数据结构实验报告 9

红黑树

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 12 月 16 日

一、 题目一

1.1 题目表述

设计一个 C++redBlackTree, 它派生于抽象类 bSTree(程序 14-1)。编写所有函数的代码并检验其正确性。

程序 14-1 C++ 抽象类 bsTree

```
template<class K, class E>
class bsTree : public dictionary<K,E>
{
   public:
      virtual void ascend() = 0;
                  // 按关键字升序输出
};
```

- find(k)：返回键为 k 的数对（这里用 v(k) 表示它的 value），找不到记为 null；
- insert(p)：插入数对 p=(k,v(k))，假设不允许重复键；
- erase(k)：删除键为 k 的数对；
- ascend()：按键升序输出所有键；
- clear()：清除树；
- height()：返回树的高度（空树为 0）。

函数 find、insert 和 erase 的时间复杂性必须是 O(logn),函数 ascend 的时间复杂性应该是 O(n)。证明它们的时间复杂性。

1.2 代码 1 的测试用例:

1. ascend(T) -> () # 空树遍历
2. find(T, 10) -> null # 空树查找
3. insert(T, (30, v30)), ascend(T) -> (30)
4. insert(T, (15, v15)),insert(T, (40, v40)), ascend(T) -> (15, 30, 40)
5. insert(T, (10, v10)),insert(T, (20, v20)),insert(T, (35, v35)),insert(T, (50, v50)), ascend(T) -> (10, 15, 20, 30, 35, 40, 50)
6. find(T, 20) -> (20, v20) # 查找存在键
7. find(T, 17) -> null # 查找不存在键
8. erase(T, 10), ascend(T) -> (15, 20, 30, 35, 40, 50)
   # 删除叶子结点
9. erase(T, 50), ascend(T) -> (15, 20, 30, 35, 40)
   # 删除只有一个子女或叶子结点
10. erase(T, 30),ascend(T), find(T, 30) -> (15, 20, 35, 40), null
    # 删除有两个子女的结点

11. insert(T, (18, v18)),insert(T, (37, v37)),ascend(T) -> (15, 18, 20, 35, 37, 40)
12. erase(T, 18), erase(T, 37),ascend(T) -> (15, 20, 35, 40) # 连续混合插入/删除后保持有序
13. clear(T), insert(T, 1), insert(T, 2), insert(T, 3), insert(T, 4), insert(T, 5), insert(T, 6), ascend(T) -> (1, 2, 3, 4, 5, 6)
14. height(T) -> 3 或者 4

时间复杂度的证明可以从每次访问进行的节点数目进行分析。

1.3 思路:

本着维持红黑树的平衡规则，继承抽象类 bsTree，通过带颜色标记、父指针的节点结构搭建基础，依托左旋/右旋完成树结构调整。find 采用 BST 二分查找定位键值，insert 先执行 BST 插入再通过变色+旋转修复红黑规则，erase 按节点子女数量分情况处理（双子女节点用后继节点替换）并修复黑高平衡，ascend 借助中序遍历实现升序输出，clear 递归释放所有节点内存，height 通过递归计算左右子树最大高度获得。

红黑树是自平衡二叉搜索树，其黑高为 O(log n)，因此所有操作的时间复杂度由树的高度决定。
find(k)：O(log n)
查找过程是二叉搜索树的标准查找，从根到目标节点的路径长度不超过树的高度。红黑树的高度为 O(log n)，因此最多访问 O(log n)个节点，时间复杂度 O(log n)。
insert(p)：O(log n)
BST 插入的路径长度为 O(log n)；
插入后修复（insertFixup）的循环次数不超过 O(log n)（每次循环将节点上移一层，最多到根），且每次旋转/变色是常数时间操作。
总时间复杂度 O(log n)。
erase(k)：O(log n)
BST 删除的路径长度为 O(log n)；
删除后修复（deleteFixup）的循环次数不超过 O(log n)（每次循环将节点上移一层，最多到根），且每次旋转/变色是常数时间操作。
总时间复杂度 O(log n)。
ascend()：O(n)
ascend 基于中序遍历实现，需访问树中每个节点恰好一次，因此时间复杂度为 O(n)。
height()：O(n)
递归遍历所有节点计算高度，需访问每个节点一次，时间复杂度 O(n)。

