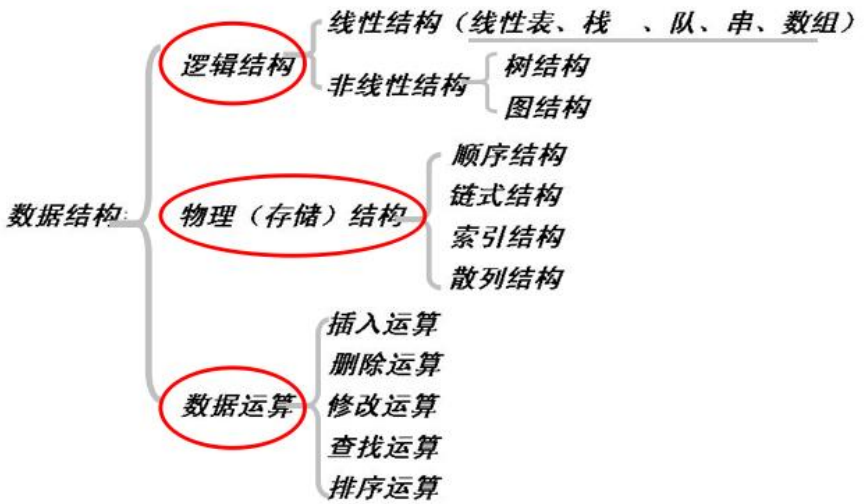
第1章　绪论

内容提要：

◆ 数据结构研究的内容。

针对非数值计算的程序设计问题，研究计算机的操作对象以及它们之间的关系和操作。

数据结构涵盖的内容：



◆ 基本概念：数据、数据元素、数据对象、数据结构、数据类型、抽象数据类型。

数据——所有能被计算机识别、存储和处理的符号的集合。

数据元素——是数据的基本单位，具有完整确定的实际意义。

数据对象——具有相同性质的数据元素的集合，是数据的一个子集。

数据结构——是相互之间存在一种或多种特定关系的数据元素的集合，表示为：

Data_Structure=（D, R）

数据类型——是一个值的集合和定义在该值上的一组操作的总称。

抽象数据类型——由用户定义的一个数学模型与定义在该模型上的一组操作，它由基本的数据类型构成。

◆ 算法的定义及五个特征。

算法——是对特定问题求解步骤的一种描述，它是指令的有限序列，是一系列输入转换为输出的计算步骤。

算法的基本特性：输入、输出、有穷性、确定性、可行性

◆ 算法设计要求。

①正确性、②可读性、③健壮性、④效率与低存储量需求

◆ 算法分析。

时间复杂度、空间复杂度、稳定性

学习重点：

◆ 数据结构的“三要素”：逻辑结构、物理（存储）结构及在这种结构上所定义的操作（运算）。

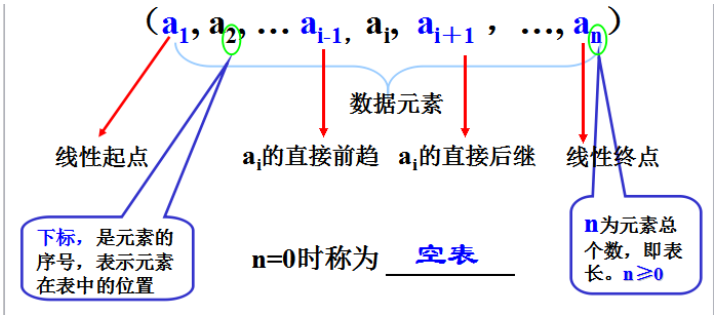
◆ 用计算语句频度来估算算法的时间复杂度。

第二章 线性表

内容提要：

◆ **线性表的逻辑结构定义，对线性表定义的操作。**

线性表的定义：用数据元素的有限序列表示



◆ **线性表的存储结构：顺序存储结构和链式存储结构。**

顺序存储定义：把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

链式存储结构: 其结点在存储器中的位置是随意的，即逻辑上相邻的数据元素在物理上不一定相邻。通过指针来实现！

◆ **线性表的操作在两种存储结构中的实现。**

数据结构的基本运算：修改、插入、删除、查找、排序

1) 修改——通过数组的下标便可访问某个特定元素并修改之。

核心语句: V[i]=x;

顺序表修改操作的时间效率是 O(1)

2) 插入——在线性表的第 i 个位置前插入一个元素

实现步骤：

①将第 n 至第 i 位的元素向后移动一个位置；

②将要插入的元素写到第 i 个位置；

③表长加 1。

注意：事先应判断: 插入位置 i 是否合法?表是否已满?

应当符合条件： 1≤i≤n+1 或 i=[1, n+1]

核心语句：

```
for (j=n; j>=i; j--)
    a[j+1]=a[ j ];
a[ i ]=x;
n++;
```

插入时的平均移动次数为：n(n+1)/2÷（n+1）=n/2≈O(n)

3) 删除——删除线性表的第 i 个位置上的元素

实现步骤：

①将第 i+1 至第 n 位的元素向前移动一个位置；

②表长减 1。

注意：事先需要判断，删除位置 i 是否合法?

应当符合条件：1≤i≤n 或 i=[1, n]

核心语句：

```
for ( j=i+1; j<=n; j++ )
    a[j-1]=a[j];
```

n--;

顺序表删除一元素的时间效率为:T（n)=(n-1)/2 ≈O(n)

顺序表插入、删除算法的平均空间复杂度为 O(1)

单链表：

（1）

用单链表结构来存放 26 个英文字母组成的线性表（a，b，c，…，z）,请写出 C 语言程序。

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node{
char data;
struct node *next;
}node;
node *p,*q,*head;              //一般需要 3 个指针变量
int n ;                        //  数据元素的个数
int m=sizeof(node);            /*结构类型定义好之后，每个 node 类型的长度就固定了，
                                 m 求一次即可*/
void build( )                  //字母链表的生成。要一个个慢慢链入
{
    int i;
    head=(node*)malloc(m);          //m=sizeof(node)  前面已求出
    p=head;
    for( i=1; i<26; i++)            //因尾结点要特殊处理，故 i≠26
    {
        p->data=i+‘a’-1;            //  第一个结点值为字符 a
        p->next=(node*)malloc(m);    //为后继结点“挖坑”!
        p=p->next；}                  //让指针变量 P 指向后一个结点
        p->data=i+‘a’-1;             //最后一个元素要单独处理
        p->next=NULL ;               //单链表尾结点的指针域要置空!
    }
}
void display()              //字母链表的输出
{
   p=head;
   while (p)                //当指针不空时循环（仅限于无头结点的情况）
   {
        printf("%c",p->data);
        p=p->next;                   //让指针不断“顺藤摸瓜”
   }
}
```

（2）单链表的修改(或读取）

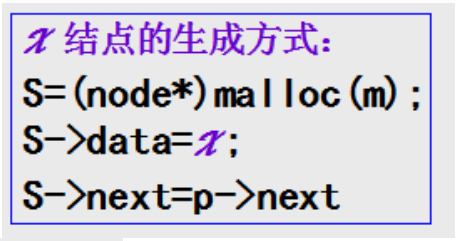
思路：要修改第 i 个数据元素，必须从头指针起一直找到该结点的指针 p，

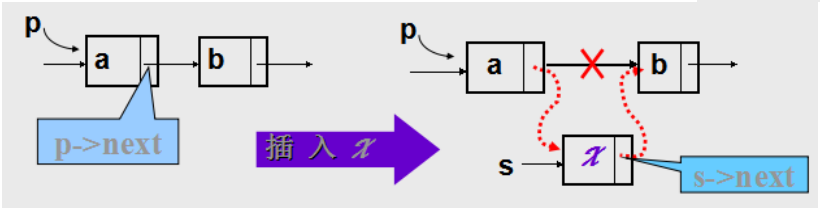
然后才能：p>data=new_value

读取第 i 个数据元素的核心语句是：

