Analysis of Mathematical Properties & Development of Parallel Algorithms for Optimized Red-Black and AVL Trees in High-Performance Computing

Bhavyanshika Gupta[1], Pankaj Kumar Gupta[2], P. K. Tyagi[3], R. K. Agrawal[4]
[1]*Engineering Scholar at Jaypee Institute of Information Technology Sector-62 Noida.*
[2]*Assistant Professor & Head, BCA Department, DPBS College, Anupshahr Distt. BulandShahr (UP) India*
[3]*Professor & Head, Department of Statistics, DPBS College, Anupshahr Distt. BulandShahr (UP) India*
[4]*Professor & Head, Department of Mathematics, DPBS College, Anupshahr Distt. BulandShahr (UP) India*

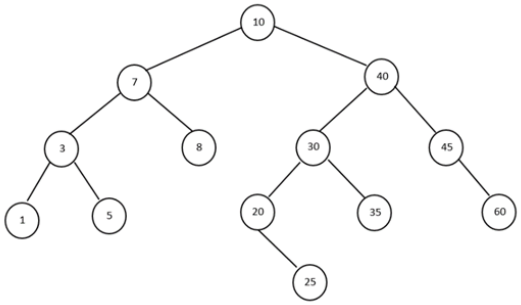
Abstract: In computer science, self-balancing binary search trees are essential data structures that have the ability of effective data retrieval as well as efficient data storage. It has a self balancing mechanism which makes them stand out among most of the other trees. With an emphasis on insertion, deletion, rebalancing, height attributes, and their effects on performance, this study provides a thorough comparison of these two tree architectures. The paper explores the mathematical characteristics of tree height and rebalancing methods, contrasting the color-based strategy of Red-Black Trees with the height-based stringent balancing of AVL Trees. Particular focus is given on the contrast between the self-balancing mechanisms of both the trees where AVL is height balanced tree and Red Black is a color-balanced tree. In dynamic data settings, experimental results show the trade-offs between the frequency of rebalancing procedures and the ensuing structural stability. This study sheds the light on the best use cases for each type of tree and give helpful suggestions for putting them into practice in real-world applications that need scalable and effective data management.

Keywords:Red Black, AVL, Self Balancing BST.

INTRODUCTION:

AVL Tree:

An AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree in computer science. The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information"[7]. It is the first self-balancing binary search tree data structure to be invented.



In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The root will be the only node at level 1.

[22]Each subsequent level will be as full as possible i.e. 2 nodes at level 2,4 nodes at level 3 and so on, i.e., in general there will be 2^{1-1} at level 1. Therefore number of nodes from level 1 through level h-1 will be:

$$1+2+2^2+2^3+\ldots\ldots\ldots\ldots+2^{h-2} = 2^{h-1}-1$$

[22]In a binary search tree, the *balance factor* of a node X is defined to be the height difference of

$$BF(X) = Height(RightSubtree(X))-Height(LeftSubtree(X))$$

its two child sub-trees rooted by node X.

A node X with BF(X)<0 is called "left-heavy", one with BF(X)>0 is called "right-heavy", and one

with BF(X)=0 is sometimes simply called "balanced".

Properties

Balance factors are kept updated by knowing the previous balance factors and the change in height – it is not necessary to know the absolute height. For holding the balance of AVL tree, information about two bits per node are sufficient.

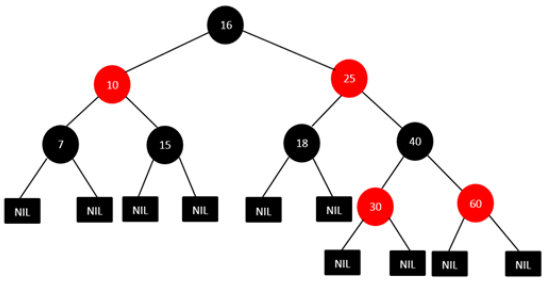
The height *h* (counted as the maximal number of levels) of an AVL tree with *n* nodes lies in the interval:

$$\mathrm{Log}_2(n+1) <= h < \log_\vartheta n + 2 + b$$

where $\vartheta = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio and $b = \frac{\log_2 5}{2 \log_2 \vartheta} - 2 \approx -0.3277$. This is because an AVL tree of height *h* contains at least $F_{h+2} - 1$ nodes where F_n is the Fibonacci series with initial values as $F_1 = F_2 = 1$[7].

Red Black Tree:

Red Black Trees are a type of balanced binary search tree that uses rules to maintain balance, ensuring lowest time complexity for operations like insertion, deletion, searching, and re-balancing regardless of the initial shape of the tree. Red Black Trees are self-balancing, that uses a simple color-coding scheme to adjust the tree after modification[9]. It has one extra bit of storage pre node: its color, which is either red or black. By constraining the way nodes can be colored on any path from root to a leaf, red black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced[11].



