算法与数据结构

教材：《数据结构(C语言版)》。严蔚敏，吴伟民 编著。清华大学出版社。

参考文献：

1 《数据结构》 。张选平，雷咏梅 编， 严蔚敏 审。 机械工业出版社。

2 《数据结构与算法分析》。**Clifford A. Shaffer**著，张 铭，刘晓丹 译。电子工业出版社。

3 《数据结构习题与解析(C语实言版)》。李春葆。清华大学出版社。

4 《数据结构与算法》。夏克俭 编著。国防工业出版社。

第1章　绪 论

目前，计算机已深入到社会生活的各个领域，其应用已不再仅仅局限于科学计算，而更多的是用于控制，管理及数据处理等非数值计算领域。计算机是一门研究用计算机进行信息表示和处理的科学。这里面涉及到两个问题：信息的表示，信息的处理。

信息的表示和组织又直接关系到处理信息的程序的效率。随着应用问题的不断复杂，导致信息量剧增与信息范围的拓宽，使许多系统程序和应用程序的规模很大，结构又相当复杂。因此，必须分析待处理问题中的对象的特征及各对象之间存在的关系，这就是数据结构这门课所要研究的问题。

计算机求解问题的一般步骤

编写解决实际问题的程序的一般过程：

- 如何用数据形式描述问题?—即由问题抽象出一个适当的数学模型;
- 问题所涉及的数据量大小及数据之间的关系;
- 如何在计算机中存储数据及体现数据之间的关系?
- 处理问题时需要对数据作何种运算?
- 所编写的程序的性能是否良好?

上面所列举的问题基本上由数据结构这门课程来回答。

1.1 数据结构及其概念

《算法与数据结构》是计算机科学中的一门综合性专业基础课。是介于数学、计算机硬件、计算机软件三者之间的一门核心课程，不仅是一般程序设计的基础，而且是设计和实现编译程序、操作系统、数据库系统及其他系统程序和大型应用程序的重要基础。

1.1.1 数据结构的例子

例1：电话号码查询系统

设有一个电话号码薄，它记录了N个人的名字和其相应的电话号码，假定按如下形式安排：(a_1, b_1)，(a_2, b_2)，…(a_n, b_n)，其中$a_i, b_i(i=1，2…n)$ 分别表示某人的名字和电话号码。 本问题是一种典型的表格问题。如表1-1，数据与数据成简单的一对一的线性关系。

姓名	电话号码
陈海	13612345588
李四锋	13056112345
∘ ∘ ∘	∘ ∘ ∘

表1-1　线性表结构

例2：磁盘目录文件系统

磁盘根目录下有很多子目录及文件，每个子目录里又可以包含多个子目录及文件，但每个子目录只有一个父目录，依此类推：

本问题是一种典型的树型结构问题，如图1-1，数据与数据成一对多的关系，是一种典型的非线性关系结构—**树形结构**。

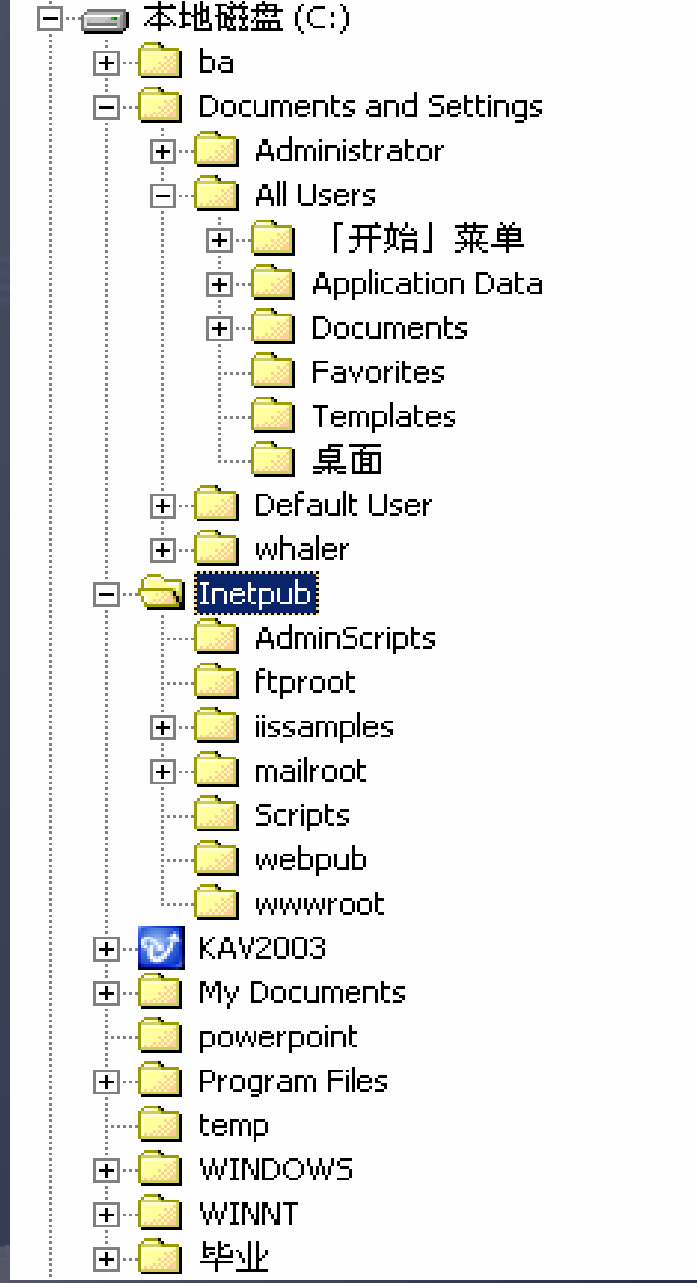


图1-1　树形结构

例3：交通网络图

从一个地方到另外一个地方可以有多条路径。本问题是一种典型的网状结构问题，数据与数据成多对多的关系，是一种非线性关系结构。

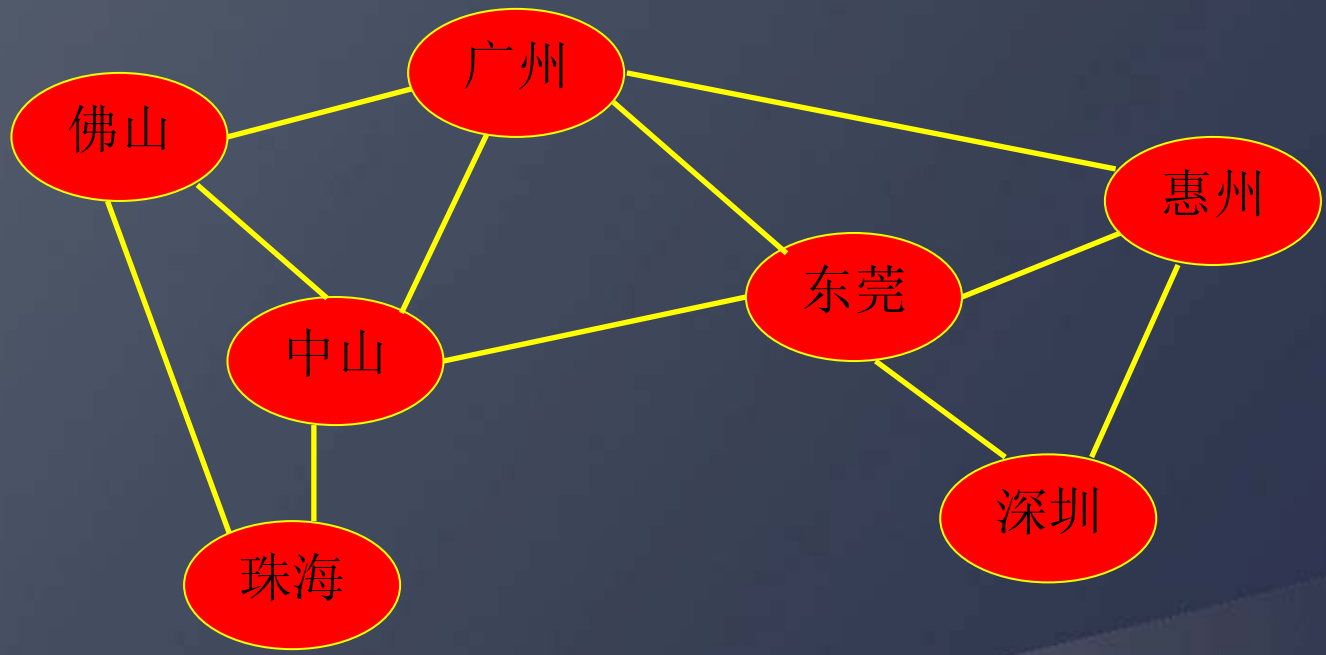


图1-2　网状结构

1.1.2 基本概念和术语

数据(Data)：是客观事物的符号表示。在计算机科学中指的是所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素(Data Element)：是数据的基本单位，在程序中通常作为一个整体来进行考虑和处理。

一个数据元素可由若干个**数据项(Data Item)**组成。数据项是数据的不可分割的最小单位。数据项是对客观事物某一方面特性的数据描述。

数据对象(Data Object)：是性质相同的数据元素的集合，是数据的一个子集。如字符集合C={‘A’,’B’,’C,…}。

数据结构(Data Structure)：是指相互之间具有(存在)一定联系(关系)的数据元素的集合。元素之间的相互联系(关系)称为逻辑结构。数据元素之间的逻辑结构有四种基本类型，如图1-3所示。

① 集合：结构中的数据元素除了“同属于一个集合”外，没有其它关系。

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 图状结构或网状结构：结构中的数据元素之间存在多对多的关系。

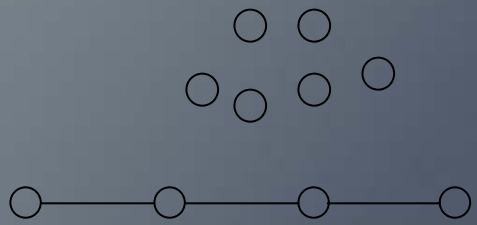
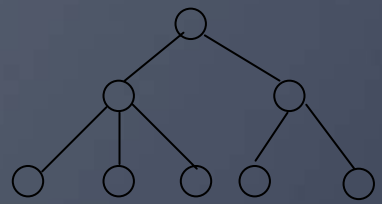
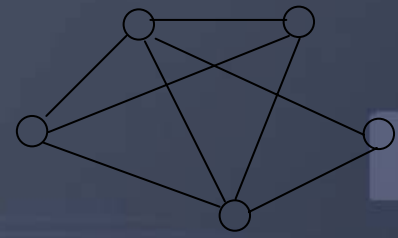




图1-3　　四类基本结构图

1.1.3 数据结构的形式定义

数据结构的形式定义是一个二元组：

Data-Structure=(D，S)

其中：D是数据元素的有限集，S是D上关系的有限集。

例2：设数据逻辑结构B=（K，R）

K={k_1, k_2, …, k_9}

R={ <k_1, k_3>，<k_1, k_8>，<k_2, k_3>，<k_2, k_4>，<k_2, k_5>，<k_3, k_9>，<k_5, k_6>，<k_8, k_9>，<k_9, k_7>，<k_4, k_7>，<k_4, k_6> }

画出这逻辑结构的图示，并确定那些是起点，那些是终点

数据元素之间的关系可以是元素之间代表某种含义的自然关系，也可以是为处理问题方便而人为定义的关系，这种自然或人为定义的“关系”称为数据元素之间的逻辑关系，相应的结构称为逻辑结构。

1.1.4 数据结构的存储方式

数据结构在计算机内存中的存储包括数据元素的存储和元素之间的关系的表示。

元素之间的关系在计算机中有两种不同的表示方法：顺序表示和非顺序表示。由此得出两种不同的存储结构：顺序存储结构和链式存储结构。

- 顺序存储结构：用数据元素在存储器中的相对位置来表示数据元素之间的逻辑结构(关系)。

– 链式存储结构：在每一个数据元素中增加一个存放另一个元素地址的指针(pointer)，用该指针来表示数据元素之间的逻辑结构(关系)。

例：设有数据集合A={3.0,2.3,5.0,-8.5,11.0}，两种不同的存储结构。

– 顺序结构：数据元素存放的地址是连续的；

– 链式结构：数据元素存放的地址是否连续没有要求。

数据的逻辑结构和物理结构是密不可分的两个方面，一个算法的设计取决于所选定的逻辑结构，而算法的实现依赖于所采用的存储结构。

在C语言中，用一维数组表示顺序存储结构；用结构体类型表示链式存储结构。

数据结构的三个组成部分：

逻辑结构： 数据元素之间逻辑关系的描述

D_S=（D，S）

存储结构： 数据元素在计算机中的存储及其逻辑关系的表现称为数据的存储结构或物理结构。

数据操作： 对数据要进行的运算。

本课程中将要讨论的三种逻辑结构及其采用的存储结构如图1-4所示。

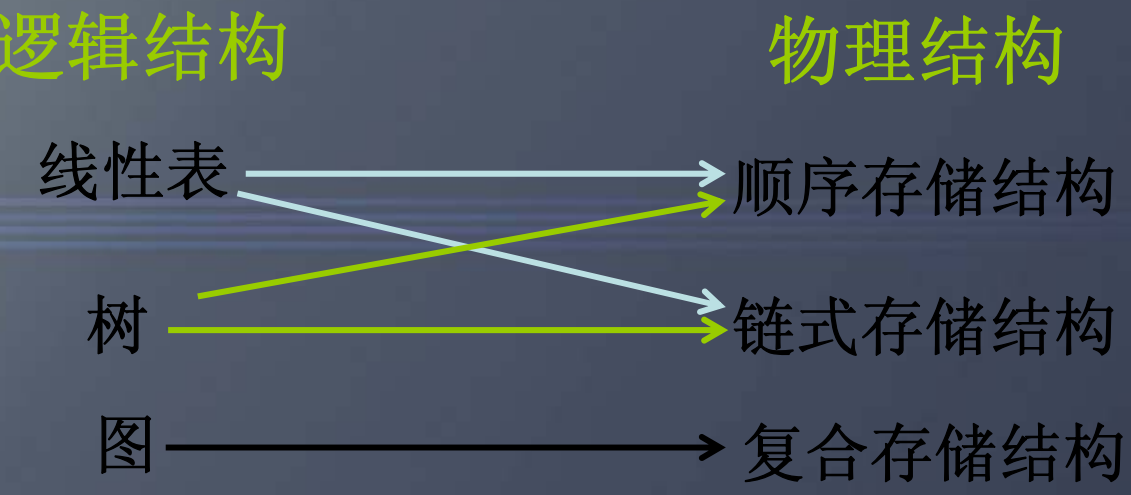


图1-4　逻辑结构与所采用的存储结构

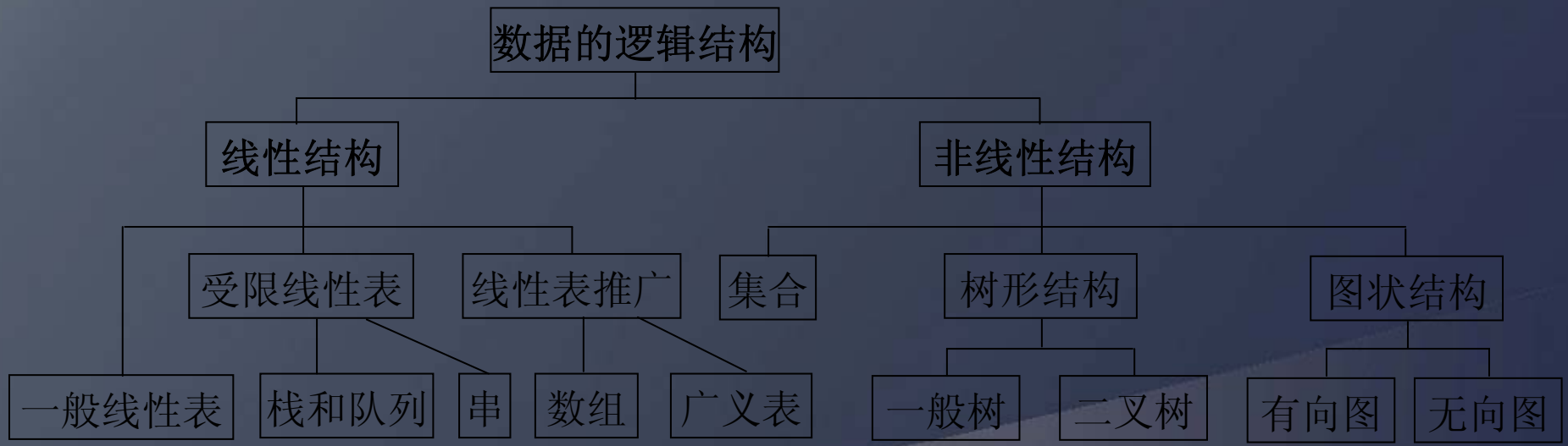


图1-5　数据逻辑结构层次关系图

1.1.5 数据类型

数据类型(Data Type)：指的是一个值的集合和定义在该值集上的一组操作的总称。

数据类型是和数据结构密切相关的一个概念。 在C语言中数据类型有：基本类型和构造类型。

数据结构不同于数据类型，也不同于数据对象，它不仅要描述数据类型的数据对象，而且要描述数据对象各元素之间的相互关系。

1.1.6 数据结构的运算

数据结构的主要运算包括：

⑴ 建立(Create)一个数据结构；

⑵ 消除(Destroy)一个数据结构；

⑶ 从一个数据结构中删除(Delete)一个数据元素；

⑷ 把一个数据元素插入(Insert)到一个数据结构中；

⑸ 对一个数据结构进行访问(Access)；

⑹ 对一个数据结构(中的数据元素)进行修改(Modify)；

⑺ 对一个数据结构进行排序(Sort)；

⑻ 对一个数据结构进行查找(Search)。

1.2 抽象数据类型

抽象数据类型(Abstract Data Type，简称ADT)：是指一个数学模型以及定义在该模型上的一组操作。

ADT的定义仅是一组逻辑特性描述，与其在计算机内的表示和实现无关。因此，不论ADT的内部结构如何变化，只要其数学特性不变，都不影响其外部使用。

ADT的形式化定义是三元组：ADT=(D，S，P)

其中：D是数据对象，S是D上的关系集，P是对D的基本操作集。

ADT的一般定义形式是：

ADT <抽象数据类型名>{

数据对象： <数据对象的定义>

数据关系： <数据关系的定义>

基本操作： <基本操作的定义>

} ADT <抽象数据类型名>

- 其中数据对象和数据关系的定义用伪码描述。
- 基本操作的定义是：

<基本操作名>(<参数表>)

初始条件： <初始条件描述>

操作结果： <操作结果描述>

– 初始条件：描述操作执行之前数据结构和参数应满足的条件;若不满足，则操作失败，返回相应的出错信息。

– 操作结果：描述操作正常完成之后，数据结构的变化状况和 应返回的结果。

1.3 算法分析初步

1.3.1 算法

算法(Algorithm)：是对特定问题求解方法(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。

算法具有以下五个特性

① **有穷性**： 一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。

② **确定性**：算法中每一条指令必须有确切的含义。不存在二义性。且算法只有一个入口和一个出口。

③ **可行性**： 一个算法是能行的。即算法描述的操作都可以通过已经实现的基本运算执行有限次来实现。

④ **输入**： 一个算法有零个或多个输入，这些输入取自于某个特定的对象集合。

⑤ **输出**： 一个算法有一个或多个输出，这些输出是同输入有着某些特定关系的量。

一个算法可以用多种方法描述，主要有：使用自然语言描述；使用形式语言描述；使用计算机程序设计语言描述。

算法和程序是两个不同的概念。一个计算机程序是对一个算法使用某种程序设计语言的具体实现。算法必须可终止意味着不是所有的计算机程序都是算法。

在本门课程的学习、作业练习、上机实践等环节，算法都用C语言来描述。在上机实践时，为了检查算法是否正确，应编写成完整的C语言程序。

1.3.2 算法设计的要求

评价一个好的算法有以下几个标准

① 正确性(Correctness)： 算法应满足具体问题的需求。

② 可读性(Readability)： 算法应容易供人阅读和交流。可读性好的算法有助于对算法的理解和修改。

③ 健壮性(Robustness)： 算法应具有容错处理。当输入非法或错误数据时，算法应能适当地作出反应或进行处理，而不会产生莫名其妙的输出结果。

④ 通用性(Generality)： 算法应具有一般性，即算法的处理结果对于一般的数据集合都成立。

⑤ 效率与存储量需求： 效率指的是算法执行的时间；存储量需求指算法执行过程中所需要的最大存储空间。一般地，这两者与问题的规模有关。

1.3.3 算法效率的度量

算法执行时间需通过依据该算法编制的程序在计算机上运行所消耗的时间来度量。其方法通常有两种：

事后统计：计算机内部进行执行时间和实际占用空间的统计。

问题：必须先运行依据算法编制的程序；依赖软硬件环境，容易掩盖算法本身的优劣；没有实际价值。

事前分析：求出该算法的一个时间界限函数。

与此相关的因素有：

- 依据算法选用何种策略；
- 问题的规模；
- 程序设计的语言；
- 编译程序所产生的机器代码的质量；
- 机器执行指令的速度；

撇开软硬件等有关部门因素，可以认为一个特定算法“运行工作量”的大小，只依赖于问题的规模（通常用n表示），或者说，它是问题规模的函数。

算法分析应用举例

算法中基本操作重复执行的次数是问题规模n的某个函数，其时间量度记作 T(n)=O(f(n))，称作算法的渐近时间复杂度(Asymptotic Time complexity)，简称时间复杂度。

一般地，常用最深层循环内的语句中的原操作的执行频度(重复执行的次数)来表示。

“O”的定义： 若f(n)是正整数n的一个函数，则 O(f(n))表示$\exists$ M≥0 ，使得当n ≥ n_0时，| f(n) | ≤ M | f(n_0) | 。

表示时间复杂度的阶有：

O(1) ：常量时间阶　　O (n)：线性时间阶

O(logn) ：对数时间阶　O(nlogn) ：线性对数时间阶

$O(n^k)$： k≥2，k次方时间阶

例1 两个n阶方阵的乘法

```
for(i=1，i<=n; ++i)
    for(j=1; j<=n; ++j)
      {  c[i][j]=0 ;
          for(k=1; k<=n; ++k)
              c[i][j]+=a[i][k]*b[k][j] ; }
```

由于是一个三重循环，每个循环从1到n，则总次数为：$n\times n\times n=n^3$ 时间复杂度为$T(n)=O(n^3)$

例2 {++x; s=0 ;}

将x自增看成是基本操作，则语句频度为1，即时间复杂度为$O(1)$。

如果将s=0也看成是基本操作，则语句频度为 2 ，其时间复杂度仍为 O(1)，即常量阶。

例 3　for(i=1; i<=n; ++i)

{ ++x; s+=x ; }

语句频度为：2n，其时间复杂度为：O(n) ，即为线性阶。

例 4　for(i=1; i<=n; ++i)

for(j=1; j<=n; ++j)

{ ++x; s+=x ; }

语句频度为：$2n^2$ ，其时间复杂度为：$O(n^2)$ ，即为平方阶。

定理：若$A(n)=a_m n^m+a_{m-1} n^{m-1}+\ldots+a_1 n+a_0$是一个m次多项式，则$A(n)=O(n^m)$

例5

```
for(i=2;i<=n;++i)
    for(j=2;j<=i-1;++j)
        {++x; a[i,j]=x; }
```

语句频度为：$1+2+3+\ldots+n-2=(1+n-2)\times(n-2)/2$

$=(n-1)(n-2)/2=n^2-3n+2$

∴时间复杂度为$O(n^2)$，即此算法的时间复杂度为平方阶。

– 一个算法时间为O(1)的算法，它的基本运算执行的次数是固定的。因此，总的时间由一个常数（即零次多项式）来限界。而一个时间为$O(n^2)$的算法则由一个二次多项式来限界。

以下六种计算算法时间的多项式是最常用的。其关系为：

$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$

– 指数时间的关系为：

$O(2^n)<O(n!)<O(n^n)$

当n取得很大时，指数时间算法和多项式时间算法在所需时间上非常悬殊。因此，只要有人能将现有指数时间算法中的任何一个算法化简为多项式时间算法，那就取得了一个伟大的成就。

– 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。

例1：素数的判断算法。

```
Void prime( int n)
   /*  n是一个正整数  */
   {   int i=2 ;
       while ( (n% i)!=0 && i*1.0< sqrt(n) )   i++ ;
       if (i*1.0>sqrt(n) )
          printf("&d 是一个素数\n" , n) ;
       else
          printf("&d 不是一个素数\n" , n) ;
   }
```

嵌套的最深层语句是i++；其频度由条件((n% i)!=0 && i*1.0< sqrt(n)) 决定，显然i*1.0< sqrt(n) ，时间复杂度$O(n^{1/2})$。

例2：冒泡排序法。

```
Void bubble_sort(int a[]，int n)
  {  change=false;
     for (i=n-1; change=TURE; i>1 && change; --i)
        for (j=0; j<i; ++j)
           if (a[j]>a[j+1])
              {    a[j] ←→a[j+1] ;  change=TURE ; }
  }
```

- 最好情况：0次
- 最坏情况：1+2+3+…+n-1=n(n-1)/2
- 平均时间复杂度为： $O(n^2)$

1.3.4 算法的空间分析

空间复杂度(Space complexity)：是指算法编写成程序后，在计算机中运行时所需存储空间大小的度量。记作： S(n)=O(f(n))

其中： n为问题的规模(或大小)

该存储空间一般包括三个方面：

- 指令常数变量所占用的存储空间;
- 输入数据所占用的存储空间;
- 辅助(存储)空间。

一般地，算法的空间复杂度指的是辅助空间。

- 一维数组a[n]： 空间复杂度 O(n)
- 二维数组a[n][m]： 空间复杂度 O(n*m)

习 题 一

1 简要回答术语：数据，数据元素，数据结构，数据类型。

2 数据的逻辑结构？数据的物理结构？逻辑结构与物理结构的区别和联系是什么？

3 数据结构的主要运算包括哪些？

4 算法分析的目的是什么？算法分析的主要方面是什么？

5 分析以下程序段的时间复杂度，请说明分析的理由或原因。

(1)

```
Sum1( int n )
   {   int p=1, sum=0, m ;
       for (m=1; m<=n; m++)
           {  p*=m ; sum+=p ;  }
       return (sum) ;
   }
```

(2)

```
Sum2( int n )
   {   int sum=0, m, t ;
       for (m=1; m<=n; m++)
           {   p=1 ;
               for (t=1; t<=m; t++)  p*=t ;
               sum+=p ;
           }
       return (sum) ;
   }
```

(3) 递归函数

```
fact( int n )
   {   if (n<=1)  return(1) ;
       else return( n*fact(n-1)) ;
   }
```

第2章　线性表

线性结构是最常用、最简单的一种数据结构。而线性表是一种典型的线性结构。其基本特点是线性表中的数据元素是有序且是有限的。在这种结构中：

① 存在一个唯一的被称为“第一个”的数据元素；

② 存在一个唯一的被称为“最后一个”的数据元素；

③ 除第一个元素外，每个元素均有唯一一个直接前驱；

④ 除最后一个元素外，每个元素均有唯一一个直接后继。

2.1 线性表的逻辑结构

2.1.1 线性表的定义

线性表(Linear List)：是由n(n≧0)个数据元素(结点)a_1，a_2，…a_n组成的有限序列。该序列中的所有结点具有相同的数据类型。其中数据元素的个数n称为线性表的长度。

当n=0时，称为空表。

当n>0时，将非空的线性表记作：(a_1，a_2，…a_n)

a_1称为线性表的第一个(首)结点，a_n称为线性表的最后一个(尾)结点。

a_1，a_2，…a_{i-1}都是a_i($2 \leqq i \leqq n$)的前驱，其中a_{i-1}是a_i的直接前驱;

a_{i+1}，a_{i+2}，…a_n都是a_i($1 \leqq i \leqq n-1$)的后继，其中a_{i+1}是a_i的直接后继。

2.1.2 线性表的逻辑结构

线性表中的数据元素a_i所代表的具体含义随具体应用的不同而不同，在线性表的定义中，只不过是一个抽象的表示符号。

- 线性表中的结点可以是单值元素(每个元素只有一个数据项)。

例1: 26个英文字母组成的字母表：（A，B，C、…、Z）

例2： 某校从1978年到1983年各种型号的计算机拥有量的变化情况：(6，17，28，50，92，188）

例3： 一副扑克的点数 (2，3，4，...，J，Q，K，A)

- 线性表中的结点可以是记录型元素，每个元素含有多个数据项，每个项称为结点的一个域。每个元素有一个可以唯一标识每个结点的数据项组，称为关键字。

例4： 某校2001级同学的基本情况：{('2001414101'，'张里户'，'男'，06/24/1983)，('2001414102'，'张化司'，'男'，08/12/1984) ...，('2001414102'，'李利辣'，'女'，08/12/1984) }

- 若线性表中的结点是按值(或按关键字值)由小到大(或由大到小)排列的，称线性表是有序的。

- 线性表是一种相当灵活的数据结构，其长度可根据需要增长或缩短。
- 对线性表的数据元素可以访问、插入和删除。

2.1.3 线性表的抽象数据类型定义

ADT List{

数据对象：$D = \{ a_i \mid a_i \in ElemSet,\ i=1,2,\ldots,n,\ n \geqq 0 \}$

数据关系：$R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D,\ i=2,3,\ldots,n \}$

基本操作：

InitList(&L)

操作结果：构造一个空的线性表L；

ListLength(L)

初始条件：线性表L已存在；

操作结果：若L为空表，则返回TRUE，否则返回FALSE；

….

GetElem(L, i, &e)

初始条件：线性表L已存在，1≦i≦ListLength(L)；

操作结果：用e返回L中第i个数据元素的值；

ListInsert (L, i, &e)

初始条件：线性表L已存在，1≦i≦ListLength(L) ；

操作结果：在线性表L中的第i个位置插入元素e；

…

} ADT List

2.2 线性表的顺序存储

2.2.1 线性表的顺序存储结构

顺序存储：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。

顺序存储的线性表的特点：

- 线性表的逻辑顺序与物理顺序一致;
- 数据元素之间的关系是以元素在计算机内“物理位置相邻”来体现。

设有非空的线性表：$(a_1, a_2, \ldots a_n)$。顺序存储如图2-1所示。

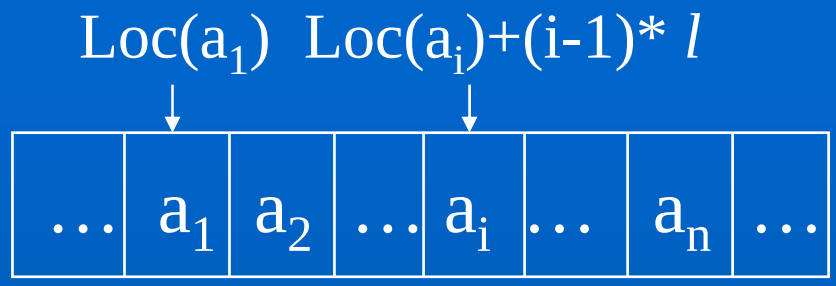


图2-1　线性表的顺序存储表示

在具体的机器环境下：设线性表的每个元素需占用*l*个存储单元，以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第**i+1**个数据元素的存储位置**LOC(a_{i+1})**和第**i**个数据元素的存储位置**LOC(a_i)**之间满足下列关系：

$$LOC(a_{i+1}) = LOC(a_i) + l$$

线性表的第**i**个数据元素**a_i**的存储位置为：

$$LOC(a_i) = LOC(a_1) + (i-1) * l$$

在高级语言(如C语言)环境下：数组具有随机存取的特性，因此，借助数组来描述顺序表。除了用数组来存储线性表的元素之外，顺序表还应该有表示线性表的长度属性，所以用结构类型来定义顺序表类型。

```
#define OK  1
#define ERROR  -1
#define MAX_SIZE 100
typedef int Status;
typedef int ElemType;
typedef struct sqlist
  {  ElemType Elem_array[MAX_SIZE];
    int  length;
  } SqList;
```

2.2.2 顺序表的基本操作

顺序存储结构中，很容易实现线性表的一些操作：初始化、赋值、查找、修改、插入、删除、求长度等。

以下将对几种主要的操作进行讨论。

1 顺序线性表初始化

```
Status Init_SqList( SqList *L )
  {  L->elem_array=( ElemType *
  )malloc(MAX_SIZE*sizeof( ElemType ) ) ;
     if ( !L -> elem_array ) return  ERROR ;
     else {   L->length= 0 ;    return OK ;  }
  }
```

2　顺序线性表的插入

在线性表 $L=(a_1, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ 中的第$i(1 \leqq i \leqq n)$个位置上插入一个新结点e，使其成为线性表：

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

实现步骤

（1） 将线性表L中的第i个至第n个结点后移一个位置。

（2） 将结点e插入到结点a_{i-1}之后。

（3） 线性表长度加1。

算法描述

```
Status Insert_SqList(Sqlist *L, int i , ElemType e)
   {  int j ;
     if ( i<0||i>L->length-1)  return  ERROR ;
     if (L->length>=MAX_SIZE)
        {   printf("线性表溢出!\n"); return  ERROR ; }
     for ( j=L->length–1; j>=i-1; --j )
        L->Elem_array[j+1]=L->Elem_array[j];
           /*  i-1位置以后的所有结点后移  */
     L->Elem_array[i-1]=e;   /*  在i-1位置插入结点  */
     L->length++ ;
     return  OK ;
   }
```

时间复杂度分析

在线性表L中的第i个元素之前插入新结点，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表L中的第i个元素之前插入结点的概率为P_i，不失一般性，设各个位置插入是等概率，则$P_i = 1/(n+1)$，而插入时移动结点的次数为n-i+1。

总的平均移动次数： $E_{insert} = \sum p_i * (n-i+1)$ $(1 \leqq i \leqq n)$

$\therefore E_{insert} = n/2$ 。

即在顺序表上做插入运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。因此算法的平均时间复杂度为O(n)。

3　顺序线性表的删除

在线性表 $L=(a_1, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ 中删除结点$a_i(1 \leqq i \leqq n)$，使其成为线性表：

$$L=(a_1, \ldots a_{i-1}, a_{i+1}, \ldots, a_n)$$

实现步骤

（1） 将线性表L中的第i+1个至第n个结点依此向前移动一个位置。

（2）线性表长度减1。

算法描述

```
ElemType  Delete_SqList(Sqlist *L, int i)
  { int k; ElemType x;
```

```
    if (L->length==0)
       {  printf("线性表L为空!\n"); return ERROR;  }
    else if ( i<1||i>L->length )
       {  printf("要删除的数据元素不存在!\n") ;
          return ERROR ; }
       else  {  x=L->Elem_array[i-1] ;   /*保存结点
       的值*/
          for ( k=i ;  k<L->length ; k++)
              L->Elem_array[k-1]=L-
          >Elem_array[k];
                    /*  i位置以后的所有结点前移  */
          L->length--;  return (x);
          }
}
```

时间复杂度分析

删除线性表L中的第i个元素，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表L中删除第i个元素的概率为P_i，不失一般性，设删除各个位置是等概率，则$P_i=1/n$，而删除时移动结点的次数为n-i。

则总的平均移动次数： $E_{delete}=\sum p_i*(n-i)$ $(1\leqslant i\leqslant n)$

$\therefore E_{delete}=(n-1)/2$ 。

即在顺序表上做删除运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。因此算法的平均时间复杂度为O(n)。

4　顺序线性表的查找定位删除

在线性表 $L=(a_1, a_2, \ldots, a_n)$ 中删除值为x的第一个结点。

实现步骤

(1) 在线性表L查找值为x的第一个数据元素。

(2) 将从找到的位置至最后一个结点依次向前移动一个位置。

(3) 线性表长度减1。

算法描述

```
Status Locate_Delete_SqList(Sqlist *L, ElemType x)
  /* 删除线性表L中值为x的第一个结点 */
{ int i=0, k;
```

```
  while (i<L->length)     /*查找值为x的第一个结点*/
    {  if (L->Elem_array[i]!=x ) i++;
       else
         { for ( k=i+1; k< L->length; k++)
              L->Elem_array[k-1]=L->Elem_array[k];
           L->length--; break;
         }
    }
  if (i>L->length)
    {   printf("要删除的数据元素不存在!\n");
       return ERROR; }
  return OK;
}
```

时间复杂度分析

时间主要耗费在数据元素的比较和移动操作上。

首先，在线性表L中查找值为x的结点是否存在；

其次，若值为x的结点存在，且在线性表L中的位置为i，则在线性表L中删除第i个元素。

设在线性表L删除数据元素概率为P_i，不失一般性，设各个位置是等概率，则$P_i=1/n$。

- 比较的平均次数： $E_{compare}=\sum p_i*i \quad (1\leqq i\leqq n)$

∴ $E_{compare}=(n+1)/2$ 。

- 删除时平均移动次数： $E_{delete}=\sum p_i*(n-i) \quad (1\leqq i\leqq n)$

∴ $E_{delete}=(n-1)/2$ 。 平均时间复杂度：

$E_{compare}+E_{delete}=n$，即为O(n)

2.3 线性表的链式存储

2.3.1 线性表的链式存储结构

链式存储：用一组任意的存储单元存储线性表中的数据元素。用这种方法存储的线性表简称线性链表。

存储链表中结点的一组任意的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。

链表中结点的逻辑顺序和物理顺序不一定相同。

为了正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其直接后继结点的地址(或位置)，称为指针(pointer)或链(link)，这两部分组成了链表中的结点结构，如图2-2所示。

链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。

每一个结只包含一个指针域的链表，称为单链表。

为操作方便，总是在链表的第一个结点之前附设一个头结点(头指针)head指向第一个结点。头结点的数据域可以不存储任何信息(或链表长度等信息)。

data：数据域，存放结点的值。**next**：指针域，存放结点的直接后继的地址。

图2-2 链表结点结构

单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名。

例1、线性表**L=(bat，cat，eat，fat，hat)**

其带头结点的单链表的逻辑状态和物理存储方式如图**2-3**所示。

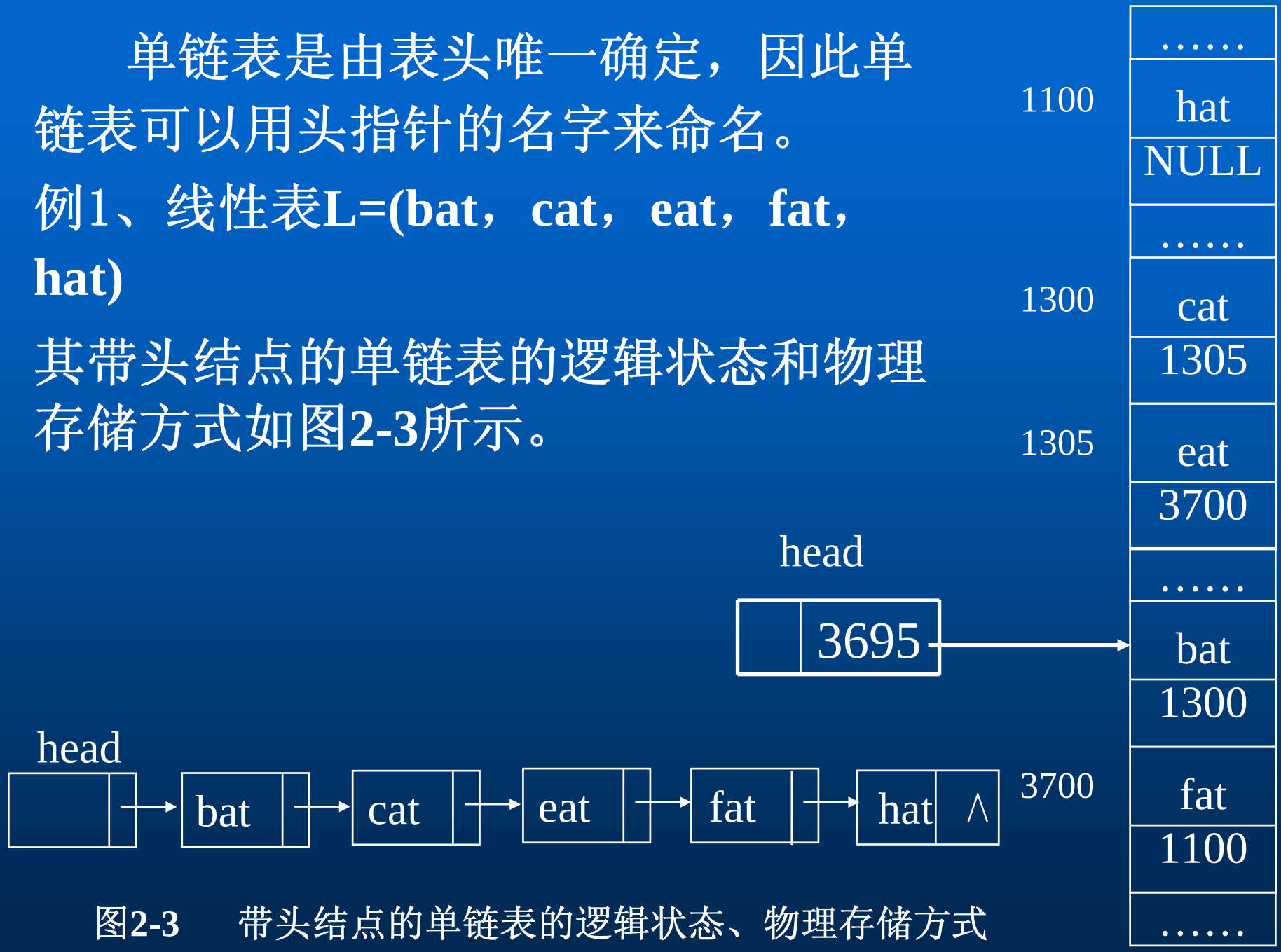


图**2-3**　带头结点的单链表的逻辑状态、物理存储方式

1 结点的描述与实现

C语言中用带指针的结构体类型来描述

```
typedef  struct  Lnode
  {  ElemType  data;     /*数据域，保存结点的值 */
     struct   Lnode  *next;     /*指针域*/
  }LNode;        /*结点的类型 */
```

2 结点的实现

结点是通过动态分配和释放来的实现，即需要时分配，不需要时释放。实现时是分别使用C语言提供的标准函数：**malloc()**，**realloc()**，**sizeof()**，**free()**。

动态分配

p=(LNode*)malloc(sizeof(LNode));

函数**malloc**分配了一个类型为**LNode**的结点变量的空间，并将其首地址放入指针变量**p**中。

动态释放　**free(p)；**

系统回收由指针变量**p**所指向的内存区。**P**必须是最近一次调用**malloc**函数时的返回值。

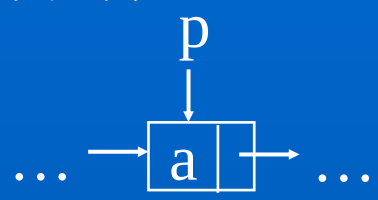
3　最常用的基本操作及其示意图

(1) 结点的赋值

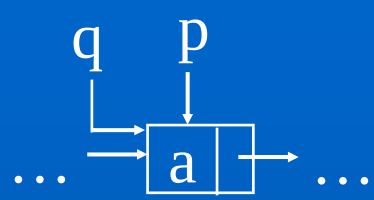
```
LNode  *p;
p=(LNode*)malloc(sizeof(LNode));
p->data=20;  p->next=NULL ;
```

(2) 常见的指针操作

① q=p；

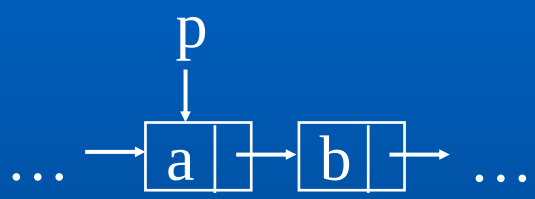


操作前

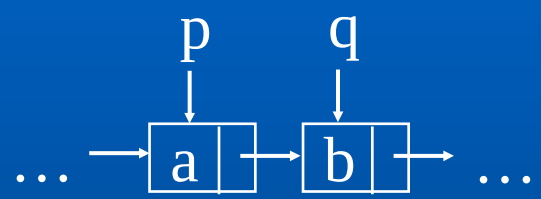


操作后

② q=p->next；

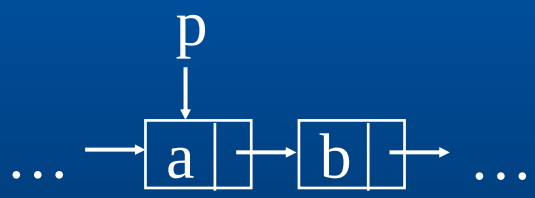


操作前

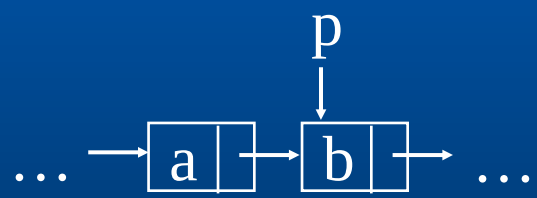


操作后

③ p=p->next；



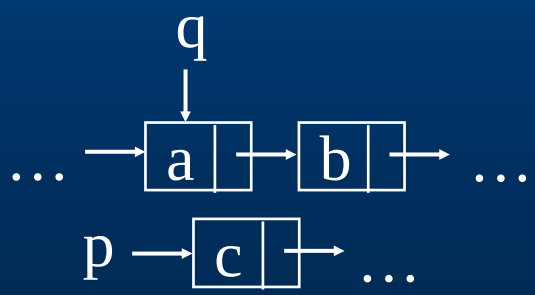
操作前



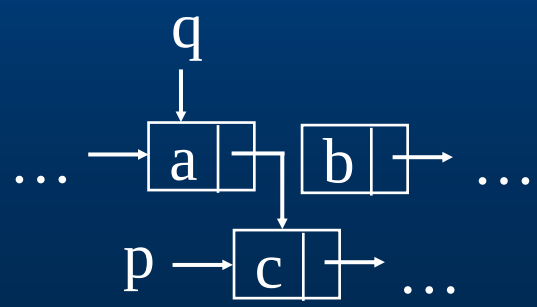
操作后

④ q->next=p；

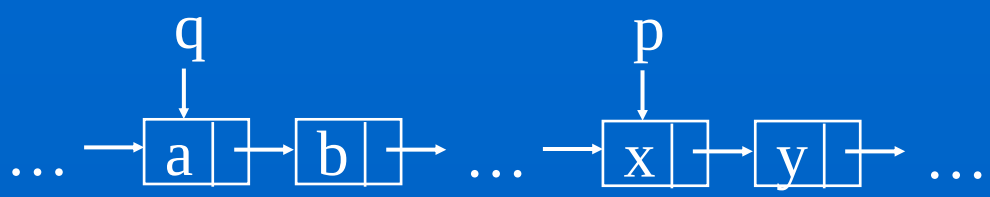
(a)



操作前



操作后



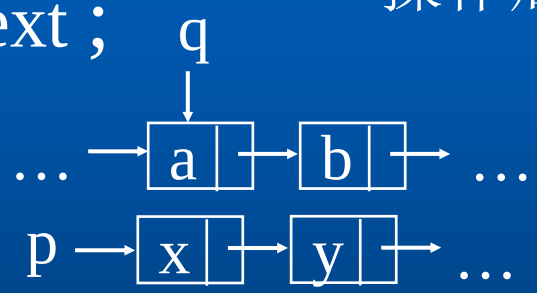
操作前

(b)

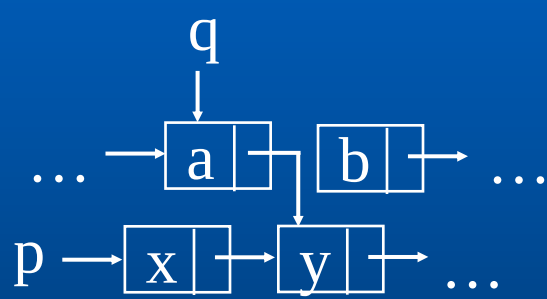
操作后

⑤ q->next=p->next；

(a)

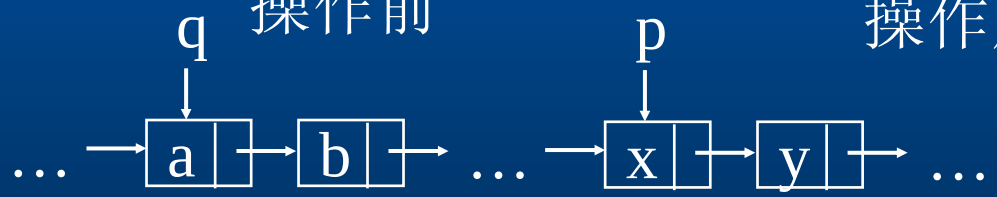


操作前



操作后

(b)



操作前

操作后

2.3.2 单线性链式的基本操作

1 建立单链表

假设线性表中结点的数据类型是整型，以32767作为结束标志。动态地建立单链表的常用方法有如下两种：头插入法，尾插入法。

(1) 头插入法建表

从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。即每次插入的结点都作为链表的第一个结点。

算法描述

```
LNode  *create_LinkList(void)
   /*  头插入法创建单链表,链表的头结点head作为返回值  */
  {   int data ;
      LNode *head, *p;
      head= (LNode  *) malloc( sizeof(LNode));
      head->next=NULL;      /*  创建链表的表头结点head  */
      while (1)
         {  scanf("%d", &data) ;
            if (data==32767)  break ;
            p= (LNode  *)malloc(sizeof(LNode));
            p–>data=data;    /*  数据域赋值  */
```

```
        p->next=head->next; head->next=p;
                /* 钩链，新创建的结点总是作为第一个结点 */
        }
    return (head);
}
```

(2) 尾插入法建表

头插入法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾，使其成为当前链表的尾结点。

算法描述

```
LNode  *create_LinkList(void)
    /*  尾插入法创建单链表,链表的头结点head作为返回值
*/
  {  int data ;
     LNode *head, *p, *q;
     head=p=(LNode  *)malloc(sizeof(LNode));
     p->next=NULL;       /*  创建单链表的表头结点head  */
     while (1)
        {   scanf("%d",& data);
          if (data==32767)  break ;
          q= (LNode  *)malloc(sizeof(LNode));
          q–>data=data;    /*  数据域赋值  */
          q–>next=p–>next;  p–>next=q; p=q ;
```

```
        /*钩链，新创建的结点总是作为最后一个结点*/
    }
  return (head);
}
```

无论是哪种插入方法，如果要插入建立的单线性链表的结点是n个，算法的时间复杂度均为O(n)。

对于单链表，无论是哪种操作，只要涉及到钩链(或重新钩链)，如果没有明确给出直接后继，钩链(或重新钩链)的次序必须是“先右后左”。

2 单链表的查找

(1) 按序号查找　取单链表中的第i个元素。

对于单链表，不能象顺序表中那样直接按序号i访问结点，而只能从链表的头结点出发，沿链域next逐个结点往下搜索，直到搜索到第i个结点为止。因此，链表不是随机存取结构。

设单链表的长度为n，要查找表中第i个结点，仅当1≦i≦n时，i的值是合法的。

算法描述

```
ElemType  Get_Elem(LNode *L ,  int  i)
  {   int j ;  LNode *p;
      p=L->next; j=1;     /*  使p指向第一个结点  */
      while  (p!=NULL && j<i)
         {   p=p–>next; j++; }       /*  移动指针p，j计数  */
      if  (j!=i)  return(-32768) ;
      else     return(p->data);
          /*   p为NULL 表示i太大；j>i表示i为0  */
  }
```

移动指针p的频度：

i<1时：0次；i∈[1,n]：i-1次；i>n：n次。

∴时间复杂度：O(n)。

(2) 按值查找

按值查找是在链表中，查找是否有结点值等于给定值**key**的结点？若有，则返回首次找到的值为**key**的结点的存储位置；否则返回**NULL**。查找时从开始结点出发，沿链表逐个将结点的值和给定值**key**作比较。

算法描述

```
LNode *Locate_Node(LNode *L，int key)
/*  在以L为头结点的单链表中查找值为key的第一个结点  */
   {  LNode *p=L->next;
      while ( p!=NULL&& p->data!=key)   p=p->next;
      if (p->data==key)  return p;
      else
         {   printf("所要查找的结点不存在!!\n");
           retutn(NULL);
         }
      }
```

算法的执行与形参key有关，平均时间复杂度为O(n)。

3 单链表的插入

插入运算是将值为e的新结点插入到表的第i个结点的位置上，即插入到a_{i-1}与a_i之间。因此，必须首先找到a_{i-1}所在的结点p，然后生成一个数据域为e的新结点q，q结点作为p的直接后继结点。

算法描述

```
void Insert_LNode(LNode *L, int i, ElemType e)
    /* 在以L为头结点的单链表的第i个位置插入值为e的结点 */
  {  int j=0; LNode *p, *q;
     p=L->next;
     while (p!=NULL&& j<i-1)
        { p=p->next; j++; }
```

```
    if (j!=i-1)    printf("i太大或i为0!!\n ");
    else
      {  q=(LNode
      *)malloc(sizeof(LNode));
        q->data=e;  q->next=p->next;
        p->next=q;
      }
  }
```

设链表的长度为n，合法的插入位置是1≦i≦n。算法的时间主要耗费移动指针p上，故时间复杂度亦为O(n)。

4 单链表的删除

(1) 按序号删除

删除单链表中的第i个结点。

为了删除第i个结点a_i，必须找到结点的存储地址。该存储地址是在其直接前趋结点a_{i-1}的next域中，因此，必须首先找到a_{i-1}的存储位置p，然后令p->next指向a_i的直接后继结点，即把a_i从链上摘下。最后释放结点a_i的空间，将其归还给“存储池”。

设单链表长度为n，则删去第i个结点仅当$1 \leqslant i \leqslant n$时是合法的。则当$i=n+1$时，虽然被删结点不存在，但其前趋结点却存在，是终端结点。故判断条件之一是p->next!=NULL。显然此算法的时间复杂度也是O(n)。

算法描述

```
void Delete_LinkList(LNode *L, int i)
   /* 删除以L为头结点的单链表中的第i个结点 */
  { int j=1; LNode *p, *q;
    p=L; q=L->next;
    while ( p->next!=NULL&& j<i)
      { p=q; q=q->next; j++; }
    if (j!=i)   printf("i太大或i为0!!\n ");
    else
      { p->next=q->next; free(q); }
  }
```

(2) 按值删除

删除单链表中值为**key**的第一个结点。

与按值查找相类似，首先要查找值为key的结点是否存在？若存在，则删除；否则返回**NULL**。

算法描述

```
void  Delete_LinkList(LNode *L, int key)
  /*  删除以L为头结点的单链表中值为key的第一个结点  */
  {    LNode *p=L, *q=L->next;
    while ( q!=NULL&& q->data!=key)
      { p=q; q=q->next;  }
    if (q->data==key)
      { p->next=q->next; free(q);  }
    else
      printf("所要删除的结点不存在!!\n");
  }
```

算法的执行与形参key有关，平均时间复杂度为O(n)。

从上面的讨论可以看出，链表上实现插入和删除运算，无需移动结点，仅需修改指针。解决了顺序表的插入或删除操作需要移动大量元素的问题。

变形之一：

删除单链表中值为key的所有结点。

与按值查找相类似，但比前面的算法更简单。

基本思想：从单链表的第一个结点开始，对每个结点进行检查，若结点的值为key，则删除之，然后检查下一个结点，直到所有的结点都检查。

算法描述

```
void Delete_LinkList_Node(LNode *L, int key)
  /* 删除以L为头结点的单链表中值为key的第一个结点 */
  {   LNode *p=L, *q=L->next;
    while ( q!=NULL)
      { if (q->data==key)
          { p->next=q->next; free(q);
        q=p->next; }
        else
          { p=q; q=q->next; }
      }
  }
```

变形之二：

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

与按值查找相类似，但比前面的算法更复杂。

基本思想：从单链表的第一个结点开始，对每个结点进行检查：检查链表中该结点的所有后继结点，只要有值和该结点的值相同，则删除之；然后检查下一个结点，直到所有的结点都检查。

算法描述

```
void  Delete_Node_value(LNode *L)
  /*  删除以L为头结点的单链表中所有值相同的结点  */
  {    LNode *p=L->next, *q, *ptr;
    while  ( p!=NULL)   /*  检查链表中所有结点
    */
      {   *q=p, *ptr=p->next;
        /*  检查结点p的所有后继结点ptr  */
        while (ptr!=NULL)
          { if (ptr->data==p->data)
              { q->next=ptr->next;  free(ptr);
                ptr=q->next;  }
            else  {  q=ptr;  ptr=ptr->next;   }
          }
```

```
        p=p->next;
    }
}
```

5 单链表的合并

设有两个有序的单链表，它们的头指针分别是La、Lb，将它们合并为以Lc为头指针的有序链表。合并前的示意图如图2-4所示。

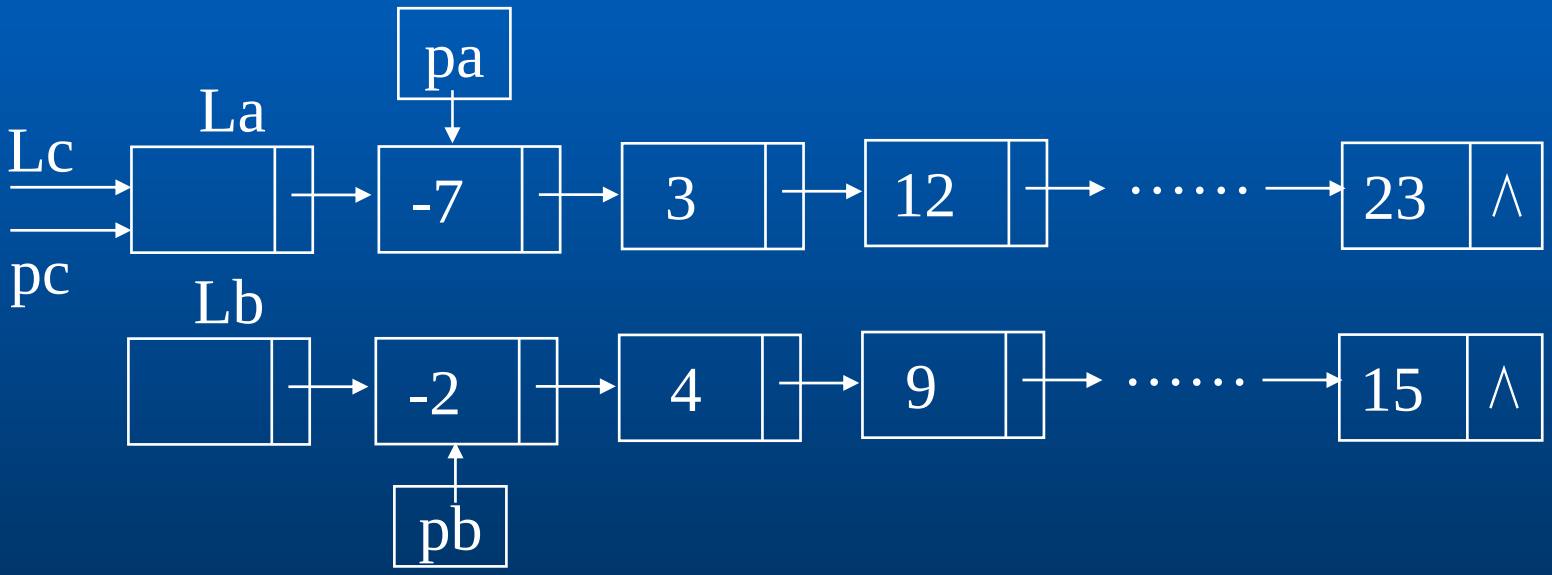


图2-4　两个有序的单链表La，Lb的初始状态

合并了值为-7，-2的结点后示意图如图2-5所示。

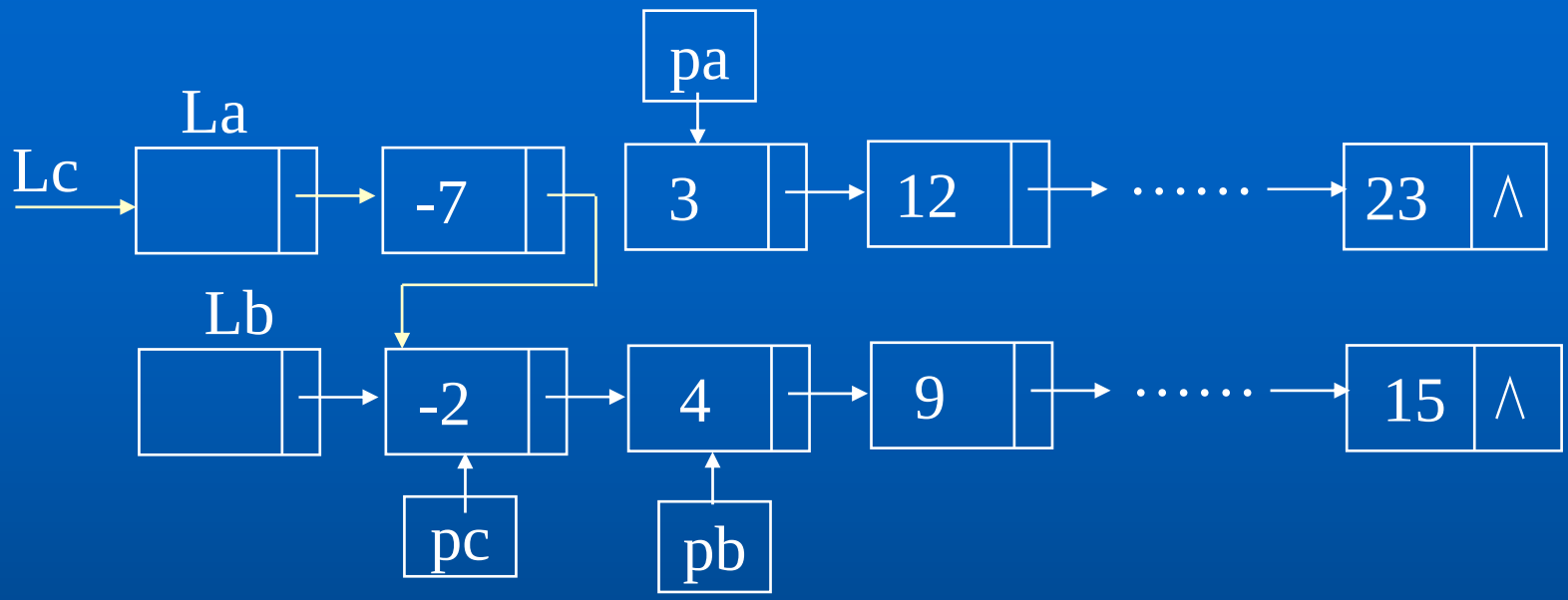


图2-5　合并了值为-7，-2的结点后的状态

算法说明

算法中pa，pb分别是待考察的两个链表的当前结点，pc是合并过程中合并的链表的最后一个结点。

算法描述

```
LNode *Merge_LinkList(LNode *La, LNode *Lb)
    /* 合并以La, Lb为头结点的两个有序单链表 */
  {  LNode *Lc, *pa , *pb , *pc, *ptr ;
     Lc=La ; pc=La ;  pa=La->next ; pb=Lb->next ;
      while (pa!=NULL   && pb!=NULL)
        { if (pa->data<pb->data)
            {  pc->next=pa ; pc=pa ;  pa=pa->next ;  }
          /* 将pa所指的结点合并，pa指向下一个结点 */
          if (pa->data>pb->data)
            {  pc->next=pb ; pc=pb ;  pb=pb->next ;  }
          /* 将pa所指的结点合并，pa指向下一个结点 */
```

```
            if (pa->data==pb->data)
               {  pc->next=pa; pc=pa;  pa=pa->next ;
                  ptr=pb;pb=pb->next;free(ptr);  }
               /*  将pa所指的结点合并，pb所指结点删除  */
          }
      if (pa!=NULL) pc->next=pa;
     else  pc->next=pb;    /*将剩余的结点链上*/
     free(Lb);
     return(Lc);
   }
```

算法分析

若La，Lb两个链表的长度分别是m，n，则链表合并的时间复杂度为O(m+n)。

2.3.3 循环链表

循环链表(Circular Linked List)：是一种头尾相接的链表。其特点是最后一个结点的指针域指向链表的头结点，整个链表的指针域链接成一个环。

从循环链表的任意一个结点出发都可以找到链表中的其它结点，使得表处理更加方便灵活。

图2-6是带头结点的单循环链表的示意图。

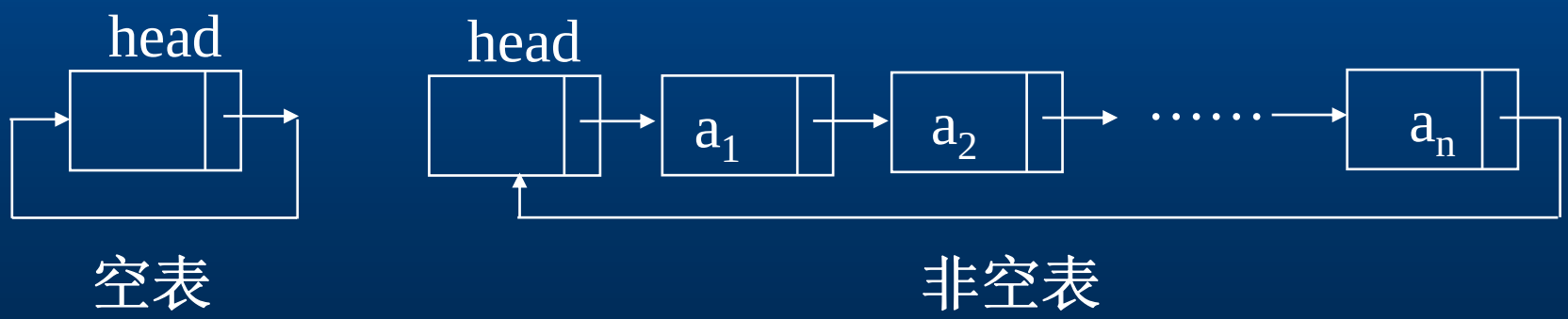


图2-6　单循环链表示意图

循环链表的操作

对于单循环链表，除链表的合并外，其它的操作和单线性链表基本上一致，仅仅需要在单线性链表操作算法基础上作以下简单修改：

⑴ 判断是否是空链表：**head->next==head** ；

⑵ 判断是否是表尾结点：**p->next==head** ；

2.4 双向链表

双向链表(Double Linked List)：指的是构成链表的每个结点中设立两个指针域：一个指向其直接前趋的指针域prior，一个指向其直接后继的指针域next。这样形成的链表中有两个方向不同的链，故称为双向链表。

和单链表类似，双向链表一般增加头指针也能使双链表上的某些运算变得方便。

将头结点和尾结点链接起来也能构成循环链表，并称之为双向循环链表。

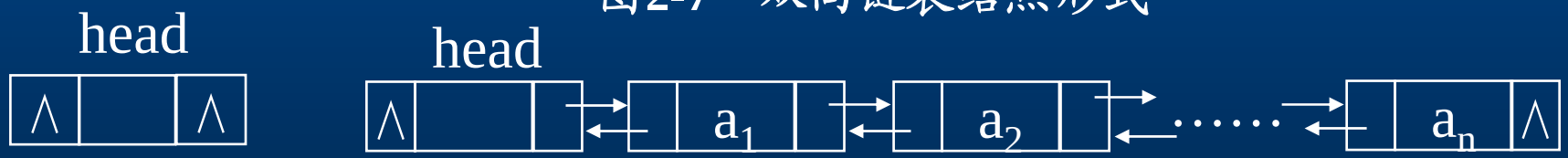
双向链表是为了克服单链表的单向性的缺陷而引入的。

1　双向链表的结点及其类型定义

双向链表的结点的类型定义如下。其结点形式如图2-7所示，带头结点的双向链表的形式如图2-8所示。

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

图2-7　双向链表结点形式



空双向链表　　非空双向链表

图2-8　带头结点的双向链表形式

双向链表结构具有对称性，设p指向双向链表中的某一结点，则其对称性可用下式描述：

```
(p->prior)->next=p=(p->next)->prior
;
```

结点p的存储位置存放在其直接前趋结点p->prior的直接后继指针域中，同时也存放在其直接后继结点p->next的直接前趋指针域中。

2 双向链表的基本操作

(1) 双向链表的插入 将值为e的结点插入双向链表中。插入前后链表的变化如图2-9所示。

图2-9 双向链表的插入

① 插入时仅仅指出直接前驱结点，钩链时必须注意先后次序是：“先右后左”。部分语句组如下：

```
S=(DulNode
*)malloc(sizeof(DulNode));
S->data=e;
S->next=p->next;  p->next->prior=S;
p->next=S; S->prior=p;   /* 钩链次序非常重要  */
```

② 插入时同时指出直接前驱结点p和直接后继结点q，钩链时无须注意先后次序。部分语句组如下：

```
S=(DulNode *)malloc(sizeof(DulNode));
S->data=e;
p->next=S;      S->next=q;
S->prior=p;       q->prior=S;
```

(2) 双向链表的结点删除

设要删除的结点为p，删除时可以不引入新的辅助指针变量，可以直接先断链，再释放结点。部分语句组如下：

```
p->prior->next=p->next;
p->next->prior=p->prior;
free(p);
```

注意：

与单链表的插入和删除操作不同的是，在双向链表中插入和删除必须同时修改两个方向上的指针域的指向。

2.5 一元多项式的表示和相加

1 一元多项式的表示

一元多项式 $p(x)=p_0+p_1x+p_2x^2+\ldots+p_nx^n$，由n+1个系数唯一确定。则在计算机中可用线性表$(p_0, p_1, p_2, \ldots, p_n)$表示。既然是线性表，就可以用顺序表和链表来实现。两种不同实现方式的元素类型定义如下：

(1) 顺序存储表示的类型

```
typedef struct
  { float coef;   /*系数部分*/
    int   expn;   /*指数部分*/
  } ElemType ;
```

(2) 链式存储表示的类型

```
typedef struct ploy
  { float coef ;   /*系数部分*/
    int   expn ;   /*指数部分*/
    struct ploy  *next ;
  } Ploy ;
```

2 一元多项式的相加

不失一般性，设有两个一元多项式：

$P(x)=p_0+p_1x+p_2x^2+\ \dots\ +p_nx^n$，

$Q(x)=q_0+q_1x+q_2x^2+\ \dots\ +q_mx^m\quad (m<n)$

$R(x)=P(x)+Q(x)$

$R(x)$由线性表$R((p_0+q_0)，(p_1+q_1)，(p_2+q_2)，\ \dots，(p_m+q_m)，\ \dots，\ p_n)$唯一表示。

(1) 顺序存储表示的相加

线性表的定义

```
typedef struct
  {  ElemType a[MAX_SIZE];
     int   length;
  }Sqlist ;
```

用顺序表示的相加非常简单。访问第5项可直接访问：L.a[4].coef， L.a[4].expn

(2) 链式存储表示的相加

当采用链式存储表示时，根据结点类型定义，凡是系数为0的项不在链表中出现，从而可以大大减少链表的长度。

一元多项式相加的实质是：

- 指数不同： 是链表的合并。
- 指数相同： 系数相加，和为0，去掉结点，和不为0，修改结点的系数域。

算法之一：

就在原来两个多项式链表的基础上进行相加，相加后原来两个多项式链表就不在存在。当然再要对原来两个多项式进行其它操作就不允许了。

算法描述

```
Ploy *add_ploy(ploy *La, ploy *Lb)
  /* 将以La，Lb为头指针表示的一元多项式相加 */
  {  ploy *Lc,*pc,*pa,*pb,*ptr;  float x;
     Lc=pc=La; pa=La->next; pb=Lb->next;
     while (pa!=NULL&&pb!=NULL)
        { if (pa->expn<pb->expn)
            { pc->next=pa; pc=pa; pa=pa->next;  }
              /* 将pa所指的结点合并，pa指向下一个结点 */
          if (pa->expn>pb->expn)
            { pc->next=pb; pc=pb; pb=pb->next;  }
              /* 将pb所指的结点合并，pb指向下一个结点 */
```

```
else
   {  x=pa->coef+pb->coef;
      if (abs(x)<=1.0e-6)
          /*  如果系数和为0，删除两个结点  */
         {  ptr=pa; pa=pa->next; free(ptr);
            ptr=pb; pb=pb->next; free(ptr); }
      else     /*  如果系数和不为0，修改其中一个结点的系数域，删除另一个结点  */
          {   pc->next=pa; pa->coef=x;
             pc=pa; pa=pa->next;
             ptr=pb; pb=pb->next; free(pb);
          }
```

```
        }
      }     /* end of while */
  if (pa==NULL) pc->next=pb;
  else pc->next=pa;
  return (Lc);
}
```

算法之二：

对两个多项式链表进行相加，生成一个新的相加后的结果多项式链表，原来两个多项式链表依然存在，不发生任何改变，如果要再对原来两个多项式进行其它操作也不影响。

算法描述

```
Ploy *add_ploy(ploy *La, ploy *Lb)
  /* 将以La，Lb为头指针表示的一元多项式相加，生成一个新的结果多项式 */
  {  ploy *Lc,*pc,*pa,*pb,*p;  float x;
     Lc=pc=(ploy *)malloc(sizeof(ploy));
     pa=La->next;pb=Lb->next;
     while (pa!=NULL&&pb!=NULL)
        { if (pa->expn<pb->expn)
            { p=(ploy *)malloc(sizeof(ploy));
              p->coef=pa->coef;p->expn=pa->expn;
              p->next=NULL;
```

```
            /* 生成一个新的结果结点并赋值 */
        pc->next=p; pc=p; pa=pa->next;
    }
        /* 生成的结点插入到结果链表的最后，pa指向下一个结点 */
if (pa->expn>pb->expn)
  { p=(ploy *)malloc(sizeof(ploy));
    p->coef=pb->coef; p->expn=pb->expn;
    p->next=NULL;
          /* 生成一个新的结果结点并赋值 */
     pc->next=p; pc=p; pb=pb->next;
    }    /* 生成的结点插入到结果链表的最后，pb指向下一个结点 */
```

```
if (pa->expn==pb->expn)
   {  x=pa->coef+pb->coef;
      if (abs(x)<=1.0e-6)
          /*  系数和为0，pa, pb分别直接后继结点  */
         {  pa=pa->next;pb=pb->next;  }
      else   /*  若系数和不为0，生成的结点插入到结果链表的最后，  pa, pb分别直接后继结点  */
          {  p=(ploy *)malloc(sizeof(ploy));
             p->coef=x;p->expn=pb->expn;
             p->next=NULL;
                  /*  生成一个新的结果结点并赋值  */
             pc->next=p;pc=p;
             pa=pa->next;pb=pb->next;
```

```
                    }
                }
        }    /* end of while */
if  (pb!=NULL)
    while(pb!=NULL)
       {   p=(ploy *)malloc(sizeof(ploy)) ;
           p->coef=pb->coef ; p->expn=pb->expn ;
           p->next=NULL ;
                 /*  生成一个新的结果结点并赋值  */
             pc->next=p ; pc=p ; pb=pb->next ;
       }
```

```
    if (pa!=NULL)
       while(pa!=NULL)
          {  p=(ploy *)malloc(sizeof(ploy));
             p->coef=pb->coef;p->expn=pa->expn;
             p->next=NULL;
                  /* 生成一个新的结果结点并赋值 */
               pc->next=p;pc=p;pa=pa->next;
          }
    return (Lc);
}
```

习 题 二

1　简述下列术语：线性表，顺序表，链表。

2　何时选用顺序表，何时选用链表作为线性表的存储结构合适?各自的主要优缺点是什么?

3　在顺序表中插入和删除一个结点平均需要移动多少个结点?具体的移动次数取决于哪两个因素?

4　链表所表示的元素是否有序?如有序，则有序性体现于何处?链表所表示的元素是否一定要在物理上是相邻的?有序表的有序性又如何理解?

