数据结构实验报告 8

AVL 树

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 12 月 2 日

一、 题目一

1.1 题目表述

编写 C++函数，计算 AVL 树的高度，要求说明该函数是所有算法中最优的。

1.2 代码 1 的测试用例:

约定空树高度为 0，非空树高度为根节点到叶子节点的最长路径上的节点数，f 为编写的函数,，T 为树。
f(T) -> 0 #空树
insert(T, 30), f(T) -> 1
insert(T, 40), insert(T, 50), f(T) -> 2
insert(T, 20), insert(T, 10), f(T) -> 3
insert(T, 25), f(T) -> 3
insert(T, 60), insert(T, 70), f(T) -> 4
erase(T, 10), erase(T, 25), f(T) -> 3
最优性的证明可以从最坏的情况角度进行分析。

1.3 思路:

函数 f 的核心思路是利用 AVL 树的自平衡特性。
从最坏情况角度分析，计算 AVL 树高度的算法时间复杂度为 O(1)。这是因为 AVL 树是一种自平衡二叉搜索树，在每次插入或删除操作后，树都会自动调整，以保持其平衡性，即任意节点的左右子树高度差不超过 1。因此，每个节点的高度信息在插入或删除操作后都会被更新，存储在节点的 height 属性中。当需要计算树的高度时，只需直接访问根节点的 height 属性即可，无需遍历整棵树。
相比之下，其他一些计算二叉树高度的算法通常需要从根节点开始遍历整棵树，时间复杂度为 O(n)，其中 n 为树中节点的数量。例如，递归遍历左右子树并计算最大深度的算法，在最坏情况下（如退化为链表的二叉树）需要访问所有节点。而 AVL 树由于其自平衡特性，使得高度信息始终可用，无需额外遍历，因此在计算高度方面具有最优的时间复杂度。

1.4 代码:

```
#include <iostream>
#include <algorithm>

// AVL 树的节点结构
struct AVLNode {
    int key; // 节点的键值
    int height; // 节点的高度
```

```
    AVLNode* left; // 左子节点指针
    AVLNode* right; // 右子节点指针

    AVLNode(int k) : key(k), height(1), left(nullptr),
right(nullptr) {} // 构造函数
};

// 辅助函数：计算节点高度
int getHeight(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return node->height;
}

// 辅助函数：更新节点高度
void updateHeight(AVLNode* node) {
    if (node != nullptr) {
        node->height = std::max(getHeight(node->left),
getHeight(node->right)) + 1;
    }
}

// 辅助函数：计算平衡因子
int getBalanceFactor(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return getHeight(node->left) - getHeight(node->right);
}

// 辅助函数：右旋操作
AVLNode* rotateRight(AVLNode* y) {
    AVLNode* x = y->left;
    y->left = x->right;
    x->right = y;

    updateHeight(y);
    updateHeight(x);

    return x;
}
```

```
// 辅助函数：左旋操作
AVLNode* rotateLeft(AVLNode* x) {
    AVLNode* y = x->right;
    x->right = y->left;
    y->left = x;

    updateHeight(x);
    updateHeight(y);

    return y;
}

// 插入操作
AVLNode* insert(AVLNode* node, int key) {
    if (node == nullptr) {
        return new AVLNode(key);
    }

    if (key < node->key) {
        node->left = insert(node->left, key);
    } else if (key > node->key) {
        node->right = insert(node->right, key);
    } else {
        return node; // 不允许插入重复键值
    }

    updateHeight(node);

    int balance = getBalanceFactor(node);

    // 左左情况
    if (balance > 1 && key < node->left->key) {
        return rotateRight(node);
    }

    // 右右情况
    if (balance < -1 && key > node->right->key) {
        return rotateLeft(node);
    }

    // 左右情况
```

```
    if (balance > 1 && key > node->left->key) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // 右左情况
    if (balance < -1 && key < node->right->key) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

// 辅助函数：找到最小值节点
AVLNode* findMin(AVLNode* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

// 删除操作
AVLNode* erase(AVLNode* node, int key) {
    if (node == nullptr) {
        return node;
    }

    if (key < node->key) {
        node->left = erase(node->left, key);
    } else if (key > node->key) {
        node->right = erase(node->right, key);
    } else {
        // 节点有左子树和右子树
        if (node->left != nullptr && node->right != nullptr) {
            AVLNode* temp = findMin(node->right);
            node->key = temp->key;
            node->right = erase(node->right, temp->key);
        } else {
            // 节点只有一个子树或没有子树
            AVLNode* temp = node->left ? node->left :
node->right;
```

```
            delete node;
            return temp;
        }
    }

    updateHeight(node);

    int balance = getBalanceFactor(node);

    // 左左情况
    if (balance > 1 && getBalanceFactor(node->left) >= 0) {
        return rotateRight(node);
    }

    // 左右情况
    if (balance > 1 && getBalanceFactor(node->left) < 0) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // 右右情况
    if (balance < -1 && getBalanceFactor(node->right) <= 0) {
        return rotateLeft(node);
    }

    // 右左情况
    if (balance < -1 && getBalanceFactor(node->right) > 0) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

// 主函数：测试用例
int main() {
    AVLNode* T = nullptr; // 初始化空树

    // 插入节点
    T = insert(T, 30);
    std::cout << "Height after inserting 30: " << getHeight(T)
<< std::endl; // 高度为 1
```

```
    T = insert(T, 40);
    T = insert(T, 50);
    std::cout << "Height after inserting 40 and 50: " <<
getHeight(T) << std::endl; // 高度为 2

    T = insert(T, 20);
    T = insert(T, 10);
    std::cout << "Height after inserting 20 and 10: " <<
getHeight(T) << std::endl; // 高度为 3

    T = insert(T, 25);
    std::cout << "Height after inserting 25: " << getHeight(T)
<< std::endl; // 高度为 3

    T = insert(T, 60);
    T = insert(T, 70);
    std::cout << "Height after inserting 60 and 70: " <<
getHeight(T) << std::endl; // 高度为 4

    // 删除节点
    T = erase(T, 10);
    T = erase(T, 25);
    std::cout << "Height after erasing 10 and 25: " <<
getHeight(T) << std::endl; // 高度为 3

    return 0;
}
```

