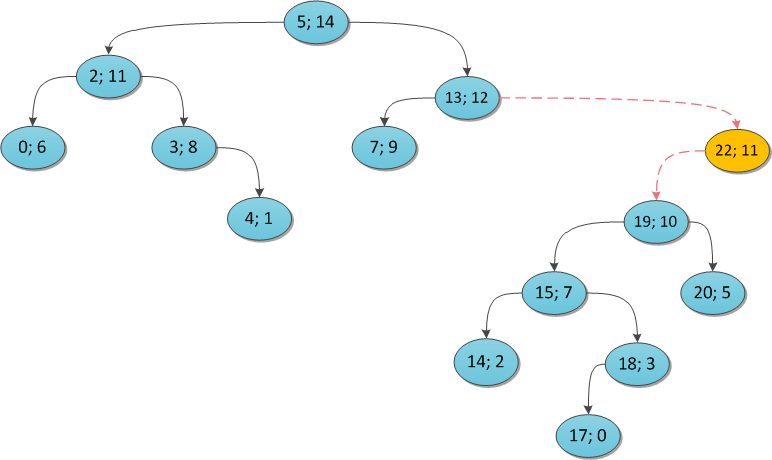
**Treap**

1. ***Что такое Treap?***

Treap – это структура данных, объединяющая в себе двоичное дерево поиска (BST) и бинарную кучу (Binary Heap). На русском название декартово дерево, лично для меня ничего говорит о самом дереве. Есть другие названия, которые больше подходят этому дереву – дерамида (дерево + пирамида) или курево (куча + дерево). Treap очень близок с рандомизированным бинарным деревом поиска (Randomized Binary Search Tree).

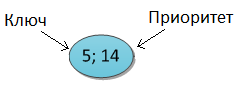
В оригинале – это сочетание слов tree + heap.



В узлах Treap содержатся две пары значений (x, y)

где x – ключ – key (основное значение, пример: элемент массива)

y – приоритет - priority (некий коэффициент)



Значение X взято из известного нам двоичного дерева поиска, а значение Y – из кучи.

Treap решает одну из известных проблем двоичного дерева поиска, когда все значения в силу отсортированного массива, помещает все значения на одну ветку, что делает работу с таким деревом при полиномиальной, вместо логарифмической.

Treap удается решить данную проблему с помощью параметра приоритета Y. В обычном дереве поиска задать параметр приоритета Y можно идя слева направо – сверху вниз. Но Treap позволяет задать для каждого узла свой приоритет. Тем самым настроить дерево под нужды. Для наибольшей производительности параметр приоритета Y лучше всего рандомизировать. При этом диапазон чисел для рандомизации выгоднее всего сделать максимально большим, чтобы числа не повторялись. Рандомизация чисел обеспечивает для нашего дерево близкое к logn высоту дерева. В среднем случае высота дерева примерно в 2-3 раза больше чем logn, что очень близко к logn.

Существует множество различных видов Treap: обычные Treap, FHQ Treap, T-Treap,

1. ***Сложность***

Treap, скажем так, является всего лишь модификацией известной всеми BST. Скрещение с Binary Heap помогает дереву приблизиться к заветной logn сложности по времени выполнения типичных алгоритмов для всех деревьев: *вставка, удаления и поиск.*

|  |  |  |
| --- | --- | --- |
| Операции | Среднее значение | В худшем случае |
| Поиск | O(logn) | O(n) |
| Вставка | O(logn) | O(n) |
| Удаление | O(logn) | O(n) |

Но, это модификация не обеспечивает 100-процентую логарифмическую сложность, как было сказано ранее из-за рандомизации или неправильной расстановки приоритетов. Поэтому худший случай останется O(n), хотя и очень редко.

В отличие от Randomized Binary Search Tree нашему гостю нужно хранить в приоритетах большие числа для наибольшей точности.

По памяти в среднем сложность остается такой же, как и в BST, но с большей константой из-за хранения случайных больших чисел.

При этом Treap можно модифицировать с помощью Ленивого распространения (Lazy Propagation). По аналогии с деревом отрезков (Segment Tree), LP позволяет хранить некоторые элементы, для скоростной работы, связанной с проходом по дереву (range). Ну и конечно никто не отменял кэшировать приоритеты, это тоже дает некий буст по скорости и по памяти даже с тем же красно-черным деревом (Red-Black Tree).

1. ***Плюсы***

К явному плюсу можно отнести легкость реализации. Реализовать Treap проще из-за схожести с BST и Binary Heap. Это очень может пригодиться при сортировке массива, когда требуются простые операции, такие как удаление или вставка, что такое некий адвантедж. Поэтому практически никогда не следует имплементировать BST, ведь можно просто сделать модификацию к нему, использовав Treap, из-за более лучшего среднего случая.

Другим плюсом является очевидность и предсказуемость действий над деревом. Всегда можно предугадать операцию вставки или удаления в любой конкретный момент, в отличие от RBST.

В Интернете можно найти многое об основных задачах Treap, такие как поиска минимума или максимума, хотя это и неверно, так как для этого есть другие, более быстрые деревья, такие как *дерево отрезков* (Segment Tree) или *дерево Фенвика* (Fenwick Tree).

Все-таки главная задача, для которой стоит использовать Treap – это реверсирование по области (Reversing an Arbitrary Interval). А также операцию, по увеличению чисел на определенной области (Adding by Range). Для этих операций и служит модификация в виде LP. Обе операции выполняются в среднем за logn.

Treap также довольно удобно пользоваться для операций *слияния* (Split) и *объединения* (Merge). Обе операции не просто выполняются за logn, но и поддерживают обработку во время выполнения операции в реальном времени за тот же logn.

Еще одним приятным свойством Treap является его каноническое представление, если приоритет элемента вычисляется с помощью последовательной хэш-функции над этим элементом. Канонические представления имеют важные варианты использования, особенно в области аутентифицированных структур данных /вычислений. Кроме того, их легко сделать постоянными, что делает их подходящими для контроля версий.

1. ***Минусы***

Пришло время ругать наш Treap.

При использовании Treap мы распределяем приоритеты исключительно надеясь на удачу, присваивая рандомное число. Существует возможность получить ряд из восходящих приоритетов, что совокупи с недостатками BST может породить сложность по времени O(n). Используя кэширование возможность возникновения этого случая уже намного меньше. Но, как и в рандомизации вероятность встретить коллизии также есть.

Treap не является самобалансирующимся, поэтому Treap приходиться постоянно балансировать самим, используя повороты (Left and Right Rotations).

Treap не обладают динамической оптимальностью, поэтому в этом они уступают в поддержке LCT, Splay и Segment деревьев.

При этом не используя LP в среднем для операций Merge/Split выделяется дополнительно logn дополнительной памяти, что не есть хорошо, ведь с такой же задачей лучше справляется Splay Tree за O(1) дополнительной памяти в худшем случае.