа со значением “7”

Однако при этом кроме поворота, показанного на рис. 5б), потребуется узел “12” открепить от узла “10” и прикрепить к узлу “14”.

Вставка узла с ключом “9” приводит почти к такому же эффекту, как и вставка ключа “7”.

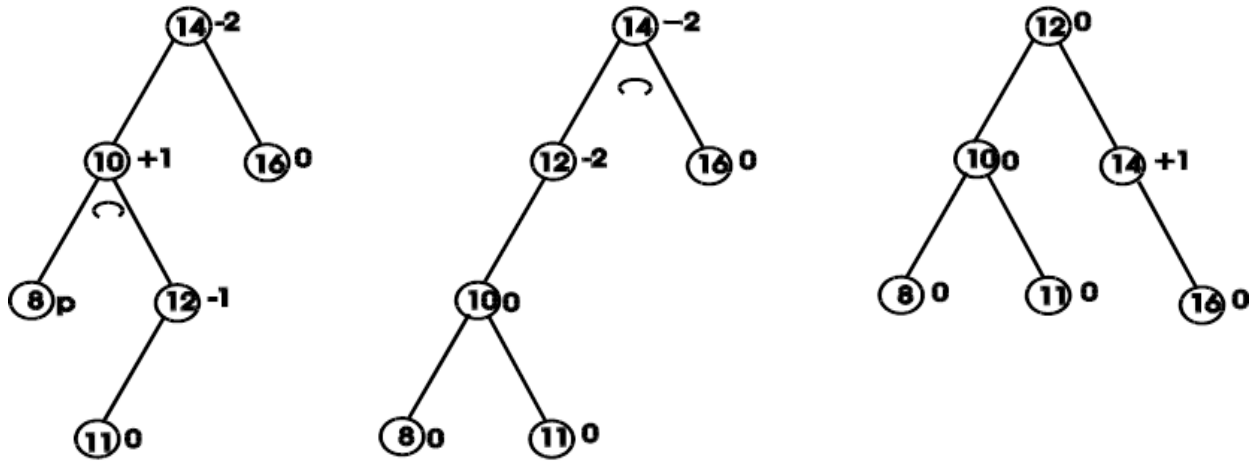


Рис. 6. Пример - вставка ключа “11”

Последовательность действий при вставке ключа “11” в дерево показана на рис.6 и, по существу, аналогична действиям, показанным на рис. 4 при вставке ключа 3; только в этом случае также при повороте узел “11” необходимо открепить от узла 12” и прикрепить к узлу “10”.

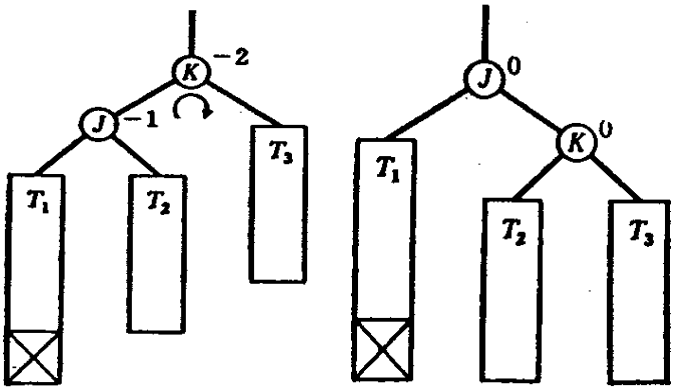


Рис. 7. Обобщение рассмотренных случаев: случай одного поворота

Если после вставки показатели сбалансированности узлов имеют одинаковый знак и отличаются только на единицу, как для узлов “К” и “J” на рис. 7, то восстановить баланс дерева можно однократным поворотом (включая одно переподкрепление поддерева), при этом вставка не будет оказывать влияния на другие участки дерева.

Если же после вставки показатели сбалансированности имеют разный знак, например как для узлов “К ” и “J” на рис. 8 (т.е. их разница по абсолютной величине равна 3), то можно восстановить баланс дерева двукратными поворотами трех узлов, включая узел “J” (используя два переподкрепления поддеревьев).

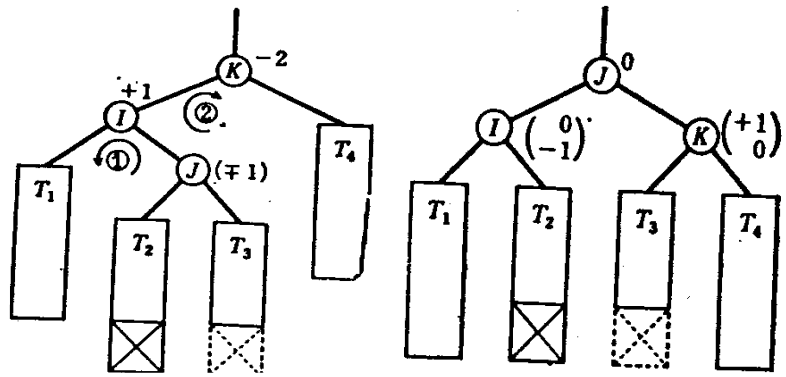


Рис. 8. Обобщение рассмотренных случаев: случай двух поворотов

Укрупненный алгоритм вставки ключа в АВЛ-дерево и балансировки его.

1. Ищем место вставки ключа;
2. Вставляем ключ путем изменения указателей у предка;
3. Проверяем балансировку узлов (при сбалансированных узлах – выход);
4. При разбалансировке узлов выявляем необходимость:
   – одного вращения, выполняем эту операцию и переходим к шагу 3;
   – двух вращений, выполняем эти операции и переходим к шагу 3;

Третий шаг представляет собой обратный проход по пути поиска: от места добавления к корню дерева. При продвижении по этому пути корректируются показатели сбалансированности проходимых вершин, и производится балансировка там, где это необходимо.

Сочетание некоторых теоретических рассуждений и эмпирических результатов позволяет сделать следующие утверждения:

Математическое ожидание значения высоты при больших n близко к значению $\log_2 n + 0{,}25$.

Вероятность того, что при вставке не потребуется дополнительная балансировка, потребуется однократный поворот или двукратный поворот, близка к значениям 2/3, 1/6 и 1/6 соответственно.

Среднее число сравнений при вставке n–го ключа в дерево выражается формулой $a \log_2 n + b$ (a, b – константы)

Алгоритм удаления элемента из сбалансированного АВЛ-дерева будет выглядеть так:

Шаг 1. Поиск по дереву.

Шаг 2. Удаление элемента из дерева.

Шаг 3. Восстановление сбалансированности дерева (обратный проход).

Первый шаг необходим, чтобы найти в дереве вершину, которая должна быть удалена. Третий шаг представляет собой обратный проход от места, из которого взят элемент для замены удаляемого, или от места, из которого удален элемент, если в замене не было необходимости. Операция удаления может потребовать балансировки всех вершин вдоль обратного пути к корню дерева, т.е. порядка $log\ n$ вершин. Таким образом, алгоритмы поиска, вставки и удаления элементов в сбалансированном АВЛ дереве имеют сложность, пропорциональную $O(log\ n)$.

2.4. Красно-черные деревья и деревья почти оптимальные по высоте 89. Нисходящая балансировка для красно-черных деревьев

Бинарные деревья работают лучше всего, когда они сбалансированы, когда длина пути от корня до любого из листьев находится в определенных пределах, связанных с числом вершин. Красно-черные деревья являются одним из способов балансировки деревьев. Название происходит от стандартной раскраски узлов таких деревьев в красный и черный цвета. Цвета вершин используются при балансировке дерева.

Красно-черное дерево (Red-Black-Tree, RB-Tree) – это бинарное дерево со следующими свойствами (рис. 3):

- каждая вершина должна быть окрашена либо в черный, либо в красный цвет;
- корень дерева должен быть черным;
- листья дерева должны быть черными и объявляться как NIL – вершины (NIL – узлы, то есть «виртуальные» узлы, наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели);
- каждый красный узел должен иметь черного предка;
- на всех ветвях дерева, ведущих от его корня к листьям, число черных вершин одинаково.

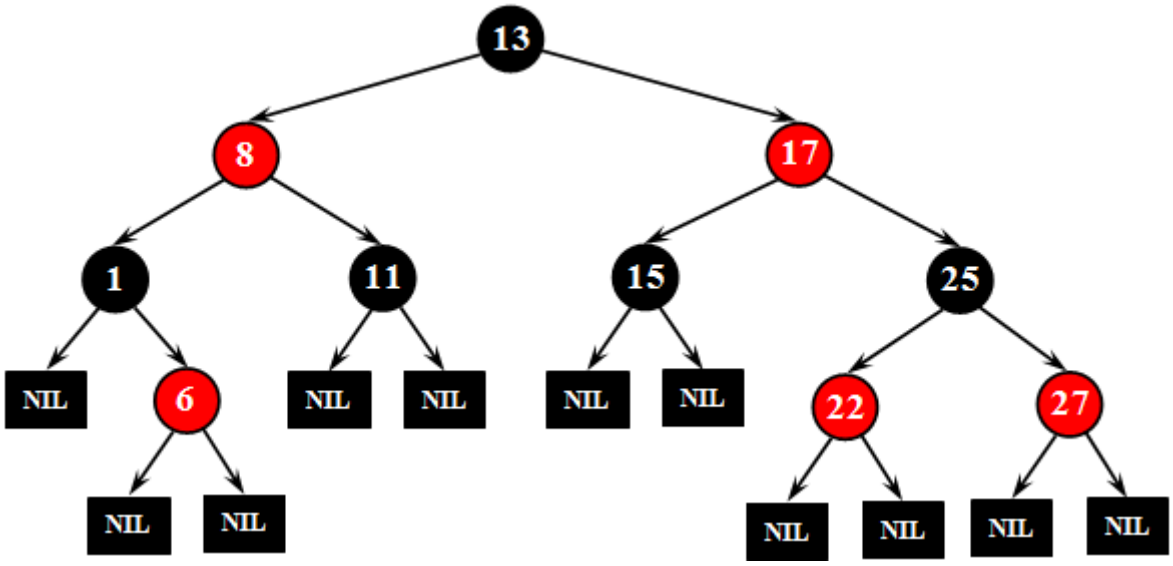


Рис. 9. Красно-черное дерево

Число черных вершин на ветви от корня до листа называется черной высотой дерева. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу.

Над красно-черными деревьями можно выполнять все те же основные операции: вставка элемента, создание дерева, печать (просмотр) дерева, обход дерева, проверка пустоты дерева и удаление дерева, что и над бинарными деревьями.

Красно-черные деревья (файл Сигаев Курс.pdf)

Красно-черные деревья-формулировку дали Гиубас и Седжвик (1978) (L.J. Guibas, R. Sedgewick: A Dichromatic Framework for Balanced Trees, in: FOCS 1978 (Proceedings 19th Annual IEEE Symposium on Foundations of Computer Science), 8–21.), которые предложили красно-чёрное дерево имеет высоту до 2log n + 1. И мы имеем алгоритм балансировки со сложностью O(log n) в худшем случае, которые меняет только узлы количеством, по амортизированной оценке равным O(1). Единственный недостаток у нашей прежней структуры – то, что этот алгоритм балансировки вместо поворотов использует более сложные операции расщепления, разделения и объединения. Но возможен также алгоритм, основанный на поворотах, с правилами, которые мы опишем позже.

Красно-чёрное дерево с цветами узлов

Другие эквивалентные версии данной структуры – полусбалансированные деревья Оливье (1982) H.J. Olivi´e: A New Class of Balanced

Search Trees: Half-Balanced Binary Search Trees, *RAIRO Informatique Th´eorique* **16** (1982) 51–71.**,** характеризующиеся тем свойством, что наиболее длинный путь к листа длиннее наиболее короткого максимум вдвое, чья эквивалентность красно-чёрным деревьям была замечена Тарьяном (1983a) R.E. Tarjan: Updating a Balanced Search Tree in *O*(1) Rotations, *Information Processing Letters* **16** (1983a) 253–257. и стандартные деревья сыновей, предложенные Оттманном и Сиксом (1976) T. Ottmann, H.-W. Six, D. Wood: Right Brother Trees, *Communications ACM* **21**(1978) 769–776 и Оливье (1980) H.J. Olivi´e: On the Relationship between Son-Trees and Symmetric Binary B-Trees, *Information Processing Letter* **10** (1980) 4–8., которые являются узлами с унарными и бинарными узлами, где все листья имеют одинаковую глубину, и где нет унарных узлов на чётных уровнях. Несколько альтернативных алгоритмов балансировки для этих структур предложили Тарьян (1983a) R.E. Tarjan: Updating a Balanced Search Tree in *O*(1) Rotations, *Information Processing Letters* **16** (1983a) 253–257.**,** Зивани, Оливье и Гоннет (1985) N. Zivani, H.J. Olivi´e, G.H. Gonnet: The Analysis of an Improved Symmetric Binary B-Tree Algorithm, *The Computer Journal* **28** (1985) 417–425**,** Чен и Чотт (1996) H. Chang, S.S. Iyengar: Efficient Algorithms to Globally Balance a Binary Search Tree, *Communications ACM* **27** (1984) 695–702**.** Гиубас и Седжвик (1978))(L.J. Guibas, R. Sedgewick: A Dichromatic Framework for Balanced Trees, in: FOCS 1978 (Proceedings 19th Annual IEEE Symposium on Foundations of Computer Science), 8–21.) также заметим, что несколько других схем балансировки могут быть выражены как разметки цветов вершин, ассоциированные с конкретными действиями по балансировке. Для деревьев, сбалансированных по высоте, уже давно было известно, что им не нужно хранить высоту в каждом узле, нужна только информация, имеют ли оба поддерева равную высота или же высота левого либо правого поддерева меньше на единицу.

Изначально такой подход замышлялся как экономичное по памяти кодирование, но он дал сбалансированные по высоте деревья, также в рамках подхода, основанного на раскраске узлов. В случае сбалансированного по высоте дерева если каждый узел нечётной высоты, чей вышележащий сосед имеет чётную высоту, пометить как красный, а все остальные узлы – как чёрные, то выполняются условия красно-чёрного дерева. Но не все красно-чёрные деревья сбалансированы по высоте; требуется дополнительное ограничение, состоящее в следующем: если узел чёрный и оба его нижележащих соседа чёрные, тогда как минимум один из их нижележащих соседей должен быть красным. За счёт этих условий возможно восстановить баланс по высоте для узла с использованием цветов нижележащих соседей и их нижележащих

соседей, а используя всю эту информацию, можно восстановить баланс по высоте для всего дерева. Гуибас и Седжвик (1978)(L.J. Guibas, R. Sedgewick: A Dichromatic Framework for Balanced Trees, in: FOCS 1978 (Proceedings 19th Annual IEEE Symposium on Foundations of Computer Science), 8–21.) предложили несколько других схем балансировки, основанных на красно-чёрных разметках вершин, среди них наиболее интересны
методы нисходящей балансировки, которые могут быть выполнена уже в процессе движения от корня к листу, устраняя необходимость второго прохождения – назад к корню.
Другая разработка, выполненная на основе реинтерпретации (a, b)-деревьев применительно к оперативной памяти, - это деревья малой высоты. В главе 2 мы видели, что высота бинарного дерева поиска с n листьями равна как минимум , и мы можем добиться верхней оценки высота 1.44log n, используя деревья, сбалансированные по высоте. Граничные оценки для сбалансированных по весу деревьев и красно-чёрных деревьев несколько хуже – 2log n для красно-чёрного дерева и как минимум 2log n (зависит от выбора параметра) для сбалансированных по весу деревьев. Это вызывает вопрос, можем ли мы сделать высоту лучше, чем 1.44log n, в то же время сохранив время обновления O(log n). Без этого мы можем только перестраивать оптимальное дерево после каждого обновления. Первым подходом, который достиг для любых (алгоритмы, зависящие от k), были k-деревья Маурера, Оттманна и соавторов (1990). Они просто берут (2k, 2k+1)-деревья в качестве исходной структуры и заменяют
каждый из узлов высокой степени небольшим деревом поиска оптимальной высота (которой является высота k+1). Для исходных деревьев у нас снова общий алгоритм балансировки (a, b)-деревьев, использующий операции расщепления, объединения и разделения, а во внедряемых бинарных деревьях эти преобразования могут быть воспроизведены на основе поворотов, поскольку, как мы показали в разделе 2.2, любое преобразование дерева поиска на одном и том же множестве листьев может быть реализовано поворотами. Поэтому данная структура дерева поиска имеет высоту как максимум, при фиксированном k балансировка выполняется за время O(log n), используя O(1) поворотов согласно амортизированной
оценке. Выбирая k = log log n, они добились дальнейшего снижения высоты – до (1+ o(1))log2n, а Андерсон и Лаи в своих диссертациях и ряде статьей с разными соавторами показали дальнейшее снижение,

сократив показатель o(log n). Последнее слова, казалось бы, означают, что высота не может поддерживаться за счёт перебалансировки сложностью o(n), потому что при n=2k, уникальные деревья поиска высотой k для [1,…,n] и [2,…n+1] отличаются по позициям; но высота +1 может поддерживаться за счёт перебалансировки сложностью O(log n) (Андерсон 1989a; Лаи и Вуд 1990; Фагерберг 1996a). Всё это, конечно, не подходит для практических приложений; алгоритмы слишком запутанные для кодинга, а преимущество во времени поисковых запросов невелико (что не делает их более сложными), и это не следовало бы оправдывать существенной экономией времени при операциях обновления.

Мы уже упоминали граничную оценку высоты 2log n + 1.

Теорема. Красно-чёрное дерево высотой h имеет как минимум листьев при чётных h

и как минимум листьев при нечётных h.

Максимальная высота красно-чёрного дерева с n листьями составляет 2log n – O(1).Доказательство. Мы уже заметили, что граничная оценка высота следует из граничной оценки для (a, b)-деревьев: (2, 4)-дерево с n листьями имеет высота как максимум log n, и каждый (2, 4)-узел заменяется бинарным деревом высоты 2, поэтому полученное бинарное дерево имеет высоту как максимум 2log n. Но мы должны показать, что это не ведёт к переоценке высоты: (2, 4)-дерево высотой имеет только узлы степени 2, поэтому бинарное дерево, полученное из экстремального случая (2, 4)-дерева, также имеет высоту лишь log n. Но мы можем определить экстремальный случай красно-чёрного дерева. Пусть - красно-чёрное дерево высотой h с минимальным числом листьев. Тогда существует путь от корня к листу глубины h, и все красные узлы должны встретиться на этом

пути; в противном случае мы можем уменьшить количество листьев. Поэтому структура - такая, в которой этой путь имеет длину h, и вне этого пути есть только полные бинарные деревья, где все узлы чёрные, высотой i, то есть с 2i листьями. Поскольку высота бинарного дерева, вместе с количеством чёрных узлов по вышеуказанному пути, одинакова для всех этих бинарных деревьев, общее количество листьев имеет вид

red black

$\eta\ T -$

red black
h T □
1+ 2*ι*1 + 2*ι*2 + 2*ι*3 + ...+ 2*ιη* ,
где *j j* 1 *i i*□ □
во л
и каждая экспонента встречается как максимум дважды, один раз ниже
красного узла и один раз ниже его чёрного вышележащего соседа.
Поэтому для чётных h
количест в *h T* истье составляет
red□*black*
0 1 2 (/2) 1 (/2) 1 1 2(2 2 2 ... 2) 2 1 *η η* − + + + + + + = −,
а для нечётных h это
1 2(20 21 22 ... 2((1)/2) 1) 2(1)/2 3 2(1)/2 1
2
□ □ □ □ □ *h*□ □ □ *h*□ □ *h*□ □
.
Поэтому в худшем случае высота красно-чёрного дерева
действительно 2log *n* □*O*(1) .
Красно-чёрное дерево высотой 8 с минимальным числом листьев
Как и в случае сбалансированных по высоте деревьев, не оценка высоты для худшего случая
является жёсткой, но возможно, что почти все листья будут на той глубине; такое красно-
чёрное дерево было спроектировано Кэмероном и Вудом (1992).
Сейчас мы опишем красно-чёрное дерево с его стандартным методом восходящей
перебалансировки, поскольку это классический учебный материал, а в разделе 3.5 –
альтернативный метод нисходящей перебалансировки. Оба работают с одной и той же
структурой. Узел красно-чёрного дерева содержит в качестве информации о
перебалансировке только поле цвета.
Нам требуется поддерживать следующие свойства сбалансированности:

(1) каждый путь от корня до листа содержит одинаковое количество чёрных узлов и
(2) если у красного узла есть нижележащие соседи, они чёрные.
Также удобно добавить условие, что корень является чёрным. Это не является ограничением, поскольку мы всегда может раскрасить корень без влияния на
другие условия; но это предположение гарантирует, что каждый красный узел имеет чёрного
вышележащего соседа, поэтому мы можем условно сжать все красные узлы в их
вышележащих соседей, чтобы установить изоморфизм с (2, 4)-деревьями.
Операции перебалансировки различны для вставки и удаления. Для вставки, мы выполняем
базовую операцию вставки и помечаем оба новых узла как красные. Это, возможно, нарушает
условие (2), но сохраняет условие (1); поэтому перебалансировка после вставки начинается с
красного узла, имеющего нижележащих красных соседей, и перемещает это противоречение
цветов вверх по пути к корню, пока противоречие не исчезнет.
Для удаления, мы выполняем базовую операцию удаления, но сохраняем цвета узлов; если
удалённые листья была чёрными, это нарушает условие (1), но сохраняет условие (2); мы
снова будем перемещать это нарушение вверх по пути к корню, пока оно не исчезнет.
Метод вставки с перебалансировкой работает следующим образом: если нарушение условия
(2) встречается в корне, мы помечаем корень как чёрный узел. В противном случае пусть
*upper – узел с нижележащими соседями *current и *other, где *current – вышележащий узел
для пары красных узлов, нарушающей условие (2). Поскольку есть только одна пара узлов,
нарушающая (2), *upper – чёрный узел. Правила выглядят следующим образом:

1. Если other – красный, пометить current и other как чёрные, а upper как красный.
2. Если current = upper->left
2.1. Если current->right->color является чёрным, выполнить правый поворот вокруг upper и
пометить upper->right как красный.
2.2. Если current->right->color является красным, выполнить левый поворот вокруг current с последующим правым поворотом вокруг upper, и пометить upper->right и upper->left как чёрные, а upper как красный.
3. Если current = upper->right
3.1. Если current->left->color является чёрным, выполнить левый поворот вокруг upper и пометить upper->left как красный.
3.2. Если current->left->color является красным, выполнить правый поворот вокруг current с последующим левым поворотом вокруг upper, и пометить upper->right и upper->left как чёрные, а upper как красный.
Легко увидеть, что условие (1) сохраняется за счёт этих операций, а нарушение условия (2) перемещается на два узла вверх по дереву в случаях 1, 2.2, 3.2 и исчезает в случаях 2.1 и 3.1 или если оно было в корне. Поскольку нам требуется только O(1) вычислений на каждом уровне по пути к корню длиной O(log n), эта балансировка занимает O(log n) времени.
Ситуация и случаи 1, 2.1 и 2.2 вставки с балансировкой: у current есть красный нижележащий сосед
На самом деле, на основе таких же рассуждений, как и в общем случае для (a, b)-деревьев, мы можем показать, что амортизированное число поворотов всего лишь O(1). Ассоциируем с
каждым чёрным узлов количество красных узлов, для которых этот узел является следующим чёрным узлом над ними, и присвоим чёрным узла потенциал 1, 0, 3, 6, если они ассоциированы с 0, 1, 2, 3 красными узлами соответственно. Тогда каждая базовая операция вставки увеличивает сумму потенциалов как минимум на 3, тогда как операции 1, 2.2 и 3.2 уменьшают сумму потенциалов как минимум на 2, а операции 2.1 и 3.1 могут встретиться только однократно в процессе балансировки. Проделана схожая работа по анализу, хотя метод балансировки с поворотами не эквивалентен балансировке с расщеплением, присоединением и разделением.

За счёт незначительного усложнения правил балансировки мы могли получить даже 4 поворота в худшем случае при балансировке, выполняемой при вставке. В случаях 2.2 и 2.3, которые являются единственными случаями поворота, которые перемещают цветовой конфликт, нам нужно помечать upper->right и upper->left как чёрные, поскольку возможно, что оба нижележащих соседа узла current являются красными; но это может быть лишь однажды – на листовом уровне. После этого, всегда есть не более одного нижележащего красного соседа. Тогда мы могла бы в случаях 2.2 и 3.2 помечать upper->right и upper->left как красные, а upper как чёрный; при этом изменении все случаи поворота выше листового уровня будут устранять конфликт цветов.

Операция удаления с балансировкой, к сожалению, более запутанная.

1 В этой ситуации у нас имеет место нарушение правила (1): узел *current, для которого все пути через этот узел
к листу содержат на один чёрный узел меньше, чем следовало бы. Есть две простых
ситуации:

1. Если current красный, мы помечаем его как чёрный.
2. Если current является корнем, (1) выполняется в любом случае.

В противном случае мы можем положить, что *current является чёрным и имеет
вышележащего соседа *upper, у которого есть другой нижележащий сосед *other. Поскольку
все пути от *other до листа содержат как минимум две дальше лежащие чёрные вершины, все
вершины, на которые мы ссылаемся в следующих случаях, существуют на самом деле.

Случаи и правила преобразования выглядят следующим образом:

3. Если current = upper->left

1 Очень легко ошибиться среди этого множества случаев; в широко известном учебнике по алгоритмам
один из случаев удаления с перебалансировкой описан неверно.

3.1. Если upper чёрный, other чёрный и other->left чёрный, выполнить левый поворот вокруг
upper и пометить upper->left как красный, а upper как чёрный. Тогда нарушение (1) возникает
в upper.

