## 序 数据结构A–课程概述

1 数据结构课程的历史
2 主要教学内容
3 课程地位
4 学习意义
5 教学方法
6 教材与教辅素材
7 教学要求与成绩评定方法



## 1 数据结构课程的历史

1968年， 美国欧·克努特教授
《计算机程序设计技巧》
1970年起，美国大学
开始广泛开设“数据结构”课程
1980年起，我国大学
开始开设“数据结构”课程
1985年起，我校“计算机应用”专业
开始开设“数据结构”课程



## 2 主要教学内容

**(1) 结构**

数据(data)在计算机存储器中的存储方法（含数据元素之间的相互关系的表示与存储方法）。

**(2) 算法**

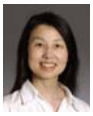
用程序处理数据的方法，即计算机利用存储器中的数据解决应用问题的程序(指令)操作步骤。

**特点**：结构与算法都属于方法论范畴，它们受到计算机硬件体系与工作原理的制约，形成**计算思维**。

**注**：本课程仅研究冯·诺依曼计算机体系上的数据结构(不含并行算法、数据流计算机算法等内容)



## 小引：什么是计算思维？



2006年3月，美国卡内基·梅隆大学计算机科学系主任周以真(Jeannette M. Wing)教授在美国计算机权威期刊《Communications of the ACM》杂志上给出并定义了**计算思维-Computational Thinking**。周教授认为：计算思维是运用计算机科学的基础概念进行问题求解、系统设计、以及人类行为理解等涵盖计算机科学之广度的一系列思维活动。

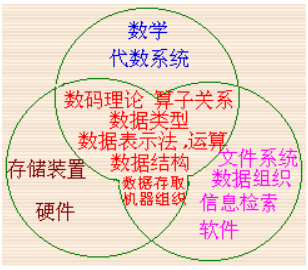
**计算思维的核心：指挥计算机工作的方法即程序设计(编程)的方法。**

**数据结构是讲授编程方法的最核心大学课程。**



## 3 课程地位

可以认为数据结构是介于数学、计算机硬件、计算机软件三者之间的一门核心课程。

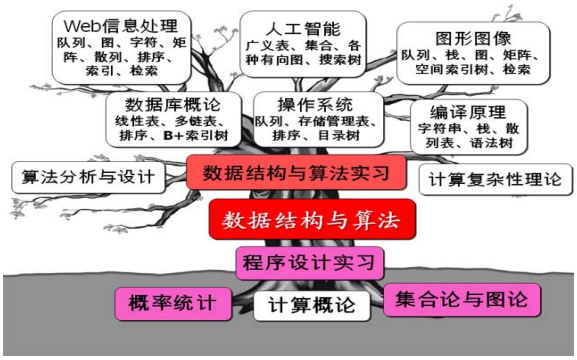



**《数据结构》所处的地位**



## 3 课程地位

续1完



## 4 学习意义

a. 提升学生的计算思维能力；
b. 培养学生比较扎实的算法设计能力；
c. 培养学生比较扎实的算法分析能力；
d. 培养学生较复杂程序的编码与调试能力。
e. 为专业后续课程学习打下坚实基础；
f. 为毕业后从事应用软件开发工作打下坚实基础。

**数据结构是计算机和软件类专业研究生入学考试全国统考专业课程要求的课程这一；**

**数据结构是大多数IT企业研发人员招聘笔试的科目之一；**

**数据结构是软件研发人员公认的大学学过的最重要和最有用的课程。**



## 5 教学方法

(1) 课堂授课
知识教学；
习题课；
程序设计讲评；
课堂提问与讨论；
**博士生助教实验上机指导课**；
(2) 上机编程实践(上机及程序作业，课程设计)
个人上机
(3) 答疑
QQ(助教)； EMail(助教及教师)；
考前见面答疑1-2次(教师)
(4) 考试(笔试)



## 6 教材与教辅素材

**(1) 推荐教材**

严蔚敏, 吴伟民.《数据结构(C语言版)》, 北京: 清华大学出版社.
配套：习题集与实验指导书

**(2) 教辅素材**

a. 学校教务网有数据结构A精品课程网站，可下载电子教案，模拟试卷，观看在线视频(3位教师主讲).
b. 学校教务网课程链接(教师动态上传资源)



## 7 教学要求与成绩评定

**(1) 上机要求**

建议准备个人电脑，安装C/C++语言系统
**推荐：Visual Studio** 2012/2013/2015 或
**Code::Blocks** (Windows平台的GCC)
(2016/01/28, codeblocks-16.01mingw-setup.exe)
操作系统：Windows 7 x64 旗舰版 或
Windows 8.1 x64 企业版
Windows 10 x64 企业版

**(2) 作业要求**

a. 笔试题型作业（分析 、写、画、算）
b. 编程作业(即实验课实验报告，含课程设计1次)
所有作业需打印，交打印纸



## 7 教学要求与成绩评定 续1完

**(3) 编程语言要求**

首选C语言；
上机编程可用C, C++, Java；
算法编程面向过程和面向对象实现不限制；
所选教材、考试题及上课的算法描述用C语言。
(以上C语言可能含少量非面向对象C++特性)

**(4) 成绩评定**

a. 平时成绩40%
作业(含实验课实验报告及课程设计)15%；
上课考勤15%；
半期考试(开卷)10%。
b. 期末笔试(闭卷) 60%

# 第1章 绪论





## 1.1 什么是数据结构

**1. 数据(Data)**

是客观事物的符号表示，在计算机科学中是指所有能够输入到计算机中并被计算机程序处理的符号总称。

计算机表示符号的方法就是二进制编码。

**2. 数据元素(Data Element)**

是数据的基本单位，也称为数据结构**结点**或**记录**。数据元素可视为由一个或多个数据项组成的集合。

**3. 数据项(Data Item)**

是数据不可分割的最小单位。数据项也称为**字段**、**属性**或**域**。



## 1.1 什么是数据结构 续1

**4. 数据对象(Data Object)**

是性质相同的数据元素组成的集合。在面向对象程序中，数据对象用类来描述，且一个数据元素称为一个实例(Instance)。

**5. 数据结构(Data Structure)**

是相互之间存在一种或多种特定关系的数据元素组成的集合。其中，数据之间的关系称为结构。

数据结构的形式定义：Data_Structure=(***D***, ***S***)

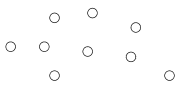
其中，***D***：数据元素的有限集合

***S***：*D*上关系的有限集合



## 1.1 什么是数据结构 续2

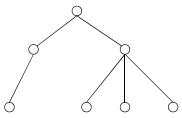
**(1) 结构的4种形式**



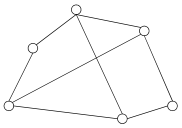
集合



线性结构



树形结构



图状结构或网状结构



## 1.1 什么是数据结构 续3完

**(2) 逻辑结构**

数据元素之间的逻辑关系称为逻辑结构。

**(3) 物理结构**

数据结构在计算机中的表示（又称映像）称为数据的物理结构，又称为**存储结构**。

数据元素在计算机中的关系有两种表示方法：

**a. 顺序映像 – 顺序存储结构**

例如，程序语言中的数组

**b. 非顺序映像 – 链式存储结构**

例如，程序语言的链表



## 1.2 什么是数据类型

**1. 数据类型(Data Type)**

是一个值的集合以及定义在该集合上的一组操作的总称。

例如，C语言的short数据类型，其值集=-32768~32767，

操作有：+, -, *, /, %, 关系运算, 位运算等

**2. 抽象数据类型(ADT, Abstract Data Type)**

是一个数学模型以及定义在该模型上的一组操作的总称。 ADT=(***D***, ***S***, ***P***)

***D***：数据对象集合

***S***：*D*上的关系集合

***P***：操作集合

## 1.2 什么是数据类型 续1完

**3. 数据处理中的基本操作**

**(1) 引用型**

只读取数据元素的操作，如：查询、比较等

**(2) 加工型**

可能改变数据元素或数据结构的操作，如：运算、插入、删除、更新、排序、编码等

**3. ADT的实现**

**(1) 数据结构的实现** 数组 、链表

**(2) 操作的实现** 算法、函数(过程)

**数据结构+算法=程序 (Pascal语言发明人N. Wirth)**

**程序+文档=软件**



## 1.3 什么是算法

**1. 算法(Algorithm)**

是计算机程序对特定问题求解步骤的一种描述。

**2. 算法的五个特性**

有穷性、确定性、可行性、输入、输出

**3. 算法设计的要求**

正确性、可读性、健壮性、高效率、低存储

**4. 算法的描述方法**

(1) 文字分步骤描述；

(2) 程序语言与类程序语言(伪代码)；

(3) 程序流程图。



## 1.4 算法分析

**1. 算法效率的度量**

算法的效率的一般用算法的执行时间表示。算法的执行时间度量分为绝对时间和相对时间。

**(1) 绝对执行时间**

指算法在某硬件上的实际执行时间，一般由事后统计得到。

**(2) 相对执行时间**

指算法的渐近时间复杂度，简称为**时间复杂度(TC, Time Complexity)**，TC一般定义为

$T(n)=O(f(n))$

其中，$n$表示问题的规模，一般用元素的数目表示；函数$f(n)$表示算法中的基本操作被重复执行的次数。



## 1.4 算法分析 续1

大$O$表示只取$f(n)$的最高数量级，且忽略与$n$无关的常数系数。

**例**：$T(n)=O(3n^3+2n^2+5n+10)=O(3n^3)=O(n^3)$

$T(n)=O(200)=O(1)$

$T(n)=O(2^n+n^2)=O(2^n)$

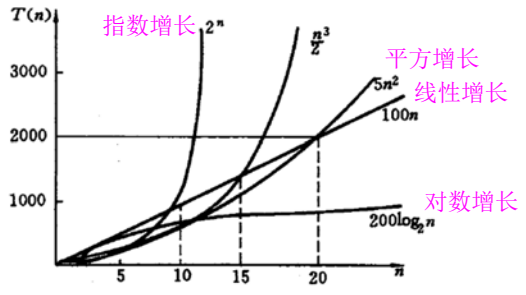
一般说来，$f(n)$是单调递增的，即问题规模$n$越大，$T(n)$越大，所需的计算时间越多。

时间复杂度$T(n)$一般由事先分析得到，这一工作称为**时间复杂度分析**。

算法的绝对执行时间受硬件性能影响，而时间算杂度无关硬件且只与问题的规模相关，它更为公平地揭示了算法执行的效率优劣。



## 1.4 算法分析 续2




常见函数的增长率



## 1.4 算法分析 续3

**例1** 分析以下算法的时间复杂度。

```
c=0;
for(i=0; i<n; i++)   //n为正整数
   for(j=0; j<=i; j++) c++;
```

**解**：分析c++;语句重复执行的次数$f(n)$，有

| i | c++;执行次数 |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| … | |
| n-1 | n |

$$f(n)=1+2+3+\cdots+n=\frac{n(n+1)}{2}$$

$$T(n)=O(f(n))=O(n^2)$$

## 1.4 算法分析 续4

**例2** 分析以下循环体执行的频度。

m=1; while(m<***n***) m*=2; //***n*为正整数**

**解**：设循环体执行次数为$k$，则

| 循环次数 | $m$初值 |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| … | |
| $k$ | $2^{k-1}$ |

$$2^{k-1} < n \Rightarrow k < \log_2 n + 1$$

$k$为满足上式的最大整数，则 $k = \lceil \log_2 n \rceil$。



## 1.4 算法分析 续5

当$n$固定时，$f(n)$可能随算法所处理数据元素不同而改变，从而产生**最优**、**最坏**以及**平均**$T(n)$的概念。

**例3** 分析以下顺序查找算法的**最优**、**最坏**$T(n)$；若各元素查找成功的概率相同，分析成功查找时的**平均**$T(n)$；若查找成功的概率为1/4(即失败查找的概率为3/4)，求算法的平均$T(n)$。

for(i=0; i<***n***; i++) if(a[i]==x) break;

**解**：a. 最优$T(n)$

当a[0]==x为真时，$T(n)=O(1)$

b. 最坏$T(n)$

查找失败时，$T(n)=O(n)$



## 1.4 算法分析 续6

c. 成功查找条件下的平均$T(n)$

$$f_{av|成功}(n) = \sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{1}{n}(1+2+\cdots+n) = \frac{n+1}{2}$$

$$T_{av|成功}(n) = O\left(\frac{n+1}{2}\right)$$

d. 算法的平均$T(n)$

$$f_{av}(n) = \frac{1}{4} f_{av|成功}(n) + \frac{3}{4} f_{av|失败}(n)$$

$$= \frac{1}{4}\cdot\frac{n+1}{2} + \frac{3}{4}\cdot n = \frac{7n+1}{8}, \quad T_{av}(n) = O\left(\frac{7n+1}{8}\right)$$



## 1.4 算法分析 续7

### 2. 算法的存储空间需求

算法的存储空间需求的度量用空间复杂度(Space Complexity)表示。空间复杂度定义为

$$S(n) = O(g(n))$$

其中，$n$表示问题的规模，$g(n)$表示存储空间大小。

类似地，可能也存在最优、最坏以及平均$S(n)$的概念，但一般只考虑最坏情况下的$S(n)$。

**例4** 分析以下求n!算法的空间复杂度。

(1) 非递归算法

fact=1.0; for(i=1; i<=***n***; i++) fact*=i;



## 1.4 算法分析 续8

(2) 递归算法

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

**解**：(1) 非递归算法 $S(n)=O(1)$

(2) 递归算法

| 递归层次 | n值 | 函数值 |
|---|---|---|
| 1 | $n$ | $n$*fact($n$-1) |
| 2 | $n$-1 | ($n$-1)*fact($n$-2) |
| … | … | … |
| $n$-1 | 2 | 2*fact(1) |
| $n$ | 1 | 1 |

$S(n)=O(n)$



## 1.4 算法分析 续9完

### 3. $T(n)$与$S(n)$关系

一般地，$T(n)$越小，$S(n)$越大；反之亦然。

为提高算法效率，常需要用空间换时间；

算法设计时，常需要折中考虑$T(n)$与$S(n)$。

## 第1章作业

（C语言版）
数据结构习题集
严蔚敏 吴伟明

1.8
1.9
1.12
1.20

# 第2章 线性表





## 2.1 线性表及其基本操作

### 1. 线性表的概念

**(1) 定义**

线性表是由$n(n\geq 0)$个数据元素(结点) $a_0, a_1, \dots, a_{n-1}$ 组成的有限长序列。

a. 空表 $n=0$时

b. 表长度 序列中结点的数目

d. 结点次序 结点按下标次序具有次序关系(**逻辑结构**)

形式定义：Linear_List=($D$, $R$)

其中，$D=\{a_0, a_1, \dots, a_{n-1}\}$，

$R=\{ <a_i, a_{i+1}> \mid i=0, 1, \dots, n-2 \}$



## 2.1 线性表及其基本操作 续1

**(2) 线性表的表示**

方法1：$(a_0, a_1, \dots, a_{n-1})$

方法2：用形式定义，Linear_List=($D$, $R$)

**(3) 结点的前趋和后继**

给定线性表$(a_0, a_1, \dots, a_{n-1})$，则结点$a_{i-1}$称为$a_i$的直接前趋；$a_{i+1}$称为$a_i$的直接后继；显然，结点$a_0$无前趋，而结点$a_{n-1}$无后继。



## 2.1 线性表及其基本操作 续2完

### 1. 线性表的基本操作

```
INITIATE(L)      //初始化线性表
CLEAR(L)         //置空表
LENGTH(L)        //求表长
GET(L, i)        //取元素a_i
PRIOR(L, e)      //求元素e的前趋
NEXT(L, e)       //求元素e的后继
LOCATE(L, x)     //查找等于x的元素
INSERT(L, i, x)  //i号下标之前插入结点x
DELETE(L, i)     //删除i号下标元素
EMPTY(L)         //判断是否为空表
```



## 2.2 线性表的顺序存储结构

### 1. 顺序表的物理结构

长度为$n$的线性表$(a_0, a_1, \dots, a_{n-1})$当各结点数据类型相同时可用容量大于或等于$n$的一维数组实现存储，元素$a_i$的存储地址由下式计算。

$address(a_i)=address(a_0)+i\cdot l$

其中，$i=0, 1, \dots, n$； $l$表示每个数据元素的存储长度。

### 2. 顺序表的C语言实现

**(1) 用固定容量的数组存储顺序表**

```
typedef 数据元素类型 ElemTp;
#define MAX_N 数组最大容量
```



## 2.2 线性表的顺序存储结构 续1

```
typedef struct { ElemTp elem[MAX_N];
                 int max_size; /* 顺序表最大容量 */
                 int n; /* 线性表实际长度 */
               } SqList;
//纯C语言初始化与置空表操作
void initiate(SqList *pL){ pL->n=0; pL->max_size=MAX_N; }
void clear(SqList *pL) { pL->n=0; }
//用C++
void initiate(SqList &L) { L.n=0; L.max_size=MAX_N; }
void clear(SqList &L) { L.n=0; }
```

## 2.2 线性表的顺序存储结构 续1

**(2) 用动态分配数组存储顺序表**

```
typedef 数据元素类型 ElemTp;
typedef struct { ElemTp *elem;
                 int max_size; /* 顺序表最大容量 */
                 int n; /* 线性表实际长度 */
               } SqList;
//纯C语言初始化、置空表与删除存储空间操作
typedef enum { ERROR, OK } Status;
//以上枚举型Status表示函数返回值
```



## 2.2 线性表的顺序存储结构 续2

```
Status initiate(SqList *pL, int max_size)
{ pL->n=0; pL->max_size=max_size;
  pL->elem=(ElemTp *)malloc(sizeof(ElemTp)*max_size));
  if(pL->elem==NULL) return ERROR;
  return OK;
}
void clear(SqList *pL) { pL->n=0; }
void destroy(SqList *pL)
{ free((void *)pL->elem); pL->n=pL->max_size=0;
}
```



## 2.2 线性表的顺序存储结构 续3

**3. 顺序表的插入、删除与定位算法**

**(1) 插入算法**

```
//插入元素e到线性表L的i号元素之前
//i=0, 1, …, n-1, n (当i=n时，表示在表尾添加元素e)
Status insert(SqList &L, int i, ElemTp e)
{ if(L.n>=L.max_size) return ERROR; //空间满不能插入
  if(i<0||i>L.n) return ERROR; //插入位置无效
  for(j=L.n-1; j>=i; j--) L.elem[j+1]=L.elem[j];
  L.elem[i]=e; L.n++; return OK;
}
```



## 2.2 线性表的顺序存储结构 续4

**时间复杂度分析**

算法时间主要消耗在向后搬动$i$号至$n$-1号元素(循环***n-i***次)。若插入位置$i$取值为0, 1, …, $n$的概率相同，则

$$f_{av}(n)=\sum_{i=0}^{n}\frac{1}{n+1}\cdot(n-i)$$

$$=\frac{1}{n+1}\cdot(0+1+2+\cdots+n)=\frac{n}{2}$$

故，$T_{av}(n)=O\left(\frac{n}{2}\right)$



## 2.2 线性表的顺序存储结构 续5

**(2) 删除算法**

```
//删除线性表L的i号元素(i=0, 1, …, n-1)
Status delete(SqList &L, int i)
{ if(L.n<=0) return ERROR; //空表不能删除
  if(i<0||i>=L.n) return ERROR; //删除元素下标无效
  for(j=i+1; j<L.n; j++) L.elem[j-1]=L.elem[j];
  L.n--;
  return OK;
}
```



## 2.2 线性表的顺序存储结构 续6

**时间复杂度分析**

算法时间主要消耗在向前搬动$i$+1号至$n$-1号元素(循环***n-i-1***次)。若删除位置$i$取值为0, 1, …, $n$-1的概率相同，则

$$f_{av}(n)=\sum_{i=0}^{n-1}\frac{1}{n}\cdot(n-i-1)$$

$$=\frac{1}{n}\cdot(0+1+2+\cdots+n-1)=\frac{n-1}{2}$$

故，$T_{av}(n)=O\left(\frac{n-1}{2}\right)$

## 2.2 线性表的顺序存储结构 续7

**(3) 定位算法**

```
//在线性表L中查找元素e
int Locate(SqList &L, ElemTp e)
{ for(i=0; i<L.n; i++) if(equal(L.elem[i], e)) break;
  if(i==L.n) i=-1;
  return i;      //返回-1表示查找失败，否则查找成功
}
```

**时间复杂度分析**(见第1章)



## 2.2 线性表的顺序存储结构 续8

**4. 两个有序顺序表的归并算法**

```
//两个升序顺序表la和lb归并为lc，使lc保持升序
Status merge (SqList &la, SqList &lb, SqList &lc)
{ if(la.n+lb.n>lc.max_size) return ERROR;
  i=j=k=0; lc.n=la.n+lb.n;
  while(i<la.n&&j<lb.n)
    if(LT(la.elem[i], lb.elem[j])) lc.elem[k++]=la.elem[i++];
      else lc.elem[k++]=lb.elem[j++];
  while(i<la.n) lc.elem[k++]=la.elem[i++];
  while(j<lb.n) lc.elem[k++]=lb.elem[j++];
  return OK; } //T(m+n)=O(m+n), m, n分别为la, lb长度
```



## 2.2 线性表的顺序存储结构 续9完

**5. 顺序表的操作特点小结**

**优点**

◊ 数组元素下标与结点序号一致，存取*i*号元素很方便；

**缺点**

◊ 插入、删除操作需移动元素，操作不便；

◊ 表的最大长度受数组最大容量限制；

◊ 当表长远远小于数组最大容量时，存储空间浪费大。



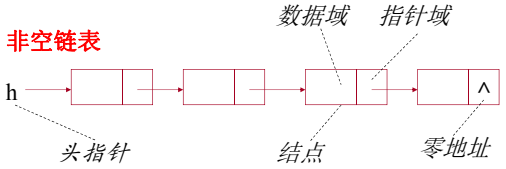
## 2.3 线性表的链式存储结构

**1. 链表的存储结构示意图**

**(1) 不带附加头结点单向链表**

空链表 h → NULL

非空链表



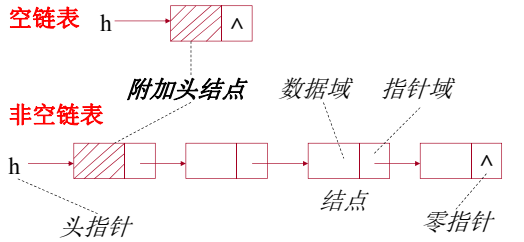

**缺点**：链表头插入、删除结点则h指针改变，增加了处理难度



## 2.3 线性表的链式存储结构 续1

**(2) 带附加头结点单向链表**

空链表 h → ^



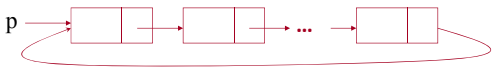

附加头结点的数据域常常弃之不用

**优点**：增加一个附加头结点，使很多算法得到简化

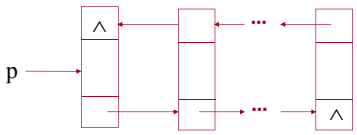


## 2.3 线性表的链式存储结构 续2

**(3) 单向循环链表**

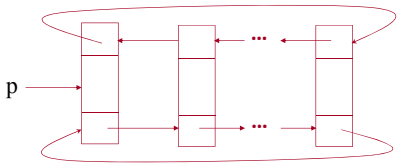



**(4) 双向链表**



## 2.3 线性表的链式存储结构 续3

**(5) 双向循环链表**






## 2.3 线性表的链式存储结构 续4

**2. 单向链表结点数据类型及基本操作的C语言实现**

**(1) 单向链表结点及结点指针数据类型**

```
typedef struct node
{ ElemTp data;          /* 结点数据域 */
  struct node *next;    /* 结点指针域, 后继指针 */
} LNode, *LinkList;
```

**(2) 建立单向链表结点以及删除结点**

```
#define CreateNode(p) p=(LinkList)malloc(sizeof(LNode))
#define DeleteNode(p) free((void *)(p))
```



## 2.3 线性表的链式存储结构 续5

**(3) 建立单向链表**

**例1** 若ElemTp为int型，输入若干整数，直到输入0时停止输入，按输入整数的次序建立带附加头结点单链表。

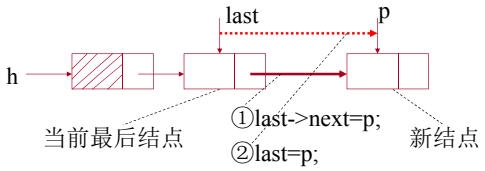
```
LinkList crt1() //方法一：先入先出法
{ CreateNode(h);  //建附加头结点
  last=h;  //指针last追踪当前最后一个结点
  while(1) { scanf("%d", &e); if(e==0) break;
             CreateNode(p); p->data=e;
             last->next=p; last=p; }
  last->next=NULL; return h;
}
```



## 2.3 线性表的链式存储结构 续6

调用格式：LinkList h;

h=ctr1();





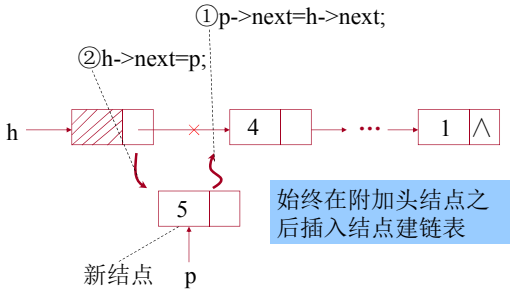

## 2.3 线性表的链式存储结构 续7

**例2** 若ElemTp为int型，输入n个整数，按输入整数的相反次序建立带附加头结点单链表。

```
LinkList crt2(int n) //方法二：先入后出法
{ CreateNode(h);  h->next=NULL; //建空链表
  for(i=0; i<n; i++)
    { CreateNode(p); scanf("%d", &p->data);
      //以下将每个新结点都插入到头结点后面
      p->next=h->next; h->next=p;
    } //连接次序与输入次序正好相反
  return h;
}
```



## 2.3 线性表的链式存储结构 续8



## 2.3 线性表的链式存储结构 续9

**(3) 输出单向链表**

**例3** 若ElemTp为int型，输出链表各数据结点。

```
void prt(LinkList h)
{ p=h->next; //p指向第1个数据结点
  while(p) { printf("%8d", p->data); p=p->next; }
}
```

**(4) 置空链表(删除所有数据结点)**

```
void clear(LinkList h)
{ p=h->next; h->next=NULL;
  while(p) { q=p; p=p->next; DeleteNode(q); }
}
```

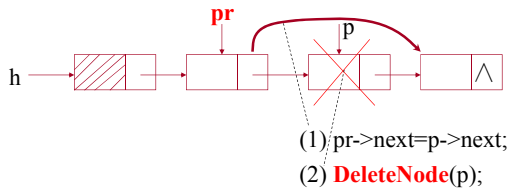


## 2.3 线性表的链式存储结构 续10

**(5) 删除结点**

算法要点(如图所示)

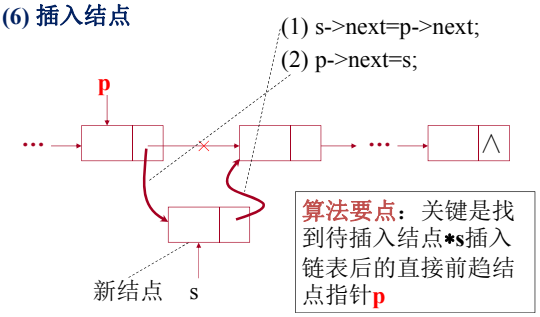
关键是找到被删除结点*p的前趋结点指针pr






## 2.3 线性表的链式存储结构 续11

**(6) 插入结点**




算法要点：关键是找到待插入结点*s插入链表后的直接前趋结点指针p



## 2.3 线性表的链式存储结构 续12

**3. 单向链表算法综合举例**

**例4** 设计算法，删除带附加头结点单链表中data域值满足某删除条件下的结点。

```
int erase(LinkList h, bool (*beErased)(ElemTp))
// 函数返回被删除的结点数;
// beErased函数指针表示删除条件判断函数.
{ pr=h; p=h->next; c=0;
  while(p) if(beErased(p->data))
                { pr->next=p->next; DeleteNode(p); c++;
                  p=pr->next;
                } else { pr=p; p=p->next; }
  return c; } //T(n)=O(n), n表示链表数据结点数目
```

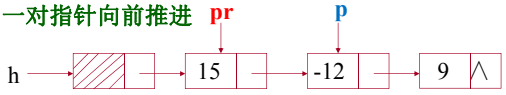


## 2.3 线性表的链式存储结构 续13

p: 主扫描指针(master)

pr: 从动指针(slave)，始终指向结点*p的直接前趋

一对指针向前推进




若ElemTp为int型，删除条件为小于0，则beErased为

```
bool beErased(int e)
{ return e<0;
}
```



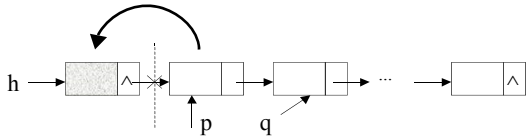
## 2.3 线性表的链式存储结构 续14

**例5** 设计算法，实现带附加头结点单链表各数据结点逆序连接。

**算法设计：**

首先断开附加头结点，使之成为空链表；

将各数据结点依次摘下，用先入后出法重构链表。



## 2.3 线性表的链式存储结构 续15

```
void reverse(LinkList h)
{ q=h->next; h->next=NULL;
  while(q)
   { p=q; q=q->next; //“摘下”结点，用p指到
     p->next=h->next; h->next=p; //结点*p前插法插入
   }
}
//T(n)=O(n), n表示链表数据结点数目
```

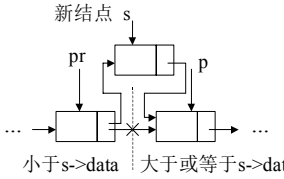


## 2.3 线性表的链式存储结构 续16

**例6** 带附加头结点单链表各数据结点升序连接。

**算法设计：** 首先断开附加头结点，使之成为空链表；然后依次摘下数据结点，插入到升序链表中(空链表也可视为升序链表)，使插入结点后的链表保持升序。

**算法要点：** 怎样实现升序链表插入结点保持升序？




将一对指针(pr, p)从头指针(h, h->next)推进到如左图所示位置。即，当第一次出现p->data≥s->data时，停止指针推进。



## 2.3 线性表的链式存储结构 续17

```
void upsort(LinkList h)
{ q=h->next; h->next=NULL; //断开头结点
  while(q) //依次摘结点并插入升序链表
  { s=q; q=q->next; //“摘下”结点*s
    //以下将*s插入到升序链表中
    pr=h; p=h->next;  //初始化主、从动指针
    //以下推进主、从动指针到上一页图示的插入位置
    while(p&&LT(p->data, s->data)) { pr=p; p=p->next; }
    //以下插入结点*s
    pr->next=s; s->next=p;
  }
} //T_av(n)=O(n^2), n表示链表数据结点数目
```



## 2.3 线性表的链式存储结构 续18

**4. 两个有序单链表的归并算法**

```
//两个升序单链表ha和hb归并为hc，使hc保持升序
//初始时, hc为空链表；ha, hb, hc均带附加头结点
//利用ha, hb原结点归并，归并后ha, hb成为空链表
void merge (LinkList ha, LinkList hb, LinkList hc)
{ pa=ha->next; pb=hb->next; pc=hc;
  whie(pa&&pb)
    if(LT(pa->data, pb->data))
      { p=pa; pa=pa->next;
        pc->next=p; pc=p;
      }
```



## 2.3 线性表的链式存储结构 续19

```
    else
      { p=pb; pb=pb->next;
        pc->next=p; pc=p;
      }
  if(pa) pc->next=pa;
  if(pb) pc->next=pb;
  ha->next=hb->next=NULL;
}
/* T_最坏(m+n)=O(m+n),
   m, n分别表示链表ha, hb的数据结点数
   T_最好(m+n)=O(min(m, n)),
*/
```



## 2.3 线性表的链式存储结构 续20

**5. 单向链表的操作特点小结**

**优点**

◊ 插入、删除元素无需移动其它元素；

◊ 无多余元素存储空间。

**缺点**

◊ 结点指针域占用额外存储空间；

◊ 访问第i号结点必须从头结点出发，效率低；

## 2.3 线性表的链式存储结构 续21

**6. 双向链表**

**(1) 双向链表结点及结点指针数据类型的C语言实现**

```
typedef struct bnode
{ ElemTp data;          /* 结点数据域 */
  struct bnode *next;   /* 结点指针域, 后继指针 */
  struct bnode *prior;  /* 结点指针域, 前趋指针 */
} BLNode, *BLPtr;
```

**(2) 删除结点**

```
    //已知某数据结点指针p(p≠NULL)，删除*p
```



## 2.3 线性表的链式存储结构 续22

```
//非循环双向链表(带附加头结点)
p->prior->next=p->next;
if(p->next) p->next->prior=p->prior;
DeleteNode(p);
//循环双向链表(不带附加头结点)
if(p!=p->next) //链表结点数大于1
{ p->prior->next=p->next;
  p->next->prior=p->prior;
}
DeleteNode(p);
```



## 2.3 线性表的链式存储结构 续23

**(3) 插入结点**

```
//已知某结点指针p(p≠NULL), 在*p后面插入结点*s
//非循环双向链表(带附加头结点)
s->next=p->next; s->prior=p;
p->next=s; if(s->next) s->next->prior=s;
//循环双向链表(不带附加头结点)
s->next=p->next; s->prior=p;
p->next=s; s->next->prior=s;
```



## 2.3 线性表的链式存储结构 续24

**例7** 已知双向循环链表(不带附加头结点)某结点地址p。编写算法，删除链表中结点data域值满足条件beErased的所有结点。若执行删除操作后链表中无剩余结点，函数返回NULL；否则，函数返回某剩余结点地址。

```
BLPtr erase(BLPtr p, bool (*beErased)(ElemTp))
{ q=p->next;  //删除*p以外其它满足条件的结点*q
  while(q!=p)
    if(!beErased(q->data)) q=q->next; //沿next方向搜索
      else { s=q; q=q->next; //“摘下”结点*q, s指到
             s->prior->next=q; q->prior=s->prior;
             DeleteNode(s);  //删除*s }
```



## 2.3 线性表的链式存储结构 续25完

```
  //判断*p是否需要删除
  if(beErased(p->data))
  { if(p==p->next)  //只剩1个结点且需删除
      { DeleteNode(p); return NULL; }
    s=p; p=p->next;
    s->prior->next=p; p->prior=s->prior;
    DeleteNode(s);
  }
  return p;
}
```



## 2.4 线性表的其它存储结构

**1. 散列表**

见第9章-查找-哈希表

**2. 压缩存储**

如广泛应用于数据压缩的游程编码

**3. 索引存储**

见第9章-查找-索引顺序表

**4. 静态链表**

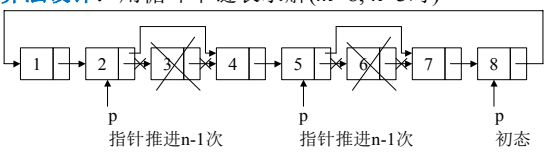
用数组实现链表：结点地址用数组元素下标表示。

某些不具备指针或引用类型的高级语言(如BASIC语言)只能用静态链表实现链表。

## 2.4 线性表的其它存储结构 续1

**例8** 约瑟夫问题：*m*个人依次编号为1~*m*，他们围成一圈，从第1个人开始报数，数到*n*的人出圈，然后继续从1开始报数，数到*n*的人继续出圈，这样直到所以人都出圈，求出圈人的编号次序。

**算法设计**：用循环单链表求解(*m*=8, *n*=3时)



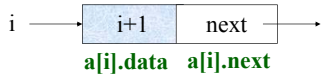

**p始终指向出圈人的直接前趋结点**



## 2.4 线性表的其它存储结构 续2

用静态链表求解，结点结构设计如下。

第i个人(结点a[i])
- 存储地址(下标)为**i** (i=0, 1, 2, …, m-1)
- 数据域值(人的编号)为**i+1**
- 指针域值为next




a[i].data a[i].next

**a[i].data可不存储**
**a[i].next用p[i]存储**

p[i]=k表示i+1号人的直接后继为k+1号人，初始时，p[i]=(i+1)%m。



## 2.4 线性表的其它存储结构 续3完

```
#include <iostream>  //C++源程序
using namespace std;
void main()
{ int m, n, i, j, *p; cout<<"Input m n:"; cin>>m>>n;
  p=new int[m]; for(i=0; i<m; i++) p[i]=(i+1)%m;
  i=m-1; //指针i指向m号人
  while(p[i]!=i)  //循环链表结点数>1时循环
  { for(j=1; j<n; j++) i=p[i];  //指针推进n-1次
    cout<<p[i]+1<<" "; p[i]=p[p[i]];  //删除p[i]所指结点
  }                                   //指针飞跃实现删除
  cout<<i+1<<" "<<endl;  //输入8 3
  delete []p; }              //输出3 6 1 5 2 8 4 7
```



## 2.5 线性表的应用-一元*n*次多项式

**1. 一元*n*次多项式的线性表表示方法**

一元*n*次多项式的数学表达式为

$$P_n(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}+a_nx^n$$

**方法1**: 用线性表顺序存储0次至*n*次幂系数，即

$(a_0, a_1, \ldots, a_{n-1}, a_n)$ //适合顺序表实现

**方法2**: 只存储非零系数及其指数

$((c_0,e_0),(c_1,e_1),\cdots,(c_{m-1},e_{m-1}))$ //适合链表实现

其中，系数$c_0, c_1, \ldots, c_{m-1}$均不等于0；

指数$0\le e_0<e_1<\cdots<e_{m-1}=n$。

即 $P_n(x)=c_0x^{e_0}+c_1x^{e_1}+\cdots+c_{m-1}x^{e_{m-1}}$。

**当*n*较大而非零系数较少时适合方法2。**

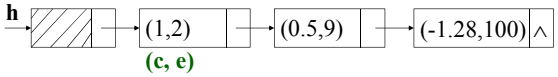


## 2.5 线性表的应用-一元*n*次多项式 续1

**2. 方法2用链表实现的C语言描述**

```
typedef struct polynode
{  double c; //系数
   int e;      //指数
  struct polynode *next;   //要求结点按指数e升序连接
} PNode, *Polyn;
```

例如，$P_{100}(x)=x^2+0.5x^9-1.28x^{100}$




(c, e)



## 2.5 线性表的应用-一元*n*次多项式 续2

**3. 方法2(链表)实现两个多项式相加**

```
//两个多项式相加算法类似于两个升序表的归并算法；
//以下算法考虑利用两个源多项式结点生成和式的结点；
//算法要求两个源多项式结点按指数e升序连接；
//算法结束后，两个源多项式成为空链表。
Polyn  add(Polyn h1, Polyn h2) //函数返回和式头指针
{ p1=h1->next; p2=h2->next;
  CreateNode(h); p3=h;   //h为和式附加头结点指针
  while(p1&&p2)
   { if(p1->e<p2->e) { p=p1; p1=p1->next; }
       else if(p2->e<p1->e) { p=p2; p2=p2->next; }
         else  //p1->e==p2->e为真时
```

## 2.5 线性表的应用-一元n次多项式 续2

```
            {  p1->c+=p2->c;  //p1->e==p2->e为真时
               if(p1->c==0)  //p1->c==0应删除两个结点
               { p=p1;p1=p1->next; DeleteNode(p);
                 p=p2;p2=p2->next; DeleteNode(p);
                 continue;
               }
               p=p2; p2=p2->next; DeleteNode(p);
               p=p1; p1=p1->next;
            }
      p3->next=p; p3=p;  //插入*p结点至和式末尾
   } //end of while
```



## 2.5 线性表的应用-一元n次多项式 续3

```
   if(p1) p3->next=p1;
    else if(p2) p3->next=p2;
              else p3->next=NULL;
   h1->next=h2->next=NULL;
   return h;
}
```



## 2.5 线性表的应用-一元n次多项式 续4

**4. 方法2(链表)实现两个多项式相乘**

设 $P(x)=a_0x^{b_0}+a_1x^{b_1}+\cdots+a_{m-1}x^{b_{m-1}}$

$Q(x)=c_0x^{d_0}+c_1x^{d_1}+\cdots+c_{n-1}x^{d_{n-1}}$

则 $R(x)=P(x)\cdot Q(x)=\sum_{i=0}^{n-1}P(x)c_ix^{d_i}$

```
//原理性算法如下
R(x)=0;
for(i=0; i<n; i++)
{ T(x)=P(x)c_i x^{d_i};      //T(x)结点数与P(x)相同
  R(x)=R(x)+T(x);   //利用加法算法
}
```



## 2.5 线性表的应用-一元n次多项式 续5

```
//细化算法如下
Polyn mul(Polyn hp, Polyn hq)
{ CreateNode(hr);hr->next=NULL;  //R(x)=0;
  CreateNode(ht); ht->next=NULL;  //T(x)=0;
  q=hq->next;
  while(q)   //实现for(i=0; i<n; i++)
    { //以下实现T(x)=P(x)c_i x^{d_i};
      pt=ht;p=hp->next;
      while(p) { CreateNode(pt->next); pt=pt->next;
        pt->c=p->c*q->c;pt->e=p->e+q->e; p=p->next;
      } //end of while(p)
      pt->next=NULL; q=q->next;
```



## 2.5 线性表的应用-一元n次多项式 续6完

```
      //以下实现R(x)=R(x)+T(x)
      p=add(hr, ht); DeleteNode(hr); hr=p;
    } //end for while(q)
  DeleteNode(ht); return hr;
}
//该算法时间复杂度T(m, n)=O(m·n)
```



## 第2章 作业

**习题集**

2.19, 2.20, 2.21(电子教案例5),
2. 24
2.31, 2.32

**上机作业**

1. 两个有序顺序表的归并。
2. 电子教案例7。
3*. 一元n次多项式乘法实现。

# 第3章 栈和队列





## 3.1 栈

**1. 栈的概念**

栈(**Stack**)也称为堆栈，它是一种特殊的线性表，这种线性表具有以下特点：

a. 所有插入与删除操作均在称为栈顶(TOP)的一端进行，不允许插入、删除操作的另一端称为栈底(BOTTOM)；

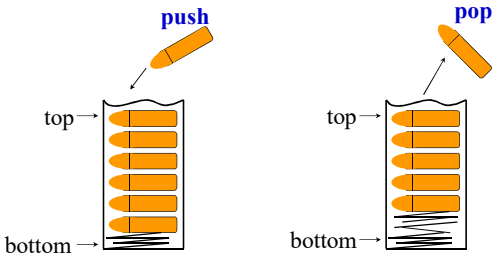
b. 最后一个进栈的结点最先出栈(**LIFO**, Last In First Out)。

压栈(**Push**)：指插入数据元素的操作，也称为入栈

退栈(**Pop**)：指删除数据元素的操作，也称为出栈



## 3.1 栈 续1




子弹匣的工作原理与堆栈相同



## 3.1 栈 续2

**2. 栈的表示与基本操作的实现**

**(1) 顺序存储栈**

typedef 入栈元素类型 **ElemTp**

```
typedef struct { ElemTp *elem;
                 int n;  //堆栈容量
                 int top; //栈顶元素下标(栈顶指针)
               }SqStack;
```

定义栈顶指针top为：**top=-1** (表示栈空)，

入栈方向为top+1方向(称为向上生长栈)。



## 3.1 栈 续4

```
int createStack(SqStack &s, int n) //建容量为n的堆栈
{ if(n<=0) return 0;
  s.n=n; s.top=-1;  //置空栈
  s.elem=(ElemTp *)malloc(sizeof(ElemTp)*n);
  if(!s.elem) return 0;
  return 1; //返回1表示成功；返回0表示失败
}
void destroyStack(SqStack &s) //销毁堆栈存储空间
{  s.n=0; s.top=-1; free((void *)s.elem);
}
void clearStack(SqStack &s) { s.top=-1; } //置空栈
```

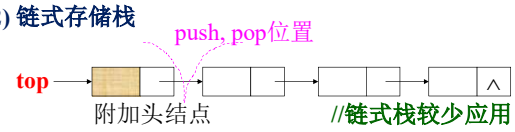


## 3.1 栈 续5

```
int empty(SqStack &s) { return s.top==-1; } //判断栈空
int full(SqStack &s) { return s.top>=s.n-1; } //判断栈满
int push(SqStack &s, ElemTp e)  //入栈操作
{ if(s.top>=s.n-1) return 0;  //栈满，入栈失败(上溢)
  s.elem[++s.top]=e;  //元素e入栈
  return 1;  //入栈成功返回1
}
int pop(SqStack &s, ElemTp &e)  //退栈操作
{ if(s.top==-1) return 0; //栈空，退栈失败(下溢)
  e=s.elem[s.top--];
  return 1; //退栈成功返回1
}
```

