Abhijeet Padhy

Roll: 214101001

DS LAB Assignment 4

# INDEX

1. Treap Data Structure Implementation
   1. Introduction
   2. Insert
   3. Delete
   4. Search Key
2. Generate Test Case and read from file
3. Performance Comparison of Various Data Structures
   1. Total Height of the Tree
   2. Average Height of each node of the tree
   3. Number of comparisons in insert and delete
   4. Number of Rotations in insert and delete

# Treap

## Introduction

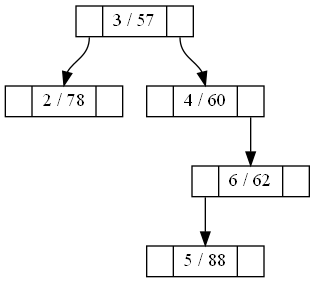
A treap is a binary tree whose nodes contain both **a key and a priority**. A treap has the **BST ordering property with respect to its keys,** **and the heap ordering property with respect to its priorities**.

### 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.

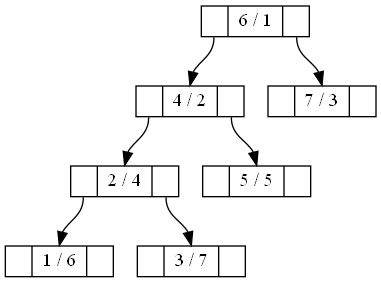
This completes the logic of the implementation of the insert function on treap data structure.

### Examples

**Example 1:**



Insert key 3 with priority 7.

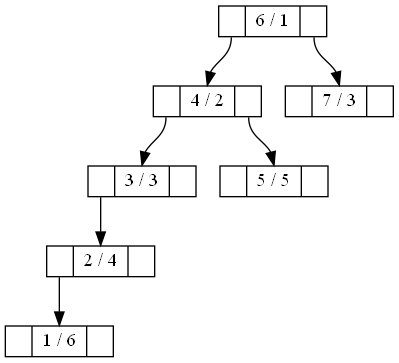


The node 3/7 is inserted as right child of 2/4 and there is no need of rotation.

**Example 2:**



Insert key 3 with priority 3.

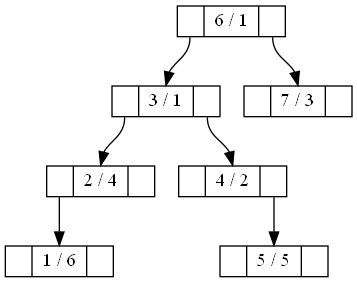


The node 3/3 is inserted as right child of 2/4 and there is a left rotation on node 2/4.

**Example 3:**



Insert key 3 with priority 1.

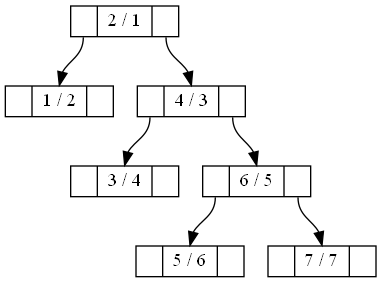


The node 3/1 is inserted as right child of 2/4 and there is a left rotation on node 2/4. Then there is a right rotation on the node 4/2. Thus we get the above treap.

**Example 4:**



Insert key 5 with priority 6.

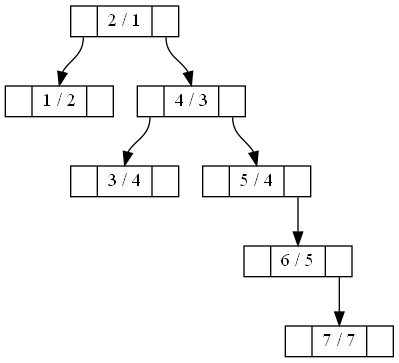


The node 5/6 is inserted as left child of 6/5 and there is no need of rotation.

**Example 5:**



Insert key 5 with priority 4.

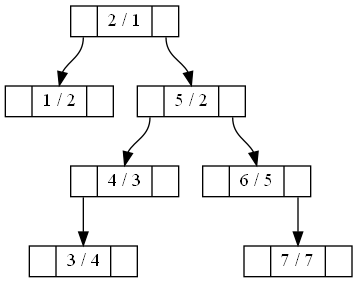


The node 5/4 is inserted as left child of 6/5 and there is a right rotation on node 6/5.

