Red Black Trees

A **Red-Black** Tree is a self-balancing binary search tree in which each node has an extra bit, which represents its color (red or black).

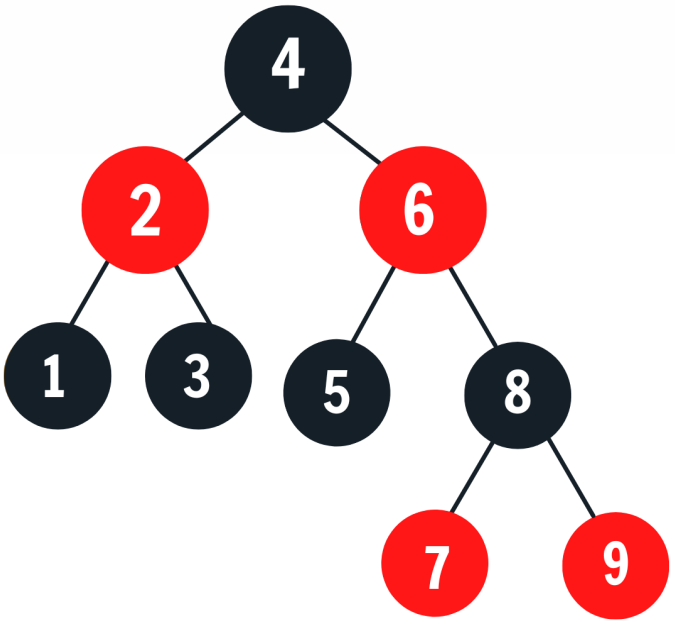
(Every Red Black Tree is a binary search tree but all the Binary Search Trees need not to be Red Black trees.)

There are few properties associated with Red-Black Trees:

- The root is black. (Root Property)
- Every external node is black. (External Property)
- The children of a red node are black. (Red Property)
- All external nodes have the same black depth.(Depth Property)
- Every New node must be inserted with Red color.
- Every leaf (i.e. NULL node) must be colored Black.

Example

The following is a Red Black Tree which is created by inserting numbers from 1 to 9 and every node is satisfying all the properties of the Red Black Tree.



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

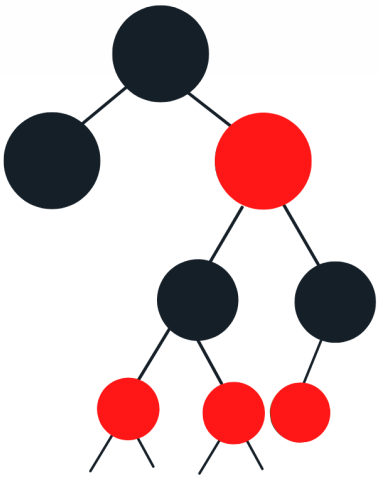
Height of a Red-Black Tree

Proving the height of the red-black tree is especially important because the height of the red-black tree is what allows us to calculate it asymptotic complexity and performance. This is one method of doing so.

First imagine a red-black tree with height. Now, we merge all red nodes into their black parents. A given black node can either have:

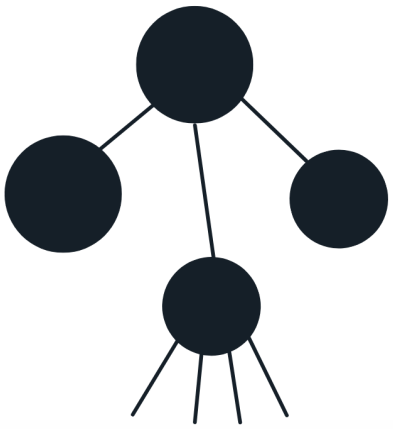
1. 2 black children, in which case the black parent still has 2 children.
2. 1 black child and 1 red child, in which case the black parent now has 3 children.
3. 2 red children, in which case the black parent now has 4 children.

Here is a graphical example of that merging process (assume any stray arrow points to a black node).



We merge all the red nodes into their parents nodes.

