

# Implementation and Visualization of Red-Black Tree

**Prashant Kanth**

  
Dept. of Computer  
Science  
Rutgers University-  
New Brunswick

**Parth Goel**

  
Dept. of Computer  
Science  
Rutgers University-  
New Brunswick

**Rishika Bhanushali**

  
Dept. of Computer  
Science  
Rutgers University-  
New Brunswick

**Utkarsh Jha**

  
Dept. of Computer  
Science  
Rutgers University-  
New Brunswick

---

**Abstract:** Data Structures are a specialized means of organizing and storing data in computers in such a way that can perform operations on the stored data more efficiently. Out of the numerous data structures present, binary search trees play an important role when it comes to efficient operations. In computer science, a self-balancing binary search tree is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions. Examples of these are AVL trees, Red-Black trees, Splay trees etc. A red-black tree is a special kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that tree remains balanced during insertions and deletions. In this project we attempt to implement the red-black tree and investigate its efficiency by analyzing time and space complexity. The end goal is then to visualize the red-black tree operations with the help of a user interface.

---

## I. INTRODUCTION

A **red-black** tree is a binary search tree. It is self-balancing like AVL tree, though it uses different properties maintain the invariant of being balanced. It has one extra bit of storage per node: its **color**, which can be either **RED** or **BLACK**. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so the tree is approximately balanced.

Balanced binary search trees are much more efficient at search than unbalanced binary search trees, so the complexity needed to maintain balance is often worth it. Red-Black trees maintain a slightly looser height invariant than AVL trees. Because the height of the red-black tree is slightly larger, lookup will be slower in a red-black tree. However, the looser height invariant makes insertion and deletion faster. Also, red-black trees are popular due to relative ease of implementation.

Each node of the tree contains the attributes *color*, *value*, *left*, *right*, *parent*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following properties:

- Every node is either red or black.
- The root node is always black, or it can be always changes from red to black.
- Every leaf (NIL) is black.

- Every new node must be inserted with RED color.
- If a node is red, then both its children are black. (There should not be any two consecutive parent-child relation between RED nodes.)
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Below is an example of a red-black tree:

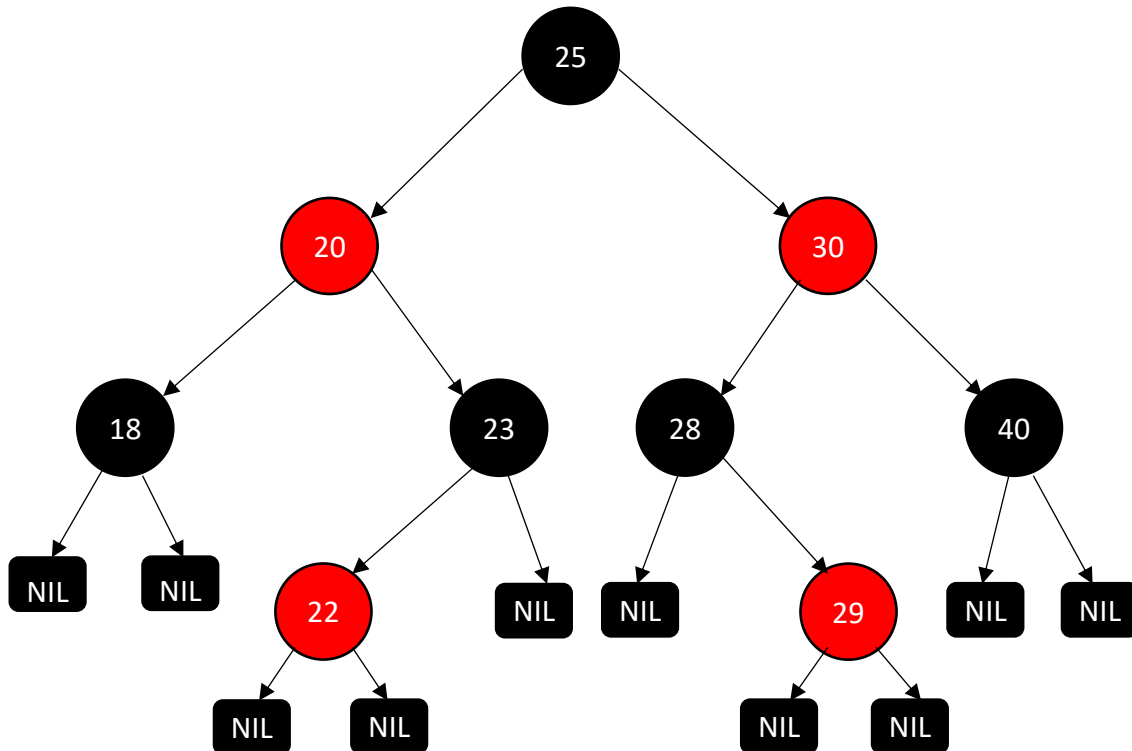


Fig 1.1 Example of red-black tree

Although there are constraints on the red nodes, there are none for the black nodes. A complete binary tree with all nodes as black nodes is still considered a red-black tree.

We assume that whenever there are no children of a node, NIL nodes take place of the children, so NIL nodes are always leaf nodes, and hence for further visualization we chose to not display NIL nodes and just assume that they are always at the leaf.

## II. REQUIREMENT GATHERING

For the requirement gathering stage of red-black tree implementation, we had to understand the working of Binary Search Tree (BST) as well as Adelson-Velsky and Landis (AVL) tree, since most of the properties/concepts of Binary Search Tree and AVL trees are common with Red-Black Trees.

Binary Search Tree has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.

- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

AVL Tree has the following properties:

- In an AVL tree, the heights of the two child subtrees of any node differ by at most one and therefore, it is also said to be height balanced.
- Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
- The balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with a balance factor 1, 0, or -1 is considered balanced.

In red-black trees, we perform operations like rotation and recoloring to balance the tree if they violate any of the properties that defines a red-black tree.

### III. DESIGN

In design phase, we look at different sequence of actions that are performed on the red-black tree. Let us try to understand the design of a red-black tree.

We will deal with integer key values throughout the course of this project. The leaf nodes of red-black trees (NIL) do not contain keys or data. All new nodes are inserted with RED color. However, the first node that is inserted into the tree is a root node, and its color is changed to black since the root node in a red-black tree should always be black. All subsequent new nodes are inserted with red color, and since as per the properties of red-black tree whenever two adjacent nodes become red, the tree needs to be balanced using combination of two operations, recoloring and rotation. The deletion operation too uses combination of recoloring and rotation to keep the tree from violating its properties.

The program essentially starts by asking the user to input the node key they want to insert in the red-black tree. The *insert(int value)* method takes as input the key to be inserted into the tree and places the node in a proper position according to its key and a proper color is assigned. If there is any violation of properties, *adjustTreeInsertion(Node node)* method performs necessary operations to balance the tree.

Similarly, when a user performs a deletion, the removal of a node starts with a call to the *delete(int value)* method, which first finds the position of the node in the tree, if not found, user gets a message that key was not found in the tree. If the key is found, the function finds either the maximum children in its left subtree (predecessor) or minimum children in its right subtree (successor). One thing to keep in mind is that, just like BSTs, we can't delete internal nodes in Red-Black tree and hence the function replaces the appropriate node with the node to be deleted and then removes the link of the leaf node (one above the NIL nodes) with the tree. Once the deletion is done, based on different cases, *adjustTreeDeletion(Node node)* method helps to balance the tree, and then the control is given back to the user for either insertion or deletion of a node.

The user can also search for a desired key in the red-black tree which is performed by *search(int value)* method. If the value is not found in the tree, user gets an appropriate message. Few other methods that help in balancing the tree are *rotateRight(Node node)*, *rotateLeft(Node node)*, *replace(Node del, Node leaf)* and *successor(Node node)*. We now look at different cases under Insertion and Deletion in a red-black tree.

### Insertion in Red-Black Tree:

We can insert a node into an  $n$ -node red-black tree in  $O(\lg n)$  time. To guarantee that the red-black tree properties are preserved, we call an auxiliary procedure *adjustTreeInsertion* to recolor nodes and perform rotations. The call *insert(value)* inserts a new node, whose key (value) is assumed to have already filled in, into the red-black tree.

To understand how, *adjustTreeInsertion* works, let us determine what violations of the red-black properties are introduced in *insert* when a new node is inserted and colored red. We need to consider six cases, but three of them are symmetric to the other three, depending on whether violated node's parent is left or right child.

