Abhijeet Padhy

Roll: 214101001

DS LAB Assignment 3

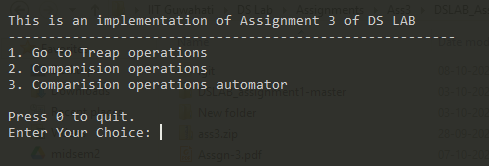
# INDEX

1. [User Manual : How to use the program](#_User_Manual)
2. [Treap Data Structure Implementation](#_Treap_(Random_Search)
   1. [Introduction](#_Introduction)
   2. [Insert](#_INSERT)
   3. [Delete](#_DELETE)
   4. [Search Key](#_SEARCH_KEY)
   5. [Print Tree](#_PRINT_TREE)
   6. [Destructor](#_Destructor)
3. [Generate Test Case and read from file](#_Generation_of_Test)
4. [Performance Comparison of Various Data Structures](#_Comparison_of_Tree)
   1. [Total Height of the Tree](#_Total_Height_of)
   2. [Average Height of each node of the tree](#_Average_Height_of)
   3. [Number of comparisons in insert and delete](#_Number_of_Comparisons)
   4. [Number of Rotations in insert and delete](#_Number_of_Rotations)
5. [Conclusion](#_CONCLUSION:_(Comparison_with)

# User Manual

The single program contains the implementation of Treap Data Structure as well as the program for generating test cases and then loading them into different data structures for comparision of performance parameters.

The beginning menu looks like this:



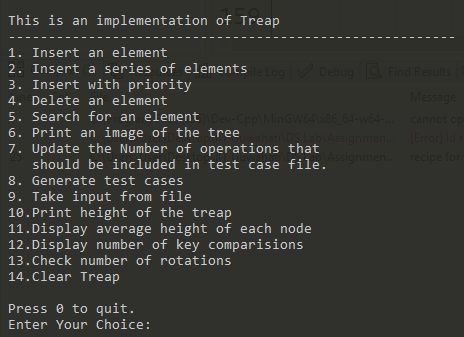
*Fig 1: Main Menu*

Option 1 is for checking treap implementation.

Option 2 is for generating test cases.

Option 3 is an automated set of codes which generate many different test cases automatically, loads them into the different kinds of tree data structures and then displays the performance parameters on the screen. Those data are copied into excel sheet for creating the plots used in this documentation.

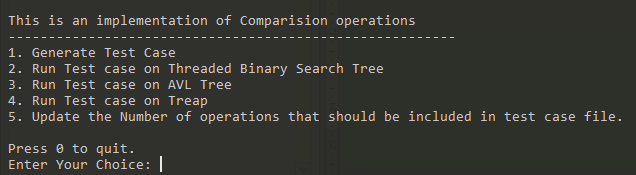
The menu for Option 1 looks like this:



*Fig 2: Menu for Treap Data Structure*

Options 1 to 6 are self-explanatory. Option 7 is used to update the number of operations that should be printed into the test case file. By default a test case file of 10000 operations is created. Option 8 is used to create a test case file. Option 9 reads the test case file and performs the required operations on the treap data structure. Options 10-13 are used for displaying the performance parameters on the screen. Option 14 is used to clear the treap data structure so as to make it ready for fresh insertions.

Had we used option 2 in the main menu in Fig 1, we would have got such a menu which is used to create test cases and load them into the three type of data structures namely Threaded BST, AVL Tree, Treap.



*Fig 3: Menu for generating test cases and loading the test case file to perform various operations on any of the data structres.*

By default the number of operations is set at 10000. We can change it via option 5. We can use option 1 to generate a test case file. Then we can use options 2, 3 or 4 to load the test case file and perform operations on TBST, AVL Tree or Treap respectively. After we select option 2, 3, or 4, a diagram of the tree is also created in the form of graph.gv file which we have to convert to png using the following command: ***dot -Tpng graph.gv -o graph.png.***

# Treap (Random Search Tree)

## Introduction

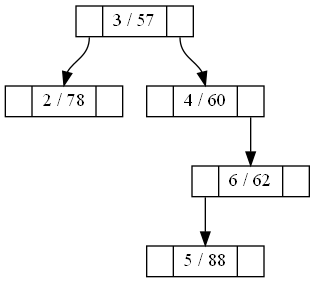
A treap is a binary tree whose nodes contain both **a key and a priority**. A treap has the **BST ordering property with respect to its keys,** **and the heap ordering property with respect to its priorities**.

