VANSHITA BANSAL
214101056
DS LAB Assignment 3

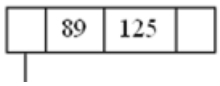
INDEX

Section 1: Logic for Implementation of Treap

Introduction

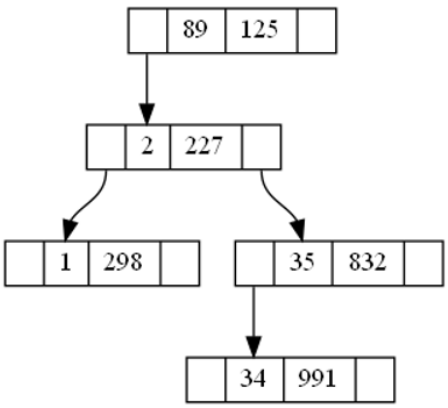
A treap is a binary tree that contains **key and priority pair**. It has the BST property w.r.t keys and Heap property w.r.t priority**.**

Structure of Treap Node



Where 89 is key and 125 is priority assigned randomly.

- Value -> Integer values stored in a node
- Priority -> Random priority given to each node
- Left pointer-> stores pointer to left child of node
- Right pointer-> stores pointer to right child of node
- Height-> stores height of each node



It can be seen in the above tree that it follows the **Binary Search Tree property** i.e, the keys in the left subtree of a node is less than its key and that in the right subtree have greater key. Also, if notice that **Heap property** is maintained in this tree where priority values follow min heap property. The convention here used is that priority values 1 means higher priority than priority value 2 (priority 1>2). Example, 89 has priority values 125 which is highest among all nodes and so it is the root of tree.

User interface: To check implementation enter choice 1.

```
This is DS Assignment number 3 (214101056)
-----------------------------------------------------------
Please choose among the following operation
1: Go to implementation of Treap data structure.
2: Do comparison between RST, BST and AVL Tree.
3: Get Automaticaly generated comparison table of three.
-----------------------------------------------------------
Enter your choice of function:
```

Menu for Implementation of Treap:

```
This is implementation of Treap Data Structure:
-----------------------------------------------------
The choices for various functions are listed below:
-------------------------------------------------
Choices:
0: Quit this menu
1: Insert
2: Search
3: Delete
4: PrintTree
5: Get a Created tree and print it
6: Create a copy of sample tree and print it
7: Insert item with priority

Enter your choice of function:
```

Menu for doing comparison:

```
--------------------Perform Operations-----------------------------
1: Generate test case file (To use your own testcase file simply copy and paste the test case
values into TestCase.txt file and perform any of the below operation)
Also, make sure that the test case file follows the same convention as this assignment has.
2: Run operations on BST
3: Run operations on AVL Tree
4: Run operations on Treaps(RST)
5: Quit this menu
```

Insert operation

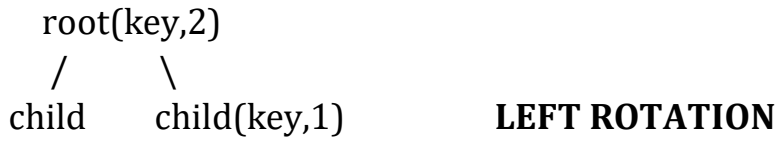
- Inserting node (key, priority) using simple BST insert:

The idea is we recursively search for position where the node will be inserted. During insertion we create a new node containing item and randomly generated priority and add it to the position. The random priority is generated and can also be user defined depending on the choice of user.

- Then we perform rotation, either left rotation or right rotation until the priority of its parent is greater than or equal to node's priority or the node becomes the root.

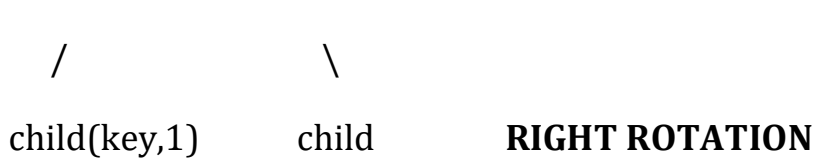
- Left rotation is done when node is added to right child and priority of parent is greater than priority of node.

Performing left rotation



- Right rotation is performed when node is added to left child and priority of parent is less than node's priority.

Performing Right rotation



- Function used:

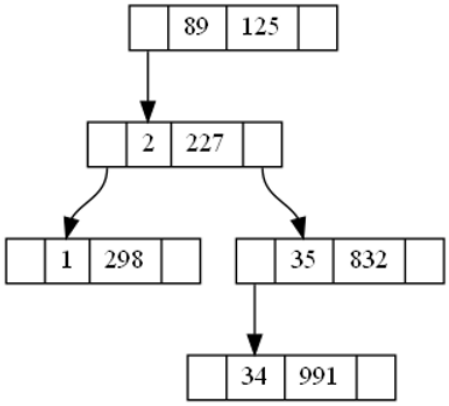
Insert(key): inserts item with key value and randomly generated priority.

Insert (key, priority): inserts item with given key and user defined priority.

TEST CASE 1: Initially tree is empty.

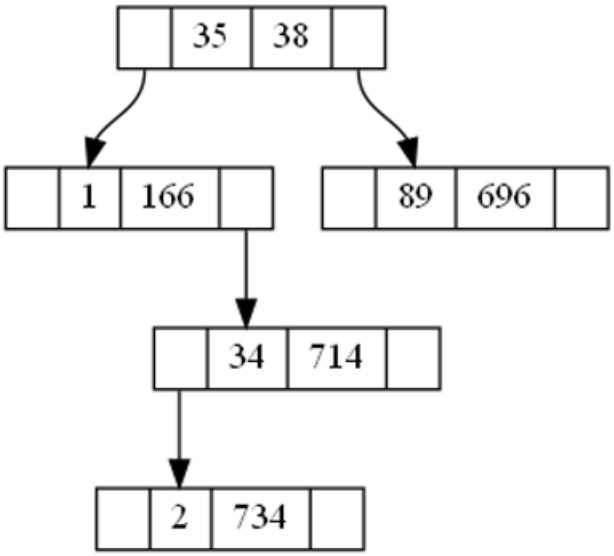
```
Enter your choice of function:
1
Enter the number of item to be inserted into the tree:
5
Enter the item(s) to be inserted:
34 1 2 89 35
34 inserted successfully!
1 inserted successfully!
2 inserted successfully!
89 inserted successfully!
35 inserted successfully!
------------------------------------------------
```

OUTPUT:



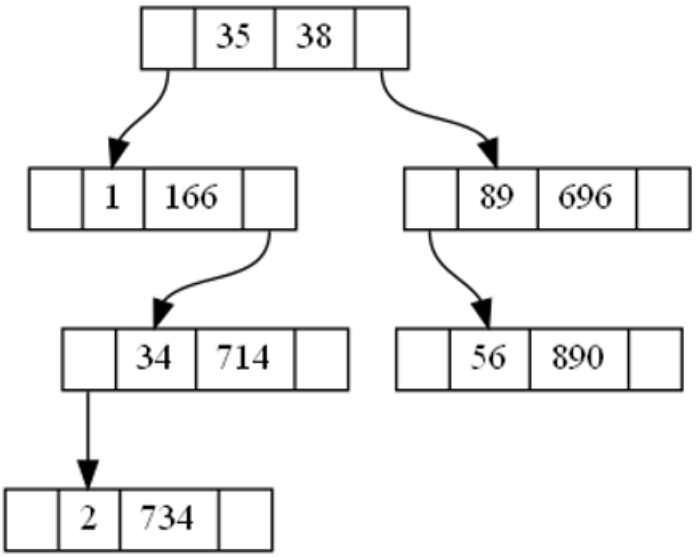
We can see that priority of 89 is highest hence it becomes root after rotations and similarly for all nodes heap property is maintained. All the keys follow BST property and priorities follow Heap property.

TEST CASE 2:



Inserting node with key 56 and priority 890

OUTPUT:



Delete operation

- Firstly, search the node containing item to be deleted recursively using BST search algorithm. If the tree does not contain the item, then we simply return.

- If the node to be deleted is leaf we simply remove it and return NULL.

- Otherwise, we perform AVL rotations to rotate the node down until it becomes a leaf and then delete it. We first assign the priority of node to be deleted as INT_MAX (infinity) and since our tree follows min heap property, we will fix priority using left and right rotations and move the node downwards till it becomes leaf and then remove it.

- If it has both the children, we rotate with child having smaller priority to fix priority and maintain heap property.

Logic to fix priority: We make the priority of node as infinity (INT_MAX). Now we will find the child with less priority value (as 1 means higher priority and 2 means less priority). If left child has higher priority, we will perform right rotation and call fix priority function on right child. Similarly, in we perform left rotation (right child has higher priority then left child) and call fix priority on left child this continues till node to be deleted becomes leaf. Finally, removing the leaf.

