DOCUMENTATION FOR ASSIGNMENT III

Treap Data Structure

Description of the data structure

A Treap is a hybrid tree with properties of both a binary tree, and heap.

Treap = Tree+Heap

The tree is defined as a class **Treap**. It has a friend class **TreapNode**, which defines the component nodes of the Treap. It is also the friend class of Treap.

Each node of the tree has four components:

- **key:** The value associated to the node (type: int)
- **priority:** The priority of the node (type: int)
- **LChild** and **RChild:** The pointers to left and right children of the node respectively (type: TreapNode*)

There exists a dummy node **head**, and the root node of the entire tree is the right child of it.

The binary tree property is managed according to the key values of each node. It is input-based.
The heap property is managed according to the priority values of each node. This is randomly selected from 1-100.

All the functions are implemented in header file **treap.h**. In the file **treap_main.cpp**, a user interface is provided where inputs can be given for the functions, which are described in the document. A sample tree is also provided.

insert_Treap(k,p)

Logic:

Inserting an element in the Treap is a three step process:

1. Finding insertion location:

Here, the location where the new element is to be inserted, is found. But first, a new node is created with the key as k (provided by user input) and the priority as p. The priority is actually a random integer chosen using current time as seed. The range of priority is 1 to 100.

To find the location of insertion, we compare the values of k and key of the current node. If at any point k == key(current node), we abort, as the element already exists.

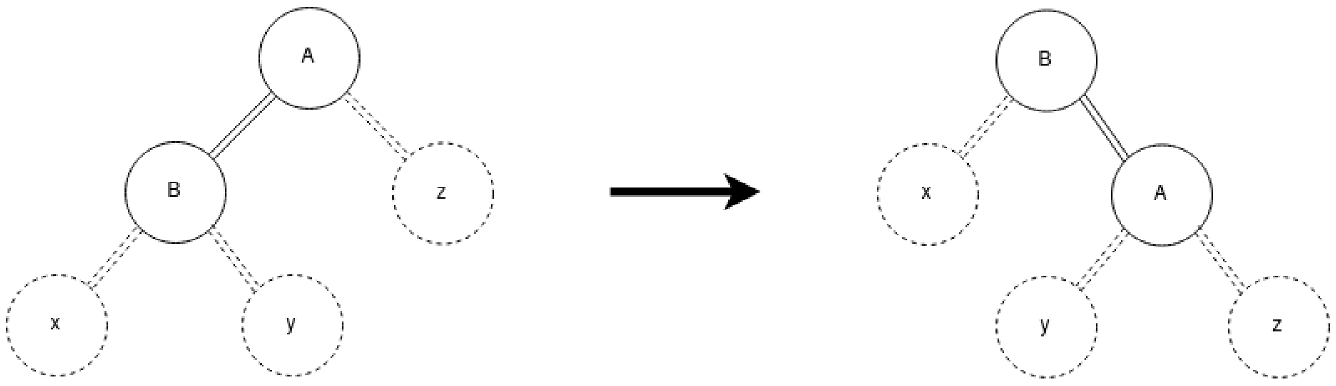
If k is less than key of the current node, current node moves to its left node. Else, it moves to its right node. This comparison will keep happening unless current node is null. Then, we will insert the new node to current node's parent's appropriate child. Eg. if k < key of current node's parent, it will be inserted as its left child.

2. Rotation according to priority:

The new node is inserted as a leaf. We will compare the priorities of the new node, and its parent. If the priority of the parent is more than the inserted node, rotation happens and the new node becomes the parent of the original parent node. There are two possible cases:

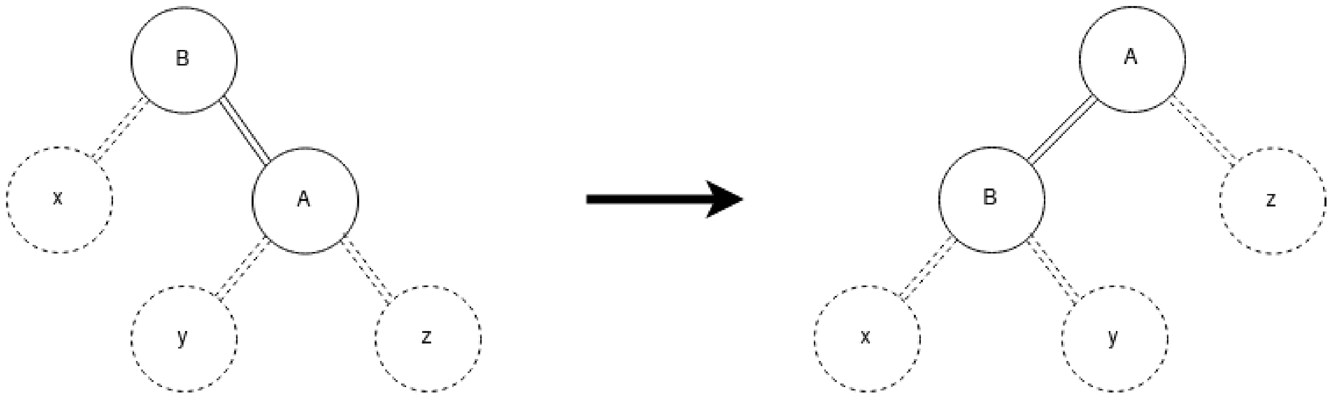
If new node is left child of parent

In this case, there will be a right rotation which will be as follows:



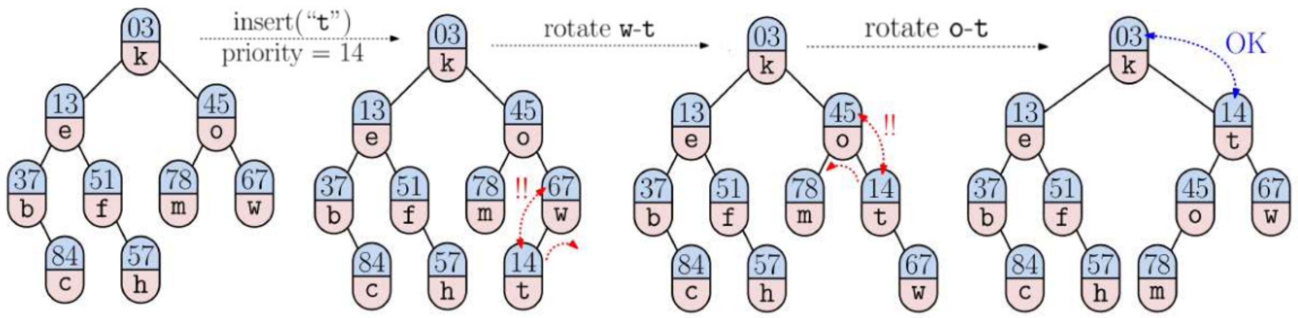
If new node is right child of parent

In this case, there will be a left rotation which will be as follows:

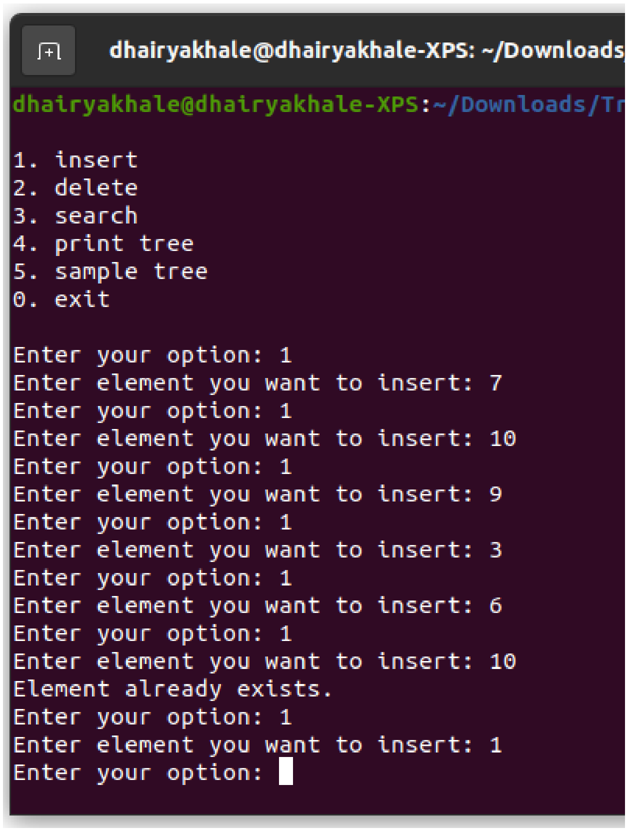


This priority check and rotation keeps happening until the priority of the parent is more than the priority of the new node.

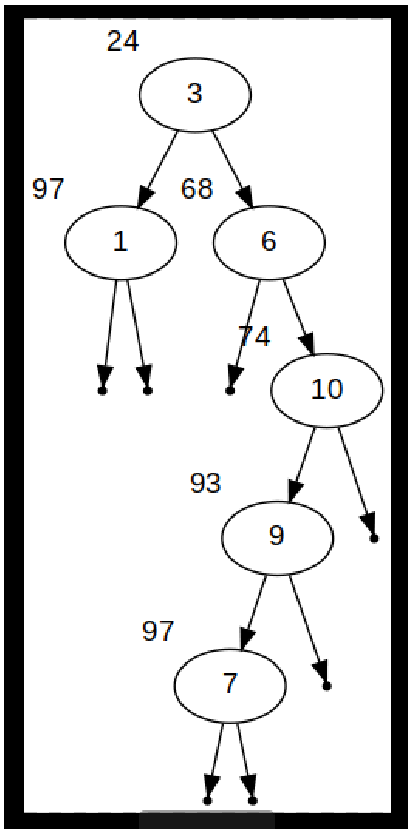
A sample insertion looks as follows:



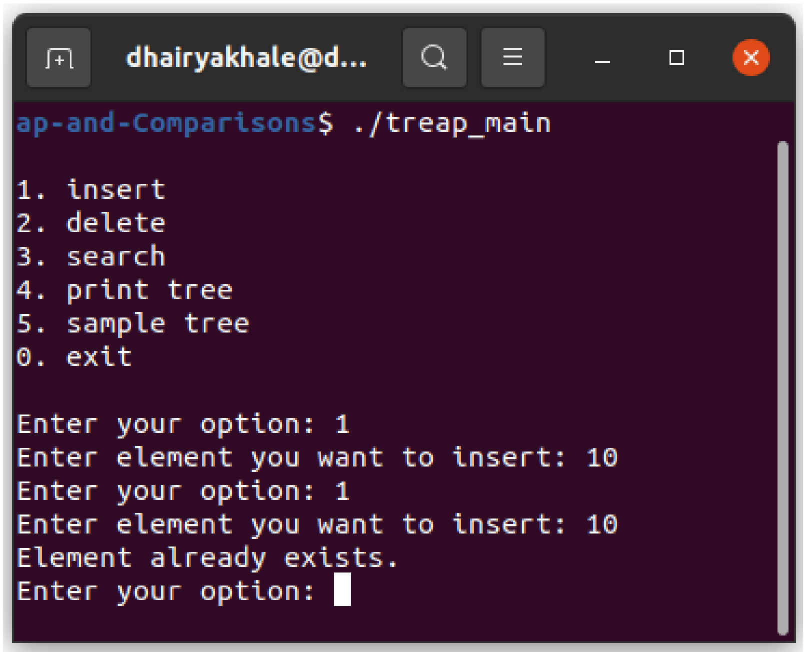
Test Cases:



```
dhairyakhale@dhairyakhale-XPS: ~/Downloads
dhairyakhale@dhairyakhale-XPS:~/Downloads/Tr

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 1
Enter element you want to insert: 7
Enter your option: 1
Enter element you want to insert: 10
Enter your option: 1
Enter element you want to insert: 9
Enter your option: 1
Enter element you want to insert: 3
Enter your option: 1
Enter element you want to insert: 6
Enter your option: 1
Enter element you want to insert: 10
Element already exists.
Enter your option: 1
Enter element you want to insert: 1
Enter your option: 
```



A sample insertion



```
dhairyakhale@d...
ap-and-Comparisons$ ./treap_main

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 1
Enter element you want to insert: 10
Enter your option: 1
Enter element you want to insert: 10
Element already exists.
Enter your option: 
```

Insertion of an already existing node

delete_Treap(k)

Logic:

Deleting an element from a Treap is also a three step process.

1. Finding location for deletion:

Here, the node which is to be deleted, is found. But along with this, we also keep pushing the parents encountered in the traversal into a stack.

To find the node for deletion, we compare the values of k and key of the current node.

We start from the root. If k is less than key of the current node, current node moves to its left node. Else, it moves to its right node. This comparison will keep happening unless key of the current node is equal to k. If at no point k is equal to key of the current node, and current node becomes null, that means the node does not exist in the tree. So we abort.

