数据结构——树和二叉树

主讲：张昱
yuzhang@ustc.edu
0551-3603804

第六章 树和二叉树

重点：二叉树的性质、遍历；二叉树与森林(树)的相互转换

难点：树和二叉树应用的算法设计

第六章 树和二叉树

6.1 树的定义和基本术语
6.2 二叉树
6.3 遍历二叉树和线索二叉树
6.4 树和森林
6.6 赫夫曼树及其应用
6.5 树与等价问题
6.7 回溯法与树的遍历
6.8 树的计数

6.1 树的定义和基本术语

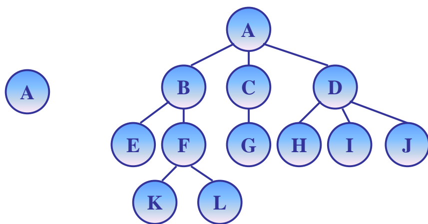
- 树的定义(递归定义)

树是$n(n \geq 0)$个结点(元素)的有限集。
若 $n=0$，称为空树。
若 $n > 0$，则
- 有且仅有一个特定的称为根的结点root;
- 当$n > 1$时，除根以外的其他结点划分为$m(m>0)$ 个互不相交的有限集$T_1, T_2, \ldots T_m$，其中每一个集合本身又是一棵树，并且称为根的子树。

且对任意的$i(m \geq i \geq 1)$, T_i存在惟一的结点x_i，有 $\langle root, x_i \rangle \in H$ H为树中元素之间的二元关系集

6.1 树的定义和基本术语

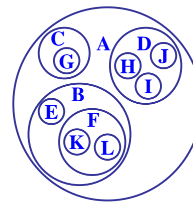
- 树的表示



6.1 树的定义-其他表示

- 树的其他表示

嵌套集合、广义表表示、凹入表示

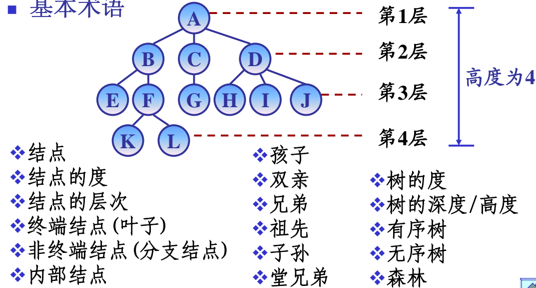


(A(B(E, F(K,L)), C(G), D(H, I, J)))

A*****************
B****************
E***************
F***************
K**************
L**************
C****************
G***************
D****************
H***************
I***************
J***************

6.1 树的定义-基本术语

- 基本术语



- ❖结点
- ❖结点的度
- ❖结点的层次
- ❖终端结点(叶子)
- ❖非终端结点(分支结点)
- ❖内部结点
- ❖孩子
- ❖双亲
- ❖兄弟
- ❖祖先
- ❖子孙
- ❖堂兄弟
- ❖树的度
- ❖树的深度/高度
- ❖有序树
- ❖无序树
- ❖森林

ADT Tree{

数据对象：$D=\{a_i \mid a_i \in ElemSet, i=1,2,\cdots,n, \quad n \geq 0\}$

数据关系：若 D 为空集，则称为空树；

若 D 仅含一个数据元素，则 R 为空集，否则 R={H}，H 是如下二元关系：

(1) 在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

(2) 若 $D-\{root\} \neq \Phi$，则存在 D-{root}的一个划分 $D_1, D_2, \cdots, D_m$ $(m>0)$（D_i 表示构成第 i 棵子树的结点集），对任意 $j \neq k$ $(1 \leq j, k \leq m)$ 有 $D_j \cap D_k = \Phi$，且对任意的 i $(1 \leq i \leq m)$，唯一存在数据元素 $x_i \in D_i$，有 $<root, x_i> \in H$（H 表示结点之间的父子关系）；

(3) 对应于 D-{root}的划分，$H-\{<root, x_1>,\cdots,<root, x_m>\}$有唯一的一个划分 $H_1, H_2, \cdots, H_m(m>0)$（$H_i$ 表示第 i 棵子树中的父子关系），对任意 $j \neq k(1 \leq j,k \leq m)$有 $H_j \cap H_k = \Phi$，且对任意 $i(1 \leq i \leq m)$，H_i 是 D_i 上的二元关系，$(D_i, \{H_i\})$是一棵符合本定义的树，称为根 root 的子树。

基本操作：

InitTree(&T)
　操作结果：构造空树 T

DestroyTree(&T)
　初始条件：树 T 已存在
　操作结果：销毁树 T

ClearTree(&T)
　初始条件：树 T 已存在
　操作结果：将树 T 清为空树

TreeEmpty(T)
　初始条件：树 T 已存在
　操作结果：若 T 为空树，则返回 TRUE，否则返回 FALSE

TreeDepth(T)
　初始条件：树 T 已存在
　操作结果：返回树 T 的深度

Root(T)
　初始条件：树 T 已存在
　操作结果：返回 T 的根

Value(T, cur_e)
　初始条件：树 T 已存在，cur_e 是 T 中某个结点
　操作结果：返回 cur_e 的值

Assign(T, &cur_e, value)
　初始条件：树 T 已存在，cur_e 是 T 中某个结点
　操作结果：结点 cur_e 赋值为 value

Parent(T, cur_e)
　初始条件：树 T 已存在，cur_e 是 T 中某个结点
　操作结果：若 cur_e 是 T 的非根结点，则返回它的双亲，否则函数值为"空"

LeftChild(T, cur_e)
　初始条件：树 T 已存在，cur_e 是 T 中某个结点
　操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子，否则返回"空"

RightSibling(T, cur_e)
　初始条件：树 T 已存在，cur_e 是 T 中某个结点
　操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，否则返回"空"

InsertChild(&T, p, i, c)
　初始条件：树 T 已存在，p 指向 T 中某个结点，$1 \leq i \leq$ p 所指结点的度+1，非空树 c 与 T 不相交
　操作结果：插入 c 为 T 中 p 所指结点的第 i 棵子树。

DeleteChild(&T, p, i)
　初始条件：树 T 已存在,p 指向 T 中某个结点，$1 \leq i \leq$ p 所指结点的度
　操作结果：删除 T 中 p 所指结点的第 i 棵子树。

TraverseTree(T, visit( ))
　初始条件：树 T 已存在，visit 是对结点操作的应用函数
　操作结果：按某种次序对 T 的每个结点调用函数 visit( )一次且至多一次。一旦 visit( )失败，则操作失败

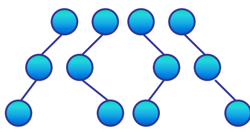
}ADT Tree

6.1 树的定义-ADT Tree

- **ADT Tree**
  - 查找：Parent(T,cur_e)　LeftChild(T, cur_e)　RightSibling(T, cur_e)
  - 插入：InsertChild(&T, &p, i, c)
  - 删除：DeleteChild(&T, &p, i)
  - 遍历：TraverseTree(T, Visit())

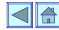
6.2 二叉树

- 二叉树的定义(递归定义)
  - 特点：每个结点至多只有两棵子树，子树有左右之分
  - 二叉树 vs 度不大于2的有序树
  - ADT BinaryTree



