演算法概論 HW2

109550030 李寀菱

Environment

DEV-C++ 5.11

Results

1. Method

實作具有插入和刪除節點功能的紅黑樹，必須注意由於紅黑樹是特化的一種 Binary Search Tree，因此任何在樹上進行的操作都必須保證紅黑樹的平衡、並且同時符合其對於顏色的特殊要求，包含

（1） 所有節點顏色皆為紅色或黑色，但根（Root）必須為黑色，葉子（Leaf）也必須是黑色。

（2） 每個紅色節點的兩個子節點都須為黑色。

（3） 從每一個任意節點到底端葉子的所有路徑上必須包含相同數量的黑色節點。

因此在插入或刪除節點後，要設計用於顏色調換的方法以將被打亂的紅黑樹調整為符合規則的狀態。

2. Solution

首先建構出紅黑樹的 class，並將其分為節點以及操作節點的函式兩個部分，接著把所有插入和刪除節點所需的函式一一列出。

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <stdlib.h>

using namespace std;
ofstream out;

class RBT;
class RBT_node {/*node constructor*/
private:
    int key;
    int color;/*red=0, black=1*/
    RBT_node* leftchild;
    RBT_node* rightchild;
    RBT_node* parent;
    friend class RBT;

public:
    RBT_node() :leftchild(0), rightchild(0), parent(0), key(0), color(0) {}
    RBT_node(int key) :leftchild(0), rightchild(0), parent(0), key(key), color(0) {}

    //int GetKey()const { return key; };
};

class RBT {
private:
    RBT_node* root;
    RBT_node* N;/*NULL*/

    void LeftRotation(RBT_node* current);/*左旋*/
    void RightRotation(RBT_node* current);/*右旋*/
    void Insert_Fix(RBT_node* current);/*調整顏色*/
    void Delete_Fix(RBT_node* current);/*調整顏色*/

    RBT_node* successor(RBT_node* current);/*找successor*/
    RBT_node* predecessor(RBT_node* current);/*找predecessor*/
```

```
public:
    RBT() {/*construct 紅黑樹*/
        N = new RBT_node;/*NULL is a RBT node*/
        N->color = 1;/*顏色為黑*/
        root = N;/*設作root*/
        root->parent = N;/*root的parent為NULL*/
    };

    void InsertRBT(int key);
    void DeleteRBT(int key);
    void InOrder(RBT_node* current);
    void Output_RBT(RBT_node* current);

    RBT_node* search(int key);
    RBT_node* getRoot();
};
```

刪除節點之函式中，先簡單將問題分成需要刪除的節點有兩個子節點、或者擁有少於兩個的子節點，若該節點沒有任何子節點，那麼便直接刪除該節點；若節點擁有單邊的 subtree，那就將要刪除的節點記為刪除，並用 delete_node_child 記錄其子節點；若是欲刪除的節點具有兩個子節點，則先尋找該節點的 successor 並將其「假設」為我們要刪除的節點。

接著 delete_node_child 與欲刪除節點的父節點連接，並根據刪除節點的位置指定 delete_node_child 為左邊或右邊的 subtree。此時再來處理前面為了方便處理而「假設」的節點，將該節點的值直接覆蓋到真正要刪除的點上。

最後判斷顏色需要重新被調整的問題，若被刪除的節點顏色為黑色，那麼刪除了此節點後就會影響到通過該點的所有路徑都少了一個黑色節點，可能違反規則三，因此需要重新考慮平衡。

```
void RBT::DeleteRBT(int key) {
    RBT_node* remove = search(key);/*要delete的node*/

    if (remove == NULL) {/*key不存在*/
        cout << "Didn't found the data" << endl;
        return;
    }

    RBT_node* delete_node = 0;/*紀錄要delete的node*/
    RBT_node* delete_node_child = 0;/*要delete的node的child*/

    if (remove->leftchild == N || remove->rightchild == N) {/*if it doesn't have two subtrees, then it's itself successor or predecessor, so no change*/
        delete_node = remove;
    }else{
        delete_node = successor(remove);
    }
```

