Assignment – 3

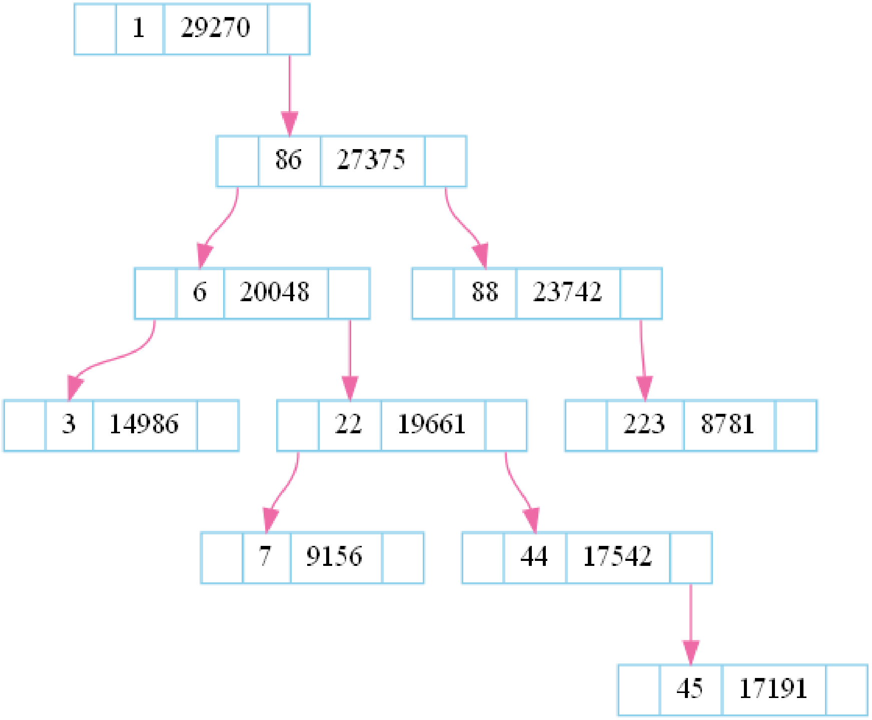
DsLab.

Ankit Kumar

214101008.

1.Introduction to Randomized BSTs – Treaps

A treap is a binary tree in which ever node has a search key as well as a random priority the keys follow the property of a binary search tree and the priorities follow the heap property .
A simple treap is shown below.



The values in the right cell is the random priority.
And the values in the left cell is nothing but the key value.

1.1 Treaps Insertion

In case of insertion, we traverse the tree from root towards the leaf by following the property of binary search tree and attach the new node at its right position. Then we check for the heap property. If the priority of the newly inserted node is less than the priority of its parent, then we do a single rotation after which the child becomes the parent and vice versa. This process of checking and rotation will go up till root or till the point where we don't get such inversion condition. The worst case time complexity is O(n) and average case time complexity is O(log n).

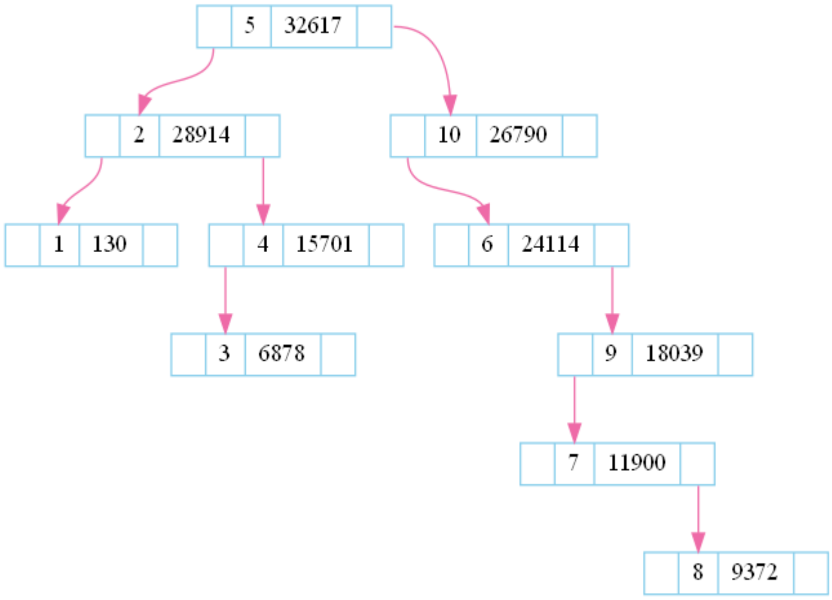
Part 1 – First we go and find the place where this node is to be inserted just like we did in bst if my new value that I'm trying to insert is less then some key we go left otherwise we just go right till we reach some leaf and then there we just insert the node.

