第一章

◆ 数据：指能够被计算机识别、存储和加工处理的信息载体。

◆ 数据元素：就是数据的基本单位，在某些情况下，数据元素也称为元素、结点、顶点、记录。数据元素有时可以由若干数据项组成。

◆ 数据类型：是一个值的集合以及在这些值上定义的一组操作的总称。
在高级语言程序中又分为：非结构的原子类型和结构类型

◆ 抽象数据类型（ＡＤＴ）：是指一个数学模型以及定义在该模型上的一组操作。
一个抽象的数据类型的软件模块通常包含　定义和表示和实现
用三元组（Ｄ，Ｓ，Ｐ）：数据对象、数据关系、基本操作

◆ 数据结构：指的是数据之间的相互关系，即数据的组织形式。一般包括三个方面的内容:
数据的逻辑结构、存储结构和数据的运算。

◆ 逻辑结构：指各数据元素之间的逻辑关系。

◆ 存储结构：就是数据的逻辑结构用计算机语言的实现。

◆ 线性结构：数据逻辑结构中的一类，它的特征是若结构为非空集，则该结构有且只有一个开始结点和一个终端结点，并且所有结点都最多只有一个直接前趋和一个直接后继。线性表就是一个典型的线性结构。

◆ 非线性结构：数据逻辑结构中的另一大类，它的逻辑特征是一个结点可能有多个直接前趋和直接后继。

常用的存储表示方法有四种:

◆ 顺序存储方法：它是把逻辑上相邻的结点存储在物理位置相邻的存储单元里，结点间的逻辑关系由存储单元的邻接关系来体现。由此得到的存储表示称为顺序存储结构。

◆ 链接存储方法：它不要求逻辑上相邻的结点在物理位置上亦相邻，结点间的逻辑关系是由附加的指针字段表示的。由此得到的存储表示称为链式存储结构。

◆ 索引存储方法：除建立存储结点信息外，还建立附加的索引表来标识结点的地址。

◆ 散列存储方法：就是根据结点的关键字直接计算出该结点的存储地址。

渐近时间复杂度的表示法 T(n)=O(f(n))，这里的"O"是数学符号，它的严格定义是"若 T(n)和 f(n)是定义在正整数集合上的两个函数，则 T(n)=O(f(n))表示存在正的常数 C 和 n0 ,使得当 n≥n0 时都满足 0≤T(n)≤C·f(n)。"用容易理解的话说就是这两个函数当整型自变量 n 趋向于无穷大时，两者的比值是一个不等于 0 的常数。这么一来，就好计算了吧。

求某一算法的时间复杂度是关于N的统计，下面的例子很有反面意义

```
 x=91; y=100;
while(y>0)
if(x>100)
{x=x-10;y--;}
else x++;
```

◆ T(n)=O(1)

◇ 这个程序看起来有点吓人，总共循环运行了 1000 次，但是我们看到 n 没有？没。

◇ 这段程序的运行是和 n 无关的，就算它再循环一万年，我们也不管他，只是一个常数阶的函数。

算法的时间复杂度仅与问题的规模相关吗?

◆ No,事实上，算法的时间复杂度不仅与问题的规模相关，还与输入实例中的元素取值等相关，但在最坏的情况下，其时间复杂度就是只与求解问题的规模相关的。我们在讨论时间复杂度时，一般就是以最坏情况下的时间复杂度为准的。

增长率由小至大的顺序排列下列各函数：2^100, (2/3)^n，(3/2)^n， n^n , , n! ，2^n ，lgn ,n^lgn, n^(3/2)

◇ 分析如下：2^100 是常数阶； (2/3)^n 和 (3/2)^n 是指数阶,其中前者是随 n 的增大而减小的； n^n 是指数方阶； √n 是方根阶, n! 就是 n(n-1)(n-2)... 就相当于 n 次方阶；2^n 是指数阶，lgn 是对数阶 ,n^lgn 是对数方阶, n^(3/2)是 3/2 次方阶。根据以上分析按增长率由小至大的顺序可排列如下：

◆ (2/3)^n < 2^100 < lgn < √n < n^(3/2) < n^lgn < (3/2)^n < 2^n < n! < n^n

算法：是对特定的问题求解步骤地一种描述，是指令的有限序列。有五个基本的特性

有穷性、确定性、可行性、输入、输出

确定性：每条指令不能有二义性，对于同样的输入有同样的输出

可行性：算法中所用到的操作都是已经实现的基本运算或通过有限次能实现的

输入：有 0 个或多个输入

输出：有一个或多个输出

设计算法的要求（追求的目标）

正确性、可读性、健壮性、效率与低存储量需求

算法原地工作：当空间复杂度为 O(1)时，称算法为就地工作（原地工作）。

多型数据类型：是指其值的成分不确定的数据类型。从抽象数据类型的角度看，具有相同的数学抽象特性，故称之为 多型数据类型。

数据结构是一门研究什么内容的学科?

研究非数值计算的程序设计问题中计算机的操作对象以及他们之间的关系和操作等学科

对于一个数据结构，一般包括哪三个方面的讨论?

数据的逻辑结构、存储结构和数据的运算。

逻辑结构有 线形结构（1） ， 树型结构 （2） ， （3）网状结构 ，集合（4） 四种。

平方和公式：$1^2+2^2+3^2+....+n^2$ =n*(n+1)*(2n+1)/6

斐波那契数列计算的时间复杂度是 O（n）

第二章

循环链表是一种首尾相接的链表。也就是终端结点的指针域不是指向 NULL 空而是指向开始结点(也可设置一个头结点)，形成一个环。采用循环链表在实用中多采用尾指针表示单循环链表。这样做的好处是查找头指针和尾指针的时间都是 O(1)，不用遍历整个链表了。
判别链表终止的条件也不同于单链表，它是以指针是否等于某一指定指针如头指针或尾指针来确定。

何时选用顺序表、何时选用链表作为线性表的存储结构为宜?
答：
在实际应用中，应根据具体问题的要求和性质来选择顺序表或链表作为线性表的存储结构，通常有以下几方面的考虑：
1.基于空间的考虑。当要求存储的线性表长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表；反之，当线性表长度变化大，难以估计其存储规模时，采用动态链表作为存储结构为好。
2.基于时间的考虑。若线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜；反之， 若需要对线性表进行频繁地插入或删除等的操作时，宜采用链表做存储结构。并且，若链表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

第三章

例如：Exp=a*b+(c-d/e)*f

若 Exp=a*b+(c-d/e)*f 则它的

前缀式为： +*ab*-c/def

中缀式为： a*b+c-d/e*f

后缀式为： ab*cde/-fx+

综合比较它们之间的关系可得下列结论：

1．三式中的 “操作数之间的相对次序相同”；

(二叉树的三种访问次序中，叶子的相对访问次序是相同的)

2．三式中的 “运算符之间的的相对次序不同”；

3．中缀式丢失了括弧信息，致使运算的次序不确定；

（而前缀和后缀运算只需要一个存储操作数的栈，而中缀求值需要两个栈，符号栈和操作数栈）

4．前缀式的运算规则为：连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；

5．后缀式的运算规则为：

运算符在式中出现的顺序恰为表达式的运算顺序；

每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式；

6．中缀求值的运算规则：

如果是操作数直接入栈。

如果是运算符。这与当前栈顶比较。个如果比当前栈顶高，则入栈，如果低则说明当前栈顶是最高的必须把他先运算完了。用编译原理的话就是说当前栈顶已经是最左素短语了）

其实中缀表达式直接求值和把中缀表达式转化成后缀表达式在求值的过程惊人的相似，只不过是直接求值是求出来，而转化成后缀是输出来。

中缀表达式直接求值算法：

```
OPNDType EvalueExpression()
{   //OPTR 和 OPND 分别为运算符栈和操作数栈
    InitStack(OPTR);Push(OPTR,'#');
    InitStack(OPND);c=getchar();
    While(c!='＃'|| GetTop(OPTR)!='#')
     {
        If(!IN(c,OP) )   //如果是操作数，直接入操作数栈
        {    push(OPND,c);
             c=getchar();
        }
        Else    //如果是运算符，则与当前的栈顶比较
         {
             Switch(Precede(GetTop(OPTR),c))
             {
                 Case '<': push(OPTR,c);//比当前栈顶高，这入栈
                           c=getchar();
                           break;
                 Case '= ':Pop(OPTR,x);      //在我们设计的优先级表中，
                           c=getchar();       //只有'('和')'存在相等的情况，
                           break;             //而在规约中间只存在 '(' ')'
                                              //所以我们直接把'('弹出就可以了
                 Case '>':                 //比当前栈顶低，则要把栈顶先运算完（先规约）
                           pop(OPTR,theta);    //把栈顶运算符弹出
                           Pop(OPND,b);        //取出操作数，并且是前操作数
                           Pop(OPND,a);        //在下面（栈的先进后出）
                           Push(OPND,Operate(a,theta,b)); //运算的结果入栈
                                                //（他是其他运算符的操作数）
                           Break;
             }//Switch
         }//else
    }//whild
    Return GetTop(OPND);//操作数栈中最后剩下的就是整个表达式的结果了。
}
```

中缀表达式转化成后缀表达式算法

```
void trans-post(char E[n] ,char B[n])   //中、后缀表达式转换//
{       //E[n]是中缀表达式、B[n]是后缀表达式存储的空间
        int i=0,j=0; char x; stype S;
        Clearstack(S); Push(S,'#')；//'#'入栈//
        do {
```

```
            x=E[i++] ; //扫描当前表达式分量//
            if (x='#')   //到了中缀表达式最后了
                while(!Emptystack(S))  //全部退栈,#和#是优先级最低的，
                        B[j++]==pop(S);//所以当前栈的所有运算都要规约
                    //输出栈顶运算符，并退栈直到遇见栈底的开始放进去的那个＃//
            else if (isdigit(x))
                B[j++]=x;        //操作数直接输出//
            else if (x=')') //遇到）那么就一定要找到（
            {
                while (Getstop(S)!='(')
                   B[j++]=pop(S); //输出栈顶，并退栈//
                pop(S); //退掉'（'//
            }
            else     //x 为运算符//
            {
                while (precede(Getstop(S), x))   //栈顶（ θ1）与 x 比较//
                    B[j++]=pop(S);                // θ1 >=x 时，输出栈顶符并退栈//
                 Push(S,x); // θ1 < x 时 x 进栈//
            }
        } while (x!='#');
        B[j]='#';
}      //置表达式结束符//
```

双端队列：

两端都可以插入和删除，但实际应用中通常是输出受限的双端对列和输入受限的双端队列

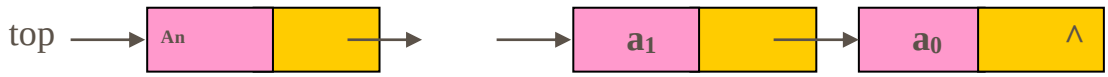
输入受限的双端队列指的是：一端可以输入输出另一端只能输出不能输入

输出受限的双端队列指的是：一端可以输入输出另一端只能输入不能输出

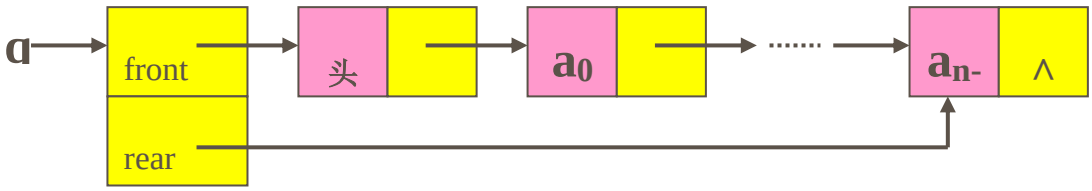
求从迷宫入口到出口的一条最短路径

要用到队列，因为队列可以用在广度优先中，队列中的元素表示离中心点依次越来越远。所以第一次找到出口肯定是半径最短的。

链式栈：



链式队列：



N 个结点的不同二叉树有：$C_{2n}^{n} - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^{n}$ 等于不同的进出栈总数

有 n 个数顺序（依次）进栈，出栈序列有 Cn 种，Cn=[1/（n+1）]*（2n）!/[(n!)*(n!)]。=

尾递归和单向递归的消除不需要借用栈

单向递归和尾递归可直接用迭代实现其非递归过程

其他情形必须借助栈实现非递归过程。

证明：若借助栈由输入序列 1, 2, …, n 得到输出序列为 $P_1, P_2, \cdots, P_n$（它是输入序列的一个排列），则在输出序列中不可能出现这样的情形：存在着 $i<j<k$, 使 $P_j<P_k<P_i$。

i<j　若 Pi<Pj 说明小的先于大的数出栈，那么也就是 Pi 出栈在 Pj 入栈前面

若 Pi>Pj 说明大的数先于小的数出栈，那么在 Pi 出栈前 Pj 一定在栈中

证明：1）j<k 且 Pj<Pk 说明 Pj 出栈在 Pk 入栈的前面；

2）i<k 且 Pi>Pk 说明 Pi 出栈前，Pk 在栈中

3）i<j 且 Pi >Pj 说明 Pi 出栈前 Pj 在栈中

而 Pi 是最先出栈的那么在 Pi 为栈顶的时候，Pj 和 Pk 一定都同时被压入栈中，那么就与 1 矛盾了，1 要求 Pj 要在 Pk 入栈前出栈，而此时 Pk Pj 都在栈中所以假设不成立。