二叉树

有序树



6.2 二叉树-性质(1)

- 二叉树的性质
  - 性质1：二叉树的第i层至多有2^{i-1}个结点(i≥1)
  - 性质2：深度为h的二叉树至多有2^h-1个结点(h≥1)

    思考：性质1和性质2推广到k叉树，结果会如何？

    k^{i-1}　$(k^h-1)/(k-1)$
  - 性质3：$n_0 = n_2 + 1$

    结点总数　$n = n_0 + n_1 + n_2$

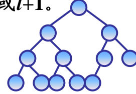
    分支数　$n\text{-}1 = n_1 + 2n_2$

    思考：若包含有n个结点的树中只有叶子结点和度为k的结点，则该树中有多少叶子结点？

    $n = n_0+n_k,\ n-1 = kn_k \Rightarrow n_0 = n-(n-1)/k$

6.2 二叉树-性质(2)

- 满二叉树：一棵深度为k且有2^k-1个结点的二叉树(k≥0)
- 完全二叉树：对于深度为k的完全二叉树，则
  1) 前k-1层为满二叉树;
  2) 第k层结点依次占据最左边的位置;
  3) 一个结点有右孩子，则它必有左孩子;
  4) 度为1的结点个数为0或1
  5) 叶子结点只可能在层次最大的两层上出现;
  6) 对任一结点，若其右分支下的子孙的最大层次为l, 则其左分支下的子孙的最大层次必为l或l+1。



6.2 二叉树-性质(3)

- 性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$

  由性质2　$2^{k-1}\text{-}1 < n \le 2^k\text{-}1$　或　$2^{k-1} \le n < 2^k$

  于是　　$k\text{-}1 \le \log_2 n < k$

  例：若一个完全二叉树有1450个结点，则度为1的结点个数为<u>1</u>，度为2的结点个数为<u>724</u>，叶子结点的个数为<u>725</u>，有<u>725</u>个结点有左孩子，有<u>724</u>个结点有右孩子；该树的高度为<u>11</u>。(性质3、性质4以及完全二叉树的特征)

6.2 二叉树-性质(4)

- 性质5：如果对一棵有n个结点的完全二叉树的结点按层序编号(从第1层到第$\lfloor \log_2 n \rfloor + 1$层，每层从左到右)，则对任一结点$i$ ($1 \le i \le n$)，有

  (1) 如果i=1，则结点i是二叉树的根，无双亲；如果i >1，则其双亲是结点$\lfloor i/2 \rfloor$；

  (2) 如果$2i > n$，则结点i无左孩子(结点i为叶子结点)；否则其左孩子是结点$2i$；

  (3) 如果$2i + 1 > n$，则结点i无右孩子；否则其右孩子是结点$2i + 1$。

  思考：性质5推广到k叉树，结果会如何？

6.2 二叉树-顺序存储结构(1)

- 二叉树的顺序存储结构
  - 类型定义

    typedef ElemType SqBiTree[MAX_TREE_SIZE];

    //0号单元存储根结点

    1) 依据性质5，用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素；

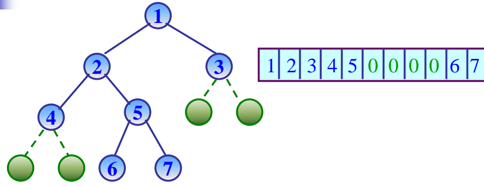
    ——结点在存储区中的相对位置反映它们逻辑上的关系

    2) 仅适用于完全二叉树
  - 一般二叉树的顺序存储方法

    通过补虚结点，将一般的二叉树变成完全二叉树

    空间开销大！

6.2 二叉树-顺序存储结构(2)



- 空间利用率问题：
  在最坏情况下，一个深度为k且只有k个结点的单支树(树中不存在度为2的结点)，则需要长度为2^k-1的一维数组。

6.2 二叉树-链式存储结构

- 二叉树的链式存储结构
  - 引入辅助空间表示结点之间的关系：双亲-孩子
    二叉链表(左、右孩子链域)
    三叉链表(双亲及左、右孩子链域)
  - 二叉链的类型定义(动态链表)

```
typedef struct BiTNode{
  ElemType data;
  struct BiTNode  *lchild, *rchild;     // 左右孩子指针
}BiTNode, *BiTree;
```

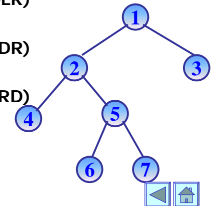
若有n个结点，则共有$2n$个链域；其中n-1不为空，指向孩子；另外n+1个为空链域

6.3 遍历二叉树和线索二叉树

- 遍历二叉树
- 基于先/中/后序遍历算法的应用
- 为什么强调使用函数调用的结果
- 先/中/后序遍历的非递归算法
- 层次遍历算法及其应用
- 二叉树的创建方法
- 线索二叉树

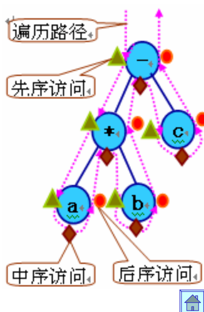
6.3 -遍历二叉树(1)

- 遍历二叉树
  按一定的搜索路径对树中的每一结点访问且仅访问一次
  - 遍历时的搜索路线(约定：先左后右，D-根,L-左子树,R-右子树)
    - 先(根)序遍历：1 2 4 5 6 7 3　(DLR)
    - 中(根)序遍历：4 2 6 5 7 1 3　(LDR)
    - 后(根)序遍历：4 6 7 5 2 3 1　(LRD)
    - 层次遍历：1 2 3 4 5 6 7



6.3 -遍历二叉树(2)

- 先/中/后序遍历的区别
  如右图，三者经过的搜索路线是相同的；只是访问结点的时机不同。每一结点在整个搜索路线中会经过3次：
  - 第一次进入到该结点
    此时访问该结点，称为**先序遍历**
  - 由左子树回溯到该结点
    此时访问该结点，称为**中序遍历**
  - 由右子树回溯到该结点
    此时访问该结点，称为**后序遍历**



6.3 -遍历二叉树(3)

- 遍历算法的递归实现
  - 二叉树的递归定义性质，决定了它的很多算法都可用递归实现，遍历算法就是其中之一。
  - 对于二叉树的遍历，可以不去具体考虑各子问题(左子树、根、右子树)的遍历结果是什么，而去考虑如何由各子问题的求解结果构成原问题(二叉树)的遍历结果——**递归规律的确定**。必须注意的是，当二叉树小到一定程度，即空树时，应直接给出解答——**递归结束条件及处理**。

6.3 -遍历二叉树(4)

- 先序遍历

```
Status  PreOrderTraverse( BiTree T,
          Status ( *Visit ) (ElemType e) ){
  if ( T != NULL ){
    if ( Visit(T->data) )
      if ( PreOrderTraverse( T->lchild, Visit ) )
        if ( PreOrderTraverse( T->rchild, Visit ) )
           return OK;
    return ERROR;
  }
  else return OK;
}
```

假设二叉树中结点数为n,高度为h,
则T(n,h)=O(n), S(n,h) = O(h)
$\lfloor \log_2 n \rfloor + 1 \leqslant h \leqslant n$

