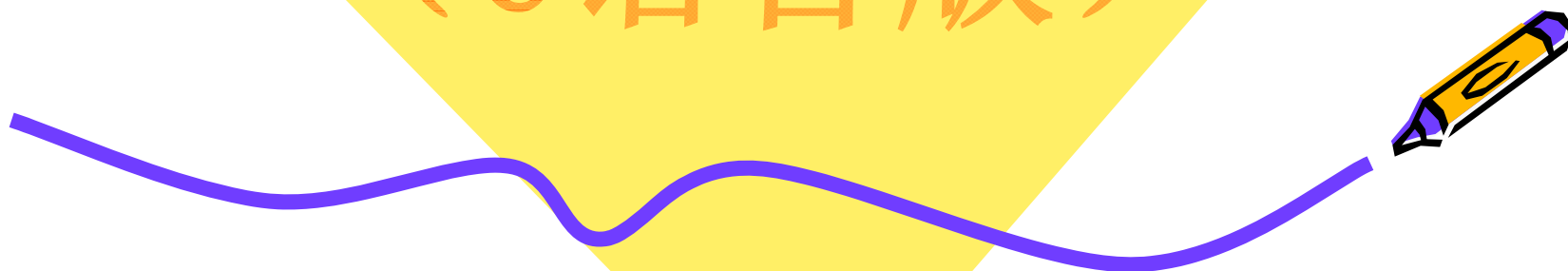




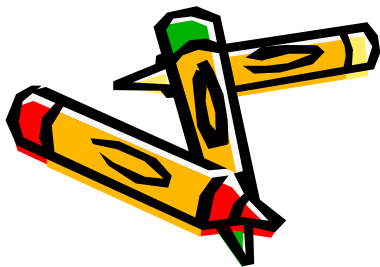
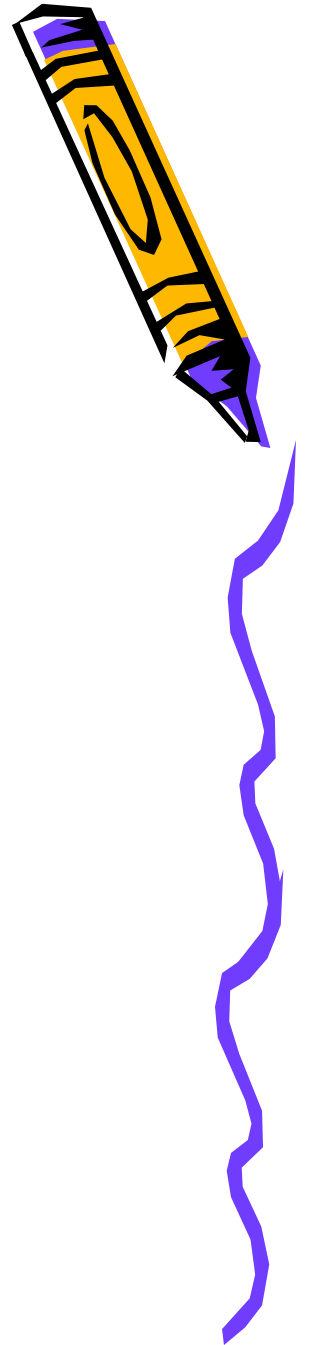
数据结构 (C语言版)



数据结构

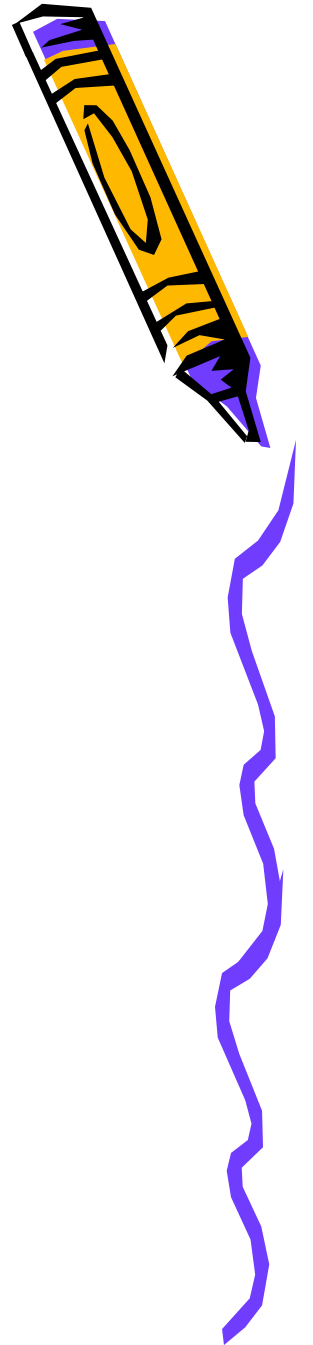
教 材： 数据结构——用C语言描述
李云清 人民邮电出版社

参考书： 数据结构（C语言版）
严蔚敏 清华大学出版社



辅 导： 马金忠

- 时间：
- 地点：
- 二教 四楼 信息对抗教研室



思考：为什么要学习数据结构？
数据结构要学习些什么？



数据结构

- 线性表
- 字符串、特殊矩阵
- 树型结构、二叉树
- 图
- 检索
- 排序





第1章 概论

数据结构讨论的是数据的逻辑结构、存储方式以及相关操作的实现等问题，为学习后续专业课程打下基础。本章讲述数据结构的基本概念及相关术语，介绍数据结构、数据类型和抽象数据类型之间的联系，介绍了算法的特点及算法的时间与空间复杂性。



1.1 数据结构

1.1.1 数据结构

1.1.2 数据的逻辑结构

1.1.3 数据的存储结构

1.1.4 数据的运算集合

1.2 数据类型和抽象数据类型

1.2.1 数据类型

1.2.2 数据结构

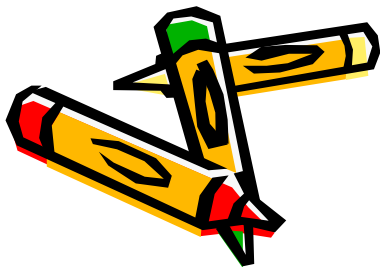
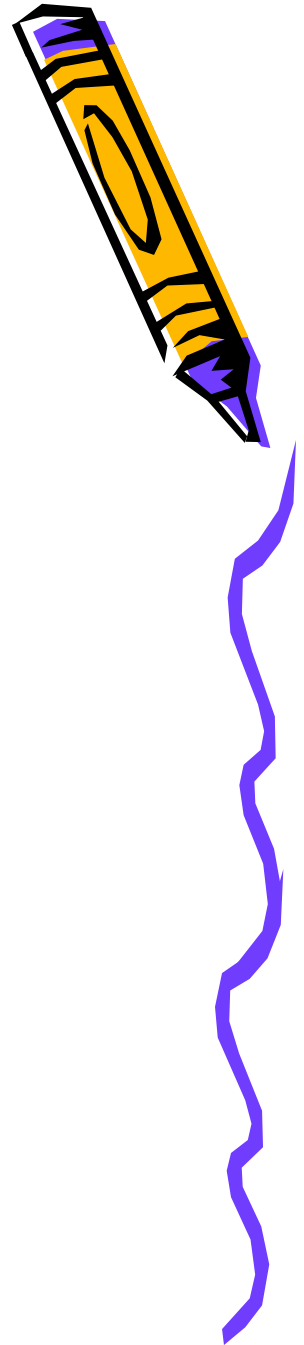
1.2.3 抽象数据类型

1.2.4 抽象数据类型的描述和实现

1.3 算法和算法分析

1.3.1 算法

1.3.2 算法的时间和空间复杂性



1.1 数据结构

1.1.1 数据结构

随着计算机软、硬件的发展，计算机的应用范围在不断扩大，计算机所处理的数据的数量也在不断扩大，计算机所处理的数据已不再是单纯的数值数据，而更多的是非数值数据。

需要处理的数据并不是杂乱无章的，它们一定有内在的联系，只有弄清楚它们之间的本质的联系，才能使用计算机对大量的数据进行有效的处理。

某电信公司的市话用户信息表格如下图所示：

序号	用户名	电话号码	用户住址	
			街道名	门牌号
00001	万方林	3800235	北京西路	1659
00002	吴金平	3800667	北京西路	2099

这里序号、用户名、电话号码等项称为基本项，它是有独立意义的最小标识单位，而用户住址称为组合项，组合项是由一个或多个基本项或组合项组成，是有独立意义的标识单位，每一行称为一个结点，每一个组合项称为一个字段。

使用计算机处理用户信息表中的数据时，必须弄清楚下面3个问题：

1 数据的逻辑结构

这些数据之间有什么样的内在联系？

除最前和最后两个结点之外，表中所有其它的结点都有且仅有一个和它相邻位于它之前的一个结点，也有且仅有一个和它相邻位于它之后的一个结点，这些就是用户信息表的逻辑结构。

2 数据的存储结构

将用户信息表中的所有结点存入计算机时，就必须考虑存储结构，使用C语言进行设计时，常见的方式是用一个结构数组来存储整个用户信息表，每一个数组元素是一个结构，它对应于用户信息表中的一个结点。数据在计算机的存储方式称为存储结构。

3 数据的运算集合

数据处理必涉及到相关的运算，在上述用户信息表中，可以有删除一个用户、增加一个用户和查找某个用户等操作。应该明确指明这些操作的含义。比如删除操作，是删除序号为**5**的用户还是删除用户名为王三的用户是应该明确定义的，如果需要可以定义两个不同的删除操作，为一批数据定义的所有运算（或称操作）构成一个运算（操作）集合。

对待处理的数据，只有分析清楚上面**3**方面的问题，才能进行有效的处理！

数据结构就是指按一定的逻辑结构组成的一批数据，使用某种存储结构将这批数据存储于计算机中，并在这些数据上定义了一个运算集合。

1.1.2数据的逻辑结构

例如，有5个人，分别记为a,b,c,d ,e,其中a是b的父亲，b是c的父亲，c是d的父亲,d是e的父亲，如果只讨论他们之间所存在的父子关系，则可以用下面的二元组形式化地予以表达。

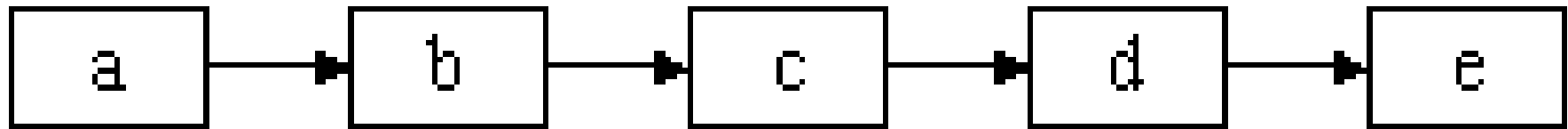
$$B = (K, R)$$

其中： $K = \{a, b, c, d, e\}$

$$R = \{r\}$$

$$r = \{ \langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle d, e \rangle \}$$

逻辑结构的图形表示方式，对K中的每个结点用一个方框表示，而结点之间的关系用带箭头的线段表示，这5人之间的逻辑结构用图形的方式表达如下图所示



若 $k_i \in K$, $k_j \in K$, $\langle k_i, k_j \rangle \in r$, 则称 k_i 是 k_j 的相对于关系 r 的前驱结点, k_j 是 k_i 的相对于关系 r 的后继结点, 前例中, 因为一般只讨论具有一种关系的逻辑结构, 即 $R=\{r\}$, 所以简称 k_i 是 k_j 前驱, k_j 是 k_i 的后继。如果某个结点没有前驱结点, 称之为开始结点; 如果某个结点没有后继结点, 称之为终端结点; 既不是开始结点也不是终端结点的结点称为内部结点。

数据的逻辑结构分为两大类

(1) 线性结构

线性结构的逻辑特征：有且仅有一个开始结点和一个终端结点，并且所有结点都最多只有一个直接前趋和一个直接后继。

(2) 非线性结构

非线性结构的逻辑特征：一个结点可能有多个直接前趋和直接后继。



1.1.3数据的存储结构

数据的逻辑结构是独立于计算机的，它与数据在计算机中的存储无关，要对数据进行处理，就必须将数据存储到计算机中。如果将数据在计算机中无规律地存储，那么在处理时是非常糟的，是没有用的。试想一下，如果一本英汉字典中的单词是随意编排的，这本字典谁会用！

对于一个数据结构 $B = (K, R)$ ，必须建立从结点集合到计算机某个存储区域 M 的一个映像，这个映像要直接或间接地表达结点之间的关系 R 。数据在计算机中的存储方式称为数据的存储结构。

数据的存储结构主要有4种。



数据的存储结构主要有4种。

1 顺序存储

顺序存储通常用于存储具有线性结构的逻辑上相邻的结点存储在连续存储区域M的存储单元中，使得逻辑相邻的结点一定是物理

对于一个数据结构 $B = (K, R)$

其中 $K = \{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9\}$

$R = \{r\}$

$r = \{ \langle k_1, k_2 \rangle, \langle k_2, k_3 \rangle, \langle k_3, k_4 \rangle, \langle k_4, k_5 \rangle, \langle k_5, k_6 \rangle, \langle k_6, k_7 \rangle, \langle k_7, k_8 \rangle, \langle k_8, k_9 \rangle \}$

它的顺序存储方式如图所示

存储地址 M

1001

k_1

1002

k_2

1003

k_3

1004

k_4

1005

k_5

1006

k_6

1007

k_7

1008

k_8

1009

k_9



2 链式存储

链式存储方式是给每个结点附加一个结点的指针所指的是该结点的后继的存为一个结点可能有多个后继，所以指针段指针，也可以是多个指针。

例，数据的逻辑结构 $B = (K, R)$

其中 $K = \{k_1, k_2, k_3, k_4, k_5\}$

$R = \{r\}$

$R = \{ \langle k_1, k_2 \rangle, \langle k_2, k_3 \rangle, \langle k_3, k_4 \rangle, \langle k_4, k_5 \rangle \}$

这是一个线性结构，它的链式存储如图所示。

存储地址	info	next
1000		
1001	k_1	1003
1002		
1003	k_2	1007
1004		
1005	k_4	1006
1006	k_5	\wedge
1007	k_3	1005
1008		



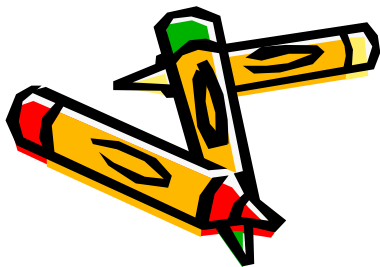
3 索引存储

就像目录一样。索引存储就是根据结点的索引号确定该结点的存储地址。



4 散列存储

散列存储的思想是构造一个从集合 K 到存储区域 M 的一个函数 h ，该函数的定义域为 K ，值域为 M ， K 中的每个结点 k_i 在计算机中的存储地址由 $h(k_i)$ 确定。



1.1.4数据的运算集合

对于一批数据，数据的运算是定义在数据的逻辑结构之上的，而运算的具体实现就依赖于数据的存储结构。

数据的运算集合要视情况而定，一般而言，数据的运算包括插入、删除、检索、输出、排序等。

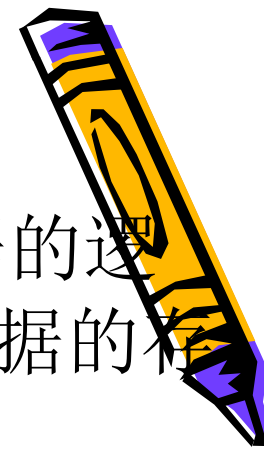
插入：在一个结构中增加一个新的结点。

删除：在一个结构中删除一个结点。

检索：在一个结构中查找满足条件的结点。

输出：将一个结构中所有结点的值打印、输出。

排序：将一个结构中所有结点按某种顺序重新排列。



第1章 内容

1.1 数据结构

1.1.1 数据结构

1.1.2 数据的逻辑结构

1.1.3 数据的存储结构

1.1.4 数据的运算集合

1.2 数据类型和抽象数据类型

1.2.1 数据类型

1.2.2 数据结构

1.2.3 抽象数据类型

1.2.4 抽象数据类型的描述和实现

1.3 算法和算法分析

1.3.1 算法

1.3.2 算法的时间和空间复杂性



1.2数据类型和抽象数据类型

在程序设计中，数据和运算是两个不可缺少的因素。所有的程序设计活动都是围绕着数据和其上的相关运算而进行的。从机器指令、汇编语言中的数据没有类型的概念，到现在的面向对象程序设计语言中抽象数据类型概念的出现，程序设计中的数据经历了一次次抽象，数据的抽象经历了三个发展阶段。

➤ 从无类型的二进制数到基本数据类型的产生

➤ 从基本数据类型到用户自定义类型的产生

➤ 从用户自定义类型到抽象数据类型的出现



1.2.1数据类型

数据类型（或简称类型）反映了数据的取值范围以及对这类数据可以施加的运算。

1.2.2数据结构

数据结构是计算机科学中广泛使用的一个术语，在计算机科学中具有非常重要的作用。数据结构包括三个方面的内容：一组数据中各数据之间的逻辑关系；这组数据在计算机中的存储方式；对这组数据所能施加的运算的集合。数据结构是数据存在的形式。所有的数据都是按照数据结构进行分类的。简单数据类型对应于简单的数据结构；构造数据类型对应于复杂的数据结构。



1.2.3抽象数据类型

抽象数据类型是与表示无关的数据类型，是一个数据模型及定义在该模型上的一组运算。对一个抽象数据类型进行定义时，必须给出它的名字及各运算的运算符号名，即函数名，并且规定这些函数的参数性质。一旦定义了一个抽象数据类型及具体实现，程序设计中就可以像使用基本数据类型那样，十分方便地使用抽象数据类型。

1.2.4抽象数据类型的描述和实现

抽象数据类型的描述包括给出抽象数据类型的名称、数据的集合、数据之间的关系和操作的集合等方面的描述。抽象数据类型的设计者根据这些描述给出操作的具体实现，抽象数据类型的使用者依据这些描述使用抽象数据类型。

抽象数据类型描述的一般形式如下：

ADT 抽象数据类型名称 {

 数据对象：

 数据关系：

 操作集合：

 操作名1：

 操作名n：

}ADT 抽象数据类型名称



第1章 内容

1.1 数据结构

1.1.1 数据结构

1.1.2 数据的逻辑结构

1.1.3 数据的存储结构

1.1.4 数据的运算集合

1.2 数据类型和抽象数据类型

1.2.1 数据类型

1.2.2 数据结构

1.2.3 抽象数据类型

1.2.4 抽象数据类型的描述和实现

1.3 算法和算法分析

1.3.1 算法

1.3.2 算法的时间和空间复杂性



1.3 算法和算法分析

1.3.1 算法

为了求解某问题，必须给出一系列的运算规则，这一系列的运算规则是有限的，表达了求解问题方法和步骤，这就是一个算法。

一个算法可以用自然语言描述，也可以用高级程序设计语言描述，也可以用伪代码描述。本书采用C语言对算法进行描述。



算法具有五个基本特征：

- ①有穷性，算法的执行必须在有限步内结束。
- ②确定性，算法的每一个步骤必须是确定的无二义性的。
- ③输入， 算法可以有0个或多个输入。
- ④输出， 算法一定有输出结果
- ⑤可行性，算法中的运算都必须是可以实现的。

算法具有有穷性，程序不需要具备有穷性。一般的程序都会在有限时间内终止，但有的程序却可以不在有限时间内终止，如一个操作系统在正常情况下是永远都不会终止的。



1.3.2 算法的时间和空间复杂性

一个算法的优劣主要从算法的执行时间和所需要占用的存储空间两个方面衡量，算法执行时间的度量不是采用算法执行的绝对时间来计算的，因为一个算法在不同的机器上执行所花的时间不一样，在不同时刻也会由于计算机资源占用情况的不同，使得算法在同一台计算机上执行的时间也不一样，所以对于算法的时间复杂性，采用算法执行过程中其基本操作的执行次数，称为计算量来度量。

算法中基本操作的执行次数一般是与问题规模有关的，对于结点个数为 n 的数据处理问题，用 $T(n)$ 表示算法基本操作的执行次数。



在评价算法的时间复杂性时，不考虑两算法执行次数之间的细小区别，而只关心算法的本质差别。为此，引入一个所谓的 $O()$ 记号，则
 $T1(n)=2n=O(n), T2(n)=n+1=O(n)$ 。

一个函数 $f(n)$ 是 $O(g(n))$ 的，则一定存在正常数 c 和 m ，使对所有的 $n>m$ ，都满足 $f(n)<c*g(n)$ 。

下面的表格给出了一些具体函数的 $O()$ 的表示，如图所示。

$f(n)$	$O(g(n))$	量级
35	$O(1)$	常数阶
$2n+7$	$O(n)$	线性阶
n^2+10	$O(n^2)$	平方阶
$2n^3+n$	$O(n^3)$	立方阶

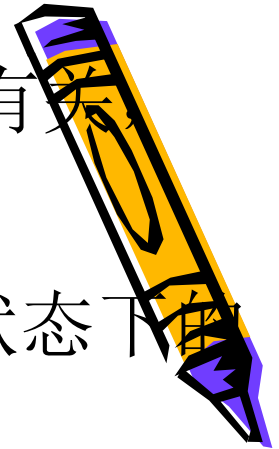
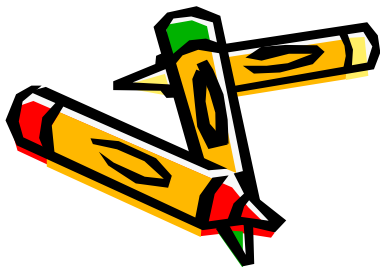
算法的时间复杂性不仅和问题的规模大小有关，还与问题数据的初始状态有关。

这样就有了算法在最好、最坏以及在平均状态下的时间复杂性的概念。

①算法在最好情况下的时间复杂性是指算法计算量的最小值。

②算法在最坏情况下的时间复杂性是指算法计算量的最大值。

③算法的平均情况下的时间复杂性是指算法在所有可能的情况下的计算量经过加权计算出的平均值。

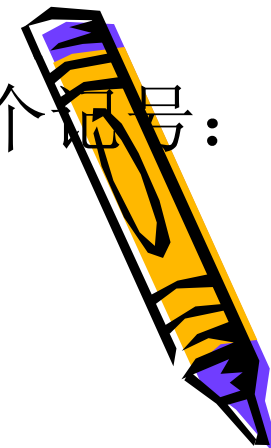


本书在对算法进行分析时，会用到如下两个记号：

$\lceil x \rceil$ ：表示不大于 x 的最大整数；

$\lfloor x \rfloor$ ：表示不小于 x 的最小整数。

作业： 1.1, 1.4, 1.5, 1.6, 1.8, 1.9, 1.10



第2章 线性表及其顺序存储

2.1 线性表概念

2.2 顺序表

2.2.1 顺序表

2.2.2 顺序表的实现

2.3 栈

2.3.1 栈概念

2.3.2 顺序栈及其实现

2.3.3 栈的应用之一-----括号匹配

2.4 队列

2.4.1 队列概念

2.4.2 顺序队列及其实现

2.4.3 顺序循环队列及其实现

2.1线性表

线性表是一个线性结构，它是一个含有 $n \geq 0$ 个结点的有限序列，对于其中的结点，有且仅有一个开始结点没有前驱但有一个后继结点，有且仅有一个终端结点没有后继但有一个前驱结点，其它的结点都有且仅有一个前驱和一个后继结点。一般地，一个线性表可以表示成一个线性序列： k_1, k_2, \dots, k_n ，其中 k_1 是开始结点， k_n 是终端结点。

第2章 线性表及其顺序存储

2.1 线性表概念

2.2 顺序表

2.2.1 顺序表

2.2.2 顺序表的实现

2.3 栈

2.3.1 栈概念

2.3.2 顺序栈及其实现

2.3.3 栈的应用之一-----括号匹配

2.4 队列

2.4.1 队列概念

2.4.2 顺序队列及其实现

2.4.3 顺序循环队列及其实现

2.2顺序表

2.2.1顺序表

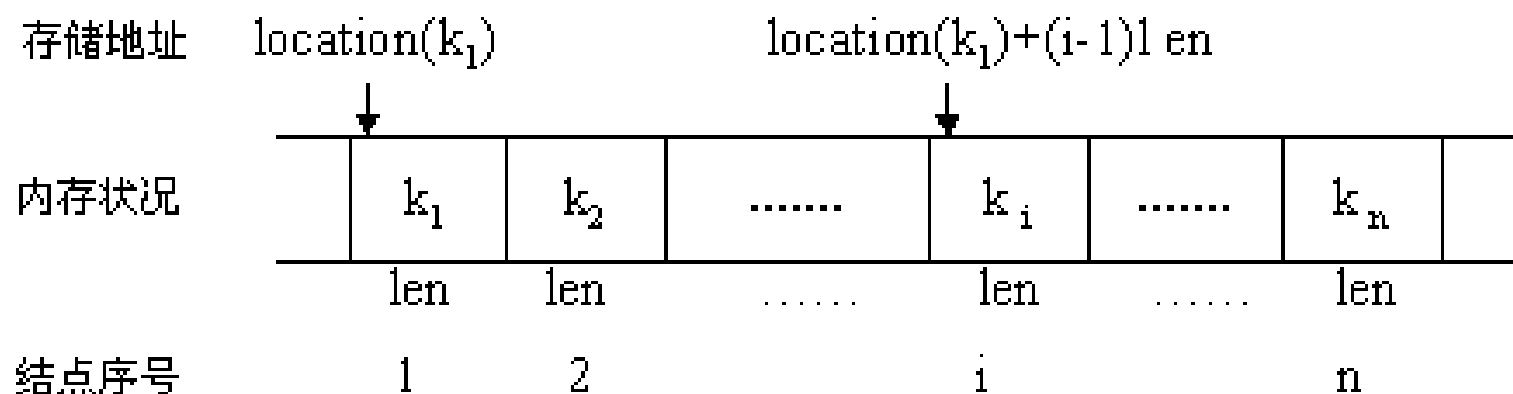
线性表采用顺序存储的方式存储就称之为顺序表。顺序表是将表中的结点依次存放在计算机内存中一组地址连续的存储单元中。

如顺序表的每个结点占用 len 个内存单元，用 $location(k_i)$ 表示顺序表中第 i 个结点 k_i 所占内存空间的第1个单元的地址。则有如下的关系

$$location(k_{i+1}) = location(k_i) + len$$

$$location(k_i) = location(k_1) + (i-1)len$$

顺序表的存储结构如下图所示：



存储结构要体现数据的逻辑结构，顺序表的存储结构中，内存中物理地址相邻的结点一定具有顺序表中的逻辑关系。

顺序表类型的描述如下：

ADT sequence_list{

数据集合K: $K=\{k_1, k_2, \dots, k_n\}, n \geq 0$, K中的元素是datatype类型

数据关系R: $R=\{r\}, r=\{ \langle k_i, k_{i+1} \rangle | i=1, 2, \dots, n-1 \}$

操作集合:

(1) void init_sequence_list(sequence_list *slt) 顺序表的初始化-----置空表

(2) void insert_sequence_list(sequence_list *slt,datatype x) 后部插入值为x结点

(3) void print_sequence_list(sequence_list slt) 打印顺序表的各结点值

(4) int is_empty_sequence_list(sequence_list slt) 判断顺序表是否为空

(5) int find_num_sequence_list(sequence_list slt,datatype x) 查找值为x结点位置

(6) int get_data_pos(sequence_list slt,int i) 取得顺序表中第i个结点的值

(7) void insert_pos_sequence_list(sequence_list *slt,int position,datatype x)

(8) void delete_pos_sequence_list(sequence_list *slt,int position)

} ADT sequence_list;

2.2.2 顺序表的实现

用C语言中的数组存储顺序表。C语言中数组的下标是从0开始的，即数组中下标为0的元素对应的是顺序表中的第1个结点，数组中下标为*i*的元素对应的是顺序表中的第*i*+1个结点。为了方便，将顺序表中各结点的序号改为和对应数组元素的下标序号一致，即将顺序表中各结点的序号从0开始编号。这样，一个长度为*n*的顺序表可以表示为

$$\{k_0, k_1, k_2, \dots, k_{n-1}\}$$

顺序表的存储结构的C语言描述如下：

```
/**
```

```
/*顺序表的头文件， 文件名sequlist.h*/
```

```
/**
```

```
#define MAXSIZE 100
```

```
typedef int datatype;
```

```
typedef struct{
```

```
    datatype a[MAXSIZE];
```

```
    int size;
```

```
}sequence_list;
```

顺序表的几个基本操作的具体实现：

```
/*  
/*          顺序表的初始化---置空表          */  
/* 文件名seqllinit.c, 函数名init_sequence_list()  */  
/*  
void init_sequence_list(sequence_list *slt)  
  
{  
    slt->size=0;  
}
```

算法2.1顺序表的初始化---置空表


```
/*  
    在顺序表后部进行插入操作  
    文件名seqlinse.c, 函数名insert_sequence_list()  
*/  
void insert_sequence_list(sequence_list *slt,datatype x)  
{  
    if(slt->size==MAXSIZE)  
        {printf("顺序表是满的!");exit(1);}  
    slt->size=slt->size+1;  
    slt->a[slt->size]=x;  
}
```

算法2.2在顺序表后部进行插入操作

```

/*****
/*      打印顺序表的各结点值      */
/*  文件名seq1prin.c, 函数名print_sequence_list()  */
*****/

void print_sequence_list(sequence_list slt)
{
    int i;
    if(!slt.size) printf("\n顺序表是空的!");
    else
        for(i=0;i<slt.size;i++) printf("%5d",slt.a[i]);
}

```

算法2.3打印顺序表的各结点值

```
/**
 * 判断顺序表是否为空
 * 文件名seqlemp.c, 函数名is_empty_sequence_list() */
/**

int is_empty_sequence_list(sequence_list slt)
{
    return(slt.size==0 ? 0:1);
}
```

算法2.4判断顺序表是否为空

```

/*****
/*      查找顺序表中值为x的结点位置      */
/*  文件名seq1find.c, 函数名find_num_sequence_list()  */
*****/

```

```
int find_num_sequence_list(sequence_list slt,datatype x)
{
    int i=0;
    while(slt.a[i]!=x&& i<slt.size) i++;
    return(i<slt.size? i:-1);
}
```

算法2.5查找顺序表中值为x的结点位置

```
/******  
/*      取得顺序表中第i个结点的值      */  
/*文件名seq1get.c,函数名get_data_pos_sequence_list() */  
/******
```

```
int get_data_pos(sequence_list slt,int i)  
{  
    if(i<0||i>=slt.size)  
        {printf("\n指定位置的结点不存在!");exit(1);}  
    else  
        return slt.a[i];  
}
```

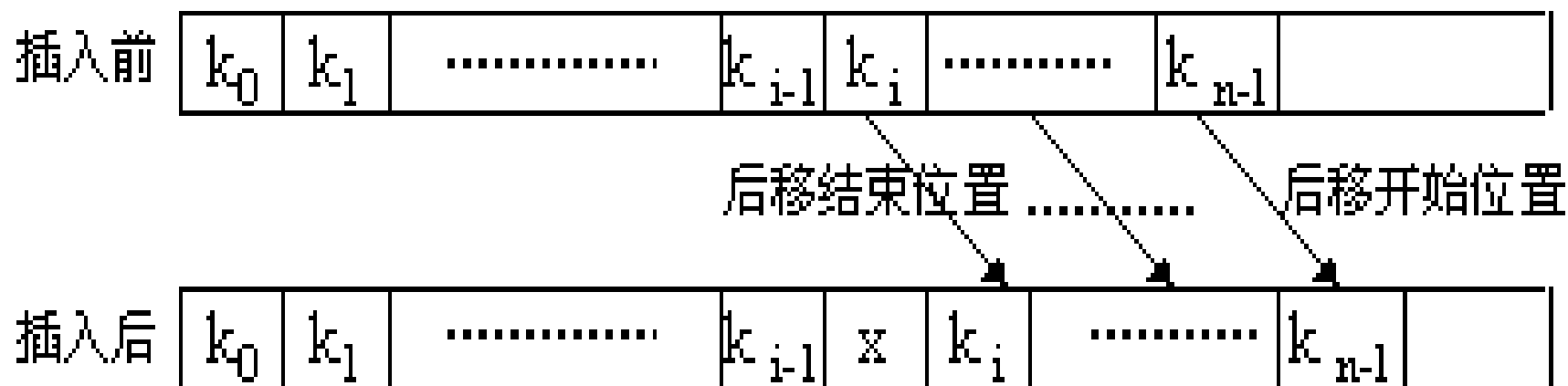
算法2.6取得顺序表中第i个结点的值

顺序表的插入运算是将一个值为 x 的结点插入到顺序表的第 i 个位置 $0 \leq i \leq n$ ，即将 x 插入到 k_{i-1} 和 k_i 之间，如果 $i=n$ ，则表示插入到表的最后，一般地可表示为：

插入前： $\{k_0, k_1, \dots, k_{i-1}, k_i, \dots, k_{n-1}\}$

插入后： $\{k_0, k_1, \dots, k_{i-1}, x, k_i, \dots, k_{n-1}\}$

插入过程的图示见下图：



```

void      insert_pos_sequence_list(sequence_list      *slt,int
position,datatype x)

{ int i;

  if(slt->size==MAXSIZE)

    {printf("\n顺序表是满的!没法插入!");exit(1);}

  if(position<0||position>slt->size)

    {printf("\n指定的插入位置不存在!");exit(1);}

  for(i=slt->size;i>position;i--) slt->a[i]=slt->a[i-1];

  slt->a[position]=x;

  slt->size++;

}

```

算法2.7在顺序表的position位置插入值为x的结点

算法2.7中，所花费的时间主要是元素后移操作，对于在第*i*个位置上插入一个新的元素，需要移动（*n-i*）个元素，设在第*i*个位置上插入一个元素的概率为 p_i ，且在任意一个位置上插入元素的概率相等，即 $p_0=p_1=p_2=\dots=p_n=1/(n+1)$ ，则在一个长度为*n*的顺序表中插入一个元素所需的平均移动次数为：

$$\sum_{i=0}^n p_i(n-i) = \sum_{i=0}^n \frac{1}{n+1}(n-i) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

即在长度为*n*的顺序表中插入一个元素平均需要移动表中的一半元素。该算法的时间复杂度为 $O(n)$ 。

顺序表的删除操作是指删除顺序表中的第 i 个结点， $0 \leq i \leq n-1$ ，一般地可表示为：

删除前： $\{k_0, k_1, \dots, k_{i-1}, k_i, k_{i+1}, \dots, k_{n-1}\}$

删除后： $\{k_0, k_1, \dots, k_{i-1}, k_{i+1}, \dots, k_{n-1}\}$

删除过程的图示见下图：



删除操作的具体实现见算法2.8

```
void delete_pos_sequence_list(sequence_list *slt,int position)
{
    int i;
    if(slt->size==0)
        {printf("\n顺序表是空的!");exit(1);}
    if(position<0||position>=slt->size)
        {printf("\n指定的删除位置不存在!");exit(1);}
    for(i=position;i<slt->size-1;i--) slt->a[i]=slt->a[i+1];
    slt->size--;
}
```

算法2.8删除顺序表中第position位置的结点

要删除顺序表中的第*i*个结点，则需要称动（*n-i-1*）个元素，设删除表中第*i*个结点的概率为 q_i ，且在表中每一个位置删除的概率相等，即：

$$q_0=q_1=\dots=q_{n-1}=1/n$$

则在一个长度为*n*的顺序表中删除一个结点的平均移动次数为：

$$\sum_{i=0}^{n-1} q_i (n-i-1) = \sum_{i=0}^{n-1} \frac{1}{n} (n-i-1) = \frac{1}{n} \times \frac{n(n-1)}{2} = \frac{n-1}{2}$$

这表明，在一个长为*n*的顺序表中删除一个元素平均需要移动表中大约一半的元素。该算法的时间复杂度为 $O(n)$ 。

第2章 线性表及其顺序存储

2.1 线性表概念

2.2 顺序表

2.2.1 顺序表

2.2.2 顺序表的实现

2.3 栈

2.3.1 栈概念

2.3.2 顺序栈及其实现

2.3.3 栈的应用之一-----括号匹配

2.4 队列

2.4.1 队列概念

2.4.2 顺序队列及其实现

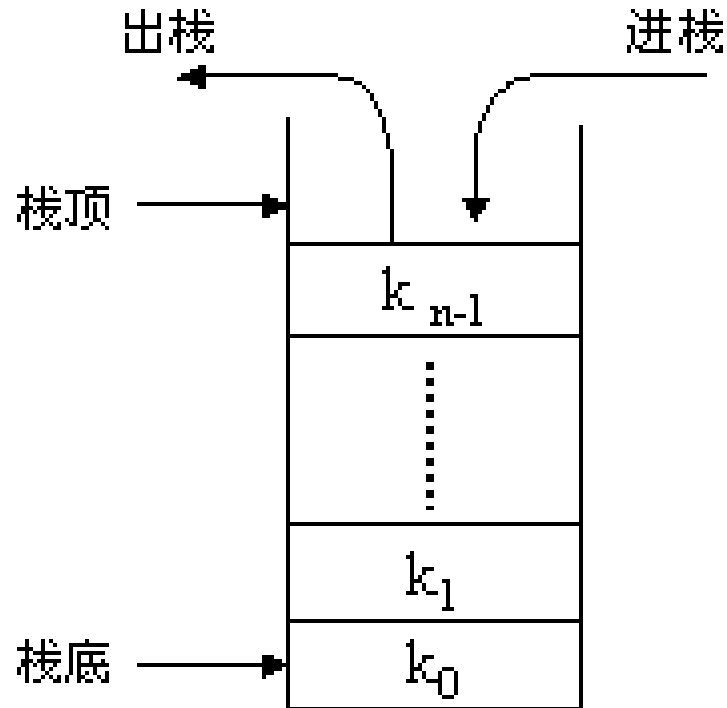
2.4.3 顺序循环队列及其实现

2.3 栈

2.3.1 栈

栈是一种特殊的线性表，对于这种线性表规定它的插入运算和删除运算均在线性表的同一端进行，进行插入和删除的那一端称为栈顶，另一端称为栈底。栈的插入操作和删除操作也分别简称进栈和出栈。

如果栈中有 n 个结点 $\{k_0, k_1, k_2, \dots, k_{n-1}\}$ ， k_0 为栈底， k_{n-1} 是栈顶，则栈中结点的进栈顺序为 $k_0, k_1, k_2, \dots, k_{n-1}$ ，而出栈的顺序为 $k_{n-1}, k_{n-2}, \dots, k_1, k_0$ 。如图所示。



栈具有后进先出或先进后出 (FILO, First In Last Out) 的性质

栈类型的描述如下：

ADT sequence_stack {

数据集合K: $K=\{k_1, k_2, \dots, k_n\}, n \geq 0$, K中的元素是datatype类型

数据关系R: $R=\{r\}$

$$r=\{ \langle k_i, k_{i+1} \rangle \mid i=1,2,\dots,n-1 \}$$

操作集合：

(1) void init_sequence_stack(sequence_stack *st) （顺序存储）
初始化

(2) int is_empty_stack(sequence_stack st) 判断栈（顺序存储）
是否为空

(3) void print_sequence_stack(sequence_stack st) 打印栈（顺序存储）
的结点值

(4) datatype get_top(sequence_stack st) 取得栈顶（顺序存储）结点值

(5) void push(sequence_stack *st,datatype x) 栈（顺序存储）的插入操作

(6) void pop(sequence_stack *st) 栈（顺序存储）的删除操作

} ADT sequence_stack

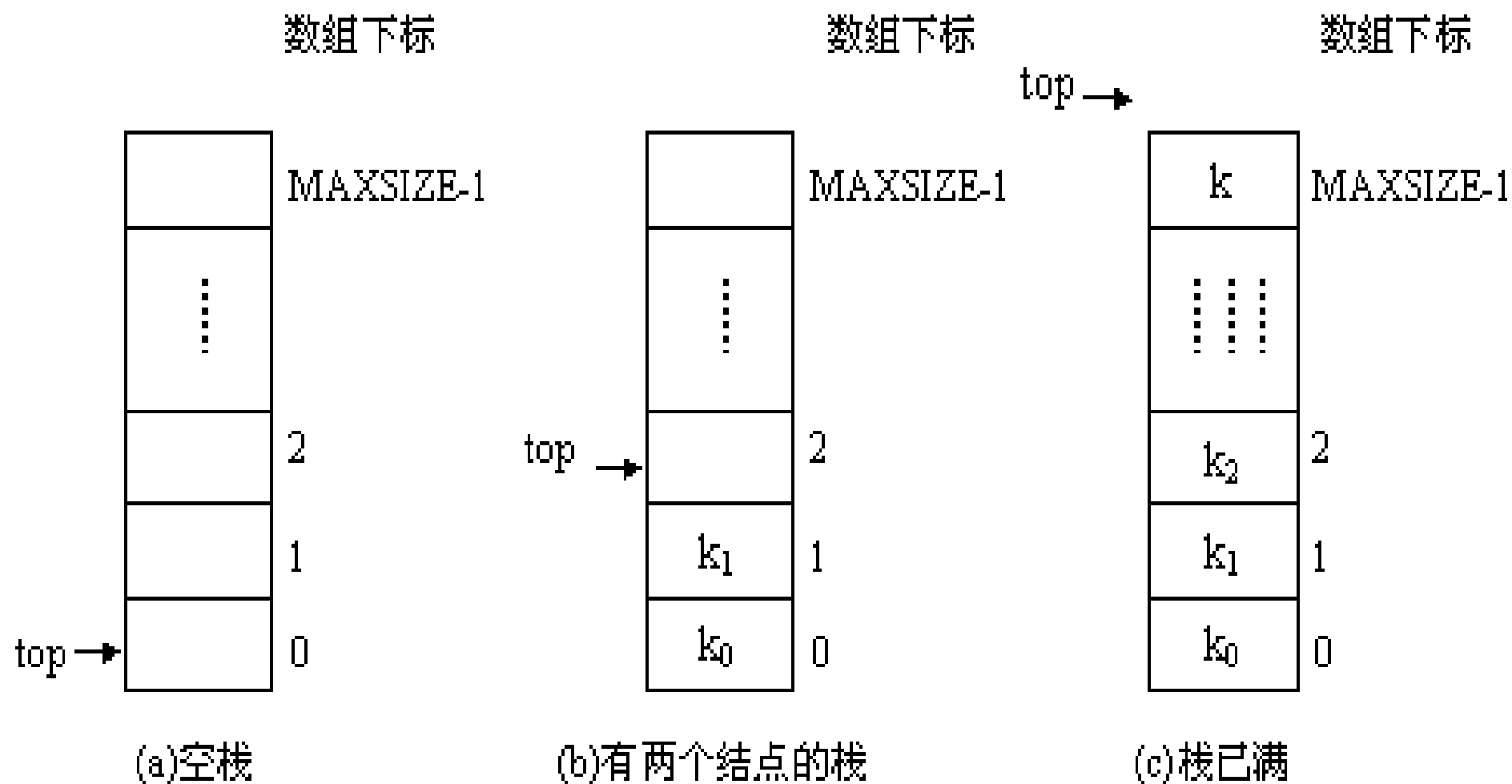
2.3.2 顺序栈及其实现

栈的实现方式一般有两种：顺序存储和链式存储。本小节将给出栈的顺序存储实现。

栈的顺序存储方式就是在顺序表的基础上对插入和删除操作限制它们在顺序表的同一端进行，所以同顺序表一样也可用一维数组表示。

一般地，可以设定一个足够大的一维数组存储栈，数组中下标为0的元素就是栈底，对于栈顶，可以设一个指针**top**指示它。

为了方便，设定**top**所指的位置是下一个将要插入的结点的存储位置，这样，当**top=0**时就表示是一个空的栈。一个栈的几种状态以及在这些状态下栈顶指针**top**和栈中结点的关系如下图所示。



栈的顺序存储结构的C语言描述如下：

```
/*  
*****/  
/*  栈（顺序存储）的头文件  */  
/*    文件名seqstack.h    */  
/*  
*****/  
  
#define MAXSIZE 100  
  
typedef int datatype;  
  
typedef struct{  
    datatype a[MAXSIZE];  
    int top;  
}sequence_stack;
```

下面是顺序存储栈的几个基本操作的具体实现

```
/*  
    栈（顺序存储）初始化  
    文件名seqsinit.c, 函数名init_sequence_stack()  
*/  
void init_sequence_stack(sequence_stack *st)  
{  
    st->top=0;  
}
```

算法2.9栈（顺序存储）初始化

```
/*  
判断栈（顺序存储）是否为空  
文件名seqsempt.c, 函数名is_empty_sequence_stack() */  
/*  
*****  
  
int is_empty_stack(sequence_stack st)  
{  
    return(st.top? 0:1);  
}
```

算法2.10判断栈（顺序存储）是否为空

```

/*****/

/*      取得栈顶（顺序存储）结点值      */
/*      文件名seqsfirs.c, 函数名get_top()      */
/*****/

datatype get_top(sequence_stack st)
{
    if (empty_stack(st))
        {printf("\n栈是空的!");exit(1);}
    else
        return st.a[st.top-1];
}

```

算法2.11取得栈顶（顺序存储）结点值

```

/*****/

/*      栈（顺序存储）的插入操作      */

/*      文件名seqspush.c, 函数名push()      */

/*****/

void push(sequence_stack *st,datatype x)
{
    if(st->top==MAXSIZE)
        {printf("\nThe sequence stack is full!");exit(1);}
    st->a[st->top]=x;
    st->top++;
}

```

算法2.12 栈（顺序存储）的插入操作

```
/*  
    栈（顺序存储）的删除操作  
    文件名seqspop.c, 函数名pop()  
*/  
  
void pop(sequence_stack *st)  
{  
    if(st->top==0)  
        {printf("\nThe sequence stack is empty!");exit(1);}  
    st->top--;  
}
```

算法2.13栈（顺序存储）的删除操作

2.3.3 栈的应用之一-----括号匹配

设一个表达式中可以包含三种括号：小括号、中括号和大括号，各种括号之间允许任意嵌套，如小括号内可以嵌套中括号、大括号，但是不能交叉。举例如下：

([]{}) 正确的

([()]) 正确的

{([])} 正确的

{[()] } 不正确的

{()}[] 不正确的

如何去检验一个表达式的括号是否匹配呢？大家知道当自左向右扫描一个表达式时，凡是遇到的开括号（或称左括号）都期待有一个和它对应的闭括号（或称右括号）与之匹配。

按照括号正确匹配的规则，在自左向右扫描一个表达式时，后遇到的开括号比先遇到的开括号更加期待有一个闭括号与之匹配。

可能会连续遇到多个开括号，且它们都期待寻求匹配的闭括号，所以必须将遇到的开括号存放好。又因为后遇到的开括号的期待程度高于其先前遇到的开括号的期待程度，所以应该将所遇到的开括号存放于一个栈中。这样，当遇到一个闭括号时，就查看这个栈的栈顶结点，如果它们匹配，则删除栈顶结点，如果不匹配，则说明表达式中括号是不匹配的，如果扫描它整个表达式后，这个栈是空的，则说明表达式中的括号是匹配的，否则表达式的括号是不匹配的。

判断表达式括号是否匹配的具体实现见算法2.14。

```

/*****
/*      判断表达式括号是否匹配      */
/*      文件名seqmatch.c， 函数名match_huohao()      */
*****/

```

```
int match_kuohao(char c[])
{
    int i=0;
    sequence_stack s;
    init_sequence_stack(&s);
    while(c[i]!='#')
    {
        switch(c[i])
        {
            case '{':
            case '[':
            case '(':push(&s,c[i]);break;
```

```

case '}':if(!is_empty_sequence_stack(s)&&get_top(s)=='{')
        {pop(&s);break;}        else return 0;
case ']':if(!is_empty_sequence_stack(s)&&get_top(s)=='[')
        {pop(&s);break;}        else return 0;
case ')':if(!is_empty_sequence_stack(s)&&get_top(s)=='(')
        {pop(&s);break;}        else return 0;
}
i++;
}
return (is_empty_sequence_stack(s));/*栈空则匹配， 否则不匹配*/
}

```

算法2.14判断表达式括号是否匹配

2.3.4 栈的应用之二-----算术表达式求值

第2章 线性表及其顺序存储

2.1 线性表概念

2.2 顺序表

2.2.1 顺序表

2.2.2 顺序表的实现

2.3 栈

2.3.1 栈概念

2.3.2 顺序栈及其实现

2.3.3 栈的应用之一-----括号匹配

2.4 队列

2.4.1 队列概念

2.4.2 顺序队列及其实现

2.4.3 顺序循环队列及其实现

2.4 队列

2.4.1 队列

队列是一种特殊的线性表，它的特殊性在于队列的插入和删除操作分别在表的两端进行。插入的那一端称为队尾，删除的那一端称为队首。队列的插入操作和删除操作也分别简称进队和出队。

对于一个队列：

$$\{k_0, k_1, k_2, \dots, k_{n-1}\}$$

如果 k_0 那端是队首， k_{n-1} 那端是队尾，则 k_0 是这些结点中最先插入的结点，若要做删除操作， k_0 将首先被删除，所以说，队列是具有“先进先出”（FIFO, First In First Out）特点的线性结构。

队列类型的描述如下：

ADT sequence_queue {

数据集合K: $K=\{k_1, k_2, \dots, k_n\}, n \geq 0$, K中的元素是datatype类型

数据关系R: $R=\{r\}$

$$r=\{ \langle k_i, k_{i+1} \rangle \mid i=1,2,\dots,n-1 \}$$

操作集合：

(1) void init_sequence_queue(sequence_queue *sq) 队列（顺序存储）初始化

(2) int is_empty_sequence_queue(sequence_queue sq) 判断队列（顺序存储）是否为空

(3) void print_sequence_queue(sequence_queue sq) 打印队列（顺序存储）的结点值

(4) datatype get_first(sequence_queue sq) 取得队列（顺序存储）的队首结点值

(5) void insert_sequence_queue (sequence_queue *sq,datatype x) 队列（顺序存储）插入操作

(6) void delete_sequence_queue(sequence_queue *sq) 队列（顺序存储）的删除操作

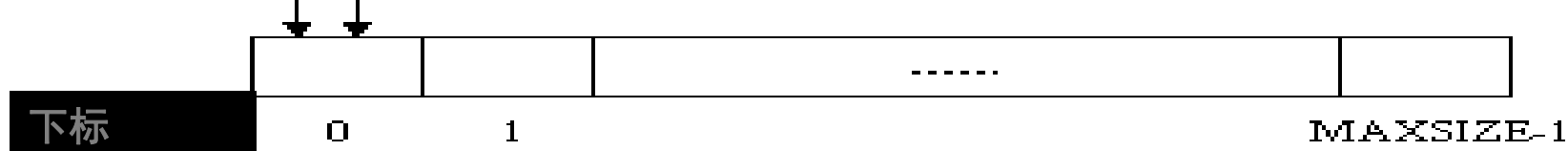
} ADT sequence_queue;

2.4.2 顺序队列及其实现

队列的顺序存储在C语言中可以用一维数组表示，为了标识队首和队尾，需要附设两个指针**front**和**rear**，**front**指示的是队列中最前面，即队首结点在数组中元素的下标，**rear**指示的是队尾结点在数组中元素的下标的下一个位置，也就是说**rear**指示的是即将插入的结点在数组中的下标。

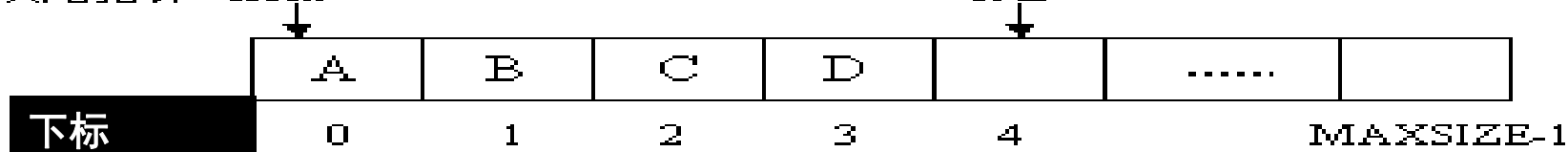
队列的几种状态：

队首、队尾指针 front rear



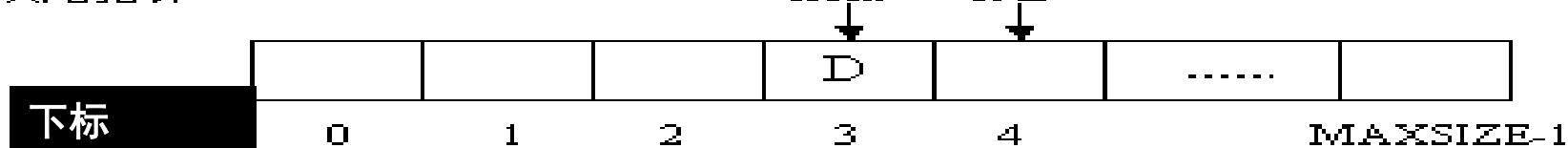
(a) 初始状态---空队列

队首、队尾指针 front



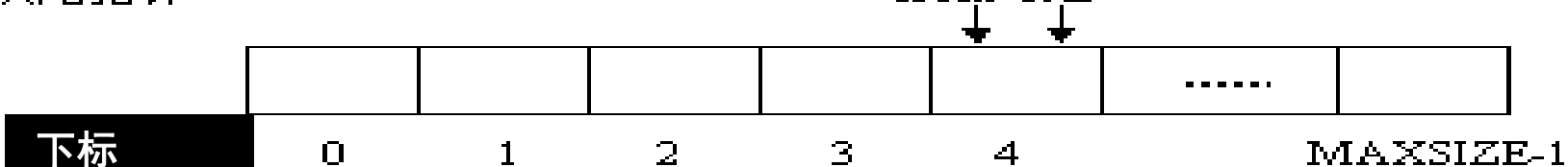
(b)连续插入几个结点后的状态

队首、队尾指针



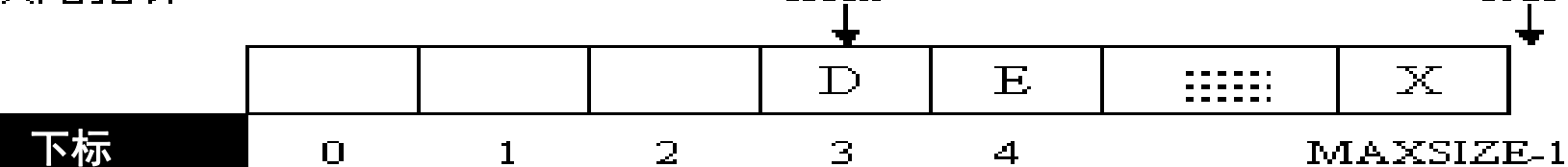
(c)连续删除几个结点后的状态---此时队列中只有一个结点

队首、队尾指针



(d) 在(c)状态下再删除一个结点后的状态---空队列

队首、队尾指针



(e)连续插入若干结点后的状态---此时队列呈现满的状态，但数组前部有空位子

队列的顺序存储结构的C语言描述如下：

```
/*  
/* 队列（顺序存储）的头文件 */  
/* 文件名seqqueue.h */  
/*  
  
#define MAXSIZE 100  
  
typedef int datatype;  
  
typedef struct{  
    datatype a[MAXSIZE];  
    int front;  
    int rear;  
}sequence_queue;
```

顺序存储队列的几个基本操作的具体实现：

```
/******  
/*      队列（顺序存储）初始化      */  
/* 文件名seqqinit.c, 函数名init_sequence_queue() */  
/******  
  
void init_sequence_queue(sequence_queue *sq)  
{  
    sq->front=sq->rear=0;  
}
```

算法2.20队列（顺序存储）初始化

```
/*  
判断队列（顺序存储）是否为空  
*/  
/*文件名seqqempt.c, 函数名is_empty_sequence_queue() */  
/*  
  
int is_empty_sequence_queue(sequence_queue sq)  
{  
    return (sq.front==sq.rear? 1:0);  
}
```

算法2.21判断队列（顺序存储）是否为空

```

/*****
/*      打印队列（顺序存储）的结点值      */
/*  文件名seqqprin.c, 函数名print_sequence_queue()  */
*****/

void print_sequence_queue(sequence_queue sq)
{
    int i;
    if(is_empty_sequence_queue(sq))
    {
        printf("\n顺序队列是空的!");
    }
    else
        for(i=sq.front;i<sq.rear;i++) printf("%5d",sq.a[i]);
}

```

算法2.22打印队列（顺序存储）的结点值

```

/*****/
/*      队列（顺序存储）的插入操作      */
/*  文件名seqqinse.c, 函数名insert_sequence_queue()  */
/*****/

void insert_sequence_queue(sequence_queue *sq,datatype x)
{
    int i;

    if(sq->rear==MAXSIZE)
        {printf("\n顺序循环队列是满的!");exit(1);}

    sq->a[sq->rear]=x;
    sq->rear=sq->rear+1;
}

```

算法2.24队列（顺序存储）的插入操作

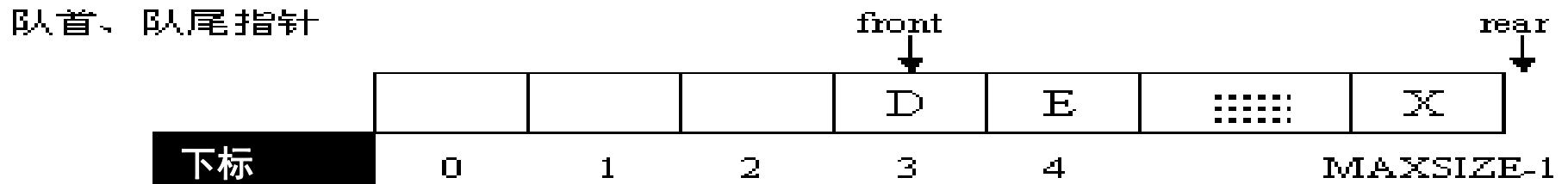

```

/*****
/*      队列（顺序存储）的删除操作      */
/*  文件名seqqdele.c, 函数名delete_sequence_queue()  */
*****/
void delete_sequence_queue(sequence_queue *sq)
{
    if(sq->front==sq->rear)
    {
        printf("\n顺序队列是空的！ 不能做删除操作！ ");  exit(1);
    }
    sq->front++;
}

```

算法2.25队列（顺序存储）的删除操作

在队列的几种状态图的（e）状态中，队列是一种队满状态，将不能再插入新的结点，而实际上数组的前部还有许多空的位置。为了充分地利用空间，可以将队列看作一个循环队列，在数组的前部继续作插入运算，这就是循环队列。

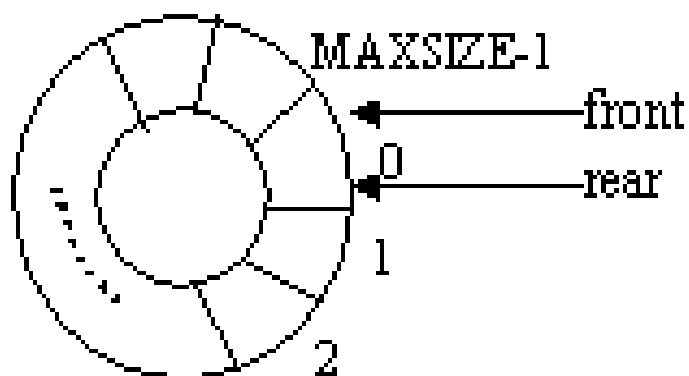


(e)连续插入若干结点后的状态---此时队列呈现满的状态，但数组前部有空位子

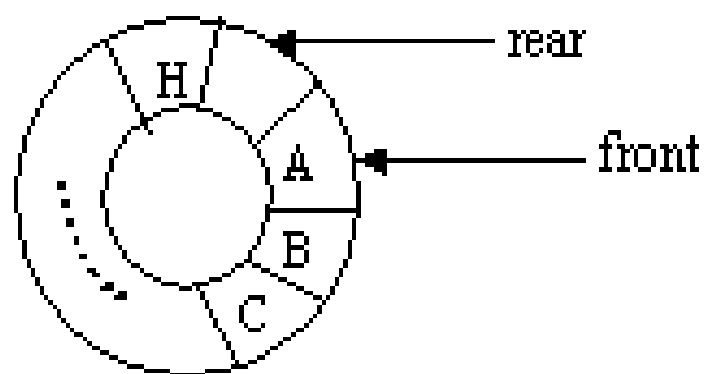
2.4.3 顺序循环队列及其实现

给定一个大小为MAXSIZE的数组存储一个队列时，经过若干次插入和删除操作后，当队尾指针 $rear = MAXSIZE$ 时，呈现队列满的状态，而事实上数组的前部可能还有空闲的位置。为了有效利用空间，将顺序存储的队列想象为一个环状，把数组中的最前和最后两个元素看作是相邻的，这就是循环队列。

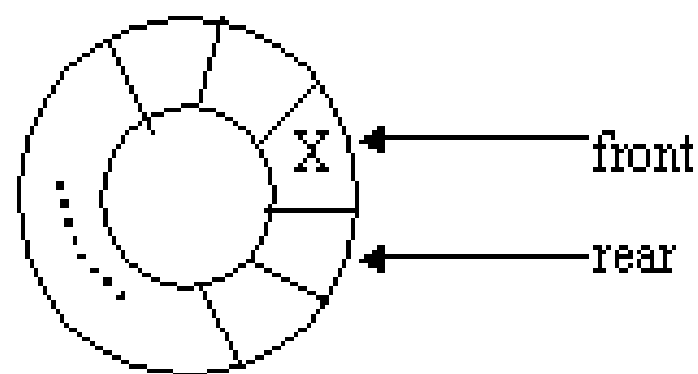
循环队列的几种状态表示：



(a)初始状态---空的循环队列

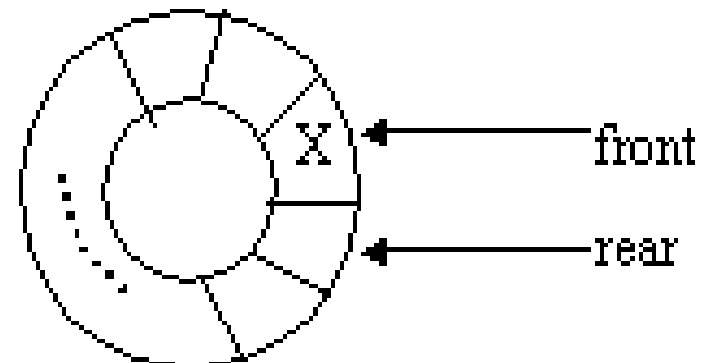
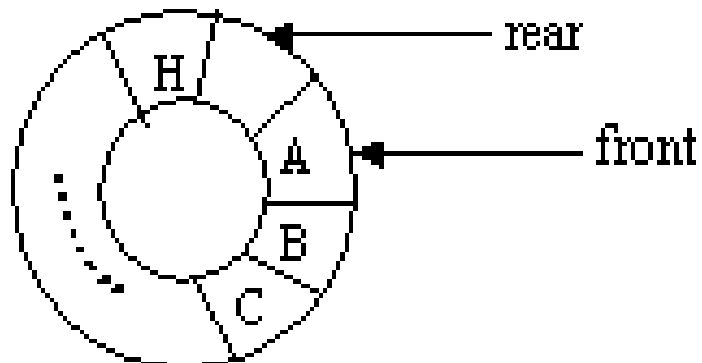


(b) 剩余一个空间的状态



(c) 循环队列中只有一个结点的状态

在 (b) 状态中, 如果再插入一个新的结点, 则数组空间将被全部占用, 队列已满, 且 $\text{rear}=\text{front}$, 而在 (c) 状态中, 若删除一个结点队列成为空队列, 此时也有 $\text{rear}=\text{front}$, 这就是说循环队列满与空的条件都是 $\text{rear}=\text{front}$ 。



