# Data Structures Lab

## Assignment - 03

Sanjeev Kumar ( 214101048 )

## **Treap Data Structure Implementation**

## Introduction :

- The Assignment 03 is all about Treap Data Structure. Implementation using C++ class.
- Treap: is a data structure which **combines binary tree and binary heap** (hence the name: tree + heap ⇒ Treap).

## Implementation Overview:

- We are trying to keep the main function as light as possible. The implementation is a menu based approach where we need to choose options based on operations we want to perform.
- We are creating the Node as a class and for each node we create an object.
- Treap is the main class which has root as the member variable and member functions to perform the operations on the AVL Tree.
- We are also defining some helper function which will be helpful for the implementation of the main member functions.
- All main member functions are kept public whereas the member variables, helper functions and wrapper functions are kept private scope.
- My TreeNode structure looks like this -

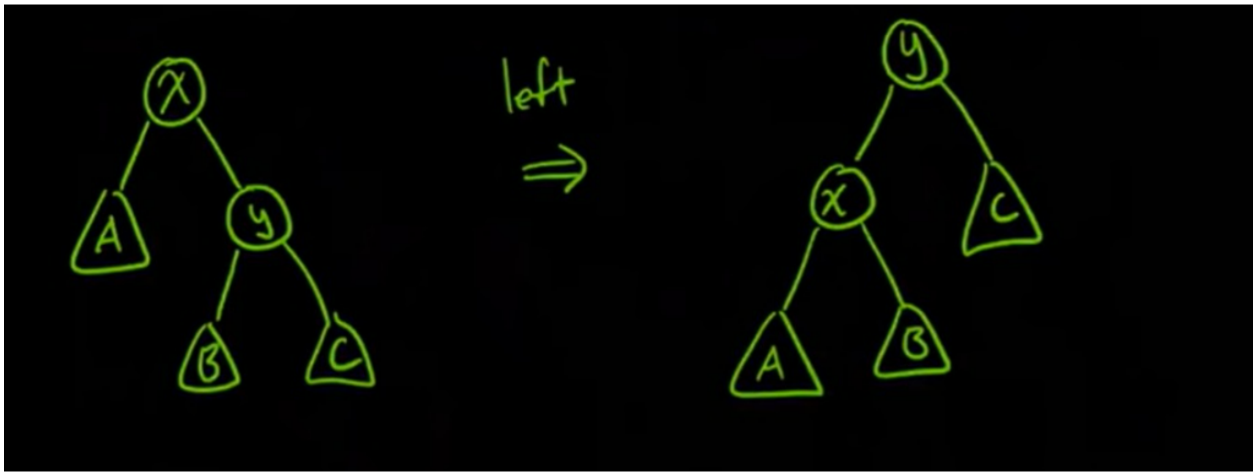
```
class Node {
   int data;
   int priority;
   Node  *left, *right;

}
```

- We are using multiple wrapper functions to implement the functionality with some changes.

## Implementation Details and Sample Input & Output:

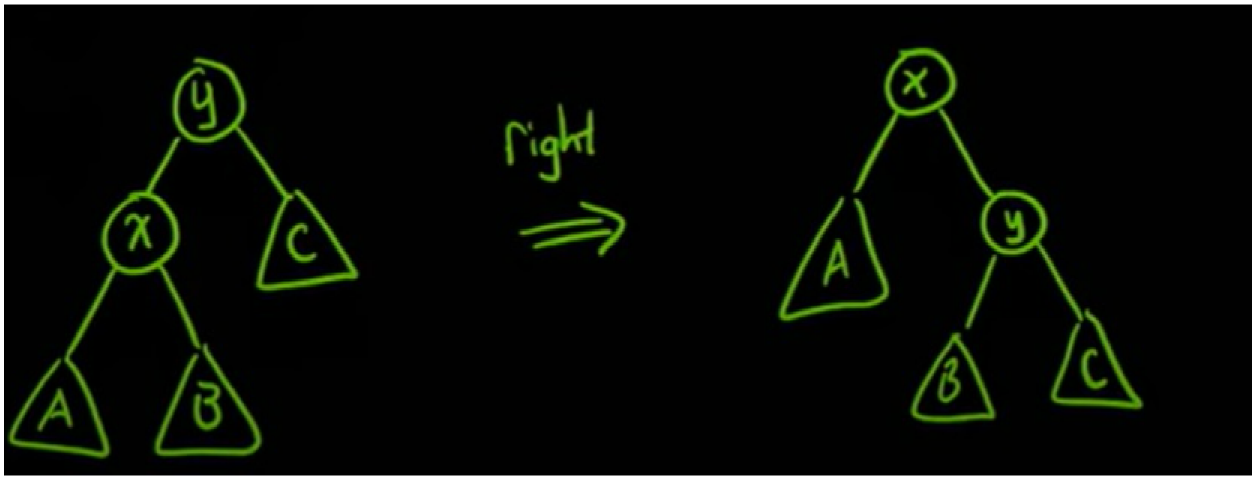
# Insert(x)

- This member function first searches for the element to be inserted. If it is found, we return back, else we go for the insertion method.
- Else we insert the element same as BST rule based on the key. The insertion will always be at the leaf node. Now we try to use heap property to balance the tree. And we are using the min heap for that. Heap property is implemented on priority, not on the key.
- Everytime we insert an element, we assign a random priority to each of the nodes. And the balancing should be based on the priority value.
- In min heap the priority of parents should be always less than that of the children and all elements in the tree should be assigned a unique priority value.
- Once we insert the element based on the BST concept based on the key value, we check whether it satisfies the heap property. If not we need one of 2 rotations namely left rotation and right rotation.
- Rotation Explained : **Left Rotation**

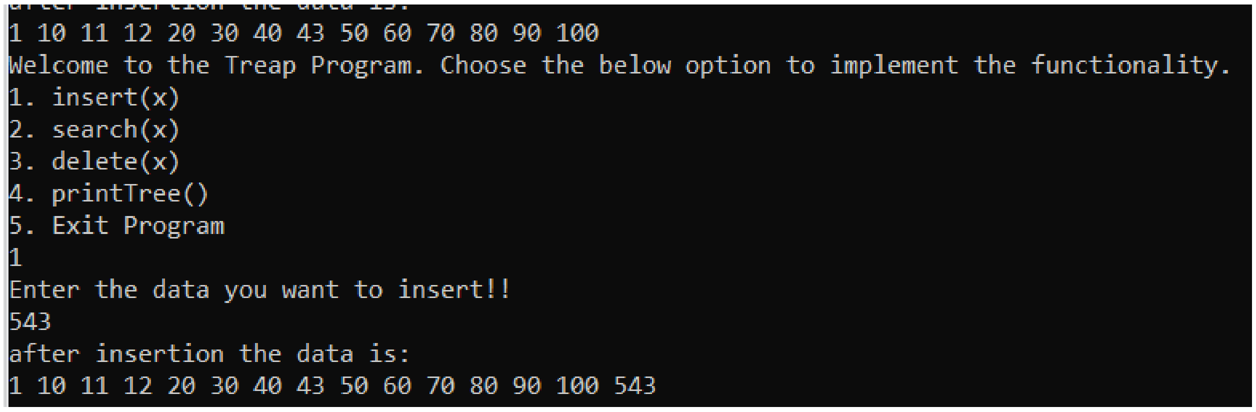


- As we are using a min heap, Left Rotation is required when priority of node X is greater than that of Y.
- We will try to understand the above example. We want to do Left Rotation. So we will assign X as the left child of Y. and assign the left child of Y to the right child of X. And finally return Y as the new root.

