

# DEPARTMENT OF INFORMATION TECHNOLOGY

**COURSE CODE: DJ19ITL502 DATE:02/11/2022**

**COURSE NAME: Advanced Data Structures Laboratory CLASS/DIV: A3**

# CO/LO: LO1

**EXPERIMENT NO. 3**

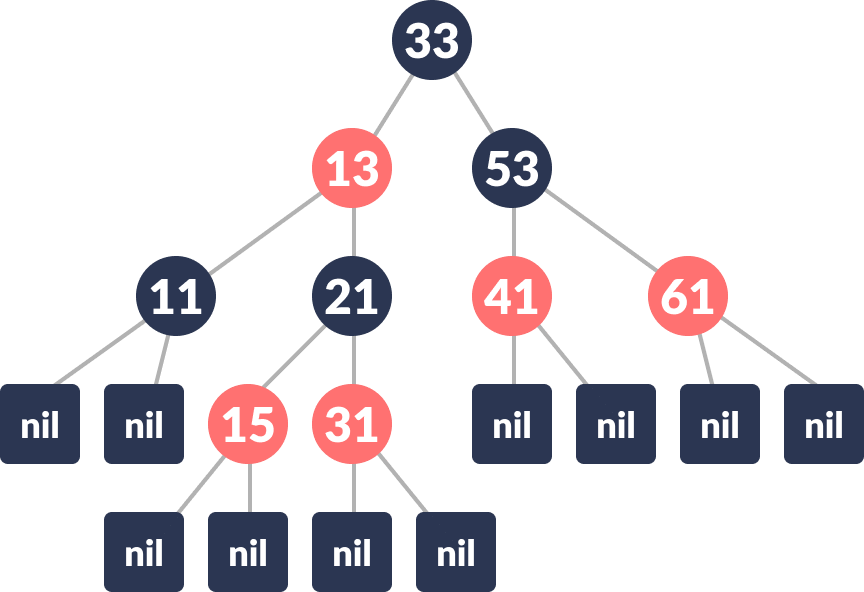
## AIM: To implement insertion and deletion in Red Black Tree

**THEORY:** Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

**A red-black tree satisfies the following properties:**

1. Red/Black Property: Every node is colored, either red or black.
2. Root Property: The root is black.
3. Leaf Property: Every leaf (NIL) is black.
4. Red Property: If a red node has children then, the children are always black.
5. Depth Property: For each node, any simple path from this node to any of itsdescendant leaf has the same black-depth (the number of black nodes).

**An example of a red-black tree is:**



**Red Black Tree**

## Each node has the following attributes:

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

How the red-black tree maintains the property of self- balancing?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.

## Various operations that can be performed on a red-black tree are: Rotating the sub trees in a Red-Black Tree

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 two types of rotations:**

## Left Rotate

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

## Algorithm

1. **Let the initial tree be: Initial tree**

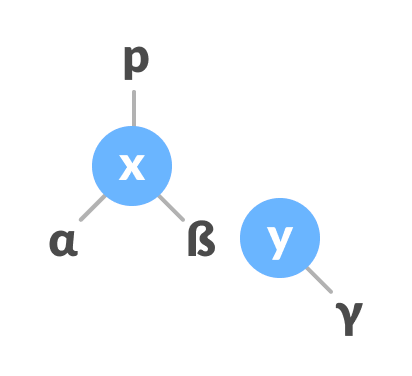
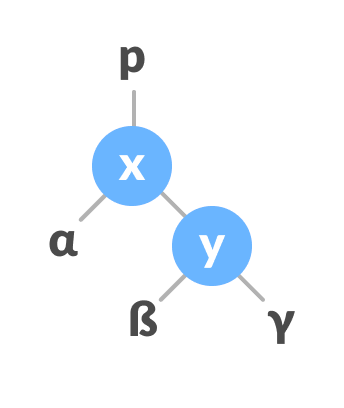
**x**

## If

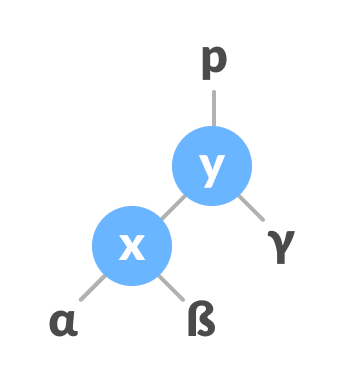
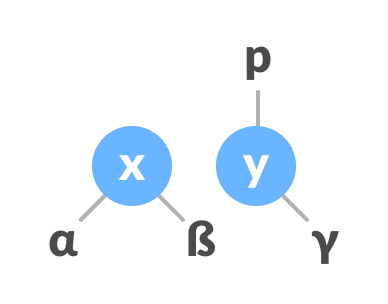
**y**