5　设顺序表L是递增有序表，试写一算法，将x插入到L中并使L仍是递增有序表。

6　写一求单链表的结点数目ListLength(L)的算法。

7　写一算法将单链表中值重复的结点删除，使所得的结果链表中所有结点的值均不相同。

8　写一算法从一给定的向量A删除值在x到y(x≤y)之间的所有元素(注意：x和y是给定的参数，可以和表中的元素相同，也可以不同)。

9　设A和B是两个按元素值递增有序的单链表，写一算法将A和B归并为按按元素值递减有序的单链表C，试分析算法的时间复杂度。

第3章 栈和队列

栈和队列是两种应用非常广泛的数据结构，它们都来自线性表数据结构，都是“操作受限”的线性表。

栈在计算机的实现有多种方式：

- 硬堆栈：利用CPU中的某些寄存器组或类似的硬件或使用内存的特殊区域来实现。这类堆栈容量有限，但速度很快；
- 软堆栈：这类堆栈主要在内存中实现。堆栈容量可以达到很大。在实现方式上，又有动态方式和静态方式两种。

本章将讨论栈和队列的基本概念、存储结构、基本操作以及这些操作的具体实现。

3.1 栈

3.1.1 栈的基本概念

1　栈的概念

栈(Stack)：是限制在表的一端进行插入和删除操作的线性表。又称为后进先出LIFO (Last In First Out)或先进后出FILO (First In Last Out)线性表。

栈顶(Top)：允许进行插入、删除操作的一端，又称为表尾。用栈顶指针(top)来指示栈顶元素。

栈底(Bottom)：是固定端，又称为表头。

空栈：当表中没有元素时称为空栈。

设栈$S=(a_1, a_2, \ldots a_n)$，则a_1称为栈底元素，a_n为栈顶元素，如图3-1所示。

栈中元素按$a_1, a_2, \ldots a_n$的次序进栈，退栈的第一个元素应为栈顶元素。即栈的修改是按后进先出的原则进行的。

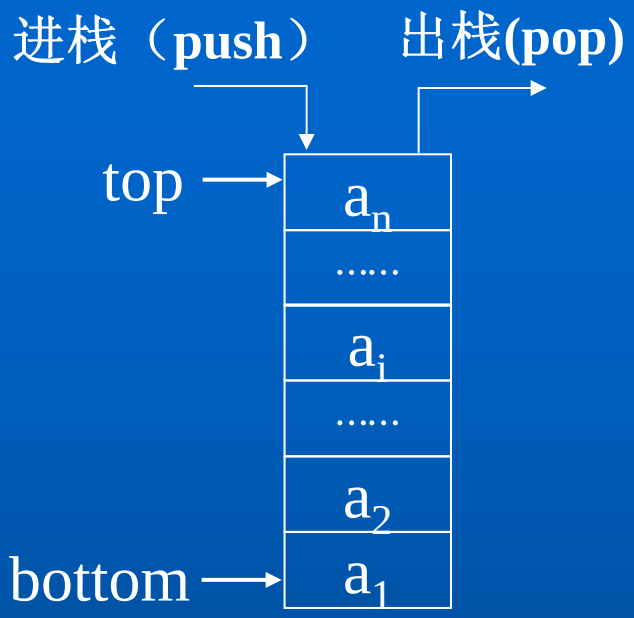


图3-1 顺序栈示意图

2 栈的抽象数据类型定义

ADT Stack{

数据对象：**D ={ a_i|a_i∈ElemSet, i=1,2,…,n， n≥0 }**

数据关系：**R ={<a_{i-1}, a_i>|a_{i-1}， a_i∈D, i=2,3,…,n }**

基本操作：初始化、进栈、出栈、取栈顶元素等

} ADT Stack

3.1.2 栈的顺序存储表示

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

3.1.2.1 栈的动态顺序存储表示

采用动态一维数组来存储栈。所谓动态，指的是栈的大小可以根据需要增加。

- 用bottom表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。用top(称为栈顶指针)指示当前栈顶位置。
- 用top=bottom作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。
- 结点进栈：首先将数据元素保存到栈顶(top所指的当前位置)，然后执行top加1，使top指向栈顶的下一个存储位置;

◆ **结点出栈**：首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。

图3-2是一个动态栈的变化示意图。

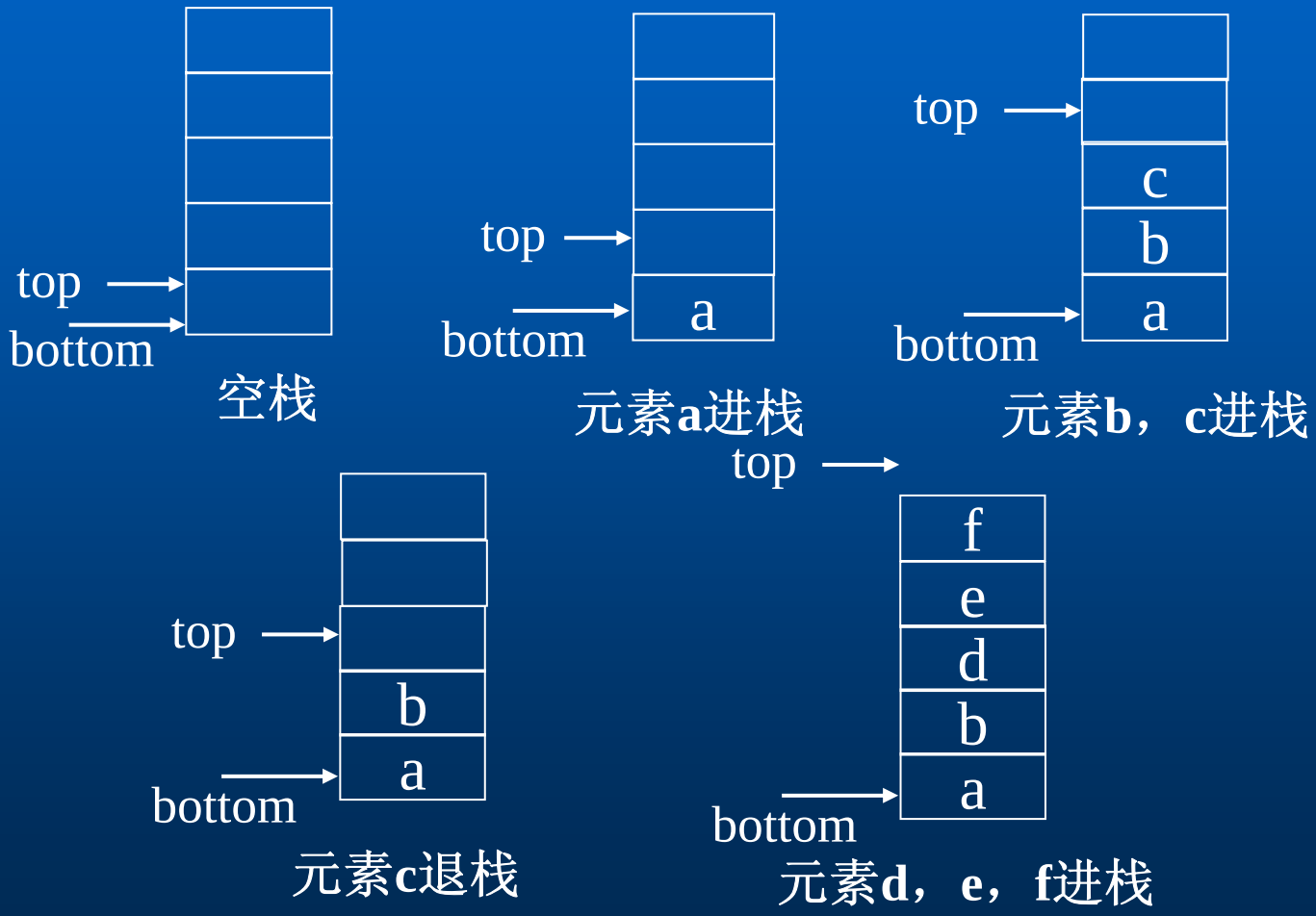


图3-2 (动态)堆栈变化示意图

基本操作的实现

1 栈的类型定义

```
#define STACK_SIZE 100    /* 栈初始向量大小 */
#define STACKINCREMENT 10   /* 存储空间分配增量 */
#typedef int ElemType;
typedef struct sqstack
   {  ElemType *bottom;    /* 栈不存在时值为NULL */
      ElemType *top;      /* 栈顶指针 */
      int  stacksize;     /* 当前已分配空间，以元素为单位 */
   }SqStack;
```

2 栈的初始化

```
Status Init_Stack(void)
  {  SqStack  S ;
    S.bottom=(ElemType
    *)malloc(STACK_SIZE
    *sizeof(ElemType));
    if (! S.bottom) return  ERROR;
    S.top=S.bottom ;    /* 栈空时栈顶和栈底指针
    相同  */
    S. stacksize=STACK_SIZE;
    return OK ;
  }
```

3 压栈(元素进栈)

```
Status push(SqStack S , ElemType  e)
   {  if  (S.top-S.bottom>=S. stacksize-1)
         {   S.bottom=(ElemType *)realloc((S.
         STACKINCREMENT+STACK_SIZE)
         *sizeof(ElemType));   /*  栈满，追加存储空间   */
            if (! S.bottom)  return  ERROR;
            S.top=S.bottom+S. stacksize ;
            S. stacksize+=STACKINCREMENT ;
         }
      *S.top=e;  S.top++ ;/*   栈顶指针加1，e成为新的栈顶 */
      return OK;
   }
```

4 弹栈(元素出栈)

```
Status pop( SqStack   S, ElemType  *e )
  /*弹出栈顶元素*/
  {   if ( S.top== S.bottom )
        return ERROR ;     /*  栈空，返回失败标志
        */
    S.top-- ; e=*S. top ;
    return  OK ;
  }
```

3.1.2.2 栈的静态顺序存储表示

采用静态一维数组来存储栈。

栈底固定不变的，而栈顶则随着进栈和退栈操作变化的，

- 栈底固定不变的；栈顶则随着进栈和退栈操作而变化，用一个整型变量top(称为栈顶指针)来指示当前栈顶位置。
- 用top=0表示栈空的初始状态，每次top指向栈顶在数组中的存储位置。
- 结点进栈：首先执行top加1，使top指向新的栈顶位置，然后将数据元素保存到栈顶(top所指的当前位置)。

◆ **结点出栈**：首先把top指向的栈顶元素取出，然后执行top减1，使top指向新的栈顶位置。

若栈的数组有Maxsize个元素，则top=Maxsize-1时栈满。图3-3是一个大小为5的栈的变化示意图。

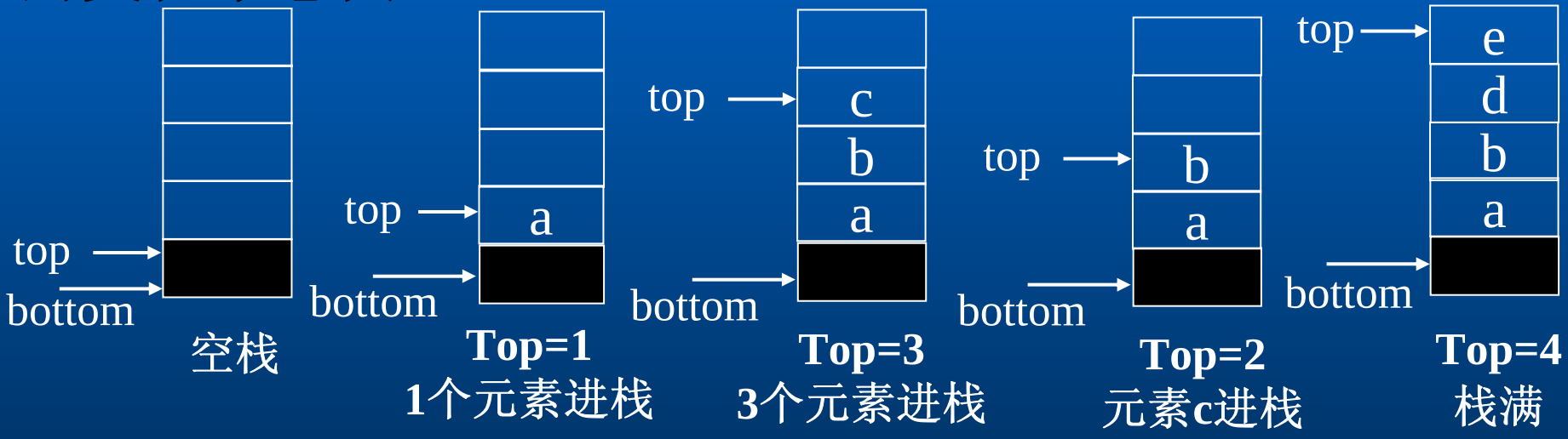


图3-3 静态堆栈变化示意图

基本操作的实现

1　栈的类型定义

```
# define MAX_STACK_SIZE 100      /* 栈向量大小 */
# typedef int ElemType;
typedef struct sqstack
  { ElemType
  stack_array[MAX_STACK_SIZE];
    int top;
  }SqStack;
```

2 栈的初始化

```
SqStack Init_Stack(void)
  {   SqStack S;
    S.bottom=S.top=0; return(S);
  }
```

3 压栈(元素进栈)

```
Status push(SqStack S , ElemType  e)
   /*  使数据元素e进栈成为新的栈顶  */
  {  if  (S.top==MAX_STACK_SIZE-1)
        return  ERROR;      /*  栈满，返回错误标志
        */
     S.top++ ;      /*  栈顶指针加1  */
     S.stack_array[S.top]=e ;  /* e成为新的栈
     顶  */
     return OK;          /*  压栈成功    */
  }
```

4 弹栈(元素出栈)

```
Status pop( SqStack S, ElemType *e )
    /*弹出栈顶元素*/
  {  if ( S.top==0 )
       return ERROR ;      /* 栈空，返回错误标志
       */
     *e=S.stack_array[S.top] ;
     S.top-- ;
     return OK ;
  }
```

当栈满时做进栈运算必定产生空间溢出，简称“上溢”。上溢是一种出错状态，应设法避免。

当栈空时做退栈运算也将产生溢出，简称“下溢”。下溢则可能是正常现象，因为栈在使用时，其初态或终态都是空栈，所以下溢常用来作为控制转移的条件。

3.1.3 栈的链式存储表示

1 栈的链式表示

栈的链式存储结构称为链栈，是运算受限的单链表。其插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针**top**就是链表的头指针。图**3-4**是栈的链式存储表示形式。

链栈的结点类型说明如下：

```
typedef  struct  Stack_Node
   {  ElemType   data ;
      struct Stack_Node  *next
      ;
   } Stack_Node ;
```

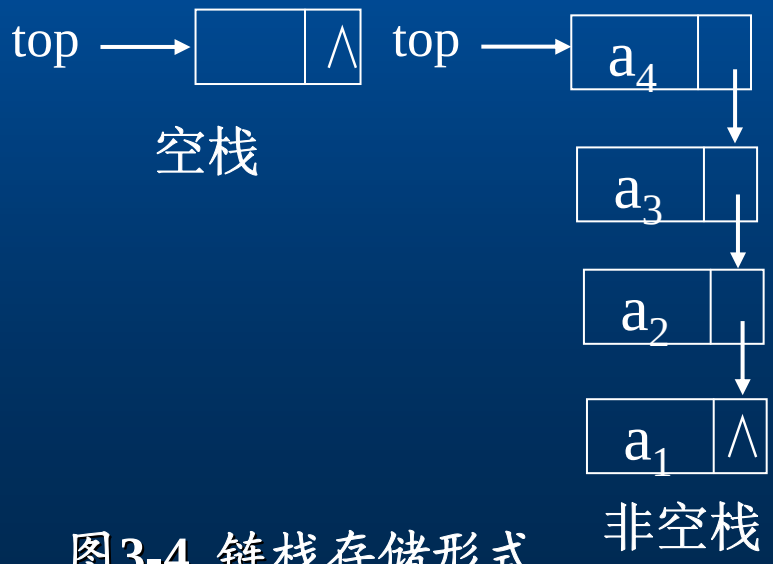


图3-4 链栈存储形式

2 链栈基本操作的实现

(1) 栈的初始化

```
Stack_Node *Init_Link_Stack(void)
  {   Stack_Node *top;
    top=(Stack_Node
    *)malloc(sizeof(Stack_Node ));
    top->next=NULL;
    return(top);
  }
```

(2) 压栈(元素进栈)

```
Status push(Stack_Node *top , ElemType  e)
   {  Stack_Node  *p ;
      p=(Stack_Node  *)malloc(sizeof(Stack_Node)) ;
      if (!p)  return  ERROR;
         /*  申请新结点失败，返回错误标志  */
      p->data=e ;
      p->next=top->next  ;
      top->next=p ;    /*  钩链  */
      return OK;
   }
```

(3) 弹栈(元素出栈)

```
Status pop(Stack_Node *top , ElemType *e)
   /*  将栈顶元素出栈  */
   {  Stack_Node *p ;
      ElemType e ;
      if (top->next==NULL )
         return ERROR ;   /* 栈空，返回错误标志   */
      p=top->next ; e=p->data ;   /*  取栈顶元素  */
      top->next=p->next ;   /*  修改栈顶指针  */
      free(p) ;
      return OK ;
   }
```

3.2 栈的应用

由于栈具有的“后进先出”的固有特性，因此，栈成为程序设计中常用的工具和数据结构。以下是几个栈应用的例子。

3.2.1 数制转换

十进制整数N向其它进制数d(二、八、十六)的转换是计算机实现计算的基本问题。

转换法则：该转换法则对应于一个简单算法原理：

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

采用静态顺序栈方式实现

```
void conversion(int  n , int  d)
  /*将十进制整数N转换为d(2或8)进制数*/
  {  SqStack  S ;    int k, *e ;
     S=Init_Stack();
     while  (n>0)  {  k=n%d ; push(S , k) ; n=n/d ;   }
      /*  求出所有的余数，进栈  */
     while  (S.top!=0)     /*  栈不空时出栈，输出  */
        {   pop(S, e) ;
            printf("%1d" , *e) ;
        }
  }
```

3.2.2 括号匹配问题

在文字处理软件或编译程序设计时，常常需要检查一个字符串或一个表达式中的括号是否相匹配?

匹配思想：从左至右扫描一个字符串(或表达式)，则每个右括号将与最近遇到的那个左括号相匹配。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。

算法思想：设置一个栈，当读到左括号时，左括号进栈。当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；否则匹配失败，返回FLASE。

算法描述

```
#define TRUE 0
#define FLASE -1
SqStack S;
S=Init_Stack(); /*堆栈初始化*/
int Match_Brackets( )
  {  char ch,x;
    scanf("%c",&ch);
     while (asc(ch)!=13)
```

```
{  if ((ch=='(')||(ch=='[')) push(S , ch) ;
   else if (ch==']')
      {  x=pop(S) ;
         if (x!='[')
               {  printf("'['括号不匹配") ;
                  return FLASE ; }  }
   else if (ch==')')
       {  x=pop(S) ;
          if (x!='(')
             { printf("'('括号不匹配") ;
               return FLASE ;}
       }
}
```

```
    if (S.top!=0)
      {   printf("括号数量不匹配！");
          return FLASE ;
      }
    else return TRUE ;
}
```

3.2.2 栈与递归调用的实现

栈的另一个重要应用是在程序设计语言中实现递归调用。

递归调用：一个函数(或过程)直接或间接地调用自己本身，简称递归(Recursive)。

递归是程序设计中的一个强有力的工具。因为递归函数结构清晰，程序易读，正确性很容易得到证明。

为了使递归调用不至于无终止地进行下去，实际上有效的递归调用函数(或过程)应包括两部分：递推规则(方法)，终止条件。

例如：求n!

$$Fact(n)=\begin{cases}1 & \text{当n=0时} \quad \text{终止条件}\\ n*fact(n-1) & \text{当n>0时} \quad \text{递推规则}\end{cases}$$

为保证递归调用正确执行，系统设立一个“递归工作栈”，作为整个递归调用过程期间使用的数据存储区。

每一层递归包含的信息如：参数、局部变量、上一层的返回地址构成一个“工作记录”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

从被调函数返回调用函数的一般步骤：

(1) 若栈为空，则执行正常返回。

(2) 从栈顶弹出一个工作记录。

(3) 将“工作记录”中的参数值、局部变量值赋给相应的变量；读取返回地址。

(4) 将函数值赋给相应的变量。

(5) 转移到返回地址。

3.3 队 列

3.3.1 队列及其基本概念

1 队列的基本概念

队列(Queue)：也是运算受限的线性表。是一种先进先出(First In First Out，简称FIFO)的线性表。只允许在表的一端进行插入，而在另一端进行删除。

队首(front)：允许进行删除的一端称为队首。

队尾(rear)：允许进行插入的一端称为队尾。

例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。

队列中没有元素时称为空队列。在空队列中依次加入元素$a_1, a_2, \ldots, a_n$之后，a_1是队首元素，a_n是队尾元素。显然退出队列的次序也只能是$a_1, a_2, \ldots, a_n$，即队列的修改是依先进先出的原则进行的，如图3-5所示。

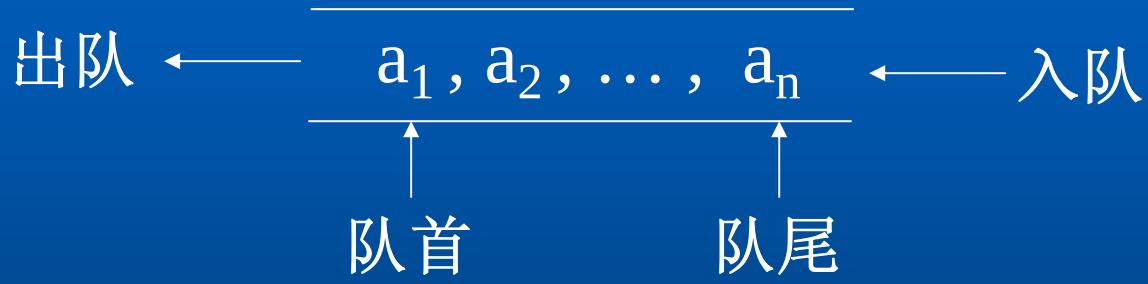


图3-5　队列示意图

2 队列的抽象数据类型定义

ADT Queue{

数据对象：**D ={ a_i|a_i∈ElemSet, i=1, 2, …, n, n >= 0 }**

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i=2,3,\ldots,n\}$

约定a_1端为队首，a_n端为队尾。

基本操作：

Create()：创建一个空队列；

EmptyQue()：若队列为空，则返回**true**，否则返回**flase**；

……

InsertQue(x)：向队尾插入元素**x**；

DeleteQue(x)：删除队首元素**x**；

} ADT Queue

3.3.2 队列的顺序表示和实现

利用一组连续的存储单元(一维数组) 依次存放从队首到队尾的各个元素，称为顺序队列。

对于队列，和顺序栈相类似，也有动态和静态之分。本部分介绍的是静态顺序队列，其类型定义如下：

```
#define  MAX_QUEUE_SIZE   100
typedef  struct  queue
   {  ElemType   Queue_array[MAX_QUEUE_SIZE] ;
      int   front ;
      int  rear ;
   }SqQueue;
```

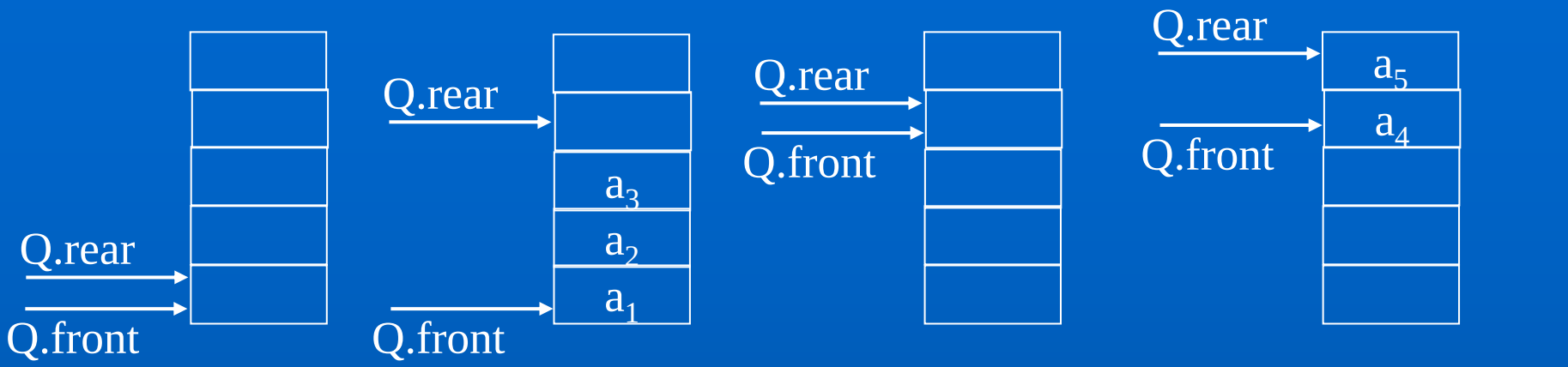
3.3.2.1 队列的顺序存储结构

设立一个队首指针**front**，一个队尾指针**rear**，分别指向队首和队尾元素。

- 初始化：**front=rear=0**。
- 入队：将新元素插入**rear**所指的位置，然后**rear**加**1**。
- 出队：删去**front**所指的元素，然后加**1**并返回被删元素。
- 队列为空：**front=rear**。
- 队满：**rear=MAX_QUEUE_SIZE-1**或**front=rear**。

在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。

顺序队列中存在“假溢出”现象。因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为假溢出。如图3-6所示是数组大小为5的顺序队列中队首、队尾指针和队列中元素的变化情况。



(a) 空队列　**(b)** 入队**3**个元素　**(c)** 出队**3**个元素　**(d)** 入队**2**个元素

图3-6　队列示意图

3.3.2.2 循环队列

为充分利用向量空间，克服上述“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列(Circular Queue)。

在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。只不过当队首、队尾指针指向向量上界(MAX_QUEUE_SIZE-1)时，其加1操作的结果是指向向量的下界0。

这种循环意义下的加1操作可以描述为：

```
if  (i+1==MAX_QUEUE_SIZE)   i=0;
else     i++ ;
```

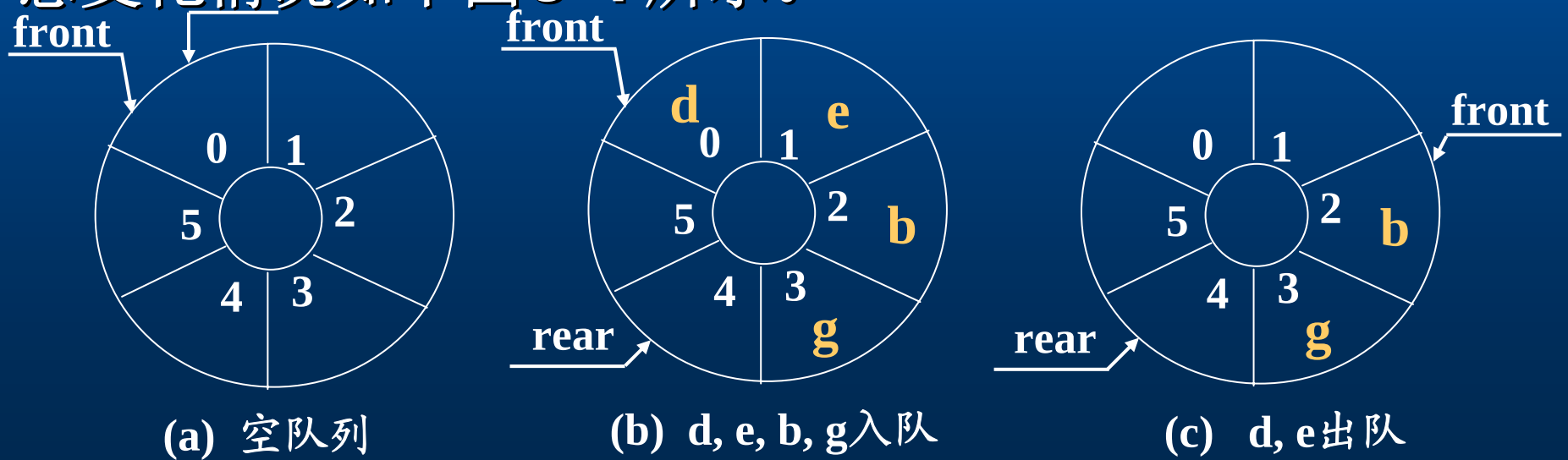
其中： i代表队首指针(front)或队尾指针(rear)

用模运算可简化为：

```
i=(i+1)%MAX_QUEUE_SIZE;
```

显然，为循环队列所分配的空间可以被充分利用，除非向量空间真的被队列元素全部占用，否则不会上溢。因此，真正实用的顺序队列是循环队列。

例：设有循环队列QU[0，5]，其初始状态是front=rear=0，各种操作后队列的头、尾指针的状态变化情况如下图3-7所示。



(a) 空队列　(b) d, e, b, g入队　(c) d, e出队

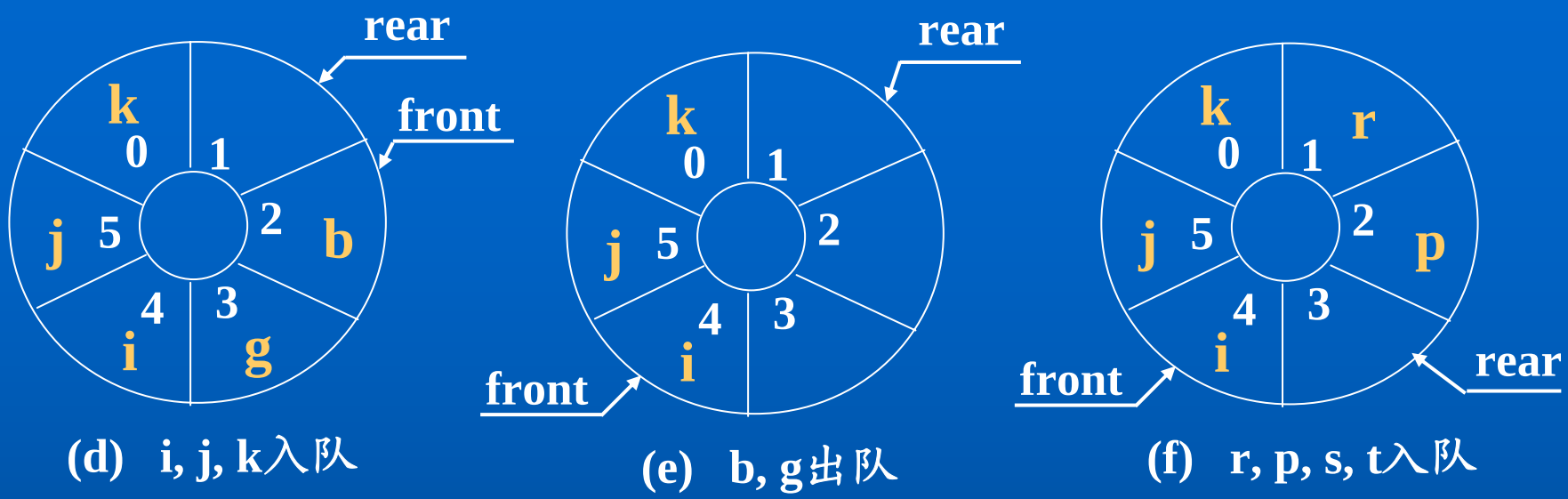


图3-7 循环队列操作及指针变化情况

入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，无法通过front=rear来判断队列“空”还是“满”。解决此问题的方法是：约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满。即：

- rear所指的单元始终为空。

- 循环队列为空：front=rear 。
- 循环队列满：

(rear+1)%MAX_QUEUE_SIZE =front。

循环队列的基本操作

1 循环队列的初始化

```
SqQueue Init_CirQueue(void)
  { SqQueue Q;
    Q.front=Q.rear=0; return(Q);
  }
```

2 入队操作

```
Status Insert_CirQueue(SqQueue  Q , ElemType  e)
   /*   将数据元素e插入到循环队列Q的队尾   */
   {  if  ((Q.rear+1)%MAX_QUEUE_SIZE== Q.front)
        return  ERROR;     /* 队满，返回错误标志   */
        Q.Queue_array[Q.rear]=e ;  /*   元素e入队   */
        Q.rear=(Q.rear+1)% MAX_QUEUE_SIZE ;
              /* 队尾指针向前移动 */
        return OK;       /* 入队成功   */
   }
```

3 出队操作

```
Status Delete_CirQueue(SqQueue  Q, ElemType  *x )
    /*   将循环队列Q的队首元素出队   */
   {   if  (Q.front+1== Q.rear)
          return ERROR ;      /* 队空，返回错误标志   */
       *x=Q.Queue_array[Q.front] ;  /* 取队首元素 */
       Q.front=(Q.front+1)% MAX_QUEUE_SIZE ;
          /* 队首指针向前移动 */
       return OK ;
   }
```

3.3.3 队列的链式表示和实现

1 队列的链式存储表示

队列的链式存储结构简称为链队列，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。

需要两类不同的结点：数据元素结点，队列的队首指针和队尾指针的结点，如图3-8所示。

数据元素结点类型定义：

```
typedef struct Qnode
  { ElemType   data ;
    struct Qnode  *next ;
  }QNode ;
```

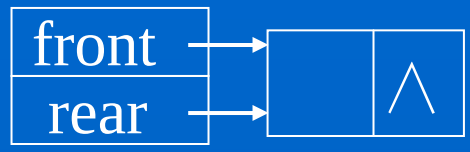
图3-8 链队列结点示意图

指针结点类型定义：

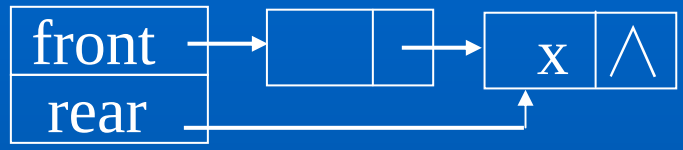
```
typedef struct link_queue
  {  QNode  *front , *rear ;
  }Link_Queue ;
```

2　链队运算及指针变化

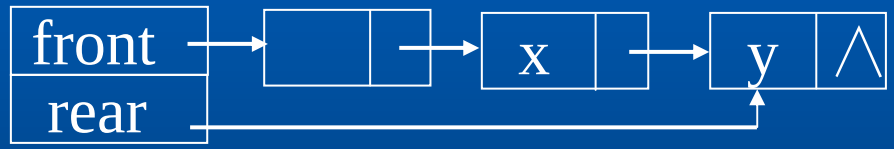
链队的操作实际上是单链表的操作，只不过是删除在表头进行，插入在表尾进行。插入、删除时分别修改不同的指针。链队运算及指针变化如图3-9所示。



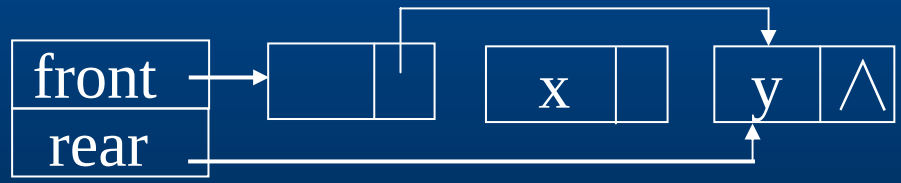
(a) 空队列



(b) x入队



(c) y再入队



(d) x出队

图3-9 队列操作及指针变化

3　链队列的基本操作

(1) 链队列的初始化

```
LinkQueue *Init_LinkQueue(void)
   {  LinkQueue  *Q ;  QNode  *p ;
      p=(QNode *)malloc(sizeof(QNode)) ; /* 开辟头结点 */
      p->next=NULL ;
      Q=(LinkQueue  *)malloc(sizeof(LinkQueue)) ;
            /*  开辟链队的指针结点  */
      Q.front=Q.rear=p ;
      return(Q) ;
   }
```

(2) 链队列的入队操作

在已知队列的队尾插入一个元素e，即修改队尾指针(Q.rear)。

```
Status  Insert_CirQueue(LinkQueue  *Q , ElemType  e)
        /*   将数据元素e插入到链队列Q的队尾   */
  {   p=(QNode *)malloc(sizeof(QNode)) ;
      if (!p)  return  ERROR;
         /*  申请新结点失败，返回错误标志 */
      p->data=e ; p->next=NULL ;      /*  形成新结点 */
      Q.rear->next=p ;  Q.rear=p ;  /*  新结点插入到队尾  */
      return OK;
  }
```

(3) 链队列的出队操作

```
Status  Delete_LinkQueue(LinkQueue  *Q, ElemType *x)
    {   QNode *p ;
        if  (Q.front==Q.rear)  return ERROR ;    /*  队空  */
        p=Q.front->next ;   /*    取队首结点    */
        *x=p->data ;
        Q.front->next=p->next ;      /*  修改队首指针  */
        if  (p==Q.rear)  Q.rear=Q.front ;
                /*  当队列只有一个结点时应防止丢失队尾指针  */
          free(p) ;
        return OK ;
    }
```

(4) 链队列的撤消

```
void  Destroy_LinkQueue(LinkQueue  *Q )
  /*  将链队列Q的队首元素出队  */
   {   while  (Q.front!=NULL)
         {  Q.rear=Q.front->next;
              /*  令尾指针指向队列的第一个结点   */
            free(Q.front);     /*  每次释放一个结点  */
              /*  第一次是头结点，以后是元素结点  */
            Q.ront=Q.rear;
         }
   }
```

习 题 三

1 设有一个栈，元素进栈的次序为a，b，c。问经过栈操作后可以得到哪些输出序列？

2 循环队列的优点是什么?如何判断它的空和满?

3 设有一个静态顺序队列，向量大小为MAX，判断队列为空的条件是什么？队列满的条件是什么？

4 设有一个静态循环队列，向量大小为MAX，判断队列为空的条件是什么？队列满的条件是什么？

5 利用栈的基本操作，写一个返回栈S中结点个数的算法int StackSize(SeqStack S)，并说明S为何不作为指针参数的算法?

6 一个双向栈S是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈设计初始化InitStack(S)，入栈Push(S,i,x)，出栈Pop(S,i,x)算法，其中i为0或1，用以表示栈号。

7 设Q[0,6]是一个静态顺序队列，初始状态为front=rear=0，请画出做完下列操作后队列的头尾指针的状态变化情况，若不能入对，请指出其元素，并说明理由。

a, b, c, d入队

a, b, c出队

i , j , k , l , m入队

d, i出队

n, o, p, q, r入队

8 假设Q[0,5]是一个循环队列，初始状态为front=rear=0，请画出做完下列操作后队列的头尾指针的状态变化情况，若不能入对，请指出其元素，并说明理由。

d, e, b, g, h入队

d, e出队

i , j , k , l , m入队

b出队

n, o, p, q, r入队

第4章　串

在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映数值计算的要求，因此，字符串的处理比具体数值处理复杂。本章讨论串的存储结构及几种基本的处理。

4.1 串类型的定义

4.1.1 串的基本概念

串(字符串)：是零个或多个字符组成的有限序列。记作： S="$a_1a_2a_3$..."，其中S是串名，$a_i(1 \leqq i \leqq n)$是单个，可以是字母、数字或其它字符。

串值：双引号括起来的字符序列是串值。

串长：串中所包含的字符个数称为该串的长度。

空串(空的字符串)：长度为零的串称为空串，它不包含任何字符。

空格串(空白串)：构成串的所有字符都是空格的串称为空白串。

注意：空串和空白串的不同，例如“ ”和“”分别表示长度为1的空白串和长度为0的空串。

子串(substring)：串中任意个连续字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。

子串的序号：将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）。

例如，设有串A和B分别是：

A=“这是字符串”，B=“是”

则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的序号为3 。

特别地，空串是任意串的子串，任意串是其自身的子串。

串相等：如果两个串的串值相等(相同)，称这两个串相等。换言之，只有当两个串的长度相等，且各个对应位置的字符都相同时才相等。

通常在程序中使用的串可分为两种：串变量和串常量。

串常量和整常数、实常数一样，在程序中只能被引用但不能不能改变其值，即只能读不能写。通常串常量是由直接量来表示的，例如语句错误(“溢出”)中“溢出”是直接量。

串变量和其它类型的变量一样，其值是可以改变。

4.1.2 串的抽象数据类型定义

ADT String{

数据对象：**D = { a_i|a_i∈CharacterSet, i=1,2,…,n, n≥0 }**

数据关系：**R = {<a_{i-1}, a_i>| a_{i-1}, a_i∈D, i=2,3,…,n }**

基本操作：

StrAssign(t , chars)

初始条件： **chars**是一个字符串常量。

操作结果：生成一个值为**chars**的串**t** 。

StrConcat(s, t)

初始条件：串**s, t** 已存在。

操作结果：将串t联结到串s后形成新串存放到s中。

StrLength(t)

初始条件：字符串t已存在。

操作结果：返回串t中的元素个数，称为串长。

SubString (s, pos, len, sub)

初始条件：串s, 已存在, 1≦pos≦StrLength(s)且0≦len≦StrLength(s) –pos+1。

操作结果：用sub返回串s的第pos个字符起长度为len的子串。

……

} ADT String

4.2 串的存储表示和实现

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- 定长顺序存储表示：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。
- 堆分配存储方式：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- 块链存储方式：是一种链式存储结构表示。

4.2.1 串的定长顺序存储表示

这种存储结构又称为串的顺序存储结构。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

```
#define MAX_STRLEN  256
typedef  struct
   {  char  str[MAX_STRLEN] ;
      int  length;
   } StringType ;
```

1 串的联结操作

```
Status StrConcat ( StringType s, StringType t)
  /*  将串t联结到串s之后，结果仍然保存在s中  */
  {  int i, j ;
     if ((s.length+t.length)>MAX_STRLEN)
        Return ERROR ;  /*  联结后长度超出范围  */
      for (i=0 ; i<t.length ; i++)
        s.str[s.length+i]=t.str[i] ;  /*  串t联结到串s之后  */
     s.length=s.length+t.length ;  /* 修改联结后的串长度 */
     return OK ;
  }
```

2 求子串操作

```
Status SubString (StringType s, int pos, int len, StringType *sub)
  {  int k,  j ;
     if (pos<1||pos>s.length||len<0||len>(s.length-pos+1))
        return ERROR ;   /*  参数非法  */
     sub->length=len-pos+1 ;   /*  求得子串长度  */
     for (j=0, k=pos ; k<=leng ; k++, j++)
        sub->str[j]=s.str[i] ;   /*  逐个字符复制求得子串  */
     return OK ;
  }
```

4.2.2 串的堆分配存储表示

实现方法：系统提供一个空间足够大且地址连续的存储空间(称为“堆”)供串使用。可使用C语言的动态存储分配函数**malloc()**和**free()**来管理。

特点是：仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef  struct
   {  char *ch;    /*  若非空，按长度分配，否则为NULL */
      int length;      /*  串的长度  */
   } HString ;
```

1 串的联结操作

```
Status Hstring *StrConcat(HString *T, HString *s1, HString *s2)
  /* 用T返回由s1和s2联结而成的串 */
  {  int k, j , t_len ;
     if (T.ch) free(T);    /* 释放旧空间 */
     t_len=s1->length+s2->length ;
     if ((p=(char *)malloc(sizeof((char)*t_len))==NULL)
        {  printf("系统空间不够，申请空间失败！\n") ;
           return ERROR ;   }
     for (j=0 ; j<s->length; j++)
        T->ch[j]=s1->ch[j] ;   /* 将串s复制到串T中 */
```

```
    for (k=s1->length, j=0 ; j<s2->length;
    k++, j++)
      T->ch[j]=s1->ch[j];    /* 将串s2复制到
      串T中 */
    free(s1->ch);
    free(s2->ch);
    return OK ;
}
```

4.2.3 串的链式存储表示

串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：

- **data域**：存放字符，**data域**可存放的字符个数称为结点的大小；
- **next域**：存放指向下一结点的指针。

若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。如图**4-1**是块大小为**3**的串的块链式存储结构示意图。

图4-1　串的块链式存储结构示意图

串的块链式存储的类型定义包括：

(1) 块结点的类型定义

```
#define BLOCK_SIZE  4
typedef  struct  Blstrtype
   {  char  data[BLOCK_SIZE] ;
      struct  Blstrtype  *next;
   }BNODE ;
```

(2) 块链串的类型定义

```
typedef  struct
 {  BNODE  head;     /*  头指针  */
    int  Strlen ;       /*  当前长度  */
  } Blstring ;
```

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结。

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

4.3 串的模式匹配算法

模式匹配(模范匹配)：子串在主串中的定位称为模式匹配或串匹配(字符串匹配）。模式匹配成功是指在主串**S**中能够找到模式串**T**，否则，称模式串**T**在主串**S**中不存在。

模式匹配的应用在非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。介绍两种主要的模式匹配算法。

4.3.1 Brute-Force模式匹配算法

设S为目标串，T为模式串，且不妨设：

S=“$s_0s_1s_2\ldots s_{n-1}$”， T=“$t_0t_1t_2\ldots t_{m-1}$”

串的匹配实际上是对合法的位置$0 \leqslant i \leqslant n-m$依次将目标串中的子串$s[i\ldots i+m-1]$和模式串$t[0\ldots m-1]$进行比较：

- 若$s[i\ldots i+m-1]=t[0\ldots m-1]$：则称从位置i开始的匹配成功，亦称模式t在目标s中出现；
- 若$s[i\ldots i+m-1]\neq t[0\ldots m-1]$：从i开始的匹配失败。位置i称为位移，当$s[i\ldots i+m-1]=t[0\ldots m-1]$时，i称为有效位移；当$s[i\ldots i+m-1]\neq t[0\ldots m-1]$时，i称为无效位移。

这样，串匹配问题可简化为找出某给定模式T在给定目标串S中首次出现的有效位移。

算法实现

```
int IndexString(StringType s，StringType t，int pos）
  /*  采用顺序存储方式存储主串s和模式t，    */
  /*  若模式t在主串s中从第pos位置开始有匹配的子串，*/
  /*  返回位置，否则返回-1   */
  {  char *p，*q；
     int  k，j；
     k=pos-1；  j=0；p=s.str+pos-1；q=t.str；
        /*  初始匹配位置设置  */
         /*  顺序存放时第pos位置的下标值为pos-1 */
```

```
while (k<s.length)&&(j<t.length)
  {  if (*p==*q)  { p++ ; q++ ; k++ ;
  j++ ; }
    else { k=k-j+1 ; j=0 ; q=t.str ;
    p=s.str+k ;  }
    /*  重新设置匹配位置    */
    }
if (j==t.length)
    return(k-t.length) ;   /*   匹配，返回位置
*/
else return(-1) ;     /*   不匹配，返回-1   */
}
```

该算法简单，易于理解。在一些场合的应用里，如文字处理中的文本编辑，其效率较高。

该算法的时间复杂度为O(n*m)，其中n、m分别是主串和模式串的长度。通常情况下，实际运行过程中，该算法的执行时间近似于O(n+m)。

理解该算法的关键点

当第一次$s_k \neq t_j$时：主串要退回到k-j+1的位置，而模式串也要退回到第一个字符（即j=0的位置）。

比较出现$s_k \neq t_j$时：则应该有$s_{k-1}=t_{j-1}$，…，$s_{k-j+1}=t_1$，$s_{k-j}=t_0$。

4.3.2 模式匹配的一种改进算法

该改进算法是由D.E.Knuth，J.H.Morris和V.R.Pratt提出来的，简称为KMP算法。其改进在于：

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”尽可能远的一段距离后，继续进行比较。

例：设有串s=“abacabab”，t=“abab”。则第一次匹配过程如图4-2所示。

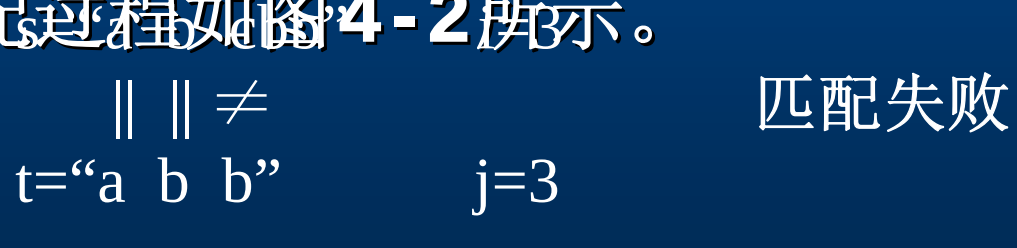


图4-2 模式匹配示例

在i=3和j=3时，匹配失败。但重新开始第二次匹配时，不必从i=1，j=0开始。因为$s_1=t_1$，$t_0\neq t_1$，必有$s_1\neq t_0$，又因为$t_0=t_2$，$s_2=t_2$，所以必有$s_2=t_0$。由此可知，第二次匹配可以直接从i=3、j=1开始。

总之，在主串s与模式串t的匹配过程中，一旦出现$s_i\neq t_j$，主串s的指针不必回溯，而是直接与模式串的t_k(0≦k<j进行比较，而k的取值与主串s无关，只与模式串t本身的构成有关，即从模式串t可求得k值。)

不失一般性，设主串s=“$s_1 s_2 \ldots s_n$”，模式串t=“$t_1\ t_2 \ldots t_m$”。

当$s_i \neq t_j$($1 \leqq i \leqq n-m$，$1 \leqq j < m$，$m < n$)时，主串s的指针i不必回溯，而模式串t的指针j回溯到第k(k<j)个字符继续比较，则模式串t的前k-1个字符必须满足4-1式，而且不可能存在k’>k满足4-1式。

$$t_1 t_2 \ldots t_{k-1} = s_{i-(k-1)}\ s_{i-(k-2)} \cdots s_{i-2}\ s_{i-1} \quad (4\text{-}1)$$

而已经得到的“部分匹配”的结果为：

$$t_{j-(k-1)}\ t_{j-k} \cdots t_{j-1} = s_{i-(k-1)}\ s_{i-(k-2)} \cdots s_{i-2}\ s_{i-1} \quad (4\text{-}2)$$

由式(4-1)和式(4-2)得：

$$t_1 t_2 \ldots t_{k-1} = t_{j-(k-1)}\ t_{j-k} \cdots t_{j-1} \quad (4\text{-}3)$$

该推导过程可用图4-3形象描述。实际上，式(4-3)描述了模式串中存在相互重叠的子串的情况。

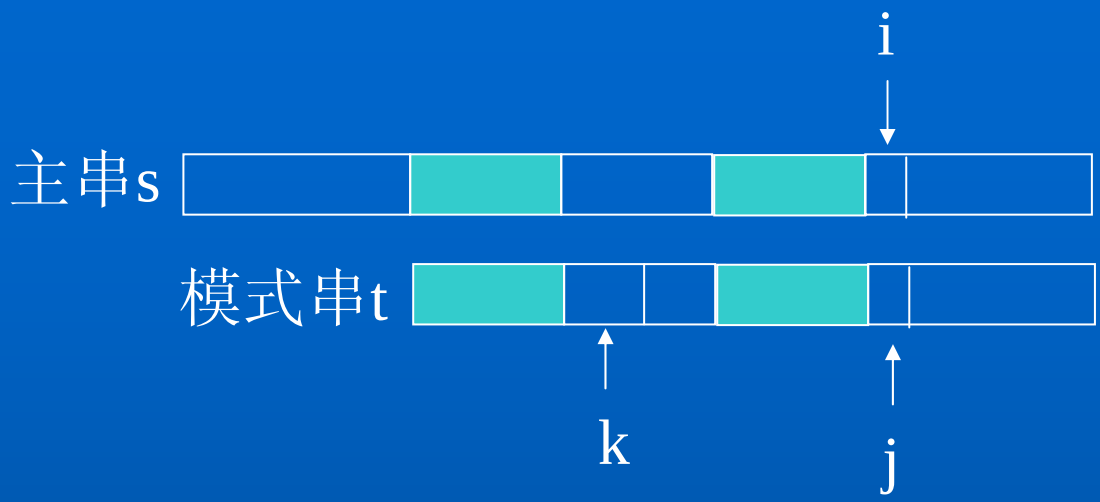


图4-3　KMP算法示例

定义next[j]函数为

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时}\\ Max\{k \mid 1<k<j \wedge t_1t_2\ldots t_{k-1}=t_{j-(k-1)}\ t_{j-k}\ldots\ t_{j-1}\} & \text{该集合不空时}\\ 1 & \text{其它情况}\end{cases}$$

在求得了next[j]值之后，KMP算法的思想是：

设目标串(主串)为s，模式串为t ，并设i指针和j指针分别指示目标串和模式串中正待比较的字符，设i和j的初值均为1。若有$s_i = t_j$，则i和j分别加1。否则，i不变，j退回到j=next[j]的位置，再比较s_i和t_j，若相等，则i和j分别加1。否则，i不变，j再次退回到j=next[j]的位置，依此类推。直到下列两种可能：

(1) j退回到某个下一个[j]值时字符比较相等，则指针各自加1继续进行匹配。

•退回到j=0，将i和j分别加1，即从主串的下一个字符s_{i+1}模式串的t_1重新开始匹配。

KMP算法如下

```
#define Max_Strlen 1024
int next[Max_Strlen];
int KMP_index (StringType  s , StringType t)
 /* 用KMP算法进行模式匹配，匹配返回位置，否则返回-1 */
 /*用静态存储方式保存字符串， s和t分别表示主串和模式串  */
  {  int  k=0 , j=0 ;     /*初始匹配位置设置 */
     while (k<s.length)&&(j<t.length
        {  if ((j==-1)|| (s. str[k]==t.str[j])) {  k++ ; j++ ;
        }
           else j=next[j] ;
        }
     if (j>= t.length)  return(k-t.length) ;
     else  return(-1) ;
  }
```

很显然，KMP_index函数是在已知下一个函数值的基础上执行的，以下讨论如何求next函数值?

由式(4-3)知，求模式串的next[j]值与主串s无关，只与模式串t本身的构成有关，则可把求next函数值的问题看成是一个模式匹配问题。由next函数定义可知：

当j=1时：next[1]=0。

设next[j]=k，即在模式串中存在：$t_1t_2\cdots t_{k-1}=t_{j-(k-1)}t_{j-k}\cdots t_{j-1}$，其中下标k满足1<k<j的某个最大值，此时求next[j+1]的值有两种可能：

(1) 若有$t_k=t_j$：则表明在模式串中有：

$t_1t_2\cdots t_{k-1}t_k=t_{j-(k-1)}t_{j-k}\cdots t_{j-1}t_j$，且不可能存在k'>k满足上式，即：next[j+1]=next[j]+1=k+1

(2) 若有$t_k \neq t_j$：则表明在模式串中有：$t_1\ t_2 \ldots t_{k-1}\ t_k \neq t_{j-(k-1)}\ t_{j-k} \cdots\ t_{j-1}\ t_j$，当$t_k \neq t_j$时应将模式向右滑动至以模式中的第next[k]个字符和主串中的第j个字符相比较。若next[k]= k’，且$t_j = t_{k'}$，则说明在主串中第j+1字符之前存在一个长度为k’(即next[k])的最长子串，与模式串中从第一个字符起长度为k’的子串相等。即

next[j+1]=k’+1

同理，若$t_j \neq t_k$，应将模式继续向右滑动至将模式中的第next[k’]个字符和t_j对齐，……，依此类推，直到t_j和模式串中的某个字符匹配成功或者不存在任何k’(1< k’<j)满足等式：$t_1\ t_2 \ldots t_{k-1}\ t_{k'} = t_{j-(k'-1)}\ t_{j-k'} \cdots\ t_{j-1}\ t_j$

则： next[j]+1=1

根据上述分析， 求next函数值的算法如下：

```
void  next(StringType  t , int next[])
  /*  求模式t的next串t函数值并保存在next数组中    */
  {   int  k=1 , j=0 ; next[1]=0;
     while (k<t.length)
       {   if ((j==0)|| (t.str[k]==t.str[j]))
             {   k++ ; j++ ;
                 if ( t.str[k]!=t.str[j] )  next[k]=j;
                  else  next[k]=next[j];
             }
          else next[j]=j ;
       }
  }
```

习 题 四

⑴ 解释下列每对术语的区别：空串和空白串；主串和子串；目标串和模式串。

⑵ 若**x**和**y**是两个采用顺序结构存储的串，写一算法比较这两个字符串是否相等。

⑶ 写一算法**void StrRelace(char *T, char *P, char *S)**，将**T**中第一次出现的与**P**相等的子串替换为**S**，串**S**和**P**的长度不一定相等，并分析时间复杂度。

第5章 数组和广义表

数组是一种人们非常熟悉的数据结构，几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型。数组这种数据结构可以看成是线性表的推广。

科学计算中涉及到大量的矩阵问题，在程序设计语言中一般都采用数组来存储，被描述成一个二维数组。但当矩阵规模很大且具有特殊结构(对角矩阵、三角矩阵、对称矩阵、稀疏矩阵等)，为减少程序的时间和空间需求，采用自定义的描述方式。

广义表是另一种推广形式的线性表，是一种灵活的数据结构，在许多方面有广泛的应用。

5.1 数组的定义

数组是一组偶对(下标值，数据元素值)的集合。在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推。

数组是由n(n>1)个具有相同数据类型的数据元素a_1，a_2，…，a_n组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。

- 数组中的数据元素具有相同数据类型。
- 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
- 数组中的数据元素个数是固定的。

5.1.1 数组的抽象数据类型定义

1 抽象数据类型定义

ADT Array{

数据对象：$j_i= 0,1,\ldots,b_i-1 , 1,2, \ldots,n$;

D = { $a_{j_1j_2\ldots j_n}$ | n>0称为数组的维数，b_i是数组第**i**维的长度，j_i是数组元素第**i**维的下标，$a_{j_1j_2\ldots j_n}$**∈ElemSet }**

数据关系：**R = {R_1, R_2, …, R_n}**

R_i={<$a_{j_1j_2\ldots j_i\ldots j_n}$, $a_{j_1j_2\ldots j_{i+1}\ldots j_n}$>|0≦$j_k$≦$b_k$-1 ，　　1≦k≦n且k≠i，0≦$j_i$≦$b_i$-2，　$a_{j_1j_2\ldots j_{i+1}\ldots j_n}$∈D }

基本操作：　……

} ADT Array

由上述定义知，n维数组中有$b_1 \times b_2 \times \ldots \times b_n$个数据元素，每个数据元素都受到n维关系的约束。

2　直观的n维数组

以二维数组为例讨论。将二维数组看成是一个定长的线性表，其每个元素又是一个定长的线性表。

设二维数组$A=(a_{ij})_{m\times n}$，则

$A=(\alpha_1, \alpha_2, \ldots, \alpha_p)$　（p＝m或n）

其中每个数据元素α_j是一个列向量(线性表)：

$\alpha_j=(a_{1j}, a_{2j}, \ldots, a_{mj})$　$1 \leqq j \leqq n$

或是一个行向量：

$\alpha_i=(a_{i1}, a_{i2}, \ldots, a_{in})$　$1 \leqq i \leqq m$

如图5-1所示。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(a) 矩阵表示形式

$$A=\begin{pmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{bmatrix} \\ \begin{bmatrix} a_{21} & a_{22} & \cdots & a_{2n} \end{bmatrix} \\ \cdots \cdots \cdots \cdots \cdots \\ \begin{bmatrix} a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \end{pmatrix}$$

(b) 列向量的一维数组形式

$$A=\begin{pmatrix} \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} & \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} & \vdots & \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} \end{pmatrix}$$

(c) 行向量的一维数组形式

图**5-1** 二维数组图例形式

5.2 数组的顺序表示和实现

数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。

问题：计算机的内存结构是一维(线性)地址结构，对于多维数组，将其存放(映射)到内存一维结构时，有个次序约定问题。即必须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放到内存中。

二维数组是最简单的多维数组，以此为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。

通常有两种顺序存储方式

(1) 行优先顺序(Row Major Order)：将数组元素按行排列，第i+1个行向量紧接在第i个行向量后面。对二维数组，按行优先顺序存储的线性序列为：

$a_{11},a_{12},\ldots,a_{1n}$, $a_{21},a_{22},\ldots a_{2n}$,……, $a_{m1},a_{m2},\ldots,a_{mn}$

PASCAL、C是按行优先顺序存储的，如图5-2(b)示。

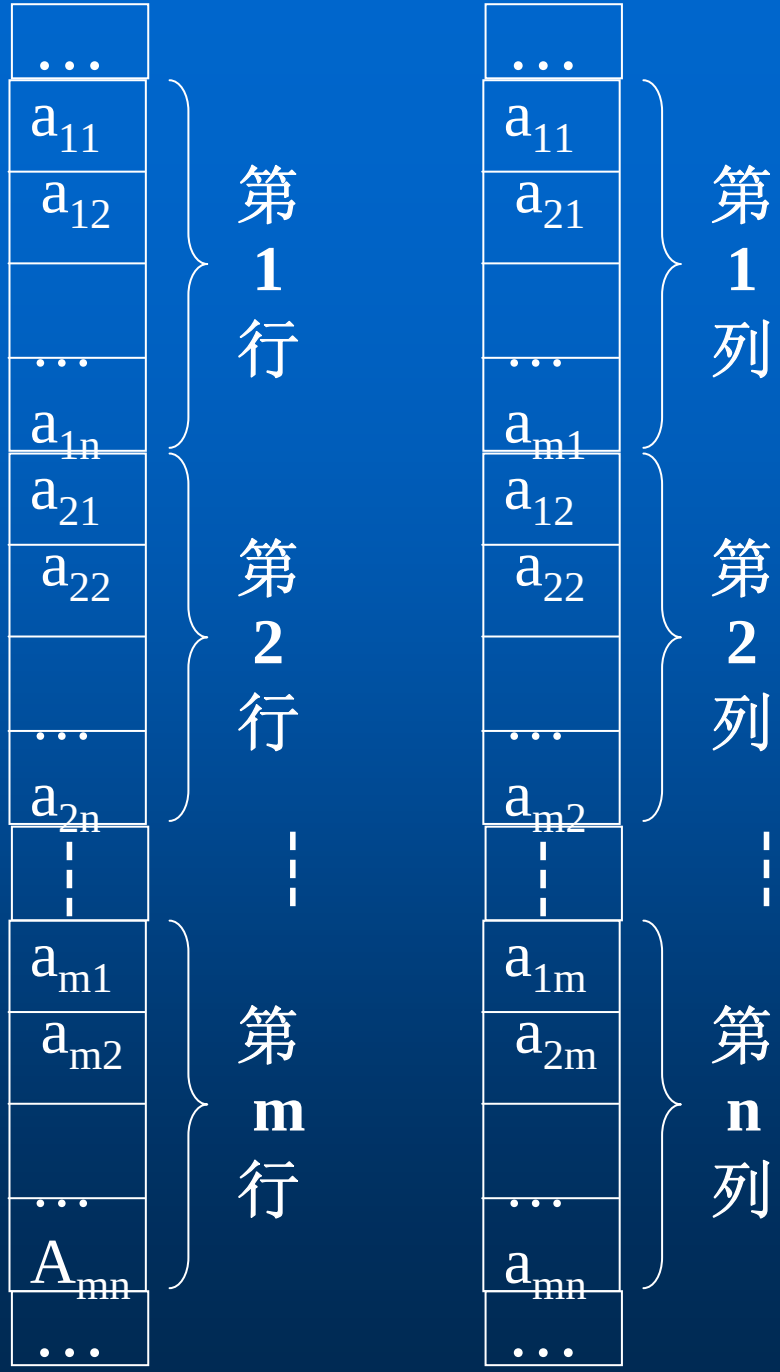
(2) 列优先顺序(Column Major Order)：将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后，对二维数组，按列优先顺序存储的线性序列为：

$a_{11},a_{21},\ldots,a_{m1}$, $a_{12},a_{22},\ldots a_{m2}$, ……, $a_{n1},a_{n2},\ldots,a_{nm}$

FORTRAN是按列优先顺序存储的，如图5-2(c)。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(a) 二维数组的表示形式



(b) 行优先顺序存储　　**(c)** 列优先顺序存储

图5-2　二维数组及其顺序存储图例形式

设有二维数组$A=(a_{ij})_{m\times n}$，若每个元素占用的存储单元数为l(个)，$LOC[a_{11}]$表示元素a_{11}的首地址，即数组的首地址。

1　以“行优先顺序”存储

⑴ 第1行中的每个元素对应的(首)地址是：

$$LOC[a_{1j}]=LOC[a_{11}]+(j-1)\times l \qquad j=1,2,\ldots,n$$

(2) 第2行中的每个元素对应的(首)地址是：

$$LOC[a_{2j}]=LOC[a_{11}]+n\times l+(j-1)\times l \qquad j=1,2,\ldots,n$$

… … …

⑶ 第m行中的每个元素对应的(首)地址是：

$$LOC[a_{mj}]=LOC[a_{11}]+(m-1)\times n\times l+(j-1)\times l \qquad j=1,2,\ldots,n$$

由此可知，二维数组中任一元素a_{ij}的(首)地址是：

$$LOC[a_{ij}]=LOC[a_{11}]+[(i-1)\times n+(j-1)]\times l \quad (5-1)$$

$$i=1,2,\ldots,m \quad j=1,2,\ldots,n$$

根据(5-1)式，对于三维数组$A=(a_{ijk})_{m\times n\times p}$，若每个元素占用的存储单元数为$l$(个)，$LOC[a_{111}]$表示元素$a_{111}$的首地址，即数组的首地址。以“行优先顺序”存储在内存中。

三维数组中任一元素a_{ijk}的(首)地址是：

$$LOC(a_{ijk})=LOC[a_{111}]+[(i-1)\times n\times p+(j-1)\times p+(k-1)]\times l \quad (5-2)$$

推而广之，对n维数组$A=(a_{j_1j_2\ldots j_n})$，若每个元素占用的存储单元数为l(个)，$LOC[a_{11\ldots1}]$表示元素$a_{11\ldots1}$的首地址。则 以“行优先顺序”存储在内存中。

n维数组中任一元素$a_{j_1 j_2 \ldots j_n}$的(首)地址是:

$$LOC[a_{j_1 j_2 \ldots j_n}] = LOC[a_{11 \ldots 1}] + [(b_2 \times \ldots \times b_n) \times (j_1 - 1) + (b_3 \times \ldots \times b_n) \times (j_2 - 1) + \ldots + b_n \times (j_{n-1} - 1) + (j_n - 1)] \times l \quad (5\text{-}3)$$

2 以“列优先顺序”存储

⑴ 第1列中的每个元素对应的(首)地址是：

$$LOC[a_{j1}]=LOC[a_{11}]+(j-1)\times l \qquad j=1,2,\ldots,m$$

(2) 第2列中的每个元素对应的(首)地址是：

$$LOC[a_{j2}]=LOC[a_{11}]+m\times l+(j-1)\times l \qquad j=1,2,\ldots,m$$

… … …

⑶ 第n列中的每个元素对应的(首)地址是：

$$LOC[a_{jn}]=LOC[a_{11}]+(n-1)\times m\times l+(j-1)\times l \qquad j=1,2,\ldots,m$$

由此可知，二维数组中任一元素a_{ij}的(首)地址是：

$$LOC[a_{ij}]=LOC[a_{11}]+[(i-1)\times m+(j-1)]\times l \qquad (5\text{-}1)$$

$i=1,2,\ldots,n \quad j=1,2,\ldots,m$

5.3 矩阵的压缩存储

在科学与工程计算问题中，矩阵是一种常用的数学对象，在高级语言编程时，通常将一个矩阵描述为一个二维数组。这样，可以对其元素进行随机存取，各种矩阵运算也非常简单。

对于高阶矩阵，若其中非零元素呈某种规律分布或者矩阵中有大量的零元素，若仍然用常规方法存储，可能存储重复的非零元素或零元素，将造成存储空间的大量浪费。对这类矩阵进行压缩存储：

- 多个相同的非零元素只分配一个存储空间；
- 零元素不分配空间。

5.3.1 特殊矩阵

特殊矩阵：是指非零元素或零元素的分布有一定规律的矩阵。

1 对称矩阵

若一个n阶方阵$A=(a_{ij})_{n\times n}$中的元素满足性质：

$$a_{ij}=a_{ji} \quad 1\leqq i,j\leqq n \text{且} i\neq j$$

则称A为对称矩阵，如图5-3所示。

$$A=\begin{pmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{pmatrix} \qquad A=\begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \cdots & \cdots & \cdots & \cdots & \\ a_{n1} & a_{n2} & \cdots & & a_{nn} \end{pmatrix}$$

图5-3 对称矩阵示例

对称矩阵中的元素关于主对角线对称，因此，让每一对对称元素a_{ij}和a_{ji}($i \neq j$)分配一个存储空间，则n^2个元素压缩存储到$n(n+1)/2$个存储空间，能节约近一半的存储空间。

不失一般性，假设按“行优先顺序”存储下三角形(包括对角线)中的元素。

设用一维数组(向量)sa[0...n(n+1)/2]存储n阶对称矩阵，如图5-4所示。为了便于访问，必须找出矩阵A中的元素的下标值（i,j）和向量sa[k]的下标值k之间的对应关系。

K	1	2	3	4	…	n(n-1)/2	…	n(n+1)/2
sa	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{n1}	a_{n2}	…	a_{nn}

图5-4　对称矩阵的压缩存储示例

若i≧j：a_{ij}在下三角形中，直接保存在sa中。a_{ij}之前的i-1行共有元素个数： 1+2+…+(i-1)=i×(i-1)/2

而在第i行上，a_{ij}之前恰有j-1个元素，因此，元素a_{ij}保存在向量sa中时的下标值k之间的对应关系是：

$$k=i\times(i-1)/2+j-1 \qquad i\geqq j$$

若i<j：则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，在向量sa中保存的是a_{ji}。依上述分析可得：

$$k=j\times(j-1)/2+i-1 \qquad i<j$$

对称矩阵元素a_{ij}保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当}i\geqq j\text{时} \\ j\times(j-1)/2+i-1 & \text{当}i<j\text{时} \end{cases} \qquad 1\leqq i,j\leqq n \qquad (5\text{-}4)$$

根据上述的下标对应关系，对于矩阵中的任意元素a_{ij}，均可在一维数组sa中唯一确定其位置k；反之，对所有k=1,2, …,n(n+1)/2，都能确定sa[k]中的元素在矩阵中的位置(i,j)。

称sa[0…n(n+1)/2]为n阶对称矩阵A的压缩存储。

2 三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。

上三角矩阵的下三角（不包括主对角线）中的元素均为常数c(一般为0)。下三角矩阵正好相反，它的主对角线上方均为常数，如图5-5所示。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ c & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \\ c & c & \cdots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & c & \cdots & c \\ a_{21} & a_{22} & \cdots & c \\ \cdots & \cdots & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

(a) 上三角矩阵示例　　**(b)** 下三角矩阵示例

图5-5　三角矩阵示例

三角矩阵中的重复元素c可共享一个存储空间，其余的元素正好有n(n+1)/2个，因此，三角矩阵可压缩存储到向量sa[0…n(n+1)/2]中，其中c存放在向量的第1个分量中。

上三角矩阵元素$a_{i\,j}$保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当}i\geqslant j\text{时} \\ n\times(n+1)/2 & \text{当}i<j\text{时} \end{cases} \qquad 1\leqslant i,j\leqslant n \qquad (5\text{-}5)$$

下三角矩阵元素$a_{i\ j}$保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当}i\leqslant j\text{时} \\ n\times(n+1)/2 & \text{当}i>j\text{时} \end{cases} \qquad 1\leqslant i,j\leqslant n \qquad (5\text{-}6)$$

3　对角矩阵

矩阵中，除了主对角线和主对角线上或下方若干条对角线上的元素之外，其余元素皆为零。即所有的非零元素集中在以主对角线为了中心的带状区域中，如图5-6所示。

$$A=\begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \cdots & 0 \\ & \cdots & \cdots & \cdots & \cdots & \\ 0 & \cdots & 0 & a_{n-1\,n-2} & a_{n-1\,n-1} & a_{n-1\,n} \\ 0 & \cdots & 0 & 0 & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

图5-6　三对角矩阵示例

如上图三对角矩阵，非零元素仅出现在主对角($a_{i\,i}$,$1 \leqq i \leqq n$)上、主对角线上的那条对角线($a_{i\,i+1}$,$1 \leqq i \leqq n-1$)、主对角线下的那条对角线上($a_{i+1\,i}$,$1 \leqq i \leqq n-1$)。显然，当$|\ i-j\ | > 1$时，元素$a_{ij}=0$。

由此可知，一个k对角矩阵(k为奇数)A是满足下述条件：　当$|\ i-j\ | > (k-1)/2$时，　$a_{i\,j}=0$

对角矩阵可按行优先顺序或对角线顺序，将其压缩存储到一个向量中，并且也能找到每个非零元素和向量下标的对应关系。

仍然以三对角矩阵为例讨论。

当i=1，j=1、2，或i=n， j=n-1、n或1<i<n-1,j=i-1、i、i+1的元素a_{ij}外，其余元素都是0。

对这种矩阵，当以按“行优先顺序”存储时， 第1行和第n行是2个非零元素，其余每行的非零元素都要是3个，则需存储的元素个数为3n-2。

K	1	2	3	4	5	6	7	8	…	3n-3	3n-2
sa	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	…	$a_{n\,n-1}$	a_{nn}

图5-7 三对角矩阵的压缩存储示例

如图5-7所示三对角矩阵的压缩存储形式。数组sa中的元素sa[k]与三对角矩阵中的元素a_{ij}存在一一对应关系，在a_{ij}之前有i-1行，共有3×i-1个非零元素，在第i行，有j-i+1个非零元素，这样，非零元素a_{ij}的地址为：

$$LOC[a_{ij}] = LOC[a_{11}] + [3\times i-1+(j-i+1)]\times l$$

$$= LOC[a_{11}] + (2\times i+j)\times l$$

上例中，a_{34}对应着sa[10]，　k=2×i+j=2×3+4=10

称sa[0…3×n-2]是n阶三对角矩阵A的压缩存储。

上述各种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并且一般都能找到矩阵中的元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

5.3.2 稀疏矩阵

稀疏矩阵(Sparse Matrix)：对于稀疏矩阵，目前还没有一个确切的定义。设矩阵A是一个n×m的矩阵中有s个非零元素，设 $\delta = s/(n \times m)$，称 δ 为稀疏因子，如果某一矩阵的稀疏因子 δ 满足 $\delta \leqslant 0.05$ 时称为稀疏矩阵，如图5-8所示。

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

图5-8 稀疏矩阵示例

5.3.2.1 稀疏矩阵的压缩存储

对于稀疏矩阵，采用压缩存储方法时，只存储非0元素。必须存储非0元素的行下标值、列下标值、元素值。因此，一个三元组(i, j, a_{ij})唯一确定稀疏矩阵的一个非零元素。

如图5-8的稀疏矩阵A的三元组线性表为：

((1,2,12), (1,3,9), (3,1,-3), (3,8,4), (4,3,24), (5,2,18), (6,7,-7), (7,4,-6))

1 三元组顺序表

若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。

1　三元组顺序表

若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。相应的数据结构定义如下：

(1) 三元组结点定义

```
#define MAX_SIZE 101
typedef int elemtype ;
typedef struct
   {   int   row ;     /*  行下标  */
       int  col ;       /*  列下标  */
       elemtype value;      /*  元素值  */
   }Triple ;
```

(2) 三元组顺序表定义

```
typedef struct
  {  int  rn ;          /*  行数   */
     int  cn ;          /*  列数   */
     int  tn ;          /*  非0元素个数   */
     Triple  data[MAX_SIZE] ;
  }TMatrix ;
```

图5-8所示的稀疏矩阵及其相应的转置矩阵所对应的三元组顺序表如图5-9所示。

7	**rn**行数
8	**cn**列数
9	**tn**元素个数

row	col	value
1	2	12
1	3	9
3	1	-3
3	8	4
4	3	24
4	6	2
5	2	18
6	7	-7
7	4	-6

(a) 原矩阵的三元组表

8	**rn**行数
7	**cn**列数
9	**tn**元素个数

row	col	value
1	3	-3
2	1	12
2	5	18
3	1	9
3	4	24
4	7	-6
6	4	2
7	6	-7
8	2	4

(b)转置矩阵的三元组表

图**5-9** 稀疏矩阵及其转置矩阵的三元组顺序表

矩阵的运算包括矩阵的转置、矩阵求逆、矩阵的加减、矩阵的乘除等。在此，先讨论在这种压缩存储结构下的求矩阵的转置的运算。

一个**m×n**的矩阵**A**，它的转置**B**是一个**n×m**的矩阵，且**b[i][j]=a[j][i]**，**0≦i≦n**，**0≦j≦m**，即**B**的行是**A**的列，**B**的列是**A**的行。

设稀疏矩阵**A**是按行优先顺序压缩存储在三元组表**a.data**中，若仅仅是简单地交换**a.data**中**i**和**j**的内容，得到三元组表**b.data**，**b.data**将是一个按列优先顺序存储的稀疏矩阵**B**，要得到按行优先顺序存储的**b.data**，就必须重新排列三元组表**b.data**中元素的顺序。

求转置矩阵的基本算法思想是：

① 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值i 、j相互交换；

② 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。

方法一：

算法思想：按稀疏矩阵A的三元组表a.data中的列次序依次找到相应的三元组存入b.data中。

每找转置后矩阵的一个三元组，需从头至尾扫描整个三元组表a.data 。找到之后自然就成为按行优先的转置矩阵的压缩存储表示。

按方法一求转置矩阵的算法如下：

```
void TransMatrix(TMatrix a , TMatrix b)
   {   int p , q , col ;
       b.rn=a.cn ;  b.cn=a.rn ;  b.tn=a.tn ;
          /*    置三元组表b.data的行、列数和非0元素个数  */
       if  (b.tn==0)    printf(“ The Matrix A=0\n” );
       else
          {   q=0;
              for  (col=1; col<=a.cn ; col++)
                    /*   每循环一次找到转置后的一个三元组  */
              for  (p=0 ;p<a.tn ; p++)
                    /*   循环次数是非0元素个数   */
```

```
            if (a.data[p].col==col)
                { b.data[q].row=a.data[p].col ;
                  b.data[q].col=a.data[p].row ;
                   b.data[q].value=a.data[p].value;
                   q++ ;
                }
        }
    }
```

算法分析：本算法主要的工作是在p和col的两个循环中完成的，故算法的时间复杂度为O(cn×tn)，即矩阵的列数和非0元素的个数的乘积成正比。

而一般传统矩阵的转置算法为：

```
for(col=1; col<=n ;++col)
    for(row=0 ; row<=m ;++row)
        b[col][row]=a[row][col] ;
```

其时间复杂度为O(n×m)。当非零元素的个数tn和m×n同数量级时，算法TransMatrix的时间复杂度为$O(m\times n^2)$。

由此可见，虽然节省了存储空间，但时间复杂度却大大增加。所以上述算法只适合于稀疏矩阵中非0元素的个数tn远远小于m×n的情况。

方法二(快速转置的算法)

算法思想：直接按照稀疏矩阵A的三元组表a.data的次序依次顺序转换，并将转换后的三元组放置于三元组表b.data的恰当位置。

前提：若能预先确定原矩阵A中每一列的(即B中每一行)第一个非0元素在b.data中应有的位置，则在作转置时就可直接放在b.data中恰当的位置。因此，应先求得A中每一列的非0元素个数。

附设两个辅助向量num[ ]和cpot[ ]。

- num[col]：统计A中第col列中非0元素的个数；
- cpot[col]：指示A中第一个非0元素在b.data中的恰当位置。

显然有位置对应关系：

$$\begin{cases} cpot[1]=1 \\ cpot[col]=cpot[col-1]+num[col-1] \qquad 2 \leqq col \leqq a.cn \end{cases}$$

例图5-8中的矩阵A和表5-9(a)的相应的三元组表可以求得num[col]和cpot[col]的值如表5-1：

表5-1　num[col]和cpot[col]的值表

col	1	2	3	4	5	6	7	8
num[col]	1	2	2	1	0	1	1	1
cpot[col]	1	3	5	6	6	7	8	9

快速转置算法如下：

```
void FastTransMatrix(TMatrix a, TMatrix b)
{   int p , q , col , k ;
    int num[MAX_SIZE] , copt[MAX_SIZE] ;
    b.rn=a.cn ; b.cn=a.rn ; b.tn=a.tn ;
        /*  置三元组表b.data的行、列数和非0元素个数  */
    if (b.tn==0)   printf(" The Matrix A=0\n" ) ;
    else
      {  for (col=1 ; col<=a.cn ; ++col)   num[col]=0 ;
           /*  向量num[]初始化为0  */
         for (k=1 ; k<=a.tn ; ++k)
           ++num[ a.data[k].col] ;
              /*  求原矩阵中每一列非0元素个数  */
```

```
        for (cpot[0]=1, col=2 ; col<=a.cn ; ++col)
             cpot[col]=cpot[col-1]+num[col-1] ;
               /*  求第col列中第一个非0元在b.data中的序号  */
        for (p=1 ; p<=a.tn ; ++p)
           {   col=a.data[p].col ;  q=cpot[col] ;
               b.data[q].row=a.data[p].col ;
               b.data[q].col=a.data[p].row ;
               b.data[q].value=a.data[p].value ;
               ++cpot[col] ;     /*至关重要!!当本列中   */
           }
      }
}
```

2　行逻辑链接的三元组顺序表

将上述方法二中的辅助向量**cpot[ ]**固定在稀疏矩阵的三元组表中，用来指示“行”的信息。得到另一种顺序存储结构：行逻辑链接的三元组顺序表。其类型描述如下：

```
#define MAX_ROW 100
typedef struct
  {  Triple data[MAX_SIZE] ;    /*  非0元素的三元组表  */
     int     rpos[MAX_ROW];    /* 各行第一个非0位置表 */
     int     rn ,cn , tn ;      /*  矩阵的行、列数和非0元个数  */
  }RLSMatrix ;
```

稀疏矩阵的乘法

设有两个矩阵：$A=(a_{ij})_{m\times n}$，$B=(b_{ij})_{n\times p}$

则：$C=(c_{ij})_{m\times p}$　　其中 $c_{ij}=\sum a_{ik}\times b_{kj}$

$1\leqq k\leqq n$，$1\leqq i\leqq m$，$1\leqq j\leqq p$

经典算法是三重循环：

```
for (i=1;i<=m;++i)
  for(j=1;j<=p;++j)
    {  c[i][j]=0;
      for(k=1;k<=n;++k)
          c[i][j]=
      c[i][j]+a[i][k]×b[k][j];
    }
```

此算法的复杂度为O(m×n×p)。

设有两个稀疏矩阵$A=(a_{ij})_{m\times n}$，$B=(b_{ij})_{n\times p}$，其存储结构采用行逻辑链接的三元组顺序表。

算法思想：对于A中的每个元素a.data[p](p=1, 2, … , a.tn)，找到B中所有满足条件：

a.data[p].col=b.data[q].row的元素b.data[q]，求得a.data[p].value×b.data[q].value，该乘积是c_{ij}中的一部分。求得所有这样的乘积并累加求和就能得到c_{ij}。

为得到非0的乘积，只要对a.data[1…a.tn] 中每个元素(i，k，a_{ik})(1≦i≦a.rn，1≦k≦a.cn)，找到b.data中所有相应的元素(k，j，b_{kj})(1≦k≦b.rn，1≦j≦b.cn) 相乘即可。则必须知道矩阵B中第k行的所有非0元素，而b.rpos[ ]向量中提供了相应的信息。

b.rpos[row]指示了矩阵B的第row行中第一个非0元素在b.data[ ]中的位置(序号)，显然，b.rpos[row+1]-1指示了第row行中最后一个非0元素在b.data[ ]中的位置(序号)。最后一行中最后一个非0元素在b.data[ ]中的位置显然就是b.tn 。

两个稀疏矩阵相乘的算法如下：

