另一种形式

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=0; i<n-1; i++ )
      for ( j=0; j<n-i-1; j++ )
         if ( a[j]>a[j+1] )
            a[j]<—>a[j+1];
}
```

或

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=1; i<n; i++ )
      for ( j=0; j<n-i; j++ )
         if( a[j]>a[j+1] )
            a[j]<—>a[j+1];
}
```

或

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=0; i<n-1; i++ ) {
      change = fasle;
      for ( j=0; j<n-i-1; j++ )
         if ( a[j]>a[j+1] ) {
            a[j]<—>a[j+1];
            change = true;
         }
      if ( !change ) break;
   }
}
```

说明：

a) 考试中要求写算法时，可用类 C，也可用 C 程序。

b) 尽量书写算法说明，言简意赅。

c) 技巧：用“边界值验证法”检查下标越界错误。

如上第一个： 第二个循环条件若写作 j<n-i，则当 i=0 时 a[j+1]会越界。

d) 时间复杂度为 $O(n^2)$，第 3 个在最好情况下（待排记录有序），时间复杂度为 O(n)。

三、习题

1.1 编写冒泡排序算法，使结果从大到小排列。

1.2 计算下面语句段中指定语句的频度：

1)
```
for ( i=1; i<=n; i++ )
    for ( j=i; j<=n; j++ )
        x++;------------------------ // @
```

2)
```
i = 1;
```

```
while ( i<=n )
   i = i*2; ------------------------- // @
```

第2章 线性表

一、基础知识和算法

1. 线性表及其特点

线性表是 n 个数据元素的有限序列。
线性结构的特点： ①“第一个” ②“最后一个” ③**前驱** ④**后继**。[6]

2. 顺序表——线性表的顺序存储结构

（1）特点

a) 逻辑上相邻的元素在物理位置上相邻。
b) 随机访问。

（2）类型定义

简而言之，“数组+长度”[7]。

```
const int MAXSIZE = 线性表最大长度;
typedef struct{
    DataType elem[MAXSIZE];
    int length;
} SqList;
```

注：a) SqList 为类型名，可换用其他写法。
b) DataType 是数据元素的类型，根据需要确定。
c) MAXSIZE 根据需要确定。如

```
const int MAXSIZE=64;
```

d) 课本上的 SqList 类型可在需要时增加存储空间，在上面这种定义下不可以。(这样做避免了动态内存分配，明显减少了算法的复杂程度，容易理解。而且，原来 Pascal 版本的《数据结构》（严蔚敏）就是这样做的。)

[6] 这里太简炼了，只是为了便于记忆。

[7] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。凡此情形，都加引号以示提醒。

e) 课本上的 SqList 类型定义中 listsize 表示已经分配的空间大小（容纳数据元素的个数）。当插入元素而遇到 L.length==L.listsize 时，用 realloc (L.elem, L.listsize+增量)重新分配内存，而 realloc()函数在必要的时候自动复制原来的元素到新分配的空间中。

（3）基本形态

1°. 顺序表空

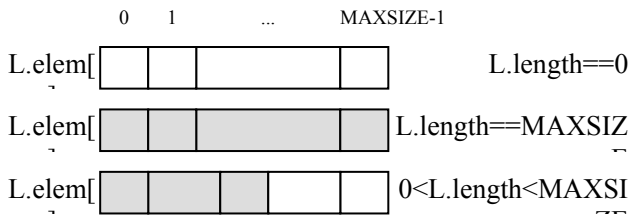
条件　L.length==0
不允许删除操作

2°. 顺序表满

条件　L.length==MAXSIZE
不允许插入操作

3°. 不空也不满

可以插入，删除



（4）基本算法——遍历

1°. 顺序访问所有元素

```
for ( i=0; i<L.length; i++ )
      visit ( L.elem[i] );
```

2°. 查找元素 x

```
for ( i=0; i<L.length; i++ )
   if ( L.elem[i]==x )    break;
if ( i<L.length )
   找到;
else
   未找到;
```

（5）插入算法 ListInsert(&L,i,x)

1°. 前提：表不满

2°. 合理的插入范围：1≤i≤L.length+1

注：位序 i 在 C/C++中对应于下标 i-1。

3°. 步骤

第 i 至最后所有元素后移一个元素
在第 i 个位置插入元素 x
表长增 1

4°. 算法

```
bool[8] ListInsert ( SqList& L, int i, DataType x )
{
  if ( L.length==MAXSIZE || i<1 || i>L.length+1 )   return false; // 失败
  // 元素后移
  for ( j=L.length-1; j>=i-1; j-- )              // 这里 j 为下标，从 L.length-1 到 i-1
        L.elem[j+1] = L.elem[j];                 //    若作为位序，有如何修改？
  // 插入 x
  L.elem[i-1] = x;
  // 表长增 1
  L.length++;
  return true;   // 插入成功
}
```

(6) 删除算法 ListDelete(&L,i,&x)

1°. 前提：表非空

2°. 合理的删除范围：1≤i≤L.length

3°. 步骤

取出第 i 个元素
第 i 个元素之后的元素向前移动一个位置
表长减 1

4°. 算法

```
bool ListDelete ( SqList& L, int i, DataType& x )
{
  if ( L.length==0 || i<1 || i>L.length )   return false; // 失败
  x = L.elem[i-1];
  for ( j=i; j<L.length; j++ )
        L.elem[j-1] = L.elem[j];
  L.length--;
  return true;   // 删除成功
}
```

[8] 这里返回 true 表示正确插入，返回 false 表示插入失败。以下常用来区分操作是否正确执行。

（7）算法分析

表 2.1 顺序表插入和删除算法的分析

	插入	删除
基本操作	移动元素	移动元素
平均移动次数	$\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$	$\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$
时间复杂度	O(n)	O(n)
尾端操作	插入第 n+1 个元素，不移动	删除第 n 个元素，不移动

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)。

（8）其他算法

1°. InitList (&L), ClearList (&L)

```
L.length = 0;
```

2°. ListEmpty (L)

```
return L.length == 0;
```

3°. ListLength (L)

```
return L.length;
```

4°. GetElem (L, i, &e)

```
e = L.elem[i-1];
```

3. 单链表——线性表的链式存储结构之一

(1) 概念

线性链表，单链表，结点；数据域，指针域；头指针，头结点。

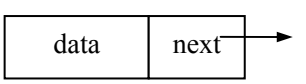
(2) 特点

用指针表示数据之间的逻辑关系（逻辑相邻的元素物理位置不一定相邻）。

(3) 类型定义

简而言之，“数据 + 指针”[9]。

```
typedef struct LNode {
        DataType data;
```



[9] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

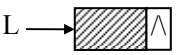
```
        struct LNode *next;
} LNode, *LinkList;
```

(4) 基本形态

带头结点的单链表的基本形态有：

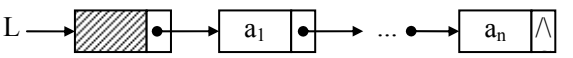
1°. 单链表空

条件： L->next == 0



2°. 单链表不空

条件：L->next != 0



(5) 基本算法 （遍历）

1°. 顺序访问所有元素

借助指针，“顺藤摸瓜”(沿着链表访问结点)。

```
p = L->next;                    // 注意起始位置的考虑
while ( p!=NULL ) {             // 判表尾，另外 (p!=0)或(p)均可
   visit( p->data );            // 访问:可以换成各种操作
   p = p->next;                 // 指针沿着链表向后移动
}
```

例：打印单链表中的数据。

```
void PrintLinkList ( LinkList L )
{
   p = L->next;
   while ( p!=NULL ) {
      print ( p->data );        // 访问：打印数据域
      p = p->next;
   }
}
```

2°. 查找元素 x

```
// 在单链表 L 中查找元素 x
// 若找到，返回指向该结点的指针；否则返回空指针
LinkList Find ( LinkList L, DataType x )
{
   p = L->next;
   while ( p!=NULL ) {
      if ( p->data == x )   return p;   // 找到 x
      p = p->next;
   }
   return NULL;   // 未找到
```

```
}

// 在单链表 L 中查找元素 x
// 若找到，返回该元素的位序；否则返回 0
int Find ( LinkList L, DataType x )
{
   p = L->next;   j = 1;
   while ( p!=NULL ) {
      if ( p->data == x )   return j;   // 找到 x
      p = p->next;   j++;        // 计数器随指针改变
   }
   return 0;   // 未找到
}
```

前一个算法的另一种写法：

```
p = L->next;
while ( p && p->data!=x )
   p = p->next;
if ( p && p->data==x ) return p;
else   return 0;
```

或者

```
p = L->next;
while ( p && p->data!=x )   p = p->next;
return   p;   // 为什么
```

3°. 查找第 i 个元素

```
LinkList Get ( LinkList L, int i )
{
   p = L->next;   j = 1;
   while ( p && j<i ) {
      p = p->next;   j++;
   }
   if ( p && j==i ) return p;
   else   return 0;
}
```

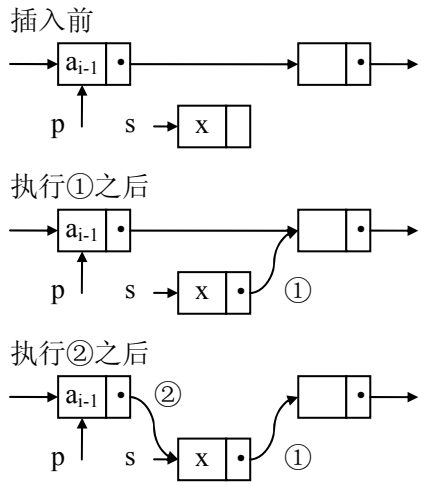
4°. 查找第 i-1 个元素

```
p = L;   j = 0;
while ( p && j<i-1 ) {
   p = p->next;   j++;
}
if ( p && j==i-1 ) return p;
else   return 0;
```

(6) 插入算法 ListInsert(&L,i,x)

技巧：画图辅助分析。
思路：
先查找第 i-1 个元素
若找到，在其后插入新结点

