DATA STRUCTURE & ALGORITHMS

# LAB 02 - RED BLACK TREES

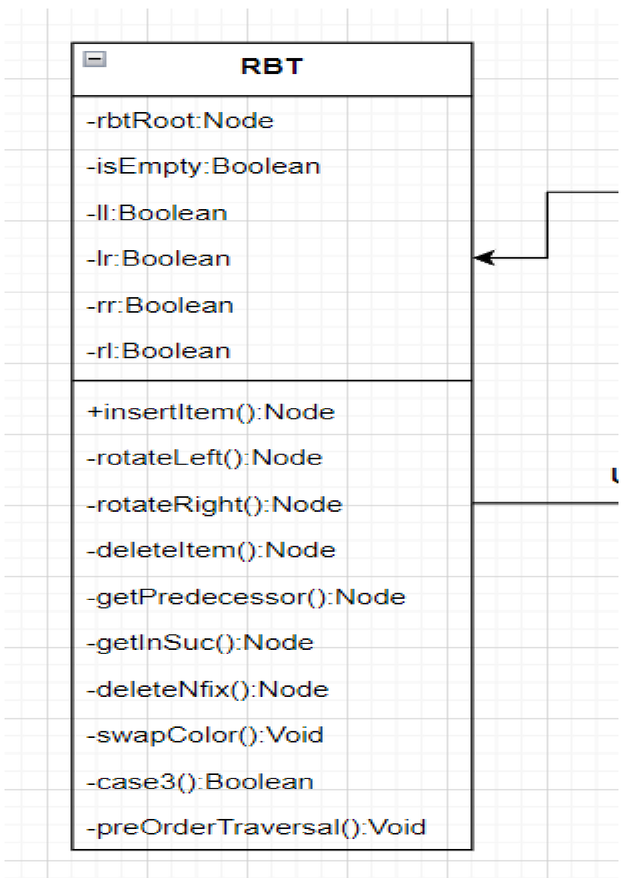

ZEYAD AHMED IBRAHIM ZIDAN 19015709

YOUSSEF SABER SAEED MOHAMMED 19016924


For an outlined view of the report, please consider reading it at **google documents.**

## Code Structure

Start by defining the red black tree class.UMl photo



Method clear which clears the tree. [The method and code is commented]

```
clear()
    rbtRoot=null
    isEmpty=true
    Return true
```

Method search.

```
Search(item)
    current=rbtRoot()
    while(current!=item and current!=null)
        if(current==item){
            Return current
        }
        if(item>current)
            current=current.right
        Else
            current=current.left
Return null
```

Method contains. (Similar to search, but varies in return type) and uses in insertion and find and deletion.

## The Insertion Algorithm

We define a normal function for insertion, which would then need help from other methods we will define and describe briefly in this section. Please be advised that the method is fully commented and self explanatory.

### Pseudocode

public *boolean* insert (item)

**If** the item is already inserted in the tree → return false; // Insertion Failed

**Else** → Unset the isEmpty attribute,

**If** there is no root → create node (item) and set it as the red black tree root.

Set the color to black and return true;

**If not** → insertItem (item, rbtRoot)

Set the red black tree root **parent** to **null** after each insertion //Corrects some errors.

**Return** true;

private *Node<T>* insertItem(item, root)

*boolean* conflict = false; // Tells if a double red conflict has occurred after insertion.

**If** insertion place is null → return **new** *Node<T>* (item);

**Else if** item < current item → insertItem (item, root.left) and set the parent

**If** a double red conflict occurred → Set the conflict to true;

**Else** → insertItem (item, root.right) and set the parent

**If** a double red conflict occurred → Set the conflict to true;

**If** sibling == root.parent.left{

**If** sibling is black

**If** the inserted node is a left child of the parent → right rotate the parent then left rotate the child

**Else** left rotate the parent

**Else**

Recolor the parent, sibling, and the grandparent if it is not the root

} **Else** {

**If** sibling is black

**If** the inserted node is a left child of the parent → right rotate the parent

**Else** left rotate the parent then right rotate the child

**Else**

Recolor the parent, sibling, and the grandparent if it is not the root

}

Unset the conflict and return;

```java
/**
     * Method that inserts a key. Returns true upon a successful insertion, false
     * otherwise.
     */
    @Override
    public boolean insert(T item) {
        // IF THE ITEM IS ALREADY INSERTED
        if (contains(item)) {
            System.out.println(item.toString() + " FOUND");
            return false;
        } else {
            isEmpty = false;
            // CHECK IF THE TREE HAS A ROOT NODE
            if (rbtRoot == null) {
                rbtRoot = new Node<T>(item);
                rbtRoot.setBlack(true);
                return true;
            }
            // IF SO, DO THE INSERTION
            rbtRoot = insertItem(item, rbtRoot);
            if (rbtRoot != null) {
                rbtRoot.parent = null;
            }
            return true;
        }

    }
```