- Rotation Explained : **Right Rotation**

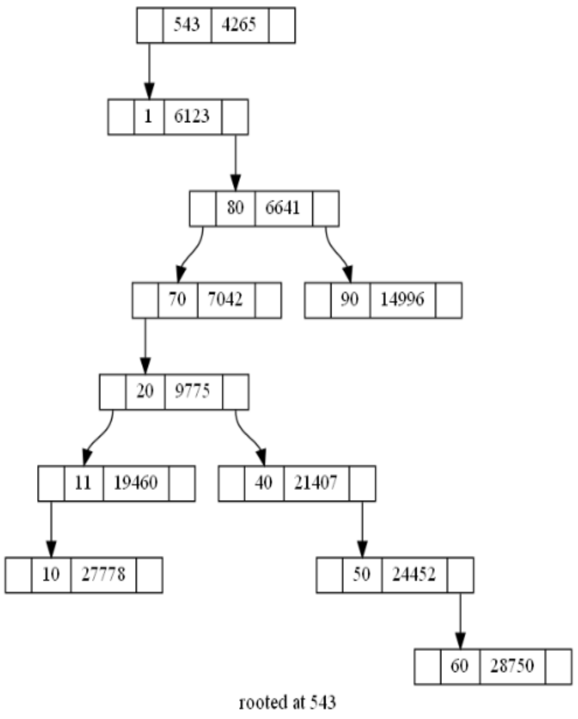


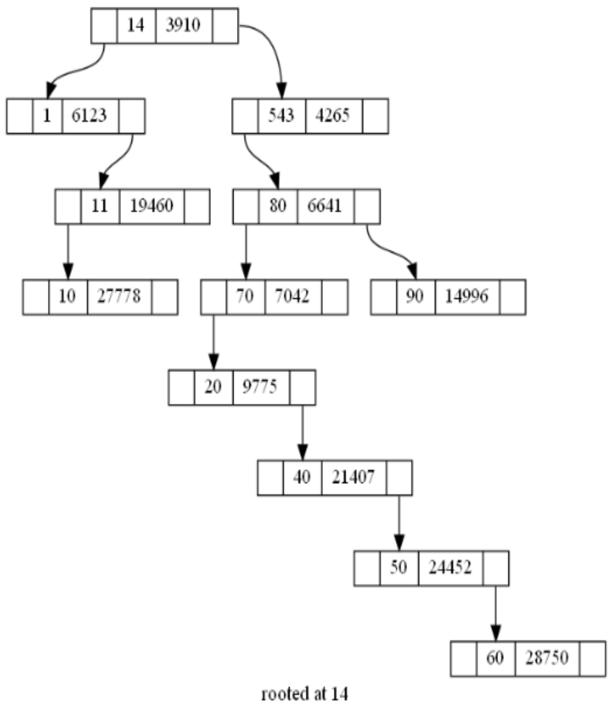
- As we are using a min heap, Right Rotation is required when priority of node Y is greater than that of X.
- We will try to understand the above example. We want to do Right Rotation. So we will assign Y as the right child of X. and assign the right child of X to the left child of Y. And finally return X as the new root.
- It does the above steps until it finds a node in a path for which min heap property is satisfied.
- Loop breaks when the node in the path from newly added node to root satisfies the min heap property.
-
- **Few cases sample Input and Output [Before - After] :**



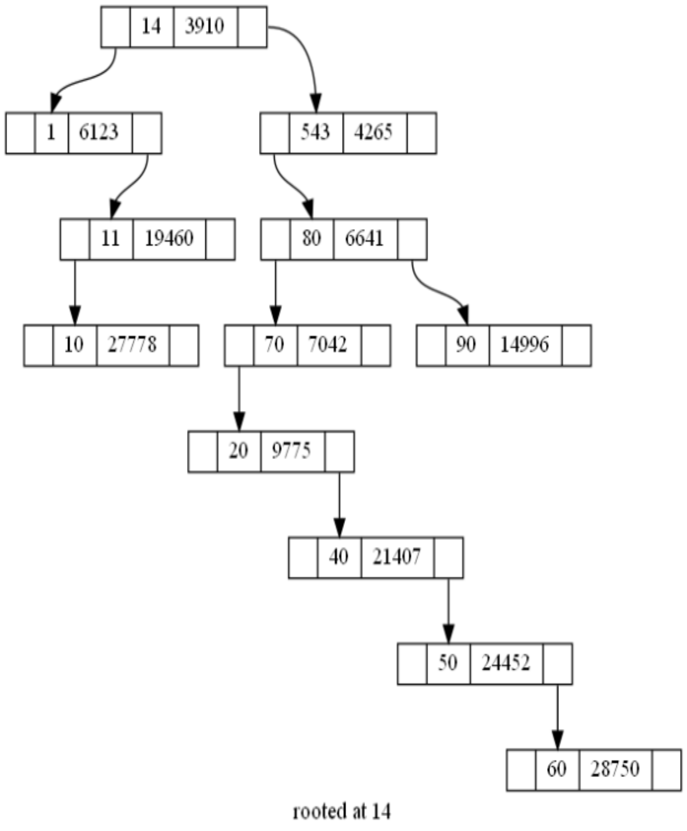
```
1 10 11 12 20 30 40 43 50 60 70 80 90 100
Welcome to the Treap Program. Choose the below option to implement the functionality.
1. insert(x)
2. search(x)
3. delete(x)
4. printTree()
5. Exit Program
1
Enter the data you want to insert!!
543
after insertion the data is:
1 10 11 12 20 30 40 43 50 60 70 80 90 100 543
```