6.3 -基于先/中/后序遍历算法的应用

- 基于先/中/后序遍历的算法应用
  - 基于先序遍历的二叉树(二叉链)的创建
  - 统计二叉树中叶子结点的数目
  - 释放二叉树的所有结点空间
  - 删除并释放二叉树中以元素值为x的结点作为根的各子树
  - 求位于二叉树先序序列中第k个位置的结点的值

分析问题本身的特征，选择合适的遍历次序！
应用递归编写算法，简洁！
注意：递归结束条件，用递归调用的结果

例1 基于先序遍历的二叉树(二叉链)的创建

【本例特征】
如何基于二叉树的先序、中序、后序遍历的递归算法进行问题求解？
【思路】

	先 序 遍 历 PreOrderTraverse	创 建 CreateBiTree
输入	二叉链表示的二叉树的头指针 T	带虚结点的先序序列 ch
输入的表现方式	参数	由输入设备输入 scanf(&ch)
输出	对结点的访问结果	二叉链表示的二叉树的头指针 T
输出的表现方式	由输出设备输出	变参
空树（递归结束）的条件及处理(直接求解)	T == NULL	ch == END_DATA（表示虚结点的值）
	空	T = NULL
根结点的访问(子问题直接求解)	Visit(T->data)	T = (BiTree)malloc(sizeof(BiTNode)) T->data = ch
左子树(使用递归调用的解)	PreOrderTraverse(T->lchild)	CreateBiTree(T->lchild)
右子树(使用递归调用的解)	PreOrderTraverse(T->rchild)	CreateBiTree(T->rchild)

例1 基于先序遍历的二叉树(二叉链)的创建

```
Status  CreateBiTree(BiTree &T) {
  //按先序次序输入二叉树中结点的值(一个字符)
  //空格字符表示空树,构造二叉链表表示的二叉树T
  scanf(&ch);
  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
```


例2 统计二叉树中叶子结点的数目(1)

【本例特征】
如何通过全局变量、变参、返回值三种渠道返回处理结果？
【思路】
在遍历二叉树时，对一些特殊的结点(无左右孩子)进行计数。可以将遍历算法的结点访问操作修改为对特殊结点的判定和计数过程，需要注意的是计数器的处理方式。
可以有以下几种计数处理：
- 用遍历函数的返回值传出求得的叶子结点的数目；
- 为遍历函数增加一个引用参数，用来传出指定二叉树的叶子结点数目。
- 引入全局的计数器，初始为0；

此处，遍历次序的选择对本算法没有太大影响。

例2 统计二叉树中叶子结点的数目(2)

【算法1 全局的计数器】

```
// n为叶子结点的计数器
int   n=0;
void leaf(BiTree T)
{
    // 利用二叉树的先序遍历
    if (T != NULL ){
        // 访问结点->叶子的判定和计数
        if( T->lchild==NULL && T->rchild==NULL ) n++;
        leaf(T->lchild);
        leaf(T->rchild);
    }
}// 调用结束，即可由n获得二叉树T的叶子结点数目
```

需注意每次调用前须执行n=0;

例2 统计二叉树中叶子结点的数目(3)

【算法2 以函数返回值返回】

```
// 函数值为T的叶子结点数
int   leaf(BiTree T)
{
    // 利用二叉树的中序遍历, n为局部变量
    n = 0;
    if ( T != NULL ){
        n = leaf ( T->lchild );
        // 访问结点->叶子结点的判定和计数
        if ( T->lchild==NULL && T->rchild==NULL )n++;
        n += leaf(T->rchild);
    }
    return (n);
}
```

例2 统计二叉树中叶子结点的数目(4-1)

【算法3 通过引用参数返回】

```
// 引用参数n为T的叶子结点数，方法一
void leaf(BiTree T, int &n)
{
    // 利用二叉树的后序遍历
    n = 0;
    if ( T != NULL ){
        leaf (T->lchild, n1);
        leaf (T->rchild, n2);
        // 访问结点->叶子结点的判定和计数
        if ( T->lchild==NULL && T->rchild==NULL )n++;
        n = n1+n2;
    }
}
```

例2 统计二叉树中叶子结点的数目(4-2)

【算法3 通过引用参数返回】

```
// 引用参数n等同于全局变量，方法二
// 把T所指向的二叉树中的叶子结点数累加到n
// 注意：在调用leaf(T,n)之前要先执行"n＝0;"
void leaf(BiTree T, int &n)
{
    // 利用二叉树的后序遍历
    if ( T != NULL ){
        leaf (T->lchild, n);
        leaf (T->rchild, n);
        // 访问结点->叶子结点的判定和计数
        if ( T->lchild==NULL && T->rchild==NULL )n++;
    }
}
```

为什么强调使用函数调用的结果？(1)

- ——算法/程序的健壮性！
  - 对于每算法/程序的任何输入，期望不异常终止
  - 养成一种编程的习惯！
- 例1
  - 接口

```
// 判断栈是否为空，返回TRUE/FALSE
Status StackEmpty(Stack S);
// 出栈（蕴涵对栈是否为空的判断）
// 若栈为空，返回ERROR，否则返回OK
Status Pop(Stack &S, ElemType &e);
```


为什么强调使用函数调用的结果？(2)

- 基于出栈的应用问题

```
// 方案1
if (StackEmpty(S) )
      …              //栈为空的一些处理
else {
      Pop(S, e);     //出栈---出栈前栈一定非空
      …e…
}
```

说明：此处不用Pop(S, e)的返回值，是因为它的返回值在该上下文下是唯一的，即永远为真

为什么强调使用函数调用的结果？(3)

- 基于出栈的应用问题

```
// 方案2
if ( Pop(S, e) ==ERROR)
      …      //栈为空的一些处理
else {
      …e… //栈不为空，使用出栈获得的元素
}
```

说明：此处直接使用Pop(S, e)的返回值，因为该操作蕴涵对栈为空的判断。在调用它后，其返回值有OK和ERROR两种可能！

为什么强调使用函数调用的结果？(4)

- 例2 printf( )的使用
  - 接口
    ```
    int printf( const char *format [, argument]... );
    ```
    Returns the number of characters printed, or a negative value if an error occurs.
    (from MSDN—Microsoft Developer Network)
  - 通常的使用
    printf("Hello, world!\n")
    Why? 一般的应用不关心输出的字符数

为什么强调使用函数调用的结果？(5)

- 例3 教材P131 算法6.4

```
Status CreateBiTree(BiTree &T){
  …
  if ( !(T=(…)malloc(…) ) ) exit(OVERFLOW);
  CreateBiTree(T->lchild);
  CreateBiTree(T->rchild);
  …
}
```

Why?
1) malloc失败，调用exit退出程序的执行
2) 返回只有一种值：OK →
　调用CreateBiTree时无须判断！

例3 释放二叉树的所有结点空间

【思路】
·二叉树为空时，不必释放；
·若T不为空，则先释放其左右子树的所有结点的空间，再释放根结点的空间——后序。
若在释放子树的空间前，先释放根结点的空间，则需要将子树的根结点的指针暂存到其他变量；否则，无法找到子树。
【算法】

```
void deleteBiTree(BiTree &T){   // 此处T应为引用参数
    if ( T != NULL ){
        deleteBiTree(T->lchild );
        deleteBiTree(T->rchild );
        // 访问结点->释放结点的空间
        free(T);
        T = NULL;
    }
}
```