**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.

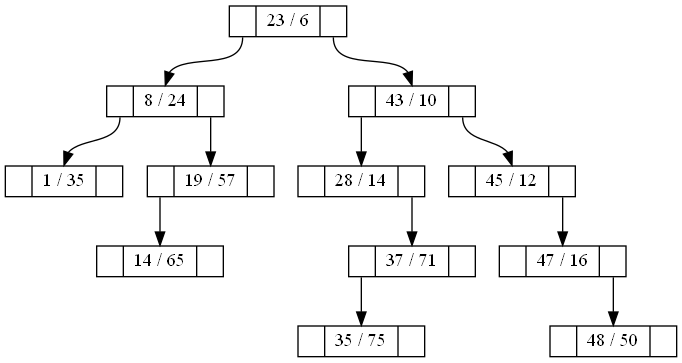
If k is less than root->key, we recur on the left child. Similarly if k is greater than root->key, we recur on the right child.

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

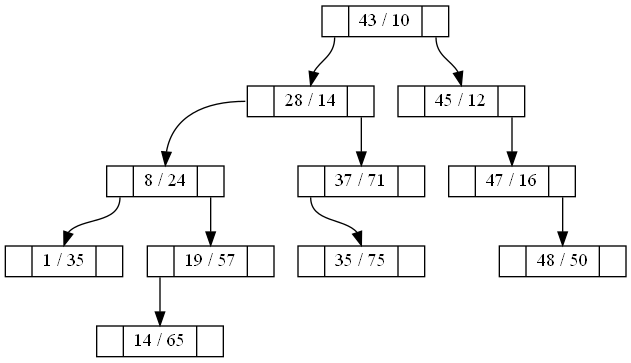
**handle\_priority\_downwards(root)**

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.

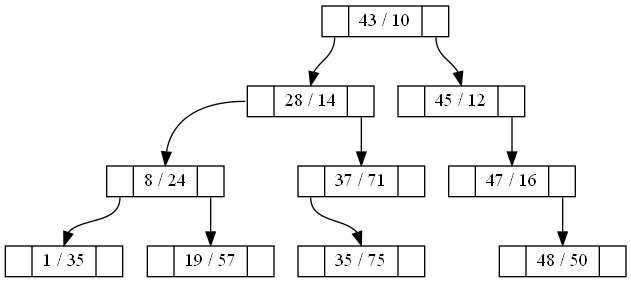
### Examples



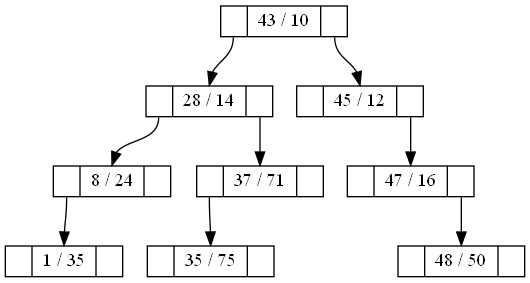
Delete 23 which is the root node.



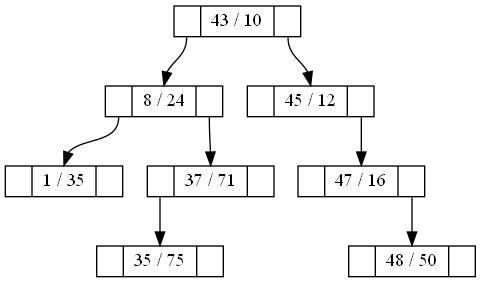
Delete 14 which is a leaf node:



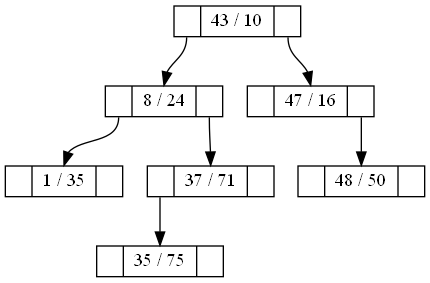
Delete 19 which is a leaf node:



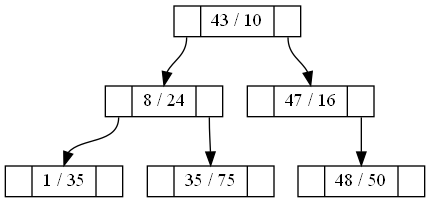
Delete 28 which is an intermediate node:



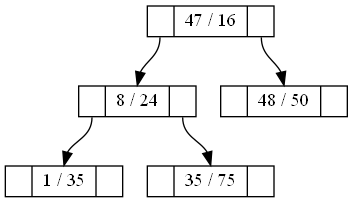
Delete 45 which is an intermediate node with right child only.



