NAME: MRUDHULA

REGNO: 192372290

//1.RED BLACK TREE:

```
// Implementing Red-Black Tree in C

#include <stdio.h>
#include <stdlib.h>
enum nodeColor {
  RED,
  BLACK
};
struct rbNode {
  int data, color;
  struct rbNode *link[2];
};
struct rbNode *root = NULL;
// Create a red-black tree
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;
}
// Insert an node
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;
}
```

```
// Delete a node
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--;
   }
  }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
```

```
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
   printf("\n");
  }
 return 0;
}
```

//OUTPUT:

```
1. Insertion  2. Deletion
3. Traverse   4. Exit
Enter your choice:1
```

Enter the element to insert:1

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:1

Enter the element to insert:2

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:1

Enter the element to insert:3

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:3

1 2 3

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:2

Enter the element to delete:2

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:3

1 3

1. Insertion 2. Deletion

3. Traverse 4. Exit

Enter your choice:4

//2.SPLAY TREE:

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

```
struct node {
  int data;
  struct node *leftChild, *rightChild;
};
struct node* newNode(int data){
  struct node* Node = (struct node*)malloc(sizeof(struct node));
  Node->data = data;
  Node->leftChild = Node->rightChild = NULL;
  return (Node)
struct node* rightRotate(struct node *x){
  struct node *y = x->leftChild;
  x->leftChild = y->rightChild;
  y->rightChild = x;
  return y;
}
struct node* leftRotate(struct node *x){
  struct node *y = x->rightChild;
  x->rightChild = y->leftChild;
  y->leftChild = x;
  return y;
}
struct node* splay(struct node *root, int data){
  if (root == NULL || root->data == data)
    return root;
  if (root->data > data) {
    if (root->leftChild == NULL) return root;
    if (root->leftChild->data > data) {
```

```
      root->leftChild->leftChild = splay(root->leftChild->leftChild, data);
      root = rightRotate(root);
    } else if (root->leftChild->data < data) {
      root->leftChild->rightChild = splay(root->leftChild->rightChild, data);
      if (root->leftChild->rightChild != NULL)
        root->leftChild = leftRotate(root->leftChild);
    }
    return (root->leftChild == NULL)? root: rightRotate(root);
  } else {
    if (root->rightChild == NULL) return root;
    if (root->rightChild->data > data) {
      root->rightChild->leftChild = splay(root->rightChild->leftChild, data);
      if (root->rightChild->leftChild != NULL)
        root->rightChild = rightRotate(root->rightChild);
    } else if (root->rightChild->data < data) {
      root->rightChild->rightChild = splay(root->rightChild->rightChild, data);
      root = leftRotate(root);
    }
    return (root->rightChild == NULL)? root: leftRotate(root);
  }
}
struct node* insert(struct node *root, int k){
  if (root == NULL) return newNode(k);
  root = splay(root, k);
  if (root->data == k) return root;
  struct node *newnode = newNode(k);
  if (root->data > k) {
```

```
    newnode->rightChild = root;
    newnode->leftChild = root->leftChild;
    root->leftChild = NULL;
  } else {
    newnode->leftChild = root;
    newnode->rightChild = root->rightChild;
    root->rightChild = NULL;
  }
  return newnode;
}
void printTree(struct node *root){
  if (root == NULL)
    return;
  if (root != NULL) {
    printTree(root->leftChild);
    printf("%d ", root->data);
    printTree(root->rightChild);
  }
int main(){
  struct node* root = newNode(34);
  root->leftChild = newNode(15);
  root->rightChild = newNode(40);
  root->leftChild->leftChild = newNode(12);
  root->leftChild->leftChild->rightChild = newNode(14);
  root->rightChild->rightChild = newNode(59);
  printf("The Splay tree is: \n");
  printTree(root);
```

```
  return 0;
}
```

//OUTPUT:

```
The Splay tree is:
12 14 15 34 40 59
Initial tree
        34
       / \
     14   40
    / \    \
   12  15   59
Zig-Zig (left-left):
        34
       / \
     15   40
    / \    \
   14  -   59
  /
 12
Zig (left):
        15
       / \
     14   34
    /       \
   12        40
              \
              59
```