//Performing right rotation

node (key, infinity)

/ \

Leftchild (key,1) rightchild (key,2) RIGHT ROTATION

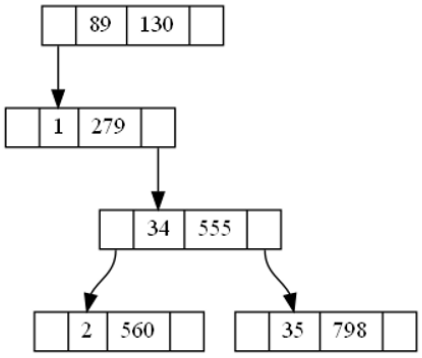
//Performing left rotation

Node (key, infinity)

/ \

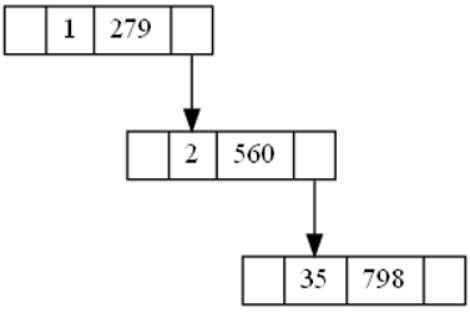
Leftchild (key,2) rightchild (key,1) LEFT ROTATION

TEST CASE 1:

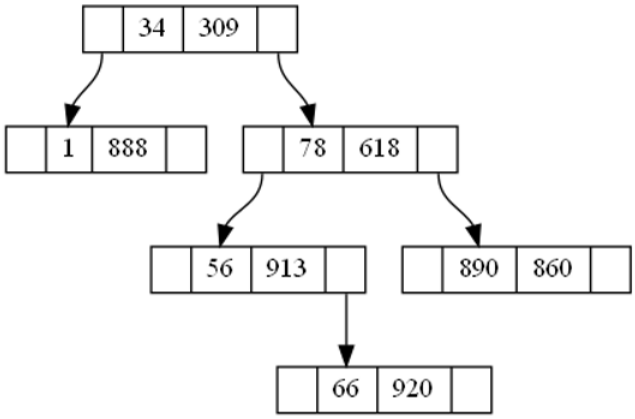



```
-------------------------------------------------
Enter your choice of function:
3
Enter number of item to be Deleted from the tree:
3
Enter item(s) to be Deleted from the tree:
89 56 34
89 Deleted successfully!
56 doesn't exist!
34 Deleted successfully!
-------------------------------------------------
```

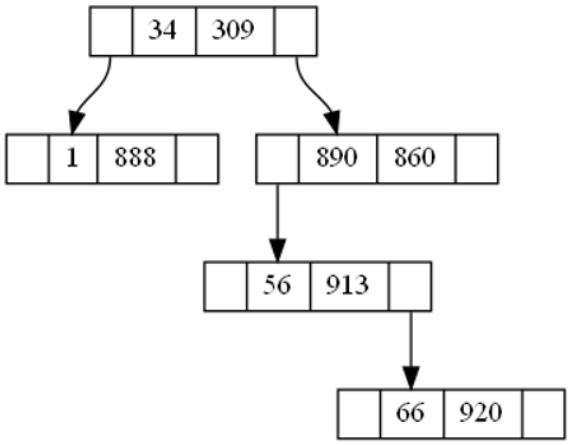
OUTPUT:



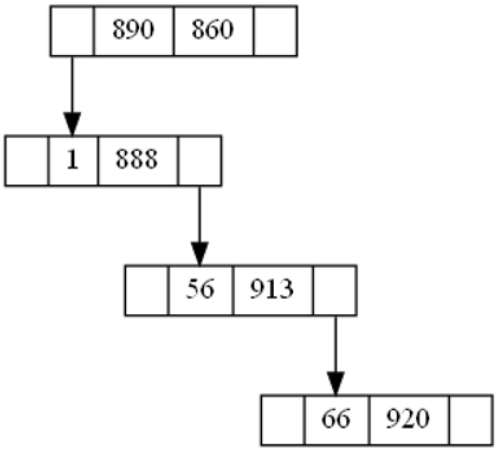
TEST CASE 2:



OUTPUT: Delete (56) Left Rotation case



OUTPUT: Delete (34) Root deletion case

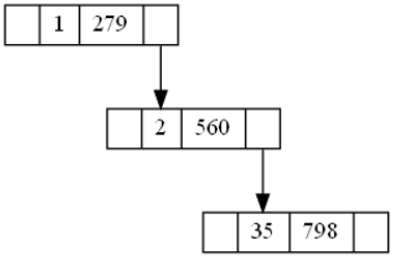


Search operation

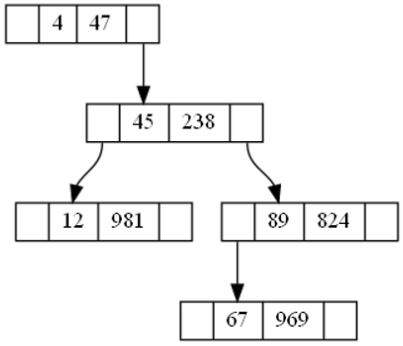
We have taken a pointer (named current) that is used to traverse the tree. Initially, it points to root of the tree. The algorithm is same as that of BST search.

- If current is empty then this means the tree does not contain the item and we will return FALSE.
- Otherwise, we will iteratively search for the item. If the current node's key is less than item then we will move to the right subtree. If the current node's key is greater than the item then we will move to the left subtree.
- Current node's key is equal to item then we have found the item and we will return TRUE.

TESTCASE 1:



TEST CASE 2:



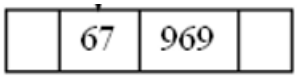
OUTPUT:

```
--------------------------------------------------
Enter your choice of function:
2
Enter number of item to be searched in the tree:
6
Enter item(s) to be searched in the tree:
1 23 45 89 34 500
The element is present in the Tree!
Not Found, element is not present in the tree!
Not Found, element is not present in the tree!
Not Found, element is not present in the tree!
Not Found, element is not present in the tree!
Not Found, element is not present in the tree!
--------------------------------------------------
```

OUTPUT:

```
--------------------------------------------------
Enter your choice of function:
2
Enter number of item to be searched in the tree:
5
Enter item(s) to be searched in the tree:
1 4 67 45 23
Not Found, element is not present in the tree!
The element is present in the Tree!
The element is present in the Tree!
The element is present in the Tree!
Not Found, element is not present in the tree!
--------------------------------------------------
```

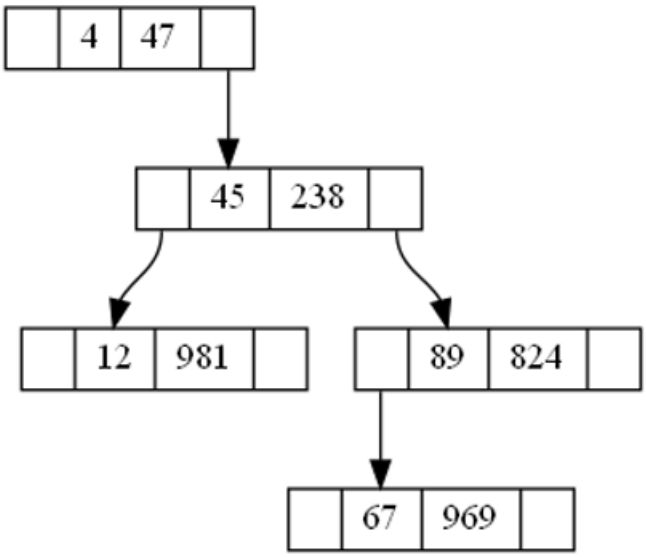
Print Operation



- The node structure is printed as where 67 is the key and 969 is the priority value.
- In this function we will create a **.dotgv** file that will contain the syntax to print the tree. This is done using recursion. By one by one traversing the nodes and writing the edges and also keeping in mind to check whether the child pointers are not NULL. Solid arrows denote pointer to child pointers.

- The command "**dot.exe -Tpng t.dotgv -o** "+ string(filename) uses t.dotgv file and outputs the Tree structure in filename. Example, "Output.png".

  Sample Output:



Section 2: Testcase Generation and Parameter calculation

We have Insert and Delete operation written in file in interleaved fashion.

The convention followed is as follows:

10000

Insert 8100

Insert 351

Insert 6830

Delete 351

Insert 1883

Delete 8100

Insert 3947

Insert 5695

Insert 4682

Delete 4942

Insert 2364

Delete 10

Insert 1179
…….

Where 10000 is the number of operations.

Testcase Generation

- Firstly, we open a text file ("TestCase.txt") in write mode. As we are going to write in file we will use "w".
- We will maintain an array that will keep track of the elements inserted in the tree. Also, a variable count for number of inserted elements that will be used in the deletion operation.
- The first operation is explicitly written as insert where a random number is generated using a rand () function.
- Each time a number is inserted in the tree we store it in the array and increment the count variable.
- The number of operations is taken from the user as input (e.g. 10000). We generate a random number for item to inserted and for each operation (taken modulus 2). Now, if the modulus is 0, we write INSERT (item) and if modulus is 1, we write DELETE (item).
- Now, for the item to be deleted we take random number modulus count/2 of inserted items. This is to reduce the probability of deletion of an item that doesn't exist in the tree.
- The division by 2 is done so that deletion operations does not empty the whole tree, i.e., some elements are left in the tree.

To use your own Test case file, you just need to copy and paste your testcase to the TestCase.txt file i.e., replace the contents of TestCase.txt with your contents. Now, you can run operations on AVL, BST and RST(Treap) by choosing from menu using option 2.

Parameters chosen

1. **Number of Key Comparisons**

   Calculation:
   For each node visit contributes we consider a comparison in case of insert or delete. We use a class member variable **total_keycomparison** 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 call the **get_total_comparison** function to obtain the value of the variable.

   Need:
   Since all the three tree has the property of BST I.e., all the nodes in left subtree have less value and those in the right subtree have greater values than root. So, if we need to search, delete and insert an element we need to check only some part of tree depending on key value. Hence, if the tree is skewed or has height of order of O(n) then these operation will take O(n) time in worst case. And, if the tree is balanced then these operation might take less time on average.

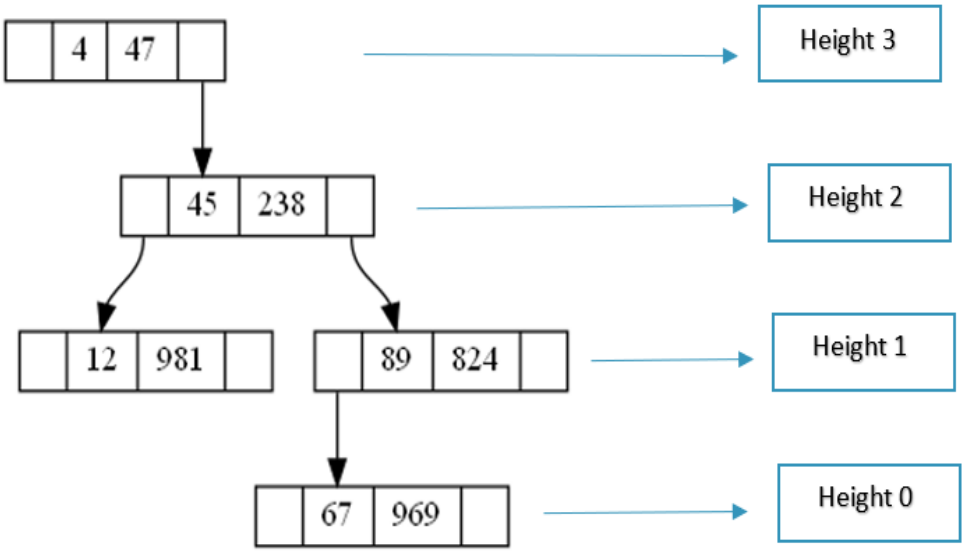
2. **Number of Rotations**

   Calculation:
   Whenever there is single rotation or double rotation in AVL insert or delete operation, left or right rotation in Treap insert or delete, we consider it as rotation. We do not have rotations in BST.