## 3.1 栈 续6

**(2) 链式存储栈**




**empty**: top->next==NULL

**full**: 不用实现(除非内存耗尽)

**push**: CreateNode(p); p->data=e;
p->next=top->next; top->next=p;

**pop**: p=top->next; top->next=p->next; e=p->data;
DeleteNode(p);



## 3.1 栈 续7

**3. 堆栈习题举例**

**例1** 若元素入栈次序为ABC，写出所有可能的元素出栈次序。

答: 所有可能的元素出栈次序共5种，即

| | |
|---|---|
| ABC | 操作PXPXPX (P表示入栈，X表示退栈) |
| ACB | PXPPXX |
| BAC | PPXXPX |
| BCA | PPXPXX |
| CBA | PPPXXX |



## 3.1 栈 续8完

**例2** 若元素入栈次序为1、2、3、4，写出所有4在2之前出栈的元素出栈次序。

答: 共有4种，即

4321
3421
1342
1432



## 3.2 队列

**1. 队列的概念**

队列(**Queue**)是具有以下特性的线性表：

a. 只能在一端(称为始端front)删除结点；

b. 只能在另一端(称为终端rear)插入结点。

队列的操作特点满足先进先出原则(**FIFO**, First In First Out)。

入队：指插入元素的操作；

出队：指删除元素的操作。

front端也称为**队头**，rear端也称为**队尾**。

出队 ← $a_0, a_1, a_2, \ldots, a_{n-1}$ ← 入队



## 3.2 队列 续1

**2. 队列的表示与基本操作的实现**

**(1) 顺序存储队列**

typedef 入栈元素类型 **ElemTp**

```
typedef struct { ElemTp *elem;
                 int n;  //队列容量
                 int f; //队头指针
                 int r; //队尾指针
               }SqQueue;
```

**规定**：入队方向为r+1方向



## 3.2 队列 续2

**a. 非循环队列的基本操作**

**init**: f=r=-1 (f指向队头之前位置；r指向队尾元素)

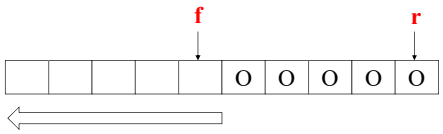
**empty**: f==r **full**: r-f==n

**enQueue**(入队):

```
if(r-f<n)  //队列不满可入队
 {  if(r==n-1) //假溢
      { elem[0..n-f-2]←elem[f+1..n-1]; //向左移动元素
        f=-1; r=n-f-2;                 //使elem[0]为队头
      }
    elem[++r]=e;
 }
```

## 3.2 队列 续3

dlQueue(出队):
if(f<r) //队列不空可出队
{ e=elem[++f];
}




发生"假溢"时，将队列中元素向左搬动至数组最开头



## 3.2 队列 续4

**b. 循环队列的基本操作**

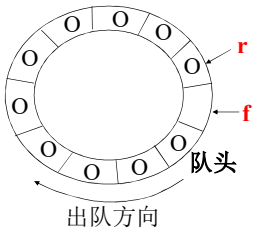
为避免“假溢”以及搬动元素的操作，在入队时若另++r;操作变为r=(r+1)%n;则数组在逻辑上成为圆环形，称为循环队列。

在循环队列中，当f==r为真时，难以区分队空和队满，为此需要采用以下两种方案实现循环队列的基本操作。

//循环队列实现方案一
**f指向队头元素之前空闲位置；r指向队尾元素**
(即队头元素和队尾元素至少间隔一个空闲元素位置)



## 3.2 队列 续5




循环队列方案一：
队满的情形

```
void clearQueue(SqQueue &q)
{ q.r=q.f=-1;
  /*r=f=-1~n-1区间的任一
    整数均可*/
}
int empty(SqQueue &q)
{ return q.f==q.r;
}
int full(SqQueue &q)
{ return
  (q.r+1)%q.n==(q.f+n)%n;
}
```



## 3.2 队列 续6

```
int enQueue(SqQueue &q, ElemTp e)
{ if((q.r+1)%q.n==(q.f+n)%n) return 0; //队满，入队失败
  q.r=(q.r+1)%q.n;
  q.elem[q.r]=e;
   return 1;  //入队成功返回1
}
int dlQueue(SqQueue &q, ElemTp &e)
{ if(q.r==q.f) return 0; //队空，出队失败
  q.f=(q.f+1)%q.n;
  e=q.elem[q.f];
  return 1;  //出队成功返回1
}
```



## 3.2 队列 续7

//循环队列实现方案二
**在SqQueue结构体中增设计数变量c，记录队列中当前元素个数**

```
void clearQueue(SqQueue &q)
{ q.r=q.f=-1; q.c=0;   //r=f=-1~n-1区间任意整数均可
}
int empty(SqQueue &q) { return q.c==0; }
int full(SqQueue &q) { return q.c==q.n; }
```

//队空、队满时q.f==q.r均为真
//优点：队满时没有空闲元素位置(充分利用了空间)

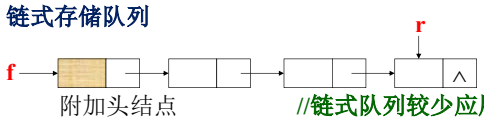


## 3.2 队列 续8

```
int enQueue(SqQueue &q, ElemTp e)
{ if(q.c==q.n) return 0; //队满，入队失败
  q.r=(q.r+1)%q.n;
  q.elem[q.r]=e; q.c++;
   return 1;  //入队成功返回1
}
int dlQueue(SqQueue &q, ElemTp &e)
{ if(q.c==0) return 0; //队空，出队失败
  q.f=(q.f+1)%q.n;
  e=q.elem[q.f]; q.c--;
  return 1;  //出队成功返回1
}
```

## 3.2 队列 续9完

**(2) 链式存储队列**




**empty**: f->next==NULL

**full**: 一般不用实现(除非内存耗尽)

**enQueue**: CreateNode(p); p->data=e; p->next=NULL;
r->next=p; r=p;

**dlQueue**: p=f->next; e=p->data;
f->next=p->next; DeleteNode(p);



## 3.3 栈的应用

**1. 系统栈**

机器语言程序用系统堆栈实现子程序参数传递、保存子程序返回地址、保存子程序断点状态(CPU内部寄存器状态)等。

机器语言指令集中包含push和pop指令实现对系统堆栈的操作。系统堆栈位于内存特定区域，由计算机硬件系统初始化程序来设置，用户程序一般可不设置。

相关内容参见“汇编语言程序设计”。



## 3.3 栈的应用 续1

**2. 栈与递归**

**(1) 递归程序的存储空间消耗**

由于函数调用的指令返回地址、形式参数以及断点状态均用系统堆栈实现存储，因此递归调用的层次数(**深度**)决定了系统堆栈必须保留的存储空间容量大小。

**例1** 以下函数用递归法实现$n$元一维数组元素逆序存储，试分析所需栈的深度。

```
void reverse(ElemTp a[], int i, int j)
//数组a下标范围i..j实现元素逆序存储
{ if(i<j) { a[i]↔a[j]; reverse(a, i+1, j-1); } }
```



## 3.3 栈的应用 续2

**分析**：对于长度为$n$的数组，在下标范围0..$n$-1实现元素逆序存储，则第一次调用该函数的形式为

**reverse**(a, 0, $n$-1); //数组a下标0..$n$-1范围逆序存储

显然，**当$i \geq j$时，递归达到最大深度**，此时堆栈深度最大。若$n$=6， 则 $i$=0, 1, 2, **3**
$j$=5, 4, 3, **2** ($i>j$为最深一层调用)

若$n$=7， 则 $i$=0, 1, 2, **3**
$j$=6, 5, 4, **3** ($i=j$为最深一层调用)

故 栈最大深度=递归最大深度=$\left\lfloor \frac{n}{2} \right\rfloor + 1$,

形参变量a, i, j的空间复杂度为$O$(上式)。



## 3.3 栈的应用 续3

**(2) 递归算法用堆栈实现非递归形式**

**例2** 以下给出计算组合数的递推公式，写出非递归和递归算法，然后用堆栈消去递归。

$$C_n^m = \begin{cases} 1 & \text{当} m=0 \\ \dfrac{n-m+1}{m} C_n^{m-1} & \text{当} m=1, 2, \ldots, n \end{cases}$$

```
int comb(int n, int m) //非递归算法
{ int i, c=1;
  for(i=1; i<=m; i++) c=c*(n-i+1)/i;
  return n;
}
```



## 3.3 栈的应用 续4

```
int comb(int n, int m)  //递归算法
{ if(m==0) return 1;
  return comb(n,m-1)*(n-m+1)/m;
}
```

**递归算法分析**：comb(6, 3)

| 递归层次 | n | m | 返回值 comb(6,3)=20 |
|---|---|---|---|
| 1 | 6 | 3 | ?*4/3=15*4/3=20 |
| 2 | 6 | 2 | ?*5/2=6*5/2=15 |
| 3 | 6 | 1 | ?*6/1=1*6/1=6 |
| 4 | 6 | 0 | 1 |

m连续入栈

m连续出栈直至栈空

## 3.3 栈的应用 续5

```
int comb(int n, int m)  //用堆栈消去递归
{ int c;
  initStack(S);
  do push(S, m--); while(m>=0);
  while(!empty(S))
  { pop(S, m);
    if(m==0) c=1;
      else c=c*(n-m+1)/m;
  }
  return c;
}
```



## 3.3 栈的应用 续6

**例3** 已知两个自然数$a$, $b$的最大公约数由下式计算，试写出非递归算法和递归算法，然后利用堆栈消去递归。

$$\gcd(a,b)=\begin{cases} a & \text{当} b=0 \\ \gcd(b, a\%b) & \text{当} b\neq 0 \end{cases}$$ //%表示求余数

```
//非递归算法
int gcd(int a, int b)
{ int t;
  while(b){ t=a%b; a=b; b=t; }
  return a;
}
```

```
//递归算法
int gcd(int a, int b)
{ if(b) return gcd(b, a%b);
  return a;
}
```



## 3.3 栈的应用 续7

递归算法分析：gcd(24, 32)

| 递归层次 | a | b | 返回值 |
|---|---|---|---|
| 1 | 24 | 32 | ? |
| 2 | 32 | 24 | ? |
| 3 | 24 | 8 | ? |
| 4 | 8 | 0 | 8 |

堆栈

```
int gcd(int a, int b)
{ initStack(S);push(S, a, b);
  while(1)
    { get_top(S, a, b);
      if(b!=0)
          push(S, b, a%b);
        else break;
    }
  get_top(S, a, b);
  clearStack(S);
  return a;
}
```



## 3.3 栈的应用 续8

**用堆栈消除递归的一些经验**：

a. 形参中的变化量一般需要按递归深入方向入栈；
b. 计算或操作一般通过连续退栈至栈空实现；
c. 有时，各层递归的返回值需单独设置变量入栈；
d. 有时，退栈操作可以简化(如**例3**的gcd函数)；
e. 当各层递归调用返回后需要再次递归深入的，可能需要在入栈元素中增设标志变量，从而在每次出栈操作后根据标志变量判断是继续出栈或者入栈；
f. 复杂递归的消除有时需要单独分析设计以及其它编程技巧。



## 3.3 栈的应用 续9

**例4** 用堆栈消除以下Fibonacci函数的递归。

```
int fib(int n)
{ if(n<=1) return 1;
  return fib(n-1)+fib(n-2);
}
```

**解**：返回值表达式中，+号两边的函数值需入栈

```
//C++源程序代码：输出Fibanacci数列前20项
#include <iostream>
#include <iomanip>
using namespace std;
```



## 3.3 栈的应用 续10

```
typedef struct { int n, r1, r2;} ElemTp; //入栈元素类型
//r1, r2分别存储fib(n-1)和fib(n-2)的返回值
//r1, r2同时也兼作标志变量
//r1, r2初值为-1表示返回值尚未求得, 需进行入栈操作
//以下为用堆栈实现的非递归算法
int fib(int n)
{ ElemTp *s=new ElemTp[n]; //建栈，递归深度不会超n
  int top=-1;  //置空栈
  ElemTp a={n, -1, -1}; //-1表示函数返回值未求得
  s[++top]=a;  //(n, -1, -1)压栈
  int r;  //临时变量r用于计算fib(n-1)+fib(n-2)等
```

## 3.3 栈的应用 续11

```
while(1)
  if(s[top].r1!=-1&&s[top].r2!=-1) //栈顶r1, r2均已求得
    { r=s[top].r1+s[top].r2;  //计算fib(n-1)+fib(n-2)
      --top; if(top==-1) break;  //退栈, 栈空退出循环
      if(s[top].r1==-1) s[top].r1=r; else s[top].r2=r;
    }
  else                    //栈顶为(x, 1, 0)其中 x≤1)
    if(s[top].n<=1) {s[top].r1=1; s[top].r2=0; }
     else { a.n=s[top].n-1;a.r1=a.r2=-1; //(n-1, -1, -1)
            if(s[top].r1!=-1) a.n--;   //(n-2, -1, -1)将入栈
            s[++top]=a; //入栈
          }
```



## 3.3 栈的应用 续12

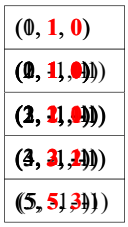
```
  delete []s; //退出while循环后删除栈
  return r;   //返回fib(n)
}
int main()
{ int i;
  for(i=0; i<20; i++)   //输出Fibanacci数列前20项值
    {  cout<<setw(12)<<fib(i); //输出每项域宽12列
       if(i%5==4) cout<<endl;  //每输出5项换行
    }
   return 0;
}
```



## 3.3 栈的应用 续13

fib(5)运行过程中的堆栈状态变化动画演示

fib(5)=8






## 3.3 栈的应用 续14

**3. 某些问题适合用堆栈求解**

**例5** 编程输入一个自然数 *n*，输出该自然数所有不增的和式。

如：*n*=4↵

4=4
4=3+1
4=2+2
4=2+1+1
4=1+1+1+1

*n*=5↵

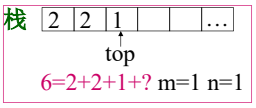
5=5
5=4+1
5=3+2
5=3+1+1
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1



## 3.3 栈的应用 续15

**解：算法设计**

(1) 置栈S为空; 输入n值;
(2) m=n;  /* m为当前求和项，n作为剩余和 */
(3) while(m>0)
  { push(S, m); n=n-m;
    if(n等于0)
    { 以入栈顺序打印各求和项;
      do { pop(S, m); n=n+m; m--; }
      while(m为0且栈不空);
    }
    m=min(m, n);
  }






## 3.3 栈的应用 续16

```
#include "stdio.h"   /* C语言源程序 */
#define MAX_N 10
void main()
{  int n,m,n0, s[MAX_N],top=-1, i;
   printf("Input n="); scanf("%d",&n);
   m=n; n0=n;
   while(m>0)
   { s[++top]=m; n-=m;
     if(n==0)
      { printf("%d=%d",n0, s[0]);
        for(i=1; i<=top; i++) printf("+%d",s[i]);
        printf("\n");
```

## 3.3 栈的应用 续17

```
        do { m=s[top--]; n+=m;m--; }
        while(m==0&&top!=-1);
      } /* end of if(n==0) */
      m=m<=n?m:n;
    } /* end of while */
}
```

其它可能用到的堆栈的算法包括回溯算法、走密宫问题等，有兴趣的同学可自学。



## 3.3 栈的应用 续18

**4. 表达式求值**

**(1) 表达式的前缀形式与后缀形式**

例如：a+b×5/(3-c) //中缀式(传统数学表达式)

+ a / × b 5 - 3 c //前缀式(波兰式)

a b 5 × 3 c - / + //后缀式(逆波兰式)

1929年，波兰逻辑学家J. Lukasiewicz最先发现表达式的前缀式形式，后人又发现后缀式是前缀式的对称形式。

波兰式和逆波兰式的特点：不需要使用任何扩号；非常适合用堆栈和计算机程序实现计算。



## 3.3 栈的应用 续19

**(2) 前缀式与后缀式的手工求值方法**

**a. 前缀式**

a+b×5/(3-c)

+ a / × b 5 - 3 c

3-c

b × 5

(b×5)/(3-c)

a+(b×5)/(3-c)



## 3.3 栈的应用 续20

**b. 后缀式**

a+b×5/(3-c)

a b 5 × 3 c - / +

b×5

3-c

(b×5)/(3-c)

a+(b×5)/(3-c)



## 3.3 栈的应用 续21

**(3) 中缀式变换为前缀式与后缀式的手工方法**

a. 对中缀表达式加完全括号；

b. 将每个运算符移动到本层次括号之前(之后)；

c. 去掉所有括号，即得前(后)缀表达式。

例如，将中缀式a+b×5/(3-c)分别化为前缀式与后缀式。

(a + ( ( b × 5 ) / ( 3 - c ) ) ) //加完全括号

+(a / (×( b 5 ) - ( 3 c ) ) ) //算符移动到本层括号之前

+ a / × b 5 - 3 c //去除所有括号得前缀式

(a ( ( b 5 ) × ( 3 c ) - ) / ) + //算符移动到本层括号之后

a b 5 × 3 c - / + //去除所有括号得后缀式



## 3.3 栈的应用 续22

**(4) 后缀表达式求值的原理性程序算法**

```
//输入以#号结束的后缀式字符序列，求表达式值
ValueTp f()
{ initStack(S); v=0; //S表示操作数栈
  while(1)
  { ch=getchar(); if(ch=='#') break;
    if(isOperator(ch)) { pop(S, b); pop(S, a);
                         push(S, compute(a, ch, b));
                       } //仅能处理二元运算符
      else push(S, ch); //是操作数则直接入栈
  } //运行过程中只要出现退空栈, 则表达式语法错
  pop(S, v); return v; }
```

## 3.3 栈的应用 续23

**(5) 中缀表达式求值的原理性程序算法**

**算法设计思想**：

a. 中缀式以#字符结束，#字符也作为算符处理；

b. ()作为算符处理，括号内运算结束后需要消除括号；

c. 除操作数栈(OPND)外，另需设立算符栈(OPTR)；

d. 算法主要步骤：

1) 初始时，置OPND为空栈，#字符作为OPTR栈底元素；

2) 依次读入表达式中每个字符，若是操作数，直接压入OPND栈；



## 3.3 栈的应用 续24

3) 若是运算符(记为$\theta$)，则和OPTR栈顶运算符(记为$\lambda$)比较优先级：

$\lambda<\theta$，$\theta$压入OPTR栈，继续读下一个字符；

$\lambda=\theta$，脱括号，继续读下一个字符；

$\lambda>\theta$，执行$\lambda$运算($\lambda$退栈)，$\theta$等在栈外(不读下一个字符)，即$\theta$继续和OPTR栈顶算符比较优先级(重复进行以上操作)，直至$\theta$能入栈。



## 3.3 栈的应用 续24

运算符优先级表

$\lambda$: 栈顶算符

$\theta$: 栈外算符

| $\lambda$ \ $\theta$ | + | - | * | / | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| - | > | > | < | < | < | > | > |
| * | > | > | > | > | < | > | > |
| / | > | > | > | > | < | < | > |
| ( | < | < | < | < | < | = | |
| ) | > | > | > | > | | > | > |
| # | < | < | < | < | < | | = |



## 3.3 栈的应用 续25

```
ValueTp g() //中缀表达式求值算法
{ initStack(OPND); initStack(OPTR); push(OPTR, '#');
  θ=getchar();
  while(θ≠'#'||get_top(OPTR)≠'#')
    if(!isOperator(θ)) { push(OPND, θ); θ=getchar(); }
      else switch(compare(get_top(OPTR), θ))
            { case <: push(OPTR, θ); θ=getchar(); break;
              case =: pop(OPTR, λ); θ=getchar(); break;
              case >: pop(OPTR, λ);
                      pop(OPND, b); pop(OPND, a);
                      push(OPND, compute(a, λ, b));break;
            }
```



## 3.3 栈的应用 续26

```
  pop(OPND, v);
  return v;
}
```

**运算符优先级的改进计算方法**

| 算符 | 栈中算符优先级(isp) | 栈外算符优先级(icp) |
|---|---|---|
| ) | 4 | 1 |
| * / | 3 | 3 |
| + - | 2 | 2 |
| ( | 1 | 4 |
| # | 0 | 0 |



## 3.3 栈的应用 续27

```
ValueTp g1() //中缀表达式求值算法(改进优先级比较)
{ initStack(OPND); initStack(OPTR); push(OPTR, '#');
  θ=getchar();
  while(θ≠'#'||get_top(OPTR)≠'#')
  {
    … //见下页PPT
  }
  pop(OPND, v);
  return v;
}
```

特点: 栈内算符优先级等于栈外算符优先级时，或者退括号(当栈外算符为')'时)，或者执行运算。

## 3.3 栈的应用 续28

```
{ if(!isOperator(θ)) push(OPND, θ);
  else {λ=get_top(OPTR); i=isp(λ); j=icp(θ);
        if(i<j) push(OPTR, θ);
        else { pop(OPTR, λ); //λ='('且i=j退括号
               if(λ≠'('&&i==j|| i>j)
               { pop(OPND, b); pop(OPND, a);
                 push(OPND, compute(a, λ, b));
                 continue; }
             } //end of i>=j
       }//处理运算符结束
  θ=getchar();
} //end of while
```



## 3.3 栈的应用 续29

例如，3+5*(7-2)#

| 步数 | OPND | OPTR | θ值 |
|---|---|---|---|
| | | # | |
| 1 | 3 | # | 3 |
| 2 | 3 | # + | + |
| 3 | 3 5 | # + | 5 |
| 4 | 3 5 | # + * | * |
| 5 | 3 5 | # + * ( | ( |
| 6 | 3 5 7 | # + * ( | 7 |
| 7 | 3 5 7 | # + * ( - | - |
| 8 | 3 5 7 2 | # + * ( - | 2 |
| 9 | 3 5 5 | # + * ( | ) |



## 3.3 栈的应用 续30

| 步数 | OPND | OPTR | θ值 | |
|---|---|---|---|---|
| 9 | 3 5 5 | # + * ( | ) | |
| 10 | 3 5 5 | # + * | ) | //消括号 |
| 11 | 3 5 5 | # + * | # | |
| 12 | 3 25 | # + | # | |
| 13 | 28 | # | # | |



## 3.3 栈的应用 续31

**(6) 中缀表达式化为后缀表达式的原理性程序算法**

**算法设计思想**：

在中缀表达式求值算法基础上改造。

a. 取消OPND栈，操作数入栈替换为输出该操作数；

b. 当isp(λ)>=icp(θ)时，执行以下操作：

pop(OPTR, λ);

if(λ!='(') 输出λ;

注意：icp(')')=1，只比'#'优先级高，因此')'实际上没有机会入栈。



## 3.3 栈的应用 续31

例如， a+b*5/(3-c)#

| 步数 | OPTR | θ值 | 输出 |
|---|---|---|---|
| | # | | |
| 1 | # | a | a |
| 2 | # + | + | a |
| 3 | # + | b | a b |
| 4 | # + * | * | a b |
| 5 | # + * | 5 | a b 5 |
| 6 | # + | / | a b 5 * |
| 7 | # + / | / | a b 5 * |
| 8 | # + / ( | ( | a b 5 * |
| 9 | # + / ( | 3 | a b 5 * 3 |



## 3.3 栈的应用 续32完

//a+b*5/(3-c)#

| 步数 | OPTR | θ值 | 输出 | |
|---|---|---|---|---|
| 9 | # + / ( | 3 | a b 5 * 3 | |
| 10 | # + / ( - | - | a b 5 * 3 | |
| 11 | # + / ( - | c | a b 5 * 3 c | |
| 12 | # + / ( | ) | a b 5 * 3 c - | //消括号 |
| 13 | # + / | # | a b 5 * 3 c - | |
| 14 | # + | # | a b 5 * 3 c - \ | |
| 15 | # | # | a b 5 * 3 c - \ + | |

## 第3章 作业

### 习题集

3.4 3.13 (做在习题集上，不交)
3.24 3.25
3.27* (选做)

### 上机作业

1. 输入一个非零正整数，输出其各位数字。要求数字之间间隔至少一个空格。
例如：输入12085, 输出为1 2 0 8 5。
要求：采用递归和非递归(用堆栈)两种算法。



## 第3章 作业

### 上机作业

2. 编写程序，实现中缀表达式化为后缀式输出。已知中缀表达式中操作数全部用小写英文字母表示，运算符只含有+, -, *, /四种，定界符只有( )以及结束符#。

# 第4章 串





## 4.1 串的概念与基本操作

**1. 串的概念**

串(**string**)也称为字符串，它是由零个或多个字符组成的有限序列，一般记为 $s$="$a_0a_1\ldots a_{n-1}$"。

**串长度**：指串中的字符数目。长度为0的串称为**空串**。

**子串**与**主串**：串任意个连续的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。

**字符位置**：指串中字符在序列中的序号，序号一般从0开始或者从1开始。

**串相等**：两个串长度相同并且对应字符位置字符完全相同，则两串相等。



## 4.1 串的概念与基本操作 续1

**2. 串的基本操作**

length(s) //求串长
charAt(s, i) //求i号位置上的字符
substr(s, i, j) //从i号到j号位置提取子串
equal(s1, s2) //比较两串是否相等
**index(s, sub)** //在主串s中查找子串sub的起始位置号
locate(s, ch) //在串S中查找字符ch的起始位置号
concat(s1, s2) //连接两个字符串
cmp(s1, s2) //比较字符串s1和s2的大小
…



## 4.1 串的概念与基本操作 续2完

**两个字符串怎样比较大小?**

字符串的每个字符用字符编码表示，则根据字符编码大小比较两串大小，比较方法如下：

设 $s_1$="$a_0a_1\ldots a_{m-1}$"，$s_2$="$b_0b_1\ldots a_{n-1}$"，则

$s_1<s_2$，iff $\exists i\left[(0\le i<\min(m,n))\wedge(a_{0..i-1}=b_{0..i-1})\wedge(a_i<b_i)\right]$
或 $(m<n)\wedge(a_{0..m-1}=b_{0..m-1})$；

$s_1>s_2$，iff $\exists i\left[(0\le i<\min(m,n))\wedge(a_{0..i-1}=b_{0..i-1})\wedge(a_i>b_i)\right]$
或 $(m>n)\wedge(a_{0..n-1}=b_{0..n-1})$；

$s_1=s_2$，iff $(m=n)\wedge(a_{0..m-1}=b_{0..m-1})$。



## 4.2 串的存储结构

**1. 顺序存储串**

一般用固定长度的字符数组存储串，串中每个字符存储的是其字符编码。

**(1) 常见形式1**

typedef unsigned char **StringTp1**[256];

其中，0号下标元素存储串长(最大允许串长为255)；串中字符序号从1开始。

**例如**，StringTp1 s={4, 'A', 'B', 'C', 'D'}, t={0};

则s串长为4，t为空串。



## 4.2 串的存储结构 续1

**(2) 常见形式2**

```
typedef struct { char s[MAX_LEN];
                 //或char *s; 内存运行时动态分配
                 int max_len; //数组最大容量
                 int len; //串长
                } StringTp2;
```

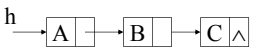
**(3) 常见形式3**

```
char s[MAX_LEN+1]={ 'A', 'B', 'C', 'D', '\0' }; //C语言串
//或char *s=malloc(MAX_LEN+1);
//用字符'\0'结尾；串中字符的序号从0开始
```

## 4.2 串的存储结构 续2

### 2. 链式存储串


//结点字符容量为4，#表示结束占位标志字符




//结点字符容量为1

//链式串多数操作比顺序存储串复杂，很少实际应用。



## 4.2 串的存储结构 续3

**例1** 分别用StringTp1以及C语言串实现两串比较大小。

```
//StringTp1两串比较大小
int cmp(StringTp1 s, StringTp1 t)
{ for(i=1; i<=s[0]&&i<=t[0]&&s[i]==t[i]; i++);
  if(i<=s[0]&&i>t[0]) return -1;
  if(i>s[0]&&i<=t[0]) return +1;
  if(i>s[0]&&i>t[0]) return 0;
  return s[i]-t[i];
}
//返回负整数表示串s<串t
//返回正整数表示串s>串t
//返回0表示两串相等
```



## 4.2 串的存储结构 续4完

```
//C语言两串比较大小
int cmp(char *s, char *t)
{ for(i=0; s[i]&&t[i]&&s[i]==t[i]; i++);
   return s[i]-t[i];
}
//返回值情况与StringTp1串比较大小相同
```



## 4.3 串的模式匹配算法

### 1. 什么是串的模式匹配？

串的模式匹配与子串查找(子串定位)的概念相同，即在主串$s$中查找子串$t$(**称为模式**)第一次出现的起始位置号。

**例如**，$s$="ABCDEF"，$t$="CD"，若字符位置序号从0开始，则$t$在$s$中的匹配位置号2；

$s$="aabaaaa"，$t$="aa"，则$t$在$s$中共有4次匹配，匹配位置号分别是0、3、4、5。

**匹配成功**：子串包含于主串中

**匹配失败**：子串不包含于主串中



## 4.3 串的模式匹配算法 续1

### 2. 子串定位基本算法

```
//匹配成功返回第1次匹配的位置号；匹配失败返回-1
int index(char *s, char *t)
{ i=j=0;
  while(s[i]&&t[j])
    if(s[i]==t[j]) { i++; j++; }
      else { i=i-j+1; j=0; }
        //失配时，i回溯至主串原出发位置的下一个位置
  if(!t[j]) return i-j; //j的值此时为子串t长度
  return -1; //匹配失败返回-1
}
```



## 4.3 串的模式匹配算法 续2

**子串失配时，主串下标i怎样回溯？**

$$\begin{array}{ccccccccc} \cdots & s_x & s_{x+1} & \cdots & s_{i-1} & s_i & \cdots \\ & \| & \| & & \| & \not\| & \\ & t_0 & t_1 & \cdots & t_{j-1} & t_j & \cdots \end{array}$$

由上图可知，当$s_{x..i-1}=t_{0..j-1}$且$s_i \neq s_j$时，匹配失败(即主串$x$位置不可能有成功匹配)；此时，$i$ 应回溯至$x+1$位置准备试探$x+1$位置是否能成功匹配子串。

显然，$i-x=j-0$，即$x=i-j$，有$x+1=i-j+1$。

## 4.3 串的模式匹配算法 续3

**3. 子串定位的一种改进算法-KMP算法**

**改进点**：**子串失配时，主串下标*i*不需要回溯**；KMP算法的时间复杂度为$O(m+n)$，其中$m$, $n$分别为主串、子串长度。

**算法发现者**：D. E. Knuth, J. H. Morris和V. R. Pratt同时发现，故命名为**KMP**算法，即克努特-莫里斯-普拉特算法。

**(1) 算法思想**

当子串$j$位置与主串$i$位置失配时，能否让主串下标$i$不动，将子串$k$号位置($k<j$)与主串$i$号位置对应比较大小?



## 4.3 串的模式匹配算法 续4

这一想法若有可能实现，需要满足以下两个条件：

**(a) t[0..k-1]=s[i-k..i-1]**

**//子串t[k]之前k个字 符与主串s[i]之前k个字符相等**

**//接下来需要比较s[i]与t[k]是否相等**

**(b) 主串小于i-k号位置不可能有成功匹配**

为满足条件**(b)**，只需要在1≤k<j范围内取满足**(a)**的最大k，于是条件**(a)(b)**可合并表示为以下条件。

**t[0..k-1]=s[i-k..i-1](1≤k<j), 其中k满足以下条件：不存在k', 使1≤k'<j且k'>k且t[0..k'-1]=s[i-k'..i-1]**。



## 4.3 串的模式匹配算法 续5

条件**(a)(b)**的另一种等价表达形式还可以是：

若存在k满足下式

$$\max\{k|1\le k<j\text{且}t[0..k-1]=s[i-k..i-1]\} \quad (c)$$

则当s[i]与t[j]失配时，可将模式串t向右滑动，使t[k]与s[i]对齐，然后从t[k]与s[i]开始比较字符；

若不存在下标k满足条件**(c)**，则说明主串i号位置之前都不可能有成功匹配，则令 j=0; (即主串从i位置开始新一轮匹配)；若s[i]≠t[0]，则i=i+1; 从主串下个位置开始新一轮匹配。



## 4.3 串的模式匹配算法 续6

**当s[i]与t[j]失配时，若存在下标k满足条件 (c)，为什么主串中下标i-k之前不可能有子串的成功匹配?**

证明：显然主串下标小于或等于i-j范围内已无成功匹配；只需说明主串下标范围i-j+1..i-k-1内不可能有成功匹配。

用反证法，设主串下标范围i-j+1..i-k-1内有下标i'是子串的成功匹配，则s[i'..i-1]=t[0..i-i'-1]，

因i-j+1≤i'≤i-k-1，有k+1≤i-i'≤j-1，即存在k'=i-i'满足条件**(c)**(t[0..k'-1]=s[i-k'..i-1])，而k'>k，这与k最大矛盾。



## 4.3 串的模式匹配算法 续7

**对子串任意下标位置j，怎样求满足条件(c)的k值?**

E. Knuth等人发现，这个问题的求解与主串无关(独立于主串)，它只与子串自身有关。

**可以证明，条件(c)与以下条件(d)等价**。

$$\max\{k|1\le k<j\text{且}t[0..k-1]=t[j-k..j-1]\} \quad (d)$$

证明：先证条件**(c)**成立时，条件**(d)**成立。

因为条件**(c)**成立，有**t[0..k-1]=s[i-k..i-1]**，

当s[i]与t[j]失配(s[i]≠t[j])时，有**s[i-j..i-1]=t[0..j-1]**，

因为k<j，故i-k>i-j，必有**s[i-k..i-1]=t[j-k..j-1]**，

所以，t[0..k-1]=t[j-k..j-1]，即条件**(d)**成立。



## 4.3 串的模式匹配算法 续8

再证条件**(d)**成立时，条件**(c)**成立。

因为条件**(d)**成立，有**t[0..k-1]=t[j-k..j-1]**，

当s[i]与t[j]失配(s[i]≠t[j])时，有**s[i-j..i-1]=t[0..j-1]**，

因为k<j，故i-k>i-j，必有**s[i-k, i-1]=t[j-k, j-1]**，

所以，t[0..k-1]=s[i-k, i-1]，即条件**(c)**成立。

## 4.3 串的模式匹配算法 续9

**(2) next数组**

根据条件**(d)**以及前面的分析，我们将子串(模式串)的每一个下标位置j对应的满足条件**(d)**的k值存于名为next的数组中(next数组长度与子串长度相同)；若某个j位置不存在满足条件**(d)**的k，令next[j]=**0**，表示子串0号位置继续与主串失配时的i位置比较；若j=0时子串与主串i位置失配，则应使i=i+1; 于是可令next[0]=@，@表示特殊标志值(比如：@可取任一负整数，为方便模式匹配算法的操作，一般令next[0]=**-1**)。

于是，我们得到完整的next数组值的定义如下：



## 4.3 串的模式匹配算法 续10

$$nex[j]=\begin{cases}-1 & \text{当j=0时}\\ \max\{k|1\le k<j\text{且}t[0..k-1]=t[j-k..j-1]\} & \text{若k存在}\\ 0 & \text{其它情况}\end{cases}$$

**注意**：若数组及串中字符下标从1开始，则每个next[j]值在上述值基础上加1。

**(3) next数组值的手工求法**

**方法**：对子串任意j位置(j=0, 1, 2, …, n-1, n为子串长度)在1≤k<j范围内观察子串j号位置之前是否有k个字符与子串开头的k个字符相同(k由大到小试探)。



## 4.3 串的模式匹配算法 续11

**例2** 若子串为"aaaaaa"，写出KMP算法next数组值。

**解**：

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 子串 | a | a | a | a | a | a |
| next | -1 | 0 | 1 | 2 | 3 | 4 |

**例3** 若子串为"abaabcac"，写出KMP算法next数组值。

**解**：

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 子串 | a | b | a | a | b | c | a | c |
| next | -1 | 0 | 0 | 1 | 1 | 2 | 0 | 1 |



## 4.3 串的模式匹配算法 续12

**(4) 计算next数组值的程序算法**

**基本思路**：用递推法

1) 设next[0]=**-1**;

2) 设next[0..j-1] (j=1, 2, …)已求得，以下分析求next[j]的方法。

令 $\begin{cases}k_0=\text{next}[j-1],\\ k_p=\text{next}[k_{p-1}],\ p=1,2,3,\ldots\end{cases}$

显然，$k_0$是满足条件$t[0..k_0-1]=t[j-k_0-1..j-2]$的最大整数。

不失一般性，设$k_p$($p$=1, 2, 3, …)是满足以下条件的最大整数：$k_{p-1}>k_p$且$t[0..k_p-1]=t[j-k_p-1..j-2]$，



## 4.3 串的模式匹配算法 续13

以下证明$k_{p+1}$是满足下列条件的最大整数：

$k_p>k_{p+1}$且$t[0..k_{p+1}-1]=t[j-k_{p+1}-1..j-2]$。

**证明**：若$k_{p+1}=\text{next}[k_p]$，则

$t[0..k_{p+1}-1]=t[k_p-k_{p+1}..k_p-1]$ (P1)

已知 $t[0..k_p-1]=t[j-k_p-1..j-2]$ (P2)

显然，(P1)式的等号右边$t[k_p-k_{p+1}..k_p-1]$是(P2)式的等号左边$t[0..k_p-1]$的右边长度为$k_{p+1}$的子串，因(P2)成立，必有$t[k_p-k_{p+1}..k_p-1]=t[j-k_{p+1}-1..j-2]$成立，

再由(P1)可得$t[0..k_{p+1}-1]=t[j-k_{p+1}-1..j-2]$。



## 4.3 串的模式匹配算法 续14

尚需证明，不存在整数$k'$，使

$k_p>k'>k_{p+1}$且$t[0..k'-1]=t[j-k'-1..j-2]$。

用反证法，因为(P2)成立，且$k_p>k'$，则$t[j-k'-1..j-2]$是$t[j-k_p-1..j-2]$右边长度为$k'$的子串，有$t[k_p-k'..k_p-1]=t[j-k'-1..j-2]$，因为$t[0..k'-1]=t[j-k'-1..j-2]$，得$t[0..k'-1]=t[k_p-k'..k_p-1]$，这表明$\text{next}[k_p]\ge k'>k_{p+1}$，这与$\text{next}[k_p]=k_{p+1}$矛盾。

## 4.3 串的模式匹配算法 续15

序列 { $k_0, k_1, k_2, \ldots$ }是一个递减的下标序列，且最后两个下标必为0, -1。设$k_{q-1}=0$, $k_q=-1$，则$k_0, k_1, \ldots, k_{q-2}$均大于0，即$k_0>k_1>k_2>\cdots>k_{q-2}>k_{q-1}=0>k_q=-1$。

$$next[j]=\begin{cases} k_p+1 & \text{若存在}p,\ p<q\text{且}t[k_0], t[k_1], t[k_2], \ldots, t[k_{p-1}]\text{均不等于}t[j-1]\text{且}t[k_p]=t[j-1] \\ 0=k_q+1 & t[k_0], t[k_1], \ldots, t[k_{q-1}]\text{均不等于}t[j-1] \end{cases}$$

因为$k_q$=next[0]=@，为了使$k_q+1=0$，令@=-1



## 4.3 串的模式匹配算法 续16

```
//求next数组值原理算法
void get_next(char *t, int next[])
//next数组长度应大于或等于strlen(t)
{ next[0]=-1;
  for(j=1; t[j]; j++)
    { k=next[j-1];   //k=k0
      while(k!=-1&&t[j-1]!=t[k]) k=next[k];
      //k=k1, k2, …
      next[j]=k+1;  //next[j]=kp+1或kq+1
    }
}
```



## 4.3 串的模式匹配算法 续16

```
//求next数组值正式算法(双重循环合并为一重循环)
void get_next(char *t, int next[])
//next数组长度应大于或等于strlen(t)
{ int j, k;
  next[0]=k=-1; j=1;
  while(t[j])
    if(k==-1||t[j-1]==t[k]) next[j++]=++k;
      else k=next[k];
}
```



## 4.3 串的模式匹配算法 续17

**(4) KMP模式匹配算法**

```
int index_kmp(char *s, char *t, int next[])
//要求next值已求取
{ i=j=0;
  while(s[i]&&t[j])
    if(j==-1||s[i]==t[j]) { i++; j++; }
      else  j=next[j];
  if(!t[j]) return i-j;
  return -1;   //返回-1表示匹配失败
}
```



## 4.3 串的模式匹配算法 续18

**(5) 改进的next数组值nextval**

**改进思路**：设next[j]=$k_0$，

则当s[i-j..i-1]=t[0..j-1]而s[i]≠t[j]时，需要将子串t向右滑动，使子串t[$k_0$]与主串s[i]对齐。

由于s[i]≠t[j]是已知的，若有t[$k_0$]=t[j]，必有t[$k_0$]≠s[i]，这时不必将t[$k_0$]与s[i]对齐，可以继续向右滑动子串，使t[$k_1$]与s[i]对齐(其中，$k_1$=next[$k_0$])；类似的，若t[$k_1$]=t[j]，则t[$k_1$]≠s[i]，则令t[$k_2$]与s[i]对齐(其中，$k_2$=next[$k_1$])，…，直到经过$p$次右滑后，有$k_p$=-1或t[$k_p$] ≠ t[j]，则停止右滑。



## 4.3 串的模式匹配算法 续19

**改进要点**：在nextval数组中，若nextval[j]=k，则k是最大整数，满足

t[0..k-1]=t[j-k..j-1]且t[k]≠t[j] **(e)**

对比next[j]=k时，t[k]≠t[j]不一定满足。

为达到改进目的，设next[j]=k，若k≥0且t[k]=t[j]，则令nextval[j]=nextval[k];否则，令nextval[j]=next[j]。

以下证明nextval[j]=nextval[k]可确保**(e)**成立。

**证明**：已知next[j]=k，k≥0，t[k]=t[j]，在这些条件下求nextval[j]。因为k<j，故nextval[k]的值已经先于nextval[j]求得，设nextval[k]=k'，则

## 4.3 串的模式匹配算法 续20

根据nextval数组的特点，必有t[k']≠t[k]，
而已知t[k]=t[j]，故**t[k']≠t[j]**；
由next[j]=k知，t[0..k-1]=t[j-k..j-1]，
由nextval[k]=k'知，t[0..k'-1]=t[k-k'..k-1]，
因为k'<k， t[k-k'..k-1]为t[0..k-1]的左边片段，必有
t[k-k'..k-1]=t[j-k'..j-1]，于是**t[0..k'-1]=t[j-k'..j-1]**。
以下用反证法证明k'是最大整数满足
**t[0..k'-1]=t[j-k'..j-1]且t[k']≠t[j]**。
设存在k">k'使t[0..k"-1]=t[j-k"..j-1]且t[k"]≠t[j];



## 4.3 串的模式匹配算法 续21

显然，k"<k，这是因为k是最大整数，使得
t[0..k-1]=t[j-k..j-1]成立，故k"≤k；而t[k]=t[j]，t[k"]≠t[j]，
故只能是k"<k(即k"=k不可能)。
于是， t[j-k"..j-1]是t[j-k..j-1]的右边长度为k"的片段，
必有t[j-k"..j-1]=t[k-k"..k-1]，因t[0..k"-1]=t[j-k"..j-1]，有
t[0..k"-1]=t[k-k"..k-1]；t[k]=t[j]且t[k"]≠t[j]，有t[k"]≠t[k]，
于是nextval[k]≥k">k'，这与nextval[k]=k'矛盾。



## 4.3 串的模式匹配算法 续22

```
//求nextval数组值算法
void get_nextval(char *t, int nextval[])
//nextval数组长度应大于或等于strlen(t)
{ int j, k;
  nextval[0]=k=-1; j=1;
  while(t[j])
    if(k==-1||t[j-1]==t[k])
       if(t[++k]==t[j]) nextval[j++]=nextval[k];
         else nextval[j++]=k;
      else k=nextval[k];
} //要点：k值仍然跟踪的是未改进的next[j]值
```



## 4.3 串的模式匹配算法 续23

**问题：在nextval数组求值算法中，为什么说变量k值跟踪的是next数组元素值？**

**答**：只需要证明当next[j-1]=k且t[k]≠t[j-1]时，以下两段迭代算法的迭代结果相同。

while(k!=-1&&t[k]!=t[j-1]) k=**next**[k]; **(i1)**

while(k!=-1&&t[k]!=t[j-1]) k=**nextval**[k]; **(i2)**

**证明**：对迭代**(i1)**，前面我们曾分析得到递减的下标序列 $k_0>k_1>k_2>\ldots>k_{q-1}=0>k_q=-1$，其中，
$k_0$=next[j-1], $k_1$=next[$k_0$], …, $k_q$=next[$k_{q-1}$]。



