1.题目名称

平衡二叉树

2.代码行数

276行

3.算法思想

1.素数筛

素数筛将[1,1e4]的素数筛出来，存入数组。

2.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。

3.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。

4.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

Select_Prime

```
void Select_Prime(){}
```

该函数是一个素数筛选函数，用于找出小于kSize的所有素数并存储在prime数组中，并对visited数组中key为素数的value定为false。

函数使用两层循环，外层循环从2开始遍历到kSize-1，内层循环用于将i的倍数标记为非素数。如果i未被标记为非素数，则将其加入prime数组中，并且将i的倍数标记为非素数。

总体来说，该函数是一个高效的素数筛选算法，使用了标记法来进行筛选，时间复杂度为O(nloglogn)。

Insert

```
Avl_Tree::Node *Avl_Tree::Insert(Node *head,int num) {
    if(head == nullptr){
        head=new Node;
        head->data=num;
    }else if(num<head->data){
        head->left=Insert(head->left,num);  //进入左子树
        if(GetHigh(head->left)- GetHigh(head->right)==2){   //判断是否平衡
```

```
            if(num<head->left->data){
                head=BalanceLL(head);
            }else{
                head=BalanceLR(head);
            }
        }
    }else if(num>head->data){
        head->right=Insert(head->right,num);    //进入右子树
        if(GetHigh(head->right)- GetHigh(head->left)==2){   //判断是否平衡
            if(num<head->right->data){
                head=BalanceRL(head);
            }else{
                head=BalanceRR(head);
            }
        }
    }

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

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

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

Find

```
bool Avl_Tree::Find(int num, Node *head) {
    if(head== nullptr) return false;
    if(head->data==num) return true;
    if(head->data>num)  return Find(num,head->left);
    if(head->data<num)  return Find(num,head->right);
}
```

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

```
Avl_Tree::Node* Avl_Tree::Delete(int num, Node *head) {
    if(head== nullptr)
        return nullptr;
    if(num<head->data){         //寻找目标节点
        head->left= Delete(num,head->left);
        if(GetHigh(head->left)- GetHigh(head->right)==2){
            if(num<head->left->data){
                head=BalanceLL(head);
            }else{
                head=BalanceLR(head);
            }
        }
    }else if(num>head->data){
        head->right= Delete(num,head->right);
        if(GetHigh(head->right)- GetHigh(head->left)==2){
            if(num<head->right->data){
                head=BalanceRL(head);
            }else{
                head=BalanceRR(head);
            }
        }
    }else{        //找到该节点
        if(head->left!= nullptr && head->right!= nullptr){        //非叶子节点
            if(GetHigh(head->left) > GetHigh(head->right)){
                Node *p=head->left;
                while(p->right!= nullptr){       //寻找左子树最大值
                    p=p->right;
                }
                head->data=p->data;
                head->left= Delete(p->data,head->left);
            }else{
                Node *p=head->right;
                while(p->left!= nullptr){      //寻找右子树最大值
                    p=p->left;
                }
                head->data=p->data;          //置根节点
                head->right= Delete(p->data,head->right); //删除尾部节点
            }
        }else if(head->left== nullptr && head->right== nullptr){    //叶子节点直接置空
            head= nullptr;
        }else if(head->left!= nullptr && head->right== nullptr){    //把非空子树移上去
            Node *p=head;
            head=head->left;
            delete p;
        }else{
            Node *p=head;
```

```
            head=head->right;
            delete p;
        }
    }
    return head;
}
```

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:小于1e4的素数以及小于1e3的偶数

测试次数:3次

6.运行结果

Tree1.txt

```
200 no
201 no
202 no
203 no
204 no
205 no
206 no
207 no
208 no
209 no
210 no
211 yes
212 no
213 no
214 no
215 no
216 no
217 no
218 no
219 no
220 no
221 no
222 no
223 yes
224 no
```

```
225 no
226 no
227 yes
228 no
229 yes
230 no
231 no
232 no
233 yes
234 no
235 no
236 no
237 no
238 no
239 yes
240 no
241 yes
242 no
243 no
244 no
245 no
246 no
247 no
248 no
249 no
250 no
251 yes
252 no
253 no
254 no
255 no
256 no
257 yes
258 no
259 no
260 no
261 no
262 no
263 yes
264 no
265 no
266 no
267 no
268 no
269 yes
270 no
271 yes
272 no
273 no
274 no
275 no
276 no
277 yes
278 no
279 no
280 no
281 yes
282 no
```