```
bool[10] ListInsert ( LinkList &L, int i, DataType x )
{
  // 查找第 i-1 个元素 p
  p = L;   j = 0;
  while ( p && j<i-1 ) {
    p = p->next;   j++;
  }
  // 若找到，在 p 后插入 x
  if ( p && j==i-1 ) {
    s = (LinkList) malloc(sizeof(LNode));
    s->data = x;
    s->next = p->next;              // ①
    p->next = s;                    // ②
    return true;   // 插入成功
  }
  else
    return false;   // 插入失败
}
```



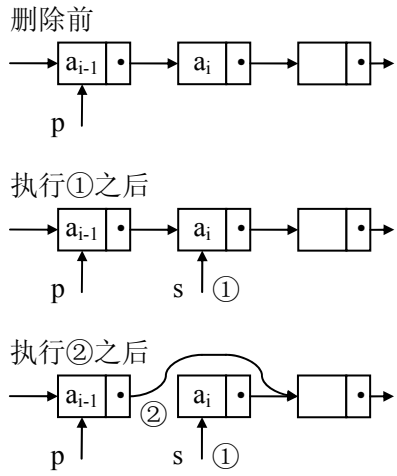
注意：
a) 要让 p 指向第 i-1 个而不是第 i 个元素（否则，不容易找到前驱以便插入）。
b) 能够插入的条件： p && j==i-1 。即使第 i 个元素不存在，只要存在第 i-1 个元素，仍然可以插入第 i 个元素。
c) 新建结点时需要动态分配内存。
s = (LinkList) malloc(sizeof(LNode));
若检查是否分配成功，可用
if (s==NULL) exit(1); // 分配失败则终止程序
d) 完成插入的步骤：①②。技巧：先修改新结点的指针域。

(7) 删除算法 ListDelete(&L,i,&x)

思路：
先查找第 i-1 个元素
若找到且其后存在第 i 个元素，则用 x 返回数据，并删除之

[10] 这里返回 true 表示正确插入，返回 false 表示插入失败。

```
bool ListDelete ( LinkList &L, int i, int &x )
{
  // 查找第 i-1 个元素 p
  p = L;   j = 0;
  while ( p && j<i-1 ) {
    p = p->next;   j++;
  }
  //若存在第 i 个元素，则用 x 返回数据，并删除之
  if ( p && j==i-1 && p->next ) { // 可以删除
    s = p->next;              // ①
    p->next = s->next;        // ②
    x = s->data;
    free (s);
    return true;
  }
  else
    return false;
}
```



注意：

a) 要求 p 找到第 i-1 个而非第 i 个元素。为什么？

b) 能够进行删除的条件： p && j==i-1 && p->next 。条件中的 p->next 就是要保证第 i 个元素存在，否则无法删除。若写成 p->next && j==i-1 也不妥，因为此时(循环结束时)可能有 p==NULL，所以必须先确定 p 不空。技巧：将条件中的“大前提”放在前面。该条件也不可以写成 p->next && p && j==i-1，因为先有 p!=0 才有 p->next，上式颠倒了这一关系。

c) 释放结点的方法。 free(s);

d) 完成删除的步骤：①②。

(8) 建立链表的两种方法

思路：

建立空表（头结点）；

依次插入数据结点（每次插入表尾得($a_1,a_2,\ldots,a_n$)，每次插入表头得($a_n,\ldots,a_2,a_1$)）。

1°. 顺序建表

```
void CreateLinkList ( LinkList &L, int n)
{
  // 建立空表
  L = (LinkList) malloc(sizeof(LNode));
  L->next = NULL;              // 空表
  p = L;                       // 用 p 指向表尾
  // 插入元素
  for ( i=0; i<n; i++ ) {
    scanf ( x );
```

```
        s = (LinkList) malloc(sizeof(LNode));
        s->data = x;
        // 插入表尾
        s->next = p->next;
        p->next = s;
        p = s;                          // 新的表尾
    }
}
```

2°. 逆序建表

```
void CreateLinkList ( LinkList &L, int n)
{
    // 建立空表
    L = (LinkList) malloc(sizeof(LNode));
    L->next = NULL;    // 空表
    // 插入元素
    for ( i=0; i<n; i++ ) {
        scanf ( x );
        s = (LinkList) malloc(sizeof(LNode));
        s->data = x;
        // 插入表头
        s->next = L->next;
        L->next = s;
    }
}
```

4. 循环链表

(1) 特点

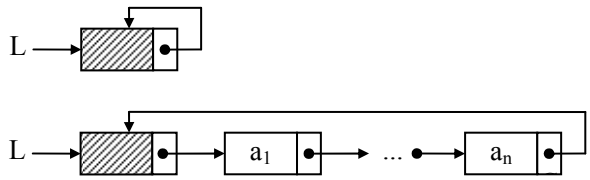
最后一个结点的指针指向头结点。

(2) 类型定义

同单链表。

(3) 基本形态

空表：L->next == L。
非空表。



(4) 与单链表的联系

判断表尾的方法不同：单链表用 p==NULL；循环链表用 p==L。
其余操作相同。

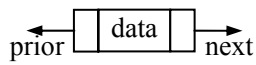
5. 双向循环链表

(1) 特点

一个结点包含指向后继(next)和指向前驱(prior)两个指针，两个方向又分别构成循环链表。

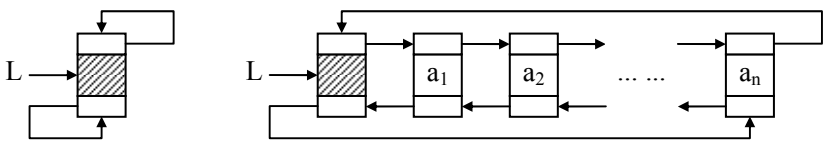
(2) 类型定义

```
typedef struct DuLNode {
    DataType data;
    struct DuLNode *prior, *next;    // 两个指针
} DuLNode, *DuLinkList;
```



(3) 基本形态

空表：用后向指针判断 L->next == L，或者用前向指针判断 L->prior == L。
非空表。



(4) 与单链表和循环链表的联系

最大不同：前驱容易求得，可以向前遍历。
判断表尾的方法与循环链表相同：p==L。
插入和删除时需要修改两个方向的指针。

(5) 插入和删除

需要修改两个方向的指针。例如：(见下表)

表 2.2 双向循环链表的插入和删除

p 之后插入 s	p 之前插入 s	删除 p 之后继 s	删除 p

s->next = p->next;
p->next = s;
s->prior = p;
s->next->prior = s;
	s->prior = p->prior;
	p->prior = s;
	s->next = p;
	s->prior->next = s;
		s = p->next;
		p->next = s->next;
		p->next->prior = p;
			p->prior->next = p->next;
			p->next->prior = p->prior;

6. 顺序表与单链表的比较

表 2.3 顺序表和单链表的比较

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)

总结：需要反复插入、删除，宜采用链表；反复提取，很少插入、删除，宜采用顺序表。

二、习题

2.1 将顺序表中的元素反转顺序。
2.2 在非递减有序的顺序表中插入元素 x，并保持有序。
2.3 删除顺序表中所有等于 x 的元素。
2.4 编写算法实现顺序表元素唯一化(即使顺序表中重复的元素只保留一个)，给出算法的时间复杂度。
2.5 非递减有序的顺序表元素唯一化(参见习题 2.4)，要求算法的时间复杂度为O(n)。
2.6 将单链表就地逆置，即不另外开辟结点空间，而将链表元素翻转顺序。
2.7 采用插入法将单链表中的元素排序。
2.8 采用选择法将单链表中的元素排序。
2.9 将两个非递减有序的单链表归并成一个，仍并保持非递减有序。

第3章 栈和队列

一、基础知识和算法

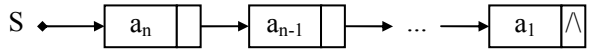
1. 栈

栈，栈顶，栈底，空栈，后进先出(LIFO)，入栈(Push)，出栈(Pop)。
顺序栈：栈的顺序存储结构；**链栈：**栈的链式存储结构。

2. 链栈

(1) 存储结构

用不带头结点的单链表实现。



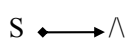
(2) 类型定义

同单链表。

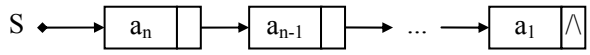
(3) 基本形态

1°. 栈空

条件: S == NULL



2°. 栈非空



3°. 栈满（一般不出现）

(4) 基本算法

1°. 入栈 Push (&s, x)

```
bool Push ( LinkList &s, DataType x )
{
  // 新建结点
```

```
   p = (LinkList) malloc (sizeof(LNode));
   if ( !p )    return false; //  失败
   p->data = x;
   //  插入栈顶
   p->next = s;
   s = p;
   return true;
}
```

2°. 出栈 Pop (&s, &x)

前提：栈非空。

```
bool Pop ( LinkList &s, DataType &x )
{
   if ( s==NULL )    return false;    //  栈空
   //  删除栈顶元素
   p = s;
   s = s->next;
   x = p->data;
   free ( p );
   return true;
}
```

3°. 栈顶元素

前提：栈非空。

```
bool Top ( LinkList &s, DataType &x )
{
   if ( s==NULL )    return false;    //  栈空
   x = s->data;
   return true;
}
```

3. 顺序栈

(1) 存储结构

类似于顺序表，插入和删除操作固定于表尾。

(2) 类型定义

简单说，“数组 + 长度”[11]。

```
const int MAXSIZE =  栈的最大容量;
typedef struct {
   DataType elem[MAXSIZE];
```

[11] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

```
   int top;
} SqStack;
```

(3) 基本形态

1°. 栈空

条件　s.top == 0;

2°. 栈满

条件　s.top == MAXSIZE

3°. 栈不空、不满

(4) 基本算法

1°. 入栈　Push (&s, x)

前提：栈不满

```
bool Push ( SqStack& s, DataType x )
{
   if ( s.top == MAXSIZE )    return false;    // 栈满
   s.elem[top] = x;                    // 或者 s.elem[top++] = x;
   top++;                              //     代替这两行
   return true;
}
```

2°. 出栈　Pop (&s, &x)

前提：栈非空

```
bool Pop ( SqStack &s, DataType &x )
{
   if ( s.top==0 )    return false;
   top--;                              // 可用 x=s.elem[--top];
   x = s.elem[top];                    //     代替这两行
   return true;
}
```

3°. 栈顶元素

前提：栈非空

s.elem[top-1] 即是。

4. 队列

队列，队头，队尾，空队列，先进先出(FIFO)。

链队列：队列的链式存储结构。
循环队列：队列的顺序存储结构之一。

5. 链队列

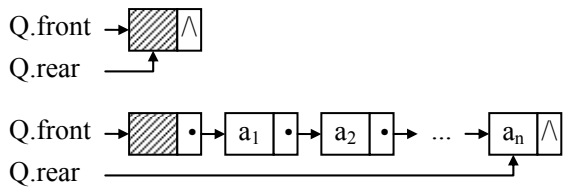
(1) 存储结构

简而言之，“单链表 + 尾指针”[12]。

(2) 类型定义

课本 P61。