- If tree is empty, create a new node as root node with color black.
- If tree is not empty, create new node as leaf node with color red.
- If parent of new node is black, then exit.
- If parent of new node is red, then check the color of its parent's sibling (uncle)
  - If color is black or null, then do suitable rotation and recolor. The rotation operation is like the one involved in AVL trees to balance height.
  - If color is red, then recolor parent and parent's sibling. And check if parent's parent (grandfather) of new node is not a root node, if grandfather is not a root node, then recolor it and re-check for other cases.

Below is an example of red-black tree insertion for a better understanding of different operations involved.

We'll insert below keys in the red-black tree one-by-one:

10, 18, 7, 15, 16

First, we insert 10:

If tree is empty, we create new node, root node, with black color



Fig 1.2 Insertion in Red-Black Tree

So, now we have:

18, 7, 15, 16

Insert 18 into the tree:

New nodes are always inserted with **RED** color and the tree maintains BST status. 18 is greater than 10, so we insert it as right child of 10.

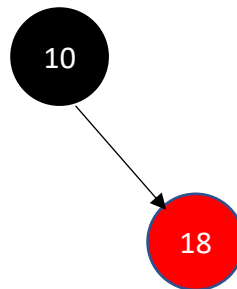


Fig 1.3 Insertion in Red-Black Tree

We are left with below keys:

7, 15, 16

Insert 7 into the tree:

New nodes are inserted with **RED** color and the tree maintains BST status. 7 is smaller than 10, so we insert it as left child of 10.

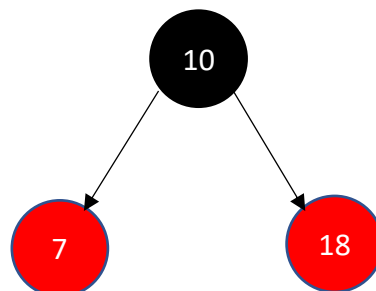


Fig 1.4 Insertion in Red-Black Tree

We are left with below keys:

15, 16

Insert 15 into the tree:

New nodes are inserted with **RED** color and the tree maintains BST status and color scheme. 15 is greater than 10, 15 is less than 18, so we insert 15 as left child of 18.

We have RED-RED violation:

*adjustTreeInsertion* is initiated

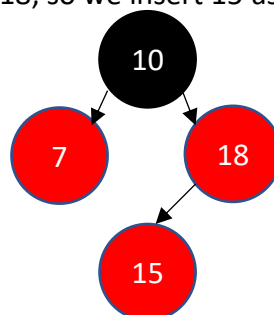


Fig 1.5 Insertion in Red-Black Tree

Parent of 15 is a red node, check the color of parent's sibling, if color is **RED** then recolor parent and parent's sibling. And check if parent's parent (grandfather) of new node is not a root node. If grandfather is not a root node, then recolor it and re-check for other cases.

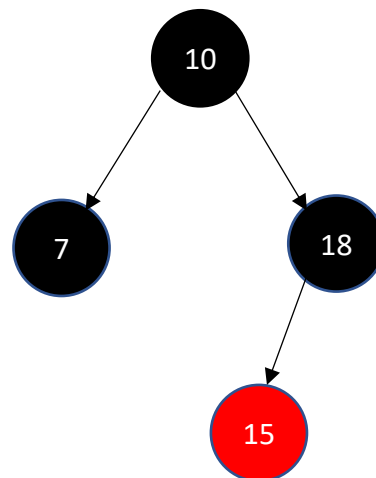


Fig 1.6 Insertion in Red-Black Tree

We are left with below key:

16

Insert 16 into the tree:

New nodes are inserted with **RED** color and the tree maintains BST status, color scheme and height. 16 is greater than 10, 16 is less than 18, 16 is greater than 15. So, we insert 16 as right child of 15.

Parent of new node is **RED**, check the color of parent's sibling, if color is black or NULL then do suitable rotation and recolor.

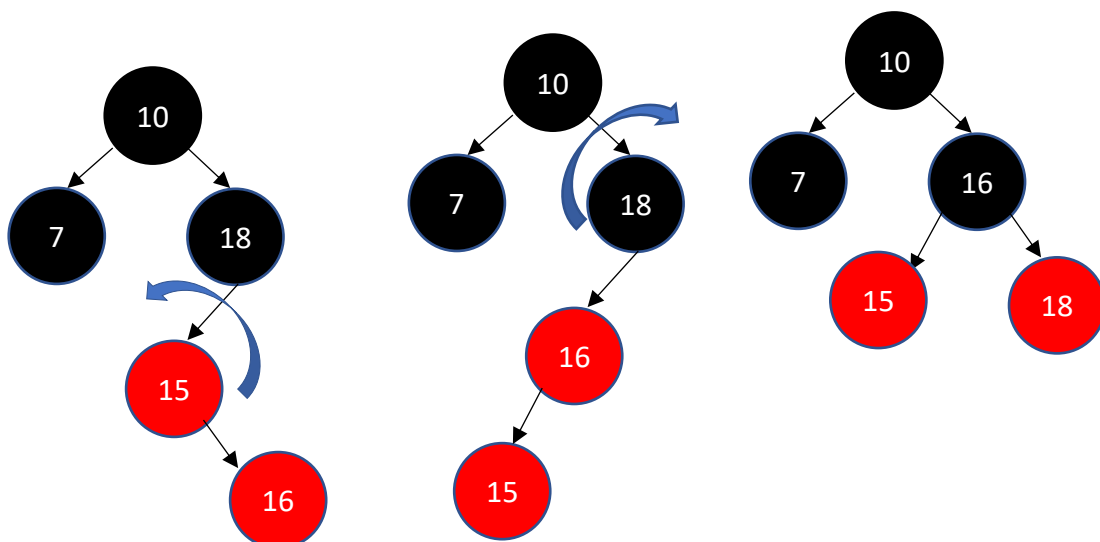


Fig 1.7 Insertion in Red-Black Tree

Perform a left rotation on node with key 15

Perform a right rotation on node with key 18

Recolor. Now the Red-Black Tree is balanced again

## Deletion in Red-Black Tree:

Like other basic operations on an  $n$ -node red-black tree, deletion of a node takes  $O(\lg n)$  time. Deleting a node from a red-black tree is a bit more complicated than inserting a node. To guarantee that the red-black tree properties are preserved, we call an auxiliary procedure *adjustTreeDeletion* to recolor nodes and perform rotations. The call *delete(value)* finds an appropriate successor/predecessor at a leaf position for a node to be deleted, replaces them, and then delete the leaf node. Since we cannot delete internal nodes, we must find predecessor/successor just like in BSTs.

To understand the *delete* operation better, we should first have an idea of DOUBLE-BLACK concept.

So, what is **DOUBLE-BLACK**?

Let us try to understand this with help of an example. Consider below red-black tree.

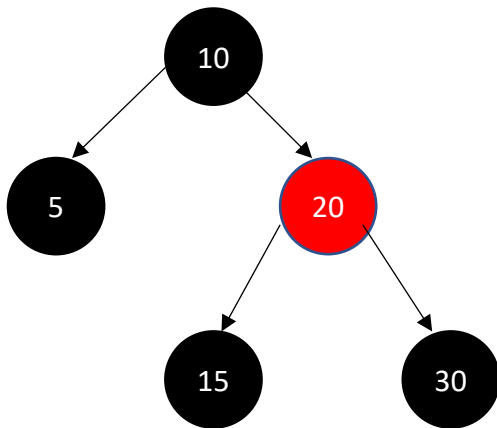


Fig 1.8 DOUBLE-BLACK Example

Suppose we wish to delete node 15 here. Now since 15 is a last node and BLACK, when we delete it, it gets replaced with NIL nodes. Remember NIL nodes are also BLACK. So, when it takes up the color of deleted node in an attempt to preserve Red-Black property, it becomes DOUBLE-BLACK.

