考研计算机专项精讲课程讲义

数据结构

主讲：崔巍

欢迎使用新东方在线电子教材

目录

数据结构

绪论

0.1 基本概念

1、数据结构

数据结构是指互相之间存在着一种或多种关系的数据元素的集合。

数据结构是一个二元组 Data_Structure ＝（D，R），其中，D 是数据元素的有限集，R 是 D 上关系的有限集。

2、逻辑结构：是指数据之间的相互关系。通常分为四类结构：

（1）集合：结构中的数据元素除了同属于一种类型外，别无其它关系。

（2）线性结构：结构中的数据元素之间存在一对一的关系。

（3）树型结构：结构中的数据元素之间存在一对多的关系。

（4）图状结构：结构中的数据元素之间存在多对多的关系。

3、存储结构：是指数据结构在计算机中的表示，又称为数据的物理结构。通常由四种基本的存储方法实现：

（1）顺序存储方式。数据元素顺序存放，每个存储结点只含一个元素。存储位置反映数据元素间的逻辑关系。存储密度大。但有些操作（如插入、删除）效率较差。

（2）链式存储方式。每个存储结点除包含数据元素信息外还包含一组（至少一个）指针。指针反映数据元素间的逻辑关系。这种方式不要求存储空间连续，便于动态操作（如插入、删除等），但存储空间开销大（用于指针），另外不能折半查找等。

（3）索引存储方式。除数据元素存储在一组地址连续的内存空间外，还需建立一个索引表，索引表中索引指示存储结点的存储位置（下标）或存储区间端点（下标）。

（4）散列存储方式。通过散列函数和解决冲突的方法，将关键字散列在连续的有限的地址空间内，并将散列函数的值解释成关键字所在元素的存储地址。其特点是存取速度快，只能按关键字随机存取，不能顺序存取，也不能折半存取。

0.2 算法和算法的衡量

1、算法是对特定问题求解步骤的一种描述，是指令的有限序列。其中每一条指令表示一个或多个操作。

算法具有下列特性：⑴有穷性⑵确定性⑶可行性⑷输入⑸输出。

算法和程序十分相似，但又有区别。程序不一定具有有穷性，程序中的指令必须是机器可执行的，而算法中的指令则无此限制。算法代表了对问题的解，而程序则是算法在计算机上的特定的实现。一个算法若用程序设计语言来描述，则它就是一个程序。

2、算法的时间复杂度：以基本运算的原操作重复执行的次数作为算法的时间度量。一般情况下，算法中基本运算次数 T(n)是问题规模 n（输入量的多少，称之为问题规模）的某个函数 f(n)，记作：

T(n) ＝ O(f(n))

也可表示 T(n) ＝m(f(n))，其中 m 为常量。记号“O”读作“大 O”，它表示随问题规模 n 的增大，算法执行时间 T(n)的增长率和 f(n)的增长率相同。

注意：有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。

常见的渐进时间复杂度有：$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)$。

3、算法的空间复杂度：是对一个算法在运行过程中临时占用的存储空间大小的量度。只需要分析除输入和程序之外的辅助变量所占额外空间。

原地工作：若所需额外空间相对于输入数据量来说是常数，则称此算法为原地工作，空间复杂度为 O(1)。

习题

1、以下算法的时间复杂度是（ ）。

```
void f (int n){
    int x=1;
    while (x<n)
        x=2*x;
}
```

A.O(log2n) B. O(n) C. O(nlog2n) D. O(n2)

分析：基本运算是语句 x=2*x，设其执行时间为 T(n)，则有 2T(n)≤n， 即 T(n)<log2 n=O(log2 n)。

解答：A。

2、 以下算法的时间复杂度是（ ）。

```
void f (int n){
    int p=1,d=n,f=n;
    while (d>0){
        if (d%2= =1)  p=p*f;
        f=f*f; d=d/2;
    }
}
```

A.$O(\log_2 n)$ B. O(n) C. $O(n\log_2 n)$ D. O(1)

分析：算法中 while 循环的 if 条件中包含的 p=p*f 语句可以不考虑，因为它执行的次数不超过 d=d/2 语句的执行次数。

基本运算是语句 d=d/2（或 f=f*f），设其执行时间为 T(n)，则有 $d=n/2^{T(n)}>0\geq 1$，$2^{T(n)}\leq n$， 即 $T(n)\leq \log_2 n=O(\log_2 n)$。

解答：A。

3.求出下列算法的时间复杂度。

1) 比较同一简单类型的两个数据 x1 和 x2 的大小，对于 x1>x2,x1=x2 和 x1<x2 这三种不同情况分别返回‘>’、‘=’和‘<’字符。 char compare(SimpleType x1,SimpleType x2) {

```
    if (x1>x2) return‘>’;
  else if (x1==x2) return ‘=’;
  else return‘<’;
}
```

其时间复杂度为 $O(1)$。

2) 将一个字符串中的所有字符按相反方向的次序重新放置。

```
    void Reverse(char *p)       {
  int n=strlen(p);
  for (int i=0; i<n/2; i++) {
            char ch;
    ch=p[i];
      p[i]=p[n-i-1];
      p[n-i-1]=ch;
  }
}
```

其时间复杂度为 $O(n)$。

3) 从二维整型数组 a[m][n]中查找出最大元素所在的行、列下标。

```
    void Find(int a[M][N],int m,int n,int  &Lin,int  &Col)    { //M 和 N 为全局常量，应满足
M>=n 和 N>=n 的条件，Lin 和 Col 为引用形参，它是对应实参的别名，其值由实参带回
  Lin=0;   Col=0;
  for (int i=0;i<m;i++)
        for (int j=0;j<n;j++)
            if (a[i][j]>a[Lin][Col])   {
                Lin=i;   Col=j;
            }
}
```

其时间复杂度为 $O(m*n)$。

4)

```
void func(int n)     {
      int y = 0;
            while (y * y <=n)
                  y ++;
   }
```

其时间复杂度为 $O(n^{1/2})$。

第一章 线性表

1.1 线性表的定义

线性表是一种线性结构，在一个线性表中数据元素的类型是相同的，或者说线性表是由同一类型的数据元素构成的线性结构，定义如下：

线性表是具有相同数据类型的n(n≥0)个数据元素的有限序列，通常记为：

$$(a_1, a_2, \ldots a_{i-1}, a_i, a_{i+1}, \ldots a_n)$$

其中n为表长， n＝0 时称为空表。

需要说明的是：a_i为序号为 i 的数据元素（i=1,2,…,n），通常将它的数据类型抽象为ElemType，ElemType根据具体问题而定。

1.2 线性表的实现

1.2.1 线性表的顺序存储结构

1.顺序表

线性表的顺序存储是指在内存中用地址连续的一块存储空间顺序存放线性表的各元素，用这种存储形式存储的线性表称其为顺序表。因为内存中的地址空间是线性的，因此，用物理上的相邻实现数据元素之间的逻辑相邻关系是既简单又自然的。

设a_1的存储地址为Loc(a_1)，每个数据元素占d个存储地址，则第i个数据元素的地址为：

$Loc(a_i)=Loc(a_1)+(i-1)*d \quad 1\le i\le n$

这就是说只要知道顺序表首地址和每个数据元素所占地址单元的个数就可求出第i个数据元素的地址来，这也是顺序表具有按数据元素的序号随机存取的特点。

线性表的动态分配顺序存储结构：

```
#define LIST_INIT_SIZE 100   //存储空间的初始分配量
#define LISTINCREMENT 10   //存储空间的分配增量
typedef   struct{
     ElemType   *elem;          //线性表的存储空间基址
     int   length;              //当前长度
     int   listsize;             //当前已分配的存储空间
}SqList;
```

2.顺序表上基本运算的实现

（1）顺序表的初始化

顺序表的初始化即构造一个空表，这对表是一个加工型的运算，因此，将L设为引用参数，首先动态分配存储空间，然后，将length置为0，表示表中没有数据元素。

```
int Init_SqList (SqList &L){
     L.elem = (ElemType   * )malloc(LIST_INIT_SIZE * sizeof(ElemType));
     if (!L.elem) exit (OVERFLOW);      //存储分配失败
     L.length=0;
```

```
    L. listsize = LIST_INIT_SIZE;          //初始存储容量
    return OK;
}
```

（2）插入运算

线性表的插入是指在表的第i(i的取值范围：1≤i≤n+1)个位置上插入一个值为 x 的新元素，插入后使原表长为 n的表:

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

成为表长为 n+1 表:

$(a_1, a_2, \ldots, a_{i-1}, x, a_i, a_{i+1}, \ldots, a_n)$ 。

顺序表上完成这一运算则通过以下步骤进行：

① 将a_i～a_n 顺序向下移动，为新元素让出位置；（注意数据的移动方向：从后往前依次后移一个元素）

② 将 x 置入空出的第i个位置；

③ 修改表长。

```
int Insert_SqList (SqList &L，int i，ElemType x){
    if (i < 1 || i > L.length+1) return ERROR;        // 插入位置不合法
    if (L.length >= L.listsize) return OVERFLOW; // 当前存储空间已满,不能插入
        //需注意的是，若是采用动态分配的顺序表，当存储空间已满时也可增加分配
    q = &(L.elem[i-1]);                                // q 指示插入位置
    for (p = &(L.elem[L.length-1]); p >= q; --p)
       *(p+1) = *p;                                    // 插入位置及之后的元素右移
    *q = e;                                            // 插入e
    ++L.length;                                        // 表长增1
    return OK;
}
```

顺序表上的插入运算，时间主要消耗在了数据的移动上，在第i个位置上插入 x ，从 a_i 到 a_n 都要向下移动一个位置，共需要移动 n－i＋1个元素。

（3）删除运算

线性表的删除运算是指将表中第 i （i 的取值范围为 ：1≤ i≤n）个元素从线性表中去掉，删除后使原表长为 n 的线性表：

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

成为表长为 n－1 的线性表：

$(a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$。

顺序表上完成这一运算的步骤如下：

① 将a_{i+1}～a_n 顺序向上移动；（注意数据的移动方向：从前往后依次前移一个元素）

② 修改表长。

```
int Delete_SqList (SqList &L;int i) {
    if ((i < 1) || (i > L.length))    return ERROR;   // 删除位置不合法
    p = &(L.elem[i-1]);                    // p 为被删除元素的位置
```

```
    e = *p;                          // 被删除元素的值赋给 e
    q = L.elem+L.length-1;           // 表尾元素的位置
    for (++p; p <= q; ++p)
        *(p-1) = *p;                 // 被删除元素之后的元素左移
    --L.length;                      // 表长减1
    return OK;
}
```

顺序表的删除运算与插入运算相同，其时间主要消耗在了移动表中元素上，删除第i个元素时，其后面的元素 a_{i+1}～a_n 都要向上移动一个位置，共移动了 n-i 个元素，

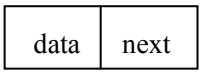
顺序表的插入、删除需移动大量元素 O(n)；但在尾端插入、删除效率高 O(1)。

1.2.2 线性表的链式存储结构

1.2.2.1 单链表

1.链表表示

链表是通过一组任意的存储单元来存储线性表中的数据元素的。为建立起数据元素之间的线性关系，对每个数据元素a_i，除了存放数据元素的自身的信息 a_i 之外，还需要和a_i一起存放其后继 a_{i+1} 所在的存储单元的地址，这两部分信息组成一个“结点”，结点的结构如图所示。



单链表结点结构

其中，存放数据元素信息的称为数据域，存放其后继地址的称为指针域。因此n个元素的线性表通过每个结点的指针域拉成了一个“链”，称之为链表。因为每个结点中只有一个指向后继的指针，所以称其为单链表。

线性表的单链表存储结构C语言描述下：

```
typedef struct LNode{
    ElemType        data;   // 数据域
    struct LNode    *next;  // 指针域
}LNode，*LinkList;
LinkList L;                 // L 为单链表的头指针
```

通常用“头指针”来标识一个单链表，如单链表L、单链表H等，是指某链表的第一个结点的地址放在了指针变量 L、H 中， 头指针为“NULL”则表示一个空表。

2.单链表上基本运算的实现

(1)建立单链表

●头插法——在链表的头部插入结点建立单链表

链表与顺序表不同，它是一种动态管理的存储结构，链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的，因此建立单链表从空表开始，每读入一个数据元素则申请一个结点，然后插在链表的头部。

```
LinkList   CreateListF ( ){
    LinkList L=NULL;   //空表
    LNode *s;
    int x;                    //设数据元素的类型为int
    scanf( " %d " ,&x);
    while (x!=flag) {
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        s->next=L;   L=s;
        scanf ( " %d " ,&x);
    }
    return L;
}
```

●尾插法——在单链表的尾部插入结点建立单链表

头插入建立单链表简单，但读入的数据元素的顺序与生成的链表中元素的顺序是相反的，若希望次序一致，则用尾插入的方法。因为每次是将新结点插入到链表的尾部，所以需加入一个指针 r 用来始终指向链表中的尾结点，以便能够将新结点插入到链表的尾部。

初始状态，头指针L=NULL，尾指针 r=NULL；按线性表中元素的顺序依次读入数据元素，不是结束标志时，申请结点，将新结点插入到 r 所指结点的后面，然后 r 指向新结点（注意第一个结点有所不同）。

```
LinkList   CreateListR1 ( ){
    LinkList L=NULL;
    LNode   *s,*r=NULL;
    int x;                          //设数据元素的类型为int
    scanf( " %d " ,&x);
    while (x!=flag) {
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        if   (L==NULL)   L=s;   //第一个结点的处理
        else   r->next=s;          //其它结点的处理
        r=s;                          //r  指向新的尾结点
        scanf( " %d " ,&x);
    }
    if ( r!=NULL)   r->next=NULL;   //对于非空表，最后结点的指针域放空指针
    return L;
```

```
}
```

在算法CreateListR1中，第一个结点的处理和其它结点是不同的，原因是第一个结点加入时链表为空，它没有直接前驱结点，它的地址就是整个链表的指针，需要放在链表的头指针变量中；而其它结点有直接前驱结点，其地址放入直接前驱结点的指针域。“第一个结点”的问题在很多操作中都会遇到，如在链表中插入结点时，将结点插在第一个位置和其它位置是不同的，在链表中删除结点时，删除第一个结点和其它结点的处理也是不同的，等等。

为了方便操作，有时在链表的头部加入一个“头结点”，头结点的类型与数据结点一致，标识链表的头指针变量L中存放该结点的地址，这样即使是空表，头指针变量L也不为空了。头结点的加入使得“第一个结点”的问题不再存在，也使得“空表”和“非空表”的处理成为一致。

头结点的加入完全是为了运算的方便，它的数据域无定义，指针域中存放的是第一个数据结点的地址，空表时为空。

尾插法建立带头结点的单链表，将算法CreateListR1改写成算法CreateListR2形式。

```
LinkList    CreateListR2( ){
    LinkList L=(LNode *)malloc(sizeof(LNode));
    L->next=NULL;                    //空表
    LNode    *s,*r=L;
    int x;                           //设数据元素的类型为int
    scanf( " %d " ,&x);
    while (x!=flag) {
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s;                         //r  指向新的尾结点
        scanf( " %d " ,&x);
    }
    r->next=NULL;
    return L;
}
```

因此，头结点的加入会带来以下两个优点：

第一个优点：由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上的操作一致，无需进行特殊处理；

第二个优点：无论链表是否为空，其头指针是指向头结点在的非空指针（空表中头结点的指针域为空），因此空表和非空表的处理也就统一了。

在以后的算法中不加说明则认为单链表是带头结点的。

(2)查找操作

●按序号查找 Get_LinkList(L,i)

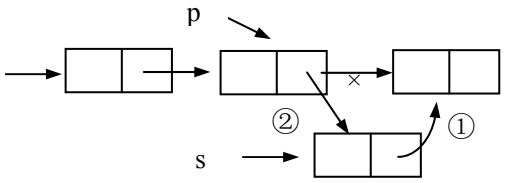
从链表的第一个元素结点起，判断当前结点是否是第i个，若是，则返回该结点的指针，否则继续后一个，表结束为止，没有第 i 个结点时返回空。

```
LNode    * Get_LinkList(LinkList    L, int    i); {
```

```
    LNode    * p=L;
    int    j=0;
    while (p->next !=NULL && j<i ){
          p=p->next;    j++;
    }
    if (j==i) return p;
    else return NULL;
}
```

(3)插入运算

●后插结点：设p指向单链表中某结点，s指向待插入的值为x的新结点，将*s插入到*p的后面，插入示意图如图所示。



在*p之后插入*s

操作如下：

①s->next=p->next;

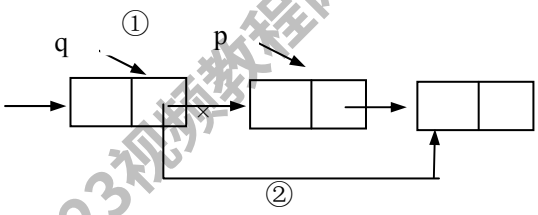
②p->next=s;

注意：两个指针的操作顺序不能交换。

(4)删除运算

●删除结点

设p指向单链表中某结点，删除*p。操作过程如图。要实现对结点*p的删除，首先要找到*p的前驱结点*q，然后完成指针的操作即可。



操作如下：①q=L;

```
    while (q->next!=p)
        q=q->next;          //找*p的直接前驱
②q->next=p->next;
    free(p);
```

因为找*p前驱的时间复杂度为O(n)，所以该操作的时间复杂度为O(n)

通过上面的基本操作我们得知：

(1) 单链表上插入、删除一个结点，必须知道其前驱结点。

(2) 单链表不具有按序号随机访问的特点，只能从头指针开始一个个顺序进行。

1.2.2.2 循环链表

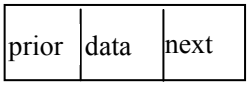
对于单链表而言，最后一个结点的指针域是空指针，如果将该链表头指针置入该指针域，则使得链表头尾结点相连，就构成了单循环链表。

在单循环链表上的操作基本上与非循环链表相同，只是将原来判断指针是否为 NULL 变为是否是头指针而已，没有其它较大的变化。

对于单链表只能从头结点开始遍历整个链表，而对于单循环链表则可以从表中任意结点开始遍历整个链表，不仅如此，有时对链表常做的操作是在表尾、表头进行，此时可以改变一下链表的标识方法，不用头指针而用一个指向尾结点的指针 R 来标识，可以使得操作效率得以提高。

1.2.2.3 双向链表

单链表的结点中只有一个指向其后继结点的指针域 next，因此若已知某结点的指针为 p，其后继结点的指针则为 p->next ，而找其前驱则只能从该链表的头指针开始，顺着各结点的 next 域进行，也就是说找后继的时间性能是 O(1)，找前驱的时间性能是 O(n)，如果也希望找前驱的时间性能达到 O(1)，则只能付出空间的代价：每个结点再加一个指向前驱的指针域，结点的结构为如图所示，用这种结点组成的链表称为双向链表。



线性表的双向链表存储结构C语言描述下：

```
typedef struct    DuLNode{
   ElemType data;
   struct DuLNode *prior,*next;
}DuLNode,*DuLinkList;
```

和单链表类似，双向链表通常也是用头指针标识，也可以带头结点。

（1）双向链表中结点的插入：设 p 指向双向链表中某结点，s 指向待插入的值为 x 的新结点，将*s 插入到*p 的前面，插入示意图如所示。

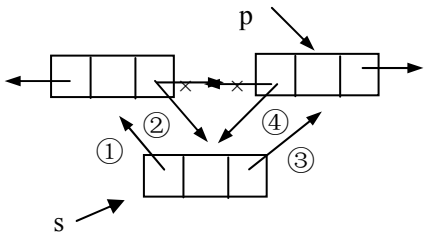


图 双向链表中的结点插入

操作如下：

① s->prior=p->prior;

② p->prior->next=s;

③ s->next=p;

④ p->prior=s;

指针操作的顺序不是唯一的，但也不是任意的，操作①必须要放到操作④的前面完成，否则*p 的前驱结点的指针就丢掉了。

（2）双向链表中结点的删除：设 p 指向双向链表中某结点，删除*p。操作示意图如图所示。

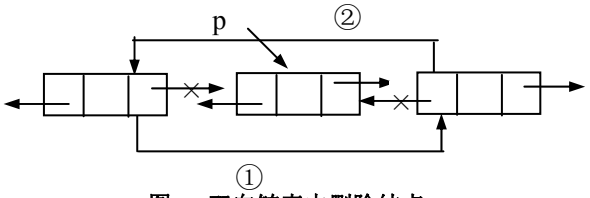


图 双向链表中删除结点

操作如下：

①p->prior->next=p->next;

②p->next->prior=p->prior;

free(p);

1.2.2.4 顺序表和链表的比较

顺序表	单链表
以地址相邻表示关系	用指针表示关系
随机访问，取元素 O(1)	顺序访问，取元素 O(n)
插入、删除需要移动元素 O(n)	插入、删除不用移动元素 O(n)(用于查找位置)