用两个栈来模拟一个队列

栈的特点是后进先出，队列的特点是先进先出。所以，用两个栈 s1 和 s2 模拟一个队列时，s1 作输入栈，逐个元素压栈，以此模拟队列元素的入队。当需要出队时，将栈 s1 退栈并逐个压入栈 s2 中，s1 中最先入栈的元素，在 s2 中处于栈顶。s2 退栈，相当于队列的出队，实现了先进先出。显然，只有栈 s2 为空且 s1 也为空，才算是队列空。

第四章

KMP 算法和朴素的匹配算法的关键区别就是解决了主串指针 i 的回溯，原理如下

设主串 S[]和模式串 T[],如比较到模式串的第 j 个字符。 $S_1S_2S_3\cdots\cdots S_{i-2}S_{i-1}S_i$

$T_1T_2T_3\cdots\cdots T_{j-2}T_{j-1}T_j$

当主串指针 i 和模式串指针 j 比较时 ，说明他们前面的所有字符都已经对应相等了。而 Next[j]＝k 的定义是 T1T2…Tk-1＝＝Tj-k+1Tj-k+2….Tj-1 且 k 是最大了，没有更长的了。

所以 Si 和 Tj 比较失败时 Si 和 Tk 去比较。不可能有 $S_1S_2S_3\cdots\cdots S_{i-2}S_{i-1}S_iS_{i+1}$ 这种

$T_1T_2\cdots\cdots\cdots T_{j-2}T_{j-1}T_j$

匹配的成功，因为 S2S3…..Si-1= =T2T3……Tj-1,而 T2T3….Tj-1 是不等于 T1T2….Tj-2。除非 next[j]=j-1;因为 next 定义的是最长的。所以任何挪动小于 next[j]的串的匹配都是不能成功的。直到 Tnext[j]和 S[i]相比是才是最早有可能成功的。

```
Int KMP_Index(Sstring S,Sstring T,int pos)
{
    i=pos;j=1;
    while(i<=S[0]&&j<=T[0])
    {
        If(j=0||S[i]=T[j])//j=0 表示模式串已经退到起点了说明在这个位置彻底不可能了，
          {   ++i; ++j;   } //i 必须下移,j 回到 1 开始
        Else          j=next[j];
    }
```

```
    If(j>T[0])    return i-T[0];
    Else    return 0;
}
```

求 next[j]的方法和原理

设 k=next[j];那么 T1T2…Tk-1= =Tj-k+1……Tj-2Tj-1;

若 Tj= =Tk,那么 T1T2…Tk-1Tk= =Tj-k+1……Tj-2Tj-1Tj;

所以 next[j+1]=k+1=next[j]+1;且 T1T2…Tk-1= =Tj-k+1……Tj-2Tj-1 已经是最长的序列，所以 k＋1 也是 next[j+1]最长的

若 Tj 不等于 Tk,那么就需要重找了。即…..Tj-1Tj ?,
T1T2….

所以 next[j+1]首先＝k＝next[j]; 即…..Tj-1Tj ?,
T1T2…Tk-1.

若不相等，则 next[j+1]=next[k]; 即…..Tj-1Tj ?,
T1T2….Tnext[k]-1

直到找到这样的序列， 即…..Tj-1Tj ?,
T1T2 ...To

那么，next[j+1]=next[next[j]]=next[next[next[j]]]…..=o+1;

```
Void get_next(Sstring T,int next[])
{
    i=1;    next[1]=0;    j=0;//i 表示当前求的 next
    While(i<T[0])//总共求 T[0]-1 个
    {
        if(j=0 | | T[i]=T[j])
        {
          ++i;
          ++j;
          next[i]=j;
        }
        Else j=next[j];
    }
}
```

因为 next[ ] 在匹配过程中，若 T[j]=T[next[j]];那么当 S[i]不等于 T[j],
S[i]肯定也不等于 T[k= next[j]];
所以 S[i]应直接与 T[next[k]]比较，而我们通过将 next[j]修正为 nextval[j]=next[next[j]];这样能使比较更少。

```
Void    get_nextval(Sstring T,int nextval[])
{
   i=1;    nextval[1]=0; j=0;
```

```
    while(i<T[0])
    {
        if(j=0 || T[i]= T[j])
         {
                ++i;
                ++j;
                if(T[i]!=T[j])
                   nextval[i]=j;
                else
                   nextval[i]=next[j];
            }
         else
                j=nextval[j];
}
```

空格串是指__由空格字符（ASCII 值 32）所组成的字符串，其长度等于__空格个数____。

在模试匹配 KMP 算法中所用失败函数 f 的定义中，为何要求 $p_1p_2\cdots\cdots p_{f(j)}$ 为 $p_1p_2\cdots\cdots p_j$ 两头匹配的真子串？且为最大真子串？

失败函数（即 next）的值只取决于模式串自身，若第 j 个字符与主串第 i 个字符失配时，主串不回溯， 模式串用第 k（即 next[j]）个字符与第 i 个相比，有‘p1…pk-1’=‘pj-k+1…pj-1’，为了不因模式串右移与主串第 i 个字符比较而丢失可能的匹配，对于上式中存在的多个 k 值，应取其中最大的一个。这样，因 j-k 最小，即模式串向右滑动的位数最小，避免因右移造成的可能匹配的丢失。

第五章

线性表结构是数组结构的一个特例，而数组结构又是线性表结构的扩展。

之所以这么说是因为数组结构中若数组元素蜕化成原子了，那么就成了线性表了。

三种特殊矩阵的压缩方法：

（1）对称矩阵

i,j 从 1 开始，k 从 0 开始

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & \text{当i} \geq \text{j} \\ \dfrac{j(j-1)}{2}+i-1 & \text{当i} < \text{j}\end{cases}$$

（2）下（上）三角矩阵

i,j 从 1 开始，k 从 1 开始

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j & \text{当i} \geq \text{j} \\ 0 & \text{当i} < \text{j}\end{cases}$$

（3）3 带形矩阵

i,j 从 1 开始，k 从 1 开始

$$k=\begin{cases}3(i-1)+j-i+1 & 当 i-1\le j\le i+1\\ 0 & 否则\end{cases}$$

数组元素的地址计算

例如： **Loc(a00) = b**

Loc(ai0) = b+(ai0 前元素个数)•L=b+(i•n)•L

Loc(aij) = b+(aij 前元素个数)•L= b+[i×n+j]×L

有一个二维数组 A[0:8,1:5],每个数组元素用相邻的 4 个字节存储，存储器按字节编址，假设存储数组元素 A[0,1]的第一个字节的地址是 100，存储数组 A 的最后一个元素的第一个字节的地址是（ ① ）。若按行存储，则 A[3,5]和 A[5,3]的第一个字节的地址是（ ② ）和（ ③ ）。若按列存储，则 A[7,1]和 A[2,4]的第一个字节的地址是（ ④ ）和（ ⑤ ）。

解：显然 A[0,1]是该数组的第一个元素。数组共有 9×5＝45 个元素，最后一个元素前面共有 44 个元素，loc(A[8,5])=loc(A[0,1])+44*L=100+44*4=276； 若按行存储，A[3,5]前面的第 0 行，第 1 行，第 2 行肯定存满了。第三行有 A[3,1],A[3,2],A[3,3],A[3,4]在他前面，所以此时在 A[3,5]前面有 3×5＋4＝19 个元素，loc(A[3,5])=loc(A[0,1])+19*L＝100＋19×4＝176；同理 A[5,3]前面的第 0 行，第 1 行，第 2 行。。。。第 4 行肯定存满了。第 5 行有 A[5,1],A[5,2]在他前面，所以此时在 A[5,3]前面有 5×5＋2＝27 个元素，loc(A[5,3])=loc(A[0,1])+27*L＝100＋27×4＝208；若按列存储，A[7,1]前面的列肯定已经存满了，而 A[7,1]在最前面的列上，在同列且在他前面的有 A[0,1],A[1,1],A[2,1],...A[6,1],所以在 A[7,1]前面共有 0＋7 个元素 loc(A[7,1])=loc(A[0,1])+7*L＝100＋7×4＝128；同理 A[2,4]，在他前面的列 第 1 列第 2 列第 3 列已经存满，同列且在他前面的有 A[0,4],A[1,4],所以 A[2,4]前面共有 3×9＋2＝29 个元素 loc(A[2,4])=loc(A[0,1])+29*L＝100＋29×4＝216

广义表两种存储结构：

1. 头尾链（常用）

结点的定义：typedef enum{ATOM,LIST} ElemTag;

```
typedef struct GLnodes{
        Elemtag Tag;
        union{
            AtomType atom;
            Struct {Struct GLnode* hp,*tp } Ptr;
        }
        }*GList;
```

2. 扩展性链(不常用)

关于广义表的一些递归算法

1） 求广义表的深度（最大的嵌套的括号数）

空表的深度是 1，原子的深度为 0，用头尾链作为存储结构

```
int GlistDepth(Glist L)
{
    if(!L) return 1;
    else if(L->tag=ATOM) return 0;
    else
        max=0;
        for(pp=L;PP!=null;pp=pp->ptr.tp)
            {
              dep=GlistDepth(pp->ptr.hp);
              if(max<dep) max=dep;
            }
}
```

2)复制广义表

```
status copyGlist(Glist &T,Glist L)//把 L 复制到 T
{
  if(!L) T=null;
  else
    {
      if(!(T=malloc(sizeofGnode)))
        exit(OVERFLOW);
      T->tag=L->Tag;
      if(L->Tag=ATOM)
        T->atom=L->atom;
      else
        {
          copyGlist(T->ptr.hp,L->ptr.hp);
          copyGlist(T->ptr.tp,L->ptr.tp);
        }
    }
  return OK;
}
```

快速排序中的分治区间的策略的应用实例

下列程序段 search(a,n,k)在数组 a 的前 n(n>=1)个元素中找出第 k(1<=k<=n)小的值。这里假设数组 a 中各元素的值都不相同。

```
#define  MAXN  100
int  a[MAXN],n,k;
int  search_c(int a[],  int  n,  int  k)
{int  low,  high,  i,  j,  m,  t;
 k--,;low=0 ;high=n-1;
 do {i=low;  j=high ;  t=a[low];
```

```
        do{while  (i<j  && t<a[j])  j--;
            if  (i<j)  a[i++]=a[j];
            while  (i<j && t>=a[i])  i++
            if  (i<j)  a[j--]=a[i];
          }  while (i<j);//一次分割
         a[i]=t;
         if (1) i==k break  ;
          if (i<k) low= (2) i+1  ; else high= (3)i-1  ;
    }while(4)low<high  _;
    return(a[k]);
}
```

稀疏矩阵的两个算法：

矩阵的转置：

用了两个数组 num[ ],cpot[ ]

num[i]表示第 i 列有多少元素；cpot[i]表示第 i 列的第一个元素转置后的位置。

cpot[1]=1;

cpot[col]=cpot[col-1]+num[col-1];

```
status FastTrans(TSMatrix M,TSMatrix&T)
{
    T.mu=M.nu;T.nu=M.mu;T.tu=M.tu;
    if(T.tu=0) return OK;
    for(col=1;col<=M.nu;col++)
        num[col]=0;
    for(t=1;t<=M.tu;t++)
        num[M.data[t].j]++;
    cpot[1]=0;
    for(col=2;col<=M.nu;col++)
        cpot[col]=cpot[col-1]+num[col-1];
    for(p=1;p<=M.tu;p++)
      {
          col=M.data[p].j;
          q=cpot[col];
          T.data[q].i=M.data[p].j;
          T.data[q].j=M.data[p].i;
          T.data[q].value=M.data[p].value;
          ++cpos[col];
       }
     return OK;
}
```

第六章

结点的出度（**OD**）：结点拥有的非空子树数目。

结点的入度(**ID**)：指向结点的分支（或有向弧、指针）的数目。

树的度（**TD**）：树中结点出度的最大值。

结点的度：该结点的出度

例如 在下述结论中，正确的是（ D ）【南京理工大学 1999 一、4 （1 分）】

①只有一个结点的二叉树的度为 0； ②二叉树的度为 2； ③二叉树的左右子树可任意交换；④深度为 K 的完全二叉树的结点个数小于或等于深度相同的满二叉树。

A．①②③ B．②③④ C．②④ D．①④

有关二叉树下列说法正确的是（ B ）【南京理工大学 2000 一、11 （1.5 分）】

A．二叉树的度为 2 B．一棵二叉树的度可以小于 2

C．二叉树中至少有一个结点的度为 2 D．二叉树中任何一个结点的度都为 2

有序树和无序树：若树中任一结点的各子树从左到右有序，则该树为有序树（强调子树的次序），否则为无序树。（只强调各子树之间相对有序，而并不像二叉树那样，当只有一个子树是要么是左子树要么是右子树。而在有序树中，只有一个子树的有序树就不唯一了。）

例题：在下列情况中，可称为二叉树的是（ B ）

A．每个结点至多有两棵子树的树 B．哈夫曼树 C．每个结点至多有两棵子树的有序树 D．每个结点只有一棵右子树 E．以上答案都不对

C 错在有序树不一定是二叉树，有序只是子树的相对保持有序，并没有严格定义具体那颗子树就是第几颗子树。

森林（或树林）：m(m≥0) 棵互不相交的有序树的有序集合。

深度＝h(h≥1)的 K（K＞1）叉树至多有(K^H-1)/(K-1)个结点。

$$S=\sum_{i=1}^{h}(\text{第i层最大结点数})=\sum_{i=1}^{h}K^{i-1}=\frac{K^h-1}{K-1}$$

包含 n（n≥0）个结点的 K（K＞1）叉树的最小深度为：$\lceil \log_k(n(k-1)+1) \rceil$

证：设有 n 个结点的 K 叉树的深度为 h，若该树 h-1 层都是满的，即每层有最大结点数 K^I-1（1≤i≤h-1），且其余结点都落在第 h 层，则该树的深度达最小，如图 6.11 所示：

$$\frac{K^{h-1}-1}{K-1}<n\le\frac{K^h-1}{K-1}$$

或：(Kh-1 -1) ＜n(K-1)≤(K h-1)

