Treaps и T-Treaps

Оглавление

1. Randomized Treaps

1.1 Treaps

В этой части статьи мы рассмотрим двоичные деревья, у которых каждая вершина содержит как ключ, так и приоритет. В наших примерах буквы будут использоваться как ключи, а числа как приоритеты. *Treap* – это двоичное дерево, где ключи расположены как в *BST* (*binary search tree*), а приоритет каждой вершины меньше, чем приоритет её детей. Другими словами, *treap* – это одновременно и *BST* для ключей и *heap* для приоритетов.

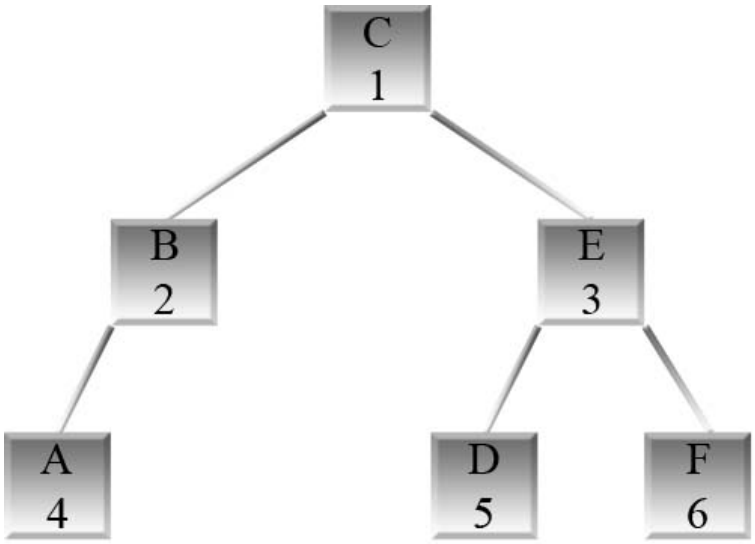

Рис. 1. *Treap*. Верх каждой вершины – её ключ, низ – приоритет.

Далее предположим, что все ключи и приоритеты различны. Благодаря этому мы можем легко убедиться по индукции, что строение *treap* полностью определяется ключами и их приоритетами. Так как это *heap*, то вершина *V* с 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 *treap*, то по индукции по высоте получаем, что структура *treap* полностью определена. Базой является пустой *treap*.

Другой способ для описания структуры основывается на том, что *treap* – это *BST*, которое получается вставкой ключей в порядке возрастания их приоритетов. Третий вариант для описания *treap* отождествляет ключи и приоритеты с координатами множества точек на

плоскости. Корень *treap* 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, но их рассмотрение выходит за рамки данной статьи. Для подробного изучения этого вопроса используйте [1].

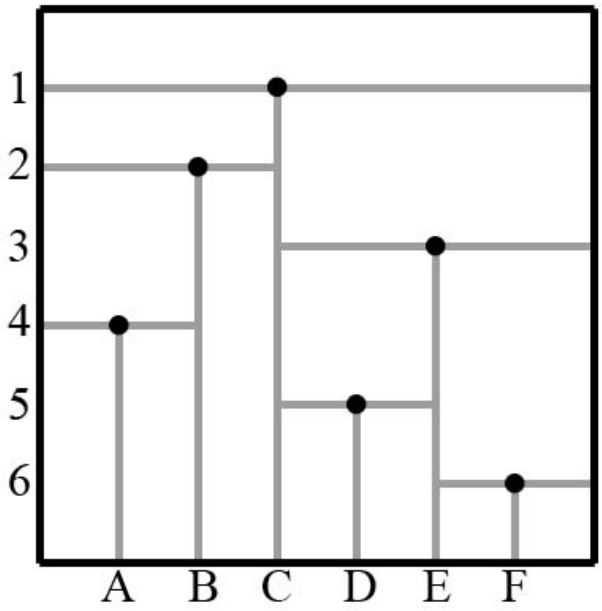

Рис. 2. Геометрическая интерпретация *treap*, показанного на рис. 1.

Treap'ы были впервые открыты Жаном Вуйлеменом (Jean Vuillemin) в 1980 году, но он назвал их декартовыми деревьями. 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*). *Treap*'ы были заново изучены и использованы Арагоном (Cecilia Aragon) и Зайделем (Raimund Seidel) для построения рандомизированных деревьев поиска в 1989 году. 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.