If rc = number of red children a red node has then each black node will now have 2*rc pointers coming out of it.



As you can see, every black node has either 2, 3, or 4 children.

This new tree has a height, h1. Because any given path in the original red-black tree had at most half its nodes red, we know that this new height is at least half the original height. So,

H1 >= H/2

The number of leaves in a tree is exactly equal to n+1, so

n+1 > 2^(H1)

log(N+1) >= H1 >= H/2

H < 2log(n+1)

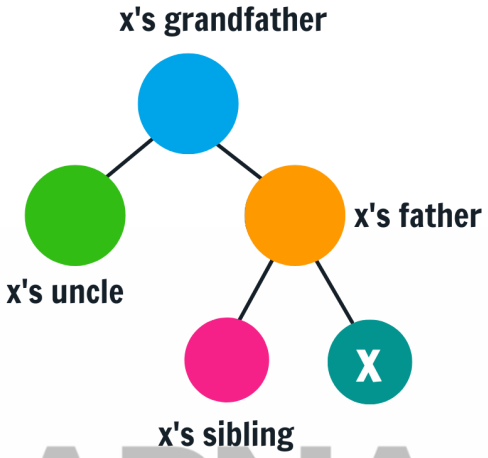
Insertion in Red Black Trees

In the Red-Black tree, we use two steps to do the balancing:

1. Recoloring
2. Rotation

Recolouring is the change in color of the node i.e. if it is red then change it to black and vice versa. It must be noted that the color of the NULL node is always black. Moreover, we always try recolouring first, if recolouring doesn't work, then we go for rotation. Following is a detailed algorithm. The

algorithms have mainly two cases depending upon the color of the uncle. If the uncle is red, we do recolour. If the uncle is black, we do rotations and/or recolouring.



Process:

Let x be the newly inserted node.

Perform standard BST insertion and make the color of newly inserted nodes as RED.

If x is the root, change the color of x as BLACK (Black height of complete tree increases by 1).

Do the following if the color of x's parent is not BLACK and x is not the root.

a) If x's uncle is RED (Grandparent must have been black from property 4)

(i) Change the color of parent and uncle as BLACK.

(ii) Color of a grandparent as RED.

(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to AVL Tree)

(i) Left Left Case (p is left child of g and x is left child of p)

(ii) Left Right Case (p is left child of g and x is the right child of p)
(iii) Right Right Case (Mirror of case i)
(iv) Right Left Case (Mirror of case ii)
Recoloring after rotations:
For Left Left Case [3.b (i)] and Right Right case [3.b (iii)], swap colors of grandparent and parent after rotations
For Left Right Case [3.b (ii)]and Right Left Case [3.b (iv)], swap colors of grandparent and inserted node after rotations

Code

```
import java.io.*;

public class RedBlackTree {
   public Node root;//root node
   public RedBlackTree() {
       super();
       root = null;
   }

   class Node {
       int data;
       Node left;
       Node right;
       char colour;
       Node parent;

       Node(int data) {
           super();
           this.data = data; // only including data. not key
           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 rechecking.
       }
   }
   // this function performs left rotation
```

```
    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);
    }


    // these are some flags.
    // Respective rotations are performed during traceback.
    // rotations are done if flags are true.
    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) {
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';
            this.ll = false;
        }
        else if(this.rr) {
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.rr = false;
        }
        else if(this.rl) {
            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) {
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.lr = false;
        }
        // when rotation and recolouring is done flags are reset.
        // Now lets take care of RED RED conflict
        if(f) {
            // to check which child is the current node of its parent
            if(root.parent.right == root) {
                // case when parent's sibling is black
                if(root.parent.left==null || root.parent.left.colour=='B') {//
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;
                }
                // case when parent's sibling is red
                else {
                    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);
}

// function to insert data into tree.
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);
    }
}
//function to print inorder traversal
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);
        }
    }
    // function to print the tree.
    public void printTree() {
        printTreeHelper(this.root, 0);
    }
    public static void main(String[] args) {
        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();
        }

        t.printTree();
    }
}
```