即：Kh－1＜n(k-1)+1≤Kh

取对数：（h-1）＜logk(n(K-1)+1)≤h

因为 h 为正整数，所以：h=$\lceil \log_k(n(k-1)+1) \rceil$

含有 n(n ≥1)个结点的完全二叉树的深度 $h=\lfloor \log_2 n \rfloor+1$

n 个叶结点的非满的完全二叉树的高度是⌈log2n⌉+1。（最下层结点数>=2）。

设完全二叉树 BT 结点数为 n，结点按层编号。对 BT 中第 i 结点(1≤i≤n)，注意结点编号从 1 开始，在数组存储时也是从数组 1 开始,若题目已然确定从 0 开始，则在计算孩子父亲结点

时都需要重新变换一下。

有：

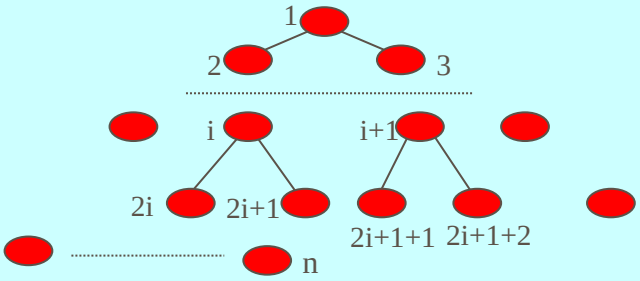
（1）若 i=1，则 i 结点(编号为 i 的结点)是 BT 之根，无双亲；否则(i＞1)，**parent(i)=** ，即 i 结点双亲的编号为 $\lfloor i/2 \rfloor$；

（2）若 2i＞n，则 i 结点无左子，否则 **Lchild(i)=2i**，即 i 结点的左子位于第 2i 号结点；

（3）若 2i+1＞n，则 i 结点无右子，否则 **Rchild(i)=2i+1**，即 i 结点的右子位于第 2i+1 号结点。

证明：采用数学归纳法，先证（2）和（3）。

设 n 个结点的完全二叉树如图所示。



i＝1时,显然 i 结点的左子编号为2，i的右子编号为2＋1＝3，除非2＞n , 3＞n 。

设对i结点，命题(2)、(3)成立，即Lchild(i)=2i，Rchild(i)=2i+1。根据按层编号规则，

i＋1时有：

Lchild(i+1)=(2i+1)+1=2(i+1)，除非2(i+1)＞n,

Rchild(i+1)=(2i+1)+1+1=2(i+1)+1，除非2(i+1)+1＞n，

故（2）、（3）得证。

再证（1），它可看作是(2)、(3)的推广。

因Lchild(j)=2j，所以Parent(2j)=j,令2j=i，有 Parent(i)=i/2= $\lfloor i/2 \rfloor$ (i/2为正整数)；

又：Rchild(j)=2j+1，所以Parent(2j+1)=j，令2j+1=i (i=3,5,7…)，有：

Parent(i)=(i-1)/2= $\lfloor i/2 \rfloor$ ，证毕。

例题：一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是(　　)【西安交通大学 1996 三、2 (3 分)】

A. 250　B. 500　C. 254　D. 505　E. 以上答案都不对 501

例题 1：由二叉树结点的公式：n=n0+n1+n2=n0+n1+(n0-1)=2n0+n1-1， 因为 n=1001,所以 1002=2n0+n1,在完全二叉树树中，n1 只能取 0 或 1,在本题中只能取 0，故 n=501，因此选 E。

例题 2：高度为 K 的完全二叉树至少有_2^{K-2}_个叶子结点。(刚好第 K 上只有一个叶子时，高度为 K，N＝$2^{K-2}-1+1=2^{K-2}$

例题 3：在顺序存储的二叉树中，编号为 i 和 j 的两个结点处在同一层的条件是

用顺序存储二叉树时，要按完全二叉树的形式存储，非完全二叉树存储时，要加“虚结点”。设编号为 i 和 j 的结点在顺序存储中的下标为 s 和 t ,则结点 i 和 j 在同一层上的条件是

⌊log2s⌋=⌊log2t⌋。

二叉树的存储结构

1．顺序存储结构

用一组连续的存储单元（一维数组）存储二叉树中元素，称为二叉树的顺序存储结构。描述如下：

```
#define    maxsige 1024    //二叉树结点数最大值//
typedef    datatype    sqtree [maxsize];
```

若说明 sqtree bt; 则(bt[o] bt[1] … bt[maxsize-1])为二叉树的存储空间。每个单元 bt[i] 可存放类型为 datatype 的树中元素。

2．链式存储结构

二叉结中结点的一般形式为：

遍历的递归算法

```
void preorder( BTptr T)          //对当前根结点指针为 T 的二叉树按前序遍历//
    {
        if (T)   { visit(T);       //   访问 T 所指结点  //
                   preorder(T–>Lchild);       //前序遍历 T 之左子树//
                   preorder(T–>Rchild);   //前序遍历 T 之右子树//
                 }
    }
void   Inorder (BTptr T)          //对当前根结点指针为 T 的二叉树按中序遍历//
    {
        if (T)   {       Inorder( T->Lchild);          //中序遍历 T 之左子树//
                         visit(T);              //访问 T 所指结点//
                         Inorder(T->Rchild);       //中序遍历 T 之右子树//
                 }
    }
  void postorder ( BTptr T)        //对当前根结点指针为 T 的二叉树按后序遍历//
    {
        if (T)   {       postorder(T–>Lchild);      //后序遍历 T 之左子树//
                         postorder(T–>Rchild);     //后序遍历 T 之右子树//
                         visit(T);                 //访问 T 所指结点//
                 }
    }
```

从上述三个算法中看出，若抹去 visit(T)语句，则三个算法是一样的，可以推断，这三个算法的递归路线是一致的(走的线路是一样的，只是对结点的访问时间不同而已，前序是第一次碰到就访问，中序是第一次返回时访问，后序是第二次返回时访问。

遍历非递归算法

1.前序遍历二叉树的非递归算法

算法思路：设一存放结点指针的栈 S。从根开始，每访问一结点后，按前序规则走左子树，

但若该结点右子树存在，则右子指针进栈，以便以后能正确地遍历相应放回到该右子树上访问。

算法描述：

```
void  Preoder-1(BTptr T)  //前序非递归遍历二叉树 T//
{     BTptr p；stacktype s;
      Clearstack(s);
      push(s,T); //置栈 S 为空、根指针 T 进栈//
      while (!Emptystack(s) )
       { p=pop(s);      //出栈,栈顶=>P//
         while (p)
          {
             visit (p);    //访问 p 结点//
             if (p–>Rchild)   push(s,p–>Rchild); //右子存在时,进栈//
             p=p–>Lchild;
          } //向左走//
       }
}
```

说明：内部循环是从 P 结点出发一直走到最左，走的过程中保存了每一个右子树的地址，（因为右子树还没有被访问）而且是先进后出的，即先保存的比后保存的更先被用作返回地址，所以是用栈。外循环正好是当内部循环不下去的时候，退一栈的情形。即换成他的右子树

2.中序遍历二叉树的非递归算法

算法思路：同前序遍历,栈 S 存放结点指针。对每棵子树(开始是整棵二叉树)，沿左找到该子树在中序下的第一结点(但寻找路径上的每个结点指针要进栈)，访问之；然后遍历该结点的右子树，又寻找该子树在中序下的第一结点，**……**直到栈 S 空为止。

算法描述：

```
void Inorder-1 (BTptr T)  //  中序非递归遍历二叉树 T//
{     BTptr   p; stacktype s;
      Clearstack(s);
      push (s,T);   //置栈 S 空、根指针进栈//
      while (!Emptystack(s))
      {
          while ((p=Getstop (s))&& p)   //  取栈顶且栈顶存在时//
                 push(s,p->lchild);          //p 之左子指针进栈//
          p=pop(s);        //去掉最后的空指针//
          if (!Emptystack (s))
          {    p=pop(s);    //取当前访问结点的指针=>P//
               visit(p);       //访问 P 结点//
               push(s,p-> Rchild); //遍历 P 之右子树//
          }
      }
}
```

说明：和前序不一样，这里的栈保存的是根结点的地址（因为中序遍历先访问左子树，而根结点没有被访问到。而前序遍历不一样，他一开始就访问根结点，所以他不保存根结点的地址而是保存右子树的地址，因为右子树还没有被访问。总之，用栈就是为了帮我们保存还没有被访问的地址，以便将来我们能找到返回的地址）

3.后序遍历二叉树的非递归算法

算法思路：后序非递归遍历较之前序、中序算法要复杂一些。原因是对一个结点是否能访问，要看它的左、右子树是否遍历完，所以每结点对应一个标志位—tag。tag=0，表示该结点暂不能访问；tag=1，表示该结点可以访问。其实是区分这次返回是遍历完左子树返回的还是遍历完右子树返回的，如果是左子树返回那么就不能访问根结点，如果是右子树返回的就能访问根结点。

当搜索到某 P 结点时，先要遍历其左子树，因而将结点地址 P 及 tag=0 进栈；当 P 结点左子树遍历完之后，再遍历其右子树，又将地址 P 及 tag=1 进栈；当 P 结点右子树遍历完后（tag=1），便可以对 P 结点进行访问。

栈元素类型：　typedef　struct

{BTptr q;　// 存放结点地址 //

int tag;　　//存放当前状态位//

}stype ;

算法步骤如下：

①若二叉树 T=∧（空树），返回，算法终止；

②初始化：置栈 S 空，根指针 T=>p；

③反复以下过程，直到 p=∧且栈 S=∧时算法终止：

若 p≠∧，(p,o)进栈，p=p–>Lchild (遍历左子树) ，……，直到 p=∧；出栈, 栈顶=>(p, tag)；若 tag=0，(p,1)进栈，p=p–>Rchild（遍历右子树），否则，访问 p 结点，并置 p=∧。

算法描述：

```
void postorder-1(BTptr　T)　　//后序非递归遍历二叉树 T//
{
    int tag;　BTptr p;　stype sdata;
    Clearstack (s);　　// 置栈空 //
    p=T;
    do {
        while (p)
        {
            sdata.q=p;　sdata.tag=0;
            push (s, sdata);　　// (p,0)进栈//
            p=p–>Lchild;　　//遍历 p 之左子树//
        }
        sdata=pop(s); //退栈
        p=sdata.q; //取指针
        tag=sdata.tag;//状态位//

        if (tag= =0)//从左子树返回时，根的 tag＝0
        {
            sdata. q=p;
            sdata.tag=1; //这时要进入根的右子树了，所以将根的 tag=1，
                         //下次碰到根时就可以访问了
            push(s,sdata);　// (p,1) 进栈,根还得进一次栈
            p=p–>Rchild;　//遍历右子树//
        }
```

```
                else //tag=1，这时说明右子树访问完了后返回，所以这次要对根访问了
                {
                    visit(p);  //访问 p 结点//
                    p=NULL;//要再退到上层，因为该棵树已经彻底访问完了
                }           //之所以把 p=null 是不让他进入 while
            }while ( p|| !Emptystack(s));
    }
```

例题：假设二叉树采用链式存储结构进行存储，root^为根结点，p^为任一给定的结点，写出求从根结点到 p^之间路径的非递归算法。

[题目分析]采用后序非递归遍历二叉树，栈中保留从根结点到当前结点的路径上所有结点。

```
void PrintPath(BiTree bt,p) //打印从根结点 bt 到结点 p 之间路径上的所有结点
{
  BiTree q=bt,s[]; //s 是元素为二叉树结点指针的栈，容量足够大
  int top=0; tag[];//tag 数组元素值为 0 或 1，访问左、右子树标志，tag 和 s 同步
  if (q==p)
  {  printf(q->data);
     return;
  } //根结点就是所找结点
  while(q!=null || top>0)
  {
       while(q!=null)  //左子女入栈，并置标记
         if（q==p）     //找到结点 p，栈中元素均为结点 p 的祖先
         {  printf(“从根结点到 p 结点的路径为\n”);
            for(i=1;i<=top;i++)
              printf(s[i]->data);
            printf(q->data);
            return;
         }
         else
         {   s[++top]=q;
             tag[top]=0;
             q=q—>lchild;
         } //沿左分支向下

       while(top>0 && tag[top]==1))
           top--; //本题不要求输出遍历序列，这里只退栈
       if (top>0)
       {   q=s[top];
           q=q->rchild;
           tag[top]=1;
       } //沿右分支向下
   }//while(q!=null || top>0)
}//结束算法 PrintPath
```

按层次遍历二叉树

算法描述：

```
void Layerorder(BTptr T)   //对二叉树 T 按层次遍历//
 {
    BTpfr p; qtype Q;
    if (T)
    {
        Clearqueue (Q);     //置队 Q 空//
        Enqueue (Q,T);     //将根指针进队//
        while (!Emptyqueue(Q) )
        {
            p=Dequeue(Q);     //出队，队头元素 ⇒p//
            visit (p);                  //访问 p 结点//
            if (p->Lchild) Enqneue (Q,p->Lchid);     //左子指针进队//
            if (p->Rchild) Enqneue (Q,p->Rchid);    //右子指针进队//
        }
    }
 }
```