```
Linklist *find(Linklist *head ,int i)
{
    int j=1;
    Linklist *p;
    P=head->next;
    While((p!=NULL)&&(j<i))
    {
        p=p->next;
        j++;
     }
     return p;
}
```

3.单链表的插入



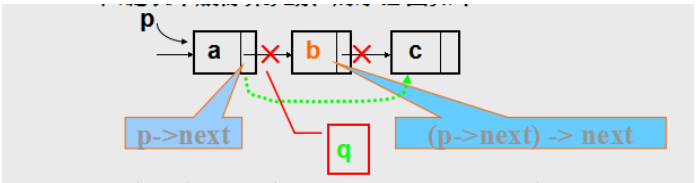


链表插入的核心语句：

Step 1：s->next=p->next;

Step 2：p->next=s ；

6.单链表的删除



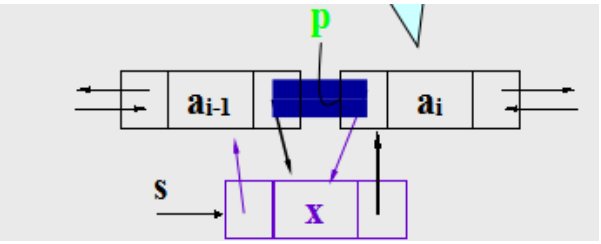
删除动作的核心语句（要借助辅助指针变量 q）：

q = p->next;　　//首先保存 b 的指针，靠它才能找到 c；

p->next=q->next;　//将 a、c 两结点相连，淘汰 b 结点；

free(q) ；　　//彻底释放 b 结点空间

7.双向链表的插入操作：



设 p 已指向第 i 元素，请在第 i 元素前插入元素 x：

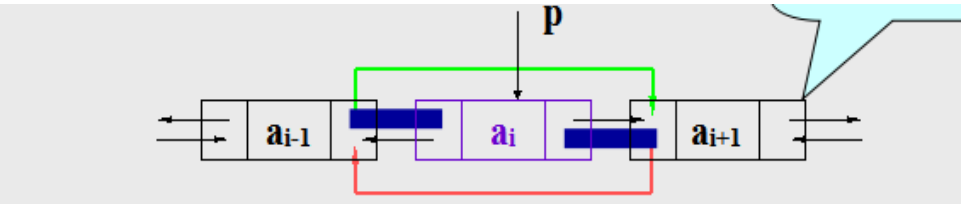
① ai-1 的后继从 ai (指针是 p)变为 x（指针是 s) :

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

② ai 的前驱从 ai-1 (指针是 p->prior)变为 x (指针是 s);

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

8.双向链表的删除操作：



设 p 指向第 i 个元素，删除第 i 个 元素

后继方向：ai-1 的后继由 ai (指针 p)变为 ai+1(指针 p ->next);

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

前驱方向：ai+1 的前驱由 ai (指针 p)变为 ai-1 (指针 p -> prior);

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

◆ 数组的逻辑结构定义及存储

数组： 由一组名字相同、下标不同的变量构成

N 维数组的特点：n 个下标，每个元素受到 n 个关系约束

一个 n 维数组可以看成是由若干个 n－1 维数组组成的线性表。

存储：事先约定按某种次序将数组元素排成一列序列，然后将这个线性序列存入存储器中。

在二维数组中，我们既可以规定按行存储，也 可以规定按列存储。

设一般的二维数组是 A[c1..d1, c2..d2]，

则行优先存储时的地址公式为：

$$LOC(a_{ij})=LOC(a_{c_1,\ c_2})+[(i-c_1)*(d_2-c_2+1)+j-c_2)]*L$$

$$A_{mn}=\begin{bmatrix} a_{c1,c2} & \cdots & a_{c1,d2} \\ \cdots & a_{ij} & \cdots \\ a_{d1,c2} & \cdots & a_{d1,d2} \end{bmatrix}$$

二维数组列优先存储的通式为：

$$LOC(a_{ij})=LOC(a_{c_1,\ c_2})+[(j-c_2)*(d_1-c_1+1)+i-c_1)]*L$$

◆ 稀疏矩阵（含特殊矩阵）的存储及运算。

稀疏矩阵：矩阵中非零元素的个数较少（一般小于 5%）

学习重点：

◆ 线性表的逻辑结构，指线性表的数据元素间存在着线性关系。在顺序存储结构中，元素存储的先后位置反映出这种线性关系，而在链式存储结构中，是靠指针来反映这种关系的。

◆ 顺序存储结构用一维数组表示，给定下标，可以存取相应元素，属于随机存取的存储结构。

◆ 链表操作中应注意不要使链意外“断开”。因此，若在某结点前插入一个元素，或删除某元素，必须知道该元素的前驱结点的指针。

◆ 掌握通过画出结点图来进行链表（单链表、循环链表等）的生成、插入、删除、遍历等操作。

◆ 数组（主要是二维）在以行序/列序为主的存储中的地址计算方法。

◆ 稀疏矩阵的三元组表存储结构。
◆ 稀疏矩阵的十字链表存储方法。

补充重点：

1.每个存储结点都包含两部分：数据域和指针域(链域)

2.在单链表中，除了首元结点外，任一结点的存储位置由 其直接前驱结点的链域的值 指示。

3.在链表中设置头结点有什么好处？

头结点即在链表的首元结点之前附设的一个结点，该结点的数据域可以为空，也可存放表长度等附加信息，其作用是为了对链表进行操作时，可以对空表、非空表的情况以及对首元结点进行统一处理，编程更方便。

4.如何表示空表？

（1）无头结点时，当头指针的值为空时表示空表；

（2）有头结点时，当头结点的指针域为空时表示空表。

5.链表的数据元素有两个域，不再是简单数据类型，编程时该如何表示？

因每个结点至少有两个分量，且数据类型通常不一致，所以要采用结构数据类型。

6.sizeof(x)—— 计算变量 x 的长度（字节数）；

malloc(m) — 开辟 m 字节长度的地址空间，并返回这段空间的首地址；

free(p) —— 释放指针 p 所指变量的存储空间，即彻底删除一个变量。

7.链表的运算效率分析：

（1）查找

因线性链表只能顺序存取，即在查找时要从头指针找起，查找的时间复杂度为 O(n)。

（2） 插入和删除

因线性链表不需要移动元素，只要修改指针，一般情况下时间复杂度为 O(1)。

但是，如果要在单链表中进行前插或删除操作，因为要从头查找前驱结点，所耗时间复杂度将是 O(n)。

例：在 n 个结点的单链表中要删除已知结点*P，需找到它的前驱结点的地址，其时间复杂度为 O（n）

8. 顺序存储和链式存储的区别和优缺点？

顺序存储时，逻辑上相邻的数据元素，其物理存放地址也相邻。**顺序存储的优点是存储密度大，存储空间利用率高；缺点是插入或删除元素时不方便。**

链式存储时，相邻数据元素可随意存放，但所占存储空间分两部分，一部分存放结点值，

另一部分存放表示结点间关系的指针。**链式存储的优点是插入或删除元素时很方便，使用灵活。缺点是存储密度小，存储空间利用率低。**

- 顺序表适宜于做查找这样的静态操作；
- 链表宜于做插入、删除这样的动态操作。
- 若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；
- 若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。

9. 判断：“数组的处理比其它复杂的结构要简单”，对吗？

答：对的。因为——

① 数组中各元素具有统一的类型；

② 数组元素的下标一般具有固定的上界和下界，即数组一旦被定义，它的维数和维界就不再改变。

③数组的基本操作比较简单，除了结构的初始化和销毁之外，只有存取元素和修改元素值的操作。

10.三元素组表中的每个结点对应于稀疏矩阵的一个非零元素，它包含有三个数据项，分别表示该元素的 行下标 、列下标 和 元素值 。

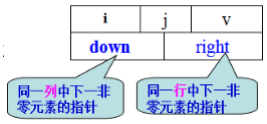
11.写出右图所示稀疏矩阵的压缩存储形式。

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

解：介绍 3 种存储形式。

法 1：用线性表表示：

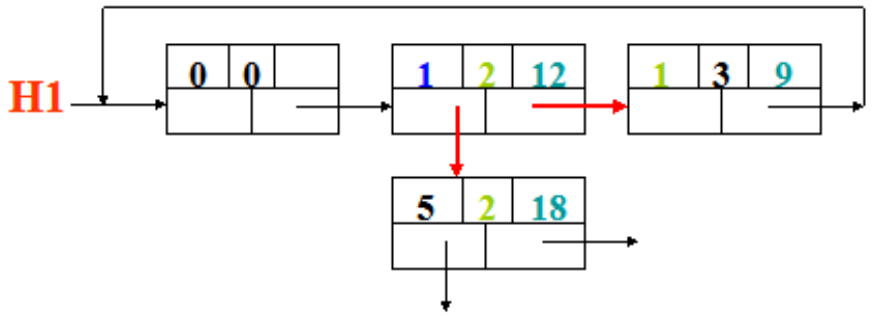
（(1,2,12) ，(1,3,9)， (3,1,-3)， (3,5,14)， (4,3,24)， (5,2,18) ，(6,1,15)， (6,4,-7)）



法 2：用十字链表表示

用途：方便稀疏矩阵的加减运算

方法：每个非 0 元素占用 5 个域



法 3：用三元组矩阵表示：

	i	j	value
0	6	6	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	5	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

稀疏矩阵压缩存储的缺点：将失去随机存取功能

代码：

1.用数组 V 来存放 26 个英文字母组成的线性表（a，b，c，…，z），写出在顺序结构上生成和显示该表的 C 语言程序。

```
char V[30];
void build()        //字母线性表的生成，即建表操作
{
      int i;
      V[0]='a';
      for( i=1;     i<=n-1;     i++ )
      V[i]=V[i-1]+1;
}
void display( ) //字母线性表的显示，即读表操作
{
      int i;
      for( i=0;    i<=n-1;    i++ )
      printf( "%c",    v[i] );
      printf( "\n " );
}

void main(void)        //主函数，字母线性表的生成和输出
{
       n=26; // n 是表长，是数据元素的个数，而不是 V 的实际下标
       build( );
      display( );
}
```

第三章 栈和队列

内容提要：

◆ 从数据结构角度来讲，栈和队列也是线性表，其操作是线性表操作的子集，属操作受限的线性表。但从数据类型的角度看，它们是和线性表大不相同的重要抽象数据类型。

◆ 栈的定义及操作。栈是只准在一端进行插入和删除操作的线性表，该端称为栈的顶端。

插入元素到栈顶的操作，称为入栈。
从栈顶删除最后一个元素的操作，称为出栈。
对于向上生成的堆栈:
入栈口诀：堆栈指针 top “先压后加”： S[top++]=an+1
出栈口诀：堆栈指针 top “先减后弹”： e=S[--top]

◆ **栈的顺序和链式存储结构，及在这两种结构下实现栈的操作。**

顺序栈入栈函数 PUSH（）