   We maintain a **total_rotations** variable that is updated for each rotation performed. It is incremented twice for a double rotation. We have a **get_total_rotation** function that return this variable finally.

Need:
To maintain the height balance property in case of AVL trees we perform single or double rotations if required. In case of Treaps we perform left or right rotations to maintain the heap property and in case of BST we do not perform any rotations. So, the number of rotations done in each case (type of tree) comparison gives us some intuition about the tradeoff between the height balance and cost of maintaining it.

3. **Final height of tree**



Calculation:

We have used a **Final_height** function that does this task for us, also notice that this function will also update all the height variables of each node in one traversal O(n) which will be used to find average height:

- If a node is empty or it is leaf node, we return height as 0.

```
Node->h=0
```

- If a node has only left child, then we return

  Node->h=1+height (left subtree)

- If a node has only right child, then we return

  Node->h=1+height (right subtree)

- If a node has both the children, we will return maximum of the height of two subtrees

  Node->h=1+max (height (left subtree), height (right subtree))

Need:
The reason is same as that of number of key comparisons. The greater the height of tree the more it takes to perform operations. As we need to search the element until we get it to either perform insert or delete and this searching requires tree traversal.
Also, we see that number of nodes say n, leads to expected height of order of O (log n).

4. **Average Height of nodes**

Calculation:
Since, in the final height function we have updated all the node heights in variable. The Average height function will simply traverse all the nodes and add the height of all nodes using node->h field. At last, we will return this sum divided by total number of nodes. This gives us the average height of each node.

Section 3: Analysis of BST, AVL and RST (Randomized search tree)

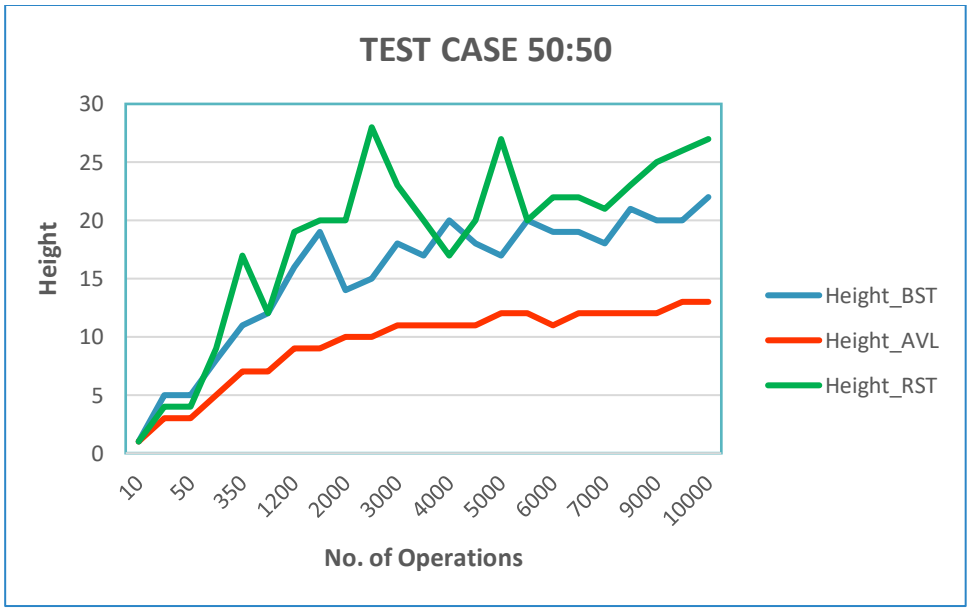
Parameter list used:

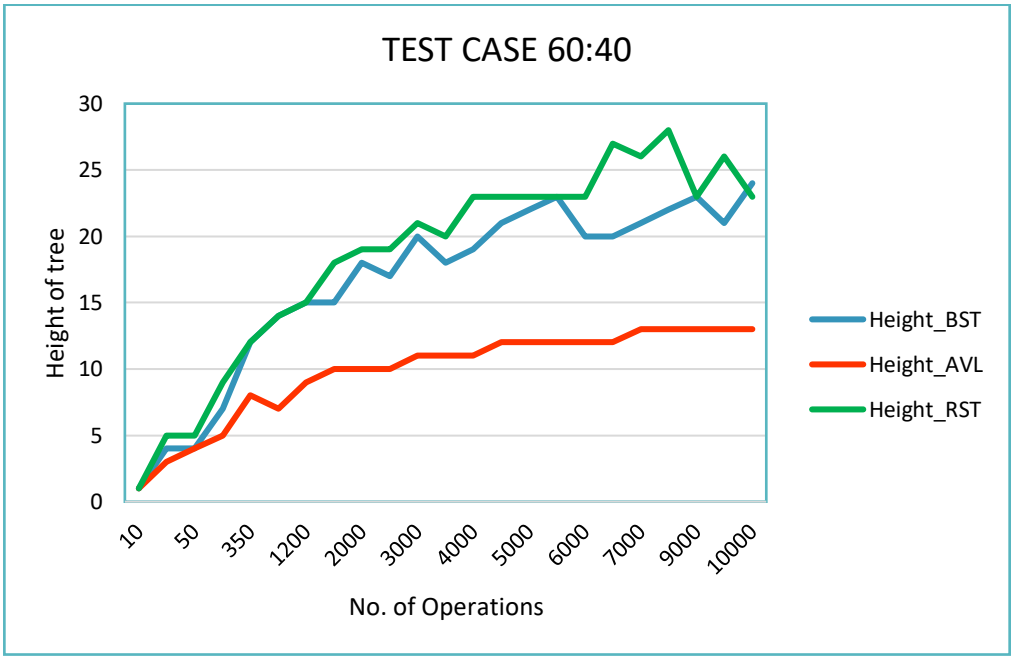
1. Height of tree
2. Average height of node of tree
3. Number of comparisons
4. Number of Rotations

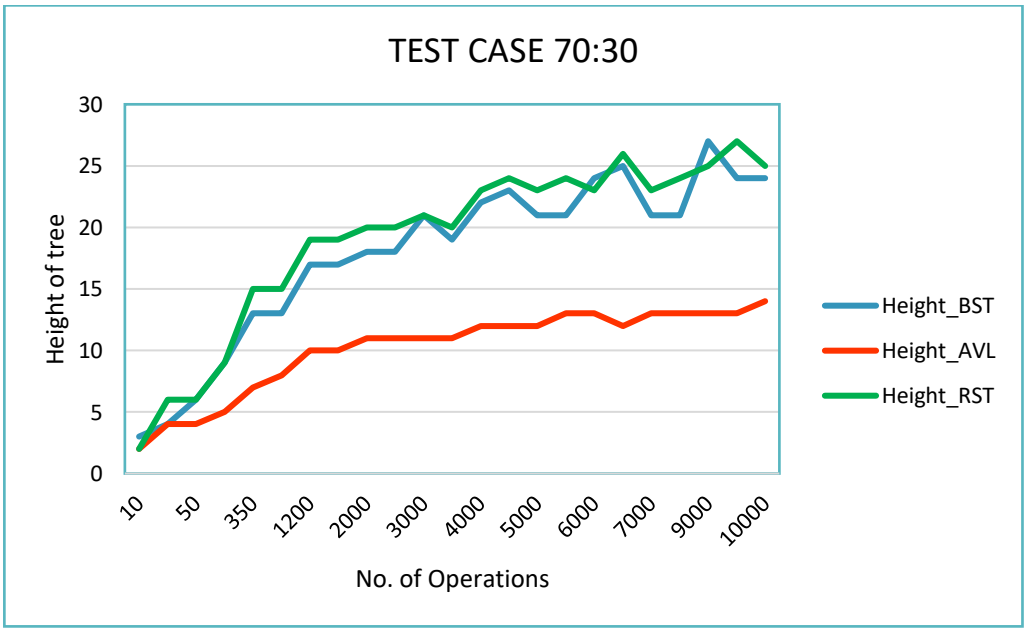
1. Height of tree

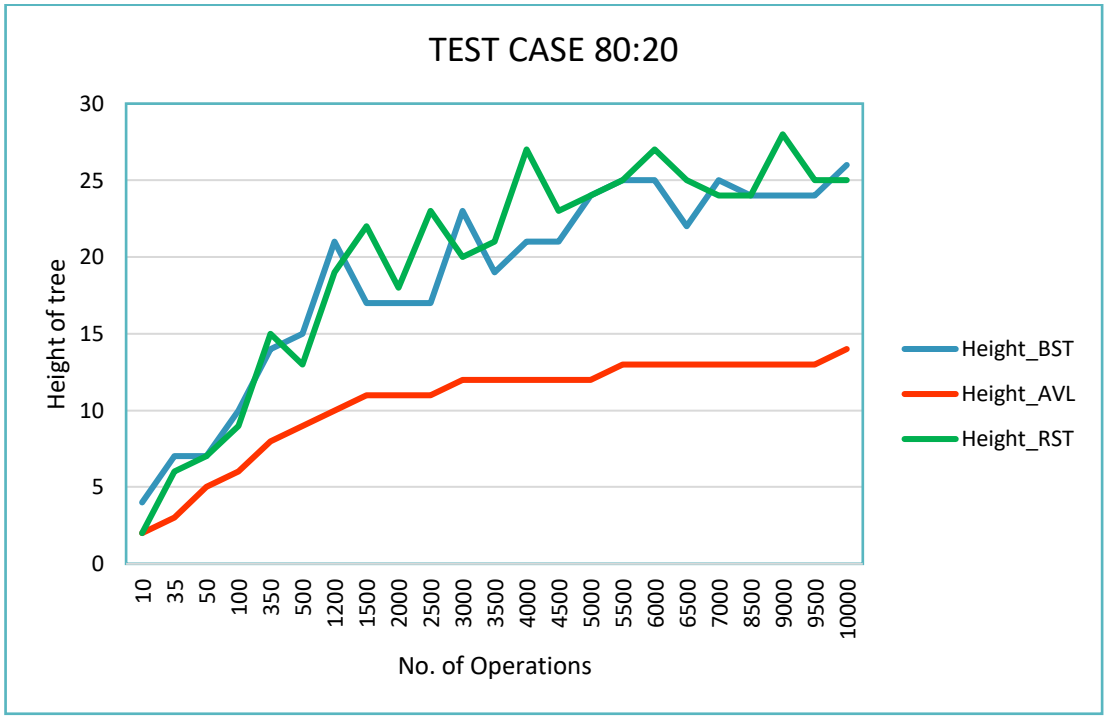
The height of tree is defined as the root to leaf path in other words height of the tree is equal to the **largest number of edged from root to the most distant leaf**.