The insert-item function helps with the insertion of a key as it does the rotations and determines if a conflict is encountered then handles it accordingly. This block of code shows the normal insertion process of a node.

```java
/**
     * Defining booleans to help with rotations within the red-black tree class
     */

    boolean ll = false;
    boolean lr = false;
    boolean rr = false;
    boolean rl = false;

    /**
     * Method to help with insertion of a key.
     */
    private Node<T> insertItem(T item, Node<T> root) {
        // DEFINING A DOUBLE RED CONFLICT PARAMETER
        boolean conflict = false;
        // IF THE GIVEN NODE IS NULL, WHICH IS ROOT IN THIS CASE, RETURN A CREATION OF A
        // NEW NODE WITH THE GIVEN KEY.
        if (root == null) {
            return new Node<T>(item);
        } else if (item.compareTo(root.getItem()) < 0) {
            // IF THE GIVEN ITEM IS LOWER THAN THE CURRENT ITEM, INSERT TO THE LEFT.
            root.left = insertItem(item, root.left);
            // SET THE PARENT OF THE INSERTED NODE TO CURRENT NODE.
            root.left.parent = root;

            // IF A DOUBLE RED IS ENCOUNTERED AFTER INSERTION, SET THE CONFLICT PARAMETER.
            if (root != this.rbtRoot) {
                if (!root.isBlack() && !root.left.isBlack()) {
                    conflict = true;
                }
            }
        } else if (item.compareTo(root.getItem()) > 0) {
            // IF THE GIVEN ITEM IS GREATER THAN THE CURRENT ITEM, INSERT TO THE RIGHT.
            root.right = insertItem(item, root.right);
            // SET THE PARENT OF THE INSERTED NODE TO THE CURRENT NODE.
            root.right.parent = root;

            // IF A DOUBLE RED IS ENCOUNTERED AFTER INSERTION, SET THE CONFLICT PARAMETER.
            if (root != this.rbtRoot) {
                if (!root.isBlack() && !root.right.isBlack()) {
                    conflict = true;
                }
            }
        }
```

Within the same function, this block shows how the function would deal with different kinds of rotations and rebalance the red-black tree.

```java
// DO ROTATIONS ACCORDING TO THE CONFLICT HAPPENED. THIS IS DONE RECURSEVILY.
        // THINK OF WHAT WOULD HAPPEN AFTER INSERTING THE NODE.
        if (this.ll) {
            System.out.println("Performing LL Rotation on " + root.getItem());
            root = rotateLeft(root);
            root.setBlack(true);
            root.left.setBlack(false);
            this.ll = false;
        } else if (this.rr) {
            System.out.println("Performing RR Rotation on " + root.getItem());
            root = rotateRight(root);
            root.setBlack(true);
            root.right.setBlack(false);
            this.rr = false;
        } else if (this.rl) {
            System.out.println("Performing RL Rotation on " + root.getItem());
            root.right = rotateRight(root.right);
            root.right.parent = root;
            root = rotateLeft(root);
            root.setBlack(true);
            root.left.setBlack(false);
            this.rl = false;
        } else if (this.lr) {
            System.out.println("Performing LR Rotation on " + root.getItem());
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.setBlack(true);
            root.right.setBlack(false);
            this.lr = false;
        }
```

Last but not least, this block of code shows how the method handles the conflict of a double red insertion, which we would encounter a lot in the process.

```
// SETTING THE ROTATION PARAMETERS ACCORDING TO CONFLICT TYPE, IF THERE IS A
        // CONFLICT.
        if (conflict) {
            // IF THE SIBLING (UNCLE) IS THE LEFT CHILD OF THE GRANDPARENT
            if (root.parent.right == root) {
                // AND THE COLOR OF SIBLING IS BLACK. (PLEASE NOT WE DIDN'T USE SENTINEL NODES
                // SO WE ASSUMED THEM TO BE NULL.)
                if (root.parent.left == null || root.parent.left.isBlack()) {
                    // AND IF THE INSERTED RED NODE IS A LEFT CHILD OF THE PARENT
                    if (root.left != null && !root.left.isBlack()) {
                        // THEN IT IS A RIGHT LEFT ROTATION
                        this.rl = true;
                    } else if (root.right != null && !root.right.isBlack()) {
                        // LEFT LEFT ROTATION OTHERWISE.
                        this.ll = true;
                    }
                } else {
                    // IF THERE IS NO ROTATION, SIMPLY RECOLOR THE NODES.
                    root.parent.left.setBlack(true);
                    root.setBlack(true);
                    if (root.parent != this.rbtRoot) {
                        root.parent.setBlack(false);
                    }
                }
            } else {
                // BUT IF THE SIBLING IS A RIGHT CHILD OF THE GRANDPARENT AND IS BLACK
                if (root.parent.right == null || root.parent.right.isBlack()) {
                    if (root.left != null && !root.left.isBlack()) {
                        // AND THE NEWLY INSERTED RED NODE IS A LEFT CHILD OF THE PARENT
                        // THEN IT IS AN RR ROTATION
                        this.rr = true;
                    } else if (root.right != null && !root.right.isBlack()) {
                        // LEFT RIGHT ROTATION OTHERWISE.
                        this.lr = true;
                    }
                } else {
                    // IF THERE IS NO ROTATION, SIMPLY RECOLOR.
                    root.parent.right.setBlack(true);
                    root.setBlack(true);
                    if (root.parent != this.rbtRoot) {
                        root.parent.setBlack(false);
                    }
                }
            }
            // UNSET THE CONFLICT AFTER BALANCING AND RECOLORING
            conflict = false;
        }
        return root;
    }
```

## The Deletion Algorithm

Simple delete method that returns true if a node with a given key is deleted successfully, false otherwise.delete it use contains if contains return false then it will return false we have three cases

Case 1 if node has no child just delete it and return it .

