Project :

Table of Contents :

I – RedBlack Tree :

1-What is a RedBlack Tree ?

- A red-black tree is a special kind of the binary search tree where each tree's node stores a color, which is either red or black. A red-black tree is a self-balancing binary search tree, in which the insert or remove operation is done intelligently to make sure that the tree is always balanced.
- The complexity of any operation in the tree such as search, insert or delete is O(logN) where N is the number of nodes in the red-black tree.
- The red-black tree data structure is used to implement associative arrays.
- In brief, A **red–black tree** is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions

2-Rules

- **1)** Every node has a color either red or black.
- **2)** Root of 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 descendant NULL node has the same number of black nodes.

3-Why RedBlack Tree ?

- 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 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.

4-Comparison with AVL:

- The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent

and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

5- Operations:

a-Insertion:

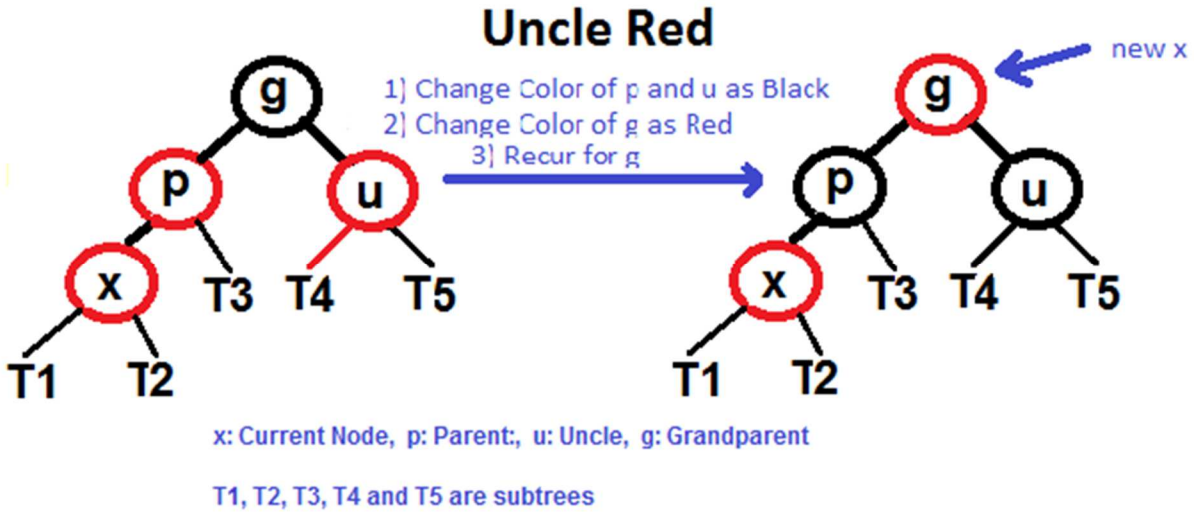
In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

1) Recoloring
2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Example :



Let x be the newly inserted node.

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

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

3) Do following if color of x's parent is not BLACK or x is not root.

….**a) If x's uncle is RED** (Grand parent must have been black from property 4)

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

……..**(ii)** color of grand parent 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)

