第五章 树和二叉树

从本章开始，讨论更复杂的数据结构（非线性的数据结构）和其他问题

树形结构是一种十分重要的数据结构。本章讨论的二叉树、树和树林都属于树形结构

在树形结构中每个结点最多只有一个前驱，但可有多个后继的结构

其共同之处是都表示某种具有层次性的分支关系

5.1 树与树林

5.1.1 树 (Tree) 的定义

树的例子：家族树，机构的结构

假设 A 有子女 B, C； B 和 C 分别有子女 D, E, F 和 G, H; E有子女 I , J。

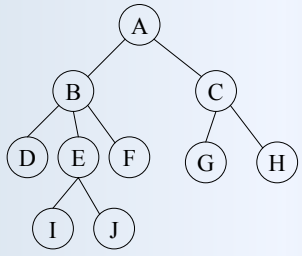
T = (N, R) ，其中：

N = {A, B, C, D, E, F, G, H, I, J}

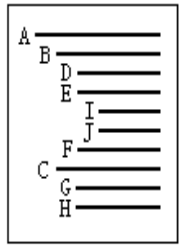
R = {< A, B>, < A, C>, < B, D> , < B, E>, < B, F>,
< C, G>, < C, H>, < E, I>, < E, J> }

关系有层次性，总是高层与低层相关，同层之间无关系，也没有低层到高层的关系。与不同元素相关的元素互不相交

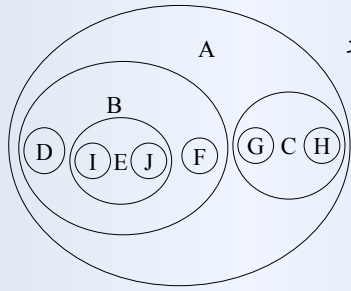
树的表示方法：



基本图形表示



凹入表



文氏图

(A (B (D) (E (I) (J))) (C (G) (H)))

嵌套括号表示法

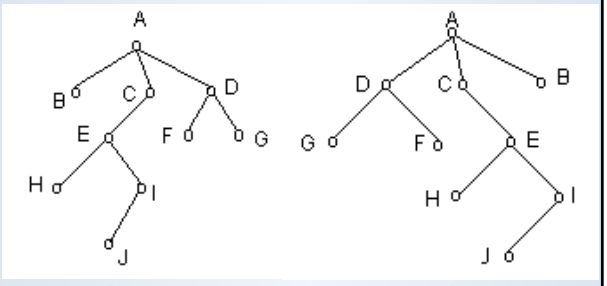
树的定义（递归定义）：

树是 n (n≥0) 个结点的有限集T。T非空时满足:

1. 有且仅有一个特殊的称为根的结点 r
2. 除根结点外，其余结点可分为m（m ≥ 0）个互不相交的非空有限集$T_1, T_2, \ldots, T_m$，其中每个集合又是一棵非空树，称为 r 的子树 (subtree)

- 结点数为 0 的树称为空树
- 一棵树也可以没有子树（m = 0），此时这棵树只包含一个根结点
- 如有子树，则子树非空（使子树的个数能够定义）

5.1.2 基本术语



(a) 树t　　(b) 树t ′

有序树和无序树：树中子树的顺序是否重要

父结点，子结点，边

兄弟结点

祖先，子孙

路径，路径长度

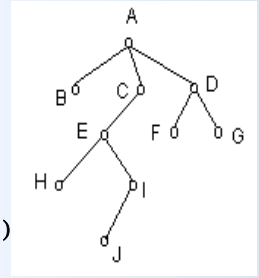
结点的层数（根的层为0）

深度或高度（结点的最大层数）

结点的度数、树的度数

树叶、分支结点

结点的顺序（最左，…，仅在考虑有序树时有意义）



5.1.3 树林

树林：m（m≥0）棵互不相交的树的集合

一棵非空树是一个二元组 Tree = (root, F)，其中

- root是树根结点
- F 是 m（m≥0）棵子树构成的树林
- $F = (T_1, T_2, \ldots, T_m)$。$T_i$称为根 root 的第 i 棵子树（有序树的情况。对无序树：$F = \{T_1, T_2, \ldots, T_m\}$）

注意树与树林的关系：

- 树由根和子树树林构成，树林由一集树组成

5.2 二叉树

二叉树是另一种树形结构：元素也有层次性，只有从上层元素到下一层元素的关联，与同一层的不同元素所关联的元素互不相交，每个元素只与下层的两个元素相关联

5.2.1 二叉树的基本概念

定义：

二叉树是结点的有限集合，该集合或为空集，或由一个根及两棵不相交的二叉树组成（递归定义）

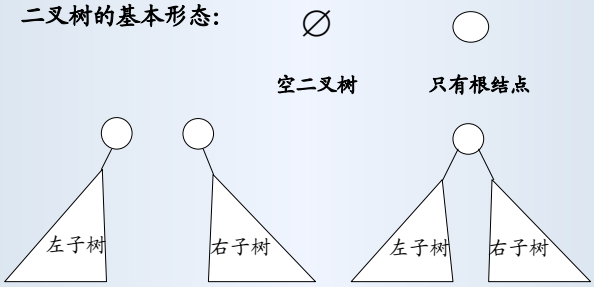
二叉树的两棵子树分别称作它的（其根的）“左子树”和“右子树”

二叉树的特点：

- 一个结点至多有两棵子树
- 子树有左右之分

二叉树是与树不同的结构（并不是树的特殊情况）

二叉树的基本形态：



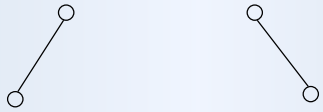
左子树非空
右子树空

左子树空
右子树非空

左、右子
树都不空

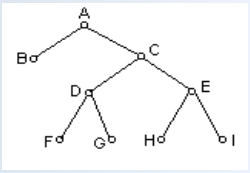
二叉树不是树的特殊情形，它们是两个不同概念。主要差别在于二叉树的子树分左子树和右子树，即使只有一棵子树也要明确指出是左子树还是右子树。

下面是两棵不同的二叉树，但作为树是相同的：



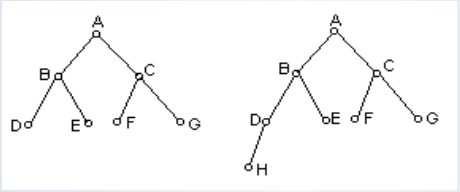
在二叉树中可定义与树中类似的各种概念：父/子结点，祖先/子孙，路径及其长度，结点的层数和度数，二叉树的高度，树叶和分支结点等等

下面介绍另外一些概念

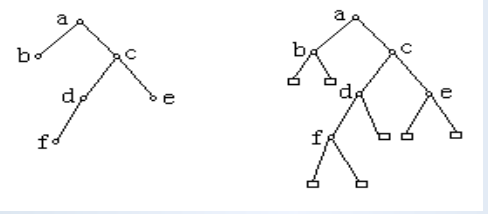


满二叉树：每个非叶结点都有两棵非空子树

完全二叉树：只有最下两层结点的度数可小于2，如果最下一层结点不满，则空位都集中在右边，左边没有空位



扩充二叉树　　扩充一些结点，使原树结点的度数都变成2



扩充二叉树里新增的外部结点个数比原内部结点的个数多1。

“外部路径长度”E：扩充二叉树里从根到各外部结点的路径长度之和。“内部路径长度”I：扩充二叉树里从根到各内部结点的路径长度之和。（n是内部结点的个数）

$$E = I + 2n$$

5.2.2 二叉树的一些性质

性质1. 非空二叉树第 i 层上至多有 2^i 个结点(i≥0)

性质2. 高度为 k 的二叉树至多有 $2^{k+1}-1$ 个结点(k ≥ 0)

性质3. 对任何非空二叉树T，若叶结点个数为 n_0，度数为 2 的结点个数为 n_2，则 $n_0 = n_2 + 1$

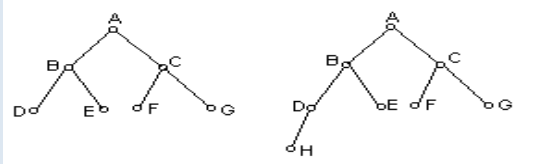
性质4. n 个结点的完全二叉树的高度 k 为 $\lfloor \log_2 n \rfloor$

