ADVANCE ALGORITHM
Experiment 5

Ayush Jain 60004200132 B3

Aim : To implement Red Black Tree Deletion.

Theory:

Red-Black Trees are a type of self-balancing binary search tree, where each node in the tree is either red or black, and it satisfies certain properties to ensure that the tree remains balanced. When a node is deleted from a Red-Black Tree, the tree may become unbalanced, and it needs to be rebalanced to maintain the properties of a Red-Black Tree. Here are the different cases and how to resolve them in Red-Black Tree deletion:

Case 1: The node to be deleted has no children (it is a leaf node).

Solution: Simply delete the node and update its parent's child pointer to NULL.

Case 2: The node to be deleted has one child.

Solution: Replace the node with its child, and color the child black.

Case 3: The node to be deleted has two children.

Solution: Find the node's successor (the node with the smallest key in its right subtree), and replace the node to be deleted with its successor. Then delete the successor node using Case 1 or Case 2. Note that the successor node cannot have a left child, since it is the smallest key in the right subtree. Case 4: The node to be deleted is red, and has two black children.

Solution: Simply delete the node and adjust the colors of its parent and sibling as follows:

If the node's sibling is black, and has at least one red child, rotate the tree to make the sibling the new parent of its red child, and recolor the nodes as necessary.

If the node's sibling is black, and has two black children, color the sibling red, and repeat the same steps on the parent node.

If the node's sibling is red, rotate the tree to make the sibling the new parent of its red child, and recolor the nodes as necessary. Then repeat the steps for Case 4.

Case 5: The node to be deleted is black, and has one red child and one black child.

Solution: Replace the node with its red child, and color the child black.

Case 6: The node to be deleted is black, and has two black children.

Solution: Rebalance the tree using a series of rotations and recolorings as follows:

If the node is the root of the tree, simply delete it.

Otherwise, let sibling be the node's sibling (the other child of its parent), and let parent be the node's parent. Then:

If sibling is red, rotate the tree to make the parent the new parent of sibling, recolor sibling black, and repeat the steps for Case 4 on the original node.

If sibling is black, and has at least one red child, rotate the tree to make sibling the new parent of its red child, recolor the nodes as necessary, and repeat the steps for Case 4 on the original node.

If sibling is black, and has two black children, color sibling red, set node to its parent, and repeat the steps from the beginning.

Code:

```
import java.util.*; class Node {   int data;   Node parent;
  Node left;
Node right;   int
color;
}

public class Main{   private
Node root;   private Node
TNULL;


  // Preorder   private void preOrderHelper(Node
node) {     if (node != TNULL) {
System.out.print(node.data + " ");
preOrderHelper(node.left);
preOrderHelper(node.right);
    }
  }

  // Inorder   private void inOrderHelper(Node
node) {     if (node != TNULL) {
inOrderHelper(node.left);
System.out.print(node.data + " ");
inOrderHelper(node.right);
```

```
    }
  }

  // Post order   private void
postOrderHelper(Node node) {     if (node !=
TNULL) {       postOrderHelper(node.left);
postOrderHelper(node.right);
System.out.print(node.data + " ");
    }
  }

  // Search the tree   private Node searchTreeHelper(Node
node, int key) {     if (node == TNULL || key == node.data)
{       return node;
    }

    if (key < node.data) {       return
searchTreeHelper(node.left, key);
    }
    return searchTreeHelper(node.right, key);
  }

  // Balance the tree after deletion of a node
private void fixDelete(Node x) {     Node s;
while (x != root && x.color == 0) {       if (x
== x.parent.left) {         s = x.parent.right;
if (s.color == 1) {           s.color = 0;
          x.parent.color = 1;
leftRotate(x.parent);           s =
x.parent.right;
        }

        if (s.left.color == 0 && s.right.color == 0) {
          s.color = 1;           x =
x.parent;         } else {           if
```