```
283 yes
284 no
285 no
286 no
287 no
288 no
289 no
290 no
291 no
292 no
293 yes
294 no
295 no
296 no
297 no
298 no
299 no
300 no

```

Tree2.txt

```
601 no
607 no
613 no
617 no
619 no
631 no
641 no
643 no
647 no
653 no
659 no
661 no
673 no
677 no
683 no
691 no

```

Tree3.txt

```
100 yes
102 yes
104 yes
106 yes
108 yes
110 yes
112 yes
114 yes
116 yes
118 yes
120 yes
122 yes
124 yes
126 yes
```

```
128 yes
130 yes
132 yes
134 yes
136 yes
138 yes
140 yes
142 yes
144 yes
146 yes
148 yes
150 yes
152 yes
154 yes
156 yes
158 yes
160 yes
162 yes
164 yes
166 yes
168 yes
170 yes
172 yes
174 yes
176 yes
178 yes
180 yes
182 yes
184 yes
186 yes
188 yes
190 yes
192 yes
194 yes
196 yes
198 yes
200 yes
```

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。

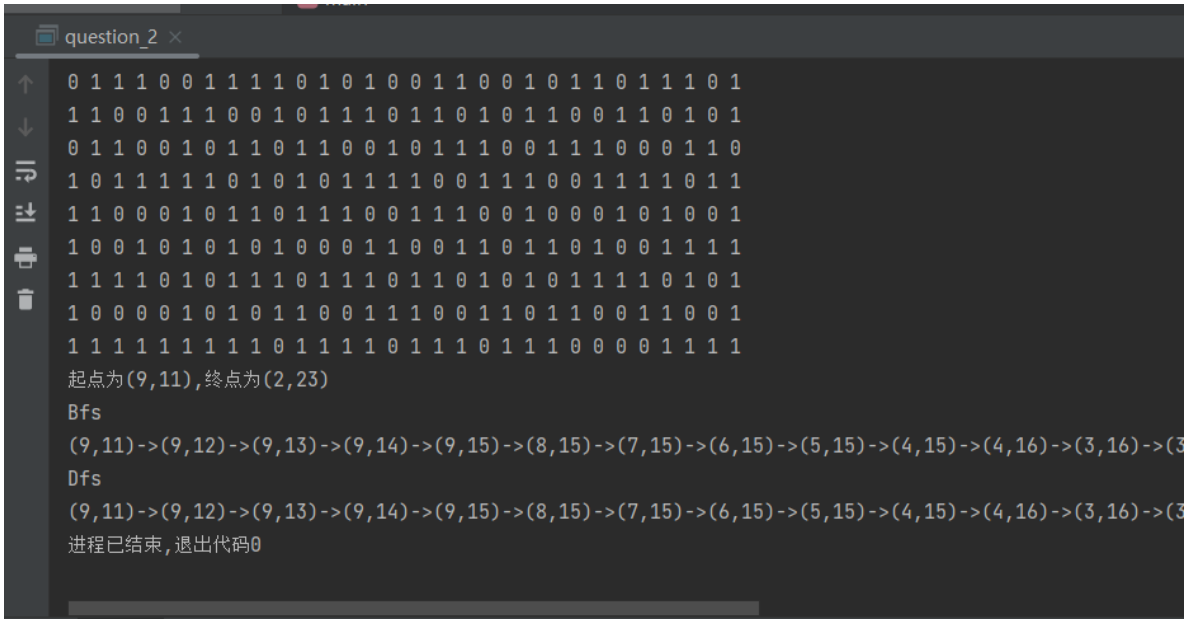
`Delete` 函数的时间复杂度也取决于树的高度，在最坏情况下为 O(log n)，因为它需要搜索要删除的节点，并在删除后重新平衡树。

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。

`Select_Prime` 函数时间复杂度为O(nloglogn)。

综上所述，在最坏情况下，函数时间复杂度是O(nloglogn)。

8.结果截屏图片



9.心得体会

通过该题对素数筛有了更多的了解。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。