解决方法是牺牲一个数组元素的空间, 即若数组的大小是 MAXSIZE , 则该数组所表示的循环队列最多允许存储 $\text{MAXSIZE}-1$ 个结点。这样, 循环队列满的条件是

$(\text{rear}+1)\% \text{MAXSIZE}=\text{front}$

循环队列空的条件是 $\text{rear}=\text{front}$

循环队列的插入与删除操作的实现：

```
/* **** */
/*      循环队列（顺序存储）的插入操作      */
/*  文件名seqinst.c, 函数名insert_sequence_cqueue()  */
/* **** */

void insert_sequence_cqueue(sequence_queue *sq, datatype x)
{
    int i;
    if((sq->rear+1)%MAXSIZE==sq->front)
        {printf("\n顺序循环队列是满的！ 无法进行插入操作！ ");exit(1);}
    sq->a[sq->rear]=x;
    sq->rear=(sq->rear+1)%MAXSIZE;
}
```

算法2.27循环队列（顺序存储）的插入操作

```

/*****
/*      循环队列（顺序存储）的删除操作      */
/*  文件名secqdele.c, 函数名delete_sequence_cqueue()  */
*****/

void delete_sequence_cqueue(sequence_queue *sq)
{
    if(sq->front==sq->rear)
    {
        printf("\n循环队列是空的！ 无法进行删除！ ");    exit(1);
    }
    sq->front=(sq->front+1)%MAXSIZE;
}

```

算法2.28循环队列（顺序存储）的删除操作

第2章 线性表及其顺序存储

2.1 线性表概念

2.2 顺序表

2.2.1 顺序表

2.2.2 顺序表的实现

2.3 栈

2.3.1 栈概念

2.3.2 顺序栈及其实现

2.3.3 栈的应用之一-----括号匹配

2.4 队列

2.4.1 队列概念

2.4.2 顺序队列及其实现

2.4.3 顺序循环队列及其实现

作业： 1, 2, 3, 4, 6, 7

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

3.7.1 链式队列

3.7.2 链式队列的实现

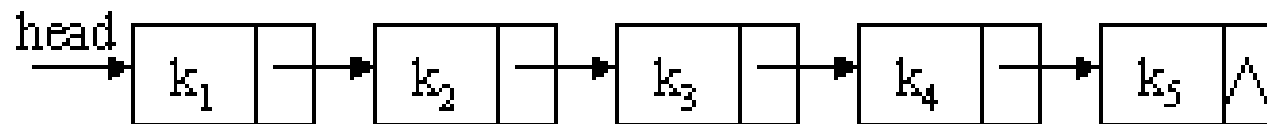
3.1链式存储

数据结构的存储方式必须体现它的逻辑关系。在链式存储方式下，实现中除存放一个结点的信息外，还需附设指针，用指针体现结点之间的逻辑关系。如果一个结点有多个后继或多个前驱，那么可以附设相应个数的指针，一个结点附设的指针指向的是这个结点的某个前驱或后继。

线性结构中，每个结点最多只有一个前驱和一个后继，这里暂且设定更关心它的后继，这样在存储时除了存放该结点的信息外，只要附设一个指针即可，该指针指向它的后继结点的存放位置。每个结点的存储形式是：

info	next
------	------

为了清晰，左图可以更简洁地用下图表示。



存储地址	info	next
1000		
1001	k ₁	1003
1002		
1003	k ₂	1007
1004		
1005	k ₄	1006
1006	k ₅	∧
1007	k ₃	1005
1008		

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

3.7.1 链式队列

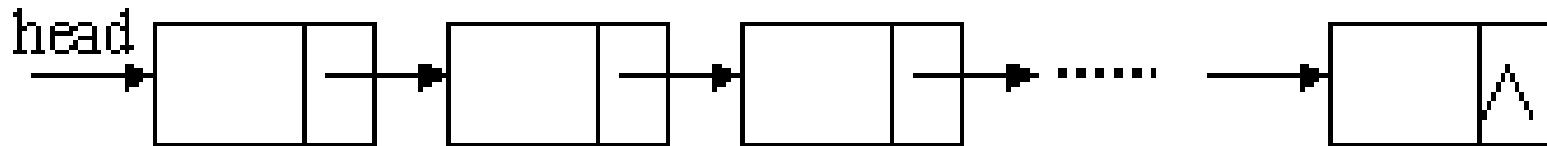
3.7.2 链式队列的实现

3.2 单链表

单链表是线性表链式存储的一种形式，其中的结点一般含有两个域，一个是存放数据信息的**info**域，另一个是指向该结点的后继结点的存放地址的指针域**next**。一个单链表必须有一个首指针指向单链表中的第一个结点。图3.3给出了空的单链表和非空的单链表的图示。

head → ^

(a) 空的单链表



(b) 一个非空的单链表

ADT link_list{

(6) node *insert_x_after_y(node *head,datatype x,datatype y)

在单链表中值为y的结点后插入一个值为x的新结点

(7) node *insert_x_after_i(node *head,datatype x,int i)

在单链表中第i个结点后插入一个值为x的新结点

(8) node *delete_num_link_list(node *head,datatype x) 在单链表中删除一个值为x的结点

(9) node *delete_pos_link_list(node *head,int i) 在单链表中删除第i个结点

} ADT link_list;

3.2.2单链表的实现

单链表结构的C语言描述如下：

```
/*  
/*链表实现的头文件，文件名slnklist.h */  
/*  
  
typedef int datatype;  
typedef struct link_node{  
    datatype info;  
    struct link_node *next;  
}node;
```

单链表几个基本操作的具体实现：

```
/*  
/*          建立一个空的单链表          */  
/*  文件名slnkinit.c, 函数名init_link_list()      */  
/*  
/*  
/*          建立一个空的单链表          */  
node *init_link_list() /*建立一个空的单链表*/  
{  
    return NULL;  
}
```

算法3.1建立一个空的单链表


```

/*****
/*      输出单链表中各个结点的值      */
/*      文件名slnkprin.c, 函数名print_link_list()      */
*****/

void print_link_list(node *head)
{
    node *p;

    p=head;

    if(!p) printf("\n单链表是空的！");

    else

        { printf("\n单链表各个结点的值为： \n");
          while(p) { printf("%5d",p->info);p=p->next;}
        }
}

```

算法3.2输出单链表中各个结点的值

```

/*****/

/*      在单链表中查找一个值为x的结点      */

/*  文件名slmkfinx.c, 函数名find_num_link_list()  */

/*****/

node *find_num_link_list(node *head,datatype x)
{
    node *p;
    p=head;
    while(p&& p->info!=x) p=p->next;
    return p;
}

```

算法3.3在单链表中查找一个值为x的结点

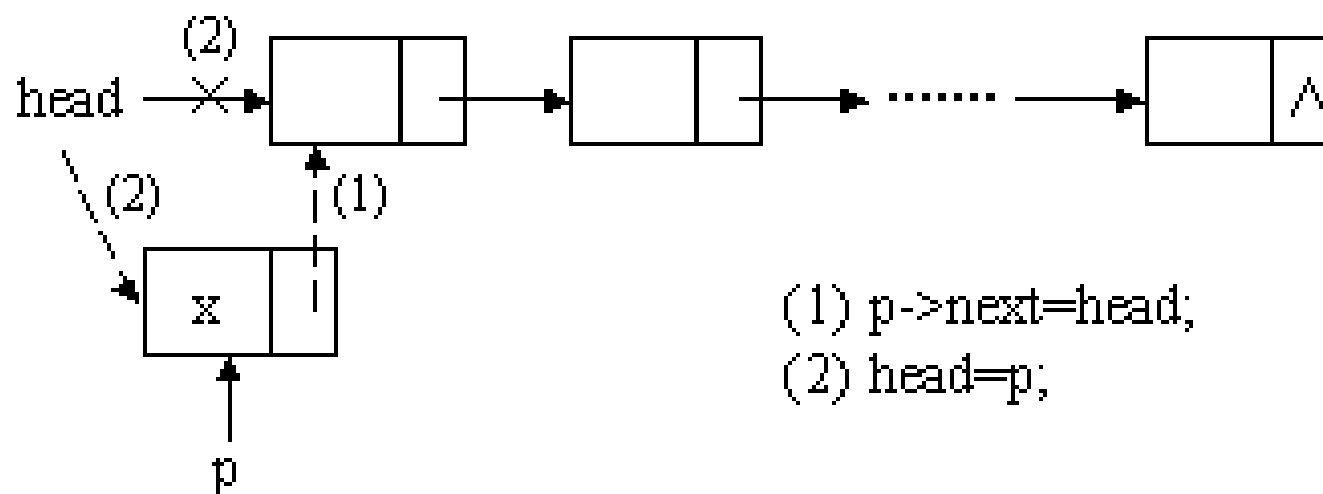
```

/*****
/*      在单链表中查找第i个结点      */
/*      文件名slnkfini.c, 函数名find_pos_link_list()      */
*****/
node *find_pos_link_list(node *head,int i)
{
    int j=1;
    node *p=head;
    if(i<1){printf("\nError!\n");exit(1);}
    while(p&& i!=j) { p=p->next ; j++; }
    return p;
}

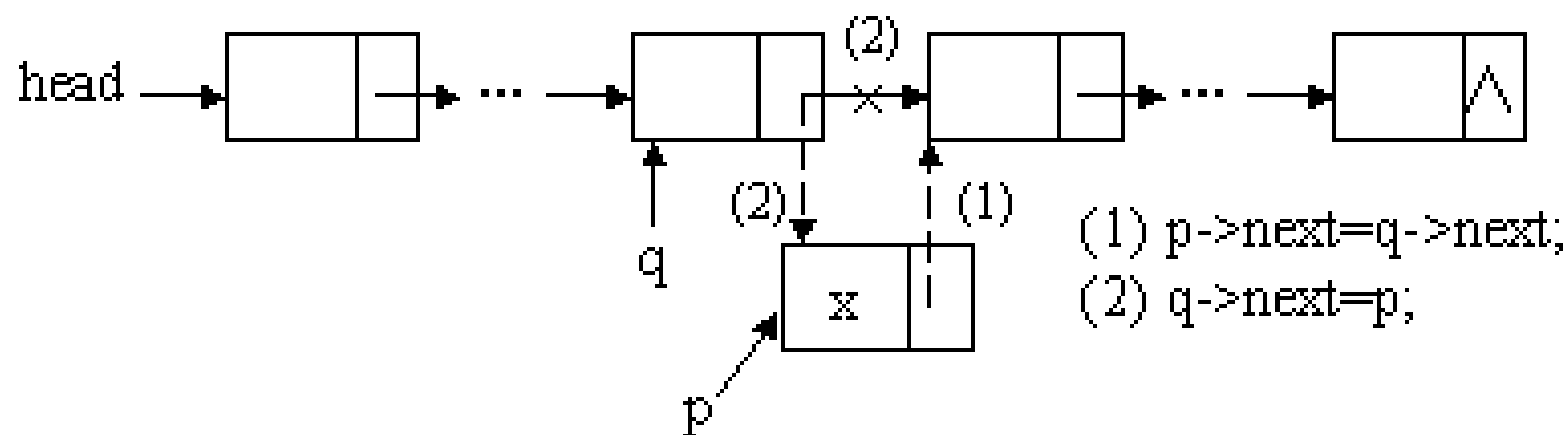
```

算法3.4在单链表中查找第i个结点

单链表的插入过程见下图所示：



(a) 在单链表的最前面插入一个值为 x 的新结点



(b) 在 q 所指的结点后插入一个 p 所指的值为 x 的新结点

```

/*****/
/*  插入一个值为x的结点作为单链表的第一个结点  */
/*  文件名slnkinfx.c, 函数名insert_in_front_link_list()  */
/*****/
node *insert_in_front_link_list(node *head,datatype x)
{
    node *p;

    p=(node*)malloc(sizeof(node)); /*分配空间*/

    p->info=x;                      /*设置新结点的值*/

    p->next=head;                   /*插入(1)*/

    head=p;                        /*插入(2)*/

    return head;

}

```

算法3.5插入一个值为x的结点作为单链表的第一个结点

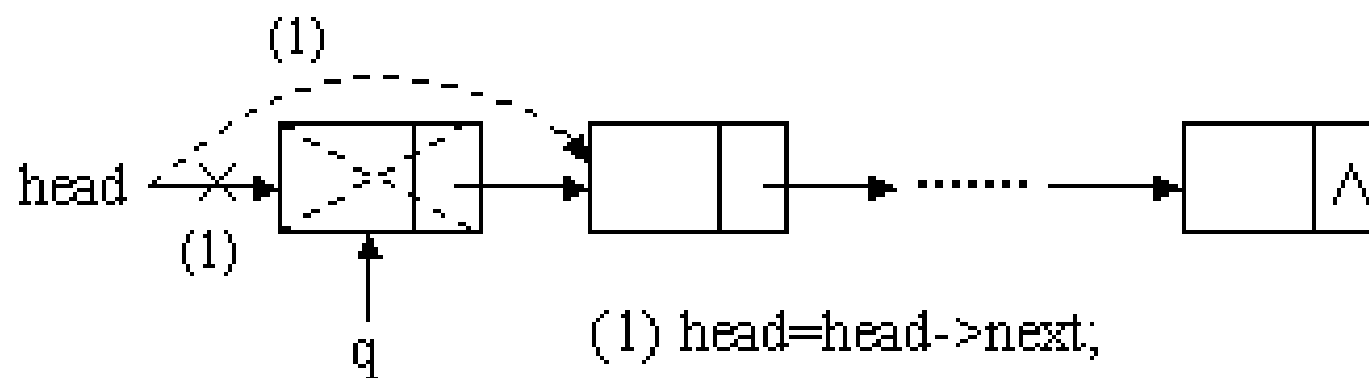
```

/*****/
/* 在单链表中第i个结点后插入一个值为x的新结点 */
/* 文件名slnkinix.c, 函数名insert_x_after_i() */
/*****/
node *insert_x_after_i(node *head,datatype x,int i)
{
    node *p,*q;
    q=find_pos_link_list(head,i);/*查找第i个结点*/
    if(!q)
    {printf("\n找不到第%d个结点，不能进行插入！ ",i,x);exit(1);}
    p=(node*)malloc(sizeof(node));/*分配空间*/
    p->info=x;/*设置新结点*/
    p->next=q->next;/*插入(1)*/
    q->next=p;/*插入(2)*/
    return head;
}

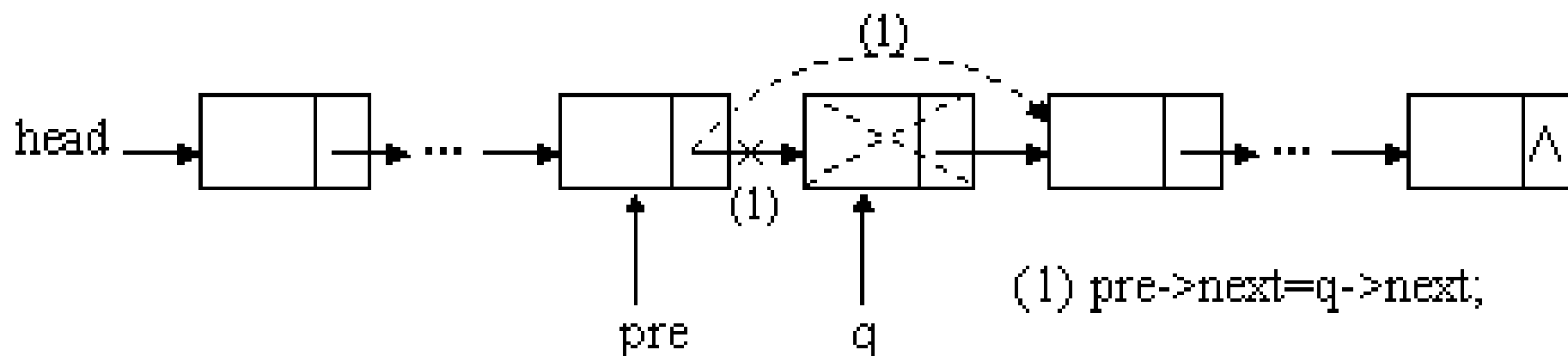
```

算法3.7在单链表中第i个结点后插入一个值为x的新结点

删除操作见下图所示：



(a) 删除单链表的最前面的（第一个）结点



(b) 删除`q`指向的结点，`pre`为`q`的前驱结点

```

node *delete_num_link_list(node *head,datatype x)
{
    node *pre=NULL,*p;
    if(!head) {printf("单链表是空的！");return head;}
    p=head;
    while(p&& p->info!=x)/*没有找到并且没有找完*/
        {pre=p;p=p->next;}/*pre指向p的前驱结点*/
    if(!pre&&p->info==x)/*要删除的是第一个结点*/
        head=head->next;/*删除(1)*/
    else
        pre->next=p->next;
    free(p);
    return head;
}

```

链式存储的插入和删除
操作比顺序存储方便，但不能
随机访问某个结点！

算法3.8在单链表中删除一个值为x的结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

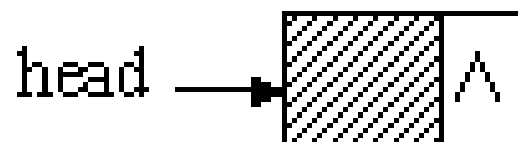
3.7.1 链式队列

3.7.2 链式队列的实现

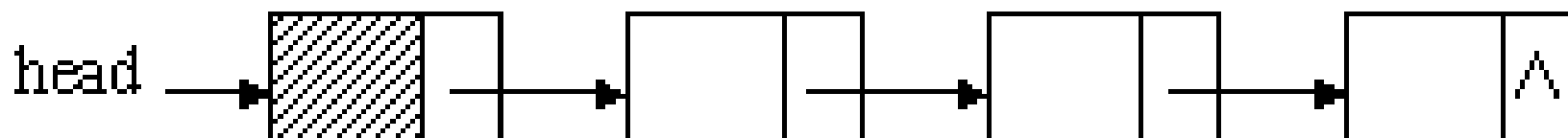
3.3带头结点单链表

3.3.1带头结点单链表

带头结点单链表见下图所示：



(a) 空的带头结点的单链表



(b) 含有三个结点的带头结点的单链表

ADT hlink_list{

(5) node *insert_in_front_hlink_list(node *head,datatype x)

插入一个值为x的结点作为带头结点单链表的第一个结点

(6) node *insert_x_after_y(node *head,datatype x,datatype y)

在带头结点单链表中值为y的结点后插入一个值为x的新结点

(7) node *insert_x_after_i(node *head,datatype x,int i)

在带头结点单链表中第i个结点后插入一个值为x的新结点

(8) node *delete_num_hlink_list(node *head,datatype x)

在带头结点单链表中删除一个值为x的结点

(9) node *delete_pos_hlink_list(node *head,int i)

在带头结点单链表中删除第i个结点

} ADT hlink_list;

3.3.2带头结点单链表的实现

一般的单链表中，第一个结点由**head**指示，而在带头结点单链表中，**head**指示的是所谓的头结点，它不是存储数据结构中的实际结点，第一个实际的结点是**head->next**指示的。在带头结点单链表的操作实现时要注意这一点。

```
node *init_hlink_list()
{
    node *head;
    head=(node*)malloc(sizeof(node));
    head->next=NULL;
    return head;
}
```

算法3.10建立一个空的带头结点单链表

```
void print_hlink_list(node *head)
{
    node *p;
    p=head->next;/*从第一个（实际）结点开始*/
    if(!p) printf("\n带头结点单链表是空的!");
    else
    {
        printf("\n带头结点的单链表各个结点的值为: \n");
        while(p) { printf("%5d",p->info);p=p->next;}
    }
}
```

算法3.11输出带头结点单链表中各个结点的值

```

/*****/

/*    在带头结点单链表中查找一个值为x的结点    */
/*    文件名hlinkfinx.c， 函数名find_num_hlink_list()    */
/*****/

node *find_num_hlink_list(node *head,datatype x)
{
    node *p;
    p=head->next;/*从第一个（实际）结点开始*/
    while(p&& p->info!=x) p=p->next;
    return p;
}

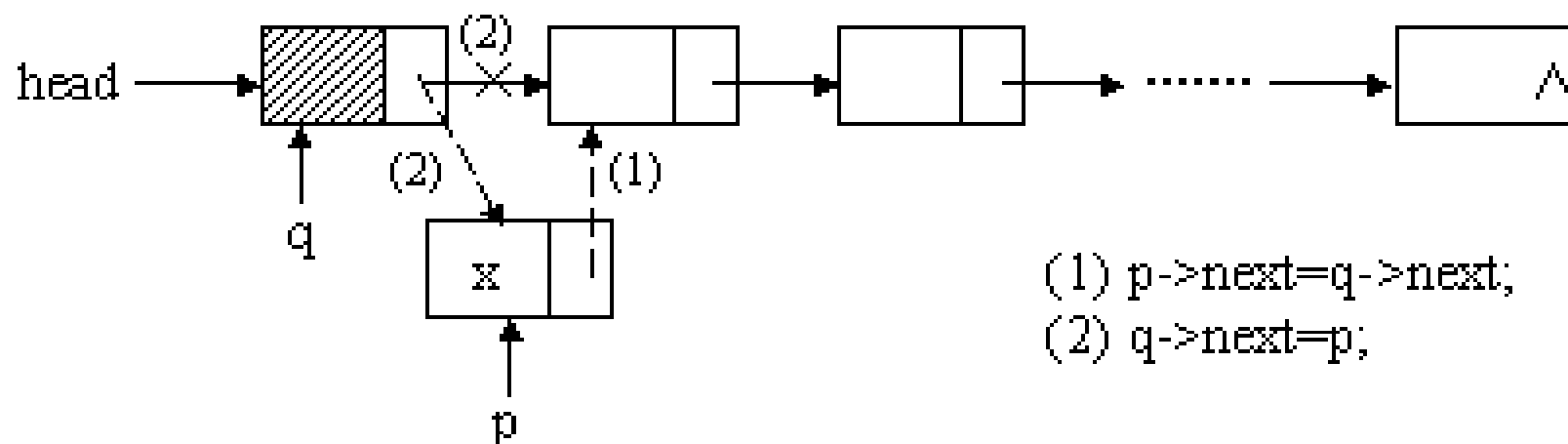
```

算法3.12在带头结点单链表中查找一个值为x的结点

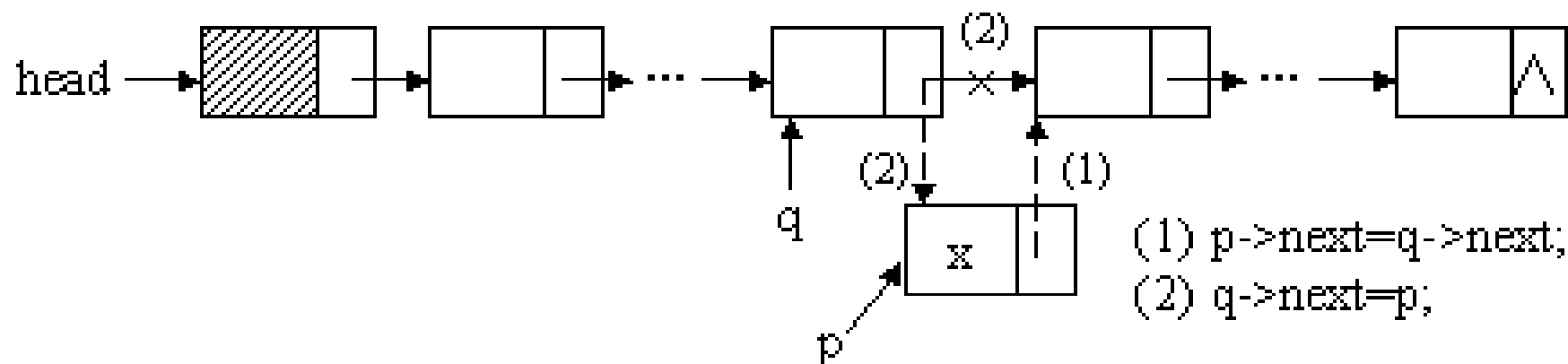
```
node *find_pos_hlink_list(node *head,int i)
{
    int j=0;
    node *p=head;
    if(i<0){printf("\n带头结点的单链表中不存在第%d个结点!",i);return NULL;}
    while(p&& i!=j)/*没有查找完并且还没有找到*/
    {
        p=p->next;j++;/*继续向后（左）查找，计数器加1*/
    }
    return p;/*返回结果，i=0时，p指示的是头结点*/
}
```

算法3.13在带头结点单链表中查找第i个结点

带头结点单链表的插入过程见图3.7：



(a) 非空带头结点单链表的最前面插入一个值为 x 的新结点



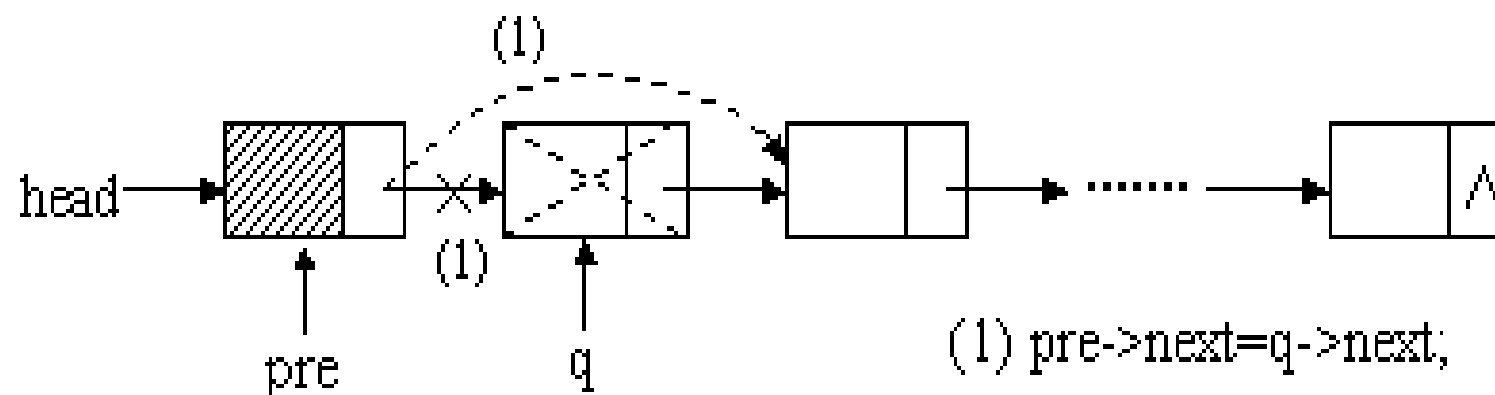
(b) 在 q 所指的结点后插入一个 p 所指的值为 x 的新结点

带头结点单链表的几个插入操作的具体实现见算法3.14~算法3.16。

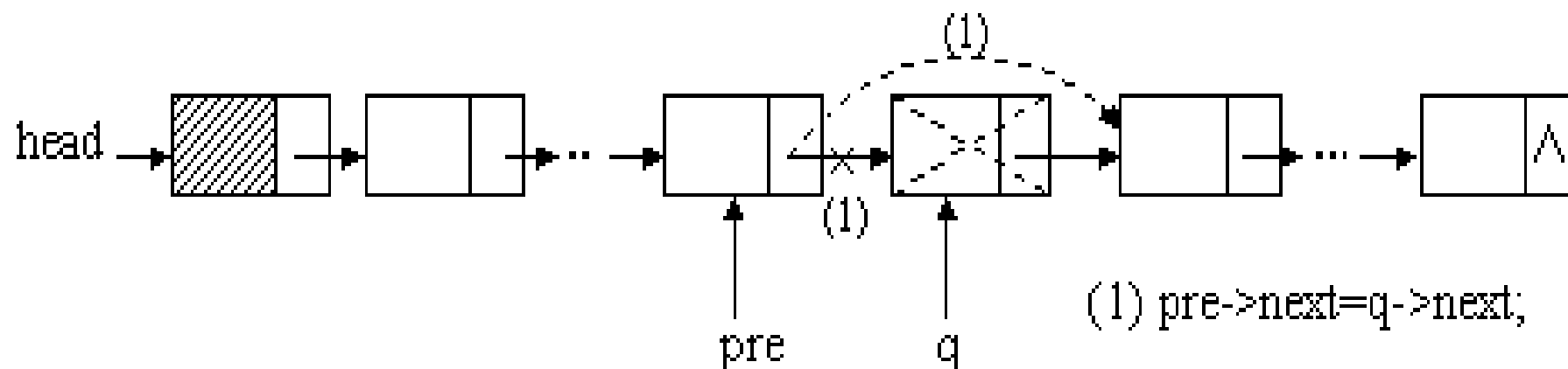
```
node *insert_x_after_y(node *head,datatype x,datatype y)
{
    node *p,*q;
    q=find_num_hlink_list(head,y);/*查找值为y的结点*/
    if(!q)/*没有找到*/
    {printf("\n没有找到值为%d的结点，不能插入%d!",y,x);return head;}
    p=(node*)malloc(sizeof(node));/*为准备插入的新结点分配空间*/
    p->info=x;/*为新结点设置值x*/
    p->next=q->next;/*插入(1)*/
    q->next=p;/*插入(2)*/
    return head;
}
```

算法3.15在带头结点单链表中值为y的结点后插入一个值为x的新结点

带头结点单链表的删除过程见图3.8。



(a) 删除带头结点单链表的最前面的（第一个）实际结点



(b) 在带头结点单链表中删除q指向的结点，pre为q的前驱结点

```
node *delete_num_hlink_list(node *head,datatype x)
{
    node *pre=head,*q;/*首先pre指向头结点*/
    q=head->next;/*q从带头结点的第一个实际结点开始找值为x
的结点*/
    while(q&&q->info!=x)/*没有查找完并且还没有找到*/
    {pre=q;q=q->next;}/*继续查找， pre指向q的前驱*/
    pre->next=q->next;/*删除*/
    free(q);/*释放空间*/
    return head;
}
```

算法3.17在带头结点单链表中删除一个值为x的结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

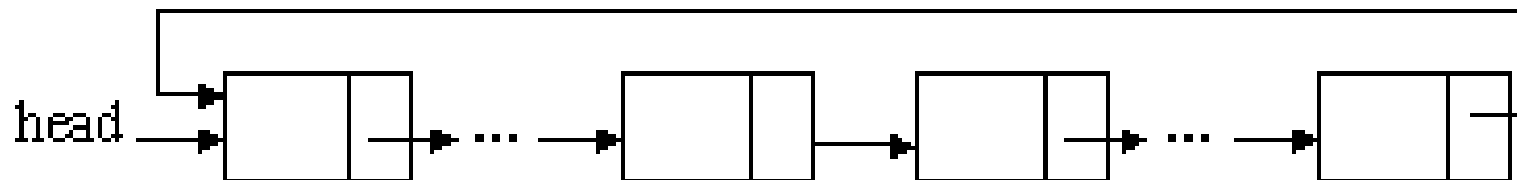
3.7.1 链式队列

3.7.2 链式队列的实现

3.4 循环单链表

3.4.1 循环单链表

无论是单链表，还是带头结点单链表，从表中的某个结点开始，只能访问到这个结点及其后面的结点，不能访问到它前面的结点，除非再从首指针指示的结点开始访问。如果希望从表中的任意一个结点开始，都能访问到表中的所有其它结点，可以设置表中最后一个结点的指针域指向表中的第一个结点，这种链表称为循环单链表。



循环单链表类型的描述（略）

3.4.2 循环单链表的实现

单链表中某个结点p是表中最后一个结点的特征是 $p \rightarrow next == NULL$ 。对于一个循环单链表，若首指针为 head，表中的某个结点p是最后一个结点的特征应该是 $p \rightarrow next == head$ 。

循环单链表的头文件和单链表的相同。

```
/*  
    建立一个空的循环单链表  
    文件名clnkinit.c, 函数名init_clink_list()  
*/  
/*  
    建立一个空的循环单链表  
*/  
node *init_clink_list() /*建立一个空的循环单链表*/  
{  
    return NULL;  
}
```

算法3.19建立一个空的循环单链表

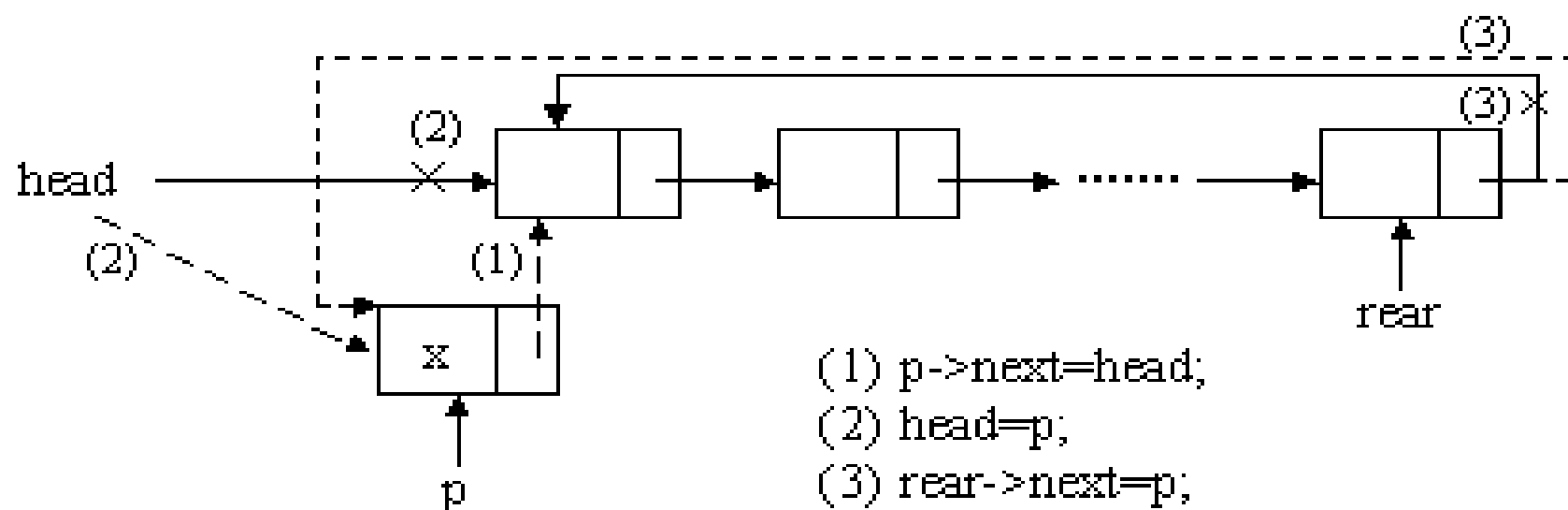

```

void print_clink_list(node *head)
{ node *p;
  if(!head) printf("\n循环单链表是空的！ \n");
  else
  {printf("\n循环单链表各个结点的值分别为:\n");
    printf("%5d",head->info);/*输出非空表中第一个结点的值*/
    p=head->next;/*p指向第一个结点的下一个结点*/
    while(p!=head)/*没有回到第一个结点*/
    {printf("%5d",p->info);
      p=p->next;
    }
  }
}

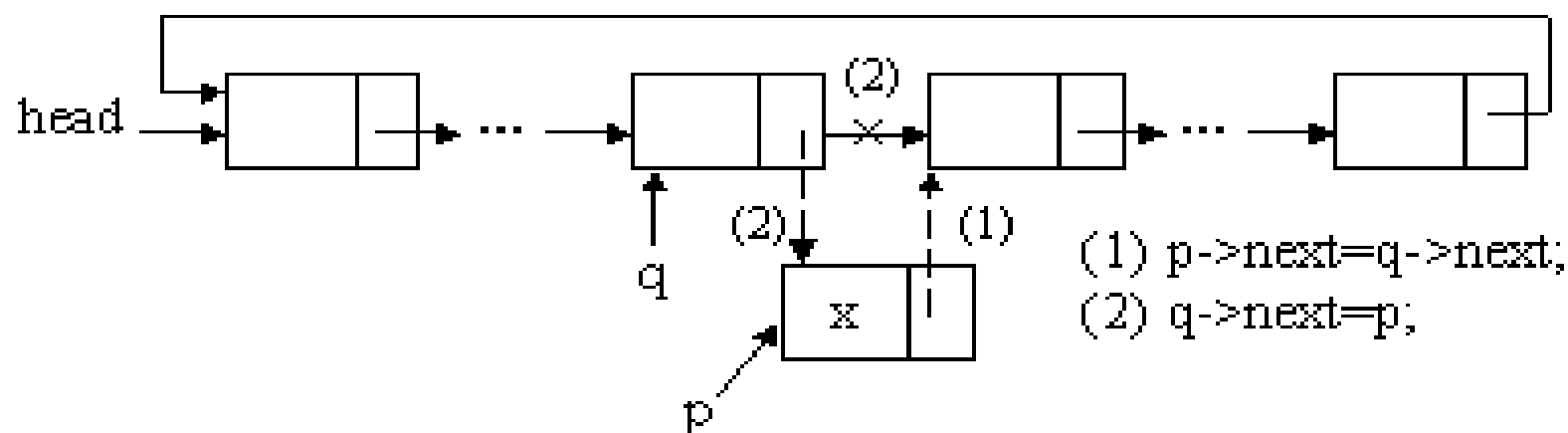
```

算法**3.21**输出循环单链表中各个结点的值

循环单链表的插入过程如图：



(a) 在循环单链表的最前面插入一个值为 x 的新结点



(b) 循环单链表，在 q 所指的结点后插入一个 p 所指的值为 x 的新结点

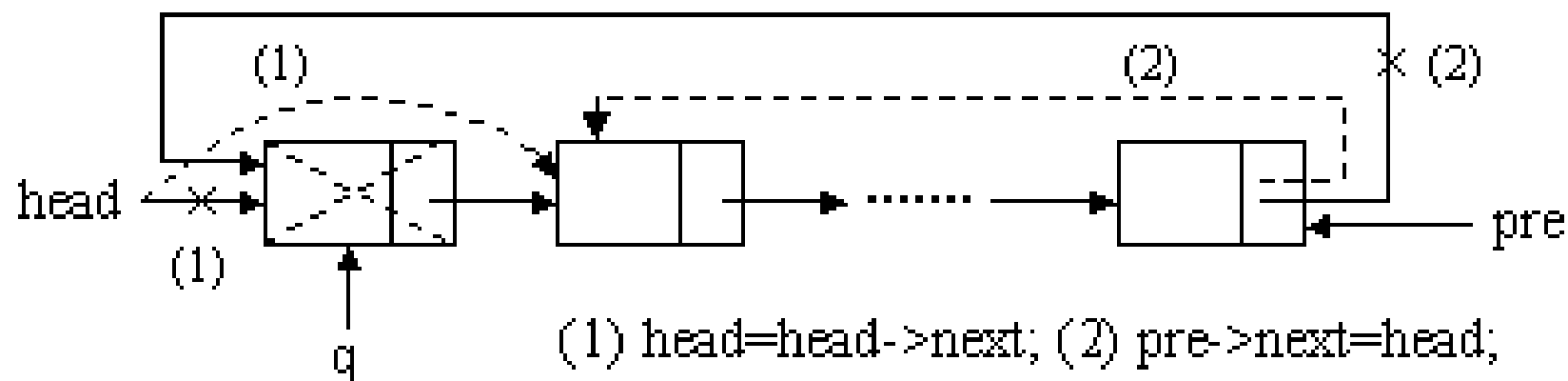
```

node *insert_x_after_i(node *head,datatype x,int i)
{ node *p,*q;
  q=find_pos_clink_list(head,i);/*查找第i个结点， q指向第i个结点*/
  if(!q)/*没有找到， 则不进行插入*/
    printf("\n表中不存在第%d个结点， 无法进行插入!\n",i);
  else
  { /*找到了第i个结点， 准备插入x*/
    p=(node*)malloc(sizeof(node));/*分配空间*/
    p->info=x;/*设置新结点的值*/
    p->next=q->next;/*插入， 修改指针(1)*/
    q->next=p;/*插入， 修改指针(2)*/
  }
  return head;
}

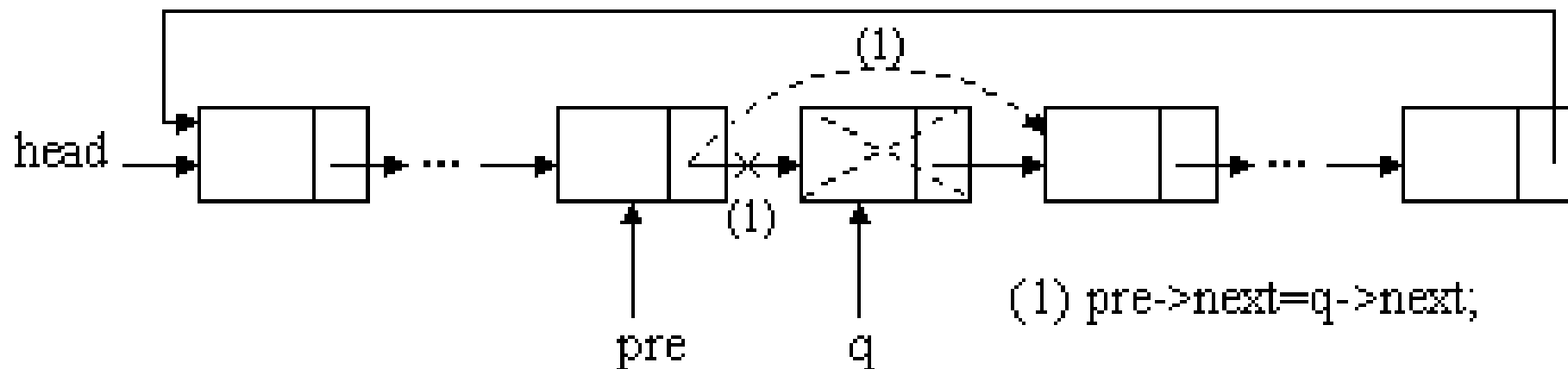
```

算法3.26在循环单链表中第i个结点后插入一个值为x的新结点

循环单链表的删除过程如图：



(a) 删除循环单链表的最前面的（第一个）结点



(b) 删除q指向的结点，pre为q的前驱结点（q指向的不是循环单链表的第一个结点）

```

node *delete_num_clink_list(node *head,datatype x)
{ node *pre=NULL,*q;/*q用于查找值为x的结点， pre指向q的前驱结点*/
  if(!head)/*表为空， 则无法做删除操作*/
  { printf("\n循环单链表为空， 无法做删除操作！ ” ); return NULL; }
  q=head;/*从第1个结点开始准备查找*/
  while(q->next!=head&&q->info!=x)/*没有找遍整个表并且没有找到*/
  { pre=q; q=q->next;/*pre为q的前驱， 继续查找*/
    }/*循环结束后， pre为q的前驱*/
  if(q->info!=x)/*没找到*/ { printf("没有找到值为%d的结点！ ",x); }
  else /*找到了， 下面要删除q*/
  { pre->next=q->next;/*删除q指向的结点*/ free(q);/*释放空间*/ }
  return head;
}

```

算法3.27在循环单链表中删除一个值为x的结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

3.7.1 链式队列

3.7.2 链式队列的实现

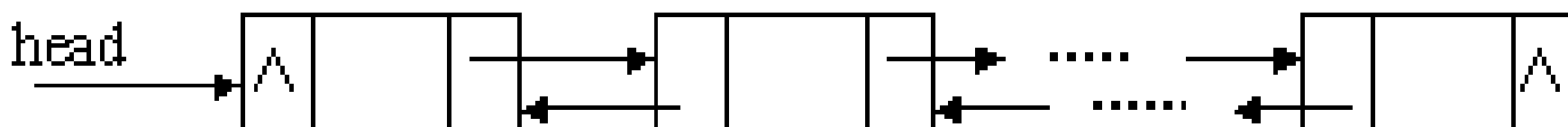
3.5 双链表

3.5.1 双链表

前面的各种链式表中，一个结点的指针域是指向它的后继结点的，如果需要找一个结点 p 的前驱结点，则必须从表首指针开始查找，当某个结点 pre 的指针域指向的是结点 p 时，即 $pre \rightarrow next == p$ 时，则说明 pre 是 p 的前驱结点。如果常常需要知道一个结点的前驱和后继结点，上述的链式表是不适合的。既然单链表中每个结点有一个指针域指向它的后继结点，那自然地想到再增设一个指针域指向它的前驱结点，这就构成了双链表。

双链表的结点包括三个域，一个是存放数据信息的info域，另外两个是指针域，这里用llink和rlink表示，llink指向它的前驱结点，rlink指向它的后继结点。

双链表的一般情形如图所示：



双链表类型的描述（略！）

3.5.2 双链表的实现

双链表结构的C语言描述如下：

```
/*  
*****/  
/* 双链表的头文件，文件名dlinklist.h /  
/*  
*****/  
  
typedef int datatype;  
  
typedef struct dlink_node{  
    datatype info;  
    struct dlink_node *llink,*rlink;  
}dnode;
```

```
void print_dlink_list(dnode *head)
{
    dnode *p;
    printf("\n"); p=head;
    if(!p) printf("\n双链表是空的!\n");
    else
    {
        printf("\n双链表中各个结点的值为: \n");
        while(p) { printf("%5d",p->info);p=p->rlink;}
    }
}
```

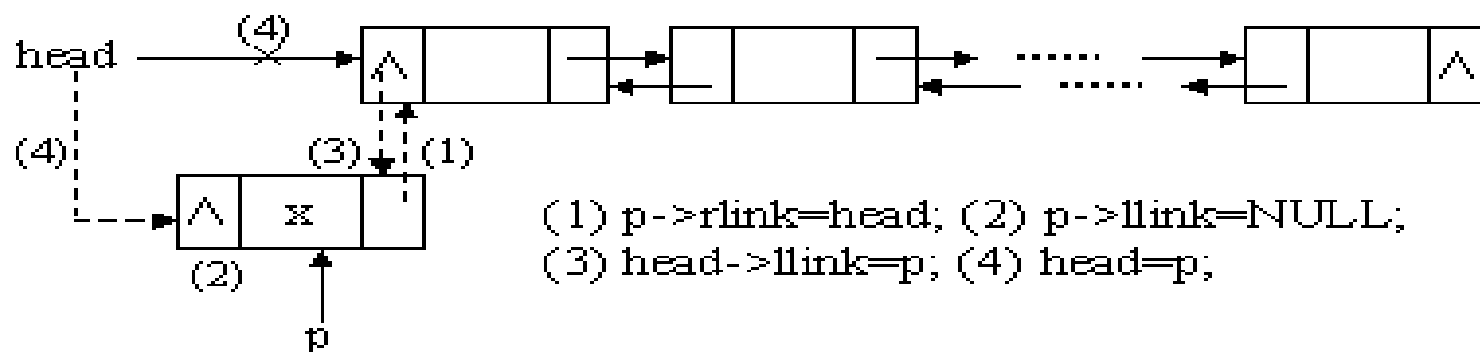
算法3.30 输出双链表中各个结点的值

```

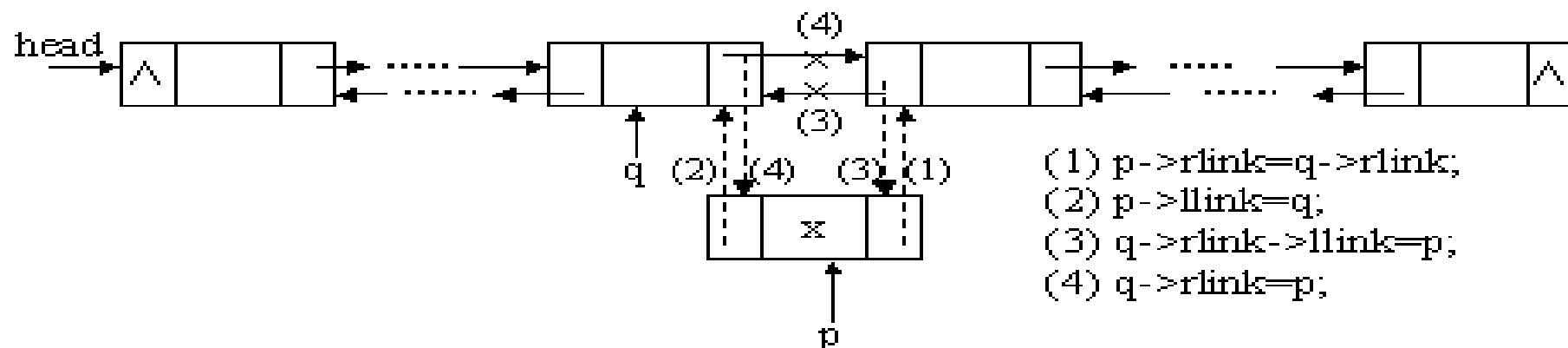
dnode *find_pos_dlink_list(dnode *head,int i)
{
    int j=1;
    dnode *p=head;
    if(i<1){printf("\n第%d个结点不存在!\n",i);return NULL;}
    while(p&& i!=j)/*没有找完整个表并且没有找到*/
    { p=p->rlink;j++;/*继续沿着右指针向后查找，计数器加1*/ }
    if(!p){printf("\n第%d个结点不存在!\n",i);return NULL;}
    return p;
}

```

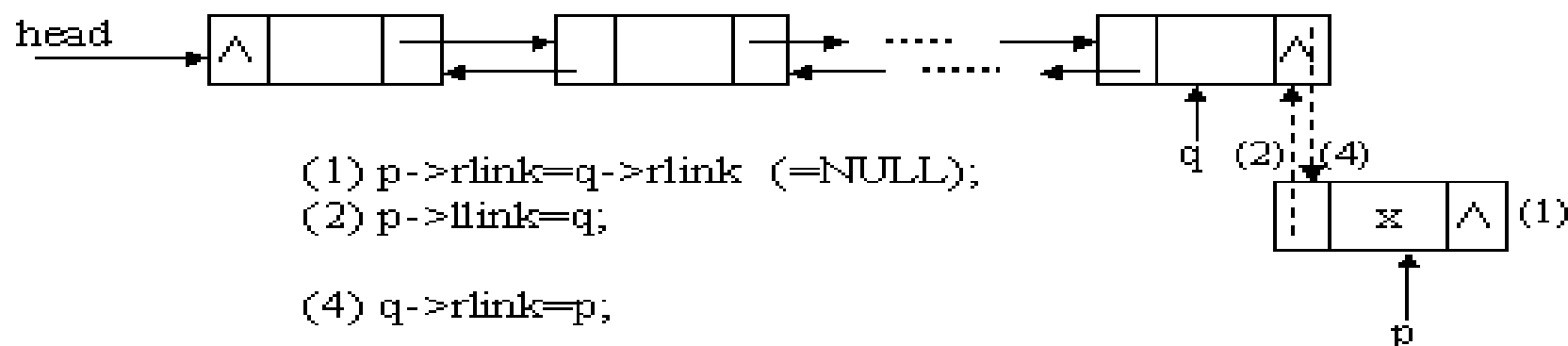
算法3.32查找双链表中第i个结点



(a) 在双链表的最前面插入一个值为 x 的新结点



(b) 在双链表中 q 所指结点的后面插入一个值为 x 的新结点



(c) 在双链表中 q 所指结点 (是最后一个结点) 的后面插入一个值为 x 的新结点

/* 在双链表中第i个结点后插入一个值为x的新结点 */

dnode *insert_x_after_i(dnode *head,datatype x,int i)

{

dnode *p,*q;

p=(dnode*)malloc(sizeof(dnode));/*分配空间*/

p->info=x;/*设置新结点的值*/

if(i==0)/*在最前面插入一个值为x的新结点*/

{ p->llink=NULL;/*新插入的结点没有前驱*/

p->rlink=head;/*新插入的结点的后继是原来双链表中的第一个结点*/

head=p;/*新结点成为双链表的第一个结点*/

return head;

}

```
q=find_pos_dlink_list(head,i);/*查找第i个结点*/
```

```
if(!q)/*第i个结点不存在*/
```

```
{printf("第%d个结点不存在，无法进行插入",i);return head;}
```

```
if(q->rlink==NULL)/*在最后一个结点后插入*/
```

```
{
```

```
    p->rlink=q->rlink;/*即为NULL，新插入的结点没有后继。插入操作(1)*/
```

```
    p->llink=q;/*插入操作(2)*/
```

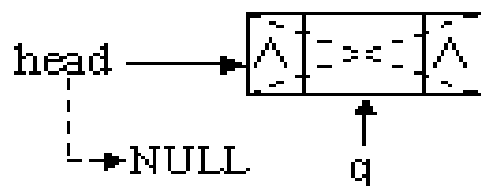
```
    q->rlink=p;/*插入操作(4)*/
```

```
    }/*注意不能和下面的一般情况一样处理，这里如执行下面的(3)将出错！*/
```

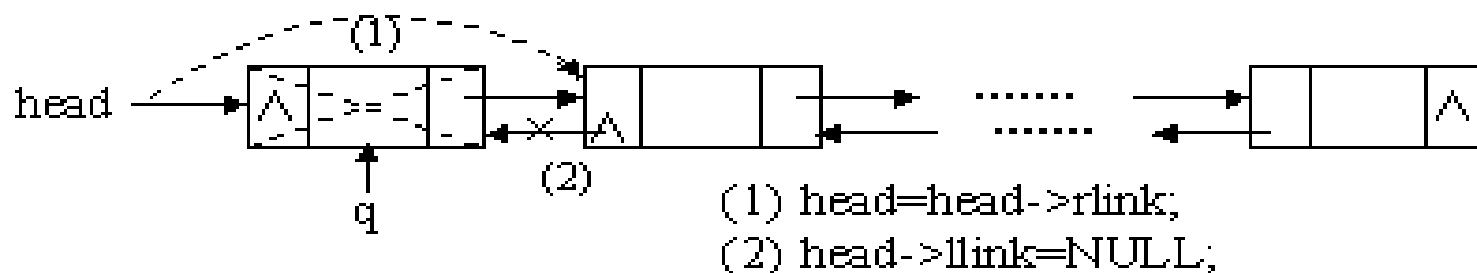
```
else/*一般情况下的插入*/
{
    p->rlink=q->rlink;/*插入操作(1)*/
    p->llink=q;/*插入操作(2)*/
    q->rlink->llink=p;/*插入操作(3)*/
    q->rlink=p;/*插入操作(4)*/
}
return head;
}
```

算法3.35 在双链表中第i个结点后插入一个值为x的新结点

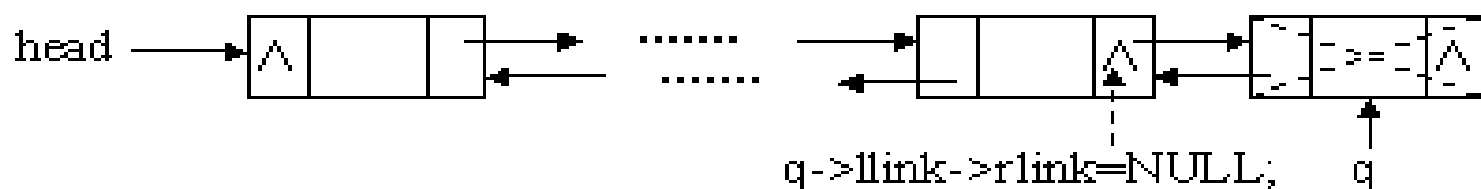
8



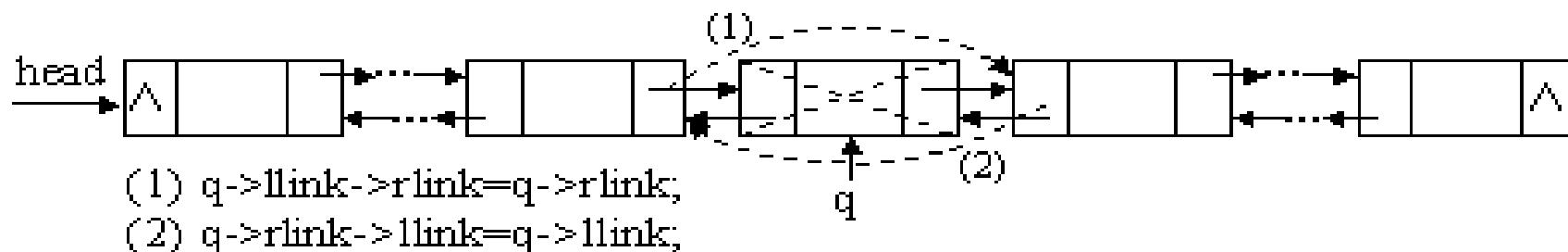
(a) 被删除的 q 是双链表中唯一的一个结点



(b) 被删除的 q 是双链表中的第一个结点（表中不只这一个结点）



(c) 被删除的 q 是双链表中的最后一个结点



(d) 被删除的 q 是双链表中既非第一也非最后的一个结点


```
/*      在双链表中删除一个值为x的结点      */
```

```
dnode *delete_num_dlink_list(dnode *head,datatype x)
```

```
{ dnode *q;
```

```
    if(!head)/*双链表为空，无法进行删除操作*/
```

```
        {printf("双链表为空，无法进行删除操作");return head;}
```

```
    q=head;
```

```
    while(q&&q->info!=x) q=q->rlink;/*循环结束后q指向的是值为x  
的结点*/
```

```
    if(!q)
```

```
    {
```

```
        printf("\n没有找到值为%d的结点！ 不做删除操作！ ",x);
```

```
    }
```

if(q==head&&head->rlink)/*被删除的结点是第一个结点并且表中不只一个结点*/

```
{  
    head=head->rlink;  
    head->llink=NULL;  
    free(q);return head;  
}
```

if(q==head&&!head->rlink)/*被删除的结点是第一个结点并且表中只有这一个结点*/

```
{  
    free(q);  
    return NULL;/*双链表置空*/  
}
```

else

```
{ if(!q->rlink)/*被删除的结点是双链表中的最后一个结点*/
```

```
{ q->llink->rlink=NULL; free(q); return head; }
```

```
else/*q是在一个有2个以上结点的双链表中的一个非开始、也非终端结点*/
```

```
{
```

```
q->llink->rlink=q->rlink;
```

```
q->rlink->llink=q->llink;
```

```
free(q); return head;
```

```
}
```

```
}
```

```
}
```

算法3.36在双链表中删除一个值为x的结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

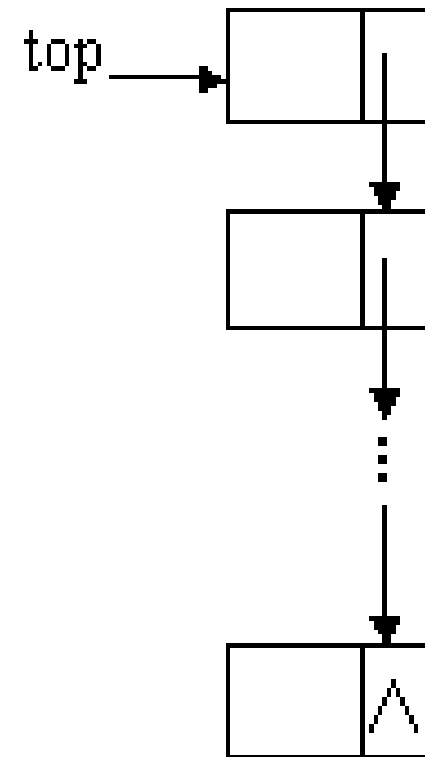
3.7.1 链式队列

3.7.2 链式队列的实现

3.6 链式栈

3.6.1 链式栈

栈的链式存储称为链式栈。链式栈就是一个特殊的单链表，对于这特殊的单链表，它的插入和删除规定在单链表的同一端进行。链式栈的栈顶指针一般用 **top** 表示，链式栈如下图所示。



链式栈类型的描述如下：

ADT link_stack{

数据集合K: $K=\{k_1, k_2, \dots, k_n\}, n \geq 0$, K中的元素是datatype类型

数据关系R: $R=\{r\}$

$r=\{ \langle k_i, k_{i+1} \rangle \mid i=1, 2, \dots, n-1 \}$

操作集合:

(1) node *init_link_stack() 建立一个空链式栈

(2) int empty_link_stack(node *top) 判断链式栈是否为空

(3) datatype get_top(node *top) 取得链式栈的栈顶结点值

(4) void print_link_stack(node *top) 输出链式栈中各个结点的值

(5) node *push_link_stack(node *top,datatype x) 向链式栈中插入一个值为x的结点

node *pop_link_stack(node *top)

/*删除链式栈的栈顶结点*/

} ADT link_stack;

3.6.2 链式栈的实现

```
datatype get_top(node *top)
{
    if(!top) {printf("\n链式栈是空的!");exit(1);}
    return(top->info);
}
```

```
int empty_link_stack(node *top)
{
    return (top? 0:1);
}
```

算法3.40取得链式栈的栈顶结点值

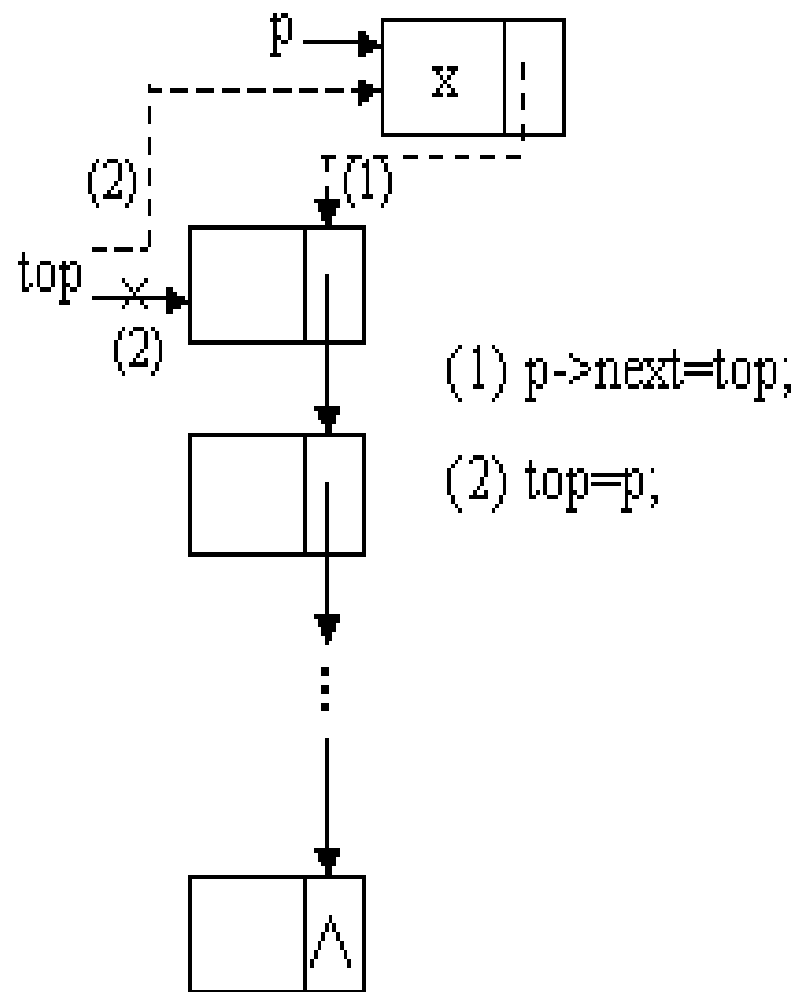
```

/*          输出链式栈中各个结点的值          */
/*  文件名lnksprin.c, 函数名print_link_stack()  */
/*****/

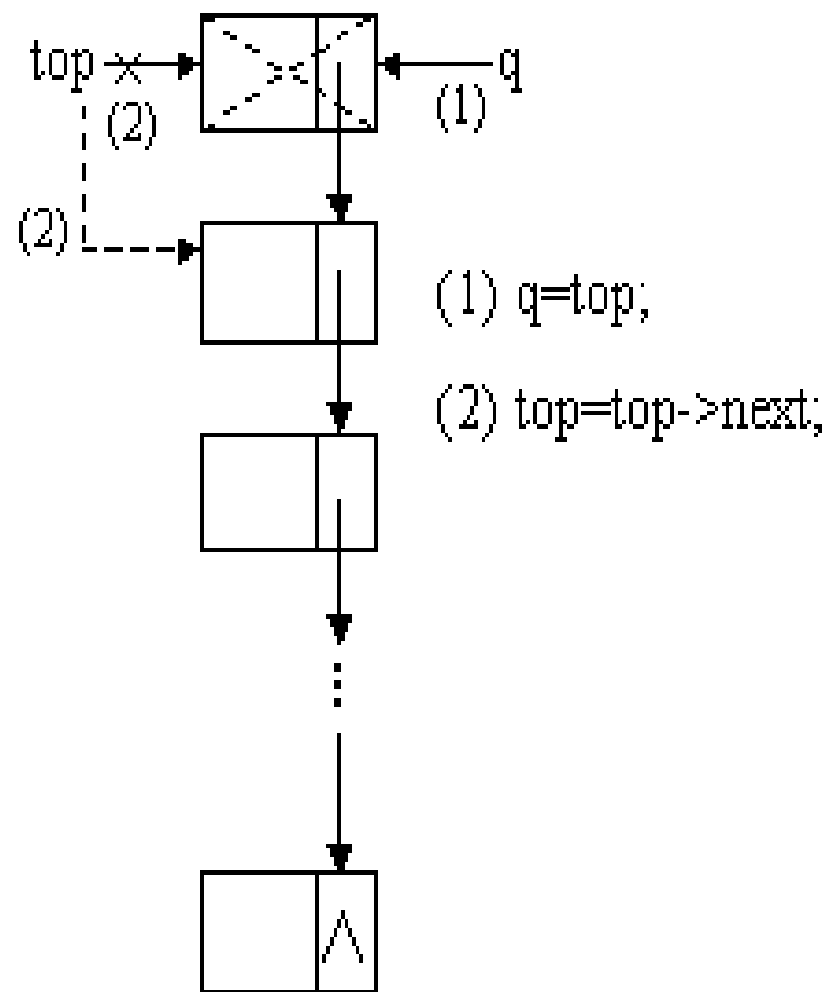
void print_link_stack(node *top)
{
    node *p;
    p=top;
    printf("\n");
    if(!p) printf("\n链式栈是空的!");
    while(p) { printf("%5d",p->info);p=p->next;}
}