2. Deletion with rotation:

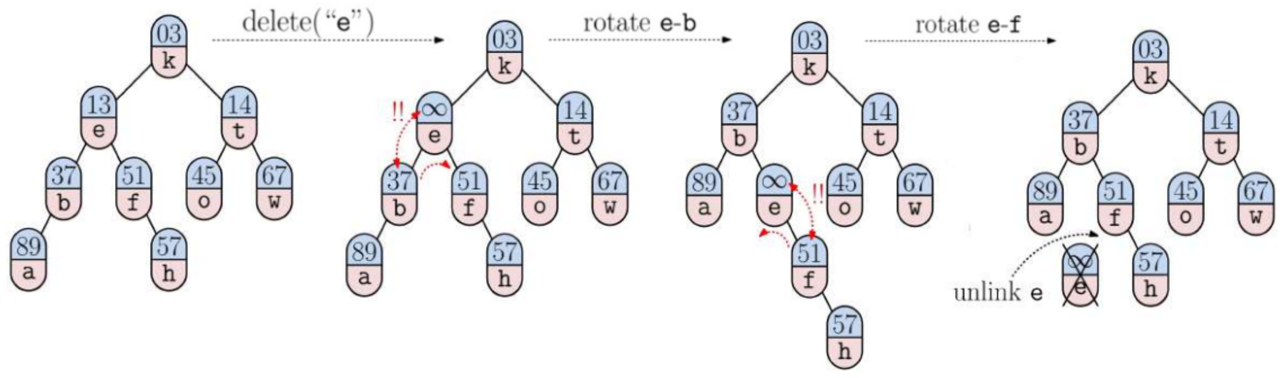
When the node is located, the priority of the node is converted to infinity. Now, we do something opposite of insertion: Priority comparison with children and rotation, hence in the downward direction, opposite to insertion.

If the node has a single left child, it will do a single right rotation. If the node has a single right child, it will do a single left rotation.

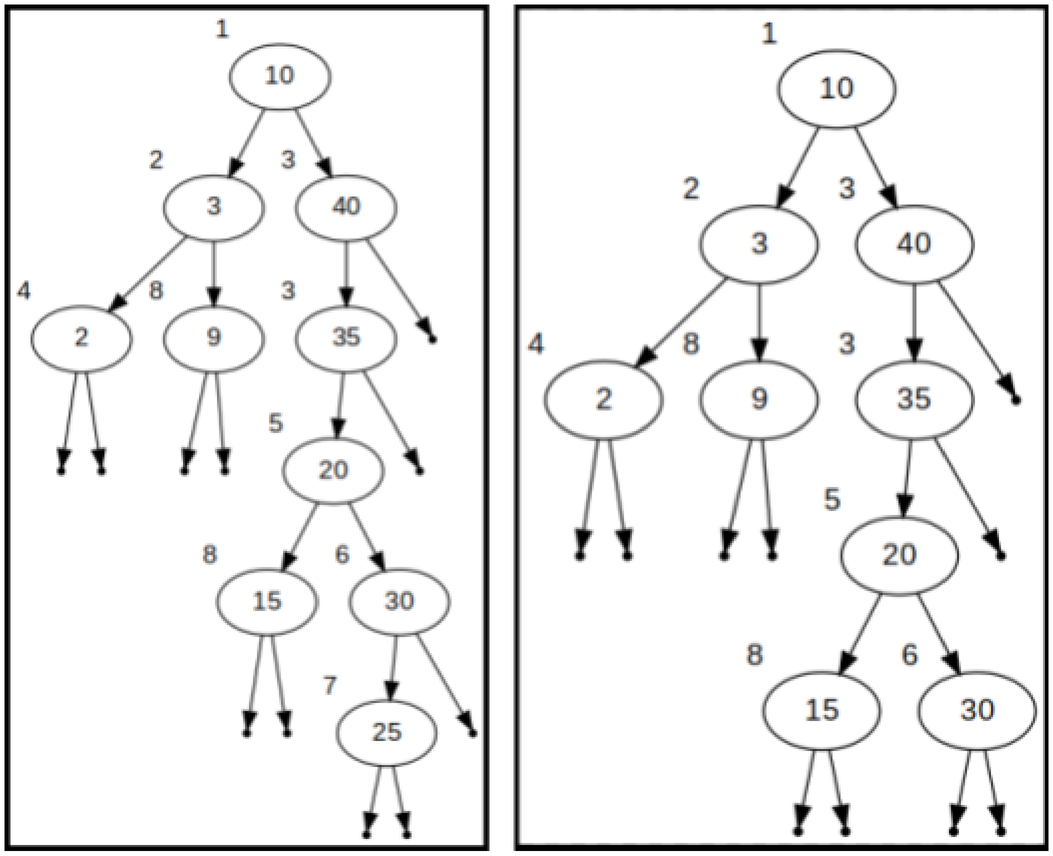
If the node has two children, it will choose the child with lower priority, and do rotation accordingly. If both the children have the same priority, it will choose any one node.

This will happen successively until the node becomes a leaf. It is then deleted.

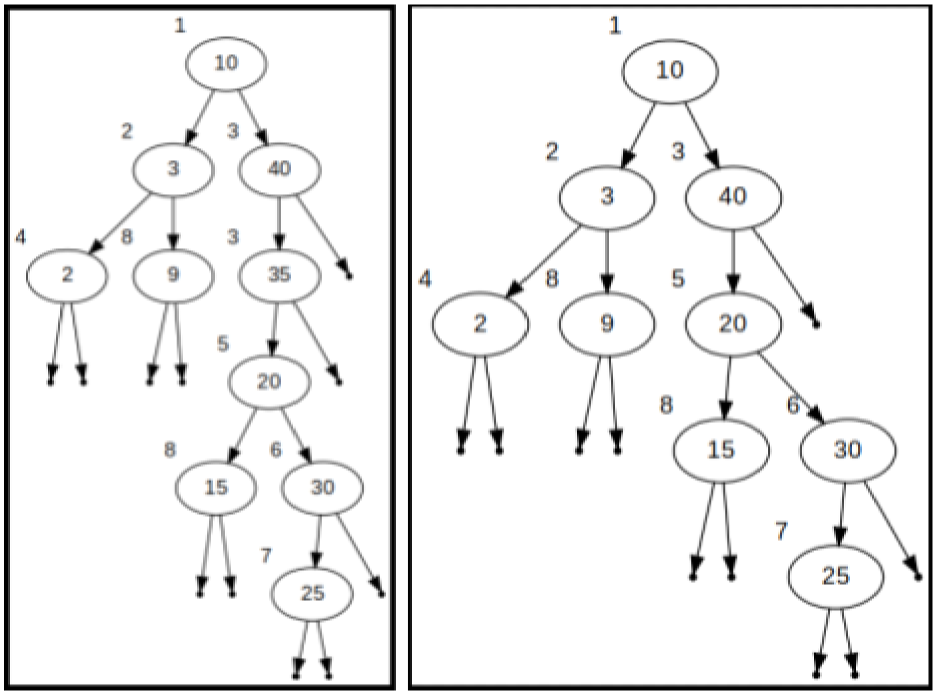
A sample deletion looks as follows:



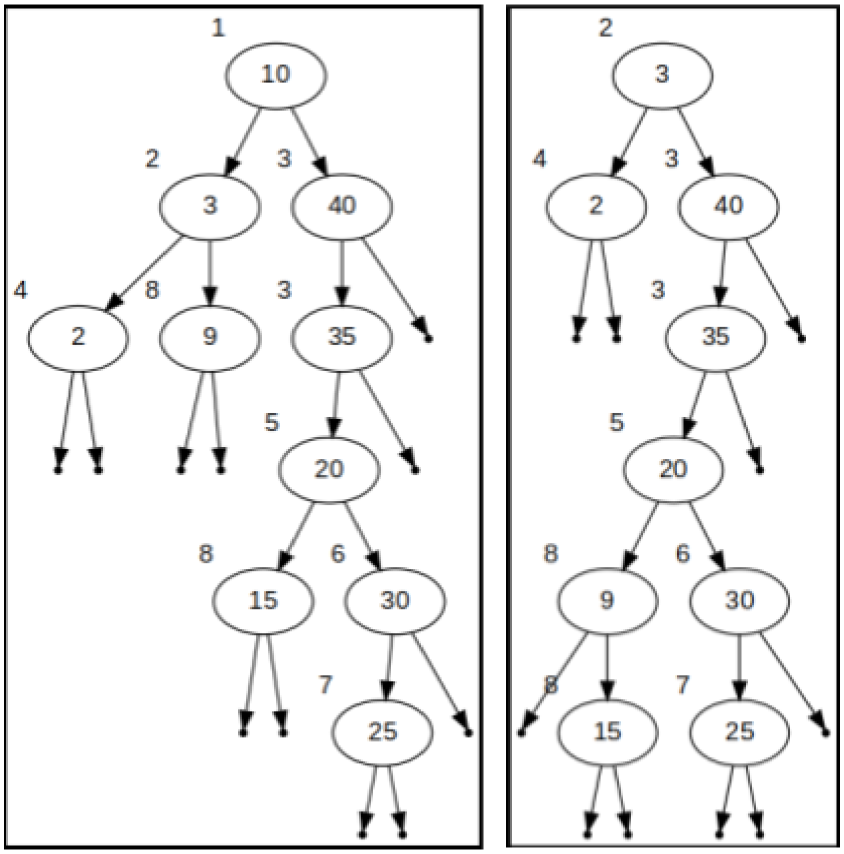
Test Cases:



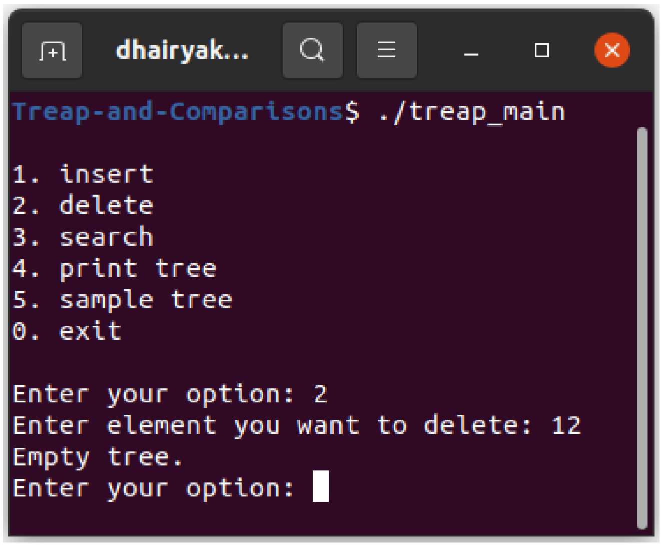
Deletion of a leaf node

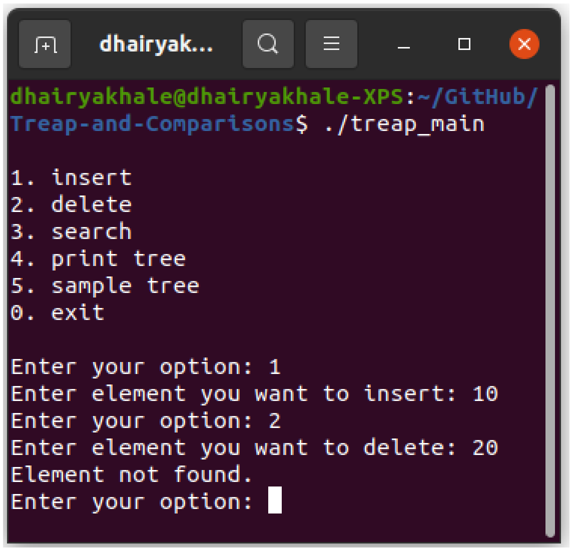


Deletion of a node with only 1 child



Deletion of a node with 2 children


```
Treap-and-Comparisons$ ./treap_main

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 2
Enter element you want to delete: 12
Empty tree.
Enter your option:
```


```
dhairyakhale@dhairyakhale-XPS:~/GitHub/
Treap-and-Comparisons$ ./treap_main

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 1
Enter element you want to insert: 10
Enter your option: 2
Enter element you want to delete: 20
Element not found.
Enter your option:
```

Deletion from an empty tree and of a non-existing node

search_Treap(k)

Logic:

Searching in a Treap is exactly the same as searching in a regular Binary Search Tree.

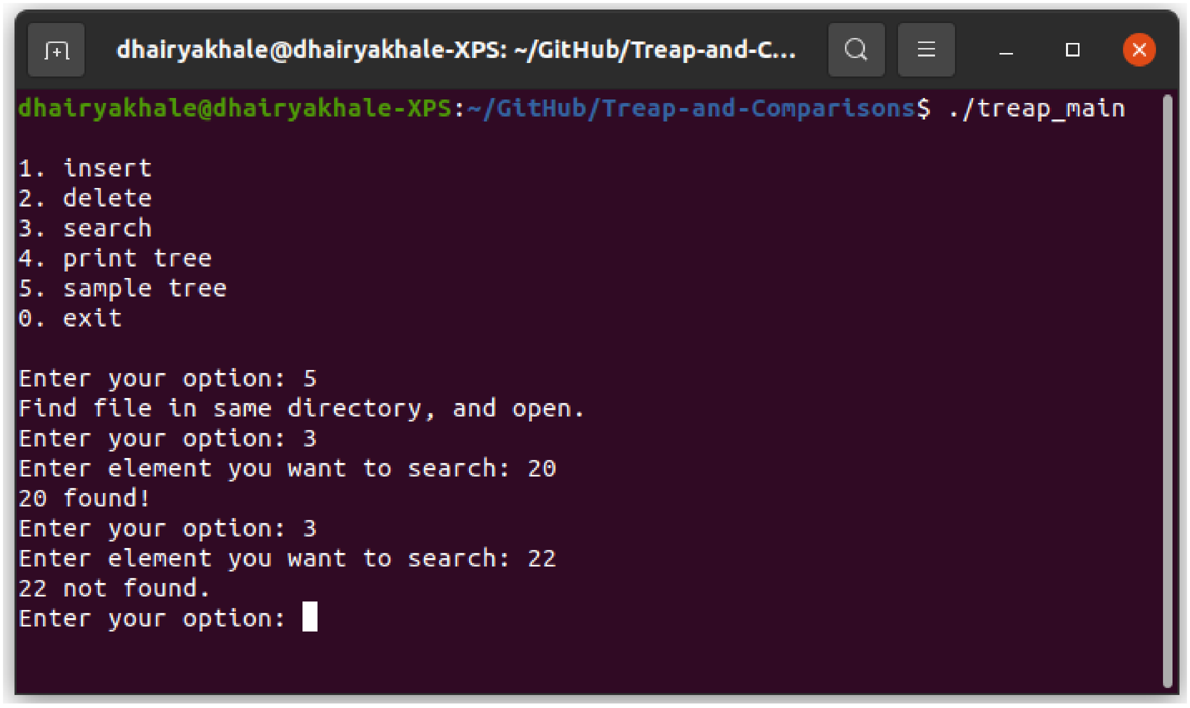
We make a node pointer, and point it to the root node. We then compare key(node) and k.