Case 2 if node has one child just delete it and return it that has two cases

Case 1 if node to be deleted is red

Delete it and replace it's child in it's place

Else

Case 1 if nood has right child

Get smallest element in right subTree

Else

Get biggest element in right subTree

Case 3 Has two child

Get biggest element in right subTree if that node is red replace data with child and delete child's Node

Else Get smallest element in right subTree element in right subTree if that node is red replace data with child and delete child's Node

The delete item method would be used to delete a node with a given key just as how it happens in a normal binary search tree. This method would be used in another delete helper method called **deleteNfix**.and return deleted node.

The **deleteNfix** function, which uses the previously defined delete item method, is used to delete and fix the red-black tree after deletion. This method also used a function called **resolveDoubleBlack**, another delete helper, to handle the double black cases after the deletion of a node.

First we check root if root null we clear tree

Else we have two cases

If node is red just done

Else call resolveDoubleBlack(Node,parent)

We pass parent because the node is deleted and parent no longer point to it

In this block of code, we define the first case of the double black resolve caused by the method **resolveDoubleBlack**.

Case 2 problem in root just remove double black

Case 3 if sibling is black and his children is black

If parent of node that has problem is red make it black and sibling red

Else parent is blak move problem to parent and make sibling red

```java
private Node<T> resolveDoubleBlack(Node<T> root, Node<T> parent) {
       // CASE 00
       if (root == rbtRoot) {
           return root;
       } else {
           // check sibling
           Node<T> sibling = root.getSibling();
           if (case3(sibling)) {
               if (!parent.isBlack()) {
                   parent.setBlack(true);
                   sibling.setBlack(false);
                   return parent;
               } else {
                   sibling.setBlack(false);
                   resolveDoubleBlack(parent, parent.parent);
                   return null;
               }
           }
```

In this block of code we handle another case of the double black resolve within the same method.

Case 4 happening if sibling is red swap color of parent and sibling we have two cases

Case 1 node that have problem is left child

Rotate parent left

Else rotate parent right

Case 5 happening if sibling nearest child is red and sibling is black

Swap color of sibling and his red child

If root is left child rotate right to sibling

Else rotate left to sibling

Case 6 happening if sibling farest child is red and sibling is black

Make that sibling child black

If root right child

Rotate parent rght

Else rotate left

**swapColor** is another delete helper, which was used within the **resolveDoubleBlack**.

## The Rotation Algorithm

```java
/**
     * Methods to help with the rotation within insertion
     */
    private Node<T> rotateLeft(Node<T> node) {
        System.out.println(node.getItem());
        Node<T> tempNode = node.right;
        Node<T> tempNode2 = tempNode.left;
        tempNode.left = node;
        node.right = tempNode2;
        node.parent = tempNode;
        if (tempNode2 != null) {
            tempNode2.parent = node;
        }
        return tempNode;
    }

    private Node<T> rotateRight(Node<T> node) {
        Node<T> tempNode = node.left;
        Node<T> tempNode2 = tempNode.right;
        tempNode.right = node;
        node.left = tempNode2;
        node.parent = tempNode;
        if (tempNode2 != null) {
            tempNode2.parent = node;
        }
        return tempNode;
    }

