数据结构实验报告

课程名称：数据结构	班级：计科 151	
姓名： 刘禾子	学号：1500170082	指导教师：刘长云
实验序号：六		实验成绩：

一、实验名称

存储管理、查找和排序

二、实验目的及要求

1、建立平衡二叉树并进行创建、增加、删除、调平等操作。
2、设计一个实现平衡二叉树的程序，可进行创建、增加、删除、调平等操作，实现动态的输入数据，实时的输出该树结构。

三、实验环境

Visual C++

四、实验内容

6.4⑤ 平衡二叉树操作的演示

【问题描述】

利用平衡二叉树实现一个动态查找表。

【基本要求】

实现动态查找表的三种基本功能：查找、插入和删除。

【测试数据】

由读者自行设定。

【实现提示】

(1) 初始，平衡二叉树为空树，操作界面给出查找、插入和删除三种操作供选择。每种操作均要提示输入关键字。每次插入或删除一个结点后，应更新平衡二叉树的显示。

(2) 平衡二叉树的显示可采用如 6.3 题要求的凹入表形式，也可以采用图形界面画出树形。

(3) 教科书已给出查找和插入算法，本题重点在于对删除算法的设计和实现。假设要删除关键字为 x 的结点。如果 x 不在叶子结点上，则用它左子树中的最大值或右子树中的最小值取代 x。如此反复取代，直到删除动作传递到某个叶子结点。删除叶子结点时，若需要进行平衡变换，可采用插入的平衡变换的反变换(如，左子树变矮对应于右子树长高)。

五、算法描述及实验步骤

算法描述：

平衡二叉树是在构造二叉排序树的过程中，每当插入一个新结点时，首先检查是否因插入新结点而破坏了二叉排序树的平衡性，若是，则找出其中的最小不平衡子树，在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。

具体步骤如下：

⑴ 每当插入一个新结点，从该结点开始向上计算各结点的平衡因子，即计算该结点的祖先结点的平衡因子，若该结点的祖先结点的平衡因子的绝对值均不超过1，则平衡二叉树没有失去平衡，继续插入结点；

⑵ 若插入结点的某祖先结点的平衡因子的绝对值大于 1，则找出其中最小不平衡子树的根结点；

⑶ 判断新插入的结点与最小不平衡子树的根结点的关系，确定是哪种类型的调整；

⑷ 如果是 LL 型或 RR 型，只需应用扁担原理旋转一次，在旋转过程中，如果出现冲突，应用旋转优先原则调整冲突；如果是 LR 型或 RL 型，则需应用扁担原理旋转两次，第一次最小不平衡子树的根结点先不动，调整插入结点所在子树，第二次再调整最小不平衡子树，在旋转过程中，如果出现冲突，应用旋转优先原则调整冲突；

⑸ 计算调整后的平衡二叉树中各结点的平衡因子，检验是否因为旋转而破坏其他结点的平衡因子，以及调整后的平衡二叉树中是否存在平衡因子大于 1 的结点。

六、调试过程及实验结果

调试过程：

在进行对插入新结点并调平时由于利用的是普通的插入方法进行 LL、LR、RL、RR 型的转换，使得在调试时经常没有更改内部变量的值，导致编译出错。

对于在空树情况下删除结点的考虑，是在后期的调试检验过程中发现的。在没有更改代码前，如果按此操作，程序就会崩溃。原因就是在删除函数中虽然考虑到了空树的情况，但是在输出树的函数中没有加入空树的考虑而只是在创建树函数中加入了 if…else…的判断。经过反复的检查，发现可以直接在输出函数中加入判断而不必再其他位置判断，并且调试成功。

