Introduction to algorithms

Exercise #2

A. Environment

a. How to run your code

- OS: windows
- Compiler version: g++
- IDE: visual studio 2019

B. Results

a. Method or solutions

- Header file:

```
#include <iostream>
#include <stdio.h>
#include <climits>
#include <limits.h>


using namespace std;
```

- 定義紅黑樹的節點類（Node）:

包括父節點、左子節點、右子節點、鍵值（key）和顏色（color，紅色是

false，黑色是 true）。

```
class Node{
    public:
        friend class RBTree;
        Node *parent;
        Node *left;
        Node *right;
        int key;
        bool color;
        //black:true; red:false

        Node():parent(0),left(0),right(0),color(false){};
        Node(int key):parent(0),left(0),right(0),key(key),color(false){};

};
```

- 定義紅黑樹類（RBTree）:

  包括根節點和指向 NIL 節點的指針。NIL 節點在紅黑樹中用於表示葉子節點下面的空節點。

  包含了各種操作紅黑樹的方法，如搜尋（search）、插入（insertRBTree）、刪除（deleteRBTree）、旋轉（leftRotation、rightRotation）等。

  - Insert + insert 之後的整理:

```
void insertRBTree(int key){
    Node *x = root;
    Node *y = NIL;
    Node *insertNode = new Node(key);
    while(x != NIL){
        y = x;
        if(insertNode->key < x->key){
            x = x->left;
        }else{
            x = x->right;
        }
    }
    insertNode->parent = y;
    if(y == NIL){
        this->root = insertNode;
    }
    else if(insertNode->key < y->key){
        y->left = insertNode;
    }else{
        y->right = insertNode;
    }

    insertNode->left = NIL;
    insertNode->right = NIL;
    insert_fixedUp(insertNode);
};
```

```
void insert_fixedUp(Node *current){
    //case 0: when parent is black, that is OK
    while(current->parent->color == false){
        Node *uncle = new Node;
        if(current->parent == current->parent->parent->left){
            uncle = current->parent->parent->right;
        }else{
            uncle = current->parent->parent->left;
        }

        //case 1: when uncle is red, change color
        if(uncle->color == false){
            current->parent->color = true;
            current->parent->parent->color = false;
            uncle->color = true;
            current = current->parent->parent;
        }
        //case 2&3: uncle is black
        else{
            if(uncle == current->parent->parent->right){
                //case 2
                if(current == current->parent->right){
                    current = current->parent;
                    leftRotation(current);
                }
                //case 3
                current->parent->color = true;
                current->parent->parent->color = false;
                rightRotation(current->parent->parent);
            }else{
                //case 2
                if(current == current->parent->left){
                    current = current->parent;
                    rightRotation(current);
                }
                //case 3
                current->parent->color = true;
                current->parent->parent->color = false;
                leftRotation(current->parent->parent);
            }
        }
    }
    root->color = true;
```

■ Delete + delete 之後的整理:

```
void deleteRBTree(int KEY){
    Node *deleteNode = search(KEY);
    if(deleteNode == NIL){
        return;
    }
    Node *y = 0;    //delete_node
    Node *x = 0;    //delete_node's child
    if(deleteNode->left == NIL){
        y = deleteNode;
    }
    else if(deleteNode->right == NIL){
        y = deleteNode;
    }else{
        y = successor(deleteNode);
    }

    if(y->left != NIL){
        x = y->left;
    }else{
        x = y->right;
    }
    x->parent = y->parent;

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

```
    //case 3:delete_node has two children
    if(y != deleteNode){
        deleteNode->key = y->key;
        //the node's color no change, since we just copy data
    }

    if(y->color == true){
        delete_fixedUp(x);
    }
};
void delete_fixedUp(Node *current){
    //case 0: if current is red, change its color to black
    while(current != root && current->color == true){
        //current is on the left
        if(current == current->parent->left){
            Node *sibling = current->parent->right;

            //case 1:when sibling is red
            if(sibling->color == false){
                sibling->color = true;
                sibling->parent->color = false;
                leftRotation(sibling->parent);
                sibling = current->parent->right;
            }
            //finishing case 1, it will enter case 2&3&4
            //case 2:when sibling is black and two children is black
            if(sibling->left->color == true && sibling->right->color == true){
                sibling->color = false;
                current = current->parent;
            }
```

```
            //finishing case 2, it will decide which case again(including case 1,2,3,4)
            else{
                //case 3:when sibling is black and leftchild is red
                if(sibling->right->color == true){
                    sibling->left->color = true;
                    sibling->color = false;
                    rightRotation(sibling);
                    sibling = current->parent->right;
                }
                //finishing case 3, it will change to case 4
                //case 4:when sibling is black and rightchild is red
                sibling->color = sibling->parent->color;
                sibling->parent->color = true;
                sibling->right->color = true;
                leftRotation(sibling->parent);
                current = root;
            }
        }
```

```
//current is on the right
else{
    Node *sibling = current->parent->left;