If k < key(node), node becomes its left child. If k > key(node), node becomes its right child. This process keeps repeating till the node is null.

If k == key(node), that means node is what we were searching for. So we return true, and exit.

If node becomes null, that means the node we were searching for does not exist in the tree. Hence, we return false.

Test Cases:



```
dhairyakhale@dhairyakhale-XPS:~/GitHub/Treap-and-Comparisons$ ./treap_main

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 5
Find file in same directory, and open.
Enter your option: 3
Enter element you want to search: 20
20 found!
Enter your option: 3
Enter element you want to search: 22
22 not found.
Enter your option: 
```

Searching into the sample tree

print_Treap(filename)

Logic:

Here, we needed the use of graphviz software. So we had to basically make a graphviz compatible file by writing data out of the cpp file.

For this, we used 3 functions:

1. **printTree_null**
2. **printTree_main**
3. **print_Treap**
4. **inorderLabel**

The first function is just used to write the command which makes the input argument node connect to a dot through an arrow.

The second function is basically creating all the edges.
It starts from root, and creates all the left edges by looping while the left child is not null. It associated to each destination and source node a label and name, both specified in the inorderLabel function. When the null is reached, it calls the printTree_null() function, where the node is connected to a dot.
Similarly, all the right edges are created by looping while the right child is not null.

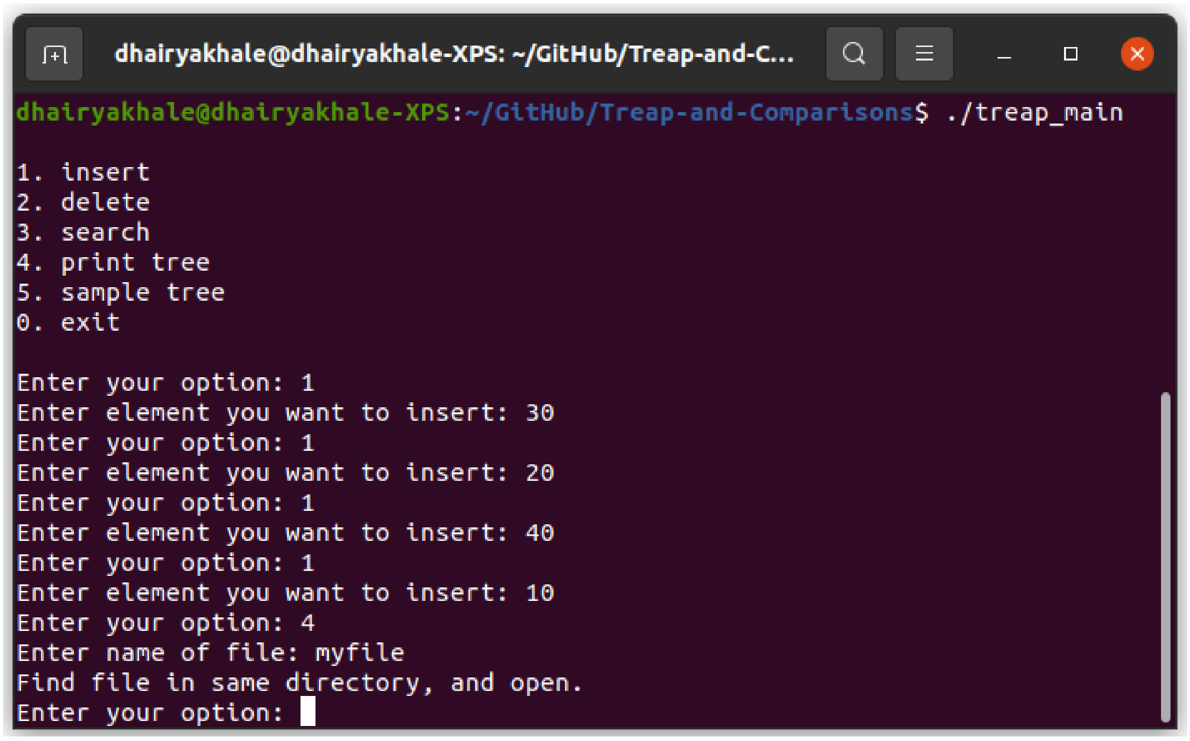
The second function also has a static int nullc. This is passed with post-increment to the printTree_null() function, so that every null node is named uniquely.

The third function is basically a driver function.
It first creates and opens a file in write mode. The name of the file is specified via the argument of the function (filename). It also creates a label associated with each node, which tells the priority of that node.
It then gives some starting commands in the graph file.
If root is null, it does not input anything in the graph file. Else if root has left and right null, it just feeds a single node value into the file. Else, it calls the printTree_main() function.
It then gives ending commands to the file, and closes the file.

The fourth function associates a unique label and unique name to each non-null node. This helps in printing, and is required for purposes of the graphviz software.

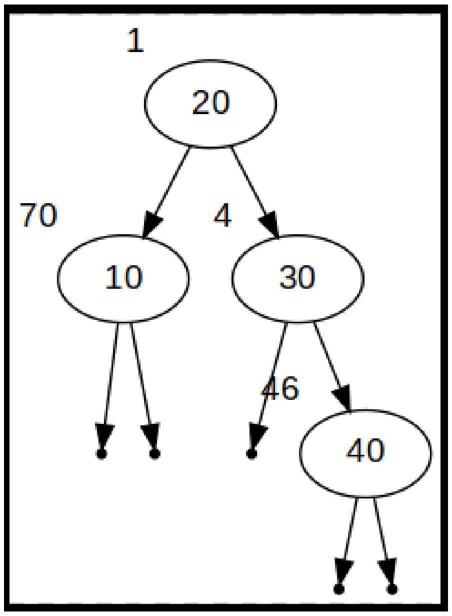
After closing, we open the cmd, convert the .gv file to .svg image, and save the image in the current working directory using the system() function.

<u>Test Cases:</u>

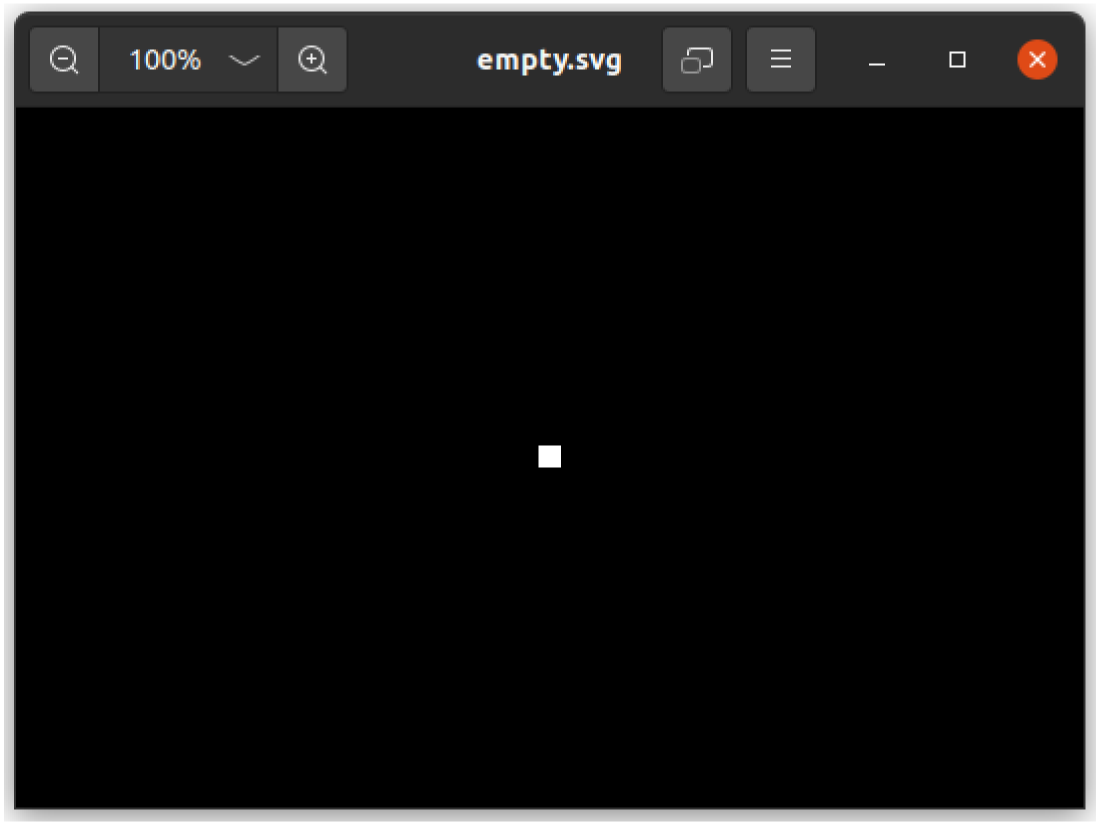


```
dhairyakhale@dhairyakhale-XPS:~/GitHub/Treap-and-Comparisons$ ./treap_main

1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 1
Enter element you want to insert: 30
Enter your option: 1
Enter element you want to insert: 20
Enter your option: 1
Enter element you want to insert: 40
Enter your option: 1
Enter element you want to insert: 10
Enter your option: 4
Enter name of file: myfile
Find file in same directory, and open.
Enter your option: 
```

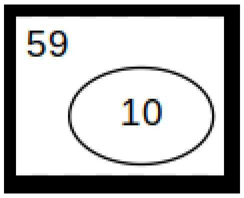
Steps for printing a tree



The output of above commands



Printing an empty tree



A tree with only single node

Comparison between Binary Search Tree, AVL Tree and Treap

Code structure and generation of test cases

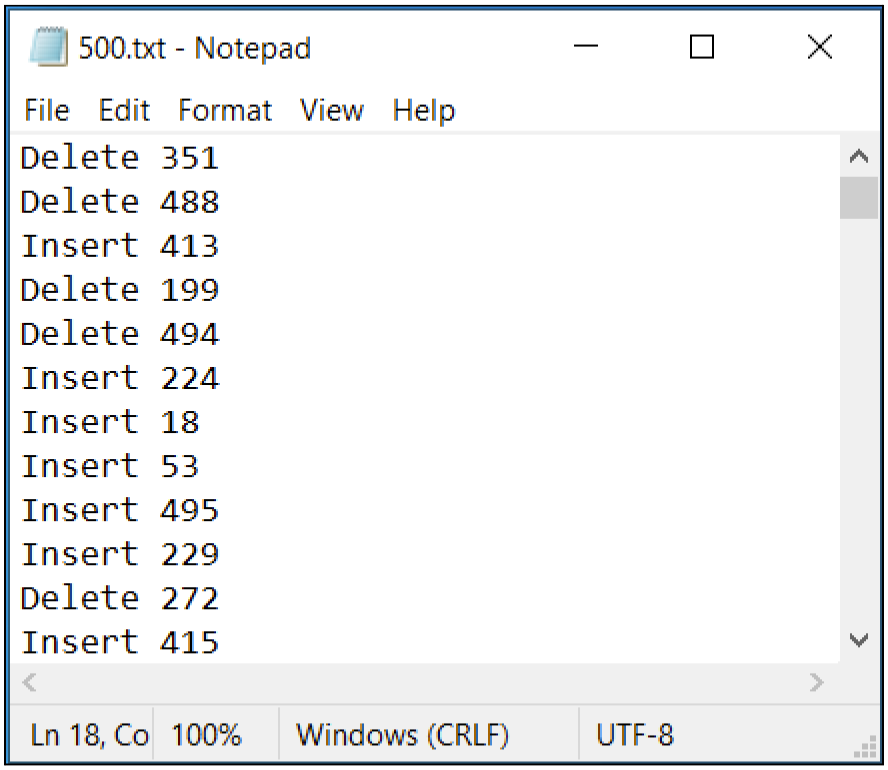
There are four files involved in the comparative study and generation of test cases:

1. **test_drive.cpp:** Creates test cases and calling all 3 trees
2. **bst.h:** Functions for creating BST and recording parameters
3. **avl.h:** Functions for creating AVL and recording parameters
4. **treap.h:** Functions for creating BST and recording parameters

The first cpp file creates a single test case as following:

1. It creates an empty .txt file and opens it.
2. It randomly writes "Insert" or "Delete" followed by a space, and a random number. The range of the random number will be from 1 to no. of rows of file.
3. It maintains a proportion of Insert and Delete to a specified ratio (eg. 70:30).
4. It then closes the file.

There are files created with the number of rows being 500 to 10000, in increments of 500. The files are named as <no. Of rows>.txt (Eg. 500.txt, 1000.txt,...) and are stored in the folder "Test_Files". We have used 70:30, 60:40, and 50:50 as the ratios of "Insert" and "Delete" operations.


