树

树的定义和基本术语

树的定义

树是n（n>=0）个结点的有限集，n=0时称为空树。在任意一棵非空树中：

1. 有且仅有一个特定的称为根root的结点；
2. 当n>1时，其余结点可分为m（m>0）个互不相交的有限集T1、T2、……、Tm，其中每一个集合本身又是一棵树，并且称为根的子树。

从逻辑结构上来看：

1. 树中只有根节点没有前驱结点；
2. 除跟外，其余节点有且仅有一个前趋；
3. 树中所有结点可以有零个或多个后继。
4. 除跟外的其它节点，都存在唯一一条从根到该节点的路径。

树的基本术语

1. 结点：数据元素+若干指向子树的分支
2. 结点的度：分支的个数；
3. 树的度：树的所有结点中最大的度数；
4. 叶结点：度为0的结点；
5. 分支结点：度大于0的结点；
6. 结点的层次：从根开始定义起，根为第一层，根的孩子为第二层，以此类推；
7. 树的深度：树中所有结点的最大层次；
8. 路径：从结点n1到nk的路径是一个结点序列n1,n2,……nk，ni是ni+1的前趋。
9. 森林：m（m>=0）棵互不相交的树的集合。
10. 有序树：树中结点的各子树从左至右是有次序的，不能互换的。
11. 无序树：树中结点的各子树从左至右是无次序的，可互换的。

二叉树

一棵二叉树是结点的一个有限集合，该集合：

1. 或者为空；
2. 或者是由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

二叉树的五种形态：

1. 空二叉树；
2. 只有一个根结点；
3. 根结点只有左子树；
4. 根结点只有右子树；
5. 根结点既有左子树又有右子树。

二叉树的应用

1. 用二叉树来表示算术表达式；

二叉树的性质

1. 在二叉树的第i层上至多有 2^{i-1} 个结点（i>=1）；
2. 深度为k的二叉树至多有 2^k-1 个结点（k>=1）；
3. 对任何一棵二叉树T，如果其终端结点(叶子结点)数为n0，度为2的结点数为n2，则n0=n2+1；

   证明： 结点总数：n0+n1+n2 分支总数：n1+2n2 结点总数=分支总数+1 即：n0+n1+n2=n1+2n2+1 即：n0=n2+1
4. 具有n个结点的完全二叉树的深度为 $log_2(n+1)$
5. 若对含n个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点有：
   1. 若i=1，则结点i是二叉树的根，无双亲, 否则，双亲是结点 $[i/2]$；
   2. 若2i>n，则结点i无左孩子， 否则，左孩子是结点 $2i$；
   3. 若2i+1>n，则结点i无右孩子， 否则，右孩子是结点 $2i+1$ 。

特殊二叉树

满二叉树

在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。即深度为k的满二叉树有 2^k-1 个结点。

完全二叉树

在一棵二叉树中，除了最后一层外，若其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干结点，则此二叉树为完全二叉树。即树中所含的n个结点和满二叉树中编号为1至n的结点一一对应。

二叉树的存储结构

顺序存储

对于一棵二叉树我们可以补全成完全二叉树，然后通过顺序存储。

二叉链表

二叉链表中每个结点包含三个域：数据域、左指针域和右指针域。

```
typedef struct BiTNode{
    int data;
    BiTNode *lchild, *rchild;
};
```

三叉链表

在二叉链表的基础上增加双亲指针域：

```
typedef struct TriTNode{
    int data;
    TriTNode *lchild, *rchild, *parent;
};
```

静态二叉链表

采用数组存储二叉树，每个结点包含三个域：数据域和两个序号域。

双亲链表

采用链表存储二叉树，每个结点包含三个域：数据域、一个双亲指针域和左右标记。

```
struct BiTNode{
    int data;
    int LRtag;
    BiTNode *parent;
};
```

建立二叉树

用字符串的形式建立二叉树

输入（在空子树处添加字符*的二叉树）的先序序列（设每一个元素是一个字符）。按先序遍历的顺序建立二叉链表。

```
typedef struct BiTNode {
    TElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

Status CreateBiTree(BiTtree &T) {

    scanf (&ch);

    if(ch==";") return;

    if(ch=="*") T=NULL;
    else {
        if (!(T=(BiTNode *)malloc(sizeof(BiTNode))))
            exit(OVERFLOW);
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return OK;

}//CreateBiTree
```

复制二叉链表

```
void CopyBiTree(BiTree T, BiTree &newT) {
    if (T == NULL) NewT = NULL;
    else {
```

```
        // 直接复制左子树并赋值给 newT->lchild
        newT->lchild = (BiTree)malloc(sizeof(BiTNode));
        newT->lchild->data = T->lchild->data;
        CopyBiTree(T->lchild->lchild, newT->lchild->lchild);
        CopyBiTree(T->lchild->rchild, newT->lchild->rchild);

        // 复制右子树
        newT->rchild = (BiTree)malloc(sizeof(BiTNode));
        newT->rchild->data = T->rchild->data;
        CopyBiTree(T->rchild->lchild, newT->rchild->lchild);
        CopyBiTree(T->rchild->rchild, newT->rchild->rchild);
    }
}
```

