Empirical Analysis Of Tree Algorithms

A) I have implemented 4 types of tree algorithms based namely:

1) Binary Search Trees (Unoptimized)
2) AVL Trees
3) Splay Trees
4) Treaps

B) Analysis was run on three types of Operations:

1) 1000 operations
2) 10000 operations
3) 100000 operations

Once the insertions are completed, Inorder traversal is executed, just to see if the elements are coming in sorted order or not

C) End to End Testing:

1) Total number of operations is set (Numbers are generated randomly)
2) Insertion for number of operations is executed
3) InOrder Traversal is executed to check if the elements are coming in sorted order
4) Search for number of operations is executed
5) Deletion for number of operations is executed
6) Total time is calculated for all the operations executed

NOTE: Time for every operation is in milliSeconds*.

The cost of every operation for the search, insert and delete seems is same irrespective of the type of tree that is created. Hence we can be certain our initial analysis of O(Log n) amortized time is correct.

Runtime Analysis On Number Of Operations And Charts Showing Operations And Time:

1) Binary Seach Trees:

- **In this algorithm, a Tree Node is defined with left and right references and insertion is done based on the condition, if key is less than root, insert in the lest subtree else insert in the right subtree.**
- **Search is conducted in the same manner where key is checked it is equal to the root otherwise traverse left subtree if key is less than root's key otherwise traverse the right subtree. If the key is not found, the program returns false**

- **Deletion is done if root's key is matched with the key that is to be deleted. If root has once child (either left or right), either of the two is returned else if root has both the children, inorder successor is taken and deletion is called upon that. If the key does not exist, the program states a flag that the key is not present.**
- **Create is just the initialization of Tree Node that (root i.e null) and the reference of BST is created.**

Amortized Time For all operations – O(Log n)

Amortized Time For Create – O(1)

Pseudo Code:

Create:

```
public BinarySearchTree()
	{
		root = null;
	}
```

Insert:

```
public TreeNode InsertRec(TreeNode root, int key)
	{
		if(root==null)
		{
			root = new TreeNode(key);
			return root;
		}

		if(root.key > key)
			root.left = InsertRec(root.left, key);
		else if(root.key <  key)
			root.right = InsertRec(root.right, key);

		return root;
	}
```

Search:

```
public TreeNode SearchRec(TreeNode root, int key)
	{
		if(root==null)
			return null;

		else if(root.key == key)
			return root;

		else if(root.key > key)
			return SearchRec(root.left, key);
		else
```

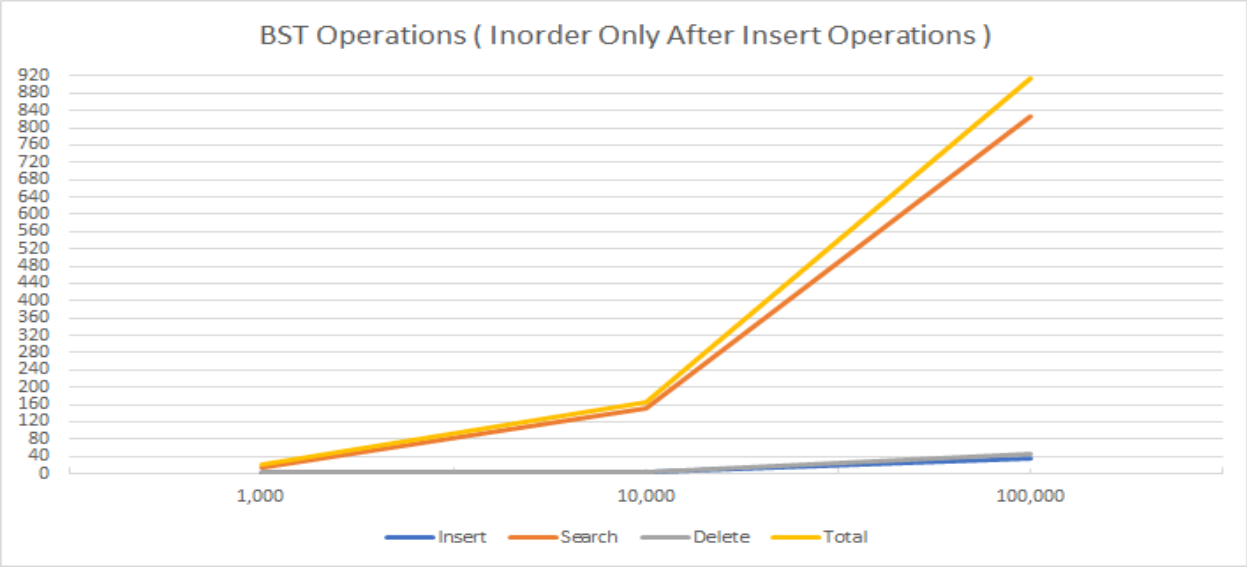
```
                return SearchRec(root.right, key);
```