```
    if (delete_node->leftchild != N) {/*有一邊的subtree*/
        delete_node_child = delete_node->leftchild;
    }else{
        delete_node_child = delete_node->rightchild;
    }

    delete_node_child->parent = delete_node->parent;/*連上delete的node的parent*/

    if (delete_node->parent == N) {/*要delete的是root*/
        this->root = delete_node_child;/*child become new root*/
    }else if (delete_node->parent->leftchild == delete_node) {/*if delete node is leftchild*/
        delete_node->parent->leftchild = delete_node_child;
    }else {/*if delete node is rightchild*/
        delete_node->parent->rightchild = delete_node_child;
    }

    if (delete_node != remove) {/*if delete_node is successor of remove, then let successor key replace remove_node key*/
        remove->key = delete_node->key;
    }

    if (delete_node->color == 1) {
        Delete_Fix(delete_node_child);
    }
}
```

第二個部分是調整刪除節點後的顏色，此時需要考慮四種情況，分別是 delete_node_child 的 uncle 為紅色、delete_node_child 的 uncle 是黑色且其子節點皆為黑色、delete_node_child 的 uncle 為黑色且其右子節點為黑色，以及 delete_node_child 的 uncle 為黑色且其左子節點為黑色。

這部分必須注意的是，第一種情況可能變形為其他案例，因此必須最優先處理，而第三種情況經過處理後則必然會轉型為第四種案例，在條件判斷的順序上必須加以留意。

另外，由於一開始我們假定該節點處於其父節點的左邊，後續處理節點在右邊的狀況時，要注意所有的函式和位置都要左右相反。

```
void RBT::Delete_Fix(RBT_node* current) {
    while (current != root && current->color == 1) {/*If current isn't root and color is black*/

        if (current->parent->leftchild == current) {/*左邊，current=current.p.left*/
            RBT_node* current_u = current->parent->rightchild;
            RBT_node* current_p = current->parent;

            if (current_u->color == 0) {/*case1: uncle is red, it can change it to other cases*/

                current_u->color = 1;/*change uncle color to black*/
                current_p->color = 0;/*change parent color to red*/
                LeftRotation(current_p);/*左旋*/
                current_u = current_p->rightchild;/*指向uncle*/ /*current = current_p->rightchild*/
            }

            if (current_u->leftchild->color == 1 && current_u->rightchild->color == 1) {/*case2: if uncle is black and its two children are black*/
                current_u->color = 0;/*change color to red*/
                current = current_p;/*current指向parent*/
            }
```

```
        else {
            if (current_u->leftchild->color == 0) {/*case3: if uncle is black and its leftchild is red, rightchild is black*/

                current_u->leftchild->color = 1;/*change leftchild's color to black*/
                current_u->color = 0;/*change uncle color to red;*/
                RightRotation(current_u);/*右旋*/
                current_u = current->parent->rightchild;/*指向新的parent*/
            }

            /*case4: uncle is black, and its rightchild is red*/
            current_u->color = current_p->color;/*let uncle become parent's color*/
            current_p->color = 1;/*parent become black*/
            current_u->rightchild->color = 1;/*uncle rightchild color be black*/
            LeftRotation(current_p);/*左旋*/
            current = root;/*指向root以脫離迴圈*/
        }
    }

    else {/*右邊，current=current.p.right*/
        RBT_node* current_u = current->parent->leftchild;
        RBT_node* current_p = current->parent;

        if (current_u->color == 0) {/*case1: uncle is red, it can change it to other cases*/

            current_u->color = 1;/*change uncle color to black*/
            current_p->color = 0;/*change parent color to red*/
            RightRotation(current_p);/*右旋*/
            current_u = current_p->leftchild;/*指向uncle*/
        }

        if (current_u->leftchild->color == 1 && current_u->rightchild->color == 1) {/*case2: if uncle is black and its two children are black*/
            current_u->color = 0;/*change color to red*/
            current = current_p;/*current指向parent*/
        }
```

```
        else {
            if (current_u->rightchild->color == 0) {/*case3: if uncle is black and its rightchild is red, leftchild is black*/

                current_u->rightchild->color = 1;/*change rightchild's color to black*/
                current_u->color = 0;/*change uncle color to red;*/
                LeftRotation(current_u);/*左旋*/
                current_u = current->parent->leftchild;/*指向新的parent*/
            }

            /*case4: uncle is black, and its leftchild is red*/
            current_u->color = current_p->color;/*let uncle become parent's color*/
            current_p->color = 1;
            current_u->leftchild->color = 1;
            RightRotation(current_p);/*右旋*/
            current = root;/*指向root以脫離迴圈*/
        }
    }
}
current->color = 1;/*額外狀況，如果current單純為red或者是root，則變黑*/
}
```

插入節點的函式中，首先以 binary search tree 的操作方式插入並將其設為紅色，因為若是多出一個黑色節點，會導致部分經過此新節點的路徑多出一個黑色節點，這可能會增加實作調整的困難性，因此將其當作紅色來調整會比較便於思考。