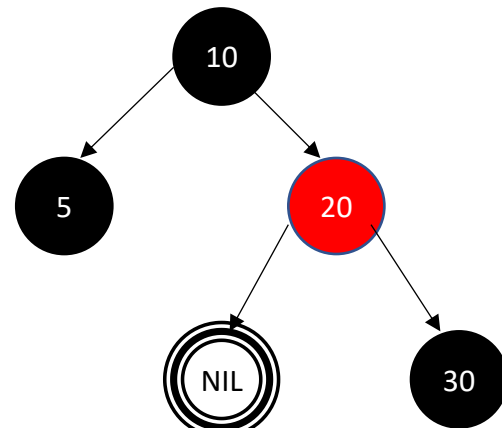


Fig 1.9 DOUBLE-BLACK Example

This is how we chose to represent a **DOUBLE-BLACK** node in RED-BLACK tree.

After deletion, all we need to do for a red-black tree to be balanced again is to get rid of this DOUBLE-BLACK. The entire problem is now drilled down to get rid of this DB (DOUBLE-BLACK). Our auxiliary procedure *adjustTreeDeletion* performs this task and ensure that the tree is balanced again and holds all its properties.

To understand how, *adjustTreeDeletion* works, let us determine what violations of the red-black properties are introduced in *delete* when a node is deleted from red-black tree. We will discuss the unique cases and then we can mirror the same steps depending on whether violated node's sibling is either left or right child.

The first always is to perform BST deletion and look at below cases:

- If node to be deleted is red, just delete it.
- If root is DOUBLE-BLACK, just remove DOUBLE-BLACK (DB).
- If DB's sibling is black and both its children are also black:
  - Remove DB
  - Add Black to its parent
    - If parent is red, it becomes black
    - If parent is black, it becomes DB
  - Make sibling red
  - If DB still exists, apply other cases
- If DB's sibling is black and sibling's child who is far from DB is black, but near child to DB is red
  - Swap color of DB's sibling and sibling's child who is near to DB
  - Rotate sibling in opposite direction to DB
  - Apply next case (we will always need to apply the next case whenever we apply this case)
- If DB's sibling is black and sibling's child who is far from DB is red
  - Swap color of parent and sibling
  - Rotate parent in DB's direction
  - Remove DB
  - Change color of red child to black
- If DB's sibling is red:
  - Swap color of parent and sibling
  - Rotate parent in DB's direction
  - Reapply cases

Below is an example of red-black tree deletion for a better understanding of different operations involved. Consider below red-black tree:

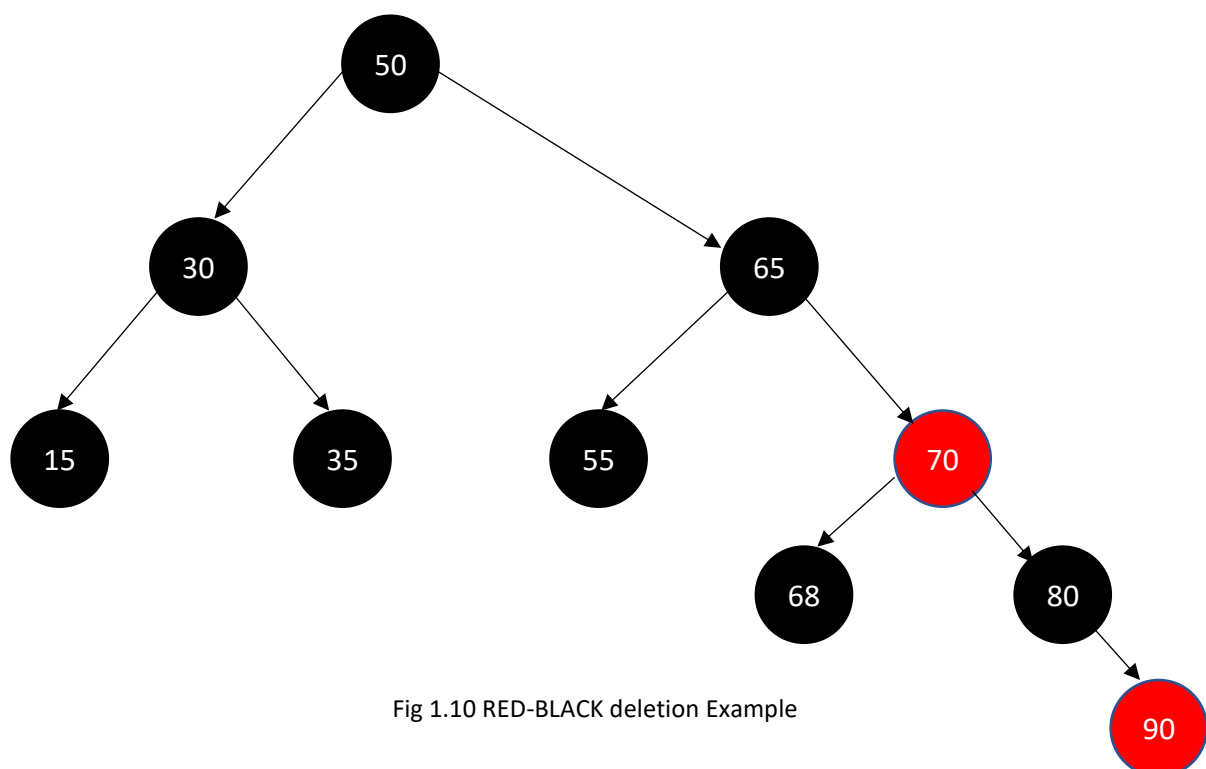


Fig 1.10 RED-BLACK deletion Example



We will delete below nodes from RED-BLACK Tree one-by-one:

50, 30, 90

*delete(50)*

Perform BST deletion by finding successor of 50 in the tree. Successor of 50 is 55. 55 is black, when we remove this, it becomes DOUBLE-BLACK. Replace 50 with 55. Now, work to remove DOUBLE-BLACK.

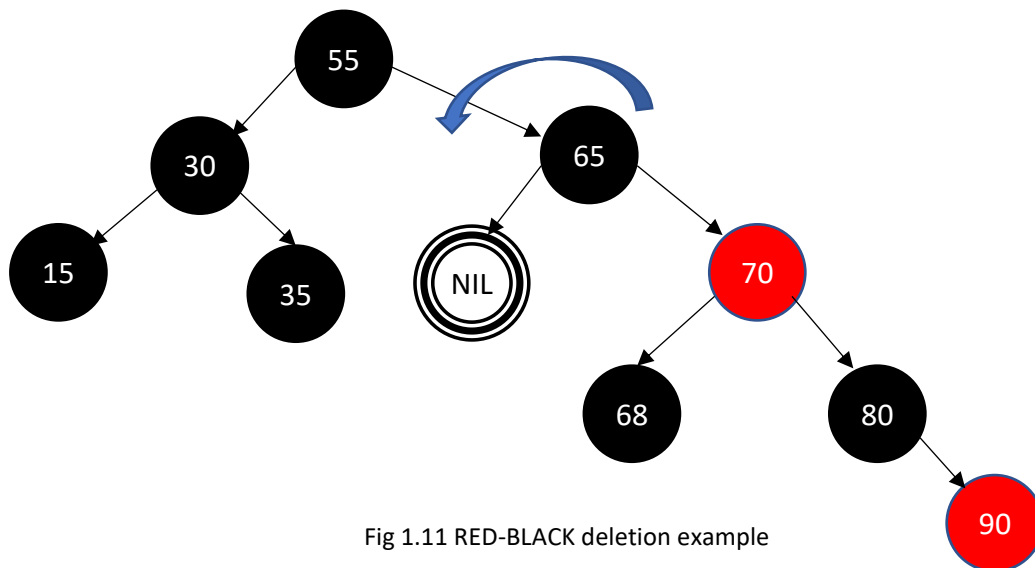


Fig 1.11 RED-BLACK deletion example

DB's sibling is RED. Apply the steps involved in that case:

Swap color of parent and sibling and rotate parent in DB's direction

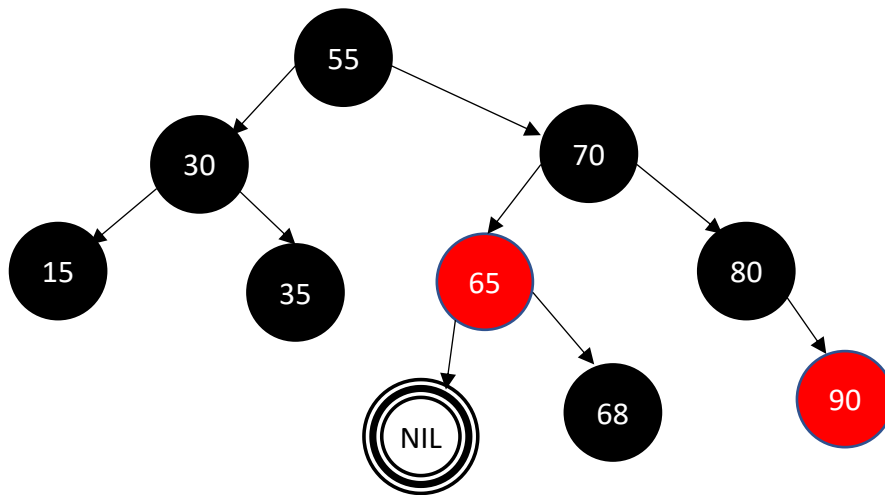


Fig 1.12 RED-BLACK deletion example

DB's sibling is black, and both its children (NIL nodes) are black. Apply steps involved, remove DB and add black to its parent, make sibling red.