```

## The Traversal Algorithm

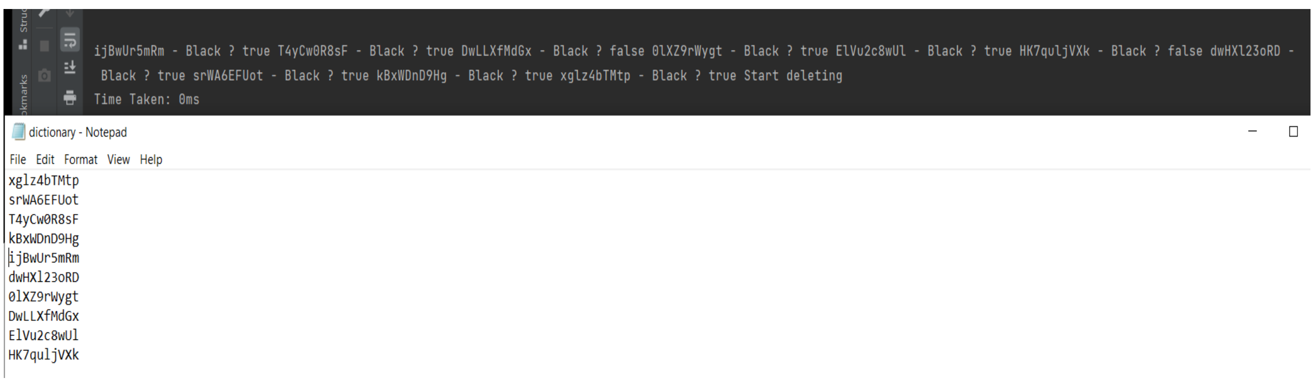
We use pre order to print our tree

## The Random String Generator

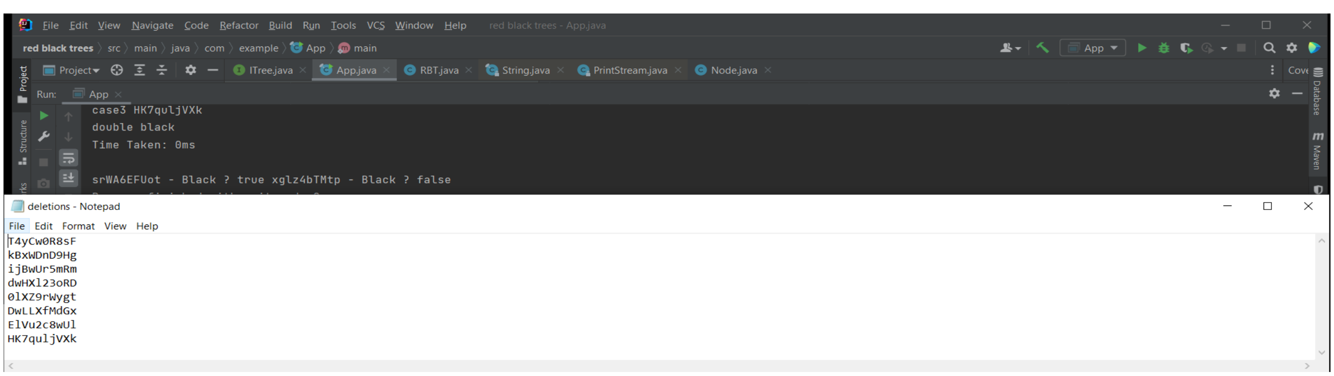
This simple application helped us generate a list of random strings of whatever size we wanted. All we needed every time was to change the index limit as we wanted through the static main method.And it was used in test case generation.

# Test Cases

Insertion




Deletion



# Insertion

```
iQuV7T1fMd - Black ? true JPXhisw7FJ - Black ? true EP8fmstGDs - Black ? false 7Cxti2Gjpp - Black ? true 73vFpRmBI5 - Black ? false 93yUWhAPKT - Black ? false HBU8OShK8k -
 Black ? true HISIBb8Z8t - Black ? false aYO1pmfdIK - Black ? false PniAdU8Yzm - Black ? true VFvCVBZG8n - Black ? false ajA8xyfkau - Black ? true mfBY8dbIJD - Black ? true
 l6vSm1kydI - Black ? true iUWTF6FTH3 - Black ? false lQ2JxtcKq3 - Black ? false siKkemWuFn - Black ? false sUoqVqAzt7 - Black ? true ofakddloWU - Black ? false uNWvJz7zYj -
  Black ? true Del 0073vFpRmBI5
```

dictionary - Notepad

File Edit Format View Help

```
lQ2JxtcKq3
aYO1pmfdIK
EP8fmstGDs
ajA8xyfkau
iQuV7T1fMd
uNWvJz7zYj
JPXhisw7FJ
mfBY8dbIJD
l6vSm1kydI
iUWTF6FTH3
7Cxti2Gjpp
93yUWhAPKT
73vFpRmBI5
PniAdU8Yzm
HBU8OShK8k
siKkemWuFn
VFvCVBZG8n
HISIBb8Z8t
sUoqVqAzt7
ofakddloWU
```

# Deletion

```
treee
iQuV7T1fMd - Black ? true 93yUWhAPKT - Black ? true 7Cxti2Gjpp - Black ? true JPXhisw7FJ - Black ? true mfBY8dbIJD - Black ? true l6vSm1kydI - Black ? true iUWTF6FTH3 -
 Black ? false uNWvJz7zYj - Black ? true ofakddloWU - Black ? false
Process finished with exit code 0
```

Version Control Debug Run TODO Problems Dependencies Profiler Terminal Build

deletions - Notepad

File Edit Format View Help

```
73vFpRmBI5
PniAdU8Yzm
HBU8OShK8k
siKkemWuFn
VFvCVBZG8n
HISIBb8Z8t
sUoqVqAzt7
lQ2JxtcKq3
aYO1pmfdIK
EP8fmstGDs
ajA8xyfkau
lQ2JxtcKq3
```