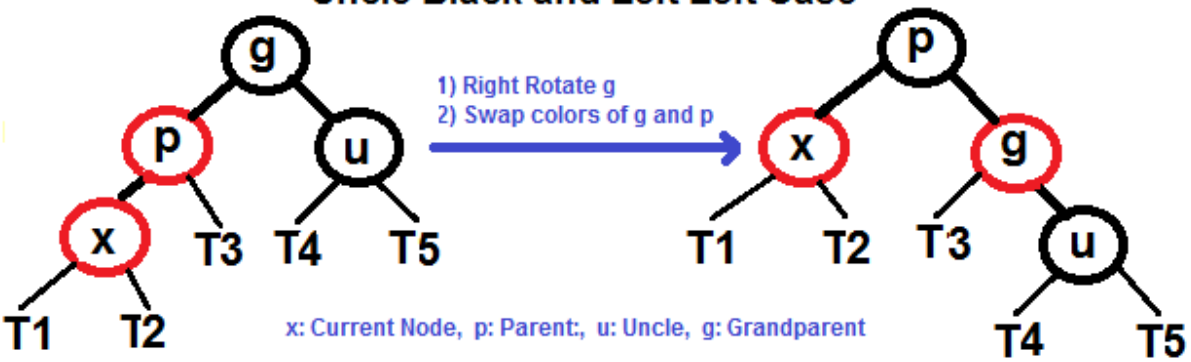
……..**ii)** Left Right Case (p is left child of g and x is right child of p)

……..**iii)** Right Right Case (Mirror of case i)

……..**iv)** Right Left Case (Mirror of case ii)

Following are operations to be performed in four subcases when uncle is BLACK.

