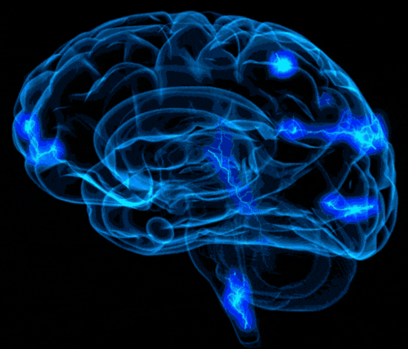


terv

practical learning redefined

One point solution for all your placement needs

terv.pro

Red black tree

Introduction

* When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree.
* However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n).
* This is where Red Black Trees come in, they are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced.
* This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Red Black Tree-

* Red-Black tree is a binary search tree in which every node is colored with either red or black.
* It is a type of self balancing binary search tree. It has a good efficient worst case running.
  Red Black Trees are self-balancing, meaning that the tree adjusts
  itself automatically after each   insertion or   deletion operation.
* It uses a simple but powerful mechanism to maintain balance,
  by coloring each node in the tree either red or .black.

Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. **Root property:** The root is black.
2. **External property:** Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. **Internal property:** The children of a red node are black. Hence possible parent of red node is a black node.
4. **Depth property:** All the leaves have the same black depth.
5. **Path property:** Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above-mentioned properties is that the Red-Black tree is roughly balanced.

Rules That Every Red-Black Tree Follows:

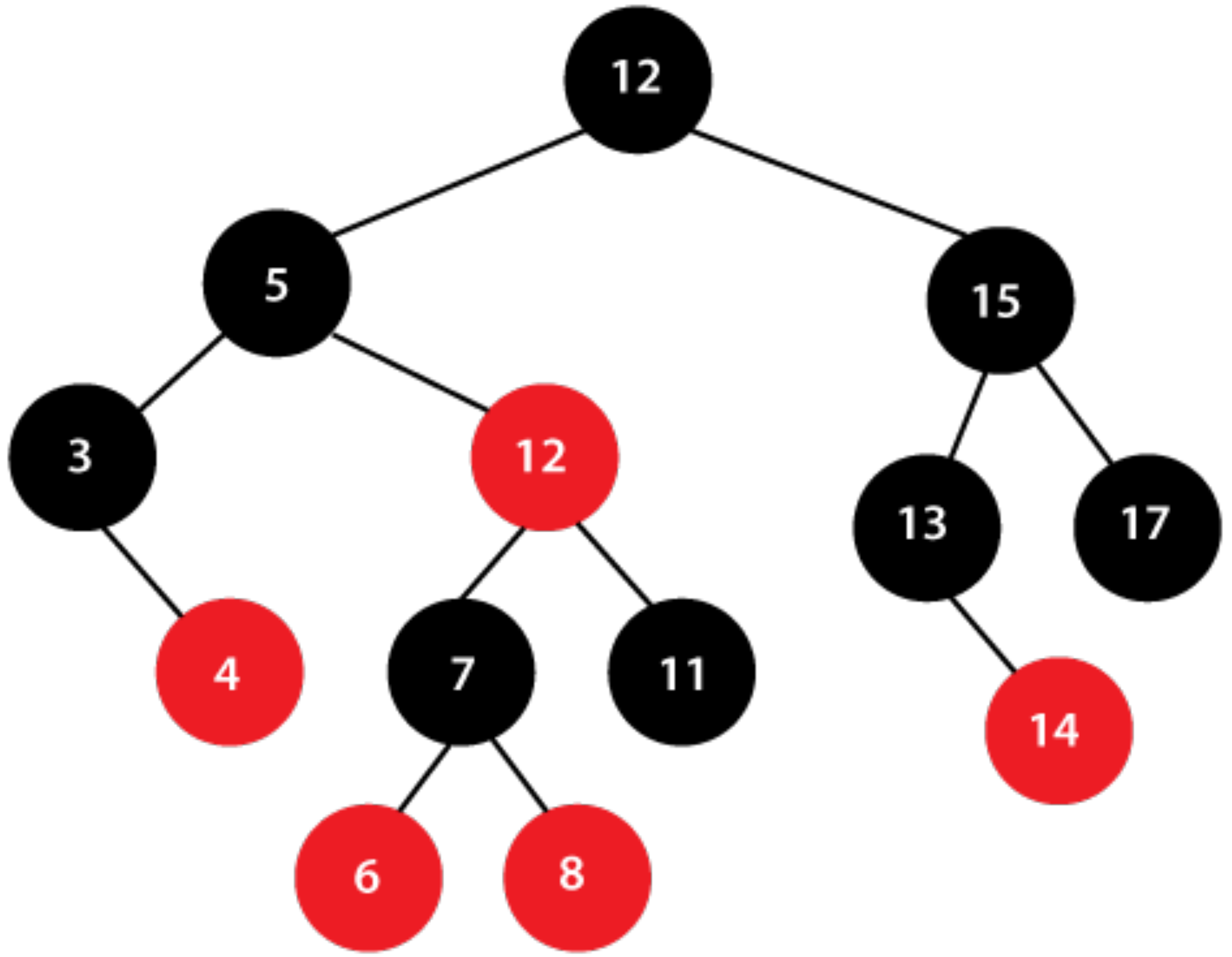
1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. Every leaf (e.i. NULL node) must be colored BLACK.