# Insertion

```
GG2KEh8RTD - Black ? true 96QFnxpur4 - Black ? true 1sPtLZR2Rh - Black ? true 0Jsv0OjoRL - Black ? false 7f35z93auT - Black ? false CDdq09D6Oy - Black ? true GCV0WTiPj4 - Black ?
false XDvT6I8gWB - Black ? false KgjlgY4t0E - Black ? true GkifZMwJ0h - Black ? true VmfVCBTZSF - Black ? true LDbzCXvOtO - Black ? false lLVXdPS7mI - Black ? true e3oMC7hqOf -
Black ? true e2Cq5qlpb5 - Black ? false iqCIfWTV5B - Black ? false sBzYp9GwM1 - Black ? false rW8Ju3VLBs - Black ? true xRECFNC6Mm - Black ? true tp0BHeF5jh - Black ? false
```

dictionary - Notepad

File Edit Format View Help

```
xRECFNC6Mm
GG2KEh8RTD
96QFnxpur4
XDvT6I8gWB
lLVXdPS7mI
rW8Ju3VLBs
VmfVCBTZSF
7f35z93auT
CDdq09D6Oy
e3oMC7hqOf
0Jsv0OjoRL
GkifZMwJ0h
iqCIfWTV5B
sBzYp9GwM1
1sPtLZR2Rh
KgjlgY4t0E
LDbzCXvOtO
GCV0WTiPj4
tp0BHeF5jh
e2Cq5qlpb5
```