500.txt - Notepad
File Edit Format View Help
Ln 18, Co 100% Windows (CRLF) UTF-8

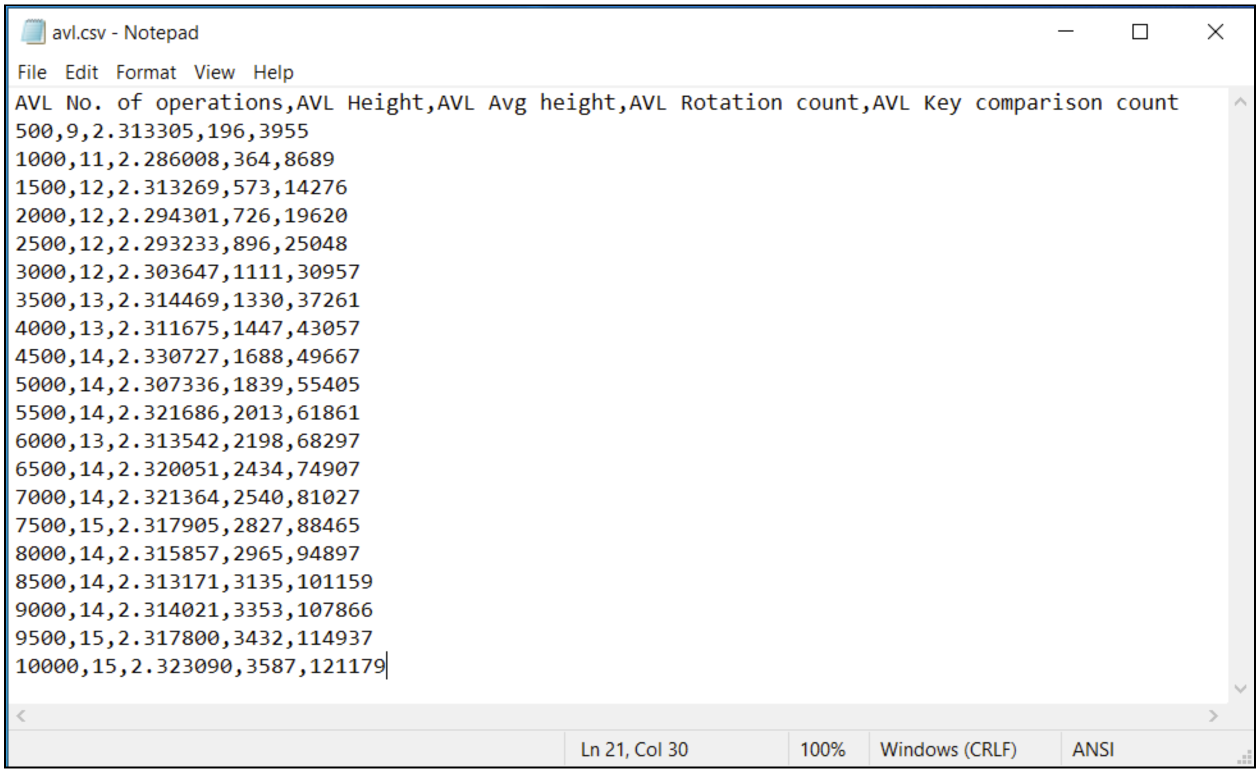
```
Delete 351
Delete 488
Insert 413
Delete 199
Delete 494
Insert 224
Insert 18
Insert 53
Insert 495
Insert 229
Delete 272
Insert 415
```

A sample test file

The three header files include functions which return the parameters which are described further in the document. They are:

- Height of the tree
- Average height of the tree
- Total key comparisons
- Total number of rotations

In each of the header files, there is a function with the nomenclature <tree name>_main (**BST_main()**, **AVL_main()**, **Treap_main()**). The job of these functions is to take a test case file as input, read each line of it and perform insertion/deletion in their respective trees accordingly. After that, it inputs the parameters named above, in their respective .csv files which they have created. These .csv files are stored in folder “Analysis”.



```
AVL No. of operations,AVL Height,AVL Avg height,AVL Rotation count,AVL Key comparison count
500,9,2.313305,196,3955
1000,11,2.286008,364,8689
1500,12,2.313269,573,14276
2000,12,2.294301,726,19620
2500,12,2.293233,896,25048
3000,12,2.303647,1111,30957
3500,13,2.314469,1330,37261
4000,13,2.311675,1447,43057
4500,14,2.330727,1688,49667
5000,14,2.307336,1839,55405
5500,14,2.321686,2013,61861
6000,13,2.313542,2198,68297
6500,14,2.320051,2434,74907
7000,14,2.321364,2540,81027
7500,15,2.317905,2827,88465
8000,14,2.315857,2965,94897
8500,14,2.313171,3135,101159
9000,14,2.314021,3353,107866
9500,15,2.317800,3432,114937
10000,15,2.323090,3587,121179
```

A sample .csv file generated

Height

Implementation

We have defined a function **height()** in all the three header files, which recursively compares the height in left subtree and right subtree of the node passed to it, and gives the maximum of them in each iteration.

This height function is called by another function **fn_height()** which then returns the height of the entire tree.

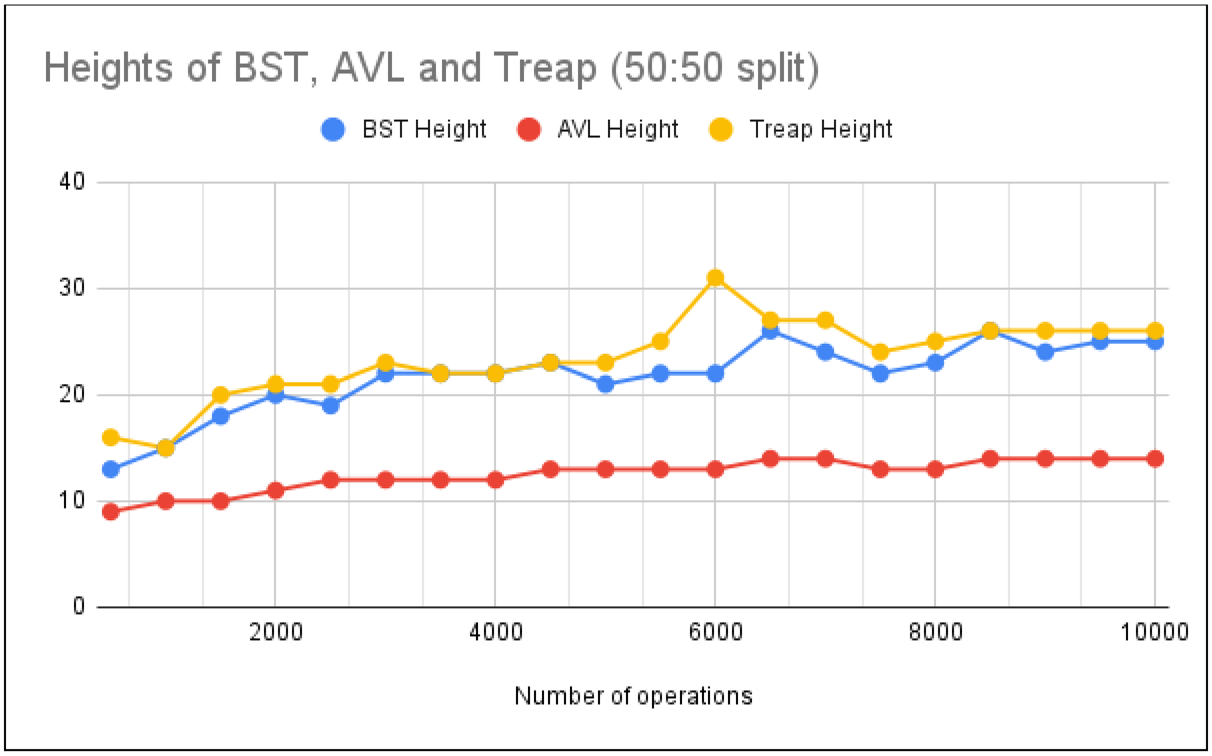
Theory:

Height of a tree is the maximum distance from the root of the tree to one of its leaves.

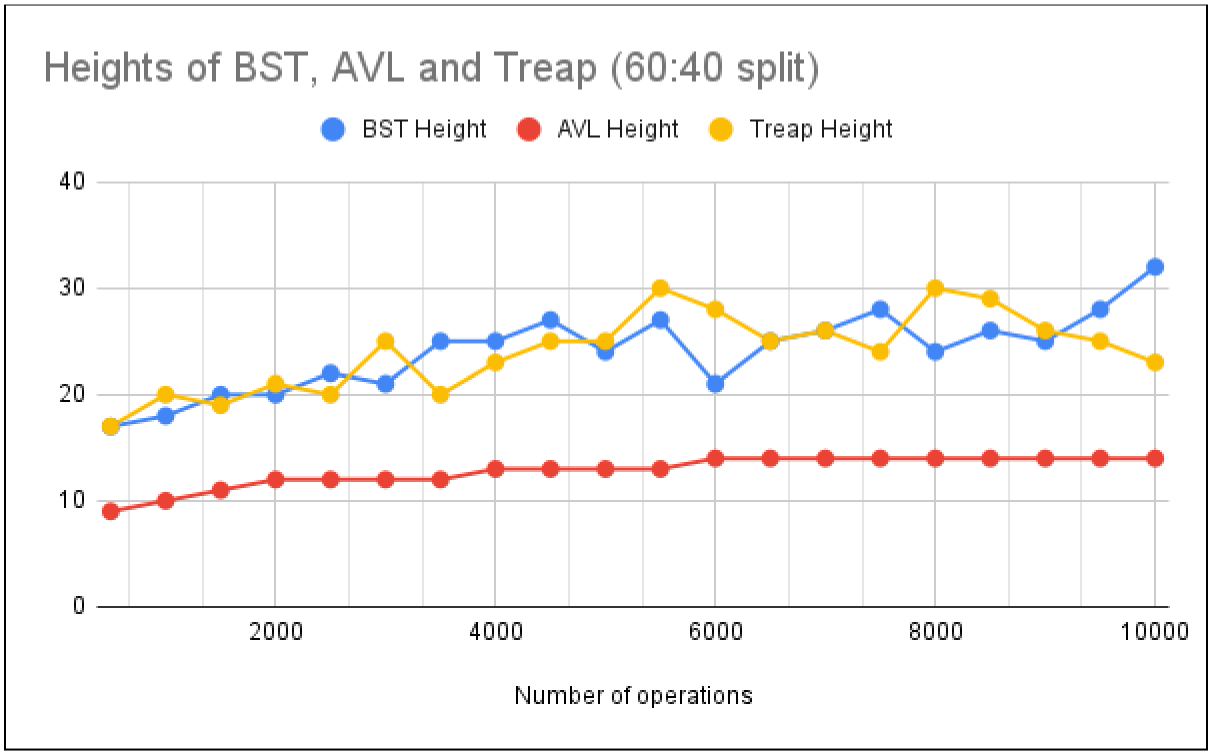
In an ideal condition, i.e. a perfectly balanced tree, the height should be logN, where N is the number of nodes in the tree. However, a treap and BST are less than ideal situations, as they are not guaranteed to be balanced. However, an AVL Tree is most certainly balanced (or off-balance by only 1), as it is self balancing hence balances itself at every insert and delete operation.

Overall, we expect Treap and BST to have similar and much larger heights than AVL. Between them, we expect Treap to have slightly less heights.

