哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：数据结构与算法

课程类型：必修

实验项目：树型查找结构与排序方法

实验题目：AVL 树存储结构建立以及插入、删除、查找算法的实现

实验日期：12 月 17

设计成绩	报告成绩	指导老师
		张岩

一、实验目的

AVL 树是一种基本的查找（搜索）结构。本实验要求编写程序实现 AVL 存储结构的建立（插入）、删除、查找算法，并反映插入和删除操作算法的各种旋转变化。

二、实验要求及实验环境

1．设计 AVL 的左右链存储结构；

2．实现 AVL 左右链存储结构上的插入（建立）、删除、查找和排序算法。

3．测试数据以文件形式保存，能反映插入和删除操作的四种旋转，并输出相应的结果。

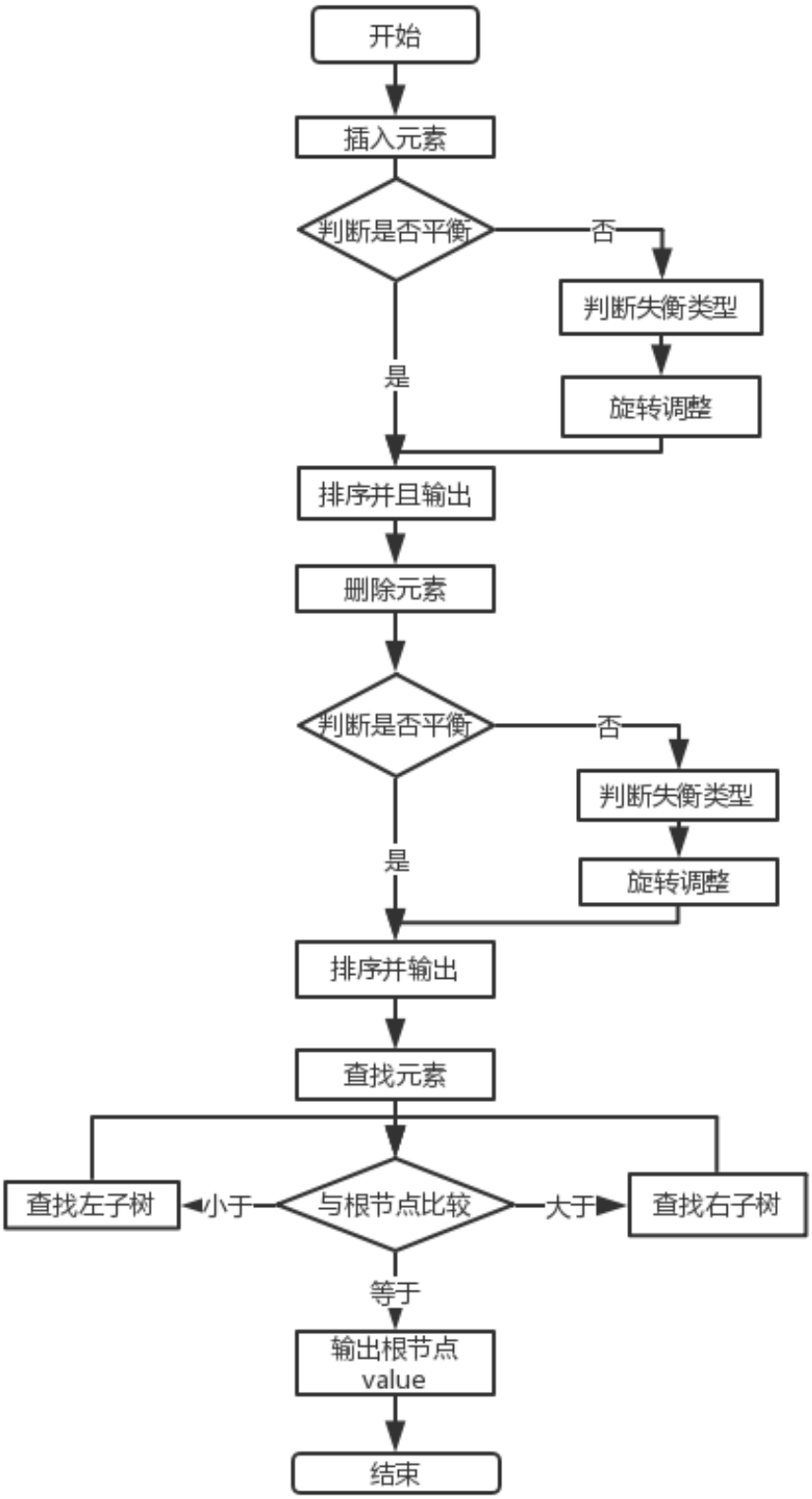
4.windows 10,codeblocks 16.01

三、设计思想（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系）

1．物理设计

```
typedef struct AVLTreeAVLTreeNode{
    Type value;                    // 关键字(键值)
    int height_count;//树的高度，用于求结点的平衡因子
    struct AVLTreeAVLTreeNode *left;    // 左孩子
    struct AVLTreeAVLTreeNode *right;    // 右孩子
}AVLTreeNode, *AVLTree;
```