```
void MultsMatrix(RLSMatrix a, RLSMatrix b, RLSMatrix c)
    /* 求矩阵A 、B的积C=A×B，采用行逻辑链接的顺序表 */
  {  elemtype ctemp[Max_Size];
     int p,q,arow,ccol,brow,t;
     if (a.cn!=b.rn)  {  printf("Error\n"); exit(0); }
```

```
else
  {   c.rn=a.rn ; c.cn=b. n ; c.tn=0 ;     /* 初始化C  */
    if (a.tn*b.tn!=0)        /* C 是非零矩阵  */
    {   for (arow=1 ; arow<=a.rn ; ++arow)
          {   ctemp[arow]=0 ;  /*  当前行累加器清零   */
                    c.rpos[arow]=c.tn+1;
    p=a.rops[arow];
              for ( ; p<a.rpos[arow+1];++p)
                          /* 对第arow行的每一个非0元素   */
                     {   brow=a.data[p].col ;
                                 /*   找到元素在b.data[]中的
    行号  */
                    if (brow<b.cn) t=( b.rpos[brow+1];
                    else t=b.tn+1 ;
```

```
        for (q=b.rpos[brow]; q<t; ++q)
            {  ccol=b.data[q].col;
                /*  积元素在c中的列号  */
    ctemp[ccol]+=a.data[p].value*b.data[q].value;
            }
    }     /* 求出c中第arow行中的非0元素   */
for (ccol=1; ccol<=c.cn; ++ccol)
    if ( ctemp[ccol] !=0 )
      {  if ( ++c.tn>MAX_SIZE)
            {  printf("Error\n"); exit(0);  }
         else
```

```
            c.data[c.tn]=(arow , ccol ,
            ctemp[ccol]) ;
               }
            }
      }
}
```

3　十字链表

对于稀疏矩阵，当非0元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。

矩阵中非0元素的结点所含的域有：行、列、值、行指针(指向同一行的下一个非0元)、列指针(指向同一列的下一个非0元)。其次，十字交叉链表还有一个头结点，结点的结构如图5-10所示。

(a)　结点结构　　(b)　头结点结构

图5-10　十字链表结点结构

由定义知，稀疏矩阵中同一行的非0元素的由right指针域链接成一个行链表， 由down指针域链接成一个列链表。则每个非0元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非0元素构成一个十字交叉的链表。称为十字链表。

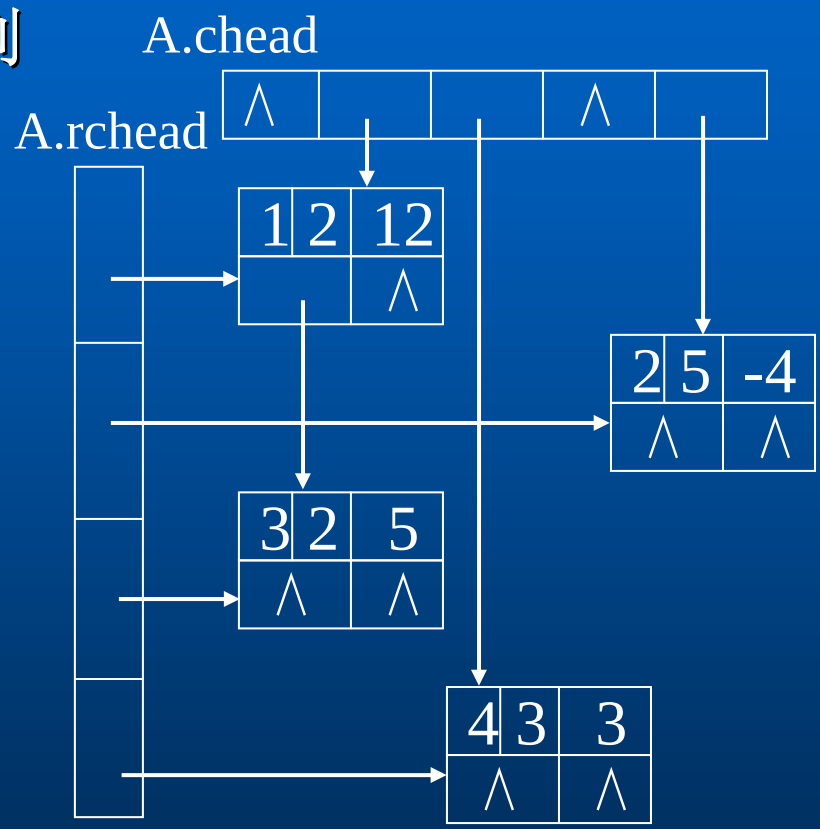
此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。对于图5-11(a)的稀疏矩阵A，对应的十字交叉链表如图5-11(b)所示，结点的描述如下：

```
typedef struct  Clnode
  {   int  row , col ;   /*  行号和列号  */
     elemtype value ;   /*  元素值  */
     struct  Clnode  *down , *right ;
  }OLNode ;   /*  非0元素结点  */
```

```
typedef struct  Clnode
  {  int  rn;        /*  矩阵的
行数  */
     int  cn;        /*  矩阵的列
     数  */
     int  tn;        /*  非0元素
     总数  */
     OLNode *rhead ;
     OLNode *chead;
  } CrossList ;
```

$$A=\begin{pmatrix} 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & -4 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

(a)　稀疏矩阵



(b)　稀疏矩阵的十字交叉链表

图5-11　稀疏矩阵及其十字交叉链表

5.4 广义表

广义表是线性表的推广和扩充，在人工智能领域中应用十分广泛。

在第2章中，我们把线性表定义为n(n≧0)个元素a_1, a_2 ,..., a_n的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项(Atom)。所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表的概念。

广义表(Lists，又称为列表)：是由n(n≧0)个元素组成的有穷序列： LS=(a_1，a_2，...，a_n)

其中a_i或者是原子项，或者是一个广义表。LS是广义表的名字，n为它的长度。若a_i是广义表，则称为LS的子表。

习惯上：原子用小写字母，子表用大写字母。

若广义表LS非空时：

- a_1(表中第一个元素)称为表头；
- 其余元素组成的子表称为表尾；(a_2，a_3，…，a_n)
- 广义表中所包含的元素(包括原子和子表)的个数称为表的长 度。
- 广义表中括号的最大层数称为表深 (度)。

有关广义表的这些概念的例子如表5-2所示。

表5-2　广义表及其示例

广　义　表	表长n	表深h
A=()	0	0
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞
F=(())	1	2

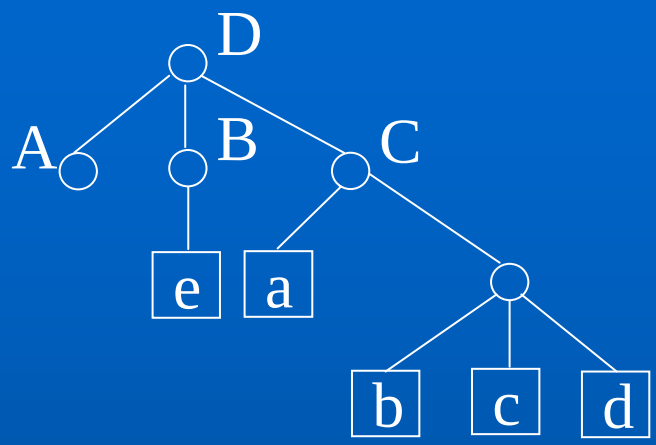


图5-12　广义表的图形表示

广义表的重要结论：

(1) 广义表的元素可以是原子，也可以是子表，子表的元素又可以是子表，…。即广义表是一个多层次的结构。

表5-2中的广义表D的图形表示如图5-12所示。

(2) 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。

(3) 广义表本身可以是一个递归表。

(4) 根据对表头、表尾的定义，任何一个非空广义表的表头可以是原子，也可以是子表，而表尾必定是广义表。

5.4.1 广义表的存储结构

由于广义表中的数据元素具有不同的结构，通常用链式存储结构表示，每个数据元素用一个结点表示。因此，广义表中就有两类结点：

- 一类是表结点，用来表示广义表项，由标志域，表头指针域，表尾指针域组成;
- 另一类是原子结点，用来表示原子项，由标志域，原子的值域组成。如图5-13所示。

只要广义表非空，都是由表头和表尾组成。即一个确定的表头和表尾就唯一确定一个广义表。

(a) 原子结点　　(b) 表结点

图5-13 广义表的链表结点结构示意图

相应的数据结构定义如下：

```
typedef struct GLNode
  { int  tag ;    /* 标志域，为1：表结点;为0 ： 原子结点  */
    union
      { elemtype value;    /* 原子结点的值域 */
        struct
          { struct GLNode  *hp , *tp ;
           }ptr ;  /* ptr和atom两成员共用  */
      }Gdata ;
  } GLNode ;    /* 广义表结点类型 */
```

例： 对A=()，B=(e)，C=(a, (b, c, d))，D=(A, B, C)，E=(a, E)的广义表的存储结构如图5-14所示。

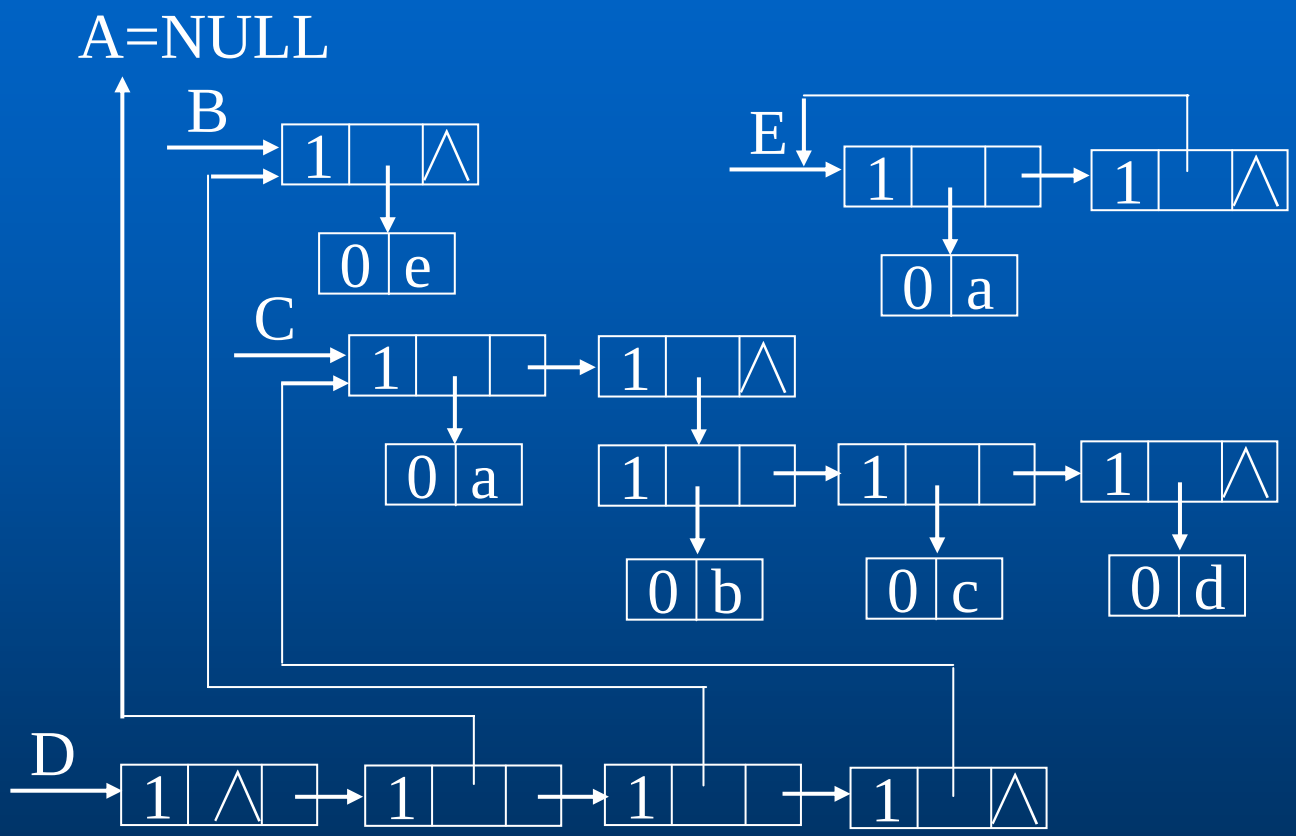


图5-14 广义表的存储结构示意图

对于上述存储结构，有如下几个特点：

(1) 若广义表为空，表头指针为空；否则，表头指针总是指向一个表结点，其中**hp**指向广义表的表头结点(或为原子结点，或为表结点），**tp**指向广义表的表尾(表尾为空时，指针为空，否则必为表结点)。

(2) 这种结构求广义表的长度、深度、表头、表尾的操作十分方便。

(3) 表结点太多，造成空间浪费。也可用图**5-15**所示的结点结构。

tag=0	原子的值	表尾指针tp

(a) 原子结点

tag=1	表头指针hp	表尾指针tp

(b) 表结点

图5-15 广义表的链表结点结构示意图

习 题 五

⑴ 什么是广义表？请简述广义表与线性表的区别？

⑵ 一个广义表是(a, (a, b), d, e, (a, (i, j), k))，请画出该广义表的链式存储结构。

⑶ 设有二维数组a[6][8]，每个元素占相邻的4个字节，存储器按字节编址，已知a的起始地址是1000，试计算：

① 数组a的最后一个元素a[5][7]起始地址；

② 按行序优先时，元素a[4][6]起始地址；

③ 按行序优先时，元素a[4][6]起始地址。

（4） 设A和B是稀疏矩阵，都以三元组作为存储结构，请写出矩阵相加的算法，其结果存放在三元组表C中，并分析时间复杂度。

（5）设有稀疏矩阵B如下图所示，请画出该稀疏矩阵的三元组表和十字链表存储结构。

$$A=\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

第6章 树和二叉树

树型结构是一类非常重要的非线性结构。直观地，树型结构是以分支关系定义的层次结构。

树在计算机领域中也有着广泛的应用，例如在编译程序中，用树来表示源程序的语法结构；在数据库系统中，可用树来组织信息；在分析算法的行为时，可用树来描述其执行过程等等。

本章将详细讨论树和二叉树数据结构，主要介绍树和二叉树的概念、术语，二叉树的遍历算法。树和二叉树的各种存储结构以及建立在各种存储结构上的操作及应用等。

6.1 树的基本概念

6.1.1 树的定义和基本术语

1 树的定义

树(Tree)是n(n≧0)个结点的有限集合T，若n=0时称为空树，否则：

⑴ 有且只有一个特殊的称为树的根(Root)结点；

⑵ 若n>1时，其余的结点被分为m(m>0)个互不相交的子集T_1，T_2，T_3...T_m，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。

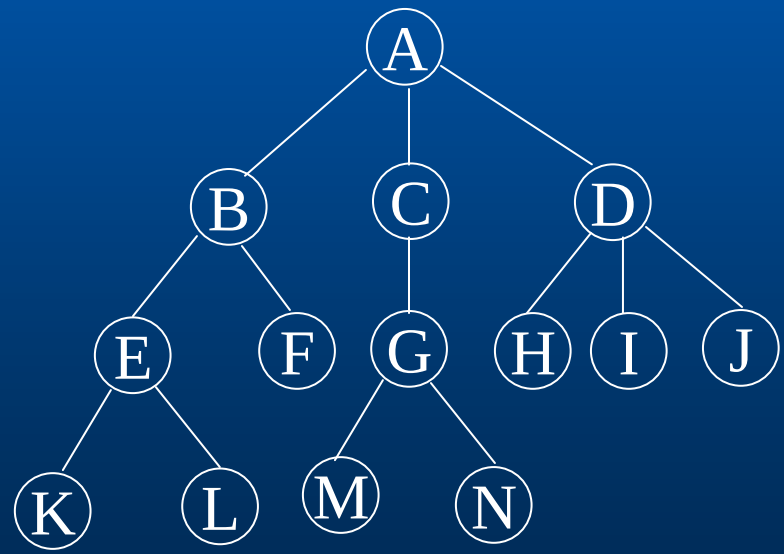
这是树的递归定义，即用树来定义树，而只有一个结点的树必定仅由根组成，如图6-1(a)所示。

2 树的基本术语

(1) **结点(node)**：一个数据元素及其若干指向其子树的分支。

(2) **结点的度(degree)**、**树的度**：结点所拥有的子树的棵数称为**结点的度**。树中结点度的最大值称为**树的度**。

(a) 只有根结点



(b) 一般的树

图6-1 树的示例形式

如图6-1(b)中结点A的度是3，结点B的度是2，结点M的度是0，树的度是3。

(3) 叶子(left)结点、非叶子结点：树中度为0的结点称为叶子结点(或终端结点)。相对应地，度不为0的结点称为非叶子结点(或非终端结点或分支结点)。除根结点外，分支结点又称为内部结点。

如图6-1(b)中结点H、I、J、K、L、M、N是叶子结点，而所有其它结点都是分支结点。

(4) 孩子结点、双亲结点、兄弟结点

一个结点的子树的根称为该结点的孩子结点(child)或子结点；相应地，该结点是其孩子结点的双亲结点(parent)或父结点。

如图6-1(b)中结点B 、C、D是结点A的子结点，而结点A是结点B 、C、D的父结点；类似地结点E 、F是结点B的子结点，结点B是结点E 、F的父结点。

同一双亲结点的所有子结点互称为兄弟结点。

如图6-1(b)中结点B 、C、D是兄弟结点；结点E 、F是兄弟结点。

(5) 层次、堂兄弟结点

规定树中根结点的层次为1，其余结点的层次等于其双亲结点的层次加1。

若某结点在第l($l \geqslant 1$)层，则其子结点在第l+1层。

双亲结点在同一层上的所有结点互称为堂兄弟结点。如图6-1(b)中结点E、F、G、H、I、J。

(6) 结点的层次路径、祖先、子孙

从根结点开始，到达某结点**p**所经过的所有结点成为结点**p**的层次路径(有且只有一条)。

结点**p**的层次路径上的所有结点（**p**除外）称为**p**的祖先**(ancester)** 。

以某一结点为根的子树中的任意结点称为该结点的子孙结点**(descent)**。

(7) 树的深度**(depth)**：树中结点的最大层次值，又称为树的高度，如图**6-1(b)**中树的高度为**4**。

(8) 有序树和无序树：对于一棵树，若其中每一个结点的子树（若有）具有一定的次序，则该树称为有序树，否则称为无序树。

(9) 森林(forest)：是m(m≧0)棵互不相交的树的集合。显然，若将一棵树的根结点删除，剩余的子树就构成了森林。

3 树的表示形式

(1) 倒悬树。是最常用的表示形式，如图6-1(b)。

(2) 嵌套集合。是一些集合的集体，对于任何两个集合，或者不相交，或者一个集合包含另一个集合。图6-2(a)是图6-1(b)树的嵌套集合形式。

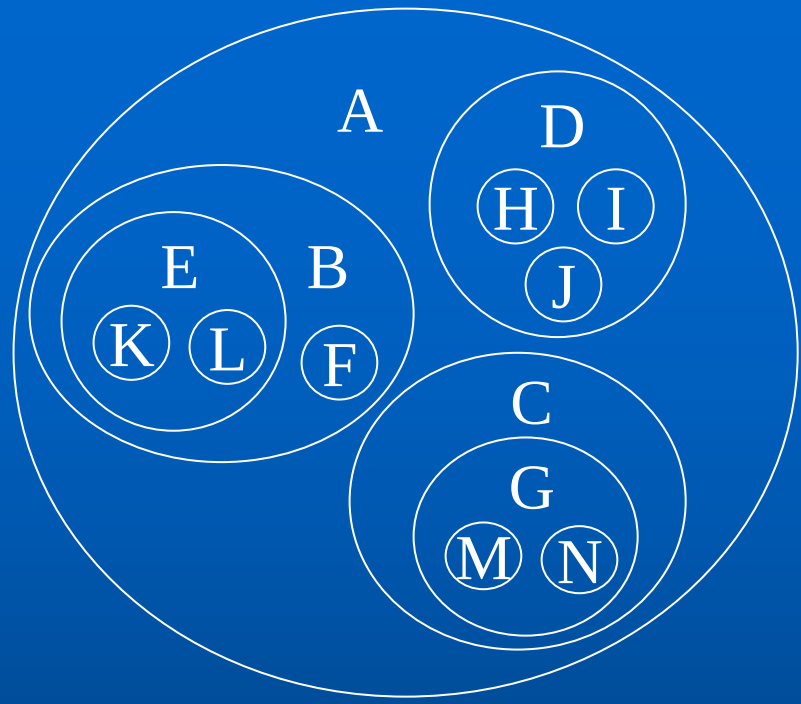
(3) 广义表形式。图6-2(b)是树的广义表形式。

(4) 凹入法表示形式。见P_{120}

树的表示方法的多样化说明了树结构的重要性。

(A(B(E(K,L),F),C(G(M,N)),D(H,I,J)

(b) 广义表形式



(a) 嵌套集合形式

图**6-2** 树的表示形式

6.1.2 树的抽象数据类型定义

ADT Tree{

数据对象**D**：**D**是具有相同数据类型的数据元素的集合。

数据关系**R**：若**D**为空集，则称为空树；

……

基本操作：

……

} ADT Tree

详见$p_{118\sim119}$。

6.2 二叉树

6.2.1 二叉树的定义

1 二叉树的定义

二叉树(Binary tree)是n(n≥0)个结点的有限集合。若n=0时称为空树，否则：

⑴ 有且只有一个特殊的称为树的根(Root)结点；

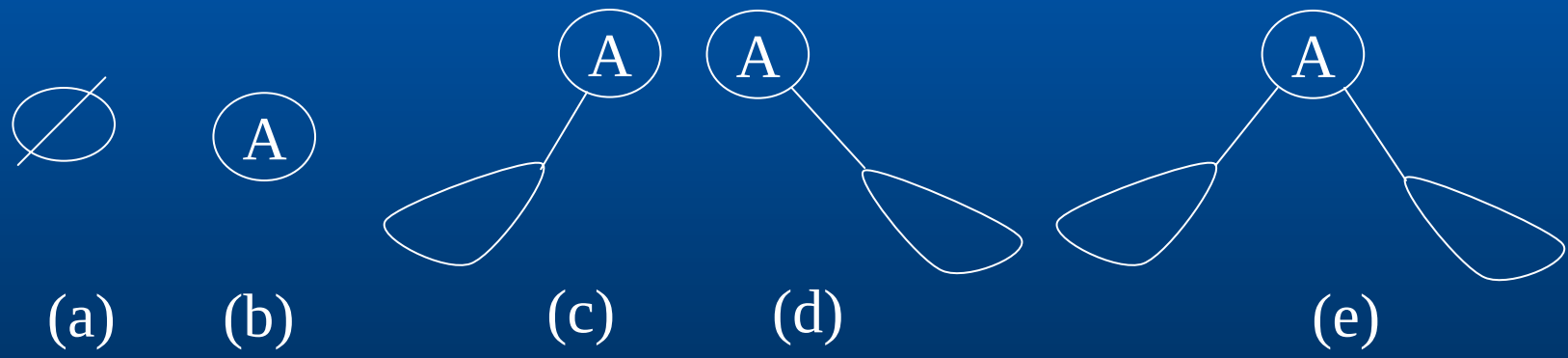
⑵ 若n>1时，其余的结点被分成为二个互不相交的子集T_1,T_2，分别称之为左、右子树，并且左、右子树又都是二叉树。

由此可知，二叉树的定义是递归的。

二叉树在树结构中起着非常重要的作用。因为二叉树结构简单，存储效率高，树的操作算法相对简单，且任何树都很容易转化成二叉树结构。上节中引入的有关树的术语也都适用于二叉树。

2　二叉树的基本形态

二叉树有**5**种基本形态，如图**6-3**所示。



(a) 空二叉树　**(b)** 单结点二叉树　**(c)** 右子树为空

(d) 左子树为空　**(e)** 左、右子树都不空

图**6-3**　二叉树的基本形态

质

性质1：在非空二叉树中，第i层上至多有2^{i-1}个结点$(i \geqq 1)$。

证明：用数学归纳法证明。

当$i=1$时：只有一个根结点，$2^{1-1}=2^0=1$，命题成立。

现假设对$i>1$时，处在第$i-1$层上至多有$2^{(i-1)-1}$个结点。

由归纳假设知，第$i-1$层上至多有2^{i-2}个结点。由于二叉树每个结点的度最大为2，故在第i层上最大结点数为第$i-1$层上最大结点数的2倍。

即　$2\times 2^{i-2}=2^{i-1}$　　证毕

性质2：深度为k的二叉树至多有2^k-1个结点（$k \geqq 1$）

证明：深度为k的二叉树的最大的结点数为二叉树中每层上的最大结点数之和。

由性质1知，二叉树的第1层、第2层…第k层上的结点数至多有：2^0、2^1 …2^{k-1} 。

∴ 总的结点数至多有：$2^0+2^1+\ldots+2^{k-1}=2^k-1$

证毕

性质3：对任何一棵二叉树，若其叶子结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。

证明：设二叉树中度为1的结点数为n_1，二叉树中总结点数为N，因为二叉树中所有结点均小于或等于2，则有：$N=n_0+n_1+n_2$

再看二叉树中的分支数：

除根结点外，其余每个结点都有唯一的一个进入分支，而所有这些分支都是由度为1和2的结点射出的。设B为二叉树中的分支总数，则有：　　$N=B+1$

∴　　$B=n_1+2\times n_2$

∴　　$N=B+1=n_1+2\times n_2+1$

∴　　$n_0+n_1+n_2=n_1+2\times n_2+1$

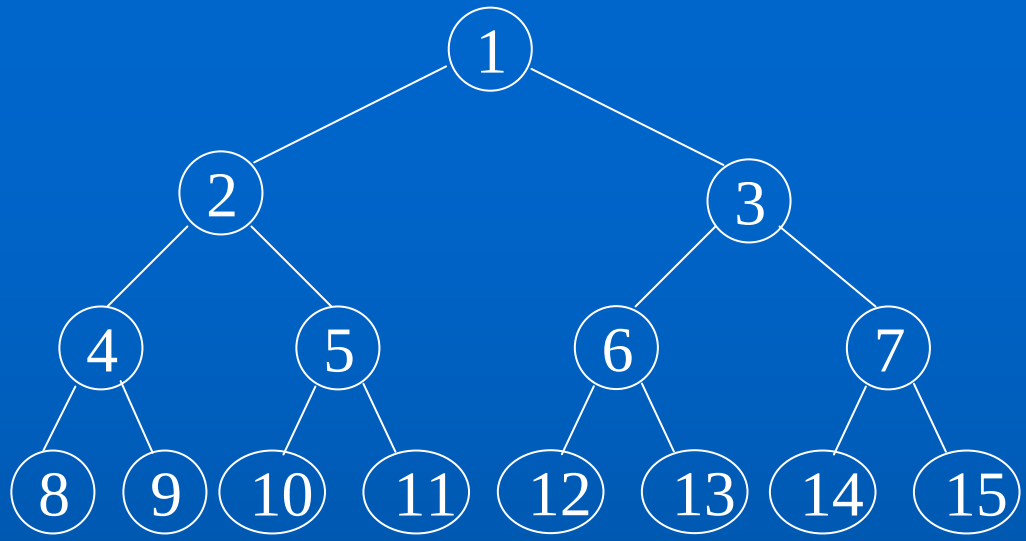
即　　$n_0=n_2+1$

证毕

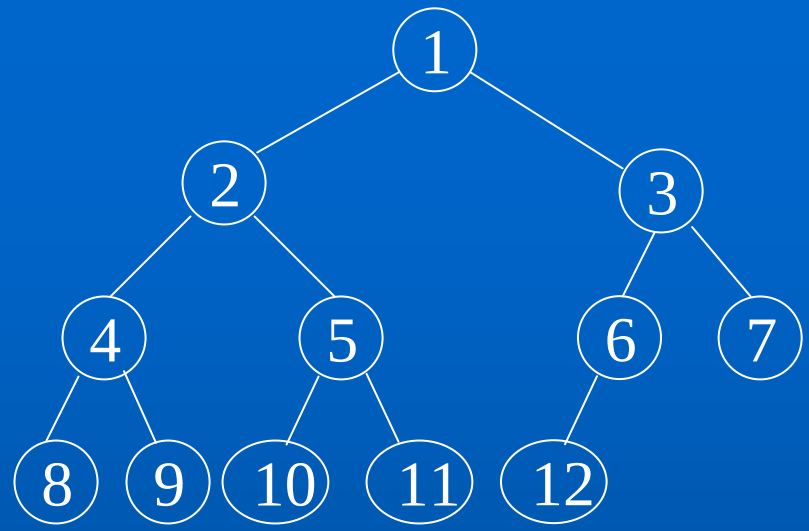
满二叉树和完全二叉树

一棵深度为k且有2^k-1个结点的二叉树称为满二叉树(Full Binary Tree)。

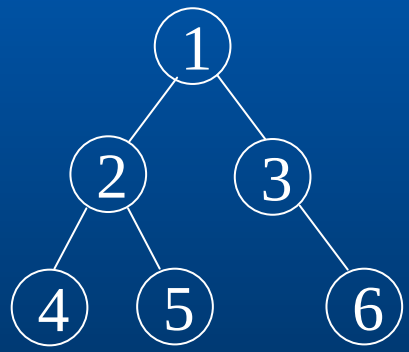
如图6-4(a)就是一棵深度为4的满二叉树。

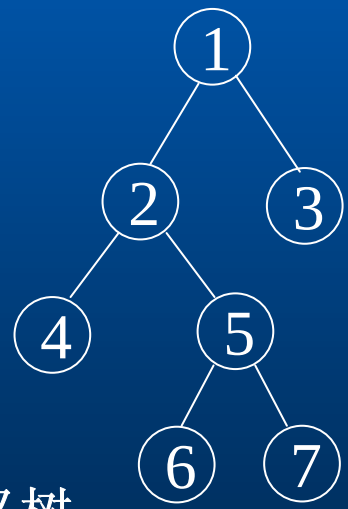


(a) 满二叉树



(b) 完全二叉树





(c) 非完全二叉树

图**6-4** 特殊形态的二叉树

满二叉树的特点：

- 基本特点是每一层上的结点数总是最大结点数。
- 满二叉树的所有的支结点都有左、右子树。
- 可对满二叉树的结点进行连续编号，若规定从根结点开始，按“自上而下、自左至右”的原则进行。

完全二叉树(Complete Binary Tree)：如果深度为k，由n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1到n的结点一一对应，该二叉树称为完全二叉树。

或深度为k的满二叉树中编号从1到n的前n个结点构成了一棵深度为k的完全二叉树。

其中 $2^{k-1} \leqq n \leqq 2^{k}-1$ 。

完全二叉树是满二叉树的一部分，而满二叉树是完全二叉树的特例。

完全二叉树的特点：

若完全二叉树的深度为**k**，则所有的叶子结点都出现在第k层或**k-1**层。对于任一结点，如果其右子树的最大层次为*l*，则其左子树的最大层次为*l*或*l*+1。

性质4：**n**个结点的完全二叉树深度为：$\lfloor \log_2 n \rfloor + 1$。

其中符号：$\lfloor x \rfloor$表示不大于**x**的最大整数。

$\lceil x \rceil$ 表示不小于**x**的最小整数。

证明：假设完全二叉树的深度为**k**，则根据性质**2**及完全二叉树的定义有：

$2^{k-1}-1<n\leqslant 2^{k}-1$ 或 $2^{k-1}\leqslant n<2^{k}$

取对数得：$k-1<\log_2 n<k$ 因为k是整数。

∴ $k=\lfloor \log_2 n \rfloor+1$ 证毕

性质5：若对一棵有n个结点的完全二叉树(深度为 $\lfloor \log_2 n \rfloor+1$)的结点按层（从第1层到第$\lfloor \log_2 n \rfloor+1$层)序自左至右进行编号，则对于编号为i（$1\leqslant i\leqslant n$)的结点：

(1) 若i=1：则结点i是二叉树的根，无双亲结点；否则，若i>1，则其双亲结点编号是$\lfloor i/2 \rfloor$。

(2) 如果2i>n：则结点i为叶子结点，无左孩子；否则，其左孩子结点编号是2i。

(3) 如果2i+1>n：则结点i无右孩子；否则，其右孩子结点编号是2i+1。

证明：用数学归纳法证明。首先证明(2)和(3)，然后由(2)和(3)导出(1)。

当i=1时，由完全二叉树的定义知，结点i的左孩子的编号是2，右孩子的编号是3。

若2>n，则二叉树中不存在编号为2的结点，说明结点i的左孩子不存在。

若3>n，则二叉树中不存在编号为3的结点，说明结点i的右孩子不存在。

现假设对于编号为j(1≤j≤i)的结点，(2)和(3)成立。即：

- 当2j≤n：结点j的左孩子编号是2j；当2j>n时，结点j的左孩子结点不存在。

◆ 当2j+1≦n：结点j的右孩子编号是2j+1；当2j+1>n时，结点j的右孩子结点不存在。

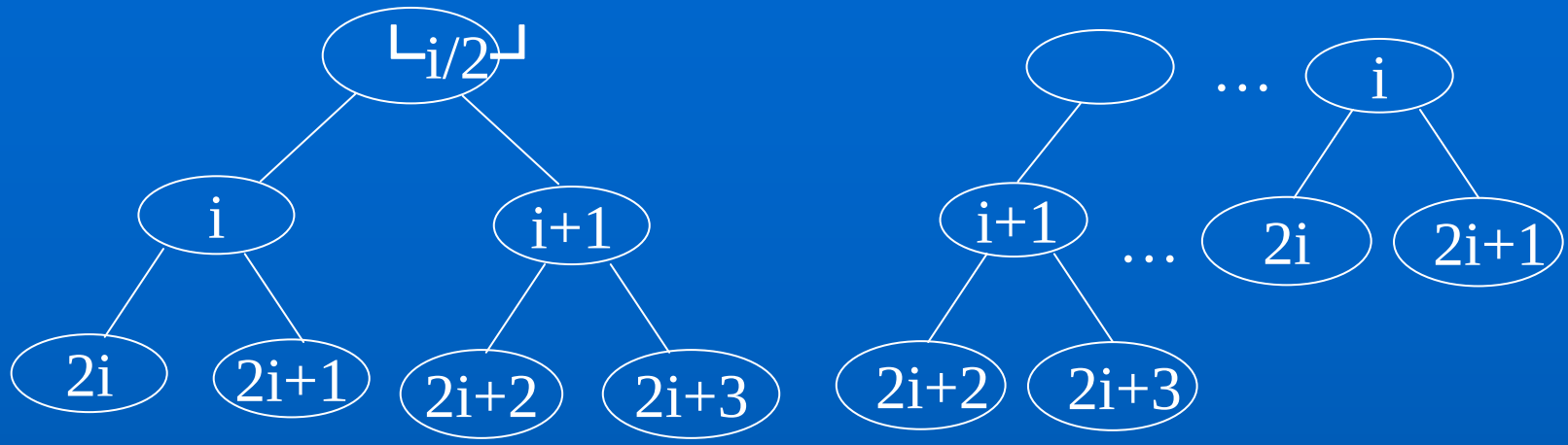
当i=j+1时，由完全二叉树的定义知，若结点i的左孩子结点存在，则其左孩子结点的编号一定等于编号为j的右孩子的编号加1，即结点i的左孩子的编号为：

$$(2j+1)+1=2(j+1)=2i$$

如图6-5所示，且有2i≦n。相反，若2i>n，则左孩子结点不存在。同样地，若结点i的右孩子结点存在，则其右孩子的编号为：2i+1，且有2i+1≦n。相反，若2i+1>n，则左孩子结点不存在。结论(2)和(3)得证。

再由(2)和(3)来证明(1)。

当i=1时，显然编号为1的是根结点，无双亲结点。



(a) i和i+1结点在同一层　　(b) i和i+1结点不在同一层

图6-5 完全二叉树中结点i和i+1的左右孩子

当i＞1时，设编号为i的结点的双亲结点的编号为m，若编号为i的结点是其双亲结点的左孩子，则由(2)有：

i＝2m，即m＝⌊i/2⌋；

若编号为i的结点是其双亲结点的右孩子，则由(3)有：

i＝2m＋1，即m＝⌊(i-1)/2⌋；

∴ 当i＞1时，其双亲结点的编号为⌊i/2⌋　　证

6.2.3　二叉树的存储结构

1　顺序存储结构

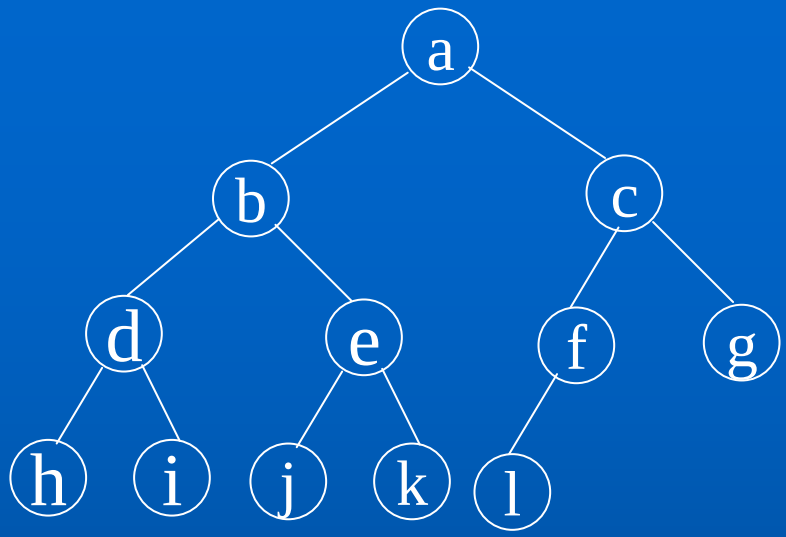
二叉树存储结构的类型定义：

```
#define MAX_SIZE  100
 typedef telemtype sqbitree[MAX_SIZE];
```

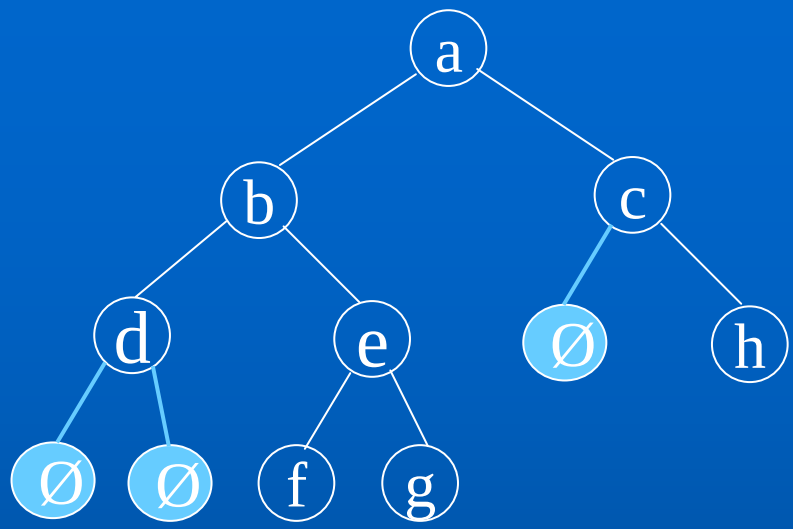
用一组地址连续的存储单元依次“自上而下、自左至右”存储完全二叉树的数据元素。

对于完全二叉树上编号为i的结点元素存储在一维数组的下标值为i-1的分量中，如图6-6(c)所示。

对于一般的二叉树，将其每个结点与完全二叉树上的结点相对照，存储在一维数组中，如图6-6(d)所示。



(a) 完全二叉树



(b) 非完全二叉树

1	2	3	4	5	6	7	8	9	10	11	12
a	b	c	d	e	f	g	h	i	j	k	l

(c) 完全二叉树的顺序存储形式

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	Ø	h	Ø	Ø	f	g

(d) 非完全二叉树的顺序存储形式

图**6-6** 二叉树及其顺序存储形式

最坏的情况下，一个深度为k且只有k个结点的单支树需要长度为2^k-1的一维数组。

2 链式存储结构

设计不同的结点结构可构成不同的链式存储结构。

(1) 结点的类型及其定义

① 二叉链表结点。有三个域：一个数据域，两个分别指向左右子结点的指针域，如图6-7(a)所示。

```
typedef struct BTNode
  {  ElemType  data ;
     struct BTNode  *Lchild , *Rchild ;
  }BTNode ;
```

② 三叉链表结点。除二叉链表的三个域外，再增加一个指针域，用来指向结点的父结点，如图6-7(b)所示。

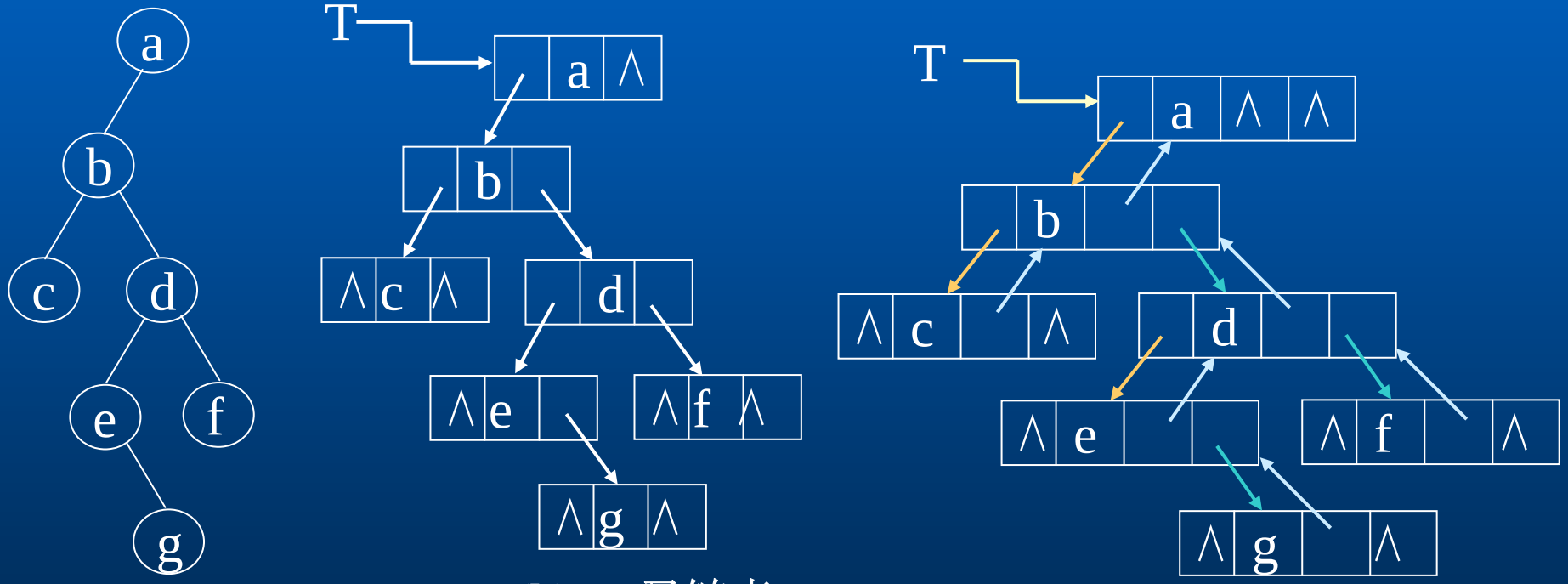
```
typedef struct BTNode_3
   {  ElemType  data ;
      struct BTNode_3  *Lchild , *Rchild , *parent ;
   }BTNode_3 ;
```

(a) 二叉链表结点　　(b) 三叉链表结点

图6-7　链表结点结构形式

(2) 二叉树的链式存储形式

例有一棵一般的二叉树，如图6-8(a)所示。以二叉链表和三叉链表方式存储的结构图分别如图6-8(b)、6-8(c)所示。



(a) 二叉树　　(b) 二叉链表　　(c) 三叉链表

图6-8　二叉树及其链式存储结构

6.3 遍历二叉树及其应用

遍历二叉树(Traversing Binary Tree)：是指按指定的规律对二叉树中的每个结点访问一次且仅访问一次。

所谓访问是指对结点做某种处理。如：输出信息、修改结点的值等。

二叉树是一种非线性结构，每个结点都可能有左、右两棵子树，因此，需要寻找一种规律，使二叉树上的结点能排列在一个线性队列上，从而便于遍历。

二叉树的基本组成：根结点、左子树、右子树。若能依次遍历这三部分，就是遍历了二叉树。

若以L、D、R分别表示遍历左子树、遍历根结点和遍历右子树，则有六种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD。若规定先左后右，则只有前三种情况三种情况，分别是：

DLR——先(根)序遍历。

LDR——中(根)序遍历。

LRD——后(根)序遍历。

对于二叉树的遍历，分别讨论递归遍历算法和非递归遍历算法。递归遍历算法具有非常清晰的结构，但初学者往往难以接受或怀疑，不敢使用。实际上，递归算法是由系统通过使用堆栈来实现控制的。而非递归算法中的控制是由设计者定义和使用堆栈来实现的。

6.3.1 先序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

(1) 访问根结点；

(2) 先序遍历左子树(递归调用本算法)；

(3) 先序遍历右子树(递归调用本算法)。

先序遍历的递归算法

```
void  PreorderTraverse(BTNode  *T)
   {  if  (T!=NULL)
         {  visit(T->data) ;      /*  访问根结点  */
            PreorderTraverse(T->Lchild) ;
            PreorderTraverse(T->Rchild) ;
         }
   }
```

说明： visit()函数是访问结点的数据域，其要求视具体问题而定。树采用二叉链表的存储结构，用指针变量T来指向。

2 非递归算法

设T是指向二叉树根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T；

⑴ 访问p所指向的结点；

⑵ q=p->Rchild，若q不为空，则q进栈；

⑶ p=p->Lchild，若p不为空，转(1)，否则转(4)；

⑷ 退栈到p，转(1)，直到栈空为止。

算法实现：

```
#define  MAX_NODE  50
void  PreorderTraverse( BTNode  *T)
    {  BTNode  *Stack[MAX_NODE] ,*p=T, *q ;
        int  top=0 ;
        if  (T==NULL)  printf(“ Binary Tree is Empty!\n”) ;
        else {  do
                    {  visit( p-> data ) ;   q=p->Rchild ;
                       if  ( q!=NULL )  stack[++top]=q ;
                       p=p->Lchild ;
                       if (p==NULL) { p=stack[top] ;  top-- ; }
                    }
                  while (p!=NULL) ;
                }
    }
```

6.3.2 中序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

(1) 中序遍历左子树(递归调用本算法)；

(2) 访问根结点；

(3) 中序遍历右子树(递归调用本算法)。

中序遍历的递归算法

```
void  InorderTraverse(BTNode  *T)
   {  if  (T!=NULL)
         {  InorderTraverse(T->Lchild) ;
            visit(T->data) ;       /*  访问根结点  */
            InorderTraverse(T->Rchild) ;
         }
   }     /*图6-8(a) 的二叉树，输出的次序是： cbegdfa */
```

2 非递归算法

设T是指向二叉树根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T

⑴ 若p不为空，p进栈， p=p->Lchild ；

⑵ 否则(即p为空)，退栈到p，访问p所指向的结点；

⑶ p=p->Rchild ，转(1)；

直到栈空为止。

算法实现：

```
#define MAX_NODE  50
void  InorderTraverse( BTNode  *T)
{  BTNode  *Stack[MAX_NODE] ,*p=T ;
    int  top=0 , bool=1 ;
    if  (T==NULL)  printf(“ Binary Tree is Empty!\n”) ;
   else  { do
              { while (p!=NULL)
                   {  stack[++top]=p ;  p=p->Lchild ;   }
                if  (top==0)  bool=0 ;
                else  {  p=stack[top] ;  top-- ;
                           visit( p->data ) ;  p=p->Rchild ; }
              }  while (bool!=0) ;
         }
 }
```

6.3.3 后序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

(1) 后序遍历左子树(递归调用本算法)；

(2) 后序遍历右子树(递归调用本算法)；

(3) 访问根结点 。

后序遍历的递归算法

```
void  PostorderTraverse(BTNode  *T)
   {  if  (T!=NULL)
         {  PostorderTraverse(T->Lchild) ;
            PostorderTraverse(T->Rchild) ;
            visit(T->data) ;      /*  访问根结点  */
         }
   }     /*图6-8(a)  的二叉树，输出的次序是：  cgefdba    */
```

遍历二叉树的算法中基本操作是访问结点，因此，无论是哪种次序的遍历，对有n个结点的二叉树，其时间复杂度均为O(n) 。

如图6-9所示的二叉树表示表达式：(a+b*(c-d)-e/f)

按不同的次序遍历此二叉树，将访问的结点按先后次序排列起来的次序是：

其先序序列为：　-+a*b-cd/ef

其中序序列为：　a+b*c-d-e/f

其后序序列为：　abcd-*+ef/-

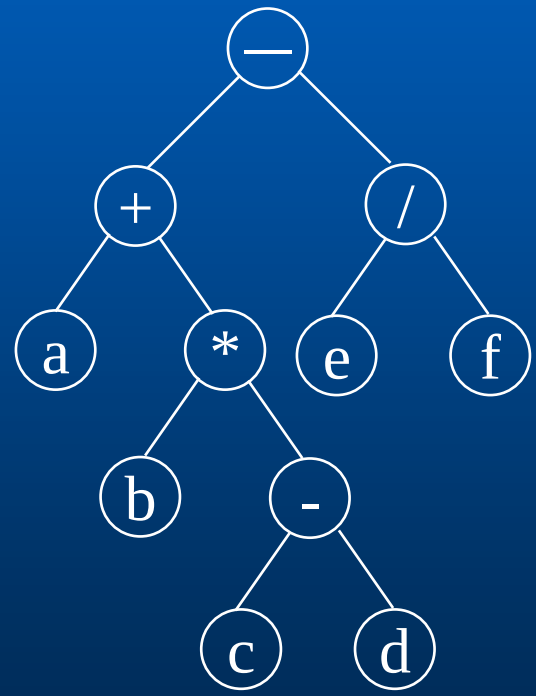


图6-9　表达式 (a+b*(c-d)-e/f)二叉树

2 非递归算法

在后序遍历中，根结点是最后被访问的。因此，在遍历过程中，当搜索指针指向某一根结点时，不能立即访问，而要先遍历其左子树，此时根结点进栈。当其左子树遍历完后再搜索到该根结点时，还是不能访问，还需遍历其右子树。所以，此根结点还需再次进栈，当其右子树遍历完后再退栈到到该根结点时，才能被访问。

因此，设立一个状态标志变量tag：

tag= { 0： 结点暂不能访问
       1： 结点可以被访问

其次，设两个堆栈S_1、S_2，S_1保存结点，S_2保存结点的状态标志变量tag。S_1和S_2共用一个栈顶指针。

设T是指向根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T；

⑴ 第一次经过根结点p，不访问：

p进栈S1， tag 赋值0，进栈S2，p=p->Lchild 。

⑵ 若p不为空，转(1)，否则，取状态标志值tag：

⑶ 若tag=0：对栈S1，不访问，不出栈；修改S2栈顶元素值(tag赋值1)，取S1栈顶元素的右子树，即p=S1[top]->Rchild，转(1)；

⑷ 若tag=1：S1退栈，访问该结点；

直到栈空为止。

算法实现：

```
#define MAX_NODE  50
void  PostorderTraverse( BTNode  *T)
   {  BTNode  *S1[MAX_NODE] ,*p=T ;
      int S2[MAX_NODE] , top=0 , bool=1 ;
      if  (T==NULL)  printf("Binary Tree is Empty!\n") ;
      else  { do
                {   while (p!=NULL)
                      {  S1[++top]=p ; S2[top]=0 ;
                         p=p->Lchild ;
                      }
                  if  (top==0)  bool=0 ;
```

```
            else if  (S2[top]==0)
                  {  p=S1[top]->Rchild ;  S2[top]=1 ;   }
               else
                  {  p=S1[top] ;  top-- ;
                     visit( p->data ) ; p=NULL ;
                      /*  使循环继续进行而不至于死循环 */
                   }
        }  while (bool!=0) ;
     }
}
```

6.3.4 层次遍历二叉树

层次遍历二叉树，是从根结点开始遍历，按层次次序“自上而下，从左至右”访问树中的各结点。

为保证是按层次遍历，必须设置一个队列，初始化时为空。

设T是指向根结点的指针变量，层次遍历非递归算法是：

若二叉树为空，则返回；否则，令p=T，p入队；

(1) 队首元素出队到p；

(2)访问p所指向的结点；

(3)将p所指向的结点的左、右子结点依次入队。直到队空为止。

```
#define MAX_NODE  50
void  LevelorderTraverse( BTNode  *T)
    {  BTNode  *Queue[MAX_NODE] ,*p=T ;
       int  front=0 , rear=0 ;
       if  (p!=NULL)
          {  Queue[++rear]=p;    /*   根结点入队  */
             while (front<rear)
                {  p=Queue[++front];  visit( p->data );
                   if (p->Lchild!=NULL)
                        Queue[++rear]=p;    /*   左结点入队  */
                   if (p->Rchild!=NULL)
                        Queue[++rear]=p;    /*   左结点入队  */
                 }
          }
    }
```

6.3.5 二叉树遍历算法的应用

“遍历”是二叉树最重要的基本操作，是各种其它操作的基础，二叉树的许多其它操作都可以通过遍历来实现。如建立二叉树的存储结构、求二叉树的结点数、求二叉树的深度等。

1 二叉树的二叉链表创建

(1) 按满二叉树方式建立 (补充)

在此补充按满二叉树的方式对结点进行编号建立链式二叉树。对每个结点，输入**i**、**ch**。

- **i**： 结点编号，按从小到大的顺序输入
- **ch**： 结点内容，假设是字符

在建立过程中借助一个一维数组**S[n]**，编号为**i**的结点保存在**S[i]**中。

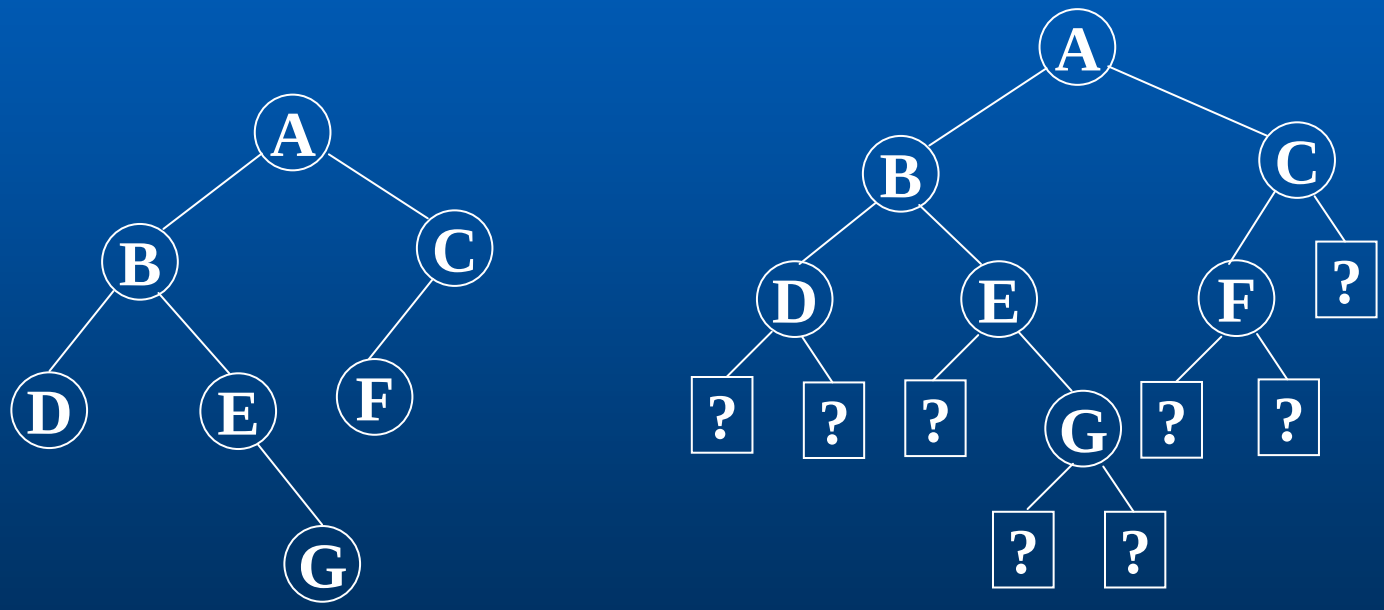
算法实现：

```
#define MAX_NODE  50
typedef struct BTNode
   {  char  data ;
      struct BTNode *Lchild , *Rchild ;
   }BTNode ;
BTNode  *Create_BTree(void)
   /*  建立链式二叉树，返回指向根结点的指针变量  */
   {  BTNode  *T , *p , *s[MAX_NODE] ;
      char ch ; int i , j ;
      while (1)
         {  scanf("%d", &i) ;
            if  (i==0)  break ;   /*  以编号0作为输入结束  */
            else
                {  ch=getchar() ;
```

```
            p=(BTNode *)malloc(sizeof(BTNode)) ;
            p–>data=ch ;
            p–>Lchild=p–>Rchild=NULL ;  s[i]=p ;
            if (i==1)  T=p ;
            else
               {  j=i/2 ;    /*    j是i的双亲结点编号  */
                  if (i%2==0)  s[j]->Lchild=p ;
                   else  s[j]->Rchild=p ;
                }
         }
      }
   return(T) ;
}
```

(2) 按先序遍历方式建立

对一棵二叉树进行“扩充”，就可以得到有该二叉树所扩充的二叉树。有两棵二叉树T_1及其扩充的二叉树T_2如图6-10所示。



(a) 二叉树T_1　　(b) T_1的扩充二叉树T_2

图6-10　二叉树T_1及其扩充二叉树T_2

二叉树的扩充方法是：在二叉树中结点的每一个空链域处增加一个扩充的结点(总是叶子结点，用方框“□”表示)。对于二叉树的结点值：

- 是char类型：扩充结点值为“？”；
- 是int类型：扩充结点值为0或-1；

下面的算法是二叉树的前序创建的递归算法，读入一棵二叉树对应的扩充二叉树的前序遍历的结点值序列。每读入一个结点值就进行分析：

- 若是扩充结点值：令根指针为NULL；
- 若是(正常)结点值：动态地为根指针分配一个结点，将该值赋给根结点，然后递归地创建根的左子树和右子树。

算法实现：

```
#define NULLKY  '?'
#define MAX_NODE   50
typedef struct BTNode
   {  char  data ;
      struct BTNode *Lchild , *Rchild ;
   }BTNode ;
BTNode  *Preorder_Create_BTree(BTNode  *T)
    /*  建立链式二叉树，返回指向根结点的指针变量  */
   {  char ch ;
      ch=getchar() ; getchar();
      if  (ch==NULLKY)
         {    T=NULL;  return(T) ;  }
```

```
    else
        {  T=(BTNode *)malloc(sizeof(BTNode)) ;
           T–>data=ch ;
           Preorder_Create_BTree(T->Lchild) ;
           Preorder_Create_BTree(T->Rchild) ;
           return(T) ;
        }
}
```

当希望创建图6-10(a)所示的二叉树时，输入的字符序列应当是：

ABD??E?G??CF???

2 求二叉树的叶子结点数

可以直接利用先序遍历二叉树算法求二叉树的叶子结点数。只要将先序遍历二叉树算法中vist()函数简单地进行修改就可以。

算法实现：

```
#define MAX_NODE 50
int search_leaves( BTNode *T)
  { BTNode *Stack[MAX_NODE] ,*p=T;
    int top=0, num=0;
    if (T!=NULL)
```

```
    {  stack[++top]=p ;
       while (top>0)
          {  p=stack[top--] ;
             if (p->Lchild==NULL&&p-
          >Rchild==NULL)  num++ ;
                if  (p->Rchild!=NULL )
                    stack[++top]=p->Rchild;
                if  (p->Lchild!=NULL )
                      stack[++top]=p->Lchild;
             }
       }
   return(num) ;
}
```

3 求二叉树的深度

利用层次遍历算法可以直接求得二叉树的深度。

算法实现：

```
#define MAX_NODE 50
int search_depth( BTNode *T)
  { BTNode *Stack[MAX_NODE] ,*p=T;
    int front=0 , rear=0, depth=0, level ;
      /* level总是指向访问层的最后一个结点在队列的位置 */
    if (T!=NULL)
      { Queue[++rear]=p;   /* 根结点入队 */
        level=rear ;   /* 根是第1层的最后一个节点 */
```

```
        while (front<rear)
            {  p=Queue[++front];
               if (p->Lchild!=NULL)
                    Queue[++rear]=p;    /*  左结点入队  */
               if (p->Rchild!=NULL)
                    Queue[++rear]=p;    /*  左结点入队  */
              if (front==level)
                 /*  正访问的是当前层的最后一个结点  */
                { depth++ ;  level=rear ;  }
             }
      }
}
```

6.4 线索树

遍历二叉树是按一定的规则将树中的结点排列成一个线性序列，即是对非线性结构的线性化操作。如何找到遍历过程中动态得到的每个结点的直接前驱和直接后继(第一个和最后一个除外)?如何保存这些信息?

设一棵二叉树有n个结点，则有n-1条边(指针连线)， 而n个结点共有2n个指针域(Lchild和Rchild)，显然有n+1个空闲指针域未用。则可以利用这些空闲的指针域来存放结点的直接前驱和直接后继信息。

对结点的指针域做如下规定：

- 若结点有左孩子，则**Lchild**指向其左孩子，否则，指向其直接前驱；
- 若结点有右孩子，则**Rchild**指向其右孩子，否则，指向其直接后继；

为避免混淆，对结点结构加以改进，增加两个标志域，如图**6-10**所示。

Lchild	Ltag	data	Rchild	Rtag

图**6-10** 线索二叉树的结点结构

$$\text{Ltag}=\begin{cases}0：\textbf{Lchild}域指示结点的左孩子\\ 1：\textbf{Lchild}域指示结点的前驱\end{cases}$$

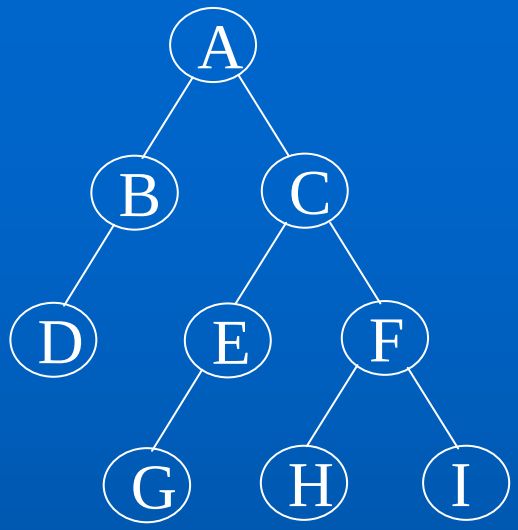
$$\text{Rtag}=\begin{cases}0：\textbf{Rchild}域指示结点的右孩子\\ 1：\textbf{Rchild}域指示结点的后继\end{cases}$$

用这种结点结构构成的二叉树的存储结构；叫做线索链表；指向结点前驱和后继的指针叫做线索；按照某种次序遍历，加上线索的二叉树称之为线索二叉树。

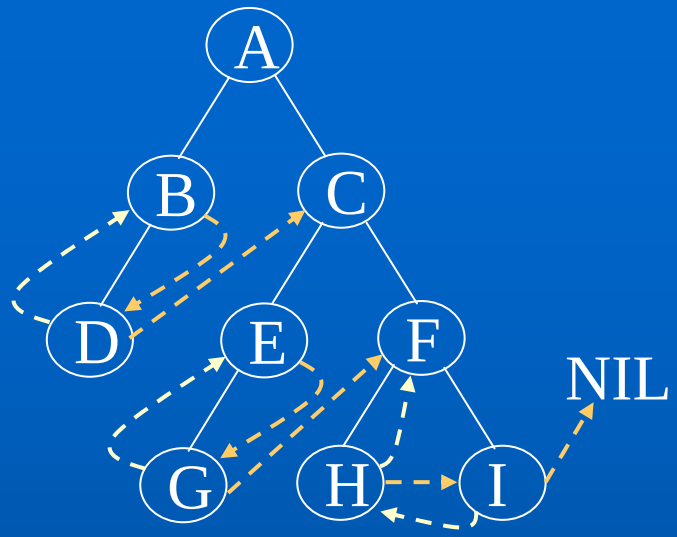
线索二叉树的结点结构与示例

```
typedef struct BiThrNode
   {   ElemType  data;
       struct BiTreeNode *Lchild , *Rchild ;
       int  Ltag , Rtag ;
   }BiThrNode ;
```

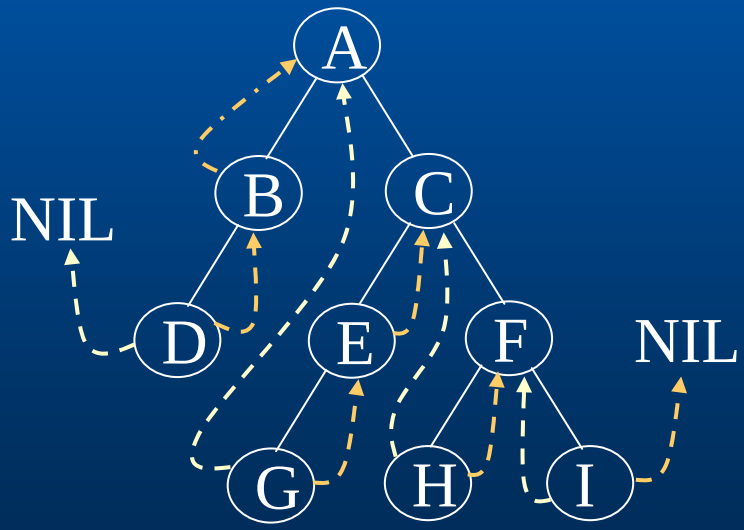
如图6-11是二叉树及相应的各种线索树示例。



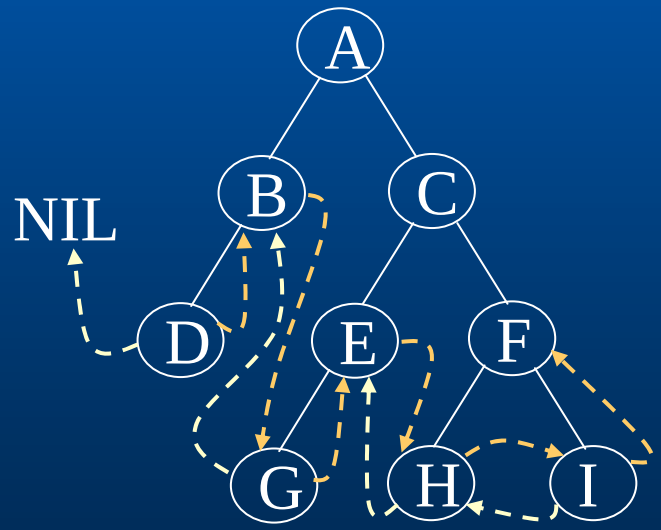
(a) 二叉树



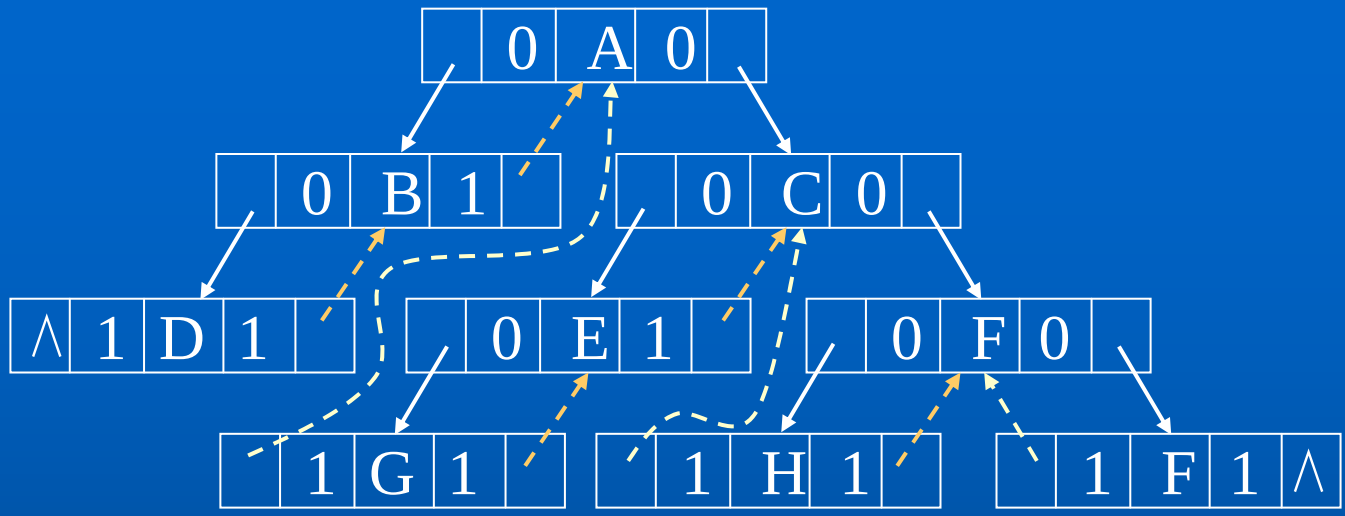
(b) 先序线索树的逻辑形式
结点序列：**ABDCEGFHI**



(c) 中序线索树的逻辑形式
结点序列：**DBAGECHFI**



(d) 后序线索树的逻辑形式
结点序列：**DBGEHIFCA**



(e) 中序线索树的链表结构

图**6-11** 线索二叉树及其存储结构

说明：画线索二叉树时，实线表示指针，指向其左、右孩子；虚线表示线索，指向其直接前驱或直接后继。

在线索树上进行遍历，只要先找到序列中的第一个结点，然后就可以依次找结点的直接后继结点直到后继为空为止。

如何在线索树中找结点的直接后继?以图6-11(d)，(e)所示的中序线索树为例：

- 树中所有叶子结点的右链都是线索。右链直接指示了结点的直接后继，如结点G的直接后继是结点E。
- 树中所有非叶子结点的右链都是指针。根据中序遍历的规律，非叶子结点的直接后继是遍历其右子树时访问的第一个结点，即右子树中最左下的(叶子)结点。如结点C的直接后继：沿右指针找到右子树的根结点F，然后沿左链往下直到Ltag=1的结点即为C的直接后继结点H。

如何在线索树中找结点的直接前驱?若结点的**Ltag=1**，则左链是线索，指示其直接前驱；否则，遍历左子树时访问的最后一个结点(即沿左子树中最右往下的结点）为其直接前驱结点。

对于后序遍历的线索树中找结点的直接后继比较复杂，可分以下三种情况：

- 若结点是二叉树的根结点：其直接后继为空；
- 若结点是其父结点的左孩子或右孩子且其父结点没有右子树：直接后继为其父结点；
- 若结点是其父结点的左孩子且其父结点有右子树：直接后继是对其父结点的右子树按后序遍历的第一个结点。

6.4.1 线索化二叉树

二叉树的线索化指的是依照某种遍历次序使二叉树成为线索二叉树的过程。

线索化的过程就是在遍历过程中修改空指针使其指向直接前驱或直接后继的过程。

仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点head，头结点的指针域的安排是：

- Lchild域：指向二叉树的根结点；
- Rchild域：指向中序遍历时的最后一个结点；
- 二叉树中序序列中的第一个结点Lchild指针域和最后一个结点Rchild指针域均指向头结点head

如同为二叉树建立了一个双向线索链表，对一棵线索二叉树既可从头结点也可从最后一个结点开始按寻找直接后继进行遍历。显然，这种遍历不需要堆栈，如图6-12所示。结点类型定义

```
#define  MAX_NODE   50
typedef enmu{Link , Thread} PointerTag ;
   /*  Link=0表示指针，  Thread=1表示线索   */
typedef struct BiThrNode
   {   ElemType  data;
       struct BiTreeNode *Lchild , *Rchild ;
       PointerTag  Ltag , Rtag ;
   }BiThrNode;
```

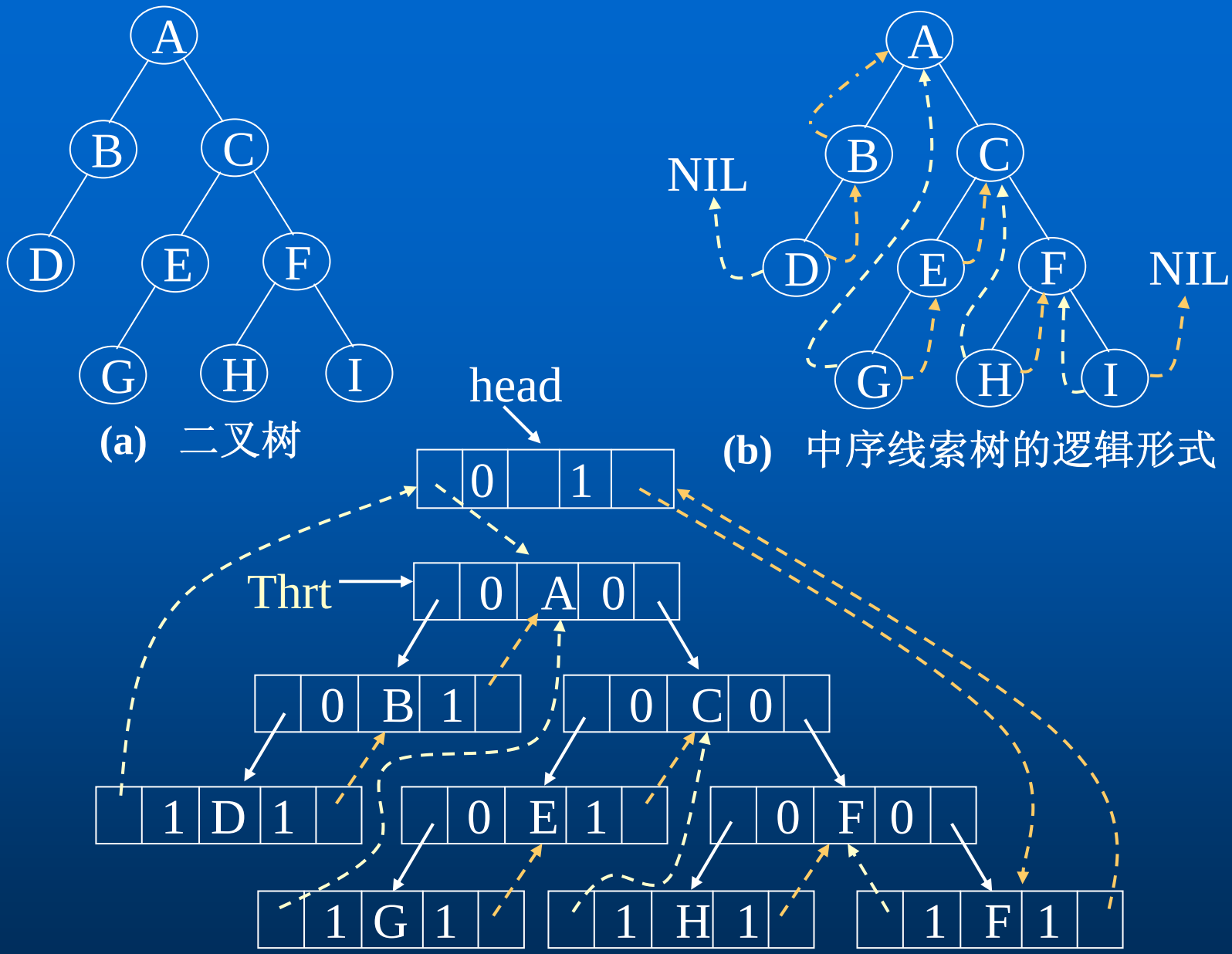


图6-12 中序线索二叉树及其存储结构

1 先序线索化二叉树

```
void preorder_Threading(BiThrNode *T)
    {  BiThrNode  *stack[MAX_NODE];
       BiThrNode  *last=NULL, *p ;
       int top=0 ;
       if  (T!=NULL)
          {  stack[++top]=T;
             while (top>0)
                {  p=stack[top--] ;
                   if (p->Lchild!=NULL)  p->Ltag=0 ;
                   else  {  p->Ltag=1 ;  p->Lchild!=last ;  }
                   if  (last!=NULL)
                      if (last->Rchild!=NULL) last->Rtag=0 ;
```

```
                else
                  {  last->Rtag=1 ; last->Rchild!=p ;  }
              last=p ;
              if (p->Rchild!=NULL)
                 stack[++top]=p->Rchild ;
              if (p->Lchild!=NULL)
                  stack[++top]=p->Lchild ;
            }
         Last->Rtag=1;  /*   最后一个结点是叶子结点 */
       }
}
```

2 中序线索化二叉树

```
void inorder_Threading(BiThrNode *T)
    {  BiThrNode  *stack[MAX_NODE];
       BiThrNode  *last=NULL, *p=T ;
       int top=0 ;
       while  (p!=NULL||top>0)
          if  (p!=NULL)  {  stack[++top]=p;  p=p->Lchild;
          }
          else
             {  p=stack[top--] ;
                if (p->Lchild!=NULL)  p->Ltag=0 ;
                else  {  p->Ltag=1 ;  p->Lchild!=last ;  }
                if  (last!=NULL)
                   if (last->Rchild!=NULL) last->Rtag=0 ;
```

```
                else {  last->Rtag=1 ; last->Rchild!=p ;  }
            last=p ;
            P=p->Rchild;
        }
    last->Rtag=1;  /*   最后一个结点是叶子结点   */
  }
}
```

6.4.2 线索二叉树的遍历

在线索二叉树中，由于有线索存在，在某些情况下可以方便地找到指定结点在某种遍历序列中的直接前驱或直接后继。此外，在线索二叉树上进行某种遍历比在一般的二叉树上进行这种遍历要容易得多，不需要设置堆栈，且算法十分简洁。

1 先序线索二叉树的先序遍历

```
void preorder_Thread_bt(BiThrNode *T)
    {  BiThrNode  *p=T ;
       while (p!=NULL)
          {  visit(p->data) ;
             if (p->Ltag==0)  p=p->Lchild ;
             else  p=p->Rchild
          }
    }
```

2 中序线索二叉树的中序遍历

```
void inorder_Thread_bt(BiThrNode *T)
    {  BiThrNode  *p ;
       if  (T!=NULL)
          {   p=T;
              while  (p->Ltag==0 )
                  p=p->Lchild;   /*  寻找最左的结点  */
              while  (p!=NULL)
                 {   visit(p->data) ;
                     if  (p->Rtag==1)
                         p=p->Rchild ;    /*  通过右线索找到后继  */
                     else    /*  否则，右子树的最左结点为后继  */
                       {  p=p->Rchild ;
```

```
                    while (p->Ltag==0 )  p=p->Lchild;
                }
            }
        }
    }
```

6.5 树与森林

本节将讨论树的存储结构、树及森林与二叉树之间的相互转换、树的遍历等。

6.5.1 树的存储结构

树的存储结构根据应用的不同而不同。

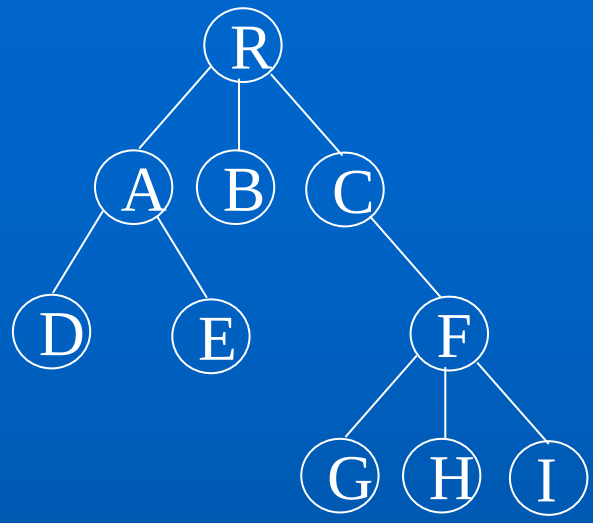
1 双亲表示法(顺序存储结构)

用一组连续的存储空间来存储树的结点，同时在每个结点中附加一个指示器(整数域)，用以指示双亲结点的位置(下标值)。数组元素及数组的类型定义如下：

