# REPORT: Analysis of Results

## INTRODUCTION

This report aims to define an overall analysis of Binary Search Tree(BST), AVL Trees and Treaps, and compare the results based on certain parameters [ like Total Number of comparisons while doing various Operations, Final height of the tree etc. ] empirically and theoritically.

A Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

### *Why AVL Trees?*

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Like AVL Trees, Treap is also a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The **expected** time complexity of search, insert and delete is O(Log n).

In ALL of the above mentioned Data-Structures, this report has considered following main Parameters which are common in all data-structures:-

- Total Number of Comparisons while doing Insertion
- Total Number of Comparisons while doing Deletion.
- Final Height of Tree Obtained
- Total Rotations Involved

Additionally I would like to consider Search time Complexity which is NOT included in the Final_Analysis File made by my program named “214101020_TestFileGenerationAndParametersCalc”, which you will get hereby zipped with the documentation.

## Empirical Results

Based on the “Final_Analysis.txt File” generated by the program “214101020_TestFileGenerationAndParametersCalc”. Observation is as follows:-

For Test File 1, if we run with 10000 Random Insert-Delete Operations, the result obtained is as follows:

- For BST:

| | |
|---|---|
| Total No. of Comparison while Inserting: | 37902 |
| Total No. of Comparison while Deleting: | 29974 |
| Height of Final Tree: | 14 |
| Total No. of Rotations: | 0 |

- For AVL:

| | |
|---|---|
| Total No. of Comparison while Inserting: | 23284 |
| Total No. of Comparison while Deleting: | 22516 |
| Height of Final Tree: | 6 |
| Total No. of Rotations: | 3128 |

- For Treap

| | |
|---|---|
| Total No. of Comparison while Inserting: | 44244 |
| Total No. of Comparison while Deleting: | 42538 |
| Height of Final Tree: | 15 |
| Total No. of Rotations: | 23419 |

For Test File 2, if we run with 8000 Random Insert-Delete Operations, the result obtained is as follows:

- For BST:

| | |
|---|---|
| Total No. of Comparison while Inserting: | 27902 |
| Total No. of Comparison while Deleting: | 22341 |

| | |
|---|---|
| Height of Final Tree: | 11 |
| Total No. of Rotations: | 0 |

- For AVL:

| | |
|---|---|
| Total No. of Comparison while Inserting: | 17975 |
| Total No. of Comparison while Deleting: | 16850 |
| Height of Final Tree: | 6 |
| Total No. of Rotations: | 2837 |

- For Treap

| | |
|---|---|
| Total No. of Comparison while Inserting: | 29574 |
| Total No. of Comparison while Deleting: | 29101 |
| Height of Final Tree: | 10 |
| Total No. of Rotations: | 14655 |

Similarly for rest 3 Test_Files we have following results:-

- BST based implementation of Test File 3 details:

  Total No. of Comparison while insertion: 31521
  Total No. of Comparison while deletion: 25691
  Height of final tree: 7
  Total No. of Rotations:0

- AVL based implementation of Test File 3 details:

  Total No. of Comparison while insertion: 20125
  Total No. of Comparison while deletion: 19233
  Height of final tree: 5
  Total No. of Rotations:2973

- Treap based implementation of Test File 3 details:

Total No. of Comparison while insertion: 33879
Total No. of Comparison while deletion: 33783
Height of final tree: 9
Total No. of Rotations:17215

- BST based implementation of Test File 4 details:

  Total No. of Comparison while insertion: 19476
  Total No. of Comparison while deletion: 15096
  Height of final tree: 13
  Total No. of Rotations:0

- AVL based implementation of Test File 4 details:

  Total No. of Comparison while insertion: 13646
  Total No. of Comparison while deletion: 11784
  Height of final tree: 6
  Total No. of Rotations:2474

- Treap based implementation of Test File 4 details:

  Total No. of Comparison while insertion: 20972
  Total No. of Comparison while deletion: 20526
  Height of final tree: 11
  Total No. of Rotations:10333

- BST based implementation of Test File 5 details:

  Total No. of Comparison while insertion: 13831
  Total No. of Comparison while deletion: 10813
  Height of final tree: 8
  Total No. of Rotations:0

- AVL based implementation of Test File 5 details:

  Total No. of Comparison while insertion: 10736
  Total No. of Comparison while deletion: 8684
  Height of final tree: 5
  Total No. of Rotations:2315

- Treap based implementation of Test File 5 details:

  Total No. of Comparison while insertion: 14733
  Total No. of Comparison while deletion: 14919
  Height of final tree: 8
  Total No. of Rotations:7333

Note: Above Figures may change when you try to implement my program because a random integer generator is used to do insertion and deletion. Also the key to insert or delete is generated randomly.