总之，两种存储结构各有长短，选择那一种由实际问题中的主要因素决定。通常“较稳定”的线性表选择顺序存储，而频繁做插入删除的即动态性较强的线性表宜选择链式存储。

习题

1、循环链表的主要优点是（ ）

A．不再需要头指针了。

B．已知某个结点的位置后，能很容易找到它的直接前驱结点。

C．在进行删除操作后，能保证链表不断开。

D．从表中任一结点出发都能遍历整个链表。

2.若线性表最常用的运算是查找第i个元素及其前驱的值，则采用（ ）存储方式节省时间。

A．单链表 B．双链表

C．单循环链表 D．顺序表

3.若某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用（ ）存储方式最节省运算时间。

A．单链表 B. 仅有头指针的单循环链表

C．双链表 D. 仅有尾指针的单循环链表

4.在具有n个结点的单链表中，实现（ ）的操作，其算法的时间复杂度都是O(n)。

A．遍历链表和求链表的第i个结点

B．在地址为p的结点之后插入一个结点

C．删除开始结点

D．删除地址为p的结点的后继结点

5.在n个结点的顺序表，算法的时间复杂度是O(1)的操作是（ ）。

A. 访问第i个结点(1≤i≤n)和求第i个结点的直接前驱(2≤i≤n)

B．在第i个结点后插入一个新结点(1≤i≤n)

C．删除第i个结点(1≤i≤n)

D．将n个结点从大到小排序

6.在非空循环双链表中q所指的结点前插入一个由p所指结点的过程依次为：

p->next=q；p->prior=q->prior；

q->prior=p；（ ）；

A．q->next=p；

B．q->prior->next=p；

C．p->prior->next=p；

D．p->next->prior=p；

7、已知 A=(a1, a2, …, am)，B=(b1, b2, …, bn)均为顺序表，试编写一个比较A,B大小的算法。

```
int compare(SqList La, SqList Lb) {
   i=0;
   while (i<La.Length && i<Lb.Length) {
      if (La.elem[i]==Lb.elem[i])    i++;
      else if (La.elem[i]<Lb.elem[i])
         return -1;
      else return 1;
   }
   if ( i>La.length && i>Lb.Length)    return 0;
   else    if (i>Lb.Length)      return 1;
           else    return -1;
}
```

8.删除有序表中所有其值大于 mink 且小于maxk的数据元素。

```
void delete(LinkList &L, int mink, int maxk) {
   p=L->next;
   while (p && p->data<=mink)
       { pre=p;    p=p->next; } //查找第一个值>mink的结点
```

```
   if (p) {
      while (p && p->data<maxk)    p=p->next;
                                   // 查找第一个值 ≥maxk 的结点
      q=pre->next;     pre->next=p;   // 修改指针
      while (q!=p)
      { s=q->next;    delete q;    q=s; } // 释放结点空间
   } // if
} // delete
```

9.逆置线性链表

```
void    inverse(LinkList &L) {
      // 逆置带头结点的单链表 L
      p=L->next;    L->next=NULL;
      while ( p) {
            succ=p->next;       // succ指向*p的后继
            p->next=L->next;
            L->next=p;            // *p插入在头结点之后
            p = succ;
      }
}
```

10.将两个非递减有序的有序表归并为非递增的有序链表（利用原表结点）。

```
void union( LinkList& Lc, LinkList& La,LinkList& Lb) {
   // 将非递减的有序表 La 和Lb归并为非递增的有序表Lc,归并之后，La和Lb表不再存在。
   // 上述三个表均为带头结点的单链表，Lc 表中的结点即为原 La 或 Lb 表中的结点。
   Lc = new LNode;    Lc->next = NULL;
   pa = La->next;    pb = Lb->next;                    // 初始化
   while ( pa || pb ) {
        if   ( !pa )   { q = pb;    pb = pb->next; }
        else if   ( !pb )   { q = pa;    pa = pa->next; }
        else    if (pa->data <= pb->data )
          { q = pa;    pa = pa->next; }
        else   { q = pb;    pb = pb->next; }
        q->next = Lc->next;    Lc->next = q;       // 插入
   }
delete La;    delete Lb;       // 释放La 和 Lb的头结点
} // union
```

11.已知A, B和C为三个有序链表，编写算法从A表中删除B表和C表中共有的数据元素。

```
void Difference_L( LinkList &La, LinkList Lb,LinkList Lc ) {
   // La, Lb 和 Lc 分别为三个非递减有序的单链表的头指针，从 La 表中删除所有的既在 Lb 表中出现，又在 Lc 表中出现的元素结点
```

```
  pre= pa;     pa=La->next;
  pb=Lb->next;   pc=Lc->next;
  while (pa && pb && pc) {
    if (pa->data<pb->data)
       { pre=pa;   pa=pa->next; }
    else if (pb->data<pc->data)
       pb=pb->next;
    else if (pc->data<pa->data)
       pc=pc->next;
    else {
       pre->next=pa->next;   delete pa; pa=pre->next;
    } // 删除，注意没有移动 pb 和 pc 才能实现删除所有满足条件的结点
  }// while
}//Difference
```

12.在双向循环链表的结点中，增加一个访问频度的数据域 freq，编写算法实现LOCATE(L,x)。

```
p=L->next; // L为双向链表的头指针
while (p!=L && p->data!=x)   p=p->next; // 搜索元素值为 x 的结点*p
if (p==L)   return NULL;
q=p->prior;
while (q!=L && q->freq<p->freq)   q=q->prior; //搜索访问频度不小于它的结点*q
将结点 *p 从当前位置上删除;
之后将结点 *p 插入在结点 *q 之后;
return p;
```

13.已知L为没有头结点的单链表中第一个结点的指针，每个结点数据域存放一个字符，该字符可能是英文字母字符或数字字符或其它字符，编写算法构造三个以带头结点的单循环链表表示的线性表，使每个表中只含同一类字符。（要求用最少的时间和最少的空间）

```
void   OneToThree（LinkedList &L， LinkedList & la， LinkedList & ld， LinkedList & lo）
{ la=（LinkedList）malloc（sizeof（LNode））；
  ld=（LinkedList）malloc（sizeof（LNode））；
  lo=（LinkedList）malloc（sizeof（LNode））；
  la->next=la；ld->next=ld；lo->next=lo；
  //建立三个链表的头结点，并置三个循环链表为空表
  while（L!=null）                                    //分解原链表。
 {   r=L; L=L->next;                                  //L指向待处理结点的后继
    if（r->data>=‘a’&& r->data<=‘z’|| r->data>=‘A’&& r->data<=‘Z’）
         {r->next=la->next；    la->next=r；}          //处理字母字符。
    else if（r->data>=‘0’&& r->data<=‘9’）
                  {r->next=ld->next；ld->next=r；}      //处理数字字符
            else {r->next=lo->next；lo->next=r；} //处理其它符号。
```

```
    }
}
```

第二章 栈、队列和数组

2.1 栈

2.1.1 栈的定义

栈是限制在表的一端进行插入和删除的线性表。允许插入、删除的这一端称为栈顶，另一个固定端称为栈底。当表中没有元素时称为空栈。

2.1.2 栈的存储实现和运算实现

栈是运算受限的线性表，线性表的存储结构对栈也是适用的，只是操作不同而已。

利用顺序存储方式实现的栈称为顺序栈。与线性表类似，栈的动态分配顺序存储结构如下：

```
#define STACK_INIT_SIZE 100     //存储空间的初始分配量
#define STACKINCREMENT 10     //存储空间的分配增量
typedef   struct{
     SElemType    *base;       //在栈构造之前和销毁之后，base 的值为 NULL
     SElemType    *top;        //栈顶指针
     int    stacksize;          //当前已分配的存储空间
}SqStack;
```

需要注意，在栈的动态分配顺序存储结构中，base 始终指向栈底元素，非空栈中的 top 始终在栈顶元素的下一个位置。

下面是顺序栈上常用的基本操作的实现。

（1）入栈：若栈不满，则将 e 插入栈顶。

```
int Push (SqStack &S, SElemType e) {
     if (S.top-S.base>=S.stacksize)
         {……}                              //栈满，追加存储空间
     *S.top++ = e;                         //top 始终在栈顶元素的下一个位置
     return OK;
}
```

（2）出栈：若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK，否则返回 ERROR。

```
int Pop (SqStack &S, SElemType &e) {
     if (S.top==S.base) return ERROR;
     e = *--S.top;
     return OK;
 }
```

出栈和读栈顶元素操作，先判栈是否为空，为空时不能操作，否则产生错误。通常栈空

常作为一种控制转移的条件。

2.1.3 栈的应用举例

由于栈的“先进先出”特点，在很多实际问题中都利用栈做一个辅助的数据结构来进行求解，下面通过几个例子进行说明。

1．数制转换

十进制数 N 和其他 d 进制数的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于下列原理：

N = (N div d)×d + N mod d （其中：div 为整除运算，mod 为求余运算）

例如：$(1348)_{10} = (2504)_8$ ，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

假设现要编制一个满足下列要求的程序：对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数。由于上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出，一般来说应从高位到低位进行，恰好和计算过程相反。因此，若将计算过程中得到的八进制数的各位顺序进栈，则按出栈序列打印输出的即为与输入对应的八进制数。

算法思想：当 N>0 时重复（1），（2）

（1）若 N≠0，则将 N % r 压入栈 s 中 ，执行（2）;若 N=0，将栈 s 的内容依次出栈，算法结束。

（2）用 N / r 代替 N。

```
void conversion () {
    InitStack(S);          // 构造空栈
    scanf ("%d",N);
    while (N) {
      Push(S, N % 8);
      N = N/8;
    }
    while (!StackEmpty(S)) {
      Pop(S,e);
      printf ( "%d", e );
    }
  }
```

2..表达式求值

表达式求值是程序设计语言编译中一个最基本的问题，它的实现也是需要栈的加入。下面的算法是由运算符优先法对表达式求值。在此仅限于讨论只含二目运算符的算术表达式。

（1）中缀表达式求值：

中缀表达式：每个二目运算符在两个运算量的中间，假设所讨论的算术运算符包括：+ 、- 、*、/、%、^（乘方）和括号（）。

设运算规则为：

．运算符的优先级为：（）——> ^ ——>*、/、%——> +、- ；

．有括号出现时先算括号内的，后算括号外的，多层括号，由内向外进行；

．乘方连续出现时先算最右面的。

表达式作为一个满足表达式语法规则的串存储，如表达式“3*2^（4+2*2- 1 *3）-5”,它的的求值过程为：自左向右扫描表达式，当扫描到 3*2 时不能马上计算，因为后面可能还有更高的运算，正确的处理过程是：需要两个栈：对象栈 s1 和运算符栈 s2。当自左至右扫描表达式的每一个字符时，若当前字符是运算对象，入对象栈，是运算符时，若这个运算符比栈顶运算符高则入栈，继续向后处理，若这个运算符比栈顶运算符低则从对象栈出栈两个运算量，从运算符栈出栈一个运算符进行运算，并将其运算结果入对象栈，继续处理当前字符，直到遇到结束符。中缀表达式表达式 “3*2^（4+2*2- 1 *3）-5”求值过程中两个栈的状态情况见图所示。

读字符	对象栈 s1	算符栈 s2	说明
3	3	#	3 入栈 s1
*	3	#*	*入栈 s2
2	3，2	#*	2 入栈 s1
^	3，2	#*^	^入栈 s2
(	3，2	#*^(	(入栈 s2
4	3，2，4	#*^(	4 入栈 s1
+	3，2，4	#*^(+	+入栈 s2
2	3，2，4，2	#*^(+	2 入栈 s1
*	3，2，4，2	#*^(+*	*入栈 s2
2	3，2，4，2，2	#*^(+*	2 入栈 s1
-	3，2，4，4	#*^(+	做 2+2=4，结果入栈 s1
	3，2，8	#*^(	做 4+4=8，结果入栈 s2
	3，2，8	#*^(-	-入栈 s2
1	3，2，8，1	#*^(-	1 入栈 s1
*	3，2，8，1	#*^(-*	*入栈 s2
3	3，2，8，1，3	#*^(-*	3 入栈 s1
)	3，2，8，3	#*^(-	做 1*3，结果 3 入栈 s1
	3，2，5	#*^(	做 8-3，结果 5 入栈 s2
	3，2，5	#*^	(出栈
-	3，32	#*	做 2^5，结果 32 入栈 s1
	96	#	做 3*32，结果 96 入栈 s1
	96	#-	-入栈 s2
5	96，5	#-	5 入栈 s1

结束符	91	#	做 96-5, 结果 91 入栈 s1

图 中缀表达式 3*2^（4+2*2-1*3）-5 的求值过程

为了处理方便，编译程序常把中缀表达式首先转换成等价的后缀表达式，后缀表达式的运算符在运算对象之后。在后缀表达式中，不在引入括号，所有的计算按运算符出现的顺序，严格从左向右进行，而不用再考虑运算规则和级别。中缀表达式“3*2^（4+2*2-1*3）-5 ”的后缀表达式为：“32422*+13*-^*5-”。

（2）后缀表达式求值

计算一个后缀表达式，算法上比计算一个中缀表达式简单的多。这是因为表达式中即无括号又无优先级的约束。具体做法：只使用一个对象栈，当从左向右扫描表达式时，每遇到一个操作数就送入栈中保存，每遇到一个运算符就从栈中取出两个操作数进行当前的计算，然后把结果再入栈，直到整个表达式结束，这时送入栈顶的值就是结果。

下面是后缀表达式求值的算法，在下面的算法中假设，每个表达式是合乎语法的，并且假设后缀表达式已被存入一个足够大的字符数组 A 中，且以‘#’为结束字符，为了简化问题，限定运算数的位数仅为一位且忽略了数字字符串与相对应的数据之间的转换的问题。

```
typedef   char SElemType ;
double   calcul_exp(char *A){      //本函数返回由后缀表达式 A 表示的表达式运算结果
  SqStack   s ;
  ch=*A++ ; InitStack(s) ;
  while ( ch != ’#’ ){
      if   (ch!=运算符)   Push (s , ch) ;
      else {
          Pop (s , &a) ;   Pop (s , &b) ; //取出两个运算量
          switch (ch).{
            case ch= =’+’:      c =a+b ; break ;
            case ch= =’-’:      c=a-b ; break ;
            case ch= =’*’:      c=a*b ; break ;
            case ch= =’/’:      c=a/b ; break ;
            case ch= =’%’:      c=a%b ; break ;
          }
          Push (s, c) ;
      }
      ch=*A++ ;
  }
  Pop ( s , result ) ;
  return   result ;
}
```

（3）中缀表达式转换成后缀表达式：

将中缀表达式转化为后缀表达示和前述对中缀表达式求值的方法完全类似，但只需要运算符栈，遇到运算对象时直接放后缀表达式的存储区，假设中缀表达式本身合法且在字符数

组 A 中，转换后的后缀表达式存储在字符数组 B 中。具体做法：遇到运算对象顺序向存储后缀表达式的 B 数组中存放，遇到运算符时类似于中缀表达式求值时对运算符的处理过程，但运算符出栈后不是进行相应的运算，而是将其送入 B 中存放。具体算法在此不再赘述。

3．栈与递归

在高级语言编制的程序中，调用函数与被调用函数之间的链接和信息交换必须通过栈进行。当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事:

（1）将所有的实在参数、返回地址等信息传递给被调用函数保存;

（2）为被调用函数的局部变量分配存储区;

（3）将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成:

（1）保存被调函数的计算结果;

（2）释放被调函数的数据区;

（3）依照被调函数保存的返回地址将控制转移到调用函数。

多个函数嵌套调用的规则是：后调用先返回，此时的内存管理实行“栈式管理”。

递归函数的调用类似于多层函数的嵌套调用，只是调用单位和被调用单位是同一个函数而已。在每次调用时系统将属于各个递归层次的信息组成一个活动记录（ActivaTion Record），这个记录中包含着本层调用的实参、返回地址、局部变量等信息，并将这个活动记录保存在系统的“递归工作栈”中，每当递归调用一次，就要在栈顶为过程建立一个新的活动记录，一旦本次调用结束，则将栈顶活动记录出栈，根据获得的返回地址信息返回到本次的调用处。

将递归程序转化为非递归程序时常使用栈来实现。

2.2 队列

2.2.1 队列的定义及基本运算

栈是一种后进先出的数据结构，在实际问题中还经常使用一种“先进先出”的数据结构：即插入在表一端进行，而删除在表的另一端进行，将这种数据结构称为队或队列，把允许插入的一端叫队尾(rear) ，把允许删除的一端叫队头(front)。

2.2.2 队列的存储实现及运算实现

与线性表、栈类似，队列也有顺序存储和链式存储两种存储方法。

1．顺序队列

循环队列的类型定义如下：

```
#define MAXQSIZE    100     //最大队列长度
typedef struct {
   QElemType    *base;        //动态分配存储空间
   int    front;              //头指针，若队列不空，指向队列头元素
   int    rear;               //尾指针，若队列不空，指向队列尾元素的下一个位置
} SqQueue;
```

下面是循环队列上基本操作的实现。

（1）入队：

```
int EnQueue (SqQueue &Q, QElemType e) {
    if((Q.rear+1)%MAXQSIZE == Q.front) return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

（2）出队：

```
int DeQueue (SqQueue &Q, QElemType &e) {
    if (Q.front = = Q.rear)    return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1) % MAXQSIZE;
    return OK;
}
```

（3）求循环队列元素个数：

```
int QueueLength（SqQueue Q）{
    return (Q.rear-Q.front+MAXQSIZE) %MAXQSIZE;
}
```

2．链队列

链式存储的队称为链队列。和链栈类似，用单链表来实现链队列，根据队的先进先出原则，为了操作上的方便，分别需要一个头指针和尾指针。

链队列的形式描述如下：

```
typedef struct QNode { //  结点类型
    QElemType      data;
    struct QNode    *next;
} QNode, *QueuePtr;
typedef struct {          //链队列类型
    QueuePtr    front;    //队头指针
    QueuePtr    rear;     //队尾指针
} LinkQueue;
```

定义一个指向链队列的指针：LinkQueue　Q;

下面是链队列的基本运算的实现。

（1）入队

```
int EnQueue (LinkQueue &Q, QElemType e) {
    QNode *p;
    p =  （QNode *）malloc（sizeof（QNode））;
    p->data = e;
    p->next = NULL;
```