1.4 代码:

```
#include <iostream>
```

```cpp
#include <utility>
#include <vector>
using namespace std;

// 抽象基类：字典（bsTree 的父类）
template<class K, class E>
class dictionary {
public:
    virtual E* find(const K& k) = 0;
    virtual void insert(const pair<K, E>& p) = 0;
    virtual void erase(const K& k) = 0;
    virtual void clear() = 0;
    virtual ~dictionary() = default;
};

// 抽象类 bsTree（继承自 dictionary）
template<class K, class E>
class bsTree : public dictionary<K, E> {
public:
    virtual void ascend() = 0;
    virtual int height() = 0;
    virtual ~bsTree() = default;
};

// 红黑树节点定义
template<class K, class E>
struct RBNode {
    pair<K, E> data;
    RBNode* left;
    RBNode* right;
    RBNode* parent;
    bool isRed;  // true: 红, false: 黑
    RBNode(const pair<K, E>& p) : data(p), left(nullptr),
right(nullptr), parent(nullptr), isRed(true) {}
};

// 红黑树实现（继承 bsTree）
template<class K, class E>
class redBlackTree : public bsTree<K, E> {
private:
    RBNode<K, E>* root;
    int nodeCount;  // 节点总数
```

```
    // 左旋
    void leftRotate(RBNode<K, E>* x) {
        RBNode<K, E>* y = x->right;
        x->right = y->left;
        if (y->left != nullptr) y->left->parent = x;
        y->parent = x->parent;
        if (x->parent == nullptr) root = y;
        else if (x == x->parent->left) x->parent->left = y;
        else x->parent->right = y;
        y->left = x;
        x->parent = y;
    }

    // 右旋
    void rightRotate(RBNode<K, E>* y) {
        RBNode<K, E>* x = y->left;
        y->left = x->right;
        if (x->right != nullptr) x->right->parent = y;
        x->parent = y->parent;
        if (y->parent == nullptr) root = x;
        else if (y == y->parent->right) y->parent->right = x;
        else y->parent->left = x;
        x->right = y;
        y->parent = x;
    }

    // 插入后修复红黑树
    void insertFixup(RBNode<K, E>* z) {
        while (z->parent != nullptr && z->parent->isRed) {
            if (z->parent == z->parent->parent->left) {  // 父是
左孩子
                RBNode<K, E>* y = z->parent->parent->right;  //
叔叔
                if (y != nullptr && y->isRed) {  // 叔叔红：变色
                    z->parent->isRed = false;
                    y->isRed = false;
                    z->parent->parent->isRed = true;
                    z = z->parent->parent;
                } else {  // 叔叔黑：旋转+变色
                    if (z == z->parent->right) {  // z 是右孩子：
先左旋
```

```
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->isRed = false;
                z->parent->parent->isRed = true;
                rightRotate(z->parent->parent);
            }
        } else {  // 父是右孩子（镜像操作）
            RBNode<K, E>* y = z->parent->parent->left;
            if (y != nullptr && y->isRed) {
                z->parent->isRed = false;
                y->isRed = false;
                z->parent->parent->isRed = true;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->isRed = false;
                z->parent->parent->isRed = true;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->isRed = false;  // 根始终为黑
}

// 查找节点（辅助函数）
RBNode<K, E>* findNode(const K& k) const {
    RBNode<K, E>* cur = root;
    while (cur != nullptr) {
        if (k == cur->data.first) return cur;
        else if (k < cur->data.first) cur = cur->left;
        else cur = cur->right;
    }
    return nullptr;
}

// 中序遍历（辅助 ascend）
void inOrder(RBNode<K, E>* node, vector<K>& keys) const {
    if (node == nullptr) return;
```

```
        inOrder(node->left, keys);
        keys.push_back(node->data.first);
        inOrder(node->right, keys);
    }

    // 计算树高（辅助 height）
    int calcHeight(RBNode<K, E>* node) const {
        if (node == nullptr) return 0;
        int leftH = calcHeight(node->left);
        int rightH = calcHeight(node->right);
        return max(leftH, rightH) + 1;
    }

    // 删除节点后修复红黑树