## 4.3 串的模式匹配算法 续24

**1)** 在迭代(i1)中，第一次循环计算的是$k_1$=next[$k_0$]。
**a.** 当$k_1$=-1时，nextval[$k_0$]= next[$k_0$]=-1；
**b.** 当$k_1$≠-1且t[$k_1$]≠t[$k_0$]时，nextval[$k_0$]=next[$k_0$]=$k_1$；
故，迭代(i1)和(i2)第一次循环时情况**a**. **b**. 结果相同。
**c.** 当$k_1$≠-1且t[$k_1$]=t[$k_0$]时，以下证明，迭代(i1)一定会迭代至k=nextval[$k_0$]。
**证**：迭代(i1)的第一次循环结束时，
若$k_1$≠-1且t[$k_1$]=t[$k_0$]，因为t[$k_0$]≠t[j-1]，有
t[$k_1$]≠t[j-1]，循环需要再次进行，即计算
$k_2$=next[$k_1$]，若$k_2$≠-1且t[$k_2$]=t[$k_1$]，则t[$k_2$]≠t[j-1]，



## 4.3 串的模式匹配算法 续25

迭代(i1)需进行第3次循环，计算$k_3$=next[$k_2$]，……
不失一般性，
**c1**. 设迭代(i1)进行了如下所示的$u$次循环，
第1次循环： $k_1$=next[$k_0$]， $k_1$≠-1且t[$k_1$]=t[$k_0$]≠t[j-1]
第2次循环： $k_2$=next[$k_1$]， $k_2$≠-1且t[$k_2$]=t[$k_1$]≠t[j-1]
…
第$u$-1次循环：$k_{u-1}$=next[$k_{u-2}$]，$k_{u-1}$≠-1且t[$k_{u-1}$]=t[$k_{u-2}$]≠t[j-1]
第$u$次循环：$k_u$=next[$k_{u-1}$]，$k_u$≠-1且t[$k_u$]≠t[$k_{u-1}$]
则易证明，**$k_u$=nextval[$k_0$]**。
证: $k_1$=next[$k_0$]⇒t[0..$k_1$-1]=t[$k_0$-$k_1$..$k_0$-1]，(e1)

## 4.3 串的模式匹配算法 续26

$k_2$=next[$k_1$]⇒t[0..$k_2$-1]=t[$k_1$-$k_2$..$k_1$-1]，(e2)
$k_1$>$k_2$，(e1)和(e2)⇒t[0..$k_2$-1]=t[$k_0$-$k_2$..$k_0$-1]，
…
$k_u$=next[$k_{u-1}$]⇒t[0..$k_u$-1]=t[$k_{u-1}$-$k_u$..$k_{u-1}$-1]
必有t[0..$k_u$-1]=t[$k_0$-$k_u$..$k_0$-1]；
又因为t[$k_0$]=t[$k_1$]=…=t[$k_{u-1}$]≠t[$k_u$]，则$k_u$是最大整数，满足t[$k_0$]≠t[$k_u$]且t[0..$k_u$-1]=t[$k_0$-$k_u$..$k_0$-1]，故nextval[$k_0$]=$k_u$。
$k_u$=nextval[$k_0$]说明迭代(i1)经过u次循环后，与迭代(i2)第一次循环结果相同。



## 4.3 串的模式匹配算法 续27

**c2**. 设迭代(i1)进行了如下所示的$v$次循环，
第1次循环： $k_1$=next[$k_0$]， $k_1$≠-1且t[$k_1$]=t[$k_0$]≠t[j-1]
第2次循环： $k_2$=next[$k_1$]， $k_2$≠-1且t[$k_2$]=t[$k_1$]≠t[j-1]
…
第$v$-1次循环：$k_{v-1}$=next[$k_{v-2}$]，$k_{v-1}$≠-1且t[$k_{v-1}$]=t[$k_{v-2}$]≠t[j-1]
第$v$次循环：$k_v$=next[$k_{v-1}$]，$k_v$=**-1**
则易证明，**-1=nextval[$k_0$]**。
即对**c2**情况，迭代(i1)经过$v$次循环后，结果与nextval[$k_0$]一致。



## 4.3 串的模式匹配算法 续28

综合**a.b.c.**三种情况，可知迭代(i1)经过$u$次或$v$次循环后，结果与迭代(i2)一次循环结果一致。

迭代(i1)经过$u$次循环后，有
$k_u$=next[$k_{u-1}$]=nextval[$k_0$]，$k_u$≠-1且t[$k_u$]≠t[$k_{u-1}$]，若t[$k_u$]≠t[j-1]，则迭代(i1)(i2)均需继续循环。
类似可证明，迭代(i1)经过若干次循环后，变量k值与迭代(i2)第二次循环后的k值结果nextval[$k_u$]相同。
总之，迭代(i1)结果与迭代(i2)完全相同。



## 4.3 串的模式匹配算法 续29完

**例4** 若子串为"aaaaaa"，写出KMP算法nextval数组值。
**解**：

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 子串 | a | a | a | a | a | a |
| next | -1 | 0 | 1 | 2 | 3 | 4 |
| nextval | -1 | -1 | -1 | -1 | -1 | -1 |

**例5** 若子串为"abaabcac"，写出KMP算法nextval数组值。
**解**：

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 子串 | a | b | a | a | b | c | a | c |
| next | -1 | 0 | 0 | 1 | 1 | 2 | 0 | 1 |
| nextval | -1 | 0 | -1 | 1 | 0 | 2 | -1 | 1 |



## 第4章 作业

**习题集**
4.7
4.31*(选作)

**上机编程作业**
从键盘输入主串s以及子串t1, t2，试将主串s中所有t1子串替换为t2子串，输出替换后得到的串以及t1被替换的次数。要求子串查找采用改进KMP算法。

# 第5章 数组与广义表





## 5.1 数组的定义

### 1. 一维数组的定义

一个长度为$n_1$($n_1 \geq 1$)的一维数组定义为含有$n_1$个相同类型数据元素的线性表。

### 2. $m$维数组的定义

一个大小为$n_1 \times n_2 \times \cdots \times n_m$的$m$维数组可视为一个由$n_1$个大小为$n_2 \times n_3 \times \cdots \times n_m$的$m$-1维数组组成的线性表。其中，$n_k$表示第$k$维($k$=1, 2, …, $m$)的大小，$n_k \geq 1$。

### 3. $m$维数组元素与元素的下标

大小为$n_1 \times n_2 \times \cdots \times n_m$的$m$维数组共有$n_1 \times n_2 \times \cdots \times n_m$个相同类型的数据元素；



## 5.1 数组的定义 续1完

每个数据元素用一个如下所示$m$元下标向量确定。

$(j_1, j_2, \ldots, j_m)$

其中，$j_k \in j_k^{(0)} .. j_k^{(0)} + n_k - 1$，$k$=1, 2, …, $m$

$j_k^{(0)}$ 是任意整数，表示第$k$维下标的起点，

对于**C/C++/Java语言**，$j_k^{(0)}$=**0**，对于MATLAB，$j_k^{(0)}$=1，

对于PASCAL语言，$j_k^{(0)}$ 可定义为任意整型值。



## 5.2 数组的顺序表示和实现

### 1. 数组的顺序存储表示

大小为$n_1 \times n_2 \times \cdots \times n_m$的$m$维数组一般采用顺序存储实现各数据元素的存储。

设数组数据元素类型为ElemTp，则上述$m$维数组的总存储空间大小为sizeof(ElemTp)×$n_1 \times n_2 \times \cdots \times n_m$。

### (1) 行序为主序

顺序存储数组元素时，元素左边的下标变化慢。对二维数组，就是按行号(第1维下标)次序存储各行，每行按列号(第2维下标)次序存储各元素。

**C/C++/Java语言**数组以行序为主序实现顺序存储。



## 5.2 数组的顺序表示和实现 续1

### (2) 列序为主序

顺序存储数组元素时，元素左边的下标变化快。对二维数组，就是按列号(第2维下标)次序存储各列，每列按行号(第1维下标)次序存储各元素。

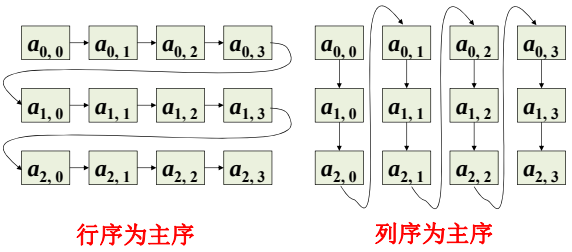
Fortran语言/Matlab等以列序为主序实现数组元素顺序存储。

**例如**，二维数组大小为3×4(3行4列)，设元素下标从0开始，$i$行$j$元素记为$a_{i,j}$，则该数组按行序为主序以及列序为主序的各元素顺序存储次序如下图所示。



## 5.2 数组的顺序表示和实现 续2

### 3行4列二维数组的元素顺序存储次序




行序为主序　　列序为主序

## 5.2 数组的顺序表示和实现 续3

**2. 实现数组元素的访问**

**(1) 已知元素下标求元素存储地址**

大小为$n_1\times n_2\times\cdots\times n_m$的$m$维数组，记元素下标为$(j_1, j_2, \ldots, j_m)$，其中，$j_k\in$**0**..$n_k$-1，$k$=1, 2, …, $m$，记每个元素的存储空间大小为***L***字节，数组第1个元素的存储地址为**0**，以下给出元素$(j_1, j_2, \ldots, j_m)$的存储地址LOC$(j_1, j_2, \ldots, j_m)$。

**a. 行序为主序时**

$$\text{LOC}(j_1, j_2, \ldots, j_m)=[j_1\times(n_2\times n_3\times\cdots\times n_m)+j_2\times(n_3\times n_4\times\cdots\times n_m)+\cdots+j_{m-1}\times n_m+j_m]\times L$$



## 5.2 数组的顺序表示和实现 续4

为简化表示，设

$$\begin{cases} c_1=1 \\ c_k=n_{m-k+2}c_{k-1} & k=2, 3, \ldots, m \end{cases}$$

则

$$\text{LOC}(j_1, j_2, \ldots, j_m)=\left(\sum_{k=1}^{m} j_{m-k+1}c_k\right)\cdot L$$

$c_1$=1
$c_2$=$n_m$
$c_3$=$n_{m-1}\times n_m$
…
$c_{m-1}$=$n_3\times n_4\times\cdots\times n_m$
$c_m$=$n_2\times n_3\times\cdots\times n_m$

令 $off(j_1, j_2, \ldots, j_m)=\left(\sum_{k=1}^{m} j_{m-k+1}c_k\right)$，数组首地址为A0，

则**完整的元素寻址公式为**

$$\text{LOC}(j_1, j_2, \ldots, j_m)=\text{A0}+off(j_1, j_2, \ldots, j_m)\cdot L$$



## 5.2 数组的顺序表示和实现 续5

若把$m$维数组视为一维数组，则 $off(j_1, j_2, \ldots, j_m)$表示的是$m$维数组元素在一维数组中的下标(假设下标从0开始)。

若多维数组各维起始下标不为0，则计算 $off(j_1, j_2, \ldots, j_m)$ 前需令 $j_k \leftarrow j_k - j_k^{(0)}$。

若下标越界，即存在$k$，$1\le k\le m$，$j_k < j_k^{(0)}$ 或 $j_k > j_k^{(0)}+n_k-1$，则不能正确找到元素地址。



## 5.2 数组的顺序表示和实现 续6

**b. 列序为主序时**

对$n_1\times n_2\times\cdots\times n_m$大小的$m$维数组，若元素下标从0开始，

$off(j_1, j_2, \ldots, j_m)=j_m\times(n_1\times n_2\times\cdots\times n_{m-1})+j_{m-1}\times(n_1\times n_2\times\cdots\times n_{m-2})+\cdots+j_2\times n_1+j_1$

为简化表示，设

$$\begin{cases} c_1=1 \\ c_k=n_{k-1}c_{k-1} & k=2, 3, \ldots, m \end{cases}$$

则 $off(j_1, j_2, \ldots, j_m)=\sum_{k=1}^{m} j_k c_k$

$c_1$=1
$c_2$=$n_1$
$c_3$=$n_2\times n_1$
…
$c_{m-1}$=$n_{m-2}\times n_{m-3}\times\cdots\times n_1$
$c_m$=$n_{m-1}\times n_{m-2}\times\cdots\times n_1$



## 5.2 数组的顺序表示和实现 续7

**(2) 已知元素存储地址求元素下标**

大小为$n_1\times n_2\times\cdots\times n_m$的$m$维数组，记元素下标为$(j_1, j_2, \ldots, j_m)$，其中，$j_k\in$**0**..$n_k$-1，$k$=1, 2, …, $m$，记每个元素的存储空间大小为***L***字节，数组第1个元素的存储地址为A0，若LOC$(j_1, j_2, \ldots, j_m)$已知，为求元素下标，首先需要计算

$$off(j_1, j_2, \ldots, j_m)=\frac{\text{LOC}(j_1, j_2, \ldots, j_m)-\text{A0}}{L},$$

然后由 $off(j_1, j_2, \ldots, j_m)$ 的值以及$n_1, n_2, \ldots, n_m$的值可计算出下标$j_1, j_2, \ldots, j_m$，具体方法如下。



## 5.2 数组的顺序表示和实现 续8

**a. 行序为主序时**

令 $\begin{cases} b_1=n_2\times n_3\times\cdots\times n_m \\ b_k=b_{k-1}/n_k \end{cases}$ ⇒

$b_1$=$n_2\times n_3\times\cdots\times n_m$
$b_2$=$n_3\times n_4\times\cdots\times n_m$
…
$b_{m-1}$=$n_m$
$b_m$=1

令 $t_1=off(j_1, j_2, \ldots, j_m)$，则

$j_1=t_1/b_1$, $t_2=t_1\%b_1$; //此处/和%为C语言运算符

$j_2=t_2/b_2$, $t_3=t_2\%b_2$;

…

$j_{m-1}=t_{m-1}/b_{m-1}$, $t_m=t_{m-1}\%b_{m-1}$;

$j_m=t_m$

## 5.2 数组的顺序表示和实现 续9

//C语言算法

**b**=$n_1$×$n_2$×…×$n_m$; **t**=*off*($j_1$, $j_2$, …, $j_m$);

for(*k*=1; *k*<=*m*; *k*++){**b**/=$n_k$; $j_k$=**t**/**b**; **t**%=**b**;}

**b. 列序为主序时**

令 $\begin{cases} b_1=n_1\times n_2\times\cdots\times n_{m-1} \\ b_k=b_{k-1}/n_{m-k+1} \end{cases} \Rightarrow$ $\begin{aligned} &b_1=n_1\times n_2\times\cdots\times n_{m-1}\\ &b_2=n_1\times n_2\times\cdots\times n_{m-2}\\ &\cdots\\ &b_{m-1}=n_1\\ &b_m=1 \end{aligned}$

//C语言算法

**b**=$n_1$×$n_2$×…×$n_m$; **t**=*off*($j_1$, $j_2$, …, $j_m$);

for(*k*=*m*; *k*>=1; *k*--){**b**/=$n_k$; $j_k$=**t**/**b**; **t**%=**b**;}



## 5.2 数组的顺序表示和实现 续10

**例1** 某3行4列二维数组每个元素的存储长度为4字节，记*i*行*j*列元素为$a_{i,j}$，行、列下标均从0开始。若数组首地址为1000，分别求行序为主序以及列序为主序顺序存储时，数组元素$a_{1,2}$的存储地址。

**解**：a. 行序为主序时

LOC($a_{1,2}$)=1000+(1×4+2)×4=1000+24=1024

b. 列序为主序时

LOC($a_{1,2}$)=1000+(2×3+1)×4=1000+28=1028



## 5.2 数组的顺序表示和实现 续11

**例2** 某大小为3×4×5×6的4维数组各元素采用顺序存储，首地址为1000，每个元素存储长度为8字节，某元素存储地址为3000，分别求出行序为主序以及列序为主序顺序存储时，该元素的下标(设元素下标从0开始)。

**解**：a. 行序为主序时

*t*=*off*($j_1$, $j_2$, $j_3$, $j_4$)=(3000-1000)/8=250，则

$j_1$=*t*/(4×5×6)=2，*t*=*t*%(4×5×6)=10

$j_2$=*t*/(5×6)=0，*t*=*t*%(5×6)=10

$j_3$=*t*/6=1，*t*=*t*%6=4

$j_4$=*t*=4



## 5.2 数组的顺序表示和实现 续12完

b. 行序为主序时

$j_4$=*t*/(3×4×5)=4，*t*=*t*%(3×4×5)=10

$j_3$=*t*/(3×4)=0，*t*=*t*%(3×4)=10

$j_2$=*t*/3=3，*t*=*t*%3=1

$j_1$=*t*=1

故，行序为主序时，该元素下标为(2, 0, 1, 4)；

列序为主序时，该元素下标为(1, 3, 0, 4)；



## 5.3 数组的压缩存储

**1. 对称矩阵**

**(1) 什么是对称矩阵？**

记*n*×*n* 阶方阵*i*行*j*列元素为$a_{i,j}$，设行、列下标均从0开始，若该矩阵与它的转置矩阵相等，即对任意*i*, *j*，有

$a_{i,j}=a_{j,i}$ (0≤*i*<*n*, 0≤*j*<*n*)

则该矩阵为对称矩阵。

**(2) 对称矩阵的压缩存储**

一般只存储下半三角(含主对角线)矩阵，且采用顺序存储，则对*n*×*n*阶对称矩阵，需要存储的元素总数为

1+2+3+…+*n*=*n*(*n*+1)/2



## 5.3 数组的压缩存储 续1

若对称矩阵下半三角元素存储首地址为A0，每个元素存储长度为*L*字节，则元素$a_{i,j}$的存储地址为

$$\text{LOC}(i,j)=\begin{cases} \text{A0}+off(i,j)\times L & \text{当}i\geq j\text{时} \\ \text{A0}+off(j,i)\times L & \text{当}i<j\text{时} \end{cases}$$

**(2.1) 已知*i*, *j*求*off*(*i*, *j*)**

**a. 行序为主序时**

**分析**：考虑*i*≥*j*，因行序为主序，在元素$a_{i,j}$之前需存储*i*行元素(行下标0~*i*-1)，在第*i*行上，$a_{i,j}$之前存储有*j*个元素(列号0~*j*-1)，故

## 5.3 数组的压缩存储 续2

$$off(i,j)=\left[\sum_{k=0}^{i-1}(k+1)\right]+j=(1+2+\cdots+i)+j=\frac{i(i+1)}{2}+j。$$

若数组下标从1开始，

$$off(i,j)=\left(\sum_{k=1}^{i-1}k\right)+j-1=\frac{i(i-1)}{2}+j-1。$$

**a. 列序为主序时**

分析：考虑**i≥j**，因列序为主序，在元素$a_{i,j}$之前需存储$j$列元素(列下标0~$j$-1)，在第$j$列上，$a_{i,j}$之前存储有$i$个元素(行号0~$i$-1)，故



## 5.3 数组的压缩存储 续3

$$off(i,j)=\left[\sum_{k=0}^{j-1}(n-k)\right]+i-j=\frac{j(2n-j+1)}{2}+i-j。$$

若数组下标从1开始，

$$off(i,j)=\left(\sum_{k=1}^{j-1}(n-k+1)\right)+i-j=\frac{(j-1)(2n-j+2)}{2}+i-j。$$

**(2.2) 已知*off*(*i*, *j*)求*i*, *j***

对于$n\times n$阶对称矩阵，顺序存储其下半三角元素时，一个*off*(*i*, *j*)位置对应两个元素$a_{i,j}$和$a_{j,i}$。

设**i≥j**以及行、列下标从0开始，求行号$i$和列号$j$。



## 5.3 数组的压缩存储 续4

**a. 行序为主序时**

分析：考虑**i≥j**，因行序为主序，有

$$off(i,j)=\frac{i(i+1)}{2}+j$$

为表达方便，设$off(i,j)=k$，由0≤$j$≤$i$知

$$\frac{i(i+1)}{2}\le k \quad\Rightarrow\quad i\le\frac{-1+\sqrt{1+8k}}{2}$$

$$\frac{i(i+1)}{2}+i\ge k \quad\Rightarrow\quad i\ge\frac{-3+\sqrt{9+8k}}{2}$$



## 5.3 数组的压缩存储 续5

设 $i_{lb}=\dfrac{-3+\sqrt{9+8k}}{2}$，$i_{ub}=\dfrac{-1+\sqrt{1+8k}}{2}$

即 $i_{lb}\le i\le i_{ub}$。

因为 $0<\sqrt{9+8k}-\sqrt{1+8k}\le 2$，易知 $0\le i_{ub}-i_{lb}<1$，

$i$为非负整数，可知

$$i=\lceil i_{lb}\rceil=\lfloor i_{ub}\rfloor=\left\lfloor\frac{-3+\sqrt{9+8k}}{2}\right\rceil=\left\lfloor\frac{-1+\sqrt{1+8k}}{2}\right\rfloor$$

最后，$j=k-\dfrac{i(i+1)}{2}$。



## 5.3 数组的压缩存储 续6

**b. 列序为主序时**

分析：考虑**i≥j**，因列序为主序，有

$$off(i,j)=\frac{j(2n-j+1)}{2}+i-j$$

为表达方便，设$off(i,j)=k$，由$j$≤$i$≤$n$-1知

$$\frac{j(2n-j+1)}{2}\le k \quad\Rightarrow\quad j\le\frac{2n+1-\sqrt{(2n+1)^2-8k}}{2}$$

$$\frac{j(2n-j+1)}{2}+n-1-j\ge k \Rightarrow$$

$$j\ge\frac{2n-1-\sqrt{(2n-1)^2-8(k-n+1)}}{2}$$



## 5.3 数组的压缩存储 续7

设 $j_{ub}=\dfrac{2n+1-\sqrt{(2n+1)^2-8k}}{2}$，

$$j_{lb}=\frac{2n-1-\sqrt{(2n-1)^2-8(k-n+1)}}{2}，$$

即 $j_{lb}\le j\le j_{ub}$。

可证明，$0\le j_{ub}-j_{lb}<1$，必有 $j=\lceil j_{lb}\rceil=\lfloor j_{ub}\rfloor$。

最后，

$$i=k+j-\frac{j(2n-j+1)}{2}。$$

## 5.3 数组的压缩存储 续8

**例3** 设6×6阶对称矩阵只顺序存储其下半三角(含主对角线)元素，若所有下标从0开始，试求顺序存储下标位置15对应的矩阵元素行、列下标(要求考虑行序为主序以及列序为主序两种情况)。

**解**：设对应矩阵元素行号为$i$，列号为$j$，已知$k$=15。

**a. 行序为主序时**

$$i=\lceil i_{lb}\rceil=\left\lceil\frac{-3+\sqrt{9+8k}}{2}\right\rceil=\left\lceil\frac{-3+\sqrt{9+8\cdot 15}}{2}\right\rceil=\lceil 4.18\rceil=5$$

$$i=\lfloor i_{ub}\rfloor=\left\lfloor\frac{-1+\sqrt{1+8k}}{2}\right\rfloor=\left\lfloor\frac{-1+\sqrt{1+8\cdot 15}}{2}\right\rfloor=\lfloor 5\rfloor=5$$



## 5.3 数组的压缩存储 续9

$$j=k-\frac{i(i+1)}{2}=15-\frac{5(5+1)}{2}=0$$

**a. 列序为主序时**

$$j=\lceil j_{lb}\rceil=\left\lceil\frac{2n-1-\sqrt{(2n-1)^2-8(k-n+1)}}{2}\right\rceil$$

$$=\left\lceil\frac{12-1-\sqrt{(12-1)^2-8(15-6+1)}}{2}\right\rceil=\lceil 2.298\rceil=3$$



## 5.3 数组的压缩存储 续10

$$j=\lfloor j_{ub}\rfloor=\left\lfloor\frac{2n+1-\sqrt{(2n+1)^2-8k}}{2}\right\rfloor$$

$$=\left\lfloor\frac{12+1-\sqrt{(12+1)^2-8\cdot 15}}{2}\right\rfloor=\lfloor 3\rfloor=3$$

$$i=k+j-\frac{j(2n-j+1)}{2}=15+3-\frac{3(12-3+1)}{2}=3$$



## 5.3 数组的压缩存储 续11

$k$=15

$a_{0,0}$
$a_{1,0}$ $a_{1,1}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ ← 列序为主序
$a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$
$a_{5,0}$ $a_{5,1}$ $a_{5,2}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$

↑ 行序为主序



## 5.3 数组的压缩存储 续12

**2. 半三角矩阵**

顺序存储方式与对称矩阵相同。

**3. 带状矩阵(对角矩阵)***

对于$n\times n$阶方阵，若它的全部非0元素落在一个以主对角线为中心的带状区域中，这个带状区域包含主对角线下面以及上面各$b(b\geq 1)$条对角线上的元素以及主对角线本身的元素(这$2b+1$条对角线以外的元素都是0)，该方阵称为**半带宽**为$b$的带状(对角)矩阵。显然，**半带宽**为$b$的带状矩阵的首行(列)以及最后一行(列)的非零元素个数$a=b+1$。



## 5.3 数组的压缩存储 续13

带状矩阵的例子：$n$=8，半带宽$b$=2，$a=b+1=3$

$$\begin{bmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & & & & & \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & & & & \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & & & \\
 & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & & \\
 & & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & \\
 & & & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\
 & & & & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\
 & & & & & a_{7,5} & a_{7,6} & a_{7,7}
\end{bmatrix}$$

## 5.3 数组的压缩存储 续14

**带状矩阵的顺序存储：只分析行序为主序的情况**

记矩阵行、列下标为$i, j$，且下标从0开始，行序为主序时，顺序存储的带状矩阵元素可分为三个部分。设第$i$行需存储的元素个数为$m_i$ ($i$=0, 1, …, $n$-1)，

**第1部分：** **0≤$i$≤$b$-1(共$b$行)，** **0≤$j$≤$i$+$b$**

$m_0=b+1$, $m_1=b+2$, … , $m_{b-1}=2b$，即 $m_i=b+i+1$；

**第2部分：** **$b$≤$i$≤$n$-$b$-1(共$n$-2$b$行)，** **$i$-$b$≤$j$≤$i$+$b$**

$m_i=2b+1$；

**第3部分：** **$n$-$b$≤$i$≤$n$-1(共$b$行)，** **$i$-$b$≤$j$≤$n$-1**

$m_{n-b}=2b$, $m_{n-b+1}=2b-1$, … , $m_{n-1}=b+1$，即 $m_i=b+n-i$；



## 5.3 数组的压缩存储 续15

所以，半带宽为$b$的$n\times n$阶带状矩阵的总元素个数为

$$N=\sum_{i=0}^{n-1}m_i=\sum_{i=0}^{b-1}(b+i+1)+(2b+1)(n-2b)+\sum_{i=n-2b}^{n-1}(b+n-i)$$

$$=(3b+1)b+(2b+1)(n-2b)$$

$$=(3(a-1)+1)(a-1)+(2(a-1)+1)(n-2(a-1))$$

$$=(3a-2)(a-1)+(2a-1)(n-2a+2)$$

以下推导顺序存储下标 $k=off(i, j)$。

**第1部分：** **0≤$i$≤$b$-1(共$b$行)，** **0≤$j$≤$i$+$b$**

$$k=\sum_{p=0}^{i-1}(b+p+1)+j=\frac{i(2b+i)}{2}+j$$



## 5.3 数组的压缩存储 续16

**第2部分：** **$b$≤$i$≤$n$-$b$-1(共$n$-2$b$行)，** **$i$-$b$≤$j$≤$i$+$b$**

$$k=\frac{(3b+1)b}{2}+(2b+1)(i-b)+j-i+b$$

**第3部分：** **$n$-$b$≤$i$≤$n$-1(共$b$行)，** **$i$-$b$≤$j$≤$n$-1**

$$k=\frac{(3b+1)b}{2}+(2b+1)(n-2b)+\sum_{p=n-b}^{i-1}(b+n-p)+j-i+b$$

$$=\frac{(3b+1)b}{2}+(2b+1)(n-2b)+\frac{(i-n+b)(3b+n-i+1)}{2}$$

$$+j-i+b$$



## 5.3 数组的压缩存储 续17

**4. 稀疏矩阵**

当矩阵阶次较高而非零元素较少时，称为稀疏矩阵。对于$m\times n$阶矩阵，记其非零元素数目为$t$，定义稀疏因子$\delta$为

$$\delta=\frac{t}{m\times n}$$

一般地，当**$\delta$≤0.05**时，就认为矩阵是稀疏的。

**(1) 三元组顺序表**

只存储非零元素，每个非零元素用三元组表示为

**($i$, $j$, $a_{i,j}$)**



## 5.3 数组的压缩存储 续18

```
//三元组顺序表的C语言描述
typedef struct {  int i, j;          //行、列下标
                  ElemTp e;          //非零元素值
               } Triple;             //三元组类型
typedef struct { Triple *data;       //三元组表
                  int m, n, t;       //行数、列数、非零元数
               } TSMatrix;           //三元组顺序表
```

**说明：在三元组表data中，要求按<i, j>升序(即行序为主序)排列各三元组。**



## 5.3 数组的压缩存储 续19

**a. 求稀疏矩阵的转置矩阵的算法一**

将矩阵A转置为矩阵B，为保证转置后矩阵B中各三元组按行序为主序顺序存储，则按B矩阵行号i由小到大次序在A中顺序查找所有列号等于i的三元组，将三元组转置操作(行号与列号互换)后顺序存入B中即可。

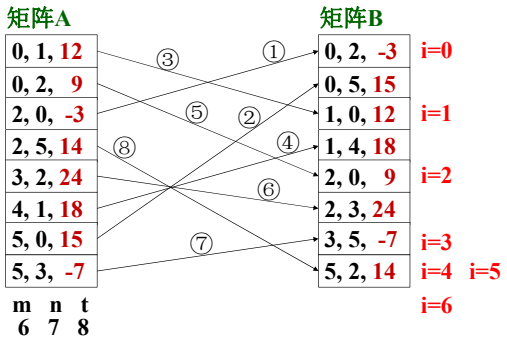
```
void rs1(TSMatrix &A, TSMatrix &B)
{ B.m=A.n; B.n=A.m; B.t=A.t; k=0;
  for(i=0; i<B.m; i++)
    for(p=0; p<A.t; p++)
      if(A.data[p].j==i)
```

## 5.3 数组的压缩存储 续20

```
        { B.data[k].i=i;  //或B.data[k].i=A.data[p].j;
          B.data[k].j=A.data[p].i;
          B.data[k].e=A.data[p].e;
           k++;
        }
}
//算法时间复杂度为O(n×t)
```



## 5.3 数组的压缩存储 续21






## 5.3 数组的压缩存储 续22

**b. 求稀疏矩阵的转置矩阵的算法二**

能不能一遍描述矩阵A的data数组得到转置矩阵B呢？

答：可以，但需要事先知道矩阵B任意行号i之前各行总的元素数目，以及第i行的元素数目。

定义辅助数组pos[0..A.n-1]，其中，pos[i]表示的是转置后矩阵B行号i之前各行元素总数。

关键：怎样求pos[i]值？

转置操作前先扫描一遍矩阵A的data数组，以统计A矩阵各列元素数目，然后通过累加得到pos[i]值。



## 5.3 数组的压缩存储 续23

转置操作过程中，在矩阵B的pos[i]位置存入一个元素后，pos[i]++，指示行号为i的下一个元素存入位置。

```
void rs2(TSMatrix &A, TSMatrix &B)
{ B.m=A.n; B.n=A.m; B.t=A.t;
  pos[i]=new int[B.m];
  //统计A矩阵第i列元素数目存于pos[i]中
  for(p=0; p<B.m; p++) pos[i]=0;
  for(p=0; p<A.t; p++) pos[A.data[p].j]++;
  //累加得到第i列起始位置存于pos[i]
  k=0;
  for(i=0; i<B.m; i++) { p=pos[i]; pos[i]=k; k+=p; }
```



## 5.3 数组的压缩存储 续24

```
  //以下开始转置
  for(p=0; p<A.t; p++)
    { i=A.data[p].j; j=A.data[p].i; k=pos[i];
      B.data[k].i=i; B.data[k].j=j; B.data[k].e=A.data[p].e;
      pos[i]++;
    }
  delete []pos;
}
//算法时间复杂度为O(2n+2t)
//算法辅助空间复杂度为O(n)
```



## 5.3 数组的压缩存储 续25

矩阵A

| | | |
|---|---|---|
| 0, 1, 12 | | |
| 0, 2, 9 | | |
| 2, 0, -3 | | |
| 2, 5, 14 | | |
| 3, 2, 24 | | |
| 4, 1, 18 | | |
| 5, 0, 15 | | |
| 5, 3, -7 | | |
| m | n | t |
| 6 | 7 | 8 |

A矩阵各列元素数目

| pos | 2 | 2 | 2 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

B矩阵第i行起始位置

| pos | 0 | 2 | 4 | 6 | 7 | 7 | 8 |
|---|---|---|---|---|---|---|---|

## 5.3 数组的压缩存储 续26

**c. 稀疏矩阵的加法与乘法***

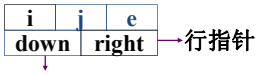
两个稀疏矩阵相加或相乘后，结果矩阵不一定是稀疏的。具体算法本教案不介绍，有兴趣的同学请自学。

**(2) 十字链表**

稀疏矩阵的每个非零元用结点表示，每个结点有水平和垂直两个方向的指针，分别按行号和列号形成水平和垂直链表。



## 5.3 数组的压缩存储 续27




列指针

$m \times n$阶稀疏矩阵有$t$个非零元，则十字链表由以下结点组成：

a. 1个**总头结点(*m*, *n*, x)**，其down指针指向第0行**行头结点**；其right指针指向第0列**列头结点。**

b. $m$个**行头结点(i, -1, x)**，每个行头结点为该行水平循环链表的附加结点，$m$个行头结点组成$m$元数组。



## 5.3 数组的压缩存储 续28

c. $n$个**列头结点(-1, j, x)**，每个列头结点为该列垂直循环链表的附加结点，$n$个列头结点组成$n$元数组。

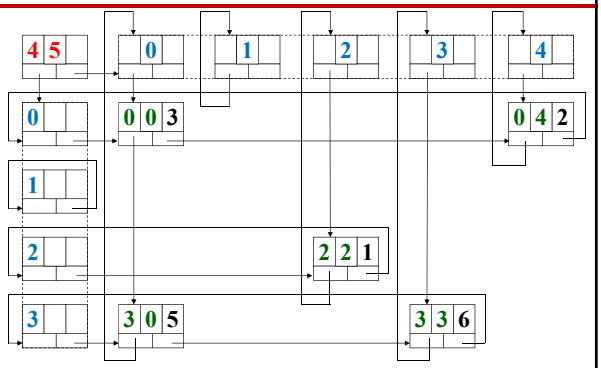
d. $t$个非零**元素结点(i, j, e)**，每个非零元素结点在水平和垂直方向构成循环链表。

例如，

$$M = \begin{bmatrix} 3 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 6 & 0 \end{bmatrix}$$



## 5.3 数组的压缩存储 续29






## 5.3 数组的压缩存储 续30

```
//十字链表结点数据类型的C语言描述
typedef struct node
{ int i, j;
  ElemTp e;
  struct node *right, *down;
} OLNode, *Olink;
```



## 5.3 数组的压缩存储 续31

```
Olink init(int m, int n)  //建空十字链表(不含元素结点)
{ h=new OLNode; h->i=m; h->j=n;  //建总头结点
  //以下建行头结点数组
  row=h->down=new OLNode[m];
  if(!h->down) return NULL;  //返回NULL表示失败
  for(i=0; i<m; i++) { row[i].i=i; row[i].right=&row[i]; }
  //以下建列头结点数组
  col=h->right=new OLNode[n];
  if(!h->right) return NULL;  //返回NULL表示失败
  for(j=0; j<n; j++) { col[j].j=j; col[j].down=&col[j]; }
  return h;
}
```

## 5.3 数组的压缩存储 续33

```
//十字链表中插入结点
Status insert(Olink h, int i, int j, ElemTp e)
//h为总头结点地址
{ m=h->i; n=h->j;
  if(i<0||i>=m||j<0||j>=n) return ERROR;
  //建新元素结点
  s=new OLNode; if(!s) return ERROR;
  s->i=i; s->j=j; s->e=e;
  //以下插入新结点*s到第i行循环链表
  //插入后使第i行循环链表各结点按列下标升序连接
  pr=&h->down[i]; p=pr->right;
```



## 5.3 数组的压缩存储 续34

```
  while(p!=&h->down[i]&&j>p->j) { pr=p; p=p->right; }
  pr->right=s; s->right=p;
  //以下插入新结点*s到第j列循环链表
  //插入后使第j列循环链表各结点按行下标升序连接
  pr=&h->right[j]; p=pr->down;
  while(p!=&h->right[j]&&i>p->i) { pr=p; p=p->down; }
  pr->down=s; s->down=p;
  return OK;
}
```



## 5.3 数组的压缩存储 续35

```
//十字链表中删除结点
Status delete(Olink h, int i, int j)
//h为总头结点地址
{ m=h->i; n=h->j;
  if(i<0||i>=m||j<0||j>=n) return ERROR;
  //以下在第i行链表中查找(i, j)
  pr1=&h->down[i]; pr2=pr1->right;
  while(pr2!=&h->down[i]&&pr2->j!=j)
    { pr1=pr2; pr2=pr2->right; }
  if(pr1==&h->down[i]) return ERROR; //查找失败
  //以下在第j列链表中查找(i, j)
```



## 5.3 数组的压缩存储 续36完

```
  pc1=&h->right[j]; pc2=pc1->down;
  while(pc2!=&h->right[j]&&pc2->i!=i)
    { pc1=pc2; pc2=pc2->down; }
  if(pc2==&h->right[j]) return ERROR; //查找失败
  if(pr2!=pc2) return ERROR; //行、列查找结果不一致
  //以下删除元素结点(i, j)
  pr1->right=pr2->right;
  pc1->down=pc2->down;
  delete pr2; //或delete pc2;
  return OK;  //成功删除
}
```



## 5.4 广义表

**1. 广义表的概念**

广义表也称为**列表**，它是一种递归定义的线性表，记为$LS=(\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{n-1})$，其中，$n$称为列表的长度，列表中每个元素$\alpha_i$可以是单个数据元素(称为**原子**)或一个广义表(称为**子表**)。

一般地，用小写英文字母表示原子，大写英文字母表示子表。

**表头(Head)**：非空的广义表$LS$的第一个元素$\alpha_0$。

**表尾(Tail)**：非空广义表$LS$除表头外其余元素组成的广义表$(\alpha_2, \alpha_3, \dots, \alpha_{n-1})$。



## 5.4 广义表 续1

**广义表举例**:

$A$=()　　//空表

$B$=($e$)　　//$n$=1，Head($B$)=$e$，Tail($B$)=()

$C$=($a$, ($b$, $c$, $d$))　　//$n$=2，Head($C$)=$a$，Tail($C$)=(($b$, $c$, $d$))

$D$=($A$, $B$, $C$)　　//$n$=3，Head($D$)=$A$，Tail($D$)=($B$, $C$)

=((), ($e$), ($a$, ($b$, $c$, $d$)))

$E$=($a$, $E$)　　//$n$=2，Head($E$)=$a$，Tail($E$)=($E$)

=($a$, ($a$, ($a$, …)))

//$E$是一个无限的列表，称为递归表

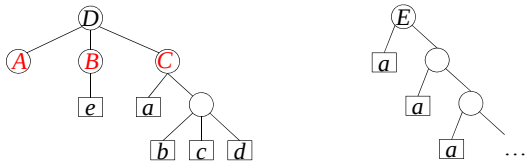
## 5.4 广义表 续2

### 2. 广义表的图形表示

用圆圈表示广义表(或子表)，方框表示原子。

**例如，**

$D$=((), ($e$), ($a$, ($b$, $c$, $d$)))　　$E$=($a$, $E$)

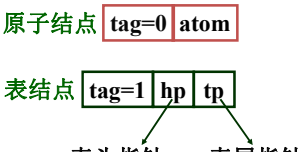





## 5.4 广义表 续3

### 3. 广义表的存储结构

广义表适合采用链式存储结构，链表中有两类结点：原子结点和表结点，如图所示。




tag为标志域，0表示**原子结点**，1表示**表结点**。

为使原子结点和表结点用一种数据类型封装，需要采用C语言的**union类型**。

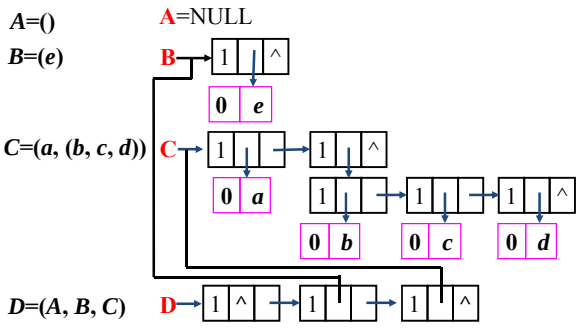


## 5.4 广义表 续4

```
typedef enum { ATOM, LIST } NodeTag;
typedef struct GLNode
{ NodeTag tag;  //tag=0表示原子，tag=1为表
  union { AtomType atom;
          struct { struct GLNode *hp, *tp;
                 } ptr;
        };
} *Glist;
// 若p为Glist类型指针，则p->atom 为原子
// p->ptr.hp为表头指针; p->ptr.tp为表尾指针
```



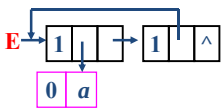
## 5.4 广义表 续5






## 5.4 广义表 续6完

$E$=($a$, $E$)






## 第5章 作业

**习题集**

5.1　5.5　5.7　5.10

**上机编程题**

1. 习题集5.19

2. 编写程序，从字符文件读入$m$, $n$, $t$以及$t$个三元组($i$, $j$, $e$)建立稀疏矩阵的十字链表存储结构。编写算法，实现矩阵转置，输出转置后的三元组到字符文件中，检查你的转置结果是否正确。要求转置时不得新建元素结点(但允许新建行头/列头结点数组以及删除行头/列头结点数组，转置前后，总头结点不允许改变)。

# 第6章 树





## 6.1 树的基本概念

### 1. 树的定义

树是由0个结点或多个结点组成的有限集合$T$，它满足下面两个条件：

(1) 有一个特定的结点，称之为**根结点**；

(2) 其余的结点分成$m$ ($m$≥0)个互不相交的有限集合$T_1$, $T_2$, … , $T_m$。其中，每一个集合都是一棵树，称$T_1$, $T_2$, … , $T_m$为根结点的**子树**。

**提要：** 定义是递归的；

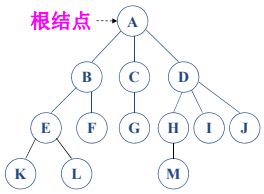
最小的非空树只含有一个根结点；

不含任何结点的树称为**空树**。



## 6.1 树的基本概念 续1

### 2. 树的表示方法




层次图形法
（根在上面，子树在下面）

可视为一个特殊的有向图。根结点用有向边指向每棵子树的根结点。

**特点：**

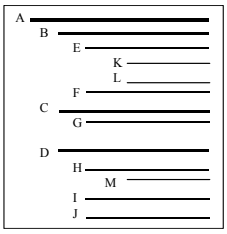
(1) 整棵树的根结点没有有向边指向它(入度为0)；

(2) 其余结点(即每棵子树的根结点)有且仅有一个结点用有向边指向它(入度为1)。

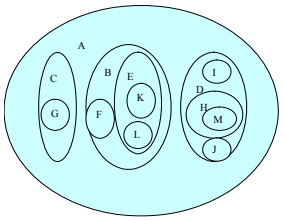


## 6.1 树的基本概念 续2

目录表示法




嵌套集合表示法




广义表表示法

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



## 6.1 树的基本概念 续3

### 3. 树的有关术语

(1) **结点的度**(次数): 结点拥有子树的数目

(2) **叶子**(端结点): 度为零的结点

(3) 分枝结点(内点): 度大于零的结点

(4) **树的度**(次数): 树中各结点度的最大值

(5) 孩子: 树中某个结点的子树的根

(6) 双亲: 结点的直接前驱

(7) 兄弟: 同一双亲的孩子互称兄弟

(8) 祖先: 从根结点到某结点路径上的所有结点称为该结点的祖先

(9) 子孙: 某结点的每个子树中的任一结点称为该结点的子孙



## 6.1 树的基本概念 续4完

(10) 树枝: 树中的路径(只能从上向下)

(11) 树枝长度: 路径中边的数目

(12) **结点层次**: 从根结点到某结点 路径上结点的数目(含该结点本身)