3.2. Если upper чёрный, other чёрный и other->left красный, выполнить правый поворот вокруг
other с последующим левым поворотом вокруг upper, и пометить upper->left, upper->right и
upper как чёрные. Тогда (1) восстановлено.
3.3. Если upper чёрный, other красный и other->left->left чёрный, выполнить левый поворот вокруг upper с последующим левым поворотом вокруг upper->left и пометить upper->left->left как красный, upper->left и upper как чёрные. Тогда (1) восстановлено.
3.4. Если upper чёрный, other красный и other->left->left красный, выполнить левый поворот вокруг upper с последующим правым поворотом вокруг upper->left->right и левым поворотом вокруг upper->left, и пометить upper->left->left и upper->left->right как чёрные, upper->left как красный и upper как чёрный. Тогда (1) восстановлено.
3.5. Если upper красный, other чёрный и other->left чёрный, выполнить левый поворот вокруг upper и пометить upper->left как красный и upper как чёрный. Тогда (1) восстановлено.
3.6. Если upper красный, other чёрный и other->left красный, выполнить правый поворот вокруг other с последующим левым поворотом вокруг upper, и пометить upper->left и upper->right как чёрный и upper как красный. Тогда (1) восстановлено.
4. Если current = upper->right
4.1. Если upper чёрный, other чёрный и other->left чёрный, выполнить правый поворот вокруг upper и пометить upper->right как красный и upper как чёрный. Тогда нарушение (1) возникает в upper.
4.2. Если upper чёрный, other чёрный и other->right красный, выполнить левый поворот вокруг other с последующим правым поворотом вокруг upper, и пометить upper->left, upper->right и upper как чёрные. Тогда (1) восстановлено.
4.3. Если upper чёрный, other красный и other->right->right чёрный, выполнить правый поворот вокруг upper с последующим правым поворотом вокруг upper->right, и пометить upper->right->right как красный, а upper->right и upper как чёрный. Тогда (1) восстановлено.
4.4. Если upper чёрный, other красный и other->right->right красный, выполнить правый поворот вокруг upper с последующим левым поворотом вокруг upper->right->left и правым

поворотом вокруг upper->right, и пометить upper->right->right и upper->right->left как чёрные, upper->right как красный и upper как чёрный. Тогда (1) восстановлено.
4.5. Если upper красный, other чёрный и other->right чёрный, выполнить правый поворот вокруг upper и пометить upper->right как красный, а upper как чёрный.
4.6. Если upper красный, other чёрный и other->right как красный, выполнить левый поворот вокруг other с последующим правым поворотом вокруг upper и пометить upper->left и upper->right как чёрные, а upper как красный. Тогда (1) восстановлено.
Ситуация и случаи 3.1-3.6 удаления с балансировкой: путей через current, имеющих один чёрный узел, слишком мало. Снова мы выполняем только O(1) операций на уровень по пути от листа к корню, поэтому в итоге получим O(log n). Только операции 3.1 и 4.1 могут встречаться более одного раза, на самом деле они могут встретиться □(log *n*) раз, это можно увидеть, когда работаем с полным бинарным деревом, помеченным целиком чёрным, удаляя одну из вершин. Это завершает доказательство того, что балансировка красно-чёрных деревьев после вставок и удалений может быть выполнена за время O(log n).
Теорема. Структура красно-чёрного дерева поддерживает выполнение операций поиска, вставки и удаления за время O(log n).
Снова приведём реализацию операции вставки для красно-чёрных деревьев.

Нисходящая балансировка для красно-чёрных деревьев.

Метод из предыдущего раздела снова был очень похож на метод, используемый в сбалансированных по высоте и сбалансированных по весу деревьях, рассмотренных в разделах 3.1 и 3.2; он отделяет поиск листового узла от балансировки, которая выполняется по восходящему пути, с возвратом назад от листа к корню. Но красно-чёрные деревья также допускают нисходящую балансировку, как это делают сбалансированные по весу деревья и (а, Ь)-деревья, которые выполняют балансировку по пути вниз к листовому узлу без необходимости возвращаться к корню. Этот метод является особым случаем метода, который упомянут нами в разделе 3.3, но сейчас мы опишем его подробно для красно-чёрных деревьев.

Чтобы выполнить вставку, мы спускаемся от корня к листу и на основе некоторых преобразований обеспечиваем, чтобы у текущего чёрного узла было не более одного красного нижележащего соседа. Поэтому каждый раз, когда нам попадается чёрный узел с двумя красными нижележащими соседями, нам требуется использовать некоторое балансировочное преобразование; это соответствует расщеплению (2, 4)-узлов степени 4. Таким образом, на листовом уровне мы всегда попадаем в чёрный лист, поэтому мы можем вставить новый лист ниже этого чёрного узла без какой- либо дальнейшей балансировки.

Чтобы выполнить удаление, мы спускаемся вниз от корня к листу и на основе некоторых преобразований обеспечиваем, чтобы у текущего чёрного узла был как минимум один красный нижележащий сосед. Поэтому каждый раз, когда мы встречаем чёрный узел с двумя чёрными нижележащими соседями, нам требуется выполнить некоторое балансировочное преобразование; это соответствует присоединению или объединению (2, 4)- узлов степени 2. Таким образом, мы попадаем на листовом уровне в чёрный узел, у которого как минимум один красный нижележащий сосед, поэтому мы можем удалить лист ниже этого чёрного узла без какой-либо дальнейшей балансировки.

Далее приведены правила балансировки для вставки сверху вниз: пусть * current - текущий чёрный узел, лежащий на пути поиска, а *upper - предшествовавший ему чёрный узел (возможно, между этими чёрными узлами будет красный узел). Согласно нашему алгоритму балансировки, у * upper уже есть не более одного красного нижележащего соседа.

1. Если как минимум один из узлов current->left и current->right является чёрным, балансировка не нужна.

2. Если и current->left, и current->right являются красными и current- >key < upper->key, то:

2.1. Если current = upper->left, пометить current->left и current->right как чёрные, а current - как красный. Если upper->left->key < new_key, установить current в значение cur->left->left, в противном случае установить current в значение upper->left->right.

2.2. Если current=upper->left->left, то выполнить правый поворот для upper, пометить upper->left и upper->right как красные, а upper->left->left и upper->left->right - как чёрные. Если upper->left->key <

newjtey, установить current в значение upper->left->left, в противном случае установить current в значение upper->left->right.

2.3. Если current=upper->left->right, выполнить левый поворот для upper->left с последующим правым поворотом для upper, пометить upper->left и upper->right как красные, а upper->lefi->right и upper->right->left как чёрные. Если upper->key < new key, установить current в значение upper->left->right, иначе установить current в upper->right->left.

3. В противном случае оба узла current->left и current->right красные и current- > key > upper- > key, тогда:

3.1. Если current=upper->right, пометить current->left и current->right как чёрные узлы, а current - как красный. Если upper->right->key < new_key, то установить current в значение upper->right->left, иначе установить current в upper->right->right.

3.2. Если current=upper->right->right, выполнить левый поворот для upper, пометить upper->left и upper->right как красные, а upper->right->left и upper->right->right - как чёрные. Если upper->right->key < new_key, установить current в значение upper->right->lefi, в противном случае установить current в upper->right->right.

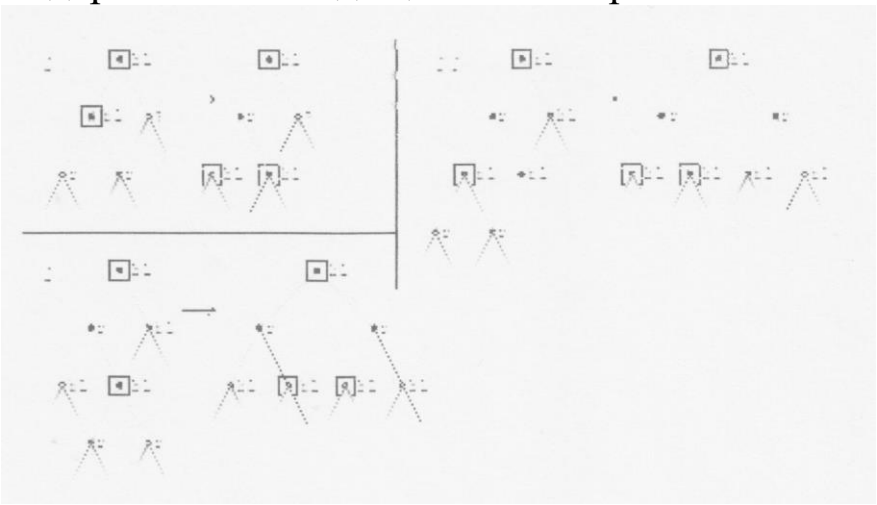
3.3. Если current=upper->right->left, выполнить правый поворот для upper->right с последующим левым поворотом для upper, и пометить upper- >left и upper->right как красные узлы, а upper->left->right и upper->right->left - как чёрные. Если upper->key<new_key, установить current в значение upper- >lefl->right, иначе установить current в значение upper->right->left.

Новый current в случаях 2 и 3 ранее был красным узлом, поэтому оба его соседа снизу чёрные. После этого балансировочного преобразования мы устанавливаем upper в значение current и перемещаем current далее вниз по пути поиска, пока не встретим или чёрный узел, или лист. Если встречаем чёрный узел, повторяем балансировочное преобразование, а если встречаем лист, выполняем вставку. Вставка порождает новый внутренний узел ниже upper, который мы помечаем как красный узел. Если upper - вышележащий сосед этого нового красного узла, мы завершаем операцию, в противном случае единственный красный узел ниже upper - это узел над новым узлом, тогда мы выполняем поворот вокруг upper и восстанавливаем свойство красно-чёрного дерева.

Нисходящая вставка в случаях 2.1-2.3: спуск по дереву, upper и current

помечены как current

Далее мы представляем реализацию операции вставки в красно-чёрные деревья с нисходящей балансировкой.



Правила балансировки при нисходящем удалении снова более сложные. Пусть * current - текущий чёрный узел на пути поиска, а *upper - предшествовавший ему чёрный узел (возможно, что между этими двумя чёрными узлами будет красный узел). Нам нужно поддерживать следующее свойство: как минимум один из узлов upper->left и upper->right должен быть красным.

1. Если хотя бы один из узлов current->left и current->right является красным, никакой балансировки не требуется. Установить upper в значение current и переместить current вниз по пути поиска, к следующему чёрному узлу.

2. Если current->left и current->right оба чёрные и current->key < upper- >key, тогда:

2.1. Если current = upper->left и

2.1.1. upper->right->left->left и upper->right->left->right - оба чёрные узлы, выполнить левый поворот для upper и пометить upper->lefi как чёрный

узел, а upper->left->left и upper->left->right как красные, и установить current и upper в значение upper->left.

2.1.2. upper->right->left->left является красным, выполнить правый поворот для upper->right->left с последующим правым поворотом для upper- bright и левым поворотом для upper, и пометить upper->left и upper->right- >left как чёрные узлы, а upper->right и

upper->left->left - как красные, и установить current и upper в значение upper->left.

2.1.3. upper->right->left->left является чёрным, а upper->right->left->right является красным: выполнить правый поворот для upper->right с последующим левым поворотом для upper и пометить upper->left и upper- >right->left как чёрные узлы, а upper->right и upper->left->left как красные, и установить current и upper в значение upper->left.

2.2. Если current=upper->left->left и

2.2.1. upper->left->right->left и upper->left->right->right оба чёрные: пометить upper->left->left и upper->left->right как красные, а upper->left как чёрный, и установить current и upper в значение upper->left.

2.2.2. upper->left->right->right красный: выполнить левый поворот для upper->left и пометить upper->left->lefi и upper->left->right как чёрные, а upper->left и upper->left->left->left - как красные, и установить current и upper в значение upper->left->left.

2.2.3. upper->left->right->left красный и upper->left->right->right чёрный: выполнить правый поворот для upper->left->right с последующим левым поворотом для upper->lefi и пометить upper->left->left и upper->left->right как чёрные, а upper->left и upper->left->left->lefi как красные, и установить current и upper в значение upper->left->left.

2.3. Если current=upper->left->right и

2.3.1. upper->left->left->left и upper->left->left->right оба чёрные: пометить upper->left->left и upper->left->right как красные, а upper->lefl как чёрный, и установить current и upper в значение upper->left.

2.3.2. upper->left->left->left красный: выполнить правый поворот для upper->left и пометить upper->left->left и upper->left->right как чёрные, а upper->lefi и upper->lefl->right->right как красные, и установить current и upper в значение upper->left->right.

2.3.3. upper->left->left->left чёрный, а upper->left->left->right красный: выполнить левый поворот для upper->left->left с последующий правый поворотом для upper->left и пометить upper->lefi->left и upper->left->right как чёрные узлы, а upper->left и upper->left->right->right как красный, и установить current и upper в значение upper->left->right.

3. В противном случае current->left и current->right оба чёрные и current- > key > upper- > key

3.1. Если current=upper->right и

3.1.1. upper->left->right->right и upper->left->right->left оба чёрные: выполнить правый поворот для upper и пометить upper->right как чёрный

узел, а upper->right->right и upper->right->left как красные узлы, и установить current и upper в значение upper->right.

3.1.2. upper->left->right->right красный: выполнить левый поворот для upper->left->right с последующим левым поворотом для upper->left и правым поворотом для upper и пометить upper->right и upper->left->right как чёрные узлы, а upper->left и upper->right->right как красные узлы, и установить current и upper в upper->right.

3.1.3. upper->left->right->right чёрный, а upper->left->right->left красный: выполнить левый поворот для upper->left с последующим правым поворотом для upper и пометить upper->right и upper->left->right как чёрные, а upper->left и upper->right->right как красные, и установить current и upper в значение upper->right.

3.2. Если current=upper->right->right и

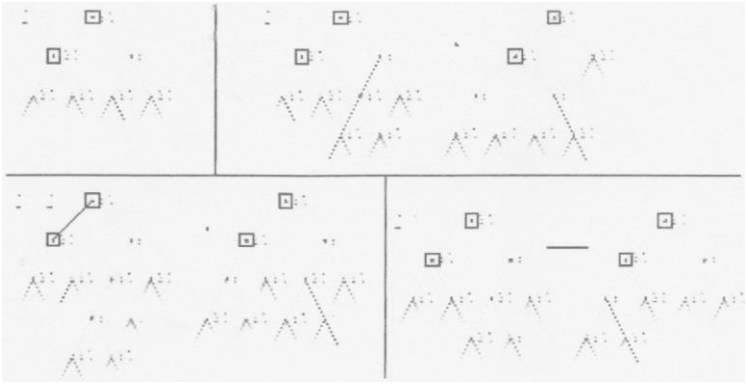
3.2.1. upper->right->left->right и upper->right->left->left оба чёрные: пометить upper->right->right и upper->right->lefl как красные, а upper->right как чёрный, и установить current и upper в значение upper->right.

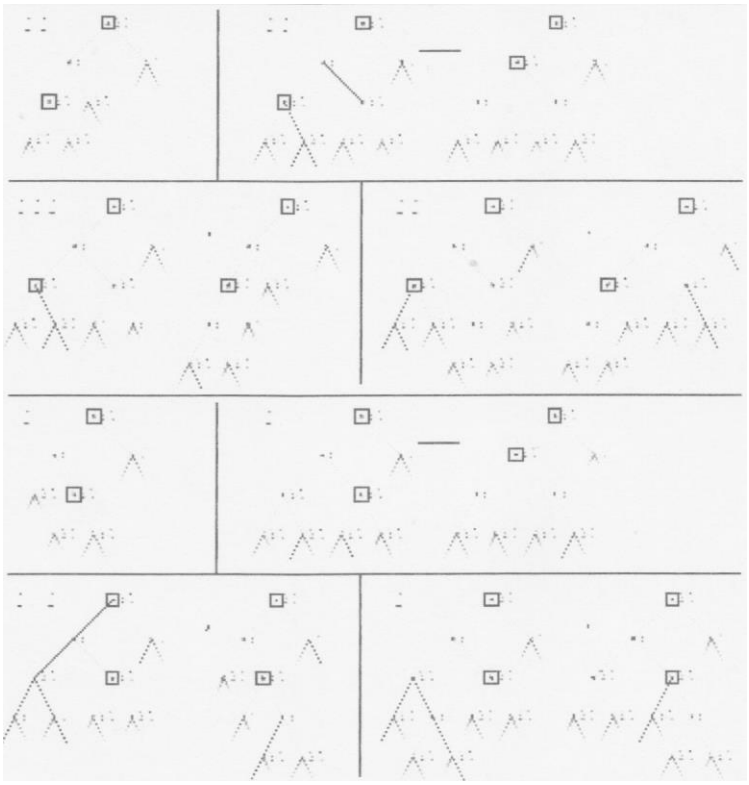
3.2.2. upper->right->left->left красный: выполнить правый поворот для upper->right и пометить upper->right->right и upper->right->left как чёрные, а upper->right и upper->right->right->right как красные, и установить current и upper в значение upper->right->right.

3.2.3. upper->right->left->right красный, а upper->right->left->left чёрный: выполнить левый поворот для upper->right->left с последующим правым поворотом для upper->right и пометить upper->right->right и upper->right->left как чёрные, а upper->right и upper->right->right->right как красные, и установить current и upper в upper->right->right.

3.3. Если current=upper->right->left и

3.3.1. upper->right->right->right и upper->right->right->left оба чёрные: пометить upper->right->right и upper->right->lefl как красные, а upper->right как чёрный, и установить current и upper в upper->right.





Случаи 2.1-2.3 нисходящего удаления и разновидности данных случаев:

отмечены upper и current

3.3.2. upper->right->right~>right красный: выполнить левый поворот для upper->right и пометить upper->right->right и upper->left->left как чёрные, и upper->right и upper->right->left->left как красные, и установить current и upper в значение upper->right->left.

3.3.3. upper->right->right->right чёрный, а upper->right->right->left красный: выполнить правый поворот для upper->right->right с последующим левым поворотом для upper->right и пометить upper->right->right и upper- >right->left как чёрные, а upper->right и upper->right->left->left как красные, и установить current и upper в значение upper->right->left.

После этого перебалансировочного преобразования мы перемещаем current далее вниз по пути поиска, пока он (current - прим. ред.) не встретит чёрный узел или лист. Если он встречает чёрный узел, мы повторяем перебалансировочное преобразование, а если он встречает лист, мы выполняем удаление. При удалении уничтожается лист и внутренний узел ниже upper, но ниже upper есть как минимум один красный узел. Если лист ниже, чем тот красный узел, мы лишь удаляем его и красный узел, иначе мы выполняем поворот вокруг upper с целью перенести красный узел так, чтобы он был над листом, и затем мы удаляем лист и красный узел. За счёт этого мы сохраняем свойство красно-чёрного дерева.

3.3. Сбалансированные по весу деревья

В 1970 годах было предложено несколько альтернативных способов поддержать в деревьях поиска высоту Log n. Естественный критерий альтернативного баланса - сбалансированный вес, т.е. число узлов или листьев, вместо высоты поддеревьев.

Существует некоторое разнообразие сбалансированных по весу бинарных деревьев:

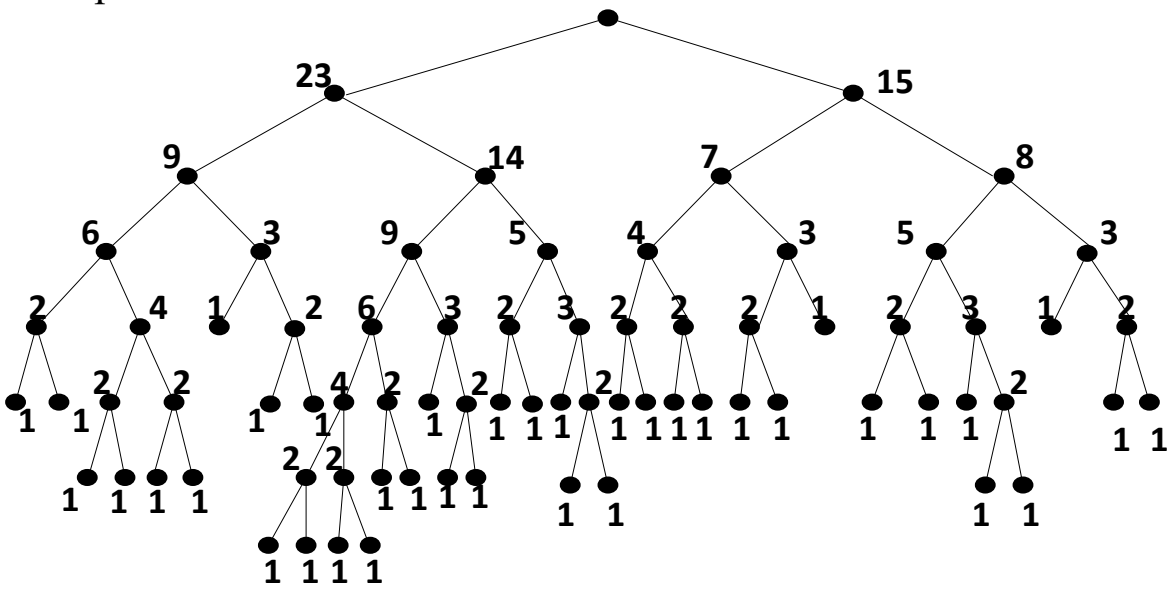
- сбалансированное бинарное дерево (bounded balanced tree). Деревья были представлены как "деревья ограниченного баланса" или BB[a]-деревья Нивергельдом и Рейнгольдом (1973) и Nievergelt (1974), и далее изученный Баером (1975), Блумом и Мехлхорном (1980).
- сбалансированное по весу бинарное дерево (Weight- balanced tree – WBT);
- дерево Адамса (bounded balanced weight – BB-w);
- вариант сбалансированного по весу дерева был предложен Чо и Сэни (2000) (weight biased leftist tree - WBLT).
- scapegoat tree и др.

В дальнейшем будем рассматривать бинарное сбалансированное по весу дерево, в котором балансируются не количество узлов, а количество листьев. Будем называть их WB(α) - weight- balanced from α, где α - показатель сбалансированности дерева. Он в свою очередь равен минимальному количеству листьев в одной из ветвей к общему количеству листьев. Вес дерева - количество листов в нём, вес левых и правых поддеревьев в каждом узле должен быть сбалансирован.

У WB(α) дерева обязательно маленькая высота.

Нивергельд и Рейнгольд рассматривали $\alpha < 1-\sqrt{2}/2$ как необходимое условие для балансировки. Но α нельзя выбирать очень маленьким, иначе балансировка может прводить к ошибке в некоторых случаях. Блум и Мехлхорн (1980) N. Blum, K. Mehlhorn: On the Average Number of Rebalancing Operations in Weight-Balanced Trees, *Theoretical Computer Science* **11** (1980) 303–320..задают $\alpha < 2/11$ как нижнюю границу, но реально если мы используем различные методы балансировки для небольших деревьев, мы можем выбрать α меньше.

В рассматриваемой модели мы ограничиваемся небольшим интервалом $\alpha \in [2/7, 1-\sqrt{2}/2] \supset [0.286, 0.292]$, но с дополнительной работой по балансировке деревьев маленького веса, но можно было выбрать α произвольно маленьким.



Для описания алгоритма балансировки в этом классе, мы, во-первых, должны выбрать α и, во-вторых параметр ε соответствующий $\varepsilon < \alpha^2 - 2\alpha + 1/2$. Как и в сбалансированном по высоте дереве, мы должны сохранить некоторую дополнительную информацию в каждом внутреннем узле дерева поиска - вес поддерева с корнем в этом узле.

Структура узла следующая:

```
typedef struct tr_n_t { key_t       key;
          struct tr_n_t *left;
          struct tr_n_t *right;
        int weight;
/* possibly other information */
 } tree_node_t;
```

Вес узла *n определяется рекурсивно следующими правилами:

{Если *n - лист (n->left = NULL), тогда n-> вес = 1.

{Иначе n-> вес - сумма веса левого и правого поддеревьев: n-> вес = n->left-> вес + n->право-> вес.

Узел *n является WB(α) если

n->лево->вес>αn->вес и n->право->вес>αn->вес,

или эквивалентно αn->лево->вес<(1-α)n->право->вес и

(1-α)n->лево->вес>αn->право->вес.

Информация о весе должна исправляться каждый раз, когда дерево изменено и используется для сохранения веса сбалансированным. Информация изменяется только во время операции вставки и удаления, и только в тех узлах на пути от измененных листов к корню максимум на 1. Интуитивно, ниже текущего узла поддеревья уже сбалансированы.

Если *n - текущий узел, и мы уже исправляли его вес, то для балансировки возможны следующие случаи:

1. n->left->weight > an->weight and
   n->right->weight > an->weight.
   В этом случае нет необходимости балансировки этого узла. Мы переходим к следующему узлу
2. n->right->weight < α n->weight
   2.1. If n->left->left->weight > (α + ε)n->weight
   выполнение правого поворота вокруг n следует после пересчёта веса в n->right
   2.2. Иначе выполняется левый поворот вокруг n->left, следующий за правым поворотом, следующий за вычислением веса n->left и n->right
3. n->left->weight < α n->weight
   3.1. If n->right->right->weight > (α + ε)n->weight.
   выполнение левого поворота вокруг n следует после пересчёта веса в n->left
   3.2. Иначе выполняется правый поворот вокруг n->left, следующий за левым поворотом, следующий за вычислением веса n->left и n->right.

Заметим, что в отличие от сбалансированных деревьев по высоте, мы должны всегда следовать от узла до корня без остановки,

потому что информация о весе в отличие от информации о высоте, изменяется вдоль всего пути. Поскольку работа над каждым узлом путивыполняется за один шаг, максимум за два вращения и самое большее за три расчета веса и путь имеет длину O(log n), то балансировка выполняется за O(log n) время. Но мы должны показать, что баланс WB(α) дерева восстанавливается.

Пусть *n^{old} будет узлом перед шагом балансировки, а *n^{new} тот же узел после шага балансировки. Обозначьте вес n^{old}->вес = n^{new}-->вес w. Мы должны проанализировать только случай 2; в случае 1 узел уже сбалансирован, и случай 3 следует из случая 2 симметрично. В случае 2, мы имеем n^{old}->right->weight < αw, но вес изменился только на 1 до того как узел был сбалансирован; тогда n^{old}->right->weight = $\alpha w - \delta$ для некоторого $\delta \in [0, 1]$. Теперь надо проверить случаи 2.1 и 2.2, который изменили все узлы на том шаге балансировки.

2.1. У нас есть n^{old} ->left->left->weight > (α+ε)w и выполнение правого поворота вокруг n^{old}. Поэтому

n^{old} ->left->left становится n^{new}->left,

n^{old} - >left->right становится n^{new} ->right->left, и

n^{old} ->right становится n^{new}->right->right.

Потому что n^{old}->left было сбалансировано, с

n^{old} ->left- >weight= $(1 - \alpha)w + \delta$,, мы имеем

n^{new} ->right->left->weight ∈ $[\alpha(1-\alpha)w+\alpha\delta, (1-2\alpha-\varepsilon)w + \delta]$,

n^{new} ->right->right->weight=$\alpha w - \delta$,,

n^{new} ->right-> weight ∈ $[\alpha(2 - \alpha)w - (1 - \alpha)\delta, (1 - \alpha - \varepsilon)w]$,

n^{new} ->left-> weight ∈ $[(\alpha + \varepsilon)w, (1 - \alpha)2w + (1 - \alpha)\delta]$.

Сейчас для n^{new} ->right условия баланса:

A) αn^{new} ->right->left->weight< (1-α)n^{new} ->right->right->weight, so $\alpha((1 - 2\alpha - \varepsilon)w + \delta) \le (1 - \alpha)(\alpha w - \delta)$,, которые соответствуют для $(\alpha 2 + \alpha\varepsilon)w \ge \delta$; и

B) $(1-\alpha)n^{new}$ ->right->left->weight>αn^{new} - >right->right->weight, так$(1 - \alpha)$ $(\alpha(1 - \alpha)w + \alpha\delta) \ge \alpha(\alpha w - \delta)$,, что соответствует $0 \le \alpha \le (3-\sqrt{5})/2$.

A для n^{new} условия баланса

C) αn^{new} ->left->weight < (1 - α) n^{new} - >right->weight, so

$\alpha((1 - \alpha)2w + (1 - \alpha)\delta) \le (1 - \alpha)(\alpha(2 - \alpha)w - (1 - \alpha)\delta)$,

которые соответствуют $\alpha w \ge \delta$; and

D) (1 - α) n^{new} ->left->weight > α n^{new} ->right->weight, так (1 − α) ((α + ε)w) ≥ α ((1 − α − ε)w) которые соответствуют для все α, со строгим неравенством для δ > 0. Вместе все это показывает в интересном интервале α ∈ [0, 1 – 1/√2], по крайней мере, если поддерево не будет очень маленьким (для $\alpha^2 w \geq 1$) в случае 2.1, WB(α) восстановлено.

