作业 HW3 实验报告

姓名：张三 学号：2054170 日期：2021 年 11 月 16 日

实验报告格式按照模板来即可，对字体大小、缩进、颜色等不做要求

实验报告要求在文字简洁的同时将内容表示清楚

1. 涉及数据结构和相关背景

2. 实验内容

2.1

HW3 树和二叉树 > 二叉树的创建与遍历
二叉树的创建与遍历
描述
二叉树是 n（n>=0）个结点的有限集合，它或为空树，或是由一个称之为根的结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成。
本题练习二叉树的基本操作，包括先序建立二叉树，先序遍历、中序遍历、后序遍历、层次遍历二叉树，输出二叉树的树型图。

输入
一行，输入先序序列，内部结点用一个字符表示，空结点用#表示
对于 20%的数据，0<n<=10
对于 40%的数据，0<n<=100
对于 100%的数据，0<n<=100000
注意：
[1]节点个数不等于输入字符串长度
[2]保证每个节点用一个字符表示，但不同节点的字符可能相同
从右上角下载并运行 p37_data.cpp 以生成随机测试数据
输出
多行
第 1 行，输出先序遍历序列
第 2 行，输出中序遍历序列
第 3 行，输出后序遍历序列
第 4 行，输出层次遍历序列
输出二叉树的树型图(逆时针旋转 90 度)，每一层之间输出 5 个空格

```
typedef struct BiTNode { // 二叉链表的定义
   char data;
    struct BiTNode* lchild, * rchild;
}BiTNode, * BiTree;

int CreateBiTree(BiTree& T)
```

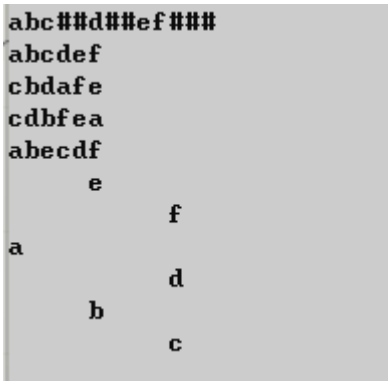
```
{
    char ch; cin >> ch;
    if (ch == '#') T = NULL;
    else {
        T = new BiTNode;
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return 1;
}
void InOrderTraverse(BiTree T)//中序遍历
{
    if (T)
    {
        InOrderTraverse(T->lchild);
        cout << T->data;
        InOrderTraverse(T->rchild);
    }
}
void PreOrderTraverse(BiTree T)//先序遍历
{

    if (T)
    {
        cout << T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
}
void PostOrderTraverse(BiTree T)//后序遍历
{
    if (T)
    {
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        cout << T->data;
    }
}
void FloorPrint_QUEUE(BiTree Tree) //层序遍历_队列实现
{
    queue < BiTNode> q;
    if (Tree != NULL)
    {
```

```
        q.push(*Tree);   //根节点进队列
    }

    while (q.empty() == false)  //队列不为空判断
    {
        cout << q.front().data ;

        if (q.front().lchild != NULL)   //如果有左孩子，leftChild入队列
        {
            q.push(*q.front().lchild);
        }

        if (q.front().rchild != NULL)   //如果有右孩子，rightChild入队列
        {
            q.push(*q.front().rchild);
        }
        q.pop();  //已经遍历过的节点出队列
    }
}


int NodeCount(BiTree T)
{//统计二叉树中结点的个数
    if (T == NULL) return 0;
    else return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;
}


void prtbtree(BiTNode* p, int cur)//逆时针旋转90度输出二叉树
{
    if (p)
    {
        prtbtree(p->rchild, cur + 1);

        for (int i = 1; i <= cur; i++)
            printf("     ");
        printf("%c", p->data);

        printf("\n");
        prtbtree(p->lchild, cur + 1);
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

简要描述调试过程

可以使用图片辅助说明



2.1.6 总结和体会

可以说说做此题的收获、分析这道题目的难点和易错点（数据边界、对算法效率的要求等）

定义三个标记 pc 标记当前访问的结点 ，pp 标记上一次访问过的结点， pt 标记栈顶的结点。先标记根节点为当前节点，从当前结点开始，找到当前结点最左边的子树，并将路径上的结点一次压栈，取出栈顶元素，如果其右子树为空或是 pp 所标记的结点，则说明当前结点的右子树已经被访问，就访问栈顶结点并将栈顶结点标记为 pp,然后出。当 pc 指向空且栈为空时， 则说明遍历结束。

2.2 题目二

HW3 树和二叉树 > 二叉树的同构
二叉树的同构
描述
给定两棵树 T1 和 T2。如果 T1 可以通过若干次左右孩子互换变成 T2，则我们称两棵树是“同构”的。例如图 1 给出的两棵树就是同构的，因为我们把其中一棵树的结点 A、B、G 的左右孩子互换后，就得到另外一棵树。而图 2 就不是同构的。

图 1

图 2

现给定两棵树，请你判断它们是否是同构的。并计算每棵树的深度。

输入:
第一行是一个非负整数 N1，表示第 1 棵树的结点数;

随后 N 行，依次对应二叉树的 N 个结点（假设结点从 0 到 N−1 编号），每行有三项，分别是 1 个英文大写字母、其左孩子结点的编号、右孩子结点的编号。如果孩子结点为空，则在相应位置上给出“-”。给出的数据间用一个空格分隔。

接着一行是一个非负整数 N2，表示第 2 棵树的结点数;

随后 N 行同上描述一样，依次对应二叉树的 N 个结点。

对于 20%的数据，有 0<N1=N2<=10

对于 40%的数据，有 0<=N1=N2<=100

对于 100%的数据，有 0<=N1,N2<=10100

注意：题目不保证每个结点中存储的字母是不同的。

下载 p81_data.cpp 并编译运行以生成随机测试数据

输出:
共三行。

第一行，如果两棵树是同构的，输出“Yes”，否则输出“No”。

后面两行分别是两棵树的深度。

