Smitha Gurunathakrishnan Balagurunatan | 58328870

CompSci 261P Project #2: Binary trees and their variants

A **binary tree** is a tree data structure whose elements have at most 2 children – left and right child. Here I have implemented 4 variants of Binary Tree as below. All the 4 algorithms mentioned have been implemented in Java and analyzed.

- **Binary Search Tree (BST)**
- **Adelson-Velskii and Landis Tree (AVL) Tree**
- **Treaps**
- **Splay Trees**

1. Binary Search Tree (BST):

A BST has node-based binary tree data structure where the left subtree contains nodes with key lesser than the root and the right subtree contains nodes with key greater than the root.

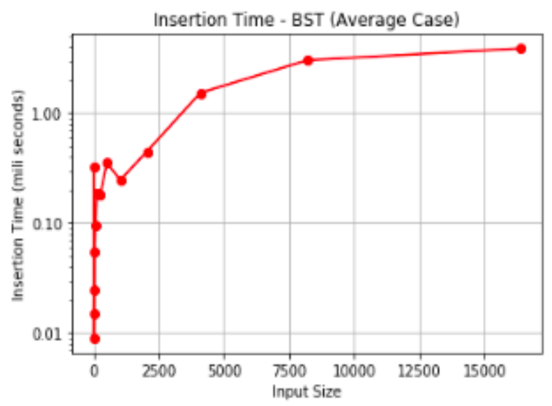
This is a naïve implementation without any balancing, where we simply iterate checking for lesser or greater condition and inserting into the tree. Pseudocode is as below –

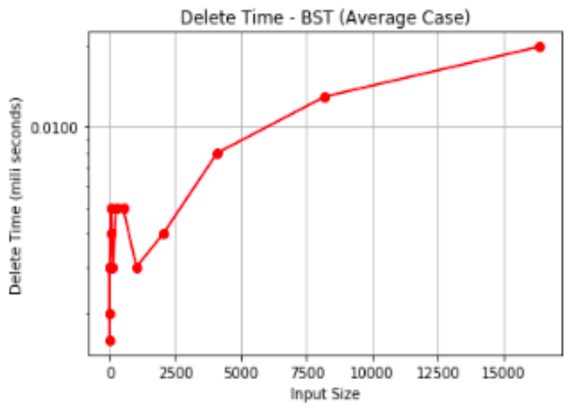
```
Void create ()
        //Initialize BST data structure and it’s root to null.

Node insert (int key, Node root)
        if (root == null)
                return (new Node(key));
        if (key < root.val) root.left = insert (key, root.left);
        if (key > root.val) root.right = insert (key, root.right);
        return root;

Node search (int key, Node root)
        if (root == null) return null;
        if (root.val == key) return root;
        if (key < root.val) return search (key, root.left);
        return search (key, root.right);

Node delete (int key, Node root)
        if (root == null) return root;
        if (key < root.val) root.left = delete (key, root.left);
        else if (key > root.val) root.right = delete (key, root.right);
        else
                if (root.left == null)  return root.right;
                else if (root.right == null) return root.left;
                root.val = minValue(root.right);
                root.right = delete (root.val, root.right);
        return root;
```

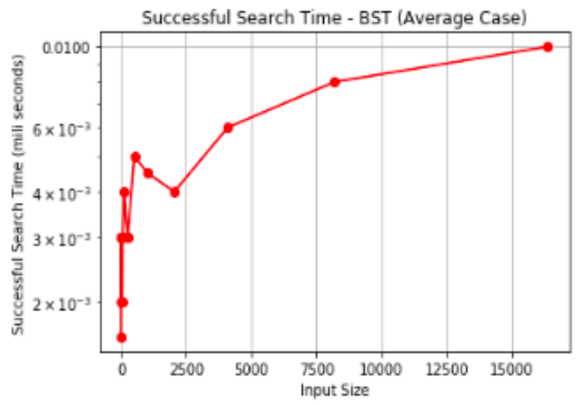
Time complexity & empirical performance analysis of BST:

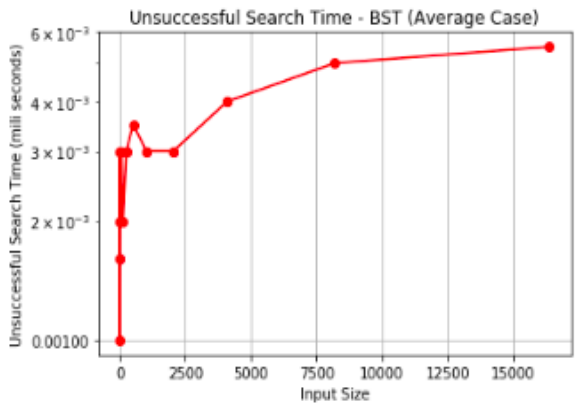
The **Best and Average case** time complexity **for Insert, Search and Delete** is **O(log n).** This occurs if elements inserted are random. Random elements of sizes 2 to 16000 elements are inserted and results of

Insert, Search, Delete and Unsuccessful search are observed as below. We can see from the graph that all the mentioned operations take log n time, as the **graphs are in log form**.

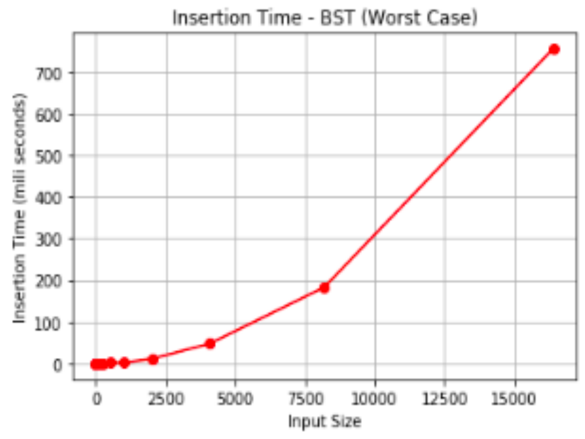


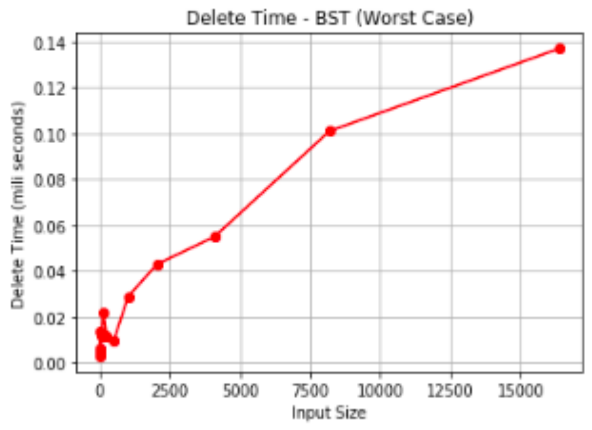


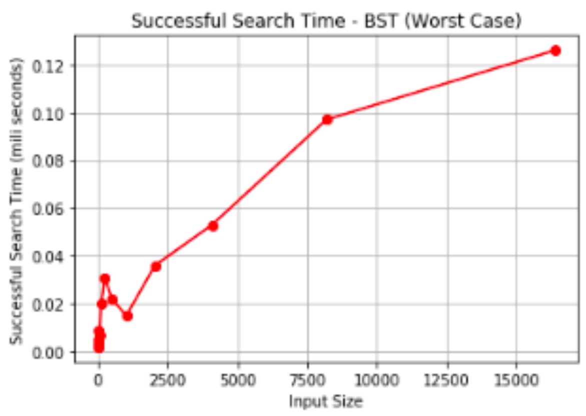


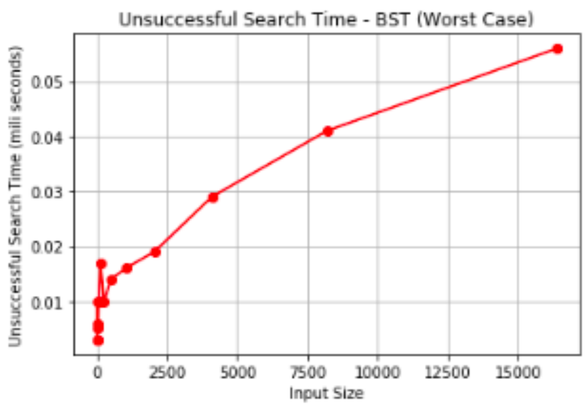


If the elements inserted are in sorted order, then the **worst-case time complexity is O(n)** for **insert/delete/search operations** as elements being inserted are restricted to either right/left subtree only. The height of the tree becomes the number of elements being inserted. Search happens similar to a linear search and hence complexity and space of this skewed tree is O(n). The graphs obtained are also in Linear form.