程序清单：

```
#include<stdarg.h>
#include <stdio.h>
#include <malloc.h>
#define EQ(a,b) ((a)==(b))
#define LT(a,b)  ((a)<(b))
#define LQ(a,b)  ((a)>(b))
#define LH +1     //左高
#define EH 0      //等高
#define RH -1     //右高
typedef struct BTNode
{
```

```
    int data;
    int bf;                          //平衡因子
    struct BTNode *lchild, *rchild;//左、右孩子
}BTNode, *BTree;
/*需要的函数声明*/
void Right_Balance(BTree &p);
void Left_Balance(BTree &p);
void Left_Root_Balance(BTree &T);
void Right_Root_Balance(BTree &T);
bool InsertAVL(BTree &T, int i, bool &taller);
void PrintBT(BTree T, int m);
void CreatBT(BTree &T);
void Left_Root_Balance_det(BTree &p, int &shorter);
void Right_Root_Balance_det(BTree &p, int &shorter);
void Delete(BTree q, BTree  &r, int &shorter);
int DeleteAVL(BTree &p, int x, int &shorter);
void Adj_balance(BTree &T);
bool SetAVL(BTree &T, int i, bool &taller);
bool Insert_Balance_AVL(BTree &T, int i, bool &taller);
/*主函数*/
void main()
{
    int input, search, m;
    bool taller = false;
    int shorter = 0;
    BTree T;
    T = (BTree)malloc(sizeof(BTNode));
    T = NULL;
    while (1)
    {
        printf("\n请选择需要的二叉树操作\n");
        printf("1.创建二叉树2.增加新结点3.直接创建平衡二叉树4.在平衡二叉树上增加新结点
并调平衡5.删除0.退出\n");
        scanf("%d", &input);
        getchar();
        switch (input)
        {
        case 1:
            CreatBT(T);
            break;
        case 2:
            printf("请输入你要增加的关键字");
            scanf("%d", &search);
            getchar();
```

```
            InsertAVL(T, search, taller);
            m = 0;
            PrintBT(T, m);
            break;
        case 3:
            Adj_balance(T);
            break;
        case 4:
            printf("请输入你要增加的关键字");
            scanf("%d", &search);
            getchar();
            SetAVL(T, search, taller);
            m = 0;
            PrintBT(T, m);
            break;
        case 5:
            printf("请输入你要删除的关键字");
            scanf("%d", &search);
            getchar();
            DeleteAVL(T, search, shorter);
            m = 0;
            PrintBT(T, m);
            break;
        case 0:
            break;
        default:
            printf("输入错误，请重新选择。");
            break;
        }
        if (input == 0)
            break;
        printf("按任意键继续.");
        getchar();
    }
}
/*对以*p为根的二叉排序树作右旋处理*/
void Right_Balance(BTree &p)
{
    BTree lc;
    lc = p->lchild;          //lc指向的*p左子树根结点
    p->lchild = lc->rchild; //rc的右子树挂接为*p的左子树
    lc->rchild = p;
    p = lc; //p指向新的结点
}
```

```
/*对以*p为根的二叉排序树作左旋处理*/
void Left_Balance(BTree &p)
{
    BTree rc;
    rc = p->rchild;          //指向的*p右子树根结点
    p->rchild = rc->lchild; //rc左子树挂接到*p的右子树
    rc->lchild = p;
    p = rc; //p指向新的结点
}
/*对以指针T所指结点为根的二叉树作左平衡旋转处理*/
void Left_Root_Balance(BTree &T)
{
    BTree lc, rd;
    lc = T->lchild;          //指向*T的左子树根结点
    switch (lc->bf)           //检查*T的左子树的平衡度，并作相应平衡处理
    {
    case LH:                 //新结点插入在*T的左孩子的左子树上，要作单右旋处理
        T->bf = lc->bf = EH;
        Right_Balance(T);
        break;
    case RH:                 //新结点插入在*T的左孩子的右子树上，要作双旋处理
        rd = lc->rchild;     //rd指向*T的左孩子的右子树根
        switch (rd->bf)       //修改*T及其左孩子的平衡因子