```
typedef struct {
    LinkList front;
    LinkList rear;
} LinkQueue;
```



(3) 基本形态

队列空：Q.front==Q.rear。
非空队列。

(4) 基本算法

1°. 入队列

课本 P62。插入队尾，注意保持 Q.rear 指向队尾。

2°. 出队列

课本 P62。删除队头元素，
特别注意：如果队列中只有一个元素，则队头也同时是队尾，删除队头元素后也需要修改队尾指针。

6. 循环队列

(1) 存储结构

简单说，“数组 + 头、尾位置”[13]。

[12] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。
[13] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(2) 类型定义

```
const int MAXSIZE =  队列最大容量;
typedef struct {
    DataType elem[MAXSIZE];
    int front, rear;                    // 队头、队尾位置
} SqQueue;
```

(3) 基本形态

通常少用一个元素区分队列空和队列满，也可以加一标志。约定 front 指向队头元素的位置，rear 指向队尾的下一个位置，队列内容为 [front, rear)。

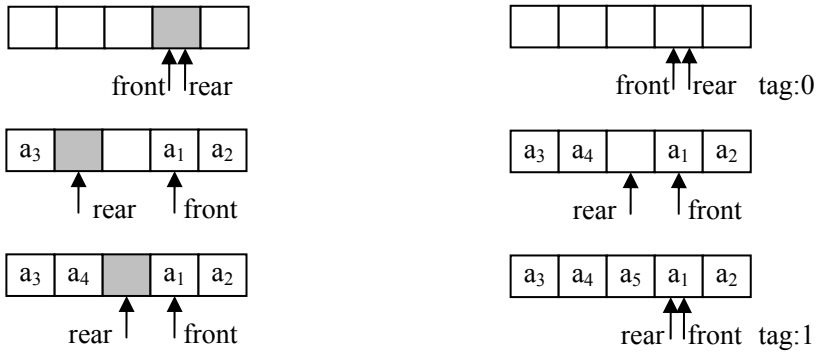
1°. 队列空

条件：Q.front == Q.rear。
不能出队列。

2°. 队列满

条件：(Q.rear+1)%MAXSIZE == Q.front （少用一个元素时）。
不能入队列。

3°. 队列不空也不满



4°. 加一标志区分队列空和队列满的情况

可以用满所有空间。队列空和队列满时都有 Q.front==Q.rear，再用标志区分。
队列空：Q.front==Q.rear **and** Q.tag==0；队列满：Q.front==Q.rear **and** Q.tag==1。

(4) 基本算法

1°. 入队列

前提：队列不满。

```
bool EnQueue ( SqQueue &Q, DataType x )
{
```

```
   if ( (Q.rear+1)%MAXSIZE==Q.front )    return false;   // 队列满
   // 入队列
   Q.elem [Q.rear] = x;
   Q.rear = (Q.rear+1)%MAXSIZE;
   return true;
}
```

2°. 出队列

前提：队列非空。

```
bool DeQueue ( SqQueue &Q, DataType &x )
{
   if ( Q.front==Q.rear )    return false;   // 队列空
   // 出队列
   x = Q.elem [Q.front];
   Q.front = (Q.front+1)%MAXSIZE;
   return true;
}
```

3°. 队列中元素个数

结论：(Q.rear-Q.front+MAXSIZE)%MAXSIZE。

注：Q.rear-Q.front 可能小于 0，需要加上 MAXSIZE。

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

4°. 用标志区分队列空和满

用标志区分队列空和满时，队列初始化、入队列、出队列和队列长度的算法如下：

```
void InitQueue ( SqQueue &Q ) {
   Q.front = Q.rear = 0;    Q.tag = 0;
}
bool EnQueue ( SqQueue &Q, DataType x ) {
   if ( Q.front==Q.rear and Q.tag==1 )    return false;
   Q.elem[ Q.rear ] = x;
   Q.rear = (Q.rear+1)%MAXSIZE;
   if ( Q.tag==0 )    Q.tag = 1;                          // 队列非空
   return true;
}
bool DeQueue ( SqQueue &Q, DataType &x ) {
   if ( Q.front==Q.rear and Q.tag==0 )    return false;
   x = Q.elem[ Q.front ];
   Q.front = (Q.front+1)%MAXSIZE;
   if ( Q.front==Q.rear )    Q.tag = 0;                   // 队列空
   return true;
}
int QueueLength ( SqQueue Q )
```

```
{
  if ( Q.front==Q.rear and Q.tag==1 )
    return MAXSIZE;                              // 队列满
  else
    return (Q.rear-Q.front+MAXSIZE)%MAXSIZE;     // 队列不满(包含队列空的情
况)
}
```

7. 栈和队列比较

都是线形结构，栈的操作 LIFO（后进先出），队列操作 FIFO（先进先出）。

8. 简化的栈和队列结构

在算法中使用栈和队列时可以采用简化的形式。

表 3.1 简化的栈和队列结构

简化栈		简化队列	
结构	“s[] + top”	结构	“q[] + front + rear”
初始化	top = 0;	初始化	front=rear=0;
入栈	s[top++] = x;	入队列	q[rear] = x; rear = (rear+1)%MAXSIZE;
出栈	x = s[--top];	出队列	x = q[front]; front = (front+1)%MAXSIZE;
栈顶	s[top-1]	队列头	q[front]
栈空	top == 0	队列空	front == rear

说明：只要栈(队列)的容量足够大，算法中可以省去检查栈(队列)满的情况。

9. 栈和队列的应用

1°. 表达式求值

参见课本 P53。

2°. 括号匹配

例：检查表达式中的括号是否正确匹配。如{ ( ) [ ] }正确，([ )] }则错误。
分析：每个左括号都“期待”对应的右括号，匹配成功则可以消去。
思路：遇到左括号则入栈，遇到右括号则与栈顶括号相比较，如果匹配则消去，否则匹配失败。当然，如果栈中没有括号可以匹配，或者最后栈中还有未匹配的左括号，也都是匹配错误。

```
// 检查输入的表达式中括号是否匹配
bool MatchBrackets ()
{
  const int MAXSIZE = 1024;                      // 栈的最大容量
```

```
   char s [MAXSIZE];                          // 简化的栈结构
   int top;                                   // 栈顶
   // 栈初始化
   top = 0;
   // 检查括号是否匹配
   ch = getchar();
   while ( ch!=EOF ) {
      switch ( ch ) {
      case '(', '[', '{':
         s [top++] = ch;                      // 所有左括号入栈
         break;
      case ')':
         if ( top==0 or s [--top]!='(' ) return false;      // 栈空或右括号与栈顶左括号失
配
      case ']':
         if ( top==0 or s [--top]!='[' ) return false;
      case '}':
         if ( top==0 or s [--top]!='{' ) return false;
      }
      ch = getchar();                         // 取下一个字符
   }
   if ( top==0 )    return true;              // 正好匹配
   else    return false;                      // 栈中尚有未匹配的左括号
}
```

3°. 递归程序的非递归化

将递归程序转化为非递归程序时常使用栈来实现。

4°. 作业排队

如操作系统中的作业调度中的作业排队，打印机的打印作业也排成队列。

5°. 按层次遍历二叉树

```
void LevelOrder ( BinTree bt, VisitFunc visit )
{
   const int MAXSIZE = 1024;     // 队列容量(足够大即可)
   BinTree q [MAXSIZE];     // 简化的队列结构
   int front, rear;                 //     队头、队尾

   if ( ! bt )    return ;
   // 初始化队列，根结点入队列
   front = rear = 0;
   q [rear] = bt;
   rear = (rear+1)%MAXSIZE;
   // 队列不空，则取出队头访问并将其左右孩子入队列
   while ( front!=rear ) {
```

```
        p = q [front];
        front = (front+1)%MAXSIZE;
        if ( p ) {
            visit ( p->data );        // 访问结点
            q [rear] = p->lchild;
            rear = (rear+1)%MAXSIZE;
            q [rear] = p->rchild;
            rear = (rear+1)%MAXSIZE;
        }
    }
}
```

二、习题

3.1 元素 1,2,3,4 依次入栈，不可能的出栈序列有哪些？
3.2 设循环队列 Q 少用一个元素区分队列空和队列满，MAXSIZE=5，Q.front=Q.rear=0，画出执行下列操作时队列空和队列满的状态。入队列 a,b,c，出队列 a,b,c，入队列 d,e,f,g。
3.3 编写算法利用栈将队列中的元素翻转顺序。

第4章 串

一、基础知识和算法

1. 概念

串，空串，空格串，串的长度；子串，子串在主串中的位置，主串；串相等。

2. 串的基本操作

表 4.1 串的基本操作

Assign (s, t), Create (s, cs)	Assign(s,t)将变量 t 赋值给 s，Create(s,cs)根据字串创建变量 s。
Equal (s, t), Length (s)	判断串相等，求串长度。如 Length(“”)=0。
Concat (s, t)	串连接。如 Concat(“ab”,”cd”)=”abcd”。
Substr (s, pos, len)	取子串，pos 为开始位置，len 为子串长度。
Index (s, t)	求子串 t 在主串 s 中的位置。如 Index(“abc”,”ab”)=1，Index(“a bc”,”bc”)=3。
Replace (s, t, v)	把串 s 中的字串 t 替换成 v。如 Replace(“aaa”,”aa”,”a”)=”aa”。
Delete (s, pos, len)	删除串 s 的一部分。

注：完成习题集 4.1~4.6。

3. 串的存储结构

表 4.2 串的存储结构

定长顺序串	最大长度固定，超过最大长度则作截断处理
堆分配存储表示	串的长度几乎没有限制
块链存储表示	块内存储空间连续，块间不连续

二、习题

4.1 长度为 n 的串的子串最多有多少个？

第6章 树和二叉树

一、基础知识和算法

1. 树及有关概念

树，根，子树；结点，结点的度，叶子（终端结点），分支结点（非终端结点），内部结点，树的度；孩子，双亲，兄弟，祖先，子孙，堂兄弟；层次（根所在层为第 1 层），**深度，高度；有序树，无序树**，二叉树是有序树；**森林**。

2. 二叉树

二叉树（二叉树与度为 2 的树不同，二叉树的度可能是 0，1，2）；**左孩子，右孩子**。二叉树的五种基本形态。

3. 二叉树的性质

1°. 二叉树的第i层[14]上至多有 2^{i-1} 个结点。

2°. 深度为 k 的二叉树至多有 2^k-1 个结点。

满二叉树：深度为 k，有 2^k-1 个结点。
完全二叉树：给满二叉树的结点编号，从上至下，从左至右，n 个结点的完全二叉树中结点在对应满二叉树中的编号正好是从 1 到 n。

3°. 叶子结点 n_0，度为 2 的结点为 n_2，则 $n_0 = n_2+1$。

考虑结点个数：$n = n_0 + n_1 + n_2$
考虑分支个数：$n-1 = 2n_2 + n_1$
可得 $n_0 = n_2+1$
例：1) 二叉树有 n 个叶子，没有度为 1 的结点，共有____个结点。 2) 完全二叉树的第 3 层有 2 个叶子，则共有____个结点。
分析：1) 度为 2 的结点有 n-1 个，所以共 2n-1 个结点。 2) 注意考虑到符合条件的二叉树的深度可能是 3 或 4，所以有 5、10 或 11 个结点。

[14] 本书中约定根结点在第 1 层，也有约定根在第 0 层的，则计算公式会有所不同。

4°. n 个结点的完全二叉树深度为$\lfloor \log n \rfloor + 1$。

5°. n 个结点的完全二叉树，结点按层次编号

有： i 的双亲是$\lfloor n/2 \rfloor$，如果 i = 1 时为根（无双亲）；
i 的左孩子是 2i，如果 2i>n，则无左孩子；
i 的右孩子是 2i + 1，如果 2i + 1>n 则无右孩子。

4. 二叉树的存储结构

1°. 顺序存储结构

用数组、编号 i 的结点存放在[i-1]处。适合于存储完全二叉树。

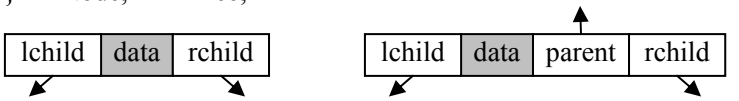
2°. 链式存储结构

二叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild;
} BTNode, *BinTree;
```

