RV Educational Institutions ®
RV College of Engineering ®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi, Accredited By NAAC, Bengaluru And NBA, New Delhi

Go, change the world

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Red and Black Trees

Data Structures & Applications

(18IS33)

2021-2022

Submitted by

Name: Neha N	**USN: 1RV20CS094**
Name: Nimisha Dey	**USN: 1RV20CS098**

Under the guidance of

SUMA B
Designation of Faculty 2
Dept. of CSE
RV College of Engineering

ACKNOWLEDGEMENT

It gives us immense pleasure to be associated with this project titled “RED AND BLACK TREES”. The project was a joyous learning process that helped us in linking our theoretical knowledge to practical knowledge. We got to learn a lot from this project and gain in depth knowledge about the implementation and practical application of red black trees.

Firstly, we would like to express our sincere gratitude to our teacher Prof. Suma B as well as our Principal Dr. K N Subramanya who gave us the golden opportunity to do this wonderful project. I extend my heartfelt thanks to our teacher who has helped us in this endeavor and has always been very cooperative in providing us with the necessary information regarding the project.

Secondly, we would like to thank our family and friends for providing us with unfailing support and continuous encouragement through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Last but not the least, we would like to thank all those who had helped us (directly or indirectly) towards the completion of this project within the limited time frame. It was a great learning experience for us.

CONTENTS

1.ABSTRACT

The data structure is a specialized method to organize and store data in the computer to be used more effectively. There are various types of data structures, such as stack, linked list and queue, in which data is arranged in sequential order. Data structures such as tree and graph store data non sequentially. A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes. Trees may be further classified as balanced and unbalanced trees. A binary tree is a tree in which every node has a maximum of 2 sub nodes. A binary search tree is a tree in which value contained in the left child is less than the node and value contained in the right child is more than the node. A height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differs by not more than 1.

A self-balancing binary search tree is any node-based binary search tree that automatically keeps its height small in the face of arbitrary item insertions and deletions. A red black tree is a type 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 the tree remains balanced during insertions and deletion operations. This report explains red black trees in detail with examples.

This report summarizes the different concepts associated with red black trees. It gives a brief overview on the different properties of red black trees with special emphasis on the height balancing property in red black trees. The report also talks about the different applications of red black trees in practical life. The report explains in detail the different operations that can be performed on a red black tree including rotate, insert, delete and search with algorithm, flowchart and code snapshot. The report also explains the advantages and disadvantages of black and red trees.

Finally the report includes screenshots of a trial run showing sample inputs and respective outputs for different operations – insert, delete, search for minimum and maximum elements.

2. INTRODUCTION

A red-black tree is a 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 the tree remains balanced during insertions and deletions.

2.1 Balance in red-black trees

The red-black color is meant for balancing the tree. The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

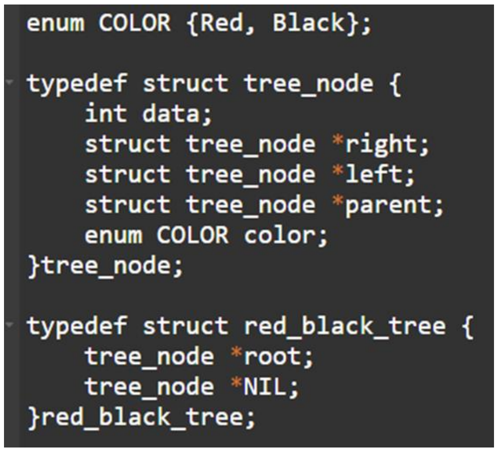
2.2 Properties of red-black trees

1. Each node is either red or black.
2. The root of the tree is always black.
3. All leaves are null and they are black.
4. The children of any red node are black.
5. Any path from a given node to any of its descendant leaves contains the same number of black nodes.
6. The black depth of a node is defined as the number of black nodes from the root to that node i.e. the number of black ancestors. The black height of a node is the number of black nodes from the given node to any of its descendant child nodes (including the child node).
7. Number of nodes from a node to its farthest descendent leaf is no more than twice the number of nodes to the nearest descendent leaf. Red-Black Tree of height h has black-height >= h/2.
8. Every Red Black Tree with n nodes has height <= 2Log2(n+1).

2.3. Structure of Red Black Trees

Each node has the following attributes as shown in Fig. 1:

- color
- key
- leftChild
- rightChild
- parent (except root node)



```c
enum COLOR {Red, Black};

typedef struct tree_node {
    int data;
    struct tree_node *right;
    struct tree_node *left;
    struct tree_node *parent;
    enum COLOR color;
}tree_node;

typedef struct red_black_tree {
    tree_node *root;
    tree_node *NIL;
}red_black_tree;
```

Fig. 1: Structure of Red Black Trees

3.<u>APPLICATIONS</u>

●Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time.

●Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures which provide worst-case guarantees

●For example, many data structures used in computational geometry can be based on red–black trees, and the Completely Fair Scheduler used in current Linux kernels uses red–black trees.

●Linux also uses red-black trees in the *mmap* and *munmap* operations for file/memory mapping.

●*TreeSet*, *TreeMap*, and *Hashmap* in the Java Collections Library use red black trees.

●Red-black trees are used for geometric range searches, k-means clustering, and text-mining.

4. OPERATIONS

As shown in Fig. 2, the following operations have been implemented.

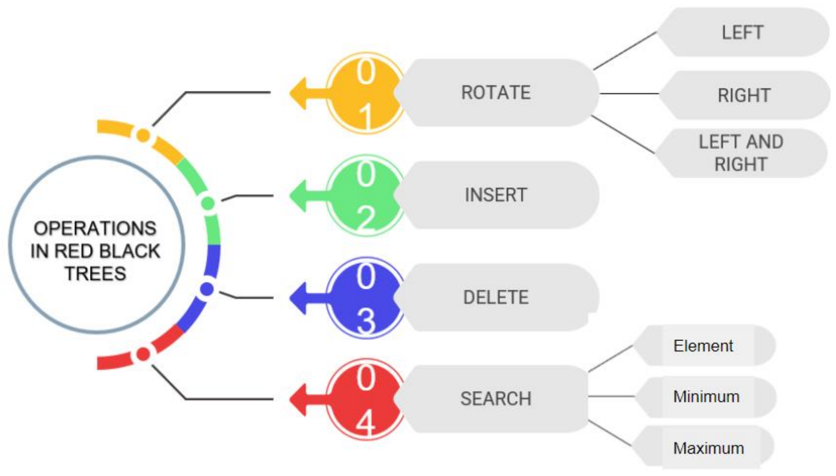


Fig 2: Operations

4.1 Rotation

In rotation operation, the positions of the nodes of a subtree are interchanged. Rotation operation is used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

There are 3 types of rotation

- Left rotate
- Right rotate
- Left-Right rotate

4.1.1 Left rotate

Algorithm

```
Step 1: If y has a left subtree, assign x as the parent of the left subtree of y
Step 2: Change parent of x to that of y
        a.If the parent of x is NULL, make y as the root of the tree
        b.Else if x is the left child of p, make y as the left child of p
        c.Else assign y as the right child of p
Step 3: Make y as the parent of x (Fig. 3).
```

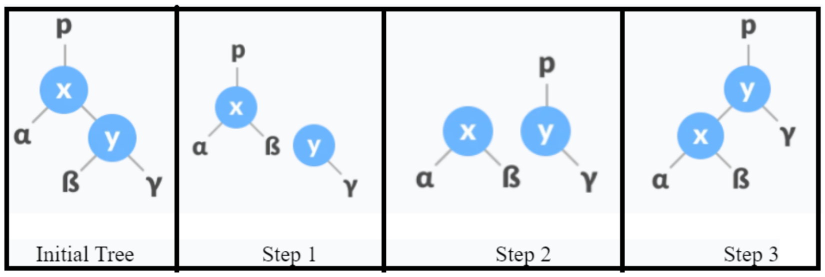
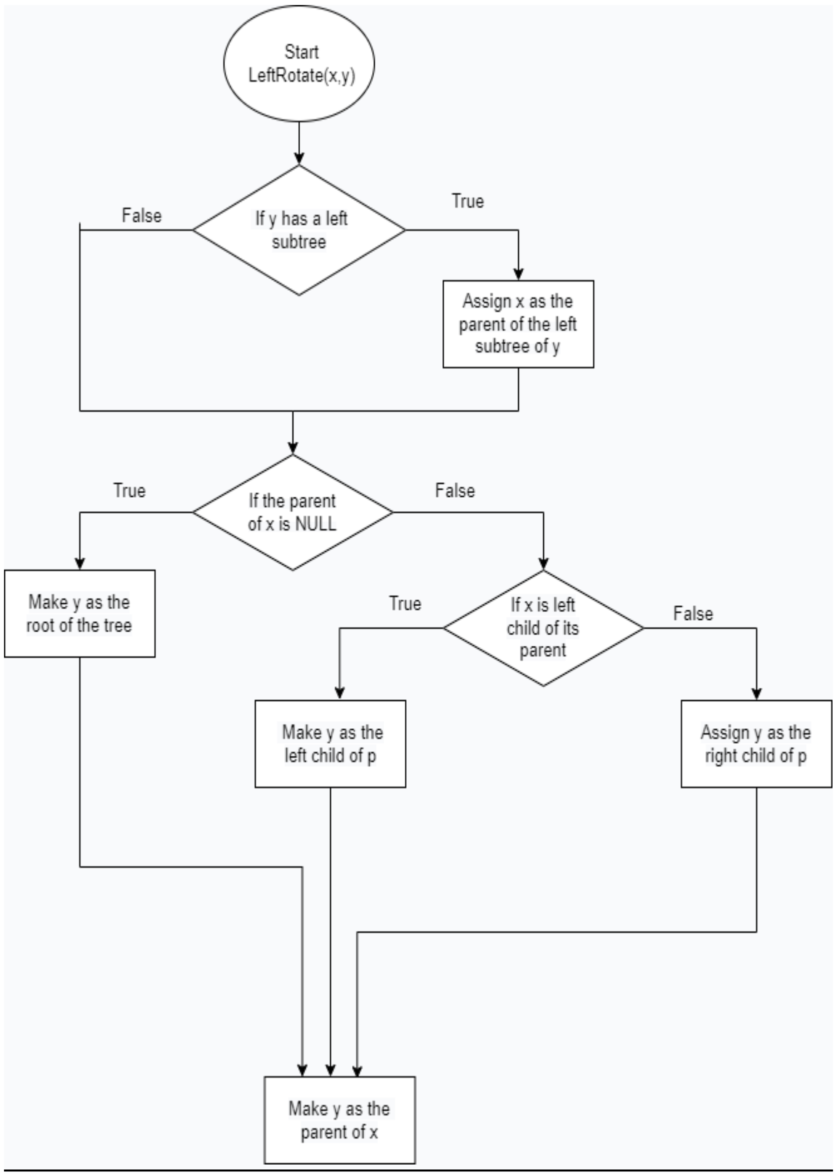


