

# Project-2

## Tree Algorithms

By Sameer Ganesh Shinde, Student ID: 19644693

1. Test Suite .....	2
- Test Case Description.....	2
- How to run project.....	3
- Directory Structure and Makefile.....	5
2. Binary Search Tree .....	6
- Pseudo Code .....	6
- Algorithm Analysis .....	7
- Space Analysis.....	7
3. AVL Tree .....	8
- Pseudo Code .....	8
- Algorithm Analysis .....	10
- Space Analysis.....	10
4. Splay Tree .....	11
- Splay Tree Operations.....	11
- Pseudo Code .....	11
- Algorithm Analysis .....	13
Amortized Analysis .....	13
- Space Analysis.....	15
5. Red-Black Tree .....	16
- Pseudo Code .....	16
- Algorithm Analysis .....	18
- Space Analysis.....	18
6. Comparative Analysis .....	19
- When Data is Uniform.....	19
- When Data is Skewed .....	22
- Ideal Tree for different use cases .....	26

---

## 1. Test Suite

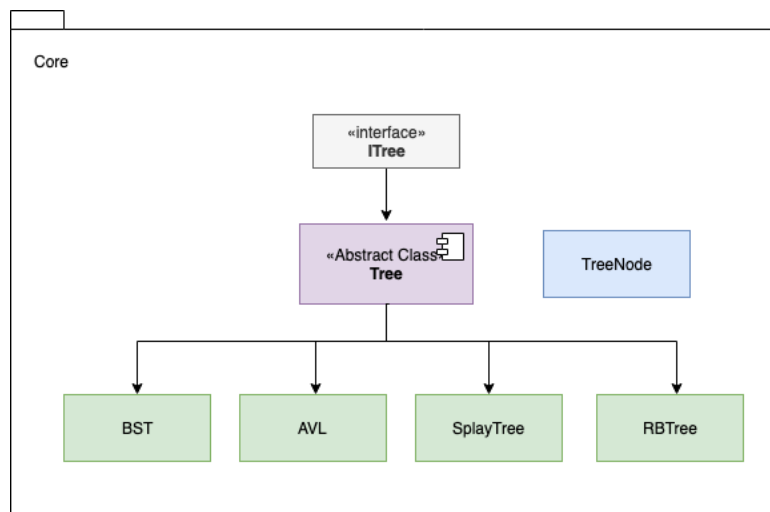
I have prepared an extensive test suite for number of test cases applicable for the tree algorithms developed. These cases cover most of the use cases in regular usage of tree.

Test Case	Description	Applicable to
1	Tree Creation and simply add operation	All
2	Test for existing and non-existing key	All
3	Simple remove operation	All
4	adding high number (100) of keys	All
5	adding very high number (10000) of keys	All
6	Adding 10000 and removing half of that	All
7	Traversing 10000 keys	All
8	Adding 10000 skewed entries	All
9	Adding 10000 skewed entries and removing half of that in same order	All

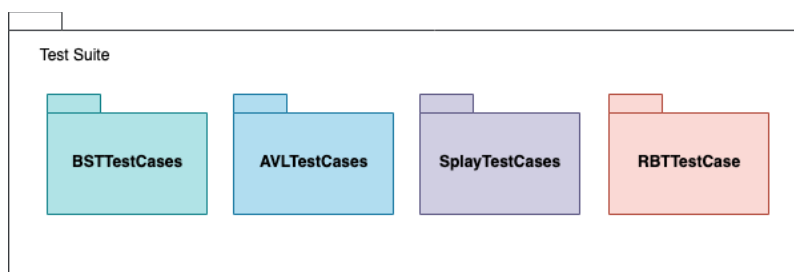
**Note: For all test cases, run time is measured in Nano Seconds.**

## How to run project

### Project File structure:



### Test Files Structure:



### Utilities.java

- This file is responsible for manipulating operations for tests on BinarySearchTree, AVLTree, SplayTree and Red-Black Tree.
- In addition, for testing purposes I have written check method for validating the AVL, BST and Red-Black Trees. At the end of each test case we will compare our algorithm's shape(balance) with the help of check method for verifying the completeness of algorithm.

## Methods

## Description

<code>createTree(String name, Logger log)</code>	This method returns the instance of the tree of specified type in name.
<code>populateTree(Tree node, String type, long seed, int count)</code>	This method performs add operation for count number of times. For randomly generating entries we are passing seed.
<code>deleteFromTree(Tree node, String type, long seed, int toBeDeleted)</code>	This method performs remove operation for toBeDeleted number of times. For maintaining the same order in all cases, we are passing seed for randomly generating entries to be deleted.
<code>traverseTree(Tree node)</code>	This method performs in-order traversal on given tree with root.

## Steps for Running Project

1. Unzip the given folder.
2. If you want to open project in eclipse, you can just import eclipse-project folder
3. Run

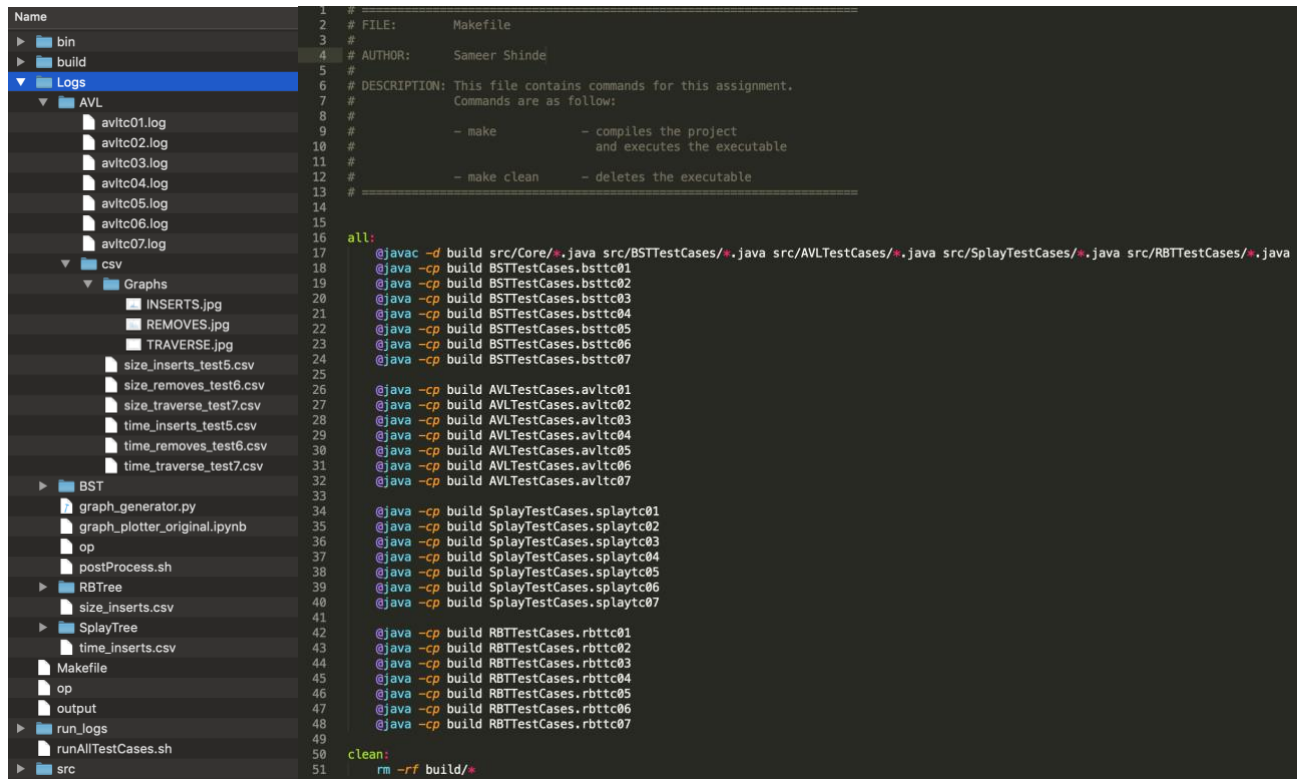
You can simply run the script [runAllTestCases.sh](#) for running the entire test suite.