When the tree is modified, the new tree is rearranged and repainted to restore the properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that there properties can be performed efficiently.

The balancing is not perfect, but guarantees searching in *O(log n)* time, where *n* is the number of entries in the tree. The insert and delete operations, along with tree rearrangement and recoloring, also execute in *O(log n)* time[7].

Properties

[9]In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

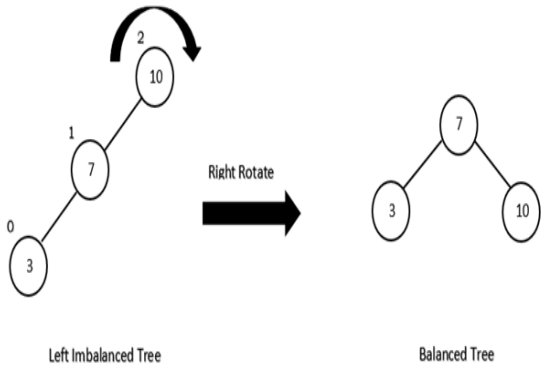
1. Every node is either red or black.
2. All NIL nodes are considered black.
3. A red node does not have a red child.
4. Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes.
5. (Conclusion) If a node N has exactly one child, the child must be red, because if it were black, its NIL descendants would sit at a different black depth than N's NIL child, violating requirement 4.

A red black tree with n internal nodes has height at most 2log(n+1)[11].

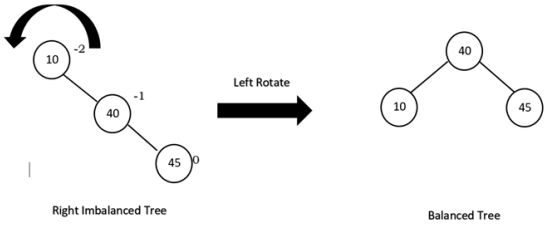
Rotations:

Rotation is a local operation in a search tree that preserves the binary search tree property[11]. There are 4 types of rotations which are as follows:

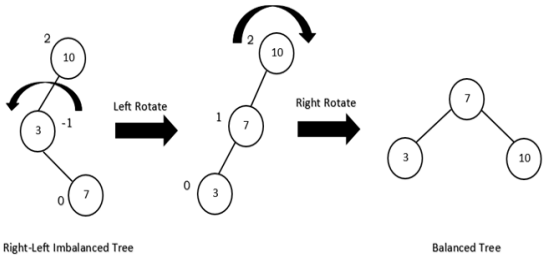
Right Rotation:



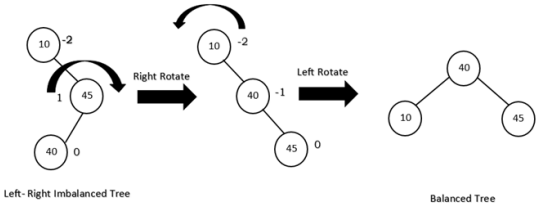
Left Rotation:



Left-Right Rotation :



Right- Left Rotation :

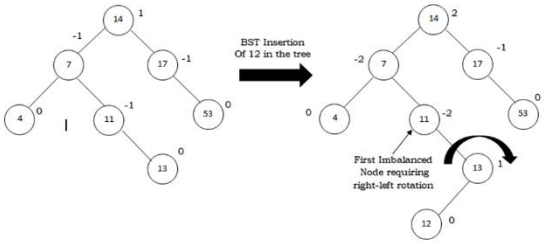


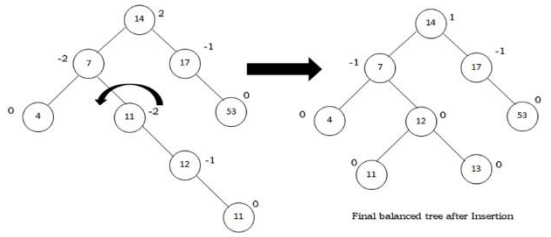
Detailed Analysis on AVL Tree :