    void deleteFixup(RBNode<K, E>* x) {
        while (x != root && (x == nullptr || !x->isRed)) {
            if (x == x->parent->left) {  // x 是左孩子
                RBNode<K, E>* w = x->parent->right;  // 兄弟
                if (w->isRed) {  // 兄弟红：变色+左旋
                    w->isRed = false;
                    x->parent->isRed = true;
                    leftRotate(x->parent);
                    w = x->parent->right;
                }
                // 兄弟的两个孩子都黑
                if ((w->left == nullptr || !w->left->isRed) &&
(w->right == nullptr || !w->right->isRed)) {
                    w->isRed = true;
                    x = x->parent;
                } else {
                    if (w->right == nullptr || !w->right->isRed)
{  // 兄弟右孩子黑：左孩子红
                        w->left->isRed = false;
                        w->isRed = true;
                        rightRotate(w);
                        w = x->parent->right;
                    }
                    // 兄弟右孩子红
                    w->isRed = x->parent->isRed;
                    x->parent->isRed = false;
                    w->right->isRed = false;
                    leftRotate(x->parent);
```

```
                    x = root;  // 退出循环
                }
            } else {  // x 是右孩子（镜像操作）
                RBNode<K, E>* w = x->parent->left;
                if (w->isRed) {
                    w->isRed = false;
                    x->parent->isRed = true;
                    rightRotate(x->parent);
                    w = x->parent->left;
                }
                if ((w->right == nullptr || !w->right->isRed) &&
(w->left == nullptr || !w->left->isRed)) {
                    w->isRed = true;
                    x = x->parent;
                } else {
                    if (w->left == nullptr || !w->left->isRed) {
                        w->right->isRed = false;
                        w->isRed = true;
                        leftRotate(w);
                        w = x->parent->left;
                    }
                    w->isRed = x->parent->isRed;
                    x->parent->isRed = false;
                    w->left->isRed = false;
                    rightRotate(x->parent);
                    x = root;
                }
            }
        }
        if (x != nullptr) x->isRed = false;
    }

    // 释放节点（辅助 clear）
    void destroy(RBNode<K, E>* node) {
        if (node == nullptr) return;
        destroy(node->left);
        destroy(node->right);
        delete node;
    }

public:
    redBlackTree() : root(nullptr), nodeCount(0) {}
```

```
    ~redBlackTree() { clear(); }

    // 查找：返回键 k 对应的值，找不到返回 null
    E* find(const K& k) override {
        RBNode<K, E>* node = findNode(k);
        return (node != nullptr) ? &(node->data.second) :
nullptr;
    }

    // 插入：不允许重复键
    void insert(const pair<K, E>& p) override {
        if (find(p.first) != nullptr) return;  // 键已存在，不插
入

        RBNode<K, E>* z = new RBNode<K, E>(p);
        RBNode<K, E>* y = nullptr;
        RBNode<K, E>* x = root;

        // BST 插入
        while (x != nullptr) {
            y = x;
            if (z->data.first < x->data.first) x = x->left;
            else x = x->right;
        }
        z->parent = y;
        if (y == nullptr) root = z;  // 空树
        else if (z->data.first < y->data.first) y->left = z;
        else y->right = z;

        nodeCount++;
        insertFixup(z);  // 修复红黑树
    }

    // 删除：删除键 k 对应的节点
    void erase(const K& k) override {
        RBNode<K, E>* z = findNode(k);
        if (z == nullptr) return;  // 键不存在

        RBNode<K, E>* y = z;
        RBNode<K, E>* x = nullptr;
        bool yOriginalColor = y->isRed;
```

```
        // 情况 1：z 只有右孩子
        if (z->left == nullptr) {
            x = z->right;
            transplant(z, z->right);
        }
        // 情况 2：z 只有左孩子
