廣東工業大學

课 程 设 计

课程名称　　数据结构

题目名称　平衡二叉树操作的演示(难度 1.3)

学生学院　　计算机学院

专业班级　　网络工程 1602

学　　号　　3116004982

学生姓名　　赵舒宇

指导教师　　李杨

2018 年 1 月 10 日

1. 需求分析

1.1 输入的形式

输入的元素为整形类型，输入值的范围即为 int 形的输入范围。

1.2 输出的形式

输出的形式为一棵或多棵以凹入表形式表示的平衡二叉树。

1.3 程序功能

程序利用平衡二叉树实现了动态查找表的基本操作——查找，插入，删除，也完成了附加功能要求——合并两棵平衡二叉树，将一棵平衡二叉树分裂为两棵平衡二叉树。

1.4 测试

测试数据由程序用户手动输入，此处提供一样例：

(1) T1 为一空树，依次插入 1，2，3，4，5，插入完成后的 T1 为：

```
        5
    4
        3
2
    1
```

(2) 在 T1 中继续插入 2，返回的 T1 依然为：

```
        5
    4
        3
2
    1
```

(3) 从 T1 中删除 2 后，T1 为：

```
        5
    4
3
    1
```

(4) 在 T1 中查找 4，输出：

Find 4 successfully.

(5) 在 T1 中查找 2，输出：

Find failed.

(6) 生成一棵新树 T2，并一次插入 6，7，8，合并 T1，T2 后 T1 为：

```
        8
    7
        6
5
        4
    3
        1
```

(7)将 T1 分裂为 T2，T3 两棵树，分裂的关键字为 6，分裂后的 T2，T3 为：

T2

```
    8
7
```

T3

6

5

4

3

1

2. 概要设计

2.1 数据类型

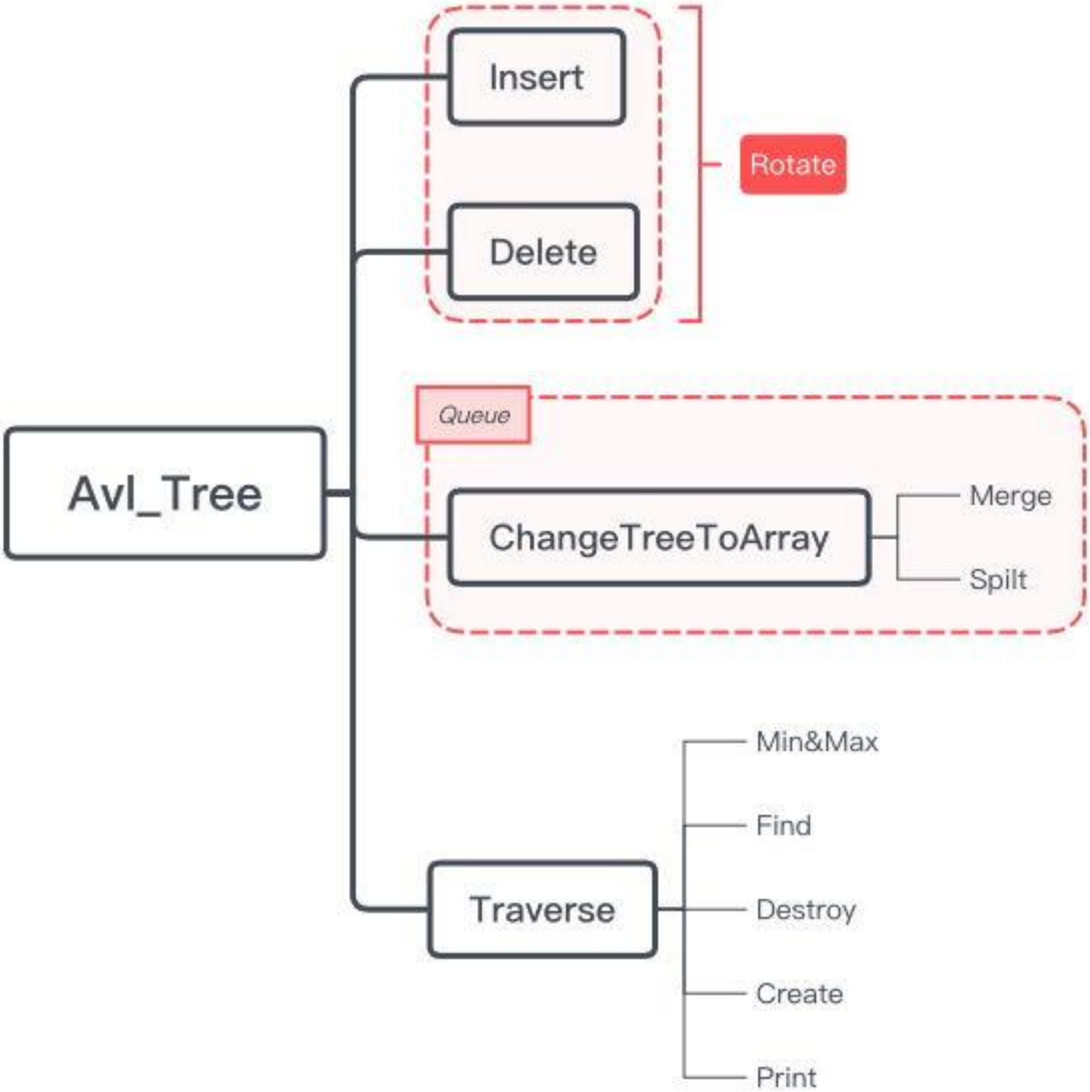
数据对象：$D=\{ a_i \mid a_i \in ElemSet,\ i=1,2,...,n,\ n \geq 0 \}$