(13) 树的深度: 树中结点的最大层次

(14) 有向树: 结点间的连线是有向的。我们所讲的树都是有向的。

(15) **有序树**: 若树中结点的各子树从左到右是有次序的，称该树为有序树，否则为无序树。

(16) **森林**: 由 $m$ 棵互不相交的树构成的集合 $F=(T_1,T_2, ..., T_m)$称为森林。

一棵树去掉根结点后就表成了森林。

## 6.2 二叉树及其基本性质

**1. 二叉树的定义**

二次有序树称为二叉树。

**(1) 二次**

结点的度最大值是2，即每个结点的子树数目只可能为0、1、 2。

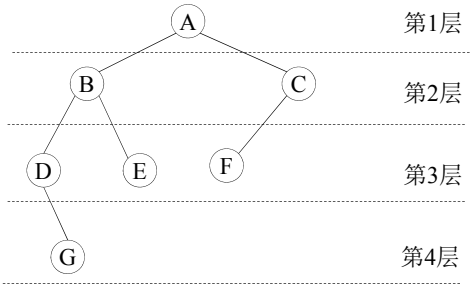
**(2) 有序**

如果结点有两棵子树，则这两棵子树有次序关系，分别称为**左子树**和**右子树**。左、右子树的根结点分别称为其双亲结点的**左儿子**和**右儿子**。度为1的结点有两种情形，即该结点只有左子树或只有右子树。



## 6.2 二叉树及其基本性质 续1

**2. 二叉树的层次号和高度(深度)**






## 6.2 二叉树及其基本性质 续2

**3. 二叉树的性质**

**性质1**：在二叉树的第$i(i\geq 1)$层上，至多有$2^{i-1}$个结点。

证明：用数学规纳法

a. $i$=1时，第1层有且只有一个根结点，则

$2^{i-1}=2^0=1$，性质1成立；

b. 设性质1对第$i$层成立，即第$i$层最多有结点$2^{i-1}$个；

c. 由b.证明性质1对第i+1层成立：

由二叉树的定义，第$i$层每个结点最多有两棵子树，故第$i$+1层结点数最大值$=2\times 2^{i-1}=2^{(i+1)-1}$，即性质1对$i$+1层成立。



## 6.2 二叉树及其基本性质 续3

**性质2**：深度为$k$的二叉树至多有$2^k-1$个结点。

证明：根据性质1，第$i$层最大结点数$=2^{i-1}$，故$k$层最大结点数由下式计算。

$$\sum_{i=1}^{k}2^{i-1}=\frac{1\cdot(1-2^k)}{1-2}=2^k-1$$

**性质3**：对任意一棵二叉树，若其叶子结点数为$n_0$，度为2的结点数为$n_2$，则$n_0=n_2+1$。



## 6.2 二叉树及其基本性质 续4

证明：设二叉树结点总数为$n$, 度为1的结点数为$n_1$, 有

$$n_0+n_1+n_2=n \qquad (1)$$

又设该二叉树的分枝总数为$B$, 除根结点外，其余每个结点入度(结点上方连接的分枝数)为1, 有

$$B+1=n \qquad (2)$$

从每个结点的出度(结点下方连接的分枝数)看，有

$$B=n_1+2n_2 \qquad (3)$$

将(3)代入(2) 得 $n_1+2n_2+1=n$，代入(1)得

$n_0+n_1+n_2=n_1+2n_2+1$, 即$n_0=n_2+1$。



## 6.2 二叉树及其基本性质 续5

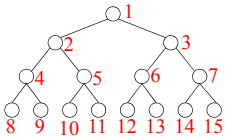
**4. 满二叉树与完全二叉树**

**满二叉树**：一棵深度为$k$且有$2^k-1$个结点的二叉树称为满二叉树。

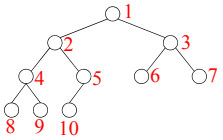
**完全二叉树**：一棵深度为$k$且有$n$个结点的二叉树，当且仅当其每个结点的编号与深度为$k$的满二叉树中编号为1~$n$的结点一一对应时，称为完全二叉树。其中，编号的方法是：从根结点开始，层次从上至少，每层从左向右，从1开始依次为每个结点编号。

**完全二叉树特点**：$k$层完全二叉树前$k$-1层必为$k$-1层满二叉树；其第$k$层上，任意结点的左边不能有空位。

## 6.2 二叉树及其基本性质 续6

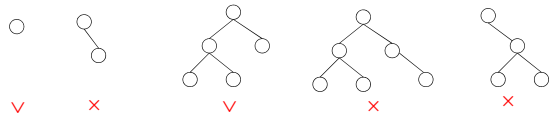



4层满2叉树




10个结点的完全二叉树

**判断以下二叉树是否为完全二叉树？**






## 6.2 二叉树及其基本性质 续7

**5. 完全二叉树的性质**

**性质1**：设$k$层完全二叉树有$n$个结点，则$n$的范围为

$2^{k-1} \le n \le 2^k-1$。

**证明**：前$k$-1层构成满二叉树，共有结点$2^{k-1}-1$个；第$k$层上，结点数范围为$1 \sim 2^{k-1}$，故

$(2^{k-1}-1)+1 \le n \le (2^{k-1}-1)+2^{k-1}$，性质1得证。

**性质2**：$n$个结点的完全二叉树的高度为$\lfloor \log_2 n \rfloor+1$。

**证明**：设树高为$k$, 性质1中不等式取对数，得

$$k-1 \le \log_2 n \le \log_2(2^k-1)$$

有 $k-1 \le \log_2 n < k$,

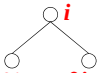
故 $k-1=\lfloor \log_2 n \rfloor$, 性质2得证。



## 6.2 二叉树及其基本性质 续8

**性质3**：$n$个结点的完全二叉树中，编号为$i$ ($1 \le i \le n$)的结点的左、右儿子若存在，则它们的编号分别为$2i$和$2i+1$。

**推论**：完全二叉树中，$i$号结点的双亲编号为$\lfloor i/2 \rfloor$。

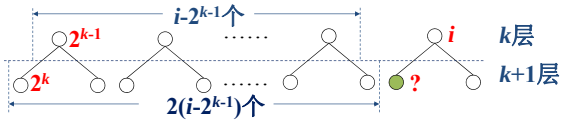



**证明**：设$i$号结点的层号为$k$，因第$k$层最左边结点的编号为$2^{k-1}$，所以在第$k$层上，$i$号结点的左边共有$i-2^{k-1}$个结点。

若$i$号结点的左儿子存在，则第$k$层上$i$号结点左边的所有结点在第$k$+1层上共有儿子$2(i-2^{k-1})$个，它们是$i$号结点的左儿子(在第$k$+1层)左边的全部结点数。又第$k$+1层最左边结点的编号为$2^k$，故$i$号结点左儿子的编号应为$2^k+2(i-2^{k-1})=2i$，性质3因此得证。**证明原理如下图所示**。



## 6.2 二叉树及其基本性质 续9




**性质4**：$n$个结点的完全二叉树中，设度为1的结点数为$n_1$。当$n$为奇数时，$n_1=0$；当$n$为偶数时，$n_1=1$。

**证明**：令$i=\lfloor n/2 \rfloor$, 当$n$为奇数时，有$2i=n-1$, $2i+1=n$, 由性质3, 知$1 \sim i$号结点均有左、右儿子，而$i+1 \sim n$号结点无儿子，即$n_1=0$; 当$n$为偶数时，有$2i=n$, 即$i$号结点只有左儿子，$1 \sim i-1$号结点均有两个儿子，$i+1 \sim n$号结点无儿子，即$n_1=1$(此时仅$i$号结点度为1)。



## 6.2 二叉树及其基本性质 续10

**习题举例：**

1. 拥有1000个结点的完全二叉树共有____层；其叶子结点的数目等于____。

**解**：令$n$=1000, 有

$$\lfloor \log_2 n \rfloor+1=\lfloor \log_2 1000 \rfloor+1=10 \quad (\text{性质2})$$

设完全二叉树度为0, 1, 2的结点数分别为$n_0$, $n_1$, $n_2$, 则

$n_1=1$, (**性质4, $n$为偶数**)

$n_0=n_2+1$, (**二叉树, 性质3**)

$n_0+n_1+n_2=n$, 即 $n_0+1+n_0-1=n$, 有$n_0=n/2=1000/2=500$



## 6.2 二叉树及其基本性质 续11完

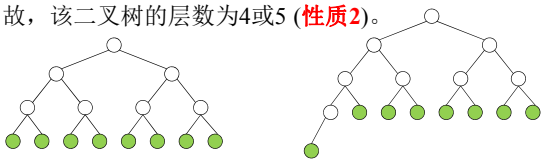
2. 拥有8个终端结点的完全二叉树共有____层。

**解**：设完全二叉树度为0, 1, 2的结点数分别为$n_0$, $n_1$, $n_2$, 结点总数为$n$, 则$n_1=0$或1 (**性质4**)

$n_0=n_2+1=8$, (**二叉树, 性质3**) 有$n_2=7$

所以, $n=n_0+n_1+n_2$, 即 $n=8+n_1+7=15$或16,

故，该二叉树的层数为4或5 (**性质2**)。



## 6.3 二叉树的存储结构与算法

### 1. 二叉树的顺序存储结构

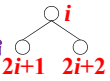
**(1) 完全二叉树**

适合用一维数组按结点编号顺序存储每个结点。

**例**：$n$个结点的完全二叉树若用C语言一维数组元素$a_0$, $a_1$, …, $a_{n-1}$顺序存储编号为1~$n$的每个结点，则结点$a_i$的左、右儿子的下标分别是多少？$a_i$双亲结点的下标是多少？

答：$a_i$的左、右儿子和双亲的下标分别为$2i$+1、$2i$+2和⌊($i$-1)/2⌋。

下标从1开始 


下标从0开始 




## 6.3 二叉树的存储结构与算法 续1

**(2) 非完全二叉树**

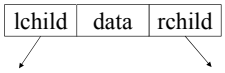
变成完全二叉树存储，无结点处用空元素占位。

**(3) 遍历顺序表**

二叉树的前序和中序遍历序列可以唯一确定一棵二叉树。(详见二叉树的遍历)

### 2. 二叉树的链式存储

**(1) 二叉链表**




```
typedef struct node
{ ElemTp data; //数据域
  struct node *lchild,  //左指针
              *rchild;  //右指针
}*BiT, BiTNode;
```

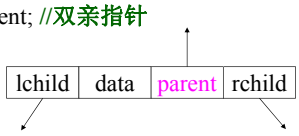


## 6.3 二叉树的存储结构与算法 续2

**(2) 三叉链表**

在二叉链表的基础上结点增加一个双亲指针。

```
typedef struct node
{ ElemTp data; //数据域
  struct node *lchild,  //左指针
              *rchild,  //右指针
              *parent; //双亲指针
}*TrT, TrTNode;
```






## 6.3 二叉树的存储结构与算法 续3

### 3. 二叉树的遍历

指按某种次序访问二叉树每个结点一次且仅一次。

**(1) 前序(先根)遍历**

若二叉树非空，则

i 访问根结点；

ii 先序(先根)遍历左子树；

iii 先序(先根)遍历右子树。

**(2) 中序(中根)遍历**

若二叉树非空，则

i 中序(中根)遍历左子树；

ii 访问根结点；

iii 中序(中根)遍历右子树。



## 6.3 二叉树的存储结构与算法 续4

**(3) 后序(后根)遍历**

若二叉树非空，则

i 后序(后根)遍历左子树；

ii 后序(后根)遍历右子树；

iii 访问根结点。

```
void preorder(BiT bt)
{ if(bt)
  { visite(bt);
    preorder(bt->lchild);
    preorder(bt->rchild);
  }
}
```

```
void midorder(BiT bt)
{ if(bt)
  { midorder(bt->lchild);
    visite(bt);
    midorder(bt->rchild);
  }
}
```



## 6.3 二叉树的存储结构与算法 续5

```
void lasorder(BiT bt)
{ if(bt)
  { lasorder(bt->lchild);
    lasorder(bt->rchild);
    visite(bt);
  }
}
```

**(4) 二叉树按层次遍历**

即结点按层次从下至下，每层从左向右依次遍历每个结点。**算法提要: 须使用队列**。

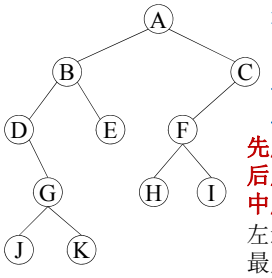
## 6.3 二叉树的存储结构与算法 续6

```
void layertravel(BiT bt) //二叉树按层次遍历
{ if(bt==NULL) return; //空树直接结束
  initQueue(Q);  //初始化结点指针队列
  enQueue(Q, bt);  //根结点指针入队
  while(!empty(Q))
  { bt=delQueue(Q);  //队头元素出队
    visite(bt);
     if(bt->lchild)  enQueue(Q, bt->lchild);
     if(bt->rchild)  enQueue(Q, bt->rchild);
  }
}
```



## 6.3 二叉树的存储结构与算法 续7

**手工遍历举例：**




先序次序：ABDGJKECFHI
中序次序：DJGKBEAHFIC
后序次序：JKGDEBHIFCA
层次遍历：ABCDEFGHIJK

**先序特点**: 根结点总在最开始；
**后序特点**: 根结点总在最后；
**中序特点**: 第一访问结点为最左边第一个无左子树的结点；最后访问结点为最右边第一个无右子树的结点。



## 6.3 二叉树的存储结构与算法 续8

**(5) 用先序遍历算法建立二叉树(二叉链表)存储结构**

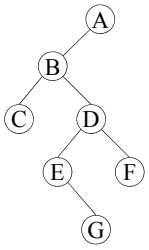
先序遍历递归算法可用于建立二叉树(二叉链表)。

```
BiT crtBT()     //算法假定二叉链表data域为char型
{ ch=getchar();
  if(ch=='#')  //输入#字符表示NULL指针域
     return NULL;
  bt=new BiTNode();  //建立根结点, new为C++操作符
  bt->data=ch;
  bt->lchild=crtBT();  //建立左子树
  bt->rchild=crtBT();  //建立右子树
  return bt;                //返回根结点指针
}
```



## 6.3 二叉树的存储结构与算法 续9

**算法crtBT()的输入方法举例：**




创建左边的二叉树，所需的输入字符序列为：
**ABC##DE#G##F###**



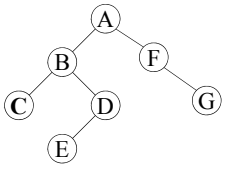
## 6.3 二叉树的存储结构与算法 续10

**(6) 用先序和中序遍历结点访问序列建立二叉树**

已知先序和中序遍历序列，则二叉树唯一确定。

**习题举例**：某二叉树结点的先序和中序访问次序如下，试画出该二叉树。

先序：ABCDEFG
中序：CBEDAFG




**方法提要**：
**a.** 先由先序序列求得根结点；
**b.** 再由根结点在中序序列中的位置，知：它之前访问的结点为其左子树结点，它之后访问的为其右子树结点；
**c.** 递归应用a. b. 两条。



## 6.3 二叉树的存储结构与算法 续11

```
//算法：用先序和中序遍历序列建立二叉树
BiT crtBT1(char *pr, int &i, char *mi, int u, int v)
{ BiT bt;int k;
  if(u>v) return NULL;
  bt=new BiTNode();     //pr[i]为子树根结点
  bt->data=pr[i++];      //mi[u..v]为子树中序序列
  for(k=u;k<=v;k++) if(mi[k]==bt->data) break;
  bt->lchild=crtBT1(pr,i,mi,u,k-1);
  bt->rchild=crtBT1(pr,i,mi,k+1,v);
  return bt;
}
```

调用形式: i=0; u=0; v=mi串长-1；
bt=crtBT1(pr, i, mi, u, v);

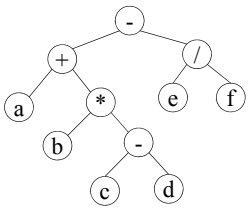
## 6.3 二叉树的存储结构与算法 续12

**(7) 二叉树先序、后序遍历求表达式的前缀、后缀式**

**例：** 表达式 a+b*(c-d)-e/f 中缀式

- + a * b – c d / e f 前缀式(**波兰式**)

a b c d - * + e f / - 后缀式(**逆波兰式**)




前缀式对应**先序遍历**次序；
后缀式对应**后序遍历**次序。



## 6.3 二叉树的存储结构与算法 续13

**4. 二叉树算法举例**

**例1**：若二叉树采用二叉链表存储结构，已知根结点指针bt，写一个算法，求二叉树的高度。

```
int height(BiT bt)
{ if(!bt) return 0;            //空树的高度为0
  hl=height(bt->lchild);   //hl为左子树高度
  hr=height(bt->rchild);   //hr为右子树高度
  return max(hl, hr)+1;
}
```



## 6.3 二叉树的存储结构与算法 续14

**例2**：若二叉树采用二叉链表存储结构，已知根结点指针bt，写一个算法，求二叉树中度为1的结点数目。

```
int f(BiT bt)
{ if(!bt) return 0;
  nl=f(bt->lchild);   //nl为左子树中度为1的结点数目
  nr=f(bt->rchild);   //nr为右子树中度为1的结点数目
  return nl+nr+ \
  (bt->lchild&&!bt->rchild||!bt->lchild&&bt->rchild);
}
```

**类似问题**：求度为2的结点数; 求叶子数; 求总结点数。



## 6.3 二叉树的存储结构与算法 续15

**例3**：若二叉树采用二叉链表存储结构，已知根结点指针bt及某结点指针p，求结点*p所在层号。

```
int  LayerNo(BiT bt, BiT p)  //利用二叉树层次遍历算法
{ if(bt==NULL) return 0;  //空树层号为0
  initQueue(Q);  //初始化结点指针队列
  enQueue(Q, (bt, 1));   //根结点指针及层号1入队
  while(!empty(Q))
  { (bt, k)=delQueue(Q);  //队头元素出队
    if(bt==p) return k;
    if(bt->lchild)  enQueue(Q, (bt->lchild, k+1));
    if(bt->rchild)  enQueue(Q, (bt->rchild, k+1));
  } return 0;  }
```



## 6.3 二叉树的存储结构与算法 续16

**例4**：若二叉树采用二叉链表存储结构，已知根结点指针bt及某结点指针p，求*p的双亲结点指针。

**算法设计**：
采用先序递归遍历；
访问根结点用测试根结点是否为*p的双亲代替；
测试到根结点是*p的双亲时，则提前结束遍历。

**难点：双亲一旦求得，怎样提前结束遍历呢？**
有两种办法：
算法定义为有值函数(返回值为求得的双亲指针)；
算法定义为无值函数(用引用型形参返回双亲指针)。

**类似问题**：已知数据域值，在二叉树中查找结点。



## 6.4 二叉树的存储结构与算法 续17

```
//有值函数求解
BiT getParent(BiT bt, BiT p)
{ if(!bt) return NULL;
  if(bt->lchild==p||bt->rchild==p) return bt;
  parent=getParent(bt->lchild, p);
  if(parent) return parent;
  return getParent(bt->rchild);
}
```

**//parent是否为NULL可以判断结果是否已经求得**

## 6.3 二叉树的存储结构与算法 续18

```
//无值函数求解
void getParent(BiT bt, BiT p, BiT &parent)
{ if(!bt||parent) return;
  //parent!=NULL表示结果已求得
  if(bt->lchild==p||bt->rchild==p) parent=bt;
  getParent(bt->lchild, p, parent);
  getParent(bt->rchild, p, parent);
}
//调用形式：
BiT parent=NULL;    //必须先将parent赋值为NULL
getParent(bt, p, parent);
```



## 6.3 二叉树的存储结构与算法 续19

**例5**：若二叉树采用二叉链表存储结构，已知根结点指针bt，求最左边的第一个叶子结点的地址。

**算法1**：用先序遍历递归算法，先序遍历访问次序中的第1个访问的叶子即为所求。

```
BiT  f(BiT bt)
{ if(!bt) return NULL;  //空树没有叶子，返回NULL
  if(!bt->lchild&&!bt->rchild) return bt;
  p=f(bt->lchild);   //在左子树中求最左边的叶子
  if(p) return p;    //若求取成功，则返回p
  return f(bt->rchild); //求取不成功，在右子树中求
}
```



## 6.3 二叉树的存储结构与算法 续20

**算法2**：用后序遍历递归算法，后序遍历的第1访问结点为最左边的叶子结点。

```
BiT  f(BiT bt)
{ if(!bt) return NULL;  //空树没有叶子，返回NULL
  p=f(bt->lchild);
  if(p) return p;
  p=f(bt->rchild);
  if(p) return p;
  return bt;
}
```

```
void f(BiT bt, BiT &p)
{ if(!bt) return;
  f(bt->lchild, p);
  if(!p) f(bt->rchild, p);
  if(!p) p=bt;
}
//调用时，p的初值应为NULL
```



## 6.3 二叉树的存储结构与算法 续21

**例6**：已知二叉树(二叉链表)根结点指针bt，求中序遍历序列中第$k$ ($k$=1, 2, 3, …)个访问结点的指针。

```
//有值函数求解
BiT f(BiT bt, int &k)
{ if(!bt) return NULL;
  p=f(bt->lchild, k);
  if(k>0)  //或if(!p)
    { k--; if(k==0) return bt;
      p=f(bt->rchild, k);
    }
  return p;
}
```

调用：

```
int k=…;
BiT p=f(bt, k);
```

调用结束后，若k==0, 说明求取成功, p为第k个访问结点指针；k>0, 说明结点总数<k, p为NULL。



## 6.3 二叉树的存储结构与算法 续22

```
//无值函数求解
void f(BiT bt, int &k, BiT &p)
{ if(!bt||!k) return;
  f(bt->lchild, k, p);
  if(k>0)
    { k--;
      if(k==0) { p=bt; return; }
      f(bt->rchild, k, p);
    }
}
```

调用：

```
int k=…; BiT p;
BiT ptr=f(bt, k, p);
```

调用结束后，若k==0, 说明求取成功, p为第k个访问结点指针；k>0, 说明结点总数<k, p的值无意义。



## 6.3 二叉树的存储结构与算法 续23

**例7**：已知二叉树(二叉链表)根结点指针bt以及某结点指针p，求结点*p在中序遍历中的访问次序。

```
//无值函数求解
void f(BiT bt, BiT p, int &k)
{ if(!bt||k>0) return;
  f(bt->lchild, p, k);
  if(k>0) return;
  k--; if(bt==p) k=-k;
  f(bt->rchild, p, k);
}
```

调用：

```
int k=0; f(bt, p, k);
```

若k<0, 说明*p不在二叉树中，这时-k为二叉树结点总数；若k>0, 则k为*p的中序遍历访问次序。

**算法技巧**：k值为负表示未完成，变正表示完成。

## 6.3 二叉树的存储结构与算法 续24

**例8**：使用堆栈实现前、中、后序遍历的非递归算法。

//前序遍历

```
void pretravel(BiT bt)
{ initStack(S);
  while(bt||!empty(S))
     if(bt) { visite(bt); push(S, bt); bt=bt->lchild; }
       else { bt=pop(S); bt=bt->rchild; }
}
```



## 6.3 二叉树的存储结构与算法 续25

//中序遍历

```
void midtravel(BiT bt)
{ initStack(S);
  while(bt||!empty(S))
    if(bt) { push(S, bt); bt=bt->lchild; }
      else { bt=pop(S); visite(bt); bt=bt->rchild; }
}
```



## 6.3 二叉树的存储结构与算法 续26

//后序遍历

**算法提要**：需在堆栈元素或二叉树结点中增加标志域mark: mark=0, 1分别表示从左、右子树返回并退栈;

```
void lastravel(BiT bt)
{ initStack(S);
  while(bt||!empty(S))
    if(bt) { mark=0; push(S, (bt, mark)); bt=bt->lchild; }
      else { (bt, mark)=pop(S);
             if(mark==0) { mark=1; push(S, (bt, mark));
                           bt=bt->rchild; }
               else { visite(bt); bt=NULL; }
           } }
```



## 6.3 二叉树的存储结构与算法 续27

**例9**：使用三叉链表实现前、中、后序非递归遍历。

已知三叉链表根结点地址bt，结点parent域已正确填充(根结点的parent域为NULL)。

//先序遍历

```
void preorder(TrT bt)
{ while(bt)
  { visite(bt);
    if(bt->lchild) bt=bt->lchild;
     else if(bt->rchild) bt=bt->rchild;
            else ①    //指针通过parent域回溯
  }
}
```



## 6.3 二叉树的存储结构与算法 续28

**bt指针怎样回溯？**

答：关键是回溯的停止条件：

(1) bt回溯到NULL；

(2) bt不是NULL，结点*bt满足条件：bt指针是从*bt的左儿子回溯上去的，并且*bt的右指针不是NULL。

若, bt是从*bt的右儿子回溯上去的，则bt需继续回溯。

```
while(1)    // ① 的具体实现
{ p=bt; bt=bt->parent; //bt回溯一次
  if(!bt) break;
  if(bt->lchild==p&&bt->rchild) { bt=bt->rchild; break; }
}
```



## 6.3 二叉树的存储结构与算法 续29

//中序遍历

```
void midorder(TrT bt)
{ mark=0;   //mark==0表示左子树未遍历
  while(bt)
    if(mark==0)
       if(bt->lchild) bt=bt->lchild;
         else mark=1;
    else
      { visite(bt);
        if(bt->rchild) { bt=bt->rchild; mark=0; }
          else ②    //bt指针回溯
      } }
```

## 6.3 二叉树的存储结构与算法 续30

**bt指针怎样回溯？**

答：回溯的停止条件：

(1) bt回溯到NULL；

(2) bt不是NULL，结点*bt满足条件：bt指针是从*bt的左儿子回溯上去的。

```
// ② bt指针回溯
while(1)
{ p=bt; bt=bt->parent; //bt指针回溯1次
  if(!bt) break;
  if(bt->lchild==p) { mark=1; break; }
}                          //标记mark为1表示左子树已遍历
```



## 6.3 二叉树的存储结构与算法 续31

```
//后序遍历
void lasorder(TrT bt)
{ mark=0;
  while(bt)
    switch(mark)
    { case 0: //左子树未遍历
        if(bt->lchild) bt=bt->lchild; else mark=1;
        break;
      case 1: //左子树已遍历，右子树未遍历
        if(bt->rchild) { bt=bt->rchild; mark=0; }
          else mark=2;
        break;
```



## 6.3 二叉树的存储结构与算法 续32

```
      case 2: //左、右子树均已遍历
        visite(bt);
        p=bt; bt=bt->parent;
        if(bt)
          if(bt->lchild==p) mark=1;
            else mark=2;
        break;
    } // switch结束
} //算法结束
```



## 6.3 二叉树的存储结构与算法 续33

**例10** 判断二叉树（二叉链表）是否为完全二叉树。

```
bool check(BiT bt, int i, int n)
{ if(i>n) return true;
  if(!bt) return false;
  return check(bt->lchild, 2*i, n)&& \
         check(bt->rchild, 2*i+1, n);
}
```

**调用**：n=结点总数；

```
if(check(bt, 1, n)) …
```

**注意**：算法需先求得二叉树结点总数

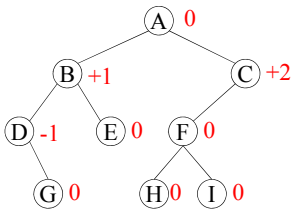


## 6.3 二叉树的存储结构与算法 续34

**5. 平衡二叉树**

**结点的平衡因子**：指该结点左子树和右子树的高度差。

**平衡二叉树**：各结点平衡因子的绝对值最大为1。即结点的平衡因子只有-1, 0, +1三种情形。






## 6.3 二叉树的存储结构与算法 续35

**例1** 为二叉链表结点增加一个整数域**bf**用于存放该结点的平衡因子。编写算法，填充各结点的平衡因子。

```
int f(BiT bt)
{ if(!bt) return 0;      //空树，高度为0
  hl=f(bt->lchild);    //hl为左子树高度
  hr=f(bt->rchild);    //hr为右子树高度
  bt->bf=hl-hr;        //bf为平衡因子
  return max(hl, hr)+1;
}
```

**算法副作用**：返回值为树高。

## 6.3 二叉树的存储结构与算法 续36完

**例2** 判断二叉树(二叉链表)是否是平衡的。

```
int tst(BiT bt, bool &isBalenced)
{ if(!bt||!isBalenced) return 0;
  hl=tst(bt->lchild,isBalenced);
  if(isBalenced)
   { hr=tst(bt->rchild, isBalenced);
     if(isBalenced) { isbalenced=abs(hl-hr)<=1;
                      return max(hl, hr)+1;      }
   }
  return 0;
}
```

**调用**：bool isBalenced=true; tst(bt, isBalenced);



## 6.4 穿线(线索)二叉树

### 1. 穿线(线索)二叉树的概念

为方便求取前/中/后序遍历次序中某结点的前趋和后继结点，利用二叉树中为NULL的左、右指针，让它们分别指向前趋后继，称为**线索**。此外，在结点中增加左、右两个标志域ltag和rtag，以指明lchild和rchild指向左、右儿子还是指向前趋、后继。

根据遍历次序，可分别生成**前序、中序、后序穿线（线索）二叉树**。

**性质**：若二叉链表二叉树中共有$n$个结点，则NULL指针域共有$n+1$个。

**证明**：$n$个结点共有$2n$个指针域，其中有$n-1$个结点被指到，故余下$n+1$个NULL指针域。

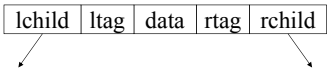


## 6.4 穿线(线索)二叉树 续1

### 2. 穿线(线索)二叉树的存储结构

```
enum PT { LINK, THREAD } ; //定义指针域类型
typedef  struct node
{ ElemTp data;
  struct node *lchild, *rchild;
  enum PT ltag, rtag;
} *SBiT, SBiT_Node;
```

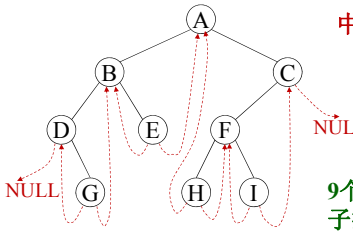
**结点结构示意图**






## 6.4 穿线(线索)二叉树 续2

### 3. 穿线(线索)二叉树的画法




**中序线索二叉树的例子**

中序遍历次序：DGBEAHFIC

**9个结点; 8个左、右儿子指针; 10个线索指针。**



## 6.4 穿线(线索)二叉树 续3

### 4. 中序穿线(线索)二叉树的算法

**(1) 中序遍历算法**

```
//求中序遍历第1访问结点
SBiT first(SBiT bt)
{  while(bt&&bt->ltag==LINK) bt=bt->lchild;
   return bt;
}
//求中序遍历某结点的后继
SBiT next(SBiT p)
{ if(p->rtag==THREAD) return p->rchild;  //线索后继
  return first(p->rchild); //右子树第1访问结点为后继
}
```



## 6.4 穿线(线索)二叉树 续4

```
//中序遍历
void midtravel(SBiT root)
{ p=first(root);
  while(p) { visite(p); p=next(p); }
}
```

**(2) 中序线索化**

将每个结点的NULL指针域、ltag和rtag正确填充称为线索化。

**算法提要**：采用中序遍历；
bt为主扫描指针，pr跟踪*bt的前驱结点；
对*pr的右指针和*bt的左指针进行操作。

## 6.4 穿线(线索)二叉树 续5

```
//中序线索化
void mit(SBiT bt, SBiT &pr)
{ if(bt)
  { mit(bt->lchild, pr);
    if(pr) if(pr->rchild) pr->rtag=LINK;
              else { pr->rtag=THREAD; pr->rchild=bt; }
    if(bt->lchild) bt->ltag=LINK;
      else { bt->ltag=THREAD; bt->lchild=pr; }
    pr=bt;
    mit(bt->rchild, pr);
  }
}
```



## 6.4 穿线(线索)二叉树 续6完

```
//调用
BiT pr=NULL;
mit(bt, pr);
pr->rtag=THREAD;
```

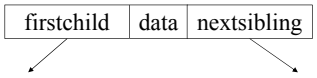
**二叉树遍历算法空间复杂度小结：**
不论递归还是非递归算法，$S(n)=O(n)$, $n$为结点总数
递归：堆栈的最大深度为$n$;
非递归：或设堆栈，或增加parent指针，或增加线索标志域，空间复杂度增加$n$或$2n$。



## 6.5 二叉树与森林的转换 续1

### 1. 孩子兄弟链表

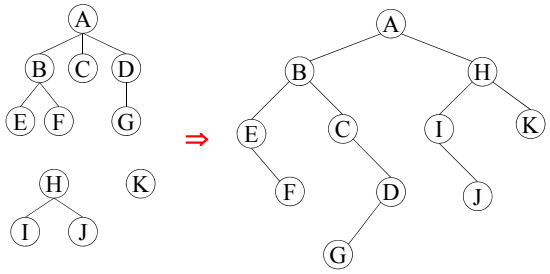
森林可用孩子兄弟链表实现存储。孩子兄弟链表本质为二叉链表，其左指针指向子树根结点的第1个孩子(**first child**)，右指针指向根结点的下一个兄弟(**next sibling**)。






## 6.5 二叉树与森林的转换 续1

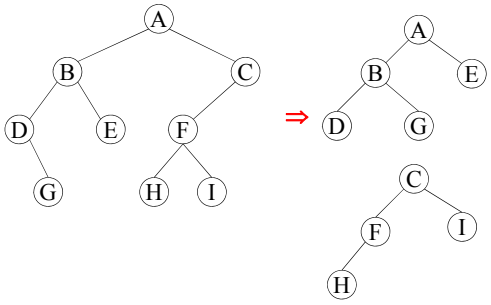
### 2. 森林转换为二叉树(孩子兄弟链表)






## 6.5 二叉树与森林的转换 续2完

### 3. 二叉树(孩子兄弟链表)转换为森林






## 6.6 Huffman二叉树

### 1. Huffman二叉树的定义

设有$n$个**非负**权值组成的集合$\{w_1, w_2, \dots, w_n\}$，Huffman二叉树是满足如下条件的二叉树：

(1) 该二叉树有 $n$ 个叶子结点(记每个叶子权值为 $w_i$)；

(2) 如下定义的带权路径长度(Weighted Path Length, WPL)最小。

$$\mathrm{WPL}=\sum_{i=1}^{n} w_i L_i$$

其中，$L_i$为根结点至第$i$个叶子结点的路径长度。

Huffman二叉树也称为**最优二叉树**。该定义可以推广至多叉树，称为$m$-叉Huffman树($m\geq 2$)。

## 6.6 Huffman二叉树 续1

Huffman二叉树中，若权值$w_i$定义为概率，且满足概率和为1，即

$$\sum_{i=1}^{n} w_i = 1 \quad (w_i \geq 0)$$

则WPL表示的是根结点至$n$个叶子结点路径长度的数学期望。当权值$w_i$不满足上式时，可以规一化为满足上式，即令

$$p_i = \frac{w_i}{\sum_{i=1}^{n} w_i} \quad (w_i \geq 0)$$

因此，**最优(WPL最小)**的含义等价为根结点至$n$个叶子结点的路径长度的**数学期望最小**。



## 6.6 Huffman二叉树 续2

**2. Huffman二叉树的构造方法**

(1) 根据给定的$n$个非负权值 $\{w_1, w_2, \dots, w_n\}$，生成 $n$ 棵二叉树的集合$F=\{T_1, T_2, \dots, T_n\}$，其中每棵二叉树$T_i$只有一个带权为$w_i$的根结点，左右子树为空；

(2) 在 $F$ 中选择两棵根结点值最小的树 $T_i$ 和$T_j$ 作为左右子树并增加一个根结点构成一棵新二叉树$T_k$，$T_k$ 根结点权值为$T_i$和$T_j$根结点权值之和；

(3) 在 $F$ 中删除$T_i$和$T_j$，并把 $T_k$ 加到 $F$中；

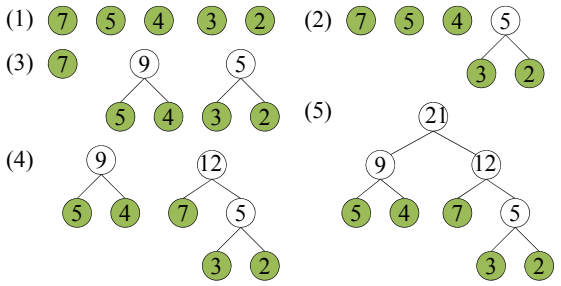
(4) 重复 (2) (3)， 直到 $F$中只含一棵树。



## 6.6 Huffman二叉树 续3

**Huffman二叉树构造过程举例：**

设权值集合={ 7, 5, 4, 3, 2 }，则$F$集合构造过程如下。






## 6.6 Huffman二叉树 续4

**3. Huffman二叉树的特点**

(1) 当叶子结点数$n$>1时，Huffman二叉树没有度为1的结点(其度为2的结点数=$n$-1，结点总数=2$n$-1)；

(2) 可以证明，上述构造方法得到的二叉树一定是Huffman二叉树，获得该最小WPL的Huffman二叉树形不一定惟一；

(3) 若采用归一化权重，且叶子上的权重表示某事件出现的概率，则出现概率越小的叶子距根结点的路径越长，反之越短。



## 6.6 Huffman二叉树 续5

**4. Huffman编码**

**(1) Huffman编码的概念**

设通信或存储系统中使用了$n$个符号$\{ s_1, s_2, \dots, s_n\}$来表示消息，各符号出现的概率分别为$\{ p_1, p_2, \dots, p_n\}$，满足

$$\sum_{i=1}^{n} p_i = 1 \quad (p_i > 0)$$

若$T$表示由这$n$个符号为叶子结点(符号出现的概率为叶子的权重)构造出的Huffman二叉树，$T$中所有左分支和右分支分别标记1, 0（或0, 1），则由根结点到叶子路径上的标记生成每个符号的二进制编码(码长即为路径长度)，称为Huffman编码。



## 6.6 Huffman二叉树 续6

**(2) Huffman编码的特点**

**a. 编码为不等长码**

出现概率小的符号编码长，概率大的符号编码短。

**b. WPL最小对应平均码长最短**

**c. Huffman编码是前缀编码**

前缀编码是指任一编码不以另一编码为前缀。前缀编码可以确保通信系统收信端收到的二进制比特流**单义可译**。

**例如**: 符号集合为{ A, B, C}，对应编码为{ 1, 11, 01}收到11101，可译为ABC或BAC(不是单义可译)，因为编码11以另一编码1为前缀；若对应编码为{ 1, 00, 01}，则11101只能惟一地译为AAAC。

## 6.6 Huffman二叉树 续7

**d. Huffman编码可实现数据压缩**

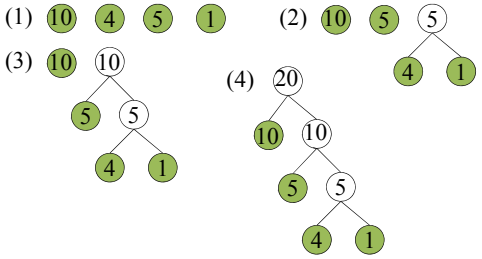
由$n$个符号构造出的Huffman二叉树，若其WPL<$\log_2 n$， 则可实现无损数据压缩。一般地，只要各符号出现概率不同，用Huffman编码均可实现数据压缩。特别地，当各符号出现概率相等时，Huffman编码与等长码等价。

**例**：已知某电文由4个符号A, B, C, D组成，一段20个符号组成的电文为CAABBAACACCABABACDAA。试用Huffman编码对这段电文进行编码，计算平均编码长度并与等长码对比。



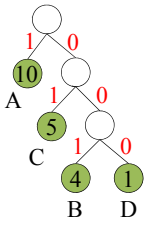
## 6.6 Huffman二叉树 续8

**解**：显然，在这段20个符号组成的电文中，A, B, C, D出现的次数分别10, 4, 5, 1(对应概率分别为1/2, 1/5, 1/4, 1/20)。






## 6.6 Huffman二叉树 续9




各符号编码为：

A:1

B: 001

C: 01

D: 000

等长码发送此20个符号组成的电文共需40比特；Huffman编码则只需要1.75×20=35比特。

$$L_{av}=\frac{1}{20}(10\times1+5\times2+4\times3+1\times3)=\frac{35}{20}=1.75 \text{ bit}$$

4个符号的等长码每个符号编码长度=$\log_2 4$=2 bit



## 6.6 Huffman二叉树 续10

**5. Huffman编码的生成算法**

设符号数为$n$($n$≥2)，则二元Huffman码树共有**2$n$-1**个结点。Huffman码树比较适合用数组以及静态链表方式实现存储。

```
typedef  struct { int parent,      //双亲下标
                      lchild,      //左儿子下标
                      rchild;      //右儿子下标
                  double w;        //结点权重
                } HF_BTNode;   //码树结点类型
```



## 6.6 Huffman二叉树 续11

```
#define N  …                    //最大允许符号数
typedef struct
{ int n;                        //实际符号数, n<=N
  ElemTp s[N];                  //符号表
  double weight[N];             //符号权重表
  char code[N][N+1];            //编码表
  HF_BTNode hf[2*N-1];          //码树
} HFT;                          //Huffman码树及码表
```

**算法功能**：

已知n, s[0..n-1], weight[0..n-1]，求hf和code。



## 6.6 Huffman二叉树 续12

**算法设计**：

(1) hf[0..n-1]为码树中的n个叶子结点，hf[i]对应符号s[i]，s[i]的编码用字符串code[i]表示；

(2) hf[n..2n-2]为n-1个度为2的结点，构造Huffman树的过程中共需要进行n-1趟两个子树的合并，每趟合并后，生成新的子树的根结点(2度结点)按下标增量次序加入，即hf[n]为第1趟合并后生成的新子树的根，而hf[2n-2]为最后一趟合并后生成的子树的根，即整棵二叉树的根结点；

(3) 算法开始前，hf表中全部2n-1个结点的parent域设为-1，表示空指针，parent为-1的结点也意味着该结点为子树的根结点。

## 6.6 Huffman二叉树 续13

```
void createHF(HFT &a)
// a.n, a.s[0..n-1], a.weight[0..n-1]已知, 求a.hf和a.code
{  //初始化
  m=2*a.n-1;
  for(i=0; i<m; i++)
     a.hf[i].parent=a.hf[i].lchild=a.hf[i].rchild=-1;
  for(i=0; i<a.n; i++) a.hf[i].w=a.weight[i];
  //构造Huffman二叉树，生成a.hf[n..m-1]
  for(i=a.n; i<m; i++)
    { for(j1=0; j1<i; j1++)  if(a.hf[j1].parent==-1) break;
      for(j=j1+1; j<i; j++)
        if(a.hf[j].parent==-1&&a.hf[j].w<a.hf[j1].w) j1=j;
```



## 6.6 Huffman二叉树 续14

```
      a.hf[j1].parent=i;
      a.hf[i].lchild=j1;
      for(j2=0; j2<i; j2++) if(a.hf[j2].parent==-1) break;
      for(j=j2+1; j<i; j++)
         if(a.hf[j].parent==-1&&a.hf[j].w<a.hf[j2].w) j2=j;
      a.hf[j2].parent=i;
      a.hf[i].rchild=j2;
      a.hf[i].w=a.hf[j1].w+a.hf[j2].w;
    }
  //以下生成code字符串数组
```



## 6.6 Huffman二叉树 续15

```
  //以下生成code字符串数组
   for(i=0; i<a.n; i++)
    { j=i; p=a.code[i]; //j从第i个叶子出发上溯至根结点
      while(a.hf[j].parent!=-1)
      { child=j; j=a.hf[j].parent;
        if(a.hf[j].lchild==child) *p++='1'; else *p++='0';
      }
      *p='\0';
      q=a.code[i]; p--;    //字符串逆序
      while(q<p) { ch=*q; *q=*p; *p=ch; q++; p--; }
    }
}
```

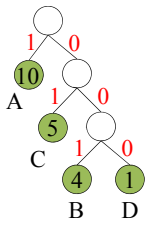


## 6.6 Huffman二叉树 续16

**5. Huffman编码的译码算法**

设接收二进制序列用字符串表示，从根结点出发，'1'向左，'0'向右（或'1'向右，'0'向左）深入，直至到达某个叶子结点时，输出叶子结点对应的符号；重复此过程，直到接收序列全部译出。

**例**：Huffman树如右图，接收到01110011000，则译码为：CAABAD






## 6.6 Huffman二叉树 续17

```
void dec(HFT &a, char receive[], ElemTp decoded[])
{ i=j=0;
  while(receive[i]!='\0')
    { k=2*a.n-1;   //k指向根结点
      while(k>=a.n&&receive[i]!='\0')
          if(receive[i++]=='1') k=a.hf[k].lchild;
            else k=a.hf[k].rchild;
      if(k<a.n) decoded[j++]=a.s[k];  //输出一个符号
      //若k<a.n，则k指向一个叶子结点
    }
  decoded[j]=结束标志;
}
```



## 第6章 作业

**第6章第一次作业：**

**1. 上机题**

(1) 编写完整程序，用先序遍历法建立二叉树的二叉链表存储结构。输出该二叉树的先、中、后序遍历结点访问次序以及层次遍历结点访问次序。（建议结点数据域类型为char）