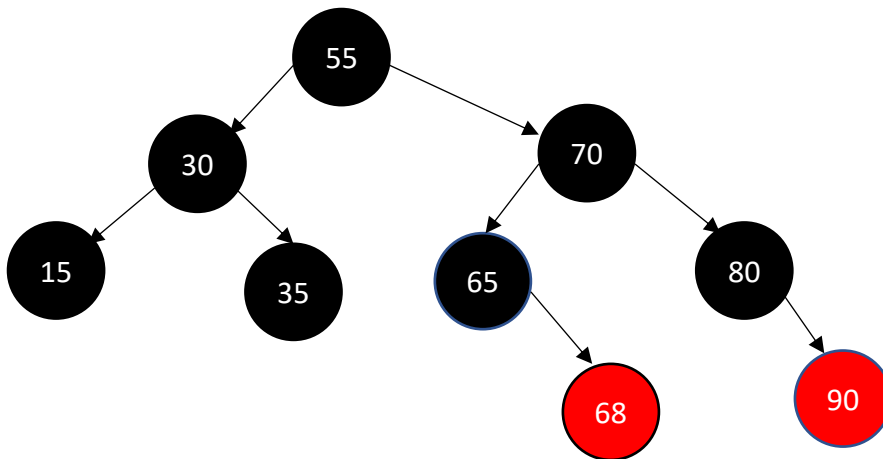


Fig 1.13 RED-BLACK deletion example

Now, we have below keys left to delete:

30, 90

*delete(30)*

Perform BST deletion by finding successor of 30 in the tree. Successor of 30 is 35. 35 is black, when we remove this, it becomes DOUBLE-BLACK. Replace 30 with 35. Now, work to remove DOUBLE-BLACK.

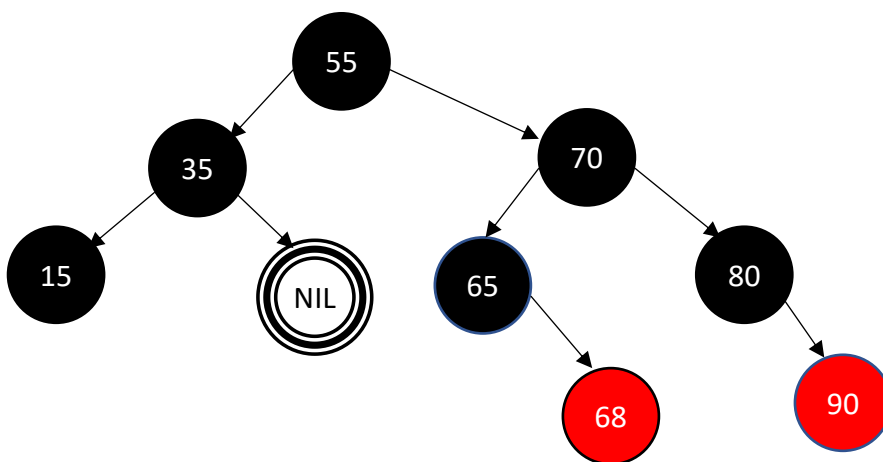


Fig 1.14 RED-BLACK deletion example

DB's sibling is black, and both its children (NIL nodes) are black. Apply steps involved, remove DB and add black to its parent, make sibling red. Now 35 becomes DB.

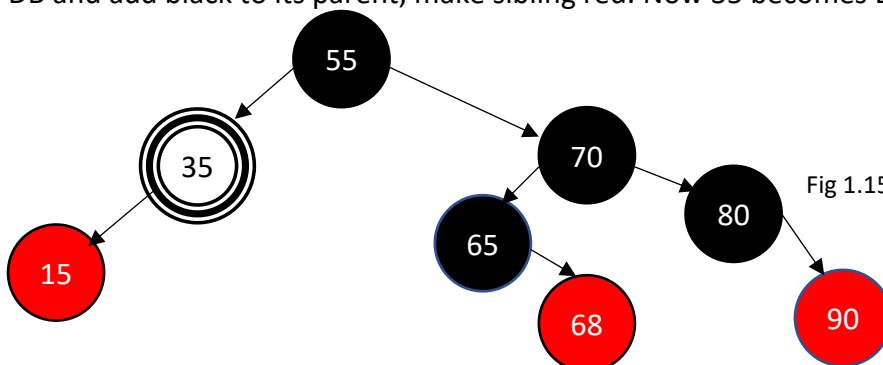


Fig 1.15 RED-BLACK deletion example

DB's sibling is black, and both its children (NIL nodes) are black. Apply steps involved, remove DB and add black to its parent, make sibling red.

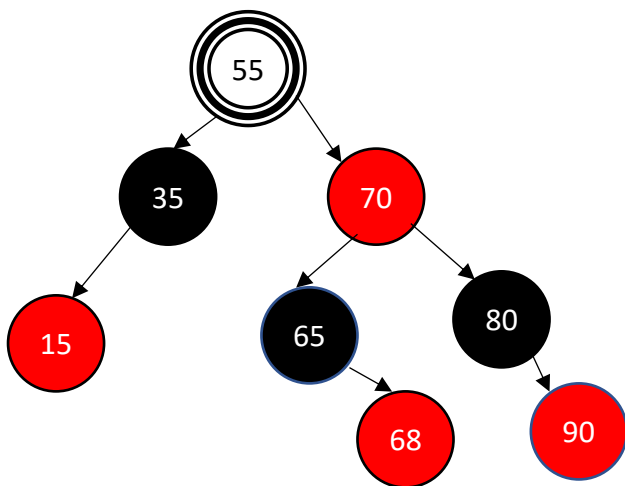


Fig 1.16 RED-BLACK deletion example

If root is DB, just remove DB

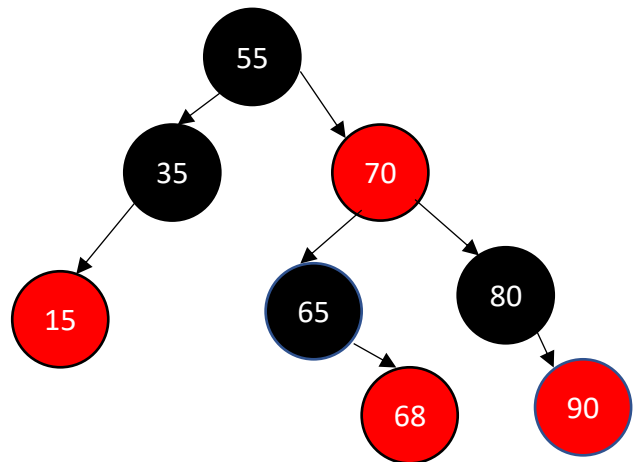


Fig 1.17 RED-BLACK deletion example

The tree is now balanced

We are left with below key to delete:

90

*delete(90)*

Perform BST deletion by finding successor of 90 in the tree. 90 is the just above leaf node (NIL) and it is a RED node. Remember that if a node to be deleted is red, we need to just simply delete it.