数据关系：$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i=2,...,n \}$

2.2 程序流程

主程序包括对两棵树 T1，T2 的各种操作，共 12 种，包括创建，插入，删除，查找，销毁，合并，分裂

2.3 程序模块



3. 详细设计

3.1 结构声明

首先完成常用声明，并定义一些常用变量增加代码可读性：

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define OVERFLOW -2

typedef int Status;
typedef int ElementType;
```

然后声明程序中储存平衡二叉树的结构体及辅助用结构体：

```
/*储存平衡二叉树结构声明*/
typedef struct AvlNode{
    ElementType element;        //数据域
    struct AvlNode *left;       //左右孩子指针域
    struct AvlNode *right;
    int height;                 //树高
}AvlNode,*AvlTree;

/*存放输入数据的数组结构体*/
typedef struct ArrayNode{
    ElementType element;        //存放记录的结点元素值
    ArrayNode *next;            //指向下一个结点
}ArrayNode, *Array;

/*链队列结构体*/
typedef struct LQNode{
    AvlTree element;            //存放遍历时树的指针
    struct LQNode *next;        //指向下一个结点
}LQNode, *QueuePtr;

/*队列结点结构体*/
typedef struct{
    QueuePtr front;             //队头指针
    QueuePtr rear;              //队尾指针
}LQueue;
```

3.2 辅助函数算法

3.2.1 树高

返回树结构体中包含的 height 值，若为空树则返回 0。

```
int AvlTree_Height(AvlTree &T){
if(NULL==T)
    return 0;
else
    return T->height;
}
```

3.2.2 比较取最大树高

调用三目运算符来修正树高

```
int MAX(int T1,int T2){
    return T1>T2 ? T1 : T2;
}
```

3.2.3 创建

创建一个节点，并把他的左孩子，右孩子，树高等值置为默认值。

```
AvlTree AvlTree_Create(ElementType x,AvlNode *left,AvlNode *right){
    AvlTree p=(AvlNode*)malloc(sizeof(AvlNode));
    if(NULL==p)
        return NULL;
    p->element=x;
    p->left=left;
    p->right=right;
    p->height=0;
    return p;
}
```

3.2.4 旋转

在应对 LL 情况时，调用右旋函数：

先暂存失衡节点 T2 的左孩子 T1，再将失衡节点 T2 的左孩子置为原左孩子的右孩子，再将暂存的节点 T1 的右孩子置为修正后的 T2，并更新根节点为 T1，修正树高。

```
AvlTree SingleRotate_Left(AvlTree &T2){
    T1=T2->left;                //暂存 T2 的左孩子
    T2->left=T1->right;         //将 T2 的左孩子置为其原左孩子的右孩子(如必要)
    T1->right=T2;               //将 T1 的右孩子置为修正后的 T2
    T2->height=MAX(AvlTree_Height(T2->left),AvlTree_Height(T2->right))+1;
    T1->height=MAX(AvlTree_Height(T1->left),T2->height)+1;        //修正树高
    return T1;                  //此时 T1 更新为原根节点(T2)
}
```

在应对 LR 情况时，调用双旋转函数：

对失衡节点进行左旋处理，再对修正后的失衡节点进行右旋处理，并返回修正后的失衡节点，同时修正树高。

```
AvlTree DoubleRotate_Left(AvlTree &T3){
    T3->left=SingleRotate_Right(T3->left);
    return SingleRotate_Left(T3);
}
```

剩余两种情况 RR 和 RL 分别是上述两种情况的镜像处理，仅列出代码。

```
/*RR*/
AvlTree SingleRotate_Right(AvlTree &T2){
    T1=T2->right;
    T2->right=T1->left;
    T1->left=T2;
    T2->height=MAX(AvlTree_Height(T2->left),AvlTree_Height(T2->right))+1;
    T1->height=MAX(AvlTree_Height(T1->right),T2->height)+1;
    return T1;
}
/* RL*/
AvlTree DoubleRotate_Right(AvlTree &T3){
    T3->right=SingleRotate_Left(T3->right);
    return SingleRotate_Right(T3);
}
```

3.2.5 将树转化为数组

在该算法中，需要调用到三个基础的队列操作——初始化队列，入队，出队：

```
/*初始化链队列*/
Status LQueue_Init(LQueue &Q){
    Q.front = NULL;
    Q.rear= NULL;
    return OK;
}
/*链队列进队操作*/
Status LQueue_EnQueue(LQueue &Q, AvlTree &T){
    QueuePtr p=(LQNode*)malloc(sizeof(LQNode));
    if(NULL==p)
        return OVERFLOW;
    p->element=T;
    p->next=NULL;
    if(NULL==Q.front)
        Q.front=p;   //当先队列为空，直接插入空队列
    else
        Q.rear->next=p;   //当先队列非空，插在 rear 后
    Q.rear=p;                      //更新 rear
    return OK;
}
/*链队列出队操作*/
Status LQueue_DeQueue(LQueue &Q, AvlTree &T){
    QueuePtr p;
    if(NULL==Q.front)
        return ERROR;//当先队列为空
    p=Q.front;
    T=p->element;
```