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树 (1)

【本例特征】
如何选择二叉树的先序、中序、后序遍历来解决问题，它们对问题求解有何影响？
【思路】
整个过程分为两个方面：
- 遍历中查找元素值为x的结点
- 查到该结点时，调用例3的算法释放子树空间。

需要考虑的问题是：
- 如何将全部的结点找到并释放？
- 外层查找采用的遍历次序对本算法有何影响？

从以下3个算法中可以看出，利用先序遍历是最合适的；中序和后序，存在一定的多余操作。

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树 (2)

【算法1】

```
void deleteXTree(BiTree &T, ElemType x)
{
    // 基于先序的查找
    if ( T != NULL ){
        // 访问结点->判断是否为指定结点->释放树空间
        if ( T->data== x ) deleteBiTree(T);
        else{
            // 此处else不能省略
            deleteXTree(T->lchild, x);
            deleteXTree(T->rchild, x);
        }
    }
}
```

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树 (3)

【算法2】

```
void deleteXTree(BiTree &T, ElemType x)
{
    // 基于中序的查找
    if ( T != NULL ){
        // 若T->data== x，则此步骤多余
        deleteXTree(T->lchild, x);
        // 访问结点->判断是否为指定结点->释放树空间
        if ( T->data== x)
            deleteBiTree(T);
        else
            deleteXTree(T->rchild, x);
    }
}
```

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树（4）

【算法3】

```
void deleteXTree(BiTree &T, ElemType x)
{
    // 基于后序的查找
    if ( T != NULL ){
        // 若T->data== x，则此步骤多余
        deleteXTree(T->lchild, x);
        // 若T->data== x，则此步骤多余
        deleteXTree(T->rchild, x);
        // 访问结点->判断是否为指定结点->释放树空间
        if ( T->data == x ) deleteBiTree(T);
    }
}
```

例5 求位于二叉树先序序列中第k个位置的结点的值（1）

【本例特征】

如何处理多个返回结果？

【思路】

·待查结点的存在性：

当二叉树为空树，或者k非法时，待查结点不存在

➔ 函数应返回待查结点是否存在的状态指示：TRUE或FALSE

·当待查找的结点存在时，需进一步返回该结点的值

问题1：该算法需要返回多个值，如何处理？

答：一种做法是用返回值返回存在性，用变参返回值。

问题2：该算法可以基于二叉树的先序遍历的递归算法来构造，如何知道当前访问的结点是先序序列中的第几个结点？

答：引入计数器，对于该计数器可以采用全局变量来存储，也可以通过变参来处理。

例5 求位于二叉树先序序列中第k个位置的结点的值（2）

【算法】

```
Status PreorderKnode(BiTree T, int k, ElemType &e, int &count){
// 输入：T为二叉链表示的二叉树，k为待查找的结点在先序序列中的位序
// 输出：TRUE：待查找的结点存在；FALSE：待查找的结点不存在
//      e—当待查结点存在时，该结点的值通过e带回
//      中间变量：count—记录当前已经访问过的结点个数
  if ( T==NULL) return FALSE;
  count++;                        // 访问结点→对已访问的结点进行计数
  if (count==k ){                 //       →判断该结点是否是待查找的结点
     e = T->data;
     return TRUE;        // 查到，则设置e，并返回TRUE
  }else if (count > k)
     return FALSE; // 计数器count已经超出k(当k<0时)，则直接返回FALSE
  else {
     if (PreorderKnode(T->lchild, k, e, count)==FALSE) // 在左子树中查找
        return PreorderKnode(T->rchild, k, e, count);  // 在右子树中查找
     return TRUE;
  }
}
```

6.3 –先序遍历的非递归算法(1)

- 思路

假设：T指向要遍历的二叉树的根，若T != NULL

对于非递归算法，引入栈模拟递归工作栈，初始栈为空。

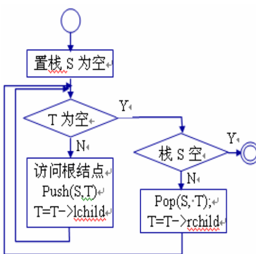
问题：如何用栈来保存信息，使得在先序遍历过左子树后，能利用栈顶信息获取T的右子树的根指针？

方法1：访问T->data后，将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，再先序遍历T的右子树。

方法2：访问T->data后，将T->rchild入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T->rchild，出栈，遍历以该指针指向的结点为根的子树。

6.3 –先序遍历的非递归算法(2)

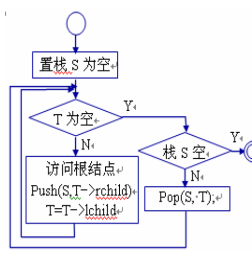
- 算法1




```
void PreOrder(BiTree T, Status (*Visit) (ElemType e))
{  // 基于方法一，流程图如右，当型循环
   InitStack(S);
   while (T!=NULL || !StackEmpty(S)){
      while (T != NULL){
         Visit(T->data) ;
         Push(S,T);
         T = T->lchild;
      }
      if( !StackEmpty(S) ){
         Pop(S,T);
         T = T->rchild;
      }
   }
}
```


6.3 –先序遍历的非递归算法(3)

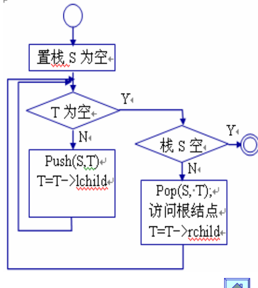
- 算法2




```
void PreOrder(BiTree T, Status (*Visit) (ElemType e))
{  // 基于方法二，流程图如右，当型循环
   InitStack(S);
   while (T!=NULL || !StackEmpty(S) ){
      while (T != NULL ){
         Visit(T->data);
         Push(S, T->rchild);
         T = T->lchild;
      }
      if ( !StackEmpty(S) ){
         Pop(S,T);
      }
   }
}
```

6.3 –中序遍历的非递归算法

- 思路
  - T是指向要遍历的二叉树的根，中序遍历要求在遍历完左子树后，访问根，再遍历右子树。
  - **问题**：如何用栈来保存信息，使得在中序遍历过左子树后，能利用栈顶信息获取T指针？
  - **方法**：先将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，访问T->data，再中序遍历T的右子树。

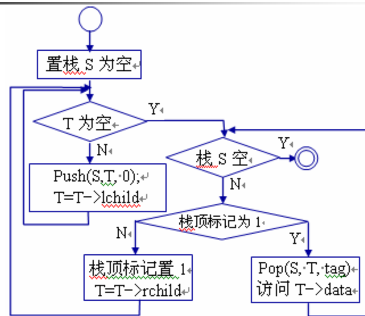


6.3 –后序遍历的非递归算法(1)

- 思路
  - T指向要遍历的二叉树的根，后序遍历要求在遍历完右子树后，再访问根。需要判断根结点的左右子树是否均遍历过。
  - 可采用标记法，结点入栈时，配一个标志tag一同入栈(0：遍历左子树前的现场保护，1：遍历右子树前的现场保护)。
  - 首先将T和tag(为0)入栈，遍历左子树；返回后，修改栈顶tag为1，遍历右子树；最后访问根结点。

```
typedef struct
stackElement{
    Bitreedata;
    char  tag;
}stackElemType;
```