After all the deletions, our final RED-BLACK tree is:

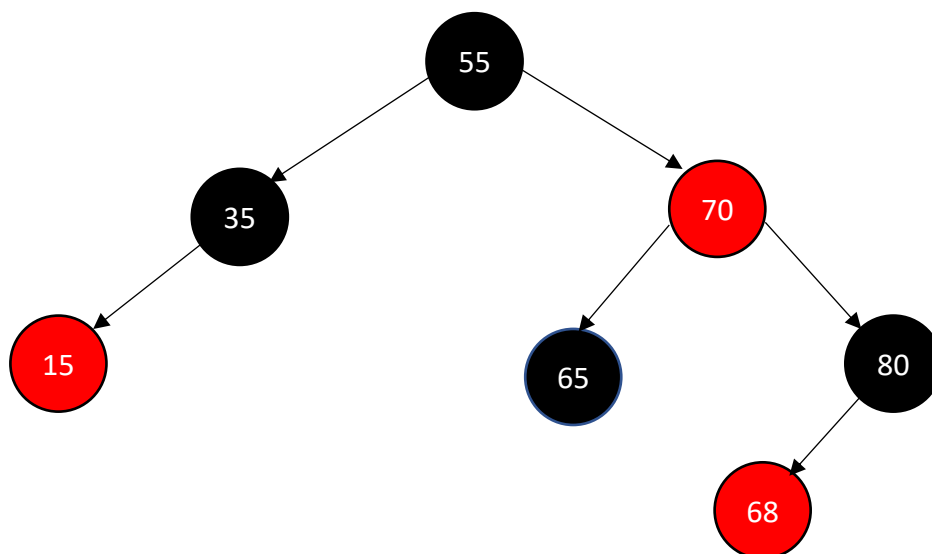


Fig 1.18 RED-BLACK deletion example

### Search Operation in Red-Black Tree:

Search operation in red-black tree is a fairly simple process. We follow the same approach as in BSTs following the value/key of each node. We start from root and traverse along the path for the desired key. If key's value is less than current node's value, we go left from that node and if it is greater than current node's value, we go right from that node. This same procedure is followed until we reach the node whose key equals the desired key. Hence, at each node we discard either its right-subtree or left-subtree. Also, since Red-Black trees are height balanced, each search can be performed in  $O(\lg n)$  time.

<b>Operations</b>	<b>Complexities</b>	
	<b>Average Case</b>	<b>Worst Case</b>
<i>Search</i>	$O(\lg n)$	$O(\lg n)$
<i>Insert</i>	$O(\lg n)$	$O(\lg n)$
<i>Delete</i>	$O(\lg n)$	$O(\lg n)$
<i>Space Complexity</i>	$O(n)$	$O(n)$

Table 1.1 Time and Space Complexity in RED-BLACK Tree

### Data transformations in Red-Black Tree:

- The user is provided with a text box and buttons like *Insert*, *Delete*, *Search*, for the fulfillment of these operations.
- The system then checks if the operation is feasible and if the parameters passed are consistent with what is needed.
- If the operation is feasible, it goes ahead and performs the operation by fetching the Red-Black Tree structure from the backend storage, else it pops up an error message.
- It then checks if the tree is balanced, if not it performs rotation and recoloring until it is balanced.
- After the desired operation is completed, the control then comes back to the user for any further operation they may want to perform.

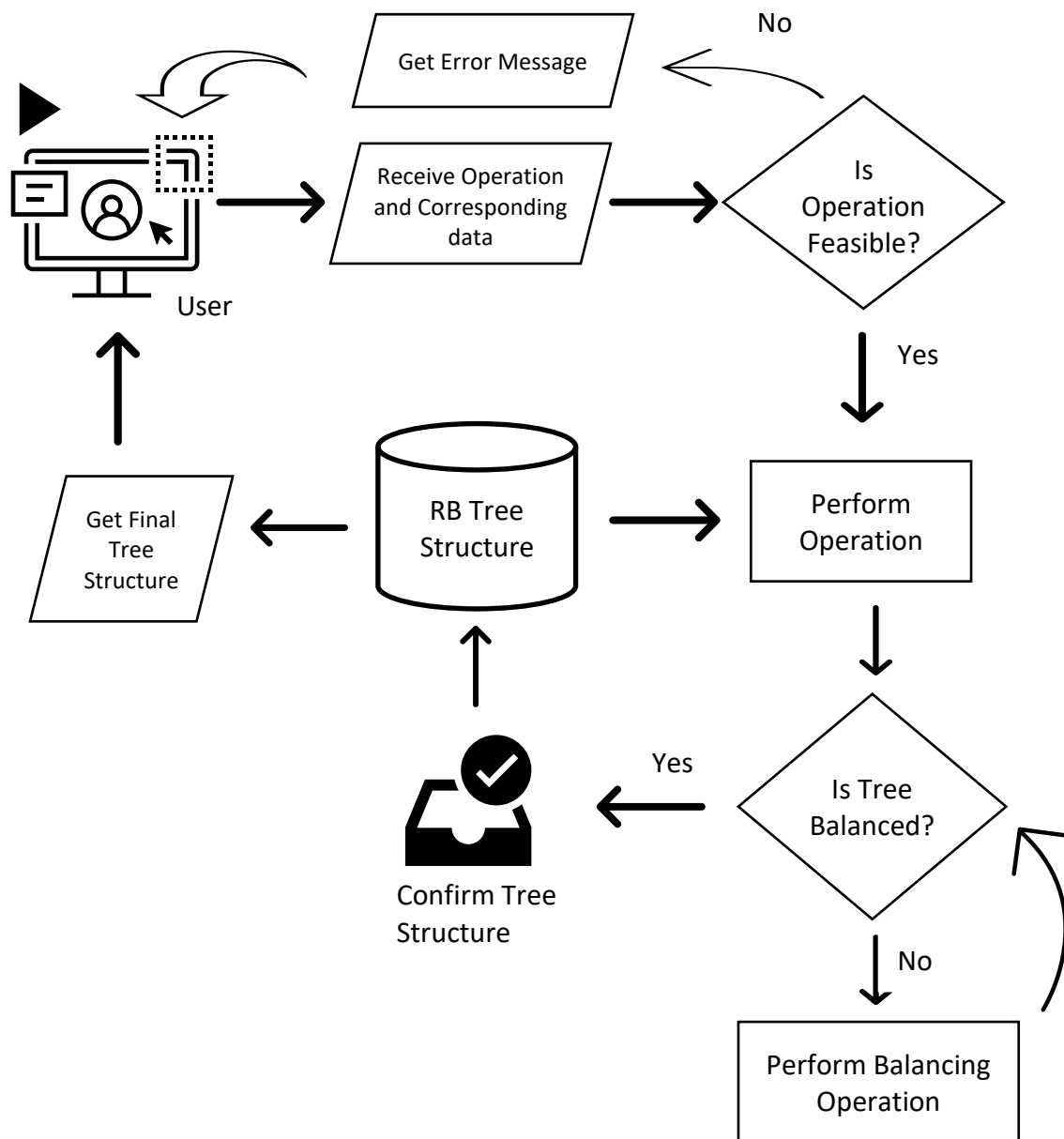


Fig 1.19 Design of Red-Black Tree Application

The above diagram is a representation of how we plan to implement the Red-Black Tree Implementation. It is a flow of how different subsequent task will be performed starting from user giving a command to getting control back from backend processing.

#### IV. IMPLEMENTATION

We have used Java 11 and JavaFX to implement the RB tree and visualize it respectively. The following are the methods and descriptions to handle the operations performed on Red Black Trees for all cases.

RBtrees.java		
CLASS	METHOD(s)	DESCRIPTION
RBTrees	1. public void start(Stage stage)  2. private void SearchInput(TextField value, String str)  3. private void WrongInput(TextField value, String str)  4. public static void main(String args[])	To visualize the entire application
RedBlackView	1. public final void setStatus(String msg)  2. public final void setTreeNodes(String v)  3. public void printRedBlackTree()  4. protected void printRedBlackTree(RedBlack.Node root, double x, double y, double hGap)	To visualize the Pane area to display the operations on RBTrees
RedBlack	1. public void insert(int value, RedBlackView view)  2. private void adjustTreeInsertion(Node node)  3. private void adjustTreeDeletion(Node db)  4. private void replace(Node del, Node leaf) 5. private Node successor(Node node)	Driver code to actually perform the operations

	6. public void delete(int value)  7. private void rotateLeft(Node node)  8. private void rotateRight(Node node)  9. public Node getRoot()  10. public boolean search(int val)	
--	---	--


## A) Insertion in RB trees

### CASE 1(a): Inserting a new Node

As a thumb rule, the inserted Node is always **RED**. The left and right child of this newly inserted leaf node are set to be NULL.