1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw8> cd hw8.1
PS C:\Users\Macro\Desktop\hw8\hw8.1> ./main.exe
Height after inserting 30: 1
Height after inserting 40 and 50: 2
Height after inserting 20 and 10: 3
Height after inserting 25: 3
Height after inserting 60 and 70: 4
Height after erasing 10 and 25: 3
PS C:\Users\Macro\Desktop\hw8\hw8.1>
```

二、 题目二

2.1 题目表述

编写函数，返回 AVL 树中距离根节点最近的叶节点的值。

2.2 代码 2 的测试用例:

f(T) -> 0

insert(T, 30),	f(T) -> 30
insert(T, 40), insert(T, 50),	f(T) -> 30
insert(T, 20), insert(T, 10),	f(T) -> 50
insert(T, 25),	f(T) -> 10
insert(T, 60), insert(T, 70),	f(T) -> 10
erase(T, 10), erase(T, 25),	f(T) -> 20

2.3 思路:

通过递归遍历树，记录每个叶节点的深度，并返回深度最小的叶节点的值。具体步骤：从根节点开始，递归遍历左右子树，对于每个叶节点（没有左右子节点的节点），比较其深度。如果当前叶节点的深度小于已记录的最小深度，则更新最小深度和最近叶节点的值。最终返回最近叶节点的值。如果树为空，则直接返回 0。

2.4 代码:

```
#include <iostream>
#include <climits> // 用于 INT_MAX

// AVL 树的节点结构
struct AVLNode {
    int key; // 节点的键值
    int height; // 节点的高度
    AVLNode* left; // 左子节点指针
    AVLNode* right; // 右子节点指针

    AVLNode(int k) : key(k), height(1), left(nullptr),
right(nullptr) {} // 构造函数
};

// 辅助函数：计算节点高度
int getHeight(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return node->height;
}

// 辅助函数：更新节点高度
void updateHeight(AVLNode* node) {
    if (node != nullptr) {
        node->height = std::max(getHeight(node->left),
getHeight(node->right)) + 1;
    }
}

// 辅助函数：计算平衡因子
int getBalanceFactor(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return getHeight(node->left) - getHeight(node->right);
}

// 辅助函数：右旋操作
AVLNode* rotateRight(AVLNode* y) {
```

```
    if (y == nullptr || y->left == nullptr) {
        std::cerr << "Error in rotateRight: Invalid node or
left child" << std::endl;
        return y;
    }
    AVLNode* x = y->left;
    y->left = x->right;
    x->right = y;

    updateHeight(y);
    updateHeight(x);

    return x;
}

// 辅助函数：左旋操作
AVLNode* rotateLeft(AVLNode* x) {
    if (x == nullptr || x->right == nullptr) {
        std::cerr << "Error in rotateLeft: Invalid node or
right child" << std::endl;
        return x;
    }
    AVLNode* y = x->right;
    x->right = y->left;
    y->left = x;

    updateHeight(x);
    updateHeight(y);

    return y;
}

// 插入操作
AVLNode* insert(AVLNode* node, int key) {
    if (node == nullptr) {
        return new AVLNode(key);
    }

    if (key < node->key) {
        node->left = insert(node->left, key);
    } else if (key > node->key) {
        node->right = insert(node->right, key);
```