Part 2 – After the insertion we go and check if the priority is lower then the parent since it's a max heap when it comes to the priority thing then we just leave it there cause all the parents will have higher priority then the new node.

If the priority of new node is greater than the priority of its parent then we rotate it once either left or right rotation depending on which child it is

Left rotation:= when new node is right child of parent then parent be comes left child of new node and new nodes left child becomes right child of parent

Right rotation:=when new node is left child of parent then we make right child of new node as parent and set parents left child as new nodes right child. We do this till new nodes priority is less than priority of parent or we reach root of the tree.



Above we have inserted 1 to 10 key values in a bst in ascending oreder since It's a random priority data structure that means that it's not necessary that 1 will be the root and 10 will be the leaf even if that's the order you choose to insert this in.

1.2 Treaps Deletion

In delete function we first search whether the node is present by calling search () function with the key value, if its present we search the location of the node else we return.

After finding the node with key value we try to find whether its a leaf or a one child parent or a two child parent.

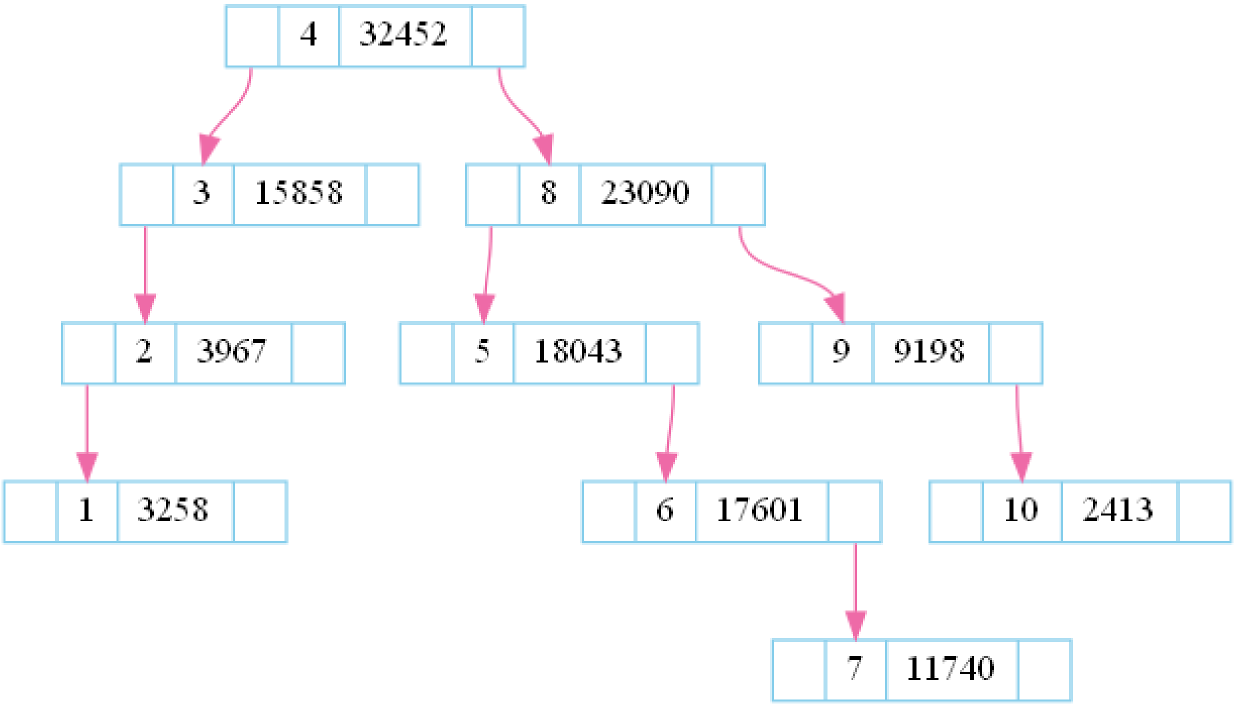
If its a leaf : We simply delete it and set its parents child pointer as null and no rotations are required.