## Deletion

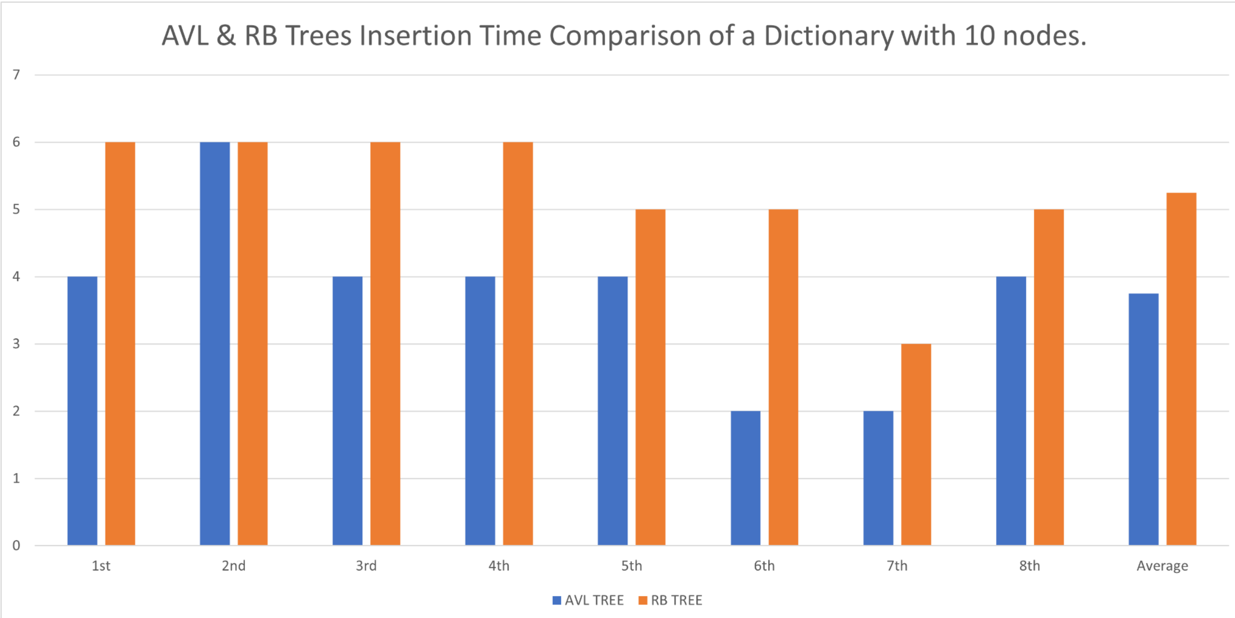

## Insertion

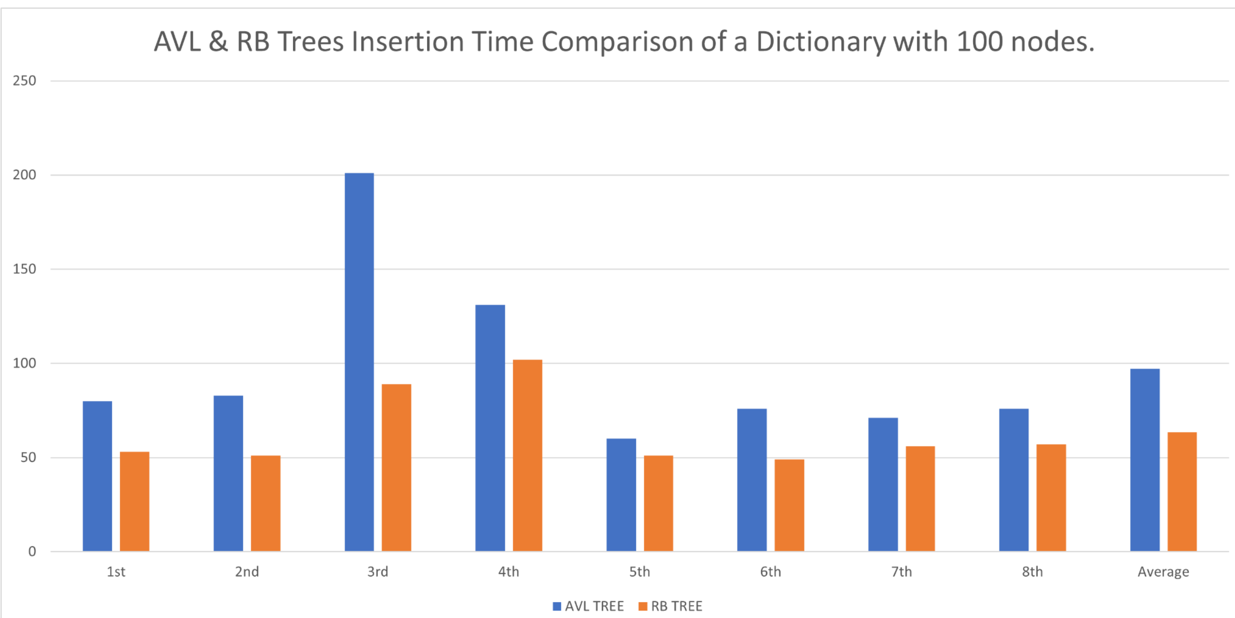

## Deletion

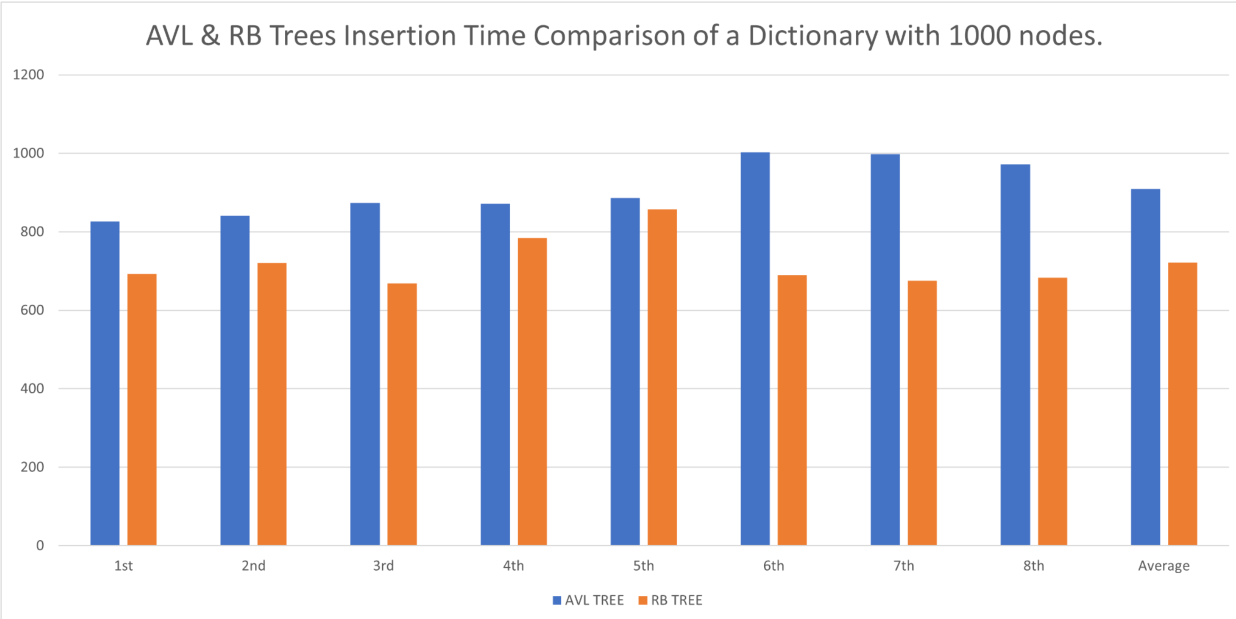

# Comparison

## Insertion

- In the following charts we tried to insert 10 words of length 10 characters to an AVL Tree and RED-BLACK Tree, printed out the time in milliseconds, then finally constructed the comparison chart of each case.
- We tried insertion of 10 nodes of 2 random-generated-words distinct dictionaries and ran each file 4 times, so that makes it 8 runs for each data structure.
- The time is on the **Y-AXIS** and is in **milliseconds**.
- The run count is on the **X-AXIS** as long as the average result.