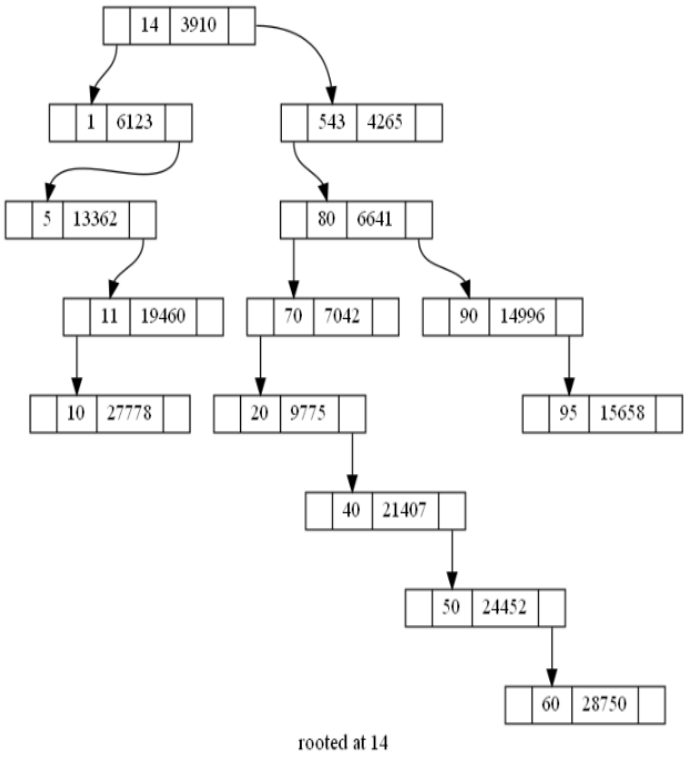
- Left Rotation [Insert 14]





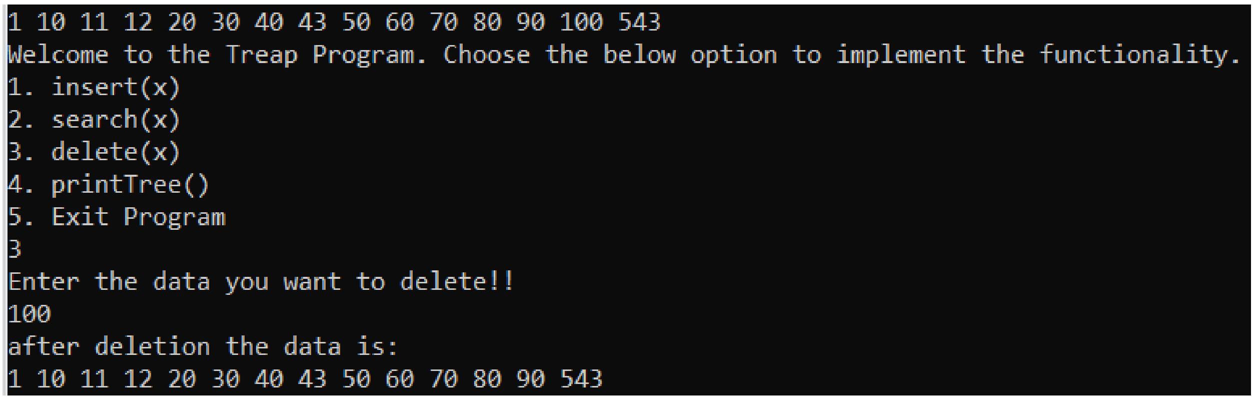
- Right Rotation [Insert 41]





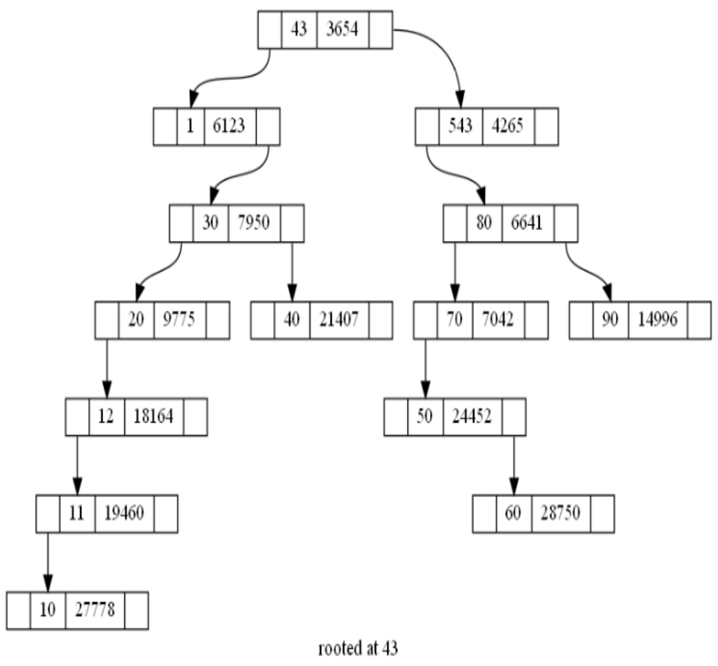
## remove(x)

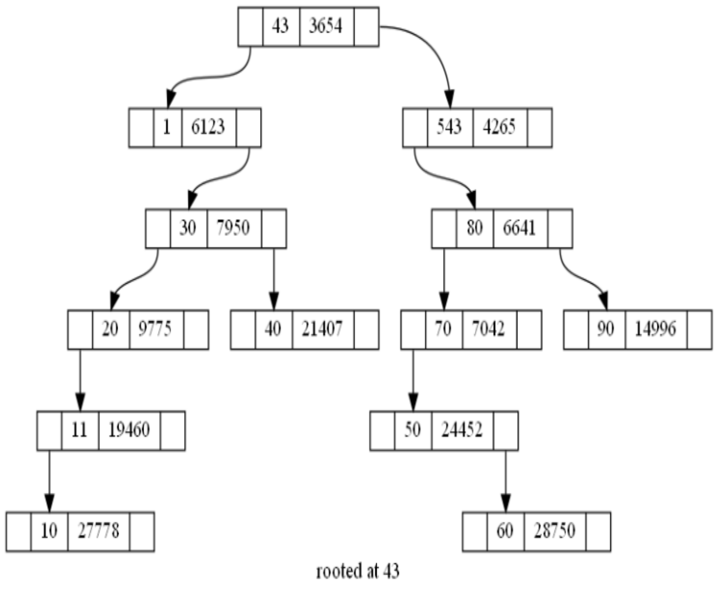
- This member function first searches for the element to be deleted. If it is found, we go to implement delete functionality else return back.
- In this member function, we will try to find a node to be deleted by iterating through the root to that node. We will use a recursive solution to do this.
- If it is not present, return and inform that element not found.
- The function will try to find the node whose key value is the same as the key to be deleted.
- If it hit to the node whose key value is same as key to be deleted then it follows below procedure:
- If it is a leaf node, simply delete that node.
- If  it has only one child (either left or right), point the parent to the children and delete that node.
- If it has 2 children then.
- 1. If the higher priority child is on right side compared to  the left side of the node to be deleted then do Right rotation
- 2. If the higher priority child is on left side compared to  the right side of the node to be deleted then do Left rotation
- Do this till the node to be deleted become the leaf node
- Unlink this leaf node.
-  If the node to be deleted is the last node in the tree then update the root pointer to point to NULL.
- **Few cases sample Input and Output [Before - After] :**