Why Red-Black Trees?

* Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
* The cost of these operations may become O(n) for a skewed Binary tree.
* If we make sure that the height of the tree remains O(log n) after every insertion and deletion
* Then we can guarantee an upper bound of O(log n) for all these operations.
* The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

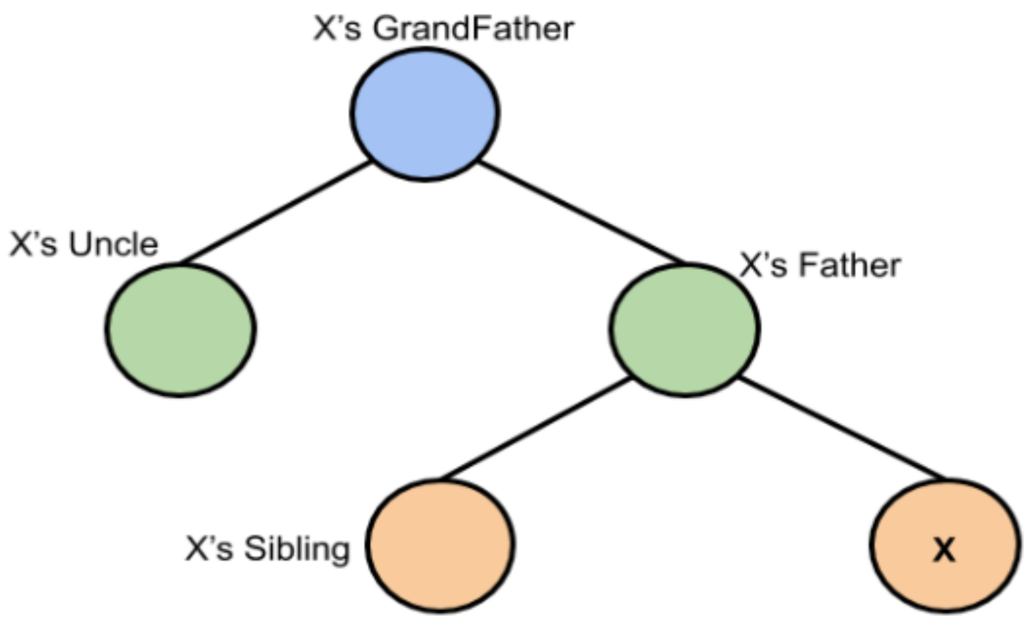
S.NO	Algorithm	Time Complexity	Efficiency
1	Search	O(lögn)	EASY
2	Insertion	O(lögn)	EASY
3	Deletion	O(lögn)	EASY

“n” is the total number of elements in the red-black tree.