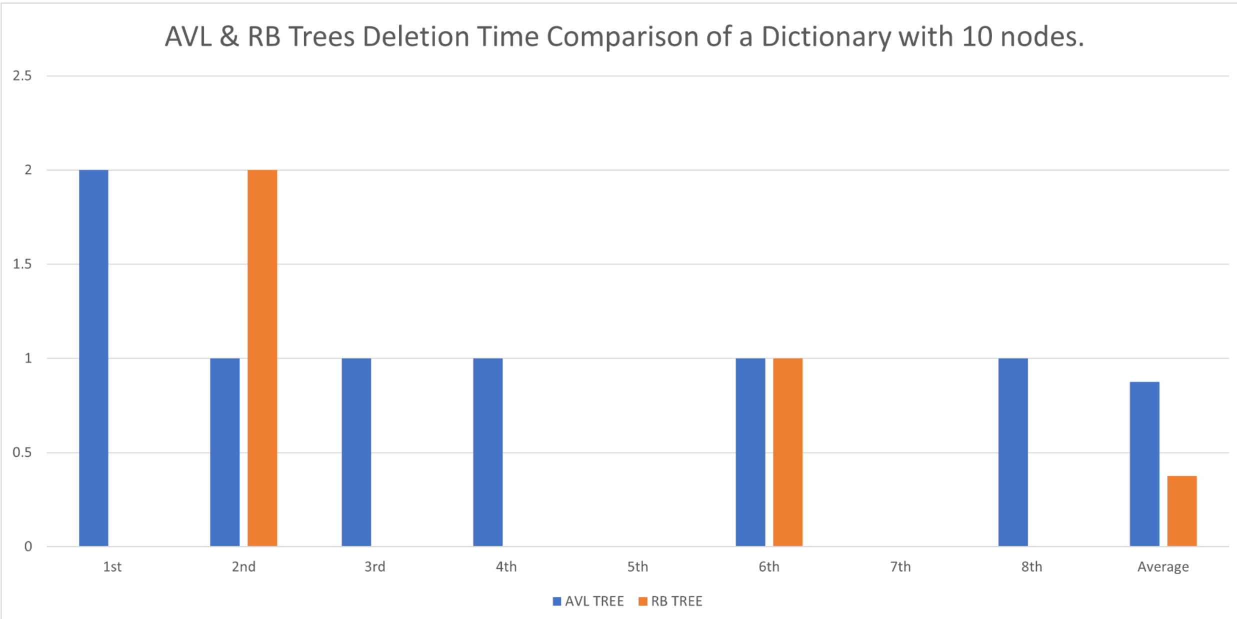

AVL & RB Trees Insertion Time Comparison of a Dictionary with 10 nodes.
7
6
5
4
3
2
1
0
1st
2nd
3rd
4th
5th
6th
7th
8th
Average
AVL TREE
RB TREE

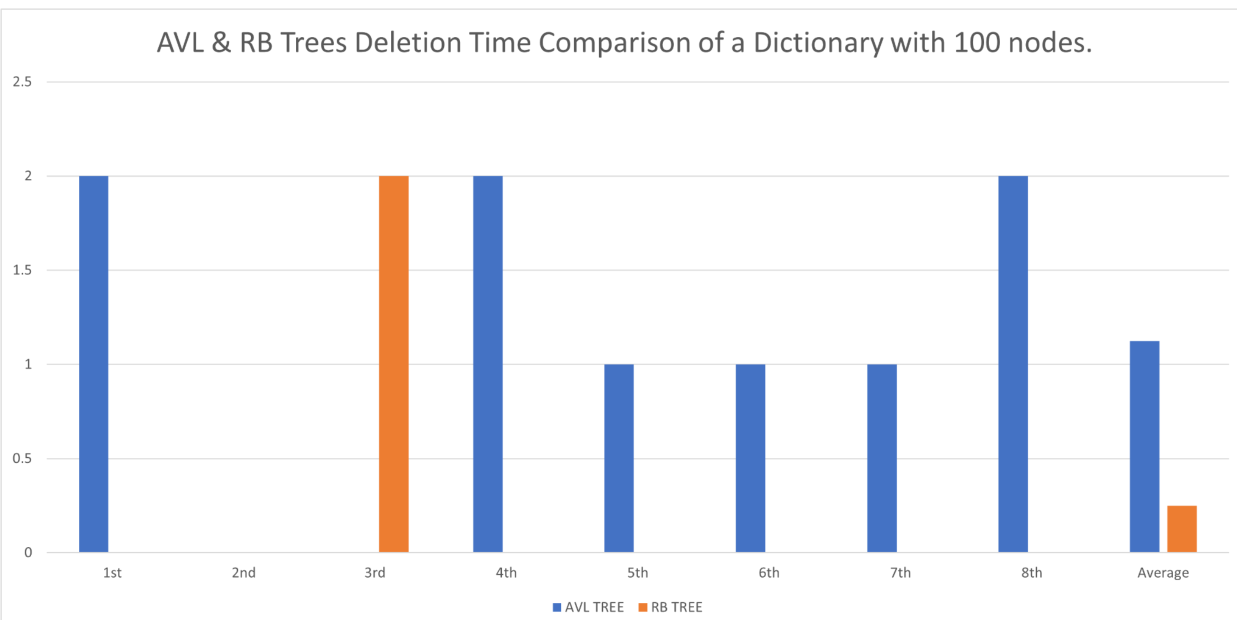

AVL & RB Trees Insertion Time Comparison of a Dictionary with 100 nodes.
250
200
150
100
50
0
1st
2nd
3rd
4th
5th
6th
7th
8th
Average
AVL TREE
RB TREE