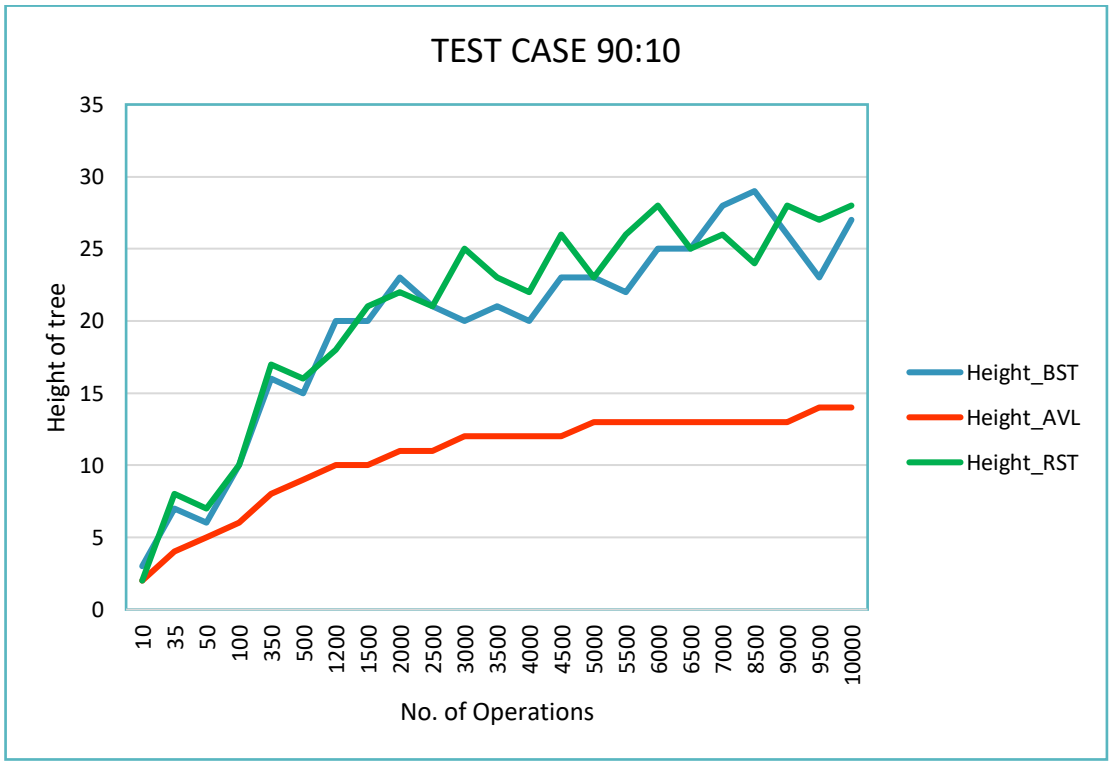
The graph with various test case types is plotted with height on y axis and Number of operations (insert + delete) on x axis.




TEST CASE 60:40
Height of tree
30
25
20
15
10
5
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
Height_BST
Height_AVL
Height_RST


TEST CASE 70:30
Height of tree
30
25
20
15
10
5
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
Height_BST
Height_AVL
Height_RST


TEST CASE 80:20
Height of tree
30
25
20
15
10
5
0
10
35
50
100
350
500
1200
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
8500
9000
9500
10000
No. of Operations
Height_BST
Height_AVL
Height_RST


TEST CASE 90:10
Height of tree
35
30
25
20
15
10
5
0
10
35
50
100
350
500
1200
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
8500
9000
9500
10000
No. of Operations
Height_BST
Height_AVL
Height_RST

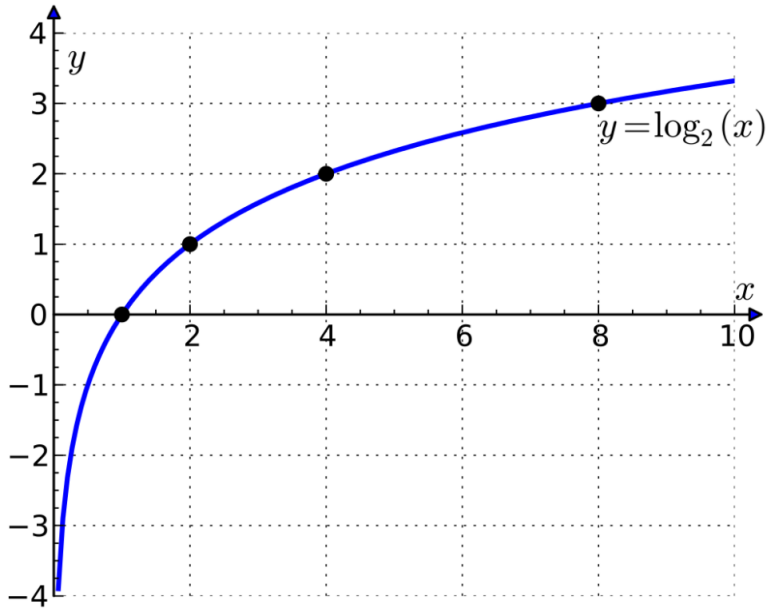
Observation:

- From the above graphs we observe that Height of AVL tree is least among the three in all cases. Whereas the height of BST and Treap is almost same.
- The height of Treap or BST does not exceed 30 (which satisfies the theoretical concept which states that:
  "**Height of treap does not exceed 100.**"
- The curve for AVL tree is smoother than that of Treap and BST that have high variance.

Theorem: A treap storing n nodes has height O (log n) in expectation (over all n! possible orderings of the random priorities present in the tree).

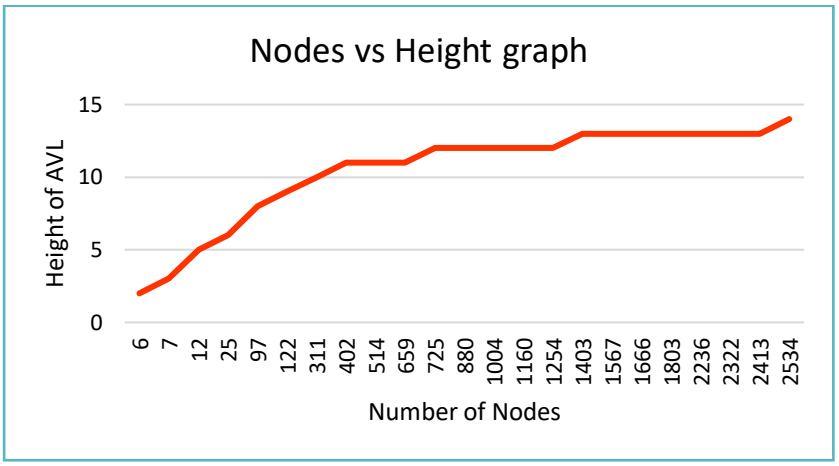
To prove this theorem practically we will use Number of nodes versus Height of Tree graph.

Graph for log(n)

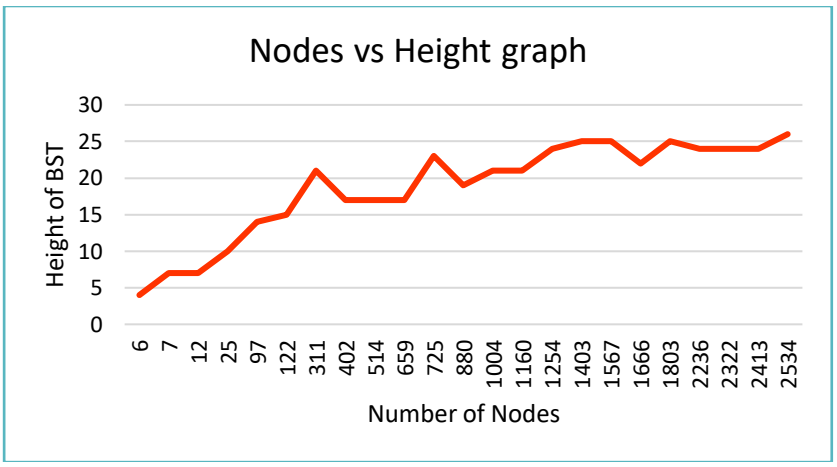


We see that the graph of log(n) is similar to that of the three trees. Therefore, number of nodes n gives nearly height of O (log n). Also, the curve for AVL tree is more accurate for log(n) than that of others.

