**Jayesh Kavedia**

**60004190053**

**TE COMPS A**

**Implementation of Advanced Data Structure for Red-Black Tree Deletion Rules**

**Theory**

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.

RB Tree Deletion Algorithm

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child. So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v.

2) Simple Case: If either u or v is red, we mark the replaced child as black.

3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black.

3.2) Do following while the current node u is double black, and it is not the root. Let sibling of node be s.

(a): If sibling s is black and at least one of sibling’s children is red, perform rotation(s). Let the red child of s be r. This case can be divided in four subcases depending upon positions of s and r.

(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red).

(ii) Left Right Case (s is left child of its parent and r is right child).

(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)

(iv) Right Left Case (s is right child of its parent and r is left child of s)

(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black. In this case, if parent was red, then we didn’t need to recur for parent, we can simply make it black.

(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black.This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

(i) Left Case (s is left child of its parent). We right rotate the parent p.

(ii) Right Case (s is right child of its parent). We left rotate the parent p.

3.3) If u is root, make it single black and return.

**Code**

#include <iostream>  
#include <queue>  
using namespace std;  
   
enum COLOR { RED, BLACK };  
   
class Node {  
public:  
  int val;  
  COLOR color;  
  Node \*left, \*right, \*parent;  
   
  Node(int val) : val(val) {  
    parent = left = right = NULL;  
    color = RED;  
  }  
   
  Node \*uncle() {  
    if (parent == NULL or parent->parent == NULL)  
      return NULL;  
   
    if (parent->isOnLeft())  
      return parent->parent->right;  
    else  
      return parent->parent->left;  
  }  
   
  bool isOnLeft() { return this == parent->left; }  
   
  Node \*sibling() {  
    if (parent == NULL)  
      return NULL;  
   
    if (isOnLeft())  
      return parent->right;  
   
    return parent->left;  
  }  
   
  void moveDown(Node \*nParent) {  
    if (parent != NULL) {  
      if (isOnLeft()) {  
        parent->left = nParent;  
      } else {  
        parent->right = nParent;  
      }  
    }  
    nParent->parent = parent;  
    parent = nParent;  
  }  
   
  bool hasRedChild() {  
    return (left != NULL and left->color == RED) or  
           (right != NULL and right->color == RED);  
  }  
};  
   
class RBTree {  
  Node \*root;  
   
  void leftRotate(Node \*x) {  
    Node \*nParent = x->right;  
    if (x == root)  
      root = nParent;  
   
    x->moveDown(nParent);  
   
    x->right = nParent->left;  
    if (nParent->left != NULL)  
      nParent->left->parent = x;  
    nParent->left = x;  
  }  
   
  void rightRotate(Node \*x) {  
    Node \*nParent = x->left;  
   
    if (x == root)  
      root = nParent;  
   
    x->moveDown(nParent);  
   
    x->left = nParent->right;  
    if (nParent->right != NULL)  
      nParent->right->parent = x;  
   
    nParent->right = x;  
  }  
   
  void swapColors(Node \*x1, Node \*x2) {  
    COLOR temp;  
    temp = x1->color;  
    x1->color = x2->color;  
    x2->color = temp;  
  }  
   
  void swapValues(Node \*u, Node \*v) {  
    int temp;  
    temp = u->val;  
    u->val = v->val;  
    v->val = temp;  
  }  
   
  void fixRedRed(Node \*x) {  
    if (x == root) {  
      x->color = BLACK;  
      return;  
    }  
   
    Node \*parent = x->parent, \*grandparent = parent->parent,  
         \*uncle = x->uncle();  
   
    if (parent->color != BLACK) {  
      if (uncle != NULL && uncle->color == RED) {  
        parent->color = BLACK;  
        uncle->color = BLACK;  
        grandparent->color = RED;  
        fixRedRed(grandparent);  
      } else {  
        if (parent->isOnLeft()) {  
          if (x->isOnLeft()) {  
            swapColors(parent, grandparent);  
          } else {  
            leftRotate(parent);  
            swapColors(x, grandparent);  
          }  
          rightRotate(grandparent);  
        } else {  
          if (x->isOnLeft()) {  
            rightRotate(parent);  
            swapColors(x, grandparent);  
          } else {  
            swapColors(parent, grandparent);  
          }  
          leftRotate(grandparent);  
        }  
      }  
    }  
  }  
   