If its a one child parent: We make its parent child as child of node to be deleted and here also no rotations are required.

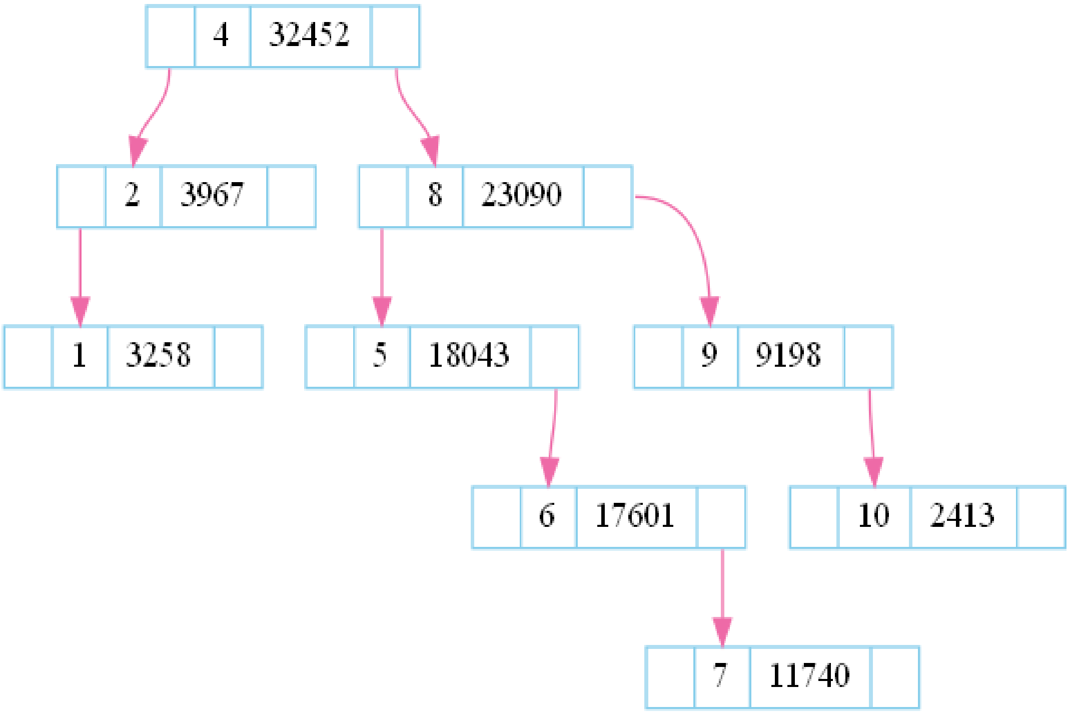
Two child parent: Here we will require to perform rotation as one of the child of node to be deleted will replace this node. If priority of left child is less priority of right child then we do right rotate else we do left rotate.

1. **Left rotate:**
   When right child of parent is having less priority than left child then right child will replace the parent. Right child of parent will be left child of parents right child . Left child of parents right child will be parent

2. **Right rotate:**
   When left child of parent is having less priority than the right child then left child will replace the parent. We set left child of par ent = right child of left child , left child right child of left child of parent becomes parent. We keeep on doing this until node becomes parent of one child or leaf



Treap before and after deleting 3



1.3 Treap search

Here the key given will be searched with respect to key of root and if it is less than root->key then we will search left subtree of root else right subtree and repeat until root becomes null or we find the key. If found we return 1 else we return 0.

1.4 Print Treap

The print function will recursively traverse my tree and create a dot file which will then be ran with a graphviz and it will create a png file in the same directory as you've already seen some pictures above.

2. Random Testcase Generation: Logic and Procedure

We have given choice to user to make as many test case files as possible with as many no of operations and percentage of deletion they want in the test cases.

For test case generation we are gonna use a srand(time(0)) function where the seed is time (0)

And according to the no of operations I take random integer key as input and take mod 20 of key into an integer type .

If the value of temp > 20*delete_percentage/100 then it is insert operation and I write insert key value onto the file and also add this key in a vector .