性质5. 满二叉树里的叶结点比分支结点多一个

这些性质都很容易从空树或者只有一个结点的二叉树出发，通过归纳法证明

性质6.（完全二叉树）如果 n 个结点的完全二叉树按层次按顺序从1 开始编号，对任一结点 i（1 ≤ i≤n) 都有：

1. 序号1的结点是根；i > 1时其双亲结点是 $\lfloor i/2 \rfloor$
2. 若 2i ≤n，其左子女结点序号为2i。否则无左子女
3. 若 2i+1 ≤ n，其右子女结点序号为2i+1; 否则无右子女



A B C D E F G
1 2 3 4 5 6 7

A B C D E F G H
1 2 3 4 5 6 7 8

如果完全二叉树中的结点从 0 开始编号，也有类似性质：

1. 序号 0 的结点为根；
2. 非根结点 i 的父结点编号是⌊(i – 1)/2⌋；
3. 结点的两个子结点（若存在）的编号分别为 2i+1和2i+2。

完全二叉树的特点：

- 可以方便地存入一系列连续位置（存入一个数组）
- 不需要另外保存其他信息，直接根据数组下标就可以找到一个结点的子结点或者父结点

5.2.3 二叉树基本运算

- 创建空二叉树

BinTree createEmptyBinTree();

- 创建一棵二叉树，其根为r，左右子树分别是 l 和 r

BinTree consBinTree(BinTreeNode r, BinTree l, BinTree r);

- 判断二叉树是否为空

int isEmpty(BinTree t);

- 求二叉树的根结点，若为空，则返回一特殊值

BinTreeNode* root(BinTree t);

- 求二叉树中某个指定结点的父结点，当指定结点为根时，返回一特殊值

BinTreeNode* parent (BinTree t , BinTreeNode p);

- 求二叉树 t 中某指定结点 p 的左子女结点，当指定结点没有左子女时，返回一特殊值

BinTreeNode* leftChild (BinTree t , BinTreeNode* p)

- 求二叉树 t 中某个指定结点 p 的右子女结点，当指定结点没有右子女时，返回一特殊值

BinTreeNode* rightChild (BinTree t , BinTreeNode* p)

- 二叉树的周游

由于二叉树概念是递归定义的，二叉树中的每个结点可唯一标识以这个结点为根的二叉树，所以在具体实现时常常把二叉树类型和二叉树中结点类型看成是同一种类型

采用这种观点可能使某些算法的描述更方便

5.2.4 二叉树的周游

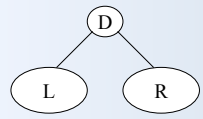
二叉树的周游(Traversing，遍历)：按某种顺序访问二叉树中所有结点，每个结点访问一次且仅一次

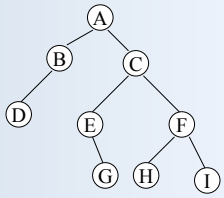
任何遍历所有结点的过程都是“周游”。由于是要实现算法，我们必须采取某种系统化的方式

同样有“深度优先方式”和“广度优先方式”

三种基本的深度优先周游方式：

先根序 (DLR)
对称序 (LDR)
后根序 (LRD)





二叉树

在各种周游方法中，如果遇到树空都立即结束

先根序（先访问根结点，而后以同样方式周游左右子树）

A B D C E G F H I

后根序（先以同样方式周游左右子树，而后访问根结点）

D B G E H I F C A

对称序（中根序）（先以同样方式周游左子树，而后访问根结点，最后再以同样方式周游右子树）

D B A E G C H F I

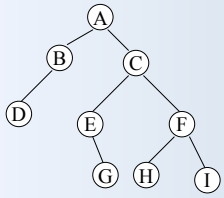
周游序列：

- 把按先根次序对一棵二叉树周游得到的线性结点（数据）序列称为这棵二叉树的先根序列
- 将按后根次序周游一棵二叉树得到的线性结点（数据）称为这棵二叉树的后根序列
- 将按对称次序周游一棵二叉树得到的线性结点（数据）称为这棵二叉树的对称（中根）序列

- 对给定二叉树，可唯一确定其先根序列、后根序列和对称序列。反过来，给定一个二叉树的任一种周游序列，无法唯一确定这个二叉树
- 如果已知某二叉树的对称序列，又知另一周游序列（无论先根序列还是后根序列），都可唯一确定这个二叉树

广度优先方式的周游

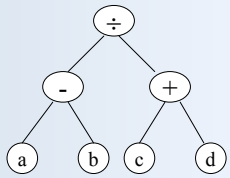
- 在二叉树中从 0 到树的高度逐层访问各层的结点
- 在每层里从左到右逐个访问结点
- 二叉树的广度优先周游又称层次序周游
- 按广度优先顺序访问形成的结点（数据）序列，称为相应二叉树的层次序列



二叉树

A B C D E F G H I

表达式可以自然地用二叉树表示：运算符是根，运算对象是其左右子树



表达式 (a – b) / (c + d)

的二叉树表示

对表达式树先根、后根和中根序周游得到的结点序列：

先根：÷ - a b + c d

前缀表示

后根：a b - c d + ÷

后缀表示 (波兰表示法)

对称序：a - b ÷ c + d

中缀表示

二叉树的周游算法

递归算法

- 先根序
- 中根序（对称序）
- 后根序

非递归算法 (自己用栈保存后面周游所需的信息)

- 先根序
- 中根序
- 后根序

现在讨论抽象周游算法，它们只依赖于二叉树抽象数据类型。有了具体表示后，很容易把它们变成具体算法

递归周游算法

周游操作的抽象规范：

先根次序周游： void preOrder(BinTree t);

中根次序周游： void inOrder(BinTree t);

后根次序周游： void postOrder(BinTree t);

```
/* 二叉树的先根序（先序）周游 */
void preOrder( BinTree p) {
   if (p==NULL) return;
   visit(root(p));
   preOrder(leftChild(p));
   preOrder(rightChild(p));
}
```



```
/* 二叉树的中根序（中序，对称序）周游 */
void inOrder(BinTree p) {
   if (p == NULL) return;
   inOrder(leftChild (p));
   visit(root(p));
   inOrder(rightChild (p));
}

/* 二叉树的后根序（后序）周游 */
void postOrder(BinTree p) {
   if (p == NULL) return;
   postOrder(leftChild (p));
   postOrder(rightChild (p));
   visit(root(p));
}
```


非递归的周游算法

利用栈的帮助，可以写出各种深度优先周游的非递归算法

利用队列的帮助，可以写出层次序的周游算法

下面给出的算法中并不涉及具体的存储结构，其中对栈或队列的使用，也不依赖于具体表示方式

在选择了具体的二叉树实现结构，可以对这里的算法做适当的改造，转变为针对具体二叉树实现的具体算法（还可能可以适当精化），就可以上机运行了

具体的算法实现中可以选择适当的栈或者队列实现方式

先根序周游的非递归算法

采用非递归算法实现先根次序周游二叉树的主要思路:

- 首先把根结点压入栈中
- 从栈中取出栈顶元素（并退栈）
  - 取出的元素非空时访问该结点，然后顺序将其右子结点和左子结点（若非空）进栈，重复
- 当栈为空时，周游结束。

算法中每个二叉树结点恰好进/出栈一次，所以其时间代价为O(n)，其中 n为二叉树中结点的个数


```
void norecPreOrder(BinTree  t) {
   Stack  s;    /*栈元素类型为 (BinTreeNode*) */
   BinTreeNode *c;
   if (t = = NULL) return;
   s = createEmptyStack();
   push(s, root(t));
   while (!isEmptyStack(s)) {  /* 栈不空则继续 */
      c = top(s);   pop(s);        /*取栈顶，退栈*/
      visit(c);                /*访问*/
      if (!isEmpty(rightChild(c))) push(s, rightChild(c));
      if (!isEmpty(leftChild(c))) push(s, leftChild(c));
   }
}
```