        {
        case LH:
            T->bf = RH;
            lc->bf = EH;
            break;
        case EH:
            T->bf = lc->bf = EH;
            break;
        case RH:
            T->bf = EH;
            lc->bf = LH;
            break;
        }
        rd->bf = EH;
        Left_Balance(T->lchild); //对*T的左子树作左旋平衡处理
        Right_Balance(T);         //对*T作右旋平衡处理
    }
}
/*对以指针T所指结点为根的二叉树作右平衡旋转处理*/
void Right_Root_Balance(BTree &T)
{
```

```
    BTree rc, ld;
    rc = T->rchild;          //指向*T的左子树根结点
    switch (rc->bf)          //检查*T的右子树的平衡度，并作相应平衡处理
    {
    case RH:                 //新结点插入在*T的右孩子的右子树上，要作单左旋处理
        T->bf = rc->bf = EH;
        Left_Balance(T); break;
    case LH:                 //新结点插入在*T的右孩子的左子树上，要作双旋处理
        ld = rc->lchild;     //ld指向*T的右孩子的左子树根
        switch (ld->bf)      //修改*T及其右孩子的平衡因子
        {
        case LH:
            T->bf = EH;
            rc->bf = RH;
            break;
        case EH:
            T->bf = rc->bf = EH;
            break;
        case RH:
            T->bf = LH;
            rc->bf = EH;
            break;
        }
        ld->bf = EH;
        Right_Balance(T->rchild);//对*T的右子树作左旋平衡处理
        Left_Balance(T);         //对*T作左旋平衡处理
    }
}
/*插入结点i,若T中存在和i相同关键字的结点，则插入一个数据元素为i的新结点，并返回 1，否则返回 0*/
bool InsertAVL(BTree &T, int i, bool &taller)
{
    if (!T)//插入新结点，树“长高”，置taller为true
    {
        T = (BTree)malloc(sizeof(BTNode));
        T->data = i;
        T->lchild = T->rchild = NULL;
        T->bf = EH;
        taller = true;
    }
    else
    {
        if (EQ(i, T->data))                 //树中已存在和有相同关键字的结点
        {
```

```
            taller = false;
            printf("已存在相同关键字的结点\n");
            return 0;
        }
        if (LT(i, T->data))                    //应继续在*T的左子树中进行搜索
        {
            if (!InsertAVL(T->lchild, i, taller))
                return 0;
        }
        else                                   //应继续在*T的右子树中进行搜索
        {
            if (!InsertAVL(T->rchild, i, taller))
                return 0;
        }
    }
    return 1;
}
/*按树状打印输出二叉树的元素,m表示结点所在层次*/
void PrintBT(BTree T, int m)
{
    if (T)
    {
        int i;
        if (T->rchild)
            PrintBT(T->rchild, m + 1);
        for (i = 1; i <= m; i++)
            printf("      ");//打印i 个空格以表示出层次
        printf("%d\n", T->data);//打印T 元素,换行
        if (T->lchild)
            PrintBT(T->lchild, m + 1);
    }
    else
    {
        printf("这是一棵空树！\n");
        getchar();
    }
}
/*创建二叉树，以输入-32767为建立的结束*/
void CreatBT(BTree &T)
{
    int m;
    int i;
    bool taller = false;
    T = NULL;
```

```
    printf("\n请输入关键字(以-32767结束建立二叉树):");
    scanf("%i", &i);
    getchar();
    while (i != -32767)
    {
        InsertAVL(T, i, taller);
        printf("\n请输入关键字(以-32767结束建立二叉树):");
        scanf("%i", &i);
        getchar();
        taller = false;
    }
    m = 0;
    printf("您创建的二叉树为：\n");
    PrintBT(T, m);
}
/*删除结点时左平衡旋转处理*/
void Left_Root_Balance_det(BTree &p, int &shorter)
{
    BTree  p1, p2;
    if (p->bf == 1) //p结点的左子树高，删除结点后p的bf减,树变矮
    {
        p->bf = 0;
        shorter = 1;
    }
    else if (p->bf == 0)//p结点左、右子树等高，删除结点后p的bf减,树高不变
    {