```
#define MAX_SIZE  100
typedef  struct PTNode
   {  ElemType  data ;
      int  parent ;
   }PTNode ;
```

```
typedef  struct
   {  PTNode  Nodes[MAX_SIZE] ;
      int  root;    /*  根结点位置  */
      int  num ;   /*  结点数   */
   }Ptree ;
```

图6-13所示是一棵树及其双亲表示的存储结构。这种存储结构利用了任一结点的父结点唯一的性质。可以方便地直接找到任一结点的父结点，但求结点的子结点时需要扫描整个数组。



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

图6-13 树的双亲存储结构

2 孩子链表表示法

树中每个结点有多个指针域，每个指针指向其一棵子树的根结点。有两种结点结构。

(1) 定长结点结构

指针域的数目就是树的度。

其特点是：链表结构简单，但指针域的浪费明显。结点结构如图**6-14(a)** 所示。在一棵有**n**个结点，度为**k**的树中必有**n(k-1)+1**空指针域。

(2) 不定长结点结构

树中每个结点的指针域数量不同，是该结点的度，如图**6-14(b)** 所示。没有多余的指针域，但操作不便。

(a) 定长结点结构

(b) 不定长结点结构

图6-14 孩子表示法的结点结构

(3) 复合链表结构

对于树中的每个结点，其孩子结点用带头结点的单链表表示，表结点和头结点的结构如图6-15所示。

n个结点的树有n个(孩子)单链表(叶子结点的孩子链表为空)，而n个头结点又组成一个线性表且以顺序存储结构表示。

(a) 头结点

(b) 表结点

图6-15 孩子链表结点结构

数据结构类型定义如下：

```
#define  MAX_NODE  100
typedef  struct  listnode
   {   int   childno ;    /*  孩子结点编号  */
       struct listno  *next ;
   }CTNode;    /*  表结点结构  */
typedef  struct
   {  ElemType   data ;
       CTNode  *firstchild ;
   }HNode;    /*  头结点结构  */
```

```
typedef  struct
   {  HNode   nodes[MAX_NODE] ;
      int  root;   /*  根结点位置  */
      int  num ;  /*  结点数   */
   }CLinkList;   /*  头结点结构  */
```

图6-13所示的树T的孩子链表表示的存储结构如图6-16所示。

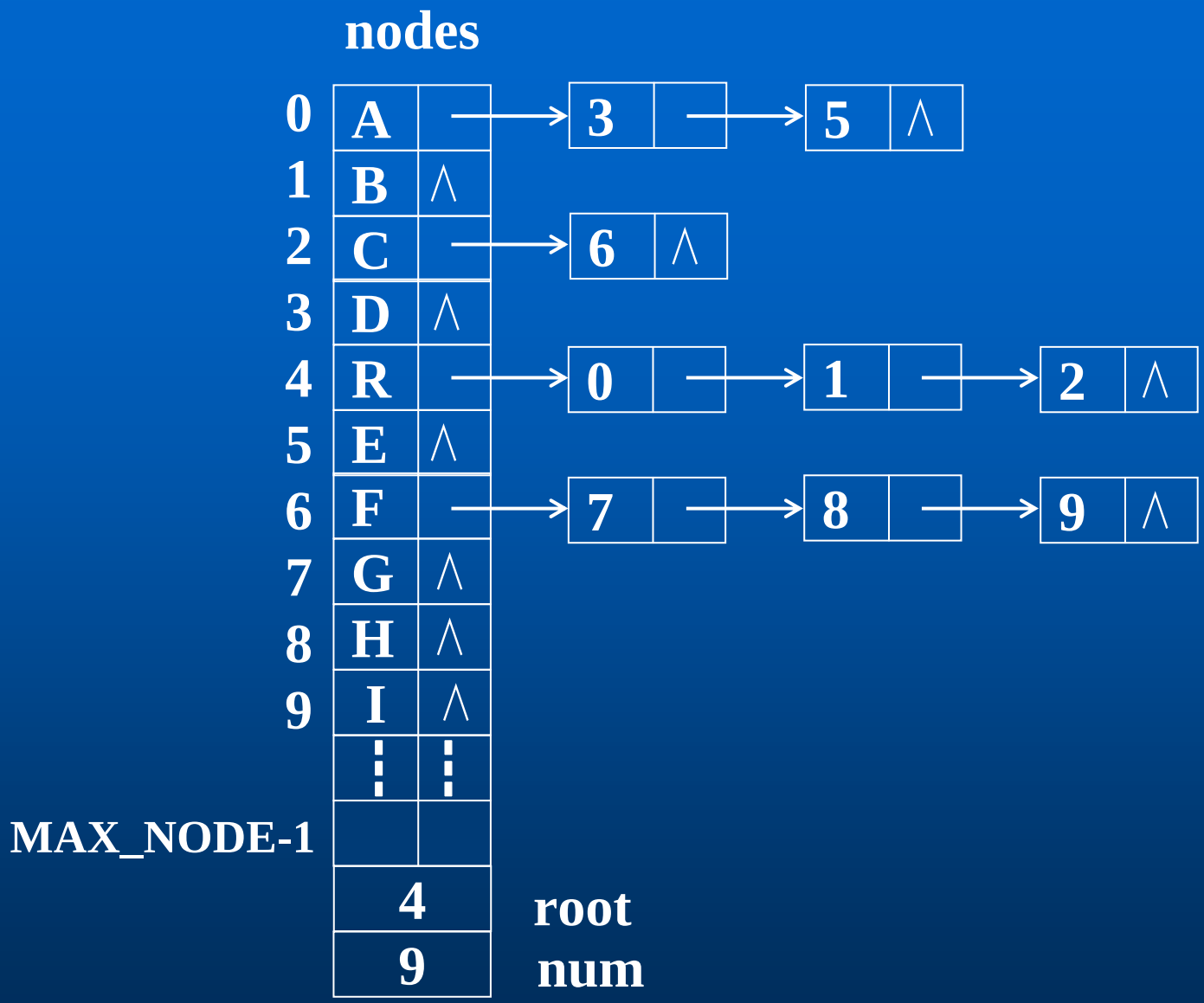


图6-16　图6-13的树T的孩子链表存储结构

3 孩子兄弟表示法(二叉树表示法)

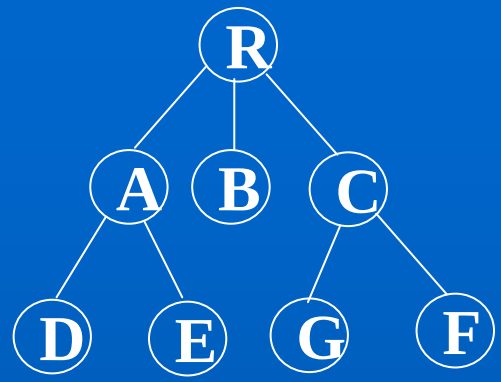
以二叉链表作为树的存储结构，其结点形式如图6-17(a)所示。

两个指针域：分别指向结点的第一个子结点和下一个兄弟结点。结点类型定义如下：

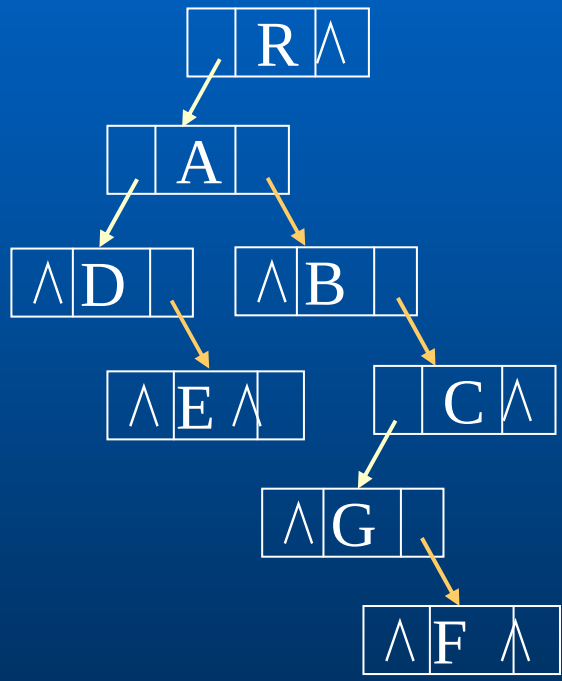
```
typedef  struct   CSnode
  {  ElemType   data ;
     struct   CSnode *firstchild, *nextsibing ;
  }CSNode;
```

图6-17(b)所示树的孩子兄弟表示的存储结构如图6-17(c)。

(a) 结点结构



(b) 树



(c) 孩子兄弟存储结构

图6-17 树及孩子兄弟存储结构

6.5.2 森林与二叉树的转换

由于二叉树和树都可用二叉链表作为存储结构，对比各自的结点结构可以看出，以二叉链表作为媒介可以导出树和二叉树之间的一个对应关系。

- 从物理结构来看，树和二叉树的二叉链表是相同的，只是对指针的逻辑解释不同而已。
- 从树的二叉链表表示的定义可知，任何一棵和树对应的二叉树，其右子树一定为空。

图6-18直观地展示了树和二叉树之间的对应关系。

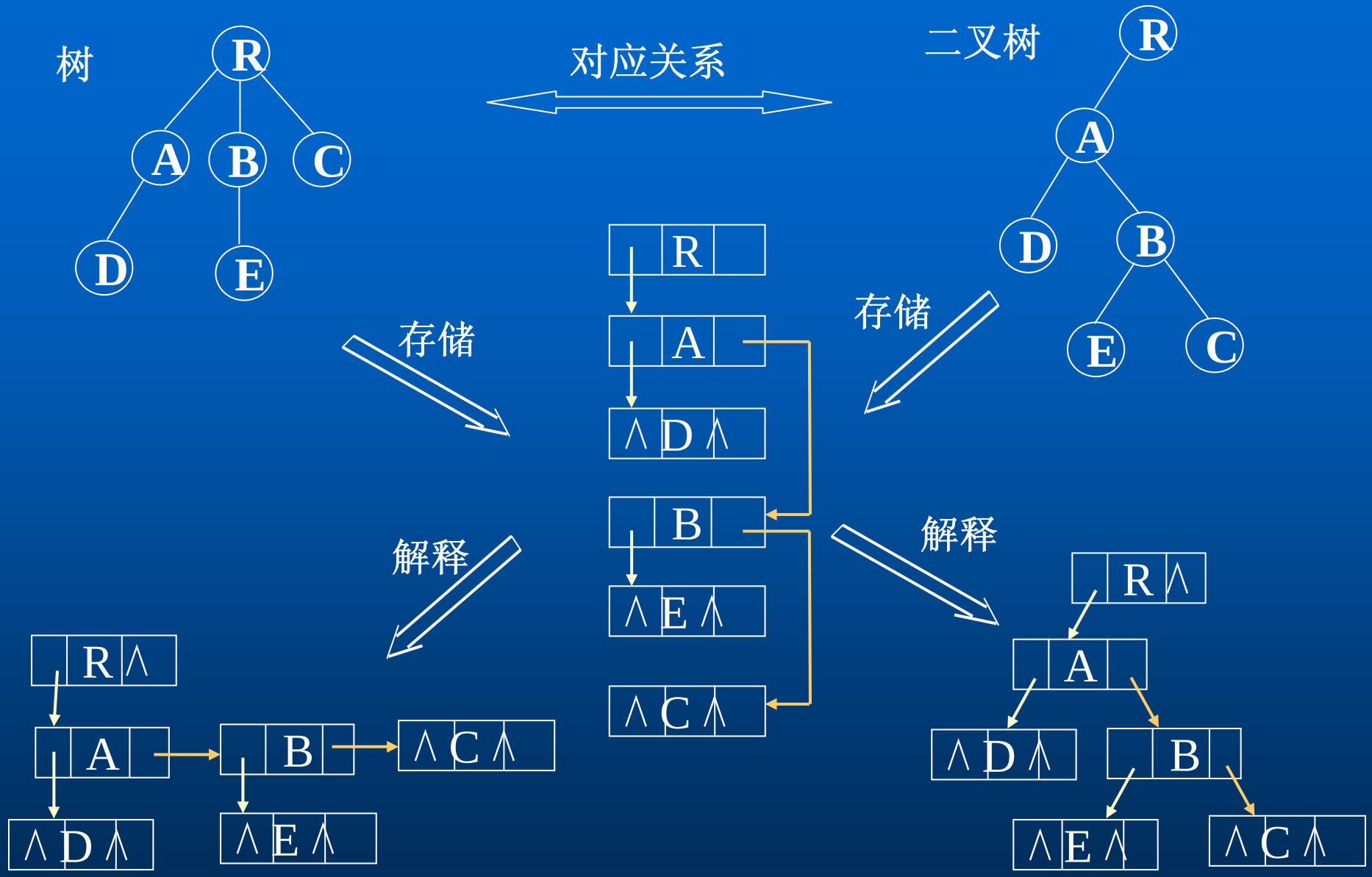


图6-18 树与二叉树的对应关系

1 树转换成二叉树

对于一般的树，可以方便地转换成一棵唯一的二叉树与之对应。将树转换成二叉树在“孩子兄弟表示法”中已给出，其详细步骤是：

(1) 加虚线。在树的每层按从“左至右”的顺序在兄弟结点之间加虚线相连。

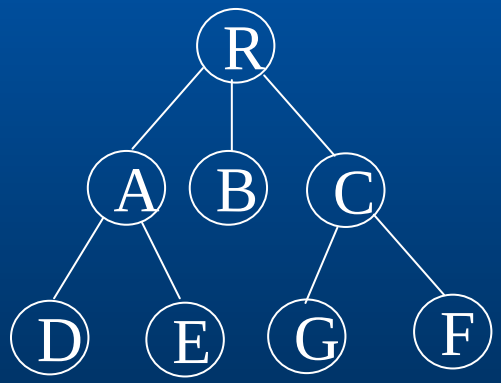
(2) 去连线。除最左的第一个子结点外，父结点与所有其它子结点的连线都去掉。

(3) 旋转。将树顺时针旋转45^0，原有的实线左斜。

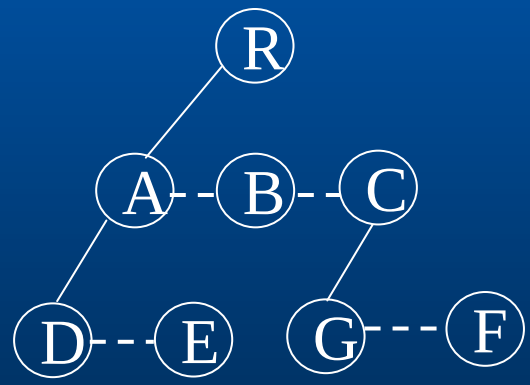
(4) 整型。将旋转后树中的所有虚线改为实线，并向右斜。该转换过程如图6-19所示。

这样转换后的二叉树的特点是：

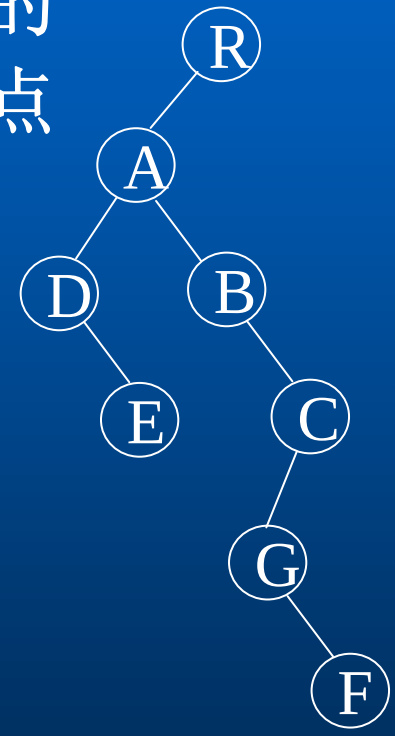
- 二叉树的根结点没有右子树，只有左子树；
- 左子结点仍然是原来树中相应结点的左子结点，而所有沿右链往下的右子结点均是原来树中该结点的兄弟结点。



(a) 一般的树



(b) 加虚线，去连线后



(C) 转换后的二叉树

图6-19 树向二叉树的转换过程

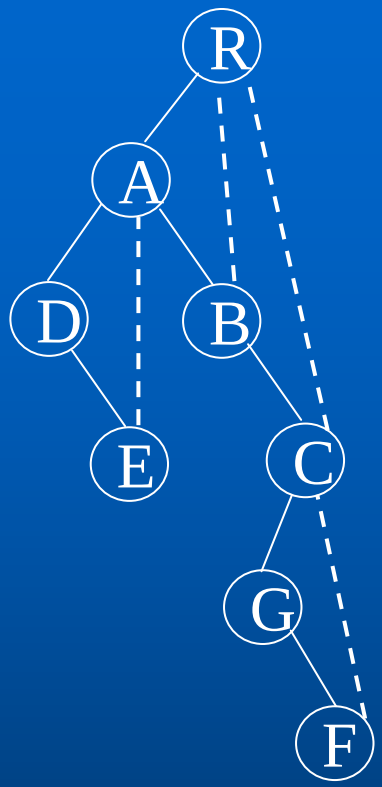
2 二叉树转换成树

对于一棵转换后的二叉树，如何还原成原来的树？其步骤是：

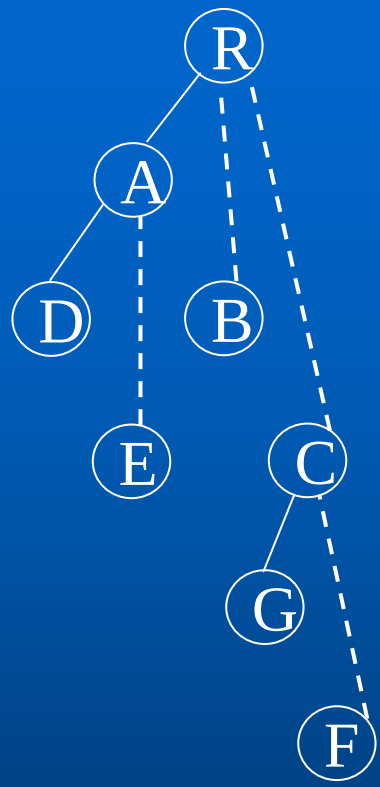
⑴ 加虚线。若某结点i是其父结点的左子树的根结点，则将该结点i的右子结点以及沿右子链不断地搜索所有的右子结点，将所有这些右子结点与i结点的父结点之间加虚线相连，如图6-20(a)所示。

⑵ 去连线。去掉二叉树中所有父结点与其右子结点之间的连线，如图6-20(b)所示。

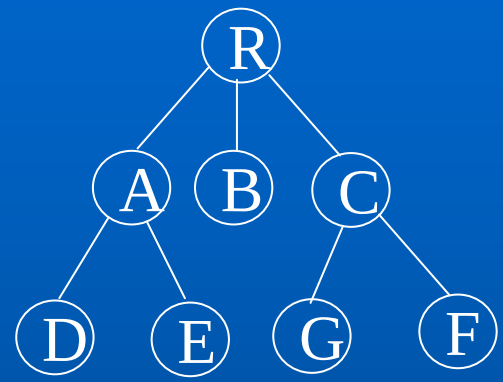
⑶ 规整化。将图中各结点按层次排列且将所有的虚线变成实线，如图6-20(c)所示。



(a) 加虚线后



(b) 去连线后



(C) 还原后的树

图6-20 二叉树向树的转换过程

3 森林转换成二叉树

当一般的树转换成二叉树后，二叉树的右子树必为空。若把森林中的第二棵树(转换成二叉树后)的根结点作为第一棵树(二叉树)的根结点的兄弟结点，则可导出森林转换成二叉树的转换算法如下：

设$F=\{T_1, T_2, \cdots, T_n\}$是森林，则按以下规则可转换成一棵二叉树B=(root，LB，RB)

① 若n=0，则B是空树。

② 若n>0，则二叉树B的根是森林T_1的根root(T_1)，B的左子树LB是B($T_{11}, T_{12}, \cdots, T_{1m}$)，其中$T_{11}, T_{12}, \cdots, T_{1m}$是$T_1$的子树(转换后)，而其右子树RB是从森林$F'=\{T_2, T_3, \cdots, T_n\}$转换而成的二叉树。

转换步骤：

① 将$F=\{T_1, T_2, \cdots, T_n\}$中的每棵树转换成二叉树。

② 按给出的森林中树的次序，从最后一棵二叉树开始，每棵二叉树作为前一棵二叉树的根结点的右子树，依次类推，则第一棵树的根结点就是转换后生成的二叉树的根结点，如图6-21所示。

(a) 森林　　(b) 森林中每棵树对应的二叉树　　(c) 森林对应的二叉树

图6-21 森林转换成二叉树的过程

4 二叉树转换成森林

若B=(root，LB，RB)是一棵二叉树，则可以将其转换成由若干棵树构成的森林：$F=\{T_1, T_2,\cdots,T_n\}$。

转换算法：

① 若B是空树，则F为空。

② 若B非空，则F中第一棵树T_1的根root(T_1)就是二叉树的根root，T_1中根结点的子森林F_1是由树B的左子树LB转换而成的森林；F中除T_1外其余树组成的的森林$F'=\{T_2, T_3,\cdots,T_n\}$ 是由B右子树RB转换得到的森林。

上述转换规则是递归的，可以写出其递归算法。以下给出具体的还原步骤。

① 去连线。将二叉树B的根结点与其右子结点以及沿右子结点链方向的所有右子结点的连线全部去掉，得到若干棵孤立的二叉树，每一棵就是原来森林F中的树依次对应的二叉树，如图6-22(b)所示。

② 二叉树的还原。将各棵孤立的二叉树按二叉树还原为树的方法还原成一般的树，如图6- 22(c)所示。

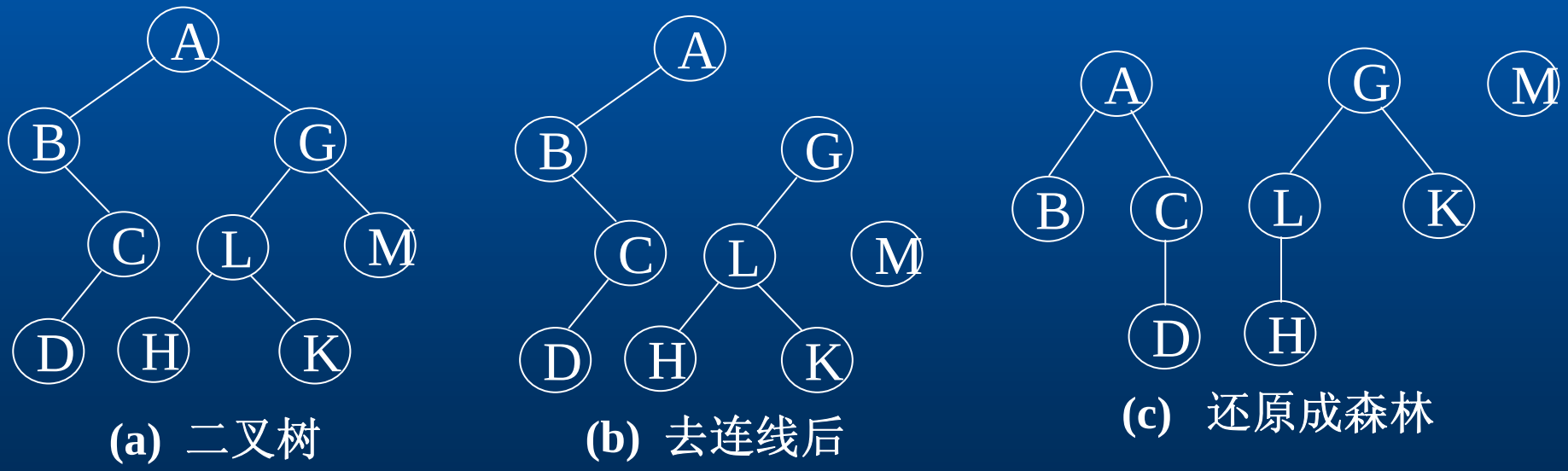


图6-22 二叉树还原成森林的过程

6.5.3 树和森林的遍历

1 树的遍历

由树结构的定义可知，树的遍历有二种方法。

(1) 先序遍历：先访问根结点，然后依次先序遍历完每棵子树。如图6-23的树，先序遍历的次序是：

ABCDEFGIJHK

(2) 后序遍历：先依次后序遍历完每棵子树，然后访问根结点。如图6-23的树，后序遍历的次序是：

CDBFGIJHEKA

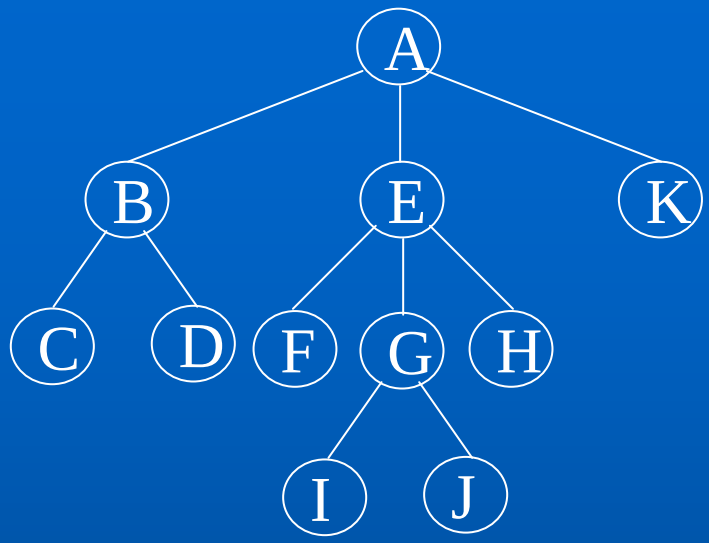


图6-23 树

说明:

- 树的先序遍历实质上与将树转换成二叉树后对二叉树的先序遍历相同。
- 树的后序遍历实质上与将树转换成二叉树后对二叉树的中序遍历相同。

2 森林的遍历

设$F=\{T_1, T_2, \cdots, T_n\}$是森林，对F的遍历有二种方法。

(1) 先序遍历：按先序遍历树的方式依次遍历F中的每棵树。

(2) 中序遍历：按后序遍历树的方式依次遍历F中的每棵树。

6.6 赫夫曼树及其应用

赫夫曼(Huffman)树又称最优树，是一类带权路径长度最短的树，有着广泛的应用。

6.6.1 最优二叉树(Huffman树)

1 基本概念

① 结点路径：从树中一个结点到另一个结点的之间的分支构成这两个结点之间的路径。

② 路径长度：结点路径上的分支数目称为路径长度。

③ 树的路径长度：从树根到每一个结点的路径长度之和。

例图6-23的树。A到F：结点路径 AEF；路径长度(即边的数目) 2；树的路径长度：

$3\times1+5\times2+2\times3=19$

④ 结点的带权路径长度：从该结点的到树的根结点之间的路径长度与结点的权(值)的乘积。

权(值)：各种开销、代价、频度等的抽象称呼。

⑤ 树的带权路径长度：树中所有叶子结点的带权路径长度之和，记做：

$$WPL = w_1 \times l_1 + w_2 \times l_2 + \cdots + w_n \times l_n = \sum w_i \times l_i \quad (i = 1,2,\cdots,n)$$

其中：n为叶子结点的个数；w_i为第i个结点的权值；l_i为第i个结点的路径长度。

⑥ Huffman树：具有n个叶子结点(每个结点的权值为w_i) 的二叉树不止一棵，但在所有的这些二叉树中，必定存在一棵WPL值最小的树，称这棵树为Huffman树(或称最优树)。

在许多判定问题时，利用Huffman树可以得到最佳判断算法。

如图6-24是权值分别为2、3、6、7，具有4个叶子结点的二叉树，它们的带权路径长度分别为：

(a) WPL=2×2+3×2+6×2+7×2=36；

(b) WPL=2×1+3×2+6×3+7×3=47；

(c) WPL=7×1+6×2+2×3+3×3=34。

其中(c)的 WPL值最小，可以证明是Huffman树。

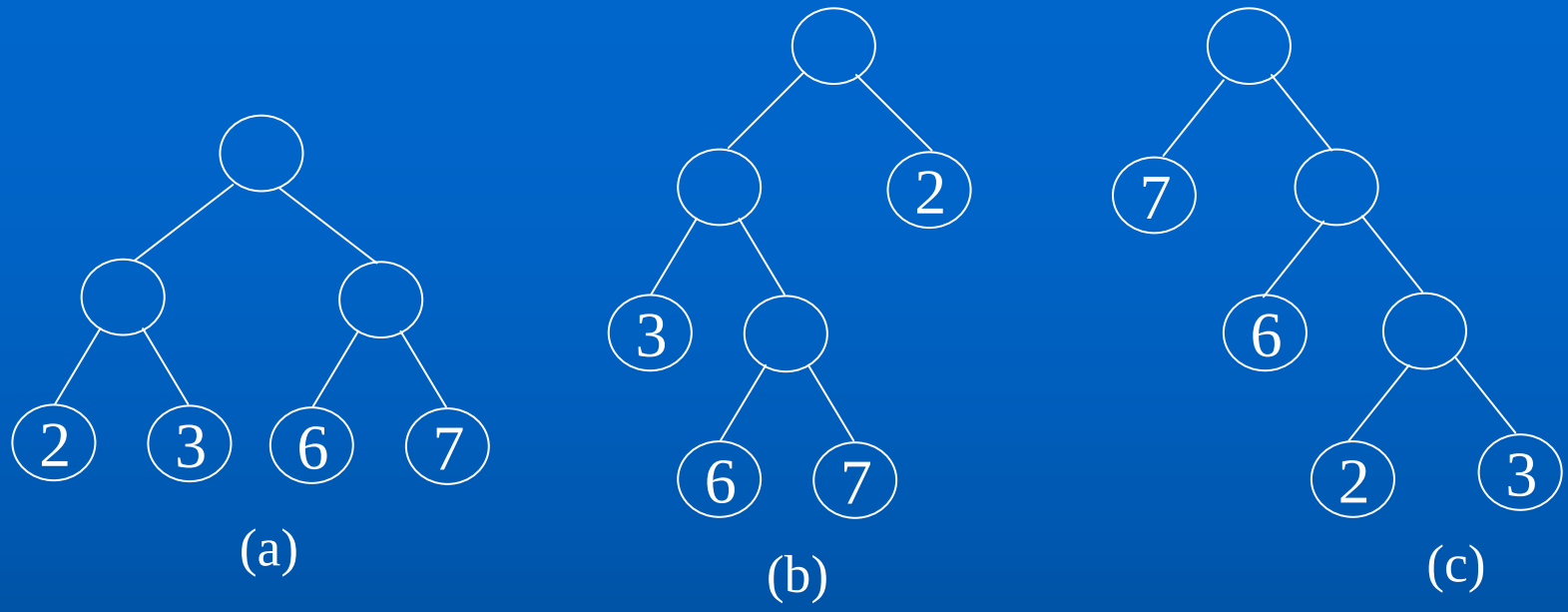


图6-24　具有相同叶子结点，不同带权路径长度的二叉树

2 Huffman树的构造

① 根据n个权值$\{w_1, w_2, \cdots, w_n\}$，构造成n棵二叉树的集合$F=\{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树只有一个权值为$w_i$的根结点，没有左、右子树；

② 在F中选取两棵根结点权值最小的树作为左、右子树构造一棵新的二叉树，且新的二叉树根结点权值为其左、右子树根结点的权值之和；

③ 在F中删除这两棵树，同时将新得到的树加入F中；

④ 重复②、③，直到F只含一颗树为止。

构造Huffman树时，为了规范，规定$F=\{T_1,T_2,\cdots,T_n\}$中权值小的二叉树作为新构造的二叉树的左子树，权值大的二叉树作为新构造的二叉树的右子树；在取值相等时，深度小的二叉树作为新构造的二叉树的左子树，深度大的二叉树作为新构造的二叉树的右子树。

图6-25是权值集合W={8, 3, 4, 6, 5, 5}构造Huffman树的过程。所构造的Huffman树的WPL是：

$$WPL=6\times2+3\times3+4\times3+8\times2+5\times3+5\times3=79$$

图6-25　Huffman树的构造过程

6.6.2 赫夫曼编码及其算法

1 Huffman编码

在电报收发等数据通讯中，常需要将传送的文字转换成由二进制字符0、1组成的字符串来传输。为了使收发的速度提高，就要求电文编码要尽可能地短。此外，要设计长短不等的编码，还必须保证任意字符的编码都不是另一个字符编码的前缀，这种编码称为前缀编码。

Huffman树可以用来构造编码长度不等且译码不产生二义性的编码。

设电文中的字符集$C=\{c_1,c_2,\cdots,c_i,\cdots,c_n\}$，各个字符出现的次数或频度集$W=\{w_1,w_2,\cdots,w_i,\cdots,w_n\}$

Huffman编码方法

以字符集C作为叶子结点，次数或频度集W作为结点的权值来构造 Huffman树。规定Huffman树中左分支代表“0”，右分支代表“1”。

从根结点到每个叶子结点所经历的路径分支上的“0”或“1”所组成的字符串，为该结点所对应的编码，称之为Huffman编码。

由于每个字符都是叶子结点，不可能出现在根结点到其它字符结点的路径上，所以一个字符的Huffman编码不可能是另一个字符的Huffman编码的前缀。

若字符集C={a, b, c, d, e, f}所对应的权值集合为W={8, 3, 4, 6, 5, 5}，如图6-25所示，则字符a,b, c,d, e,f所对应的Huffman编码分别是：10，010，011，00，110，111。

2 Huffman编码算法实现

(1) 数据结构设计

Huffman树中没有度为1的结点棵有n个叶子结点的Huffman树共有2n-1个结点，则可存储在大小为2n-1的一维数组中。实现编码的结点结构如图6-26所示。

原因：

- 求编码需从叶子结点出发走一条从叶子到根的路径。

Weight	Parent	Lchild	Rchild

Weight：权值域；　Parent：双亲结点下标

Lchild, Rchild：分别标识左、右子树的下标

图6-26　Huffman编码的结点结构

◆ 译码需从根结点出发走一条到叶子结点的路径。

结点类型定义：

```
#define  MAX_NODE  200     /*   Max_Node>2n-1    */
typedef struct
    {     unsigned int Weight ;    /*  权值域    */
          unsinged int Parent , Lchild , Rchild ;
    } HTNode ;
```

(2) Huffman树的生成

算法实现

```
void Create_Huffman(unsigned n, HTNode HT[ ], unsigned m)
    /* 创建一棵叶子结点数为n的Huffman树 */
  { unsigned int w ;  int k , j ;
    for (k=1 ; k<m ; k++)
      {  if (k<=n)
           { printf("\n Please Input Weight : w=?");
             scanf("%d", &w) ;HT[k].weight=w ;
           }      /* 输入时，所有叶子结点都有权值 */
         else HT[k].weight=0; /*  非叶子结点没有权值 */
```

```
        HT[k].Parent=HT[k].Lchild=HT[k].Rchild=0 ;
    }     /* 初始化向量HT */
for (k=n+1; k<m ; k++)
    {   unsigned w1=32767 , w2=w1 ;
            /*  w1 , w2分别保存权值最小的两个权值  */
        int  p1=0 , p2=0 ;
            /*  p1 , p2保存两个最小权值的下标  */
        for (j=1 ; j<=k-1 ; j++)
            {   if (HT[k].Parent==0)    /* 尚未合并 */
                    {   if (HT[j].Weight<w1)
                                {   w2=w1 ; p2=p1 ;
                                    w1=HT[j].Weight ; p1=j ;
                                }
```

```
                else if (HT[j].Weight<w2)
                        {  w2=HT[j].Weight ;  p2=j ;  }
              }   /*   找到权值最小的两个值及其下标  */
           }
        HT[k].Lchild=p1 ; HT[k].Rchild=p2 ;
        HT[k].weight=w1+w2 ;
        HT[p1].Parent=k ; HT[p2].Parent=k ;
     }
  }
```

说明：生成Huffman树后，树的根结点的下标是2n-1，即m-1。

(3) Huffman编码算法

根据出现频度(权值)Weight，对叶子结点的Huffman编码有两种方式：

① 从叶子结点到根逆向处理，求得每个叶子结点对应字符的Huffman编码。

② 从根结点开始遍历整棵二叉树，求得每个叶子结点对应字符的Huffman编码。

由Huffman树的生成知，n个叶子结点的树共有2n-1个结点，叶子结点存储在数组HT中的下标值为1∽n。

① 编码是叶子结点的编码，只需对数组HT[1…n]的n个权值进行编码；

② 每个字符的编码不同，但编码的最大长度是n。

求编码时先设一个通用的指向字符的指针变量，求得编码后再复制。

算法实现

```
void Huff_coding(unsigned n , Hnode HT[] , unsigned m)
    /*  m应为n+1,编码的最大长度n加1  */
  {  int  k , sp ,fp ;
     char *cd , *HC[m] ;
     cd=(char *)malloc(m*sizeof(char)) ;
        /*  动态分配求编码的工作空间  */
     cd[n]='\0'        /*  编码的结束标志  */
     for (k=1 ; k<n+1 ; k++)      /*  逐个求字符的编码    */
        {  sp=n ; p=k ; fp=HT[k].parent ;
```

```
        for ( ; fp!=0 ;p=fp , fp=HT[p].parent)
              /*  从叶子结点到根逆向求编码   */
            if  (HT[fp].parent==p)  cd[--sp]='0' ;
            else  cd[--sp]='1' ;
        HC[k]=(char *)malloc((n-sp)*sizeof(char)) ;
              /*   为第k个字符分配保存编码的空间 */
        trcpy(HC[k] , &cd[sp]) ;
    }
  free(cd) ;
}
```

习 题 六

⑴ 假设在树中， 结点x是结点y的双亲时，用(x,y)来表示树边。已知一棵树的树边集合为 { (e,i), (b,e), (b,d), (a,b), (g,j), (c,g), (c,f), (h,l), (c,h), (a,c) } ，用树型表示法表示该树，并回答下列问题：

① 哪个是根结点？哪些是叶子结点？哪个是g的双亲？哪些是g的祖先？哪些是g的孩子？那些是e的子孙？哪些是e的兄弟？哪些是f的兄弟？

② b和n的层次各是多少？树的深度是多少？以结点c为根的子树的深度是多少？

(2) 一棵深度为h的满k叉树有如下性质： 第h层上的结点都是叶子结点，其余各层上每个结点都有k棵非空子树。 如果按层次顺序(同层自左至右)从1开始对全部结点编号，问：

① 各层的结点数是多少?

② 编号为i的结点的双亲结点(若存在)的编号是多少?

③ 编号为i的结点的第j个孩子结点(若存在)的编号是多少?

④ 编号为i的结点的有右兄弟的条件是什么? 其右兄弟的编号是多少?

(3) 设有如图6-27所示的二叉树。

① 分别用顺序存储方法和链接存储方法画出该二叉树的存储结构。

② 写出该二叉树的先序、中序、后序遍历序列。

(4) 已知一棵二叉树的先序遍历序列和中序遍历序列分别为ABDGHCEFI和GDHBAECIF，请画出这棵二叉树，然后给出该树的后序遍历序列。

(5) 设一棵二叉树的中序遍历序列和后序遍历序列分别为BDCEAFHG和DECBHGFA，请画出这棵二叉树，然后给出该树的先序序列。

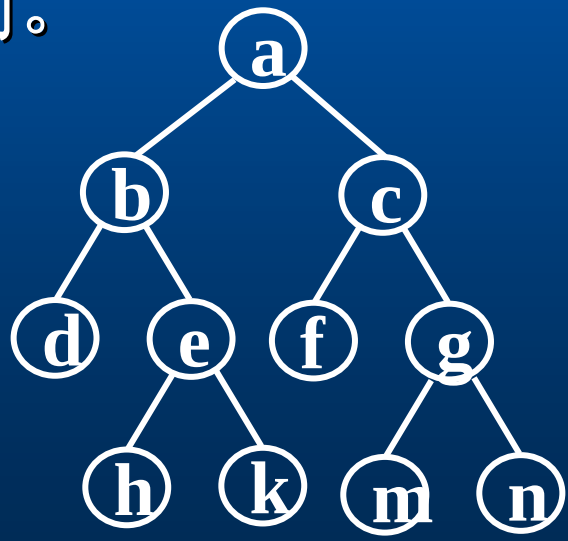


图6-27 二叉树

(6) 已知一棵二叉树的中序遍历序列和后序遍历序列分别为dgbaekchif和gdbkeihfca，请画出这棵二叉树对应的中序线索树和后序线索树。

(7) 以二叉链表为存储结构，请分别写出求二叉树的结点总数及叶子结点总数的算法。

(8) 设图6-27所示的二叉树是森林F所对应的二叉树，请画出森林F。

(9) 设有一棵树，如图6-28所示。

① 请分别用双亲表示法、孩子表示法、孩子兄弟表示法给出该树的存储结构。

② 请给出该树的先序遍历序列和后序遍历序列。

③ 请将这棵树转换成二叉树。

⑽ 设给定权值集合w={3,5,7,8,11,12}，请构造关于w的一棵huffman树，并求其加权路径长度WPL。

⑾ 假设用于通信的电文是由字符集{a, b, c, d, e, f, g, h}中的字符构成，这8个字符在电文中出现的概率分别为{0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10}。

① 请画出对应的huffman树(按左子树根结点的权小于等于右子树根结点的权的次序构造)。

② 求出每个字符的huffman编码。

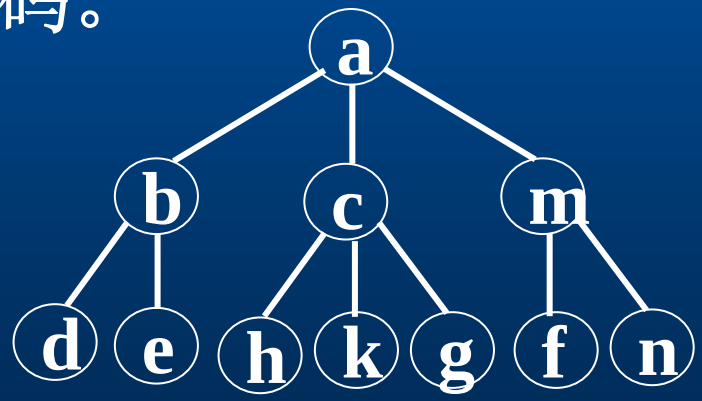


图6-28 一般的树

第7章　图

图(Graph)是一种比线性表和树更为复杂的数据结构。

线性结构：是研究数据元素之间的一对一关系。在这种结构中，除第一个和最后一个元素外，任何一个元素都有唯一的一个直接前驱和直接后继。

树结构：是研究数据元素之间的一对多的关系。在这种结构中，每个元素对下(层)可以有0个或多个元素相联系，对上(层)只有唯一的一个元素相关，数据元素之间有明显的层次关系。

图结构：是研究数据元素之间的多对多的关系。在这种结构中，任意两个元素之间可能存在关系。即结点之间的关系可以是任意的，图中任意元素之间都可能相关。

图的应用极为广泛，已渗入到诸如语言学、逻辑学、物理、化学、电讯、计算机科学以及数学的其它分支。

7.1 图的基本概念

7.1.1 图的定义和术语

一个图(G)定义为一个偶对(V,E)，记为G=(V,E)。其中：V是顶点(Vertex)的非空有限集合，记为V(G)；E是无序集V&V的一个子集，记为E(G)，其元素是图的弧(Arc)。

将顶点集合为空的图称为空图。其形式化定义为：

G=(V，E)

V={v|v∈data object}

E={<v,w>| v,w∈V∧p(v,w)}

P(v,w)表示从顶点v到顶点w有一条直接通路

弧(Arc)：表示两个顶点v和w之间存在一个关系，用顶点偶对<v,w>表示。通常根据图的顶点偶对将图分为有向图和无向图。

有向图(Digraph)：若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是有序的，称图G是有向图。

在有向图中，若 <v,w>∈E(G)，表示从顶点v到顶点w有一条弧。其中：v称为弧尾(tail)或始点(initial node)，w称为弧头(head)或终点(terminal node)。

无向图(Undigraph)：若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是无序的，称图G是无向图。

在无向图中，若∀<v,w>∈E(G)，有<w,v>∈E(G)，即E(G)是对称，则用无序对(v,w)表示v和w之间的一条边(Edge)，因此(v,w)和(w,v)代表的是同一条边。

例1：设有有向图G1和无向图G2，形式化定义分别是：

G1=(V1，E1)

V1={a,b,c,d,e}

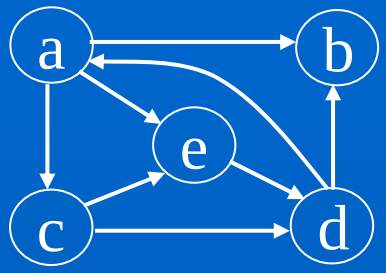
E1={<a,b>,<a,c>,<a,e>,<c,d>,<c,e>,<d,a>,<d,b>,<e,d>}

G2=(V2，E2)

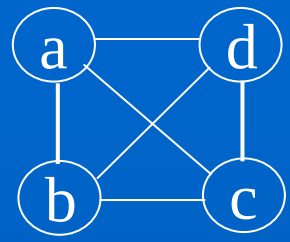
V2={a,b,c,d}

E2={(a,b),(a,c),(a,d),(b,d),(b,c),(c,d)}

它们所对应的图如图7-1所示。



(a) 有向图G1



(b) 无向图G2

图7-1 图的示例

完全无向图：对于无向图，若图中顶点数为**n**，用**e**表示边的数目，则**e** ∈**[0，n(n-1)/2]** 。具有**n(n-1)/2**条边的无向图称为完全无向图。

完全无向图另外的定义是：

对于无向图**G=(V，E)**，若∀v_i，v_j ∈**V**，当$v_i \neq v_j$时，有$(v_i, v_j) \in E$，即图中任意两个不同的顶点间都有一条无向边，这样的无向图称为完全无向图。

完全有向图：对于有向图，若图中顶点数为n，用e表示弧的数目，则$e \in [0, n(n-1)]$。具有$n(n-1)$条边的有向图称为完全有向图。

完全有向图另外的定义是：

对于有向图$G=(V, E)$，若$\forall v_i, v_j \in V$，当$v_i \neq v_j$时，有$<v_i, v_j> \in E \wedge <v_j, v_i> \in E$，即图中任意两个不同的顶点间都有一条弧，这样的有向图称为完全有向图。

有很少边或弧的图（$e < n^{\log n}$）的图称为稀疏图，反之称为稠密图。

权(Weight)：与图的边和弧相关的数。权可以表示从一个顶点到另一个顶点的距离或耗费。

子图和生成子图：设有图G=(V，E)和G'=(V'，E')，若V'⊂V且E'⊂E，则称图G'是G的子图；若V'=V且E'⊂E，则称图G'是G的一个生成子图。

顶点的邻接(Adjacent)：对于无向图G=(V，E)，若边(v,w)∈E，则称顶点v和w 互为邻接点，即v和w相邻接。边(v,w)依附(incident)与顶点v和w 。

对于有向图G=(V ，E)，若有向弧<v,w>∈E，则称顶点v "邻接到"顶点w，顶点w "邻接自"顶点v，弧<v,w> 与顶点v和w "相关联"。

顶点的度、入度、出度：对于无向图G=(V，E)，∀v_i∈V，图G中依附于v_i的边的数目称为顶点v_i的度(degree)，记为TD(v_i)。

显然，在无向图中，所有顶点度的和是图中边的2倍。 即 $\sum TD(v_i)=2e$ $i=1, 2, \ldots, n$ ， e为图的边数。

对有向图G=(V，E)，若$\forall v_i \in V$ ，图G中以v_i作为起点的有向边(弧)的数目称为顶点v_i的出度(Outdegree)，记为$OD(v_i)$ ；以v_i作为终点的有向边(弧)的数目称为顶点v_i的入度(Indegree)，记为$ID(v_i)$ 。顶点v_i的出度与入度之和称为v_i的度，记为$TD(v_i)$ 。即

$$TD(v_i)=OD(v_i)+ID(v_i)$$

路径(Path)、路径长度、回路(Cycle)：对无向图G=(V，E)，若从顶点v_i经过若干条边能到达v_j，称顶点v_i和v_j是连通的，又称顶点v_i到v_j有路径

或路径是图G中连接两顶点之间所经过的顶点序列。即

$Path=v_{i0}v_{i1}\ldots v_{im}$， $v_{ij}\in V$且$(v_{ij-1}, v_{ij})\in E$
$j=1,2,\ldots,m$

或

$Path=v_{i0}v_{i1}\ldots v_{im}$， $v_{ij}\in V$且$<v_{ij-1}, v_{ij}>\in E$
$j=1,2,\ldots,m$

路径上边或有向边(弧)的数目称为该路径的长度。

在一条路径中，若没有重复相同的顶点，该路径称为简单路径；第一个顶点和最后一个顶点相同的路径称为回路(环)；在一个回路中，若除第一个与最后一个顶点外，其余顶点不重复出现的回路称为简单回路(简单环)。

连通图、图的连通分量：对无向图G=(V，E)，若$\forall v_i$，$v_j \in V$，v_i和v_j都是连通的，则称图G是**连通图**，否则称为**非连通图**。若G是非连通图，则**极大的连通子图**称为G的**连通分量**。

对有向图G=(V，E)，若$\forall v_i$，$v_j \in V$，都有**以v_i为起点，v_j为终点**以及以v_j为起点，v_i为终点的有向路径，称图G是**强连通图**，否则称为**非强连通图**。若G是非强连通图，则**极大的强连通子图**称为G的**强连通分量**。

“**极大**”的含义：指的是对子图再增加图G中的其它顶点，子图就不再连通。

生成树、生成森林：一个连通图(无向图)的生成树是一个极小连通子图，它含有图中全部n个顶点和只有足以构成一棵树的n-1条边，称为图的生成树，如图7-2所示。

关于无向图的生成树的几个结论：

- 一棵有n个顶点的生成树有且仅有n-1条边；
- 如果一个图有n个顶点和小于n-1条边，则是非连通图；
- 如果多于n-1条边，则一定有环；
- 有n-1条边的图不一定是生成树。

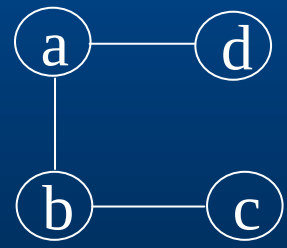
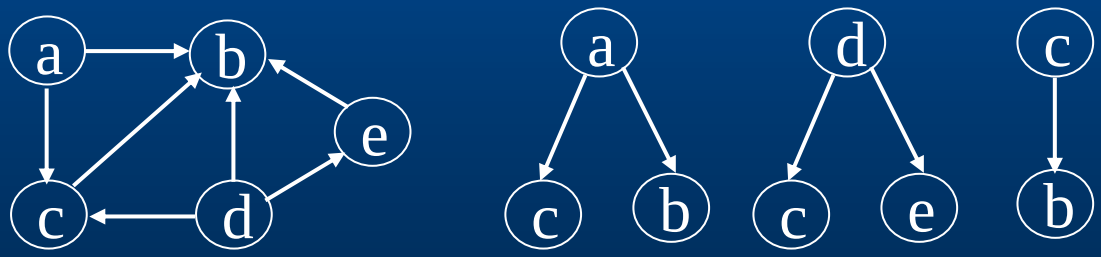


图7-2 图G2的一棵生成树

有向图的生成森林是这样一个子图，由若干棵有向树组成，含有图中全部顶点。

有向树是只有一个顶点的入度为0，其余顶点的入度均为1的有向图，如图7-3所示。

网：每个边(或弧)都附加一个权值的图，称为带权图。带权的连通图(包括弱连通的有向图)称为网或网络。网络是工程上常用的一个概念，用来表示一个工程或某种流程，如图7-4所示。



(a) 有向图　　(b) 生成森林

图7-3 有向图及其生成森林

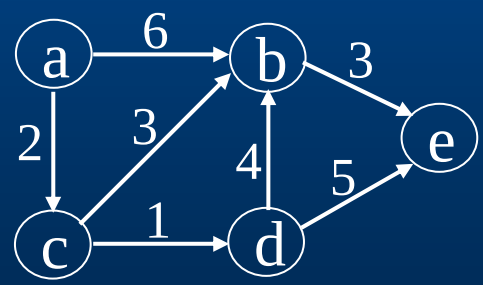


图7-4 带权有向图

7.1.2 图的抽象数据类型定义

图是一种数据结构，加上一组基本操作就构成了图的抽象数据类型。

图的抽象数据类型定义如下：

ADT Graph{

数据对象**V**：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,w∈V∧p(v,w)，**<v,w>**表示从**v**到**w**的弧，**P(v,w)**定义了弧**<v,w>**的信息 **}**

基本操作P：

Create_Graph()： 图的创建操作。

初始条件：无。

操作结果：生成一个没有顶点的空图G。

GetVex(G, v)： 求图中的顶点v的值。

初始条件：图G存在，v是图中的一个顶点。

操作结果：生成一个没有顶点的空图G。

… …

DFStraver(G,V)：从v出发对图G深度优先遍历。

初始条件：图G存在。

操作结果：对图G深度优先遍历，每个顶点访问且只访问一次。

……

BFStraver(G,V)：从**v**出发对图**G**广度优先遍历。

初始条件：图**G**存在。

操作结果：对图**G**广度优先遍历，每个顶点访问且只访问一次。

} ADT Graph

详见**p**$_{156\sim157}$。

7.2 图的存储结构

图的存储结构比较复杂，其复杂性主要表现在：

- 任意顶点之间可能存在联系，无法以数据元素在存储区中的物理位置来表示元素之间的关系。
- 图中顶点的度不一样，有的可能相差很大，若按度数最大的顶点设计结构，则会浪费很多存储单元，反之按每个顶点自己的度设计不同的结构，又会影响操作。

图的常用的存储结构有：邻接矩阵、邻接链表、十字链表、邻接多重表和边表。

7.2.1 邻接矩阵(数组)表示法

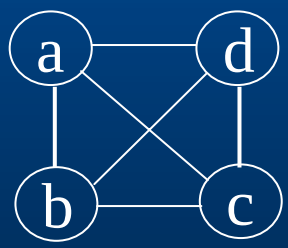
基本思想：对于有n个顶点的图，用一维数组vexs[n]存储顶点信息，用二维数组A[n][n]存储顶点之间关系的信息。该二维数组称为邻接矩阵。在邻接矩阵中，以顶点在vexs数组中的下标代表顶点，邻接矩阵中的元素A[i][j]存放的是顶点i到顶点j之间关系的信息。

1 无向图的数组表示

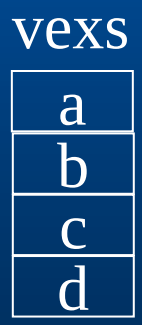
(1) 无权图的邻接矩阵

无向无权图G=(V，E)有n(n≧1)个顶点，其邻接矩阵是n阶对称方阵，如图7-5所示。其元素的定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}(v_i, v_j)\in E\text{，即}v_i, v_j\text{邻接}\\ 0 & \text{若}(v_i, v_j)\notin E\text{，即}v_i, v_j\text{不邻接}\end{cases}$$



(a) 无向图



(b) 顶点矩阵

$$\begin{pmatrix}0 & 1 & 1 & 1\\ 1 & 0 & 1 & 1\\ 1 & 1 & 0 & 1\\ 1 & 1 & 1 & 0\end{pmatrix}$$

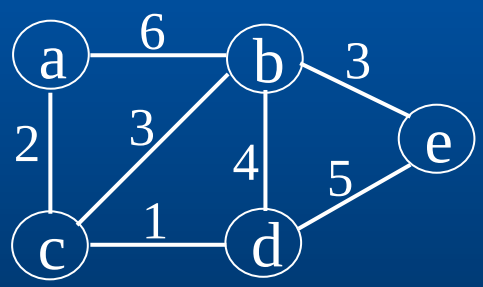
(c) 邻接矩阵

图7-5 无向无权图的数组存储

(2) 带权图的邻接矩阵

无向带权图G=(V，E) 的邻接矩阵如图7-6所示。其元素的定义如下：

$$A[i][j]=\begin{cases} W_{ij} & \text{若}(v_i, v_j)\in E\text{，即}v_i, v_j\text{邻接，权值为}w_{ij} \\ \infty & \text{若}(v_i, v_j)\notin E\text{，即}v_i, v_j\text{不邻接时} \end{cases}$$



(a) 带权无向图

vexs
a
b
c
d
e

(b) 顶点矩阵

$$\begin{pmatrix} \infty & 6 & 2 & \infty & \infty \\ 6 & \infty & 3 & 4 & 3 \\ 2 & 3 & \infty & 1 & \infty \\ \infty & 4 & 3 & \infty & 5 \\ \infty & 3 & \infty & 5 & \infty \end{pmatrix}$$

(c) 邻接矩阵

图7-6 无向带权图的数组存储

(3) 无向图邻接矩阵的特性

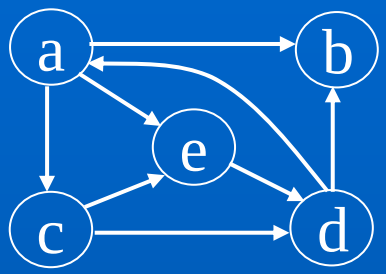
- 邻接矩阵是对称方阵；
- 对于顶点v_i，其度数是第i行的非0元素的个数；
- 无向图的边数是上(或下)三角形矩阵中非0元素个数。

2 有向图的数组表示

(1) 无权图的邻接矩阵

若有向无权图G=(V，E)有n(n≧1)个顶点，则其邻接矩阵是n阶对称方阵，如图7-7所示。元素定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}<v_i, v_j>\in E\text{，从}v_i\text{到}v_j\text{有弧}\\ 0 & \text{若}<v_i, v_j>\notin E\text{ 从}v_i\text{到}v_j\text{ 没有弧}\end{cases}$$



(a) 有向图

vexs

a
b
c
d
e

(b) 顶点矩阵

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

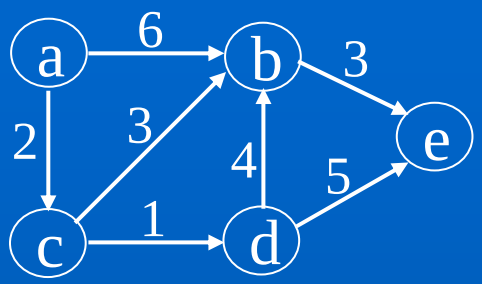
(c) 邻接矩阵

图7-7 有向无权图的数组存储

(2) 带权图的邻接矩阵

有向带权图G=(V，E)的邻接矩阵如图7-8所示。其元素的定义如下：

$$A[i][j]=\begin{cases} w_{ij} & \text{若} <v_i,v_j> \in E，\text{即} v_i，v_j \text{邻接，权值为} w_{ij} \\ \infty & \text{若} <v_i,v_j> \notin E，\text{即} v_i，v_j \text{不邻接时} \end{cases}$$



(a) 带权有向图

vexs

a
b
c
d
e

(b) 顶点矩阵

$$\begin{pmatrix} \infty & 6 & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 \\ \infty & 3 & \infty & 1 & \infty \\ \infty & 4 & \infty & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

(c) 邻接矩阵

图7-8 带权有向图的数组存储

(3) 有向图邻接矩阵的特性

- 对于顶点v_i，第i行的非0元素的个数是其出度$OD(v_i)$；第i列的非0元素的个数是其入度$ID(v_i)$。
- 邻接矩阵中非0元素的个数就是图的弧的数目。

3 图的邻接矩阵的操作

图的邻接矩阵的实现比较容易，定义两个数组分别存储顶点信息(数据元素)和边或弧的信息(数据元素之间的关系)。其存储结构形式定义如下：

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
   /* 根据图的权值类型，分别定义为最大整数或实数 */
#define MAX_VEX  30     /*  最大顶点数目 */
typedef enum {DG, AG, WDG,WAG} GraphKind ;
     /*  {有向图，无向图，带权有向图，带权无向图}  */
```

```
typedef struct ArcType
   {  VexType  vex1, vex2 ;   /*  弧或边所依附的两个顶点 */
      ArcValType  ArcVal ;    /*  弧或边的权值  */
      ArcInfoType  ArcInfo ;  /*  弧或边的其它信息   */
   }ArcType ;   /*  弧或边的结构定义  */
typedef struct
   {  GraphKind  kind ;   /*  图的种类标志   */
      int  vexnum , arcnum ;   /*  图的当前顶点数和弧数  */
      VexType   vexs[MAX_VEX] ;  /*  顶点向量  */
      AdjType   adj[MAX_VEX][MAX_VEX];
      }MGraph ;    /*  图的结构定义   */
```

利用上述定义的数据结构，可以方便地实现图的各种操作。

(1) 图的创建

```
AdjGraph  *Create_Graph(MGraph * G)
  {   printf(“请输入图的种类标志：”) ;
      scanf(“%d”, &G->kind) ;
      G->vexnum=0 ;       /*  初始化顶点个数  */
      return(G) ;
  }
```

(2) 图的顶点定位

图的顶点定位操作实际上是确定一个顶点在vexs数组中的位置(下标)，其过程完全等同于在顺序存储的线性表中查找一个数据元素。

算法实现：

```
int  LocateVex(MGraph *G , VexType *vp)
  {  int  k ;
     for (k=0 ; k<G->vexnum ; k++)
         if (G->vexs[k]==*vp)  return(k) ;
     return(-1) ;     /*  图中无此顶点  */
  }
```

(3) 向图中增加顶点

向图中增加一个顶点的操作，类似在顺序存储的线性表的末尾增加一个数据元素。

算法实现：

```
int  AddVertex(MGraph *G , VexType *vp)
   {  int  k , j ;
      if  (G->vexnum>=MAX_VEX)
         {  printf("Vertex Overflow !\n") ; return(-1) ;  }
      if  (LocateVex(G , vp)!=-1)
         {  printf("Vertex has existed !\n") ; return(-1) ; }
      k=G->vexnum ; G->vexs[G->vexnum++]=*vp ;
```

```
    if (G->kind==DG||G->kind==AG)
        for (j=0 ; j<G->vexnum ; j++)
            G->adj[j][k].ArcVal=G->adj[k][j].ArcVal=0 ;
                /*  是不带权的有向图或无向图  */
    else
        for (j=0 ; j<G->vexnum ; j++)
            {   G->adj[j][k].ArcVal=INFINITY ;
                G->adj[k][j].ArcVal=INFINITY ;
                    /*  是带权的有向图或无向图  */
            }
    return(k) ;
}
```

(4) 向图中增加一条弧

根据给定的弧或边所依附的顶点，修改邻接矩阵中所对应的数组元素。

算法实现：

```
int  AddArc(MGraph *G , ArcType *arc)
   {   int  k , j ;
       k=LocateVex(G , &arc->vex1)  ;
       j=LocateVex(G , &arc->vex1)  ;
       if (k==-1||j==-1)
          {  printf("Arc's Vertex do not existed !\n") ;
             return(-1) ;
          }
```

```
    if (G->kind==DG||G->kind==WDG)
       {  G->adj[k][j].ArcVal=arc->ArcVal;
          G->adj[k][j].ArcInfo=arc->ArcInfo ;
                /*  是有向图或带权的有向图*/
       }
    else
       {  G->adj[k][j].ArcVal=arc->ArcVal ;
          G->adj[j][k].ArcVal=arc->ArcVal ;
          G->adj[k][j].ArcInfo=arc->ArcInfo ;
          G->adj[j][k].ArcInfo=arc->ArcInfo ;
              /*  是无向图或带权的无向图,需对称赋值  */
       }
    return(1) ;
}
```

7.2.2 邻接链表法

基本思想：对图的每个顶点建立一个单链表，存储该顶点所有邻接顶点及其相关信息。每一个单链表设一个表头结点。

第i个单链表表示依附于顶点V_i的边(对有向图是以顶点V_i为头或尾的弧)。

1 结点结构与邻接链表示例

链表中的结点称为表结点，每个结点由三个域组成，如图7-9(a)所示。其中邻接点域(adjvex)指示与顶点V_i邻接的顶点在图中的位置(顶点编号)，链域(nextarc)指向下一个与顶点V_i邻接的表结点，数据域(info)存储和边或弧相关的信息，如权值等。对于无权图，如果没有与边相关的其他信息，可省略此域。

每个链表设一个表头结点(称为顶点结点)，由两个域组成，如图7-9(b)所示。链域(firstarc)指向链表中的第一个结点，数据域(data) 存储顶点名或其他信息。

表结点：

adjvex	info	nextarc

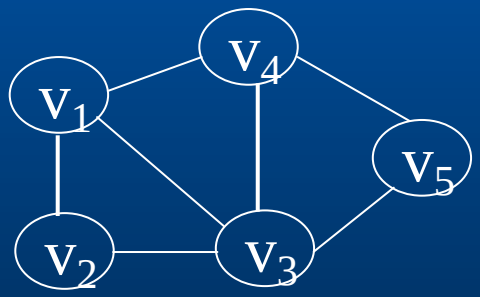
顶点结点：

data	firstarc

图7-9 邻接链表结点结构

在图的邻接链表表示中，所有顶点结点用一个向量 以顺序结构形式存储，可以随机访问任意顶点的链表，该向量称为表头向量，向量的下标指示顶点的序号。

用邻接链表存储图时，对无向图，其邻接链表是唯一的，如图7-10所示；对有向图，其邻接链表有两种形式，如图7-11所示。



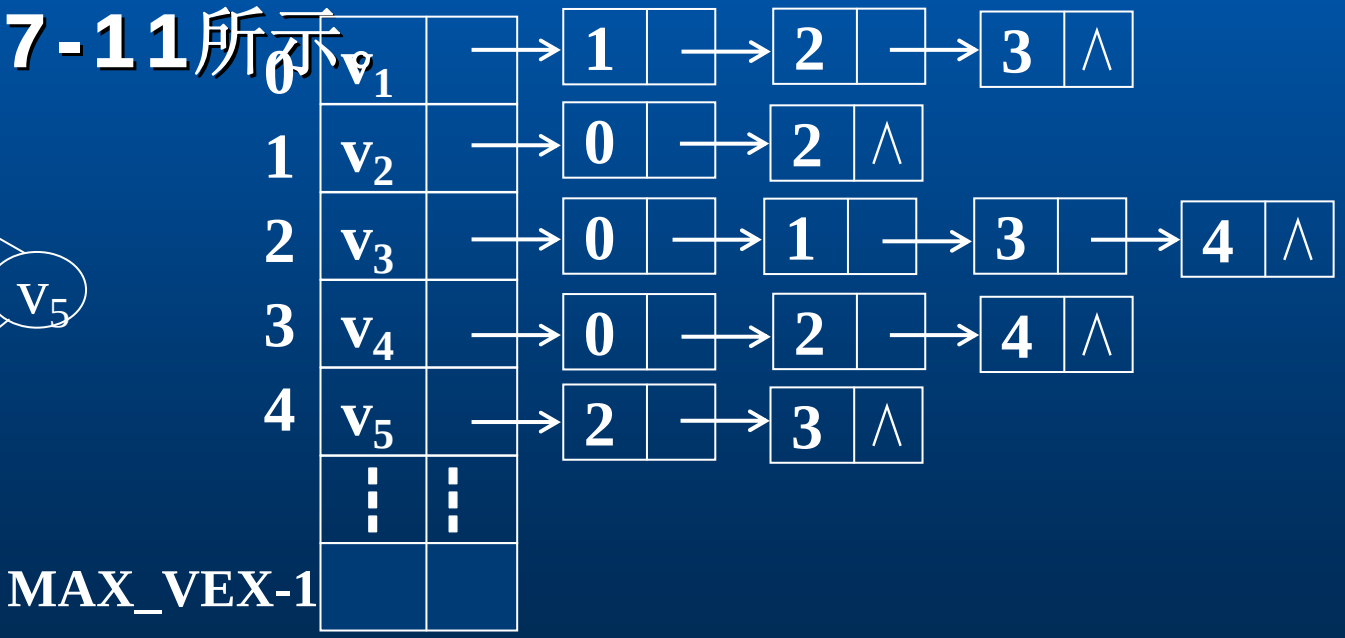
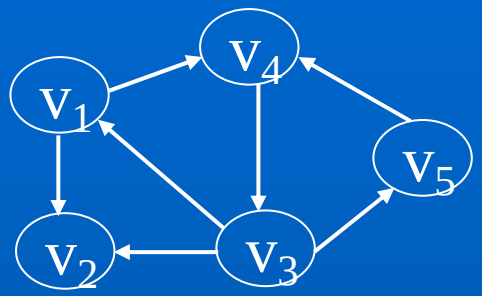
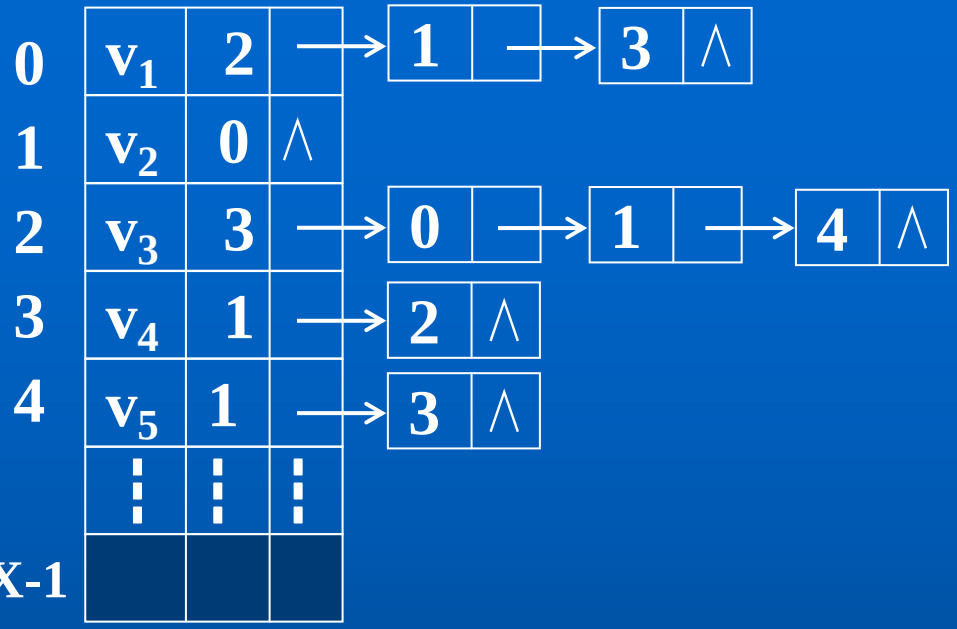


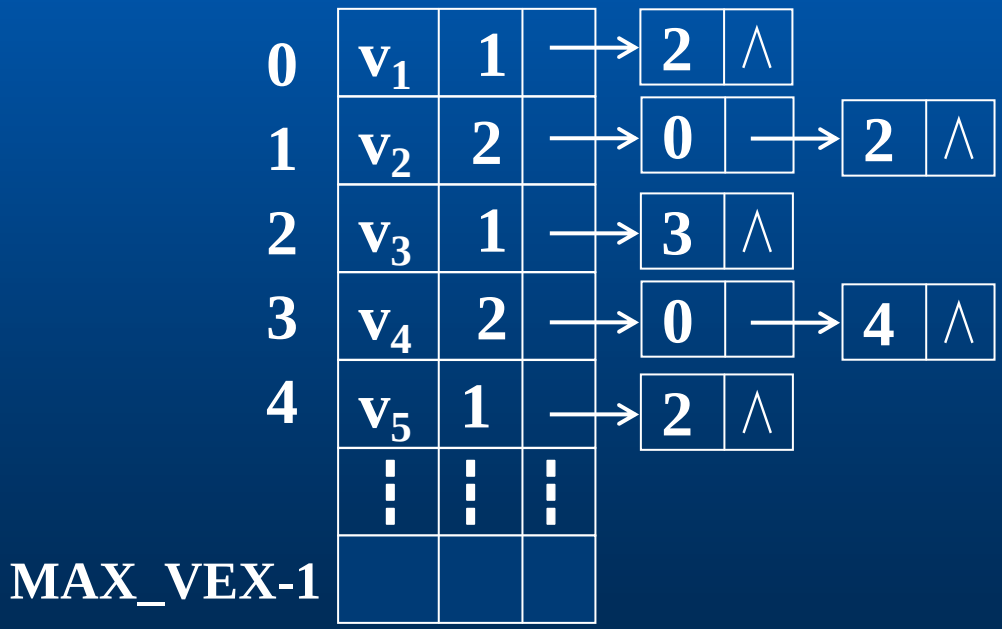
图7-10 无向图及其邻接链表



(a) 有向图



(b) 正邻接链表，出度直观



(c) 逆邻接链表，入度直观

图7-11 有向图及其邻接链表

2 邻接表法的特点

- 表头向量中每个分量就是一个单链表的头结点，分量个数就是图中的顶点数目；
- 在边或弧稀疏的条件下，用邻接表表示比用邻接矩阵表示节省存储空间；
- 在无向图，顶点V_i的度是第i个链表的结点数；
- 对有向图可以建立正邻接表或逆邻接表。正邻接表是以顶点V_i为出度(即为弧的起点)而建立的邻接表；逆邻接表是以顶点V_i为入度(即为弧的终点)而建立的邻接表；
- 在有向图中，第i个链表中的结点数是顶点V_i的出(或入)度；求入(或出)度，须遍历整个邻接表；

◆ 在邻接表上容易找出任一顶点的第一个邻接点和下一个邻接点；

3 结点及其类型定义

```
#define MAX_VEX  30      /*  最大顶点数  */
typedef int  InfoType;
typedef enum {DG, AG, WDG,WAG}
GraphKind ;
typedef struct LinkNode
   {  int  adjvex ;       // 邻接点在头结点数组中的位置(下标)
      InfoType   info ;       // 与边或弧相关的信息, 如权值
      struct LinkNode  *nextarc ;     // 指向下一个表结点
   }LinkNode ;    /*  表结点类型定义  */
```

```
typedef struct VexNode
  { VexType data;     // 顶点信息
    int indegree;   // 顶点的度,有向图是入度或出度或没有
    LinkNode *firstarc;    // 指向第一个表结点
  }VexNode;     /* 顶点结点类型定义 */
typedef struct ArcType
  { VexType vex1,vex2;    /* 弧或边所依附的两个顶点 */
    InfoType info;       // 与边或弧相关的信息,如权值
  }ArcType;     /* 弧或边的结构定义 */
```

```
typedef struct
  {  GraphKind  kind ;       /*  图的种类标志   */
    int vexnum ;
    VexNode   AdjList[MAX_VEX] ;
  }ALGraph ;     /*  图的结构定义   */
```

利用上述的存储结构描述，可方便地实现图的基本操作。

(1) 图的创建

```
ALGraph *Create_Graph(ALGraph * G)
  {   printf("请输入图的种类标志：");
    scanf("%d", &G->kind);
    G->vexnum=0;        /* 初始化顶点个数 */
    return(G);
  }
```

(2) 图的顶点定位

图的顶点定位实际上是确定一个顶点在AdjList数组中的某个元素的data域内容。

算法实现：

```
int  LocateVex(ALGraph *G , VexType *vp)
   {  int  k ;
      for (k=0 ; k<G->vexnum ; k++)
         if (G->AdjList[k].data==*vp)  return(k) ;
      return(-1) ;     /*  图中无此顶点  */
   }
```

(3) 向图中增加顶点

向图中增加一个顶点的操作，在AdjList数组的末尾增加一个数据元素。

算法实现：

```
int  AddVertex(ALGraph *G , VexType *vp)
   {  int  k , j ;
      if  (G->vexnum>=MAX_VEX)
         {  printf("Vertex Overflow !\n") ;  return(-1) ; }
      if  (LocateVex(G , vp)!=-1)
         {  printf("Vertex has existed !\n") ; return(-1) ; }
      G->AdjList[G->vexnum].data=*vp ;
```

```
    G->AdjList[G->vexnum].degree=0 ;
    G->AdjList[G->vexnum].firstarc=NULL ;
    k=++G->vexnum ;
    return(k) ;
  }
```

(4) 向图中增加一条弧

根据给定的弧或边所依附的顶点，修改单链表：无向图修改两个单链表；有向图修改一个单链表。

算法实现：

```
int  AddArc(ALGraph *G , ArcType *arc)
  {  int  k , j ;
     LinkNode *p ,*q ;
```

```
k=LocateVex(G , &arc->vex1) ;
j=LocateVex(G , &arc->vex2) ;
if (k==-1||j==-1)
   {  printf("Arc's Vertex do not existed !\n") ;
      return(-1) ;
   }
p=(LinkNode *)malloc(sizeof(LinkNode)) ;
p->adjvex=arc->vex1 ; p->info=arc->info ;
p->nextarc=NULL ;  /* 边的起始表结点赋值  */
q=(LinkNode *)malloc(sizeof(LinkNode)) ;
q->adjvex=arc->vex2 ; q->info=arc->info ;
q->nextarc=NULL ;  /* 边的末尾表结点赋值  */
```

```
    if (G->kind==AG||G->kind==WAG)
       {  q->nextarc=G->adjlist[k].firstarc ;
          G->adjlist[k].firstarc=q ;
          p->nextarc=G->adjlist[j].firstarc ;
          G->adjlist[j].firstarc=p ;
       }  /*  是无向图, 用头插入法插入到两个单链表  */
    else       /*  建立有向图的邻接链表, 用头插入法  */
       {  q->nextarc=G->adjlist[k].firstarc ;
          G->adjlist[k].firstarc=q ;  /*  建立正邻接链表用 */
          //q->nextarc=G->adjlist[j].firstarc ;
          //G->adjlist[j].firstarc=q ;  /* 建立逆邻接链表用 */
       }
    return(1);
}
```

7.2.3 十字链表法

十字链表(Orthogonal List)是有向图的另一种链式存储结构，是将有向图的正邻接表和逆邻接表结合起来得到的一种链表。

在这种结构中，每条弧的弧头结点和弧尾结点都存放在链表中，并将弧结点分别组织到以弧尾结点为头(顶点)结点和以弧头结点为头(顶点)结点的链表中。这种结构的结点逻辑结构如图7-12所示。

图7-12 十字链表结点结构

- **data域**：存储和顶点相关的信息；
- 指针域**firstin**：指向以该顶点为弧头的第一条弧所对应的弧结点；
- 指针域**firstout**：指向以该顶点为弧尾的第一条弧所对应的弧结点；
- 尾域**tailvex**：指示弧尾顶点在图中的位置；
- 头域**headvex**：指示弧头顶点在图中的位置；
- 指针域**hlink**：指向弧头相同的下一条弧；
- 指针域**tlink**：指向弧尾相同的下一条弧；
- **Info域**：指向该弧的相关信息；

结点类型定义

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     //  最大顶点数
typedef struct ArcNode
   {   int  tailvex , headvex ;   //  尾结点和头结点在图中的位置
       InfoType    info  ;       //  与弧相关的信息, 如权值
       struct ArcNode  *hlink , *tlink ;
   }ArcNode ;    /*  弧结点类型定义    */
typedef struct VexNode
   {  VexType  data;     //  顶点信息
       ArcNode  *firstin , *firstout ;
   }VexNode ;    /*  顶点结点类型定义    */
```

