Pseudo-codes

Binary Search Tree:

The Binary search tree is a special type of binary tree where the left child's value is less that the root's value and the right child's value is greater than the root's value.

To implement a binary search tree I will create a node class which will have 3 parameters, value (integer), left node and right node.

Following are the pseudo-codes for every method implemented in a Binary Search Tree:

Create:

Initialize the root of the binary search tree to null.

```
def create():
        root = null
```

Insert:

The first time the insert element is called, the first element is inserted as the root. Every other time this function is called, a comparison is made with the root element, and if it is lesser than the root element, the function is recursively called with the parameter (root.left), since the BST property states that the element lesser than a node go on to its left side. If the node which is inserted is greater than the root, then the function is recursively called with the root.right.

```
def insert(root, k):
        if root == null:
                root = new node(k)
        else if k < root.k:
                insert(root.left, k)
        else:
                insert(root.right, k)
```

Search:

Find the value to be searched and return true if the value is found and return false if value not found. We will take the advantage of BST's property to search the value.

```
def search(root, k):
        if root == null:
                ret false
        if root.k == k:
                ret true
        if k < root.k:
                search(root.left, k)
        else:
                search(root.right, k)
```

Delete:

The element to be deleted is first identified in the tree. There are 3 types of nodes which are being deleted. Every type of node has its own process of deletion:

1. Node is a leaf: If the node to be deleted is leaf, we just delete it.
2. Node has 1 child: If the node to be deleted has one child, then delete the node and make its child the node.
3. Node has 2 children: If the node to be deleted has 2 children, then replace the node with the node found by in-order predecessor traveling.

```
def delete(root, k):
        if root == null:
                ret root
        if root.k == k:
                ret true
        if k < root.k:
                delete(root.left, k)
        else if k > root.k:
                delete(root.right, k)
        else:
                if root.left == null:
                        root = root.right
                else if node.right == null:
                        root = root.left
                else:
                        child = root.left
                        while child.right != null:
                                child = child.right
                        root.k = child.k
                        delete(root.left, child.k)
```

AVL Tree:

The AVL tree is a special type of binary tree where the left child's value is less that the root's value and the right child's value is greater than the root's value. Also the AVL tree is always balanced, i.e. the difference between the left height and right height of every node is never more than 1.

To implement an AVL tree I will create a node class which will have 5 parameters, value (integer), left node, right node, left height (integer) and right height (integer).

Following are the pseudo-codes for every method implemented in an AVL Tree:

Create:
Initialize the root of the AVL tree to null.

```
def create():
        root = null
```

Insert:
The first time the insert element is called, the first element is inserted as the root. Every other time this function is called, a comparison is made with the root element, and if it is lesser than the root element, the function is recursively called with the parameter (root.left), since the BST property states that the element lesser than a node go on to its left side. If the node which is inserted is greater than the root, then the function is recursively called with the root.right. After adding the new node, we balance the tree if it is unbalanced. We left rotate the node if its right height is greater

than the left height by more than 1 and we right rotate the node if its left height is greater than the right height by more than 1.

```
def insert(root, k):
        if root == null:
                root = new node(k)
        else if k < root.k:
                insert(root.left, k)
                root.leftHeight += 1
        else:
                insert(root.right, k)
                root.rightHeight += 1
        balance(root)
```

Search:
Find the value to be searched and return true if the value is found and return false if value not found. We will take the advantage of BST's property to search the value.

```
def search(root, k):
        if root == null:
                ret false
        if root.k == k:
                ret true
        if k < root.k:
                search(root.left, k)
        else:
                search(root.right, k)
```

Delete:
The element to be deleted is first identified in the tree. There are 3 types of nodes which are being deleted. Every type of node has its own process of deletion:

1. Node is a leaf: If the node to be deleted is leaf, we just delete it.
2. Node has 1 child: If the node to be deleted has one child, then delete the node and make its child the node.
3. Node has 2 children: If the node to be deleted has 2 children, then replace the node with the node found by in-order predecessor traveling.

After deleting the node, we balance the tree if it is unbalanced. We left rotate the node if its right height is greater than the left height by more than 1 and we right rotate the node if its left height is greater than the right height by more than 1.

```
def delete(root, k):
        if root == null:
                ret root
        if root.k == k:
                ret true
        if k < root.k:
                delete(root.left, k)
        else if k > root.k:
```

```
                delete(root.right, k)
        else:
                if root.left == null:
                        root = root.right
                else if node.right == null:
                        root = root.left
                else:
                        child = root.left
                        while child.right != null:
                                child = child.right
                        root.k = child.k
                        delete(root.left, child.k)
        balance(root)
```

Treap:

The Treap is a special type of binary tree which is a combination of a max-heap and a binary search tree. So, children of every node have less priority than the node itself and the left child's value is less that the root's value and the right child's value is greater than the root's value.

To implement a Treap I will create a node class which will have 4 parameters, value (integer), left node, right node and priority (integer with higher value will have higher priority).

Following are the pseudo-codes for every method implemented in a Treap:

Create:

Initialize the root of the Treap to null.

```
def create():
        root = null
```

Insert:

The first time the insert element is called, the first element is inserted as the root. Every other time this function is called, a comparison is made with the root element, and if it is lesser than the root element, the function is recursively called with the parameter (root.left), since the BST property states that the element lesser than a node go on to its left side. If the node's priority is lesser than the left child's priority then we right rotate the node. If the node which is inserted is greater than the root, then the function is recursively called with the root.right. Later, if node's priority is lesser than the right child's priority then we left rotate the node.

```
def insert(root, k):
        if root == null: root = new node(k)
        else if k < root.k:
                insert(root.left, k)
                if root.priority < root.left.priority:
                        rightRotate(root)
        else:
                insert(root.right, k)
                if root.priority < root.right.priority:
                        leftRotate(root)
```

Search:

Find the value to be searched and return true if the value is found and return false if value not found. We will take the advantage of BST's property to search the value.

```
def search(root, k):
        if root == null:
                ret false
        if root.k == k:
                ret true
        if k < root.k:
                search(root.left, k)
        else:
                search(root.right, k)
```

Delete:

The element to be deleted is first identified in the tree. There are 3 types of nodes which are being deleted. Every type of node has its own process of deletion:

1. Node is a leaf: If the node to be deleted is leaf, we just delete it.
2. Node has 1 child: If the node to be deleted has one child, then either rotate left or rotate right as per the priority of the child and make the node as the leaf node, then delete the leaf.
3. Node has 2 children: If the node to be deleted has 2 children, then replace the node with the child node having higher priority. After replacing the node, left rotate or right rotate the nodes as per the priority of the node.

```
def delete(root, k):
        if root == null:
                ret root
        if root.k == k:
                ret true
        if k < root.k:
                delete(root.left, k)
        else if k > root.k:
                delete(root.right, k)
        else:
                if root.right != null or root.left != null:
                                if root.left == null:
                                        leftRotate(root)
                                        delete(root.left, val)
                                else if root.right == null or root.left.priority > root.right.priority:
                                        rightRotate(root)
                                        delete(root.right, val)
                                else:
                                        leftRotate(root)
                                        delete(root.left, val)
                else: return null
```

Splay Tree:
The Splay Tree is a special type of binary tree which is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. So, children of every node have less priority than the node itself and the left child's value is less that the root's value and the right child's value is greater than the root's value. The most recently accessed node will always be the root of the tree. This property of splaying the most recently accessed node to the root is mostly used in cache memory implementation.

To implement a Splay Tree I will create a node class which will have 3 parameters, value (integer), left node and right node.