6.3 –后序遍历的非递归算法(2)



6.3 –层次遍历算法及其应用

- 层次遍历算法及其应用
  - 二叉树的层次遍历算法
    - 先访问的结点，其孩子结点必然也先访问——队列
  - 基于层次遍历的应用
    - 判断一棵二叉树是否为完全二叉树
    - 找出距指定结点最近或最远的叶子子孙
    - 找出指定层中含有的叶子/度为2/度为1的结点

例6 二叉树的层次遍历(1)

【思路】

先访问的结点，其孩子结点必然也先访问。引入队列存储已被访问的、但其左右孩子尚未访问的结点的指针。若使用链队列，其元素可定义为：

```
typedef struct QNode{
    Bitree          data;          // 指向二叉树结点的指针
    struct QNode    *next;
} QNode, *QueuePtr;
```

【算法】

```
StatusLevelTraverse (BiTree T, Status ( *Visit ) (ElemType e) )
{
    // 初始化队列，队列的元素为二叉树的结点指针
    InitQueue(Q);
    if ( T != NULL ){
        // 访问根
        if ( !Visit( T->data ) ) return ERROR;
        // EnQueue( )为入队函数，T为待入队的元素
        EnQueue(Q, T);
```

例6 二叉树的层次遍历(2)

```
        // 从队列中取已被访问的、但其左右孩子尚未访问的结点指针
        // 访问其孩子
        while( !QueueEmpty(Q) ){ // 队不为空,则尚有未访问其孩子的结点
            // DeQueue( )为出队函数，它将原队头元素作为返回值返回
            p = DeQueue (Q);
            // 访问左孩子
            if ( p->lchild != NULL ) {
                if ( !Visit( p->lchild ->data ))     return ERROR;
                EnQueue(Q, p->lchild );
            }
            // 访问右孩子
            if( p->rchild != NULL ) {
                if ( !Visit( p->rchild ->data ))     return ERROR;
                EnQueue(Q, p->rchild );
            }
        }
    }
    return OK;
}
```

例7 判断一棵二叉树是否为完全二叉树(1)

【思路】
由完全二叉树的定义，对完全二叉树按层次遍历应满足：
1）若某结点没有左孩子，则它一定无右孩子；
2）若某结点缺孩子，则其后继必无孩子。
利用层次遍历，需要附加一个标志量bFlag反映是否已扫描过的结点均有左右孩子。在第一次遇到缺孩子的结点时，可将bFlag置为FALSE；此时存在以下两种情况：
·若它满足1），需要继续扫描后继结点，以判断是否满足2）；
·若不满足1），则说明不是完全二叉树。
【算法】

```
typedef     char          BOOL;
#define     TRUE   1
#define     FALSE  0
BOOL        JudgeCBT(BiTree T)
{
    InitQueue (Q);     bFlag = TRUE;
   if ( T != NULL ){
      EnQueue(Q, T);
```

例7 判断一棵二叉树是否为完全二叉树(2)

```
  while ( !QueueEmpty(Q) ){ // 队不为空，则尚有未访问其孩子的结点
    p = DeQueue(Q);
    if ( bFlag == FALSE ) {
      // 前驱结点缺左孩子，则若该结点有孩子，则此树非完全二叉树
      if ( p->lchild!=NULL || p->rchild!=NULL )    return FALSE;
    }else if ( p->lchild == NULL ) {
      // 第一次遇到结点缺左孩子
      bFlag = FALSE;
      // 若有右孩子，不满足1），则非完全二叉树
      if ( p->rchild != NULL ) return FALSE;
    }else if ( p->lchild != NULL ) {
      // 若有左孩子，则入队
      EnQueue(Q, p->lchild );
      if ( p->rchild == NULL )
        // 第一次遇到结点有左孩子，缺右孩子
        bFlag = FALSE;
      else    EnQueue(Q, p->rchild );
    }
  }
}
return TRUE;  // T为空，也是完全二叉树
}
```

6.3 –层次遍历算法及其应用

- 灵活运用：问题的延伸与解决
  - 找出距结点 x 最近的叶子子孙
  - 求距结点 x 最近的叶子子孙到 x 的距离
    如何获得距离信息？
    - 方法1：扩展队列元素的类型，增加距离域
    - 方法2：交替使用两个队列，以获得结点的层次
    - 方法3：在每一层末尾增加一个虚结点，…
  - 统计距结点 x 最近的叶子子孙的数目
  - 输出距结点 x 最近的所有叶子子孙
  - 输出距结点 x 最近的叶子子孙到 x 的路径
    - 出队时不删除元素，元素类型中增加结点的双亲在队列中的位置域

6.3 –二叉树的创建方法

- 二叉树(链式存储)的创建方法
  输入序列与二叉树的映射关系：
  - 完全二叉树的顺序映射
    通过补虚结点，将一般的二叉树转变成完全二叉树，再对该完全二叉树的结点按自上而下、自左至右进行输入。
  - 二叉树的先序遍历
    通过补虚结点，使二叉树中各实际结点均具有左右孩子，再对该二叉树按先序遍历进行输入。
  - 二叉树的两种遍历序列：先序+中序，后序+中序

6.3 –线索二叉树

- 线索二叉树——二叉树的结构扩展
  - 二叉树的遍历：非线性结构的线性化
  - 问题：在二叉树的链式存储中，如何快速找到某结点在某一序列(先序/后序/中序)中的直接前驱和直接后继？
    - 为每一结点增加fwd和bkwd指针域
      ——降低存储密度
    - 利用二叉链中的空链域(n个结点有n+1个空链域)
      ——标识链域的含义：孩子？线索(指向结点前驱和后继的指针)？

6.3 –线索二叉树类型定义

- 线索链表：包含线索的链表

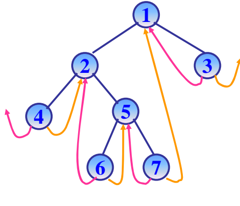
```
// Link== 0 :指针, Thread==1: 线索
typedef  enum { Link, Thread} PointerTag;
typedef struct BiThrNode{
  ElemType    data;
  // 左右孩子指针/线索
  struct BiThrNode *lchild, *rchild;
  // 左右标志
  PointerTag  ltag, rtag;
}BiThrNode, *BiThrTree;
```

- 线索二叉树：加上线索的二叉树
- 线索化：对二叉树以某种次序遍历使其变为线索二叉树的过程。

6.3 –中序线索二叉树(1)

- 中序线索二叉树——结构对称
  - 在中序线索二叉树上找结点的(直接)后继/前驱？

第一个结点：最左下的结点；
最后一个结点：最右下的结点
某结点的后继：
a) 若该结点有右孩子，其后继为其右子树中最左下的结点；
b) 若该结点无右孩子，其后继由rchild指向。
其后继为满足以下条件的最小子树的根r：该结点为r的左子树中最右下的结点。



6.3 –中序线索二叉树(2)

- 在中序线索二叉树上遍历

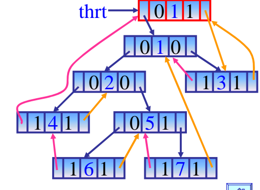
☑在二叉树的线索链表上添加一个头结点，令其lchild域指向二叉树的根—Link，rchild域指向中序遍历时访问的最后一个结点—Thread；中序序列中的第一个结点的lchild域和最后一个结点的rchild域均指向头结点。