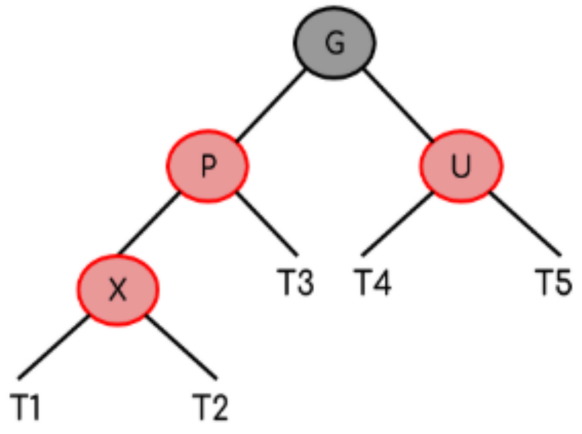
Red Black Tree

Insertion Red black tree



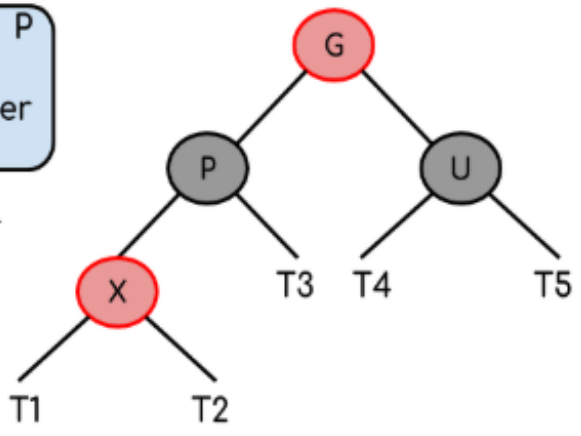
Logic:
First, you have to insert the node similarly to that in a binary tree and assign a red colour to it. Now, if the node is a root node then change its colour to black, but if it is not then check the colour of the parent node. If its colour is black then don't change the colour but if it is not i.e. it is red then check the colour of the node's uncle. If the node's uncle has a red colour then change the colour of the node's parent and uncle to black and that of grandfather to red colour and repeat the same process for him (i.e. grandfather).If grandfather is root then don't change grandfather to red colour.