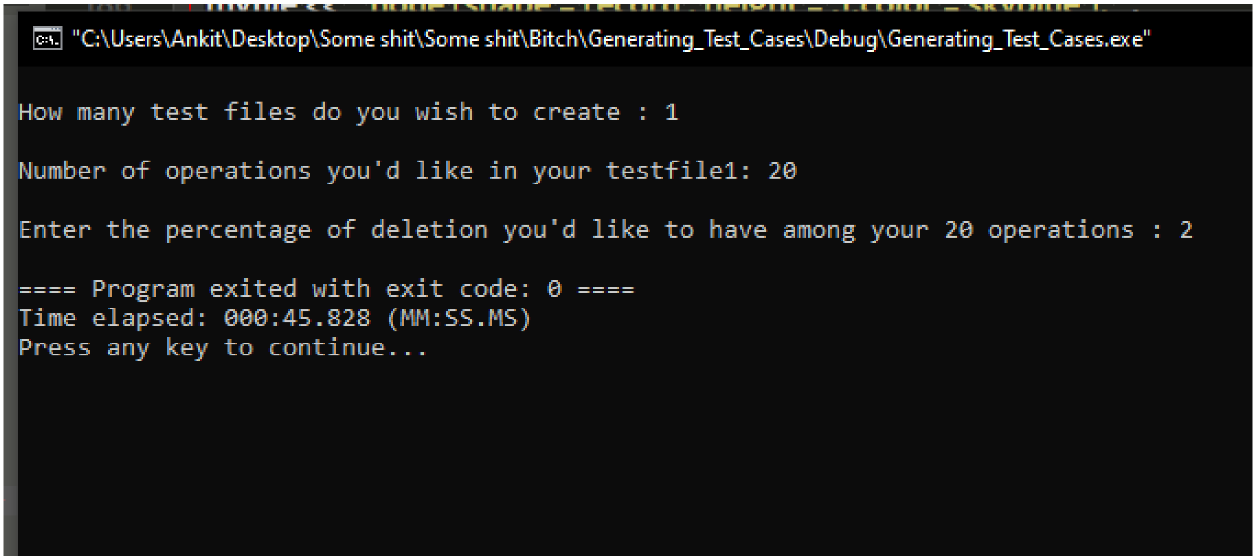
Else I just make the operation to be delete this helps bcz it just put the insert and delete operations in the ratio we want it to.

The test case files will then be generated in the same directory I've also taken care of the case where someone just puts the deletion percentage very high even higher then 100% there it will just ask you for re enter.

Here you can just see the procedure in action as I am making one test case file.

This will give us a testfile1 in the same directory.

Look below that's the console



"C:\Users\Ankit\Desktop\Some shit\Some shit\Bitch\Generating_Test_Cases\Debug\Generating_Test_Cases.exe"

```
How many test files do you wish to create : 1

Number of operations you'd like in your testfile1: 20

Enter the percentage of deletion you'd like to have among your 20 operations : 2

==== Program exited with exit code: 0 ====
Time elapsed: 000:45.828 (MM:SS.MS)
Press any key to continue...
```

3. Performance Evaluation of Treap : Comparison with Binary Search Tree and AVL Tree

3.1 Parameters taken for analysis:

I have taken 4 parameters for analysis which are

- **3.1.1 no. of comparisions while inserting and deleting**
- **3.1.2 no. of rotations for performing all the above operations**
- **3.1.3 height of each tree**
- **3.1.4 avg height of each tree**

3.1.1 No. of comparisions while inserting and deleting

The no of comparisons are used to check how mant node comparisons we do in order to reach the node when we are inserting or deleting a node.

In case of BST:

For insertion we search the node from root till we are at leaf or the node we are inserting is already there in the tree, for each comparison we increase the comparisons count of bst insertion.

Similarly for BST delete we search the node from root till we find it and each time we go downwards we increase the comparisons of bst delete.

In case of AVL:

for insertion we do exactly like in case of insertion in bst and make a variable comparisons_avl to store count of all comparisons while inserting.

And for deletion in AVL it is same as in case of deletion in bst.

In case of TREAP:

for insertion and deletion we do exactly like in case of bst and avl.

Why I take this parameter:

We have seen that in case of a bst the worst case height can be O(n) that is no of nodes in bst , and in case of avl the worst case height is 1.44 log(n), and in case of treap even if we give no is ascending order or descending still the height will not reach O(n) because of random priority we are assigning to the node.

