PROGRAM 5: RED-BLACK TREE IMPLEMENTATION

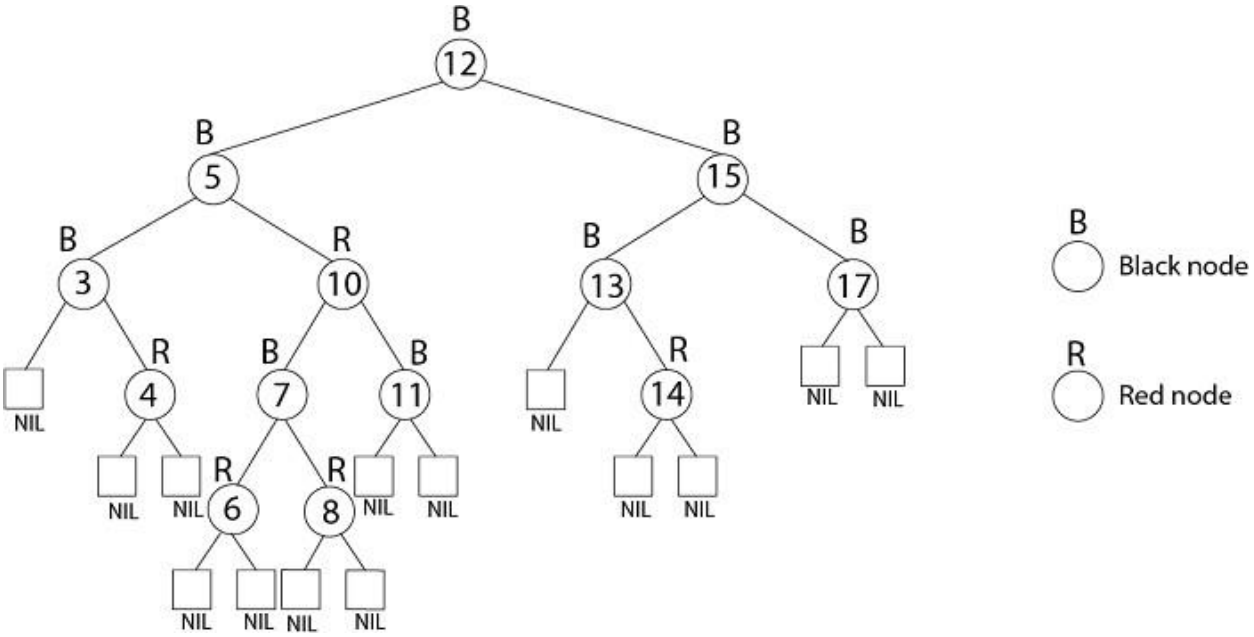
Aim: To write a program in C++ for implementing Red-Black tree.

Description:

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 colour (red or black). This tree was invented in 1972 by Rudolf Bayer.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the 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 descendants NULL nodes has the same number of black nodes.



Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

1. Recolor
2. Rotation
3. Rotation followed by Recolor

Algorithm:

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Program:

```
// Implementing Red-Black Tree in C++
#include <iostream>
using namespace std;
struct Node {
  int  data;
  Node *parent;
  Node *left;
  Node *right;
  int color;
};

typedef Node *NodePtr;

class RedBlackTree {
   private:
  NodePtr root;
  NodePtr TNULL;

  void initializeNULLNode(NodePtr node, NodePtr parent) {
    node->data = 0;
    node->parent = parent;
    node->left = nullptr;
    node->right = nullptr;
    node->color = 0;
  }
```

```
// Preorder
void preOrderHelper(NodePtr node) {
  if (node != TNULL) {
    cout << node->data << " ";
    preOrderHelper(node->left);
    preOrderHelper(node->right);
  }
}

// Inorder
void inOrderHelper(NodePtr node) {
  if (node != TNULL) {
    inOrderHelper(node->left);
    cout << node->data << " ";
    inOrderHelper(node->right);
  }
}

// Post order
void postOrderHelper(NodePtr node) {
  if (node != TNULL) {
    postOrderHelper(node->left);
    postOrderHelper(node->right);
    cout << node->data << " ";
  }
```

```
}

NodePtr searchTreeHelper(NodePtr 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);
}

// For balancing the tree after deletion
void deleteFix(NodePtr x) {
  NodePtr 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;
}

void rbTransplant(NodePtr u, NodePtr v) {
  if (u->parent == nullptr) {
    root = v;
  } else if (u == u->parent->left) {
```

```
    u->parent->left = v;
  } else {
    u->parent->right = v;
  }
  v->parent = u->parent;
}

void deleteNodeHelper(NodePtr node, int key) {
  NodePtr z = TNULL;
  NodePtr 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) {
    cout << "Key not found in the tree" << endl;
    return;
  }
```

```
y = z;
int y_original_color = 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);
  y_original_color = 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;
}
```

```
  delete z;
  if (y_original_color == 0) {
    deleteFix(x);
  }
}

// For balancing the tree after insertion
void insertFix(NodePtr k) {
  NodePtr 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;
}

void printHelper(NodePtr root, string indent, bool last) {
```

```
  if (root != TNULL) {
    cout << indent;
    if (last) {
      cout << "R---- ";
      indent += "    ";
    } else {
      cout << "L---- ";
      indent += "|   ";
    }

    string sColor = root->color ? "RED" : "BLACK";
    cout << root->data << "(" << sColor << ")" << endl;
    printHelper(root->left, indent, false);
    printHelper(root->right, indent, true);
  }
}

 public:
RedBlackTree() {
  TNULL = new Node;
  TNULL->color = 0;
  TNULL->left = nullptr;
  TNULL->right = nullptr;
  root = TNULL;
}
```

```
void preorder() {
  preOrderHelper(this->root);
}

void inorder() {
  inOrderHelper(this->root);
}

void postorder() {
  postOrderHelper(this->root);
}

NodePtr searchTree(int k) {
  return searchTreeHelper(this->root, k);
}

NodePtr minimum(NodePtr node) {
  while (node->left != TNULL) {
    node = node->left;
  }
  return node;
}

NodePtr maximum(NodePtr node) {
  while (node->right != TNULL) {
    node = node->right;
```

```
  }
  return node;
}

NodePtr successor(NodePtr x) {
  if (x->right != TNULL) {
    return minimum(x->right);
  }

  NodePtr y = x->parent;
  while (y != TNULL && x == y->right) {
    x = y;
    y = y->parent;
  }
  return y;
}

NodePtr predecessor(NodePtr x) {
  if (x->left != TNULL) {
    return maximum(x->left);
  }

  NodePtr y = x->parent;
  while (y != TNULL && x == y->left) {
    x = y;
    y = y->parent;
```

```
  }

  return y;
}

void leftRotate(NodePtr x) {
  NodePtr y = x->right;
  x->right = y->left;
  if (y->left != TNULL) {
    y->left->parent = x;
  }
  y->parent = x->parent;
  if (x->parent == nullptr) {
    this->root = y;
  } else if (x == x->parent->left) {
    x->parent->left = y;
  } else {
    x->parent->right = y;
  }
  y->left = x;
  x->parent = y;
}

void rightRotate(NodePtr x) {
  NodePtr y = x->left;
  x->left = y->right;
```

```
  if (y->right != TNULL) {
    y->right->parent = x;
  }
  y->parent = x->parent;
  if (x->parent == nullptr) {
    this->root = y;
  } else if (x == x->parent->right) {
    x->parent->right = y;
  } else {
    x->parent->left = y;
  }
  y->right = x;
  x->parent = y;
}

// Inserting a node
void insert(int key) {
  NodePtr node = new Node;
  node->parent = nullptr;
  node->data = key;
  node->left = TNULL;
  node->right = TNULL;
  node->color = 1;

  NodePtr y = nullptr;
  NodePtr 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 == nullptr) {
  root = node;
} else if (node->data < y->data) {
  y->left = node;
} else {
  y->right = node;
}

if (node->parent == nullptr) {
  node->color = 0;
  return;
}

if (node->parent->parent == nullptr) {
  return;
```

```
    }

    insertFix(node);
  }

  NodePtr getRoot() {
    return this->root;
  }

  void deleteNode(int data) {
    deleteNodeHelper(this->root, data);
  }

  void printTree() {
    if (root) {
      printHelper(this->root, "", true);
    }
  }
};

int main() {
  RedBlackTree bst;
  bst.insert(55);
  bst.insert(40);
  bst.insert(65);
  bst.insert(60);
```

```
  bst.i
  nsert
  (75);
  bst.i
  nsert
  (57);

  bst.pri
  ntTre
  e();
  cout
  <<
  endl
     << "After
  deleting" << endl;
  bst.deleteNode(40);
  bst.printTree();
}
```

Result:

Thus a program is implemented and executed successfully for Red-Blacktree.