        p->bf = -1;
        shorter = 0;
    }
    else  //p结点的右子树高
    {
        p1 = p->rchild;//p1指向p的右子树
        if (p1->bf == 0)//p1结点左、右子树等高,删除结点后p的bf为-2,进行左旋处理，树高
不变
        {
            Left_Balance(p);
            p1->bf = 1;
            p->bf = -1;
            shorter = 0;
        }
        else if (p1->bf == -1)//p1的右子树高，左旋处理后，树变矮
        {
            Left_Balance(p);
            p1->bf = p->bf = 0;
```

```
            shorter = 1;
        }
        else //p1的左子树高,进行双旋处理(先右旋后左旋), 树变矮
        {
            p2 = p1->lchild;
            p1->lchild = p2->rchild;
            p2->rchild = p1;
            p->rchild = p2->lchild;
            p2->lchild = p;
            if (p2->bf == 0)
            {
                p->bf = 0;
                p1->bf = 0;
            }
            else if (p2->bf == -1)
            {
                p->bf = 1;
                p1->bf = 0;
            }
            else
            {
                p->bf = 0;
                p1->bf = -1;
            }
            p2->bf = 0;
            p = p2;
            shorter = 1;
        }
    }
}
/*删除结点时右平衡旋转处理*/
void Right_Root_Balance_det(BTree &p, int &shorter)
{
    BTree  p1, p2;
    if (p->bf == -1)
    {
        p->bf = 0;
        shorter = 1;
    }
    else if (p->bf == 0)
    {
        p->bf = 1;
        shorter = 0;
    }
```

```
    else
    {
        p1 = p->lchild;
        if (p1->bf == 0)
        {
            Right_Balance(p);
            p1->bf = -1;
            p->bf = 1;
            shorter = 0;
        }
        else if (p1->bf == 1)
        {
            Right_Balance(p);
            p1->bf = p->bf = 0;
            shorter = 1;
        }
        else
        {
            p2 = p1->rchild;
            p1->rchild = p2->lchild;
            p2->lchild = p1;
            p->lchild = p2->rchild;
            p2->rchild = p;
            if (p2->bf == 0)
            {
                p->bf = 0;
                p1->bf = 0;
            }
            else if (p2->bf == 1)
            {
                p->bf = -1;
                p1->bf = 0;
            }
            else
            {
                p->bf = 0;
                p1->bf = 1;
            }
            p2->bf = 0;
            p = p2;
            shorter = 1;
        }
    }
}
```

```
/*删除结点*/
void Delete(BTree q, BTree  &r, int &shorter)
{
    if (r->rchild == NULL)
    {
        q->data = r->data;
        q = r;
        r = r->lchild;
        free(q);
        shorter = 1;
    }
    else
    {
        Delete(q, r->rchild, shorter);
        if (shorter == 1)
            Right_Root_Balance_det(r, shorter);
    }
}
/*二叉树的删除操作*/
int DeleteAVL(BTree &p, int x, int &shorter)
{
    int k;
    BTree q;
    if (p == NULL)
    {
        printf("不存在要删除的关键字!!\n");
        return 0;
    }
    else if (x<p->data)//在p的左子树中进行删除
    {
        k = DeleteAVL(p->lchild, x, shorter);
        if (shorter == 1)
            Left_Root_Balance_det(p, shorter);
        return k;
    }
    else if (x>p->data)//在p的右子树中进行删除
    {
        k = DeleteAVL(p->rchild, x, shorter);
        if (shorter == 1)
            Right_Root_Balance_det(p, shorter);
        return k;
    }
    else
    {
```