```
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}
```

（2）出队

```
int DeQueue (LinkQueue &Q, QElemType &e) {
    if (Q.front == Q.rear)    return ERROR; //队空，出队失败
    p = Q.front->next;
    e = p->data;                                  //队头元素放 e 中
    Q.front->next = p->next;
    if(Q.rear==p)      Q.rear= Q.front;         //只有一个元素时，此时还要修改队尾指针
    free (p);
    return OK;
}
```

3．除了栈和队列之外，还有一种限定性数据结构是双端队列。

（1）双端队列：可以在双端进行插入和删除操作的线性表。

（2）输入受限的双端队列：线性表的两端都可以输出数据元素，但是只能在一端输入数据元素。

（3）输出受限的双端队列：线性表的两端都可以输入数据元素，但是只能在一端输出数据元素。

2.3 特殊矩阵的压缩存储

2.3.1 数组

数组可以看作线性表的推广。数组作为一种数据结构其特点是结构中的元素本身可以是具有某种结构的数据，但属于同一数据类型，数组是一个具有固定格式和数量的数据有序集，每一个数据元素有唯一的一组下标来标识，因此，在数组上不能做插入、删除数据元素的操作。通常在各种高级语言中数组一旦被定义，每一维的大小及上下界都不能改变。

现在来讨论数组在计算机中的存储表示。通常，数组在内存被映象为向量，即用向量作为数组的一种存储结构，这是因为内存的地址空间是一维的，数组的行列固定后，通过一个映象函数，则可根据数组元素的下标得到它的存储地址。

对于一维数组按下标顺序分配即可。对多维数组分配时，要把它的元素映象存储在一维存储器中，一般有两种存储方式：一是以行为主序（或先行后列）的顺序存放，即一行分配完了接着分配下一行。另一种是以列为主序（先列后行）的顺序存放，即一列一列地分配。以行为主序的分配规律是：最右边的下标先变化，即最右下标从小到大，循环一遍后，右边第二个下标再变，…，从右向左，最后是左下标。以列为主序分配的规律恰好相反：最左边的下标先变化，即最左下标从小到大，循环一遍后，左边第二个下标再变，…，从左向右，最后是右下标。

设有 m×n 二维数组 A_{mn}，下面我们看按元素的下标求其地址的计算：

以“以行为主序”的分配为例：设数组的基址为 $LOC(a_{11})$，每个数组元素占据 d 个地址单元，那么 a_{ij} 的物理地址可用一线性寻址函数计算：

$$LOC(a_{ij}) = LOC(a_{11}) + ((i-1)*n + j-1) * d$$

这是因为数组元素 a_{ij} 的前面有 i-1 行，每一行的元素个数为 n，在第 i 行中它的前面还有 j-1 个数组元素。

在 C 语言中，数组中每一维的下界定义为 0，则：

$$LOC(a_{ij}) = LOC(a_{00}) + (i*n + j) * d$$

推广到一般的二维数组：$A[c_1..d_1]\ [c_2..d_2]$，则 a_{ij} 的物理地址计算函数为：

$$LOC(a_{ij})=LOC(a_{c1\ c2})+((i- c_1) *(d_2 - c_2 + 1)+ (j- c_2))d$$

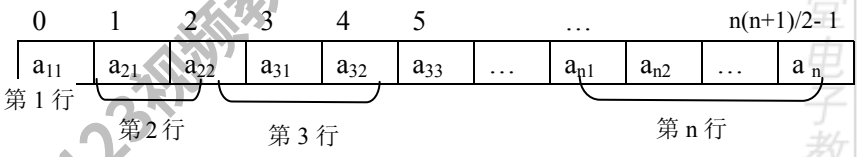
2.3.2 特殊矩阵

对于一个矩阵结构显然用一个二维数组来表示是非常恰当的，矩阵在这种存储表示之下，可以对其元素进行随机存取，各种矩阵运算也非常简单，并且存储的密度为 1。但是在矩阵中非零元素呈某种规律分布或者矩阵中出现大量的零元素的情况下，比如常见的一些特殊矩阵，如三角矩阵、对称矩阵、对角矩阵、稀疏矩阵等，从节约存储空间的角度考虑，这种存储是不太合适的，看起来存储密度仍为 1，但实际上占用了许多单元去存储重复的非零元素或零元素，这对高阶矩阵会造成极大的浪费，为了节省存储空间，我们可以对这类矩阵进行压缩存储：即为多个相同的非零元素只分配一个存储空间；对零元素不分配空间。

1.对称矩阵

对称矩阵的特点是：在一个 n 阶方阵中，有 $a_{ij}=a_{ji}$ ，其中 $1\le i, j\le n$，对称矩阵关于主对角线对称，因此只需存储上三角或下三角部分即可，比如，我们只存储下三角中的元素 a_{ij}，其特点是中 $j\le i$ 且 $1\le i\le n$，对于上三角中的元素 a_{ij} ，它和对应的 a_{ji} 相等，因此当访问的元素在上三角时，直接去访问和它对应的下三角元素即可，这样，原来需要 n*n 个存储单元，现在只需要 n(n+1)/2 个存储单元了，节约了 n(n-1)/2 个存储单元。

对下三角部分以行为主序顺序存储到一个向量中去，在下三角中共有 n*(n+1)/2 个元素，因此，不失一般性，设存储到向量 SA[n(n+1)/2]中，存储顺序可用图示意，这样，原矩阵下三角中的某一个元素 a_{ij} 则具体对应一个 sa_k，下面的问题是要找到 k 与 i、j 之间的关系。



对称矩阵的压缩存储

对于下三角中的元素 a_{ij}，其特点是：$i\ge j$ 且 $1\le i\le n$，存储到 SA 中后，根据存储原则，它前面有 i-1 行，共有 1+2+…+i-1=i*(i-1)/2 个元素，而 a_{ij} 又是它所在的行中的第 j 个，所以在上面的排列顺序中，a_{ij} 是第 i*(i-1)/2+j 个元素，因此它在 SA 中的下标 k 与 i、j 的关系为：

$$k=i*(i-1)/2+j-1 \quad (0 \le k<n*(n+1)/2)$$

若 i<j，则 a_{ij} 是上三角中的元素，因为 $a_{ij}=a_{ji}$ ，这样，访问上三角中的元素 a_{ij} 时则去访问和它对应的下三角中的 a_{ji} 即可，因此将上式中的行列下标交换就是上三角中的元素在 SA

中的对应关系：

$$k=j*(j-1)/2+i-1 \quad (0 \le k<n*(n+1)/2)$$

综上所述，对于对称矩阵中的任意元素 a_{ij}，若令 I=max(i,j)，J=min(i,j)，则将上面两个式子综合起来得到：

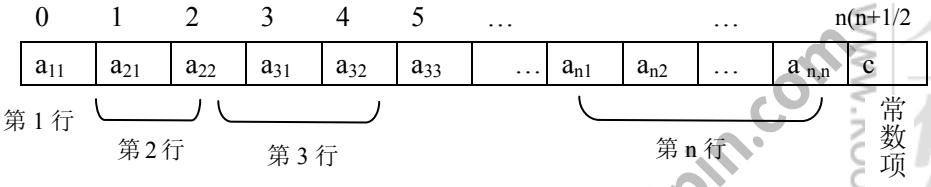
$$k=I*(I-1)/2+J-1。$$

2.三角矩阵

（1）下三角矩阵

与对称矩阵类似，不同之处在于存完下三角中的元素之后，紧接着存储对角线上方的常量，因为是同一个常数，所以存一个即可，这样一共存储了 n*(n+1)+1 个元素，设存入向量：SA[n*(n+1)+1]中，这种的存储方式可节约 n*(n-1)-1 个存储单元，sa_k 与 a_{ji} 的对应关系为：

$$k=\begin{cases} i*(i-1)/2+j-1 & \text{当 } i \ge j \\ n*(n+1)/2 & \text{当 } i<j \end{cases}$$



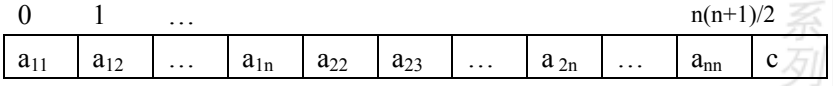
下三角矩阵的压缩存储

（2）上三角矩阵

对于上三角矩阵，存储思想与下三角类似，以行为主序顺序存储上三角部分，最后存储对角线下方的常量。对于第 1 行，存储 n 个元素，第 2 行存储 n-1 个元素，…，第 p 行存储(n-p+1)个元素，a_{ij} 的前面有 i-1 行，共存储：$n+(n-1)+\ldots+(n-i+1)=\sum_{p=1}^{i-1}(n-p)+1$ =(i-1)*(2n-i+2)/2 个元素，而 a_{ij} 是它所在的行中要存储的第（j-i+1）个；所以，它是上三角存储顺序中的第(i-1)*(2n-i+2)/2+(j-i+1)个，因此它在 SA 中的下标为：k=(i-1)*(2n-i+2)/2+j-i。

综上， sa_k 与 a_{ji} 的对应关系为：

$$k=\begin{cases} (i-1)*(2n-i+2)/2+j-i & \text{当 } i \le j \\ n*(n+1)/2 & \text{当 } i>j \end{cases}$$



3.对角矩阵

对角矩阵也称为带状矩阵。，在这种三对角矩阵中，所有非零元素都集中在以主对角线为

第 1 行　　第 2 行　　第 n 行　常数项

上三角矩阵的压缩存储

中心的对角区域中，即除了主对角线和它的上下方若干条对角线的元素外，所有其他元素都为零(或同一个常数 c)。

三对角矩阵 A 也可以采用压缩存储,将三对角矩阵压缩到向量 SA 中去，按以行为主序，顺序的存储其非零元素，如图所示，按其压缩规律，找到相应的映象函数。

sa_k 与 a_{ji} 的对应关系为：

$$k=2*i+j-3$$

$$A=\begin{pmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & 0 & 0 & 0 \\ \mathbf{a_{21}} & \mathbf{a_{22}} & \mathbf{a_{23}} & 0 & 0 \\ 0 & \mathbf{a_{32}} & \mathbf{a_{33}} & \mathbf{a_{34}} & 0 \\ 0 & 0 & \mathbf{a_{43}} & \mathbf{a_{44}} & \mathbf{a_{45}} \\ 0 & 0 & 0 & \mathbf{a_{54}} & \mathbf{a_{55}} \end{pmatrix}$$

(a) 三对角矩阵

0	1	2	3	4	5	6	7	8	9	10	11	12
a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}	a_{45}	a_{54}	a_{55}

(b) 压缩为向量

图 对角矩阵及压缩存储

习题

1、若循环队列以数组 Q[0..m-1]作为其存储结构，变量 rear 表示循环队列中的队尾元素的实际位置，其移动按 rear=(rear+1) Mod m 进行，变量 length 表示当前循环队列中的元素个数，则循环队列的队首元素的实际位置是（ ）.

A．rear-length

B．(rear-length+m) Mod m

C．(1+rear+m-length) Mod m

D．M-length

2.用链接方式存储的队列，在进行删除运算时（ ）。

A.仅修改头指针

B.仅修改尾指针

C.头、尾指针都要修改

D.头、尾指针可能都要修改

3、一个栈的入栈序列是 1,2,3,4,5，则栈的不可能的输出序列是

A. 5,4,3,2,1 B. 4,5,3,2,1 C. 4,3,5,1,2 D. 1,2,3,4,5

【分析】此类问题是常见题型。解答的基本原理是：一串数据依次通过一个栈，并不能保证出栈数据的次序总是倒置，可以产生多种出栈序列。一串数据通过一个栈后的次序由每个数据之间的进栈、出栈操作序列决定，只有当所有数据“全部进栈后再全部出栈”才能使数据倒置。事实上，存在一种操作序列——“进栈、出栈、进栈、出栈……”——可以使数据通过栈后仍然保持次序不变。

【解题技巧】将一组数据入栈后，判断题目备选项中的不可能的出栈顺序，上述这类题目有一个解题技巧：在输出序列中任意元素后面不能出现比该元素小并且是升序（指的是元素的序号）的两个元素。

4 识别读入的一个字符序列是否为反对称的字符序列。

```
int symmetry(char Ch[] ) {
  // 若 Ch[] 为反对称字符序列，则返回 1,否则返回 0。
  p = Ch;   InitStack(S);
  while (*p!='&') { Push(S,*p);   p++; }
  state = 1;   p++;    // 滤去字符 '&'
  while (*p!='@' && state ) {
   if (NOT StackEmpty(S)&&GetTop(S)==*p )
       { Pop(S,e);   p++; }
   else state = 0;
  }
```

5 判别读入的字符序列是否为“回文”。

```
Status ex ( )  {// 若从终端依次输入的字符序列是“回文”，//  则返回 TRUE,否则返回 FALSE
    InitStack(S);   InitQueue(Q);
    scanf(ch);
    while(ch!='@') {
         Push(S, ch);   EnQueue(Q, ch);
         scanf(ch);
    }
  state=TRUE;
    while(!StackEmpty && state)
    {
         if(GetTop(S)==GetHead(Q))
             { Pop(S);   DeQueue(Q); }
         else state=FALSE;
    }
    return state;
}
```

6.指出下列程序段的功能是什么？

(1) void demo1(sqstack &s){
int i;arr[64];n=0;
while (!stackempty(s)) arr[n++]=pop(s);
for(i=0;<n;i++) push(s,arr[i]);
}

(2) void demo2(sqstack &s,int m){
sqstack t; int i; initstack(t);
while(! stackempty(s))
if(i=pop(s)!=m) push(t,i);
while(! Stackempty(t)) {
i=pop(t); push(s,i);
}
}

7、将一个 A[1..100，1..100]的三对角矩阵，按行优先存入一维数组 B[1‥298]中，A 中元素 $A_{66,65}$（即该元素下标 i=66，j=65），在 B 数组中的位置 K 为（ ）。

A. 198 B. 195 C. 197 D. 196

8、已知有一维数组 A[0..m*n-1]，若要对应为 m 行、n 列的矩阵，则下面的对应关系（ ）可将元素 A[k](0≦k < m*n)表示成矩阵的第 i 行、第 j 列的元素(0≦i < m, 0≦j < n)。

A. i=k/n, j=k%m B. i=k/m, j=k%m

C. i=k/n, j=k%n D. i=k/m, j=k%n

第三章 树与二叉树

3.1 树的概念

1.树的定义

树（Tree）是 n（n≥0）个有限数据元素的集合。当 n＝0 时，称这棵树为空树。在一棵非树 T 中：

（1）有一个特殊的数据元素称为树的根结点，根结点没有前驱结点；

（2）若 n>1，除根结点之外的其余数据元素被分成 m（m>0）个互不相交的集合 T_1，T_2，…，T_m，其中每一个集合 T_i（1≤i≤m）本身又是一棵树。树 T_1，T_2，…，T_m 称为这个根结点的子树。

2．相关术语

（1）结点的度：结点所拥有的子树的个数称为该结点的度。

（2）叶结点：度为 0 的结点称为叶结点，或者称为终端结点。

（3）分支结点：度不为 0 的结点称为分支结点，或者称为非终端结点。一棵树的结点除叶结点外，其余的都是分支结点。

（4）孩子、双亲、兄弟：树中一个结点的子树的根结点称为这个结点的孩子。这个结点称为它孩子结点的双亲。具有同一个双亲的孩子结点互称为兄弟。

（5）路径、路径长度：如果一棵树的一串结点 $n_1,n_2,\ldots,n_k$ 有如下关系：结点 n_i 是 n_{i+1} 的父结点（1≤i<k）,就把 $n_1,n_2,\ldots,n_k$ 称为一条由 n_1 至 n_k 的路径。这条路径的长度是 k-1。

（6）祖先、子孙：在树中，如果有一条路径从结点 M 到结点 N，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

（7）结点的层数：树的根结点的层数为 1，其余结点的层数等于它的双亲结点的层数加 1。

（8）树的深度：树中所有结点的最大层数称为树的深度。

（9）树的度：树中各结点度的最大值称为该树的度。

（10）有序树和无序树：如果一棵树中结点的各子树从左到右是有次序的，即若交换了某结点各子树的相对位置，则构成不同的树，称这棵树为有序树；反之，则称为无序树。

（11）森林：零棵或有限棵不相交的树的集合称为森林。自然界中树和森林是不同的概念，但在数据结构中，树和森林只有很小的差别。任何一棵树，删去根结点就变成了森林。

3.2 二叉树

3.2.1 定义与性质

1.二叉树

二叉树（Binary Tree）是 n 个有限元素的集合，该集合或者为空、或者由一个称为根(root)的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成。当集合为空时，称该二叉树为空二叉树。在二叉树中，一个元素也称作一个结点。

二叉树是有序的，即若将其左、右子树颠倒，就成为另一棵不同的二叉树。即使树中结

点只有一棵子树，也要区分它是左子树还是右子树。

2.满二叉树与完全二叉树

（1）满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的一棵二叉树称作满二叉树。

（2）完全二叉树

一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i（1≤i≤n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。完全二叉树的特点是：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。

3.二叉树的性质

性质 1 一棵非空二叉树的第 i 层上最多有 2^{i-1} 个结点（i≥1）。（证明略）

性质 2 一棵深度为 k 的二叉树中，最多具有 2^k-1 个结点。

证明：设第 i 层的结点数为 x_i（1≤i≤k），深度为 k 的二叉树的结点数为 M，x_i 最多为 2^{i-1}，则有：

$$M=\sum_{i=1}^{k}x_i\leq\sum_{i=1}^{k}2^{i-1}=2^k-1$$

性质 3 对于一棵非空的二叉树，如果叶子结点数为 n_0，度数为 2 的结点数为 n_2，则有:

$$n_0=n_2+1。$$

证明：设 n 为二叉树的结点总数，n_1 为二叉树中度为 1 的结点数，则有：

$$n=n_0+n_1+n_2 \qquad （式 1-3-1）$$

在二叉树中，除根结点外，其余结点都有唯一的一个进入分支。设 B 为二叉树中的分支数，那么有：

$$B=n-1 \qquad （式 1-3-2）$$

这些分支是由度为 1 和度为 2 的结点发出的，一个度为 1 的结点发出一个分支，一个度为 2 的结点发出两个分支，所以有：

$$B=n_1+2n_2 \qquad （式 1-3-3）$$

综合（式 1-3-1）、（式 1-3-2）、（式 1-3-3）式可以得到：

$$n_0=n_2+1$$

性质 4 具有 n 个结点的完全二叉树的深度 k 为$\lfloor \log_2 n \rfloor+1$ 或$\lceil \log_2(n+1) \rceil$。

证明：根据完全二叉树的定义和性质 2 可知，当一棵完全二叉树的深度为 k、结点个数为 n 时，有

$$2^{k-1}-1<n\leq 2^k-1$$

即

$$2^{k-1}\leq n<2^k$$

对不等式取对数，有

$$k-1\leq \log_2 n<k$$

由于 k 是整数，所以有 k=$\lfloor \log_2 n \rfloor$+1。

性质 5　对于具有 n 个结点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有：

（1）如果 i>1，则序号为 i 的结点的双亲结点的序号为$\lfloor i/2 \rfloor$；如果 i=1，则序号为 i 的结点是根结点，无双亲结点。

（2）如果 2i≤n，则序号为 i 的结点的左孩子结点的序号为 2i；如果 2i>n，则序号为 i 的结点无左孩子。

（3）如果 2i＋1≤n，则序号为 i 的结点的右孩子结点的序号为 2i＋1；如果 2i＋1>n，则序号为 i 的结点无右孩子。

此外，若对二叉树的根结点从 0 开始编号，则相应的 i 号结点的双亲结点的编号为$\lfloor (i-1)/2 \rfloor$，左孩子的编号为 2i＋1，右孩子的编号为 2i＋2。

此性质可采用数学归纳法证明。证明略。

3.2.2 二叉树的存储

1．顺序存储结构

所谓二叉树的顺序存储，就是用一组连续的存储单元存放二叉树中的结点。一般是按照二叉树结点从上至下、从左到右的顺序存储。这样结点在存储位置上的前驱后继关系并不一定就是它们在逻辑上的邻接关系，然而只有通过一些方法确定某结点在逻辑上的前驱结点和后继结点，这种存储才有意义。

依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。

对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的结点顺序存储在一维数组中，则数组元素下标之间的关系不能够反映二叉树中结点之间的逻辑关系，只有增添一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。显然，这种存储对于需增加许多空结点才能将一棵二叉树改造成为一棵完全二叉树的存储时，会造成空间的大量浪费，不宜用顺序存储结构。最坏的情况是右单支树，一棵深度为 k 的右单支树，只有 k 个结点，却需分配 2^k-1 个存储单元。

二叉树的顺序存储表示可描述为：

```
#define MAX_TREE_SIZE 100                    //二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE]    //0 号单元存放根结点
SqBiTree b;   //将 b 定义为含有 MAX_TREE_SIZE 个 TElemType 类型元素的一维数组
```

2．链式存储结构

所谓二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示着元素的逻辑关系。通常有下面两种形式。

（1）二叉链表

链表中每个结点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。结点的存储的结构为：

lchild	data	rchild

其中，data 域存放某结点的数据信息；lchild 与 rchild 分别存放指向左孩子和右孩子的指针，当左孩子或右孩子不存在时，相应指针域值为空。

二叉树的二叉链表存储表示可描述为：

```
typedef struct BiTNode{
    TElemType data;
    struct BiTNode *lchild;*rchild;      //左右孩子指针
}BiTNode,*BiTree;
BiTree b；//即将 b 定义为指向二叉链表结点结构的指针类型。
```

（2）三叉链表

每个结点由四个域组成，结点的存储的结构为：

lchild	data	rchild	parent

其中，data、lchild 以及 rchild 三个域的意义与二叉链表结构相同；parent 域为指向该结点双亲结点的指针。这种存储结构既便于查找孩子结点，又便于查找双亲结点；但是，相对于二叉链表存储结构而言，它增加了空间开销。

尽管在二叉链表中无法由结点直接找到其双亲，但由于二叉链表结构灵活，操作方便，因此，二叉链表是最常用的二叉树存储方式，本书后面所涉及到的二叉树的链式存储结构不加特别说明的都是指二叉链表结构。

3.2.3 二叉树的遍历

二叉树的遍历是指按照某种顺序访问二叉树中的每个结点，使每个结点被访问一次且仅被访问一次。遍历是二叉树中经常要用到的一种操作。因为在实际应用问题中，常常需要按一定顺序对二叉树中的每个结点逐个进行访问，查找具有某一特点的结点，然后对这些满足条件的结点进行处理。通过一次完整的遍历，可使二叉树中结点信息由非线性排列变为某种意义上的线性序列。也就是说，遍历操作实质是对一个非线性结构进行线性化操作。

1．先序遍历

先序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1）访问根结点；

（2）先序遍历根结点的左子树；

（3）先序遍历根结点的右子树。

```
void PreOrder（BiTree b）{
    if (b!=NULL){
        Visit（b->data）;          //访问结点的数据域
        PreOrder（b->lchild）;  //先序递归遍历 b 的左子树
```

```
        PreOrder（b->rchild）;  //先序递归遍历 b 的右子树
    }
}
```

2．中序遍历

中序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1）中序遍历根结点的左子树；

（2）访问根结点；

（3）中序遍历根结点的右子树。

```
void InOrder（BiTree b）{
    if (b!=NULL)){
        InOrder（b->lchild）;  //中序递归遍历 b 的左子树
        Visit（b->data）;      //访问结点的数据域
        InOrder（b->rchild）;  //中序递归遍历 b 的右子树
    }
}
```

3．后序遍历

后序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1）后序遍历根结点的左子树；

（2）后序遍历根结点的右子树。

（3）访问根结点；

```
void PostOrder（BiTree b）{
    if (b!=NULL)){
        PostOrder（b->lchild）; //后序递归遍历 b 的左子树
        PostOrder（b->rchild）; //后序递归遍历 b 的右子树
        Visit（b->data）;       //访问结点的数据域
    }
}
```

4．层次遍历

所谓二叉树的层次遍历，是指从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

由层次遍历的定义可以推知，在进行层次遍历时，对一层结点访问完后，再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问，这样一层一层进行，先遇到的结点先访问，这与队列的操作原则比较吻合。因此，在进行层次遍历时，可设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队列，然后从对头取出一个元素，每取一个元素，执行下面两个操作：

（1）访问该元素所指结点；

（2）若该元素所指结点的左、右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。

此过程不断进行，当队列为空时，二叉树的层次遍历结束。

在下面的层次遍历算法中，二叉树以二叉链表存放，一维数组 Queue[MAX_TREE_SIZE]用以实现队列，变量 front 和 rear 分别表示当前对队首元素和队尾元素在数组中的位置。