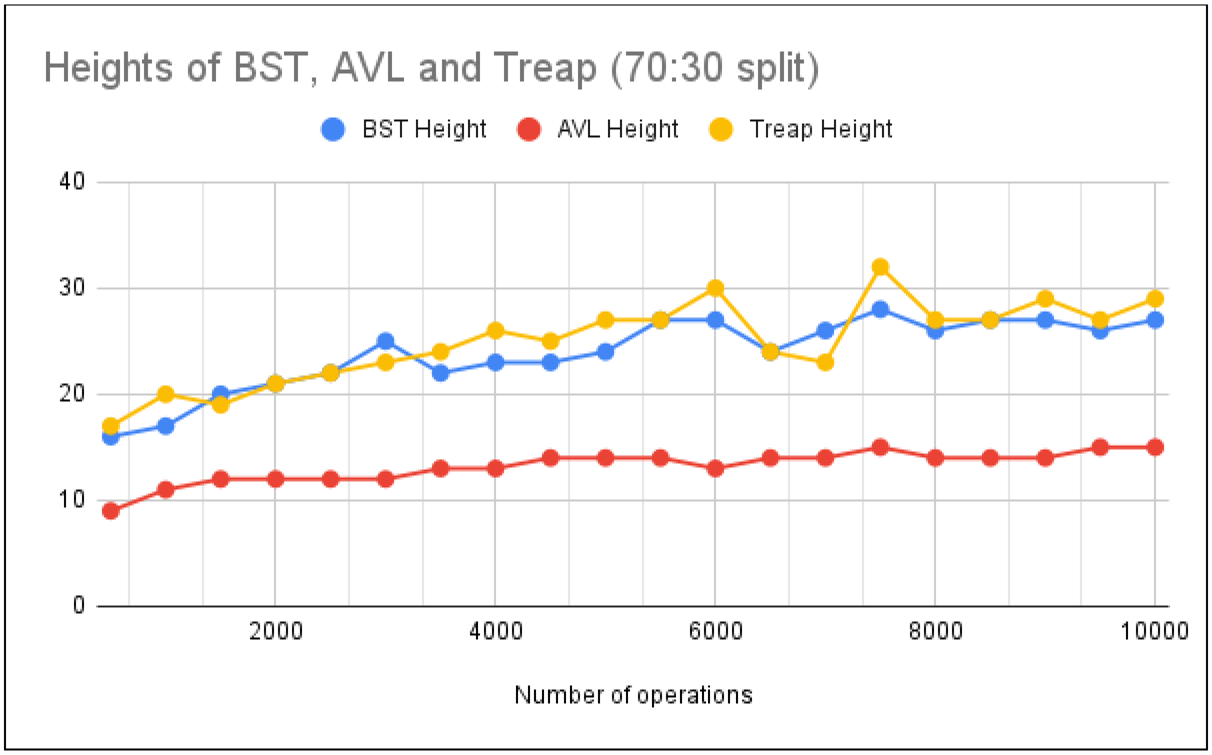
Results

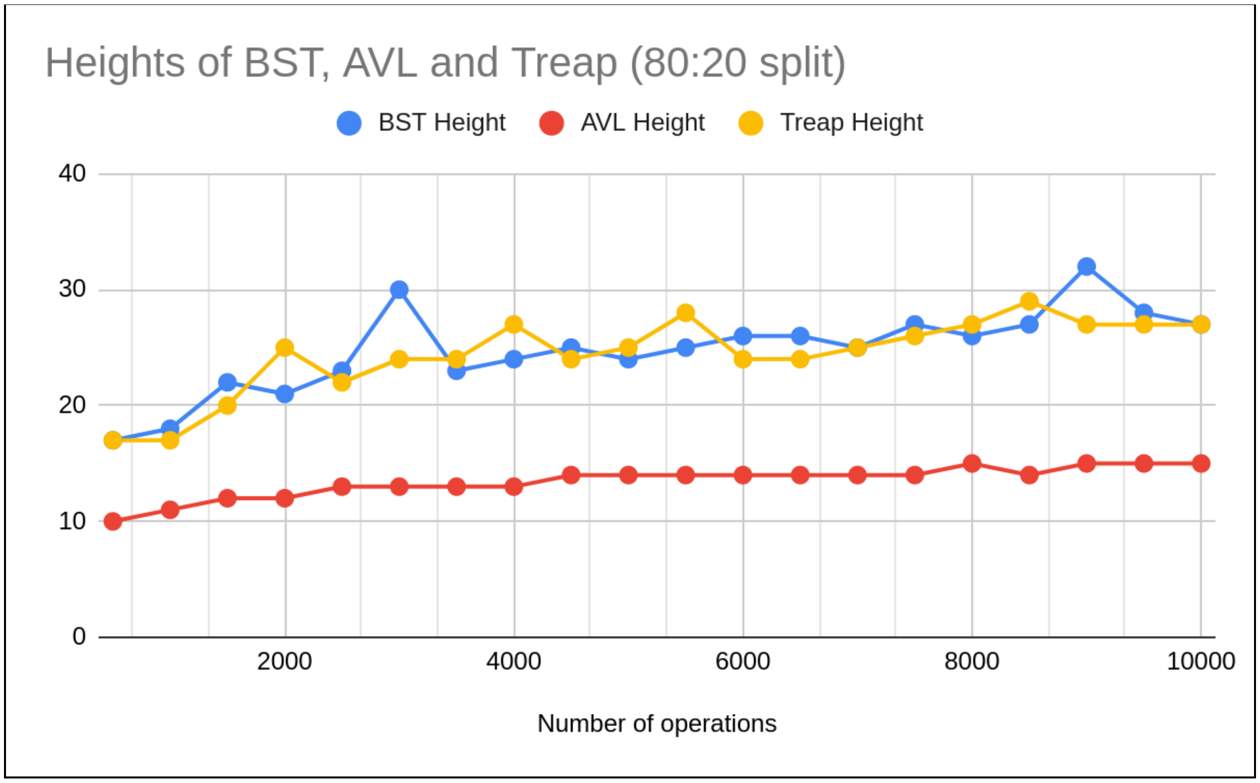


Heights of BST, AVL and Treap (50:50 split)
BST Height
AVL Height
Treap Height
40
30
20
10
0
2000
4000
6000
8000
10000
Number of operations



Heights of BST, AVL and Treap (60:40 split)
BST Height
AVL Height
Treap Height
40
30
20
10
0
2000
4000
6000
8000
10000
Number of operations


Heights of BST, AVL and Treap (70:30 split)
BST Height
AVL Height
Treap Height
40
30
20
10
0
2000
4000
6000
8000
10000
Number of operations


Heights of BST, AVL and Treap (80:20 split)
BST Height
AVL Height
Treap Height
40
30
20
10
0
2000
4000
6000
8000
10000
Number of operations

Average height of the tree

Implementation

To calculate this, we have created a function **avg_height()** which calls the function **height()** discussed previously in a manner of inorder traversal. This means that the function goes to every node in an inorder fashion, and calculates the height of each node reached.

This function is called by another function **fn_avg_height()** which returns the average height for the entire tree.

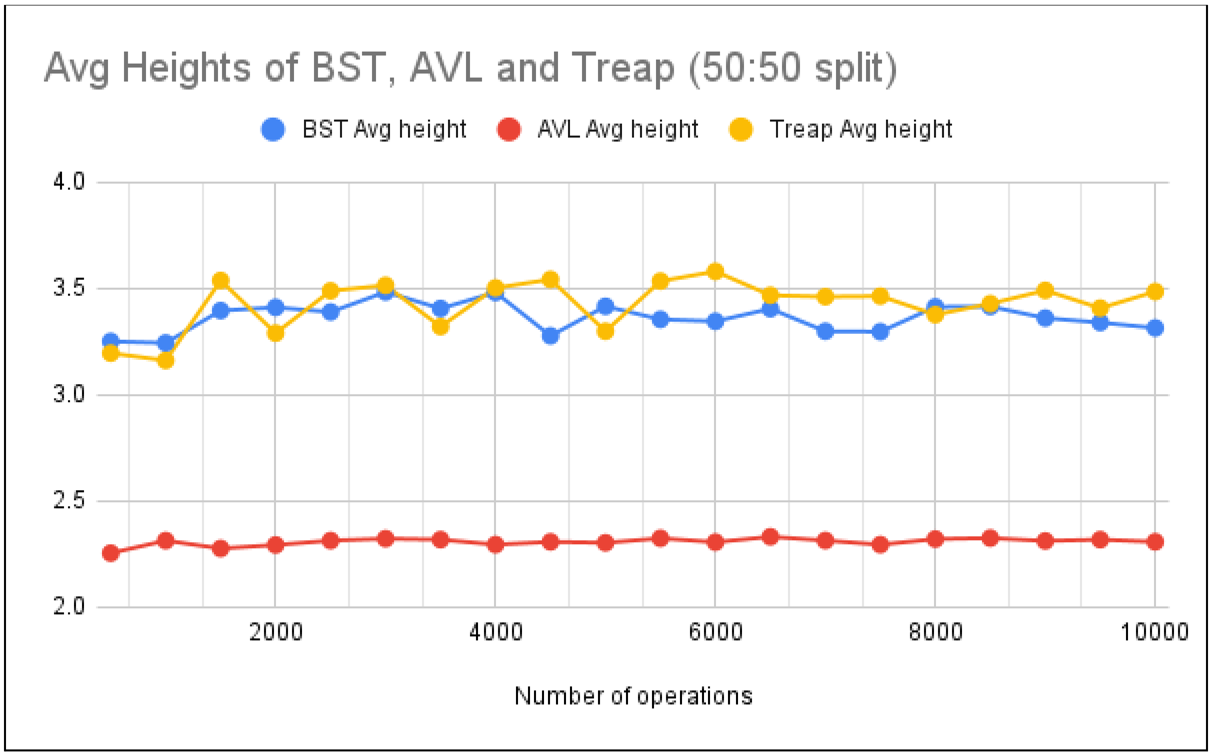
Theory

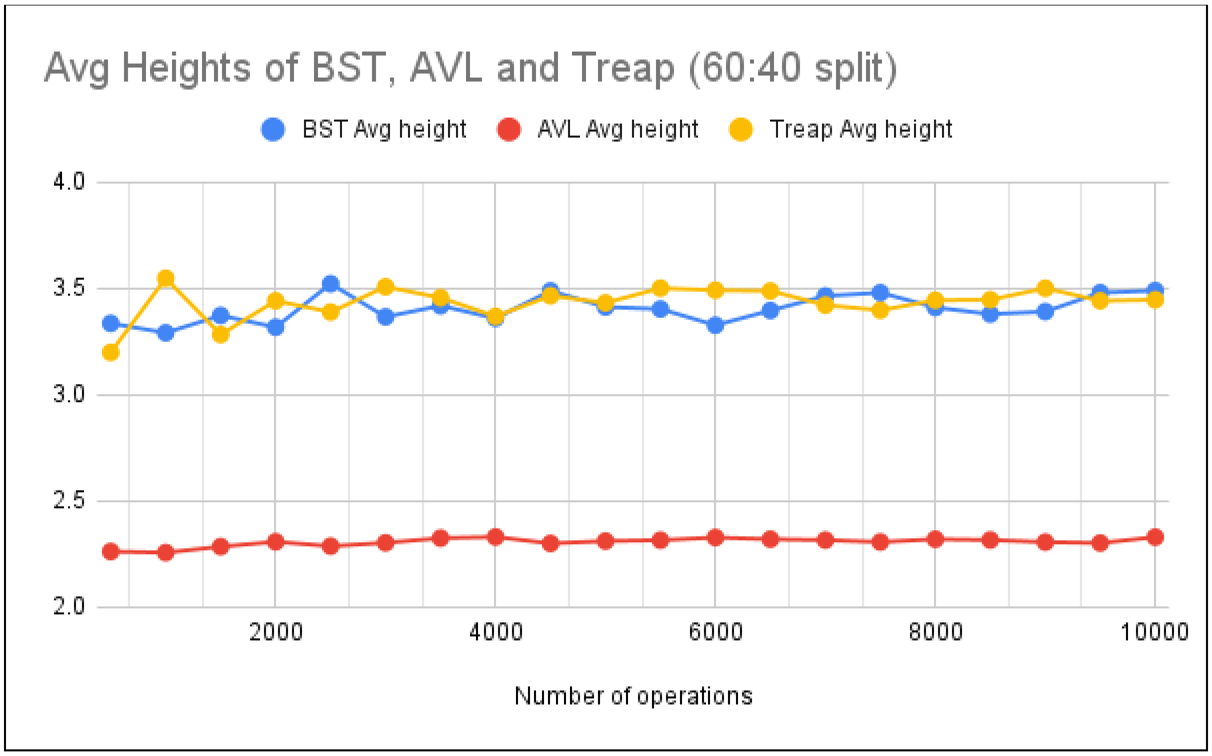
The formula for calculating average height of the tree is as follows:

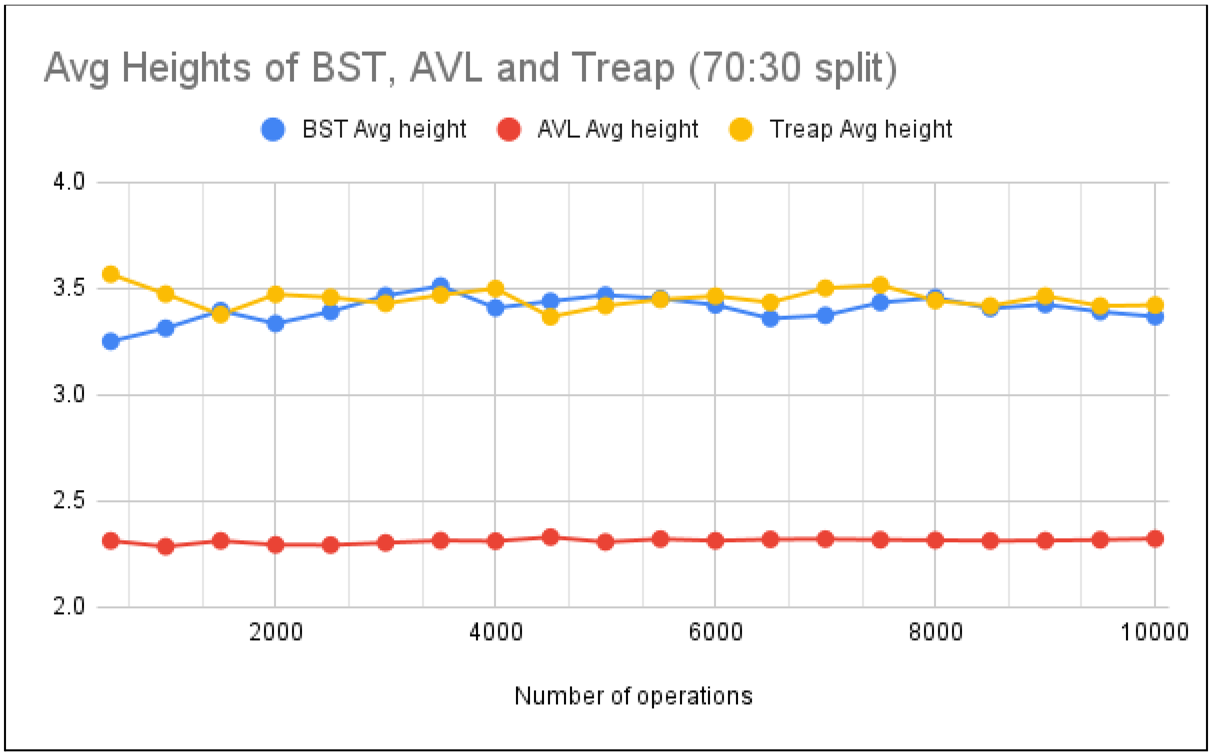
$$average\ height\ = \frac{\Sigma(height\ of\ nodes)}{number\ of\ nodes}$$