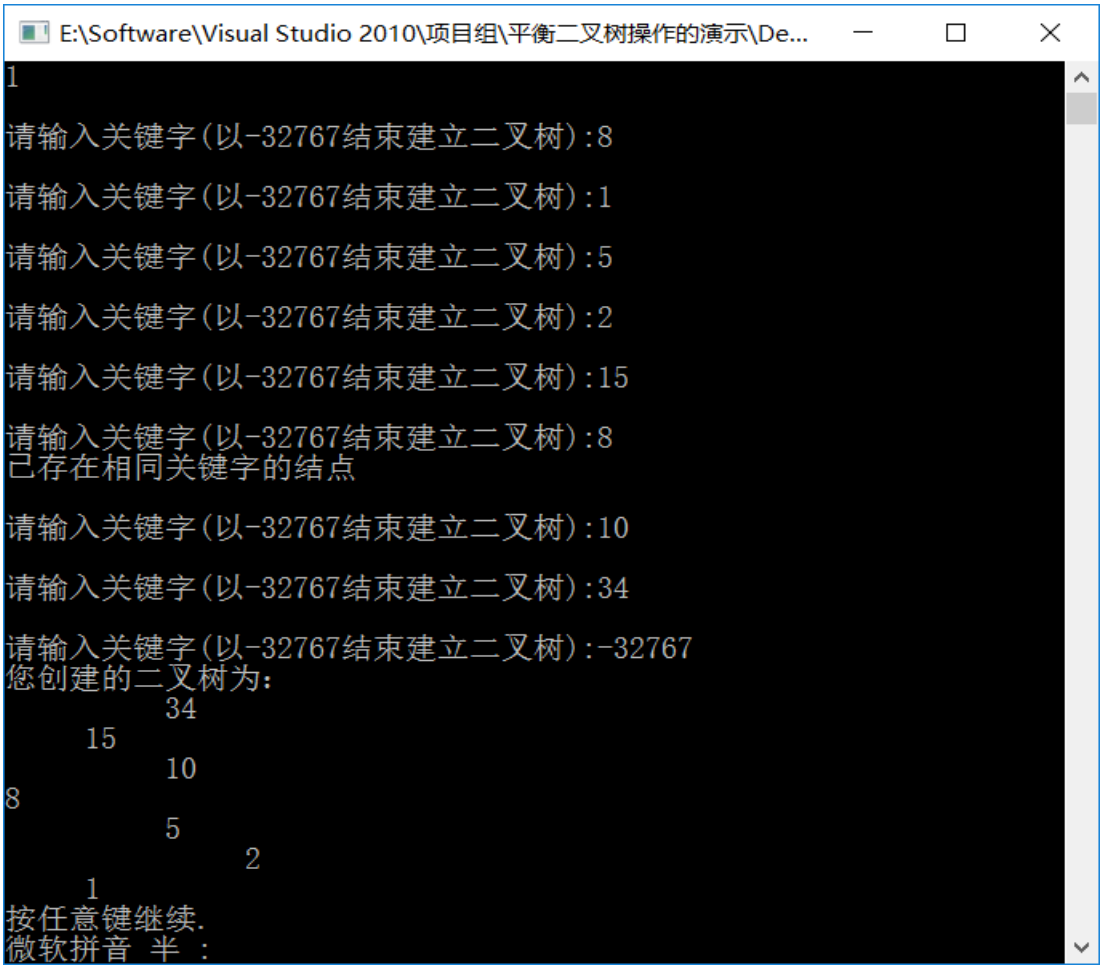
```
            q = p;
            if (p->rchild == NULL) //右子树空则只需重接它的左子树
            {
                p = p->lchild;
                free(q);
                shorter = 1;
            }
            else if (p->lchild == NULL)//左子树空则只需重接它的右子树
            {
                p = p->rchild;
                free(q);
                shorter = 1;
            }
            else//左右子树均不空
            {
                Delete(q, q->lchild, shorter);
                if (shorter == 1)
                    Left_Root_Balance_det(p, shorter);
                p = q;
            }
            return 1;
        }
    }
    /*二叉树调平操作*/
    void Adj_balance(BTree &T)
    {
        int m;
        int i;
        bool taller = false;
        T = NULL;
        printf("\n请输入关键字(以-32767结束建立平衡二叉树):");
        scanf("%d", &i);
        getchar();
        while (i != -32767)
        {
            SetAVL(T, i, taller);
            printf("\n请输入关键字(以-32767结束建立平衡二叉树):");
            scanf("%d", &i);
            getchar();
            taller = false;
        }
        m = 0;
        printf("平衡二叉树创建结束.\n");
        if (T)
```

```
        PrintBT(T, m);
    else
        printf("这是一棵空树.\n");
}
/*调平二叉树具体方法*/
bool SetAVL(BTree &T, int i, bool &taller)
{
    if (!T)//插入新结点，树“长高”，置taller为true
    {
        T = (BTree)malloc(sizeof(BTNode));
        T->data = i;
        T->lchild = T->rchild = NULL;
        T->bf = EH;
        taller = true;
    }
    else
    {
        if (EQ(i, T->data))                    //树中已存在和有相同关键字的结点
        {
            taller = false;
            printf("已存在相同关键字的结点\n");
            return 0;
        }
        if (LT(i, T->data))                    //应继续在*T的左子树中进行搜索
        {
            if (!SetAVL(T->lchild, i, taller))
                return 0;
            if (taller)                        //已插入到*T的左子树中且左子树“长高”
                switch (T->bf)                 //检查*T的平衡度
            {
                case LH:                   //原本左子树比右子树高，需要作左平衡处理
                    Left_Root_Balance(T);
                    taller = false;
                    break;
                case EH:                   //原本左子树、右子等高，现因左子树增高而使树
增高
                    T->bf = LH;
                    taller = true;
                    break;
                case RH:                   //原本右子树比左子树高，现左、右子树等高
                    T->bf = EH;
                    taller = false;
                    break;
            }
```