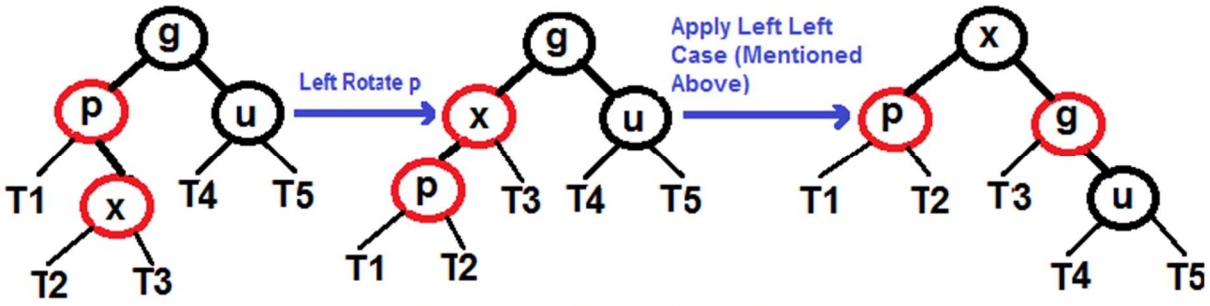
Uncle Black and Left Left Case



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

Uncle Black and Left Right Case

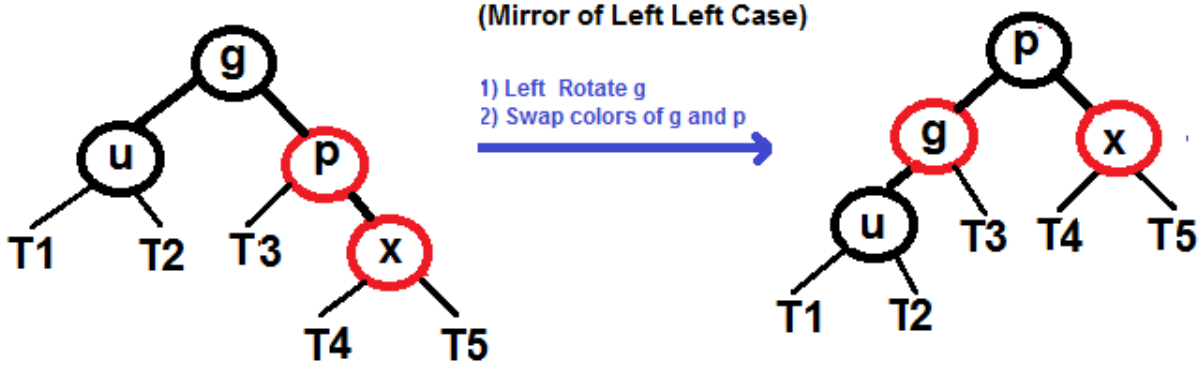


x: Current Node, p: Parent:, u: Uncle, g: G

T1, T2, T3, T4 and T5 are subtrees

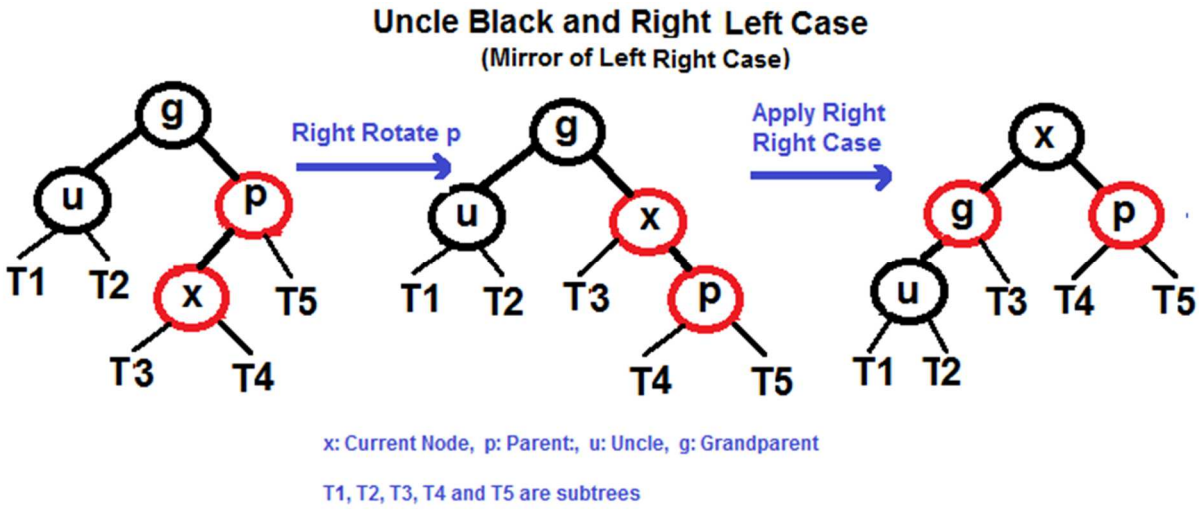
Uncle Black and Right Right Case

(Mirror of Left Left Case)



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees



b)Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

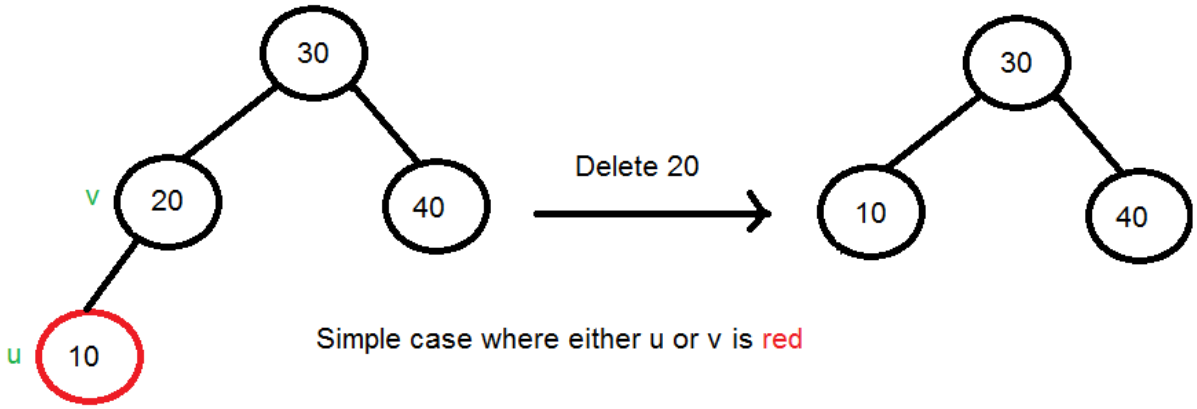
Deletion Steps

Following are detailed steps for deletion.