也可以在入栈时不检查，出栈时检查是否为空

中根序（对称序）周游的非递归算法

采用非递归算法实现对称序周游二叉树的主要思路:

- 当前二叉树不空或者栈不空时继续
  - 若二叉树不空时则沿其左子树前进，在前进过程中把所经过的结点逐个压入栈中
  - 当左子树为空时，弹出栈顶元素并访问这个结点
  - 如果这个结点有右子树，再进入它的右子树并从头执行上述过程；如果它没有右子树，则弹出栈顶元素，回到前面继续执行
- 到当前二叉树为空并且栈也为空时周游结束。


```
void norecInOrder(BinTree  t) {
   Stack s = createEmptyStack();
       /*栈元素为 (BinTreeNode*) */
   BinTreeNode*  c = root(t);
   while ( !isEmpty(c) || !isEmptyStack(s)) {
       while ( !isEmpty(c) )  {
          push(s, c);  c = leftChild(c);
       }
       c = top(s); pop(s);
       visit(c);
       c = rightChild(c);
   }
}
```


后根序周游的非递归算法

每个问题都可能有多种不同的算法，下面介绍本问题的一种算法（是比较简短的算法）

不变关系:

- 栈中结点是对二叉树的划分，左边是已周游部分
- 栈中每个结点的父结点总是它下面那个结点
- 当前结点的父结点是栈顶
- 根据本结点是其父结点的左子结点或右子结点，可以决定下一步怎么做

算法里的主要技术是“下行循环”：找到下一个应访问的结点


```
void norecPostOrder( PBinTree t ) {
   Stack st = createEmptyStack ( );
   BinTreeNode * p = root(t);
   while ( !isEmpty(p) || !isEmptyStack(st) ) {
      while (!isEmpty(p)) { /* 循环到栈顶结点的两子树都空 */
         push( st, p );
         p = leftChild(p);
         if (isEmpty(p)) p = rightChild(top(st));
      }
      p = top(st); pop(st); /* 栈顶是应访问结点 */
      visit(p); /* 当前结点已访问 */
      if ( !isEmptyStack(st) && leftChild(top(st)) == p )
         /* 栈不空且当前结点是栈顶的左子结点 */
         p = rightChild(top(st)) ;
      else p = NULL; /* 从右子树返回则强迫退栈 */
   }
}
```

书上另给出了两个算法，一个标记各结点访问到左子树还是右子树，第三个先把全部结点按后序的逆序压栈（压入根，后按同样方式压入右子树和左子树）。在弹出过程中逐个访问

三个算法的时间复杂性都是 O(n)，n 为二叉树结点数

第三个算法的空间复杂性（栈）与二叉树结点数成正比，另外两个算法空间复杂性与最长路径的长度成正比

最坏情况下两者具有同样数量级（有些二叉树里存在特别长的路径）。一般而言不是这样

通过对二叉树的计数分析（“图论”的工作），可知二叉树的平均结点个数 n 与最长路径的长度 L 的关系是:

$$n = O(2^L)$$

第三个算法（第一版）在空间复杂性方面有缺陷

层次序周游的非递归算法

根据广度优先周游的思想比较简单：

- 利用一个队列实现
- 首先把二叉树送入队列
- 其后，每当从队首取出一个二叉树访问之后，马上把它的子二叉树按从左到右的次序送入队列尾端
- 重复此过程直到队列为空


```
void levelOrder (BinTree  t) {
   BinTreeNode *c, *cc;
   Queue  q = createEmptyQueue();/*元素为(BinTreeNode*)*/
   if ( t== NULL)  return;              /*空二叉树返回*/
   c = root(t);  enQueue(q,c);          /*将二叉树送入队列*/
   while ( !isEmptyQueue(q) ) {
      c = frontQueue(q);  deQueue(q);
      visit( root( c ) );  /*从队列取结点并访问*/
      if ( !isEmpty(leftChild(c)) )  enQueue(q, leftChild(c));
      if ( !isEmpty(rightChild(c)) )  enQueue(q, rightChild(c));
    }
}
```


5.3 二叉树的存储表示

5.3.1 顺序表示

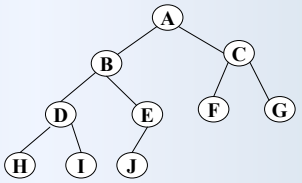
5.3.2 链接表示

5.3.3 二叉树的生成

5.3.4 线索二叉树

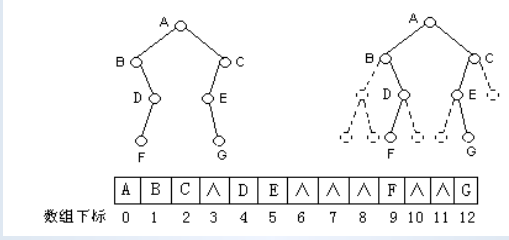
5.3.1 顺序表示

用一组连续存储单元按层次序存储完全二叉树的结点。可以方便地从子结点找父结点，从父结点找子结点。



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

一般二叉树可扩充到完全二叉树后存储入一维数组。



一般二叉树及其顺序表示

用顺序方式存储非完全二叉树时，数组里可能出现大量空位，因此可能浪费大量存储空间

二叉树顺序表示的类型声明：

```
typedef  struct SeqBTree  { /* 顺序树类型定义 */
   int   n; /* 改造成完全二叉树后的结点个数 */
   DataType  nodelist[MAXNODE];
} SeqBTree, *PSeqBTree;
```

nodelist数组元素表示结点。

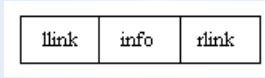
二叉树抽象数据类型的操作都可以映射到这里

如果用动态顺序表的实现方式，最大结点数可作为参数提供给二叉树的创建函数，数组溢出时可以通过程序扩充空间

5.3.2 链接表示

用链接表示来存储二叉树

- 每个树结点对应链接结构中的一个结点
- 二叉树是非线性结构，每个结点最多两个后继，最常用的链接表示方式是左-右指针表示法
- 结点中除存储结点数据外，还包含两个指针llink和rlink，分别存放本结点的左子结点和右子结点信息（通用指针）
- 当结点的某个子树为空时，相应的指针为空指针。



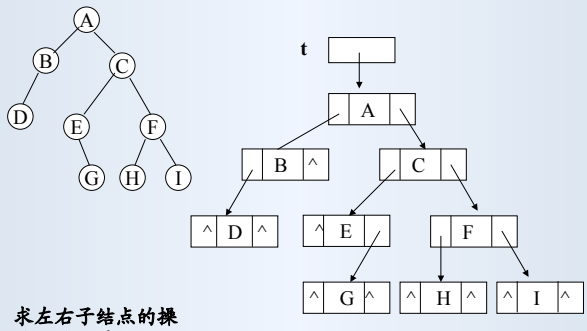
具体定义的结构和类型：

```
typedef struct BNode BNode, *PBNode; /* 结点和指针类型 */
typedef struct BNode *BinTree; /* 注意，和 PBNode 一样 */

struct BNode {/*二叉树结点 */
   DataType  info;   /* 数据域 */
   BinTree llink;    /* 指向左子女 */
   BinTree rlink;    /* 指向右子女 */
};
```

