CS261P Data Structures

Project 2

Balanced vs Unbalanced Trees

Binary Search Trees, AVL Trees & Splay Trees

Name	UCI NetID	Student ID
Parth Shah	parths4	54809514
Akansha Kaushik	akanksk1	14123771

1. Introduction

A tree data structure is a widely used data type which simulates a hierarchical relationship between its data items.

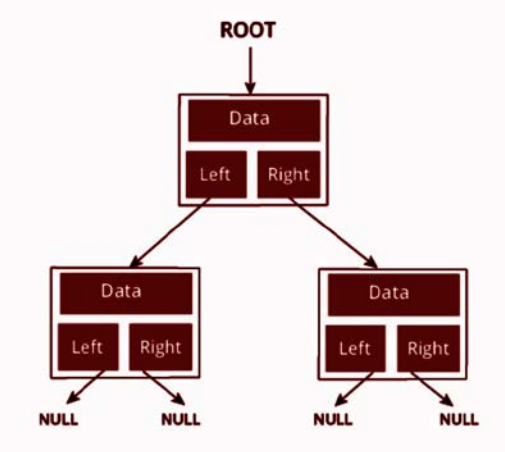


Fig.1.1. Representation of a Tree Data Structure

Trees have a wide range of applications - storing hierarchical data like folder structure, organization structure, used in compilers, implementing priority queues, for quick pattern searching, implementing dictionaries, finding shortest paths in a network etc.

2. Data Structures

Trees are especially beneficial in designing databases for performing insertions, deletions, search and implementing indexing in databases. Choosing the right type of tree can impact performance significantly. An inefficient approach will slow down the system and will lead to unnecessarily increased costs.

The chosen data structures for comparative analysis in this project are:

1. Binary Search Tree
2. AVL Trees
3. Splay Trees

The above data structures are being compared using insertion costs, deletion costs and search costs. This is achieved by varying the range size of the input - 100, 1000, 5000, 10000. Additionally, the dataset has been arranged in different orders - random vs sorted order.

2.1. Binary Search Tree

2.1.1. Properties and structure

The properties of the Binary Search Tree:

1. The left subtree of any node contains only nodes whose key values are lesser than the value of the node in consideration.
2. The right subtree of any node contains only nodes whose key values are greater than the value of the node in consideration.
3. Both the left and right subtree at any node must also be a binary search tree.
4. No two nodes will have the same key value.

```
static class Node{
    int key;
    Node left,right;
    public Node(int value){
        key=value;
        left=null;
        right=null;
    }
}
```

Fig 2.1.1. BST Node

2.1.2. Theoretical Analysis

2.1.2.1. Insertions in Random Order

The expected cost for insertion of key x = E(the numbers of nodes traversed in the search path of key x).

The number of nodes traversed in the search path of key x can be evaluated - For any other key y, let C(y) = 1 if we encounter y when searching for x, C(y) = 0 otherwise.

So average insertion cost is **O(log n)** if insertions are in random order.

2.1.2.2. Insertions in Sorted Order

The tree structure tends to become a right-skewed tree in case of ascending order and left-skewed tree for descending order dataset.

Hence, the insertion cost, in this case, would be the Number of nodes already present in the tree i.e. **O(n)**.

2.1.2.3. Time and Space Complexities

Algorithm	**Average case**	**Worst Case**
Space Complexity	O(n)	O(n)
Time Complexity		
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

2.2. AVL Trees

2.2.1. Properties and structure

In an AVL Tree, at each node, the difference between the height of the left subtree and the height of the right subtree can be at most one. If at any given point, this property is violated then rebalancing is done at the node to restore this property.

Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

Rebalancing of an AVL Tree is done using rotations:

a. Single Rotations
    i. Left Rotation
    ii. Right Rotation
b. Double Rotations
    i. Left-Right Rotation
    ii. Right-Left Rotation

```
static class Node{
    int key;
    int height;
    Node left;
    Node right;
    public Node(int value){
        height = 1;
        key = value;
        left = null;
        right = null;
    }
};
```

Fig.2.2.1. AVL Tree Node

2.2.2. Theoretical Analysis

2.2.2.1. Number of Rotations - Insertion

During insertion of a node at **most one (single or double) rotation is needed**, at the lowest point where the tree is out of balance.