```
        Q.front=p->next;
        if(Q.rear==p)
            Q.rear=NULL;//遍历完了整个队列
        free(p);
        return OK;
    }
```

在转化操作中，设置一个标记变量来判断插入元素是否为数组的首元素，若是首元素则将 head 指针等于 p，并把 flag 置为 false。

```
    if(FLAG==TRUE){
        head=p;
        q=p;
        FLAG=FALSE;
    }
    else{
        q->next=p;
        q=q->next;
    }
```

然后依次将树中的元素入队，把队列中的元素保存到数组中，区别在于队列中的 element 为二叉树节点，而数组中的 element 为二叉树节点的 element 值。

```
    LQueue_EnQueue(*Q,X);
    while(LQueue_DeQueue(*Q,X)){
        if(X->left!=NULL){
            p->element=X->left->element;
            q->next=p;
            q=q->next;
            LQueue_EnQueue(*Q,X->left);
        }
        if(X->right!=NULL){
            p->element=X->right->element;
            q->next=p;
            q=q->next;
            LQueue_EnQueue(*Q,X->right);
        }
    }
```

最后返回数组的头指针用于之后算法的遍历。

```
    return head;
```

3.3 核心操作算法

3.3.1 寻找

使用非递归算法，判断寻找节点与根节点的大小关系进行查找，直到找到待查找节点或没找到时返回 NULL。

```
    AvlTree AvlTree_Find(ElementType x,AvlTree &T){
        while((T!=NULL)&&(T->element!=x))
        {
            if(x<T->element)
```

```
            T=T->left;
        else
            T=T->right;
    }
    return T;
}
```

3.3.2 插入

判断待插入节点应该位于哪一分支再进行插入，插入后判断二叉树是否依然平衡，如必要就进行相应的旋转操作来恢复平衡，并在最后修正高度。

```
AvlTree AvlTree_Insert(ElementType x,AvlTree &T){
    if(NULL==T)
        T=AvlTree_Create(x,NULL,NULL);
    else if(x<T->element){
        T->left=AvlTree_Insert(x,T->left);    //插入至左子树
        if(AvlTree_Height(T->left)-AvlTree_Height(T->right)==2){
            if(x<T->left->element)
                T=SingleRotate_Left(T);
            else
                T=DoubleRotate_Left(T);
        }
    }
    else if(x>T->element){
        …上述的镜像操作…
    }
    T->height=MAX(AvlTree_Height(T->right),AvlTree_Height(T->left))+1;
    return T;
}
```

3.3.3 删除

若待删除节点是叶子节点或只有一个孩子节点等简单情况时，进行处理。

```
if(!T)
    return FALSE;
if(X==T->element){            //找到结点
    if(!T->left&&!T->right)   //待删除结点为叶子结点
        T=NULL;
    else if(!T->left)         //待删除结点只有右孩子
        T=T->right;
    else if(!T->right)        //待删除结点只有左孩子
        T=T->left;
```

若待删除节点左右孩子均存在，则判断并寻找左右孩子子树的最高的那一棵，来选择用根节点的前驱结点还是后继结点来替代他。将根节点替换后，再递归删除掉之前他的前驱结点或后继结点。

```
if (AvlTree_Height(T->left)>AvlTree_Height(T->right)){
    pre=T->left;
    while(pre->right)    //寻找前驱结点 pre
```

```
            pre=pre->right;
        T->element=pre->element;        //用 pre 替换 T
        AvlTree_Delete(pre->element,T->left);//删除 pre
    }
    else{
        post=T->right;
        while(post->left)    //寻找后继节点 post。
            post = post->left;
        T->element=post->element;      //用 post 替换 T
        AvlTree_Delete(post->element,T->right);//删除 post
    }
```

而在寻找删除节点的位置和处理删除后的恢复平衡操作，代码类似于插入操作，值得注意的是，在删除节点后调整树高时，需要处理好谨慎度。不同于插入操作，删除后调整树高需要判断树是否为空树，否则在递归过程中会影响到最终的树高，以至于影响到旋转操作。若采用平衡因子法来编写代码，则代码量比单纯采用树高更繁琐，但不用考虑修正的谨慎度也算是其优点之一。

```
    if(X < T->element){                //在左子树中递归删除。
        if (!AvlTree_Delete(X,T->left))
            return FALSE;
        else{
            //删除成功，修改树的高度。
            T->height = MAX(AvlTree_Height(T->left), AvlTree_Height(T->right)) + 1;
            if (-2==AvlTree_Height(T->left)-AvlTree_Height(T->right)){
                if(AvlTree_Height(T->right->left)>AvlTree_Height(T->right->right))
                        DoubleRotate_Right(T);
                    else
                        SingleRotate_Right(T);
            }
            return TRUE;
        }
    }
    else{                                          //在右子树中递归删除
        if(!AvlTree_Delete(X,T->right))
            return FALSE;
        else{
            //删除成功，修改树的高度。
            T->height = MAX(AvlTree_Height(T->left), AvlTree_Height(T->right)) + 1;
            //已在 T 的右子树删除结点 X，修正平衡
            if (2 == AvlTree_Height(T->left) - AvlTree_Height(T->right)){
                if (AvlTree_Height(T->left->left) > AvlTree_Height(T->left->right))
                    SingleRotate_Left(T);
                else
                    DoubleRotate_Left(T);
            }
```