```
struct Btree
{
    char data;
    Btree* lchild;
    Btree* rchild;
};
```

```
bool judge(Btree* B1, Btree* B2)//判断是否同构
{

    if (B1 == NULL && B2 == NULL)
    {
        return true;
    }
    else if (B1 == NULL && B2 != NULL || B2 == NULL && B1 != NULL)
        return false;
    else if (B1 != NULL && B2 != NULL)
    {
        if (B1->data == B2->data)
        {
            if (B1->lchild && B1->rchild && B2->lchild && B2->rchild &&
                B1->lchild->data == B1->rchild->data && B2->lchild->data ==
B2->rchild->data)
                return (judge(B1->lchild, B2->lchild) && judge(B1->rchild,
B2->rchild) ||
                    judge(B1->lchild, B2->rchild) && judge(B1->rchild, B2->lchild));
            else if (B1->lchild && B2->lchild && B1->lchild->data == B2->lchild->data)
            {
                return (judge(B1->lchild, B2->lchild) && judge(B1->rchild,
B2->rchild));
            }
            else if (B1->lchild && B2->rchild && B1->lchild->data == B2->rchild->data)
            {
                return (judge(B1->lchild, B2->rchild) && judge(B1->rchild,
B2->lchild));
            }
            else if (B1->rchild && B2->lchild && B1->rchild->data == B2->lchild->data)
            {
                return (judge(B1->rchild, B2->lchild) && judge(B1->lchild,
B2->rchild));
            }
            else if (B1->rchild && B2->rchild && B1->rchild->data == B2->rchild->data)
            {

                return (judge(B1->rchild, B2->rchild) && judge(B1->lchild,
B2->lchild));
            }
            else if (!B1->lchild && !B1->rchild && !B2->lchild && !B2->rchild)
                return true;
            else
```

```
            {
                return false;
            }
        }
        else
            return false;
    }
    return false;
}

int fine_root(char*** Tree, int level)//结点数
{
    int n = 0;
    int num = 0;
    for (int i = 0; i < level; ++i)
    {
        for (int k = 1; k < 3; ++k)
            if (Tree[i][k][0] != '-')
                n += atoi(Tree[i][k]);
        num += i;
    }
    return num - n;
}

void InitTree(Btree** B, char*** Tree, int pos, int& n, int& m)//初始化
{

    if (*B == NULL)
    {

        *B = new Btree;
        (*B)->data = Tree[pos][0][0];
        (*B)->lchild = NULL;
        (*B)->rchild = NULL;
    }
    if (m < n)
        m = n;
    if (strcmp(Tree[pos][1], "-") != 0)
    {
        InitTree(&(*B)->lchild, Tree, atoi(Tree[pos][1]), ++n, m);
        --n;
    }
    if (strcmp(Tree[pos][2], "-") != 0)
    {
```