Insertion in AVL Tree :
Let the newly inserted node be n

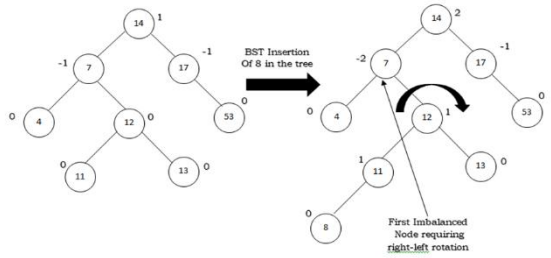
1. Perform standard BST insert for n.
2. Starting from n, travel up and find the first unbalanced node. Let t be the first unbalanced node, u be the child of t that comes on the path from n to t and v be the grandchild of t that comes on the path from n to t.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with t. There can be 4 possible cases that need to be handled as v, u and t can be arranged in 4 ways.
4. Following are the possible 4 arrangements:
    - u is the left child of t and v is the left child of u (Left Left Insertion)
    - u is the left child of t and v is the right child of u (Left Right Insertion)
    - u is the right child of t and v is the right child of u (Right Right Insertion)
    - u is the right child of t and v is the left child of u (Right Left Insertion)

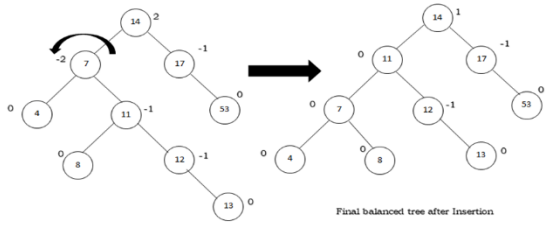
Below is an Example that demonstrates all cases of insertion :
Example 1 : Insertion of 12 in the tree given below





Example 2 : Insertion of 8 in the tree given below

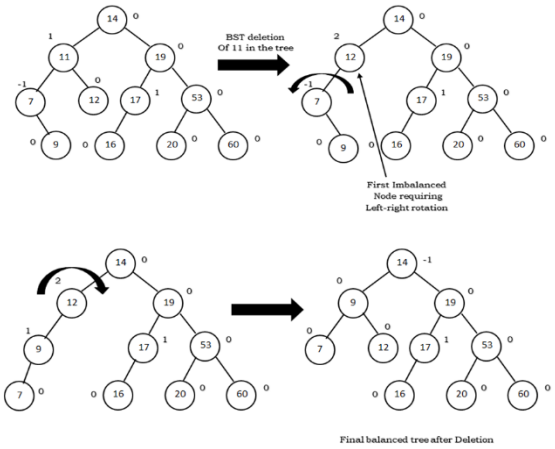




Deletion in AVL Tree :
Let n be the node to be deleted

1. Perform standard BST delete for n.
2. Starting from n, travel up and find the first unbalanced node. Let u be the first unbalanced node, v be the larger height child of u, and t be the larger height child of v. Note that the definitions of t and v are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with u. There can be 4 possible cases that needs to be handled as t, v and u can be arranged in 4 ways. Following are the possible 4 arrangements:
    - v is left child of u and t is left child of v (Left Left Case)
    - v is left child of u and t is right child of v (Left Right Case)
    - v is right child of u and t is right child of v (Right Right Case)
    - v is right child of u and t is left child of v (Right Left Case)

Example: Deletion of 11 from the tree given below



Code :

```
#include<bits/stdc++.h>
using namespace std;
struct avlTree
{
   int data;
   avlTree *left;
   avlTree *right;
   int height;
};
int getHeight(avlTree *root)
{
   if(root==NULL)
      return 0;
   return root->height;
}
int balance(avlTree *root)
{
   if(root==NULL)
      return 0;
   return      (getHeight(root->left)-getHeight(root-
>right));
}
avlTree* rightRotate(avlTree *y)
{
   avlTree *x=y->left;
   avlTree *t2=x->right;
   x->right=y;
   y->left=t2;
   x->height=max(getHeight(x->left),getHeight(x-
>right))+1;
   y->height=max(getHeight(y->left),getHeight(y-
>right))+1;
   return x;
}
avlTree *leftRotate(avlTree *x)
{
   avlTree *y=x->right;
   avlTree *t2=y->left;
   y->left=x;
   x->right=t2;
   x->height=max(getHeight(x->left),getHeight(x-
>right))+1;
   y->height=max(getHeight(y->left),getHeight(y-
>right))+1;
   return y;
}
avlTree* maximum(avlTree *root)
{
   if(root==NULL)
      return NULL;
   if(root->right==NULL)
      return root;
   else
      return maximum(root->right);
}
avlTree* insert(avlTree *root, int key)
{
   if(root==NULL)
   {
      root=new avlTree;
      root->data=key;
      root->left=NULL;
      root->right=NULL;
      root->height=1;
      return root;
   }
   else if(root->data>key)
      root->left=insert(root->left,key);
   else
      root->right=insert(root->right,key);
   root->height=max(getHeight(root-
>left),getHeight(root->right))+1;
   int bf=balance(root);

   if(bf>1 && root->left->data>key)
      root=rightRotate(root);
   else if(bf<-1 && root->right->data<key)
      root=leftRotate(root);
   else if(bf>1 && root->left->data<key)
   {
      root->left=leftRotate(root->left);
      root=rightRotate(root);
   }
   else if(bf<-1 && root->right->data>key)
   {
      root->right=rightRotate(root->right);
      root=leftRotate(root);
```