### 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 throw an exception that element already exists.

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.

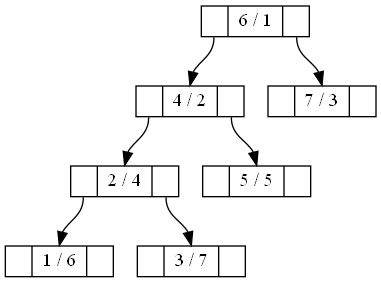
This completes the logic of the implementation of the insert function on treap data structure.

### Examples

**Example 1:**



Insert key 3 with priority 7.

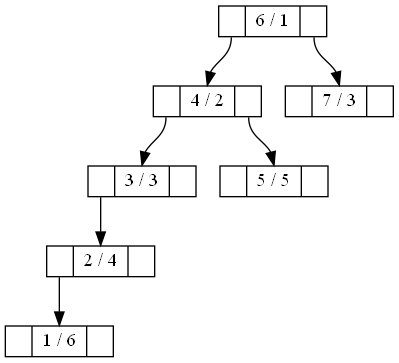


The node 3/7 is inserted as right child of 2/4 and there is no need of rotation.

**Example 2:**



Insert key 3 with priority 3.

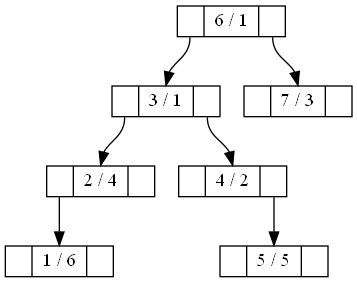


The node 3/3 is inserted as right child of 2/4 and there is a left rotation on node 2/4.

**Example 3:**



Insert key 3 with priority 1.

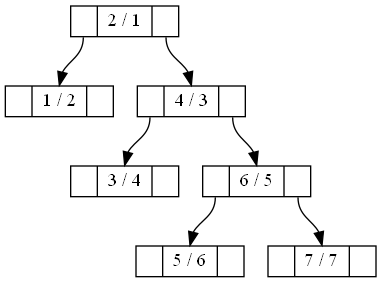


The node 3/1 is inserted as right child of 2/4 and there is a left rotation on node 2/4. Then there is a right rotation on the node 4/2. Thus we get the above treap.

**Example 4:**



Insert key 5 with priority 6.

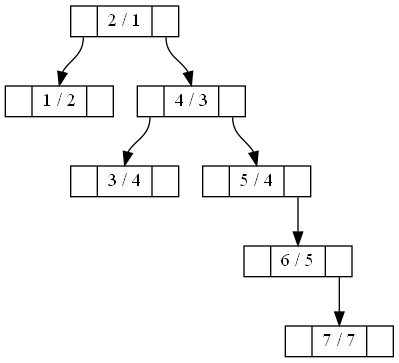


The node 5/6 is inserted as left child of 6/5 and there is no need of rotation.

**Example 5:**



Insert key 5 with priority 4.

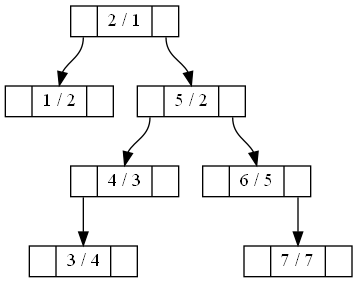


The node 5/4 is inserted as left child of 6/5 and there is a right rotation on node 6/5.

**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 throw an exception that element is not found. This happens when the element is not present in the treap.

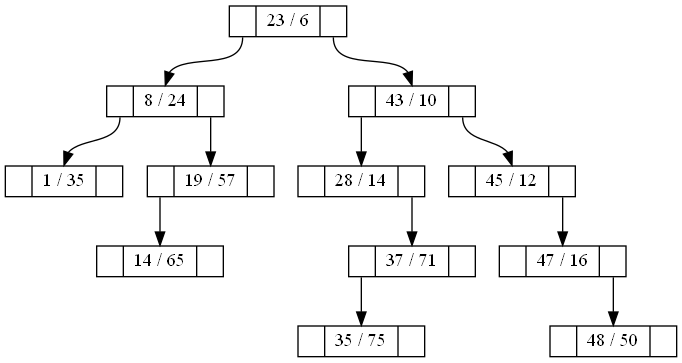
If k is less than root->key, we recur on the left child. Similarly if k is greater than root->key, we recur on the right child.