If we plot Insertion, Deletion And Height Curves. It would be as follows:-

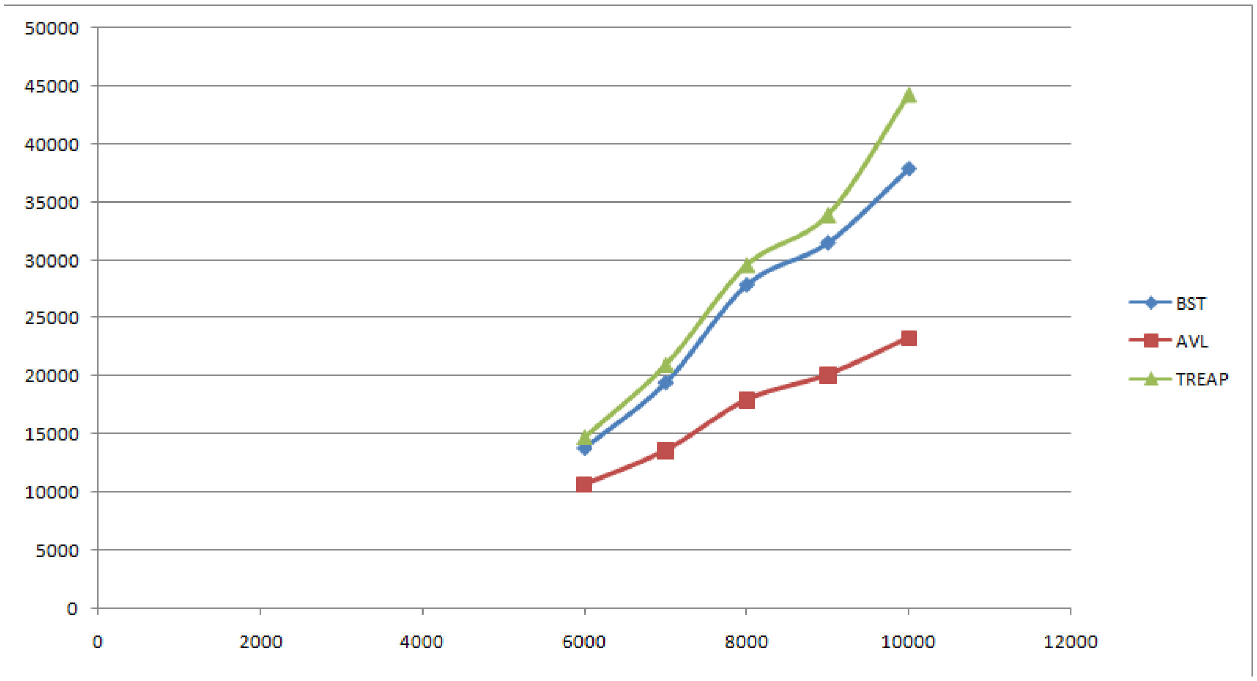

Fig: Insert Curve of all 3 Data-Structures. Y-axis Represent No. of comparison while inserting & X-axis represent No. Of Elements