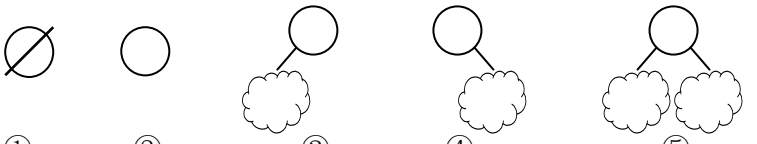
三叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild, *parent;
} BTNode, *BinTree;
```



例：用二叉链表存储n个结点的二叉树(n>0)，共有(_a_)个空指针域；采用三叉链表存储，共有(_b_)个空指针域。[15]

5. 二叉树的五种基本形态



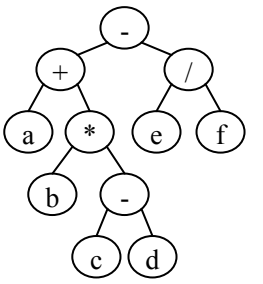
①空树：bt==NULL ②左右子树均空：bt->lchild==NULL **and** bt->rchild==NULL
③右子树为空：bt->rchild==NULL ④左子树为空：bt->lchild==NULL
⑤左右子树均非空。
前两种常作为递归结束条件，后三者常需要递归。

[15] 答案：(a) n+1 (b) n+2。提示：只有根结点没有双亲。

6. 遍历二叉树

1°. 常见有四种遍历方式

按层次遍历，先序遍历，中序遍历，后序遍历。
按层次遍历：“从上至下，从左至右”，利用队列。
先序遍历：DLR；**中序遍历**：LDR；**后序遍历** LRD。



例：写出 a+b*(c-d)-e/f 的前缀、中缀和后缀表达式。
画出二叉树，分别进行前序、中序、后序遍历即可得到。
前缀表达式：- + a * b - c d / e f
中缀表达式：a + b * c - d - e / f
后缀表达式：a b c d - * + e f / -

2°. 先序遍历算法

```
void Preorder ( BinTree bt )
{
   if ( bt ) {
      visit ( bt->data );
      Preorder ( bt->lchild );
      Preorder ( bt->rchild );
   }
}
```

3°. 中序遍历算法

```
void Inorder ( BinTree bt )
{
   if ( bt ) {
      Inorder ( bt->lchild );
      visit ( bt->data );
      Inorder ( bt->rchild );
   }
}
```

4°. 后序遍历

```
void Postorder ( BinTree bt )
{
   if ( bt ) {
      Postorder ( bt->lchild );
      Postorder ( bt->rchild );
      visit ( bt->data );
   }
}
```

5°. 按层次遍历

思路：利用一个队列，首先将根（头指针）入队列，以后若队列不空则取队头元素 p，如果 p 不空，则访问之，然后将其左右子树入队列，如此循环直到队列为空。

```
void LevelOrder ( BinTree bt )
{
  // 队列初始化为空
  InitQueue ( Q );
  // 根入队列
  EnQueue ( Q, bt );
  // 队列不空则继续遍历
  while ( ! QueueEmpty(Q) ) {
    DeQueue ( Q, p );
    if ( p!=NULL ) {
      visit ( p->data );
      // 左、右子树入队列
      EnQueue ( Q, p->lchild );
      EnQueue ( Q, p->rchild );
    }
  }
}
```

若队列表示为“数组 q[] + 头尾 front, rear”有：

```
void LevelOrder ( BinTree bt )
{
  const int MAXSIZE = 1024;
  BinTree q[MAXSIZE];
  int front,  rear;
  // 队列初始化为空
  front = rear = 0;
  // 根入队列
  q[rear] = bt;  rear = ( rear+1 ) % MAXSIZE;
  // 队列不空则循环
  while ( front != rear ) {
    p = q[front];  front = ( front+1 ) % MAXSIZE;
    if ( p ) {
      visit ( p->data );
      // 左、右子树入队列
      q[rear] = p->lchild;  rear = ( rear+1 ) % MAXSIZE;
      q[rear] = p->rchild;  rear = ( rear+1 ) % MAXSIZE;
    }
  }
}
```

6°. 非递归遍历二叉树

一般借助栈实现。设想一指针沿二叉树中序顺序移动，每当向上层移动时就要出栈。

(a) 中序非递归遍历

指针 p 从根开始，首先沿着左子树向下移动，同时入栈保存；当到达空子树后需要退栈访问结点，然后移动到右子树上去。

```
void InOrder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   p = bt;
   while ( p || ! StackEmpty(S) ) {
      if ( p ) {
         Push ( S, p );
         p = p->lchild;
      } else {
         Pop ( S, p );
         visit ( p );                          // 中序访问结点的位置
         p = p->rchild;
      }
   }
}
```

(b) 先序非递归遍历

按照中序遍历的顺序，将访问结点的位置放在第一次指向该结点时。

```
void Preorder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   p = bt;
   while ( p || ! StackEmpty(S) ) {
      if ( p ) {
         visit ( p );                    // 先序访问结点的位置
         Push ( S, p );
         p = p->lchild;
      } else {
         Pop ( S, p );
         p = p->rchild;
      }
   }
}
```

或者，由于访问过的结点便可以弃之不用，只要能访问其左右子树即可，写出如下算法。

```
void Preorder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   Push ( S, bt );
   while ( ! StackEmpty(S) ) {
      Pop ( S, p );
      if ( p ) {
         visit ( p );
         Push ( S, p->rchild );          // 先进栈，后访问，所以
         Push ( S, p->lchild );          //    这里先让右子树进栈
      }
```

```
    }
  }
```

(c) 后序非递归遍历

后序遍历时，分别从左子树和右子树共两次返回根结点，只有从右子树返回时才访问根结点，所以增加一个栈标记到达结点的次序。

```
void PostOrder ( BinTree bt, VisitFunc visit )
{
  InitStack ( S ),   InitStack ( tag );
  p = bt;
  while ( p || ! StackEmpty(S) ) {
    if ( p ) {
      Push ( S, p ),   Push ( tag, 1);     // 第一次入栈
      p = p->lchild;
    } else {
      Pop ( S, p ),   Pop ( tag, f );
      if ( f==1 ) {
        // 从左子树返回，二次入栈，然后 p 转右子树
        Push ( S, p ),   Push ( tag, 2 );
        p = p->rchild;
      } else {
        // 从右子树返回(二次出栈)，访问根结点，p 转上层
        visit ( p );
        p = NULL;                          // 必须的，使下一步继续退栈
      }
    }
  }
}
```

注：后序非递归遍历的过程中，栈中保留的是当前结点的所有祖先。这是和先序及中序遍历不同的。在某些和祖先有关的算法中，此算法很有价值。

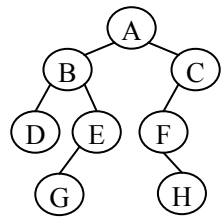
7°. 三叉链表的遍历算法

下面以中序遍历为例。

```
// 中序遍历三叉链表存储的二叉树
void Inorder ( BinTree bt, VisitFunc visit )
{
  if ( bt==NULL )   return;                // 空树，以下考虑非空树
  // 找到遍历的起点
  p = bt;                                  // Note: p!=null here
  while ( p->lchild )   p = p->lchild;
  // 开始遍历
  while ( p ) {
    // 访问结点
    visit ( p );
    // p 转下一个结点
    if ( p->rchild ) {                     // 右子树不空，下一个在右子树
      p = p->rchild;
```

```
        while ( p->lchild )   p = p->lchild;   //    转右子树的最左下结点
      } else {                                 // 右子树为空，下一个在上层
        f = p->parent;
        while ( p == f->rchild ) {             //    若 p 是右子树则一直上溯
          p = f;   f = f->parent;
        }
      }
    }
}
```

7. 遍历二叉树的应用



1°. 写出遍历序列（前、中、后序）

2°. 根据遍历序列画出二叉树

(a) 已知前序和中序序列，唯一确定二叉树。
例：前序：ABDEGCFH，中序：DBGEAFHC，画出二叉树。
分析：前序序列的第一个是根，这是解题的突破口。
步骤：①前序序列的第一个是根 ②在中序序列中标出根，分成左右子树 ③在前序序列中标出左右子树(根据结点个数即可) ④分别对左右子树的前序和中序序列重复以上步骤直至完成。
(b) 已知后序和中序序列，唯一确定二叉树。
例：后序：DGEBHFCA，中序：DBGEAFHC，画出二叉树。
分析：后序序列的最后一个是根，这是解题的突破口。
步骤：①后序序列的最后一个是根 ②在中序序列中标出根，分成左右子树 ③在后序序列中标出左右子树(根据结点个数即可) ④分别对左右子树的后序和中序序列重复以上步骤直至完成。
(c) 已知前序和后序序列，不存在度为 1 的结点时能唯一确定二叉树。
例：前序：ABDEC，后序：DEBCA，画出二叉树。又前序 AB，后序 BA 则不能唯一确定二叉树。
注：对于不存在度为 1 的结点的二叉树，首先确定根结点，然后总可以将其余结点序列划分成左右子树，以此类推即可确定二叉树。

说明：画出二叉树后可以进行遍历以便验证。

3°. 编写算法

思路：按五种形态(①--⑤)分析，适度简化。
例：求二叉树结点的个数。
分析：① 0; ② 1; ③ L+1; ④ 1+R; ⑤ 1+L+R。
简化：② $1+L_{=0}+R_{=0}$ ③ $1+L+R_{=0}$ ④ $1+L_{=0}+R$ ⑤ 1+L+R 可合并成⑤一种情况。