算法见教材P134 算法6.5

- 二叉树的中序线索化

二叉树中序遍历算法的应用

算法见教材P134 算法6.6



6.3 –中序线索二叉树(3)

	中序遍历 InOrderTraverse	中序线索化二叉树 InThreading
输入	二叉链表示的二叉树的头指针 T	线索二叉链表示的二叉树的头指针 p
输入的表现方式	参数	参数
输出	对结点的访问结果	修改 p 指向的结点的链域
输出的表现方式	由输出设备输出	参数
空树（递归结束）的条件及处理(直接求解)	T＝＝NULL	p＝＝NULL
	空	空
左子树(使用递归调用的解)	InOrderTraverse(T->lchild)	InThreading(p->lchild)

6.3 –中序线索二叉树(4)

	中序遍历 InOrderTraverse	中序线索化二叉树 InThreading

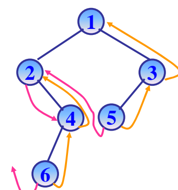
根结点的访问(子问题直接求解)	Visit(T->data)	if (p->lchild== NULL) {
		  p->ltag = Thread;
		  p->lchild = pre;
		}
		if (pre->rchild== NULL) {
		  pre->rtag = Thread;
		  pre->rchild = p;
		}
		pre = p;

右子树(使用递归调用的解)	PreOrderTraverse(T->rchild)	InThreading(p->rchild)

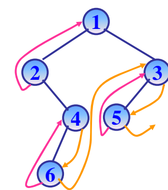
6.3 –后序线索二叉树 vs 先序线索二叉树(1)

- 后序线索二叉树 与 先序线索二叉树 互为对称
  - 后序线索二叉树

第一个结点：未必是树中最左下的结点，但一定是叶结点；
最后一个结点：根结点



后序线索二叉树

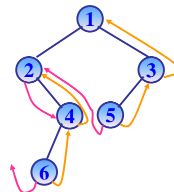


先序线索二叉树

6.3 –后序线索二叉树 vs 先序线索二叉树(2)

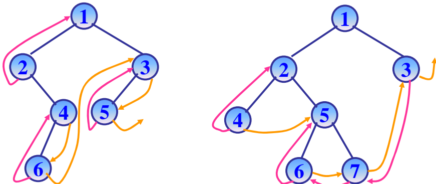
- 如何在后序线索二叉树上找结点的前驱？

a) 若该结点无左孩子，则其前驱由lchild指向。
b) 若该结点有左孩子且有右孩子，则其前驱为其右孩子；
c) 若该结点有左孩子且无右孩子，则其前驱为其左孩子。



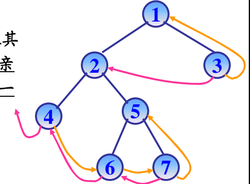
6.3 –后序线索二叉树 vs 先序线索二叉树(3)

- 如何在先序线索二叉树上找结点的后继?
  - a) 若该结点无右孩子，则其后继由rchild指向。
  - b) 若该结点有右孩子且有左孩子，则其后继为其左孩子;
  - c) 若该结点有右孩子且无左孩子，则其后继为其右孩子。



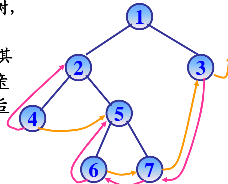
6.3 –后序线索二叉树 vs 先序线索二叉树(4)

- 如何在后序线索二叉树上找结点的后继? (复杂! ☹)
  - a) 若该结点是二叉树的根，则其后继为空;
  - b) 若该结点是其双亲的右孩子或是其双亲的左孩子且其双亲没有右子树，则其后继为其双亲;
  - c) 若该结点是其双亲的左孩子且其双亲有右子树，则其后继为其双亲的右子树中的后序遍历列出的第一个结点。
  - 涉及求结点的双亲:
  - 二叉链表→三叉链表



6.3 –后序线索二叉树 vs 先序线索二叉树(5)

- 如何在先序线索二叉树上找结点的前驱? (复杂! ☹)
  - a) 若该结点是二叉树的根，则其前驱为空;
  - b) 若该结点是其双亲的左孩子或是其双亲的右孩子且其双亲没有左子树，则其前驱为其双亲;
  - c) 若该结点是其双亲的右孩子且其双亲有左子树，则其前驱为其双亲的左子树中的先序遍历列出的最后一个结点。
  - 涉及求结点的双亲:
  - 二叉链表→三叉链表



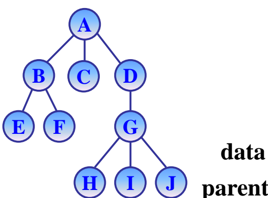
6.4 树和森林

- 树的存储结构
- 森林与二叉树的转换
- 树和森林的遍历

6.4 树和森林-树的存储结构(1)

- 树的存储结构
  - 双亲表示法

```
#define  MAX_TREE_SIZE  100
typedef  struct {
    Elemtype data;
    int parent;
} PTNode;
typedef struct {
    PTNode nodes[MAX_TREE_SIZE];
    int     n; // 结点数
}PTree;
```

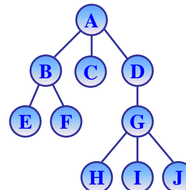


	0	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I	J
parent	-1	0	0	0	1	1	3	6	6	6

6.4 树和森林-树的存储结构(2)

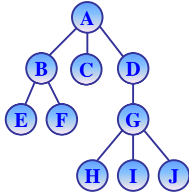
- 树的存储结构
  - 孩子表示法

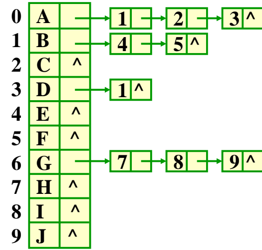
```
#define  MAX_TREE_SIZE  100
typedef  struct CTNode{
    int     child;
    struct CTNode *next;
} *ChildPtr;
typedef struct{
    TElemType      data;
    //孩子链表的头指针
    ChildPtr        firstchild;
}CTBox;
typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int      n, r; // 结点数和根的位置
}CTree;
```



6.4 树和森林-树的存储结构(3)

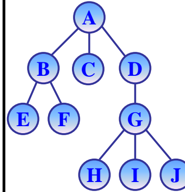
- 树的存储结构
  - 孩子表示法





6.4 树和森林-树的存储结构(4)

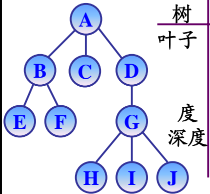
- 树的存储结构
  - 孩子兄弟表示法



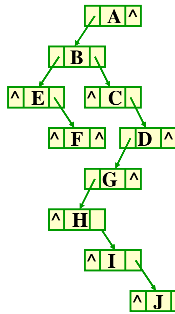

```
typedef struct CSNode{
   ElemType      data;
   struct CSNode  *firstchild;  //第一个孩子指针
   struct CSNode *nextsibling; //下一个兄弟指针
}CSNode, *CSTree;
```

6.4 树和森林-树的存储结构(5)