Fig. 3: Example of Left Rotation

Flowchart and code for Left Rotation is as follows (Fig. 4 and 5):

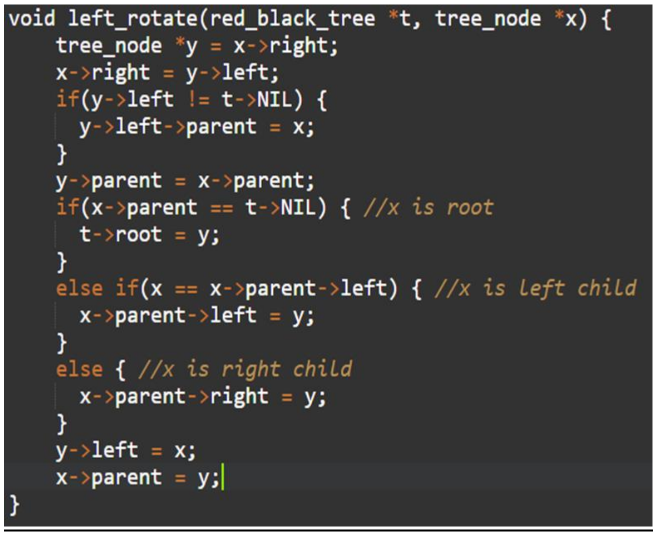
Flowchart



Start LeftRotate(x,y)
If y has a left subtree
False
True
Assign x as the parent of the left subtree of y
If the parent of x is NULL
True
False
Make y as the root of the tree
If x is left child of its parent
True
False
Make y as the left child of p
Assign y as the right child of p
Make y as the parent of x

Fig. 4: Flowchart for Left Rotation

Code



```c
void left_rotate(red_black_tree *t, tree_node *x) {
    tree_node *y = x->right;
    x->right = y->left;
    if(y->left != t->NIL) {
      y->left->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == t->NIL) { //x is root
      t->root = y;
    }
    else if(x == x->parent->left) { //x is left child
      x->parent->left = y;
    }
    else { //x is right child
      x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
```

Fig. 5: Function for Left Rotation

4.1.2 Right rotate

Algorithm

Step 1: If y has a right subtree, assign x as the parent of the right subtree of y

Step 2: Change parent of y to that of x

a.If the parent of y is NULL, make x as the root of the tree

b.Else if y is the right child of p, make x as the right child of p

c.Else assign x as the left child of p

Step 3: Make y as the parent of x (Fig. 6).

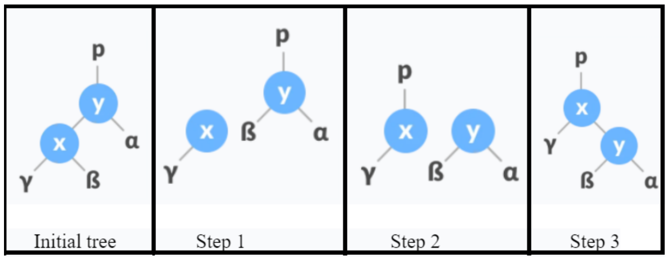


Fig. 6: Example of Right Rotation

Flowchart and code for Left Rotation is as follows (Fig. 7 and 8):

Flowchart

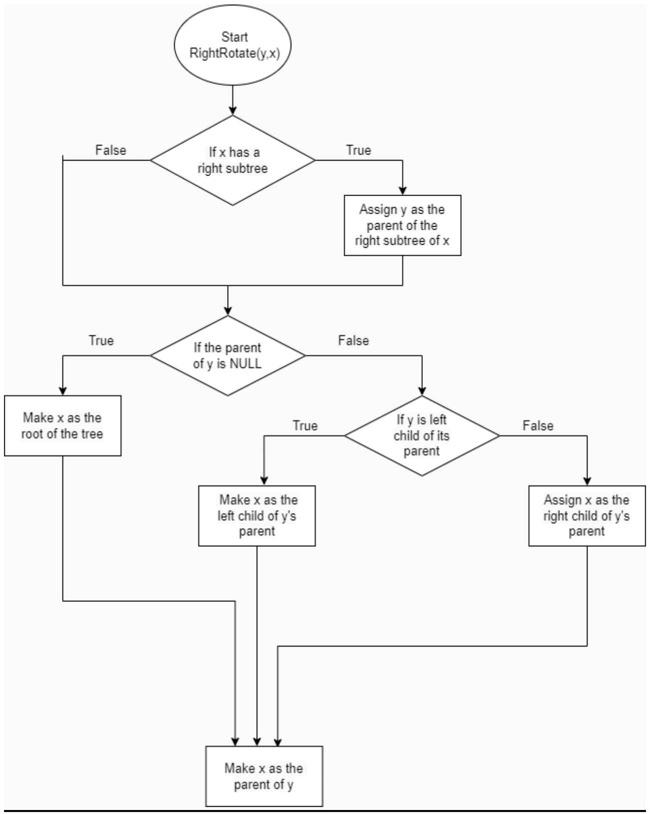
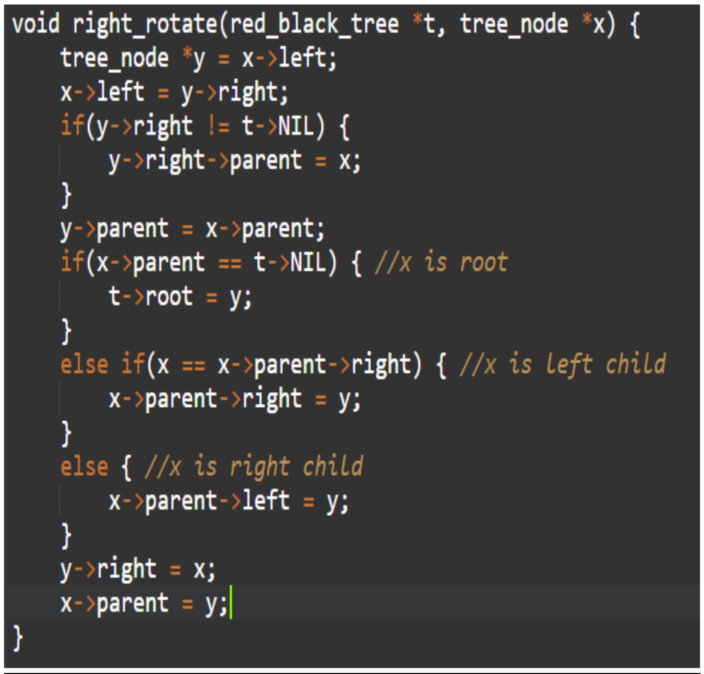

Start
RightRotate(y,x)
If x has a right subtree
False
True
Assign y as the parent of the right subtree of x
If the parent of y is NULL
True
False
Make x as the root of the tree
If y is left child of its parent
True
False
Make x as the left child of y's parent
Assign x as the right child of y's parent
Make x as the parent of y

Fig. 7: Flowchart for Right Rotation

Code



```
void right_rotate(red_black_tree *t, tree_node *x) {
    tree_node *y = x->left;
    x->left = y->right;
    if(y->right != t->NIL) {
        y->right->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == t->NIL) { //x is root
        t->root = y;
    }
    else if(x == x->parent->right) { //x is left child
        x->parent->right = y;
    }
    else { //x is right child
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
```