```
status Push(ElemType e)
{ if(top>M){上溢}
   else s[top++]=e;
}
```

顺序栈出栈函数 POP()

```
status Pop( )
{ if(top=L) { 下溢 }
else { e=s[--top];        return(e);}
 }
```

◆ **队列的定义及操作，队列的删除在一端（队尾），而插入则在队列的另一端（队头）。因此在两种存储结构中，都需要队头和队尾两个指针。**

队列：只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表。
链队列
结点类型定义：

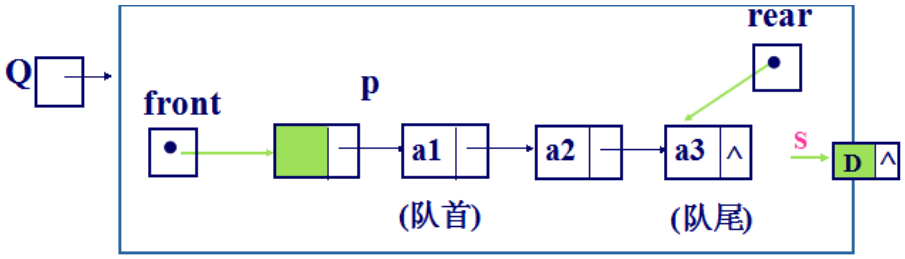
```
typedef Struct QNode{
```

```
        QElemType              data;        //元素
        Struct     QNode     *next;    //指向下一结点的指针
    }Qnode , * QueuePtr ;
```

链队列类型定义：

```
 typedef   struct {
   QueuePtr        front ; //队首指针
   QueuePtr        rear ; //队尾指针
   }   LinkQueue;
```

链队示意图：



① 空链队的特征：front=rear

② 链队会满吗？一般不会，因为删除时有 free 动作。除非内存不足！

③ 入队（尾部插入）：rear->next=S; rear=S;

出队（头部删除）：front->next=p->next;

2.顺序队

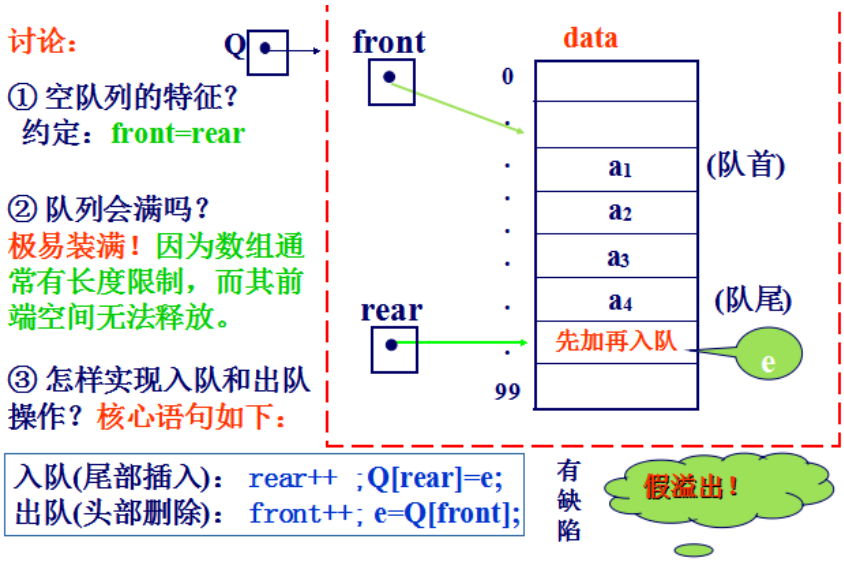
顺序队类型定义：

```
#define       QUEUE-MAXSIZE    100    //最大队列长度
        typedef       struct {
                    QElemType         *base;       //队列的基址
                      int                              front;        //队首指针
                      int                               rear;       //队尾指针
            }SqQueue
```

建队核心语句：

```
q . base=(QElemType *)malloc(sizeof (QElemType  ）
* QUEUE_MAXSIZE;              //分配空间
```

顺序队示意图：



循环队列：

队空条件 ： front = rear (初始化时：front = rear)

队满条件： front = (rear+1) % N (N=maxsize)

队列长度（即数据元素个数）：L=（N＋rear－front）% N

1）初始化一个空队列

```
Status    InitQueue ( SqQueue    &q ) //初始化空循环队列 q
{
    q . base=(QElemType *)malloc(sizeof(QElemType）
    * QUEUE_MAXSIZE);    //分配空间
    if (!q.base)  exit(OVERFLOW);//内存分配失败，退出程序
        q.front =q.rear=0; //置空队列
        return OK;
 } //InitQueue;
```

2）入队操作

```
Status    EnQueue(SqQueue  &q,    QElemType e)
{//向循环队列 q 的队尾加入一个元素 e
   if (    (q.rear+1) %  QUEUE_MAXSIZE = =  q.front  )
                    return  ERROR ; //队满则上溢，无法再入队
        q.rear = ( q . rear + 1 ) %  QUEUE_MAXSIZE;
    q.base [ q.rear ] = e;     //新元素 e 入队
      return  OK;
 }// EnQueue;
```

3）出队操作

```
Status      DeQueue ( SqQueue  &q,    QElemType &e)
 {//若队列不空，删除循环队列 q 的队头元素，
        //由 e 返回其值，并返回 OK
      if ( q.front = = q.rear )    return ERROR;//队列空
      q.front=(q.front+1) % QUEUE_MAXSIZE ;
```

```
    e = q.base [ q.front ] ;
  return OK;
}// DeQueue
```

◆ 链队列空的条件是首尾指针相等，而循环队列满的条件的判定，则有队尾加 1 等于队头和设标记两种方法。

补充重点：

1.为什么要设计堆栈？它有什么独特用途？

① 调用函数或子程序非它莫属；
② 递归运算的有力工具；
③ 用于保护现场和恢复现场；
④ 简化了程序设计的问题。

2.为什么要设计队列？它有什么独特用途？

① 离散事件的模拟（模拟事件发生的先后顺序,例如 CPU 芯片中的指令译码队列）；
② 操作系统中的作业调度（一个 CPU 执行多个作业）；
③ 简化程序设计。

3.什么叫“假溢出” ？如何解决？

答：在顺序队中，当尾指针已经到了数组的上界，不能再有入队操作，但其实数组中还有空位置，这就叫“假溢出”。解决假溢出的途径———采用循环队列。

4.在一个循环队列中，若约定队首指针指向队首元素的前一个位置。那么，从循环队列中删除一个元素时，其操作是先 移动队首位置 ，后 取出元素。

5.线性表、栈、队的异同点：

相同点：逻辑结构相同，都是线性的；都可以用顺序存储或链表存储；栈和队列是两种特殊的线性表，即受限的线性表（只是对插入、删除运算加以限制）。

不同点：① 运算规则不同：

线性表为随机存取；

而栈是只允许在一端进行插入和删除运算，因而是后进先出表 LIFO；

队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表 FIFO。

② 用途不同，线性表比较通用；堆栈用于函数调用、递归和简化设计等；队列用于离散事件模拟、OS 作业调度和简化设计等。

第四章 串

内容提要：

◆ 串是数据元素为字符的线性表，串的定义及操作。

串即字符串，是由零个或多个字符组成的有限序列，是数据元素为单个字符的特殊线性表。

串比较：int strcmp(char *s1,char *s2);

求串长：int strlen(char *s);

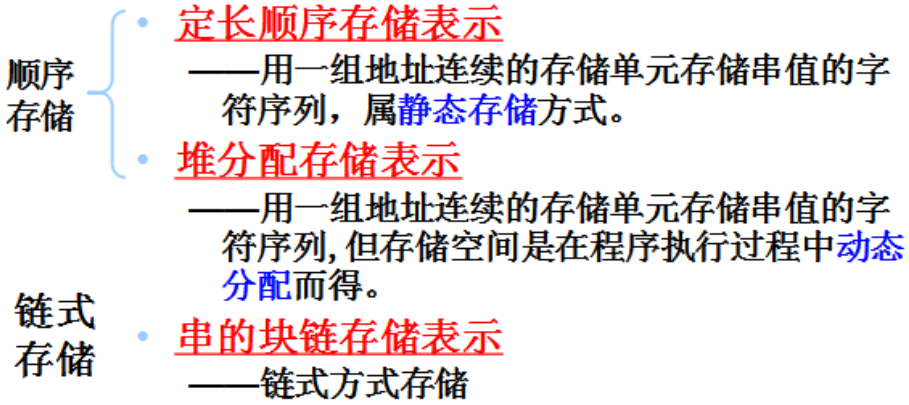
串连接：char strcat(char *to,char *from)

子串 T 定位：char strchr(char *s,char *c);

◆ 串的存储结构，因串是数据元素为字符的线性表，所以存在“结点大小”的问题。

模式匹配算法 。

串有三种机内表示方法：



模式匹配算法：

算法目的：确定主串中所含子串第一次出现的位置（定位）

定位问题称为串的模式匹配，典型函数为 Index(S,T,pos)

BF 算法的实现—即编写 Index(S, T, pos)函数

BF 算法设计思想：

将主串 S 的第 pos 个字符和模式 T 的第 1 个字符比较，

若相等，继续逐个比较后续字符；

若不等，从主串 S 的下一字符（pos+1）起，重新与 T 第一个字符比较。

直到主串 S 的一个连续子串字符序列与模式 T 相等。返回值为 S 中与 T 匹配的子序列第一个字符的序号，即匹配成功。

否则，匹配失败，返回值 0。