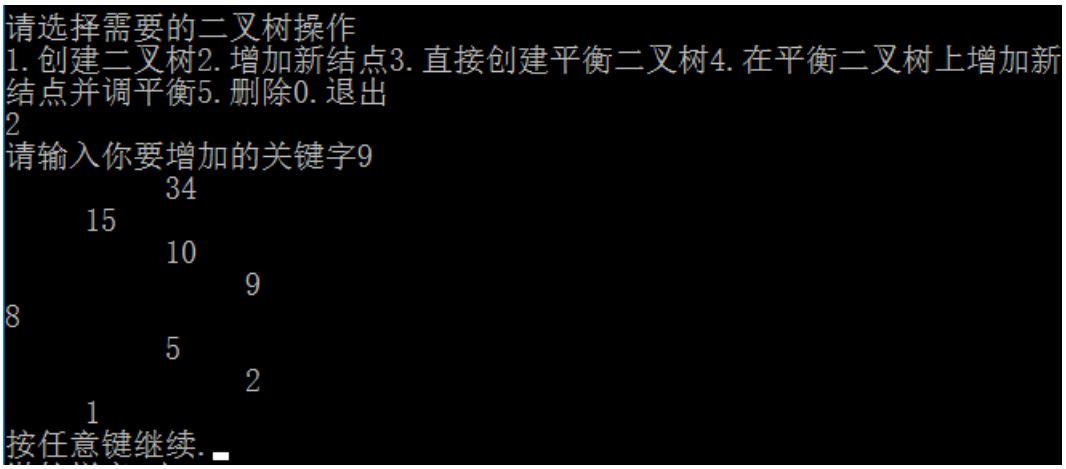
```
        }
        else                                    //应继续在*T的右子树中进行搜索
        {
            if (!SetAVL(T->rchild, i, taller))
                return 0;
            if (taller)                         //已插入到*T的右子树中且右子树“长高”
                switch (T->bf)                  //检查*T的平衡度
            {
                case LH:                    //原本左子树比右子树高，现左、右子树等高
                    T->bf = EH;
                    taller = false;
                    break;
                case EH:                    //原本左子树、右子等高，现因右子树增高而使树增
高
                    T->bf = RH;
                    taller = true;
                    break;
                case RH:                    //原本右子树比左子树高，需要作右平衡处理
                    Right_Root_Balance(T);
                    taller = false;
                    break;
            }
        }
        return 1;
    }
}
```