遍历算法的应用

凡是对二叉树中各结点进行一次处理的问题，都可以用遍历算法来完成。

1．利用遍历算法对二叉树中各类结点计数

设二叉树中出度=0、1、2 的结点数分别为 **n0**、 **n1** 和 **n2** ，初值均为 0。

套用遍历算法（前序、中许、后序均可），扫描到树中某 **p** 结点时，若：

```
if ((p->Lchild==NULL)&&(p->Rchild==NULL))
    n0++;   //p 为叶子//
 else if((p->Lchild)&&(p->Rchild))
            n2++;            //p 为出度＝2 的结点//
             else n1++;     // p 为出度＝1 的结点//
```

如：只要把遍历算法在遍历时稍微改变一下。

```
n0=n1=n2=0;
void preorder( BTptr T)          //对当前根结点指针为 T 的二叉树按前序遍历//
{
     if (T)   {                  //   visit(T);   访问 T 所指结点  //
                  if ((T->Lchild==NULL)&&(T->Rchild==NULL))
                        n0++;   //p 为叶子//
                  else if((T->Lchild)&&(T->Rchild))
                        n2++;            //p 为出度＝2 的结点//
                  else
                        n1++;     // p 为出度＝1 的结点//

                  preorder(T–>Lchild);        //前序遍历 T 之左子树//
                  preorder(T–>Rchild);   //前序遍历 T 之右子树//
              }
```

}

2．前序遍历算法交换二叉树中各结点左、右子树(递归的交换)，对每一个结点都只需要一次访问就够了，所以用遍历算法稍微改一下，把 VISIT 改成我们要做的事就 OK 了

```
void preexchange (BTpfr T)
{
        BTptr p;
        if(T)
        {       p=T->Lchild;T->Lchild=T->Rchild;T->Rchild = p;//交换当前 T 的左右子树//
                preexchage(T->Lchild);      //处理左子树//
                preexchage(T->Rchild);      //处理右子树//
        }
}
```

例题 1：一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反，则该二叉树一定满足（　　）【南开大学 2000 一、2】

A．所有的结点均无左孩子 B．所有的结点均无右孩子 C．只有一个叶子结点 D．是任意一棵二叉树

前序序列是“根左右”，后序序列是“左右根”，若要这两个序列相反，只有单支树，所以本题的 A 和 B 均对，单支树的特点是只有一个叶子结点，故 C 是最合适的，选 C。A 或 B 都不全。

例题 2：某二叉树的前序序列和后序序列正好相反，则该二叉树一定是（）的二叉树。【武汉大学 2000 二、4】

A．空或只有一个结点　　B．任一结点无左子树　　C．高度等于其结点数　　D．任一结点无右子树

和上题类似的 BD 都可以但是是单选题就只能选 C 了

例题 3：在二叉树结点的先序序列，中序序列和后序序列中，所有叶子结点的先后顺序（　　）

A．都不相同　　B．完全相同　　C．先序和中序相同，而与后序不同

D．中序和后序相同，而与先序不同

二叉树的线索化

线索化分成，前序线索化和中序线索化后序线索化，他们的区别在于每个结点的前驱和后继的不同，各种不同的遍历得到不同的前驱和后继。如果考手画线索化二叉树，则三种都有可能，如果考算法描述，那么就只能考中序线索化二叉树了（其它两个的比较繁琐）。

我们讨论建立中序线索二叉树。

Lchild	Ltag	data	Rtag	Rchild

算法思路：利用中序递归遍历算法，即：

1 线索化根的左子树　2 线索化当前结点　　3 线索化根的右子树。

其中 **pre** 为外部指针(初值=NULL)，在算法运行中，**pre** 为当前搜索结点的前驱指针。

算法描述：

```
typedef struct Bnode
   { int Ltag,Rtag;   //左右特征位//
     datatype data;
     struct Bnode *Lchild,*Rchild ;
   }BTnode , *BTptr;
```

总控函数:中序线索化二叉树另外加了一个结点（相当于循环链表的头结点）。

```
Status InorderTheaing(BinThrtree& Thrt,BiThrTree T)
{
```

```
        if(!(Thr=(BinThrTree)malloc(sizeof(BiThrNode))))    exit(OVERFLOW);
        Thrt->LTag=Link;Thrt->RTag=Threak;
        Thrt->rchild=Thrt;//相当于空的循环链表，先将尾指针指向头结点。
        if(!T)    Thrt->lchild=Thrt;//空树那么整个也是空的了，只有一个附加的头结点了
        else
        {
            Thrt->lchild=T;
            pre=Thrt;//因为 pre 始终是当前结点的前驱结点，那么初始值就应该是头结点
            Inthreadbt（T）；将整个树线索化
            pre->RTag=Thread;//当整个树都线索化了那么，pre 肯定指向最后一个结点了
            pre->rchhild=Thrt;//所以 pre 的后继应该是头结点
        }
        return OK;
}

   void    Inthreadbt (BTptr T)     //中序线索二叉树//
      {
        if (T)
        {
            Inthreadbt(T->Lchild);//线索化左子树//
            visit(T);   改成                    if(T->Lchild==NULL)
                                                {
                                                    T->Ltag=Thread;
                                                    T->Lchild=pre;
                                                }
                                                if(pre ->rchild= =NULL)
                                                {     pre->RTag=Thread;
                                                    pre->Rchild=T;
                                                }
                                                pre=T;
          Inthreadbt(T->Rchild); //线索化右子树//
      }
}
```

线索化二叉树的遍历

有了线索的二叉链表那么遍历就方便多了，不需要借助栈也不需要用递归了，因为他已经把前驱后继都连起来了，而不像二叉树那样，走不动的时候就只能退栈了。 而且遍历速度快。

算法思路：先找到中序下的第一结点，访问之；若被访问的结点的右指针为线索指针，直接取其后继结点访问；**……**，直到被访问结点的右子树存在，再从相应右子起找中序下的第一结点，**……**，依次类推，直到整个树遍历完毕。

算法描述：

```
 BTptr Tinorder(BTptr Thrt)   //对中序线索二叉树的遍历//
   {
     BTptr p=T->lchild; //P  指向的是根结点
     while(p!=Thrt)     //循环链表没有到头结点
```

```
    {
        while(p->Ltag==Link) p=p->Lchild;   //找到中序下的第一结点//
        visit(p);
        while(p->Rtag==Thread&&p->Rchild!=Thrt)//如果右指针指向的是后继结点
        {    p=p->Rchild; //那么就大胆的访问吧，取 p 后继结点,访问之//
             visit(p);
        }
        p=p->Rchild;    // 如果不是后继结点，那么还得按照中序遍历的方法求得后继//
    }
  }
```

在中序线索二叉树中，每一非空的线索均指向其祖先结点。

在二叉树上，对有左右子女的结点，其中序前驱是其左子树上按中序遍历的最右边的结点（该结点的后继指针指向祖先），中序后继是其右子树上按中序遍历的最左边的结点（该结点的前驱指针指向祖先）。

非空二叉树中序遍历第一个结点无前驱，最后一个结点无后继，这两个结点的前驱线索和后继线索为空指针。

树的存储结构

1.双亲表示法

树中结点形式：

其中 **data** 域存放结点的数据值（意义同前）；**parent** 域为该结点之父结点的地址（或序号）。描述如下：

```
typedef struct tnode
{    datatype data;
     int parent ;
} PTnode ;
typedef struct
{     PTnode nodes[maxsize];   //树存储空间//
      int n ;                  //当前树的结点数//
}Ptree ;
```

2 孩子表示法

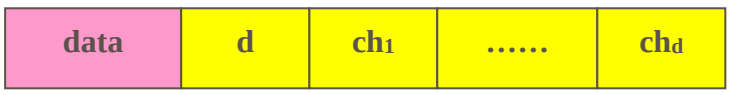
该表示法是采用链表结构来存储树的信息。

（1）固定指针数表示法：设树 T 的度为 d（d 叉树），即树中任一结点最多发出 d 个分支，

所以结点定义为：

（2）可变指针数表示法

结点形式：



其中 d 为本结点的出度，chi 为第 i 个孩子结点的指针。

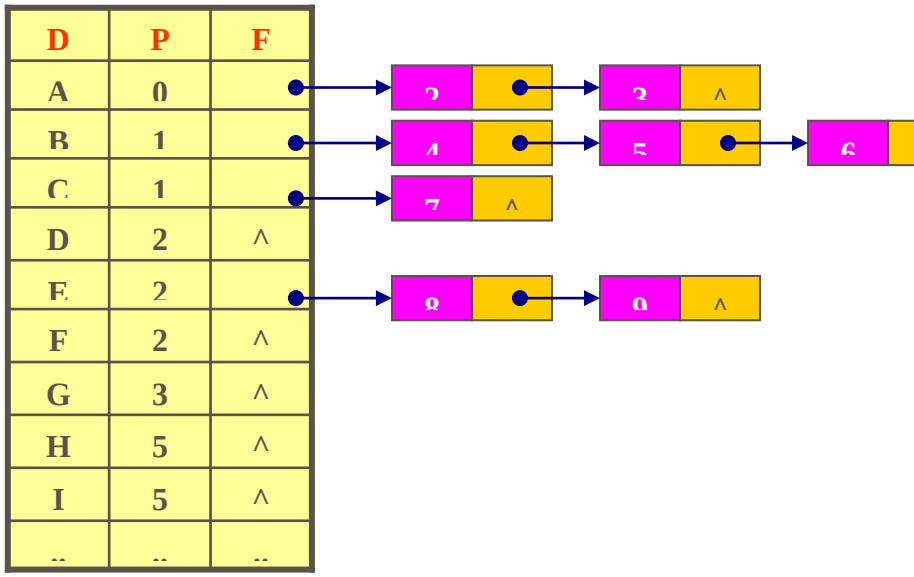
（3）孩子链表示法

该表示法将树中每一结点的诸孩子组成单链表，若树中结点数为 n，则有 n 个孩子链表(叶结点的链表为空)。又将 n 个链表的头结点组成头结点表。

头结点形式：

其中 data 域存放结点的数据值；parent 域为该结点之父结点的序号；fchild 为指向本结点第一个孩子的指针。

```
typedef struct CTnode              //链表结点//
    {    int child;
         struct CTnode * next;
    } *Childptr ;
typedef struct                     //头结点//
    {    datatype data; int parent;
         Childptr firstchild;
    } CTBox ;
typedef struct
    {    CTBox nodes[maxsize];      //头结点数组//
         int n ,root;   //n 为当前树中结点数，root 为根结点所在位置//
    }CTree;
```



3.孩子-兄弟表示法（或二叉树表示法

结点形式（同二叉树链式结构）：

其中 fchild 为指向本结点第一孩子的指针，而 nextsiling 为指向本结点右兄弟的指针。

1.树 T 转换成二叉树 BT（T⇒BT）

转换方法：对树 T 中每一结点，除保留第一孩子外，断开它到其它孩子的指针，并将各兄弟连接起来。转换后，原结点的第一孩子为左子，而原结点的右兄弟为其右子。

在转换成的二叉树中，根结点的右子一定为空

2 森林 F 转换成二叉树 BT(F⇒BT)

方法：先将 F 中各树转换成二叉树；然后各二叉树通过根的右指针相连。

3.二叉树 BT 恢复成森林 F（BT⇒F）

这是 F⇒BT 的逆变换。

方法：对 BT 中任一结点，其 Lchild 所指结点仍为孩子，而 Rchild 所指结点为它的右兄弟，即“左孩子，右兄弟”。

先序遍历森林 F

设 F={ T1,T2,……..,Tm }，其中 Ti(1≤i≤m)为 F 中第 i 棵子树。

方法：若 F≠ φ，则：

（1）访问 F 中 T1 之根；

（2）先序遍历 T1 之根下的各子树（子森林）；

（3）先序遍历除 T1 之外的森林（T2,……,Tm）。

显然(2)、(3)为递归调用，即：若子森林存在，仍按先序遍历方法对其遍历。

方法等价为：先将 F 转换二叉树 BT，然后对 BT 按前序（DLR）遍历，其结果是一样的。

后序遍历森林 F

方法：若 F≠ φ，则：

（1）后序遍历 F 中 T1 之根下的各子树（子森林）；

（2）访问 T1 之根；

（3）后序遍历除 T1 之外的森林{ T2,……,Tm }。

显然，(1)、(3)递归调用，即：若子森林存在，仍按后序遍历方法对其遍历。

此方法等价为：先将 F 转换成二叉树 BT，然后对 BT 按中序(LDR)遍历

例题：设 F 是一个森林，B 是由 F 变换得的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有（ C ）个。 我不明白的题

A. n-1 B. n C. n+1 D. n+2

Huffman 树

设 H 树中叶结点数为 n（即给定的权值个数），因 H 树中出度=1 的结点数为 0，出度=2 的结点数为 n-1，故 H 树中总的结点数 m=2n-1。因而可将树中全部结点存储在一个一维数组中。因而可将树中全部结点存储在一个一维数组中。

由此写出构造 H 树的 C 语言描述算法：

```
void   HuffmTree (huffm HT[m+1] )     //构造 H 树的算法//
{
        int i, j, p1, p2;   int w, s1, s2;
        for (i=1,i<=m,i++)                  //初始化//
          {
            HT[i]. wi=0;
            HT[i].parent=0;
            HT[i].Lchild=HT[i].Rchild=0;
          }
        for (i=1; i<=n; i++)
            scanf(“%d”,& HT[i]. wi );       //读入权值//
        for (i=n+1; i<=m; i++)   //进行 n-1 次循环，产生 n-1 个新结点，构造 H 树//
        {
            p1=p2=0;                        //p1、p2 为所选权值最小的根结点序号//
```

```
            s1=s2=max;                  //设 max 为机器能表示的最大整数//
            for(j=1;j<=i-1;j++)          //从 HT[1]~HT[i-1]中选两个权值最小的根结点//
              if (HT[j].parent==0)
                if (HT[j]. wi<s1)
                {       s2=s1; p2=p1;       //以 j 结点为第一个权值最小的根结点//
                        s1=HT[j].wi; p1=j;
                }
                else if (HT[j].wi<s2)
                {
                      s2=HT[j].wi ;
                      p2=j; //以 j 为第二个权值最小的根//
                 }

              HT[p1].parent=HT[p2].parent=i; //构造新树//
              HT[i].Lchild=p1; HT[i].Rchild=p2;
              HT[i].wi =HT[p1].wi +HT[p2].wi;       //权值相加送新结点//
        }
 }
```