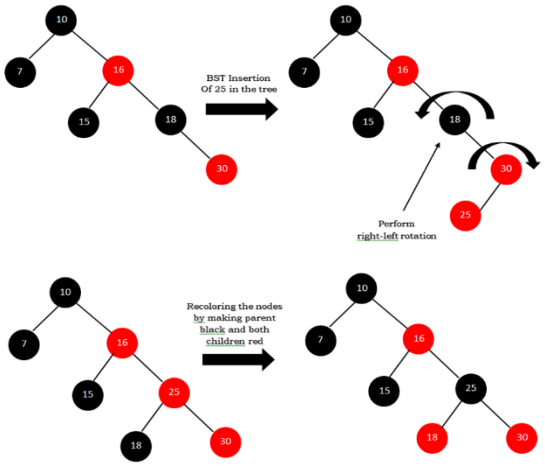
```
    }
    return root;
}
void preorder(avlTree *root)
{
    if(root==NULL)
        return;

    preorder(root->left);
    cout<<root->data<<" ";
    preorder(root->right);
}
avlTree* deletion(avlTree *root, int val)
{
    if(root==NULL)
        return NULL;
    if(root->data>val)
    root->left=deletion(root->left,val);
    else if(root->data<val)
    root->right=deletion(root->right,val);
    else
    {
        if(root->left==NULL && root->right==NULL)
            return NULL;
        else if(root->left==NULL && root->right!=NULL)
        {
            avlTree *temp=root->right;
            delete root;
            return temp;
        }
        else if(root->left!=NULL && root->right==NULL)
        {
            avlTree *temp=root->left;
            delete root;
            return temp;
        }
        else
        {
            avlTree *temp=maximum(root->left);
            root->data=temp->data;
            root->left=deletion(root->left,root->data);
        }
    }
    root->height=max(getHeight(root->left),getHeight(root->right))+1;
    int bf=balance(root);
    if(bf>1 && balance(root->left)>=0)
        root=rightRotate(root);
    else if(bf<-1 && balance(root->right)<0)
        root=leftRotate(root);
    else if(bf>1 && balance(root->left)<0)
    {
        root->left=leftRotate(root->left);
        root=rightRotate(root);
    }
    else if(bf<-1 && balance(root->right)>=0)
    {
        root->right=rightRotate(root->right);
        root=leftRotate(root);
    }
    return root;
}
int main()
{
    avlTree *root=NULL;
     root = insert(root, 10);
    root = insert(root, 15);
    root = insert(root, 20);
    root = insert(root, 25);
    root = insert(root, 35);
    root = insert(root, 40);
    root = insert(root, 45);
    root = insert(root, 50);
     root = insert(root, 55);
     root = insert(root, 60);
     root = insert(root, 65);
     root = insert(root, 70);
     root = insert(root, 75);
    preorder(root);
    cout<<endl;
    root=deletion(root,60);
    preorder(root);
    cout<<endl;
}
```

Detailed Analysis on Red Black Tree:

Insertion in Red Black Tree:

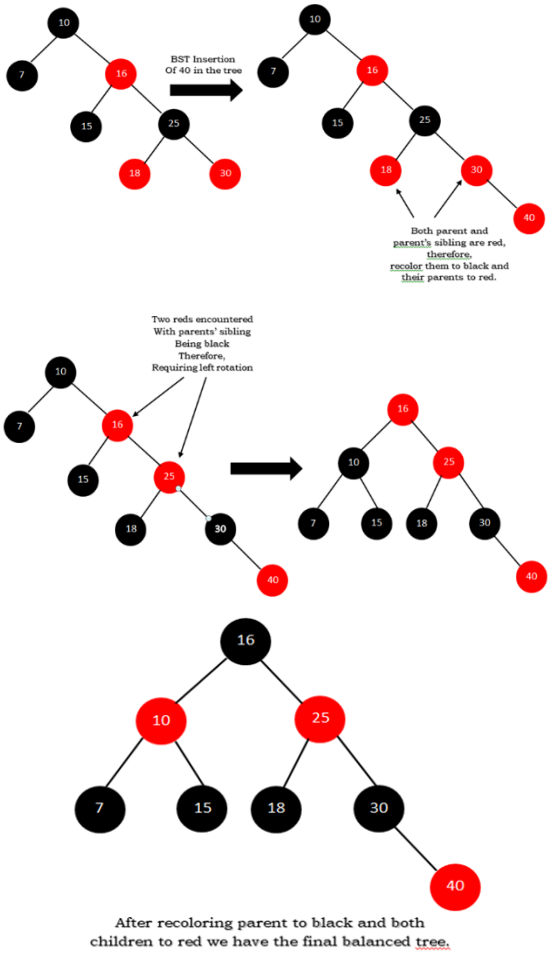
1. If the tree is empty, create a new node with black color.
2. If tree is not empty, create a new node of red color.
3. Insert the new node just as in BST.
4. If the parent of new node is black, then exit.
5. If the parent of new node is red, then check parent's sibling
   - If it's black or null then do rotations and then recoloring.