```
            return TRUE;
        }
    }
```

3.3.4 销毁

递归销毁树的左右分支即可

```
    Status AvlTree_Destroy(AvlTree &T){
        if (NULL==T)
            return ERROR;
        if (T->left != NULL)
            AvlTree_Destroy(T->left);
        if (T->right != NULL)
            AvlTree_Destroy(T->right);
        free(T);
    }
```

3.3.5 合并

将 T2 转化为数组，调用插入操作依次插入进 T1 即完成了合并操作

```
    AvlTree AvlTree_Merge(AvlTree &T1,AvlTree &T2){
        a=ChangeTreeToArray(T2);
        while(a!=NULL){
            AvlTree_Insert(a->element,T1);
            a=a->next;
        }
        return T1;
    }
```

3.3.6 分裂

分裂操作也需调用 ChangeTreeToArray 函数，将待分裂的树转化为数组，将大于分裂关键字的元素插入另一树中，同时删除掉本身树中的该元素，即完成了分裂操作。

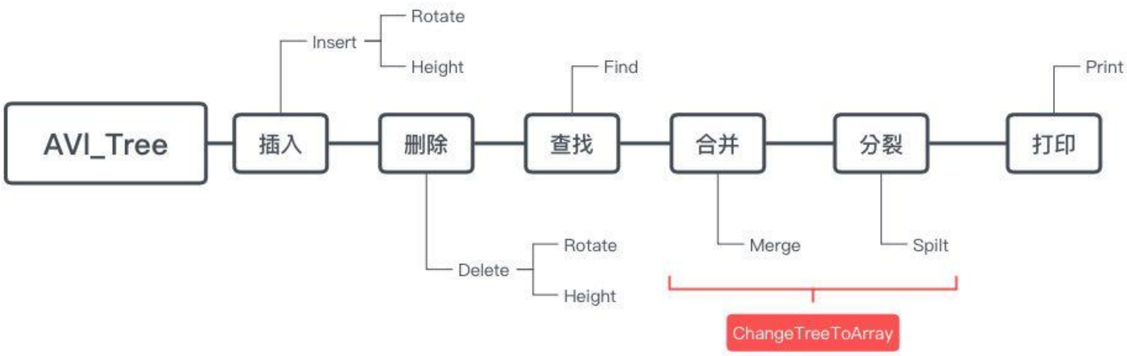
```
    Status AvlTree_Spilt(AvlTree &T1,AvlTree &T2,ElementType x){
        a=ChangeTreeToArray(T1);
        if(T1==NULL)
            return FALSE;
        else{
            while(a!=NULL){
                if(a->element<=x)
                    a=a->next;
                else{
                    AvlTree_Insert(a->element,T2);
                    AvlTree_Delete(a->element,T1);
                    a=a->next;
                }
            }
        }
        return TRUE;
    }
```

3.3.7 显示

根据层深度递归的打印相应数量的空格来实现凹入表的输出形式