```
void LevelOrder（BiTree b）{
  BiTree Queue[MAX_TREE_SIZE];
  int front,rear;
  if (b==NULL) return;
  front=-1;
  rear=0;
  Queue[rear]=b;
  while(front!=rear) {
    Visit(Queue[++front]->data);          //访问队首结点数据域
    if (Queue[front]->lchild!=NULL)       //将队首结点的左孩子结点入队列
      Queue[++rear]= Queue[front]->lchild;
    if (Queue[front]->rchild!=NULL)       //将队首结点的右孩子结点入队列
      Queue[++rear]= Queue[front]->rchild;
  }
}
```

5．二叉树遍历的非递归实现

前面给出的二叉树先序、中序和后序三种遍历算法都是递归算法。当给出二叉树的链式存储结构以后，用具有递归功能的程序设计语言很方便就能实现上述算法。然而，并非所有程序设计语言都允许递归；另一方面，递归程序虽然简洁，但执行效率不高。因此，就存在如何把一个递归算法转化为非递归算法的问题。解决这个问题的方法可以通过对三种遍历方法的实质过程的分析得到。

三种遍历路线正是从根结点开始沿左子树深入下去，当深入到最左端，无法再深入下去时，则返回，再逐一进入刚才深入时遇到结点的右子树，再进行如此的深入和返回，直到最后从根结点的右子树返回到根结点为止。先序遍历是在深入时（第一次经过）遇到结点就访问，中序遍历是在从左子树返回时（第二次经过）遇到结点访问，后序遍历是在从右子树返回时（第三次经过）遇到结点访问。

在这一过程中，返回结点的顺序与深入结点的顺序相反，即后深入先返回，正好符合栈结构后进先出的特点。因此，可以用栈来帮助实现这一遍历路线。其过程如下：在沿左子树深入时，深入一个结点入栈一个结点，若为先序遍历，则在入栈之前访问之；当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点，若为中序遍历，则此时访问该结点，然后从该结点的右子树继续深入；若为后序遍历，则将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍为深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

（1）前序遍历的非递归实现

二叉树以二叉链表存放，一维数组 Stack[MAX_TREE_SIZE]用以实现栈，变量 top 用来表示当前栈顶的位置。

```
void NRPreOrder（BiTree b）{          //非递归先序遍历二叉树
```

```
BiTree Stack[MAX_TREE_SIZE],p;
int top=0;
if (b==NULL) return;
p=b;
while(!(p==NULL&&top==0)) {
    while(p!=NULL) {
        Visit(p->data);                    //访问结点的数据域
        if(top＜ MAX_TREE_SIZE-1) //将当前指针 p 压栈
            Stack[top++]=p;
        else {
            printf("栈溢出");
            return;
        }
        p=p->lchild;                       //指针指向 p 的左孩子结点
    }
    if (top<=0) return;                    //栈空时结束
    else {
        p=Stack[- -top];                   //从栈中弹出栈顶元素
        p=p->rchild;                       //指针指向 p 的右孩子结点
    }
}
}
```

（2）中序遍历的非递归实现

只需将先序遍历的非递归算法中的 Visit(p->data)移到 p=Stack[- -top]和 p=p->rchild 之间即可。

3.2.4 线索二叉树

1.线索二叉树的定义

按照某种遍历方式对二叉树进行遍历，可把二叉树中所有结点排列为一个线性序列，二叉树中每个结点在这个序列中的直接前驱结点和直接后继结点是什么，二叉树的存储结构中并没有反映出来，只能在对二叉树遍历的动态过程中得到这些信息。为了保留结点在某种遍历序列中直接前驱和直接后继的位置信息，可以利用具有 n 个结点的二叉树中的叶子结点和一度结点的 n＋1 个空指针域来指示，这些指向直接前驱结点和指向直接后继结点的指针被称为线索，加了线索的二叉树称为线索二叉树。线索二叉树将为二叉树的遍历提供许多便利，遍历时可不需要栈，也不需要递归了。

2.线索二叉树的结构

一个具有 n 个结点的二叉树若采用二叉链表存储结构，在 2n 个指针域中只有 n－1 个指针域是用来存储结点孩子的地址，而另外 n＋1 个指针域存放的都是 NULL。因此，可以利用某结点空的左指针域（lchild）指出该结点在某种遍历序列中的直接前驱结点的存储地址，利

用结点空的右指针域（rchild）指出该结点在某种遍历序列中的直接后继结点的存储地址；对于那些非空的指针域，则仍然存放指向该结点左、右孩子的指针。这样，就得到了一棵线索二叉树。

由于序列可由不同的遍历方法得到，因此，线索树有先序线索二叉树、中序线索二叉树和后序线索二叉树三种。把二叉树改造成线索二叉树的过程称为线索化。

那么，在存储中，如何区别某结点的指针域内存放的是指针还是线索？

通常为每个结点增设两个标志位域 ltag 和 rtag，令：

$$ltag=\begin{cases}0 & \text{lchild 指向结点的左孩子}\\ 1 & \text{lchild 指向结点的前驱结点}\end{cases}$$

$$rtag=\begin{cases}0 & \text{rchild 指向结点的右孩子}\\ 1 & \text{rchild 指向结点的后继结点}\end{cases}$$

每个标志位令其只占一个 bit，这样就只需增加很少的存储空间。线索二叉树的结点结构为：

ltag	lchild	data	rchild	rtag

线索二叉树的存储表示的描述：

```
typedef enum PointerTag {Link,Thread};
typedef struct BiThrNode {
    TElemType          data;
    struct BiThrNode   *lchild,*rchild;
    PointerTag         LTag,RTag;
}BiThrNode, *BiThrTree;
```

为了将二叉树中所有空指针域都利用上，以及操作便利的需要，在存储线索二叉树时往往增设一头结点，其结构与其它线索二叉树的结点结构一样，只是其数据域不存放信息，其左指针域指向二叉树的根结点，右指针域指向自己。而原二叉树在某序遍历下的第一个结点的前驱线索和最后一个结点的后继线索都指向该头结点。

3.线索二叉树的基本操作实现

（1）建立一棵中序线索二叉树

实质上就是遍历一棵二叉树。在遍历过程中，Visit 操作是检查当前结点的左、右指针域是否为空，如果为空，将它们改为指向前驱结点或后继结点的线索。为实现这一过程，设指针 pre 始终指向刚刚访问过的结点，即若指针 p 指向当前结点，则 pre 指向它的前驱，以便增设线索。

另外，在对一棵二叉树加线索时，必须首先申请一个头结点，建立头结点与二叉树的根结点的指向关系，对二叉树线索化后，还需建立最后一个结点与头结点之间的线索。

```
int InOrderThr(BiThrTree &head,BiThrTree T) {
//中序遍历二叉树 T，将其中序线索化，head 指向头结点，其中 pre 为全局变量。
    if (!(head=(BiThrNode *)malloc(sizeof(BiThrNode))))   return 0;
    head->LTag=0;
```

```
        head->RTag=1;                  //建立头结点
        head->rchild=head;             //右指针回指
        if (!T) head->lchild =head;    //若二叉树为空，则左指针回指
        else{
              head->lchild=T;
              pre= head;
              InThreading(T);           //中序遍历进行中序线索化
              pre->rchild=head;
              pre->RTag=1;              //最后一个结点线索化
              head->rchild=pre;
        }
      return OK;
    }
    void InTreading(BiThrTree p){
        if (p) {
              InThreading(p->lchild);      //左子树线索化
              if (!p->lchild) {            //前驱线索
                    p->LTag=1;
                    p->lchild=pre;
              }
              if (!pre->rchild) {          //后继线索
                    pre->RTag=1;
                    pre->rchild=p;
              }
              pre=p;
              InThreading(p->rchild);      //右子树线索化
        }
    }
```

（2）在中序线索二叉树上查找任意结点的中序前驱结点

对于中序线索二叉树上的任一结点，寻找其中序的前驱结点，有以下两种情况：

①如果该结点的左标志为 1，那么其左指针域所指向的结点便是它的前驱结点；

②如果该结点的左标志为 0，表明该结点有左孩子，根据中序遍历的定义，它的前驱结点是以该结点的左孩子为根结点的子树的最右结点，即沿着其左子树的右指针链向下查找，当某结点的右标志为 1 时，它就是所要找的前驱结点。

```
    BiThrTree InPreNode（BiThrTree p）{
        BiThrTree pre;
        pre=p->lchild;
        if (p->ltag!=1)
            while (pre->rtag==0) pre=pre->rchild;
```

```
    return(pre);
}
```

（3）在中序线索二叉树上查找任意结点的中序后继结点

对于中序线索二叉树上的任一结点，寻找其中序的后继结点，有以下两种情况：

①如果该结点的右标志为 1，那么其右指针域所指向的结点便是它的后继结点；

②如果该结点的右标志为 0，表明该结点有右孩子，根据中序遍历的定义，它的前驱结点是以该结点的右孩子为根结点的子树的最左结点，即沿着其右子树的左指针链向下查找，当某结点的左标志为 1 时，它就是所要找的后继结点。

```
BiThrTree InPostNode（BiThrTree p）{
    BiThrTree post;
    post=p->rchild;
    if (p->rtag!=1)
        while (post->rtag==0) post=post->lchild;
    return(post);
}
```

以上给出的仅是在中序线索二叉树中寻找某结点的前驱结点和后继结点的算法。在前序线索二叉树中寻找结点的后继结点以及在后序线索二叉树中寻找结点的前驱结点可以采用同样的方法分析和实现。在此就不再讨论了。

3.3 树和森林

3.3.1 树的存储结构

在计算机中，树的存储有多种方式，既可以采用顺序存储结构，也可以采用链式存储结构，但无论采用何种存储方式，都要求存储结构不但能存储各结点本身的数据信息，还要能唯一地反映树中各结点之间的逻辑关系。下面介绍几种基本的树的存储方式。

1．双亲表示法

由树的定义可以知道，树中的每个结点都有唯一的一个双亲结点，根据这一特性，可用一组连续的存储空间（一维数组）存储树中的各个结点，数组中的一个元素表示树中的一个结点，数组元素为结构体类型，其中包括结点本身的信息以及结点的双亲结点在数组中的序号，树的这种存储方法称为双亲表示法。其存储表示可描述为：

```
#define MAXNODE    100 //  树中结点的最大个数
typedef struct {
    TElemType    data;
    int    parent;
}NodeType;
NodeType    t[MAXNODE];
```

2．孩子链表表示法

孩子链表法主体是一个与结点个数一样大小的一维数组，数组的每一个元素有两个域组成，一个域用来存放结点信息，另一个用来存放指针，该指针指向由该结点孩子组成的单链

表的首位置。单链表的结构也由两个域组成，一个存放孩子结点在一维数组中的序号，另一个是指针域，指向下一个孩子。这种存储表示可描述为：

```
#define MAXNODE 100    //树中结点的最大个数
typedef struct ChildNode{
    int childcode;
    struct ChildNode *nextchild;
}
typedef struct {
    TElemType    data;
    struct ChildNode *firstchild;
 }NodeType;
NodeType    t[MAXNODE];
```

3．孩子兄弟表示法

这是一种常用的存储结构。在树中，每个结点除其信息域外，再增加两个分别指向该结点的第一个孩子结点和下一个兄弟结点的指针。在这种存储结构下，树中结点的存储表示可描述为：

```
typedef struct TreeNode {
    TElemType data;
    struct TreeNode    *son;
    struct TreeNode    *next;
}NodeType;
```

3.3.2 森林和二叉树的转换

从树的孩子兄弟表示法可以看到，如果设定一定规则，就可用二叉树结构表示树和森林，这样，对树的操作实现就可以借助二叉树存储，利用二叉树上的操作来实现。（详见 PPT）

3.3.3 树和森林的遍历

1．树的遍历

（1）先根遍历

先根遍历的定义为：

①访问根结点；

②按照从左到右的顺序先根遍历根结点的每一棵子树。

（2）后根遍历

后根遍历的定义为：

①按照从左到右的顺序后根遍历根结点的每一棵子树。

②访问根结点；

根据树与二叉树的转换关系以及树和二叉树的遍历定义可以推知，树的先根遍历与其转换的相应二叉树的先序遍历的结果序列相同；树的后根遍历与其转换的相应二叉树的中序遍历的结果序列相同。因此树的遍历算法是可以采用相应二叉树的遍历算法来实现的。

2.森林的遍历

（1）前序遍历

前序遍历的定义为：

①访问森林中第一棵树的根结点；

②前序遍历第一棵树的根结点的子树；

③前序遍历去掉第一棵树后的子森林。

（2）中序遍历

中序遍历的定义为：

①中序遍历第一棵树的根结点的子树；

②访问森林中第一棵树的根结点；

③中序遍历去掉第一棵树后的子森林。

根据森林与二叉树的转换关系以及森林和二叉树的遍历定义可以推知，森林的前序遍历和中序遍历与所转换的二叉树的先序遍历和中序遍历的结果序列相同。

3.4 哈夫曼（Huffman）树和哈夫曼编码

1．哈夫曼树的基本概念

二叉树的路径长度是指由根结点到所有叶结点的路径长度之和。如果二叉树中的叶结点都具有一定的权值，则可将这一概念加以推广。设二叉树具有 n 个带权值的叶结点，那么从根结点到各个叶结点的路径长度与相应结点权值的乘积之和叫做二叉树的带权路径长度，记为：

$$WPL=\sum_{k=1}^{n} W_k \cdot L_k$$

其中 W_k 为第 k 个叶结点的权值，L_k 为第 k 个叶结点的路径长度。

最优二叉树，也称哈夫曼（Haffman）树，是指对于一组带有确定权值的叶结点，构造的具有最小带权路径长度的二叉树。由相同权值的一组叶子结点所构成的二叉树有不同的形态和不同的带权路径长度，那么如何找到带权路径长度最小的二叉树（即哈夫曼树）呢？根据哈夫曼树的定义，一棵二叉树要使其 WPL 值最小，必须使权值越大的叶结点越靠近根结点，而权值越小的叶结点越远离根结点。哈夫曼（Haffman）依据这一特点提出了一种方法，这种方法的基本思想是：

（1）由给定的 n 个权值{W1，W2，…，Wn}构造 n 棵只有一个叶结点的二叉树，从而得到一个二叉树的集合 F＝{T1，T2，…，Tn}；

（2）在 F 中选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树，这棵新的二叉树根结点的权值为其左、右子树根结点权值之和；

（3）在集合 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合 F 中；

（4）重复（2）（3）两步，当 F 中只剩下一棵二叉树时，这棵二叉树便是所要建立的哈夫曼树。

2.哈夫曼编码

在数据通讯中，经常需要将传送的文字转换成由二进制字符 0，1 组成的二进制串，称之为编码。

在哈夫曼编码树中，树的带权路径长度的含义是各个字符的码长与其出现次数的乘积之

和，也就是电文的代码总长，所以采用哈夫曼树构造的编码是一种能使电文代码总长最短的不等长编码。

然而，采用哈夫曼树进行编码，则不会产生上述二义性问题。因为，在哈夫曼树中，每个字符结点都是叶结点，它们不可能在根结点到其它字符结点的路径上，所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀，从而保证了译码的非二义性。

求哈夫曼编码，实质上就是在已建立的哈夫曼树中，从叶结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值，由于一个字符的哈夫曼编码是从根结点到相应叶结点所经过的路径上各分支所组成的 0，1 序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为所求编码的高位码。

习题

1、引入二叉线索树的目的是（ ）。

A．加快查找结点的前驱或后继的速度

B．为了能在二叉树中方便的进行插入与删除

C．为了能方便的找到双亲

D．使二叉树的遍历结果唯一

2、已知某二叉树的中序、层序序列为DBAFCE、FDEBCA，则该二叉树的后序序列为（ ）。

A. BCDEAF　　B. ABDCEF

C. DBACEF　　D. DABECF

3、设结点x和y是二叉树中任意的两个结点，在该二叉树的前序遍历序列中x在y之前，而在其后序遍历序列中x在y之后，则x和y的关系是（ ）。

A．x是y的左兄弟　　B. x是y的右兄弟

C．x是y的祖先　　D. x是y的后裔

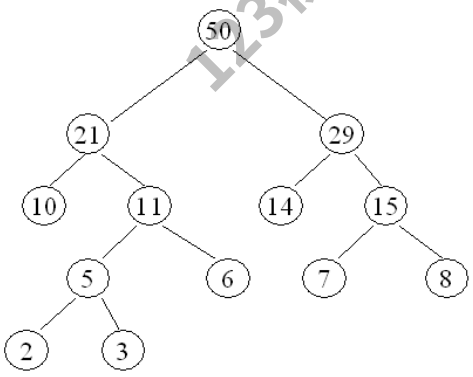
4、设n，m为一棵二叉树的两个结点，在中序遍历时，n在m前的条件是（ ）。

A. n在m的右方　　B. n是m的祖先

C. n在m的左方　　D. n是m的子孙

5、应用题：假定一篇电文仅由a,b,c,d,e,f,g七个字母组成，字母a,b,c,d,e,f,g出现频度分别为2,3,6,7,8,10,14，试以它们为叶子结点构造一棵哈夫曼树，并给出哈夫曼编码？计算平均长度是多少？

答：构造的哈夫曼树如下：



哈夫曼编码如下：约定左分支表示字符‘0’，右分支表示字符‘1’。

a: 0100

b: 0101

c: 011

d: 110

e: 111

f: 00

g: 10

平均长度= 2*4+3*4+6*3+7*3+8*3+10*2+14*2)/50 = 131/50 = 2.62

6、编写算法求以孩子—兄弟表示法存储的森林的叶子结点数。

```
typedef   struct   node
{ElemType   data; //数据域
  struct   node *fch, *nsib; //孩子与兄弟域
}*Tree;
int Leaves (Tree t)
{   if(t=null) return 0;
    else   if(t->fch==null) //若结点无孩子,则该结点必是叶子
                return(1+Leaves(t->nsib));
                //返回叶子结点和其兄弟子树中的叶子结点数
           else return (Leaves(t->fch)+Leaves(t->nsib));
                //孩子子树和兄弟子树中叶子数之和
}
```

第四章 图

4.1 图的概念

1．图的定义

图是由一个顶点集 V 和一个弧集 R 构成的数据结构。

ADT Graph {

数据对象 V：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系 R：

R＝{VR}

VR＝{<v,w>| v,w∈V 且 P(v,w)，<v,w>表示从 v 到 w 的弧，

谓词 P(v,w)定义了弧<v,w>的意义或信息 }

2.图的重要术语；

（1）无向图：在一个图中，如果任意两个顶点构成的偶对（v,w）∈E 是无序的，即顶点之间的连线是没有方向的，则称该图为无向图。

（2）有向图：在一个图中，如果任意两个顶点构成的偶对（v,w）∈E 是有序的，即顶点之间的连线是有方向的，则称该图为有向图。

（3）无向完全图：在一个无向图中，如果任意两顶点都有一条直接边相连接，则称该图为无向完全图。在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。

（4）有向完全图：在一个有向图中，如果任意两顶点之间都有方向互为相反的两条弧相连接，则称该图为有向完全图。在一个含有 n 个顶点的有向完全图中，有 n(n-1)条边。

（5）稠密图、稀疏图：若一个图接近完全图，称为稠密图；称边数很少（**e<nlogn**）的图为稀疏图。

（6）顶点的度、入度、出度：

顶点的度（degree）是指依附于某顶点 v 的边数，通常记为 TD (v)。

在有向图中，要区别顶点的入度与出度的概念。顶点 v 的入度是指以顶点为终点的弧的数目，记为 ID (v)；顶点 v 出度是指以顶点 v 为始点的弧的数目，记为 OD (v)。

TD (v)=ID (v)＋OD (v)。

可以证明，对于具有 n 个顶点、e 条边的图，顶点 v_i 的度 TD (v_i)与顶点的个数以及边的数目满足关系：

$$e=\left(\sum_{i=1}^{n} TD(v_i)\right)/2$$

（7）边的权、网图：与边有关的数据信息称为权（weight）。在实际应用中，权值可以有某种含义。边上带权的图称为网图或网络（network）。如果边是有方向的带权图，则就是一个有向网图。

（8）路径、路径长度：顶点 v_p 到顶点 v_q 之间的路径（path）是指顶点序列 v_p,v_{i1},v_{i2}，…，v_{im},v_q。其中，（v_p,v_{i1}），(v_{i1},v_{i2})，…,($v_{im},.v_q$)分别为图中的边。

路径上边的数目称为路径长度。

（9）简单路径、简单回路：序列中顶点不重复出现的路径称为简单路径。除第一个顶点与最后一个顶点之外，其他顶点不重复出现的回路称为简单回路，或者简单环。

（10）子图：对于图 G=（V，E），G'=（V'，E'），若存在 V'是 V 的子集 ，E'是 E 的子集，则称图 G'是 G 的一个子图。

（11）连通图、连通分量：在无向图中，如果从一个顶点 v_i 到另一个顶点 $v_j(i\neq j)$有路径，则称顶点 v_i 和 v_j 是连通的。如果图中任意两顶点都是连通的，则称该图是连通图。无向图的极大连通子图称为连通分量。

（12）强连通图、强连通分量：对于有向图来说，若图中任意一对顶点 v_i 和 $v_j(i\neq j)$均有从一个顶点 v_i 到另一个顶点 v_j 有路径，也有从 v_j 到 v_i 的路径，则称该有向图是强连通图。有向图的极大强连通子图称为强连通分量。

（13）生成树：所谓连通图 G 的生成树，是 G 的包含其全部 n 个顶点的一个极小连通子图。它必定包含且仅包含 G 的 n-1 条边。在生成树中添加任意一条属于原图中的边必定会产生回路，因为新添加的边使其所依附的两个顶点之间有了第二条路径。若生成树中减少任意一条边，则必然成为非连通的。

（14）生成森林：在非连通图中，由每个连通分量都可得到一个极小连通子图，即一棵生成树。这些连通分量的生成树就组成了一个非连通图的生成森林。

4.2 图的存储及基本操作

4.2.1 邻接矩阵

所谓邻接矩阵存储结构，就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。

假设图 G＝（V，E）有 n 个确定的顶点，即 $V=\{v_0,v_1,\ldots,v_{n-1}\}$,则表示 G 中各顶点相邻关系为一个 n×n 的矩阵，矩阵的元素为：

$$A[i][j]=\begin{cases}1 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

若 G 是网图，则邻接矩阵可定义为：

$$A[i][j]=\begin{cases}w_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0\text{或}\infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

其中，w_{ij} 表示边(v_i,v_j)或$<v_i,v_j>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

从图的邻接矩阵存储方法容易看出，这种表示具有以下特点：

（1）无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。

（2）对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的度 $TD(v_i)$。

（3）对于有向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的出度 $OD(v_i)$（或入度 $ID(v_i)$）。

（4）用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，

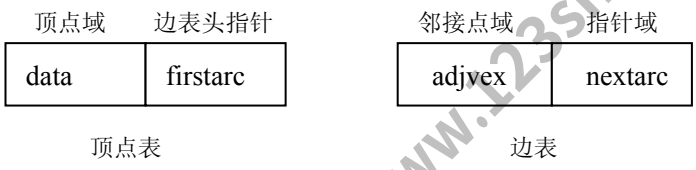
要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大，这是用邻接矩阵存储图的局限性。

在用邻接矩阵存储图时，除了用一个二维数组存储用于表示顶点间相邻关系的邻接矩阵外，还需用一个一维数组来存储顶点信息，另外还有图的顶点数和边数。故可将其形式描述如下：