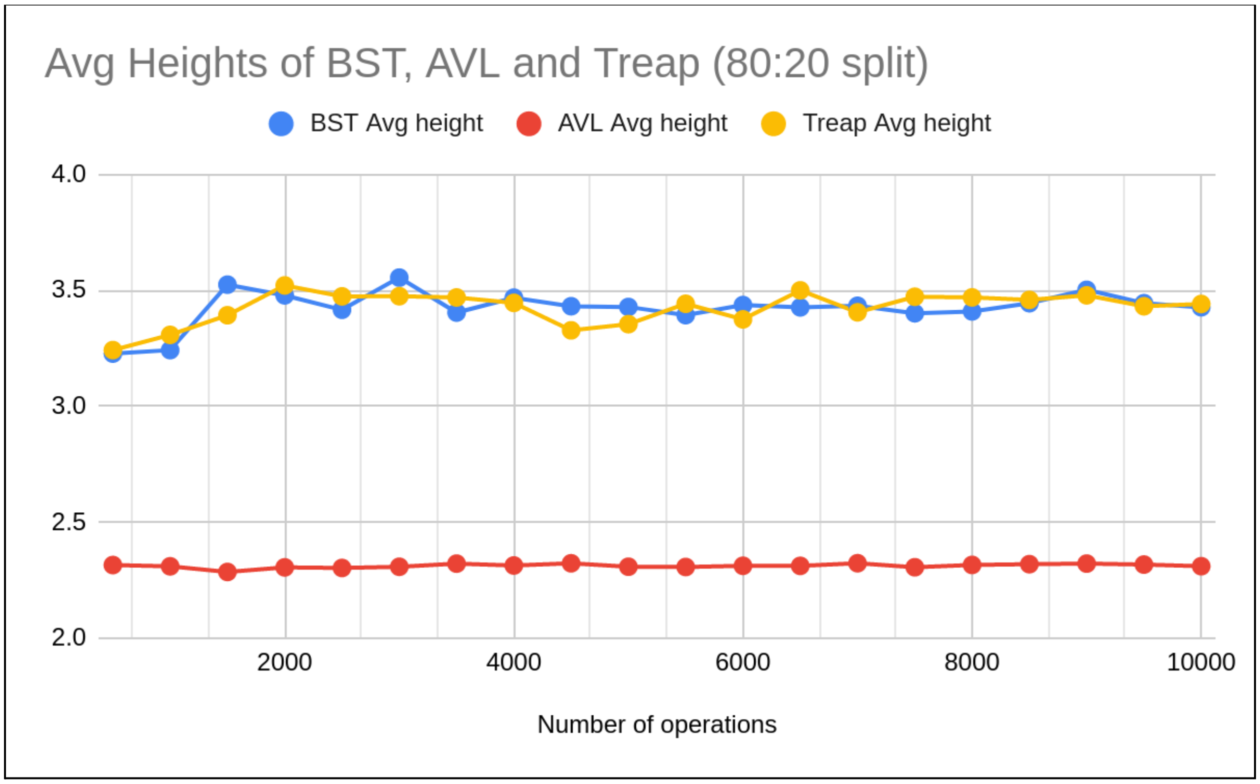
Just like heights, we expect the average heights of BST and Treap to be similar, with treap having slightly less average height. AVL will have marginally less average height than both BST and Treap, because it is balanced and every node's height is at its minimum.

Results


Avg Heights of BST, AVL and Treap (50:50 split)
BST Avg height
AVL Avg height
Treap Avg height
4.0
3.5
3.0
2.5
2.0
2000
4000
6000
8000
10000
Number of operations


Avg Heights of BST, AVL and Treap (60:40 split)
BST Avg height
AVL Avg height
Treap Avg height
4.0
3.5
3.0
2.5
2.0
2000
4000
6000
8000
10000
Number of operations


Avg Heights of BST, AVL and Treap (70:30 split)
BST Avg height
AVL Avg height
Treap Avg height
4.0
3.5
3.0
2.5
2.0
2000
4000
6000
8000
10000
Number of operations


Avg Heights of BST, AVL and Treap (80:20 split)
BST Avg height
AVL Avg height
Treap Avg height
4.0
3.5
3.0
2.5
2.0
2000
4000
6000
8000
10000
Number of operations

Total key comparisons

Implementation

We have created a private variable of type int named **key_comparison** in each class of the respective tree. Whenever in insertion or deletion a key comparison occurs, the variable is incremented.

Since it is a private variable, we have created a function **fn_key_comparison()** which returns the value of this variable.

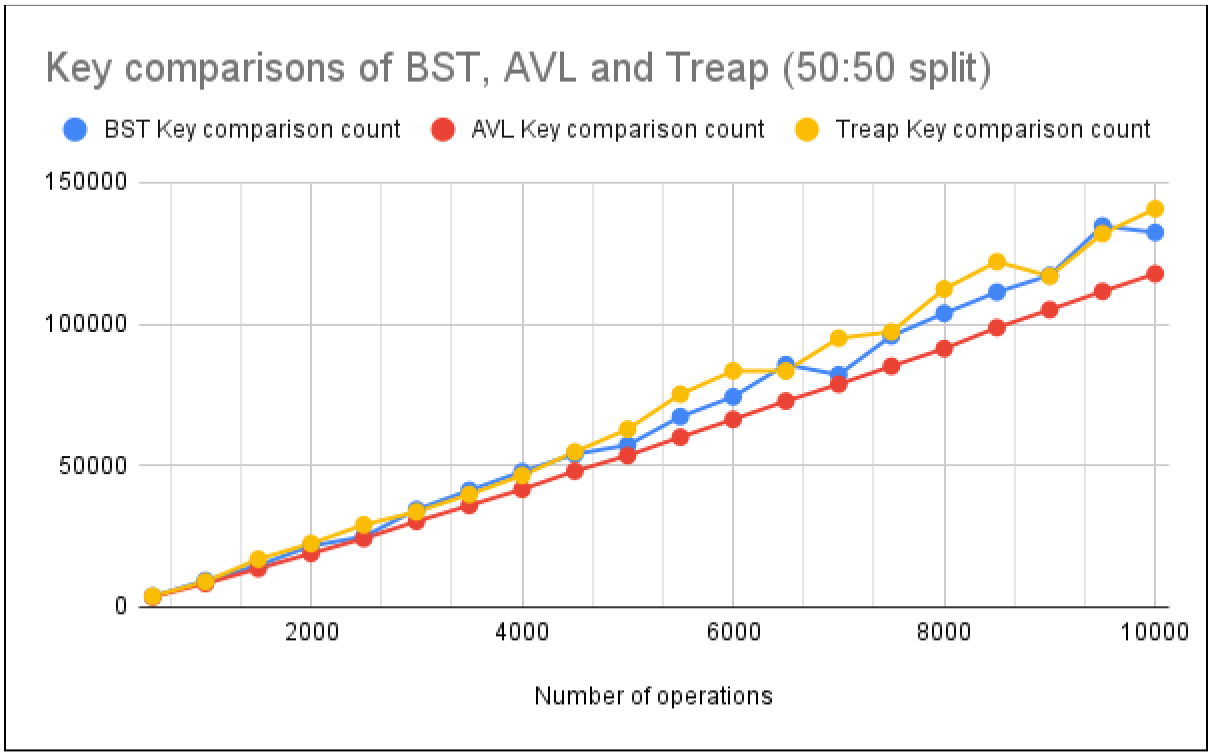
Theory

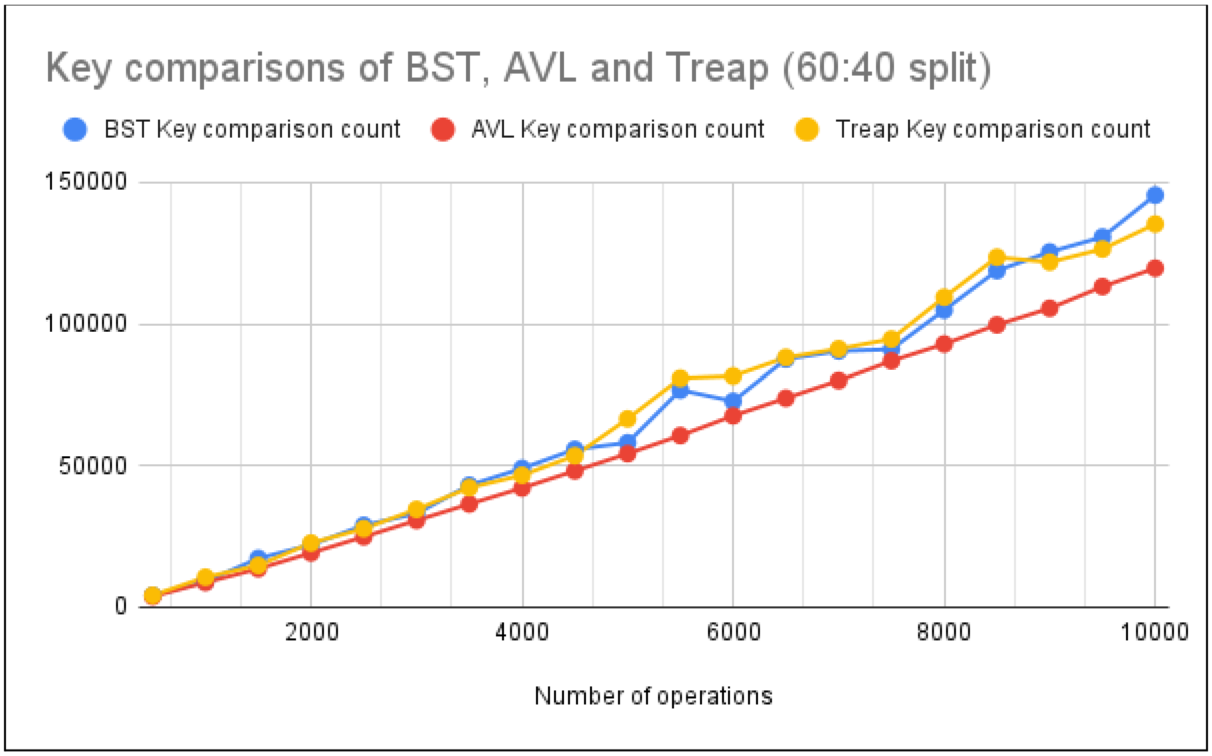
Whenever we insert a node in for eg. a BST, we check if the key value is less than or greater than the value of the current node. If it is lesser, we traverse to the left subtree. If it is greater, we traverse to the right subtree.

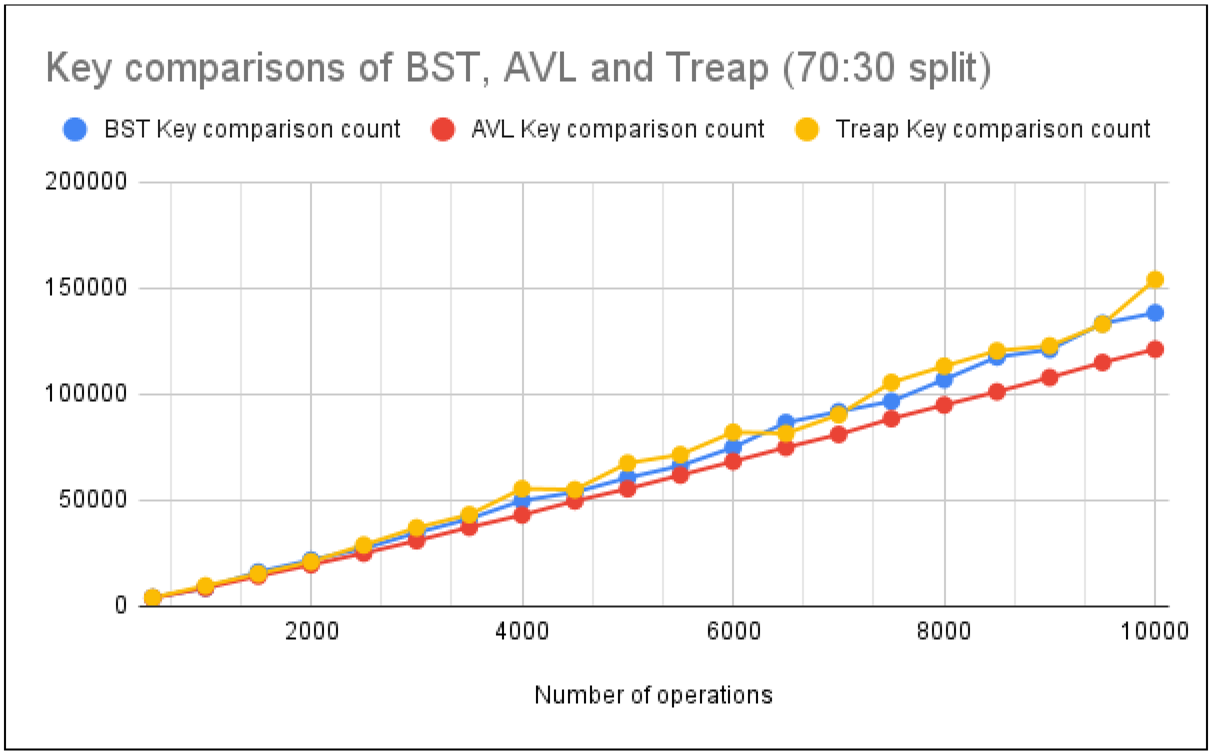
Apart from Insertion, similar happens while deletion. In AVL, we also require key comparisons to balance the tree.

