Abhijeet Padhy

Roll: 214101001

DS LAB Assignment 4

# INDEX

1. Treap Data Structure Implementation
   1. Introduction
   2. Insert
   3. Delete
   4. Search Key
2. Performance Comparison of Various Data Structures

# Treap

## Introduction

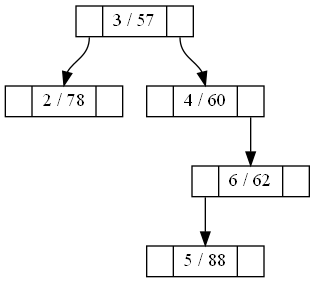
A treap is a binary tree whose nodes contain both **a key and a priority**. A treap has the **BST ordering property with respect to its keys,** **and the heap ordering property with respect to its priorities**.

### Node Structure

A node of a treap consists of **a key value** and **a priority**. In the diagram shown below, 15 is the key value and 69 is the priority associated with it. Apart from that **a pointer to left and right child** is also stored.

C:\Users\User\Desktop\IIT Guwahati\DS Lab\Assignments\Ass3\DSLAB_Assignment3\graph.png

It can be seen in the below diagram that 2, 3, 4, 5 and 6 are the key values and 78, 57, 60, 88 and 62 are their respective priorities. It can be seen that **BST ordering property** is maintained with respect to its keys, and the **heap ordering property** is maintained with respect to its priorities. We maintain a **min-heap structure**. This means the node with **smallest value of priority** is considered here as having the **highest priority** and will be present at the top.



## INSERT

Whenever we have to insert an element with a key k into the treap, we call the function:

**insert(k)**

This function generates a random number between 0 and 99 to be used as a random priority for that element. Let this be called prio. Then the below function is called:

**insert(root, k, prio)**

We insert the pair as a new leaf using the BST insert algorithm using the key value k. Then we rotate the newly inserted node up using AVL rotations as necessary, until the priority of its parent is less than or equal to prio, or the node becomes the root.

**insert(root, k, prio)**

In the above function call, if **k is less than root->key**, we recur the procedure on the left child. If after the recursive call, **root->priority is greater than root->LChild->priority**, we do a right rotation and return the left child of the node before rotation.

If **k is greater than root->key**, we recur the procedure on the right child. If after the recursive call, **root->priority is greater than root->RChild->priority**, we do a left rotation and return the right child of the node before rotation.

If **k is equal to root->key**, we don’t do anything and simply return the root.

If **root is equal to NULL** this means that we have reached the position where the new node has to be created. Thus we create a new node and return it.

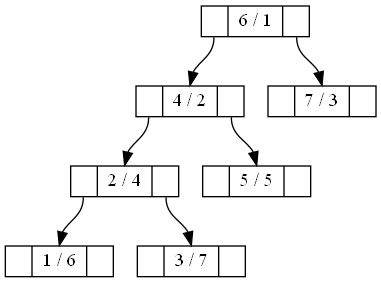
This completes the logic of the implementation of the insert function on treap data structure.

### Examples

**Example 1:**



Insert key 3 with priority 7.

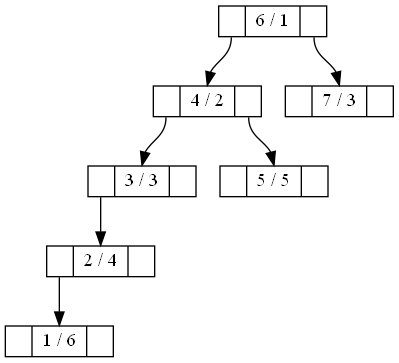


The node 3/7 is inserted as right child of 2/4 and there is no need of rotation.

**Example 2:**



Insert key 3 with priority 3.

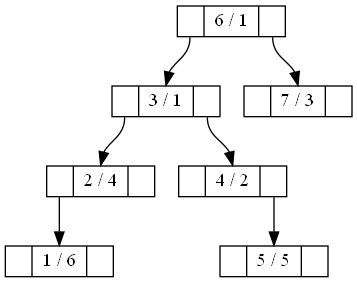


The node 3/3 is inserted as right child of 2/4 and there is a left rotation on node 2/4.