2．逻辑设计


开始
插入元素
判断是否平衡
否
判断失衡类型
旋转调整
是
排序并且输出
删除元素
判断是否平衡
否
判断失衡类型
旋转调整
是
排序并输出
查找元素
查找左子树
小于
与根节点比较
大于
查找右子树
等于
输出根节点
value
结束

四、测试结果

```
LL之前
 3
2
1



LL之后
 2
1 3



RR之前
 3
  4
  5



RR之后
 4
3 5
```

```


RR之前
   2
 1   4
    3 5
      6


RR之后
   4
 2   5
1 3   6


RR之前
 5
  6
  7



RR之后
 6
5 7
```

```
RL之前
 7
 16
 15


RL之后
 15
7 16


RL之前
   6
 5  15
    716
    14

RL之后
   7
 6   15
5   14 16
```

```
RR之前
       4
   2       7
 1   3   6  15
         5  14 16
            13

RR之后
       7
   4       15
 2   6   14   16
1 3 5   13


LL之前
 14
13
12


LL之后
 13
12 14
```

```
LL之前
   15
 13   16
12 14
11


LL之后
   13
 12   15
11   14 16


LL之前
 12
11
10


LL之后
 11
10 12
```

```
LR之前
10
8
9


LR之后
 9
8 10


排序:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
               7
        4                13
    2       6        11       15
 1    3   5        9   12   14   16
                  8 10

删除元素11后
1 2 3 4 5 6 7 8 9 10 12 13 14 15 16
               7
        4                13
    2       6        10       15
 1    3   5        9   12   14   16
                  8

10在0x5e7b90
0x5e7b90处:10
```

五、经验体会与不足

对 AVL 树的旋转维护平衡的操作更加清楚，同时更加深入的明白了平衡二叉树的在查找的时候的高效率以及为排序操作带来的方便。不足：在打印树的时候，结构还是不是很清楚完美。