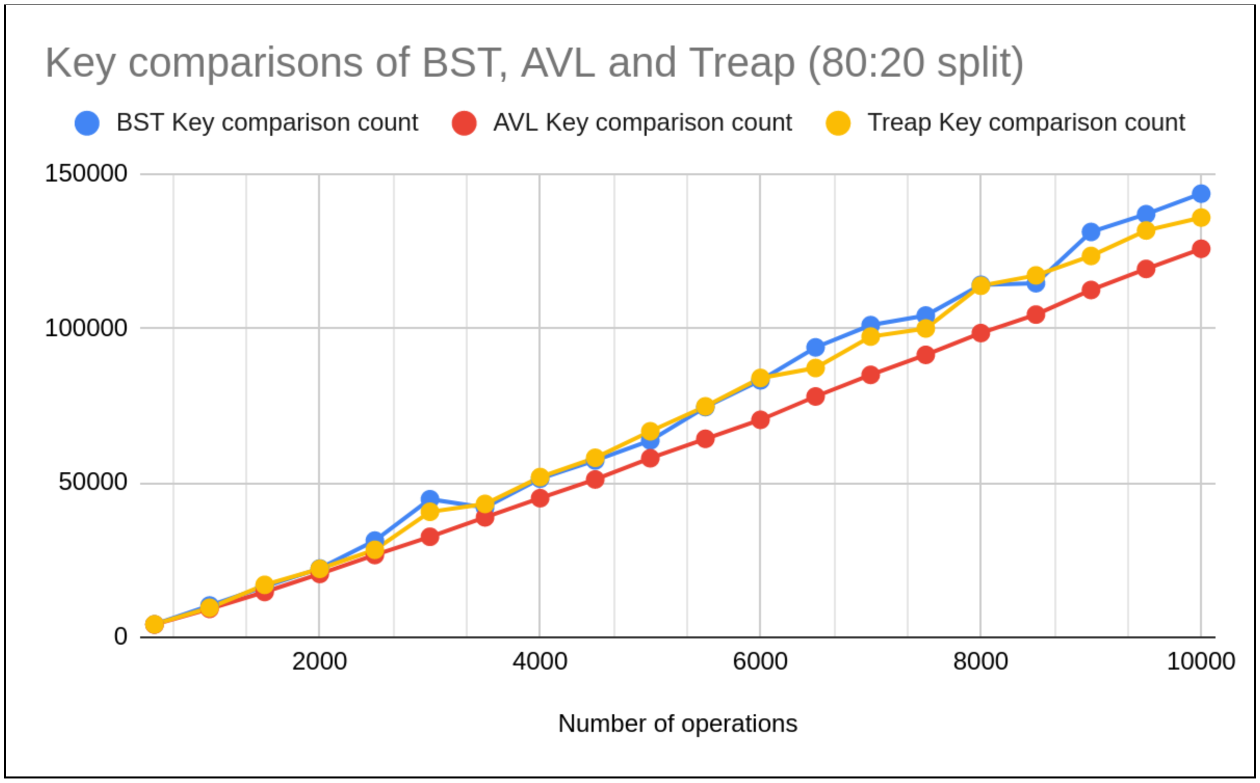
We are expecting the key comparisons of all the trees to be similar to each other, and no major difference is supposed to exist.

Results


Key comparisons of BST, AVL and Treap (50:50 split)
● BST Key comparison count
● AVL Key comparison count
● Treap Key comparison count
150000
100000
50000
0
2000
4000
6000
8000
10000
Number of operations


Key comparisons of BST, AVL and Treap (60:40 split)
● BST Key comparison count
● AVL Key comparison count
● Treap Key comparison count
150000
100000
50000
0
2000
4000
6000
8000
10000
Number of operations


Key comparisons of BST, AVL and Treap (70:30 split)
BST Key comparison count
AVL Key comparison count
Treap Key comparison count
200000
150000
100000
50000
0
2000
4000
6000
8000
10000
Number of operations


Key comparisons of BST, AVL and Treap (80:20 split)
BST Key comparison count
AVL Key comparison count
Treap Key comparison count
150000
100000
50000
0
2000
4000
6000
8000
10000
Number of operations

Total number of rotations

Implementation

We have created a private variable of type int named **rotation_count** in each class of the respective tree. Whenever a rotation occurs in the operations of AVL and Treap, the variable is incremented.

Since it is a private variable, we have created a function **fn_rotation_count()** which returns the value of this variable.

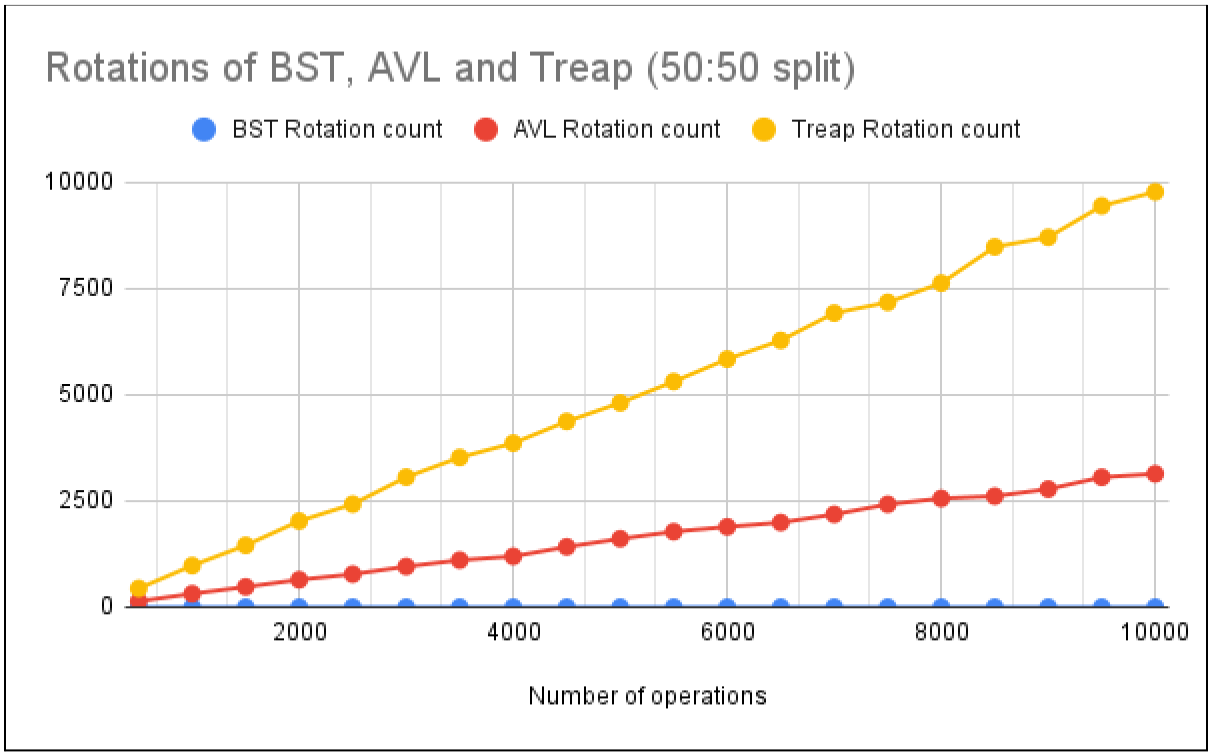
Theory

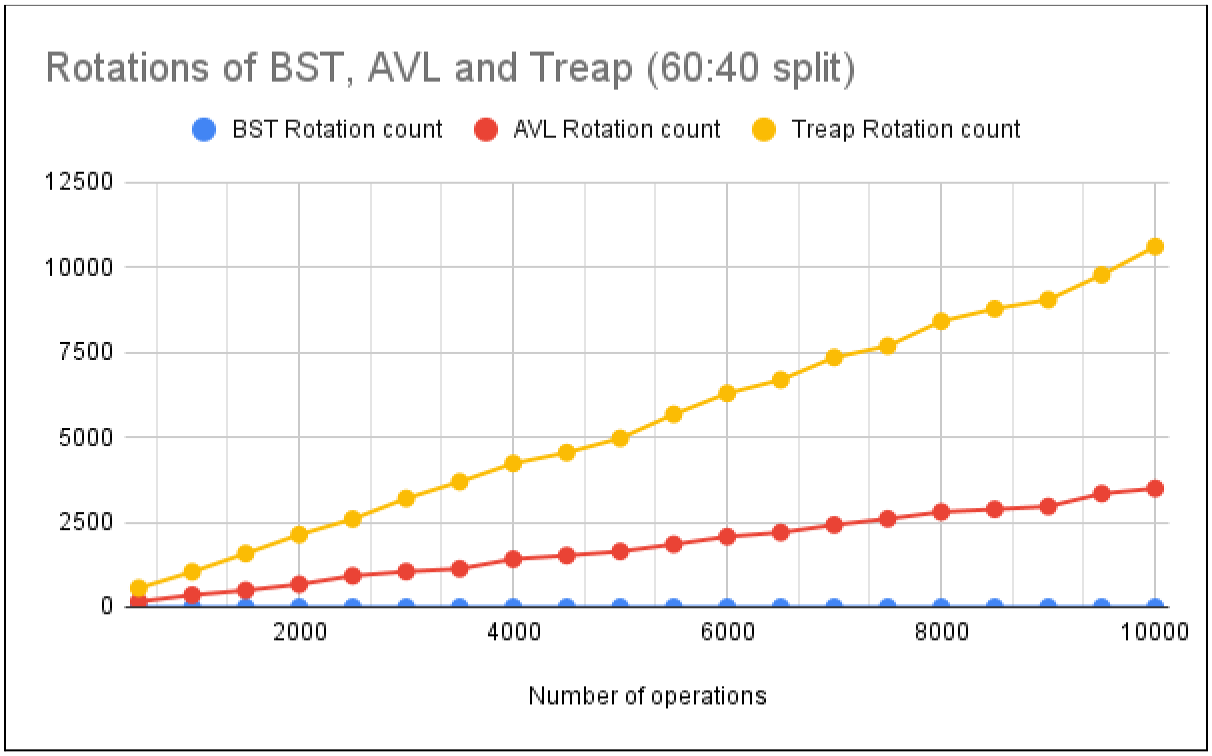
In AVL trees, whenever an operation causes imbalance in the tree (difference between heights of left and right subtree of the node exceeds 1), it rotates according to the present imbalance, and hence balances itself. Sometimes it also does a double rotation, so that is counted accordingly.

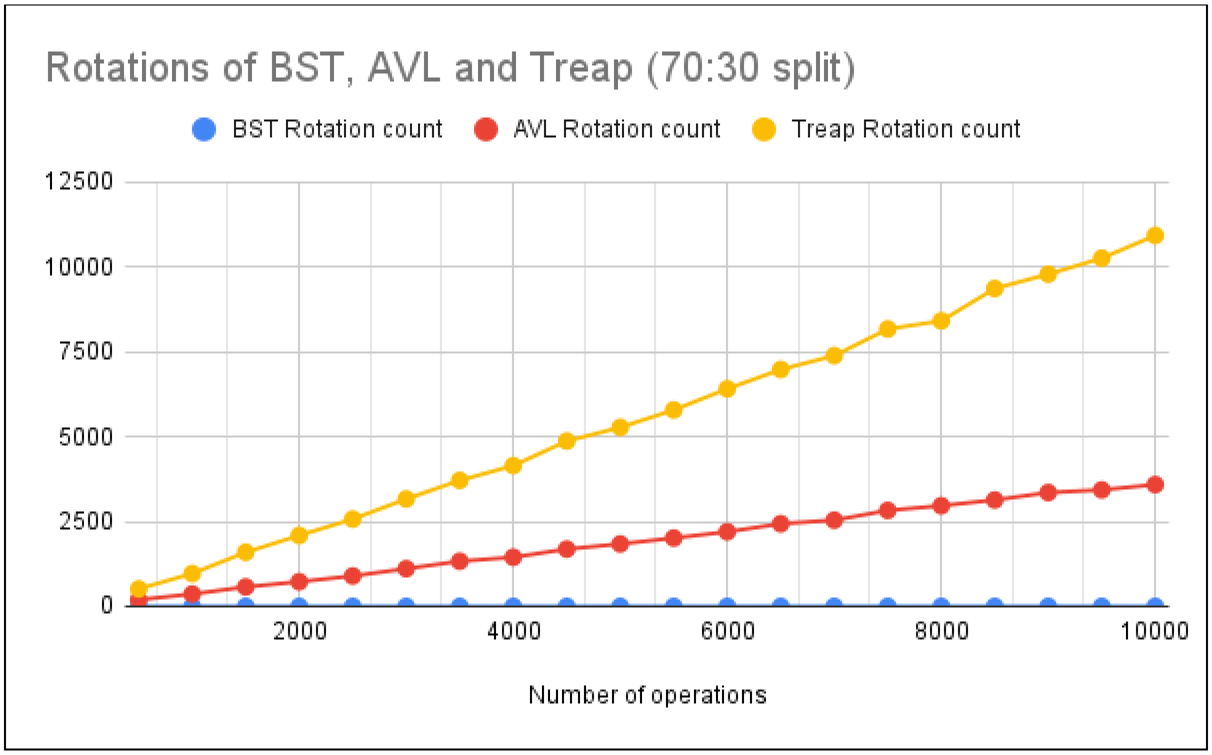
Treap does rotation every time a non-leaf node is inserted or deleted from it. So it has a marginally high number of rotations. The number of rotations can be reduced by increasing the numbers from which priorities are awarded (currently it is set at 100), but even then it will have a much higher count of rotations.

Since BST always inserts at the leaf, and never self-balances, it has no rotations.