Following are the pseudo-codes for every method implemented in a Splay Tree:

Create:
Initialize the root of the Splay Tree to null.

```
def create():
        root = null
```

Insert:
The first time the insert element is called, the first element is inserted as the root. Every other time this function is called, a comparison is made with the root element, and if it is lesser than the root element, the function is recursively called with the parameter (root.left), since the BST property states that the element lesser than a node go on to its left side. If the node which is inserted is greater than the root, then the function is recursively called with the root.right. After inserting the new node, that node is splayed to the top as the root of the tree either by zig rotation, zig-zig rotation or zig-zag rotation.

```
def insert(root, k):
        if root == null: root = new node(k)
        else if k < root.k:
                insert(root.left, k)
        else:
                insert(root.right, k)
        splay(root)
```

Search:
Find the value to be searched and return true if the value is found and return false if value not found. We will take the advantage of BST's property to search the value. The searched node will be splayed to the top as the root of the tree either by zig rotation, zig-zig rotation or zig-zag rotation. So, next time if the same value is searched, it can be accessed quickly.

```
def search(root, k):
        if root == null:
                ret false
        if root.k == k:
                splay(root)
                ret true
        if k < root.k:
```

```
            search(root.left, k)
    else:
            search(root.right, k)
```

Delete:

The element to be deleted is first identified in the tree. There are 3 types of nodes which are being deleted. Every type of node has its own process of deletion:

1. Node is a leaf: If the node to be deleted is leaf, we just delete it.
2. Node has 1 child: If the node to be deleted has one child, then delete the node and make its child the node.
3. Node has 2 children: If the node to be deleted has 2 children, then replace the node with the node found by in-order predecessor traveling.

After deleting the node, the parent of the node is splayed to the top as the root of the tree either by zig rotation, zig-zig rotation or zig-zag rotation.

```
def delete(root, k):
    if root == null:
            ret root
    if root.k == k:
            ret true
    if k < root.k:
            delete(root.left, k)
    else if k > root.k:
            delete(root.right, k)
    else:
            if root.left == null:
                    root = root.right
            else if node.right == null:
                    root = root.left
            else:
                    child = root.left
                    while child.right != null:
                            child = child.right
                    root.k = child.k
                    delete(root.left, child.k)
            splay(root.parent)
```

Experimental Analysis:

Test Case 1: (Generic example)

Insert: (5, 11, 16, 10, 20, 4, 8, 9, 15, 1, 12)

Search: (20, 4, 17, 1, 6, 14, 11, 5)

Delete: (11, 10, 12, 8, 5, 15, 16, 20)

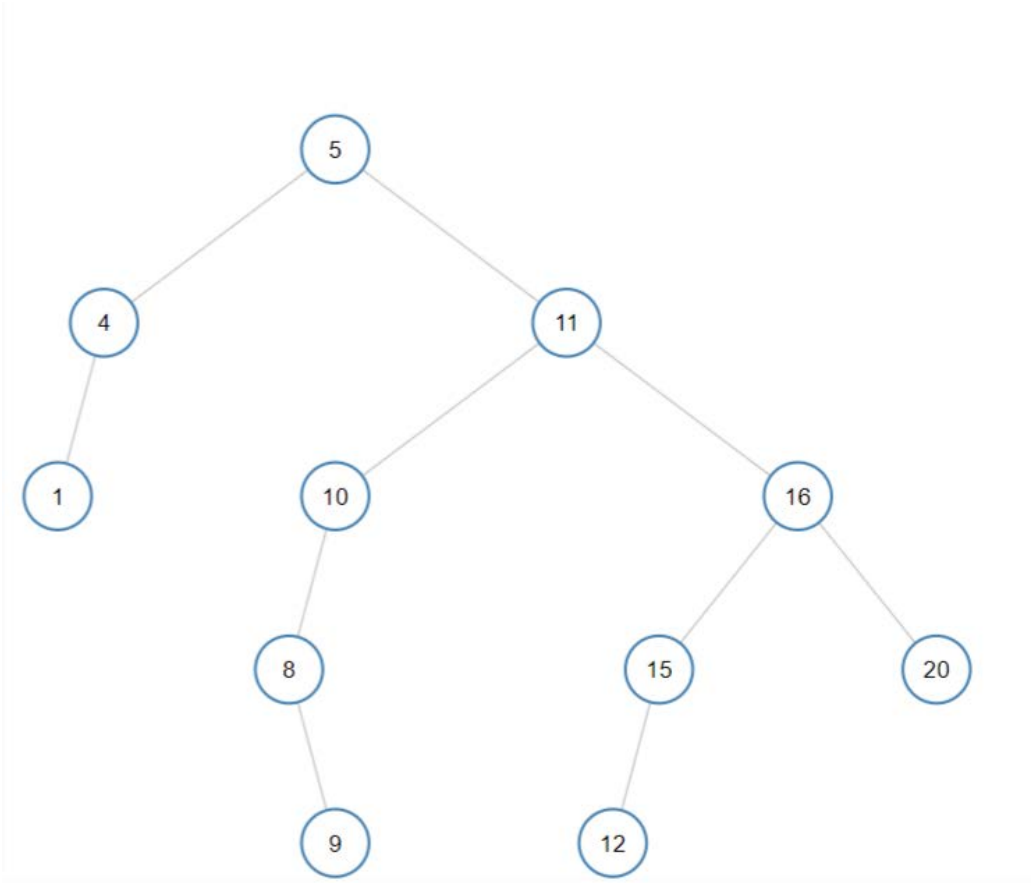
Binary Search Tree:

Insertions –

Time: 67526ns (11 inserts)

Level Order:

[[5], [4, 11], [1, null, 10, 16], [null, null, 8, null, 15, 20], [null, 9, 12, null, null, null]]



Search –

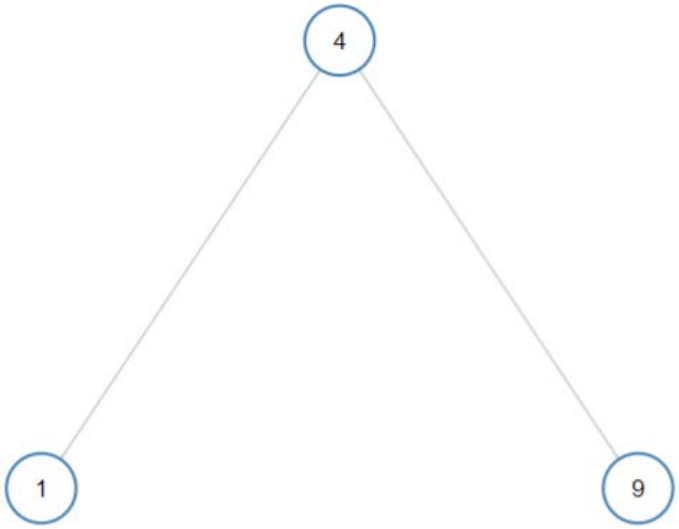
Time: 5884ns (8 searches)

Deletions –

Time: 9409ns (8 deletes)

Level Order:

[[4], [1, 9]]



