

# 《数据结构》上机报告

2018 年 10 月 26 日

姓名：刘思源 学号：1651390 班级：电子三班 得分：\_\_\_\_\_

实验题目	树		
问题描述	<p>树状图是一种数据结构，它是由 <math>n</math> (<math>n \geq 1</math>) 个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：</p> <p>每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树</p>		
基本要求	<p>1. (p1) 二叉树是 <math>n</math> (<math>n \geq 0</math>) 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，包括先序遍历建立二叉树、先序遍历、中序遍历、后序遍历、层次遍历、输出二叉树。</p> <p>2. (p2) 本题练习用递归程序对二叉树进行计算，包括二叉树的叶子结点数，总结点数，高度，复制二叉树（左右互换）。</p> <p>3. (p3) 本题练习用非递归完成二叉树的中序遍历。</p>		
	<table> <tr> <td>已完成基本内容（序号）：</td><td>1, 2, 3</td></tr> </table>	已完成基本内容（序号）：	1, 2, 3
已完成基本内容（序号）：	1, 2, 3		
选做要求	<table> <tr> <td>已完成选做内容（序号）</td><td></td></tr> </table>	已完成选做内容（序号）	
已完成选做内容（序号）			
数据结构设计	<p>二叉树：</p> <pre>typedef struct Node {     ElemType data;     struct Node *lchild, *rchild; } *BiTree, BiTNode;</pre> <p>遍历树非递归算法采用的栈结构：</p> <pre>typedef BiTNode * StackElemType;</pre> <pre>typedef struct stacknode {     StackElemType data;</pre>		

	<pre> stacknode * next; }StackNode; </pre>
功能(函数)说明	<p>1、 二叉树的初始化</p> <p>采用递归算法。先序输入的顺序是：根节点——左子树——右子树。设定一个中间变量 ch，持续输入，如果 ch 是空符号#，就将该结点置为 NULL。否则，申请一个新的树节点，将该输入的值赋给新的结点的数据域，同时创造该结点的左子节点和右子节点。进入下一步的递归。因为是先序输入，所以可以直接将根节点赋值，接下来再对左节点递归直到结束，再对右节点递归直到结束。</p> <pre> void CreateBiTree(BiTree &amp;T) {     ElemType ch;     cin &gt;&gt; ch;     if (ch == '#') {         T = NULL;     }     else {         T = new BiTNode;         T-&gt;data = ch;         CreateBiTree(T-&gt;lchild);         CreateBiTree(T-&gt;rchild);     }     return; } </pre> <p>2、 先序递归遍历</p> <p>先序遍历的顺序是：根节点——左子树——右子树。开始遍历时，如果树非空，就访问根节点，同时输出根节点的值，进入根节点的左子树进行遍历，直到左子树下的每一个结点都遍历完成，一步一步回退，开始遍历右子树，直到右子树都遍历完成。一步一步回退直到根节点，遍历完成。</p> <pre> void PreOrderTraverse(BiTree T) {     if (T){         cout &lt;&lt; T-&gt;data;         PreOrderTraverse(T-&gt;lchild);     } } </pre>

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

```
}
```

```
}
```

### 3、中序递归遍历

中序遍历的顺序是：左子树——根节点——右子树。开始遍历时，如果树非空，就进入根节点的左子树进行遍历，直到左子树下的每一个结点都遍历完成，都某一个节点不存在左子树时，输出该节点的值，一步回退，同时输出上一级根节点的值。开始遍历右子树，直到右子树都遍历完成。一步一步回退直到根节点，输出根节点的值，开始遍历根节点的右子树，重复上述的步骤，遍历完成。

```
void InOrderTraverse(BiTree T)
```

```
{
```

```
    if (T){
```

```
        InOrderTraverse(T->lchild);
```

```
        cout << T->data;
```

```
        InOrderTraverse(T->rchild);
```

```
    }
```

```
}
```