求 Huffman 编码的算法：

```
 typedef struct
 {    char bits[n+1];          //存放一个字符的 Huffman 编码
      int start;                 //存放该字符编码的最后一个位置
      char ch;                   //待编码字符//
 }ctype;

void Huffmcode(ctype code[n+1])        //求 n 个字符的Huffman 编码的算法//
{
      int   i, j, p, s;   huffm   HT[m+1];      //H 树存储空间//
      ctype md;                                  //存放当前编码的变量//
       for (i=1;i<=n; i++)                      //读入待编码的字符//
           HT[i].data=code[i].ch=getchar( );
       //从叶子到根逆向求每个字符的哈佛满编码
      HuffmanTree(HT);                    //构造H 树//

      for (i=1;i<=n;i++)                 //求 n 个字符的 Huffman 编码//
      {
            md.ch=code[i].ch;
            md.start=n-1; //就算是单支树最多也就是  n-1
            s=i;            //第 i 个字符地址（或下标）⇒s//
            p=HT[i].parent;          //p 为 s 之父结点地址//
            while (p!=0)             //p 存在时//
            {
                md.start-- ;
                if (HT[p].Lchild= =s)
```

```
                md.bits[md.start]='0';      //左走一步为' 0 '//
            else
                md.bits[md.start]='1' ;     //右走一步为' 1 '//
            s=p;
            p=HT[p].parent; //求下一位//
        }
        strcpy(code[i],md) ;                //存入第 i 字符的编码//
    }
}
```

Huffman 译码

译码是编码的逆运算。设电文（二进制码）已存入字符型文件 fch 中，译码过程：根据编码时建造的H树和相应的 Huffman 编码，从H树的根（序号为 m）出发，逐个取电文中的二进制码，若当前二进制码＝" 0 "，则走左子，否则走右子，一旦到达H树的叶结点，取相应叶结点中字符 code[i].ch。重复上述译码过程，直到电文结束。算法如下：

```
void Transcode(HuffmTree HT[m+1],ctype code[n+1])
{   int i, chat c;   FILE *fp;
    if ((fp=fopen("fch","r"))==NULL)   Error(fch);
    //打开文件 fch，只读，文件指针 ⇒fp，打不开时出错处理//
    i=m;   //取H树根结点序号//
    while ((c=fgetc(fp))!=EOF)        //读入一个二进制码//
        {
            if(c= ='0')
                i=HT[i].Lchild;   //向左走//
            else
                i=HT[i].Rchild;              //向右走//
            if（HT[i].Lchild= =0）           //HT[i]为叶子//
            {   putchar (code[i].ch);        //输出译出的字符//
                i=m;
            }
        }
    fclose(fp);    //关闭文件 fch//
    if (HT[i].Lchild!=0) Error(HT); //电文结束 i 未达到叶结点，则电文有误//
}
```

第七章：

对于无向图，**e** 的范围是：$\left[0,\frac{1}{2}n(n-1)\right]$

数据结构中所讨论的图都是简单图，任意两结点间不会有双重的边。

对于有向图，e 的范围是：$[0,n(n-1)]$

图的各种存储结构

邻接矩阵很方便访问任意两点的边，但是不方便计算其邻接点。在深度和广度遍历中广泛的需要求某点的邻接点。所以邻接矩阵只在 Floyed 和 Prim 和 Dijstra 中采用。

邻接表能很方便的求某顶点的邻接点，索引对于与遍历有关的算法大多都采用邻接表。如深度、广度、拓扑排序、关键路径。但他也有不足的地方，就是不方便求入度或是那些点可以到他的操作。所以有人引进逆邻接表。最后人们把这两种表结合到一起就是十字链表和邻接多重表。一个是存储有向图，另一个是存储无向图。

在十字链表和邻接多重表很方便求邻接点的操作和对应的逆操作。所以实际应用中，凡是能用邻接表实现的一定能用十字链表和邻接多重表实现。并且它们的存储效率更高。

1. 邻接矩阵（有向图和无向图和网）又称为数组表示法

```
typedef struct
{       vextype vexs[maxn];       //顶点存储空间//
        adjtype A[maxn][maxn];    //邻接矩阵//
        int vexnum,arcnum;        //图的顶点数和边数
        GraphKind  Kind;         //图的类型
} mgraph;
```

2. 邻接表（有向图和无向图和网）

```
typedef struct node    //边
{  int adj; int w;        //邻接点、权//
   struct node  *next;  //指向下一弧或边//
}linknode;
typedef struct         //顶点类型//
{  vtype data;        //顶点值域//
   linknode *farc;   //指向与本顶点关联的第一条弧或边//
}Vnode;
typedef struct
{
   Vnode G[maxn];   //顶点表//
   int vexnum,arcnum;
   GraphKind kind;
}ALGraph;
```

边结点

adjvex	nextarc	info

顶点结点

data	firstarc

3. 十字链表（有向图和有向网）

边结点

headvex	taivex	hlink	tlink	info

顶点结点

data	firstin	firstout

4. 邻接多重表（无向图）

边结点

mark	ivex	jvex	ilink	jlink	info

顶点结点

data	firstedge

有向无环图（DAG）：是描述含有公共子式的表达式的有效工具。二叉树也能表示表达式，但是利用有向无环图可以实现对相同子式的共享，从而节省存储空间。

顶点的度：

无向图：某顶点 V 的度记为 D(V)，代表与 V 相关联的边的条数

有向图：顶点 V 的度 D(V)=ID(V)+OD(V)

强连通分量：在有向图中，若图中任意两顶点间都存在路径，则称其是强连通图。**图中极大强连通子图称之为强连通分量**

“极大”在这里指的是：往一个连通分量中再加入顶点和边，就构不成原图中的一个连通子图，即连通分量是一个最大集的连通子图。有向图的连通就是指该有向图是

强连通的

遍历图的过程实质上是 对每个顶点查找其邻接点的过程 其耗费的时间主要取决于采用的存储结构。当用邻接矩阵存储图时，查找每个顶点的邻接点所需的时间 O（n^2），其中 n 是图中顶点数。而当用邻接表存储图时，找邻接点的所需时间为 O（e），其中 e 为图中边的个数或有向弧的个数，由此，当以邻接表作为存储结构时，深度优先搜索遍历图的时间复杂度 O（n+e）.

广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同，两者的不同之处仅在于对结点访问的顺序不同。也就是说他们的时间复杂度都取决于说采用的存储结构，当用邻接矩阵存储时，复杂度为 O（n^2），当用邻接表存储时，时间复杂度为 O（n+e）.

建图的算法：（邻接表是常考的，邻接矩阵简单，十字链表和 多重表和建邻接表十分的相似）

```
void CreatGraph (AdjList &g) //建立有 n 个顶点和 m 条边的无向图的邻接表存储结构
  { int n,m;
    scanf("%d%d",&n,&m);//输入顶点数和边数
    for (i =1,i<=n;i++)//输入顶点信息,建立顶点向量
     {  scanf(&g[i].vertex);
        g[i].firstarc=null;
      }
    for (k=1;k<=m;k++)//输入边信息
     {  scanf(&v1,&v2);//输入两个顶点
        i= LocateVertex (g,v1); j=GraphLocateVertex (g,v2); //顶点定位
        p=(ArcNode *)malloc(sizeof(ArcNode));//申请边结点
        p->adjvex=j;
        p->next=g[i].firstarc; g[i].firstarc=p;//将边结点链入出边链表的头部
        p=(ArcNode *)malloc(sizeof(ArcNode));   //因为是无向图所以要在另外一个
        p->adjvex=i;
        p->next=g[j].firstarc;  g[j].frstarc=p;// 顶点的出边表中插入该结点
     }
  }//算法 CreatGraph 结束
```

两种求最小生成树的算法（Prim 和 Kruskal）

Prim 算法中有双重循环，外层是求 n－1 条边内层是在 closedge[v].lowcost 中求最小值和并列的求得当前加入点对 closedge[]的影响。所以他的时间复杂度是 O（n^2），它与途中边的数目没有关系，所以比较适合用在边比较稠密的图中。（顶点数相同，不管边数，都相同）

Kruskal 和他相对应，他的时间复杂度为 O（eloge），与图中的结点数目无关，至于边的个数有关。所以适合用在稀疏图中。（边数一定，不管顶点数，复杂度都相同）

求最小生成树的普里姆(Prim)算法中边上的权不可以为负，

```
typedef struct {
      VertexType adjvex;
      VRType lowcost;
   }closedge[MAX_VERTEX_NUM];
```

假设 cost(u,w)表示边(u,w)上的权值，则对于集合 V-U 中每个顶点 w，

```
    closedge[LocateVex(G, w)].lowcost = Min{cost(u,w)|u∈U}
  void MiniSpanTree_PRIM( MGraph G, VertexType u,SqList& MSTree)
  {
    // G 为数组存储表示的连通网，按普里姆算法从顶点 u 出发构
      k = LocateVex ( G, u );
      for ( j=0; j<G.vexnum; ++j )          // 辅助数组初始化
          if (j!=k) closedge[j] = { u, G.arcs[k][j].adj };//{adjvex,lowcost}
      closedge[k].lowcost = 0;              // 初始状态，U＝{u}
      for (i=1; i<G.vexnum; ++i)        // 选择其余 G.vexnum-1 个顶点
      {     k = minimum(closedge);              // 求出 T 的下一个结点(图中第 k 顶点)
          // 此时 closedge[k].lowcost =
          // Min{ closedge[vi].lowcost | closedge[vi].lowcost>0, vi∈V-U }
            printf(closedge[k].adkvex,   G.vexs[k]); //输出生成编
            closedge[k].lowcost = 0;            // 第 k 顶点并入 U 集
            for (j=0; j<G.vexnum; ++j)
                if (G.arcs[k][j].adj < closedge[j].lowcost) //新顶点并入 U 后重新选最小边
                    closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
      } // for
    } // MiniSpanTree
```

拓扑排序问题

```
Status ToplogicalSort(ALGraph G)
{
    FindIndegree(G,indegree);//求各点的入度放在 Indegree[vnum];
    InitStack(S);
    for(i=0;i<G.vexnum;++i)
        if(Indegree[i]= =0)
            push(S,i);
    count=0;
    while(!StackEmpty(S))
    {   Pop(S,i);       printf(i,G.vex[i].data);    ++count;
        for(p=G..vex[i].firstarc;  p;    p=p->nextarc)
        {   k=p->adjvex;
            Indegree[k]--;
            if( Indegree[k]= =0)   push(S,k);
        }//for
    }//while
    if(count<G.vexnum)
        return ERROR;
    else
        return OK
}
```

算法分析：求各顶点的入度的时间复杂度为 O（e），入度为零的点入栈 O（n），在循环中，每个顶点进一次栈，出栈一次，入度减 1 操作在 while 共执行了 e 次，所以总的时间复杂度为 O（n+e）.

当图中无环时，也可以利用深度优先遍历进行拓扑排序，因为图中无环，所以最先退出 DFS 函数的顶点即出度为零的点，是拓扑排序中最后一个顶点。由此，按 DFS 函数的先后记录下来的顶点序列即为逆向的拓扑有序序列。

Dijkstra 算法

首先引进一个辅助向量，　Dist[i]表示当前找到的从源点到 vi 的最短路径长度。
final[v]为 true，即已经求得从 v0 到 v 的最短路径。p[v][w]为 true，则 w 是从 v0 到 v 当前求得最短路径上的顶点。该算法弧上的权出现__负数__情况时，不能正确产生最短路径

```
void ShortestPath_DIJ( MGraph G,  int v0, PathMatrix&p,ShortPathTable& Dist )
{  // 用 Dijkstra 算法求有向网 G 从源点 u 到其余顶点的最短路径
    for (v=0; v<G.vexnum; ++v)
    {
      final[v] = FALSE; dist[v] = G.arcs[v0][v];
      for(w=0;w<G.vexnum;w++)      p[v][w]=FALSE;//设空路径
      if(dist[v]<INFINITY){p[v][v0]=true;p[v][v]=true;}
    }
    dist[v0] = 0; final[v0] = TRUE; // 初始化，顶点 v0 属于 S 集
    for (i=1; i<G.vexnum; ++i)       // 求从源点到其余顶点的最短路径顶点
    {
       min = INFINITY
       for(w=0;w<G.vexnum;w++)
          if(!final[w])   //w 在 V－S 中
              if(D[w]<min)
                  {v=w;min=D[w];}//w 顶点离 vo 更近
      final[v]=true;            //离 vo 最近的 v 加入 S 集
      for(w=0;w<G.vexnum;w++)            //更新当前最短路径及距离
          if(!final[w]&&(min+G.arc[v][w]<D[w]))
          {    D[w]=min+G.arc[v][w];
               p[w]=p[v];p[w][w]=true;//p[w]=p[v]+[w];
          }//if
      }//for
} // ShortestPath_DIJ
```