```

算法3.41 输出链式栈中各个结点的值



链式栈插入操作



链式栈删除操作

```

/*          向链式栈中插入一个值为x的结点          */
/*  文件名lnkspush.c, 函数名push_link_stack()  */
node *push_link_stack(node *top,datatype x)
{
    node *p;
    p=(node*)malloc(sizeof(node)); /*分配空间*/
    p->info=x;                      /*设置新结点的值*/
    p->next=top;                    /*插入(1)*/
    top=p;                          /*插入(2)*/
    return top;
}

```

算法3.42向链式栈中插入一个值为x的结点

```

/*          删除链式栈的栈顶结点          */
/*  文件名lnkspop.c, 函数名pop_link_stack()  */
node *pop_link_stack(node *top)
{
    node *q;
    if(!top) {printf("\n链式栈是空的!");return NULL;}
    q=top; /*指向被删除的结点(1)*/
    top=top->next; /*删除栈顶结点(2)*/
    free(q);
    return top;
}

```

算法3.43 删除链式栈的栈顶结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

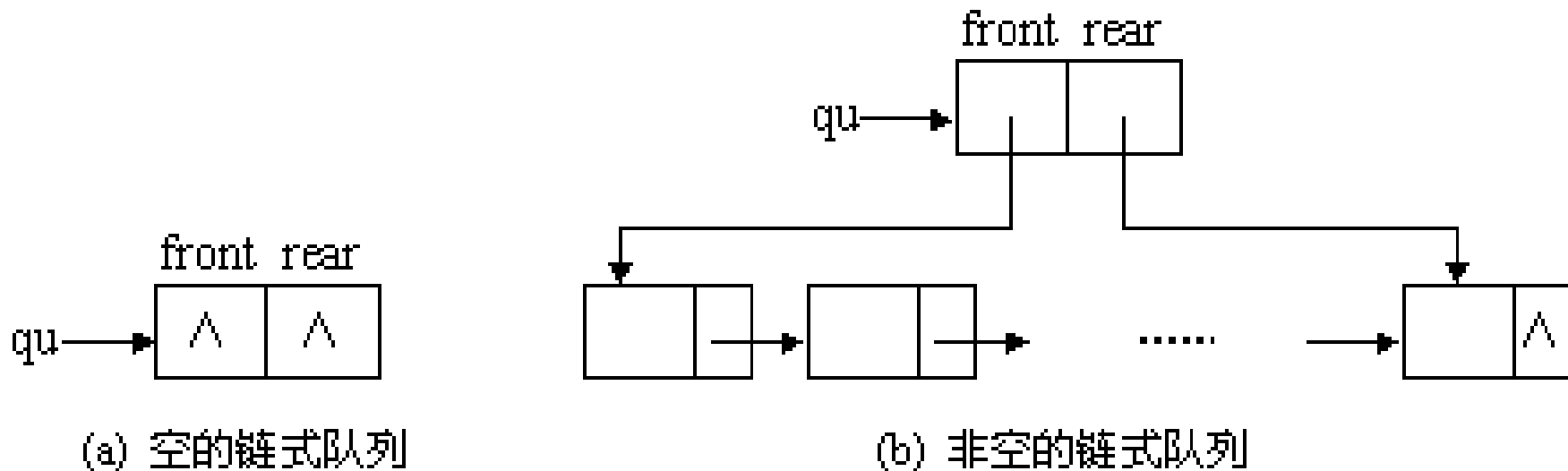
3.7.1 链式队列

3.7.2 链式队列的实现

3.7 链式队列

3.7.1 链式队列

队列的链式存储称为链式队列。链式队列就是一个特殊的单链表，对于这种特殊的单链表，它的插入和删除规定在单链表的不同端进行。链式队列的队首和队尾指针分别用**front**和**rear**表示，链式队列如下图所示。



链式队列类型的描述如下：

ADT link_queue{

数据集合**K**: $K=\{k_1, k_2, \dots, k_n\}, n \geq 0$, **K**中的元素是**datatype**类型

数据关系**R**: $R=\{r\}$

$r=\{ \langle k_i, k_{i+1} \rangle \mid i=1, 2, \dots, n-1 \}$

操作集合：

(1) **queue *init_link_queue()** 建立一个空的链式队列

(2) **int empty_link_queue(queue qu)** 判断链式队列是否为空

(3) **void print_link_queue(queue *qu)** 输出链式队列中各个结点的值

(4) **datatype get_first(queue qu)** 取得链式队列的队首结点值

(5) **queue *insert_link_queue(queue *qu, datatype x)** 向链式队列中插入一个值为**x**的结点

(6) **queue *delete_link_queue(queue *qu)** 删除链式队列中队首结点

} ADT link_queue;

3.7.2链式队列的实现:

链式队列的结点定义和单链表一样。队列必须有队首和队尾指针，因此增加定义一个结构类型，其中的两个域分别为队首和队尾指针。其定义如下：

```
typedef struct{  
    node *front,*rear; /*定义队首与队尾指针*/  
}queue;
```

```

/*          建立一个空的链式队列          */
/*  文件名lnkqinit.c, 函数名init_link_queue()  */
/*****/
queue *init_link_queue() /*建立一个空的链式队列*/
{
    queue *qu;
    qu=(queue*)malloc(sizeof(queue)); /*分配空间*/
    qu->front=NULL; /*队首指针设置为空*/
    qu->rear=NULL; /*队尾指针设置为空*/
    return qu;
}

```

算法3.44建立一个空的链式队列


```

/*****/

/*      取得链式队列的队首结点值      */

/*      文件名lnkqget.c, 函数名get_first()      */

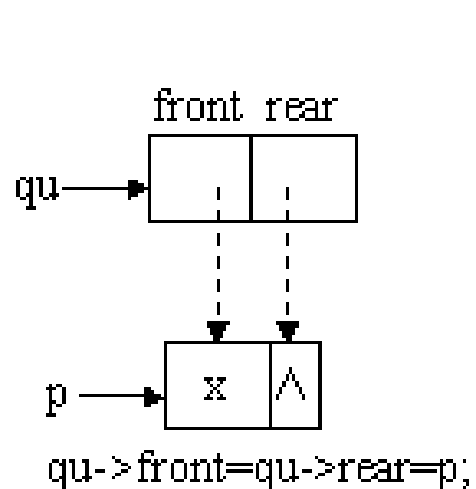
/*****/

datatype get_first(queue qu)
{
    if(!qu.front) {printf("\n链式队列是空的!");exit(1);}
    return(qu.front->info);
}

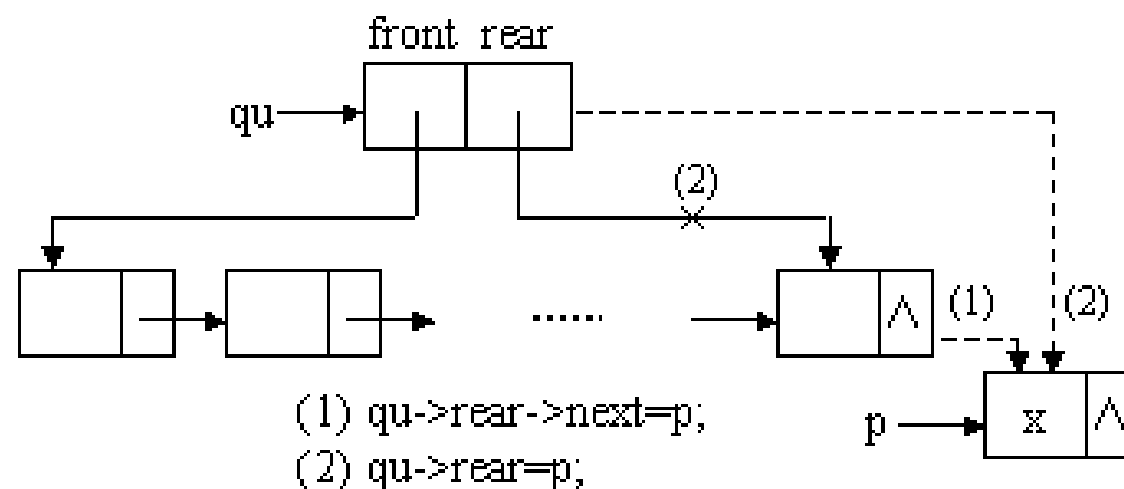
```

算法3.47取得链式队列的队首结点值

链式队列的插入过程图示见下图：



(a) 向空链式队列中插入值为 x 的新结点



(b) 在非空链式队列中插入值为 x 的新结点

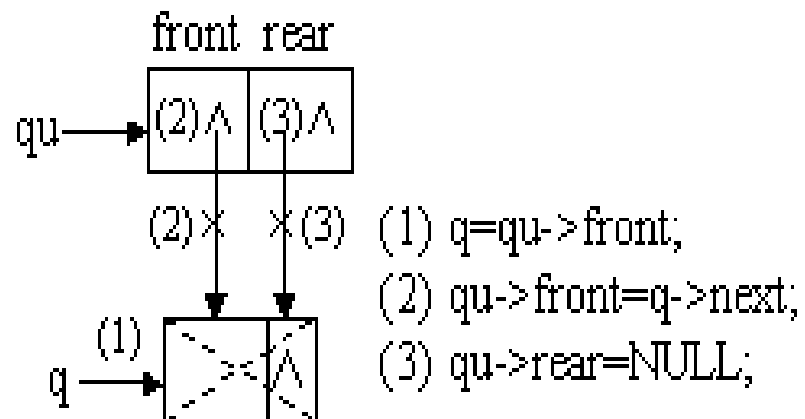
```

/*      向链式队列中插入一个值为x的结点      */
queue *insert_link_queue(queue *qu,datatype x)
{ node *p;
  p=(node*)malloc(sizeof(node)); /*分配空间*/
  p->info=x; /*设置新结点的值*/  p->next=NULL;
  if (qu->front==NULL) qu->front=qu->rear=p;
  else
  { qu->rear->next=p; /*队尾插入*/  qu->rear=p;
    }
  return qu;
}

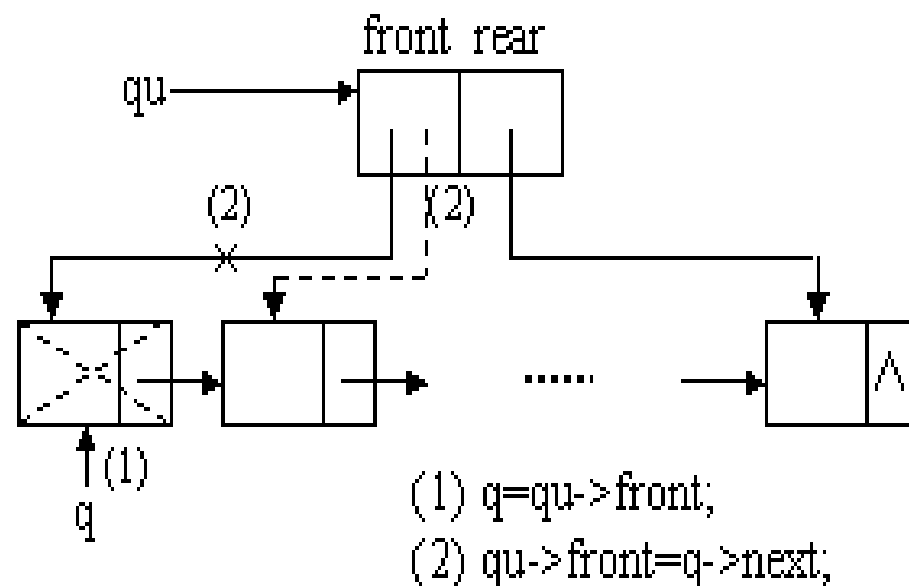
```

算法3.48向链式队列中插入一个值为x的结点

链式队列的删除过程图示见下图：



(a) 链式队列删除操作（队列中只有一个结点）



(b) 链式队列删除操作（队列中不只一个结点）

```

/*          删除链式队列中队首结点          */
queue *delete_link_queue(queue *qu)/*删除队首结点*/
{ node *q;
  if(!qu->front) {printf("队列为空，无法删除！ ");return qu;}
  q=qu->front; /*q指向队首结点(1)*/
  qu->front=q->next; /*队首指针指向下一个结点(2)*/
  free(q);      /*释放原队首结点空间*/
  if (qu->front==NULL) qu->rear=NULL; /*队列中的唯一结点
被删除后，队列变空(3)*/
  return qu;
}

```

算法3.49删除链式队列中队首结点

第3章 线性表的链式存储

3.1 链式存储

3.2 单链表

3.2.1 单链表的概念

3.2.2 单链表的实现

3.3 带头结点单链表

3.3.1 带头结点单链表

3.3.2 带头结点单链表的实现

3.4 循环单链表

3.4.1 循环单链表

3.4.2 循环单链表的实现

3.5 双链表

3.5.1 双链表

3.5.2 双链表的实现

3.6 链式栈

3.6.1 链式栈

3.6.2 链式栈的实现

3.7 链式队列

3.7.1 链式队列

3.7.2 链式队列的实现

作业： 1~10

第4章 字符串、数组 和特殊矩阵

➤ 字符串

➤ 字符串的模式匹配

➤ 数组

➤ 特殊矩阵

➤ 稀疏矩阵

4.1 字符串

4.1.1 字符串的基本概念

字符串是由零个或多个字符构成的有限序列，一般可表示成如下形式：

$$“c_1 c_2 c_3 \dots c_n” \quad (n \geq 0)$$

串中所含字符的个数 n 称为字符串的长度；当 $n=0$ 时，称字符串为**空串**。

串中任意个连续的字符构成的子序列称为该串的**子串**，包含子串的串称为**主串**。通常称字符在字符串序列中的序号为该字符在串中的位置。子串在主串中的位置以子串的第一个字符在主串中的位置来表示。例如： $T = \text{“STUDENT”}$ ， $S = \text{“UDEN”}$ ，则 S 是 T 的子串， S 在 T 中出现的位置为3。

两个字符串相等，当且仅当两个串的长度相等，并且各个对应位置的字符都相等。例如：

T1="REDROSE" T2="RED ROSE"

由于T1和T2的长度不相等，因此 $T1 \neq T2$ 。

若

T3="STUDENT" T4="STUDENS"

虽然T3和T4的长度相等，但两者有些对应的字符不同，因而 $T3 \neq T4$ 。

值得一提的是，若 $S=" "$ ，此时S由一个空格字符组成，其长度为1，它不等价于空串，因为空串的长度为0。

4.1.2 字符串类的定义

ADT string {

数据对象D: 由零个或多个字符型的数据元素构成的有限集合;

数据关系R: $\{ \langle a_i, a_{i+1} \rangle \mid \text{其中 } a_i, a_{i+1} \in D, i=1, 2, \dots, n-1 \}$

字符串的基本操作如下:

- (1) Strcreate (S)
- (2) Strassign(S, T)
- (3) **Strlength(S)**
- (4) **Strempty(S)**

- (5) **Strclear(S)**
- (6) **Strcompare(S₁, S₂)**
- (7) **Strconcat(S₁, S₂)**
- (8) **Substring(S, i, len)**
- (9) **Index(P,T)**
- (10) **Strinsert(S, i, T)**
- (11) **Strdelete(S,i,len)**
- (12) **Replace(S, T₁, T₂)**
- (13) **Strdestroy(S)**

} ADT string

4.1.3 字符串的存储及其实现

1、串的顺序存储及其部分运算的实现

串的顺序存储使用数组存放，具体类型定义如下：

```
#define MAXSIZE 100
typedef struct {
    char str[MAXSIZE];
    int length ;
} seqstring;
```

(1) 插入运算strinsert (S, i, T)

```
void strinsert(seqstring *S, int i , seqstring T)
{ int k;
  if (i<1 || i>S->length+1 || S->length + T.length>MAXSIZE)
    printf("cannot insert\n");
  else
  {
    for(k=S->length-1;k>=i-1;k--)
      S->str[T.length+k]=S->str[k];
    for (k=0;k<T.length;k++)
      S->str[i+k-1]=T.str[k];
    S->length= S->length + T.length;
    S->str[S->length]='\0';
  }
}
```

(2) 删除运算strdelete(S, i, len)

```
void strdelete(seqstring *S, int i, int len)
{ int k ;
  if (i<1 || i>S->length || i+len-1>S->length)
    printf(" cannot delete\n");
  else
  {
    for(k=i+len-1; k<S->length; k++)
      S->str[k-len]= S->str[k];
    S->length=S->length-len;
    S->str[S->length]='\0';
  }
}
```

(3) 连接运算 $\text{strconcat}(S_1, S_2)$

```
seqstring * strconcat(seqstring S1, seqstring S2)
{
    int i; seqstring *r;
    if (S1.length+S2.length>MAXSIZE)
        { printf('cannot concate'); return(NULL); }
    else
        {
            r=(seqstring*)malloc (sizeof(seqstring));
            for (i=0; i<S1.length;i++) r->str[i]= S1.str[i];
            for (i=0; i<S2.length;i++)
                r->str[ S1.length+i]= S2.str[i];
            r->length= S1.length+ S2.length;
            r->str[r->length]='\0';
        }
    return (r);
}
```

(4) 求子串运算substring(S, i, len)

```
seqstring *substring(seqstring S,int i, int len)
{ int k; seqstring *r;
  if (i<1 || i>S.length || i+len-1>S.length)
    { printf("error\n"); return(NULL);}
  else
    { r=(seqstring*) malloc (sizeof(seqstring));
      for(k=0;k<len;k++) r->str[k]= S.str[i+k-1];
      r->length=len;
      r->str[r->length]='\0';
    }
  return(r);
}
```


2 串的链接存储及其部分运算的实现

串的链接存储采用单链表的形式实现，其中每个结点的定义如下：

```
typedef struct node
{
    char data;
    struct node *next;
} linkstrnode;
typedef linkstrnode *linkstring;
```

例如，串S=“abcde”，其链接存储结构如下图所示：



(1) 创建字符串运算strcreate (S)

```
void strcreate (linkstring *S)
{ char ch; linkstrnode *p,*r;
  *S=NULL; r=NULL;
  while ((ch=getchar())!='\n')
  { p=(linkstrnode *)malloc(sizeof(linkstrnode));
    p->data=ch;
    if (*S==NULL) *S=p;
    else r->next=p;
    r=p; /*r移向当前链接串的最后一个字符的位置*/
  }
  if (r!=NULL) r->next=NULL; /*处理表尾结点指针域*/
}
```

(2) 插入运算strinsert (S, i, T)

```
void strinsert(linkstring *S,int i,linkstring T)
{ int k ; linkstring p,q;
  p=*S, k=1;
  while (p && k<i-1)
    { p=p->next ; k++; }
  if (!p) printf("error\n");
  else
  { q=T;
    while(q->next) q=q->next;
    q->next=p->next; p->next=T;
  }
}
```

(3) 删除运算strdelete(S, i, len)

```
void strdelete(linkstring*S, int i, int len)
{
    int k ;    linkstring p, q, r;
    p=*S, q=null; k=1;
    /*用p查找S的第i个元素， q始终跟踪p的前驱*/
    while (p && k<i)
        {q=p; p=p->next ; k++;}
    if (!p) printf("error1\n");
    else
        { k=1;
        /*p从第i个元素开始查找长度为len子串的最后元素*/
        while(k<len && p )
            { p=p->next ;k++;}
        if(!p)  printf("error2\n");
```

```
else
{
    if (!q) { r=*S; *S=p->next; }
    /*被删除的子串位于S的最前面*/
    else
    { /*被删除的子串位于 S 的中间或最后*/
        r=q->next; q->next= p->next;
    }
    p->next=null;
    while (r !=null)
        {p=r; r=r->next; free(p);}
}
}
```

(4) 连接运算strconcat(S_1, S_2)

```
void strconcat(linkstring *S1, linkstring S2)
{
    linkstring p;
    if (!(*S1) ) { *S1=S2; return;}
    else
        if (S2) /*S1和S2均不为空串的情形*/
        {
            p=*S1; /*用p查找S1的最后一个字符的位置*/
            while(p->next ) p= p->next;
            p->next=S2; /*将串S2连接到S1之后*/
        }
}
```

(5) 求子串运算substring(S, i, len)

```
linkstring substring(linkstring S, int i, int len)
{
    int k; linkstring p, q, r, t;
    p=S, k=1;
    /*用p查找S中的第i个字符*/
    while (p && k<i) {p= p->next;k++;}
    if (!p) {printf("error1\n"); return(null);}
    else
    {
        r=(linkstring) malloc (sizeof(linkstrnode));
        r->data=p->data; r->next=null;
    }
}
```

```
k=1; q=r; /*用q始终指向子串的最后一个字符的位置*/
while (p->next && k<len) /*取长度为len的子串*/
{
    p=p->next ;k++;
    t=(linkstring) malloc (sizeof (linkstrnode));
    t->data=p->data;
    q->next=t; q=t;
}
if (k<len) {printf("error2\n") ; return(null);}
else
    {q->next=null; return(r);} /*处理子串的尾部*/
}
```


4.2 字符串的模式匹配

寻找字符串p在字符串t中首次出现的起始位置称为字符串的**模式匹配**，其中称p为**模式**(pattern)，t为**正文**(text)，t的长度远远大于p的长度。

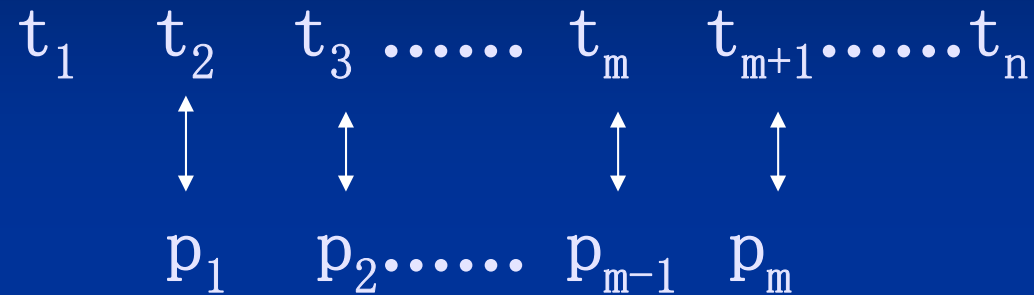
4.2.1 朴素的模式匹配算法

朴素模式匹配算法的基本思想是：用p中的每个字符去与t中的字符一一比较：

正文t:	t_1	t_2	t_m	t_n
	↑	↑		↑		
	↓	↓		↓		
模式p:	p_1	p_2	p_m		

如果 $t_1=p_1, t_2=p_2, \dots, t_m=p_m$ ，则模式匹配成功；否则

将p向右移动一个字符的位置，重新用p中的字符从头开始与t中相对应的字符依次比较，即：



如此反复，直到匹配成功或者p已经移到使t中剩下的字符个数小于p的长度的位置，此时意味着模式匹配失败，表示t中没有子串与模式p相等，我们约定返回-1代表匹配失败。

朴素模式匹配算法的具体实现如下：

```
int index(seqstring p, seqstring t)
{  int i, j, succ;
   i=0; succ=0;  /* succ为匹配成功的标志*/
   while((i<=t.length-p.length+1) && (!succ))
       { j=0 ; succ=1;    /*用j扫描模式p*/
         while ((j<=p.length-1) && succ)
             if (p.str[j]==t.str[i+j] )  j++;
             else succ=0;
         ++i;
       }
   if (succ) return (i-1);
   else  return (-1);
}
```

4.2.2 快速模式匹配算法（KMP算法）

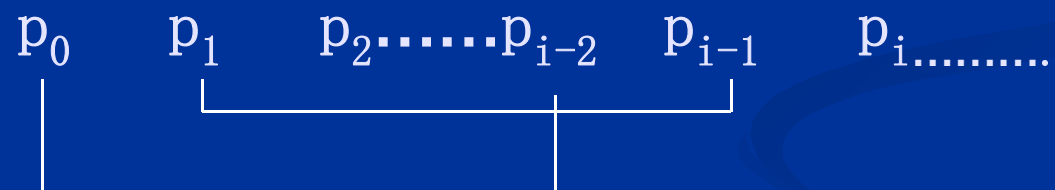
首先我们来分析下图所示的情况：

$$\begin{array}{ccccccccccc}
 t_0 & t_1 & t_2 & \cdots & t_k & t_{k+1} & t_{k+2} & \cdots & t_{r-2} & t_{r-1} & t_r \cdots \\
 & & & & \parallel & \parallel & \parallel & & \parallel & \parallel & \parallel \\
 & & & & p_0 & p_1 & p_2 & \cdots & p_{i-2} & p_{i-1} & p_i \cdots
 \end{array}
 \quad (4-1)$$

$$\begin{array}{ccccccccccc}
 t_0 & t_1 & t_2 & \cdots & t_k & t_{k+1} & t_{k+2} & \cdots & t_{r-2} & t_{r-1} & t_r \cdots \\
 & & & & & \parallel & \parallel & & & \parallel & \parallel \\
 & & & & & p_0 & p_1 & \cdots & & p_{i-2} & p_{i-1} \\
 p_i \cdots & & & & & & & & & &
 \end{array}
 \quad (4-2)$$

$$\begin{array}{ccccccccccc}
 t_0 & t_1 & t_2 & \cdots & t_k & t_{k+1} & t_{k+2} & \cdots & t_{r-2} & t_{r-1} & t_r \cdots \\
 & & & & & & \parallel & & & \parallel & \parallel \\
 & & & & & & p_0 & \cdots & & p_{i-3} & p_{i-2} & p_{i-1} & p_i \cdots
 \end{array}
 \quad (4-3)$$

(4-1)式表明此次匹配从 p_0 与 t_k 开始比较, 当比较到 p_i 与 t_r 时出现不等情况, 于是将模式右移一位, 变成(4-2)所示的状态, 若此次比较成功, 则必有 $p_0 = t_{k+1}$, $p_1 = t_{k+2}$, $\dots\dots p_{i-2} = t_{r-1}$, 且 $p_{i-1} \neq p_i$; 而根据(4-1)的比较结果有: $p_1 = t_{k+1}$, $p_2 = t_{k+2}$, $\dots\dots p_{i-1} = t_{r-1}$, 因此有: $p_0 = p_1$, $p_1 = p_2$, $\dots\dots p_{i-2} = p_{i-1}$ 。这个性质说明在模式 p 中 p_i 之前存在一个从 p_0 开始长度为 $i-1$ 的连续序列 $p_0 \ p_1 \ \dots\dots p_{i-2}$ 和以 p_{i-1} 为结尾, 长度同样为 $i-1$ 的连续序列 $p_1 \ p_2 \ \dots\dots p_{i-1}$ 其值对应相等, 即:



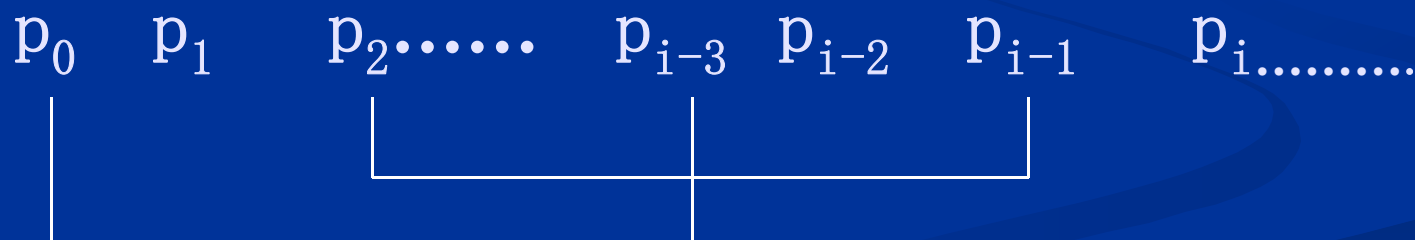
简记为:

$$[p_0 \text{---} p_{i-2}] = [p_1 \text{---} p_{i-1}]$$

称模式 p 中 p_i 之前存在长度为 $i-1$ 的真前缀和真后缀的匹配。

反之，若在(4-1)所示的状态下，模式p中 p_i 之前存在长度为 $i-1$ 的真前缀和真后缀的匹配，即 $[p_0—p_{i-2}]=[p_1—p_{i-1}]$ 且 $p_{i-1} \neq p_i$ ；当 p_i 与 t_r 出现不等时，根据前面已比较的结果 $p_1 = t_{k+1}$, $p_2 = t_{k+2}$, $p_{i-1} = t_{r-1}$ ，于是可得 $p_0 = t_{k+1}$, $p_1 = t_{k+2}$, $p_{i-2} = t_{r-1}$ ，因此接下来只需从 p_{i-1} 与 t_r 开始继续后继对应字符的比较即可。

再假设在(4-1)所示的状态下，模式右移一位成为状态(4-2)后，匹配仍然不成功，说明 $[p_0—p_{i-2}] \neq [p_1—p_{i-1}]$ 或 $p_{i-1} = p_i$ ，于是模式再右移一位，成为状态(4-3)，若此次匹配成功，仿照上述分析，必有：



即：

$$[p_0 \text{---} p_{i-3}] = [p_2 \text{---} p_{i-1}]$$

说明模式p中 p_i 之前存在长度为 $i-2$ 的真前缀和真后缀的匹配。由(4-3)表明，在(4-1)所示的状态下，若模式p中 p_i 之前最长真前缀和真后缀匹配的长度为 $i-2$ ，当 p_i 与 t_r 出现不等时，接下来只需从 p_{i-2} 与 t_r 开始继续后继对应字符的比较。

考虑一般情况。在进行模式匹配时，若模式p中 p_i 之前最长真前缀和真后缀匹配的长度为 j ，当 $p_i \neq t_r$ 时，则下一步只需从 p_j 与 t_r 开始继续后继对应字符的比较，而不应该将模式一位一位地右移，也不应该反复从模式的开头进行比较。这样既不会失去任何匹配成功的机会，又极大地加快了匹配的速度。

根据上述分析，在模式匹配过程中，每当出现 $p_i \neq t_r$ 时，下一次与 t_r 进行比较的 p_j 和模式 p 中 p_i 之前最长真前缀和真后缀匹配的长度密切相关；而模式 p 中 p_i 之前最长真前缀和真后缀匹配的长度只取决于模式 p 的组成，与正文无关。

于是我们可以针对模式 p 定义一个数组 $next[m]$ ，其中 $next[i]$ 表示当 $p_i \neq t_r$ 时，下一次将从 $p_{next[i]}$ 与 t_r 开始继续后继对应字符的比较。显然， $next[i]$ 的值与模式 p 中 p_i 之前最长真前缀和真后缀匹配的长度密切相关。下面考虑如何根据模式 p 的组成求数组 $next$ 的值。我们规定：

$$next[0] = -1$$

这表明当 $p_0 \neq t_r$ 时，将从 p_{-1} 与 t_r 开始继续后继对应字符的比较；然而 p_{-1} 是不存在的，我们可以将这种情况理解成下一步将从 p_0 与 t_{r+1} 开始继续后继对应字符的比较。

以下假设 $\text{next}[0]$, $\text{next}[1]$,, $\text{next}[i]$ 的值均已求出, 现要求 $\text{next}[i+1]$ 的值。由于在求解 $\text{next}[i]$ 时已得到 p_i 之前最长真前缀和真后缀匹配的长度, 设其值为 j , 即:

$p_0 \quad p_1 \quad p_2 \cdots p_{i-j} \cdots p_{j-1} \quad p_j \cdots p_{i-2} \quad p_{i-1} \quad p_i \quad p_{i+1} \cdots$

如果此时进一步有 $p_j = p_i$, 则 p_{i+1} 之前最长真前缀和真后缀匹配的长度就为 $j+1$, 且 $\text{next}[i+1] = j+1$; 反之, 若 $p_j \neq p_i$, 注意到, 求 p_{i+1} 之前最长真前缀和真后缀匹配问题本质上仍然是一个模式匹配问题, 只是在这里模式和正文都是同一个串 p 而已。因此当 $p_j \neq p_i$ 时, 应该检查 $p_{\text{next}[j]}$ 与 p_i 是否相等; 若相等, 则 $\text{next}[i+1] = \text{next}[j] + 1$; 如仍然不相等, 则再取 $p_{\text{next}[\text{next}[j]]}$ 与 p_i 进行比较,直至要将 p_{-1} 与 p_i 进行比较为止, 此时 $\text{next}[i+1] = 0$ 。

以下给出了根据模式p的组成求数组next值的算法:

```
void getnext(seqstring p, int next[])
{
    int i, j;
    next[0] = -1;
    i = 0;    j = -1;
    while (i < p.length)
    {
        if (j == -1 || p.str[i] == p.str[j])
            {++i; ++j; next[i] = j;}
        else
            j = next[j];
    }
    for (i = 0; i < p.length; i++)
        printf("%d", next[i]);
}
```

KMP算法基本思想如下：

假设以 i 和 j 分别指示正文 t 和模式 p 中正待比较的字符，令 i 、 j 的初值为0；若在匹配过程中 $t_i = p_j$ ，则 i 与 j 分别加1；否则 i 不变，而 j 退到 $\text{next}[j]$ 的位置继续比较（即 $j = \text{next}[j]$ ）；若相等，则指针各自增加1；否则 j 再退到下一个 $\text{next}[j]$ 值的位置，依此类推，直至下列两种可能：

- (1) 一种是 j 退到某个 $\text{next}(\text{next}[\dots[\text{next}[j]]\dots])$ 时， t_i 与 p_j 字符比较相等，则 i 、 j 指针各自增加1后继续进行比较；
- (2) 一种是 j 退到-1（即模式的第一个字符“失配”），此时需将正文指针 i 向右滑动一个位置，即从正文的下一个字符 t_{i+1} 起和模式 p 重新从头开始比较。

KMP算法的具体实现如下:

```
int kmp(seqstring t, seqstring p, int next[])
{ int i, j;
  i=0; j=0;
  while (i<t.length && j<p.length)
  {
    if (j==-1 || t.str[i]==p.str[j])
      {i++; j++;}
    else j=next[j];
  }
  if (j==p.length) return (i-p.length);
  else return(-1);
}
```

4.3 数 组

4.3.1 数组和数组元素

数组是线性表的一种存储方式。其实，数组本身也可以看成是线性表的推广，数组的每个元素由一个值和一组下标确定，在数组中，对于每组有定义的下标都存在一个与之相对应的值；而线性表是有限结点的有序集合，若将其每个结点的序号看成下标，线性表就是一维数组（向量）；当数组为多维数组时，其对应线性表中的每个元素又是一个数据结构而已。

例如，对于一个 $m \times n$ 的二维数组 $A[m][n]$ ：

$$A = \left\{ \begin{array}{cccc} a_{00} & a_{01} & a_{02} \cdots a_{0(n-1)} \\ a_{10} & a_{11} & a_{12} \cdots a_{1(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} \cdots a_{(m-1)(n-1)} \end{array} \right\}$$

当把二维数组看成是线性表时，它的每一个结点又是一个向量（一维数组）。例如，上述二维数组 A 可以看成是如下的线性表：

$$(A_0, A_1, A_2, \dots, A_{m-1})$$

即 A 中每一行成为线性表的一个元素，其中每个元素 A_i ($0 \leq i \leq m-1$) 都是一个向量；

$$(a_{i0}, a_{i1}, a_{i2} \cdots a_{i(n-1)})$$

当然，也可以将上述二维数组A看成如下的线性表：

$$(A_0', A_1', A_2', \dots, A_{n-1}')$$

即A中每一列成为线性表的一个元素，其中每一个元素 A_i' ($0 \leq i \leq n-1$) 都是一个向量：

$$(a_{0i}, a_{1i}, a_{2i}, \dots, a_{(m-1)i})$$

二维数组A中的每一个元素 a_{ij} 都同时属于两个向量，即：第 $i+1$ 行的行向量和第 $j+1$ 列的列向量，因此每个元素 a_{ij} 最多有两个前驱结点 $a_{(i-1)j}$ 和 $a_{i(j-1)}$ ，也最多有两个后继结点 $a_{(i+1)j}$ 和 $a_{i(j+1)}$ （只要这些结点存在）；特别地， a_{00} 没有前驱结点， $a_{(m-1)(n-1)}$ 没有后继结点，边界上的结点均只有一个后继结点或一个前驱结点。

对于 m ($m > 2$) 维数组，可以依据上述规律类推。

4.3.2 数组类的定义

ADT array {

数据对象D: 具有相同类型的数据元素构成的有序集合;

数据关系R: 对于n维数组, 其每一个元素均位于n个向量中,
每个元素最多具有n个前驱结点和n个后继结点;

数组的基本操作如下:

(1) **Initarray (A, n, index1, index2,index n)**

(2) **Destroyarray(A)**

(3) **Value(A, index1, index2,index n, x)**

(4) **Assign (A, e, index1, index2,index n)**

} ADT array

4.3.3 数组的顺序存储及实现

由于数组是由有限的元素构成的有序集合，数组的大小和元素之间的关系一经确定，就不再发生变化，因此数组均采用顺序存储结构实现，它要求一片连续的存储空间存储。

多维数组数据元素的顺序存储有两种方式：

- 按行优先存储
- 按列优先存储

例如：对于二维数组A[m][n]：

$$A = \left\{ \begin{array}{cccc} a_{00} & a_{01} & \dots\dots\dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots\dots\dots & a_{1(n-1)} \\ \vdots & \vdots & & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \dots\dots\dots & a_{(m-1)(n-1)} \end{array} \right\}$$

若将A按行优先存储，其存储顺序为： $a_{00}, a_{01}, \dots\dots a_{0(n-1)}, a_{10}, a_{11}, \dots\dots a_{1(n-1)}, \dots\dots a_{(m-1)0}, a_{(m-1)1}, \dots\dots a_{(m-1)(n-1)}$ ；而按列优先存储，其顺序为： $a_{00}, a_{10}, \dots\dots a_{(m-1)0}, a_{01}, a_{11}, \dots\dots a_{(m-1)1}, \dots\dots a_{0(n-1)}, \dots\dots a_{1(n-1)}, \dots\dots a_{(m-1)(n-1)}$ 。

对于数组，一旦确定了它的维数和各维的长度，便可以为它分配存储空间；当规定了数组元素的存储次序后，便可根据给定的一组下标值求得相应数组元素的存储位置。

现假设数组中每个元素占用L个存储单元，若考虑按行优先存储方式，则上述A数组中任何一个元素 a_{ij} 的存储位置可以按以下公式确定：

$$\text{address}(a_{ij}) = \text{address}(a_{00}) + (i \times n + j) \times L$$

若考虑按列优先的存储方式，数组中任何一个元素 a_{ij} 存储位置的地址计算公式为：

$$\text{address}(a_{ij}) = \text{address}(a_{00}) + (j \times m + i) \times L$$

多维数组的存储也和二维数组一样，存在两种存储方式：按行优先和按列优先。但由于多维数组中数据元素间的关系较二维数组复杂，因此数据元素的地址计算公式也相对复杂些，但两者所采用的原理是相同的。

考虑n维数组的情形：

datatype A[b₁][b₂].....[b_n];

其中b₁、b₂、.....b_n为数组每一维的长度。仍假设每个元素占用L个单元，则n维数组A中任何一个元素A[j₁][j₂].....[j_n]在按行优先存储方式下的地址计算公式为：

$$\text{address}(A[j_1][j_2].....[j_n]) = \text{address}(A[0][0]...[0]) + (b_2 \times b_3 \times b_n \times j_1 + b_3 \times b_4 \times b_n \times j_2 + b_n \times j_{n-1} + j_n) \times L$$

上式可以简写为：

$$\text{address}((A[j_1][j_2].....[j_n])) = \text{address}(A[0][0]...[0]) + c_1 * j_1 + c_2 * j_2 + c_n * j_n$$

其中c_n=L，c_{i-1}=b_i×c_i，1<i≤n。

以下以三维数组为例，给出三维数组的顺序存储表示及其部分运算的实现。

```
typedef int datatype;
    /*假设数组元素的值为整型*/
typedef struct {
    datatype *base; /* 数组存储区的首地址指针*/
    int      index[3]; /* 存放三维数组各维的长度*/
    int      c[3]      /* 存放三维数组各维的 $c_i$ 值*/
} array;
```

1、 数组初始化运算initarray (A, b1, b2, b3)

```
int initarray (array *A, int b1 , int b2, int b3)
{
    int elements;
    if (b1<=0||b2<=0||b3<=0)  return( 0 );
    A->index[0]=b1; A->index[1]=b2; A->index[2]=b3;
    elements = b1 × b2 × b3;
    A->base=(datatype*)malloc(elements×sizeof(datatype));
    if (! (A->base)) return(0);
    A->c[0]= b2 × b3;  A->c[1]= b3;  A->c[2]= 1;
    return(1);
}
```

2、访问数组元素值的运算value(A, i1, i2, i3, x)

```
int value(array A, int i1 , int i2, int i3; datatype *x)
{
    int off;
    if (i1<0 || i1>=A.index[0] || i2< 0 || i2>=A.index[1] ||
        i3<0 || i3>=A.index[2])
        return(0);  /*处理下标非法的情况*/
    off= i1 × A.c[0]+ i2 × A.c[1]+ i3 × A.c[2];
    *x=*(A.base + off);  /*赋值*/
    return(1);
}
```

3、数组元素的赋值运算assign(A, e, i1, i2, i3)

```
int assign( array *A, datatype e, int i1, int i2, int i3)
{
    int off;
    if (i1<0 || i1>=A->index[0] || i2< 0 || i2>=A->index[1]
        || i3<0 || i3>=A->index[2])
        return (0 ); /*处理下标非法的情况*/
    off= i1 × A->c[0]+ i2 × A->c[1]+ i3 × A->c[2];
    *(A->base + off)=e; /*赋值*/
    return(1);
}
```


4.4 特殊矩阵

本节主要研究对称矩阵、三角矩阵和带状矩阵的压缩存储。所谓压缩存储即为：多个相同值的结点只分配一个存储空间，值为零的结点不分配空间。

4.4.1 对称矩阵的压缩存储

如果矩阵的行数和列数相等，则称该矩阵为方阵。若 $n \times n$ 阶的方阵A满足：

$$a_{ij} = a_{ji} \quad (0 \leq i \leq n-1, 0 \leq j \leq n-1)$$

则称矩阵A为对称矩阵。在对称矩阵中，几乎有一半元素的值是对应相等的。如果将A中所有元素进行存储，那将会造成空间的浪费，且n值越大，浪费将越严重。

对于对称矩阵压缩存储时只需存储对角线以上或对角线以下的部分，未存储的部分可以利用元素之间的对称性来访问。

现考虑只存储对称矩阵A对角线以下的部分（即下标满足 $i \geq j$ 的数组元素 a_{ij} ）：

$$A = \begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ \vdots & \vdots & \vdots & \\ a_{(n-1)0} & \dots & \dots & a_{(n-1)(n-1)} \end{pmatrix}$$

若采用按行优先的存储方式，A进行压缩存储后任何一个元素 a_{ij} 的地址计算公式为：

$$\text{address}(a_{ij}) = \begin{cases} \text{address}(a_{00}) + (i * (i+1) / 2 + j) \times L & \text{当 } i \geq j \\ \text{address}(a_{00}) + (j * (j+1) / 2 + i) \times L & \text{当 } i < j \end{cases}$$

4.4.2 三角矩阵的压缩存储

1、下三角矩阵

$$A = \begin{Bmatrix} a_{00} & 0 & 0 & \dots\dots\dots 0 \\ a_{10} & a_{11} & 0 & \dots\dots\dots 0 \\ a_{20} & a_{21} & a_{22} & \dots\dots\dots 0 \\ \vdots & \vdots & \vdots & & \vdots \\ a_{(n-1)0} & \dots\dots\dots a_{(n-1)(n-1)} \end{Bmatrix}$$

仍考虑采用按行优先方式，A中下三角部分的任何一个元素 a_{ij} ($i \geq j$) 压缩存储后的地址计算公式为：

$$\text{address}(a_{ij}) = \text{address}(a_{00}) + (i * (i+1) / 2 + j) \times L \quad \text{当 } i \geq j$$

与对称矩阵不同的是，当 $i < j$ 时， a_{ij} 的值为0，其没有对应的存储单元。

2、 上三角矩阵

$$A = \begin{Bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0(n-1)} \\ 0 & a_{11} & a_{12} & \dots & a_{1(n-1)} \\ 0 & 0 & a_{22} & \dots & a_{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & a_{(n-1)(n-1)} \end{Bmatrix}$$

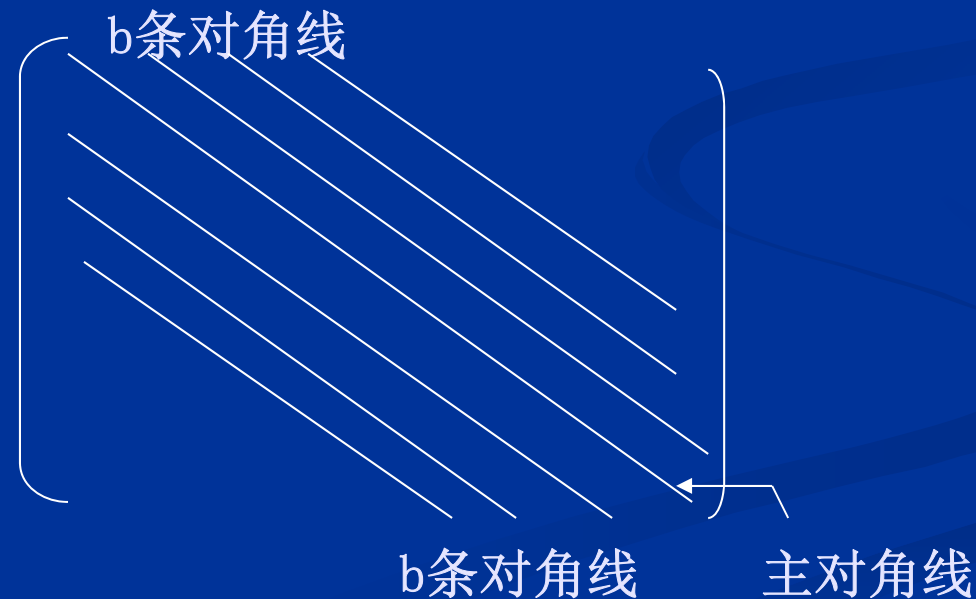
对于上三角矩阵，只需考虑存储对角线以上的部分，对角线以下为0的部分无需存储。仍采用按行优先存储方式，矩阵A中被存储元素 a_{ij} ($i \leq j$)在压缩存储方式下的地址计算公式为：

$$\begin{aligned} \text{address}(a_{ij}) &= \text{address}(a_{00}) + [(n + (n-1) + (n-2) + \dots + (n - (i-1))) + j - i] \times L \\ &= \text{address}(a_{00}) + (i * n - (i-1) * i / 2 + j - i) * L \end{aligned}$$

而当 $i > j$ 时， a_{ij} 的值为0，其没有对应的存储空间。

4.4.3 带状矩阵的压缩存储

对于 $n \times n$ 阶方阵，若它的全部非零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及上面各 b 条对角线上的元素以及主对角线上的元素，那么称该方阵为半带宽为 b 的带状矩阵。带状矩阵的特点是：对于矩阵元素 a_{ij} ，若 $i-j > b$ 或 $j-i > b$ ，即 $|i-j| > b$ ，则 $a_{ij} = 0$ 。



带状矩阵进行压缩存储时，只存储带状部分内部的元素，对于带状区域以外的元素，即 $|i-j|>b$ 的 a_{ij} ，均不分配存储空间。为了方便起见，我们规定按如下方法进行存储：除第一行和最后一行外，每行都分配 $2b+1$ 个元素的空间，将带状区域中的元素存储于

$((2b+1) \times n - 2b) \times L$ 个存储单元之中，其中 L 为每个元素占用空间的大小。仍考虑采用按行优先的存储方式，于是可以得到带状区域中任何一个元素 a_{ij} 的地址计算公式为：

$$\begin{aligned} \text{address}(a_{ij}) &= \text{address}(a_{00}) + ((i \times (2b+1) - b) + (j - i + b)) \times L \\ &= \text{address}(a_{00}) + (i \times (2b+1) + j - i) \times L \\ &\quad (\text{当 } |i-j| \leq b \text{ 时}) \end{aligned}$$

4.5 稀疏矩阵

如果一个矩阵中很多元素的值为零，即零元素的个数远远大于非零元素的个数时，称该矩阵为**稀疏矩阵**。

4.5.1 稀疏矩阵类的定义

ADT spmatrix {

数据对象D: 具有相同类型的数据元素构成的有限集合;

数据关系R: D中的每个元素均位于2个向量中，每个元素最多具有2个前驱结点和2个后继结点，且D中零元素的个数远远大于非零元素的个数;

稀疏矩阵的基本运算如下:

(1) Createspmatrix (A)

```
(2) compressmatrix(A, B)
(3) Destroyspmatrix(A)
(4) Printspmatrix(A)
(5) Copyspmatrix(A, B)
(6) Addspmatrix(A, B, C)
(7) Subspmatrix(A, B, C)
(8) Multspmatrix(A,B,C)
(9) Transpmatrix(B,C)
(10) locatespmatrix(A, x, rowx, colx)
} ADT spmatrix
```


4.5.2 稀疏矩阵的顺序存储及其实现

稀疏矩阵的顺序存储方法包括：三元组表示法、带辅助行向量的二元组表示法和伪地址表示法，其中以三元组表示法最常用，故在此主要介绍稀疏矩阵的三元组表示。

在三元组表示法中，稀疏矩阵的每个非零元素可以采用如下形式表示：

(i, j, value)

其中i表示非零元素所在的行号，j表示非零元素所在的列号，value表示非零元素的值。采用三元组表示法表示一个稀疏矩阵时，首先将它的每一个非零元素表示成上述的三元组形式，然后按行号递增的次序、同一行的非零元素按列号递增的次序将所有非零元素的三元组表示存放一片连续的存储单元中即可。

以下是稀疏矩阵 $A_{7 \times 6}$ 及其对应的三元组表示。

0	0	-5	0	1	0
0	0	0	2	0	0
3	0	0	0	0	0
0	0	0	0	0	0
12	0	0	0	0	0
0	0	0	0	0	4
0	0	21	0	0	0

稀疏矩阵A

B	0	1	2
0	7	6	7
1	0	2	-5
2	0	4	1
3	1	3	2
4	2	0	3
5	4	0	12
6	5	5	4
7	6	2	21

A的三元组表示

稀疏矩阵A及其对应的三元组表示矩阵B的数据
类型定义如下：

```
typedef struct {  
    int  data[100][100];  /*存放稀疏矩阵的二  
                           维数组*/  
    int  m,n;             /*分别存放稀疏矩阵  
                           的行数和列数*/  
} matrix;  
  
typedef int  spmatrix[100][3];  
  
/*稀疏矩阵对应的三元组表示的类型 */
```

1、产生稀疏矩阵三元组表示的算法

```
void compressmatrix(matrix A , spmatrix B)
{   int i, j, k=1;
    for ( i=0; i<A.m; i++)
        for (j=0; j<A.n; j++)
            if (A.data[i][j] !=0)
                {   B[k][0]=i;
                    B[k][1]=j;
                    B[k][2]=A.data[i][j];
                    k++;
                }
    B[0][0]=A.m; B[0][1]=A.n;
    B[0][2]=k-1;
}
```

2、求稀疏矩阵的转置算法transpmatrix(B, C)

按照矩阵转置的定义，要实现矩阵的转置，只要将矩阵中以主对角线为对称轴的元素 a_{ij} 和 a_{ji} 的值互换。但三元组的排列要求采用按行优先方式，如果只是简单地将非零元素的行号和列号交换，则新产生的三元组表示将不再满足按行优先的原则。

解决的办法是：首先确定B中每一列非零元素的个数，也即将来C中每一行非零元素的个数，从而可计算出C中每一行非零元素三元组的起始位置，这样便可实现将B中的非零元素交换行号和列号后逐一放到它们在C中的最终位置上了。为了求B中每一列非零元素的个数和C中每一行非零元素三元组的起始位置，可以设置两个数组x和y来实现相应的功能。

```
void transpmatrix (spmatrix B, spmatrix C)
{
    int i, j, t, m, n;
    int x[100]; /*用来存放B中每一列非零元素的个数*/
    int y[100]; /*存放C中每一行非零元素三元组的起始
                  位置*/
    m=B[0][0]; n=B[0][1]; t=B[0][2];
    C[0][0]=n; C[0][1]=m; C[0][2]=t;
    if (t>0)
    {
        for (i=0; i<n; i++) x[i]=0;
        for (i=1; i<=t; i++) x[B[i][1]]=x[B[i][1]]+1;
        /*统计B中每一列非零元素的个数*/
        /*求矩阵C中每一行非零元素三元组的起始位置*/
        y[0]=1;
        for (i=1; i<n; i++) y[i]=y[i-1]+x[i-1];
    }
}
```

```
for (i=1; i<=t; i++)  
{ /*将B中非零元素交换行号、列号后写入C中  
   其最终的位置上*/  
    j=y[B[i][1]];  
    C[j][0]= B[i][1];  
    C[j][1]= B[i][0];  
    C[j][2]= B[i][2];  
    y[B[i][1]]=j+1;  
}  
}  
}
```

4.5.3 稀疏矩阵的链式存储及实现

十字链表的表示法是稀疏矩阵的链式存储方法之一，其基本思想为：将稀疏矩阵同一行的所有非零元素串成一个带表头的环形链表，同一列的所有非零元素也串成一个带表头的环形链表，且第 i 行非零元素链表的表头和第 i 列非零元素链表的表头共用一个表头结点，同时所有表头结点也构成一个带表头的环形链表。因此，在十字链表的表示中有两类结点，非零元素结点和表头结点。非零元素结点的结构：

row	col	val
right		down

为了程序实现方便，我们将表头结点的结构定义成与非零元素结点的结构相同，只是将其行域和列域的值置为0；另外，由于所有的表头结点也要串成一个带表头的环形链表，且表头结点本身没有数据值，因此可将非零元素结点中的val域改为指向本表头结点的下一个表头结点的指针域next，即val域和next域共用一片存储空间，于是得到表头结点的结构如下：

row	col	next
right		down

具体实例见书第99页。

稀疏矩阵十字链表表示中结点的类型定义如下：

```
typedef struct matrixnode
/*十字链表中结点的结构*/
{
    int    row,    col;
    struct matrixnode  *right, * down;
    union{ int    val;
           struct matrixnode *next;
        } tag;
} matrixnode;
typedef matrixnode *spmatrix;
typedef spmatrix headspmatrix[100];
```

1、稀疏矩阵十字链表的创建算法

```
void Createspmatrix (headspmatrix h)
{ int m,n,t,s,i,r,c,v;  spmatrix p,q;
  printf(“矩阵的行数、列数和非零元素的个数: ” );
  scanf(“%d%d%d”,&m,&n,&t);
  p=(spmatrix) malloc (sizeof(matrixnode));
  h[0]=p;  /* h[0]为表头环形链表的表头结点*/
  p->row=m; p->col=n;
  s=m>n?m:n;
  for (i=1;i<=s;++i)
  { p=(spmatrix) malloc (sizeof(matrixnode));
    h[i]=p;  h[i-1]->tag.next=p;
    p->row=p->col=0;  p->down=p->right=p;
  }

  h[s]->tag.next=h[0];
```

```
for (i=1;i<=t;++i)
{ printf("输入非零元素的行号、列号和值: ");
  scanf("%d%d%d",&r,&c,&v);
  p=(spmatrix) malloc (sizeof(matrixnode));
  p->row=r; p->col=c; p->tag.val=v;
  q=h[r];
  while (q->right!=h[r] && q->right->col<c)
    q=q->right;
  p->right=q->right;
  q->right=p;
  q=h[c]; /*将非零元素插入到其所在列的环形链表*/
  while (q->down!=h[c] && q->down->row<r)
    q=q->down;
  p->down=q->down;
  q->down=p;
}
}
```

2、稀疏矩阵十字链表的查找算法

```
int locatespmatrix(headspmatrix h, int x, int *rowx,
                  int *colx)
{ spmatrix p, q;
  p=h[0]->tag.next;
  while (p!=h[0])
  { q=p->right;
    while (p!=q)
    {if (q->tag.val==x)
      { *rowx=q->row;  *colx=q->col; return(1); }
      q=q->right;
    }
    p=p->tag.next;          /*准备进入下一行查找*/
  }
  return(0);
}
```

第4章 字符串、数组 和特殊矩阵

➤字符串

➤字符串的模式匹配

➤数组

➤特殊矩阵

➤稀疏矩阵

➤作业： 1, 2, 3, 4, 6, 7, 8, 9, 10

第6章 树型结构

➤树的基本概念

➤树类的定义

➤树的存储结构

➤树的遍历

➤树的线性表示

6.1 树的基本概念

树是由 n ($n \geq 0$)个结点构成的有限集合, $n=0$ 的树称为**空树**; 当 $n \neq 0$ 时, 树中的结点应该满足以下两个条件:

- (1) 有且仅有一个特定的结点称之为**根**;
- (2) 其余结点分成 m ($m \geq 0$)个互不相交的有限集合 T_1, T_2, \dots, T_m , 其中每一个集合又都是一棵树, 称 T_1, T_2, \dots, T_m 为根结点的**子树**。

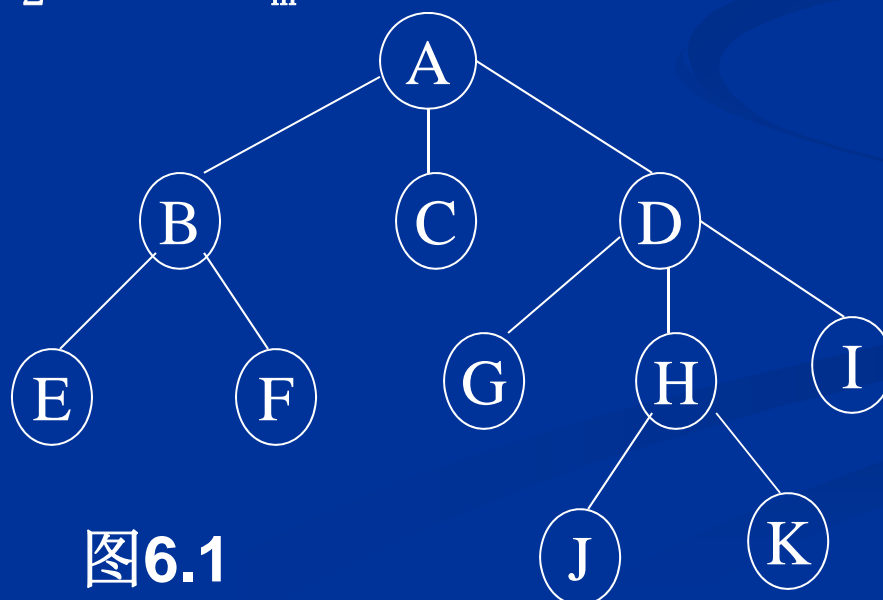


图6.1

在树中采用线段连接两个相关联的结点，如A和B，D和H等。其中A和D是上端结点，B和H是下端结点。称A、D分别是B、H的**双亲**（或**父母**或**前件**），B和H分别为A和D的**子女**（或**孩子**或**后件**）。显然，双亲和子女的关系是相对而言的。图6.1中，B是A的子女，但又是E和F的双亲。由于E和F的双亲为同一结点，称E和F互为**兄弟**。在任何一棵树中，除根结点外，其它任何一个结点有且仅有一个双亲，有0个或多个子女，且它的子女恰巧为其子树的根结点。我们将一结点拥有的子女数称为该**结点的度**，树中所有结点度的最大值称为**树的度**。图6.1中，A的度为3，B的度为2，而C的度为0，整棵树的度为3。称度为0的结点为**终端结点**或**叶子结点**，称度不为0的结点为**非终端结点**或**分支结点**。显然，A、B、D、H均为分支结点，而E、F、C、G、J、K、I均为叶子结点。

称树中连接两个结点的线段为**树枝**。在树中，若从结点 K_i 开始沿着树枝自上而下能到达结点 K_j ，则称从 K_i 到 K_j 存在一条**路径**，路径的长度等于所经过的树枝的条数。在图6.1中，从结点A到结点J存在一条路径，路径的长度为3；从D到K也存在一条路径，路径的长度为2。仔细观察不难发现，从树根到树中任何一个结点均存在一条路径。

将从树根到某一结点 K_i 的路径中 K_i 前所经过的所有结点称为 K_i 的**祖先**；反之，以某结点 K_i 为根的子树中的任何一个结点都称为 K_i 的**子孙**。图6.1中，A、D、H均为J和K的祖先，而G、H、I、J和K均为D的子孙。

树中**结点的层次**：从树根开始定义，根结点为第一层，根的子结点构成第二层，依次类推，若某结点 K_j 位于第 i 层，则其子女就位于第 $i+1$ 层。称树中结点的最大层次数为树的**深度或高度**。图6.1中，A结点位于第一层，B、C、D位于第2层，E、F、G、H和I位于第三层等等，整棵树的高度为4。

若树中任意结点的子树均看成是从左到右有次序的，不能随意交换，则称该树是**有序树**；否则称之为**无序树**。下图6.3中的两棵树，若看成是有序树，它们是不等价的；若看成是无序树，两者相等。