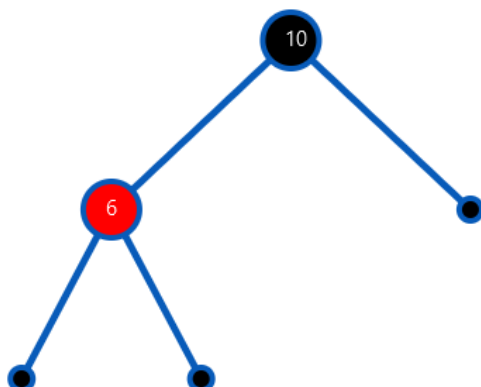
```
//initialise a new ndoe
Node node = new Node();
node.color = Color.RED; //new node must be RED
node.left = TNULL;
node.right = TNULL;
node.value = value;
node.nill = false;
node.parent = null;
```

Eg: insert(6)

 Red Black Tree

6 is inserted in the tree

No. of nodes: 2

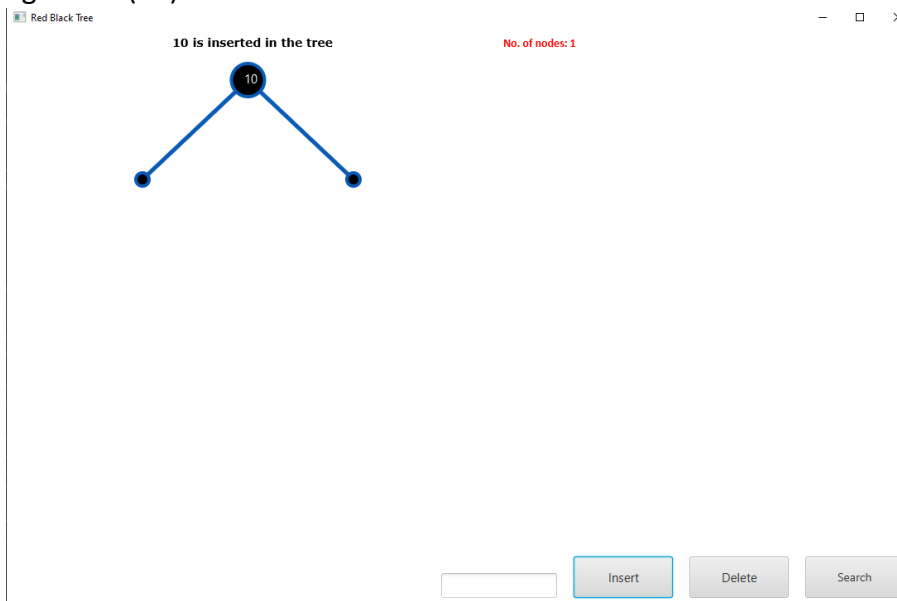


### CASE 1(b): If this newly inserted leaf node is a Root

As a thumb rule, the root Node is always BLACK. We then recolor the leaf node in Case 1 to Black and assign it to be a Root Node.

```
if(node.parent == null){
    node.color = Color.BLACK;
    root = node;
    return;
}
```

Eg: insert(10)



CASE 2: If parent of the leaf Node is black

If the newly inserted Node has a BLACK parent Node, we keep it as it is. No need to recolor the tree.

```
//Case 2) if parent node is BLACK, return
if(node.parent.color == Color.BLACK){
    return;
}
```

CASE 3: If parent of the leaf Node is RED

CASE 3(a): If parent is a left child of Grandparent

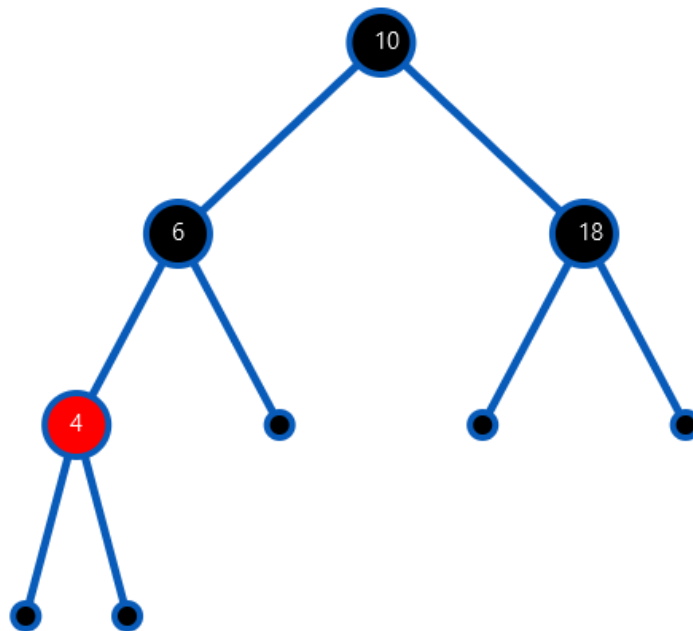
We then Go to check the color of the parent's sibling Node, aka 'Uncle Node'. If the **uncle** Node is RED, we then need to recolor both **parent** and **uncle** Node to BLACK, and recolor **grandparent** Node to RED

Eg: 6 was initially RED, now is BLACK.



4 is inserted in the tree

No. of nodes: 4



```

while(node != root && node.parent.color == Color.RED){
    //if parent is right child
    if(node.parent == node.parent.parent.right){
        p_sibling = node.parent.parent.left; //parent's sibling
        //Case 4.1) parent's sibling is red, change color of parent and parent's sibling
        if(p_sibling.color == Color.RED){
            p_sibling.color = Color.BLACK;
            node.parent.color = Color.BLACK;
            //make parent's parent as current node
            node = node.parent.parent;
            node.color = Color.RED;
        }
    }
}

```

However, if the uncle Node is BLACK (before recoloring),

- If the Current inserted Node is the **Left** child of the parent, We have to perform RL (right rotation) to balance the tree.

```

//if node is a left child, RL rotation
if(node == node.parent.left){
    node = node.parent;
    rotateRight(node);
}

```

- If the Current inserted Node is the **Right** child of the parent, We have to perform LR (Left rotation) to balance the tree.

```

//if node is right child, L rotation
node.parent.color = Color.BLACK;
node.parent.parent.color = Color.RED;
rotateLeft(node.parent.parent);
}

```

CASE 3(b): If parent is a right child of Grandparent

We then Go to check the color of the parent's sibling Node, aka 'Uncle Node'. If the **uncle** Node is **RED**, we then need to recolor both **parent** and **uncle** Node to BLACK, and recolor **grandparent** Node to **RED**

```

if(p_sibling.color == Color.RED){
    p_sibling.color = Color.BLACK;
    node.parent.color = Color.BLACK;
    //make parent's parent as current node
    node = node.parent.parent;
    node.color = Color.RED;
}