1.2 Операции BST

Поиск происходит так же, как и в *BST*. Время успешного поиска пропорционально глубине вершины. Время неудачного поиска пропорционально глубине её предшествующего или последующего элемента (в порядке возрастания ключей). Для вставки нового ключа *K*,

мы начинаем использовать стандартный алгоритм для *BST*. Когда ключи станут образовывать правильно построенное *BST*, приоритеты могут нарушать *heap-order*. Чтобы восстановить его, нужно делать простые повороты с участием вершины *K*, пока у её родителя приоритет не станет меньше, чем у неё. Время работы пропорционально глубине *K* до начала поворотов. Сами повороты выполняются за постоянное время, и их количество не превышает глубины *K*.

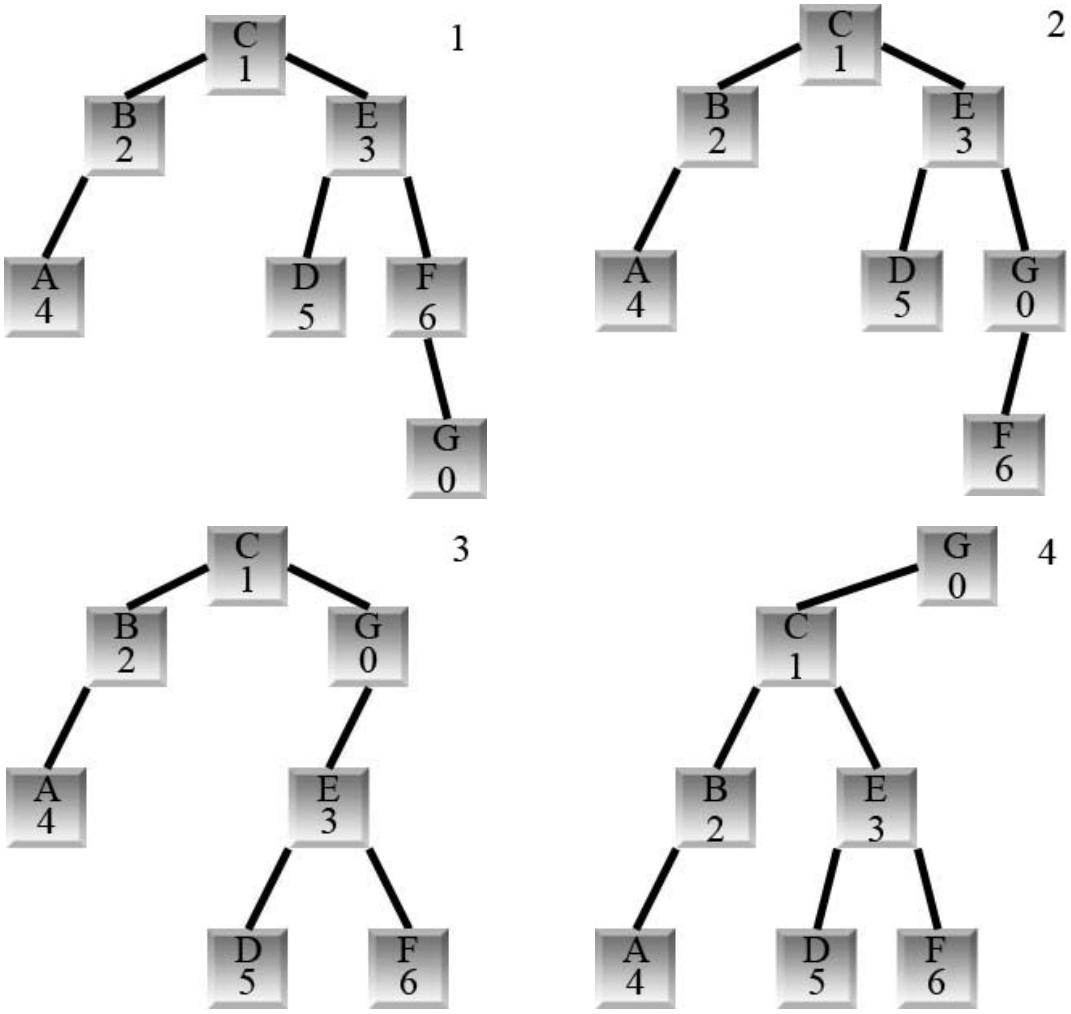

Рис. 3.