2.2 Мы имеем n^{old} ->left->left->weight ≤ (α + ε)w и выполним левый поворот вокруг n^{old} ->left, сопровождаемый правым поворотом вокруг n^{old}. Так n^{old} ->left->left становится n^{new} ->left->left, n^{old} ->left->right->left становится n^{new} ->left->right, n^{old} ->left->right->right становиться n^{new} ->right- >left, и n^{old} ->right становится n^{new} ->right->right. Так как n^{old} ->left был сбалансирован, с n^{old} ->left->weight=(1 − α)w + δ, мы имеем условия случая 2.2

n^{new} ->left->left->weight ∈ [α(1 − α)w + $\alpha\delta$, (α + ε)w],

n^{old} ->left->right->weight ∈ [(1 − 2α − ε)w + δ, (1 − α)2w + (1 − α)δ],

n^{new} ->left->right- >weight,

n^{new} ->right->left->weight ∈ [α(1−2α−ε)w + $\alpha\delta$, (1 −α)3w + (1 −α)2δ],

n^{new} - >right->right->weight= $\alpha w - \delta$,

n^{new}->left->weight ∈[(2α−3α^2+α^3)w+α(2−α)δ, (1−2α+2α^2+$\alpha\varepsilon$)w+(1−α)δ],

n^{new}->right->weight∈[(2α−2α^2−$\alpha\varepsilon$)w−(1−α)δ,(1−2α+3α^2−α^3)w+α(α−2)δ]

Тогда условие баланса для n^{new} ->left будет

a. α n^{new} ->left->left- >weight<(l- α) n^{new} ->left->right->weight, so $\alpha((\alpha+\varepsilon)w) \leq (1-\alpha)(\alpha(1-2\alpha-\varepsilon)w+\alpha\delta)$, что верно для $\alpha \in [0, 1-1/\sqrt{2}]$ и $\varepsilon \leq \alpha^2 - 2\alpha + 1/2$;

b. (1 - α) n^{new} ->left->left->weight> α n^{new} ->left->right->weight, so $(1-\alpha)\ (\alpha(1-\alpha)w+\alpha\delta) \geq \alpha((1-\alpha)^3 w + (1-\alpha)^2\delta)$, что верно для $\alpha \in [0, 1]$.

Условие баланса для n^{new} ->right будет

c. a n^{new} ->right->left->weight < (1 - a) n^{new} ->right->right->weight, также $\alpha((1-\alpha)^3 w + (1-\alpha)^2\delta) \leq (1-\alpha)\ (\alpha w - \delta)$,, что верно как минимум для $(2-\alpha)\alpha^2 w \geq (1+\alpha-\alpha^2)\delta$;

d. (1 - α) n^{new} - >right->left->weight ≥ α n^{new} ->right->right->weight, также $(1-\alpha)\ (\alpha(1-2\alpha-\varepsilon)w + \alpha\delta) \geq \alpha\ (\alpha w - \delta)$, что верно для

$\alpha \in [0, 1-1/\sqrt{2}]$ и $\varepsilon \leq 2\alpha^2 - 4\alpha + 1$.

Условие баланса для n^{new} следующие:

e. α n^{new} ->left->weight < (1 — α) n^{new} - >right->weight, также

$\alpha((1 - 2\alpha + 2\alpha 2 + \alpha\varepsilon)w + (1 - \alpha)\delta) \leq (1 - \alpha)((2\alpha - 2\alpha 2 - \alpha\varepsilon)w - (1 - \alpha)\delta)$, что верно для $\alpha(1 - 2\alpha - \varepsilon)w \geq 1$;

f. (1 - α) n^{new} ->left->weight > α n^{new} ->right->weight, также

$(1 - \alpha)((2\alpha - 3\alpha^2 + \alpha^3)w + \alpha(2 - \alpha)\delta) \geq \alpha((1 - 2\alpha + 3\alpha^2 - \alpha^3)w + \alpha(\alpha - 2)\delta)$,

что верно для $\alpha \in [0, (3-\sqrt{5})/2]$.

Вместе все показывает интересный интервал $\alpha \in [0, 1 - 1/\sqrt{2}]$

с $\varepsilon \leq \alpha^2 - 2\alpha + 1/2$, если поддерево не будет слишком маленьким (для $\alpha^2 w \geq 1$, которое подразумевает $(2-\alpha)\alpha^2 w \geq (1+\alpha-\alpha^2)\delta$), $\alpha(1-2\alpha-\varepsilon)w \geq 1$ в интересующем интервале) в случае 2.2, WB(α) восстановлено.

Но мы все еще должны показать, что повторно балансирующийся алгоритм работает на $w < \alpha^{-2}$. Это, к сожалению, не является случаем. Однако это верно для $\alpha \in [2/7, 1-1/\sqrt{2}]$; здесь нам необходимо проверить это только для $w \leq 12$ и n- >right->weight= $\lfloor \alpha w \rfloor$

В случае 2.1 мы имеем n^{old} ->left->left->weight $\geq \lceil \alpha w \rceil$,, и есть только одно неравенство баланса, которое может не работать: мы можем проверить это n^{new} ->right->right->weight-> α n^{new} ->right->weight, так $\lfloor \alpha w \rfloor > \alpha(w - \lceil \alpha w \rceil)$, что просто проверяется.

В случае, если 2.2, мы имеем дополнительно n^{old} ->left->left->weight< $\lfloor \alpha w \rfloor$. Потому что n^{old} ->left->weight= $w - \lfloor \alpha w \rfloor$. Условие баланса в n->left определяет веса n^{old} ->left->left и n^{old} ->left->right, и это просто проверяется для этих деревьев, что баланс восстановлен.

Это завершает доказательство, что балансирование может быть сделано для сбалансированных по весу деревьев после вставок, и удаления в O (log n) время.

Теорема: Структура WB(α) дерева поддерживает поиск, вставку и удаление за O(log n) время.

Возможная реализация вставки в WB(α) теперь следующая:

```
#define ALPHA 0.288
#define EPSILON 0.005
int insert(tree_node_t *tree, key_t new_key,
object J; *new_object)
{ tree_node_t *tmp_node;
if( tree->left == NULL)
{ tree->left = (tree_node_t *) new_object;
tree->key = new_key;
```

```
tree->weight = 1;
tree->right = NULL; }
else
{ create_stack(); tmp_node = tree;
while( tmp_node->right != NULL ) { push( tmp_node ); if( new_key < tmp_node->key ) tmp_node = tmp_node->left; else
tmp_node = tmp_node->right; }
/* found the candidate leaf. Test whether key distinct */
if( tmp_node->key == new_key ) return( -1); /* key alreay exists, insert failed */
/* key is distinct, now perform the insert */
{ tree_node_t *old_leaf, *new_leaf; old_leaf = get_node(); old_leaf->left = tmp_node->left; old_leaf->key = tmp_node->key; old_leaf->right = NULL; old_leaf->weight = 1; new_leaf = get_node(); new_leaf->left = (tree_node_t *) new_object;
new_leaf->key = new_key; new_leaf->right = NULL; new_leaf->weight = 1; if( tmp_node->key < new_key ) { tmp_node->left = old_leaf; tmp_node->right = new_leaf;
tmp_node->key = new_key; >
else
{ tmp_node->left = new_leaf;
tmp_node->right = old_leaf; }
tmp_node->weight = 2; }
/* rebalance */ while( !stack_empty()) { tmp_node = pop(); tmp_node->weight = tmp_node->left->weight + tmp_node->right->weight; if( tmp_node->right->weight < ALPHA* tmp_node->weight) { if(tmp_node->left->left->weight > (ALPHA+EPSILON) *tmp_node->weight)
{ right_rotation( tmp_node ); tmp_node->right->weight = tmp_node->right->left->weight + tmp_node->right->right->weight;
}
else
{ left_rotation( tmp_node->left); right_rotation( tmp_node); tmp_node->right->weight = tmp_node->right->left->weight + tmp_node->right->right->weight; tmp_node->left->weight = tmp_node->left->left->weight
+ tmp_node->left->right->weight; }
}
```

```
        else if (tmp_node->left->weight < ALPHA* tmp_node->weight) { if(
tmp_node->right->right->weight > (ALPHA+EPSILON) *tmp_node-
>weight) { left_rotation( tmp_node); tmp_node->left->weight =
tmp_node->left->left->weight
        + tmp_node->left->right->weight; }
        else
        { right_rotation( tmp_node->right); left_rotation( tmp_node );
tmp_node->right->weight = tmp_node->right->left->weight + tmp_node-
>right->right->weight; tmp_node->left->weight = tmp_node->left->left-
>weight
        + tmp_node->left->right->weight; }
        }
        } /* end rebalance */
        return( 0 ); }
```

Базовая функция удаления нуждается в такой же модификации, с тем же кодом балансировки для восстановления работоспособности дерева, и нет никаких изменений в функции поиска. Поскольку мы знаем, что высота стека ограничена $(\log 1/(1-\alpha))^{-1} \log n$,, который является меньше чем 2.07 log n для нашего интервала α, и $n<2^{100}$, стек, основанный на массиве фиксированного максимального размера, разумный выбор. Алгоритм балансировки, описанный здесь, подобен балансировке дерева по высоте в двух фазах: спуск до листа и затем балансировка восхождением. В принципе сбалансированные деревья по весу также позволяют нисходящую балансировку, которая имеет место при спуске до листа и делает вторую фазу ненужной. Это возможно, потому что мы уже знаем правильный вес поддерева при нарушении показателя сбалансированности, таким образом, мы видим, будет ли оно нуждаться в балансировке, тогда как высота поддерева доступна только, когда мы достигаем листа.

Алгоритм был первоначально обрисован в общих чертах для BB [α] дерева (Nievergelt и 1973 Reingold) J. Nievergelt, E.M. Reingold: Binary Trees of Bounded Balance, *SIAM Journal on Computing* **2** (1973) 33–43., и обсуждался Лае и Вуде (1993) T.W. Lai, D. Wood: A Top-Down Updating Algorithm for Weight-Balanced Trees,*International Journal Foundations of Computer Science* **4** (1993) 309–324., но корректный анализ восстановления баланса, еще больше работы для нисходящего балансирования потому что предположение, которое мы имеем ниже текущего узла, более слабо: узел ниже не

обязательно сбалансированный, потому что мы не выполнили балансировку ниже, но самое большее один прочь от баланса.

Относительно максимальной высоты сбалансированные деревья веса не столь же хороши как сбалансированные деревья высоты; для нашего интервала α, коэффициента $\log 1/(1-\alpha))^{-1}$ приблизительно равен 2 вместо 1.44, и для большего α, это будет еще хуже. Это уже наблюдалось в Nievergelt и Reingold (1973) что средняя глубина листьев немного лучше, чем для сбалансированных деревьев высоты.

Теорема. Средняя глубина WB(α) дерева с n листами равна

$-1/(\alpha \log \alpha+(1-\alpha) \log(1-\alpha) \log n)$.

Для нашего интервала α этот коэффициент приблизительно равен 1.15, тогда как для деревьев сбалансированных по высоте мы имели этот коэффициент 1.44.

Доказательство. Мы снова используем максимальный depthsum вместо средней глубины. Это удовлетворяет рекурсивному ограничению depthsum (n) ≤ n + depthsum (a) + depthsum (b) для некоторого a, b c a + b = n, a, b ≥ α n. Мы показываем, что depthsum (n) ≤ cn log n для вышеупомянутого с индукцией, используя

depthsum(n) ≤ n + ca log a + cb log b = cn(l/c+ a/n log a + b/n log b=

cn log n + cn(l/c+ a/n log a/n+ b/n log b/n.

Поскольку функция x log x + (1 - x)log(1-x)

отрицательна и уменьшается на x ∈ [0, 0.5], второй терм неположителен для $c = -1/\alpha \log \alpha+(1-\alpha) \log(1-\alpha)$.

Более замечательное свойство WB(α) деревьев следующее:

Теорема. Во время от одной балансировки определенного узла до следующей балансировки этого узла, положительная часть всех листьев ниже этого узла изменена.

Это замечательно, потому что это вынуждает почти все операции балансировки проводить с листьями. Это обсуждалось в Блуме и Мехлхорне (1980). N. Blum, K. Mehlhorn: On the Average Number of Rebalancing Operations in Weight-Balanced Trees, *Theoretical Computer Science* **11** (1980) 303–320.**.**

Доказательство. Легко проверить, что операции балансировки оставляют каждый из измененных узлов не только α балансировки по весу, но и даже α* балансировки по весу для некоторого $\alpha^*(\alpha,\varepsilon)>\alpha$. Но тогда вес должен измениться положительной частью, нарушив условия баланса. Таким образом, положительная часть листьев должна быть вставлена или удалена прежде, чем тот узел должен

быть повторно сбалансирован. Это причина дополнительного $\varepsilon>0$ используемых в алгоритме балансировки; без этого, в случае 2.1 один из узлов не будет иметь более сильного свойства баланса.

Для сбалансированных деревьев по высоте мы ограничивались различием высот, тогда как для сбалансированных деревьев по весу, мы ограничивались отношением весов. Поэтому некоторые виды сбалансированных деревьев по высоте будут логарифмическими и по весу. Не удивительно, что эти условия имеют тот же эффект. Намного более слабое условие ограничение отношения высот было изучено в Gonnet, Olivine и Вуд (1983) G.H. Gonnet, H. Olivi´e, D. Wood: Height-Ratio Balanced Trees, *The Computer Journal* **26** (1983) 106–108... Оказывается, что эти условия не сильно достаточно дать логарифмическую высоту; максимальная высота отношения высоты сбалансированное дерево с n листами равна $2^{\Theta(\sqrt{\log n})}$ вместо $\Theta(\log n) = 2\log\log n + \Theta(1)$.

BB-дерево (Scapegoat tree)

Сегодня мы посмотрим на структуру данных, называемую Scapegoat-деревом. Дословный перевод «Scapegoat», какой-то странный, поэтому будем использовать оригинальное название.

Scapegoat-дерево тоже имеет свой подход к решению проблемы балансировки дерева. Как и для всех остальных случаев он не идеален, но вполне применим в некоторых ситуациях.

К достоинствам Scapegoat-дерева можно отнести:

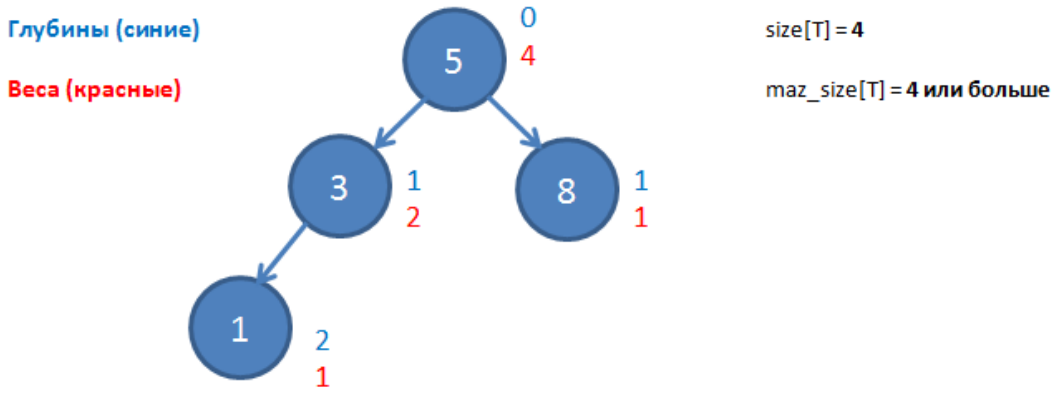
- Отсутствие необходимости хранить какие-либо дополнительные данные в вершинах (а значит мы выигрываем по памяти у красно-черных, АВЛ и декартовых деревьев)
- Отсутствие необходимости балансировать дерево при операции поиска (а значит мы можем гарантировать максимальное время поиска O(log N), в отличии от Splay-деревьев, где гарантируется только амортизированное O(log N))
- Амортизированная сложность операций вставки и удаления O(log N) — это в общем-то аналогично остальным типам деревьев
- При построении дерева мы выбираем некоторый предельный коэффициент сбалансированности α, который позволяет регулировать сбалансированность дерева, делая операции поиска более быстрыми за счет замедления операций модификации или наоборот. Можно реализовать структуру данных, а дальше уже подбирать коэффициент по результатам тестов на реальных данных и специфики использования дерева.

К недостаткам можно отнести:

- В худшем случае операции модификации дерева могут занять O(n) времени (амортизированная сложность у них по-прежнему O(log N), но защиты от «плохих» случаев нет).
- Можно неправильно оценить частоту разных операций с деревом и ошибиться с выбором коэффициента α — в результате часто используемые операции будут работать долго, а редко используемые — быстро.

Используемые понятия (квадратные скобки в обозначениях выше означают, что мы храним это значение явно, а значит можем взять за время O(1). Круглые скобки означают, что значение будет вычисляться по ходу дела т.е. память не расходуется, но зато нужно время на вычисление)

- T — так будем обозначать дерево
- root[T] — корень дерева T
- left[x] — левый «сын» вершины x
- right[x] — правый «сын» вершины x
- brother(x) — брат вершины x (вершина, которая имеет с x общего родителя)
- depth(x) — глубина вершины x. Это расстояние от неё до корня (количество ребер)
- height(T) — глубина дерева T. Это глубина самой глубокой вершины дерева T
- size(x) — вес вершины x. Это количество всех её дочерних вершин + 1 (она сама)
- size[T] — размер дерева T. Это количество вершин в нём (вес корня)



Теперь введем понятия, необходимые нам для Scapegoat-дерева:

- max_size[T] — максимальный размер дерева. Это максимальное значение, которое параметр size[T] принимал с момента последней балансировки. Если балансировка произошла только что, то max_size[T] = size[T]
- Коэффициэнт α — это число в диапазоне от [0.5; 1), определяющее требуемую степень качества балансировки дерева. Некоторая вершина x называется "α-сбалансированной по весу", если вес её левого сына меньше либо равен α *

size(x) и вес ей правого сына меньше либо равен α * size(x).

size(left[x]) ≤ α * size(x)
size(right[x]) ≤ α * size(x)

Давайте попробуем понять этот критерий. Если α = 0.5, то мы требуем чтобы в левом поддереве было ровно столько же вершин, сколько в правом (± 1). Т.е. фактически это требование идеально сбалансированного дерева. При α > 0.5 мы говорим пусть балансировка будет не идеальной, пусть одному из поддеревьев будет позволено иметь больше вершин, чем другому». При α = 0.6 одно из поддеревьев вершины x может содержать до 60% всех вершин поддерева x чтобы вершина x всё ещё считалась "α-сбалансированной по весу". Легко увидеть, что при α стремящемуся к 1 мы фактически соглашаемся считать "α-сбалансированным по весу" даже обычный связный список.

Теперь давайте посмотрим как в Scapegoat-дереве выполняются обычные операции.
Мы выбираем параметр α в диапазоне [0.5; 1). Также заводим две переменные для хранения текущих значений size[T] и max_size[T] и обнуляем их.

Поиск

Итак, у нас есть некоторое Scapegoat-дерево и мы хотим найти в нём элемент. Поскольку это двоичное дерево поиска, то и поиск будет стандартным: идём от корня, сравниваем вершину с искомым значением, если нашли — возвращаем, если значение в вершине меньше — рекурсивно ищем в левом поддереве, если больше — в правом. Дерево по ходу поиска не модифицируется. Сложность операции поиска зависит от α и выражается формулой $O(\log_{1/\alpha} n)$. Т.е. сложность, конечно, логарифмическая, вот только основание логарифма интересное. При α близком к 0.5 мы получаем двоичный (или почти двоичный) логарифм, что означает идеальную (или почти идеальную) скорость поиска. При α близком к единице основание логарифма стремится к единице, а значит общая сложность стремится к O(n).

Вставка

Начинается вставка нового элемента в Scapegoat-дерево классически: поиском ищем место, куда бы подвесить новую вершину, ну и подвешиваем. Легко понять, что это действие могло нарушить α-балансировку по весу для одной или более вершин дерева. И вот теперь начинается то, что и дало название структуре данных: мы ищем «козла отпущения» (Scapegoat-вершину) — вершину, для которой потерян α-баланс и её поддерево должно быть перестроено. Сама только что вставленная вершина, хотя и виновата в потере баланса, «козлом отпущения» стать не может — у неё ещё нет «детей», а значит её баланс идеален. Соответственно, нужно пройти по дереву от этой вершины к корню, пересчитывая веса для каждой вершины по пути. Если на

этом пути встретится вершина, для которой критерий α-сбалансированности по весу нарушился — мы полностью перестраиваем соответствующее ей поддерево так, чтобы восстановить α-сбалансированность по весу.

По ходу дела возникает ряд вопросов:

— А как пройти от вершины «вверх» корню? Нам нужно хранить ссылки на родителей?

— Нет. Поскольку мы пришли к месту вставки новой вершины из корня дерева — у нас есть стек, в котором находится весь путь от корня к новой вершине. Берём родителей из него.

— А как посчитать «вес» вершины — мы же его не храним в самой вершине?

— Нет, не храним. Для новой вершины вес = 1. Для её родителя вес = 1 (вес новой вершины) + 1 (вес самого родителя) + size(brother(x)).

— Ну ок, а как посчитать size(brother(x)) ?

— Рекурсивно. Да, это займёт время O(size(brother(x))). Понимая, что в худшем случае нам, возможно, придётся посчитать вес половины дерева — мы видим ту самую сложность O(n) в худшем случае, о которой говорилось в начале. Но поскольку мы обходим поддерево α-сбалансированного по весу дерева можно показать, что амортизированная сложность операции не превысит O(log n).

— А ведь вершин, для которых нарушился α-баланс может быть и несколько — что делать?

— Короткий ответ: можно выбирать любую. Длинный ответ: в оригинальном документе с описанием структуры данных есть пара эвристик, как можно выбрать «конкретного козла отпущения» (вот это терминология пошла!), но вообще-то они сводятся к «вот мы попробовали так и так, эксперименты показали, что вроде вот так лучше, но почему — объяснить не можем».

— А как делать балансировку найденной Scapegoat-вершины?

— Есть два пути: тривиальный и чуть более хитрый.

Тривиальная балансировка

1. Обходим всё поддерево Scapegoat-вершины (включая её саму) с помощью in-order обхода — на выходе получаем отсортированный список (свойство In-order обхода бинарного дерева поиска).
2. Находим медиану на этом отрезке, подвешиваем её в качестве корня поддерева.

3. Для «левого» и «правого» поддерева рекурсивно повторяем ту же операцию.

Видно, что всё это дело требует O(size(Scapegoat-вершина)) времени и столько же памяти.

Чуть более хитрая балансировка

Мы вряд ли улучшим время работы балансировки — всё-таки каждую вершину нужно «подвесить» в новое место. Но мы можем попробовать сэкономить память. Давайте посмотрим на «тривиальный» алгоритм внимательнее. Вот мы выбираем медиану, подвешиваем в корень, дерево делится на два поддерева — и делится весьма однозначно. Никак нельзя выбрать «какую-то другую медиану» или подвесить «правое» поддерево вместо левого. Та же самая однозначность преследует нас и на каждом из следующих шагов. Т.е. для некоторого списка вершин, отсортированных в возрастающем порядке, у нас будет ровно одно порождённое данным алгоритмом дерево. А откуда же мы взяли отсортированный список вершин? Из in-order обхода изначального дерева. Т.е. каждой вершине, найденной по ходу in-order обхода балансируемого дерева соответствует одна конкретная позиция в новом дереве. И мы можем эту позицию рассчитать в принципе и без создания самого отсортированного списка. А рассчитав — сразу её туда записать. Возникает только одна проблема — мы ведь этим затираем какую-то (возможно ещё не просмотренную) вершину — что же делать? Хранить её. Где? Ну, выделять для списка таких вершин память. Но этой памяти нужно будет уже не O(size(N)), а всего лишь O(log N).

В качестве упражнения представьте себе в уме дерево, состоящее из трёх вершин — корня и двух подвешенных как «левые» сыновья вершин. In-order обход вернёт нам эти вершины в порядке от самой «глубокой» до корня, но хранить в отдельной памяти по ходу этого обхода нам придётся всего одну вершину (самую глубокую), поскольку когда мы придём во вторую вершину, мы уже будем знать, что это медиана и она будет корнем, а остальные две вершины — её детьми. Т.е. расход памяти здесь — на хранение одной вершины, что согласуется с верхней оценкой для дерева из трёх вершин — log(3).

Удаление вершины

Удаляем вершину обычным удалением вершины бинарного дерева поиска (поиск, удаление, возможное переподвешивание детей). Далее проверяем выполнение условия size[T] < α * max_size[T], если оно выполняется — дерево могло потерять α-балансировку по весу, а значит нужно выполнить полную перебалансировку дерева (начиная с корня) и присвоить max_size[T] = size[T]

Насколько это всё производительно? Отсутствие оверхеда по памяти и балансировки при поиске — это хорошо, это работает на нас. С другой стороны, балансировки при модификациях не так чтобы уж очень дешевы. Ну и выбор α

существенно влияет на производительность тех или иных операций. Сами изобретатели сравнивали производительность Scapegoat-дерева с красно-черными и Splay-деревьями. У них получилось подобрать α так, что на случайных данных Scapegoat-дерево превзошло по скорости все другие типы деревьев, на любых операциях (что вообще-то весьма неплохо). К сожалению, на монотонно-возрастающем наборе данных Scapegoat-дерево работает хуже в части операций вставки и выиграть у Scapegoat не вышло ни при каком α.

3.4. Адаптивная структура данных - Splay дерево

3.9. Расширяемые деревья: адаптивные структуры данных

Идея адаптивных структур данных в том, что она адаптируется под запросы так, что на часто поступающие вопросы ответы будут приходить быстрее. Итак, адаптивная структура изменяет не только операции обновления, но также пока отвечает на запрос. Первое адаптивное дерево поиска было разработано Алленом и Мунро (1978), которые показали что дерево поиска второй модели, которое движется после каждого запроса запрашиваемого элемента к корню, поступающих в ряде независимых запросов, генерируемых согласно фиксированному распределению, только постоянный фактор хуже, чем оптимальное дерево поиска для этого распространения. Похожие структуры были также обнаружены Битнером (1979) и Мехлхорном (1979), чьи D-деревья сочетают адаптивность с вниманием к запросам с разумным поведением при обновлениях. D-деревья, кроме того, основаны на деревьях поиска (Бент, Слейтор, Тарьян 1985) и взвешенных AVL деревьях Вайшнави (Вайшнави 1987), достигли такой производительности также для индивидуальных операций с явно предоставленными вероятностями доступа, также поддерживая обновления для этих вероятностей.

Самая известная адаптивная структура - это splay дерево, изобретённое Слейтором и Тарьяном (1985); они также двигают запрашиваемый элемент вверх в слегка более запутанном способе и имеют несколько дополнительных адаптивных свойств. Другие структуры со схожими свойствами были также обнаружены (Макинен 1987; Хью и Мартел 1993; Шонмэйкерс 1993; Иаконо 2001), так же как и некоторые основные классы правил трансформации, которые генерируют те же свойства (Субраманиан 1996; Георгакопулус и МакКларкин 2004); также версии с блоками узлов схожих с B-деревьями (Мартел 1991; Шерк 1995).