Delete:

```
public TreeNode DeleteRec(TreeNode root, int key)
      {
            if (root == null)  return root;

            if (key < root.key)
                root.left = DeleteRec(root.left, key);
            else if (key > root.key)
                root.right = DeleteRec(root.right, key);
            else
            {
                if (root.left == null)
                    return root.right;
                else if (root.right == null)
                    return root.left;

                root.key = InOrderSuccessor(root.right);

                root.right = DeleteRec(root.right, root.key);
            }

            return root;
```

OPERATIONS AND GRAPHS

Operations	1000	10000	100000
Insert	1	5	35
Search	15	151	826
Delete	3	5	45
Total	22	164	916



2) **AVL Trees:**

- **In this algorithm, a Tree Node is defined with left and right references and insertion is done based on the condition, if key is less than root, insert in the lest subtree else insert in the right subtree. After that height is checked for all the left and right subtrees and if they differ by m ore than one in the absolute sense, AVL rotations are applied.**
- **Search is conducted in the same manner where key is checked it is equal to the root otherwise traverse left subtree if key is less than root's key otherwise traverse the right subtree. If the key is not found, the program returns false.**
- **Deletion is done if root's key is matched with the key that is to be deleted. If root has once child (either left or right), either of the two is returned else if root has both the children, inorder successor is taken and deletion is called upon that. If the key does not exist, the program states a flag that the key is not present. Once the deletion is successful, height is checked for all the left and right subtrees and if they differ by m ore than one in the absolute sense, AVL rotations are applied.**
- **Create is just the initialization of Tree Node that (root i.e null) and the reference of AVL Tree is created.**

Amortized Time For all operations – O(Log n)

Amortized Time For Create – O(1)

Pseudo Code:

Create:

```
public AVLTree()
	{
		root=null;
	}
```

Insert:

```
public TreeNode InsertRec(TreeNode root, int key)
	{
		if(root==null)
		{
			root = new TreeNode(key);
			return root;
		}

		if(root.key > key)
			root.left = InsertRec(root.left, key);
		else if(root.key <  key)
			root.right = InsertRec(root.right, key);
		else
			return root;
```

```
            root.height = 1 + Math.max(height(root.left), height(root.right));

            int balanceFactor = getBalanceFactor(root);

            // Left Left Case
            if (balanceFactor > 1 && key < root.left.key)
           return rightRotate(root);

        // Right Right Case
        if (balanceFactor < -1 && key > root.right.key)
            return leftRotate(root);

        // Left Right Case
        if (balanceFactor > 1 && key > root.left.key) {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Left Case
        if (balanceFactor < -1 && key < root.right.key) {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
```

Search:

```
public TreeNode SearchRec(TreeNode root, int key)
      {
            if(root==null)
                  return null;

            else if(root.key == key)
                  return root;

            else if(root.key > key)
                  return SearchRec(root.left, key);
            else
                  return SearchRec(root.right, key);
```

Delete:

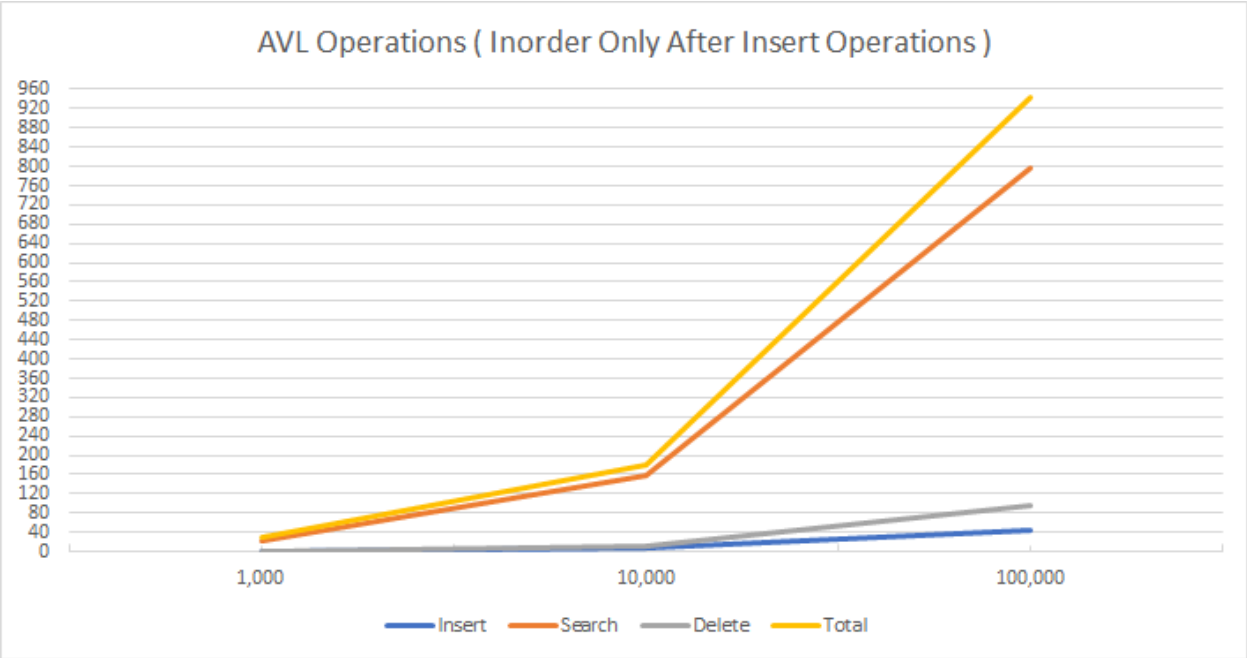
```
public TreeNode DeleteRec(TreeNode root, int key)
      {
             if (root == null)
                 return root;

             if (key < root.key)
                 root.left = DeleteRec(root.left, key);

             else if (key > root.key)
                 root.right = DeleteRec(root.right, key);
```

```
else
{

    if ((root.left == null) || (root.right == null))
    {
        TreeNode temp = null;
        if (temp == root.left)
            temp = root.right;
        else
            temp = root.left;

        if (temp == null)
        {
            temp = root;
            root = null;
        }
        else
            root = temp;
    }
    else
    {

      root.key  = InOrderSuccessor(root.right);

        root.right = DeleteRec(root.right, root.key);
    }
}

if (root == null)
    return root;

root.height = Math.max(height(root.left), height(root.right)) + 1;

int balance = getBalanceFactor(root);

// Left Left Case
if (balance > 1 && getBalanceFactor(root.left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalanceFactor(root.left) < 0)
{
    root.left = leftRotate(root.left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalanceFactor(root.right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalanceFactor(root.right) > 0)
{
    root.right = rightRotate(root.right);
```

```
        return leftRotate(root);
    }

    return root;
```

OPERATIONS AND GRAPHS

Operations	1000	10000	100000
Insert	2	7	45
Search	23	157	795
Delete	2	11	97
Total	31	180	944



3) **Splay Trees:**

- **In this algorithm, a Tree Node is defined with left and right references and insertion is done based on the condition, when you access any node in the tree it becomes the root Node (that is called as splay (Also Known as Search In Splay Trees) functionality where depending upon the factors of left and right subtrees, zigzag is called). Once the splay functionality is over, the same conditions as BST is followed where key is greater than root node we make root as right child of new node and left child of root is copied to new node, else it if smaller vice-versa is done.**

- **Search Is also known as splay functionality, where if key is present it becomes the root otherwise last accessed key is made the root by returning false instead.**
- **Deletion is done in a manner where first the splay functionality is called and if the key is present we make it as the root. After that zigzag is called depending upon the value of the root and both the subtrees.**
- **Create is just the initialization of Tree Node that (root i.e null) and the reference of Splay Tree is created.**

Amortized Time For all operations – O(Log n)

Amortized Time For Create – O(1)

Pseudo Code:

Create:

```
public SplayTree()
	{
		root = null;
	}
```

Insert:

```
public TreeNode InsertRec(TreeNode root, int key)
	{
		if(root == null)
		{
			root = new TreeNode(key);
			return root;
		}

		root = Splay(root, key);

		if(root.key > key)
		{
			TreeNode node = new TreeNode(key);
			node.left = root.left;
			node.right = root;
			root.left = null;
			root = node;
		}
		else if(root.key < key)
		{
			TreeNode node = new TreeNode(key);
			node.right = root.right;
			node.left = root;
			root.right = null;
			root = node;
		}
		else
			root.key = key;
```

```
		return root;
```

Search (Splay):