Uncle is Red



Current Tree Structure

1. Change the colour of X's parent P and uncle U to black.
2. Change the colour of its Grandfather G to red.

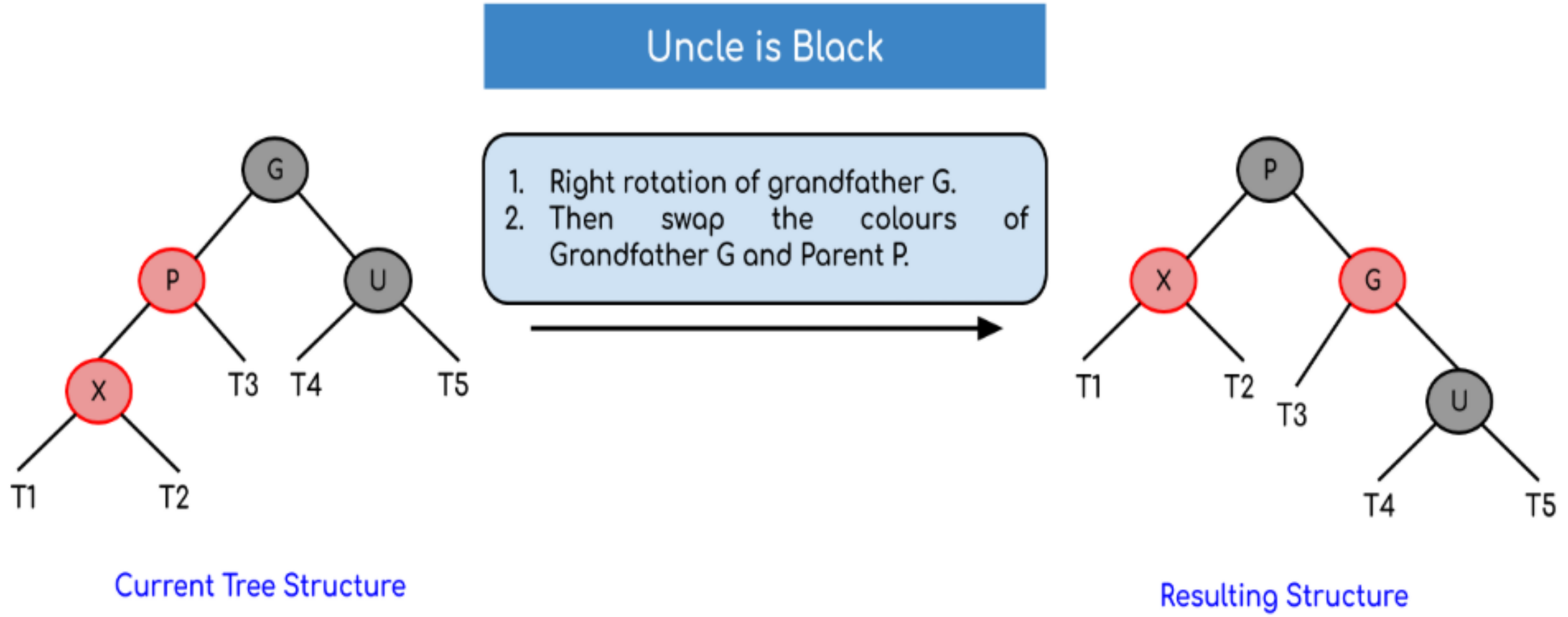


Resulting Structure

1. Repeat the all recolouring steps for Grandfather considering it as X.

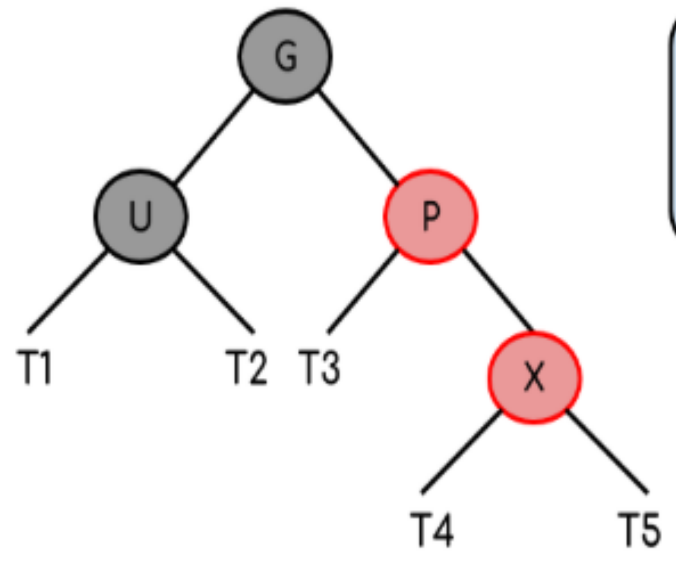
But, if the node’s uncle has black colour then there are 4 possible cases:

- Left Left Case (LL rotation):

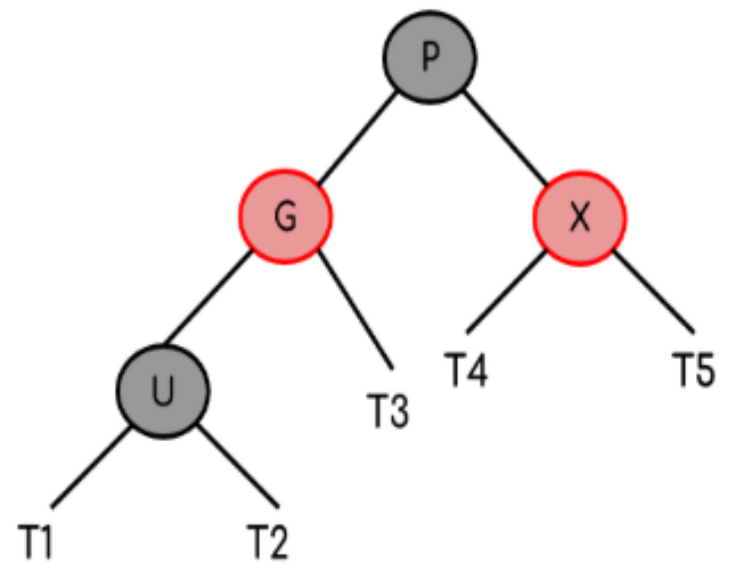


1. Right rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.

- Right Right Case (RR rotation):



1. Left rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.



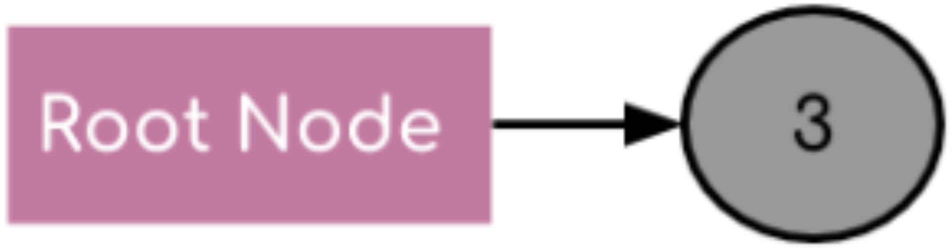
Current Tree Structure

Resulting Structure

Example: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.