- This will compile all the programs by make.
- Each run for testcase will generate the logs for that run in Logs directory.
- This script will call another script "postprocess.sh" for post processing the logs after run is completed. This will generate intermediate csv file for feeding input to the graph generator.
- At the end, it will call a python script graph\_generator.py for generating graphs for the output.
- You can find all the files according to the following structure.
- If you wish to run individual test cases, please refer makefile for individual command.

## Directory Structure and Makefile.

Directory

Makefile



The screenshot shows a code editor with a directory tree on the left and a Makefile on the right. The directory tree includes folders like bin, build, Logs, AVL, csv, BST, SplayTree, and src. The Makefile contains comments and commands for building the project, including javac and java commands for various test cases.

```
1 # Makefile
2 # FILE: Makefile
3 #
4 # AUTHOR: Sameer Shinde
5 #
6 # DESCRIPTION: This file contains commands for this assignment.
7 # Commands are as follow:
8 #
9 # - make - compiles the project
10 # and executes the executable
11 #
12 # - make clean - deletes the executable
13 #
14 #
15
16 all:
17 @javac -d build src/Core/*.java src/BSTTestCases/*.java src/AVLTestCases/*.java src/SplayTestCases/*.java src/RBTTestCases/*.java
18 @java -cp build BSTTestCases.bsttc01
19 @java -cp build BSTTestCases.bsttc02
20 @java -cp build BSTTestCases.bsttc03
21 @java -cp build BSTTestCases.bsttc04
22 @java -cp build BSTTestCases.bsttc05
23 @java -cp build BSTTestCases.bsttc06
24 @java -cp build BSTTestCases.bsttc07
25
26 @java -cp build AVLTestCases.avltc01
27 @java -cp build AVLTestCases.avltc02
28 @java -cp build AVLTestCases.avltc03
29 @java -cp build AVLTestCases.avltc04
30 @java -cp build AVLTestCases.avltc05
31 @java -cp build AVLTestCases.avltc06
32 @java -cp build AVLTestCases.avltc07
33
34 @java -cp build SplayTestCases.splaytc01
35 @java -cp build SplayTestCases.splaytc02
36 @java -cp build SplayTestCases.splaytc03
37 @java -cp build SplayTestCases.splaytc04
38 @java -cp build SplayTestCases.splaytc05
39 @java -cp build SplayTestCases.splaytc06
40 @java -cp build SplayTestCases.splaytc07
41
42 @java -cp build RBTTestCases.rbttc01
43 @java -cp build RBTTestCases.rbttc02
44 @java -cp build RBTTestCases.rbttc03
45 @java -cp build RBTTestCases.rbttc04
46 @java -cp build RBTTestCases.rbttc05
47 @java -cp build RBTTestCases.rbttc06
48 @java -cp build RBTTestCases.rbttc07
49
50 clean:
51 rm -rf build/*
```

## Pseudo Code for Create Tree ( Common for all)

```
public static Tree createTree(String name, Logger log) {
    if(name.equals("BST")) {
        return new BST(log);
    }
    else if(name.equals("AVL")) {
        return new AVL(log);
    }
    else if(name.equals("SPLAY")) {
        return new SplayBST(log);
    }
    else if(name.equals("RBTTree")) {
        return new RedBlackTree(log);
    }
    return null;
}
```

---

## 2. Binary Search Tree

This implementation of BST is implemented using templates, therefore supports any type of values to be passed.

### Pseudo Code:

```
def add(root, val):
    if (root == null) {
        return new TreeNode<T>(val);
    }
    int cmp = val.compareTo(root.value);
    if (cmp < 0)
        root.left = add(root.left, val);
    else if (cmp > 0)
        root.right = add(root.right, val);
    else
        root.value = val;
    root.size = 1 + size(root.left) + size(root.right);
    return root;

def remove(root, key):
    if (x == null) return null;
    int cmp = key.compareTo(x.value);
    if (cmp < 0) x.left = remove(x.left, key);
    else if (cmp > 0) x.right = remove(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        TreeNode<T> t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = (t.left);
    }
    x.size = size(x.left) + size(x.right) + 1;
    return x;

def search(k):
    low = 0
    high = n-1
    while low <= high
        mid = (low + high)/2
        // compare k with A[mid]:
        if k < A[mid]: high = mid - 1
        else if k > A[mid]: low = mid + 1
```

```

        else: return mid
    k is not in data

```

## Algorithm Analysis:

### Binary Search Tree - Tree Insertions in Random Order

- Expected Cost Assume keys are  $0, 1, \dots, n - 1$ .
- When we insert key  $x$  . . . The expected cost of the insertion is  $E$  [the number of nodes encountered on the search path for  $x$ ]
- To compute the expected number of nodes encountered on the search path for  $x$ :
  - For any other key  $y$ , let  $C(y) = 1$  if we encounter  $y$  when searching for  $x$ ,  $C(y) = 0$  otherwise.
  - $C(y) = 1$  iff the first key in the range  $[x, y]$  or  $[y, x]$  is  $y$ .
  - The probability of this happening is  $\frac{1}{|y-x|+1}$ .