```
int NodeCount ( BinTree bt )
{
  if ( bt==0 )   return 0;
  else   return 1 + NodeCount(bt->lchild) + NodeCount(bt->rchild);
}
```

例：求二叉树叶子结点的个数。
分析：① 0; ② 1; ③ L; ④ R; ⑤ L+R。简化：③④⑤可合并成⑤。

```
int LeafCount ( BinTree bt )
{
  if ( bt==0 )   return 0;
  else if ( bt->lchild==0 and bt->rchild==0 )   return 1;
  else   return LeafCount(bt->lchild) + LeafCount(bt->rchild);
}
```

例：求二叉树的深度。
分析：① 0; ② 1; ③ 1+L; ④ 1+R; ⑤ 1+max(L,R)。简化：②③④⑤可合并成⑤。

```
int Depth ( BinTree bt )
{
  if ( bt==0 )   return 0;
  else   return 1 + max(Depth(bt->lchild), Depth(bt->rchild));
}
```

8. 线索二叉树

1°. 线索

n 个结点的二叉链表中有 n+1 个空指针，可以利用其指向前驱或后继结点，叫**线索**，同时需附加一个标志，区分是子树还是线索。

lchild　有左子树，则指向左子树，标志 ltag == 0；
　　没有左子树，可作为前驱线索，标志 ltag == 1。
rchild　有右子树，则指向右子树，标志 rtag == 0；
　　没有右子树，可作为后继线索，标志 rtag == 1。

2°. 线索化二叉树

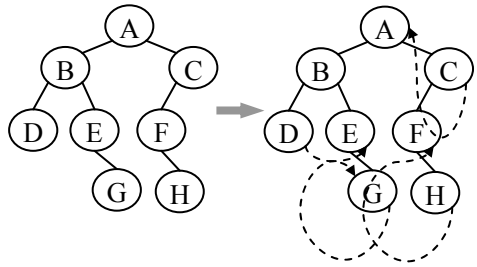
利用空指针作为线索指向前驱或后继。左边空指针可以作为前驱线索，右边空指针可以作为后继线索，可以全线索化或部分线索化。

表 6.1 线索化二叉树的类型

	前驱、后继线索	前驱线索	后继线索
中序线索化	中序全线索	中序前驱线索	中序后继线索
前序线索化	前序全线索	前序前驱线索	前序后继线索
后序线索化	后序全线索	后序前驱线索	后序后继线索

3°. 画出线索二叉树

思路：先写出遍历序列，再画线索。
步骤： 标出必要的空指针（前驱→左指针；后继→右指针，要点：“不要多标，也不要少标”）。
写出对应的遍历序列（前序，中序或后序）。
对照遍历结果画线索。
例：画出图中二叉树的后序后继线索。
步骤：先标出所有空的右指针(DGCH)；写出后序遍历结果：DGEBHFCA；标出后继线索：D→G, G→E, C→A, H→F。如图。



4°. 遍历线索二叉树

反复利用孩子和线索进行遍历，可以避免递归。

9. 树和森林

1°. 树的存储结构

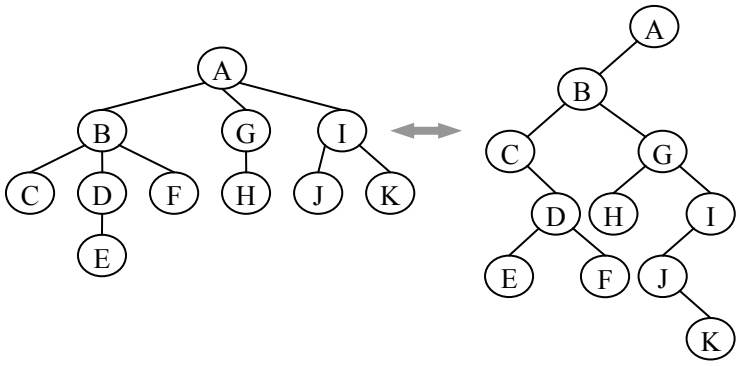
双亲表示法，孩子表示法，孩子兄弟表示法。
特点：双亲表示法容易求得双亲，但不容易求得孩子；孩子表示法容易求得孩子，但求双亲麻烦；两者可以结合起来使用。孩子兄弟表示法，容易求得孩子和兄弟，求双亲麻烦，也可以增加指向双亲的指针来解决。

2°. 树与二叉树的转换

表 6.2 树和二叉树的对应关系

树	对应的二叉树
根	根
第一个孩子	左孩子
下一个兄弟	右孩子

特点：由树转化成的二叉树，根结点没有右孩子
例：树转换成二叉树。



3°. 森林与二叉树的转换

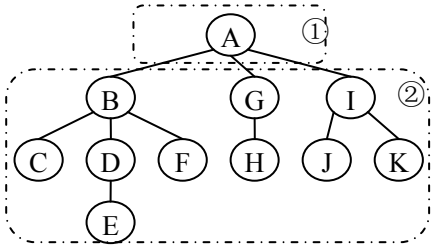
森林中第 1 棵树的根作为对应的二叉树的根；其他的树看作第 1 棵树的兄弟；森林中的树转换成对应的二叉树。则森林转换成对应的二叉树。
例：将森林转换成对应的二叉树。参见课本 P138。

4°. 树的遍历

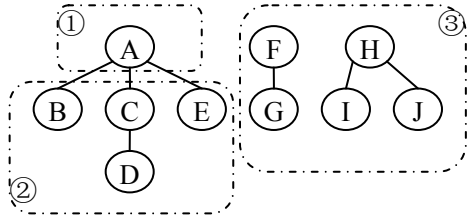
树的结构：①根，②根的子树。
先根遍历：①②。例：ABCDEFGHIJK。
后根遍历：②①。例：CEDFBHGJKIA。

5°. 遍历森林

森林的结构：①第一棵树的根，②第一棵树的根的子树森林，③ 其余树(除第一棵外)组成的森林。
先序遍历：①②③。例：ABCDEFGHIJ。
中序遍历：②①③。例：BDCEAGFIJH。
注：先序遍历森林，相当于依次先根遍历每一棵树；中根遍历森林相当于后根遍历每一棵树。



树的结构划



森林的结构划分

6°. 遍历树、森林与遍历二叉树的关系

表 6.3 遍历树、森林和二叉树的关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

10. 赫夫曼树及其应用

1°. 最优二叉树(赫夫曼树，哈夫曼树)

树的带权路径长度：所有叶子结点的带权路径长度之和。

$$WPL = \sum_{k=1}^{n} w_k l_k$$

路径长度 l_k 按分支数目计算。
带权路径长度最小的二叉树称为**最优二叉树**，或**赫夫曼树(哈夫曼树)**。

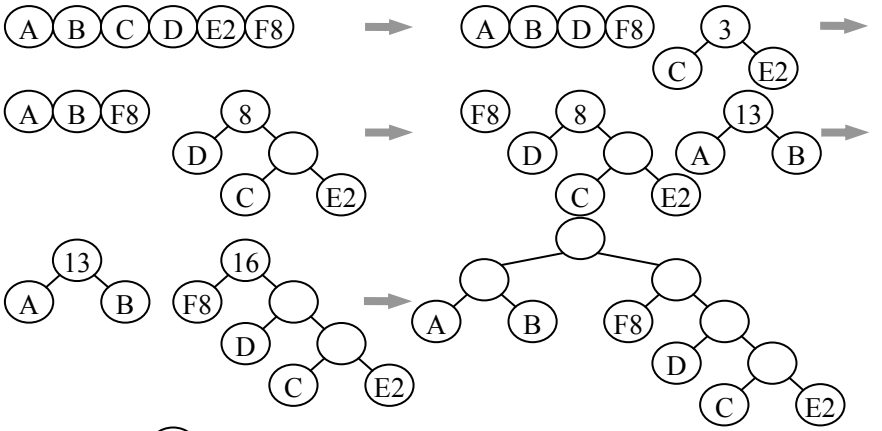
2°. 构造赫夫曼树

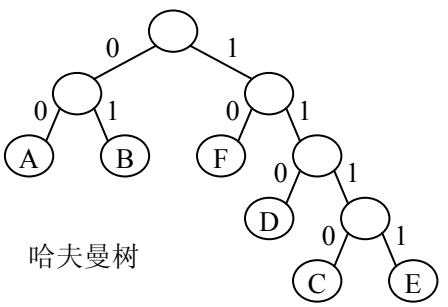
算法：参见课本 P145。
简单说，“每次取两个最小的树组成二叉树”[16]。

3°. 赫夫曼编码(前缀码)

向左分支为 0，向右分支为 1，从根到叶子的路径构成叶子的前缀编码。

例：字符及其权值如下：A(6), B(7), C(1), D(5), E(2), F(8)，构造哈夫曼树和哈夫曼编码，计算带权路径长度。





哈夫曼树

哈夫曼编码：
A:00
B:01
C:1110
D:110
E:1111
F:10

WPL = (6+7+8)*2+5*3+(1+2)*4 = 69

或采用课本上的算法计算，如下。

表 6.4 赫夫曼算法

		weight	parent	lchild	rchild

		weight	parent	lchild	rchild

[16] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

1	A	6	0	0	0		1	A	6	9	0	0
2	B	7	0	0	0		2	B	7	9	0	0
3	C	1	0	0	0		3	C	1	7	0	0
4	D	5	0	0	0		4	D	5	8	0	0
5	E	2	0	0	0		5	E	2	7	0	0
6	F	8	0	0	0		6	F	8	10	0	0
7		0	0	0	0		7		3	8	3	5
8		0	0	0	0		8		8	10	4	7
9		0	0	0	0		9		13	11	1	2
10		0	0	0	0		10		16	11	6	8
11		0	0	0	0		11		29	0	9	10

结果同上。

说明：同样的一组权值可能构造出不同的哈夫曼树，结果不一定唯一，但带权路径长度都是最小的。

技巧：要使前一种方法构造出的赫夫曼树和课本上算法产生的一样，只要把每次合并产生的二叉树放在树的集合的末尾，并且总是用两个最小树的前者作为左子树后者作为右子树。

二、习题

6.1 度为 k 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，…，n_k 个度为 k 的结点，问该树中有多少个叶子结点。

6.2 有 n 个叶子结点的完全二叉树的高度是多少?

6.3 编写算法按照缩进形式打印二叉树。

6.4 编写算法按照逆时针旋转 90 度的形式打印二叉树。

6.5 编写算法判断二叉树是否是完全二叉树。

6.6 编写算法求二叉树中给定结点的所有祖先。

6.7 编写算法求二叉树中两个结点的最近的共同祖先。

6.8 编写算法输出以二叉树表示的算术表达式（中缀形式），要求在必要的地方输出括号。

6.9 树采用孩子-兄弟链表存储，编写算法求树中叶子结点的个数。

6.10 采用孩子-兄弟链表存储树，编写算法求树的度。

6.11 采用孩子-兄弟链表存储树，编写算法求树的深度。

6.12 已知二叉树的前序和中序序列，编写算法建立该二叉树。

6.13 树 T 的先根遍历序列为 GFKDAIEBCHJ，后根遍历序列为 DIAEKFCJHBG，画出树 T。

6.14 一森林 F 转换成的二叉树的先序序列为 ABCDEFGHIJKL，中序序列为 CBEFDGAJIKLH。画出森林 F。

6.15 某通信过程中使用的编码有 8 个字符 A,B,C,D,E,F,G,H，其出现的次数分别为 20, 6, 34, 11, 9, 7, 8, 5。若每个字符采用 3 位二进制数编码，整个通信需要多少字节？请给出哈夫曼编码，以及整个通信使用的字节数?

6.16 n 个权值构造的哈夫曼树共有多少个结点?

第7章 图

一、基础知识和算法

1. 图的有关概念

图，顶点，弧，弧头，弧尾；有向图（顶点集+弧集），0≤e≤n(n-1)，**无向图**（顶点集+边集），0≤e≤n(n-1)/2；**稀疏图**（e<nlogn），**稠密图**；**完全图** e=n(n-1)/2，**有向完全图** e=n(n-1)；**网，有向网，无向网。子图，邻接点，顶点的度，入度，出度；路径，路径长度**（经过边或弧的数目），**简单路径，回路**（环），**简单回路**（简单环）；**连通图，连通分量，强连通分量**。

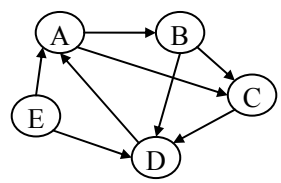
例：有 6 个顶点组成的无向图构成连通图，最少需要(_a_)条边；当边的数目大于(_b_)时，该图必定连通。
分析：a. 5。最少有 n-1 条边就可以构成连通图。 b. 10。考虑将 n 个顶点分成两组，一组有 n-1 个顶点，另一组只有 1 个顶点。首先在第一组中添加边，直到 n-1 个顶点构成全连通子图，共(n-1)(n-2)/2 条边，此后若再在图中任意添加一条边将必定连通两组顶点，从而构成连通图。
思考：对有向图有何结论。

2. 图的存储结构

(1) 图的存储结构

常见图的存储结构有：**邻接矩阵，邻接表，逆邻接表，十字链表，邻接多重表**。
邻接多重表只适用于存储无向图，其他存储结构可以存储无向图和有向图。

例：画出图的邻接矩阵、邻接表、逆邻接表和十字链表。



(2) 邻接矩阵

简言之，“数组(顶点)+二维数组(弧)+个数”[17]。
const int MAX_VERTEX = 最大顶点个数;

[17] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