六、附录：源代码（带注释）

```
#include<iostream>
#include<cstdio>
#include <malloc.h>
#include<math.h>
#define MAX(a, b)    ( (a) > (b) ? (a) : (b) )

using namespace std;


typedef int Type;

typedef struct AVLTreeAVLTreeNode{
    Type value;                    // 关键字(键值)
    int height_count;
    struct AVLTreeAVLTreeNode *left;    // 左孩子
    struct AVLTreeAVLTreeNode *right;    // 右孩子
}AVLTreeNode, *AVLTree;

int Pheightc;
char Phbuf[6][200];
int Phx;

void pprint_tree(AVLTree tree, int level){
    if (tree == NULL){
        Phx += (pow(2, Pheightc - level) - 1);
        return;
    }
    char(*a)[200] = Phbuf;
    pprint_tree(tree->left, level + 1);
    a[level][Phx++] = tree->value;
    pprint_tree(tree->right, level + 1);

}

int height_count(AVLTreeNode *p)
{
```

```
    if(p==NULL)return 0;
    else{return ((AVLTreeNode *)(p))->height_count; }
}
void printTree(AVLTree tree){
    Pheightc=0;
    Phx=0;
    if (tree == NULL) return;
    char(*a)[200] = Phbuf;
    for (int i = 0; i<6; i++){
        for (int j = 0; j<200; j++){
            a[i][j] = '#';
        }
    }
    //先获取树高度
    Pheightc = height_count(tree);
    if (Pheightc > 6){
        cout << "error" << endl;
        return;
    }
    pprint_tree(tree, 0);
    for (int i = 0; i < 6; i++){
        for (int j = 0; j < 200; j++){
            if (a[i][j] =='#') cout << " ";
            else cout << (int)a[i][j];
        }
        cout << endl;
    }
}

/*
  获取 AVL 树的高度
 */
int avltree_height_count(AVLTree tree)
{
    return height_count(tree);
}

/*
  LL：左左对应的情况(左单旋转)。

  返回值：旋转后的根节点
 */
AVLTreeNode* LLRotation(AVLTree temp2)
{
```

```
81.     AVLTree temp1;
82.
83.     temp1 = temp2->left;
84.     temp2->left = temp1->right;
85.     temp1->right = temp2;
86.
87.     temp2->height_count = MAX( height_count(temp2->left), height_count(temp2
    ->right)) + 1;
88.     temp1->height_count = MAX( height_count(temp1->left), temp2->height_coun
    t) + 1;
89.
90.     return temp1;
91. }
92.
93. /*
94.   RR：右右对应的情况(右单旋转)。
95.
96.   返回值：旋转后的根节点
97.  */
98.  AVLTreeNode* RRRotation(AVLTree temp1)
99. {
100.     AVLTree temp2;
101.
102.     temp2 = temp1->right;
103.     temp1->right = temp2->left;
104.     temp2->left = temp1;
105.
106.     temp1->height_count = MAX( height_count(temp1->left), height_count(temp
    1->right)) + 1;
107.     temp2->height_count = MAX( height_count(temp2->right), temp1->height_co
    unt) + 1;
108.
109.     return temp2;
110. }
111.
112.
113. /*
114.   LR：左右对应的情况(左双旋转)。
115.
116.   返回值：旋转后的根节点
117.  */
118.  AVLTreeNode* LRRotation(AVLTree temp3)
119. {
120.     temp3->left = RRRotation(temp3->left);
```

```
121.
122.     return LLRotation(temp3);
123. }
124.
125. /*
126.   RL：右左对应的情况(右双旋转)。
127.
128.   返回值：旋转后的根节点
129.  */
130.  AVLTreeNode* RLRotation(AVLTree temp1)
131. {
132.     temp1->right = LLRotation(temp1->right);
133.
134.     return RRRotation(temp1);
135. }
136.
137. /*
138.   将结点插入到 AVL 树中，并返回根节点
139.
140.   参数说明：
141.       tree AVL 树的根结点
142.       value 插入的结点的键值
143.   返回值：
144.       根节点
145.  */
146. AVLTreeNode* avltree_insert(AVLTree tree, Type value)
147. {
148.     if (tree == NULL)
149.     {
150.         // 新建节点
151.         //tree = avltree_create_AVLTreeNode(value, NULL, NULL);
152.         AVLTreeNode* p;
153.         p = (AVLTreeNode *)malloc(sizeof(AVLTreeNode));
154.         p->value = value;
155.         p->height_count = 0;
156.         p->left = NULL;
157.         p->right = NULL;
158.         tree=p;
159.         if (tree==NULL)
160.         {
161.             printf("ERROR!\n");
162.             return NULL;
163.         }
164.     }
```

```
165.     else if (value < tree->value) // 应该将 value 插入到"tree 的左子树"的情况
166.     {
167.         tree->left = avltree_insert(tree->left, value);
168.         // 插入节点后，若 AVL 树失去平衡，则进行相应的调节。
169.         if (height_count(tree->left) - height_count(tree->right) == 2)
170.         {
171.             if (value < tree->left->value)
172.             {
173.                 cout<<"LL 之前"<<endl;
174.                 printTree(tree);
175.                 tree = LLRotation(tree);
176.                 cout<<"LL 之后"<<endl;
177.                 printTree(tree);
178.             }
179.             else
180.             {
181.                 cout<<"LR 之前"<<endl;
182.                 printTree(tree);
183.                 tree = LRRotation(tree);
184.                 cout<<"LR 之后"<<endl;
185.                 printTree(tree);
186.             }
187.         }
188.     }
189.     else if (value > tree->value) // 应该将 value 插入到"tree 的右子树"的情况
190.     {
191.         tree->right = avltree_insert(tree->right, value);
192.         // 插入节点后，若 AVL 树失去平衡，则进行相应的调节。
193.         if (height_count(tree->right) - height_count(tree->left) == 2)
194.         {
195.             if (value > tree->right->value)
196.             {
197.                 cout<<"RR 之前"<<endl;
198.                 printTree(tree);
199.                 tree = RRRotation(tree);