$E[\text{number of nodes on the path to insert } x]$

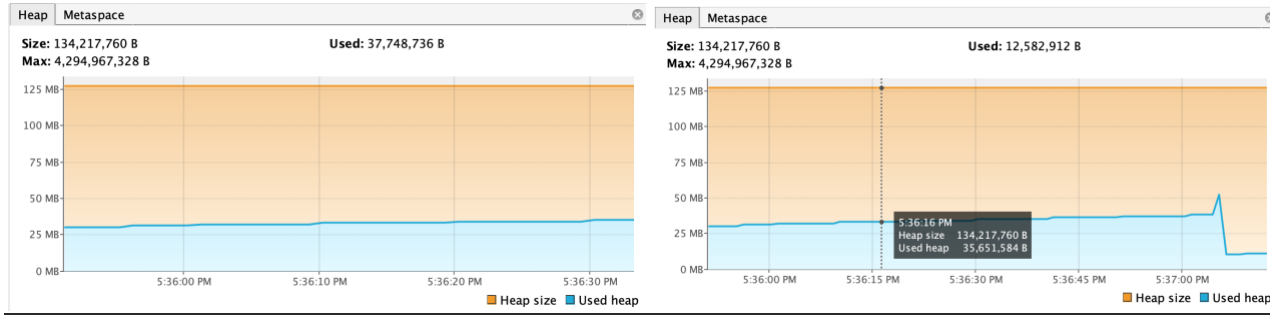
$$\begin{aligned}
 &= E \sum_y C(y) \\
 &= \sum_y E[C(y)] \\
 &= \sum_y \Pr[C(y) = 1] \\
 &= \sum_y \frac{1}{|y-x|+1} \\
 &\leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \frac{1}{3} + \frac{1}{2} + \dots + \frac{1}{n} + \frac{1}{n} \\
 &< 2 \ln n
 \end{aligned}$$

So average insertion cost is  $O(\log n)$  if insertions are in random order and there are no deletions.

## Space Complexity:

Used Heap Size after adding 10000 elements

Used Heap Size after deleting 5k keys



### 3. AVL Trees

In an AVL Tree, at each node, the height of the left subtree and the height of the right subtree differ by at most one. Each node stores either

- Its height;
- Whether its height differs from the height of its parent by 1 or 2 (A single bit of information)

Number of nodes in AVL tree  $\geq \text{Fib}(h) - 1$ .

This implementation of AVL is implemented using templates, therefore supports any type of values to be passed.

#### Pseudo Code:

```
def add(root, val):
    if (x == null) return new TreeNode<T>(val, 0, 1);
    int cmp = val.compareTo(x.value);
    if (cmp < 0) {
        x.left = put(x.left, val);
    }
    else if (cmp > 0) {
        x.right = put(x.right, val);
    }
    else {
        x.value = val;
        return x;
    }
    x.size = 1 + size(x.left) + size(x.right);
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);

def balance(x):
    if (balanceFactor(x) < -1) {
        if (balanceFactor(x.right) > 0) {
            x.right = rotateRight(x.right);
        }
        x = rotateLeft(x);
    }
```



```

    else if (balanceFactor(x) > 1) {
        if (balanceFactor(x.left) < 0) {
            x.left = rotateLeft(x.left);
        }
        x = rotateRight(x);
    }
    return x;

```

```

def balanceFactor(x):
    return height(x.left) - height(x.right);

```

```

def remove(root, key):
    int cmp = key.compareTo(x.value);
    if (cmp < 0) {
        x.left = remove(x.left, key);
    }
    else if (cmp > 0) {
        x.right = remove(x.right, key);
    }
    else {
        if (x.left == null) {
            return x.right;
        }
        else if (x.right == null) {
            return x.left;
        }
        else {
            TreeNode<T>y = x;
            x = min(y.right);
            x.right = deleteMin(y.right);
            x.left = y.left;
        }
    }
    x.size = 1 + size(x.left) + size(x.right);
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);

```

```

def deleteMin(x)
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.size = 1 + size(x.left) + size(x.right);
    x.height = 1 + Math.max(height(x.left), height(x.right));
    return balance(x);

```

```

def search(root, k):
    if (x == null) return null;
    int cmp = key.compareTo(x.value);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);

```

```
else return x;
```

## Algorithm Analysis:

To insert into an AVL Tree:

- Do standard insertion.
- Node gets inserted at leaf.
- Follow a path from the leaf back up to the root, updating heights
- If a node has become unbalanced, do a rotation to rebalance it.
- Case analysis shows: never need more than two rotations.

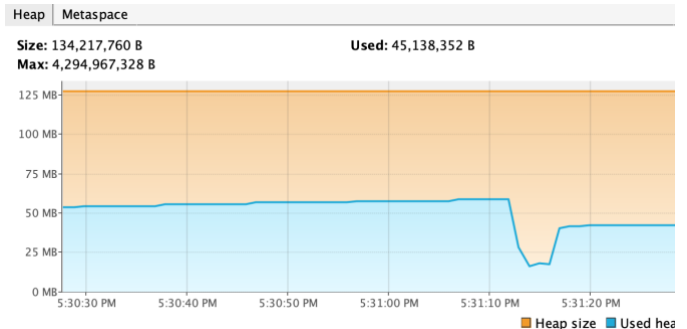
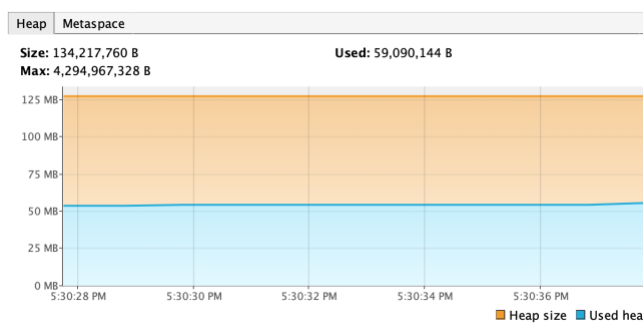
Deletion

- follows the same basic principle but is messier.
- May need to do as many as  $O(\log n)$  rotations.

## Space Complexity:

Used Heap Size after adding 10000 elements

Used Heap Size after deleting 5k keys



---

## 4. Splay Trees

- No explicit balance condition
- Amortized (not worst-case) time bound on operations
- Basic operation is splay. Splaying at node  $x$  means making  $x$  the root through a specific series of rotations.
- Note: we cannot use an arbitrary sequence of rotations that brings  $x$  to the root. The analysis depends on using the specific sequence described here.
- 3 cases:
  1.  $x$  has no grandparent (zig)
  2.  $x$  is LL or RR grandchild (zig-zig)
  3.  $x$  is LR or RL grandchild (zig-zag)
- Illustrate cases where  $x$  is a left child of its parent (other cases are mirror images)

### Splay tree operations:

like binary search tree, except . . .