```
typedef struct Graph {    // 图
   VertexType                                    vexs[MAX_VERTEX];    // 顶点向量
   ArcType     arcs[MAX_VERTEX] [MAX_VERTEX];    // 邻接矩阵
   int         vexnum, arcnum;                   // 顶点和弧的个数
} Graph;
```

图：有边(弧)为 1；否则为 0。网：有边(弧)为权值；否则为∞。
存储空间个数为 n^2，与边的数目无关。
无向图的邻接矩阵是对称的。

A	0	1	1	0	0
B	0	0	1	1	0
C	0	0	0	1	0
D	1	0	0	0	0
E	1	0	0	1	0

(3) 邻接表

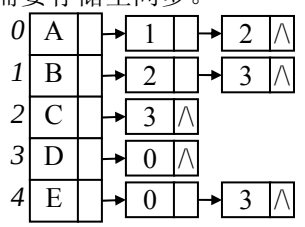
简言之，“数组(弧尾顶点)+链表(邻接点)+个数”[18]。

```
typedef struct ArcNode {    // 弧结点
   int         adjvex;                           // 邻接点
   struct ArcNode *nextarc;                      // 下一个邻接点
} ArcNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                    data;  // 顶点信息
   ArcNode *firstarc;                            // 第一个邻接点
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode     vexs[MAX_VERTEX];                 // 顶点向量
   int         vexnum, arcnum;                   // 顶点和弧的个数
} Graph;
```

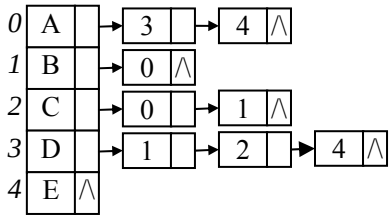
边(弧)多则需要存储空间多。



[18] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(4) 逆邻接表

简言之，“数组(弧头顶点)+链表(逆邻结点)+个数”[19]。
类型定义类似邻接表。



(5) 十字链表

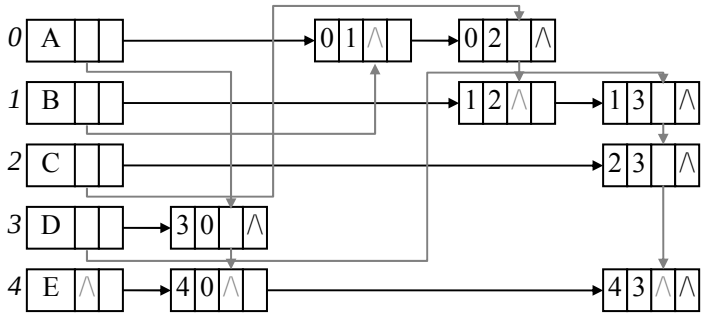
简言之，“数组(顶点)+弧结点(含头和尾)+个数”[20]。边可以看作两条弧。

```
typedef struct ArcNode {    // 弧结点
   int          vtail, vhead;                  // 弧尾和弧头顶点编号
   struct ArcNode   *nexttail, *nexthead;      // 指向同弧尾和同弧头的弧结点
} ArcNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                  data;  // 顶点信息
   ArcNode     *firstin, *firstout;            // 指向第一条入弧和第一条出弧
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode     vexs[MAX_VERTEX];               // 顶点向量
   int         vexnum, arcnum;                 // 顶点和弧的个数
} Graph;
```

弧结点中包含两个指针分别指向同一弧头的下一个弧和同一个弧尾的下一个弧。顶点结点则指向第一个同弧头和弧尾的弧。十字链表相当于邻接表和逆邻接表的结合。



[19] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。
[20] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

技巧：把弧结点按行排整齐，然后画链表。同弧尾的弧组成链表，同弧头的弧组成链表。

(6) 邻接多重表

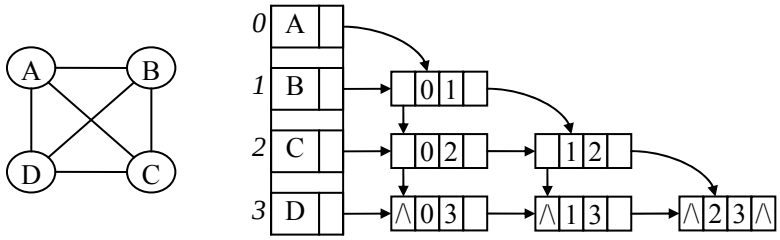
简言之，“数组(顶点)+边结点”[21]。

```
typedef struct EdgeNode {    // 边结点
   int          vexi, vexj;                    // 边的两个顶点
   struct EdgeNode   *nexti, *nextj;           // 两个顶点所依附的下一条边
} EdgeNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                  data;  // 顶点信息
   EdgeNode   *firstedge;                      // 指向第一条边
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode    vexs[MAX_VERTEX];                // 顶点向量
   int        vexnum, edgenum;                 // 顶点和边的个数
} Graph;
```

只适合存储无向图，不能存储有向图。



技巧：把边结点按列排整齐，然后画链表。相同顶点组成链表，这里没有起点和终点的区别。

3. 图的遍历

(1) 深度优先搜索

1°. 遍历方法

从图中某个顶点出发，访问此顶点，然后依次从其未被访问的邻接点出发深度优先遍历图；若图中尚有顶点未被访问，则另选图中一个未被访问的顶点作为起始点，重复

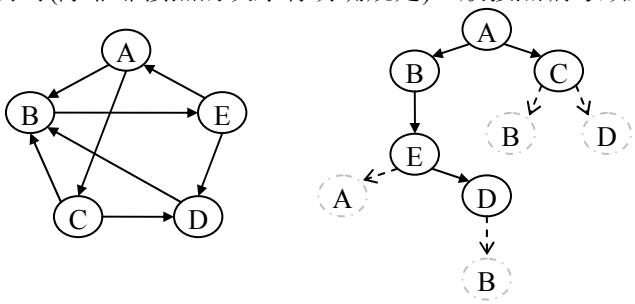
[21] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

上述过程，直到图中所有顶点都被访问为止。

2°. 分析方法

方法：画一棵“深度优先搜索树”。

例：下图从 A 出发深度优先搜索的结果是：ABEDC。
分析：画“深度优先搜索树”。从 A 出发，访问 A(画圈作标记)，A 的邻接点有 B 和 C(作为 A 的孩子)，B 未访问，访问 B(画圈)，B 的邻接点有 E(作 B 的孩子)，...，以此类推，画出搜索树。深度优先搜索的过程就是沿着该搜索树先根遍历的过程。
技巧：顶点的邻接点是无所谓次序的，所以同一个图的深度优先遍历序列可能不同，但在遍历时(除非邻接点的次序有明确规定)一般按照编号顺序安排邻接点的次序。



3°. 算法

课本上的算法稍加改动：