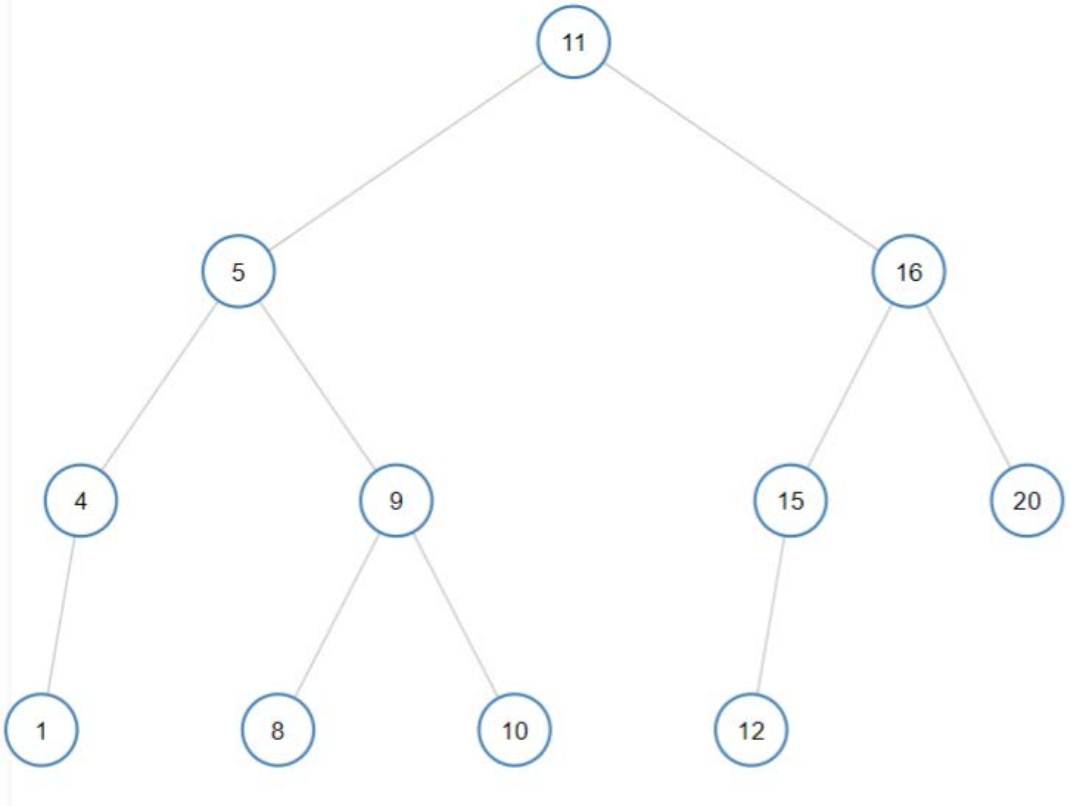
AVL Tree:

Insertions –

Time: 60769ns (11 inserts)

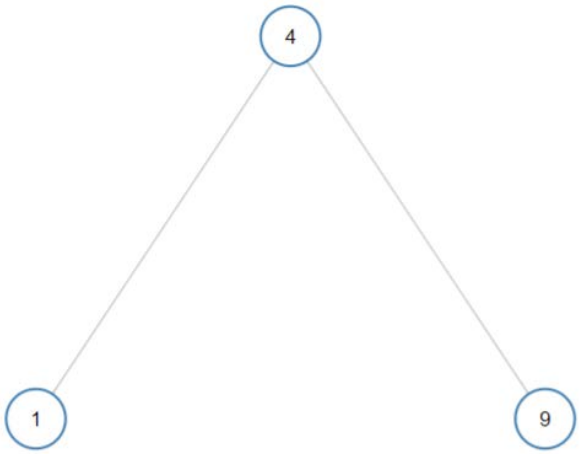
Level Order:

[[11], [5, 16], [4, 9, 15, 20], [1, null, 8, 10, 12, null, null, null]]



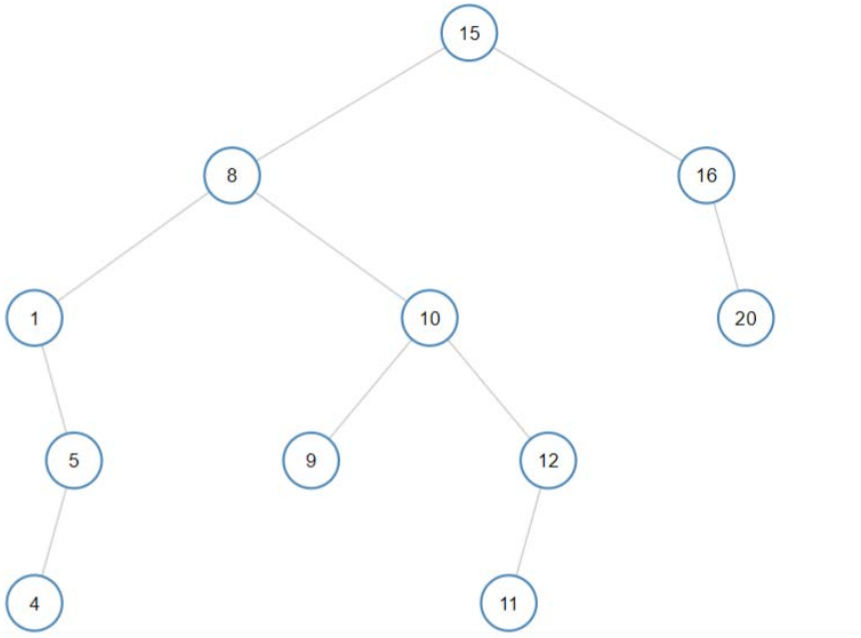
Search –
Time: 4704ns (8 searches)

Deletions –
Time: 7698ns (8 deletes)
Level Order:
[[4], [1, 9]]



Treap:

Insertions –
Time: 87284ns (11 inserts)
Level Order:
[[15], [8, 16], [1, 10, null, 20], [null, 5, 9, 12, null, null], [4, null, null, null, 11, null]]



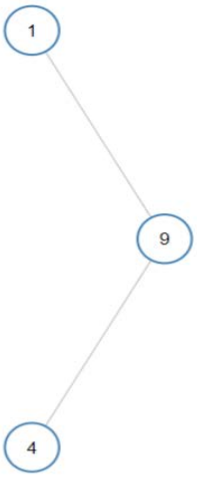
Search –
Time: 5132ns (8 searches)

Deletions –

Time: 7270ns (8 deletes)

Level Order:

[[1], [null, 9], [4, null]]



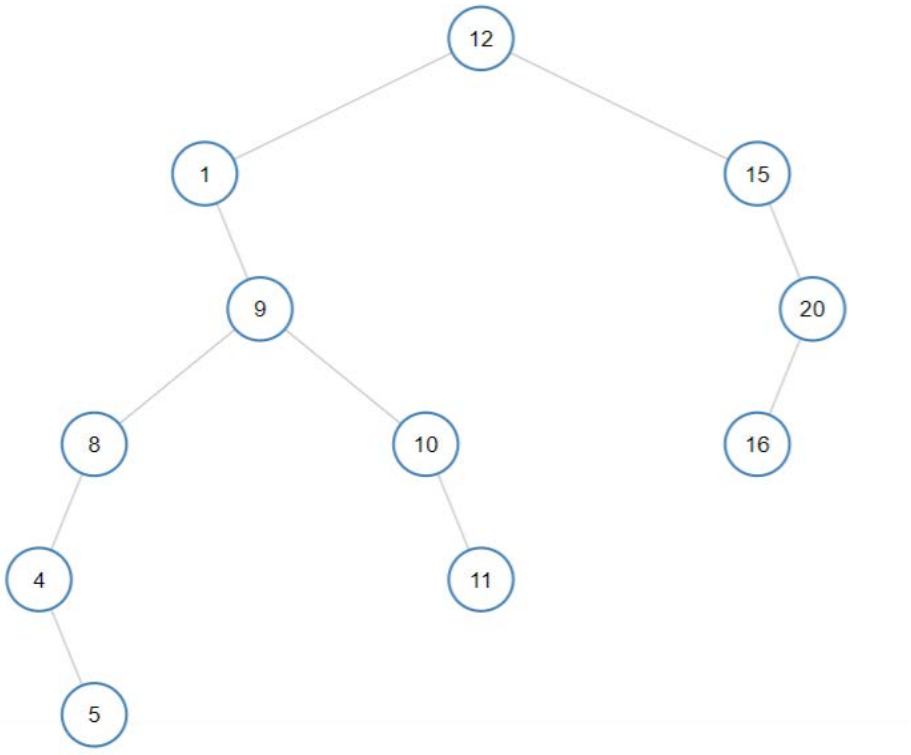
Splay Tree:

Insertions –

Time: 96094ns (11 inserts)

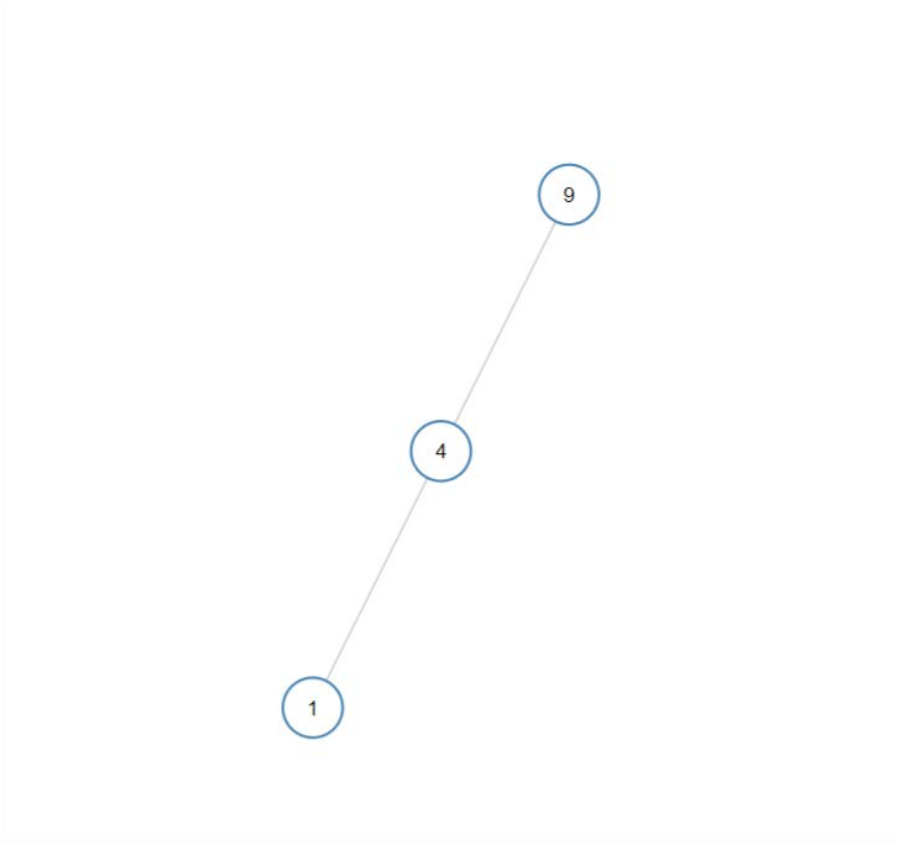
Level Order:

[[12], [1, 15], [null, 9, null, 20], [8, 10, 16, null], [4, null, null, 11, null, null], [null, 5, null, null]]

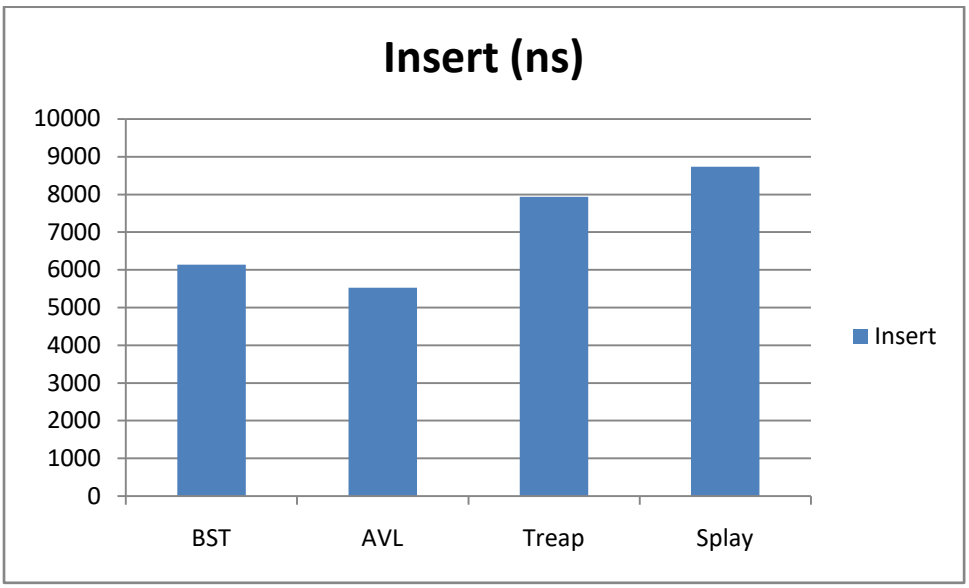


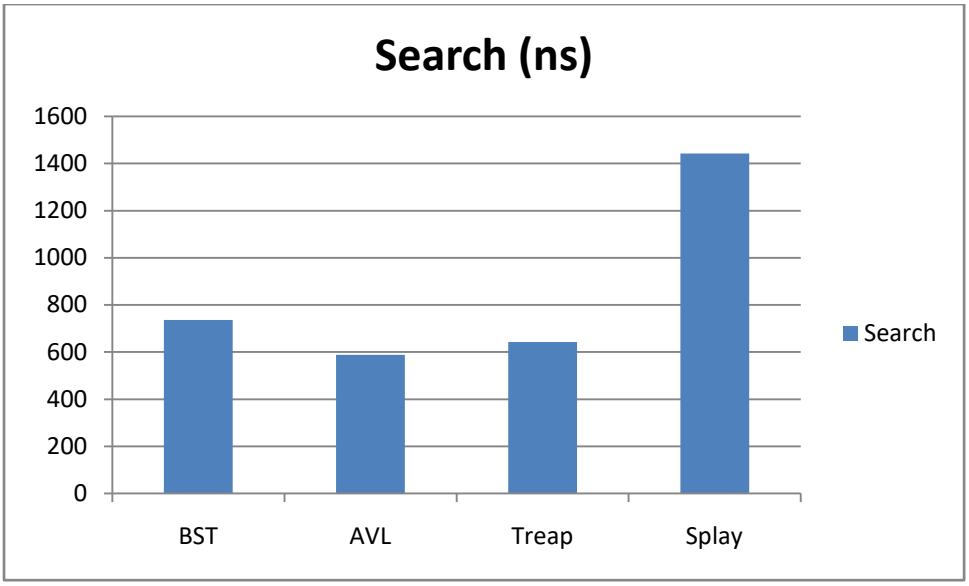
Search –
Time: 11547ns (8 searches)