```
        InitTree(&(*B)->rchild, Tree, atoi(Tree[pos][2]), ++n, m);
        --n;
    }

    return;
}
```

```
6
A 1 4
B 2 3
C - -
D - -
E 5 -
F - -
6
B 4 3
F - -
A 5 0
C - -
D - -
E 1 -
Yes
3
3
```

给定两棵树 T1 和 T2。如果 T1 可以通过若干次左右孩子互换就变成 T2，则我们称两棵树是“同构”的。
步骤：
1.建立两颗二叉树 2.判断这两颗树是否同构
核心：同构算法如果两个二叉树为空，则是同构其中一个是空树，一个不是空树，则非同构
树根的值不一样，非同构
左孩子为空，则判断右孩子是否是同构，左孩子非空，且左孩子相同
则判断两个树的左子树与左子树是否同构，右子树与右子树是否同构，否则判断 T1 左子树和 T2 右子树是否同构，T1 右子树和 T2 左子树是否同构
至于如何判定根节点，可以遍历所有结点，排除有指向的节点，那么没有指向的就是 root 了。

2.3 题目三

HW3 树和二叉树 > 二叉树的非递归遍历
二叉树的非递归遍历
描述
二叉树的非递归遍历可通过栈来实现。例如对于由 abc##d##ef###先序建立的二叉树，如下图 1 所示，中序非递归遍历（参照课本 p131 算法 6.3）可以通过如下一系列栈的入栈出栈操作来完成：push(a) push(b) push(c) pop pop push(d)pop pop push(e) push(f) pop pop。

图 1

如果已知中序遍历的栈的操作序列，就可唯一地确定一棵二叉树。请编程输出该二叉树的后

序遍历序列。

提示:本题有多种解法,仔细分析二叉树非递归遍历过程中栈的操作规律与遍历序列的关系,可将二叉树构造出来。

输入
第一行一个整数 n，表示二叉树的结点个数。
接下来 2n 行，每行描述一个栈操作，格式为：push X 表示将结点 X 压入栈中，pop 表示从栈中弹出一个结点。
(X 用一个字符表示)
对于 20%的数据，0<n<=10
对于 40%的数据，0<n<=20
对于 100%的数据，0<n<=83
从右上角下载并运行 p37_data.cpp 以生成随机测试数据
输出
一行，后序遍历序列。

```
typedef int Status;
typedef char TElemType;
typedef struct BiTnode {
    TElemType data;
    BiTnode* lchild;
    BiTnode* rchild;
}BiTnode, * BiTree;

typedef BiTree SElemType;

struct Stack {
    SElemType* base;
    SElemType* top;
    int stacksize;
};

//入栈

Status Push(Stack& S, BiTree T)
{
    char e;
    cin >> e;

    T->data = e;

    if (S.top - S.base >= S.stacksize)
    {
        cout << "Stack is Full" << endl;
```

```
        return 0;
    }
    *S.top++ = T;
    return 0;
}

//出栈

Status Pop(Stack& S, BiTree& T)
{
    if (S.top == S.base)
    {
        cout << "Stack is Empty" << endl;
        return 0;
    }
    --S.top;
    T = *S.top;
    return 0;
}

//中序建立二叉树
Status InOrderCreate(BiTree T, Stack& S, int node_num)
{
    char cmd[10];
    int flag = 0;
    int count = 0;
    int srmp = 0;
    for (int i = 0; i < 2 * node_num; i++)
    {
        cin >> cmd;
        if (!strcmp(cmd, "push"))
        {
            if (count == 0)
            {
                Push(S, T);
                count = 1;
            }
            else
            {
                BiTnode* p = (BiTnode*)malloc(sizeof(BiTnode));

                p->lchild = NULL;
                p->rchild = NULL;
                Push(S, p);
```

```
                if (flag == 1)
                {
                    T->rchild = p;
                    flag = 0;
                    srmp++;
                }
                else
                    T->lchild = p;
                T = p;
            }
        }
        else
            if (!strcmp(cmd, "pop"))
            {
                Pop(S, T);
                flag = 1;
            }
    }
    return 0;
}

// 后序遍历二叉树
void PostOrderTraverse(BiTree T)
{
    if (T != NULL)
    {
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        cout << T->data;
    }
}
```