- After access( $i$ ):
  - Successful: splay at node containing  $i$
  - Unsuccessful: splay at last node on search path
- After insert( $i$ ): splay at node containing  $i$
- After delete( $i$ ):
  - Found: Let  $x$  be the node that was actually deleted; splay at the node that was the parent of  $x$
  - Not found: Splay at last node on search path
- As we will show next, each of these operations can be performed in  $O(\log n)$  amortized time.
- NOTE: Nice property of splay trees: can make any given node the root (at cost of  $O(\log n)$  amortized time).

This implementation of Splay Tree is implemented using templates, therefore supports any type of values to be passed.

### Pseudo Code:

```
def add(root, val):
```

```
    TreeNode<T> n;
    int c;
    if (root == null) {
        root = new TreeNode<T>(key);
        return;
    }
    splay(key);
    if ((c = key.compareTo(root.value)) == 0) {
        return;
    }
    n = new TreeNode<T>(key);
    if (c < 0) {
        n.left = root.left;
        n.right = root;
        root.left = null;
    } else {
        n.right = root.right;
        n.left = root;
        root.right = null;
    }
    root = n;
```

```
def splay(key):
```

```
    TreeNode<T> l, r, t, y;
    l = r = header;
    t = root;
    header.left = header.right = null;
    for (;;) {
        if (key.compareTo(t.value) < 0) {
            if (t.left == null) break;
            if (key.compareTo(t.left.value) < 0) {
                rotateRight(t.left)
            }
            r.left = t;                                /* link right */
            r = t;
            t = t.left;
        } else if (key.compareTo(t.value) > 0) {
            if (t.right == null) break;
            if (key.compareTo(t.right.value) > 0) {
                rotateRight(t.right)
            }
            l.right = t;                                /* link left */
            l = t;
            t = t.right;
        }
    }
```

```

        } else {
            break;
        }
    }
    l.right = t.left;
    r.left = t.right;
    t.left = header.right;
    t.right = header.left;
    root = t;
}
/* assemble */

```

```

def remove(key):
    TreeNode<T> x;
    splay(key);
    if (key.compareTo(root.value) != 0) {
        return;
    }
    // Now delete the root
    if (root.left == null) {
        root = root.right;
    } else {
        x = root.right;
        root = root.left;
        splay(key);
        root.right = x;
    }
}

def search(root, key):
    if (root == null) return null;
    splay(key);
    if (root.value.compareTo(key) != 0) return null;
    return root;

```

### Algorithm 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  $w$ ).

Define the potential function by

$$\Phi = \sum_{\text{all nodes } x} \lg |x|.$$

Equivalently,

$$\Phi = \lg \left( \prod_{\text{all nodes } x} |x| \right).$$

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

1. The change in potential prior to the splay (due to the basic binary tree operation)
2. The change in potential during the splay

We will analyze these two terms separately.

### Amortized analysis:

#### $\Delta\Phi$ prior to the splay.

How does the potential change when we do a search, delete, or insert, prior to the splay?

- Search: no change
- Delete: decreases
- Insert: Let  $z_1, z_2, \dots, z_k$  be the ancestors of the newly inserted node. Let  $s_i$  be the value of  $|z_i|$  before the insertion, and  $n$  be the number of keys before the insertion.
- Note that  $1 < s_1 < s_2 < \dots < s_k = n + 1$ .
- The change in the potential function is

$$\Delta\Phi = \lg 2 + \lg \prod_{i=1}^k \frac{s_i + 1}{s_i} \leq \lg \prod_{i=1}^{n+1} \frac{i + 1}{i} = \lg(n + 2) = O(\log n)$$

#### $\Delta\Phi$ during the splay.

**Lemma:** Let  $\Delta\Phi$  be the change in the potential function due to one of the rebalance operations shown for Case 1, Case 2, or Case 3, and let  $r$  be the factor by which  $|x|$  increases during that operation.

Then:

- In Case 1,  
 $\Delta\Phi - \lg r^3 \leq 0$
- In Cases 2 and 3,  
 $\Delta\Phi - \lg r^3 \leq -2$

**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

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

**Theorem:** 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.

**Proof:**

- Actual time:  $O(d)$  ( $d$  = distance from splayed node to root node, immediately before the splay).
- Change in potential:
  - Before the splay:
    - Search: no change
    - Insert:  $+O(\log n)$
    - Delete: decrease

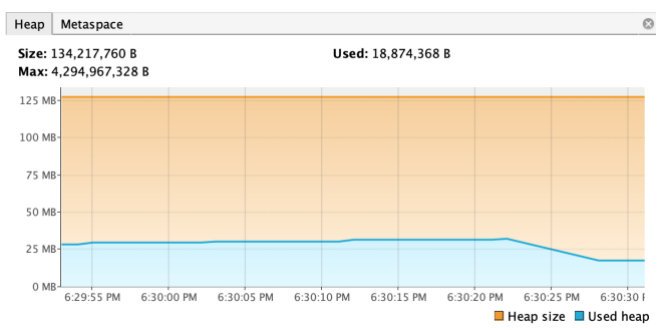
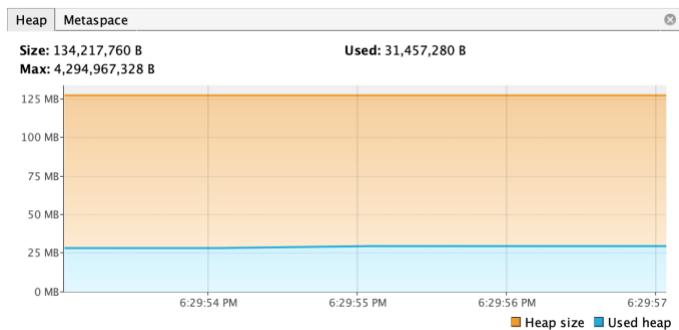
- During the splay:  $\Delta\Phi \leq -d + 1 + 3 \lg(n + 1)$

Hence amortized time is  $O(\log n)$ .

## Space Complexity:

Heap Size after adding 10000 elements

Heap Size after deleting 5000 keys

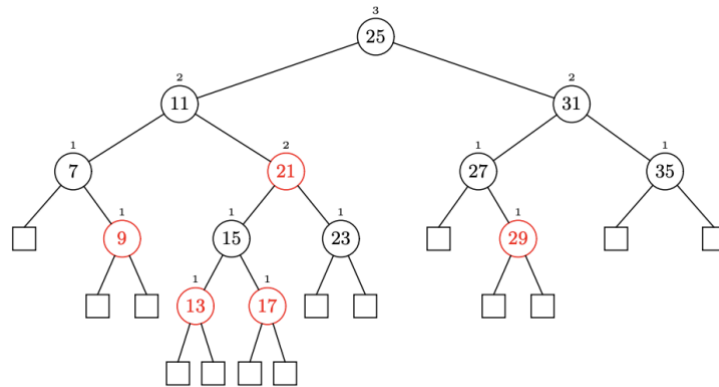


---

## 5. Red Black Trees

Red-black tree: extended binary tree:

- Every node is colored either red or black
- All external nodes are black
- The children of a red node must be black
- The path from every external node to the root contains the same number of black nodes
- The black height of a node x is the number of black nodes on a path from x to a leaf, not counting x



This implementation of Red-Black Tree is implemented using templates, therefore supports any type of values to be passed.