- Insert a node via standard BST insertion. Follow a path from the leaf back up to the root, updating the heights of each node on the way.
- If a node becomes unbalanced, perform rotations to abide by the balance condition.
- After rotation the height of that subtree is the same as in the original subtree, so all nodes upwards are balanced.

2.2.2.2. Number of Rotations - Deletion

Deletion usually requires more rotations to maintain the balanced condition of the AVL Tree.

- A single rotation does not necessarily restore the original tree height, so the tree has to be updated at other levels higher up in the tree.
- Worst case trees are those which are minimal AVL trees,i.e. no node can be removed without violating the AVL property.
- There you might have to rotate every level, thus **a logarithmic number of times, i.e. O(log n).**

2.2.2.3. Time and Space Complexities

Algorithm	**Average case**	**Worst Case**
Space Complexity	O(n)	O(n)
Time Complexity		
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)
Search	O(log n)	O(log n)

2.3. Splay Trees

2.3.1. Properties and structure

A Splay tree is a self-adjusting binary search tree with an additional property that the recently accessed node becomes the root of the tree - this operation is called “Splaying”. No explicit balancing condition is maintained.

Splaying at Node X is achieved with the help of three rotations:

a) X has no grandparent
b) X is Left-Left or Right-Right grandchild
c) X is Left-Right or Right-Left grandchild

Tree operations are like Binary Search Tree with an additional splaying operation as follows:

After Insert (Node X): splay the Node X
After Delete (Node X):
    Successful: splay the parent of X
    Unsuccessful: splay at the last node on search path
After Search (Node X):
    Successful: splay node X
    Unsuccessful: splay at the last node on search path

The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again.

```
static class Node
{
    int key;
    Node left;
    Node right;
    Node(int value)
    {
        this.key = value;
        this.left = null;
        this.right = null;
    }
}
```

Fig. 2.3.1 Splay Tree Node

2.3.2. Theoretical Analysis

For a node w in a splay tree, let |w| be the number of external nodes descending from w (i.e., one more than the number of keys in the subtree rooted at w).

The change in potential for an operation is the sum of:

1. The change in potential during standard binary tree operations - prior to splay.
2. The change in potential during the splay.

2.3.2.1. Amortized analysis

ΔΦ prior to the splay:

- Search - no change
- Delete - decreases
- Insert - +O(log n)

ΔΦ during the splay:

Let ΔΦ be the change in the potential function due to one of the rebalance operations shown for Rotation 1, Rotation 2, or Rotation 3 mentioned in Section 2.3.1, and let r be the factor by which |x| increases during that operation. Then:

- For Rotation 1 : $\Delta\Phi - \lg r \leq 0$
- For Rotation 2 and 3: $\Delta\Phi - \lg r3 \leq -2$

Corollary: Let d be the initial distance from x to the root. Then when we splay at x, so that it becomes the root, the change in the potential function is

$$\Delta\Phi \leq -(d - 1) + 3 \lg(n + 1).$$

Hence, The amortized cost of a search, insert, or delete operation on a splay tree is **O(log n)**, where n is the number of elements present.

2.3.2.2. Number of Rotations

Rotations in splay tree and necessary for insertions, deletions as well as search operations. Number of rotations depend on the position of the node to be splayed in the tree. If the node is close to the root node less number of rotations are required whereas a leaf node might require more rotations.

2.3.2.3. Time and Space Complexities

Algorithm	**Average case**	**Worst Case**
Space Complexity	O(n)	O(n)
Time Complexity		
Insert	O(log n)	Amortized O(log n)
Delete	O(log n)	Amortized O(log n)
Search	O(log n)	Amortized O(log n)

3. Design and Implementation

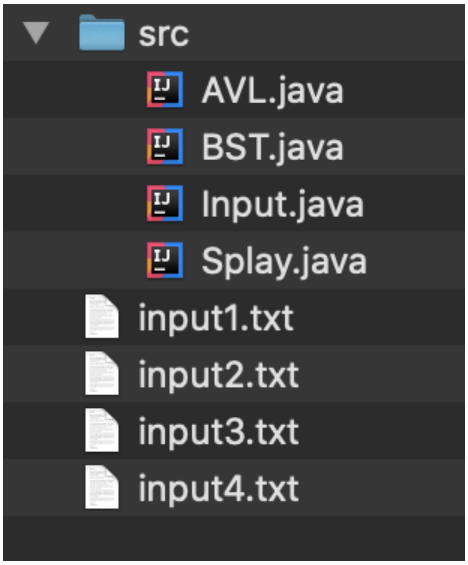


Fig. 3.1. Implementation Files

BST.java, AVL.java, Splay.java - Implement all the operations of the Binary Search Tree, AVL trees and Splay trees respectively.