Fig. 8: Example of Right Rotation

4.1.3 Left and Right Rotation

Left-Right rotation

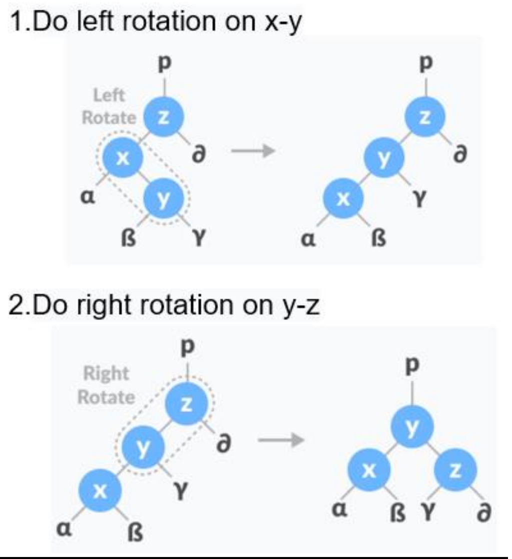


Fig. 9: Example of Left-Right Rotation

Right-Left rotation

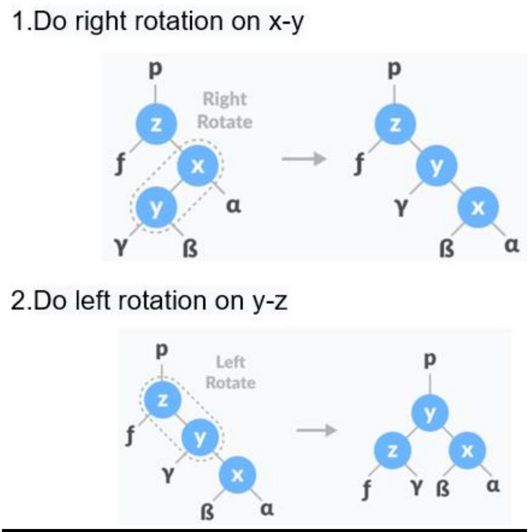


Fig. 10: Example of Right-Left Rotation

4.2 Insert

While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

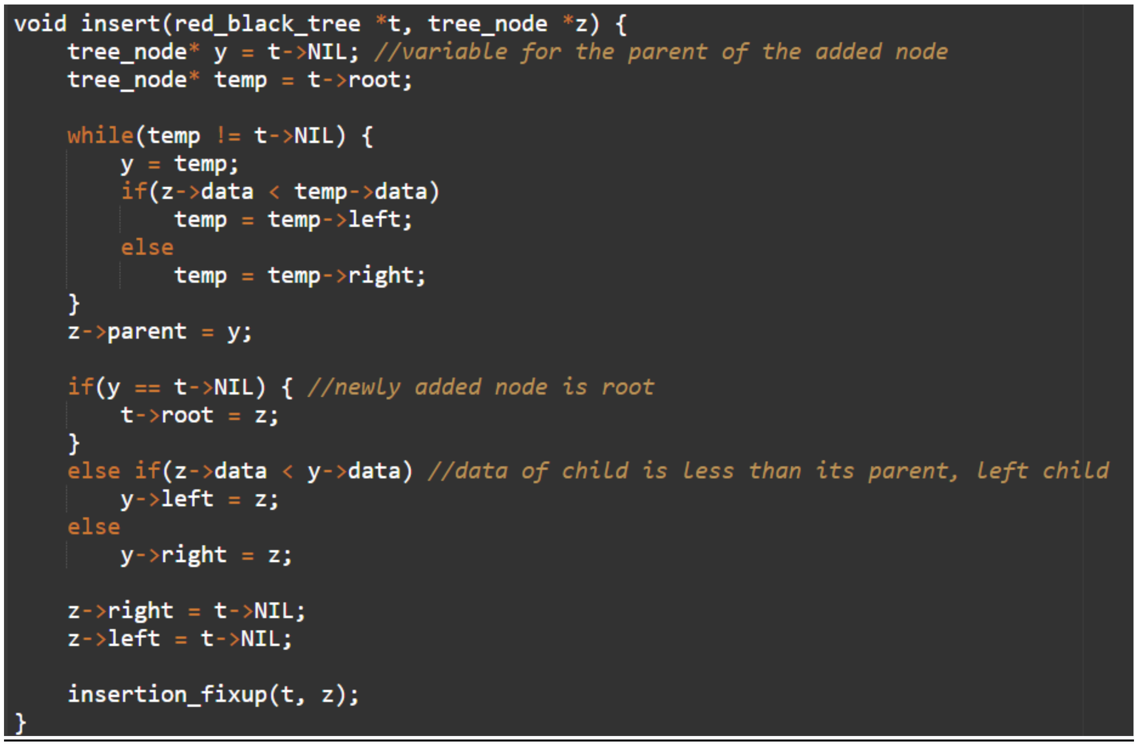
- Recolor
- Rotation

Newly inserted nodes are always red because inserting a red node does not violate the depth property of a red-black tree.

If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Algorithm for insert operation

1. Let ‘y’ be the leaf (ie. NIL) and ‘temp’ be the root of the tree.
2. Check if the tree is empty (ie. whether x is NIL). If yes, insert newNode as a root node and color it black.
3. Else, repeat steps following steps until leaf (NIL) is reached.
    a. Compare newKey with rootKey.
    b. If newKey is greater than rootKey, traverse through the right subtree.
    c. Else traverse through the left subtree.
4. Assign the parent of the leaf as a parent of newNode.
5. If leafKey is greater than newKey, make newNode as rightChild.
6. Else, make newNode as leftChild.
7. Assign NULL to the left and rightChild of newNode.
8. Assign RED color to newNode.
9. Call InsertFix-algorithm to maintain the property of red-black tree if violated (Fig. 11).



```
void insert(red_black_tree *t, tree_node *z) {
    tree_node* y = t->NIL; //variable for the parent of the added node
    tree_node* temp = t->root;

    while(temp != t->NIL) {
        y = temp;
        if(z->data < temp->data)
            temp = temp->left;
        else
            temp = temp->right;
    }
    z->parent = y;

    if(y == t->NIL) { //newly added node is root
        t->root = z;
    }
    else if(z->data < y->data) //data of child is less than its parent, left child
        y->left = z;
    else
        y->right = z;

    z->right = t->NIL;
    z->left = t->NIL;

    insertion_fixup(t, z);
}
```

Fig.11 Code for insert operation

<u>Algorithm for maintaining red black property after insertion</u>

1. Do the following while the parent of newNode i.e., p is RED.

2.If p is the left child of grandParent gP of z, do the following.

Case-I:

a.If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

b.Assign gP to newNode.

Case-II:

c.Else if newNode is the right child of p then, assign p to newNode.

d.Left-Rotate newNode.

Case-III:

e.Set color of p as BLACK and color of gP as RED.

f.Right rotate gp

3. Else do the following

a. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

b. Assign gP to newNode

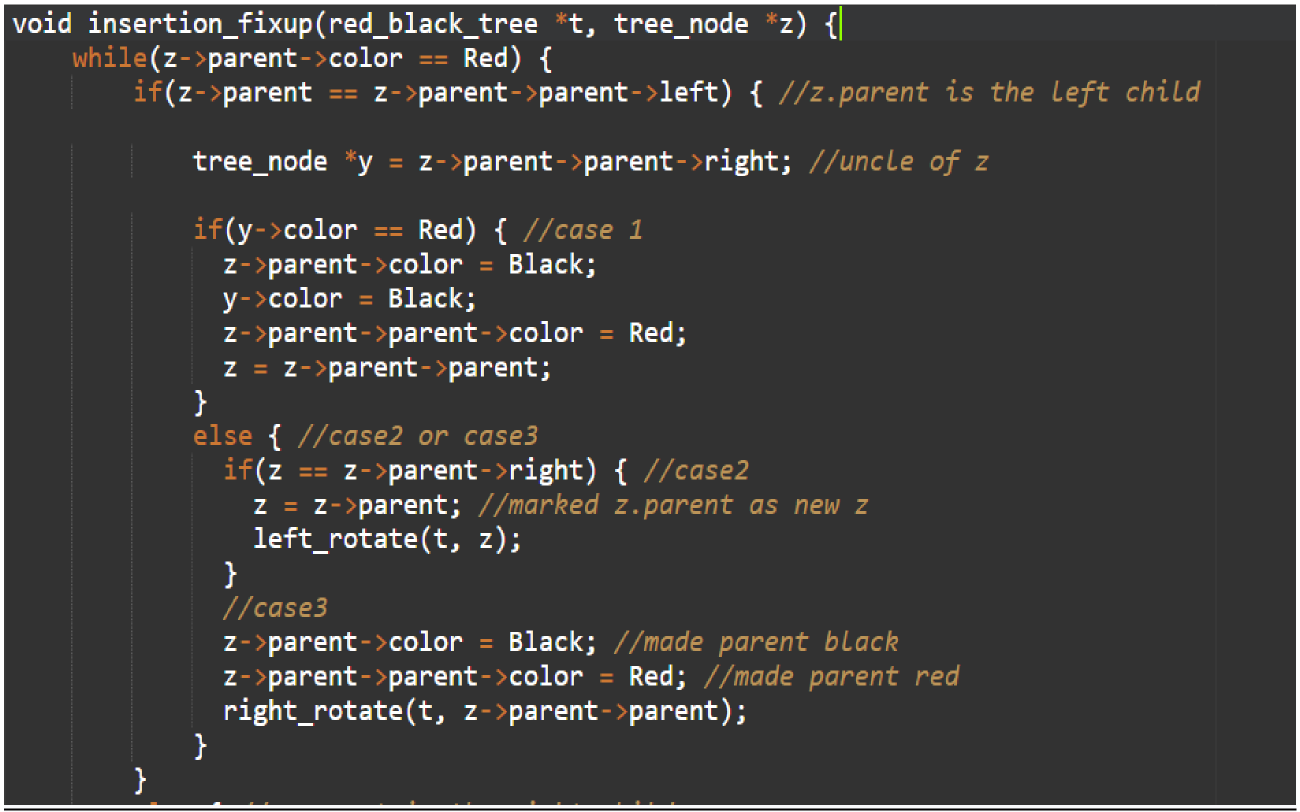
c. Else if newNode is the left child of p then, assign p to newNode and Right-Rotate newNode

d. Set color of p as BLACK and color of gP as RED.

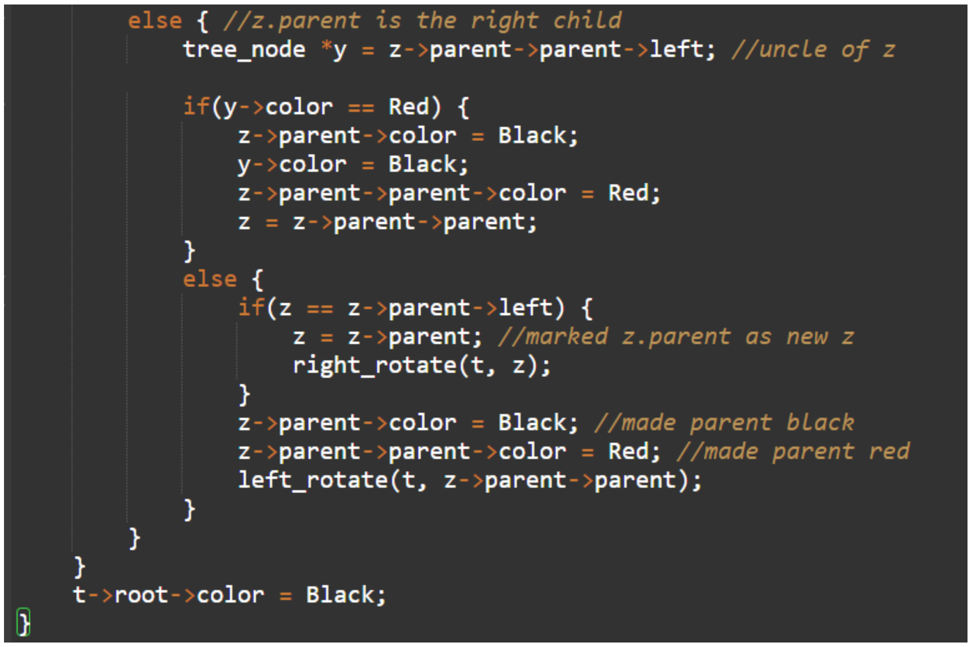
e. Left-Rotate gP

4.Set the root of the tree as BLACK (Fig. 12).