Floyed 算法:

```
void Floyd (mgraph G, int n)     //求网 G 中任意两点间最短路径的 Floyd 算法//
{    int i,j,k;   int D[ ][n],path[ ][n];
     //最短路径长度及最短路径标志矩阵，
     即 path[i][j]存放路径（vi…vj）上 vi 之后继顶点的序号//
     for (i=0;i<n;i++)   //初始化//
       for (j=0;j<n;j++)
        {  if (G.A[i][j]<max)
              path[i][j]=j;   //若<vi,vj>∈R,vi 当前后继为 vj//
           else
              path[i][j]=-1;       //否则为-1//
           D[i][j]=G.A[i][j];
        }
```

```
    for (k=0;k<n;k++)   //进行 n 次试探//
     for (i=0;i<n;i++)   //对任意的 vi，vj//
       for (j=0;j<n;j++)
         if (D[i][j]>D[i][k]+D[k][j])
           {   D[i][j]=D[i][k]+D[k][j];   //取小者//
               Path[i][j]=path[i][k];  //改 vi 的后继//
           }
    for (i=0;i<n;i++)    //输出每对顶点间最短路径长度及最短路径//
        for (j=0;j<n;j++)
        {      printf ("\n %d", D[i][j]);   //输出 vi 到 vj 的最短路径长度//
               k=path[i][j];   //取路径上 vi 的后继 vk//
               if (k==-1)
                     printf ("%d to %d no path \n",i,j);   //vi 到 vj 路径不存在//
              else
              {      printf ("(%d",i);   //输出 vi 的序号 i//
                     while (k!=j)   //k 不等于路径终点 j 时//
                     {   printf (",%d",k);   //输出 k//
                         k=path[k][j];    //求路径上下一顶点序号//
                     }
                  printf ("%d) \n",j);   //输出路径终点序号//
              }
        }
  }
```

深度优先搜索遍历（非递归的）

```
void  Traver(AdjList g,vertype v)
  //图 g 以邻接表为存储结构，算法从顶点 v 开始实现非递归深度优先遍历。
{   struct arc *stack[];
    visited[v]=1;
    printf(v);  //输出顶点 v
    top=0;
    p=g[v].firstarc;
    stack[++top]=p;
    while(top>0 || p!=null)
    {    while (p)
           if (p && visited[p->adjvex]) p=p->next;
           else
           {   printf(p->adjvex);
               visited[p->adjvex]=1;
               stack[++top]=p;
               p=g[p->adjvex].firstarc;
           }//else
        if (top>0) {p=stack[top--]; p=p->next; }
     }//while
}//算法结束。
```

以上算法适合连通图，若是非连通图，则再增加一个主调算法,其核心语句是

```
for (vi=1;vi<=n;vi++)
  if(!visited[vi])
     Traver(g,vi);
```

判断回路问题：（通常有向图的回路问题，无向图的回路比较繁琐）

两种判断有向图中有回路主要方法：

1：利用深度优先遍历

```
int visited[]=0; finished[]=0; //finished[i]=1 表示 i 结点的所有邻接点都访问完了
int flag=0;//回路的标记。有回路时值为 1
int DFS-travor(Adjlist g)
{   i=1;
    while(flag==0&&i<=n)
       if(visited[i]==0)
       {  dfs(g,i);
          finished[i]=1;
        }//if
}//总控程序，如果图有多个连通分量，分别进入每个连通分量
void  dfs(AdjList g,vertype v)
//以顶点 v 开始深度优先遍历有向图 g，顶点信息就是顶点编号.
{
       printf("%d",v); visited[v]=1;
       p=g[v].firstarc;
       while(p!=null)   //访问 v 的所有邻接点
       {  j=p->adjvex;
          if(visited[j]==1&&finished[j]==0) //如果在 v 的邻接点中存在 vj 有访问过
            flag=0 ; //的但 vj 的邻接点有没有全部访问完的，说明访问 v 是从 vj 那里
                     // 进来的，而现在 v 和 vj 有直接的边说明存在回边，
                     //那么就一定有回路了。（该结论只有在有向图中成立，在无向图中
                     // 不成立，因为在无向图中 vj 可能就是 v 刚刚进来的上一个结点，
                     //而这时在 v 中发现 Vj 也不奇怪，边是双向的，不能作为他是有回
                     //路的充分条件。而有向图边是单向的）
          else if(visited[j]==0)
          {  dfs(g,j);
             finished[j]=1;
          } //if
          p=p->next;
       }//while
} //dfs 结束
```

2.利用拓扑排序

在拓扑排序中，有一变量 count 记录访问到的结点数。在算法结束前判断一下

if(count<n) printf(“存在回路”)

求图的连通分量的个数

```
void  Count(AdjList g) //求图中连通分量的个数
{   int k=0 ;    visited[1...n]=0;
    for (i=1;i<=g.vexnum;i++ )
        if (visited[i]==0)
        {  printf ("第%d 个连通分量:\n",++k);
           dfs(i);
        }//if
  }//Count
  void  dfs(AdjList g,vextype v)
  {  visited[v]=1;
     printf ( "%3d",v); //输出连通分量的顶点。
      p=g[v].firstarc;
      while (p!=null)
      {
        if（visited[p->adjvex]==0）
           dfs(p->adjvex);
        p=p->next;
       }//while
  }// dfs
```

算法中 visited[]数组是全程变量，每个连通分量的顶点集按遍历顺序输出。这里设顶点信息就是顶点编号，否则应取其 g[i].vertex 分量输出。

第九章 查找

查找分成静态查找和动态查找，静态查找只是找，返回查找位置。而动态查找则不同，若查找成功，返回位置，若查找不成功，则要返回新记录的插入位置。也就是说，静态查找不改变查找表，而动态查找则会有插入操作，会改变查找表的。

不同的查找所采用的存储结构也不同，静态查找采用顺序表，而动态查找由于经常变动，所以用二叉排序树，二叉平衡树、B－和 B＋。

静态查找有，顺序查找，折半查找，分块查找(索引顺序查找）

顺序查找(Sequential Search)是最简单的一种查找方法。

算法思路

设给定值为 k，在表(R1 R2……Rn)中，从 Rn 即最后一个元素开始，查找 key=k 的记录。若存在一个记录 Ri（l≤i≤n）的 key 为 k，则查找成功，返回记录序号 i；否则，查找失败，返回 0。

算法描述

```
int sqsearch(sqlist r,keytype k)   //对表 r 顺序查找的算法//
{    int i;
     r.data[0].key=k;   //k 存入监视哨//
     i=r.len;   //取表长//
     while(r.data[i].key!=k)
        i--;   //顺序查找//
    return(i);
}
```

算法用了一点技巧：先将 k 存入监视哨，若对某个 i(≠0)有 r.data[i].key=k，则查找成功，返回 i；若 i 从 n 递减到 1 都无记录的 key 为 k，i 再减 1 为 0 时，必有 r.data[0].key=k，说明

$$\sum_{i=1}^{n} P_i C_i = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$$

查找失败，返回 i=0。

平均查找成功长度 ASL＝ $\frac{n+1}{2}$ ，而查找失败时，查找次数等于 n+l。

折半查找算法及分析

当记录的 key 按关系≤或≥有序时，不管是递增的还是递减的，只要有序且采用顺序存储。

算法描述

```
int Binsearch(sqlist r,keytype k)       //对有序表 r 折半查找的算法//
{     int low,high,mid;
      low=1;high=r.len;     //上下界初值//
      while(low<=high)          //表空间存在时//
      {     mid=(low+high)/2;     //求当前 mid//
            if (k==r.data[mid].key)
                  return(mid);     //查找成功，返回 mid//
            if (k<r.data[mid].key)
                  high=mid-1;      //调整上界，向左部查找//
           else
                  low=mid+1;   //调整下界，向右部查找//
      }
      return(0);                    //low>high，查找失败//
}
```

判定树：用来描述二分查找过程的二叉树。n 个结点的判定树的深度和 n 个结点的完全二叉树深度相同＝ $h=\lfloor \log_2 n \rfloor+1$ 。但判断树不一定是完全二叉树，但他的叶子结点所在层次之差不超过 1。所以，折半查找在查找成功时和给定值进行比较的关键字个数至多为 $h=\lfloor \log_2 n \rfloor+1$

ASL＝ $\frac{1}{n}((n+1)\log_2(n+1)-n)=\frac{n+1}{n}\log_2(n+1)-1$

分块查找算法及分析

分块查找(Blocking Search)，又称索引顺序查找（Indexed Sequential Search），是顺序查找方法的一种改进，目的也是为了提高查找效率。

1．分块

设记录表长为 n，将表的 n 个记录分成 b=$\lceil n/s \rceil$ 个块，每块 s 个记录（最后一块记录数可以少于 s 个），即：

（$\underbrace{R_1\cdots\cdots R_s}_{\text{块 1}}$ $\underbrace{R_{s+1}\cdots\cdots R_{2s}}_{\text{块 2}}$ …… $\underbrace{\cdots\cdots\ R_n}_{\text{块 b}}$）

且表分块有序，即第 i（1≤i≤b-1）块所有记录的 key 小于第 i+1 块中记录的 key，但块内记录可以无序。

2．建立索引

每块对应一索引项：

Keymax	Link

其中 Keymax 为该块内记录的最大 key；link 为该块第一记录的序号（或指针）。

3．算法思路　分块索引查找分两步进行：

(1)由索引表确定待查找记录所在的块；(可以折半查找也可顺序因为索引表有序)

(2)在块内顺序查找。（只能用顺序查找，块内是无序的）

5．算法分析

分块查找算法的时间复杂度为：　$ASL=Lb+Ls$　$Lb=\log_2(b+1)-1$　$Ls=\frac{s+1}{2}$

二叉排序树：对二叉排序树中序遍历，得到的肯定是递增的有序序列。所以插入和删除都要保证二叉排序树这一性质。

1．二叉排序树的建立

```
typedef struct Bsnode;   //二叉排序树结点//
{     Retype data;              //结点数据//
      struct Bsnode *Lchild,*Rchild;
}BSN,*BSP;       //结点及指针说明符//
BSP createBst( )    //从键盘文件读入记录，建立二叉排序树的算法//
{       BSP T,S;   keytype key;
        T=NULL;   //置空树，T 为二叉排序树根结点指针//
        scanf("%d",&key);   //读入第一个记录的 key//
        while(key!=0)             //设 key 以 0 为结束符//
        {     S=(BSP)malloc(sizeof(BSN));   //申请结点//
              S->data.key=key;   //存入 key//
              S->Lchild=S->Rchild=NULL;   //置 S 结点左、右子树为空//
              T=BSTinsert(T,S);   //将 S 结点插入到当前的二叉排序树 T 中//
              scanf("%d",&key); //读下一记录的 key//
        }
        return(T); //返回根指针//
  }
```

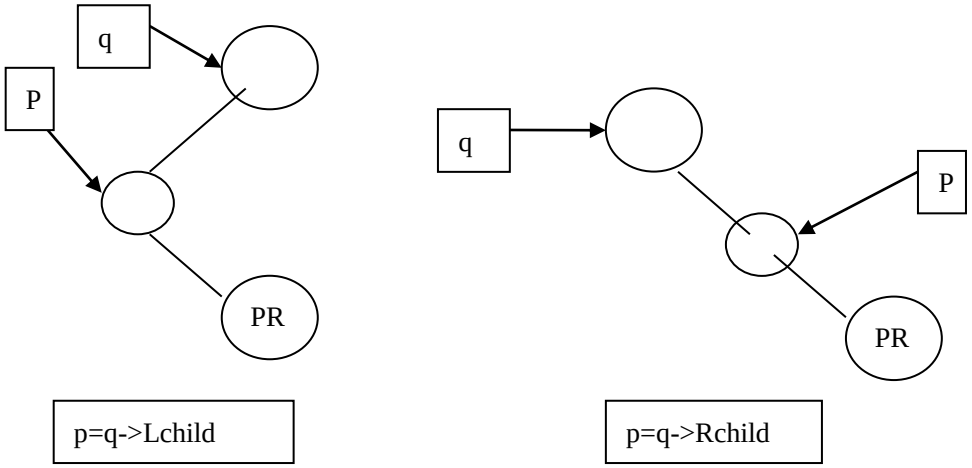
2．二叉排序树的插入新结点

```
BSP BSTinsert(BSP T,BSP S)
 //二叉排序树的插入算法，T、S 分别为根结点和待插入结点的指针//
{           BSP q, p;
            if (T==NULL)
                 return(S); //树为空时，以 S 为根//
            p=T; q=NULL;   //q 为 p 的父结点指针//
          while(p)   //寻找插入位置//
          {     q=p;
                if(S->data.key==p->data.key)
                {       free(S);
                        return(T);
                }     //S 结点已存在，返回//
                if(S->data.key<p->data.key)
                        p=p->Lchild;   //向左找//
                else
                        p=p->Rchild;   //向右找//
           }
         if(S->data.key<q->data.key)
              q->Lchild=S;   //S 为 q 的左子插入//
         else
              q->Rchild=S;   //S 为 q 的右子插入//
         return(T);
}
```