Splay деревья имеют некоторые адаптивные свойства; возможно, само собой, что если запросы приходят согласно какому-то фиксированному распределению по набору ключей, затем ожидаемое время запроса для splay дерева является постоянным фактором, который хуже чем ожидаемое время запроса для дерева, которое оптимально для этого распространения. Конечно, как и пальцевые деревья, для того чтобы компенсировать потерю постоянного фактора, распределение должно быть далеко от однородного, иначе любое сбалансированное дерево имело бы это свойство.

Другое выдающееся свойство splay деревьев это то, что они просто не несут никакой балансировочной информации, ни в узлах, ни в каком-либо глобальном счётчике. Они просто следую простым трансформационным правилам, которые чудесным образом уравновешивают дерево, по крайней мере в амортизационном смысле. Splay деревья, в отличие от других деревьев в этой книге, это то что они обязательно следуют модели 2 для деревьев поиска, с объектами вместе с ключами в узлах. Для разных других

балансировочных критериев мы можем объединить их в иную модель, но это невозможно для стандартной модели splay деревьев. Адаптивность splay деревьев основывается на использовании факта во второй модели дерева, некоторые объекты сталкивались намного ранее, чем средняя глубина предположения. Также есть объект в корне, который, если запрошен, уже найден после двух сравнений. И splay дерево перемещает запрошенный объект в корень, производя некоторые перестановки по пути, так что если этот объект запрошен не так давно, то вскоре он будет в каком-то узле рядом с корнем.

Узел splay дерева содержит только ключ, указатель на ассоциируемый объект, и обычные правые и левые указатели; никакой балансировочной информации ненужно. Итак, структура такова:

```
typedef struct tr_n_t { key_t        key;
                        struct tr_n_t   *left;
                        struct tr_n_t  *right;
                        object_t      *object;
                      /* possibly other information */
                           } tree_node_t;
```

Левые и правые вращения должны быть адаптированы к тому, что они двигают не только ключ, но и указатель. Мы сохраняем условие отмечать листья, используя `NULL` в качестве правого указателя. Интервалы, ассоциированные с узлами стали открытыми интервалами. Операции вставки и удаления - это просто обычная вставка и удаление с соответствующими изменениями для этой модели дерева. Здесь нет балансировки после вставки или удаления; единственное место где структура дерева изменяется, заключается в операции поиска.

Правила поиска таковы: Сначала мы идём вниз, к узлу содержащим объект, отслеживая путь вверх. Позволим `current` первоначально обозначить этот узел. Мы повторяем следующие шаги, которые полностью отслеживают `current` узла, содержащего запрашиваемый объект, пока `current` не станет корнем, и мы вернем запрашиваемый объект.

1. Если `current` корень, мы возвращаем `current -> object.`
2. Если `current` имеет верхнего соседа `upper`

   Если `upper` корень и

   2.1 если `current=upper->left`,

   производим правое вращение в upper, устанавливая current в upper, и возвращая `current->object`,

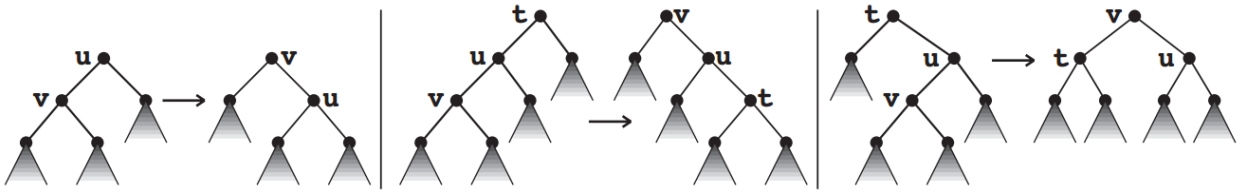
   2.2 иначе `current = upper->right`,

   производим левое вращение в upper, устанавливая current в upper, и возвращая `current->object`.
3. Иначе upper сам будем верхним соседом upper2.

   3.1 Если `current = upper->left` и `upper = upper2->right`, производим вращение вправо в `upper`, следуя левому вращению в `upper2`, и устанавливая `current` в `upper2`

   3.2 Если `current=upper->right` и `upper=upper->left`, производим вращение влево в upper, следуя правому вращению в `upper2`, устанавливая `current` до `upper2.`

Случаи 2.1 и 2.2 известны как “зиг”, 3.1 и 3.4 как “зиг-зиг”, и 3.2 и 3.3 как “зиг-заг” операции.



Операции уравновешивания 2.1, 3.1, и 3.2 в splay дереве

Мы должны показать, что эти операции реорганизовывают дерево, которое эффективно в амортизационном смысле. Мы также получим несколько таких результатов одним доказательством, выбрав другой вес функции. Вес функции ω определен на объектах с суммой всех весов, нормализованных до n и неотрицательных.

Для данного веса функции и дерева поиска, мы определяем несколько функций:

{ сумма весов s(n) узла n это сумма всех весов объекта поддерева ниже n;

{ ранг r(n) узла n это логарифм суммы веса: r(n)=log(s(n)); и

{потенциальный pot дерева - это сумма всех рангов узлов в дереве/

Сейчас главным инструментом будет следующая лемма, описывающая потенциальные изменения операции запроса. В последствии, мы используем pot_{before}, r_{before}, s_{before} и pot_{after}, r_{after}, s_{after} обзначающие соответсвенно ранг функции и вес сумы до и после балансировки.

Лемма 3.1 Если операция запроса доступа к узлу υ использовала *k* вращений, затем, мы имеем

$k+(pot_{after}-pot_{before})\le 1+3(r_{after}(v)-r_{before}(v))$.

Доказательство. Барансировка состоит из ряда операций, и сокращающейся структуры заявленного неравенства, необходимо доказать

$2+(pot_{after}-pot_{before})\le 3(r_{after}(v)-r_{before}(v))$

для любой операции типа 3.1, 3.2, 3.3, and 3.4, которые совершают n два вращения, каждый, и

$1+(pot_{after}-pot_{before})\le 1+3(r_{after}(v)-r_{before}(v))$

для операции типа 2.1 или 2.2, которые случаются по крайней мере однажды и которые совершают одно вращение.

{Для операций типа 2.1 и 2.2 позволим u быть верхним соседом v. Потому что $r_{before}(u)=r_{after}(v)$, $r_{after}(v) \ge r_{after}(u)$, and $r_{after}(v)\ge r_{before}(v)$, из заявленного неравенства следует:

$$\begin{aligned} pot_{after} - pot_{before} &= r_{after}(v) - r_{before}(v) + r_{after}(u) - r_{before}(u) \\ &= r_{after}(u) - r_{before}(v) \\ &\le r_{after}(v) - r_{before}(v) \\ &\le 3\,(r_{after}(v) - r_{before}(v))\,. \end{aligned}$$

{ Для операци типа 3.1 и 3.4, позволим u быть верхним соседом v и t, быть верхним соседом u. Затем замечаем это

$s_{before}(t)=s_{after}(v)\ge s_{before}(v)+s_{after}(t)$,

Итак,

$$
\begin{aligned}
&(r_{\text{before}}(\mathtt{v}) - r_{\text{after}}(\mathtt{v})) + (r_{\text{after}}(\mathtt{t}) - r_{\text{after}}(\mathtt{v})) \\
&\quad = \log\left(\frac{s_{\text{before}}(\mathtt{v})}{s_{\text{after}}(\mathtt{v})}\right) + \log\left(\frac{s_{\text{after}}(\mathtt{t})}{s_{\text{after}}(\mathtt{v})}\right) \\
&\quad \leq \max_{\substack{\alpha,\beta>0 \\ \alpha+\beta\leq 1}} (\log\alpha + \log\beta) \quad \leq -2.
\end{aligned}
$$

Используя это и $r_{before}(v) \leq r_{before}(u)$ и $r_{after}(u) \leq r_{after}(v)$, мы снова получим утверждённое неравенство:

$$
\begin{aligned}
pot_{\text{after}} - pot_{\text{before}} &= r_{\text{after}}(\mathtt{v}) + r_{\text{after}}(\mathtt{u}) + r_{\text{after}}(\mathtt{t}) \\
&\quad - r_{\text{before}}(\mathtt{v}) - r_{\text{before}}(\mathtt{u}) - r_{\text{before}}(\mathtt{t}) \\
&= r_{\text{after}}(\mathtt{u}) + r_{\text{after}}(\mathtt{t}) - r_{\text{before}}(\mathtt{v}) - r_{\text{before}}(\mathtt{u}) \\
&= 3\,(r_{\text{after}}(\mathtt{v}) - r_{\text{before}}(\mathtt{v})) \\
&\quad + (r_{\text{after}}(\mathtt{u}) - r_{\text{after}}(\mathtt{v})) + (r_{\text{after}}(\mathtt{t}) - r_{\text{after}}(\mathtt{v})) \\
&\quad + (r_{\text{before}}(\mathtt{v}) - r_{\text{after}}(\mathtt{v})) + (r_{\text{before}}(\mathtt{v}) - r_{\text{before}}(\mathtt{u})) \\
&\leq 3\,(r_{\text{after}}(\mathtt{v}) - r_{\text{before}}(\mathtt{v})) - 2.
\end{aligned}
$$

Это завершает доказательство леммы.

Сейчас мы можем использовать лемму для доказательства амортизированной границы на сложности любого ряда `find` операций. Сложность операций пропорционально числу сделанных вращений в этих операциях. Согласно лемма, количество вращений в единственной операции поиска ограничено потенциальным изменением дерева, плюс трёхкратное различие ранга корня, минус ранг запрошенного узла перед тем, как он стал новым корнем, плюс 1. Через ряд операций становится:

Количество операций $\leq \Sigma_{\text{операций}}(pot_{before} - pot_{after}) + \Sigma_{\text{операций}}(r(\text{корень}) - r_{before}(\text{запрашиваемый узел})) +$ количество операций. Первая сумма отображает сумму, которая сводится к потенциалу в начале отрицательным потенциалом в конце, и может быть ограничена независимым рядом операций, максимальным потенциалом дерева с данными весами минус минимальный потенциал такого дерева. Для амортизированной границы сложность `find` операции, это означает количество вращений, которое он совершает, и которое мы должны ограничить другой суммой.

Если мы дадим каждому n объекту дерева вес 1, то тогда вес корня будет n и вес любого узла будет равен как минимум 1. Итак ранги это числа между 0 и $\log n$, и разница в рангах не более $\log n$. Также у дерева есть n узлов, так что потенциал, сумма рангов между 0 и $n \log n$ и любой потенциальной разницы $O(n \log n)$. Это даёт амортизированную $O(\log n)$ границу.

Теорема: Любой ряд m `find` операций в splay дереве с n объектами занимает время $O(m \log n + n \log n)$.

Другая модель заключается в том, что запросы поступают в соответствии с некоторыми свойствами распространения $(p_i)_{i=1}^n$ в объекте. Затем мы даём объекту i вес $p_i n$. Снова сумма весов n, итак ранг корня $\log n$, и с возможностью p_i запрашиваемый объект будет иметь ранг $\log(p_i n) = \log(p_i) + \log n$, так что ожидаемая разница в ранге будет

$$
\sum_{i=1}^{n} p_i\,(\log n - \log(p_i n)) = -\sum_{i=1}^{n} p_i \log p_i =: H(p_1, \ldots, p_n),
$$

Которая является энтропией распространения. Максимальный и минимальный потенциал дерева с такими весами зависит от распространения $(p_i)^n_{i=1}$, и мы не имеем простой границы у них, но та максимальная разность потенциала это некоторое количество $\Delta pot_{max}(p_1,...,p_n)$ которое независимо от ряда `find` операций. Это накладывает следующее ограничение:

Теорема: Ожидаемая сложность последовательность *m* `find` операций в splay дереве, если запросы выбраны независимо и случайно согласно распространению $(p_i)^n_{i=1}$ $O(\Delta pot_{max}(p_1, \dots , p_n) + m(1+H(p_1, \dots ,p_n)))$

Но энтропия $H(p1, \dots , p_n)= -\Sigma^n_{i=1} p_i \log p_i$ существенным образом ожидаема глубока для оптимального дерева с данным распределением. Это нижняя граница даже в слабой модели, когда мы используем дерево модели 1, разрешено изменять порядок ключей и только для сохранения возможности распространения; эта ситуация с кодом переменной длины и нижняя граница последствие неравенства Крафта. В этой модели, та глубина, плюс единица, может быть достигнута деревьями Хаффмана или Шэннон-Фаер. Изменяя модель дерева 1 в модель 2, мы теряем не больше чем второй фактор, потому что каждая вторая модель дерева может быть трансформирована в модель дерева 1, путём замены каждого узла модели 2 двумя узлами модели 1. Постройка оптимальной или около-оптимального поискового дерева, особенно модели 2, была особо изучаемым предметом (смотри Кнут (1973) или Мехлхорн (1979) для различных ссылок). Итак splay деревья нуждаются в средне-ожидаемом времени доступа внутри постоянного фактора оптимально-ожидаемого доступа для этого распространения, для чего $H (p_1, \dots , p_n)$ является нижней границей. Splay деревья достигают этого путём адаптации к ряду запросов, без знания распространения. Мы использовали распространения только в анализе значения веса функции, не алгоритма.

Еще одна модель адаптивности - это пальцевый поиск. Splay деревья поддерживают пальцевый поиск без знания пальца. Считая фиксированный элемент `finger` и присваивают каждому элементу *x* вес, $\frac{n}{\text{расстояние}(finger,\ x)^2+1}$, где расстояние (`finger, x`) отмечает количество элементов между `finger` и `x`. Затем вес суммы Θ(n), потому что $\sum_{v=1}^{\infty} \frac{1}{v^2} = \frac{\pi^2}{6} < \infty$, итак ранг дерева log *n* – O(1) и ранг запрашиваемого элемента `q`

$$\log\left(\frac{n}{\text{distance}(\texttt{finger}, \texttt{q})^2 + 1}\right) = \log n - O(\log(\text{distance}(\texttt{finger}, \texttt{q}))).$$

Итак расщличие ранга O(log(расстояние(`finger, q`))). Потому что каждый узле будет иметь ранг между log *n* и $\log\frac{n}{(n-1)^2+1} > -\log n$, ток любое различие потенциалов O(n log n). Это влияет на следующее:

Теорема: Ряду m операций поиска для элементов $q_1, \dots , q_m$ в splay дереве с n элементами требуется время

$$O(n \log n + \sum_{i=1}^{m} \log(\text{distance}(\texttt{finger}, \texttt{q}_i))).$$

Итак splay дерево адаптируется к безформенности или локольностью запросов для фиксированной установки, косвенным образом исключая операции обновления. Мы можем произвести обновления путём базовой вставки и удаления, возможно следуя тому же движению к лучшим запросам. И если мы используем постоянную весовую единицу, к которой применяется амортизационный анализ, потому что на самом деле нет никаких различий между запросами вставки или удаления. Для адаптивного анализа, однако, даже модель становится менее понятной, потому что мы не можем изменить вес функции, когда текущая установка изменилась.

Мы наконец даём код для `find` в splay деревьях вместе с базовой вставкой и удалением для этих деревьев модели 2. Наши условия нуждаются в изменении для этого дерево-узловой модели; потому что каждый узел содержит объект вместе с ключом, вращения нужны для того чтобы передвигать объект и ключ, и мы используем NULL указатель для поля объекта, чтобы зашифровать пустое дерево. Удаление более запутанно, чем в предпочитаемом листьевой модели дерева, потому что ключи из внутренних узлов могут быть удалены; в таком случае, необходимо передвинуть другой ключ вверх и заменить

```
object_t *find(tree_node_t *tree,
               key_t query_key)
{  int finished = 0;
   if( tree->object == NULL )
     return(NULL); /* tree empty */
   else
   {  tree_node_t *current_node;
      create_stack();
      current_node = tree;
      while( ! finished )
      {  push( current_node );
         if( query_key < current_node->key
               && current_node->left != NULL )
            current_node = current_node->left;
         else if( query_key > current_node->key
               && current_node->right != NULL )
            current_node = current_node->right;
         else
            finished = 1;
      }
      if( current_node->key != query_key )
         return( NULL );
      else
      {  tree_node_t *upper, *upper2;
         pop(); /* pop the node containing
         the query_key */
         while( current_node != tree )
         {  upper = pop(); /* node
            above current_node */
            if( upper == tree )
            {  if( upper->left == current_node )
```

его.

```
                  right_rotation( upper );
               else
                  left_rotation( upper );
               current_node = upper;
            }
            else
            {  upper2 = pop(); /* node
               above upper */
               if( upper == upper2->left )
               {  if( current_node ==
                  upper->left )
                     right_rotation( upper2 );
                  else
                     left_rotation( upper );
                  right_rotation( upper2 );
               }
               else
               {  if( current_node ==
                  upper->right )
                     left_rotation( upper2 );
                  else
                     right_rotation( upper );
                  left_rotation( upper2 );
               }
               current_node = upper2;
            }
         }
      return( current_node->object );
      }
   }
}


int insert(tree_node_t *tree, key_t new_key,
object_t *new_object)
{  tree_node_t *tmp_node, *next_node;
   if( tree->object == NULL )
   {  tree->object =  new_object;
      tree->key  = new_key;
```

```
      tree->left   = NULL;
      tree->right  = NULL;
   }
   else /* tree not empty: root contains a key */
   {  next_node = tree;
      while( next_node != NULL )
      {   tmp_node = next_node;
          if( new_key < tmp_node->key )
             next_node = tmp_node->left;
          else if( new_key > tmp_node->key )
             next_node = tmp_node->right;
          else /* new_key == tmp_node->key:
          key already exists */
             return(-1);
      }
      /* next_node == NULL. This should
      point to new leaf */
      {  tree_node_t *new_leaf;
         new_leaf = get_node();
         new_leaf->object =  new_object;
         new_leaf->key = new_key;
         new_leaf->left  = NULL;
         new_leaf->right = NULL;
         if( new_key < tmp_node->key )
            tmp_node->left  = new_leaf;
         else
            tmp_node->right = new_leaf;
      }
   }
   return( 0 );
}

object_t *delete(tree_node_t *tree,
key_t delete_key)
{  tree_node_t *tmp_node, *upper_node,
   *next_node, *del_node;
   object_t *deleted_object;
   if( tree->object == NULL )
     return( NULL ); /* delete from empty tree */
```

```
else
{  next_node = tree; tmp_node = NULL;
   while( next_node != NULL )
   {  upper_node = tmp_node;
      tmp_node = next_node;
      if( delete_key < tmp_node->key )
         next_node = tmp_node->left;
      else if( delete_key > tmp_node->key )
         next_node = tmp_node->right;
      else /* delete_key == tmp_node->key */
         break; /* found delete_key */
   }
   if( next_node == NULL )
      return( NULL );
      /* delete key not found */
   else /* delete tmp_node */
   {  deleted_object = tmp_node->object;
      if( tmp_node->left == NULL
      && tmp_node->right == NULL )
      {  /* degree 0 node: delete */
         if( upper_node != NULL )
         {  if( tmp_node == upper_node->left )
               upper_node->left = NULL;
            else
               upper_node->right = NULL;
            return_node( tmp_node );
         }
         else /* delete last object,
         make tree empty */
            tmp_node->object = NULL;
      }
      else if ( tmp_node->left == NULL )
      {  tmp_node->left =
         tmp_node->right->left;
         tmp_node->key  =
         tmp_node->right->key;
         tmp_node->object =
         tmp_node->right->object;
         del_node = tmp_node->right;
```

```
            tmp_node->right =
            tmp_node->right->right;
            return_node( del_node );
         }
         else if ( tmp_node->right == NULL )
         {  tmp_node->right =
            tmp_node->left->right;
            tmp_node->key  = tmp_node->left->key;
            tmp_node->object =
            tmp_node->left->object;
            del_node = tmp_node->left;
            tmp_node->left =
            tmp_node->left->left;
            return_node( del_node );
         }
         else /* interior node needs to
         be deleted */
         {  upper_node = tmp_node;
            del_node = tmp_node->right;
            while( del_node->left != NULL )
            {  upper_node = del_node;
               del_node = del_node->left;
            }
            tmp_node->key = del_node->key;
            tmp_node->object = del_node->object;
            if( del_node = tmp_node->right )
               tmp_node->right = del_node->right;
            else
               upper_node->left =
               del_node->right;
            return_node( del_node );
         }
         return( deleted_object );
      }
   }
}
```

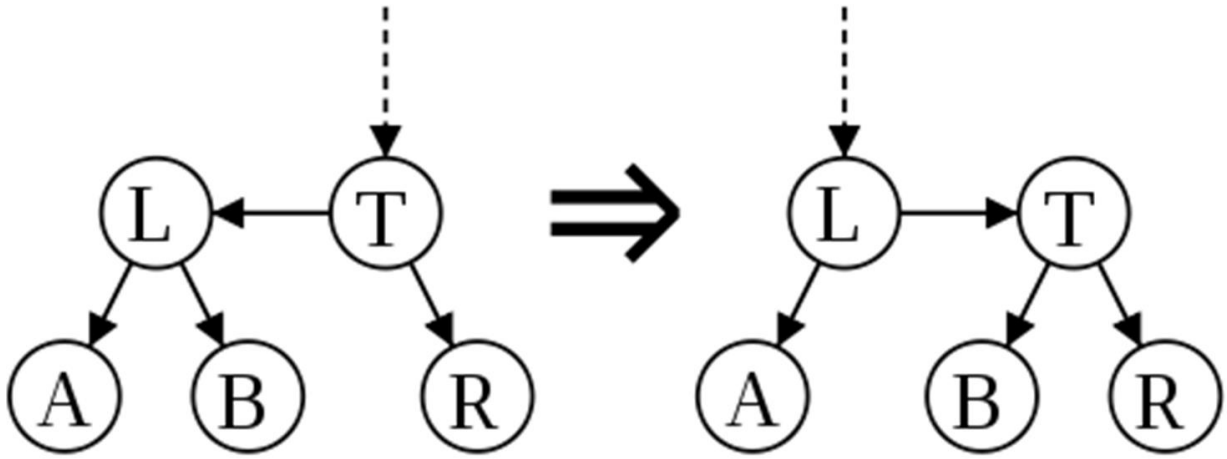
Здесь мы не можем использовать основанный на массиве стек, потому что глубина элемента может быть n-1, в худшем случае. Мы должны использовать одну из реализация связного списка для стека. На самом деле, используя указатели назад вместо стека, для отслеживания пути, будут предпочтительней, но мы не можем заявить, что мы не используем дополнительное пространство в узлах для балансировки.

3.5. AA tree AA-дерево Arne Andersson

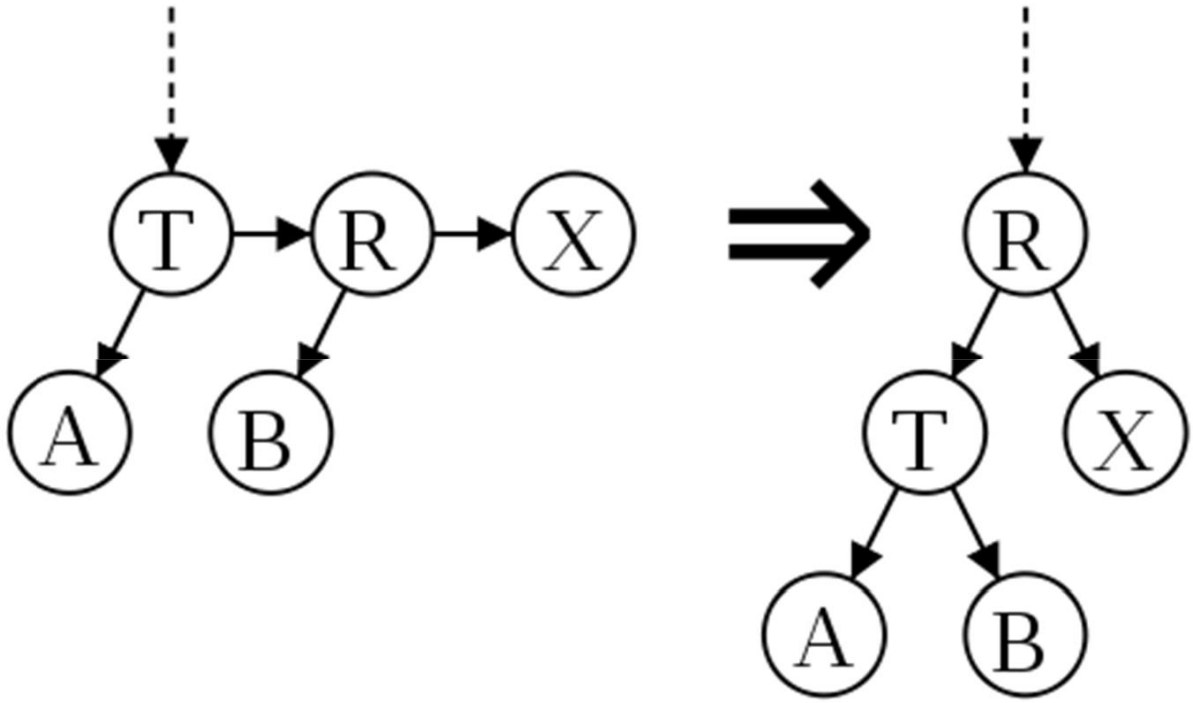
Для упрощения балансировки дерева вводится понятие уровень (level) вершины (уровень любой листовой вершины равен 1). Правило, которому должно удовлетворять АА-дерево: к одной вершине можно присоединить другую вершину того же уровня но только одну и только справа. После добавления узла поднимаемся вверх по дереву и выполняем балансировку (операции skew и split для каждой вершины)

АА-дерево :: skew Устранение левой связи на одном уровне

Замечание: горизонтальная стрелка обозначает связь между вершинами одного уровня, а наклонная (вертикальная) — между вершинами разного уровня.



AA-дерево :: split Устранение двух правых связей на одном уровне



Замечание: горизонтальная стрелка обозначает связь между вершинами одного уровня, а наклонная (вертикальная) — между вершинами разного уровня.

AA-дерево :: erase

После удаления узла необходимо подняться вверх начиная с родителя фактически удаленного узла и выполнить балансировку. Для этого необходимо проверить уровень вершины и если он на 2 больше чем у потомков, то снизить уровень на 1. Если после этого уровень правого потомка больше уровня в узле, то сделать уровень

правого потомка равным уровню текущего узла. Так как изменение уровней могло вызвать нарушение правила построения дерева, необходимо осуществить операцию skew для текущего узла, потом для правого потомка, потом для правого потомка правого потомка текущего узла и операцию split для текущего узла и его правого потомка

3.6. Treap

17. Декартово дерево

Декартово дерево (cartesion tree) были впервые рассмотрены Жаном Вуйлеменом (Jean Vuillemin) в 1980 году. Слово *treap* было впервые использовано Эдвардом МакКрайтом (Edward McCreight) в 80-х годах прошлого века, чтобы описать немного отличающуюся структуру данных, но позднее он поменял имя *treap* на *PST* (*priority search trees*). Декартовые деревья были заново изучены и использованы Арагоном (Cecilia Aragon) и Зайделем (Raimund Seidel) для построения рандомизированных деревьев поиска в 1989 году [12]. Другая модификация рандомизированных двоичных деревьев поиска, которая использует рандомную перебалансировку вместо приоритетов, была позднее проанализирована Мартинесом (Conrado Martınez) и Роурой (Salvador Roura) в 1996 году [14].