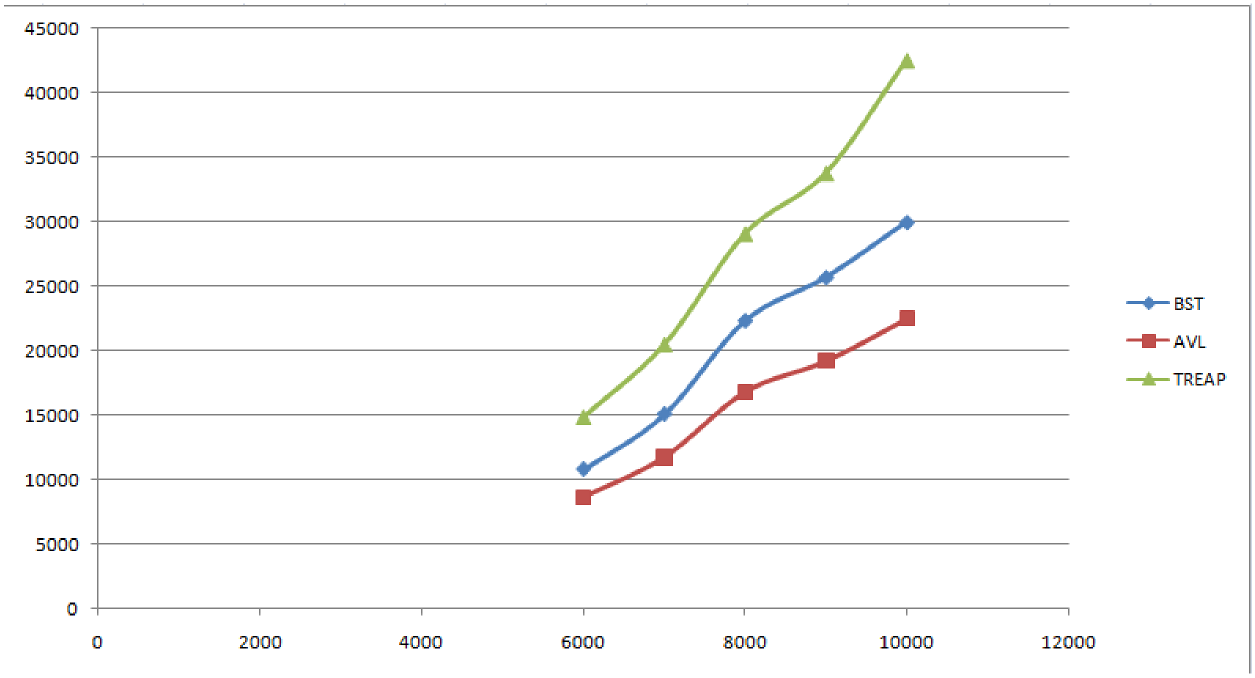

Fig: Delete Curve of all 3 Data-Structures. Y-axis Represent No. of comparison while deleting & X-axis represent No. Of Elements

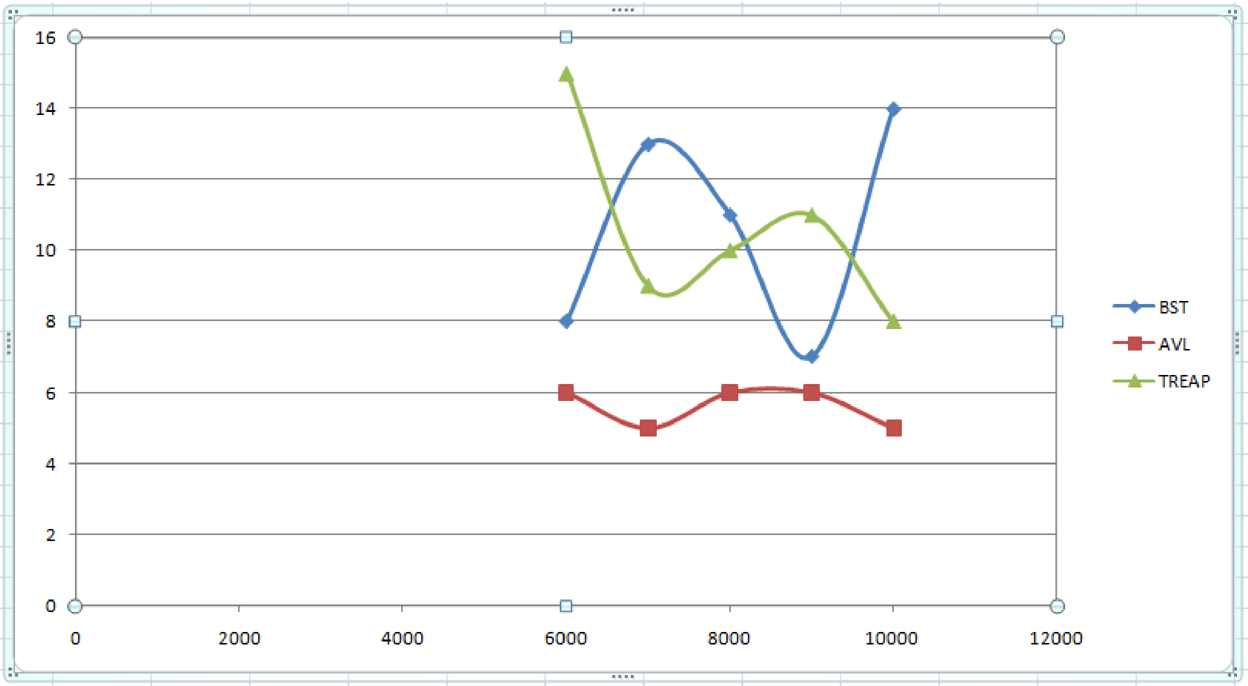

Fig: Height Curves all 3 Data-Structures. Y-axis Represent final height of tree & X-axis represent No. Of Elements

## Theoretical Expectation

For BST:
- The tree making time of BST is O(n)
- The insert time complexity in BST is O(n)
- Deletion takes O(n).
- Search requires O(n) time.
- No rotations are involved.
- Tree split is a complex Operation.

