# 上机实验3：树和二叉树

### 班级：2022211305 学号：2022211119 姓名：赵宇鹏

### **Email:zhaoyupeng@bupt.edu.cn**

1、**实验内容**：完成了上机作业二叉树要求作业：

（1）输入两个字符串，分别为二叉树的层序序列和中序序列构造出二叉树t

（2）打印二叉树的前序、中序、后序和层序序列，以凹入表缩进方式打印二叉

树【printf("%\*s", level \* 4, "");】

（3）求二叉树t的高度

（4）验证二叉树t是否具备这样的性质：任何一个子树中的结点集合，树根结

点的值总是最小。（按字母序）

（5）拷贝二叉树t为新的二叉树t1

（6）将二叉树t1左右子树交换，然后对t1执行（2）

（7）剪除t中值为E的节点以及其全部子孙，然后对执行（2）

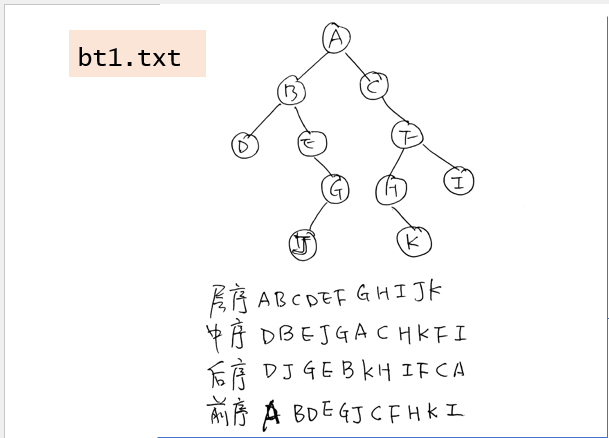
（8）销毁二叉树t和t1

无选做部分。

1. **算法与源程序结构：**

创建了BiTNode结构体，有level\_in函数用来通过层序遍历和中序遍历生成二叉树，pre\_order, in\_order, later\_order, lever\_order以及printTree函数分别用来打印先序遍历，中序遍历，后序遍历，层序遍历以及凹入缩进二叉树，Print\_all函数用来将上述打印函数封装以便使用，height函数用来返回二叉树的深度，vertify函数用来检验：任何一个子树中的结点集合，树根结点的值总是最小，若是，则返回1，不是则返回0。copy函数用来复制生成的二叉树，swap函数用来交换二叉树的左右子树，delete函数将需要删除的值删除后返回处理后的二叉树，destory函数用来销毁二叉树。

思路：假设我们输入的是ppt中给出的bt1.txt数据，



层序为ABCDEFGHIJK

中序为DBEJGACHKFI

由于中序遍历的顺序是左中右，所以中序遍历排列顺序一定都是按照左中右的顺序来排列的，而层序遍历按照层级顺序进行遍历，我们以层序遍历的元素为根，在中序遍历中查找它的位置，在它左边的结点就是它的左子树所有的结点，在它右边的则为右子树的结点，层序遍历后面的结点所查找到的左右子树结点不能跨过前面的结点，因为前面的是该结点的母树。首先查找A，则DBEJG都为A的左子树，CHKFI都为A的右子树，再查找B，则D为B的左子树，EJG为B的右子树，对C，没有左子树，HKFI为其右子树，对D，无左右子树，依此类推进行递归。可以使用s1, e1; s2, e2两组参数来分别定位中序字符数组的遍历空间，以防遍历到母树，发生错误。

打印层序遍历序列时，利用上述类似思想，创建了一个数组，定义了front和rear变量来模拟队列操作，若某结点有左右子树，则储存完该结点后按顺序储存其左右子树。

其余函数皆可利用顺序遍历，中序遍历以及后序遍历的递归思想来实现。

1. **实验总结：**

完成该作业大概花费两小时，在完成过程中最大的问题就是思考出如何把层序遍历和中序遍历结合起来生成二叉树，后经过长时间思考加多次尝试才发现其中规律。在编写递归程序的过程中，发现不好把控递归范围，于是加入了s1, e1这样的数据来进行限制。

1. 代码及输出结果

/\*（1）输入两个字符串，分别为二叉树的层序序列和中序序列构造出二叉树t

（2）打印二叉树的前序、中序、后序和层序序列，以凹入表缩进方式打印二叉树【printf("%\*s", level \* 4, "");】

（3）求二叉树t的高度

（4）验证二叉树t是否具备这样的性质：任何一个子树中的结点集合，树根结点的值总是最小。（按字母序）

（5）拷贝二叉树t为新的二叉树t1

（6）将二叉树t1左右子树交换，然后对t1执行（2）

（7）剪除t中值为E的节点以及其全部子孙，然后对执行（2）

（8）销毁二叉树t和t1\*/

/\*1可用递归实现，以层序序列为参考，每次递归时，在中序序列中找到相应根节点，然后递归左右子树，

若有左结点则该节点有左子树，

\*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

typedef struct BiTNode

{

    char data;

    struct BiTNode \*lchild, \*rchild;