  Node \*successor(Node \*x) {  
    Node \*temp = x;  
   
    while (temp->left != NULL)  
      temp = temp->left;  
   
    return temp;  
  }  
  Node \*BSTreplace(Node \*x) {  
    if (x->left != NULL and x->right != NULL)  
      return successor(x->right);  
    if (x->left == NULL and x->right == NULL)  
      return NULL;  
    if (x->left != NULL)  
      return x->left;  
    else  
      return x->right;  
  }  
  void deleteNode(Node \*v) {  
    Node \*u = BSTreplace(v);  
    bool uvBlack = ((u == NULL or u->color == BLACK) and (v->color == BLACK));  
    Node \*parent = v->parent;  
   
    if (u == NULL) {  
      if (v == root) {  
        root = NULL;  
      } else {  
        if (uvBlack) {  
          fixDoubleBlack(v);  
        } else {  
          if (v->sibling() != NULL)  
            v->sibling()->color = RED;  
        }  
   
        if (v->isOnLeft()) {  
          parent->left = NULL;  
        } else {  
          parent->right = NULL;  
        }  
      }  
      delete v;  
      return;  
    }  
   
    if (v->left == NULL or v->right == NULL) {  
      if (v == root) {  
        v->val = u->val;  
        v->left = v->right = NULL;  
        delete u;  
      } else {  
        if (v->isOnLeft()) {  
          parent->left = u;  
        } else {  
          parent->right = u;  
        }  
        delete v;  
        u->parent = parent;  
        if (uvBlack) {  
          fixDoubleBlack(u);  
        } else {  
          u->color = BLACK;  
        }  
      }  
      return;  
    }  
   
    swapValues(u, v);  
    deleteNode(u);  
  }  
   
  void fixDoubleBlack(Node \*x) {  
    if (x == root)  
      return;  
   
    Node \*sibling = x->sibling(), \*parent = x->parent;  
    if (sibling == NULL) {  
      fixDoubleBlack(parent);  
    } else {  
      if (sibling->color == RED) {  
        parent->color = RED;  
        sibling->color = BLACK;  
        if (sibling->isOnLeft()) {  
          rightRotate(parent);  
        } else {  
          leftRotate(parent);  
        }  
        fixDoubleBlack(x);  
      } else {  
        if (sibling->hasRedChild()) {  
          if (sibling->left != NULL and sibling->left->color == RED) {  
            if (sibling->isOnLeft()) {  
              sibling->left->color = sibling->color;  
              sibling->color = parent->color;  
              rightRotate(parent);  
            } else {  
              sibling->left->color = parent->color;  
              rightRotate(sibling);  
              leftRotate(parent);  
            }  
          } else {  
            if (sibling->isOnLeft()) {  
              sibling->right->color = parent->color;  
              leftRotate(sibling);  
              rightRotate(parent);  
            } else {  
              sibling->right->color = sibling->color;  
              sibling->color = parent->color;  
              leftRotate(parent);  
            }  
          }  
          parent->color = BLACK;  
        } else {  
          sibling->color = RED;  
          if (parent->color == BLACK)  
            fixDoubleBlack(parent);  
          else  
            parent->color = BLACK;  
        }  
      }  
    }  
  }  
   
  void levelOrder(Node \*x) {  
    if (x == NULL)  
      return;  
   
    queue<Node \*> q;  
    Node \*curr;  
   
    q.push(x);  
   
    while (!q.empty()) {  
      curr = q.front();  
      q.pop();  
   
      cout << curr->val << " ";  
   
      if (curr->left != NULL)  
        q.push(curr->left);  
      if (curr->right != NULL)  
        q.push(curr->right);  
    }  
  }  
   