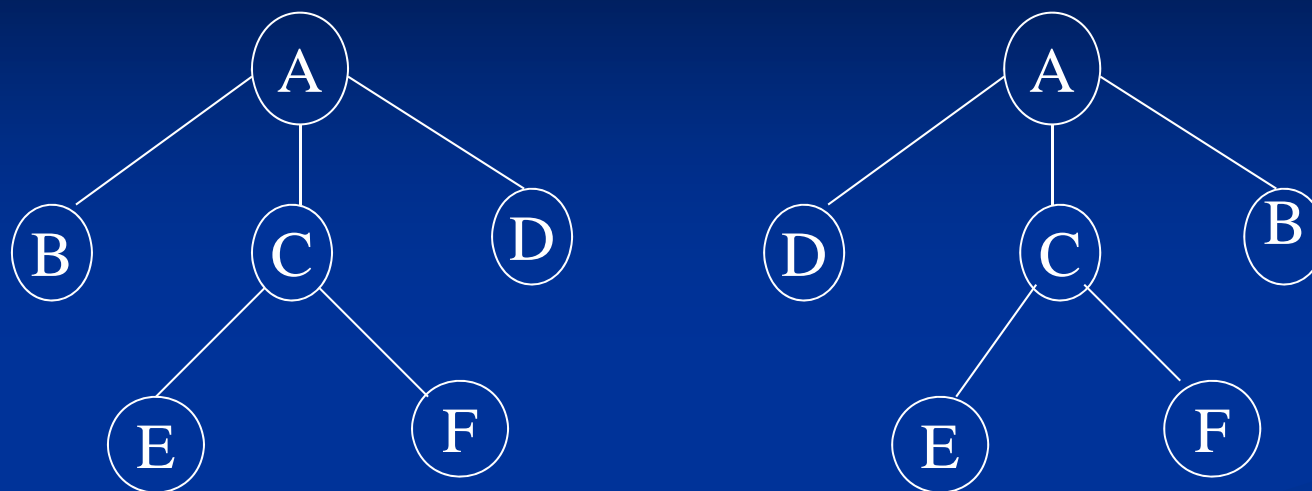


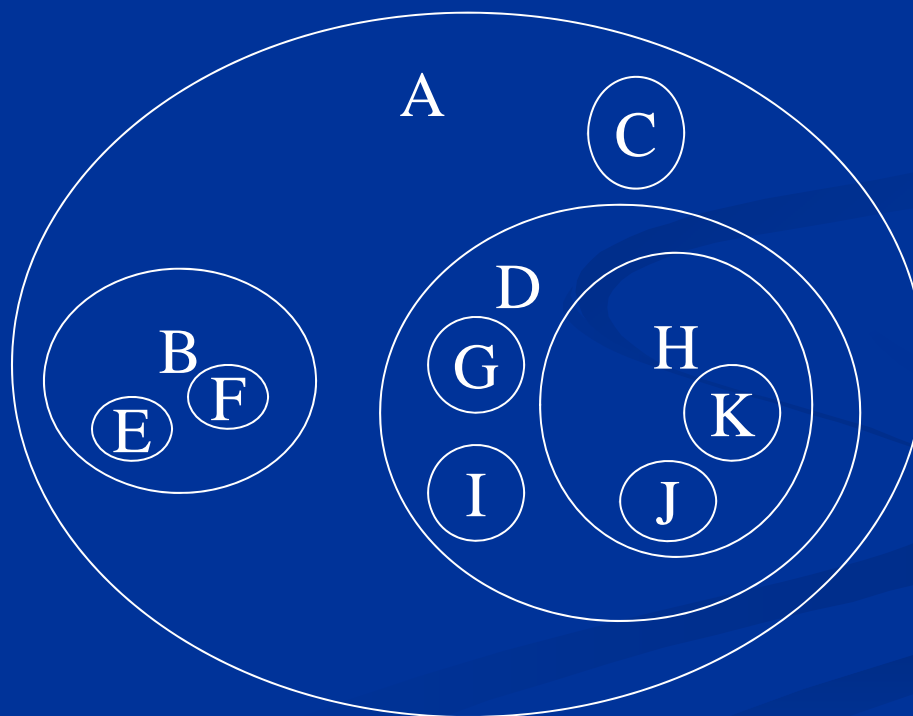
图6.3 有序树和 unordered 树的比较

由 m ($m \geq 0$) 棵互不相交的树构成的集合称为森林。森林和树的概念十分相近，一棵树中每个结点，其子树的集合即为一个森林；而在森林中的每棵树之上加一个共同的根，森林就成为了一棵树。

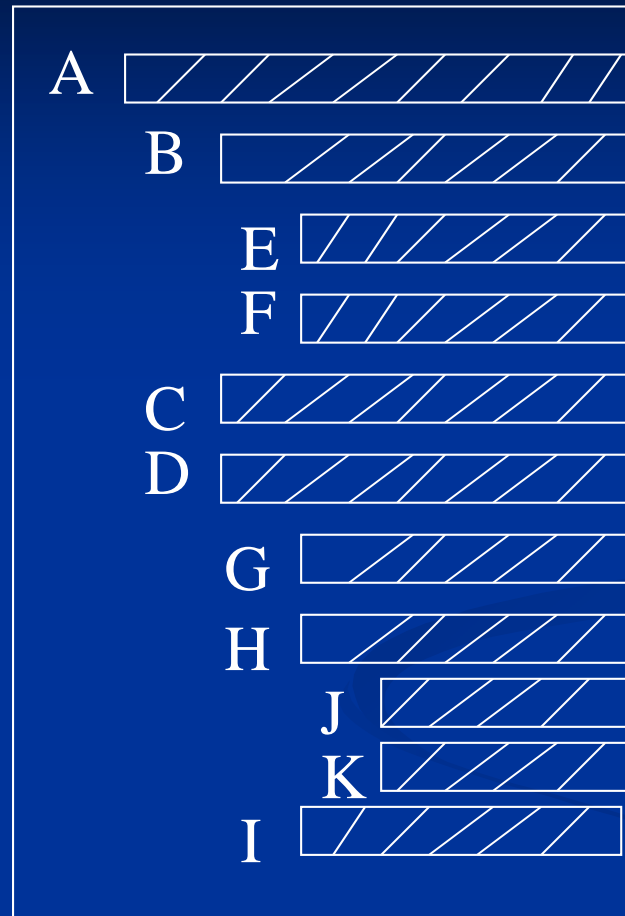
树型结构的其它表示方法:

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

(a) 图6.1的括号表示法



(b) 图6.1的嵌套集合表示法



(C) 图6. 1的凹入表示法

第6章 树型结构

- 树的基本概念

 - 树类的定义

 - 树的存储结构

 - 树的遍历

- 树的线性表示

6.2 树类的定义

ADT tree {

数据对象D: 具有相同性质的数据元素构成的有限集合。

数据关系R: 如果D为空或D仅含一个元素, 则R为空; 否则D中存在一个特殊的结点root, 称之为根结点, 其无前驱; 其它结点被分成互不相交的 $m(m \geq 0)$ 个集合, 分别构成root的 m 棵子树; 若这些子树非空, 则它们的根结点 $root_i$ 均称为整棵树根结点root的后继结点; 而每棵子树也是一棵树, 因而它们中数据元素间的关系也同样满足R的定义。

树的基本操作如下:

- (1) Inittree(T)
- (2) Cleartree(T)


```
(3) Emptytree(T)
(4) Root(T)
(5) Child(T, a, i)
(6) Parent(T, a)
(7) Degree(T, a)
(8) Depth(T)
(9) Choose(T , C)
(10) Addchild (T,a, i, t1)
(11) Delchild(T,a,i)
(12) Createtree(a, F)
(13) Equaltree(T1,T2)
(14) Numofnode(T)
(15) preorder(T)
(16) postorder(T)
(17) levelorder(T)
(18) Destroytree(T)
```

```
} ADT Tree
```

第6章 树型结构

- 树的基本概念

- 树类的定义

- 树的存储结构

- 树的遍历

- 树的线性表示

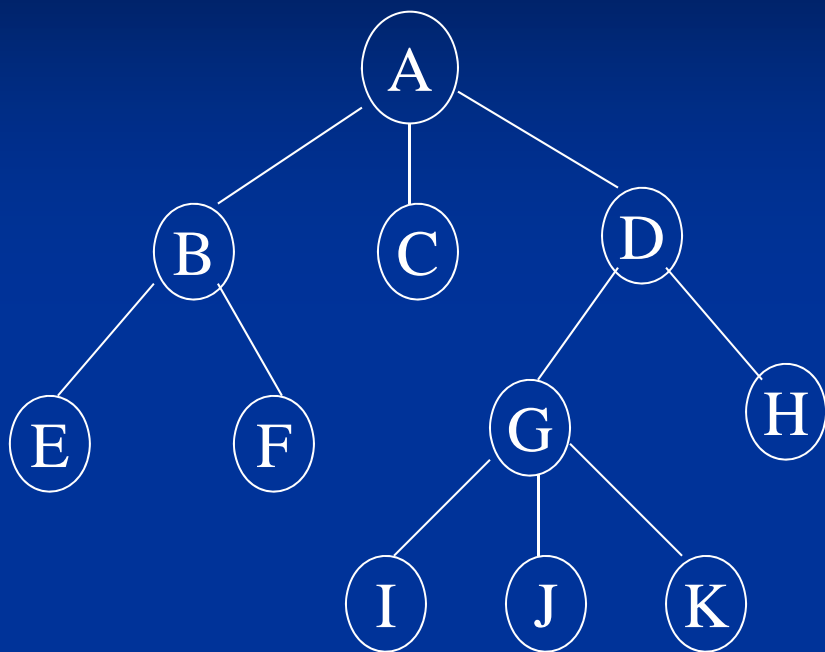
6.3 树的存储结构

根据数据元素之间关系的不同表示方式，常用的树存储结构主要有三种：双亲表示法、孩子表示法和孩子兄弟表示法。

6.3.1 双亲表示法

在树中，除根结点没有双亲外，其他每个结点的双亲是唯一确定的。因此，根据树的这种性质，存储树中结点时，可以包含两个信息：结点的值data和体现结点之间相互关系的属性__该结点的双亲parent。借助于每个结点的这两个信息便可唯一地表示任何一棵树。这种表示方法称为**双亲表示法**。

```
#define MAXSIZE 100
typedef char datatype;      /*结点值的类型*/
typedef struct node         /*结点的类型*/
{
    datatype data;
    int parent;             /*结点双亲的下标*/
} node;
typedef struct tree
{
    node treelist[MAXSIZE]; /*存放结点的数组*/
    int length, root ;      /* 树中实际所含结点的
                               个数及根结点的位置*/
} tree;
```



(a) 一棵树

	data	parent	
0	A	-1	← root
1	B	0	
2	C	0	
3	D	0	
4	E	1	
5	F	1	
6	G	3	
7	H	3	
8	I	6	
9	J	6	
10	K	6	

(b) (a)图的双亲表示法

图6.4

6.3.2 孩子表示法

采用孩子表示法表示一棵树时，树中每个结点除了存储其自身的值之外，还必须指出其所有子女的位置，即整棵树中所有结点的相互关系是通过指明结点子女的位置来体现的，称这种表示法为孩子表示法。根据子女位置的实现方法不同，孩子表示法分为三种：指针方式的孩子表示法、数组方式的孩子表示法、链表方式的孩子表示法。

1、指针方式的孩子表示法

指针方式的孩子表示法中每个结点通常包含两个域：一个是元素的值域data，另一个为指针数组，数组中的每个元素均为一个指向该结点子女的指针；一棵 m 度的树，其指针数组的大小即为 m 。

```
#define m 3                      /*树的度数*/
typedef char datatype;           /*结点值的类型*/
typedef struct node
{   /*结点的类型*/
    datatype data;
    struct node *child[m]; /*指向子女的指针数组*/
} node, *tree;
tree root;
```

其中root表示指向树根结点的指针。

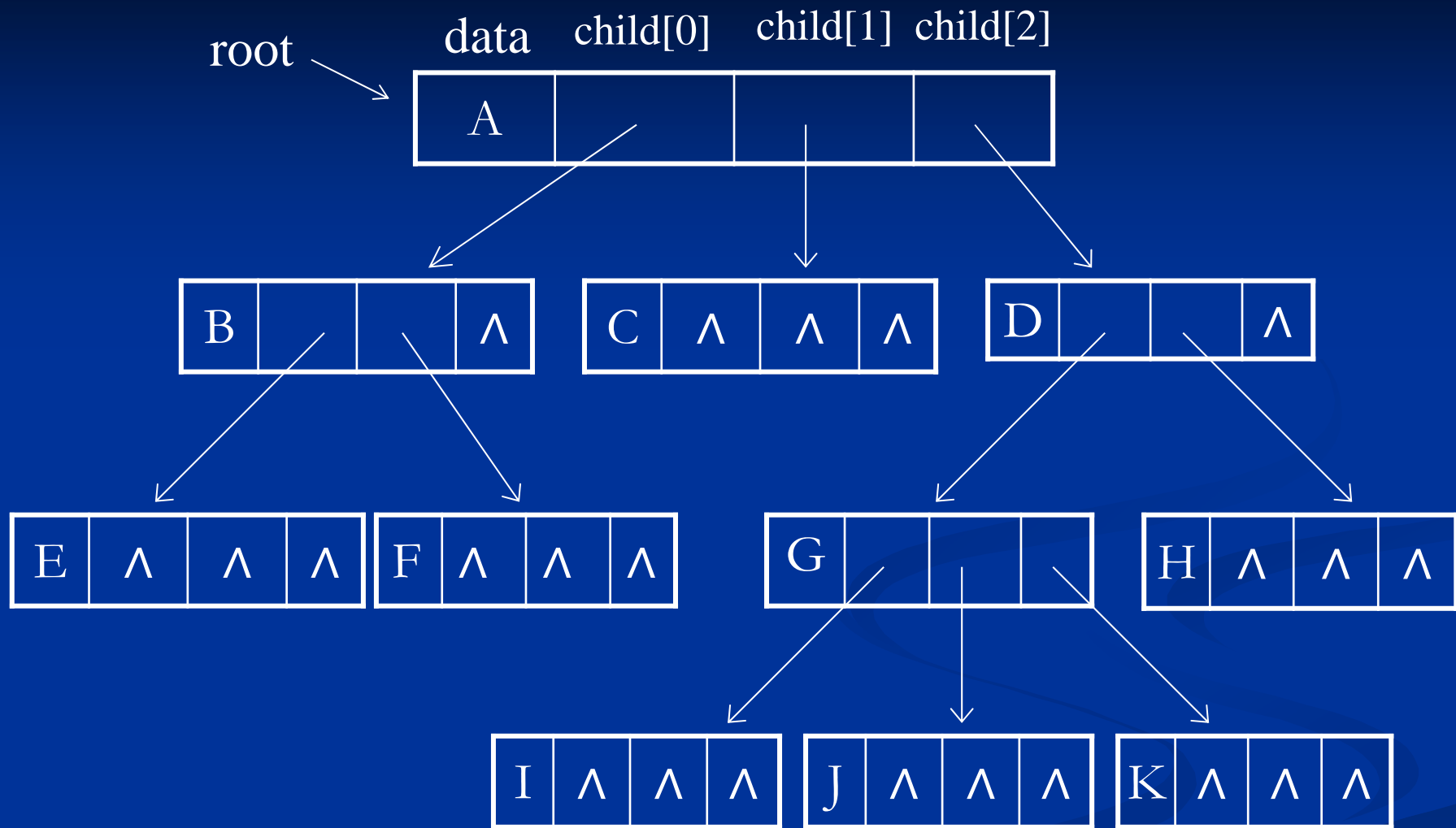


图6.4中(a)图的指针方式的孩子表示法

2、数组方式的孩子表示法

为了查找方便，可以将树中的所有结点存储在一个一维数组中，这样每个结点子女的位置便可以通过数组的下标来体现，称这种孩子表示法为数组方式的孩子表示法。

```
#define m 3
#define MAXSIZE 20
typedef char datatype;
typedef struct node {
    datatype data;
    int child[m];
} treenode;
treenode tree[MAXSIZE];
int root ; int length;
```

root →		data	child[0]	child[1]	child[2]
	0	A	1	2	3
	1	B	4	5	-1
	2	C	-1	-1	-1
	3	D	6	7	-1
	4	E	-1	-1	-1
	5	F	-1	-1	-1
	6	G	8	9	10
	7	H	-1	-1	-1
	8	I	-1	-1	-1
	9	J	-1	-1	-1
	10	K	-1	-1	-1

图6.4中(a)图的数组方式的孩子表示法

3、链表方式的孩子表示法

在树的链表方式的孩子表示法中，把每个结点的子女排列起来形成一个单链表，这样n个结点就形成n个单链表；而n个单链表的头指针又组成一个线性表，为了查找方便，使用数组加以存储。

```
# define MAXSIZE 50
typedef char datatype;
typedef struct chnode { /*孩子结点的类型*/
    int child;
    struct chnode *next;
} chnode, * chpoint;
```

```
typedef struct { /* 树中每个结点的类型 */
    datatype data;
    chpoint firstchild; /*指向第一个子女的指针*/
} node;
typedef struct { /*树的类型*/
    node treelist [MAXSIZE];
    int length, root;
} tree;
```

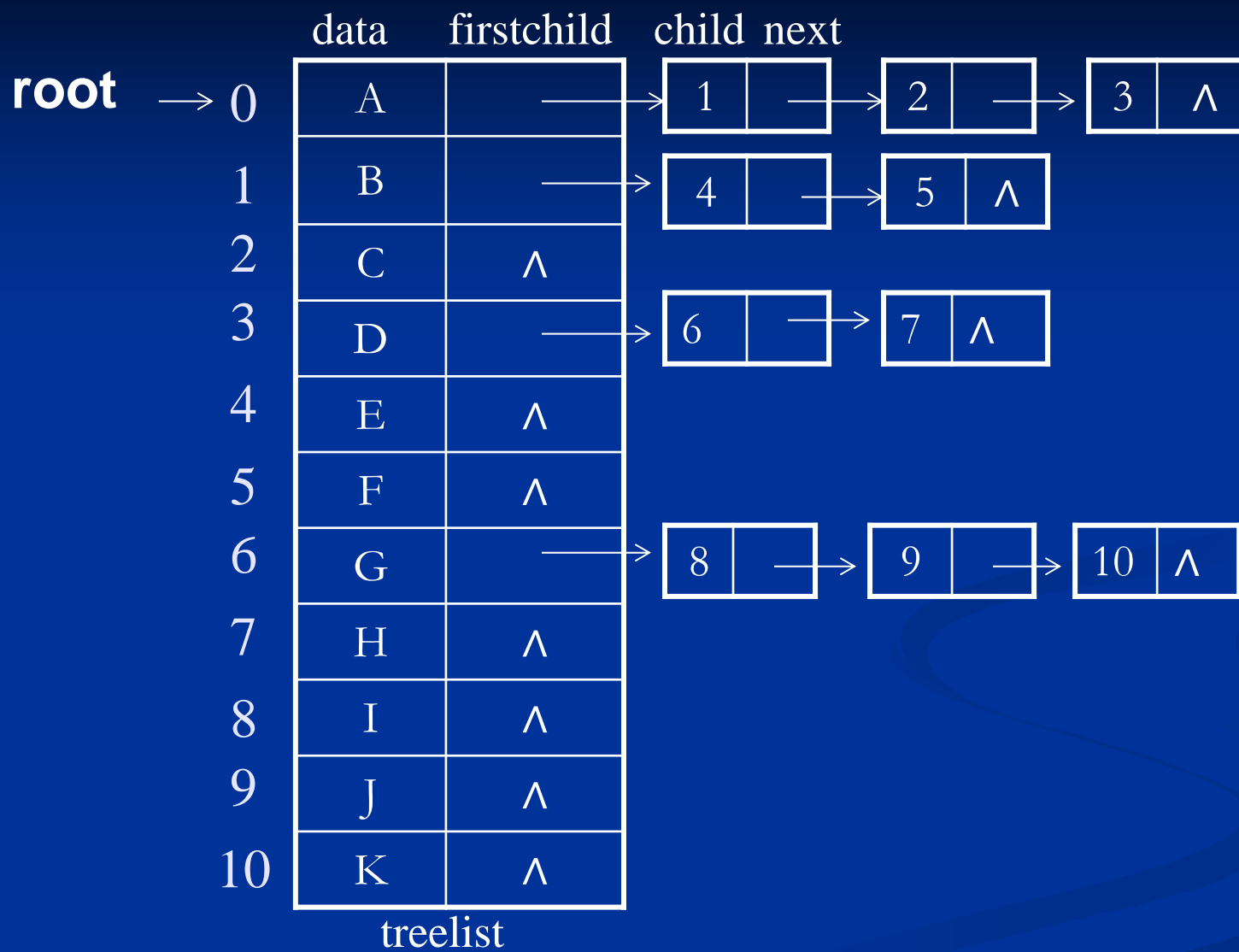


图6.4中(a)图的链表方式的孩子表示法

6.3.3 孩子兄弟表示法

所谓孩子兄弟表示法，即在存储树中每个结点时，除了包含该结点值域外，还设置两个指针域firstchild和rightsibling，分别指向该结点的第一个子女和其右兄弟，即以二叉链表方式加以存储，因此该方法也常被称为二叉树表示法。

```
typedef char datatype; /*树中结点值的类型*/
typedef struct node { /*树中每个结点的类型*/
    datatype data;
    struct node * firstchild, *rightsibling;
} node, * pnode;

pnode root; /*指向树根结点的指针*/
```

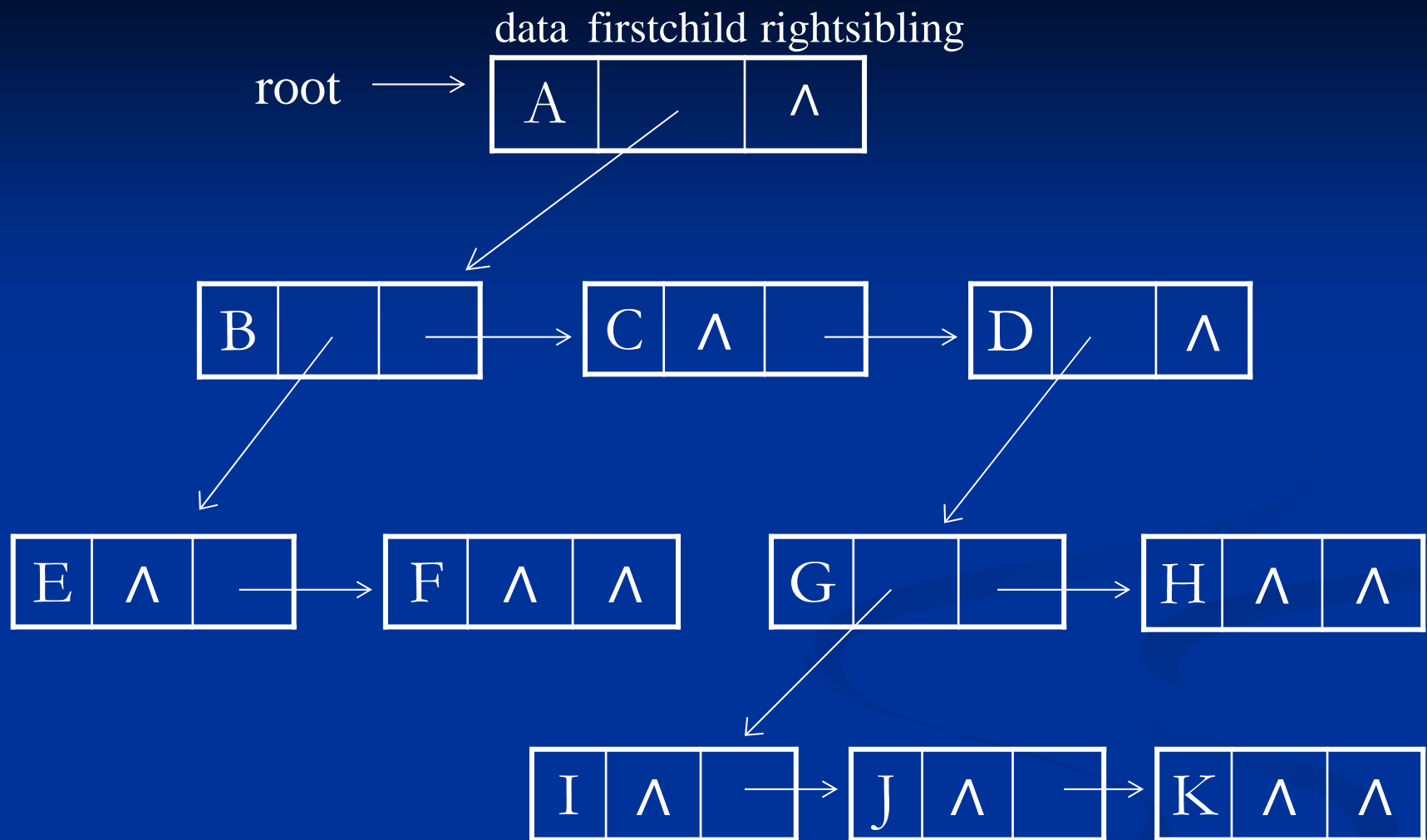


图6.4中(a)图的孩子兄弟表示法

第6章 树型结构

- 树的基本概念

 - 树类的定义

 - 树的存储结构

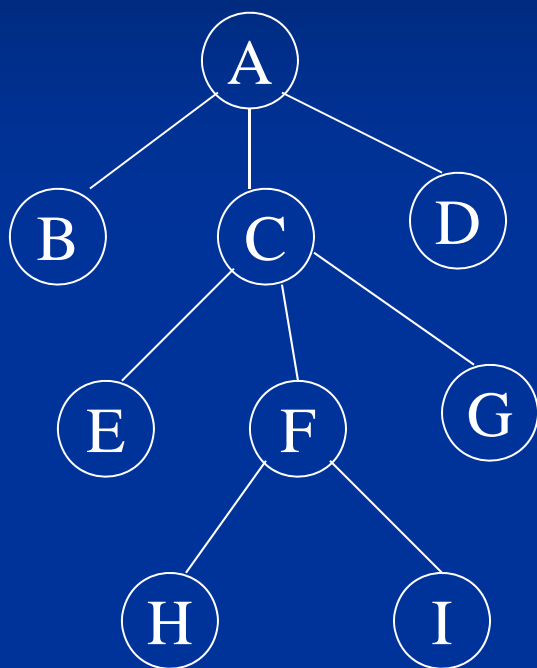
 - 树的遍历

- 树的线性表示

6.4 树的遍历

所谓**树的遍历**，指按某种规定的顺序访问树中的每一个结点一次，且每个结点仅被访问一次。树的遍历方式分为以下三种：

- (1) 树的**前序遍历**：首先访问根结点，再依次按前序遍历的方式访问根结点的每一棵子树。
- (2) 树的**后序遍历**：首先按后序遍历的方式访问根结点的每一棵子树，然后再访问根结点。
- (3) 树的**层次遍历**：首先访问第一层上的根结点，然后从左到右依次访问第二层上的所有结点，再以同样的方式访问第三层上的所有结点，……，最后访问树中最低一层的所有结点。



前序遍历的结果:

ABCEFHIGD

后序遍历的结果:

BEHIFGCDA

层次遍历的结果:

ABCDEFGHI

以下以指针方式的孩子表示法作为树的存储结构，分别实现树的各种遍历算法。

1、树的前序遍历的递归实现

```
void preorder(tree p) /*p为指向树根结点的指针*/
{
    int i;
    if (p!=NULL) /*树不为空*/
    {
        printf("%c", p->data);
        for (i=0; i<m; ++i)
            preorder(p->child[i]);
    }
}
```

2、树的后序遍历的递归实现

```
void postorder(tree p)
    /*p为指向树根结点的指针*/
{
    int i;
    if (p!=NULL)    /*树不为空*/
    {
        for (i=0;i<m;++i)
            postorder(p->child[i]);
        printf("%c", p->data);
    }
}
```

3、按前序遍历顺序建立一棵3度树的递归算法

```
void createtree (tree *p )
{ int i; char ch;
  if ((ch=getchar())== ' ') *p=NULL;
  else
  { *p=(tree) malloc (sizeof(node));
    /*产生树的根结点*/
    (*p)->data=ch;
    for (i=0;i<m;++i)
      /*按前序遍历顺序依次产生每棵子树*/
      createtree(&(*p)->child[i]);
    }
}
```

4、树的层次遍历算法

在树的层次遍历过程中，对于某一层上的每个结点被访问后，应立即将其所有子女结点按从左到右的顺序依次保存起来，该层上所有结点的这些子女结点正好构成下一层的所有结点，接下来应该被访问的就是它们。显然，这里用于保存子女结点的数据结构选择队列最合适，队列中的每个元素均为在排队等待访问的结点。

由于树的层次遍历首先访问的是根结点，因此初始时队列中仅包含根结点。只要队列不为空，就意味着还有结点未被访问，遍历就必须继续进行；每次需访问一个结点时均取队头元素，访问完成后，若其子女非空，则将其所有子女按顺序依次进队；不断重复以上过程，直到队列为空。

```
void levelorder(tree t)
{tree queue[20]; /*存放等待访问的结点队列*/
  int f,r,i; /*f、r分别为队头、队尾指针*/
  tree p;
  f=0; r=0; queue[0]=t;
  while (f<=r) /*队列不为空*/
  { p=queue[f]; f++; printf("%c",p->data);
    for (i=0;i<m;++i)
      if (p->child[i])
      {
        ++r; queue[r]=p->child[i];
      }
  }
}
```

第6章 树型结构

- 树的基本概念

 - 树类的定义

 - 树的存储结构

 - 树的遍历

- 树的线性表示

6.5 树的线性表示

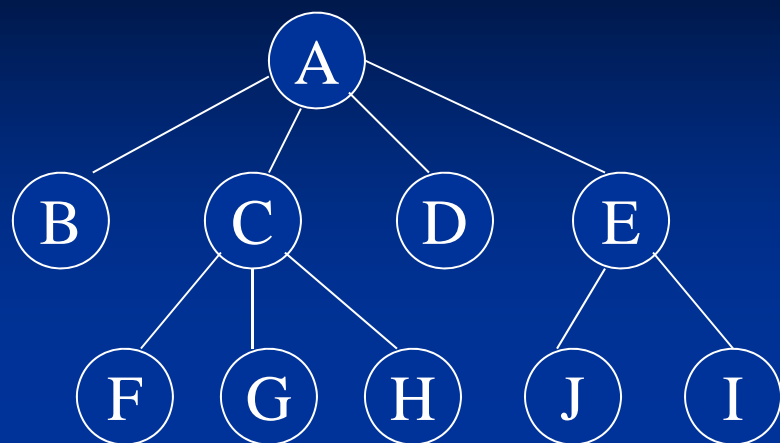
树的线性表示便于树的输入、输出，同时在存储时也比较节省存储空间。本节主要介绍树的两种线性表示方法：括号表示法和层号表示法。

6.5.1 树的括号表示

1、树的括号表示的规则为：

- (1) 若树 T 为空树，则其括号表示为空；
- (2) 若树 T 只包含一个结点，则其括号表示即为该结点本身；
- (3) 如果树 T 由根结点 A 和它的 m 棵子树 T_1, T_2, \dots, T_m 构成，则其括号表示为：

$A(T_1 \text{的括号表示}, T_2 \text{的括号表示}, \dots, T_m \text{的括号表示})$
其中子树的括号表示同样应该遵循以上规则。



$A(B, C(F, G, H), D, E(J, I))$

图6.10

2、树的括号表示具有以下特点：

(1) “(”前面的元素一定为某棵树或子树的根结点，而其所有子树中的结点一定位于该“(”和与之对应的“)”之间；

(2) 任何“(”和与之配对的“)”之间的括号表示序列同样满足(1)中的性质。

3、树的括号表示到树的孩子表示的转换算法

- (1) 从左到右扫描树的括号表示;
- (2) 每当遇到左括号时, 其前一个结点进栈, 并读下一个符号;
- (3) 每当遇到右括号时, 栈顶元素出栈。说明以栈顶元素为根的子树构造完毕, 此时若栈为空, 算法结束, 否则读下一个符号;
- (4) 每当遇见结点, 则它一定为栈顶元素的子女, 将其挂到栈顶元素的某子女位置上, 并读下一个符号;
- (5) 每当遇到 “ , ” , 则滑过该符号, 并读下一个符号。

```

#define m 3          /* 树的度数*/
#define MAXSIZE 20 /* 树的孩子表示法对应的数组大小*/
#define BMAXSIZE 50 /*树的括号表示对应的数组大小*/
typedef char datatype; /* 树中结点值的类型*/
typedef struct node { /*树的孩子表示法中结点的类型*/
    datatype data;
    int child[m];
} treenode;
treenode tree[MAXSIZE]; /*树孩子表示法的存储数组*/
int root ; /*根结点的下标*/
int length; /*树中实际所含结点的个数*/
char p[BMAXSIZE]; /*存放树括号表示的数组*/
void bracktotree(char p[],int *root, int *length,treenode tree[])
{ /*将树的括号表示法转换成树的孩子表示法*/
    int stack[MAXSIZE]; int top; int i,j,k,l,done;
    k=0; j=0; *root=0; top=-1; done=1;
    tree[j].data=p[k]; ++k;
    for (i=0;i<m;++i) tree[j].child[i]=-1;

```

```

while (done)
    { if (p[k]== '(' )
        { ++top;  stack[top]=j; ++k; }
      else if (p[k]==')')
        {--top; if (top==-1)  done=0;  else ++k;}
      else if (p[k]==' , ') ++k;
      else { ++j;  tree[j].data=p[k];
            for (i=0;i<m;++i)
                tree[j].child[i]=-1;
            l=stack[top];  i=0;
            while (tree[l].child[i]!=-1)
                ++i;
            tree[l].child[i]=j;  ++k;
            }
    }
*length=j;
}

```

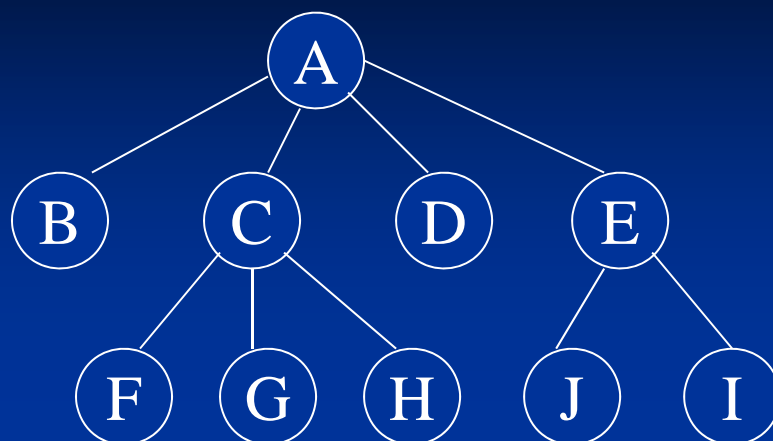
6.5.2 树的层号表示

设 j 为树中的一个结点，若为 j 赋予的一个整数值 $lev(j)$ 满足以下两个条件：

- (1) 如果结点 i 为 j 的后件，则 $lev(i) > lev(j)$ ；
- (2) 如果结点 i 与 j 为同一结点的后件，则 $lev(i) = lev(j)$ 。

称满足以上条件的整数值 $lev(j)$ 为结点 j 的层号。

树的层号表示为：首先根据层号的定义为树中的每个结点规定一个层号，然后按前序遍历的顺序写出树中所有的结点，并在每个结点之前加上其层号即可。



以下是上图中树的两种层号表示:

- ① 10A, 20B, 20C, 30F, 30G, 30H, 20D, 20E, 40J, 40I
- ② 1A, 2B, 2C, 5F, 5G, 5H, 2D, 2E, 3J, 3I

树的层号表示到树的扩充孩子表示转换算法：

- (1) 从前往后扫描树的层号表示；
- (2) 若结点 i 的层号比其前一个结点 j 的层号大，说明结点 i 位于结点 j 的下一层，且正好为 j 的第一个子女；
- (3) 若结点 i 的层号与结点 j 的层号相等，说明两结点位于同一层，它们拥有共同的双亲；
- (4) 若结点 i 的层号比结点 j 的层号小，说明结点 i 与结点 j 的某个祖先结点互为兄弟，于是应该沿着 j 的双亲向树根方向寻找 i 的兄弟，从而找到它们共同的双亲。


```
#define m 3
#define MAXSIZE 20
typedef char datatype;
typedef struct node {
    datatype data;
    int child[m];
    int parent;
} treenode;
typedef struct { /*层号表示法中结点的类型*/
    datatype data;
    int lev; /*存储结点的层号*/
} levelnode;
treenode tree[MAXSIZE];
    int root ;
    int length;

levelnode ltree[MAXSIZE];
```

```
void leveltotree(int length, levelnode ltree[],
                int *root, treenode tree[])
{ /*将树的层号表示法转换成树的扩充孩子表示法*/
    int i, j, k;
    for (i=0; i<length; ++i)
        for (j=0; j<m; ++j)
            tree[i].child[j]=-1;
    *root=0; tree[0].data=ltree[0].data;
    tree[0].parent=-1;
    for (i=1; i<length; ++i)
    { tree[i].data=ltree[i].data;
      j=i-1;
      if (ltree[i].lev>ltree[j].lev)
      {
          tree[i].parent=j; tree[j].child[0]=i;
      }
    }
```

```
else {  
    while (ltree[i].lev<ltree[j].lev)  
        j=tree[j].parent;  
    tree[i].parent=tree[j].parent;  
    j=tree[j].parent;  
    k=0;  /*将结点i挂到双亲结点上*/  
    while (tree[j].child[k]!=-1)  
        ++k;  
    tree[j].child[k]=i;  
}  
  
}  
  
}
```

第6章 树型结构

- 树的基本概念

- 树类的定义

- 树的存储结构

- 树的遍历

- 树的线性表示

- 作业： 1, 2, 3, 4, 5, 9

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

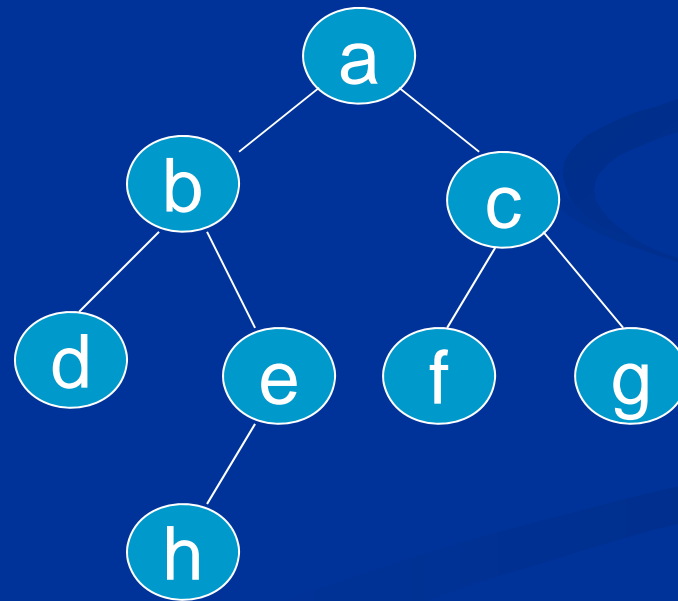
- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7.1 二叉树的基本概念

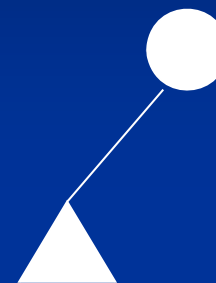
二叉树的定义为：**二叉树**是一个由结点构成的有限集合，这个集合或者为空，或者由一个根结点及两棵互不相交的分别称作这个根的左子树和右子树的二叉树组成。当二叉树的结点集合为空时，称为**空二叉树**。



二叉树有以下五种基本形态：

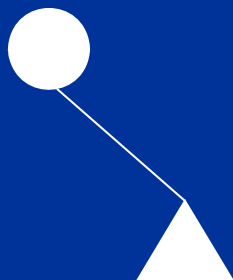


(a) 空二叉树

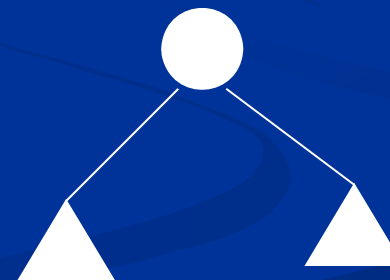


(b) 根和空的左、右子树

(c) 根和非空左子树、空右子树



(d) 根和空左子树、非空右子树



(e) 根和非空的左、右子树

树型结构中使用的术语如父母（双亲或前件）、子女（后件）、祖先、子孙、兄弟和路径等在二叉树中仍然可以沿用，但值得注意的是，二叉树并非一般树型结构的特殊形式，它们为两种不同的数据结构。

二叉树与一般树型结构的主要区别在于：

- （1）二叉树中每个非空结点最多只有两个子女，而一般的树型结构中每个非空结点可以有0到多个子女；
- （2）二叉树中结点的子树要区分左子树和右子树，即使在结点只有一棵子树的情况下也要明确指出是左子树还是右子树。

二叉树具有以下重要性质：

性质1 一棵非空二叉树的第 i 层上至多有 2^{i-1} 个结点
($i \geq 1$)。

证明：当 $i=1$ 时，只有根结点，此时 $2^{1-1}=2^0=1$ ，显然上述性质成立；又由于在二叉树中每个结点最多只能具有两个子女，因而第 i 层上结点的最大个数是第 $i-1$ 层上结点的最大个数的两倍。于是第2层上结点的最大个数为2，第3层上结点的最大个数为4，……，则第 i 层上结点的最大个数即为 2^{i-1} 。

性质2 深度为 h 的二叉树至多有 2^h-1 个结点 ($h > 1$)。
根据性质1，深度为 h 的二叉树最多具有的结点的个数为 $2^0+2^1+2^2+\dots+2^{h-1}=2^h-1$ 。

性质3 对于任何一棵二叉树T，如果其终端结点数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0=n_2+1$ 。

证明：假设二叉树中总的结点个数为 n ，度为1的结点个数为 n_1 ，则有：

$$n=n_0+n_1+n_2$$

又由于在二叉树中除根结点外，其它结点均通过一条树枝且仅通过一条树枝与其父母结点相连，即除根结点外，其它结点与树中的树枝存在一一对应的关系；而二叉树中树枝的总条数为 n_1+2*n_2 ，因而二叉树总结点的个数为：

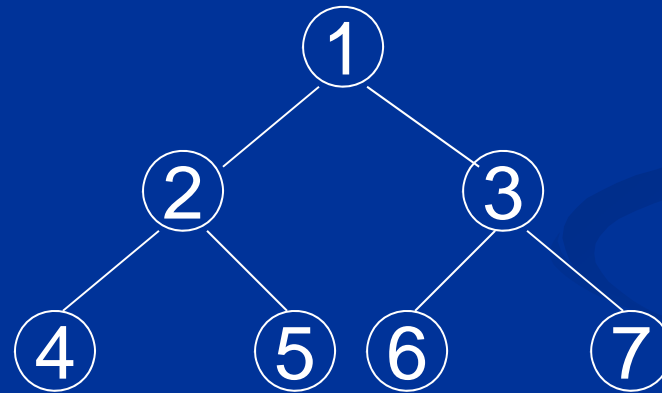
$$n=n_1+2*n_2+1$$

于是有：

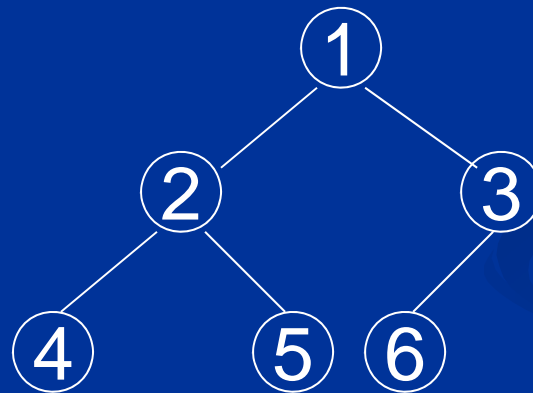
$$n_0+n_1+n_2=n_1+2*n_2+1$$

显然 $n_0=n_2+1$ 成立。

如果一棵二叉树中所有终端结点均位于同一层次，而其它非终端结点的度数均为2，则称此二叉树为**满二叉树**。在满二叉树中，若其深度为 h ，则其所包含的结点个数必为 2^h-1 。下图中的二叉树即为一棵深度为3的满二叉树，其结点的个数为 $2^3-1=7$ 。



如果一棵二叉树扣除其最大层次那层后即成为一棵满二叉树，且层次最大那层的所有结点均向左靠齐，则称该二叉树为**完全二叉树**。通俗地说，完全二叉树中只有最下面的两层结点的度数可以小于2，且最下面一层的结点都集中在该层最左边的若干位置上。下图所示的二叉树即为一棵深度为3的完全二叉树。



若对深度相同的满二叉树和完全二叉树中的所有结点按自上而下、同一层次按自左向右的顺序依次编号，则两者对应位置上的结点编号应该相同。

对于完全二叉树，具有以下性质：

性质4 对于具有 n 个结点的完全二叉树，如果按照从上到下、同一层次上的结点按从左到右的顺序对二叉树中的所有结点从1开始顺序编号，则对于序号为 i 的结点，有：

- (1) 如果 $i > 1$ ，则序号为 i 的结点其双亲结点的序号为 $\lfloor i/2 \rfloor$ ($\lfloor i/2 \rfloor$ 表示对 $i/2$ 的值取整)；如果 $i=1$ ，则结点 i 为根结点，没有双亲；
- (2) 如果 $2i > n$ ，则结点 i 无左子女（此时结点 i 为终端结点）；否则其左子女为结点 $2i$ ；
- (3) 如果 $2i+1 > n$ ，则结点 i 无右子女；否则其右子女为结点 $2i+1$ 。

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7.2 二叉树的基本运算

ADT bintree {

数据对象D: D是具有相同性质的数据元素构成的集合。

数据关系R: 如果D为空或D仅含一个元素, 则R为空;
否则D中存在一个特殊的结点root, 称之为根结点, 其无前驱; 其它结点被分成互不相交的两个集合, 分别构成root的左子树l和右子树r; 若l和r非空, 则它们的根结点lroot和rroot分别称为整棵二叉树根结点root的后继结点; 左子树l和右子树r也是二叉树, 因而它们中数据元素间的关系也同样满足R的定义。

二叉树的基本操作如下:

(1) createbintree(t)

(2) destroybintree(t)

```
(3) root(t)
(4) leftchild(t)
(5) rightchild(t)
(6) locate(t, x)
(7) parent(t, x)
(8) isempty(t)
(9) depth(t)
(10) numofnode(t)
(11) addchild(t, x, t1, b)
(12) deletechild(t, x, b)
(13) setnull(t)
(14) isequal(t1, t2)
(15) preorder(t)
(16) inorder(t)
(17) postorder(t)
(18) transform(t1, t2)
    } ADT bintree.
```


第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7.3 二叉树的存储结构

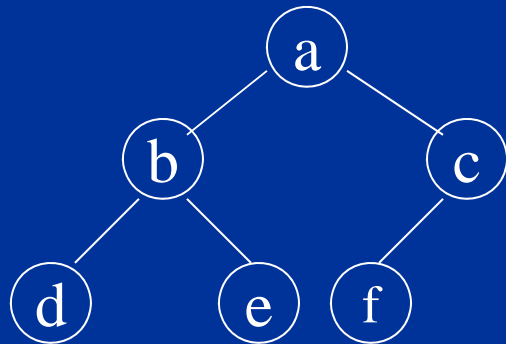
二叉树常用的存储结构有两种：顺序存储结构和链式存储结构。

7.3.1 顺序存储结构

顺序存储结构是使用一组连续的空间存储二叉树的数据元素和数据元素之间的关系。因此必须将二叉树中所有的结点排成一个适当的线性序列，在这个线性序列中应采用有效的方式体现结点之间的逻辑关系。

1、完全二叉树的顺序存储

对于一棵具有 n 个结点的完全二叉树，我们可以按从上到下、同一层次按从左到右的顺序依次将结点存入一个一维数组中。根据上述性质4，无须附加任何其它信息就能根据每个结点的下标找到它的子女结点和双亲结点。



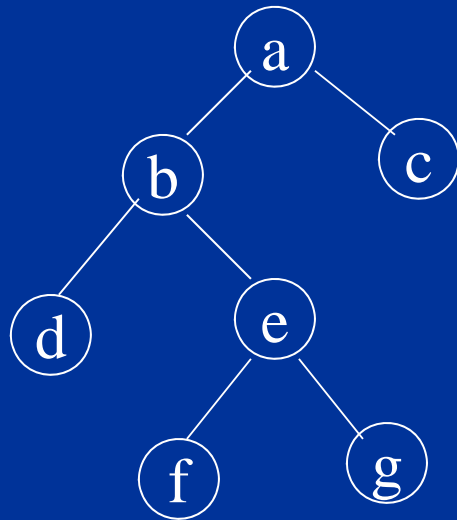
(a) 完全二叉树



(b) 完全二叉树的顺序存储

2 一般二叉树的顺序存储

由于二叉树中每个结点最多只有两个子女，于是存储一个结点时，除了包含结点本身的属性值外，另外增加两个域，分别用来指向该结点的两个子女在数组中的下标。



(a) 一棵二叉树

	0	1	2	3	4	5	6
lchild	1	3	-1	-1	5	-1	-1
data	a	b	c	d	e	f	g
rchild	2	4	-1	-1	6	-1	-1

root=0 n=7

(b) 二叉树的顺序存储

一般二叉树顺序存储数据结构的定义如下：

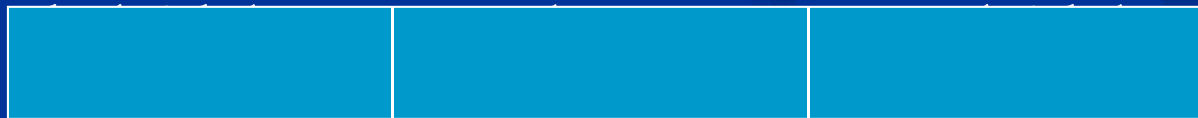
```
#define MAXSIZE 20
typedef char datatype; /*结点值的类型*/
typedef struct {
    datatype data;
    int lchild, rchild;
} node; /*二叉树结点的类型*/
node tree[MAXSIZE];
int n; /*树中实际所含结点的个数*/
int root; /*存放根结点的下标*/
```

带双亲指示的二叉树顺序存储数据结构的定义如下：

```
#define MAXSIZE 20
typedef char datatype;      /*结点值的类型*/
typedef struct {
    datatype data;
    int    lchild, rchild;
    int    parent;  /*存放双亲结点的下标*/
} node; /*二叉树结点的类型*/
node tree[MAXSIZE];
int  n;  /*树中实际所含结点的个数*/
int  root; /*存放根结点的下标*/
```

7.3.2 链式存储结构

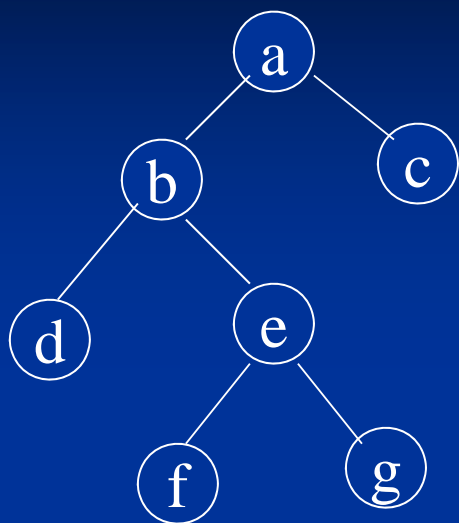
二叉树的链式存储方式下每个结点也包含三个域，分别记录该结点的属性值及左、右子树的位置。与顺序存储结构不同的是，其左、右子树的位置不是通过数组的下标，而是通过指针方式体现，如下图所示：



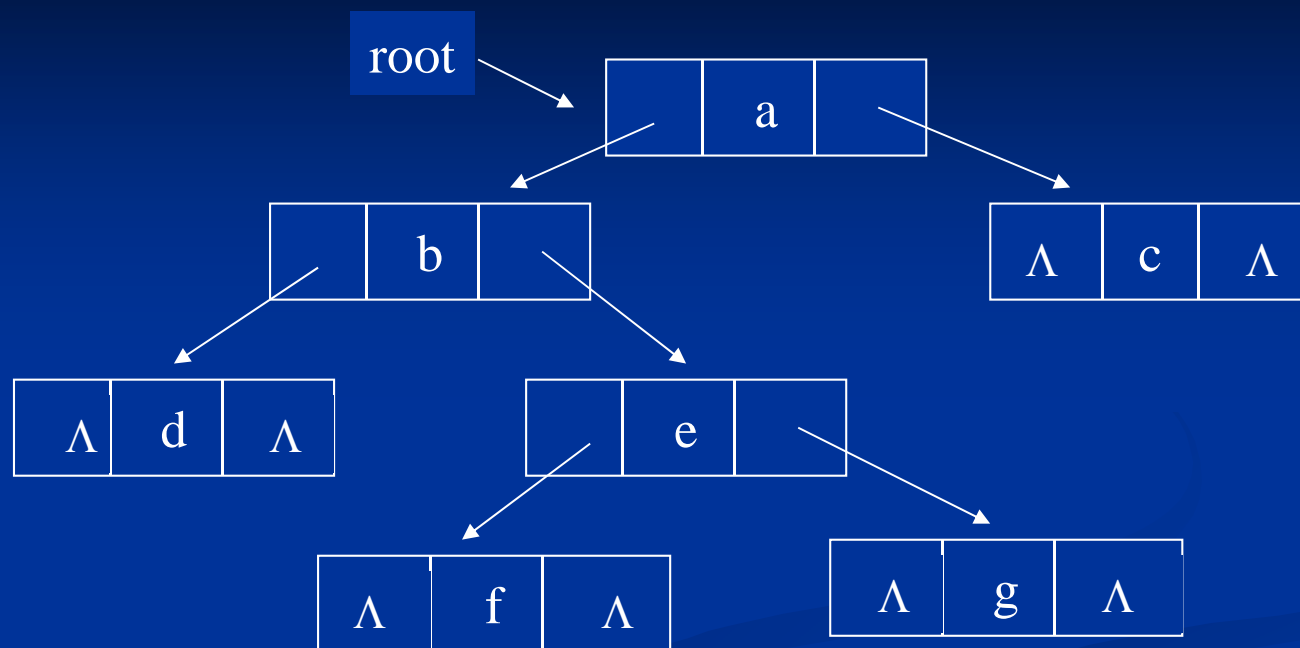
指针域

属性值

指针域



(a) 一棵二叉树



(b) 二叉树的链式存储

链式存储方式下二叉树结点数据结构的定义如下：

```
typedef char datatype; /*结点属性值的类型*/  
typedef struct node{ /*二叉树结点的类型*/  
    datatype data;  
    struct node *lchild, *rchild;  
} bintnode;  
typedef bintnode *bintree;  
bintree root;
```

链式存储方式下带双亲指针的二叉树结点数据结构的定义如下：

```
typedef char datatype; /*结点属性值的类型*/
typedef struct node{    /*二叉树结点的类型*/
    datatype data;
    struct node *lchild, *rchild;
    struct node *parent; /*指向双亲的指针*/
} bintnode;
typedef bintnode *bintree;
bintree root;    /*指向二叉树根结点的指针*/
```

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7.4 二叉树的遍历

7.4.1 二叉树遍历的定义

所谓**二叉树的遍历**，是指按一定的顺序对二叉树中的每个结点均访问一次，且仅访问一次。按照根结点访问位置的不同，通常把二叉树的遍历分为三种：**前序遍历**、**中序遍历**和**后序遍历**。

(1) 二叉树的前序遍历

首先访问根结点；

然后按照前序遍历的顺序访问根结点的左子树；

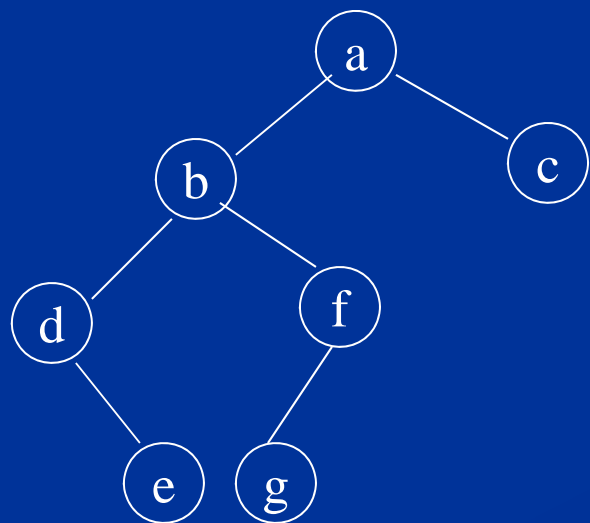
再按照前序遍历的顺序访问根结点的右子树。

(2) 二叉树的中序遍历

首先按照中序遍历的顺序访问根结点的左子树；
然后访问根结点；
最后按照中序遍历的顺序访问根结点的右子树。

(3) 二叉树的后序遍历

首先按照后序遍历的顺序访问根结点的左子树；
然后按照后序遍历的顺序访问根结点的右子树；
最后访问根结点。



前序遍历: abdefgc

中序遍历: debgfac

后序遍历: edgfbca

7.4.2 二叉树遍历的递归实现

由于二叉树的遍历是递归定义的，因此采用递归方式实现二叉树遍历的算法十分方便，只要按照各种遍历规定的次序，访问根结点时就输出根结点的值，访问左子树和右子树时进行递归调用即可。

1、前序遍历二叉树的递归算法

```
void preorder(bintree t)  
    { if (t) { printf("%c",t->data);  
        preorder(t->lchild);  
        preorder(t->rchild);  
    }  
  
}
```

2、中序遍历二叉树的递归算法

```
void inorder(bintree t)  
    { if (t) {  
        inorder(t->lchild);  
        printf("%c",t->data);  
        inorder(t->rchild); }  
    }
```

3、后序遍历二叉树的递归算法

```
void postorder(bintree t)  
    { if (t) {  
        postorder(t->lchild);  
        postorder(t->rchild);  
        printf("%c",t->data); }  
    }
```

4、二叉树的创建算法

利用二叉树前序遍历的结果可以非常方便地生成给定的二叉树，具体做法是：将第一个输入的结点作为二叉树的根结点，后继输入的结点序列是二叉树左子树前序遍历的结果，由它们生成二叉树的左子树；再接下来输入的结点序列为二叉树右子树前序遍历的结果，应该由它们生成二叉树的右子树；而由二叉树左子树前序遍历的结果生成二叉树的左子树和由二叉树右子树前序遍历的结果生成二叉树的右子树的过程均与由整棵二叉树的前序遍历结果生成该二叉树的过程完全相同，只是所处理的对象范围不同，于是完全可以使用递归方式加以实现。


```
void createbintree(bintree *t)
{ char ch;
  if ((ch=getchar())==' ') *t=NULL;
  else {
    *t=(bintnode *)malloc(sizeof(bintnode));
    /*生成二叉树的根结点*/
    (*t)->data=ch;
    createbintree(&(*t)->lchild);
    /*递归实现左子树的建立*/
    createbintree(&(*t)->rchild);
    /*递归实现右子树的建立*/
  }
}
```

7.4.3 二叉树遍历的非递归实现

在第5章，已经介绍了由递归程序转换成非递归程序的两种方法：简单递归程序的转换和复杂递归程序的转换；二叉树的遍历问题应该属于后者，即在采用非递归方式实现二叉树遍历时，必须使用一个堆栈记录回溯点，以便将来进行回溯。以下为一个顺序栈的定义及其部分操作的实现。

```
typedef struct stack      /*栈结构定义*/  
    { bintree data[100];  
      int tag[100];        /*为栈中每个元素设置的  
                           标记，用于后序遍历*/  
      int top;           /*栈顶指针*/  
    } seqstack;
```

```
void push(seqstack *s,bintree t) /*进栈*/  
{ s->data[++s->top]=t;  
}
```

```
bintree pop(seqstack *s) /*出栈*/  
{  
    if (s->top!=-1)  
        { s->top--;  
          return(s->data[s->top+1]); }  
    else  
        return NULL;  
}
```

1、二叉树前序遍历的非递归实现

前序遍历一棵非空树 t 时，访问完 t 的根结点后，就应该进入 t 的左子树，但此时必须将 t 保存起来，以便访问完其左子树后，进入其右子树的访问，即在 t 处必须设置一个回溯点；对 t 的左子树和右子树的遍历也是如此。仔细观察不难发现，这些回溯点应该使用栈来进行管理。在整个二叉树前序遍历的过程中，程序要做的工作始终分成两个部分：当前正在处理的树（子树）和保存在栈中待处理的部分，只有这两部分的工作均完成后，程序方能结束。

```
void preorder1(bintree t) /*非递归实现二叉树的前序遍历*/
{ seqstack s;
  s.top=-1;
  while ((t) || (s.top!=-1))
    /*当前处理的子树不为空或栈不为空则循环*/
    { while (t)
      { printf("%c ",t->data); s.top++;
        s.data[s.top]=t; t=t->lchild;
      }
      if (s.top>-1)
      { t=pop(&s);
        t=t->rchild;
      }
    }
}
```

2、 二叉树中序遍历的非递归实现

中序遍历一棵非空树 t 时，首先应该进入 t 的左子树访问，此时由于 t 的根结点及右子树尚未访问，因此必须将 t 保存起来，放入栈中，以便访问完其左子树后，从栈中取出 t ，进行其根结点及右子树的访问；对 t 的左子树和右子树的遍历也是如此。在整个二叉树中序遍历的过程中，程序要做的工作始终分成两个部分：当前正在处理的树（子树）和保存在栈中待处理的部分，只有这两部分的工作均完成后，程序方能结束。

```
void inorder1(bintree t)
{ seqstack s;
  s.top=-1;
  while((t!=NULL) || (s.top!=-1))
  { while (t)
    { push(&s,t); t=t->lchild; }
    if (s.top!=-1)
    { t=pop(&s);
      printf("%c ",t->data);
      t=t->rchild;
    }
  }
}
```

3 、二叉树后序遍历的非递归实现

后序遍历一棵非空树 t 时，首先应该进入 t 的左子树访问，此时由于 t 的右子树及根结点尚未访问，因此必须将 t 保存在栈中，以便访问完其左子树后，从栈中取出 t ，进行其右子树及根结点的访问。值得注意的是，当一个元素位于栈顶即将被处理时，其左子树的访问一定已经完成，如果其右子树尚未遍历，接下来应该进入其右子树的访问，而此时该栈顶元素是不能出栈的，因为其根结点还未被访问；只有等到其右子树也访问完成后，该栈顶元素才能出栈，并输出其根结点的值。因此一个元素位于栈顶时，必须设法识别其右子树是否已被访问。

解决的方法为：使用seqstack类型中的数组tag，用于标识栈中每个元素的状态：

- 每个元素刚进栈时，其tag值初始化为0；
- 当某一元素位于栈顶即将被处理时：
 - (1) 如果其tag值为0，意味着其右子树尚未访问，于是将其右子树作为当前处理的对象，此时该栈顶元素仍应该保留在栈中，并将其tag的值改为1；
 - (2) 如果其tag值为1，意味着其右子树已被访问，接下来应该访问其根结点，并将其出栈。

