树和二叉树

树和二叉树的定义

树（Tree）

是n(n>=0)个结点的有限集，它或为**空树**（n=0）；或为非空树，对于非空树T：

1. 有且仅有一个称之为**根**的结点。
2. 除根结点以外的其余节点可分为m个互不相交的有限子集合称为**子树**（`SubTree`）。

树可以用**嵌套集合、凹入表示、广义表**来表示。

名词含义

- 根——根结点（**没有前驱**）
- 叶子——终端结点（**没有后继**）
- 森林——m棵不相交的树的集合（*例如将树T的根A删除后的子树个数*）
- 有序树——结点各子树从左到右有序，不能互换（左为一）
- 无序树——结点各子树可以互换位置
- 双亲——即上层的结点（直接前驱）
- 孩子——即下层节点的子树的根（直接后继）
- 兄弟——同一双亲下的同层节点
- 堂兄弟——即双亲位于同一层的结点。
- 祖先——即从根到该结点所经过的所有结点。
- 子孙——即该结点下层子树中的任一结点。
- 结点——树的数据元素。**树中结点数 = 总分叉数 （总度数）+1**
- 结点的度——结点直接挂接的子树数，等于该结点的孩子数。
- 结点的层次——从根到该结点的层数（根结点算第一层）。
- 终端结点——即度为0的结点，即叶子。
- 分支结点——即度不为0的结点（也称内部结点）。
- 树的度——该树中所有结点的度中的最大值。

- 树的深度——指所有结点中最大的层数。

二叉树（Binary Tree）

是n（n>=0）个结点所构成的集合，它或为空树（n=0）；或为非空树，对于非空树T：

1. 有且仅有一个根结点。
2. 除根结点外的其余节点分为两个互不相交的子集T_1和T_2，分别为T的左子树和右子树，且T_1T_2本身又都是二叉树。

特点

- 二叉树结构简单，规律性强。
- 可以证明，所有普通树都可以转化为唯一对应的二叉树，不失一般性。
- 普通树（多叉树）若不转化为二叉树，则运算很难实现。
- **结点的度都小于或等于2。**
- **有序树（子树有序，不可颠倒）。**

数据编码

将数据文件转换成由0、1组成的二进制串，称为编码。

1. 等长编码
2. 哈夫曼编码
3. 不等长编码

利用二叉树表达式求值

> 用二叉树表示表达式的递归定义如下：
> (1) 若表达式仅为数或简单变量，则相应二叉树中仅有—个根结点，其数据域存放该表达式信息；
> (2) 若表达式为"第—操作数 运算符 第二操作数"的形式，则相应的二叉树中以左子树表示第一操作数，右子树表示第二操作数，根结点的数据域存放运算符（若为一元运算符，则左子树为空），其中，操作数本身又为表达式。

树和二叉树的抽象数据类型定义

```
ADT BinarTree{
   Data_Object:
   Data_Relationship:
}

//基本操作P:
CreateBiTree(&T,definition)
//初始条件;  definition给出二叉树T的定义。
//操作结果：按definition构造二叉树T。

PreOrderTraverse(T)
//初始条件：二叉树T存在。
//操作结果：先序遍历T，对每个结点访问一次。

InOrderTraverse(T)
//初始条件：二叉树T存在。
//操作结果：中序遍历T，对每个结点访问一次。

PostOrderTraverse(T)
//初始条件：二叉树T存在。
//操作结果：后序遍历T，对每个结点访问一次。
```