<u>Code</u>



```
void insertion_fixup(red_black_tree *t, tree_node *z) {
    while(z->parent->color == Red) {
        if(z->parent == z->parent->parent->left) { //z.parent is the left child

            tree_node *y = z->parent->parent->right; //uncle of z

            if(y->color == Red) { //case 1
              z->parent->color = Black;
              y->color = Black;
              z->parent->parent->color = Red;
              z = z->parent->parent;
            }
            else { //case2 or case3
              if(z == z->parent->right) { //case2
                z = z->parent; //marked z.parent as new z
                left_rotate(t, z);
              }
              //case3
              z->parent->color = Black; //made parent black
              z->parent->parent->color = Red; //made parent red
              right_rotate(t, z->parent->parent);
            }
        }
```



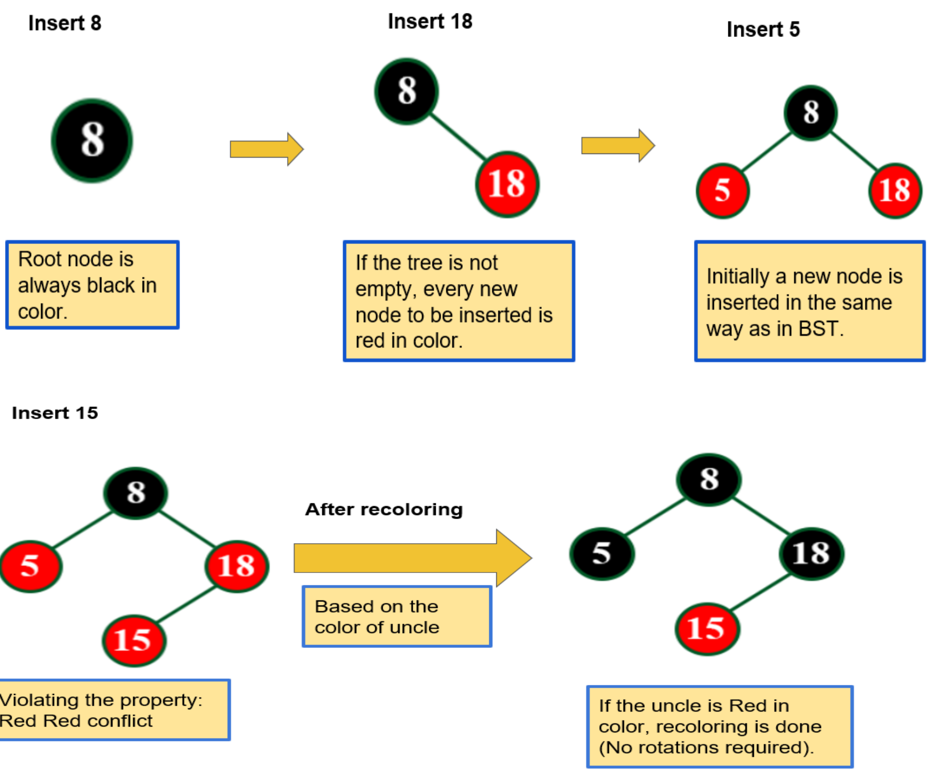
```
        else { //z.parent is the right child
            tree_node *y = z->parent->parent->left; //uncle of z

            if(y->color == Red) {
                z->parent->color = Black;
                y->color = Black;
                z->parent->parent->color = Red;
                z = z->parent->parent;
            }
            else {
                if(z == z->parent->left) {
                    z = z->parent; //marked z.parent as new z
                    right_rotate(t, z);
                }
                z->parent->color = Black; //made parent black
                z->parent->parent->color = Red; //made parent red
                left_rotate(t, z->parent->parent);
            }
        }
    }
    t->root->color = Black;
}
```

Fig. 12: Code for maintaining the red black property after insertion

<u>Algorithm example</u>

Creating a RED BLACK Tree by inserting following sequence of number as shown in Fig. 13:

8,18, 5, 17, 25, 40, 80


Insert 8
8
Root node is always black in color.
Insert 18
8
18
If the tree is not empty, every new node to be inserted is red in color.
Insert 5
8
5
18
Initially a new node is inserted in the same way as in BST.
Insert 15
8
5
18
15
Violating the property: Red Red conflict
After recoloring
Based on the color of uncle
8
5
18
15
If the uncle is Red in color, recoloring is done (No rotations required).

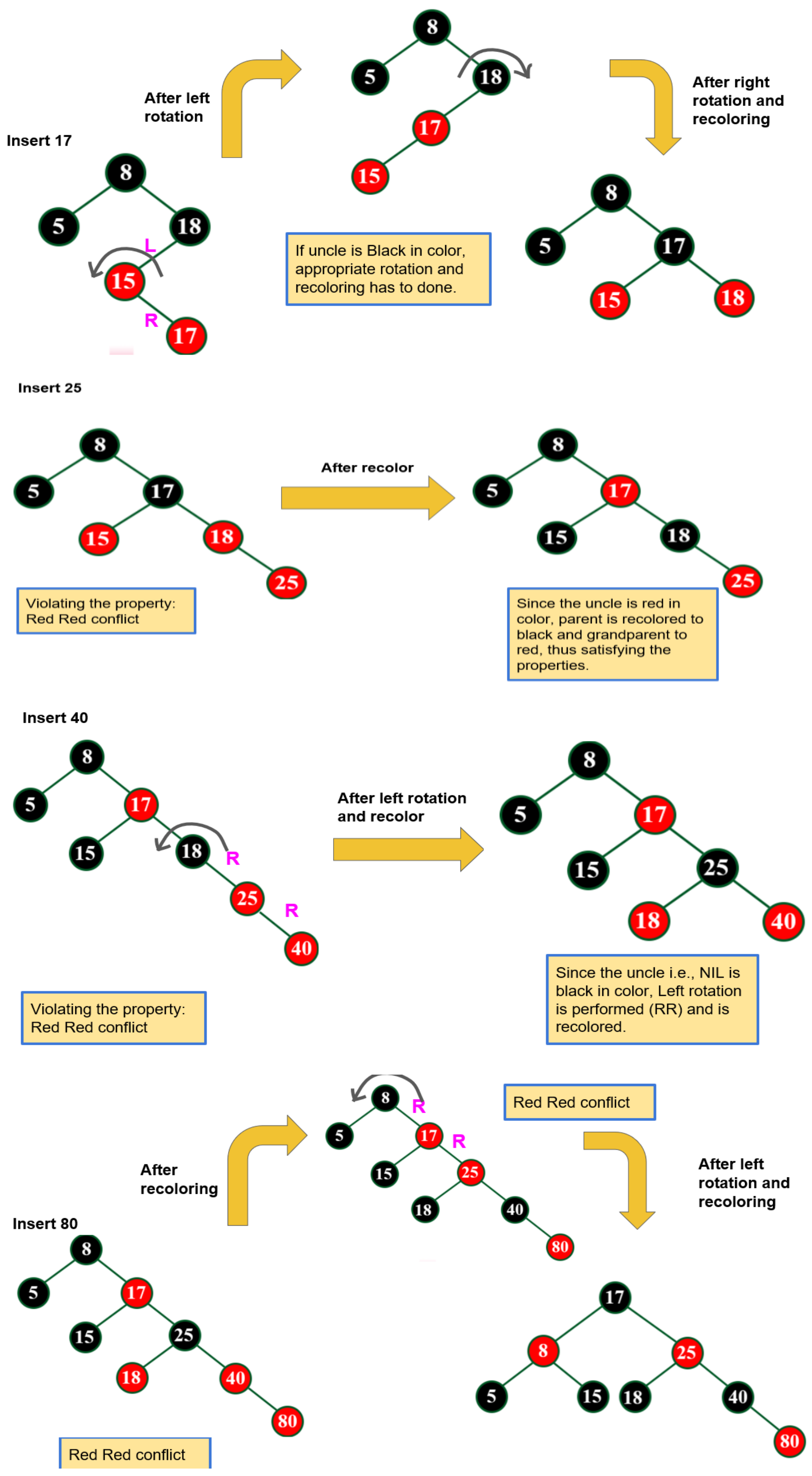

Insert 17
L
R
After left rotation
If uncle is Black in color, appropriate rotation and recoloring has to done.
After right rotation and recoloring
Insert 25
After recolor
Violating the property: Red Red conflict
Since the uncle is red in color, parent is recolored to black and grandparent to red, thus satisfying the properties.
Insert 40
After left rotation and recolor
Violating the property: Red Red conflict
Since the uncle i.e., NIL is black in color, Left rotation is performed (RR) and is recolored.
Insert 80
After recoloring
Red Red conflict
After left rotation and recoloring
Red Red conflict

Fig. 13: Example of insertion of node (without violating the red black property)

4.3 Delete

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.

Algorithm for delete operation

```
1.Save the color of nodeToBeDeleted in original Color.
2.If the left child of nodeToBeDeleted is NULL
    a..Assign the right child of nodeToBeDeleted to x.
    b.Transplant nodeToBeDeleted with x.
3. Else if the right child of nodeToBeDeleted is NULL
    a. Assign the left child of nodeToBeDeleted into x.
    b. Transplant nodeToBeDeleted with x.
4. Else
    a. Assign the minimum of right subtree of noteToBeDeleted into y.
    b. Save the color of y in originalColor.
    c. Assign the rightChild of y into x.
    d. If y is a child of nodeToBeDeleted, then set the parent of x as y.
    e. Else, transplant y with rightChild of y.
    f. Transplant nodeToBeDeleted with y.
    g. Set the color of y with originalColor.
5.If the originalColor is BLACK, call DeleteFix(x)
```

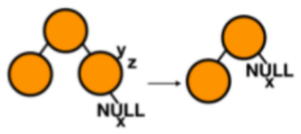
Algorithm for maintaining red black property after deletion

```
  1. Do the following until the x is not the root of the tree and the color of x is BLACK
  2. If x is the left child of its parent then,
        a. Assign w to the sibling of x.
        b. If the right child of parent of x is RED,
           Case-I:
           i.Set the color of the right child of the parent of x as BLACK.
```

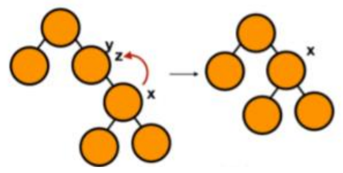
```
            ii.Set the color of the parent of x as RED.
            iii.Left-Rotate the parent of x.
            iv.Assign the rightChild of the parent of x to w.
        c.If the color of both the right and the leftChild of w is BLACK
            Case-II:
              i.Set the color of w as RED
              ii.Set the color of w as RED
        d.Else if the color of the rightChild of w is BLACK
            Case-III
              i.Set the color of the leftChild of w as BLACK
              ii.Set the color of w as RED
              iii.Right-Rotate w.
              iv.Assign the rightChild of the parent of x to w.
        e.If any of the above cases do not occur, then do the following.
            Case-IV
              i.Set the color of w as the color of the parent of x.
              ii.Set the color of the parent of x as BLACK.
              iii.Set the color of the right child of w as BLACK.
              iv.Left-Rotate the parent of x.
              v.Set x as the root of the tree.
    3.Else the same as above with right changed to left and vice versa.
    4.Set the color of x as BLACK.
```

<u>Algorithm explanation</u>

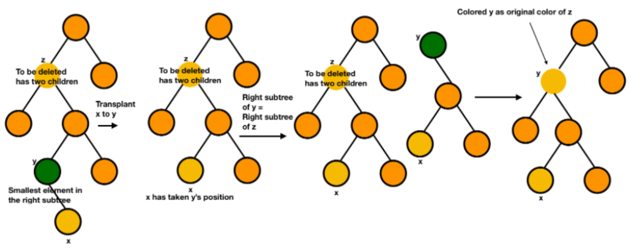
Delete function is performed similar to BST delete using following 3 cases as shown below in Fig.14 .