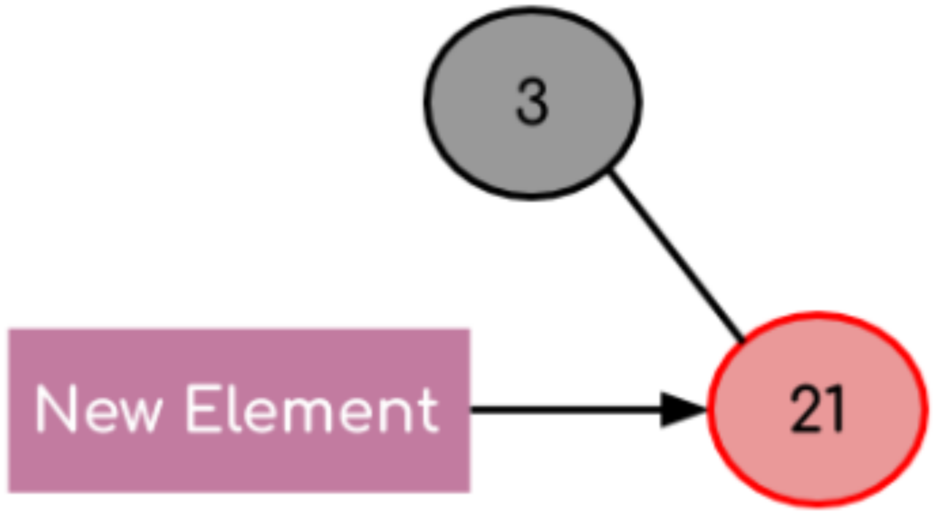
Solution:

Step 1: Inserting element 3 inside the tree.



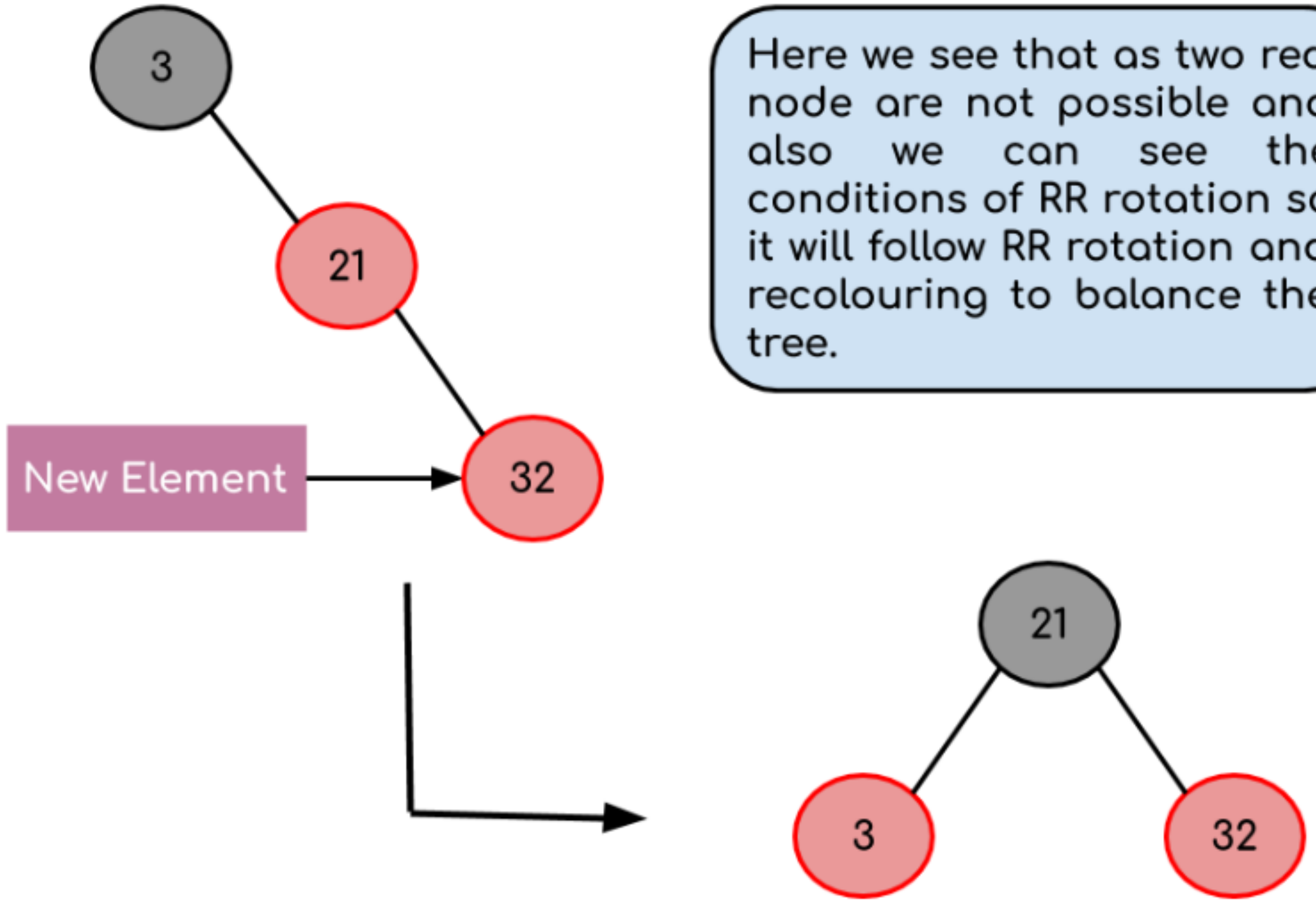
When the first element is inserted it is inserted as a root node and as root node has black colour so it acquires the colour black.