2. AVL Tree:

AVL tree is a height-balanced Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. No matter what input is given, height of the tree remains **O(log n)** after every insertion and deletion and doesn't become O(h) like BST.

It achieves this by performing 4 types of rotation in-order to balance the tree based on its height-

a) y is left child of root and x is left child of y (Left Left Case)
b) y is left child of root and x is right child of y (Left Right Case)
c) y is right child of root and x is right child of y (Right Right Case)
d) y is right child of root and x is left child of y (Right Left Case)

Pseudocode for rotations is as below –

```
Node rotateLeft (Node n)
        Node down = n.right;
        Node toNull = down.left;
        down.left = n;
        n.right = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
        return down;

Node rotateRight (Node n)
        Node down = n.left;
        Node toNull = down.right;
        down.right = n;
        n.left = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
        return down;

Node doubleRotateLeftRight (Node n)
        n.left = rotateLeft(n.left);
        return rotateRight(n);

Node doubleRotateRightLeft (Node n)
        n.right = rotateRight(n.right);
        return rotateLeft(n);
```

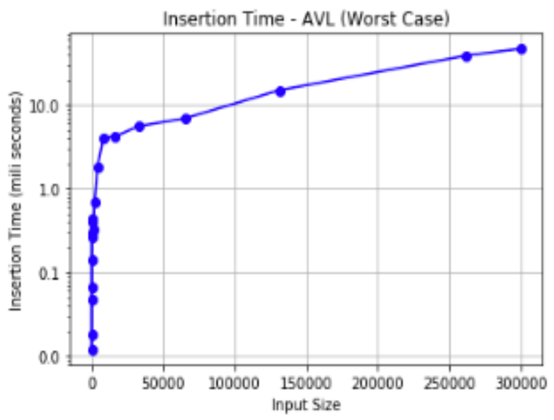
Pseudocode for insert, delete, search, create is as below –

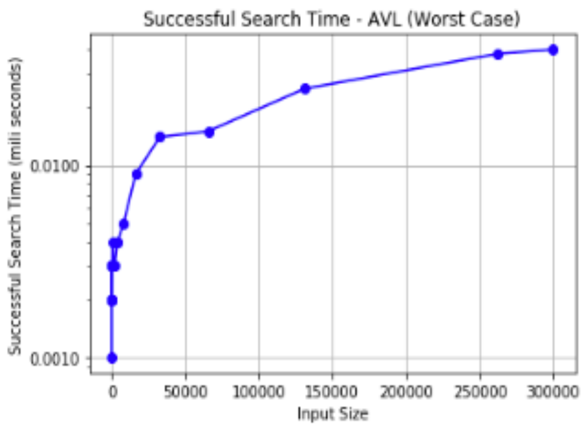
```
Void create ()
```

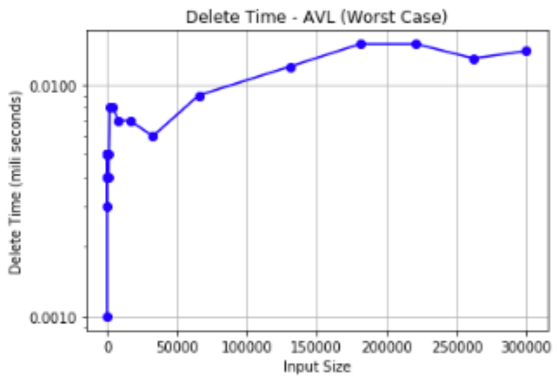
```
        //Initialize AVL data structure and it's root to null.

Node insert (int key, Node n)
        if (key < n.val)
                n.left = insert  (n.left,key);
        if (key > n.val)
                n.right = insert  (n.right,key);
        n.height = 1 + Math.max(height(n.left), height(n.right));
        int b = isBalanced(n);
        if (b > 1 && key < n.left.val)
                return rotateRight(n);
        if (b < -1 && key > n.right.val)
                return rotateLeft(n);
        if (b > 1 && key > n.left.val)
                return doubleRotateLeftRight(n);
        if (b < -1 && key < n.right.val)
                return doubleRotateRightLeft(n);
        return n;

Node search (int key, Node root)
        if (root.val == key) return root;
        if (key < root.val) return search (key, root.left);
        return search (key, root.right);

Node delete (int key, Node root)
        if (key < n.val) n.left = delete  (n.left, key);
        else if (key > n.val) n.right = delete  (n.right, key);
        else  : if (n.left == null)  n= n.right;
                else if (n.right == null) n= n.left;
                else :  n.val = minValue(n.right);  // takes the min value from left sub-tree
                        n.right = delete(n.right,n.val);
```