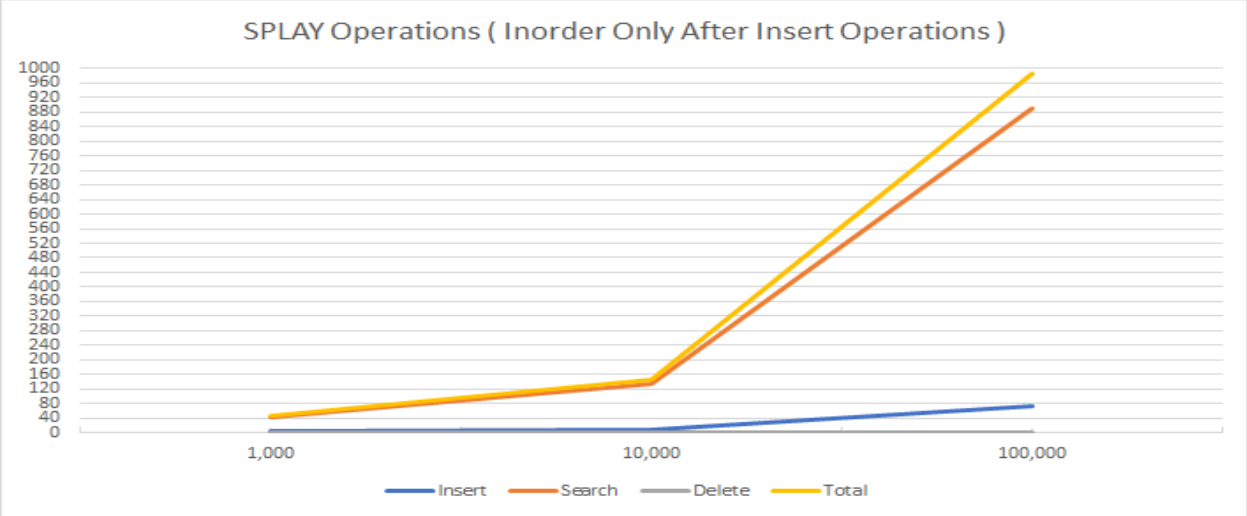
```
public TreeNode Splay(TreeNode root, int key)
	{
		if(root == null)
			return root;

		if(root.key > key)
		{
			if(root.left == null)
				return root;

			if(root.left.key > key)
			{
				root.left.left = Splay(root.left.left, key);
				root = rotateRight(root);
			}
			else if(root.left.key < key)
			{
				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 if(root.key < key)
		{
			if(root.right == null)
				return root;
			if(root.right.key > key)
			{
				root.right.left = Splay(root.right.left, key);

				if(root.right.left != null)
					root.right = rotateRight(root.right);
			}
			else if(root.right.key < key)
			{
				root.right.right = Splay(root.right.right, key);

				root = rotateLeft(root);
			}

			return (root.right == null)? root: rotateLeft(root);
		}
		else
			return root;
```

Delete:

```
public TreeNode DeleteRec(TreeNode root, int key)
	{
		if(root == null)
			return root;

		root = Splay(root, key);

		if(root.key == key)
		{
			if(root.left == null)
				root = root.right;
			else
			{
				TreeNode x = root.right;
				root = root.left;
				Splay(root, key);
				root.right = x;
			}
		}
		else
		{
			System.out.println("key not present in the tree to delete");
		}
		return root;
```

OPERATIONS AND GRAPHS

Operations	1000	10000	100000
Insert	3	8	73
Search	42	134	891
Delete	1	1	1
Total	48	144	985



4) **Treaps:**

- **In this algorithm, a Treap Node is defined with left and right references and a random priority is assigned to every node. Higher the node more the priority, this is different than normal heap priorities. Insertion is done based on the condition, if key is less than root, insert in the lest subtree else insert in the right subtree. After the subtree is decided tree is rotated left or right depending upon the priority as the higher priority will go up towards the root and vice-versa.**
- **Search is conducted in the same manner where key is checked it is equal to the root otherwise traverse left subtree if key is less than root's key otherwise traverse the right subtree. If the key is not found, the program returns false.**
- **Deletion is done if root's key is matched with the key that is to be deleted. If root has once child (either left or right), either of the two is returned else if root has both the children, the priorities come into picture. If root's priority is more than left child's priority, treap is rotated left and the delete is again called on the left subtree else vice-versa on the right subtree/**
- **Create is just the initialization of Treap Node that (root i.e null) and the reference of Treap Tree is created.**

Amortized Time For all operations – O(Log n)

Amortized Time For Create – O(1)

Pseudo Code:

Create:

```
public Treap()
	{
		root = null;
	}
```

Insert:

```
public TreapNode InsertRec(TreapNode root, int key)
	{
		if(root==null)
		{
			root = new TreapNode(key);
			return root;
		}

		if(root.key > key)
		{
			root.left = InsertRec(root.left,key);
			if(root.left!=null && root.left.priority > root.priority)
```

```
                {
                        root = rotateRight(root);
                }
        }
        else
        {
                root.right = InsertRec(root.right,key);
                if(root.right!=null && root.right.priority > root.priority)
                {
                        root = rotateLeft(root);
                }
        }

        return root;
```

Search:

```
public TreapNode SearchRec(TreapNode root, int key)
    {
        if(root==null)
            return null;

        else if(root.key == key)
            return root;

        else if(root.key > key)
            return SearchRec(root.left, key);
        else
            return SearchRec(root.right, key);

    }
```

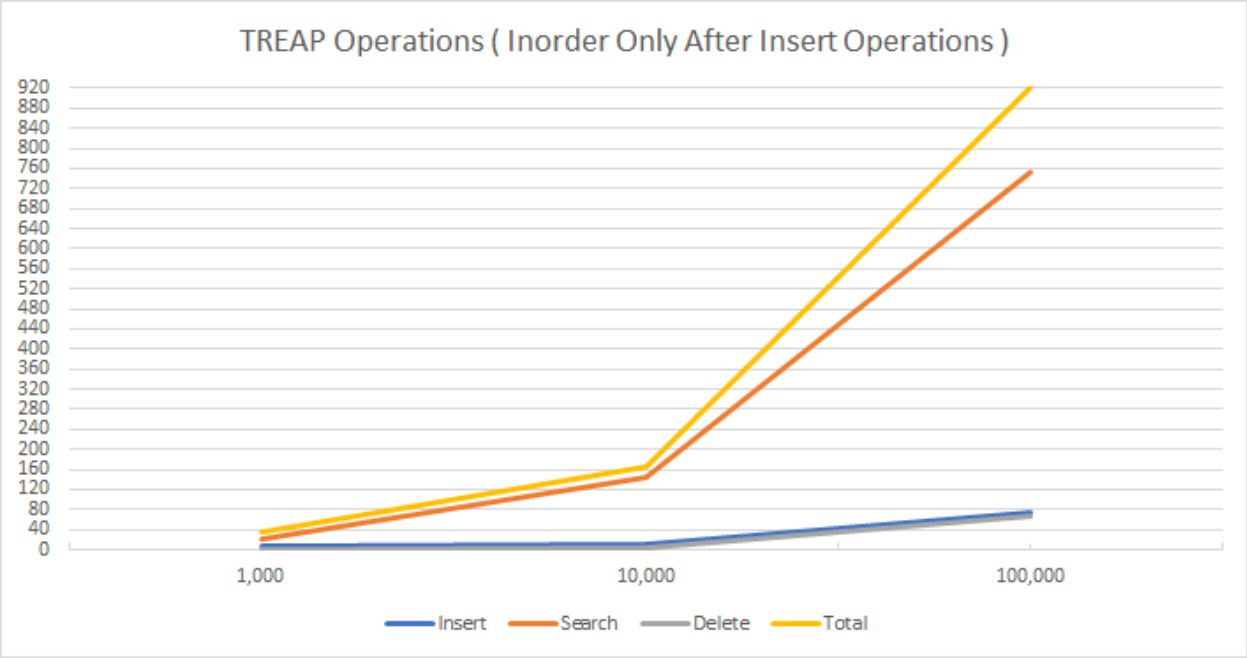
Delete:

```
public TreapNode DeleteRec(TreapNode root, int key)
    {
         if (root == null)  return root;

         if (key < root.key)
             root.left = DeleteRec(root.left, key);
         else if (key > root.key)
             root.right = DeleteRec(root.right, key);
         else
         {
             if(root.left == null && root.right == null)
                 root = null;
             else if(root.left != null && root.right != null)
             {
                 if(root.left.priority < root.right.priority)
                 {
                     root = rotateLeft(root);
                     root.left = DeleteRec(root.left, key);
```

```
                }
                else
                {
                        root = rotateRight(root);
                        root.right = DeleteRec(root.right, key);
                }
        }
        else
        {
                root = (root.left!=null)? root.left : root.right;
        }
}

return root;
```

OPERATIONS AND GRAPHS

Operations	1000	10000	100000
Insert	6	11	74
Search	23	144	753
Delete	2	4	67
Total	35	164	920



Note: If we see above in the operations table of every tree, as we are randomly distributing the data the cost for search operation is always more as the worst case search can take O(n) and similar for

other two operations that is insert and delete. But if the data is equally distributed as in case of height balanced trees or the trees implemented above, the amortized time comes to be O(Log n)