Case 1: z has no subtrees



Case 2: z has either left or right subtree



Case 3: z has both left and right subtrees

Fig. 14: 3 cases for delete operation

Delete operation can cause violation to depth property of red black tree if the deleted node was a black node. This violation is corrected by assuming that node x (which is occupying y's original position) has an extra black. This makes node x either doubly black or black-and-red. This violates the red-black properties.

The extra black has to be removed

- If x is red and black, we can simply color it black and this will fix the violation
- If x is doubly black and the root, we can simply remove one extra black
- If x is doubly black and it's not the root then suitable rotations and recoloring are performed according to the following 4 cases as shown in Fig.15.
    - ❖ w is red.
    - ❖ w is black and its both children are black.
    - ❖ w is black and its right child is black and its left child is red.
    - ❖ w is black and its right child is red.

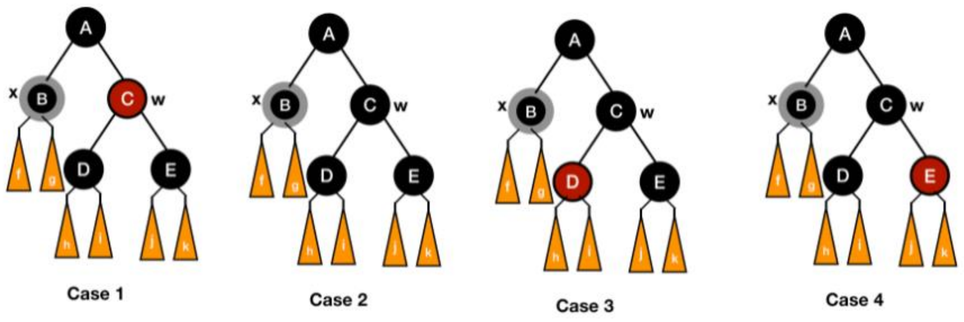


Fig. 15: 4 cases when x is doubly black

Case 1: For the first case, we can switch the colors of *w* and its parent and then left rotate the parent of *x*. In this way, we will enter either case 2, 3 or 4 as shown in Fig.16.

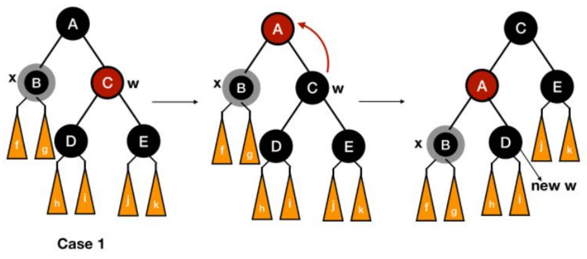


Fig. 16: w is red

Case 2: For the second case, we will take out one black from both *x* and *w*. This will leave *x* black (it was double black) and *y* red. For the compensation, we will put one extra black on the parent of *x* and mark it as the new *x* and repeat the entire process of fixing the violations for the new *x*. Also if we have entered case 2 from case 1, the parent must be red. Now the node becomes both red and black, so it will be simply fixed by coloring it black.
This case is shown below in Fig.17.

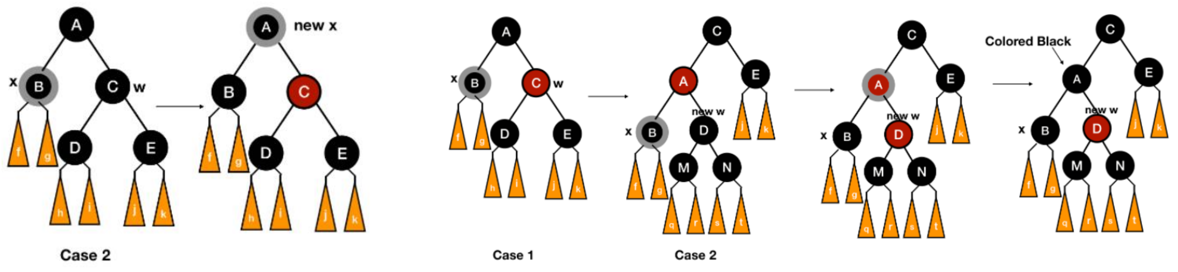


Fig. 17: w is black and its both children are black

Case 3: We will transform case 3 to case 4 by switching the colors of *w* and its left child and then rotating right *w* as shown in Fig.18.

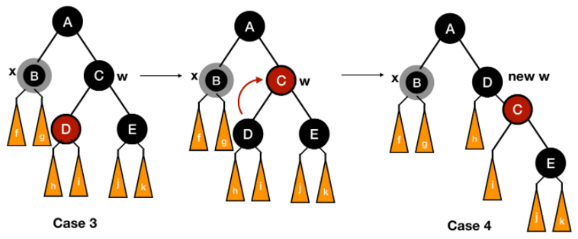


Fig. 18: w is black and its right child is black and its left child is red

Case 4: We first colored *w the same* as the parent of *x* and then colored the parent of *x* black. After this, we colored the right child of *w* black and then left rotated the parent of x. At last, we removed extra black from *x* without violating any properties as shown in Fig.19.

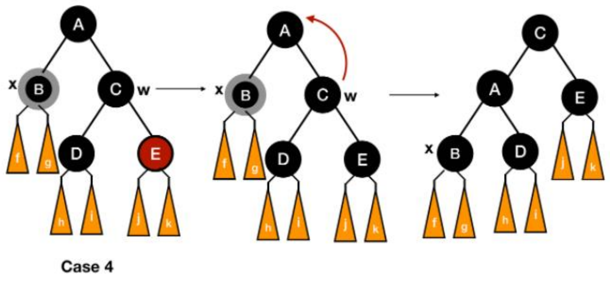
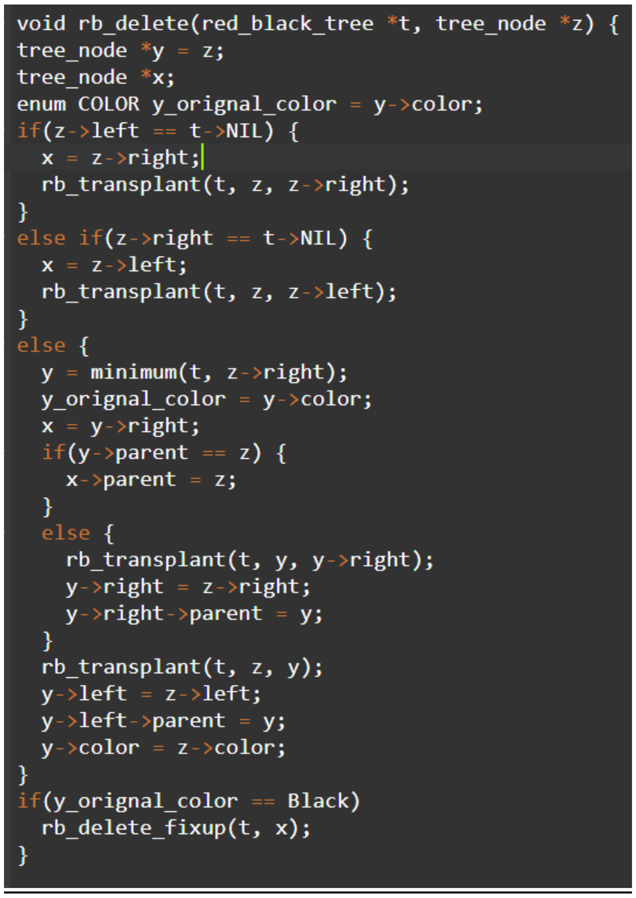


Fig. 19: w is black and its right child is red

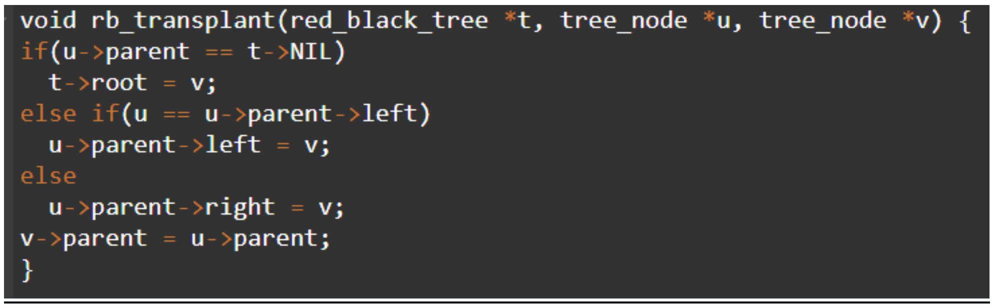
The code for deleting a node followed by code to maintain red black properties of tree is shown below in Fig. 20, Fig. 21 and Fig. 22.