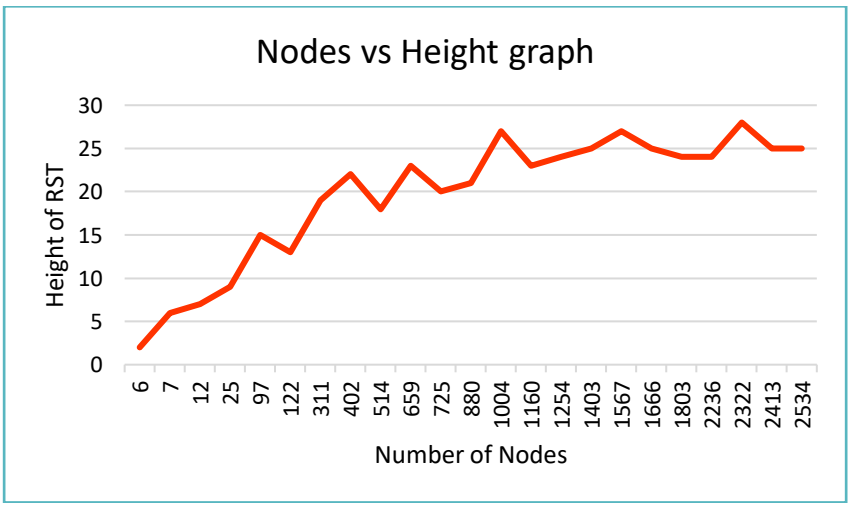
Graph of Number of nodes versus Height of AVL Tree



Graph of Number of nodes versus Height of BST Tree



Graph of Number of nodes versus Height of Treap

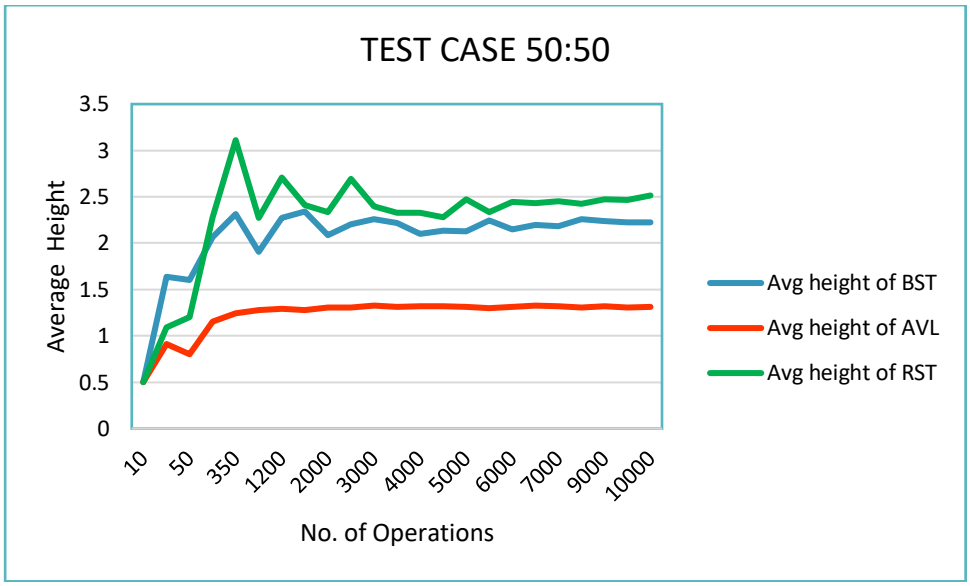


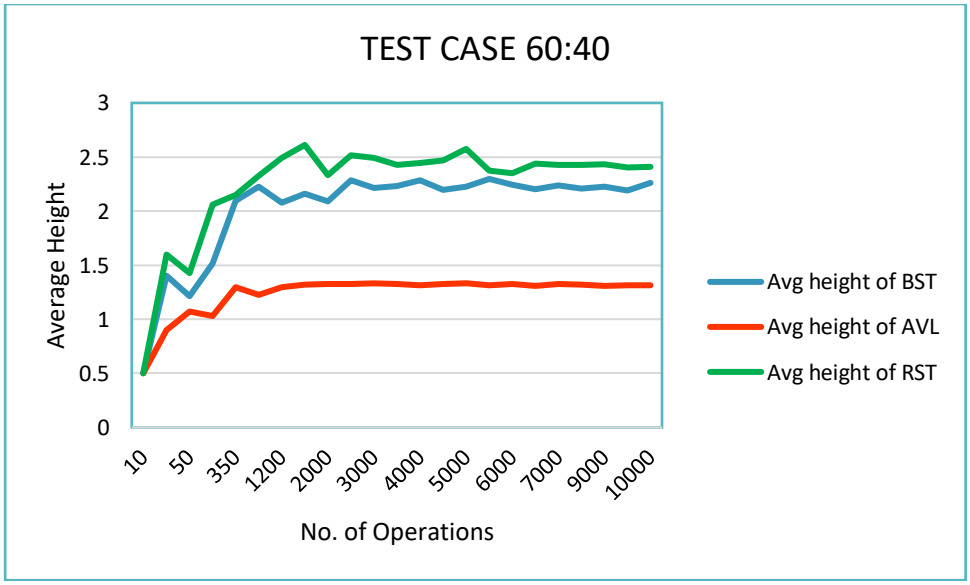
2. Average height of node of tree

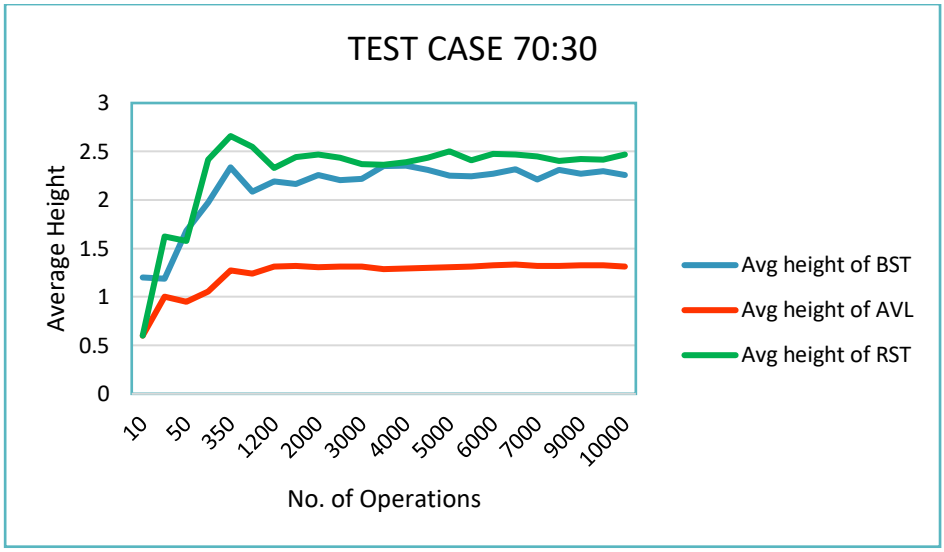
It is calculated by summing the height of each node of the tree and dividing it by the number of nodes i.e.,

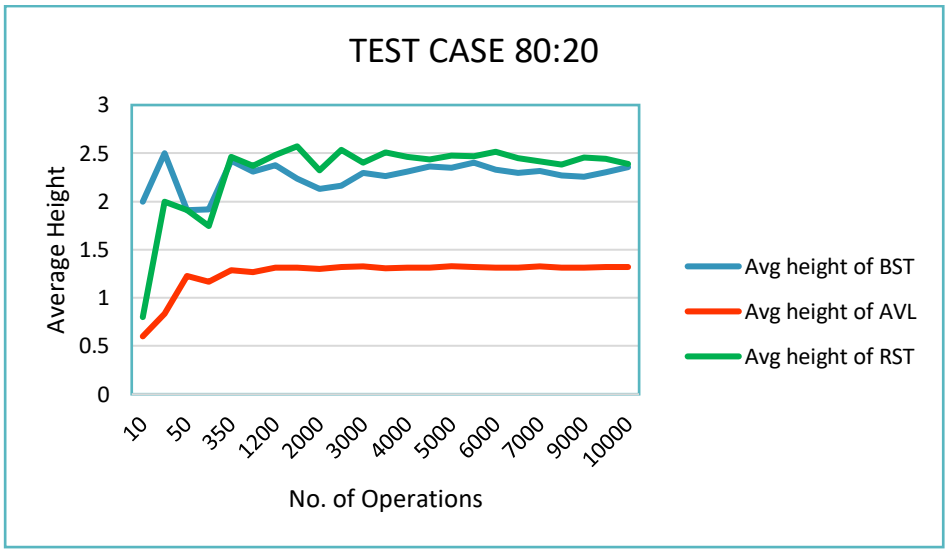
$$\text{Average height of each node} = \frac{\sum height\ of\ a\ node}{Number\ of\ nodes}$$

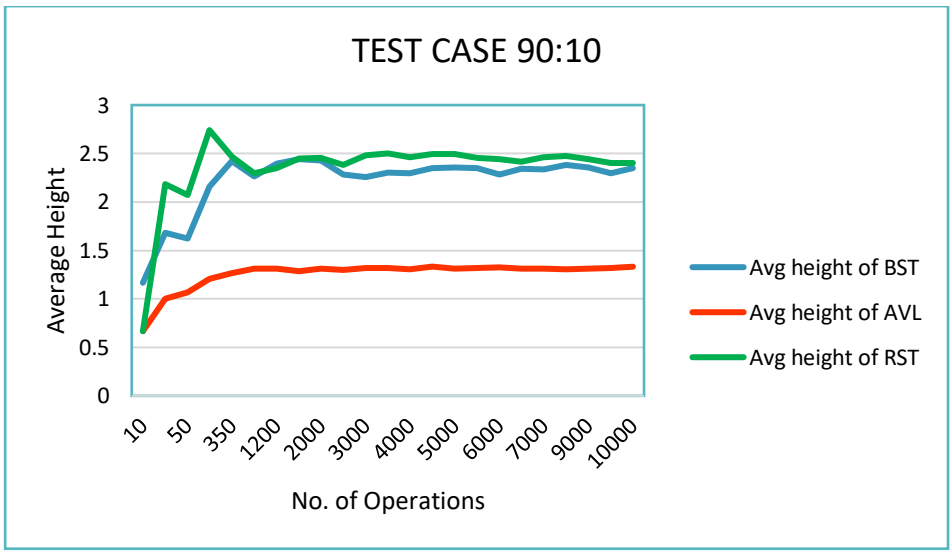
The graph with various test case types is plotted with average height on y axis and Number of operations (insert + delete) on x axis.