```
Status AvlTree_Print(AvlTree &T, int dep){
    int i;
    if(T->right)
        AvlTree_Print(T->right,dep+1);
    for(i=0;i<dep;i++)
        printf("      ");
    printf("%d\n",T->element);
    if(T->left)
        AvlTree_Print(T->left,dep+1);
    return OK;
}
```

3.4 函数调用关系



4. 调试分析

4.1 调试中的问题

在调试中总体遇到的问题数量并不多，但是处理起来却很棘手。主要原因是在递归算法中，问题藏得很深，debug 时往往需要多次调试才能找出 bug，在调试删除算法时，出现了删除错误，一棵形如 的树删除节点 3 后，反而变成了形如的 一棵树，

5 5
4 3
3
2
1

出现了剪枝现象，经过 debug 发现是在递归寻找前驱和后继节点时递归出错，某一步返回了 NULL 指针，才剪掉了原树中的整个左子树，修改了算法后程序运行恢复了正常

4.2 算法分析

	存储结构	二叉链表
基本操作时间复杂度	Min(&T), Max(&T)	O(log n)
	Rotate(&T)	O(1)
	Find(&T)	O(log n)
	Create(&T)	O(1)
	Delete(x,&T)	O(log n)
	Insert(x,&T)	O(log n)
	Traverse(&T)	O(n)
	Print(&T,dep)	O(n)
	Destroy(&T)	O(n)
	Merge(&T1, &T2)	O(n)
	Spilt(&T1,&T2,x)	O(n)

而对于空间复杂度来说，平衡二叉树的遍历及其他类遍历操作空间复杂度为 O(logn)，即树高。而对于分裂和合并则利用了一个数组辅助空间，空间复杂度为 O(n),旋转操作等的空间复杂度均为 O(1)。

合并和分裂算法依然存在改进的余地，当先的时间复杂度为 O(n),最优情况下应该可以优化至 O(mlog(n/m+1))[1]。

4.3 体会

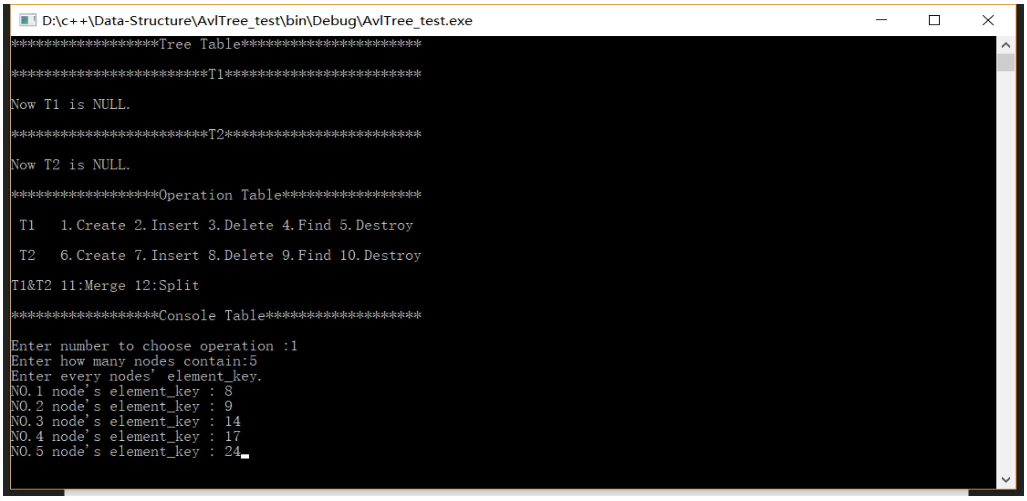
在编写递归程序中，一定要注意返回值和递归边界的设定，很容易牵一发而动全身，影响整个程序的运行。

5. 用户使用说明

1. 创建：选择操作 1，输入树 T1 包含的节点数，然后依次输入每个节点关键字的值
2. 插入：选择操作 2，输入想要树 T1 插入节点关键字的值
3. 删除：选择操作 3，输入想要从树 T1 中删除节点关键字的值
4. 寻找：选择操作 4，输入想要在树 T1 找到的节点关键字的值
5. 销毁：选择操作 5 来销毁树 T1
6. 创建：选择操作 6，输入树 T2 包含的节点数，然后依次输入每个节点关键字的值
7. 插入：选择操作 7，输入想要树 T2 插入节点关键字的值
8. 删除：选择操作 8，输入想要从树 T2 中删除节点关键字的值
9. 寻找：选择操作 9，输入想要在树 T2 找到的节点关键字的值
10. 销毁：选择操作 10 来销毁树 T2
11. 合并：选择操作 11，输入 1 以让 T1 合并至 T2，输入 2 以让 T2 合并至 T1
12. 分裂：选择操作 12，输入 1 以分裂 T1，输入 2 以分裂 T2，再输入分裂的关键字

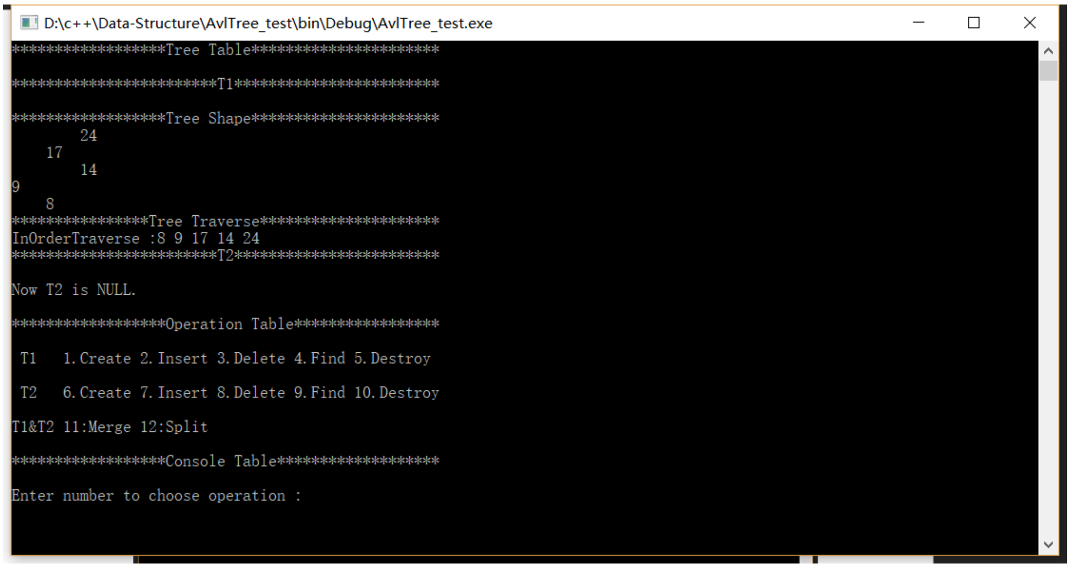
6. 测试结果

依次向 T1 插入 8，9，14，17，24



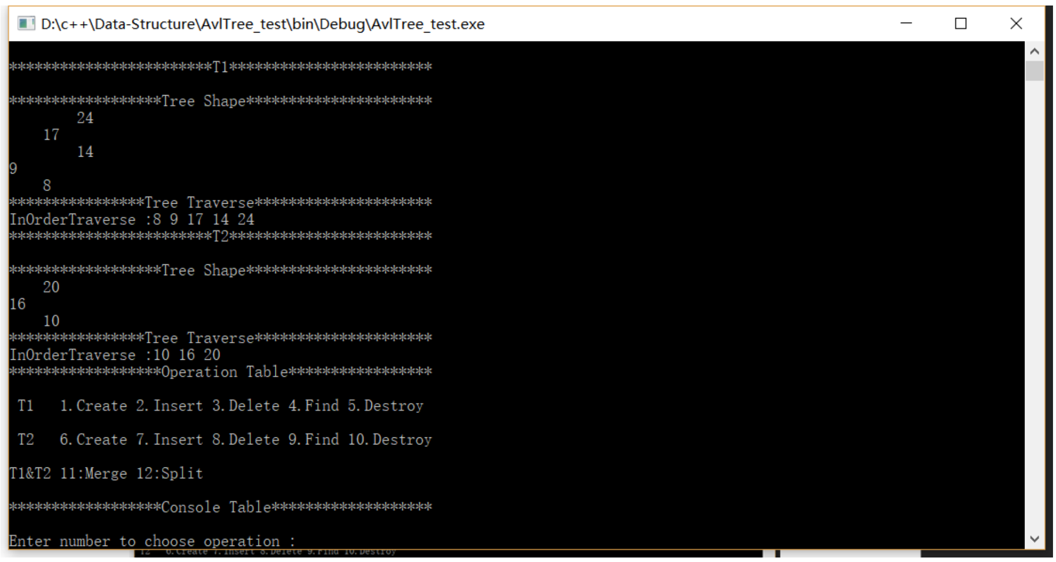
```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
******************Tree Table**********************