```
typedef struct
  {  int vexnum ;
     VexNode  xlist[MAX_VEX] ;
  }OLGraph ;   /*  图的类型定义   */
```

图7-13所示是一个有向图及其十字链表(略去了表结点的info域)。

从这种存储结构图可以看出，从一个顶点结点的firstout出发，沿表结点的tlink指针构成了正邻接表的链表结构，而从一个顶点结点的firstin出发，沿表结点的hlink指针构成了逆邻接表的链表结构。

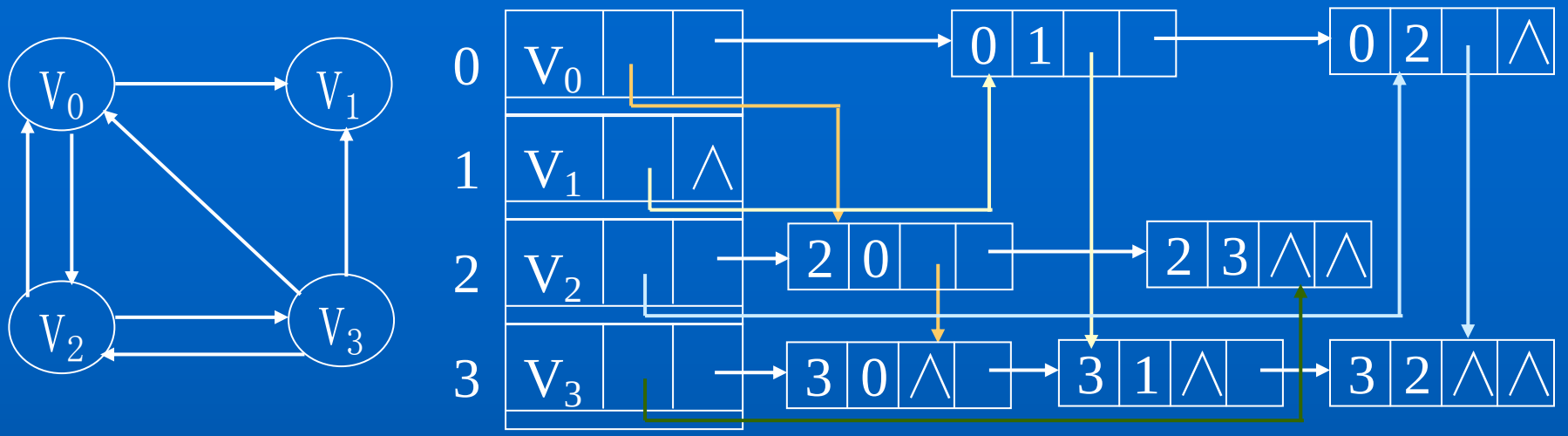


图7-13　有向图的十字链表结构

7.2.4 邻接多重表

邻接多重表(Adjacency Multilist)是无向图的另一种链式存储结构。

邻接表是无向图的一种有效的存储结构，在无向图的邻接表中，一条边(v,w)的两个表结点分别初选在以v和w为头结点的链表中，很容易求得顶点和边的信息，但在涉及到边的操作会带来不便。

邻接多重表的结构和十字链表类似，每条边用一个结点表示；邻接多重表中的顶点结点结构与邻接表中的完全相同，而表结点包括六个域如图7-14所示。

顶点结点

data	firstedge

表结点

mark	ivex	jvex	info	ilink	jlink

图7-14 邻接多重表的结点结构

- **Data域**：存储和顶点相关的信息；
- 指针域**firstedge**：指向依附于该顶点的第一条边所对应的表结点；
- 标志域**mark**：用以标识该条边是否被访问过；
- **ivex**和**jvex**域：分别保存该边所依附的两个顶点在图中的位置；
- **info**域：保存该边的相关信息；
- 指针域**ilink**：指向下一条依附于顶点**ivex**的边；
- 指针域**jlink**：指向下一条依附于顶点**jvex**的边；

结点类型定义

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     /*  最大顶点数  */
typedef  emnu {unvisited , visited}
Visitting ;
typedef struct EdgeNode
   {  Visitting  mark ;    // 访问标记
      int  ivex , jvex  ;   // 该边依附的两个结点在图中的位置
      InfoType    info  ;       // 与边相关的信息, 如权值
      struct EdgeNode  *ilink , *jlink ;
         // 分别指向依附于这两个顶点的下一条边
   }EdgeNode ;    /*  弧边结点类型定义   */
```

```
typedef struct VexNode
  {  VexType  data;     // 顶点信息
     ArcNode  *firsedge ;  //  指向依附于该顶点的第一条边
  }VexNode ;    /*  顶点结点类型定义   */
typedef struct
  {  int vexnum ;
     VexNode mullist[MAX_VEX] ;
}AMGraph ;
```

图7-15所示是一个无向图及其邻接多重表。

邻接多重表与邻接表的区别：

后者的同一条边用两个表结点表示，而前者只用一个表结点表示；除标志域外，邻接多重表与邻接表表达的信息是相同的，因此，操作的实现也基本相似。

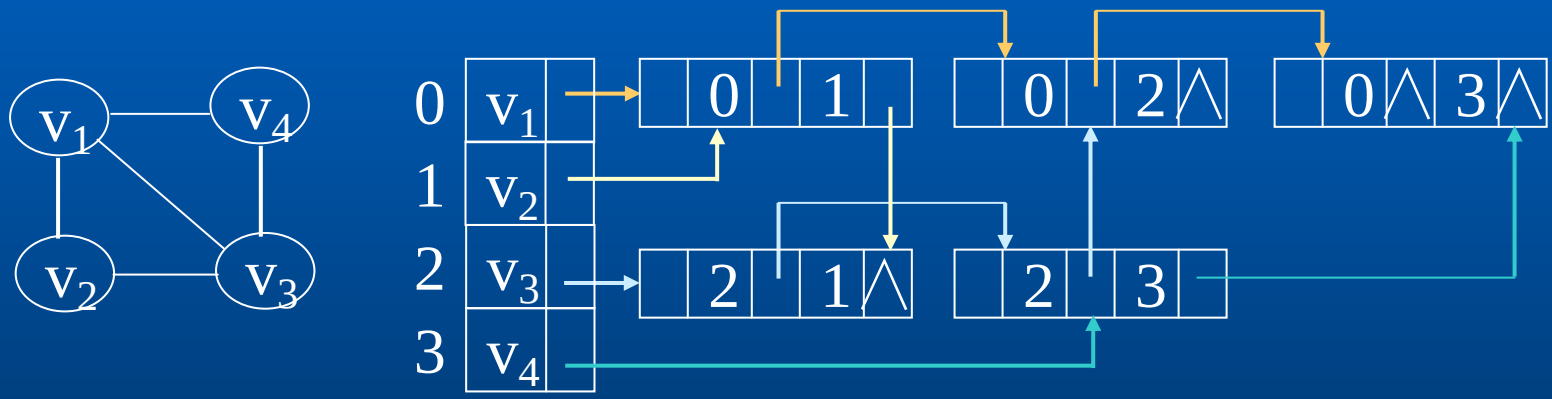


图7-15 无向图及其多重邻接链表

7.2.5 图的边表存储结构

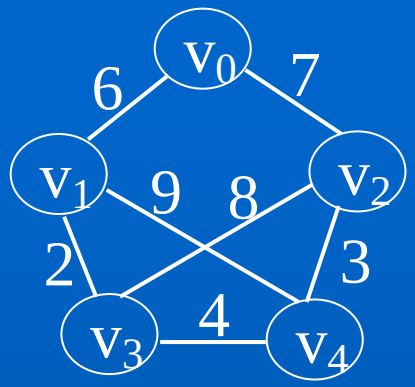
在某些应用中，有时主要考察图中各个边的权值以及所依附的两个顶点，即图的结构主要由边来表示，称为边表存储结构。

在边表结构中，边采用顺序存储，每个边元素由三部分组成：边所依附的两个顶点和边的权值；图的顶点用另一个顺序结构的顶点表存储。如图7-16所示。

边表存储结构的形式描述如下：

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     /*  最大顶点数  */
#define MAX_EDGE  100     /*  最大边数  */
```

```
typedef struct ENode
  { int ivex,jvex;   /* 边所依附的两个顶点 */
    WeightType  weight;      /* 边的权值 */
  }ENode;   /* 边表元素类型定义 */
typedef struct
  { int vexnum,edgenum;    /* 顶点数和边数 */
    VexType vexlist[MAX_VEX];   /* 顶点表 */
    ENode edgelist[MAX_EDGE];    /* 边表 */
  }ELGraph;
```



顶点表

0	v_0
1	v_1
2	v_2
3	v_3
4	v_4

边　表

0	1	6
0	2	7
1	3	2
1	4	9
2	3	8
2	4	3
3	4	4

图7-16　无向图的边表表示

7.3 图的遍历

图的遍历(Travering Graph)：从图的某一顶点出发，访遍图中的其余顶点，且每个顶点仅被访问一次。图的遍历算法是各种图的操作的基础。

- 复杂性：图的任意顶点可能和其余的顶点相邻接，可能在访问了某个顶点后，沿某条路径搜索后又回到原顶点。
- 解决办法：在遍历过程中记下已被访问过的顶点。设置一个辅助向量Visited[1...n](n为顶点数)，其初值为0，一旦访问了顶点v_i后，使Visited[i]为1或为访问的次序号。

图的遍历算法有深度优先搜索算法和广度优先搜索算法。采用的数据结构是(正)邻接链表。

7.3.1 深度优先搜索算法

深度优先搜索(Depth First Search--DFS)遍历类似树的先序遍历，是树的先序遍历的推广。

1 算法思想

设初始状态时图中的所有顶点未被访问，则：

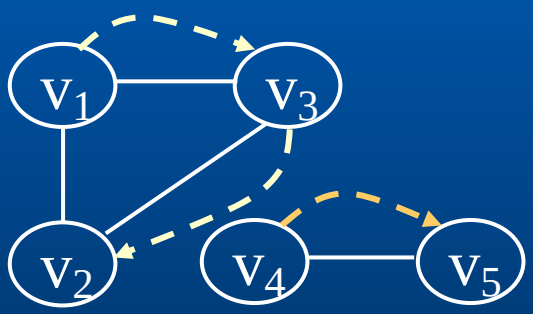
(1)：从图中某个顶点v_i出发，访问v_i；然后找到v_i的一个邻接顶点v_{i1}；

(2)：从v_{i1}出发，深度优先搜索访问和v_{i1}相邻接且未被访问的所有顶点；

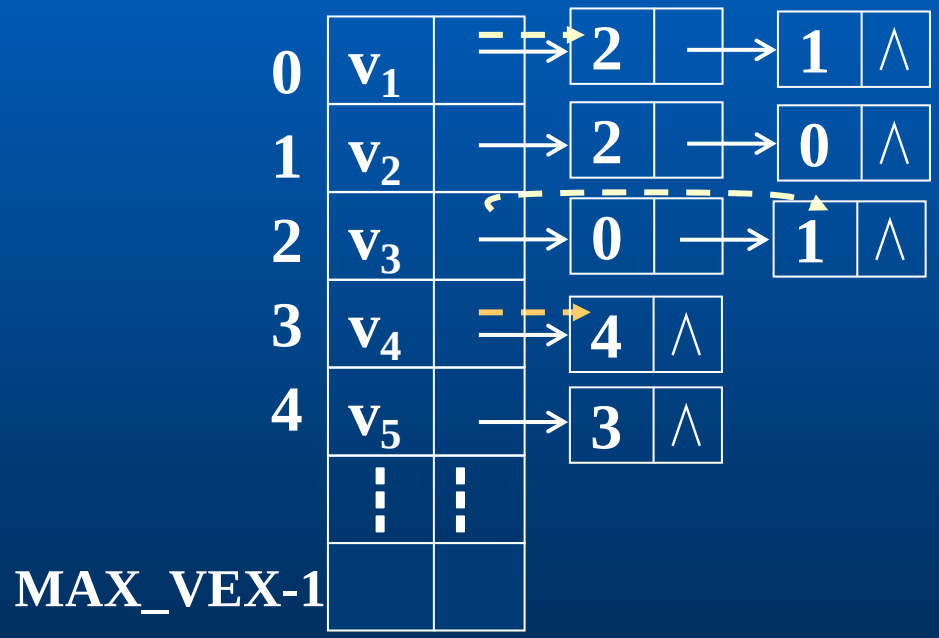
(3)：转(1)，直到和v_i相邻接的所有顶点都被访问为止

(4)：继续选取图中未被访问顶点vj作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

图7-17是无向图的深度优先搜索遍历示例(红色箭头)。某种DFS次序是：$V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5$



(a) 无向图G



(b) G的邻接链表

图7-17 无向图深度优先搜索遍历

2 算法实现

由算法思想知，这是一个递归过程。因此，先设计一个从某个顶点(编号)为v_0开始深度优先搜索的函数，便于调用。

在遍历整个图时，可以对图中的每一个未访问的顶点执行所定义的函数。

```
typedef emnu {FALSE , TRUE} BOOLEAN ;
BOOLEAN Visited[MAX_VEX] ;
```

```
void  DFS(ALGraph *G , int v)
   {  LinkNode  *p ;
      Visited[v]=TRUE ;
      Visit[v] ;       /*  置访问标志，访问顶点v  */
      p=G->AdjList[v].firstarc;   /*  链表的第一个结点  */
      while (p!=NULL)
         {  if  (!Visited[p->adjvex]) DFS(G, p->adjvex) ;
                /*  从v的未访问过的邻接顶点出发深度优先搜索   */
            p=p->nextarc ;
         }
   }
```

```
void DFS_traverse_Grapg(ALGraph *G)
   {  int v ;
      for (v=0 ; v<G->vexnum ; v++)
         Visited[v]=FALSE ;    /*  访问标志初始化  */
      p=G->AdjList[v].firstarc ;
      for (v=0 ; v<G->vexnum ; v++)
         if (!Visited[v])   DFS(G , v);
   }
```

3 算法分析

遍历时，对图的每个顶点至多调用一次**DFS**函数。其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。当图有**e**条边，其时间复杂度为**O(e)**，总时间复杂度为**O(n+e)** 。

7.3.2 广度优先搜索算法

广度优先搜索(Breadth First Search--BFS)遍历类似树的按层次遍历的过程。

1 算法思想

设初始状态时图中的所有顶点未被访问，则：

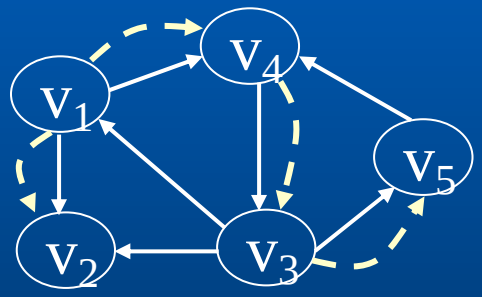
⑴：从图中某个顶点v_i出发，访问v_i；

⑵：访问v_i的所有相邻接且未被访问的所有顶点v_{i1}，v_{i2}，…，v_{im}；

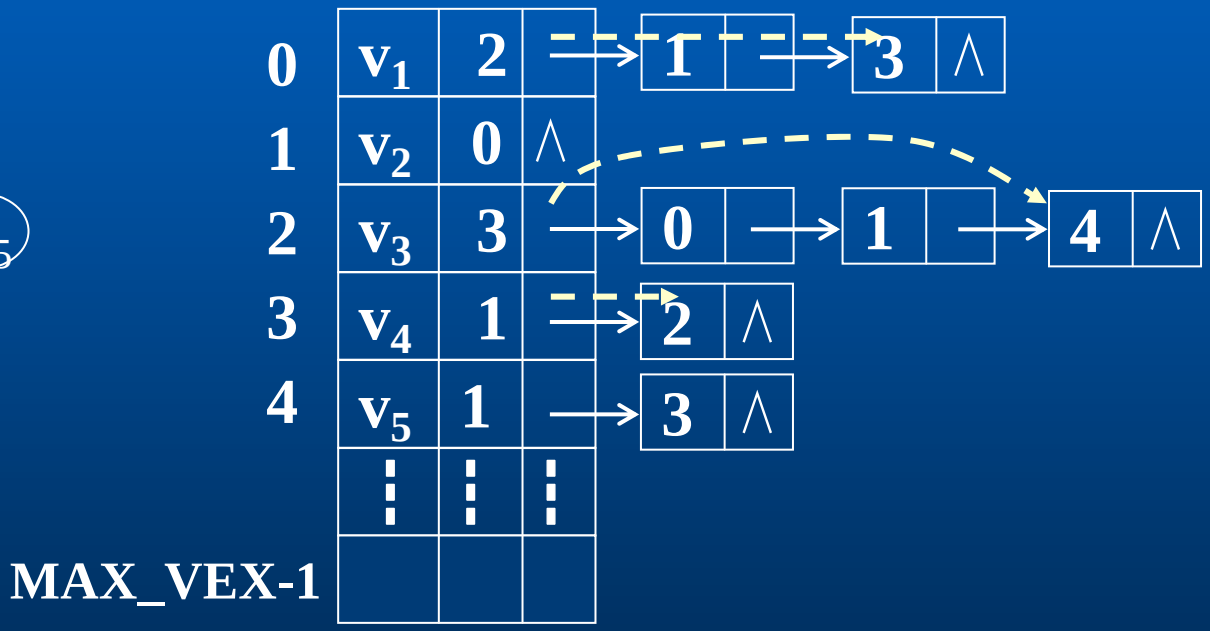
⑶：以v_{i1}，v_{i2}，…，v_{im}的次序，以$v_{ij}(1 \leqq j \leqq m)$依此作为$v_i$，转⑴；

(4)：继续选取图中未被访问顶点v_k作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

图7-18是有向图的广度优先搜索遍历示例(红色箭头)。上述图的BFS次序是：$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_5$



(a) 有向图G'



(b) G'的正邻接链表

图7-18 有向图广度优先搜索遍历

2 算法实现

为了标记图中顶点是否被访问过，同样需要一个访问标记数组；其次，为了依此访问与v_i相邻接的各个顶点，需要附加一个队列来保存访问v_i的相邻接的顶点。

```
typedef emnu {FALSE , TRUE} BOOLEAN ;
BOOLEAN Visited[MAX_VEX] ;
typedef struct Queue
  { int  elem[MAX_VEX] ;
    int front , rear ;
  }Queue ;    /*  定义一个队列保存将要访问顶点  */
```

```
void BFS_traverse_Grapg(ALGraph *G)
   {  int k ,v , w ;
      LinkNode  *p ; Queue  *Q ;
      Q=(Queue *)malloc(sizeof(Queue)) ;
      Q->front=Q->rear=0 ;   /*  建立空队列并初始化  */
      for (k=0 ; k<G->vexnum ; k++)
         Visited[k]=FALSE ;     /*  访问标志初始化  */
      for (k=0 ; k<G->vexnum ; k++)
         {  v=G->AdjList[k].data ;    /*  单链表的头顶点  */
            if (!Visited[v])    /*   v尚未访问   */
               {  Q->elem[++Q->rear]=v ;   /*   v入对   */
                  while (Q->front!=Q->rear)
```

```
        {  w=Q->elem[++Q->front] ;
           Visited[w]=TRUE ;    /* 置访问标志 */
           Visit(w) ;    /* 访问队首元素 */
           p=G->AdjList[w].firstarc ;
           while (p!=NULL)
              {  if (!Visited[p->adjvex])
                    Q->elem[++Q->rear]=p->adjvex ;
                 p=p->nextarc ;
              }
        }  /* end while */
     }   /* end if */
  } /* end for */
}
```

用广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是邻接点搜索次序不同，因此，广度优先搜索算法遍历图的总时间复杂度为**O(n+e)** 。

图的遍历可以系统地访问图中的每个顶点，因此，图的遍历算法是图的最基本、最重要的算法，许多有关图的操作都是在图的遍历基础之上加以变化来实现的。

7.4 图的连通性问题

本节所讨论的内容是图的遍历算法的具体应用。

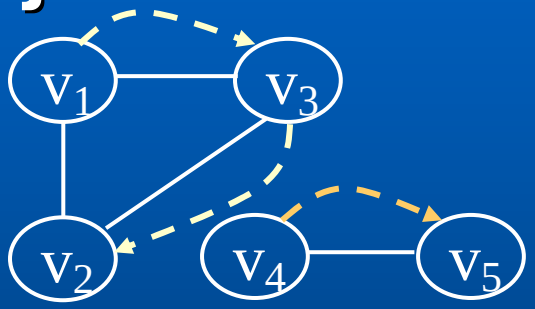
7.4.1 无向图的连通分量与生成树

1 无向图的连通分量和生成树

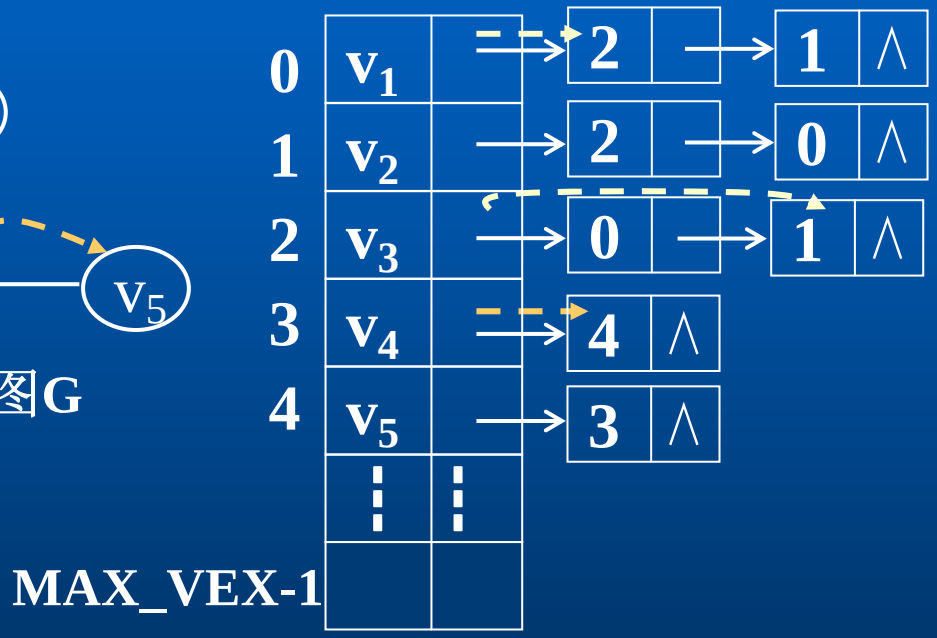
对于无向图，对其进行遍历时：

- 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
- 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

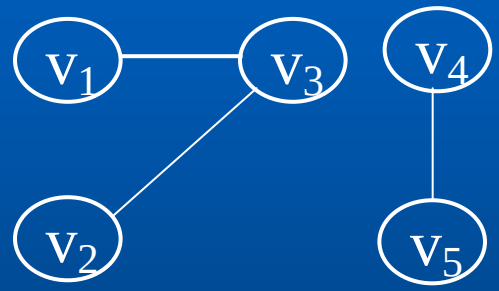
如图7-19所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，2次调用DFS所得到的顶点访问序列集是：{ v1 ,v3 ,v2}和{ v4 ,v5 }



(a) 无向图G



(b) G的邻接链表



(c) 深度优先生成森林

图7-19 无向图及深度优先生成森林

（1）若G=(V,E)是无向连通图， 顶点集和边集分别是V(G)，E(G)。若从G中任意点出发遍历时，E(G)被分成两个互不相交的集合：

T(G)：遍历过程中所经过的边的集合；

B(G)：遍历过程中未经过的边的集合；

显然： E(G)=T(G)∪B(G)，T(G)∩B(G)=Ø

显然，图G'=(V, T(G))是G的极小连通子图，且G'是一棵树。G'称为图G的一棵生成树。

从任意点出发按DFS算法得到生成树G'称为深度优先生成树；按BFS算法得到的G'称为广度优先生成树。

⑵ 若$G=(V,E)$是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集：$V_1(G)$，$V_2(G)$，…，$V_n(G)$和相应所经过的边集：$T_1(G)$，$T_2(G)$，…，$T_n(G)$。

则对应的顶点集和边集的二元组：

$G_i=(V_i(G),T_i(G))$

$(1\leqslant i\leqslant n)$是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。

说明：当给定无向图要求画出其对应的生成树或生成森林时，必须先给出相应的邻接表，然后才能根据邻接表画出其对应的生成树或生成森林。

2 图的生成树和生成森林算法

对图的深度优先搜索遍历DFS(或BFS)算法稍作修改，就可得到构造图的DFS生成树算法。

在算法中，树的存储结构采用孩子—兄弟表示法。首先建立从某个顶点V出发，建立一个树结点，然后再分别以V的邻接点为起始点，建立相应的子生成树，并将其作为V 结点的子树链接到V结点上。显然，算法是一个递归算法。

算法实现：

(1) DFStree算法

```
typedef  struct  CSNode
   {  ElemType  data ;
      struct  CSNode *firstchild , *nextsibling ;
   }CSNode ;
CSNode  *DFStree(ALGraph *G , int v)
   {  CSNode  *T , *ptr , *q ;
      LinkNode  *p ;  int w ;
      Visited[v]=TRUE ;
      T=(CSNode *)malloc(sizeof(CSNode)) ;
      T->data=G->AdjList[v].data ;
      T->firstchild=T->nextsibling=NULL ; //  建立根结点
```

```
    q=NULL ; p=G->AdjList[v].firstarc ;
    while (p!=NULL)
       {  w=p->adjvex ;
          if  (!Visited[w])
             {  ptr=DFStree(G,w) ;      /*  子树根结点  */
                if  (q==NULL)  T->firstchild=ptr ;
                else  q->nextsibling=ptr ;
               q=ptr ;
             }
          p=p->nextarc ;
       }
    return(T) ;
}
```

(2) BFStree算法

```
typedef struct Queue
   {  int  elem[MAX_VEX] ;
      int front , rear ;
   }Queue ;     /*  定义一个队列保存将要访问顶点  */
CSNode  *BFStree(ALGraph *G ,int v)
  {  CSNode  *T , *ptr , *q ;
     LinkNode  *p ; Queue  *Q ;
     int w , k ;
     Q=(Queue *)malloc(sizeof(Queue)) ;
     Q->front=Q->rear=0 ;  /*建立空队列并初始化*/
     Visited[v]=TRUE ;
```

```
T=(CSNode *)malloc(sizeof(CSNode)) ;
T->data=G->AdjList[v].data ;
T->firstchild=T->nextsibling=NULL ; // 建立根结点
Q->elem[++Q->rear]=v ;  /*  v入队  */
while (Q->front!=Q->rear)
   {  w=Q->elem[++Q->front] ;  q=NULL ;
      p=G->AdjList[w].firstarc ;
      while (p!=NULL)
         {  k=p->adjvex ;
            if  (!Visited[k])
               {  Visited[k]=TRUE ;
```

```
                ptr=(CSNode *)malloc(sizeof(CSNode)) ;
                ptr->data=G->AdjList[k].data ;
                ptr->firstchild=T->nextsibling=NULL ;
                if  (q==NULL)  T->firstchild=ptr ;
                else  q->nextsibling=ptr ;
                q=ptr ;
                Q->elem[++Q->rear]=k ;  /*   k入对   */
            }   /*   end  if   */
          p=p->nextarc ;
        }    /*  end  while  p  */
    }   /*  end whil  Q  */
  return(T) ;
} /*求图G广度优先生成树算法BFStree*/
```

(3) 图的生成森林算法

```
CSNode  *DFSForest(ALGraph *G)
   {  CSNode  *T , *ptr , *q ;  int w ;
      for (w=0; w<G->vexnum; w++) Visited[w]=FALSE;
      T=NULL ;
      for (w=0 ; w<G->vexnum ; w++)
         if (!Visited[w])
            {  ptr=DFStree(G, w) ;
               if  (T==NULL)  T=ptr ;
               else  q->nextsibling=ptr ;
               q=ptr ;   }
      return(T) ;
   }
```

7.4.2 有向图的强连通分量

对于有向图，在其每一个强连通分量中，任何两个顶点都是可达的。 $\forall V \in G$，与V可相互到达的所有顶点就是包含V的强连通分量的所有顶点。

设从V可到达 (以V为起点的所有有向路径的终点) 的顶点集合为$T_1(G)$，而到达V (以V为终点的所有有向路径的起点)的顶点集合为$T_2(G)$，则包含V的强连通分量的顶点集合是： $T_1(G) \cap T_2(G)$ 。

求有向图G的强连通分量的基本步骤是：

⑴ 对G进行深度优先遍历，生成G的深度优先生成森林T。

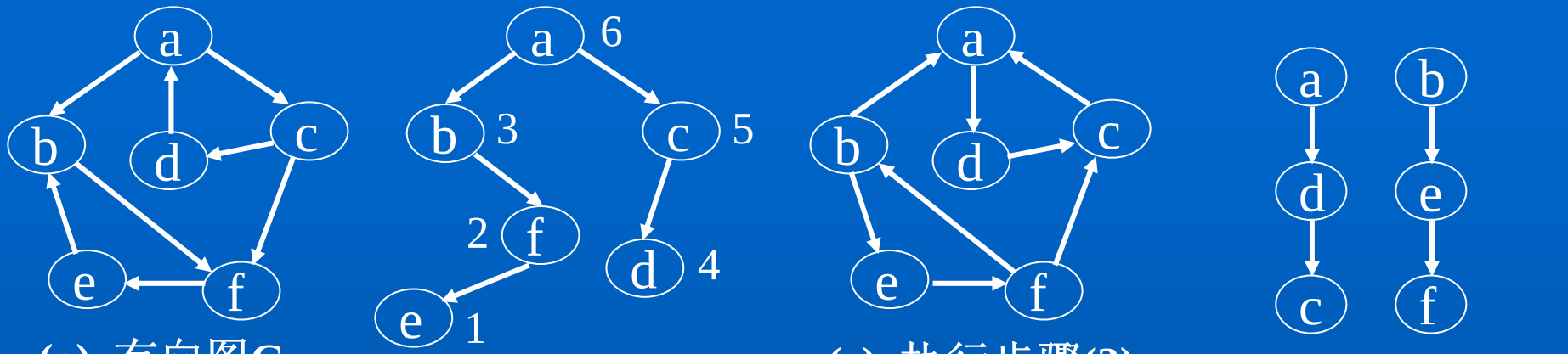
⑵ 对森林T的顶点按中序遍历顺序进行编号。

⑶ 改变G中每一条弧的方向，构成一个新的有向图G’。

⑷ 按⑵中标出的顶点编号，从编号最大的顶点开始对G’进行深度优先搜索，得到一棵深度优先生成树。若一次完整的搜索过程没有遍历G’的所有顶点，则从未访问的顶点中选择一个编号最大的顶点，由它开始再进行深度优先搜索，并得到另一棵深度优先生成树。在该步骤中，每一次深度优先搜索所得到的生成树中的顶点就是G的一个强连通分量的所有顶点。

⑸ 重复步骤⑷，直到G’中的所有顶点都被访问。

如图7-20(a)是求一棵有向树的强连通分量过程。



(a) 有向图G　(b) 执行步骤(1)和(2)　(c) 执行步骤(3)　(d) 执行步骤(4)和(5)

图7-20 利用深度优先搜索求有向图的强连通分量

在算法实现时，建立一个数组**in_order[n]**存放深度优先生成森林的中序遍历序列。对每个顶点**v**，在调用**DFS**函数结束时，将顶点依次存放在数组**in_order[n]**中。图采用十字链表作为存储结构最合适。

算法实现：

```
int in_order[MAX_VEX] ;
```

```
void  DFS(OLGraph *G , int v)  // 按弧的正向搜索
   {  ArcNode  *p ;
      Count=0 ;
      Visited[v]=TRUE ;
      for  (p=G->xlist[v].firstout ; p!=NULL ; p=p->tlink)
         if  (!Visited[p->headvex])
            DFS(G , p->headvex) ;
      in_order[count++]=v ;
   }
```

```
void  Rev_DFS(OLGraph *G , int v)
   {  ArcNode  *p ;
      Visited[v]=TRUE ;
      printf("%d" , v) ;     /*  输出顶点  */
      for  (p=G->xlist[v].firstin ; p!=NULL ; p=p->hlink)
         if  (!Visited[p->tailvex])
            Rev_DFS(G , p->tailvex) ;
   }   /*  对图G按弧的逆向进行搜索  */

void  Connected_DG(OLGraph *G)
   {   int  k=1, v, j ;
      for (v=0; v<G->vexnum; v++)
         Visited[v]=FALSE ;
```

```
    for (v=0; v<G->vexnum; v++)    /* 对图G正向遍历 */
        if (!Visited[v])  DFS(G,v) ;
    for (v=0; v<G->vexnum; v++)
        Visited[v]=FALSE ;
    for (j=G->vexnum-1; j>=0; j--)    /* 对图G逆向遍历 */
        {  v=in_order[j] ;
           if (!Visited[v])
              {  printf("\n第%d个连通分量顶点: ", k++) ;
                 Rev_DFS(G, v) ;
              }
        }
}
```

7.5 最小生成树

如果连通图是一个带权图，则其生成树中的边也带权，生成树中所有边的权值之和称为生成树的代价。

最小生成树(Minimum Spanning Tree)：带权连通图中代价最小的生成树称为最小生成树。

最小生成树在实际中具有重要用途，如设计通信网。设图的顶点表示城市，边表示两个城市之间的通信线路，边的权值表示建造通信线路的费用。n个城市之间最多可以建n×(n-1)/2条线路，如何选择其中的n-1条，使总的建造费用最低?

构造最小生成树的算法有许多，基本原则是：

构造最小生成树的算法有许多，基本原则是：

- 尽可能选取权值最小的边，但不能构成回路；
- 选择n-1条边构成最小生成树。

以上的基本原则是基于MST的如下性质：

设G=(V，E)是一个带权连通图，U是顶点集V的一个非空子集。若u∈U，v∈V-U，且(u, v)是U中顶点到V-U中顶点之间权值最小的边，则必存在一棵包含边(u, v)的最小生成树。

证明： 用反证法证明。

设图G的任何一棵最小生成树都不包含边(u,v)。设T是G的一棵生成树，则T是连通的，从u到v必有一条路径(u,…,v)，当将边(u,v)加入到T中时就构成了回路。则路径(u, …,v)中必有一条边(u’,v’)，满足u’∈U，v’∈V-U。删去边(u’,v’) 便可消除回路，同时得到另一棵生成树T’。

由于(u,v)是U中顶点到V-U中顶点之间权值最小的边，故(u,v)的权值不会高于(u’,v’)的权值，T’的代价也不会高于T， T’是包含(u,v) 的一棵最小生成树，与假设矛盾。

7.5.1 普里姆(Prim)算法

从连通网N=(U，E)中找最小生成树T=(U，TE)。

1 算法思想

⑴ 若从顶点v_0出发构造，U={v_0}，TE={}；

⑵ 先找权值最小的边(u，v)，其中u∈U且v∈V-U，并且子图不构成环，则U= U∪{v}，TE=TE∪{(u，v)}；

⑶ 重复⑵，直到U=V为止。则TE中必有n-1条边，T=(U，TE)就是最小生成树。

如图7-21所提示。

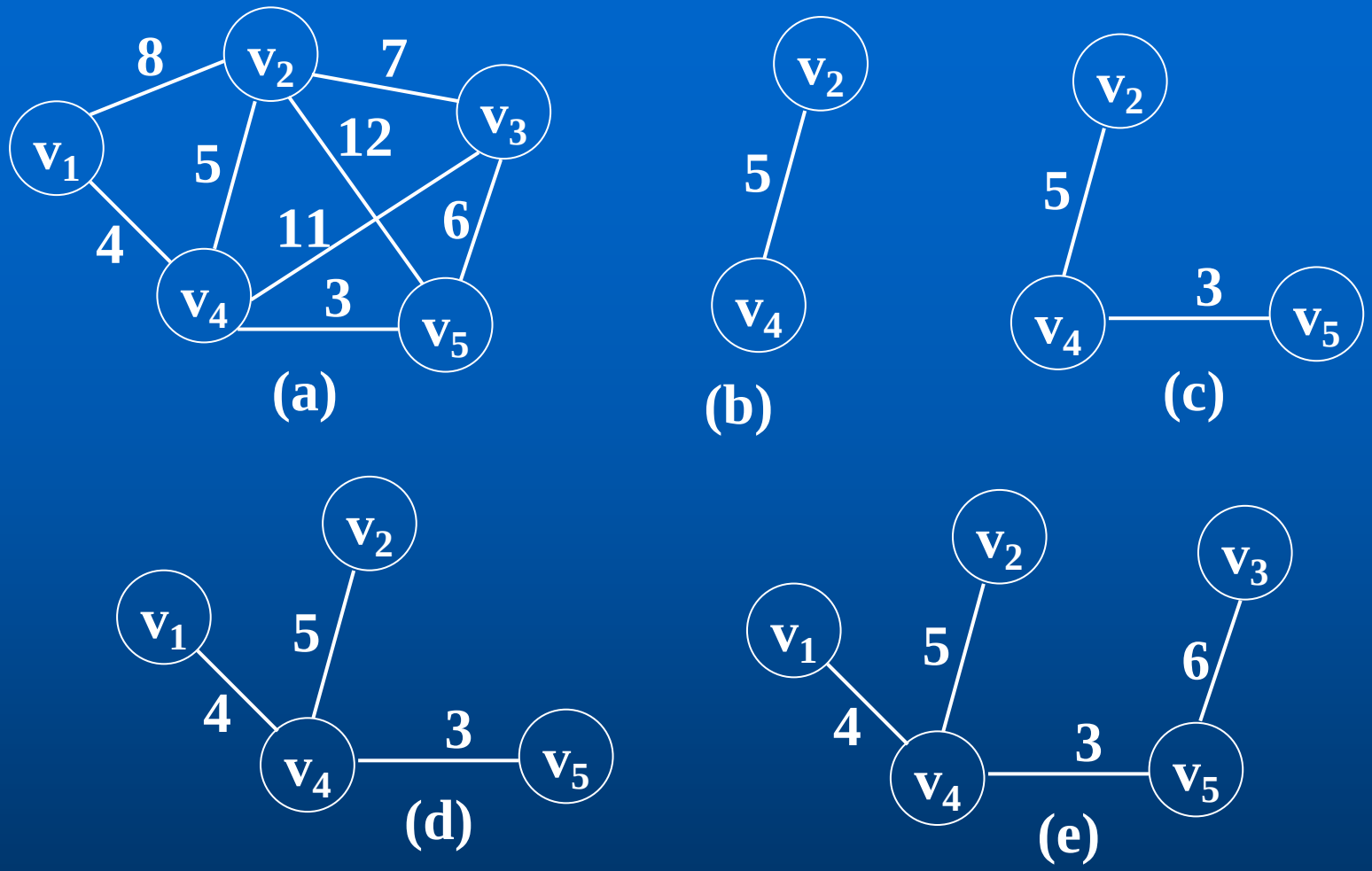


图7-21 按prime算法从v2出发构造最小生成树的过程

2 算法实现说明

设用邻接矩阵(二维数组)表示图，两个顶点之间不存在边的权值为机内允许的最大值。

为便于算法实现，设置一个一维数组closedge[n]，用来保存V- U中各顶点到U中顶点具有权值最小的边。数组元素的类型定义是：

```
struct
  {  int adjvex;     /*  边所依附于U中的顶点  */
     int lowcost;    /*  该边的权值  */
  }closedge[MAX_EDGE];
```

例如： closedge[j].adjvex=k，表明边(v_j, v_k)是V-U中顶点v_j到U中权值最小的边，而顶点v_k是该边所依附的U中的顶点。 closedge[j].lowcost存放该边的权值。

假设从顶点v_s开始构造最小生成树，表明顶点v_s首先加入到U中；初始时令

{ Closedge[s].lowcost=0；

Closedge[k].adjvex=s， Closedge[k].lowcost=cost(k, s)

表示V-U中的各顶点到U中权值最小的边(k≠s)，cost(k, s)表示边(v_k, v_s) 权值。

3 算法步骤

⑴ 从closedge中选择一条权值(不为0)最小的边(v_k, v_j)，然后做：

① 置closedge[k].lowcost为0，表示v_k已加入到U中。

② 根据新加入v_k的更新closedge中每个元素：

∀v_i∈V-U，若cost(i, k)≦colsedge[i].lowcost，表明在U中新加入顶点v_k后，(v_i, v_k)成为v_i到U中权值最小的边，置

Closedge[i].lowcost=cost(i, k)

Closedge[i].adjvex=k

⑵ 重复⑴n-1次就得到最小生成树。

如表7-1所提示。

在Prime算法中，图采用邻接矩阵存储，所构造的最小生成树用一维数组存储其n-1条边，每条边的存储结构描述：

```
typedef struct MSTEdge
   {  int  vex1, vex2 ;    /*  边所依附的图中两个顶点 */
      WeightType  weight ;     /*  边的权值  */
   }MSTEdge ;
```

算法实现

```
#define INFINITY  MAX_VAL     /* 最大值 */
MSTEdge *Prim_MST(AdjGraph *G , int u)
    /*   从第u个顶点开始构造图G的最小生成树   */
   {  MSTEdge TE[] ;  // 存放最小生成树n-1条边的数组指针
```

```
int j , k , v , min ;
for (j=0; j<G->vexnum; j++)
    {  closedge[j].adjvex=u  ;
       closedge[j].lowcost=G->adj[j][u]  ;
    }   /*  初始化数组closedge[n]  */
closedge[u].lowcost=0 ;     /*  初始时置U={u}  */
TE=(MSTEdge *)malloc((G->vexnum-
1)*sizeof(MSTEdge)) ;
for (j=0; j<G->vexnum-1; j++)
    { min= INFINITY ;
       for (v=0; v<G->vexnum; v++)
          if  (closedge[v].lowcost!=0&&
       closedge[v].Lowcost<min)
```

```
            {  min=closedge[v].lowcost ; k=v ;  }
        TE[j].vex1=closedge[k].adjvex ;
        TE[j].vex2=k ;
        TE[j].weight=closedge[k].lowcost ;
        closedge[k].lowcost=0 ;     /*  将顶点k并入U中  */
        for (v=0; v<G->vexnum; v++)
            if (G->adj[v][k]<closedge[v]. lowcost)
              {  closedge[v].lowcost= G->adj[v][k] ;
                 closedge[v].adjvex=k ;
              } /*  修改数组closedge[n]的各个元素的值  */
    }
    return(TE) ;
} /*  求最小生成树的Prime算法  */
```

表7-1 构造过程中辅组数组closedge中各分量的值的变化情况

i / closedge	0	1	2	3	4	U	V-U	K
adjvex	v_2		v_2	v_2	v_2	$\{v_2\}$	$\{v_1, v_3, v_4, v_5\}$	3
lwcost	8		7	5	12			
adjvex	v_4		v_2	v_2	v_4	$\{v_2, v_4\}$	$\{v_1, v_3, v_5\}$	4
lwcost	4		7	0	3			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5\}$	$\{v_1, v_3\}$	0
lwcost	4		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1\}$	$\{v_3\}$	2
lwcost	0		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1, v_3\}$	$\{\}$	
lwcost	0		0	0	0			

算法分析：设带权连通图有n个顶点，则算法的主要执行是二重循环： 求closedge中权值最小的边，频度为n-1； 修改closedge数组，频度为n 。因此，整个算法的时间复杂度是$O(n^2)$，与边的数目无关。

7.5.2 克鲁斯卡尔(Kruskal)算法

1 算法思想

设G=(V, E)是具有n个顶点的连通网，T=(U, TE)是其最小生成树。初值：U=V，TE={}。

对G中的边按权值大小从小到大依次选取。

(1) 选取权值最小的边(v_i，v_j)，若边(v_i，v_j)加入到TE后形成回路，则舍弃该边(边(v_i，v_j)；否则，将该边并入到TE中，即TE=TE∪{(v_i，v_j)}。

(2) 重复(1)，直到TE中包含有n-1条边为止。

如图7-22所提示。

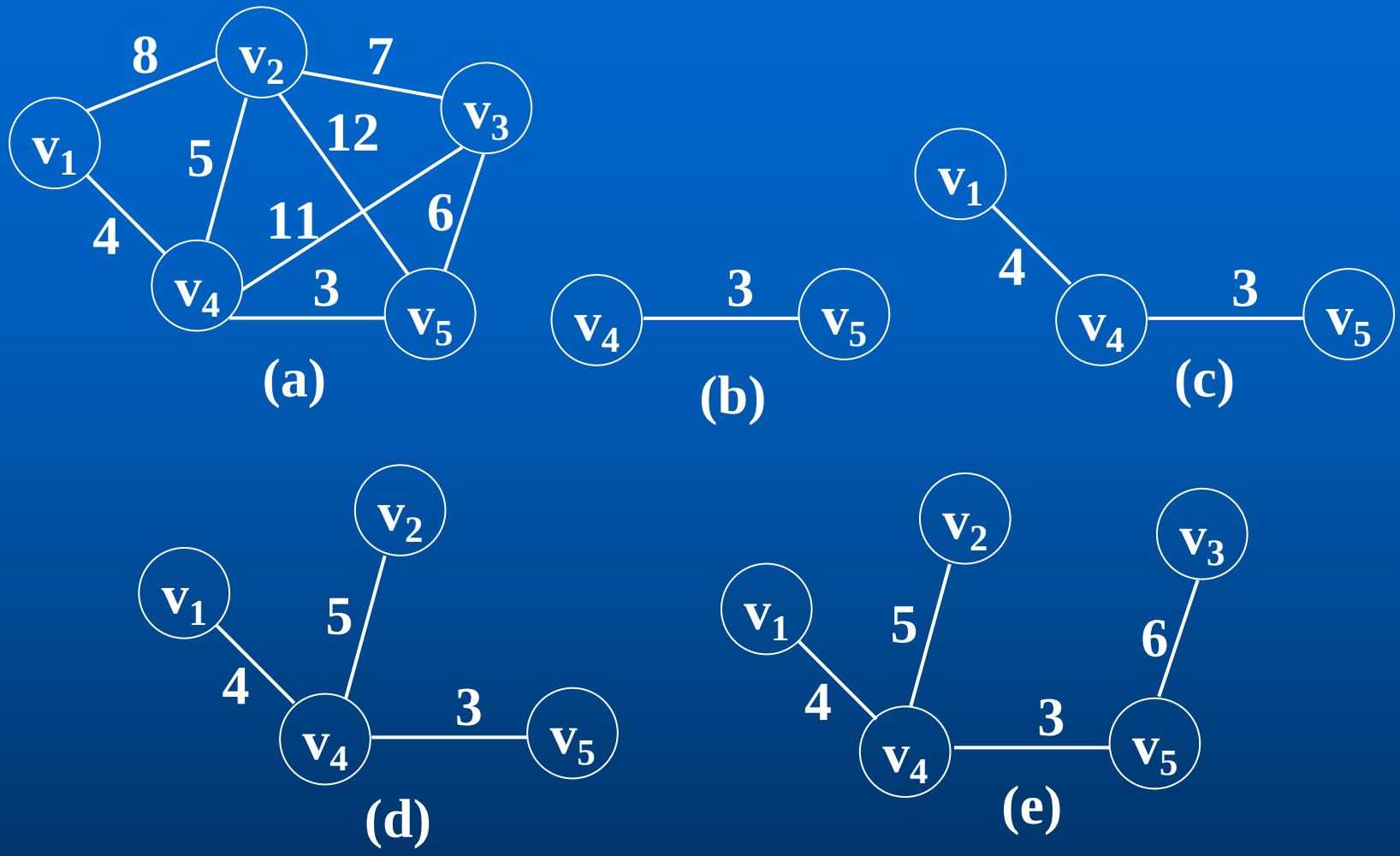


图7-22　按kruskal算法构造最小生成树的过程

2 算法实现说明

Kruskal算法实现的关键是：当一条边加入到TE的集合后，如何判断是否构成回路?

简单的解决方法是：定义一个一维数组Vset[n]，存放图T中每个顶点所在的连通分量的编号。

- 初值：Vset[i]=i，表示每个顶点各自组成一个连通分量，连通分量的编号简单地使用顶点在图中的位置(编号)。
- 当往T中增加一条边(v_i，v_j) 时，先检查Vset[i]和Vset[j]值：
  - ☆ 若Vset[i]=Vset[j]：表明v_i和v_j处在同一个连通分量中，加入此边会形成回路；

☆ 若Vset[i]≠Vset[j]，则加入此边不会形成回路，将此边加入到生成树的边集中。

◆ 加入一条新边后，将两个不同的连通分量合并：将一个连通分量的编号换成另一个连通分量的编号。

算法实现

```
MSTEdge *Kruskal_MST(ELGraph *G)
    /*   用Kruskal算法构造图G的最小生成树    */
  {  MSTEdge TE[];
     int  j, k, v, s1, s2, Vset[];
     WeightType  w;
     Vset=(int  *)malloc(G->vexnum*sizeof(int))
     ;
```

```
for (j=0; j<G->vexnum; j++)
    Vset[j]=j ;     /*  初始化数组Vset[n]  */
sort(G->edgelist) ;  /*  对表按权值从小到大排序  */
j=0 ; k=0 ;
while (k<G->vexnum-1&&j< G->edgenum)
   {  s1=Vset[G->edgelist[j].vex1] ;
      s2=Vset[G->edgelist[j].vex2] ;
/*  若边的两个顶点的连通分量编号不同, 边加入到TE中  */
      if  (s1!=s2)
         {  TE[k].vex1=G->edgelist[j].vex1 ;
            TE[k].vex2=G->edgelist[j].vex2 ;
            TE[k].weight=G->edgelist[j].weight ;
```

```
                k++ ;
                for (v=0; v<G->vexnum; v++)
                    if  (Vset[v]==s2)  Vset[v]=s1 ;
            }
        j++ ;
    }
  free(Vset) ;
  return(TE) ;
}    /*  求最小生成树的Kruskal算法  */
```

算法分析：设带权连通图有n个顶点，e条边，则算法的主要执行是：

- Vset数组初始化：时间复杂度是O(n)；
- 边表按权值排序：若采用堆排序或快速排序，时间复杂度是$O(e^{\log}e)$；
- while循环：最大执行频度是O(n)，其中包含修改Vset数组，共执行n-1次，时间复杂度是$O(n^2)$；

整个算法的时间复杂度是$O(e^{\log}e+n^2)$。

7.6 有向无环图及其应用

有向无环图(Directed Acycling Graph)：是图中没有回路(环)的有向图。是一类具有代表性的图，主要用于研究工程项目的工序问题、工程时间进度问题等。

一个工程(project)都可分为若干个称为活动(active)的子工程(或工序)，各个子工程受到一定的条件约束：某个子工程必须开始于另一个子工程完成之后；整个工程有一个开始点(起点)和一个终点。人们关心：

- 工程能否顺利完成?影响工程的关键活动是什么?
- 估算整个工程完成所必须的最短时间是多少?

对工程的活动加以抽象：图中顶点表示活动，有向边表示活动之间的优先关系，这样的有向图称为**顶点表示活动的网(Activity On Vertex Network，AOV网)**。

7.6.1 拓扑排序

1 定义

拓扑排序(Topological Sort)：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。

- 集合上的关系：集合A上的关系是从A到A的关系(A×A)。
- 关系的自反性：若∀a∈A有(a，a)∈R，称集合A上的关系R是自反的。
- 关系的对称性：如果对于a，b∈A，只要有(a，b)∈R就有(b，a)∈R，称集合A上的关系R是对称的。

- 关系的对称性与反对称性：如果对于a，b∈A，只要有(a，b)∈R就有(b，a)∈R，称集合A上的关系R是对称的。如果对于a，b∈A，仅当a=b时有(a，b)∈R和(b，a)∈R，称集合A上的关系R是反对称的。
- 关系的传递性：若a，b，c∈A，若(a，b)∈R，并且(b，c)∈R，则(a，c)∈R，称集合A上的关系R是传递的。
- 偏序：若集合A上的关系R是自反的，反对称的和传递的，则称R是集合A上的偏序关系。
- 全序：设R是集合A上的偏序关系，∀a，b∈A，必有aRb或bRa，则称R是集合A上的全序关系。

即偏序是指集合中仅有部分元素之间可以比较，而全序是指集合中任意两个元素之间都可以比较。

在AOV网中，若有有向边<i, j>，则i是j的直接前驱，j是i的直接后继；推而广之，若从顶点i到顶点j有有向路径，则i是j的前驱，j是i的后继。

在AOV网中，不能有环，否则，某项活动能否进行是以自身的完成作为前提条件。

检查方法：对有向图的顶点进行拓扑排序，若所有顶点都在其拓扑有序序列中，则无环。

有向图的拓扑排序：构造AOV网中顶点的一个拓扑线性序列$(v'_1, v'_2, \cdots, v'_n)$，使得该线性序列不仅保持原来有向图中顶点之间的优先关系，而且对原图中没有优先关系的顶点之间也建立一种(人为的)优先关系

手工实现

如图7-23是一个有向图的拓扑排序过程，其拓扑序列是：（v_1,v_6,v_4,v_3,v_2,v_5）

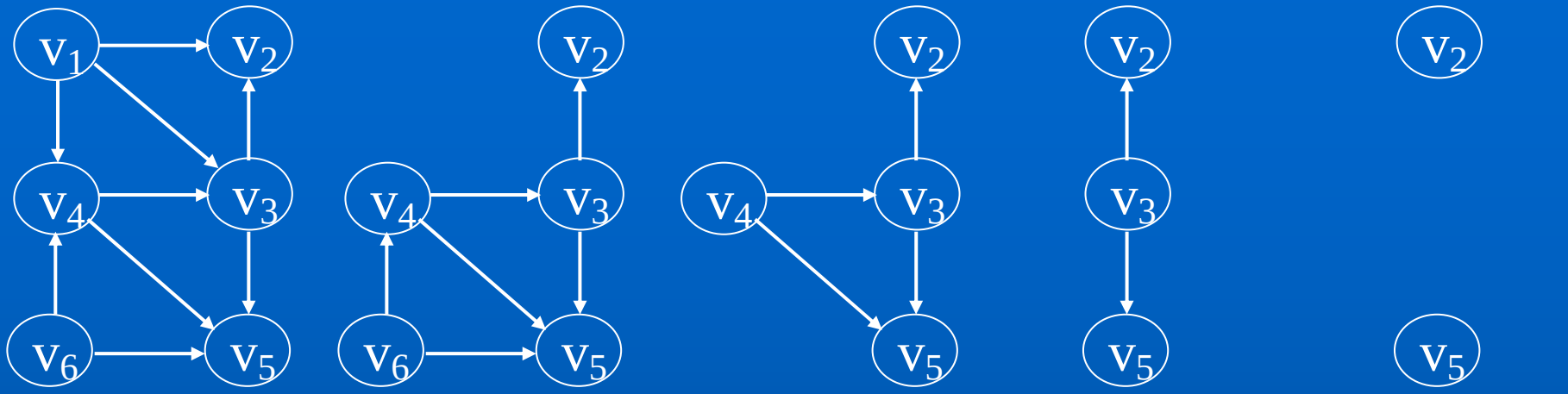
2 拓扑排序算法

算法思想

① 在AOV网中选择一个没有前驱的顶点且输出；

② 在AOV网中删除该顶点以及从该顶点出发的(以该顶点为尾的弧)所有有向弧(边)；

③ 重复①、②，直到图中全部顶点都已输出(图中无环)或图中不存在无前驱的顶点(图中必有环)。



(a) 有向图　(b) 输出v_1后　(c) 输出v_6后　(d) 输出v_4后　(e) 输出v_3后

图7-23 有向图的拓扑排序过程

3 算法实现说明

- 采用正邻接链作为AOV网的存储结构；
- 设立堆栈，用来暂存入度为0的顶点；
- 删除顶点以它为尾的弧：弧头顶点的入度减1。

算法实现

(1) 统计各顶点入度的函数

```
void count_indegree(ALGraph *G)
   {  int k ; LinkNode *p ;
      for (k=0; k<G->vexnum; k++)
         G->adjlist[k].indegree=0 ;     /*  顶点入度初始化  */
      for (k=0; k<G->vexnum; k++)
         {  p=G->adjlist[k].firstarc ;
            while (p!=NULL)     /*  顶点入度统计  */
               {  G->adjlist[p->adjvex].indegree++ ;
                  p=p->nextarc ;
               }
         }
   }
```

(2) 拓扑排序算法

```
int  Topologic_Sort(ALGraph *G, int topol[])
    /*  顶点的拓扑序列保存在一维数组topol中  */
   {  int  k, no, vex_no, top=0, count=0, boolean=1 ;
      int  stack[MAX_VEX] ;       /*  用作堆栈  */
      LinkNode *p ;
      count_indegree(G) ;   /*  统计各顶点的入度  */
      for (k=0; k<G->vexnum; k++)
         if  (G->adjlist[k].indegree==0)
            stack[++top]=G->adjlist[k].data ;
      do
         {  if (top==0)  boolean=0 ;
```

```
else
   {   no=stack[top--] ;     /*  栈顶元素出栈  */
       topl[++count]=no ;    /*  记录顶点序列  */
       p=G->adjlist[no].firstarc ;
       while (p!=NULL)     /*删除以顶点为尾的弧*/
           {   vex_no=p->adjvex ;
               G->adjlist[vex_no].indegree-- ;
               if  (G->adjlist[vex_no].indegree==0)
                   stack[++top]=vex_no  ;
               p=p->nextarc ;
           }
   }
}while(boolean==0) ;
```

```
    if (count<G->vexnum)  return(-1) ;
    else  return(1) ;
}
```

算法分析：设AOV网有n个顶点，e条边，则算法的主要执行是：

- 统计各顶点的入度：时间复杂度是O(n+e)；
- 入度为0的顶点入栈：时间复杂度是O(n)；
- 排序过程：顶点入栈和出栈操作执行n次，入度减1的操作共执行e次，时间复杂度是O(n+e)；

因此，整个算法的时间复杂度是O(n+e)。

7.6.2 关键路径(Critical Path)

与AOV网相对应的是AOE(Activity On Edge)，是边表示活动的有向无环图，如图7-24所示。图中顶点表示事件(Event)，每个事件表示在其前的所有活动已经完成，其后的活动可以开始；弧表示活动，弧上的权值表示相应活动所需的时间或费用。

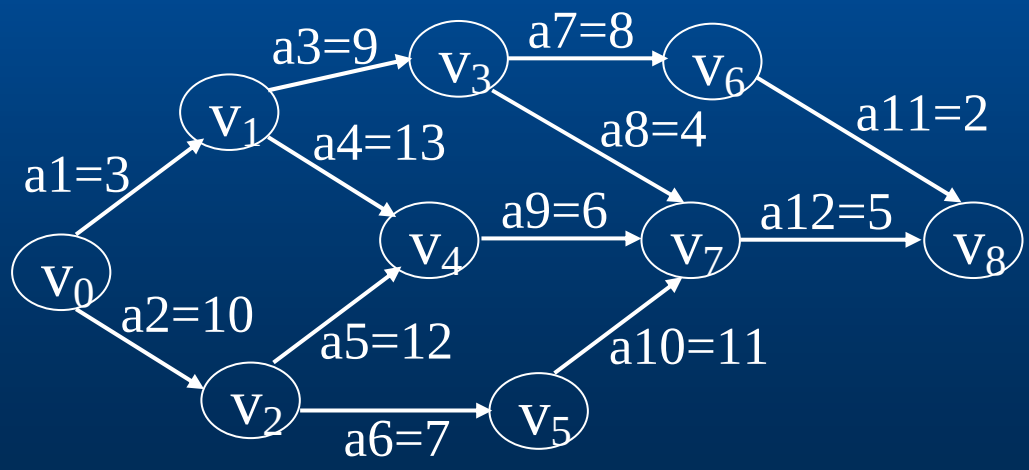


图7-24 一个AOE网

1 与AOE有关的研究问题

- 完成整个工程至少需要多少时间?
- 哪些活动是影响工程进度(费用)的关键?

工程完成最短时间：从起点到终点的最长路径长度(路径上各活动持续时间之和)。长度最长的路径称为关键路径，关键路径上的活动称为关键活动。关键活动是影响整个工程的关键。

设v_0是起点，从v_0到v_i的最长路径长度称为事件v_i的最早发生时间，即是以v_i为尾的所有活动的最早发生时间。

若活动a_i是弧<j, k>，持续时间是dut(<j, k>)，设：

- e(i)：表示活动a_i的最早开始时间；

- ***l*(i)**：在不影响进度的前提下，表示活动a_i的最晚开始时间； 则***l*(i)-e(i)**表示活动a_i的时间余量，若***l*(i)-e(i)=0**，表示活动a_i是关键活动。
- **ve(i)**：表示事件v_i的最早发生时间，即从起点到顶点v_i的最长路径长度；
- **v*l*(i)**：表示事件v_i的最晚发生时间。则有以下关系：

$$\begin{cases} e(i)=ve(j) \\ l(i)= vl(k)-dut(<j, k>) \end{cases} \quad 7\text{-}1$$

$$ve(j)=\begin{cases} 0 & j=0，表示v_j是起点 \\ Max\{ve(i)+dut(<i, j>)|<v_i, v_j>是网中的弧\} \end{cases} \quad 7\text{-}2$$

含义是：源点事件的最早发生时间设为0；除源点外，只有进入顶点v_j的所有弧所代表的活动全部结束后，事件v_j才能发生。即只有v_j的所有前驱事件v_i的最早发生时间ve(i)计算出来后，才能计算ve(j)。

方法是：对所有事件进行拓扑排序，然后依次按拓扑顺序计算每个事件的最早发生时间。

$$vl(j)=\begin{cases} ve(n-1) & j=n-1，表示v_j是终点 \\ Min\{vl(k)-dut(<j,k>)|<v_j,v_k>是网中的弧\} \end{cases} \quad 7\text{-}3$$

含义是：只有v_j的所有后继事件v_k的最晚发生时间vl(k)计算出来后，才能计算vl(j)。

方法是：按拓扑排序的逆顺序，依次计算每个事件的最晚发生时间。

2 求AOE中关键路径和关键活动

(1) 算法思想

① 利用拓扑排序求出AOE网的一个拓扑序列；

② 从拓扑排序的序列的第一个顶点(源点)开始，按拓扑顺序依次计算每个事件的最早发生时间ve(i)；

③ 从拓扑排序的序列的最后一个顶点(汇点)开始，按逆拓扑顺序依次计算每个事件的最晚发生时间v/(i)；

对于图7-24的AOE网，处理过程如下：

- 拓扑排序的序列是：(v_0，v_1，v_2，v_3，v_4，v_5，v_6，v_7，v_8)

- 根据计算ve(i)的公式(7-2)和计算vl(i)的公式(7-3)，计算各个事件的ve(i)和vl(i)值，如表7-2所示。
- 根据关键路径的定义，知该AOE网的关键路径是：（v_0，v_2，v_4，v_7，v_8）和（v_0，v_2，v_5，v_7，v_8）。
- 关键路径活动是：<v_0，v_2>，<v_2，v_4>，<v_2，v_5>，<v_4，v_7>，<v_5，v_7>，<v_5，v_8>。

顶点	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
ve(i)	0	3	10	12	22	17	20	28	33
vl(i)	0	9	10	23	22	17	31	28	33

表7-2 图7-24的ve(i)和vl(i)的值

(2) 算法实现

```
void critical_path(ALGraph *G)
   {  int j, k, m ; LinkNode *p ;
      if  (Topologic_Sort(G)==-1)
          printf("\nAOE网中存在回路，错误!!\n\n") ;
      else
          {  for ( j=0; j<G->vexnum; j++)
                 ve[j]=0 ;     /*  事件最早发生时间初始化  */
             for (m=0 ; m<G->vexnum; m++)
                 {   j=topol[m] ;
                     p=G->adjlist[j].firstarc ;
                     for (; p!=NULL; p=p->nextarc )
```

```
            {  k=p->adjvex ;
               if  (ve[j]+p->weight>ve[k])
                    ve[k]=ve[j]+p->weight ;
            }
    }    /* 计算每个事件的最早发生时间ve值  */
for ( j=0; j<G->vexnum; j++)
    vl[j]=ve[j] ;    /*  事件最晚发生时间初始化  */
for (m=G->vexnum-1; m>=0; m--)
    {   j=topol[m] ; p=G->adjlist[j].firstarc ;
        for (; p!=NULL; p=p->nextarc )
            {  k=p->adjvex ;
               if  (vl[k]-p->weight<vl[j])
                    vl[j]=vl[k]-p->weight ;
```

```
            }
        }   /* 计算每个事件的最晚发生时间vl值 */
    for (m=0 ; m<G->vexnum; m++)
        {   p=G->adjlist[m].firstarc ;
            for (; p!=NULL; p=p->nextarc )
                {  k=p->adjvex ;
                   if  ( (ve[m]+p->weight)==vl[k])
                       printf("<%d, %d>, m, j") ;
                }
        }    /* 输出所有的关键活动 */
    }    /*  end of else  */
}
```

(3) 算法分析

设AOE网有n个事件，e个活动，则算法的主要执行是：

- 进行拓扑排序：时间复杂度是O(n+e)；
- 求每个事件的ve值和vl值：时间复杂度是O(n+e)；
- 根据ve值和vl值找关键活动：时间复杂度是O(n+e)；

因此，整个算法的时间复杂度是O(n+e)。

7.7 最短路径

若用带权图表示交通网，图中顶点表示地点，边代表两地之间有直接道路，边上的权值表示路程(或所花费用或时间)。从一个地方到另一个地方的路径长度表示该路径上各边的权值之和。问题：

- 两地之间是否有通路?
- 在有多条通路的情况下，哪条最短?

考虑到交通网的有向性，直接讨论的是带权有向图的最短路径问题，但解决问题的算法也适用于无向图。

将一个路径的起始顶点称为源点，最后一个顶点称为终点。

7.7.1 单源点最短路径

对于给定的有向图G=(V，E)及单个源点V_s，求V_s到G的其余各顶点的最短路径。

针对单源点的最短路径问题，Dijkstra提出了一种按路径长度递增次序产生最短路径的算法，即迪杰斯特拉(Dijkstra)算法。

1 基本思想

从图的给定源点到其它各个顶点之间客观上应存在一条最短路径，在这组最短路径中，按其长度的递增次序，依次求出到不同顶点的最短路径和路径长度。

即按长度递增的次序生成各顶点的最短路径，即先求出长度最小的一条最短路径，然后求出长度第二小的最短路径，依此类推，直到求出长度最长的最短路径。

2 算法思想说明

设给定源点为V_s，S为已求得最短路径的终点集，开始时令$S=\{V_s\}$。当求得第一条最短路径(V_s, V_i)后，S为$\{V_s, V_i\}$。根据以下结论可求下一条最短路径。

设下一条最短路径终点为V_j，则V_j只有：

- 源点到终点有直接的弧$<V_s, V_j>$；
- 从V_s出发到V_j的这条最短路径所经过的所有中间顶点必定在S中。即只有这条最短路径的最后一条弧才是从S内某个顶点连接到S外的顶点V_j。

若定义一个数组dist[n]，其每个dist[i]分量保存从V_s出发中间只经过集合S中的顶点而到达V_i的所有路径中长度最小的路径长度值，则下一条最短路径的终点V_j必定是不在S中且值最小的顶点，即：

$$dist[i]=Min\{\ dist[k]\mid V_k\in V-S\ \}$$

利用上述公式就可以依次找出下一条最短路径。

3 算法步骤

① 令S={Vs}，用带权的邻接矩阵表示有向图，对图中每个顶点Vi按以下原则置初值：

$$dist[i]=\begin{cases}0 & i=s\\ W_{si} & i\neq s\text{且}<v_s,v_i>\in E,\ w_{si}\text{为弧上的权值}\\ \infty & i\neq s\text{且}<v_s,v_i>\text{不属于}E\end{cases}$$

② 选择一个顶点V_j，使得：

$dist[j]=Min\{\ dist[k]\mid V_k\in V\text{-}S\ \}$

V_j就是求得的下一条最短路径终点，将V_j并入到S中，即$S=S\cup\{V_j\}$。

③ 对V-S中的每个顶点V_k，修改dist[k]，方法是：

若$dist[j]+W_{jk}<dist[k]$，则修改为：

$dist[k]=dist[j]+W_{jk}\ (\forall V_k\in V\text{-}S\ )$

④ 重复②，③，直到S=V为止。

4 算法实现

用带权的邻接矩阵表示有向图， 对Prim算法略加改动就成了Dijkstra算法，将Prim算法中求每个顶点V_k的lowcost值用dist[k]代替即可。

- 设数组pre[n]保存从V_s到其它顶点的最短路径。若pre[i]=k，表示从V_s到V_i的最短路径中，V_i的前一个顶点是V_k，即最短路径序列是(V_s，…，V_k，V_i)。
- 设数组final[n]，标识一个顶点是否已加入S中。

算法实现的关键

待求点的最短路径长度本身就是待求的，又如何找出其中的最短呢？

```
BOOLEAN  final[MAX_VEX] ;
int  pre[MAX_VEX] , dist[MAX_VEX] ;
void Dijkstra_path (AdjGraph *G, int v)
        /*  从图G中的顶点v出发到其余各顶点的最短路径   */
   {  int j, k, m, min ;
      for ( j=0; j<G->vexnum; j++)
         {   pre[j]=v ;  final[j]=FALSE ;
             dist[j]=G->adj[v][j] ;
         }    /*  各数组的初始化  */
      dist[v]=0 ; final[v]=TRUE ;     /*  设置S={v}  */
      for ( j=0; j<G->vexnum-1; j++) /*  其余n-1个顶点  */
         {   m=0 ;
```

```
while (final[m])  m++;   /*  找不在S中的顶点v_k */
min=INFINITY ;
for ( k=0; k<G->vexnum; k++)
    {  if  (!final[k]&&dist[m]<min)
            {   min=dist[k] ;  m=k ; }
    }     /* 求出当前最小的dist[k]值  */
final[m]=TRUE ;      /*  将第k个顶点并入S中  */
for ( j=0; j<G->vexnum; j++)
    { if (!final[j]&&(dist[m]+G-
>adj[m][j]<dist[j]))
          {   dist[j]=dist[m]+G->adj[m][j] ;
              pre[j]=m ;
          }
     }     /* 修改dist和pre数组的值  */
```

```
        }     /* 找到最短路径 */
    }
```

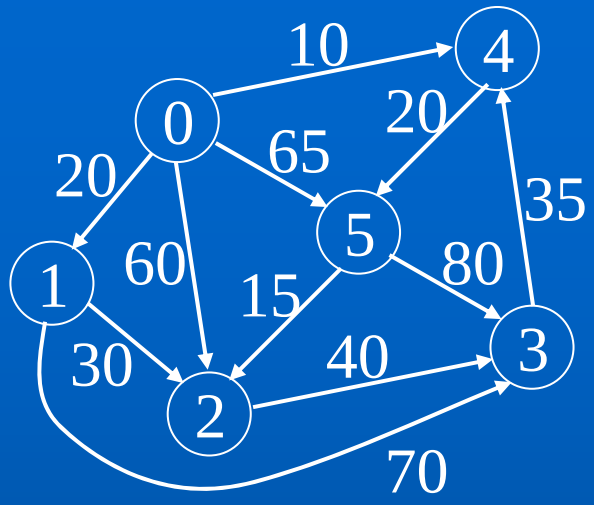
5 算法分析

Dijkstra算法的主要执行是：

- 数组变量的初始化：时间复杂度是**O(n)**；
- 求最短路径的二重循环：时间复杂度是**O(n2)**；

因此，整个算法的时间复杂度是**O(n2)**。

对图**7-25**的带权有向图，用**Dijkstra**算法求从顶点**0**到其余各顶点的最短路径，数组**dist**和**pre**的各分量的变化如表**7-3**所示。



$$\begin{pmatrix} \infty & 20 & 60 & \infty & 10 & 65 \\ \infty & \infty & 30 & 70 & \infty & \infty \\ \infty & \infty & \infty & 40 & \infty & \infty \\ \infty & \infty & \infty & \infty & 35 & \infty \\ \infty & \infty & \infty & \infty & \infty & 20 \\ \infty & \infty & 15 & 80 & \infty & \infty \end{pmatrix}$$

图**7-25** 带权有向图及其邻接矩阵

表7-3　求最短路径时数组dist和pre的各分量的变化情况

顶点 / 步骤		1	2	3	4	5	S
初态	Dist	20	60	∞	10	65	{0}
	pre	0	0	0	0	0	
1	Dist	20	60	∞	10	30	{0, 4}
	pre	0	0	0	0	4	
2	Dist	20	50	90	10	30	{0, 4, 1}
	pre	0	1	1	0	4	
3	Dist	20	45	90	10	30	{0, 4, 1, 5}
	pre	0	5	1	0	4	
4	Dist	20	45	85	10	30	{0, 4, 1, 5, 2}
	pre	0	5	2	0	4	
5	Dist	20	45	85	10	30	{0, 4, 1, 5, 2, 3}
	pre	0	5	2	0	4	

7.7.2 每一对顶点间的最短路径

用Dijkstra算法也可以求得有向图G=(V，E)中每一对顶点间的最短路径。方法是：每次以一个不同的顶点为源点重复Dijkstra算法便可求得每一对顶点间的最短路径，时间复杂度是$O(n^3)$。

弗罗伊德(Floyd)提出了另一个算法，其时间复杂度仍是$O(n^3)$，但算法形式更为简明，步骤更为简单，数据结构仍然是基于图的邻接矩阵。

1 算法思想

设顶点集S(初值为空)，用数组A的每个元素A[i][j]保存从V_i只经过S中的顶点到达V_j的最短路径长度，其思想是：

① 初始时令S={ }， A[i][j]的赋初值方式是：

$$A[i][j]=\begin{cases}0 & i=j\text{时}\\ W_{ij} & i\neq j\text{且}<v_i,v_j>\in E,\ w_{ij}\text{为弧上的权值}\\ \infty & i\neq j\text{且}<v_i,v_j>\text{不属于}E\end{cases}$$

② 将图中一个顶点V_k加入到S中，修改A[i][j]的值，修改方法是：

A[i][j]=Min{A[i][j] ,
(A[i][k]+A[k][j]) }

原因： 从V_j只经过S中的顶点(V_k)到达V_j的路径长度可能比原来不经过V_k的路径更短。

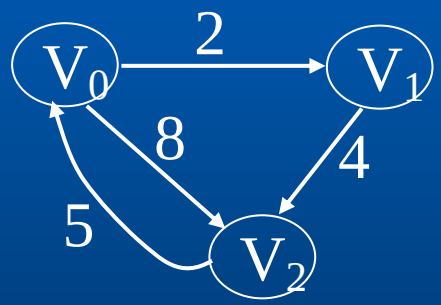
③ 重复②，直到G的所有顶点都加入到S中为止。

2 算法实现

- 定义二维数组Path[n][n](n为图的顶点数)，元素Path[i][j]保存从V_i到V_j的最短路径所经过的顶点。
- 若Path[i][j]=k：从V_i到V_j 经过V_k，最短路径序列是(V_i，…，V_k，…，V_j)，则路径子序列：(V_i，…，V_k)和(V_k，…，V_j)一定是从V_i到V_k和从V_k到V_j 的最短路径。从而可以根据Path[i][k]和Path[k][j]的值再找到该路径上所经过的其它顶点，…依此类推。

◆ 初始化为Path[i][j]=-1，表示从V_i到V_j不经过任何(S中的中间)顶点。当某个顶点V_k加入到S中后使A[i][j]变小时，令Path[i][j]=k。

表7-4给出了利用Floyd算法求图7-26的带权有向图的任意一对顶点间最短路径的过程。



$$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$$

图7-26 带权有向图及其邻接矩阵

表7-4 用Floyd算法求任意一对顶点间最短路径

步骤	初态	k=0	K=1	K=2
A	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ 9 & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$
Path	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ 2 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$
S	{ }	{ 0 }	{ 0, 1 }	{ 0, 1, 2 }

根据上述过程中Path[i][j]数组，得出：

V_0到V_1： 最短路径是{ 0, 1 }， 路径长度是2；

V_0到V_2： 最短路径是{ 0, 1, 2 }， 路径长度是6；

V_1到V_0： 最短路径是{ 1, 2, 0 }， 路径长度是9；

V_1到V_2：最短路径是{ 1, 2 }，路径长度是4；

V_2到V_0：最短路径是{ 2, 0 }，路径长度是5；

V_2到V_1：最短路径是{ 2, 0, 1 }，路径长度是7；

算法实现

```
int  A[MAX_VEX][MAX_VEX];
int  Path[MAX_VEX][MAX_VEX];
void Floyd_path (AdjGraph *G)
  {  int j, k, m ;
    for ( j=0; j<G->vexnum; j++)
      for ( k=0; k<G->vexnum; k++)
        {  A[j][k]=G->adj[j][k];
          Path[j][k]=-1 ;  }
            /*  各数组的初始化  */
```

```
for ( m=0; m<G->vexnum; m++)
    for ( j=0; j<G->vexnum; j++)
        for ( k=0; k<G->vexnum; k++)
            if ((A[j][m]+A[m][k])<A[j][k])
                {  A[j][k]=A[j][m]+A[m][k] ;
                   Path[j][k]=k ;
                }    /*  修改数组A和Path的元素值  */
for ( j=0; j<G->vexnum; j++)
    for ( k=0; k<G->vexnum; k++)
        if (j!=k)
            {  printf("%d到%d的最短路径为:\n", j, k) ;
               printf("%d  ",j) ; prn_pass(j, k) ;
               printf("%d  ", k) ;
```