**has a left subtree, assign**

## as the parent of the left subtree of y.



**Assign x as the parent of the left subtree of y**



**3. If the parent of is NULL, make as the root of the tree.**

**4. Else if is the left child of p, make as the left child of p.**

**5. Else assign as the right child of p.**

**Change the parent of x to**

**that of y**

**6. Make as the parent of .**

**Assign y as the parent of x.**

**x**

**y**

**y**

**y**

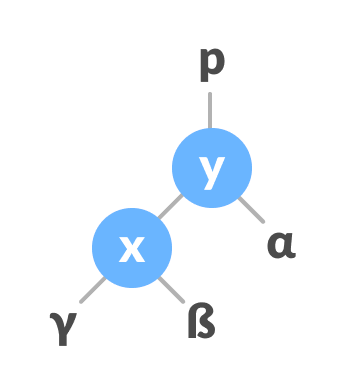
**x**

**y**

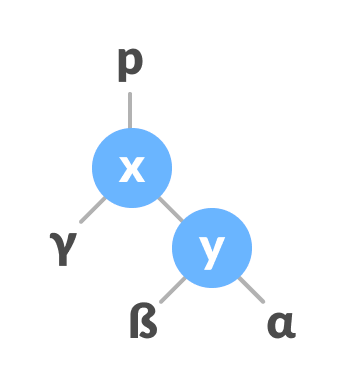
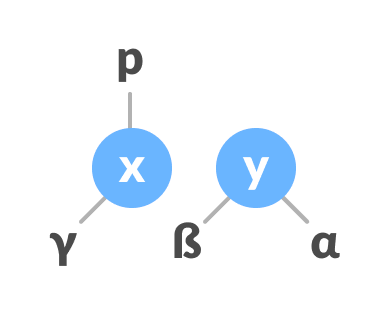
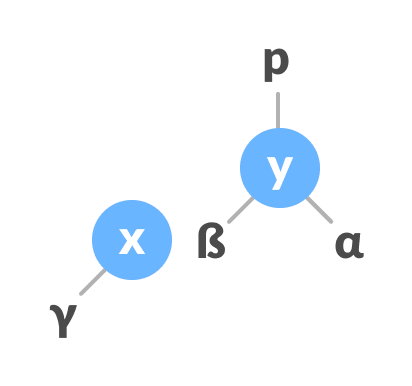
**x**

**Right Rotate**

**In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.**



**1. Let the initial tree be: Initial Tree**



**2. If has a right subtree, assign y as the parent of the right subtree of x.**

**Assign y as the parent of the right subtree of x**

**3. If the parent of is NULL, make as the root of the tree.**

**4. Else if is the right child of its parent p, make as the right child of p.**

**5. Else assign as the left child of p.**

**Assign the parent of y as**

**the parent of x**

**6. Make as the parent of .**

**Assign x as the parent of y**

**y**

**x**

**x**

**x**

**y**

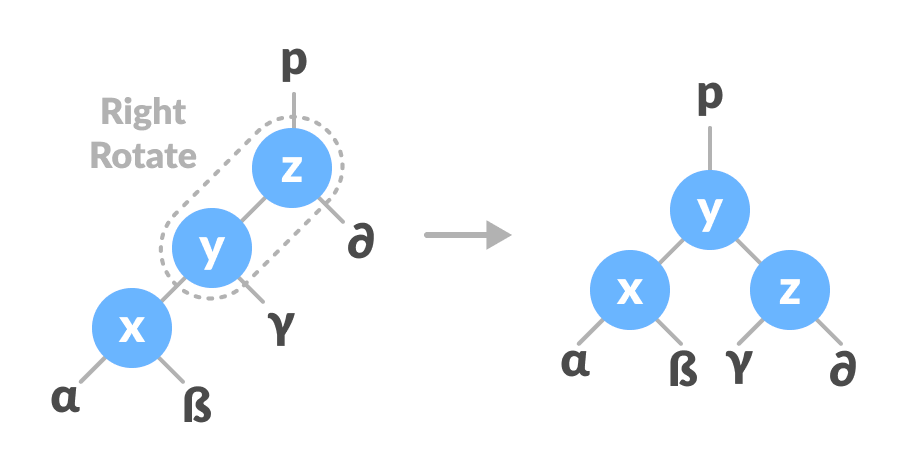
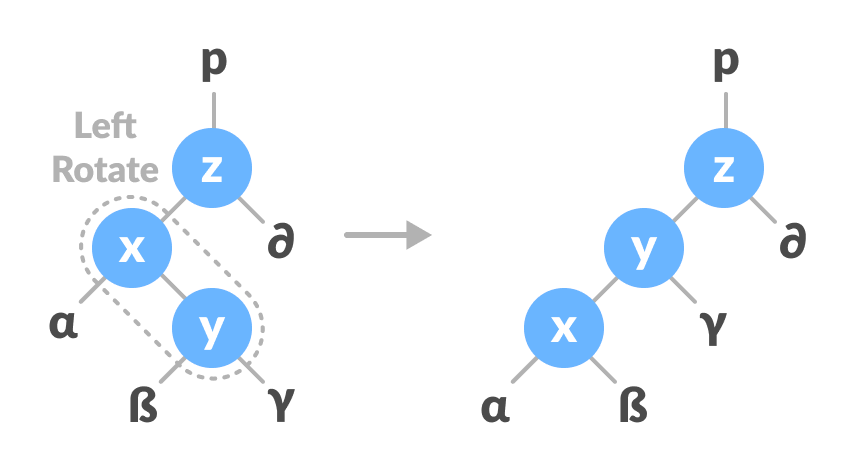
**x**