3．二叉排序树的删除结点

(1)**算法思路**：设指针 p 指向待删除结点，q 为 p 的父结点指针，分两种情况讨论如下：

①当 p 结点无左子树时，如**图**



其中 PR 为 p 结点的右子树(PR=空时，P 为叶结点)。此时删除操作只要令：

q->Lchild=p->Rchild；　　或　　q->Rchild=p->Rchild；

②当 p 结点的左子树 PL 存在时,有两种删除方法。

删除前：

二叉排序树中结点的删除

②当p结点的左子树P_L存在时，如**图8.8**：

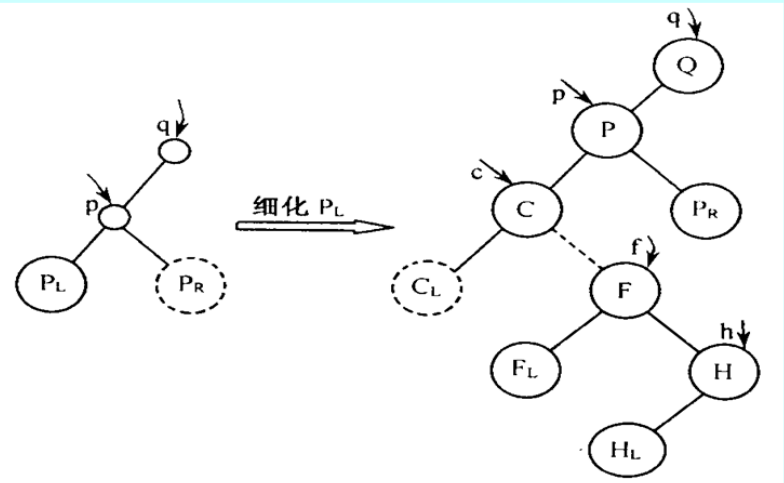


图8.8中二叉排序中序遍历序列为：

{…，C_L，C，…F_L，F，H_L，H，P，P_R，Q…}

删除p结点时，为保持树的二叉排序性，即让未删结点的中序遍历序列不变，一般有两种做法：

一种是令p结点左子树P_L为q结点的左子树，而p的右子树P_R挂在P_L的最右边。

其一是 直接删除 P，再将 P 的右子树接到新子树的最后下脚

二叉排序树中结点的删除

②当p结点的左子树P_L存在时，如图8.8：

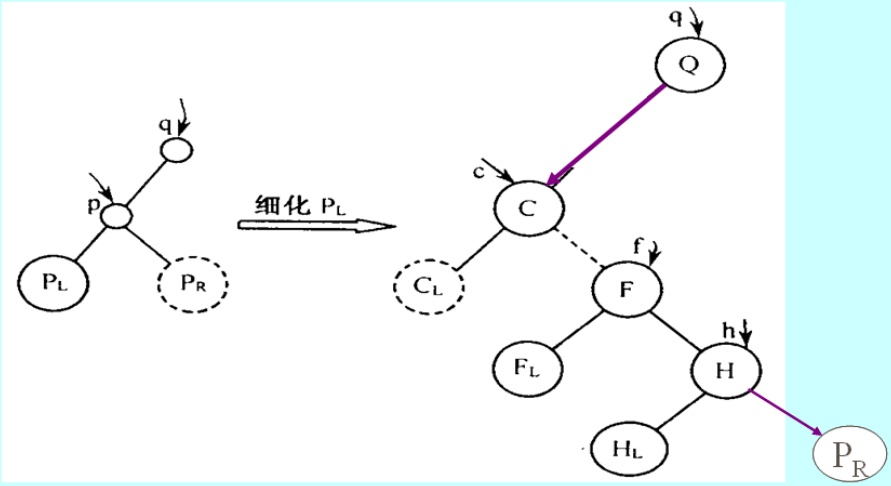


图8.8中二叉排序中序遍历序列为：

{…，C_L，C，…F_L，F，H_L，H，P，P_R，Q…}

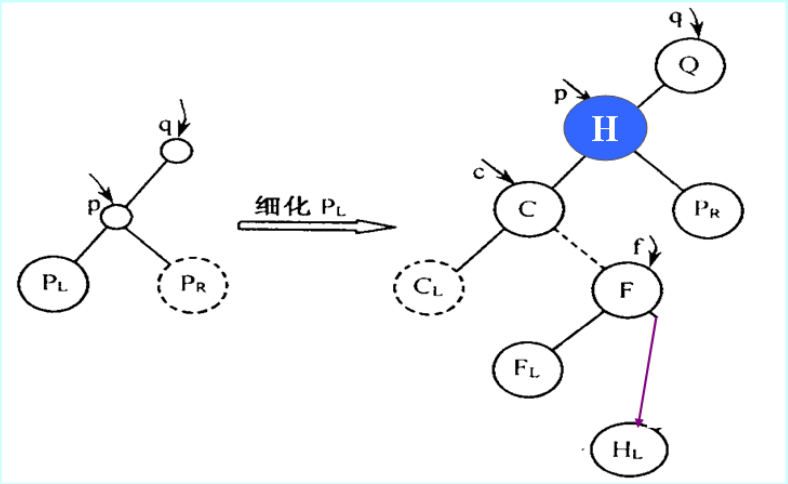
删除p结点时，为保持树的二叉排序性，即让未删结点的中序遍历序列不变，一般有两种做法：

一种是令p结点左子树P_L为q结点的左子树，而p的右子树P_R挂在P_L的最右边。

另一种：将 P 被最右下的元素替换而不用删除结点（用中序前驱替换）需要掌握的

二叉排序树中结点的删除

另一种是令p结点中序下的直接前驱h结点代替p结点（即删除p），然后抹去h结点，而原h结点的左子树H_L为f结点（f为h之父）的右子树。



两种删除方法的关键是如何找到p的直接前驱h结点。从图看出，h结点是P_L最右边的结点，即P_L中key值最大的结点。那么可以从p的左子c结点开始，一直向右搜索，直到有个结点的Rchild=∧为止，则该结点就是h结点。

（2）**算法描述**

```
Status BSTdelete(BSP t,keytype k)
 //在根指针为 t 的二叉排序树中，删除 key=k 的结点的算法//
{   BSP p,q,f,h;
    p=t;q=NULL;   //q 为 p 的父结点指针//
    while(p)              //寻找被删除结点//
     {   if(p->data.key==k)
              break;   //找到被删 p 结点，退出//
         q=p;
         if(k<p->data.key)
              p=p->Lchild;   //向左找//
         else
              p=p->Rchild;   //向右找//
     }
    if(p==NULL)
       return(t);   //若 k 不在树中，返回//
    if(p->Lchild==NULL)   //p 无左子树时//
    {     if(q==NULL)
            t=p->Rchild;   //p 为根,删除后,其右子为根//
          else if(q->Lchild==p)   //p 为 q 之左子时//
            q->Lchild=p->Rchild;
          else
            q->Rchild=p->Rchild;   //p 为 q 之右子时//
          free(p);
   }
   else //p 的左子树存在//
   {     f=p;h=p->Lchild;   //寻找 p 在中序下的直接前驱 h//
         while(h->Rchild)
         {   f=h;
             h=h->Rchild;
         }
         p->data=h->data;      //以 h 结点代替 p 结点，即删除 p 结点//
         if（f!=p）
             f->Rchild=h->Lchild;
         else
             f->Lchild=h->Lchild;
         free(h);
   }
   return(t);
}
```

二叉排序树查找**算法描述**

```
BSP BSTSearch(BSP t,keytype k)     //在二叉排序树 t 中，查找 key=k 的结点//
{   BSP p=t;
    while(p)
```

```
    {     if(p->data.key==k)
              break;   //查找成功，退出循环//
          if(k<p->data.key)
              p=p->Lchild;   //向左找//
          else
              p=p->Rchild; //向右找//
    }
  return(p);
} //查找成功时返回的是对应的位置，不成功时是插入位置。
```

平衡二叉树(AVL)

平衡因子 BF(Balance Factor)：BF=HL−HR

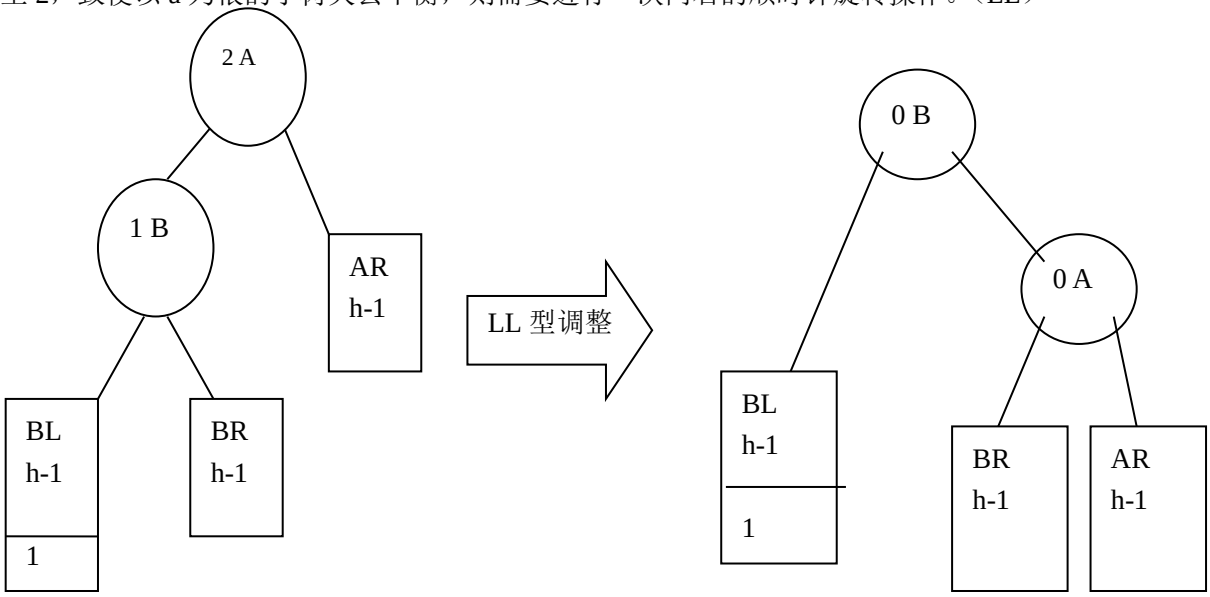
树的结点形式：

Lchild	BF	data	Rchild

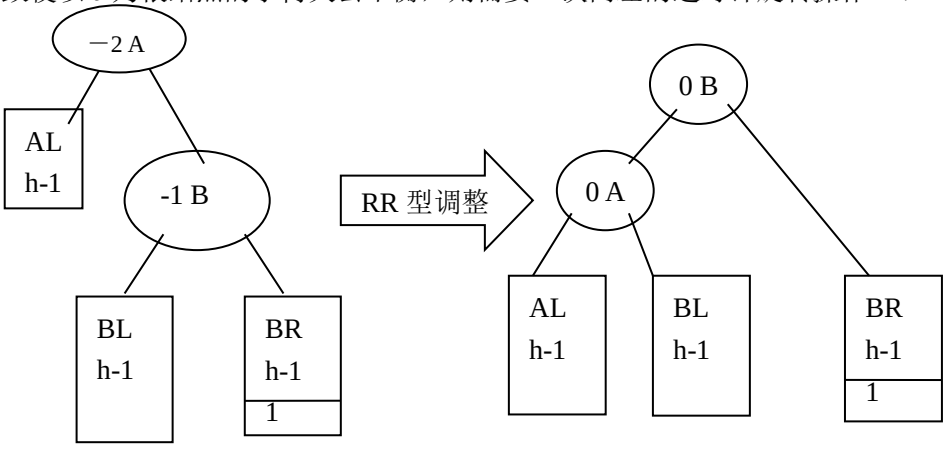
平衡二叉排序树 :若在构造二叉排序树的同时，使其始终保持为 **AVL** 树，则此时的二叉排序树为平衡的二叉排序树。

4 中调整方法：设指针 a 是失去平衡的最小子树根。

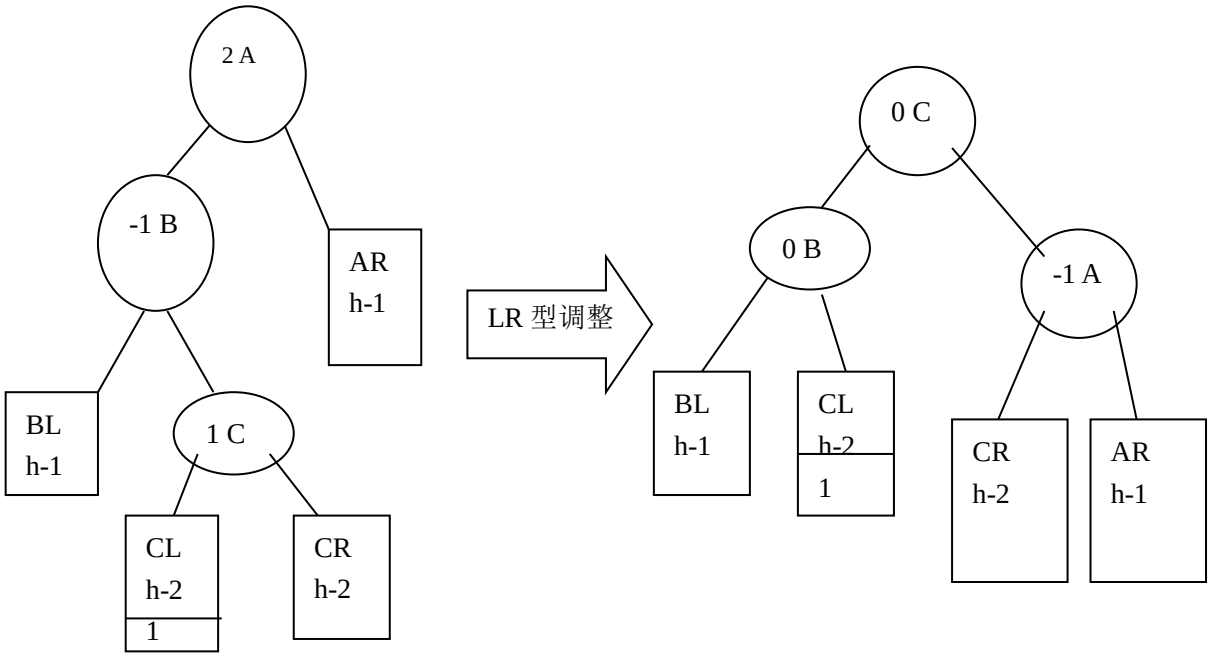
（1）单向右旋平衡处理：由于在 a 的左子树根结点的左子树插入结点，a 的平衡因子由 1 增至 2，致使以 a 为根的子树失去平衡，则需要进行一次向右的顺时针旋转操作。（LL）