- 树的存储结构
  - 孩子兄弟表示法

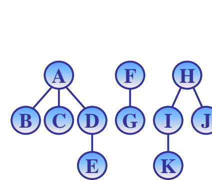


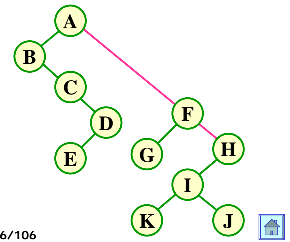
树	孩子兄弟链
叶子	孩子域为空
	无右子树
度	沿兄弟域来统计
深度	沿孩子域来统计



6.4 树和森林-森林与二叉树的转换

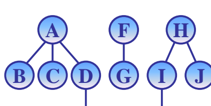
- 森林与二叉树的转换
  森林的孩子兄弟链⇔二叉树的二叉链





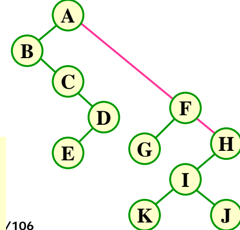
6.4 树和森林-树(森林)的遍历

- 树(森林)的遍历
  - 先根遍历 ⇔对应二叉树的先序遍历
  - 后根遍历 ⇔对应二叉树的中序遍历



先根：ABCDEFGHIKJ

后根：BCEDAGFKIJH



6.4 树和森林-树(森林)的遍历应用

- 树(森林)的遍历的应用——基于二叉链
  - 统计树(森林)中叶子结点的个数
  - 求树的高度
  - 求树的度

例8 统计树(森林)中叶子结点的个数

【思路】
采用孩子-兄弟表示法(二叉链)表示树或森林;
树或森林中叶子结点的特征：二叉链中结点的firstchild为空
统计叶子结点数→在遍历的过程中统计firstchild域为空的结点数。
【算法】
```
// n为叶子结点的计数器
int    n=0;
void   CSleaf(CSTree T) {
    // 利用树的先序遍历
    if ( T != NULL ){
        // 访问结点→叶子结点的判定和计数
        if ( T->firstchild == NULL )    n++;
        CSleaf(T->firstchild);
        CSleaf(T->nextsibling);
    }
}// 结束，即可由n获得树T的叶子结点数目，注意下次调用前须n = 0;
```

例9 求树的高度

【思路】
采用孩子-兄弟表示法(二叉链)表示树
树的高度 = max(相应左子树的高度+1，右子树的高度)。
【算法】
```
// 引用参数h为T的高度
void   CShigh(CSTree T, int &h){
    // 利用二叉链表的后序遍历
    if ( T == NULL )    h = 0;
    else {
        CShigh(T->firstchild, h);
        CShigh(T->nextsibling, h1);
        h = max(h+1, h1);
    }
}
```

例10 求树的度

【思路】
采用孩子-兄弟表示法(二叉链)表示树；访问结点→求结点的度
【算法】
```
// degree表示表示树的度
void   CSDegree(CSTree T, int &degree){
    // 利用二叉树的先序遍历，d表示T指向的结点的度
    if ( T == NULL )    degree = 0;
    else{
        if ( T->firstchild == NULL )  d = 0;
        else{
            d = 1; p = T->firstchild;
            while ( p->nextsibling != NULL ){
                p = p->nextsibling; d++;
            }
        }
        CSDegree (T->firstchild, d1); CSDegree (T->nextsibling, d2 );
        degree = max (d1, d2, d );
    }
}
```

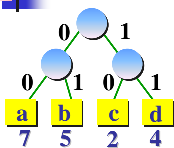
6.6 赫夫曼树及其应用

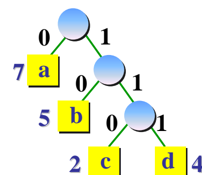
- 最优树
  - 路径：从树中一个结点到另一个结点之间的分支
  - 路径长度：路径上的分支数目
  - 树的路径长度：树根到每一结点的路径长度之和
    完全二叉树是这种路径长度最短的二叉树
  - 结点的带权路径长度：该结点到树根之间的路径长度与结点上权的乘积
  - 树的带权路径长度：树中…度之和

    WPL: Weighted Path Length of Tree

    $$WPL = \sum_{i=1}^{n} w_i l_i$$
  - 最优二叉树或赫夫曼树：树的带权路径长度最小的二叉树

6.6 -不同带权路径长度的二叉树



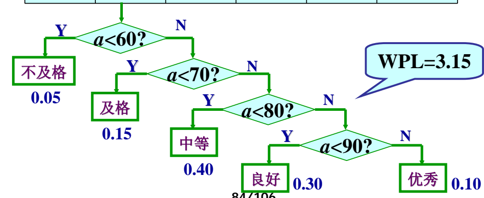


WPL = 2*(7+5+2+4)=36   WPL=1*7+2*5+3*(2+4)=35

6.6 –赫夫曼树的应用(1)

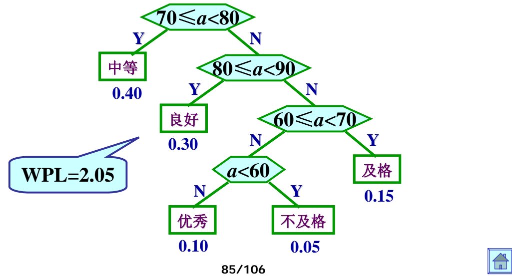
- 赫夫曼树的应用——最佳判定算法

分数	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10
等级	不及格	及格	中等	良好	优秀



6.6 –赫夫曼树的应用(2)

- 赫夫曼树的应用——最佳判定算法



6.6 –赫夫曼算法

- 赫夫曼算法
  1) 根据给定的n个权值构成n棵二叉树的集合F，其中每棵二叉树中只有一个带权值的结点；
  2) 在F中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和；
  3) 在F中删除这两棵树，同时将新得到的二叉树加入到F中；
  4) 重复2)和3)，直到F中只含一棵树为止。

6.6 –赫夫曼树的特点

- 赫夫曼树(最优二叉树)的特点
  - 不存在度为1的结点(这类树又称严格的或正则的二叉树)，为什么？
    若有n个权值，则形成的赫夫曼树中有$2n$-1个结点
  - 赫夫曼树的高度 h 与结点数 m 之间的关系
    若赫夫曼树的高度为 h (h >0)，则树中
    最多有 2^h-1 个结点→满二叉树
    最少有$2h$-1个结点

6.6 –赫夫曼编码(1)

- 赫夫曼编码
  - 问题描述：已知电文中出现的一组字符及出现频率，试为该组字符编码以使得电文总长最短。
  - 前缀编码：某字符的编码不是其他字符编码的前缀。
  - 赫夫曼编码过程：
    1) 由n个权值，形成包含有$2n$-1个结点的赫夫曼树
    2) 约定赫夫曼树的左分支表示字符'0'，右分支表示字符'1'，则从根结点到叶子结点的路径上的分支字符组成的字符串作为该叶子结点字符的编码。

6.6 –赫夫曼编码(2)

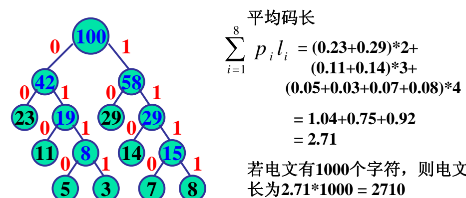
- 赫夫曼编码
  - 赫夫曼树和赫夫曼编码的存储结构选取
    - 由权值数n唯一确定树中的结点数为 ($2n$ - 1)
      →一次性分配结点空间
    - 编码：从叶子到根的路径
      →快速访问双亲
    - 译码：从根到叶子的路径
      →快速访问孩子