**y**

**x**

**Left-Right and Right-Left Rotate**

**In left-right rotation, the arrangements are first shifted to the left and then to the right.**



**1. Do left rotation on x-y.**

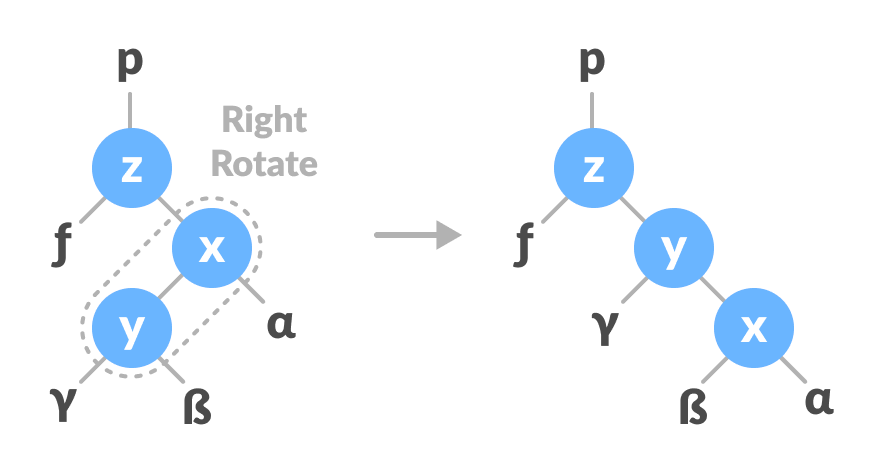
**rotate x-y**

**Left**

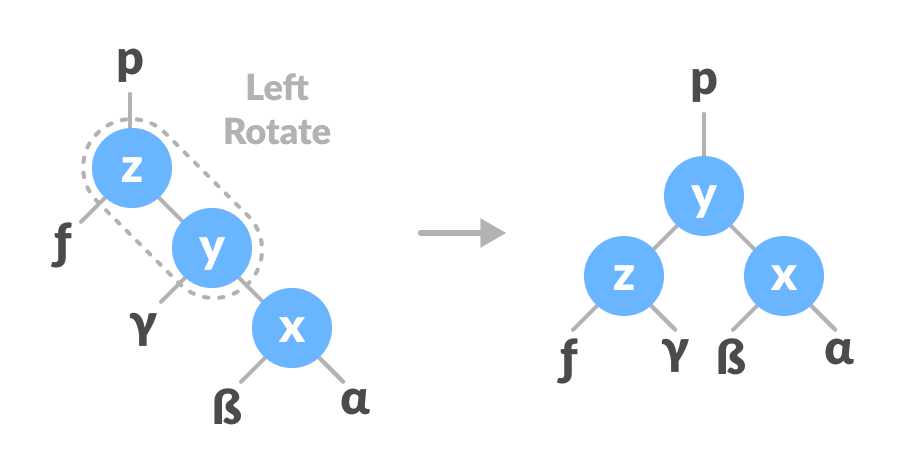
**2. Do right rotation on y-z.**

**Right rotate z-y**

**In right-left rotation, the arrangements are first shifted to the right and then to the left.**



**1. Do right rotation on x-y. Right rotate x-y**



**2. Do left rotation on z-y.**

**rotate z-y**

**Left**

**Inserting an element into a Red-Black Tree**

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.

1. Recolor
2. Rotation

**Algorithm to insert a node**

**Following steps are followed for inserting a new element into a red-black tree:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1. Let y be the leaf (ie. NIL) and x be the root of the tree.** | | | | |
| **2. Check if the tree is empty (ie. whether** |  | **x is NIL). If yes, inse** | **rt newNod** | **e as a** |

## rootnode and color it black.

1. **Else, repeat steps following steps until leaf ( NIL) is reached.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a. Compare** | **newKey** | **with** | **rootKey** | **.** |



**b. If**

**is greater than rootKey, traverse through the right subtree.**

**c. Else traverse through the left subtree.**

**newKey**

**4.**

**5.**

**Assign the parent of the leaf as a parent of newNode.**

**If**

**is greater than newKey, make**

**as rightChild.**

**6. Else, make**

**as leftChild.**

**7. Assign**

**to the left and**

**of newNode.**

**8. Assign RED color to newNode.**

**9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.**

**rightChild**

**NULL**

**newNode**

**newNode**

**leafKey**

**Why newly inserted nodes are always red in a red-black tree?**

**This is 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 to maintain red-black property after insertion**

**This algorithm is used for maintaining the property of a red-black tree if the insertion of anewNode violates this property.**

**1. Do the following while the parent of newNode p is RED.**

**2. If is the left child of**

**gP of z, do the following.**

**Case-I:**

**grandParent**

**p**



1. **If the color of the right child of gP of is RED, set the color of both the**

**children of gP as BLACK and the color of gP as RED.**

1. **Assign gP to newNode.**

**Case-II:**

**c. Else if**

**is the right child of then, assign to newNode.**

**d. Left-Rotate newNode.**

**Case-III:**

**e.**

**f.**

**Set color of**

**as BLACK and color of gP as RED.**

**Right-Rotate gP.**

**3. Else, do the following.**

**p**

**p**

**p**

**newNode**

**z**

1. **If the color of the left child of gP of is RED, set the color of both the**

**children of gP as BLACK and the color of gP as RED.**

1. **Assign gP to newNode.**

**c.**

**Else if newNode is the left child of**

**Rotate newNode.**

**then, assign to**

**and Right-**

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

**e. Left-Rotate gP.**

**p**

**newNode**

**p**

**p**

**z**

**4. Set the root of the tree as BLACK.**

**Deleting an element from a Red-Black Tree**

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



**Algorithm to delete a node**

1. **Save the color of**
2. **If the left child of**

**in origrinalColor.**

**is**

**a. Assign the right child of**

**to x.**

**b. Transplant**

**3. Else if the right child of**

**with x.**

**is**

1. **Assign the left child of**
2. **Transplant**

**into x.**

**with x.**

**4. Else**

**nodeToBeDeleted**

**nodeToBeDeleted**

**NULL**

**nodeToBeDeleted**

**nodeToBeDeleted**

**nodeToBeDeleted**

**NULL**

**nodeToBeDeleted**

**nodeToBeDeleted**

**a.**

**b.**

**Assign the minimum of right subtree of**

**into y.**

**Save the color of**

**in originalColor.**

**c. Assign the**

**of into x.**

**d. If is a child of nodeToBeDeleted, then set the parent of as y.**

**e. Else, transplant with**

**of y.**

**f. Transplant**

**with y.**

**g. Set the color of y with originalColor.**

**nodeToBeDeleted**

**rightChild**

**y**

**x**

**y**

**y**

**rightChild**

**y**

**noteToBeDeleted**

|  |  |  |
| --- | --- | --- |
| **5. If the** |  | **is BLACK, call DeleteFix(x).** |
| **originalColor** |