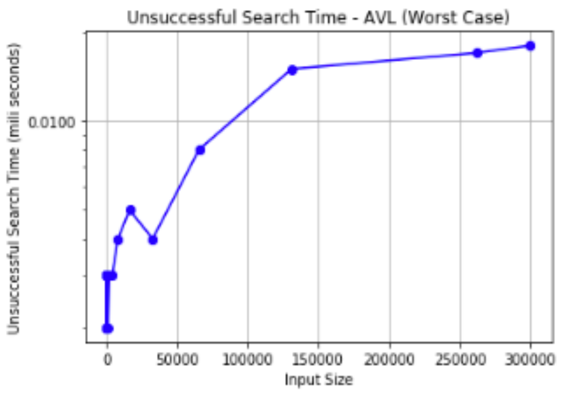
Time complexity & empirical performance analysis of AVL:

The **best/average/worst case time complexity** of insert/delete/search operation is O(log n). All rotations, getting and updating the heights take constant time. Since AVL tree is balanced through rotations, the height is O (log n) regardless of the nature of input elements. I have proved this by inserting 2 to 3lakhs sorted elements and the graphs obtained from this worst-case scenario is in log n form. Thus the **insert/delete/search operation is O (log n).**









3. Treaps (BST + Heap):

It's a randomized alternative to conventional BST. The time complexity of insert/delete/search operation is O(log n).

Each node contains:
a. Key value, child pointer (standard binary tree stuff)
b. A priority value - Assigned when node is created, never changes. It is chosen as a random number in range 1 to 100 in this implementation.

The key is inserted and organized in the tree following these 2 constraints:
a. Binary search tree property with respect to key values is maintained.
b. Heap-ordered tree with respect to priority values, max-heap property is maintained in this implementation.

Rotations similar to AVL are performed here if insert/delete operation violates BST or max-heap property. Pseudocode for rotations is as below –

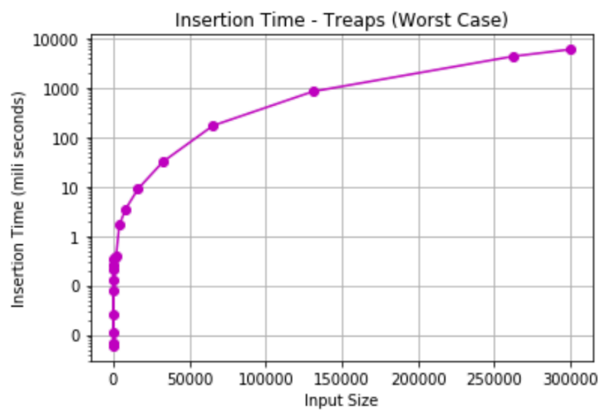
```
Node rotateLeft (Node n)
        Node down = n.right;
        Node toNull = down.left;
        down.left = n;
        n.right = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
        return down;

Node rotateRight (Node n)
        Node down = n.left;
        Node toNull = down.right;
        down.right = n;
        n.left = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
```

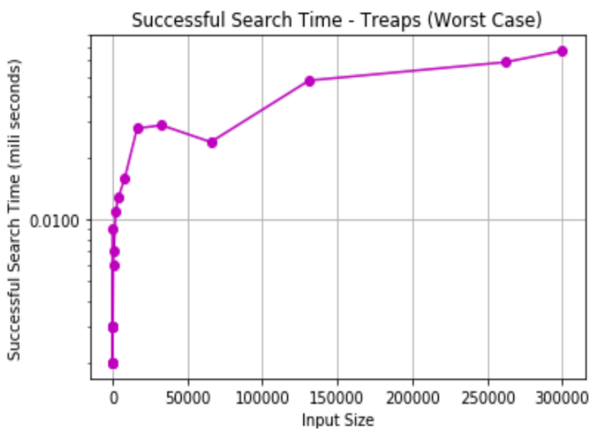
Pseudocode for insert, delete, search, create is as below –