```
6
push a
push b
push c
pop
pop
push d
pop
pop
push e
push f
pop
pop
cdbfea
```

非递归实现后序遍历【简单的方法】

非递归实现后序遍历还是很复杂的，变来变去，指来指去的，但是，我发现一种简单的方法，

我们知道 后序遍历的顺序为:，后序：左->右->根，那么可以把后序当作：根->右->左，最后再反转回来即可。由于我们是使用的栈，所以先保存根节点，然后先放进去左节点，再放进去右节点。
前序遍历按照“根结点-左孩子-右孩子”的顺序进行访问。
具体算法：先遍历左孩子，并输出。当左孩子遍历完后，取栈顶，找右孩子。此时循环还没有结束，再遍历它的左孩子，右孩子直至孩子全部遍历结束。
后序遍历按照“左孩子-右孩子-根节点”的顺序进行访问。
具体算法：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还为被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量标识该结点是否是第一次出现在栈顶。

2.4 题目四

HW3 树和二叉树 > 中序遍历线索二叉树
中序遍历线索二叉树
描述
练习线索二叉树的基本操作，包括二叉树的线索化，中序遍历线索二叉树，查找某元素在中序遍历的后继结点、前驱结点。

输入
2 行
第 1 行，输入先序序列，内部结点用一个字符表示，空结点用#表示
第 2 行，输入一个字符 a，查找该字符的后继结点元素和前驱结点元素
输出
第 1 行，输出中序遍历序列
第 2 行，输出查找字符的结果，若不存在，输出一行 Not found，结束
接下来包含 N 组共 2N 行（N 为树种与 a 相等的所有节点个数）
每一组按其代表节点的中序遍历顺序排列（即按中序遍历搜索线索树，找到节点即可输出，直到遍历完整棵树）。
第 2k+3 行输出该字符的直接后继元素及该后继元素的 RTag，格式为：succ is 字符+RTag
第 2k+4 行输出该字符的直接前驱继元素及前驱元素的 LTag，格式为：prev is 字符+LTag
若无后继或前驱，用 NULL 表示。上述 k 有（0<=k<N/2）

```
typedef char TElemType;

typedef int Status;

typedef struct BiTnode {
    TElemType data;
    BiTnode* lchild;
    BiTnode* rchild;
    bool RTag, LTag;
}BiTnode, * BiTree;
```

```
Status PreOrderTraverse(BiTree& T)
//先序建立二叉树
{
    static int i = 0;

    if (sTree[i] == '\0')
    {
        T = NULL;
        return 0;
    }
    if (i == 0)
    {
        T->data = sTree[i];
        ++i;
    }
    else
        if (sTree[i] != '#')
        {
            BiTnode* p = (BiTnode*)malloc(sizeof(BiTnode));
            T = p;
            p->data = sTree[i];
            p->lchild = NULL;
            p->rchild = NULL;
            ++i;
        }
        else
        {
            T = NULL;
            i++;
            return 0;
        }

    PreOrderTraverse(T->lchild);
    PreOrderTraverse(T->rchild);

    return 0;
}

Status InOrderTraverse(BiTree T)//中序遍历二叉树

{
    if (T != NULL)
    {
```

```
        InOrderTraverse(T->lchild);
        cout << T->data;
        InOrderTraverse(T->rchild);
    }

    return 0;
}

void InThring(BiTree T)//中序线索化

{
    static BiTnode* pre = NULL;
    static int flag = 0;
    if (T != NULL)
    {
        InThring(T->lchild);
        if (pre != NULL && flag == 1)
            pre->rchild = T;
        if (T->lchild == NULL)
        {
            T->LTag = 1;
            T->lchild = pre;
        }
        else
            T->LTag = 0;

        if (T->rchild == NULL)
        {
            flag = 1;
            T->RTag = 1;
        }
        else
        {
            flag = 0;
            T->RTag = 0;
        }

        pre = T;

        InThring(T->rchild);
    }

}
```

```
void InThrSearch(BiTree T, char e)//中序线索化查找

{
    int flag = 0;
    int search = 0;
    bool found = false;
    BiTnode* prev = NULL;
    BiTnode* succ = NULL;
    BiTnode* prev_1 = NULL;
    while (T != NULL)
    {
        if (search == 0)
            while (T->LTag != 1)
                T = T->lchild;
        if (flag == 1)
        {
            succ = T;
            flag = 0;
            if (succ != NULL)
                cout << "succ is " << succ->data << succ->RTag << endl;
            else
                cout << "succ is " << "NULL" << endl;
            if (prev != NULL)
                cout << "prev is " << prev->data << prev->LTag << endl;
            else
                cout << "prev is " << "NULL" << endl;
            prev = prev_1;
        }
        if (T->data == e)
        {
            flag = 1;
            found = true;
            prev_1 = T;
        }
        else
            prev = T;
        if (T->RTag == 1)
            search = 1;
        else
            search = 0;
        T = T->rchild;
    }
    if (found == false)
        cout << "Not found" << endl;
```