    赫夫曼编码：按实际长度动态分配空间
    需要一个求编码的工作空间，长度为n

赫夫曼树：
三叉静态链表

6.6 –算法与例子

- 赫夫曼编码的算法
  教材P147 算法6.12
- 教材P148 例6-2



平均码长

$$\sum_{i=1}^{8} p_i l_i = (0.23+0.29)*2+(0.11+0.14)*3+(0.05+0.03+0.07+0.08)*4$$

$$= 1.04+0.75+0.92 = 2.71$$

若电文有1000个字符，则电文的码长为2.71*1000 = 2710

6.5 6.7 6.8 开拓问题求解的思路

6.5 树与等价问题

6.7 回溯法与树的遍历

6.8 树的计数

6.5 树与等价问题(1)

- 等价关系和等价类
  - 如果集合S中的关系R是自反的、对称的和传递的，则称它为一个等价关系。
  - 设R是集合S的等价关系。
    对任何$x \in S$,由$[x]_R=\{ y \mid y \in S \wedge xRy\}$给出的集合$[x]_R \subseteq S$称为由$x \in S$生成的一个$R$等价类。

6.5 树与等价问题(2)

- 如何划分等价类?
  - 问题描述：设集合S有n个元素，m个形如(x, y) $(x,y \in S)$的等价偶对确定了等价关系R，现需要求S的划分。
  - 算法思想
    - S中每个元素各自形成一个只含单个成员的子集
    - 对于输入的每一偶对(x, y)，判定x和y所属子集S_i和S_j，若$S_i \neq S_j$，则将S_i并入S_j并置S_i为空(或将S_j并入S_i并置S_j为空)

6.5 树与等价问题(3)

- 划分等价类需对集合进行的操作
  - 构造只含单个成员的集合;
  - 判定某个单元素所在子集;
  - 归并两个互不相交的集合为一个集合。
  - ➔抽象数据类型MFSet（见教材P140)
  - Initial(&S, n, x1, x2, …, xn)
  - Find(S, x)
  - Merge(&S, i, j)

6.5 树与等价问题(4)

- 集合的表示
  - 位向量、有序表等
  - MFSet: 树型结构
    根据其Find(S, x)和Merge(&S, i, j)操作特点
    - 属于一个集合中的元素在同一棵树中
    - 用森林表示多个集合
    - 存储结构：双亲表示法，根结点的成员兼作集合的名称

6.5 树与等价问题(5)

- MFSet: 树型结构
  - Find(S, x)：由x顺着parent域查找到根
  - Merge(&S, i, j)：将一棵树的根结点作为另一棵树的根结点的双亲
  教材P141 算法6.8和算法6.9
- 算法6.8和6.9 的时间复杂度分别为$O(d)$和$O(1)$
  d—树的深度 (d与树的形成有关。最坏情况下，包含有n个结点的集合树深度为n)

6.5 树与等价问题(6)

- 问题：如何降低树的深度？
  →改进Merge(&S, i, j)
- Merge(&S, i, j) 的改进思路一
  将成员少的子集树根结点指向含成员多的子集的根
  → 根结点的parent域存储子集中所含成员数目的负值
  算法见教材P142的算法6.10，按此算法得到的集合树，其深度不超过 $\lfloor \log_2 n \rfloor + 1$

6.5 树与等价问题(7)

- Merge(&S, i, j) 的改进思路二
  增加"路径压缩"的功能(当所查元素i 不在树的第1,2层)
  →将所有从根到元素i 路径上的元素都变成树根的孩子。
- 算法见教材P143的算法6.11，按此算法得到的集合树，其深度不超过$O(\alpha(n))$,
  $\alpha(n)$ 是一个增长极其缓慢的函数，$\alpha(n) \leq 4$

6.7 回溯法与树的遍历(1)

- 回溯(backtracking)法
  - 求解过程实质上是一个先序遍历一棵"状态树"的过程。
  - "状态树"隐含在遍历过程中，并未在遍历前预先建立起来。

6.7 回溯法与树的遍历(2)

- 第3章 递归与回溯——N皇后问题
  - 棋盘状态的变化 → 棋盘状态树(N叉树)
    结点：一个局部或完整的布局
    根结点：棋盘的初始状态—棋盘中无任何棋子
    四皇后问题的棋盘状态树见教材P151的图6.29
  - 求所有合法的布局：对该N叉树的先根遍历

6.7 回溯法与树的遍历(3)

- 求含n个元素的集合的幂集
  - 依次对集合中元素进行"取"或"舍" 的过程
  - n个元素→高度为n的满二叉树
    根结点：空集
    叶子：形成幂集中的一个元素
    左分支："取"　　右分支："舍"
  - 算法：教材P150 算法6.14
    若用线性表表示集合和幂集中的一个元素，则相应的算法见教材P150 算法6.15

6.8 树的计数(1)

- 树的计数问题
  具有n个结点的不同形态的树有多少棵？
- 二叉树T和T′相似
  - 二者均为空树，或
  - 二者均不为空树，且它们的左右子树分别相似
- 二叉树T和T′等价
  - 二者相似，且
  - 所有对应结点上的数据元素均相同

6.8 树的计数(2)

- 二叉树的计数问题

  具有n个结点、互不相似的二叉树的数目b_n

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} = \frac{1}{n+1} \cdot \frac{(2n)!}{n!n!} = \frac{1}{n+1} C_{2n}^{n} \quad n \geq 1 \end{cases}$$

- 二叉树的计数问题的另一种考虑角度
  - 依据：已知二叉树的结点的先序序列和中序序列，能唯一确定一棵二叉树。（为什么？）

6.8 树的计数(3)

- 假设：二叉树的 n 个结点从1到n 加以编号，且其先序序列为1,2,…, n
- 不同形态的二叉树的数目⇔先序序列均为1,2,…, n的二叉树所能得到的中序序列的数目，即为Catalan函数

  中序遍历的过程：一个结点进栈和出栈的过程

  (图6.33, P155)

6.8 树的计数(4)

二叉树的形态确定了其结点的进栈和出栈的顺序，也确定了其结点的中序序列。

$$C_{2n}^{n} - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^{n}$$

假设S表示入栈操作，X表示出栈操作

---Knuth的解释

- n个S，n个X的序列共有(2n,n)种可能；(2n,n)表示从2n个数中取n个数的可能的组合数

6.8 树的计数(5)

- 对于这样的序列从左到右进行扫描，遇到第一个X个数大于S个数的位置j，把从1到j这j个位置的所有X换成S、S换成X。这个序列就是一个n+1个S、n-1个X的序列
  - 这样的由一个n+1个S、n-1个X组成的序列，可以与n个S、n个X所组成的不合法的序列一一对应
  - 这样的序列共有(2n,n+1)中可能

6.8 树的计数(6)

- 故合法序列个数为

  (2n,n) -(2n,n+1)=(2n,n)/(n+1)

- 具有n 个结点的树的数目$t_n=b_{n-1}$

  依据：森林与二叉树的相互转换