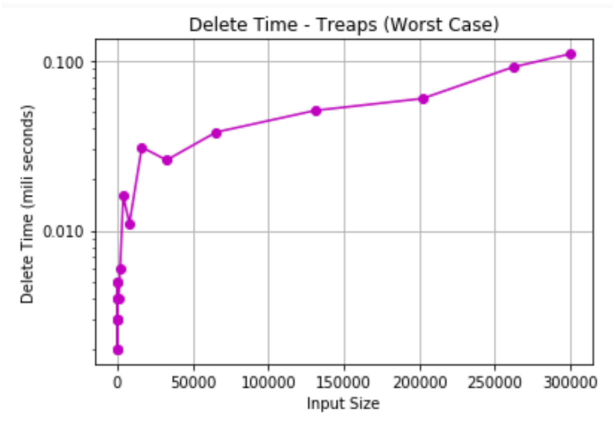
```
Void create ()
        //Initialize Treap data structure and it's root, priority to null.

Node insert (int key, Node n)
        if (key < n.val) {
                n.left = insert (n.left,key);
                if (n.left.priority > n.priority) n = rotateRight(n);
        }
        else if (key > n.val) {
                n.right = insert (n.right,key);
                if (n.right.priority > n.priority) n = rotateLeft(n);
        }
        return n;

Node search (int key, Node root)
        if (root.val == key) return root;
        if (key < root.val) return search (key, root.left);
        return search (key, root.right);

Node delete (int key, Node root)
        if (key < root.val) root.left = delete (key, root.left);
        else if (key > root.val) root.right = delete (key, root.right);
        else if (root.left == null)  root = root.right;
        else if (root.right == null) root = root.left;
        else if (root.right.priority > root.left.priority) {
                root = rotateLeft(root);
                root.left = delete  (key,root.left);}
        else :  root = rotateRight(root);
                root.right = delete  (key,root.right);}
return root
```

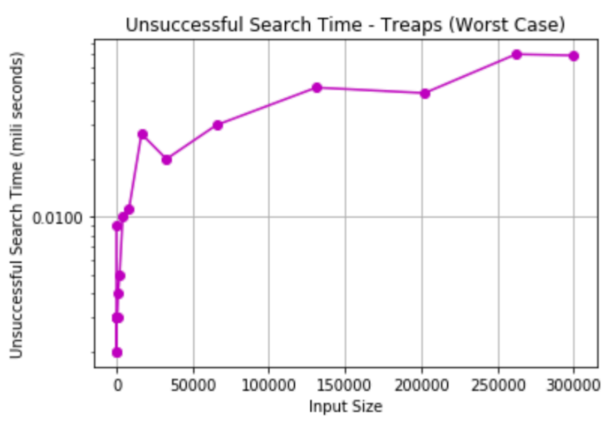
Time complexity & empirical performance analysis of Treaps:

The time complexity for **best/average/worst case is O(log n)** for **insert/delete/search operations** as the tree's height never becomes equal to number of elements inserted (O(h)). This is achieved by an additional randomization of priority values that ensures to balance the tree by maintaining max-heap property in addition to BST. Others checking and updating priority, roations take O(1) time. The graph has been plotted for 2 to 3lakhs sorted elements, where priority for each is randomized [1 to 100 range] and the graphs obtained from this worst-case scenario is in log n form.









4. **Splay Trees:**

Splay is like BST but performs an operation called splaying which performs tree rotations and brings the last/recently accessed element up to the root. It is mainly used in cache implementations. The rotations here are similar to AVL rotations. The time complexity of insert/delete/search operation is O(log n). Pseudocode for rotations is as below –

```
Node rotateLeft (Node n)
        Node down = n.right;
        Node toNull = down.left;
        down.left = n;
        n.right = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
        return down;

Node rotateRight (Node n)
        Node down = n.left;
        Node toNull = down.right;
        down.right = n;
        n.left = toNull;
        n.height = 1+ Math.max(height(n.left),height(n.right));
        down.height = 1+ Math.max(height(down.left),height(down.right));
```

Pseudocode for Splay operation is as below. It typically brings the recently accessed element to the top without disturbing the BST property.

```
Node splay(Node root, int key)
        if (key < root.val)  // Key lies in left subtree
        if (root.left == null) return root;
        if (key < root.left.val)   // Zig-Zig (Left Left)
            root.left.left = splay(root.left.left, key);
            root = rotateRight(root);
        else if (key > root.left.val)  // Zig-Zag (Left Right)
            root.left.right = splay(root.left.right, key);
            if (root.left.right != null)
                        root.left = rotateLeft(root.left);

        return (root.left == null)? root: rotateRight(root);

    else  // Key lies in right subtree
        if (root.right == null) return root;
        if (key > root.right.val)  // Zag-Zag (Right Right)
```