二叉树的性质和存储结构

> 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。
> 3. 任一二叉树，其终端结点数（度为0的结点）为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。计算结点总数：$n=n_0+n_1+n_2=2n_0+n_1-1$。
> 4. 具有n个结点的完全二叉树的深度必为$\lfloor log_2n \rfloor+1$。
> 5. 如果对一棵有n个结点的完全二叉树（其深度为Llog2 n」+1)的结点按层序编号（从第1层到第|log2n|+1层，每层从左到右)，则对任一结点i(1=<i=<n)，有：
>   (1)如果i=1, 则结点i是二叉树的根，无双亲；如果i>l，则其双亲PARENT(i)是结点i/2」。
>   (2) 如果2i>n，则结点i无左孩子（结点i为叶子结点）；否则其左孩子LCHILD(i)是结点2i。
>   (3)如果2i+1>n，则结点i无右孩子；否则其右孩子RCHILD(i)是结点2i+1。

满二叉树和完全二叉树的区别

> 满二叉树是每一层都是满的；完全二叉树前n-1层都是满的，最底层在右侧可以缺少若干个连续结点。
>
> **满二叉树是完全二叉树的一个特例。**
>
> 满二叉树：一棵深度为空且有2^k-1个结点的二叉树，每一层都满了。
>
> 完全二叉树：深度为k的，有n个结点的二叉树，当且仅当每个节点都与深度k的满二叉树中编号从1到n的结点对应。最后一层叶子可以不满，但必须集中在左边。

二叉树的顺序存储

实现：按满二叉树的结点层次编号，一次存放而二叉树中的数据元素。

特点：结点间关系蕴含在其存储位置中；浪费空间，***适于存储满二叉树和完全二叉树***。

0	1	2	3	4	5
	a	b	c	d	e

从一号单元开始存储数据，方便下标编号体现双亲孩子关系。

二叉树的链式存储

二叉链表结点结构：

`lchild`	`data`	`rchild`

三叉链表结点结构：

`lchild`	data	parent	`rchild`

```
     parent
       |
      data
     /  \
   lchild rchild
```

```
//二叉链表
typedef struct BiNode{
    TElemType data;
    struct BiNode *lchild,*rchild;
}BiNode,*BiTree;
```

在n个结点的二叉链表中，有 `n+1` 个空指针域。空指针数目=2n-(n-1)=n+1。

```
//三叉链表
typedef struct TriTNode{
    TElemType data;
    struct TriTNode *lchild,*parent,*rchild;
}TriTNode,*TriTree;
```

遍历二叉树和线索二叉树

遍历二叉树

> **遍历**：按某条搜索路线边防每个节点且不重复。它是树结构进行插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。

分类

DLR——先序遍历——先根，再左，再右

LDR——中序遍历——先左，再根，再右

LRD——后序遍历——先左，再右，再根

> 示例：A/B*C*D+E
>
> ---
>
> **先序遍历：**
>
> +**/ABCDE
>
> 前缀表达式、波兰表示
>
> ---

> **中序遍历：**
>
> A / B * C * D + E
>
> 中缀表达式
>
> ---
>
> **后序遍历：**
>
> A B / C * D * E +
>
> 后缀表达式，逆波兰表示
>
> ---
>
> **层序遍历：**
>
> +* E * D / C A B

先序遍历

```
Status PreOrderTraverse(BiTree T){
    if(T == NULL)
        return ok;//空二叉树
    else{
        cout<< T->data;//访问根节点
        PreOrderTraverse(T->lchild);//遍历左子树
        PreOrderTraverse(T->rchild);//遍历右子树
    }
}
```

中序遍历

```
Status InOrderTraverse(BiTree T){
    if(T == NULL)
        return ok;
    else{
        InOrderTraverse(T->lchild);
        cout<< T->data;
        InOrderTraverse(T->rchild);
    }
}
```

中序遍历的非递归算法:

```
void InOrderTracese(BiTree T){
    InitStack(S);
    p=T;
    q= new BiTNode;
    while(p||!StackEmpty(S)){
        if(p){//p非空时
            push(S,p);//根指针进栈
            p=p->lchild;//根指针进栈，遍历左子树
        }
        else{//p为空时
            pop(S,q);//退栈
            cout<<q->data;//访问根节点
            p=q->rchild;//遍历右子树
        }
    }
}
```