    //case 1:when sibling is red
    if(sibling->color == false){
        sibling->color = true;
        sibling->parent->color = false;
        rightRotation(sibling->parent);
        sibling = current->parent->left;
    }
    //finishing case 1, it will enter case 2&3&4
    //case 2:when sibling is black and two children is black
    if(sibling->left->color == true && sibling->right->color == true){
        sibling->color = false;
        current = current->parent;
    }
    //finishing case 2, it will decide which case again(including case 1,2,3,4)
    else{
        //case 3:when sibling is black and rightchild is red
        if(sibling->left->color == true){
            sibling->right->color = true;
            sibling->color = false;
            leftRotation(sibling);
            sibling = current->parent->left;
        }
        //finishing case 3, it will change to case 4
        //case 4:when sibling is black and rightchild is red
        sibling->color = sibling->parent->color;
        sibling->parent->color = true;
        sibling->left->color = true;
        rightRotation(sibling->parent);
        current = root;
    }
```

- Search

```
Node* search(int KEY){
    Node *current = root;
    while(current != NIL && KEY != current->key){
        if(KEY < current->key){
            current = current->left;
        }else{
            current = current->right;
        }
    }
    return current;
};
```

- Successor

```
Node* successor(Node* current){
    if(current->right != NIL){
        current = current->right;
        while(current->left != NIL){
            current = current->left;
        }
        return current;
    }

    Node *next = current->parent;
    while(next != NIL && current != next->left){
        current = next;
        next = next->parent;
    }
    return next;
};
```

- Left rotation

```
void leftRotation(Node *x){
    Node *y = x->right;
    x->right = y->left;
    if(y->left != NIL){
        y->left->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == NIL){
        root = y;
    }
    else if(x == x->parent->left){
        x->parent->left = y;
    }
    else if(x == x->parent->right){
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
};
```

- Right rotation

```
void rightRotation(Node *x){
    Node *y = x->left;
    x->left = y->right;
    if(y->right != NIL){
        y->right->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == NIL){
        root = y;
    }
    else if(x == x->parent->left){
        x->parent->left = y;
    }
    else if(x == x->parent->right){
        x->parent->right = y;
    }
    y->right = x;
    x->parent = y;
};
```

- Main function:
    - 先宣告兩個陣列分別儲存 insert 的數字和 delete 的數字，進行讀取執行次數，宣告一個 RBT。

```
int main(void){
    int tree_insert[1000];
    int tree_delete[1000];

    int times;
    int op;// f the number of function
    int n;//n the times of input num.

    cin>>times;

    RBTree *RBT = new RBTree;
```

    - 讀取 operation 的種類和要插入的數字
    - 當 operation 為 insert

```
for(int p=0;p<times;p++){
    cin>>op>>n;
    if(op==1)
    {
        int cnt1=0;

        for(int i=0; i<n; i++)
        {
            int num;
            cin>>num;
            tree_insert[i]=num;
            cnt1=cnt1+1;
        }
```

■ 當迴圈讀取完畢，將 insert 的結果印出來

```cpp
cout<<"Insert: ";
for(int i=0;i<cnt1-1;i++)
{
    cout<<tree_insert[i]<<", ";
    RBT->insertRBTree(tree_insert[i]);
}
cout<<tree_insert[cnt1-1]<<endl;
RBT->insertRBTree(tree_insert[cnt1-1]);

Node *current = new Node;
current = RBT->root;
while(current != RBT->NIL && current->left != RBT->NIL)
{
    current = current->left;
}
while(current != RBT->NIL)
{
    cout<<"key: "<<current->key<<" parent: ";
    if(current->parent == RBT->NIL)
    {
        cout<<" "<<" color: ";
    }
    else
    {
        cout<<current->parent->key<<" color: ";
    }
    if(current->color==false)
    {
        cout<<"red"<<endl;
    }
    else if(current->color==true)
    {
        cout<<"black"<<endl;
    }
    current = RBT->successor(current);
}
```

- 當 operation 為 delete

```cpp
else if(op==2)
{
    int cnt2=0;

    for(int i=0;i<n;i++)
    {
        int num2;
        cin>>num2;
        tree_delete[i]=num2;
        cnt2=cnt2+1;
    }
```

- 當讀取完畢，將 delete 後的答案印出

```cpp
cout<<"Delete: ";
for(int i=0;i<cnt2-1;i++)
{
    cout<<tree_delete[i]<<", ";
    RBT->deleteRBTree(tree_delete[i]);
}
cout<<tree_delete[cnt2-1]<<endl;
RBT->deleteRBTree(tree_delete[cnt2-1]);

//print_inoder
Node *current = new Node;
current = RBT->root;
while(current != RBT->NIL && current->left != RBT->NIL)
{
    current = current->left;
}
while(current != RBT->NIL)
{
    cout<<"key: "<<current->key<<" parent: ";
    if(current->parent == RBT->NIL)
    {
        cout<<" "<<" color: ";
    }
    else
    {
        cout<<current->parent->key<<" color: ";
    }
    if(current->color==false)
    {
        cout<<"red"<<endl;
    }
    else if(current->color==true)
    {
        cout<<"black"<<endl;
    }
    current = RBT->successor(current);
}
```

b. Anything you want to share

這份作業能夠結合演算法概論這門課程中學習到的紅黑樹的資料結構，並且創建紅黑樹，對於紅黑樹的功能進行實作包刮建立、搜尋、插入、刪除等動作，讓我學到很多，未來有機會將會更進一步的學習相關知識並且嘗試實作。