**Algorithm to maintain Red-Black property after deletion**

**This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.**



**This violation is corrected by assuming that node (which is occupying y's original position) has an extra black. This makes node neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.**

**However, the color attribute of is not changed rather the extra black is represented in x's pointing to the node.**

**The extra black can be removed if**

**x**

**x**

**x**

1. **It reaches the root node.**
2. **If points to a red-black node. In this case, is colored black.**
3. **Suitable rotations and recoloring are performed.**

**x**

**x**

**The following algorithm retains the properties of a red-black tree.**

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

**b. Set the color of the parent of as RED.**

**c. Left-Rotate the parent of x.**

**d. Assign the**

**of the parent of to .**

**w**

**x**

**rightChild**

**x**

**x**

**x**

**x**

**x**

**x**

**c. If the color of both the right and the**

**of is BLACK,**

**Case-II:**

**w**

**leftChild**

1. **Set the color of w as RED**
2. **Assign the parent of x to x.**

**a. Set the color of the of w as BLACK**

**leftChild**

## Set the color of as RED

**w**



**d. Else if the color of the**

**of is BLACK**

**Case-III:**

**w**

**rightChild**

* 1. **Right-Rotate w.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **d. 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:**

|  |  |  |
| --- | --- | --- |
| 1. **Set the color of w as the color of the parent of x.** 2. **Set the color of the parent of x as BLACK.** 3. **Set the color of the right child of w as BLACK.** 4. **Left-Rotate the parent of x.** | | |
| **e. 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.** |

# PROGRAM:

#include<stdio.h> #include<conio.h> #include<stdlib.h>

enum nodeColor {

RED, BLACK



};

struct rbNode { int data, color;

struct rbNode \*link[2];

};

struct rbNode \*root = NULL;

struct rbNode \*createNode(int data) { struct rbNode \*newnode;

newnode = (struct rbNode \*)malloc(sizeof(struct rbNode)); newnode->data = data;

newnode->color = RED;

newnode->link[0] = newnode->link[1] = NULL; return newnode;

}

void insertion(int data) {



struct rbNode \*stack[98], \*ptr, \*newnode, \*xPtr, \*yPtr; int dir[98], ht = 0, index;

ptr = root; if (!root) {

root = createNode(data); return;

}

stack[ht] = root; dir[ht++] = 0;

while (ptr != NULL) {

if (ptr->data == data) { printf("Duplicates Not Allowed!!\n"); return;

}

index = (data - ptr->data) > 0 ? 1 : 0; stack[ht] = ptr;

ptr = ptr->link[index]; dir[ht++] = index;

}



stack[ht - 1]->link[index] = newnode = createNode(data); while ((ht >= 3) && (stack[ht - 1]->color == RED)) {

if (dir[ht - 2] == 0) {

yPtr = stack[ht - 2]->link[1];

if (yPtr != NULL && yPtr->color == RED) { stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK; ht = ht - 2;

} else {

if (dir[ht - 1] == 0) { yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1]; yPtr = xPtr->link[1];

xPtr->link[1] = yPtr->link[0]; yPtr->link[0] = xPtr; stack[ht

- 2]->link[0] = yPtr;

}

xPtr = stack[ht - 2]; xPtr->color = RED;

yPtr->color = BLACK;



xPtr->link[0] = yPtr->link[1]; yPtr->link[1] = xPtr;

if (xPtr == root) { root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

} else {

yPtr = stack[ht - 2]->link[0];

if ((yPtr != NULL) && (yPtr->color == RED)) { stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK; ht = ht - 2;

} else {

if (dir[ht - 1] == 1) { yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[0];



xPtr->link[0] = yPtr->link[1]; yPtr->link[1] = xPtr; stack[ht

- 2]->link[1] = yPtr;

}

xPtr = stack[ht - 2]; yPtr->color = BLACK; xPtr->color = RED;

xPtr->link[1] = yPtr->link[0]; yPtr->link[0] = xPtr;

if (xPtr == root) { root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

}

}

root->color = BLACK;

}

void deletion(int data) {

struct rbNode \*stack[98], \*ptr, \*xPtr, \*yPtr; struct rbNode \*pPtr, \*qPtr, \*rPtr;

int dir[98], ht = 0, diff, i; enum nodeColor color;

if (!root) {

printf("Tree not available\n"); return;

}

ptr = root;

while (ptr != NULL) {

if ((data - ptr->data) == 0) break;

diff = (data - ptr->data) > 0 ? 1 : 0; stack[ht] = ptr;

dir[ht++] = diff;

ptr = ptr->link[diff];

}

if (ptr->link[1] == NULL) {

if ((ptr == root) && (ptr->link[0] == NULL)) { free(ptr);

root = NULL;

} else if (ptr == root) { root = ptr->link[0]; free(ptr);

} else {

stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];

}

} else {

xPtr = ptr->link[1];

if (xPtr->link[0] == NULL) { xPtr->link[0] = ptr->link[0]; color = xPtr->color;

xPtr->color = ptr->color; ptr->color = color;

if (ptr == root) { root = xPtr;

} else {



stack[ht - 1]->link[dir[ht - 1]] = xPtr;