Step 2: Inserting element 21 inside the tree.



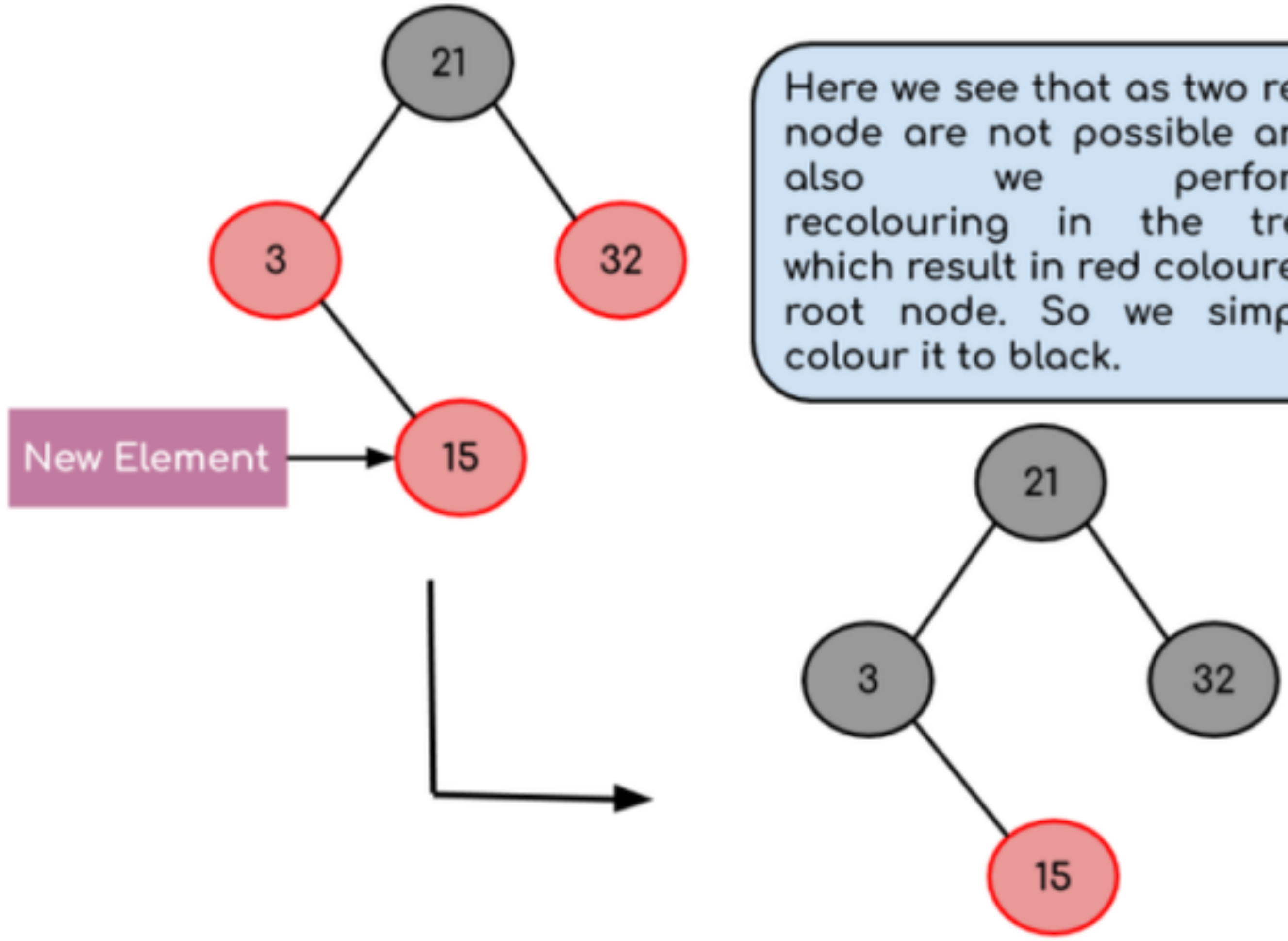
The new element is always inserted with a red colour and as 21 > 3 so it becomes the part of the right subtree of the root node.