```
#define MAX_VERTEX_NUM 20        //最大顶点数设为 20
typedef char VertexType;         //顶点类型设为字符型
typedef int VRType;              //边的权值设为整型
typedef struct {
    VertexType vexs[MAX_VERTEX_NUM];                    //顶点表
    VRType edges[MAX_VERTEX_NUM][ MAX_VERTEX_NUM];   //邻接矩阵，即边表
    int vexnum,arcnum;           //顶点数和边数
}MGragh;                         //MGragh 是邻接矩阵存储的图类型
```

4.2.2 邻接表

邻接表(Adjacency List)是图的一种顺序存储与链式存储结合的存储方法。邻接表表示法类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 v_i，将所有邻接于 v_i 的顶点 v_j 链成一个单链表，这个单链表就称为顶点 v_i 的邻接表，再将所有点的邻接表表头放到数组中，就构成了图的邻接表。在邻接表表示中有两种结点结构，如图所示。



邻接矩阵表示的结点结构

一种是顶点表的结点结构，它由顶点域（data）和指向第一条邻接边的指针域（firstarc）构成，另一种是边表（即邻接表）结点，它由邻接点域(adjvex)和指向下一条邻接边的指针域(nextarc)构成。对于网图的边表需再增设一个存储边上信息（如权值等）的域（info）。

邻接表表示的形式描述如下：

```
＃define MAX_VERTEX_NUM   20      //最大顶点数为 20
typedef struct ArcNode{           //边表结点
    int      adjvex;              //邻接点域
    struct ArcNode *nextarc;      //指向下一个邻接点的指针域
                                  //若要表示边上信息，则应增加一个数据域 info
}ArcNode;
typedef struct VNode{             //顶点表结点
    VertexType       data;        //顶点域
    ArcNode      *firstarc;       //边表头指针
}VNode, AdjList[MAX_VERTEX_NUM]; //AdjList 是邻接表类型
typedef struct{
```

```
    AdjList adjlist;                    //邻接表
    int vexnum,arcnum;                  //顶点数和边数
}ALGraph;                               //ALGraph 是以邻接表方式存储的图类型
```

从图的邻接表存储方法容易看出，这种表示具有以下特点：

（1）若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 2e 个表结点。显然，在边稀疏(e<<n(n-1)/2)的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

（2）在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数.

（3）而在有向图中，第 i 个链表中的结点个数只是顶点 v_i 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。

有时，为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的链表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的复杂度为 O（n+e），否则，需要通过查找才能得到顶点在图中位置，则时间复杂度为 O（n·e）。

（4）在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（v_i 和 v_j）之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

4.3 图的遍历

图的遍历是指从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。图的遍历操作和树的遍历操作功能相似。图的遍历是图的一种基本操作，图的许多其它操作都是建立在遍历操作的基础之上。图的遍历通常有深度优先搜索和广度优先搜索两种方式。

4.3.1 深度优先搜索

深度优先搜索（Depth_Fisrst Search）遍历类似于树的先根遍历，是树的先根遍历的推广。

假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可从图中某个顶点发 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

显然，这是一个递归的过程。为了在遍历过程中便于区分顶点是否已被访问，需附设访问标志数组 visited[0:n-1], ，其初值为 FALSE ，一旦某个顶点被访问，则其相应的分量置为 TRUE。

从图的某一点 v 出发，递归地进行深度优先遍历的过程算法如下。

```
void DFSTraverse (Graph G) {          //深度优先遍历图 G
    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;           //访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v]) DFS(G,v);    //对尚未访问的顶点调用 DFS
}
```

```
void DFS(Graph G,int v ) {              //从第 v 个顶点出发递归地深度优先遍历图 G
    visited[v]=TRUE;Visit(v);           //访问第 v 个顶点
    for (w=FisrtAdjVex(G,v);w>=0; w=NextAdjVex(G,v,w))
        if (!visited[w]) DFS(G,w);      //对 v 的尚未访问的邻接顶点 w 递归调用 DFS
}
```

分析上述算法，在遍历时，对图中每个顶点至多调用一次 DFS 函数，因为一旦某个顶点被标志成已被访问，就不再从它出发进行搜索。因此，遍历图的过程实质上是对每个顶点查找其邻接点的过程。其耗费的时间则取决于所采用的存储结构。当用二维数组表示邻接矩阵图的存储结构时，查找每个顶点的邻接点所需时间为 $O(n^2)$ ，其中 n 为图中顶点数。而当以邻接表作图的存储结构时，找邻接点所需时间为 O(e)，其中 e 为无向图中边的数或有向图中弧的数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为 O(n+e) 。

4.3.2 广度优先搜索

广度优先搜索（Breadth_First Search） 遍历类似于树的按层次遍历的过程。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过和邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。换句话说，广度优先搜索遍历图的过程中以 v 为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1,2,…的顶点。

广度优先搜索和深度优先搜索类似，在遍历的过程中也需要一个访问标志数组。并且，为了顺次访问路径长度为 2、3、…的顶点，需附设队列以存储已被访问的路径长度为 1、2、…的顶点。

从图的某一点 v 出发，递归地进行广度优先遍历的过程算法如下。

```
void BFSTraverse (Graph G) {            //按广度优先非递归遍历图 G，使用辅助队列 Q
    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;             //访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v])  BFS(G, v); //对尚未访问的顶点调用 BFS
}
void BFS (Graph G,int v) {
    InitQueue(Q);                       //置空的辅助队列 Q
    visited[v]=TRUE; Visit(v);          //访问 v
    EnQueue(Q,v);                       //v 入队列
    while (!QueueEmpty(Q)) {
        DeQueue(Q,u);                   //队头元素出队并置为 u
        for(w=FistAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w))
        if (!visited[w]){
            visited[w]=TRUE; Visit(w);
```

```
                EnQueue(Q,w);          //u 尚未访问的邻接顶点 w 入队列 Q
            }
        }
    }
```

分析上述算法，每个顶点至多进一次队列。遍历图的过程实质是通过边或弧找邻接点的过程，因此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，两者不同之处仅仅在于对顶点访问的顺序不同。

4.4 图的基本应用

4.4.1 最小生成树

1.最小生成树的基本概念

由生成树的定义可知，无向连通图的生成树不是唯一的。连通图的一次遍历所经过的边的集合及图中所有顶点的集合就构成了该图的一棵生成树，对连通图的不同遍历，就可能得到不同的生成树。可以证明，对于有 n 个顶点的无向连通图，无论其生成树的形态如何，所有生成树中都有且仅有 n－1 条边。

如果无向连通图是一个网，那么，它的所有生成树中必有一棵边的权值总和最小的生成树，我们称这棵生成树为最小生成树，简称为最小生成树。

2.构造最小生成树的 Prim 算法

假设 G＝（V，E）为一网图，其中 V 为网图中所有顶点的集合，E 为网图中所有带权边的集合。设置两个新的集合 U 和 T，其中集合 U 用于存放 G 的最小生成树中的顶点，集合 T 存放 G 的最小生成树中的边。令集合 U 的初值为 U＝{u1}（假设构造最小生成树时，从顶点 u1 出发），集合 T 的初值为 T＝{}。

Prim 算法的思想是:从所有 u∈U，v∈V－U 的边中，选取具有最小权值的边（u，v），将顶点 v 加入集合 U 中，将边（u，v）加入集合 T 中，如此不断重复，直到 U＝V 时，最小生成树构造完毕，这时集合 T 中包含了最小生成树的所有边。

Prim 算法可用下述过程描述，其中用 w_{uv} 表示顶点 u 与顶点 v 边上的权值。

（1）U＝{u1},T={};

（2）while (U≠V)do

(u，v)＝min{w_{uv}；u∈U，v∈V－U }

T＝T＋{(u，v)}

U＝U＋{v}

（3）结束。

Prim 算法的时间复杂度为 $O(n^2)$，与网中的边数无关，因此适用于求边稠密的网的最小生成树。

3.构造最小生成树的 Kruskal 算法

Kruskal 算法是一种按照网中边的权值递增的顺序构造最小生成树的方法。其基本思想是：设无向连通网为 G＝（V，E），令 G 的最小生成树为 T，其初态为 T＝（V，{}），即开始时，最小生成树 T 由图 G 中的 n 个顶点构成，顶点之间没有一条边，这样 T 中各顶点各自

构成一个连通分量。然后，按照边的权值由小到大的顺序，考察 G 的边集 E 中的各条边。若被考察的边的两个顶点属于 T 的两个不同的连通分量，则将此边作为最小生成树的边加入到 T 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

Kruskal 算法需对 e 条边按权值进行排序，时间复杂度为 O(eloge)(e 为网中边的数目)，因此适用于求边稀疏的网的最小生成树。

4.4.2 最短路径

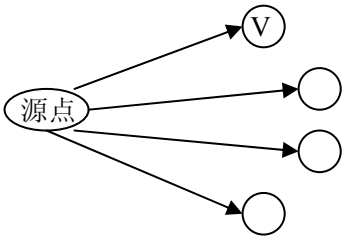
1.从一个源点到其它各点的最短路径

单源点的最短路径问题：给定带权有向图 G＝（V，E）和源点 v∈V，求从 v 到 G 中其余各顶点的最短路径。

（1）求从源点到其余各点的最短路径的算法的基本思想：

依最短路径的长度递增的次序求得各条路径。

①假设图中所示为从源点到其余各点之间的最短路径，则在这些路径中，必然存在一条长度最短者。其中，从源点到顶点 v 的最短路径是所有最短路径中长度最短者。



例图

②路径长度最短的最短路径的特点：

在这条路径上，必定只含一条弧，并且这条弧的权值最小。

③下一条路径长度次短的最短路径的特点：

它只可能有两种情况：或者是直接从源点到该点(只含一条弧)；或者是，从源点经过顶点 v1，再到达该顶点(由两条弧组成)。

④再下一条路径长度次短的最短路径的特点:

它可能有三种情况：或者是，直接从源点到该点(只含一条弧)；或者是，从源点经过顶点 v1，再到达该顶点(由两条弧组成)；或者是，从源点经过顶点 v2，再到达该顶点。

⑤其余最短路径的特点：

它或者是直接从源点到该点(只含一条弧)；或者是，从源点经过已求得最短路径的顶点，再到达该顶点。

（2）迪杰斯特拉（Dijkstra）算法

迪杰斯特拉提出的一个按路径长度递增的次序产生最短路径的算法。

该算法的基本思想是：设置两个顶点的集合 S 和 T＝V－S，集合 S 中存放已找到最短路径的顶点，集合 T 存放当前还未找到最短路径的顶点。初始状态时，集合 S 中只包含源点 v_0，然后不断从集合 T 中选取到顶点 v_0 路径长度最短的顶点 u 加入到集合 S 中，集合 S 每加入一

个新的顶点 u，都要修改顶点 v_0 到集合 T 中剩余顶点的最短路径长度值，集合 T 中各顶点新的最短路径长度值为原来的最短路径长度值与顶点 u 的最短路径长度值加上 u 到该顶点的路径长度值中的较小值。此过程不断重复，直到集合 T 的顶点全部加入到 S 中为止。

Dijkstra 算法的正确性可以用反证法加以证明。假设下一条最短路径的终点为 x，那么，该路径必然或者是弧（v_0，x），或者是中间只经过集合 S 中的顶点而到达顶点 x 的路径。因为假若此路径上除 x 之外有一个或一个以上的顶点不在集合 S 中，那么必然存在另外的终点不在 S 中而路径长度比此路径还短的路径，这与我们按路径长度递增的顺序产生最短路径的前提相矛盾，所以此假设不成立。

下面介绍 Dijkstra 算法的实现。

首先，引进一个辅助向量 D，它的每个分量 D[i] 表示当前所找到的从始点 v 到每个终点 v_i 的最短路径的长度。它的初态为：若从 v 到 v_i 有弧，则 D[i]为弧上的权值；否则置 D[i]为∞。显然，长度为：

$$D[j]=Min\{D[i]|\ v_i \in V\}$$

的路径就是从 v 出发的长度最短的一条最短路径。此路径为（v ,vj）。

那么，下一条长度次短的最短是哪一条呢？假设该次短路径的终点是 v_k，则可想而知，这条路径或者是（v, v_k），或者是（v, v_j, v_k）。它的长度或者是从 v 到 v_k 的弧上的权值，或者是 D[j]和从 v_j 到 v_k 的弧上的权值之和。

依据前面介绍的算法思想，在一般情况下，下一条长度次短的最短路径的长度必是：

$$D[j]=Min\{D[i]|\ vi \in V\text{-}S\}$$

其中，D[i] 或者弧（v, v_i）上的权值，或者是 D[k]($v_k \in S$ 和弧（v_k, v_i）上的权值之和。

根据以上分析，可以得到如下描述的算法：

①假设用带权的邻接矩阵 edges 来表示带权有向图，edges[i][j] 表示弧〈v_i, v_j〉上的权值 。若〈v_i, v_j〉不存在，则置 edges[i][j]为∞（在计算机上可用允许的最大值代替）。S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。那么，从 v 出发到图上其余各顶点（终点）v_i 可能达到最短路径长度的初值为：

$$D[i]= edges[Locate\ Vex(G,v)][i] \quad vi \in V$$

②选择 v_j，使得

$$D[j]=Min\{D[i]|\ vi \in V\text{-}S\}$$

v_j 就是当前求得的一条从 v 出发的最短路径的终点。令

$$S = S \cup \{j\}$$

③修改从 v 出发到集合 V-S 上任一顶点 v_k 可达的最短路径长度。如果

$$D[j]+ edges[j][k]<D[k]$$

则修改 D[k]为

$$D[k]=D[j]+ edges[j][k]$$

④重复操作②、③共 n-1 次。由此求得从 v 到图上其余各顶点的最短路径是依路径长度递增的序列。

Dijkstra 算法的时间复杂度为 $O(n^2)$。

2.每一对顶点之间的最短路径

解决这个问题的一个办法是：每次以一个顶点为源点，重复招待迪杰斯特拉算法 n 次。

这样，便可求得每一结顶点的最短路径。总的执行时间为 $O(n^3)$。

这里要介绍由弗洛伊德(Floyd)提出的另一个算法。这个算法的时间复杂度也是 $O(n^3)$,但形式上简单些。

弗洛伊德算法仍从图的带权邻接矩阵 cost 出发，其基本思想是：

假设求从顶点 v_i 到 v_j 的最短路径。如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 arcs[*i*][j] 的路径，该路径不一定是最短路径，尚需进行 *n* 次试探。首先考虑路径（v_i, v_0, v_j）是否存在（即判别弧（v_i, v_0）和(v_0, v_j)是否存在）。如果存在，则比较（v_i, v_j）和（v_i, v_0, v_j）的路径长度取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。假如在路径上再增加一个顶点 v_1，也就是说，如果 （v_i, …, v_1）和（v_1, …, v_j）分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么（v_i, …, v_1, … , v_j）就有可能是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径。将它和已经得到的从 v_i 到 v_j 中间顶点序号不大于 0 的最短路径相比较，从中选出中间顶点的序号不大于 1 的最短路径之后，再增加一个顶点 v_2，继续进行试探。依次类推。在一般情况下，若（v_i, …, v_k）和（v_k, …, v_j）分别是从 v_i 到 v_k 和从 v_k 到 v_j 的中间顶点的序号不大于 *k*-1 的最短路径，则将（v_i, …, v_k, …, v_j）和已经得到的从 v_i 到 v_j 且中间顶点序号不大于 *k*-1 的最短路径相比较，其长度较短者便是从 v_i 到 v_j 的中间顶点的序号不大于 *k* 的最短路径。这样，在经过 *n* 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。按此方法，可以同时求得各对顶点间的最短路径。

现定义一个 n 阶方阵序列。

$D^{(-1)}$，$D^{(0)}$，$D^{(1)}$，…，$D^{(k)}$，$D^{(n-1)}$

其中

$D^{(-1)}[i][j]=edges[i][j]$

$D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\}$　$0 \leqq k \leqq n-1$

从上述计算公式可见，$D^{(1)}[i][j]$是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径的长度；$D^{(k)}[i][j]$ 是从 v_i 到 v_j 的中间顶点的个数不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$ 就是从 v_i 到 v_j 的最短路径的长度。

4.4.3 拓扑排序

1.AOV 网

所有的工程或者某种流程可以分为若干个小的工程或阶段，这些小的工程或阶段就称为活动。若以图中的顶点来表示活动，有向边表示活动之间的优先关系，则这样活动在顶点上的有向图称为 AOV 网。在 AOV 网中，若从顶点 i 到顶点 j 之间存在一条有向路径，称顶点 i 是顶点 j 的前驱，或者称顶点 j 是顶点 i 的后继。若<i,j>是图中的弧，则称顶点 i 是顶点 j 的直接前驱，顶点 j 是顶点 i 的直接后驱。

2．拓扑排序

AOV 网所代表的一项工程中活动的集合显然是一个偏序集合。为了保证该项工程得以顺利完成，必须保证 AOV 网中不出现回路；否则，意味着某项活动应以自身作为能否开展的先决条件，这是荒谬的。

测试 AOV 网是否具有回路（即是否是一个有向无环图）的方法，就是在 AOV 网的偏序集合下构造一个线性序列，该线性序列具有以下性质：

① 在 AOV 网中，若顶点 i 优先于顶点 j ，则在线性序列中顶点 i 仍然优先于顶点 j；

② 对于网中原来没有优先关系的顶点 i 与顶点 j，在线性序列中也建立一个先后关系，或者顶点 i 优先于顶点 j ，或者顶点 j 优先于 i。

满足这样性质的线性序列称为拓扑有序序列。构造拓扑序列的过程称为拓扑排序。也可以说拓扑排序就是由某个集合上的一个偏序得到该集合上的一个全序的操作。

若某个 AOV 网中所有顶点都在它的拓扑序列中，则说明该 AOV 网不会存在回路，这时的拓扑序列集合是 AOV 网中所有活动的一个全序集合。显然，对于任何一项工程中各个活动的安排，必须按拓扑有序序列中的顺序进行才是可行的。

3.拓扑排序算法

对 AOV 网进行拓扑排序的方法和步骤是：

①从 AOV 网中选择一个没有前驱的顶点（该顶点的入度为 0）并且输出它；

②从网中删去该顶点，并且删去从该顶点发出的全部有向边；

③重复上述两步，直到剩余的网中不再存在没有前驱的顶点为止。

这样操作的结果有两种：一种是网中全部顶点都被输出，这说明网中不存在有向回路；另一种就是网中顶点未被全部输出，剩余的顶点均不前驱顶点，这说明网中存在有向回路。

对一个具有 n 个顶点、e 条边的网来说，整个算法的时间复杂度为 O（e+n）。

4.4.4 关键路径

1．AOE 网

若在带权的有向图中，以顶点表示事件，以有向边表示活动，边上的权值表示活动的开销（如该活动持续的时间），则此带权的有向图称为 AOE 网。

AOE 网具有以下两个性质：

①只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始。

② 只有在进入一某顶点的各有向边所代表的活动都已经结束，该顶点所代表的事件才能发生。

2．关键路径

由于 AOE 网中的某些活动能够同时进行，故完成整个工程所必须花费的时间应该为源点到终点的最大路径长度（这里的路径长度是指该路径上的各个活动所需时间之和）。具有最大路径长度的路径称为关键路径。关键路径上的活动称为关键活动。关键路径长度是整个工程所需的最短工期。这就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时要需解决的主要问题是：

①计算完成整个工程的最短路径。

②确定关键路径，以找出哪些活动是影响工程进度的关键。

3．关键路径的确定

为了在 AOE 网中找出关键路径，需要定义几个参量，并且说明其计算方法。

（1）事件的最早发生时间 ve[k]

ve[k]是指从源点到顶点的最大路径长度代表的时间。这个时间决定了所有从顶点发出的有向边所代表的活动能够开工的最早时间。根据 AOE 网的性质，只有进入 vk 的所有活动< vj,vk>都结束时，vk 代表的事件才能发生；而活动< vj, vk>的最早结束时间为 ve[j]+dut(< vj,

vk>)。所以计算 vk 发生的最早时间的方法如下：

$$\begin{cases} ve[l]=0 \\ ve[k]=Max\{ve[j]+dut(<vj, vk>)\} & <vj, vk>\in p[k] \end{cases} \quad (式\ 1\text{-}4\text{-}1)$$

其中，p[k]表示所有到达 vk 的有向边的集合；dut(< vj, vk>)为有向边< vj,vk>上的权值。

（2）事件的最迟发生时间 vl[k]

vl[k]是指在不推迟整个工期的前提下，事件 vk 允许的最晚发生时间。设有向边< vk，vj>代表从 vk 出发的活动，为了不拖延整个工期,vk 发生的最迟时间必须保证不推迟从事件 vk 出发的所有活动< vk，vj>的终点 vj 的最迟时间 vl[j]。vl[k] 的计算方法如下：

$$\begin{cases} vl[n]=ve[n] \\ vl[k]=Min\{vl[j]-dut(<vk，vj>)\} & <vk, vj>\in s[k] \end{cases} \quad (式\ 1\text{-}4\text{-}2)$$

其中，s[k]为所有从 vk 发出的有向边的集合。

（3）活动 ai 的最早开始时间 e[i]

若活动 ai 是由弧<vk,vj>表示，根据 AOE 网的性质，只有事件 vk 发生了，活动 ai 才能开始。也就是说，活动 ai 的最早开始时间应等于事件 vk 的最早发生时间。因此，有：

$$e[i]=ve[k] \quad (式\ 1\text{-}4\text{-}3)$$

（4）活动 ai 的最晚开始时间 l[i]

活动 ai 的最晚开始时间指，在不推迟整个工程完成日期的前提下，必须开始的最晚时间。若 由弧< vk,vj>>表示，则 ai 的最晚开始时间要保证事件 vj 的最迟发生时间不拖后。因此，应该有：

$$l[i]=vl[j]-dut(<vk,vj>) \quad (式\ 1\text{-}4\text{-}4)$$

根据每个活动的最早开始时间 e[i]和最晚开始时间 l[i]就可判定该活动是否为关键活动，也就是那些 l[i]=e[i]的活动就是关键活动，而那些 l[i]>e[i]的活动则不是关键活动，l[i]-e[i]的值为活动的时间余量。关键活动确定之后，关键活动所在的路径就是关键路径。

由上述方法得到求关键路径的算法步骤为：

（1）输入 e 条弧<j,k>,建立 AOE-网的存储结构；

（2）从源点 v_0 出发，令 ve[0]＝0，按拓扑有序求其余各顶点的最早发生时间 ve[i](1≤i≤n-1)。如果得到的拓扑有序序列中顶点个数小于网中顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤（3）。

（3）从汇点 v_n 出发，令 vl[n-1]=ve[n-1],按逆拓扑有序求其余各顶点的最迟发生时间 vl[i](n-2≥i≥2)；

（4）根据各顶点的 ve 和 vl 值，求每条弧 s 的最早开始时间 e(s)和最迟开始时间 1（s）。若某条弧满足条件 e(s)=l(s),则为关键活动。

习题

1、简单无向图的邻接矩阵是对称的，可以对其进行压缩存储。若无向图 G 有 n 个结点，其邻接矩阵为 A[1…n,1…n]，且压缩存储在 B[1…k]，则 k 的值至少为（ ）。

A．n(n+1)/2　　B. $n^2/2$

C. (n-1)(n+1)/2　　D. n(n-1)/2

分析：简单无向图的邻接矩阵是对称的，且对角线元素均是 0，故压缩存储只须存储下

三角或上三角（均不包括对角线）即可。

2、若一个具有 n 个结点、k 条边的非连通无向图是一个森林（n>k），则该森林中必有（ ）棵树。

A．k B. n C. n-k D. n+k

3、若 G 是一个具有 36 条边的非连通无向图（不含自回路和多重边），则图 G 至少有（ ）个顶点。

A．11 B. 10 C. 9 D. 8

【分析】n 个结点的无向图中，边数 e≤n(n-1)/2，将 e=36 代入，有 n≥9，现已知无向图非连通，则 n=10。

4、判断有向图是否存在回路，除了可以利用拓扑排序方法外，还可以利用（ ）。

A．求关键路径的方法

B．求最短路径的 DIJKSTRA 方法

C．深度优先遍历算法

D．广度优先遍历算法

【分析】当有向图中无回路时，从某顶点出发进行深度优先遍历时，出栈的顺序（退出 DFSTraverse 算法）即为逆向的拓扑序列。

第五章 查找

5.1 查找的基本概念

（1）数据元素（记录）

数据元素是由若干项、组合项构成的数据单位，是在某一问题中作为整体进行考虑和处理的基本单位。

（2）关键码

关键码是数据元素（记录）中某个项或组合项的值，用它可以标识一个数据元素（记录）。能唯一确定一个数据元素（记录）的关键码，称为主关键码；而不能唯一确定一个数据元素（记录）的关键码，称为次关键码。

（3）查找表

查找表是由具有同一类型（属性）的数据元素（记录）组成的集合。

（4）查找

根据给定的关键字值，在特定的列表中确定一个其关键字与给定值相同的数据元素，并返回该数据元素在列表中的位置。若找到相应的数据元素， 则称查找是成功的，否则称查找是失败的，此时应返回空地址及失败信息，并可根据要求插入这个不存在的数据元素。

（5）平均查找长度

为确定数据元素在列表中的位置，需和给定值进行比较的关键字个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于长度为 n 的列表， 查找成功时的平均查找长度为：

$$ASL = P_1C_1 + P_2C_2 + \cdots + P_nC_n = \sum_{i=1}^{n} P_iC_i$$

其中 P_i 为查找列表中第 i 个数据元素的概率，C_i 为找到列表中第 i 个数据元素时，已经进行过的关键字比较次数。由于查找算法的基本运算是关键字之间的比较操作，所以可用平均查找长度来衡量查找算法的性能。

（6）数据元素类型说明

在计算机中存储的表需要定义表的结构，并根据表的大小为表分配存储单元，可以用数组分配，即顺序存储结构；也可以用链式存储结构实现动态分配。

本章以后讨论中，涉及的关键码类型和数据元素类型统一说明如下：