```

However, if the uncle Node is BLACK (before recoloring),

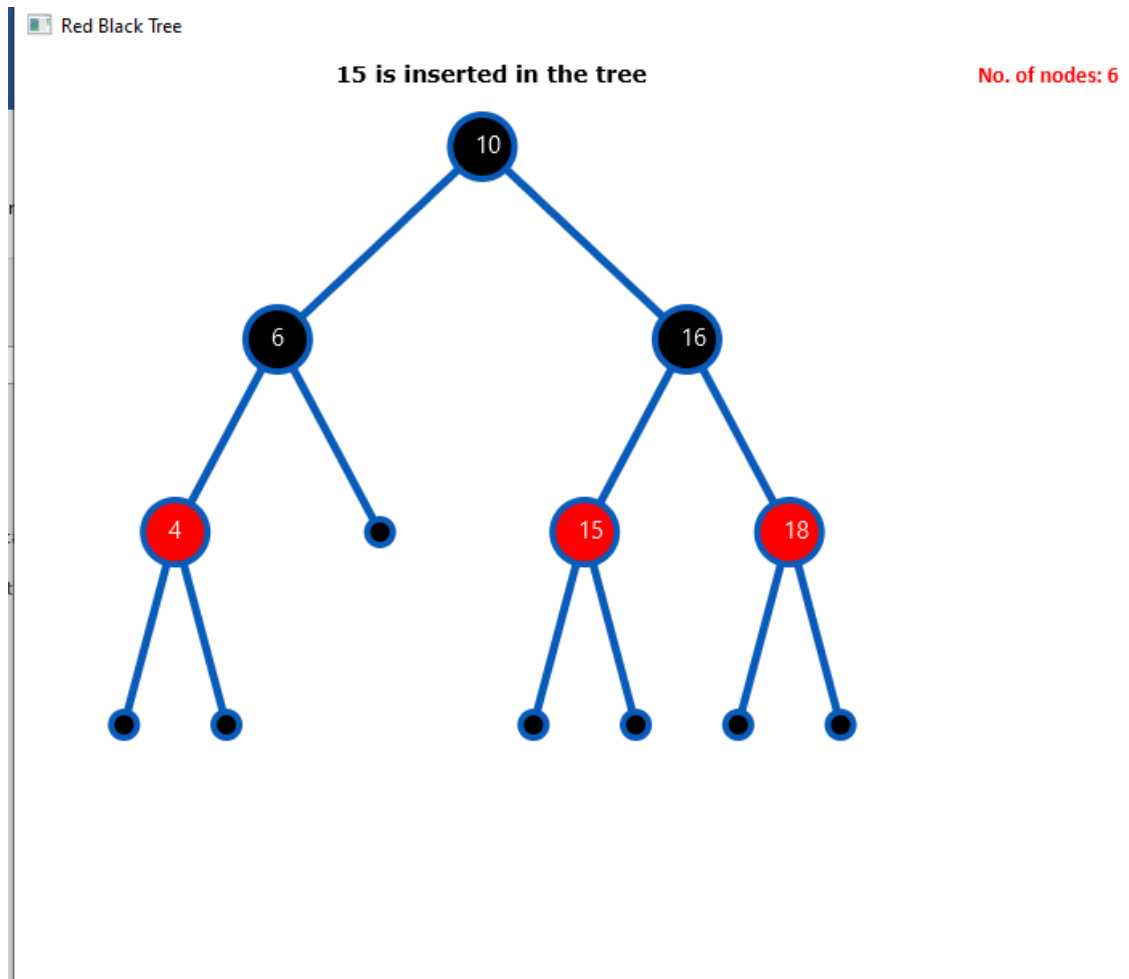
- If the Current inserted Node is the **Left** child of the parent, We have to perform RL (right rotation) to balance the tree.

```

//if node is left child, R rotation
node.parent.color = Color.BLACK;
node.parent.parent.color = Color.RED;
rotateRight(node.parent.parent);
}

```

Eg: right rotation happened when insert(15) took place.



- If the Current inserted Node is the **Right** child of the parent, We have to perform LR (Left rotation) to balance the tree.

```
//if node is right child, LR rotation
if(node == node.parent.right){
    node = node.parent;
    rotateLeft(node);
}
```

## B) Deletion in RB trees

We need to traverse to the Node that needs to be deleted. The User is expected to enter the value of the Node he/she wishes to delete. If this Node is found, it will be marked current and will be deleted. If not, An Error message will be displayed stating that the Node is absent. The tree traversal in RB tree is very similar to that of AVL/ BST tree, wherein we choose to navigate a particular branch(left , right) depending on the value of the Node to be searched. One important remark here, we don't delete internal nodes, replace node to be deleted with appropriate predecessor

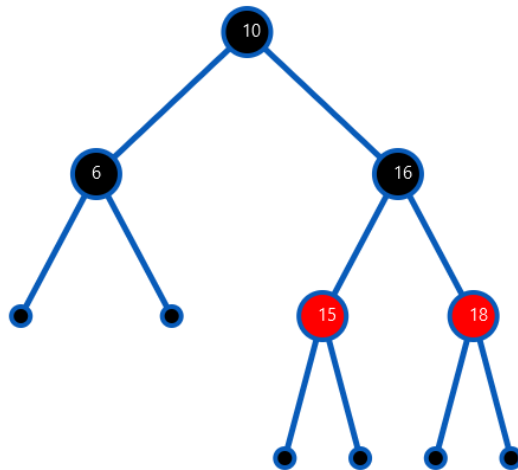
CASE 1: if left child of the Node to be deleted is NULL

Right child becomes the current Node and the new parent Node.

Eg: delete(4)

4 is deleted from the tree

No. of Nodes: 5



```
if(current.left == TNULL){
    db = current.right;
    replace(current, current.right);
}
```

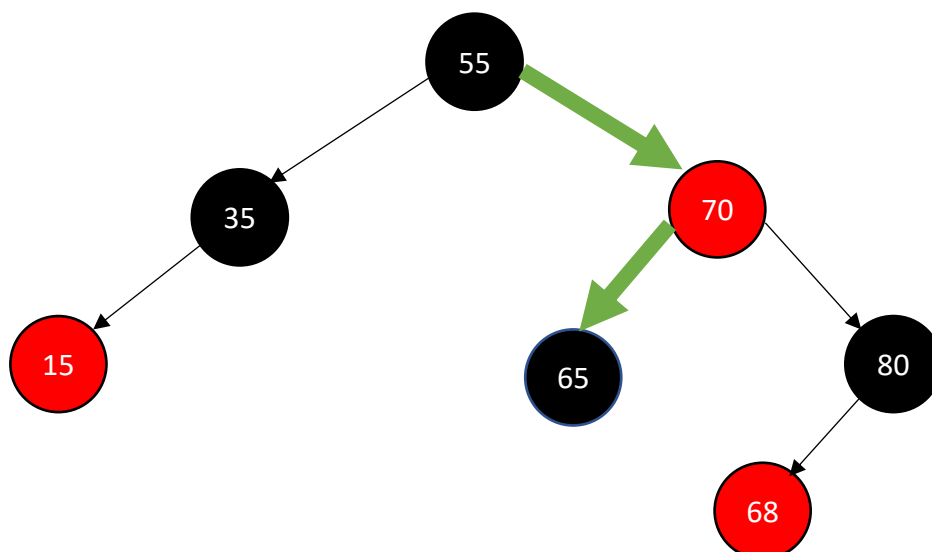
CASE 2: if right child of the Node to be deleted is NULL

Left child becomes the current Node and the new parent Node.


```
else if(current.right == TNULL){
    db = current.left;
    replace(current, current.left);
}
```

CASE 3: if Both children of the Node to be deleted are not NULL

We then traverse the left side of the right subtree, and replace the value of leftmost node with the deleted parent's node. For Eg:

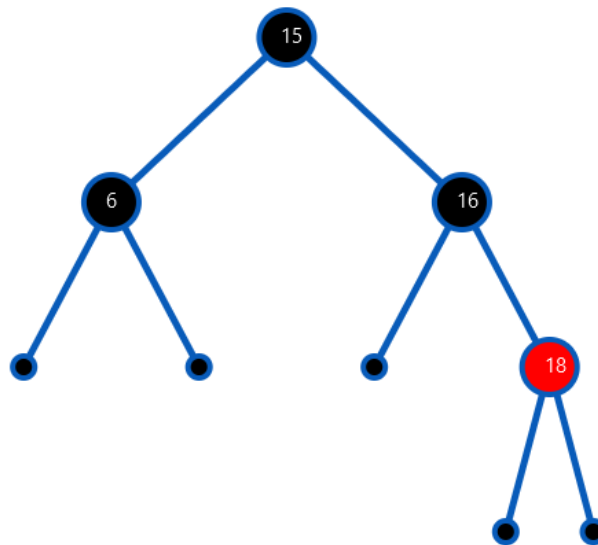


Eg: delete(10)

 Red Black Tree

10 is deleted from the tree

No. of Nodes: 4



If we delete(55), 65 being the leftmost element of Right subtree will take the place of the deleted 55 Node, followed by a left rotation on 80.

```
del = successor(current.right);
originalColor = del.color;
db = del.right;
if(del.parent == current){
    db.parent = del;
}
```

```
replace(current, del);
del.left = current.left;
del.left.parent = del;
del.color = current.color;
```

#### CASE 4: Double Black tree case

If double back case arises , we need to adjust the tree accordingly. Let db be the double back node.

