Project Red Black Tree

Ali Hamza L1F21BSCS0791

```
#pragma once
#include <iostream>
using namespace std;

template <class T>
class Node {
public:
	T key;
	bool color;
	Node<T>* left;
	Node<T>* right;
	Node<T>* parent;
	Node() {
		key = 0;
		color = 1;
		left = nullptr;
		parent = nullptr;
		right = nullptr;
	}
};

template <class T>
class RBT :public Node<T> {
protected:
	Node<T>* root;

public:

	RBT() {
		root = nullptr;
	}

	Node<T>* insert(Node<T>* root, Node<T>* newNode) {
		if (root == nullptr) {
			return newNode;
		}


		if (newNode->key < root->key) {
			root->left = insert(root->left, newNode);
			root->left->parent = root;
		}

		/

		else {
```

```
            root->right = insert(root->right, newNode);
            root->right->parent = root;
      }
      return root;
}

void Inorder(Node<T>* proot) {
      if (proot == nullptr)
      {
            return;
      }

      Inorder(proot->left);
      cout << "\t\tIt is a  ";
      if (proot->color == 0)
      {
            cout << "Black Node\t";
      }
      else
      {
            cout << "Red Node\t";
      }
      cout << proot->key << endl;
      Inorder(proot->right);
}

void Preorder(Node<T>* proot) {
      if (proot == nullptr)
      {
            return;
      }
      cout << proot->key << endl;
      Preorder(proot->left);

      Preorder(proot->right);
}

void Postorder(Node<T>* proot) {
      if (proot == nullptr)
      {
            return;
      }

      Postorder(proot->left)
      Postorder(proot->right);
      cout << proot->key << endl;
}

Node<T>* searchTree(T entry, Node<T>* root) {
      if (root != nullptr) {
            if (entry == root->key) {
                  return root;
            }
            else if (entry < root->key)
```

```
                {
                        return searchTree(entry, root->left);
                }
                else {
                        return searchTree(entry, root->right);
                }
        }
        else
                return nullptr;
}

void delNode(Node<T>* GrandChild) {
        if ((root == nullptr) || (GrandChild == nullptr)) {
                return;
        }

        Node<T>* Parent = GrandChild->parent;
        Node<T>* Temp = switcNode(GrandChild);
        Node<T>* ParentSibling = getParentSibling(GrandChild);

        if (Temp == nullptr) {
                if (root == GrandChild)
                        root = nullptr;
                else {
                        if ((Temp == nullptr) && (GrandChild->color == 0)) {
                                delRules(GrandChild);
                        }
                        else {
                                if (ParentSibling != nullptr) {
                                        ParentSibling->color = 1;
                                }
                        }
                        if (!isLeftChild(GrandChild)) {
                                Parent->right = nullptr;
                        }
                        else {
                                Parent->left = nullptr;
                        }
                }
                free(GrandChild);
                return;
        }
        else {
                if (GrandChild == root) {
                        GrandChild->key = Temp->key;
                        free(GrandChild);
                }
                else {
                        if (!isLeftChild(GrandChild)) { Parent->right = Temp; }
                        else { Parent->left = Temp; }
                        Temp->parent = Parent;
                        if ((Temp == nullptr || Temp->color == 0) && (GrandChild-
>color == 0)) {
                                delRules(Temp);
```

```
                }
                else {
                      Temp->color = 0;
                }
                free(GrandChild);
          }
          return;
     }
     swapkeys(Temp->key, GrandChild->key);
     delNode(Temp);
}
Node<T>* getParentSibling(Node<T>* p) {
     if (p->parent == nullptr) {
          return nullptr;
     }
     else if (isLeftChild(p->parent)) {
          return p->parent->right;
     }
     else {
          return p->parent->left;
     }
}
Node<T>* getParent(Node<T>* root, Node<T>* p) {
     if (root == NULL) { return NULL; }
     else {
          if (root->key == p->key) { return p; }
          else {
                return getParent(root->left, p);
                return getParent(root->right, p);
          }
     }
}
bool twoChild(Node<T>* Parent) {
     if (Parent->right != nullptr && Parent->left != nullptr)
     {
          return true;
     }
     return false;
}
int  countBlack(Node<T>* root) {
     if (root == nullptr) {
          return 1;
     }
     int rightSubtree = countBlack(root->right);
     int leftSubtree = countBlack(root->left);
     if (rightSubtree == leftSubtree) {
          if (root->color == 0) {
                leftSubtree++;
          }
     }
     return leftSubtree;
}
int height(Node<T>* root) {
     if (root == nullptr) {
```

```
            return 0;