后序遍历

```
Status PostOrderTraverse(BiTree T){
    if(T == NULL)
        return OK;
    else{
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        cout<< T->data;
    }
}
```

> 如果去掉输出语句，从递归角度看，三种算法访问路径相同，只是访问结点的实际不同。
>
> **时间效率：O(n)** //每个结点只访问一次
>
> **空间效率：O(n)** //树的深度，嘴还情况栈占用最大辅助空间为n

二叉树的建立

```
//先序输入序列创建二叉树

void CreateBiTree(BiTree &T){
    cin>>ch;
    if(ch=='#')
        T=NULL;//递归结束，建空树
    else{
        T= new BiTNode;
        T->data = ch;//生成根节点
        CreateBiTree(T->lchild);//递归创建左子树
        CreateBiTree(T->rchild);//递归创建右子树
    }
}
```

二叉树遍历算法的应用

1. 统计二叉树中结点总数

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

2. 描述叶子

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

3. 计算二叉树的深度

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

总结

1. 由**前序序列和中序序列**能确定唯一二叉树，由**后序序列和中序序列**能确定唯一二叉树。
2. 由前序序列和后序序列不一定能确定唯一二叉树。

线索二叉树

线索二叉树：有n个结点的二叉链表有n+1个空指针域。可以用它们来存放当前节点的**直接前驱和后继**等线索，以加快查找速度。

解决方法：

- 增加两个指针域：`fwd` 和 `bwd`
- 利用空链域（n+1个空链域）

具体操作：

1. 若结点有左子树，则 `lchild` 指向其左孩子；否则，指向其直接前驱（即线索）。
2. 若结点有右子树，则 `rchild` 指向其有孩子；否则，指向其直接后继。

增加两个标志域避免混淆：

`lchild`	`LTag`	**data**	`RTag`	`rchild`

```
LTag (int) : 若LTag==0, lchild域指向左孩子;
             若LTag==l, lchild域指向其前驱;
RTag (int) : 若RTag==0, rchil心或指向右孩子;
             若RTag== 1, rchild域指向其后继;
```

树和森林

树的存储结构

双亲表示法

```c
#define MAX_TREE_SIZE 100
typedef struct PTNode{
    TElemType data;
    int parent;//双亲位置域
}PTNode;//树结点结构

typedef struct{
    PTNode nodes[MAX_TREE_SIZE];
    int n,r;//结点总数和根的位置
}PTree;//树结构
```

- 便于涉及双亲的操作、根的操作
- 涉及结点的孩子操作时不方便。

孩子表示法

```
#define MAX_TREE_SIZE 100
typedef struct PTNode{
    TElemType data;
    int child1;
    int child2;
    ...
    int childd;//第d个孩子位置域
}PTNode;

typedef struct{
    PTNode nodes[MAX_TREE_SIZE];
    int n,r;
}PTree;
```

- 便于涉及孩子的操作，求双亲不方便
- 采用同构的结点，空间浪费。

孩子链表表示法

```
typedef struct CTNode{
    int child;
    struct CTNode *next;
}*ChildPtr;

typedef struct{
    TElemType data;
    ChildPtr firstchild;
}CTBox;

typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int n,r;
}CTree;
```

- 解决了空间浪费问题
- 便于涉及孩子的操作

- 求结点的双亲时不方便

孩子兄弟法（二叉链表表示法）

```
typedef struct CSNode{
    ElemType data;
    struct CSNode
        *firstchild,*nextsibling;
}CSNode,*CSTree;
```