Input.java - This file is used to generate different input files with a varied range of dataset size or ordered dataset.

Input1.txt - contains 100 data items in random order.

Input2.txt - contains 1000 data items in random order.

Input3.txt - contains 5000 data items in random order.

Input4.txt - contains 10000 data items in sorted order.

We use time as a measure to compute the performance, as it gives a more general way to compare the efficiency of different operations in different trees.

(Counting number of nodes accessed or number of rotations for evaluating performance would complicate comparisons. For eg- No. of rotations for insertion in BST=0, AVL <=2 , Splay= log n)

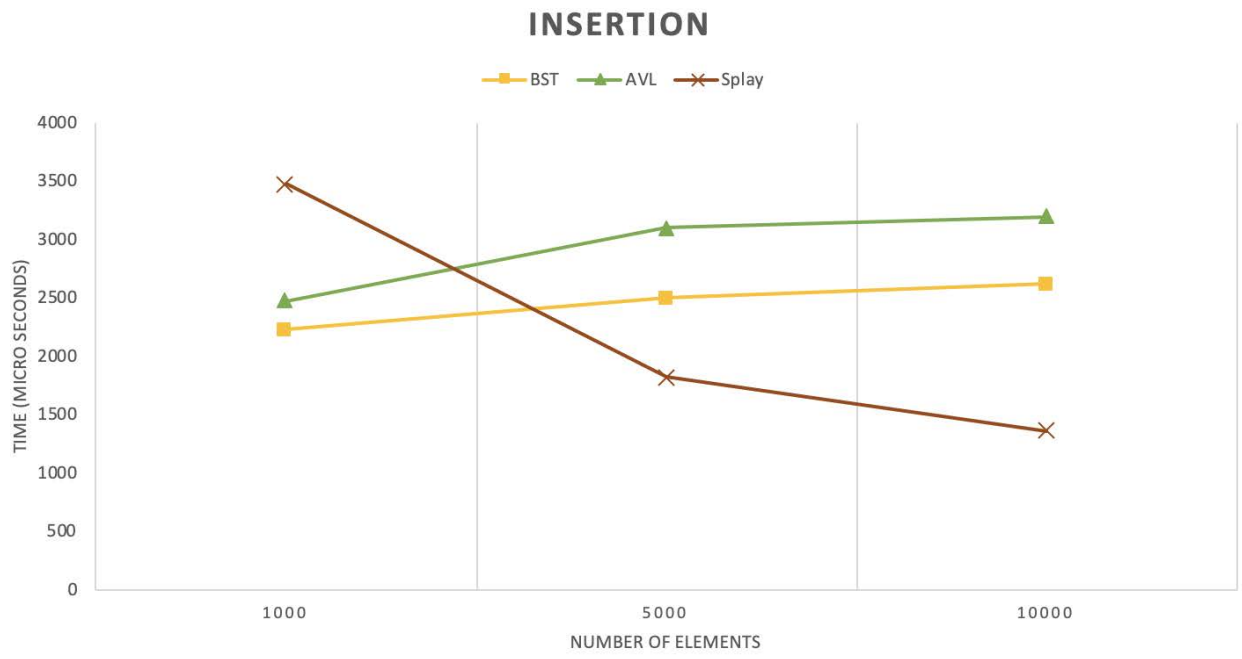
In our analysis, we compute the maximum time required for a particular node for a particular operation.

4. Visualization of Analysis

4.1. Operation Costs on a varying range size of data

(Input files- input2.txt, input3.txt, input4.txt)