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

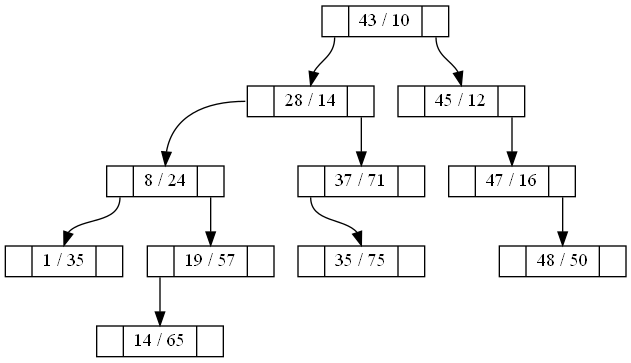
**handle\_priority\_downwards(root)**

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.

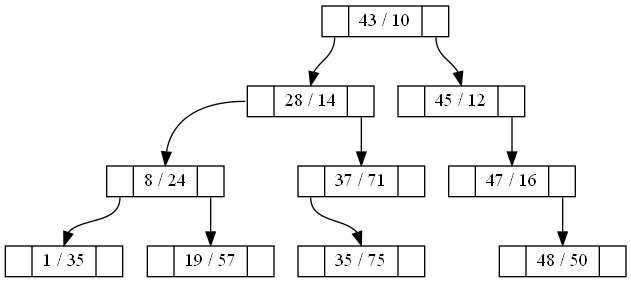
### Examples



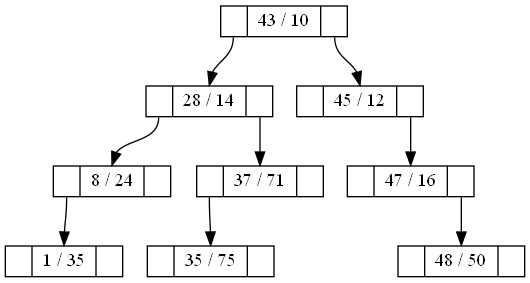
Delete 23 which is the root node.



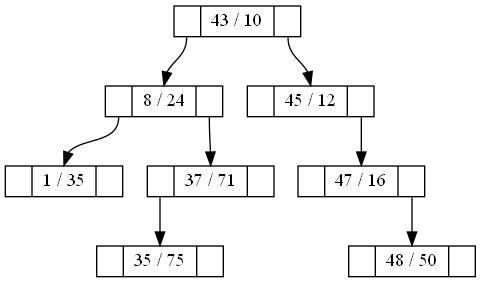
Delete 14 which is a leaf node:



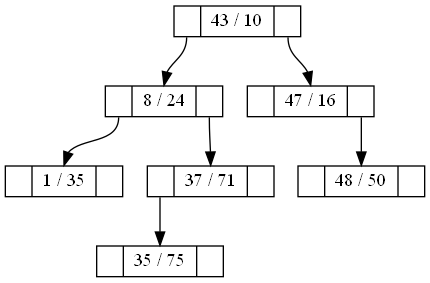
Delete 19 which is a leaf node:



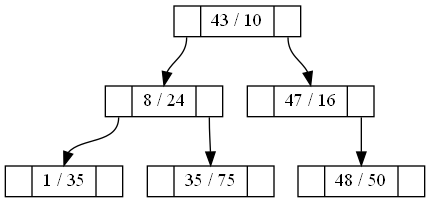
Delete 28 which is an intermediate node:



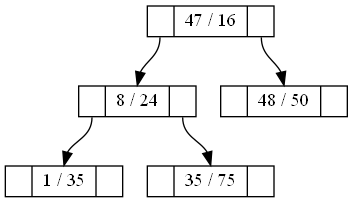
Delete 45 which is an intermediate node with right child only.