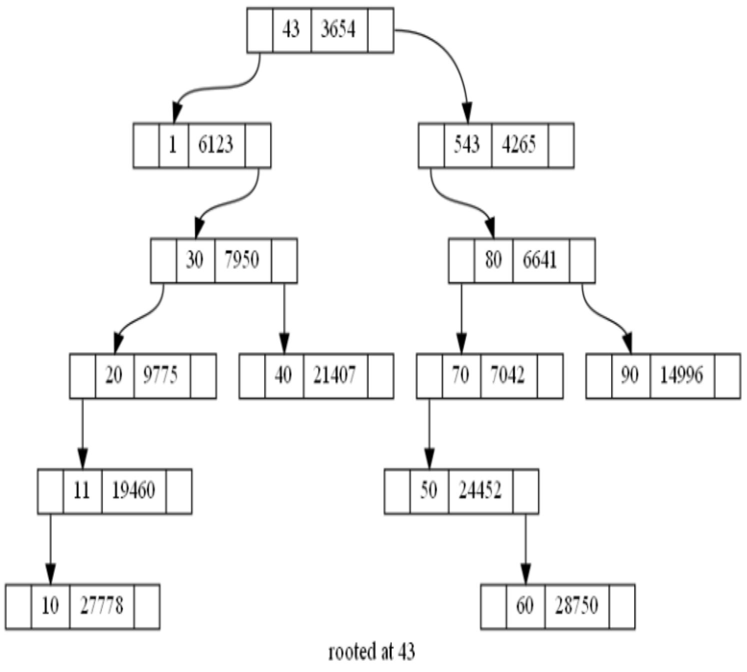
```
1 10 11 12 20 30 40 43 50 60 70 80 90 100 543
Welcome to the Treap Program. Choose the below option to implement the functionality.
1. insert(x)
2. search(x)
3. delete(x)
4. printTree()
5. Exit Program
3
Enter the data you want to delete!!
100
after deletion the data is:
1 10 11 12 20 30 40 43 50 60 70 80 90 543
```

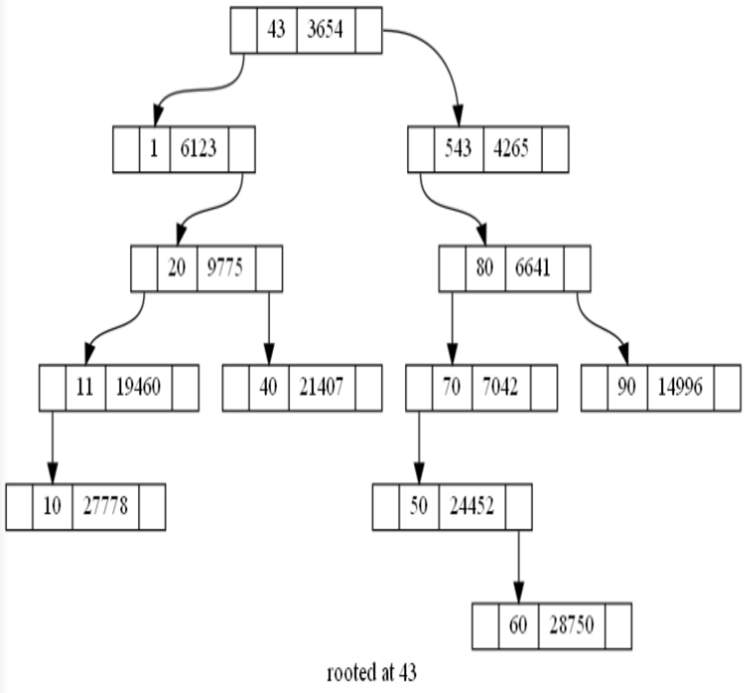
- The Node to be deleted has only one child. [Delete 12]



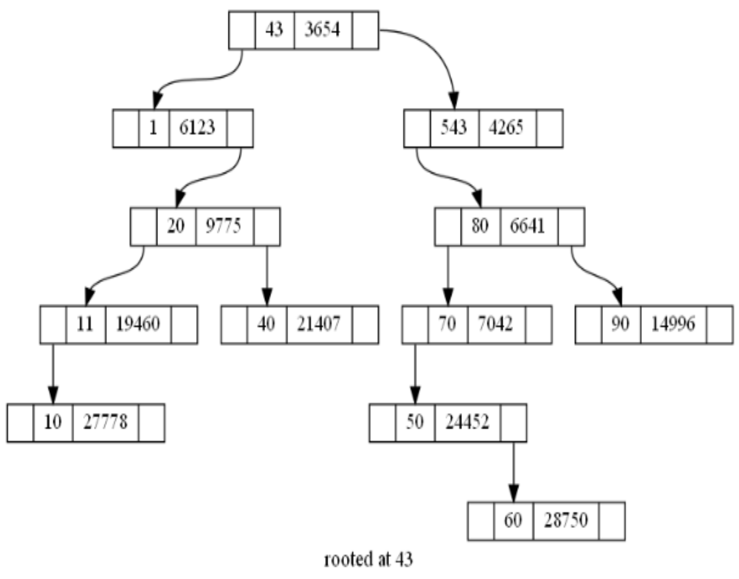


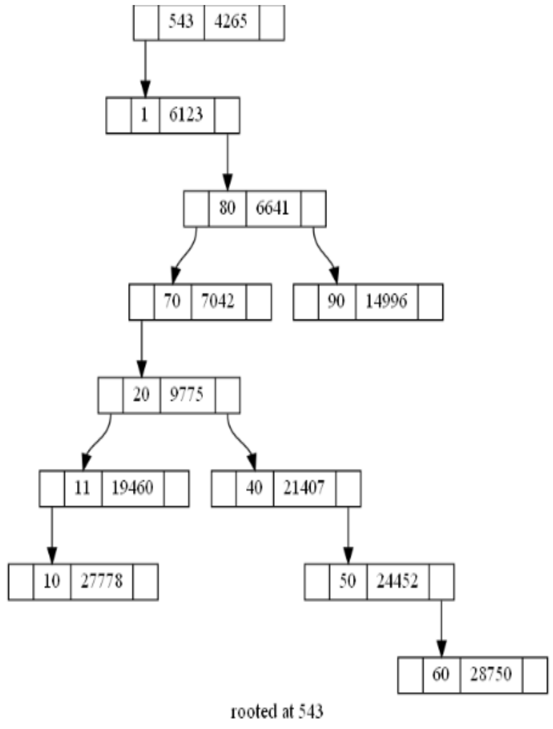
- The Node to be deleted has both children. [Delete 30]- Right Rotation