************************T1************************

Now T1 is NULL.

************************T2************************

Now T2 is NULL.

******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :1
Enter how many nodes contain:5
Enter every nodes' element_key.
NO.1 node's element_key : 8
NO.2 node's element_key : 9
NO.3 node's element_key : 14
NO.4 node's element_key : 17
NO.5 node's element_key : 24
```

插入后的结果



```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
******************Tree Table**********************

************************T1************************

******************Tree Shape**********************
        24
    17
        14
9
    8
****************Tree Traverse*********************
InOrderTraverse :8 9 17 14 24
************************T2************************

Now T2 is NULL.

******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :
```

向 T2 中依次插入 10，16，20 后



```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe

************************T1************************

******************Tree Shape**********************
        24
    17
        14
9
    8
****************Tree Traverse*********************
InOrderTraverse :8 9 17 14 24
************************T2************************

******************Tree Shape**********************
    20
16
    10
****************Tree Traverse*********************
InOrderTraverse :10 16 20
******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :
```

在 T1 中插入 30

```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
*************************T1*************************

******************Tree Shape*********************
        24
    17
        14
9
    8
****************Tree Traverse********************
InOrderTraverse :8 9 17 14 24
*************************T2*************************

******************Tree Shape*********************
    20
16
    10
****************Tree Traverse********************
InOrderTraverse :10 16 20
******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :2
Enter the element_key you want to insert :30
```

Parallelism in Algorithms and Architectures:253-264

插入后的 T1

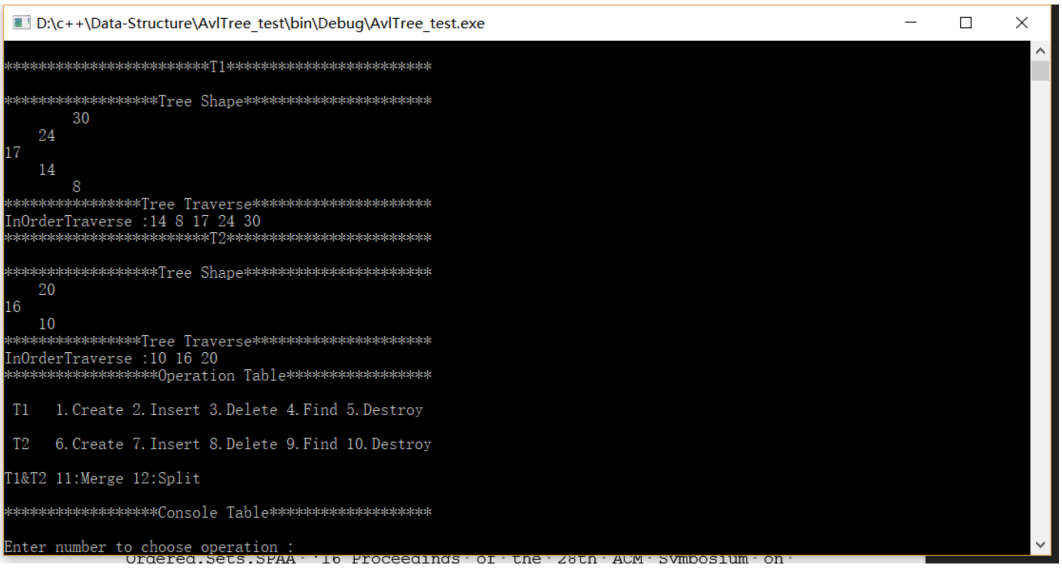
```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
*************************T1*************************

******************Tree Shape*********************
        30
    24
17
        14
    9
        8
****************Tree Traverse********************
InOrderTraverse :9 8 14 17 24 30
*************************T2*************************