```
                printf("最短路径长度为: %d\n",A[j][k]) ;
            }
    }     /*   end of Floyd  */

void  prn_pass(int j , int k)
    {   if (Path[j][k]!=-1)
          {  prn_pass(j, Path[j][k]) ;
             printf(", %d" , Path[j][k]) ;
             prn_pass(Path[j][k],  k) ;
          }
    }
```

习 题 七

⑴ 分析并回答下列问题：

① 图中顶点的度之和与边数之和的关系？

② 有向图中顶点的入度之和与出度之和的关系？

③ 具有n个顶点的无向图，至少应有多少条边才能确保是一个连通图？若采用邻接矩阵表示，则该矩阵的大小是多少？

④ 具有n个顶点的有向图，至少应有多少条弧才能确保是强连通图的？为什么？

⑵ 设一有向图G=(V,E)，其中V={a,b,c,d,e}，E={<a,b>, <a,d>, <b,a>, <c,b>, <c,d>, <d,e>,<e,a>, <e,b>, <e,c>}

① 请画出该有向图，并求各顶点的入度和出度。

② 分别画出有向图的正邻接链表和逆邻接链表。

(3) 对图7-27所示的带权无向图。

① 写出相应的邻接矩阵表示。

② 写出相应的边表表示。

③ 求出各顶点的度。

(4) 已知有向图的逆邻接链表如图7-28所示。

① 画出该有向图。

② 写出相应的邻接矩阵表示。

③ 写出从顶点a开始的深度优先和广度优先遍历序列。

④ 画出从顶点a开始的深度优先和广度优先生成树。

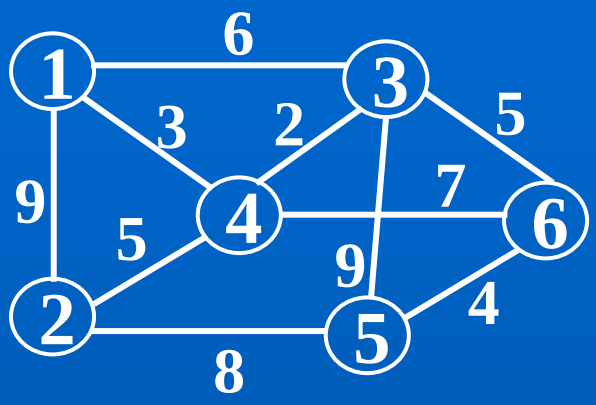


图7-27 带权无向图

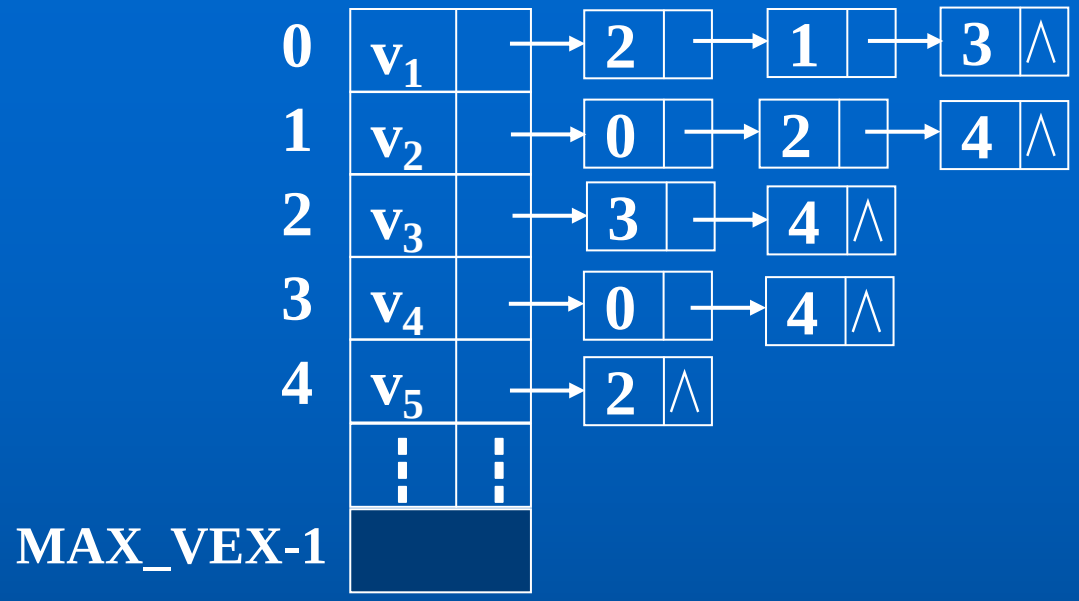


图7-28 有向图的逆邻接链表

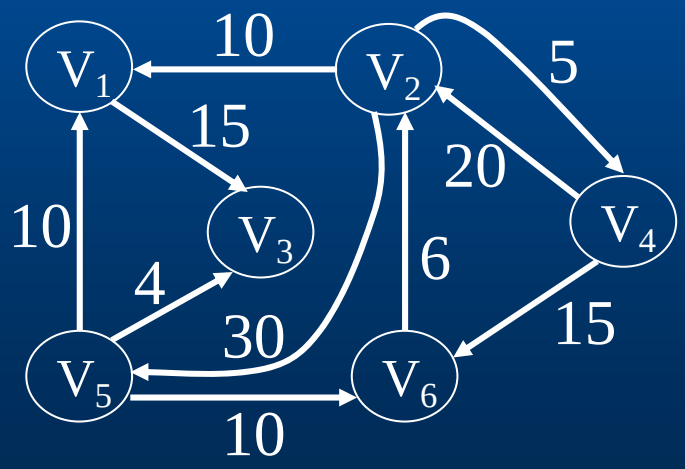


图7-29 带权有向图

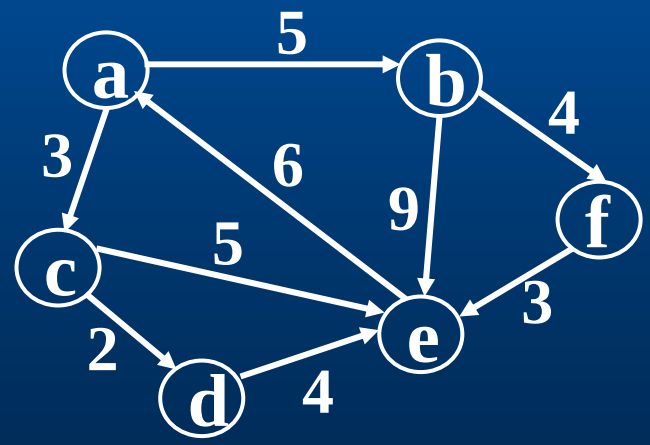


图7-30 带权有向图

(5) 一个带权连通图的最小生成树是否唯一?在什么情况下可能不唯一?

(6) 对于图7-27所示的带权无向图。

① 按照Prime算法给出从顶点2开始构造最小生成树的过程。

② 按照Kruskal算法给出最小生成树的过程。

(7) 已知带权有向图如图7-29所示，请利用Dijkstra算法从顶点V_4出发到其余顶点的最短路径及长度，给出相应的求解步骤。

(8) 已知带权有向图如图7-30所示，请利用Floyd算法求出每对顶点之间的最短路径及路径长度。

(9) 一个AOV网用邻接矩阵表示，如图7-31。用拓扑排序求该AOV网的一个拓扑序列，给出相应的步骤

⑽ 拓扑排序的结果不是唯一的，请给出如图7-32所示的有向图的所有可能的拓扑序列。

⑾ 请在深度优先搜索算法的基础上设计一个对有向无环图进行拓扑排序的算法。

⑿ 设计一个算法利用图的遍历方法输出一个无向图G中从顶点V_i到V_j的长度为S的简单路径，设图采用邻接链表作为存储结构。

⒀ 假设一个工程的进度计划用AOE网表示，如图7-33所示。

① 求出每个事件的最早发生时间和最晚发生时间。

② 该工程完工至少需要多少时间?

③ 求出所有关键路径和关键活动。

V_0	0	1	1	0	0	0	0
V_1	0	0	0	1	1	1	0
V_2	0	0	0	0	1	0	1
V_3	0	0	0	0	0	0	0
V_4	0	0	0	0	0	0	1
V_5	0	0	0	0	0	0	1
V_6	0	0	0	0	0	0	0

图7-31 一个AOV网的邻接矩阵

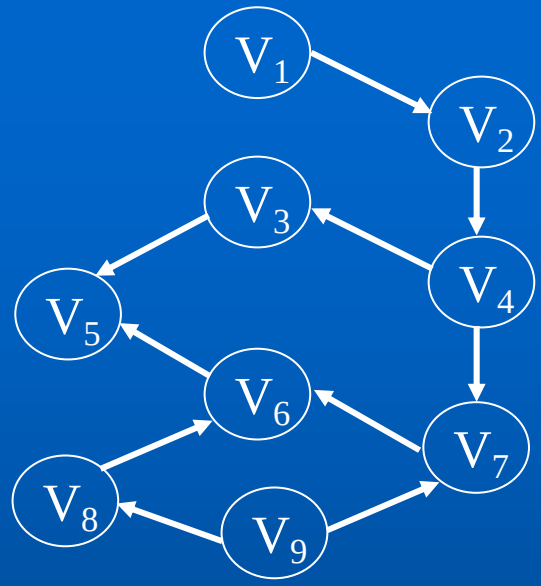


图7-32 有向图

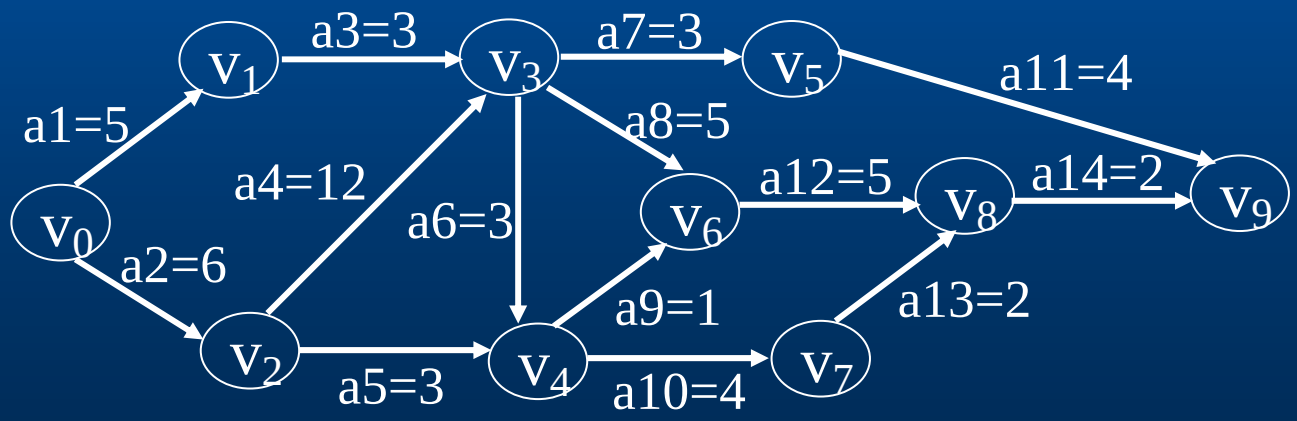


图7-33 一个AOE网

第8章 动态存储管理

8.1 概述

程序执行过程中，(数据)结构中的每一个数据元素都对应一定的存储空间，数据元素的访问都是通过对应的存储单元来进行的。存储空间的分配与管理是由操作系统或编译程序负责实现的，是一个复杂而又重要的问题，现代的存储管理往往采用动态存储管理思想。

动态存储管理：如何根据“存储请求”分配内存空间？如何回收被释放的(或不再使用的)内存空间？

对于允许进行动态存储分配的程序设计语言，操作系统在内存中划出一块地址连续的大区域(称为堆)，由设计者在程序中利用语言提供的内存动态分配函数(如C的malloc()，calloc()，free()函数，C++的new，delete函数等)来实现对堆的使用。

1 两个基本概念

- 占用块：已分配给用户使用的一块地址连续的内存区域；
- 空闲块：未曾分配的地址连续的内存区域；

2 用户请求分配内存，系统的处理方式

当有用户程序进入系统请求分配内存时，系统有两种处理方式：

(1) 系统从高地址空闲块中进行分配，直到分配无法进行时，才回收所有用户不再使用的空闲块，重新组织一个大的空闲块来再分配；

(2) 用户程序一旦运行结束，便将它所占内存区释放成为空闲块，同时，每当新用户请求分配内存时，系统需要巡视整个内存区中所有空闲块，并从中找出一个“合适”的空闲块分配之。

对于(2)的情况，系统需建立一张“可利用空间表”。

程序运行过程中，不断地对堆中的部分区域进行分配和释放，堆中会出现占用块和空闲块交错的状态，如图8-1所示。

3 动态存储分配的基本问题

(1) 当某一时刻用户程序请求分配400个字节的存储空间，如何分配?

- 将块A分配给用户程序?
- 从大块C中划出一部分分配给用户程序?

(2) 当某一时刻分配B块的用户程序运行结束，B块要进行回收，如何回收?

- B块直接回收并成为一个独立的空闲块?
- B块回收并和前、后的空闲块A、C合并后形成一个更大的空闲块?

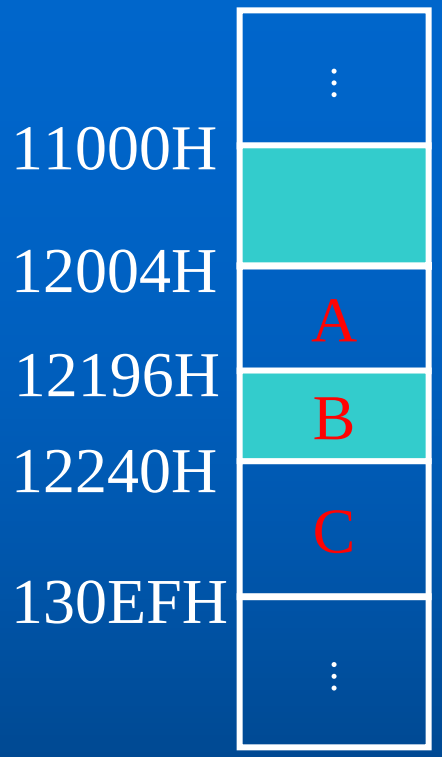


图8-1 堆的状态

8.2 可利用空间表及分配方法

可利用空间表中包含所有可分配的空闲块，当用户请求分配时，系统从可利用空间表中删除一个结点分配之；当用户释放其所占内存时，系统即回收并将它插入到可利用空间表中。因此，可利用空间表亦称做“存储池”。

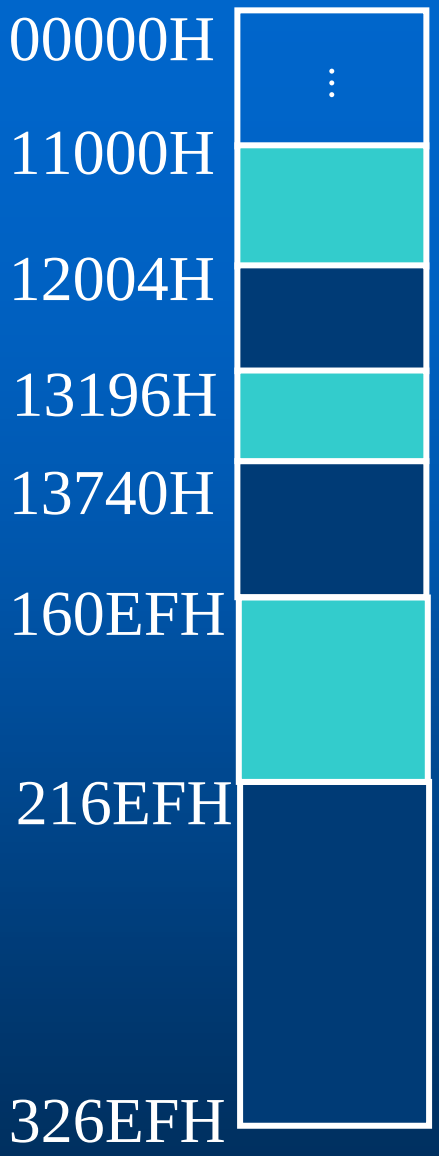
8.2.1 可利用空间表的组织

可用空间表的组织有两种方式：目录表方式和链表方式，如图8-2所示 。动态存储管理中需要不断地进行空闲块的分配和释放，对目录表来说管理复杂，因此，可利用空间表通常以链表方式组织 。

当可利用空间表以链表方式组织时，每个空闲块就是链表中的一个结点。

- 分配时：从链表中找到一个合适的结点加以分配，然后将该结点删除之；
- 回收时：将空闲块插入到链表中。

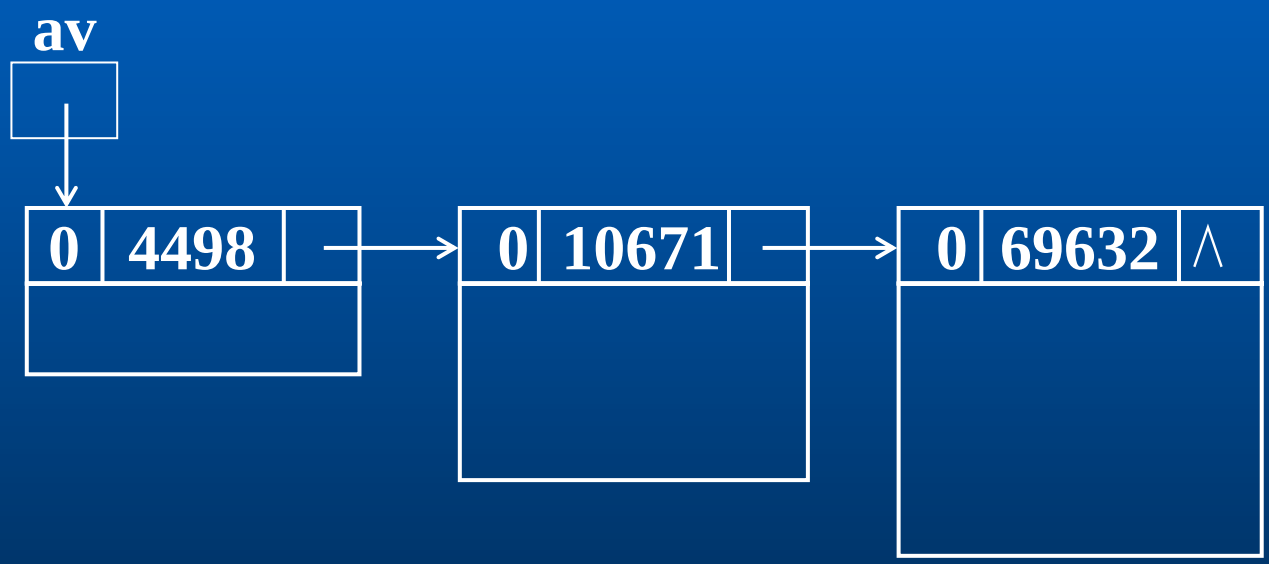
实际的动态存储管理实施时，具体的分配和释放的策略取决于结点(空闲块)的结构。



(a) 堆的状态

起始地址	空闲块大小	使用情况
12004H	4498	空闲
13740H	10671	空闲
216EFH	69632	空闲

(b) 目录表方式



(c) 链表方式

图8-2 动态存储管理过程中的内存状态和空闲表结构

8.2.2 结点结构方式与分配策略

1 请求分配的块大小相同

将进行动态存储分配的整个内存区域(堆)按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。

- 分配时：从表的首结点分配，然后删除该结点；
- 回收时：将释放的空闲块插入表头。

2 请求分配的块大小只有几种规格

根据统计概率事先对动态分配的堆建立若干个可利用空间链表，同一链表中的结点(块)大小都相同。

- 分配时：根据请求的大小，将最接近该大小的某个链表的首结点分配给用户。若剩余部分正好差不多是另一种规格大小，则将剩余部分插入到另一种规格的链表中，然后删除该结点；
- 回收时：只要将所释放的空闲块插入到相应大小的表头。

存在的问题：

当请求分配的块空间大小比最大规格的结点还大时，分配不能进行。而实际上内存空间却可能存在比所需大小还要大的的连续空间，应该能够分配。

3　请求分配的块大小不确定

系统开始时，整个堆空间是一个空闲块，链表中只有一个大小为整个堆的结点，随着分配和回收的进行，链表中的结点大小和个数动态变化。

由于链表中结点大小不同，结点中除标志域和链域之外，尚需有一个结点大小域(size)，以保存空闲块的大小，如图8-2(b)。

问题：若用户请求分配大小为n(kB)的内存，而链表中有若干大小不小于n的空闲块时，如何分配？有3种分配策略。

(1) 首次拟合法(First fit)

- 分配时：从表头指针开始查找可利用空间表，将找到的第一个不小于n的空闲块的部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；
- 回收时：将释放的空闲块插入在链表的表头。

特点：分配时随机的；回收时仅需插入到表头。

(2) 最佳拟合法(Best fit)

- 分配时：扫描整个可利用空间链表，找到一个大小满足要求且最接近n空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；

◆ 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从小到大排序(升序) 。

优点：适用于请求分配的内存块大小范围较广的系统；

缺点：系统容易产生无法分配的内存碎片；无论分配与回收，都需要查找表，最费时；

(3) 最差拟合法(Worst fit)

◆ 分配时：扫描整个可利用空间链表，找到一个大小最大的空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；

◆ 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从大到小排序(降序）。

特点：适用于请求分配的内存块的大小范围较窄的系统；分配无需查找，回收需要查找适当的位置。

4 选择分配策略需考虑的因素

用户的逻辑要求、请求分配量的大小分布、分配和释放的频率以及效率对系统的重要性。

8.3 边界标识法

边界标识法(Boundary Tag Method)是操作系统中一种常用的进行动态分配的存储管理方法。

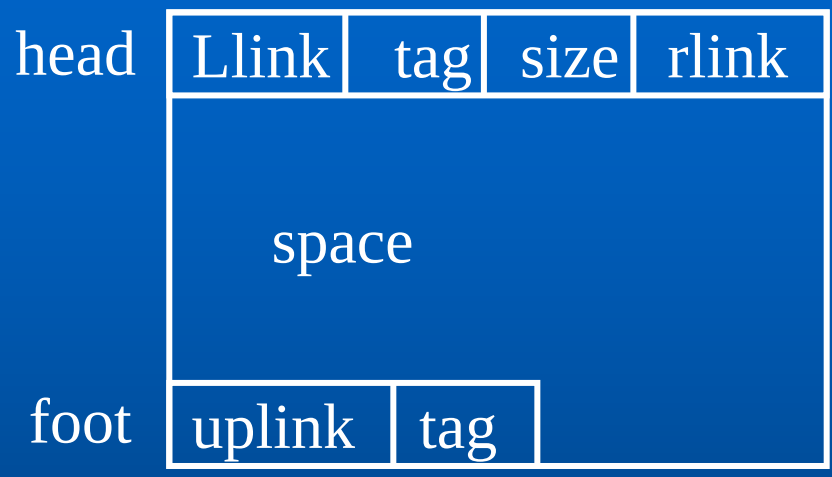
系统将所有的空闲块链接成一个双重循环链表，分配可采用几种方法(前述)。

系统的特点

每个内存区域的头部和底部两个边界上分别设置标识，以标识该区域为占用块或空闲块，在回收块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便于将所有地址连续的空闲存储区合并成一个尽可能大的空闲块。

8.3.1 可利用空闲表结点结构

```
typedef  struct word
   {  Union
      {  struct word *llink;
         struct word *uplink;
      };
      int tag;
      int size;
      struct word  *rlink;
      OtherType other;
   }WORD, head, foot, *Space;
#define FootLoc(p) p+p->size-1
```



8.3.2 分配算法

分配算法比较简单，可采用前述三种方法中的任一种进行分配。设采用首次拟合法，为了使系统更有效地运行，在边界标识法中还做了两条约定：

① 选定适当常量e，设待分配空闲块、请求分配空间的大小分别为m 、 n 。

- 当m-n≤e时：将整个空闲块分配给用户；
- 当m-n>e时：则只分配请求的大小n给用户；

作用：尽量减少空闲块链表中出现小碎片(容量≤e)，提高分配效率；减少对空闲块链表的维护工作量。为了避免修改指针，约定将高地址部分分配给用户。

② 空闲块链表中的结点数可能很多，为了提高查找空闲块的速度和防止小容量结点密集，每次查找时从不同的结点开始——上次刚分配结点的后继结点开始。

```
Space AllocBoundTag( Space  *pav, int n )
  {  p = pav ;
    for  ( ; p &&p->size<n && p ->rlink!=pav; p=p->rlink )
      if ( !p || p->size<n ) return NULL ;
      else
        {  f=FootLoc( p ) ; Pav=p->rlink ;
          if  ( p->size–n<=e )
```

```
            {  if ( pav==p ) pav=NULL ;
               else
                  {  pav->llink=p->link ;
                     p->llink->rlink=pav ;  }
               p->tag=f->tag=1 ;
            }
         else
           {  f->tag=1 ; p->size-=n ; f=
       FootLoc( p ) ;
            f->tag= 0 ; f->uplink=p ; p=f+1;
           p->tag=1 ; p->size=n ;
         }
       return p ;
    }
}
```

8.3.3 回收算法

当用户释放占用块，系统需立即回收以备新的请求产生时进行再分配。关键的是使物理地址毗邻的空闲块合并成一个尽可能大的结点，则需检查刚释放的占用块的左、右紧邻是否为空闲块。

假设所释放的块的头地址为p，则与其低地址紧邻的块的底部地址为p-1；与其高地址紧邻的块的头地址为p+p->size，它们中的标志域就表明了两个相邻块的使用状况：

- 若(p-1)->tag=0：则左邻块为空闲块；
- 若(p+p->size)->tag=0：则右邻块为空闲块；

回收算法需要考虑的4种情况：

(1) 释放块的左、右邻块均为占用块

将被释放块简单地插入到空闲块链表中即可。

```
p->tag=0；FootLoc(p)->uplink=p；
FootLoc(p)->tag=0；
if （!pav） pav=p->llink=p->rrlink=p；
else
   {  q=pav->llink； P->rlink=pav；
     p->llink=q；q->rlink=pav->llink=p；
     Pav=p；
   }
```

(2) 释放块的左邻块空闲而右邻块为占用

和左邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

```
n=p->size；s=(p-1)->uplink；s->size+=n;
f=p+n–1；f->uplink=s；f->tag=0；
```

(3) 释放块的左邻占用而右邻空闲

和右邻块合并成一个大的空闲块结点，改变右邻块的size域及重新设置(合并后)结点的头部。

```
t=p+p->size；p->tag=0；q=t->llink；p->llink=q；
q->rlink=p；q1=t->rlink；p->rlink=q1；
q1->llink=p；p->size+=t->size；
FootLoc(t)->uplink=p；
```

(4) 释放块的左、右邻块均为空闲块

和左、右邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

```
n=p->size；s=(p-1)->uplink；t=p+p->size；
s->size+=n+t->size；  q=t->llink；q1=t->rlink；
q->rlink=q1；q1->llink=q；
FootLoc(t)->uplink=s;
```

8.4 伙伴系统

伙伴系统是一种非顺序内存管理方法，不是以顺序片段来分配内存，是把内存分为两个部分，只要有可能，这两部分就可以合并在一起；且这两部分从来不是自由的，程序可以使用伙伴系统中的一部分或者两部分都不使用。与边界标识法类似，所不同是：无论占用块或空闲块，其大小均为2的k次幂。

8.4.1 可利用空间表的结构

为了再分配时查找方便起见，我们将所有大小相同的空闲块建于一张子表中。每个子表是一个双重链表，这样的链表可能有m+1个，将这m+1个表头指针用向量结构组织成一个表，这就是伙伴系统的可利用空间表。

可利用空间表的数据类型描述如下：

```
#define M 16
typedef struct WORD_b
  {  WORD_b * llink;    /*  前驱结点  */
    int   tag;        /*  使用标识  */
    int   kval;       /*  块的大小,是2的幂次  */
    WORD_b *rlink;    /*  后继结点  */
    OtherType   other;
  } WORD_b, head;
typedef struct HeadNode
  {  int   nodesize;
    WORD_b * first;
  }FreeList[M+1];
```

8.4.2 分配算法

当程序提出大小为n的内存分配请求时，首先在可利用表中查找大小与n相匹配的子表.

1 算法思想

- 若存在$2^{k-1}<n\leqslant 2^k-1$的空闲子表结点：则将子表中的任意一个结点分配之；
- 若不存在$2^{k-1}<n\leqslant 2^k-1$的空闲子表结点：则从结点大小为2^k的子表中找到一个空闲结点，将其中一半分配给程序，剩余的一半插入到结点大小为2^{k-1}的子表中。

2 说明

在进行大小为n($2^{k-i-1}<n\leqslant 2^{k-i}-1$，i=1,2,...,k-1) 的内存分配请求时，若所有小于2^k的子表均为空(没有空闲结点)，则同样需要从大小为2^k的子表中找到一个空闲结点，将其中2^{k-i}一小部分分配给用户，而将剩余部分分割成若干个结点分别插入对应的子表。

8.4.3 回收算法

当程序释放所占用的块时，系统将该新的空闲块插入到可利用空闲表中，需要考虑合并成大块问题。在伙伴系统中，只有“互为伙伴”的两个子块均空闲时才合并；即使有两个相邻且大小相同的空闲块，如果不是“互为伙伴”(从同一个大块中分裂出来的)也不合并。

1 伙伴空闲块的确定

设p是大小为2^k的空闲块的首地址，且p MOD 2^{k+1} ＝0，则首地址为p和p＋2^k的两个空闲块“互为伙伴”。

首地址为p大小为2^k的内存块的，其伙伴的首地址为：

$$buddy(p,k)=\begin{cases} p+2^k & \text{若 } p \text{ MOD } 2^{k+1}=0 \\ P-2^k & \text{若 } p \text{ MOD } 2^{k+1}=2^k \end{cases}$$

2 回收算法

设要回收的空闲块的首地址是p，其大小为2^k的，算法思想是：

(1) 判断其“互为伙伴”的两个空闲块是否为空：

若不为空，仅将要回收的空闲块直接插入到相应的子表中；否则转(2)；

(2) 按以下步骤进行空闲块的合并：

- 在相应子表中找到其伙伴并删除之；
- 合并两个空闲块；

(3) 重复(2)，直到合并后的空闲块的伙伴不是空闲块为止。

系统的特点：算法简单；速度快；但容易产生碎片。

第9章　查找

数据的组织和查找是大多数应用程序的核心，而查找是所有数据处理中最基本、最常用的操作。特别当查找的对象是一个庞大数量的数据集合中的元素时，查找的方法和效率就显得格外重要。

本章主要讨论顺序表、有序表、树表和哈希表查找的各种实现方法，以及相应查找方法在等概率情况下的平均查找长度。

9.1 查找的概念

查找表(Search Table)：相同类型的数据元素(对象)组成的集合，每个元素通常由若干数据项构成。

关键字(Key，码)：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素。若关键字能唯一标识一个数据元素，则关键字称为主关键字；将能标识若干个数据元素的关键字称为次关键字。

查找/检索(Searching)：根据给定的K值，在查找表中确定一个关键字等于给定值的记录或数据元素。

- 查找表中存在满足条件的记录：查找成功；结果：所查到的记录信息或记录在查找表中的位置。
- 查找表中不存在满足条件的记录：查找失败。

查找有两种基本形式：静态查找和动态查找。

静态查找(Static Search)：在查找时只对数据元素进行查询或检索，查找表称为静态查找表。

动态查找(Dynamic Search)：在实施查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

查找的对象是查找表，采用何种查找方法，首先取决于查找表的组织。查找表是记录的集合，而集合中的元素之间是一种完全松散的关系，因此，查找表是一种非常灵活的数据结构，可以用多种方式来存储。

根据存储结构的不同，查找方法可分为三大类：

① 顺序表和链表的查找：将给定的K值与查找表中记录的关键字逐个进行比较， 找到要查找的记录；

② 散列表的查找：根据给定的K值直接访问查找表，从而找到要查找的记录；

③ 索引查找表的查找：首先根据索引确定待查找记录所在的块，然后再从块中找到要查找的记录。

查找方法评价指标

查找过程中主要操作是关键字的比较，查找过程中关键字的平均比较次数(平均查找长度ASL：Average Search Length)作为衡量一个查找算法效率高低的标准。ASL定义为：

$$ASL=\sum_{i=1}^{n} P_i \times C_i \quad n\text{为查找表中记录个数} \qquad \sum_{i=1}^{n} P_i=1$$

其中：

P_i：查找第i个记录的概率，不失一般性，认为查找每个记录的概率相等，即

$P_1 = P_2 = \ldots = P_n = 1/n$；

C_i：查找第i个记录需要进行比较的次数。

一般地，认为记录的关键字是一些可以进行比较运算的类型，如整型、字符型、实型等，本章以后各节中讨论所涉及的关键字、数据元素等的类型描述如下：

典型的关键字类型说明是：

```
typedef float  KeyType;      /* 实型 */
typedef int  KeyType;        /* 整型 */
typedef char KeyType;        /* 字符串型 */
```

数据元素类型的定义是：

```
typedef struct RecType
  { KeyType key;        /* 关键字码 */
    ┆ /* 其他域 */
  }RecType;
```

对两个关键字的比较约定为如下带参数的宏定义：

```
  /* 对数值型关键字 */
#define EQ(a, b)  ((a)==(b))
#define LT(a, b)  ((a)<(b))
#define LQ(a, b)  ((a)<=(b))
  /* 对字符串型关键字 */
#define EQ(a, b)  (!strcmp((a), (b)) )
#define LT(a, b)  (strcmp((a), (b))<0 )
#define LQ(a, b)  (strcmp((a), (b))<=0 )
```

9.2 静态查找

静态查找表的抽象数据类型定义如下：

ADT Static_SearchTable{

数据对象**D**：**D**是具有相同特性的数据元素的集合，各个数据元素有唯一标识的关键字。

数据关系**R**：数据元素同属于一个集合。

基本操作**P**：

⋮

} ADT Static_SearchTable 详见p_{216}。

线性表是查找表最简单的一种组织方式，本节介绍几种主要的关于顺序存储结构的查找方法。

9.2.1 顺序查找(Sequential Search)

1 查找思想

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败。顺序表的类型定义如下：

```
#define MAX_SIZE 100
typedef struct SSTable
  { RecType elem[MAX_SIZE];   /* 顺序表 */
    int length;    /* 实际元素个数 */
  }SSTable;
```

```
int  Seq_Search(SSTable  ST , KeyType key)
   {   int p ;
       ST. elem[0].key=key ;    /*  设置监视哨兵,失败返回0  */
       for (p=ST.length; !EQ(ST. elem[p].key, key); p--)
       return(p) ;
   }
```

比较次数：

查找第n个元素：　1

……

查找第i个元素：　n-i+1

查找第1个元素：　n

查找失败：　n+1

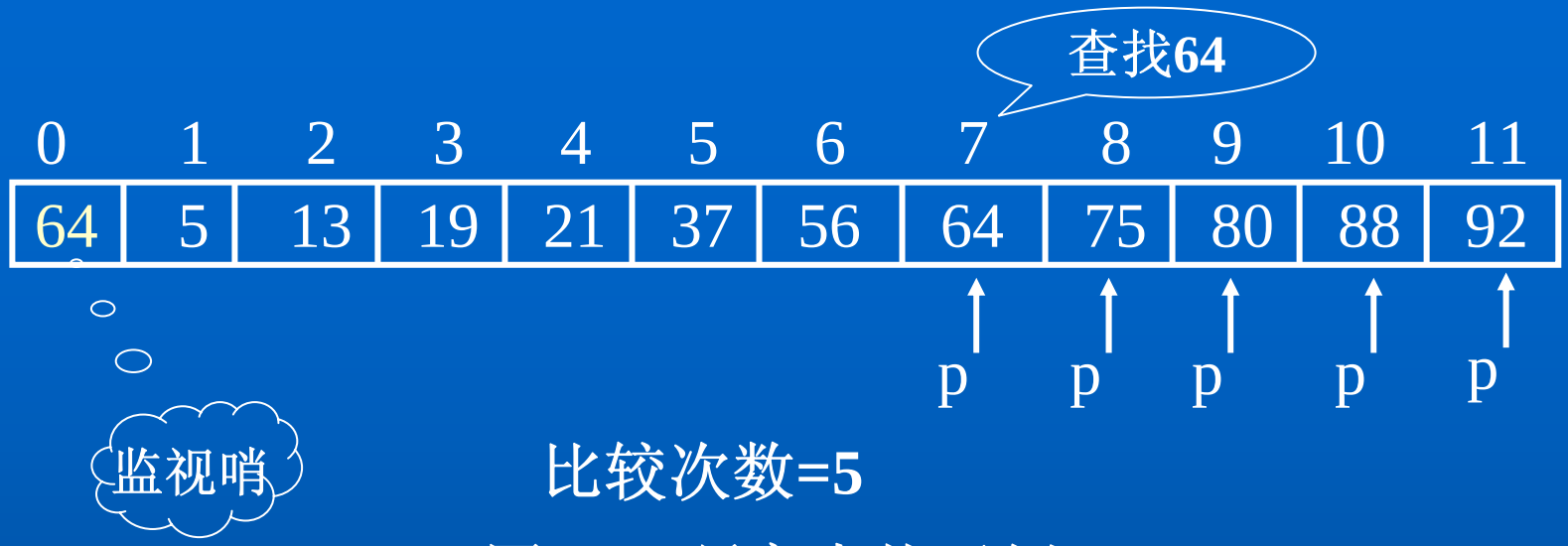


图9-1　顺序查找示例

2　算法分析

不失一般性，设查找每个记录成功的概率相等，即$P_i=1/n$；查找第i个元素成功的比较次数$C_i=n-i+1$；

- 查找成功时的平均查找长度ASL：

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

- 包含查找不成功时：查找失败的比较次数为n+1，若成功与不成功的概率相等，对每个记录的查找概率为$P_i=1/(2n)$，则平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{n+1}{2}=3(n+1)/4$$

9.2.2 折半查找(Binary Search)

折半查找又称为二分查找，是一种效率较高的查找方法。

前提条件：查找表中的所有记录是按关键字有序(升序或降序)。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

1 查找思想

用Low、High和Mid表示待查找区间的下界、上界和中间位置指针，初值为Low＝1，High＝n。

(1) 取中间位置Mid：Mid=⌊(Low+High)/2⌋；

(2) 比较中间位置记录的关键字与给定的K值：

① 相等： 查找成功；

② 大于：待查记录在区间的前半段，修改上界指针： High=Mid-1，转(1)；

③ 小于：待查记录在区间的后半段，修改下界指针：Low=Mid+1，转(1)；

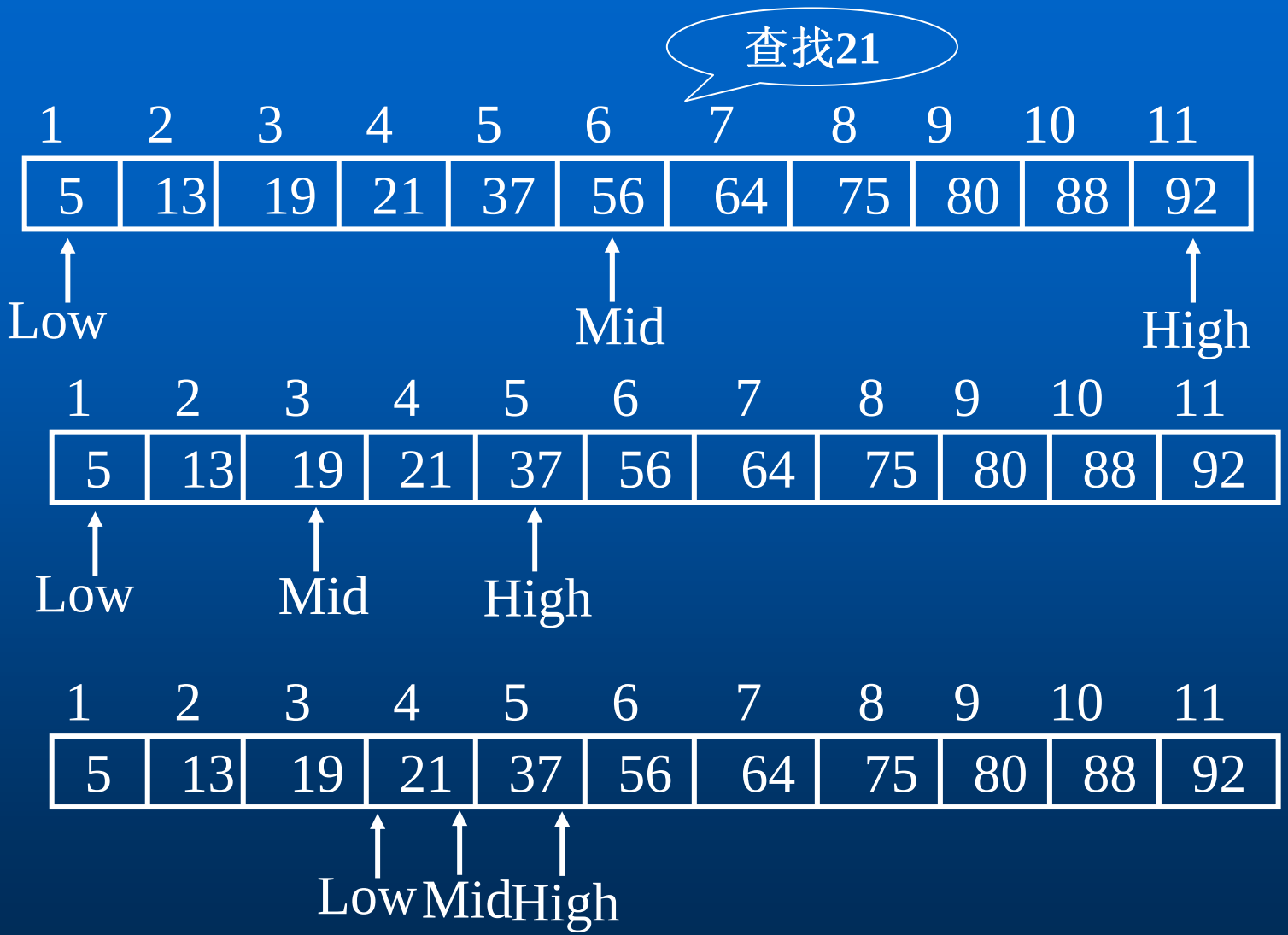
直到越界(Low>High)，查找失败。

2 算法实现

```
int Bin_Search(SSTable ST，KeyType key)
  {   int Low=1，High=ST.length, Mid；
    while (Low<High)
      {    Mid=(Low+High)/2；
        if (EQ(ST. elem[Mid].key, key))
           return(Mid)；
        else if (LT(ST. elem[Mid].key, key))
                Low=Mid+1；
             else  High=Mid-1；
      }
    return(0)；     /*  查找失败  */
  }
```

3 算法示例

如图9-2(a)，(b)所示。



(a) 查找成功示例

(b) 查找不成功示例

图**9-2** 折半查找示例

4 算法分析

① 查找时每经过一次比较，查找范围就缩小一半，该过程可用一棵二叉树表示：

- 根结点就是第一次进行比较的中间位置的记录；
- 排在中间位置前面的作为左子树的结点；
- 排在中间位置后面的作为右子树的结点；

对各子树来说都是相同的。这样所得到的二叉树称为判定树(Decision Tree)。

② 将二叉判定树的第$\lfloor \log_2 n \rfloor+1$层上的结点补齐就成为一棵满二叉树，深度不变，$h=\lfloor \log_2(n+1) \rfloor$。

③ 由满二叉树性质知，第i 层上的结点数为2^{i-1}(i≤h)，设表中每个记录的查找概率相等，即$P_i=1/n$，查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i\times C_i=\frac{1}{n}\sum_{j=1}^{h} j\times 2^{j-1}=\frac{n+1}{n}\log_2(n+1)-1$$

当n很大 (n>50)时， $ASL\approx \log_2(n+1)-1$。

9.2.3 分块查找

分块查找(Blocking Search)又称索引顺序查找，是前面两种查找方法的综合。

1 查找表的组织

① 将查找表分成几块。块间有序，即第i+1块的所有记录关键字均大于(或小于)第i块记录关键字；块内无序。

② 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

最大关键字
起始指针

2 查找思想

先确定待查记录所在块，再在块内查找(顺序查找)。

3 算法实现

```
typedef struct IndexType
  { keyType  maxkey;     /* 块中最大的关键字 */
    int startpos;     /* 块的起始位置指针  */
  }Index;
```

```
int Block_search(RecType ST[] , Index ind[] , KeyType key , int n , int b)
    /*  在分块索引表中查找关键字为key的记录 */
    /*表长为n ， 块数为b */
   {   int i=0 , j , k ;
       while ((i<b)&&LT(ind[i].maxkey, key) )  i++  ;
          if (i>b) {  printf("\nNot found");   return(0);  }
       j=ind[i].startpos ;
       while ((j<n)&&LQ(ST[j].key, ind[i].maxkey) )
          {   if ( EQ(ST[j].key, key) )  break ;
              j++ ;
          }    /*  在块内查找  */
```

```
    if (j>n||!EQ(ST[j].key, key) )
        {  j=0; printf("\nNot found");  }
    return(j);
}
```

4 算法示例

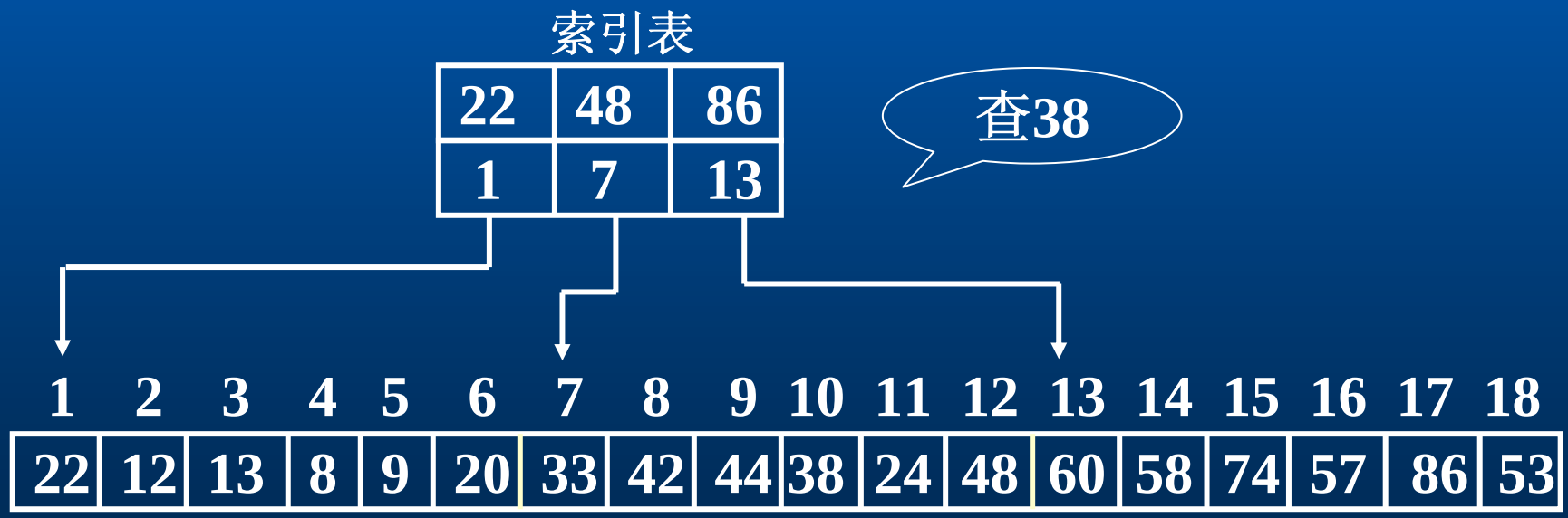


图9-3 分块查找示例

5 算法分析

设表长为n个记录，均分为b块，每块记录数为s，则b=⌈n/s⌉。设记录的查找概率相等，每块的查找概率为1/b，块中记录的查找概率为1/s，则平均查找长度ASL：

$$ASL=L_b+L_w=\sum_{j=1}^{b} j+\frac{1}{s}\sum_{i=1}^{s} i=\frac{b+1}{2}+\frac{s+1}{2}$$

9.2.4 Fibonacci查找

Fibonacci查找方法是根据Fibonacci数列的特点对查找表进行分割。Fibonacci数列的定义是：

F(0)=0，F(1)=1，F(j)=F(j-1)+F(j-2)

。

1 查找思想

设查找表中的记录数比某个Fibonacci数小1，即设n=F(j)-1。用Low、High和Mid表示待查找区间的下界、上界和分割位置，初值为Low=1，High=n。

⑴ 取分割位置Mid：Mid=F(j-1)；

⑵ 比较分割位置记录的关键字与给定的K值：

① 相等： 查找成功；

② 大于：待查记录在区间的前半段(区间长度为F(j-1)-1)，修改上界指针： High=Mid-1，转(1)；

③ 小于：待查记录在区间的后半段(区间长度为F(j-2)-1)，修改下界指针：Low=Mid+1，转(1)；

直到越界(Low>High)，查找失败。

2 算法实现

在算法实现时，为了避免频繁计算Fibonacci数，可用两个变量f1和f2保存当前相邻的两个Fibonacci数，这样在以后的计算中可以依次递推计算出。

```
int fib(int n)
    {    int i, f , f0=0 , f1=1 ;
         if (n==0)  return(0) ;
         if (n==1)  return(1) ;
         for (i=2 ; i<=n ; i++ )
             {   f=f0+f1 ; f0=f1 ; f1=f ;   }
         return(f) ;
    }
int Fib_search(RecType ST[] , KeyType key , int n)
   /*  在有序表ST中用Fibonacci方法查找关键字为key的记录  */
    {    int Low=1, High, Mid, f1, f2 ;
         High=n ; f1=fib(n-1) ; f2=fib(n-2) ;
         while (Low<=High)
             {   Mid=Low+f1-1;
```

```
            if ( EQ(ST.[Mid].key, key) )   return(Mid) ;
            else if ( LT(key, ST.[Mid].key) )
                  {   High=Mid-1 ; f2=f1-f2 ; f1=f1-f2 ;    }
                   else
                      {   Low=Mid+1 ;f1=f1-f2 ; f2=f2-f1 ;  }
        }
    return(0) ;
  }
```

由算法知，**Fibonacci**查找在最坏情况下性能比折半查找差，但折半查找要求记录按关键字有序；**Fibonacci**查找的优点是分割时只需进行加、减运算。

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

9.3 动态查找

当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。

利用树的形式组织查找表，可以对查找表进行动态高效的查找。

9.3.1 二叉排序树(BST)的定义

二叉排序树(Binary Sort Tree或Binary Search Tree) 的定义为：二叉排序树或者是空树，或者是满足下列性质的二叉树。

(1)：若左子树不为空，则左子树上所有结点的值(关键字)都小于根结点的值；

(2)：若右子树不为空，则右子树上所有结点的值(关键字)都大于根结点的值；

(3)：左、右子树都分别是二叉排序树。

结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。

BST仍然可以用二叉链表来存储，如图9-4所示。

结点类型定义如下：

```
typedef  struct  Node
   {  KeyType  key ;    /*  关键字域  */
      …       /*  其它数据域  */
      struct  Node  *Lchild , *Rchild ;
   }BSTNode ;
```

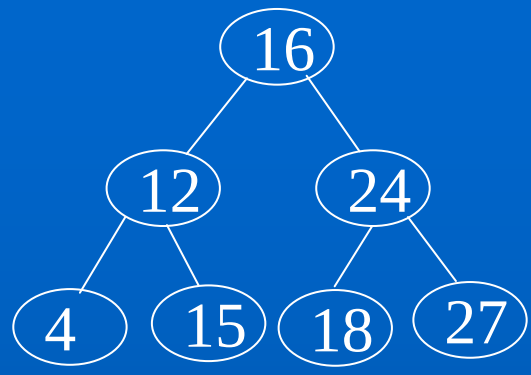


图9-4 二叉排序树

9.3.2 BST树的查找

1 查找思想

首先将给定的K值与二叉排序树的根结点的关键字进行比较：若相等： 则查找成功；

① 给定的K值小于BST的根结点的关键字：继续在该结点的左子树上进行查找；

② 给定的K值大于BST的根结点的关键字：继续在该结点的右子树上进行查找。

2 算法实现

⑴ 递归算法

```
BSTNode *BST_Serach(BSTNode *T ,
KeyType key)
  {  if (T==NULL)  return(NULL) ;
     else
       {  if  (EQ(T->key, key) )  return(T) ;
          else if ( LT(key, T->key) )
                    return(BST_Serach(T->Lchild,
          key)) ;
               else  return(BST_Serach(T->Rchild,
          key)) ;
       }
  }
```

(2) 非递归算法

```
BSTNode *BST_Serach(BSTNode *T , KeyType key)
  {  BSTNode p=T ;
     while (p!=NULL&& !EQ(p->key, key) )
       {  if ( LT(key, p->key) )  p=p->Lchild ;
          else p=p->Rchild ;
       }
     if  (EQ(p->key, key) )  return(p) ;
     else return(NULL) ;
  }
```

在随机情况下，二叉排序树的平均查找长度ASL和$\log(n)$(树的深度)是等数量级的。

9.3.3 BST树的插入

在BST树中插入一个新结点，要保证插入后仍满足BST的性质。

1 插入思想

在BST树中插入一个新结点x时，若BST树为空，则令新结点x为插入后BST树的根结点；否则，将结点x的关键字与根结点T的关键字进行比较：

① 若相等： 不需要插入；

② 若x.key<T->key： 结点x插入到T的左子树中；

③ 若x.key>T->key： 结点x插入到T的右子树中。

2 算法实现

(1) 递归算法

```
void  Insert_BST (BSTNode *T , KeyType  key)
   {  BSTNode *x ;
      x=(BSTNode *)malloc(sizeof(BSTNode)) ;
      X->key=key; x->Lchild=x->Rchild=NULL ;
      if (T==NULL)  T=x ;
      else
         {  if (EQ(T->key, x->key) ) return  ;/*  已有结点  */
            else if (LT(x->key, T->key) )
                     Insert_BST(T->Lchild, key) ;
                 else   Insert_BST(T->Rchild, key) ;   }
   }
```

(2) 非递归算法

```
void Insert_BST (BSTNode *T , KeyType key)
   {  BSTNode *x, *p , *q ;
      x=(BSTNode *)malloc(sizeof(BSTNode)) ;
      X->key=key; x->Lchild=x->Rchild=NULL ;
      if (T==NULL)  T=x ;
      else
         {  p=T ;
            while (p!=NULL)
               {  if  (EQ(p->key, x->key) )  return  ;
                  q=p ;   /*q作为p的父结点  */
```

```
            if (LT(x->key, p->key) )  p=p->Lchild ;
             else p=p->Rchild ;
          }
        if (LT(x->key, q->key) )  q->Lchild=x ;
        else q->Rchild=x ;
      }
  }
```

由结论知，对于一个无序序列可以通过构造一棵BST树而变成一个有序序列。

由算法知，每次插入的新结点都是BST树的叶子结点，即在插入时不必移动其它结点，仅需修改某个结点的指针。

利用BST树的插入操作，可以从空树开始逐个插入每个结点，从而建立一棵BST树，算法如下：

```
#define ENDKEY  65535
BSTNode *create_BST()
   {   KeyType  key ;
       BSTNode *T=NULL ;
       scanf(“%d”, &key) ;
       while (key!=ENDKEY)
          {   Insert_BST(T, key) ;
              scanf(“%d”, &key) ;
          }
       return(T) ;
   }
```

9.3.4 BST树的删除

1 删除操作过程分析

从BST树上删除一个结点，仍然要保证删除后满足BST的性质。设被删除结点为p，其父结点为f，删除情况如下：

① 若p是叶子结点： 直接删除p，如图9-5(b)所示。

② 若p只有一棵子树(左子树或右子树)：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树。即原来p是f的左子树，则p的子树成为f的左子树；原来p是f的右子树，则p的子树成为f的右子树，如图9-5(c)、(d)所示。

③ 若p既有左子树又有右子树：处理方法有以下两种，可以任选其中一种。

- 用p的直接前驱结点代替p。即从p的左子树中选择值最大的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的左子树中的最右边的结点且没有右子树，对s的删除同②，如图9-5(e)所示。
- 用p的直接后继结点代替p。即从p的右子树中选择值最小的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的右子树中的最左边的结点且没有左子树，对s的删除同②，如图9-5(f)所示。

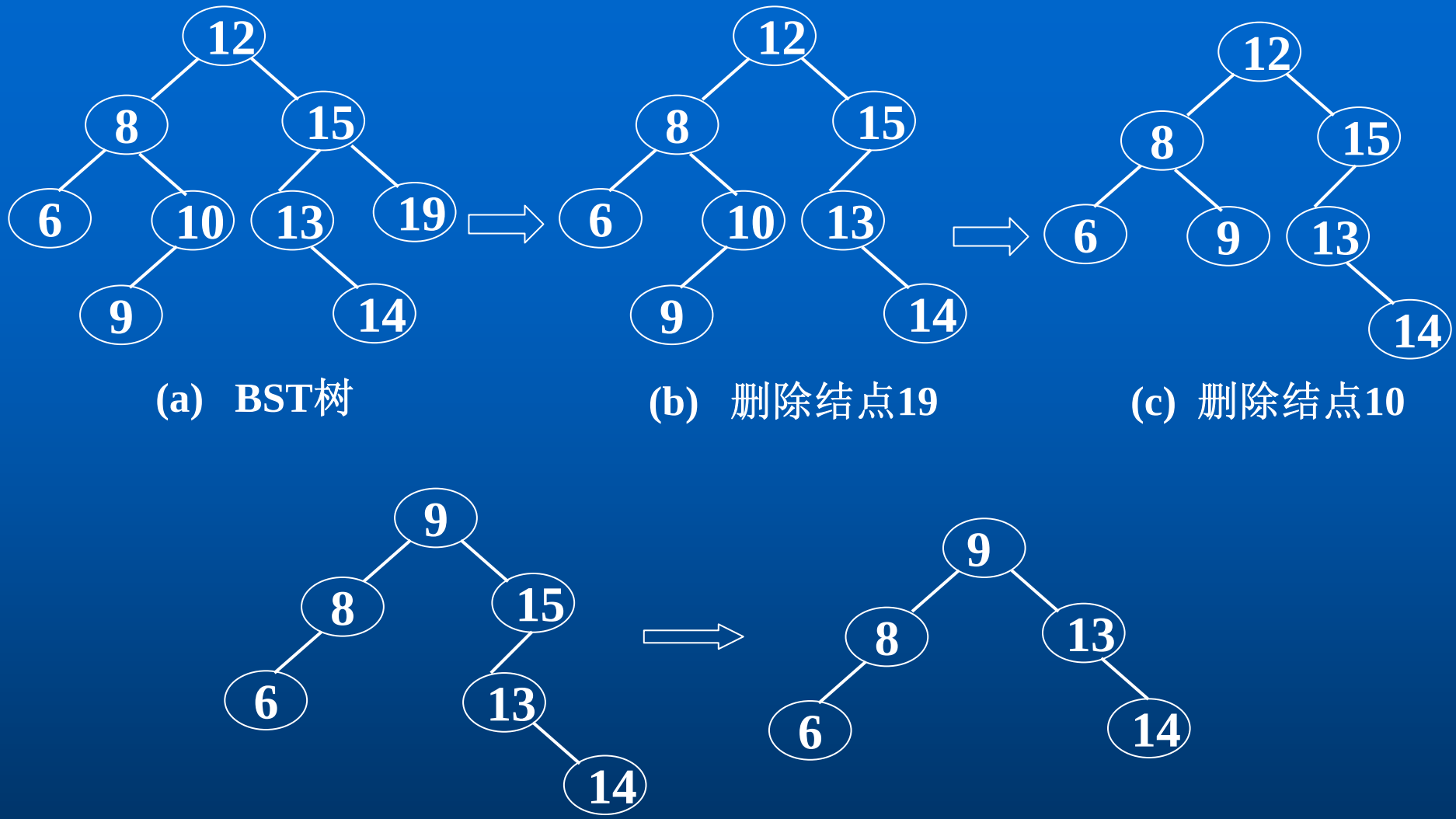


图9-5 BST树的结点删除情况

2 算法实现

```
void Delete_BST (BSTNode *T , KeyType  key )
   /*  在以T为根结点的BST树中删除关键字为key的结点   */
   {  BSTNode *p=T , *f=NULL , *q , *s ;
      while ( p!=NULL&&!EQ(p->key, key) )
         {  f=p ;
            if (LT(key, p->key) ) p=p->Lchild ;  /*  搜索左子
            树 */
            else p=p->Rchild ;  /*  搜索右子树 */
         }
      if  (p==NULL)  return ;     /*  没有要删除的结点   */
```

```
s=p ;    /* 找到了要删除的结点为p  */
if (p->Lchild!=NULL&& p->Rchild!=NULL)
   {  f=p ; s=p->Lchild ;        /* 从左子树开始找  */
      while (s->Rchild!=NULL)
          {   f=s ; s=s->Rchild ;  }
      /*  左、右子树都不空，找左子树中最右边的结点  */
      p->key=s->key ; p->otherinfo=s->otherinfo ;
              /* 用结点s的内容替换结点p内容  */
   }  /* 将第3种情况转换为第2种情况*/
if  (s->Lchild!=NULL)  /* 若s有左子树，右子树为空 */
   q=s->Lchild ;
```

```
    else q=s->Rchild ;
    if  (f==NULL)  T=q ;
        else if (f->Lchild==s)  f->Lchild=q ;
                else f->Rchild=q ;
    free(s) ;
}
```

9.4 平衡二叉树(AVL)

BST是一种查找效率比较高的组织形式，但其平均查找长度受树的形态影响较大，形态比较均匀时查找效率很好，形态明显偏向某一方向时其效率就大大降低。因此，希望有更好的二叉排序树，其形态总是均衡的，查找时能得到最好的效率，这就是平衡二叉排序树。

平衡二叉排序树(Balanced Binary Tree或Height-Balanced Tree)是在1962年由Adelson-Velskii和Landis提出的，又称AVL树。

9.4.1 平衡二叉树的定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树。

⑴：左子树和右子树深度之差的绝对值不大于1；

⑵：左子树和右子树也都是平衡二叉树。

平衡因子(Balance Factor)：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。

因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1， 该二叉树就不是平衡二叉树。

如果一棵二叉树既是二叉排序树又是平衡二叉树，称为平衡二叉排序树(Balanced Binary Sort Tree)。

结点类型定义如下：

```
typedef  struct  BNode
  {  KeyType  key ;    /*  关键字域  */
     int  Bfactor ;     /*  平衡因子域  */
     …       /*  其它数据域  */
     struct  BNode  *Lchild , *Rchild ;
  }BSTNode ;
```

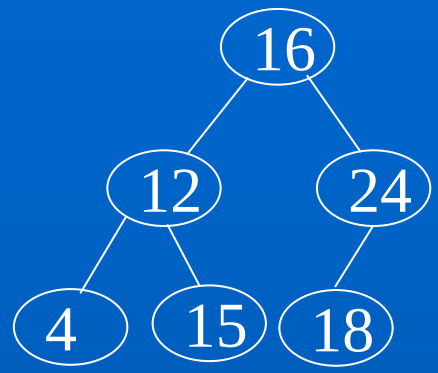


图9-6 平衡二叉树

在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，则在AVL树上执行查找时，和给定的K值比较的次数不超过树的深度。

设深度为h的平衡二叉排序树所具有的最少结点数为N_h，则由平衡二叉排序树的性质知：

$N_0=0$，$N_1=1$，$N_2=2$，…，$N_h=N_{h-1}+N_{h-2}$

该关系和Fibonacci数列相似。根据归纳法可证明，当$h\geqslant 0$时，$N_h=F_{h+2}-1$，…而

$$F_h\approx\frac{\Phi^h}{\sqrt{5}}\quad 其中\ \Phi=\frac{1+\sqrt{5}}{2}\qquad 则N_h\approx\frac{\Phi^h}{\sqrt{5}}-1$$

这样，含有n个结点的平衡二叉排序树的最大深度为

$$h\approx\log_{\Phi}(\sqrt{5}\times(n+1))-2$$

则在平衡二叉排序树上进行查找的平均查找长度和$\log_2 n$是一个数量级的，平均时间复杂度为$O(\log_2 n)$。

9.4.2 平衡化旋转

一般的二叉排序树是不平衡的，若能通过某种方法使其既保持有序性，又具有平衡性，就找到了构造平衡二叉排序树的方法，该方法称为平衡化旋转。

在对AVL树进行插入或删除一个结点后，通常会影响到从根结点到插入(或删除)结点的路径上的某些结点，这些结点的子树可能发生变化。以插入结点为例，影响有以下几种可能性

- 以某些结点为根的子树的深度发生了变化；
- 某些结点的平衡因子发生了变化；
- 某些结点失去平衡。

沿着插入结点上行到根结点就能找到某些结点，这些结点的平衡因子和子树深度都会发生变化，这样的结点称为失衡结点。

1 LL型平衡化旋转

(1) 失衡原因

在结点a的左孩子的左子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后的平衡因子是2。设b是a的左孩子，b在插入前的平衡因子只能是0，插入后的平衡因子是1(否则b就是失衡结点)。

(2) 平衡化旋转方法

通过顺时针旋转操作实现，如图9-7所示。

用b取代a的位置，a成为b的右子树的根结点，b原来的右子树作为a的左子树。

(3) 插入后各结点的平衡因子分析

① 旋转前的平衡因子

设插入后b的左子树的深度为H_{bL}，则其右子树的深度为$H_{bL}-1$； a的左子树的深度为$H_{bL}+1$。

a的平衡因子为2，则a的右子树的深度为：

$H_{aR}=H_{bL}+1-2=H_{bL}-1$。

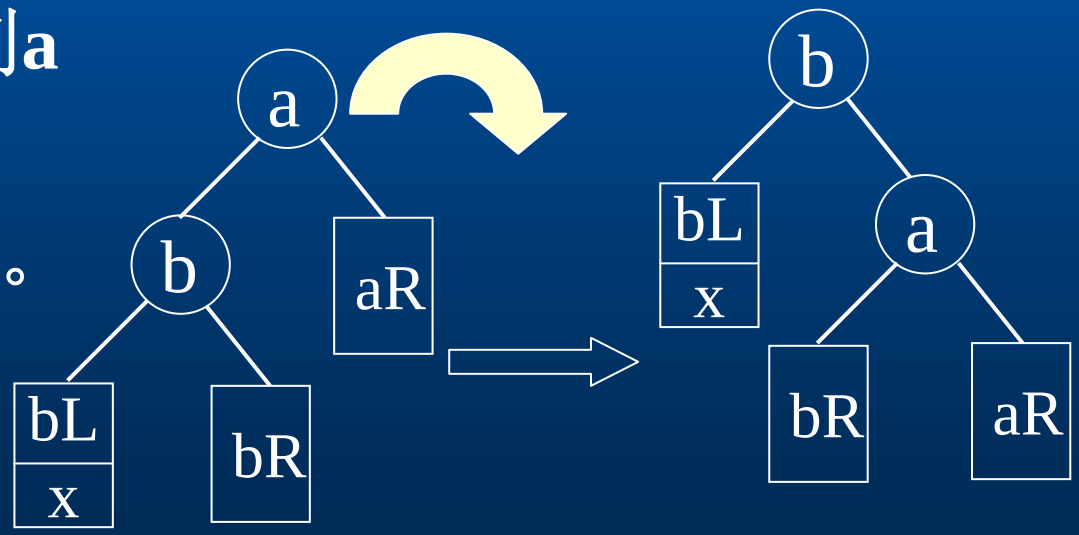


图9-7 LL型平衡化旋转示意图

② 旋转后的平衡因子

a的右子树没有变，而左子树是b的右子树，则平衡因子是：$H_{aL}- H_{aR}=(H_{bL}-1)-(H_{bL}-1)=0$

即a是平衡的，以a为根的子树的深度是H_{bL}。

b的左子树没有变化，右子树是以a为根的子树，则平衡因子是： $H_{bL}-H_{bL}=0$

即b也是平衡的，以b为根的子树的深度是$H_{bL}+1$，与插入前a的子树的深度相同，则该子树的上层各结点的平衡因子没有变化，即整棵树旋转后是平衡的。

(4) 旋转算法

```
void LL_rotate(BBSTNode *a)
  { BBSTNode *b;
    b=a->Lchild; a->Lchild=b->Rchild;
    b->Rchild=a;
    a->Bfactor=b->Bfactor=0; a=b;
  }
```

2 LR型平衡化旋转

(1) 失衡原因

在结点a的左孩子的右子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后a的平衡因子是2。设b是a的左孩子，c为b的右孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是-1；c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：即在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}-1；b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2。

因插入后a的平衡因子是2，则a的右子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；插入后a的平衡因子是2，则a的右子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：即在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}+1$，以c为根的子树的深度是$H_{cL}+2$；因b插入后的平衡因子是-1，则b的左子树的深度为$H_{cL}+1$，以b为根的子树的深度是$H_{cL}+3$；则a的右子树的深度是$H_{cL}+1$。

(3) 平衡化旋转方法

先以b进行一次逆时针旋转(将以b为根的子树旋转为以c为根)，再以a进行一次顺时针旋转，如图9-8所示。将整棵子树旋转为以c为根，b是c的左子树，a是c的右子树；c的右子树移到a的左子树位置， c的左子树移到b的右子树位置。

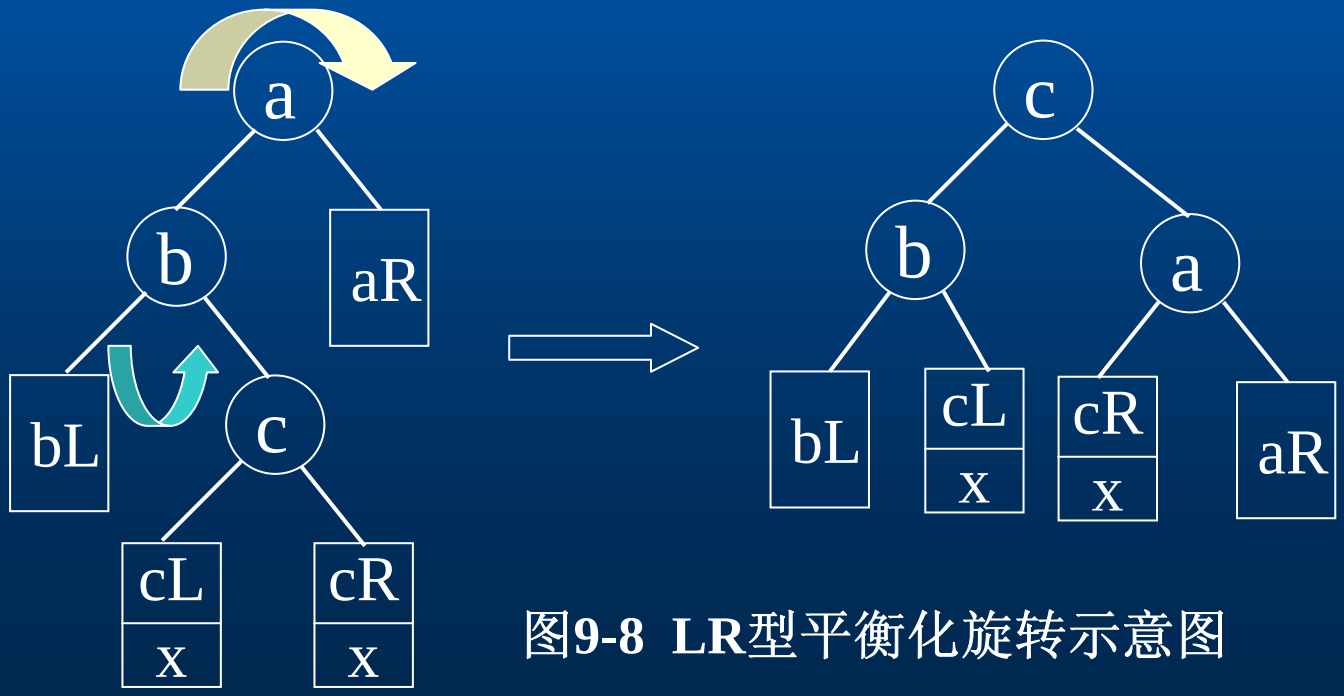


图9-8 LR型平衡化旋转示意图

(4) 旋转后各结点(a,b,c)平衡因子分析

① 旋转前(插入后)c的平衡因子是1：

a的左子树深度为H_{cL}-1，其右子树没有变化，深度是H_{cL}，则a的平衡因子是-1；b的左子树没有变化，深度为H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则b的平衡因子是0； c的左、右子树分别是以b和a为根的子树，则c的平衡因子是0 。

② 旋转前(插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前(插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是0，-1，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
void LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c ;
     b=a->Lchild ; c=b->Rchild ;     /*  初始化  */
     a->Lchild=c->Rchild ; b->Rchild=c->Lchild ;
     c->Lchild=b ; c->Rchild=a ;
     if (c->Bfactor==1)
        {   a->Bfactor=-1 ;b->Bfactor=0 ;   }
     else if (c->Bfactor==0)  a->Bfactor=b-
     >Bfactor=0 ;
          else {   a->Bfactor=0 ;b->Bfactor=1 ;   }
  }
```

3 RL型平衡化旋转

(1) 失衡原因

在结点a的右孩子的左子树上进行插入，插入使结点a失去平衡，与LR型正好对称。对于结点a，插入前的平衡因子是-1，插入后a的平衡因子是-2。设b是a的右孩子，c为b的左孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是1；同样，c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}-1$。

因b插入后的平衡因子是1，则其右子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}+1$，以c为根的子树的深度是$H_{cL}+2$；因b插入后的平衡因子是1，则b的右子树的深度为$H_{cL}+1$，以b为根的子树的深度是$H_{cL}+3$；则a的右子树的深度是$H_{cL}+1$。

(3) 平衡化旋转方法

先以b进行一次顺时针旋转，再以a进行一次逆时针旋转，如图9-9所示。即将整棵子树(以a为根)旋转为以c为根，a是c的左子树，b是c的右子树；c的右子树移到b的左子树位置，c的左子树移到a的右子树位置。

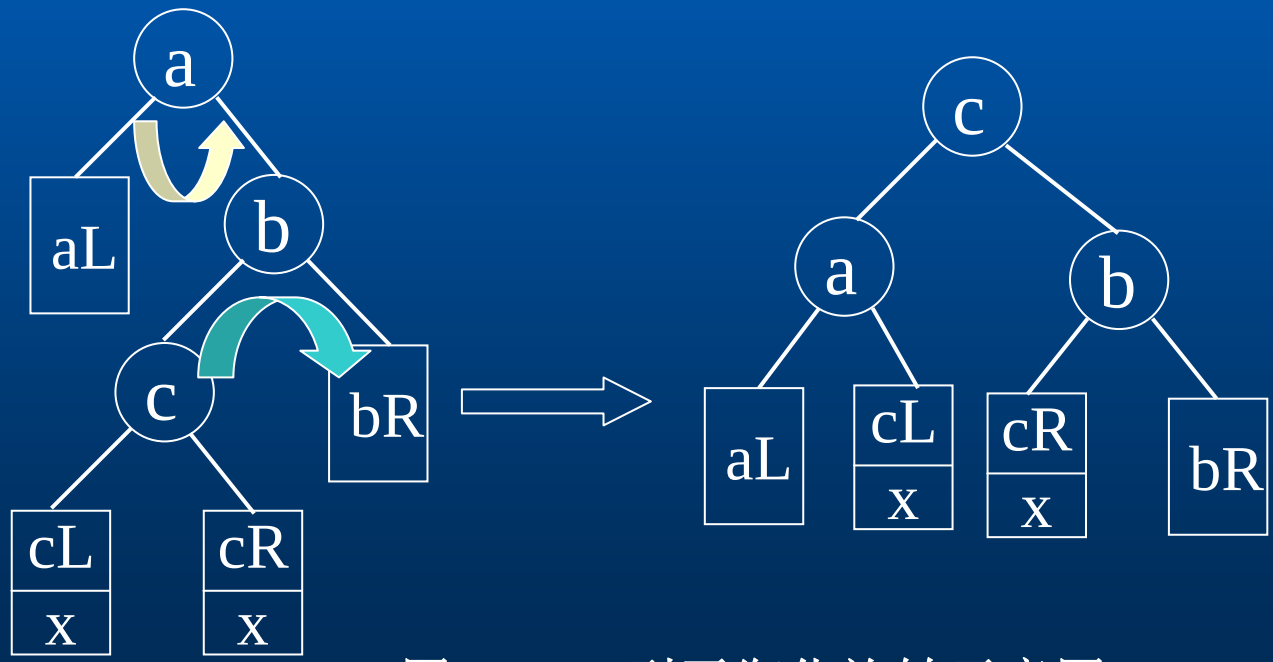


图9-9 RL型平衡化旋转示意图

(4) 旋转后各结点(a, b, c)的平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树没有变化，深度是H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则a的平衡因子是0；b的右子树没有变化，深度为H_{cL}，左子树是c旋转前的右子树，深度为H_{cL}-1，则b的平衡因子是-1； c的左、右子树分别是以a 和b为根的子树，则c的平衡因子是0。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是1，0，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
Void LR_rotate(BBSTNode *a)
  { BBSTNode *b,*c ;
    b=a->Rchild ; c=b->Lchild ;     /* 初始化 */
    a->Rchild=c->Lchild ; b->Lchild=c->Rchild ;
    c->Lchild=a ; c->Rchild=b ;
    if (c->Bfactor==1)
      {  a->Bfactor=0 ; b->Bfactor=-1 ;  }
    else if (c->Bfactor==0) a->Bfactor=b-
    >Bfactor=0 ;
        else {  a->Bfactor=1 ;b->Bfactor=0 ;  }
  }
```

4 RR型平衡化旋转

(1) 失衡原因

在结点a的右孩子的右子树上进行插入，插入使结点a失去平衡。要进行一次逆时针旋转，和LL型平衡化旋转正好对称。

(2) 平衡化旋转方法

设b是a的右孩子，通过逆时针旋转实现，如图9-10所示。用b取代a的位置，a作为b的左子树的根结点，b原来的左子树作为a的右子树。

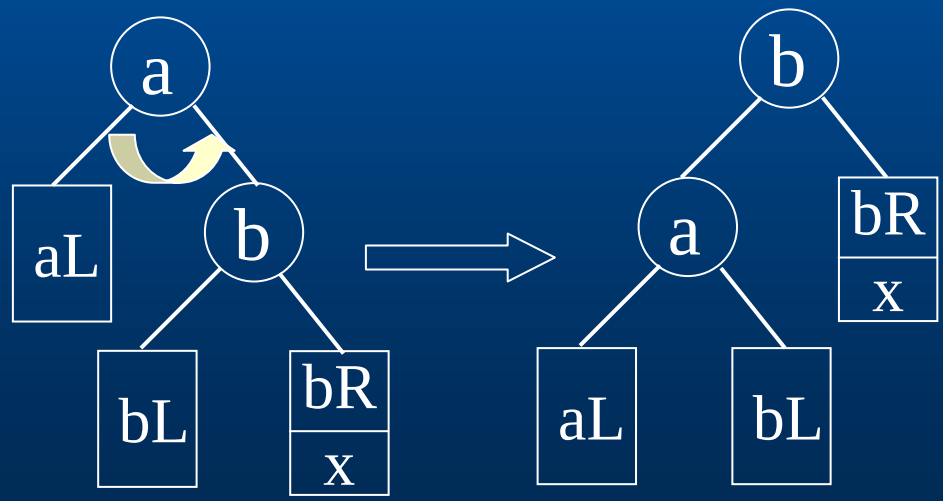


图9-10 RR型平衡化旋转示意图

(3) 旋转算法

```
BBSTNode *RR_rotate(BBSTNode *a)
  { BBSTNode *b;
    b=a->Rchild; a->Rchild=b->Lchild; b->Lchild=a;
    a->Bfactor=b->Bfactor=0; a=b;
  }
```

对于上述四种平衡化旋转，其正确性容易由“遍历所得中序序列不变”来证明。并且，无论是哪种情况，平衡化旋转处理完成后，形成的新子树仍然是平衡二叉排序树，且其深度和插入前以a为根结点的平衡二叉排序树的深度相同。所以，在平衡二叉排序树上因插入结点而失衡，仅需对失衡子树做平衡化旋转处理。

9.4.3 平衡二叉排序树的插入

平衡二叉排序树的插入操作实际上是在二叉排序插入的基础上完成以下工作：

(1)：判别插入结点后的二叉排序树是否产生不平衡?

(2)：找出失去平衡的最小子树；

(3)：判断旋转类型，然后做相应调整。

失衡的最小子树的根结点a在插入前的平衡因子不为0，且是离插入结点最近的平衡因子不为0的结点的。

若a失衡，从a到插入点的路径上的所有结点的平衡因子都会发生变化，在该路径上还有一个结点的平衡因子不为0且该结点插入后没有失衡，其平衡因子只能是由1到0或由-1到0，以该结点为根的子树深度不变。该结点的所有祖先结点的平衡因子也不变，更不会失衡。

1 算法思想(插入结点的步骤)

①：按照二叉排序树的定义，将结点s插入；

②：在查找结点s的插入位置的过程中，记录离结点s最近且平衡因子不为0的结点a，若该结点不存在，则结点a指向根结点；

③： 修改结点a到结点s路径上所有结点的；

④：判断是否产生不平衡，若不平衡，则确定旋转类型并做相应调整。

2 算法实现

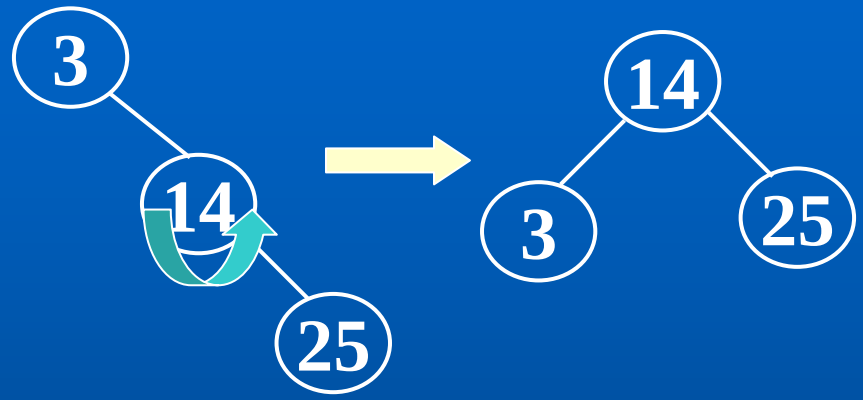
```
void  Insert_BBST(BBSTNode *T, BBSTNode *S)
   {  BBSTNode *f,*a,*b,*p,*q;
      if (T==NULL) {  T=S ; T->Bfactor=1 ; return ; }
      a=p=T ;    /* a指向离s最近且平衡因子不为0的结点 */
      f=q=NULL ;     /*  f指向a的父结点,q指向p父结点   */
      while (p!=NULL)
         { if (EQ(S->key, p->key) ) return ;   /* 结点已存在
         */
           if (p->Bfactor!=0) { a=p ; f=q ; }
           q=p ;
           if (LT(S->key, p->key) ) p=p->Lchild ;
           else p=p->Rchild ;    /* 在右子树中搜索  */
         }    /* 找插入位置  */
```