}

dir[ht] = 1; stack[ht++] = xPtr;

} else {

i = ht++; while (1) {

dir[ht] = 0; stack[ht++] = xPtr; yPtr = xPtr->link[0]; if (!yPtr->link[0]) break;

xPtr = yPtr;

}

dir[i] = 1; stack[i] = yPtr; if (i > 0)

stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];



xPtr->link[0] = yPtr->link[1]; yPtr->link[1] = ptr->link[1];

if (ptr == root) { root = yPtr;

}

color = yPtr->color;

yPtr->color = ptr->color; ptr->color = color;

}

}

if (ht < 1) return;

if (ptr->color == BLACK) { while (1) {

pPtr = stack[ht - 1]->link[dir[ht - 1]]; if (pPtr && pPtr->color == RED) { pPtr->color = BLACK;



break;

}

if (ht < 2) break;

if (dir[ht - 2] == 0) {

rPtr = stack[ht - 1]->link[1];

if (!rPtr) break;

if (rPtr->color == RED) { stack[ht - 1]->color = RED; rPtr->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0]; rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) { root = rPtr;



} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 0;

stack[ht] = stack[ht - 1]; stack[ht - 1] = rPtr; ht++;

rPtr = stack[ht - 1]->link[1];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) && (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) { rPtr->color = RED;

} else {

if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) { qPtr = rPtr->link[0];

rPtr->color = RED; qPtr->color = BLACK;

rPtr->link[0] = qPtr->link[1]; qPtr->link[1] = rPtr;



rPtr = stack[ht - 1]->link[1] = qPtr;

}

rPtr->color = stack[ht - 1]->color; stack[ht - 1]->color = BLACK;

rPtr->link[1]->color = BLACK; stack[ht - 1]->link[1] = rPtr->link[0]; rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) { root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

} else {

rPtr = stack[ht - 1]->link[0]; if (!rPtr)

break;

if (rPtr->color == RED) { stack[ht - 1]->color = RED; rPtr->color = BLACK;



stack[ht - 1]->link[0] = rPtr->link[1]; rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) { root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 1;

stack[ht] = stack[ht - 1]; stack[ht - 1] = rPtr; ht++;

rPtr = stack[ht - 1]->link[0];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) && (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) { rPtr->color = RED;

} else {



if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) { qPtr = rPtr->link[1];

rPtr->color = RED; qPtr->color = BLACK;

rPtr->link[1] = qPtr->link[0]; qPtr->link[0] = rPtr;

rPtr = stack[ht - 1]->link[0] = qPtr;

}

rPtr->color = stack[ht - 1]->color; stack[ht - 1]->color = BLACK;

rPtr->link[0]->color = BLACK; stack[ht - 1]->link[0] = rPtr->link[1]; rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) { root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

}



ht--;

}

}

}

void traversal(struct rbNode \*node) { if (node) {

traversal(node->link[0]); printf("%d ", node->data); traversal(node->link[1]);

}

return;

}

void main(){ int ch,data; while(1){

printf("1.Insertion\n2.Deletion\n3.Traverse\n4.Exit\nEnter your choice:"); scanf("%d",&ch);

switch(ch){

case 1:printf("Enter the element to be inserted:"); scanf("%d",&data);



insertion(data); break;

case 2:printf("Enter the element to be deleted:"); scanf("%d",&data);

deletion(data); break;

case 3:traversal(root); printf("\n");

break;