TEST CASE 70:30
Average Height
3
2.5
2
1.5
1
0.5
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
Avg height of BST
Avg height of AVL
Avg height of RST


TEST CASE 80:20
Average Height
3
2.5
2
1.5
1
0.5
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
Avg height of BST
Avg height of AVL
Avg height of RST


TEST CASE 90:10
Average Height
3
2.5
2
1.5
1
0.5
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
Avg height of BST
Avg height of AVL
Avg height of RST

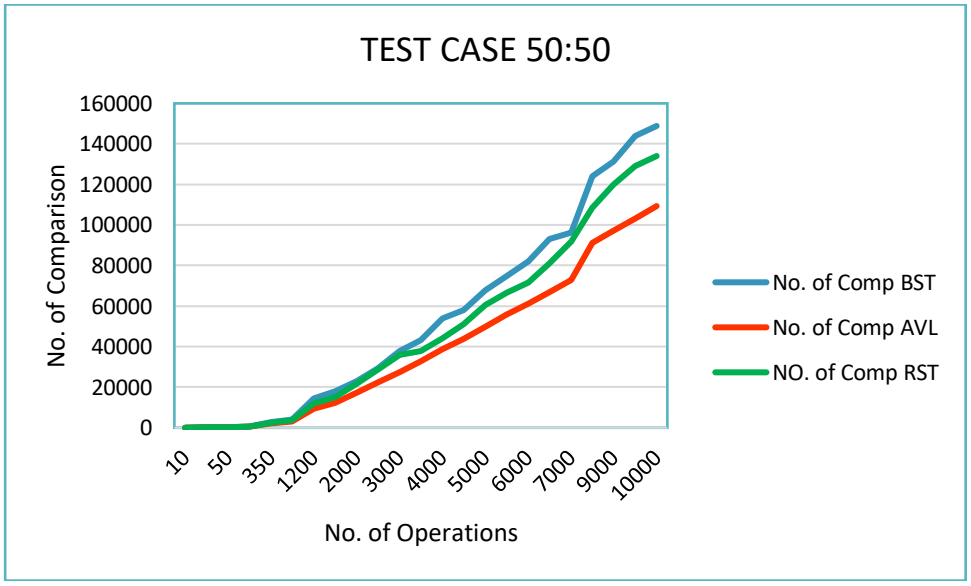
Observation:

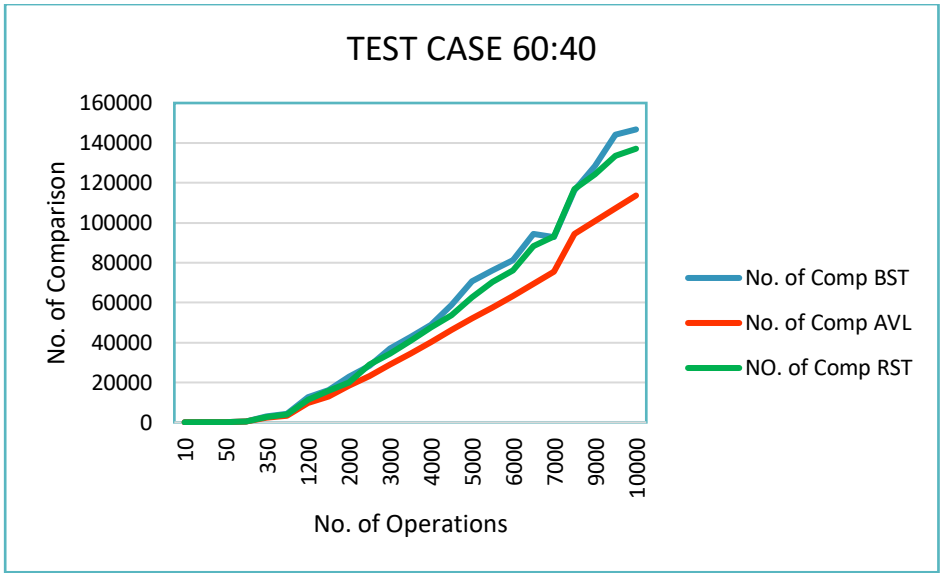
- From the above graphs we observe that Average height of AVL tree is least among the three in all cases and after certain number of operations it becomes almost constant.
- The Average height of Treap is slightly greater than that of BST and we see a lot of variances.
- The Average height of Treap or BST does not exceed 3.
- The Average height of AVL tree does not exceed 2.

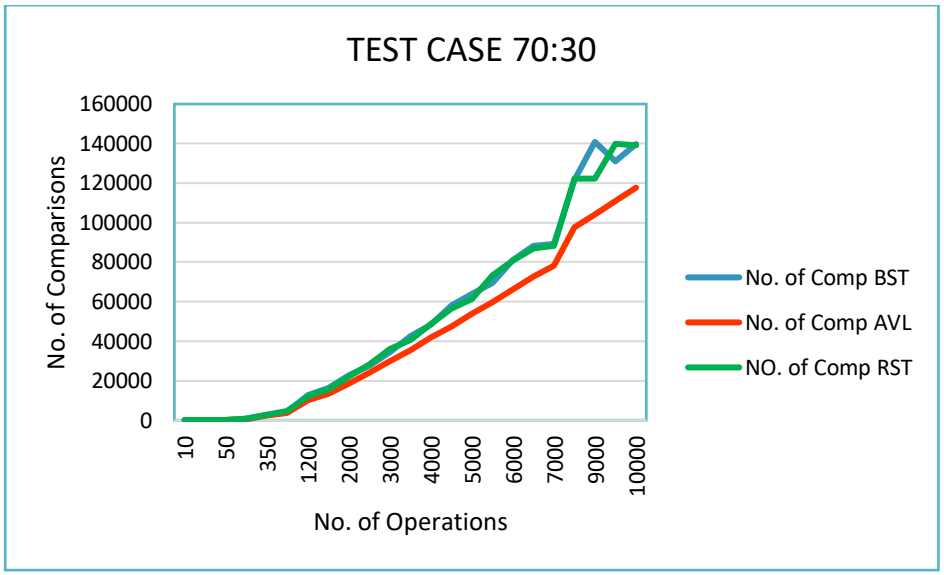
3. Number of Comparisons

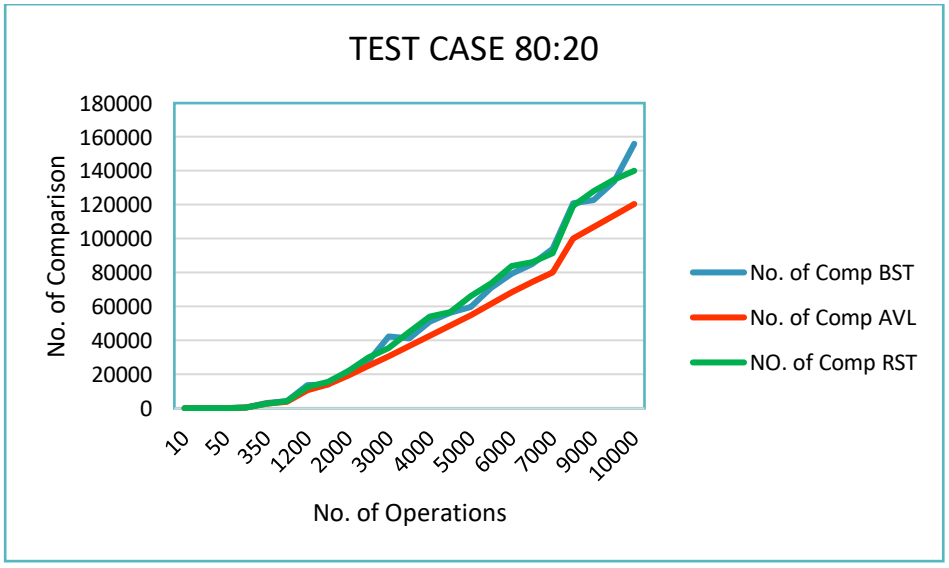
For each node visit contributes we consider a comparison in case of insert or delete. We use a class member variable total_comparison which keeps track of the number of comparisons done till that point of time. So, whenever a node is visited during insert or delete, the variable gets incremented. At the end of all the operations, we call the get_total_comparison function to obtain the value of the variable to plot the graphs as seen below.

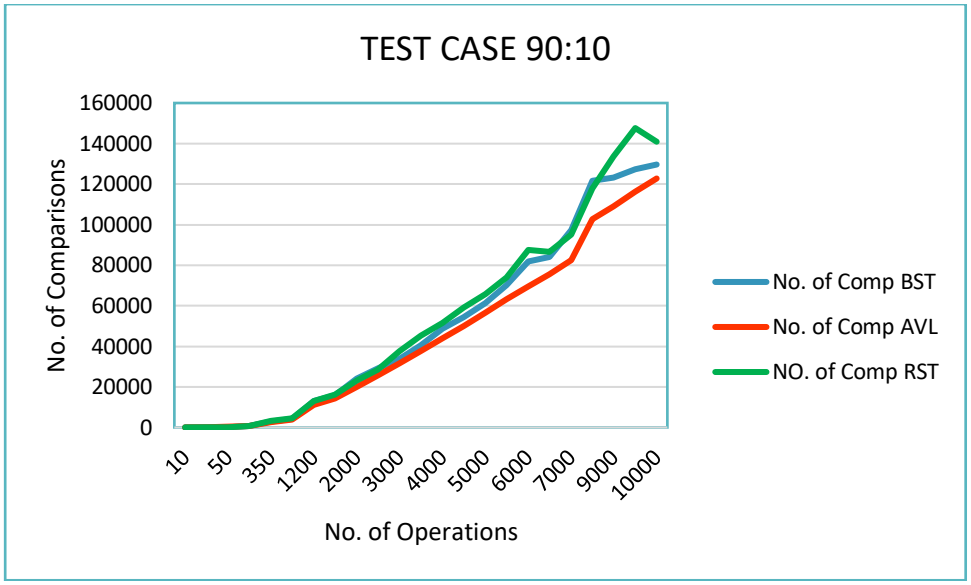
The graph with various test case types is plotted with No. of comparison on y axis and Number of operations (insert + delete) on x axis.