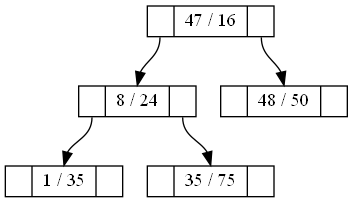
Delete 37, which is an intermediate node with left child only:



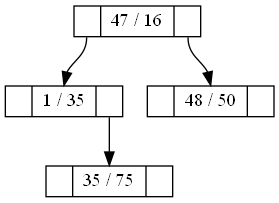
Delete 43:



Delete 24, an element which is not present:



Delete 8:



## SEARCH KEY

Searching is very simple. We search for the element recursively. If element value is equal to the current node’s values, return true as the element is found. If element’s value is less than node’s key value, search recursively in left sub tree. Otherwise search in right sub tree. If node is NULL, return false. So if we cannot find the element, we will ultimately reach NULL and hence return false.

## PRINT TREE

The purpose of this function is to generate .gv file which will be parsed by graphwiz utility to  
produce an image of the tree. We traverse the tree in a preorder manner. When we arrive at a node  
in our traversal, we add node in the .gv file. Then we recur for left and right subtrees. When we  
return from recursive call from a sub tree, we add an edge from the root node to the left or right sub  
trees’ root node. At the end we have the complete .gv file. From outside of the program, run the  
following command to convert a .gv file to .png file which is our required image.

dot -Tpng graph.gv -o graph.png

## Destructor

This is called when we execute delete(ptr) function. This helps to free all the dynamically allocated memory during program runtime. The destructor calls the clear\_tree function whose work is to recursively delete all the tree nodes. At the end, the tree object is also deleted.

If some allocated memory is not freed after program exit, there will be memory leak which will be detected using valgrind.

# 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.

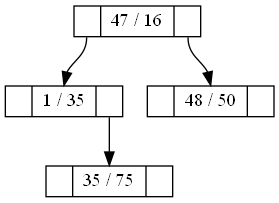
Let us define a test case by the ratio of number of insert operations to the number of delete operations. Thus Test Case 40:60 means the ratio of number of insert operations to number of delete operations is 40:60.

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 tree is the length of the longest path. In this assignment, 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 height 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.

The plot of Height of a data structure is plotted against number of operations for various test cases. Each test case has a definite ration of number of insert operations to the number of delete operations as mentioned in the heading of the plot.

**Observation**:

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 20 for 10000 operations and the curve is smooth.
3. The height of Treap or BST does not exceed 35 for 10000 operations. The curve fluctuates a lot for both of these data structures.
4. But for further insertions, the height will slowly increase. Actually the height of the trees varies logarithmically with number of nodes present in the tree. Log(n) is a very slow growing function and thus the height increases very slowly for large number of nodes.

**Theorem**: The expected height of a treap storing n nodes is O (log n).

To verify this theorem with practical data, we plot the Total Height of Treap data structure with respect to number of nodes.

**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. The data fits very well with a trendline having logarithmic equation.

## 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:

The plot of Average Height of each node of the three data structures is plotted against number of operations for various test cases. Each test case has a definite ratio of number of insert operations to the number of delete operations as mentioned in the heading of the plot.

**Observations**:

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 for 10000 operations.
5. The average height of BST and Treap does not exceed 4 for 10000 operations.
6. The curve of AVL tree is smoother as compared to those of BST or Treap where the curve fluctuates.

## 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.

The plot of number of comparisons made during insert and delete operations of the three data structures is plotted against number of operations for various test cases. Each test case has a definite ratio of number of insert operations to the number of delete operations as mentioned in the heading of the plot.

**Observations**:

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 above. 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. In fact in the test case of 90:10, it can be seen that number of comparisons in BST exceeds that of AVL Tree.
3. The curve of AVL tree is smoother as compared to those of Treap and BST which have fluctuating curves. A possible reason for this could be that AVL Tree is height balanced.

## 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.

The plot of number of rotations made during insert and delete operations of AVL Tree and Treap data structures is plotted against number of operations for various test cases. Each test case has a definite ratio of number of insert operations to the number of delete operations as mentioned in the heading of the plot.