```
typedef    struct {
    KeyType      key;          // 关键码字段，可以是整型、字符串型、构造类型等
    ……                         // 其它字段
} SElemType;
```

5.2 顺序查找法

顺序查找是一种最简单的查找方法。其基本思想是将查找表作为一个线性表，可以是顺序表，也可以是链表，依次用查找条件中给定的值与查找表中数据元素的关键字值进行比较，若某个记录的关键字值与给定值相等，则查找成功，返回该记录的存储位置，反之，若直到

最后一个记录，其关键字值与给定值均不相等，则查找失败，返回查找失败标志。

下面以顺序存储为例实现顺序查找：

```
typedef   struct {
    ElemType *elem; //数据元素存储空间基址，建表时按实际长度分配，0 号单元留空
    int       length;   // 表的长度
} SSTable;
int Search_Seq(SSTable ST, KeyType key) {
    ST.elem[0].key = key;                          // 设置“哨兵”
    for (i=ST.length; ST.elem[i].key!=key; --i);   // 从后往前找
    return i;                                      // 找不到时，i 为 0
}
```

就顺序查找法而言，对于 n 个数据元素的表，给定值 key 与表中第 i 个元素关键码相等，即定位第 i 个记录时，需进行 n-i+1 次关键码比较，即 C_i=n-i+1。则查找成功时，顺序查找的平均查找长度为：

$$ASL= \sum_{i=1}^{n} P_i \cdot (n-i+1)$$

设每个数据元素的查找概率相等，即 $P_i= \frac{1}{n}$ ，则等概率情况下有

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

查找不成功时，关键码的比较次数总是 n+1 次。

算法中的基本工作就是关键码的比较，因此，查找长度的量级就是查找算法的时间复杂度，其为 O(n)。

顺序查找缺点是当 n 很大时，平均查找长度较大，效率低；优点是对表中数据元素的存储没有要求，顺序存储或是链式存储均可。另外，对于线性链表，只能进行顺序查找。

5.3 折半查找法

折半查找要求查找表用顺序存储结构存放且各数据元素按关键字有序（升序或降序）排列，也就是说折半查找只适用于对有序顺序表进行查找。折半查找的基本思想是：首先以整个查找表作为查找范围，取中间元素作为比较对象，若给定值与中间元素的关键码相等，则查找成功；若给定值小于中间元素的关键码，则在中间元素的左半区继续查找；若给定值大于中间元素的关键码，则在中间元素的右半区继续查找。不断重复上述查找过程，直到查找成功，或所查找的区域无数据元素，查找失败。

```
int Search_Bin ( SSTable ST, KeyType key) {
    low = 1;   high = ST.length;                  // 置区间初值
    while (low <= high) {
        mid = (low + high) / 2;
        if （key == ST.elem[mid].key  )    return   mid; // 找到待查元素
        else   if (key < ST.elem[mid].key) )
                    high = mid - 1;               // 继续在前半区间进行查找
```

```
            else    low = mid + 1;              // 继续在后半区间进行查找
        }
        return 0;                               // 顺序表中不存在待查元素
    }
```

从折半查找过程看，以表的中点为比较对象，并以中点将表分割为两个子表，对定位到的子表继续这种操作。所以，对表中每个数据元素的查找过程，可用二叉树来描述，称这个描述查找过程的二叉树为判定树。

可以看到，查找表中任一元素的过程，即是判定树中从根到该元素结点路径上各结点关键码的比较次数，也即该元素结点在树中的层次数。对于 n 个结点的判定树，树高为 k，则有 $2^{k-1}-1<n\le 2^k-1$，即 $k-1<\log_2(n+1)\le k$，所以 $k=\lceil \log_2(n+1) \rceil$。因此，折半查找在查找成功时，所进行的关键码比较次数至多为 $\lceil \log_2(n+1) \rceil$。 。

接下来讨论折半查找的平均查找长度。为便于讨论，以树高为 k 的满二叉树($n=2^k-1$)为例。假设表中每个元素的查找是等概率的，即 $P_i=\frac{1}{n}$，则树的第 i 层有 2^{i-1} 个结点，因此，折半查找的平均查找长度为：

$$ASL=\sum_{i=1}^{n} P_i \cdot C_i$$

$$=\frac{1}{n}[1\times 2^0+2\times 2^1+\ldots+k\times 2^{k-1}]$$

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

所以，折半查找的时间效率为 $O(\log_2 n)$。

5.4 动态查找树表

5.4.1 二叉排序树

1.二叉排序树定义

二叉排序树（Binary Sort Tree）或者是一棵空树；或者是具有下列性质的二叉树：

⑴ 若左子树不空，则左子树上所有结点的值均小于根结点的值；若右子树不空，则右子树上所有结点的值均大于根结点的值。

⑵ 左右子树也都是二叉排序树。

通常，取二叉链表作为二叉排序树的存储结构。

2.二叉排序树查找过程

从其定义可见，二叉排序树的查找过程是一个递归过程，具体为：

① 若查找树为空，查找失败。

② 查找树非空，将给定值 key 与查找树的根结点关键码比较。

③ 若相等，查找成功，结束查找过程，否则，

a．当给 key 小于根结点关键码，查找将在以左孩子为根的子树上继续进行，转①

b．当给 key 大于根结点关键码，查找将在以右孩子为根的子树上继续进行，转①

●递归算法：

```
BiTree   SearchBST(BiTree b,   KeyType key) {
    if(!b) return NULL;                                  //查找失败
    else if(b-> data.key ==key) return b;                //查找成功
        else if(key＜b-> data.key)
                return SearchBST(b->lchild,key);     //在左子树中继续查找
            else return SearchBST(b->rchild,key);   //在右子树中继续查找
}
```

由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

3.二叉排序树插入操作和构造一棵二叉排序树

在二叉排序树中插入一个结点的过程：设待插入结点的关键码为 key，为将其插入，先要在二叉排序树中进行查找，若查找成功，按二叉排序树定义，待插入结点已存在，不用插入；查找不成功时，则插入之。因此，新插入结点一定是作为叶子结点添加上去的。构造一棵二叉排序树就是逐个插入结点的过程。

```
void InsertBST(BiTree &b,   KeyType key) {
    BiTree s;
    if (bt==NULL) { //  递归结束条件
        s=(BiTNode *)malloc(sizeof(BiTNode));
        s-> data.key =key;
        s->lchild=NULL;
        s->rchild=NULL;
        b=s;
    }
   else if(key＜b-> data.key) InsertBST(b->lchild,key);        //将 s 插入左子树
      else if(key＞b-> data.key)   InsertBST(b->rchild,key); //将 s 插入右子树
}
```

4.二叉排序树删除操作

和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。可分三种情况讨论：

（1）被删除的结点是叶子；

（2）被删除的结点只有左子树或者只有右子树；

（3）被删除的结点既有左子树，也有右子树。

```
int DeleteBST (BiTree &b,   KeyType key ) {
    if (!p)   return FALSE;                              //不存在关键字等于 key 的数据元素
    else{
        if (EQ(key, b->data.key)) Delete (b);       //  找到关键字等于 key 的数据元素
        else if (LT(key, b->data.key)) DeleteBST (b->lchild, key);
            else     DeleteBST (b->rchild, key);
        return TRUE;
```