```
void RBT::InsertRBT(int key) {
    RBT_node* x = root;
    RBT_node* y = N;
    RBT_node* insert_node = new RBT_node(key);/*a new node*/

    while (x != N) {/*存在root*/
        y = x;/*Record parent*/
        if (insert_node->key < x->key) {/*往左邊*/
            x = x->leftchild;
        }else {/*往右邊*/
            x = x->rightchild;
        }
    }
    insert_node->parent = y;/*連接parent*/
```

```
    if (y == N) {
        this->root = insert_node;
    }else if (insert_node->key < y->key) {/*比parent小為leftchild*/
        y->leftchild = insert_node;
    }else {/*比parent大為rightchild*/
        y->rightchild = insert_node;
    }

    insert_node->leftchild = N;
    insert_node->rightchild = N;
    insert_node->color = 0;/*設成紅色*/
    Insert_Fix(insert_node);/*換顏色*/
}
```

調整插入節點後紅黑樹的顏色，分成三種情況來處理，並且和刪除節點後調整平衡相同，因為一開始預設了該節點的位置，因此後續要注意把旋轉方向及位置全部左右調換。

```
void RBT::Insert_Fix(RBT_node* current) {
    while (current->parent->color == 0) {/*if the parent of insert node is red*/

        if (current->parent->parent->leftchild == current->parent) {/*左邊*/
            RBT_node* record_u = current->parent->parent->rightchild;

            if (record_u->color == 0) {/*case 1: record_u is red*/
                current->parent->color = 1;/*change color to black*/
                record_u->color = 1;
                current->parent->parent->color = 0;
                current = current->parent->parent;
            }else {
                if (current == current->parent->leftchild) {/*case2: if current is left child of parent*/
                    current->parent->color = 1;
                    current->parent->parent->color = 0;
                    RightRotation(current->parent->parent);
                }else {/*case3: if current is right child of record_p*/
                    current = current->parent;
                    LeftRotation(current); // become leftchild case
                    current->parent->color = 1;
                    current->parent->parent->color = 0;
                    RightRotation(current->parent->parent);
                }
            }
```

```
        }else {/*右邊*/
            RBT_node* record_u = current->parent->parent->leftchild;

            if (record_u->color == 0) {/*case 1: record_u is red*/
                current->parent->color = 1;
                record_u->color = 1;
                current->parent->parent->color = 0;
                current = current->parent->parent;
            }else {
                if (current == current->parent->leftchild) {/*case2*/
                    current = current->parent;
                    RightRotation(current);
                }
                current->parent->color = 1;/*case 3*/
                current->parent->parent->color = 0;
                LeftRotation(current->parent->parent);
            }
        }
    }
    root->color = 1;/*root's color must be black*/
}

RBT_node* RBT::getRoot()
{
    return this->root;
}
```

尋訪按照要求使用 inorder 之排序，並使用遞迴函式呼叫，由最左至中、最後尋訪右側。

而輸出則先確定紅黑樹的 root 不為空，並根據要求按順序輸出值、父節點和顏色，且若沒有父節點，則輸出空格。

```
void RBT::InOrder(RBT_node* current) {
    if (current) {/*不為NULL*/
        InOrder(current->leftchild);
        Output_RBT(current);
        InOrder(current->rightchild);
    }
}

void RBT::Output_RBT(RBT_node* current) {/*輸出用*/
    if (current == N) {/*NULL*/
        return;
    }

    cout << "key: " << current->key << " parent: ";
    out << "key: " << current->key << " parent: ";
```

```
    if (current->parent == N) {
        cout << " ";
        out << " ";
    }else {
        cout << current->parent->key;
        out << current->parent->key;
    }

    cout << " color: ";
    out << " color: ";

    if (current->color == 0) {
        cout << "red" << endl;
        out << "red" << endl;
    }else {
        cout << "black" << endl;
        out << "black" << endl;
    }
}
```

```
RBT_node* RBT::predecessor(RBT_node* current) {
    while (current->leftchild != N) {
        current = current->leftchild;
    }
    return current;
}

RBT_node* RBT::successor(RBT_node* current) {
    if (current->rightchild != N) {
        return predecessor(current->rightchild);
    }
}
```

```
RBT_node* RBT::search(int key) {
    RBT_node* current = root;
    while (current != NULL && key != current->key) {
        if (key < current->key) {/*往左邊找*/
            current = current->leftchild;
        }else {/*往右邊找*/
            current = current->rightchild;
        }
    }
    return current;
}
```

左旋及右旋的處理，必須注意移動的節點和它們之間互相連接的關係，只要完成其中一個、另一邊則左右相反。