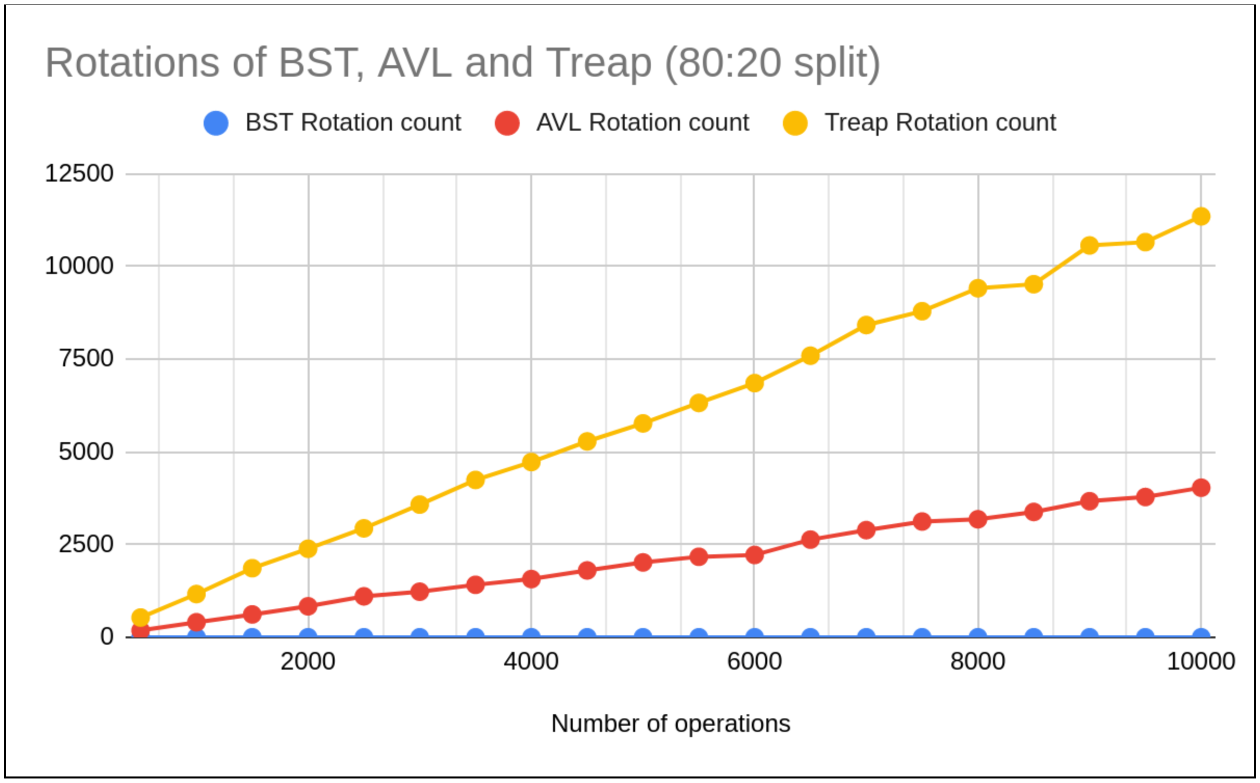
Results


Rotations of BST, AVL and Treap (50:50 split)
BST Rotation count
AVL Rotation count
Treap Rotation count
10000
7500
5000
2500
0
2000
4000
6000
8000
10000
Number of operations


Rotations of BST, AVL and Treap (60:40 split)
BST Rotation count
AVL Rotation count
Treap Rotation count
12500
10000
7500
5000
2500
0
2000
4000
6000
8000
10000
Number of operations



Rotations of BST, AVL and Treap (70:30 split)
BST Rotation count
AVL Rotation count
Treap Rotation count
12500
10000
7500
5000
2500
0
2000
4000
6000
8000
10000
Number of operations

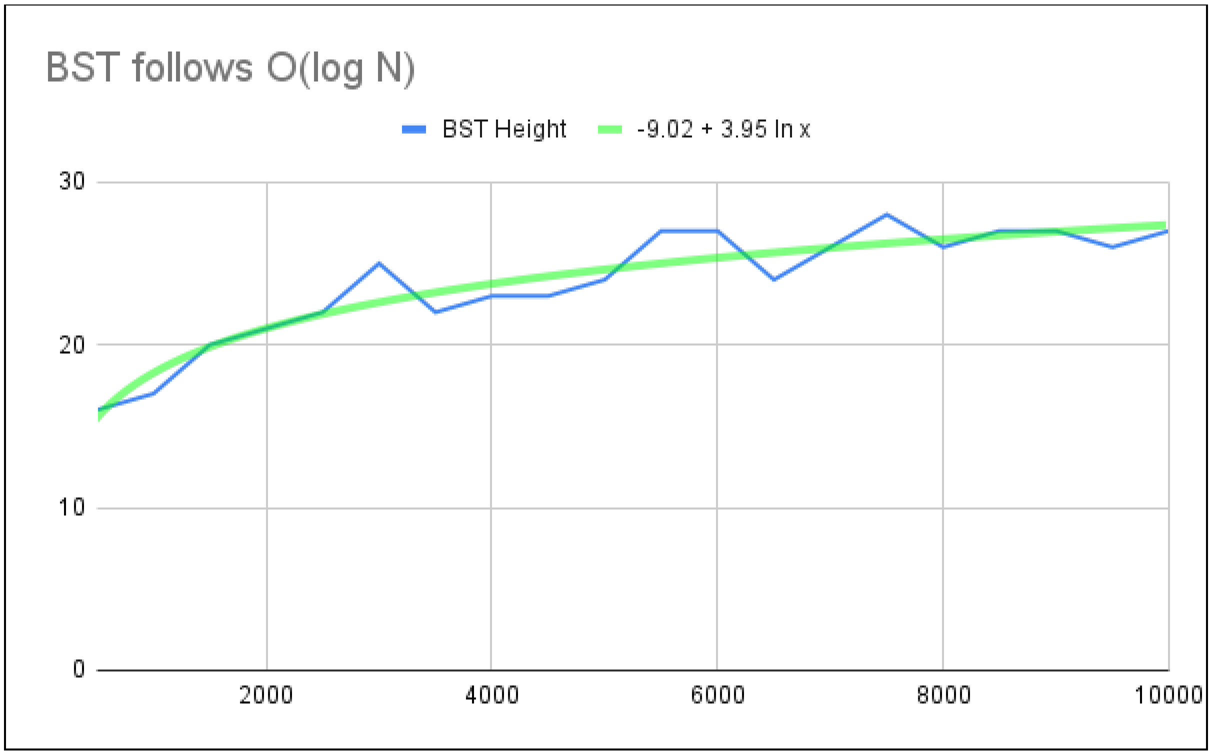


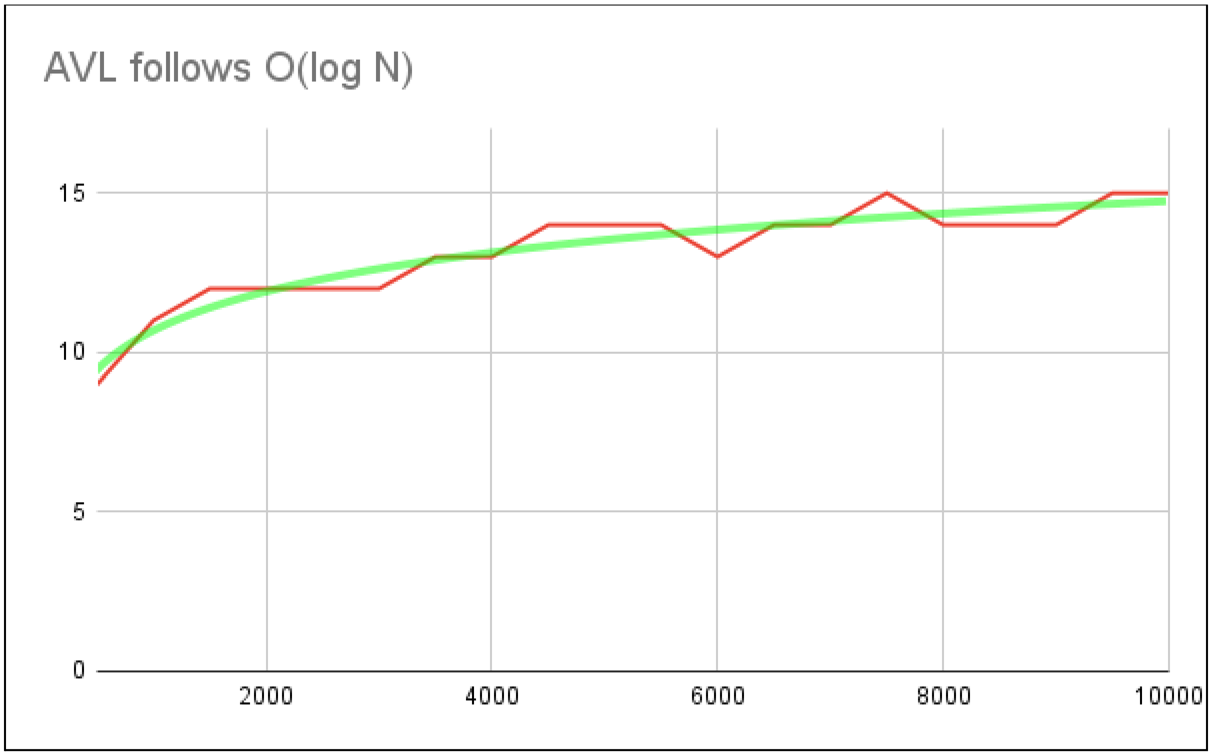
Rotations of BST, AVL and Treap (80:20 split)
BST Rotation count
AVL Rotation count
Treap Rotation count
12500
10000
7500
5000
2500
0
2000
4000
6000
8000
10000
Number of operations

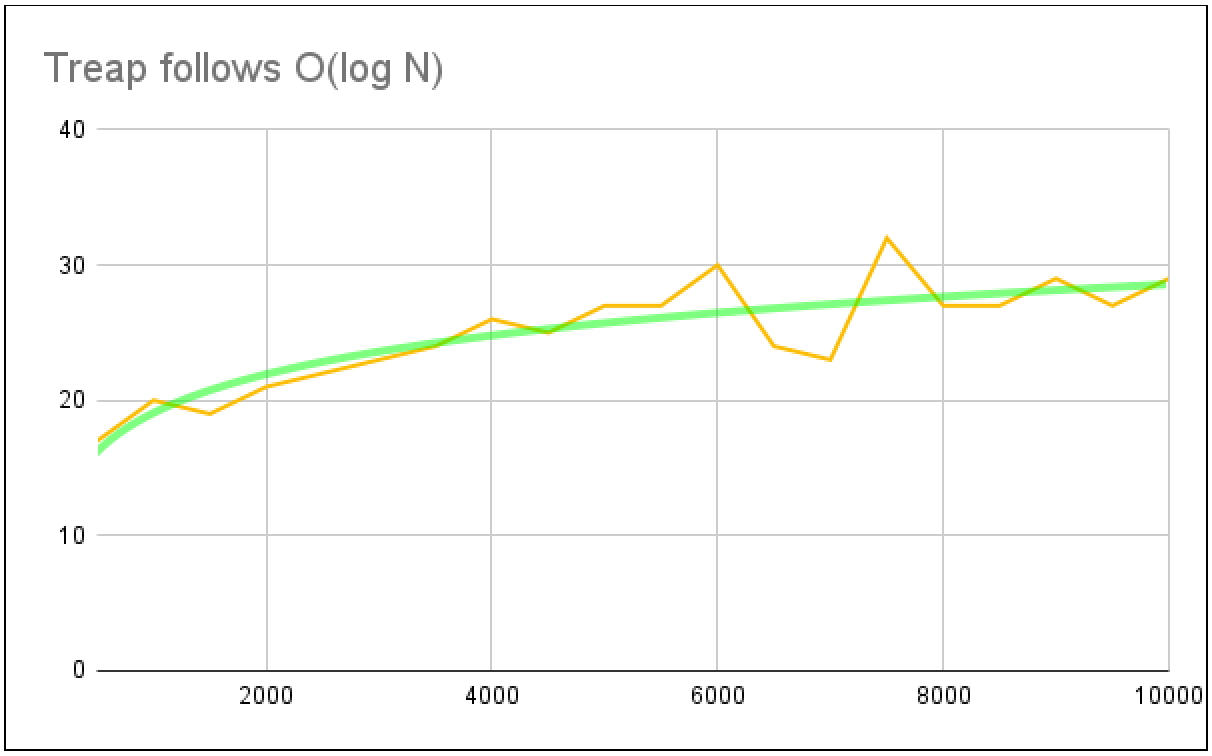
Conclusions

From the above analysis, we draw the following conclusions:

- The size of the treap is highly dependent on the nature of priorities assigned to the nodes, as that is the factor which distinguishes it from being a BST.
  If the priorities allowed to be assigned are large in amount, the tree has more chance to be smaller in size, although that is not a guarantee.
- The treap is extremely similar to BST, both in size and in behaviour. It can also be called a priority based BST.
- The worst case of Treap is not similar to the worst case of BST, as in treap, the random priority balances the tree a little bit because of the introduction of heap property.
- The average height of AVL is least, as expected. The average heights of Treap are closer to that of BST.
- There are no rotations in BST, and Treap has a large amount of rotations because in every insert and delete of a non-leaf node, it rotates repeatedly until the heap condition for priority is satisfied.
- The number of comparisons of all the three trees is more or less similar. One would expect that AVL has marginally lesser comparisons due to its less height, but that gets compensated while rotation and other operations.
- We saw that the height of the Treap is similar to the height of BST, and AVL has a much lower height for each iteration, because it is self-balancing. So its height is much closer to logN.
- We can fit a curve of order logN in all the above trees including BST and AVL, so the order is still O(log N) for Treap, although AVL is much closer to logN than others.


BST follows O(log N)
BST Height
-9.02 + 3.95 ln x
30
20
10
0
2000
4000
6000
8000
10000


AVL follows O(log N)
15
10
5
0
2000
4000
6000
8000
10000



Hence, we have empirically verified the theoretical results, and visually depicted them using graphs.