**Observations**:

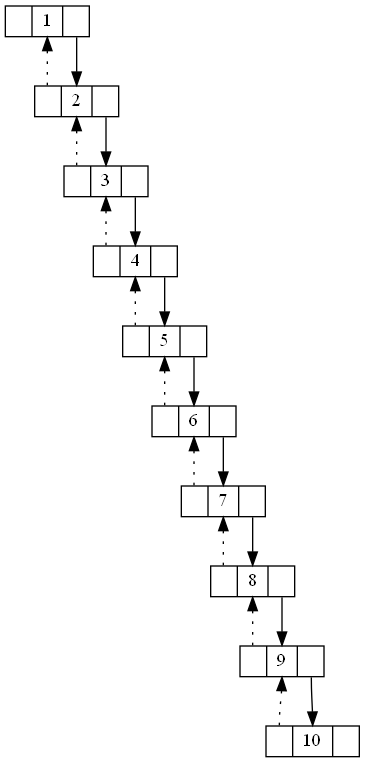
1. It can be seen that number of rotations in case of Treap is much higher as compared to that of AVL Tree.
2. There is no major change in pattern of plots for various test cases. This implies the nature of the plot does not depend on the ratio of number of insertions and deletions.

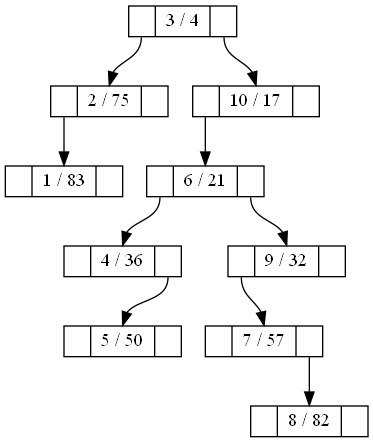
# CONCLUSION: (Comparison with Theoretical and Experimental results)

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability.

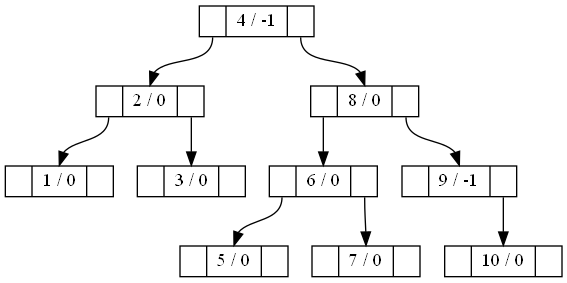
* On checking the plot of Total height with respect to number of nodes in AVL Tree, we can find that, the curve matches very closely to that of log (n). But in case of BST or Treap, the curves are very fluctuating, although they still match log (n) trendline.
* The Total Height of an AVL Tree is minimum and heights of BST and Treap are nearly same.
* The average height of each node is less in case of AVL Tree as compared to those in BST or Treap which have nearly same values for average height.
* The number of comparisons in case of all of the three data structures is nearly same.
* A BST does not undergo any kind of rotation. The number of rotations in case of a Treap is much higher as compared to that of AVL Tree. A possible reason for this could be that in almost all insertions and deletions, a Treap has to go through multiple rotations so as to satisfy the heap property. But rotations in case of an AVL Tree occur only when the tree is no more balanced and does not occur in all insertions or deletions.
* In worst case the height of A BST can go up to O (n) when the tree is skewed. But a treap will have less height as compared to a BST in such a case because the random priorities which we use to maintain heap property help to balance the height to some extent.

Suppose we insert the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 in this sequence, let us examine the behaviour of all the three kinds of trees.

**Threaded Binary Search Tree: Treap:**



**AVL Tree:**

****

This is one such case where the performance of BST fails miserably because of skewed nature of the tree. Earlier we got nearly log (n) height because the elements were inserted in random order.

|  |  |  |  |
| --- | --- | --- | --- |
|  | BST | AVL | Treap |
| Total Height | 10 | 4 | 6 |
| Average Height | 5.5 | 1.8 | 2.7 |
| Comparisons | 45 | 36 | 20 |
| Rotations |  | 6 | 8 |

It can be seen that a treap performs better than BST in such a case. But a treap is not as good as an AVL Tree in terms of height or average height.

In most of the cases, we can see that AVL Tree performs better than Treap. Still Treaps have some advantages which are as follows:

**Treaps are simple to implement. The join and split operations can also be implemented easily. Although AVL Trees usually provide better performance in terms of number of rotations, comparisons, and height, they are difficult to implement. Even the basic insert and delete operations are difficult to implement in case of AVL Trees as compared to that in Treaps**