二叉树的许多处理可用递归算法描述，为方便，没对二叉树进行封装，直接将二叉树定义为指向结点的指针类型

有些操作可能要改二叉树的根，为方便描述，可定义：

```
typedef BinTree *PBinTree;
```

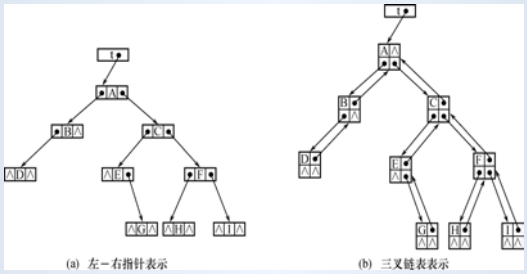



求左右子结点的操作极其简单，就是直接取结构成分

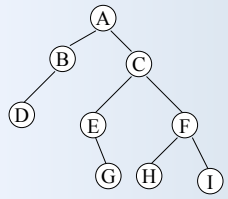
二叉树的链接表示

求父结点操作较难实现，只能从根周游，检查当前结点是否所求结点的父结点。最坏时间代价为 O(n)。

若求父结点的操作使用频繁，可采用三叉链表表示，加一个父结点指针。但这样也增加了空间开销



二叉树的构造



创建二叉树与表示方式有关。

用顺序表示时，应先扩充为完全二叉树（用特殊符号表示无结点），按层次方式列出，直接读入并装入数组

创建链接二叉树有多种方式。这里介绍一种：

设计一种线性输入方式：

- 按给出先序周游的结点序列，如：A B D C E G F H I
- 加入空二叉树信息（支持二叉树构造），如用 @

A B D @ @ @ C E @ G @ @ F H @ @ I @ @

相当于做出扩充二叉树，所有外部结点用@标记

```
/* 递归地创建二叉树 */
BinTree createBTree() {
    PBNode  pnd;     char ch;
    scanf(" %c", &ch);
    if (ch == '@') return NULL;
    pnd = (PBNode )malloc(sizeof(BNode));
    if ( pnd == NULL ) { printf("NoSpace!"); return NULL; }
    pnd->info = ch;
    pnd->llink = createBTree();      /* 构造左子树 */
    pnd->rlink = createBTree();      /* 构造右子树 */
    return pnd;
}
```

本函数创建以字符为数据的二叉树。如果用其他数据类型，程序需修改。完整二叉树的字符序列使函数结束

另一方法是用二叉树构造函数:

```
BinTree consBinTree(DataType a, BinTree l, BinTree r) {
   PBNode  pbnode = (PBNode )malloc(sizeof(BNode));
   if( pbnode == NULL )       {
      printf("Out of space!\n");        return NULL;
   }
   pbnode->info = a;
   pbnode->llink = l; pbnode->rlink = r;
   return pbnode;
}
```

构造二叉树的例（设 DataType 是 int，可根据需要定义）:

```
BinTree ta = consBinTree(3, NULL, consBinTree(5, NULL, NULL));
BinTree tb = consBinTree(4, ta, consBinTree(8, NULL, NULL));
```

用这一函数可以逐步构造出任意的二叉树

5.4 二叉树的应用

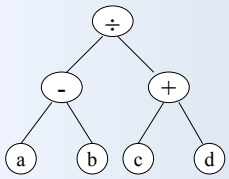
5.4.1 表达式树

二元运算符直接表示，一元运算符只有一个子树，一元和二元函数用与运算符同样的方式

多元函数特殊方式（例如用一个右子树序列）

计算表达式值（给定一组变量值），可通过后序周游完成

数学软件都采用这类表示方式



表达式 (a – b) / (c + d)

的二叉树表示

从后缀表达式构造表达式树

例：a 2 c b * 2 / + d - *

方法：用一个栈，其中存放构造的部分树

反复读入运算对象和运算符:

- 遇运算对象x：以x为数据构造一个结点的二叉树，入栈
- 遇运算符o：弹出两个子树，设分别为r和l，构造二叉树 consBinTree(o, l, r) 并将其入栈。

如果读完整个表达式时栈里只有一个元素，那就是构造出的表达式树

从中缀表达式构造表达式树

运算对象栈（保存构造好二叉树）和运算符栈 ost。逐个读入单词:

- 遇运算对象x，构造以x为数据的单结点二叉树压入 dst
- 遇运算符 o，与 ost 栈顶 o' 比较。o' 优先级不低于 o 时弹出 o'，从 dst 弹出两个二叉树，构造新二叉树并压入 dst。反复至栈顶运算符优先级低于o 时将运算符 o 压入 ost
- 遇左括号，直接进栈 ost
- 遇右括号，逐个弹出运算符，从 dst 弹出两个二叉树，构造新二叉树并压入 dst，直至把对应左括号弹出
- 遇表达式结束，逐个弹出运算符并构造，直至处理完全部运算符。若 ost 只剩一个元素则成功，否则表达式有错

5.4.2 优先队列和堆

优先队列

一种常用抽象数据类型，它遵循“最小元素先出”的原则

基本操作（代表优先队列的特征）:

- 向优先队列里插入一个元素(add)
- 在优先队列中找出最小元素（min）
- 删除优先队列中最小元素（removeMin）

优先级别代表了数据的某种性质，如用于描述一个大项目中各种工作任务的急迫程度，顾客信任度决定优先贷款

创建空优先队列:

PriorityQueue createEmptyPriQueue(void)

判断S是否为空（返回1/0值）:

int isEmpty(PriorityQueue S)

向S中加入元素e:

void add(PriorityQueue S, DataType e)

求出S中的最小元素:

DataType min(PriorityQueue S)

删除S中的最小元素:

void removeMin(PriorityQueue S)

简单实现可用线性表或元素按优先序排列的表。但插入，取最小，或删除操作中必然有 O(n) 操作

堆和优先队列

二叉树结构支持优先队列的高效实现，这里需要堆的概念

一棵完全二叉树称为堆，如果其结点的值满足堆序:

任何父结点值小于等于其两个子结点值（若子结点存在）:

$$\begin{cases} k_i \le k_{2i+1} \\ k_i \le k_{2i+2} \end{cases} \quad (i = 0, 1, \ldots, \lfloor (n-1)/2 \rfloor, \text{下标从 0 开始})$$

上面定义的是“小顶堆”。也可定义“大顶堆”:

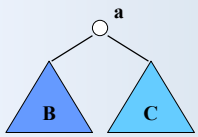
$$\begin{cases} k_i \ge k_{2i+1} \\ k_i \ge k_{2i+2} \end{cases}$$

下面考虑“小顶堆”，其最小元素在堆顶，O(1)时间可取得

完全二叉树可自然存放在数组里，因此堆的实现很简单

几个事实:

1. 若数组 $\{k_0, k_1, \ldots, k_{n-1}\}$ 是小顶堆，则堆顶元素必为值最小的元素
2. 去掉堆顶，其余元素形成两个堆
3. 去掉最后元素，剩下仍然是堆



用堆实现优先队列，需要解决:

1. 怎样加入一个元素，使得到的结果仍然是一个堆
2. 删除最小元素后，如何将剩下的元素恢复为堆

下面说明，这两个操作均可在 O(log n) 时间内完成

其他操作都简单，取最小元素是 O(1) 操作

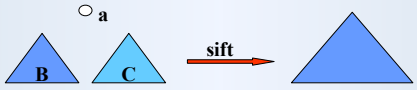
向上筛选：堆B后增加一个元素a，把它们做成一个堆:

方法：不断用 a 与其父结点数据比较，如果a小就交换两个元素的位置。直至a的父结点的数据小于等于a时停止

插入操作的实现：把新元素放在已有元素之后，执行一次向上筛选动作

完全二叉树高度是 O(log n)，插入可在 O(log n) 时间完成

向下筛选：两个堆B,C和元素a，把它们做成一个堆:



方法：用a与B、C的顶元比，最小者作为整个堆的顶。若a非最小，最小的必为B（或C）的顶元素。下面考虑把 a 放入去掉堆顶的B（或C）（是同样问题）。结束情况:

1，在某次比较中a最小，以它为顶的这个局部是堆

2，a 落到底，这时它本身就是堆

两种情况下，堆的构造都完成了。

删除操作的实现：删去堆顶，从堆最后取一个元素放在堆顶，执行一次向下筛选。O(log n) 操作


```
/* 程序实现 */
typedef struct PriorityQueue {
   int MAXNUM;  /*元素个数上限 */
   int n;              /*堆中的实际元素个数*/
   DataType *pq;   /*堆中元素的顺序表示*/
} PriorityQueue, * PPriorityQueue;

void add_heap(PPriorityQueue papq, DataType x) {
   int i = paqu->n; DataType *elems = papq->pq;
   if (papq->n >= papq->MAXNUM) {
      printf("Full!\n");        return;
   }
   for ( ; i > 0 && elems[(i - 1) / 2] > x; i = (i - 1) / 2 )
      elems[i] = elems[(i - 1) / 2];  /*找插入的位置*/
   elems[i] = x;  /*存入x*/
   papq->n++;  /* 计数增加 */
}
```

```
void removeMin_heap(PPriorityQueue papq) {
   int s, i, child;  DataType temp, *elems = papq->pq;
   if (isEmpty_heap(papq)) { printf("Empty!\n"); return; }
   s = --papq->n; /* 改元素计数（删除） */
   temp = elems[s];  /* 取原最后元素到temp */
   /* 从根结点出发，找 temp 的正确插入位置 */
   for (i =0, child = 1; child < s; ) { /*找存元素位置*/
      if (child < s - 1 && elems[child] > elems[child + 1])
         child++; /* 选择 i 的较小子结点 */
      if (temp > elems[child]) { /* 需要继续向下考虑 */
         elems[i] = elems[child]; i = child; child = 2 * i + 1;
      }
      else break;  /* 已找到适当位置 */
   }
   elems[i] = temp;  /* 把最后元素填入 */
}
```


利用堆的概念和操作，得到了优先队列的一种高效实现

- 实现基础是连续表（或者数组），无需保存其他附加信息
- 插入和删除元素的操作都具有 O(log n) 复杂性，取最小元素操作具有 O(1) 复杂性

但是：

- 这种优先队列保存的元素个数受到数组大小的限制
- 插入元素时存储区溢出的问题可采用动态顺序表技术，但这一次插入就需要 O(n) 的时间

优先队列在被作为辅助结构用在许多复杂算法的实现中

其操作效率对许多高级算法的复杂性有重大影响。人们还提出另外一些实现技术，例如斜堆（也是一种二叉树）

5.4.3 哈夫曼(Huffman)树

哈夫曼树可看作二叉树的一种应用，哈夫曼编码是一组信息的最短编码，在信息领域有重要理论和实际价值

带权外部路径长度：

扩充二叉树的外部路径长度：（外部结点的路径长度之和）

$$E=\sum_{i=1}^{m} l_i$$

l_i为从根到外部结点i的路径长度，m为外部结点个数

扩充二叉树的带权外部路径长度

$$WPL=\sum_{i=1}^{m} w_i l_i$$

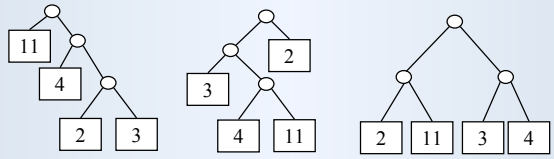
m为外部结点个数，w_i是外部结点i的权值，l_i为从根到i的路径长度。权值是与结点关联的某种量

哈夫曼树

定义：设有实数集$\{w_1, w_2, w_3, \ldots, w_m\}$，要求构造一棵扩充二叉树，包含 m 个分别以 w_i（i = 1, 2, …, m）为权的外部结点，其带权外部路径长度 WPL 达到最小

这样的扩充二叉树称为**哈夫曼树**或**最优二叉树**

例：以 { 2, 3, 4, 11} 为外部结点权的几棵扩充二叉树：



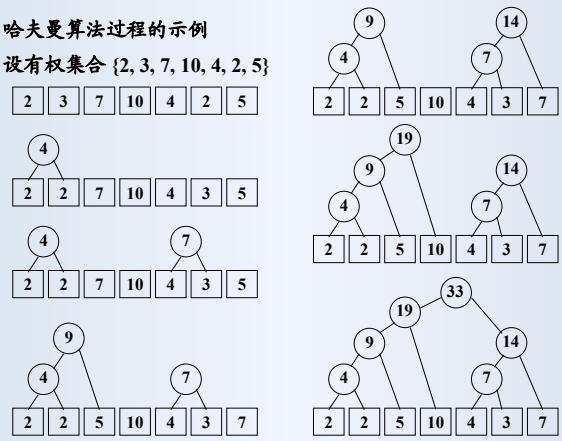
哈夫曼算法

给定权值 $\{w_1, w_2, \ldots, w_m\}$。算法：

1. 构造包含 m 棵二叉树的集合 $F = \{T_1, T_2, \ldots, T_m\}$，其中二叉树 T_i 只含权为 w_i 的根结点
2. 从 F 中选取两棵权最小的树作为左右子树，构造一棵新二叉树，其根结点的权值为两棵子树的根结点权值之和
3. 从 F 删除所选的两棵树，把新构造的二叉树加入F
4. 重复（2）和（3），直到 F 中只含一棵树为止

哈夫曼算法过程的示例

设有权集合 {2, 3, 7, 10, 4, 2, 5}



哈夫曼(huffman)算法的实现

存储结构:

```
typedef struct HtNode { /* 哈夫曼树结点的结构 */
      int ww;
      int parent, llink, rlink;
} HtNode;

typedef  struct HtTree { /*  哈夫曼树定义  */
      HtNode  *ht;
      int   root;/* 哈夫曼树根在数组中的下标 */
} HtTree, *PHtTree;
```

用顺序表表示哈夫曼树的结点，用数组下标表示子结点和父结点关系。在构造哈夫曼树的过程中逐步建立这些链接

构造过程:

- m 个权值（整数）存放在（参数）数组 w 里
- 在可容纳 2m-1 个 HtNode 结点的数组里构造哈夫曼树

1. 初始化: 把给定权值存入前 m 个结点，其父指针和子指针都赋值为 –1，表示无链接
2. 反复做m-1 遍: 在数组的有效结点范围里找权值最小且无父结点的两个结点，为它们建立一个父结点（在数组右面部分的下一个空闲位置），并建立父子链接
3. 哈夫曼树的树根是最后一个结点（下标2*m-2）

不变关系：

当前树集合的结点连续存在结点表左边部分。各树根结点的权是其两子结点的权之和，其父结点链接为 -1


```
/* 分配并初始化Huffman 树数据结构 */
PHtTree initHFT (int m, int *w) {
   int i;
   PHtTree pht = (PHtTree)malloc(sizeof(HtTree));
   if (pht == NULL) {  printf("NoSpace!\n");  return NULL; }
   pht->ht = (HtNode*) malloc((2*m-1)*sizeof(HtNode) );
   if (pht->ht == NULL) {
      printf("NoSpace!\n");  return NULL; }
   for (i = 0; i < 2*m-1; i++) { /* 置初态 */
      HtNode *p = &(pht->ht[i]);
      p->llink = p->rlink = p->parent = -1;
      p->ww = (i < m) ? w[i] : -1;  /*其他结点的权置为 -1*/
   }
   return pht;
}
```



```
/* 构造具有 m 个叶结点的哈夫曼树*/
PHtTree huffman(int m, int *w) {
   PHtTree pht = initHFT(m, w);    int i, j, x1, x2, w1, w2;
   if (pht == NULL) return NULL;
   for ( i = 0; i < m - 1; i++) {    /* 每次构造一个内部结点，共m-1次 */
      w1 = w2 = MAXINT; x1 = x2 = -1; /* x1和x2是结点下标 */
      for (j = 0; j < m+i; j++) {  /* 找两个权值最小且无父结点的结点 */
         /* 不变式: w1 为已知最小权值, w2 为次小, x1, x2 为它们的位置 */
         HtNode *p = &(pht->ht[j]); /* 考察一个结点 */
         if (p->parent != -1) continue;   /* p 不是子树根，向下找 */
         if (p->ww < w1) { w2 = w1; x2 = x1; w1 = p->ww; x1 = j; }
         else if (p->ww < w2) {  w2 = p->ww; x2 = j; }
      }
      pht->ht[x1].parent = pht->ht[x2].parent = m + i; /* 构造内部结点 */
      pht->ht[m+i].ww = w1 + w2;
      pht->ht[m+i].llink = x1;  pht->ht[m+i].rlink = x2;
   }
   pht->root = 2*m - 2;
   return pht;
}
```

本算法具有平方复杂性。能加快其速度吗?