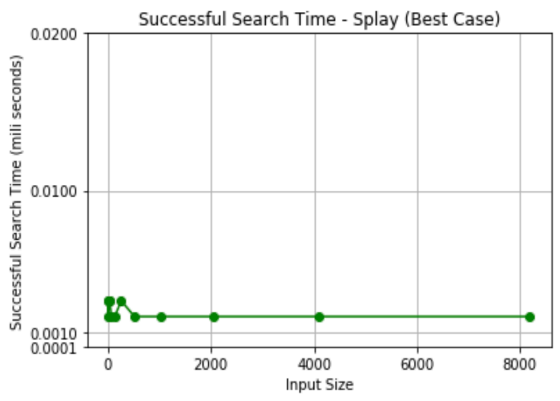
```
                root.right.right = splay(root.right.right, key);
                root = rotateLeft(root);
            else if (key < root.right.val)  // Zag-Zig (Right Left)
                root.right.left = splay(root.right.left, key);
                if (root.right.left != null)
                    root.right = rotateRight(root.right);
            return (root.right == null)? root:rotateLeft(root);
```

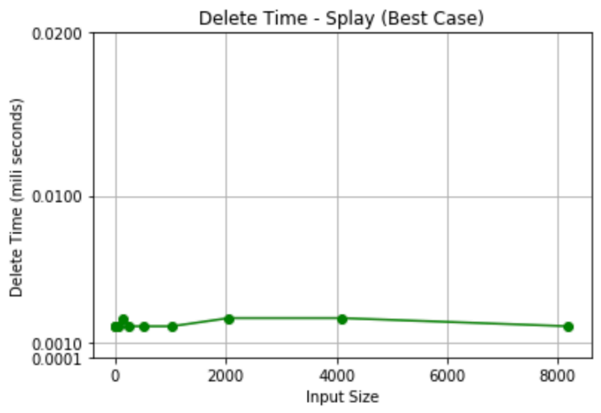
Pseudocode for insert, delete, search, create is as below –

```
    Void create ()
        //Initialize Splay data structure and it's root to null.

    Node insert (int key, Node root)
        root = splay (root, key);  // brings newly inserted element to top
        Node n = new Node(key);
        if (key < root.val)
                n.right = root;
                n.left = root.left;
                root.left = null;
        else
                n.left = root;
                n.right = root.right;
                root.right = null;
        return n;

    Node search (int key, Node root)
        root = splay(root, key);   // brings the search element to top

    Node delete (int key, Node root)
        Node x;
        if (root == null) return null;
        root = splay(root, key);    // brings element to be deleted to top
        if (root.val != key) return root;
            if (root.left == null)
                root = root.right;
            else
                x = root.right;
                root = root.left;
                root = splay(root, key);
                root.right = x;
            return root;
```

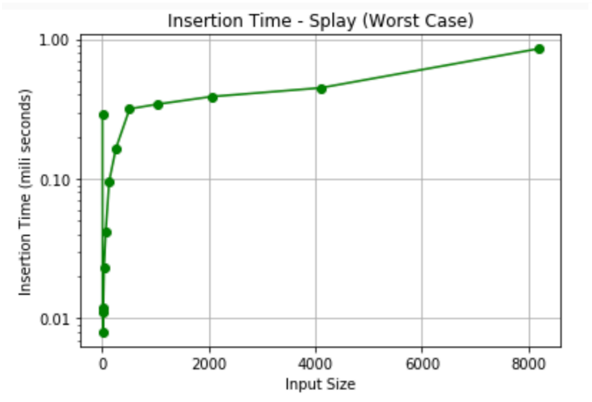
Time complexity & empirical performance analysis of Splay Trees:

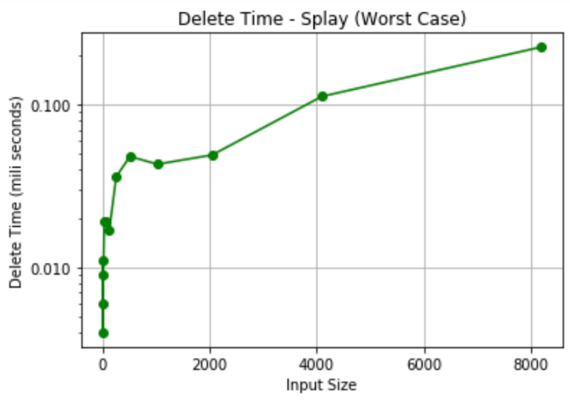
The **Best-case** time complexity of Treaps is **O(1)** - **Search/Delete** operations and **O(log n**) - **Insert operation**. This occurs in a case where the element to be found/deleted is the recently accessed element. I have taken a case where I have inserted 8100 elements and I'm trying to search/delete the last element I inserted. The graph is as follows –