```
Int Index_BP(SString S, SString T, int pos)
{ //返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在，则函数值为 0.
 // 其中，T 非空，1≤pos≤StrLength(S)
     i=pos;          j=1;
    while ( i<=S[0] && j<=T[0] ) //如果 i,j 二指针在正常长度范围，
      {
        if (S[i] = = T[j] ) {++i, ++j; }     //则继续比较后续字符
        else {i=i-j+2; j=1;} //若不相等，指针后退重新开始匹配
       }
  if(j>T[0]) return i-T[0];   //T 子串指针 j 正常到尾，说明匹配成功，    else return 0;              //否则属于 i>S[0]情况，i 先到尾就不正常
} //Index_BP
```

补充重点：

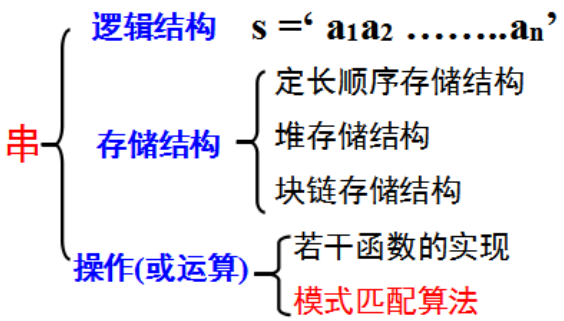
1.空串和空白串有无区别？

答：有区别。

空串(Null String)是指长度为零的串；

而空白串(Blank　String),是指包含一个或多个空白字符‘　’(空格键)的字符串.

2.“空串是任意串的子串；任意串 S 都是 S 本身的子串，除 S 本身外，S 的其他子串称为 S 的真子串。”



模式匹配即子串定位运算，即如何实现 Index(S,T,pos)函数

第六章 树和二叉树

内容提要：

◆ 树是复杂的非线性数据结构，树，二叉树的递归定义，基本概念，术语。

树：由一个或多个(n≥0)结点组成的有限集合 T，有且仅有一个结点称为根（root），当 n>1 时，其余的结点分为 m(m≥0)个互不相交的有限集合 T1,T2，…，Tm。每个集合本身又是棵树，被称作这个根的子树 。

二叉树：是 n（n≥0）个结点的有限集合，由一个根结点以及两棵互不相交的、分别称为左子树和右子树的二叉树组成。

术语：P88

◆ 二叉树的性质，存储结构。

性质 1: 在二叉树的第 i 层上至多有 2i-1 个结点（i>0）。

性质 2: 深度为 k 的二叉树至多有 2k-1 个结点（k>0）。

性质 3: 对于任何一棵二叉树，若 2 度的结点数有 n2 个，则叶子数（n0）必定为 n2＋1

性质 4: 具有 n 个结点的完全二叉树的深度必为 ·

性质 5: 对完全二叉树，若从上至下、从左至右编号，则编号为 i 的结点，其左孩子编号必为 2i，其右孩子编号为 2i＋1；其双亲的编号必为 i/2（i＝1 时为根,除外）。

二叉树的存储结构：

一、顺序存储结构

按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。

若是完全/满二叉树则可以做到唯一复原。

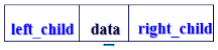
不是完全二叉树：一律转为完全二叉树！

方法很简单，将各层空缺处统统补上“虚结点”，其内容为空。

缺点：①浪费空间；②插入、删除不便

二、链式存储结构

用二叉链表即可方便表示。一般从根结点开始存储。



优点：①不浪费空间；②插入、删除方便

◆ 二叉树的遍历。

指按照某种次序访问二叉树的所有结点，并且每个结点仅访问一次，得到一个线性序列。

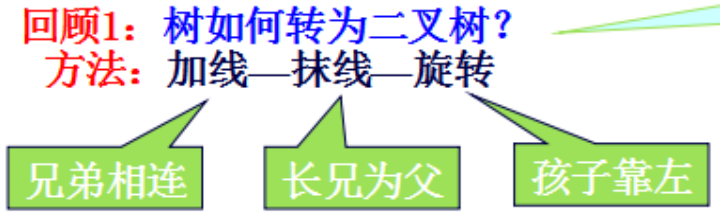
遍历规则———

二叉树由根、左子树、右子树构成，定义为 D、 L、R

若限定先左后右，则有三种实现方案：

DLR	LDR	LRD
先序遍历	中序遍历	后序遍历

◆ 树的存储结构，树、森林的遍历及和二叉树的相互转换。



回顾 2：二叉树怎样还原为树？

要点：逆操作，把所有右孩子变为兄弟！

讨论 1：森林如何转为二叉树？

法一：① 各森林先各自转为二叉树；② 依次连到前一个二叉树的右子树上。

法二：森林直接变兄弟，再转为二叉树

讨论 2：二叉树如何还原为森林？

要点：把最右边的子树变为森林，其余右子树变为兄弟

树和森林的存储方式：

树有三种常用存储方式：

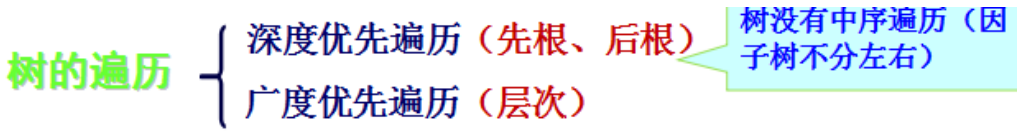
①双亲表示法　　②孩子表示法　　③孩子—兄弟表示法

问：树→二叉树的“连线—抹线—旋转” 如何由计算机自动实现？

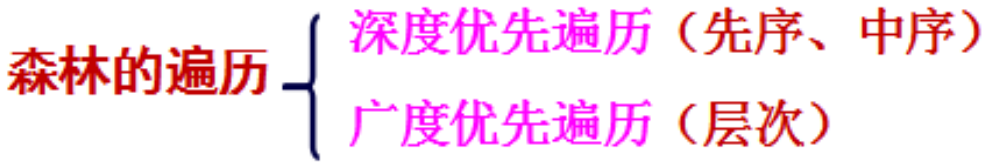
答：用“左孩子右兄弟”表示法来存储即可。

存储的过程就是树转换为二叉树的过程！

树、森林的遍历：



① 先根遍历：访问根结点；依次先根遍历根结点的每棵子树。
② 后根遍历：依次后根遍历根结点的每棵子树；访问根结点。
讨论：树若采用“先转换，后遍历”方式，结果是否一样？
1. 树的先根遍历与二叉树的先序遍历相同；
2. 树的后根遍历相当于二叉树的中序遍历；
3. 树没有中序遍历，因为子树无左右之分。



① 先序遍历
若森林为空，返回；
访问森林中第一棵树的根结点；
先根遍历第一棵树的根结点的子树森林；
先根遍历除去第一棵树之后剩余的树构成的森林。
② 中序遍历
若森林为空，返回；
中根遍历森林中第一棵树的根结点的子树森林；
访问第一棵树的根结点；
中根遍历除去第一棵树之后剩余的树构成的森林。

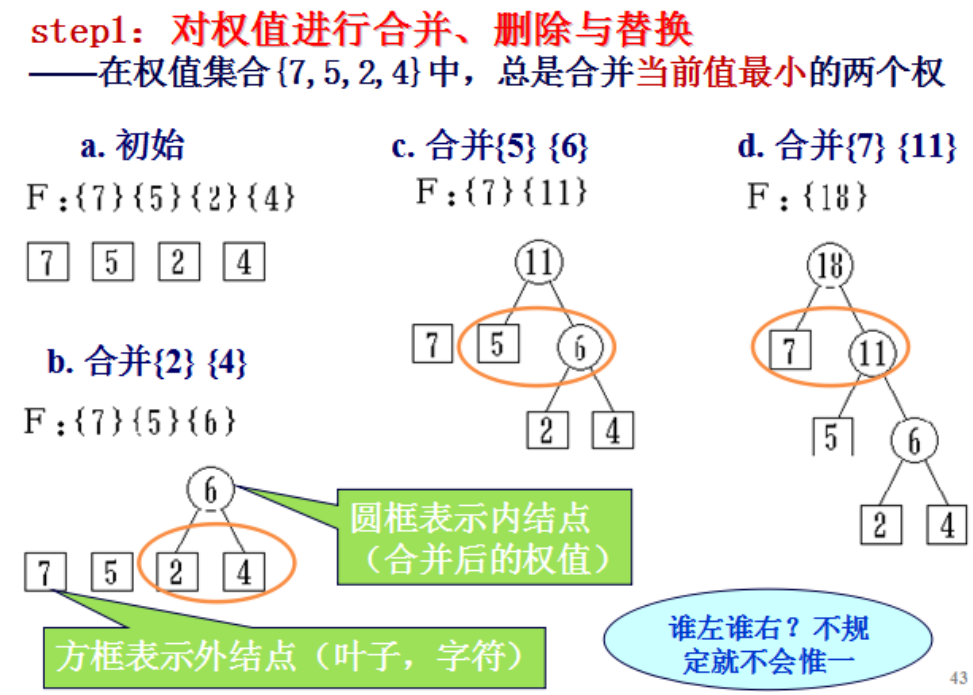
◆ 二叉树的应用：哈夫曼树和哈夫曼编码。

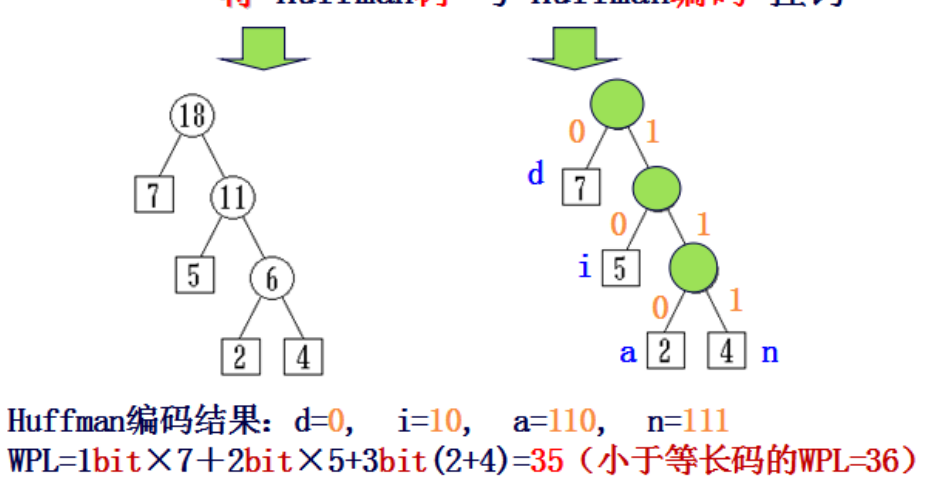
Huffman 树：最优二叉树（带权路径长度最短的树）
Huffman 编码：不等长编码。

树的带权路径长度：$WPL = \sum_{k=1}^{n} w_k l_k$ （树中所有叶子结点的带权路径长度之和）

构造 Huffman 树的基本思想：权值大的结点用短路径，权值小的结点用长路径。
构造 Huffman 树的步骤（即 Huffman 算法）：
(1) 由给定的 n 个权值{ w1, w2, …, wn }构成 n 棵二叉树的集合 F = { T1, T2, …, Tn } （即森林），其中每棵二叉树 Ti 中只有一个带权为 wi 的根结点，其左右子树均空。
(2) 在 F 中选取两棵根结点权值最小的树 做为左右子树构造一棵新的二叉树，且让新二叉树根结点的权值等于其左右子树的根结点权值之和。
(3) 在 F 中删去这两棵树，同时将新得到的二叉树加入 F 中。
(4) 重复(2) 和(3)，直到 F 只含一棵树为止。这棵树便是 Huffman 树。

具体操作步骤：





Huffman编码结果：d=0，　i=10，　a=110，　n=111
WPL=1bit×7＋2bit×5+3bit(2+4)=35（小于等长码的WPL=36）

学习重点：（本章内容是本课程的重点）