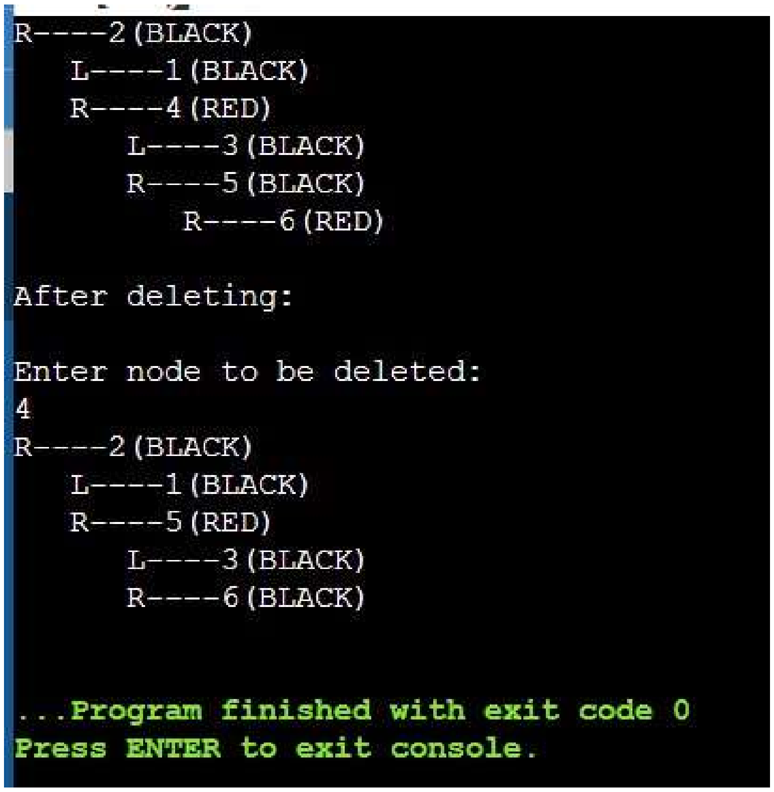
```
(s.right.color == 0) {
s.left.color = 0;
            s.color = 1;
rightRotate(s);             s =
x.parent.right;
          }

          s.color = x.parent.color;
          x.parent.color = 0;
          s.right.color = 0;
leftRotate(x.parent);           x
= root;         }
      } else {         s =
x.parent.left;         if
(s.color == 1) {
s.color = 0;
          x.parent.color = 1;
rightRotate(x.parent);           s =
x.parent.left;
        }

        if (s.right.color == 0 && s.right.color == 0) {
s.color = 1;           x = x.parent;         } else {
          if (s.left.color == 0) {
s.right.color = 0;
            s.color = 1;
leftRotate(s);             s =
x.parent.left;
          }

          s.color = x.parent.color;
          x.parent.color = 0;
          s.left.color = 0;
rightRotate(x.parent);           x =
root;
```

```
        }
      }
    }
    x.color = 0;
  }


  private void rbTransplant(Node u, Node v) {
if (u.parent == null) {       root = v;
    } else if (u == u.parent.left) {
u.parent.left = v;
    } else {
      u.parent.right = v;
    }
    v.parent = u.parent;
  }

  private void deleteNodeHelper(Node node, int key) {
    Node z = TNULL;
    Node x, y;     while (node !=
TNULL) {       if (node.data ==
key) {         z = node;
      }

      if (node.data <= key) {
node = node.right;       } else
{         node = node.left;
      }
    }

    if (z == TNULL) {
      System.out.println("Couldn't find key in the tree");       return;
    }

    y = z;     int yOriginalColor =
y.color;     if (z.left == TNULL) {
x = z.right;       rbTransplant(z,
z.right);     } else if (z.right ==
TNULL) {       x = z.left;
rbTransplant(z, z.left);
    } else {
      y = minimum(z.right);
yOriginalColor = y.color;       x
= y.right;       if (y.parent == z)
{         x.parent = y;       } else {
rbTransplant(y, y.right);
y.right = z.right;
        y.right.parent = y;
      }

      rbTransplant(z, y);
y.left = z.left;
      y.left.parent = y;
      y.color = z.color;     }
    if (yOriginalColor == 0) {
fixDelete(x);
    }
  }

  // Balance the node after insertion   private
void fixInsert(Node k) {     Node u;     while
(k.parent.color == 1) {       if (k.parent ==
k.parent.parent.right) {         u =
k.parent.parent.left;         if (u.color == 1) {
u.color = 0;
          k.parent.color = 0;
          k.parent.parent.color = 1;
k = k.parent.parent;         } else {
if (k == k.parent.left) {             k =
k.parent;             rightRotate(k);
          }
          k.parent.color = 0;
          k.parent.parent.color = 1;
leftRotate(k.parent.parent);
        }       } else {         u =
k.parent.parent.right;


        if (u.color == 1) {
u.color = 0;
          k.parent.color = 0;
          k.parent.parent.color = 1;
k = k.parent.parent;         } else {
if (k == k.parent.right) {              k
= k.parent;             leftRotate(k);
          }
          k.parent.color = 0;
          k.parent.parent.color = 1;
rightRotate(k.parent.parent);
        }
      }      if (k ==
root) {         break;
      }     }
root.color = 0;
  }

  private void printHelper(Node root, String indent, boolean last) {     if
(root != TNULL) {       System.out.print(indent);       if (last) {
        System.out.print("R----");
indent += "    ";
      } else {
        System.out.print("L----");
indent += "|  ";
      }
      String sColor = root.color == 1 ? "RED" : "BLACK";
System.out.println(root.data + "(" + sColor + ")");