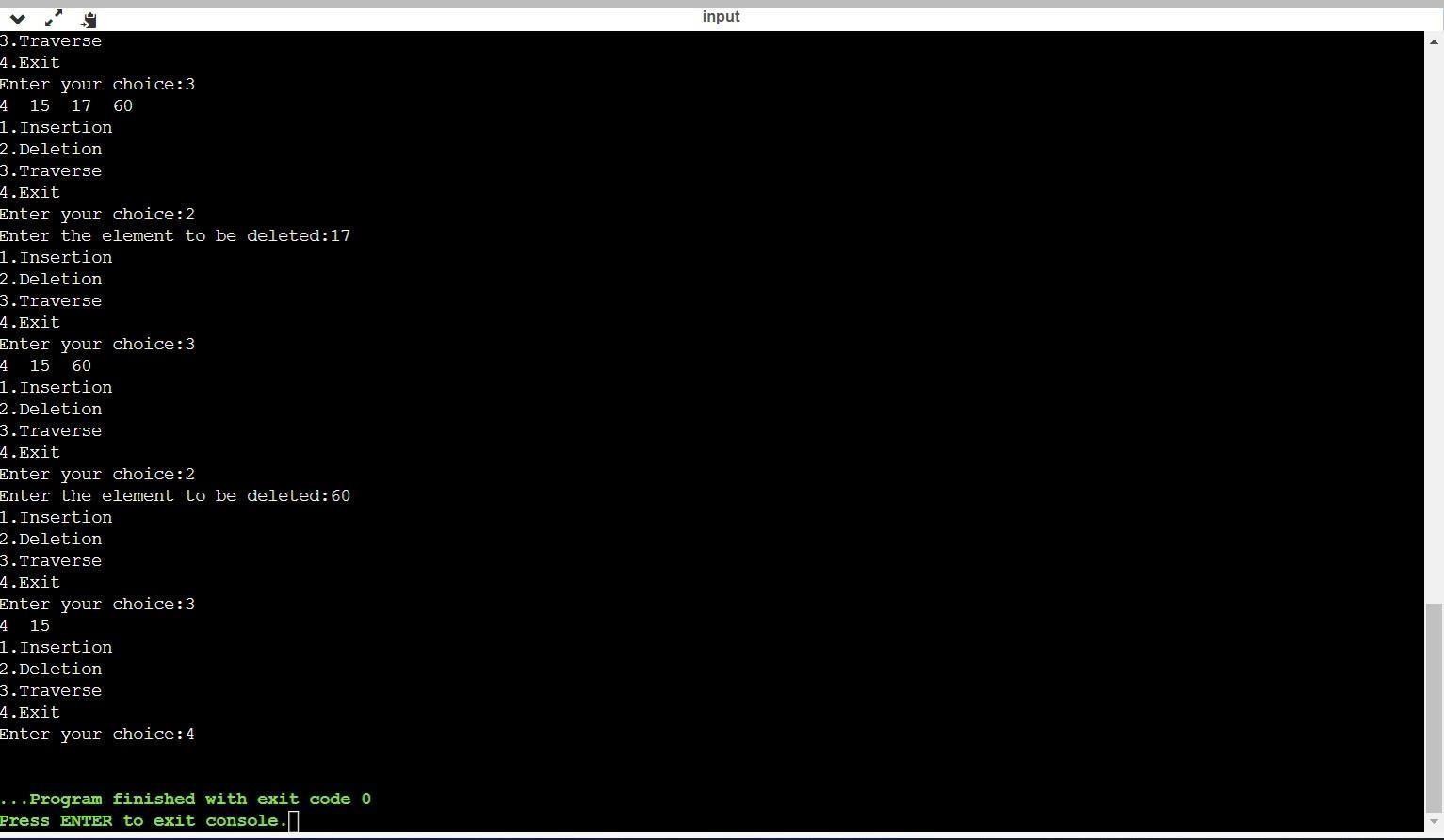
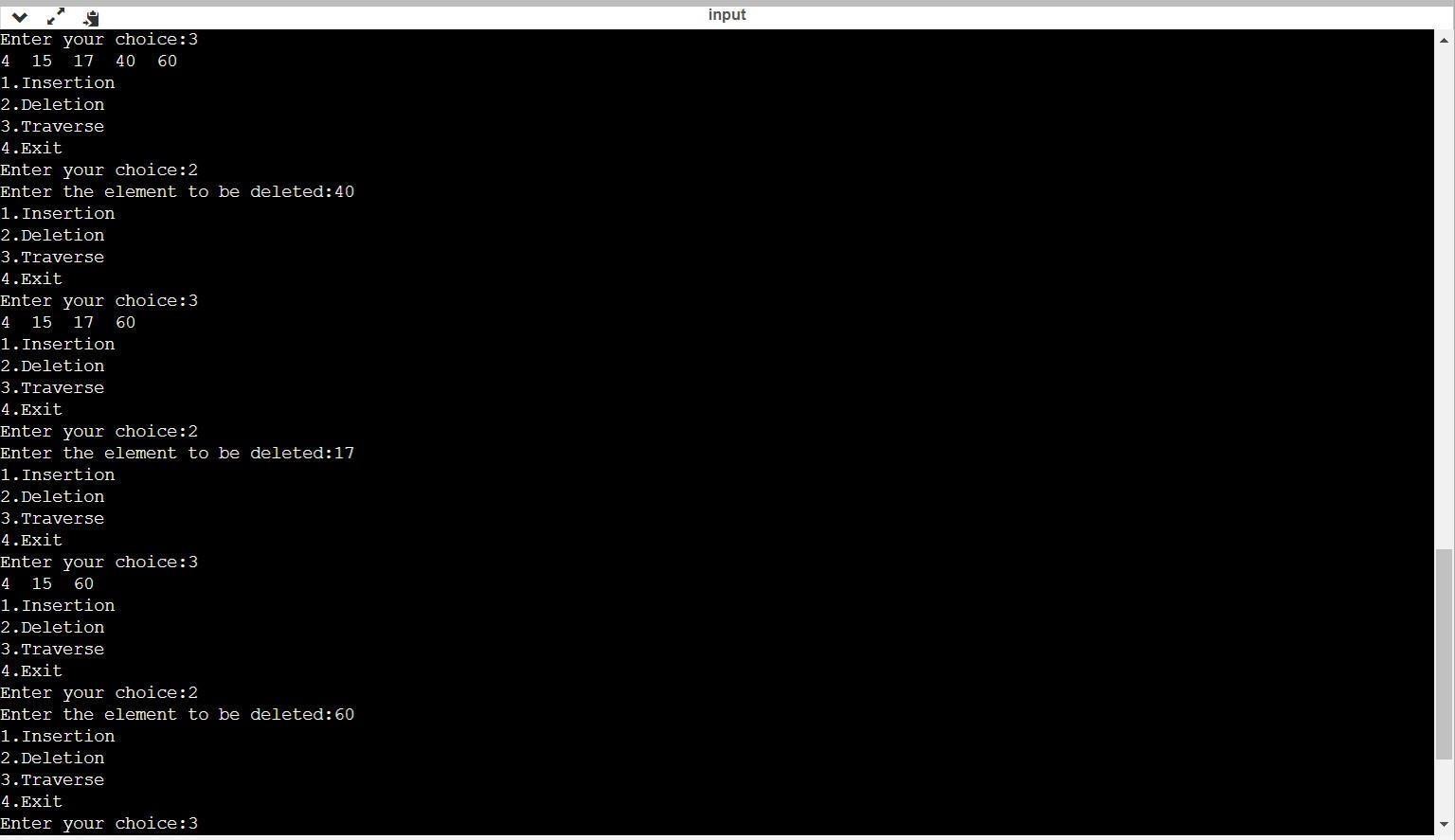
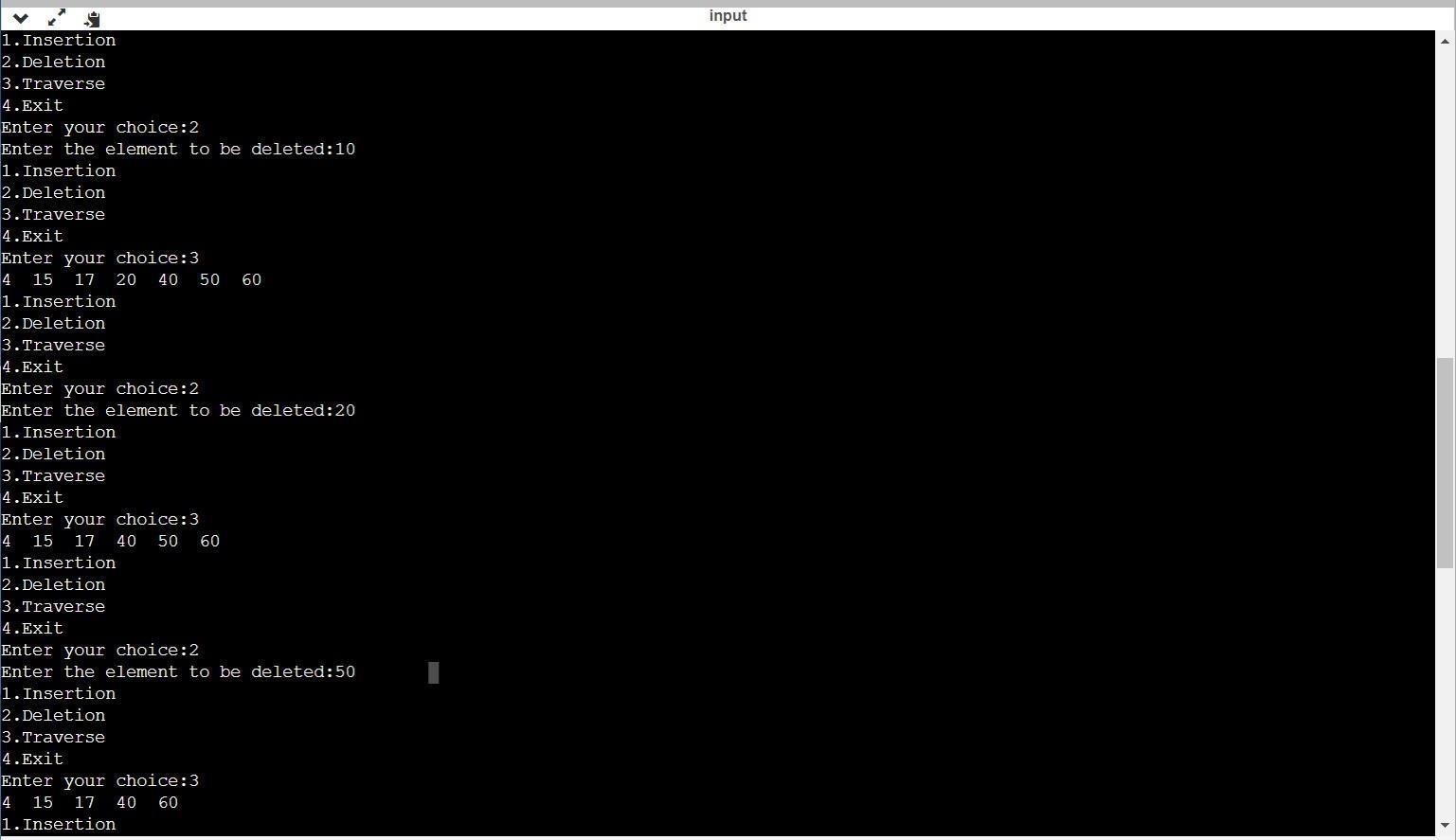