17.1. Определение и терминология.

Декартово дерево - это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) и дерамида (дерево+пирамида). В русском языке было предложено ещё одно аналогичное слову *treap* по построению слово *дуча*[1] (*дерево+куча*).

Более строго, это структура данных, которая хранит пары (X,Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска (*BST - binary search tree*)по x и бинарной пирамидой по y. Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0,Y_0), то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y \leq Y_0$.

Каждый узел этого дерева представляет собой запись, состоящую из следующих полей:

- **ссылка на левое поддерево;**
- **ссылка на правое поддерево;**
- **ссылка на родительский узел (необходима только для некоторых алгоритмов, например, для линейного алгоритма построения дерева);**
- **ключ X - ключ поиска;**
- **ключ Y - приоритет.**

Другими словами, декартовое дерево – это одновременно и *BST* для ключей и *heap* для приоритетов.

Строение декартового дерева полностью определяется ключами и их приоритетами, если они различны. Так как это пирамида, то вершина *V* с её наименьшим приоритетом будет находиться в корне. Так как это *BST*, то вершины *U*, у которых *key(U) < key(V),* лежат в левом поддереве, а остальные находятся в правом. Наконец, так как поддеревья являются декартовыми деревьями, то по индукции по высоте получаем, что структура декартового дерева полностью определена (Рис. 1). Базой является пустое декартовое дерево.

Другой способ для описания структуры основывается на том, что декартовое дерево – это *BST*, которое получается вставкой ключей в порядке возрастания их приоритетов.

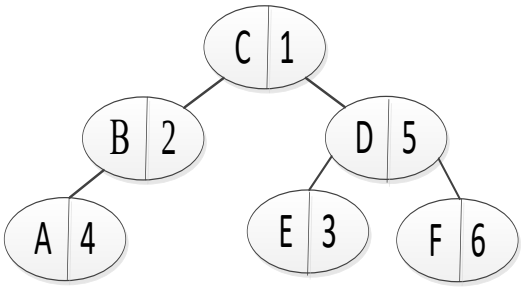


Рис. 1. Пример декартового дерева (буквами представлены ключи, а числами приоритеты).

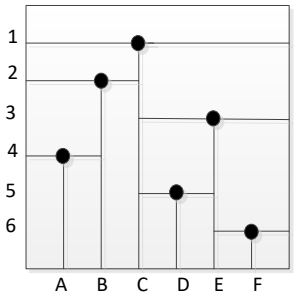


Рис. 2. Геометрическая интерпретация декартового дерева, показанного на рис. 1.

Третий вариант для описания декартового дерева отождествляет ключи и приоритеты с координатами множества точек на плоскости. Корень декартового дерева соответствует точке, которая лежит на самом верху. Эта точка делит плоскость на три части. Верхняя часть по определению пуста; левая и правая разделяются рекурсивно. Эта интерпретация представляет интерес для приложений в компьютерной геометрии [1].

Случайные приоритеты. Рандомизированное дерево двоичного поиска – это декартовое дерево, в котором приоритеты задаются случайной функцией. Это значит, что при добавлении нового ключа, мы придумываем рандомное вещественное число между 0 и 1, которое будет приоритетом новой вершины. Использование вещественных чисел обусловлено необходимостью отсутствия одинаковых приоритетов у разных вершин. Это требуется для анализа структуры декартового дерева.

На практике мы можем выбирать случайные целые числа из большого промежутка или использовать числа с плавающей точкой. Так как приоритеты

независимы, то вероятность того, что какая-то вершина будет иметь минимальный приоритет, будет одинакова для всех вершин.

Стоимость всех основных операций, как было показано выше, пропорциональна глубине какой-то вершины в декартовом дереве. Доказано, что средняя глубина каждой вершины составит *O(log n)* [11]. Это показывает, что ожидаемое время работы каждой из операций будет *O(log n)*.

Так как декартово дерево – это двоичное дерево, которое получается добавлением ключей в порядке возрастания приоритетов, то рандомизированное декартово дерево получается включением ключей в случайном порядке. Получается, что проведённый анализ автоматически даёт нам ожидаемую глубину произвольной вершины в обыкновенном двоичном дереве, построенном случайными вставками (без использования приоритетов).

Простейший алгоритм построения декартового дерева. Простейший для понимания алгоритм построения декартового дерева по множеству данных пар (x, y) выглядит следующим образом. Упорядочим все пары по ключу x и пронумеруем получившуюся последовательность ключей y: y(1), y(2), y(3), …, y(n).

Алгоритм построения декартового дерева основан на рекурсии: находим в последовательности минимальный y и назначаем его корнем. Найденный y разбивает последовательность на две части, для каждой из частей запускаем алгоритм построения декартового дерева.

Из рассмотренного алгоритма следует, что множество пар (x, y) однозначно определяет структуру декартового дерева. Заметим для сравнения, что множество ключей, которые хранятся в двоичном дереве поиска, не определяют однозначно структуру дерева. То же самое касается двоичной пирамиды — какова будет структура двоичной пирамиды (как ключи распределятся по узлам) зависит не только от самого множества ключей, но и от последовательности их добавления. В декартовом дереве такой неоднозначности нет.

Линейный алгоритм построения декартового дерева. Этот алгоритм построения дерева также основан на рекурсии. Только теперь мы последовательно будем добавлять элементы y и перестраивать дерево. Дерево T(y(1), …, y(k+1)) будет строиться из дерева T(y(1), …, y(k)) и следующего элемента y(k+1).

T(y(1), ..., y(k+1)) = F (T(y(1), ..., y(k)), y(k+1))

На каждом шаге будем помнить ссылку на последний добавленный узел. Он будет самым правым. Действительно, мы упорядочили ключи y по прикреплённому к ним ключу x. Так как декартово дерево — это дерево поиска, то после проекции на горизонтальную прямую ключи x должны возрастать слева направо. Самый правый узел всегда имеет максимально возможное значение ключа x.

Функция F, которая отображает декартово дерево T(y(1), …, y(k)) предыдущего шага и очередное y(k+1) в новое дерево T(y(1), …, y(k+1)),

выглядит следующим образом. Вертикаль для узла y(k+1) определена. Нам необходимо определиться с его горизонталью. Для начала мы проверяем, можно ли новый узел y(k+1) сделать правым ребёнком узла y(k) — это следует сделать, если y(k+1) > y(k). Иначе мы делаем шаг по склону от узла y(k) вверх и смотрим на значение y, которое там хранится. Поднимаемся вверх по склону, пока не найдём узел, в котором значение y меньше, чем y(k+1), после чего делаем y(k+1) его правым ребёнком, а его предыдущего правого ребёнка делаем левым ребёнком узла y(k+1).

Это алгоритм амортизационно (в сумме за все шаги) работает линейное (по числу добавляемых узлов) время. Действительно, как только мы «перешагнули» через какой-либо узел, поднимаясь вверх по склону, то мы его уже никогда не встретим при добавлении следующих узлов. Таким образом, суммарное число шагов вверх по склону не может быть больше общего числа узлов.

В то же время, приоритеты позволяют однозначно указать дерево, которое будет построено (разумеется, не зависящее от порядка добавления элементов) (это доказывается соответствующей теоремой). Теперь очевидно, что если выбирать приоритеты случайно, то этим мы добьёмся построения невырожденных деревьев в среднем случае, что обеспечит асимптотику O (log N) в среднем. Отсюда и понятно ещё одно название этой структуры данных - *рандомизированное бинарное дерево поиска*.

17.2. Операции над декартовым деревом

Для реализации операций над декартовым деревом понадобится реализовать две вспомогательные операции: Split и Merge.

Split (T, X) - разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньше ключа X, а R содержит все элементы, больше X. Эта операция выполняется за O (log N). Реализуется с помощью рекурсии.

Merge (T_1, T_2) - объединяет два поддерева T_1 и T_2, и возвращает это новое дерево. Эта операция также реализуется за O (log N). Она работает в предположении, что T_1 и T_2 обладают соответствующим порядком (все значения в первом меньше значений во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y. Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

Итак, над декартовым деревом выполняются следующие операции:

Insert (X, Y) - за O (log N) в среднем. Выполняет добавление в дерево нового элемента. Возможен вариант, при котором значение приоритета Y не передаётся функции, а выбирается случайно (правда, нужно учесть, что оно не должно совпадать ни с каким другим Y в дереве). Реализация Insert (X, Y). Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по X),

но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше Y. Мы нашли позицию, куда будем вставлять наш элемент. Теперь вызываем Split(X) от найденного элемента (от элемента вместе со всем его поддеревом), и возвращаемые ею L и R записываем в качестве левого и правого сына добавляемого элемента.

Search (X) - за O (log N) в среднем. Ищет элемент с указанным значением ключа X. Реализуется так же, как и для обычного бинарного дерева поиска.

Erase (X) - за O (log N) в среднем. Ищет элемент и удаляет его из дерева. Реализация Erase (X). Спускаемся по дереву (как в обычном бинарном дереве поиска по X), ища удаляемый элемент. Найдя элемент, мы просто вызываем Merge от его левого и правого сыновей, и возвращаемое ею значение ставим на место удаляемого элемента.

Build (X_1, ..., X_N) - за O (N). Строит дерево из списка значений. Эту операцию можно реализовать за линейное время (в предположении, что значения X_1, ..., X_N отсортированы), но здесь эта реализация рассматриваться не будет. Здесь будет использоваться только простейшая реализация - в виде последовательных вызовов Insert, т.е. за O (N log N).

Union (T_1, T_2) - за O (M log (N/M)) в среднем. Объединяет два дерева, в предположении, что все элементы различны (впрочем, эту операцию можно реализовать с той же асимптотикой, если при объединении нужно удалять повторяющиеся элементы).

Иногда нам требуется разделить декартовое дерево *T* на 2 декартовых дерева *T1* и *T2* относительно какого-то ключа *K* так, что все ключи в *T1* меньше *K*, а все ключи в *T2* больше. Простейший способ сделать это – добавить вершину с ключом *K* и приоритетом -∞. После вставки новая вершина станет корнем декартового дерева, а левое и правое поддерево будут представлять собой *T1* и *T2* соответственно. Время для разделения декартового дерева примерно равно времени двух неудачных поисков для *K*.

Подобным образом нам бы хотелось научиться объединять декартовые деревья *T1* и *T2*, где любой ключ из *T1* меньше любого ключа из *T2*. Объединение подобно разделению наоборот – нужно создать мнимую вершину с поддеревьями *T1* и *T2*, поворотами сделать её листом и удалить.

Так как глубина вершины в декартовом дереве в худшем случае составляет $\Theta(n)$, где n – общее количество вершин, то каждая из операций в худшем случае будет выполняться за $\Theta(n)$.

Intersect (T_1, T_2) - за O (M log (N/M)) в среднем. Находит пересечение двух деревьев (т.е. их общие элементы). Реализацию этой операции предлагается рассмотреть самостоятельно.

Кроме того, за счёт того, что декартово дерево является и бинарным деревом поиска по своим значениям, к нему применимы такие операции, как нахождение K-го по величине элемента, и, наоборот, определение номера элемента. Для удобства необходимо для каждой вершины хранить количество вершин в её поддереве. Если дерево поддерживает отсортированный массив, то

это доступ к элементам с индексом K. Отсортированный массив можно получить с помощью симметричного обхода (левое, корень, правое).

Как полноценное решение задачи расценивать такой подход не стоит, ведь он требует O(N) времени, а это нежелательно. Можно попытаться его улучшить, Будем хранить в каждой вершине размер её поддерева. На рис. 17.3 показано дерево с размерами поддеревьев, проставленными у каждой вершины.

Найдем теперь K-й элемент в индексации, начинающейся с нуля (!).

Алгоритм:

1. Рассмотрим корень дерева и размер его левого поддерева SL, размер правого даже не понадобится.
2. Если SL = K, то искомый элемент мы нашли, и это — корень.
3. Если SL > K, то искомый элемент находится где-то в левом поддереве, спускаемся туда и повторяем процесс.
4. Если SL < K, то искомый элемент находится где-то в правом поддереве. Уменьшим K на число SL+1, чтобы корректно реагировать на размеры поддеревьев справа, и повторим процесс для правого поддерева.

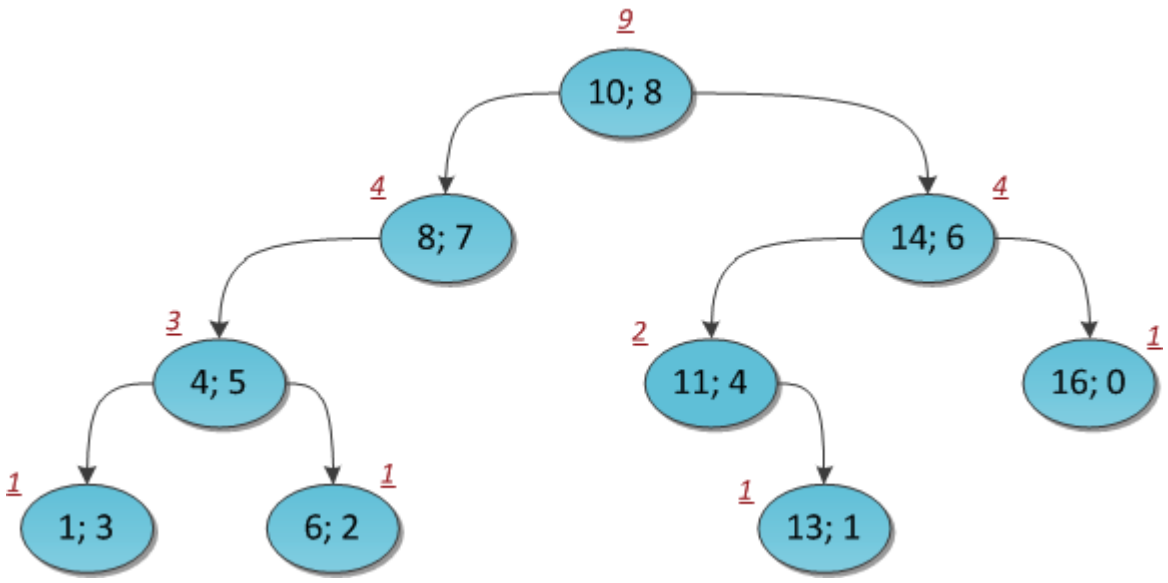


Рис 17.3. Декартово дерево с размерами поддеревьев

Промоделирую этот поиск для K = 6 на этом дереве:

Вершина (10; 8), SL = 4, K = 6. Идем вправо, уменьшая K на 4+1=5. Вершина (14; 6), SL = 2, K = 1. Идем влево, не меняя K.

Вершина (11; 4), SL = 0 (нет левого сына), K = 1. Идем вправо, уменьшая K на 0+1=1.

Вершина (13; 1), SL = 0 (нет левого сына), K = 0. Ключ найден.

Ответ: ключ под индексом 6 в декартовом дереве равен 13.

В функциональном языке можно написать решение через рекурсию.

Время выполнения поиска K-го элемента $O(\log_2 N)$. Ключевым вопросом все так же остается тот факт, что мы до сих пор не знаем, как поддерживать в

дереве корректные значения размера поддеревьев. Ведь после первого же добавления в дерево нового ключа все эти числа пойдут прахом, а пересчитывать их заново — O(N)! Впрочем, нет. O(N) это заняло бы после какой-нибудь операции, которая полностью перестроила дерево в структуру непонятной конструкции. А здесь добавление ключа, которое действует аккуратнее и не задевает все дерево, а только его маленькую часть. Значит, можно упростить. Если приспособить две основные функции Split и Merge под поддержку дополнительной информации в дереве — в данном случае размеров поддеревьев — то все остальные операции автоматически станут выполняться корректно, ведь своих изменений в дерево они не вносят.

Пересчет величины в вершине выделяется в отдельную функцию. Это базовая функция, на которой держится функционал всего декартова дерева. Для примера назовём такое новое поле узла декартового дерева Cost.

Итак, пусть нам на вход постоянно поступают (а порою удаляются) ключи x, и с каждым из них связана соответствующая цена — Cost. И вам нужно поддерживать во всей этой каше быстрые запросы на максимум цены. Можно спрашивать максимум во всей структуре, а можно только на каком-то её подотрезке: скажем, пользователя может интересовать максимальная цена за 2007 год (если ключи связаны со временем, то это можно интерпретировать как запрос максимума цены на множестве таких элементов, где $A \le x < B$).

Та же ситуация и с минимумом, и с суммой, и с какими-то булевыми характеристиками элемента (так называемой «окрашенностью» или «помеченностью»).

Запросы на подотрезках тоже не представляют труда, если опять вспомнить свойство бинарного дерева поиска. Ключ каждой вершины больше всех ключей левого поддерева и меньше всех ключей правого поддерева. Поэтому можно виртуально считать, что с каждой вершиной ассоциирован какой-то интервал ключей, которые могут в теории встретиться в ней и в её поддереве. Так, у корня это интервал $(-\infty; +\infty)$, у его левого сына $(-\infty; x)$, правого - $(x; +\infty)$, где x - значение ключа в корне. Все интервалы открыты с обеих сторон, X среди ключей правого поддерева встретиться не может. Если позволить в дереве одинаковые ключи и, как в первой части, бросать их все в одно и то же поддерево — скажем, в левое, — то интервалом для левого сына станет $(-\infty; x]$.

На рис. 17.4 показан соответствующий интервал для каждой вершины декартового дерева.

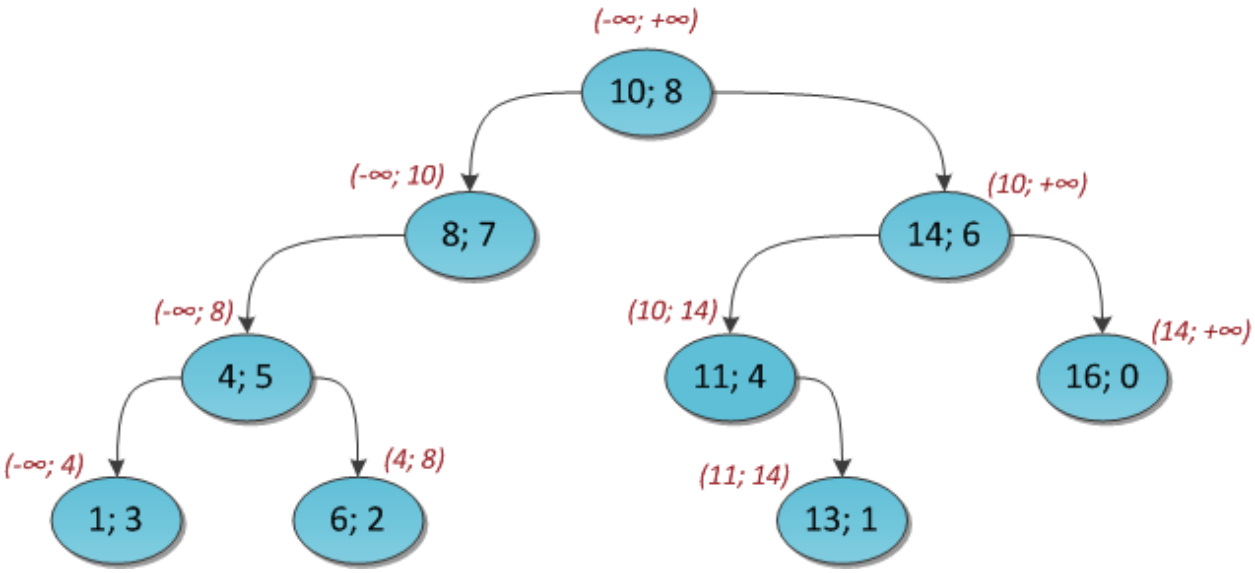


Рис. 17.4. декартовое дерево с соответствующими интервалами для каждой вершины.

Теперь ясно, что параметр, хранящийся в вершине, отвечает за значение соответствующей характеристики (максимума, суммы и т.д.) для всей ключей на интервале этой вершины. И мы можем отвечать на запросы по интервалам [A; B). Кто-то, знакомый с деревом отрезков, может подумать, что здесь стоит применить тот же рекурсивный спуск, локализируя постепенно текущую вершину до такой, которая полностью соответствует нужному интервалу или его кусочку. Но мы поступим проще и быстрее, сведя задачу к предыдущей.

Разделим дерево сначала по ключу A-1. Правый результат Split, хранящий все ключи, большие либо равные A, снова разделим — на этот раз по ключу B. В середине мы получим дерево со всеми элементами, у которых ключи принадлежат искомому интервалу. Для выяснения параметра достаточно посмотреть на его значение в корне — ведь всю чёрную работу по восстановлению справедливости для каждого дерева функция Split уже сделала за нас.

Время работы запроса $O(\log_2 N)$: два выполнения Split.

Отложенные вычисления. Сегодня очень важно иметь возможность изменения пользовательской информации по ходу жизни дерева. Понятно, что после изменения значения Cost в какой-то вершине все наши прежние параметры, накопившие ответы для величин в своих поддеревьях, уже недействительны. Можно, конечно, пройтись по всему дереву и пересчитать их заново, но это снова O(N).

Если речь идет о простом изменении Cost в одной-единственной вершине, то это не такая уж и проблема. Сначала мы, двигаясь сверху вниз, находим нужный элемент. Меняем в нем информацию. А потом, двигаясь обратно снизу вверх к корню, просто пересчитываем значения параметров стандартной функцией — ведь ни на какие другие поддеревья это изменение не повлияет, кроме как на те, которые мы посетили по пути от корня к вершине. Время работы равно $O(\log_2 N)$.

Сложнее, если надо поддерживать множественные операции. Пускай есть у нас декартово дерево, в каждой его вершине хранится пользовательская информация Cost. И мы хотим к каждому значению Cost в дереве (или поддереве — см. рассуждения об интервалах) прибавить какое-то одно и то же число A. Это пример множественной операции, в данном случае добавления константы на отрезке. Простого прохода к корню не достаточно.

Давайте заведем в каждой вершине дополнительный параметр, назовем его Add. Его суть следующая: он будет сигнализировать о том, что всему поддереву, растущему из данной вершине, полагается добавить константу, лежащую в Add. Получается такое себе запаздывающее прибавление: при необходимости изменить значения на A в некотором поддереве мы изменяем в этом поддереве только корневой Add, как бы давая обещание потомкам, что «когда-нибудь в будущем вам всем полагается еще дополнительное прибавление, но мы его пока выполнять фактически не будем, пока не потребуется».

Тогда запрос Cost из корня дерева должен совершить еще одно дополнительное действие, прибавив к Cost корневой Add, и полученную сумму расценивать как фактический Cost, будто бы лежащий в корне дерева. То же самое с запросами дополнительных параметров, к примеру, суммы цен в дереве: у нас есть корректно (!) посчитанное значение SumTreeCost в корне, которое хранит сумму всех элементов дерева, но не учтя того, что ко всем этим элементам нам полагается прибавить некий Add. Для получения истинно правильного значения суммы с учетом всех отложенных операций достаточно прибавить к SumTreeCost значение Add, умноженное на Size — количество элементов в поддереве.

Не очень понятно, что делать со стандартными операциями декартова дерева — Split и Merge — и когда нам потребуется все-таки выполнить обещание и добавить потомкам обещанный им Add. Эти вопросы рассмотрены в [15].

Множественные операции, конечно же, не ограничиваются одним только прибавлением на отрезке. Можно также на отрезке «красить» — устанавливать всем элементам булевый параметр, изменять — устанавливать все значения Cost на отрезке в одно значение, и так далее, что только придумает фантазия программиста. Главное условие на операцию — чтобы ее можно было за O(1) протолкнуть вниз от корня к потомкам, передав отложенное обещание чуть ниже по дереву, как мы и поступали с Merge и Split. Ну и, конечно же, информация должна легко восстанавливаться из обещания во время запроса.

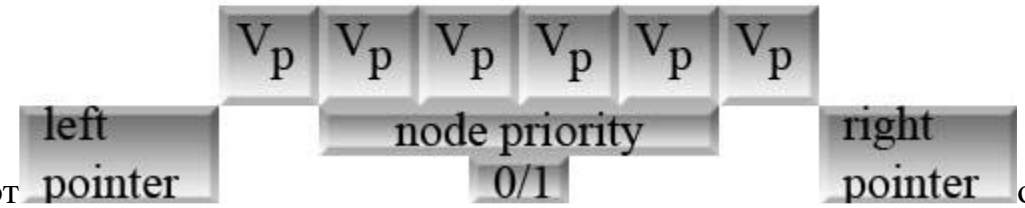
17.4. Модификации декартовых деревьев.

17.4.1. T-Treaps

Структура. Целью создания *T-Treap* была попытка совместить в одном АТД свойство сбалансированности, используя случайные приоритеты, как в *treap*, а так же кластеризацию данных в одной вершине для лучшей организации памяти. С каждым *T-Treap*’ом мы будем связывать вместимость

его вершин, которая показывает минимальное и максимальное количество значений в вершине. Листьям разрешено иметь меньшее количество значений в вершине, чем установлено. *T-Treap* реализует основные операции словаря с множеством ключей.

Каждая вершина *T-Treap*'а содержит соседние по значению ключи, ссылки на правого и левого ребёнка. Внутри каждой вершины вместе с самим ключом хранится его индивидуальный приоритет. Кроме того, у каждой вершины имеется направляющий бит и приоритет самой вершины,



который используется для поворотов.

Рис. 17.5. Представление вершины в *T-Treap*'е: *V* – значения, *p* – их приоритеты.

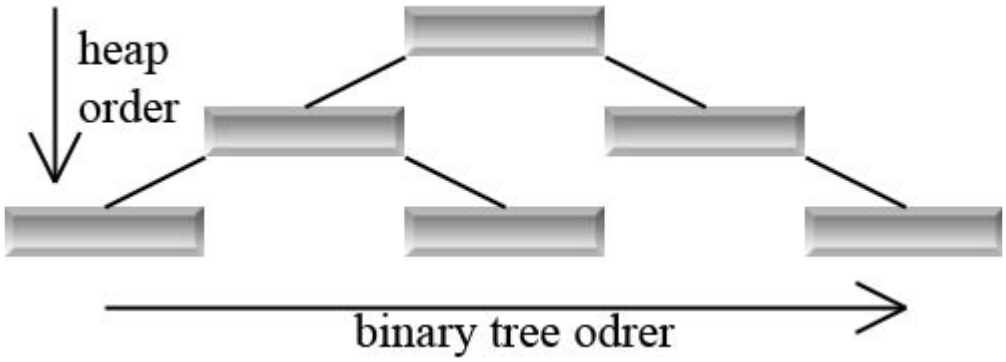


Рис. 17.6. Общее представление *T-Treap*'а.

Так как значения в вершинах упорядочены по "горизонтали", а приоритеты – по "вертикали", то необходимы дополнительные затраты для поддержания такой сложной структуры.

Операции BST

• ***Добавление*** происходит так же, как и в *BST*. Прежде всего, каждому новому ключу придумывается случайный приоритет. Потом, начиная с корня, значение ключа сравнивается с минимальным и максимальным значением в каждой посещённой вершине. Операция добавления проходит по *T-Treap*'у, останавливаясь, когда ключ попадает в интервал значений вершины, или когда данная вершина не имеет детей в том направлении, в котором нам нужно идти. После остановки на какой-то вершине в неё добавляется пара (ключ, приоритет), и приоритет данной вершины обновляется по определённому правилу.