        else if (z->right == nullptr) {
            x = z->left;
            transplant(z, z->left);
        }
        // 情况 3：z 有两个孩子
        else {
            y = minimum(z->right);  // 后继节点
            yOriginalColor = y->isRed;
            x = y->right;
            if (y->parent == z) {
                if (x != nullptr) x->parent = y;
            } else {
                transplant(y, y->right);
                y->right = z->right;
                y->right->parent = y;
            }
            transplant(z, y);
            y->left = z->left;
            y->left->parent = y;
            y->isRed = z->isRed;
        }
        delete z;
        nodeCount--;
        if (!yOriginalColor) {  // 若删除的是黑节点，修复红黑树
            if (x != nullptr) deleteFixup(x);
            else if (root != nullptr)
deleteFixup(root->left);  // 处理 x 为 null 的情况
        }
    }

    // 按升序输出所有键
    void ascend() override {
        vector<K> keys;
        inOrder(root, keys);
        cout << "(";
        for (size_t i = 0; i < keys.size(); ++i) {
```

```
            if (i > 0) cout << ", ";
            cout << keys[i];
        }
        cout << ")" << endl;
    }

    // 清空树
    void clear() override {
        destroy(root);
        root = nullptr;
        nodeCount = 0;
    }

    // 返回树的高度（空树为 0）
    int height() override {
        return calcHeight(root);
    }

    // 辅助函数：替换子树（红黑树删除用）
    void transplant(RBNode<K, E>* u, RBNode<K, E>* v) {
        if (u->parent == nullptr) root = v;
        else if (u == u->parent->left) u->parent->left = v;
        else u->parent->right = v;
        if (v != nullptr) v->parent = u->parent;
    }

    // 辅助函数：找子树的最小节点（红黑树删除用）
    RBNode<K, E>* minimum(RBNode<K, E>* node) const {
        while (node->left != nullptr) node = node->left;
        return node;
    }
};


// 测试用例
int main() {
    redBlackTree<int, string> T;

    // 1. 空树遍历
    cout << "1. ascend(T) -> ";
    T.ascend();
```

```
    // 2. 空树查找
    cout << "2. find(T, 10) -> ";
    if (T.find(10) == nullptr) cout << "null" << endl;
    else cout << "(" << 10 << ", " << *T.find(10) << ")" <<
endl;

    // 3. 插入 30 后遍历
    T.insert({30, "v30"});
    cout << "3. ascend(T) -> ";
    T.ascend();

    // 4. 插入 15、40 后遍历
    T.insert({15, "v15"});
    T.insert({40, "v40"});
    cout << "4. ascend(T) -> ";
    T.ascend();

    // 5. 插入 10、20、35、50 后遍历
    T.insert({10, "v10"});
    T.insert({20, "v20"});
    T.insert({35, "v35"});
    T.insert({50, "v50"});
    cout << "5. ascend(T) -> ";
    T.ascend();

    // 6. 查找存在的键 20
    cout << "6. find(T, 20) -> ";
    if (T.find(20) != nullptr) cout << "(20, " << *T.find(20)
<< ")" << endl;
    else cout << "null" << endl;

    // 7. 查找不存在的键 17
    cout << "7. find(T, 17) -> ";
    if (T.find(17) == nullptr) cout << "null" << endl;
    else cout << "(17, " << *T.find(17) << ")" << endl;

    // 8. 删除叶子 10 后遍历
    T.erase(10);
    cout << "8. ascend(T) -> ";
    T.ascend();

    // 9. 删除叶子 50 后遍历
```

```
    T.erase(50);
    cout << "9. ascend(T) -> ";
    T.ascend();

    // 10. 删除有两个子女的 30 后遍历+查找
    T.erase(30);
    cout << "10. ascend(T) -> ";
    T.ascend();
    cout << "    find(T, 30) -> ";
    if (T.find(30) == nullptr) cout << "null" << endl;
    else cout << "(30, " << *T.find(30) << ")" << endl;

    // 11. 插入 18、37 后遍历
    T.insert({18, "v18"});
    T.insert({37, "v37"});
    cout << "11. ascend(T) -> ";
    T.ascend();

    // 12. 删除 18、37 后遍历
    T.erase(18);
    T.erase(37);
    cout << "12. ascend(T) -> ";
    T.ascend();

    // 13. 清空后插入 1-6 再遍历
    T.clear();
    T.insert({1, "v1"});
    T.insert({2, "v2"});
    T.insert({3, "v3"});
    T.insert({4, "v4"});
    T.insert({5, "v5"});
    T.insert({6, "v6"});
    cout << "13. ascend(T) -> ";
    T.ascend();

    // 14. 输出树高
    cout << "14. height(T) -> " << T.height() << endl;

    return 0;
}
```