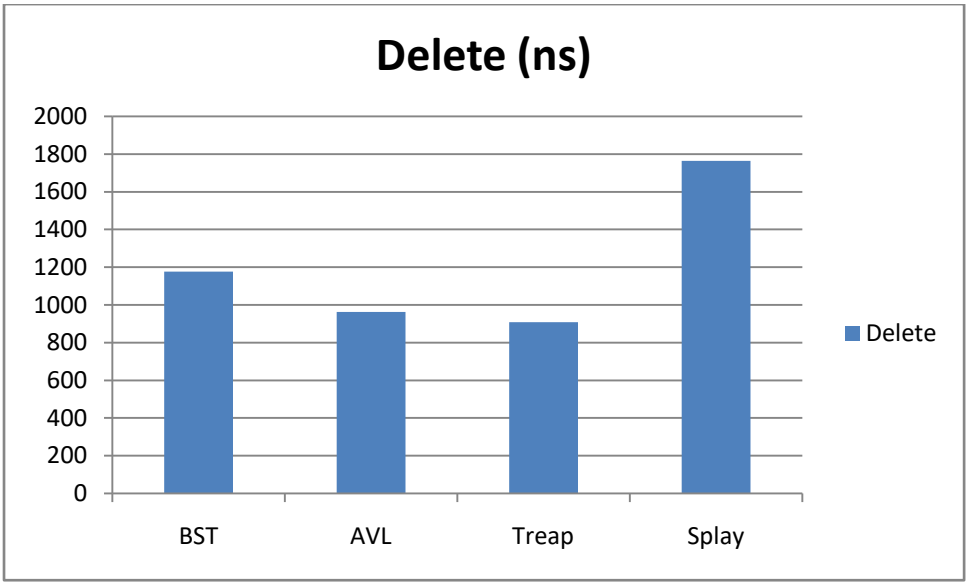
Deletions –
Time: 14112ns (8 deletes)
Level Order:
[[9], [4, null], [1, null]]



Graphs:







Observation:

While inserting the values, Splay Tree takes more time than others as it always splays the node inserted to the root of the tree and AVL Tree takes the least time as the tree is always balanced. For searching a value in a tree, Splay Tree takes more time than others as it always splays the node accessed to the root of the tree and AVL Tree takes the least time as the tree is always balanced. While deleting, again Splay Tree takes more time as it always splays the parent of the node deleted to the root of the tree and AVL Tree takes the least time as the tree is always balanced.

Test Case 2: (Sorted Insertion (skewed) example)

Insert: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

Search: (11, 10, 9, 8, 7, 6, 5, 4)

Delete: (11, 10, 9, 8, 7, 6, 5, 4)

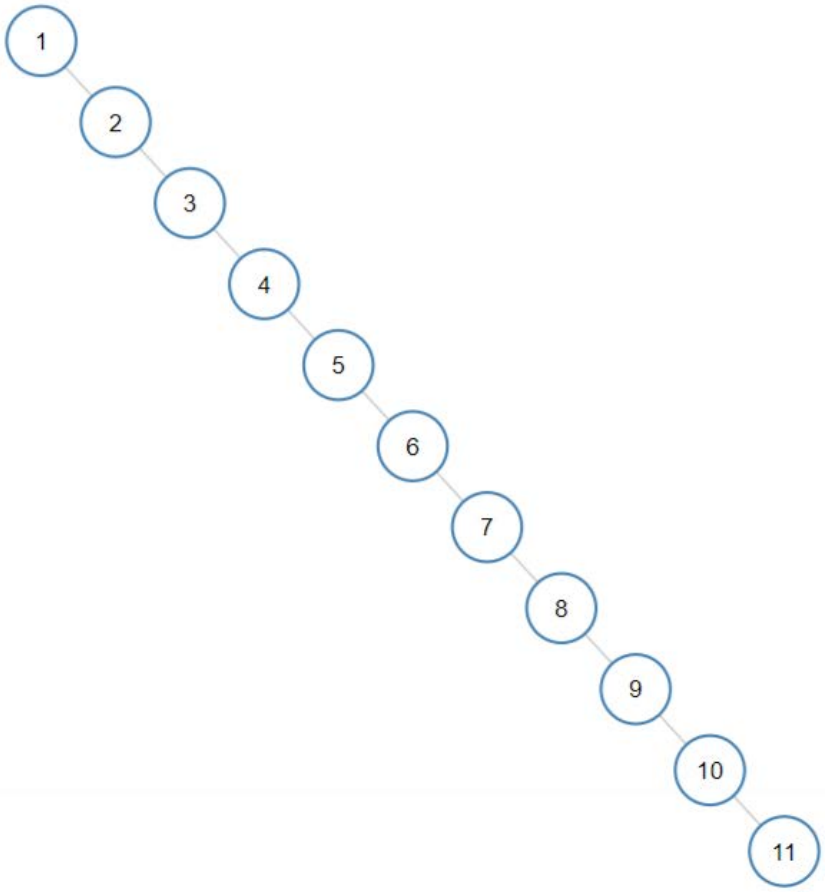
Binary Search Tree:

Insertions –

Time: 82665ns (11 inserts)

Level Order:

[[1], [null, 2], [null, 3], [null, 4], [null, 5], [null, 6], [null, 7], [null, 8], [null, 9], [null, 10], [null, 11]]



Search –

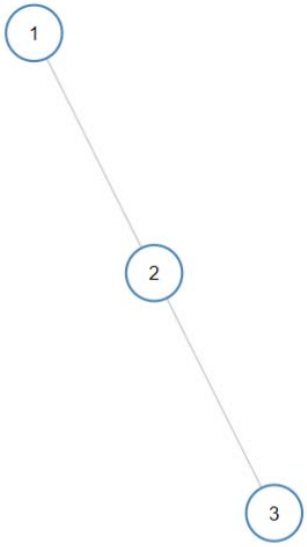
Time: 6124ns (8 searches)

Deletions –

Time: 11235ns (8 deletes)

Level Order:

[[1], [null, 2], [null, 3]]



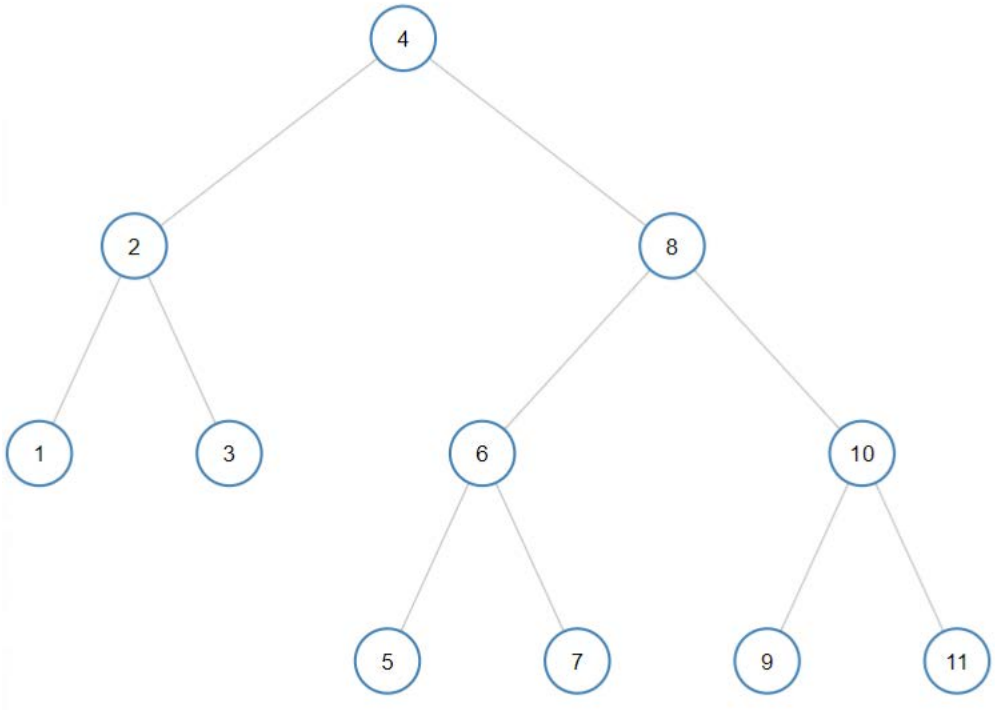
AVL Tree:

Insertions –

Time: 61240ns (11 inserts)

Level Order:

[[4], [2, 8], [1, 3, 6, 10], [null, null, null, null, 5, 7, 9, 11]]



Search –

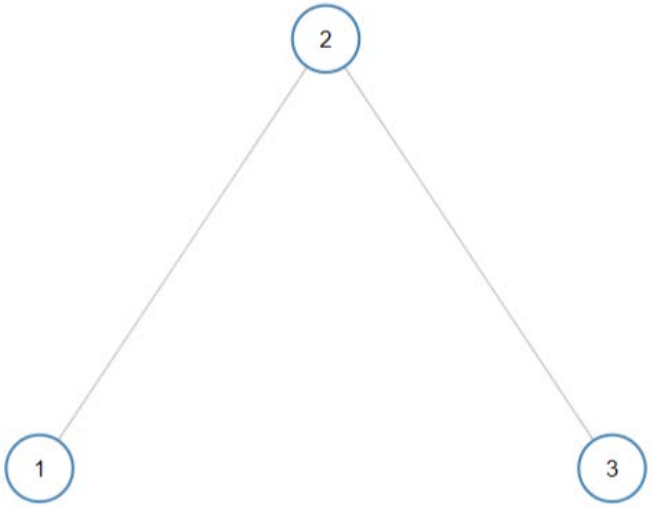
Time: 5025ns (8 searches)

Deletions –

Time: 6498ns (8 deletes)

Level Order:

[[2], [1, 3]]



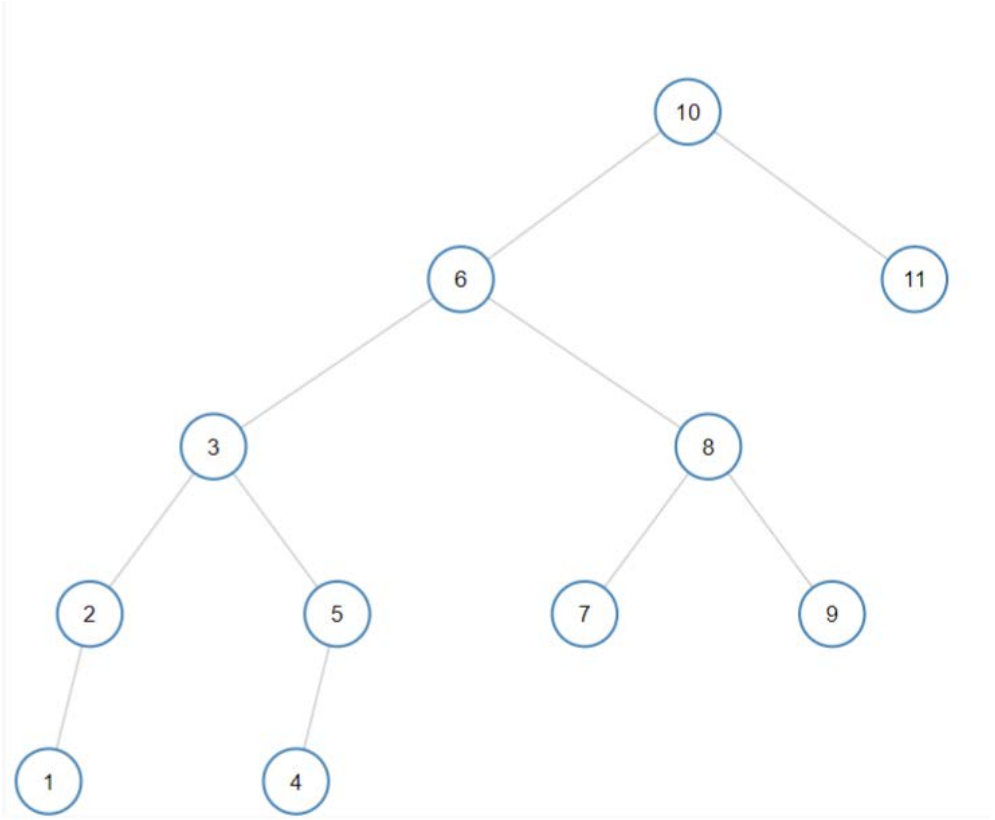
Treap:

Insertions –

Time: 93143ns (11 inserts)

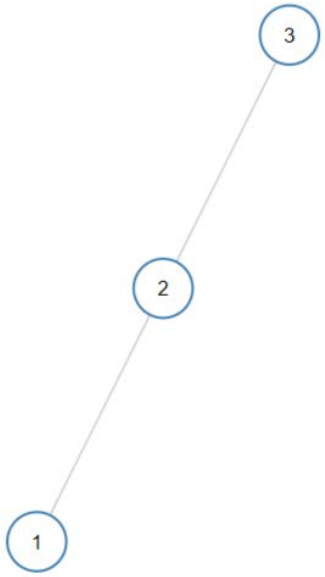
Level Order:

[[10], [6, 11], [3, 8, null, null], [2, 5, 7, 9], [1, null, 4, null, null, null, null, null]]



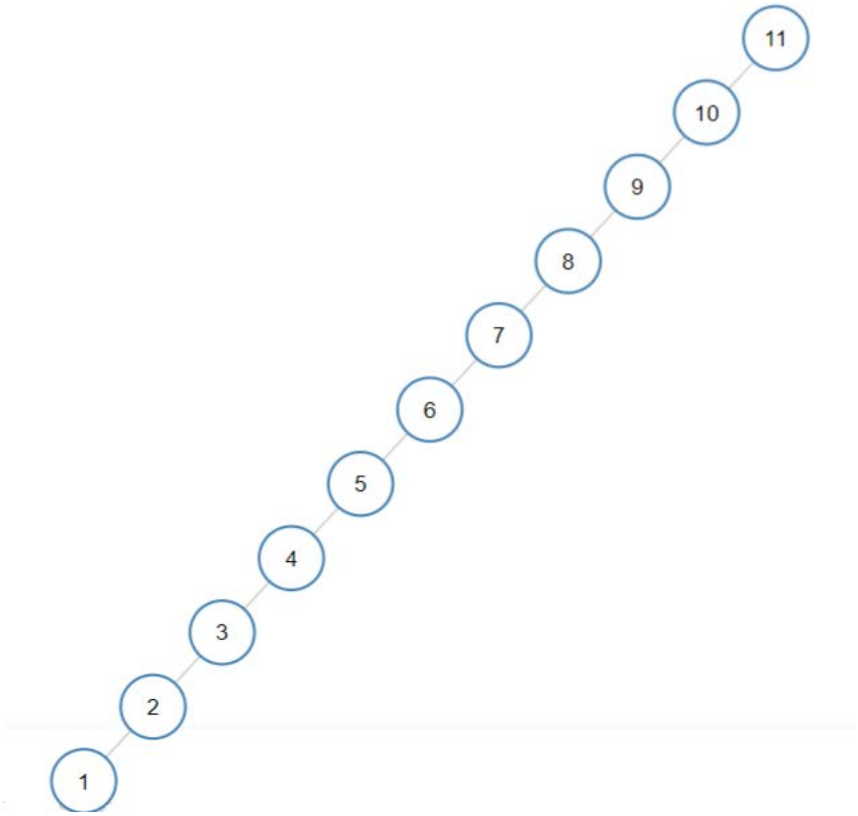
Search –
Time: 6398ns (8 searches)