3.1.2 No. of rotations while inserting and deleting

The reason we took rotations to check which among the avl and treap has more rotations

In case of BST: the no of rotations in bst for insertion and deletion is zero as no rotations are done.

In case of AVL: for insertion we require 4 types of rotations .

LL : when the new node is inserted in the left child of a node which was alredy having a height imbalance of 1 then we have a single LL rotation.

RR: when a new node is inserted in the right child of a node which was already having a imbalance of -1 so we do a single RR rotation.

LR: when a node is inserted in right child of a node which itself is a left child of a node having imbalance +1 so we do a double rotation.

RL:when a node is inserted in left child of a node which itself is right child of a node which was having a imbalance of -1, so we do a double rotation.

Here in avl tree insertion rotation is happend atmost once per insertion.

For deletion in avl :

Here we do rotations after the node is deleted , we store all the nodes from root till the node we are deleting and after deletion we check the balance factors of each node and if there is imbalance then we do rotations.

There are 4 types of rotations here also.

In case of TREAP: for insertion in treap:

Here we do rotations for after insertion of the node and do single rotations only either left rotaion or right rotation when the priority value of parent is greater than that of its child.

For deletion: when we want to delete the node which is leaf or which is parent of only one child then there is no rotation. But if are deleting the node which has 2 child then we rotate till the node becomes a node with one child.

We do single rotation each time it has two child and new parent is the child having less priority no.

3.1.3 Height of each tree

For bst:we call a function height which will calculate the height of final bst . here height of root is 1 .

For avl we call a function which finds hight of final avl tree

For treap we also call a height function to calculate final height of final treap

Here the height of all three trees vary from O(log(n)) to O(n).

Usually height of bst will be more towards O(n)

And height of AVL and Treap will be lower more towards O(logn).

3.1.4 Avg height of each tree

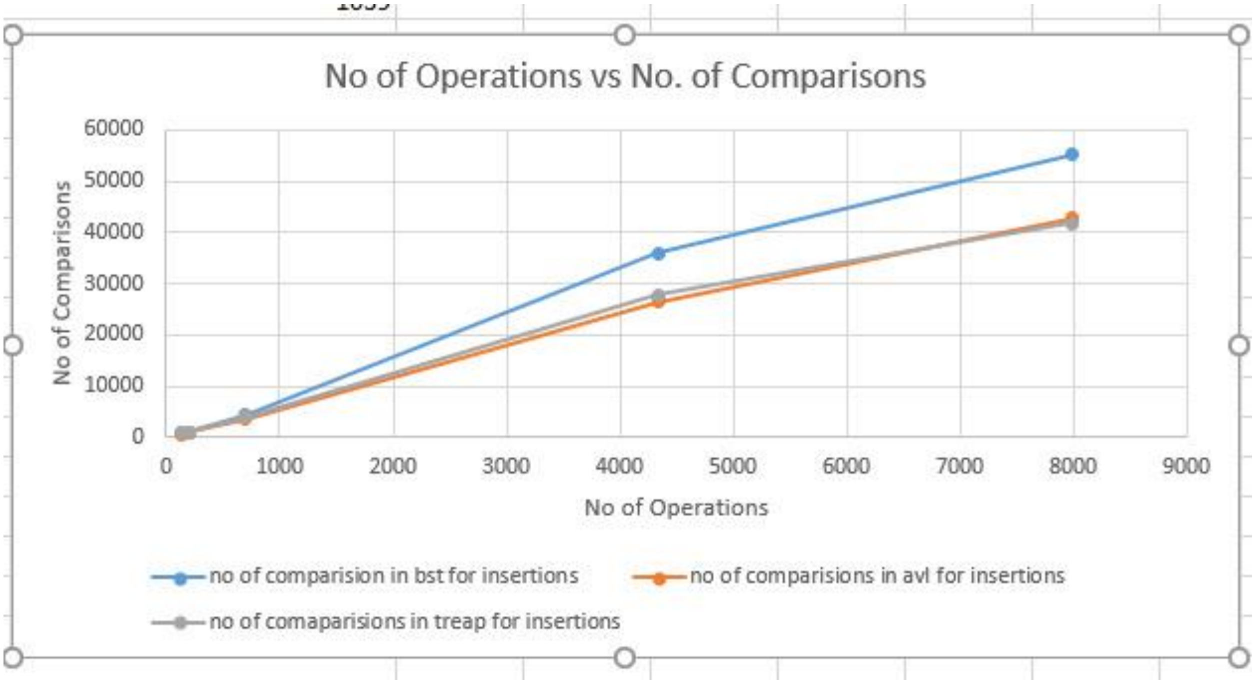
The average height of a tree is calculated by finding height of each and every node form root and then dividing it by total no of nodes in tree.