Step 3: Inserting element 32 inside the tree.



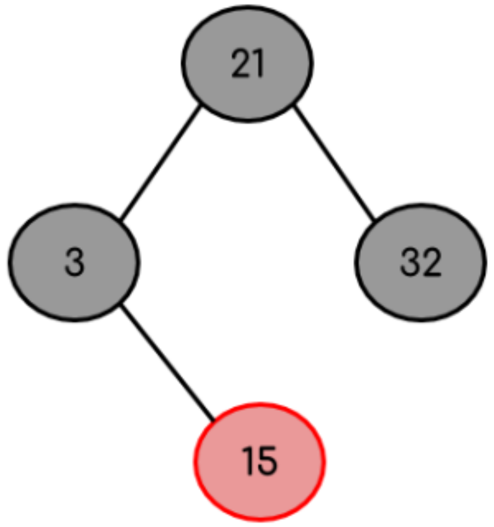
Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.

Step 4: Inserting element 15 inside the tree.



Here we see that as two red node are not possible and also we perform recolouring in the tree which result in red coloured root node. So we simply colour it to black.

Final Tree Structure:



Code for Insertion in Java.

Here is the code written in **java** for **implementation of RED-BLACK Trees**
The following code also implements tree insertion as well as tree traversal.
at the end you can visualize the constructed tree too!!!.

```
import java.io.*;

// considering that you know what are red-black
trees here is the implementation in java for
insertion and traversal.
// RedBlackTree class. This class contains
subclass for public class RedBlackTree
{
    public Node root;//root node
    public RedBlackTree()
    {
        super();
        root = null;
    }
    // node creating subclass
    class Node
    {
        int data;
        Node left;
        Node right;
        char colour;
        Node parent;
```

```
        Node(int data)
            {
                super();
                this.data = data;   // only including data. not
                this.left = null; // left subtree
                this.right = null; // right subtree
                this.colour = 'R'; // colour . either 'R' or 'B'
                this.parent = null; // required at time of.
            }
        }
```

```
Node rotateLeft(Node node)
    {
        Node x = node.right;
        Node y = x.left;
        x.left = node;
        node.right = y;
        node.parent = x; // parent resetting is also important.
        if(y!=null)
            y.parent = node;
        return(x);
    }
    //this function performs right rotation
    Node rotateRight(Node node)
    {
        Node x = node.left;
        Node y = x.right;
        x.right = node;
        node.left = y;
        node.parent = x;
        if(y!=null)
            y.parent = node;
        return(x);}
```

```
boolean ll = false;
    boolean rr = false;
    boolean lr = false;
    boolean rl = false;
    // helper function for insertion. Actually this function performs all tasks in single pass only.
    Node insertHelp(Node root, int data)
    {
        // f is true when RED RED conflict is there.
        boolean f=false;

        //recursive calls to insert at proper position according to BST properties.
        if(root==null)
            return(new Node(data));
        else if(data<root.data)
        {
            root.left = insertHelp(root.left, data);
            root.left.parent = root;
            if(root!=this.root)
            {
                if(root.colour=='R' && root.left.colour=='R')
                    f = true;
            }
        }
```

```
else
        {
            root.right = insertHelp(root.right,data);
            root.right.parent = root;
            if(root!=this.root)
            {
                if(root.colour=='R' && root.right.colour=='R')
                    f = true;
            }
        // at the same time of insertion, we are also assigning parent nodes
        // also we are checking for RED RED conflicts
        }

        // now lets rotate.
        if(this.ll) // for left rotate.
        {
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';
            this.ll = false;
        }
```

```
else if(this.rr) // for right rotate
        {
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.rr  = false;
        }
        else if(this.rl)  // for right and then left
        {
            root.right = rotateRight(root.right);
            root.right.parent = root;
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';

            this.rl = false;
        }
        else if(this.lr)  // for left and then right.
        {
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.lr = false;
        }
```

```
if(f)
        {
            if(root.parent.right == root)  // to check which child is the current node of its parent
            {
                if(root.parent.left==null || root.parent.left.colour=='B')  // case when parent's sibling
is black
                {// perform certaing rotation and recolouring. This will be done while backtracking. Hence
setting up respective flags.
                    if(root.left!=null && root.left.colour=='R')
                        this.rl = true;
                    else if(root.right!=null && root.right.colour=='R')
                        this.ll = true;
                }
                else // case when parent's sibling is red
                {
                    root.parent.left.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
                }
            }
else
            {
                if(root.parent.right==null || root.parent.right.colour=='B')
                {
                    if(root.left!=null && root.left.colour=='R')
                        this.rr = true;
                    else if(root.right!=null && root.right.colour=='R')
                        this.lr = true;
                }
                else
                {
                    root.parent.right.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
                }
            }
            f = false;
        }
        return(root);
    }
```

```
public void insert(int data)
    {
        if(this.root==null)
        {
            this.root = new Node(data);
            this.root.colour = 'B';
        }
        else
            this.root = insertHelp(this.root,data);
    }
    // helper function to print inorder traversal
    void inorderTraversalHelper(Node node)
    {
        if(node!=null)
        {
            inorderTraversalHelper(node.left);
            System.out.printf("%d ", node.data);
            inorderTraversalHelper(node.right);
        }
    }
```

```
public void inorderTraversal()
    {
        inorderTraversalHelper(this.root);
    }
    // helper function to print the tree.
    void printTreeHelper(Node root, int space)
    {
        int i;
        if(root != null)
        {
            space = space + 10;
            printTreeHelper(root.right, space);
            System.out.printf("\n");
            for ( i = 10; i < space; i++)
            {
                System.out.printf("  ");
            }
            System.out.printf("%d", root.data);
            System.out.printf("\n");
            printTreeHelper(root.left, space);
        }
    }
```

```
public void printTree()
    {
        printTreeHelper(this.root, 0);
    }
    public static void main(String[] args)
    {
        // let us try to insert some data into tree and try to visualize the tree as well as 
traverse.
        RedBlackTree t = new RedBlackTree();
        int[] arr = {1,4,6,3,5,7,8,2,9};
        for(int i=0;i<9;i++)
        {
            t.insert(arr[i]);
            System.out.println();
            t.inorderTraversal();
        }
        // you can check colour of any node by with its attribute node.colour
        t.printTree();
    }
}
```

OUTPUT

```
1
1 4
1 4 6
1 3 4 6
1 3 4 5 6
1 3 4 5 6 7
1 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

```
 9

                    8

                              7

          6

                    5

4

                    3

          2

                    1
```

Time Complexity : O(log N) , here N is the total number of nodes in the red-black trees.

Space Complexity: O(N), here N is the total number of nodes in the red-black trees.

terv

THANK YOU