```
        }
    }
```

其中删除操作过程如下描述：

```
void Delete ( BiTree &p ){
// 从二叉排序树中删除结点 p，并重接它的左或右子树
    if (!p->rchild) {            // 右子树空则只需重接它的左子树
        q = p;
        p = p->lchild;
        free(q);
    }
    else if (!p->lchild) {   // 只需重接它的右子树
            q = p;
            p = p->rchild;
            free(q);
        }
        else {                      // 左右子树均不空
            q = p;
            s = p->lchild;
            while (!s->rchild) {     //转左，然后向右到尽头
                q = s;
                s = s->rchild;
            }
            p->data = s->data;        // s 指向被删结点的前驱
            if (q != p )    q->rchild = s->lchild;       // 重接*q 的右子树
            else    q->lchild = s->lchild;               // 重接*q 的左子树
            free(s);
        }
}
```

对给定序列建立二叉排序树，若左右子树均匀分布，则其查找过程类似于有序表的折半查找。但若给定序列原本有序，则建立的二叉排序树就蜕化为单链表，其查找效率同顺序查找一样。由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。例如：

由关键字序列 1，2，3，4，5 构造而得的二叉排序树，ASL =（1+2+3+4+5）/ 5 = 3。

由关键字序列 3，1，2，5，4 构造而得的二叉排序树，ASL =（1+2+3+2+3）/ 5 = 2.2。

因此，对均匀的二叉排序树进行插入或删除结点后，应对其调整，使其依然保持均匀。

5.4.2 平衡二叉树

1.平衡二叉树定义

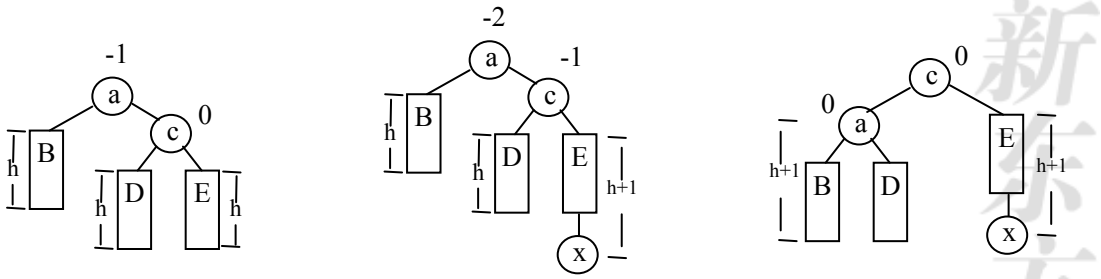
平衡二叉树或者是一棵空树，或者是具有下列性质的二叉排序树：它的左子树和右子树

都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过 1。

2.平衡二叉树的四种调整方式

在平衡二叉树上插入或删除结点后，可能使树失去平衡，因此，需要对失去平衡的树进行平衡化调整。设 a 结点为失去平衡的最小子树根结点，对该子树进行平衡化调整归纳起来有以下四种情况：

（1）左单调整



（a）插入前　　（b）插入后，调整前　　（c）调整后

图 RR 型

图(a)为插入前的子树。其中，B 为结点 a 的左子树，D、E 分别为结点 c 的左右子树，B、D、E 三棵子树的高均为 h。图(a)所示的子树是平衡二叉树。

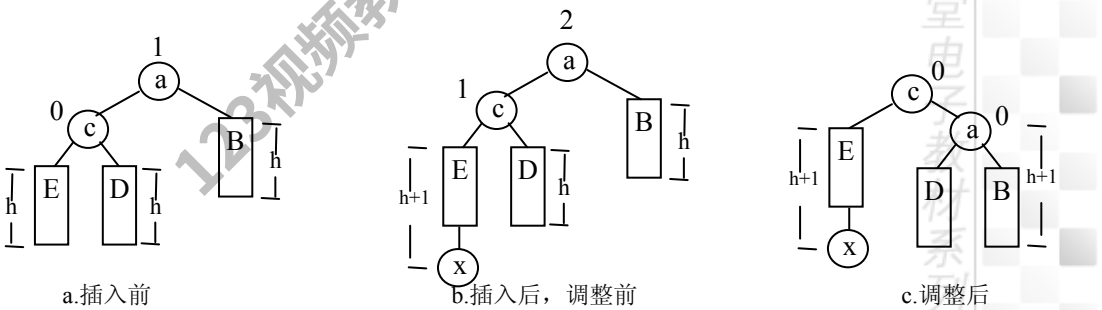
在图(a)所示的树上插入结点 x，如图(b)所示。结点 x 插入在结点 c 的右子树 E 上，导致结点 a 的平衡因子绝对值大于 1，以结点 a 为根的子树失去平衡。

调整策略：调整后的子树除了各结点的平衡因子绝对值不超过 1，还必须是二叉排序树。由于结点 c 的左子树 D 可作为结点 a 的右子树，将结点 a 为根的子树调整为左子树是 B，右子树是 D，再将结点 a 为根的子树调整为结点 c 的左子树，结点 c 为新的根结点，如图(c)。

平衡化调整操作判定：沿插入路径检查三个点 a、c、E，若它们处于“\”直线上的同一个方向，则要作左单旋转，即以结点 c 为轴逆时针旋转。

（2）右单调整

右单旋转与左单旋转类似，沿插入路径检查三个点 a、c、E，若它们处于“/”直线上的同一个方向，则要作右单旋转，即以结点 c 为轴顺时针旋转，如图所示。



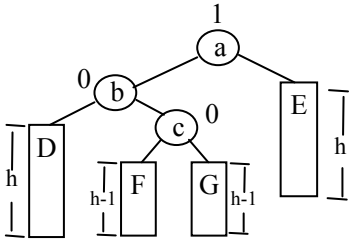
a.插入前　　b.插入后，调整前　　c.调整后

图 LL 型

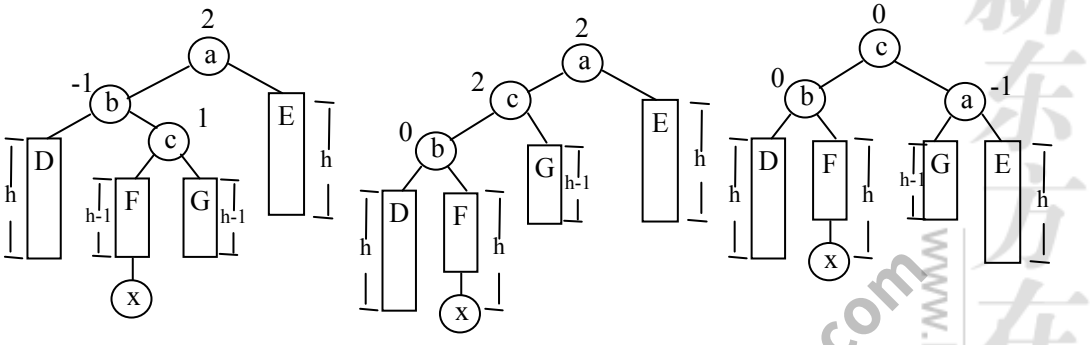
（3）先左后右双向旋转

第一图为插入前的子树，根结点 a 的左子树比右子树高度高 1，待插入结点 x 将插入到结点 b 的右子树上，并使结点 b 的右子树高度增 1，从而使结点 a 的平衡因子的绝对值大

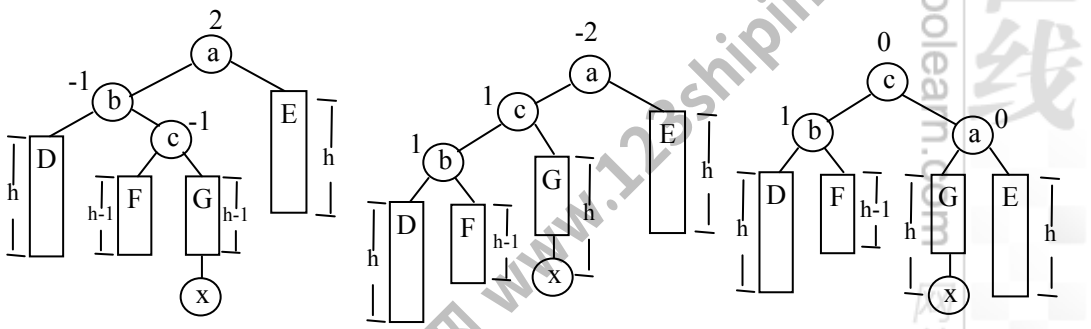
于 1，导致结点 a 为根的子树平衡被破坏，如图所示。



LR 型插入前



a.插入后，调整前　　b.先左旋转　　c.再右旋转



d.插入后，调整前　　e.先左旋转　　f.再右旋转

LR 型调整过程

沿插入路径检查三个点 a、b、c，若它们呈“<”字形，需要进行先左后右双向旋转：

①首先，对结点 b 为根的子树，以结点 c 为轴，向左逆时针旋转，结点 c 成为该子树的新根，如图(b、e)；

②由于旋转后，待插入结点 x 相当于插入到结点 b 为根的子树上，这样 a、c、b 三点处于“/”直线上的同一个方向，则要作右单旋转，即以结点 c 为轴顺时针旋转，如图(c、f)。

（4）先右后左双向旋转

先右后左双向旋转和先左后右双向旋转对称，在此不再赘述。

3.平衡树的查找分析

在平衡树上进行查找的过程和二叉排序树相同，因此，在查找过程中和给定值进行比较的关键码个数不超过树的深度。在平衡树上进行查找的时间复杂度为 O(logn)。

5.4.3B 树及其基本操作、B+树的基本概念

1.B-树及其查找

定义：一棵 m 阶的 B-树，或者为空树，或为满足下列特性的 m 叉树：

⑴树中每个结点至多有 m 棵子树；

⑵若根结点不是叶子结点，则至少有两棵子树；

⑶除根结点之外的所有非终端结点至少有$\lceil m/2 \rceil$ 棵子树；

⑷所有的非终端结点中包含以下信息数据：（n，A_0，K_1，A_1，K_2，…，K_n，A_n），

其中：K_i（i=1,2,…,n）为关键码，且 $K_i<K_{i+1}$，A_i 为指向子树根结点的指针(i=0,1,…,n)，且指针 A_{i-1} 所指子树中所有结点的关键码均小于 K_i (i=1,2,…,n)，A_n 所指子树中所有结点的关键码均大于 K_n，$\lceil m/2 \rceil -1 \le n \le m-1$ ，n 为关键码的个数。

⑸所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

B-树的查找类似二叉排序树的查找，所不同的是 B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。即在 B-树上的查找过程是一个顺指针查找结点和在结点中查找关键码交叉进行的过程。

B-树的查找是由两个基本操作交叉进行的过程，即①在 B-树上找结点；②在结点中找关键码。由于，通常 B-树是存储在外存上的，操作①就是通过指针在磁盘相对定位，将结点信息读入内存，之后，再对结点中的关键码有序表进行顺序查找或折半查找。因为，在磁盘上读取结点信息比在内存中进行关键码查找耗时多，所以，在磁盘上读取结点信息的次数，即 B-树的层次树是决定 B-树查找效率的首要因素。

那么，对含有 n 个关键码的 m 阶 B-树，最坏情况下达到多深呢？可按二叉平衡树进行类似分析。首先，讨论 m 阶 B-数各层上的最少结点数。

由 B-树定义：第一层至少有 1 个结点；第二层至少有 2 个结点；由于除根结点外的每个非终端结点至少有$\lceil m/2 \rceil$棵子树，则第三层至少有 2 ($\lceil m/2 \rceil$) 个结点；……；以此类推，第 k+1 层至少有 2 ($\lceil m/2 \rceil$)$^{k-1}$ 个结点。而 k+1 层的结点为叶子结点。若 m 阶 B-树有 n 个关键码，则叶子结点即查找不成功的结点为 n+1，由此有：

$$n+1 \ge 2*(\lceil m/2 \rceil)^{k-1}$$

即 $k \le \log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$。

这就是说，在含有 n 个关键码的 B-树上进行查找时，从根结点到关键码所在结点的路径上涉及的结点数不超过 $\log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$ 。

2. B+树

B+树是应文件系统所需而产生的一种 B-树的变形树。一棵 m 阶的 B+树和 m 阶的 B-树的差异在于：

⑴有 n 棵子树的结点中含有 n 个关键码；

⑵所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且

叶子结点本身依关键码的大小自小而大的顺序链接；

⑶所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码。

在 B+树上进行随机查找、插入和删除的过程基本上与 B-树类似。只是在查找时，若非终端结点上的关键码等于给定值，并不终止，而是继续向下直到叶子结点。因此，在 B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。若在结点内查找时，给定值$\leq K_i$，则应继续在 A_i 所指子树中进行查找。B+树查找的分析类似于 B-树。

5.5 散列表

5.5.1 散列表与散列方法

前面讨论的表示查找表的各种结构，有一个共同点：记录在表中的位置和它的关键字之间不存在一个确定的关系，因此，查找的过程为给定值依次和关键字集合中各个关键字进行比较，查找的效率取决于和给定值进行比较的关键字个数。因此，用这类方法表示的查找表，其平均查找长度都不为零，不同表示方法的差别仅在于：和给定值进行比较的关键字的顺序不同。理想的情况是依据关键码直接得到其对应的数据元素位置，即要求关键码与数据元素间存在一一对应关系，通过这个关系，能很快地由关键码得到对应的数据元素位置。

散列表与散列方法：选取某个函数，依该函数按关键码计算元素的存储位置，并按此存放；查找时，由同一个函数对给定值 key 计算地址，将 key 与地址单元中元素关键码进行比较，确定查找是否成功，这就是散列方法(杂凑法)；散列方法中使用的转换函数称为散列函数(杂凑函数)；按这个思想构造的表称为散列表(杂凑表)。

对于 n 个数据元素的集合，总能找到关键码与存放地址一一对应的函数。若最大关键为 m，可以分配 m 个数据元素存放单元，选取函数 f(key)=key 即可，但这样会造成存储空间的很大浪费，甚至不可能分配这么大的存储空间。通常关键码的集合比散列地址集合大得多，因而经过散列函数变换后，可能将不同的关键码映射到同一个散列地址上，这种现象称为冲突(Collision)，映射到同一散列地址上的关键码称为同义词。可以说，冲突不可能避免，只能尽可能减少。

所以，散列方法需要解决以下两个问题：

（1）构造好的散列函数

①所选函数尽可能简单，以便提高转换速度。

②所选函数对关键码计算出的地址，应在散列地址集中大致均匀分布，以减少空间浪费。

（2）制定解决冲突的方案。

5.5.2 常用的散列函数

对数字的关键字可有下列散列函数的构造方法，若是非数字关键字，则需先对其进行数字化处理。

1. 直接定址法

H(key) = key　或者　H(key) = a × key + b

即取关键码的某个线性函数值为散列地址，这类函数是一一对应函数，不会产生冲突。

此法仅适合于：地址集合的大小 == 关键字集合的大小

2. 数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成(k1, k2, …, kn)，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

此法仅适合于：能预先估计出全体关键字的每一位上各种数字出现的频度。

3．平方取中法

若关键字的每一位都有某些数字重复出现频度很高的现象，则先求关键字的平方值，以通过“平方”扩大差别，同时平方值的中间几位受到整个关键字中各位的影响。

此方法适合于：关键字中的每一位都有某些数字重复出现频度很高的现象。

4. 折叠法

若关键字的位数特别多，则可将其分割成几部分，然后取它们的叠加和为散列地址。可有：移位叠加和间界叠加两种处理方法。

（1）移位法：将各部分的最后一位对齐相加。

（2）间界叠加法：从一端向另一端沿各部分分界来回折叠后，最后一位对齐相加。

此方法适合于：关键字的数字位数特别多。

5. 除留余数法

H(key) = key MOD p　　p≤m (表长)

即取关键码除以 p 的余数作为散列地址。使用除留余数法，选取合适的 p 很重要，若散列表表长为 m，则要求 p≤m，且接近 m 或等于 m。p 一般选取质数，也可以是不包含小于 20 质因子的合数。

6. 随机数法

H(key) = Random(key)，其中，Random 为伪随机函数。

通常，此方法用于对长度不等的关键字构造散列函数。

实际造表时，采用何种构造散列函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是使产生冲突的可能性降到尽可能地小。

5.5.3 处理冲突的方法

处理冲突的实际含义是：为产生冲突的地址寻找下一个散列地址。

1. 开放定址法

所谓开放定址法，即是由关键码得到的散列地址一旦产生了冲突，也就是说，该地址已经存放了数据元素，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将数据元素存入。

找空散列地址方法很多，下面介绍三种：

（1）线性探测法

$H_i=(Hash(key)+d_i) \bmod m$　　($1 \le i < m$)

其中： Hash(key)为散列函数，m 为散列表长度，d_i 为增量序列 1，2，……，m-1，且 $d_i=i$。

这种方法的特点是：冲突发生时，顺序查看表中下一单元， 直到找出一个空单元或查遍

全表。

线性探测法可能使第 i 个散列地址的同义词存入第 i+1 个散列地址，这样本应存入第 i+1 个散列地址的元素变成了第 i+2 个散列地址的同义词，……，因此，可能出现很多元素在相邻的散列地址上“堆积”起来，大大降低了查找效率。为此，可采用二次探测法改善“堆积”问题。

（2）二次探测法

$H_i=(Hash(key)\pm d_i) \bmod m$

其中：Hash(key)为散列函数，m 为散列表长度，m 要求是某个 4k+3 的质数(k 是整数)，d_i 为增量序列 1^2，-1^2，2^2，-2^2，……，q^2，$-q^2$ 且 $q\le\frac{1}{2}m$。

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

（3）随机探测再散列

di 是一组伪随机数列，具体实现时，应建立一个伪随机数发生器，（如 i=(i+p) % m），并给定一个随机数做起点。

2. 拉链法

这种方法的基本思想是将所有散列地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在散列表的第 i 个单元中， 因而查找、插入和删除主要在同义词链中进行。 链地址法适用于经常进行插入和删除的情况。

5.5.4 散列表的查找

查找过程和造表过程一致。假设采用开放定址处理冲突，则查找过程为：

对于给定值 K, 计算散列地址 i = H(K)，

若 r[i] = NULL ，则查找不成功

若 r[i].key = K ，则查找成功

否则求下一地址 Hi，直至 r[Hi] = NULL (查找不成功)

或 r[Hi].key = K (查找成功)为止。

5.5.5 散列表的查找分析

散列表的查找过程基本上和造表过程相同。一些关键码可通过散列函数转换的地址直接找到，另一些关键码在散列函数得到的地址上产生了冲突，需要按处理冲突的方法进行查找。在介绍的处理冲突的方法中，产生冲突后的查找仍然是给定值与关键码进行比较的过程。所以，对散列表查找效率的量度，依然用平均查找长度来衡量。

查找过程中，关键码的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。因此，影响产生冲突多少的因素，也就是影响查找效率的因素。

影响产生冲突多少有以下三个因素：散列函数是否均匀；处理冲突的方法；散列表的装填因子。

分析这三个因素，尽管散列函数的“好坏”直接影响冲突产生的频度，但一般情况下，我们总认为所选的散列函数是“均匀的”，因此，可不考虑散列函数对平均查找长度的影响。就

线性探测法和二次探测法处理冲突的例子看，相同的关键码集合、同样的散列函数，但在数据元素查找等概率情况下，它们的平均查找长度却不同。

散列表的装填因子定义为：$\alpha=\frac{\text{填入表中的元素个数}}{\text{散列表的长度}}$

α 是散列表装满程度的标志因子。由于表长是定值，α 与“填入表中的元素个数”成正比，所以，α 越大，填入表中的元素较多，产生冲突的可能性就越大；α 越小，填入表中的元素较少，产生冲突的可能性就越小。

实际上，散列表的平均查找长度是装填因子 α 的函数，只是不同处理冲突的方法有不同的函数。

处理冲突的方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl}\approx\frac{1}{2}(1+\frac{1}{1-\alpha})$	$U_{nl}\approx\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
二次探测法 与双散列法	$S_{nr}\approx-\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr}\approx\frac{1}{1-\alpha}$
拉链法	$S_{nc}\approx1+\frac{\alpha}{2}$	$U_{nc}\approx\alpha+e^{-\alpha}$

散列方法存取速度快，也较节省空间，但由于存取是随机的，因此，不便于顺序查找。

习题

1、利用逐点插入建立序列（50，72，43，85，75，20，35，45，65，30）对应的二叉排序树以后，要查找元素 30 要进行（　）次元素间的比较。

A．4　　B. 5　　C．6　　D. 7

2、在平衡二叉树中，（　）。

A．任意结点的左、右子树结点数目相同

B．任意结点的左、右子树高度相同

C．任意结点的左右子树高度之差的绝对值不大于 1

D．不存在度为 1 的结点

3、由元素序列（27，16，75，38，51）构造平衡二叉树，则首次出现的最小不平衡子树的根（即离插入结点最近且平衡因子的绝对值为 2 的结点）为（　）

A．27　　B．38　　C．51　　D. 75

4、在常用的描述二叉排序树的存储结构中，关键字值最大的结点（　）

A．左指针一定为空　B. 右指针一定为空

C．左右指针均为空　D. 左右指针均不为空

5、设顺序存储的某线性表共有 123 个元素，按分块查找的要求等分为 3 块。若对索引表采用顺序查找方法来确定子块，且在确定的子块中也采用顺序查找方法，则在等概率的情况下，分块查找成功的平均查找长度为（　）。

A．21 B. 23 C. 41 D. 62

分析：分块查找成功的平均查找长度为 ASL=(s^2+s+n)/2s。在本题中，n=123，s=123/3=41，故平均查找长度为 23。

6、从理论上讲，将数据以（ ）结构存放，则查找一个数据所用时间不依赖于数据个数 n。

A．二叉查找树 B．链表

C．二叉树 D．散列表

7、设有一个含 200 个表项的散列表，用线性探查法解决冲突，按关键码查询时找到一个表项的平均探查次数不超过 1.5，则散列表项应能够至少容纳（ ）个表项。

A．400 B．526 C．624 D．676

分析：设搜索成功的平均搜索长度为 Snl={1+1/（1－α）}/2，其中α为装填因子。计算可得。

8、将两个长度为 n 的递增有序表归并成一个长度为 2n 的递增有序表，最少需要进行关键字比较（ ）次。

A．1 B．n-1 C．n D．2n

9、已知一组关键字为（26，36，41，38，44，15，68，12，6，51，25），用链地址法解决冲突。假设装填因子α=0.75，散列函数的形式为 H(K) = K MOD P，回答下列问题：

（1）构造散列函数。

（2）画出散列表。

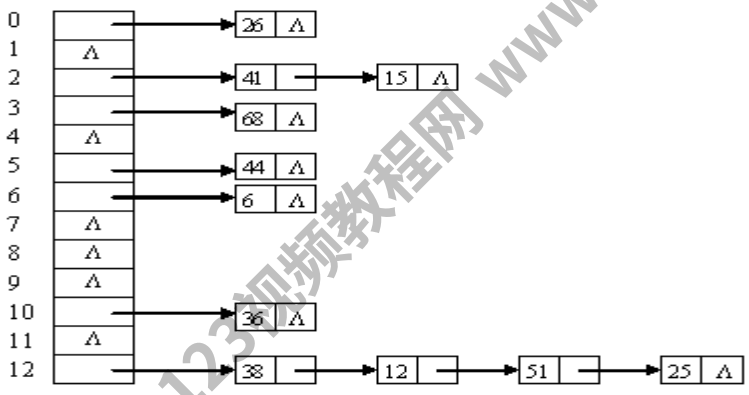
（3）计算等概率情况下查找成功的平均查找长度。

（4）计算等概率情况下查找不成功的平均查找长度。

解答：由α=0.75，得表长 m=11/0.75=15.

（1）散列函数 H(K) = K mod13

（2）散列表



（3）等概率情况下查找成功的平均查找长度：

ASL=（1×7+2×2+3×1+4×1）/11=18/11

（4）等概率情况下查找不成功的平均查找长度：

ASL=（1×5+2×1+4×1）/13=11/13

第六章 排序

6.1 排序的基本概念

排序(Sorting)是计算机程序设计中的一种重要操作，其功能是对一个数据元素集合或序列重新排列成一个按数据元素某个项值有序的序列。

有 n 个记录的序列{R_1，R_2，…，R_n}，其相应关键字的序列是{K_1，K_2，…，K_n}，相应的下标序列为 1，2，…，n。通过排序，要求找出当前下标序列 1，2，…， n 的一种排列 p1，p2， …，pn，使得相应关键字满足如下的非递减（或非递增）关系，即：$K_{p1}\leq K_{p2}\leq\ldots\leq K_{pn}$，这样就得到一个按关键字有序的记录序列{$R_{p1}$，$R_{p2}$，…，$R_{pn}$}。

作为排序依据的数据项称为“排序码”，也即数据元素的关键码。若关键码是主关键码，则对于任意待排序序列，经排序后得到的结果是唯一的；若关键码是次关键码，排序结果可能不唯一。

实现排序的基本操作有两个：（1）“比较”序列中两个关键字的大小；（2）“移动”记录。

若对任意的数据元素序列，使用某个排序方法，对它按关键码进行排序：若相同关键码元素间的位置关系，排序前与排序后保持一致，称此排序方法是稳定的；而不能保持一致的排序方法则称为不稳定的。

6.2 插入排序

6.2.1 直接插入排序

直接插入排序是最简单的插入类排序。仅有一个记录的表总是有序的，因此，对 n 个记录的表，可从第二个记录开始直到第 n 个记录，逐个向有序表中进行插入操作，从而得到 n 个记录按关键码有序的表。它是利用顺序查找实现“在 R[1..i-1]中查找 R[i]的插入位置”的插入排序。

注意直接插入排序算法的三个要点：

（1）从 R[i-1]起向前进行顺序查找，监视哨设置在 R[0]；

```
R[0] = R[i];                          // 设置“哨兵”
for (j=i-1; R[0].key<R[j].key; --j);  // 从后往前找
return j+1;                           // 返回 R[i]的插入位置为 j+1
```

（2）对于在查找过程中找到的那些关键字不小于 R[i].key 的记录，可以在查找的同时实现向后移动；

```
for (j=i-1; R[0].key<R[j].key; --j);
    R[j+1] = R[j]
```

（3）i = 2，3，…, n, 实现整个序列的排序。

【算法如下】

```
void InsertionSort ( Elem R[ ],   int n) {
   for ( i=2; i<=n; ++i ) {
      R[0] = R[i];                    // 复制为监视哨
```

```
        for ( j=i-1; R[0].key < R[j].key;   --j )
            R[j+1] = R[j];                  // 记录后移
        R[j+1] = R[0];                      // 插入到正确位置
    }
}
```

【性能分析】

（1）空间效率：仅用了一个辅助单元，空间复杂度为 O(1)。

（2）时间效率：向有序表中逐个插入记录的操作，进行了 n-1 趟，每趟操作分为比较关键码和移动记录，而比较的次数和移动记录的次数取决于待排序列按关键码的初始排列。

直接插入排序的最好情况的时间复杂度为 O(n)，平均时间复杂度为 $O(n^2)$。

（3）稳定性：直接插入排序是一个稳定的排序方法。

6.2.2 折半插入排序

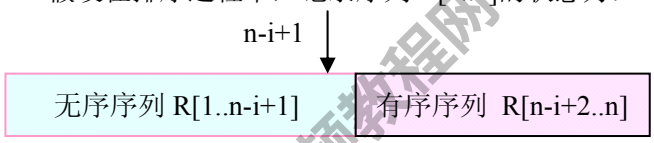
直接插入排序的基本操作是向有序表中插入一个记录，插入位置的确定通过对有序表中记录按关键码逐个比较得到的。平均情况下总比较次数约为 $n^2/4$。既然是在有序表中确定插入位置，可以不断二分有序表来确定插入位置，即一次比较，通过待插入记录与有序表居中的记录按关键码比较，将有序表一分为二，下次比较在其中一个有序子表中进行，将子表又一分为二。这样继续下去，直到要比较的子表中只有一个记录时，比较一次便确定了插入位置。

折半插入排序是利用折半查找实现“在 R[1..i-1]中查找 R[i]的插入位置”。

6.3 冒泡排序

交换排序主要是通过两两比较待排记录的关键码，若发生与排序要求相逆，则交换之。冒泡排序是最简单的一种交换排序。

假设在排序过程中，记录序列 R[1..n]的状态为：



则第 i 趟起泡插入排序的基本思想为：借助对无序序列中的记录进行“交换”的操作，将无序序列中关键字最大的记录“交换”到 R[n-i+1]的位置上。

【算法如下】