200.                 cout<<"RR 之后"<<endl;
201.                 printTree(tree);
202.             }
203.             else
204.             {
205.                 cout<<"RL 之前"<<endl;
206.                 printTree(tree);
207.                 tree = RLRotation(tree);
208.                 cout<<"RL 之后"<<endl;
```

```
                printTree(tree);
            }
        }
    }
    else //value == tree->value)
    {
        printf("error!\n");
    }
    tree->height_count = MAX( height_count(tree->left), height_count(tree->right)) + 1;
    return tree;
}
/*
    查找
*/
AVLTreeNode *avltree_search(AVLTree tree,Type value)
{
        if(tree->value==value)return tree;
        if(tree->value<value) return avltree_search(tree->right,value);
        if(tree->value>value) return avltree_search(tree->left,value);
}

/*
    找最大,最小节点
*/

AVLTreeNode *avltree_maximum(AVLTreeNode * T)
{
    AVLTreeNode *temp=T;
    while(temp->right!=NULL)
    {
        temp=temp->right;
    }
    return temp;
}
AVLTreeNode *avltree_minimum(AVLTreeNode * T)
{
    AVLTreeNode *temp=T;
    while(temp->left!=NULL)
    {
        temp=temp->left;
    }
    return temp;
}
```

```
/*
  删除结点(z)，返回根节点

  参数说明：
      ptree AVL 树的根结点
      z 待删除的结点
  返回值：
      根节点
 */
AVLTreeNode* delete_AVLTreeNode(AVLTree tree, AVLTreeNode *z)
{
    // 根为空 或者 没有要删除的节点，直接返回 NULL。
    if (tree==NULL || z==NULL)
        return NULL;

    if (z->value < tree->value)        // 待删除的节点在"tree 的左子树"中
    {
        tree->left = delete_AVLTreeNode(tree->left, z);
        // 删除节点后，若 AVL 树失去平衡，则进行相应的调节。
        if (height_count(tree->right) - height_count(tree->left) == 2)
        {
            AVLTreeNode *r =  tree->right;
            if (height_count(r->left) > height_count(r->right))
                tree = RLRotation(tree);
            else
                tree = RRRotation(tree);
        }
    }
    else if (z->value > tree->value)// 待删除的节点在"tree 的右子树"中
    {
        tree->right = delete_AVLTreeNode(tree->right, z);
        // 删除节点后，若 AVL 树失去平衡，则进行相应的调节。
        if (height_count(tree->left) - height_count(tree->right) == 2)
        {
            AVLTreeNode *l =  tree->left;
            if (height_count(l->right) > height_count(l->left))
                tree = LRRotation(tree);
            else
                tree = LLRotation(tree);
        }
    }
    else    // tree 是对应要删除的节点。
    {
        // tree 的左右孩子都非空
```

```
296.         if ((tree->left) && (tree->right))
297.         {
298.             if (height_count(tree->left) > height_count(tree->right))
299.             {
300.                 AVLTreeNode *max = avltree_maximum(tree->left);
301.                 tree->value = max->value;
302.                 tree->left = delete_AVLTreeNode(tree->left, max);
303.             }
304.             else
305.             {
306.                 AVLTreeNode *min = avltree_minimum(tree->right);
307.                 tree->value = min->value;
308.                 tree->right = delete_AVLTreeNode(tree->right, min);
309.             }
310.         }
311.         else
312.         {
313.             AVLTreeNode *tmp = tree;
314.             tree = tree->left ? tree->left : tree->right;
315.             delete(tmp);
316.         }
317.     }
318. 
319.     return tree;
320. }
321. 
322. /*
323.   删除结点(value 是节点值)，返回根节点
324. 
325.   参数说明:
326.       tree AVL 树的根结点
327.       value 待删除的结点的键值
328.   返回值:
329.       根节点
330.  */
331. 
332. AVLTreeNode* avltree_delete(AVLTree tree, Type value)
333. {
334.     AVLTreeNode *z;
335.     if ((z = avltree_search(tree, value)) != NULL)
336.         tree = delete_AVLTreeNode(tree, z);
337.     return tree;
338. }
339. 
```

```
340. void inorder(AVLTreeNode* T)
341. {
342.     if (T == NULL)
343.         return;
344.     else
345.     {
346.         inorder(T->left);
347.         printf("%d ", T->value);
348.         inorder(T->right);
349.     }
350. }
351.
352. int* search(int value,AVLTreeNode * T)
353. {
354.     if(value==T->value){cout<<value<<"在
    "<<&(T->value)<<endl;return &(T->value);}
355.     if(value>T->value)return search(value,T->right);
356.     if(value<T->value)return search(value,T->left);
357. }
358.
359. int main()
360. {
361.     freopen("test.txt","r",stdin);
362.     freopen("ans.txt","w",stdout);
363.     AVLTreeNode* T=NULL;
364.
365.     int arr[16];
366.     for(int i=0;i<=15;i++)
367.         scanf("%d",&arr[i]);
368.
369.
370.     for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++)
371.        {
372.          T = avltree_insert(T,arr[i]);
373.        }
374.     cout<<"排序:"<<endl;
375.     inorder(T);
376.     cout<<endl;
377.     printTree(T);
378.     cout<<"删除元素 11 后"<<endl;
379.     T=avltree_delete(T, 11);
380.     inorder(T);
381.     cout<<endl;
382.     printTree(T);
```

```
    cout<<endl;
    int *temp;
    temp=search(10,T);
    cout<<temp<<"处:"<<*temp<<endl;
    fclose(stdin);
    fclose(stdout);
    return 0;
}
```