      }
      else {
            if (height(root->left) > height(root->right)) {
                  return height(root->left) + 1;
            }
            else {
                  return height(root->right) + 1;
            }
      }
}
bool isbalanced() {
      int h = height(root->left) - height(root->right);
      if (h == 0) {
            return true;
      }
      else {
            return false;
      }
}
Node<T>* switcNode(Node<T>* root) {
      if (root == nullptr) {
            return nullptr;
      }
      else if (root->left == nullptr) {
            return root->right;
      }
      else if (root->right == nullptr) {
            return root->left;
      }
      else {
            return inorderSuccessor(root->right);
      }
}
Node<T>* inorderSuccessor(Node<T>* root) {
      while (root->left != nullptr) {
            root = root->left;
      }
      return root;
}
bool isLeftChild(Node<T>* root) {
      if (root->parent->right == root) {
            return false;
      }
      else return false;
}
void swapkeys(T Parent, T GrandChild) {
      T temp = Parent;
      Parent = GrandChild;
      GrandChild = Parent;
}
void swapNodes(Node<T>* Parent, Node<T>* GrandChild) {
      if (Parent->parent == nullptr) {
            root = GrandChild;
```

```
		}
		else if (isLeftChild(Parent)) {
			Parent->parent->left = GrandChild;
		}
		else {
			Parent->parent->right = GrandChild;
		}
		GrandChild->parent = Parent->parent;
	}
	void leftRotation(Node<T>* root) {
		Node<T>* temp = root->right;
		root->right = temp->left;
		if (temp->left != nullptr) {
			temp->left->parent = root;
		}
		temp->parent = root->parent;
		if (root->parent == nullptr) { root = temp; }
		else {
			if (isLeftChild(root)) { root->parent->left = temp; }
			else { root->parent->right = temp; }
		}
		root->parent = temp;
		temp->left = root;
	}
	void rightRotation(Node<T>*) {
		Node<T>* temp = root->left;
		root->left = temp->right;
		if (temp->right != nullptr) { temp->right->parent = root; }
		temp->parent = root->parent;
		if (root->parent == nullptr) { root = temp; }
		else {
			if (isLeftChild(root)) { root->parent->left = temp; }
			else { root->parent->right = temp; }
		}
		root->parent = temp;
		temp->right = root;
	}
	void checkRules(Node<T>* GrandChild) {
		Node<T>* GrandParent = GrandChild->parent->parent;
		Node<T>* Parent = GrandChild->parent;
		while (Parent->color == 1) {
			if (GrandChild == root || GrandChild->color == 0) {
				return;
			}
			if (isLeftChild(Parent))
			{
				Node<T>* ParentSibling = GrandParent->right;
				if (ParentSibling != nullptr && ParentSibling->color ==
1) {
					Parent->color = 0;
					GrandParent->color = 1;
					ParentSibling->right->color = 0;
					GrandChild = GrandParent;
				}
```

```
                        else {
                              if (!isLeftChild(GrandChild)) {
                                    GrandChild = Parent;
                                    GrandChild->parent = Parent;
                                    leftRotation(GrandChild);
                              }
                              rightRotation(GrandParent);
                              int temp = Parent->color;
                              Parent->color = GrandParent->color;
                              GrandParent->color = temp;
                              GrandChild = Parent;
                        }
                  }
                  else {
                        Node<T>* ParentSibling = GrandParent->left;
                        if (ParentSibling != nullptr && ParentSibling->color ==
1) {
                              Parent->color = 0;
                              GrandParent->color = 1;
                              ParentSibling->color = 0;
                              GrandChild = GrandParent;
                        }
                        else {
                              if (isLeftChild(GrandChild)) {
                                    rightRotation(Parent);
                                    GrandChild = Parent;
                                    Parent = GrandChild->parent;
                              }
                              leftRotation(GrandParent);
                              int temp = Parent->color;
                              Parent->color = GrandParent->color;
                              GrandParent->color = temp;
                              GrandChild = Parent;

                        }
                  }
            }

            root->color = 0;
      }
      void delRules(Node<T>* GrandChild) {
            if (root == GrandChild) {
                  return;
            }
            Node<T>* Sibling = getParentSibling(GrandChild);
            Node<T>* Parent = GrandChild->parent;

            if (Sibling != nullptr) {
                  if (Sibling->color == 1) {
                        Parent->color = 1;
                        Sibling->color = 0;
                        if (isLeftChild(Sibling)) {
                              rightRotation(Parent);