Example 1: Insertion of 25 in the tree given below



- If it's red then recolor both parent and its sibling and also check parent's parent is a root or not. If it's root then leave it as black otherwise recolor and recheck for parent's grandparents and so on.

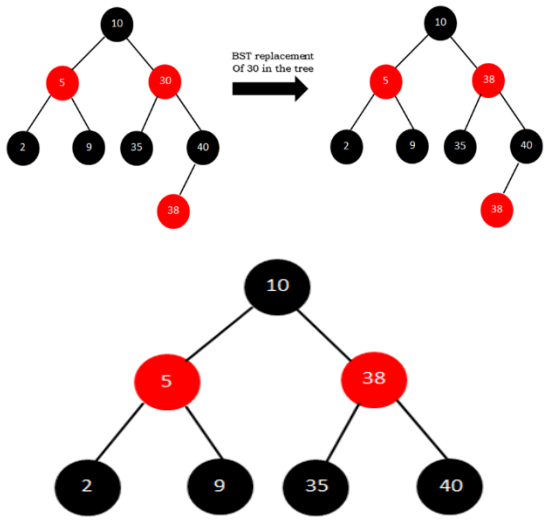
Example 2 : Insertion of 40 in the tree given below



Deletion in Red Black Tree:

1. Perform the BST deletion by replacing the given node with its inorder predecessor or inorder successor and then delete the leaf node by using the cases given below.
2. If the node to be deleted is red, just remove it.

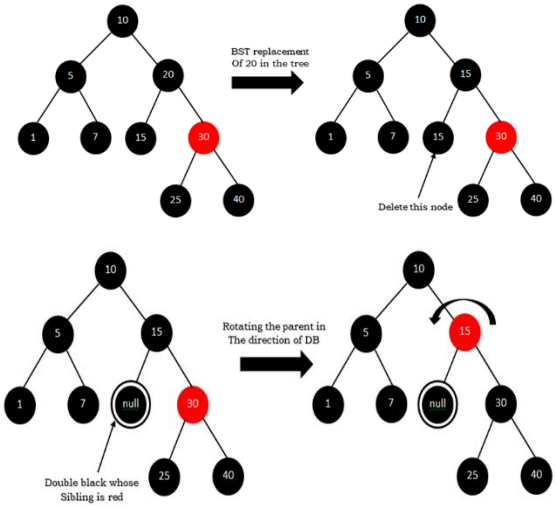
Example 1 : Deletion of 30 in the tree given below

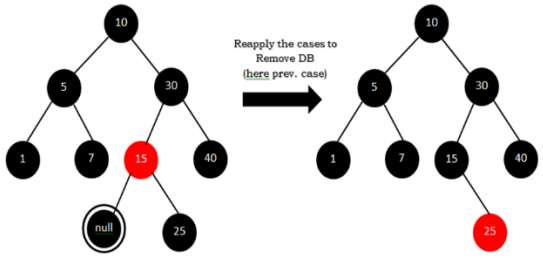


3. If the node to be deleted is black deleting it will give rise to double black condition, to remove it, check the for the following cases:

- If the root is double black (DB), just make it black.
- If DB's sibling is black and both of the sibling's children are black or sibling's children are NULL, then remove DB, add black to its parent and make sibling red. If the parent was already black, now it becomes double black, therefore now repeat the steps for parent. Example of this case can be seen in further cases.
- If DB's sibling is red, swap the colors of DB's parent and sibling, rotate parent in the direction of DB and reapply all the cases of DB.