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either

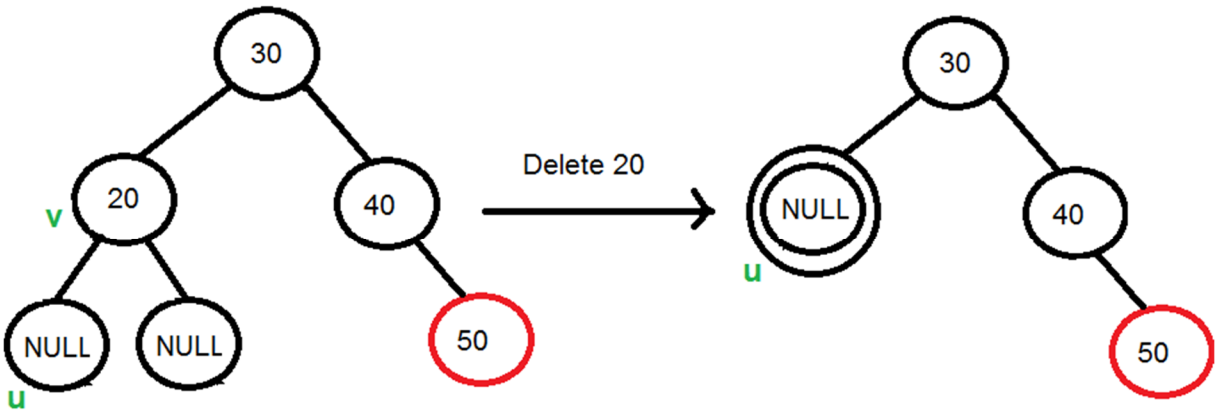
leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

3.2) Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

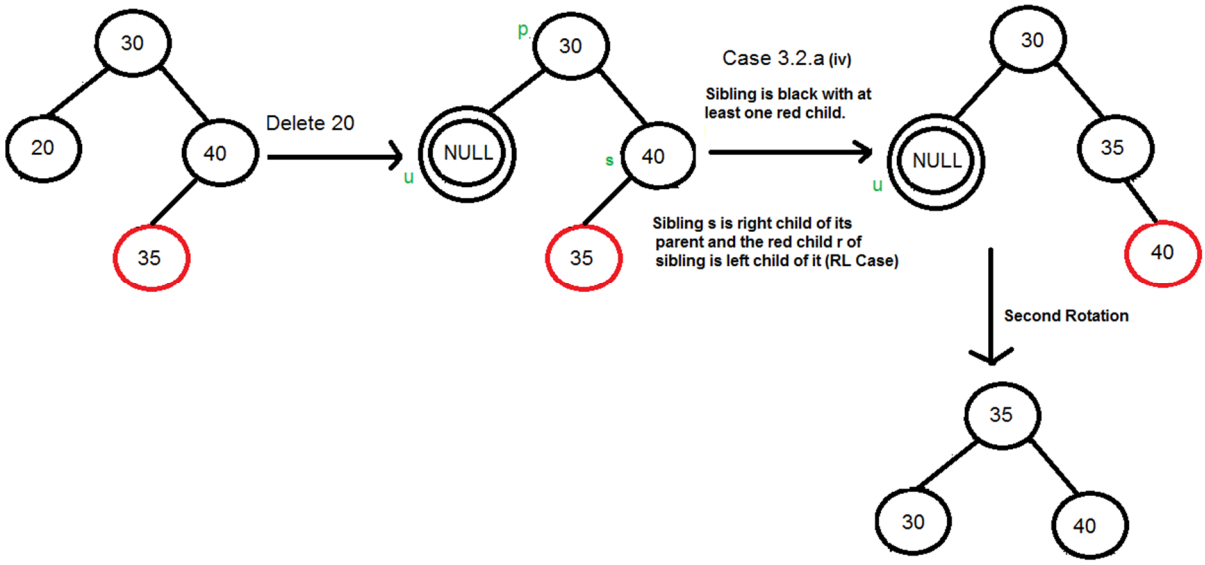
….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