### 4、后序递归遍历

后序遍历的顺序是：左子树——右子树——根节点。开始遍历时，如果树非空，就进入根节点的右子树进行遍历，直到左子树下的每一个结点都遍历完成，都某一个节点不存在左子树时，一步回退，开始遍历右子树，直到右子树都遍历完成。一步回退直到根节点，输出根节点的值重复上述的步骤，遍历完成。

```
void PostOrderTraverse(BiTree T)
```

```
{
```

```
    if (T){
```

```
        PostOrderTraverse(T->lchild);
```

```
        PostOrderTraverse(T->rchild);
```

```
        cout << T->data;
```

```
    }
```

```
}
```

### 5、中序递归遍历的非递归算法

递归函数的本质就是栈。所以我们可以使用栈的数据结构来模拟递归函数的运行方式。首先我们要定义一个新的结构SqStack，它的数据类型是BiTNode\*。用来存储每一个树的节点。接下来开始遍历，我们定义一个指向树根节点的指针p。如果p非空，即树非空，或者栈非空的时候，我们进入遍历的循环。如果p非空，就将p的节点压入栈内，同时输出提示语句。将p指向自己的左节点。否则，如果p这个时候是空的，则证明上一个节点没有左子树，就将栈顶的节点Pop出去，

输出提示语句（这个时候输出的是根节点的值）。再将p指向它的右子树。  
直到栈为空的时候，就完成了中序遍历。

```
void InOrderTraverse2(BiTree T)
{
    StackNode * S;
    BiTNode * p;
    S = NULL;
    p = T;
    S = InitStack(S);

    if (NULL == p){
        return;
    }

    while (p || !StackEmpty(S)){
        if (p){
            StackPush(S, p);
            cout << "push " << p->data << endl;
            p = p->lchild;
        }
        else
        {
            StackPop(S, p);
            cout << "pop" << endl;
            cout << p->data << endl;
            p = p->rchild;
        }
    }
    free(S);
}
```

## 6、先序遍历的非递归算法

先序的非递归算法和中序的十分接近，唯一不同的是输出的位置不一样。由于题目中并未出现这个算法的要求，该算法就没有用中序算法的格式进行输出，只完成遍历输出的部分。

可见，先序遍历的顺序是根节点——左子树——右子树。所以将根节点入栈的同时就进行输出，Pop的时候不进行输出。

```
void PreTraverseTree2(BiTree T)
{
    StackNode *S;
    BiTNode * p;
    S = NULL;
    p = T;
    S = InitStack(S);
```

```

if (NULL == p){
    return;
}
while (p || !StackEmpty(S)){
    if (p){
        StackPush(S, p);
        cout << p->data << endl;
        p = p->lchild;
    }
    else{
        StackPop(S, p);
        p = p->rchild;
    }
}
free(S);
}

```

#### 7、后序遍历的非递归算法

后序遍历的非递归算法会稍显复杂。首先我们定义两个指针 pre, cur。一个指向当前访问的树节点，另一个指向上一个访问的树节点。首先将树的根节点压入栈中，当栈非空时，进入循环。每次都将 cur 置为 NULL，再将当前栈顶的节点赋给 cur。如果 cur 是一个叶节点，没有左子树和右子树的话；或者 pre 节点有值，并且是 cur 的左子节点或者右子节点的话，就输出当前 cur 的值，把 cur 的节点赋给 pre，再把 cur 节点 Pop 出去。

如果不满足上述情况，意味着该结点下还有左子树或者右子树，那么就先把右子节点压进栈（如果存在的话），再把左子节点压进栈。这样顺序的目的在于每次要先访问左子节点，栈的特点是 LIFO，所以后压进左子节点，先访问左子节点。

```

void LastTraverseTree2(BiTree T)
{
    StackNode * S;
    BiTNode * cur, *pre;
    S = NULL;
    S = InitStack(S);
    if (NULL == T){
        return;
    }
    pre = NULL;
    cur = NULL;
    StackPush(S, T);
    while (!StackEmpty(S)){
        cur = NULL;
        StackGetTop(S, cur);
        if ((cur->lchild == NULL && cur->rchild == NULL) || (pre != NULL && (pre ==
cur->lchild