For bst ,avl and treap I call the average height function which perform inorder traversal on the tree and find distance of each node form root and add then in a variable then we divide the sum of all height with the total no of nodes in the tree and get the average height.

This will usually be in O(logn) for our AVL and Treap apart in BST it will usually be higher in case the insertion is in somewhat of an ascending or descending order.

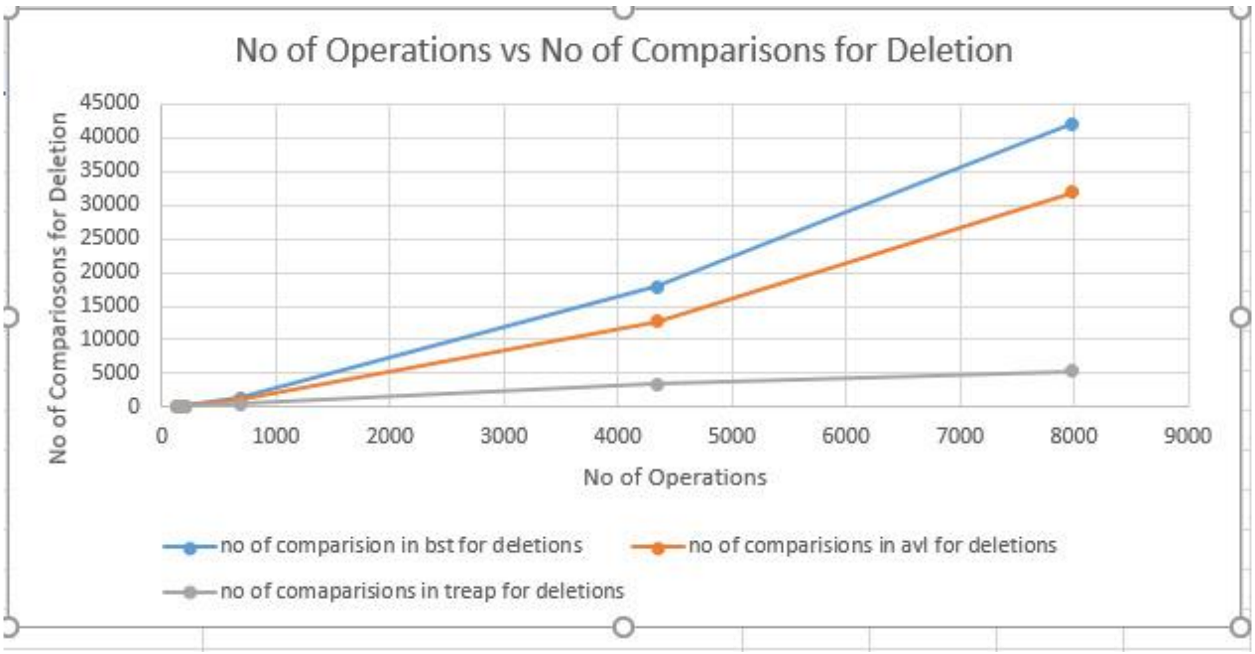
The graphs below specify how all the above parameters change as the data sturcures containg them changes