TEST CASE 60:40
No. of Comparison
160000
140000
120000
100000
80000
60000
40000
20000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
No. of Comp BST
No. of Comp AVL
NO. of Comp RST


TEST CASE 70:30
No. of Comparisons
160000
140000
120000
100000
80000
60000
40000
20000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
No. of Comp BST
No. of Comp AVL
NO. of Comp RST


TEST CASE 80:20
No. of Comparison
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
No. of Comp BST
No. of Comp AVL
NO. of Comp RST



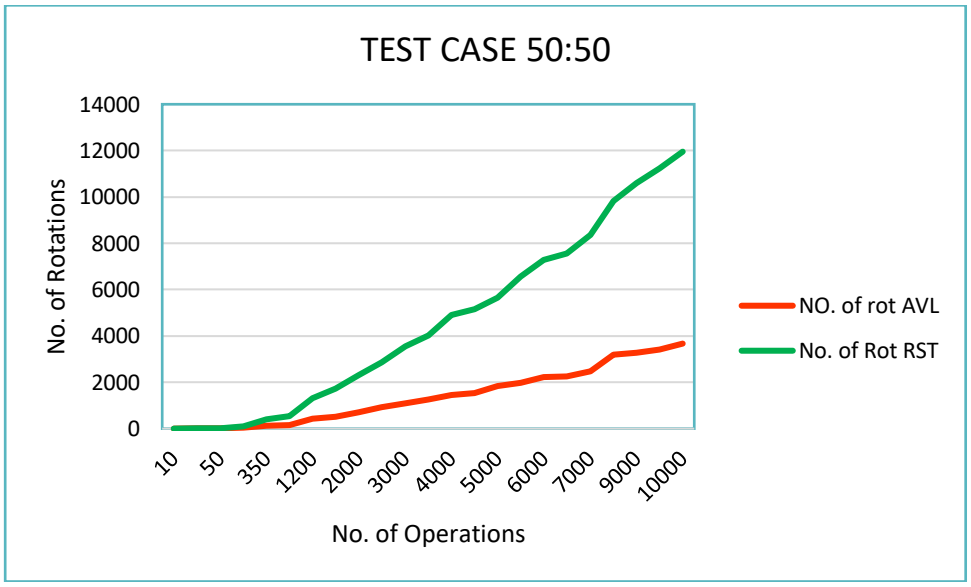
Observation:

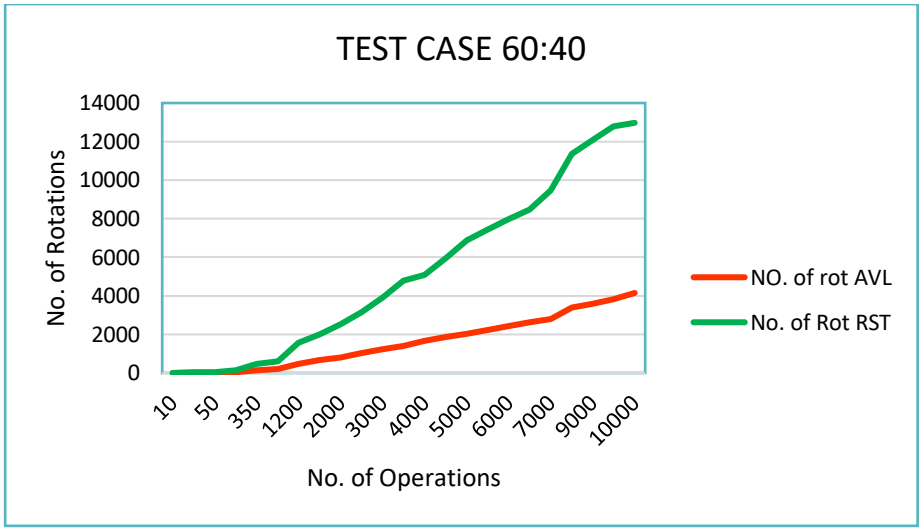
- From the above graphs we observe that the number of comparisons in AVL tree for insert and delete operation is least among the three in all cases.
- There is an interesting point to observe that with increase in insert operation and decrease in delete operation, the number of comparisons for BST and Treap become almost equal.
- We notice that these curves are close to each showing that number of comparisons ranges are similar for all three. And almost equal for a smaller number of operations as the curves are overlapping.
- The variance increases as the number of operations increases.

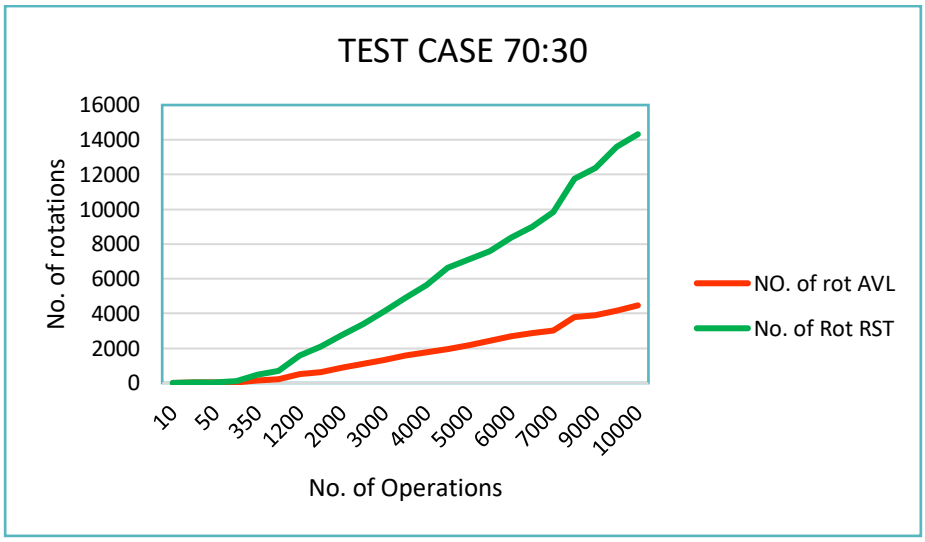
4. Number of Rotations

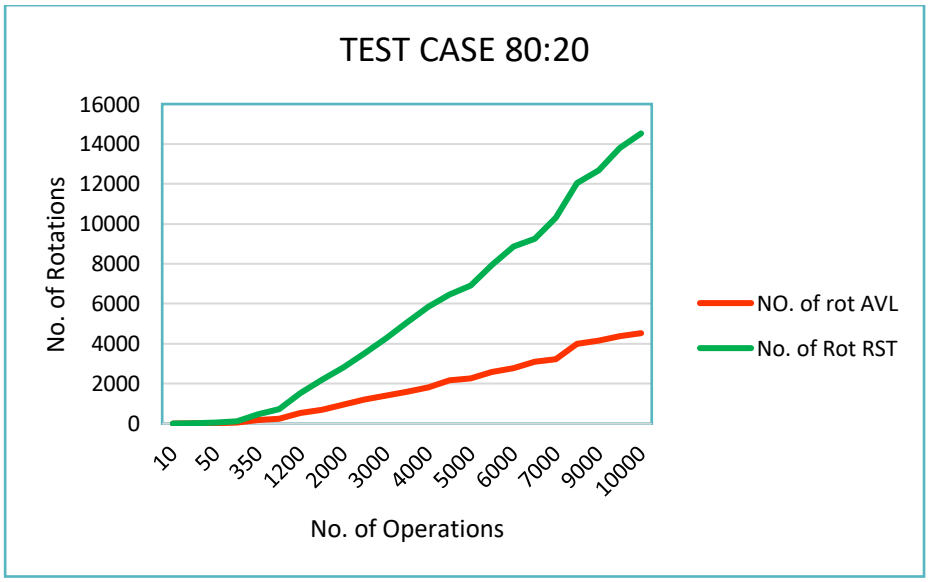
- Whenever there is single rotation or double rotation in AVL insert or delete operation, left or right rotation in Treap insert or delete, we consider it as rotation. We do not have rotations in BST.
- We maintain a total_rotations variable that is updated for each rotation performed. It is incremented twice for a double rotation.

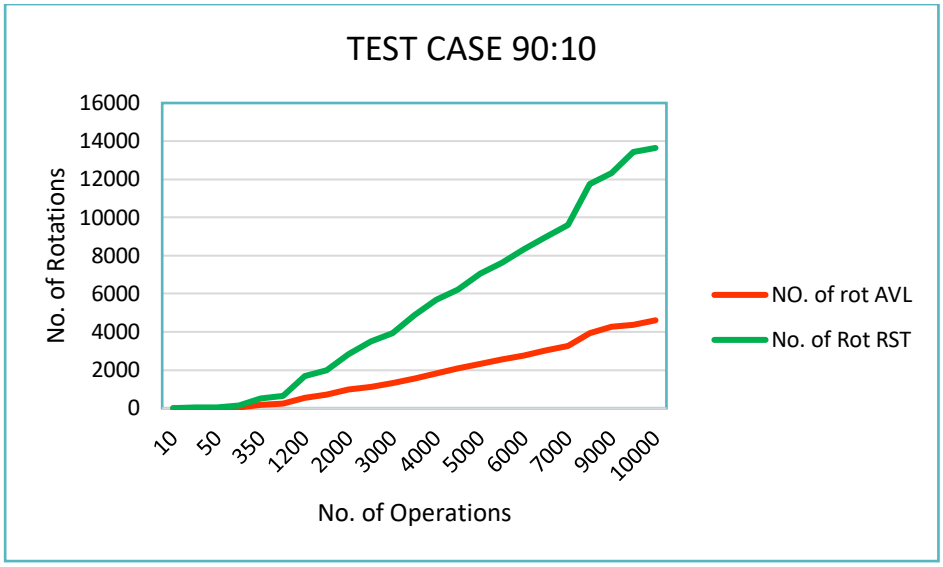
The graph with various test case types is plotted with No. of rotations on y axis and Number of operations (insert + delete) on x axis.