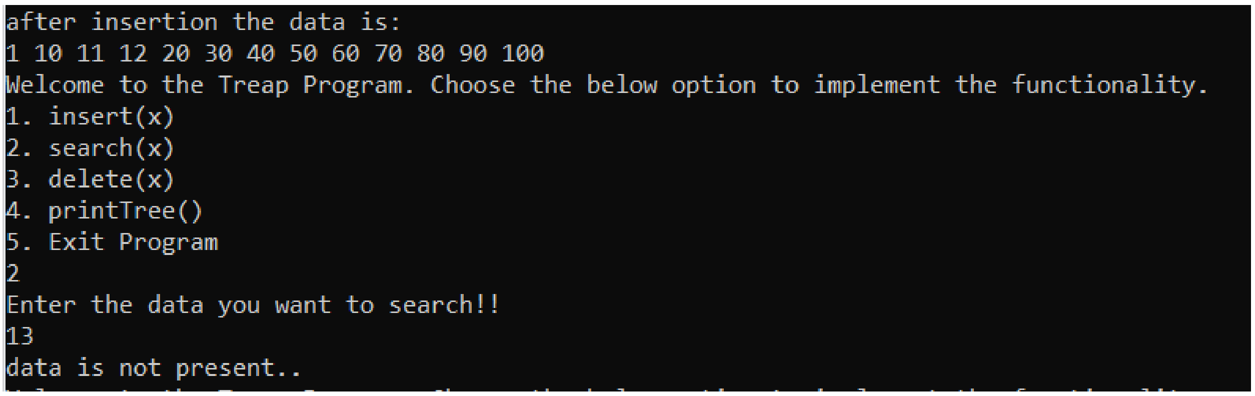


- The Node to be deleted has both children. [Delete 43]- Left Rotation





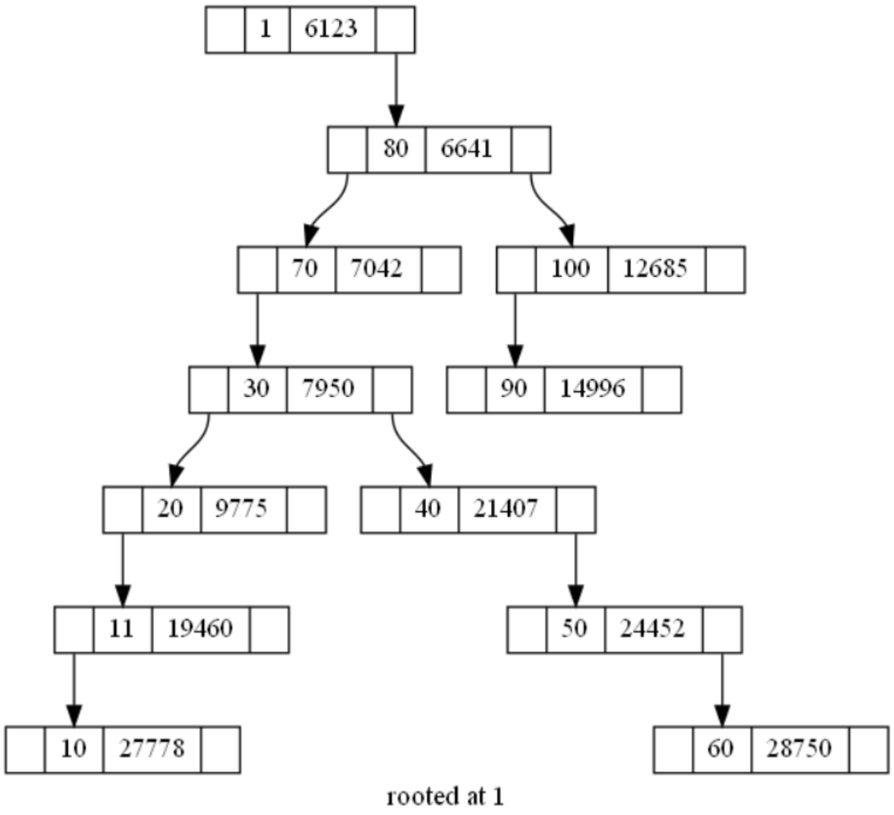
## search(x)

- Searching in Treap is the same as BST. The worst case time complexity is O(n) if the tree is totally skewed.
- This member function searches for the search key data in the tree. If it is found then it returns true otherwise it returns false.
- We are traversing the tree from top to bottom. For every node we check if the node value is greater than the search value, search is forwarded to the left side, else it is forwarded to the right side.
- I implemented an iterative approach to implement search functionality.
- In case if we found the element, we print that the **element is present**, else we say **not present**.
- 

```
after insertion the data is:
1 10 11 12 20 30 40 50 60 70 80 90 100
Welcome to the Treap Program. Choose the below option to implement the functionality.
1. insert(x)
2. search(x)
3. delete(x)
4. printTree()
5. Exit Program
2
Enter the data you want to search!!
13
data is not present..
```

## printTree()

- This member function prints the tree by storing it in a PNG file.
- We try to do level order traversal (or say BFS), and keep writing the edges in a dot file. We write all the edges in the dot file finally.
- And then execute the dot command which generates the JPG image file from the dot file.
- Sample Input & Output
    - Input Sequence: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 11, 1]