******************Tree Shape*********************
    20
16
    10
****************Tree Traverse********************
InOrderTraverse :10 16 20
******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :
```

在 T1 中删除 9

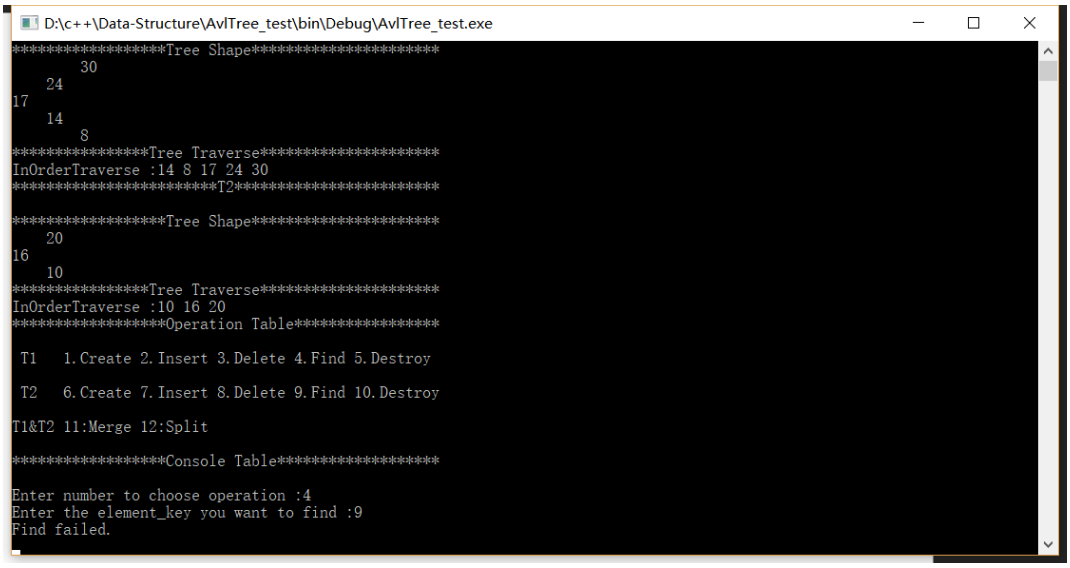
```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe

******************Tree Shape*********************
        30
    24
17
        14
    9
        8
****************Tree Traverse********************
InOrderTraverse :9 8 14 17 24 30
*************************T2*************************

******************Tree Shape*********************
    20
16
    10
****************Tree Traverse********************
InOrderTraverse :10 16 20
******************Operation Table*****************

 T1   1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2   6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table*******************