4.1.1. Insertion Costs

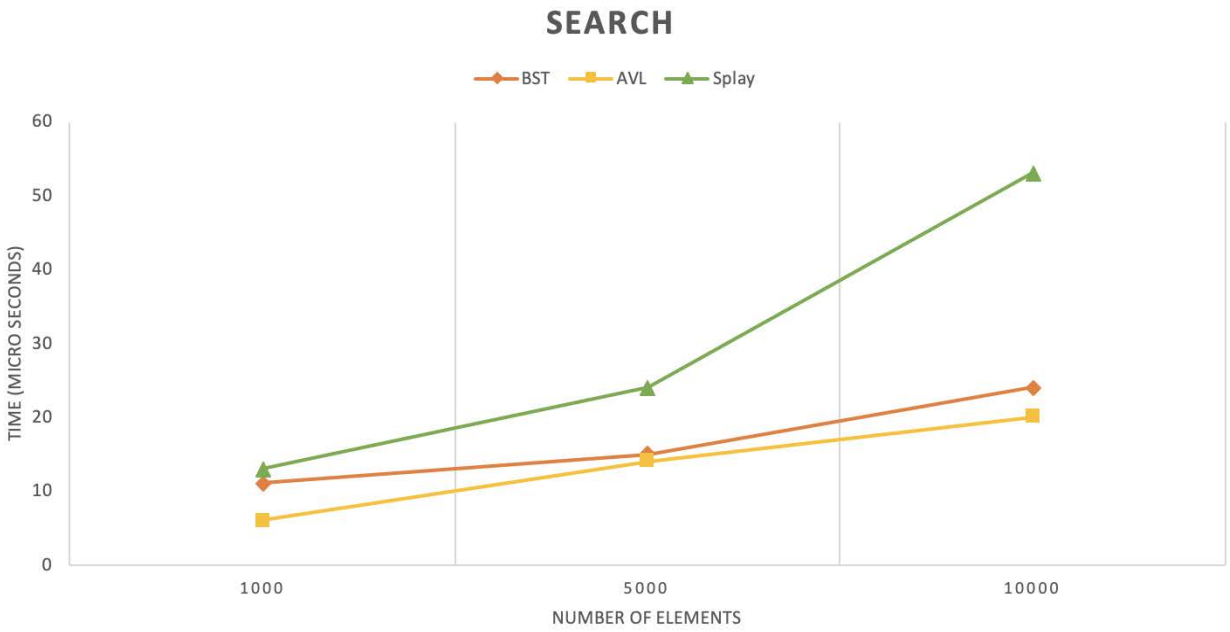


- Here we compare the worst case insertion time for BST, AVL and Splay trees on datasets of different sizes containing items in random order.
- Binary Search Tree performs insertions much faster in a small dataset than AVL and Splay trees, whereas for splay tree performs the best for bigger datasets.
- AVL tree will always have a greater time for insertion compared to BST as it might require additional time to perform rotations.
- As the number of elements increase, the time for insertion for BST and AVL is bound to increase as more nodes need to be visited. But for Splay tress as every node is splayed upon insertion, it is possible that fewer nodes might need to be visited to insert a new element.

4.1.2. Deletion Costs

- Here we compare the worst case deletion time for BST, AVL and Splay trees on datasets of different sizes containing items in random order.
- Binary Search Tree performs deletions much faster in a small dataset than AVL and Splay trees, whereas for splay tree performs the best for bigger datasets.
- The same reasoning for insertion hold true here as well.
- AVL tree performs an additional balancing operation which is why it will always require more time than a BST.
- At every splay operation the tree reorders itself and hence for bigger datasets splay tree gives a better performance.