• ***Удаление*** происходит следующим образом: сначала мы находим ключ в какой-то вершине, потом удаляем это значение вместе с его приоритетом. Наконец, мы корректируем приоритет вершины, из которой было произведено удаление, чтобы он соответствовал своему набору ключей с приоритетами.

• ***Поиск*** проходит по вершинам *T-Treap*'а, пока не найдет вершину с подходящим интервалом значений, или когда в нужном направлении будет невозможно двигаться. Если мы нашли подходящую вершину, то поиск продолжается в ней дихотомией.

Одним из инвариантов *T-Treap*'а является свойство заполнения вершин – количество ключей в каждой вершине лежит между установленными заранее числами. Во время добавления или удаления ключа этот инвариант может нарушиться. Если в вершине не будет хватать значений, то их нужно будет забрать у соседних листьев. Если в вершине будет слишком много значений, то одно из них нужно будет переправить листьям. Так как только избыток или недостаток значений в вершине могут привести к созданию или удалению целой вершины, то метод, который будет этим управлять, окажет большое влияние на сбалансированную структуру *T-Treap*'а.

Есть несколько вариантов, куда можно переправлять лишние значения. В общем, мы хотели бы иметь функцию, которая бы распределяла переправление между правым и левым направлениями таким образом, чтобы свойство сбалансированности *treap*'а сохранялось. Мы можем сделать это, используя направляющий бит, который будет инвертироваться после каждого переправления.

Когда удаление ключа заставляет нас забирать значения у соседних листьев, то мы опять можем выбрать два направления. Теперь мы будем забирать значения оттуда, куда указывает инверсия направляющего бита. После перехода в одного из детей вершины мы инвертируем её направляющий бит. Вышеуказанный способ позволяет переправлению проходить в том же направлении, что и последний забор значения. Это позволяет поддерживать относительный баланс при перемещении значений между вершинами.

Если мы достаточно часто используем операцию чтения какого-либо значения, то мы можем менять приоритет этого значения, что непосредственно отразиться на приоритете вершины, в которой происходит чтение, а изменение приоритета вершины может вызвать серию поворотов, которая может поднять вершину на более высокий уровень.

Балансировка и управление приоритетами. Из-за того, что вся структура *T-Treap*'а зависит от приоритетов, то выбор того способа, каким следует их генерировать, определит, в какой степени *T-Treap* будет сбалансирован. Существует несколько алгоритмов, которые могут управлять приоритетами вершин. Основными тремя из них можно назвать: *Min, Max* и *Avg* (*Average*), которые применяются к множеству приоритетов ключей в вершине. Для *Min* и *Max* пересчёт приоритета занимает одно сравнение, за исключением случая, когда удаляется элемент, имеющий приоритет самой вершины. Тогда

потребуется полный просмотр всех приоритетов у ключей в вершине. Для *Avg* новый приоритет может быть посчитан вообще без операции сравнения.

После того, как у вершины поменяется приоритет, *T-Treap* проверяется на соответствие *основному порядку пирамиды*. Вершины вращаются так же, как в *treap*'е, если они нарушают *heap-order*. Но существует одно исключение – лист с недостаточно большой вместимостью может нарушать порядок *heap*'а. Это вызвано тем, что все внутренние вершины *T-Treap*'а должны подчиняться ограничениям на вместимость.

Начальные параметры и трудоёмкость T-Treap'а. У *T-Treap*'а есть несколько изменяемых параметров. Каждый из них влияет на производительность и может регулироваться отдельно от других. Алгоритмы для чтения значений обсуждаются в [8]. Существует модификация *T-Treap*'а, в которой у каждого ключа не хранится свой приоритет, а для обновления приоритетов вершин используется специальная функция. Важно отметить, что основное преимущество *T-Treap*'а – это кластеризация данных, откуда следует, что для ускорения его работы на различных машинах рекомендуется выбирать размер вершины кратным количеству байт в кластере.

Время работы основных операций будет пропорционально высоте *T-Treap*'а. Мы будем считать, что поиск в вершине ведётся за постоянное время. В [8] даётся оценка высоты *T-Treap*'а – $O(\log (N/X))$, где X – количество значений в вершине, N – общее число ключей.

Будущее. *T-Treap* является новым и не до конца разработанным АТД. Его создатели в [8] лишь обозначили основные идеи его конструирования, оставив читателям ряд вопросов для самостоятельного рассмотрения. Авторы предполагают, что со временем структуры, спроектированные на подобии *T-Treap*'а, приобретут большую популярность в связи с тенденциями роста производительности компьютерных систем.

17.4.2. Неявные декартовы деревья

В западной литературе, статьях и Интернете ни одного упоминания о декартовом дереве по неявному ключу пока нет. В русской практике спортивного программирования ACM ICPC эта структура была использована впервые в 2000 году, придумана членом команды Kitten Computing Николаем Дуровым

Неявное декартово дерево - это простая модификация обычного декартового дерева, которая, тем не менее, оказывается очень мощной структурой данных. Фактически, неявное декартово дерево можно воспринимать как массив, над которым можно реализовать следующие операции (все за O (log N)):

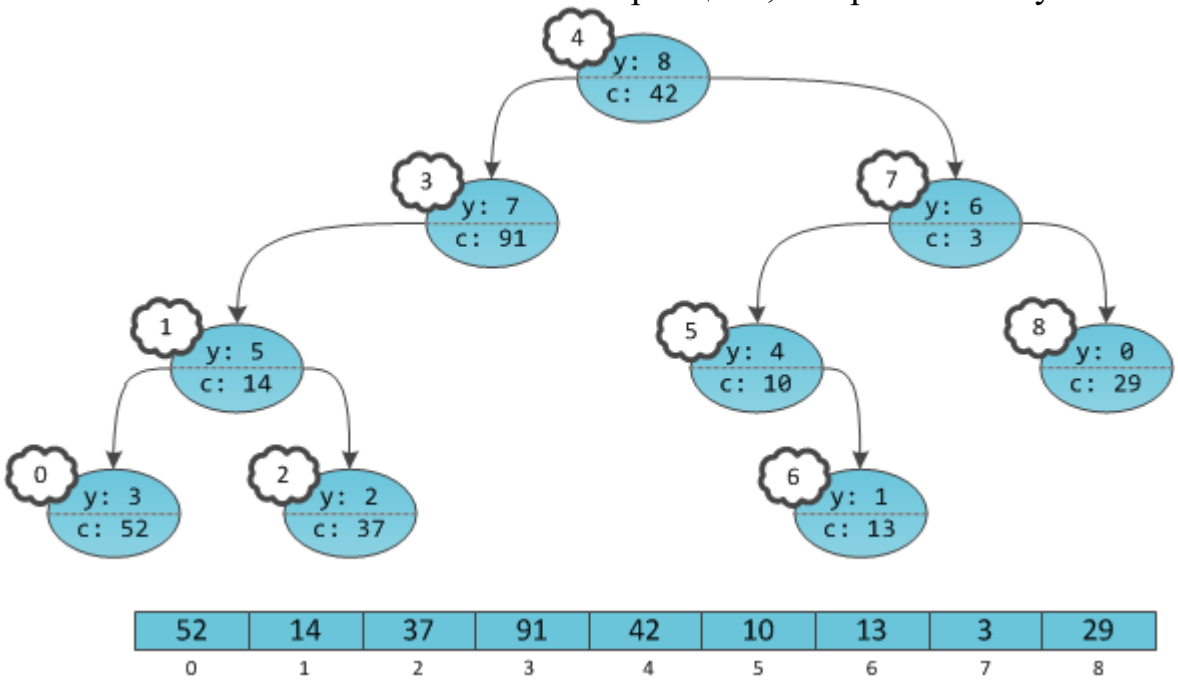
- Вставка элемента в массив в любую позицию
- Удаление произвольного элемента
- Сумма, минимум/максимум на произвольном отрезке, и т.д.

- Прибавление, покраска на отрезке
- Переворот (перестановка элементов в обратном порядке) на отрезке

Памяти при этом тратится все так же O(N). Конечно, O-нотация не отражает реальности. Фактическая память, занимаемая деревом: N на информацию, N на приоритеты, 2N на ссылки на потомков, N на размеры поддеревьев — это минимум, а если добавить еще и множественные запросы, то получим еще N на каждую операцию. Однако, возможности декартового дерева и простота его кода — преимущества, превосходящие проблему большого расхода памяти.

Ключевая идея заключается в том, что в качестве ключей следует использовать **индексы** элементов в массиве. Однако явно хранить эти значения ключей мы не будем (иначе, например, при вставке элемента пришлось бы изменять ключи в O (N) вершинах дерева).

Получается, что в дереве будто бы не ключи в вершинах проставлены, а сами вершины пронумерованы. Причем пронумерованы в уже знакомом порядке in-order обхода. Дерево с четко пронумерованными вершинами можно рассматривать как массив, в котором индекс — это тот самый неявный ключ, а содержимое — пользовательская информация `C`. Приоритеты (Y) нужны только для балансировки, это внутренние детали структуры данных, ненужные пользователю. Ключей *на самом деле* нет в принципе, их хранить не нужно.



Этот массив не приобретает автоматически никаких свойств, вроде отсортированности. Ведь на информацию-то у нас нет никаких структурных ограничений, и она может храниться в вершинах произвольно.

Теперь перейдём к реализации различных дополнительных операций на неявных декартовых деревьях:

- **Вставка** элемента. Пусть нам надо вставить элемент в позицию pos. Разобьём декартово дерево на две половинки: соответствующую массиву

[0..pos-1] и массиву [pos..sz]; для этого достаточно вызвать split (t, t1, t2, pos). После этого мы можем объединить дерево t1 с новой вершиной; для этого достаточно вызвать merge (t1, t1, new_item) (нетрудно убедиться в том, что все предусловия для merge выполнены). Наконец, объединим два дерева t1 и t2 обратно в дерево t - вызовом merge (t, t1, t2).

- **Удаление** элемента. Здесь всё ещё проще: достаточно найти удаляемый элемент, а затем выполнить merge для его сыновей l и r, и поставить результат объединения на место вершины t. Фактически, удаление из неявного декартова дерева не отличается от удаления из обычного декартова дерева.
- **Сумма/минимум** и т.п. на отрезке. Во-первых, для каждой вершины создадим дополнительное поле f в структуре item, в котором будет храниться значение целевой функции для поддерева этой вершины. Такое поле легко поддерживать, для этого надо поступить аналогично поддержке размеров cnt (создать функцию, вычисляющую значение этого поля, пользуясь его значениями для сыновей, и вставить вызовы этой функции в конце всех функций, меняющих дерево). Во-вторых, нам надо научиться отвечать на запрос на произвольном отрезке [A;B]. Научимся выделять из дерева его часть, соответствующую отрезку [A;B]. Нетрудно понять, что для этого достаточно сначала вызвать split (t, t1, t2, A), а затем split (t2, t2, t3, B-A+1). В результате дерево t2 и будет состоять из всех элементов в отрезке [A;B], и только них. Следовательно, ответ на запрос будет находиться в поле f вершины t2. После ответа на запрос дерево надо восстановить вызовами merge (t, t1, t2) и merge (t, t, t3).
- **Прибавление/покраска** на отрезке. Здесь мы поступаем аналогично предыдущему пункту, но вместо поля f будем хранить поле add, которое и будет содержать прибавляемую величину (или величину, в которую красят всё поддерево этой вершины). Перед выполнением любой операции эту величину add надо "протолкнуть" - т.е. соответствующим образом изменить t->l->add и t->r->add, а у себя значение add снять. Тем самым мы добьёмся того, что ни при каких изменениях дерева информация не будет потеряна.
- **Переворот** на отрезке. Этот пункт почти аналогичен предыдущему - нужно ввести поле bool rev, которое ставить в true, когда требуется произвести переворот в поддереве текущей вершины. "Проталкивание" поля rev заключается в том, что мы обмениваем местами сыновья текущей вершины, и ставим этот флаг для них.

- **17.4.3. B-Treaps**

Небольшим недостатком декартового дерева есть то, что Y нигде не используется практически, кроме балансировки дерева. Тогда вместо Y можно брать реальную информацию, то есть не случайно сгенерированные числа. Если после этого умножить все Y на некоторое простое число P и взять по модулю 2^64 (что очень удобно и быстро), то можно доказать, что Y будут распределены случайным образом. Для того, что бы при необходимости узнать Y, нужно будет просто домножить на обратный элемент к P по модулю 2^64.

Таким образом, память у декартового дерева не будет тратиться попусту и в ней можно хранить некоторую информацию, помимо ключей.

Использование. Декартово дерево может использоваться во всех приложениях, где требуется АТД (абстрактный тип данных), который поддерживает стандартные словарные операции. Например, на основе декартового дерева в [2] была построена сортировка. На основе тестов в [3] можно увидеть, что декартово дерево составляет серьёзную конкуренцию *RBT* (*red-black trees*) по скорости исполнения основных операций. С точки зрения программирования, код декартового дерева намного короче, чем у *RBT*. В [4] декартовое дерево используется для организации DVD библиотеки. Самые подробные тесты приведены в [5], здесь авторы так же исследуют зависимость общего времени работы декартового дерева от количества сделанных поворотов. В [6] декартовые деревья используются для построения диаграмм Вороного, но, по результатам авторов, декартовые деревья проигрывает даже BST. В [7] предлагается применять декартово дерево в многопоточной системе для увеличения производительности.

Литература

1. Ласло М. Вычислительная геометрия и компьютерная графика на C++: Пер. с англ. — М.: БИНОМ, 1997. — 304 с.: ил.

2. Jeff Erickson. Randomized treaps // University of Illinois at Urbana-Champaign. CS373 lecture notes. 2002.

3. Dominique A. Heger. A Disquisition on The Performance Behaviour of Binary Search Tree Data Structures // Cepis Upgrade Journal. Vol. 5. October 2004. № 5, p. 67-74.

4. Jim Moiani, Michael Crocker. DVD Library System // University of Notre Dame. CSE 331 data structures. 2003.

5. Hugh E. Williams, Justin Zobel, Steffen Heinz. Self-adjusting trees in practice for large text collections // Software: Practice and Experience. Vol. 31. August 2001. № 10, p. 925-939.

6. Kenny Wong, Hausi A.Muller. An Efficient Implementation of Fortune's Plane-Sweep Algorithm for Voronoi Diagrams // Technical Report DCS-182-IR. 1991, p. 8-9.

7. Albrecht Schmidt, Christian S. Jensen. Efficient Management of Short-Lived Data // A TIMECENTER Technical Report. 2005.

8. Joshua P. MacDonald, Ben Y. Zhao. T-treap and Cache Performance of Indexing Data Structures // University of California, Berkeley. CS252 course projects. 1999.

11. Райков Павел, Treaps и T-Treaps 2006 http://rain.ifmo.ru/cat/

12. Seidel Raimund, Aragon Cecilia R. Randomized Search Trees // Algorithmica. – 1996. – 16 (4/5). – pp. 464–497.

13. Дональд Кнут Искусство программирования, том 3. Сортировка и поиск. — 2-е изд. — М.: «Вильямс», 2007.

14. Salvador Roura and Conrado Martínez Randomization of Search Trees by Subtree Size. (DOI) Proceedings of the Algorithms, 1996

3.7. Объединение и расщепление сбалансированных деревьев поиска

До этого момента мы только обсуждали `find`, `insert`, `delete` операции в наборах, хранившихся в сбалансированных деревьях поиска. Некоторые дополнительные операции, которые поддерживаются большинством деревьев поиска описаны в этой главе. Мы уже упоминали в параграфе 2.7 `interval_find` запрос в списке всех ключей в интервале запроса, и относящихся запросов для следующих малых или больших ключей. Методы описанные там также работают в любом сбалансированном дереве поиска.

Теорема: Любое сбалансированное дерево может расшириться с O(1) накладкой `find`, `insert`, и `delete` операций для поддержки дополнительных операций `find_next_larger` и `find_next_smaller` со временем O(log n) и interval_find в вывод-чувствительному времени O(log(n)+k), если k объектов в интервале запроса.

Связанные уровнем деревья имеют связь для двойного связного списка листьев в любом случае; они находятся на низшем уровне. Skip листы имеют эти связи в одном направлении, которого недостаточно. Итак, эти структуры не нуждаются ни в какой модификации для поддержки интервала запроса.

Больше сложных операций требует расщепление набора в заданном пороге, и в другом направлении объединяя два набора, чьи ключи разделены порогом. Обе операции, `split` и `join`, могут быть встроены для самого уравновешенного дерева поиска, описанного в этой главе во время O(log n).

Это самое лёгкое для skip списка, потому что элементы skip списка присваивают свои уровни независимо. Для разделения, мы просто находим точку для разделения и вставляем новый элемент заполнения для списка, который выходит за порог разделения, и вставляем NULL указатели для выполнения на всех уровнях, которые мы урезали. И в другом направлении, объединяя два skip списка, где все ключи вначале меньше, чем все ключи позже, довольно просто; мы удаляем элемент заполнения в начале второго skip списка и соединяем все уровни списков, возможно вставляя дополнительные заполняющие элементы в первом skip списке, если его максимальный уровень был меньше чем максимальный уровень второго skip списка. Общая работа в каждой операции O(log n): мы должны найти точку, где разделить и затем проделать O(1) работу на каждом уровне.

Теорема: Структура skip списков поддерживает разделение у порога и соединение двух отдельных skip списков в ожидаемое врем O(log n).

Для «худшего случая» сбалансированных деревьев, разделение и соединение требует чуть больше обдумывания, но разделение следует за тем, после того как мы получили соединение. В случае деревьев с сбалансированной высотой, это будет работать следующим образом. Предположим у нас есть два сбалансированной высоты дерева поиска T_1 и T_2 высоты h_1 и h_2, которые разделены, и все ключи в T_1 меньше чем все ключи в T_2.

1. Если h_1 и h_2 различаются на единицу, мы можем добавить новый общий корень, чей ключ - это ключ к крайнему слева листу в T_2.
2. Если $h_1 \le h_2-2$, мы следуем по крайнему левому пути в T_2, отслеживая пути назад к корню до того, пока мы найдем узел чья высота как минимум h_1. Потому что два последовательных узла на пути различают в весе одним или двумя, следуя этим случая возможно:
   2.1 Узле на крайнем левом пути T_2 имеет высоту h_1 и верхнего соседа, имеющего высоту h_1+2: Затем мы просто создаём новый узел с высотой h_1+1 ниже верхнего

соседа на пути, который имеет как правого нижнего узла-соседа, так и левого-нижнего у корня T_1, и как ключ ключа нижнего левого листа в T_2. Новое дерево снова сбалансировано по высоте.

2.2 Узел в левом нижнем пути T_2 имеет высоту h_1 и его верхнего соседа с h_1+1: Затем мы просто создаём новый узел с высотой h_1+1 ниже верхнего соседа на пути, который как правый нижний сосед узла с высотой h_1 на пути, как левый нижний сосед корня T_1, и ключ ключа самого левого листа в T_2. Затем мы исправляем высоту верхнего соседа до h_1+2 и производим балансировку, поднимаясь к корню.

2.3 Узел на самом левом пути T_2 имеет высоту h_1-1 и его верхнего соседа имеющего высоту h_1+1: Затем мы просто создаём новый узел с высотой h_1+1 ниже верхнего соседа на пути, который имеет правого нижнего соседа-узла высотой h_1-1 на пути, как левый сосед корня T_1, и как ключ ключа самого левого листа в T_2. Затем мы исправляем высоту верхнего соседа на h_1+2 и проводим балансировку дерева, поднимаясь по корню вверх.

3. Если $h_2 \leq h_1-2$, мы следуем самого правого пути в T_1, сохраняя путь назад к корню до того, пока мы не найдем узел, чья высота как минимум h_2. Потому что два последовательных узла на пути, различаясь в высоте на один или два, возможны следующие случаи:

3.1 Узел на крайнем левом пути T_2 имеет высоту h_2 и верхнего соседа, имеющего высоту h_2+2: Затем мы просто создаём новый узел с высотой h_2+1 ниже верхнего соседа на пути, который имеет как правого нижнего узла-соседа, так и левого-нижнего у корня T_1, и как ключ ключа нижнего левого листа в T_2. Новое дерево снова сбалансировано по высоте.

3.2 Узел на крайнем левом пути T_2 имеет высоту h_2 и верхнего соседа, имеющего высоту h_2+1: Затем мы просто создаём новый узел с высотой h_2+1 ниже верхнего соседа на пути, который имеет как правого нижнего узла-соседа, так и левого-нижнего у корня T_1, и как ключ ключа нижнего левого листа в T_2. Затем мы исправляем высоту верхнего соседа до h_2+2 и производим балансировку, поднимаясь к корню.

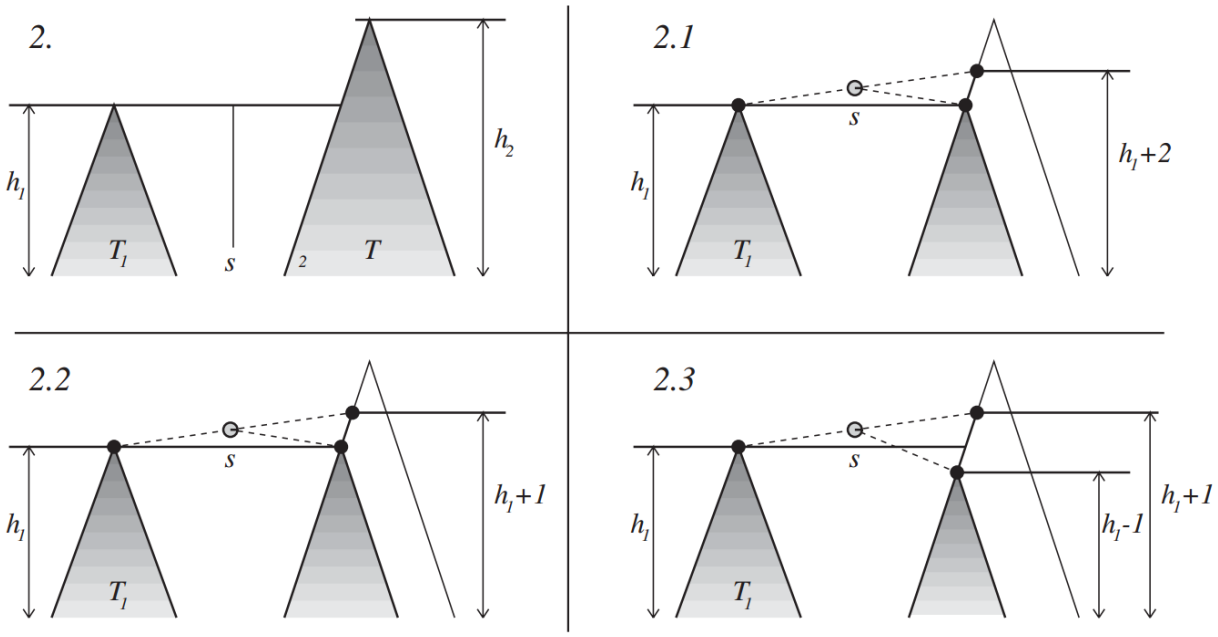
3.3 Узел на самом левом пути T_2 имеет высоту h_1-1 и его верхнего соседа имеющего высоту h_1+1: Затем мы просто создаём новый узел с высотой h_1+1 ниже верхнего соседа на пути, который имеет правого нижнего соседа-узла высотой h_1-1 на пути, как левый сосед корня T_1, и как ключ ключа самого левого листа в T_2. Затем мы исправляем высоту верхнего соседа на h_2+2 и проводим балансировку дерева, поднимаясь по корню вверх.

Итак, идея в том, чтобы вставить новый узел, отходящий к меньшему дереву от правильного внешнего пути к дереву выше и использовать методы балансировки для восстановления условий равновесия. Нам нужно опуститься вниз на правое дерево, только для восстановления ключа, значение которого мы используем для разделения деревьев; если это значение ключа уже известно, то сложность будет только $O(|h_1-h_2|+1)$.

Теорема: Два разделённых сбалансированных по высоте деревьев поиска могут объединяться со временем занимаемым $O(\log n)$. Если разделяющие ключи уже известны, это время сокращается до $O(|\text{высота}(T_1)-\text{высота}(T_1)+1)$.

Мы можем сократить разделение единственного дерева поиска в заданном пороге в раздел посреди поиска пути в два набора поисковых деревьев, ведомы рядом операций объединения для сборки этих деревьев вместе в левые и правые деревья раздела. Это работает следующим образом; дан key_{split}, мы следуем поиску пути для этого ключа от корня

к листу. Каждый раз, мы следуем левому указателю, мы вставляем правый указатель впереди правого списка дерева, вместе с ключом, который разделяет правое поддерево от всего, к левому оригинальному дереву.



Объединение двух сбалансированных по высоте деревьев, чьи ключи разделены S. Случаи 2.1, 2.2, 2.3: Вставка дерева между левым ограничивающим путём.

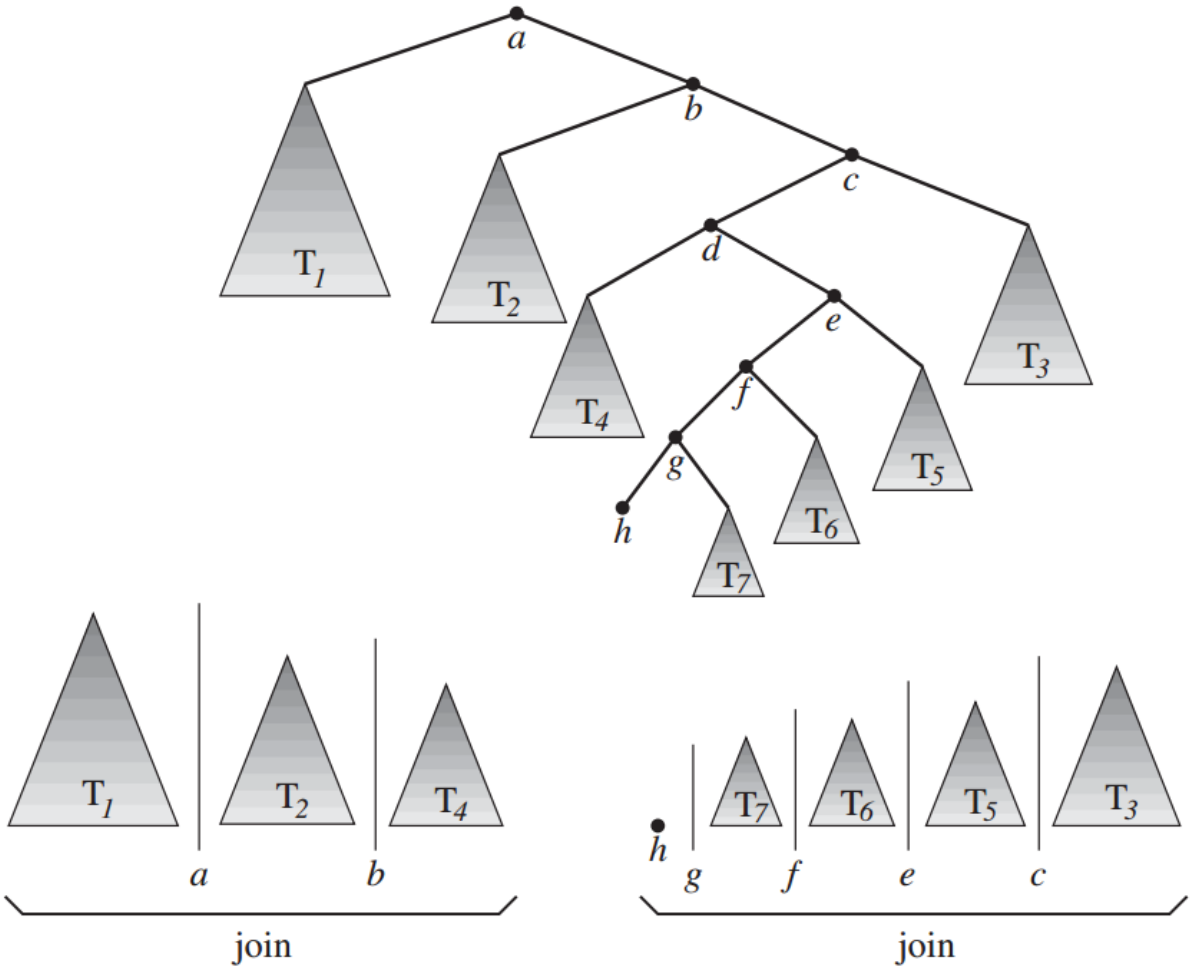
И каждый раз мы следуем правому указателю, вставляя левый указатель впереди левого списка дерева, вместе с ключом, который разделяет левое поддерево от всего остального к правому оригинальному дереву.