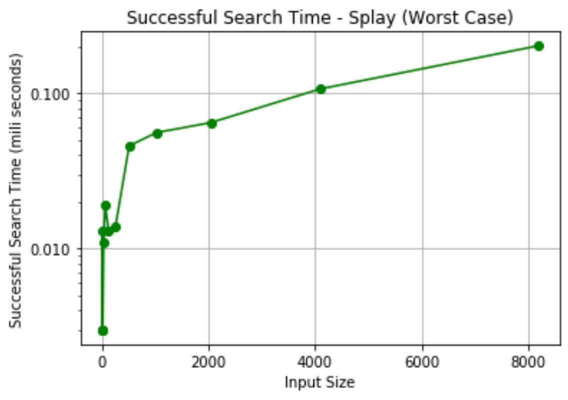


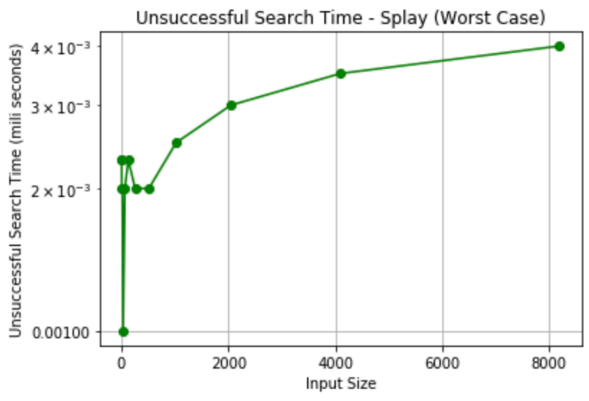


The **Average and worst-case** time complexity of Treaps is **O(log n**) for **insert/delete/search operations.** A range of 2 to 8100 sorted elements have been inserted and graph has been plotted and observed to be in log form.





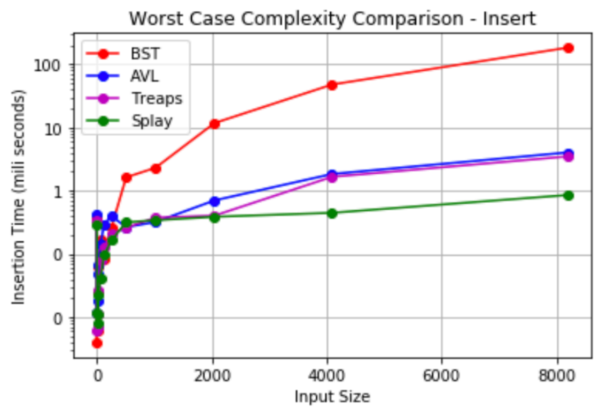




Experimental Analysis

Comparison between Worst-Case Time Complexity of all the 4 Trees:

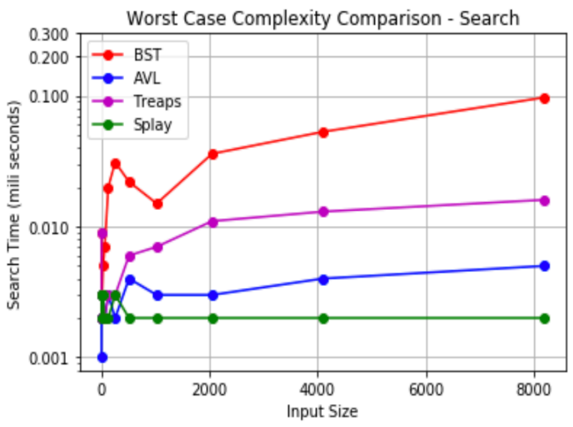
Insert – We see that **splay** trees performs **best** in worst case insertion of 2-8000 sorted elements as it is balanced followed by Treaps and AVL (O(log n)) and **BST** is the **worst** performer as its not balanced (O(n)).