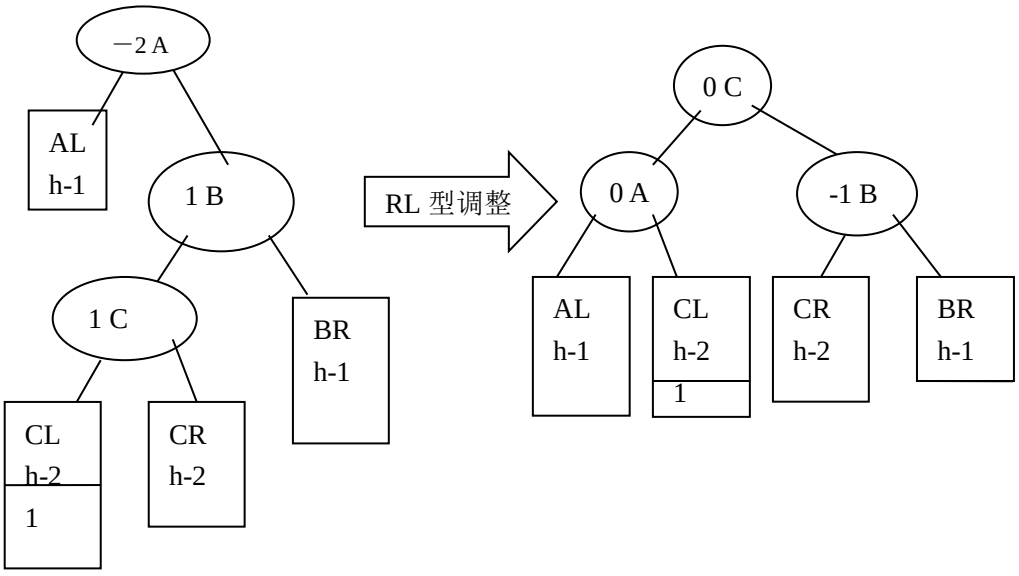
（2）单向左旋平衡处理：由于在 a 的右子树根结点的右子树插入结点，a 的平衡因子由－1 变成－2，致使以 a 为根结点的子树失去平衡，则需要一次向左的逆时针旋转操作。（RR）



（3）先左后右平衡处理：由于在 a 的左子树根结点的右子树上插入结点，a 的平衡因子由 1 增至 2，致使以 a 为根结点的子树失去平衡，需要进行两次旋转（先左旋后右旋）操作（LR）



(4)先向右后向左平衡处理：由于在 a 的右子树根结点的左子树上插入结点，a 的平衡因子由－1 变成－2，致使以 a 为根结点的子树失去平衡，则需要进行两次旋转（先向右再向左）操作。



B-树：是一种平衡的多路查找树。一棵 M 阶的 B－树，或者为空树，或满足

(1)树中的每个结点至多有 M 棵子树；

(2)若根结点不是叶子结点，则至少有两颗子树

（3）除根之外的所有非终端结点至少有$\lceil m/2 \rceil$ 棵子树。

（4）所有结点的关键字的个数比他的子树个数少 1

一棵 3 阶 B-树（又称 2-3 树）。

掌握 B-树的插入与删除

哈希表：**处理冲突的方法**

1．开放地址法

（1）di=1，2，3，……(m-1)——称为线性探查法；

（2）di= 1^2，-1^2，2^2，-2^2……——称为二次探查法。

2．链地址法

发生冲突时，将各冲突记录链在一起，即同义词的记录存于同一链表。

装填因子定义：$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$，

一般，设 Hash 表的装填因子为 α，采用线性、二次探查法及链地址法解决冲突时，查找成功的平均查找长度分别用公式表示如下：

线性探查法：

$$ASL_1 \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$$

二次探查法：

$$ASL_2 \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

链地址法：

$$ASL_3 \approx 1+\frac{\alpha}{2}$$

第十章 内部排序（在内存中进行的排序不需要访问外存的）外部排序（排序量很大，通过分批的读写外存，最终完成排序）

稳定排序和非稳定排序：看相同记录的相对次序是否回发生改变。主要看在排序过程中的比较是不是相邻记录，如果是相邻比较，一定是稳定的排序。如果不是相邻的比较，就是不稳定的。

内排序方法 截止目前，各种内排序方法可归纳为以下五类：

（1）插入排序；（2）交换排序；（3）选择排序；（4）归并排序； （5）基数排序。

插入排序：包括直接插入排序和希尔排序

直接插入排序：（稳定的）

算法描述

```
void Insort (sqList &L)   // 对顺序文件 F 直接插入排序的算法 //
{   int i,j;
    for (i=2;i<=L.len;i++)     // 插入 n－1 个记录 //
    {
        if(L.R[i].key<L.R[i-1])
        {    L.R[0]=L.R[i];      // 待插入记录先存于监视哨 //
             L.R[i]=L.R[i-1];
             for(j=i-2;L.R[0].key<L.R[j];j--)
                  L.R[j+1]=L.R[j];     // 记录顺序后移 //
             L.R[j+1]=L.R[0];   // 原 R[i]插入 j+1 位置 //
         }
     }
```

}

算法分析

设排序中 key 比较次数为 C，C 最小时记为 Cmin，最大时记为 Cmax。

（1）当原来就有序(正序)时，每插入一个 R[i]只需比较 key 一次，即：

$$C_{\min}=\sum_{i=2}^{n}1=n-1=O(n)$$

（2）当原本逆序（key 从大到小）时，每插入一个 R[i]要和子表中 i－1 个 key 比较，加上同自身 R[0]的比较，此时比较次数最多，即：

$$C_{\max}=\sum_{i=2}^{n}i=\tfrac{1}{2}(n+2)(n-1)=O(n^2)$$

(3)记录总的移动次数 m（m 最小时记为 mmin，最大时记为 mmax）

正序时，子表中记录的移动免去，即：

$$m_{\min}=0$$

逆序时，插入 R[i]牵涉移动整个子表。移动次数为 2+(i－1)=i+1，此时表的移动次数最大，即：

$$m_{\max}=\sum_{i=2}^{n}(i+1)=\tfrac{1}{2}(n+4)(n-1)=O(n^2)$$

排序的时间复杂度取耗时最高的量级，故直接插入排序的 **T(n)=O(n2)。**

Shell（希尔）排序又称“缩小增量”排序(不稳定的)

交换类的排序：（起泡排序和快速排序）

起泡排序算法描述

```
void Bubsort (sqList &L)
{    int i,flag;   //flag 为记录交换的标记//
     Retype temp;
     for (i=L.len;i>=2;i--)     //最多 n－1 趟排序//
     {    flag=0;               //记录每一趟是否发生了交换
          for (j=1;j<=i-1;j++)   //一趟的起泡排序//
               if (L.R[j].key>L.R[j+1].key)   //两两比较//
               {      temp=L.R[j];     //R[j] ⇔ R[j+1]//
                      L.R[j]=L.R[j+1];
                      L.R[j+1]=temp;
                      flag=1;
               }
          if (flag==0)  break;   //无记录交换时排序完毕//
     }
}
```

算法分析

设待排长度为 n，算法中总的 key 比较次数为 C。若正序，第一趟就无记录交换，退出循环，Cmin=n－1=O(n)；若逆序，则需 n－1 趟排序，每趟 key 的比较次数为 i－1（2≤i≤n），故：

$$C_{\max}=\sum_{i=n}^{2}(i-1)=\sum_{i=2}^{n}(i-1)=\tfrac{1}{2}n(n-1)=O(n^2)$$

同理，记录的最大移动次数为：

$$m_{\max}=\sum_{i=2}^{n}3(i-1)=\tfrac{3}{2}n(n-1)=O(n^2)$$

故起泡排序的时间复杂度 **T(n)=O(n2)**。并且是稳定的。

快速排序：（不稳定的，时间复杂度 **O(nlogn)**），不需要辅助空间，但有最好和最差之分

分割算法

```
int Partition(Sqlist&L,int low,int high)
{   L.R[0]=L.R[low];
    pivotkey=L.R[low].key;
    while(low<high)
    {   while(low<high&&L.R[high].key>=pivotkey)
            --high;
        L.R[low]=L.R[high];
        while(low<high&&L.R[low].key<=pivotkey)
            ++low;
        L.R[high]= L.R[low];
    }
    return low;
}
```

总控函数：

```
void QSort(Sqlist&L,int low,int high)
{   if(low<high)
    {
        pivotloc=Partition(L,low,high);
        QSort(L,low,pivotloc-1);
        QSort(L,pivotloc+1,high);
    }
}
```

调用方法：QSort(L,1,L,lenght);

算法分析：

若 原本就正序或逆序，每次调用一次后，记录数只减少了一个，故此时 T(n)=C(n+(n－1)+……+1)=O(n2)。这是快速排序效率最差的情况。所以快速排序算法有待改进。

简单选择排序： 属于稳定排序时间复杂度（O（n2））

算法描述

```
void Slectsort (Sqlist& L)     // 直接选择排序的算法 //
{
   for (i=1;i<L.len;i++)      // 选择 n－1 趟 //
   {
      j=SeloctMinkey(L,i); //从 i 到 L.len 中选择 key 最小的记录
      if (i!=j)
      {   temp=L.R[i];
          L.R[i]=L.R[j];
          L.R[j]=temp;
       }
```

```
    }
}
```

堆排序：属于选择排序 不稳定，时间复杂度（O（nlogn）），没有最好和最差的分别，也需要辅助的栈空间

若 ki≥k2i、ki≥k2i+1。此时，根结点 k1 的值最大，称为“大根堆”。

若 ki≤k2i、ki≤k2i+1 满足“”关系，则 k1 最小，称为“小根堆”。

在堆排序中，我们采用的是大根堆，原因是大根堆中，根最大，对删除很方便，直接把它与最后一个叶子结点交换就可以了。

记录 key 集合 k={k1 k2……kn}， 排序 分两步进行：

（1）将{k1 k2……kn}建成一个大根堆；

（2）取堆的根（key 最大者），然后将剩余的（n－1）个 key 又调整为堆，再取当前堆的根（key 次大者），……，直到所有 key 选择完毕。

一个元素的堆调整算法：

```
//已知 H.R[s...m]除 H.R[s]之外，均满足堆的定义，本函数只是将 H.R[s]放到已经是堆的堆中
void HeapAdjust (SqList& H, int s, int m)   //将（H.R[s]……H.R[m]）调整成大根堆//
{
    rc=H.R[s];   //暂存 H.R[s]//
    for(j=2*s;j<=m; j*=2 )//沿 key 较大的孩子结点向下筛选
    {
        if (j<m&&L.R[j+1].key>L.R[j].key)   j++;  // 令 j 为 s 的左右孩子 key 最大者的序号
        if (rc.key>L.R[j].key)
            break;//说明 H.R[s]已经比当前堆中的所有元素大，已经是堆了，不需要调整了
        L.R[s]=L.R[j];     //把最大的放到根结点
        s=j;               //把 s 与 j 交换，使得 s 与下层的堆比较
    }
    L.R[s]=rc;     //最初的根回归//
}

void Heapsort (SqList& H)     //  堆排序算法//
{
        //初始建堆
        for (i=L.len/2; i>=1; i--)   //从完全二叉树的最后一个非叶子结点开始
            HeapAdjust (L,i,L.len);     //调整（R[i]……R[n]）为堆//
       //每次取下根（最大的元素）所谓的取下，只是放到数组最后，改变一下数组的下界
        for (i=L.len;i>=2;i--)   //总共摘 n－1 次//
        {     temp=F.R[1];          //根与当前最后一个结点互换//
            F.R[1]=F.R[i];
            F.R[i]=temp;
            HeapAdjust (L ,1,i-1);     //把最后一个元素调整到合适的位置//
        }
}
```

二路归并排序：（稳定的，时间复杂度 O(nlogn）又稳定又效率高，但需要辅助空间 TR[1....n]

二路归并得核心操作是将一维数组中前后相邻的两个有序序列归并为一个有序序列

(注意这就是线型表中我们常考的那个，两个有序链表合成一个新的有序表)

算法描述如下：

```
void Merge(RcdType SR[],RcdType&TR[],int i,int m,int n)
//将有序的 SR[i...m]和 SR[m+1....n]归并为有序的 TR[i....n]
{
    for(j=m+1,k=i; i<=m&&j<=n; ++k)     //谁小就先将谁放进 TR
    {   if(SR[i].key<=SR[j].key)
            TR[k]=SR[i++];
        else
            TR[k]=SR[j++];
    }
    if(i<=m)                            //将剩余的 SR  直接放到 TR
        TR[k....n]=SR[i...m];
    if(j<=n)                            //将剩余的 SR  直接放到ＴＲ
        TR[k...n]=SR[j....n];
}

void MSort(RcdType SR[], RcdType&TR1[], int s, int t)
{   //将 SR[s...t]归并排序为 TR1[s...t]
    if(s= =t)  //只有一个元素
        TR1[s]=SR[s];
    else
    {
        m=(s+t)/2;//将 SR[]平分为两半
        MSort(SR, TR2, s, m);
        MSort(SR, TR2, m+1, t);
        Merge(TR2, TR1, s, m, t);
    }
}
```