Когда мы достигнем листа с расщепляющим по key_{split}, мы создали два списка сбалансированных деревьев поиска, увеличивающих дерево поиска. Мы сейчас объединим эти деревья поиска в порядке возрастания размера (как находятся в списке), используя как разделяющий ключ, ассоциированный со следующим деревом, которое мы возьмём из списка. Ключ первого дерева в списке отбрасывается. Затем сложность построения двух списков будет O(log n), потому что мы только что следовали по пути вниз к листу, и общая сложность операции объединения O(log n), складывающаяся как сумма высот деревьев в списке. Здесь мы используем высоту объединённых двух деревьев и по крайней мере высоту большего дерева. Вместе это влияет на следующее:

Теорема: Сбалансированное по высоте дерево поиска можно разделить на заданном значение ключа в два сбалансированных дерева поиска со временем O(log n)

Для красно-чёрных деревьев и (a, б)-деревьев, работают схожие методы.

Финальным вариантом, изучаемые в некоторых газетах это разделение обновления и уравновешивание, известное как расслабленная балансировка. Это мотивировано внешней моделью памяти: В порядке минимизации количество переводов блоков и передвижения их в то время, когда загрузка системы мала, хотелось бы выполнить только необходимые вставки и удаления, но придётся производит уравновешение позже в расцепленном «очистке» запуске (Нурми, Соисалон-Соининен, и Вуд 1987).



Разделение дерева в ключе *h*

Алгоритмическая проблема здесь в описании и в анализе балансирующих методов, мы всегда предполагали, что перед вставкой, дерево было сбалансированно, так что в порядке применения этих методов, нам нужно сбалансировать дерево перед следующей вставкой (Ларсен и Фагерберг 1996). Версия расслабленной балансировки для большинства деревьев уже обсуждалась здесь (Нурми 1987; Нурми и Соисалон-Соининен 1996; Соисалон-Соининен и Виндмайер 1997; Ларсен 1998, 2000, 2002, 2003), также для основных структур памяти этот вопрос несёт только теоретический интерес, потому что проблемы с балансировкой затрагивают только параллельные системы, где несколько процессов действуют на дереве поиска, хранящееся в общей памяти. Связанный это также ленивая балансировка, выполняемая только во время следования запросам, выдвинуто Кессельсом (1983).

Сбалансированные деревья способны поддерживать эффективное выполнение многих операций - вставки, удаления, точного поиска, диапазонного поиска, поиска минимума и максимума и т.д. Рассмотрим более сложный вопрос: как разделить множество ключей, хранящихся в сбалансированном дереве на два множества, ключи одного из которых меньше некоторого значения *V*, а ключи второго больше *V*, и как выполнять обратную операцию. Важно, что в

результате этих операций также должны быть сбалансированные деревья, в противном случае задача была бы тривиальной и для объединения выполнялась бы очень легко. Пусть *F*, *S* - бинарные деревья поиска. Если в *F* все элементы меньше некоторого *V*, а в *S* - больше *V*, то в качестве корня результирующего дерева *R* берём минимальный элемент *M* дерева *S*, левым поддеревом *R* становится *F*, а правым поддеревом - *S* (из которого исключён *M*). Пример объединения бинарных деревьев поиска приведён на рис. 1.4.

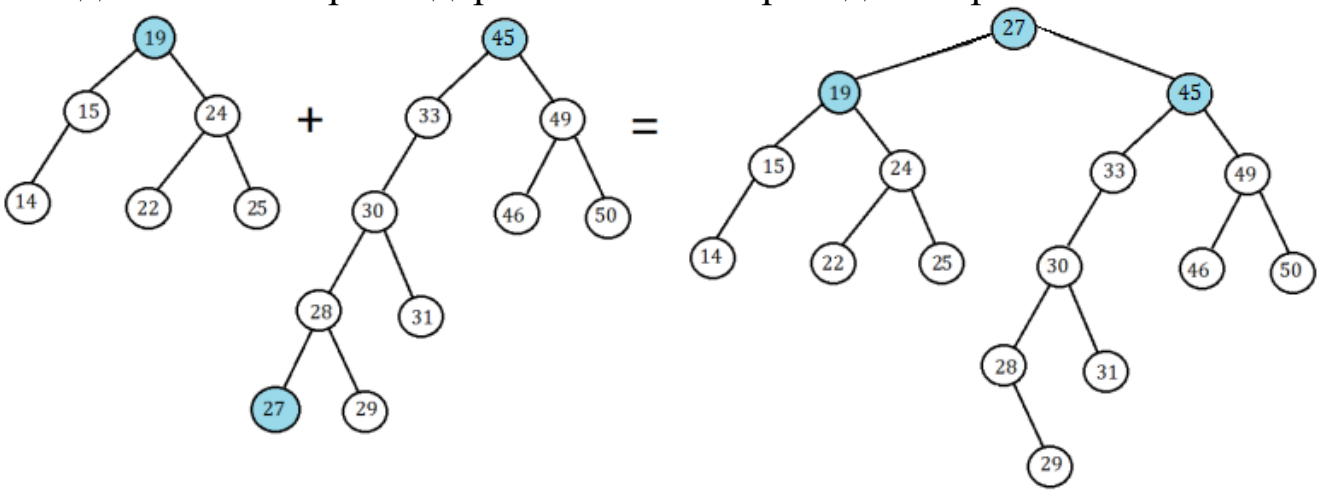


Рис. 1.4. Объединение несбалансированных бинарных деревьев поиска

Довольно просто выполнить разделение и объединение слоёных списков. При разделении списков основная часть операций приходится на поиск точки расщепления - O(log *n*) операций, где *n* - количество записей. Далее просто запоминаем указатели на начало уровней правого подсписка. Также в каждом уровневом списке левого подсписка указатель *next* самого правого элемента обнуляется, чтобы обозначить новый хвост уровневого списка; для каждого уровневого списка выполняется O(1) операций. Таким образом, после того, как мы нашли точку расщепления, операция разделения слоёного списка будет не сильно отличаться от операции разделения линейного списка, только сложность будет составлять O(*m*), где *m* - количество уровней (слоёв). Исходя из средних показателей эффективности слоёных списков, сложность составит O(log *n*). При объединении слоёных списков следует учесть следующее: если максимальный уровень первого списка (в котором находятся элементы меньше *V*) меньше максимального уровня второго списка (в котором находятся элементы больше *V*), то в первом списке создаются дополнительные указатели, чтобы количество уровней в

списках стало одинаковым. Сложность объединения также логарифмическая.

Теперь рассмотрим, как выполнить объединение двух деревьев поиска, сбалансированных по высоте. Пусть есть два дерева T_1 и T_2, имеющие высоты h_1 и h_2 соответственно, причём все ключи в T_1 меньше всех ключей в T_2.

1.Если h_1 и h_2 отличаются не более, чем на единицу, то корнем объединённого дерева становится ключ самого левого листа в T_2. Эта ситуация весьма похожа на таковую для несбалансированных деревьев поиска.

2.Если $h_1 \le h_2$-2, мы идём по самому левому пути в T_2, запоминая пройденный от корня путь, до тех пор пока не найдём узел высотой не более h_1 (см. рис. 1.5). Поскольку два последовательных узла на этом пути отличаются высотой на 1 или на 2, возможны следующие случаи.

2.1.Узел на самом левом пути T_2 имеет высоту h_1, и его вышележащий сосед имеет высоту h_1+2. Тогда просто создаём новый узел с высотой h_1+1 ниже вышележащего соседа на пути, который имеет в качестве правого нижележащего соседа узел с высотой h_1 на этом пути и в качестве левого нижележащего соседа - корень T_1, и в качестве ключа - ключ самого левого листа в T_2. Полученное дерево является деревом, сбалансированным по высоте.

2.2.Узел на самом левом пути T_2 имеет высоту h_1, и его вышележащий сосед имеет высоту h_1+1. Тогда создаём новый узел с высотой h_1+1 ниже вышележащего соседа на пути, который имеет в качестве правого нижележащего соседа - узел с высотой h_1 на этом пути, в качестве левого нижележащего соседа – корень T_1, и в качестве ключа - ключ самого левого листа в T_2. Затем мы корректируем высоту вышележащего соседа и восстанавливаем сбалансированность, поднимаясь к корню.

2.3.Узел на самом левом пути T_2 имеет высоту h_1-1, и его вышележащий сосед имеет высоту h_1+1. Тогда создаём новый узел с высотой h_1+1 ниже вышележащего соседа на пути, который имеет в качестве правого нижележащего соседа - узел с высотой h_1-1 на этом пути, в качестве левого нижележащего соседа – корень T_1, а в качестве ключа - ключ самого левого листа в T_2. Затем мы корректируем высоту вышележащего соседа и восстанавливаем сбалансированность, поднимаясь к корню.

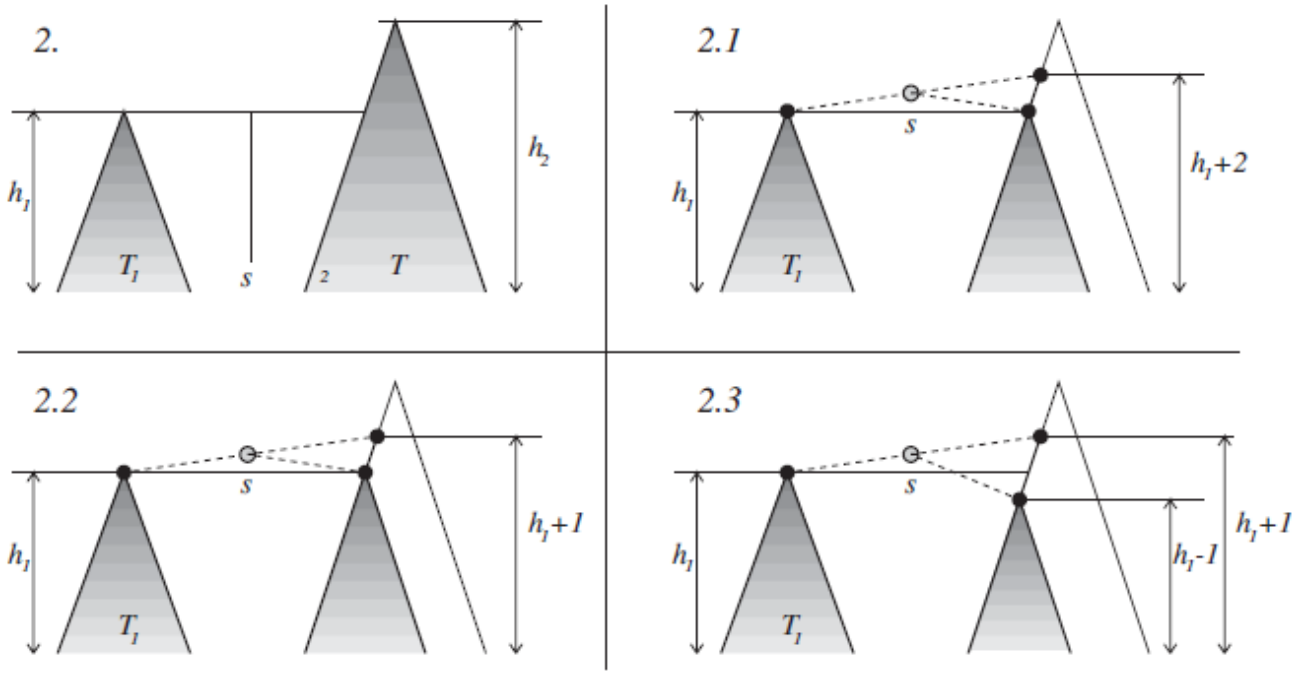


Рис. 1.5. Объединение двух сбалансированных деревьев при $h_1 \le h_2-2$

3.Если $h_2 \le h_1-2$, мы идём по самому правому пути в T_1, запоминая пройденный от корня путь, до тех пор пока не найдём узел высотой не более h_2. Поскольку два последовательных узла на этом пути отличаются высотой на 1 или на 2, возможны следующие 3 случая.

3.1.Узел на самом правом пути T_1 имеет высоту h_2, и его вышележащий сосед имеет высоту h_2+2. Тогда просто создаём новый узел с высотой h_2+1 ниже вышележащего соседа на пути, который имеет в качестве левого нижележащего соседа узел с высотой h_2 на этом пути и в качестве правого нижележащего соседа - корень T_2, и в качестве ключа - ключ самого левого листа в T_2. Полученное дерево является деревом, сбалансированным по высоте.

3.2.Узел на самом правом пути T_1 имеет высоту h_2, и его вышележащий сосед имеет высоту h_2+1. Тогда создаём новый узел с высотой h_2+1 ниже вышележащего соседа на пути, который имеет в качестве левого нижележащего соседа - узел с высотой h_2 на этом пути, в качестве правого нижележащего соседа - корень T_2, и в качестве ключа - ключ самого левого листа в T_2. Затем мы корректируем высоту вышележащего соседа и восстанавливаем сбалансированность, поднимаясь к корню.

3.3.Узел на самом правом пути T_1 имеет высоту h_2-1, и его вышележащий сосед имеет высоту h_2+1. Тогда создаём новый узел с высотой h_2+1 ниже вышележащего соседа на пути, который имеет в

качестве левого нижележащего соседа - узел с высотой h_2-1 на этом пути, в качестве правого нижележащего соседа - корень T_2, а в качестве ключа - ключ самого левого листа в T_2. Затем мы корректируем высоту вышележащего соседа и восстанавливаем сбалансированность, поднимаясь к корню.

Объединение двух сбалансированных по высоте деревьев в общем случае характеризуется оценкой O($\log n$). Если же заранее известно значение, которое не меньше любого ключа в T_1 и не больше любого ключа в T_2, то оценка сложности сокращается до O($|h_1$-$h_2|$+1). [4]

Приведём примеры объединения деревьев, сбалансированных по высоте (АВЛ-деревьев).

Начнём с простой ситуации. Требуется объединить два АВЛ-дерева, приведённых на рис. 1.6. В качестве корня результирующего дерева выберем ключ K такой, что: 1) все ключи в левом дереве меньше K; 2) все ключи в правом дереве не меньше K. Исходя из свойства деревьев поиска, идём по самому левому пути правого дерева - так мы находим минимальный ключ в нём. Это ключ 24. Он становится корнем результирующего дерева. Узел 8 становится левым потомком корня результирующего дерева, а узел 42 - правым потомком. Получившееся дерево удовлетворяет свойству АВЛ-дерева.

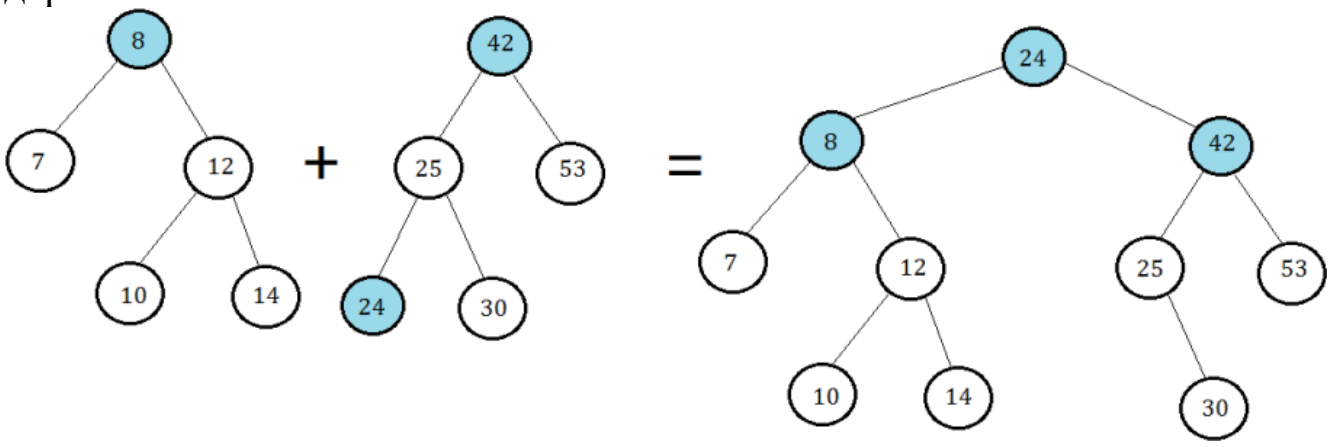


Рис. 1.6. Объединение АВЛ-деревьев – простая ситуация

Теперь рассмотрим более сложную ситуацию: левое дерево на 2 выше, чем правое (см. рис. 1.7). Сложность в том, что не получится объединение как в предыдущем примере, которое по существу не отличалось от объединения обычных бинарных деревьев поиска. Высота поддеревьев результирующего дерева будет отличаться на 2,

что нарушает АВЛ-свойства. На этот раз разделяющий ключ берётся в левом дереве как более высоком. Поскольку все ключи правого дерева больше ключей левого дерева, а АВЛ-дерево является деревом поиска, в качестве разделяющего ключа берём максимальный ключ левого дерева. Это можно сделать, когда идём по самому правому пути. Объединяем деревья: корень результирующего дерева - 27, его сыновья - 14 и 42 соответственно (см. рис. 1.7).

Выполненных действий в данном случае недостаточно: результирующее дерево нарушает АВЛ-свойства. В результирующем дереве корень не удовлетворяет АВЛ-свойствам, поскольку левое поддерево на 2 выше, чем правое. Остальные узлы имеют допустимые показатели сбалансированности. Заметим, что полученное дерево можно сбалансировать большим правым вращением вокруг корня (схема большого правого вращения приведена на рис. 1.8). После этого результирующее дерево становится сбалансированным, и объединение деревьев завершено (см. рис. 1.9).

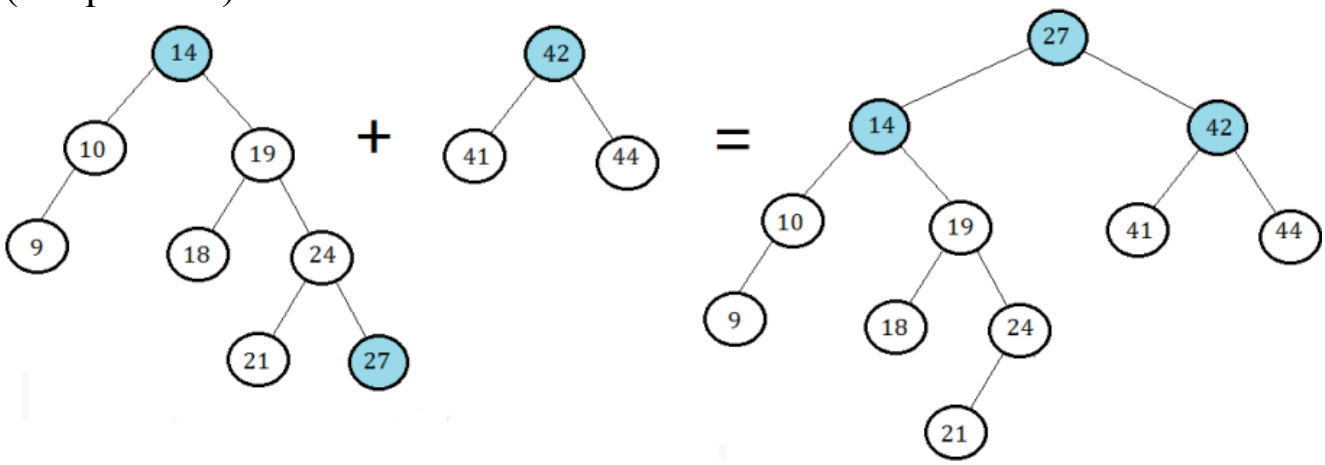


Рис. 1.7. Объединение АВЛ-деревьев – сложная ситуация. Шаг 1

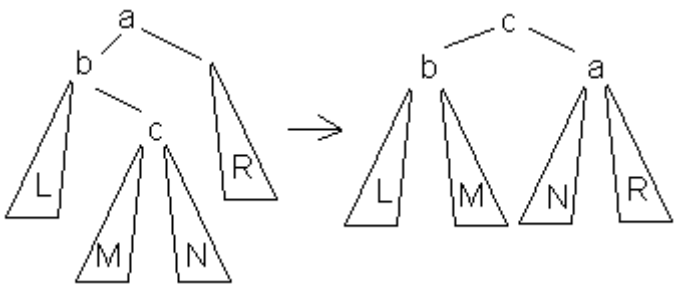


Рис. 1.8. Большое правое вращение

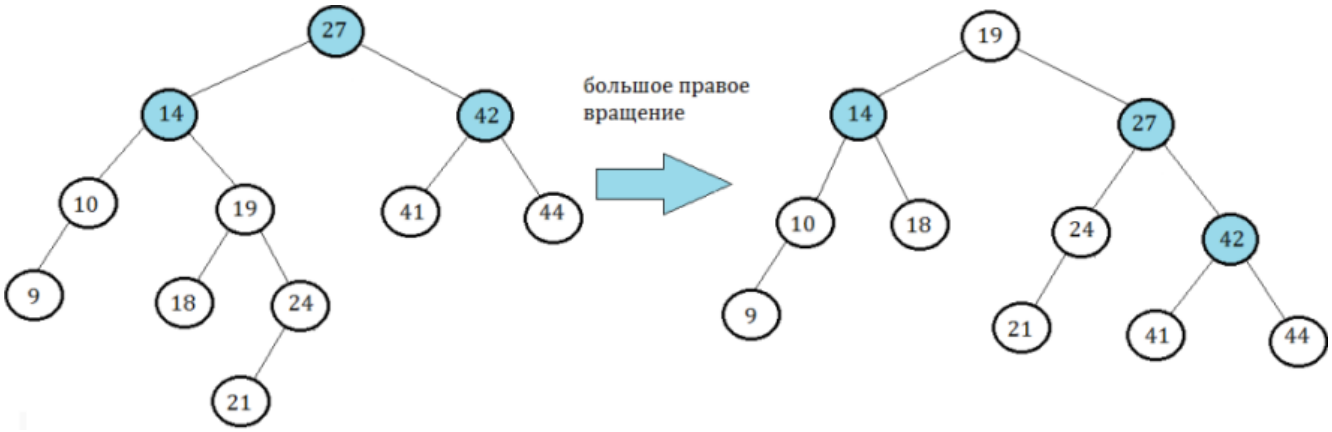


Рис. 1.9. Объединение АВЛ-деревьев – сложная ситуация. Шаг 2

Мы можем свести разделение дерева поиска по данному «пороговому» значению *V* к разделению вдоль пути поиска на два множества деревьев поиска, за которым следует ряд операций объединения, выполняемых, чтобы собрать эти деревья в левое и правое деревья, выступающие в качестве результатов разделения. Это выполняется следующим образом. Дан *K* - ключ, по которому ведётся разделение. Идём по пути поиска этого ключа от корня к листу. Каждый раз, когда мы идём по указателю *left* (указателю на левого потомка), мы вставляем указатель *right* перед списком правого дерева, вместе с ключом, который отделяет правое поддерево от всех ключей левого. Каждый раз, когда мы идём по указателю *right*, мы вставляем указатель *left* перед списком левого дерева, вместе с ключом, который отделяет левое поддерево от всех ключей правого.

Когда мы достигаем листа по пути поиска *K*, у нас оказываются созданы два списка сбалансированных деревьев поиска, расположенных в порядке возрастания ключей. Теперь мы объединяем эти деревья поиска в порядке возрастания, используя в качестве разделяющего ключа ключ, ассоциированный со следующим деревом из списка. Тогда сложность построения двух списков составляет O(log *n*), поскольку мы лишь идём к листу по пути поиска, и общая сложность операций объединения также O(log *n*) – это сумма высот деревьев из списка. Итого общая сложность разделения составляет O(log *n*).

Общая схема разделения бинарных деревьев поиска приведена на рис. 1.10.

Коротко рассмотрим пример разделения АВЛ-дерева (см. рис. 1.11). Пусть разделение ведётся по ключу 24. Сначала спускаемся по

пути к этому ключу, получая «материал» для левого и правого поддеревьев.

Далее выполняем «сборку» левого и правого деревьев по правилам объединения деревьев. Там, где требуется объединить дерево и одиночный ключ, последний рассматривается как дерево с единственным узлом. Заметим, что при получении левого дерева потребуется выполнить балансировку. В данном случае используется малое правое вращение (схема малого правого вращения приведена на рис. 1.12). Результат приведён на рис. 1.13.