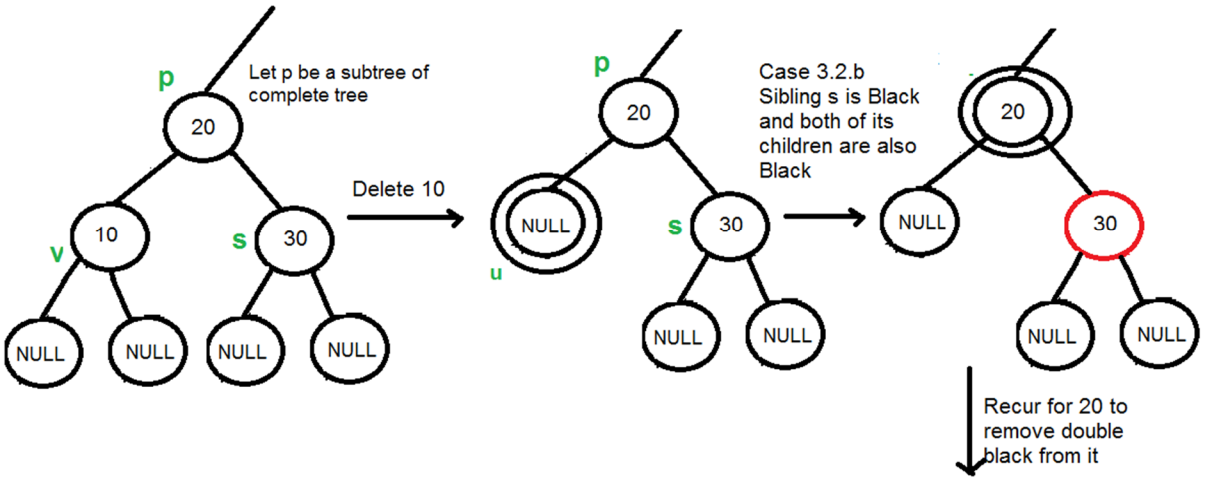
(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

(iii) Right Right Case (s is right child of its parent and r is right child of

s or both children of s are red)



(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.



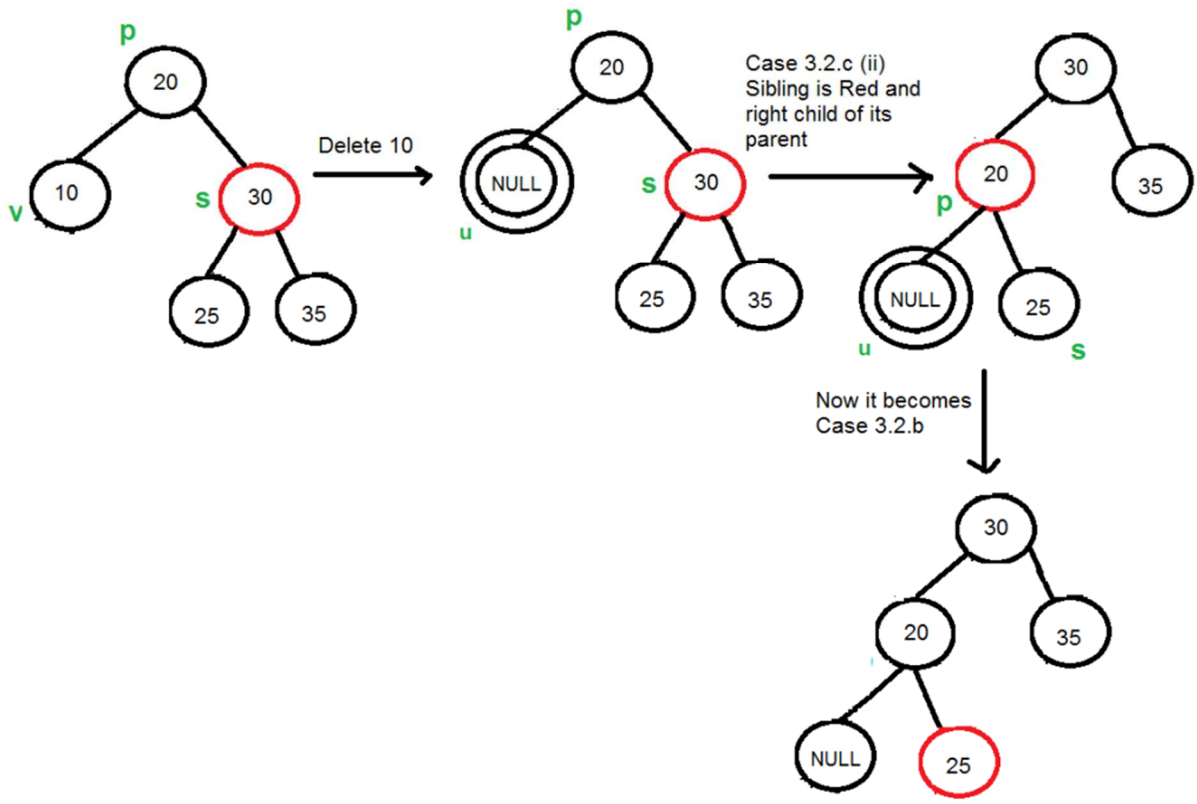
In this case, if parent was red, then we didn't need to recur for prent, we can simply make it black (red + double black = single black)

