

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

```
1 | 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 $O(n\log\log n)$ 。

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: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

```

1  200 no
2  201 no
3  202 no
4  203 no
5  204 no
6  205 no
7  206 no
8  207 no
9  208 no
10 209 no
11 210 no
12 211 yes
13 212 no
14 213 no
15 214 no
16 215 no
17 216 no
18 217 no
19 218 no
20 219 no
21 220 no
22 221 no
23 222 no
24 223 yes
25 224 no

```

26	225	no
27	226	no
28	227	yes
29	228	no
30	229	yes
31	230	no
32	231	no
33	232	no
34	233	yes
35	234	no
36	235	no
37	236	no
38	237	no
39	238	no
40	239	yes
41	240	no
42	241	yes
43	242	no
44	243	no
45	244	no
46	245	no
47	246	no
48	247	no
49	248	no
50	249	no
51	250	no
52	251	yes
53	252	no
54	253	no
55	254	no
56	255	no
57	256	no
58	257	yes
59	258	no
60	259	no
61	260	no
62	261	no
63	262	no
64	263	yes
65	264	no
66	265	no
67	266	no
68	267	no
69	268	no
70	269	yes
71	270	no
72	271	yes
73	272	no
74	273	no
75	274	no
76	275	no
77	276	no
78	277	yes
79	278	no
80	279	no
81	280	no
82	281	yes
83	282	no

84	283	yes
85	284	no
86	285	no
87	286	no
88	287	no
89	288	no
90	289	no
91	290	no
92	291	no
93	292	no
94	293	yes
95	294	no
96	295	no
97	296	no
98	297	no
99	298	no
100	299	no
101	300	no
102		

Tree2.txt

1	601	no
2	607	no
3	613	no
4	617	no
5	619	no
6	631	no
7	641	no
8	643	no
9	647	no
10	653	no
11	659	no
12	661	no
13	673	no
14	677	no
15	683	no
16	691	no
17		

Tree3.txt

1	100	yes
2	102	yes
3	104	yes
4	106	yes
5	108	yes
6	110	yes
7	112	yes
8	114	yes
9	116	yes
10	118	yes
11	120	yes
12	122	yes
13	124	yes
14	126	yes

15	128	yes
16	130	yes
17	132	yes
18	134	yes
19	136	yes
20	138	yes
21	140	yes
22	142	yes
23	144	yes
24	146	yes
25	148	yes
26	150	yes
27	152	yes
28	154	yes
29	156	yes
30	158	yes
31	160	yes
32	162	yes
33	164	yes
34	166	yes
35	168	yes
36	170	yes
37	172	yes
38	174	yes
39	176	yes
40	178	yes
41	180	yes
42	182	yes
43	184	yes
44	186	yes
45	188	yes
46	190	yes
47	192	yes
48	194	yes
49	196	yes
50	198	yes
51	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 $O(n \log \log n)$ 。

综上所述，在最坏情况下，函数时间复杂度是 $O(n \log \log n)$ 。

8.结果截图图片

```
question_2 x
↑ 0 1 1 1 0 0 1 1 1 1 0 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 1 1 0 1
↓ 1 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 0 1
| 0 1 1 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0
| 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1
| 1 1 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 0 1 0 0 0 1 0 1 0 0 1
| 1 0 0 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 1 1
| 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 0 1 0 1
| 1 0 0 0 0 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1
| 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 1 1 1 1
起点为(9,11),终点为(2,23)
Bfs
(9,11)->(9,12)->(9,13)->(9,14)->(9,15)->(8,15)->(7,15)->(6,15)->(5,15)->(4,15)->(4,16)->(3,16)->(3,17)
Dfs
(9,11)->(9,12)->(9,13)->(9,14)->(9,15)->(8,15)->(7,15)->(6,15)->(5,15)->(4,15)->(4,16)->(3,16)->(3,17)
进程已结束,退出代码0
```

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。