# Performance Comparison

# Treap Vs BST Vs AVL Tree

## SUMMARY

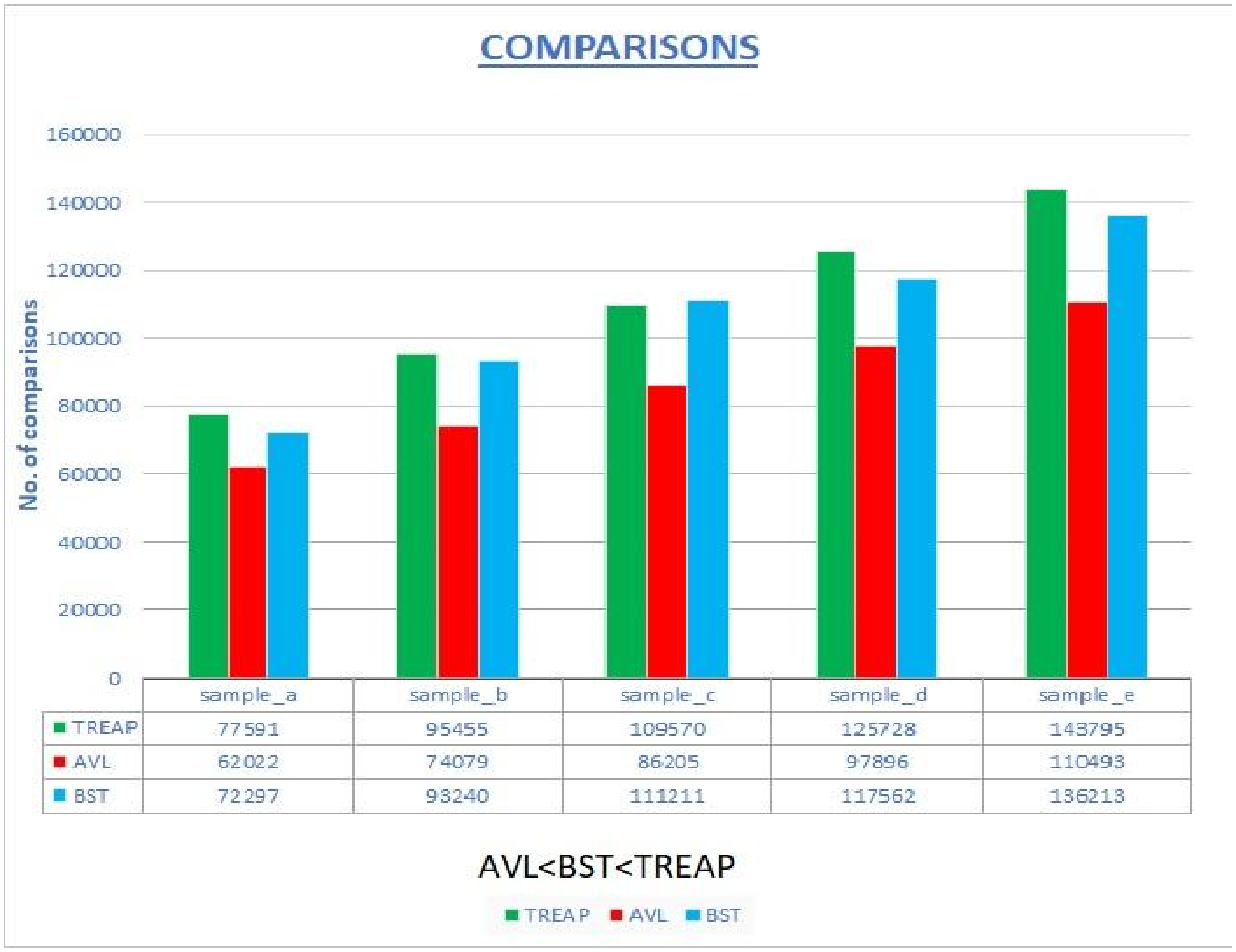
- We have created a random dataset with a mix of insertion and deletion operations.

| File Name | Insertion # | Deletion # |
|---|---|---|
| file sample_a.txt | 4667 | 1333 |
| file sample_b.txt | 5445 | 1555 |
| file sample_c.txt | 6223 | 1777 |
| file sample_d.txt | 7000 | 2000 |
| file sample_e.txt | 7777 | 2223 |

- We used the following parameters as the metrics of comparison.
    - **Total Key Comparison** : Includes both Insert and Delete operation.
    - **Height of Tree**: Max depth of the tree.
    - **Avg Height** : Avg height of nodes of the tree.
    - **Rotation Count**: Total number of rotation including both insertion and deletion.
- On analysis, the summary of observations are:
    - AVL Tree gives the best performance compared to Treap and BST. Thus wins in key comparison for insertion and deletion, avg height, max height. Since these parameters are totally dependent on the height.
    - BST and Treap are giving almost similar performance except avg height. Since we are inserting the element in both after applying the randomness element, the BST is also somewhat treated as similar to treap for insertion operation. Avg height of BST is always better than or equal to the height of Treap.
    - Avg height of BST <= Treap but max depth of Treap >= BST.
    - Randomness is the key element of the Treap data structure.

- The detailed observations for each parameter are explained below.