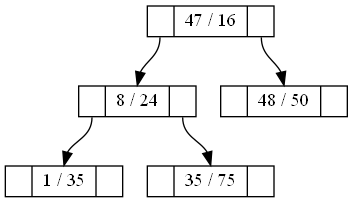
Delete 37, which is an intermediate node with left child only:



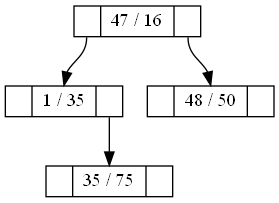
Delete 43:



Delete 24, an element which is not present:



Delete 8:



# Generation of Test Cases

In this assignment, we have to compare performance of the three types of data structures under a large number of insert and delete operations arranged in an interleaved manner. We create a function to produce a file as output which will contain a random number of insert and delete operations such as this:

10

Insert 6

Delete 6

Delete 6

Insert 4

Insert 1

Insert 7

Insert 2

Insert 0

Delete 2

Insert 2

To do this, we make use of the following function present in main.cpp:

**void generate\_test\_case()** and

**void generate\_test\_case(int ratio)**

If we call the first function, it calls the second function with an argument of 5. The function parameter ratio determines the approximate ratio of **insert : delete** operations which is determined as ***ratio : (10-ratio)*** where **ratio** is the function parameter. For example if the value of **ratio** is 7, it implies the number of insert and delete operations will be in the ratio 7:3.

There is a global variable called **NO\_OF\_OPERATIONS** which determines the total number of insert and delete operations that will be printed to the test case file. In the beginning we write the total number of operations in the first line of the test\_case file.

For each operation, we have to determine the kind of operation and then the element. We fix the first operation to be an insert. For rest of the operations we randomly select either insert or delete. To make this choice, we choose a random number between 0 and 9 inclusive and save it in a variable called **operation**. If the value of operation is less than **ratio** parameter passed in the argument of the function, we choose the operation to be **insert**. Otherwise, it is a **delete** operation.

If this function is called without any argument, the first version of the function will be called which calls the second version with an argument of 5. This implies that the number of insert and delete operations are equally likely to occur.

Suppose we got an insert operation. The next task is to generate another random number whose value lies between 0 and **NO\_OF\_OPERATIONS.** This is the element which needs to be inserted and hence ***Insert <element>*** is printed in a new line of the file. In addition to this, we add this newly inserted element to an array of elements called **insert[]**. This array maintains the list of elements inserted till now. This array will help to choose elements that need to be deleted when a delete operation is chosen.

If instead, we got a delete operation, we randomly choose the elements from the list of already inserted elements which is maintained in **insert[]** array. Then we simply print a new line in the test case file containing ***Delete <element>***.

Same procedure is followed for all the iterations to generate all the operations. At the end what we get is a test case file containing **NO\_OF\_OPERATIONS** number of operations.

Our next task will be to read this test case file and load the elements on to a tree data structure. For this we use the following functions:

**int take\_input\_from\_file(treap \*treap\_obj)**

**int take\_input\_from\_file(AVL\_Tree \*avl\_tree\_object)**

**int take\_input\_from\_file(TreeAPI \*bst\_object)**

The purpose of these functions is the same which is to read the file and load the elements into the objects they receive in argument. For three different kinds of data structures, we have three overloaded functions.

This function opens the test case file and reads it line by line. The first line contains the number of operations and the subsequent lines contain the kind of operation followed by the element. If we encounter an insert statement, then the element is inserted into the tree. If we encounter a delete statement, the element is deleted from the tree.

# Comparison of Tree Data Structures

In this section, we compare the performance of three kinds of data structures (Treap, BST and AVL Tree) on the basis of four parameters:

1. Total Height of the tree
2. Average height of each node of the tree.
3. Number of comparisons made in insert and delete operations
4. Number of rotations made in insert and delete operations.

To make the test cases diverse, we follow a particular approach.

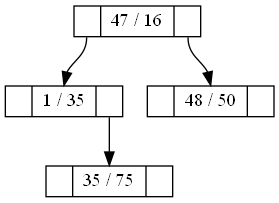
Let us define a test case by the ratio of number of insert operations to the number of delete operations. Thus Test Case 40:60 means the ratio of number of insert operations to number of delete operations is 40:60.

