

|  |
| --- |
| DATA STRUCTURE  CHALLENGE ASSIGNMENT |
|  |
| Submitted to : Sir Jamal Hussain  Authored by: Summiya Zulfiqar (EP-1851048) MCS Final (Department of Computer Science) |

Treap:

Treap is a data structure which combines binary tree and binary heap (hence the name: tree + heap ⇒⇒ Treap).

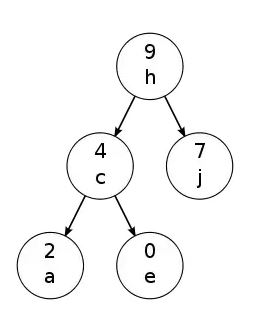
Treap is a Balanced Binary Search Tree

The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).

Treaps have been proposed by Siedel and Aragon in 1989.

# Treap

|  |
| --- |
| More specifically, Treap is a data structure that stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y. Assuming that all X and all Y are different, we can see that if some node of the tree contains values (X0X0, Y0Y0), all nodes in the left subtree have X<X0X<X0, all nodes in the right subtree have X>X0X>X0, and all nodes in both left and right subtrees have Y<Y0Y<Y0. |



The numbers indicate the heap arrangement of the data structure (arranged in max heap order), and the alphabets indicate the tree part of things (left child < parent <= right child).

Every node of Treap maintains two values.  
1) Key Follows standard BST ordering (left is smaller and right is greater.  
2) Priority randomly assigned value that follows Max-Heap property.

# Treap Operations:

# Insert in Treaps:

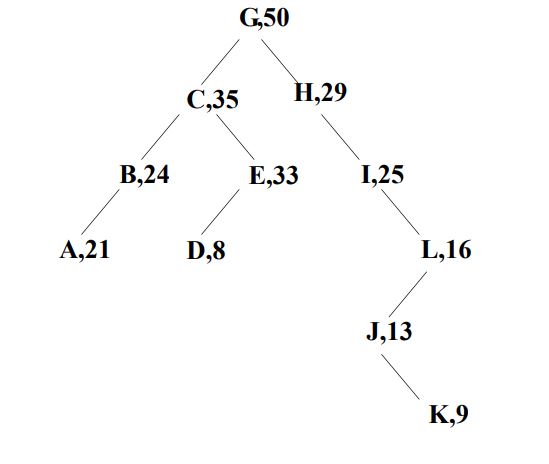
To insert a key-priority pair (K, P) into a Treap, do the following:

* Insert the pair as a new leaf, using the usual BST insert algorithm which pays attention to the key value K
* Then rotate the node up using AVL rotations as necessary, until the priority of its parent is greater than or equal to P, or the node becomes the root
* To rotate up, use a left rotation if the node is a right child of its parent, a right rotation if it is a left child

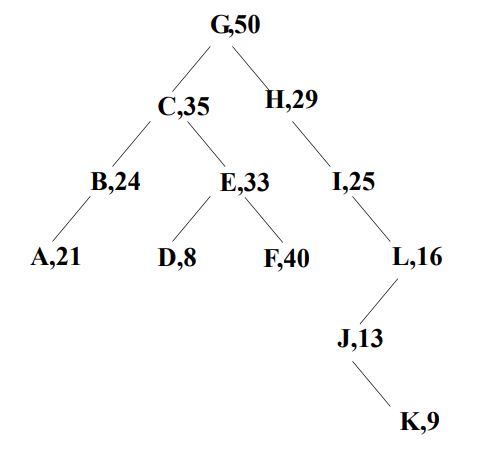
Since AVL rotations are constant-time operations, insert in a Treap can be performed in time O(H), where H is the height of the Treap.

For Example:

Insert the (key, priority) pair (F, 40) in this Treap:

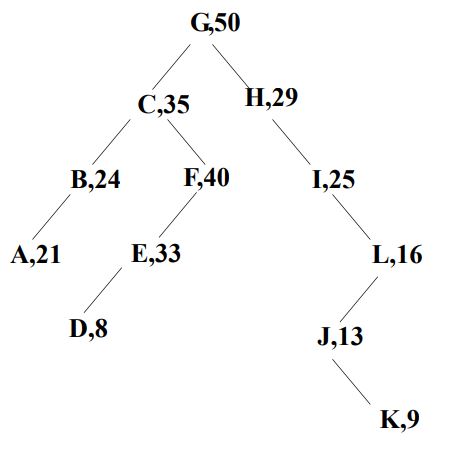


Ordinary BST insert gives this:



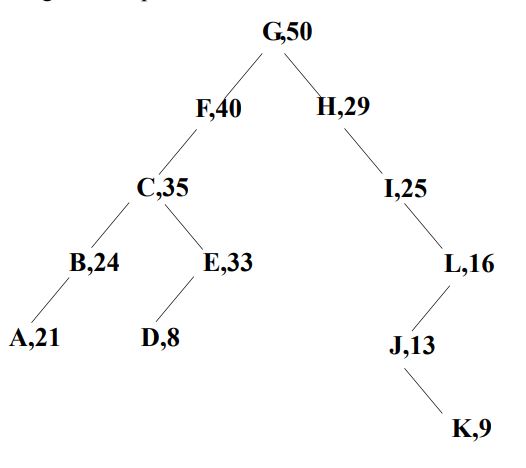
But the heap ordering property is violated. Need to rotate up to correct it.