(2) 从键盘输入n个数据建立n元完全二叉树顺序存储结构。实现该完全二叉树的先、中、后序遍历。

**2. 写算法**

(1) 已知二叉树（二叉链表）根结点指针为bt，求该二叉树中的叶子数目。

## 第6章 作业 续1

(2) 已知某二叉树（三叉链表）的根结点地址root，该树中各结点的左、右儿子指针域已正确填充，写一个算法将所有结点的双亲指针域正确填充。
(3) 已知某二叉树（二叉链表）的根结点指针bt。编写算法，将该二叉树中所有结点的左右子树互换。
(4) 已知n个结点的完全二叉树结点数据域值按结点编号次序顺序存于一维数组（元素下标范围0..n-1）。编写算法，由该数组首地址以及数组长度n建立对应的二叉链表存储结构。



## 第6章 作业 续2

**第6章第二次作业：**

**1. 上机题**

(1) 编写完整程序，实现中序遍历线索二叉树存储结构、线索化以及中序遍历。
(2) 编写完整程序，用堆栈实现前/中/后序非递归遍历，并与递归遍历结果比较。

**2. 写算法**

(1) 二叉树的直径定义为从根结点至叶子的最大路径长度。编写算法，求二叉树(二叉链表)的直径。
(2) 已知二叉树(二叉链表)根结点指针bt，树中两个结点的指针p、q。编写算法求距离结点*p和*q最近的公共祖先的地址。



## 第6章 作业 续3

(3) 已知二叉树(二叉链表)根结点指针bt，利用二叉树叶子结点的rchild指针域将所有叶子结点从左向右连接成一个单向链表。算法返回单向链表头结点指针(即最左边第1个叶子结点的地址)。

**第6章第三次作业：**

**上机题**

1. 哈夫曼编码生成与译码。输入符号数(序号用英文字母A, B, C, …表示)以及各符号出现概率，以字符串形式输出各符号对应的二进制哈夫曼编码。以字符串形式输入接收到的比特序列，输出译码后的符号序列。建议用菜单形式提供功能。

# 第7章 图





## 7.1 图的概念和术语

**1. 图的概念**

图(**Graph**)可以简单表示为$G=<V, E>$，其中$V$称为顶点(vertex)集合，$E$称为边(edge)集合。

(1) 无向图(undirect graph)

$E$中的每条边不带方向，称为无向图。

(2) 有向图(direct graph)

$E$中的每条边具有方向，称为有向图。

(3) 混合图

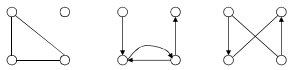
$E$中的一些边不带方向， 另一些边带有方向。

(4) 图的阶

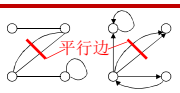
指图的顶点数目，即顶点集$V$中的元素个数。



## 7.1 图的概念和术语 续1



无向图 有向图 混合图




多重图

简单图

(5) 多重图

拥有平行边或自环的图。

(6) 简单图

不含平行边和自环的图。(**本课程仅讨论简单图，以后若非特别声明，所谓图仅指简单图**)



## 7.1 图的概念和术语 续2

(7) 边的表示方法与有关术语

**a**. 无向图的边称为**无向边**(edge)，它用**无序偶**表示

$v_i$ ○——○ $v_j$ $(v_i, v_j) \Leftrightarrow (v_j, v_i)$ 表示该边

称顶点$v_i$与$v_j$**相互邻接**或互为**邻接点**(adjacent)；边$(v_i, v_j)$**依附于**(incident)顶点$v_i$和$v_j$或与顶点$v_i$和$v_j$**相关联**。

**b**. 有向图的边称为**有向边**或**弧**(arc)，它用有**序偶**表示

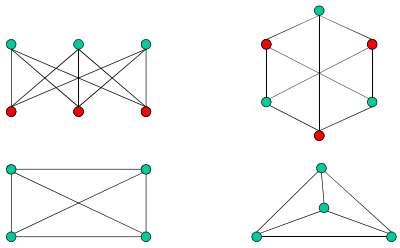
$v_i$ ○——→○ $v_j$ $<v_i, v_j>$ 表示该弧，$<v_i, v_j> \neq <v_j, v_i>$

称顶点$v_i$为**弧尾**(tail)或**始点**(initial node)，顶点$v_j$为**弧头**(head)或**终端点**(terminal node)；$v_i$邻接至$v_j$，而$v_j$邻接自$v_i$；弧$<v_i, v_j>$依附于或关联顶点$v_i$和$v_j$。



## 7.1 图的概念和术语 续3

图论中的图(graph)表示的是顶点之间的邻接关系，以下“形状”完全不同的“图”在图论中是相同的图。





## 7.1 图的概念和术语 续4

(8) 顶点的度(degree)

**a**. 无向图顶点的度定义为与该顶点相关联的边的数目；

**性质1**：无向图中，各顶点的度数和等于边数的2倍。

**b**. 有向图顶点的度定义为与该顶度相关联的弧的数目。即，有向图顶点的度=**入度**(indegree)+**出度**(outdegree)，其中入度定义为连接该顶点的弧头的数目；出度定义为连接该顶点的弧尾的数目。

**性质2**：有向图中，顶点的入度和=出度和=弧的数目。

(9) 完全图

**a**. $n$阶无向简单图中，若每个顶点的度均为$n$-1，称该图为无向完全图。

## 7.1 图的概念和术语 续5

**性质3**：$n$阶无向完全图边的数目为

$$C_n^2 = \frac{n(n-1)}{2}$$

**b.** $n$阶有向简单图中，若每个顶点的入度=出度=$n$-1，称该图为$n$阶有向完全图。

**性质4**：$n$阶有向完全图弧的数目为$n(n-1)$。

(10) 网(Network)

若图中的边带有**权重**(weight)，称为网。边上的权重一般代表某种代价或耗费。比如：顶点表示城市，边表示城市间的公路，则权重可以用公路里程来表示。若边上的**权重为无穷大**，一般表示代价无穷大等含义。



## 7.1 图的概念和术语 续6

(11) 稀疏图(sparse graph)与稠密图(dense graph)

若无向图或有向图有$e$条边或弧，若$e$很小(如$e<n\log_2 n$)，称为稀疏图，否则称为稠密图。

(12) 子图

对于图$G=<V, E>$，若有另一图$G'=<V', E'>$满足$V'\subseteq V$且$E'\subseteq E$，称图$G'$为$G$的子图。

### 2. 图的路径

(1) 路径(path)

图$G=<V, E>$中，从任一顶点开始，由边或弧的**邻接至**关系构成的有限长顶点序列称为路径。



## 7.1 图的概念和术语 续7

**注意**：

a. 有向图的路径必须沿弧的方向构成顶点序列；

b. 构成路径的顶点可能重复出现(即允许反复绕圈)。

(2) 路径长度

路径中边或弧的数目。

(3) 简单路径

除第一个和最后一个顶点外，路径中无其它重复出现的顶点，称为简单路径。

(4) 回路或环(cycle)

路径中的第一个顶点和最后一个顶点相同时，称为回路或环。



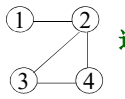
## 7.1 图的概念和术语 续8

### 3. 图的连通性

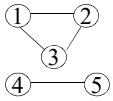
(1) 无向连通图

在无向图中，若从顶点$v_i$到$v_j$有路径，则称$v_i$和$v_j$是连通的。若该图中任意两个顶点都是连通的，则称它是连通图。

(2) 连通分量：无向图中的极大连通子图(包括子图中的所有顶点和所有边)称为连通分量或连通分支。连通图也可以定义为连通分支数等于1的图。




连通图




2个连通分支的非连通图



## 7.1 图的概念和术语 续9

(3) 有向连通图

在有向图中，任一对顶点$v_i$和$v_j$($v_i \neq v_j$)，若从$v_i$到$v_j$以及从$v_j$到$v_i$均连通(即存在路径)，称它是**强连通**的。

(4) 强连通分量

有向图中的极大强连通子图称为强连通分量。

**性质1**：有向强连通图的充要条件是该图存在一个回路经过每个顶点至少1次。

**性质2**：$n$阶无向连通图中至少有$n$-1条边； $n$阶有向连通图中至少有$n$条边。

**例如**，3个顶点组成的最小无向和有向连通图



## 7.1 图的概念和术语 续10完

(5) 生成树

一个$n$阶连通图的生成树是一个极小连通子图，它包含图中全部$n$个顶点以及保证该子图是连通图的最少的$n$-1条边。

**性质3**：在生成树上增加任何一条边，必形成回路。

(6) 有向树与生成森林

如果一个有向图恰有一个顶点入度为0，其余顶点的入度均为1，则是一棵**有向树**。一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，但只有中以构成若干棵不相交的有向树的弧。

## 7.2 图的存储结构

### 1. 邻接矩阵

若$n$阶图表示为$G=<V, E>$，其中$V=\{v_0, v_1, \ldots, v_{n-1}\}$，则定义

$$(a_{ij})_{n\times n}=\begin{cases}1 & (v_i,v_j)\in E \text{或} <v_i,v_j>\in E\\ 0 & \text{反之}\end{cases}$$

若图$G$为$n$阶网，则定义

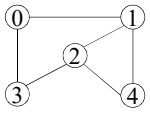
$$(a_{ij})_{n\times n}=\begin{cases}w_{ij} & (v_i,v_j)\in E \text{或} <v_i,v_j>\in E\\ \infty & \text{反之}\end{cases}$$

其中，$w_{ij}$为边$(v_i, v_j)$或弧$<v_i, v_j>$上的权重。



## 7.2 图的存储结构 续1

**例1** 无向图邻接矩阵的示例




$$A=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

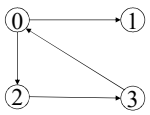
**无向简单图邻接矩阵的性质**：

(1) 关于主对角线对称，即$A=A^T$；
(2) 主对角线元素全为0；
(3) 矩阵中1的数目=边数的2倍；
(4) 第$i$行1的数目=第$i$列1的数目=顶点$v_i$的度。



## 7.2 图的存储结构 续2

**例2** 有向图邻接矩阵的示例




$$A=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}$$

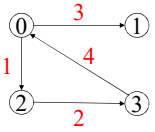
**有向简单图邻接矩阵的性质**：

(1) 不一定关于主对角线对称；
(2) 主对角线元素全为0；
(3) 矩阵中1的数目=弧的数目；
(4) 第$i$行1的数目=顶点$v_i$的出度；
(5) 第$i$列1的数目=顶点$v_i$的入度。



## 7.2 图的存储结构 续3

**例3** 网的邻接矩阵示例




$$A=\begin{bmatrix}\infty&3&1&\infty\\\infty&\infty&\infty&\infty\\\infty&\infty&\infty&2\\4&\infty&\infty&\infty\end{bmatrix}$$

```
//邻接矩阵的C语言实现
#define MAX_N   最大顶点数
typedef struct
{ int n;  //n为实际结点数, n<=MAX_N
  bool arcs[MAX_N][MAX_N];  //邻接矩阵
} Graph;
```



## 7.2 图的存储结构 续4

### 2. 邻接表与逆邻接表

若$n$阶图表示为$G=<V, E>$，其中$V=\{v_0, v_1, \ldots, v_{n-1}\}$，则可用链表实现图的存储结构。

**(1) 邻接表**

**a. 无向图**：关联顶点$v_i$的所有边组成的集合用单链表实现存储，头结点存储顶点$v_i$的编号和信息，其余结点存储邻接于顶点$v_i$的其它顶点的编号、边的权重和信息。这样共形成$n$个单链表，称为邻接表。

**b. 有向图**：以顶点$v_i$为弧尾的所有弧组成的集合用单链表实现存储，头结点存储弧尾$v_i$的编号和信息，其余结点存储弧头顶点编号、弧的权重和信息。



## 7.2 图的存储结构 续5

**头结点**(存储顶点$v_i$)

| data | info | i | firstarc |
|---|---|---|---|

```
typedef struct
{ //顶点数据(可选)
  ElemTp data;
  //顶点信息(可选)
  InfoTp info;
  int i;  //顶点下标
  ArcNode *firstarc;
} HNode;
```

**表结点**(存储边或弧)

| w | info | j | nextarc |
|---|---|---|---|

```
typedef struct node
{ //边或弧的权重(可选)
  double w;
  //边或弧的信息(可选)
  InfoTp info;
  int j;  //邻接点下标
  struct node *nextarc;
} ArcNode;
```

## 7.2 图的存储结构 续6

```
#define MAX_N  最大顶点数
typedef  enum { DG, UDG, DN, UDN } GraphKind;
// DG:有向图, UDG:无向图, DN:有向网, UDN:无向网
typedef   struct
{  HNode h[MAX_N];  //头结点形成数组
   int n, e;   //n:实际顶点数; e:边或弧的数目
   Graphkind   kind;   //图的类型(可选)
} ALGraph;
```

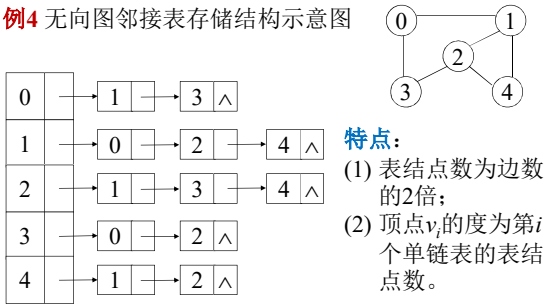
**(2) 逆邻接表**

有向图中，表结点存储**邻接至**顶点$v_i$的所有弧，即头结点是弧头，表结点是弧尾。



## 7.2 图的存储结构 续7

**例4** 无向图邻接表存储结构示意图







**特点**：

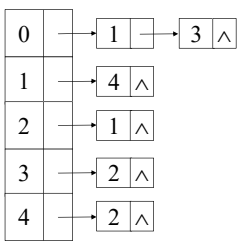
(1) 表结点数为边数的2倍；

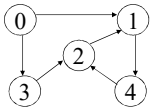
(2) 顶点$v_i$的度为第$i$个单链表的表结点数。



## 7.2 图的存储结构 续8

**例5** 有向图邻接表存储结构示意图




**特点**：

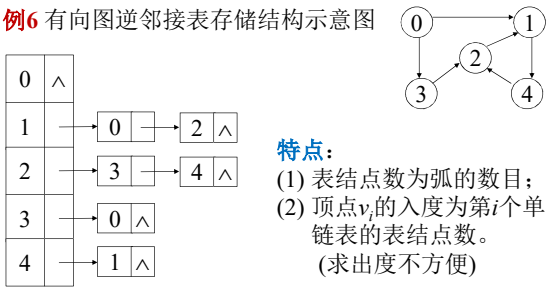
(1) 表结点数为弧的数目；

(2) 顶点$v_i$的出度为第$i$个单链表的表结点数。
(求入度不方便)



## 7.2 图的存储结构 续9

**例6** 有向图逆邻接表存储结构示意图







**特点**：

(1) 表结点数为弧的数目；

(2) 顶点$v_i$的入度为第$i$个单链表的表结点数。
(求出度不方便)



## 7.2 图的存储结构 续10

**3. 有向图的十字链表**

**头结点**(存储顶点$v_i$)

| data | info | i | firstin | firstout |
|---|---|---|---|---|

**表结点**(存储弧<$v_i$, $v_j$>)

| w | info | i | j | hlink | vlink |
|---|---|---|---|---|---|

每个表结点(弧<$v_i$, $v_j$>)在水平方向构成单链表，形成以$v_i$为弧尾的所有弧组成的集合；

每个表结点(弧<$v_i$, $v_j$>)在垂直方向构成单链表，形成以$v_j$为弧头的所有弧组成的集合。



## 7.2 图的存储结构 续11

```
typedef struct
{ //顶点数据(可选)
  ElemTp data;
  //顶点信息(可选)
  InfoTp info;
  int i;  //顶点下标
  OLANode *firstin;
  OLANode *firstout;
} OLHNode;
```

```
typedef struct node
{ //弧的权重(可选)
  double w;
  //弧的信息(可选)
  InfoTp info;
  int i, j;  //弧的端点下标
  struct node *hlink;
  struct node *vlink;
} OLANode;
```

## 7.2 图的存储结构 续12

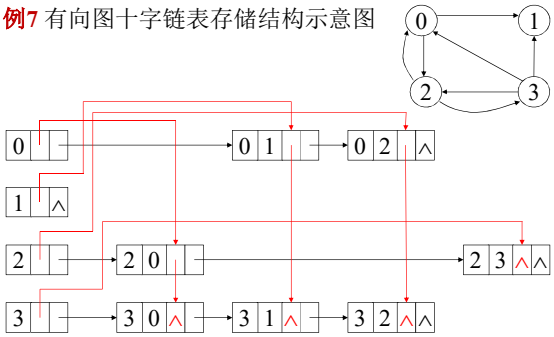
```
typedef  struct
{  OLHNode h[MAX_N];  //头结点形成数组
   int n, e;   //n:实际顶点数; e:边或弧的数目
   Graphkind  kind;   //图类型(可选)
} OLGraph;
```

**十字链表特点**:
(1) 表结点数等于弧的数目;
(2) 求入度和出度都很方便。



## 7.2 图的存储结构 续13

**例7** 有向图十字链表存储结构示意图






## 7.2 图的存储结构 续14

**4. 无向图的邻接多重表**

采用类似十字链表的思想实现无向图存储。任意边$(v_i, v_j)$只存储一个表结点，每个表结点有inext和jnext两个指针域，inext指向关联于顶点$v_i$的下一条边，而jnext指向关联于顶点$v_j$的下一边条。顶点$v_i$的头结点仅含一个指针域，指向关联于$v_i$的第1条边。

**头结点**(存储顶点$v_i$)

| data | info | i | firstedge |
|---|---|---|---|

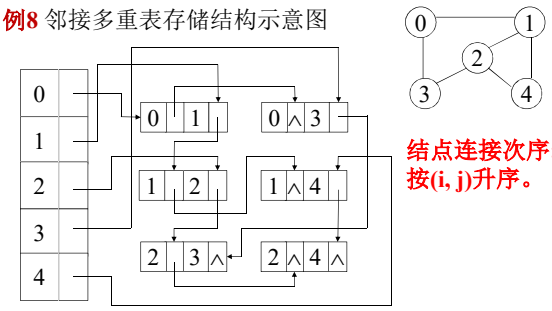
**表结点**(存储边$(v_i, v_j)$)

| w | info | i | j | inext | jnext |
|---|---|---|---|---|---|



## 7.2 图的存储结构 续15

**例8** 邻接多重表存储结构示意图




**结点连接次序: 按(i, j)升序。**



## 7.2 图的存储结构 续16

**5. 图的存储结构建立算法**

**数据输入怎样实现?**

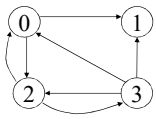
使用字符文件存储图(网)的顶点以及边(弧)信息。该字符文件作为程序输入建立图的存储结构。

a. 顶点数***n***和边(弧)的数目***e***;
b. 可选***n***个顶点数据(data)和信息(info);
*c.* ***e***条边(弧)的信息(每条边或弧用多元组表示):
***i*** ***j*** $w_{ij}$ $info_{ij}$
其中，$w_{ij}$和$info_{ij}$根据需要可选。



## 7.2 图的存储结构 续17

**例9** 有向图十字链表存储结构的建立




数据输入文件**input.txt**的内容:

```
4  7
0 1   0 2   2 0   2 3
3 0   3 1   3 2
```

```
void crtGraph1(OLGraph &G)
{ int i, j, k; ifstream f("input.txt"); if(!f) return;
  f>>G.n>>G.e;  //读入顶点数和弧数
  for(i=0; i<G.n; i++)
  {  G.h[i].i=i;   //初始化顶点序号
     G.h[i].firstin=G.h[i].firstout=NULL;
  }
```

## 7.2 图的存储结构 续18

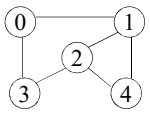
```
for(k=0; k<e; k++) //建立e个表结点
{ f>>i>>j;  //读入弧的始点和终点序号<i, j>
  OLANode *pr, *p, *q=new OLANode;
  q->i=i; q->j=j; q->hlink=q->vlink=NULL;
  //水平链表按弧头序号升序
  pr=NULL; p=G.h[i].firstout;
  while(p&&p->j<j) { pr=p; p=p->hlink; }
  if(pr==NULL) { q->hlink=p; G.h[i].firstout=q; }
    else { pr->hlink=q; q->hlink=p; }
  //垂直链表按弧尾序号升序
  pr=NULL; p=G.h[i].firstin;
  while(p&&p->i<i) { pr=p; p=p->vlink; }
```



## 7.2 图的存储结构 续19

```
  if(pr==NULL) { q->vlink=p; G.h[i].firstin=q; }
    else { pr->vlink=q; q->vlink=p; }
} // 建立表结点循环结束
f.close();
}  //函数结束
```

**例10** 无向图邻接表存储结构的建立




输入文件**input.txt**内容：

```
5 6
0 1   0 3
1 2   1 4
2 3   2 4
```



## 7.2 图的存储结构 续20

```
void crtGraph2(ALGraph &G)
{ int i, j, k; ifstream f("input.txt");
  f>>G.n>>G.e;  //读入顶点数和弧数
  for(i=0; i<G.n; i++)
  {  G.h[i].i=i;  //初始化顶点序号
     G.h[i].firstarc=NULL;
  }
  for(k=0; k<G.e; k++)
  { ArcNode *pr, *p, *q=new ArcNode;
    q->j=j; q->nextarc=NULL;  //建第i行表结点并插入
    pr=NULL; p=G.h[i].firstarc;
```



## 7.2 图的存储结构 续21完

```
    while(p&&p->j<j) { pr=p; p=p->nextarc; }
    if(pr==NULL) { q->nextarc=p; G.h[i].firstarc=q; }
      else { pr->nextarc=q; q->nextarc=p; }
    q=new ArcNode; q->j=i;  //建立第j行表结点并插入
    pr=NULL; p=G.h[j].firstarc;
    while(p&&p->j<i) { pr=p; p=p->nextarc; }
    if(pr==NULL) { q->nextarc=p; G.h[j].firstarc=q; }
      else { pr->nextarc=q; q->nextarc=p; }
  }
  f.close();
}
```



## 7.3 图的遍历

**1. 遍历的定义**

从图中某顶点出发，沿路径方向访问每个顶点一次且仅一次。

**2. 图遍历算法的辅助数据结构**

为避免顶点重复访问，需定义一个顶点标志数组**visited**[0..n-1]，标记每个顶点是否已访问。

**3. 图的深度优先搜索(Depth First Search)算法**

**搜索原则**：沿出发顶点的第1条路径尽量深入，遍历路径上的所有顶点；然后退回到该顶点，搜索第2条，第3条，…，路径，直到以该顶点为始点的所有路径上的顶点都已访问过(这是**递归算法**)。对于非连通图，需从每个顶点出发，尝试深度优先遍历。



## 7.3 图的遍历 续1

```
void DFStravel(Graph  &G)  //Graph为邻接矩阵
{  bool *visited=new bool[G.n];
   for(i=0; i<G.n; i++) visted[i]=false;
   for(i=0; i<G.n; i++) //保证非连通图的遍历
      if (!visited[i])  DFS(G, i);
   delete []visited;
}
void DFS(Graph  &G, int i) //从vi出发深度优先搜索
{  visit(i); visited[i]=true;
   for (j=First_Adj(G, i); j!=-1; j=Next_Adj(G, i, j))
      if (!visited[j])  DFS(G, j);
}
```

## 7.3 图的遍历 续2

**4. 图的宽度优先搜索(Breadth First Search)算法**

**搜索原则**：

(1) 访问遍历出发顶点，该顶点入队；

(2) 队列不空，则队头顶点出队；

(3) 访问出队顶点所有的未访问邻接点并将访问的顶点入队；

(4) 重复(2), (3), 直到队列为空。

以上为非递归算法，需设队列实现算法。

对于非连通图，需从每个顶点出发，尝试宽度优先搜索。



## 7.3 图的遍历 续3

```
void BFStravel(Graph &G) //Graph为邻接矩阵
{ bool *visited=new bool[G.n];
  for(i=0; i<G.n; i++) visted[i]=false;
  InitQuene(Q);
  for(i=0; i<G.n; i++)
    if (!visited[i])
    { visit(i); visited[i]=true; enQueue(Q, i);
      while(!Empty(Q))
      { u=delQueue(Q);
        for(v=First_Adj(G,u);v!=-1;v=Next_Adj(G,u,v))
          if(!visited[v])
```



## 7.3 图的遍历 续4

```
        { visit(v); visted[v]=true; enQueue(Q, v);
        } // end of if !visited[v]
      }   // end of while
    }     // end of if !visited[i]
  delete []visited;
}
```



## 7.3 图的遍历 续5

**5. 求第1邻接点和下一个邻接点算法**

```
//邻接矩阵
int First_Adj(Graph &G, int u)
{ for(v=0; v<G.n; v++) if(G.arcs[u][v]!=0) break;
  if(v<G.n) return v;
  return -1;
}
int  Next_Adj(Graph &G, int u, int v)
{ for(++v; v<G.n; v++) if(G.arcs[u][v]) break;
  if(v<G.n) return v;
  return -1;
}
```



## 7.3 图的遍历 续6

```
//邻接表和十字链表
for(v=First_Adj(G, u); v!=-1; v=Next_Adj(G, u, v))
用以下循环语句代替
for(p=G.h[u].firstarc, v=p?p->j:-1; v!=-1;  \
p=p->nextarc, v=p?p->j:-1)
若为十字链表，则用以下循环语句代替
for(p=G.h[u].firstout, v=p?p->j:-1; v!=-1;  \
p=p->hlink, v=p?p->j:-1)
```



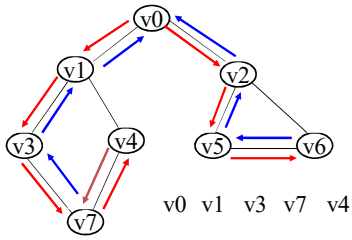
## 7.3 图的遍历 续7

**6. 图的遍历算法的复杂度**

| 算法 | DFSTravel | | BFSTravel | |
|---|---|---|---|---|
| 存储结构 | 邻接矩阵 | 邻接表 | 邻接矩阵 | 邻接表 |
| $T(n)$ | $O(n^2)$ | $O(n+e)$ | $O(n^2)$ | $O(n+e)$ |
| $S(n)$ | 辅助空间及栈深度 $O(n)$ | 辅助空间及栈深度 $O(n)$ | 辅助空间及队列长 $O(n)$ | 辅助空间及队列长 $O(n)$ |

## 7.3 图的遍历 续8

**例1** 深度优先遍历顶点访问次序(从顶点v0出发)



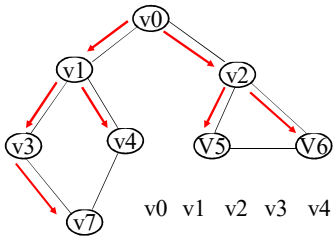

v0 v1 v3 v7 v4 v2 v5 v6

求邻接点次序不同，可得到不同的访问序列，如：
v0, v2, v5, v6, v1, v3, v7, v4等



## 7.3 图的遍历 续9

**例2** 宽度优先遍历顶点访问次序(从顶点v0出发)

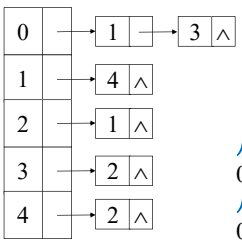


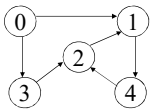

v0 v1 v2 v3 v4 v5 v6 v7



## 7.3 图的遍历 续10完

**例3** 给定存储结构示意图，则遍历次序唯一确定




**从0出发深度优先次序**：
0, 1, 4, 2, 3

**从0出发宽度优先次序**：
0, 1, 3, 4, 2



## 7.4 连通性与最小生成树

**1. 连通性的判断方法**

a. 无向图从任一顶点出发，若DFS或BFS可访问所有顶点，则该图是连通图；

b. 有向图从每个顶点出发，若DFS或BFS均可访问所有顶点，则该图是强连通图。

**2. 求连通分支**

无向图DFSTravel或BFSTravel过程中，从顶点出发进行DFS或BFS的次数为连通分支数。

**3. 求生成树**

DFSTravel或BFSTravel经历的路径和顶点构成连通分支的生成树森林。若图是连通的，则得到生成树。



## 7.4 连通性与最小生成树 续1

**4. 最小生成树**

**(1) 最小生成树的概念**

对于带权无向图(无向网)，其所有生成树中，边上权值之和最小的称为最小生成树。

**注意**：最小生成树的构形不一定唯一。

**(2) 最小生成树生成算法的基本原理-MST性质**

**MST性质**：假设$G=(V, E)$是一个连通网，$U$是顶点$V$的一个非空子集。若$(u, v)$是满足条件$u\in U$且$v\in V-U$的所有边中一条具有最小权值的边，则必存在一棵包含边$(u, v)$的最小生成树。



## 7.4 连通性与最小生成树 续2

**MST性质的证明**:

用反证法。

设$G$的所有最小生成树都不包含边$(u, v)$。

若$T$为$G$的任意一棵最小生成树，则

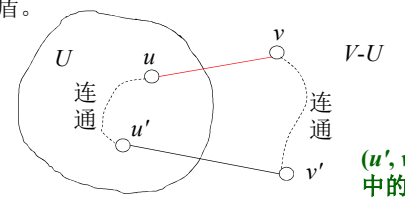
$T$中必有边$(u', v')$，满足$u'\in U$， $v'\in V\text{-}U$，使得$u'$和$u$在$U$中连通，$v'$和$v$在$V\text{-}U$中连通。

理由：

因为，若不存在满足上述条件的边$(u', v')$，则$u$和$v$不可能连通，这与$T$是生成树矛盾(生成树中，任意一对顶点都是连通的)。

## 7.4 连通性与最小生成树 续3

将边$(u, v)$加入$T$中，则形成回路，如下图所示。去掉边$(u', v')$，所得图所有顶点仍是连通的，即构成另一棵生成树$T'$。因$(u', v')$的权重大于$(u, v)$的权重，故生成树$T$的权重和大于$T'$的权重和，这与$T$是最小生成树矛盾。






## 7.4 连通性与最小生成树 续4

**(3) 普里姆(Prim)算法**

**算法思想**：直接运用MST性质。

假设$G=(V, E)$是连通网，$TE$是$G$上最小生成树中边的集合。算法从$U=\{u_0\}$ $(u_0 \in V)$且$TE=\{\}$开始，重复执行下列操作：

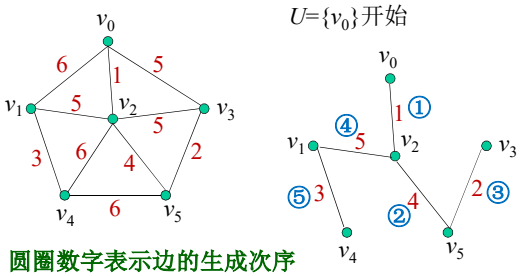
在所有$u \in U$且$v \in V-U$的边$(u, v)$ 中找一条权值最小的边$(u', v')$并入集合$TE$中，同时$v'$并入$U$，直到$V=U$为止。

最后，$TE$中必有$n$-1条边。$T=(V, TE)$就是$G$的一棵最小生成树。



## 7.4 连通性与最小生成树 续5

**例1** 用Prim算法手工构造最小生成树






## 7.4 连通性与最小生成树 续6

**Prim算法的实现**：

设置辅助数组closedge[0..$n$-1]，其中$n$表示无向连通网的顶点总数。

设$n$个顶点组成的集合$V=\{v_0, v_1, \ldots, v_{n-1}\}$且各顶点编号与closedge数组下标对应。若初始时$U=\{v_0\}$，在Prim算法执行过程中，对任意顶点$v_i \in V-U$，closedge[$i$]包含两个域，即

closedge[$i$].lowcost=min{cost($v_i$, $v_j$)| $v_j \in U$}

closedge[$i$].vex=$j'$，满足$v_{j'} \in U$ 且 cost($v_i$, $v_{j'}$)=上述lowcost



## 7.4 连通性与最小生成树 续7

若顶点$v_i$已并入集合$U$，则令closedge[$i$].lowcost=0；
若顶点$v_i$在$V-U$中，且与$U$中每个顶点无边相边，可令closedge[i].lowcost=∞。

(1) 每趟从所有$v_i \in V-U$中(closedge[$i$].lowcost>0表示$v_i \in V-U$)选择lowcost最小的$v_i$，将$v_i$并入集合$U$。

(2) 假设每趟并入$U$集合的顶点为$v_i$，则

a. 令closedge[$i$].lowcost=0;

b. 调整其它lowcost>0的所有closedge元素，即
$\forall v_j \in V-U$，若cost($v_i$, $v_j$)<closedge[$j$].lowcost，则更新
closedge[$j$].lowcost=cost($v_i$, $v_j$)
closedge[$j$].vex=$i$
否则，closedge[$j$]不更新。



## 7.4 连通性与最小生成树 续8

**问题**：

为什么更新closedge[$j$]时，只需要将其lowcost与cost($v_i$, $v_j$)比较，就可保证更新后的closedge[$j$].lowcost是从顶点$v_j$连入$U$集合的所有边中权重最小的？

**答**：因为$U$集合在并入顶点$v_i$前，closedge[$j$].lowcost是$v_j$连入$U$的所有边中权重最小的，故$U$并入$v_i$后，$U$集合只增加了顶点$v_i$，$v_j$连入$U$集合的所有边的权值最小值只需要和这个新加入的顶点引起的边($v_i$, $v_j$)比较。

## 7.4 连通性与最小生成树 续9

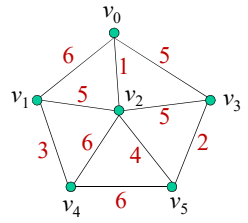
**例1的closedge数组动态变化过程** 看图

| i / closedge | 0 | 1 | 2 | 3 | 4 | 5 | U |
|---|---|---|---|---|---|---|---|
| 初始 | 0, **0** | 0, 6 | 0, 1 | 0, 5 | 0, ∞ | 0, ∞ | $\{v_0\}$ |
| 第0趟 | 0, **0** | 2, 5 | 0, **0** | 0, 5 | 2, 6 | 2, 4 | $\{v_0,v_2\}$ |
| 第1趟 | 0, **0** | 2, 5 | 0, **0** | 5, 2 | 2, 6 | 2, **0** | $\{v_0,v_2,v_5\}$ |
| 第2趟 | 0, **0** | 2, 5 | 0, **0** | 5, **0** | 2, 6 | 2, **0** | $\{v_0,v_2,v_5,v_3\}$ |
| 第3趟 | 0, **0** | 2, **0** | 0, **0** | 5, **0** | 1, 3 | 2, **0** | $\{v_0,v_2,v_5,v_3,v_1\}$ |
| 第4趟 | 0, **0** | 2, **0** | 0, **0** | 5, **0** | 1, **0** | 2, **0** | $\{v_0,v_2,v_5,v_3,v_1,v_4\}$ |

vex, lowcost



## Prim算法closedege数组动态变化过程看图




回去



## 7.4 连通性与最小生成树 续10

**Prim算法**：由连通网邻接矩阵输出最小生成树的顶点产生次序、所有边及边上权重。

```
typedef  struct { double lowcost;
                  int vex;
                } CD_TP;  //closedge数组元素类型
void  prim(Network &G, int k)   //从顶点v_k出发
{ CD_TP *closedge=new CD_TP[G.n];
  //初始化closedge
  for(i=0; i<G.n; i++)   //从k号顶点出发
    { closedge[i].vex=k; closedge[i].lowcost=G.w[i][k]; }
  closedge[k].lowcost=0;
```



## 7.4 连通性与最小生成树 续11

```
  cout<<k;   //输出k号顶点
  for(m=0; m<G.n-1; m++)  //n-1趟循环
  { for(i=0; i<G.n; i++) if(closedge[i].lowcost>0) break;
    for(j=i+1; j<G.n; j++)
      if(closedge[j].lowcost>0&& \
         closedge[j].lowcost<closedge[i].lowcost)
         i=j;
    cout<<","<<i; //输出i号顶点
    closedge[i].lowcost=0;
    for(j=0; j<G.n; j++)
      if(closedge[j].lowcost>0&& \
         closedge[j].lowcost>G.w[j][i])
```



## 7.4 连通性与最小生成树 续12

```
          { closedge[j].lowcost=G.w[j][i];
            closedge[j].vex=i;
          }
  } //for m 循环结束
  cout<<endl; //以下输出n-1条边及边上的权重
  for(i=0; i<G.n; i++)
    if(i!=k)
    { cout<<"("<<i<<","<<closedge[i].vex<<")";
      cout<<"-"<<G.w[i][closedge[i].vex]<<endl;
    }
  delete []closedge;
}
```



## 7.4 连通性与最小生成树 续13

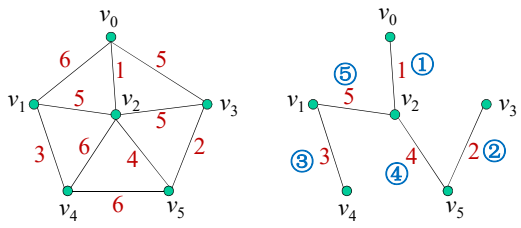
**(4) 克鲁斯卡尔(Kruskl)算法**

给定连通网$N=(V, E)$，令最小生成树的初始状态为只有$n$个顶点而无边的非连通图$T$，图中每个顶点自成一个连通分量。在$E$中选择最小权重的边，若该边依附的顶点落在$T$中不同的连通分量中，则将该边加入到$T$中，否则舍去此边而选择下一条权重最小的边。依次类推，直到$T$中所有顶点都在同一连通分量上为止。

**核心**：每次选择一条具有最小权值、且跨越两个不同连通分量的边，使两个不同连通分量变成一个连通分量。

## 7.4 连通性与最小生成树 续14

**例2 利用Kruskl算法手工求连通网的最小生成树**




圆圈数字表示边的生成次序



## 7.4 连通性与最小生成树 续15完

**Kruskl算法**：需使用堆和求等价类算法，不用掌握。

**(5) Prim和Kruskl算法的时间复杂度**

**Prim**: $T(n)=O(n^2)$, 适合边多的稠密度

**Kruskl**: $T(n)=O(e\log_2 e)$, 适合边少的稀疏图



## 7.5 AOV网络与拓扑排序

**1. 什么是AOV网络？**

以顶点表示活动(activity)，边表示活动之间的优先关系，这种有向图称为以顶点表示活动的网(Activity on Vertex Network),简称AOV网。

**AOV网的特点**：是有向无环图(Directed Acycline Graph, DAG)。

**AOV网的应用**：用于表示工程施工流程图或产品生产流程图等。如，DAG中的顶点表示子工程或工序(统称活动)，则弧<$u$, $v$>表示活动$v$的进行必须直接依赖活动$u$的完成。AOV网中，若顶点$u$至顶点$v$连通(存在路径)，称$u$为$v$的**前趋**，$v$为$u$的**后继**。

无环确保任何活动不以自身的完成为前提。



## 7.5 AOV网络与拓扑排序 续1

对整个工程和系统，人们关心的是两个方面的问题：

(1) 工程能否顺利进行

**方法**：本节的**拓扑排序**(即检验AOV网是否有环)

(2) 估算整个工程完成所必须的最短时间

**方法**：下一节求**关键路径**

**2. 什么是拓扑排序？**

**拓扑有序的概念**：DAG中，所有顶点的一个线性次序如果满足如下条件，则称该线性次序为拓扑有序：在该线性次序中，对于任意前后两个顶点$u$和$v$，或者$u$是$v$的前趋，或者$u$与$v$在DAG中不连通。(总之，在顶点的拓扑有序序列中，不能出现前面的顶点是后面顶点的后继。)



## 7.5 AOV网络与拓扑排序 续2

**拓扑有序序列的特点**：

a. DAG一定能够得到所有顶点的一个拓扑有序序列；

b. DAG所有顶点的拓扑有序序列一般不唯一；

c. DAG至少存在一个入度为0的顶点(入度为0的顶点代表着整个工程开始时必须进行的活动)；

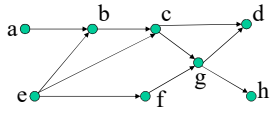
d. 若有向图无法得到包含所有顶点的拓扑有序序列，该图必有环，即该图不是DAG。

**拓扑排序**：生成DAG的一个拓扑有序序列的过程称为拓扑排序。



## 7.5 AOV网络与拓扑排序 续3

**例1** 对以下DAG图进行拓扑排序




a, e, b, f, c, g, d, h
e, a, f, b, c, g, h, d
…

**方法**：

(1) 输出入度为0的所有顶点(入度为0的顶点不止一个时，它们的输出次序可随意)；

(2) 从图中删除已输出的顶点和这些顶点射出的弧；

(3) 重复(1)(2), 直到图中存在环(即找不到入度为0的顶点)或剩余顶点数为0。

## 7.5 AOV网络与拓扑排序 续4

**3. 拓扑排序算法**

(1) 采用邻接表作为有向图的存储结构；

(2) 头结点增加一个数据域存放顶点入度，或者用一个数组indegree[0..n-1]来保存各顶点当前的入度；

(3) 为避免检测入度为0的顶点，设置一个栈(或队列)存放入度为0的顶点。

开始排序前，扫描indegree向量，将入度为0的顶点入栈。每次输出入度为0的顶点时，只需做出栈操作即可。

删除入度为0的顶点及其所有以它为尾的弧：只需将该顶点的所有邻接至顶点的入度减1，若减到0的，则入栈。



## 7.5 AOV网络与拓扑排序 续5

```
bool TopologicalSort(ALGraph &G)
//返回true，表示拓扑排序成功；返回false表示有环
{ //首先创建并初始化indegree数组
  int *indegree=new int[G.n];
  int i, count; ArcNode *p; Stack S;
  for(i=0; i<G.n; i++) indegree[i]=0;
  for(i=0; i<G.n; i++)
    for(p=G.h[i].firstarc; p!=NULL; p=p->nextarc)
      indegree[p->j]++;
  //以下开始拓扑排序算法
  InitStack(S);
```



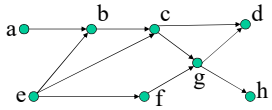
## 7.5 AOV网络与拓扑排序 续6

```
  for(i=0; i<G.n; i++) if(!indegree[i]) Push(S, i);
  count=0;
  while(!StackEmpty(S))
    { i=Pop(S); cout<<i; count++;
      for(p=G.h[i].firstarc; p; p=p->nextarc)
        if(!(--indegree[p->j])) Push(S, p->j);
    }
  delete []indegree;
  if(count<G.n) return false;
  return true;
}
```



## 7.5 AOV网络与拓扑排序 续7

**例2** 例1中的DAG应用拓扑排序算法，分别采用栈和队列保存入度为0的顶点，试给出对应的拓扑有序序列。




**答：** 设字母次序与下标次序相同，

用堆栈时，拓扑有序序列为：e, f, a, b, c, g, h, d

用队列时，拓扑有序序列为：a, e, b, f, c, g, d, h



## 7.5 AOV网络与拓扑排序 续8完

**4. 拓扑排序算法的复杂度**

(1) 辅助空间复杂度: $O(n)$ ***n*表示顶点数**

**问题**：堆栈的最大深度? 什么情况下获得最大深度?

**答**：$n$-1；

当1个入度为0的顶点邻接至其它$n$-1个顶点时。

(2) 时间复杂度: $O(n+e)$

**分析**：初始化indegree：$n+e$次循环 ***e*表示边数**

最开始入度为0的顶点压栈：$n$次循环

输出所有顶点：$e$次循环



## 7.6 AOE网络与关键路径

**1. 什么是AOE网络?**

AOE(Activity on Edge)网络是带权的有向无环图(DAG)，其中，顶点表示事件(event)，弧表示活动(activity)，弧上的权值表示活动持续的时间。通常，AOE网用来估算工程的完成时间。

一般说来，整个工程只有一个开始点和一个结束点，即正常情况下，网中只有一个入度为0的顶点(称为**源点**)和一个出度为0的顶点(称为**汇点**)。

针对AOE网研究的问题是：

**a**. 完成整个工程至少需要多少时间(最短时间)；

**b**. 哪些活动是影响工程进度的关键。

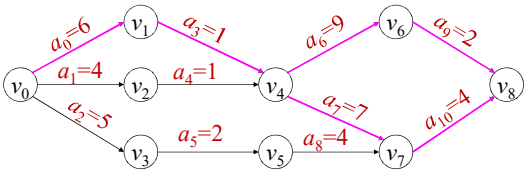
## 7.6 AOE网络与关键路径 续1

**2. 什么是关键路径？**

AOE网中，从源点到汇点的所有路径中，边上权值之和最大的路径称为关键路径。显然，关键路径上的权值之和就是完成工程所需的**最短时间**。

**注意**：关键路径可能不止一条。

**例1** AOE网络与关键路径示例






## 7.6 AOE网络与关键路径 续2

**3. 求关键路径的算法设计**

**(1) 定义如下辅助数据**

$v_e(i)$: 顶点$v_i$的最早开始时间

$v_l(i)$: 顶点$v_i$的最晚开始时间

$w<i, j>$: 活动$<v_i, v_j>$的持续时间，即该弧上的权值

$e(i, j)$: 活动$<v_i, v_j>$的最早开始时间

$l(i, j)$: 活动$<v_i, v_j>$的最晚开始时间

为保证工程完成时间尽可能短，令$v_e$(汇点)=$v_l$(汇点)