二叉树的遍历

令:

```
L：遍历左子树
R：遍历右子树
D：访问根结点
```

有六种遍历二叉树的方法： 基本：DLR、LDR、LRD 镜像：DRL、RDL、RLD

约定先左后右，有三种遍历方法：

1. 先序遍历：DLR
2. 中序遍历：LDR
3. 后序遍历：LRD

先序遍历:DLR

```
typedef int ElemType;//替换成树的数据类型
typedef enum { OK, ERROR } Status;

typedef struct BiTree{
    int data;
    BiTree *lchild, *rchild;
};

void PreOrder(BiTree T,Status(*Visit)(ElemType e)){
    if(T){
        if (Visit(T->data))
```

```
            if (PreOrder(T->lchild, Visit))
                if (PreOrder(T->rchild, Visit))
                    return OK;
        return ERROR;
    }
    else return ok;
}//PreOrder

Status Visit(const ElemType e){
    printf("%d", e);
    return OK;
}//Visit
```

中序遍历:LDR

采用递归算法

```
typedef int ElemType;//替换成树的数据类型
typedef enum { OK, ERROR } Status;

typedef struct BiTree{
    int data;
    BiTree *lchild, *rchild;
};

void InOrder(BiTree T, Status(*Visit)(ElemType e)) {
    if (T) {
        InOrder(T->lchild, Visit);
        Visit(T->data);
        InOrder(T->rchild, Visit);
    }
};//InOrder

Status Visit(const ElemType e) {
    printf("%d", e);
    return OK;
}//Visit
```

采用非递归算法（用栈）：

```
Status InTrav(BiTree T, void (*Visit)(ElemType e)) {
    InitStack(S);
```

```
    BiTree p = T;
    while (p || !StackEmpty(S)) {
        if (p) {
            Push(S, p);
            p = p->lchild;
        }
        else {
            Pop(S, p);
            Visit(p->data);
            p = p->rchild;
        }
    }//while

    DestroyStack(S);
    return OK;
}//InTrav
```

后序遍历:LRD

```
typedef int ElemType;//替换成树的数据类型
typedef enum { OK, ERROR } Status;

typedef struct BiTree{
    int data;
    BiTree *lchild, *rchild;
};

void PostOrder(BiTree T, Status(*Visit)(ElemType e)) {
    if (T) {
        PostOrder(T->lchild, Visit);
        PostOrder(T->rchild, Visit);
        Visit(T->data);
    }
}
```

编写求二叉树叶子结点个数的算法

```
int num=0;

void LeafCount(BiTree T) {
    if (T) {
        if (T -> lchild == NULL && T->rchild == NULL)
            num++;
```

```
        else {
            LeafCount(T->lchild);
            LeafCount(T->rchild);
        }
    } else {
        num = 0;
    }
}// LeafCount
```

```
int LeafCount(BiTree T) {
    if (T) {
        if (T -> lchild == NULL && T->rchild == NULL)
            return 1;
        else {
            return LeafCount(T->lchild) + LeafCount(T->rchild);
        }
    } else {
        return 0;
    }
}// LeafCount
```

求二叉树的深度

```
int getDepth(BiTree T) {
    if (!T) return 0;
    else {
        int leftdepth = getDepth(T->lchild);
        int rightdepth = getDepth(T->rchild);
        return 1 + leftdepth > rightdepth ? leftdepth : rightdepth
;
    }
}
```

根据遍历还原二叉树

| | DLR | LDR | LRD | | DLR | no | yes | no | | LDR | yes | no | yes | | LRD | no | yes | no |

根据先序和中序来还原

1. 先序遍历的第一个元素是根节点的值。
2. 在中序遍历中找到根节点的值，它将中序遍历序列分为左子树和右子树两部分。
3. 根据左子树和右子树的长度，在先序遍历中确定左子树和右子树的范围。

4. 递归地构建左子树和右子树。

根节点的位置 ps 分情况鉴别：

1. 无右子树：

   ||长度|在pre中的起点|在ino中的起点| |左子树|k-is|ps+1|is| ||n-1|ps+1|is| |右子树|k==is+n-1|||

2. 无左子树：

   ||长度|在pre中的起点|在ino中的起点| |左子树|k==is||| |右子树|n-(k-is)-1|ps+(k-is)+1|k+1| ||n-1|ps+1|k+1|

3. 有左子树又有右子树：

   ||长度|在pre中的起点|在ino中的起点| |左子树|k-is|ps+1|is| |右子树|n-(k-is)-1|ps+(k-is)+1|k+1|

```
typedef struct BiTNode {
    char data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

int Search(char ino[], int start, int end, char value) {
    for (int i = start; i <= end; i++) {
        if (ino[i] == value) {
            return i;
        }
    }
    return -1;
}

void createBT(BiTree &T, char pre[], char ino[], int ps, int is,
int n) {
    if ( n <= 0) T=NUll;
    else {
        k = Search(ino, pre[ps]);
        if (k==-1) T=NULL;
        else {
            T = (BiTNode *) malloc (sizeof (BiTNode));
            T -> data = pre[ps];

            if (k==is) T->lchild = NULL;
            else createBT(T->lchild, pre, ino, ps+1, is, k-is);

            if (k==is+n-1) T->rchild = NULL;
```

```
            else createBT(T->rchild, pre, ino, ps+(k-is)+1, k+1,
n-(k-is)-1);

        }
    }
}
```

线索二叉树

现在用的少,借用结点的空的左右指针域来存放指向其一级前驱和二级前驱的指针。

```
typedef enum { Link, Thread } PointerTag;

typedef struct BiThrNode {
    char data;
    struct BiThrNode *lchild, *rchild;
    PointerTag ltag, rtag;
} BiThrNode, *BiThrTree;
```