```
    if (flag == 1)
    {
        succ = T;
        flag = 0;
        if (succ != NULL)
            cout << "succ is " << succ->data << succ->RTag << endl;
        else
            cout << "succ is " << "NULL" << endl;
        if (prev != NULL)
            cout << "prev is " << prev->data << prev->LTag << endl;
        else
            cout << "prev is " << "NULL" << endl;
    }
}
```

```
abc##d##ef###
cbdafe
e
succ is NULL
prev is f1
```

第一步创建二叉树结构:
根据输入一个广义表来创建二叉树（A（B(D,E(H)), C(F(, I),G))）,使用二叉树的链式表示,左右孩子表示法。用一个栈来存“（"后面的字母的指针，遇见“)"回退 top–，再等于向 top-1 的指。遍历广义表字符串，令 pre 指向树的头结点，开始遍历，遇见“（"或“，",往前走一步 i++；遇见每个字母,申请一块内存 p,判断 i-1 是否为“（",是则 pre->lchild =p，p 入栈；若为“，"，则 pre->rchild = p;
第二步按照中序进行遍历，线索化：左子树线索化：找到最左下端结点，访问：右子树线索化，第三步按照前驱后继进行访问：

2.5 题目五

HW3 树和二叉树 > 家族树
问题描述
人类学教授对居住在孤岛上的人们及其历史感兴趣。他收集了他们的家谱进行一些人类学实验。为了实验，他需要用电脑来处理家谱。为此，他把它们翻译成文本文件。以下是表示家谱的文本文件的示例。

```
John
 Robert
  Frank
  Andrew
 Nancy
  David
```

每行包含一个人的名字。第一行的名字是这个家谱中最古老的祖先。家谱只包含该祖先的后代。他

们的丈夫或妻子没有在家庭树上显示。一个人的孩子比父母缩进一个空格。例如，Robert和Nancy是John的孩子，Frank和Andrew是Robert的孩子。David比Robert缩进了一个空格，但他不是Robert的孩子，而是Nancy的孩子。为了用这一方法表示一个家谱，教授将一些人从家谱中排除，使得家谱中没有人有两个父母。
为了实验，教授还收集了家属的文件，并提取了每个家谱中关于两人关系的陈述语句。以下是关于上述家庭陈述语句的一些例子。
John is the parent of Robert.
Robert is a sibling of Nancy.
David is a descendant of Robert.
对于实验，他需要判断每个陈述语句是否正确。例如，上面的前两个语句是正确的，最后一个语句是错误的。但是这个任务是十分无聊的，他想通过电脑程序来判断。
需要支持的查询有以下几种：
X is a child of Y.
X is the parent of Y.
X is a sibling of Y.
X is a descendant of Y.
X is an ancestor of Y.

注意：一个人的祖先、后代、兄弟可以是自己，但父母、孩子则不行。
输入
输入包含若干组测试用例，每个测试用例由一个家谱和一个陈述句集合组成。
每个测试用例的第一行给出两个整数n,m,(0<n,m<1000)，分别表示家谱中的名字和陈述语句的数目。
接下来输入的每行少于70个字符。名字的字符串仅由不超过20个字母字符组成。在家谱的第一行中给出名字前没有前导空格，在家谱中其他的名字至少缩进一个空格，表示是第一行给出的那个名字（家族的最早祖先）的后代。若家谱中一个名字缩进k个空格，则下一行最多缩进k+1个空格。
本题假定，除了最早的祖先外，在家谱中，每个人都有其父母。在同一个家谱中同样的名字不会出现两次。家谱的每一行结束的时候无多余的空格。
再接下来每句陈述句占一行。在家谱中没有出现的名字不会出现在陈述句中。陈述句中连续的单词被一个空格分开。每句陈述句在行首和行尾没有多余的空格。
用两个0表示输入的结束。
对于20%的数据，有0<n<=20, 0<m<=40，且只有1个测试集
对于40%的数据，有0<=n<=100, 0<=m<=200，且超过一个数据集
对于100%的数据，有0<=n<=5000, 0<=m<=10000，且超过一个数据集
本题拿满分需要使用哈希表，可以使用std::map
输出
对于测试用例中的每句陈述语句，程序输出一行，给出True或False。
每个测试用例后要给出一个空行，就算只有一个测试用例。