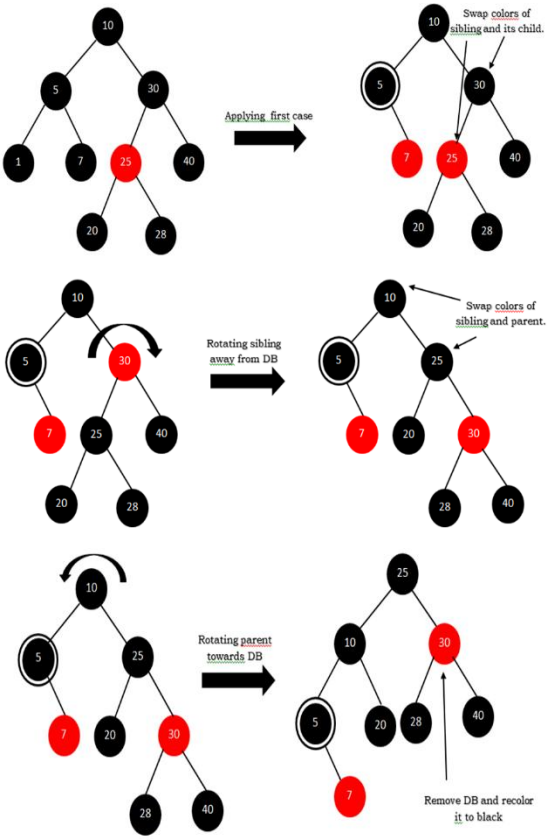
Example 2 : Deletion of 20 in the tree given below

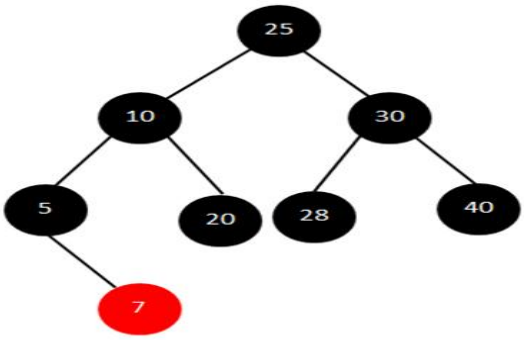




- If DB's sibling is black but the child who is near to DB is red, swap the color of DB's sibling and sibling's child, rotate the sibling in the direction opposite to DB and then move to next case.
- If the DB's sibling is black but the far away child is red, then swap the color of parent and sibling, rotate the parent in DB's direction, remove DB and change color of far red child to black.