显然，我们有 $e(i, j)=v_e(i)$，

$l(i, j)=v_l(j)-w<i, j>$

即，活动的最早和最晚开始时间可由它所依附的顶点的最早开始和最晚开始时间计算。



## 7.6 AOE网络与关键路径 续3

对任一活动$<v_i, v_j>$，若

$e(i, j)=l(i, j)$，说明活动$<v_i, v_j>$不能延迟，否则整个工程会延迟，于是称$<v_i, v_j>$为**关键活动**；

$e(i, j)<l(i, j)$，说明活动$<v_i, v_j>$延迟0~$l(i, j)-e(i, j)$时间，不会影响整个工程进度。

**性质**：关键路径上的活动都是关键活动。

**(2) 怎样求关键路径？**

求关键路径须求得所有活动(弧)的$e(i, j)$和$l(i, j)$，以确定其中哪些是关键活动。而求$e(i, j)$和$l(i, j)$需先求得所有顶点(事件)的$v_e(i)$和$v_l(i)$。



## 7.6 AOE网络与关键路径 续4

**(3) 怎样求$v_e(i)$和$v_l(i)$？**

为表述方便，设AOE网络共有$n$个顶点，且源点为$v_s$，汇点为$v_t$。

**a**. 第1步，前向递推计算所有顶点的最早开始时间

$$\begin{cases} v_e(s)=0 \\ v_e(j)=\max_i\{v_e(i)+w<i, j>\} \end{cases}$$

其中，$i$满足$v_i\in${邻接至$v_j$的所有顶点组成的集合}；

**下标 $j$ 的次序为全部顶点的一个拓扑有序序列**。

$j$按拓扑有序递推计算上式，可确保在计算每个$v_e(j)$之前，邻接至$v_j$的所有顶点$v_i$的最早开始时间$v_e(i)$已经得到。故，**前向递推可由拓扑排序算法实现**。



## 7.6 AOE网络与关键路径 续5

**b**. 第2步，反向递推计算所有顶点的最早开始时间

$$\begin{cases} v_l(t)=v_e(t) \\ v_l(i)=\min_j\{v_l(j)-w<i, j>\} \end{cases}$$

其中，$j$满足$v_j\in${邻接自$v_i$的顶点集合}；

**下标 $i$ 的次序为全部顶点的一个逆拓扑有序序列**。

$i$按逆拓扑有序递推计算上式，可确保在计算每个$v_l(i)$之前，邻接自$v_i$的所有顶点$v_j$的最晚开始时间$v_l(j)$已经得到。

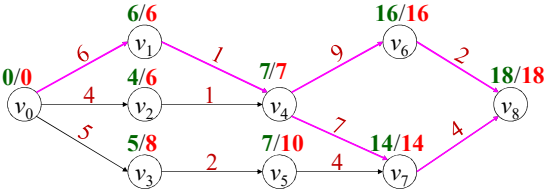
**问题**：**反向递推(逆拓扑排序)怎样实现？**

**答**：前向递推时，顶点按拓扑有序次序入栈，则反向递推时，顶点依次退栈即得到逆拓扑有序序列。



## 7.6 AOE网络与关键路径 续6

**例2** 例1中的AOE网络用手工方法求所有顶点的最早与最晚开始时间并给出关键路径。




**关键路径**：$v_0, v_1, v_4, v_6, v_8$

$v_0, v_1, v_4, v_7, v_8$

## 7.6 AOE网络与关键路径 续7

**4. 求所有顶点最早与最晚开始时间算法**

**说明**：因需要进行拓扑排序，适合采用邻接表实现AOE网络的存储。

算法所需辅助空间包括：

indegree[0..*n*-1] 所有顶点入度数组(*n*为顶点数)；

顶点序号栈S(最大深度*n*-1)，用于拓扑排序；

顶点序号栈T(最大深度*n*)，用于存储拓扑有序序列。

**输出**：ve[0..*n*-1] 所有顶点的最早开始时间；

vl[0..*n*-1] 所有顶点的最晚开始时间；



## 7.6 AOE网络与关键路径 续8

```
int CalVex(ALGraph &G, double ve[], double vl[])
//返回值为源点下标(-1表示有环); ve[], vl[]为结果
{ //首先创建并初始化indegree数组
  int *indegree=new int[G.n];
  int i, j, k, count; ArcNode *p; Stack S, T;
  for(i=0; i<G.n; i++) indegree[i]=0;
  for(i=0; i<G.n; i++)
    for(p=G.h[i].firstarc; p!=NULL; p=p->nextarc)
      indegree[p->j]++;
  //以下开始前向递推(拓扑排序)
  InitStack(S); InitStack(T);
  //所有顶点的最早开始时间初始化为0
  for(i=0; i<G.n; i++) ve[i]=0;
```



## 7.6 AOE网络与关键路径 续9

```
  for(i=0; i<G.n; i++)  //源点入栈(这里假定源点仅1个)
    if(!indegree[i]) { Push(S, i); k=i; break; }
  count=0;                          //k保存源点下标
  while(!EmptyStack(S))
    { i=Pop(S);Push(T, i); count++;
      for(p=G.h[i].firstarc; p; p=p->nextarc)
        { if(!(--indegree[p->j])) Push(S, p->j);
          double w=ve[i]+p->w;
          if(ve[p->j]<w) ve[p->j]=w;
        }  //注意此处实现前向递推公式的方法
    }
  delete []indegree;
  if(count<G.n) return -1;
```



## 7.6 AOE网络与关键路径 续10

```
  //以下开始反向递推
  if(!EmptyStack(T)) j=Pop(T); //汇点出栈
  //所有顶点最晚开始时间初始化为汇点最早开始时间
  for(i=0; i<G.n; i++) vl[i]=ve[j];
  while(!EmptyStack(T))
    { i=Pop(T);
      for(p=G.h[i].firstarc; p; p=p->nextarc)
        { double w=vl[p->j]-p->w;
          if(vl[i]>w) vl[i]=w;
        }
    }
  return k; //返回源点下标
}
```



## 7.6 AOE网络与关键路径 续11

**4. 输出所有关键活动算法**

**说明**：已知AOE网络的邻接表存储结构G，ve[0..n-1]，vl[0..n-1] (n表示顶点数)。

```
void PrtKA(ALGraph &G, double ve[], double vl[])
{ for(i=0; i<G.n; i++)
    for(p=G.h[i].firstarc; p; p=p->nextarc)
      { ee=ve[i]; ll=vl[p->j]-p->w;
        if(ee==ll) write(i, p->j, p->w);
      }
}
```



## 7.6 AOE网络与关键路径 续12完

**5. 算法时间复杂度**

前向递推(拓扑排序): $O(n+e)$

后向递推: $O(e)$

输出关键活动: $O(e)$

故，输出关键活动所需的总时间复杂度为$O(n+e)$

**6. 关键路径的意义**

当关键路径只有一条时，减少关键路径中关键活动的持续时间能够加快工程进度；若关键路径不止一条时，需要同时加快不同关键路径上的关键活动才可能加快工程进度。

## 7.7 最短路径

**1. 最短路径的概念**

给定$n$阶有向或无向网$N=(V, E)$，其中，$V=\{v_0, v_1, \dots, v_{n-1}\}$。设$P$表示顶点$v_i$到$v_j$的一条路径中全部边(弧)组成的集合，则该条路径的**带权路径长度**定义为$P$中的所有边(弧)的权值之和。顶点$v_i$到$v_j$的**最短路径**是指$v_i$到$v_j$的所有路径中带权路径长度最小的路径。

**3点说明**：

(1) 顶点$v_i$到$v_j$的最短路径不一定唯一；

(2) 若$v_i$到$v_j$不连通，则$v_i$到$v_j$的最短路径长度为∞；

(3) 对于$n$阶无向网，顶点对的组合数为$n(n-1)/2$，即共有$n(n-1)/2$个最短路径；对于$n$阶有向网，则总共有$n(n-1)$个最短路径。



## 7.7 最短路径 续1

**2. 求最短路径的迪杰斯特拉算法(Dijkstra)**

**算法说明**：

对于$n$阶网$N=(V, E)$，Dijkstra算法按最短路径长度递增的次序求任意给定的某顶点(作为始点)到其它的$n-1$个顶点的最短路径。若需要求出全部顶点对间的最短路径，必须以每个顶点为源点应用Dijkstra算法$n$次。

首先，引入辅助向量dist[0..$n$-1]，该向量用于存储$n$-1条最短路径的长度。设始点为$v_k$，则算法结束后，dist[$i$]($i\neq k$)的值为始点$v_k$至顶点$v_i$的最短路径长度。

**初始化**：dist[$i$]=$w_{k,i}$ $i$=0, 1, 2, …, $n$-1

其中，若$v_i$邻接自$v_k$，则$w_{k,i}$为边上权值，否则$w_{k,i}=\infty$。



## 7.7 最短路径 续2

**第1步：求$n$-1个最短路径长度中的最小值以及对应路径终点**

显然，始点$v_k$到其它$n$-1个顶点的最短路径的最小值应为依附于始点$v_k$的所有边(弧)中权值的最小值，对应路径终点为该最小权值边(弧)依附的另一邻接点。故，最短的最短路径的终点下标可用下式计算。

$$j = \arg\min_i \text{dist}[i] \qquad (1)$$

(1)式中，arg表示求下标$i$，使得$i$满足条件：dist[$i$]是所有dist[·]中的最小值。

总之，若下标$j$满足(1)式，则$v_k$至$v_j$的最短路径长度为dist[$j$]，且dist[$j$]是$n$-1个最短路径中长度最短的。



## 7.7 最短路径 续3

**第2步：循环$n$-2趟($m$=1, 2, … , $n$-2)，按长度递增次序生成其它最短路径**

若视算法第1步为第0趟，记第$m$($m$=0, 1, … , $n$-2)趟生成的最短路径终点下标为$j_m$，则必须使

$$\text{dist}[j_0]<\text{dist}[j_1]<\text{dist}[j_2]<\cdots<\text{dist}[j_{n-2}] \qquad (2)$$

**问题**：对任意第$m$趟($1\le m\le n$-2)，$j_m$该怎样产生呢?

**答**：设$S$为已经产生的最短路径的终点下标集合，即$S=\{j_0, j_1, \dots, j_{m-1}\}$，则可以证明以下性质：

**性质**：满足(2)式的$j_m$($j_m\notin S$)，

或者是始点$v_k$的邻接点；

或者是$S$中某顶点(下标)的邻接点(下标)。



## 7.7 最短路径 续4

**证明**：由性质可知，第$m$趟产生的最短路径中，除路径终点(下标)外，其它顶点(下标)都含于集合$S$中。

用反证法。

假设除终点(下标) $j_m\notin S$外，第$m$趟产生的最短路径中还有其它顶点(下标) $j'\notin S$，则从始点$v_k$至顶点$v_{j'}$有一条路径$v_k\to v_{j'}$，其长度比第$m$趟产生的最短路径长度dist[$j_m$]要小。

显然，

$v_k$至$v_{j'}$的最短路径长度≤路径$v_k\to v_{j'}$的长度<dist[$j_m$]

根据式(2)，必有$j'\in S$，故假设不可能成立。

**算法**：由性质，可以统一**第1步**和**第2步**操作，得到如下Dijkstra算法。



## 7.7 最短路径 续5

**算法**：由上述性质，可以统一**第1步**和**第2步**操作为$n$-1趟循环，得到如下Dijkstra算法。

**//Dijkstra算法核心版(未保存路径中的顶点序列)**

**a**. $S=\varphi$;

**b**. dist[$i$]=$w_{k,i}$, for $i$=0, 1, 2, …, $n$-1;

**c**. for $m$=0, 1, …, $n$-2, 对任意第$m$趟循环, 执行**c1**-**c3**:

**c1**. $j = \arg\min_i \text{dist}[i]$, for $\forall i\in V$-$S$且$i\neq k$;

**c2**. $S=S\cup\{j\}$;

**c3**. for $\forall i\in V$-$S$且$i\neq k$,

若dist[$j$]+$w_{j,i}$<dist[$i$]，则令dist[$i$]=dist[$j$]+$w_{j,i}$;

## 7.7 最短路径 续6

//Dijkstra算法完整版(保存路径中的顶点序列)

**a**. $S=\varphi$;

**b**. for $i$=0, 1, 2, …, $n$-1, 执行操作**b1**-**b2**;

**b1**. dist[$i$]=$w_{k,i}$;

**b2**. 若dist[$i$]=∞, 则path[$i$]=$\varphi$, 否则 path[$i$]={$k$, $i$};

**c**. for $m$=0, 1, …, $n$-2, 执行操作**c1**-**c3**;

**c1**. $j = \arg\min_i \text{dist}[i]$, for $\forall i \in V\text{-}S$且$i \neq k$;

**c2**. $S=S\cup\{j\}$;

**c3**. for $\forall i \in V\text{-}S$且$i \neq k$,

若dist[$j$]+$w_{j,i}$<dist[$i$]，则执行**c31**-**c32**;

**c31**. dist[$i$]=dist[$j$]+$w_{j,i}$;

**c32**. path[$i$]=path[$j$]$\cup${$i$};



## 7.7 最短路径 续7

**问题与思考：**

1. 若Dijkstra算法包括求始点$v_k$至$v_k$的最短路径(即以$v_k$为始点和终点的最短环)，算法怎样修改？

**答**：去掉约束条件$i \neq k$；

循环变量$m$的循环次数由$n$-1次改为$n$次。

2. 下标集合$S$和path[$i$]怎样用C/C++语言实现？

**答**：**方法1**，标志数组实现。

$S$[0..$n$-1]: 若$S[i]$为0, $i \notin S$, 否则, $i \in S$;

path[$i$][0..$n$-1]: 若path[$i$][$j$]为0, $j \notin$path[$i$], 否则$j \in$path[$i$];

优点: 判断、设置元素属于或不属于集合操作简单;

缺点: 辅助存储空间大; $\forall i \in V\text{-}S$循环次数固定为$n$次。



## 7.7 最短路径 续8

**方法2**，用单链表存储集合元素。

优点: 辅助空间小；path[$i$]链表中顶点的连接次序可以正好是拓扑有序；$S$集合链表中顶点(最短路径的终点)的连接次序可以正好是最短路径由小至大的次序。

缺点: 判断元素是否属于集合效率低，使得$\forall i \in V\text{-}S$循环实现比较复杂；在集合中加入和删除元素的操作所需计算时间更多。

**方法3**，使用C++的ATL模板实现集合操作。

优点: 源代码简单，可读性好；

缺点: 算法空间和时间复杂度可能不够优化。



## 7.7 最短路径 续9

```
/* Dijkstra算法完整版的C++实现
   说明：使用ATL的set实现S集合；
         使用ATL的vector实现path[i]。*/
typedef enum { UDG, DG } NetTp;
#define MAX_N 最大顶点数
typedef struct
{ int n, e;     //实际顶点数和边(弧)数
  NetTp tp;  //UDG:无向网, DG:有向网
  double w[MAX_N][MAX_N];
}ALNet;    //网的邻接矩阵类型
#define INFINITY  1E100   //定义∞
double dist[MAX_N]; vector<int>  path[MAX_N];
```



## 7.7 最短路径 续10

```
void SP(ALNet &G,int k,double dist[],vector<int> path[])
//k为始点；计算结果存于dist和path数组
{ int i, j, m;          //#include <vector>即可使用vector
  //step a
  set<int> S;           //#include <set>即可使用set
  //step b
  for(i=0; i<G.n; i++)
  { dist[i]=G.w[k][i];          //step b1
    if(dist[i]!=INFINITY) // step b2
      { path[i].push_back(k);path[i].push_back(i); }
  }
  //step c
```



## 7.7 最短路径 续11

```
  for(m=0; m<G.n-1; m++)
  { //step c1
    for(j=0; j<G.n; j++)
      if(j!=k&&S.find(j)==S.end())break;
    for(i=j+1; i<G.n; i++)
      if(i!=k&&dist[i]<dist[j]&&S.find(i)==S.end())
        j=i;
    //step c2
    S.insert(j);
    //step c3
```

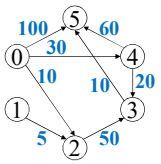
## 7.7 最短路径 续12

```
    for(i=0; i<G.n; i++)
      if(i!=k&&S.find(i)==S.end())
        { double d=dist[j]+G.w[j][i];
          if(dist[i]>d)
            { dist[i]=d;          //step c31
              path[i]=path[j]; //step c32
              path[i].push_back(i);
            }
        }
    } //for m 循环体结束
}
```



## 7.7 最短路径 续13

**例1** Dijkstra算法动态运行过程(从顶点**0**出发)。




$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

| i / m | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | ∞ {} | 10 {02} | ∞ {} | 30 {04} | 100 {05} |
| 1 | ∞ {} | | 60 {023} | 30 {04} | 100 {05} |
| 2 | ∞ {} | | 50 {043} | | 90 {045} |
| 3 | ∞ {} | | | | 60 {0435} |
| 4 | ∞ {} | | | | |



## 7.7 最短路径 续14

**3. 求最短路径的弗洛依德(Floyd)算法**

**算法说明**：

对于$n$阶网$N=(V, E)$，Floyd算法由权值邻接矩阵一次性求出各对顶点间的最短路径。

设顶点集合$V$中$n$个顶点的序号为0, 1, 2, …, $n$-1, 定义$n\times n$阶方阵序列，$A^{(-1)}, A^{(0)}, A^{(1)}, \dots, A^{(n-1)}$如下。

记矩阵$A^{(k)}$的$i$行$j$列元素为$a_{ij}^{(k)}$ ($k$=-1, 0, … , $n$-1)，若令集合$P$表示顶点$v_i$至$v_j$满足如下条件的所有路径组成的集合：路径或者是弧<$v_i$, $v_j$>，或者从$v_i$出发途经其它顶点(称为中间顶点)到达$v_j$，则中间顶点的序号不大于$k$。定义$a_{ij}^{(k)}$为$P$集合中所有路径长度的最小值。即$a_{ij}^{(k)}$为$v_i$至$v_j$中间顶点序号不大于$k$的最短路径长度。



## 7.7 最短路径 续15

**算法描述**：

**//Floyd算法核心版(未存储最短路径上的顶点序列)**

**a**. $a_{ij}^{(-1)} = w_{ij}$, for 所有$i, j$ **//$w_{ij}$为弧<$v_i$, $v_j$>权值**

**b**. for $k$=0, 1, 2, … , $n$-1, 执行以下操作:

$$a_{ij}^{(k)} = \min\left\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right\}, \text{ for 所有} i, j$$

算法按中间顶点$v_0, v_1, \dots, v_{n-1}$的次序经过$n$轮迭代后，$a_{ij}^{(n-1)}$即为$v_i$至$v_j$的最短路径长度。

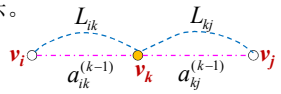
**证明**：$a_{ij}^{(k)} = \min\left\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right\}$

(1) 若$v_i$至$v_j$中间顶点序号不大于$k$的路径集合中不存在途经顶点$v_k$的路径，则$a_{ij}^{(k)} = a_{ij}^{(k-1)}$；



## 7.7 最短路径 续16

(2) $v_i$至$v_j$中间顶点序号不大于$k$的路径集合中存在途经顶点$v_k$的路径，则可证明$v_i$至$v_j$中间顶点序号不大于$k$且途经$v_k$的最短路径是$v_i$至$v_k$中间顶点序号不大于$k$-1的最短路径连接$v_k$至$v_j$中间顶点序号不大于$k$-1的最短路径，如下图粉红色点划线所示。




**反证法**：假设$v_i$至$v_j$中间顶点序号不大于$k$且途经$v_k$的最短路径为$\boldsymbol{v_i\cdots v_k\cdots v_j}$，如上图中**蓝色虚线**所示。显然，$v_k$在该最短路径上最多出现一次。记该路径前段$v_i\cdots v_k$的长度为$L_{ik}$，后段$v_k\cdots v_j$的长度为$L_{kj}$，则



## 7.7 最短路径 续17

$$L_{ik} + L_{kj} < a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$$

即假设**粉红色路径**不是最短，**蓝色路径**才是最短，以下证明这不可能。

因为$v_i\cdots v_k$(**蓝色虚线**)也是$v_i$至$v_k$中间顶点序号不超过$k$-1的一条路径，由$A^{(k-1)}$的定义知 $L_{ik} \ge a_{ik}^{(k-1)}$，类似地有$L_{kj} \ge a_{kj}^{(k-1)}$，故$L_{ik} + L_{kj} \ge a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$，即假设不可能成立。

综合(1), (2), 我们有

$$a_{ij}^{(k)} = \min\left\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right\}$$

## 7.7 最短路径 续18

**//Floyd算法完整版(存储最短路径上的顶点序列)**

**a**. $a_{ij}^{(-1)} = w_{ij}$, for 所有$i, j$ //$w_{ij}$为弧<$v_i$, $v_j$>权值

**b**. $p_{ij}^{(-1)} = \{i, j\}$, for 所有$i, j$

**c**. for $k$=0, 1, 2, … , $n$-1,

for 所有$i, j$,

若 $a_{ij}^{(k-1)} > a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$, 则执行**c1**和**c2**;

**c1**. $a_{ij}^{(k)} = a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$

**c2**. $p_{ij}^{(k)} = p_{ik}^{(k-1)} \cup p_{kj}^{(k-1)}$

算法运行结束后，$a_{ij}^{(n-1)}$ 和 $p_{ij}^{(n-1)}$ 分别为顶点$v_i$至$v_j$的最短路径长度和该最短路径上的顶点集合。



## 7.7 最短路径 续19

**问题与思考：**

Floyd算法实现$A^{(k)}$矩阵迭代时，只需设置1个矩阵迭代还是需设两个矩阵实现迭代？

两个矩阵迭代方法如下所述：

若**A**为$A^{(k-1)}$, 每个 $a_{ij}^{(k)}$先存入与**A**相同大小的矩阵**B**，直到$i, j$循环结束(即计算过程中**A**矩阵不变)，然后将矩阵**B**拷贝给矩阵**A**，使**A**矩阵更新为$A^{(k)}$。

**答**：只需设1个矩阵迭代，这是因为执行如下迭代操作时，即使 $a_{ik}^{(k-1)}$和 $a_{kj}^{(k-1)}$已更新为 $a_{ik}^{(k)}$和 $a_{kj}^{(k)}$, $a_{ij}^{(k)}$计算结果不变。这是因为 $a_{ik}^{(k)} \equiv a_{ik}^{(k-1)}$ 且 $a_{kj}^{(k)} \equiv a_{kj}^{(k-1)}$。

**迭代式**: $a_{ij}^{(k)} = a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$



## 7.7 最短路径 续20

**证明**：$a_{ik}^{(k)} \equiv a_{ik}^{(k-1)}$ 且 $a_{kj}^{(k)} \equiv a_{kj}^{(k-1)}$

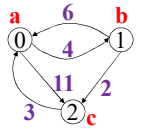
若$v_i$至$v_k$中间顶点序号不大于$k$的某路径的中间顶点至少包含一次$v_k$，即该路径形如$v_i \cdots v_k \cdots v_k$，(其中$v_k \cdots v_k$为环)，显然，该路径的长度一定大于$v_i \cdots v_k$中间顶点不大于$k$-1的最短路径长度，故 $a_{ik}^{(k)} \equiv a_{ik}^{(k-1)}$;

类似地，可证明 $a_{kj}^{(k)} \equiv a_{kj}^{(k-1)}$。



## 7.7 最短路径 续21

**例2** Floyd算法动态运行过程




$$A^{(-1)} = \begin{bmatrix} \infty & 4 & 11 \\ 6 & \infty & 2 \\ 3 & \infty & \infty \end{bmatrix} \quad P^{(-1)} = \begin{bmatrix} aa & ab & ac \\ ba & bb & bc \\ ca & cb & cc \end{bmatrix}$$

$$A^{(0)} = \begin{bmatrix} \infty & 4 & 11 \\ 6 & 10 & 2 \\ 3 & 7 & 14 \end{bmatrix} \quad A^{(1)} = \begin{bmatrix} 10 & 4 & 6 \\ 6 & 10 & 2 \\ 3 & 7 & 9 \end{bmatrix} \quad A^{(2)} = \begin{bmatrix} 9 & 4 & 6 \\ 5 & 9 & 2 \\ 3 & 7 & 9 \end{bmatrix}$$

$$P^{(0)} = \begin{bmatrix} aa & ab & ac \\ ba & bab & bc \\ ca & cab & cac \end{bmatrix} \quad P^{(1)} = \begin{bmatrix} aba & ab & abc \\ ba & bab & bc \\ ca & cab & cabc \end{bmatrix} \quad P^{(2)} = \begin{bmatrix} abca & ab & abc \\ bca & bcab & bc \\ ca & cab & cabc \end{bmatrix}$$



## 7.7 最短路径 续22完

**问题与思考：**

若要Floyd算法避免求环的最短长度，应该怎么办？

**答**：只需令$A^{(-1)}$矩阵的主对角线元素全为0，含义相当于顶点自己到自己的最短路径长度为0。

**4. 算法时间复杂度**

Dijkstra算法：$O(n^2)$，求所有顶点对 $O(n^3)$

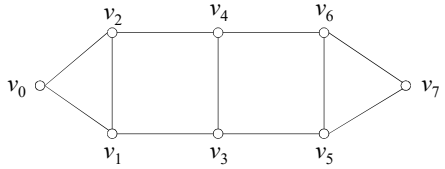
Floyd算法：$O(n^3)$



## 第7章 作业

**第1次作业**

1. 已知某无向图如下图所示。画出该图的多重邻接表存储结构示意图。根据该存储结构，写出从顶点$v_0$出发，深度和宽度优先遍历顶点访问次序。



## 第7章 作业 续1

2. 判断无向图是否有环。算法提要：深度优先遍历过程中，访问某顶点后，该顶点的邻接点中有已访问的顶点且该已访问邻接点不是该顶点的上一级递归出发顶点(即存在回边)，则有环。
3. 编程题: 建立无向图邻接表存储结构，输出深度和宽度优先遍历顶点访问次序。
4. 编程题: 建立连通网邻接矩阵存储结构，实现Prim算法，输出最小生成树的各条边及边上的权重。



## 第7章 作业 续2

**第2次作业**

1. 算法设计*：已知AOE网络的邻接表存储结构G，ve[]和vl[]值已全部求取，写出算法，输出所有关键路径。要求每条关键路径用源点至汇点的顶点序列表示(如本教案7.6例2所示)。
2. 编程题：建立AOE网络存储结构，计算并输出ve[]和vl[]；验证1.算法的正确性*。



## 第7章 作业 续3完

**第3次作业** (课程设计作业)
题目：北京至其它省会城市的最短路径
任务要求描述：
1. 建立教材图7.33(中国省会城市交通网)的数据文件，构建邻接矩阵存储结构。以北京为始点，求出北京到其它城市的最短路径。结果输出到字符文件中保存。
2. 每人完成课程设计报告1份(需打印)，内容包括：
(1) 封面(课程设计题目, 班级, 姓名, 学号, 日期)
(2) 课程设计任务描述 (3)算法要点描述
(4) 源程序代码(注释不低于1/4总字符数)
(5) 输出结果 (6)问题与讨论

# 第8章 查找





## 8.1 查找的基本概念

**1. 查找表**

由同一类型的数据元素(数据对象)组成的集合。查找表的存储结构一般采用**数组**、**树**或**散列表**。

**2. 静态与动态查找表**

查询某个特点元素是否在查找表中或者从查找表中提取某个特定元素的属性值称为**静态查找表**；若在查找表中插入元素或者删除元素，称为**动态查找表**。

从信息处理的角度看，静态查找表就是对查找表进行“**只读**”型查找；动态查找表则对查找表进行“**加工**”型查找。



## 8.1 查找的基本概念 续1

**3. 关键字**

数据元素(对象)的某个属性值称为关键字。若关键字可以唯一标识一个数据元素，称为**主关键字**，否则称为**次关键字**。几个次关键字组成的关键字组也可以构成主关键字。

**关键字的大小**：查找操作要求数据元素(对象)的关键字应可以比较大小(至少应该可以比较“相等”)。对于数值型关键字(C语言的整型、实型、指针类型)，比较大小操作可以直接由高级语言的关系运算符(C语言的<, <=, >, >=, ==, !=)实现；对于非数值型关键字，比如：字符串等，关键字比较大小的操作需定义专门的函数实现(C语言的strcmp函数)。



## 8.1 查找的基本概念 续2

**通用的两个关键字$k_1$, $k_2$比较大小函数形如**：

| | | | |
|---|---|---|---|
| LT($k_1$, $k_2$) | //< | GT($k_1$, $k_2$) | //> |
| LQ($k_1$, $k_2$) | //<= | GQ($k_1$, $k_2$) | //>= |
| EQ($k_1$, $k_2$) | //== | UEQ($k_1$, $k_2$) | //!= |

某些关键字类型可能无法定义出LT, GT操作，但一般都可以定义出EQ操作。

C++可以使用关系运算符重载代替上述比较大小函数，从而使非数值类型关键字比较大小可以直接使用6个关系运算符实现。

为了表述简单并增强可读性，本电子教案不论何种类型关键字，均使用C语言的6个关系运算符来表示关键字比较大小。



## 8.1 查找的基本概念 续3

**4. 查找与查找结果**

根据给定关键字查找数据元素(对象)在查找表中的存储位置或者实现数据元素的插入与删除称为**查找**。查找的结果分为**查找成功**与**查找失败(查找不成功)**两种情况。

**5. 查找操作的时间复杂度分析方法**

(1) $ASL_{成功}$分析

查找成功时的时间复杂度由ASL(Average Search Length, 平均查找长度)决定，对于含有$n$个数据元素的查找表$\{a_0, a_1, \ldots, a_{n-1}\}$，成功查找的ASL定义为

$$ASL_{成功}=\sum_{i=0}^{n-1} p_i c_i$$

**$p_i$表示查找$a_i$的概率，**
**$c_i$为成功找到$a_i$的关键字比较次数。**



## 8.1 查找的基本概念 续4完

(2) $ASL_{失败}$分析

查找失败时平均关键字比较次数一般需要根据查找表的存储结构具体分析。

(3) 综合查找成功与失败的ASL分析

$ASL=p_{成功}ASL_{成功}+p_{失败}ASL_{失败}$

**其中，$p_{成功}$与$p_{失败}$分别表示查找成功与失败的概率($p_{成功}+p_{失败}=1$)，有时候可根据经验估计它们的值。**

在教材和本电子教案中，一般只考虑(1), (2)。

## 8.2 静态查找表

**1. 顺序表的查找**

**(1)** 顺序表的存储结构

```
//用数组实现顺序查找表存储结构
#define MAX_N 最大元素个数
typedef struct
{ KeyTp  K;  //某关键字
  …          //其它属性
}ElemTp;    //数据元素类型
typedef struct { ElemTp elem[MAX_N];
                 int n; //实际元素个数
               }SqList;  //数组实现的顺序查找表类型
```



## 8.2 静态查找表 续1

**(2)** 顺序查找算法

```
int sqsearch(SqList &R, KeyTp key)
{ for(i=0; i<R.n; i++) if(R.elem[i].K==key) break;
  if(i>=R.n) return -1;  //返回-1表示查找失败
  return i;    //查找成功，返回元素下标
}
//用监视哨改进上述算法(避免每次循环判断i<R.n)
int sqsearch1(SqList &R, KeyTp key)
{ R.elem[R.n]=key;  //设置监视哨，要求R.n<MAX_N
  for(i=0; R.elem[i].K!=key; i++);
  if(i==R.n) return -1;
  return i; }
```



## 8.2 静态查找表 续2

对上述顺序查找，若每个元素查找概率相同，则

$$ASL_{成功}=\sum_{i=0}^{n-1}p_i c_i=\frac{1}{n}\sum_{i=0}^{n-1}(i+1)=\frac{n+1}{2}$$

查找失败时，有

$$ASL_{失败}=n$$

顺序查找同样适用于单链表，$ASL_{成功}$和$ASL_{失败}$与数组存储时相同，但查找成功时，应返回元素结点的指针，查找失败时应返回零指针NULL。



## 8.2 静态查找表 续3

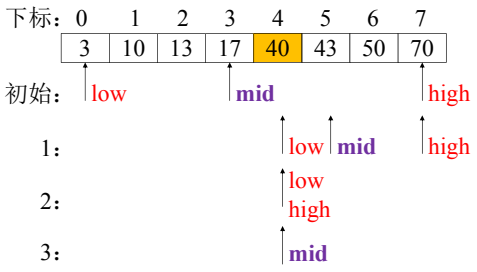
**2. 有序表的查找**

```
//对分(二分、折半、拆半)查找算法
int BinarySearch(SqList &R, KeyTp key)
{ low=0; high=R.n-1;  //R.elem[·]按K域升序存储
  while (low<=high)
  { mid=(low+high)/2;
    if(R.elem[mid].K==key)  return mid;
    if(key<R.elem[mid].K)  high=mid-1;else  low=mid+1;
  }
  return -1;  //查找失败
}
```



## 8.2 静态查找表 续4

**例1** 对分查找运态运行过程(key=40)

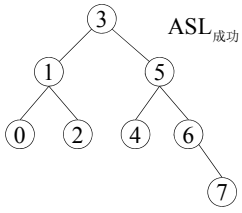





## 8.2 静态查找表 续5

**例2** 若8元升序数组每个元素等概率查找，分析对分查找的平均查找长度。

**解**：设8个元素按升序排列为$a_0, a_1, \ldots, a_7$，则对分查找每次循环比较的元素下标可以表示为如下二叉判定树：




$$ASL_{成功}=\frac{1}{8}(1+2\cdot 2+3\cdot 4+4\cdot 1)=2.625$$

根结点为mid位置元素；
左子树为low至mid-1元素；
右子树为mid+1至high元素。

## 8.2 静态查找表 续6

**对分查找的时间复杂度：**

对于$n$个有序存储元素，二叉判定树的最大深度为

$$\lfloor \log_2 n \rfloor + 1$$

这是查找成功或失败时所需的最大循环次数。

此外，可以证明，当$n$>50时，

$$ASL_{成功} \approx \log_2(n+1) - 1$$



## 8.2 静态查找表 续7

**3. 索引顺序表的查找-分块查找**

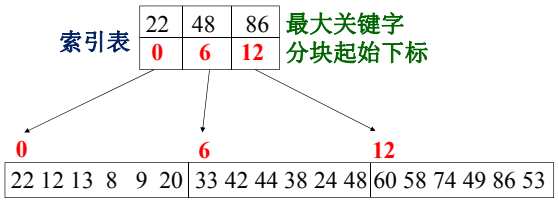
将顺序表中元素按元素下标顺序均匀分块，每个分块称为子表。子表内部元素不要求按关键字有序存储，但子表与子表之间必须是**分块有序**的，所谓分块有序，是指对任意两个子表$a$, $b$，或者$a$中所有元素关键字小于$b$中所有元素关键字，或者或者$b$中所有元素关键字小于$a$中所有元素关键字。

建立一个**关键字表**(也称**索引表**)，存放每个子表中的最大(或最小)关键字值以及对应子表在顺序表中的起始下标。



## 8.2 静态查找表 续8

**例如**，以下18个元素组成的顺序表分3块分块有序，则其索引表如图所示。






## 8.2 静态查找表 续9

**索引顺序的查找方法：**

第1步：先在索引表中对分或顺序查找关键字所在块；

第2步：在分块中顺序查找关键字。

**索引顺序表查找的时间复杂度：**

设元素总数为$n$，分块长度为$s$，则分块数为$\lceil n/s \rceil$。

(1) 索引表采用对分查找

$$ASL_{成功} = \log_2\left(\frac{n}{s}+1\right) - 1 + \frac{s+1}{2} \approx \log_2\left(\frac{n}{s}+1\right) + \frac{s}{2}$$

(2) 索引表采用顺序查找

$$ASL_{成功} = \frac{n/s+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s}+s\right)+1$$



## 8.2 静态查找表 续10完

**例3** 若索引顺序表长度为$n$，索引表和每个分块内部均采用顺序查找，求最佳分块长度。

**解**：设分块长度为$s$，则

$$ASL_{成功} = \frac{n/s+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s}+s\right)+1$$

当$n/s$=$s$时，上式取得最小值。

即最佳分块长度(使ASL最小的分块长度)$s$满足

$s^2 = n \Rightarrow s = \sqrt{n}$ 。



## 8.3 动态查找表

**1. 二叉排序树**

**(1) 二叉排序树的概念**

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

a. 若它的左子树不空，则左子树上所有结点的关键字值小于根结点的值；

b. 若它的右子树不空，则右子树上所有结点的关键字值大于(或等于)根结点的值；

c. 它的左、右子树也分别是二叉排序树。

**(2) 二叉排序树的性质**

中序遍历二叉排序树，所得结点访问次序为结点关键字值的递增有序序列。

## 8.3 动态查找表 续1

**(3) 二叉排序树的C语言数据类型定义**

```
typedef struct node
{ KeyTp K;    //关键字
  ElemTp data; //数据域
  struct node *lchild, *rchild; //左、右儿子指针
}BSTNode, *BST;
```

**(4) 二叉排序树的查找算法**

```
BST search(BST bt, KeyTp key)   //非递归算法
{ while(bt&&bt->K!=key)
     if(key<bt->K) bt=bt->lchild; else bt=bt->rchild;
  return bt; //返回NULL表示查找失败
}
```



## 8.3 动态查找表 续2

```
BST search1(BST bt, KeyTp key)  //递归算法
{ if(bt==NULL||bt->K==key) return bt;
  if(key<bt->K) return search1(bt->lchild, key);
  return search1(bt->rchild, key);
}
```

**(5) 二叉排序树插入结点算法**

```
void insert(BST &bt, BST p)
{ parent=NULL; pt=bt;
  while(pt)
  { parent=pt;
    if(p->K<pt->K) pt=pt->lchild; else pt=pt->rchild;
  }
```



## 8.3 动态查找表 续3

```
  if(parent)
     if(p->K<parent->K) parent->lchild=p;
       else parent->rhcild=p;
   else bt=p;  //空树插入第一个结点*p
}
```

**二叉排序树的建立方法：**

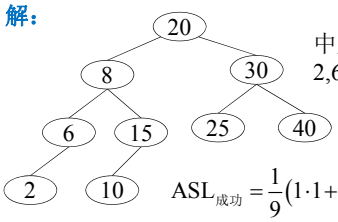
```
BST CrtBst(int n) //从空树开始，反复插入新结点
{ bt=NULL;
  for(i=0; i<n; i++)
  { read(Ki); p=new BSTNode(Ki, NULL, NULL);
    inseart(bt, p);
  } return bt; }
```



## 8.3 动态查找表 续4

**例1** 9个关键字输入次序为20, 8, 30, 15, 6, 40, 10, 25, 2。按此输入次序构造二叉排序树；写出中序遍历次序；若每个元素等概率查找，试计算成功查找时的ASL。

**解：**




中序遍历次序：
2,6,8,10,15,20,25,30,40

$$ASL_{成功}=\frac{1}{9}(1\cdot1+2\cdot2+3\cdot4+4\cdot2)=\frac{25}{9}$$



## 8.3 动态查找表 续5

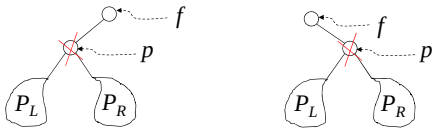
**(6) 二叉排序树删除结点**

**问题**：删除结点后，怎样使剩余结点仍然构成一棵二叉排序树？

**分析**：

a. 被删除结点为叶子结点，则直接删除；

b. 被删除结点有子树，设被删除结点的指针为***p***，其双亲结点指针为***f***, 左、右子树分别为$P_L$和$P_R$。






## 8.3 动态查找表 续6

删除结点*p之前的中序遍历次序为：

*p为*f的左子树：$P_L$, ***p***, $P_R$, ***f***

*p为*f的右子树：***f***, $P_L$, ***p***, $P_R$

删除*p后中序遍历次序应为：$P_L$, $P_R$, ***f*** 或 ***f***, $P_L$, $P_R$

**情况b1**: *p只有一棵子树

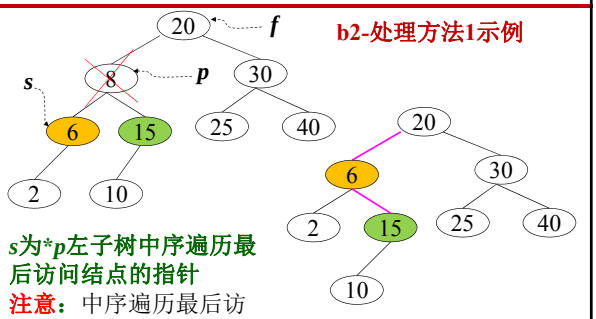
若$P_L$为空，只需删除*p, 然后将$P_R$的根结点与*f连接；

若$P_R$为空，只需删除*p, 然后将$P_L$的根结点与*f连接。

**情况b2**: *p的左右子树均不为空

**b2-处理方法1**: 将$P_R$连接为$P_L$的中序遍历最后访问结点的右子树， $P_L$的根结点与*f连接。

## 8.3 动态查找表 续7




b2-处理方法1示例

s为*p左子树中序遍历最后访问结点的指针

注意：中序遍历最后访问结点的右子树必为空

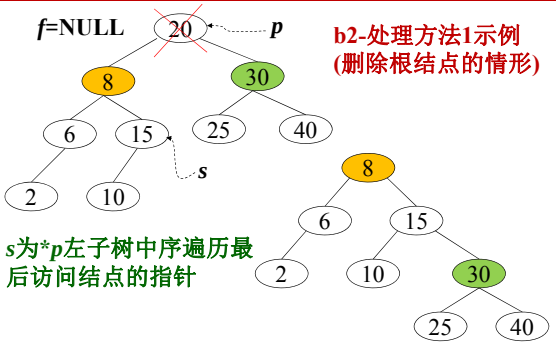


## 8.3 动态查找表 续8

```
//b2-处理方法1算法(已知bt, p和f)
pl=p->lchild; pr=p->rchild;
delete p; //删除*p
//求*p左子树中序遍历最后访问结点指针s
s=pl; while(s->rchild) s=s->rchild;
//*pl与*f连接, 若f为NULL, 则*pl成为*bt
if(f==NULL) bt=pl;
else if(f->lchild==p) f->lchild=pl;
        else f->rchild=pl;
//*pr连接至*s右边
s->rchild=pr;
```



## 8.3 动态查找表 续9




b2-处理方法1示例
(删除根结点的情形)

s为*p左子树中序遍历最后访问结点的指针



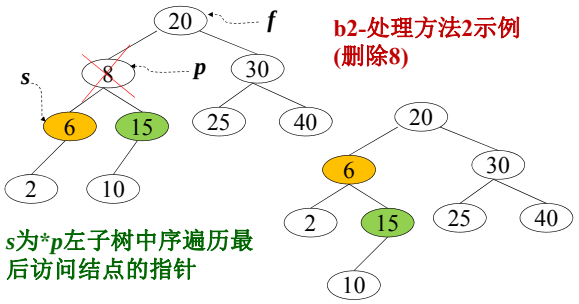
## 8.3 动态查找表 续10

b2-处理方法2: 设$P_L$的中序遍历最后访问结点地址为*s*，删除*p; 在$P_L$中删除*s(使$P_L$仍为二叉排序树, *s不是真删，只是使*s成为孤立结点); *s的左、右指针分别指向$P_L$与$P_R$的根结点; 用*s代替*p位置并与*f连接。

```
//b2-处理方法2算法(已知bt, p和f)
pl=p->lchild; pr=p->rchild; delete p; //删除*p
ps=NULL; s=pl;
while(s->rchild) {ps=s; s=s->rchild; } //求s和其双亲ps
if(!ps) pl=s->lchild; else ps->rchild=s->lchild;  //孤立*s
s->lchild=pl; s->rchild=pr;
if(!f) bt=s;
else if(f->lchild==p) f->lchild=s; else f->rchild=s;
```