Enter number to choose operation :3
Enter the element_key you want to delete :9
```

Ordered Sets SPAA '16 Proceedings of the 28th ACM Symposium on

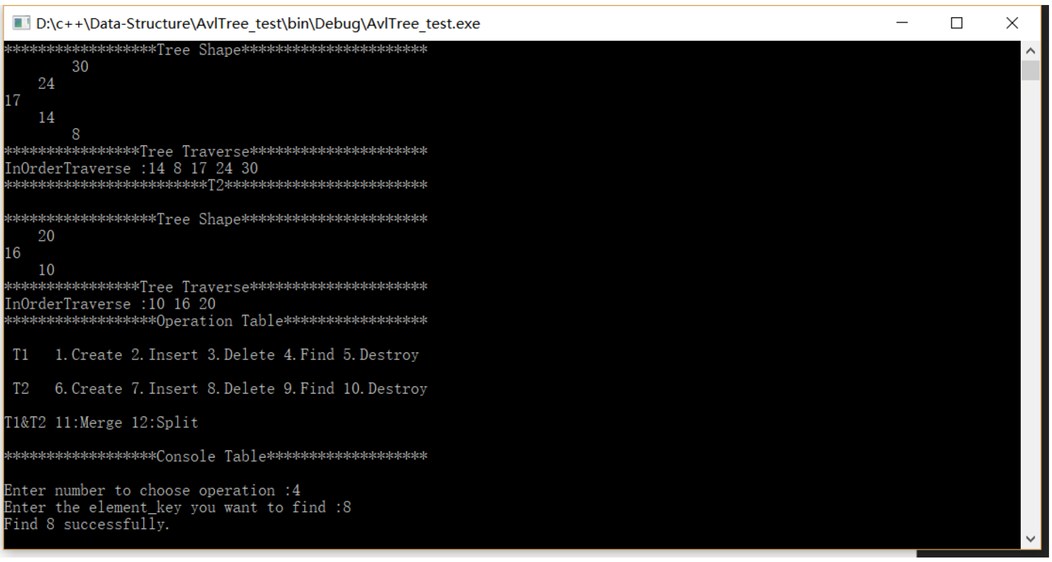
删除后的 T1



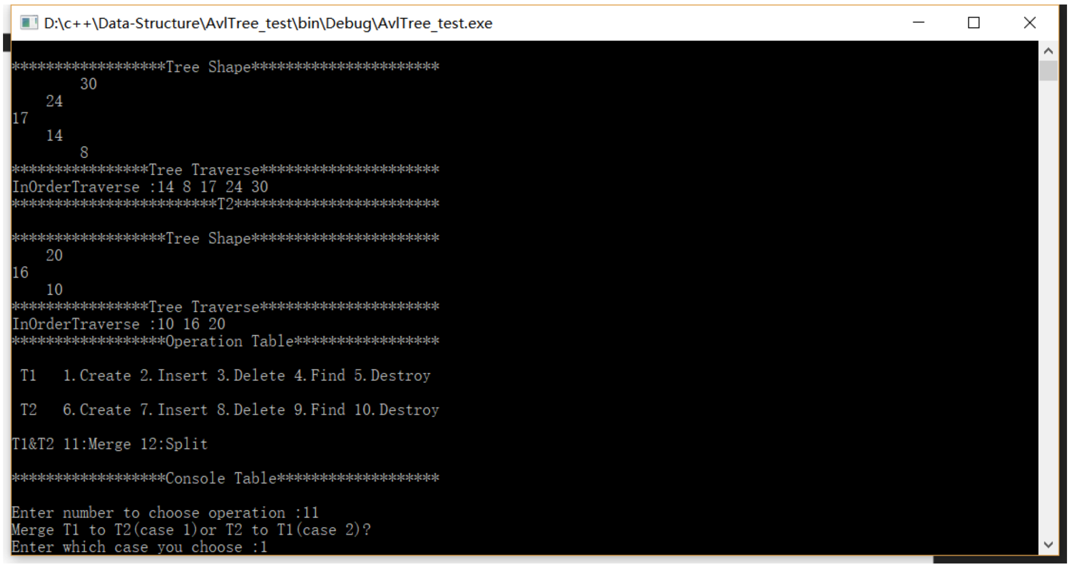
在 T1 中寻找 9



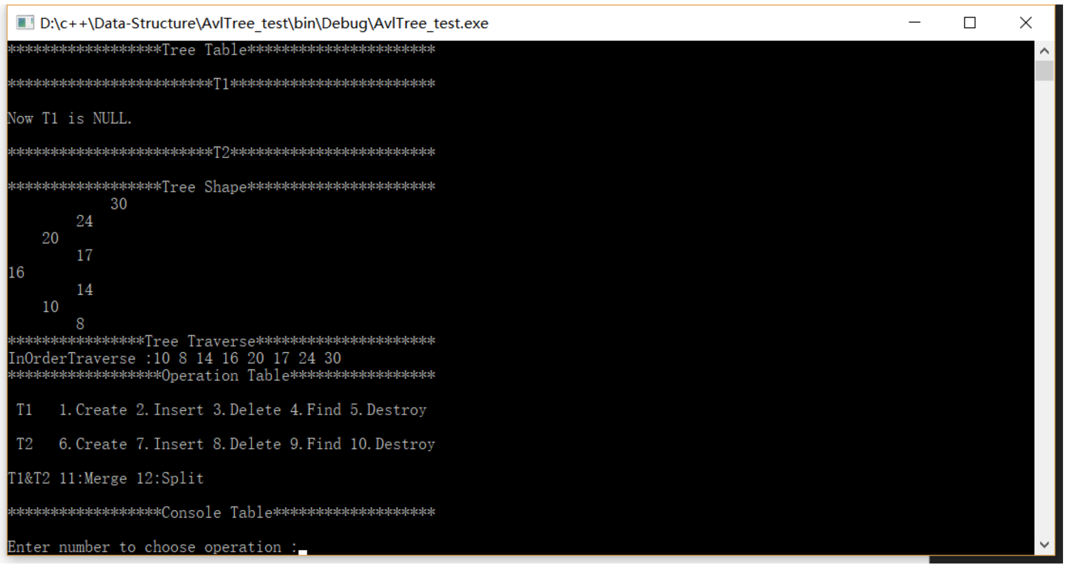
在 T1 中寻找 8



将 T1 合并至 T2



合并后的 T1，T2



将 T2 分裂，分裂关键字为 20

分裂后的 T1，T2

```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
******************Tree Table**********************
*************************T1************************
******************Tree Shape**********************
    30
24
****************Tree Traverse*********************
InOrderTraverse :24 30
*************************T2************************
******************Tree Shape**********************
    20
        17
16
        14
    10
        8
****************Tree Traverse*********************
InOrderTraverse :10 8 14 16 20 17
******************Operation Table******************

 T1    1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2    6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table********************
```

销毁 T1

```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
******************Tree Shape**********************
    30
24
****************Tree Traverse*********************
InOrderTraverse :24 30
*************************T2************************
******************Tree Shape**********************
    20
        17
16
        14
    10
        8
****************Tree Traverse*********************
InOrderTraverse :10 8 14 16 20 17
******************Operation Table******************

 T1    1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2    6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table********************

Enter number to choose operation :5
Destroy T1 Successfully.
```

销毁 T1 后的 T1，T2

```
D:\c++\Data-Structure\AvlTree_test\bin\Debug\AvlTree_test.exe
******************Tree Table**********************
*************************T1************************
Now T1 is NULL.
*************************T2************************
******************Tree Shape**********************
    20
        17
16
        14
    10
        8
****************Tree Traverse*********************
InOrderTraverse :10 8 14 16 20 17
******************Operation Table******************

 T1    1.Create 2.Insert 3.Delete 4.Find 5.Destroy

 T2    6.Create 7.Insert 8.Delete 9.Find 10.Destroy

T1&T2 11:Merge 12:Split

******************Console Table********************

Enter number to choose operation :
```

7. 参考文献

[1] Guy.E.Blelloch,Daniel.Ferizovic,Yihan.Sun.Just Join for Parallel Ordered.Sets.SPAA '16 Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures.253-264

8. 附录

```
卷 Data 的文件夹 PATH 列表
卷序列号为 D041-625A
D:.
│  3116004982_zsy.pdf
│  main.cpp
│  main.h
│
├─bin
│  └─Debug
│          AvlTree.exe
```