```
void BubbleSort(Elem R[], int n){
    i = n;                          // i 指示无序序列中最后一个记录的位置
    while (i >1) {
        lastExchangeIndex = 1;
        for (j = 1; j < i; j++)
        if   (A[j+1] < A[j]) {
            Swap(A[j],A[j+1]);
            lastExchangeIndex = j;
```

```
        }
    i = lastExchangeIndex;
    }
}
```

【性能分析】

(1)空间效率：仅用了一个辅助单元，空间复杂度为 O(1)。

(2)时间效率：最好情况的时间复杂度为 O(n)，平均时间复杂度为 $O(n^2)$。

(3)稳定性：冒泡排序法是一种稳定的排序方法

$$\text{总比较次数} = \sum_{j=2}^{n}(j-1) = \frac{1}{2}n(n-1)$$

6.4 简单选择排序

简单选择排序是最简单的一种选择类的排序方法。假设排序过程中，待排记录序列的状态为：

无序序列 R[i..n]

并且有序序列中所有记录的关键字均小于无序序列中记录的关键字，则第 i 趟简单选择排序是，从无序序列 R[i..n]的 n-i+1 记录中选出关键字最小的记录加入有序序列。

操作方法：第一趟，从 n 个记录中找出关键码最小的记录与第 1 个记录交换；第二趟，从第二个记录开始的 n-1 个记录中再选出关键码最小的记录与第 2 个记录交换；如此，第 i 趟，则从第 i 个记录开始的 n-i+1 个记录中选出关键码最小的记录与第 i 个记录交换，直到整个序列按关键码有序。

【算法如下】

```
void SelectSort (Elem R[], int n ) {   // 对记录序列 R[1..n]作简单选择排序。
    for (i=1; i<n; ++i) {              //选择第 i 小的记录，并交换到位
    k = SelectMinKey(R, i);              // 在 R[i..n]中选择 key 最小的记录
    if (i!=j) R[i]←→R[k];              // 与第 i 个记录交换
    }
}
```

【性能分析】

(1)空间效率：仅用了一个辅助单元，空间复杂度为 O(1)。

(2)时间效率：简单选择排序的最好和平均时间复杂度均为 $O(n^2)$。

(3)稳定性：不同教材对简单选择排序的稳定性有争议，一般认为，若是从前往后比较来选择第 i 小的记录则该算法是稳定的，若是从后往前比较来选择第 i 小的记录则该算法是不稳定的。

6.5 希尔排序

希尔排序又称缩小增量排序，较直接插入排序和折半插入排序方法有较大的改进。直接

插入排序算法简单，在 n 值较小时，效率比较高，在 n 值很大时，若序列按关键码基本有序，效率依然较高，其时间效率可提高到 O(n)。希尔排序即是从这两点出发，给出插入排序的改进方法。希尔排序的基本思想是：先将待排序记录序列分割成若干个“较稀疏的”子序列，分别进行直接插入排序。经过上述粗略调整， 整个序列中的记录已经基本有序，最后再对全部记录进行一次直接插入排序。具体实现时，首先选定两个记录间的距离 d_1，在整个待排序记录序列中将所有间隔为 d_1 的记录分成一组，进行组内直接插入排序，然后再取两个记录间的距离 $d_2<d_1$，在整个待排序记录序列中，将所有间隔为 d_2 的记录分成一组，进行组内直接插入排序，直至选定两个记录间的距离 $d_t=1$ 为止，此时只有一个子序列，即整个待排序记录序列。

【性能分析】

（1）空间效率：仅用了一个辅助单元，空间复杂度为 O(1)。

（2）时间效率：希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于步长因子序列的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的步长因子序列的方法。步长因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：步长因子中除 1 外没有公因子，且最后一个步长因子必须为 1。

(3)稳定性：希尔排序方法是一个不稳定的排序方法。

6.6 快速排序

快速排序是通过比较关键码、交换记录，以某个记录为界(该记录称为支点)，将待排序列分成两部分。其中，一部分所有记录的关键码大于等于支点记录的关键码，另一部分所有记录的关键码小于支点记录的关键码。我们将待排序列按关键码以支点记录分成两部分的过程，称为一次划分。对各部分不断划分，直到整个序列按关键码有序。

【算法如下】

一趟快速排序算法：

```
int Partition1 (Elem R[], int low, int high) {
    pivotkey = R[low].key;          // 用子表的第一个记录作枢轴记录
    while (low<high) {              // 从表的两端交替地向中间扫描
        while (low<high && R[high].key>=pivotkey)
            --high;
        R[low]←→R[high];            // 将比枢轴记录小的记录交换到低端
        while (low<high && R[low].key<=pivotkey)
            ++low;
        R[low]←→R[high];            // 将比枢轴记录大的记录交换到高端
    }
    return low;                     // 返回枢轴所在位置
}
```

容易看出，调整过程中的枢轴位置并不重要，因此，为了减少记录的移动次数，应先将枢轴记录“移出”，待求得枢轴记录应在的位置之后(此时 low=high),再将枢轴记录到位。

将上述“一次划分”的算法改写如下：

```
int Partition2 (Elem R[], int low, int high) {
```

```
        R[0] = R[low];                  // 用子表的第一个记录作枢轴记录
        pivotkey = R[low].key;        // 枢轴记录关键字
        while (low<high) {              // 从表的两端交替地向中间扫描
            while (low<high&& R[high].key>=pivotkey)
                --high;
            R[low] = R[high];         // 将比枢轴记录小的记录移到低端
            while (low<high && R[low].key<=pivotkey)
                ++low;
            R[high] = R[low];         // 将比枢轴记录大的记录移到高端
        }
        R[low] = R[0];                  // 枢轴记录到位
        return low;                     // 返回枢轴位置
    }
    递归形式的快速排序算法：
    void QSort (Elem R[], int low, int high) {
    // 对记录序列 R[low..high]进行快速排序
        if (low < high-1) {                                 // 长度大于 1
            pivotloc = Partition(L, low, high); // 将 L.r[low..high]一分为二
            QSort(L, low, pivotloc-1); // 对低子表递归排序，pivotloc 是枢轴位置
            QSort(L, pivotloc+1, high);             // 对高子表递归排序
        }
    }
    void QuickSort(Elem R[],    int n) {   // 对记录序列进行快速排序
        QSort(R, 1, n);
    }
```

【性能分析】

(1)空间效率：快速排序是递归的，每层递归调用时的指针和参数均要用栈来存放，递归调用层次数与上述二叉树的深度一致。因而，存储开销在理想情况下为 $O(\log_2 n)$，即树的高度；在最坏情况下，即二叉树是一个单链，为 O(n)。

(2)时间效率：在 n 个记录的待排序列中，一次划分需要约 n 次关键码比较，时效为 O(n)，若设 T(n)为对 n 个记录的待排序列进行快速排序所需时间。

理想情况下：每次划分，正好将分成两个等长的子序列，则

$T(n) \le cn+2T(n/2)$　　c 是一个常数

$\le cn+2(cn/2+2T(n/4))=2cn+4T(n/4)$

$\le 2cn+4(cn/4+T(n/8))=3cn+8T(n/8)$

.....

$\le cn\log_2 n+nT(1)=O(n\log_2 n)$

最坏情况下：即每次划分，只得到一个子序列，时效为 $O(n^2)$。

快速排序是通常被认为在同数量级 $O(n\log_2 n)$的排序方法中平均性能最好的。但若初始序

列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取支点记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。

(3)快速排序是一个不稳定的排序方法。

6.7 堆排序

堆排序的特点是，在以后各趟的“选择”中利用在第一趟选择中已经得到的关键字比较的结果。

堆的定义: 堆是满足下列性质的数列{r1, r2, …，rn}：

$$\begin{cases} r_i \le r_{2i} \\ r_i \le r_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} r_i \ge r_{2i} \\ r_i \ge r_{2i+1} \end{cases}$$

若将此数列看成是一棵完全二叉树，则堆或是空树或是满足下列特性的完全二叉树：其左、右子树分别是堆，并且当左/右子树不空时，根结点的值小于(或大于)左/右子树根结点的值。

由此，若上述数列是堆，则 r_1 必是数列中的最小值或最大值，分别称作小顶堆或大顶堆。

堆排序即是利用堆的特性对记录序列进行排序的一种排序方法。具体作法是：设有 n 个元素，将其按关键码排序。首先将这 n 个元素按关键码建成堆，将堆顶元素输出，得到 n 个元素中关键码最小(或最大)的元素。然后，再对剩下的 n-1 个元素建成堆，输出堆顶元素，得到 n 个元素中关键码次小(或次大)的元素。如此反复，便得到一个按关键码有序的序列。称这个过程为堆排序。

因此，实现堆排序需解决两个问题：

（1）如何将 n 个元素的序列按关键码建成堆。

建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。n 个结点的完全二叉树，则最后一个结点是第 $\left\lfloor \frac{n}{2} \right\rfloor$ 个结点的孩子。对第 $\left\lfloor \frac{n}{2} \right\rfloor$ 个结点为根的子树筛选，使该子树成为堆，之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

（2）输出堆顶元素后，怎样调整剩余 n-1 个元素，使其按关键码成为一个新堆。

调整方法：设有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。将堆底元素送入堆顶，堆被破坏，其原因仅是根结点不满足堆的性质。将根结点与左、右孩子中较小(或小大)的进行交换。若与左子女交换，则左子树堆被破坏，且仅左子树的根结点不满足堆的性质；若与右子女交换，则右子树堆被破坏，且仅右子树的根结点不满足堆的性质。继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。称这个自根结点到叶子结点的调整过程为筛选。

【算法如下】

堆排序的算法如下所示：

```
void HeapSort ( Elem R[], int n ) { //  对记录序列 R[1..n]进行堆排序。
    for ( i=n/2; i>0; --i )                 //  把 R[1..n]建成大顶堆
        HeapAdjust ( R, i, n );
    for ( i=n; i>1; --i ) {
```

```
            R[1]←→R[i];
            //将堆顶记录和当前未经排序子序列,R[1..i]中最后一个记录相互交换
            HeapAdjust(R, 1, i-1);          // 将 R[1..i-1] 重新调整为大顶堆
        }
    }
```

其中筛选的算法如下所示。为将 R[s..m]调整为“大顶堆”，算法中“筛选”应沿关键字较大的孩子结点向下进行。

```
    void HeapAdjust (Elem R[], int s, int m) {
    /* 已知 R[s..m]中记录的关键字除 R[s].key 之外均满足堆的定义，本函数调整 R[s] 的关键字，使 R[s..m]成为一个大顶堆（对其中记录的关键字而言）*/
        rc = 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 && R[j].key<R[j+1].key )   ++j;   // j 为 key 较大的记录的下标
            if ( rc.key >= R[j].key )   break;          // rc 应插入在位置 s 上
            R[s] = R[j];   s = j;
        }
        R[s] = rc;                                      // 插入
    }
```

【性能分析】

（1）空间效率：仅用了一个辅助单元，空间复杂度为 O(1)。

（2）时间效率：

①对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为 2(k-1);

②对 n 个关键字，建成深度为 h(=⌊log2n⌋+1)的堆，所需进行的关键字比较的次数至多为 4n;

③调整“堆顶”n-1 次，总共进行的关键字比较的次数不超过

2(⌊log2(n-1)⌋+⌊log2(n-2)⌋+ …+log22)<2n(⌊log2n⌋)

因此，堆排序的平均和最坏时间复杂度均为 O(nlogn)。

(3)堆排序是一个不稳定的排序方法。

6.8 二路归并排序

【算法思想】

归并排序的基本思想是：将两个或两个以上的有序子序列“归并”为一个有序序列。

在内部排序中，通常采用的是 2-路归并排序。即：将两个位置相邻的有序子序列，

有序子序列 R[*l*..*m*]	有序子序列 R[*m*+1..*n*]

归并为一个有序序列。

有 序 序 列 R[*l*..*n*]

【算法如下】

```
void Merge (Elem SR[], Elem 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)   { // 将 SR 中记录由小到大地并入 TR
          if (SR[i].key<=SR[j].key)   TR[k] = SR[i++];
          else TR[k] = SR[j++];
    }
    if (i<=m) TR[k..n] = SR[i..m];                    // 将剩余的 SR[i..m]复制到 TR
    if (j<=n) TR[k..n] = SR[j..n];                   // 将剩余的 SR[j..n]复制到 TR
}
```

归并排序的算法可以有两种形式：递归的和递推的，它是由两种不同的程序设计思想得出的。

在此，只讨论递归形式的算法，这是一种自顶向下的分析方法：如果记录无序序列 R[s..t]的两部分 R[s..⌊(s+t)/2⌋]和 R[⌊(s+t)/2+1..t⌋]分别按关键字有序，则利用上述归并算法很容易将它们归并成整个记录序列是一个有序序列，由此，应该先分别对这两部分进行 2-路归并排序。

```
void Msort ( Elem SR[], Elem TR1[], int s, int t ) {/
     if (s==t)   TR1[s] = SR[s];
     else {
          m = (s+t)/2;       // 将 SR[s..t]平分为 SR[s..m]和 SR[m+1..t]
          Msort (SR, TR2, s, m);
                          // 递归地将 SR[s..m]归并为有序的 TR2[s..m]
          Msort (SR, TR2, m+1, t);
                         //递归地 SR[m+1..t]归并为有序的 TR2[m+1..t]
          Merge (TR2, TR1, s, m, t);
                         // 将 TR2[s..m]和 TR2[m+1..t]归并到 TR1[s..t]
     }
}
void MergeSort (Elem R[]) {  // 对记录序列 R[1..n]作 2-路归并排序。
     MSort(R, R, 1, n);
}
```

【性能分析】

(1)空间效率：需要一个与表等长的辅助元素数组空间，所以空间复杂度为 O(n)。

(2)时间效率：对 n 个元素的表，将这 n 个元素看作叶结点，若将两两归并生成的子表看作它们的父结点，则归并过程对应由叶向根生成一棵二叉树的过程。所以归并趟数约等于二叉树的高度-1，即 $\log_2 n$，每趟归并需移动记录 n 次，故时间复杂度为 $O(n\log_2 n)$。

(3)稳定性：归并排序是一个稳定的排序方法。

6.9 基数排序

基数排序是一种借助于多关键码排序的思想，是将单关键码按基数分成“多关键码”进行排序的方法。

对于数字型或字符型的单关键字，可以看成是由多个数位或多个字符构成的多关键字，此时可以采用这种“分配-收集”的办法进行排序，称作基数排序法。其好处是不需要进行关键字间的比较。

例如：对下列这组关键字{278, 109, 063, 930, 589, 184, 505, 269, 008，083 },

首先按其“个位数”取值分别为 0, 1, …，9“分配”成 10 组，之后按从 0 至 9 的顺序将它们“收集”在一起;然后按其“十位数” 取值分别为 0, 1, …，9“分配”成 10 组，之后再按从 0 至 9 的顺序将它们“收集”在一起;最后按其“百位数”重复一遍上述操作，便可得到这组关键字的有序序列。

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

(1)待排序记录以指针相链，构成一个链表；

(2)“分配”时，按当前“关键字位”所取值，将记录分配到不同的“链队列”中，每个队列中记录的“关键字位”相同；

(3)“收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

(4)对每个关键字位均重复(2)和(3)两步。

【性能分析】

(1)空间效率：链式基数排序空间复杂度为 O(rd)。

(2)时间效率：基数排序的时间复杂度为 O(d(n+rd))。

其中，分配为 O(n); 收集为 O(rd)(rd 为“基”), d 为“分配-收集”的趟数。

(3）稳定性：基数排序是一个稳定的排序方法。

6.10 各种内部排序算法的比较

1、时间性能

（1）按平均的时间性能来分，有三类排序方法：

时间复杂度为 O(nlogn)的方法有：快速排序、堆排序和归并排序，其中以快速排序为最好；

时间复杂度为 O(n^2)的有：直接插入排序、起泡排序和简单选择排序，其中以直接插入为最好，特别是对那些对关键字近似有序的记录序列尤为如此；

时间复杂度为 O(n)的排序方法只有，基数排序。

（2）当待排记录序列按关键字顺序有序时，直接插入排序和起泡排序能达到 O(n)的时间复杂度;而对于快速排序而言，这是最不好的情况，此时的时间性能蜕化为 O(n^2)，因此是应该尽量避免的情况。

（3）简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

2、空间性能

（1）所有的简单排序方法(包括：直接插入、起泡和简单选择)和堆排序的空间复杂度为 O(1)；

（2）快速排序为 O(log*n*)，为栈所需的辅助空间;

（3）归并排序所需辅助空间最多，其空间复杂度为 O(*n*);

（4）链式基数排序需附设队列首尾指针，则空间复杂度为 O(*rd*)。

3、排序方法的稳定性能

稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和经过排序之后，没有改变。

当对多关键字的记录序列进行 LSD 方法排序时，必须采用稳定的排序方法。

对于不稳定的排序方法，只要能举出一个实例说明即可。

快速排序和堆排序是不稳定的排序方法。

4、各种排序方法的综合比较和选择

（1）各种排序方法的比较如表。

算法	时间复杂度			空间复杂度	稳定性	复杂性
	最好	平均	最坏			
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	是	简单
冒泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	是	简单
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	否	简单
希尔	-	O(nlogn) ~ $O(n^2)$	O(nlogn) ~ $O(n^2)$	O(1)	否	复杂
快速	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	否	复杂
堆	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	否	复杂
归并	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	是	复杂
基数	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	是	复杂

（2）选择

选择排序方法时需要考虑的因素有：待排序的记录个数 n；记录本身的大小；关键字的分布情况；对排序稳定性的要求；语言工具的条件；辅助空间的大小。

依据这些元素，可以得到以下几点结论：

①待排序的记录数目 n 较小时，则采用插入排序和简单选择排序；

②若待排序记录按关键字基本有序，则宜采用直接插入排序和冒泡排序；

③当 n 很大且关键字的位数较少时，采用链式基数排序较好；

④若 n 较大，则应采用时间复杂度为 O(nlogn)的排序方法：快速排序、堆排序或归并排序。

习题

1.下列排序算法中，时间复杂度为 $O(n\log_2 n)$且占用额外空间最少的是（）

A. 堆排序

B. 起泡排序

C. 快速排序

D. 希尔排序

2. 下列序列中，满足堆定义的是( )

A.（100，86，48，73，35，39，42，57，66，21）

B.（12，70，33，65，24，56，48，92，86，33）

C.（103，97，56，38，66，23，42，12，30，52，6，26）
D.（5，56，20，23，40，38，29，61，36，76，28，100）
3.（ ）在最好情况下的算法时间复杂度为 O(n)。
A．插入排序 B. 归并排序
C．快速排序 D. 堆排序
4. 在最好和最坏情况下的时间复杂度均为 O(nlogn)且稳定的排序方法是（ ）。
A．基数排序 B. 归并排序
C．快速排序 D. 堆排序
5.若要求尽可能快地对序列进行稳定的排序，则应选（ ）
A．快速排序 B．归并排序
C．冒泡排序 D. 堆排序
6.下列排序算法中( )不能保证每趟排序至少能将一个元素放到其最终的位置上。
A. 快速排序 B. shell 排序
C. 堆排序 D. 冒泡排序
7.设有 10000 个无序元素，希望用最快的速度挑选出其中前 10 个最大的元素，用（ ）排序方法最好？
A. 堆排序 B. 快速排序
C. 基数排序 D. 希尔排序
8.若一组纪录的关键码（46，79，56，38，40，84），利用快速排序的方法，以第一个纪录为基准得到的一次划分结果为（ ）
A．38，40，46，56，79，84
B．40，38，46，79，56，84
C．40，38，46，56，79，84
D．40，38，46，84，56，79
9、请编写一个双向起泡的排序算法，即每一趟通过每两个相邻的关键字进行比较，产生最小和最大的元素。
解：

```
void Bubble_Sort2(int a[],int n)//相邻两趟向相反方向起泡的冒泡排序算法
{
    low=0; high=n-1;                    //冒泡的上下界
     change=1;
  while(low<high&&change)
    {
        change=0;                       //设不发生交换
        for(i=low;i<high;i++)           //从上向下起泡
                if(a[i]>a[i+1])
            {a[i]<->a[i+1];
             change=1;}
        high--;                         //修改上界
```

```
        for(i=high;i>low;i--)          //从下向上起泡
            if(a[i]<a[i-1])
          {a[i]<->a[i-1];
           change=1;}
        low++;                         //修改下界
    } //while
} //BubbleSort2
```