Code



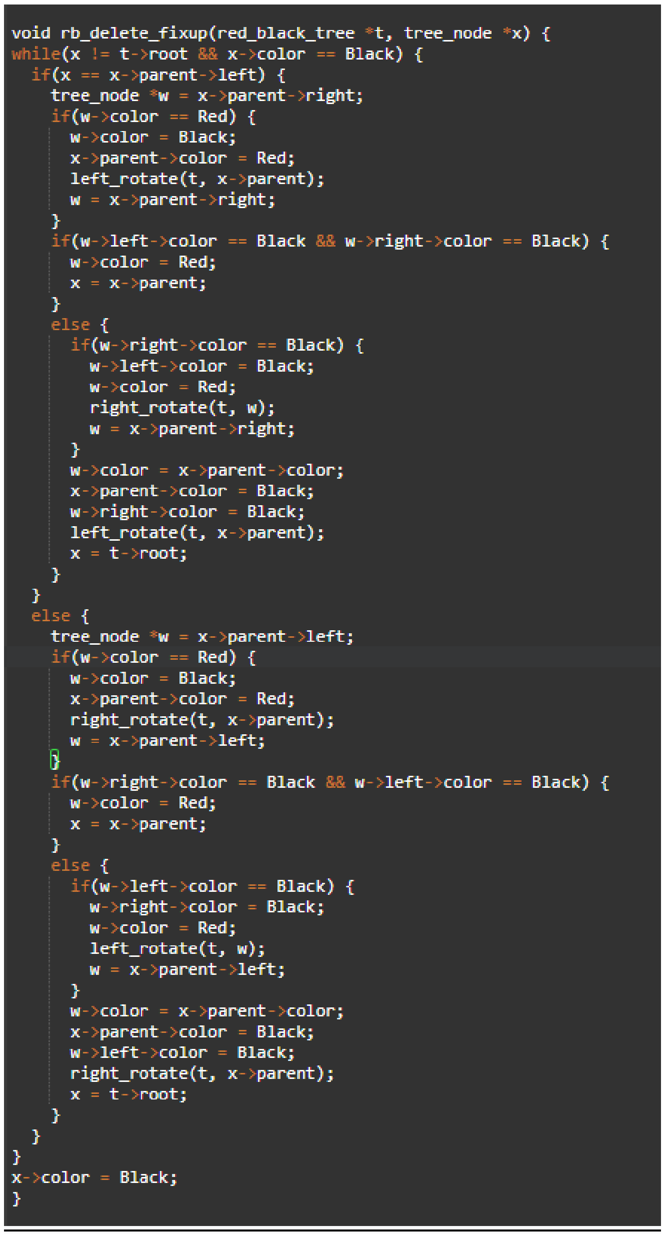
```
void rb_delete(red_black_tree *t, tree_node *z) {
tree_node *y = z;
tree_node *x;
enum COLOR y_orignal_color = y->color;
if(z->left == t->NIL) {
  x = z->right;
  rb_transplant(t, z, z->right);
}
else if(z->right == t->NIL) {
  x = z->left;
  rb_transplant(t, z, z->left);
}
else {
  y = minimum(t, z->right);
  y_orignal_color = y->color;
  x = y->right;
  if(y->parent == z) {
    x->parent = z;
  }
  else {
    rb_transplant(t, y, y->right);
    y->right = z->right;
    y->right->parent = y;
  }
  rb_transplant(t, z, y);
  y->left = z->left;
  y->left->parent = y;
  y->color = z->color;
}
if(y_orignal_color == Black)
  rb_delete_fixup(t, x);
}
```

Fig. 20: Code for delete operation



```
void rb_transplant(red_black_tree *t, tree_node *u, tree_node *v) {
if(u->parent == t->NIL)
  t->root = v;
else if(u == u->parent->left)
  u->parent->left = v;
else
  u->parent->right = v;
v->parent = u->parent;
}
```

Fig. 21: Code for transplant function used in delete operation



```
void rb_delete_fixup(red_black_tree *t, tree_node *x) {
while(x != t->root && x->color == Black) {
  if(x == x->parent->left) {
    tree_node *w = x->parent->right;
    if(w->color == Red) {
      w->color = Black;
      x->parent->color = Red;
      left_rotate(t, x->parent);
      w = x->parent->right;
    }
    if(w->left->color == Black && w->right->color == Black) {
      w->color = Red;
      x = x->parent;
    }
    else {
      if(w->right->color == Black) {
        w->left->color = Black;
        w->color = Red;
        right_rotate(t, w);
        w = x->parent->right;
      }
      w->color = x->parent->color;
      x->parent->color = Black;
      w->right->color = Black;
      left_rotate(t, x->parent);
      x = t->root;
    }
  }
  else {
    tree_node *w = x->parent->left;
    if(w->color == Red) {
      w->color = Black;
      x->parent->color = Red;
      right_rotate(t, x->parent);
      w = x->parent->left;
    }
    if(w->right->color == Black && w->left->color == Black) {
      w->color = Red;
      x = x->parent;
    }
    else {
      if(w->left->color == Black) {
        w->right->color = Black;
        w->color = Red;
        left_rotate(t, w);
        w = x->parent->left;
      }
      w->color = x->parent->color;
      x->parent->color = Black;
      w->left->color = Black;
      right_rotate(t, x->parent);
      x = t->root;
    }
  }
}
x->color = Black;
}
```

Fig. 22: Code for maintaining red black properties of tree after deletion

4.4 Search

This operation searches for a particular element by traversing through the red black tree. The search operation is especially efficient in red black trees due to their balancing property.

4.41 Search for an element

Algorithm

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if the item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right subtree.
5. Repeat this procedure recursively until a match is found.
6. If an element is not found then return NULL.

The flowchart and code for searching for an element in a red black tree is shown below in Fig. 23 and Fig. 24.

Flowchart

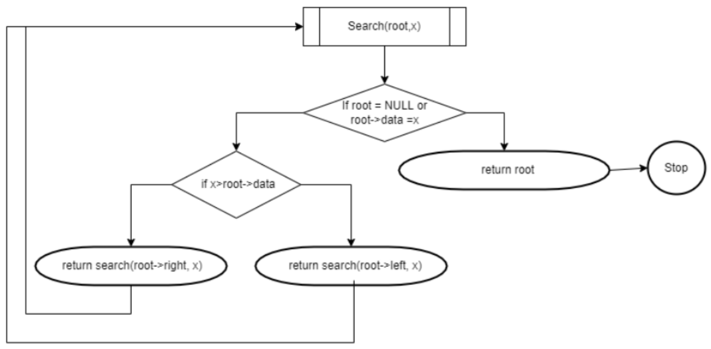
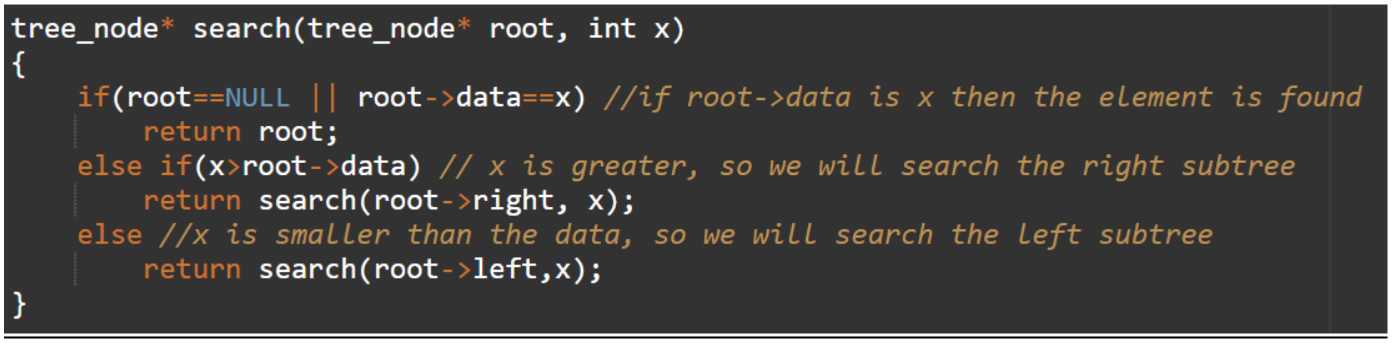

Search(root,x)
If root = NULL or root->data =x
return root
Stop
if x>root->data
return search(root->right, x)
return search(root->left, x)

Fig. 23: Flowchart for searching an element

Code



```
tree_node* search(tree_node* root, int x)
{
    if(root==NULL || root->data==x) //if root->data is x then the element is found
        return root;
    else if(x>root->data) // x is greater, so we will search the right subtree
        return search(root->right, x);
    else //x is smaller than the data, so we will search the left subtree
        return search(root->left,x);
}
```

Fig. 24: Code for search an element

4.42 Search for minimum element

Algorithm

1. Start from root node
2. Go to left child
3. Keep on repeating this process recursively until leftmost leaf node is found - this node has the minimum element

The flowchart and code for searching for the minimum element in a red black tree is shown below in Fig. 25 and Fig. 26.

Flowchart

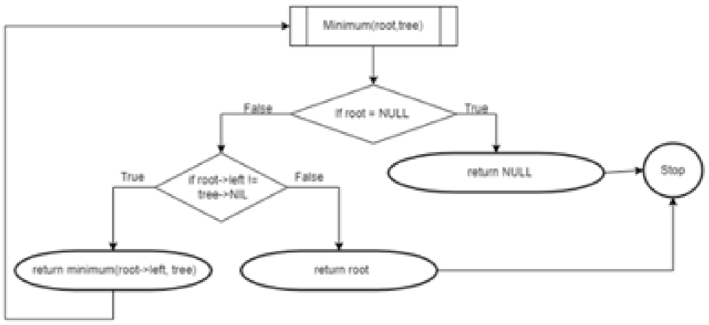
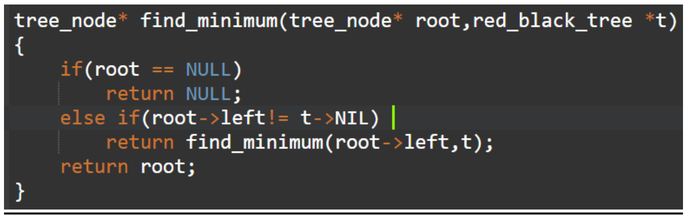


Fig. 25: Flowchart for searching minimum element

Code



```
tree_node* find_minimum(tree_node* root,red_black_tree *t)
{
    if(root == NULL)
        return NULL;
    else if(root->left!= t->NIL)
        return find_minimum(root->left,t);
    return root;
}
```

Fig. 26: Code for searching minimum element

4.43 Search for maximum element

Algorithm

1. Start from root node
2. Go to right child
3. Keep on repeating this process recursively until the rightmost leaf node is found - this node has the minimum element.

The flowchart and code for searching for the maximum element in a red black tree is shown below in Fig. 27 and Fig. 28.