用优先队列可加快哈夫曼树的构造速度，有兴趣的同学可以自己改进算法

哈夫曼编码

设有如下基本数据集合:

$D=\{d_1, d_2, \ldots, d_n\}$

$W=\{w_1, w_2, \ldots, w_n\}$

其中：D为需要编码的字符集，W为D中各字符在实际信息传递（或者存储）中出现的频率

要求：（1）编码的平均总长度最短;

（2）如果$d_i \neq d_j$，那么 d_i 编码不是 d_j 编码的前缀。

第二个条件使解码时容易判断是否已经得到一个字符的编码

哈夫曼提出了一种解决这一问题的方法，即哈夫曼编码

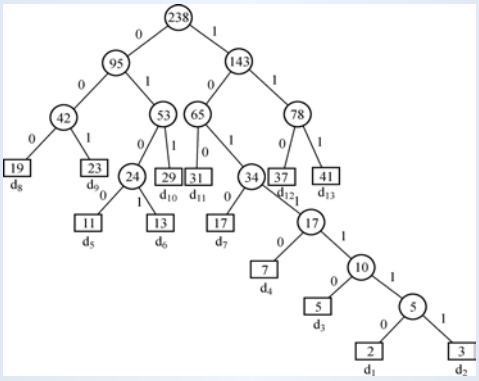
通过构造哈夫曼树，实现哈夫曼编码:

以字符 $d_1, d_2, \ldots, d_n$ 为外部结点标注，把$w_1, w_2, \ldots, w_n$分别作为这 n 个外部结点的权，构造一棵哈夫曼树。

在得到的哈夫曼树中，将所有从一个结点引向其左子女的边标二进制数字 0；引向其右子女的边上标 1。

以从根结点到一个叶结点的路径上的二进制数字序列，作为这个叶结点的字符的编码。这就是哈夫曼编码

可以证明：对于任意的集合对 D 和 W，这样得到的哈夫曼编码是最优（最短）编码



w = {2，3，5，7，11，13，17，19，23，29，31，37，41}

补充：线索二叉树

把周游二叉树得到的信息保存在树结构里供后面操作使用

（1）增加两个指针

（2）利用结构中的空链域，并设立标志。

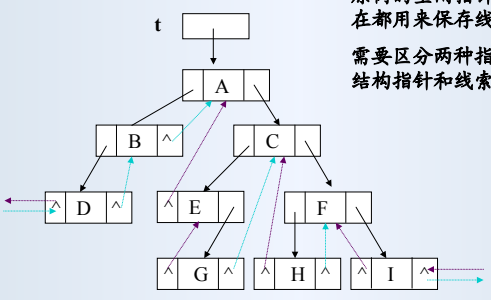
线索：指向结点前驱或后继的指针。

线索二叉树：加了线索的二叉树。可以结点的空指针域表示相关线索，需要加标记区分正常子结点指针和线索指针

线索化：对二叉树以某种序周游将其修改为线索二叉树

按对称序线索化二叉树：按对称序周游二叉树，周游到一个结点时就为它增加线索信息。

按对称序周游对称序穿线树，可以利用线索进行周游。

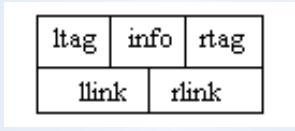


原树的空闲指针域，现在都用来保存线索信息。

需要区分两种指针：树结构指针和线索指针

线索化后的二叉树


```
typedef struct ThrTreeNode ThrTreeNode, * PThrTreeNode;
    /* 线索二叉树结点类型和指向它的指针类型 */
struct ThrTreeNode {        /* 线索二叉树中结点的定义 */
   DataType info;
   PThrTreeNode  llink, rlink;
   int  ltag, rtag;  /* 0:正常的二叉树指针，1:线索指针 */
};
typedef ThrTreeNode * ThrTree;/* 线索二叉树类型的定义 */
typedef ThrTree * PThrTree;   /* 线索二叉树类型的指针类型 */
```



按对称序线索化的算法：

- 操作前所有指针都是正常指针，tag都为0
- 周游中用线索指针代替空指针
- 线索化的过程首先是一个非递归的对称序周游过程
- 其中在遇到空指针时修改它，并修改相应的tag

线索树的最大优点是可以较方便地找到周游序列的前一个和下一个结点

由于这种情况，再做这种周游时就不需要使用栈了（第二版上有相应算法）


```
/*按对称序线索化二叉树*/
void thread(ThrTree  t) {
   PSeqStack st = createEmptyStack ();/* 栈元素是ThrTree */
   ThrTree p = t，pr = NULL;
   while ( p != NULL || !isEmptyStack_seq(st) ) {
      while (p!=NULL) {push_seq(st,p);p= p->llink;}
      p = top_seq(st);  pop_seq(st);
      if (pr!=NULL) {
         if (pr->rlink==NULL) { pr->rlink = p;pr->rtag = 1; }
         /*修改前驱结点的右指针*/
         if (p->llink==NULL) { p->llink = pr; p->ltag = 1;}
         /*修改该结点的左指针*/
      }
      pr = p;  p = p->rlink; /* 进入右子树 */
   }
}
```

5.5 树的运算和树的实现

5.5.1 树的基本运算

抽象运算（操作）

- 创建树（空树，或者包含若干子树的树）

Tree createTree(Node p, Tree t1, Tree t2, …, Tree ti)

i = 1, 2, 3, …

- 判断某棵树是否为空

int isEmpty (Tree t)

- 求树中的根结点，空树时返回特殊值

Node root (Tree t)

- 求指定结点的父结点，结点是树根时返回特殊值

Node parent (Node p)

- 求树中某个指定结点的最左子结点，指定结点为树叶时返回特殊值

Tree leftChild (Tree t)

- 求树中某个指定结点的右兄弟结点，当指定结点没有右兄弟时返回特殊值

Tree rightSibling (Tree t, Tree child)

- 周游：按某种方式访问树中所有结点且每结点只访问一次

void trivesal (Tree t)

void trivesal (Tree t, Operation op)

5.5.2 树和树林的周游

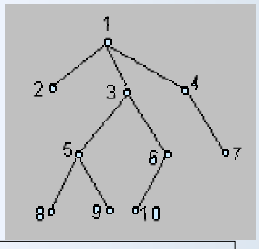
1. 周游：按某种系统的方式访问树中所有结点，而且每个结点恰好访问一次的过程。
2. 周游方法：按深度周游和按宽度周游。

注意：按深度优先和宽度优先周游，访问结点的顺序就是深度优先搜索和宽度优先搜索中访问结点的顺序。

在一般的搜索问题里，搜索过程经历的结点和结点间联系形成了一棵“树”，称为“搜索树”

由于非空树总有一个根结点，下面算法中用于表示树的 t 也用于表示其根结点

(I) 按深度周游



先根序

1）访问根结点

2）从左到右按先根次序周游根结点的每棵子树

1, 2, 3, 5, 8, 9, 6, 10, 4, 7

中根序（称为“中根”不太有理）

1）按中根次序周游根结点的最左子树

2）访问根结点

3）从左到右按中根次序周游根结点的其它各子树

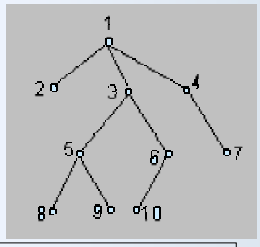
2, 1, 8, 5, 9, 3, 10 , 6, 7, 4

后根序

1）从左到右按后根次序周游根结点的每棵子树

2）访问根结点

2, 8, 9, 5, 10, 6, 3, 7, 4, 1



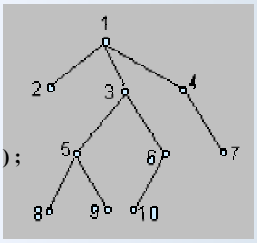
（II）按宽度（层次）周游

先访问层数为0的结点，然后从左到右逐个访问层数为1的结点，依此类推，直到访问完树中全部结点。

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

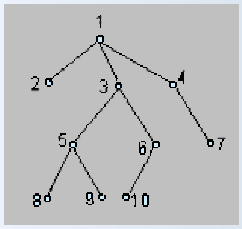
先根序周游的递归算法

```
void preOrder( Tree t ) { /*先根序*/
   Tree c;
   visit( root( t ) );   /* 访问根结点 */
   /* 考查剩余部分 */
   for ( c = leftChild ( t ); !isEmpty ( c ) ;
         c = rightSibling ( t, c ) )
      preOrder ( c );
}
```