```
void DFSTraverse ( Graph G )
{
  visited [0 .. G.vexnum-1] = false;          // 初始化访问标志为未访问(false)
  for ( v=0; v<G.vexnum; v++ )
    if ( ! visited[v] )   DFS ( G, v );       // 从未被访问的顶点开始 DFS
}

void DFS ( Graph G, int v )
{
  visit ( v );   visited [v] = true;          // 访问顶点 v 并作标记
  for ( w=FirstAdjVex(G,v); w>=0; w=NextAdjVex(G,v,w) )
    if ( ! visited[w] )   DFS ( G, w );       // 分别从每个未访问的邻接点开始 DFS
}
```

其中的 FirstAdjVex(G,v)表示图 G 中顶点 v 的第一个邻接点，NextAdjVex(G,v,w)表示图 G 中顶点 v 的邻接点 w 之后 v 的下一个邻接点。
深度优先搜索算法有广泛的应用，以上算法是这些应用的基础。

(2) 广度优先搜索

1°. 遍历方法

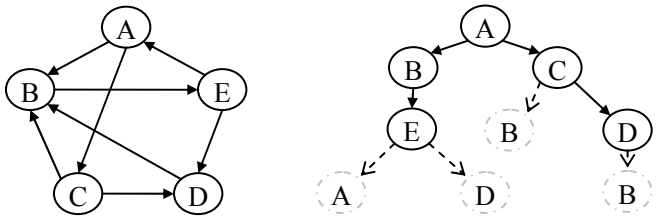
从图中某顶点出发，访问此顶点之后依次访问其各个未被访问的邻接点，然后从这些

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”要先于“后被访问的顶点的邻接点”被访问，直至所有已被访问的顶点的邻接点都被访问。若图中尚有顶点未被访问，则另选图中未被访问的顶点作为起始点，重复以上过程，直到图中所有顶点都被访问为止。
广度优先搜索从某顶点出发，要依次访问路径长度为 1，2，… 的顶点。

2°. 分析方法

方法：画一棵“广度优先搜索树”。

例：下图从 A 出发广度优先遍历的结果是：ABCED。
分析：画“广度优先搜索树”。与深度优先搜索树类似，A 为根，其邻接点为其孩子，访问一个顶点，则扩展出其孩子。不过广度优先搜索的访问次序是对该树按层遍历的结果。



3°. 算法

利用队列(类似按层遍历二叉树)。

```
void BFSTraverse ( Graph G )
{
   visited [0 .. G.vexnum-1] = false;          // 初始化访问标志为未访问(false)
   InitQueue ( Q );
   for ( v=0; v<G.vexnum; v++ )
      if ( ! visited[v] )    {
         // 从 v 出发广度优先搜索
         visit ( v );    visited [v] = true;
         EnQueue ( Q, v );
         while ( ! QueueEmpty(Q) )    {
            DeQueue ( Q, u );
            for ( w=FirstAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w) )
               if ( ! visited[w] )    {
                  visit ( w );    visited [w] = true;
                  EnQueue ( Q, w );
               }
         }
      }
}
```

(3) 时间复杂度分析

观察搜索树可以看出，无论是深度优先搜索还是广度优先搜索，其搜索过程就是对每

11. 各种排序方法比较

表 10.2 排序方法的比较

排序方法	时间复杂性			空间复杂性	稳定	特点
	平均	最好	最坏			
简单插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	是	元素少或基本有序时高效
希尔排序	$O(n^{3/2})$			O(1)	否	
冒泡排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	是	
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	否	平均时间性能最好
简单选择排序	$O(n^2)$			O(1)	否	比较次数最多
堆排序	O(nlogn)			O(1)	否	辅助空间少
归并排序	O(nlogn)			O(n)	是	稳定的
基数排序	O(d(n+rd))			O(rd)	是	适合个数多关键字较小

基于关键字比较的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。

二、习题

10.1 对待排序列(24, 86, 48, 56, 72, 36)进行 1) 直接插入排序，2) 希尔排序，3) 起泡排序，4) 快速排序，5) 简单选择排序，6) 堆排序，7) 归并排序，8) 链式基数排序。

10.2 证明：借助比较进行的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。

附录A: 习题解答

1.1 编写冒泡排序算法，使结果从大到小排列。

```
void BubbleSortDec ( DataType a[], int n )
{
  for ( i=0; i<n-1; i++ ) {
     change = false;
     for( j=0; j<n-i-1; j++ )
        if ( a[j]<a[j+1] ) {          // 逆序
           swap( a[j], a[j+1] );
           change = true;
        }
     if ( not change ) break;        // 没有交换则完成排序
  }
}
```

1.2 计算下面语句段中指定语句的频度：

```
1)  for ( i=1; i<=n; i++ )
        for ( j=i; j<=n; j++ )
            x++;  // @
2)  i = 1;
    while ( i<=n )
        i = i*2;  // @
```

分析：计算语句频度是计算时间复杂度的基础。可以用观察和归纳进行简单的计算。

1)

问题规模 n	执行次数
1	1
2	2+1
3	3+2+1
...	...
n	n+...+2+1=n(n+1)/2

结论：语句频度为 n(n+1)/2。

2)

问题规模 n	执行次数
1	1
2	2
3	2
4	3
...	...
2^k	k+1

结论：语句频度为$\lfloor \log n \rfloor + 1$。

2.1 将顺序表中的元素反转顺序。

```
void Reverse ( SqList& L )
```

```
{
   for ( i=0; i<L.length/2; i++)          // ㊀ < 还是 <= [33]
      L.elem[i]←→L.elem[L.length-i-1];
}
```

2.2 在非递减有序的顺序表中插入元素x，并保持有序。

思路 1：先查找适合插入的位置 i
　　向后移动元素（从后向前处理）
　　插入元素，表长加 1

思路 2：从后向前查找插入位置，同时向后移动大于 x 的元素
　　插入元素，表长加 1

注意：表满时不能插入。

```
// 顺序表结构
const int MAXSIZE = 1024;
typedef struct {
      DataType elem[MAXSIZE];
      int length;
} SqList;

// 向非递减有序的顺序表 L 中插入元素 x，仍保持非递减有序
// 插入成功返回 true，否则返回 false
bool OrderListInsert ( SqList &L, DataType x )
{
   if ( L.length==MAXSIZE )   return false;   // 表满，插入失败
   // 将大于 x 的元素后移
   for ( i=L.length-1; i>=0 && L.elem[i]>x; i--)
      L.elem[i+1] = L.elem[i];
   // 插入 x (因为最后执行 i--，故应在 i+1 处)
   L.elem[i+1] = x;
   L.length++;
   return true;
}
```

2.3 删除顺序表中所有等于x的元素。

```
void Remove ( SqList &L, DataType x )
{
   i = 0;                              // 剩余元素个数，同时是下一个元素的插入位置
   for ( j=0; j<L.length; j++ )
      if ( L.elem[j]!=x ) {            // 复制不等于 x 的元素组成新表
         if ( i!=j )   L.elem[i] = L.elem[j];   // 当 i==j 时不必复制
         i++;
      }
   L.length = i;                       // 剩余元素个数
}
```

[33] 用边界值验证法说明对于偶数个元素的表会有何种情况。

本算法的时间复杂度为 O(n)；若改用反复调用查找和删除算法，时间复杂度会达到 $O(n^2)$。

2.4 编写算法实现顺序表元素唯一化(即使顺序表中重复的元素只保留一个)，给出算法的时间复杂度。
思路：设已经唯一化的序列是$(a_0, \ldots, a_{i-1})$，剩余序列是$(a_j, \ldots, a_n)$。所要做的就是在已经唯一化的序列 L.elem[0..i-1]中查找 L.elem[j]，如果未找到则复制到 L.elem[i]处。如此重复直到剩余序列为空。

```
void Unique ( SqList &L )
{
  if ( L.length<=1 )   return;          // 空表或只有一个元素的表已经唯一化了
  i = 1;                                // 开始 L.elem[0..0]是唯一化序列
  for ( j=1; j<L.length; j++ ) {
    // 在 L.elem[0..i-1]中查找 L.elem[j]
    for ( k=0; k<i; k++ )
      if ( L.elem[k]==L.elem[j] )   break;
    if ( k==i ) {                       // L.elem[j]未出现过，复制到 L.elem[i]处
      if ( j!=i )   L.elem[i] = L.elem[j];
      i++;
    }
  }
  L.length = i;   // 表长为 i
}
```

以上算法的时间复杂度为 $O(n^2)$。当然，可以反复将重复元素删除达到唯一化，时间复杂度仍是 $O(n^2)$，但是与以上算法相比要移动更多元素。

2.5 非递减有序的顺序表元素唯一化(参见习题 2.4)，要求算法的时间复杂度为O(n)。
分析：由于该表是有序的，相等的元素必然靠在一起，不必从头开始查找，所以算法的时间复杂度可以降低。
思路：类似习题 2.4，但是查找部分只要与L.elem[i-1]比较就可以了。

```
void Unique ( SqList &L )
{
  i = 0;   // 开始的唯一化序列为空(㊀对比习题 2.4 思考为什么不用i=1 开始 [34])
  for ( j=1; j<L.length; j++ )
    if ( L.elem[j]!=L.elem[i-1] ) {       // Note：写成 L.elem[j]!=L.elem[j-1]亦可
      if ( j!=i )   L.elem[i] = L.elem[j];
      i++;
    }
  L.length = i;   // 表长
}
```

[34] 原因是这里不能确定表是否为空，而习题 2.4 则用开始的 if 语句排除了空表的情况。事实上，习题 2.4 也可以仿照此处修改，请读者自己完成。

2.6 将单链表就地逆置，即不另外开辟结点空间，而将链表元素翻转顺序。
思路：从链上依次取下结点，按照逆序建表的方法(参见 2.3. (8) 2°. 节)重新建表。

```
void Reverse ( LinkList &L )
{
   p = L->next;                    // 原链表
   L->next = NULL;                 // 新表(空表)
   while ( p ) {
      // 从原链表中取下结点 s
      s = p;
      p = p->next;
      // 插入 L 新表表头
      s->next = L->next;
      L->next = s;
   }
}
```

2.7 采用插入法将单链表中的元素排序。