## 8.3 动态查找表 续11

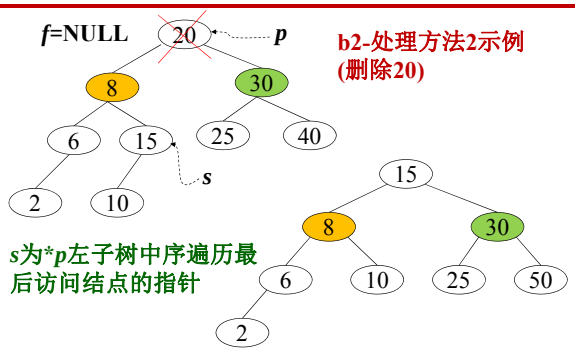



b2-处理方法2示例
(删除8)

s为*p左子树中序遍历最后访问结点的指针



## 8.3 动态查找表 续12




b2-处理方法2示例
(删除20)

s为*p左子树中序遍历最后访问结点的指针

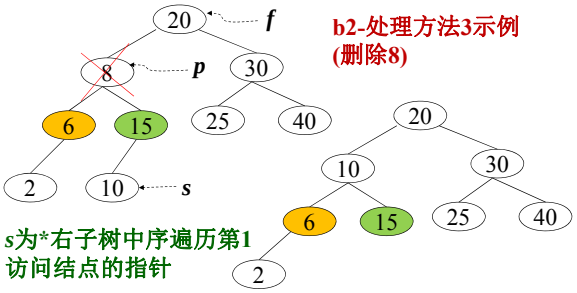
## 8.3 动态查找表 续13

**b2-处理方法3**: 设$P_R$的中序遍历第1访问结点地址为***s***，删除**p*; 在$P_R$中删除**s*(使$P_R$仍为二叉排序树, **s*不是真删，只是使**s*成为孤立结点); **s*的左、右指针分别指向$P_R$与$P_L$的根结点; 用**s*代替**p*位置并与**f*连接。

```
//b2-处理方法3算法(已知bt, p和f)
pl=p->lchild; pr=p->rchild; delete p; //删除*p
ps=NULL; s=pr;
while(s->lchild) {ps=s; s=s->lchild; } //求s和其双亲ps
if(!ps) pr=s->rchild; else ps->lchild=s->rchild;  //孤立*s
s->lchild=pl; s->rchild=pr;
if(!f) bt=s;
else if(f->lchild==p) f->lchild=s; else f->rchild=s;
```



## 8.3 动态查找表 续14

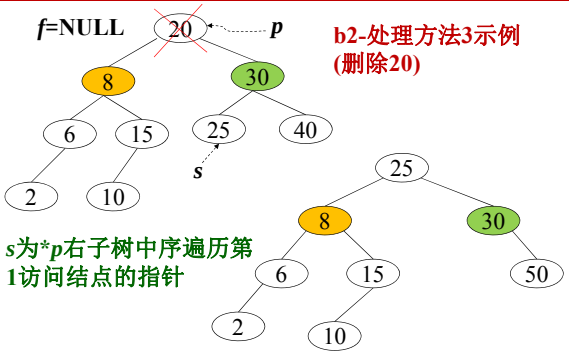



b2-处理方法3示例(删除8)

s为*右子树中序遍历第1访问结点的指针



## 8.3 动态查找表 续15




b2-处理方法3示例(删除20)

s为*p右子树中序遍历第1访问结点的指针



## 8.3 动态查找表 续16

**例2** 已知某二叉排序树根结点地址bt，若该二叉排序树中无重复键值结点，给定键值key，给出算法删除键值为key的结点，使删除结点后的二叉树仍为二叉排序树。

```
//采用b2-处理方法2
void erase(BST &bt, KeyTp key)
{  //查找key
   f=NULL; p=bt;
   while(p&&key!=p->K)
   { f=p;
     if(key<p->K) p=p->lchild; else p=p->rchild;
   }
   if(!p)  return;  //查找失败，直接结束
```



## 8.3 动态查找表 续17

```
   //b2-处理方法2
   pl=p->lchild; pr=p->rchild;
  if(pl&&pr)
   { ps=NULL; s=pl;
     while(s->rchild) {ps=s; s=s->rchild; }
     if(!ps) pl=s->lchild; else ps->rchild=s->lchild;
     s->lchild=pl; s->rchild=pr;
   } else if(pl) s=pl; else s=pr;
   if(!f) bt=s;
     else if(f->lchild==p) f->lchild=s; else f->rchild=s;
   delete p; //删除*p
}
```



## 8.3 动态查找表 续18

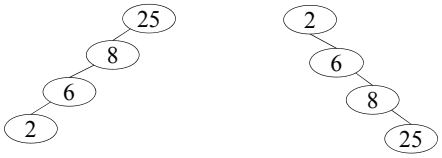
**(7) 二叉排序树查找算法的时间复杂度**

$ASL_{成功}=1+4\log_2 n$

**最坏情况**：二叉排序树为单链，等效于顺序查找

$ASL_{成功(最坏)}=(n+1)/2$

如：




**理想情况**：二叉排序树是平衡的，其$ASL_{成功}$与对分查找近似。

## 8.3 动态查找表 续19

**2. 平衡的二叉排序树**

只介绍平衡的二叉排序树**插入结点**使二叉排序树仍为平衡二叉树的方法。

**数据结构**：需为每个结点增设bf域，存储该结点为根的子树的平衡因子。

设由于在二叉排序树上插入结点而失去平衡的最小子树的根结点指针为a(即a指向离插入结点最近，且平衡因子绝对值超过1的祖先结点)。则只要将以*a为根的子树调整为平衡因子=0且高度与原*a为根的子树相同的平衡的二叉排序树即可。



## 8.3 动态查找表 续20

**(1) 平衡二叉排序树插入结点算法**

设待插入的新结点地址为s，则

a. 在查找s结点的插入位置的过程中，记下离s结点最近且平衡因子不等于0的结点，令指针a指向该结点；
b. 修改自a至s路径上所有结点的平衡因子；
c. 判断以*a为根的子树是否失衡，若失衡，根据4种旋转类型进行调整，使*a的平衡因子=0且*a子树高度与原*a子树高度相同。



## 8.3 动态查找表 续21

```
//step a.的C语言算法
a=bt; p=bt; f=NULL;s->bf=0;
while(p)
  { if(p->bf!=0) a=p;
    f=p;
    if(s->K<p->K) p=p->lchild; else p=p->rchild;
  }
if(!f) bt=s;
   else if(s->K<f->K) f->lchild=s; else f->rchild=s;
```

**注意**：若从根结点*bt至插入位置*f路径上所有结点的bf域为0(比如：二叉排序树在插入结点前是满二叉树)，则a为bt。



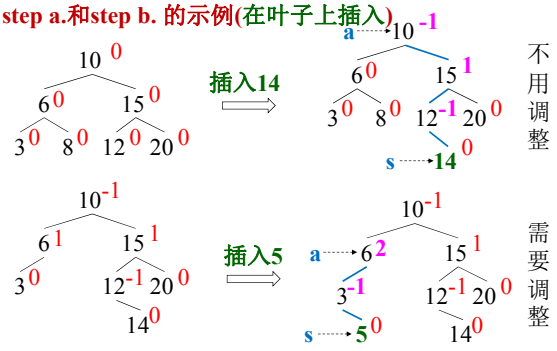
## 8.3 动态查找表 续22

```
//step b.的C语言算法
//注意: *a至*s路径上除*a外, 其余结点的bf域均为0
if(a)   //a为NULL说明是空树插入第1个结点
{ if(s->K<a->K)
    { p=a->lchild; a->bf+=1; }//是在*a的左子树上插入
  else
    { p=a->rchild; a->bf-=1; } //是在*a的右子树上插入
   while(p!=s)
      if(s->K<p->K) { p->bf=1; p=p->lchild; }
         else { p->bf=-1; p=p->rchild; }
   //以下为step c.
}
```



## 8.3 动态查找表 续23

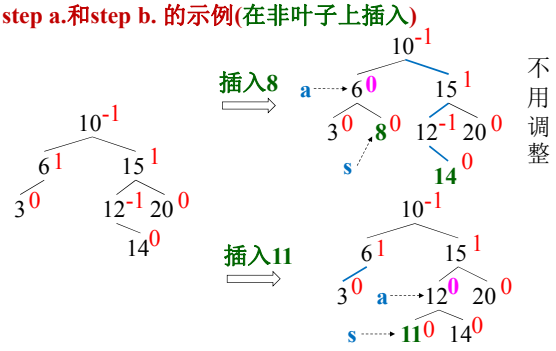
**step a.和step b. 的示例(在叶子上插入)**






## 8.3 动态查找表 续24

**step a.和step b. 的示例(在非叶子上插入)**



## 8.3 动态查找表 续25

**(2) 平衡二叉排序树查入结点算法之step c.**

在执行算法step b.之前，若a->bf!=0(若a->bf原值为0，则a==bt，执行step b.后，a->bf=+1或-1，不需要调整)，则执行step b.后，a->bf只有三种情形：

**a->bf=2** (需调整)

**a->bf=-2** (需调整)

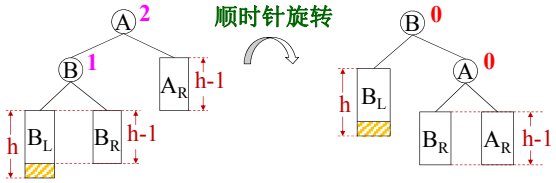
**a->bf=0** (说明*a子树的高度未改变，即原a->bf=1时，是在*a的右子树上插入了结点，或者原a->bf=-1时，是在*a的左子树上插入了结点)

对于a->bf=2或-2的情形，分**4种类型**进行调整，使以*a为根的子树高度在插入结点前后不变，且*a子树仍为平衡二叉排序树，且*a的平衡因子为0。



## 8.3 动态查找表 续26

**LL型**：执行算法step b.后，a->bf域变为2(原a->bf=1则新结点*s一定是插入到*a的左子树中)，记a所指结点为A，A的左儿子记为B，*s是插入到B的左子树中。




插入结点

**中序遍历次序(旋转前后相同)**: $B_L$, B, $B_R$, A, $A_R$



## 8.3 动态查找表 续26

**问题**：A的左儿子B一定存在吗？

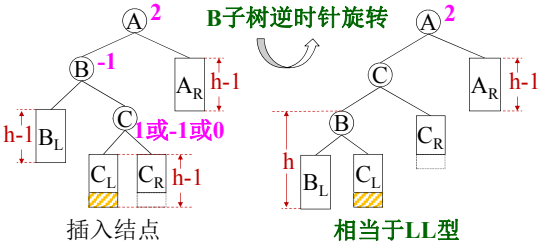
**答**：一定存在，因为*s是插入到原来的叶子上的，则*a至*s路径上至少有一个原bf域为0的结点(即*a不可能是*s的双亲)；又因为*s是插入到A的左子树上的，故*s左儿子B一定存在。

```
//LL型调整算法
b=a->lchild;
a->lchild=b->rchild; a->bf=0;
b->rchild=a; b->bf=0;
//以下实现*b与原*a的双亲结点连接
if(!fa) bt=b;   //fa为*a的双亲结点地址
  else if(b->K<fa->K) fa->lchild=b; else fa->rchild=b;
```



## 8.3 动态查找表 续27

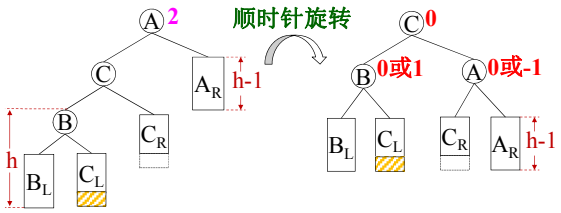
**LR型**：执行算法step b.后，a->bf域变为2(原a->bf=1则新结点*s一定是插入到*a的左子树中)，记a所指结点为A，A的左儿子记为B，*s是插入到B的右子树中。




插入结点　　相当于LL型



## 8.3 动态查找表 续28




插入结点*s并执行算法step b.后，若

**c->bf=0**: 则B为叶子且C为新插入结点且A的右子树$A_R$必为空(高为0), 调整后b->bf=a->bf=0;

**c->bf=1**: 则调整后b->bf=0, a->bf=-1;

**c->bf=-1**: 则调整后b->bf=1, a->bf=0;

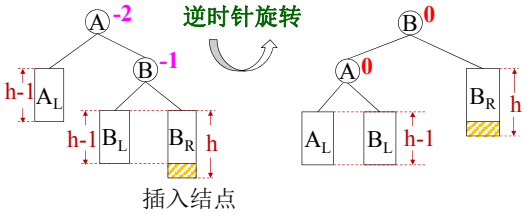


## 8.3 动态查找表 续29

```
//LR型调整算法
b=a->lchild; c=b->rchild;
a->lchild=c->rchild;
b->rchild=c->lchild;
c->lchild=b; c->rchild=a;
switch(c->bf)
{ case 0: a->bf=b->bf=0;break;
  case 1: a->bf=-1; b->bf=0; break;
  case -1: a->bf=0; b->bf=1; break;
}
c->bf=0;
//以下实现*c与原*a的双亲结点连接(省略)
```

## 8.3 动态查找表 续30

**RR型**：执行算法step b.后，a->bf域变为-2(原a->bf=-1则新结点*s一定是插入到*a的右子树中)，记a所指结点为A，A的右儿子记为B，*s是插入到B的右子树中。





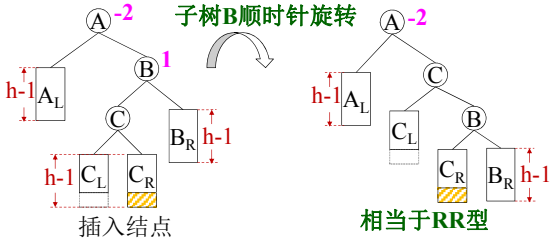

## 8.3 动态查找表 续31

```
//RR型调整算法
b=a->rchild;
a->rchild=b->lchild; a->bf=0;
b->lchild=a; b->bf=0;
//以下实现*b与原*a的双亲结点连接(略)
```



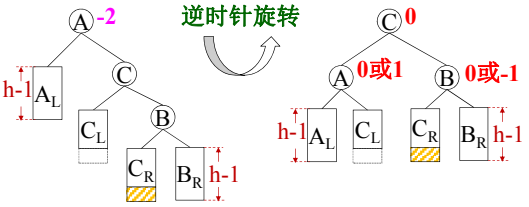
## 8.3 动态查找表 续32

**RL型**：执行算法step b.后，a->bf域变为-2(原a->bf=-1则新结点*s一定是插入到*a的右子树中)，记a所指结点为A，A的右儿子记为B，*s是插入到B的左子树中。






## 8.3 动态查找表 续33




插入结点*s并执行算法step b.后，若
**c->bf=0**: 则B为叶子且C为新插入结点且A的左子树$A_R$必为空(高为0), 调整后b->bf=a->bf=0;
**c->bf=1**: 则调整后b->bf=-1, a->bf=0;
**c->bf=-1**: 则调整后b->bf=0, a->bf=1;



## 8.3 动态查找表 续34

```
//RL型调整算法
b=a->rchild; c=b->lchild;
a->rchild=c->lchild;
b->lchild=c->rchild;
c->lchild=a; c->rchild=b;
switch(c->bf)
{ case 0: a->bf=b->bf=0;break;
  case 1: a->bf=0; b->bf=-1; break;
  case -1: a->bf=1; b->bf=0; break;
}
c->bf=0;
//以下实现*c与原*a的双亲结点连接(省略)
```



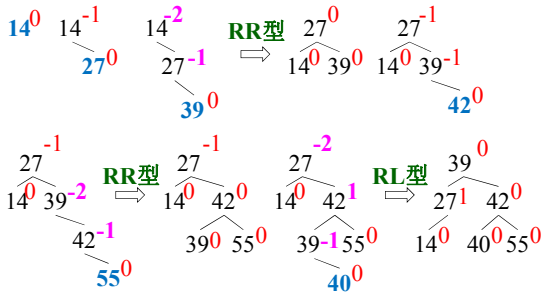
## 8.3 动态查找表 续35

**最后两个问题**
**问题1**：*a的双亲指针fa怎么求？
**答**：在step a.中求a时易求。
**问题2**：旋转类型怎么求？
**答**：typedef enum { **LL**, **LR**, **RR**, **RL** } ROT_TP;

```
//在step b.之后执行以下操作：
if(a->bf==2)
  if(a->lchild->bf==1) rot=LL;  else rot=LR;
if(a->bf==-2)
  if(a->rchild->bf==-1) rot=RR; else rot=RL;
```

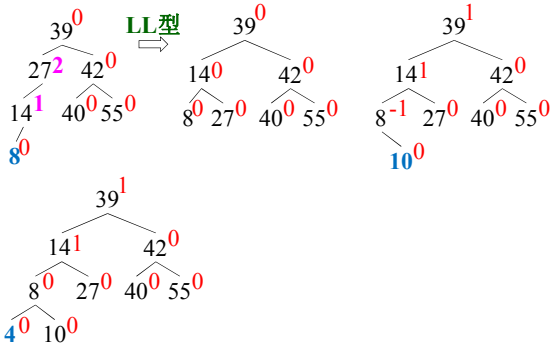
## 8.3 动态查找表 续36

**例3** 11元输入序列为14, 27,39, 42, 55, 40, 8, 10, 4, 9, 32构造平衡的二叉排序树。



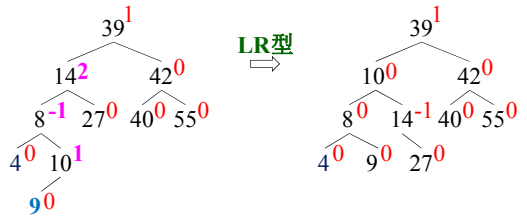



## 8.3 动态查找表 续37



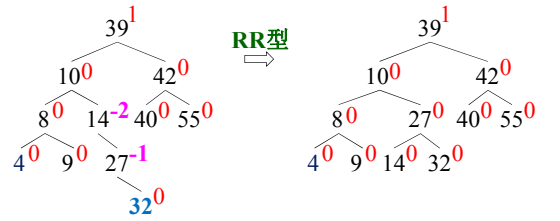



## 8.3 动态查找表 续38






## 8.3 动态查找表 续39






## 8.3 动态查找表 续40

**3. B-树**

1972年R.Bayer和E.M.Cright提出了一种称为B-树的多路平衡查找树，以适合磁盘等直接存取设备上组织动态查找表。

**(1) B-树的定义**

一棵$m$阶的B-树，或为空树，或为满足下列特性的$m$叉树：

a. 树中每个结点至多有$m$棵子树；
b. 若根结点不是叶子结点，则它至少有两棵子树；
c. 除根结点外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；
d. 所有的非终端结点中包含下列信息数据
$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$



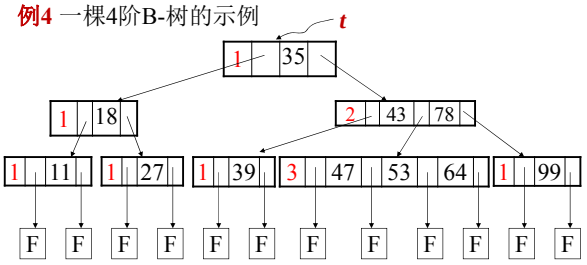
## 8.3 动态查找表 续41

其中, $n$为关键字个数；
$K_1, K_2, \ldots, K_n$为$n$个**递增**的关键字；
$A_0, A_1, \ldots, A_n$为$n$+1个子树的根结点指针；
当$j$=0, 1, …, $n$-1时，$A_j$所指子树中所有结点的关键字均大于$K_j$且小于$K_{j+1}$；$A_n$所指子树中所有结点的关键字均大于$K_n$。
显然，$\lceil m/2 \rceil - 1 \le n \le m-1$ 。

e. 所有的叶子结点都出现在同一层次上，并且不带信息(叶子视为外部结点或查找失败的结点)。

## 8.3 动态查找表 续42

**例4** 一棵4阶B-树的示例




$m$=4，故除根结点外，非终端结点的关键字数目$n$为1~3, 子树的数目为2~4。



## 8.3 动态查找表 续43

**(2) B-树的查找**

```
#define  m  4   //B-树的数据类型定义
typedef struct node
{ int n;                    //结点中关键字的个数
  KeyTp  K[m+1];            //关键字向量，0号单元未用
  struct node *A[m+1]; //子树指针向量
}BSubTNode, *BSubT;
```

在根结点为t的m阶B-树上查找关键字key，返回三元组(pt, i, tag)。若tag=1，表示查找成功，此时有pt->K[i]==key；若tag=0，表示查找失败，此时有key应插入到pt->K[i]和pt->K[i+1]之间。



## 8.3 动态查找表 续44

```
Result BSubTSearch(BSubT t, KeyTp key)
{ p=t; parent=NULL; found=false; i=0;
  while(p&&!found)
  { n=p->n;
    i=Search(p, key); //在结点内查找
    if(i>0&&p->K[i]==key) found=true;
      else { parent=p; p=p->A[i]; }
  }
  if(found) return(p, i, 1);   //查找成功
  return(parent, i, 0);        //查找失败
}
```



## 8.3 动态查找表 续45

```
int Search(BSubT p, KeyTp key)
/* 结点内查找key，返回下标i (0≤i≤n),
   满足p->K[i]≤key<p->K[i+1], 其中K[i]递增有序 */
{ for(i=1; i<=p->n; i++) if(p->K[i]>key) break;
  return i-1;  }
```

**(3) B-树的查找分析**

a. 在B-树上查找结点(在磁盘上进行，算法中未体现)

b. 在结点中找关键字(在内存中进行)

主要时间消耗由操作a.决定，在磁盘上查找的次数取决于B-树上的层次数。可以证明，对任意一棵总共包含$N$个关键字的$m$阶B-树，其树高度$h$(不含 叶子层)至多为$\log_{\lceil m/2 \rceil}\{(N+1)/2\}+1$。



## 8.3 动态查找表 续46

证明如下。

**引理1**：若$m$阶B-树总共包含$N$个关键字，则叶子结点数(即F结点数, F表示查找失败fail)为$N$+1。

**证**：设B-树共有$t$个叶子和$s$个非终端结点，若该树分枝总数为$r$，则除根结点外，每个结点上方有1个分枝，

$t+s=r+1$ ①　(结点数=分枝数+1)

又因为$N$个关键字对应$N$个分枝，每个结点发出的分枝比结点的关键字数目多1，$s$个非终端结点共多出$s$个分支，故所有结点发出的分枝总数

$r=N+s$ ②

②代入①得 $t+s=N+s+1$，即$t=N+1$，证毕。



## 8.3 动态查找表 续47

**引理2**：若$m$阶B-树的非终端结点共有$h$层，则第$h$+1层上的叶子数大于或等于$2\left(\lceil m/2 \rceil\right)^{h-1}$。

**证**：第1层只有1个根结点；

第2层非终端结点数的最小值为2；

第3层非终端结点数的最小值为$2\lceil m/2 \rceil$；

第4层非终端结点数的最小值为$2\left(\lceil m/2 \rceil\right)^2$；

....

第$h$层非终端结点数的最小值为$2\left(\lceil m/2 \rceil\right)^{h-2}$；

第$h$+1层叶子结点数的最小值为$2\left(\lceil m/2 \rceil\right)^{h-1}$。

## 8.3 动态查找表 续48

由引理1, 2得

$N+1 \geq 2\left(\lceil m/2 \rceil\right)^{h-1}$ , 即

$h \leq \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)+1$, 证毕。

**(4) B-树中插入关键字**

B-树中插入结点时，首先在最底层的非终端结点中添加1个关键字(见前面查找算法)，若该结点的关键字个数不超过$m$-1，则算法停止，否则要产生结点的“分裂”，如后面给出的示例所示。

B-树的建立是从空树开始，反复插入结点。



## 8.3 动态查找表 续49

**非终端结点“分裂”方法:**

设指针p所指结点插入关键字值后共有$m$个关键字(最多只允许$m$-1个关键字)和$m$个子树，则以关键字$K_{\lceil m/2 \rceil}$为中心，将该结点**1分为2**。

前1个结点(用指针p1指到)信息为:

$\lceil m/2 \rceil-1, A_0, (K_1, A_1), (K_2, A_2) \cdots, (K_{\lceil m/2 \rceil-1}, A_{\lceil m/2 \rceil-1})$

共含$\lceil m/2 \rceil-1$个关键字和$\lceil m/2 \rceil$棵子树；

后1个结点(用指针p2指到)信息为:

$m-\lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (K_{\lceil m/2 \rceil+1}, A_{\lceil m/2 \rceil+1}), \cdots, (K_m, A_m)$

共含$m-\lceil m/2 \rceil$个关键字和$m-\lceil m/2 \rceil+1$棵子树。



## 8.3 动态查找表 续50

*p中的关键字$K_{\lceil m/2 \rceil}$插入到*p的双亲结点*q中，即若q->$A_j$==p，则执行以下操作：

a. 将*q的$K_{j+1}$~$K_{q->n}$及$A_{j+1}$~$A_{q->n}$后移一个下标位置；

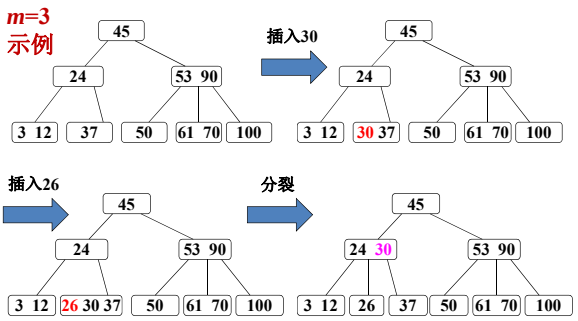
b. q->$A_j$=p1; q->$K_{j+1}$=p->$K_{\lceil m/2 \rceil}$; q->$A_{j+1}$=p2;

完成上述插入后，若*q的关键字数目等于$m$，则对*q进行同样的分裂操作，直到不再需要分裂为止。(有可能一直回溯至根结点分裂后算法才停止)

为了方便求双亲结点，可以给B-树的结点增加双亲指针域或者在搜索关键字的插入位置时，将根结点至最后一层非终端结点路径上的所有非终端结点压入堆栈保存。

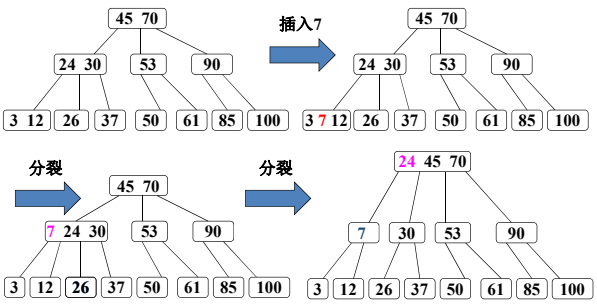


## 8.3 动态查找表 续51



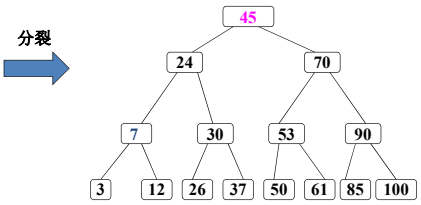



## 8.3 动态查找表 续52






## 8.3 动态查找表 续53



## 8.3 动态查找表 续54

**(4) B-树中删除关键字**

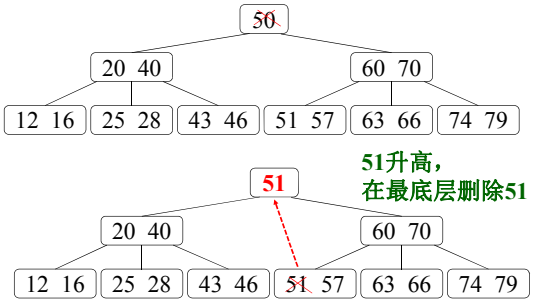
首先找到包含待删除关键字的非终端结点指针p，设 *p中应被删除的关键字为$K_i$。若结点*p位于最后一层非终端结点以上各层中，则用指针p->$A_i$指向的子树(该子树中的关键字均大于$K_i$)中的最小关键字$Y$替换*p中的$K_i$(该最小关键$Y$一定位于最底层非终端结点)，然后在最底层非终端结点中删除关键字$Y$。总之，B-树中删除一个关键字最终对应于在最底层非终端结点中删除一个关键字。

在最底层非终端结点中删除1个关键字后，若关键字数目变为$\lceil m/2 \rceil - 2$则进行结点的“合并”操作。



## 8.3 动态查找表 续55

**例如：$m$=5 删除50**






## 8.3 动态查找表 续56

**最底层非终端结点的“合并”方法：**

设删除关键字的结点指针为p，若删除1个关键字后，*p的关键字数目还剩$\lceil m/2 \rceil - 2$个(即需要进行合并处理)，设*p的双亲结点指针为q且q->$A_i$指向*p。

**情况a**. 若*p的左兄弟的关键字数目≥$\lceil m/2 \rceil$

**a1**. $K$=q->$K_i$;

**a2**. q->$K_i$=左兄弟中的最大关键字; //关键字升高

**a3**. 左兄弟中删除最大关键字;

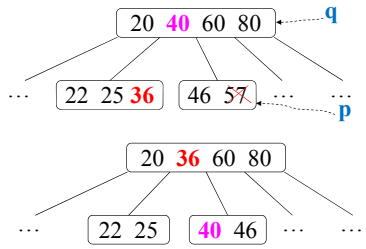
**a4**. $K$插入*p的最开头; //关键字下降

经上述操作后，左兄弟中关键字数目≥$\lceil m/2 \rceil - 1$，*q中关键字数目未改变，*p中关键字数目=$\lceil m/2 \rceil - 1$。



## 8.3 动态查找表 续57

**情况a示例：$m$=5 删除57**






## 8.3 动态查找表 续58

**情况b**. 若*p的右兄弟的关键字数目≥$\lceil m/2 \rceil$

**a1**. $K$=q->$K_{i+1}$;

**a2**. q->$K_{i+1}$=右兄弟中的最小关键字; //关键字升高

**a3**. 右兄弟中删除最小关键字;

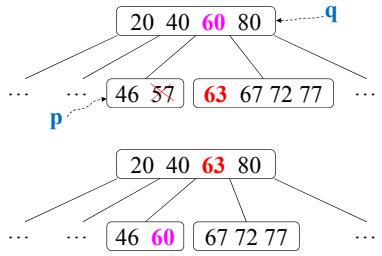
**a4**. $K$插入*p的结尾; //关键字下降

经上述操作后，右兄弟中关键字数目≥$\lceil m/2 \rceil - 1$，*q中关键字数目未改变，*p中关键字数目=$\lceil m/2 \rceil - 1$。



## 8.3 动态查找表 续59

**情况b示例：$m$=5 删除57**



## 8.3 动态查找表 续60

**情况c**. 若*p的左兄弟的关键字数目=$\lceil m/2\rceil-1$

**a1**. $K$=q->$K_i$; *q中删除$K_i$和$A_i$;

**a2**. $K$插入左兄弟中;

**a3**. 将*p中剩余关键字和指针全部并入左兄弟。

经过上述操作后，左兄弟中关键字数为$2(\lceil m/2\rceil-1)$。

若$m$为偶数，则$2(\lceil m/2\rceil-1)=m-2$，

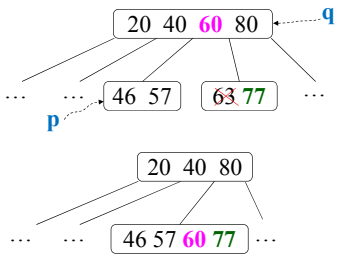
若$m$为奇数，则$2(\lceil m/2\rceil-1)=m-1$。

经过上述操作后，若*q的关键字数=$\lceil m/2\rceil-2$，则*q采用类似方法与其左兄弟或右兄弟合并(有可能直到将原根结点合并，即根结点可能改变，删除结点可能使B-树层数减1)。



## 8.3 动态查找表 续61

**情况c示例：$m$=5 删除63**






## 8.3 动态查找表 续62

**情况d**. 若*p的右兄弟的关键字数目=$\lceil m/2\rceil-1$

**a1**. $K$=q->$K_{i+1}$; *q中删除$K_{i+1}$和$A_i$;

**a2**. $K$插入右兄弟中;

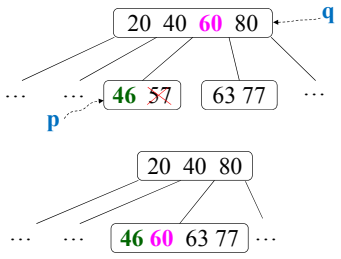
**a3**. 将*p中剩余关键字和指针全部并入右兄弟。

操作结束后，若*q关键字数=$\lceil m/2\rceil-2$，*q也需进行合并操作。



## 8.3 动态查找表 续63

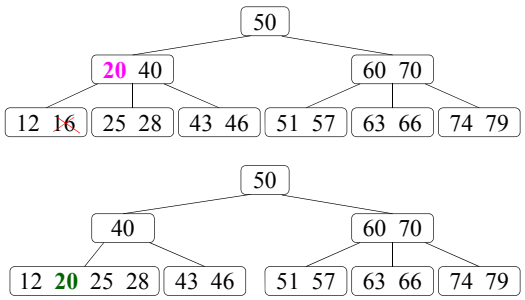
**情况d示例：$m$=5 删除57**





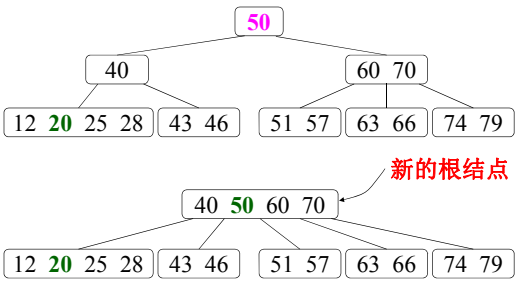

## 8.3 动态查找表 续64

**根结点需要合并的例子：$m$=5 删除16**






## 8.3 动态查找表 续65



## 8.3 动态查找表 续66

**4. B+树**

**(1) B+树的定义**

一棵$m$阶的B+树，或为空树，或为满足下列特性的$m$叉树：

a. 树中每个结点至多有$m$棵子树；
b. 若根结点不是叶子结点，则它至少有两棵子树；
c. 除根结点外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；
d. 所有的非终端结点中包含下列信息数据
$(n, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

其中，$n$为关键字个数；$K_i$为第$i$个关键字且递增有序；$A_i$为第$i$棵子树的根结点指针；关键字$K_i$是第$i$棵子树根结点中的最大(或最小)关键字。



## 8.3 动态查找表 续67

e. 所有叶子结点包含全部关键字，叶子结点中的每个关键字指针指向该关键字对应的物理记录；全部叶子结点从左向右串联成单链表，该单链表中各结点的关键字是递增有序的。

**特点**：

a. B+树有两个出发指针，一是根结点指针，二是最左边叶子结点的指针；
b. 查询的最终目标是叶子中的某关键字，可以从根结点出发查询至叶子上的某个关键字(即随机查询，也称为索引查询)；也可以从最左边叶子出发，顺序查询叶子上的某个关键字。



## 8.3 动态查找表 续68

**(2) B+树插入关键字**

插入关键字只对叶子结点进行。

某叶子结点插入关键字后，若该叶子中的关键字数目≤$m$，此时只需判断插入关键字后该叶子中的最大(最小)关键字是否改变，若有改变，需更新双亲结点中对应的最大(最小)关键字；

某叶子结点插入关键字后，若该叶子中的关键字数目=$m$+1，则需将该叶子分裂为两个叶子结点，所含关键字数目分别为$\lfloor (m+1)/2 \rfloor$和$\lceil (m+1)/2 \rceil$，此时，应删除双亲结点中原对应的最大(最小)关键字，然后将两个新叶子中的最大(最小)关键字插入至双亲结点中；



## 8.3 动态查找表 续69

双亲结点增加了1个关键字后，若其关键字数目也增大到等于$m$+1，则双亲结点继续分裂，如此向上层传导，可能直至根结点分裂；

若根结点一分为二，则将原根结点分裂得到的两个新结点中的最大(最小)关键字生成一个新的含两个关键字的根结点，新根结点的两个指针域分别指向这两个分裂得到的结点(此时，B+树的高度加1)。

**(3) B+树删除关键字**

删除关键字只对叶子结点进行。

某叶子结点删除关键字后，若该叶子中的关键字数目≥$\lceil m/2 \rceil$，此时只需根据该叶子中的最大(最小)关键字是否改变更新双亲中对应的最大(最小)关键字；



## 8.3 动态查找表 续70

某叶子结点删除关键字后，若该叶子中的关键字数目<$\lceil m/2 \rceil$，此时采用与B-树类似的方法，实现结点的“合并”；

“合并”可能向上传导直到根结点，若根结点最后只剩1个关键字(因为第2层两个结点合二为一)，则删除原根结点并令第2层合并后得到的结点成为新的根结点，此时树高减1。

**“合并”操作的两种类型**：

**I型** 关键字数=$\lceil m/2 \rceil - 1$的结点与关键字数=$\lceil m/2 \rceil$的左兄弟或右兄弟合并，则合并后的新结点中的关键字数目为$m$(当$m$为奇数时)或$m$-1(当$m$为偶数时)。此时，在双亲结点中删除两个关键字并插入一个新关键字；



## 8.3 动态查找表 续71

注意：**I型**合并使得双亲结点的关键字数减1，因此双亲结点可能也需要合并，并且若双亲结点中的最大(最小)关键字变化了，需要沿双亲指针向上更新各层结点的最大(最小)关键字。

**II型** 关键字数=$\lceil m/2 \rceil - 1$的结点与关键字数>$\lceil m/2 \rceil$的左兄弟或右兄弟合并。只需将左兄弟中的最大关键字或右兄弟中的最小键字移动到该结点即可。此时，可能有必要更新双亲结点中这两个结点对应的最大(最小)关键字。

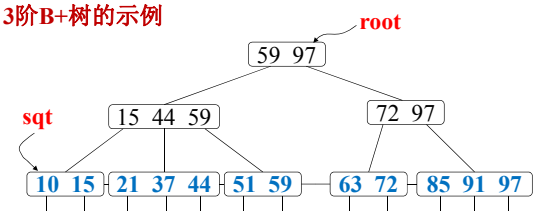
为简化操作，避免向上逐层更新最大(最小)关键字，当删除的关键字为叶子中的最大(最小)关键字时，双亲结点中对应的关键字可不变，成为“**分界关键字**”。

## 8.3 动态查找表 续72

**(4) B+树的应用**

广泛应用于数据库索引。

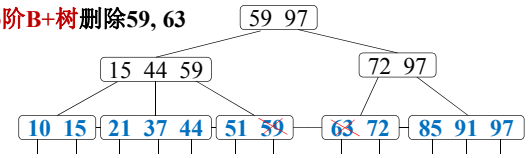
**3阶B+树的示例**



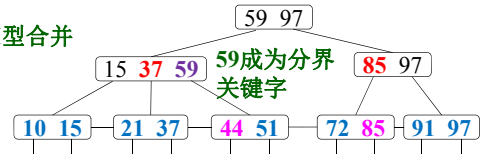



## 8.3 动态查找表 续73

**3阶B+树删除59, 63**



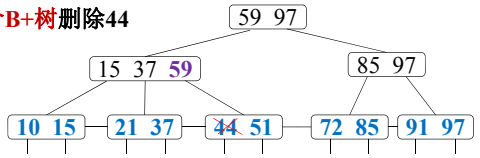

**II型合并**



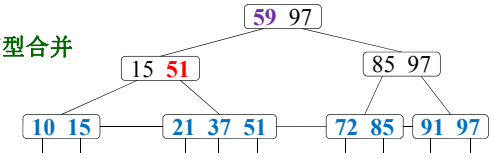



## 8.3 动态查找表 续74

**3阶B+树删除44**



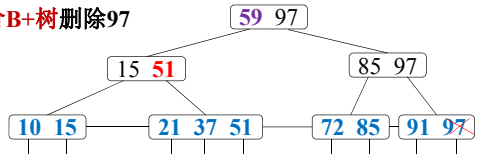

**I型合并**

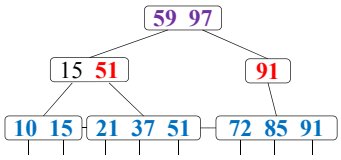





## 8.3 动态查找表 续75

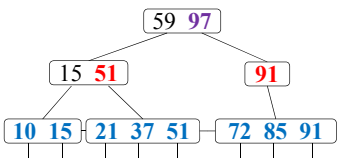
**3阶B+树删除97**



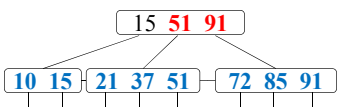






## 8.3 动态查找表 续76



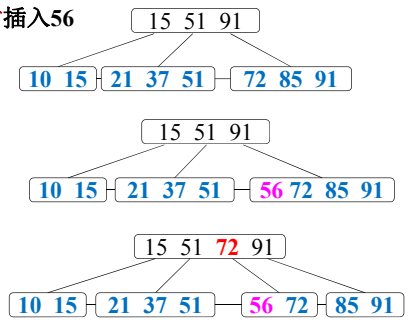

**继续合并**



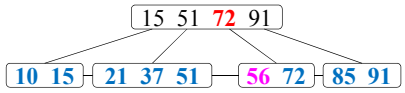



## 8.3 动态查找表 续77

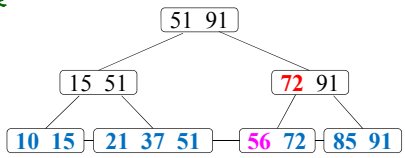
**3阶B+树插入56**



## 8.3 动态查找表 续78完




继续分裂






## 8.4 哈希表

**1. 哈希表的基本概念**

由关键字$K$检索数据元素(对象)时，数据元素(对象)的存储地址可以由函数$H(K)$直接计算得到，这样可以大大加快查找速度。若查找表中的所有数据元素均存储到函数$H(\cdot)$指定的地址，称为哈希(Hash)表。

$H(K)$一般称为**哈希函数**或**散列函数**，因此，哈希表也常称为**散列表**。

**2. 哈希函数设计应考虑的因素**

a. 关键字的特点(类型、长度、键值分布情况等)
b. 地址空间大小及哈希表的大小
c. 数据元素的查询频度
d. 哈希函数本身的计算效率



## 8.4 哈希表 续1

**地址及地址空间**：

数据元素的地址一般用数组元素下标表示，则地址空间大小就是数组长度，也称为**哈希表长度**；

**地址冲突**：

当$K_i \neq K_j$时，$H(K_i)=H(K_j)$，称$K_i$和$K_j$为**同义词**

$H(\cdot)$应尽量减低冲突的概率，但冲突难以完全避免。

**装填因子**：

$\alpha$＝表中数据元素数/哈希表的长度

$\alpha$越大说明哈希表的存储空间利用率越高，但$\alpha$越大往往也意味着冲突概率越大，减少冲突的难度更高。

**哈希函数的设计准则**：

计算**简单**，地址分布**均匀**(发生冲突可能性小)。



## 8.4 哈希表 续2

**3. 哈希函数的常用设计方法**

**(1) 直接定址法**

$H(K)=aK+b$ ($a$, $b$为一常数)

适合于给定的一组关键字为关键字集合中的连续取值或均匀取值的情况。

例如：以年号$Y$为关键字，已知年号范围为2000年以后，则可令$H(Y)=Y-2000$

**优点**：不冲突

**缺点**：适用的关键字类型较少



## 8.4 哈希表 续3

**(2) 数字分析法**

设关键字由$s$个不同符号组成，选择$s$个符号中具有均匀分布的$d'$位($d'$为地址空间大小)作为散列地址。

例：关键字由9个0~9的数字组成，地址空间0~999

| | |
|---|---|
| 0 0 0 3 1 9 4 2 6 | 3 2 6 |
| 0 0 0 7 1 8 3 0 9 | 7 0 9 |
| 0 0 0 6 2 9 4 4 3 | 6 4 3 |
| 0 0 0 7 5 8 6 1 5 | 7 1 5 |
| 0 0 0 9 1 9 6 9 7 | 9 9 7 |
| 0 0 0 3 1 0 3 2 9 | 3 2 9 |
| ①②③④⑤⑥⑦⑧⑨ | 地址取④⑧⑨位 |
| 1 1 1 4 3 3 3 5 5 | |



## 8.4 哈希表 续4

**优点**：简单直观、冲突概率小。

**缺点**：需要事先知道每个关键字的情况并进行关键字各位的统计分析或概率估算。

**(3) 平方取中法**

先将关键字平方，然后取其中间几位作为散列地址(相当于用关键字产生随机数)。所取位数由地址空间范围决定。

若地址空间小于所取位数值决定的范围，可通过乘以一比例因子来解决。

例：地址空间为0~4000，关键字$K$=172148，

$K^2$=29634933904，取中间4位为：3493或4933

比例因子取0.4，则 散列地址为：1397或1973

## 8.4 哈希表 续5

**优点**：平方后的中间几位与关键字的各位都有关系，散列地址均匀分布好，适用多数类型关键字。
**缺点**：计算略为复杂。

**(4) 折叠法与移位法**

把关键字分割成位数相等(最后一部分的位数可以不同)的几部分，然后通过折叠后将几部分进行相加，丢掉进位位，所得值即为散列地址。散列的位数由地址空间的位数而定。

**分割方法**：从右至左
**移位叠加**：将分割后的各部分低位对齐相加
**界间叠加**：将某些部分倒置后再相加



## 8.4 哈希表 续6

例：$K$=24 591 638

移位叠加:

```
  638
  591
+)  24
 1253
```

散列地址=253

界间叠加:

```
  638
  195
+)  24
  857
```

散列地址=857

**优点**：散列地址均匀性好；
**缺点**：哈希函数计算略复杂。



## 8.4 哈希表 续7

**(5) 除留取余法**

$H(K)=K \bmod p$

其中，$p$一般取为素数，且$p<m$($m$为哈希表长)。

**优点**：哈希函数形式简单；
**缺点**：地址冲突可能较多。

**(6) 随机数法**

用其它随机数生成函数实现$H(K)$。



## 8.4 哈希表 续8

**4. 冲突处理**

当 $H(K)$ 的地址中已经存储了数据元素时，应为哈希地址相同的其它数据元素按照某种策略找到空闲的哈希地址单元，以插入这些数据元素。

**(1) 开放定址法**

当发生冲突时，将依次探测“下一个位置”，直到找到其关键字相匹配的元素或找到一个空位插入。“下一个位置”的确定方法是：

$H_i=(H(K)+d_i) \bmod m$ ($i$=1, 2, 3, … 称为探测次数)

$H(\cdot)$: 哈希函数
$m$: 哈希表长度
$d_i$: 求“下一个位置”的增量



## 8.4 哈希表 续9

$d_i$的确定方法有：

**a. 线性探测再散列**

$d_i$=1, 2, …, $m$-1。

若当$d_i=m$-1时仍未查到空闲单元，则说明哈希表已满，这时还要查找另外的溢出表。

**b. 二次探测再散列**

$d_i=1^2, -1^2, 2^2, -2^2, \ldots, \pm k^2$ ($k \le m/2$)

**c. 伪随机探测再散列**

$d_i=p_i$

$p_i$是一个伪随机数序列，该序列应尽量设计来可以遍历哈希表所有位置。



## 8.4 哈希表 续10

**例1** 12个关键字{19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79}用线性探测再散列填充哈希表，计算等概率查找的$ASL_{成功}$。已知$H(K)=K \bmod 13$，哈希表长$m$=16

$H_i=(H(K)+i) \bmod m$，$i$=1, 2, 3, …, $m$-1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 01 | 68 | 27 | 55 | 19 | 20 | 84 | 79 | 23 | 11 | 10 | | | |
| | 1 | 2 | 1 | 4 | 3 | 1 | 1 | 3 | 9 | 1 | 1 | 3 | | | |

$$ASL_{成功}=\frac{1}{12}(1\cdot 6+2\cdot 1+3\cdot 3+4\cdot 1+9\cdot 1)=\frac{30}{12}=2.5$$

**注意**：同义词容易产生“二次聚集”，如元素79

## 8.4 哈希表 续11

**(2) 再哈希法**

$H_i=RH_i(K)$

设置多个哈希函数，即在同义词产生地址冲突时，计算另一个哈希函数地址，直到冲突不再发生。

**优点**：不易产生“聚集”

**缺点**：增加了计算时间。

**(3) 链地址法**

将所有关键字为同义词的数据元素存储在同一链表中。设哈希地址范围为0~$m$-1，设置一个指针向量：

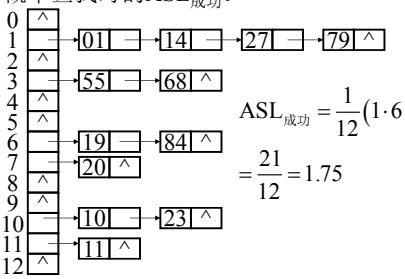
Chain chainhash[$m$];

每个分量的初始状态为NULL指针，凡哈希地址为$i$的的记录则插入到chainhash[$i$]的链表中。



## 8.4 哈希表 续12

**例2** 12个关键字与例1相同，$H(K)=K \bmod 13$，地址空间0~12，用链地址法处理冲突，画出哈希表并计算等概率查找时的$ASL_{成功}$。




$$ASL_{成功}=\frac{1}{12}(1\cdot 6+2\cdot 4+3\cdot 1+4\cdot 1)$$

$$=\frac{21}{12}=1.75$$



## 8.4 哈希表 续13

**(4) 建立一个公共溢出区**

存储空间由基本表hashtable[0..$m$-1]和溢出表overtable[0..$v$]组成。当冲突发生时，不管它的哈希地址是什么，都填入溢出表(从overtable[0]开始依次填入)。

**5. 哈希表的查找**

在哈希表上进行查找的过程与哈希造表的构造过程基本一致：给定$K$值，根据哈希函数求哈希地址。

若表中此位置没有记录，则查找不成功；

否则，比较关键字，若与给定值相等，则查找成功；

否则，按造表时的冲突处理方法，找“下一个地址”，直到哈希表某个位置为空或所填记录关键字等于给定值为止。



## 8.4 哈希表 续14完

**7. 哈希表查找算法的时间复杂度**

**(1) 线性探测再散列**

$$ASL_{成功}\approx\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad ASL_{失败}\approx\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

**(2) 随机探测、二次探测、再哈希**

$$ASL_{成功}\approx-\frac{1}{\alpha}\ln(1-\alpha) \qquad ASL_{失败}\approx\frac{1}{1-\alpha}\ (随机探测)$$

**(3) 链地址**

$$ASL_{成功}\approx 1+\frac{\alpha}{2} \qquad ASL_{失败}\approx\alpha+e^{-\alpha}$$

**结论**：装填因子越大，查找效率越低。



## 第8章 作业

**第1次作业：**

1. 算法设计题：已知$n$元顺序表$a_0, a_1, \ldots, a_{n-1}$按关键字递增有序存储。给定关键字值key，编写算法用对分查找求下标$i$，满足$a_{i-1}$<key且$a_i$≥key。

2. 编程题：输入$n$个两两互不相等的整数，以这些整数为关键字建立平衡的二叉排序树。判断该二叉树是否为平衡的，输出判断结果；输出该二叉树的中序遍历关键字访问次序。

**第2次作业：**

3. 从空树起连续插入以下20个关键字构建$m$=4的B-树。

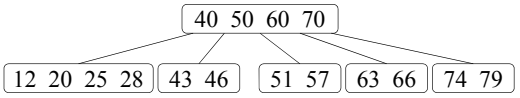
50, 15, 09, 18, 03, 85, 33, 72, 48, 22,

91, 88, 11, 99, 06, 56, 68, 77, 43, 36。



## 第8章 作业 续1完

4. 16个关键字组成的5阶B-树如下图所示，请按关键字递减的次序删除所有结点至空树，画出每删除1个关键字后得到B-树，直至空树。




5. 12个关键字如本电子教案例1所示，设$H(K)=K \bmod 13$，地址空间范围0~15，用二次探测再散列解决冲突。画出哈希表；若各元素等概率查找，求成功查找时的平均查找长度。

# 第9章 内部排序





## 9.1 内部排序的基本概念

**1. 什么是排序(sort)?**

由$n$个数据元素(也称为数据对象或记录)组成的集合$\{R_0, R_1, \dots, R_{n-1}\}$，记$R_i.K$表示$R_i$的关键字$K$，需确定下标集合$\{0, 1, \dots, n-1\}$的一个排列$\{p_0, p_1, \dots, p_{n-1}\}$，满足

$R_{p_0}.K \le R_{p_1}.K \le \cdots \le R_{p_{n-1}}.K$ ($n$个元素按关键字$K$非递减有序)

或 $R_{p_0}.K \ge R_{p_1}.K \ge \cdots \ge R_{p_{n-1}}.K$ ($n$个元素按关键字$K$非递增有序)

这样一种操作称为**排序**或**分类**。



## 9.1 内部排序的基本概念 续1

**2. 排序的稳定性**

当$0 \le i < j \le n-1$时，若$R_i.K=R_j.K$，排序结束后必有$R_i$排列在$R_j$之前，称排序为**稳定的**，否则为**不稳定的**。

排序稳定性的另一种等价定义形式为：排序结束后，对任意$i<j$，若$R_{p_i}.K = R_{p_j}.K$，必有$p_i<p_j$。

**3. 内部与外部排序**

**内部排序**：待排序数据元素全部在内存中存储。

**外部排序**：待排序数据元素数目太多，不能全部存于内存中，排序过程必须从外存读写数据元素。

**本章仅讨论内部排序算法**。



## 9.1 内部排序的基本概念 续2完

**4. 待排序数据元素的存储结构**

本章仅讨论数组顺序存储。其它存储结构也可以实现数据元素的排序，如：二叉排序树(中序遍历得数据元素的非递减有序序列)，单链表数据元素按关键字有序连接等。

**5. 数组元素排序的主要操作**

(1) 关键字比较大小；

(2) 元素存储位置的移动。

为避免数据元素存储位置移动操作，排序结果可以只生成下标集合的排列$\{p_0, p_1, \dots, p_{n-1}\}$(只需要移动下标数组$p$中的下标)，而数据元素本身的存储位置不需要移动，称为**地址排序**。



## 9.2 三种$T(n)=O(n^2)$的内部排序

**1. 简单选择排序**

**算法原理**：基于1维数组元素选最大值/最小值

R[0..$n$-1]升序排列方法：

(0) R[0..$n$-1]选出最小值元素和R[0]交换；

(1) R[1..$n$-1]选出最小值元素和R[1]交换；

…

($i$) R[$i$..$n$-1]选出最小值元素和R[$i$]交换；

…

($n$-2) R[$n$-2..$n$-1]选出最小值和R[n-2]交换。



## 9.2 三种$T(n)=O(n^2)$的内部排序 续1

```
void SelSort(ElemTp R[], int n)  //非递归实现
{ for(i=0; i<=n-2; i++)
    { m=i; for(j=i; j<n; j++) if(R[j].K<R[m].K) m=j;
      if(m!=i) swap(R[i], R[m]);
    }                    //用<为稳定的，用<=不稳定
}
void SelSort1(ElemTp R[], int n, int i)  //递归实现
{ if(i>=n-2) return;
  m=i; for(j=i; j<n; j++) if(R[j].K<R[m].K) m=j;
  if(m!=i) swap(R[i], R[m]);
  SelSort1(R, n, i+1);
}
```

## 9.2 三种$T(n)=O(n^2)$的内部排序 续2

**2. 直接插入排序**

**算法原理**：基于有序数组插入元素保持有序算法

R[0..*n*-1]升序排列方法：

(1) R[0..0]升序, R[1]插入使R[0..1]升序排列；

(2) R[0..1]升序, R[2]插入使R[0..2]升序排列；

…

(*i*) R[0..*i*-1]升序, R[*i*]插入使R[0..*i*]升序排列；

…

(*n*-1) R[0..*n*-2], R[*n*-1]插入使R[0..*n*-1]升序排列；



## 9.2 三种$T(n)=O(n^2)$的内部排序 续3

```
void InsertSort(ElemTp R[], int n)   //非递归实现
{ for(i=1; i<n; i++)
    { e=R[i]; key=R[i].K;
      for(j=i-1; j>=0&&R[j].K>key; j--) R[j+1]=R[j];
      R[j+1]=e;                  //用>为稳定的，>=不稳定
    }
}
```

**特点**：

a. 排序前不必知道所有记录，接收一个元素排序1次；

b. 也适用于单链表结点有序连接；

c. 初始序列“正序”时(各趟内循环次数均为0，即无元素向后搬运)，时间复杂度降低至$O(n)$



## 9.2 三种$T(n)=O(n^2)$的内部排序 续4

**3. 基于元素交换的简单排序-冒泡排序**

**算法原理**：

R[0..*n*-1]升序排列方法：

(1) R[0..*n*-1]将最大元浮动到R[*n*-1]位置；

(2) R[0..*n*-2]将最大元浮动到R[*n*-2]位置；

…

(*i*) R[0..*n*-*i*]将最大元浮动到R[*n*-*i*]位置；

…

(*n*-1) R[0..1]将最大元浮动到R[1]位置；



## 9.2 三种$T(n)=O(n^2)$的内部排序 续5完

```
void BubbleSort(ElemTp R[], int n)
{ for(i=1; i<=n-1; i++)
    for(j=0; j<n-i; j++)
      if(R[j].K>R[j+1].K) swap(R[j], R[j+1]);
}     //此处用>是稳定的，用>=是不稳定的
```

**4. 三种简单排序算法的时间复杂度及稳定性**

时间复杂度均为$O(n^2)$；

均为稳定的排序方法。



## 9.3 希尔排序

**1. 算法原理**

**希尔(Shell)**排序的思想将整个待排记录序列分割成为若干子序列，分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

**子序列的分割方法**：

采用*m*个缩小增量$d_0, d_1, \ldots, d_{m-1}$实现分割，要求

(1) $d_0 > d_1 > \cdots > d_{m-1}$

(2) $d_{m-1} = 1$

(3) $\gcd(d_0, d_1, \cdots d_{m-1}) = 1$ (gcd表示最大公约数)



## 9.3 希尔排序 续1

对每个第*s*趟分割(*s*=0, 1, … , *m*-1)，将全体待排元素按下标的模$d_s$剩余类分组(即元素下标模$d_s$同余的分在同一组)，每组按下标递增次序进行直接插入排序。

**如**：排序下标范围R[0..9]，$d_s$=3，因模3的余数=0, 1, 2故共分3组，即

模3余0：R[0], R[3], R[6], R[9]

模3余1：R[1], R[4], R[7]

模3余2：R[2], R[5], R[8]

显然，$d_{m-1}$=1时，全体记录只分为一组，即最后一趟进行一次普通直接插入排序。由于前面已进行了*m*-1趟分组的插入排序，使整个序列接近正序，因此最后一趟直接插入排序的效率较高。

## 9.3 希尔排序 续2

**例** 设递减增量序列为5, 3, 1, 对以下10元序列进行由小到大希尔排序，写出各趟增量排序结果。

49 38 65 97 76 13 27 49 55 04

**解**：

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 元素 | 49 | 38 | 65 | 97 | 76 | 13 | 27 | 49 | 55 | 04 | 第1趟 |
| 分组 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 增量=5 |
| 结果 | 13 | 27 | 49 | 55 | 04 | 49 | 38 | 65 | 97 | 76 | 第2趟 |
| 分组 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 增量=3 |
| 结果 | 13 | 04 | 49 | 38 | 27 | 49 | 55 | 65 | 97 | 76 | 第3趟 |
| 结果 | 04 | 13 | 27 | 38 | 49 | 49 | 55 | 65 | 76 | 97 | 增量=1 |



## 9.3 希尔排序 续3

**2. 增量的选取方法**

**(1)** $m \le \log_2(n+1)-1$,

$d_s = 2^{m-s}-1 \ (s=0,1,...,m-1)$

**例如**，$n$=31, 则可取$m$=4，得4个递减的增量序列为: 15, 7, 3, 1

**(2)** $m \le \log_3(2n+1)$,

$d_s = \frac{1}{2}\left(3^{m-s}-1\right) \ (s=0,1,...,m-1)$

**例如**，$n$=31, 则可取$m$=3，得3个递减的增量序列为: 13, 4, 1



## 9.3 希尔排序 续4

**3. Shell排序算法描述**

```
void ShellSort(ElemTp R[], int n, int d[], int m)
// R[0..n-1]升序排序, d[0..m-1]为递减的增量
{ for(s=0; s<m; s++)  //m趟剩余类分组插入排序
    for(i=0; i<d[s]; i++)  //模d[s]余i类插入排序
      for(j=i+d[s]; j<n; j+=d[s])
      { e=R[j]; key=R[j].K;
        for(t=j-d[s]; t>=i&&R[t].K>key; t-=d[s])
          R[t+d[s]]=R[t];
        R[t+d[s]]=e;
      }
}
```



## 9.3 希尔排序 续5完

**4. Shell排序算法的时间复杂度**

(1) 增量序列取2.(1)时，$T(n)=O(n^{1.5})$

(2) $n$在某个特定范围内，$T(n)=O(n^{1.3})$

(3) $n\to\infty$，$T(n)=O(n(\log_2 n)^2)$

**5. Shell排序算法的稳定性**

不一定稳定。



## 9.4 快速排序

**1. 算法原理**

**快速排序**(**Quick Sort**)是对冒泡排序的一种改进，其原理如下。

设待排记录下标范围为R[s..t]，任选一个记录称为枢轴(或支点)(pivot)，设法将枢轴通过移动和交换元素的操作放置到排序后的正确位置R[i]，然后对R[s..i-1]和R[i+1..t]分别进行快速排序(递归思想)。

对排序范围R[s..t]，将枢轴元素放置到正确位置的操作称为**一趟快排**。

**2. 枢轴元素的选取方法**

(1) 选R[s]作为枢轴，操作比较方便

(2) “三者取中”法可明显改善时间复杂度



## 9.4 快速排序 续1

所谓“三者取中”，是指比较R[s].K, R[(s+t)/2].K和R[t].K三个元素的关键字，取中间大小元素作枢轴。为方便操作，若枢轴不是R[s]，将它先与R[s]交换，即仍然选R[s]为枢轴。

```
// “三者取中”算法
m=(s+t)/2;
//首先，使R[j1].K<R[j2].K
if(R[m].K<R[t].K) { j1=m; j2=t; } else { j1=t; j2=m; }
if(R[s].K<R[j1].K)
   swap(R[s],R[j1]); //R[j1].K为中间大小
 else if(R[s].K>R[j2].K)
        swap(R[s],R[j2]); //R[j2].K为中间大小
```

## 9.4 快速排序 续2

**3. 一趟快排算法**

```
int qkpass(ElemTp R[], int s, int t)
/*R[s..t]范围以R[s]为支点进行一趟快排，返回一趟快排结束后支点元素的下标 */
{ pivot=R[s]; key=R[s].K; //保存支点
  i=s; j=t;
  while(i<j)
  { //从后向前扫描
    while(i<j&&R[j].K>=key) j--;
    //以上循环结束后R[j+1..t]范围均大于或等于支点
    R[i]=R[j];  //将原R[j]放到支点位置
    //R[j]成为新的支点位置, 但支点实际并未存入R[j]
```



## 9.4 快速排序 续3

```
    //从后向前扫描
    while(i<j&&R[i].K<=key) i++;
    //以上循环结束后R[s..i-1]范围均小于或等于支点
    R[j]=R[i];  //将原R[i]放到支点位置
    //R[i]成为新的支点位置, 但支点实际并未存入R[i]
  }
  R[i]=pivot; //实际放入支点
  return i;  //返回支点下标
}
```



## 9.4 快速排序 续4

**例** 8个待排序关键字为49, 38, 65, 97, 76, 13, 27, $\overline{49}$, 排序下标范围0~7，以0号元素为支点进行非递减的快速排序，求一趟快排的结果以及支点元素的下标。

**解**：

49 38 65 97 76 13 27 $\overline{49}$　　pivot=49

i　　j　　红色表示支点位置

27 38 65 97 76 13 27 $\overline{49}$　第1次向前扫描结束

i　　j

27 38 65 97 76 13 65 $\overline{49}$　第1次向后扫描结束

i　　j



## 9.4 快速排序 续5

27 38 13 97 76 13 65 $\overline{49}$　第2次向前扫描结束

i　　j

27 38 13 97 76 97 65 $\overline{49}$　第2次向后扫描结束

i　　j

27 38 13 97 76 97 65 $\overline{49}$　第3次向前扫描结束

ij

27 38 13 49 76 97 65 $\overline{49}$　放入支点(下标=3位置)

ij



## 9.4 快速排序 续6

**4. 快速排序算法**

```
void QuickSort(ElemTp R[], int s, int t)
{ if(s<t)
  { i=qkpass(R, s, t);
    QuickSort(R, s, i-1);
    QuickSort(R, i+1, t);
  }
}
```

**5. 快速排序的复杂度及稳定性**

(1) 时间复杂度

平均：$O(n\log_2 n)$

是所有同量级排序算法中平均性能最好的



## 9.4 快速排序 续7完

最坏：$O(n^2)$

当待排序列正序或基本正序时，蜕化为起泡排序

(2) 空间复杂度

最好：堆栈深度为$\lfloor \log_2 n \rfloor + 1$

(每趟支点近似等分数组时)

最坏：堆栈深度为$n$

(每趟快排结束后，支点下标均为s)

(3) 稳定性

快速排序不是稳定的。

## 9.5 堆排序

**1. 什么是堆?**

$n$ ($n$≥1)个元素组成的序列{ $R_0, R_1, \dots, R_{n-1}$ }，当且仅当满足如下关系时，称之为**堆**。

$$\begin{cases} R_i.K \geq R_{2i+1}.K \\ R_i.K \geq R_{2i+2}.K \end{cases} \quad \text{或} \quad \begin{cases} R_i.K \leq R_{2i+1}.K \\ R_i.K \leq R_{2i+2}.K \end{cases}$$

其中，$i = 0,1,2,\dots,\left\lfloor \frac{n}{2} \right\rfloor - 1$ 。

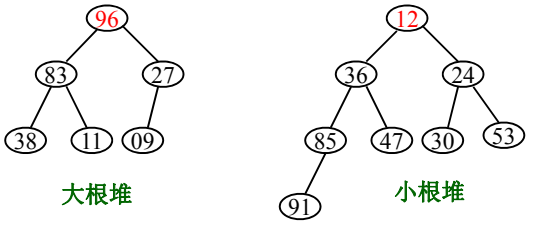
**(1) 堆的存储结构**：一维数组

**(2) 堆的逻辑结构**：完全二叉树，每个结点的关键字小于等于(或大于等于)其左右儿子的关键字。



## 9.5 堆排序 续1

**例如：** { 96, 83, 27, 38, 11, 09}
{12, 36, 24, 85, 47, 30, 53, 91}
对应的完全二叉树结构为：




堆顶96必为序列中最大元　堆顶12必为序列中最小元



## 9.5 堆排序 续2

**2. 堆排序思想**

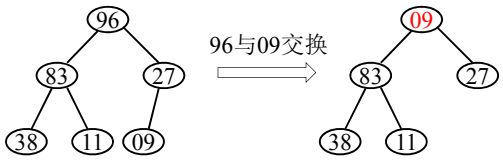
以$n$元大根堆R[0...$n$-1]为例，由小到大排序过程如下：
(1) R[0]↔R[$n$-1]，R[0..$n$-2]调整为大根堆；
(2) R[0]↔R[$n$-2]，R[0..$n$-3]调整为大根堆；
…
($i$) R[0]↔R[$n$-$i$]，R[0..$n$-$i$-1]调整为大根堆；
…
($n$-1) R[0]↔R[1]，R[0..0]调整为大根堆。

**关键问题：**
**a.** 输出堆顶元素(堆顶元素R[0]与R[$n$-$i$]交换)后，怎样将剩余元素R[0..$n$-$i$-1]调整为堆?
**b.** 初始序列R[0..$n$-1]怎样构建为堆？



## 9.5 堆排序 续3

**a.** 输出堆顶元素后，剩余元素调整为堆的方法
**例如** { 96, 83, 27, 38, 11, 09}　{ 09, 83, 27, 38, 11, **96**}

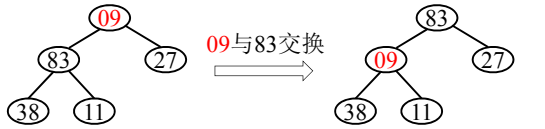



从根结点出发，向下调整：取根结点、左、右儿子中的最大元与根结点元素交换，直到调整到叶子或根结点元素就是三者中的最大元。

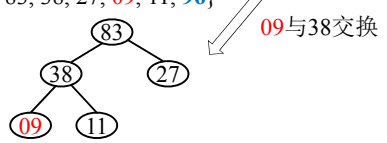


## 9.5 堆排序 续4

{ 09, 83, 27, 38, 11, **96**}　{ 83, 09, 27, 38, 11, **96**}



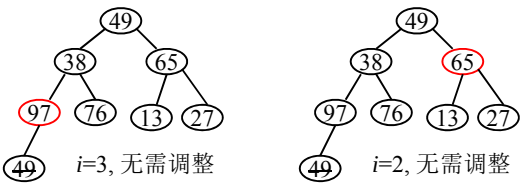

{ 83, 38, 27, 09, 11, **96**}



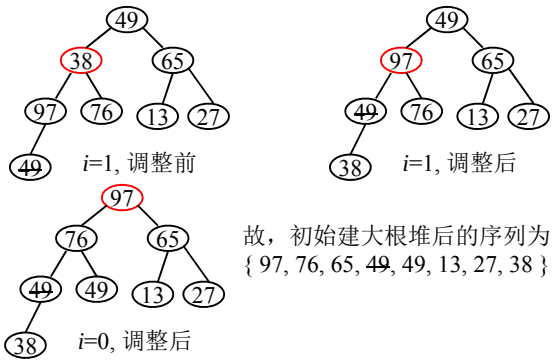



## 9.4 堆排序 续5

**b.** 初始序列R[0..$n$-1]建堆方法
从下标最大的非终端结点R[$\lfloor n/2 \rfloor - 1$ ]开始，按下标减1次序循环到R[0]，对第$i$趟循环，用**a.**方法从R[$i$]出发向下进行调整($i = \lfloor n/2 \rfloor - 1, \dots, 2, 1, 0$)。
**例** 将序列{ 49, 38, 65, 97, 76, 13, 27, ~~49~~}构建为大根堆。



## 9.5 堆排序 续6




故，初始建大根堆后的序列为
{ 97, 76, 65, 49, 49, 13, 27, 38 }



## 9.5 堆排序 续7

**3. 堆排序算法**

**(1) 从R[i]出发，向下调整算法**

```
void HAdjust(ElemTp R[], int i, int n)  //大根堆调整
//i为出发结点下标，n为堆的结点总数
{ while(i<=n/2-1)   //叶子停止，非叶子继续
  { iL=2*i+1; iR=iL+1; //求R[i]的左/右儿子下标
    j=iL;  //j为R[i]要交换的结点下标
    if(iR<n&&R[iR].K>R[iL].K) j=iR;
    if(R[i].K>R[j].K) break; //无需交换，则结束
    swap(R[i], R[j]); i=j;
  }
}
```



## 9.5 堆排序 续8

```
//每次换到下面的元素是固定的，可改进(即不swap)
void HAdjust(ElemTp R[], int i, int n) //避免swap
{ temp=R[i]; key=temp.K;
  while(i<=n/2-1)   //叶子停止，非叶子继续
  { iL=2*i+1; iR=iL+1; //求R[i]的左/右儿子下标
    j=iL;  //j为R[i]要交换的结点下标
    if(iR<n&&R[iR].K>R[iL].K) j=iR;
    if(key>R[j].K) break; //根结点最大则结束调整
    R[i]=R[j]; i=j;
  }
  R[i]=temp;
}
```



## 9.5 堆排序 续9

**(2) 堆排序算法(关键字由小到大排序)**

```
void HeapSort(ElemTp R[], int n)
{ //初始建大根堆
  for(i=n/2-1; i>=0; i--) HAdjust(R, i, n);
  for(i=1; i<n; i++)
  { swap(R[0], R[n-i]);
    HAdjust(R, 0, n-i);
  }
}
```



## 9.5 堆排序 续10完

**4. 堆排序算法的复杂度**

**(1) 时间复杂度**

对于深度为$k$的堆，从堆顶元素向下调整，所需的关键比较次数最多为$2(k-1)$，故$n$个元素组成的序列初始建堆时，关键字比较次数$<4n$；排序过程关键字比较次数$<2n\log_2 n$。堆排序最坏情况下的时间复杂度也是$O(n\log_2 n)$。$n$较小时，不推荐；$n$较大时，推荐堆排序。

**(2) 空间复杂度**

辅助空间需求为$O(1)$

**5. 堆排序算法的稳定性**

不稳定



## 9.6 归并排序

**1. 2-路归并排序思想**

假设初始序列含有$n$个记录，则可以看成$n$个有序的子序列，每个子序列的长度为1，然后进行两两归并操作，得到$\lceil n/2 \rceil$个长度为2或1的有序子序列，再两两归并，……，如此重复，直至得到一个长度为$n$的有序序列为止，这种排序方法称为**2-路归并排序**。2-路归并既适用于顺序存储结构，也适用于链表。

当用一维数组存储$n$个记录时，2-路归并排序需要另一个大小相同的1维数组作为辅助存储空间。

**2. 2-路归并排序算法**

**(1) 2-路归并的非递归实现**

## 9.6 归并排序 续1

例 {20}{-8}{9}{26}{7}{-9}{-26}{34}{15}{-1}
第1趟归并后{-8, 20} {9, 26} {-9, 7} {-26, 34} {-1, 15}
第2趟归并后{-8, 9, 20, 26} {-26, -9, 7, 34} {-1, 15}
第3趟归并后{-26, -9, -8, 7, 9, 20, 26, 34} {-1, 15}
第4趟归并后{-26, -9, -8, -1, 7, 9, 15, 20, 26, 34}

分析：
(1) 归并总趟数=$\lceil \log_2 n \rceil$
(2) 对第$i$趟($i$=1, 2, … ,$\lceil \log_2 n \rceil$)归并，
生成的总分块数=$\left\lceil \frac{n}{2^i} \right\rceil$，
生成的分块长度= $2^i$或小于$2^i$ 。



## 9.6 归并排序 续2

```
void merge(ElemTp p[], int s, int m, int t, ElemTp q[])
//p[s..m]已升序; p[m+1..t]已升序(s≤m≤t)
//归并p[s..m]和p[m+1..t], 结果存于q[s..t], 使q[s..t]升序
{ i=s; j=m+1; k=s;
  while(i<=m&&j<=t)
     if(p[i].K<=p[j].K) q[k++]=p[i++];
       else q[k++]=p[j++];   //用<=是稳定的, 用<不稳定
  while(i<=m) q[k++]=p[i++];
  while(j<=t) q[k++]=p[j++];
}
```

注意：调用时若m=t(及第2个分块为空)，算法只将p[s..m]拷贝至q[s..m]



## 9.6 归并排序 续3

```
void MergeSort(ElemTp R[], int n)
{ ElemTp *p=R, *A=new ElemTp[n], *q=A; int w, s, m, t;
  for(w=1; w<n; w*=2)   //w为每趟分块长度
    { for(s=0; s<n; s+=2*w)
        { m=min(n-1, s+w-1); t=min(n-1, m+w);
          merge(p, s, m, t, q);
        }
      swap(p, q);  //交换使p为归并源, q为归并目标
    }
  if(p==A) for(i=0; i<n; i++) R[i]=A[i];
  delete []A;  //删除辅助数组A
}
```



## 9.6 归并排序 续4

(2) 2-路归并的递归实现

```
void MergeSort(ElemTp R[], ElemTp A[], int s, int t)
//对R[s..t]进行2-路归并升序排序
//结果存于A[s..t]
{ if(s==t) { A[s]=R[s]; return; }
  m=(s+t)/2;
  MergeSort(R, B, s, m);   //B为另一辅助数组
  MergeSort(R, B, m+1, t);
  merge(B, s, m, t, A);
}
```

注意：递归算法的空间复杂度差，不推荐使用



## 9.6 归并排序 续5完

3. 2-路归并排序的复杂度
(1) 时间复杂度
每趟归并：总共$n$次循环
总趟数：$\lceil \log_2 n \rceil$
故平均时间复杂度(也是最坏情况) $T(n)=O(n\log_2 n)$
(2) 空间复杂度
辅助空间：$O(n)$
4. 2-路归并排序的稳定性
是稳定的。



## 9.7 基数排序

1. 基数排序的原理
若有$d(d\geq 2)$个关键字$K^0, K^1, K^2, \dots, K^{d-1}$组成如下所示关键字组进行排序，
$(K^0, K^1, K^2, \dots, K^{d-1})$
其中，$K^0$称为最高位关键字，$K^{d-1}$称为最低位关键字，
对任意两个包含上述关键字组的记录$R_i$和$R_j$， 定义
$(R_i.K^0, R_i.K^1, \dots, R_i.K^{d-1}) < (R_j.K^0, R_j.K^1, \dots, R_j.K^{d-1})$
$\Leftrightarrow \exists t(0\leq t<d$且$R_i.K^t<R_j.K^t$且$\forall s(0\leq s<t \Rightarrow R_i.K^s=R_j.K^s))$
例如: $d$=4, $(K^0, K^1, K^2, K^3)$=(千, 百, 十, 个)
则 (1, 2, 3, 4)<(1, 2, 4, 1) (3, 3, 3, 3)<(3, 3, 3, 9)
(5, 0, 0, 0)<(6, 8, 0, 9)

## 9.7 基数排序 续1

**(1) MSD(Most Significant Digit First)排序准则**

对记录按关键字组($K^0$, $K^1$, $K^2$, … , $K^{d-1}$)进行排序时，先按$K^0$排序，使$K^0$相同的记录排列在一起，然后对$K^0$相同的每个子序列按$K^1$排序，使$K^0$相同的每个子序列进一步细分为$K^1$相同的若干个更小的子序列，如此重复，直至$K^{d-1}$排序结束。

基于MSD准则的排序等效于按照关键字组比较大小的定义直接对关键字组比较大小来实现排序，即只需要定义如下关键字组比较大小的函数即可实现MSD准则排序。



## 9.7 基数排序 续2

```
//MSD准则的实现-关键字组直接比较大小
int compare(ElemTp &a, ElemTp &b)
//返回0表示a==b，返回-1表示a<b，返回+1表示a>b
{ int i, d=…;
  for(i=0; i<d; i++) if(a.K^i!=b.K^i) break;
  if(i>=d) return 0;
  if(a.K^i<b.K^i) return -1;
  return +1;
}
```



## 9.7 基数排序 续3

**例1** ElemTp定义为如下所示时间结构体，实现两个时间按关键字组(时, 分, 秒)比较大小。

```
typedef struct { int h, m, s; }ElemTp;
```

**解**：

```
int compare(ElemTp &a, ElemTp &b)
{ int aK[3]={a.h, a.m, a.s} , bK[3]={b.h, b.m, b.s};
  int i;
  for(i=0; i<3; i++) if(aK[i]!=bK[i]) break;
  if(i>=3) return 0;
  if(aK[i]<bK[i]) return -1;
  return +1;
}
```



## 9.7 基数排序 续4

**(2) LSD(Least Significant Digit First)排序准则**

各关键字位按由低位到高位的次序排序。这需要通过“分配”和“收集”两种操作来实现。对于$d$元关键字组，需要进行$d$趟分配和收集操作。LSD准则实现的排序常称为**基数排序**。

**例2** ElemTp定义为与例1相同的时间结构体，按LSD准则实现排序的方法如下。

**第1趟 分配**操作(按最低位关键字“秒”进行)

记待排序的记录序列为**R**，将**R**中所有记录分为60组，使“秒”值相同的记录在同一组。即包含于同一个分组的记录的“秒”属性值相同。注意：60个分组中，可能有些分组中的记录数为0。



## 9.7 基数排序 续5

在每个不为空的分组中，记录的排列次序满足排序稳定性的定义。比如，原序列中15:20:10排在08:00:10之前，则在“秒”=10这一组，15:20:10仍然排列在08:00:10之前。

**第1趟 收集**操作(对所有“秒”分组求并集)

对60个“秒”分组求并集，并集中所有分组按“秒”由小到大的次序顺序存储，从而得到与原序列长度相同的序列，记为$\mathbf{R}^{(1)}$。

显然，$\mathbf{R}^{(1)}$中所有记录按“秒”有序(“秒”相同的记录排列在一起)；在 “秒”属性值相同的分组内部，记录的先后次序与它们在**R**中的先后次序相同。



## 9.7 基数排序 续6

**第2趟 分配**操作(按关键字“分”进行)

对序列$\mathbf{R}^{(1)}$按“分”属性值是否相同分为60组，对于每个分组中的记录，它们的“分”值相同且相对$\mathbf{R}^{(1)}$中的记录次序保持稳定性，则在“分”相同的子序列中，记录是按“秒”有序的。

**第2趟 收集**操作(对所有“分”分组求并集)

对60个“分”分组求并集，并集中所有分组按“分”由小到大的次序顺序存储，从而得到与原序列长度相同的序列，记为$\mathbf{R}^{(2)}$。

显然，在$\mathbf{R}^{(2)}$中，所有记录是按“分”有序的，“分”相同的记录又是按“秒”有序的。

## 9.7 基数排序 续7

**第3趟 分配**操作(按关键字“时”进行)

对序列$\mathbf{R}^{(2)}$按“时”属性值是否相同分为24组(这里假定“时”的取值范围为0~23)，对于每个分组中的记录，它们的“时”值相同且相对$\mathbf{R}^{(2)}$中的记录次序保持稳定性，即在“时”相同的序列中，记录一定是按(“分”,“秒”)有序的。

**第3趟 收集**操作(对所有“时”分组求并集)

对24个“时”分组求并集，并集中所有分组按“时”由小到大的次序顺序存储，从而得到与原序列长度相同的序列，记为$\mathbf{R}^{(3)}$。

显然，在$\mathbf{R}^{(3)}$中，所有记录是按(“时”，“分”，“秒”)有序的，即排序结束。



## 9.7 基数排序 续8

**2. 链式基数排序**

基数排序非常适合采用链式存储结构。为方便操作，每趟分配得到的每个分组用带头、尾指针的链式队列来实现存储。

分配时，序列中的每个结点按连接次序依次摘下并连接到分组关键字相同的分组(队列)的队尾(这样可以确保同一队列中记录的连接先后次序与它们在原序列中的先后次序相同)。当序列中的所有结点摘完后，则一趟分配操作结束。

收集时，只需将所有分组(队列)按分组关键字有序的次序首尾相连即生成新的序列。



## 9.7 基数排序 续9

**问题与讨论：基数排序的适用范围？**

每趟分配得到的队列数目由对应分组关键字的取值数目决定。如果关键字取值范围为无限种取值，基数排序将无法应用。因此，基数排序中的每个关键字的取值集合应为离散有限个元素集合。

**例3** 对以下3位十进制数组成的序列按由小到排序。

{278, 109, 063, 930, 589, 184, 505, 269, 008, 083}

**解**：可采用3个关键字组成的关键字组(百, 十, 个)实现链式基数排序。由于每个关键字的取值为0~9(共10个值)，故每趟分配得到的分组为10个，用10个带头、尾指针的队列表示。详细排序过程如下。



## 9.7 基数排序 续10

**第1趟分配与收集**(按个位进行)

输入序列：278,109,063,930,589,184,505,269,008,083

**第1趟分配结果**：

| | |
|---|---|
| 0：930 | 5：505 |
| 1： | 6： |
| 2： | 7： |
| 3：063, 083 | 8：278, 008 |
| 4：184 | 9：109, 589, 269 |

**第1趟收集结果**：

930, 063, 083, 184, 505, 278, 008, 109, 589, 269



## 9.7 基数排序 续11

**第2趟分配与收集**(按十位进行)

输入序列：930,063,083,184,505,278,008,109,589,269

**第2趟分配结果**：

| | |
|---|---|
| 0：505, 008, 109 | 5： |
| 1： | 6：063, 269 |
| 2： | 7：278 |
| 3：930 | 8：083, 184, 589 |
| 4： | 9： |

**第2趟收集结果**：

505, 008, 109, 930, 063, 269, 278, 083, 184 , 589



## 9.7 基数排序 续12

**第3趟分配与收集**(按百位进行)

输入序列：505,008,109,930,063,269,278,083,184,589

**第3趟分配结果**：

| | |
|---|---|
| 0：008, 063, 083 | 5：505, 589 |
| 1：109, 184 | 6： |
| 2：269, 278 | 7： |
| 3： | 8： |
| 4： | 9：930 |

**第3趟收集结果**：

008, 063, 083, 109, 184, 269, 278, 505, 589, 930

## 9.7 基数排序 续13

**3. 链式基数排序的算法描述**

```
typedef struct
{ KeyTp0 K0;
  KeyTp1 K1;
  …
  KeyTpd-1 Kd-1;  //共d个关键字(K0为最高位)
  int next;       //静态链表的指针域(用元素下标表示)
} ElemTp;         //元素类型
```

设关键字$K^i$($i$=0, 1, …, $d$-1)的取值个数为$rd(i)$，其$rd(i)$个取值按由小到大的次序排列为

$$K_0^i \le K_1^i \le K_2^i \le \cdots \le K_{rd(i)-1}^i$$



## 9.7 基数排序 续14

定义函数$sp(K^i)$表示：当$K^i = K_j^i$时，$sp(K^i)=j$，其中$j$=0, 1, 2, …, $rd(i)$-1，即

```
int sp(KeyTpi Ki)
{ for(j=0; j<rd(i); j++) if(Ki == Kji) break;
  return j;
}
//链式基数排序算法
int RadixSort(ElemTp R[], int n)
/*n元记录基数排序，假设各记录d元关键字组已初始化，但各记录的指针域尚未初始化。本算法实现由小到大基数排序，算法结束后返回静态链表头结点下标
*/
```



## 9.7 基数排序 续15

```
{ //各结点next域初始化
  for(i=0; i<=n-2; i++) R[i].next=i+1;
  h=0; R[n-1].next=-1;   //h为头指针, -1表示链表结束
  for(i=d-1; i>=0; i--)  //低位关键字向高位关键字循环
  {  //第i趟分配
    int *head=new int[rd(i)], *read=new int[rd(i)];
    for(k=0; k<rd(i); k++) head[k]=-1; //队头指针初始化
    while(h!=-1)
    { k=sp(R[h].Ki); //R[h]应插入到第k个队列中
      if(head[k]==-1) head[k]=h; else R[rear[k]].next=h;
      rear[k]=h; h=R[h].next;
    }
```



## 9.7 基数排序 续16

```
    //第i趟收集
    for(k=0; k<rd(i); k++) if(head[k]!=-1) break;
    h=head[k]; p=rear[k];
    //以上查找第1个非空队列, h为收集后的链表头
    //p跟踪当前链表尾结点
    while(++k<rd(i)) //连接后续的非空队列
      if(head[k]!=-1) { R[p].next=head[k]; p=rear[k]; }
    R[p].next=-1;  //使链表结束
    delete []head; delete []rear;  //删除队列头、尾指针
  }  //end for
  return h;  //返回链表头结点下标
}
```



## 9.7 基数排序 续17

```
//输出排序结果算法
void output(ElemTp R[], int h) //h为头结点下标
{ while(h!=-1)
  { write(R[h]);
    h=R[h].next;
  }
}
```

**4. 基数排序的复杂度**

**(1) 时间复杂度**

第$i$趟分配：循环$n$次($n$为记录总数)；

第$i$趟收集：循环$rd(i)$次；

共$d$趟分配和收集，故



## 9.7 基数排序 续18完

总循环次数= $nd + \sum_{i=0}^{d-1} rd(i)$

为表示简单，记$rd$为$rd(i)$的最大值，于是

$T(n)=O(d(n+rd))$

**(2) 空间复杂度**

辅助空间：$O(rd)$

## 9.8 各种排序方法对比

| 排序方法 | 平均时间 | 最坏情况 | 辅助存储 | 稳定性 |
|---|---|---|---|---|
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 希尔排序 | $O(n^{1.5})$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 快速排序 | $O(n\log n)$ | $O(n^2)$ | $O(\log n)$ | 不稳定 |
| 堆排序 | $O(n\log n)$ | $O(n\log n)$ | $O(1)$ | 不稳定 |
| 归并排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n)$ | 稳定 |
| 基数排序 | $O(d(n+rd))$ | $O(d(n+rd))$ | $O(rd)$ | 稳定 |



## 9.8 各种排序方法对比 续1完

1. 从平均时间看，快速排序最佳，所需时间最省，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后两者相比较，在$n$较大时，归并排序所需时间较堆排序省，但它的空间复杂度最高。
2. 简单排序方法中，直接插入最简单，当序列基本有序或$n$较小时，它最佳。
3. 基数排序的时间复杂度可写成$O(d*n)$，它最适用于$n$值很大而关键字较小($n$远大于$rd$时)的序列。
4. 可以证明，基于关键字比较大小的排序算法中，在最坏情况下能达到的最好的时间复杂度只能是$O(n\log n)$，如堆排序和归并排序。



## 第9章 作业

**第1次作业：**

1. 算法设计与分析题：将直接插入排序的内循环改造为使用对分查找实现元素插入，请写出基于对分查找的插入排序算法并给出其时间复杂度分析。
2. 算法设计：将教案给出的非递归直接插入排序和冒泡排序算法用递归算法实现。
3. 算法设计：带附加头结点单链表将各数据结点按关键字升序连接。

**第2次作业：**

4. 编程题：输入20个整数，分别用希尔排序、快速排序、堆排序和归并排序实现由小到大排序并输出排序结果。



## 第9章 作业 续1完

5. 编程题：键盘输入$n$个无符号整数，用链式基数排序实现由小到大排序，输出排序结果。
提示：对于C语言32bit宽的unsigned类型，可以采用16进制形式来实现基数排序，即32bit共有8个16进制位，每个16进制位进行一趟分配和收集，共8趟。