### Pseudo Code:

```
def add(h, val):
    if (h == null) return new TreeNode<T>(key, RED, 1);
    int cmp = key.compareTo(h.value);
    if (cmp < 0) h.left = add(h.left, key);
    else if (cmp > 0) h.right = add(h.right, key);
    else
        h.value = key;

    // fix-up any right-leaning links
    if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
```



```

        if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left) && isRed(h.right)) flipColors(h);
        h.size = size(h.left) + size(h.right) + 1;

    return h;

```

```

def remove(key):
    if (key == null)
        throw new IllegalArgumentException("argument to delete() is null");
    if (!contains(key)) return;
    // if both children of root are black, set root to red
    if (!isRed(root.left) && !isRed(root.right))
        root.color = RED;
    root = delete(root, key);
    if (!isEmpty()) root.color = BLACK;

```

```

def delete(h, key):
    if (key.compareTo(h.value) < 0) {
        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    }
    else {
        if (isRed(h.left))
            h = rotateRight(h);
        if (key.compareTo(h.value) == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (key.compareTo(h.value) == 0) {
            TreeNode<T> x = min(h.right);
            h.value = x.value;
            h.right = deleteMin(h.right);
        }
        else h.right = delete(h.right, key);
    }
    return balance(h);

```

```

def balance(h):
    if (isRed(h.right)) h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right)) flipColors(h);

    h.size = size(h.left) + size(h.right) + 1;
    return h;

```

```

def deleteMin(h)
    if (h.left == null)
        return null;

```

```

if (!isRed(h.left) && !isRed(h.left.left))
    h = moveRedLeft(h);

h.left = deleteMin(h.left);
return balance(h);

```

```

def search(x, key):
    while (x != null) {
        int cmp = key.compareTo(x.value);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else return x.value;
    }
    return null;

```

## Algorithm Analysis:

### To insert into a Red-Black Tree:

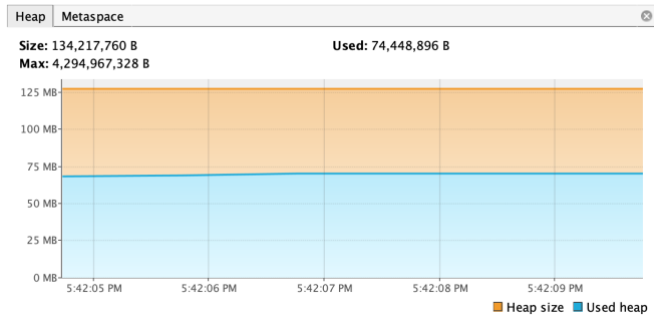
- Do standard insertion.
- New node replaces an external node
  - New node is colored red with two (black) external children
  - All properties are satisfied except that parent of new (red) node may also be red.
- Red-red anomaly can be fixed with
  - Recoloring of at most  $O(\log n)$  nodes
  - At most 2 rotations.

### Deletion

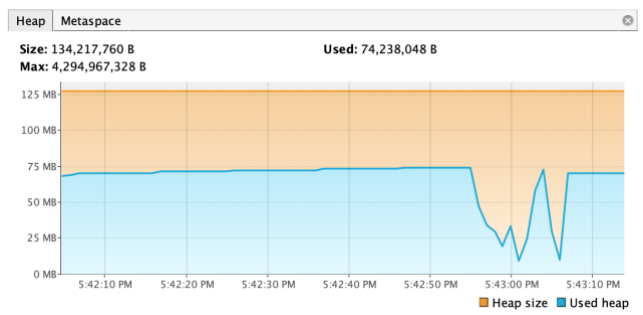
- Follows the same basic principle, but again is messier.
- Requires at most  $O(1)$  rotations (better than AVL trees)

## Space Complexity:

Used Heap Size after adding 10000 elements



Used Heap Size after deleting 5k keys

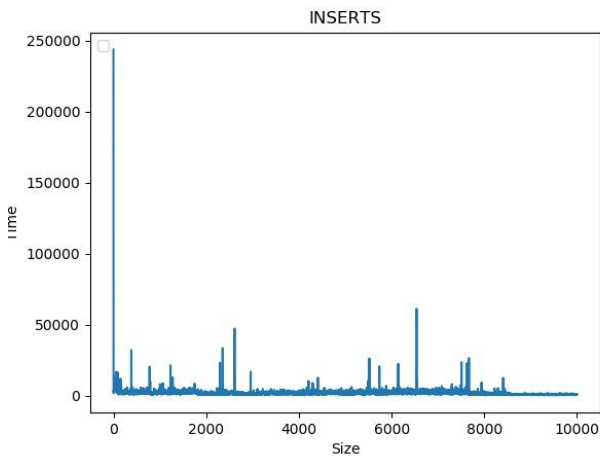


---

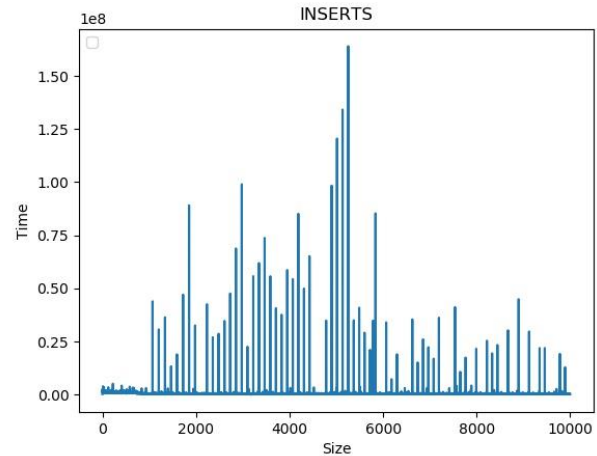
## 6. Comparative Analysis

### When data is Uniform

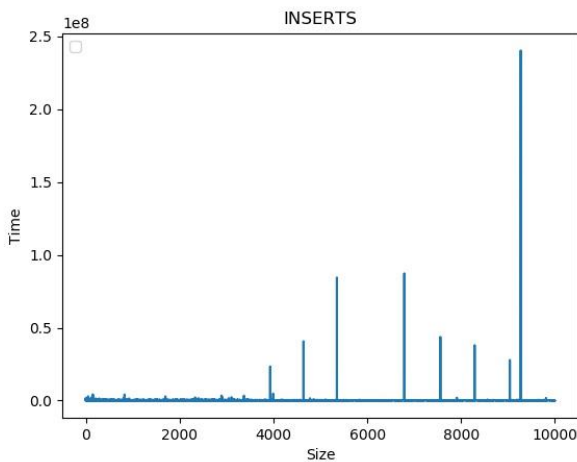
#### Add operation:



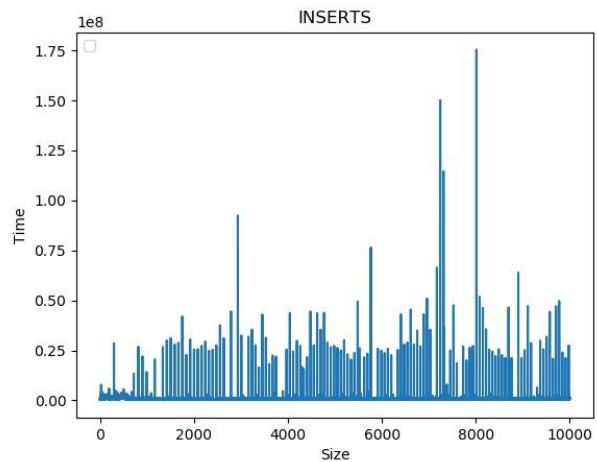
**BST – Total Time:** 20101885



**AVL- Total Time:** 7943240868



**Red-Black Tree- Total Time:** 1517909130



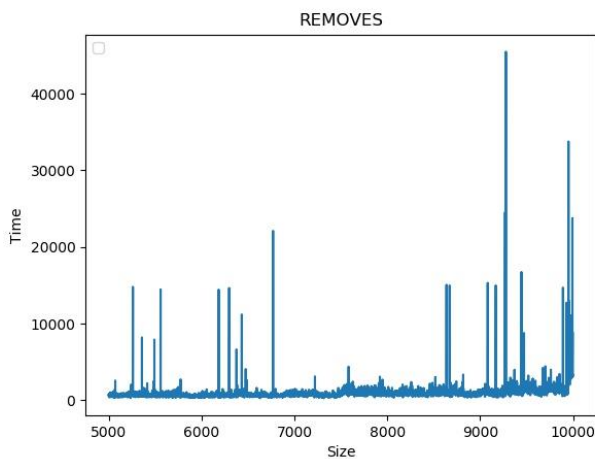
**SPLAY- Total time:** 11279265699

#### Observations:

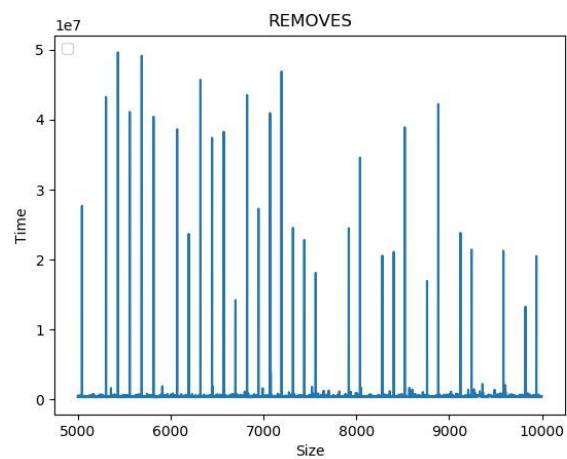
- During my analysis I found that BST has faster insertion performance.
- But BST is not always balanced.

- Therefore, the next best Insertion performance we got was from Red Black Trees since there is relatively a smaller number of rotations required.
  - On the other hand, AVL Tree takes little extra number of rotations, therefore it takes more time than RBTREE
  - In splay tree, every addition requires splay operation to be performed and therefore it has the worst performance for insertion
- 

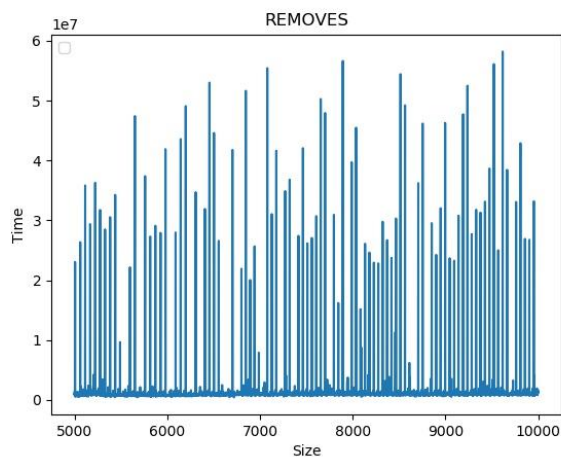
## Remove operation:



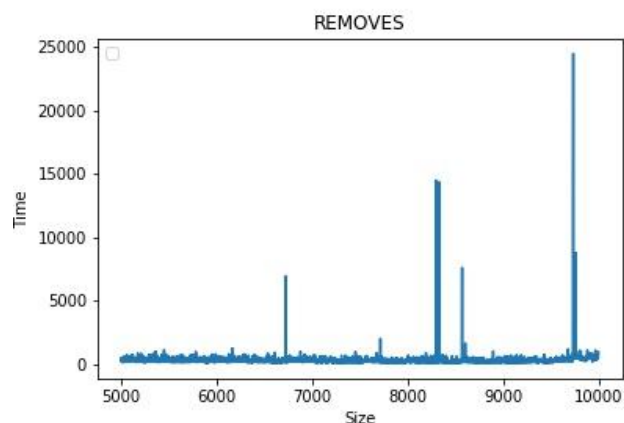
**BST – Total Time:** 5076188



**AVL- Total Time:** 3050462883



**Red-Black Tree- Total Time:** 7727197516



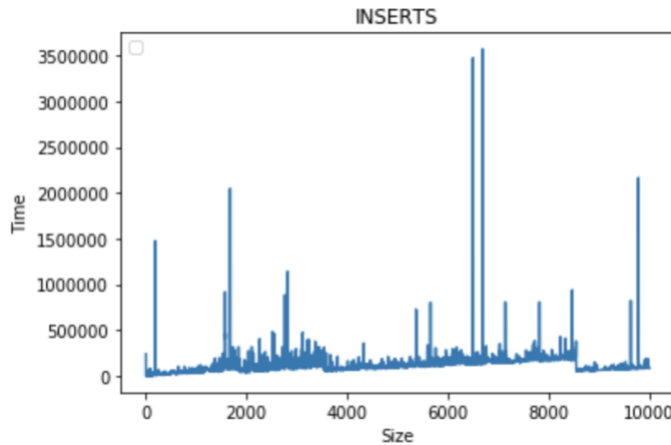
**SPLAY- Total time:** 1611943

### Observations:

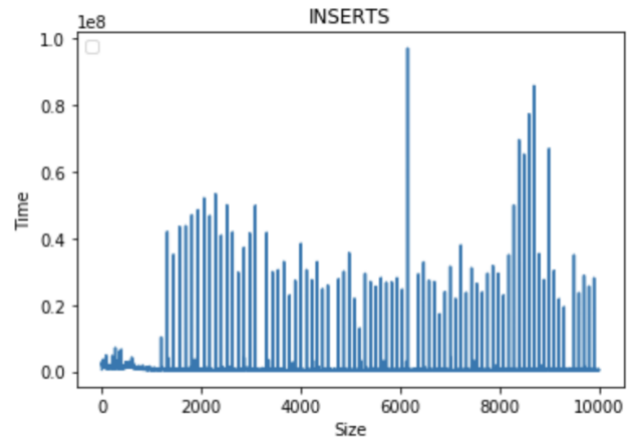
- During my analysis I found that Splay Tree has best performance for removal, thanks to its self-balancing optimization.
- BST comes second but as mentioned, not always balanced.
- Therefore, the next best removal performance we got was from AVL Trees. We have got this performance because in every removal, we are calling contains to check if element exist or not. AVL provides faster look ups and therefore in this case, it gave better performance.

## When data is skewed

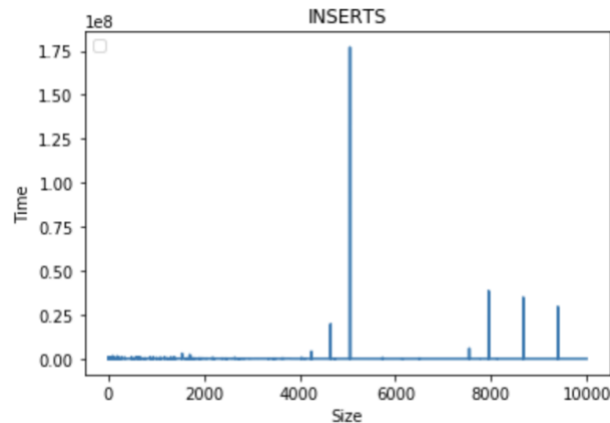
### Add operation:



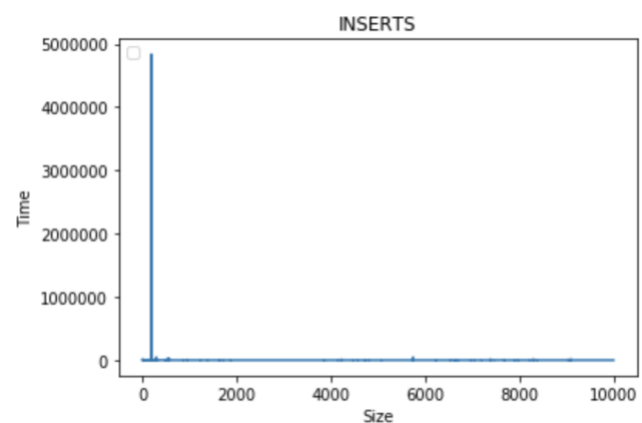
**BST – Total Time:** 1134489946



**AVL- Total Time:** 8481660272



**Red-Black Tree- Total Time:** 1122885339

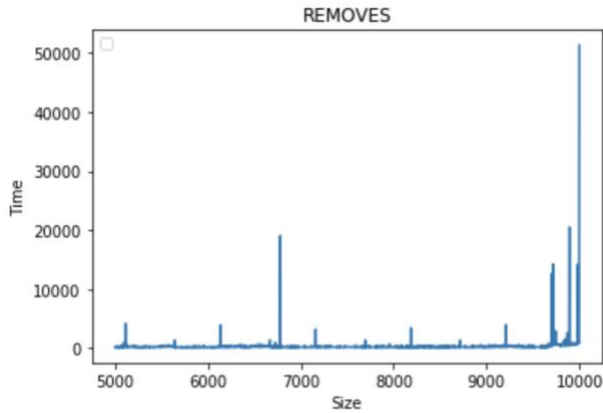


**SPLAY- Total time:** 10219735

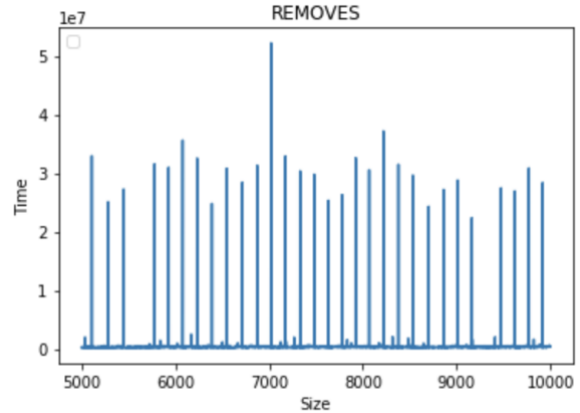
### Observations:

- During my analysis I found that Splay Tree has the best performance for insertion when data is skewed.
- Since BST is not balanced, it has the worst performance for insertion in this case.
- Following Splay Tree, Red Black tree has better performance for insertion than AVL trees, because it requires a smaller number of rotations.

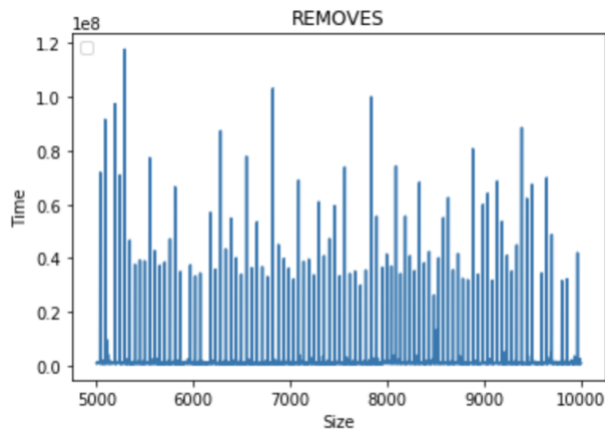
## Remove operation:



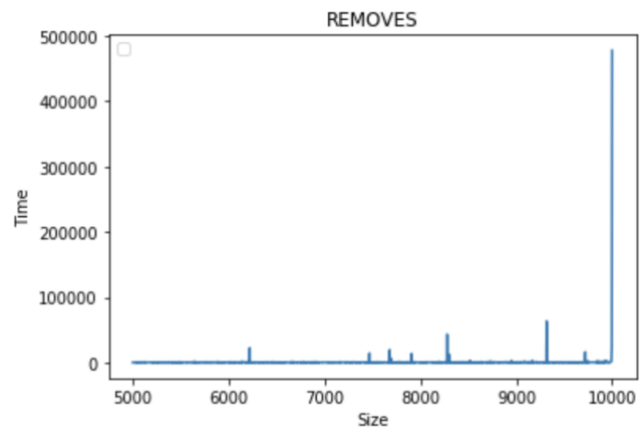
**BST – Total Time:** 1276743



**AVL- Total Time:** 2499779066



**Red-Black Tree- Total Time:** 9116085686



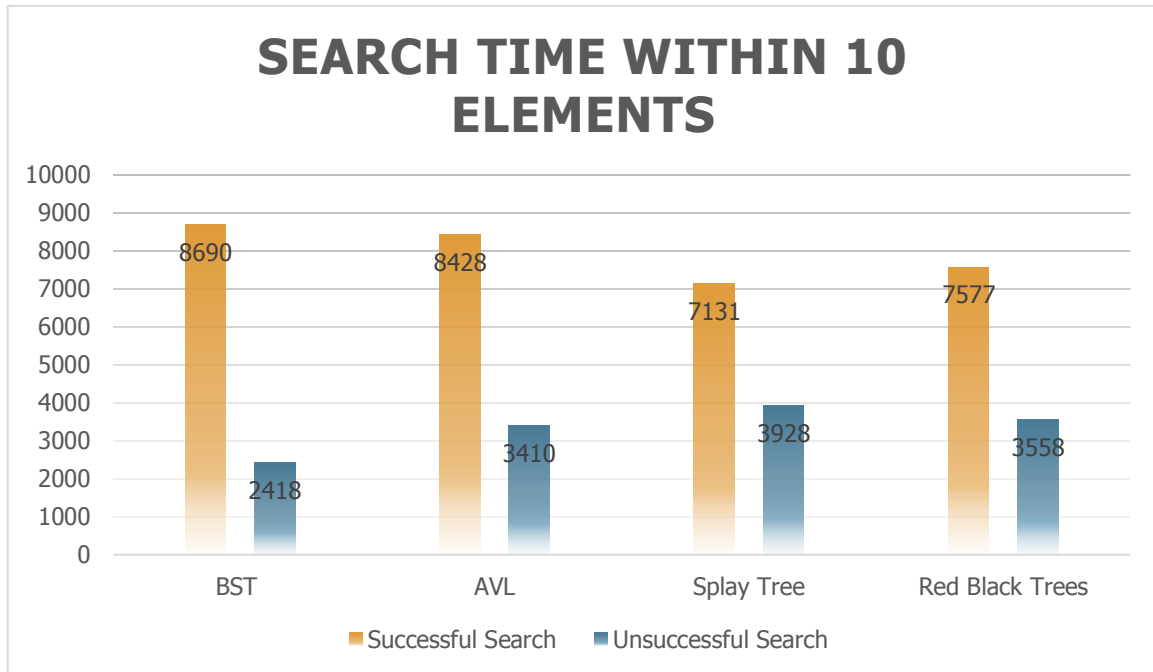
**SPLAY- Total time:** 2806933

## Observations:

- During my analysis I found that BST has the best performance for deletion since we removed elements in the order they were inserted.
- Following that splay tree performs constantly well in skewed and uniform data.
- Following Splay Tree, AVL has better performance for deletion than Red Black trees



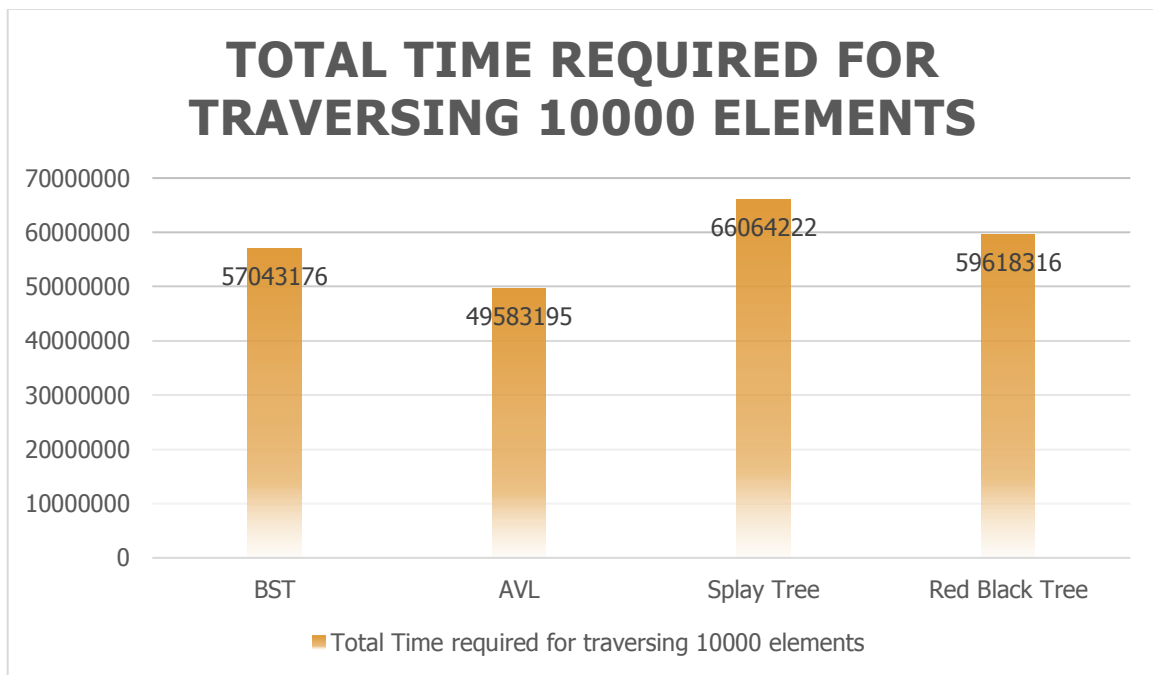
## Search Operation:



## Observations:

- During my analysis I found that for successful look ups, BST is the fastest.
- Following that, AVL tree < Red Black Trees < Splay Tree
- For Unsuccessful look ups, Splay Tree provided the best performance because of self-balancing.
- Following that Red Black Tree < AVL Tree < BST

## Traverse operation:



## Ideal Tree for different use-cases

From this analysis, we can say that, based on different use cases we have to choose different algorithms.

Use cases	Ideal Trees
When data is uniform, faster addition	BST
When data is uniform, faster removal	Splay Trees
When data is uniform, faster traversal	AVL Trees
When data is uniform, constant performance across all operations	Red Black tress/ Splay trees
When data is skewed, faster addition	Splay Trees
When data is skewed, faster removal	Splay Trees
Random data and constant performance across all operations	Splay Trees