1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw9> cd hw9.1
PS C:\Users\Macro\Desktop\hw9\hw9.1> .\RBT.exe
1. ascend(T) -> ()
2. find(T, 10) -> null
3. ascend(T) -> (30)
4. ascend(T) -> (15, 30, 40)
5. ascend(T) -> (10, 15, 20, 30, 35, 40, 50)
6. find(T, 20) -> (20, v20)
7. find(T, 17) -> null
8. ascend(T) -> (15, 20, 30, 35, 40, 50)
9. ascend(T) -> (15, 20, 30, 35, 40)
10. ascend(T) -> (15, 20, 35, 40)
    find(T, 30) -> null
11. ascend(T) -> (15, 18, 20, 35, 37, 40)
12. ascend(T) -> (15, 20, 35, 40)
13. ascend(T) -> (1, 2, 3, 4, 5, 6)
14. height(T) -> 4
PS C:\Users\Macro\Desktop\hw9\hw9.1>
```

二、 题目二

2.1 题目表述

设计一个 C++类 dRedBlackTree,它派生于抽象类 dBSTree(即有重复值的二叉搜索树)。编写所有函数的代码并检验其正确性。函数 find、insert 和 erase 必须具有复杂性 O(logn),函数 ascend 的时间复杂性应该是 O(n)。证明它们的时间复杂性。允许重复键。

操作定义：

- find(k)：返回键为 k 的数对（这里用 v(k) 表示），若有多个返回其中任意一个，找不到记为 null；
- insert(p)：插入数对 p=(k,v(k))，允许重复键（Duplicate Keys）；
- erase(k)：删除键为 k 的数对，若有多个重复键，仅删除其中一个；
- ascend()：按键升序输出所有键（重复键也全部输出）；
- clear()：清空树；
- height()：返回树的高度（空树为 0）。

2.2 代码 2 的测试用例:

ascend(T) -> () # 1. 空树遍历

find(T, 10) -> null # 2. 空树查找

insert(T, (20, v20)), insert(T, (10, v10)), insert(T, (30, v30)), ascend(T) -> (10, 20, 30)

insert(T, (10, v10_2)), insert(T, (10, v10_3)), ascend(T) -> (10, 10, 10, 20, 30) # 3. 插入重复键：应存在 3 个 10

find(T, 10) -> (10, v?) # 4. 查找重复键：返回非 null，问号是通配符

erase(T, 10), ascend(T) -> (10, 10, 20, 30) # 5. 部分删除：只删一个 10，还剩两个

erase(T, 10), erase(T, 10), ascend(T) -> (20, 30) # 6. 完全删除：删光剩余的 10

find(T, 10) -> null # 7. 验证删除干净

clear(T), insert(T, (5, A)), insert(T, (5, B)), insert(T, (5, C)), insert(T, (3, D)), insert(T, (8, E)), ascend(T) -> (3, 5, 5, 5, 8)

erase(T, 5), ascend(T) -> (3, 5, 5, 8) # 8. 混合场景：带重复键的删除维持有序

clear(T), insert(T, 1), insert(T, 2), insert(T, 3), insert(T, 4), insert(T, 5), insert(T, 6), ascend(T) -> (1, 2, 3, 4, 5, 6)

height(T) -> 3 或者 4

clear(T), insert(T, 50), insert(T, 40), insert(T, 30), insert(T, 20), insert(T, 10), ascend(T) -> (10, 20, 30, 40, 50)

height(T) -> 3 或者 4

时间复杂度的证明可以从每次访问进行的节点数目进行分析。

2.3 思路:

基于红黑树规则，修改 BST 插入/删除逻辑以支持重复键（重复键统一插入到右子树），其余平衡修复、遍历等逻辑与普通红黑树一致。

时间复杂度证明：
核心依据：红黑树的 黑高为 O(log n)，树高为 O(log n)，重复键不改变树的平衡特性（仅增加节点数量，不增加树高的阶数）。
find(k)：O(log n)二分查找路径长度 = 树高 O(log n)，找到任意一个键为 k 的节点即返回，无额外操作。
insert(p)：O(log n)BST 插入路径长度 = 树高 O(log n)；插入后修复（insertFixup）循环次数不超过树高，旋转/变色为常数操作，总复杂度 O(log n)。
erase(k)：O(log n)查找待删除节点的路径长度 = 树高 O(log n)；删除后修复（deleteFixup）循环次数不超过树高，旋转/变色为常数操作，总复杂度 O(log n)。
ascend()：O(n)中序遍历访问所有节点（包括重复键），每个节点访问一次，复杂度 O(n)。

2.4 代码:

```
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

// 抽象基类：字典（dBSTree 的父类）
template<class K, class E>
class dictionary {
public:
    virtual E* find(const K& k) = 0;
    virtual void insert(const pair<K, E>& p) = 0;
    virtual void erase(const K& k) = 0;
    virtual void clear() = 0;
    virtual ~dictionary() = default;
};

// 抽象类 dBSTree（支持重复键的二叉搜索树）
template<class K, class E>
class dBSTree : public dictionary<K, E> {
public:
    virtual void ascend() = 0;
    virtual int height() = 0;
    virtual ~dBSTree() = default;
};
```

```
// 红黑树节点定义（同普通红黑树，无修改）
template<class K, class E>
struct RBNode {
    pair<K, E> data;
    RBNode* left;
    RBNode* right;
    RBNode* parent;
    bool isRed;  // true:红，false:黑
    RBNode(const pair<K, E>& p) : data(p), left(nullptr),
right(nullptr), parent(nullptr), isRed(true) {}
};

// 带重复键的红黑树实现
template<class K, class E>
class dRedBlackTree : public dBSTree<K, E> {
private:
    RBNode<K, E>* root;
    int nodeCount;

    // 左旋（复用普通红黑树逻辑）
    void leftRotate(RBNode<K, E>* x) {
        RBNode<K, E>* y = x->right;
        x->right = y->left;
        if (y->left) y->left->parent = x;
        y->parent = x->parent;
        if (!x->parent) root = y;
        else if (x == x->parent->left) x->parent->left = y;
        else x->parent->right = y;
        y->left = x;
        x->parent = y;
    }

    // 右旋（复用普通红黑树逻辑）
    void rightRotate(RBNode<K, E>* y) {
        RBNode<K, E>* x = y->left;
        y->left = x->right;
        if (x->right) x->right->parent = y;
        x->parent = y->parent;
        if (!y->parent) root = x;
        else if (y == y->parent->right) y->parent->right = x;
        else y->parent->left = x;
```

```
    x->right = y;
    y->parent = x;
}

// 插入后修复（复用普通红黑树逻辑）
void insertFixup(RBNode<K, E>* z) {
    while (z->parent && z->parent->isRed) {
        if (z->parent == z->parent->parent->left) {
            RBNode<K, E>* y = z->parent->parent->right;
            if (y && y->isRed) {
                z->parent->isRed = false;
                y->isRed = false;
                z->parent->parent->isRed = true;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->isRed = false;
                z->parent->parent->isRed = true;
                rightRotate(z->parent->parent);
            }
        } else {
            RBNode<K, E>* y = z->parent->parent->left;
            if (y && y->isRed) {
                z->parent->isRed = false;
                y->isRed = false;
                z->parent->parent->isRed = true;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->isRed = false;
                z->parent->parent->isRed = true;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->isRed = false;
```

```
    }

    // 查找节点（返回任意一个键为 k 的节点，支持重复键）
    RBNode<K, E>* findNode(const K& k) const {
        RBNode<K, E>* cur = root;
        while (cur) {
            if (k == cur->data.first) return cur;  // 找到任意一
个即返回
            else if (k < cur->data.first) cur = cur->left;
            else cur = cur->right;
        }
        return nullptr;
    }

    // 查找待删除的节点（找最左侧的 k 节点，确保删除一个且不破坏有序
性）
    RBNode<K, E>* findEraseNode(const K& k) const {
        RBNode<K, E>* cur = root;
        RBNode<K, E>* target = nullptr;
        while (cur) {
            if (k == cur->data.first) {
                target = cur;  // 记录第一个找到的节点（最左侧）
                cur = cur->left;  // 继续向左找，确保删除最左侧的 k
（不影响其他重复键）
            } else if (k < cur->data.first) {
                cur = cur->left;
            } else {
                cur = cur->right;
            }
        }
        return target;
    }

    // 中序遍历（输出所有键，包括重复键）
    void inOrder(RBNode<K, E>* node, vector<K>& keys) const {
        if (!node) return;
        inOrder(node->left, keys);
        keys.push_back(node->data.first);
        inOrder(node->right, keys);
    }

    // 计算树高
```

```
    int calcHeight(RBNode<K, E>* node) const {
        if (!node) return 0;
        return max(calcHeight(node->left),
calcHeight(node->right)) + 1;
    }