树与二叉树的转换

树转换成二叉树

> 兄弟与双亲断开
>
> 变成长兄（左边哥哥）的孩子

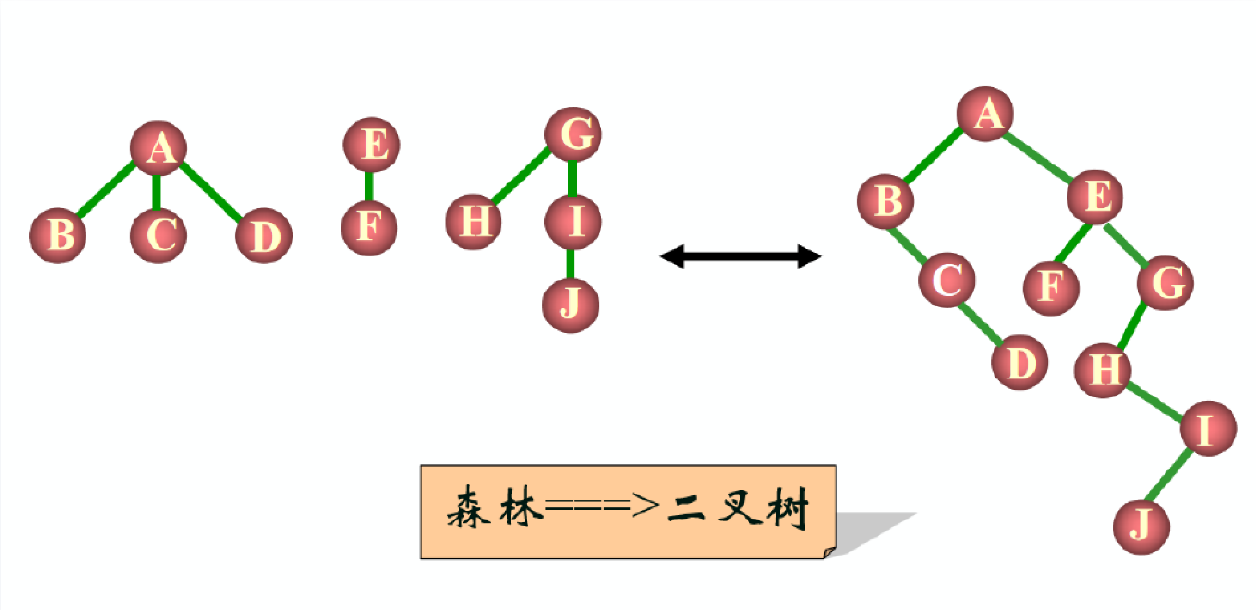
二叉树转换成树

> 有孩子与双亲接上
>
> 变成长兄（左边哥哥）的兄弟

森林与二叉树转换

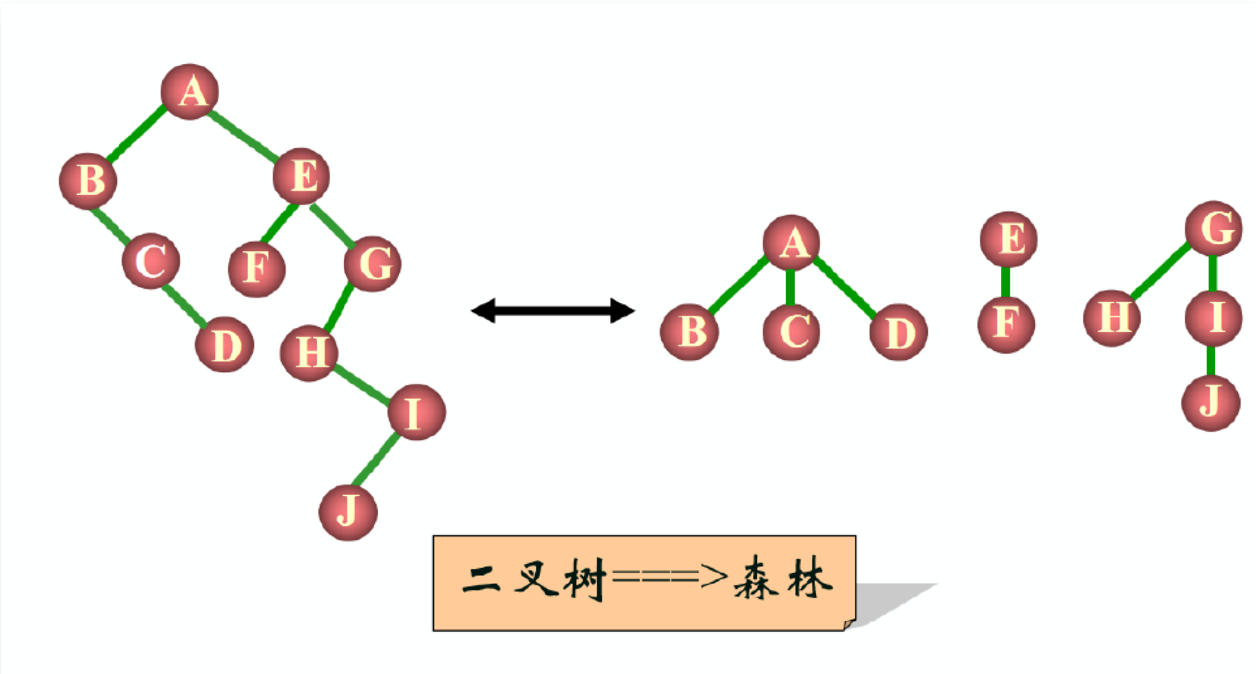
森林转换成二叉树

> 如果F={$T_1, T_2, ..., T_m$}是森林则可按如下规则转换成一棵二叉树B=(root,LB,RB)。
> (1)若F为空，即m=0则B为空二叉树；
> (2)若F非空，即$m! = 0$，则B的根root即为森林中第一棵树的根ROOT(T_1); **B的左子树LB是从T_1中根结点的子树森林F_1 ={$T_{11}, T_{12}, ..., T_{1m_1}$ }转换而成的二叉树；其右子树RB是从森林F={$T_2, T_3, ..., T_m$ }转换而成的二叉树。**



二叉树转换成森林

> 如果B=(root,LB,RB)是一棵二叉树，则可按如下规则转换成森林F={$T_1, T_2, ..., T_m$}:
> (1)若B为空，则F为空;
> (2)若B非空，则F中**第一棵树T_1的根ROOT(T1)即为二叉树B的根root**; T_1中的**根结点的子树森林F_1是由B的左子树LB转换而成的森林**；F中**除T_1之外其余树组成的森林F'=={$T_2, T_3, ..., T_m$} 是由B的右子树RB转换而成的森林**。



树和森林的遍历

树的遍历

1. 先根遍历

（1）访问树的根结点；