```

```

        || pre == cur->rchild))){
    cout<<cur->data<<endl;
    pre = cur;
    StackPop(S, cur);
}
else{
    if (cur->rchild != NULL){
        StackPush(S, cur->rchild);

    }

    if (cur->lchild != NULL){
        StackPush(S, cur->lchild);
    }

}

}
free(S);
}

```

## 8、层次遍历

层次遍历是根据树的每一层进行遍历。我们用指针数组模拟一个循环队列。首先将根节点入队，用 rear 指向我们要入队判断的节点，用 front 来指向要输出的节点。如果 front 不等于 rear，就将 front 后移一位，将当前 front 指向的节点赋给 q。输出 q 的值，如果 q 的左子树不为空，就将左子节点压入队列，同样的将右子节点入队列。逐个输出完成遍历。

```

void LevelOrderTraverse(BiTree T)
{
    int front, rear;
    BiTNode *que[MAXSIZE];
    front = rear = 0;
    BiTNode *q;
    if (T){
        rear = (rear + 1) % MAXSIZE;
        que[rear] = T;
        while (front != rear){
            front = (front + 1) % MAXSIZE;
            q = que[front];
            cout << q->data;
            if (q->lchild != 0){
                rear = (rear + 1) % MAXSIZE;
                que[rear] = q->lchild;
            }
            if (q->rchild != 0){
                rear = (rear + 1) % MAXSIZE;
                que[rear] = q->rchild;
            }
        }
    }
}

```

```

    }
}
cout << endl;
return;
}

```

## 9、 树形打印

树形打印要求是树形逆时针旋转 90 度后完成打印。如果树本身是空的，就返回。使用一个 level 的整型变量记录树的层数。Level 初始为 0，使用递归算法，先对根节点的右子树进行递归。直到没有右子树，打印 level\*5 个空格，在打印当前节点的值。返回，对上一级节点的左子树进行递归。

```

void TreePrint(BiTree T,int level)
{
    int i;
    if (!T){
        return;
    }
    TreePrint(T->rchild, level + 1);
    for (i = 0; i<level; i++){
        cout<<"    ";
    }
    cout<<T->data<<endl;
    TreePrint(T->lchild, level + 1);
}

```

## 10、 求树的高度

仍然使用递归算法。使用两个整型变量 m, n 记录左右子树的高度。对每一个节点而言，遍历它下面左右子树的高度，求得较大的值+1 返回给上一层。

```

int Depth(BiTree T)
{
    if (T == NULL)
        return 0;
    else {
        int m = Depth(T->lchild);
        int n = Depth(T->rchild);
        if (m>n) return (m + 1);
        else return (n + 1);
    }
}