(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

(i) Left Case (s is left child of its parent). This is mirror of right right

case shown in below diagram. We right rotate the parent p.

(iii) Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

6)Code:

RedBlack.h

```c
#include <stdlib.h>
#include "fatal.h"
typedef int ElementType;
#define NegInfinity (-10000)
#ifndef _RedBlack_H
#define _RedBlack_H

struct RedBlackNode;
typedef struct RedBlackNode *Position;
typedef struct RedBlackNode *RedBlackTree;

RedBlackTree MakeEmpty(RedBlackTree T);
Position Find(ElementType X, RedBlackTree T);
Position FindMin(RedBlackTree T);
Position FindMax(RedBlackTree T);
RedBlackTree Initialize(void);
RedBlackTree Insert(ElementType X, RedBlackTree T);
RedBlackTree Remove(ElementType X, RedBlackTree T);
ElementType Retrieve(Position P);
void PrintTree(RedBlackTree T);
 #endif  /* _RedBlack_H */
```

main.c

The main code:

fatal.h

fatal.c

Fatal:

II– AVL Tree :

1)Definition:

- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights (h) of left and right subtrees cannot be more than one for all nodes.

2) Use:

- 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 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 an AVL tree is always O(Logn) where n is the number of nodes in the tree.

3)Complexity:

- The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

4)Comparison with RedBlack:

- The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

5) Operations:

a)Insertion:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1) Left Rotation

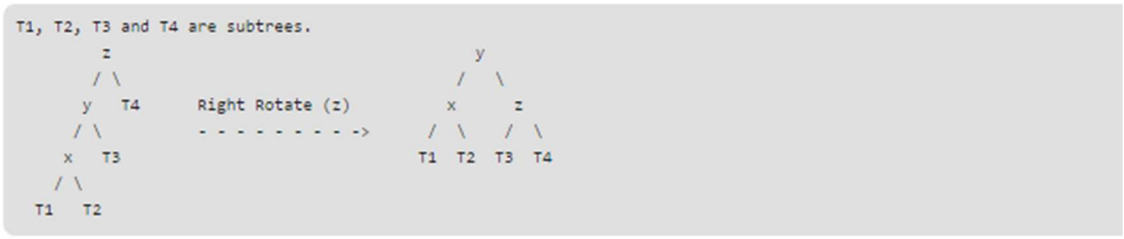
2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -           / \
 T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the
following order
 keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