（2）从左到右依次先根遍历每颗子树。

2. 后根遍历

（1）从左到右先孩子再双亲，依次后根遍历每颗子树。

（2）最后访问树的根节点。

森林的遍历

1. 先序遍历森林

前提，森林非空。

（1）访问森林中第一棵树的根结点；

（2）先序遍历第一棵树的根结点的子树森林；

（3）先序遍历除去第一棵树后的森林。

即对森林中的树依次进行先序遍历，对单棵树的遍历要将根的子树视作森林。

2. 中序遍历森林

前提，森林非空。

（1）中序遍历第一棵树的根结点的子树森林。

（2）访问森林中第一棵树的根结点。

（3）中序遍历出去第一棵树之后的森林。

森林的先序遍历=二叉树的先序遍历；森林的中序遍历=二叉树的中序遍历。

哈夫曼树及其应用

> 哈夫曼编码是最有不等长编码方案，是前缀编码方案。

关键：要设计长度不等的编码，必须使**任意字符**的编码**都不是**另一**字符编码的前缀**-前缀编码（前缀无歧义编码）

编码与译码

> 哈夫曼**编码过程**：根据字符出现概率构造哈夫曼树，左分支为0，右分支为1编码。

> 哈夫曼**译码过程**：从根出发，遇“0”向左，遇“1”向右；到达叶子结点，则译出一个字符，反复由根出发，直到译码完成。

> 哈夫曼**编码特点**：每一码都不是另一码的前缀，绝不会译错，称为前缀编码（前缀无歧义编码）。

哈夫曼树的基本概念

- 路径——从树中一结点到另一结点间的分支所构成。
- 路径长度——路径上的分支数目。
- 结点的带权路径长度——该结点到树根的路径长度与结点上权的乘积。
- 树的带权路径长度——树中所有叶子结点的带权路径长度之和。
- 哈夫曼树——带权路径长度最小的二叉树，最优树。