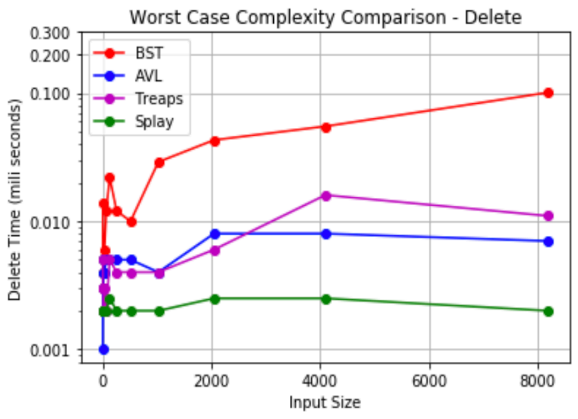


Search – Splay tree again wins by performing **best** as splays the recent element to top and also balances with rotation. AVL and Treaps follow the next all having complexity of O(log n). **BST** performs **worst** as O(n).

Delete – Splay performs **best** for deletion similar to search due to its rotation and splaying property. AVL and Treaps follow next with (O(log n)) and **BST** performs **worst** with O(n) complexity.

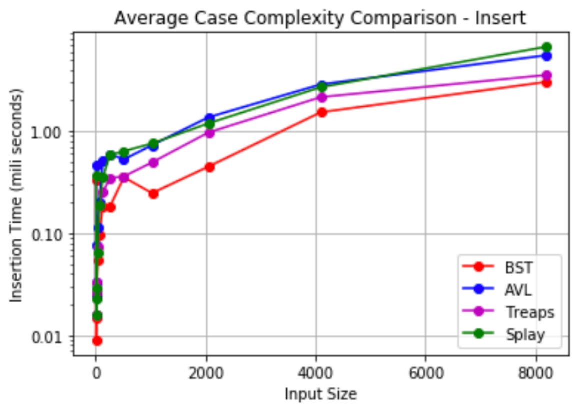
To conclude, splay trees proves to be the phenomenal and BST the worst performer for the worst-case scenario where elements inserted are in sorted order.

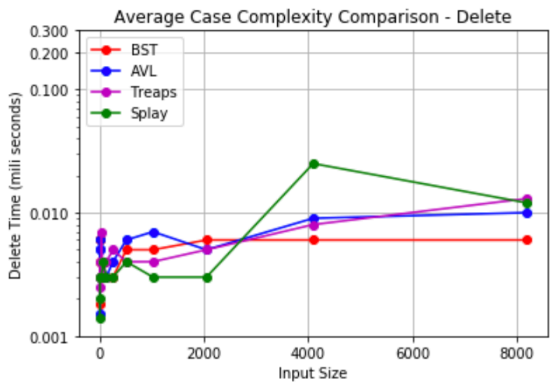




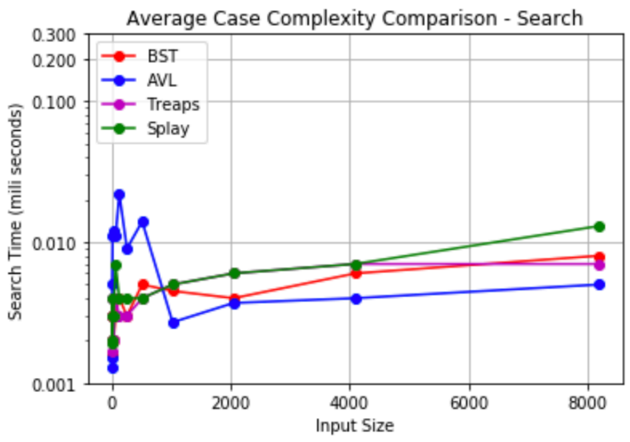
<u>Comparison between Average-Case Time Complexity of all the 4 Trees:</u>

Insert & Delete – We see that **BST** performs **best** in insertion/deletion in 2-8000 random elements as there is no overhead of balancing after operations. Treaps, which invokes rotations based on BST, heap order and AVL, which invokes rotations based on height regularly follows next. **Splay** Tree is the **worst** performer as it splays/rotates for every element insert/delete which becomes a major over-head of time. All the operations take **O(log n)**





Search – AVL tree again by performing **best** as AVL is a perfect height balanced tree and search is divided correctly. Treaps and BST follow next and **Splay** is the **worst** performer as there is **overhead** of balancing for every find in Splay tree. All trees search take O(log n) time for average case.


Average Case Complexity Comparison - Search
BST
AVL
Treaps
Splay
Search Time (mili seconds)
Input Size