在整个二叉树后序遍历的过程中，程序要做的工作始终分成两个部分：当前正在处理的树（子树）和保存在栈中待处理的部分。只有这两部分的工作均完成后，程序方能结束。

```
void postorder1(bintree t)  
{ seqstack s;  
  s.top=-1;  
  while ((t)||(s.top!=-1))  
  { while (t)  
    { s.top++;  
      s.data[s.top]=t;  
      s.tag[s.top]=0;  
      t=t->lchild;  
    }  
  }
```

```
while ((s.top>-1)&& (s.tag[s.top]==1))
    { t=s.data[s.top];
      printf("%c ",t->data);
      s.top--;
    }
if (s.top>-1)
    { t=s.data[s.top];
      s.tag[s.top]=1;
      t=t->rchild;
    }
else t=NULL;
}
}
```

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7. 5 二叉树其它运算的实现

由于二叉树本身的定义是递归的，因此关于二叉树的许多问题或运算采用递归方式实现非常地简单和自然。

1、二叉树的查找locate(t, x)

```
bintree locate(bintree t, datatype x)
{ bintree p;
  if (t==NULL) return NULL;
  else if (t->data==x) return t;
  else
  { p=locate(t->lchild,x);
    if (p) return p;
    else return locate(t->rchild,x);
  }
}
```

2 统计二叉树中结点的个数numofnode(t)

```
int numofnode(bintree t)
{ if (t==NULL) return 0;
  else
    return(numofnode(t->lchild)+numofnode(t->rchild)+1);
}
```

3 、判断二叉树是否等价isequal(t1, t2)

```
int isequal(bintree t1, bintree t2)
{ int t;
  t=0;
  if (t1==NULL && t2==NULL) t=1;
  else if (t1!=NULL && t2!=NULL)
    if (t1->data==t2->data)
      if (isequal(t1->lchild,t2->lchild))
        t=isequal(t1->rchild,t2->rchild);
  return(t);
}
```

4、 求二叉树的高(深)度depth(t)

```
int depth(bintree t)
{ int h,lh,rh;
  if (t==NULL) h=0;
  else {
    lh=depth(t->lchild);
    rh=depth(t->rchild);
    if (lh>=rh) h=lh+1;
    else h=rh+1;
  }
  return h;
}
```

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

- 树、森林和二叉树的转换

7.6 穿线二叉树

7.6.1 穿线二叉树的定义

所谓穿线二叉树，即在一般二叉树的基础上，对每个结点进行考察。若其左子树非空，则其左指针不变，仍指向左子女；若其左子树为空，则让其左指针指向某种遍历顺序下该结点的前驱；若其右子树非空，则其右指针不变，仍指向右子女；若其右子树为空，则让其右指针指向某种遍历顺序下该结点的后继。如果规定遍历顺序为前序，则称为前序穿线二叉树；如果规定遍历顺序为中序，则称为中序穿线二叉树；如果规定遍历顺序为后序，则称为后序穿线二叉树。本小节主要介绍中序穿线二叉树。

在穿线二叉树的每个结点中，增加两个标志位：
ltag和rtag，其含义为：

ltag=0 表示结点的左指针指向其左子女；

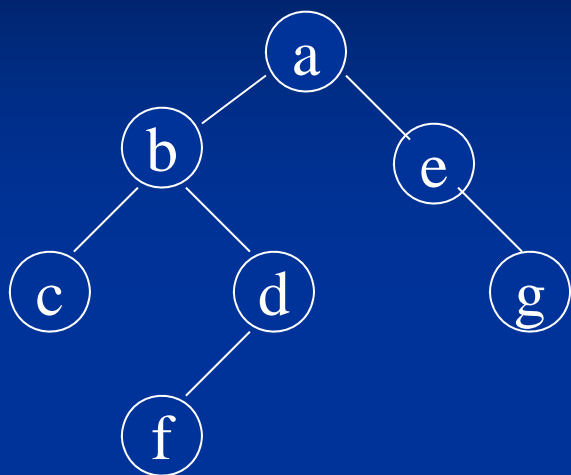
ltag=1 表示结点的左指针指向其中序遍历的前驱；

rtag=0 表示结点的右指针指向其右子女；

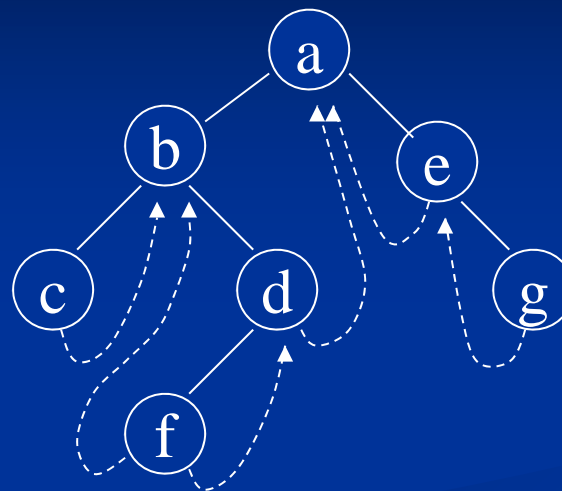
rtag=1 表示结点的右指针指向其中序遍历的后继。

每个结点的结构如下图所示：

ltag	lchild	data	rchild	rtag
------	--------	------	--------	------



(a)一棵二叉树



(b)中序穿线二叉树

(b)图中实线表示指针，虚线表示线索。

7.6.2中序穿线二叉树的基本运算

ADT binthrtree {

数据对象D: 具有相同性质的数据元素构成的有限集合;

数据关系R: 如果D为空或D仅含一个元素, 则R为空; 否则

D中存在一个特殊的结点root, 称之为根结点, 其无前驱; 其它结点被分成互不相交的两个集合, 分别构成root的左子树l和右子树r; 若l和r非空, 则它们的根结点lroot和rroot分别称为整棵二叉树根结点root的后继结点; 左子树l和右子树r也是二叉树, 因而它们中数据元素之间也同样满足上述关系。对于二叉树中的任何结点, 如其左子树非空, 则其lchild指向其左子树, 否则指向其中序遍历顺序下的前驱结点; 如其右子树非空, 则其rchild指向其右子树, 否则指向其中序遍历顺序下的后继结点。

基本操作集为:

- (1) createthrtree(p)
- (2) inthreading(p)
- (3) locate(p,x)
- (4) infirstnode(p)
- (5) inlastnode(p)
- (6) inprednode(p)
- (7) insuccnode(p)
- (8) preinsert(p,x,y)
- (9) succinsert(p,x,y)
- (10) delete(p,x)
- (11) inthrtree(p)
- (12) prethrtree(p)
- (13) postthrtree(p)

} ADT binthrtree

7.6.3 中序穿线二叉树的存储结构及其实现

1、中序穿线二叉树在链接方式下的数据类型定义

```
typedef char datatype;
typedef struct node
{
    datatype data;
    int ltag,rtag; /*左、右标志位*/
    struct node *lchild,*rchild;
}binthrnnode;

typedef binthrnnode *binthrtree;
```

2、 创建中序穿线二叉树createthrtree(p)

创建一棵中序穿线二叉树的办法之一为：首先建立一棵一般的二叉树，然后对其进行中序线索化。实现二叉树中序线索化可以借助于二叉树中序遍历的算法，只需将二叉树中序遍历算法中对当前结点的输出操作改为对该结点进行穿线；为了实现当前结点的穿线，必须设置一个指针pre，用于记录当前结点中序遍历的前驱结点。

```
binthrtree pre=NULL;    /*初始化前驱结点*/
void createbintree(binthrtree *t)
{ char ch;
  if ((ch=getchar())==' ') *t=NULL;
  else { *t=(binthrtree *)malloc(sizeof(binthrtree));
        (*t)->data=ch;
        createbintree(&(*t)->lchild);
        createbintree(&(*t)->rchild);
      }
}
```

```
void inthreading(binthrtree *p)
{ /*对二叉树进行中序线索化*/
if (*p)
{ inthreading(&((*p)->lchild));
  (*p)->ltag=((*p)->lchild)?0:1;
  (*p)->rtag=((*p)->rchild)?0:1;
  if (pre)
  { if(pre->rtag==1) pre->rchild=*p;
    if((*p)->ltag==1) (*p)->lchild=pre;
  }
  pre=*p;
  inthreading(&((*p)->rchild));
}
}

void createthrtree(binthrtree *p)
{ createbintree(p);
  inthreading(p); }
```


3、 中序遍历中序穿线二叉树 `inlhrtree(p)`

基本思想：首先找到中序遍历下的第一个结点（从根结点出发，沿着左指针不断往左下走，直到左指针为空，到达“最左下”的结点即可），访问它后，然后不断寻找结点在中序下的后继结点并输出，直至所有的结点均被输出为止。

`inlhrtree insuccnode(inlhrtree p)`

```
{ /*寻找结点p在中序遍历下的后继结点*/
    inlhrtree q;
    if (p->rtag==1) return p->rchild;
    else { q=p->rchild;
          while (q->ltag==0) q=q->lchild;
          return q;
        }
}
```

```
void inthrtree(binthrtree p)
{ /*中序遍历中序穿线二叉树p*/
  if (p)
  { /*求二叉树p中序遍历下的第一个结点*/
    while (p->ltag==0)
      p=p->lchild;
    do
    { printf("%c ",p->data);
      p=insuccnode(p); }
    while (p);
  }
}
```

第7章 二叉树

- 二叉树的基本概念

- 二叉树的基本运算

- 二叉树的存储结构

- 二叉树的遍历

- 二叉树其它运算的实现

- 穿线二叉树

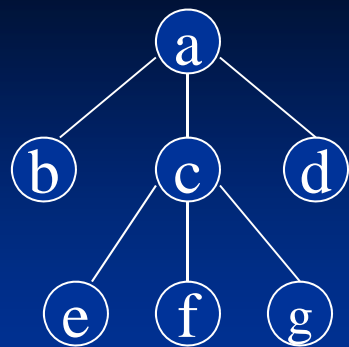
- 树、森林和二叉树的转换

7.7 树、森林和二叉树的转换

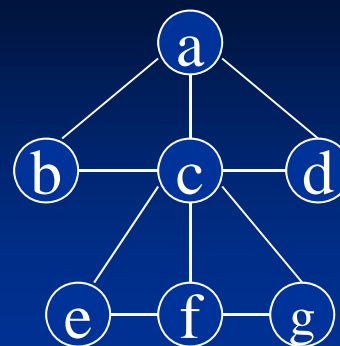
7.7.1 树、森林到二叉树的转换

将树或森林转换成其对应二叉树的方法为：

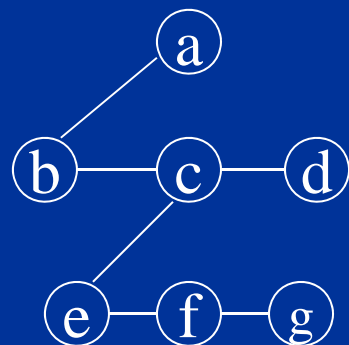
- (1) 在所有兄弟结点之间添加一条连线，如果是森林，则在其所有树的树根之间同样也添加一条连线；
- (2) 对于树、森林中的每个结点，除保留其到第一个子女的连线外，撤消其到其它子女的连线；
- (3) 将以上得到的树按照顺时针方向旋转45度。



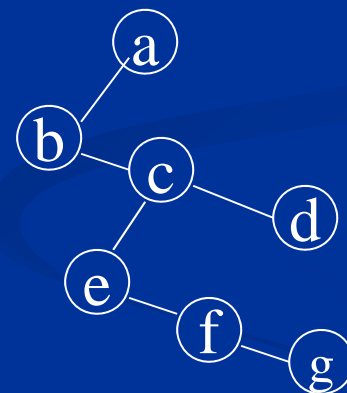
(a)



(b)

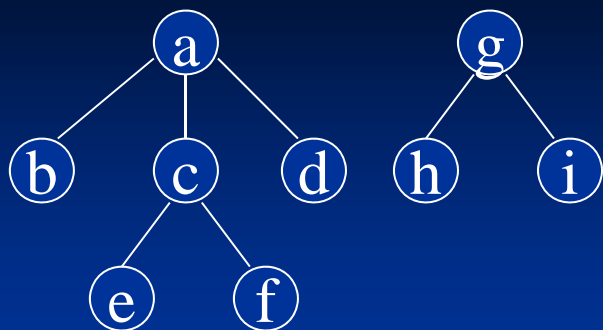


(c)

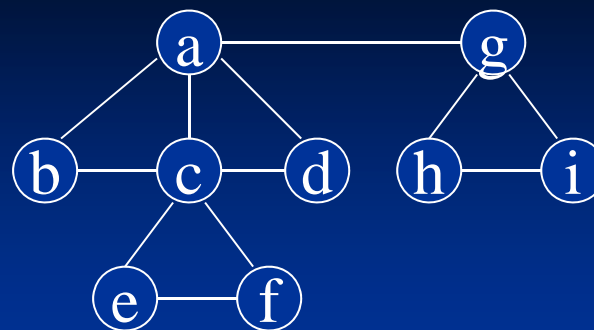


(d)

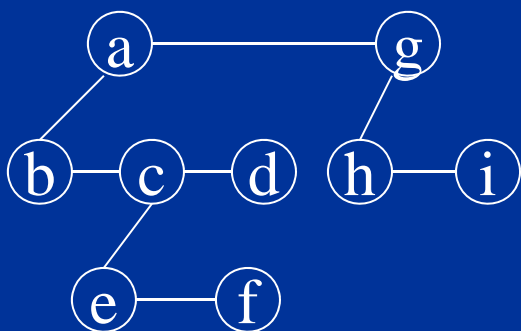
树到二叉树的转换



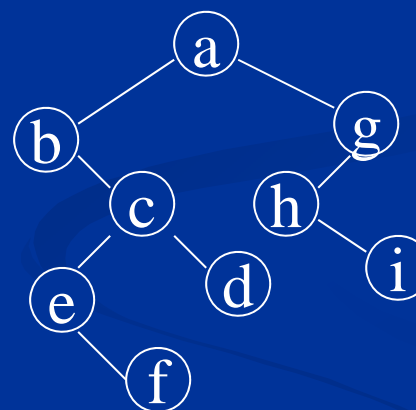
(a)



(b)



(c)



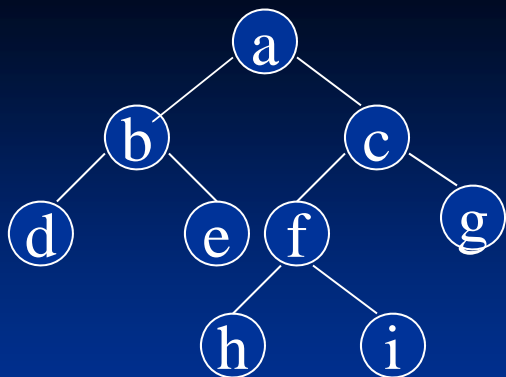
(d)

森林到二叉树的转换

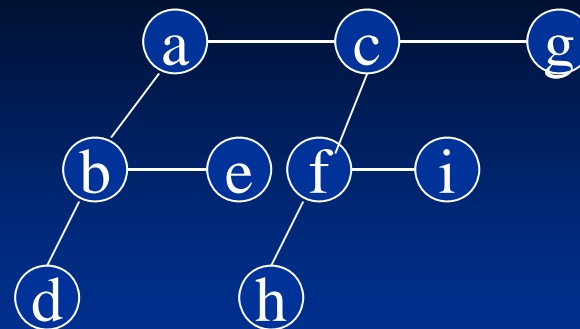
7.7.2 二叉树到树、森林的转换

二叉树到树、森林也有一种对应的转换关系，其过程恰巧为上述过程的逆过程，具体方法如下：

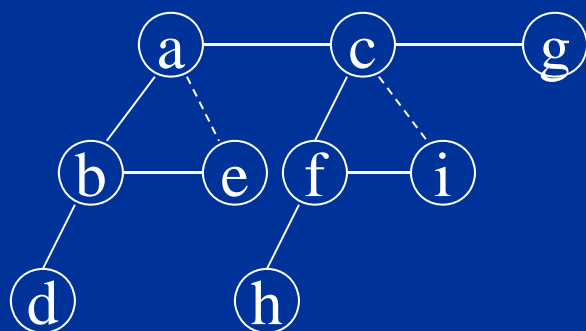
- (1) 首先将二叉树按照逆时针方向旋转45度；
- (2) 若某结点是其双亲的左子女，则把该结点的右子女，右子女的右子女，……都与该结点的双亲用线连起来；
- (3) 最后去掉原二叉树中所有双亲到其右子女的连线。



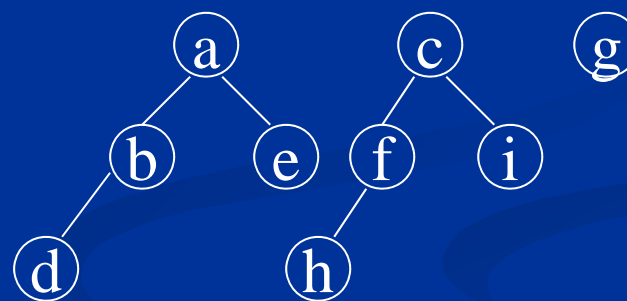
(a)



(b)



(c)



(d)

二叉树到树、森林的转换

第7章 二叉树

- 二叉树的基本概念
 - 二叉树的基本运算
 - 二叉树的存储结构
 - 二叉树的遍历
 - 二叉树其它运算的实现
 - 穿线二叉树
- 树、森林和二叉树的转换
- 作业：1, 3, 4, 5, 7, 8, 9, 13, 14, 15

第 8 章 图

➤图的基本概念

➤ 图的基本运算

➤ 图的基本存储结构

➤ 图的遍历

➤生成树与最小生成树

➤最短路径

➤拓扑排序

➤关键路径

8.1 图的基本概念

一、图的定义

图是由一个非空的顶点集合和一个描述顶点之间多对多关系的边（或弧）集合组成的一种数据结构，它可以形式化地表示为：

$$\text{图} = (V, E)$$

其中 $V = \{x | x \in \text{某个数据对象集}\}$ ，它是顶点的有穷非空集合； $E = \{(x, y) | x, y \in V\}$ 或 $E = \{<x, y> | x, y \in V \text{ 且 } P(x, y)\}$ ，它是顶点之间关系的有穷集合，也叫做边集合， $P(x, y)$ 表示从 x 到 y 的一条单向通路。

图的应用举例

例1 交通图（公路、铁路）

顶点：地点

边：连接地点的公路

交通图中的有单行道双行道，分别用有向边、无向边表示；

例2 电路图

顶点：元件

边：连接元件之间的线路

例3 通讯线路图

顶点：地点

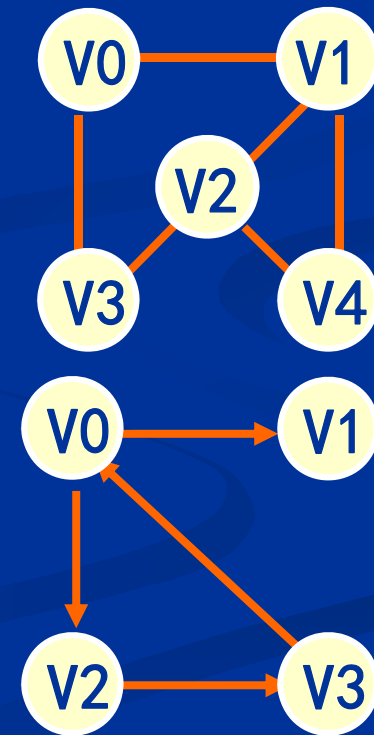
边：地点间的连线

例4 各种流程图

如产品的生产流程图

顶点：工序

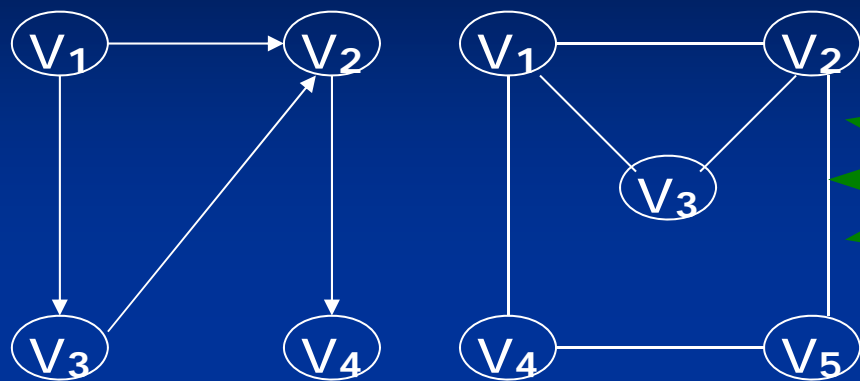
边：各道工序之间的顺序关系



通常，也将图 G 的顶点集和边集分别记为 $V(G)$ 和 $E(G)$ 。 $E(G)$ 可以是空集，若 $E(G)$ 为空，则图 G 只有顶点而没有边。

若图 G 中的每条边都是有方向的，则称 G 为有向图。在有向图中，一条有向边是由两个顶点组成的有序对，有序对通常用尖括号表示。例如，有序对 $\langle v_i, v_j \rangle$ 表示一条由 v_i 到 v_j 的有向边。有向边又称为弧，弧的始点称为弧尾，弧的终点称为弧头。若图 G 中的每条边都是没有方向的，则称 G 为无向图。无向图中的边均是顶点的无序对，无序对通常用圆括号表示。

例 图8-1



(a) 有向图 G_1 (b) 无向图 G_2

有序对 $\langle v_i, v_j \rangle$:

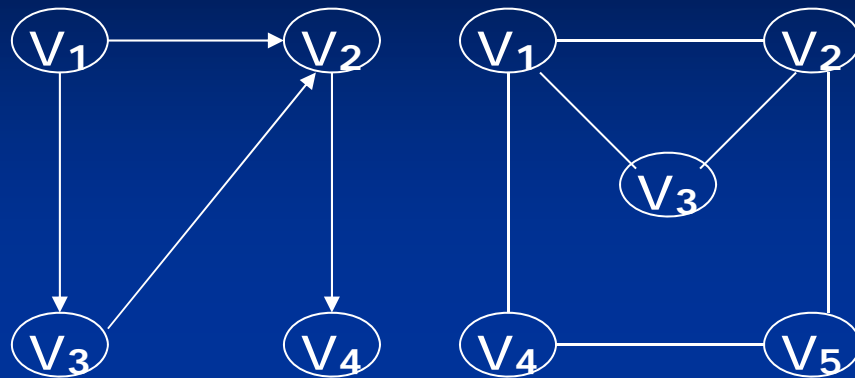
用以为 v_i 起点、以 v_j 为终点的有向线段表示, 称为有向边或弧;

图8.1 (a) 表示的是有向图 G_1 , 该图的顶点集和边集分别为:

$$V(G_1) = \{v_1, v_2, v_3, v_4\}$$

$$E(G_1) = \{\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_2, v_4 \rangle, \langle v_3, v_2 \rangle\}$$

例：图8-1



(a) 有向图 G_1 (b) 无向图 G_2

无序对 (v_i, v_j) ：
用连接顶点 v_i 、 v_j 的线段
表示，称为无向边；

图8.1 (b) 表示的是无向图 G_2 ，该图的顶点集和边集分别为：

$$V(G_2) = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E(G_2) = \{ (v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_5), (v_4, v_5) \}$$

在以后的讨论中，我们约定：

(1) 一条边中涉及的两个顶点必须不相同，即：若 (v_i, v_j) 或 $\langle v_i, v_j \rangle$ 是 $E(G)$ 中的一条边，则要求 $v_i \neq v_j$ ；

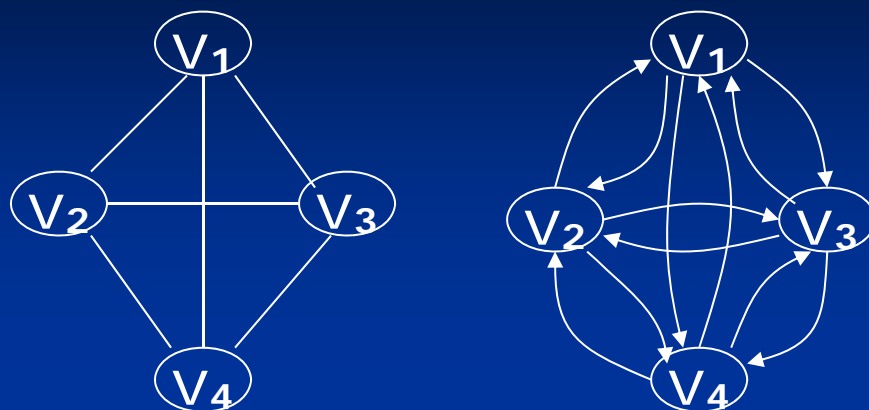
(2) 一对顶点间不能有相同方向的两条有向边；

(3) 一对顶点间不能有两条无向边，即只讨论简单的图。

二、完全图

若用 n 表示图中顶点的数目，用 e 表示图中边的数目，按照上述规定，容易得到下述结论：对于一个具有 n 个顶点的无向图，其边数 e 小于等于 $n(n-1)/2$ ，边数恰好等于 $n(n-1)/2$ 的无向图称为**无向完全图**；对于一个具有 n 个顶点的有向图，其边数 e 小于等于 $n(n-1)$ ，边数恰好等于 $n(n-1)$ 的有向图称为**有向完全图**。也就是说完全图具有最多的边数，任意一对顶点间均有边相连。

例：图8-2



(a) 无向完全图 G_3 (b) 有向完全图 G_4

图8.2所示的 G_3 与 G_4 分别是具有4个顶点的无向完全图和有向完全图。图 G_3 共有4个顶点6条边；图 G_4 共有4个顶点12条边。

若 (v_i, v_j) 是一条无向边，则称顶点 v_i 和 v_j 互为邻接点。

若 $\langle v_i, v_j \rangle$ 是一条有向边，则称 v_i 邻接到 v_j ， v_j 邻接于 v_i ，并称有向边 $\langle v_i, v_j \rangle$ 关联于 v_i 与 v_j ，或称有向边 $\langle v_i, v_j \rangle$ 与顶点 v_i 和 v_j 相关联。

三、度、入度、出度

在图中，一个顶点的度就是与该顶点相关联的边的数目，顶点 v 的度记为 $D(v)$ 。例如在图8.2

(a)所示的无向图 G_3 中，各顶点的度均为3。

若 G 为有向图，则把以顶点 v 为终点的边的数目称为顶点 v 的入度，记为 $ID(v)$ ；把以顶点 v 为始点的边的数目称为 v 的出度，记为 $OD(v)$ ，有向图中顶点的度数等于顶点的入度与出度之和，即 $D(v) = ID(v) + OD(v)$ 。

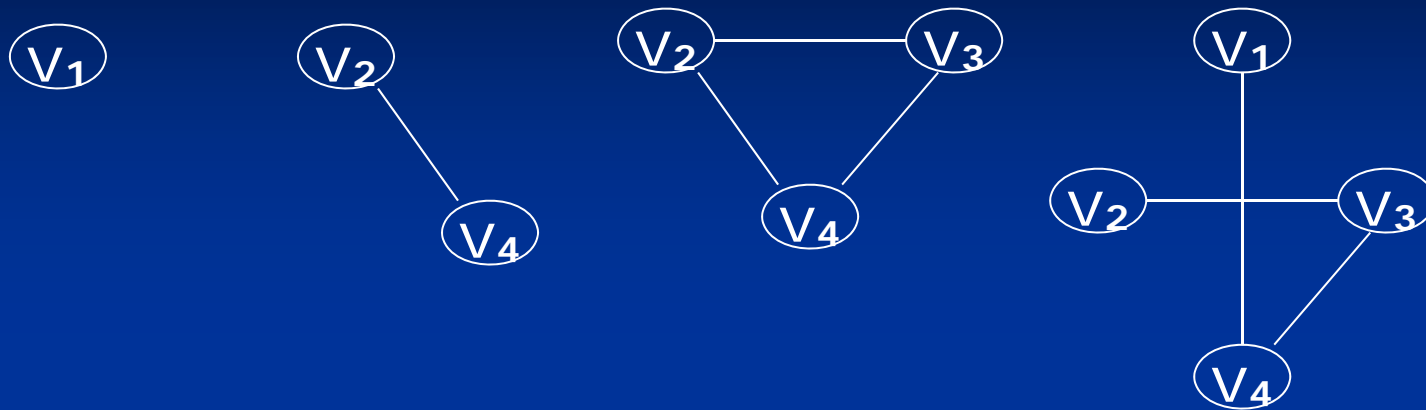
无论有向图还是无向图，图中的每条边均关联于两个顶点，因此，顶点数 n 、边数 e 和度数之间有如下关系：

$$e = \frac{1}{2} \sum_{i=1}^n D(v_i) \quad \dots\dots\dots (\text{式8-1})$$

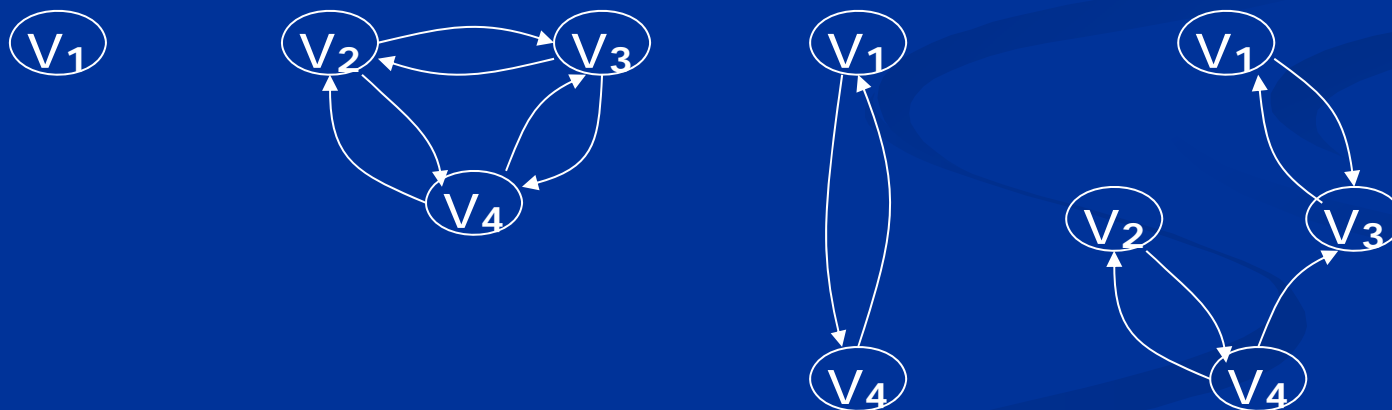
四、子图

给定两个图 G_i 和 G_j ，其中 $G_i = (V_i, E_i)$ ， $G_j = (V_j, E_j)$ ，若满足 $V_i \subseteq V_j$ ， $E_i \subseteq E_j$ ，则称 G_i 是 G_j 的子图。

子图示例



(a) 无向图 G_3 的部分子图



(b) 有向图 G_4 的部分子图

五、路径

无向图 G 中若存在着一个顶点序列 v 、 v_1' 、 v_2' 、...、 v_m' 、 u ，且 (v, v_1') 、 (v_1', v_2') 、...、 (v_m', u) 均属于 $E(G)$ ，则称该顶点序列为顶点 v 到顶点 u 的一条路径，相应地，顶点序列 u 、 v_m' 、 v_{m-1}' 、...、 v_1' 、 v 是顶点 u 到顶点 v 的一条路径。

如果 G 是有向图，路径也是有向的，它由 $E(G)$ 中的有向边 $\langle v, v_1' \rangle$ 、 $\langle v_1', v_2' \rangle$ 、...、 $\langle v_m', u \rangle$ 组成。路径长度是该路径上边或弧的数目。

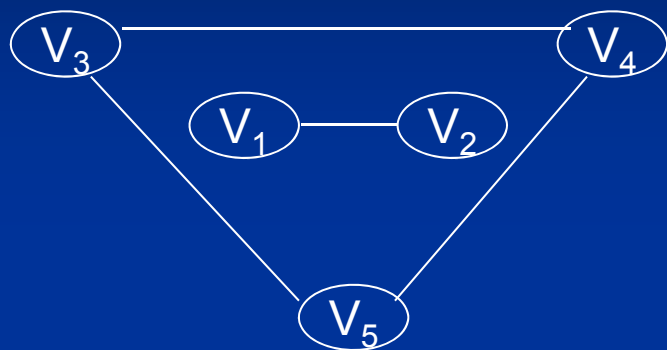
如果一条路径上除了起点 v 和终点 u 相同外，其余顶点均不相同，则称此路径为一条简单路径。起点和终点相同（ $v=u$ ）的简单路径称为简单回路或简单环。

六、连通图与强连通图

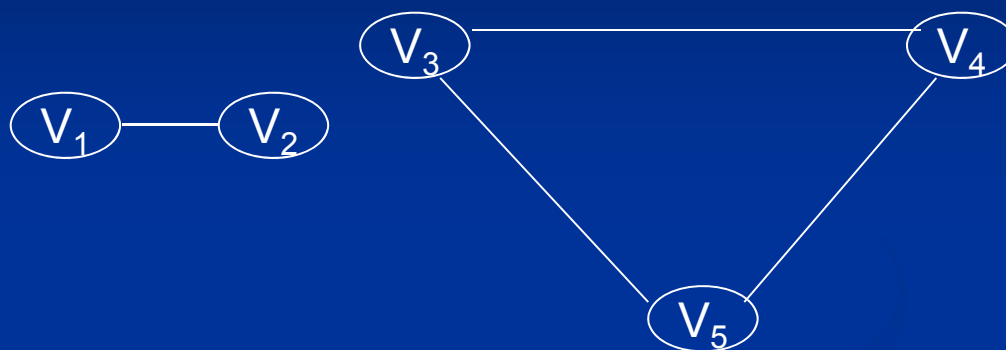
在无向图 G 中，若从顶点 v_i 到顶点 v_j 有路径，则称 v_i 与 v_j 是连通的。若 $V(G)$ 中任意两个不同的顶点 v_i 和 v_j 都连通（即有路径），则称 G 为连通图。例如，图8.1（b）所示的无向图 G_2 、图8.2（a）所示的无向图 G_3 是都是连通图。

无向图 G 的极大连通子图称为 G 的连通分量。根据连通分量的定义，可知任何连通图的连通分量是其自身，非连通的无向图有多个连通分量。

例：非连通图及其连通分量示例



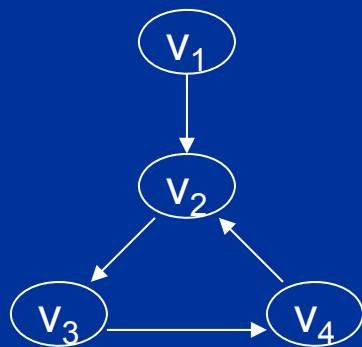
(a) 非连通图 G_5



(b) G_5 的两个连通分量 H_1 和 H_2

在有向图 G 中，若对于 $V(G)$ 中任意两个不同的顶点 v_i 和 v_j ，都存在从 v_i 到 v_j 以及从 v_j 到 v_i 的路径，则称 G 是**强连通图**。

有向图的极大强连通子图称为 G 的强连通分量。根据强连通图的定义，可知强连通图的唯一强连通分量是其自身，而非强连通的有向图有多个强连通分量。例如，图8.2 (b) 所示的有向图 G_4 是一个具有4个顶点的强连通图，图8.5 (a) 所示的有向图 G_6 不是强连通图 (v_2 、 v_3 、 v_4 没有到达 v_1 的路径)，它的两个强连通分量 H_3 与 H_4 如图8.5 (b) 所示。



(a) 非强连通图 G_6

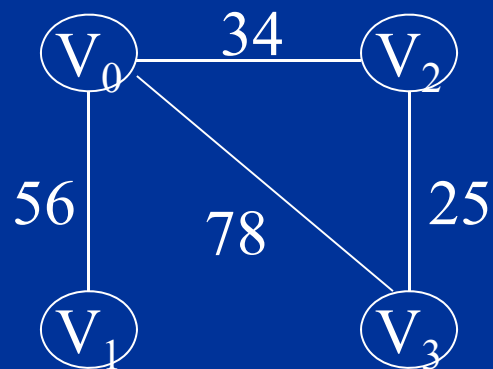


(b) G_6 的两个强连通分量 H_3 和 H_4

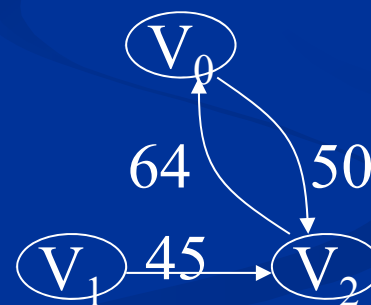
七、网络

有时在图的每条边上附上相关的数值，这种与图的边相关的数值叫**权**。

权可以表示两个顶点之间的距离、耗费等具有某种意义的数。若将图的每条边都赋上一个权，则称这种带权图为网络。



(a) 无向网络 G_7



(b) 有向网络 G_8

作业:

8.1 对于无向图8.29, 试给出

- (1) 图中每个顶点的度;
- (2) 该图的邻接矩阵;
- (4) 该图的连通分量。

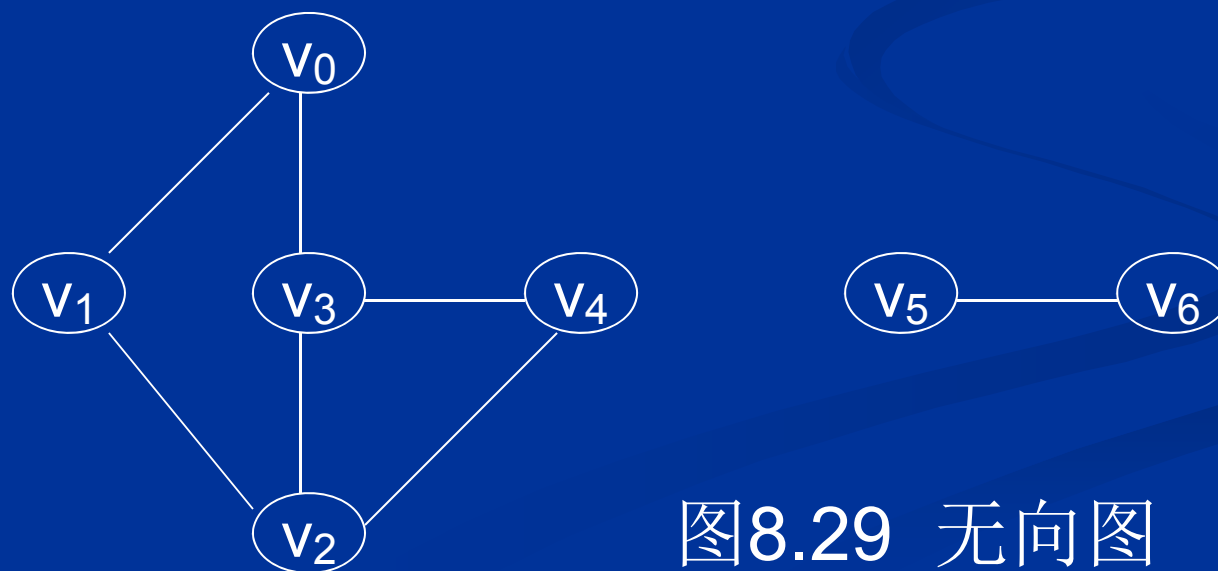


图8.29 无向图

8.2 给定有向图8.30，试给出

- (1) 顶点D的入度与出度；
- (2) 该图的出边表与入边表；
- (3) 该图的强连通分量。

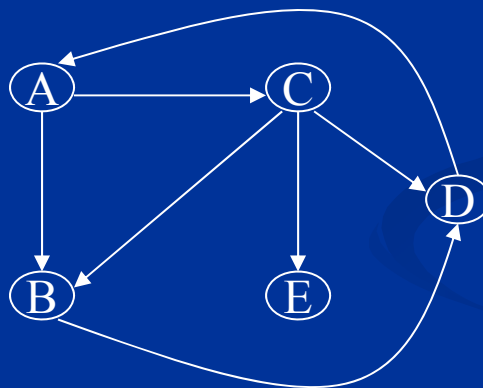


图8.30 有向图

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
 - 最短路径
 - 拓扑排序
 - 关键路径

8.2 图的基本运算

图是一种复杂数据结构，由图的定义及图的一组基本操作构成了图的抽象数据类型。

ADT Graph{

数据对象V: V是具有相同特性的数据元素的集合，称为顶点集。

数据关系R:

$R=\{<v, w>|v, w \in V \text{ 且 } P(v, w), P(v, w) \text{ 定义了边 (或弧) } (v, w) \text{ 的信息}\}$

图的基本操作如下：

- (1) `creatgraph (&g)` 创建一个图的存储结构。
- (2) `insertvertex (&g, v)` 在图g中增加一个顶点v。
- (3) `deletevertex (&g, v)` 在图g中删除顶点v及所有和顶点v相关联的边或弧。
- (4) `insertedge (&g, v, u)` 在图g中增加一条从顶点v到顶点u的边或弧。
- (5) `deleteedge (&g, v, u)` 在图g中删除一条从顶点v到顶点u的边或弧。

(6) `trave (g)` 遍历图g。

(7) `locatevertex (g, v)` 求顶点v在图g中的位序。

(8) `fiirstvertex (g, v)` 求图g中顶点v的第一个邻接点。

(9) `degree (g, v)` 求图g中顶点v的度数。

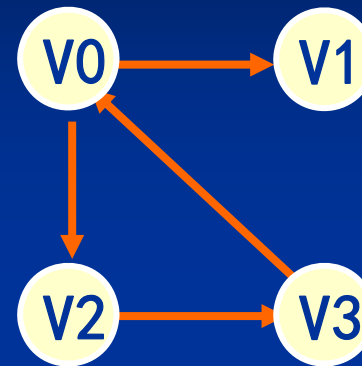
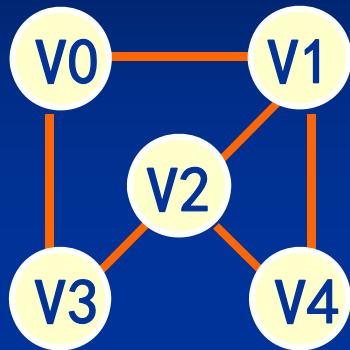
(10) `nextvertex (g, v, w)` 求图g中与顶点v相邻接的顶点w的下一个邻接点。即求图g中顶点v的某个邻接点，它的存储顺序排在邻接点w的存储位置之后。

} ADT Graph

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
- 最短路径
- 拓扑排序
- 关键路径

8.3图的基本存储结构



图的存储结构至少要保存两类信息

- 1) 顶点的数据
- 2) 顶点间的关系

如何表示顶点间的关系?

约定:

$G = \langle V, E \rangle$ 是图, $V = \{v_0, v_1, v_2, \dots, v_{n-1}\}$, 设顶点的角标为它的编号

8.3.1 邻接矩阵及其实现

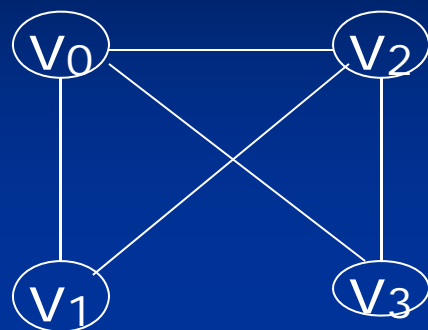
一、非网络的邻接矩阵

给定图 $G = (V, E)$ ，其中 $V(G) = \{v_0, \dots, v_i, \dots, v_{n-1}\}$ ， G 的邻接矩阵（Adjacency Matrix）是具有如下性质的 n 阶方阵：

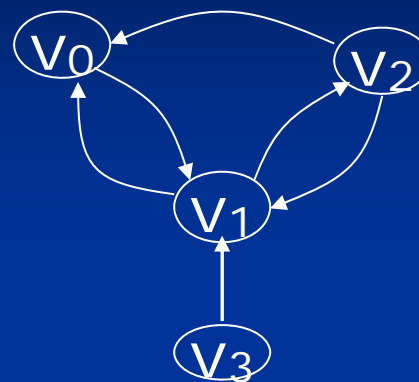
$$A[i, j] = \begin{cases} 1, & \text{如果 } \langle i, j \rangle \in E \text{ 或者 } (i, j) \in E \\ 0, & \text{否则} \end{cases}$$

无向图的邻接矩阵是对称的，有向图的邻接矩阵可能是不对称的。

图的邻接矩阵示例



$$A1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



$$A2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

图8.7 无向图 G_9 及有向图 G_{10} 的邻接矩阵表示

用邻接矩阵表示图，很容易判定任意两个顶点之间是否有边相连，并求得各个顶点的度数。对于无向图，顶点 v_i 的度数是邻接矩阵中第 i 行或第 i 列值为1的元素个数，即：

$$D(v_i) = \sum_{j=0}^{n-1} A[i, j] = \sum_{j=0}^{n-1} A[j, i] \quad \dots (8-2)$$

对于有向图，邻接矩阵中第 i 行值为1的元素个数为顶点 v_i 的出度，第 i 列值为1的元素的个数为顶点 v_i 的入度，即：

$$OD(v_i) = \sum_{j=0}^{n-1} A[i, j] \quad ; \quad ID(v_i) = \sum_{j=0}^{n-1} A[j, i] \quad \dots (8-3)$$

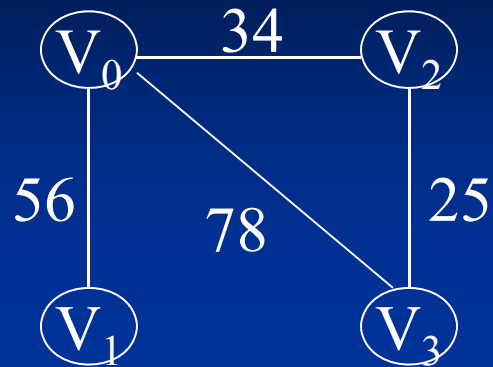
二、网络的邻接矩阵

当 $G=(V, E)$ 是一个网络时， G 的邻接矩阵是具有如下性质的 n 阶方阵：

$$A[i, j] = \begin{cases} W_{ij} & \text{当 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \\ 0 & \text{当 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \text{ 且 } i=j \\ \infty & \text{当 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \notin E(G) \text{ 且 } i \neq j \end{cases}$$

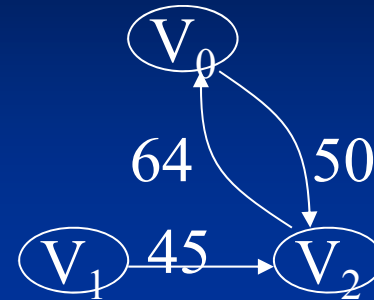
其中 W_{ij} 表示边上的权值； ∞ 表示一个计算机允许的、大于所有边上权值的数。

网络的邻接矩阵示例



$$A_3 = \begin{bmatrix} 0 & 56 & 34 & 78 \\ 56 & 0 & \infty & \infty \\ 34 & \infty & 0 & 25 \\ 78 & \infty & 25 & 0 \end{bmatrix}$$

(a) G_7 的邻接矩阵



$$A_4 = \begin{bmatrix} 0 & \infty & 50 \\ 0 & \infty & 45 \\ 64 & \infty & 0 \end{bmatrix}$$

(b) G_8 的邻接矩阵

图8.8 网络邻接矩阵示例

邻接矩阵存储结构

文件名:mgraph.h

```
#define FINITY 5000      /*此处用5000代表无穷大*/  
#define m 20            /*最大顶点数*/  
typedef char vertextype; /*顶点值类型*/  
typedef int edgetype;    /*权值类型*/  
typedef struct{  
    vertextype vexs[m];    /*顶点信息域*/  
    edgetype edges[m][m]; /*邻接矩阵*/  
    int n,e;               /*图中顶点总数与边数*/  
} mgraph;                /*邻接矩阵表示的图类型*/
```



```
/* **** */
/*      图的邻接矩阵创建算法      */
/*      文件名: c_ljz.c  函数名: creatmgraph1()  */
/* **** */

#include <stdio.h>
#include "mgraph.h"
void creatmgraph1(mgraph *g)
{int i,j,k,w; /*建立有向网络的邻接矩阵存储结构*/
    printf("please input n and e:\n");
    scanf("%d%d",&g->n,&g->e); /*输入图的顶点数与边数*/
    getchar();                /*取消输入的换行符*/
    printf("please input vexs:\n");
```

```
for(i=0;i<g->n;i++)           /*输入图中的顶点值*/
    g->vexs[i]=getchar();
for(i=0;i<g->n;i++)           /*初始化邻接矩阵*/
    for(j=0;j<g->n;j++)
        if (i==j) g->edges[i][j]=0;
        else g->edges[i][j]=FINITY;
printf("please input edges:\n");
for (k=0;k<g->e;k++)          /*输入网络中的边*/
    { scanf("%d%d%d", &i,&j,&w);
      g->edges[i][j]=w;
/*若是建立无向网，只需在此加入语句g->edges[j][i]=w;即可*/
    } }
```

说明:

- 当建立有向网时，边信息以三元组 (i, j, w) 的形式输入， i 、 j 分别表示两顶点的序号， w 表示边上的权。对于每一条输入的边信息 (i, j, w) ，只需将 `g->edges[i][j]`赋值为 w 。
- 算法8.5中用到的`creatmgraph2()`是用于建立无向网络的函数，它与`creatmgraph1()`的区别在于对每一条输入的边信息 (i, j, w) ，需同时将`g->edges[i][j]` 和 `g->edges[j][i]`赋值为 w 。
- 当建立非网络的存储结构时，所有的边信息只需按二元组 (i, j) 的形式输入。

8.3.2邻接表及其实现

用邻接矩阵表示法存储图，占用的存储单元个数只与图中顶点的个数有关，而与边的数目无关。一个含有 n 个顶点的图，如果其边数比 n^2 少得多，那么它的邻接矩阵就会有太多空元素，浪费了存储空间。

■ 无向图的邻接表

对于图 G 中的每个顶点 v_i ，该方法把所有邻接于 v_i 的顶点 v_j 链成一个带头结点的单链表，这个单链表就称为顶点 v_i 的邻接表。单链表中的每个结点至少包含两个域，一个为邻接点域（adjvex），它指示与顶点 v_i 邻接的顶点在图中的位序，另一个为链域（next），它指示与顶点 v_i 邻接的下一个结点。



为了便于随机访问任一顶点的邻接表，可将所
 表头结点结构 边结点结构 构成了图的邻接表
 等信息与邻接表

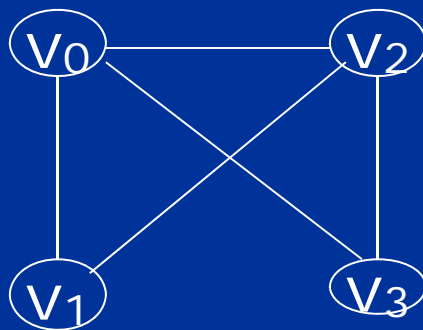


图8.7 无向图 G_9

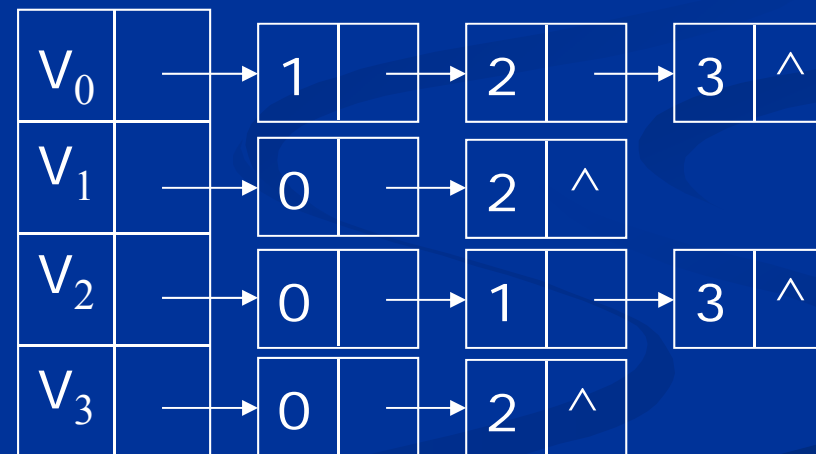


图8.9 G_9 的邻接表

对于无向图， v_i 的邻接表中每个表结点都对应于与 v_i 相关联的一条边；对于有向图来说，如果每一顶点 v_i 的邻接表中每个表结点都存储以 v_i 的为始点射出的一条边，则称这种表为有向图的**出边表**（有向图的邻接表），反之，若每一顶点 v_i 的邻接表中每个表结点都对应于以 v_i 为终点的边（即射入 v_i 的边），则称这种表为有向图的**入边表**（又称逆邻接表）。

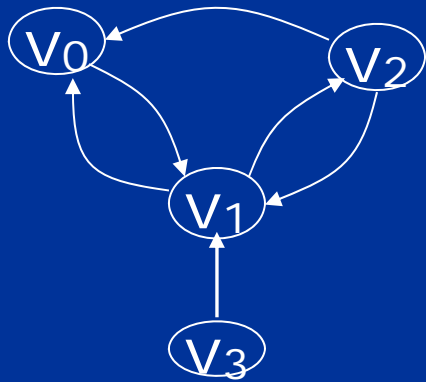
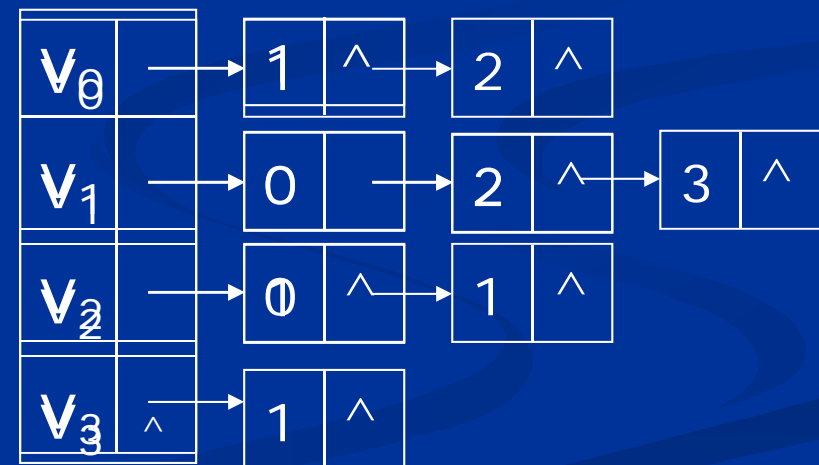


图8.7(b)有向图 G_{10}



G_{10} 的入边表

在无向图的邻接表中，顶点 v_i 的度为第 i 个链表中结点的个数；而在有向图的出边表中，第 i 个链表中的结点个数是顶点 v_i 的出度；为了求入度，必须对整个邻接表扫描一遍，所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。



G_{10} 的出边表

邻接表的存储结构

```
/******
```

```
/* 邻接表存储结构 文件名: adjgraph.h */
```

```
*****
```

```
#define m 20 /*预定义图的最大顶点数*/
```

```
typedef char datatype; /*顶点信息数据类型*/
```

```
typedef struct node{ /*边表结点*/
```

```
    int adjvex; /*邻接点*/
```

```
    struct node *next;
```

```
}edgenode;
```

边结点结构

adjvex

next


```
typedef struct vnode{           /*头结点类型*/
    datatype vertex;           /*顶点信息*/
    edgenode *firstedge;       /*邻接链表头指针*/
}vertexnode;

typedef struct{                 /*邻接表类型*/
    vertexnode adjlist [m];     /*存放头结点的顺序表*/
    int n,e;                    /*图的顶点数与边数*/
}adjgraph;
```

头结点结构



```
/******
```

```
/*          无向图的邻接表创建算法          */
```

```
/*    文件名c_ljb.c 函数名: createadjgraph()    */
```

```
/******
```

```
void createadjgraph(adjgraph *g)
```

```
{ int i,j,k;
```

```
    edgenode *s;
```

```
    printf("Please input n and e:\n");
```

```
    scanf("%d%d",&g->n,&g->e);    /*输入顶点数与边数*/
```

```
    getchar();
```

```
    printf("Please input %d vertex:",g->n);
```

```
for(i=0;i<g->n;i++)  
    {scanf("%c",&g->adjlist[i].vertex);    /*读入顶点信息*/  
      g->adjlist[i].firstedge=NULL;        /*边表置为空表*/  
    }  
printf("Please input %d edges:",g->e);  
for(k=0;k<g->e;k++)                        /*循环e次建立边表*/  
    { scanf("%d%d",&i,&j);                /*输入无序对 (i,j) */  
      s=(edgenode *)malloc(sizeof(edgenode));  
      s->adjvex=j;                        /*邻接点序号为j*/  
      s->next=g->adjlist[i].firstedge;  
      g->adjlist[i].firstedge=s;    /*将新结点*s插入顶点vi  
的边表头部*/
```

```
s=(edgenode *)malloc(sizeof(edgenode));  
s->adjvex=i;           /*邻接点序号为i*/  
s->next=g->adjlist[j].firstedge;  
g->adjlist[j].firstedge=s;  
/*将新结点*s插入顶点vj的边表头部*/  
}  
}
```

算法8.2 建立无向图的邻接表算法

说明：一个图的邻接矩阵表示是唯一的，但其邻接表表示不唯一，这是因为在邻接表结构中，各边表结点的链接次序取决于建立邻接表的算法以及边的输入次序。

例: 若需建立下图所示的无向图邻接表存储结构, 则在执行程序c_ljb.c时如果输入的信息为:

4 5

ABCD

0 1 0 2 0 3 1 2 2 3

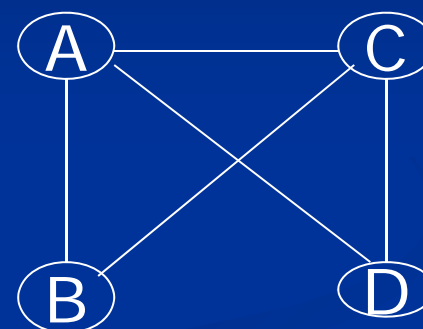
则将建立如下的邻接表存储结构。

A 3-->2-->1

B 2-->0

C 3-->1-->0

D 2-->0



8.3.3邻接多重表

■在邻接多重表中，每一条边只有一个边结点。为有关边的处理提供了方便。

—边结点的结构

<i>mark</i>	<i>vex_i</i>	<i>link_i</i>	<i>vex_j</i>	<i>link_j</i>
-------------	------------------------	-------------------------	------------------------	-------------------------

其中，*mark* 是记录是否处理过的标记；*vex_i*和*vex_j*是依附于该边的两顶点位置。*link_i*域是链接指针，指向下一条依附于顶点*vex_i*的边；*link_j*也是链接指针，指向下一条依附于顶点*vex_j*的边。需要时还可设置一个存放与该边相关的权值的域 *cost*。

— 顶点结点的结构

vertex

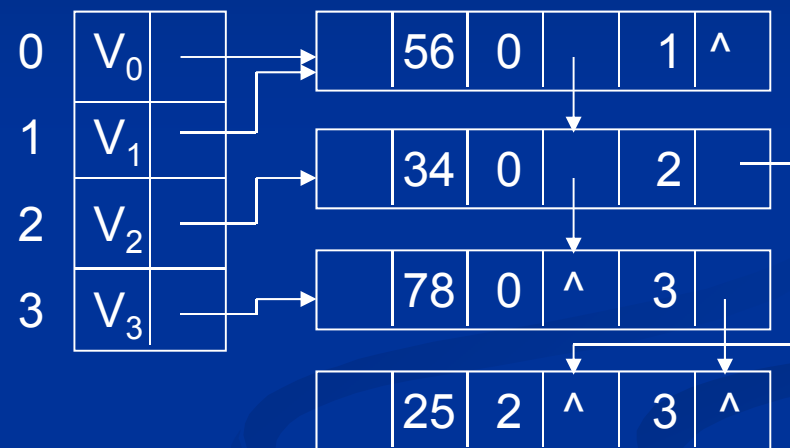
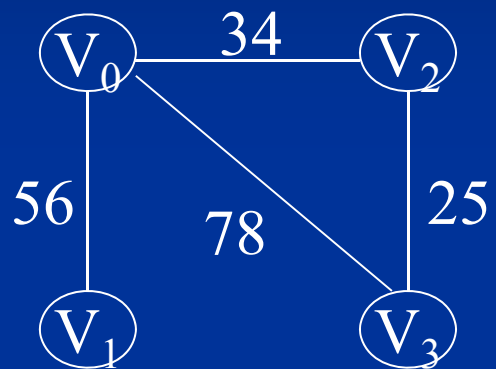
Firstedge

存储顶点信息的结点表以顺序表方式组织，每一个顶点结点有两个数据成员：其中，**vertex** 存放与该顶点相关的信息，**firstedge** 是指示第一条依附于该顶点的边的指针。

在邻接多重表中，所有依附于同一个顶点的边都链接在同一个单链表中。

从顶点 i 出发，可以循链找到所有依附于该顶点的边，也可以找到它的所有邻接顶点。

无向网络的邻接多重表示例



其中边表结点增加了一个存储权值的数据域。

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
- 最短路径
- 拓扑排序
- 关键路径

8.4 图的遍历

图的遍历：从图的某顶点出发，访问图中所有顶点，并且每个顶点仅访问一次。在图中，访问部分顶点后，可能又沿着其他边回到已访问过的顶点，为保证每一个顶点只被访问一次，图的遍历与树的遍历有什么不同？

一般用一个辅助数组 $visit[n]$ 作为对顶点的标记，当顶点 v_i 未被访问， $visit[i]$ 值为0；当 v_i 已被访问，则 $visit[i]$ 值为1。

有两种遍历方法（它们对无向图，有向图都适用）

- 深度优先遍历
- 广度优先遍历

8.4.1 深度优先遍历

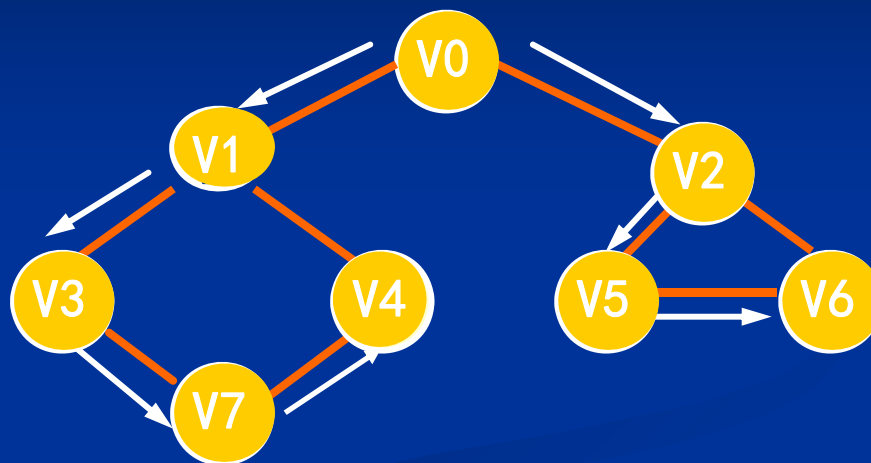
从图对某连通图 $G=(V, E)$ ，首先将 V 中每一个顶点都标记为未被访问，然后，选取一个源点 $v \in V$ ，将 v 标记为已被访问，再递归地用深度优先搜索方法，依次搜索 v 的所有邻接点 w 。若 w 未曾访问过，则以 w 为新的出发点继续进行深度优先遍历，如果从 v 出发有路的顶点都已被访问过，则从 v 的搜索过程结束。此时，如果图中还有未被访问过的顶点（该图有多个连通分量或强连通分量），则再任选一个未被访问过的顶点，并从这个顶点开始做新的搜索。上述过程一直进行到 V 中所有顶点都已被访问过为止。

例

深度优先遍历过程:

序列1:

V0,V1,V3,V7,V4,V2,V5,V6

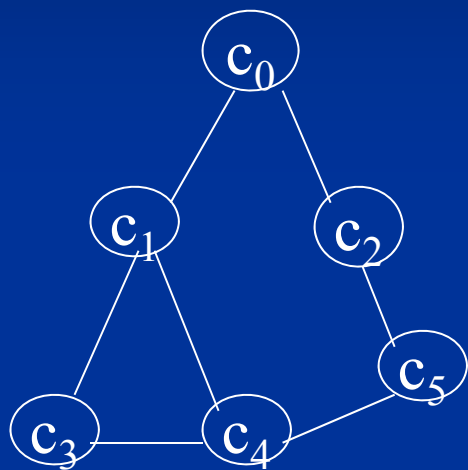


由于没有规定
访问邻接点的顺序,
深度优先序列不是唯一的

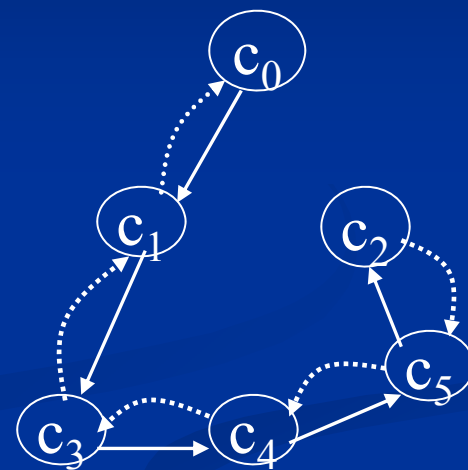
序列2:

V0,V1,V4,V7,V3,V2,V5,V6

但是，当采用邻接表存储结构并且存储结构已确定的情况下，遍历的结果将是确定的。



0	C0	→	1	→	2	^
1	C1	→	0	→	3	→ 4 ^
2	C2	→	0	→	5	^
3	C3	→	1	→	4	^
4	C4	→	1	→	3	→ 5 ^
5	C5	→	2	→	4	^



DFS序列: $c_0 \rightarrow c_1 \rightarrow c_3 \rightarrow c_4 \rightarrow c_5 \rightarrow c_2$

采用邻接表存储结构的深度优先遍历算法实现：

```
/*  
    图的深度优先遍历算法  
    文件名: dfs.c  函数名: dfs()、dfstraverse()  
*/  
/*  
*****/  
  
int visited[m];  
  
void dfs(adjgraph g,int i)  
{ /*以 $v_i$ 为出发点深度优先遍历顶点 $v_i$ 所在的连通分量*/  
    edgenode *p;  
    printf("visit vertex: %c \n",g.adjlist[i].vertex); /*访问顶点i*/  
    visited[i]=1;
```

```
p=g.adjlist[i].firstedge;
while (p)          /*从p的邻接点出发进行深度优先搜索*/
{ if (!visited[p->adjvex])
    dfs(g,p->adjvex);  /*递归*/
  p=p->next;
}
}
```

```
void dfstraverse(adjgraph g)
{ /* 深度优先遍历图g */
  int i;
  for (i=0;i<g.n;i++)
    visited[i]=0;      /*初始化标志数组*/
  for (i=0;i<g.n;i++)
    if (!visited[i])   /* $v_i$ 未访问过*/
      dfs(g,i);
}
```

算法8.3 图的深度优先遍历算法（邻接表表示法）

算法分析:

对于具有 n 个顶点和 e 条边的无向图或有向图，遍历算法`dfstraverse`对图中每个顶点至多调用一次`dfs`。从`dfstraverse`中调用`dfs`或`dfs`内部递归调用自己的最大次数为 n 。当访问某顶点 v_i 时，`dfs`的时间主要耗费在从该顶点出发搜索它的所有邻接点上。用邻接表表示图时，需搜索第 i 个边表上的所有结点，因此，对所有 n 个顶点访问，在邻接表上需将边表中所有 $O(e)$ 个结点检查一遍。所以，`dfstraverse`算法的时间复杂度为 $O(n+e)$ 。

8.4.2 广度优先遍历

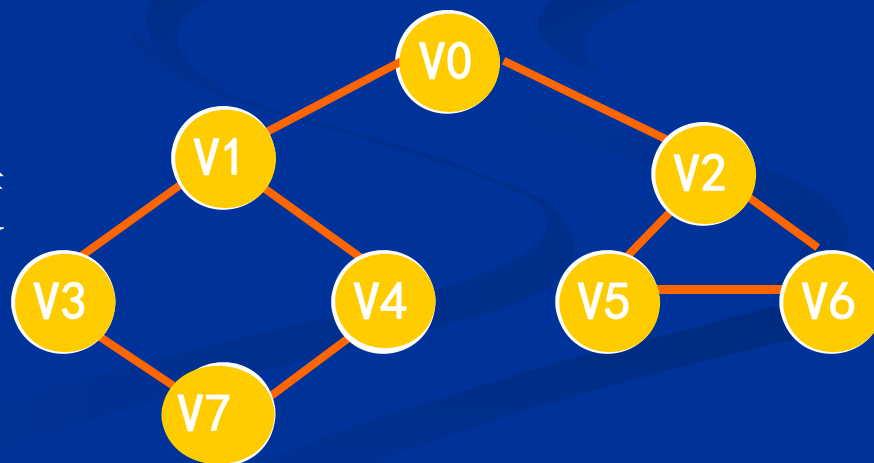
图中某未访问过的顶点 v_i 出发:

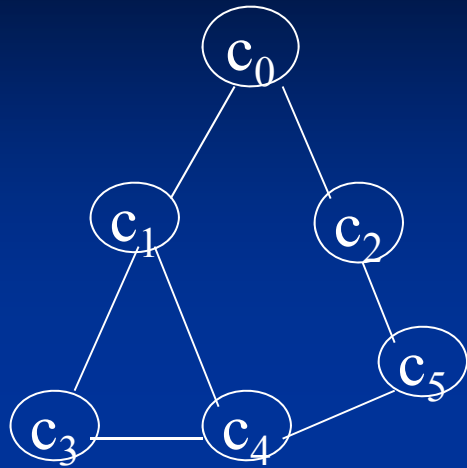
- 1) 访问顶点 v_i ;
- 2) 访问 v_i 的所有未被访问的邻接点 w_1, w_2, \dots, w_k ;
- 3) 依次从这些邻接点出发, 访问它们的所有未被访问的邻接点; 依此类推, 直到图中所有访问过的顶点的邻接点都被访问;

例

求图G 的以 V_0 起点的的广度
优先序列

$V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7$





从 C_0 出发的BFS序列为:

$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5$

由于没有规定
访问邻接点的顺序,
广度优先序列不是唯一的

广度优先算法:

从图中某顶点 v_i 出发:

- 1) 访问顶点 v_i ; (容易实现)
- 2) 访问 v_i 的所有未被访问的邻接点 w_1, w_2, \dots, w_k ;
- 3) 依次从这些邻接点(在步骤2)访问的顶点)出发, 访问它们的所有未被访问的邻接点; 依此类推, 直到图中所有访问过的顶点的邻接点都被访问;

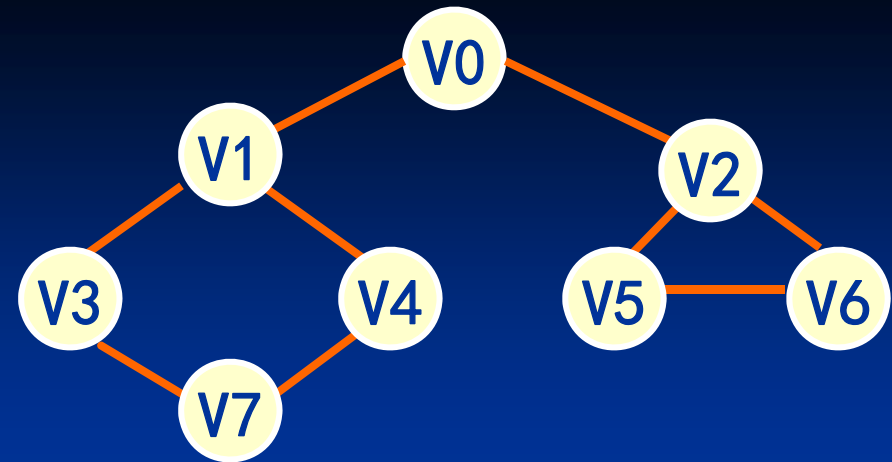
为实现3), 需要保存在步骤(2)中访问的顶点, 而且访问这些顶点邻接点。在广度优先遍历算法中, 其邻接点先被访问。

需设置一队列Q,
保存已访问的顶点,
并控制遍历顶点的顺序。

QUEUE



V0 V1 V2 V3 V4 V5 V6 V7



数据结构:

1) 全局标志数组

`int visited[m]; /*全局标志向量*/`

2) 邻接表存储结构

```
/*  
    图的广度优先遍历算法  
    程序名bfs.c  函数名bfs()、bfstraverse()  
*/
```

```
void bfs(adjgraph g, int i)
```

```
{ int j; /*从顶点i出发广度优先遍历顶点i所在的连通分量*/
```

```
    edgenode *p;
```

```
    int queue[20], head,tail;          /*FIFO队列*/
```

```
    head=-1; tail=-1;                /*初始化空队列*/
```

```
    printf("%c ",g.adjlist[i].vertex); /*访问源点v*/
```

```
    visited[i]=1;
```

```
    queue[++tail]=i;                  /*被访问结点进队*/
```

```
while (tail>head)      /*当队列非空时，执行下列循环体*/
{ j=queue[++head]; /*出队*/
  p=g.adjlist[j].firstedge;
  while (p)            /*广度优先搜索邻接表*/
  { if (visited[p->adjvex]==0)
    { printf("%c ",g.adjlist[p->adjvex].vertex);
      queue[++tail]=p->adjvex;
      visited[p->adjvex]=1;
    }
    p=p->next;
  }
}
```

```
int bfstraverse(adjgraph g,datatype v)
{ int i,count=0;      /*广度优先遍历图g*/
  for (i=0;i<g.n;i++)   visited[i]=0;      /*初始化标志数组*/
  i=loc(g,v);           /*寻找顶点v在邻接表中的位序*/
  if (i!=-1)
  { count++;           /*连通分量个数加1*/
    bfs(g,i);
  }
  for (i=0;i<g.n;i++)
    if (!visited[i])   /*vi未访问过*/
      { printf("\n");
        count++;      /*连通分量个数加1*/
      }
```



```
bfs(g,i);          /*从顶点i出发广度优先遍历图g*/  
    }  
return count;      /*返回无向图g中连通分量的个数*/  
}
```

算法8.4 图的广度优先遍历算法（邻接表表示法）

算法的时间复杂度与深度优先算法相同。

作业：

8.4 图8.31是某个无向图的邻接表，请

(1) 画出此图；

(2) 写出从顶点A开始的DFS遍历结果。

(3) 写出从顶点A开始的BFS遍历结果。

第 8 章 图

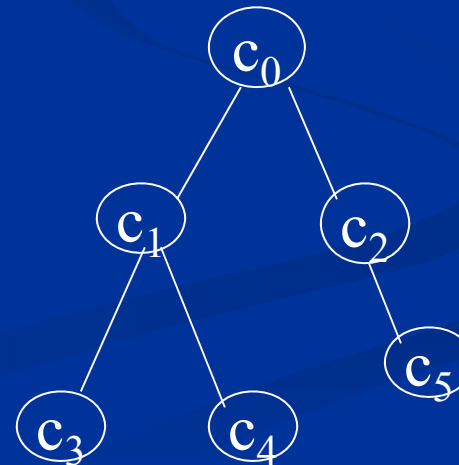
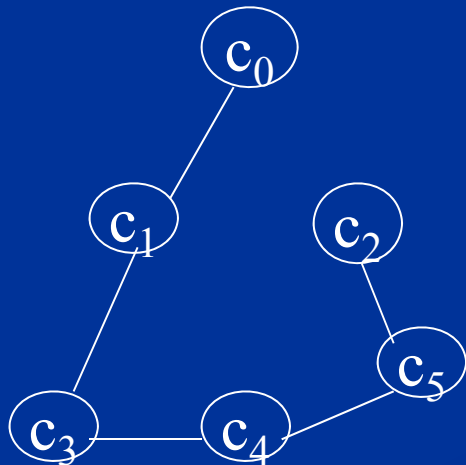
- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
 - 最短路径
 - 拓扑排序
 - 关键路径

8.5生成树与最小生成树

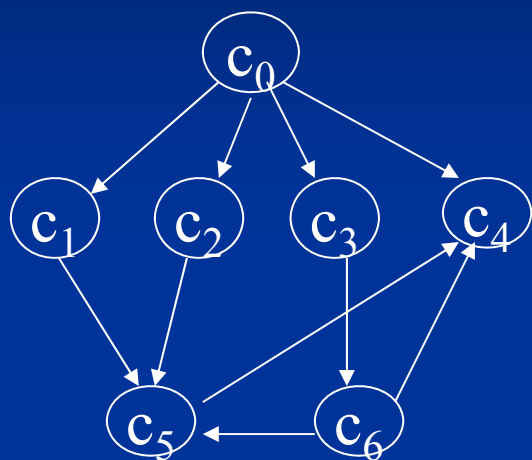
对于一个无向的连通图 $G=(V, E)$ ，设 G' 是它的一个子图，如果 G' 中包含了 G 中所有的顶点（即 $V(G')=V(G)$ ）且 G' 是无回路的连通图，则称 G' 为 G 一棵的生成树。

深度优先生成树：按深度优先遍历生成的生成树

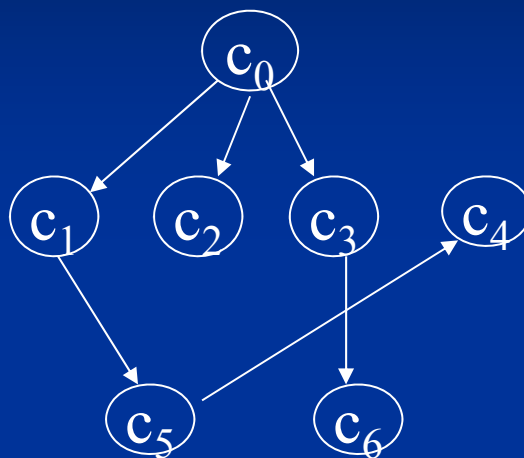
广度优先生成树：按广度优先遍历生成的生成树



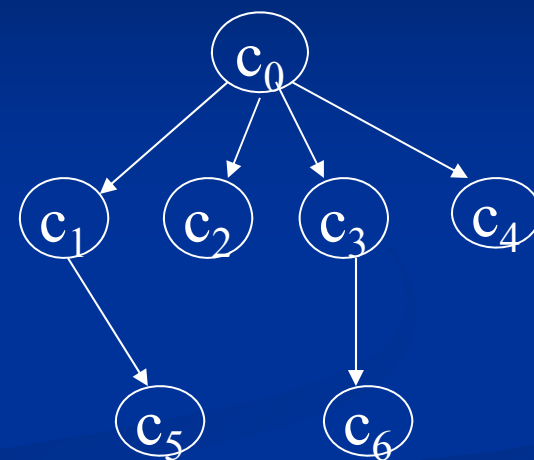
有向图的生成树



(a) 以 c_0 为根的有向图

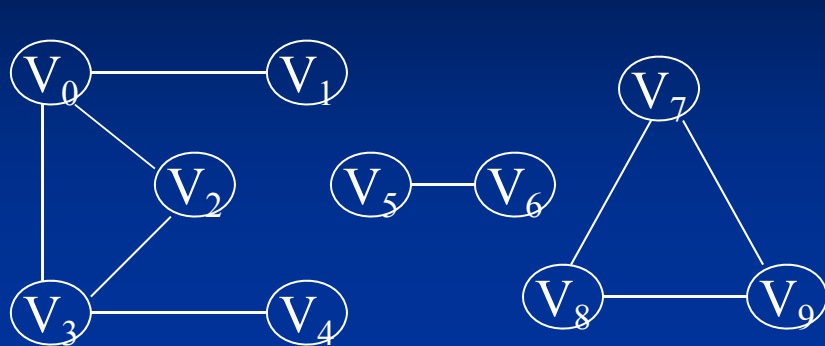


(b) DFS生成树

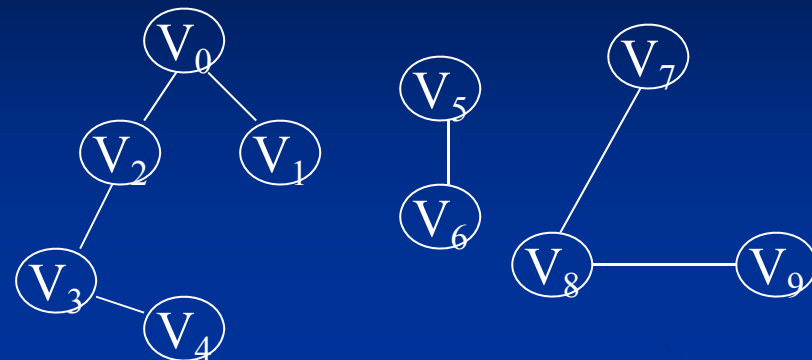


(c) BFS生成树

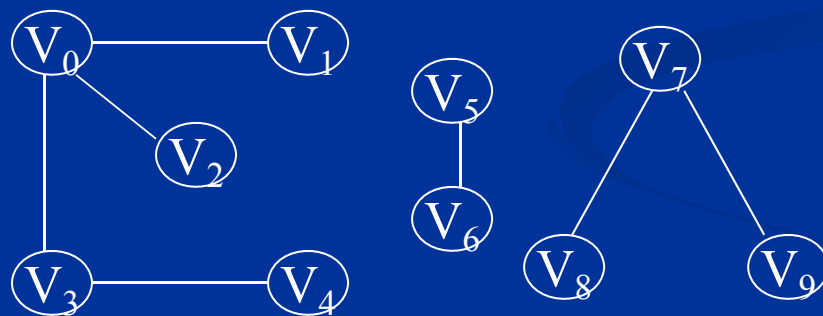
非连通图的生成森林



(a) 不连通的无向图 G_{12}



(b) 图 G_{12} 的一个DFS生成森林



(c) 图 G_{12} 的一个BFS生成森林

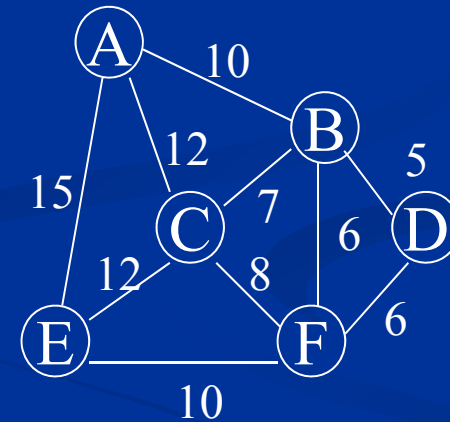
8.5.1 最小生成树的定义

若有一个连通的无向图 G ，有 n 个顶点，并且它的边是有权值的。在 G 上构造生成树 G' ，使这 $n-1$ 条边的权值之和在所有的生成树中最小。

例

要在 n 个城市间建立交通网，要考虑的问题如何在保证 n 点连通的前题下最节省经费？

上述问题即要使得生成树各边权值之各最小，即：



$$W(T) = \sum_{(u,v \in E)} w_{uv}$$

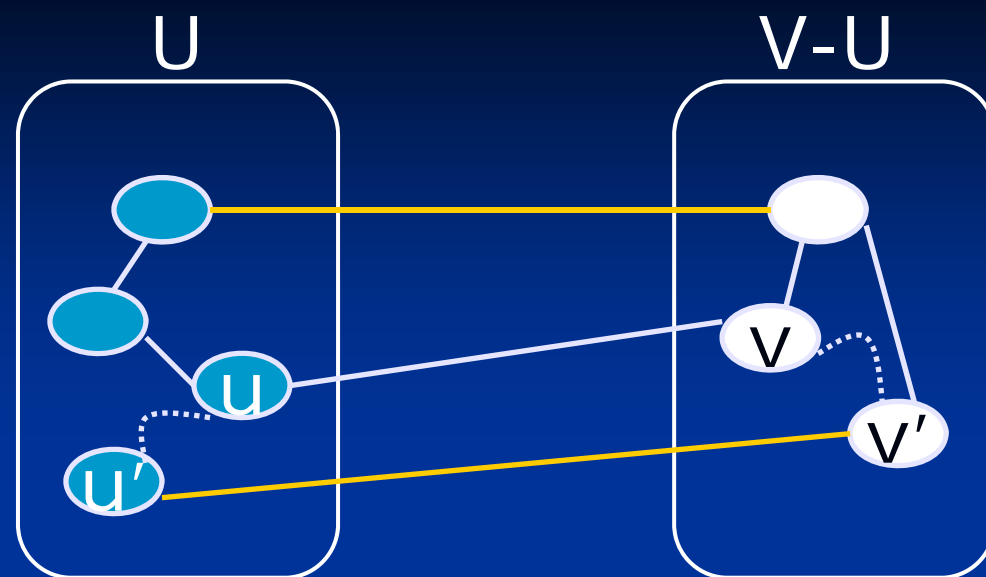
构造最小生成树的准则：

- 必须只使用该网络中的边来构造最小生成树；
- 必须使用且仅使用 $n-1$ 条边来联接网络中的 n 个顶点；
- 不能使用产生回路的边。

MST性质：

假设 $G=(V, E)$ 是一个连通网， U 是顶点集 V 的一个非空真子集，若 (u, v) 是满足 $u \in U, v \in V-U$ 的边（称这种边为两栖边）且 (u, v) 在所有的两栖边中具有最小的权值（此时，称 (u, v) 为最小两栖边），则必存在一棵包含边 (u, v) 的最小生成树。

证明:



设 (u, v) 是连接 U 与 $(V-U)$ 之间所有边中的最小边。假设 G 中任何边 (u, v) 的代价都不高于 (u, v) 的代价。设 T 是 G 的最小生成树。如果 T 包含 (u, v) ，那么 T 包含 (u, v) 的边 (u, v) 的代价都不高于 (u, v) 的代价。另一方面，由于 T 是生成树，则在 T 上必存在另一条边 (u', v') ，其中 $u' \in U, v' \in V-$

(Prim)算法和(Kruskal)算法是两个利用MST性质构造最小生成树的算法。

8.5.2最小生成树的普里姆算法

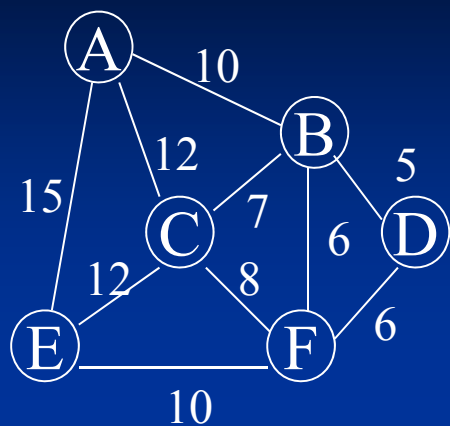
■ 普里姆算法的基本思想:

从连通网络 $G = \{ V, E \}$ 中的某一顶点 u_0 出发, 选择与它关联的具有最小权值的边 (u_0, v) , 将其顶点加入到生成树的顶点集合 U 中。

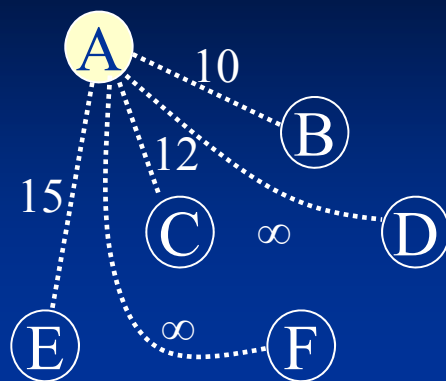
以后每一步从一个顶点在 U 中, 而另一个顶点不在 U 中的各条边中选择权值最小的边 (u, v) , 把它的顶点加入到集合 U 中。如此继续下去, 直到网络中的所有顶点都加入到生成树顶点集合 U 中为止。

Prim算法的基本步骤如下:

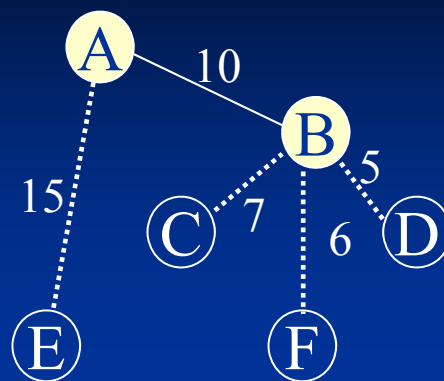
- (1) 初始化: $U=\{u_0\}$, $TREE=\{\}$;
- (2) 如果 $U=V(G)$, 则输出最小生成树 T , 并结束算法;
- (3) 在所有两栖边中找一条权最小的边 (u, v)
(若候选两栖边中的最小边不止一条, 可任选其中的一条), 将边 (u, v) 加入到边集 $TREE$ 中, 并将顶点 v 并入集合 U 中。
- (4) 由于新顶点的加入, U 的状态发生变化, 需要对 U 与 $V-U$ 之间的两栖边进行调整。
- (5) 转步骤 (2)



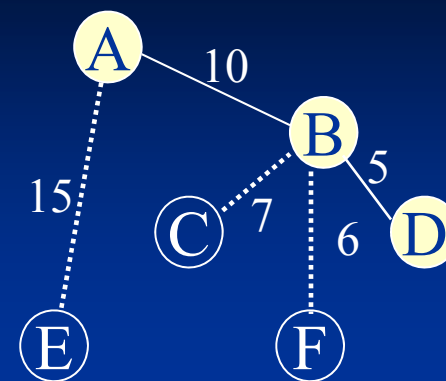
(a) 无向网



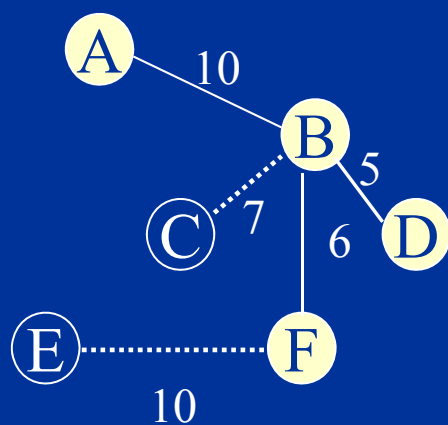
(b) 初始状态



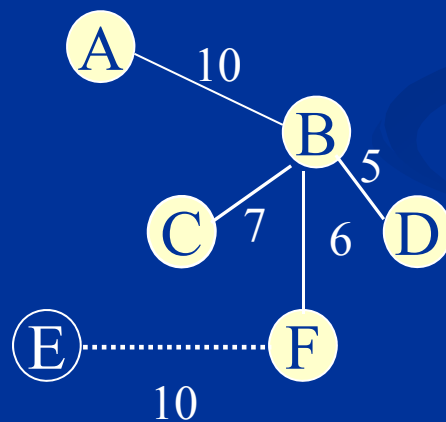
(c) 选取 (A、B)



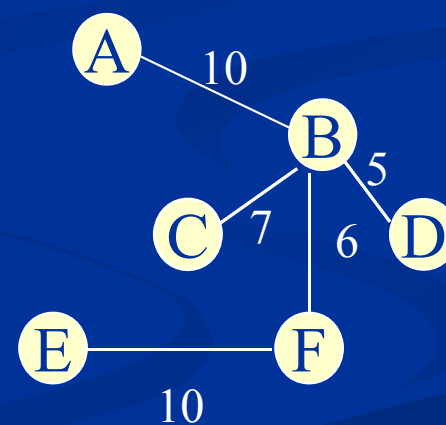
(d) 选取 (B、D)



(e) 选取 (B、F)



(f) 选取 (B、C)



(g) 选取 (E、F)

Prim算法实现:

1、连通图用邻接矩阵net表示:

$$\text{edges}[i][j] = \begin{cases} W_{ij} & \text{当 } (v_i, v_j) \in E(G) \text{ 且权为 } W_{ij} \\ 0 & \text{当 } i=j \\ \infty & \text{否则} \end{cases}$$

2、边tree（生成树） edge tree[n-1]

```
typedef struct edgedata
```

```
{ int beg,en;    /*beg,en是结点序号*/
```

```
  int length;    /*边长*/
```

```
} edge;
```

Prim算法构造最小生成树的过程

tree	0	1	2	3	4
beg	0	0	0	0	0
en	1	2	3	4	5
length	10	12	∞	15	∞

(a)初始态 $K=0$ $m=1$

tree	0	1	2	3	4
beg	0	0	0	0	0
en	1	2	3	4	5
length	10	12	∞	15	∞

(b)最小两栖边 (0, 1) $K=0$

tree 0 1 2 3 4

beg	0	1	1	0	1
en	1	2	3	4	5
length	10	7	5	15	6

(c) 最小两栖边 (0, 3) $K=1$



tree 0 1 2 3 4

beg	0	1	1	0	1
en	1	3	2	4	5
length	10	5	7	15	6

(d) 最小两栖边 (1, 5) $K=2$



tree 0 1 2 3 4

beg	0	1	1	5	1
en	1	3	5	4	2
length	10	5	6	10	7

(e) 最小两栖边 (1, 2)  K=3

tree 0 1 2 3 4

beg	0	1	1	1	5
en	1	3	5	2	4
length	10	5	6	7	10

(f) tree中存储了最小生成树的边

算法关键一步：求第k条轻边，将其加入tree中

1) 求当前最小两栖边及应添加的点v

```
min=tree[k].length;
```

```
s=k;
```

```
for (j=k+1;j<=g.n-2;j++)
```

```
    if (tree[j].length<min)
```

```
        {min=tree[j].length;
```

```
        s=j;
```

```
    }
```

```
v=tree[s].en; /*入选顶点为v*/
```

2) 通过交换, 将当前轻边加入tree中

```
x=tree[s];      tree[s]=tree[k];      tree[k]=x;
```

3) 调整各剩余点对应的最小两栖边 (由v加入引起)

```
for (j=k+1;j<=g.n-2;j++)  
    { d=g.edges[v][tree[j].en];  
      if (d<tree[j].length)  
          { tree[j].length=d;  
            tree[j].beg=v;  
          }  
    }
```

算法总体控制:

1) 初始化: 建立初始入选点, 并初始化生成树边集tree。

```
for (v=1;v<=g.n-1;v++)  
    { tree[v-1].beg=0;    /* 此处从顶点 $v_0$ 开始求最小生成树 */  
      tree[v-1].en=v;  
      tree[v-1].length=g.edges[0][v];  
    }
```

2) 依次求当前最小两栖边, 并将其加入tree

```
for (k=0;k<=g.n-3;k++) 执行关键一步
```

程序演示: prim.c

一般来讲,

由于普里姆算法的时间复杂度为 $O(n^2)$, 则适于稠密图。

8.5.3最小生成树的克鲁斯卡尔算法

Kruskal算法基本思想:

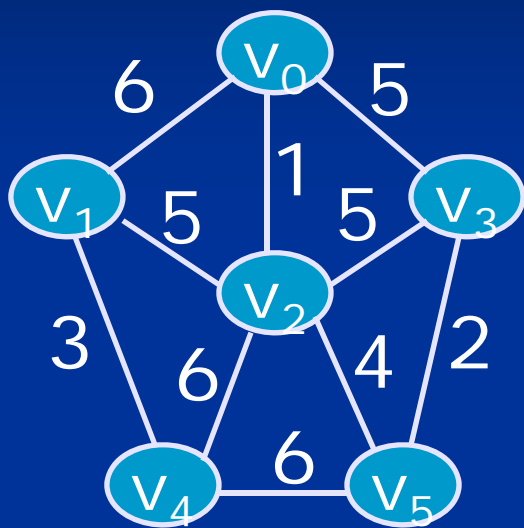
为使生成树上边的权值之和最小, 显然, 其中每一条边的权值应该尽可能地小。克鲁斯卡尔算法的做法就是: 先构造一个只含 n 个顶点的子图 SG , 然后从权值最小的边开始, 若它的添加不使 SG 中产生回路, 则在 SG 上加上这条边, 如此重复, 直至加上 $n-1$ 条边为止。

克鲁斯卡尔算法需对 e 条边按权值进行排序, 其时间复杂度为 $O(e \log e)$, 则适于稀疏图。

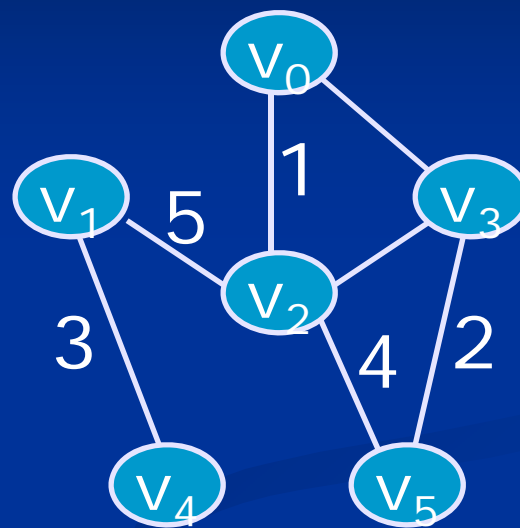
算法:

- (1) 初始化, $TV = \{v_0, v_1, \dots, v_n\}$, $TE = \{\}$;
- (2) 如果 TE 具有 $n-1$ 条边, 则输出最小生成树 T , 并结束算法。
- (3) 在有序的 $E(G)$ 边表序列中, 从当前位置向后寻找满足下面条件的一条边 (u, v) : 使得 u 在一个连通分量上, v 在另一个连通分量上, 即 (u, v) 是满足此条件权值最小的边, 将其加入到 T 中, 合并 u 与 v 所在的两个连通分量为一个连通分量。
- (4) 转 (2)

Kruskal算法动态演示:

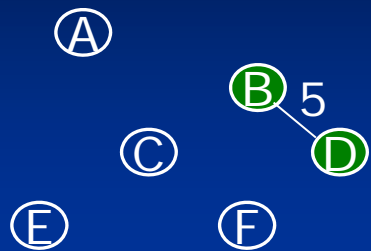


(a) 无向网络图

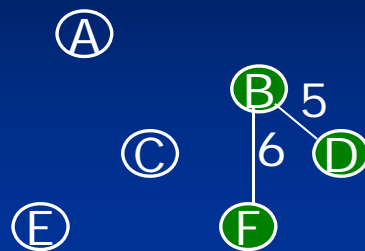


(b) 最小生成树求解过程

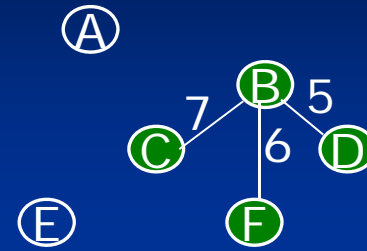
Kruskal算法构造最小生成树的过程



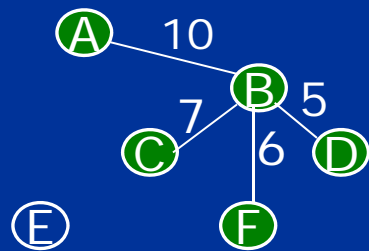
(a)



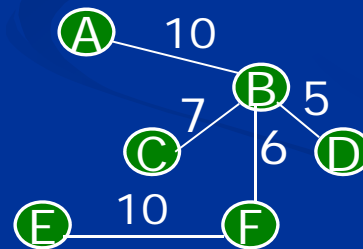
(b)



(c)



(d)



(e)


```

/*****/

/*    kruskal求解最小生成树算法    */

/*    文件名kruskal.c  函数名kruskal()  */

/*****/

void kruskal(edge adjlist[],edge tree[],int cnvx[],int n)
{ int v=0,j,k;
  for (j=0;j<n;j++)
      cnvx[j]=j;    /* 设置每一个顶点的连通分量 */
  for (k=0;k<n-1;k++) /*树中共有n-1条边*/
      { while (cnvx[adjlist[v].beg]==cnvx[adjlist[v].en] ) v++;
        /*找到属于两个连通分量权最小的边*/
      }
}

```

```
tree[k]=adjlist[v];    /*将边v加入到生成树中*/  
    for (j=0;j<n;j++) /*两个连通分量合并为一个连通分量*/  
        if (cnvx[j]==cnvx[adjlist[v].en])  
            cnvx[j]=cnvx[adjlist[v].beg];  
        v++;  
    }  
printf("最小生成树是: \n");  
for (j=0;j<n-1;j++)  
    printf("%3d%3d%6d\n",tree[j].beg,tree[j].en,tree[j].length);  
}
```

算法8.6 Kruskal求解最小生成树

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
 - 最短路径
 - 拓扑排序
 - 关键路径

8.6最短路径

问题的提出:

交通咨询系统、通讯网、计算机网络常要寻找两结点间最短路径;

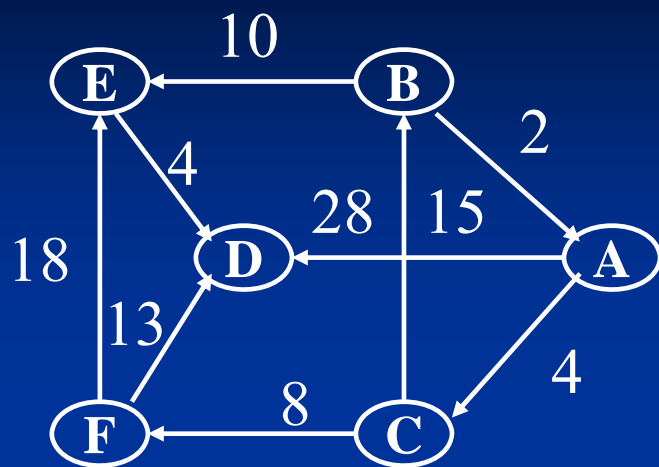
交通咨询系统: A 到 B 最短路径;

计算机网络: 发送Email节省费用 A到B沿最短路径传送;

路径长度: 路径上边数

路径上边的权值之和

最短路径: 两结点间权值之和最小的路径;



始点	终点	最短路径	路径长度
A	B	(A, C, B)	19
	C	(A, C)	4
	D	(A, C, F, D)	25
	E	(A, C, B, E)	29
	F	(A, C, F)	12

如何求从某源点
到其余各点的最短路径？

本节介绍求最短路径的两个算法

- 求从某个源点到其他各顶点的最短路径（单源最短路径）。
- 求每一对顶点之间的最短路径。

8.6.1 单源最短路径

单源最短路径问题是指：对于给定的有向网 $G=(V, E)$ ，求源点 v_0 到其它顶点的最短路径。

Dijkstra提出了一个按路径长度递增的顺序逐步产生最短路径的方法，称为**Dijkstra**算法。

Dijkstra算法的基本思想:

把图中所有顶点分成两组，**第一组**包括已确定最短路径的顶点，初始时只含有一个源点，记为集合**S**；**第二组**包括尚未确定最短路径的顶点，记为**V-S**。按最短路径长度递增的顺序逐个把**V-S**中的顶点加到**S**中去，直至从 v_0 出发可以到达的所有顶点都包括到**S**中。在这个过程中，总保持从 v_0 到第一组（**S**）各顶点的最短路径都不大于从 v_0 到第二组（**V-S**）的任何顶点的最短路径长度，第二组的顶点对应的距离值是从 v_0 到此顶点的只包括第一组（**S**）的顶点为中间顶点的最短路径长度。对于**S**中任意一点 j ， v_0 到 j 的路径长度皆小于 v_0 到（**V-S**）中任意一点的路径长度。

- 引入一个辅助数组 $d[]$ 。它的每一个分量 $d[i]$ 表示当前找到的从源点 v_0 到顶点 v_i 的最短路径的长度。初始状态：
 - 若从源点 v_0 到顶点 v_i 有边，则 $d[i]$ 为该边上的权值
 - 若从源点 v_0 到顶点 v_i 没有边，则 $d[i]$ 为 $+\infty$ 。
- 一般情况下，假设 S 是已求得的最短路径的终点的集合，则可证明：下一条最短路径必然是从 v_0 出发，中间只经过 S 中的顶点便可到达的那些顶点 v_x ($v_x \in V-S$)的路径中的一条。
- 每次求得一条最短路径之后，其终点 v_k 加入集合 S ，然后对所有的 $v_i \in V-S$ ，修改其 $d[i]$ 值。

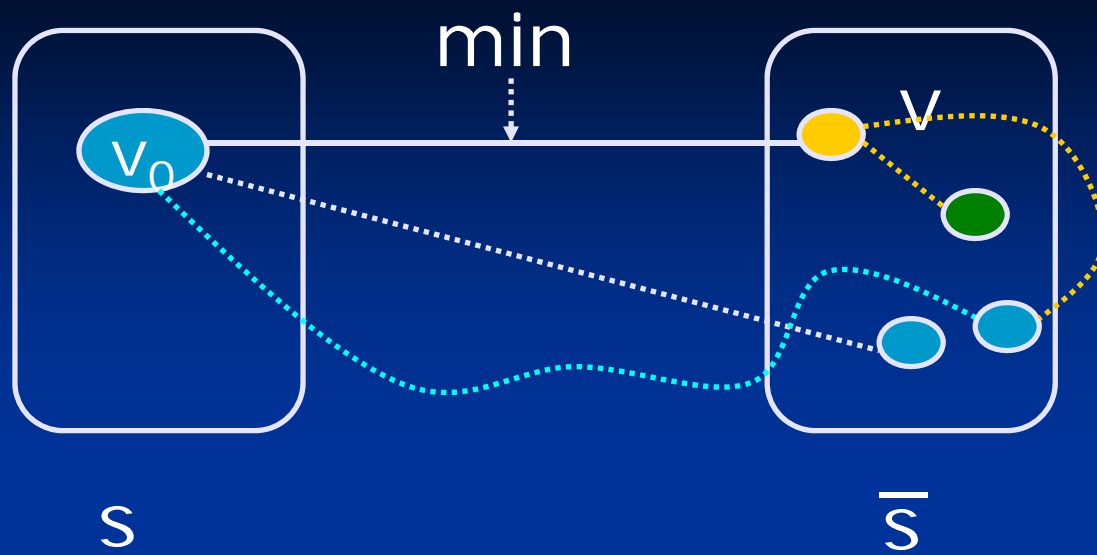
Dijkstra算法可描述如下：

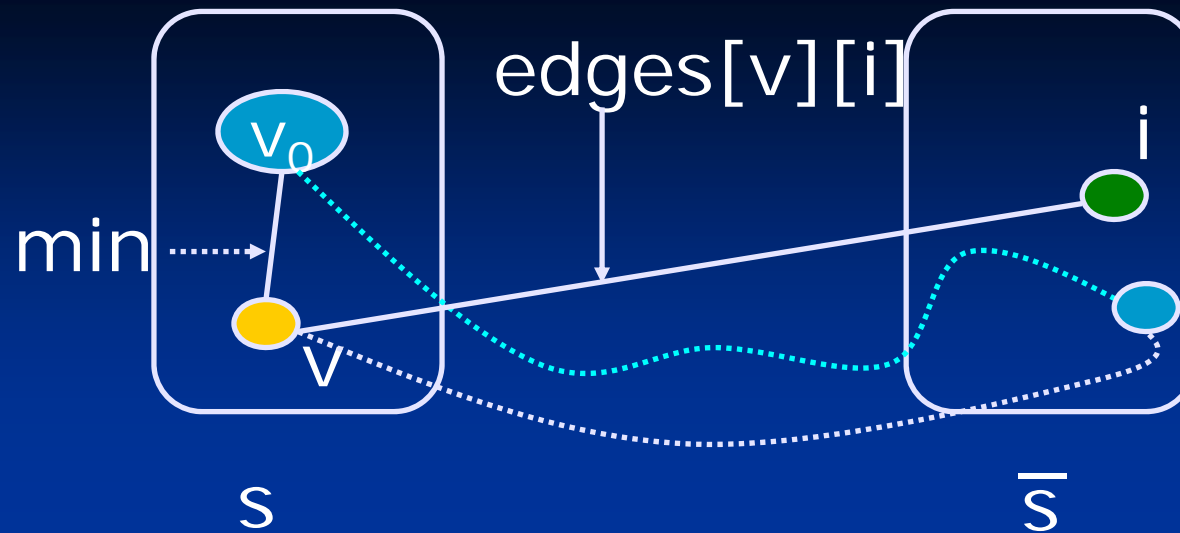
1) 初始化：把图中的所有顶点分成两组；初始化源点到各点的距离向量。

$\begin{cases} S: \text{已确定最短路径的点集, 初始 } S \leftarrow \{ v_0 \}; \\ \bar{S}: \text{尚未确定最短路径的点集, 初始 } \bar{S} \leftarrow V(G) - v_0; \end{cases}$

$d[j] \leftarrow g.edges[v_0][j], \quad j = 1, 2, \dots, n-1;$
// n 为图中顶点个数

2) 求出 S 与 \bar{S} 间的最短路径，及相应的点 v
 $d[v] \leftarrow \min\{ d[i] \}, \quad i \in V(G) - S;$
 $S \leftarrow S \cup \{ v \};$





3) 由于 v 的加入, 修改 S 中各结点与 \bar{S} 中各点的最短距离:

$$d[i] \leftarrow \min\{ d[i], d[v] + edges[v][i] \},$$

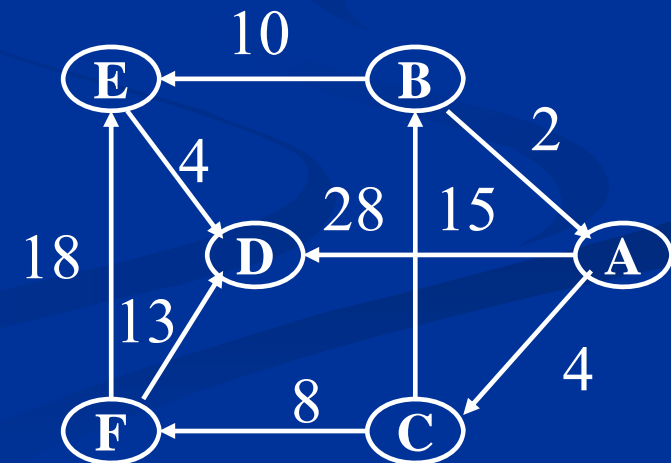
对于每一个 $i \in V - S$;

4) 判断: 若 $S = V$, 则算法结束, 否则转 2) 。

Dijkstra算法中各辅助数组的变化

循环	集合 S	v	距离向量 d						路径向量 p					
			0	1	2	3	4	5	0	1	2	3	4	5
初始化	{A}	-	0	∞	<u>4</u>	28	∞	∞	-1	-1	0	0	-1	-1
1	{AC}	2	0	19	4	28	∞	<u>12</u>	-1	2	0	0	-1	2
2	{ACF}	5	0	<u>19</u>	4	25	30	12	-1	2	0	5	5	2
3	{ACFB}	1	0	19	4	<u>25</u>	29	12	-1	2	0	5	1	2
4	{ACFBD}	3	0	19	4	25	<u>29</u>	12	-1	2	0	5	1	2
5	{ACFBDE}	4	0	19	4	25	29	12	-1	2	0	5	1	2

如何从表中读取源点O到终点v的最短路径？例如顶点A到D的最短距离是 $d[3]=25$ ，根据 $p[3]=5 \rightarrow p[5]=2 \rightarrow p[2]=0$ ，反过来排列，得到路径0, 2, 5, 3（即A、C、F、D）。



算法实现如下：

```
/******  
/*    单源最短路径算法    文件名:dijkstra.c    */  
/*    函数名:spath_dij()、 print_gpd()    */  
/******  
  
#include "c_ljz.c"    /*引入邻接矩阵创建程序*/  
  
typedef enum{FALSE,TRUE} boolean; /*false为0,true为1*/  
  
typedef int dist[m];    /* 距离向量类型*/  
  
typedef int path[m];    /* 路径向量类型*/  
  
void spath_dij(mgraph g,int v0,path p,dist d)  
{ boolean final[m];  
  int i,k,j,v,min,x;
```

/* 第1步 初始化集合S与距离向量d */

```
for (v=0;v<g.n;v++)
```

```
{ final[v]=FALSE;
```

```
  d[v]=g.edges[v0][v];
```

```
    if (d[v]<FINITY && d[v]!=0) p[v]=v0; else p[v]=-1;
```

/* v无前驱 */

```
}
```

```
final[v0]=TRUE; d[v0]=0; /*初始时s中只有 $v_0$ 一个顶点*/
```

/* 第2步 依次找出n-1个结点加入S中 */

```
for (i=1;i<g.n;i++)
```

```
{ min=FINITY;
```

```
  for (k=0;k<g.n;++k) /*找最小边及对应的入选顶点*/
```

```
if (!final[k] && d[k]<min) {v=k;min=d[k];}  /* !final[k] 表示k还在V-S中 */
```

```
    printf("\n%c---%d\n",g.vexs[v],min);    /*输出本次入  
选的顶点及距离*/
```

```
    if (min==FINITY) return;
```

```
    final[v]=TRUE;    /* V加入S*/
```

```
/*第3步 修改S与V-S中各结点的距离*/
```

```
for (k=0;k<g.n;++k)
```

```
    if ( !final[k] && (min+g.edges[v][k]< d[k]) )
```

```
        { d[k]=min+g.edges[v][k];
```

```
          p[k]=v;          }
```

```
    } /* end for */
```

```
}
```

```
void print_gpd(mgraph g,path p,dist d)
{ /*输出有向图的最短路径*/
    int st[20],i,pre,top=-1;    /*定义栈st并初始化空栈*/
    for (i=0;i<g.n;i++)
    { printf("\nDistanced: %7d , path:" ,d[i]);
        st[++top]=i;
        pre=p[i]; /*从第i个顶点开始向前搜索最短路径上的顶点*/
        while (pre!=-1)
        { st[++top]=pre;
            pre=p[pre];        }
        while (top>0)
            printf("%2d",st[top--]);    }    }
```



```
void main()    /*主程序*/
{ mgraph g; /* 有向图 */
    path p;    /* 路径向量 */
    dist d;    /* 最短路径向量 */
    int v0;

    creatmgraph1(&g); /*创建有向网的邻接矩阵*/
    print(g);          /*输出图的邻接矩阵*/
    printf("please input the source point v0:");
    scanf("%d",&v0);    /*输入源点*/
    spath_dij(g,v0,p,d); /*求v0到其他各点的最短距离*/
    print_gpd(g,p,d); /*输出V0到其它各点的路径信息及距离*/
}
```

8.6.2所有顶点对的最短路径

- 问题的提法：已知一个各边权值均大于0的带权有向图，对每一对顶点 $v_i \neq v_j$ ，要求求出 v_i 与 v_j 之间的最短路径和最短路径长度。

解决这个问题显然可以利用单源最短路径算法，具体做法是依次把有向网G中的每个顶点作为源点，重复执行Dijkstra算法n次，即执行循环体：总的时间复杂度为 $O(n^3)$ 。

```
for (v=0; v<g.n; v++)  
{ spath_dij (g, v, p, d) ;  
  print_gpd (g, p, d) ;  
}
```

下面将介绍用弗洛伊德 (Floyd) 算法来实现此功能, 时间复杂度仍为 $O(n^3)$, 但该方法比调用 n 次迪杰斯特拉方法更直观一些。

2. 弗洛伊德算法的基本思想

弗洛伊德算法仍然使用前面定义的图的邻接矩阵 `edges[N][N]` 来存储带权有向图。算法的基本思想是:

设置一个 $N \times N$ 的矩阵 `A[N][N]`, 其中除对角线的元素都等于0外, 其它元素 `A[i][j]` 的值表示顶点 i 到顶点 j 的最短路径长度, 运算步骤为:

开始时, 以任意两个顶点之间的有向边的权值作为路径长度, 没有有向边时, 路径长度为 ∞ , 此时, `A[i][j]=edges[i][j]`,

以后逐步尝试在原路径中加入其它顶点作为中间顶点，如果增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径，修改矩阵元素。具体做法为：

第一步，让所有边上加入中间顶点0，取 $A[i][j]$ 与 $A[i][0]+A[0][j]$ 中较小的值作 $A[i][j]$ 的值。

第二步，让所有边上加入中间顶点1，取 $A[i][j]$ 与 $A[i][1]+A[1][j]$ 中较小的值，完成后得到 $A[i][j]$...，如此进行下去，当第 n 步完成后，得到 $A[i][j]$,即为我们所求结果, $A[i][j]$ 表示顶点 i 到顶点 j 的最短距离。

因此，弗洛伊德算法可以描述为：

$$\left\{ \begin{array}{l} A^{(-1)}[i][j] = \text{edges}[i][j]; \text{ /*edges为图的邻接矩阵*/} \\ A^{(k+1)}[i][j] = \min\{A^k[i][j], A^k[i][k+1] + A^k[k+1][j]\} \end{array} \right.$$

其中 $k = -1, 1, 2, \dots, n-2$

下面给出Floyd的算法实现。

```
/*  
*****  
/*      Floyd 所有顶点对最短路径算法      */  
/*      文件名: floyd.c  函数名: floyd1()  */  
/*  
*****  
  
typedef int dist[m][m];    /* 距离向量*/  
typedef int path[m][m];    /* 路径向量*/  
void floyd1(mgraph g,path p,dist d)  
{ int i,j,k;  
  for (i=0;i<g.n;i++)      /*初始化*/  
    for (j=0;j<g.n;j++)  
      { d[i][j]=g.edges[i][j];
```

```

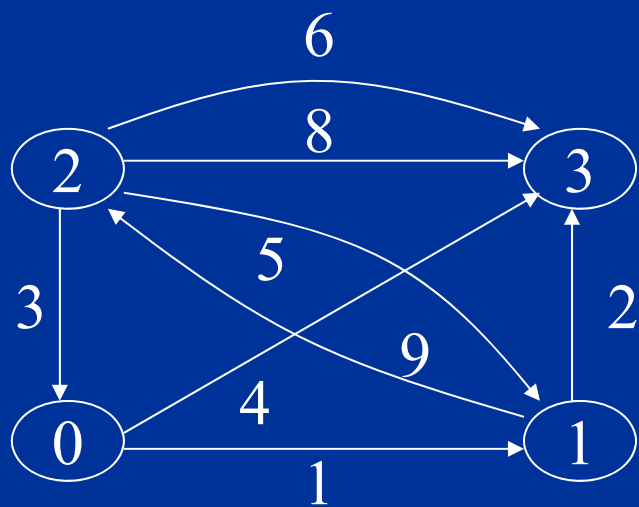
if (i!=j && d[i][j]<FINITY ) p[i][j]=i; else p[i][j]=-1;
}
for (k=0;k<g.n;k++) /*递推求解每一对顶点间的最短距离*/
{ for (i=0;i<g.n;i++)
    for (j=0;j<g.n;j++)
        if (d[i][j]>(d[i][k]+d[k][j]))
            { d[i][j]=d[i][k]+d[k][j];
              p[i][j]=k;
            }
    }
}

```

算法8.8 求网络中每一对顶点之间的最短路径

例

求下图中所在顶点对之间的最短路径。



0	1	∞	4
∞	0	9	2
3	5	0	8
∞	∞	6	0

D	D ₋₁				D ₀				D ₁				D ₂				D ₃			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	<u>10</u>	<u>3</u>	0	1	10	3	0	1	<u>9</u>	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	<u>12</u>	0	9	2	<u>11</u>	0	<u>8</u>	2
2	3	5	0	8	3	<u>4</u>	0	<u>7</u>	3	4	0	<u>6</u>	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	<u>9</u>	<u>10</u>	6	0	9	10	6	0
P	P ₋₁				P ₀				P ₁				P ₂				P ₃			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	0	-1	0	-1	0	-1	0	-1	0	<u>1</u>	<u>1</u>	-1	0	1	1	-1	0	<u>3</u>	1
1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	<u>2</u>	-1	1	1	<u>3</u>	-1	<u>3</u>	1
2	2	2	-1	2	2	<u>0</u>	-1	<u>0</u>	2	0	-1	<u>1</u>	2	0	-1	1	2	0	-1	1
3	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	<u>2</u>	<u>2</u>	3	-1	2	2	3	-1

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
- 最短路径
- 拓扑排序
- 关键路径

8.7 拓扑排序

一 AOV网

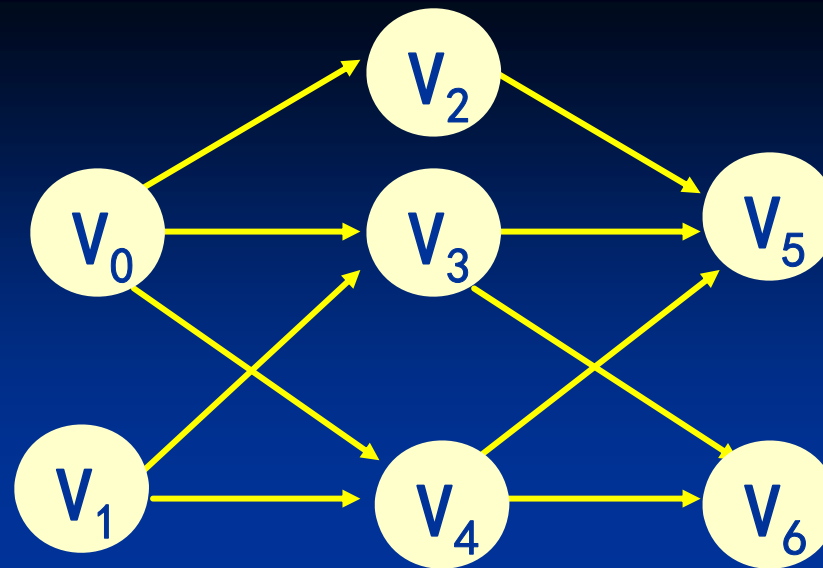
有向无环图：没有回路的有向图

应用： 工程流程、生产过程中各道工序的流程、程序流程、课程的流程。

AOV网(activity on vertex net)

用顶点表示活动，边表示活动的顺序关系的有向图称为AOV网。

某工程可分为7个子工程，若用顶点表示子工程（也称活动），用弧表示子工程间的顺序关系。工程流程可用如下AOV网表示



二 AOV网与拓扑排序

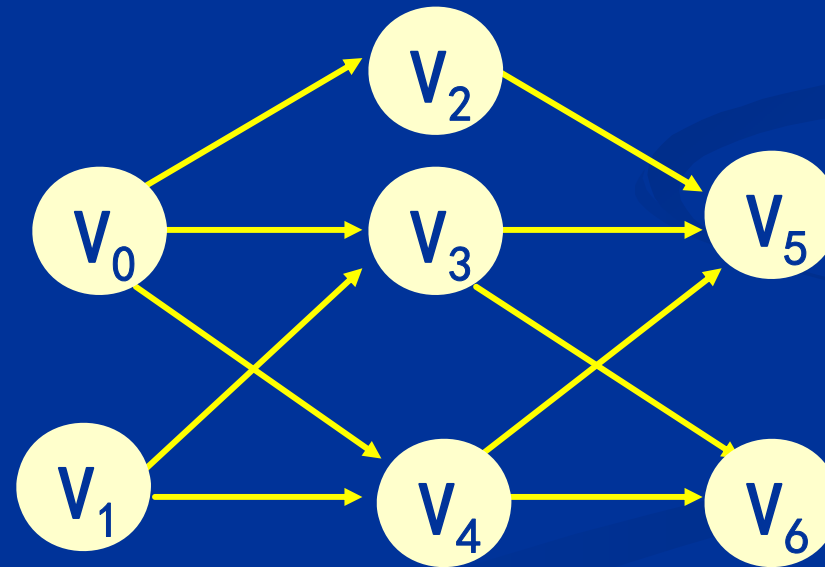
➤ 拓扑排序

对工程问题，人们至少关心如下两类问题：

- 1) 工程能否顺序进行，即工程流程是否“合理”？
- 2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程？

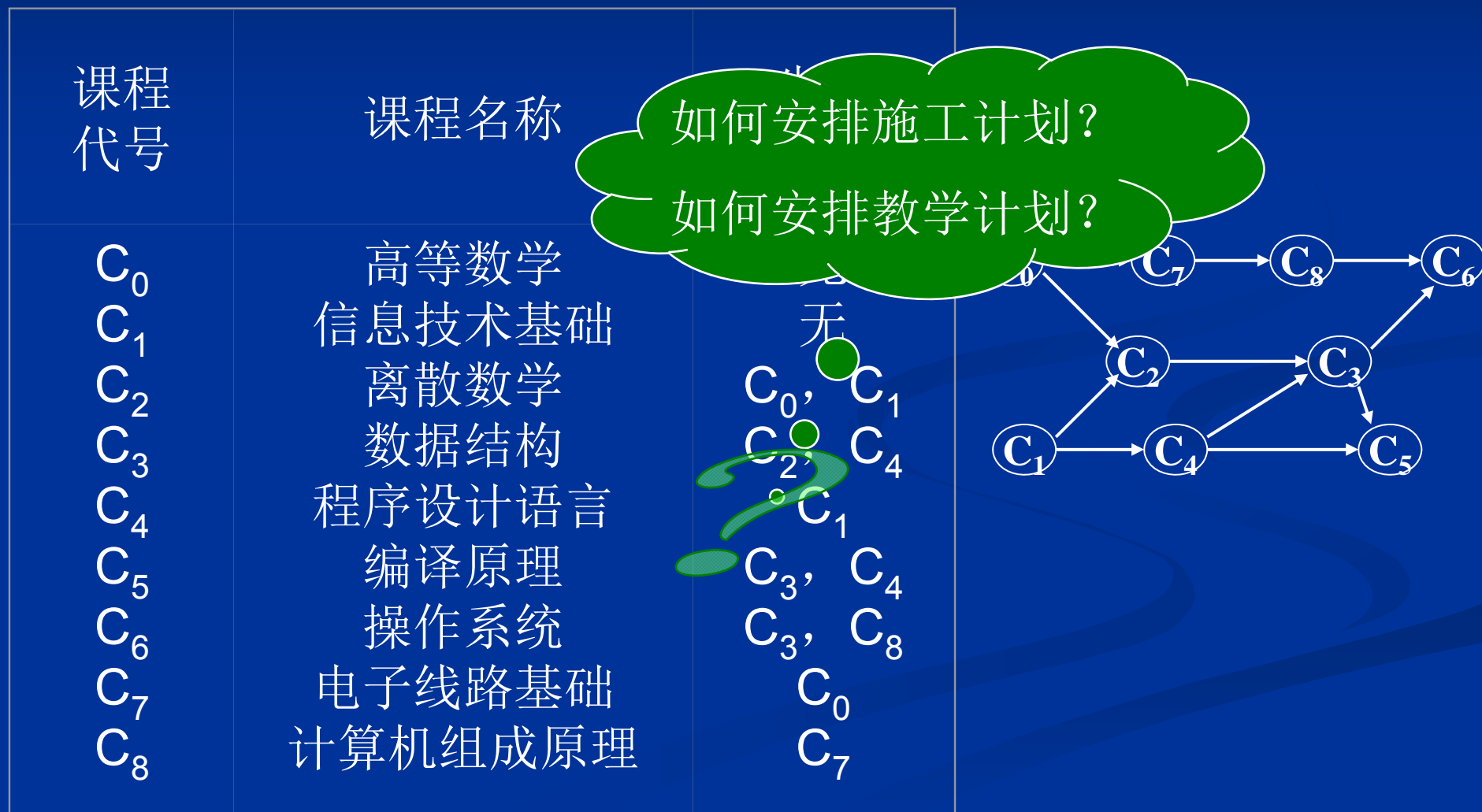
为求解工程流程是否“合理”，通常用AOV网的有向图表示工程流程。

例1 某工程可分为 V_0 、 V_1 、 V_2 、 V_3 、 V_4 、 V_5 、 V_6 7个子工程，工程流程可用如下AOV网表示。其中顶点：表示子工程（也称活动），弧：表示子工程间的顺序关系。