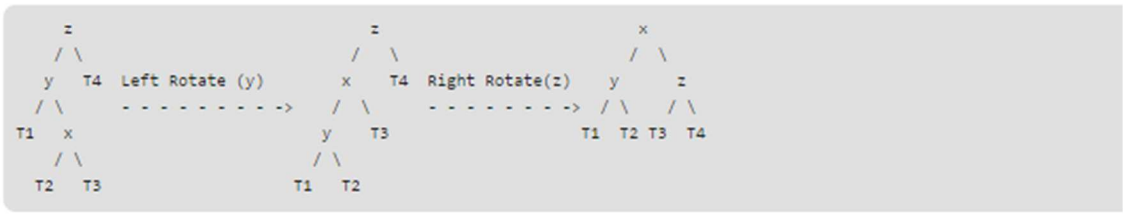
Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

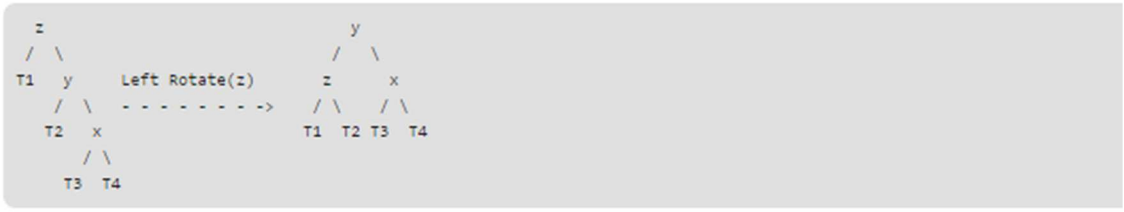
a) Left Left Case



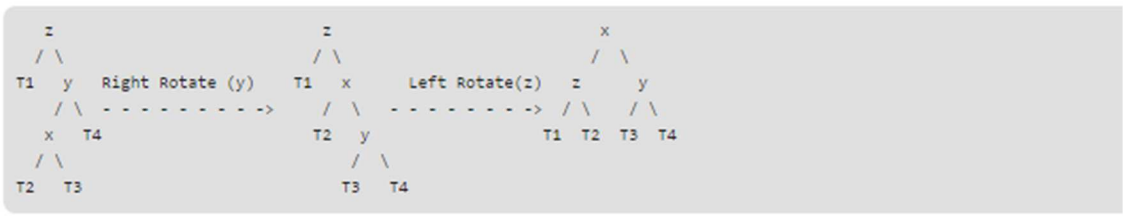
b) Left Right Case



c) Right Right Case



d) Right Left Case



Time Complexity: The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Comparison with Red Black Tree

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

b)Deletion:

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1) Left Rotation

2) Right Rotation

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case).

6)Code:

avltree.h

```h
#ifndef AVLTREE_H_INCLUDED
#define AVLTREE_H_INCLUDED

typedef struct node
{
    int data;
    struct node*  left;
    struct node*  right;
    int      height;
} node;


void dispose(node* t);
node* find( int e, node *t );
node* find_min( node *t );
node* find_max( node *t );
node* insert( int data, node *t );
node* delete( int data, node *t );
void display_avl(node* t);
int get( node* n );
#endif // AVLTREE_H_INCLUDED
```

```c
#include <stdio.h>
#include "avltree.h"

int main()
{
    node *t , *p;
    int i;
    int j = 0;
    const int max = 10;

    printf("--- C AVL Tree Demo  ---\n");

    t = NULL;

    printf("Insert: ");
    for( i = 0; i < max; i++, j = ( j + 7 ) % max )
    {

        t = insert( j, t );
        printf("%d ",j);

    }
    printf(" into the tree\n\n");

    display_avl(t);

    dispose(t);

    return 0;
}
```