Flowchart

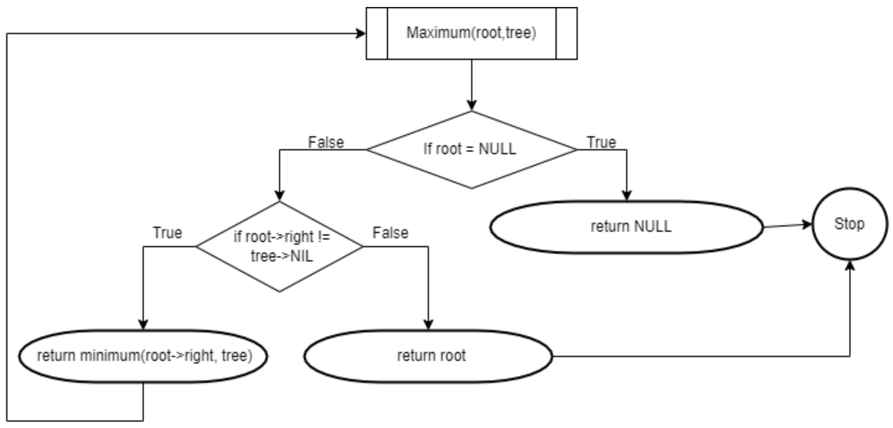
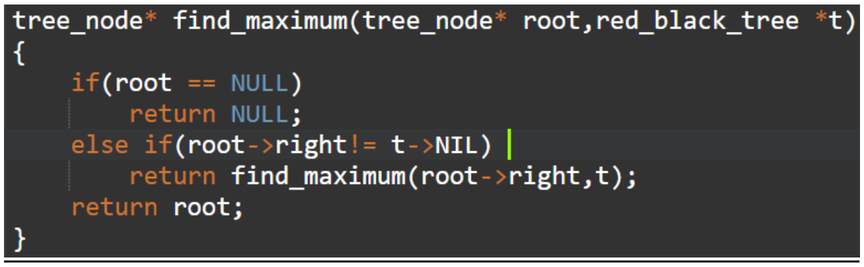


Fig. 27: Flowchart for searching maximum element

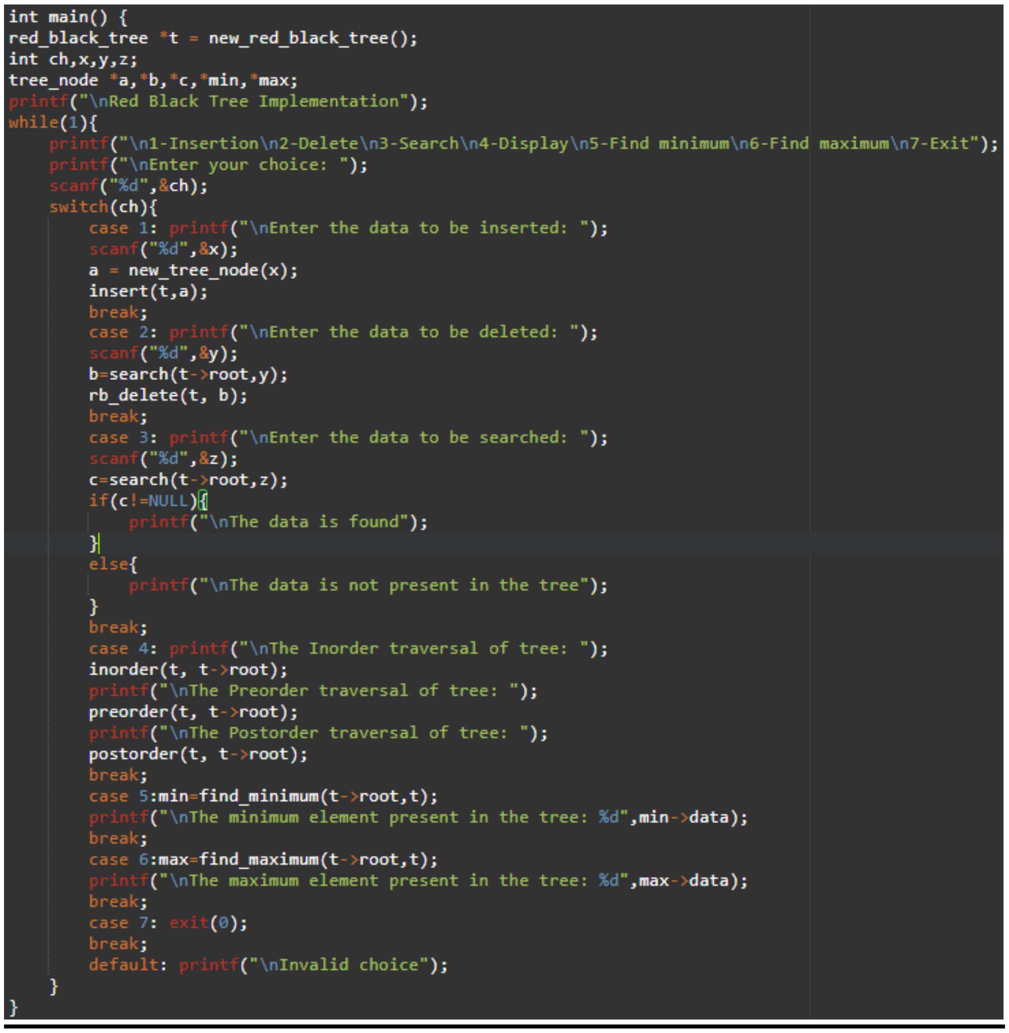
<u>Code</u>



```
tree_node* find_maximum(tree_node* root,red_black_tree *t)
{
    if(root == NULL)
        return NULL;
    else if(root->right!= t->NIL)
        return find_maximum(root->right,t);
    return root;
}
```

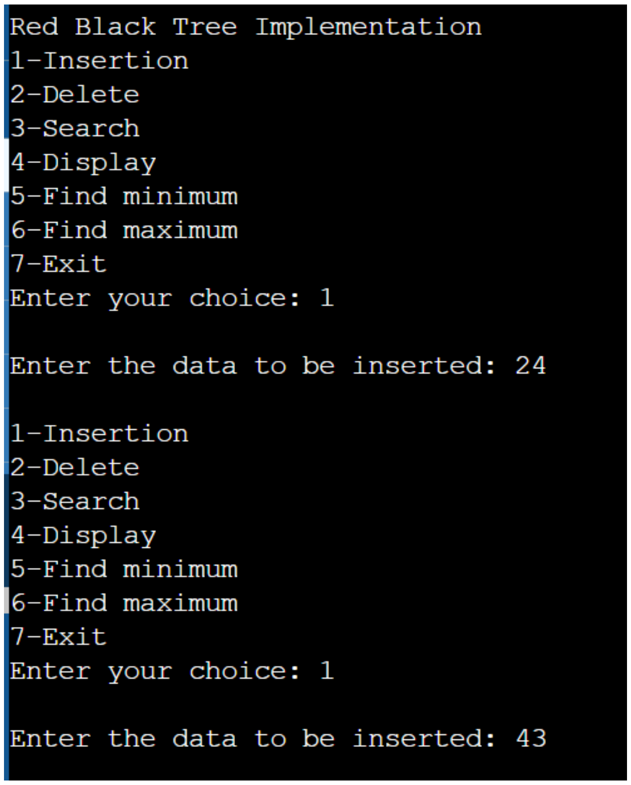
Fig. 28: Code for searching maximum element

5.Output of Implementation



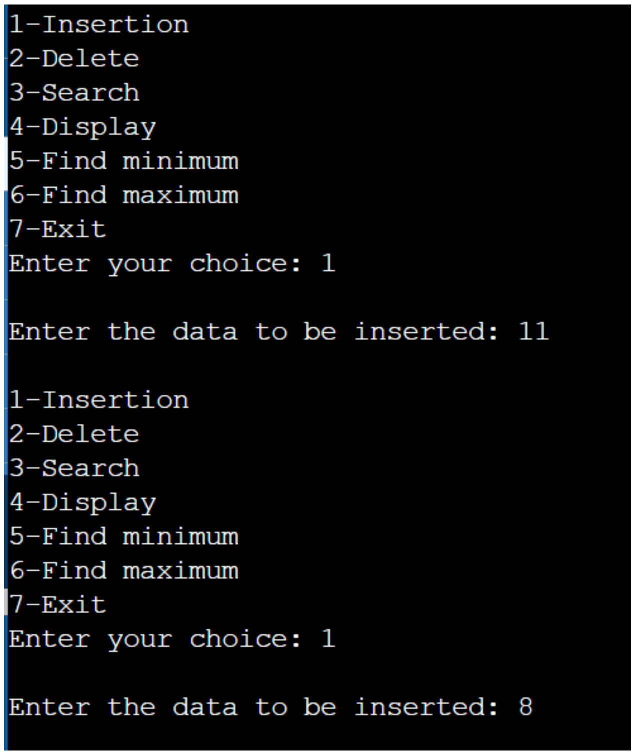
```c
int main() {
red_black_tree *t = new_red_black_tree();
int ch,x,y,z;
tree_node *a,*b,*c,*min,*max;
printf("\nRed Black Tree Implementation");
while(1){
    printf("\n1-Insertion\n2-Delete\n3-Search\n4-Display\n5-Find minimum\n6-Find maximum\n7-Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&ch);
    switch(ch){
        case 1: printf("\nEnter the data to be inserted: ");
        scanf("%d",&x);
        a = new_tree_node(x);
        insert(t,a);
        break;
        case 2: printf("\nEnter the data to be deleted: ");
        scanf("%d",&y);
        b=search(t->root,y);
        rb_delete(t, b);
        break;
        case 3: printf("\nEnter the data to be searched: ");
        scanf("%d",&z);
        c=search(t->root,z);
        if(c!=NULL){
            printf("\nThe data is found");
        }
        else{
            printf("\nThe data is not present in the tree");
        }
        break;
        case 4: printf("\nThe Inorder traversal of tree: ");
        inorder(t, t->root);
        printf("\nThe Preorder traversal of tree: ");
        preorder(t, t->root);
        printf("\nThe Postorder traversal of tree: ");
        postorder(t, t->root);
        break;
        case 5:min=find_minimum(t->root,t);
        printf("\nThe minimum element present in the tree: %d",min->data);
        break;
        case 6:max=find_maximum(t->root,t);
        printf("\nThe maximum element present in the tree: %d",max->data);
        break;
        case 7: exit(0);
        break;
        default: printf("\nInvalid choice");
    }
}
```