```
void RBT::LeftRotation(RBT_node* current) {
    RBT_node* current_right = current->rightchild;/*紀錄current's rightchild*/
    current->rightchild = current_right->leftchild;/*把current_right的left subtree接到current的右邊*/

    if (current_right->leftchild != N) {/*若current_right有left subtree，就把parent設成current*/
        current_right->leftchild->parent = current;
    }

    current_right->parent = current->parent;/*把current_right往上提，連接current的parent*/

    if (current->parent == N) {/*if current is root*/
        root = current_right;/*current_right變成root*/
    }else if (current == current->parent->leftchild) {/*current在左邊*/
        current->parent->leftchild = current_right;
    }else {/*current在右邊*/
        current->parent->rightchild = current_right;
    }
    current_right->leftchild = current;/*current變成current_right的left subtree*/
    current->parent = current_right;/*current_right變成current的parent*/
}
```

```
void RBT::RightRotation(RBT_node* current) {/*跟左旋全部相反*/
    RBT_node* current_left = current->leftchild;
    current->leftchild = current_left->rightchild;

    if (current_left->rightchild != N) {
        current_left->rightchild->parent = current;
    }

    current_left->parent = current->parent;

    if (current->parent == N) {
        root = current_left;
    }else if (current->parent->leftchild == current) {
        current->parent->leftchild = current_left;
    }else {
        current->parent->rightchild = current_left;
    }
    current_left->rightchild = current;
    current->parent = current_left;
}
```

主函式部分需要注意的是讀取檔案的格式問題，首先第一個讀入的數字為共有幾次操作需要進行，接著在迴圈內部判斷要進行的動作為插入或刪除節點。

為了方便操作，紀錄插入的數字部分使用 vector，並且利用函式將從檔案讀入的字串轉變成整數，再依序輸出插入的數值和要求的紅黑樹資料。

```
int main() {
    ifstream inFile;
    inFile.open("input.txt");/*Read the input file*/
    out.open("output.txt");

    if (!inFile)/*If file fails*/
    {
        cout << "Read input.txt error" << endl;
        exit(1);
    }

    int test;
    inFile >> test;/*Test number*/
```

```
    RBT rbt;
    while (test--) {
        int op;
        inFile >> op;

        if (op == 1) {/*Insert*/

            string s;
            getline(inFile, s);/*'enter'*/
            getline(inFile, s);/*'the string of number'*/
            istringstream ss(s);/*Make string cut space*/

            string str;
            vector<int>key;/*Create vector and empty it*/
            key.clear();

            while (ss >> str) {/*Give one char to str*/
                int num;
                num = atoi(str.c_str());/*Make char become int*/
                rbt.InsertRBT(num);/*insert into 紅黑樹*/
                key.push_back(num);
            }

            cout << "Insert: ";
            out << "Insert: ";
            for (int i = 0; i < key.size(); i++) {
                cout << key[i];
                out << key[i];
                int k = i;

                if (k < key.size() - 1) {/*Touch the end*/
                    cout << ", ";
                    out << ", ";
                }
            }
            cout << endl;
            out << endl;

            rbt.InOrder(rbt.getRoot());
        }
```

```
        else {/*Delete*/
            string s;
            getline(inFile, s); /*'enter'*/
            getline(inFile, s); /*'the string of number'*/
            istringstream ss(s); /*Make string cut space*/

            string str;
            vector<int>key;
            key.clear();

            while (ss >> str) {
                int num;
                num = atoi(str.c_str()); /*Make char become int*/
                rbt.DeleteRBT(num);
                key.push_back(num);
            }

            cout << "Delete: ";
            out << "Delete: ";
            for (int i = 0; i < key.size(); i++) {
                cout << key[i];
                out << key[i];
                int k = i;

                if (k < key.size() - 1) {/*Touch the end*/
                    cout << ", ";
                    out << ", ";
                }
            }
            cout << endl;
            out << endl;
            rbt.InOrder(rbt.getRoot());
        }
    }
    inFile.close();
    out.close();
}
```

3. 心得

這是演算法概論的第二次作業，相比第一次作業，無論是複雜程度還是規模都直線上升，也花費了比第一次更多上幾倍的時間。

總結來說，紅黑樹就是一種進階版的 binary search tree，其機制和規則都很特殊，也讓我驚訝於這樣簡單的幾條準則就能讓時間複雜度下降到這樣理想的程度，尤其是在寫程式之前、在網路上搜索資料時瞭解了其原理，更是讓我印象深刻。除此之外，這次由於教授在課堂上講解得非常仔細，再加上紅黑樹本身帶有平衡的特色，因此像是左旋和右旋的函式就是左右相反，這些都讓我感到十分新奇、也意外地節省了不少時間。

總而言之，儘管在一些紅黑樹操作的判斷條件上，其邏輯思考真的讓我費了一番功夫，但也是一次十分有趣的經驗，令我對這種特殊的資料結構有了更多認識。