```
struct tree
{
    int level;
    int parent;
}*family[10000] = { NULL };
```

```
void judge_child(int n1, int n2)//判断孩子
{
    if (family[n1]->parent == n2)
        cout << "True"<<endl;
    else
        cout << "False"<<endl;
}

void judge_parent(int n1, int n2)//判断双亲
{
    if (family[n2]->parent == n1)
        cout << "True"<<endl;
    else
        cout << "False"<<endl;
}

void judge_sibling(int n1, int n2)//判断兄弟
{
    if (family[n1]->parent == family[n2]->parent)
        cout << "True"<<endl;
    else
        cout << "False"<<endl;
}

void judge_descendant(int n1, int n2)//判断子孙
{
    int ance = n1;
    for (int i = 0; i < family[n1]->level - family[n2]->level; i++)
    {
        ance = family[ance]->parent;
    }
    if (ance == n2)
        cout << "True"<<endl;
    else
        cout << "False"<<endl;
}

void judge_ancestor(int n1, int n2)//判断祖先
{
    int ance = n2;
    for (int i = 0; i < family[n2]->level - family[n1]->level; i++)
    {
        ance = family[ance]->parent;
```

```
    }
    if (ance == n1)
        cout << "True"<<endl;
    else
        cout << "False"<<endl;
}
```

```
6 5
John
 Robert
  Frank
  Andrew
 Nancy
  David
Robert is a child of John.
True
Robert is an ancestor of Andrew.
True
Robert is a sibling of Nancy.
True
Nancy is the parent of Frank.
False
John is a descendant of Andrew.
False
2 1
abc
 xyz
xyz is a child of abc.
True
0 0
```

程序的关键是掌握二叉树的相关操作、二叉树的创建和运用、结点的查找、祖宗结点的查找等。在编程的过程中，出现了很多问题，如二叉树无法建立、程序内存读取不了、忘了添加头文件等错误。在单步调试和添加提示输出的情况下修改程序运行正确。查找首先要判断该结点是否为空，再与查找到的结点比较，否则会内存无法读取，强行结束程序。祖宗结点的查找一直是个大问题，在参考书的帮助下想到了后序遍历，是可以从孩子往上找到。家谱的功能是查询家族每个人的信息，并且输出它们的信息，还要具有查询输出功能。这样复习了一下查询、插入、删除函数的应用。并学习了 i/o 流的文件储存及调用功能，复习了子函数和递归调用功能，并熟练运用这些函数。

3. 实验总结

节点深度：对任意节点 x，x 节点的深度表示为根节点到 x 节点的路径长度。所以根节点深度为 0，第二层节点深度为 1，以此类推
节点高度：对任意节点 x，叶子节点到 x 节点的路径长度就是节点 x 的高度
树的深度：一棵树中节点的最大深度就是树的深度，也称为高度
父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点
子节点：一个节点含有的子树的根节点称为该节点的子节点
节点的层次：从根节点开始，根节点为第一层，根的子节点为第二层，以此类推

兄弟节点：拥有共同父节点的节点互称为兄弟节点
度：节点的子树数目就是节点的度
叶子节点：度为零的节点就是叶子节点
祖先：对任意节点 x，从根节点到节点 x 的所有节点都是 x 的祖先（节点 x 也是自己的祖先）
后代：对任意节点 x，从节点 x 到叶子节点的所有节点都是 x 的后代（节点 x 也是自己的后代）
森林：m 颗互不相交的树构成的集合就是森林
该树想要达成有效查找，势必需要维持如下一种结构：

树的子叶节点中，左子树一定小于等于当前节点，而当前节点的右子树则一定大于当前节点。只有这样，才能够维持全局有序，才能够进行查询。

这也就决定了只有取得某一个子叶节点后，才能够根据这个节点知道他的子树的具体的值情况。这点非常之重要，因为二叉平衡树，只有两个子叶节点，所以如果想找到某个数据，他必须重复更多次“拿到一个节点的两个子节点，判断大小，再从其中一个子节点取出他的两个子节点，判断大小。”这一过程。

这个过程重复的次数，就是**树的高度**。那么既然每个子树只有两个节点，那么 N 个数据的树的高度也就很容易可以算出了。
树的存储结构：
顺序存储结构：类似数组形式（查询快，插入慢）
链式存储结构：类似链表形式（查询的时候需要大量的寻道时间、
优点：没有空间浪费，不会存在空余的空间，而且查询速度比较快。
缺点：需要取出多个节点，且无法预测下一个节点的位置。这种取出的操作，在内存内进行的时候，速度很快，但如果到磁盘，那么就意味着大量随机寻道，基本磁盘就被查死了。