例 课程流程图

某校计算机专业课程流程可AOV网表示。其中顶点：表示课程（也称活动），弧：表示课程间的先修关系；



两个可行的学习计划为：

$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_4 \rightarrow C_7 \rightarrow C_8 \rightarrow C_3 \rightarrow C_6 \rightarrow C_5$

和

$C_1 \rightarrow C_0 \rightarrow C_7 \rightarrow C_8 \rightarrow C_2 \rightarrow C_4 \rightarrow C_3 \rightarrow C_5 \rightarrow C_6$

可行的计划的特点：若在流程图中顶点 v 是顶点 u 的前趋，则在计划序列中顶点 v 也是 u 的前趋。

拓扑序列：有向图D的一个顶点序列称作一个拓扑序列，如果该序列中任两顶点 v 、 u ，若在D中 v 是 u 前趋，则在序列中 v 也是 u 前趋。

拓扑排序：就是将有序向图中顶点排成拓扑序列。

拓扑排序的应用

- ☞ 安排施工计划
- ☞ 判断工程流程的是否合理

如何判断AOV网（有向图）

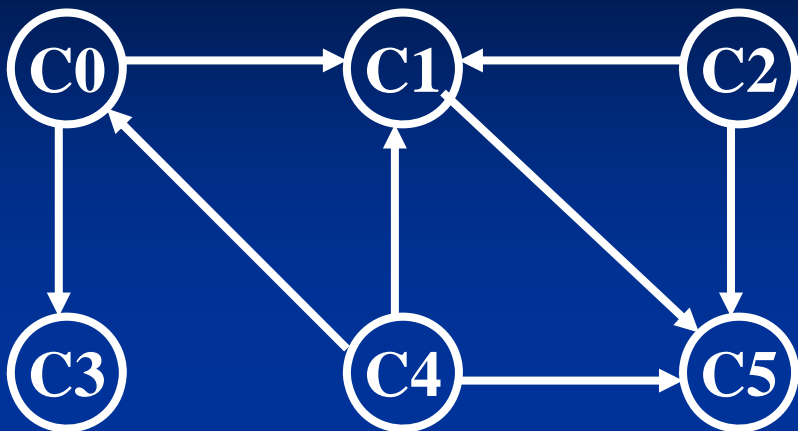
是否存在有向回路？

AOV网（有向图）
不存在有向回路
当且仅当能对AOV网
进行拓扑排序

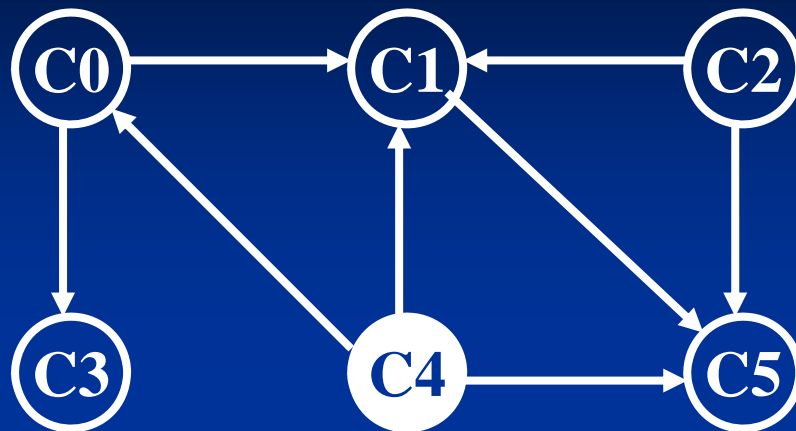
拓扑排序过程

- 1) 输入AOV网络。令 n 为顶点个数。
- 2) 在AOV网络中选一个没有直接前驱（入度为0）的顶点，并输出之；
- 3) 从图中删去该顶点，同时删去所有它发出的有向边；
- 4) 重复以上(2)、(3)步，直到
 - 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
 - 图中还有未输出的顶点，但已跳出处理循环。这说明图中还剩下一些顶点，它们都有直接前驱，再也找不到没有前驱的顶点了。这时AOV网络中必定存在有向环。

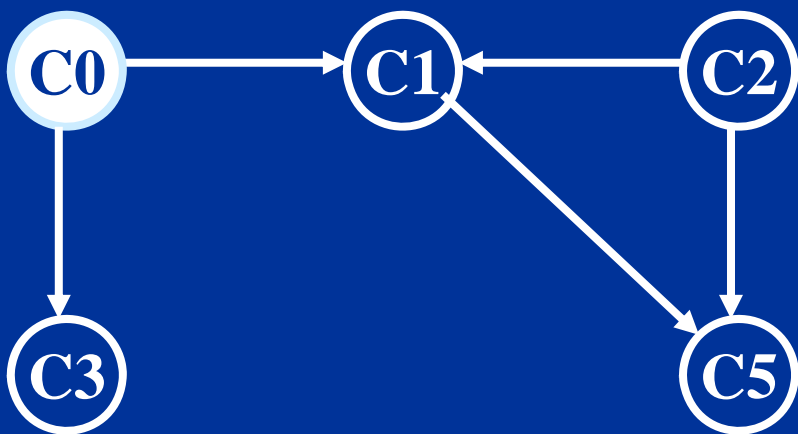
拓扑排序的过程



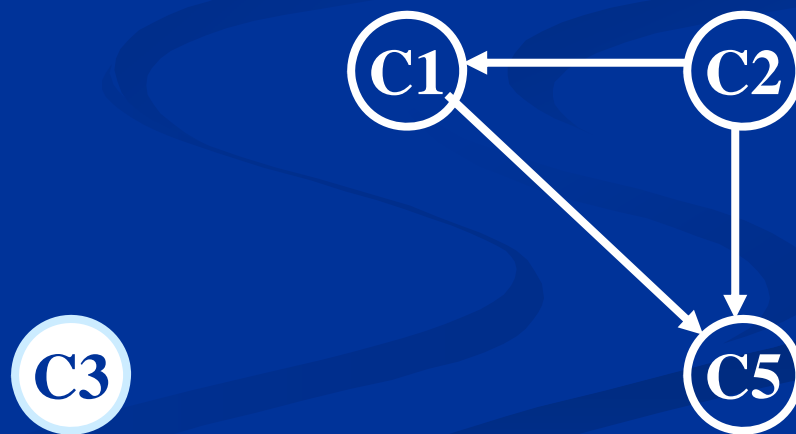
(a) 有向无环图



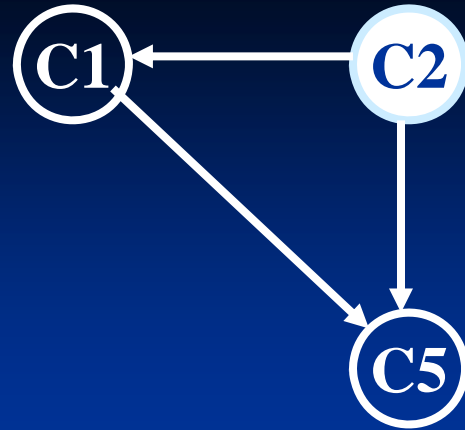
(b) 输出C4



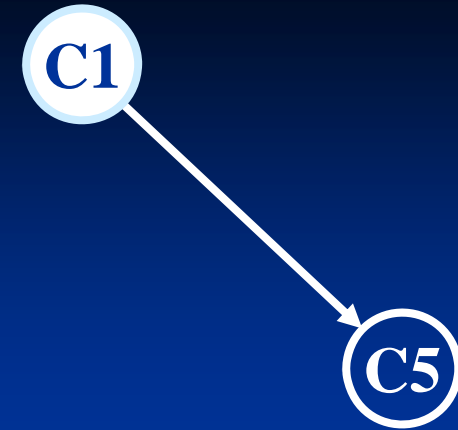
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

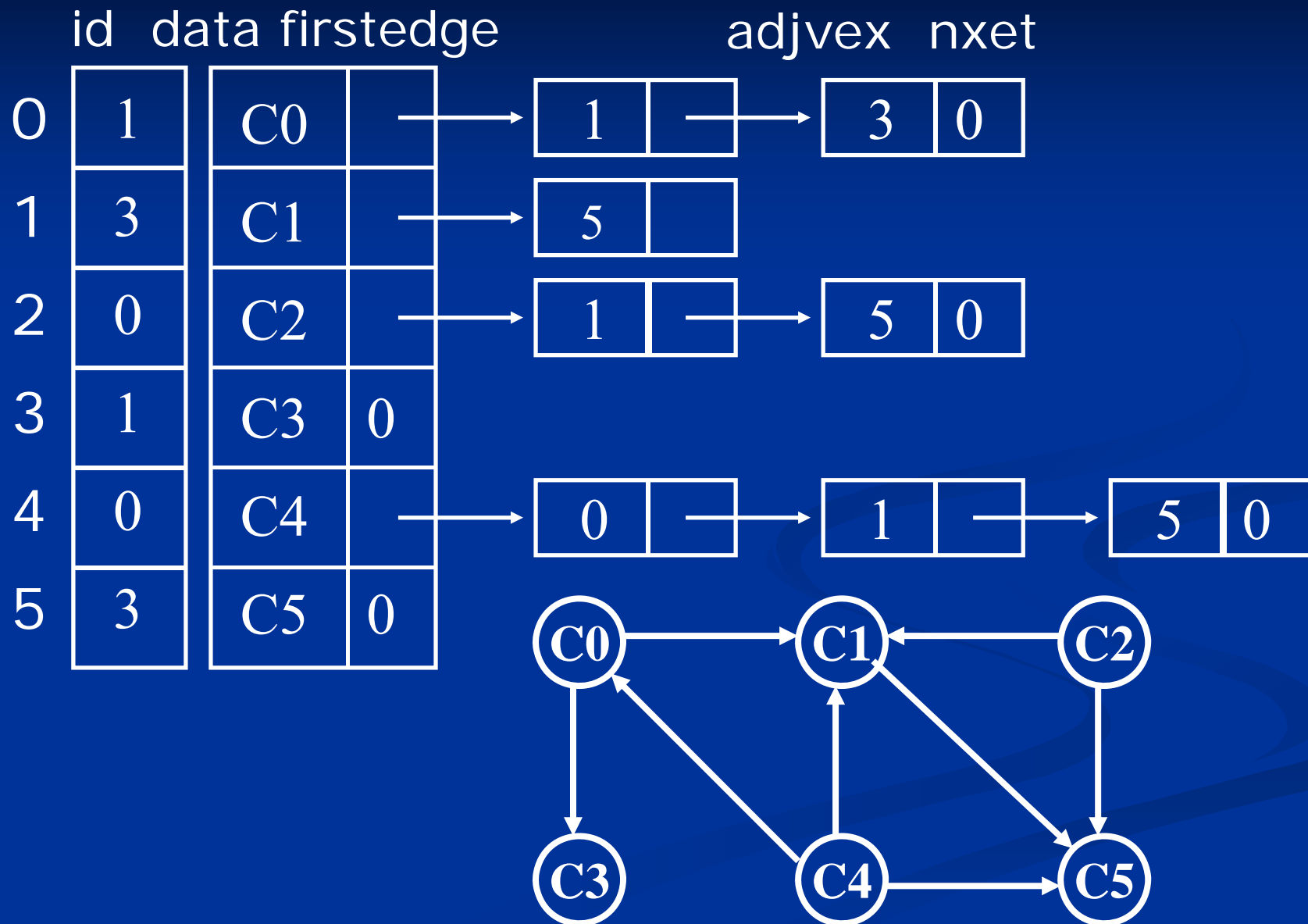


(g) 输出顶点C5

(h) 拓扑排序完成

最后得到的拓扑有序序列为 $C_4, C_0, C_3, C_2, C_1, C_5$ 。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如 C_4 和 C_2 ，也排出了先后次序关系。

AOV网络及其邻接表表示



这种带入度的邻接表存储结构定义如下：

```
#define m 20
```

```
typedef char vertextype;
```

```
typedef struct node{      /*边结点类型定义*/
```

```
    int adjvex;
```

```
    struct node *next;
```

```
}edgenode;
```

```
typedef struct de          /*带顶点入度的头结点定义*/
```

```
{ edgenode* firstedge;
```

```
    vertextype vertex;
```

```
    int id;                /*顶点的入度域*/
```

```
}vertexnode;
```

```
typedef struct{          /*AOV网络的邻接表结构*/
    vertexnode adjlist[m];
    int n,e;
}aovgraph;
```

基于这种存储结构，拓扑排序算法可描述为算法8.9。

```
/*******/
```

```
/*          拓扑排序算法          */
```

```
/*      文件名:topsort.c 函数名:topsort()      */
```

```
/*******/
```

```
int topsort(aovgraph g) /*函数返回拓扑排序输出的顶点个数*/
{int k=0,i,j,v, flag[m];
```

```
int queue[m]; /*队列*/
int h=0,t=0;
edgenode* p;
for (i=0;i<g.n;i++) flag[i]=0; /*访问标记初始化*/
for(i=0;i<g.n;i++) /*先将所有入度为0的顶点进队*/
    if( g.adjlist[i].id==0 && flag[i]==0)
        { queue[++t]=i;flag[i]=1; }
while (h<t) /*当队列不空时*/
{
    v=queue[++h]; /*队首元出队*/
    printf("%c----->",g.adjlist[v].vertex);
    k++; /*计数器加1*/
    p=g.adjlist[v].firstedge;
```

```
while(p)    /*将所有与v邻接的顶点的入度减1*/
{
    j=p->adjvex;
    if (--g.adjlist[j].id==0 && flag[j]==0)
        /*若入度为0则将其进队*/
        {queue[++t]=j; flag[j]=1;}
        p=p->next;
    }
}
return k;
}
```

算法8.9 拓扑排序

第 8 章 图

- 图的基本概念
 - 图的基本运算
 - 图的基本存储结构
 - 图的遍历
- 生成树与最小生成树
- 最短路径
- 拓扑排序
- 关键路径

8.8 关键路径

- 如果在无有向环的带权有向图中
 - 用有向边表示一个工程中的活动(Activity)
 - 用边上权值表示活动持续时间(Duration)
 - 用顶点表示事件 (Event)
- 则这样的有向图叫做用边表示活动的网络，简称AOE (Activity On Edges) 网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
 - (1) 完成整个工程至少需要多少时间(假设网络中没有环)?

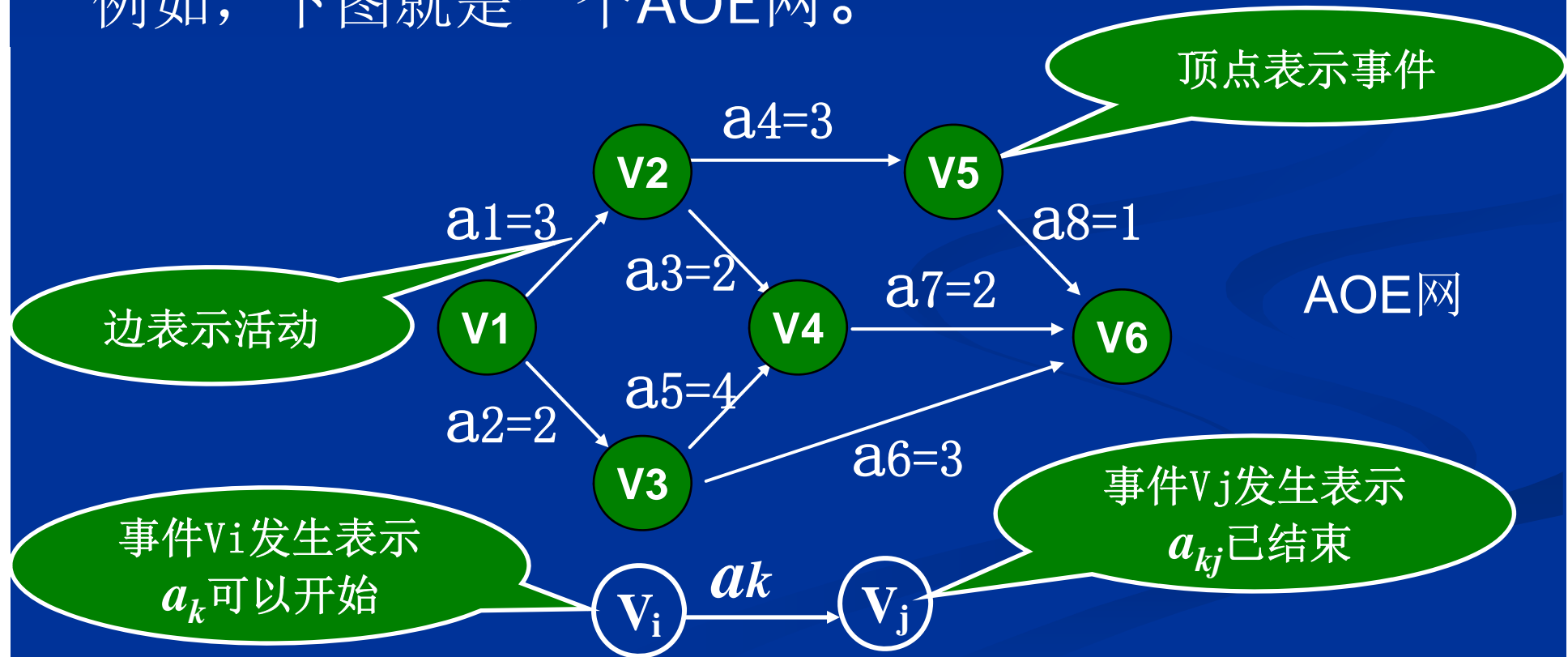
(2) 为缩短完成工程所需的时间, 应当加快哪些活动?

- 在AOE网络中, 有些活动顺序进行, 有些活动并行进行。
- 从源点到各个顶点, 以至从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同, 但只有各条路径上所有活动都完成了, 整个工程才算完成。
- 因此, 完成整个工程所需的时间取决于从源点到汇点的最长路径长度, 即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)。

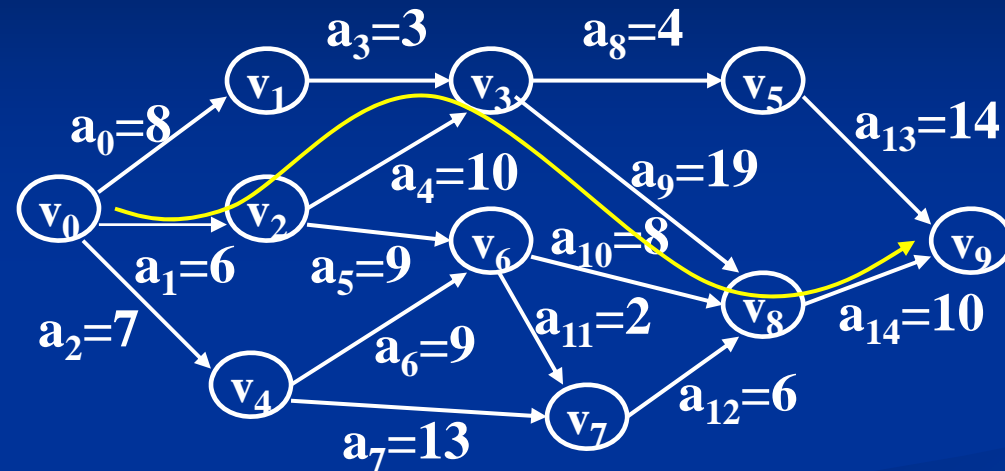
要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。

关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。

例如，下图就是一个AOE网。



关键路径求解方法：



➤在AOE网中从源点 v_0 到事件 v_i 的最长路径长度是事件 v_i 的最早发生时间。这个时间决定了所有以 v_i 为尾的弧表示的活动的最早开始时间。

定义以下几个量：

➤ $e(i)$: 表示活动 a_i 的最早开始时间。

➤ $l(i)$: 表示活动最迟开始时间的向量。

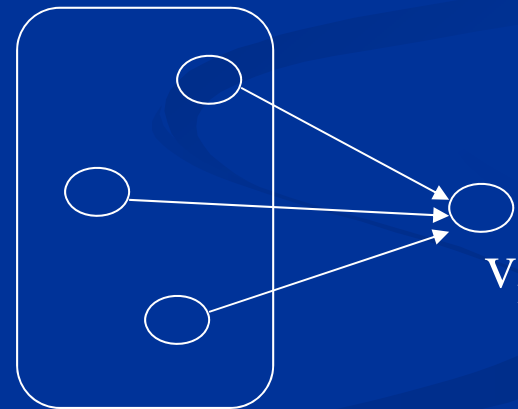
关键活动特征: $e(i) = l(i)$

$l(j) - e(j)$ 的值表示完成活动 a_j 的时间余量，提前完成非关键活动并不能提高整个工程的进度。

➤ 事件可能的最早开始时间 $v_e(i)$: 对于某一事件 v_i ，它可能的最早发生时间 $v_e(i)$ 是从源点到顶点 v_i 的最大路径长度。

➤事件允许的最晚发生时间 $v_l(i)$ ：对于某一事件 v_i ，它允许的最晚发生时间是在保证按时完成整个工程的前提下，该事件最晚必须发生的时间

$$\begin{cases} v_e(0) = 0; \\ v_e(i) = \max\{ve(j) + \text{活动} \langle v_j, v_i \rangle \text{ 持续的时间} \} (1 \leq i \leq n-1) \\ j \in p(i) \end{cases}$$



集合 $p(i)$

对于图8.25所示的AOE网络，可按其中的一个拓扑序列（ v_0 、 v_1 、 v_2 、 v_4 、 v_3 、 v_6 、 v_7 、 v_5 、 v_8 、 v_9 ）求解每个事件的最早开始时间：

$$v_e(0) = 0$$

$$v_e(1) = 8, v_e(2) = 6, v_e(4) = 7;$$

$$v_e(3) = \max\{v_e(1) + \text{len}(a_3), v_e(2) + \text{len}(a_4)\} = 16;$$

$$v_e(6) = \max\{v_e(2) + \text{len}(a_5), v_e(4) + \text{len}(a_6)\} = 16;$$

$$v_e(7) = \max\{v_e(6) + \text{len}(a_{11}), v_e(4) + \text{len}(a_7)\} = 20;$$

$$v_e(5) = v_e(3) + \text{len}(a_8) = 20;$$

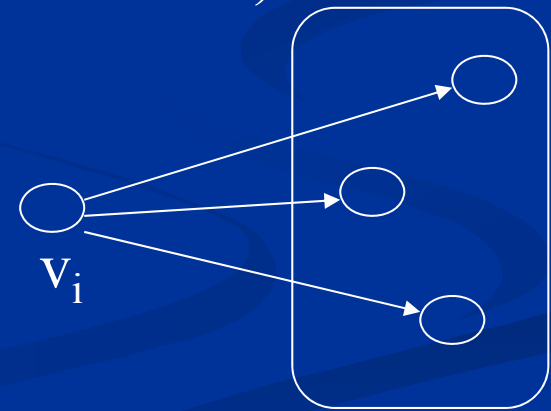
$$v_e(8) = \max\{v_e(3) + \text{len}(a_9), v_e(6) + \text{len}(a_{10}), v_e(7) + \text{len}(a_{12})\} = 35;$$

$$v_e(9) = \max\{v_e(5) + \text{len}(a_{13}), v_e(8) + \text{len}(a_{14})\} = 45;$$

$$\begin{cases} e(k) = v_e(i) ; \\ l(k) = v_l(j) - \text{len}(\langle v_i, v_j \rangle) ; \end{cases}$$

求每一个顶点*i*的最晚允许发生时间 $v_l(i)$ 可以沿图中的汇点开始，按图中的逆拓扑序逐个递推出每个顶点的 $v_l(i)$ 。

$$\begin{cases} v_l(n-1) = v_e(n-1) ; \\ v_l(i) = \min_{j \in s(i)} \{v_l(j) - \text{len}(\langle v_i, v_j \rangle)\} (0 \leq i \leq n-2) \end{cases}$$



集合 $s(i)$

对于图8.25所示的AOE网，按照（8-7）式求得的各个事件允许的最晚发生时间如下：

$$v_l(9) = v_e(9) = 45$$

$$v_l(8) = v_l(9) - \text{len}(<v_8, v_9>) = 45 - 10 = 35$$

$$v_l(5) = v_l(9) - \text{len}(<v_5, v_9>) = 45 - 14 = 31$$

$$v_l(7) = v_l(8) - \text{len}(<v_7, v_8>) = 35 - 6 = 29$$

$$v_l(6) = \min\{v_l(7) - \text{len}(<v_6, v_7>), v_l(8) - \text{len}(<v_6, v_8>)\} = \min\{27, 27\} = 27$$

$$v_l(3) = \min\{v_l(5) - \text{len}(<v_3, v_5>), v_l(8) - \text{len}(<v_3, v_8>)\} = \min\{27, 16\} = 16$$

$$v_l(4) = \min\{v_l(6) - \text{len}(<v_4, v_6>), v_l(7) - \text{len}(<v_4, v_7>)\} = \min\{18, 16\} = 16$$

$$v_l(2) = \min\{v_l(3) - \text{len}(\langle v_2, v_3 \rangle), v_l(6) - \text{len}(\langle v_2, v_6 \rangle)\} = \min\{6, 18\} = 6$$

$$v_l(1) = v_l(3) - \text{len}(\langle v_1, v_3 \rangle) = 13$$

$$v_l(0) = \min\{v_l(1) - 8, v_l(2) - 6, v_l(4) - 7\} = \min\{5, 0, 9\} = 0$$

顶点	v_e	v_l	活动	e	l	$l-e$	关键 活动
v_0	0	0	a_0	0	5	5	
v_1	8	13	a_1	0	0	0	√
v_2	6	6	a_2	0	9	9	
v_3	16	16	a_3	8	13	5	
v_4	7	16	a_4	6	6	0	√
v_5	20	31	a_5	6	18	12	
v_6	16	27	a_6	7	18	11	
v_7	20	29	a_7	7	16	9	
v_8	35	35	a_8	16	27	11	
v_9	45	45	a_9	16	16	0	√
			a_{10}	16	27	11	
			a_{11}	16	27	11	
			a_{12}	20	29	9	
			a_{13}	20	31	11	
			a_{14}	35	35	0	√

第 8 章 图

- 图的基本概念

 - 图的基本运算

 - 图的基本存储结构

 - 图的遍历

- 生成树与最小生成树

 - 最短路径

 - 拓扑排序

- 关键路径

作业：1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 15

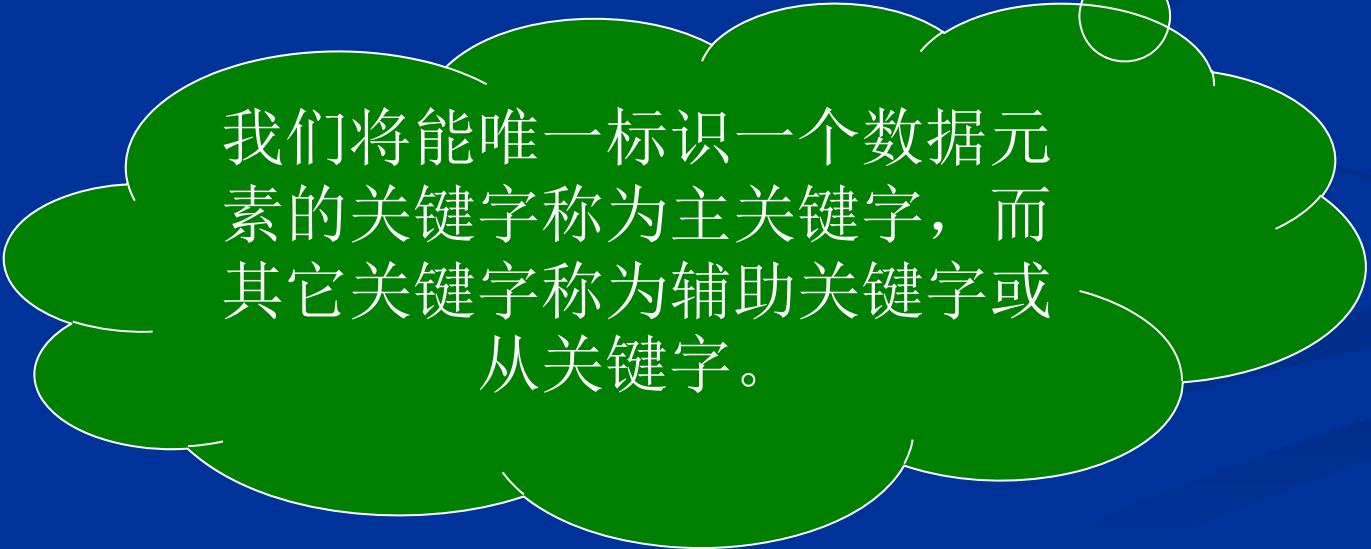
第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
- B-树
 - 散列表检索
 - 作业： 1, 3, 10, 14, 15

9.1 检索的基本概念

检索是确定数据元素集合中是否存在数据元素等于特定元素或是否存在元素满足某种给定特征的过程。

要进行检索，必须知道待检索对象的特征，也就是要知道待检索数据元素的**关键字**。



我们将能唯一标识一个数据元素的关键字称为主关键字，而其它关键字称为辅助关键字或从关键字。

静态检索表：检索的前后不会改变查找表的内容。

动态检索表：检索过程中可能会改变数据元素的存储位置。

检索算法的评价标准：平均查找长度ASL（Average Search Length），也就是为确定某一结点在数据集合中的位置，给定值与集合中的结点关键字所需进行的比较次数。

对于具有n个数据元素的集合，查找某元素成功的平均查找长度为：

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

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
 - B-树
 - 散列表检索

9.2 线性表的检索

线性结构是数据元素间最常见的数据结构，基于线性表的检索运算在各类程序中应用非常广泛，本节介绍三种在线性表上进行检索的方法，它们分别是顺序检索、二分法检索与分块检索。为简化问题，本节所介绍的检索方法均视为是基于静态查找表上的操作。

9.2.1 顺序检索

从表的一端开始，顺序（逐个）扫描线性表，依次将扫描到的结点关键字和给定值**Key**相比较，若当前扫描到的结点关键字与**Key**相等，则检索成功；若扫描结束后，仍未找到关键字等于**Key**的结点，则检索失败。

存储结构：顺序存储或链式存储

本节介绍基于顺序表的顺序检索算法。

```
/******  
/*  线性表检索用的头文件      */  
/*  文件名:seqlist.h          */  
/******  
  
#include <stdio.h>  
  
#define maxsize 100    /*预定义最大的数据域空间*/  
  
typedef int datatype;  /*假设数据类型为整型*/  
  
typedef struct {  
    datatype data[maxsize]; /*此处假设数据元素只包含一个整  
型的关键字域*/  
    int len;    /*线性表长度*/  
} seqlist;    /*预定义的顺序表类型*/
```

算法9.1给出了基于顺序查找表的顺序检索方法。

```
/******
```

```
/* 顺序检索算法 文件名:s_search.c */
```

```
/* 函数名: seqsearch1()、seqsearch2() */
```

```
*****
```

```
#include "seqlist.h"
```

```
/*-----顺序查找的非递归实现-----*/
```

```
int seqsearch1(seqlist l,datatype key)
```

```
{ int k=l.len-1;
```

```
while (k>=0 && l.data[k]!=key ) k--;
```

```
return(k);
```

```
}
```

/*-----顺序查找的递归实现-----*/

```
int seqsearch2(seqlist l,int n,datatype key)
```

```
{ int k=0;
```

```
  if (n== -1)
```

```
    k=-1;
```

```
    else if (l.data[n]==key) k=n;
```

```
    else k=seqsearch2(l,n-1,key);
```

```
  return(k);
```

```
}
```

算法9.1 线性表的顺序检索（顺序存储）

算法分析:

顺序检索的缺点是查找时间长。假设顺序表中每个记录的查找概率相同, 即 $P_i=1/n$ ($i=0, 1, \dots, n-1$), 查找表中第 i 个记录所需的进行的比较次数 $C_i=n-i$ 。因此, 顺序查找算法查找成功时的平均查找长度为:

$$ASL_{seq} = \sum_{i=0}^{n-1} P_i \cdot C_i = \sum_{i=0}^{n-1} \frac{1}{n} \cdot (n-i) = (n+1)/2$$

查找失败时, 算法的平均查找长度为:

$$ASL_{seq} = \sum_{i=0}^{n-1} \frac{1}{n} \cdot n = n$$

9.2.2 二分法检索

二分法检索又称为折半查找，采用二分法检索可以大大提高查找效率，它要求线性表结点按其关键字从小到大（或从大到小）按序排列并采用顺序存储结构。

■采用二分搜索时，先求位于搜索区间正中的对象的下标 mid ，用其关键码与给定值 x 比较：

➤ $I[mid].Key = x$ ，搜索成功；

➤ $I[mid].Key > x$ ，把搜索区间缩小到表的前半部分，再继续进行对分搜索；

➤ $I[mid].Key < x$ ，把搜索区间缩小到表的后半部分，再继续进行对分搜索。

■每比较一次，搜索区间缩小一半。如果搜索区间已缩小到一个对象，仍未找到想要搜索的对象，则搜索失败。

例

有一组有序的线性表如下：

(10, 14, 20, 32, 45, 50, 68, 90, 100, 120)

下面分析在其中二分检索关键字20的过程。

下标:	0	1	2	3	4	5	6	7	8	9
	10	14	20	32	45	50	68	90	100	120

↑↑ ↑

low=2 mid=2 high=3

第 3 次比较: 20=20, 检索成功, 返回位置 2。

下面分析二分检索关键字95的过程:

下标:	0	1	2	3	4	5	6	7	8	9
	10	14	20	32	45	50	68	90	100	120
								↑	↑	
								high=7	low=8	

此时, $high < low$, 即查找区间为空, 说明检索失败, 返回检索失败信息。

下面给出二分检索法的非递归与递归实现算法，算法中使用seqlist.h中定义的顺序查找表。

```
/******  
/* 二分查找算法    文件名: b_search.c    */  
/* 函数名: binsearch1()、binsearch2()    */  
/******  
/*-----二分查找的非递归实现-----*/  
int binsearch1(seqlist l,datatype key)  
{ int low=0,high=l.len-1,mid;  
  while (low<=high)  
    { mid=(low+high)/2;          /*二分*/
```

```
if (l.data[mid]==key) return mid; /*检索成功返回*/
    if (l.data[mid]>key)
        high=mid-1; /*继续在前半部分进行二分检索*/
    else low=mid+1; /*继续在后半部分进行二分检索*/
}
return -1; /* 当low>high时表示查找区间为空，检索失败*/
}
```

/*-----二分查找的递归实现-----*/

int binsearch2(seqlist l,datatype key,int low,int high)

{ int mid,k;

if (low>high) return -1; /*检索不成功的出口条件*/

else

{ mid=(low+high)/2; /*二分*/

if (l.data[mid]==key) return mid; /*检索成功返回*/

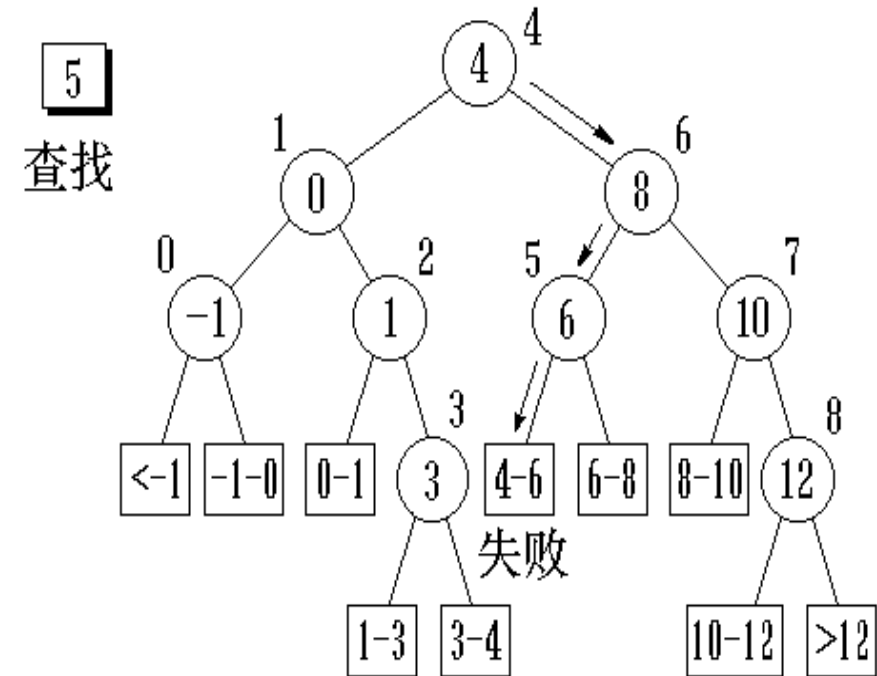
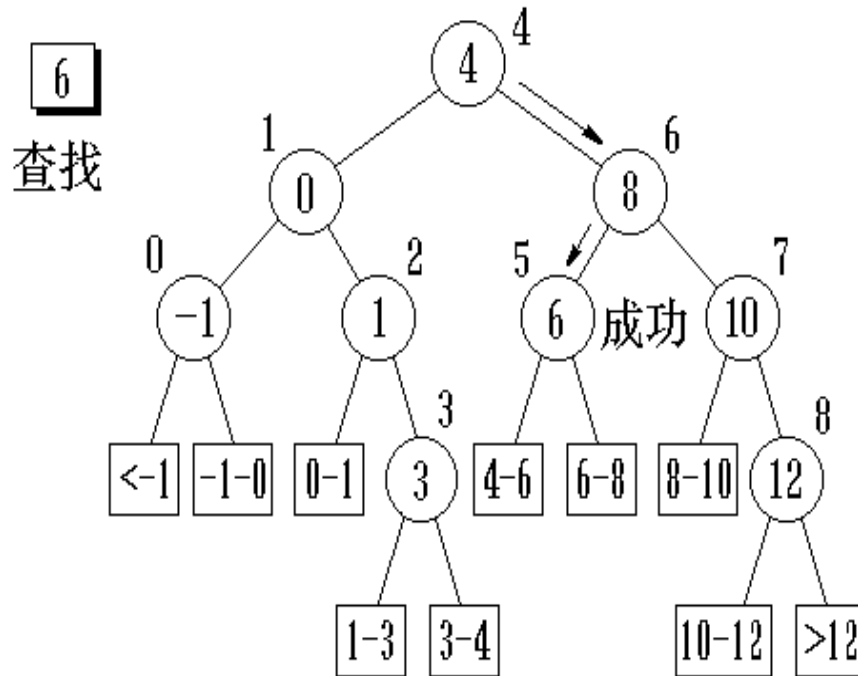
if (l.data[mid]>key)

return binsearch2(l,key,low,mid-1); /*递归地
在前半部分检索*/

else return binsearch2(l,key,mid+1,high); /*递归地
在后半部分检索*/

} }

从有序表构造出的二叉搜索树(判定树)



搜索成功的情形

搜索不成功的情形

- 若设 $n = 2^h - 1$ ，则描述对分搜索的二叉搜索树是高度为 $h-1$ 的满二叉树。 $2^h = n+1, h = \log_2(n+1)$ 。
- 第0层结点有1个，搜索第0层结点要比较1次；第1层结点有2个，搜索第1层结点要比较2次；...

在假定每个结点的查找概率相同的情况下，二分检索的平均查找次数为：

$$ASL_{bins} = \frac{1}{n}(1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + \dots + i \cdot 2^{i-1} + \dots + k \cdot 2^{k-1})$$

用数学归纳法容易证明： $\sum_{i=1}^k i \cdot 2^{i-1} = 2^k(k-1) + 1$

$$\begin{aligned} ASL_{bins} &= \frac{1}{n} \cdot (2^k \cdot (k-1) + 1) = \frac{1}{n} ((n+1)(\log_2(n+1) - 1) + 1) \\ &= \log_2(n+1) - 1 + \frac{1}{n} \log_2(n+1) \\ &= \log_2(n+1) - 1 \end{aligned}$$

9.2.3分块检索

分块查找(Blocking Search)又称索引顺序查找。它是一种性能介于顺序查找和二分查找之间的查找方法。

1、 查找表存储结构

查找表由“分块有序”的线性表和索引表组成。

(1) “分块有序”的线性表

线性表R被均分为若干块，每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字，即表是“分块有序”的。

(2) 索引表

抽取各块中的最大关键字及其起始位置构成一个索引表 $ID[l..b]$ ，即：

$ID[i](1 \leq i \leq b)$ 中存放第 i 块的最大关键字及该块在表 R 中的起始位置。由于表 R 是分块有序的，所以索引表是一个递增有序表。

【例】例如，图9.2就是一个带索引的分块有序的线性表。其中线性表 L 共有20个结点，被分成3块，第一块中最大关键字25小于第二块中的最小关键字27，第二块中最大关键字55小于第三块中的最小关键字60。

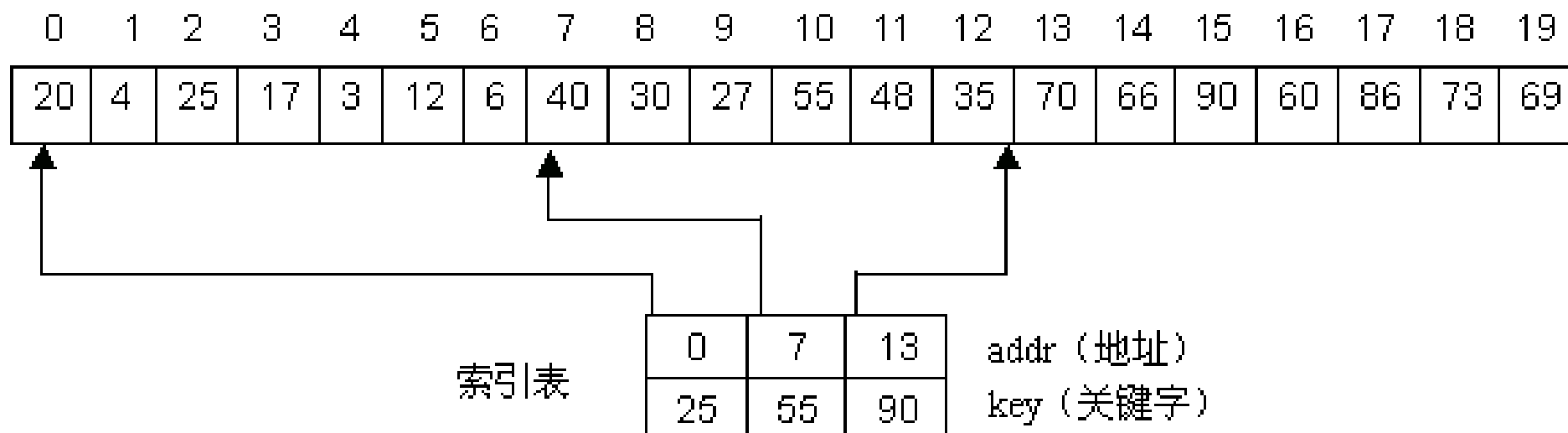


图9.2 分块有序表的索引存储表示

2、分块查找的基本思想

分块查找的基本思想是：

(1) 首先查找索引表

索引表是有序表，可采用二分查找或顺序查找，以确定待查的结点在哪一块。

(2) 然后在已确定的块中进行顺序查找

由于块内无序，只能用顺序查找。

分块检索方法通过将查找缩小在某个块中从而提高了检索的效率，其查找的效率由两部分组成，一是为确定某一块对索引表的平均查找长度 E_l ，二是块内查找所需的平均查找长度 E_b 。

➤ 若以顺序检索来确定块，则分块查找成功时的平均查找长度为：

$$ASL_{ids} = E_l + E_b = \frac{b+1}{2} + \frac{s+1}{2} = \frac{n/s + s}{2} + 1 = \frac{n + s^2}{2s} + 1$$

当 $s = \sqrt{n}$ 时， ASL_{ids} 取最小值 $\sqrt{n} + 1$

➤ 若以二分检索来确定块，则分块检索查找成功时的平均查找长度为：

$$ASL'_{ids} = E_l + E_b \approx \log_2 (b+1) - 1 + (s+1) / 2 \\ \approx \log_2 (n/s+1) + s/2$$

```
/*  
/*          分块查找算法          */  
/* 文件名: i_search.c  函数名: indexseqsearch()  */  
/*  
#include "seqlist.h"  
  
typedef struct    /*索引表结点类型*/  
{ datatype key;
```

```
    int address;  
    } indexnode;  
  
/*-----分块查找-----*/  
  
int indexseqsearch(seqlist l, indexnode index[], int m, datatype  
key)  
{ /*分块查找关键字为Key的记录，索引表为index[0..m-1]*/  
    int i=0, j, last;  
    while (i<m && key>index[i].key) i++;  
    if (i>=m) return -1;  
    else  
    { /*在顺序表中顺序检索*/
```

```
if (i==m-1) j=l.len-1;

    else j=index[i+1].address-1;    /*j初始时指向本块的
最后一个结点*/

    while (j>=index[i].address && key!=l.data[j] )
        j--;    /*从后向前逐个查找*/

    if (j<index[i].address) return -1;
    else return j;

}

}
```

算法9.3 分块检索

习题

9.1 在分块检索中，对256个元素的线性表分成多少块最好？每块的最佳长度是多少？若每块的长度为8，其平均检索的长度是多少？

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
 - B-树
 - 散列表检索

9.3 二叉排序树

1、~~在线性表的三种检索方法中二分检索法具有最高的查找效率，但是它只适合于顺序存储结构，查找(检索)树(Binary Search Tree)中数据的增、删带来不便。~~二叉排序树(Binary Sort Tree)的定义：二叉排序树或者是空树，或者是满足如下性质的二叉树：

- ①若它的左子树非空，则左子树上所有结点的值均小于根结点的值；
- ②若它的右子树非空，则右子树上所有结点的值均大于根结点的值；
- ③左、右子树本身又各是一棵二叉排序树。

上述性质简称二叉排序树性质(BST性质)，故二叉排序树实际上是满足BST性质的二叉树

2、二叉排序树的特点

由BST性质可得：

- (1) 二叉排序树中任一结点 x ，其左(右)子树中任一结点 y (若存在)的关键字必小(大)于 x 的关键字。
- (2) 二叉排序树中，各结点关键字是惟一的。

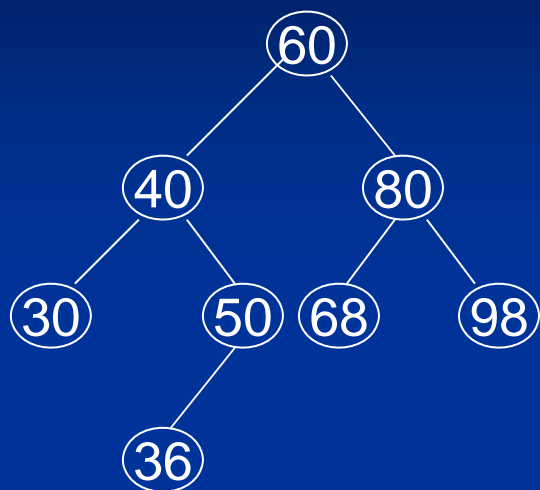
注意：

实际应用中，不能保证被查找的数据集中各元素的关键字互不相同，所以可将二叉排序树定义中BST性质(1)里的"小于"改为"大于等于"，或将BST性质(2)里的"大于"改为"小于等于"，甚至可同时修改这两个性质。

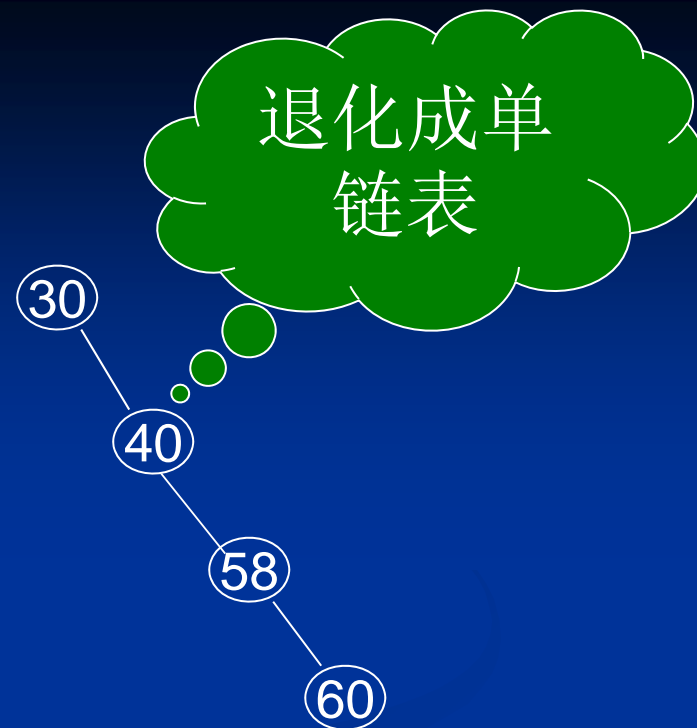
- (3) 按中序遍历该树所得到的中序序列是一个递增有序序列。

例

两棵二叉排序树



(a)



(b)

图9.3二叉排序树示例

对图9.3 (a) 所示的二叉排序树进行中序遍历得到的结果是30, 36, 40, 50, 60, 68, 80, 98; 对图9.3 (b) 所示的二叉排序树进行中序遍历的结果为30, 40, 50, 60。

```
/******
```

```
/*      二叉排序树用的头文件      */
```

```
/*      文件名: bstree.h      */
```

```
/******
```

二叉排序树存储结构
定义

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
typedef int datatype;
```

```
typedef struct node
```

```
{ datatype key;
```

```
    struct node *lchild,*rchild;
```

```
    }bnode;
```

```
typedef bnode *bstree;
```

/*二叉排序树结点定义*/

/*结点值*/

/*左、右孩子指针*/

一、基于二叉排序树的查找运算

对于一棵给定的二叉排序树，树中的查找运算很容易实现，其算法可描述如下：

- (1) 当二叉树为空树时，检索失败；
- (2) 如果二叉排序树根结点的关键字等于待检索的关键字，则检索成功；
- (3) 如果二叉排序树根结点的关键字小于待检索的关键字，则用相同的方法继续在根结点的右子树中检索；
- (4) 如果二叉排序树根结点的关键字大于待检索的关键字，则用相同的方法继续在根结点的左子树中检索。

```
/******
```

```
/* 基于二叉排序树的检索算法 文件名: t_search.c */
```

```
/*      函数名: bssearch1()、bssearch2()      */
```

```
/******
```

```
/*-----二叉排序树的非递归查找-----*/
```

```
void bssearch1(bstree t,datatype x, bstree *p,bstree *q)
```

```
{ /* q返回待查结点x在二叉排序树中的地址, p返回待查结点x  
的父结点地址 */
```

```
    *p=NULL;
```

```
    *q=t;
```

```
    while (*q)
```

```
        { if (x==(*q)->key) return;
```

```
*p=*q;  
    *q=(x<(*q)->key)? (*q)->lchild:(*q)->rchild;  
    }  
    return;  
}
```

/*-----二叉排序树的递归查找-----*/

bstree bssearch2(bstree t,datatype x)

{ /*在二叉排序树t中查找关键字为x的结点，若找到则返回该结点的地址，否则返回NULL*/

if (t==NULL || x==t->key)

return t;

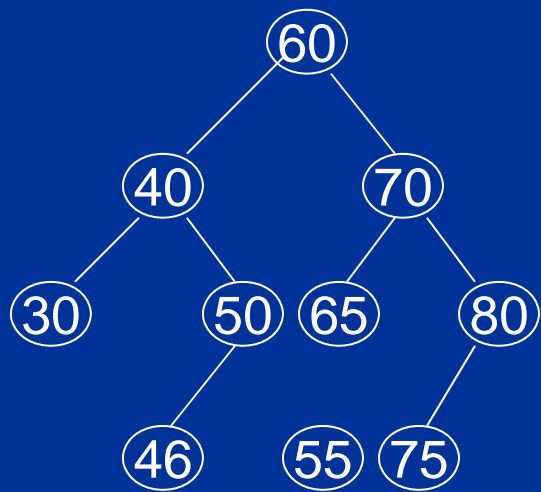
```
if (x<t->key)
    return bssearch2(t->lchild,x); /*递归地在左子树中检索*/
else
    return bssearch2(t->rchild,x); /*递归地在右子树中检索*/
}
```

算法分析:

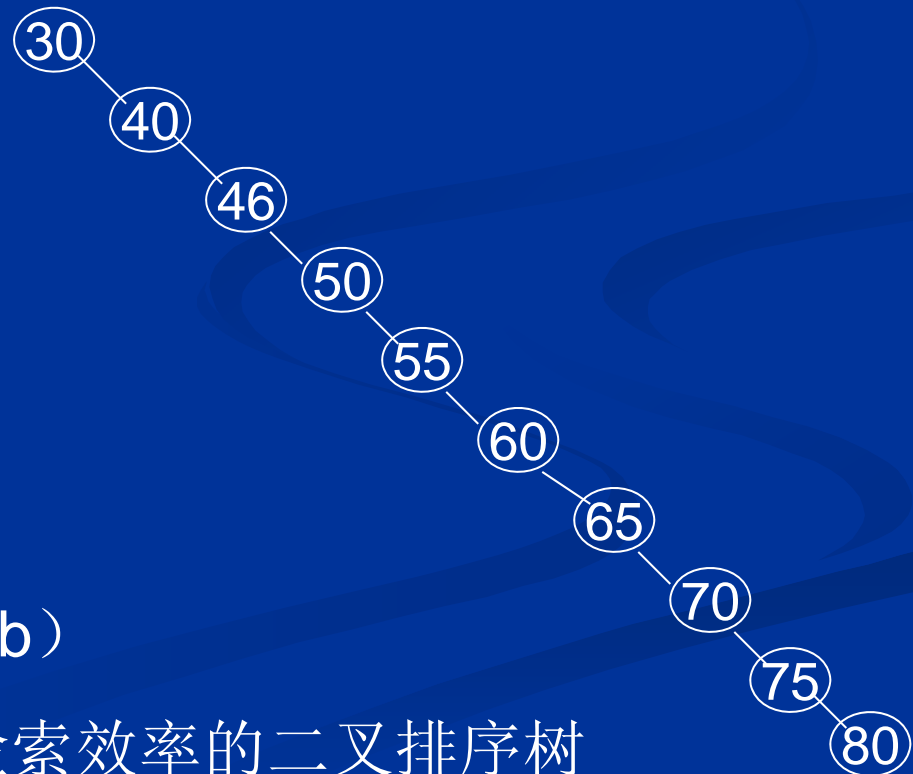
在二叉排序树上进行检索的方法与二分检索相似，和关键字的比较次数不会超过树的深度。

因此，在二叉排序树上进行检索的效率与树的形状有密切的联系。

在最坏的情况下，含有 n 个结点的二叉排序树退化成一棵深度为 n 的单支树（类似于单链表），它的平均查找长度与单链表上的顺序检索相同，即 $ASL = (n+1)/2$ 。在最好的情况下，二叉排序树形态比较匀称，对于含有 n 个结点的二叉排序树，其深度不超过 $\log_2 n$ ，此时的平均查找长度为 $O(\log_2 n)$ 。



(a)



(b)

图9.4两棵具有不同检索效率的二叉排序树

例如，对于图9.4中的两棵二叉排序树，其深度分别是4和10，在检索失败的情况下，在这两棵树上的最大比较次数分别是4和10；在检索成功的情况下，若检索每个结点的概率相等，则对于图9.4 (a) 所示的二叉排序树其平均查找长度为：

$$ASL_a = \sum_{i=1}^{10} p_i c_i = (1 + 2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 3$$

对于图9.4 (b) 所示的二叉排序树其平均查找长度为： $ASL_b = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10) / 10 = 5.5$

二、基于二叉树的插入运算

假设待插入的数据元素为 x ，则二叉排序树的插入算法可以描述为：

- 若二叉排序树为空，则生成一个关键字为 x 的新结点，并令其为二叉排序树的根结点；
- 否则，将待插入的关键字 x 与根结点的关键字进行比较，若二者相等，则说明树中已有关键字 x ，无须插入；
- 若 x 小于根结点的关键字，则将 x 插入到该树的左子树中，否则将 x 插入到该树的右子树中去。
- 将 x 插入子树的方法与在整个树中的插入方法是相同的，如此进行下去，直到 x 作为一个新的叶结点的关键字插入到二叉排序树中，或者直到发现树中已有此关键字为止。

```
/* **** */
/*      基于二叉排序树的结点插入算法      */
/*  文件名: t_insert.c 函数名: insertbstree()  */
/* **** */

void insertbstree(bstree *t,datatype x)
{ bstree f,p;
  p=*t;
  while (p) /*查找插入位置*/
  { if (x==p->key) return; /* 若二叉排序树t中已有key, 则
    无需插入 */
    f=p; /* f用于保存新结点的最终插入位置 */
    p=(x<p->key)? p->lchild:p->rchild;
  }
}
```

```
p=(bstree) malloc(sizeof(bsnode)); /*生成待插入的新结点*/  
p->key=x;  
p->lchild=p->rchild=NULL;  
if (*t==NULL)    *t=p;    /*原树为空*/  
    else  
        if (x<f->key)  
            f->lchild=p;  
        else f->rchild=p;  
}
```

程序9.5 基于二叉排序树的结点的插入算法

建立二叉排序树的算法如下：

```
/******
```

```
/*      二叉排序树的建立算法      */
```

```
/*  文件名: t_creat.c  函数名: creatbstree()  */
```

```
/******
```

```
bstree creatbstree(void)
```

```
{ /*根据输入的结点序列，建立一棵二叉排序树，并返回根结  
点的地址*/
```

```
    bstree t=NULL;
```

```
    datatype key;
```

```
    printf("\n请输入一个以-1为结束标记的结点序列: \n");
```

```
scanf("%d",&key);    /*输入一个关键字*/  
while (key!=-1)  
{   insertbstree(&t,key); /*将key插入二叉排序树t*/  
    scanf("%d",&key);  
}  
return t; /*返回建立的二叉排序树的根指针*/  
}
```

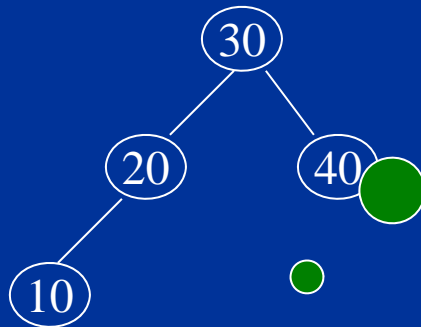
算法9.6 生成一棵二叉排序树

对于输入实例 (30, 20, 40, 10, 25, 45), 算法 9.6 创建二叉排序树的过程如下

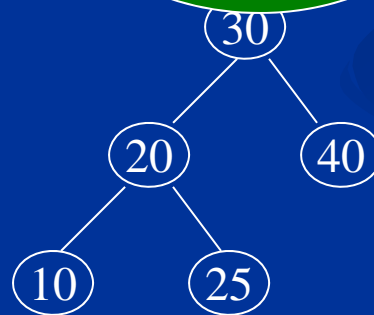
按照
45)

但若按照 (10, 20, 25, 30, 40, 45) 或 (45, 40, 30, 25, 20, 10) 的次序输入, 将分别生成只具有单个右分支和单个左分支的两棵二叉排序树。

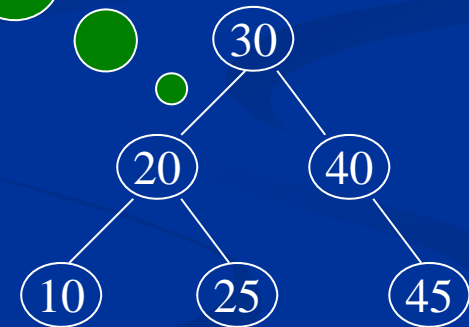
(a) 空树



(e) 插入10

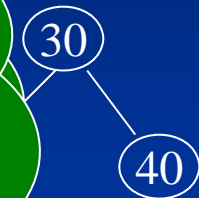


(f) 插入25



(g) 插入45

(d) 插入40



结论：

- 二叉排序树的形态与数据的输入次序相关；
- 由无序序列输入建立二叉排序树，再对其进行中序遍历可得一个有序序列，这种方法便是树排序。

三、二叉排序树的删除

从二叉排序树中删除一个结点，不能把以该结点为根的子树都删去，并且还要保证删除后所得的二叉树仍然满足**BST**性质。

➤删除操作的一般步骤

(1) 进行查找

查找时，令 p 指向当前访问到的结点， $parent$ 指向其双亲(其初值为 $NULL$)。若树中找不到被删结点则返回，否则被删结点是 $*p$ 。

(2) 删去 $*p$ 。

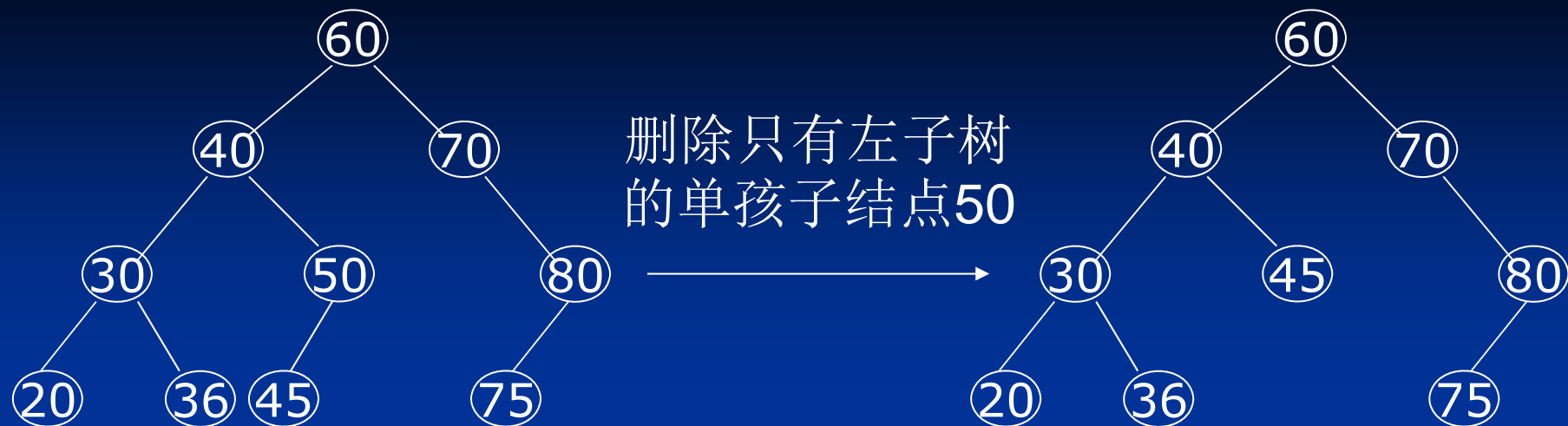
删 $*p$ 时，应将 $*p$ 的子树(若有)仍连接在树上且保持BST性质不变。按 $*p$ 的孩子数目分三种情况进行处理。

根据二叉排序树的结构特征，删除 $*p$ 可以分四种情况来考虑：

(1) 待删除结点为叶结点，则直接删除该结点即可。若该结点同时也是根结点，则删除后二叉排序树变为空树。图9.6(a)给出了一个删除叶结点的例子。



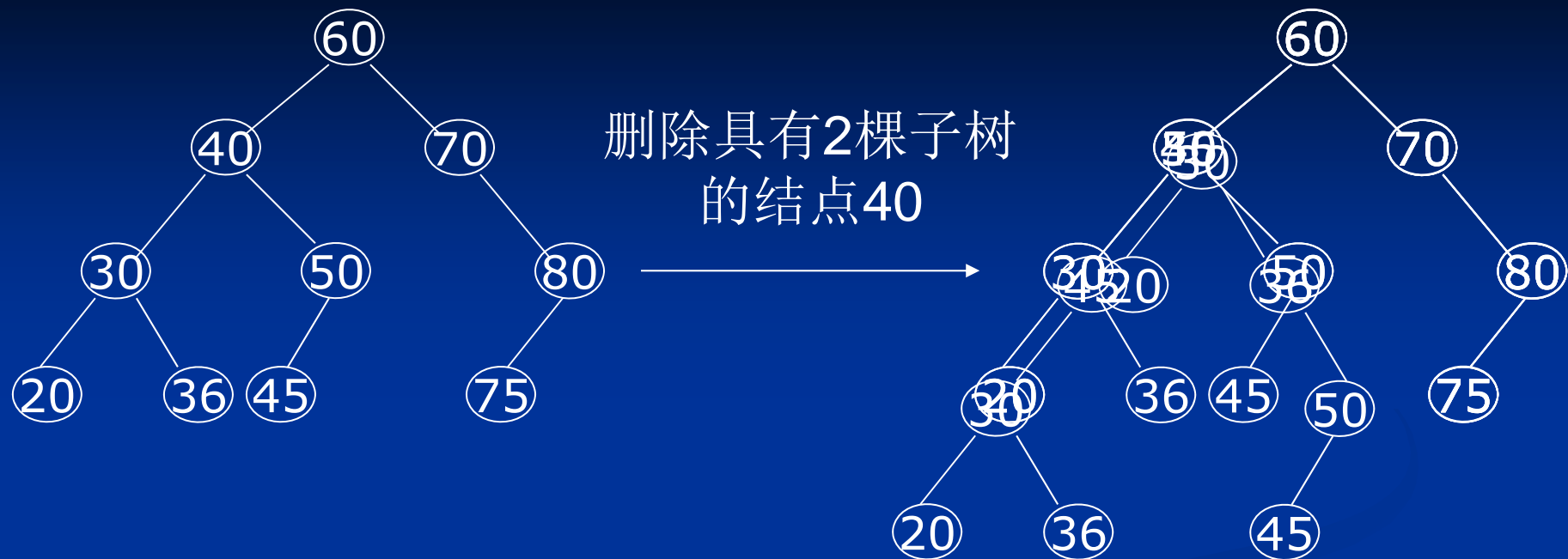
(2) 待删除结点只有左子树，而无右子树。根据二叉排序树的特点，可以直接将其左子树的根结点替代被删除结点的位置。即如果被删结点为其双亲结点的左孩子，则将被删结点的唯一左孩子收为其双亲结点的左孩子，否则收为其双亲结点的右孩子。图9.6 (b) 给出了一个例子。



(3) 待删除结点只有右子树，而无左子树。与情况(2)类似，可以直接将其右子树的根结点替代被删除结点的位置。即如果被删结点为其双亲结点的左孩子，则将被删结点的唯一右孩子收为其双亲结点的左孩子，否则收为其双亲结点的右孩子。图9.6(c)给出了一个例子。



(4) 待删除结点既有左子树又有右子树。根据二叉排序树的特点，可以用被删除结点中序下的前趋结点（或其中序下的后继结点）代替被删除结点，同时删除其中序下的前趋结点（或中序下的后继结点）。而被删除结点的中序前趋无右子树，被删除结点的中序后继无左子树，因而问题转换为第（2）种情况或第（3）种情况。



除此之外，还可以直接将删结点的右子树代替被删除结点，同时将被删除结点的左子树收为被删结点右子树中序首点的左孩子。也可以直接将删除结点的左子树代替被删除结点，同时将被删结点的右子树收为被删结点左子树中序尾点的右孩子。图9.6（d）给出的示例是直接将被删结点的右子树代替被删结点。