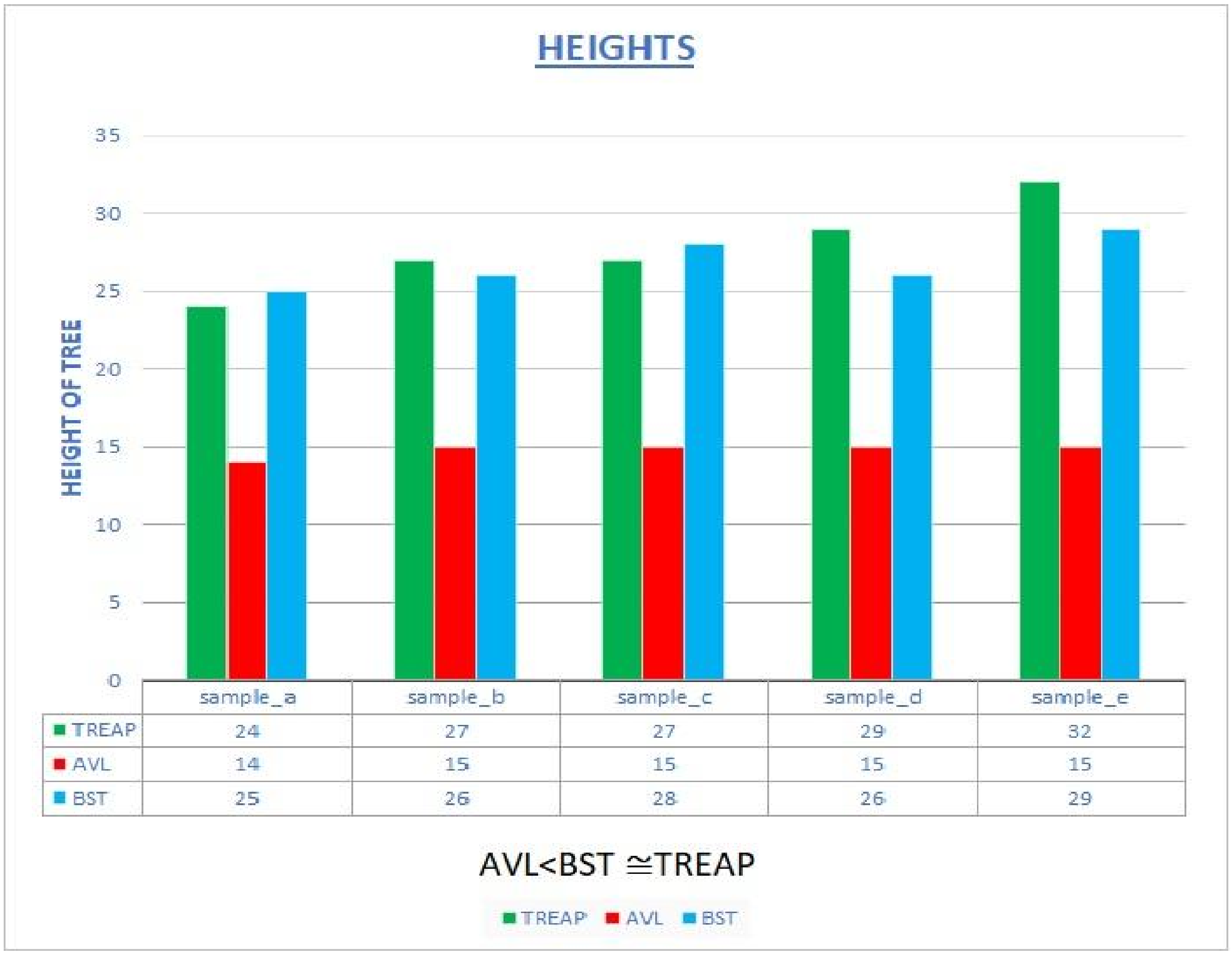
## Total Key Comparison



| | sample_a | sample_b | sample_c | sample_d | sample_e |
|---|---|---|---|---|---|
| 🟩 TREAP | 77591 | 95455 | 109570 | 125728 | 143795 |
| 🟥 AVL | 62022 | 74079 | 86205 | 97896 | 110493 |
| 🟦 BST | 72297 | 93240 | 111211 | 117562 | 136213 |

- AVL gives the best performance in the terms of key comparison. Since the height of BST is strictly bound to log(n), compared to other data structures.
- For example let’s take sample_a file. Total operation 4600+1300 close to 6000.
- Number of comparison close to 6000* log(6000) = 72000. So if we see, the comparison is close to 72k only. This validates the empirical and theoretical result.
- When it comes to Treap and BST, the best case height is O(logn) so the comparison is a bit more than the 72k we calculated above.

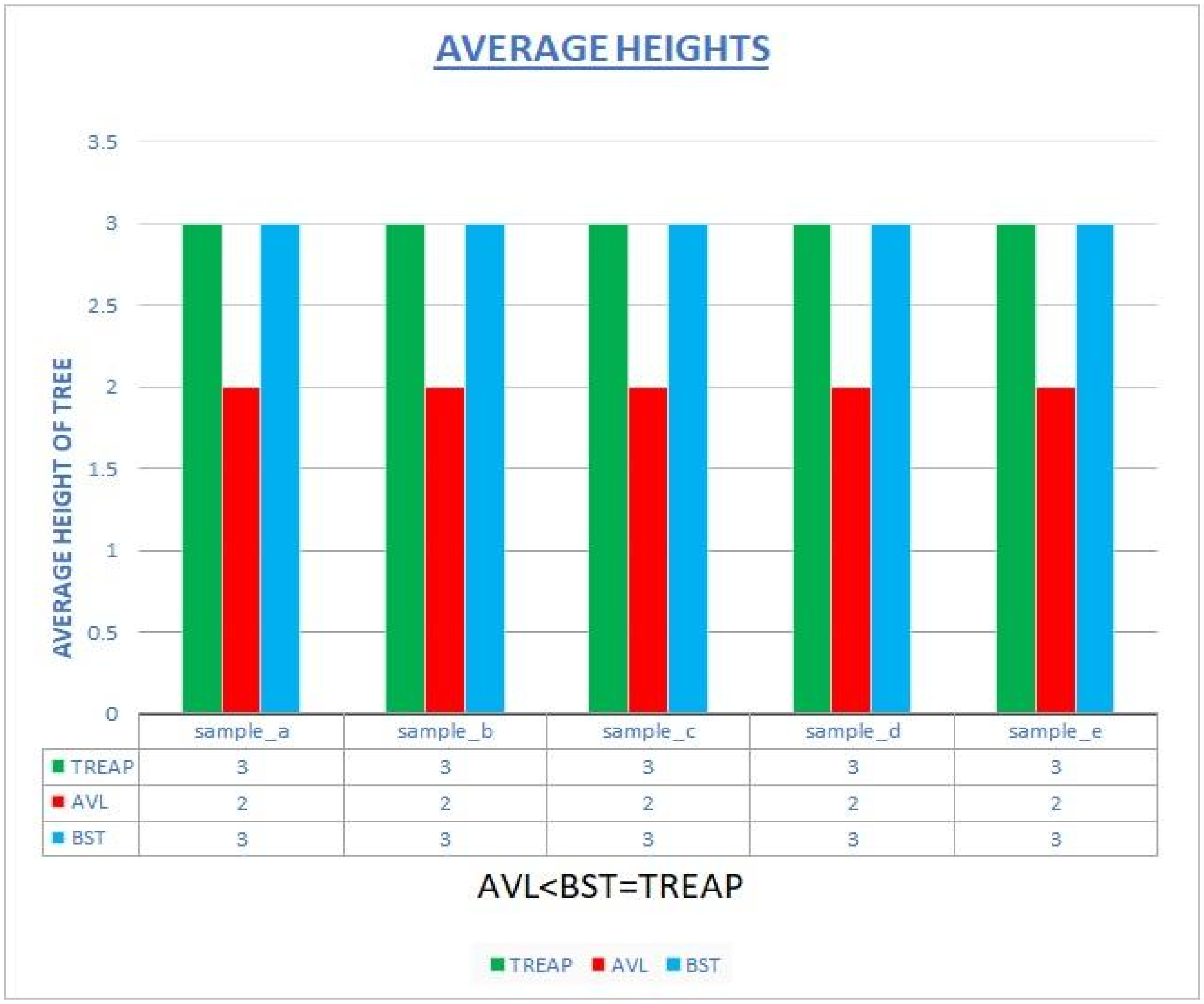
## Height of Tree



| | sample_a | sample_b | sample_c | sample_d | sample_e |
|---|---|---|---|---|---|
| 🟩 TREAP | 24 | 27 | 27 | 29 | 32 |
| 🟥 AVL | 14 | 15 | 15 | 15 | 15 |
| 🟦 BST | 25 | 26 | 28 | 26 | 29 |

- AVL gives the best performance in terms of height(max depth). Since the height of BST is strictly bound to log(n), compared to other data structures.
- Whereas in other data structures like Treap and BST the height of the tree in the worst case can go to O(n).
- For example let's take sample_a file. Total operation 4600+1300 close to 6000.
- Height of the AVL Tree should be close to log(6000) = 13. So if we see, the comparison is close to 13 only for AVL. And it is less than **$1.44*\log_2 n$** also. This validates the empirical and theoretical result.
- When it comes to Treap and BST, the best case height is O(logn) so the comparison is a bit more than the 13 we calculated above.
- When it comes to BST Vs Treap, height of Treap >= BST. It is mainly because of the difference in deletion approach for both the data structure. Also BST is not going for it's worst case as we are inserting data randomly.