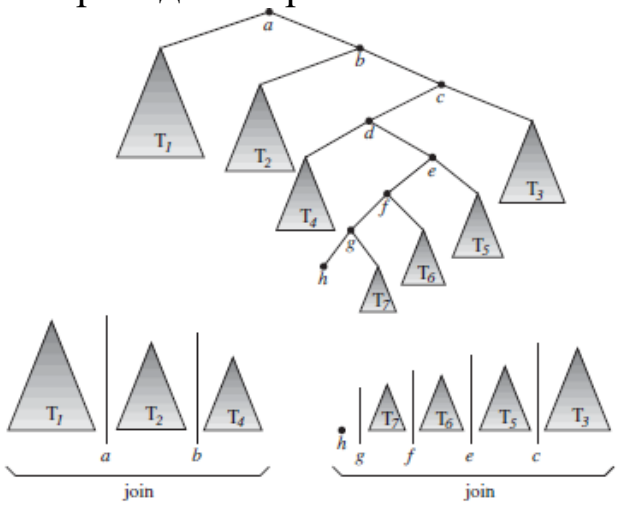


Рис. 1.10. Разделение дерева по ключу h

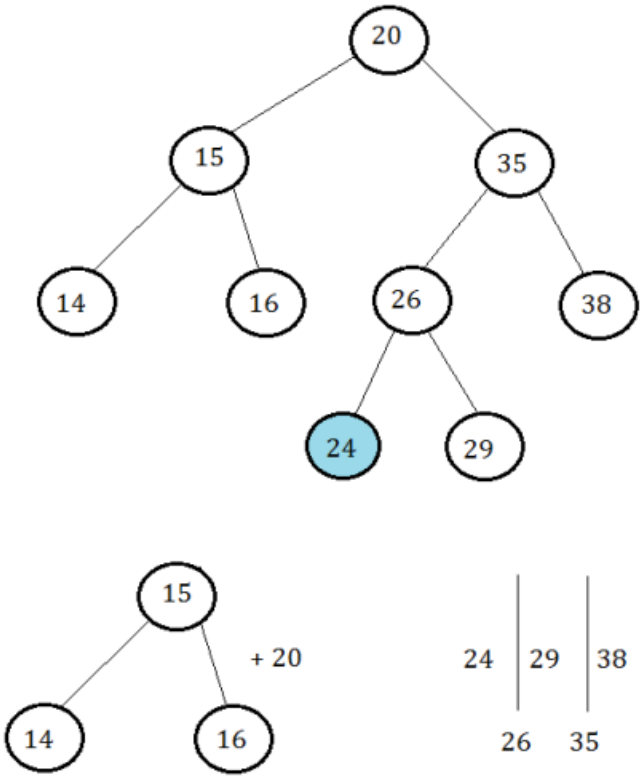


Рис. 1.11. Разделение АВЛ-дерева. Шаг 1

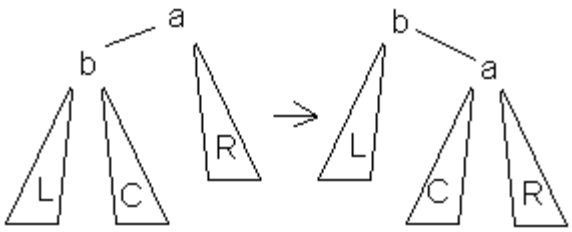


Рис. 1.12. Схема малого правого вращения

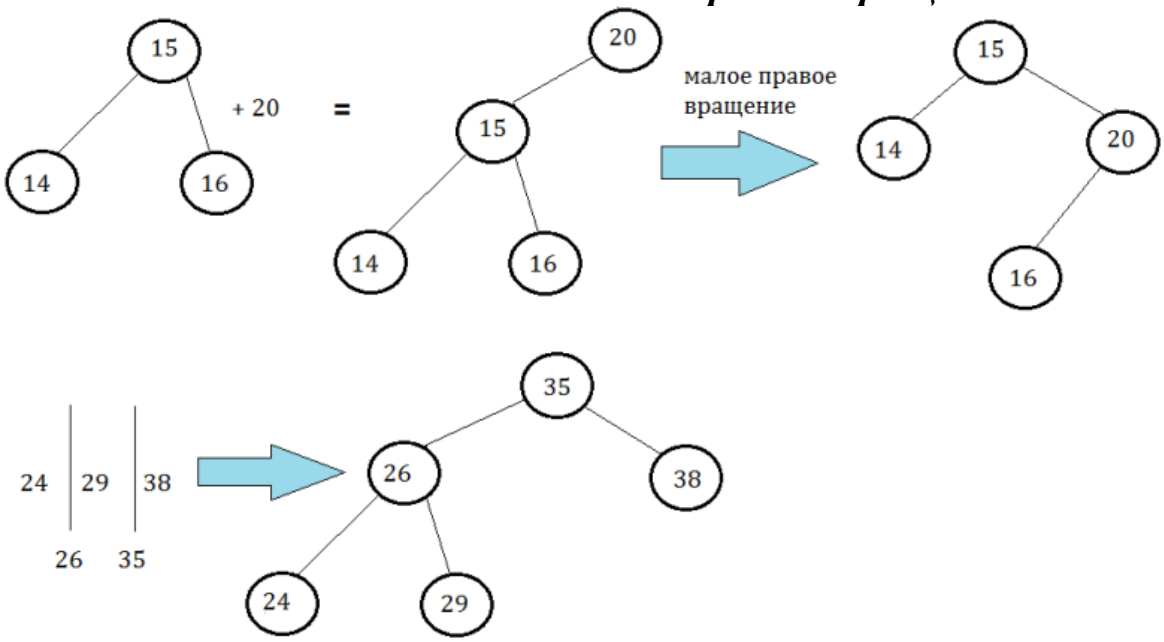


Рис. 1.13. Разделение АВЛ-дерева. Шаг 2

3.6. Деревья с постоянным временим обновления в заданном месте 111

3.7. Лиственные деревья и уровень связывания 114

3.8. Деревья с частичной реорганизацией: Амортизационная анализ

Совершенно другой подход для того чтобы оставить деревья поиска сбалансированными нужно перестроить их. Конечно, перестройка всего дерева занимает Ω(n) времени, так что нет разумной альтернативы в обновлении методов O(log n) сложности, если будем делать в каждом обновлении для всего дерева. Но оказывается это сопоставимо с амортизированной сложностью, если мы только иногда перестроим и перестроим только поддеревья. Это впервые наблюдали Овермарс, которые изучали частичную перестройку как метод общего характера, превратить статические структуры данных (не позволявших обновления) в динамические структуры данных (поддерживаются операции обновления) (Овермарс 1983). Мы потеряем на этом наихудшую гарантию во время обновления но всё еще имеем амортизированную связь между рядом обновлений. Каждое из них может, однако, занять Θ(n) время.

Использование частичных перестроек сбалансированных деревьев поиска было заново открыто в другом контексте Адерссоном (1989, 1990, 1999) и Галпериным и Ривестом (1993). Они были заинтересованы в вопросе насколько мало информации необходимо для

сбалансирования дерева. Красно-чёрные деревья всё еще нуждаются в одном бите на узел, но несомненно никакой информации в узлах необходимо. Один может держать дерево сбалансированным только с общим количеством листьев в качестве балансирующей информации, потому что это необходимо для успешного обнаружения того, что лист слишком низко.

Даётся количество n элементов, один может установить высоту порога с log n для некоторых (*c)* достаточно много. Затем мы можем решить, куда вниз мы направимся по дереву к листу, является ли глубина этого листа слишком велика, для текущего номера элемента. В таком случаю некоторые поддеревья содержат листья, нуждающиеся в балансировке, но мы не знаем где начинается поддерево. Возможно, что следующий log n уровень над листом, это бинарное дерево; только это идеально сбалансированное дерево прикрепляется длинным путём к корню. Итак мы имеем путь вверх от листа к корню и проверяем для каждого узла лежит ли поддерево ниже того узла и достаточно ли оно не сбалансированно, чтобы балансировка дала какое-либо улучшение. Это звучит очень неэффективно, но потому что поддерево, которое мы ищем экспоненциально растёт в размере, общая работа определяется последним поддеревом – тем, который мы решили сбалансировать.

Наши меры по уравновешиванию это α-вес-баланса. Потому что мы используем другую стратегию балансировки, ограничения на α параграфа 3.2 здесь не подойдут. Здесь мы заинтересованы в $\alpha<\frac{1}{4}$. Для α-вес-баланса, наша глубина привязана на $\log\left(\frac{1}{1-\alpha}\right)^{-1}$ log n, как в параграфе 3.2: если по пути все узлы α-вес-сбалансированы, то это верхняя граница для длины пути. Но мы не можем напрямую использовать нарушение α-вес-баланса как критерий для перестройки, потому что это гарантирует эффективность уменьшения высоты оптимальной перестройкой. Оптимальное дерево снизу-вверх с 2^k+1 листьями чрезвычайно неуравновешенно в корне, но всё еще оптимальной высоты. Вместо этого, мы примем поддерево, как необходимое в перестройке если его высота больше максимальной высоты α-вес-баланс дерева с тем же количеством листьев или равному, если его количество листьев меньше чем минимального количества листьев α-вес-сбалансированного дерева той же высоты, которое $\left(\frac{1}{1-\alpha}\right)^h$. Это гарантия того, что перестройка уменьшит высоту.

Итак, метод вставок таков: мы делаем базовую вставку, отслеживая глубину и путь. Если после вставки глубина листа ниже порога, нет необходимости в балансировке. Если у листа глубина выше порога, мы снова идём вверх по пути и переделываем поддерево под текущим узлом в связный список, используя метод 2.8, и связываем два списка с левых и правых поддеревьев. Если узел это i-й узел на пути от листа и количество листьев в этом списке больше чем $\left(\frac{1}{1-\alpha}\right)^i$, мы движемся вверх к узлу на пути к корню, иначе переделываем список в оптимальное дерево, используя метод «сверху-вниз» из параграфа 2.7 и заканчиваем балансировку. Потому что длина пути выше порога $\log\left(\frac{1}{1-\alpha}\right)^{-1}$ log n, здесь должен быть узел по пути, где количество листьев очень мало для высоты (не позднее, корень).

Мы наблюдаем, что высота привязана $\log\left(\frac{1}{1-\alpha}\right)^{-1}$, поддерживая таким методом через любую последовательность вставок. Если граница высоты была удовлетворена перед вставкой, затем после вставки она была нарушена более чем на единицу; но если она нарушена, затем несбалансированное поддерево будет найдено и оптимально перестроена, что сократит высоту этого поддерева по крайней мере на единицу.

Сейчас мы докажем амортизационную сложность вставки O(log n). Для этого мы представим потенциальную функцию на поисковых деревьях. Потенциал дерева - это сумма внутренних узлов абсолютной величины, и различия весов левых и правых поддеревьев. Потенциал любого дерева неотрицателен и единственная вставка изменит только потенциал узлов вдоль его пути поиска, каждого хотя бы на единицу, так что это увеличит потенциал дерева на $\log\left(\frac{1}{1-\alpha}\right)^{-1}\log n$. Но поддерево, которое подвергнется балансировке, первое вдоль пути, у которого высота слишком велика чтобы быть α-вес-сбалансированным, так что это α-вес-баланс в корне. Итак, это поддерево имеет потенциал как минимум (1-2 α)ω, если бы оно имело ω листья. Если мы выберем это дерево для балансировки, мы производим O(ω) работу, для получения «сверху-вниз» дерева на ω узлах.

Теорема: Оптимальное дерево «сверху-вниз» с ω листьями имеет потенциал не больше чем $\left(\frac{1}{2}\right)\omega$

Доказательство: В «сверху-вниз» оптимальном дереве, каждый внутренний узел имеет потенциал 0 или 1, в зависимости от количества листьев в поддереве чётных или нечётных. Но одно из нижних соседей нечётного узла должен быть чётным, так что по меньшей мере столько чётных узлов как нечётных узлов.

Итак балансировка уменьшает потенциал, по крайней мере, из $(1\text{-}2\,\alpha)\,\omega$ до $\frac{1}{2}\,\omega$. Значит, если $\alpha < \frac{1}{4}$, мы имеем $\Omega(\omega)$ уменьшение в потенциальном использовании O(ω) работы. Но среднее уменьшение через последовательность вставок не может быть больше чем среднее увеличение, так что средняя работа в единицу балансировки после вставки будет O(log n).

Для удаления ситуация намного проще; удаления не увеличивают высоту дерева, но очень медленно уменьшают нашу ссылку на меру для максимально позволительной высоты. Итак, для того чтобы сохранить ограничение высоты, мы иногда полностью перестраиваем дерево, когда достаточно много элементов были удалены, которым нужно уменьшение высоты на единицу. Для этого мы оставляем второй счётчик, который установлен на αb после полной перестройки дерева, когда оно имело n листьев. Каждый раз когда мы производим основное удаление, мы уменьшаем счётчик, и когда он достигнет 0, мы снова полностью перестраиваем дерево как оптимальное дерево «сверху-вниз». Когда счётчик достигнет 0, останется по крайней мере (1-α) листьев, возможно больше, если это были операции вставок. Итак, граница высоты не может уменьшиться больше чем на единицу с момента прошлой перестройки. Эта операция сохраняет границу высоты. Но его амортизированная сложность очень мала, только O(1) в единицу операции удаления, потому что мы производим одну полную перестройку, занимая O(n) время, на каждые Ω(n) операции. Конечно, амортизированное O(1) удаление не стоит подразумевать в любом превосходстве на O(log n) потому что амортизированная вставка стоит O(log n) и здесь так же много вставок как и удалений. Но мы амортизируем время обновления O(log n) очень простыми инструментами, только полная оптимальная перестройка «сверху-вниз», вместе с двумя глобальными счётчиками для числа листьев и числа недавних удалений.

Теорема: Мы можем осуществить частичную перестройку дерева поиска высоты не более чем $(\log\frac{1}{1-\alpha})^{-1}\log n$, для $\alpha \in [,\frac{1}{4}]$, с амортизированным O(log n) `insert` и `delete` операциями, без какой либо балансировочной информации в узлах.

Сохраняя биты балансировочной информации в узлах это не серьёзное практическое утверждение, но такая структура не должна выглядеть как альтернатива высотно-сбалансированным деревьям. Но как демонстрация мощности случайной реорганизации,

которая даёт только амортизированные границы, но которые также доступны на более сложных статических структурах данных, и в большинстве случаев лучшее решение, мы должны сделать статическую структуру динамической.

Основные понятия

Бинарное дерево цифрового поиска – дерево, в каждой вершине которого хранится полный ключ, а переход по ветвям происходит на основе значения очередного бита аргумента.

Двоичное (бинарное) дерево – иерархическая структура, в которой каждый узел имеет не более двух потомков.

Идеально сбалансированное дерево – дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1.

Ключ поиска – это поле, по значению которого происходит поиск.

Оптимальное бинарное дерево поиска – бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных.

Поиск – процесс нахождения конкретной информации в ранее созданном множестве данных.

Сбалансированное АВЛ дерево – дерево, для каждой вершины которого выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1.

Случайные деревья поиска –бинарные деревья поиска, при создании которых элементы вставляются в случайном порядке.

Упорядоченное двоичное дерево – двоичное дерево, в котором для любой его вершины x справедливы свойства: все элементы в левом поддереве меньше элемента, хранимого в *x*; все элементы в правом поддереве больше элемента, хранимого в *x*; все элементы дерева различны.

Красно-черное дерево (Red-Black-Tree, RB-Tree) – бинарное дерево со следующими свойствами:

- каждая вершина должна быть окрашена либо в черный, либо в красный цвет;
- корень дерева должен быть черным;

- листья дерева должны быть черными и объявляться как NIL-вершины;
- каждый красный узел должен иметь черного предка;
- на всех ветвях дерева, ведущих от его корня к листьям, число черных вершин одинаково.

Черная высота дерева – количество черных вершин на ветви красно-черного дерева от корня до листа.

Резюме

1. Поиск данных предполагает использование соответствующих алгоритмов в зависимости от ряда факторов: способ представления данных, упорядоченность множества поиска, объем данных, расположение их во внешней или во внутренней памяти.

2. Двоичные деревья представляют собой иерархическую структуру, в которой каждый узел имеет не более двух потомков. Поиск на двоичных деревьях не дает преимущества по времени в сравнении с линейными структурами.

3. Упорядоченное двоичное дерево – двоичное дерево, в котором для любой его вершины x справедливы свойства: все элементы в левом поддереве меньше элемента, хранимого в x; все элементы в правом поддереве больше элемента, хранимого в x; все элементы дерева различны. Поиск в худшем случае на таких деревьях имеет сложность $O(n)$.

4. Случайные деревья поиска представляют собой упорядоченные бинарные деревья поиска, при создании которых элементы (их ключи) вставляются в случайном порядке. Высота дерева зависит от случайного поступления элементов, поэтому трудоемкость определяется построением дерева.

5. Оптимальное бинарное дерево поиска – это бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных. Поиск на таких деревьях имеет сложность порядка $O(n^2)$.

6. Дерево считается сбалансированным по АВЛ, если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1. Алгоритмы поиска,

добавления и удаления элементов в таком дереве имеют сложность, пропорциональную $O(\log n)$.

7. В деревьях цифрового поиска осуществляется поразрядное сравнение ключей.

8. Особенности указателей в языке C++ позволяют строить динамические структуры памяти на основе статически объявленных переменных или на смеси статических и динамических переменных.

9. Циклический (кольцевой) список является структурой данных, представляющей собой последовательность элементов, последний элемент которой содержит указатель на первый элемент списка, а первый (в случае двунаправленного списка) – на последний.

10. Основными операциями с циклическим списком являются: создание списка; печать (просмотр) списка; вставка элемента в список; удаление элемента из списка; поиск элемента в списке; проверка пустоты списка; удаление списка.

11. Дек является структурой данных, представляющей собой последовательность элементов, в которой можно добавлять и удалять в произвольном порядке элементы с двух сторон. Первый и последний элементы дека соответствуют входу и выходу дека.

12. Частные случаи дека – это ограниченные деки.

13. Основными операциями с деком являются: создание дека; печать (просмотр) дека; добавление элемента в левый конец дека; добавление элемента в правый конец дека; извлечение элемента из левого конца дека; извлечение элемента из правого конца дека; проверка пустоты дека; очистка дека.

14. Красно-черные деревья являются одним из способов балансировки деревьев, что определяется свойствами данной структуры.

15. Над красно-черными деревьями можно выполнять все те же основные операции, что и над бинарными деревьями.

16. При вставке/удалении элемента необходима поддержка баланса дерева через проверку и перекрашивание узлов при необходимости.

3. ТЕХНИКА БЕЗОПАСНОСТИ ПРИ ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Меры безопасности при работе с электротехническими устройствами соответствуют мерам безопасности, принимаемым при эксплуатации установок с напряжением до 1000 В и разработанным в соответствии с «Правилами техники безопасности при эксплуатации электроустановок потребителей», утвержденными Главгосэнерго-надзором 21 декабря 1984 г.

Студенту не разрешается приступать к выполнению лабораторной работы, если замечены какие-либо неисправности в лабораторном оборудовании.

Студент не должен прикасаться к токоведущим элементам электрооборудования, освещения и электропроводке, открывать двери электрошкафов и корпусов системных блоков, мониторов.

Студенту запрещается прикасаться к неизолированным или поврежденным проводам и электрическим устройствам, наступать на переносные электрические провода, лежащие на полу, самостоятельно ремонтировать электрооборудование и инструмент.

Обо всех замеченных неисправностях электрооборудования студент должен немедленно поставить в известность преподавателя или лаборанта.

При выполнении лабораторной работы необходимо соблюдать осторожность и помнить, что только человек, относящийся серьезно к своей безопасности, может быть застрахован от несчастного случая.

4. ЗАДАЧИ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Сформировать простое бинарное дерево и АВЛ дерево поиска из случайного множества n элементов. Сравнить производительность поиска в обоих случаях.
2. Сформировать АВЛ дерево поиска из случайного множества n элементов. Реализуйте алгоритм удаления элемента из АВЛ-дерева.
3. В упорядоченном АВЛ-дереве с целочисленными ключами возведите в квадрат корневой элемент. Восстановите свойства АВЛ-дерева.
4. Найдите в АВЛ-дереве такие поддеревья, которые являются полными упорядоченными бинарными деревьями.
5. Найдите в АВЛ-дереве такое поддерево максимальной высоты, которое является полным упорядоченным бинарным деревом.

6. Имеется файл записей с некоторым ключевым полем. Постройте в оперативной памяти идеально сбалансированное бинарное дерево поиска и организуйте поиск указанных записей.

7. Составьте программу поиска записи с включением в сбалансированном бинарном дереве поиска (АВЛ-дерева).

8. Составьте программу вставки записи в сбалансированное бинарное дерево поиска (АВЛ-дерево) и балансировки его в случае необходимости.

9. Разработайте алгоритм и программу, работающую за $O(Logn)$ времени, для нахождения ключа k в сбалансированном по весу дереве. Считайте, что в каждом узле хранится мощность поддерева с корнем в этом узле.

10. Имеется несколько AVL деревьев. Разработайте алгоритм и программу поиска в этих деревьях. Затем разработайте алгоритм и программу конкатенации этих деревьев и поиска в нем. Сравните эффективность обоих подходов к поиску.

11. Написать программу формирования сбалансированного АВЛ-дерева из случайного множества n целочисленных ключей и определить его высоту.

12. Написать программы формирования двух сбалансированных АВЛ-деревьев из случайного множества, у которых разница высот равна 2. Объединить эти деревья в одно сбалансированное АВЛ дерево. Определить его высоту.

13. Написать программу формирования сбалансированного АВЛ-дерева из случайного множества n целочисленных ключей. Добавить в это дерево произвольный ключ, проверить сбалансированность и в случае необходимости сбалансировать его.

14. Написать программу формирования сбалансированного АВЛ-дерева из случайного множества n целочисленных ключей. Удалить из этого дерева произвольный ключ, проверить сбалансированность и в случае необходимости сбалансировать его.

15. Написать программу формирования сбалансированного RB-дерева из случайного множества n целочисленных ключей и определить его высоту.

16. Составьте программу удаления записи из сбалансированного бинарного дерева поиска (АВЛ-дерево) и балансировки его в случае необходимости.

17. Разработайте алгоритм и программу, работающую за $O(Logn)$ времени, для нахождения числа ключей, которые находятся между двумя произвольными ключами x и y в сбалансированном по весу дереве. Считайте, что в каждом узле хранится мощность поддерева с корнем в этом узле.

18. Написать программу формирования сбалансированного ВВ-дерева из случайного множества n целочисленных ключей с заданным показателем сбалансированности и определить его высоту.

19. Написать программу формирования сбалансированного ВВ-дерева из случайного множества n целочисленных ключей. Добавить в это дерево произвольный ключ, проверить сбалансированность и в случае необходимости сбалансировать его.

20. В первоначально пустое АВЛ-дерево были занесены (согласно стандартному алгоритму вставки) в указанном порядке следующие ключи: 20, 15, 9, 18, 40, 35, 51, 27, 37, 36. Сформировать и напечатать АВЛ-деревья, которые получаются после добавления каждого из этих ключей.

21.

5. УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНОЙ РАБОТЫ

Каждую задачу решить в соответствии с изученными алгоритмами поиска на основе деревьев, реализовав программный код на языке C++. Рекомендуется воспользоваться материалами лекции, где подробно рассматриваются описание используемых в работе алгоритмов, примеры их реализации на языке C++. Программу для решения каждой задачи разработать методом процедурной абстракции, используя функции. Этапы решения сопровождать комментариями в коде. В отчете отразить разработку и обоснование математической модели решения задачи и привести примеры входных и выходных файлов, полученных на этапе тестирования программ.

Каждую задачу реализовать в соответствии с приведенными этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;
- выбрать метод решения задачи, если это необходимо;

- разработать графическую схему алгоритма;
- записать разработанный алгоритм на языке C++;
- разработать контрольный тест к программе;
- отладить программу;
- составить отчет о лабораторной работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему поиск на бинарных деревьях не дает преимущество по сложности по сравнению с линейными структурами?

3. С какой целью производится балансировка деревьев?

4. Какое из деревьев: упорядоченное, случайное, оптимальное или сбалансированное по АВЛ – дает наибольшее преимущество по трудоемкости? Рассмотрите различные случаи.

5. Выполните левое малое вращение дерева, приведенного на рис 3.

6. Выполните левое большое вращение дерева, приведенного на рис 4.

7. Как выполняется балансировка элементов в упорядоченных после вставки или удаления элемента?

8. Всегда ли возможна балансировка упорядоченных деревьев? Ответ обоснуйте.

9. Как выполняется балансировка элементов в АВЛ-деревьях после вставки или удаления элемента?

10. На основании чего в красно-черном дереве самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу?

11. Куда может быть добавлен элемент в красно-черное дерево? Вид дерева при этом должен сохраниться.

12. Как можно охарактеризовать красно-черное дерево: полное, неполное, строгое, нестрогое?

13. Каким образом при удалении элемента из красно-черного дерева перекрашиваются узлы?

7. СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Кнут, Д. Искусство программирования для ЭВМ Т.1, Основные алгоритмы / Д. Кнут – М.: Вильямс 2000. – 215 с.

2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт – М.: МИР, 1989. – 363 с.

3. Кнут, Д. Искусство программирования для ЭВМ Т. 3. Сортировка и поиск / Д. Кнут – М: Вильямс, 2000. – 245 с.

4. Пападимитриу, Х. Комбинаторная оптимизация. Алгоритмы и сложность: пер. с англ. / Х. Пападимитриу, К.М. Стайглиц. – М.: Мир 1985. – 512 с.

5. Топп, У. Структуры данных в С ++: пер. с англ. / У. Топп, У. Форд. – М.: БИНОМ, 1999. – 816 с.

6. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М: МЦНМО, 1999. –960 с.

7. Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М.: Мир, 1982. – 416 с.

8. Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. – М.: Вильямс, 2000. – 384 с.

9. Кубенский, А. А. Создание и обработка структур данных в примерах на Java / А. А.Кубенский. – СПб.: БХВ-Петербург, 2001. – 336 с.

10. Седжвик, Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск: пер. с англ. /Р. Седжвик. – К.: ДиаСофт, 2001. – 688 с.

11. Хэзфилд, Р. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: пер. с англ. / Р. Хэзфилд, Л. Кирби [и др.]. – К.: ДиаСофт, 2001. – 736 с.

12. Мейн, М. Структуры данных и другие объекты в С++: пер. с англ. / М .Мейн, У. Савитч. – 2-е изд.– М.: Вильямс, 2002. – 832 с.

13. Хусаинов, Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си (+ CD): учеб. Пособие / Б.С. Хусаинов. – М.: Финансы и статистика, 2004. – 464 с.

14. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл. – 2-е изд. – М.: Техносфера, 2004. – 368с.

15. Гудман, С.Введение в разработку и анализ алгоритмов / С. Гудман, С.Хидетнием. – М.: Мир, 1981. – 206 с.

16. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж.Хопкрофт, Дж.Ульман. – М.: МИР, 1979. – 329 с.

17. Сибуя, М. Алгоритмы обработки данных / М. Сибуя, Т. Яматото. – М.:МИР, 1986. – 473 с.

18. Лэнгсам, Й.Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбаум. – М.: МИР, 1989. – 327 с.

19. Гулаков, В.К. Деревья: алгоритмы и программы / В.К. Гулаков. – М.: Машиностроение-1, 2005. – 206 с.

20. Гулаков, В.К. Многомерные структуры данных / В.К. Гулаков, А.О. Трубаков. – Брянск: БГТУ, 2010. – 387 с.

Структуры и алгоритмы обработки данных. Сбалансированные деревья: методические указания к выполнению лабораторной работы №5 для студентов очной, очно-заочной и заочной форм обучения по направлениям подготовки 230100 «Информатика и вычислительная техника», 010500 «Математическое обеспечение и администрирование информационных систем», 231000 «Программная инженерия»

ВАСИЛИЙ КОНСТАНТИНОВИЧ ГУЛАКОВ

Научный редактор В.В. Конкин
Редактор издательства Л.Н. Мажугина
Компьютерный набор В.К. Гулаков

Темплан 2013 г., п.202

Подписано в печать Формат 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 1,33 Уч.-изд.л. 1,33 Тираж 40 экз. Заказ Бесплатно

Издательство Брянского государственного технического университета
241035, Брянск, бульвар им.50-летия Октября, 7, БГТУ, тел. 58-82-49

Лаборатория оперативной полиграфии БГТУ, ул. Институтская, 16