**Example 3:**



Insert key 3 with priority 1.

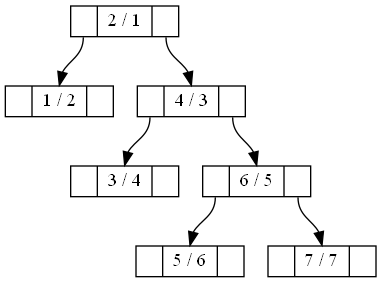


The node 3/1 is inserted as right child of 2/4 and there is a left rotation on node 2/4. Then there is a right rotation on the node 4/2. Thus we get the above treap.

**Example 4:**



Insert key 5 with priority 6.

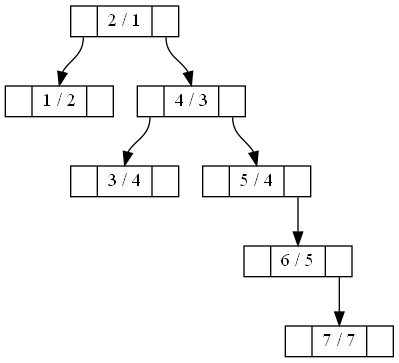


The node 5/6 is inserted as left child of 6/5 and there is no need of rotation.

**Example 5:**



Insert key 5 with priority 4.

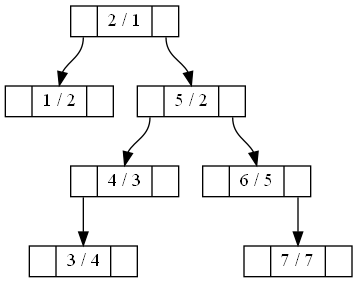


The node 5/4 is inserted as left child of 6/5 and there is a right rotation on node 6/5.

**Example 6:**



Insert key 5 with priority 2.



The node 5/2 is inserted as left child of 6/5 and there is a right rotation on node 6/5. Then there is a left rotation on the node 4/3. Thus we get the above treap.

## DELETE

In order to delete an element from the treap, we first search for the element. Then we increase the priority of the element to be deleted to MAX\_INT. Since we consider a min heap structure, this element should be moved to the bottom. Thus we keep on performing left or right rotation on the element until we move this element to the bottom. At the end it becomes a leaf node and we simply delete the node.

We call the following method on the root element:

**delete\_key(treap\_node \*root, int k)**

If root is NULL, we return NULL and do nothing. This happens when the element is not present in the treap.

If k is less than root->key, we recur on the left child. Similarly if k is greater than root->key, we recur on the right child.

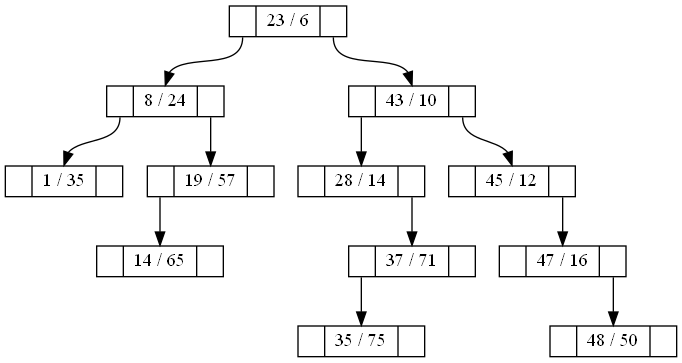
If k is equal to root->key, this is the node to be deleted. We increase the priority of this element to INT\_MAX and then call