Deletions –
Time: 7234ns (8 deletes)
Level Order:
[[3], [2, null], [1, null]]



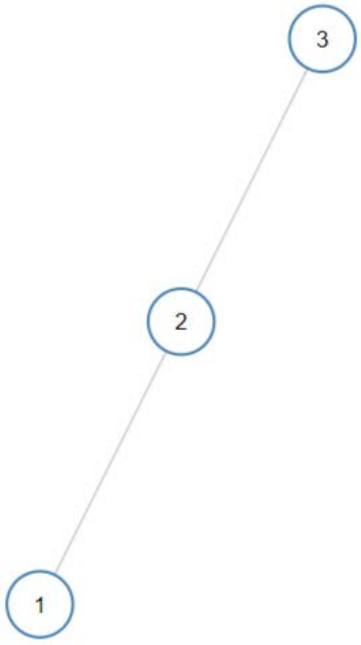
Splay Tree:

Insertions –
Time: 99344ns (11 inserts)
Level Order:
[[11], [10, null], [9, null], [8, null], [7, null], [6, null], [5, null], [4, null], [3, null], [2, null], [1, null]]

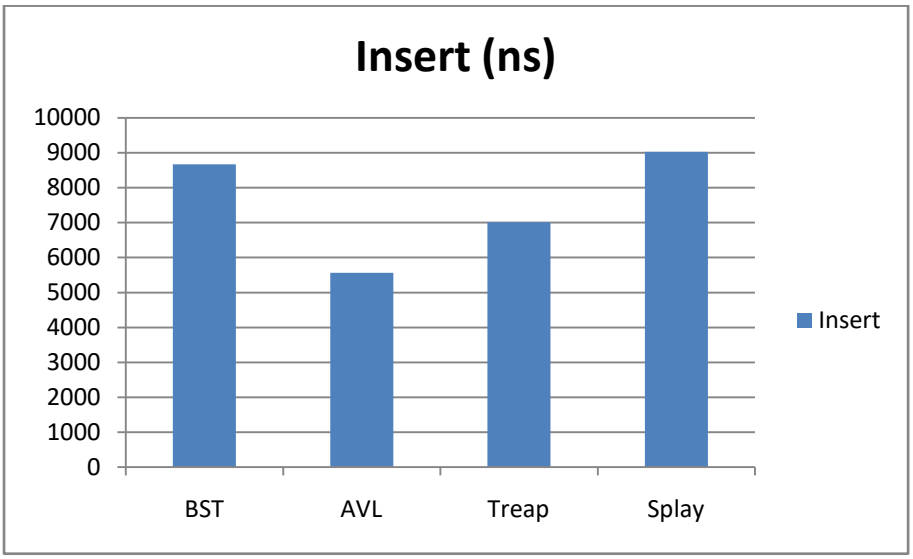


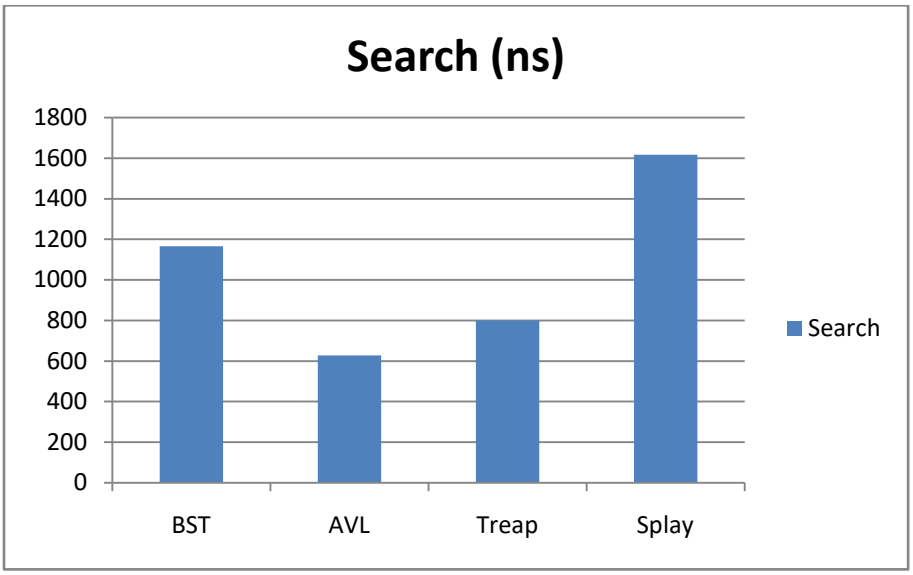
Search –
Time: 12934ns (8 searches)

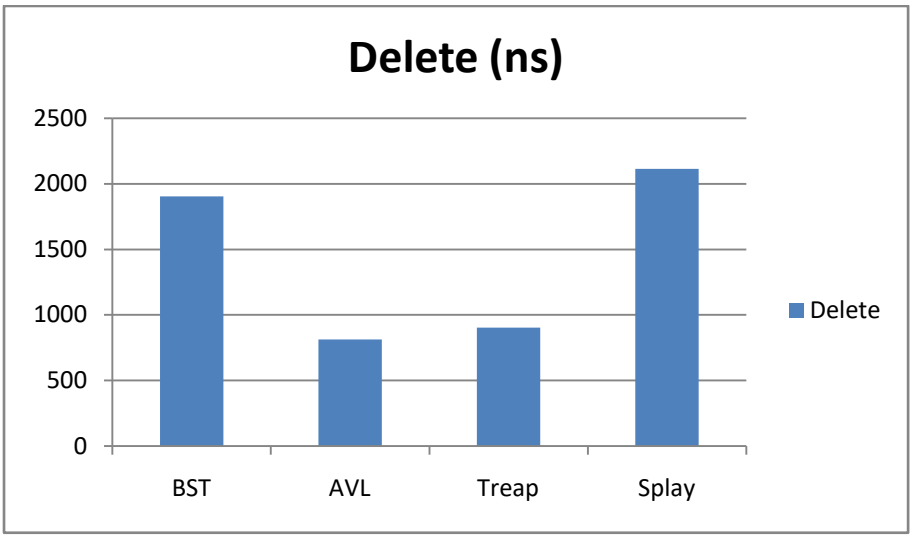
Deletions –
Time: 16922ns (8 deletes)
Level Order:
[[3], [2, null], [1, null]]



Graphs:


Insert (ns)
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
BST
AVL
Treap
Splay
Insert


Search (ns)
1800
1600
1400
1200
1000
800
600
400
200
0
BST
AVL
Treap
Splay
Search


Delete (ns)
2500
2000
1500
1000
500
0
BST
AVL
Treap
Splay
Delete

Observation:

While inserting the values, Splay Tree and Binary Search Tree takes more time than others due to the formation of skewed tree and AVL Tree takes the least time as the tree is always balanced.
For searching a value in a tree, Splay Tree takes more time than others as it searches in a skewed tree and also splays the node accessed to the root of the tree and AVL Tree takes the least time as the tree is always balanced.
While deleting, again Splay Tree and Binary Search Tree takes more time than others due to the formation of skewed tree and AVL Tree takes the least time as the tree is always balanced.