Example 3 : Deletion of 1 in the tree given below





Final balanced tree

Code :

```
#include<iostream>
#include<queue>
using namespace std;
struct node
{
   int val;
   char color;
   node *left;
   node *right;
   node *parent;
};
class RBTree
{
   public:
   node *root;

   RBTree()
   {
      root=NULL;
   }
   void disp()
   {
      display(root);
   }
   void display(node *root)
   {
      if(root==NULL)
         return;
      display(root->left);
      cout<<root->val<<"("<<root->color<<") ,";
      display(root->right);
   }
   void leftRotate(node *x)
   {
      node *y=x->right;
      if(y->left!=NULL)
      {
         x->right=y->left;
         y->left->parent=x;
      }
```

```
    else
        x->right=NULL;
    if(x->parent!=NULL)
        y->parent=x->parent;
    if(x->parent==NULL)
    {
        root=y;
        y->parent=NULL;
    }
    else
    {
        if(x->parent->left==x)
            x->parent->left=y;
        else
            x->parent->right=y;
    }
    y->left=x;
    x->parent=y;
}
void rightRotate(node *y)
{
    node *x=y->left;
    if(x->right!=NULL)
    {
        y->left=x->right;
        x->right->parent=y;
    }
    else
        y->left=NULL;
    if(y->parent!=NULL)
        x->parent=y->parent;
    if(y->parent==NULL)
    {
        root=x;
        x->parent=NULL;
    }
    else
    {
        if(y->parent->left==y)
            y->parent->left=x;
        else
            y->parent->right=x;
    }
   x->right=y;
    y->parent=x;
}
void insertion(int val)
{
    node *n_node=new node;
    n_node->val=val;
    n_node->left=n_node->right=NULL;
    n_node->color='r';
    n_node->parent=NULL;
    node *temp=root;
    node *par=NULL;
    while(temp!=NULL)
    {
        par=temp;
        if(temp->val>val)
            temp=temp->left;
        else
            temp=temp->right;
    }
    if(par==NULL)
    {
        n_node->color='b';
        root=n_node;
        return;
    }
    if(par->val>val)
    {
        par->left=n_node;
        n_node->parent=par;
    }
    else
    {
        par->right=n_node;
        n_node->parent=par;
    }
    balance(n_node);
  }
  void balance(node *t)
  {
    if(t==root)
    {
        t->color='b';
        root=t;
        return;
    }
    while(t->parent!=NULL      &&      t->parent-
>color=='r')
    {
        node *g=t->parent->parent;
        if(g->left==t->parent)
        {
            node *u=g->left;
            if(g->right!=NULL      &&      g->right-
>color=='r')
            {
                u->color='b';
                g->right->color='b';
                g->color='r';
                t=g;
            }
```

```
            else
            {
               if(t->parent->right==t)
               {
                  t=t->parent;
                  leftRotate(t);
               }
               t->color='r';
               t->parent->color='b';
               g->color='r';
               rightRotate(g);
            }
         }
         else
         {
            node *u=g->right;
            if(g->left!=NULL && g->left->color=='r')
            {
               u->color='b';
               g->left->color='b';
               g->color='r';
               t=g;
            }
            else
            {
               if(t->parent->left==t)
               {
                  t=t->parent;
                  rightRotate(t);
               }
               cout<<"value of t"<<t->val<<endl;
               t->color='r';
               t->parent->color='b';
               g->color='r';
               leftRotate(g);
               t=t->parent;
            }
         }
         root->color='b';
      }
      if(t->parent==NULL)
      root=t;
      root->color='b';
   }
   node *serch(int n)
   {
      if(root==NULL)
         return NULL;
      node *temp=root;
      while(temp!=NULL)
      {
         if(temp->val==n)
            return temp;
         else if(temp->val<n)
            temp=temp->right;
         else
            temp=temp->left;
      }
      return NULL;
   }
   void deleteByVal(int n)
   {
      node *cur=serch(n);
      if(cur==NULL)
      {
         cout<<"Element Not found in Tree\n";
         return;
      }
      deleteNode(cur);
   }
   node* getSibling(node *cur)
   {
      if(cur->parent==NULL)
         return NULL;
      if(cur==cur->parent->left && cur->parent->right!=NULL)
         return cur->parent->right;
      else if(cur==cur->parent->right && cur->parent->left!=NULL)
         return cur->parent->left;
      return NULL;
   }
   node* successor(node *cur)
   {
      if(cur==NULL)
         return NULL;
      if(cur->left==NULL)
         return cur;
      return successor(cur->left);
   }
   node* predecessor(node *cur)
   {
      if(cur==NULL)
         return NULL;
      if(cur->right==NULL)
         return cur;
      return predecessor(cur->right);
   }
   bool hasRedChild(node* n)
   {
      return ((n->left!=NULL && n->left->color=='r') || (n->right!=NULL && n->right->color=='r'));
```

```
}
void fixDoubleBlack(node *cur)
{
   if(cur==root)
      return;
   node *sibling=getSibling(cur);
   node *parent=cur->parent;
   if(sibling==NULL)
      fixDoubleBlack(parent);
   else
   {
      if(sibling->color=='b')
      {
         if(!hasRedChild(sibling))
         {
            sibling->color='r';
            if(parent->color=='r')
               parent->color='b';
            else
               fixDoubleBlack(parent);
         }
         else if(sibling->left!=NULL && sibling->left->color=='r')
         {
            if(cur==parent->left)
            {
               char temp=sibling->color;
               sibling->color=sibling->left->color;
               sibling->left->color=temp;
               rightRotate(sibling);
               sibling=getSibling(cur);
            }
            char temp=parent->color;
            parent->color=sibling->color;
            sibling->color=temp;
            if(cur==parent->left)
            {
               leftRotate(parent);
               sibling->right->color='b';
            }
            else
            {
               rightRotate(parent);
               sibling->left->color='b';
            }
         }
         else
         {
            if(cur==parent->right)
            {
               char temp=sibling->color;
               sibling->color=sibling->right->color;
               sibling->right->color=temp;
               leftRotate(sibling);
               sibling=getSibling(cur);
            }
            char temp=parent->color;
            parent->color=sibling->color;
            sibling->color=temp;
            if(cur==parent->right)
            {
               rightRotate(parent);
               sibling->left->color='b';
            }
            else
            {
               leftRotate(parent);
               sibling->right->color='b';
            }
         }
      }
      else
      {
         parent->color='r';
         sibling->color='b';
         if(parent->right==cur)
            rightRotate(parent);
         else
            leftRotate(parent);
         fixDoubleBlack(cur);
      }
   }
}
void deleteNode(node *cur)
{
   if(cur->left==NULL && cur->right==NULL)
   {
      node *parent=cur->parent;
      if(parent==NULL)
      {
         root=NULL;
         return;
      }
      if(cur->color=='r')
      {
         if(parent->left==cur)
            parent->left=NULL;
         else
            parent->right=NULL;
         delete cur;
         return;
      }
```

```
        fixDoubleBlack(cur);
        if(parent->left==cur)
          parent->left=NULL;
        else
          parent->right=NULL;
        delete cur;
      }
      else if(cur->left!=NULL && cur->right!=NULL)
      {
        node *suc=successor(cur->right);
        int temp=cur->val;
        cur->val=suc->val;
        suc->val=temp;
        deleteNode(suc);
      }
      else if(cur->left!=NULL)
      {
        node *pre=predecessor(cur->left);
        int temp=cur->val;
        cur->val=pre->val;
        pre->val=temp;
        deleteNode(pre);
      }
      else
      {
        node *suc=successor(cur->right);
        int temp=cur->val;
        cur->val=suc->val;
        suc->val=temp;
        deleteNode(suc);
      }
    }
};
void levelorder(node *root)
{
  queue<node*>q;
  q.push(root);
  while(!q.empty())
  {
    node *temp=q.front();
    q.pop();
    cout<<temp->val<<" ";
    if(temp->left)
      q.push(temp->left);
    if(temp->right)
      q.push(temp->right);
  }
}
int main()
{
  RBTree r;
  r.insertion(50);
  r.insertion(20);
  r.insertion(65);
  r.insertion(15);
  r.insertion(35);
  r.insertion(55);
  r.insertion(70);
  r.insertion(68);
  r.insertion(80);
  r.insertion(75);
  r.disp();
  levelorder(r.root);
  cout<<endl;
  r.deleteByVal(55);
  r.disp();
  cout<<endl;
  r.deleteByVal(68);
  r.disp();
  cout<<endl;
  r.deleteByVal(90);
  r.disp();
  cout<<endl;
  r.deleteByVal(80);
  r.disp();
  cout<<endl;
  r.deleteByVal(50);
  r.disp();
  cout<<endl;
  r.deleteByVal(35);
  r.disp();
  cout<<endl;
  r.deleteByVal(15);
  r.disp();
  cout<<endl;
}
```