```
void InsertionSort ( LinkList &L )
{
   h = L->next;                    // 原链表
   L->next = NULL;                 // 新空表
   while ( h ) {
      // 从原链表中取下结点 s
      s = h;   h = h->next;
      // 在新表中查找插入位置
      p = L;
      while ( p->next && p->next->data<=s->data )
         p = p->next;
      // 在 p 之后插入 s
      s->next = p->next;
      p->next = s;
   }
}
```

2.8 采用选择法将单链表中的元素排序。

```
void SelectionSort ( LinkList &L )
{
   p = L;
   while ( p->next ) {
      // 选择最小(从 p->next 至表尾)
      q = p;                       // 最小元素的前驱 q
      s = p;
      while ( s->next ) {
         if ( s->next->data < q->next->data )   q = s;
         s = s->next;
      }
      m = q->next;                 // 找到最小 m
```

```
      // 最小元素 m 插入有序序列末尾(p 之后)
      if ( q!=p ) {
         q->next = m->next;    // 解下最小 m
         m->next = p->next;    // 插入 p 之后
         p->next = m;
      }
      p = p->next;                  // L->next 至 p 为有序序列
   }
}
```

2.9 将两个非递减有序的单链表归并成一个，仍并保持非递减有序。

```
// 将非递减有序的单链表 lb 合并入 la，保持非递减有序
// 结果 la 中含有两个链表中所有元素，lb 为空表
void Merge ( LinkList &la, LinkList &lb )
{
   p = la,   q = lb;
   while ( p->next and q->next ) {
      // 跳过 la 中较小的元素
      while ( p->next and (p->next->data <= q->next->data) )
         p = p->next;
      // 把 lb 中较小的元素插入 la 中
      while ( p->next and q->next and (q->next->data < p->next->data) ) {
         s = q->next;
         q->next = s->next;
         s->next = p->next;
         p->next = s;
         p = s;
      }
   }
   if ( lb->next ) {       // 表 lb 剩余部分插入 la 末尾
      p->next = lb->next;
      lb->next = NULL;
   }
}
```

3.1 元素 1,2,3,4 依次入栈，不可能的出栈序列有哪些？

分析：什么是不可能的出栈序列？如果后入栈的数(如 4)先出栈，则此前入栈元素(如 1,2,3)在栈中的顺序就确定了，它们的出栈顺序一定是逆序(如 3,2,1)，否则就是不可能的出栈序列(如 2,1,3)。

不可能的出栈序列有：4123，4132，4213，4231，4312，3412，3142，3124。其中后 3 种都含 312 这一不可能序列。

3.2 设循环队列Q少用一个元素区分队列空和队列满，MAXSIZE=5，Q.front=Q.rear=0，画出执行下列操作时队列空和队列满的状态。入队列a,b,c，出队列a,b,c，入队列d,e,f,g。[ra][r][][][]→[fa][b][r][][]→[fa][b][c][r][]→[][fb][c][r][]→[][][fc][r][]→ [][][][fr][] 队列空 → ... →[f][r][][fd][e]→[f][g][r][fd][e] 队列满。

3.3 编写算法利用栈将队列中的元素翻转顺序。
思路：先将队列中的元素入栈，然后从栈中取出重新入队列。

```
void Reverse ( SqQueue &Q )
{
  InitStack ( S );
  while ( ! QueueEmpty(Q) ) {
    DeQueue ( Q, x ); Push ( S, x );
  }
  while ( ! StackEmpty(S) ) {
    Pop ( S, x ); EnQueue ( Q, x );
  }
}
```

4.1 长度为n的串的子串最多有多少个？
思路：对子串长度归纳。
子串的长度是 0，1，2，...，n，对应子串的个数分别是 1(空串)，n，n-1，...，1，总起来就是 1+n+(n-1)+...+2+1=1+n(n+1)/2。

6.1 度为k的树中有n_1个度为 1 的结点，n_2个度为 2 的结点，…，n_k个度为k的结点，问该树中有多少个叶子结点。
分析：分别从结点个数和分支个数考虑。
设叶子个数为 n0，结点总数：$n = n_0+n_1+n_2+...+n_k$，分支数目：$n-1 = n_1+2\ n_2+...+k\ n_k$，于是得到叶子个数

$$n_0 = 1 + \sum_{i=1}^{k} (i-1) n_i$$

6.2 有n个叶子结点的完全二叉树的高度是多少?
分析：完全二叉树中度为 1 的结点至多有一个。
完全二叉树中的结点数 n+(n-1) ≤ N ≤ n + (n-1) + 1，即 2n-1 ≤ N ≤ 2n，二叉树的高度是

$$\lfloor \log(2n-1) \rfloor + 1 \le h \le \lfloor \log(2n) \rfloor + 1$$

于是，(1) 当 $n=2^k$ 时，$h = \lfloor \log n \rfloor + 1$，当没有度为 1 的结点时；$h = \lfloor \log n \rfloor + 2$，当有 1 个度为 1 的结点时。(2) 其他情况下，$h = \lfloor \log n \rfloor + 2$。

6.3 编写算法按照缩进形式打印二叉树。

```
void PrintBinaryTree ( BinTree bt, int indent )
{
  if ( ! bt )   return;
  for ( i=0; i<indent; i++ )   print ( "  ");   // 缩进
  print ( bt->data );
  PrintBinaryTree ( bt->lchild, indent+1 );
  PrintBinaryTree ( bt->rchild, indent+1 );
}
```

6.4 编写算法按照逆时针旋转 90 度的形式打印二叉树。

```
void PrintBinaryTree ( BinTree bt, int level )
{
   if ( ! bt )    return;
   PrintBinaryTree ( bt->rchild, level+1 );   // 旋转后先打印右子树
   for ( i=0; i<level; i++ )    print ( "    " );   // 缩进
   print ( bt->data );
   PrintBinaryTree ( bt->lchild, level+1 );
}
```

6.5 编写算法判断二叉树是否是完全二叉树。

分析：按层遍历完全二叉树，当遇到第一个空指针之后应该全都是空指针。

```
bool IsComplete ( BinTree bt )
{
   // 按层遍历至第一个空指针
   InitQueue ( Q );
   EnQueue ( Q, bt );
   while ( ! QueueEmpty(Q) ) {
      DeQueue ( Q, p );
      if ( p ) {
         EnQueue ( Q, p->lchild );
         EnQueue ( Q, p->rchild );
      } else
         break;   // 遇到第一个空指针时停止遍历
   }
   // 检查队列中剩余元素是否全部是空指针
   while ( ! QueueEmpty(Q) ) {
      DeQueue ( Q, p );
      if ( ! p )    return false;   // 不是完全二叉树
   }
   return true;   // 完全二叉树
}
```

6.6 编写算法求二叉树中给定结点的所有祖先。

分析：进行后序遍历时，栈中保存的是当前结点的所有祖先。所以，后序遍历二叉树，遇到该结点时，取出栈中的内容即是所有祖先。

```
// 求二叉树 bt 中结点 xptr 的所有祖先
vector Ancestors ( BinTree bt, BinTree xptr )
{
   stack s;    stack tag;
   p = bt;
   while ( p || ! s.empty() ) {
      if ( p ) {
         s.push ( p );    tag.push ( 1 );
         p = p->lchild;
      } else {
         p = s.pop();    f = tag.pop();
```

```
        if ( f==1 ) {
           s.push ( p );    tag.push ( 2 );
           p = p->rchild;
        } else {
           if ( p==xptr ) {
              v = s;   // 当前栈的内容就是 xptr 的所有祖先
              return v;
           }
           p = NULL;
        }
     }
  } // while
  return vector();    // return a null vector
}
```

注：这里为描述方便借助了 C++中的某些描述方式。

6.7 编写算法求二叉树中两个结点的最近的共同祖先。

思路：用后序遍历求出两者的所有祖先，依次比较。

```
// 求二叉树 bt 中两个结点 q 和 r 的最近的共同祖先
BinTree LastAncestor ( BinTree bt, BinTree q, BinTree r )
{
  stack sq, sr;
  stack s;    stack tag;
  // 求 q 和 r 的所有祖先
  p = bt;
  while ( p || ! s.empty() ) {
     if ( p ) {
        s.push ( p );    tag.push ( 1 );
        p = p->lchild;
     } else {
        p = s.pop();    f = tag.pop();
        if ( f==1 ) {
           s.push ( p );    tag.push ( 2 );
           p = p->rchild;
        } else {
           if ( p==q )    sq = s;        // q 的所有祖先
           if ( p==r )    sr = s;              // s 的所有祖先
           p = NULL;
        }
     }
  }
  // 先跳过不同层的祖先，然后依次比较同一层的祖先
  if ( sq.size()>sr.size() )    while ( sq.size()>sr.size() )    sq.pop();
  else    while ( sr.size()>sq.size() )    sr.pop();
  // 求 q 和 r 的最近的共同祖先
  while ( !sq.empty() and (sq.top()!=sr.top()) ) {    //寻找共同祖先
     sq.pop();    sr.pop();
```

24, 86, 48, 56, 72, 36
(86, 72, 48, 56, 24, 36) (建立堆)
(72, 56, 48, 36, 24), 86
(56, 36, 48, 24), 72, 86
(48, 36, 24), 56, 72, 86
(36, 24), 48, 56, 72, 86
(24), 36, 48, 56, 72, 86

7) 归并排序

(24),(86),(48),(56),(72),(36)
(24, 86), (48, 56), (36, 72)
(24, 48, 56, 86), (36, 72)
(24, 36, 48, 56, 72, 86)

8) 链式基数排序

(24, 86, 48, 56, 72, 36)
分配：
[0][1][2][3][4][5][6][7][8][9]
72 24 86 48
56
36
收集：72, 24, 86, 56, 36, 48
分配：
[0][1][2][3][4][5][6][7][8][9]
24 36 48 56 72 86
收集：(24, 36, 48, 56, 72, 86)

10.2 证明：借助比较进行的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。

分析：含有 n 个记录的序列排序可能的初始状态有 n!个，所以描述 n 个记录排序过程的判定树必有 n!个叶子，二叉树的高度 $h \geqslant \log_2(n!)+1$。该判定树上必定存在长度为 $\log_2(n!)$的路径。所以借助比较的排序方法在最坏情况下的所需要比较次数至少为 $\log_2(n!)$。时间复杂度为 $O(\log_2(n!)) = O(n\log n)$。