                        }
```

```
                        else {
                              leftRotation(Parent);
                        }
                        delRules(GrandChild);
                  }
                  else  if ((Sibling->right->color == 1) || (Sibling->left-
>color == 1)) {
                        if (Sibling->left != nullptr && Sibling->left->color ==
1) {
                              if (isLeftChild(Sibling)) {
                                    Sibling->left->color = Sibling->color;
                                    Sibling->color = Parent->color;
                                    rightRotation(Parent);
                              }
                              else {
                                    Sibling->left->color = Parent->color;
                                    rightRotation(Sibling);
                                    leftRotation(Parent);
                              }
                        }
                        else {
                              if (!isLeftChild(Sibling)) {
                                    Sibling->right->color = Sibling->color;
                                    Sibling->color = Parent->color;
                                    leftRotation(Parent);
                              }
                              else {
                                    Sibling->right->color = Parent->color;
                                    leftRotation(Parent);
                                    rightRotation(Sibling);
                              }
                        }
                        Parent->color = 0;
                  }
                  else if (Sibling == nullptr) {
                        delRules(Parent);
                  }
                  else {
                        Sibling->color = 1;
                        if (Parent->color == 0) {
                              delRules(Parent);
                        }
                        else {
                              Parent->color = 0;
                        }
                  }
            }
      }



      void insertion(T user) {
            Node<T>* newNode = new Node<T>;
            if (root == nullptr) {
```

```
                newNode->key = user;
                newNode->color = 0;
                root = newNode;
          }
          else {
                newNode->key = user;
                insert(root, newNode);

                checkRules(newNode);
          }

          cout << "\t\t      <--Insertion in RBT-->          ";
          cout << "\n\t\t-------------------------------\n";
          inorder();
          cout << "\n\n";
    }
    void Deletion(T user) {
          if (root == nullptr) { return; }
          Node<T>* delME = searchTree(user, root);
          if (delME == nullptr) {
                cout << "Value Not Found!\n";
                return;
          }
          delNode(delME);
          cout << "\t\t     <--Deletion in RBT-->           ";
          cout << "\n\t\t-------------------------------\n";
          inorder();
          cout << "\n\n";

    }

    void preorder()
    {
          Preorder(root);
    }

    void inorder() {
          Inorder(root);
    }

    void postorder()
    {
          Postorder(root);
    }

};

int main() {

    RBT<int> tree;
    const int Size = 8;
    int arr[Size] = {};
    int insert = 0;
    int selection = 0;
```

```
cout << "\t\t----------- \tWelcome To Red Black Tree\t -----------\n\n";
cout << "\t\t The Object Created is of Integer Type\n\n";

do {

      cout << "\t\t\t<Main Menu>\n\n";
      cout << "Press 1 for insertion :\n";
      cout << "Press 2 for In-Order Display :\n";
      cout << "Press 3 for Pre-Order Display :\n";
      cout << "Press 4 for Post-Order Display :\n";
      cout << "Press 5 for Deletion :\n";
      cout << "Press 6 for Exist :\n";
      cout << "Enter Your Selection:";
      cin >> selection;
      system("cls");
      if (selection == 1) {

             system("cls");


                  tree.insertion(6);
            tree.insertion(3);
            tree.insertion(10);
            tree.insertion(11);
            tree.insertion(2);
            tree.insertion(14);
            tree.insertion(26);
            tree.insertion(2);

      }


      else if (selection == 2) {
            tree.inorder();
      }
      else if (selection == 3) {
            tree.preorder();
      }
      else if (selection == 4) {
            tree.postorder();
      }
      else if (selection == 5) {
            int delSel = 0;
            cout << "\n\tEnter the Tree Value which you want to delete: ";
            cin >> delSel;
            tree.Deletion(delSel);
      }
      else if (selection == 6) {

            break;
      }
      else {
```

```
                cout << "\n\nWrong Selection Enter Your Selection Again
ranging from 1-6:";
                cin >> selection;
                system("cls");
            }
      } while (selection != 6);


      return 0;
}
```