```

## 11、 求树节点的个数

如果树是空的，返回 0。否则，采用递归算法，将每个节点下面的左右节点数量相加。

```

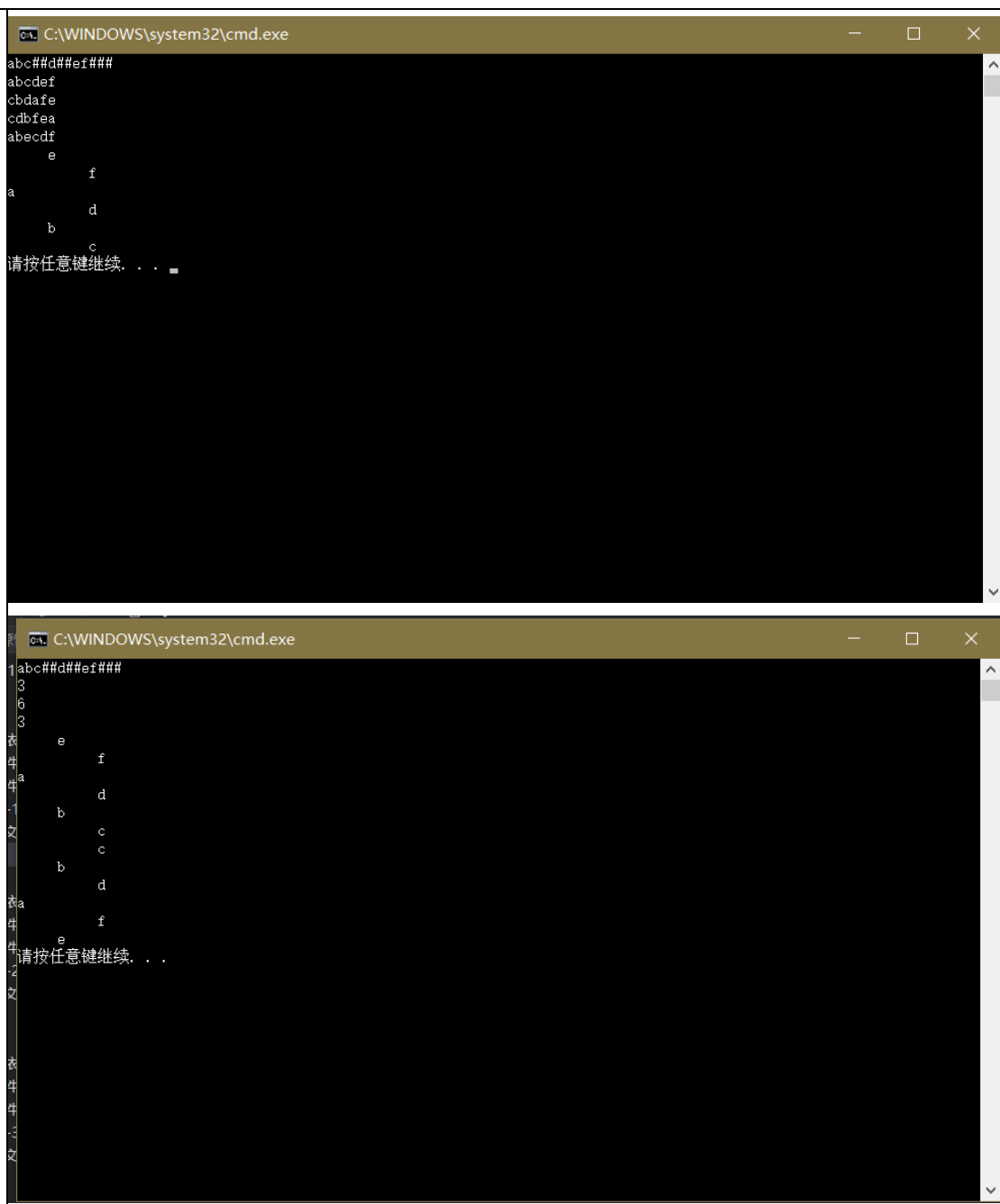
int NodeCount(BiTree T)
{
    if (T == NULL) {
        return 0;
    }
}

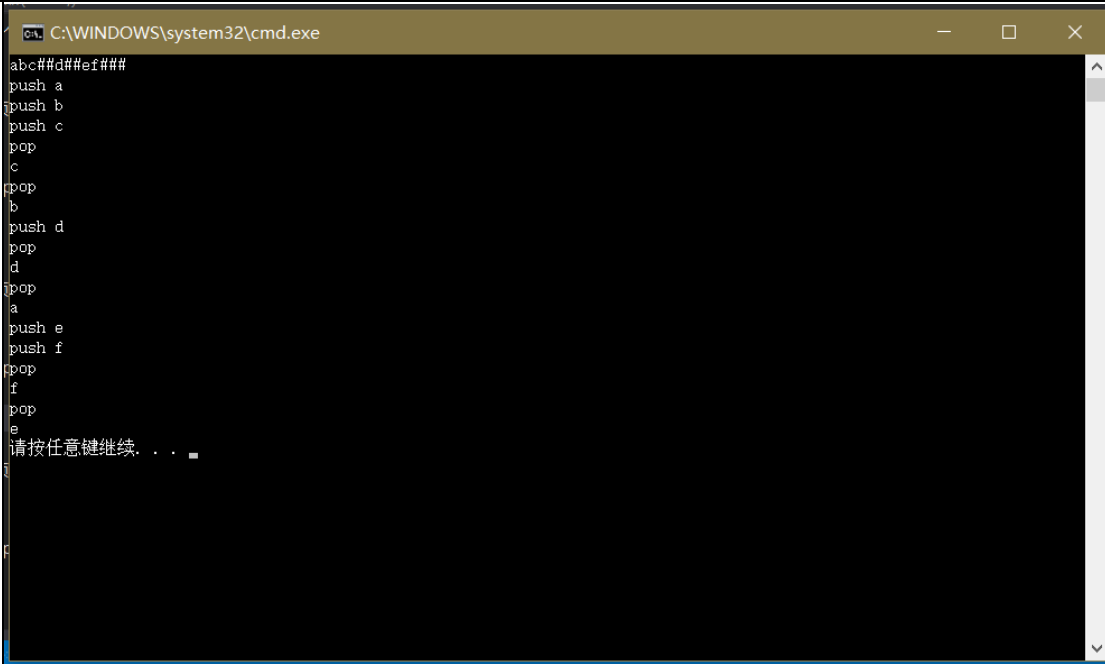
```