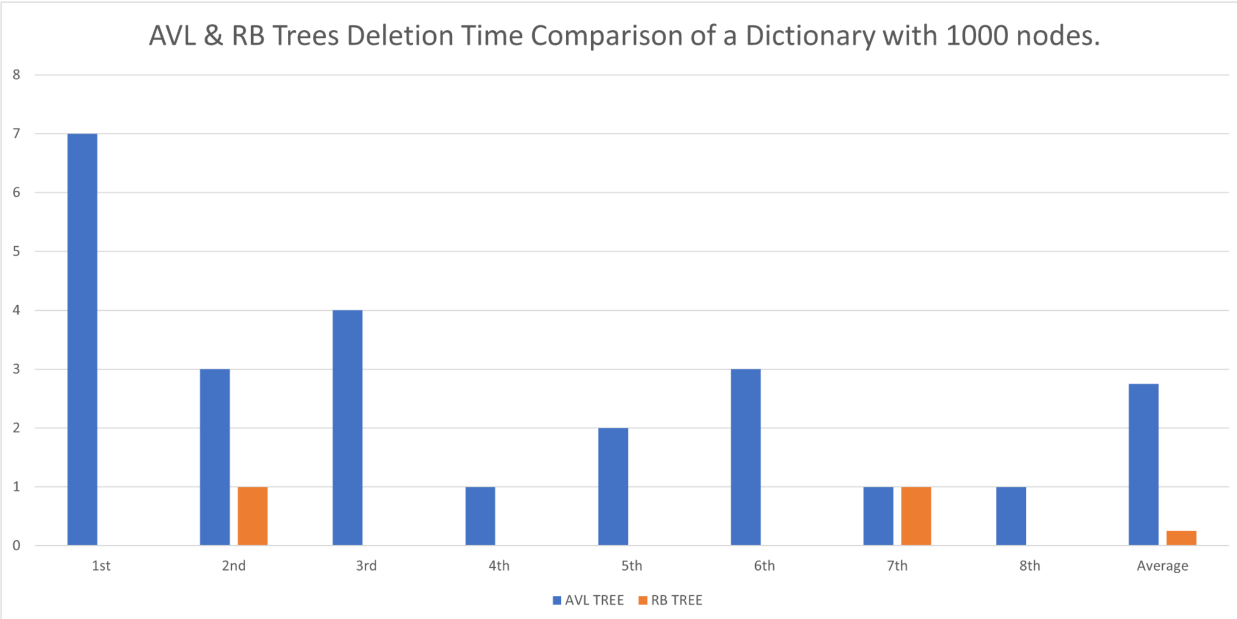

AVL & RB Trees Insertion Time Comparison of a Dictionary with 1000 nodes.
1200
1000
800
600
400
200
0
1st
2nd
3rd
4th
5th
6th
7th
8th
Average
AVL TREE
RB TREE

## Deletion

For deletion, we have done the same approach and process taken in the insertion analysis.






As we observe from the charts, we can see that Red Black Trees are considerably much faster than the AVL Trees.

## Time Complexity

**Red Black Tree:** Search, Insertion, and Deletion is a O(log n) where n is the total number of nodes in the red-black tree. Whereas, the space complexity of the red-black tree is O(n).

## Difference between Red-Black Tree and AVL Tree

| Red-Black Tree | AVL Tree |
|---|---|
| It does not provide efficient searching as red-black tree are roughly balanced | It provides efficient searching as AVL trees are strictly balanced |

| Red-Black Tree | AVL Tree |
|---|---|
| Insertion and deletion operation is easier as require less number of rotation to balance the tree | Insertion and deletion operation is difficult as require more number of rotation to balance the tree |
| The nodes are either red or black in color | The nodes have no colors |
| It does not contain any balance factor to balance the height of the tree | It contains the balance factor to maintain the difference in the height of the tree. |
| Mostly used for insertion and deletion operations | Mostly used for searching operations |

## Advantages of Red-Black Tree

- Red-black tree balance the height of the binary tree
- Red-black tree takes less time to structure the tree by restoring the height of the binary tree
- The time complexity for search operation is O(log n)
- It has comparatively low constants in a wide range of scenarios

## Disadvantages of Red-Black Tree

- Relatively complicated to implement
- The red-black tree is not rigidly balanced in comparison to the AVL tree

## Conclusion

The red-black tree is one of the members of the binary search tree which helps to maintain the height of the binary tree just like the AVL tree. Every node in the binary search tree is colored either red or black which further helps to maintain the properties of the tree and provides an effective and efficient method for insertion and deletion operations of the node in the binary tree by undergoing a fewer rotation.