```
if (LT(S->key,p->key)) q->Lchild=S;/* s为左孩子 */
else  q->Rchild=S;     /* s插入为q的右孩子 */
p=a;
while (p!=S)
   {  if (LT(S->key, p->key) )
         {  p->Bfactor++; p=p->Lchild;  }
      else {  p->Bfactor--; p=p->Rchild;  }
   }    /* 插入到左子树,平衡因子加1,插入到左子树,减1 */
if (a->Bfactor>-2&& a->Bfactor<2)
   return;  /*  未失去平衡,不做调整 */
if (a->Bfactor==2)
   {  b=a->Lchild;
```

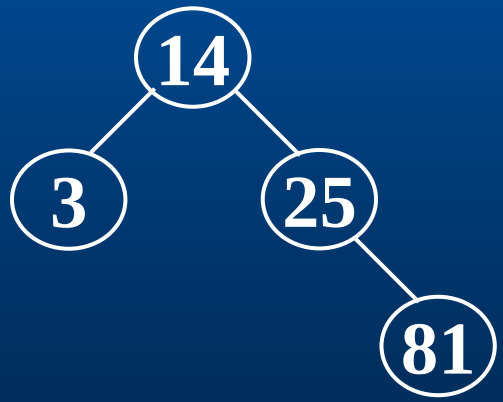
```
        if (b->Bfactor==1)   p=LL_rotate(a);
        else p=LR_rotate(a);
    }
  else
    {  b=a->Rchild;
        if (b->Bfactor==1)   p=RL_rotate(a);
        else  p=RR_rotate(a);
    }  /*  修改双亲结点指针  */
  if (f==NULL)  T=p;     /*  p为根结点   */
  else   if (f->Lchild==a) f->Lchild=p;
        else  f->Lchild=p;
}
```

例： 设要构造的平衡二叉树中各结点的值分别是(3, 14, 25, 81, 44)，平衡二叉树的构造过程如图9-11所示。

(a) 插入不超过两个结点



(b) 插入新结点失衡,RR平衡旋转



(c) 插入新结点未失衡

(d) 插入结点失衡,RL平衡旋转

图9-11 平衡二叉树的构造过程

9.5 索引查找

索引技术是组织大型数据库的重要技术，索引结构的基本组成是索引表和数据表两部分，如图9-12所示。

- 数据表：存储实际的数据记录；
- 索引表：存储记录的关键字和记录(存储)地址之间的对照表，每个元素称为一个索引项。

通过索引表可实现对数据表中记录的快速查找。索引表的组织有线性结构和树形结构两种。

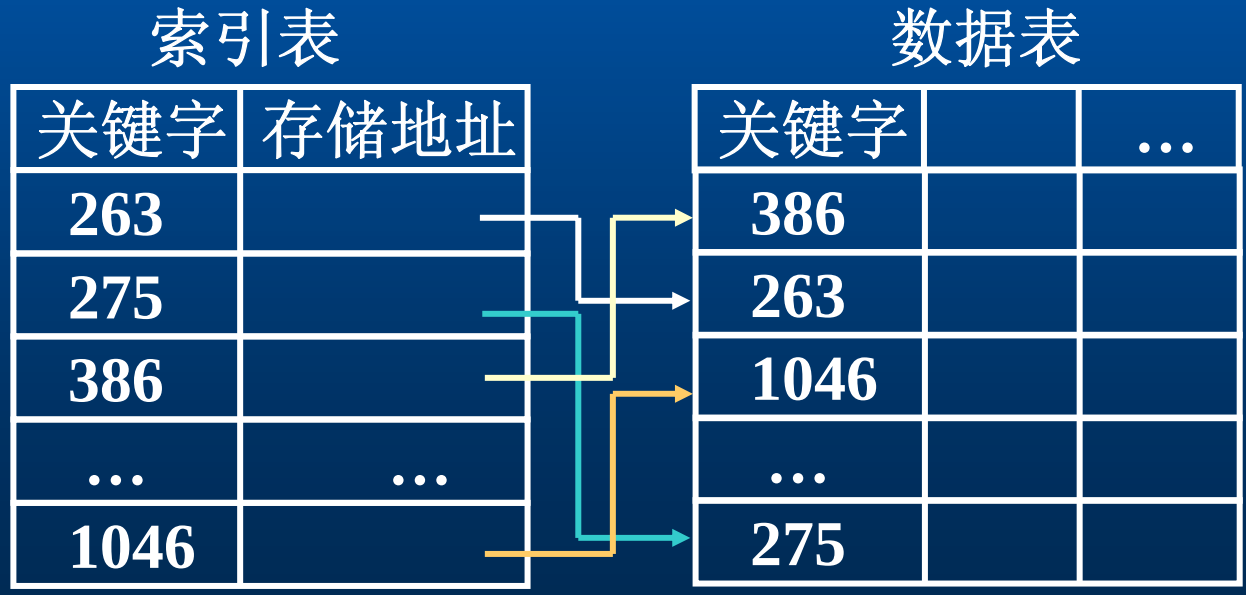


图9-12 索引结构的基本形式

9.5.1 顺序索引表

是将索引项按顺序结构组织的线性索引表，而表中索引项一般是按关键字排序的，其特点是：

优点：

- 可以用折半查找方法快速找到关键字，进而找到数据记录的物理地址，实现数据记录的快速查找；
- 提供对变长数据记录的便捷访问；
- 插入或删除数据记录时不需要移动记录，但需要对索引表进行维护。

缺点：

- 索引表中索引项的数目与数据表中记录数相同，当索引表很大时，检索记录需多次访问外存；
- 对索引表的维护代价较高，涉及到大量索引项的移动，不适合于插入和删除操作。

9.5.2 树形索引表

平衡二叉排序树便于动态查找，因此用平衡二叉排序树来组织索引表是一种可行的选择。当用于大型数据库时，所有数据及索引都存储在外存，因此，涉及到内、外存之间频繁的数据交换，这种交换速度的快慢成为制约动态查找的瓶颈。若以二叉树的结点作为内、外存之间数据交换单位，则查找给定关键字时对磁盘平均进行$\log_2 n$次访问是不能容忍的，因此，必须选择一种能尽可能降低磁盘I/O次数的索引组织方式。树结点的大小尽可能地接近页的大小。

R.Bayer和E.Mc Creight在1972年提出了一种多路平衡查找树，称为B_树(其变型体是B+树)。

1 B_树

B_树主要用于文件系统中，在B_树中，每个结点的大小为一个磁盘页，结点中所包含的关键字及其孩子的数目取决于页的大小。一棵度为m的B_树称为m阶B_树，其定义是：

一棵m阶B_树，或者是空树，或者是满足以下性质的m叉树：

⑴ 根结点或者是叶子，或者至少有两棵子树，至多有m棵子树；

⑵ 除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，至多有m棵子树；

⑶ 所有叶子结点都在树的同一层上；

⑷ 每个结点应包含如下信息：

$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

其中$K_i(1 \leqslant i \leqslant n)$是关键字，且$K_i < K_{i+1}$ $(1 \leqslant i \leqslant n-1)$；$A_i(i=0, 1, \ldots, n)$为指向孩子结点的指针，且$A_{i-1}$所指向的子树中所有结点的关键字都小于$K_i$，$A_i$所指向的子树中所有结点的关键字都大于$K_i$；n是结点中关键字的个数，且$\lfloor m/2 \rfloor - 1 \leqslant n \leqslant m-1$，$n+1$为子树的棵数。

当然，在实际应用中每个结点中还应包含n个指向每个关键字的记录指针，如图9-13是一棵包含13个关键字的4阶B_树。

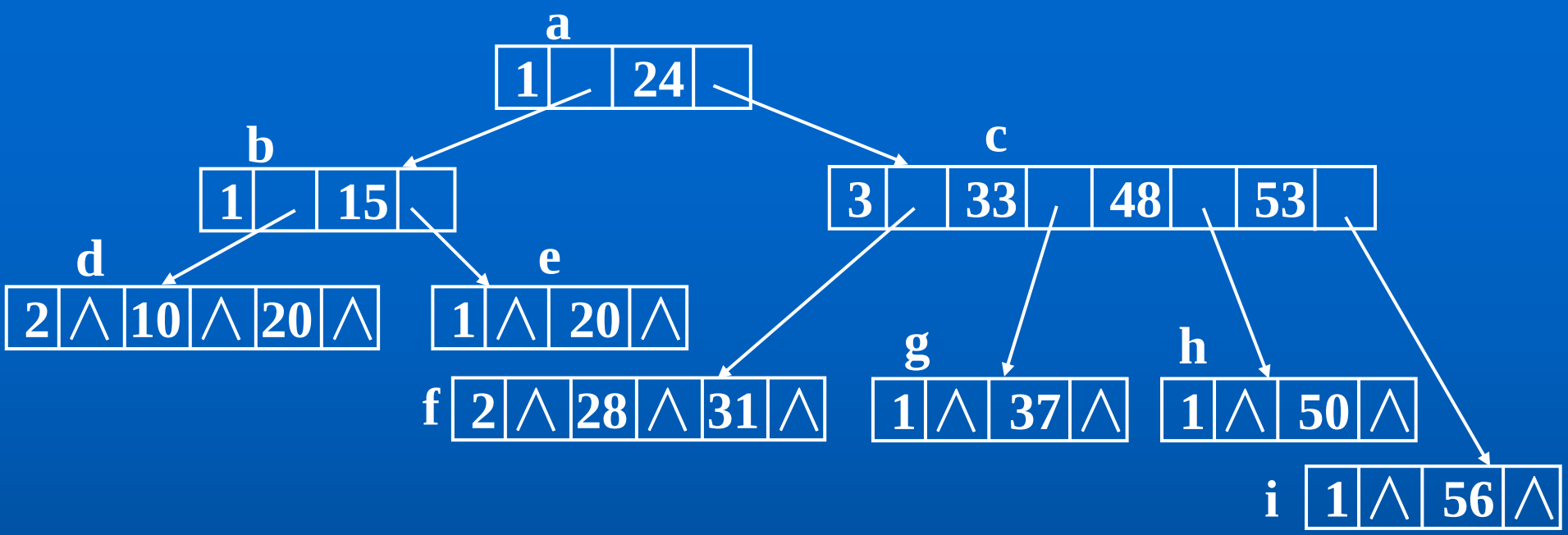


图9-13　一棵包含13个关键字的4阶B_树

根据m阶B_树的定义，结点的类型定义如下：

```
#define M   5      /* 根据实际需要定义B_树的阶数 */
typedef  struct BTNode
  {  int  keynum;  /*  结点中关键字的个数  */
     struct BTNode  *parent;   /*  指向父结点的指针  */
     KeyType  key[M+1];   /*  关键字向量,key[0]未用  */
     struct BTNode  *ptr[M+1]; /*  子树指针向量  */
     RecType  *recptr[M+1];
        /*  记录指针向量,recptr[0]未用  */
  }BTNode;
```

2 B_树的查找

由B_树的定义可知，在其上的查找过程和二叉排序树的查找相似。

(1) 算法思想

① 从树的根结点T开始，在T所指向的结点的关键字向量key[1...keynum]中查找给定值K(用折半查找)：

若key[i]=K(1≤i≤keynum)，则查找成功，返回结点及关键字位置；否则，转(2)；

② 将K与向量key[1...keynum]中的各个分量的值进行比较，以选定查找的子树：

- 若K<key[1]：T=T->ptr[0]；

◆ 若key[i]<K<key[i+1](i=1, 2, ...keynum-1)：

T=T->ptr[i]；

◆ 若K>key[keynum]：T=T->ptr[keynum]；

转①，直到T是叶子结点且未找到相等的关键字，则查找失败。

(2) 算法实现

```
int  BT_search(BTNode *T, KeyType K, BTNode *p)
   /*   在B_树中查找关键字K, 查找成功返回在结点中的位置 */
   /*   及结点指针p; 否则返回0及最后一个结点指针    */
  {   BTNode *q ; int n ;
      p=q=T ;
```

```
    while (q!=NULL)
      {  p=q ; q->key[0]=K ;     /*  设置查找哨兵
      */
        for (n=q->keynum ; K<q->key[n] ; n--)
            if (n>0&&EQ(q->key[n], K) )   return
        n ;
        q=q->ptr[n] ;
      }
    return 0 ;
  }
```

(3) 算法分析

在B_树上的查找有两中基本操作：

- 在B_树上查找结点(查找算法中没有体现)；

◆ 在结点中查找关键字：在磁盘上找到指针ptr所指向的结点后，将结点信息读入内存后再查找。因此，磁盘上的查找次数(待查找的记录关键字在B_树上的层次数)是决定B_树查找效率的首要因素。

根据m阶B_树的定义，第一层上至少有1个结点，第二层上至少有2个结点；除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，…，第h层上至少有$\lceil m/2 \rceil^{h-2}$个结点。在这些结点中：根结点至少包含1个关键字，其它结点至少包含$\lceil m/2 \rceil-1$个关键字，设$s=\lceil m/2 \rceil$，则总的关键字数目n满足：

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

因此有： $h \leqq 1+\log_s((n+1)/2)=1+\log_{\lceil m/2 \rceil}((n+1)/2)$

即在含有n个关键字的B_树上进行查找时，从根结点到待查找记录关键字的结点的路径上所涉及的结点数不超过$1+\log_{\lceil m/2 \rceil}((n+1)/2)$ 。

3 B_树的插入

B_树的生成也是从空树起，逐个插入关键字。插入时不是每插入一个关键字就添加一个叶子结点，而是首先在最低层的某个叶子结点中添加一个关键字，然后有可能“分裂”。

(1) 插入思想

① 在B_树的中查找关键字K，若找到，表明关键字已存在，返回；否则，K的查找操作失败于某个叶子结点，转 ②；

② 将K插入到该叶子结点中，插入时，若：

- 叶子结点的关键字数 < m-1：直接插入；
- 叶子结点的关键字数=m-1：将结点“分裂”。

(2) 结点“分裂”方法

设待“分裂”结点包含信息为：

$(m, A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m)$，从其中间位置分为两个结点：

$(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

$(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \ldots, K_m, A_m)$

并将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点中，以分裂后的两个结点作为中间关键字$K_{\lceil m/2 \rceil}$的两个子结点。

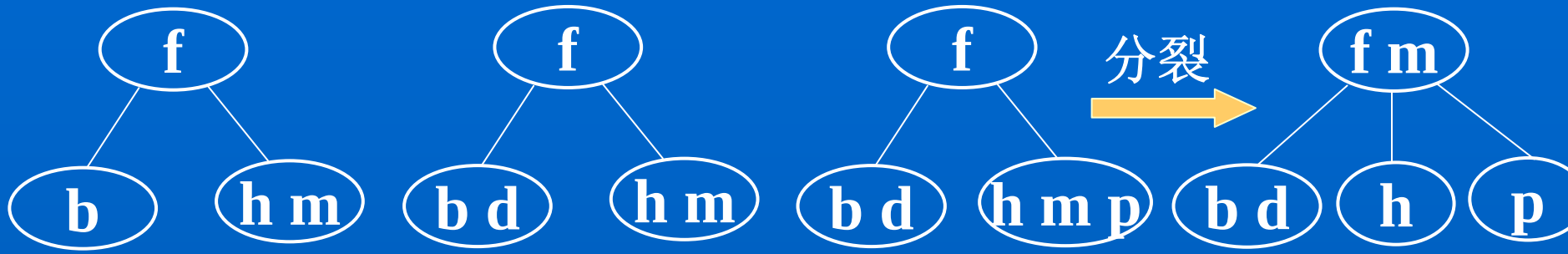
当将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点后，父结点也可能不满足m阶B_树的要求(分枝数大于m)，则必须对父结点进行“分裂”，一直进行下去，直到没有父结点或分裂后的父结点满足m阶B_树的要求。

当根结点分裂时，因没有父结点，则建立一个新的根，B_树增高一层。

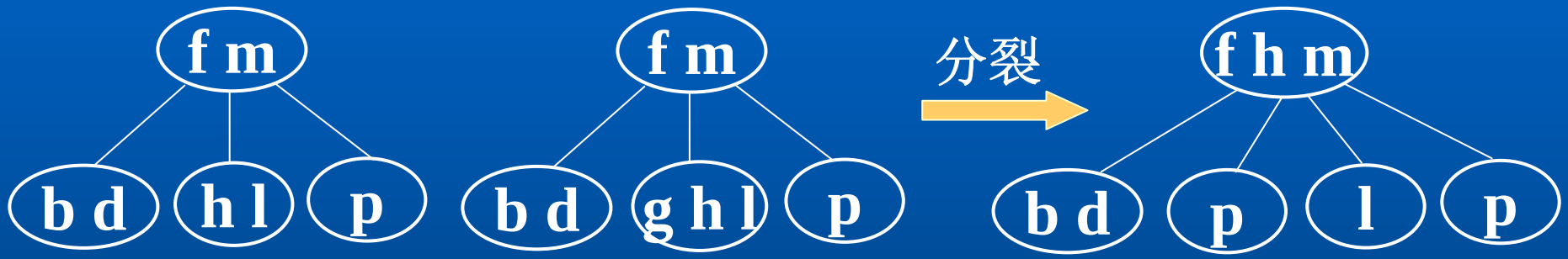
例：在一个3阶B_树(2-3树)上插入结点，其过程如图9-14所示。

(3) 算法实现

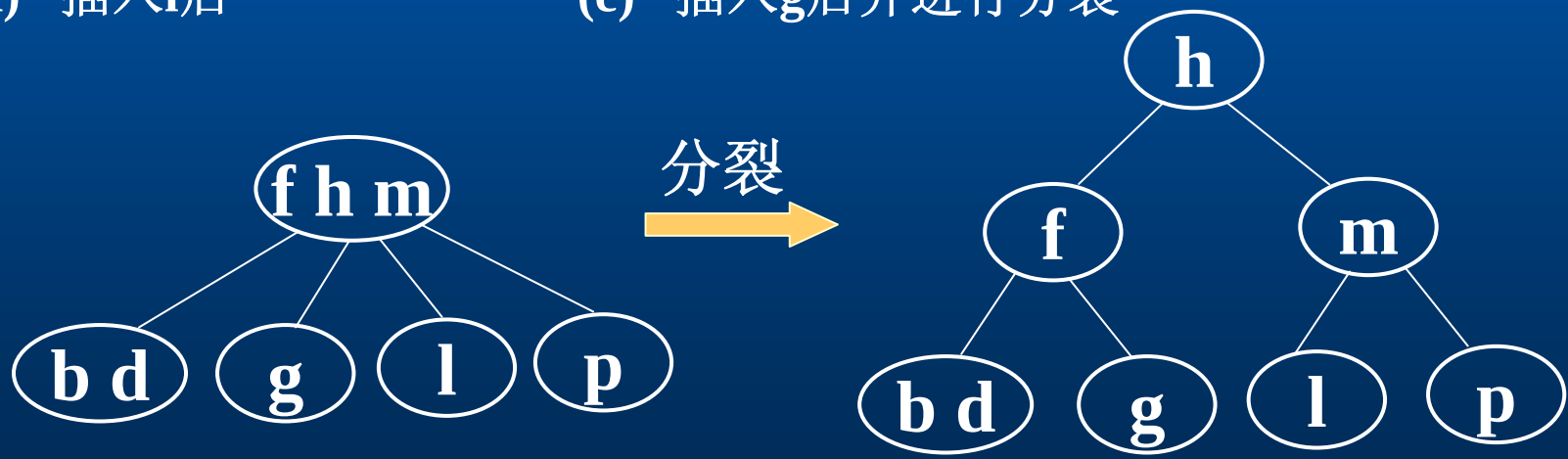
要实现插入，首先必须考虑结点的分裂。设待分裂的结点是p，分裂时先开辟一个新结点，依此将结点p中后半部分的关键字和指针移到新开辟的结点中。分裂之后，而需要插入到父结点中的关键字在p的关键字向量的p->keynum+1位置上。



(a) 一棵2-3树　(b) 插入d后　(c) 插入p后并进行分裂



(d) 插入l后　(e) 插入g后并进行分裂



(f) 继续进行分裂

图9-14 在B_树中进行插入的过程

```
BTNode *split(BTNode *p)
  /*  结点p中包含m个关键字，从中分裂出一个新的结点 */
  {  BTNode *q; int k, mid, j;
     q=(BTNode *)malloc(sizeof( BTNode));
     mid=(m+1)/2;  q->ptr[0]=p->ptr[mid];
     for (j=1,k=mid+1; k<=m; k++)
        {  q->key[j]=p->key[k];
           q->ptr[j++]=p->ptr[k];
        }  /*  将p的后半部分移到新结点q中  */
     q->keynum=m-mid; p->keynum=mid-1;
     return(q);
  }
```

```
void insert_BTree(BTNode *T, KeyType K)
    /*  在B_树T中插入关键字K，*/
  {  BTNode *q, *s1=NULL, *s2=NULL ;
     int n ;
     if (!BT_search(T, K, p))     /*  树中不存在关键字K  */
        {  while (p!=NULL)
           {  p->key[0]=K ;    /*  设置哨兵   */
              for (n=p->keynum ; K<p->key[n] ; n--)
                 {  p->key[n+1]=p->key[n] ;
                    p->ptr[n+1]=p->ptr[n] ;
                 }   /*  后移关键字和指针   */
              p->key[n]=K ; p->ptr[n-1]=s1 ;
```

```
        p->ptr[n+1]=s2;  /*  置关键字K的左右指针
*/
   if (++(p->keynum ))<m  break;
   else {  s2=split(p);s1=p;  /*  分裂结点p  */
          K=p->key[p->keynum+1];
          p=p->parent;   /*  取出父结点*/
         }
   if (p==NULL)      /*  需要产生新的根结点   */
      {  p=(BTNode *)malloc(sizeof( BTNode));
         p->keynum=1;p->key[1]=K;
         p->ptr[0]=s1;p->ptr[1] =s2;
      }
}
```

利用m阶B_树的插入操作，可从空树起，将一组关键字依次插入到m阶B_树中，从而生成一个m阶B_树。

4 B_树的删除

在B_树上删除一个关键字K，首先找到关键字所在的结点N，然后在N中进行关键字K的删除操作。

若N不是叶子结点，设K是N中的第i个关键字，则将指针A_{i-1}所指子树中的最大关键字(或最小关键字)K'放在(K)的位置，然后删除K'，而K'一定在叶子结点上。如图9-15(b)，删除关键字h，用关键字g代替h的位置，然后再从叶子结点中删除关键字g。

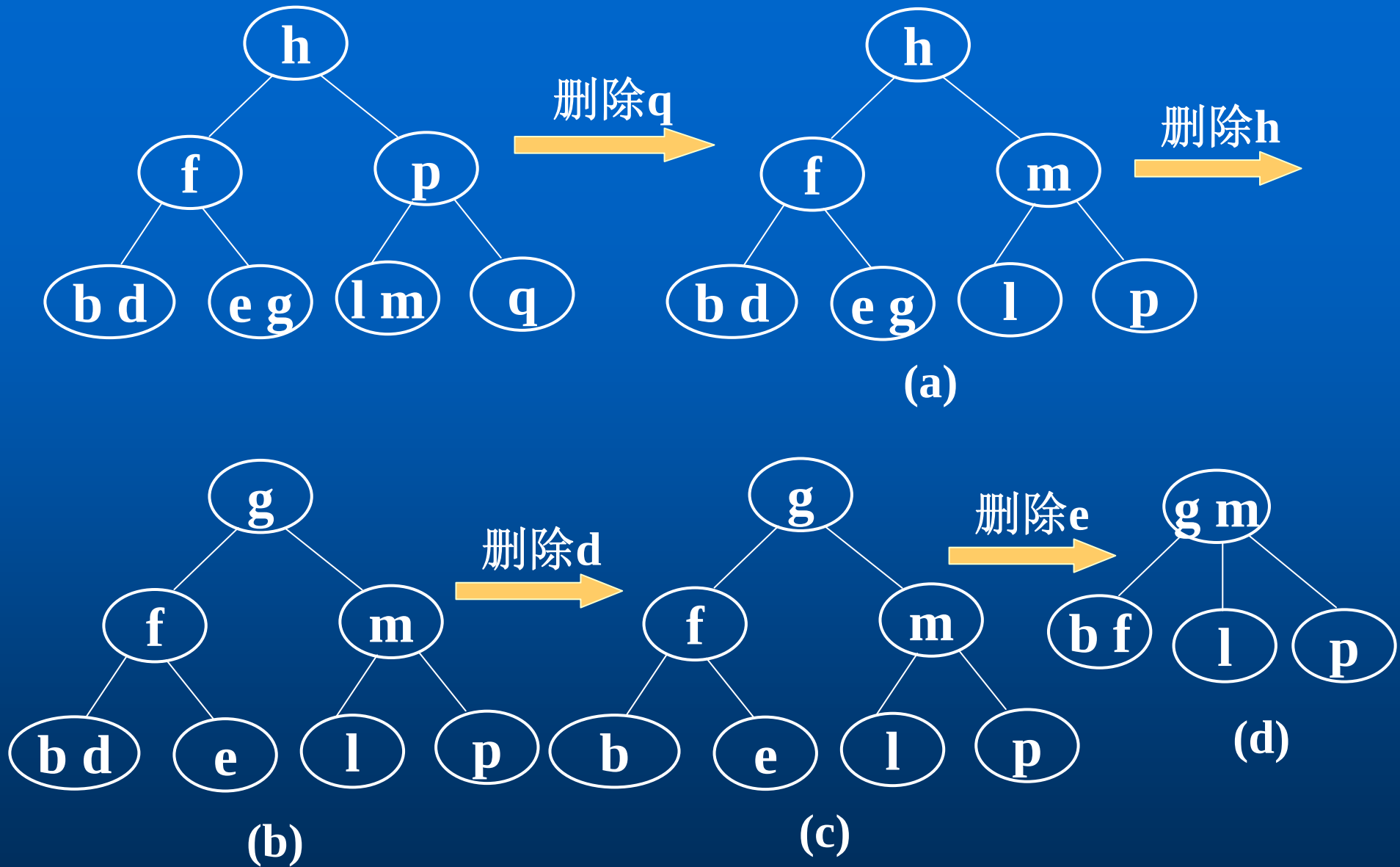


图9-15　在B_树中进行删除的过程

从叶子结点中删除一个关键字的情况是：

⑴ 若结点N中的关键字个数>⌈m/2⌉-1：在结点中直接删除关键字K，如图9-15(b)∽©所示。

⑵ 若结点N中的关键字个数=⌈m/2⌉-1：若结点N的左(右)兄弟结点中的关键字个数>⌈m/2⌉-1，则将结点N的左(或右)兄弟结点中的最大(或最小)关键字上移到其父结点中，而父结点中大于(或小于)且紧靠上移关键字的关键字下移到结点N，如图9-15(a)。

⑶ 若结点N和其兄弟结点中的关键字数=⌈m/2⌉-1：删除结点N中的关键字，再将结点N中的关键字、指针与其兄弟结点以及分割二者的父结点中的某个关键字K_i，合并为一个结点，若因此使父结点中的关键字个数<⌈m/2⌉-1，则依此类推，如图9-15(d)。

算法实现

在B_树上删除一个关键字的操作，针对上述的⑵和⑶的情况，相应的算法如下：

```
int  BTNode  MoveKey(BTNode *p)
   /*   将p的左(或右)兄弟结点中的最大(或最小)关键字上移 */
    /*    到其父结点中,父结点中的关键字下移到p中    */
  {  BTNode *b , *f=p->parent ;    /*   f指向p的父结点    */
     int k, j ;
     for (j=0; f->ptr[j]!=p; j++)    /*   在f中找p的位置    */
        if (j>0)       /*   若p有左邻兄弟结点    */
           {  b=f->ptr[j-1] ;      /*   b指向p的左邻兄弟    */
```

```
if (b->keynum>(m-1)/2)
      /*   左邻兄弟有多余关键字   */
   {  for (k=p->keynum; k>=0; k--)
        {   p->key[k+1]=p->key[k];
            p->ptr[k+1]=p->ptr[k];
        }      /*   将p中关键字和指针后移    */
      p->key[1]=f->key[j];
      f->key[j]=b->key[keynum];
    /*  f中关键字下移到p, b中最大关键字上移到f   */
      p->ptr[0]= b->ptr[keynum];
      p->keynum++;
      b->keynum--;
```

```
            return(1);
        }
    if (j<f->keynum)    /*   若p有右邻兄弟结点    */
        {  b=f->ptr[j+1];      /*   b指向p的右邻兄弟    */
           if (b->keynum>(m-1)/2)
                              /*   右邻兄弟有多余关键字   */
              {  p->key[p->keynum]=f->key[j+1];
                 f->key[j+1]=b->key[1];
                 p->ptr[p->keynum]=b->ptr[0];
              /*  f中关键字下移到p, b中最小关键字上移到f   */
                 for (k=0; k<b->keynum; k++)
```

```
                    {  b->key[k]=b->key[k+1];
                   b->ptr[k]=b->ptr[k+1];
                }     /*  将b中关键字和指针前移  */
              p->keynum++;
              b->keynum--;
              return(1);
            }
        }
      return(0);
    }   /*  左右兄弟中无多余关键字,移动失败  */
}
```

```
BTNode *MergeNode(BTNode *p)
   /*  将p与其左(右)邻兄弟合并,返回合并后的结点指针  */
  {  BTNode *b, f=p->parent ;
     int j, k ;
     for (j=0; f->ptr[j]!=p; j++)   /*  在f中找出p的位置   */
     if (j>0) b=f->ptr[j-1];    /*  b指向p的左邻兄弟   */
     else {  b=p; p=p->ptr[j+1]; }    /*  p指向p的右邻   */
     b->key[++b->keynum]=f->key[j] ;
     b->ptr[p->keynum]=p->ptr[0] ;
     for (k=1; k<=b->keynum ; k++)
        {  b->key[++b->keynum]=p->key[k] ;
           b->ptr[b->keynum]=p->ptr[k] ;
        }    /*  将p中关键字和指针移到b中   */
```

```
    free(p);
    for (k=j+1; k<=f->keynum ; k++)
      {   f->key[k-1]=f->key[k] ;
        f->ptr[k-1]=f->ptr[k] ;
      }     /*  将f中第j个关键字和指针前移   */
    f->keynum-- ;
    return(b) ;
}
```

```
void DeleteBTNode(BTNode *T, KeyType K)
  {  BTNode  *p, *S ;
     int j,n ;
     m=BT_search(T, K, p) ; /* 在T中查找K的结点  */
     if (j==0)  return(T) ;
     if (p->ptr[j-1])
        {  S=p->ptr[j-1] ;
           while (S->ptr[S->keynum])
              S=S->ptr[S->keynum] ;
            /*  在子树中找包含最大关键字的结点   */
           p->key[j]=S->key[S ->keynum] ;
           p=S ; j=S->keynum ;
        }
```

```
    for (n=j+1; n<p->keynum; n++)
      p->key[n-1]=p->key[n];
         /*  从p中删除第m个关键字    */
    p->keynum--;
    while (p->keynum<(m-1)/2&&p->parent)
      {   if (!MoveKey(p) )  p=MergeNode(p);
          p=p->parent;
      }    /*  若p中关键字树目不够,按⑵处理     */
     if (p==T&&T->keynum==0)
      {   T=T->ptr[0];  free(p);   }
}
```

5 B^+树

在实际的文件系统中，基本上不使用B_树，而是使用B_树的一种变体，称为m阶B^+树。它与B_树的主要不同是叶子结点中存储记录。在B^+树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B^+树与m阶B_树的主要差异是：

(1) 若一个结点有n棵子树，则必含有n个关键字；

(2) 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；

(3) 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

如图9-16是一棵3阶B+树。

由于B+树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：

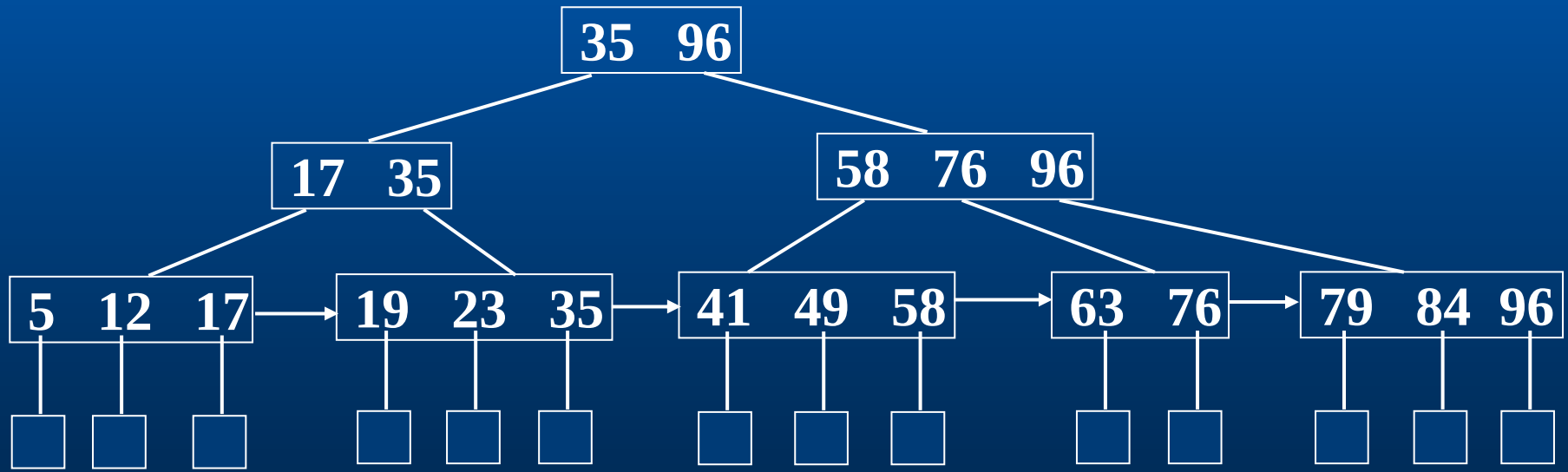


图9-16　一棵3阶B+树

```
typedef enum{branch, left} NodeType;
typedef struct BPNode
  {  NodeTag  tag;    /*  结点标志   */
     int  keynum;  /*  结点中关键字的个数   */
     struct BTNode  *parent;   /*  指向父结点的指针   */
     KeyType  key[M+1];    /*  组关键字向量,key[0]未用  */
     union pointer
        {  struct BTNode  *ptr[M+1]; /*  子树指针向量  */
           RecType  *recptr[M+1]; /*  recptr[0]未用   */
        }ptrType;   /*  用联合体定义子树指针和记录指针  */
  }BPNode;
```

与B_树相比，对B+树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。在B+树上进行随机查找、插入、删除的过程基本上和B_树类似。

在B+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录)，即无论查找成功与否，都走了一条从根结点到叶子结点的路径。

B+树的插入仅仅在叶子结点上进行。当叶子结点中的关键字个数大于m时，“分裂”为两个结点，两个结点中所含有的关键字个数分别是$\lfloor (m+1)/2 \rfloor$和$\lceil (m+1)/2 \rceil$，且将这两个结点中的最大关键字提升到父结点中，用来替代原结点在父结点中所对应的关键字。提升后父结点又可能会分裂，依次类推。

9.6 哈希(散列)查找

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

例 30个地区的各民族人口统计表

编号	省、市(区)	总人口	汉族	回族	.....
1	北京				.
2	上海				
.....	.....				

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1， H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19 H(Shenyang)=19

9.6.1 基本概念

哈希函数：在记录的关键字与记录的存储地址之间建立的一种对应关系叫哈希函数。

哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。可写成：$addr(a_i)=H(k_i)$，其中i是表中一个元素，$addr(a_i)$是a_i的地址，k_i是a_i的关键字。

哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表。

哈希查找(又叫散列查找)：利用哈希函数进行查找的过程叫哈希查找。

冲突：对于不同的关键字k_i、k_j，若$k_i \neq k_j$，但$H(k_i)=H(k_j)$的现象叫冲突(collision)。

同义词：具有相同函数值的两个不同的关键字，称为该哈希函数的同义词。

哈希函数通常是一种压缩映象，所以冲突不可避免，只能尽量减少；当冲突发生时，应该有处理冲突的方法。设计一个散列表应包括：

① 散列表的空间范围，即确定散列函数的值域；

② 构造合适的散列函数，使得对于所有可能的元素(记录的关键字)，函数值均在散列表的地址空间范围内，且出现冲突的可能尽量小；

③ 处理冲突的方法。即当冲突出现时如何解决。

9.6.2 哈希函数的构造

哈希函数是一种映象，其设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。哈希函数“好坏”的主要评价因素有：

- 散列函数的构造简单；
- 能“均匀”地将散列表中的关键字映射到地址空间。所谓“均匀”(uniform)是指发生冲突的可能性尽可能最少。

1 直接定址法

取关键字或关键字的某个线性函数作哈希地址，即H(key)=key 或 H(key)=a·key+b(a,b为常数)

特点：直接定址法所得地址集合与关键字集合大小相等，不会发生冲突，但实际中很少使用。

2 数字分析法

对关键字进行分析，取关键字的若干位或组合作为哈希地址。

适用于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

例： 设有80个记录，关键字为8位十进制数，哈希地址为2位十进制数。

①	②	③	④	⑤	⑥	⑦	⑧
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

分析： ① 只取8
② 只取1
③ 只取3、4
⑧ 只取2、7、5
④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

3 平方取中法

将关键字平方后取中间几位作为哈希地址。

一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的散列地址也是随机的。散列函数所取的位数由散列表的长度决定。这种方法适于不知道全部关键字情况，是一种较为常用的方法。

4 折叠法

将关键字分割成位数相同的几部分(最后一部分可以不同)，然后取这几部分的叠加和作为哈希地址。

数位叠加有移位叠加和间界叠加两种。

- 移位叠加：将分割后的几部分低位对齐相加。
- 间界叠加：从一端到另一端沿分割界来回折迭，然后对齐相加。

适于关键字位数很多，且每一位上数字分布大致均匀情况。

例： 设关键字为**0442205864**，哈希地址位数为**4**。两种不同的地址计算方法如下：

```
  5864
  4220
    04
------
 10088
```

H(key)=0088

移位叠加

```
  5864
  0224
    04
------
  6092
```

H(key)=6092

间界叠加

5 除留余数法

取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即H(key)=key MOD p (p≤m)

是一种简单、常用的哈希函数构造方法。

利用这种方法的关键是p的选取，p选的不好，容易产生同义词。p的选取的分析：

- 选取$p=2^i$(p≤m)：运算便于用移位来实现，但等于将关键字的高位忽略而仅留下低位二进制数。高位不同而低位相同的关键字是同义词。
- 选取p=q×f(q、f都是质因数，p≤m)：则所有含有q或f因子的关键字的散列地址均是q或f的倍数

- 选取p为素数或p=q×f(q、f是质数且均大于20，p≤m)：常用的选取方法，能减少冲突出现的可能性。

6 随机数法

取关键字的随机函数值作哈希地址，即

H(key)=random(key)

当散列表中关键字长度不等时，该方法比较合适。

选取哈希函数，考虑以下因素

- 计算哈希函数所需时间；
- 关键字的长度；
- 哈希表长度（哈希地址范围）；
- 关键字分布情况；
- 记录的查找频率。

9.6.3 冲突处理的方法

冲突处理：当出现冲突时，为冲突元素找到另一个存储位置。

1 开放定址法

基本方法：当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。散列地址的计算公式是：

$H_i(key)=(H(key)+d_i)$ MOD m，i=1, 2, …, k(k≤m-1)

其中：H(key)：哈希函数；m：散列表长度；

d_i：第i次探测时的增量序列；

$H_i(key)$：经第i次探测后得到的散列地址。

(1) 线性探测法

将散列表T[0 ...m-1]看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。

增量序列为：$d_i = 1, 2, 3, \ldots, m-1$

设初次发生冲突的地址是h，则依次探测T[h+1]，T[h+2]...，直到T[m-1]时又循环到表头，再次探测T[0]，T[1]...，直到T[h-1]。探测过程终止的情况是：

- 探测到的地址为空：表中没有记录。若是查找则失败；若是插入则将记录写入到该地址；
- 探测到的地址有给定的关键字：若是查找则成功；若是插入则失败；

◆ 直到T[h]：仍未探测到空地址或给定的关键字，散列表满。

例1：设散列表长为7，记录关键字组为：15, 14, 28, 26, 56, 23，散列函数：H(key)=key MOD 7，冲突处理采用线性探测法。

解：H(15)=15 MOD 7=1　　H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突　$H_1(28)=1$ 又冲突

$H_2(28)=2$　　H(26)=26 MOD 7=5

H(56)=56 MOD 7=0　冲突　$H_1(56)=1$　又冲突

$H_2(56)=2$ 又冲突　$H_3(56)=3$

H(23)=23 MOD 7=2　冲突　$H_1(23)=3$　又冲突

$H_3(23)=4$

0	1	2	3	4	5	6
14	15	28	56	23	26	

线性探测法的特点

- 优点：只要散列表未满，总能找到一个不冲突的散列地址；
- 缺点：每个产生冲突的记录被散列到离冲突最近的空地址上，从而又增加了更多的冲突机会(这种现象称为冲突的“聚集”)。

(2) 二次探测法

增量序列为：$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \ldots\ldots \pm k^2 \quad (k \le \lfloor m/2 \rfloor)$

上述例题若采用二次探测法进行冲突处理，则：

H(15)=15 MOD 7=1　　H(14)=14 MOD 7=0

H(28)=28 MOD 7=0　冲突　$H_1(28)=1$　又冲突

$H_2(28)=4$

H(26)=26 MOD 7=5

H(56)=56 MOD 7=0　冲突　$H_1(56)=1$　又冲突

$H_2(56)=0$　又冲突　$H_3(56)=4$　又冲突

$H_4(56)=2$

H(23)=23 MOD 7=2　冲突　$H_1(23)=3$

二次探测法的特点

- 优点：探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；
- 缺点：不能保证探测到散列表的所有地址。

0	1	2	3	4	5	6
14	15	56	23	28	26	

(3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间[1，m-1]的随机序列。

例2： 表长为11的哈希表中已填有关键字为17，60，29的记录，散列函数为H(key)=key MOD 11。现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

(1) H(38)=38 MOD 11=5　冲突

H1=(5+1) MOD 11=6　冲突

H2=(5+2) MOD 11=7　冲突

H3=(5+3) MOD 11=8　不冲突

(2) H(38)=38 MOD 11=5　冲突

H_1=(5+1^2) MOD 11=6　冲突

H_2=(5-1^2) MOD 11=4　不冲突

(3) H(38)=38 MOD 11=5　冲突

设伪随机数序列为9，则H_1=(5+9) MOD 11=3 不冲突

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

2 再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止。即：$H_i = RH_i(key)$　　$i = 1, 2, \ldots, k$

RH_i：一组不同的哈希函数。第一次发生冲突时，用RH_1计算，第二次发生冲突时，用RH_2计算…依此类推知道得到某个H_i不再冲突为止。

- 优点：不易产生冲突的“聚集”现象；
- 缺点：计算时间增加。

3 链地址法

方法：将所有关键字为同义词(散列地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针。

设散列表长为m，定义一个一维指针数组：

RecNode *linkhash[m]，其中RecNode是结点类型，每个分量的初值为空。凡散列地址为k的记录都插入到以linkhash[k]为头指针的链表中，插入位置可以在表头或表尾或按关键字排序插入。

例： 已知一组关键字(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)，哈希函数为：H(key)=key MOD 13，用链地址法处理冲突，如右图图9-17所示 。

优点：不易产生冲突的“聚集”；删除记录也很简单。

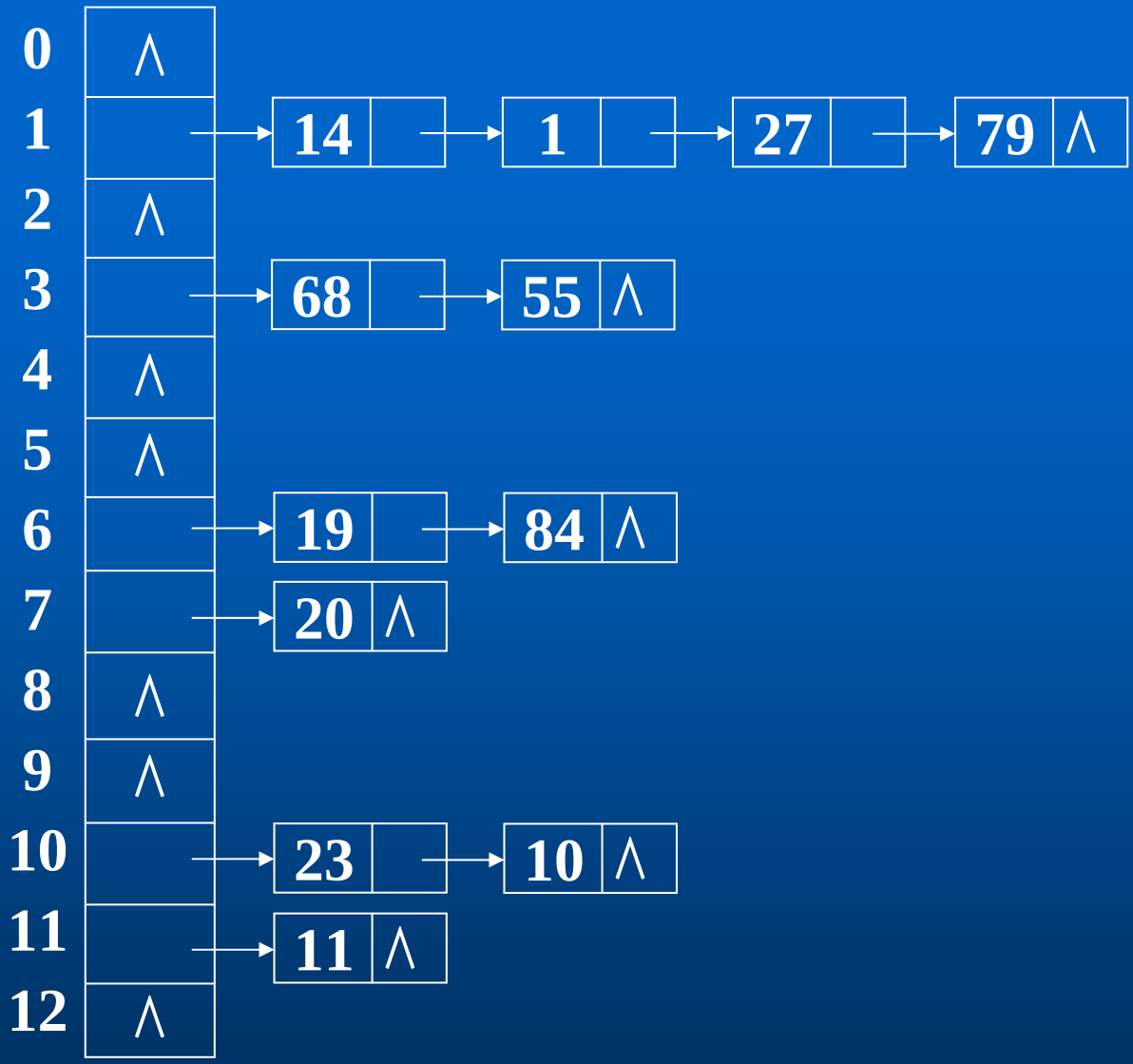


图9-17 用链地址法处理冲突的散列表

4 建立公共溢出区

方法：在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为m，设立基本散列表hashtable[m]，每个分量保存一个记录；溢出表overtable[m]，一旦某个记录的散列地址发生冲突，都填入溢出表中。

例： 已知一组关键字(15, 4, 18, 7, 37, 47)，散列表长度为7，哈希函数为：H(key)=key MOD 7，用建立公共溢出区法处理冲突。得到的基本表和溢出表如下：

Hashtable表：

散列地址	0	1	2	3	4	5	6
关键字	7	15	37		4	47	

overtable表：

溢出地址	0	1	2	3	4	5	6
关键字	18						

9.6.4 哈希查找过程及分析

1 哈希查找过程

哈希表的主要目的是用于快速查找，且插入和删除操作都要用到查找。由于散列表的特殊组织形式，其查找有特殊的方法。

设散列为HT[0...m-1]，散列函数为H(key)，解决冲突的方法为R(x, i)，则在散列表上查找定值为K的记录的过程如图9-18所示。

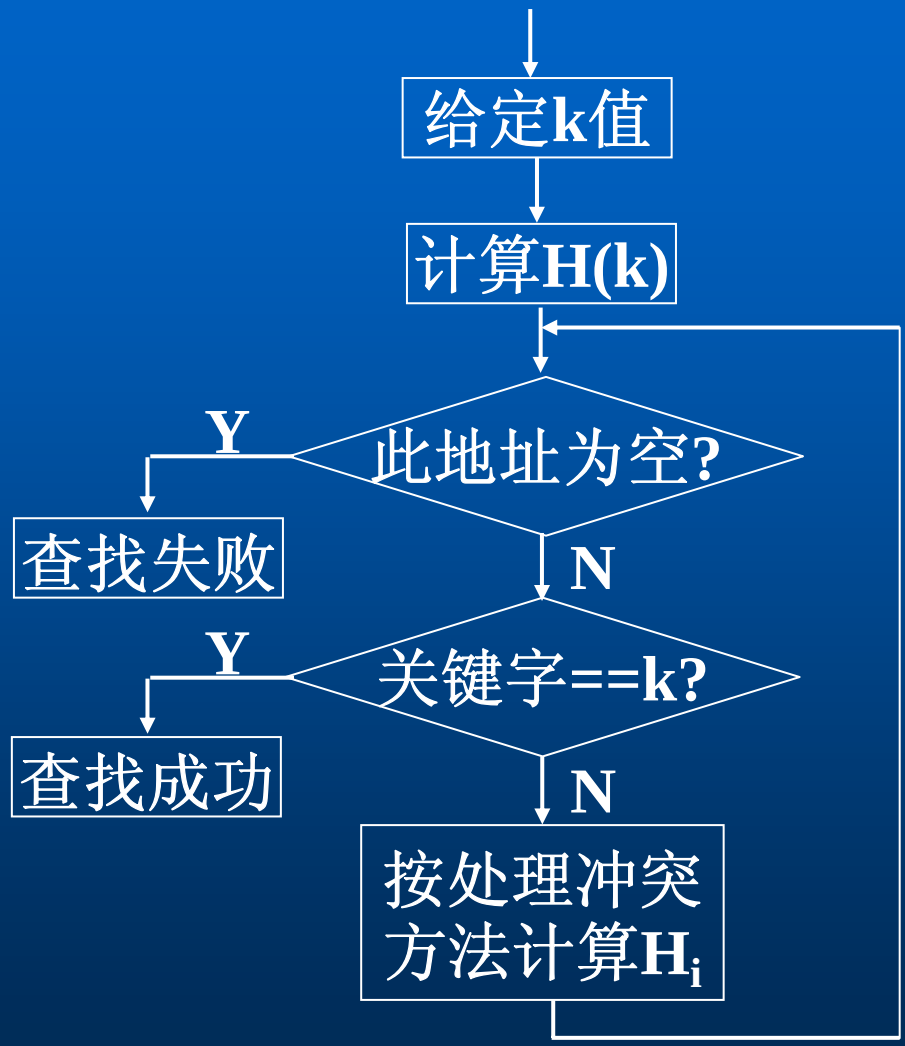


图9-18 散列表的查找过程

2 查找算法

```
#define NULLKEY  -1    /*  根据关键字类型定义空标识  */
typedef struct
  {  KeyType  key;     /*  关键字域  */
     otherType  otherinfo;  /*  记录的其它域  */
  }RecType;
int  Hash_search(RecType HT[], KeyType k, int m)
  /*   查找散列表HT中的关键字K,用开放定址法解决冲突
  */
  {  int h, j;
     h=h(k);
     while (j<m && !EQ(HT[h].key, NULLKEY) )
```

```
    {   if (EQ(HT[h].key, k) )   return(h) ;
      else h=R(k, ++j) ;
    }
  return(-1) ;
 }

#define M 15
typedef struct node
 {  KeyType key;
   struct node *link;
 }HNode;
```

```
HNode *hash_search(HNode *t[],
KeyType k)
  {  HNode *p;   int i;
     i=h(k);
     if (t[i]==NULL)    return(NULL);
     p=t[i];
     while(p!=NULL)
       if (EQ(p->key, k))  return(p);
       else  p=p->link;
     return(NULL);
  }      /*  查找散列表HT中的关键字K,用链地址法解决冲
  突  */
```

3 哈希查找分析

从哈希查找过程可见：尽管散列表在关键字与记录的存储地址之间建立了直接映象，但由于“冲突”，查找过程仍是一个给定值与关键字进行比较的过程，评价哈希查找效率仍要用**ASL**。

哈希查找时关键字与给定值比较的次数取决于：

- 哈希函数；
- 处理冲突的方法；
- 哈希表的填满因子α 。填满因子α的定义是：

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表长度}}$$

各种散列函数所构造的散列表的ASL如下：

(1) 线性探测法的平均查找长度是：

$$S_{nl成功} \approx \frac{1}{2} \times \left(1+\frac{1}{1-\alpha}\right)$$

$$S_{nl失败} \approx \frac{1}{2} \times \left(1+\frac{1}{(1-\alpha)^2}\right)$$

(2) 二次探测、伪随机探测、再哈希法的平均查找长度是：

$$S_{nl成功} \approx -\frac{1}{\alpha} \times \ln(1-\alpha)$$

$$S_{nl失败} \approx \frac{1}{1-\alpha}$$

(3) 用链地址法解决冲突的平均查找长度是：

$$S_{nl成功} \approx 1+\frac{\alpha}{2}$$

$$S_{nl失败} \approx \alpha+e^{-\alpha}$$

习 题 九

(1) 对于一个有n个元素的线性表，若采用顺序查找方法时的平均查找长度是什么？若结点是有序的，则采用折半查找法是的平均查找长度是什么？

(2) 设查找表采用单链表存储，请分别写出对该表进行顺序查找的静态查找和动态查找的算法。

(3) 设二叉排序树中的关键字互不相同：则

① 最小元素无左孩子，最大元素无右孩子，此命题是否正确？

② 最大和最小元素一定是叶子结点吗？

③ 一个新结点总是插入在叶子结点上吗？

(4) 试比较哈希表构造时几种冲突处理方法的优点和缺点。

(5) 将关键字序列(10, 2, 26, 4, 18, 24, 21, 15, 8, 23, 5, 12, 14)依次插入到初态为空的二叉排序树中，请画出所得到的树T； 然后画出删除10之后的二叉排序树T_1； 若再将10插入到T_1中得到的二叉排序树T_2是否与T_1相同？请给出T_2的先序、中序和后序序列。

(6) 设有关键字序列为：(Dec, Feb, Nov, Oct, June, Sept, Aug, Apr, May, July, Jan, Mar)，请手工构造一棵二叉排序树。该树是平衡二叉排序树？若不是，请为其构造一棵平衡二叉排序树。

(7) 设关键字序列是(19, 14, 23, 01, 68, 84, 27, 55, 11, 34, 79)，散列表长度是11，散列函数是H(key)=key MOD 11，

① 采用开放地址法的线性探测方法解决冲突，请构造该关键字序列的哈希表。

② 采用开放地址法的二次探测方法解决冲突，请构造该关键字序列的哈希表。

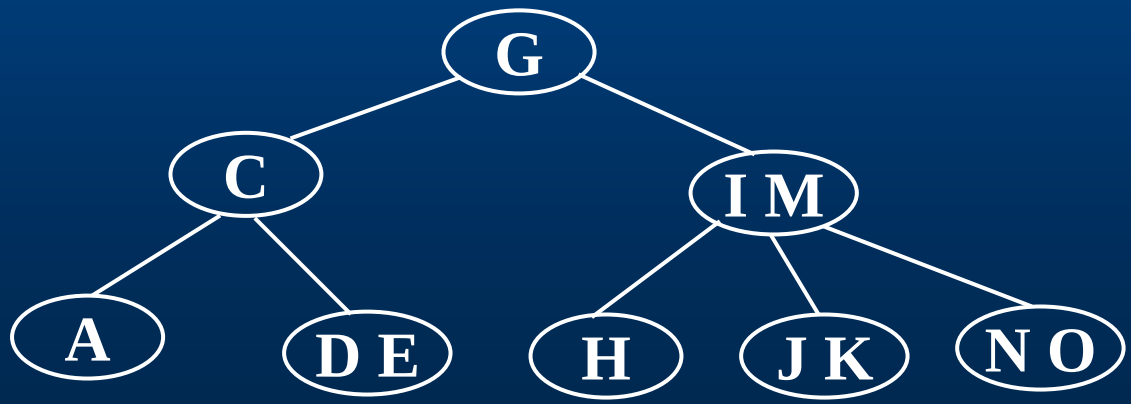
(8) 试比较线性索引和树形索引的优点和缺点。

⑼ 设关键字序列是(19, 24, 23, 17, 38, 04, 27, 51, 31, 34, 69)，散列表长度是11，散列函数是H(key)=key MOD 11，

① 采用开放地址法的线性探测方法解决冲突，请构造该关键字序列的哈希表。

② 求出在等概率情况下，该方法的查找成功和不成功的平均查找长度ASL。

⑽ 下图是一棵3阶B_树，请画出插入关键字B,L,P,Q后的树形。



第10章　内部排序

在信息处理过程中，最基本的操作是查找。从查找来说，效率最高的是折半查找，折半查找的前提是所有的数据元素(记录)是按关键字有序的。需要将一个无序的数据文件转变为一个有序的数据文件。

将任一文件中的记录通过某种方法整理成为按(记录)关键字有序排列的处理过程称为排序。

排序是数据处理中一种最常用的操作。

10.1 排序的基本概念

(1) 排序(Sorting)

排序是将一批(组)任意次序的记录重新排列成按关键字有序的记录序列的过程，其定义为：

给定一组记录序列：$\{R_1, R_2, \ldots, R_n\}$，其相应的关键字序列是$\{K_1, K_2, \ldots, K_n\}$。确定1，2，…n的一个排列$p_1, p_2, \ldots, p_n$，使其相应的关键字满足如下非递减(或非递增)关系：$K_{p_1} \leqslant K_{p_2} \leqslant \ldots \leqslant K_{p_n}$的序列$\{K_{p_1}, K_{p_2}, \ldots, K_{p_n}\}$，这种操作称为排序。

关键字K_i可以是记录R_i的主关键字，也可以是次关键字或若干数据项的组合。

- K_i是主关键字：排序后得到的结果是唯一的；
- K_i是次关键字：排序后得到的结果是不唯一的

。

(2) 排序的稳定性

若记录序列中有两个或两个以上关键字相等的记录：$K_i = K_j (i \neq j, i, j = 1, 2, \ldots n)$，且在排序前$R_i$先于$R_j (i < j)$，排序后的记录序列仍然是$R_i$先于$R_j$，称排序方法是稳定的，否则是不稳定的。

排序算法有许多，但就全面性能而言，还没有一种公认为最好的。每种算法都有其优点和缺点，分别适合不同的数据量和硬件配置。

评价排序算法的标准有：执行时间和所需的辅助空间，其次是算法的稳定性。

若排序算法所需的辅助空间不依赖问题的规模n，即空间复杂度是O(1)，则称排序方法是就地排序，否则是非就地排序。

(3) 排序的分类

待排序的记录数量不同，排序过程中涉及的存储器的不同，有不同的排序分类。

① 待排序的记录数不太多：所有的记录都能存放在内存中进行排序，称为内部排序；

② 待排序的记录数太多：所有的记录不可能存放在内存中， 排序过程中必须在内、外存之间进行数据交换，这样的排序称为外部排序。

(4) 内部排序的基本操作

对内部排序地而言，其基本操作有两种：

- 比较两个关键字的大小；
- 存储位置的移动：从一个位置移到另一个位置。

第一种操作是必不可少的；而第二种操作却不是必须的，取决于记录的存储方式，具体情况是：

① 记录存储在一组连续地址的存储空间：记录之间的逻辑顺序关系是通过其物理存储位置的相邻来体现，记录的移动是必不可少的；

② 记录采用链式存储方式：记录之间的逻辑顺序关系是通过结点中的指针来体现，排序过程仅需修改结点的指针，而不需要移动记录；

③ 记录存储在一组连续地址的存储空间：构造另一个辅助表来保存各个记录的存放地址(指针)：排序过程不需要移动记录，而仅需修改辅助表中的指针，排序后视具体情况决定是否调整记录的存储位置。

①比较适合记录数较少的情况；而②、③则适合记录数较少的情况。

为讨论方便，假设待排序的记录是以①的情况存储，且设排序是按升序排列的；关键字是一些可直接用比较运算符进行比较的类型。

待排序的记录类型的定义如下：

```
#define MAX_SIZE 100
Typedef int KeyType;
typedef struct RecType
  { KeyType key;          /* 关键字码 */
    infoType otherinfo;   /* 其他域 */
  }RecType;
typedef struct Sqlist
  { RecType R[MAX_SIZE];
    int length ;
  }Sqlist;
```

10.2 插入排序

采用的是以“玩桥牌者”的方法为基础的。即在考察记录R_i之前，设以前的所有记录$R_1, R_2, \ldots, R_{i-1}$已排好序，然后将$R_i$插入到已排好序的诸记录的适当位置。

最基本的插入排序是直接插入排序(Straight Insertion Sort)。

10.2.1　直接插入排序

1 排序思想

将待排序的记录R_i，插入到已排好序的记录表R_1, R_2 ,…., R_{i-1}中，得到一个新的、记录数增加1的有序表。 直到所有的记录都插入完为止。

设待排序的记录顺序存放在数组R[1…n]中，在排序的某一时刻，将记录序列分成两部分：

- R[1…i-1]：已排好序的有序部分；
- R[i…n]：未排好序的无序部分。

显然，在刚开始排序时，R[1]是已经排好序的。

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接插入排序的过程如下图10-1所示：

初始记录的关键字： [7] 4 -2 19 13 6

第一趟排序： [4 7] -2 19 13 6

第二趟排序： [-2 4 7] 19 13 6

第三趟排序： [-2 4 7 19] 13 6

第四趟排序： [-2 4 7 13 19] 6

第五趟排序： [-2 4 6 7 13 19]

图10-1 直接插入排序的过程

2 算法实现

```
void straight_insert_sort(Sqlist *L)
   {   int i, j ;
       for (i=2; i<=L->length; i++)
          {  L->R[0]=L->R[i]; j=i-1;     /*   设置哨兵   */
             while( LT(L->R[0].key, L->R[j].key) )
                {   L->R[j+1]=L->R[j];
                    j--;
                 }          /*   查找插入位置   */
             L->R[j+1]=L->R[0];      /*   插入到相应位置   */
          }
   }
```

3 算法说明

算法中的**R[0]**开始时并不存放任何待排序的记录，引入的作用主要有两个：

① 不需要增加辅助空间： 保存当前待插入的记录**R[i]**，**R[i]**会因为记录的后移而被占用；

② 保证查找插入位置的内循环总可以在超出循环边界之前找到一个等于当前记录的记录，起“哨兵监视”作用，避免在内循环中每次都要判断**j**是否越界。

4 算法分析

(1) 最好情况：若待排序记录按关键字从小到大排列(正序)，算法中的内循环无须执行，则一趟排序时：关键字比较次数1次，记录移动次数2次(R[i]→R[0], R[0]→R[j+1])。

则整个排序的关键字比较次数和记录移动次数分别是：

比较次数：$\sum_{i=2}^{n} 1=n-1$　　移动次数：$\sum_{i=2}^{n} 2=2(n-1)$

(2) 最坏情况：若待排序记录按关键字从大到小排列(逆序)，则一趟排序时：算法中的内循环体执行i-1，关键字比较次数i次，记录移动次数i+1。

则就整个排序而言：

比较次数：$\sum_{i=2}^{n} i = \frac{(n-1)(n+1)}{2}$

移动次数：$\sum_{i=2}^{n} (i+1) = \frac{(n-1)(n+4)}{2}$

一般地，认为待排序的记录可能出现的各种排列的概率相同，则取以上两种情况的平均值，作为排序的关键字比较次数和记录移动次数，约为$n^2/4$，则复杂度为$O(n^2)$。

10.2.2 其它插入排序

1 折半插入排序

当将待排序的记录R[i] 插入到已排好序的记录子表R[1...i-1]中时，由于R_1, R_2 ,..., R_{i-1}已排好序，则查找插入位置可以用“折半查找”实现，则直接插入排序就变成为折半插入排序。

(1) 算法实现

```
void Binary_insert_sort(Sqlist *L)
  {  int i, j, low, high, mid ;
     for (i=2; i<=L->length; i++)
        {  L->R[0]=L->R[i];      /*  设置哨兵  */
```

```
        low=1 ; high=i-1 ;
        while (low<=high)
            {  if ( LT(L->R[0].key, L->R[mid].key) )
                    high=mid-1 ;
                else   low=mid+1 ;
            }        /*  查找插入位置  */
        for (j=i-1; j>=high+1; j--)
        L->R[j+1]=L->R[j];
        L->R[high+1]=L->R[0];  /*  插入到相应位置  */
    }
}
```

从时间上比较，折半插入排序仅仅减少了关键字的比较次数，却没有减少记录的移动次数，故时间复杂度仍然为$O(n^2)$。

(2) 排序示例

设有一组关键字30, 13, 70, 85, 39, 42, 6, 20，采用折半插入排序方法排序的过程如图10-2所示：

图10-2 折半插入排序过程

2 2-路插入排序

是对折半插入排序的改进，以减少排序过程中移动记录的次数。附加n个记录的辅助空间，方法是：

① 另设一个和L->R同类型的数组d，L->R[1]赋给d[1]，将d[1]看成是排好序的序列中中间位置的记录；

② 分别将L->R[ ]中的第i个记录依次插入到d[1]之前或之后的有序序列中，具体方法：

- L->R[i].key<d[1].key： L->R[i]插入到d[1]之前的有序表中；
- L->R[i].key≥d[1].key： L->R[i]插入到d[1]之后的有序表中；

关键点：实现时将向量d看成是循环向量，并设两个指针first和final分别指示排序过程中得到的有序序列中的第一个和最后一个记录。

排序示例

设有初始关键字集合{49, 38, 65, 13, 97, 27, 76}，采用2-路插入排序的过程如右图10-3所示。

在2-路插入排序中，移动记录的次数约为n2/8。但当L->R[1]是待排序记录中关键字最大或最小的记录时，2-路插入排序就完全失去了优越性。

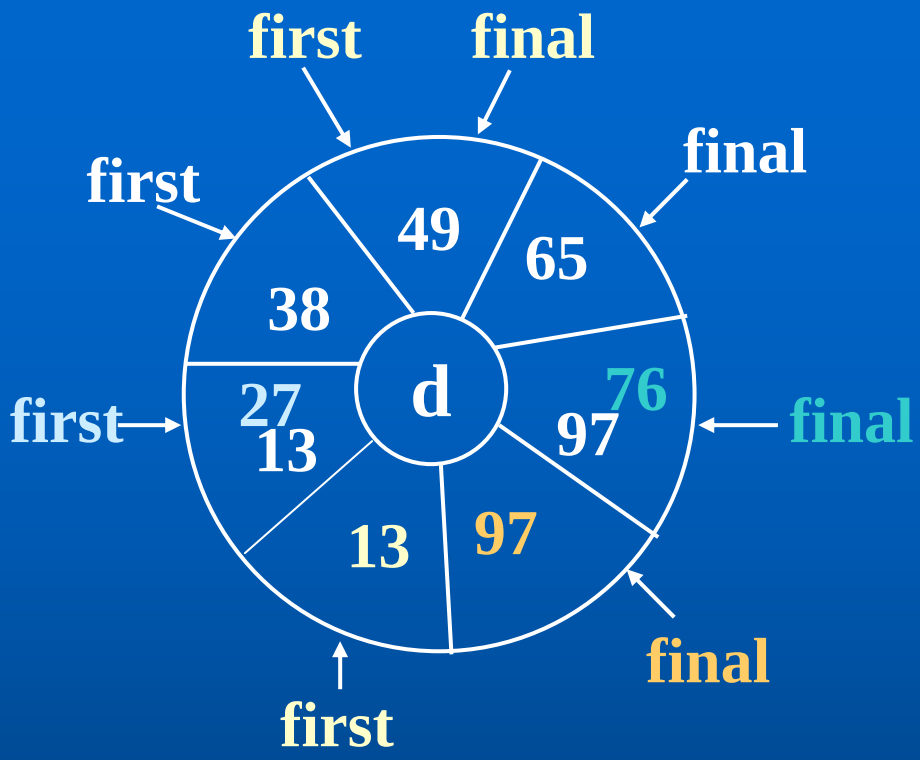


图10-3　2-路插入排序过程

3　表插入排序

前面的插入排序不可避免地要移动记录，若不移动记录就需要改变数据结构。附加n个记录的辅助空间，记录类型修改为：

```
typedef struct  RecNode
  {  KeyType  key ;
     infotype  otherinfo ;
     int *next;
  }RecNode ;
```

初始化：下标值为0的分量作为表头结点，关键字取为最大值，各分量的指针值为空；

① 将静态链表中数组下标值为1的分量(结点)与表头结点构成一个循环链表；

② i=2，将分量R[i]按关键字递减插入到循环链表；

③ 增加i，重复②，直到全部分量插入到循环链表

例：设有关键字集合{49, 38, 65, 97, 76, 13, 27, 49}，采用表插入排序的过程如下图10-4所示。

	0	1	2	3	4	5	6	7	8	
	MAXINT	49	38	65	13	97	27	76	49	key域
	1	0	-	-	-	-	-	-	-	next域
i=2	MAXINT	49	38	65	13	97	27	76	49	
	2	0	1	-	-	-	-	-	-	
i=3	MAXINT	49	38	65	13	97	27	76	49	
	2	3	1	0	-	-	-	-	-	
i=4	MAXINT	49	38	65	13	97	27	76	49	
	4	3	1	0	2	-	-	-	-	
i=5	MAXINT	49	38	65	13	97	27	76	49	
	4	3	1	5	2	0	-	-	-	

i=6

MAXINT	49	38	65	13	97	27	76	49
4	3	1	5	6	0	2	-	-

i=7

MAXINT	49	38	65	13	97	27	76	49
4	3	1	7	6	0	2	5	-

i=8

MAXINT	49	38	65	13	97	27	76	49
4	8	1	7	6	0	2	5	3

图10-4　表插入排序过程

和直接插入排序相比，不同的是修改2n次指针值以代替移动记录，而关键字的比较次数相同，故时间复杂度为$O(n^2)$。

表插入排序得到一个有序链表，对其可以方便地进行顺序查找，但不能实现随即查找。根据需要，可以对记录进行重排，记录重排详见P_{268}。

10.2.3 希尔排序

希尔排序(Shell Sort，又称缩小增量法)是一种分组插入排序方法。

1 排序思想

① 先取一个正整数$d_1(d_1<n)$作为第一个增量，将全部n个记录分成d_1组，把所有相隔d_1的记录放在一组中，即对于每个$k(k=1, 2, \ldots d_1)$，$R[k]$, $R[d_1+k]$, $R[2d_1+k]$, ...分在同一组中，在各组内进行直接插入排序。这样一次分组和排序过程称为一趟希尔排序；

② 取新的增量$d_2<d_1$，重复①的分组和排序操作；直至所取的增量$d_i=1$为止，即所有记录放进一个组中排序为止。

2 排序示例

设有10个待排序的记录，关键字分别为9，13，8，2，5，13，7，1，15，11，增量序列是5，3，1，希尔排序的过程如图10-5所示。

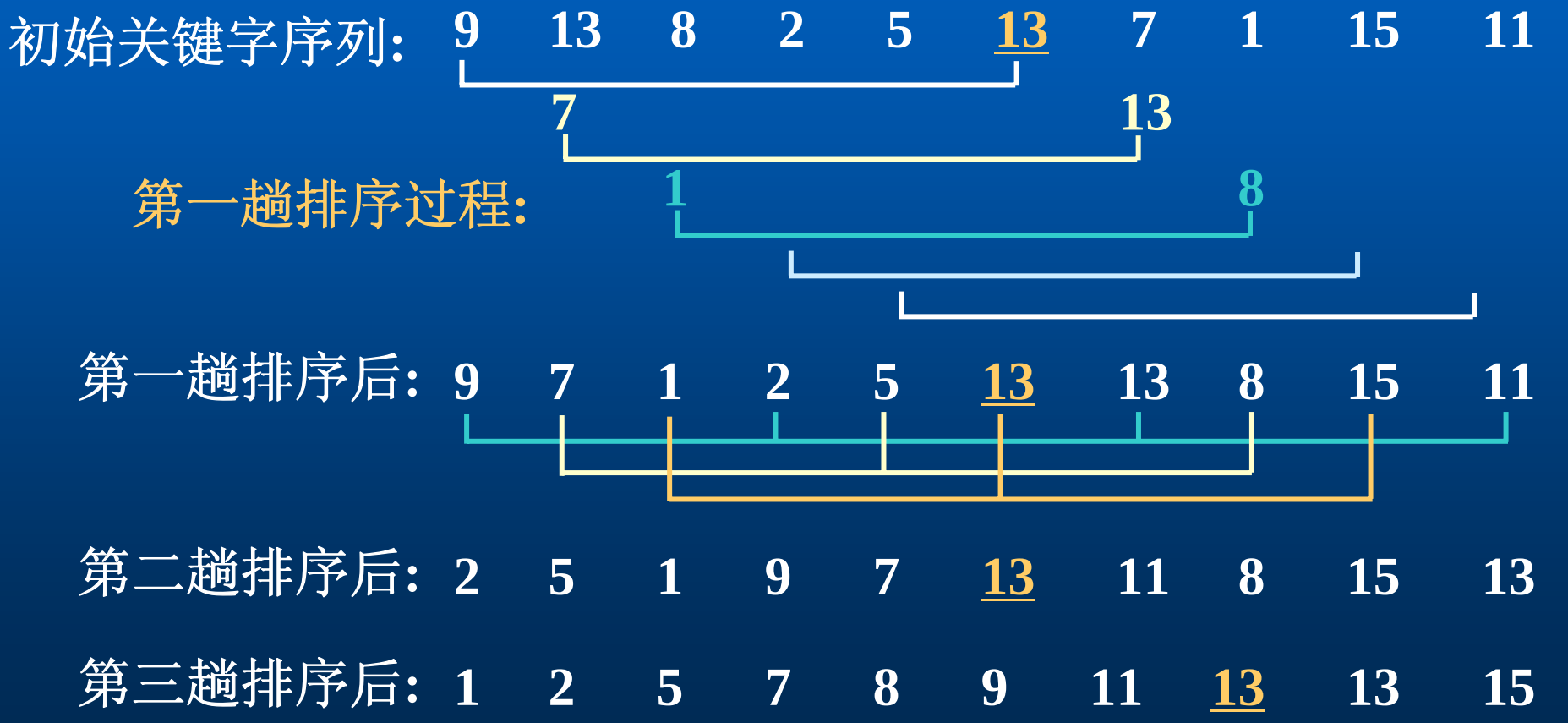


图10-5 希尔排序过程

3 算法实现

先给出一趟希尔排序的算法，类似直接插入排序。

```
void shell_pass(Sqlist *L, int d)
    /*  对顺序表L进行一趟希尔排序, 增量为d  */
   {  int j, k ;
      for (j=d+1; j<=L->length; j++)
         {  L->R[0]=L->R[j] ;        /*  设置监视哨兵  */
            k=j-d ;
            while (k>0&&LT(L->R[0].key, L->R[k].key) )
               {   L->R[k+d]=L->R[k] ; k=k-d ;   }
            L->R[k+j]=L->R[0] ;
         }
   }
```

然后在根据增量数组dk进行希尔排序。

```
void shell_sort(Sqlist *L, int dk[], int t)
   /*   按增量序列dk[0 … t-1],对顺序表L进行希尔排序   */
   {  int m ;
      for (m=0; m<=t; m++)
         shll_pass(L, dk[m]) ;
   }
```

希尔排序的分析比较复杂，涉及一些数学上的问题，其时间是所取的“增量”序列的函数。

希尔排序特点

子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。

希尔排序可提高排序速度，原因是：

- 分组后n值减小，n^2更小，而$T(n)=O(n^2)$,所以T(n)从总体上看是减小了；
- 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。

增量序列取法

- 无除1以外的公因子；
- 最后一个增量值必须为1。

10.3 快速排序

是一类基于交换的排序，系统地交换反序的记录的偶对，直到不再有这样一来的偶对为止。其中最基本的是冒泡排序(Bubble Sort)。

10.3.1 冒泡排序

1 排序思想

依次比较相邻的两个记录的关键字，若两个记录是反序的(即前一个记录的关键字大于后前一个记录的关键字)，则进行交换，直到没有反序的记录为止。

① 首先将L->R[1]与L->R[2]的关键字进行比较，若为反序(L->R[1]的关键字大于L->R[2]的关键字)，则交换两个记录；然后比较L->R[2]与L->R[3]的关键字，依此类推，直到L->R[n-1]与L->R[n]的关键字比较后为止，称为一趟冒泡排序，L->R[n]为关键字最大的记录。

② 然后进行第二趟冒泡排序，对前n-1个记录进行同样的操作。

一般地，第i趟冒泡排序是对L->R[1 … n-i+1]中的记录进行的，因此，若待排序的记录有n个，则要经过n-1趟冒泡排序才能使所有的记录有序。

2 排序示例

设有9个待排序的记录，关键字分别为23, 38, 22, 45, 23, 67, 31, 15, 41，冒泡排序的过程如图10-6所示。

3 算法实现

```
#define FALSE 0
#define TRUE 1
```

初始关键字序列:	23	38	22	45	23	67	31	15	41
第一趟排序后:	23	22	38	23	45	31	15	41	67
第二趟排序后:	22	23	23	38	31	15	41	45	67
第三趟排序后:	22	23	23	31	15	38	41	45	67
第四趟排序后:	22	23	23	15	31	38	41	45	67
第五趟排序后:	22	23	15	23	31	38	41	45	67
第六趟排序后:	22	15	23	23	31	38	41	45	67
第七趟排序后:	15	22	23	23	31	38	41	45	67

图10-6　冒泡排序过程