<u>1.Number of comparisions due to insertion</u>

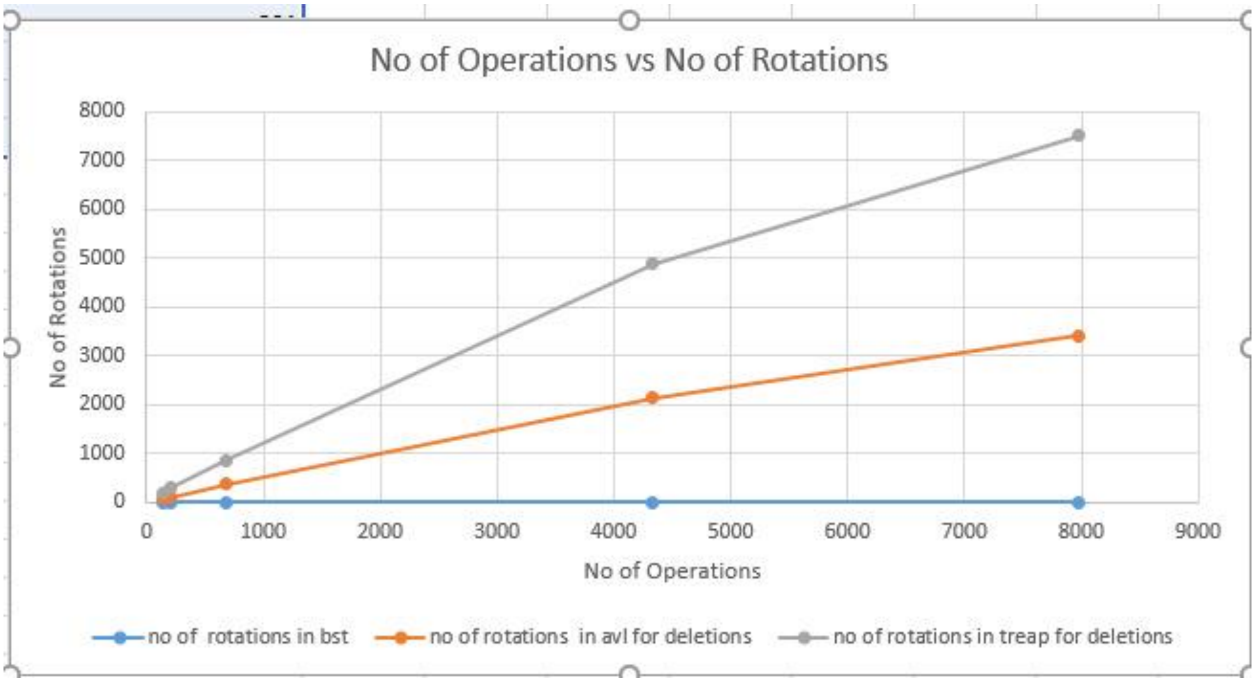


The points on each of these lines specify the actual value for example when oeperations we about 8000 the no of comparisions in bst we the highest at about 55000

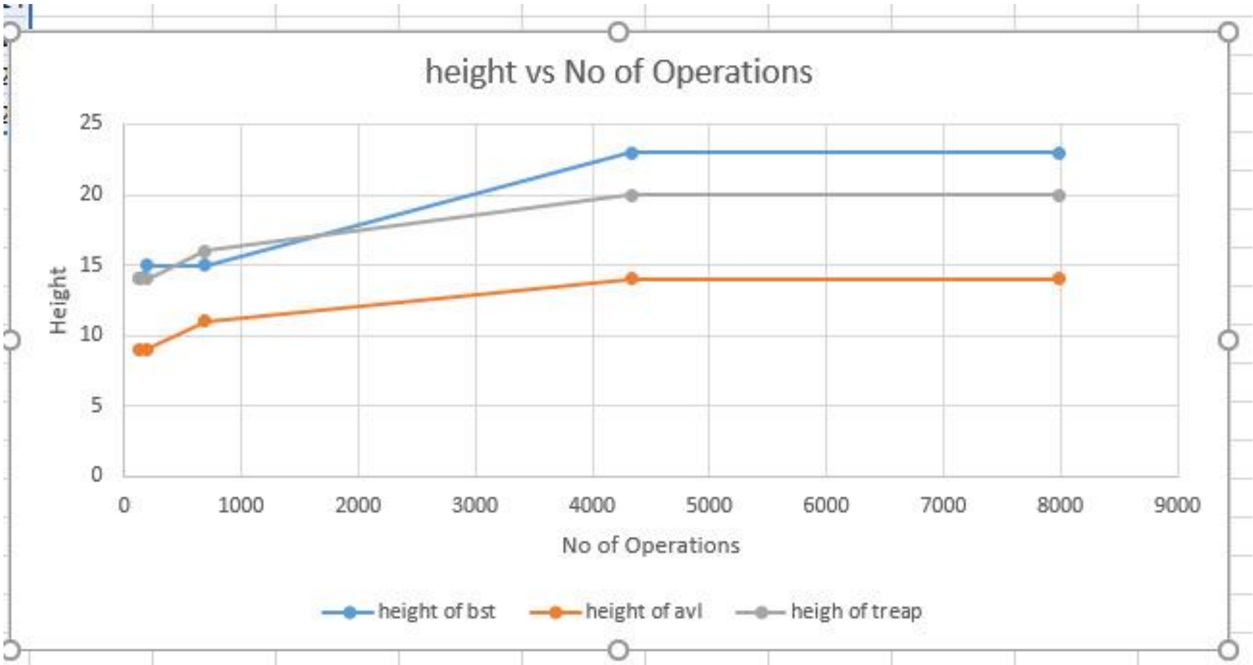
<u>2. No of comparisions due to deletion</u>