На рисунке 3 (в порядке 1, 2, 3, 4) представлено добавление *(G, 0)*: вставляем её как в *BST* и восстанавливаем *heap-order* поворотами с её родителями. Если смотреть в порядке убывания номеров картинок, то получим удаление вершины *(G, 0)*: спускаем её вниз поворотами, чтобы сделать листом, и удаляем её.

Удаление вершины происходит так же, как добавление, только в обратном порядке. Предположим, мы хотим удалить вершину *V*. Пока *V* не является листом, делаем повороты *V* с её ребёнком, имеющим меньший приоритет. Это действие опускает *V* на более низкий

уровень и поднимает её ребёнка. Выбор нужного ребёнка при поворотах позволяет сохранить *heap-order* во всех вершинах кроме *V*. Когда *V* станет листом, просто отрежем её.

Иногда нам требуется разделить *treap T* на 2 *treap*'а *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 *treap*'а, а левое и правое поддерево будут представлять собой *T1* и *T2* соответственно. Время для разделения *treap* примерно равно времени двух неудачных поисков для *K*.

Подобным образом нам бы хотелось научиться объединять *treap*'ы *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.

1.3 Анализ

Стоимость каждой операции пропорциональна глубине *d(V)* какой-то вершины *V* в *treap*.

- Поиск: Успешный поиск ключа *K* займёт *O(d(V)),* где *key(V) = K*. Для неуспешного поиска, обозначим *Vleft* и *Vright* предыдущий и следующий элементы *K* в порядке возрастания ключей. Так как последняя обработанная вершина будет либо *Vleft*, либо *Vright*, то время неуспешного поиска будет занимать соответственно *O(d(Vleft))* или *O(d(Vright))*.

- Удаление/Вставка: Вставка нового элемента *V* будет занимать *O(d(Vleft))* или *O(d(Vright))*. Удаление элемента – это просто вставка в обратном порядке.

- Разделение/Объединение: Разделение *treap*'а по ключу *K* займёт *O(d(Vleft))* или *O(d(Vright)),* так как разделение получается вставкой вспомогательной вершины с ключом *K* и приоритетом -∞. Объединение *treap*'ов займет столько же времени, сколько и разделение, потому что повторяет действия разделения в обратном порядке.

Так как глубина вершины в *treap*'е в худшем случае составляет *Θ(n),* где *n* – общее количество вершин, то каждая из операций в худшем случае будет выполняться за *Θ(n).*

1.4 Случайные приоритеты

Рандомизированное дерево двоичного поиска – это *treap*, 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 потребуется для анализа структуры *treap*'а. 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 *treap*'е. Здесь мы докажем, что средняя глубина каждой вершины составит $O(\log n)$. Это показывает, что ожидаемое время работы каждой из операций будет $O(\log n)$.

Пусть $x[k]$ будет обозначать вершину с k-ым по величине ключом. Определим индикаторную функцию A, принимающую значения 0 и 1 (ложь и истина):

$$A(i, k) = (x[i] - \text{предок } x[k])$$

Воспользуемся тем, что $d(V)$ - это количество предков V, отсюда получим следующее равенство:

$$d(x[k]) = \sum_{i=1}^{n} A(i, k)$$

Теперь мы можем выразить математическое ожидание глубины вершины, используя введённую индикаторную функцию:

$$E(d(x[k])) = \sum_{i=1}^{n} Pr[A(i, k) = 1]$$

Здесь мы используем линейность E и то, что $E[X] = Pr[X=1]$ для индикаторных значений X. Итак, для подсчёта средней глубины вершин нам нужно сосчитать вероятность того, что вершина i является предком вершины k.

К счастью, это легко доказывается с помощью следующей леммы. Для этого введём некоторые обозначения: $X(i, k)$ – множество ключей $\{x[i],\dots x[k]\}$ или $\{x[k],\dots x[i]\}$, в

зависимости от результата сравнения *i* и *k (i < k* или *i > k). X(i, k)* и *X(k, i)* обозначают одно и тоже, их мощность равна *|k - i| + 1*.