Fig. 29: The main functions with various choices shown



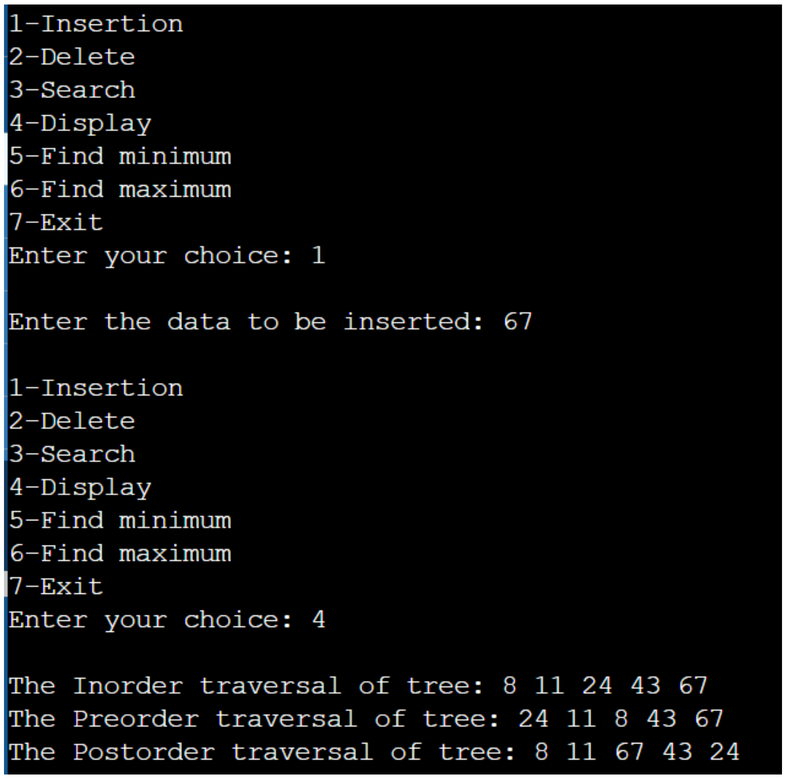
```
Red Black Tree Implementation
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 1

Enter the data to be inserted: 24

1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 1

Enter the data to be inserted: 43
```

Fig. 30: 24 and 43 has been inserted



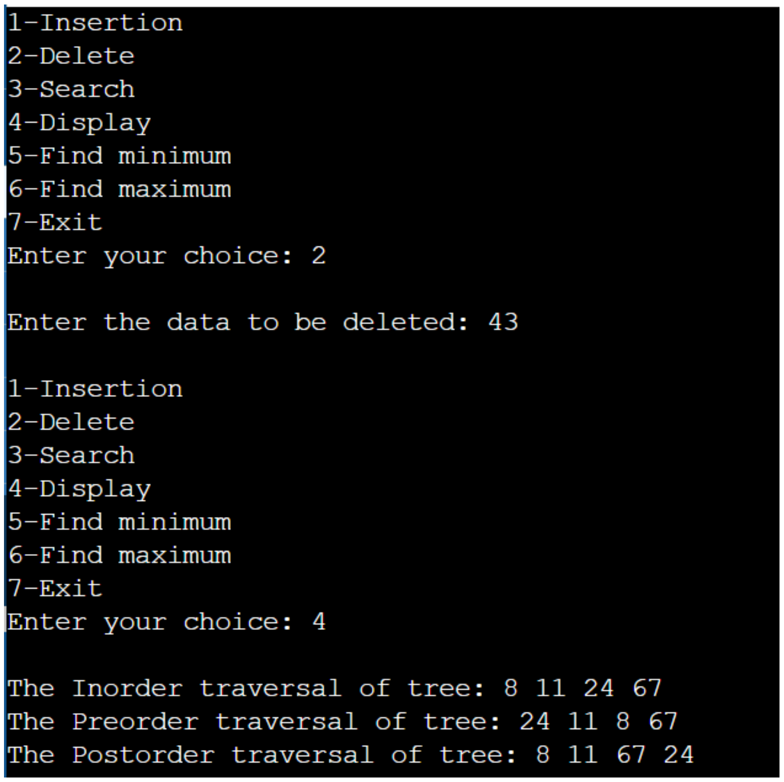
```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 1

Enter the data to be inserted: 11

1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 1

Enter the data to be inserted: 8
```

Fig. 31: 11 and 8 has been inserted



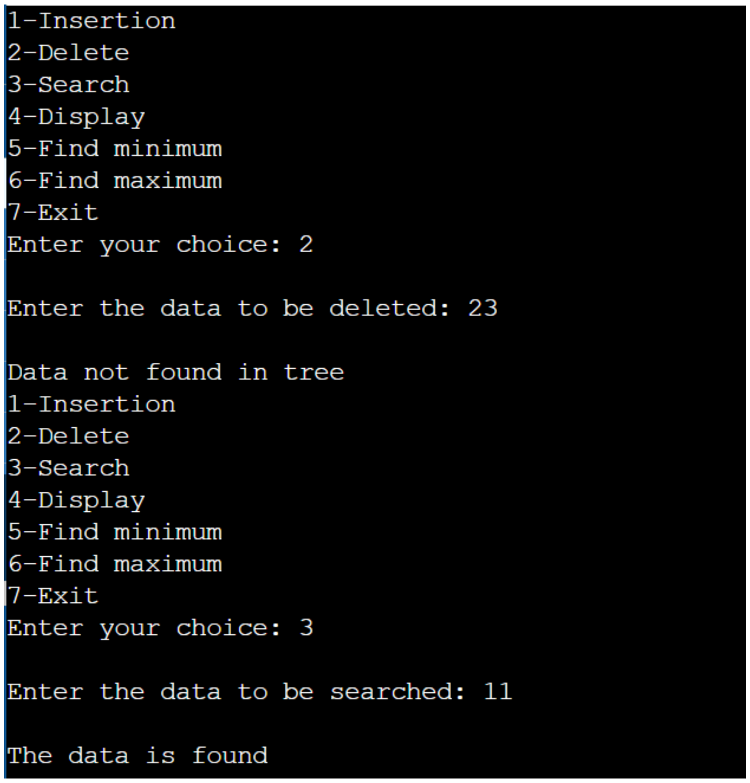
```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 1

Enter the data to be inserted: 67

1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 4

The Inorder traversal of tree: 8 11 24 43 67
The Preorder traversal of tree: 24 11 8 43 67
The Postorder traversal of tree: 8 11 67 43 24
```

Fig. 32: 67 has been inserted and the tree has been displayed (inorder, preorder, postorder)



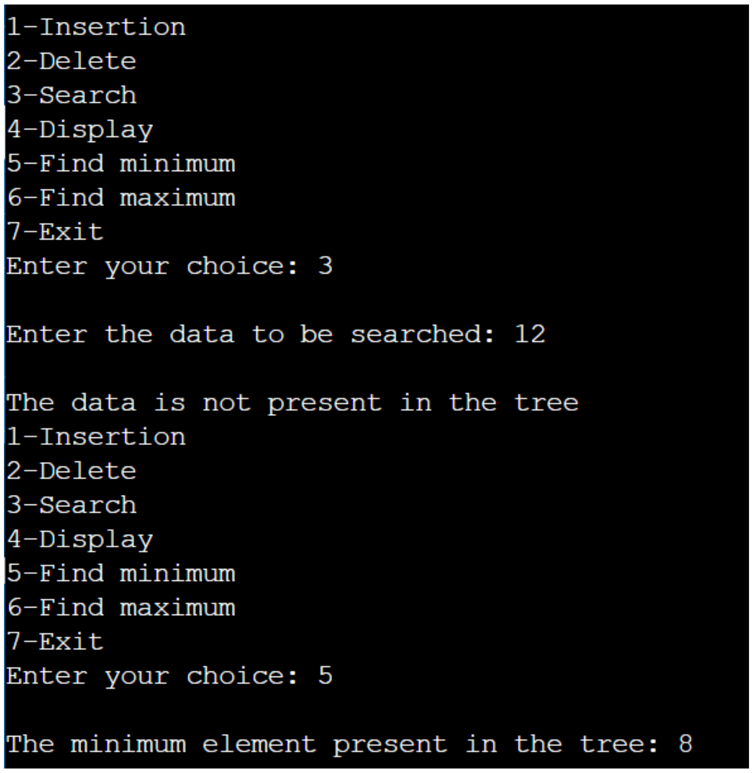
```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 2

Enter the data to be deleted: 43

1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 4

The Inorder traversal of tree: 8 11 24 67
The Preorder traversal of tree: 24 11 8 67
The Postorder traversal of tree: 8 11 67 24
```

Fig. 33: Node 43 has been deleted and the tree has been displayed



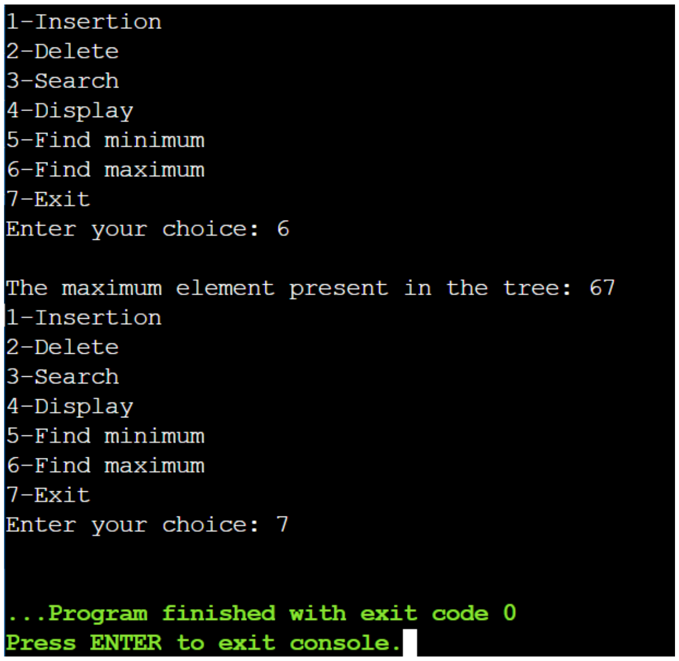
```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 2

Enter the data to be deleted: 23

Data not found in tree
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 3

Enter the data to be searched: 11

The data is found
```

Fig. 34: Implementation of choice 2 and choice 3



```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 3

Enter the data to be searched: 12

The data is not present in the tree
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 5

The minimum element present in the tree: 8
```

Fig. 35: Implementation of choice 3(Searching an element) and choice 5(Finding minimum)



```
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 6

The maximum element present in the tree: 67
1-Insertion
2-Delete
3-Search
4-Display
5-Find minimum
6-Find maximum
7-Exit
Enter your choice: 7


...Program finished with exit code 0
Press ENTER to exit console.
```

Fig. 36: Implementation of choice 6(Finding maximum) and choice 7(Exit)

6. CONCLUSION

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. But Red Black Trees are relatively complicated to implement because of all the operation edge cases.

7.REFERENCES

[1] https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/?ref=lbp

[2] https://www.programiz.com/dsa/red-black-tree

[3] https://www.codesdope.com/course/data-structures-red-black-trees-deletion/

[4] https://secondboyet.com/Articles/RedBlack4.html

[5] https://www.baeldung.com/cs/red-black-trees

[6] https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

[7] Data Structures and Algorithms analysis in C++ - Mark Allen Weiss

[8] Data structures using C and C++ - Aaron M. Tenenbaum

[9] Data Structures and Algorithms made easy – Narasimha Karumanchi