    // 删除后修复（复用普通红黑树逻辑）
    void deleteFixup(RBNode<K, E>* x) {
        while (x != root && (!x || !x->isRed)) {
            if (x == x->parent->left) {
                RBNode<K, E>* w = x->parent->right;
                if (w->isRed) {
                    w->isRed = false;
                    x->parent->isRed = true;
                    leftRotate(x->parent);
                    w = x->parent->right;
                }
                if ((!w->left || !w->left->isRed) && (!w->right
|| !w->right->isRed)) {
                    w->isRed = true;
                    x = x->parent;
                } else {
                    if (!w->right || !w->right->isRed) {
                        w->left->isRed = false;
                        w->isRed = true;
                        rightRotate(w);
                        w = x->parent->right;
                    }
                    w->isRed = x->parent->isRed;
                    x->parent->isRed = false;
                    w->right->isRed = false;
                    leftRotate(x->parent);
                    x = root;
                }
            } else {
                RBNode<K, E>* w = x->parent->left;
                if (w->isRed) {
                    w->isRed = false;
                    x->parent->isRed = true;
                    rightRotate(x->parent);
                    w = x->parent->left;
                }
```

```
                if ((!w->right || !w->right->isRed) && (!w->left
|| !w->left->isRed)) {
                    w->isRed = true;
                    x = x->parent;
                } else {
                    if (!w->left || !w->left->isRed) {
                        w->right->isRed = false;
                        w->isRed = true;
                        leftRotate(w);
                        w = x->parent->left;
                    }
                    w->isRed = x->parent->isRed;
                    x->parent->isRed = false;
                    w->left->isRed = false;
                    rightRotate(x->parent);
                    x = root;
                }
            }
        }
        if (x) x->isRed = false;
    }

    // 释放节点（清空树）
    void destroy(RBNode<K, E>* node) {
        if (!node) return;
        destroy(node->left);
        destroy(node->right);
        delete node;
    }

public:
    dRedBlackTree() : root(nullptr), nodeCount(0) {}
    ~dRedBlackTree() { clear(); }

    // 查找：返回任意一个键为 k 的值，找不到返回 null
    E* find(const K& k) override {
        RBNode<K, E>* node = findNode(k);
        return node ? &(node->data.second) : nullptr;
    }

    // 插入：支持重复键（重复键插入到右子树）
    void insert(const pair<K, E>& p) override {
```

```
        RBNode<K, E>* z = new RBNode<K, E>(p);
        RBNode<K, E>* y = nullptr;
        RBNode<K, E>* x = root;

        // BST 插入：k <= 当前节点键时，继续查找左子树；k > 时查找右子树（重复键插入右子树）
        while (x) {
            y = x;
            if (z->data.first < x->data.first) {
                x = x->left;
            } else {  // 关键修改：重复键（==）插入到右子树
                x = x->right;
            }
        }
        z->parent = y;
        if (!y) root = z;
        else if (z->data.first < y->data.first) y->left = z;
        else y->right = z;

        nodeCount++;
        insertFixup(z);  // 修复红黑树平衡
    }

    // 删除：删除任意一个键为 k 的节点（优先删最左侧的 k）
    void erase(const K& k) override {
        RBNode<K, E>* z = findEraseNode(k);
        if (!z) return;  // 无该键，直接返回

        RBNode<K, E>* y = z;
        RBNode<K, E>* x = nullptr;
        bool yOriginalColor = y->isRed;

        // 按子女数量分情况替换（同普通红黑树）
        if (!z->left) {
            x = z->right;
            transplant(z, z->right);
        } else if (!z->right) {
            x = z->left;
            transplant(z, z->left);
        } else {
            y = minimum(z->right);
            yOriginalColor = y->isRed;
```

```
            x = y->right;
            if (y->parent == z) {
                if (x) x->parent = y;
            } else {
                transplant(y, y->right);
                y->right = z->right;
                y->right->parent = y;
            }
            transplant(z, y);
            y->left = z->left;
            y->left->parent = y;
            y->isRed = z->isRed;
        }

        delete z;
        nodeCount--;
        if (!yOriginalColor) {
            if (x != nullptr) deleteFixup(x);
            else if (root) deleteFixup(root);  // 处理 x 为 null 的
情况
        }
    }