TEST CASE 60:40
No. of Rotations
14000
12000
10000
8000
6000
4000
2000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
NO. of rot AVL
No. of Rot RST


TEST CASE 70:30
No. of rotations
16000
14000
12000
10000
8000
6000
4000
2000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
NO. of rot AVL
No. of Rot RST


TEST CASE 80:20
No. of Rotations
16000
14000
12000
10000
8000
6000
4000
2000
0
10
50
350
1200
2000
3000
4000
5000
6000
7000
9000
10000
No. of Operations
NO. of rot AVL
No. of Rot RST



Observation:

- It can be observed that the number of rotations in AVL insert and delete operations is less than that of Number of rotations in RST.
- And, this difference is quite large.
- Notice that the difference increases as the number of operations increases.
- However, the behavior is independent of ratio of number of insert to delete.

CONCLUSION: (Comparison with Theoretical and Experimental results)

Theoretically, Idea behind randomized search trees (RSTs): If priorities are generated randomly by the insert algorithm, balance can be maintained with high probability and the operations stay very simple. The expected height becomes O (log n).

•The expected value of node depth in an RST is O (log n), and thus average time cost for successful find is O (log n)

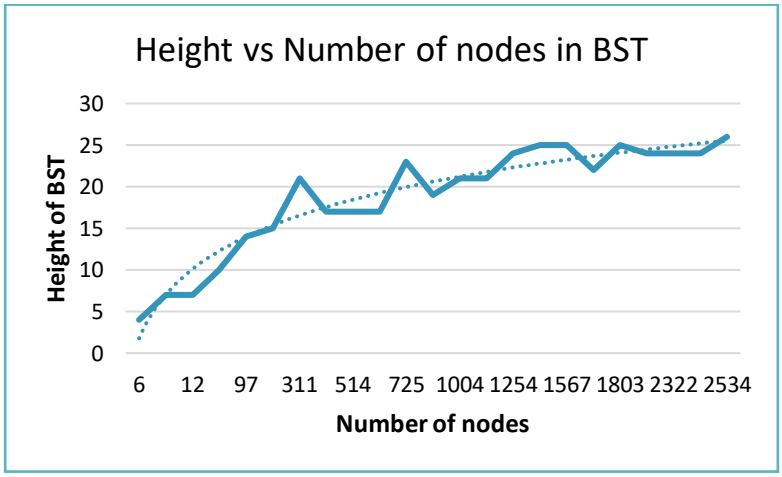
•Similar considerations show that unsuccessful find, insert, delete in an RST all have average time cost O (log n)

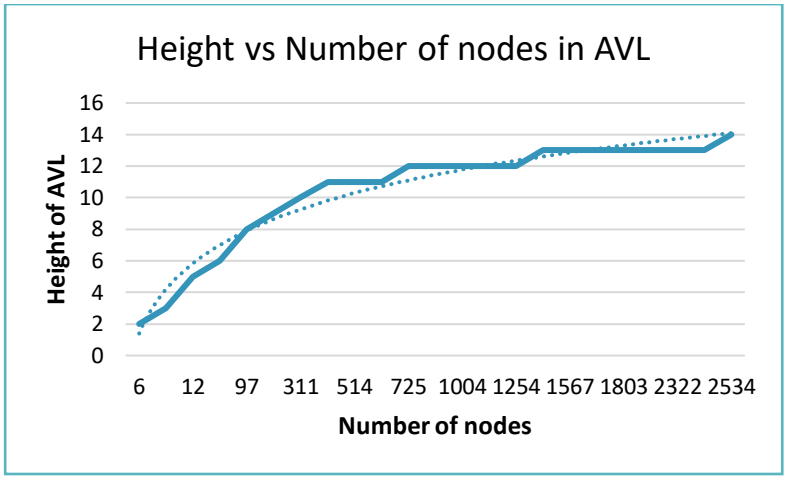
•It is possible for a randomized search tree to be badly unbalanced, with height significantly worse than log n, where n is the number of nodes in the treap; however, this is unlikely to happen.

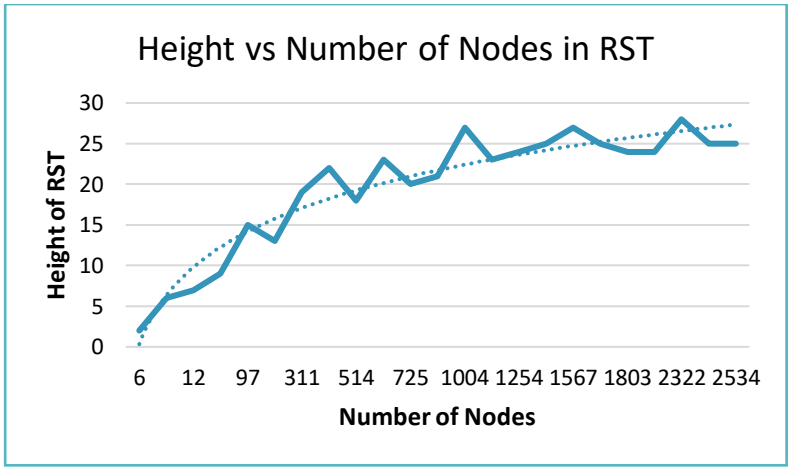
•To be badly unbalanced, the random priorities have to be correlated in a certain way with key values, and with a good random number generator this will be unlikely to occur.

Practical Results,

We can see the plots between number of nodes and height of tree. Also, we have plotted the **trendlines** that matches the curve most (shown by dotted curve). We notice that **this trendline is a logarithmic function**. Hence, this proves the above condition which states that the expected value of node depth is O (log n).







For inserting elements in sorted order (**worst test case**) e.g.,

10

Insert 1

Insert 2

Insert 3

Insert 4

Insert 5

Insert 6

Insert 7

Insert 8

Insert 9

Insert 10

Results:

BST

```
The number of key comparisons: 45
The number of total rotations: 0, As we do no perform any rotations in BST
The final height of tree is: 9
The Average height of nodes is: 4.5
```

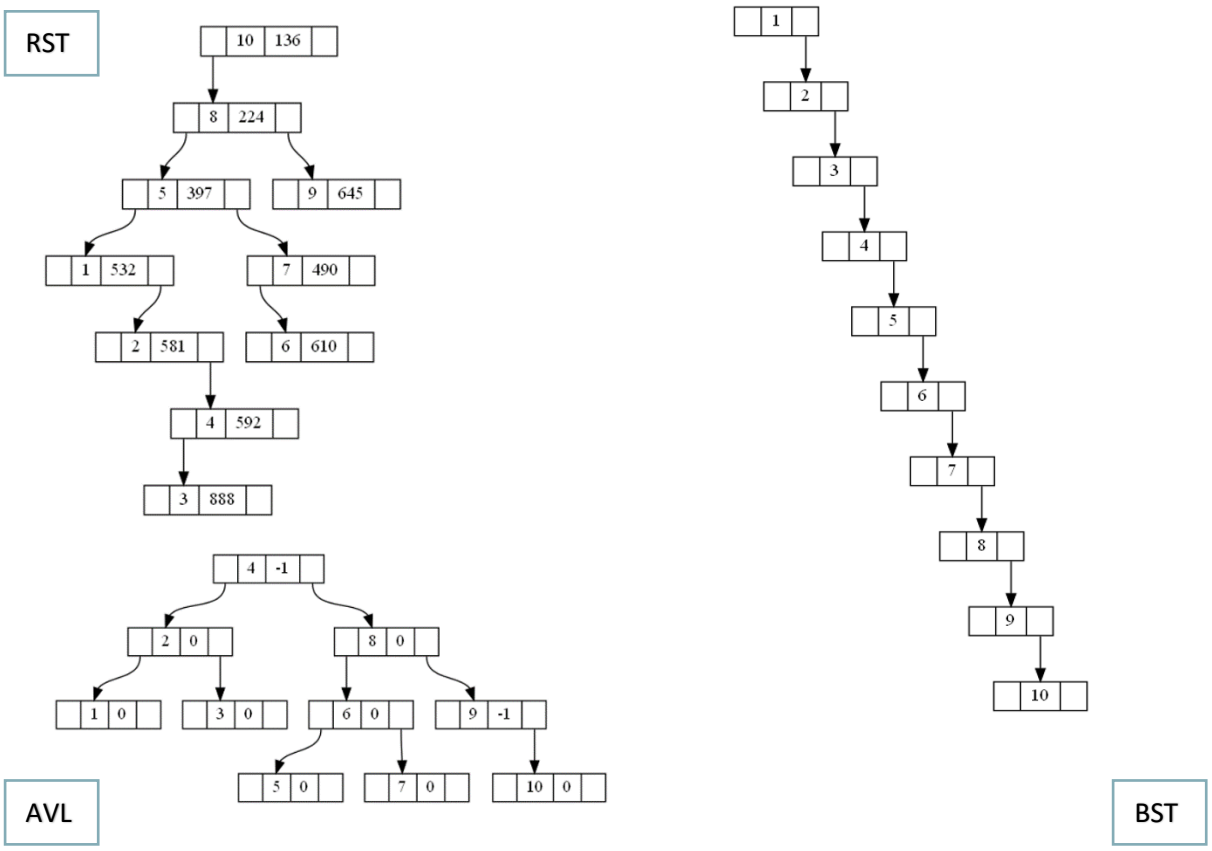
AVL

```
The number of key comparisons: 36
The number of total rotations: 6
The final height of tree is: 3
The Average height of nodes is: 0.8
```

Treap

```
The number of key comparisons: 17
The number of total rotations: 9
The final height of tree is: 6
The Average height of nodes is: 2.2
```

Hence, we see that for worst case input also the height of Treap is less than that of BST is n-1 (10-1, skewed tree) and as the number of nodes increases for worst case input RST will have lesser height than BST. Height of BST will be O(n) and that of RST will be nearly O (log n). It is very less probable that the priority order also has worst case.



Finally, we conclude with following results after rigorous experiments and comparison:

Treaps are worth studying as:

- They are very easy to implement. The join and split operations can be implemented very easily.

- They have performance comparable to that of BST.

- The worst case of being a skewed in treaps is rare but BST may have it often. This can lead to worst case complexity of operation as O(n).

  Also,

- Avl Trees have a strict balance condition therefore, height of AVL trees for even the worst case will be less. But the cost of maintaining it is, complex implementation of insert and delete operations.

- Whereas, the number of comparisons done are almost similar in all the three.

- The number of rotations however, are less in AVL than in Treaps.

----------THANK YOU----------