哈夫曼树的构造过程

1. 根据给定的n个权值$\{W_1, W_2, ..., W_n\}$，构造**n棵只有根结点的二叉树**。
2. 在森林中选取两棵根结点**权值最小的树作左右子树**，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。
3. 在森林中**删除这两棵树**，同时将新得到的二叉树加入森林中。
4. 重复上述两步， 直到只含一棵树为止，这棵树即哈夫曼树。

> 思想要点：权值小的结点远离根，权值大的结点靠近根。
>
> 要点：对权值的合并、删除与替换，总是合并当前值最小的两个。

> 基本思想：出现概率大的字符用短码，出现概率小的字符用长码。

哈夫曼树构造算法的实现

一棵有n个叶子结点的哈夫曼树有**2n-1**个结点。

采用**顺序存储**结构——**一维数组**。

```
//结点类型定义
typedef struct {
    int weight;
    int parent ,lchild,rchild;
}HTNode,*HuffmanTree;
```

01 初始化HT[1...2n-1]: `HT[i].lch = HT[i].rch = HT[i]. parent = 0`
02 初始化HT[1...n]： 输入HT[i].weight //第1,第2步HT树初始状态
03 进行以下n-1次合并，依次产生HT[i]，i=n+ 1,...,2n-l

- - 在HT[1...i-1] 中选两个未被选过的weight最小的两个结点HT[s1]和HT[s2] （从parent= 0 的结点中选）
  - 修改HT[s1]和HT[s2] 的parent值： `parent=i`
  - 置HT[i]: `weight=HT[s1].weight + HT[s2].weight ,lch=s1, rch=s2`

算法

```
void CreatHuffmanTree(HuffmanTree HT,int n){
    if(n≤1)
        return 0;
    m=2*n-1;
    HT= new HTNode[m+1];

    //01 0号单元未用，所以需要动态分配m+1个单元，HT[m]表示根结点
    for(i=1;i≤m;i++){
        HT[I].lch=0;
        HT[i].rch=0;
        HT[i].parent=0;
    }

    //02
```

```
    for(i=1;i≤n;++i){
        cin>>HT[i].weigth;
    }

    //03
    for(i=n+1;i≤m;++i){//构造哈夫曼树，n-1次合并
        Select(HT,i-1,s1,s2);
        //在HT[K]中选择两个其双亲域为0
        //且权值最小的结点
        //并返回他们在HT中的序号s1和s2
        HT[s1].parent=i;
        HT[s2].parent=i;
        //表示从F中删除s1，s2
        HT[i].lch=s1;
        HT[i].rch=s2;
        //s1和s2分别作为i的左右孩子
        HT[i].weight=HT[s1].weight+HT[s2].weight;
        //i的权值为左右孩子权值之和
    }
}
```

哈夫曼编码结论

- 哈夫曼编码是不等长编码，按字符出现概率构建哈夫曼树编码分配码长，平均码长最短。
- 哈夫曼编码是前缀编码（前缀无歧义编码），即任一字符的编码都不是另一字符编码的前缀。
- 哈夫曼编码树中没有度为1 的结点。若叶子结点的个数为n, 则哈夫曼编码树的结点总数为2n-1 。
- 发送过程：根据由哈夫曼树得到的编码表送出传输字符对应的编码数据。
- 接收过程：按左0、右l1的约定，从哈夫曼树根结点走到—个叶结点，完成—个字符的译码。反复此过程，直到接收数据译码结束。