One AVL left rotation gives this:



But the heap ordering property is still violated. Need to rotate again to correct it

Now the tree is again a Treap:



# Delete in Treaps:

To delete a key K, do the following:

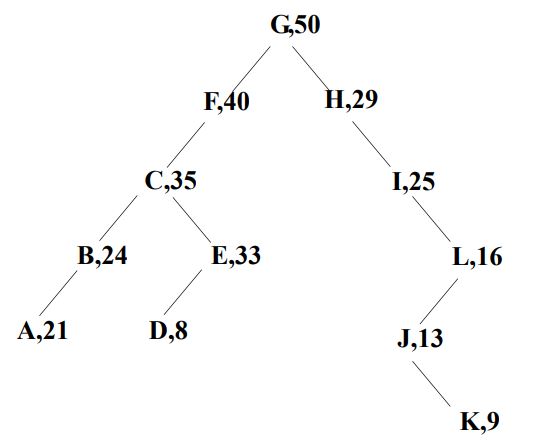
* Search for the node X containing K using the usual BST find algorithm.
* If the node X is a leaf, just delete the node (unlink it from its parent).
* Otherwise, use AVL rotations to rotate the node down until it becomes a leaf; then delete it.
* (If there are 2 children, always rotate with the child that has the larger priority, to preserve heap ordering: use a left rotation if the right child has larger priority, right rotation otherwise)

Since AVL rotations are constant-time operations, delete in a Treap can be performed in time O(H), where H is the height of the Treap.

(Note that the AVL-rotation-to-leaf trick also works for delete in ordinary BST’s).

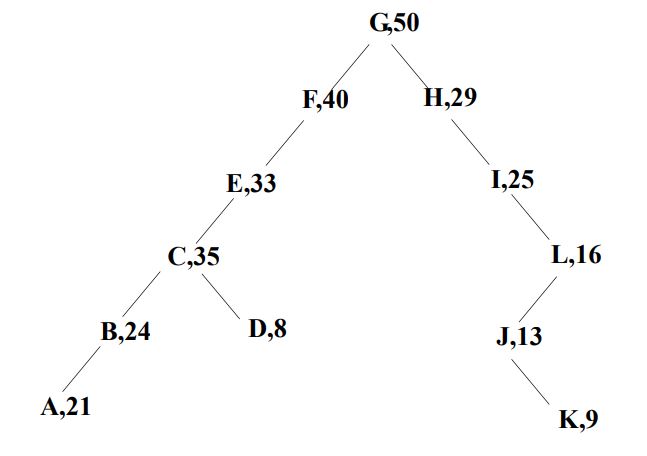
For Example:

Delete the key C from this Treap:



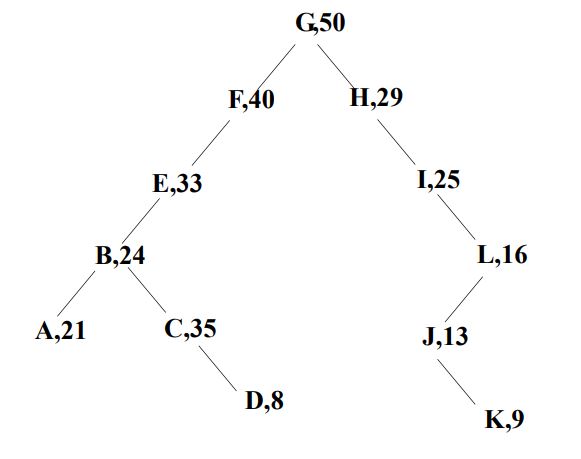
Find the node containing C. It is not a leaf, so need to rotate it down

Rotating the node with its larger priority child gives this:



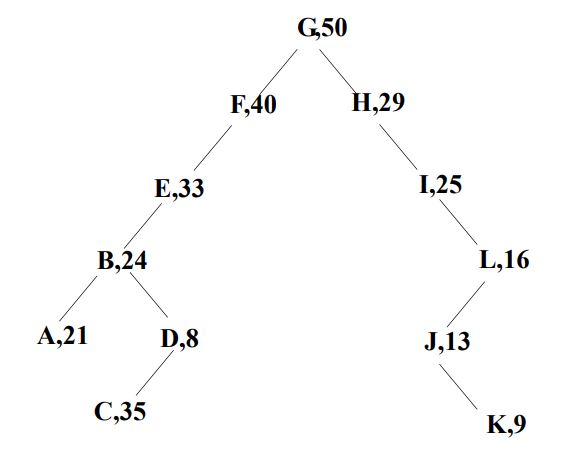
The node containing C is still not a leaf, so need to rotate down again