```

```
printHelper(root.left, indent, false);       printHelper(root.right,
indent, true);
    }
  }

  public Main() {     TNULL
= new Node();
    TNULL.color = 0;
    TNULL.left = null;
TNULL.right = null;     root =
TNULL;
  }

  public void preorder() {
preOrderHelper(this.root);
  }

  public void inorder() {
inOrderHelper(this.root);
  }

  public void postorder() {
postOrderHelper(this.root);
  }

  public Node searchTree(int k) {     return
searchTreeHelper(this.root, k);
  }

  public Node minimum(Node node) {
while (node.left != TNULL) {       node =
node.left;
    }     return
node;
  }
  public Node maximum(Node node) {
while (node.right != TNULL) {       node =
node.right;
    }     return
node;
  }
  public Node successor(Node x) {
if (x.right != TNULL) {       return
minimum(x.right);
    }

    Node y = x.parent;     while (y != TNULL
&& x == y.right) {
      x = y;       y =
y.parent;     }
return y;
  }
  public Node predecessor(Node x) {
if (x.left != TNULL) {       return
maximum(x.left);
    }
    Node y = x.parent;     while (y != TNULL
&& x == y.left) {       x = y;       y =
y.parent;
    }

    return y;
  }
  public void leftRotate(Node x) {
Node y = x.right;     x.right = y.left;
if (y.left != TNULL) {
y.left.parent = x;
```

```
    }
    y.parent = x.parent;     if
(x.parent == null) {       this.root =
y;     } else if (x == x.parent.left) {
x.parent.left = y;
    } else {
      x.parent.right = y;
    }
    y.left = x;
    x.parent = y;
  }

  public void rightRotate(Node x) {
Node y = x.left;     x.left = y.right;     if
(y.right != TNULL) {
y.right.parent = x;
    }
    y.parent = x.parent;
    if (x.parent == null) {
this.root = y;     } else if (x ==
x.parent.right) {       x.parent.right =
y;
    } else {
      x.parent.left = y;
    }
    y.right = x;
    x.parent = y;
  }

  public void insert(int key) {
Node node = new Node();
node.parent = null;     node.data
= key;     node.left = TNULL;
node.right = TNULL;
node.color = 1;

    Node y = null;
    Node x = this.root;
    while (x != TNULL) {
      y = x;       if (node.data <
x.data) {         x = x.left;       }
else {         x = x.right;
      }
    }

    node.parent = y;     if (y == null)
{       root = node;     } else if
(node.data < y.data) {       y.left =
node;
    } else {
      y.right = node;
    }

    if (node.parent == null) {
node.color = 0;       return;
    }

    if (node.parent.parent == null) {
return;
    }

    fixInsert(node);
  }

  public Node getRoot() {     return
this.root;
  }

  public void deleteNode(int data) {
deleteNodeHelper(this.root, data);   }

  public void printTree() {
printHelper(this.root, "", true);
  }
  public static void main(String[] args) {
Main bst = new Main();     bst.insert(1);
bst.insert(2);     bst.insert(3);
bst.insert(4);     bst.insert(5);
bst.insert(6);     bst.printTree();
    System.out.println("\nAfter deleting:");
    System.out.println("\nEnter node to be deleted:");
Scanner sc=new Scanner(System.in);     int
toBeDeleted=sc.nextInt();
bst.deleteNode(toBeDeleted);     bst.printTree();
  }
}
```

Output:



```
R----2(BLACK)
   L----1(BLACK)
   R----4(RED)
      L----3(BLACK)
      R----5(BLACK)
         R----6(RED)

After deleting:

Enter node to be deleted:
4
R----2(BLACK)
   L----1(BLACK)
   R----5(RED)
      L----3(BLACK)
      R----6(BLACK)


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

Conclusion: In conclusion, we have studied how to perform deletion in Red Black Tree.