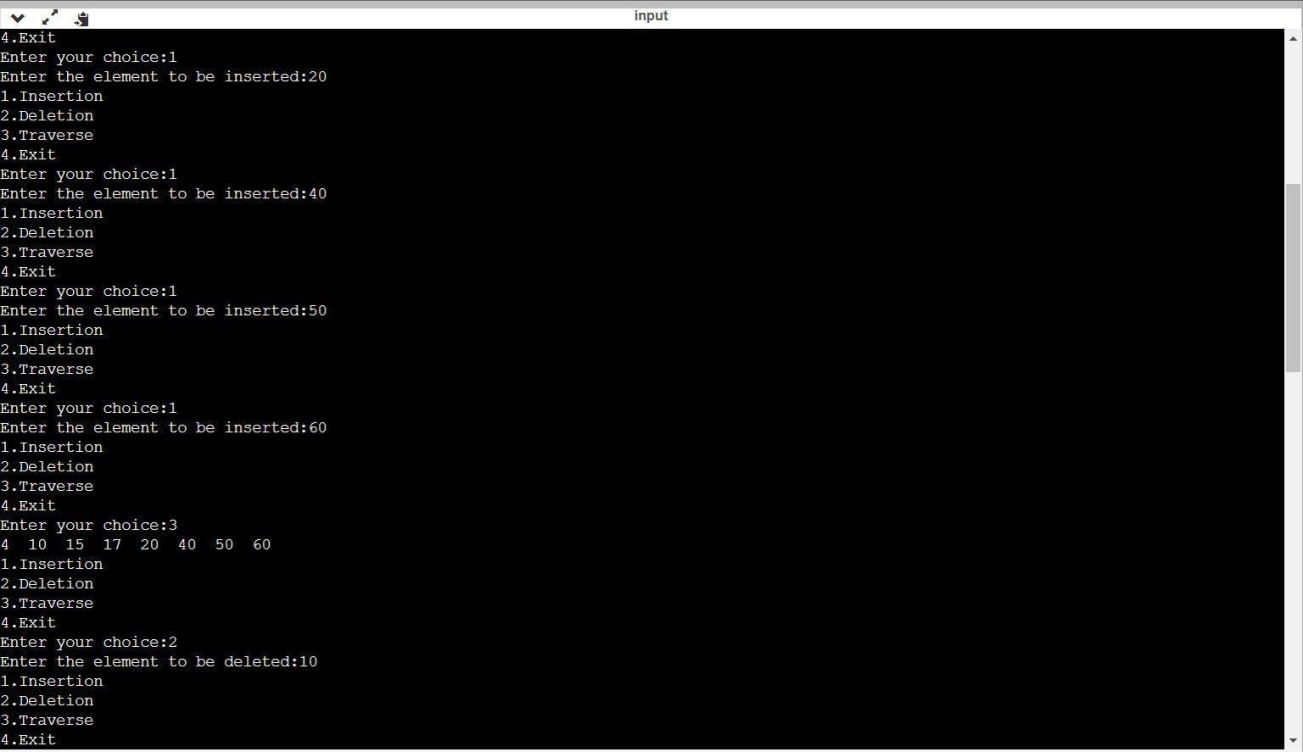
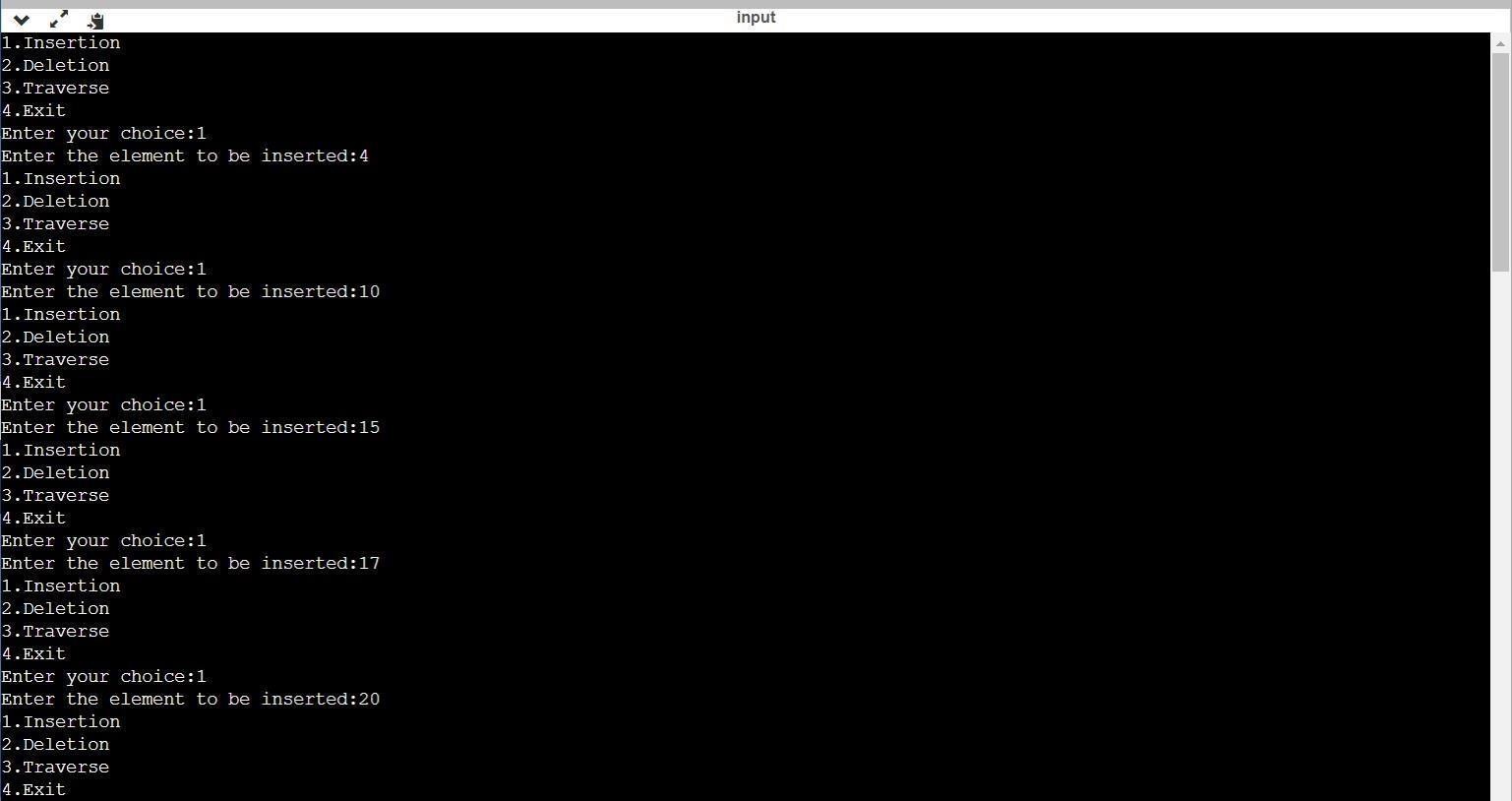
case 4:exit(0); default:printf("Invalid choice\n");

}

}

}

# OUTPUT:



**CONCLUSION**: Hence we have successfully implemented insertion and deletion in Red Black Tree.