}Tree;

Tree \*level\_in(char \*level, int s1, int e1, char \*in, int s2, int e2)

{

    if(s1 > e1 || s2 > e2)

        return NULL;

    Tree \*T = (Tree \*)malloc(sizeof(Tree));

    int i, j;

    for(i = s1; i <= e1; i++)

    {

        for(j = s2; j <= e2; j++)

        {

            if(level[i] == in[j])

                break;

        }

        if(j <= e2)

            break;

    }//找到每次递归的根节点

    T->data = level[i];

    T->lchild = level\_in(level, s1 + 1, e1, in, s2, j - 1);//j-1结束代表递归左子树

    T->rchild = level\_in(level, s1 + 1, e1, in, j + 1, e2);//j+1开始代表递归右子树

    return T;

}

void pre\_order(Tree \*T)

{

    if(T)

    {

        printf("%c ", T->data);

        pre\_order(T->lchild);

        pre\_order(T->rchild);

    }

}

void in\_order(Tree \*T)

{

    if(T)

    {

        in\_order(T->lchild);

        printf("%c ", T->data);

        in\_order(T->rchild);

    }

}

void later\_order(Tree \*T)

{

    if(T)

    {

        later\_order(T->lchild);

        later\_order(T->rchild);

        printf("%c ", T->data);

    }

}

void level\_order(Tree \*T)

{

    Tree \*queue[100];

    int front = 0, rear = 0;

    queue[rear] = T;

    rear++;

    while(front < rear)

    {

        Tree \*p = queue[front];

        front++;

        printf("%c ", p->data);

        if(p->lchild)

            queue[rear++] = p->lchild;

        if(p->rchild)

            queue[rear++] = p->rchild;

    }

}

void printTree(Tree \*T, int level)

{

    if(T == NULL)

        return;

    printf("%\*s%c\n", level \* 4, "", T->data);

    printTree(T->lchild, level + 1);

    printTree(T->rchild, level + 1);

}

void Print\_all(Tree\*T){

    pre\_order(T);

    printf("\n");

    in\_order(T);

    printf("\n");

    later\_order(T);

    printf("\n");

    level\_order(T);

    printf("\n");

    printTree(T, 0);

}

int height(Tree \*T)

{

    if(T == NULL)

        return 0;

    int l = height(T->lchild);

    int r = height(T->rchild);

    if(l > r)

        return l + 1;

    else

        return r + 1;

}

int vertify(Tree \*T)

{

    if(T == NULL)

        return 1;

    if(T->lchild && T->lchild->data < T->data)

        return 0;

    if(T->rchild && T->rchild->data < T->data)

        return 0;

    return vertify(T->lchild) && vertify(T->rchild);

}

Tree \*copy(Tree \*T)

{

    if(T == NULL)

        return NULL;

    Tree \*t1 = (Tree \*)malloc(sizeof(Tree));

    t1->data = T->data;

    t1->lchild = copy(T->lchild);

    t1->rchild = copy(T->rchild);

    return t1;

}

void swap(Tree \*T)

{

    if(T == NULL)

        return;

    Tree \*p;

    p = T->lchild;

    T->lchild = T->rchild;

    T->rchild = p;

    swap(T->lchild);

    swap(T->rchild);

}

Tree \*delete(Tree \*T, char e)

{

    if(T == NULL)

        return NULL;

    T->lchild = delete(T->lchild, e);

    T->rchild = delete(T->rchild, e);

    if(T->data == e)

    {

        free(T);

        return NULL;

    }

    return T;

}

void destroy(Tree \*T)

{

    if(T == NULL)

        return;

    destroy(T->lchild);

    destroy(T->rchild);

    free(T);

}

int main()

{

    char level[100], in[100];

    scanf("%s", level);

    scanf("%s", in);

    int len1 = strlen(level);

    int len2 = strlen(in);

    int i;

    if(len1 == len2){

        for(i = 0; i < len1; i++){

            if(level[i] < 'A' || level[i] > 'Z'|| in[i] < 'A' || in[i] > 'Z')

            {

                printf("输入错误\n");

                return 0;

            }

        }

        Tree \*T = level\_in(level, 0, len1 - 1, in, 0, len2 - 1);

        Print\_all(T);

        printf("%d\n", height(T));

        if(vertify(T))

            printf("Yes\n");

        else

            printf("No\n");

        Tree \*t1 = copy(T);

        swap(t1);

        Print\_all(t1);

        t1 = delete(t1, 'E');

        Print\_all(t1);

        destroy(T);

        destroy(t1);

        return 0;

    }else{

        printf("输入错误\n");

        return 0;

    }

}

输出结果：