◆ 二叉树性质及证明方法，并能把这种方法推广到 K 叉树。

◆ 二叉树遍历，遍历是基础，由此导出许多实用的算法，如求二叉树的高度、各结点的层次数、度为 0、1、2 的结点数。

◆ 由二叉树遍历的前序和中序序列或后序和中序序列可以唯一构造一棵二叉树。由前序和后序序列不能唯一确定一棵二叉树。

◆ 完全二叉树的性质。

◆ 树、森林和二叉树间的相互转换。

◆ **哈夫曼树的定义、构造及求哈夫曼编码。**

补充：

1.满二叉树和完全二叉树有什么区别？

答：满二叉树是叶子一个也不少的树，而完全二叉树虽然前 k-1 层是满的，但最底层却允许在右边缺少连续若干个结点。满二叉树是完全二叉树的一个特例。

2.Huffman 树有什么用？

最小冗余编码、信息高效传输

第七章 图

内容提要：

◆ **图的定义，概念、术语及基本操作。**

图：记为　G＝(V, E)

其中：V 是 G 的顶点集合，是有穷非空集；

　　E 是 G 的边集合，是有穷集。

术语：见课件

◆ **图的存储结构。**

1.邻接矩阵(数组)表示法

① 建立一个顶点表和一个邻接矩阵

② 设图 A = (V, E) 有 n 个顶点，则图的邻接矩阵是一个二维数组 A.Edge[n][n]。

注：在有向图的邻接矩阵中，

第 i 行含义：以结点 vi 为尾的弧(即出度边)；

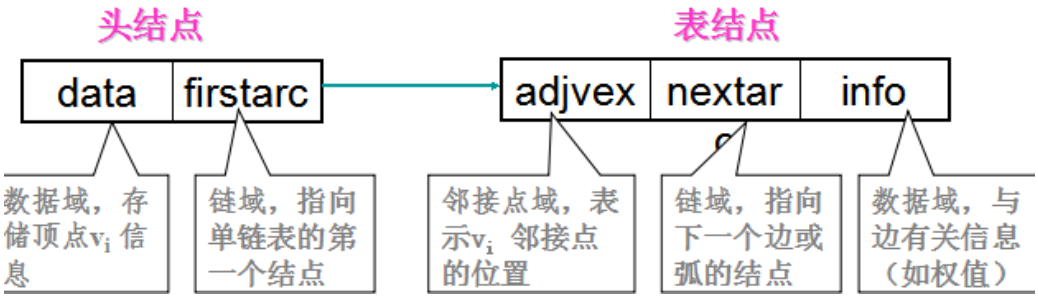
第 i 列含义：以结点 vi 为头的弧(即入度边)。

邻接矩阵法优点：容易实现图的操作，如：求某顶点的度、判断顶点之间是否有边（弧）、找顶点的邻接点等等。

邻接矩阵法缺点：n 个顶点需要 n*n 个单元存储边(弧);空间效率为 O(n2)。

2.邻接表(链式)表示法

① 对每个顶点 vi 建立一个单链表，把与 vi 有关联的边的信息（即度或出度边）链接起来，表中每个结点都设为 3 个域:



② 每个单链表还应当附设一个头结点（设为 2 个域），存 vi 信息；

③ 每个单链表的头结点另外用顺序存储结构存储。

邻接表的优点：空间效率高；容易寻找顶点的邻接点；

邻接表的缺点：判断两顶点间是否有边或弧，需搜索两结点对应的单链表，没有邻接矩阵方便。

◆ **图的遍历。**

遍历定义：从已给的连通图中某一顶点出发，沿着一些边，访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历，它是图的基本运算。

图常用的遍历：一、深度优先搜索；二、广度优先搜索

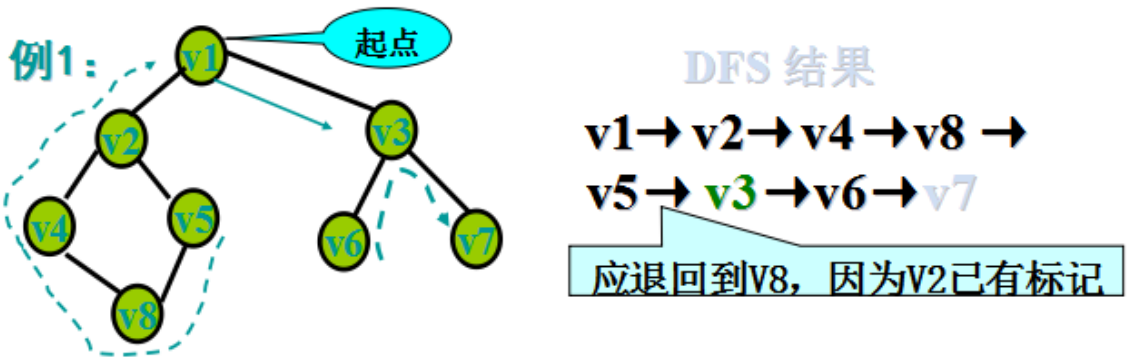
深度优先搜索（遍历）步骤：

① 访问起始点 v;

② 若 v 的第 1 个邻接点没访问过，深度遍历此邻接点；

③ 若当前邻接点已访问过，再找 v 的第 2 个邻接点重新遍历。

基本思想：——仿树的先序遍历过程。

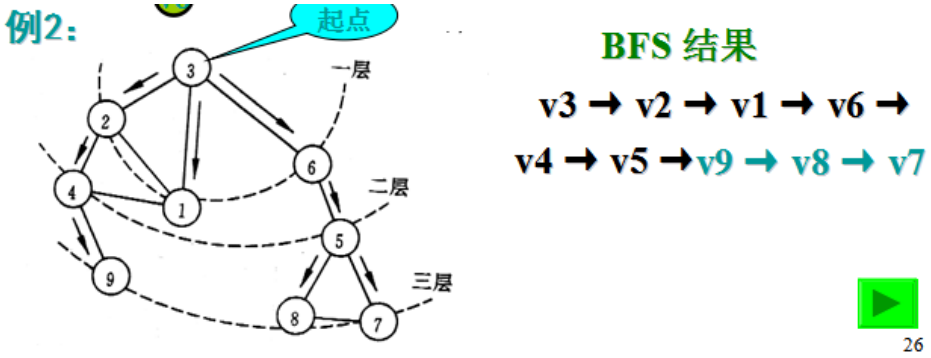


广度优先搜索（遍历）步骤：

① 在访问了起始点 v 之后，依次访问 v 的邻接点；

② 然后再依次（顺序）访问这些点（下一层）中未被访问过的邻接点；

③ 直到所有顶点都被访问过为止。



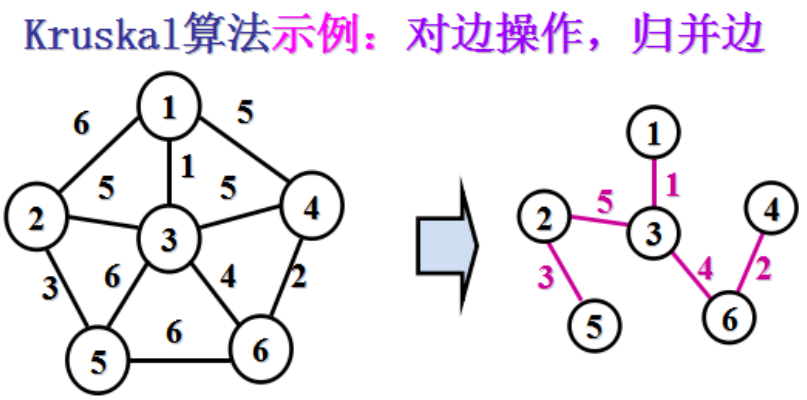
◆ **图的应用（最小生成树，最短路经）**

最小生成树（MST）的性质如下：若 U 集是 V 的一个非空子集，若(u0, v0)是一条最小权值的边，其中 u0 · U，v0 · V-U；则：(u0, v0)必在最小生成树上。

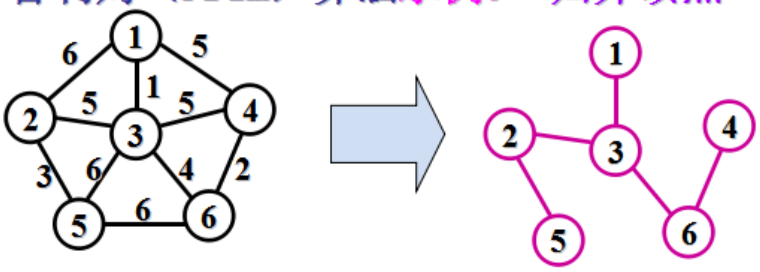
求 MST 最常用的是以下两种：Kruskal（克鲁斯卡尔）算法、Prim（普里姆）算法

Kruskal 算法特点：将边归并，适于求稀疏网的最小生成树。

Prime 算法特点: 将顶点归并，与边数无关，适于稠密网。



普利姆（Prim）算法示例： 归并顶点



在带权有向图中 A 点（源点）到达 B 点（终点）的多条路径中，寻找一条各边权值之和最小的路径，即最短路径。

两种常见的最短路径问题：

一、 单源最短路径—用 Dijkstra（迪杰斯特拉）算法

二、所有顶点间的最短路径—用 Floyd（弗洛伊德）算法

一、单源最短路径 (Dijkstra 算法)一顶点到其余各顶点（v0→j）

目的： 设一有向图 G=（V, E），已知各边的权值，以某指定点 v0 为源点，求从 v0 到图的其余各点的最短路径。限定各边上的权值大于或等于 0。

二、所有顶点之间的最短路径

可以通过调用 n 次 Dijkstra 算法来完成，还有更简单的一个算法：Floyd 算法（自学）。

学习重点： 图是应用最广泛的一种数据结构，本章也是这门课程的重点。

◆ 基本概念中，连通分量，生成树，邻接点是重点。

① **连通图：**在无向图中, 若从顶点 v1 到顶点 v2 有路径, 则称顶点 v1 与 v2 是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。
非连通图的极大连通子图叫做连通分量。

② **生成树：**是一个极小连通子图，它含有图中全部 n 个顶点，但只有 n-1 条边。

③ **邻接点：**若 (u, v) 是 E(G) 中的一条边，则称 u 与 v 互为邻接顶点。

◆ 图是复杂的数据结构，也有顺序和链式两种存储结构：数组表示法（重点是邻接距阵）和邻接表。这两种存储结构对有向图和无向图均适用

◆ 图的遍历是图的各种算法的基础，应熟练掌握图的深度、广度优先遍历。

◆ 连通图的最小生成树不是唯一的，但最小生成树边上的权值之和是唯一的。 应熟练掌握 prim 和 kruscal 算法，特别是手工分步模拟生成树的生成过程。

◆ 从单源点到其他顶点，以及各个顶点间的最短路径问题，掌握熟练手工模拟。

补充：

1.问：当有向图中仅 1 个顶点的入度为 0,其余顶点的入度均为 1，此时是何形状？

答：是树！而且是一棵有向树！

2.讨论：邻接表与邻接矩阵有什么异同之处？

1. 联系：邻接表中每个链表对应于邻接矩阵中的一行，
链表中结点个数等于一行中非零元素的个数。

2. 区别：

对于任一确定的无向图，邻接矩阵是唯一的（行列号与顶点编号一致），
但邻接表不唯一（链接次序与顶点编号无关）。

3. 用途：

邻接矩阵多用于稠密图的存储
而邻接表多用于稀疏图的存储

3.若对连通图进行遍历，得到的是生成树

若对非连通图进行遍历，得到的是生成森林。

第八章　　查找

内容提要：

◆ 查找表是称为集合的数据结构。是元素间约束力最差的数据结构：元素间的关系是元素仅共在同一个集合中。（同一类型的数据元素构成的集合）

◆ 查找表的操作：查找，插入，删除。

◆ 静态查找表：顺序表，有序表等。

针对静态查找表的查找算法主要有:顺序查找、折半查找、分块查找

一、顺序查找（线性查找）

技巧：把待查关键字 key 存入表头或表尾（俗称“哨兵”），这样可以加快执行速度。