- 访问各子树的方式与访问整个树一样
- 对各子树的访问按规定顺序进行，用循环描述
- 各种深度优先周游都可以定义非递归算法，其中需要用一个栈保存信息。这里讨论先根序的非递归算法

先根序周游的非递归算法




```
void npreOrder ( Tree t ) {
   PSeqStack s =
        createEmptyStack ( );
   Tree c = t;
   while ( !isEmpty ( c ) ) {
      visit ( root( c ) );
      push ( s, c );
      c = leftChild ( t, c );
      while ( isEmpty ( c ) && !isEmptyStack (s) ) {
         c = rightSibling ( t, top (s) ); /* 回溯并考虑其他分支 */
         pop (s);
      }
   }
}
```



```
void inOrder( Tree t ) { /*中根序周游*/
   Tree c = leftChild ( t );
   if ( !isEmpty ( c ) ) inOrder( c ); /* 按中序周游最左子树 */
   visit( root( t ) );                 /* 访问根结点 */
   if ( !isEmpty ( c ) )
      while ( !isEmpty(c = rightSibling ( t, c )) ) /* 剩余部分 */
         inOrder ( c );
}

void postOrder( Tree t ) { /*后根序周游*/
   Tree c = leftChild ( t );
   while ( !isEmpty ( c ) ) { /* 首先按后序周游树的各子树 */
      postOrder ( c );
      c = rightSibling ( t, c );
   }
   visit( root( t ) ); /* 访问根结点 */
}
```

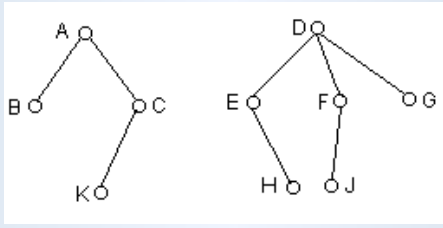


```
void levelOrder( Tree t) {
    PSeqQueue q = createEmptyQueue( );
    Tree c = t;
    if (isEmpty(c) ) return;
    enQueue_seq(q, c);  /* 根结点入队 */
    while ( !isEmptyQueue (q) ) {
       c = frontQueue (q);
       deQueue (q);
       visit( root( c ) );    /* 访问队列里最先入队的结点 */
       c = leftChild ( t, c );
       while ( !isEmpty(c) ) { /* 被访问结点的各子结点入队 */
          enQueue (q, c);
          c = rightSibling ( t, c );
       }
    }
}
```


树林的周游

是其中的树的周游的总和

1. 先根（A, B, C, K, D, E, H, F, J, G)
2. 后根（B, K, C, A, H, E, J, F, G, D)



5.5.3 树的存储表示

树的存储表示

- 父指针表示法
- 子表表示法
- 长子–兄弟表示法

首先，树可以借助于二叉树结构实现。下面先讨论这个问题

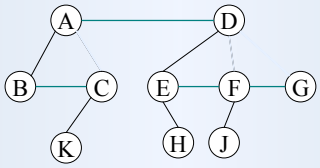
树、树林与二叉树的转换

树、树林 ⟷ 二叉树

树（树林）与二叉树之间都存在一一对应关系，因此二叉树的所有实现技术也都是树和树林的实现技术

树、树林转换为二叉树，步骤:

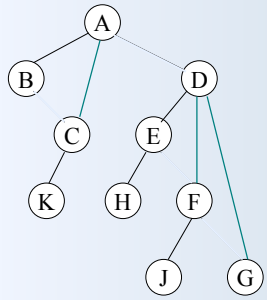
1. 在所有相邻的兄弟结点之间连一条线;
2. 对每个非叶结点，只保留它到其最左子女的连线，删去它到其它子女的连线



树林转换为二叉树

二叉树转换为树、树林:

1. 若某结点是其父母的左子女，则把该结点的右子女，右子女的右子女，…都与该结点的父母连线;
2. 去掉原二叉树中所有父母到右子女的连线。



由于二叉树比较规范。树的最常见表示方式就是转换到二叉树，而后用二叉树的表示技术

父指针表示法

用连续空间存储树结点，每个结点设一个指示器，指示其双亲结点（父结点）位置。可用结点下标

```
typedef struct  ParTreeNode { /* 树结点结构 */
   DataType info;     /* 结点中的数据 */
   int    parent;     /* 结点的父结点位置 */
} ParTreeNode;

typedef struct  ParTree  {
   int  n;            /* 记录树中结点个数 */
   ParTreeNode  nodelist[MAXNUM]; /* 存储树中各结点 */
} ParTree,  *PParTree;
```

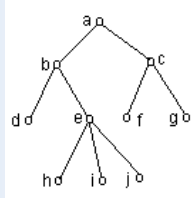
- 就是用顺序表保存树中结点及树的结构信息
- 树的结构信息通过每个结点的 parent 成分表示

优点: a) 容易找到父结点及所有祖先（沿 parent 指针）
b) 能找到结点的子女和兄弟（通过穷尽检查）

缺点: a) 没表示兄弟结点之间的左右顺序
b) 找结点的子女和兄弟比较麻烦

改进方法:

按某种周游序在数组中保存结点。常用按先根序存放结点，如图:



	info	parent
0	a	-1
1	b	0
2	d	1
3	e	1
4	h	3
5	i	3
6	j	3
7	c	0
8	f	7
9	g	7

- 子结点在根结点之后
- 左边的兄弟结点在前，右边的兄弟结点在后


```
int rightSibling_partree(PParTree t, int p) {
   int i; /* 找结点p的右兄弟结点（下标对应着结点） */
   if (p >= 0 && p < t->n) {
      for (i = p+1; i <= t->n; i++)
         if (t->nodelist[i].parent == t->nodelist[p].parent)
            return i;
   }
   return -1;
}

/* 依先根序列存储时，求最左子结点的运算简化为*/
int leftChild_partree(PParTree t, int p) {
   if (t->nodelist[p+1].parent == p)
      return p+1; /* 最左子结点总排在下一位置 */
   else
      return -1;
}
```

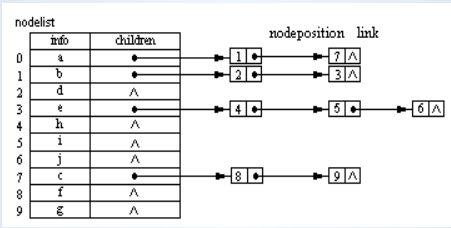

子表表示法

- 结点表中每个元素（结点）关联一个子结点（链）表，其中存放该结点的所有子结点
- 结点表顺序存放，子表用链接表示

```
struct EdgeNode {  /* 子表中节点的结构 */
   int  nodeposition;
   struct EdgeNode *link;
};

typedef struct ChiTreeNode  {  /* 结点表中节点结构 */
   DataType info;
   struct EdgeNode *children;
} ChiTreeNode;
```

子表表示的树结构定义:

```
typedef struct ChiTree  {       /* 树结构 */
    struct  ChiTreeNode  nodelist[MAXNUM];
    int root;                   /* 根结点的位置 */
    int n;                      /* 结点的个数 */
} ChiTree,  * PChiTree;
```




```
/* 求 p 的右兄弟结点 */
int rightSibling_chitree(PChiTree t, int p) {
   int i;     struct EdgeNode  *v;
   for (i = 0; i < t->n; i++) {
      v = t->nodelist[i].children;
      while (v != NULL)
         if (v->nodeposition == p)
            if (v->link == NULL)
               return -1; /* 已没有下一个兄弟结点 */
            else
               return(v->link->nodeposition);
         else
            v = v->link;
   }
   return -1;
}
```