	<pre>     }     else {         return NodeCount(T-&gt;lchild) + NodeCount(T-&gt;rchild) + 1;     } } </pre> <p>12、 求叶节点的数量</p> <p>如果树是空的，返回 0。如果树的根节点下面没有左右节点，就返回 1。否则，对每个节点使用递归，记录没有下属节点的节点的个数并相加。</p> <pre> int LeafCount(BiTree T) {     if (!T) {         return 0;     }     if (!T-&gt;lchild &amp;&amp; !T-&gt;rchild) {         return 1;     }     else {         return LeafCount(T-&gt;lchild) + LeafCount(T-&gt;rchild);     } } </pre> <p>13、 树的复制（左右颠倒）</p> <p>形参是直接新的树传进。使用递归，将原树的左节点赋给新树的右节点。直到左子树都完成后，一步一步回退，对右子树进行复制。（如果题目不要左右颠倒，就将左子节点赋给左子节点，右子节点赋给右子节点）。</p> <pre> void Copy(BiTree T, BiTree &amp;NewT) {     if (T == NULL) {         NewT = NULL;         return;     }     else{         NewT = new BiTNode;         NewT-&gt;data = T-&gt;data;         Copy(T-&gt;lchild, NewT-&gt;rchild);         Copy(T-&gt;rchild, NewT-&gt;lchild);     } } </pre>
开发环境	Visual studio 2017



调试分析



	 <pre> C:\WINDOWS\system32\cmd.exe abc##d##ef### push a push b push c pop c pop b push d pop d pop a push e push f pop f pop e 请按任意键继续... </pre>
心得体会	<p>1、 树的递归使用</p> <p>涉及到树的各种算法，遍历，求值，一般都使用递归算法。因为对于每一个树都可以看成是很多个小的子树组成的。但递归函数可能会存在溢出的情况，所以当数据量过大的时候，可以考虑使用非递归的方法，直接使用栈的数据结构对算法进行优化。</p> <p>2、 关于树的存储栈</p> <p>使用非递归方法对树的节点继续存储时，注意定义好栈的数据类型。一般是树的节点指针类型。</p>