实验结果：

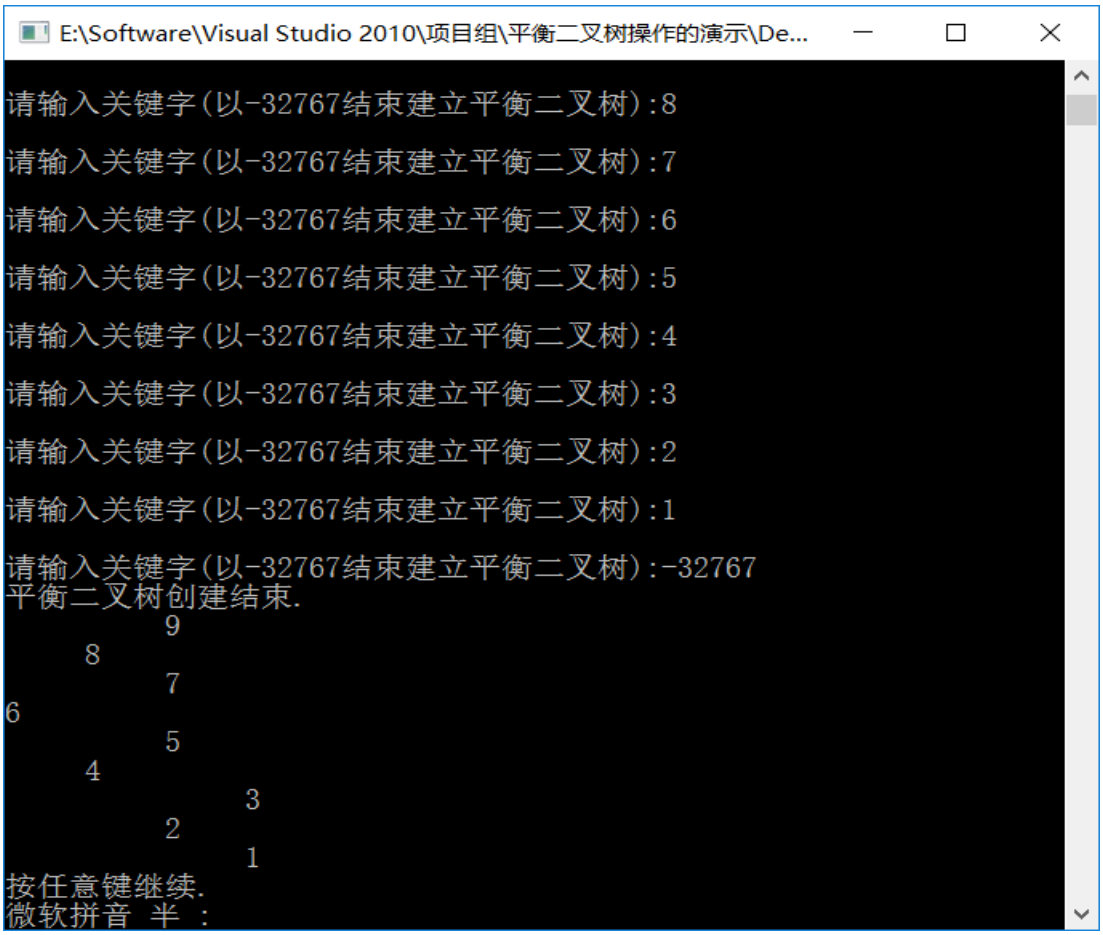
创建二叉树：



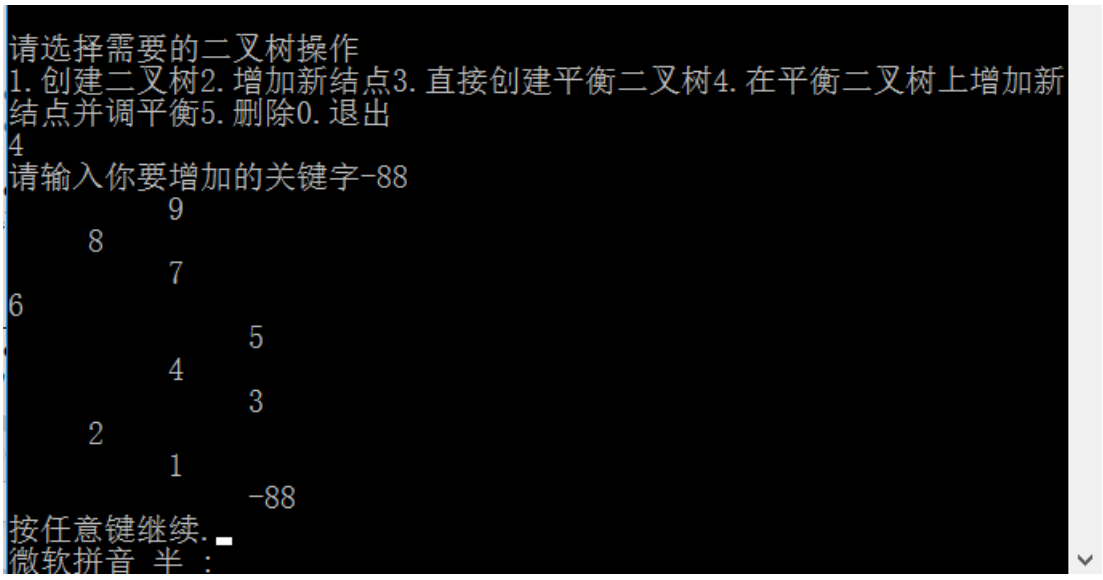
增加二叉树：



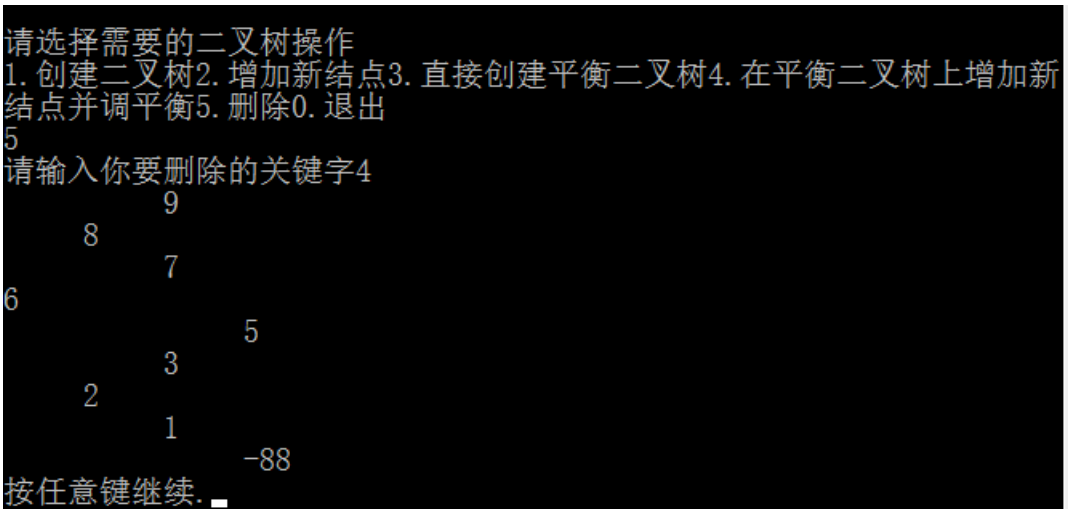
直接创建平衡二叉树：



平衡二叉树加入新节点并调平：



删除结点：



七、总结

1.经过此次实验深刻理解了建立树的方法；
2.掌握了利用二分法建立树结构和二叉树的调平方法；
3.掌握了向一个已知树插入或删除结点的方法。