```
int Search_Seq( SSTable    ST , KeyType    key ){
ST.elem[0].key =key;
for( i=ST.length; ST.elem[ i ].key!=key;    - - i    );
```

```
        return i;
    } // Search_Seq
```

//ASL＝（1＋n）/2，时间效率为 O(n)，这是查找成功的情况:

顺序查找的特点：

优点：算法简单，且对顺序结构或链表结构均适用。

缺点： ASL 太大，时间效率太低。

二、折半查找（二分或对分查找）

若关键字不在表中，怎样得知并及时停止查找？

典型标志是：当查找范围的上界≤下界时停止查找。

ASL 的含义是“平均每个数据的查找时间”，而前式是 n 个数据查找时间的总和，所以：

三、分块查找（索引顺序查找）

思路：先让数据分块有序，即分成若干子表，要求每个子表中的数据元素值都比后一块中的数值小（但子表内部未必有序）。然后将各子表中的最大关键字构成一个索引表，表中还要包含每个子表的起始地址（即头指针）。

特点：块间有序，块内无序。

查找：块间折半，块内线性

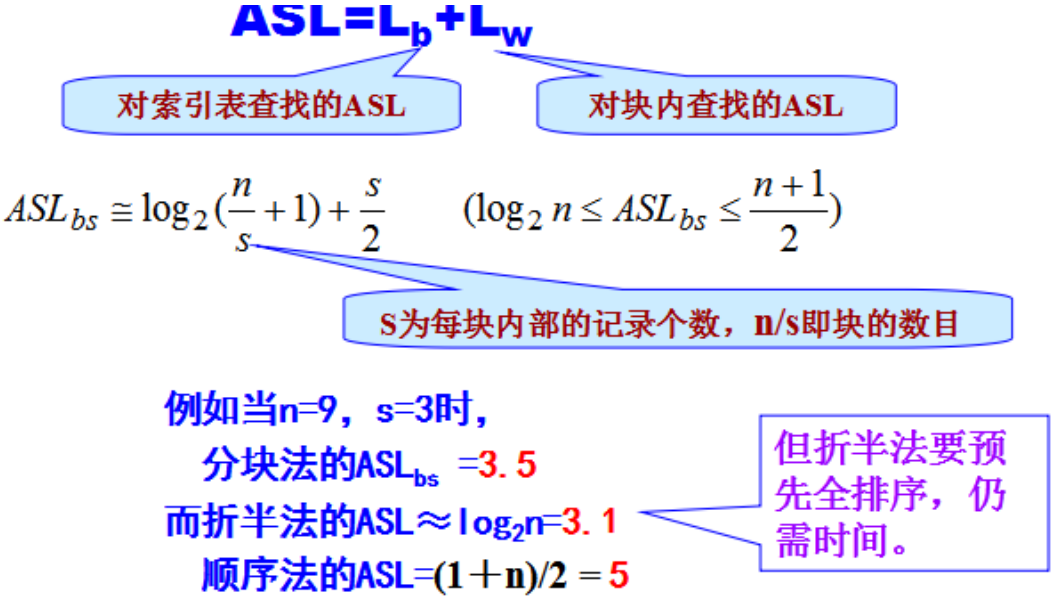
查找步骤分两步进行：

① 对索引表使用折半查找法（因为索引表是有序表）；

② 确定了待查关键字所在的子表后，在子表内采用顺序查找法（因为各子表内部是无序表）；

查找

$$ASL = \frac{1}{n}\sum_{j=1}^{m} j \cdot 2^{j-1} = \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2 n$$



◆ **动态查找表：二叉排序树，平衡二叉树。**

特点：表结构在查找过程中动态生成。

要求：对于给定值 key, 若表中存在其关键字等于 key 的记录，则查找成功返回；否则插入关键字等于 key 的记录。

① 二叉排序树的定义

----或是一棵空树；或者是具有如下性质的非空二叉树：

（1）左子树的所有结点均小于根的值；

（2）右子树的所有结点均大于根的值；

（3）它的左右子树也分别为二叉排序树。

② 二叉排序树的插入与删除

思路：查找不成功，生成一个新结点 s，插入到二叉排序树中；查找成功则返回。