```
void Bubble_Sort(Sqlist *L)
   {  int j ,k , flag ;
      for (j=0; j<L->length; j++)      /*  共有n-1趟排序  */
         {  flag=TRUE ;
            for (k=1; k<=L->length-j; k++)  /*  一趟排序  */
               if (LT(L->R[k+1].key, L->R[k].key ) )
                  {   flag=FALSE ; L->R[0]=L->R[k] ;
                      L->R[k]=L->R[k+1] ;
                      L->R[k+1]=L->R[0] ;
                   }
               if  (flag==TRUE)  break ;
         }
   }
```

4 算法分析

时间复杂度

- 最好情况(正序)：比较次数：n-1；移动次数：0；
- 最坏情况(逆序)：

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

移动次数：$3\sum_{i=1}^{n-1}(n-i)=\frac{3n(n-1)}{2}$

故时间复杂度：$T(n)=O(n^2)$

空间复杂度：$S(n)=O(1)$

10.3.2 快速排序

1 排序思想

通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，再分别对这两部分记录进行下一趟排序，以达到整个序列有序。

2 排序过程

设待排序的记录序列是R[s...t]，在记录序列中任取一个记录(一般取R[s])作为参照(又称为基准或枢轴)，以R[s].key为基准重新排列其余的所有记录，方法是：

- 所有关键字比基准小的放R[s]之前；
- 所有关键字比基准大的放R[s]之后。

以R[s].key最后所在位置i作为分界，将序列R[s...t]分割成两个子序列，称为一趟快速排序。

3 一趟快速排序方法

从序列的两端交替扫描各个记录，将关键字小于基准关键字的记录依次放置到序列的前边；而将关键字大于基准关键字的记录从序列的最后端起，依次放置到序列的后边，直到扫描完所有的记录。

设置指针low，high，初值为第1个和最后一个记录的位置。

设两个变量i，j，初始时令i=low，j=high，以R[low].key作为基准(将R[low]保存在R[0]中)。

① 从j所指位置向前搜索：将R[0].key与R[j].key进行比较：

- 若R[0].key≤R[j].key：令j=j-1，然后继续进行比较，直到i=j或R[0].key>R[j].key为止；
- 若R[0].key>R[j].key：R[j]⇒R[i]，腾空R[j]的位置，且令i=i+1；

② 从i所指位置起向后搜索：将R[0].key与R[i].key进行比较：

- 若R[0].key≥R[i].key：令i=i+1，然后继续进行比较，直到i=j或R[0].key<R[i].key为止；

◆ 若R[0].key<R[i].key：R[i]⇒R[j]，腾空R[i]的位置，且令j=j-1；

③重复①、②，直至i=j为止，i就是R[0](基准)所应放置的位置。

4 一趟排序示例

设有6个待排序的记录，关键字分别为29, 38, 22, 45, 23, 67，一趟快速排序的过程如图10-7所示。

5 算法实现

⑴ 一趟快速排序算法的实现

```
int  quick_one_pass(Sqlist  *L , int low, int high)
    {  int i=low, j=high ;
```

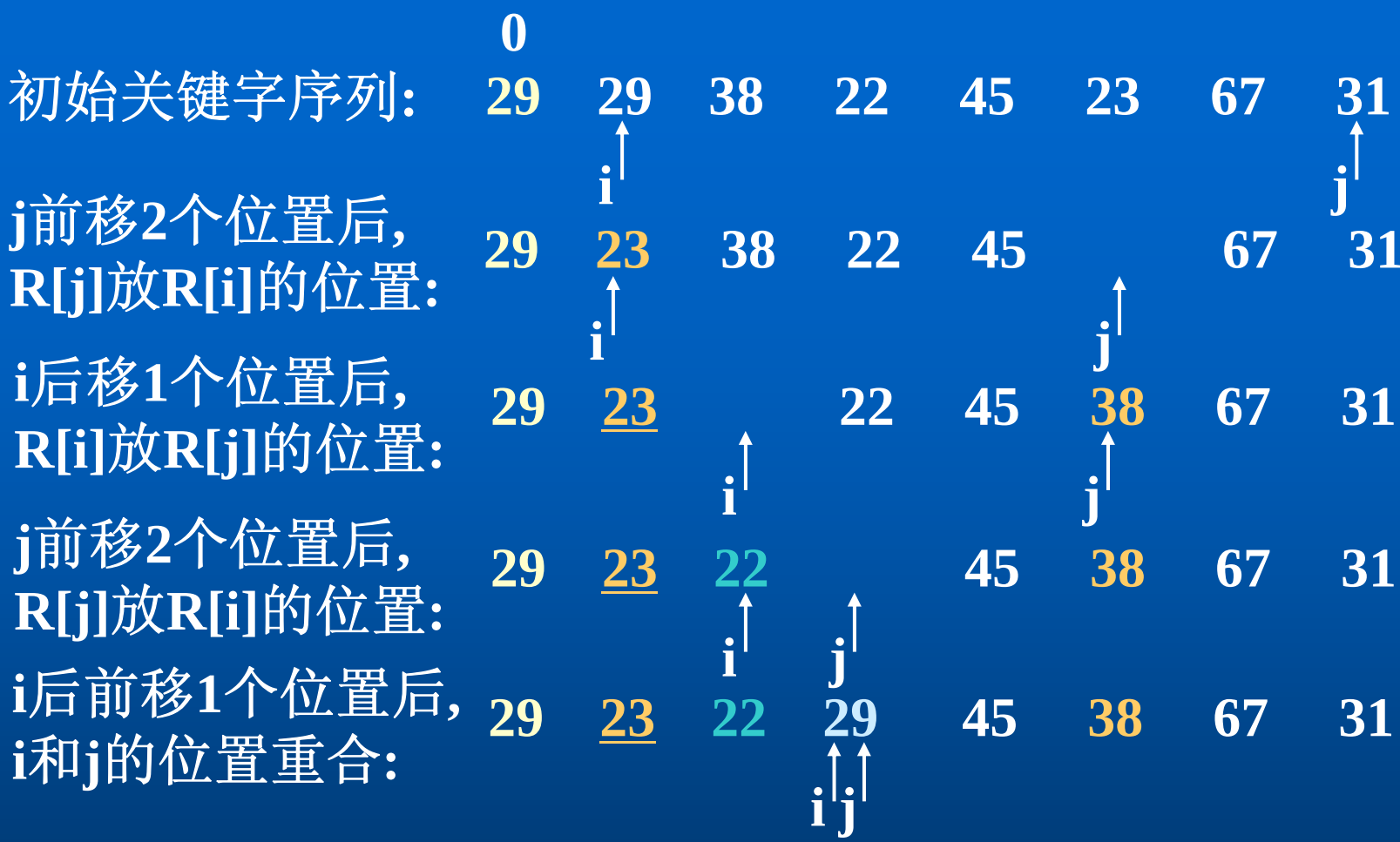


图10-7　一趟快速排序过程

```
    L->R[0]=L->R[i];     /*  R[0]作为临时单元和哨兵  */
    do
      {  while (LQ(L->R[0].key, L->R[j].key)&&(j>i))
            j--;
         if (j>i) { L->R[i]=L->R[j]; i++;  }
         while (LQ(L->R[i].key, L->R[0].key)&&(j>i))
            i++;
         if (j>i) { L->R[j]=L->R[i]; j--;  }
      } while(i!=j);   /*  i=j时退出扫描  */
    L->R[i]=L->R[0];
    return(i);
}
```

(2) 快速排序算法实现

当进行一趟快速排序后，采用同样方法分别对两个子序列快速排序，直到子序列记录个为1为止。

① 递归算法

```
void  quick_Sort(Sqlist  *L , int low, int high)
  {  int k ;
     if  (low<high)
        {  k=quick_one_pass(L, low, high);
           quick_Sort(L, low, k-1);
           quick_Sort(L, k+1, high);
        }     /*  序列分为两部分后分别对每个子序列排序
        */
  }
```

② 非递归算法

```
# define MAX_STACK 100
void quick_Sort(Sqlist *L , int low, int high)
  { int k , stack[MAX_STACK] , top=0;
    do { while (low<high)
         { k=quick_one_pass(L,low,high);
           stack[++top]=high ;
         stack[++top]=k+1 ;
                  /* 第二个子序列的上,下界分别入栈 */
           high=k-1 ;
         }
         if (top!=0)
          { low=stack[top--] ; high=stack[top--]
         ; }
```

```
    }while (top!=0&&low<high);
  }
```

6 算法分析

快速排序的主要时间是花费在划分上，对长度为k的记录序列进行划分时关键字的比较次数是k-1 。设长度为n的记录序列进行排序的比较次数为C(n)，则C(n)=n-1+C(k)+C(n-k-1) 。

- 最好情况：每次划分得到的子序列大致相等，则

$C(n) \leqslant n+2\times C(n/2)+C(n-k-1)$

$\leqslant n+2\times[n/2+2\times C((n/2)/2) \leqslant 2n+4\times C(n/4)$

$\leqslant \ldots$

$\leqslant h\times n+2^h\times C(n/2^h)$，当$n/2^h=1$时排序结束。

即$C(n) \leqslant n \times \log_2 n + n \times C(1)$，$C(1)$看成常数因子，

即$C(n) \leqslant O(n \times \log_2 n)$；

- **最坏情况**：每次划分得到的子序列中有一个为空，另一个子序列的长度为n-1。即每次划分所选择的基准是当前待排序序列中的最小(或最大)关键字。

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$　　即$C(n)=O(n^2)$

- **一般情况**：对n个记录进行快速排序所需的时间T(n)组成是：

① 对n个记录进行一趟划分所需的时间是：$n \times C$，C是常数；

② 对所得到的两个子序列进行快速排序的时间：

$T_{avg}(n)=C(n)+T_{avg}(k-1)+T_{avg}(n-k)$　　……⑴

若记录是随机排列的，k取值在1～n之间的概率相同，则：

$$T_{avg}(n)=n\times C+\frac{1}{n}\sum_{k=0}^{n}[T_{avg}(k-1)+T_{avg}(n-k)]$$

$$=n\times C+\frac{2}{n}\sum_{k=0}^{n-1}T_{avg}(k) \quad \cdots\cdots (2)$$

当n＞1时，用n-1代替(2)中的n，得到：

$$T_{avg}(n)=(n-1)\times C+\frac{2}{n}\sum_{k=0}^{n-2}T_{avg}(k) \quad \cdots\cdots (3)$$

∴ $nT_{avg}(n)-(n-1)T_{avg}(n-1)=(2n-1)\times C+2T_{avg}(n-1)$，即

$T_{avg}(n)=(n+1)/n\times T_{avg}(n-1)+(2n-1)/n\times C$

$<(n+1)/n\times T_{avg}(n-1)+2C$

$<(n+1)/n\times[n/(n-1)\times T_{avg}(n-2)+2C]+2C$

$$=(n+1)/(n-1)\times T_{avg}(n-2)+2(n+1)[1/n+1/(n+1)]\times C$$

$$< \cdots$$

$$\therefore\ T_{avg}(n)< \frac{n+1}{2}T_{avg}(1)+2(n+1)\times C[\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}+\frac{1}{n+1}]$$

只有1个记录的排序时间是一个常数，

∴ 快速排序的平均时间复杂度是：$T(n)=O(n^{\log_2 n})$

从所需要的附加空间来看，快速排序算法是递归调用，系统内用堆栈保存递归参数，当每次划分比较均匀时，栈的最大深度为$[\log 2n]+1$ 。

∴ 快速排序的空间复杂度是：$S(n)=O(\log 2n)$

从排序的稳定性来看，快速排序是不稳定的。

10.4 选择排序

选择排序(Selection Sort)的基本思想是：每次从当前待排序的记录中选取关键字最小的记录表，然后与待排序的记录序列中的第一个记录进行交换，直到整个记录序列有序为止。

10.4.1 简单选择排序

简单选择排序(Simple Selection Sort，又称为直接选择排序)的基本操作是：通过n-i次关键字间的比较，从n-i+1个记录中选取关键字最小的记录，然后和第i个记录进行交换，i=1, 2, … n-1 。

1 排序示例

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接选择排序的过程如下图10-8所示。

初始记录的关键字： 7 4 -2 19 13 6

第一趟排序：-2 4 7 19 13 6

第二趟排序：-2 4 7 19 13 6

第三趟排序：-2 4 6 19 13 7

第四趟排序：-2 4 6 7 13 19

第五趟排序：-2 4 6 7 13 19

第六趟排序：-2 4 6 7 13 19

图10-8 直接选择排序的过程

2 算法实现

```
void simple_selection_sort(Sqlist *L)
   {  int m, n , k;
      for (m=1; m<L->length; m++)
         {  k=m ;
            for  (n=m+1; n<=L->length; n++)
               if  ( LT(L->R[n].key, L->R[k].key) )  k=n ;
            if  (k!=m)      /*  记录交换  */
               {  L->R[0]=L->R[m]; L->R[m]=L->R[k];
                  L->R[k]=L->R[0];
                }
         }
   }
```

3 算法分析

整个算法是二重循环：外循环控制排序的趟数，对n个记录进行排序的趟数为n-1趟；内循环控制每一趟的排序。

进行第i趟排序时，关键字的比较次数为n-i，则：

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

∴ 时间复杂度是：$T(n)=O(n^2)$

空间复杂度是：$S(n)=O(1)$

从排序的稳定性来看，直接选择排序是不稳定的。

10.4.2 树形选择排序

借助“淘汰赛”中的对垒就很容易理解树形选择排序的思想。

首先对n个记录的关键字两两进行比较，选取⌈n/2⌉个较小者；然后这⌈n/2⌉个较小者两两进行比较，选取⌈n/4⌉个较小者... 如此重复，直到只剩1个关键字为止。

该过程可用一棵有n个叶子结点的完全二叉树表示，如图10-9所示。

每个枝结点的关键字都等于其左、右孩子结点中较小的关键字，根结点的关键字就是最小的关键字。

输出最小关键字后，根据关系的可传递性，欲选取次小关键字，只需将叶子结点中的最小关键字改为“最大值”，然后重复上述步骤即可。

含有n个叶子结点的完全二叉树的深度为$\lceil \log_2 n \rceil + 1$，则总的时间复杂度为$O(n^{\log_2 n})$。

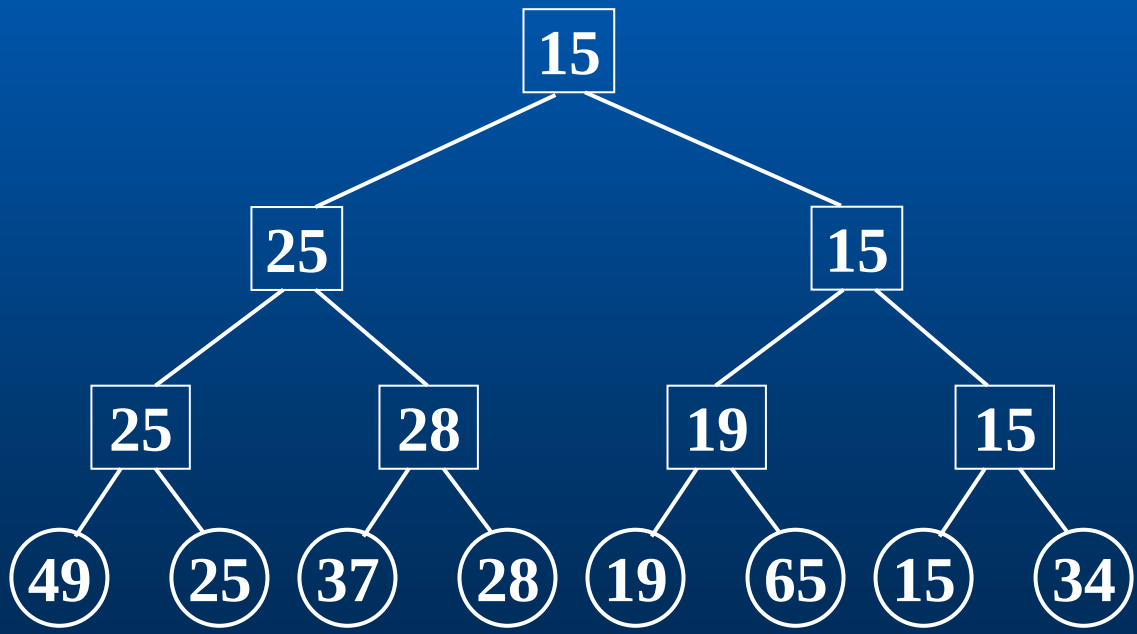


图10- 9 “淘汰赛”过程示意图

10.4.3 堆排序

1 堆的定义

是n个元素的序列$H=\{k_1, k_2, \ldots k_n\}$，满足

$$\begin{cases} k_i \leqslant k_{2i} & \text{当} 2i \leqslant n \text{时} \\ k_i \leqslant k_{2i+1} & \text{当} 2i+1 \leqslant n \text{时} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geqslant k_{2i} & \text{当} 2i \leqslant n \text{时} \\ k_i \geqslant k_{2i+1} & \text{当} 2i+1 \leqslant n \text{时} \end{cases}$$

其中： $i=1,2, \ldots, \lfloor n/2 \rfloor$

由堆的定义知，堆是一棵以k_1为根的完全二叉树。若对该二叉树的结点进行编号(从上到下，从左到右)，得到的序列就是将二叉树的结点以顺序结构存放，堆的结构正好和该序列结构完全一致。

2 堆的性质

① 堆是一棵采用顺序存储结构的完全二叉树，k_1是根结点；

② 堆的根结点是关键字序列中的最小(或最大)值，分别称为小(或大)根堆；

③ 从根结点到每一叶子结点路径上的元素组成的序列都是按元素值(或关键字值)非递减(或非递增)的；

④堆中的任一子树也是堆。

利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中依次选取关键字最小(或最大)的记录，就可以实现对数据记录的排序，这种排序方法称为堆排序。

3 堆排序思想

① 对一组待排序的记录，按堆的定义建立堆；

② 将堆顶记录和最后一个记录交换位置，则前n-1个记录是无序的，而最后一个记录是有序的；

③ 堆顶记录被交换后，前n-1个记录不再是堆，需将前n-1个待排序记录重新组织成为一个堆，然后将堆顶记录和倒数第二个记录交换位置，即将整个序列中次小关键字值的记录调整(排除)出无序区；

④ 重复上述步骤，直到全部记录排好序为止。

结论：排序过程中，若采用小根堆，排序后得到的是非递减序列；若采用大根堆，排序后得到的是非递增序列。

堆排序的关键

① 如何由一个无序序列建成一个堆？

② 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

4 堆的调整——筛选

(1) 堆的调整思想

输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直到是叶子结点或其关键字值小于等于左、右子树的关键字的值，将得到新的堆。称这个从堆顶至叶子的调整过程为“筛选”，如图**10-10**所示。

注意：筛选过程中，根结点的左、右子树都是堆，因此，筛选是从根结点到某个叶子结点的一次调整过程。

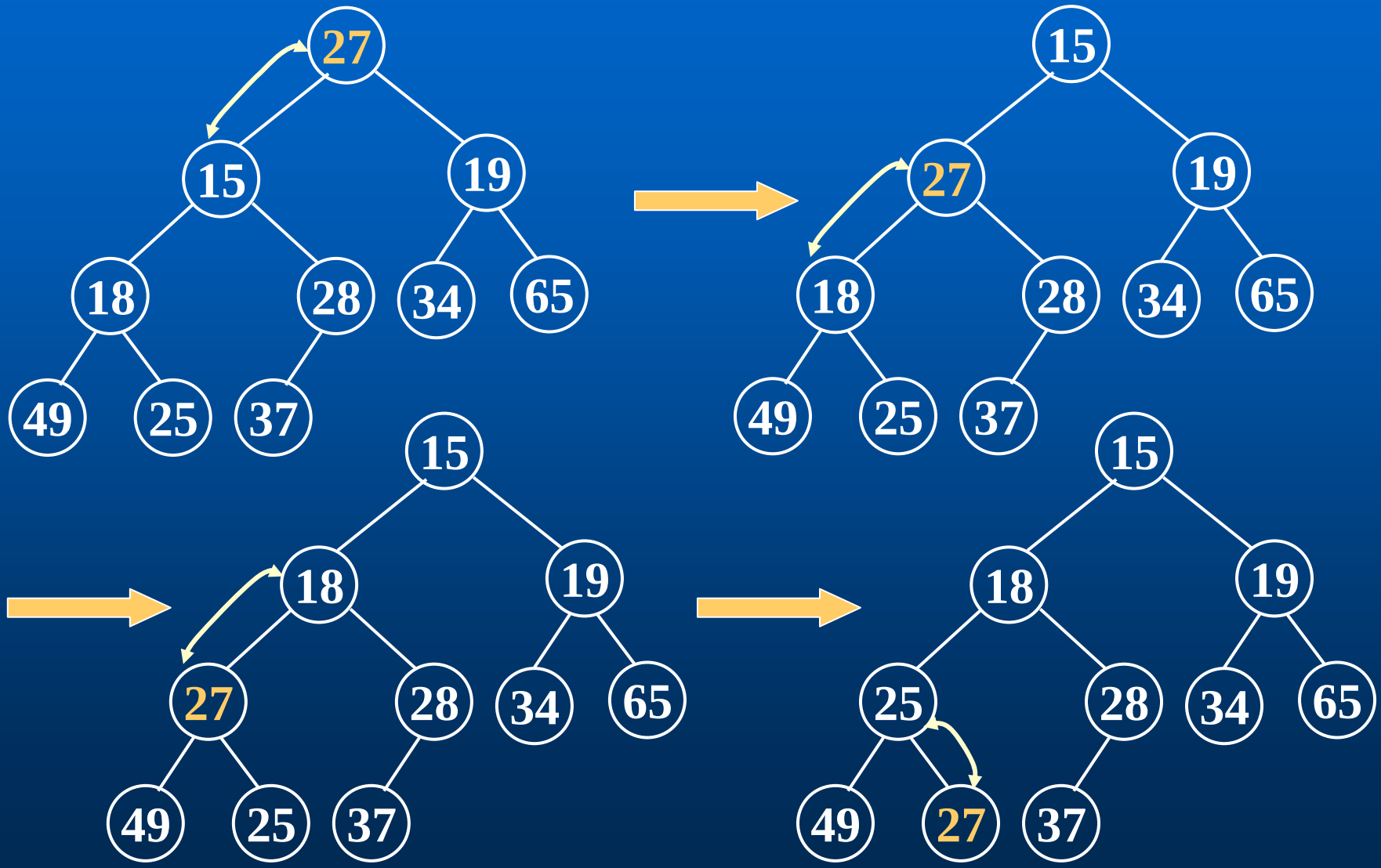


图10-10　堆的筛选过程

(2) 堆调整算法实现

```
void Heap_adjust(Sqlist *H, int s, int m)
   /*  H->R[s…m]中记录关键字除H->R[s].key均满足堆定义  */
   /*  调整H->R[s]的位置使之成为小根堆  */
   {  int j=s, k=2*j ;     /*  计算H->R[j]的左孩子的位置  */
      H->R[0]=H->R[j] ;       /*  临时保存H->R[j]  */
      for (k=2*j; k<=m; k=2*k)
         {  if ((k<m)&&(LT(H->R[k+1].key, H->R[k].key))
               k++ ;     /*  选择左、右孩子中关键字的最小者  */
            if  ( LT(H->R[k].key, H->R[0].key) )
               {   H->R[j]=H->R[k] ; j=k ; k=2*j  }
            else  break ;
         }
```

```
    H->R[j]=H->R[0] ;
  }
```

5 堆的建立

利用筛选算法，可以将任意无序的记录序列建成一个堆，设R[1],R[2], …,R[n]是待排序的记录序列。

将二叉树的每棵子树都筛选成为堆。只有根结点的树是堆。第⌊n/2⌋个结点之后的所有结点都没有子树，即以第⌊n/2⌋个结点之后的结点为根的子树都是堆。因此，以这些结点为左、右孩子的结点，其左、右子树都是堆，则进行一次筛选就可以成为堆。同理，只要将这些结点的直接父结点进行一次筛选就可以成为堆…。

只需从第⌊n/2⌋个记录到第1个记录依次进行筛选就可以建立堆。

可用下列语句实现：

```
for (j=n/2; j>=1; j--)
    Heap_adjust(R, j , n) ;
```

6 堆排序算法实现

堆的根结点是关键字最小的记录，输出根结点后，是以序列的最后一个记录作为根结点，而原来堆的左、右子树都是堆，则进行一次筛选就可以成为堆。

```
void  Heap_Sort(Sqlist *H)
  {  int j ;
     for (j=H->length/2; j>0; j--)
        Heap_adjust(H, j , H->length) ;  /*  初始建堆  */
```

```
    for (j=H->length/2; j>=1; j--)
        {  H->R[0]=H->R[1] ; H->R[1]=H->R[j] ;
           H->R[j]=H->R[0] ;  /*  堆顶与最后一个交换  */
           Heap_adjust(H, 1, j-1) ;
        }
}
```

7 算法分析

主要过程：初始建堆和重新调整成堆。设记录数为n，所对应的完全二叉树深度为h 。

- 初始建堆：每个非叶子结点都要从上到下做“筛选”。第i层结点数≤2i-1，结点下移的最大深度是h-i，而每下移一层要比较2次，则比较次数C1(n)为：

$$C_1(n) \leqslant 2\sum_{i=1}^{h-1}(2^{i-1}\times(h-i)) \leqslant 4(2^h-h-1)$$

∵ $h=\lfloor \log_2 n \rfloor+1$，　∴ $C_1(n) \leqslant 4(n-\log_2 n-1)$

◆ 筛选调整：每次筛选要将根结点“下沉”到一个合适位置。第i次筛选时：堆中元素个数为n-i+1；堆的深度是$\lfloor \log_2(n-i+1) \rfloor+1$，则进行n-1次“筛选”的比较次数$C_2(n)$为：

$$C_2(n) \leqslant \sum_{i=1}^{n-1}(2\times \log_2(n-i+1))$$

∴ $C_2(n)<2n\log_2 n$

∴ 堆排序的比较次数的数量级为：$T(n)=O(n\log_2 n)$；而附加空间就是交换时所用的临时空间，故空间复杂度为：$S(n)=O(1)$。

10.5 归并排序

归并(Merging)：是指将两个或两个以上的有序序列合并成一个有序序列。若采用线性表(无论是那种存储结构)易于实现，其时间复杂度为O(m+n)。

归并思想实例：两堆扑克牌，都已从小到大排好序，要将两堆合并为一堆且要求从小到大排序。

- 将两堆最上面的抽出(设为C_1，C_2)比较大小，将小者置于一边作为新的一堆(不妨设$C_1<C_2$)；再从第一堆中抽出一张继续与C_2进行比较，将较小的放置在新堆的最下面；
- 重复上述过程，直到某一堆已抽完，然后将剩下一堆中的所有牌转移到新堆中。

1 排序思想

① 初始时，将每个记录看成一个单独的有序序列，则n个待排序记录就是n个长度为1的有序子序列；

② 对所有有序子序列进行两两归并，得到⌈n/2⌉个长度为2或1的有序子序列——一趟归并；

③ 重复②，直到得到长度为n的有序序列为止。

上述排序过程中，子序列总是两两归并，称为2-路归并排序。其核心是如何将相邻的两个子序列归并成一个子序列。设相邻的两个子序列分别为：

{R[k], R[k+1], …, R[m]}和{R[m+1], R[m+2],…, R[h]}，将它们归并为一个有序的子序列：

{DR[/], DR[/+1], …, DR[m], DR[m+1], …, DR[h] }

例：设有9个待排序的记录，关键字分别为23，38，22，45，23，67，31，15，41，归并排序的过程如图10-11所示。

图10-11 归并排序过程

归并的算法

```
void Merge(RecType R[], RecType DR[],
int k, int m, int h)
  {  int p, q, n ; p=n=k, q=m+1 ;
     while ((p<=m)&&(q<=h))
        {   if (LQ(R[p].key, R[q].key) )  /*  比较两个子序列 */
              DR[n++]=R[p++] ;
            else   DR[n++]=R[q++] ;
        }
     while (p<=m)     /*  将剩余子序列复制到结果序列中  */
        DR[n++]=R[p++] ;
     while (q<=h)  DR[n++]=R[q++] ;
  }
```

2 一趟归并排序

一趟归并排序都是从前到后，依次将相邻的两个有序子序列归并为一个，且除最后一个子序列外，其余每个子序列的长度都相同。设这些子序列的长度为d，则一趟归并排序的过程是：

从j=1开始，依次将相邻的两个有序子序列R[j...j+d-1]和R[j+d...j+2d-1]进行归并；每次归并两个子序列后，j后移动2d个位置，即j=j+2d；若剩下的元素不足两个子序列时，分以下两种情况处理：

① 剩下的元素个数>d：再调用一次上述过程，将一个长度为d的子序列和不足d的子序列进行归并；

② 剩下的元素个数≤d：将剩下的元素依次复制到归并后的序列中。

(1) 一趟归并排序算法

```
void Merge_pass(RecType R[], RecType DR[], int d, int n)
  {  int j=1 ;
     while ((j+2*d-1)<=n)
        {   Merge(R, DR, j, j+d-1, j+2*d-1) ;
            j=j+2*d ;
        }     /*   子序列两两归并   */
     if (j+d-1<n)     /*   剩余元素个数超过一个子序列长度d   */
        Merge(R, DR, j, j+d-1, n) ;
     else  Merge(R, DR, j, n, n) ;/*   剩余子序列复制   */
  }
```

(2) 归并排序的算法

开始归并时，每个记录是长度为1的有序子序列，对这些有序子序列逐趟归并，每一趟归并后有序子序列的长度均扩大一倍；当有序子序列的长度与整个记录序列长度相等时，整个记录序列就成为有序序列。算法是：

```
void Merge_sort(Sqlist *L, RecType DR[])
  {  int d=1 ;
     while(d<L->length)
       {   Merge_pass(L->R, DR, d, L->length) ;
         Merge_pass(DR, L->R, 2*d, L->length) ;
         d=4*d ;
       }
  }
```

3 算法分析

具有n个待排序记录的归并次数是$\log_2 n$，而一趟归并的时间复杂度为O(n)，则整个归并排序的时间复杂度无论是最好还是最坏情况均为$O(n\log_2 n)$。在排序过程中，使用了辅助向量DR，大小与待排序记录空间相同，则空间复杂度为O(n)。归并排序是稳定的。

10.6 基数排序

基数排序(Radix Sorting) 又称为桶排序或数字排序：按待排序记录的关键字的组成成分(或“位”)进行排序。

基数排序和前面的各种内部排序方法完全不同，不需要进行关键字的比较和记录的移动。借助于多关键字排序思想实现单逻辑关键字的排序。

10.6.1 多关键字排序

设有n个记录$\{R_1, R_2, \ldots, R_n\}$，每个记录$R_i$的关键字是由若干项(数据项)组成，即记录$R_i$的关键字Key是若干项的集合：$\{K_i^1, K_i^2, \ldots, K_i^d\}(d>1)$。

记录$\{R_1, R_2, \ldots, R_n\}$有序的，指的是$\forall i, j \in [1,n]$，$i<j$，若记录的关键字满足：

$\{K_i^1, K_i^2, \ldots K_i^d\} < \{K_j^1, K_j^2, \ldots K_j^d\}$，

即$K_i^p \leqslant K_j^p$ $(p=1, 2, \ldots d)$

多关键字排序思想

先按第一个关键字K^1进行排序，将记录序列分成若干个子序列，每个子序列有相同的K^1值；然后分别对每个子序列按第二个关键字K^2进行排序，每个子序列又被分成若干个更小的子序列；如此重复，直到按最后一个关键字K^d进行排序。

最后，将所有的子序列依次联接成一个有序的记录序列，该方法称为最高位优先(**M**ost **S**ignificant **D**igit first)。

另一种方法正好相反，排序的顺序是从最低位开始，称为最低位优先(**L**east **S**ignificant **D**igit first)。

10.6.2 链式基数排序

若记录的关键字由若干确定的部分(又称为“位”)组成，每一位(部分)都有确定数目的取值。对这样的记录序列排序的有效方法是基数排序。

设有n个待排序记录$\{R_1, R_2, \ldots, R_n\}$，(单)关键字是由d位(部分)组成，每位有r种取值，则关键字R[i].key可以看成一个d元组： $R[i].key=\{K_i^1, K_i^2, \ldots, K_i^d\}$ 。

基数排序可以采用前面介绍的MSD或LSD方法。以下以LSD方法讨论链式基数排序。

1 排序思想

⑴ 首先以静态链表存储n个待排序记录，头结点指针指向第一个记录结点；

⑵ 一趟排序的过程是：

① 分配： 按K^d值的升序顺序，改变记录指针，将链表中的记录结点分配到r个链表(桶)中，每个链表中所有记录的关键字的最低位(K^d)的值都相等，用f[i]、e[i]作为第i个链表的头结点和尾结点；

② 收集：改变所有非空链表的尾结点指针，使其指向下一个非空连表的第一个结点，从而将r个链表中的记录重新链接成一个链表；

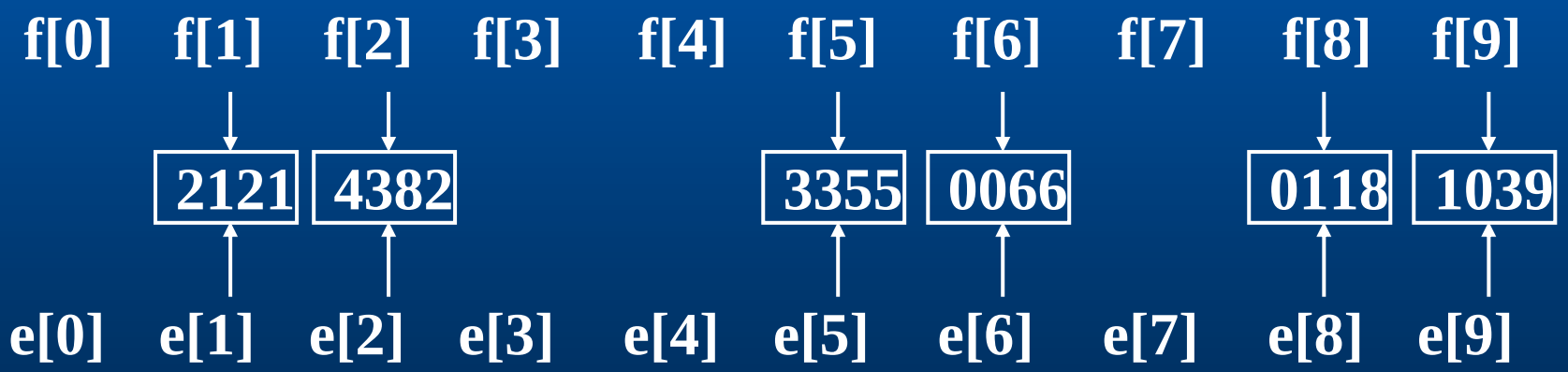
⑶ 如此依次按K^{d-1}, K^{d-2}, … K^1分别进行，共进行d趟排序后排序完成。

2 排序示例

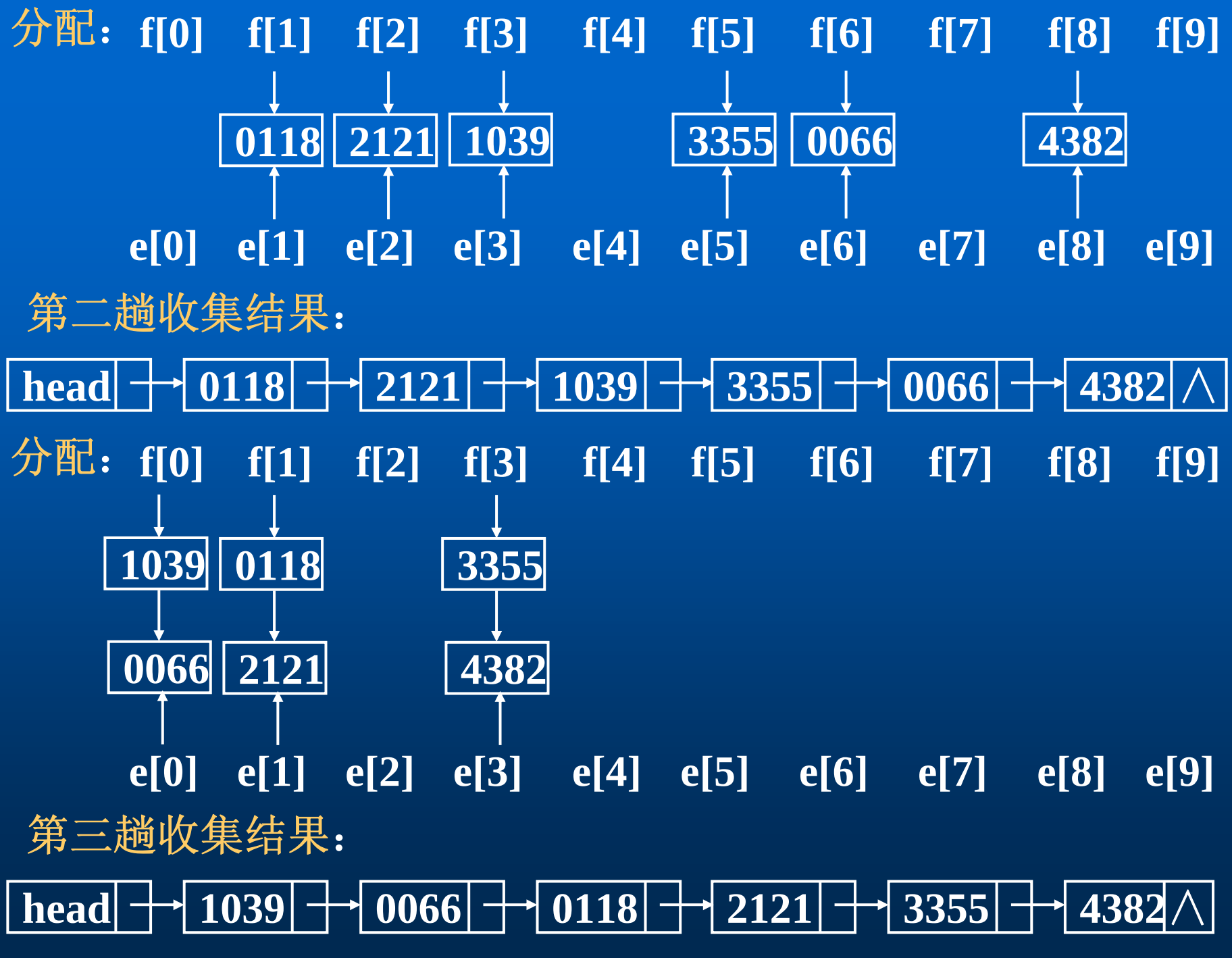
设有关键字序列为1039, 2121, 3355, 4382, 66, 118的一组记录，采用链式基数排序的过程如下图10-12所示。

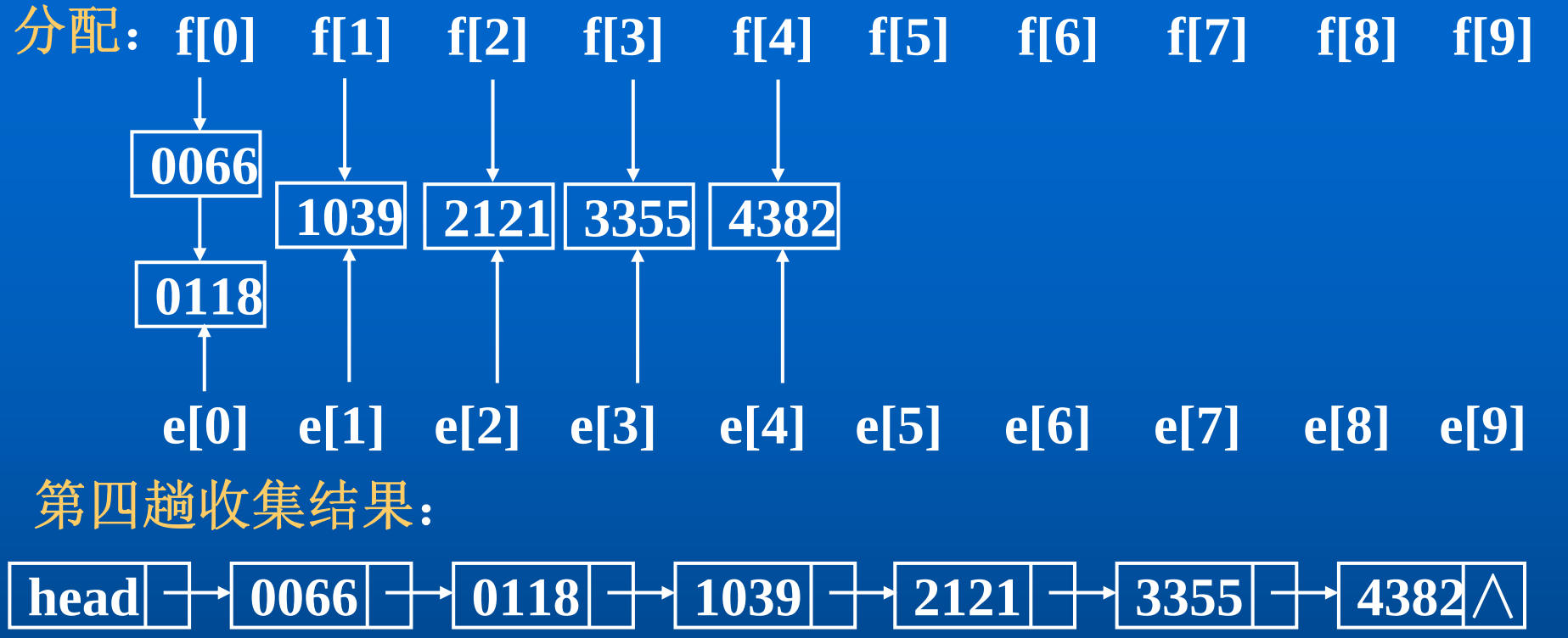
初始链表

分配：



第一趟收集结果：


分配：
f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]
0118 2121 1039 3355 0066 4382
e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]
第二趟收集结果：
head 0118 2121 1039 3355 0066 4382 ∧
分配：
f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]
1039 0118 3355
0066 2121 4382
e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]
第三趟收集结果：
head 1039 0066 0118 2121 3355 4382 ∧



第四趟收集结果：

图10-12 以LSD方法进行链式基数排序的过程

3 链式基数排序算法

为实现基数排序，用两个指针数组来分别管理所有的缓存(桶)，同时对待排序记录的数据类型进行改造，相应的数据类型定义如下：

```
#define BIT_key  8    /* 指定关键字的位数d  */
#define RADIX  10     /* 指定关键字基数r  */
typedef struct RecType
  { char key[BIT_key];      /* 关键字域  */
    infoType otheritems;
    struct RecType  *next;
  }SRecord, *f[RADIX] ,*e[RADIX];
    /*  桶的头尾指针数组   */
```

```
void Radix_sort(SRecord *head )
  {  int j, k, m ;
     SRecord  *p, *q, *f[RADIX], *e[RADIX] ;
     for (j=BIT_key-1; j>=0; j--)
           /*  关键字的每位一趟排序  */
        {  for (k=0; k<RADIX; k++)
              f[k]=e[k]=NULL ;   /*  头尾指针数组初始化  */
           p=head ;
           while (p!=NULL)      /*  一趟基数排序的分配  */
             {   m=ord(p->key[j]) ;   /*  取关键字的第j位kʲ  */
                 if  (f[m]==NULL)  f[m]=p ;
                 else  e[m]->next=p ;
                 p=p->next ;
```

```
        }
    head=NULL;   /* 以head作为头指针进行收集 */
    q=head;      /* q作为收集后的尾指针 */
    for (k=0; k<RADIX; k++)
      { if (f[k]!=NULL)  /* 第k个队列不空则收集 */
          { if (head!=NULL) q->next=f[k];
            else head=f[k];
            q=e[k];
          }
       }     /* 完成一趟排序的收集 */
    q->next=NULL;   /* 修改收集链尾指针 */
  }
}
```

4 算法分析

设有n个待排序记录，关键字位数为d，每位有r种取值。则排序的趟数是d；在每一趟中：

- 链表初始化的时间复杂度：O(r)；
- 分配的时间复杂度：O(n)；
- 分配后收集的时间复杂度：O(r)；

则链式基数排序的时间复杂度为：O(d(n+r))

在排序过程中使用的辅助空间是：2r个链表指针，n个指针域空间，则空间复杂度为：O(n+r)

基数排序是稳定的。

10.7 各种内部排序的比较

各种内部排序按所采用的基本思想(策略)可分为：插入排序、交换排序、选择排序、归并排序和基数排序，它们的基本策略分别是：

1 插入排序：依次将无序序列中的一个记录，按关键字值的大小插入到已排好序一个子序列的适当位置，直到所有的记录都插入为止。具体的方法有：直接插入、表插入、2-路插入和shell排序。

2 交换排序：对于待排序记录序列中的记录，两两比较记录的关键字，并对反序的两个记录进行交换，直到整个序列中没有反序的记录偶对为止。具体的方法有：冒泡排序、快速排序。

3 选择排序：不断地从待排序的记录序列中选取关键字最小的记录，放在已排好序的序列的最后，直到所有记录都被选取为止。具体的方法有：简单选择排序、堆排序。

4 归并排序：利用“归并”技术不断地对待排序记录序列中的有序子序列进行合并，直到合并为一个有序序列为止。

5 基数排序：按待排序记录的关键字的组成成分(“位”)从低到高(或从高到低)进行。每次是按记录关键字某一“位”的值将所有记录分配到相应的桶中，再按桶的编号依次将记录进行收集，最后得到一个有序序列。

各种内部排序方法的性能比较如下表。

表7-1 主要内部排序方法的性能

方法	平均时间	最坏所需时间	附加空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
Shell排序	$O(n^{1.3})$		$O(1)$	不稳定的
直接选择	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定的
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定的
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定的

讲稿中讨论的排序方法是在顺序存储结构上实现的，在排序过程中需要移动大量记录。当记录数很多、时间耗费很大时，可以采用静态链表作为存储结构。但有些排序方法，若采用静态链表作存储结构，则无法实现表排序。

选取排序方法的主要考虑因素：

- 待排序的记录数目n；
- 每个记录的大小；
- 关键字的结构及其初始状态；
- 是否要求排序的稳定性；
- 语言工具的特性；
- 存储结构的初始条件和要求；
- 时间复杂度、空间复杂度和开发工作的复杂程度的平衡点等。

习 题 十

(1) 回答下列各题：

① 从未排序序列中挑选元素，并将其依次放入到已排序序列中(初始时为空)的一端的方法是什么？

② 在待排序的元素基本有序的前提下，效率最高的排序方法是什么？

③ 从未排序序列中依次取出元素与已排序序列(初始时为空)中的元素进行比较，将其放入已排序序列的正确位置方法是什么？

④ 设有1000个元素， 希望采用最快的速度挑选出其中前10个最大的元素， 最好的方法是什么？

⑵ 若对关键字序列为(54, 37, 93, 25, 17, 68, 58, 41, 76)的一组记录进行快速排序时，递归调用使用的栈所能到达的最大深度是多少?共需递归调用多少次?其中第二次递归调用是对哪组记录进行排序?

⑶ 在堆排序，快速排序和归并排序中，若只从存储空间考虑，应选择哪种方法；若只从排序结果的稳定性考虑，应选择哪种方法；若只从平均情况下排序最快考虑，应选择哪种方法；

⑷ 设有关键字序列为(14, 17, 53, 35, 9, 32, 68, 41, 76, 23)的一组记录，请给出用希尔排序法(增量序列是5, 3, 1)排序时的每一躺结果。

⑸ 设有关键字序列为(14, 17, 53, 35, 9, 37, 68, 21, 46)的一组记录，请给出冒泡排序法排序时的每一躺结果。

⑹ 设有关键字序列为(14, 17, 53, 35, 9, 37, 68, 21, 46)的一组记录，利用快速排序法进行排序时，请给出以第一个记录为基准得到的一次划分结果。

⑺ 设关键字序列为(14, 17, 53, 35, 9, 37, 68, 21)的一组记录，请给出按非递增采用堆排序时的每一躺结果。

⑻ 设关键字序列为(314, 617, 253, 335, 19, 237, 464, 121, 46, 231, 176, 344)的一组记录，请给出采用基数排序时的每一躺结果。

⑼ 将哨兵放在R[n]中，被排序的记录存放在R[1...n-1]中，重写直接插入排序算法。

⑽ 实际中常采用单链表存储数据记录，请写出排序记录的结构的定义并修改。

第11章　文件与外部排序

在许多实际应用中，特别是数据处理时，都需要长期存储海量数据，这些数据通常以文件的方式组织并存储在外存。如何有效地管理这些数据，从而给使用者提供方便而高效的使用数据的方法称为文件管理。

在实际存取这些海量数据时，为了方便使用，往往以某种顺序排序后再存储在外存上，这种排序称为外部排序。在排序时由于一次不能将数据文件中的所有数据同时装入内存中进行，因此就必须研究如何对外存上的数据进行排序的技术。

11.1 文件的基本概念

1 文件的基本概念

(1) 数据项(Item或field)：数据文件中最小的基本单位，反映实体某一方面的特征—属性的数据表示。

(2) 记录(Record)：一个实体的所有数据项的集合。 用来标识一个记录的数据项集合(一个或多个)称为关键字项(Key)，关键字项的值称为关键字； 能唯一标识一个记录的关键字称为主关键字(Primary Key)，其它的关键字称为次关键字(Secondary Key)。

通常的记录指的是逻辑记录，是从用户角度所看到的对数据的表示和存取的方式。

文件存储在外存上，通常是以块(I/O读写的基本单位，称为物理记录)存取。

物理记录和逻辑记录之间的关系是：

① 一个物理记录存放一个逻辑记录；

② 一个物理记录包含多个逻辑记录；

③ 多个物理记录存放一个逻辑记录。

⑶ 文件(File)：大量性质相同的数据记录的集合。文件的所有记录是按某种排列顺序呈现在用户面前，这种排列顺序可以是按记录的关键字，也可以是按记录进入文件的先后等。则记录之间形成一种线性结构(逻辑上的)，称为文件的逻辑结构；文件在外存上的组织方式称为文件的物理结构。基本的物理结构有：顺序结构，链接结构，索引结构 。

(4) 文件的分类

(1) 按记录类型，可分为操作系统文件和数据库文件：

① 操作系统文件(流式文件)： 连续的字符序列(串)的集合；

② 数据库文件： 有特定结构(所有记录的结构都相同)的数据记录的集合。

(2) 按记录长度，可分为定长记录文件和不定长记录文件：

① 定长记录文件：文件中每个记录都有固定的数据项组成，每个数据项的长度都是固定的；

② 不定长记录文件：与定长记录文件相反。

2 文件的有关操作

文件是由大量记录组成的线性表，因此，对文件的操作主要是针对记录的，通常有：记录的检索、插入、删除、修改和排序，其中检索是最基本的操作。

(1) 检索记录

根据用户的要求从文件中查找相应的记录。

① 查找下一个记录：找当前记录的下一个逻辑记录；

② 查找第k个记录：给出记录的逻辑序号，根据该序号查找相应的记录；

③ 按关键字查找：给出指定的关键字值，查找关键字值相同或满足条件的记录。对数据库文件，有以下四种按关键字查找的方式：

- **简单匹配**：查找关键字的值与给定的值相等的记录；
- **区域匹配**：查找关键字的值在某个区域范围内的记录；
- **函数匹配**：给出关键字的某个函数，查找符合条件的记录；
- **组合条件匹配**：给出用布尔表达式表示的多个条件组合，查找符合条件的记录。

(2) 插入记录

将给定的记录插入到文件的指定位置。插入是首先要确定插入点的位置(检索记录)，然后才能插入。

(3) 删除记录

从文件中删除给定的记录。记录的删除有两种情况：

① 在文件中删除第k个记录；

② 在文件中删除符合条件的记录。

(4) 修改记录

对符合条件的记录，更改某些属性值。修改时首先要检索到所要修改的记录，然后才能修改。

(5) 记录排序

根据指定的关键字，对文件中的记录按关键字值的大小以非递减或非递增的方式重新排列(或存储)。

11.2 文件的组织方式

文件的组织方式指的是文件的物理结构。

11.2.1 顺序文件

记录按其在文件中的逻辑顺序依次进入存储介质。在顺序文件中，记录的逻辑顺序和存储顺序是一致的。

(1) 根据记录是否按关键字排序：可分为排序顺序文件和一般顺序文件；

(2) 根据逻辑上相邻的记录的物理位置关系：可分为连续顺序文件和链接顺序文件。

顺序文件类似于线性表的顺序存储结构，比较简单，适合于顺序存取的外存介质，但不适合随机处理。

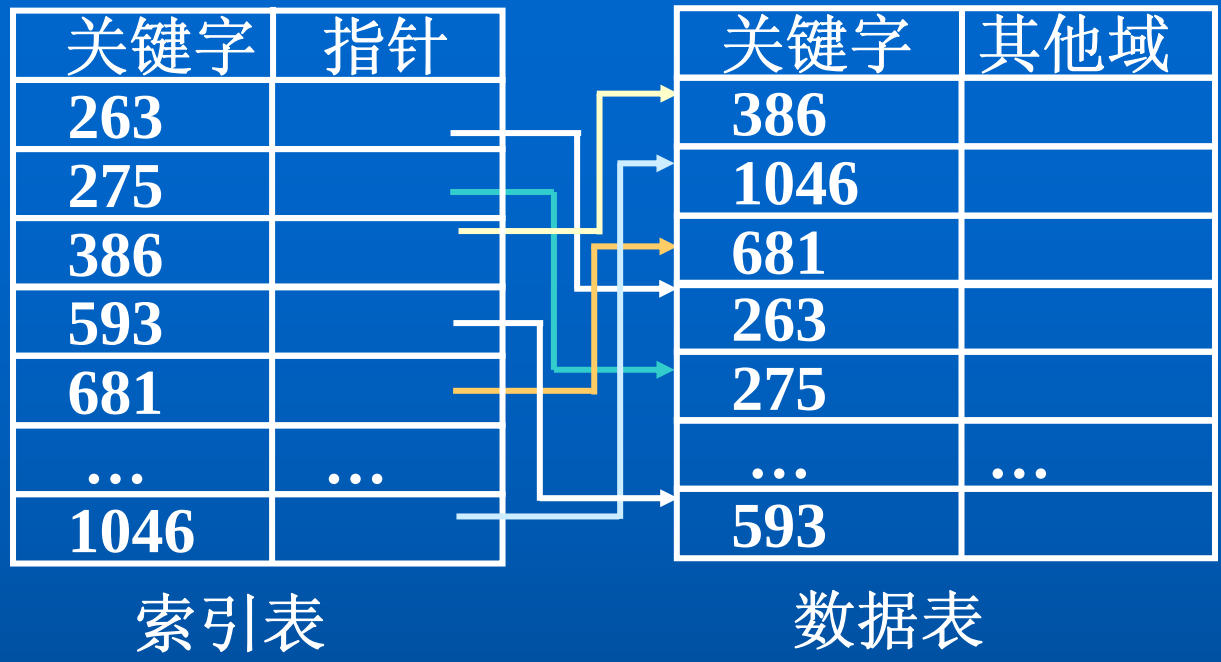
11.2.2 索引文件

索引技术是组织大型数据库的一种重要技术，索引是记录和记录存储地址之间的对照表。索引结构(称为索引文件)由索引表和数据表两部分，如图11-1所示。

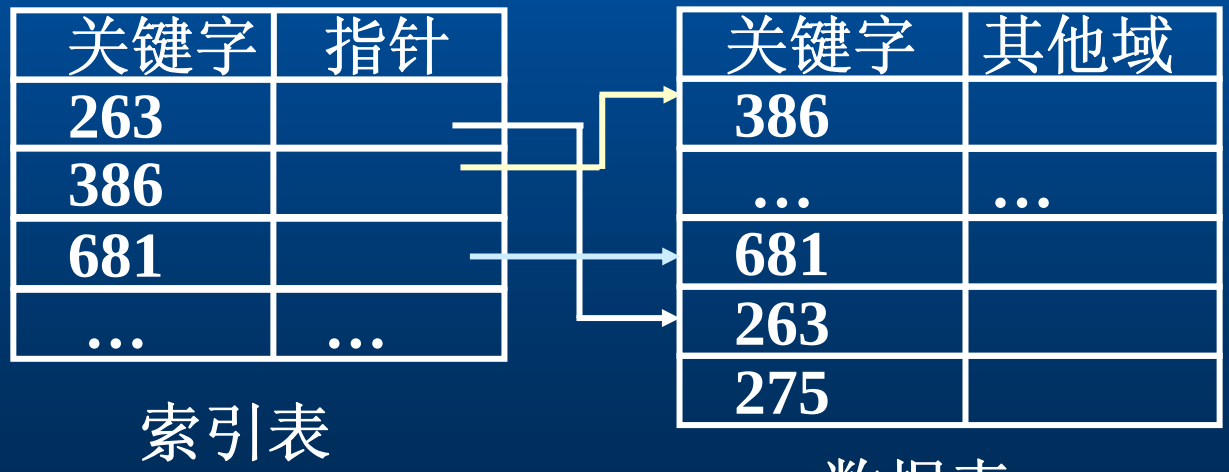
- 数据表：存储实际的数据记录；
- 索引表：存储记录的关键字和记录(存储)地址之间的对照表，每个元素称为一个索引项。

如果数据文件中的每一个记录都有一个索引项，这种索引称为稠密索引，否则，称为非稠密索引。

对于非稠密索引，通常将文件记录划分为若干块，块内记录可以无序，但块间必须有序。若块内记录是有序的，称为索引顺序文件，否则称为索引非顺序文件。对于索引非顺序文件，只需对每一块建立一个索引项。



(a) 稠密索引文件



(b) 非稠密索引文件

图11-1 索引结构的基本形式

对于稠密索引，可以根据索引项直接查找到记录的位置。若在索引表中采用顺序查找，查找时间复杂度为O(n)；若采用折半查找，查找时间复杂度为$O(\log_2 n)$。

对于稠密索引，索引项数目与数据表中记录数相同，当索引表很大时，检索记录需多次访问外存。

对于非稠密索引，查找的基本思想是：

首先根据索引找到记录所在块，再将该块读入到内存，然后再在块内顺序查找。

平均查找长度由两部分组成：块地址的平均查找长度L_b，块内记录的平均查找长度L_w，即$ASL_{bs}=L_b+L_w$

若将长度为n的文件分为b块，每块内有s个记录，则b=n/s。设每块的查找概率为1/b，块内每个的记录查找概率为1/b，则采用顺序查找方法时有：

$ASL_{bs}=L_b+L_w=(b+1)/2+(s+1)/2=(n/s+s)+1$

显然，当$s=n^{1/2}$时，ASL_{bs}的值达到最小；

若在索引表中采用折半查找方法时有：

$ASL_{bs}=L_b+L_w=\log_2(n/s+1)+s/2$

如果文件中记录数很庞大，对非稠密索引而言，索引也很大，可以将索引表再分块，建立索引的索引，形成树形结构的多级索引，如后面将要介绍的ISAM文件和VSAM文件。

11.2.3 ISAM文件

ISAM(Indexed Sequential Access Method，顺序索引存取方法)，是专为磁盘存取设计的一种文件组织方式，采用静态索引结构，是一种三级索引结构的顺序文件。图11-2是一个磁盘组的结构图。

ISAM文件由基本文件、磁道索引、柱面索引和主索引组成。

基本文件按关键字的值顺序存放，首先集中存放在同一柱面上，然后再顺序存放在相邻柱面上；对于同一柱面，则按盘面的次序顺序存放。

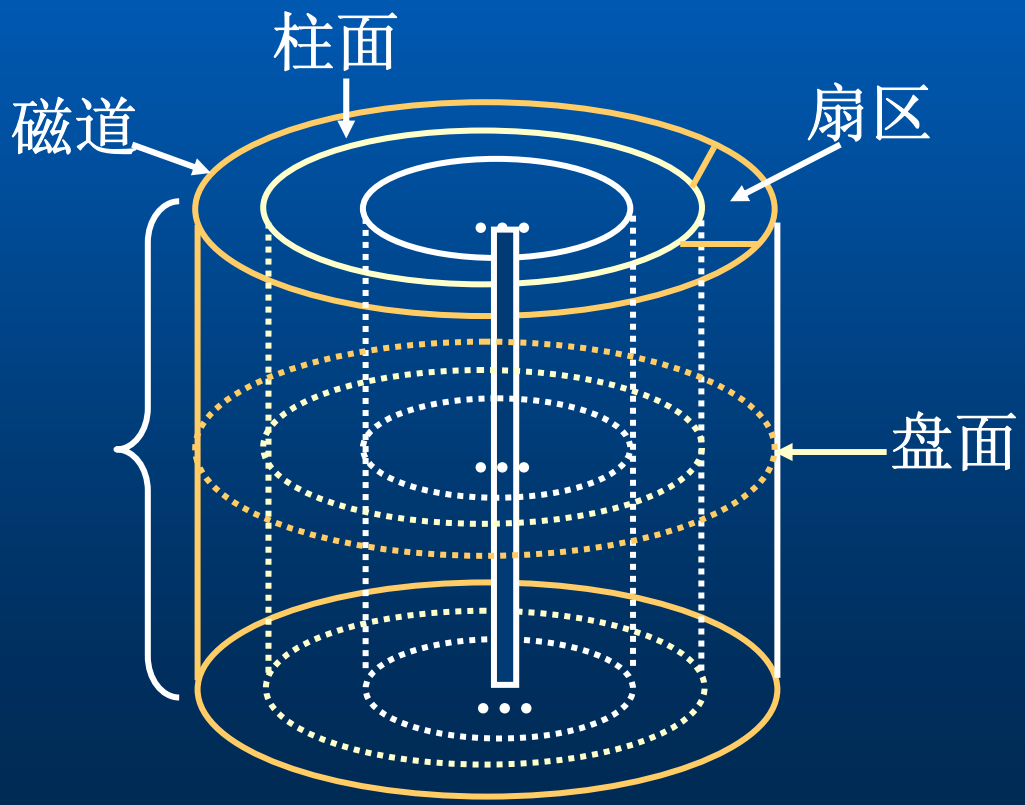
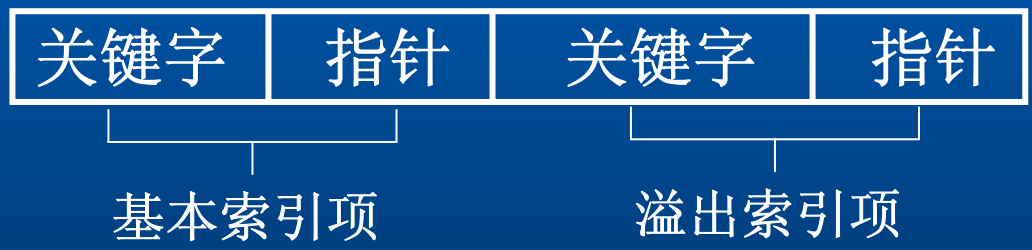


图11-2 一个磁盘组结构形式

在每个柱面上，还开辟了一个溢出区，存放从该柱面的磁道上溢出的记录。同一磁道上溢出的记录通常由指针相链接。

ISAM文件为每个磁道建立一个索引项，相同柱面的磁道索引项组成一个索引表，称为磁道索引，由基本索引项和溢出索引项组成，其结构是：



◆ 基本索引项：关键字域存放该磁道上的最大关键字；指针域存放该磁道的首地址。

◆ 溢出索引项：是为插入记录设置的。关键字域存放该磁道上溢出的记录的最大关键字；指针域存放溢出记录链表的头指针。

在磁道索引的基础上，又为文件所占用的柱面建立一个柱面索引，其结构是：

关键字	指针

关键字域存放该柱面上的最大关键字；指针域指向该柱面的第1个磁道索引项。

当柱面索引很大时，柱面索引本身占用很多磁道，又可为柱面索引建立一个主索引。则ISAM文件的三级索引结构如图11-3所示。

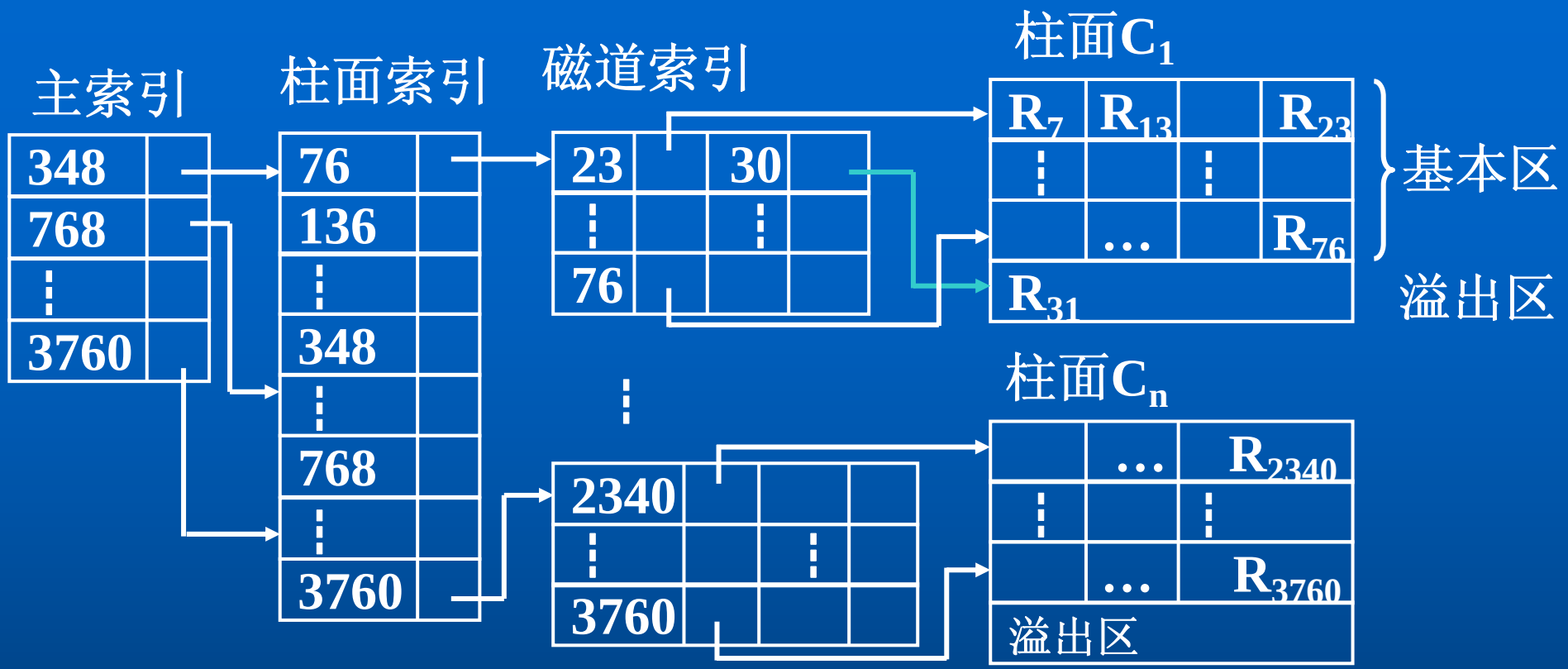


图11-3　ISAM文件结构示意图

1　ISAM文件的检索

根据关键字查找时，首先从主索引中查找记录所在的柱面索引块的位置；再从柱面索引块中查找磁道索引块的位置；然后再从磁道索引块中查找出该记录所在的磁道位置；最后从磁道中顺序查找要检索的记录。

2　记录的插入

首先根据待插入记录的关键字查找到相应位置；然后将该磁道中插入位置及以后的记录后移一个位置(若溢出，将该磁道中最后一个记录存入同一柱面的溢出区，并修改磁道索引)；最后将记录插入到相应位置。

3　记录的删除

只需找到要删除的记录，对其做删除标记，不移动记录。当经过多次插入和删除操作后，基本区有大量被删除的记录，而溢出区也可能有大量记录，则周期性地整理ISAM文件，形成一个新的ISAM文件。

4　ISAM文件的特点

- 优点：节省存储空间，查找速度快；
- 缺点：处理删除记录复杂，多次删除后存储空间的利用率和存取效率降低，需定期整理ISAM文件。

11.2.4 VSAM文件

VSAM(Virtual Storage Access Method，虚拟存取方法)，也是一种索引顺序文件组织方式，利用OS的虚拟存储器功能，采用的是基于B^+树的动态索引结构。

文件的存取不是以柱面、磁道等物理空间为存取单位，而是以逻辑空间——控制区间(Control Interval)和控制区域(Control Range)为存取单位。

一个VSAM文件由索引集、顺序集和数据集组成，如图11-4所示。

文件的记录都存放在数据集中，数据集又分成多个控制区间；VSAM进行I/O操作的基本单位是控制区间，由一组连续的存储单元组成，同一文件的控制区间大小相同；

图11-4 VSAM文件结构示意图

每个控制区间存放一个或多个逻辑记录，记录是按关键字值顺序存放在控制区间的前端，尾端存放记录的控制信息和控制区间的控制信息，如图11-5所示。

图11-5　控制区间的结构

顺序集是由B+树索引结构的叶子结点组成。每个结点存放若干个相邻控制区间的索引项，每个索引项存放一个控制区间中记录的最大关键字值和指向该控制区间的指针。顺序集中的每个结点及与它所对应的全部控制区间组成一个控制区域。

顺序集中的结点之间按顺序链接成一个链表，每个结点又在其上层建立索引，并逐层向上按B+树的形式建立多级索引。则顺序集中的每一个结点就是B+树的叶子结点；在顺序集之上的索引部分称为索引集。

在VSAM文件上既可以按B+树的方式实现记录的查找，又可以利用顺序集索引实现记录顺序查找。

VSAM文件中没有溢出区，解决方法是留出空间：

- 每个控制区间中留出空间；
- 每个控制区域留出空的控制空间，并在顺序集的索引中指出。

1 记录的插入

首先根据待插入记录的关键字查找到相应的位置：

- 若该控制区间有可用空间：将关键字大于待插入记录的关键字的记录全部后移一个位置，在空出的位置存放待插入记录；
- 若控制区间没有可用空间：利用同一控制区域的一个空白控制空间进行区间分裂，将近一半记录移到新的控制区间中，并修改顺序集中相应的索引，插入新的记录；

◆ 若控制区域中没有空白控制空间：则开辟一个新的控制区域，进行控制区间域分裂和相应的顺序集中的结点分裂。也可按B^+树的分裂方法进行。

2 记录的删除

先找到要删除的记录，然后将同一控制区间中比删除记录关键字大的所有记录逐个前移，覆盖要删除的记录。当一个控制区间的记录全部删除后，需修改顺序集中相应的索引项。

3 VSAM文件的特点

(1) 优点

◆ 能动态地分配和释放空间；

- 能保持较高的查询效率，无论是查询原有的还是后插入的记录，都有相同的查询速度；
- 能保持较高的存储利用率(平均75%)；
- 永远不需定期整理文件或对文件进行再组织。

(2) 缺点

- 为保证具有较好的索引结构，在插入或删除时索引结构本身也在变化；
- 控制信息和索引占用空间较多，因此，VSAM文件通常比较庞大。

基于B^+树的VSAM文件通常被作为大型索引顺序文件的标准。

11.2.5 散列文件

散列文件(直接存取文件)：利用散列存储方式组织的文件。类似散列表，即根据文件中记录关键字的特点，设计一个散列函数和冲突处理方法，将记录散列到存储介质上。

在散列文件中，磁盘上的记录是成组存放的，若干个记录组成一个存储单位，称为桶(Bucket)，同一个桶中的记录都是同义词(关键字的角度)。

设一个桶中能存放m个记录，当桶中已有m个同义词的记录时，要存放第m+1个同义词就“溢出”。冲突处理方法一般是拉链法。

检索记录时，先根据给定值求出散列桶地址，将基

桶的记录读入内存进行顺序查找，若找到关键字等于给定值的记录，则查找成功；否则，依次读入各溢出桶中的记录继续进行查找。

在散列文件中删除记录，是对记录加删除标记 。

散列文件的特点

(1) 优点

- 文件随机存取，记录不需进行排序；
- 插入、删除方便，存取速度快；
- 不需要索引区，节省存储空间。

(2) 缺点

- 不能进行顺序存取，只能按关键字随机存取；
- 检索方式仅限于简单查询。

11.2.6 多关键字文件

数据库文件常常是多关键字文件，多关键字文件的特点是不仅可以对主关键字进行各种查询，而且可以对次关键字进行各种查询。因此，对多关键字文件除了可按前面的方法组织主关键字索引外，还需要建立各个次关键字的索引。由于建立次关键字的索引的结构不同，多关键字文件有多重表文件和倒排文件。

1 多重表文件

多重表文件(Multilist Files)的特点是：记录按主关键字的顺序构成一个串联文件(物理上的)，并建立主关键字索引(称为主索引)；对每个次关键字都建立次关键字索引(称为次索引)，所有具有同一次关键字值的记录构成一个链表(逻辑上的)。

主索引一般是非稠密索引，其索引项一般有两项：主关键字值、头指针。

次索引一般是稠密索引，其索引项一般有三项：次关键字值、头指针、链表长度。头指针指向数据文件中具有该次关键字值的第1个记录，在数据文件中为各个次关键字增加一个指针域，指向具有相同次关键字值的下一个记录的地址。

对于任何次关键字的查询，都应首先查找对应的索引，然后顺着相应指针所指的方向查找属于本链表的记录。

多重表文件的特点

(1) 优点

易于构造和修改、查询方便。

(2) 缺点

插入和删除一个记录时，需要修改多个次关键字的指针(在链表中插入或删除记录)，同时还要修改各索引中的有关信息。

2 倒排文件

倒排文件又称逆转表文件。与多重表文件类似，可以处理多关键字查询。其差别是：

- 多重表文件：将具有相同关键字值的记录链接在一起，在数据文件中设有与各个关键字对应的指针域；
- 倒排文件：将具有相同关键字值的记录的地址收集在一起，并保存到相应的次关键字的索引项中，在数据文件中不设置对应的指针域，见$\mathbf{p}_{321}$。

次索引是次关键字倒排表，倒排表由次关键字值、记录指针(地址)，索引中保持次关键字的逻辑顺序。

倒排表文件的特点

(1) 优点

检索速度快，插入和删除操作比多重表文件简单。当插入一个记录时，只要将记录存入数据文件，并将其存储地址加入各倒排表中；删除也很方便。

(2) 缺点

倒排表维护比较困难。在同一索引表中，不同关键字值的记录数目不同，同一倒排表中的各项长度不等。

11.3 外部排序

当对数据记录量巨大的数据文件进行排序时，由于受到内存容量的限制，无法将所有数据记录一次全部读入到内存进行。排序过程中需要多次进行内、外存之间的数据交换。利用外存对数据文件进行排序称为外部排序。

11.3.1 外部排序方法

外部排序最基本的方法是归并。这种方法是由两个相对独立的阶段组成：

① 按内存(缓冲区)的大小，将n个记录的数据文件分成若干个长度为*l*的段或子文件，依次读入内存并选择有效的内部排序方法进行排序；然后将排好序的有序子文件重新写入到外存。子文件称为归并段或顺串。

② 采用归并的办法对归并段进行逐趟归并，使归并段的长度逐渐增大，直到最后合并成只有一个归并段的文件—排好序的文件。

1 外部排序的简单方法

归并排序有多种方法，最简单的就是2-路归并。

设有一个磁盘上的数据文件，共有100,000个记录(A_1，A_2，…，A_{100000})，页块长为200个记录，供排序使用的缓冲区可提供容纳1000个记录的空间，现要对该文件进行排序，排序过程可按如下步骤进行：

第一步：每次将5个页块(1000个记录)由外存读到内存，进行内排序，整个文件共得到10个初始顺串R_1~R_{10} (每一个顺串占5个页块)，然后把它们写回到磁盘上去，如图11-6所示。

第二步：然后两两归并，直到成为一个有序文件为止。

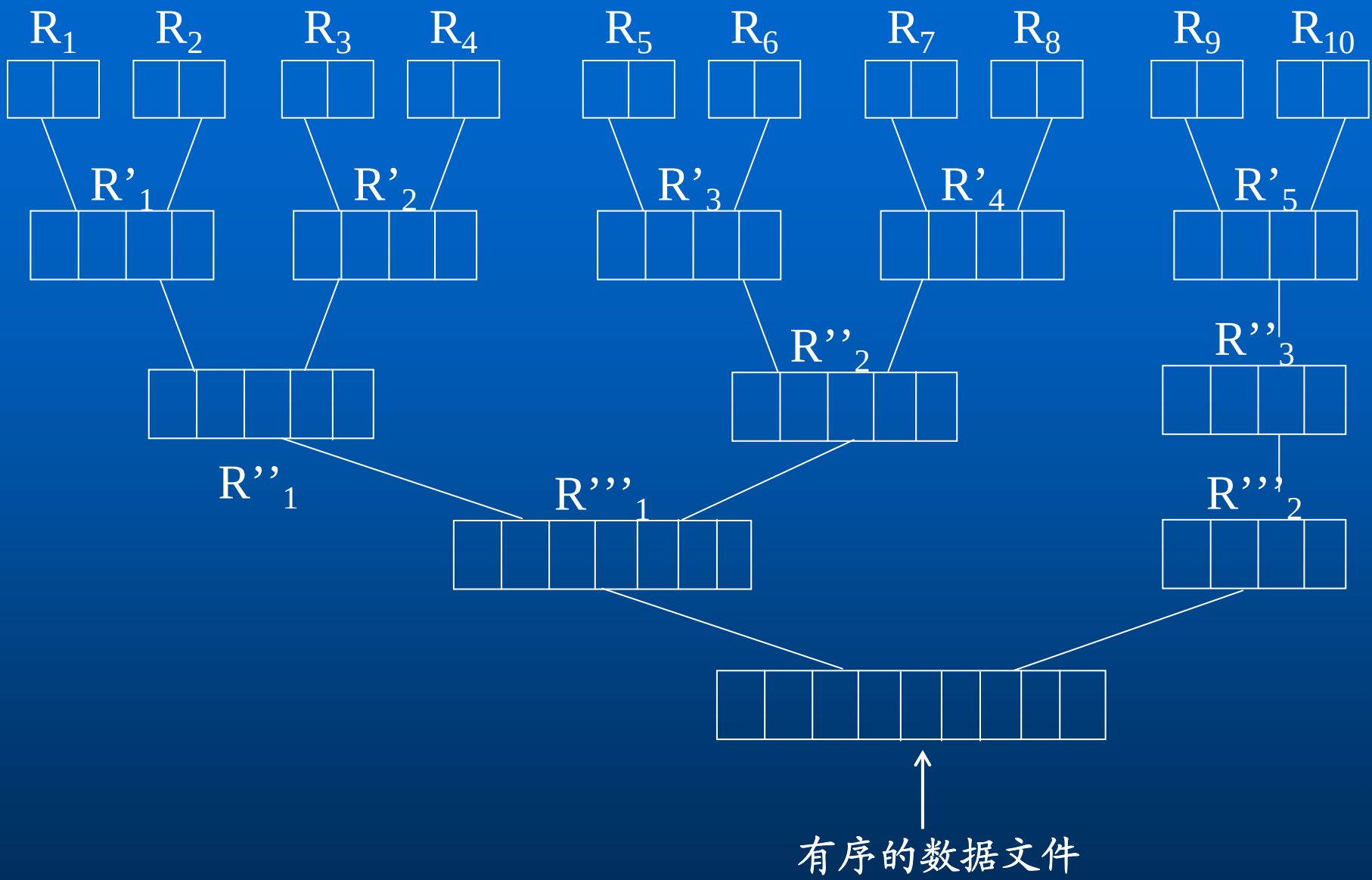


图11-6 外部排序过程示意图

由图可知，每趟归并由m个归并段得到 ⌈m/2⌉ 个归并段。

2 外排序的时间分析

外排序的时间消耗比内排序大得多，原因是：

- 外排序的数据量(记录)一般很大；
- 外排序涉及到内、外存之间的数据交换操作；
- 外存的操作速度远远比内存中的操作慢。

外排序的总时间由三部分组成：

外排序的时间=产生初始归并段的时间(内排序) $m \times t_{is}$

$+$ I/O操作的时间 $d \times t_{io}$

$+$ 内部归并的时间 $s \times ut_{mg}$

其中：

m：初始归并段数目；t_{is}：得到一个归并段的内排序时间；

d：总的读、写次数；t_{io}：一次读、写的时间；

s：归并的趟数；ut_{mg}：对u个记录进行一趟内部归并排序的时间。

一般地，$t_{io} >> t_{is}$，$t_{io} >> t_{mg}$，t_{io}而取决于所用外存,因此，影响外排序效率的主要原因是内、外存之间数据交换(读、写外存)。提高效率的主要方法(途径)有：

- 进行多路归并，减少文件归并的趟数；
- 增加归并段的长度，减少初始归并的数目；
- 根据不同归并段的长度，采取最佳归并方案。