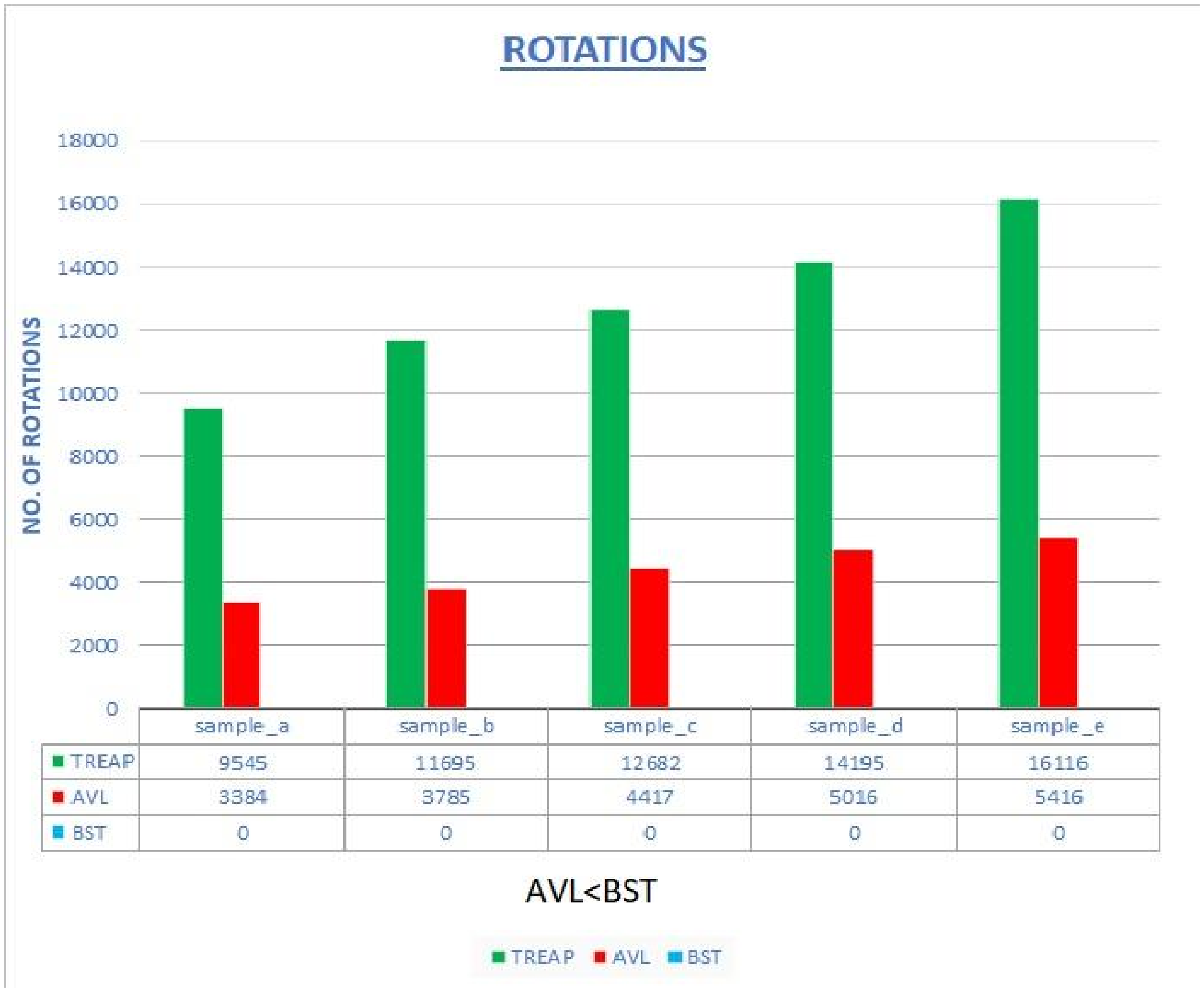
## Avg Height of Nodes



| | sample_a | sample_b | sample_c | sample_d | sample_e |
|---|---|---|---|---|---|
| 🟩 TREAP | 3 | 3 | 3 | 3 | 3 |
| 🟥 AVL | 2 | 2 | 2 | 2 | 2 |
| 🟦 BST | 3 | 3 | 3 | 3 | 3 |

- AVL gives the best performance in terms of avg height of nodes of the tree compared to the BST and Treap.
- In all cases, the avg height of nodes of the AVL tree is 2. And in most cases, the avg height of nodesTreap and BST is less than equal to 3.
- It means that in an AVL tree the majority of elements lies either at the leaf and immediate parent of leaf node.

## Rotation Count



| | sample_a | sample_b | sample_c | sample_d | sample_e |
|---|---|---|---|---|---|
| 🟩 TREAP | 9545 | 11695 | 12682 | 14195 | 16116 |
| 🟥 AVL | 3384 | 3785 | 4417 | 5016 | 5416 |
| 🟦 BST | 0 | 0 | 0 | 0 | 0 |

- There is no rotation when it comes to BST. Rotation only happens in AVL and Treap data structure.
- Since the max depth of Treap tree is always greater than AVL by around 2 times, as the empirical and theoretical result says. The rotation cnt in Treap is more than AVL (more than 2 times), as the rotation needs to be balanced from leaf to root node in the worst case.
- For example let's take sample_a file. Total operation 4600+1300 close to 6000.
- Height of the AVL Tree should be close to log(6000) = 13. So if we see, the comparison is close to 13 only for AVL. Treap rotation 9500 is close to 2.5 times of AVL which is 3384 which is justifiable. This validates the empirical and theoretical result.
- Hence, AVL rotation is taking less operation compared to the Treap.