Comparison between AVL and Red Black tree

Basis of comparison	Red Black Trees	AVL Trees
Colour	The color of the node is either Red or Black.	There is no color of the node.
Insertion and removal	They provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing[7].	They provide complex insertion and removal operations as more rotations are done due to relatively strict balancing[7].

Storage	They requires only 1 bit of information per node[9].	They store balance factors or heights with each node thus requiring storage for an integer per node[9].
Searching	It does not provide efficient searching.	It provides efficient searching.
Uses	They are used in most of the language libraries like map, multimap, multiset in C++, etc.	They are used in databases where faster retrievals are required.
Balance Factor	It does not have balance factor	Each node has a balance factor whose value is either 1 or 0 or -1
Balancing	Requires less processing to balance the tree, i.e. maximum 2 rotations required.	Requires more processing to balance the tree.

REFERENCE

[1] www.biitonline.co.in (Author Pankaj Kumar Gupta, Head, BCA Department, DPBS College, Anupshahr)
[2] Computer Science with C++ By SumitaArora.
[3] Let Us C by Yashavant Kanetkar
[4] Sartajsahni, "Data structures, algorithms and applications in C++", University press.
[5] Seymour Lipschutz. "Theory and problems of data structures", Tata Mcgraw hill international editions".
[6] The C Programming Language by Brian W. Kernighan / Dennis Ritchie
[7] www.en.wikipedia.org/wiki/Array.
[8] https://www.codecademy.com
[9] www.geeksforgeeks.org
[10] www.w3schools.com
[11] Introduction to Algorithms, 3rd Edition (The MIT Press).
[12] Data Structures (Revised First Edition) | Schaum's Outline Series by Seymour Lipschutz
[13] Algorithms and Data Structures Foundations and Probabilistic Methods for Design and Analysis By Helmut Knebl
[14] Algorithms in a Nutshell By George T. Heineman, Gary Pollice, Stanley Selkow .
[15] Data Structures and Algorithm Analysis in C++, Third Edition By Clifford A. Shaffer.
[16] Arrays: A Theoretical Approach of Memory Allocation, Pankaj Kumar Gupta & Dr. P. K. Tyagi, International Journal of Essential Sciences, Vol-14 No. 1 & 2 2020.
[17] Conceptual Discussion on Operations of Array: Traveral, Insertion & Deletion by Pankaj Kumar Gupta & Dr. P. K. Tyagi, International Journal of Research in all subjects in Multi Languages, Vol-10 Issue 3, March: 2022.
[18] https://www.programiz.com/dsa/stack
[19] https://www.geeksforgeeks.org/stack-data-structure/
[20] Fundamental of computer algorithms by Ellis Horowitz and Sartaj Sahni
[21] Data Structure Programming With C by Gopal pathak, L.K. Vishwamitra and Sudhir Goswami
[22] Data Structures Through C by G.S. Baluja
[23] Data Structures using C and C++ by Rajesh K. Shukla.