如果知道结点所在的链表（子表）结点，求右兄弟就非常简单了

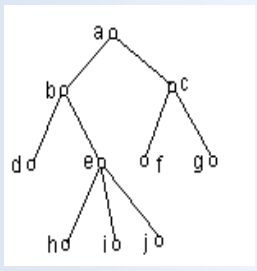

```
/* 求父结点 */
int parent_chitree(PChiTree t, int p) {
    int i;
    struct EdgeNode *v;
    for (i = 0; i < t->n; i++) {   /* 逐个检查是否父结点 */
        v = t->nodelist[i].children;
        while ( v != NULL )
            if ( v->nodeposition == p )
                return  i;  /* 结点 i 的子结点表中有p，返回 i */
            else
                v = v->link;
    }
    return -1;  /* 无父结点，返回值为-1 */
}
```

求最左子结点、顺序访问一个结点的各子结点都很简单

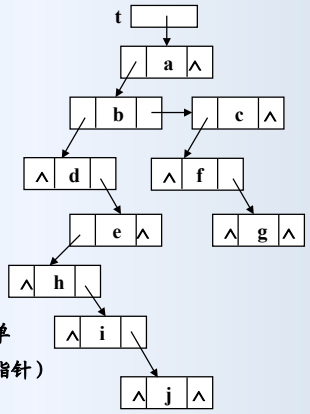
长子-兄弟表示法

除信息域外，每个结点包含指向其最左子女和右兄弟的指针

```
typedef struct  CSNode *PCSNode; /* 结点指针 */

typedef struct  CSNode { /* 结点结构 */
    DataType  info;   /* 结点中的元素 */
    PCSNode  lchild; /* 结点的最左子女的指针 */
    PCSNode  rsibling;        /* 结点的右兄弟的指针 */
} CSNode; /* 结点类型 */

typedef  struct CSNode  *CSTree;  /* 树类型定义 */
```

树的长子兄弟表法



找长子或者右兄弟都比较简单

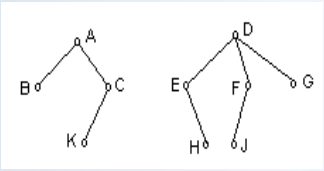
找父结点很困难（可增加父指针）

就是树的二叉树表示!

5.5.4 树林的存储表示

是树的各种表示法的变形，同样可以采用:

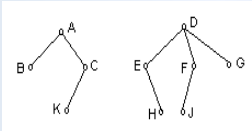
- 父指针表示法
- 子表表示法
- 长子-兄弟表示法



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

树林的父结点表示方法

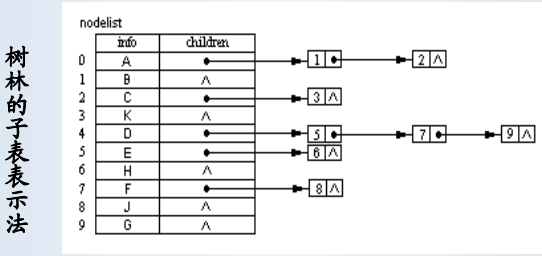
- 需要设法表示各树的树根结点



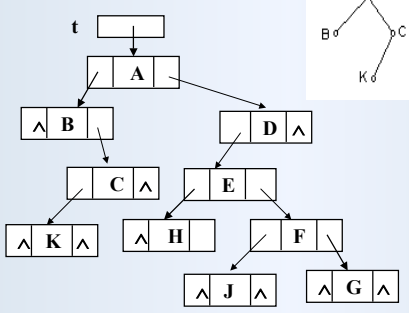
树林的子表表示法

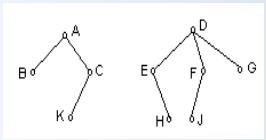
nodelist

	info	children
0	A	1
1	B	∧
2	C	3
3	K	∧
4	D	5
5	E	6
6	H	∧
7	F	8
8	J	∧
9	G	∧



需要各树根结点的信息（可增加其他结构来表示）





沿着第一个树根的右兄弟指针，可以找到所有树根

树林的长子兄弟表示法

在实际的算法和程序里，树的使用远不如二叉树那么广泛

主要原因是树的结构不规整，一棵树中各个结点的子结点个数可能不同，而且没有限制。在计算机里表示和处理都比较麻烦

如果实际中真正需要树结构，通常也是利用树与二叉树的关系（下面介绍），把树转换为二叉树后存储和处理

小　　结

- 树、树林、二叉树的基本概念和术语;
- 二叉链表存储结构
- 树、二叉树的周游
- 哈夫曼树的构造方法及应用

非递归周游算法:

前序和中序周游较简单，易实现（自己考虑）。书上给了一个后序周游算法，其中用一个栈，栈里记录指针和一个标志

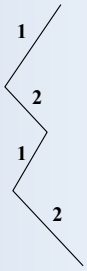
```
/* 算法用的堆栈元素 */
typedef struct {
   PBNode ptr;    /* 进栈结点 */
   int  tag;          /* 标记 */
} Elem;

/*栈顺序表示*/
#define MAXNUM 20          /* 栈中最大元素个数 */
typedef struct  SeqStack {    /* 顺序栈类型定义 */
   int  t;           /* 指示栈顶位置 */
   Elem s[MAXNUM];
} SeqStack, *PSeqStack;/* 顺序栈类型的指针类型 */
```



```
void nPostOrder( PBTree t ) {
   PSeqStack st; Elem stnode;  PBNode  p; /* 当前处理的结点*/
   char continueflag; /* 继续退栈标志，从右子树返回访问完根后继续退栈 */
   if (*t == NULL) return;
   st = createEmptyStack_seq( );  p = t;       /* 从根结点开始 */
   do {                              /* 每次大循环进入由p所指子树周游 */
      while (p != NULL) { /* 反复把所遇结点进栈并进入其左子树 */
         stnode.ptr = p; stnode.tag = 1;
         push_seq(st, stnode);  p = leftChild_btree(p);
      }
      continueflag = 't';        /* 很容易用break语句，消除这个flag */
      while ( continueflag == 't' && !isEmptyStack_seq(st) ) {
         stnode = top_seq(st); pop_seq(st);   p = stnode.ptr;
         if (stnode.tag == 1) { /* 从左子树回来，改标志重进栈并进入右子树 */
            stnode.tag = 2; push_seq(st, stnode);
            continueflag = 'f'; p = rightChild_btree(p);
         }
         else visit(p);
      }
   } while ( !isEmptyStack_seq (st) );        /* 栈为空时，全部周游完 */
}
```


不变式:

栈里保存经过但未访问的树（子树）根结点。结点的右子树尚未访问时 tag=1；已经或正在访问时 tag=2



实际上并不需要 tag 标记:

- 因为栈里每一个结点的父结点（如果存在）总是它的下面一个结点
- 正考虑的结点的父结点（如果存在）就是栈顶结点
- 根据本结点是其父结点的左子结点或者右子结点，就可以决定下一步怎么做
- 根据这个想法，完全可以去掉标志，简化算法
- 下面算法另外做了一些调整，重要的是“下行循环”


```
/* 简化算法。栈里只存结点指针 */

void nPostOrder2( PBinTree t ) {
   PSeqStack st = createEmptyStack_seq( );
   PBNode p = t;
   while ( p != NULL || !isEmptyStack_seq(st) ) {
      while (p != NULL) { /* 循环到栈顶结点的左右子树都空 */
         push_seq( st, p );
         p = leftChild_btree(p) ? leftChild_btree(p)
                                : rightChild_btree(p);
      }
      p = top_seq(st); pop_seq(st); visit(p); /* 栈顶是应访问结点 */
      if ( !isEmptyStack_seq(st) && leftChild_btree(top_seq(st)) == p )
         /* 栈不空且是从左子树退回 *
         p = rightChild_btree(top_seq(st)) ; /
      else p = NULL; /* 从右子树返回则强迫退栈 */
   }
}
```