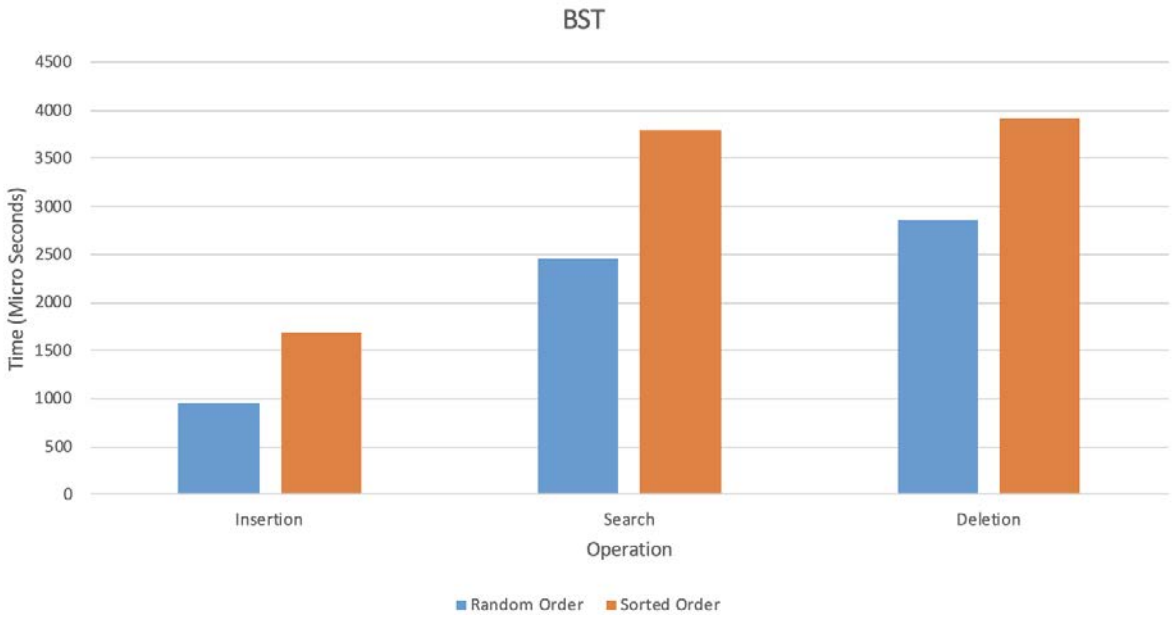
4.1.3. Search Costs



- Here we compare the worst case search time for BST, AVL and Splay trees on datasets of different sizes containing items in random order.
- AVL Tree performs the search operation faster than Binary Search Tree and Splay Tree on this randomly ordered dataset when distinct elements are selected to be searched in the respective trees.
- In our analysis, we use a different element to be searched every time. If the same element is searched multiple times, splay trees will give the best performance as compared to here where every element search will incur an additional cost for splaying.
- AVL tree is balanced and hence will always perform better than BST.

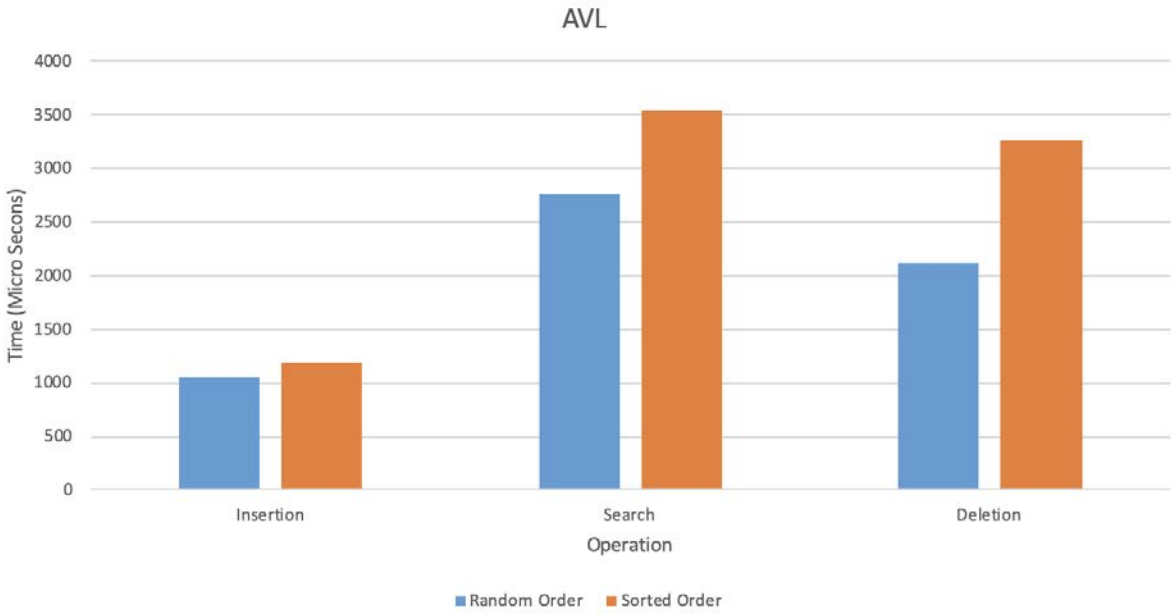
4.2. Sorted Order vs Random Order (Input file- Input1.txt)