**handle\_priority\_downwards(root)**

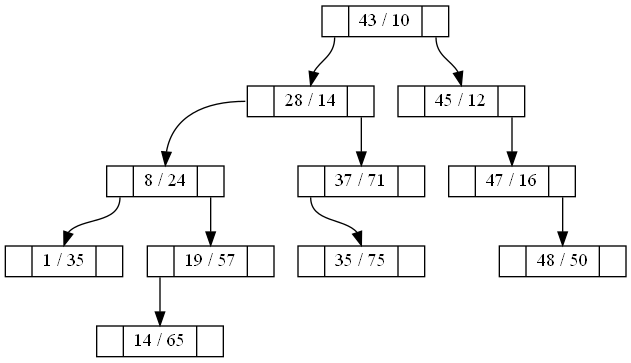
The purpose of this function is to move the element to be deleted downwards the treap until it becomes the leaf. We find the priority of left child and right child of the node passed in argument. If priority of left child is less than that of right child, we perform a right rotation. Otherwise we perform a left rotation. After the rotation is over, we call this function again on the element to be deleted. This way, the element to be deleted is moved down. At the end of each rotation, we return the new root after rotation is performed.

### Examples

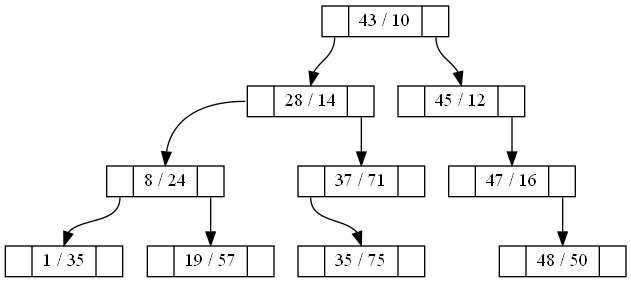
**Example 1:**



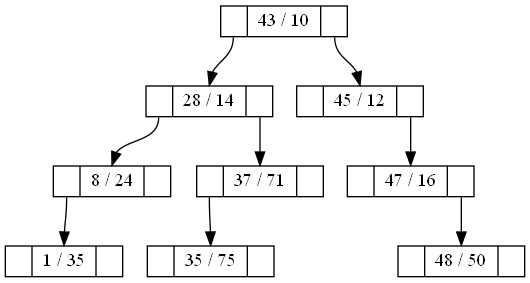
Delete 23 which is the root node.



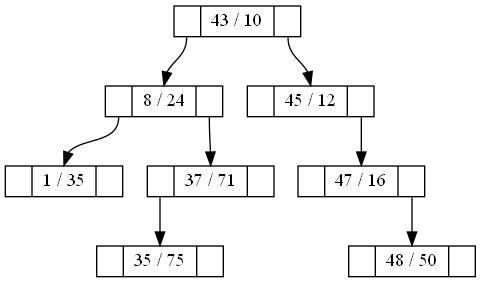
Delete 14 which is a leaf node:



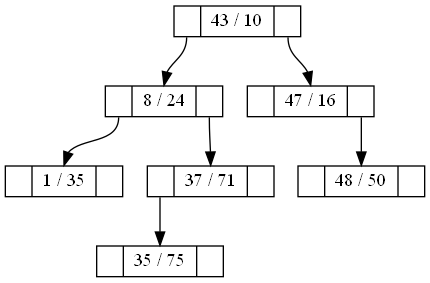
Delete 19 which is a leaf node:



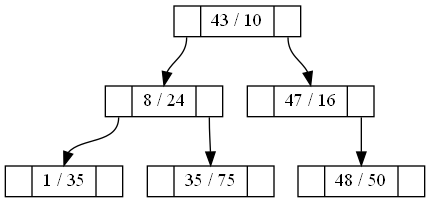
Delete 28 which is an intermediate node:



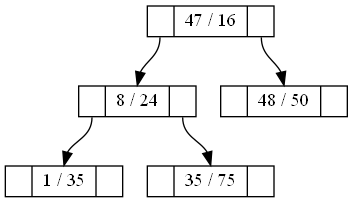
Delete 45 which is an intermediate node with right child only.



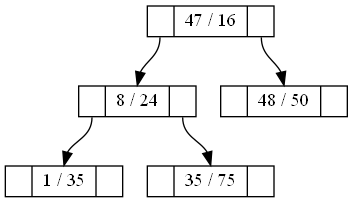
Delete 37, which is an intermediate node with left child only:



Delete 43:



Delete 24, an element which is not present:



Delete 8:

