

1.题目名称

平衡二叉树操作的演示

2.代码行数

230行

3.算法思想

1.创建AVL树

如果头节点为空，则直接对头节点开地址并赋值，如果非空则判断插入值与根节点值大小，随后进入对应子树。

重复进行上述操作，深入AVL树，直到当前头节点为非空节点停止，对其开地址并赋值。

最后不断回到上一层节点，刷新树的高度并判断AVL树是否平衡，根据平衡情况进行调整。

2.查找

利用递归的思想，将查找值与头节点值进行判断，不断深入子树。

3.删除

采用了递归的方式来搜索要删除的节点，并在删除后重新平衡树。

4.主要/核心函数分析

Insert

```
1  Avl_Tree::Node *Avl_Tree::Insert(Node *head,int num) {
2      if(head == nullptr){
3          head=new Node;
4          head->data=num;
5      }else if(num<head->data){
6          head->left=Insert(head->left,num); //进入左子树
7          if(GetHigh(head->left)- GetHigh(head->right)==2){ //判断是否平衡
8              if(num<head->left->data){
9                  head=BalanceLL(head);
10             }else{
11                 head=BalanceLR(head);
12             }
13         }
14     }else if(num>head->data){
15         head->right=Insert(head->right,num); //进入右子树
16         if(GetHigh(head->right)- GetHigh(head->left)==2){ //判断是否平衡
17             if(num<head->right->data){
18                 head=BalanceRL(head);
19             }else{
20                 head=BalanceRR(head);
21             }
22         }
23     }
24 }
```

```

25     head->leftHigh= GetHigh(head->left);    //刷新树的高度
26     head->rightHigh= GetHigh(head->right);
27     return head;
28 }

```

1. 函数接受一个 `Node* head` 和一个 `int num` 作为参数。它会递归地将值为 `num` 的新节点插入到以 `head` 为根的AVL树中。
2. 如果 `head` 为空，则创建一个值为 `num` 的新节点，并将其赋给 `head`。
3. 如果 `num` 小于当前节点的值 (`head->data`)，则在左子树上递归调用 `Insert` 函数。插入后，它会检查左子树的高度和右子树的高度之差是否为2。如果是，则根据具体情况进行LL型或LR型旋转。
4. 如果 `num` 大于当前节点的值 (`head->data`)，则在右子树上递归调用 `Insert` 函数。插入后，它会检查右子树的高度和左子树的高度之差是否为2。如果是，则根据具体情况进行RR型或RL型旋转。
5. 最后返回根节点 `head`。

这个函数实现了在AVL树中插入新节点并保持树的平衡。

Balance

```

1  Avl_Tree::Node *Avl_Tree::BalanceLL(Node *head): LL型旋转，保持平衡。
2  Avl_Tree::Node *Avl_Tree::BalanceLR(Node *head): LR型旋转，保持平衡。
3  Avl_Tree::Node *Avl_Tree::BalanceRR(Node *head): RR型旋转，保持平衡。
4  Avl_Tree::Node *Avl_Tree::BalanceRL(Node *head): RL型旋转，保持平衡。

```

Find

```

1  bool Avl_Tree::Find(int num, Node *head) {
2      if(head== nullptr) return false;
3      if(head->data==num) return true;
4      if(head->data>num) return Find(num,head->left);
5      if(head->data<num) return Find(num,head->right);
6  }

```

这个函数是一个递归函数，用于在AVL树中搜索给定的数字。它接受两个参数：要搜索的数字和树的头节点。

函数首先检查头节点是否为空，如果是，则返回false，表示树中未找到该数字。如果头节点的数据等于给定的数字，则返回true，表示在树中找到了该数字。

如果头节点的数据大于给定的数字，则在头节点的左子树上递归调用Find函数。如果头节点的数据小于给定的数字，则在头节点的右子树上递归调用Find函数。

Delete

```

1  Avl_Tree::Node* Avl_Tree::Delete(int num, Node *head) {
2      if(head== nullptr)
3          return nullptr;
4      if(num<head->data){    //寻找目标节点
5          head->left= Delete(num,head->left);
6          if(GetHigh(head->left)- GetHigh(head->right)==2){
7              if(num<head->left->data){
8                  head=BalanceLL(head);
9              }else{
10                 head=BalanceLR(head);

```

```

11     }
12 }
13 }else if(num>head->data){
14     head->right= Delete(num,head->right);
15     if(GetHigh(head->right)- GetHigh(head->left)==2){
16         if(num<head->right->data){
17             head=BalanceRL(head);
18         }else{
19             head=BalanceRR(head);
20         }
21     }
22 }else{ //找到该节点
23     if(head->left!= nullptr && head->right!= nullptr){ //非叶子节点
24         if(GetHigh(head->left) > GetHigh(head->right)){
25             Node *p=head->left;
26             while(p->right!= nullptr){ //寻找左子树最大值
27                 p=p->right;
28             }
29             head->data=p->data;
30             head->left= Delete(p->data,head->left);
31         }else{
32             Node *p=head->right;
33             while(p->left!= nullptr){ //寻找右子树最大值
34                 p=p->left;
35             }
36             head->data=p->data; //置根节点
37             head->right= Delete(p->data,head->right); //删除尾部节点
38         }
39     }else if(head->left== nullptr && head->right== nullptr){ //叶子节点直接置空
40         head= nullptr;
41     }else if(head->left!= nullptr && head->right== nullptr){ //把非空子树移上去
42         Node *p=head;
43         head=head->left;
44         delete p;
45     }else{
46         Node *p=head;
47         head=head->right;
48         delete p;
49     }
50 }
51 return head;
52 }

```

这个函数是用于在AVL树中删除指定值的节点。它采用了递归的方式来搜索要删除的节点，并在删除后重新平衡树。

函数首先检查头节点是否为空，如果是，则返回空指针。然后它比较要删除的值和当前节点的值，根据比较结果决定是向左子树还是右子树递归调用Delete函数。在递归调用后，函数会检查树是否失去平衡，并进行相应的旋转操作来恢复平衡。

如果找到了要删除的节点，函数会根据节点的情况进行不同的处理：

- 如果节点是非叶子节点，则找到左子树中的最大值或右子树中的最小值来替换当前节点，并递归删除该最大值或最小值节点。
- 如果节点是叶子节点，则直接将其置为空。

- 如果节点只有一个子树，则将子树移动到当前节点的位置，并删除原节点。

最后，函数返回更新后的头节点。

5.测试数据(规模,测试次数)

规模:随机选取的20个数

测试次数:5

测试用例:见测试文件

6.运行结果

```
1 F:\data_structure\Choice\question23\cmake-build-debug\question_3.exe
2 1. 查找
3 2. 插入
4 3. 删除
5 4. 退出
6 input:1
7 输入你要查找的数字
8 1
9 Find
10 input:2
11 请输入你要插入的数字
12 9999
13 input:1
14 输入你要查找的数字
15 9999
16 Find
17 input:3
18 请输入你要删除的数字
19 9999
20 input:1
21 输入你要查找的数字
22 9999
23 Not Find
24 input:4
25
26 进程已结束,退出代码0
27
```

7.时间复杂度分析

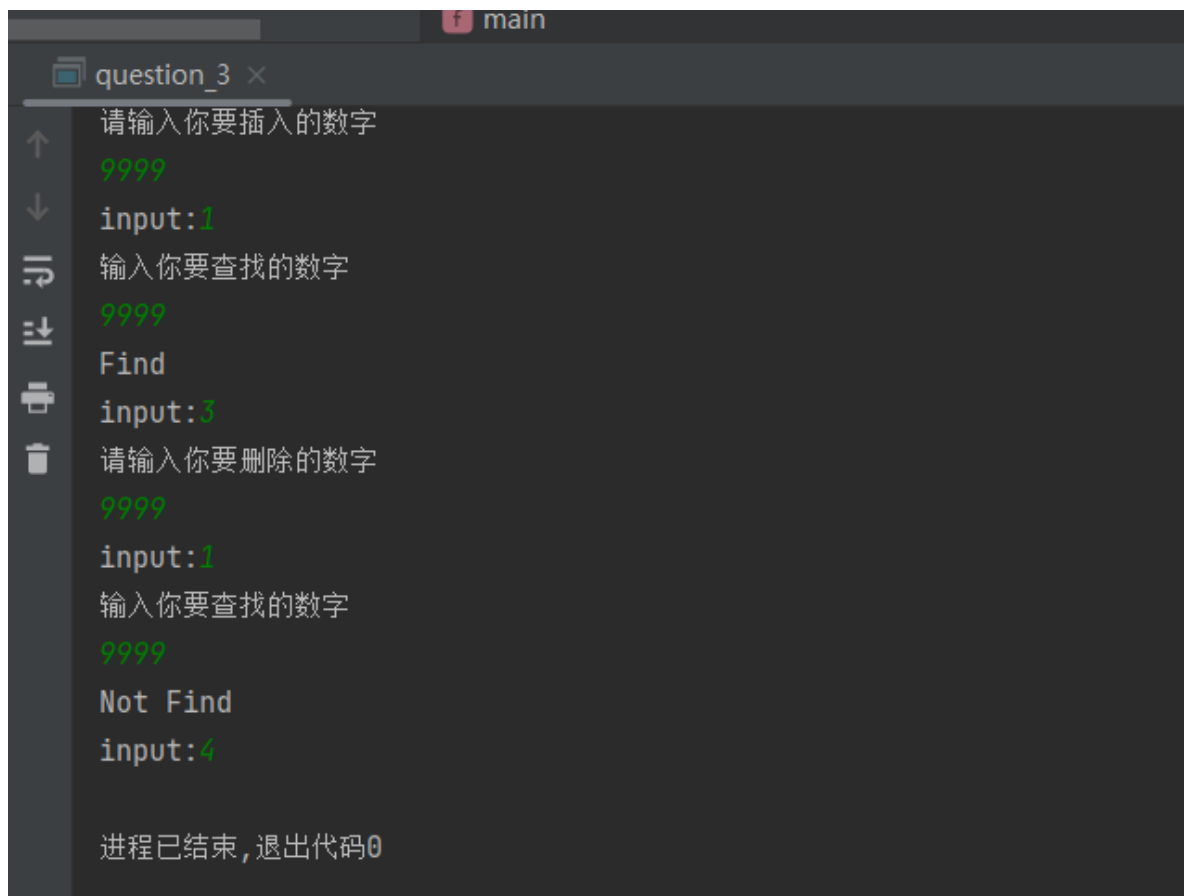
Insert 函数的时间复杂度取决于树的高度，最坏情况下为 $O(\log n)$ ，其中 n 是树中节点的数量。在最坏情况下，需要沿着树的高度向下进行插入，并在插入后重新平衡树。

Delete 函数的时间复杂度也取决于树的高度，在最坏情况下为 $O(\log n)$ ，因为它需要搜索要删除的节点，并在删除后重新平衡树。

1. 查找操作 (**Find** 函数) 的时间复杂度也与树的高度相关，在最坏情况下为 $O(\log n)$ 。
2. 平衡操作 (LL、LR、RL、RR) 的时间复杂度都是 $O(1)$ ，因为它们只是重新连接节点，不需要遍历整个树。

综上所述，在最坏情况下，函数时间复杂度是 $O(\log n)$ 。

8.结果截图图片



The screenshot shows a terminal window with a dark background. At the top, there are two tabs: 'question_3' (active) and 'main'. The terminal displays a series of prompts and user inputs for an AVL tree program. The prompts are in Chinese, and the inputs are in green. The program performs insert, find, and delete operations, and finally prints 'Not Find' and 'input:4' before ending with '进程已结束,退出代码0'.

```
question_3 x
↑ 请输入你要插入的数字
9999
↓ input:1
↺ 输入你要查找的数字
9999
↻ Find
input:3
input:3
input:3
↓ 请输入你要删除的数字
9999
input:1
输入你要查找的数字
9999
Not Find
input:4

进程已结束,退出代码0
```

9.心得体会

通过实现AVL树，对其的基本性质更加熟悉，同时对该树增删查的代码实现有了深入的了解。