4.2.1. Binary Search Tree



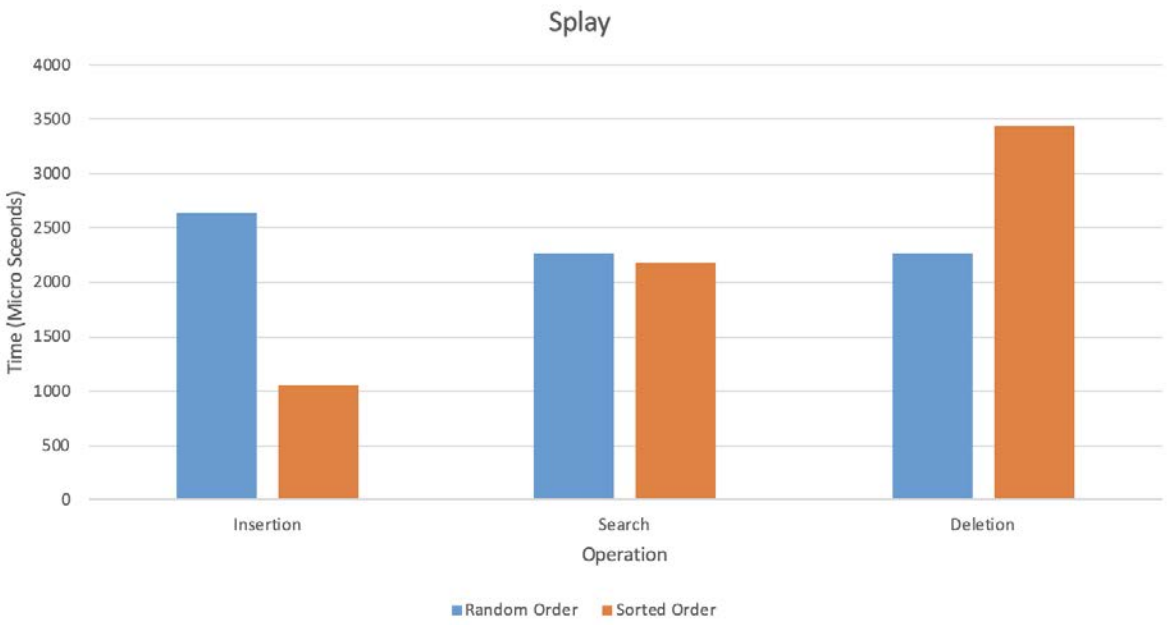
- Here, we compare the different operations for a BST on randomly ordered and sorted data containing the same data items.
- All Binary Search Tree operations take substantially more time for processing sorted dataset.
- For sorted data BST basically becomes a list and takes O(n) time.

4.2.2. AVL Tree



- Here, we compare the different operations for a AVL tree on randomly ordered and sorted data containing the same data items.
- All AVL tree operations take comparatively more time for processing sorted dataset.
- Each alternate insert would require one rotation for sorted data, which might also be required for randomly ordered data. Hence on average insertion time differs marginally.
- Here two lists will be formed, one containing values less than root node sorted in descending order and other containing values greater than root node sorted in ascending order.

4.2.3. Splay Tree



- Here, we compare the different operations for a Splay tree on randomly ordered and sorted data containing the same data items.
- Insertion time for sorted data will be significantly less than randomly ordered data as only one node will be visited and one rotation performed for each new element.
- Insertion in sorted order will form a list, however after splaying few times the tree will be reordered and access time will gradually decrease.
- In our analysis we compare the worst time for operation, hence maximum time to delete a node when the dataset is sorted is more as the tree will basically be a list.

5. Conclusion

a) For bigger datasets, splay trees will require less time for insertions and deletions as compared to BST and AVL.

b) AVL trees will take more time for insertions and deletions compared to BST, but have an advantage when more searches are performed or the data is in sorted order.

c) Splay trees provide best performance in a scenario where the recently accessed items have a higher probability to be accessed again.

d) BST and AVL perform better when the data is randomly ordered.

e) Using Splay trees is advised when the type of operations performed more frequently or ordering of data is not known, as it will provide better performance in the long run.

6. References

a. https://www.ics.uci.edu/~dillenco/compsci261p/notes/notes8-handout.pdf
b. https://www.ics.uci.edu/~dillenco/compsci261p/notes/notes10-handout.pdf
c. https://en.wikipedia.org/wiki/Binary_search_tree
d. https://en.wikipedia.org/wiki/AVL_tree
e. https://en.wikipedia.org/wiki/Splay_tree