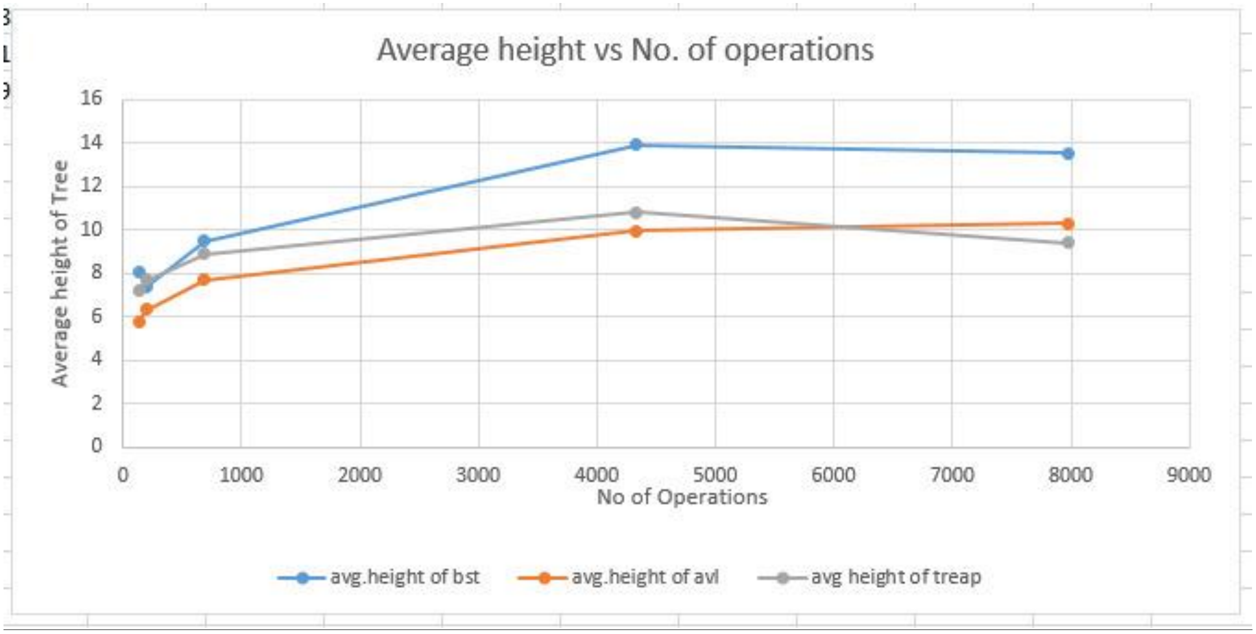
3.) No. of rotations



4.) Height of Each tree



5.) Average height of each of the data structure



Analysis portion

AVL Tree clearly beats both treap and binary search tree. For most of the cases, we can observe a stability or a stable increase (or decrease) in case of AVL tree whereas the results for treap and binary search tree are not very stable. Also, AVL tree gives better performance in almost all types of parameters. As discussed, randomized search tree should perform better than binary search tree in most of the cases if we have a good random number generator in case of priority generation. But, here is a catch. Since in our experiment, we generated the key values randomly too, binary search tree is giving almost similar performance or somewhat better performance than randomized search tree, which might not happen in real world scenario if we have skewed or partially skewed key set. In that case binary search tree will give much worse results than randomized search tree.

Treap due to having a randomized priority beats bst obviously as it is proven that if you insert 10,000 Nodes in a treap the probability that your tree exceeds the height of 100 is 1 in 2.5 billion so it will always beat bst.

But AVL is almost always better then Treap and Bst.