  void inorder(Node \*x) {  
    if (x == NULL)  
      return;  
    inorder(x->left);  
    cout << x->val << " ";  
    inorder(x->right);  
  }  
   
public:  
  RBTree() { root = NULL; }  
   
  Node \*getRoot() { return root; }  
   
  Node \*search(int n) {  
    Node \*temp = root;  
    while (temp != NULL) {  
      if (n < temp->val) {  
        if (temp->left == NULL)  
          break;  
        else  
          temp = temp->left;  
      } else if (n == temp->val) {  
        break;  
      } else {  
        if (temp->right == NULL)  
          break;  
        else  
          temp = temp->right;  
      }  
    }  
   
    return temp;  
  }  
   
  void insert(int n) {  
    Node \*newNode = new Node(n);  
    if (root == NULL) {  
      newNode->color = BLACK;  
      root = newNode;  
    } else {  
      Node \*temp = search(n);  
   
      if (temp->val == n) {  
        return;  
      }  
      newNode->parent = temp;  
   
      if (n < temp->val)  
        temp->left = newNode;  
      else  
        temp->right = newNode;  
   
      fixRedRed(newNode);  
    }  
  }  
   
  void deleteByVal(int n) {  
    if (root == NULL)  
      return;  
   
    Node \*v = search(n), \*u;  
   
    if (v->val != n) {  
      cout << "No node found to delete with value:" << n << endl;  
      return;  
    }  
   
    deleteNode(v);  
  }  
   
  void printInOrder() {  
    cout << "Inorder: " << endl;  
    if (root == NULL)  
      cout << "Tree is empty" << endl;  
    else  
      inorder(root);  
    cout << endl;  
  }  
   
  void printLevelOrder() {  
    cout << "Level order: " << endl;  
    if (root == NULL)  
      cout << "Tree is empty" << endl;  
    else  
      levelOrder(root);  
    cout << endl;  
  }  
};  
   
int main() {  
  RBTree tree;  
   
    tree.insert(8);  
    tree.insert(18);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(17);  
    tree.insert(25);  
    tree.insert(40);  
    tree.insert(80);  
  
   
  tree.printInOrder();  
  tree.printLevelOrder();  
   
  cout<<endl<<"Deleting 8"<<endl;  
  tree.deleteByVal(8);  
  tree.printLevelOrder();  
  cout<<endl<<"Deleting 80"<<endl;  
  tree.deleteByVal(80);  
  tree.printLevelOrder();  
  cout<<endl<<"Deleting 5"<<endl;   
  tree.deleteByVal(5);  
  tree.printLevelOrder();  
  return 0;  
}

**Output**

// Original Tree

Inorder:   
5 8 15 17 18 25 40 80   
Level order:   
17 8 25 5 15 18 40 80   
  
Deleting 8  
Level order:   
17 15 25 5 18 40 80   
  
Deleting 80  
Level order:   
17 15 25 5 18 40   
  
Deleting 5  
Level order:   
17 15 25 18 40

**Observations**

1. Deletion follows BST deletion property and finds successor node, along with a fixup method to balance the tree in case of any property violation.
2. In case of Double Black Node there are eleven possible cases to be handled.
3. Red-black trees offer logarithmic average and worst-case time complexity for deletion.
4. Rebalancing has a time complexity of O(1) and worst-case complexity of O(log n).

**Conclusion**

Red Black Tree is a self-balanced tree similar to BST with one extra bit of storage for the color value. While Deleting a node in RB Tree the deleted node’s successor needs to be found and fixed to avoid any property violations. The Average Case Time Complexity for RB Tree Deletion is O (log n).

**References**

1. [https://www.baeldung.com/cs/red-black-trees#:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.](https://www.baeldung.com/cs/red-black-trees%23:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.)
2. <https://www.cs.umanitoba.ca/~hacamero/Research/RBTreesKim.pdf>
3. “Introduction to Algorithms, Second Edition,” by Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest and Clifford Stein.