For AVL:
- The tree making time  is O(nlogn).
- The insert time complexity is O(logn).
- Deletion takes O(logn) time.
- Search too requires O(logn) time complexity.
- Overhead of rotation required to maintain tree property.
- Tree split is a complex operation.

For Treap:
- The tree making complexity is O(n).
- The insert time complexity is O(n).
- Deletion takes O(n) time
- Search is expected to be O(n) time operation.
- Overhead of rotation to maintain heap property.
- Tree split is very easy using Treaps.

## Method Used For Construction

BST Construction:-
- Insert: Insertion happens at the leaf of the Tree.

    - Start comparing from root, if Element less than root then go to left subtree else if Element greater than root go to right subtree until there is a NULL pointer signifying Position to insert.

- Delete: Deletion has 3 cases:-
    - If a leaf is to be deleted, delete it directly and adjust the parent pointer.
    - If Node with one child, replace it with child, then delete child.
    - If Node with 2 children, Swap with inorder successor then delete the inorder successor.

- Search: Search is easy just start with root and traverse LST if Node->value less than Root->value else go to right subtree. Repeat until element found or else output NULL.

AVL Construction:-

- Insert: Same as BST. At the end, Check balance condition. Rotate if necessary. Reassign balancing factor

- Delete:Same as BST. At the end, Check balance condition. Rotate if necessary. Reassign balancing factor

- Search:Same as BST.

Treap Construction:- The structure of Node is such that has an additional parameter called priority. It helps in maintaining heap property.

- Insert: To insert a key-priority pair (K,P) into a treap, do the following:

    Insert the pair as a new leaf, using the usual BST insert algorithm using the key value K. Then rotate the newly inserted node up using AVL rotations as necessary, until the priority of its parent is greater than or equal to P, or the node becomes the root. Since AVL rotations are constant-time operations, insert in a treap can be performed in time O(H), where H is the height of the treap.

- Delete: To delete a key K, do the following:

  Search for the node X containing K using the usual BST find algorithm. If the node X is a leaf, unlink it from its parent. Otherwise, use AVL rotations to rotate the node down until it becomes a leaf; then delete it. If there are 2 children, always rotate with the child that has the smaller priority, to preserve heap ordering. Since AVL rotations are constant-time operations, delete in a treap can be performed in time O(h), where h is the height of the treap.

- Search: Same as BST.

## Analysis of Empirical Results

As per the empirical data of Test File 1,2,3,4 & 5, for a Large Number of Operations say, 10000,8000,9000, 7000 & 6000 respectively, following points can be incurred:-

1. The AVL Tree Performs much better than Treap or a BST while doing insertion and Deletion. If we see theoretically also, AVL tree should outperform the other 2 data-structures. Since insertion & deletion happens O(log n). So Total Number of Comparison is minimum in AVL Tree.
2. BST has the least penalty on the number of rotations. Since, it does NOT balance the tree, ever.

3. The TREE_MAKING_TIME ( Theoretically ) of BST is minimum, Since there is an overhead of Rotation in other 2 data-structures. Theoretically, it takes O(nlogn) time to build an AVL Tree, O(n) for BST and EXPECTED COMPLEXITY of O(n) in Treap.
4. Height of Final tree is minimum in case of AVL Tree. But Nothing can be said about the minimum between Treap and BST. Sometimes Treap has the minimum height, sometimes BST.
5. Considering Search ( Theoretically ). The Search in AVL Tree has a complexity of O(logn), BST has O(n) and Treap has EXPECTED COMPLEXITY of O(logn).
6. Based on above parameters Treap seems much worse than the other 2 data-structures BUT It has an important property. Tree Splitting is easier with treap. Formally-

Problem: Given a tree and a key value K not in the tree, create two trees: One with keys less than K, and one with keys greater than K.

This is easy to solve with a treap, once the insert operation has been implemented:

• Insert (K,-∞) in the treap

• Since this has a higher priority than any node in the heap, it will become the root of the treap after insertion.

• Because of the BST ordering property, the left subtree of the root will be a treap with keys less than K, and the right subtree of the root will be a treap with keys greater than K. Since insertion can be done in time O(h) where h is the height of the treap, splitting can also be done in time O(h).

## Conclusion

The Data-Structure to be used should be based on requirement. For example, if we need insert-delete optimal data-structure then use AVL Tree. Likewise all data-structures have their own pro’s and con’s. We need to choose them based on a case-to-case basis.