Rotating the node with its larger priority child now gives this:



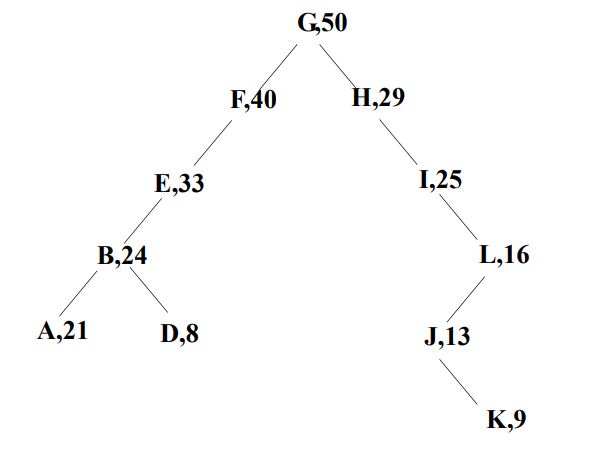
The node containing C is still not a leaf, so need to rotate down yet again

Rotating the node with its child now gives this:



Now the node containing C is a leaf, so just delete it

After clipping off the leaf, the result is again a Treap:



# Tree splitting:

The tree splitting problem is this:

* Given a tree and a key value K not in the tree, create two trees: One with keys less than K, and one with keys greater than K

This is easy to solve with a Treap, once the insert operation has been implemented:

* Insert (K,INFINITY) in the Treap
* Since this has a higher priority than any node in the heap, it will become the root of the Treap after insertion
* Because of the BST ordering property, the left subtree of the root will be a Treap with keys less than K, and the right subtree of the root will be a Treap with keys greater than K. These subtrees then are the desired result of the split

Since insert can be done in time O(H) where H is the height of the Treap, splitting can also be done in time O(H).

# Tree joining:

The tree joining or merging problem is this:

* Given two trees T1, T2, such that each key in T1 is less than all keys in T2, create a new tree T that contains all and only the keys from T1 and T2.

This is easy to do with a Treap, once the delete operation has been implemented:

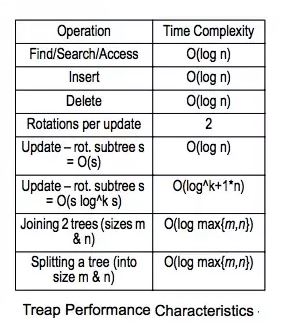
* Create a “dummy” node with any key value and any priority
* Make the root of T1 be the left child, and the root of T2 be the right child, of this dummy node
* Perform a delete operation on the dummy node

Since delete can be done in time O(H) where H is the height of the Treap, joining can also be done in time O(H).

# Why use Treaps:

Treaps are worth using because

* They permit very easy implementations of split and join operations, as well as pretty simple implementations of insert, delete, and find.
* They are the basis of randomized search trees, which have performance comparable to balanced search trees but are simpler to implement.
* They also lend themselves well to more advanced tree concepts, such as weighted trees, interval trees, etc.



# Advantages of Treaps:

* It’s a self-organizing data structure, as explained above. They look after themselves, without the need to be managed. Only, unlike other self-balancing trees, they don’t need complex algorithms (simple tree rotations will suffice, although there are simpler algorithms involving arrays that can do the job too).
* It is also a randomized data structure.
* It’s a binary search tree essentially, so to print a sorted order of keys, just traverse it in order like we would regular BSTs. Searching a Treap is like searching a tree.
* Regardless of the order of how we add, delete, etc., because of the randomized weights, there is a high probability that the tree will be balanced (a random binary search tree has logarithmic height).
* A Treap basically combines probability, randomization, two popular CS data structures, etc. to make a highly useful data structure. From a CS student’s point of view, try to learn about Treaps first, and we get to learn BSTs, traversals, hashing, tree rotations, heaps, recursion, sorting, etc., all in one go.

# Disadvantages of Treaps:

* Treaps permit easy implementations of find, insert, delete, split, and join operations.
* All these operations take worst case time O(H), where H is the height of the Treap.
* However, Treaps (like BST’s) can become very unbalanced, so that H = O(N), and that’s bad.
* Maintaining a strict balance condition (like the AVL or red-black property) in a Treap would be impossible in general, if the user supplies both key values and priorities: remember that Treaps with unique keys and priorities are unique.
* However, if priorities are generated randomly by the insert algorithm, balance can be maintained with high probability and the operations stay very simple.