    // 升序输出所有键（包括重复键）
    void ascend() override {
        vector<K> keys;
        inOrder(root, keys);
        cout << "(";
        for (size_t i = 0; i < keys.size(); ++i) {
            if (i > 0) cout << ", ";
            cout << keys[i];
        }
        cout << ")" << endl;
    }

    // 清空树
    void clear() override {
        destroy(root);
        root = nullptr;
        nodeCount = 0;
    }
```

```
    // 返回树高（空树为 0）
    int height() override {
        return calcHeight(root);
    }

    // 辅助函数：替换子树
    void transplant(RBNode<K, E>* u, RBNode<K, E>* v) {
        if (!u->parent) root = v;
        else if (u == u->parent->left) u->parent->left = v;
        else u->parent->right = v;
        if (v) v->parent = u->parent;
    }

    // 辅助函数：找子树最小节点（后继节点）
    RBNode<K, E>* minimum(RBNode<K, E>* node) const {
        while (node->left) node = node->left;
        return node;
    }
};


// 测试用例
int main() {
    dRedBlackTree<int, string> T;

    // 1. 空树遍历
    cout << "1. ascend(T) -> ";
    T.ascend();

    // 2. 空树查找
    cout << "2. find(T, 10) -> ";
    cout << (T.find(10) ? "非 null" : "null") << endl;

    // 3. 插入 20、10、30、两个 10（共 3 个 10）
    T.insert({20, "v20"});
    T.insert({10, "v10"});
    T.insert({30, "v30"});
    T.insert({10, "v10_2"});
    T.insert({10, "v10_3"});
    cout << "3. ascend(T) -> ";
    T.ascend();
```

```
// 4. 查找重复键 10
cout << "4. find(T, 10) -> ";
string* val = T.find(10);
if (val) cout << "(10, " << *val << ")" << endl;
else cout << "null" << endl;

// 5. 删除一个 10（剩两个 10）
T.erase(10);
cout << "5. ascend(T) -> ";
T.ascend();

// 6. 再删除两个 10（删光）
T.erase(10);
T.erase(10);
cout << "6. ascend(T) -> ";
T.ascend();

// 7. 验证 10 已删除
cout << "7. find(T, 10) -> ";
cout << (T.find(10) ? "非 null" : "null") << endl;

// 8. 混合场景：插入 3、三个 5、8，删除一个 5
T.clear();
T.insert({5, "A"});
T.insert({5, "B"});
T.insert({5, "C"});
T.insert({3, "D"});
T.insert({8, "E"});
cout << "8. after insert ascend(T) -> ";
T.ascend();
T.erase(5);
cout << "   after erase ascend(T) -> ";
T.ascend();

// 9. 插入 1-6（无重复），输出并查高
T.clear();
T.insert({1, "v1"});
T.insert({2, "v2"});
T.insert({3, "v3"});
T.insert({4, "v4"});
T.insert({5, "v5"});
T.insert({6, "v6"});
```

```
    cout << "9. ascend(T) -> ";
    T.ascend();
    cout << "   height(T) -> " << T.height() << endl;

    // 10. 逆序插入 10-50（无重复），输出并查高
    T.clear();
    T.insert({50, "v50"});
    T.insert({40, "v40"});
    T.insert({30, "v30"});
    T.insert({20, "v20"});
    T.insert({10, "v10"});
    cout << "10. ascend(T) -> ";
    T.ascend();
    cout << "    height(T) -> " << T.height() << endl;

    return 0;
}
```

2.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw9\hw9.2> .\DRBT.exe
1. ascend(T) -> ()
2. find(T, 10) -> null
3. ascend(T) -> (10, 10, 10, 20, 30)
4. find(T, 10) -> (10, v10_2)
5. ascend(T) -> (10, 10, 20, 30)
6. ascend(T) -> (20, 30)
7. find(T, 10) -> null
8. after insert ascend(T) -> (3, 5, 5, 5, 8)
   after erase ascend(T) -> (3, 5, 5, 8)
9. ascend(T) -> (1, 2, 3, 4, 5, 6)
   height(T) -> 4
10. ascend(T) -> (10, 20, 30, 40, 50)
    height(T) -> 3
PS C:\Users\Macro\Desktop\hw9\hw9.2>
```