```
SearchBST (K,  &t) { //K 为待查关键字，t 为根结点指针
    p=t;          //p 为查找过程中进行扫描的指针
    while（p!=NULL）{
    case {
    K= p->data:   {查找成功，return }
    K< p->data :   {q=p；p=p->L_child }   //继续向左搜索
    K> p->data :   {q=p；p=p->R_child } //继续向右搜索
              }
}   //查找不成功则插入到二叉排序树中
  s =(BiTree)malloc(sizeof(BiTNode));
  s->data=K; s ->L_child=NULL; s ->R_child=NULL;
    //查找不成功，生成一个新结点 s，插入到二叉排序树叶子处
  case {
  t=NULL：      t=s;    //若 t 为空，则插入的结点 s 作为根结点
  K < q->data: q->L_child=s;   //若 K 比叶子小，挂左边
  K > q->data: q->R_child=s; //若 K 比叶子大，挂右边
            }
return OK
  }
```

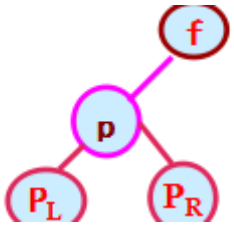
③ 二叉排序树的删除操作如何实现？

如何删除一个结点？

假设：*p 表示被删结点的指针； PL 和 PR 分别表示*P 的左、右孩子指针；

*f 表示*p 的双亲结点指针；并假定*p 是*f 的左孩子；则可能有三种情况：

- ***p为叶子： 删除此结点时，直接修改*f指针域即可；**
- ***p只有一棵子树（或左或右）：令P_L或P_R为*f的左孩子即可；**
- ***p有两棵子树：情况最复杂 →**



*p 有两棵子树时，如何进行删除操作？

设删除前的中序遍历序列为：…. PL　s　p　PR　f

//显然 p 的直接前驱是 s ，s 是*p 左子树最右下方的结点

希望删除 p 后，其它元素的相对位置不变。有两种解决方法：

法 1：令*p 的左子树为 *f 的左子树，*p 的右子树接为*s 的右子树；　//即 fL=PL ；SR=PR ；

法 2：直接令*s 代替*p　// *s 为*p 左子树最右下方的结点

二叉排序树的

$$ASL \le 2(1+\frac{1}{n})\ln n$$

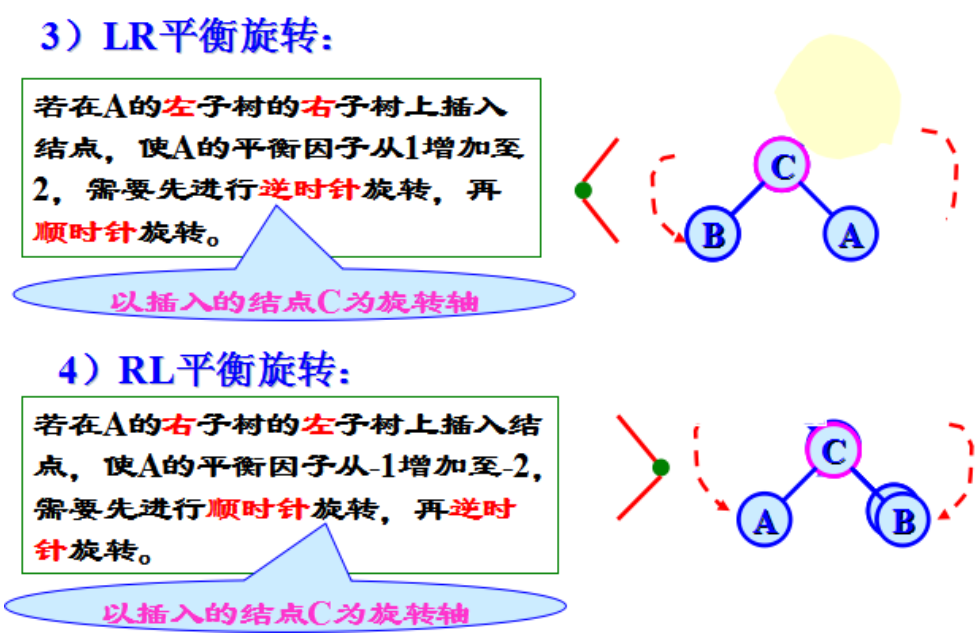
④ 平衡二叉树的定义：又称 AVL 树，即它或者是一颗空树，或者是它的左子树和右子树都是平衡二叉树，且左子树与右子树的深度之差的绝对值不超过 1。

平衡因子：——该结点的左子树的深度减去它的右子树的深度。

平衡二叉树的特点：任一结点的平衡因子只能取：-1、0 或 1。

如果在一棵 AVL 树中插入一个新结点，就有可能造成失衡，此时必须重新调整树的结构，使之恢复平衡。我们称调整平衡过程为平衡旋转。

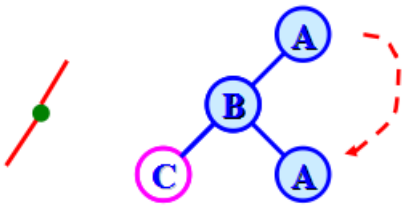
平衡旋转可以归纳为四类：



1）LL平衡旋转：

若在A的左子树的左子树上插入结点，使A的平衡因子从1增加至2，需要进行一次顺时针旋转。

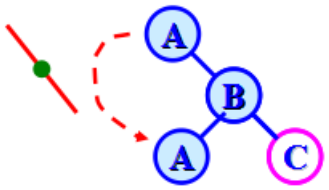
以B为旋转轴



2）RR平衡旋转：

若在A的右子树的右子树上插入结点，使A的平衡因子从-1增加至-2，需要进行一次逆时针旋转。

以B为旋转轴



学习重点：

◆ 查找表是称为集合的数据结构。因元素间关系非常松散，其操作需借助其它数据结构来实现。本章列举了三种方法（静态查找表，动态查找表）实现查找表的运算。

◆ 顺序表因设置了监视哨使查找效率大大提高。有序表的平均查找长度不超过树的深度。

◆ 查找的 ASL

◆ 二叉排序树的形态取决于元素的输入顺序。按中序遍历可得到结点的有序序列，应熟练掌握其建立、查找，插入和删除算法。

◆ 平衡二叉树的概念，应熟练掌握手工绘制平衡二叉树。

补充：

1.查找的过程是怎样的？

给定一个值 K，在含有 n 个记录的文件中进行搜索，寻找一个关键字值等于 K 的记录，如找到则输出该记录，否则输出查找不成功的信息。

2.对查找表常用的操作有哪些？

查询某个“特定的”数据元素是否在表中；

查询某个“特定的”数据元素的各种属性；

在查找表中插入一元素；

从查找表中删除一元素。

3.哪些查找方法？

查找方法取决于表中数据的排列方式;

4.如何评估查找方法的优劣？

用比较次数的平均值来评估算法的优劣。称为平均查找长度 ASL。

ASL=∑ Pi. Ci

5.使用折半查找算法时，要求被查文件：采用顺序存贮结构、记录按关键字递增有序

6.将线性表构造成二叉排序树的优点：

① 查找过程与顺序结构有序表中的折半查找相似，查找效率高；

② 中序遍历此二叉树，将会得到一个关键字的有序序列（即实现了排序运算）；

③ 如果查找不成功，能够方便地将被查元素插入到二叉树的叶子结点上，而且插入或删除时只需修改指针而不需移动元素。

第九章　内部排序

内容提要：

◆ **排序的定义，排序可以看作是线性表的一种操作**

排序:将一组杂乱无章的数据按一定的规律顺次排列起来。

◆ **排序的分类，稳定排序与不稳定排序的定义。**

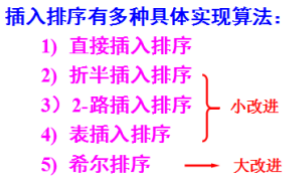
稳定性——若两个记录 A 和 B 的关键字值相等，但排序后 A、B 的先后次序保持不变，则称这种排序算法是稳定的。

◆ **插入排序（直接插入、折半插入，索引表插入、希尔插入排序）。**

插入排序的基本思想是：

每步将一个待排序的对象，按其关键码大小，插入到前面已经排好序的一组对象的适当位置上，直到对象全部插入为止。

简言之，边插入边排序，保证子序列中随时都是排好序的。



1) 直接插入排序

在已形成的有序表中线性查找，并在适当位置插入，把原来位置上的元素向后顺移。

例1：关键字序列T=（13，6，3，31，9，27，5，11），
请写出直接插入排序的中间过程序列。

【13】, 6, 3, 31, 9, 27, 5, 11
【6, 13】, 3, 31, 9, 27, 5, 11
【3, 6, 13】, 31, 9, 27, 5, 11
【3, 6, 13，31】, 9, 27, 5, 11
【3, 6, 9, 13，31】, 27, 5, 11
【3, 6, 9, 13，27, 31】, 5, 11
【3, 5, 6, 9, 13，27, 31】, 11
【3, 5, 6, 9, 11，13，27, 31】

时间效率： 因为在最坏情况下，所有元素的比较次数总和为（0＋1＋…＋n-1)→O(n2)。
其他情况下也要考虑移动元素的次数。 故时间复杂度为 O(n2)

空间效率：仅占用 1 个缓冲单元——O（1）

算法的稳定性：因为 25*排序后仍然在 25 的后面——稳定

直接插入排序算法的实现：

```
void InsertSort ( SqList &L ) { //对顺序表 L 作直接插入排序
      for ( i = 2;   i <=L.length; i++) //假定第一个记录有序
{ L.r[0]= L.r[i];
    j=i-1 ;                          //先将待插入的元素放入“哨兵”位置
 while（L[0] .key<L[j].key)
{   L.r[j+1]= L.r[j];
     j--   ;                        }        //只要子表元素比哨兵大就不断后移
L.r[j+1]= L.r[0];          //直到子表元素小于哨兵，将哨兵值送入
                                    //当前要插入的位置（包括插入到表首）
}
}
```

2） 折半插入排序

既然子表有序且为顺序存储结构，则插入时采用折半查找定可加速。

优点：比较次数大大减少，全部元素比较次数仅为 O(nlog2n)。

时间效率：虽然比较次数大大减少，可惜移动次数并未减少， 所以排序效率仍为 O(n2) 。

空间效率：仍为 O（1）

稳 定 性： 稳定

若记录是链表结构，用直接插入排序行否？

答：行，而且无需移动元素，时间效率更高！

但请注意：单链表结构无法实现“折半查找”

3） 表插入排序

基本思想：在顺序存储结构中，给每个记录增开一个指针分量，在排序过程中将指针内容逐个修改为已经整理（排序）过的后继记录地址。

优点：在排序过程中不移动元素，只修改指针。

此方法具有链表排序和地址排序的特点

表插入排序算法分析：

① 无需移动记录，只需修改指针值。但由于比较次数没有减少，故时间效率仍为 O(n2) 。

② 空间效率肯定低，因为增开了指针分量（但在运算过程中没有用到更多的辅助单元）。

③ 稳定性：25 和 25*排序前后次序未变，稳定。

注：此算法得到的只是一个有序链表，查找记录时只能满足顺序查找方式。

5）希尔（shell）排序

基本思想：先将整个待排记录序列分割成若干子序列,分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

优点：让关键字值小的元素能很快前移，且序列若基本有序时，再用直接插入排序处理，时间效率会高很多。

例：关键字序列 T=(49，38，65，97, 76, 13, 27, 49*，55，04），请写出希尔排序的具体实现过程。

r[i]	0	1	2	3	4	5	6	7	8	9	10
初态：		49	38	65	97	76	13	27	49*	55	04
第1趟 (dk=5)		13	27	49*	55	04	49	38	65	97	76
第2趟 (dk=3)		13	04	49*	38	27	49	55	65	97	76
第3趟 (dk=1)		04	13	27	38	49*	49	55	65	76	97

时间效率：$0(n^{1.25})$～$0(1.6n^{1.25})$——由经验公式得到

空间效率：0（1）——因为仅占用1个缓冲单元

算法的稳定性：不稳定——因为49*排序后却到了49的前面

◆ 交换排序（冒泡排序、快速排序）。

交换排序的基本思想是：两两比较待排序记录的关键码，如果发生逆序（即排列顺序与排序后的次序正好相反），则交换之，直到所有记录都排好序为止。

1) 冒泡排序

基本思路：每趟不断将记录两两比较，并按“前小后大”（或“前大后小”）规则交换。

优点：每趟结束时，不仅能挤出一个最大值到最后面位置，还能同时部分理顺其他元素；一旦下趟没有交换发生，还可以提前结束排序。

前提：顺序存储结构

例：关键字序列 T=(21，25，49，25*，16，08），请写出冒泡排序的具体实现过程。

初态：21，25，49，25*，16，08

第1趟 21，25，25*，16，08，49

第2趟 21，25，16，08，25*，49

第3趟 21，16，08，25，25*，49

第4趟 16，08，21，25，25*，49

第5趟 08，16，21，25，25*，49

冒泡排序的算法分析：

时间效率：O（n2) —因为要考虑最坏情况
空间效率：O（1） —只在交换时用到一个缓冲单元
稳 定 性： 稳定 —25 和 25*在排序前后的次序未改变
冒泡排序的优点：每一趟整理元素时，不仅可以完全确定一个元素的位置（挤出一个泡到表尾），还可以对前面的元素作一些整理，所以比一般的排序要快。

2） 快速排序
基本思想：从待排序列中任取一个元素 (例如取第一个) 作为中心，所有比它小的元素一律前放，所有比它大的元素一律后放，形成左右两个子表；然后再对各子表重新选择中心元素并依此规则调整，直到每个子表的元素只剩一个。此时便为有序序列了。
优点：因为每趟可以确定不止一个元素的位置，而且呈指数增加，所以特别快！
前提：顺序存储结构

例1：关键字序列 T=(21，25，49， 25*， 16，08），
请写出快速排序的算法步骤。

设以首元素为枢轴中心

初态： 21， 25， 49， 25*， 16， 08
第1趟：(08，16)，21 ，(25*，49, 25)
第2趟： (08)，16，21，25*，(25，49)
第3趟： 08，16，21，25*， 25，(49)

时间效率：O(nlog2n) —因为每趟确定的元素呈指数增加
空间效率：O（log2n）—因为递归要用栈(存每层 low，high 和 pivot)
稳 定 性： 不 稳 定 —因为有跳跃式交换。

◆ 选择排序（简单选择排序、树形选择排序、堆排序）。
选择排序的基本思想是：每一趟在后面 n-i 个待排记录中选取关键字最小的记录作为有序序列中的第 i 个记录。
1）简单选择排序
思路异常简单：每经过一趟比较就找出一个最小值，与待排序列最前面的位置互换即可。
——首先，在 n 个记录中选择最小者放到 r[1]位置；然后，从剩余的 n-1 个记录中选择最小者放到 r[2]位置；…如此进行下去，直到全部有序为止。
优点：实现简单
缺点：每趟只能确定一个元素，表长为 n 时需要 n-1 趟
前提：顺序存储结构

例：关键字序列T=（21，25，49，25*，16，08），请给出简单选择排序的具体实现过程。

直接选择排序

原始序列：21，25，49，25*，16，08　　最小值 08 与 r[1]交换位置

第1趟　08，25，49，25*，16，21

第2趟　08，16，49，25*，25，21

第3趟　08，16，21，25*，25，49

第4趟　08，16，21，25*，25，49

第5趟　08，16，21，25*，25，49

时间效率：0(n²)——虽移动次数较少，但比较次数仍多。
空间效率：0（1）——没有附加单元（仅用到1个temp）
算法的稳定性：不稳定——因为排序时，25*到了25的前面。

```
Void SelectSort(SqList   &L ) {
      for (i=1;   i<L.length; ++i){
       j = SelectMinKey(L,i);
      if( i!=j )   r[i]  ·  r[j];
        } //for
   }   //SelectSort
```

2）锦标赛排序　(又称树形选择排序)

基本思想：与体育比赛时的淘汰赛类似。

首先对 n 个记录的关键字进行两两比较，得到 ·n/2· 个优胜者(关键字小者)，作为第一步比较的结果保留下来。然后在这 ·n/2· 个较小者之间再进行两两比较，…，如此重复，直到选出最小关键字的记录为止。

优点：减少比较次数，加快排序速度

缺点：空间效率低

3）堆排序

1.堆的定义：设有 n 个元素的序列 k1，k2，…，kn，当且仅当满足下述关系之一时，称之为堆。

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad \text{或者} \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \qquad i=1,2,\dots n/2$$

解释：如果让满足以上条件的元素序列 （k1，k2，…，kn）顺次排成一棵完全二叉树，则此树的特点是：树中所有结点的值均大于（或小于）其左右孩子，此树的根结点（即堆顶）必最大（或最小）。

2. 怎样建堆？

步骤：从最后一个非终端结点开始往前逐步调整，让每个双亲大于（或小于）子女，直到根结点为止。

堆排序算法分析：

时间效率：　O(nlog2n)。因为整个排序过程中需要调用 n-1 次 HeapAdjust( )算法，而算法本身耗时为 log2n；

空间效率：O(1)。仅在第二个 for 循环中交换记录时用到一个临时变量 temp。

稳定性：　不稳定。

优点：对小文件效果不明显，但对大文件有效。

学习要点：

◆ 各种排序所基于的基本思想。

◆ 在“最好”和“最差”情况下，排序性能的分析，是否是稳定排序的结论，时间效率和空间效率。

◆ 对每种排序方法的学习，应掌握其本质（排序所基于的思想），熟练掌握手工模拟各种排序的过程。

补充：

1.排序算法的好坏如何衡量？

时间效率——排序速度（即排序所花费的全部比较次数）

空间效率——占内存辅助空间的大小

稳定性——若两个记录 A 和 B 的关键字值相等，但排序后 A、B 的先后次序保持不变，则称这种排序算法是稳定的。

2.“快速排序”是否真的比任何排序算法都快？

——基本上是，因为每趟可以确定的数据元素是呈指数增加的。