```
    } else {
        return node; // 不允许插入重复键值
    }

    updateHeight(node);

    int balance = getBalanceFactor(node);

    // 左左情况
    if (balance > 1 && key < node->left->key) {
        return rotateRight(node);
    }

    // 右右情况
    if (balance < -1 && key > node->right->key) {
        return rotateLeft(node);
    }

    // 左右情况
    if (balance > 1 && key > node->left->key) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // 右左情况
    if (balance < -1 && key < node->right->key) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

// 辅助函数：找到最小值节点
AVLNode* findMin(AVLNode* node) {
    if (node == nullptr) {
        std::cerr << "Error in findMin: Invalid node" <<
std::endl;
        return nullptr;
    }
    while (node->left != nullptr) {
        node = node->left;
```

```
    }
    return node;
}

// 删除操作
AVLNode* erase(AVLNode* node, int key) {
    if (node == nullptr) {
        return node;
    }

    if (key < node->key) {
        node->left = erase(node->left, key);
    } else if (key > node->key) {
        node->right = erase(node->right, key);
    } else {
        // 节点有左子树和右子树
        if (node->left != nullptr && node->right != nullptr) {
            AVLNode* temp = findMin(node->right);
            node->key = temp->key;
            node->right = erase(node->right, temp->key);
        } else {
            // 节点只有一个子树或没有子树
            AVLNode* temp = node->left ? node->left :
node->right;
            delete node;
            return temp;
        }
    }

    updateHeight(node);

    int balance = getBalanceFactor(node);

    // 左左情况
    if (balance > 1 && getBalanceFactor(node->left) >= 0) {
        return rotateRight(node);
    }

    // 左右情况
    if (balance > 1 && getBalanceFactor(node->left) < 0) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
```

```
    }

    // 右右情况
    if (balance < -1 && getBalanceFactor(node->right) <= 0) {
        return rotateLeft(node);
    }

    // 右左情况
    if (balance < -1 && getBalanceFactor(node->right) > 0) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

// 辅助函数：找到最近的叶节点
void findNearestLeaf(AVLNode* node, int depth, int& minDepth,
int& nearestLeaf) {
    if (node == nullptr) {
        return;
    }

    if (node->left == nullptr && node->right == nullptr) { //
叶节点
        if (depth < minDepth) {
            minDepth = depth;
            nearestLeaf = node->key;
        }
    }

    findNearestLeaf(node->left, depth + 1, minDepth,
nearestLeaf);
    findNearestLeaf(node->right, depth + 1, minDepth,
nearestLeaf);
}

// 主函数：找到最近的叶节点
int f(AVLNode* T) {
    if (T == nullptr) {
        return 0; // 空树
    }
```

```
    int minDepth = INT_MAX;
    int nearestLeaf = 0;

    findNearestLeaf(T, 0, minDepth, nearestLeaf);

    return nearestLeaf;
}

// 主函数：测试用例
int main() {
    AVLNode* T = nullptr; // 初始化空树

    // 插入节点
    T = insert(T, 30);
    std::cout << "Nearest leaf after inserting 30: " << f(T) <<
std::endl; // 30

    T = insert(T, 40);
    T = insert(T, 50);
    std::cout << "Nearest leaf after inserting 40 and 50: " <<
f(T) << std::endl; // 30

    T = insert(T, 20);
    T = insert(T, 10);
    std::cout << "Nearest leaf after inserting 20 and 10: " <<
f(T) << std::endl; // 50

    T = insert(T, 25);
    std::cout << "Nearest leaf after inserting 25: " << f(T) <<
std::endl; // 10

    T = insert(T, 60);
    T = insert(T, 70);
    std::cout << "Nearest leaf after inserting 60 and 70: " <<
f(T) << std::endl; // 10

    // 删除节点
    T = erase(T, 10);
    T = erase(T, 25);
    std::cout << "Nearest leaf after erasing 10 and 25: " <<
f(T) << std::endl; // 20
```

```
    return 0;
}
```

2.5 测试用例截图