In a particular test case, defined as above, we generate a lot of sub test cases each containing different number of operations in an increasing order. Thus for each test case defined above, we will create sub cases having the following number of operations: 10,50,100,300,500,1000,1500,2000,2500,3000,3500,4000,4500,5000,5500,6000,6500,7000,7500,8000,8500,9000,9500,10000

Consider such a sub case have X number of operations. We will extract the data of four parameters mentioned above (on each such sub case) to compare the performance of a data structure with respect to others.

We plot the data for all the test cases which were defined on the basis of ratios.

## Total Height of the Tree



Height 1

Height 2

Height 3

The height of a leaf node is considered to be 1, as we move upwards, the height increases till root as can be seen above in the diagram.

In order to calculate of BST or AVL tree, we write a function which recursively finds out height of the left sub tree and right sub tree. The height of the current node is given by:

**height of a node = 1 + max (height of left sub tree, height of right sub tree)**

The total height of the tree is then given by height of the root node.

In case of Treap, height of each node is maintained in the node itself and it gets updated if required in insertion or deletion function itself.

**Conclusion**:

1. It can be seen that height of AVL data structure is less than that of BST and Treap. Height of BST and Treap is nearly same.
2. The height of AVL Tree does not exceed 15. And the curve is smooth.
3. The height of Treap or BST does not exceed 35. The curve fluctuates a lot for both of these data structures.

**Variation of Height with Number of Nodes**: It can be seen that the total height of any of the tree structures varies logarithmically with respect to number of nodes present in the tree.

## Average Height of each node of the tree

It is calculated as the sum of heights of each node divided by total number of nodes. While calculation of height of the tree, we had to calculate the height of each node and we had stored this height inside the node itself. So to find the sum of the heights of all the nodes, we just do a traversal of the tree and add all the heights. Similarly we can find out the number of nodes present in the tree by a traversal. Thus average height is obtained using the formula:

**Conclusion**:

1. For any of the data structures under consideration, the curve of average height becomes flat for large number of operations (which includes a random combination of insert and delete).
2. The average height of a node of AVL tree is less as compared to BST or Treap.
3. The average height of BST and Treap is almost similar with Treap being slightly on the higher side usually.
4. The average height of AVL tree does not exceed 2.5.
5. The average height of BST and Treap does not exceed 4.
6. The curve of AVL tree is smoother as compared to those of BST or Treap where the curve fluctuates.

**Variation of Average Height of each node with respect to number of nodes**: It can be seen that the average height of each node is nearly constant when a large number of nodes are present in the tree.

Average of all the average heights in case of BST= 3.44

Average of all the average heights in case of AVL Tree= 2.31

Average of all the average heights in case of Treap = 3.45

## Number of Comparisons in Insert and Delete

Each node visit contributes a value of one when we compute number of comparisons in case of insert or delete. So we maintain a class member variable called no\_of\_comparisions which keeps track of the number of comparisons done till that point of time. So whenever a node is visited during insert or delete, the variable gets incremented. At the end of all the operations, we obtain the value of the variable to plot the graphs as seen below.

**Conclusion**:

1. The number of comparisons in case of AVL tree lies in between those of Treap and BST with Treap having the highest number of comparisons amongst the three.
2. There is an interesting trend which can be seen from the graphs below. The gap between the three curves decreases when the ratio of Insert:Delete increases. This means that if the number of insert operations is higher, the number of comparisons become similar for all the three data structures. More appropriately we can see that curve of BST and AVL tree get closer.
3. The curve of AVL tree is smoother as compared those of Treap and BST which have fluctuating curves.

**Variation of number of comparisons with number of nodes:** It can be seen that when the number of nodes are very high, the number of comparisons varies linearly with the number of nodes present in the data structure. All the three curves fit very closely with linear trendlines.

## Number of Rotations in Insert and Delete

Rotations take place inside only in case of AVL Tree and Treap. We maintain a class member variable called **no\_of\_rotations** which get incremented once in case of a single rotation and twice in case of double rotation.

**Conclusion**:

It can be seen that number of rotations in case of Treap is much higher as compared to that of AVL Tree.

**Variation of number of rotations with number of nodes**: It can be seen that the number of rotations varies linearly with number of nodes present in the tree for AVL Tree and Treap. The curves fit very well with linear trendlines.