Test Case 3: (Multiple random values example)

Insert: 500 values
Search: 200 values
Delete: 300 values

Binary Search Tree:

Insertions –
Time: 3059943ns (500 inserts)

Search –
Time: 152198ns (200 searches)

Deletions –
Time: 362814ns (300 deletes)

AVL Tree:

Insertions –
Time: 2779482ns (500 inserts)

Search –
Time: 122561ns (200 searches)

Deletions –
Time: 287741ns (300 deletes)

Treap:

Insertions –
Time: 3981842ns (500 inserts)

Search –
Time: 128998ns (200 searches)

Deletions –
Time: 273932ns (300 deletes)

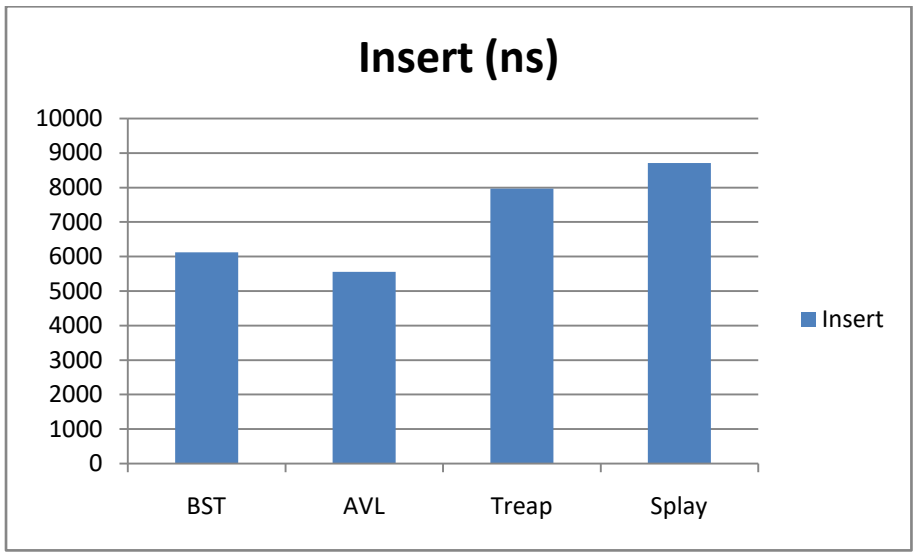
Splay Tree:

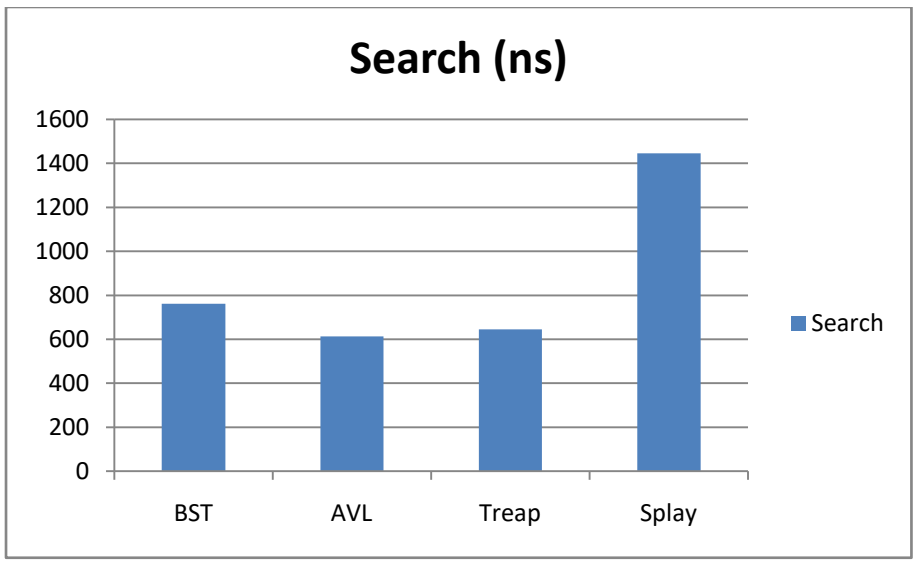
Insertions –
Time: 4355129ns (500 inserts)

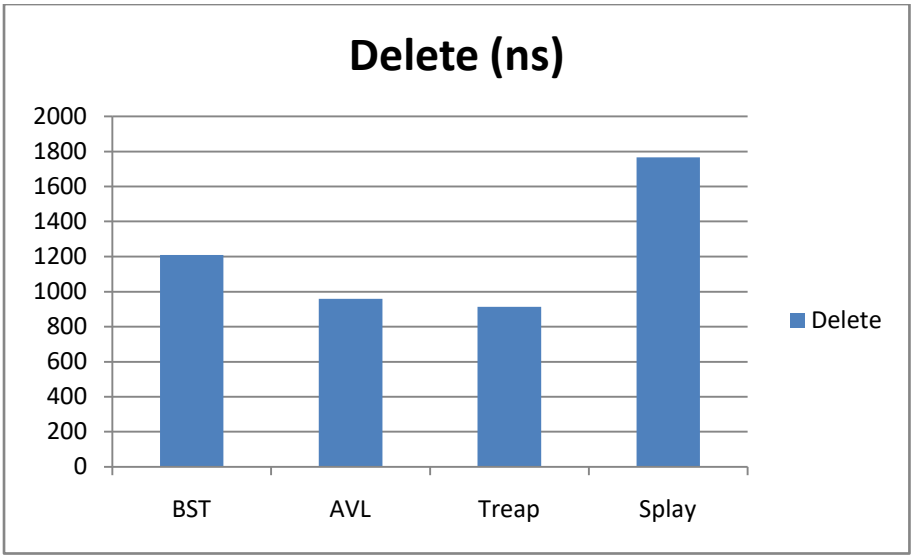
Search –
Time: 288913ns (200 searches)

Deletions –
Time: 530127ns (300 deletes)

Graphs:







Observation:

While inserting the values, Splay Tree takes more time than others as it always splays the node inserted to the root of the tree and AVL Tree takes the least time as the tree is always balanced. For searching a value in a tree, Splay Tree takes more time than others as it always splays the node accessed to the root of the tree and AVL Tree takes the least time as the tree is always balanced. While deleting, again Splay Tree takes more time as it always splays the parent of the node deleted to the root of the tree and AVL Tree takes the least time as the tree is always balanced.

Conclusion:

Binary Search Tree:

Insert: Average time complexity: O(log n) and worst time complexity: O(n)
Search: Average time complexity: O(log n) and worst time complexity: O(n)
Delete: Average time complexity: O(log n) and worst time complexity: O(n)

AVL Tree:

Insert: Average time complexity: O(log n) and worst time complexity: O(log n)
Search: Average time complexity: O(log n) and worst time complexity: O(log n)
Delete: Average time complexity: O(log n) and worst time complexity: O(log n)

Treap:

Insert: Average time complexity: O(log n) and worst time complexity: O(n)
Search: Average time complexity: O(log n) and worst time complexity: O(n)
Delete: Average time complexity: O(log n) and worst time complexity: O(n)

Splay Tree:

Insert: Average time complexity: O(log n) and worst time complexity: O(n)
Search: Average time complexity: O(log n) and worst time complexity: O(n)
Delete: Average time complexity: O(log n) and worst time complexity: O(n)