Лемма: Для любых $i \neq k$, *x[i]* является предком *x[k]* тогда и только тогда, когда *x[i]* имеет наименьший приоритет среди *X(i, k).*

Доказательство: Если *x[i]* является корнем, то оно является предком *x[k]* и по определению имеет минимальный приоритет среди всех вершин, следовательно, и среди *X(i, k).*

С другой стороны, если *x[k]* – корень, то *x[i]* – не предок *x[k],* и *x[k]* имеет минимальный приоритет в *treap*'е; следовательно, *x[i]* не имеет наименьший приоритет среди *X(i, k).*

Теперь предположим, что какая-то другая вершина *x[m]* – корень. Тогда, если *x[i]* и *x[k]* лежат в разных поддеревьях, то $i < m < k$ или $i > m > k$, следовательно, *x[m]* содержится в *X(i , k).* В этом случае *x[i]* – не предок *x[k]*, и наименьший приоритет среди *X(i, k)* имеет вершина с номером *m*.

Наконец, если *x[i]* и *x[k]* лежат в одном поддереве, то доказательство применяется рекурсивно, так как это поддерево является меньшим *treap*'ом. Пустой *treap* есть тривиальная база. □

Так как каждая вершина среди *X(i, k)* может иметь минимальный приоритет, мы немедленно приходим к следующему равенству:

$$\Pr[A(i,k)=1]=\begin{cases}\frac{1}{k-i+1}, & i<k\\ 0, & i=k\\ \frac{1}{i-k+1}, & i>k\end{cases}$$

Чтобы посчитать среднюю глубину *x[k],* мы просто подставим полученную вероятность в нашу формулу и упростим выражение:

$$E[d(x[k])]=\sum_{i=1}^{n}\Pr[A(i,k)=1]=$$

$$=\sum_{i=1}^{k-1}\frac{1}{k-i+1}+\sum_{i=k+1}^{n}\frac{1}{i-k+1}=$$

$$=H_k-1+H_{n-k}-1<\ln k+\ln(n-k)-2<2\ln n-2$$

Подытожим полученные результаты: каждое удаление, вставка, поиск, разделение и объединение в *n*-вершинном рандомизированном *treap*'е занимает *O(log n)*.

Так как *treap* – это двоичное дерево, которое получается добавлением ключей в порядке возрастания приоритетов, то рандомизированный *treap* 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).

1.5 Использование

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

2. T-Treaps

2.1 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.

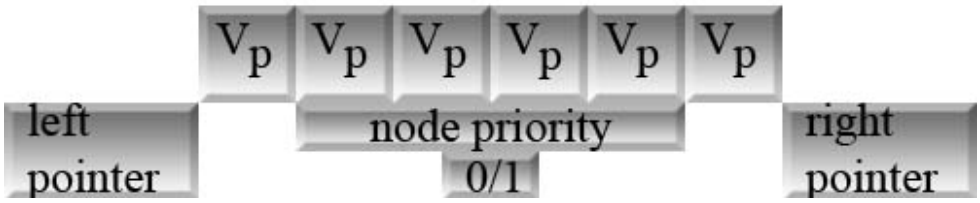

Рис. 4. Представление вершины в *T-Treap*'е: *V* – значения, *p* – их приоритеты.

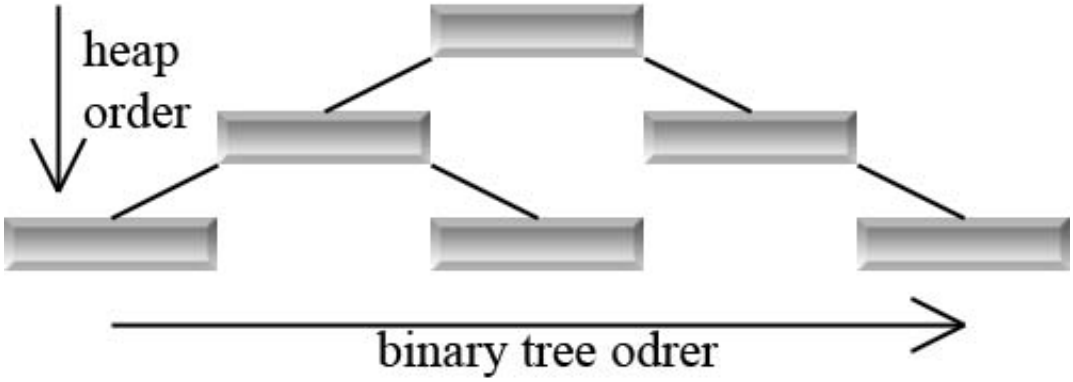

Рис. 5. Общее представление *T-Treap*'а.

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

2.2 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.

2.3 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 *heap-order*'у. 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.

2.4 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.

2.5 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.

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.

Визуализаторы

9. Randomized binary search trees.

10. Рандомизированное дерево поиска с приоритетами.