##### CASE 4(a) If db is left child , check sibling of db

- If sibling is black and both it's children are black, the color of the sibling changes to **RED**

```

if(db == db.parent.left){
    db_sibling = db.parent.right;
    if(db_sibling.color == Color.BLACK){
        //Case 3.1) if db's sibling is black and both it's children are black
        if(db_sibling.left.color == Color.BLACK && db_sibling.right.color == Color.BLACK){
            db_sibling.color = Color.RED;
            db = db.parent; //if parent was black, now it becomes db, else if it was red- loop breaks
        }
    }
}

```

- If sibling is black and it's left child is **RED**, the color of the sibling becomes **RED**, Left child is assigned **BLACK**, and Right rotation takes places.
- Sibling inherits parent's color, db's parent's color becomes **BLACK**. Db\_sibling's right child becomes black and then a left rotation is performed on db's parent

```

db_sibling.color = db.parent.color;
db.parent.color = Color.BLACK; //sibling's color must have been black as one of it's child is red
db_sibling.right.color = Color.BLACK; //make red child as black
rotateLeft(db.parent);
break;

```

- If sibling is **RED** change the color db\_sibling to **BLACK**, and change the color of db's parent as **RED**. and then a left rotation is performed on db's parent

```

else{
    db.parent.color = Color.RED;
    db_sibling.color = Color.BLACK;
    rotateLeft(db.parent);
    db_sibling = db.parent.right;
}

```

- when db is removed, single black (original color) remains

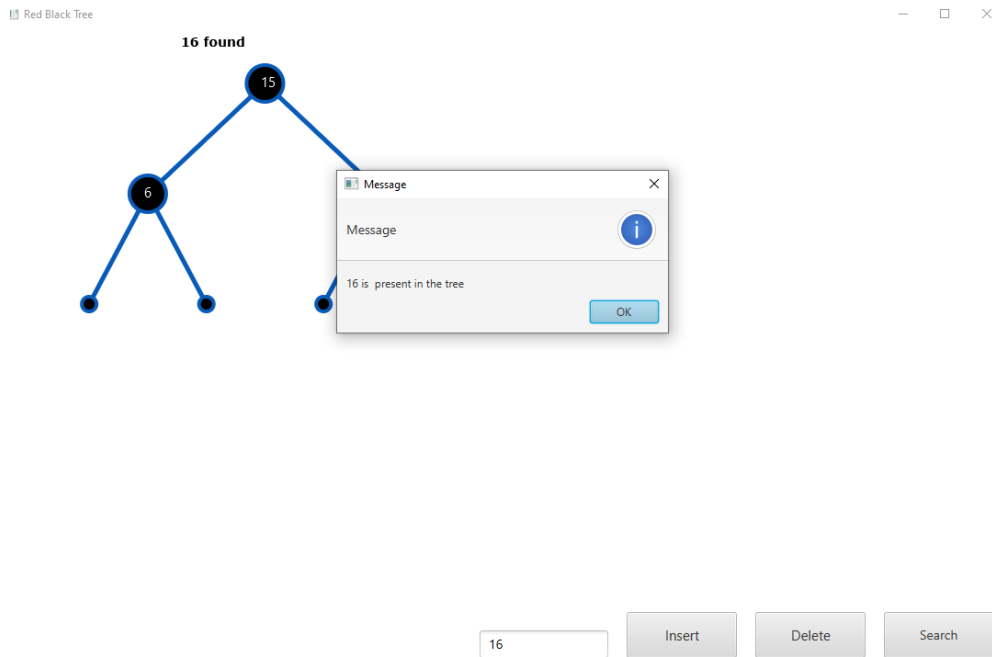
```

//when db is removed, single black (original color) remains
db.color = Color.BLACK;
}

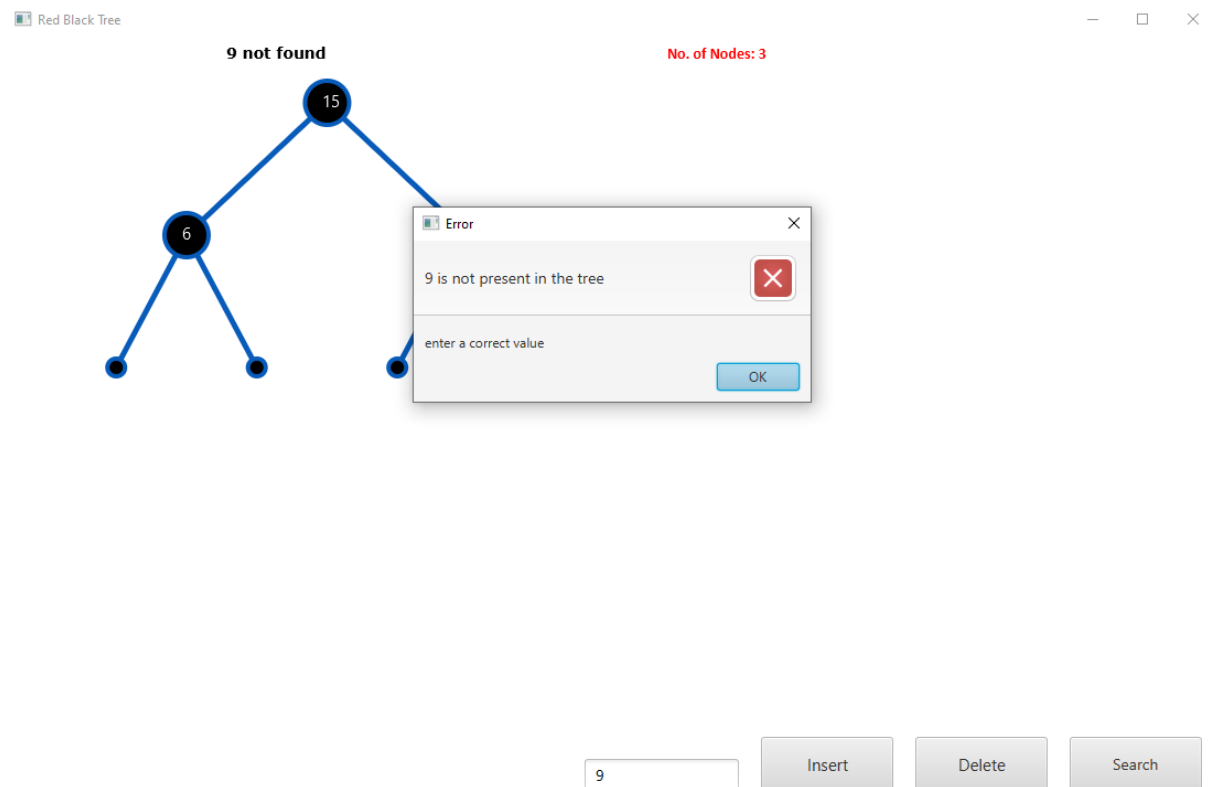
```

### C) Searching in RB trees

#### Case 1: Node present



## Case 2: Node absent



## V. APPLICATIONS

### Completely Fair Scheduler (CFS)

- CFS is a process scheduler used in linux kernel.

- It is used for CPU resource allocation, and it also increases utilization of CPU and interactive processes.
- It uses red-black tree for its processing.
- Nodes in red-black tree used are schedulable time entities and are sorted. This allows CFS to give share to tasks instead of giving more time to priority tasks.

### **Tree Map and Hash Map**

- Red-Black tree is used in Tree Map in JAVA SE 7 in backend.
- Since its insertion and deletion take  $O(\lg n)$  time, it is preferred over normal binary trees.
- Map in C++ and Hash Map in Java 8 have now replaced linked list to red-black tree due to same reason as mentioned above.
- STL in C++ also uses red-black tree.

### **K-Means Clustering**

- Red-Black tree is new modern approach along with min heap to calculate K-means clustering.
- Red-Black tree calculates initial mean and uses better heuristic approach to assign data to nearest means.
- Red-Black tree ensures that run time of K-Means clustering is less than previous iteration.
- Using Red-Black tree only those clusters are moved which are affected and unlike before where distance to all the clusters were calculated.
- Each node of red-black tree has object and min-heap contains the value of the object. This ensures overall algorithm is very efficient.

Linux also uses red-black trees in the mmap and munmap operations for file/memory mapping. Moreover, MySQL also uses the Red-Black Tree for indexes on tables.



## REFERENCES

- [1] Introduction to Algorithms, Third Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- [2] <https://www.youtube.com/watch?v=w5cvkTXy0vQ>
- [3] <https://www.youtube.com/watch?v=qA02XWRTBdw>
- [4] [https://en.wikipedia.org/wiki/Red-black\\_tree](https://en.wikipedia.org/wiki/Red-black_tree)
- [5] [http://www.btechsmartclass.com/data\\_structures/red-black-trees.html](http://www.btechsmartclass.com/data_structures/red-black-trees.html)
- [6] <https://medium.com/@khallilbailey/simple-red-black-trees-b54642bd7652>