```
/* **** */
```

```
/* 基于二叉排序树的结点删除算法 */
```

```
/* 文件名: t_dele.c 函数名: delbstree() */
```

```
/* **** */
```

```
bstree delbstree(bstree t,datatype x)
```

```
{ /*在二叉排序树t中删除结点值为x的结点*/
```

```
    bstree p,q,child;
```

```
    bssearch1(t,x,&p,&q); /*查找被删结点*/
```

```
    if (q) /*找到了待删除结点*/
```

```
    { if (q->lchild==NULL && q->rchild==NULL) /*情况1，待删  
    结点为叶结点*/
```

```
        { if (p) /*待删除结点有双亲*/
```

```
{ if (p->lchild==q) p->lchild=NULL; else p->rchild=NULL;}
```

```
    else t=NULL; /*待删结点为树根*/
```

```
    free (q);
```

```
}
```

else /*情况2，待删结点的左子树为空，用待删结点的右子树替代该结点*/

```
if (q->lchild==NULL)
```

```
{ if (p) /*待删结点的双亲结点不为空*/
```

```
{ if (p->lchild==q)
```

```
    p->lchild=q->rchild; /*q是其双亲结点的左儿子*/
```

```
    else p->rchild=q->rchild; /*q是其双亲结点的右儿子*/
```

```
}
```



```
else t=q->rchild;  
    free(q);  
}
```

else /*情况3，待删结点的右子树为空，用待删结点的左子树替代该结点*/

```
if (q->rchild==NULL)
```

```
{ if (p) /*待删结点的双亲结点不为空*/
```

```
{if (p->lchild==q)
```

```
    p->lchild=q->lchild; /*q是其双亲结点的左  
    儿子*/
```

```
    else p->lchild=q->lchild; /*q是其双亲结点的  
    右儿子*/
```

```
} else t=q->lchild;
```

```
free(q);
```

```
}
```

else /*情况4, 待删结点的左右子树均不为空, 用右子树代替待删结点, 同时将待删结点的左子树收为右子树中序首点的左儿子*/

```
{ child=q->rchild;
```

```
while (child->lchild) /*找待删结点右子树中的中序首点*/
```

```
child=child->lchild;
```

```
child->lchild=q->lchild; /*将待删结点的左子树收为child的左孩子*/
```

```
if (p) /*待删结点不是树根*/
```

```
{ if (p->lchild==q)
```

```
        p->lchild=q->rchild;
        else p->rchild=q->rchild;
    } else t=q->rchild; /*待删结点为树根*/
    free(q);
}
}
return t;
}
```

算法9.7 基于二叉排序树的结点删除

二叉排序树中结点的删除操作的主要时间在于查找被删除结点及查找被删结点的右子树的中序首点上，而这个操作的时间花费与树的深度密切相关。因此，删除操作的平均时间亦为 $O(\log_2 n)$ 。

习题

9.2 设有关键码A、B、C和D，按照不同的输入顺序，共可能组成多少不同的二叉排序树。请画出其中高度较小的6种。

9.3 已知序列17，31，13，11，20，35，25，8，4，11，24，40，27。请画出由该输入序列构成的二叉排序树，并分别给出下列操作后的二叉排序树。

（1）插入数据9； （2）删除结点17； （3）再删除结点13

9.4 试写一算法判别给定的二叉树是否为二叉排序树，设此二叉树以二叉链表为存储结构，且树中结点的关键字均不相同。

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
 - B-树
 - 散列表检索

9.4 丰满树和平衡树

二叉排序树上实现的插入、删除和查找等基本操作的平均时间虽然为 $O(\log_2 n)$ ，但在最坏情况下，二叉排序树退化成一个具有单个分支的单链表，此时树高增至 n ，这将使这些操作的时间相应地增至 $O(n)$ 。为了避免这种情况发生，人们研究了许多种动态平衡的方法，包括如何建立一棵“好”的二叉排序树；如何保证往树中插入或删除结点时保持树的“平衡”，使之既保持二叉排序树的性质又保证树的高度尽可能地为 $O(\log_2 n)$ 。本节将介绍两种特殊的二叉树：丰满树和平衡树。

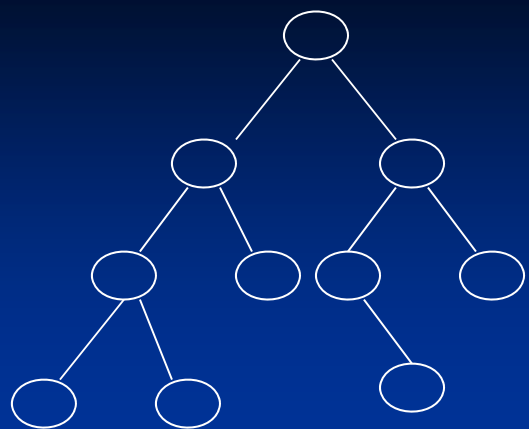
9.4.1 丰满树

设 T 是一棵二叉树， k_i 和 k_j 是 T 中孩子结点个数少于2的任意两个结点， λk 是结点 k 的树枝长度，如果满足：

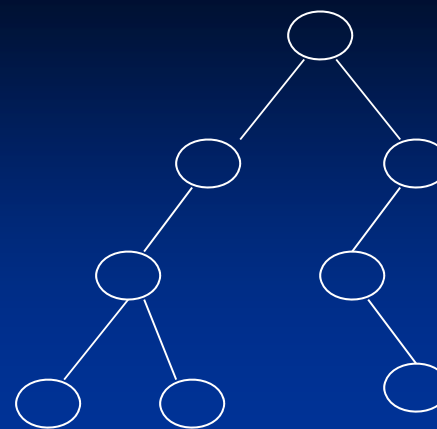
$$|\lambda k_i - \lambda k_j|$$

则称 T 是一棵丰满树。

从上述概念可知，在丰满树中，任意两个非双孩子结点的高度之差的绝对值要小于等于1，由于树的最下面一层为叶子结点，因此，在丰满树中，子女结点个数少于2的结点只出现在树的最低两层之中。图9.7给出了一棵丰满树和一棵非丰满树。



(a) 一棵丰满树



(b) 一棵非丰满树

图9.7 丰满树与非丰满树示例

对于 n 个结点的任意序列，用平分法构造结点序列的丰满树的步骤是：

(1) 如果结点序列为空，则得到一棵空的二叉树；

(2) 如果序列中有 $n+1$ 个结点 k_1, k_2, \dots, k_n ，那么令 $m = \lfloor (n+1) / 2 \rfloor$ ，所求的树是由根 k_m ，左子树 T_l 和右子树 T_r 组成。其中 T_l 和 T_r 分别是用平分法由 k_1, k_2, \dots, k_{m-1} 和 $k_{m+1}, k_{m+2}, \dots, k_n$ 创建的丰满树。

要用平分法构造丰满的二叉排序树，需保证 n 个序列 k_1, k_2, \dots, k_n 是按升序排列的。根据有序数组创建丰满二叉排序树的算法见算法9.8。

```
/******  
/*      丰满树构造算法      */  
/*  文件名: creatfmt.c  函数名: creatfmt()  */  
/******  
  
#include<stdio.h>  
  
#include<stdlib.h>  
  
typedef int datatype;  
  
typedef struct node      /*二叉树结点定义*/  
{  datatype data;  
    struct node *lchild,*rchild;  
} bintnode;  
  
typedef bintnode *bintree;
```

```
/*-----平分法创建一棵丰满树-----*/
```

```
bintree creatfmt(int node[],int low,int high)
```

```
{ int mid; bintree s;
```

```
  if (low<=high)
```

```
  { mid=(low+high)/2;
```

```
    s=(bintree)malloc(sizeof(bintnode)); /*生成一个新结点*/
```

```
    s->data=node[mid];
```

```
    s->lchild=creatfmt(node,low,mid-1); /*平分法建左子树*/
```

```
    s->rchild=creatfmt(node,mid+1,high); /*平分法建右子树*/
```

```
    return s;
```

```
  } else  return NULL;  }
```

算法9.8 建立丰满树

对于具有n个结点的丰满二叉排序树，如果树中所有结点都具有相同的使用概率，那么其平均检索长度为：

$$ASL \approx \log_2 n$$

但对动态的二叉排序树进行插入和删除等操作后，丰满树很容易变为非丰满二叉排序树，并且将非丰满二叉排序树改造成丰满二叉排序树非常困难。因此，实际应用中经常使用一种称为“平衡树”的特殊二叉排序树。

9.4.2 平衡二叉排序树

平衡二叉树又称为**AVL**树，它或是一棵空树，或是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过1。

此处规定二叉树的高度是二叉树的树叶的最大层数，也就是从根结点到树叶的最大路径长度，空的二叉树的高度定义为-1。

相应地，二叉树中某个结点的左子树高度与右子树高度之差称为该结点的平衡因子（或平衡度）。由此可知，**平衡二叉树**也就是树中任意结点的平衡因子的绝对值小于等于1的二叉树。

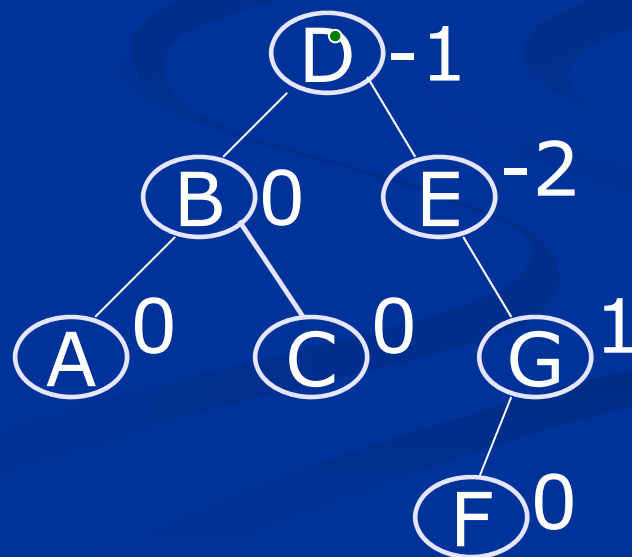
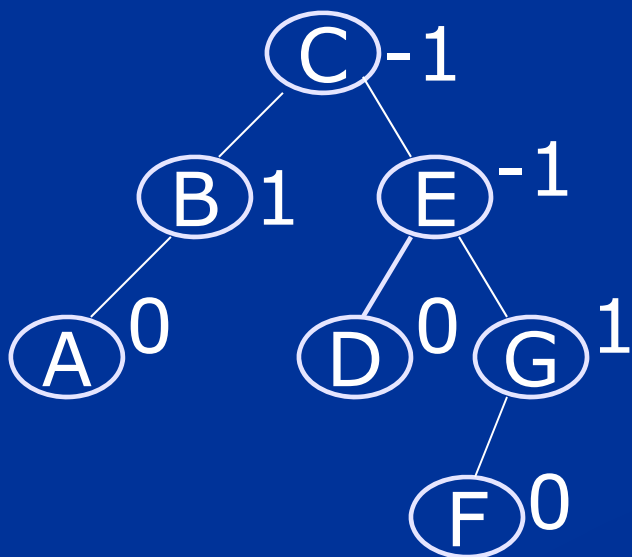
如果一棵二叉排序树满足平衡二叉树的定义便是一棵平衡的二叉排序树，平衡的二叉排序树又称为平衡查找树。

研究平衡二叉树的目的在于如何动态地使一棵二叉排序树保持平衡

平衡非满树

非平衡二叉树

由丰满树和平衡树的定义可知，丰满树一定是平衡树，但平衡树却不一定是丰满树。



G.M.Adelson-Velskii和E.M.Landis在1962年提出了动态保持二叉排序树平衡的一个有效办法，后称为Adelson方法。下面介绍Adelson方法如何将一个新结点 k 插入到一棵平衡二叉排序树 T 中去。

Adelson方法由三个依次执行的过程——插入、调整平衡度和改组所组成：

(1) 插入：不考虑结点的平衡度，使用在二叉排序树中插入新结点的方法，把结点 k 插入树中，同时置新结点的平衡度为0。

(2) 调整平衡度：假设 $k_0, k_1, \dots, k_m=k$ 是从根 k_0 到插入点 k 路径上的结点，由于插入了结点 k ，就需要对这条路径上的结点的平衡度进行调整。

调整方法是：从结点 k 开始，沿着树根的方向进行扫描，当首次发现某个结点 k_j 的平衡度不为零，或者 k_j 为根结点时，便对 k_j 与 k_{m-1} 之间结点进行调整。令调整的结点为 k_i ($j \leq i \leq m$)，若 k 在 k_i 的左子树中，则 k_i 的平衡度加1；若 k 在 k_i 的右子树中，则 k_i 的平衡度减1；此时， k_{j+1} ， k_{j+2} ， \dots ， k_{m-1} 结点不会失去平衡，唯一可能失去平衡的结点是 k_j 。若 k_j 失去平衡，即 k_j 的平衡因子不是-1，0和1时，便对以 k_j 为根的子树进行改组，且保证改组以后以 k_j 为根的子树与未插入结点 k 之前的子树高度相同，这样， k_0 ， k_1 ， \dots ， k_{j-1} 的平衡度将保持不变，这就是为何不需要对这些结点进行平衡度调整的原因。反之，若 k_j 不失去平衡，则说明，新结点 k 的加入并未改变以 k_j 为根的子树的高度，整棵树无需进行改组。

(3) 改组：改组以 k_j 为根的子树除了满足新子树高度要和原来以 k_j 为根子树的高度相同外，还需使改造后的子树是一棵平衡二叉排序树。

下面具体讨论AVL树上因插入新结点而导致失去平衡时的调整方法。

为叙述方便，假设在AVL树上因插入新结点而失去平衡的最小子树的根结点为A（即A为距离插入结点最近的，平衡因子不是-1、0和1的结点）。失去平衡后的调整操作可依据失去平衡的原因归纳为下列四种情况分别进行。

(1) LL型平衡旋转：由于在A的左孩子的左子树上插入新结点，使A的平衡度由1增至2，致使以A为根的子树失去平衡，如图9.9 (a) 所示。此时应进行一次顺时针旋转，“提升”B（即A的左孩子）为新子树的根结点，A下降为B的右孩子，同时将B原来的右子树 B_r 调整为A的左子树。

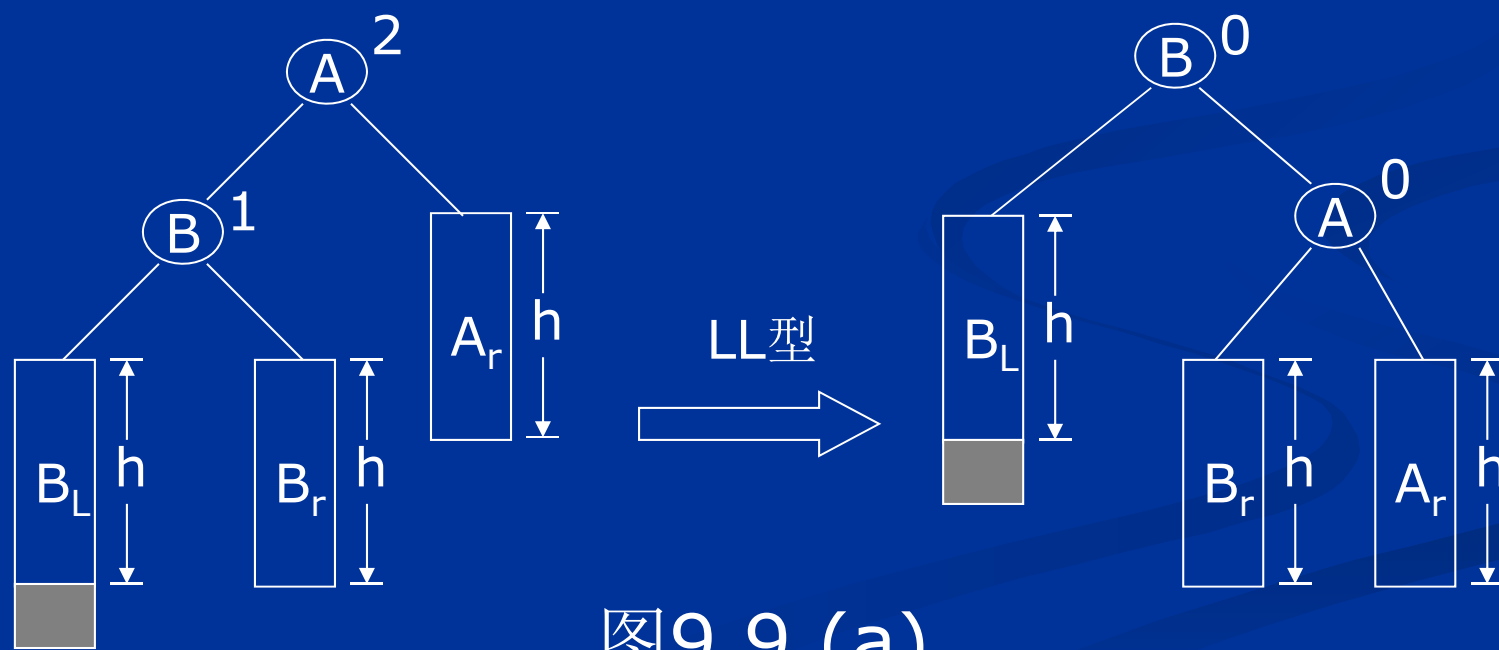


图9.9 (a)

(2) **RR型平衡旋转**：由于在**A**的右孩子的右子树上插入新结点，使**A**的平衡度由-1变为-2，致使以**A**为根的子树失去平衡，如图9.9 (b) 所示。此时应进行一次逆时针旋转，“提升”**B**（即**A**的右孩子）为新子树的根结点，**A**下降为**B**的左孩子，同时将**B**原来的左子树**B_L**调整为**A**的右子树。

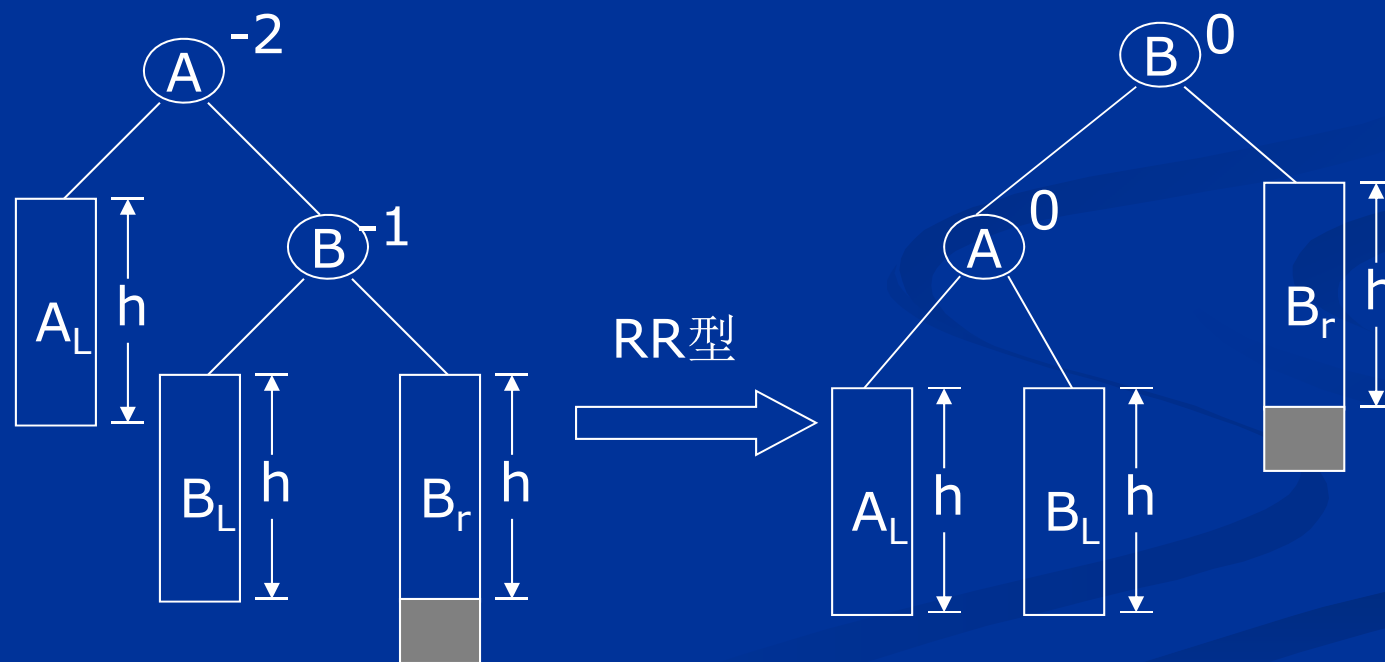


图9.9 (b)

(3) **LR型平衡旋转**：由于在**A**的左孩子的右子树上插入新结点，使**A**的平衡度由**1**变成**2**，致使以**A**为根的子树失去平衡，如图9.9 (c) 所示。此时应进行两次旋转操作（先逆时针，后顺时针），即“提升”**C**（即**A**的左孩子的右孩子）为新子树的根结点；**A**下降为**C**的右孩子；**B**变为**C**的左孩子；**C**原来的左子树**C_L**调整为**B**现在的右子树；**C**原来的右子树**C_r**调整为**A**现在的左子树

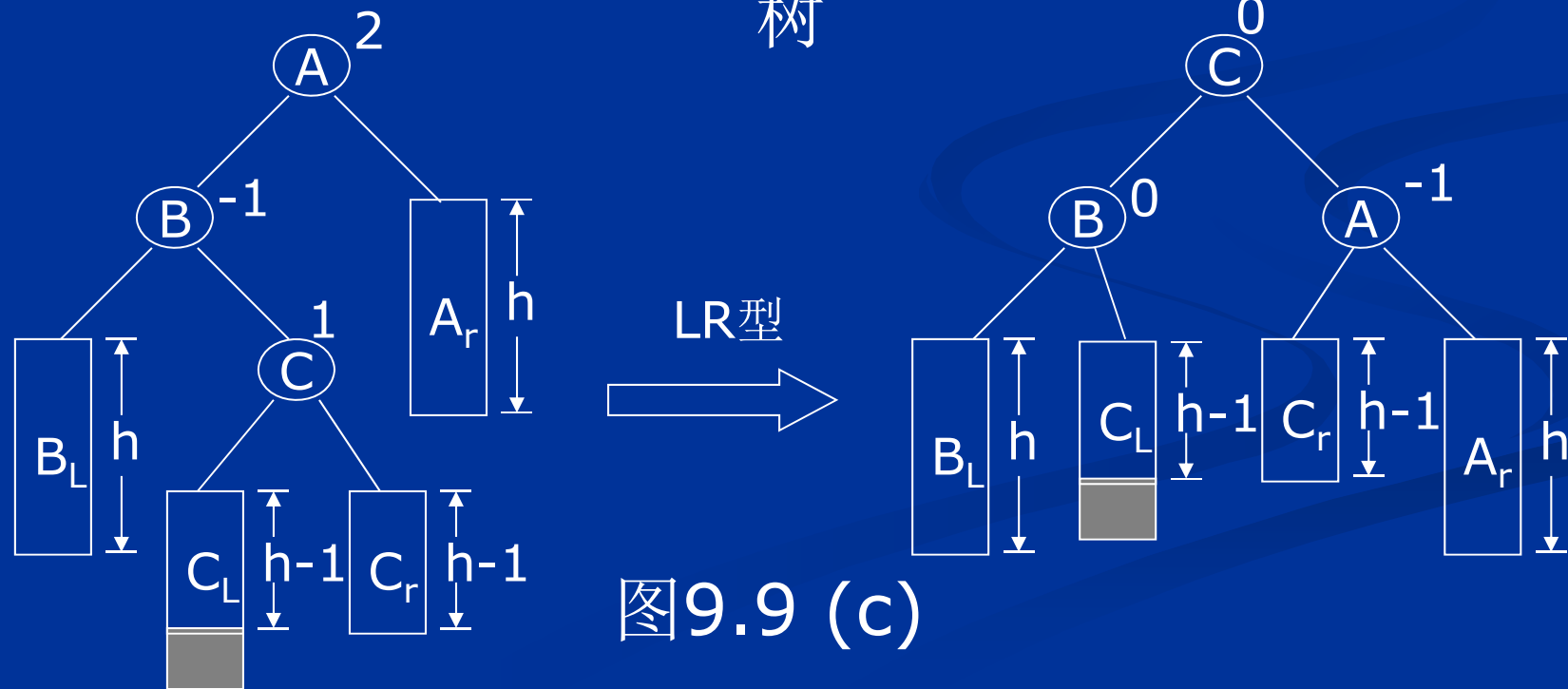
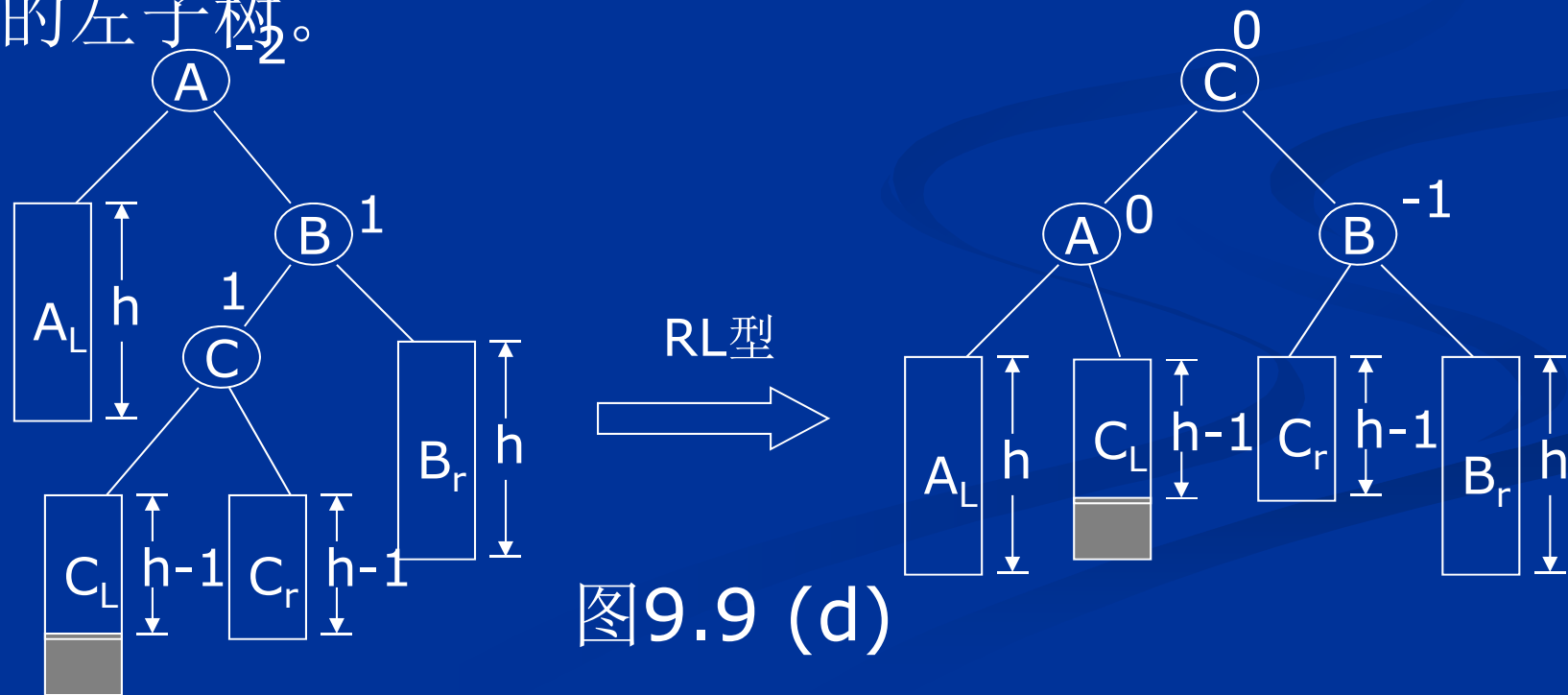


图9.9 (c)

(4) **RL型平衡旋转**：由于在A的右孩子的左子树上插入新结点，使A的平衡度由-1变成-2，致使以A为根的子树失去平衡，如图9.9 (d) 所示。此时应进行两旋转操作（先顺时针，后逆时针），即“提升”C（即A的右孩子的左孩子）为新子树的根结点；A下降C的左孩子；B变为C的右孩子；C原来的左子树 C_L 调整为A现在的右子树；C原来的右子树 C_r 调整为B现在的左子树。



综上所述，在平衡的二叉排序树 t 上插入一个新的数据元素 x 的算法可描述如下：

（一）若AVL树 t 为空树，则插入一个数据元素为 x 的新结点作为 t 的根结点，树的深度增1；

（二）若 x 的关键字和AVL树 t 的根结点的关键字相等，则不进行插入；

（三）若 x 的关键字小于AVL树 t 的根结点的关键字，则将 x 插入在该树的左子树上，并且当插入之后的左子树深度增加1时，分别就下列不同情况进行分情形处理：

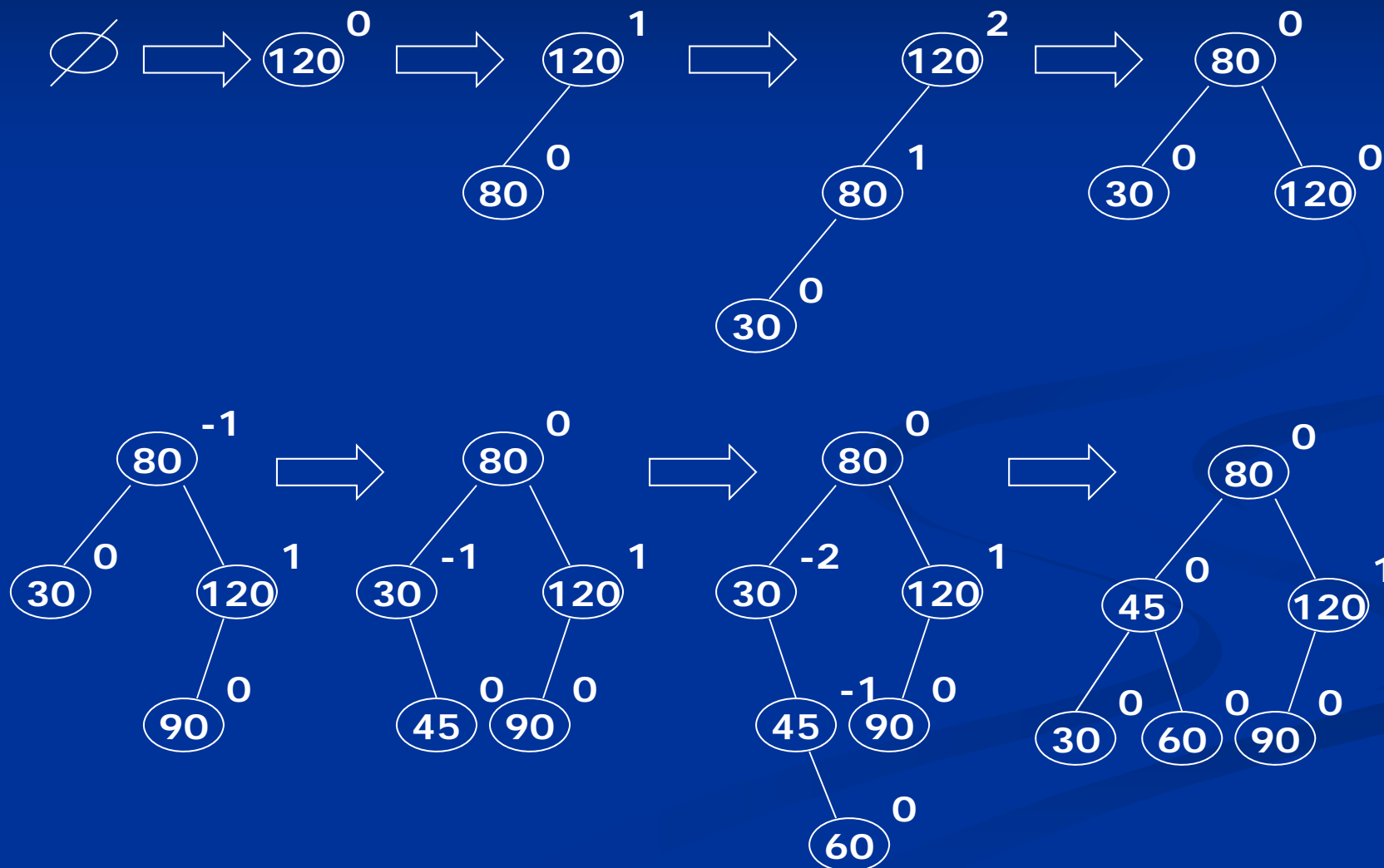
（1）若AVL树的根结点的平衡因子为-1（右子树的深度大于左子树的深度），则将根结点的平衡因子调整为0，并且树的深度不变；

(2) 若AVL树的根结点的平衡因子为0（左、右子树的深度相等），则将根结点的平衡因子调整为1，树的深度同时增1；

(3) 若AVL树的根结点的平衡因子为1（左子树的深度大于右子树的深度），则当该树的左子树的根结点的平衡因子为1时需进行LL型平衡旋转；当该树的左子树的根结点的平衡因子为-1时需进行LR型平衡旋转。

(四) 若x的关键字大于AVL树t的根结点的关键字，则将x插入在该树的右子树上，并且当插入之后的右子树深度增加1时，需要分别就不同情况进行处理。其处理操作和（三）中所述相对称，读者可以自行分析。

结点序列 (120, 80, 30, 90, 45, 60) 逐个插入一棵空的AVL树的过程如下:



为实现Adelson方法，先定义AVL树的存储结构如下：

```
/*  
*****  
/*    AVL树使用的头文件    */  
/*    文件名:AVL.H          */  
/*  
*****  
  
typedef int datatype;  
typedef struct node  
{ datatype key;  
  struct node *lchild,*rchild;  
  int bal;      /*结点的平衡度*/  
} bsnode;  
  
typedef bsnode *bstree;
```



算法9.9 给出了
基于AVL树的
结点插入算法

习题

9.6 含有12个节点的平衡二叉树的最大深度是多少（设根结点深度为0），并画出一棵这样的树。

9.7 试用Adelson插入方法依次把结点值为60, 40, 30, 150, 130, 50, 90, 80, 96, 25的记录插入到初始为空的平衡二叉排序树中，使得在每次插入后保持该树仍然是平衡查找树。请依次画出每次插入后所形成的平衡查找树。

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
 - B-树
 - 散列表检索

9.5最佳二叉排序树和Huffman树

对于具有 n 个结点的二叉排序树，在考虑每个结点查找概率的情况下，如何使得整棵树的平均检索效率最高，即使得比较的平均次数最少、代价最小，这与二叉排序树的形状密切相关。假定 n 个结点的关键字序列为 k_1, k_2, \dots, k_n ， k_i 对应的使用概率为 $P(k_i)$ ，树枝长度为 λk_i ，则对 n 个结点构成的二叉排序树其查找成功的平均比较次数为：

$$ASL = \sum_{i=1}^n p(k_i)(1 + \lambda k_i) \quad (9-9)$$

当每个结点具有相同的使用概率，即 $P(k_i) = 1/n$ 时，

$$ASL = \sum_{i=1}^n \frac{1}{n} (1 + \lambda k_i) = \frac{1}{n} \sum_{i=1}^n (1 + \lambda k_i) \quad (9-10)$$

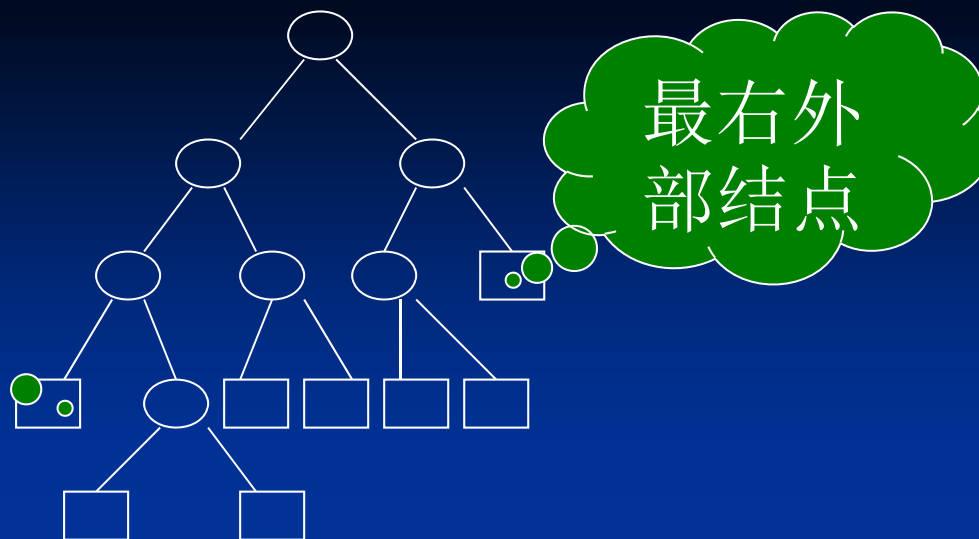
本节讨论如何构造ASL最小的二叉排序树。

9.5.1 扩充二叉树

给定一棵二叉树，对树中不足两个孩子的结点（包括叶子结点）都添上附加结点，使每个结点都有两个孩子结点，所得的二叉树称为原二叉树的扩充二叉树，称那些附加的结点为**外部结点**，称树中原来的结点为**内部结点**。对于具有n个内部结点的二叉树，其外部结点数为n+1个。图9.11给出了一棵二叉树及其对应的扩充二叉树，图中圆圈表示内部结点，方框表示外部结点。



(a) 一棵二叉树t



(b) t的扩充二叉树

对一棵二叉排序树进行扩充后便可得到一棵扩充的二叉排序树（又称为扩充查找树）。对扩充查找树进行中序遍历，**最左外部结点**代表码值小于内部结点最小码值的可能结点集；**最右外部结点**代表码值大于内部结点最大码值的可能结点集；而其余外部结点代表码值处于原二叉排序树在中序序列下相邻结点码值之间的可能结点集。

在对扩充二叉查找树进行检索过程中，若检索到达外部结点，则表明相应码值的结点不在原二叉排序树中，故也称外部结点为失败结点。

若一棵扩充二叉查找树中所有内部结点的树枝长度之和记为 I ，所有外部结点的树枝长度之和记为 E 。则对于一个具有 n 个内部结点的扩充二叉树，其内部树枝长度 I_n 与外部树枝长度 E_n 存在下列关系：

$$E_n = I_n + 2n \quad (9-11)$$

根据式（9-11）可知，在具有 n 个结点的所有二叉树中，具有最大（或最小）内部路枝长度的二叉树也一定是具有最大（或最小）外部路枝长度的二叉树，反之亦然。

当二叉树退化为线性表时，其内部路枝长度最大，其值为 $I_n=0+1+2+\dots+(n-1)=n(n-1)/2$ 。为了得到具有最小 I_n 的二叉树，必须使内部结点尽量靠近根结点。由二叉树结构可知，根结点只有一个，路枝长度为1的结点至多有两个，路枝长度为2的结点至多为四个，路枝长度为 k 的结点至多为 2^k 个。所以 I_n 的最小值为：

$$I_n=1 \times 0 + 2 \times 1 + 4 \times 2 + 8 \times 3 + \dots = \sum_{i=1}^n \lfloor \log_2 i \rfloor$$

9.4.1节介绍的平分法构造的丰满树具有最小内部路径长度。

9.5.2最佳二叉排序树

在一棵扩充的二叉排序树中，假设内部结点由序列 $(a_1, a_2, a_3, \dots, a_i, \dots, a_n)$ 构成，且有 $a_1 < a_2 < a_3 \dots < a_i \dots < a_n$ 。外部结点由 $(b_0, b_1, b_2, \dots, b_i, \dots, b_n)$ 构成，且有 $b_0 < b_1 < b_2 \dots < b_i \dots < b_n$ 。这里将内部结点 a_i 的检索概率记为 $p(a_i)$ ，简记为 p_i ($1 \leq i \leq n$)；外部结点 b_i 的检索概率记为 q_i ($0 \leq i \leq n$)，它们对应的树枝长度分别记为 a_i ($1 \leq i \leq n$) 和 b_i ($0 \leq i \leq n$)，则成功查找所需的平均比较次数为：

$$\sum_{i=1}^n p_i (a_i \text{的树枝长度} + 1) = \sum_{i=1}^n p_i (\lambda a_i + 1)$$

而不成功查找所需的平均比较次数为：

$$\sum_{i=0}^n q_i (b_i \text{的树枝长度}) = \sum_{i=0}^n q_i (\lambda b_i)$$

不失一般性，一棵二叉排序树的平均查找长度（代价）为：

$$ASL = \sum_{i=1}^n (p_i (\lambda a_i + 1)) + \sum_{j=0}^n (q_j (\lambda b_j)) \quad (9-18)$$

对于 n 个有序序列 ($a_1, a_2, a_3, \dots, a_i, \dots, a_n$) 作为内部结点和 $n+1$ 个有序序列 ($b_0, b_1, b_2, \dots, b_i, \dots, b_n$) 作为外部结点构成的所有可能的扩充查找树中, 具有最少平均比较次数, 即式 (9-18) 取值最小的扩充二叉排序树称为最佳二叉排序树, 或最优查找树。

当查找成功与查找失败具有相等概率, 即 $p_1=p_2=\dots=p_n=q_0=q_1=\dots=1/(2n+1)$ 时, 平均检索

$$\text{长度ASL} = \sum_{i=1}^n (p_i(\lambda a_i + 1)) + \sum_{j=0}^n (q_j(\lambda b_j))$$

$$= \frac{1}{2n+1} \left(\sum_{i=1}^n (\lambda a_i + 1) + \sum_{j=0}^n (\lambda b_j) \right)$$

$$\begin{aligned}
&= \frac{1}{2n+1}(I_n + n + E_n) \\
&= \frac{1}{2n+1}(I_n + n + I_n + 2n) \\
&= \frac{1}{2n+1}(2I_n + 3n) \quad \dots\dots (9-19)
\end{aligned}$$

由式（9-19）可知，只要 I_n 取最小值，ASL就达到最小。

而平分法构造的丰满树具有最小内部路径长度。

$$I_n = \sum_{i=1}^n \lfloor \log_2 i \rfloor \leq n(\log_2^n) = O(n \log_2^n)$$

此时

$$ASL = \frac{1}{2n+1} (2I_n + 3n) = O(n \log_2^n)$$

所以，在检索成功和不成功具有相等概率的情况下，用平分法构造出来的丰满二叉排序树是最佳二叉排序树。

现考虑查找概率不相等的情况下如何构造最佳二叉排序树。也就是对于给定的 n 个内部结点 $a_1, a_2, a_3, \dots, a_i, \dots, a_n$ 和 $n+1$ 个外部结点 $b_0, b_1, b_2, \dots, b_i, \dots, b_n$ ，它们对应的查找概率分别是 $p_1, p_2, p_3, \dots, p_i, \dots, p_n$ 及 q_0, q_1, \dots, q_n ，找使得

$$ASL = \sum_{i=1}^n (p_i(\lambda a_i + 1)) + \sum_{j=0}^n (q_j(\lambda b_j))$$

为最小的二叉排序树（以下我们又称ASL为二叉排序树的代价（或花费））。

构造最佳二叉排序树时要满足以下两个要求：同一序列构造的不同二叉排序树应具有相同的中序遍历结果；一棵最佳二叉排序树的任何子树都是最佳二叉排序树。因此，构造一棵最佳二叉排序树可以先构造包括一个结点的最佳二叉排序树，再根据包括一个结点的最佳二叉排序树构造包括两个结点的最佳二叉排序树，...，如此进行下去，直到把所有的结点都包括进去。

为描述构造最佳二叉排序树的算法，这里用 $T[i, j]$ 表示 a_{i+1}, \dots, a_j ($i < j$) 组成的一棵最佳二叉排序树，规定当 $0 \leq i \leq n$ 且 $i = j$ 时， $T[i, j]$ 为空树（即内部结点为空）；当 $i > j$ 时表示 $T[i, j]$ 无定义；用 $C[i, j]$ 表示查找树 $T[i, j]$ 的代价；用 r_{ij} 表示 $T[i, j]$ 的根，用 $W[i, j]$ 表示 $T[i, j]$ 的权值，它的值是 $T[i, j]$ 中内部结点和外部结点查找概率之和。即：

$$W[i, j] = q_i + \sum_{k=i+1}^j (p_k + q_k)$$

根据最佳二叉排序树的构造要求，其构造过程可以按以下步骤进行：

- (1) 构造包括一个结点的最佳二叉排序树，也就是 $T[0, 1]$, $T[1, 2]$, ..., $T[n-1, n]$ 。
- (2) 构造包括两个结点的最佳二叉排序树，也就是 $T[0, 2]$, $T[1, 3]$, ..., $T[n-2, n]$ 。
- (3) 再构造包括三个，四个，..., $n-1$ 个结点的最佳二叉排序树，直到最后构造 $T[0, n]$ 。

用 $(a_{i+1}, a_{i+2}, \dots, a_j)$ 作为内部结点构造最佳二叉排序树 $T[i, j]$ 的方法可如下进行：分别用 $a_{i+1}, a_{i+2}, \dots, a_j$ 为根，共考虑 $j-i$ 棵二叉排序树，以 a_k 为根的二叉排序树其左子树包括 a_{i+1}, \dots, a_{k-1} ，而包括这些关键码为内部结点的最佳二叉排序树 $T[i, k-1]$ 已在前面的步骤确定， $C[i, k-1]$ 已求出，而以 a_k 为根的二叉排序树其右子树包括 $a_{k+1}, a_{k+2}, \dots, a_j$ ，以这些关键码为内部结点的最佳二叉排序树 $T[k, j]$ 也已在前面的步骤确定， $C[k, j]$ 已求出。对于 $i < k \leq j$ ，找出使 $C[i, k-1] + C[k, j]$ 最小的那个 k' ，以 $a_{k'}$ 为根， $T[i, k'-1]$ 为左子树， $T[k', j]$ 为右子树的那棵二叉排序树就是所求的 $T[i, j]$ 。其花费 $C[i, j]$ 等于其根的左子树花费 $C[i, k'-1]$ 加上右子树花费 $C[k', j]$ ，再加上结点总的权 $W[i, j]$ ，即 $C[i, j] = W[i, j] + C[i, k'-1] + C[k', j]$ 。

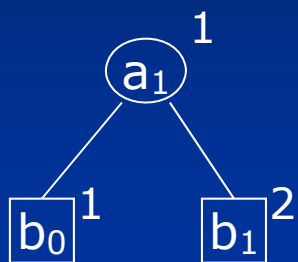
综上所述， $T[i, j]$ 是最优查找树必须满足条件：

$$C[i, j] = W[i, j] + \min_{i < k \leq j} (C[i, k-1] + C[k, j]) \quad (9-20)$$

例

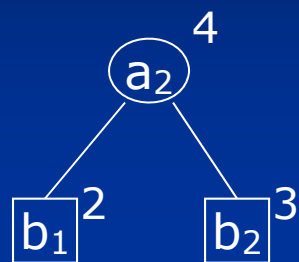
假设 $n=4$ ，且 $(a_1, a_2, a_3, a_4) = (10, 18, 26, 50)$ ； $(p_1, p_2, p_3, p_4) = (1, 4, 3, 2)$ ， $(q_0, q_1, q_2, q_3, q_4) = (1, 2, 3, 3, 1)$ ，试构造出有序序列 a_1, a_2, a_3, a_4 所组成的最优查找树。

首先根据式(9-20)构造包括一个内部结点的最佳二叉排序树，其花费的代价分别是 $C[0, 1]=4$ ， $C[1, 2]=9$ ， $C[2, 3]=9$ 和 $C[3, 4]=6$ ，如图9.12(a)所示。



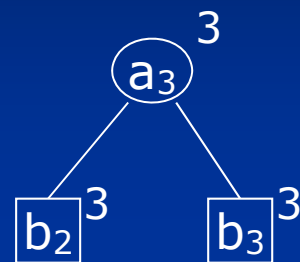
花费 $C[0, 1]=\underline{4}$

总权 4



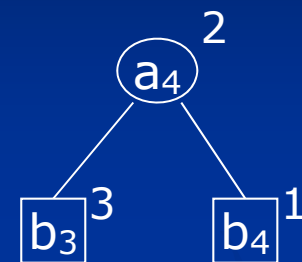
$C[1, 2]=\underline{9}$

9



$C[2, 3]=\underline{9}$

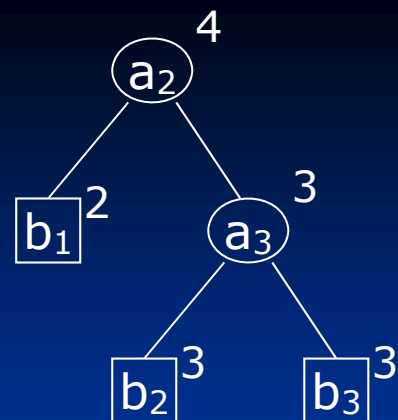
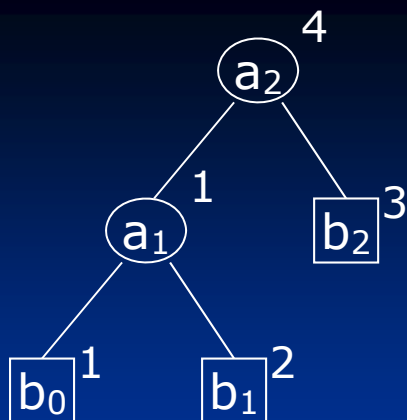
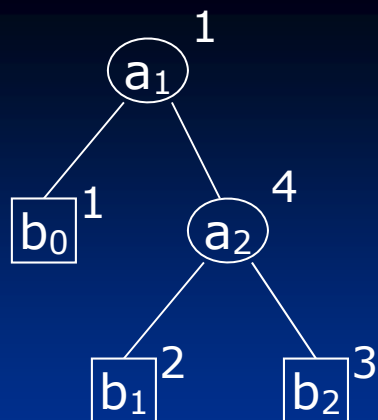
9



$C[3, 4]=\underline{6}$

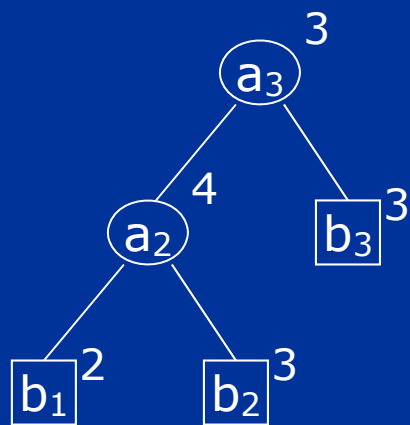
6

(a) 包括一个内部结点的最佳二叉排序树

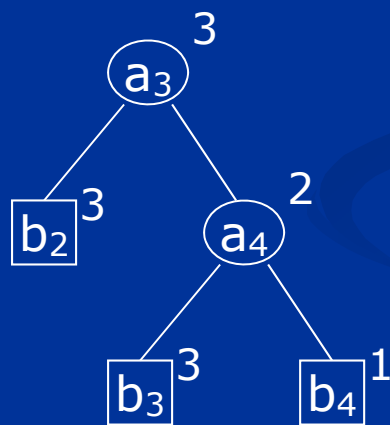


花费 20 $C[0, 2]=C[0, 1]+11=\underline{15}$ $C[1, 3]=C[2, 3]+15=\underline{24}$

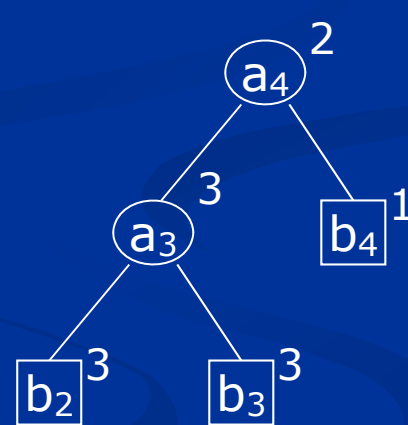
总权 11



11



15



花费 24

$C[2, 4]=C[3, 4]+12=\underline{18}$

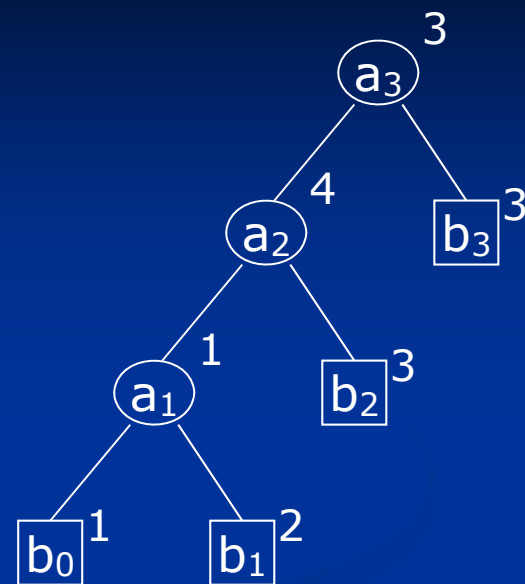
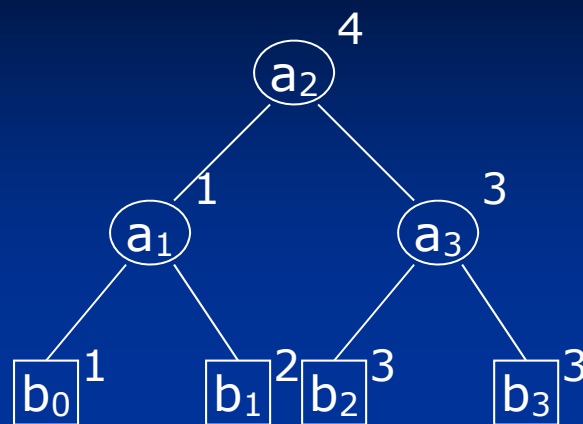
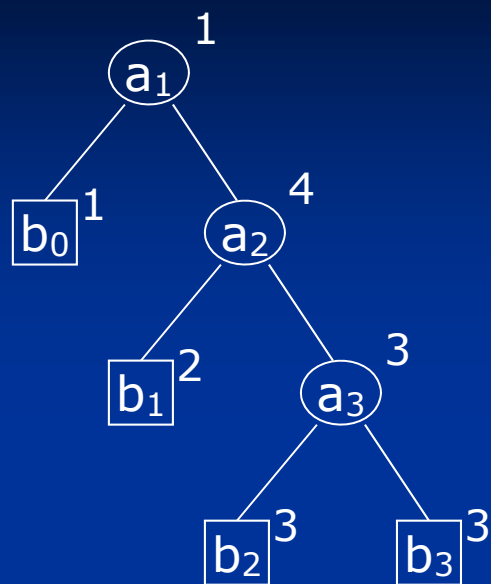
21

总权 15

12

12

(b) 包括二个内部结点的最佳二叉排序树



花费 41

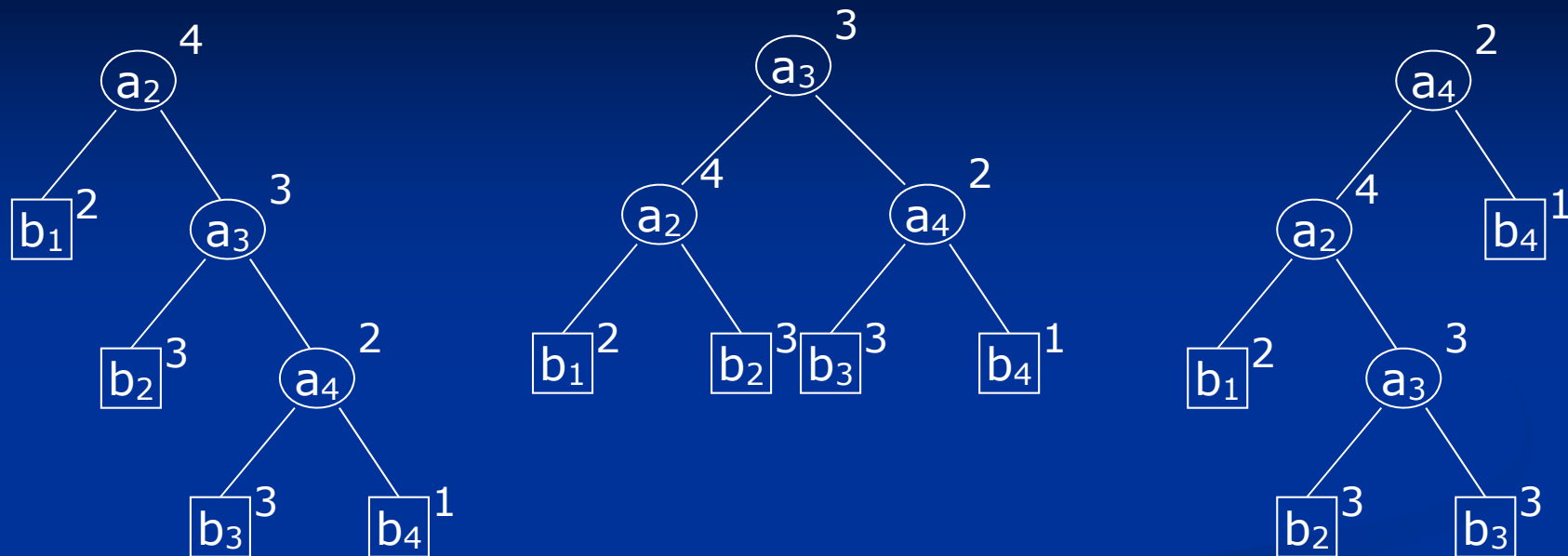
$C[0, 3] = C[0, 1] + C[2, 3] + 17 = \underline{30}$

32

总权 17

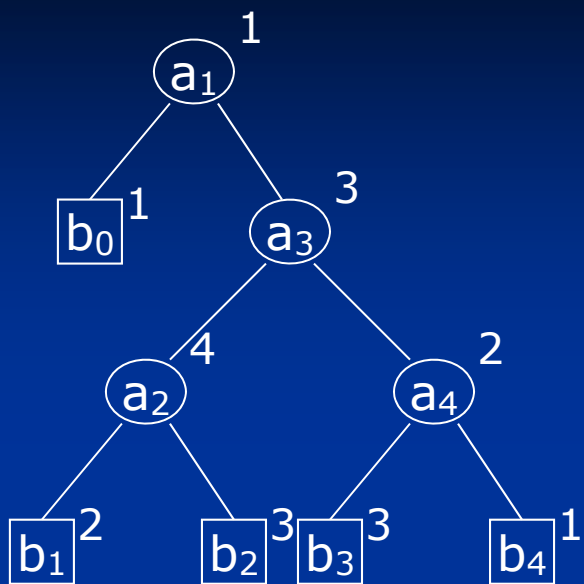
17

17

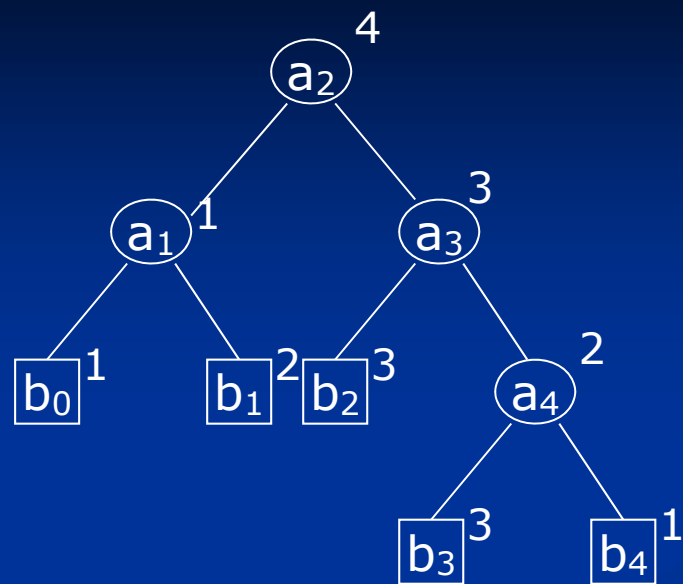


花费	36	$C[1, 4]=C[1, 2]+C[3, 4]+18=\underline{33}$	42
总权	18	18	18

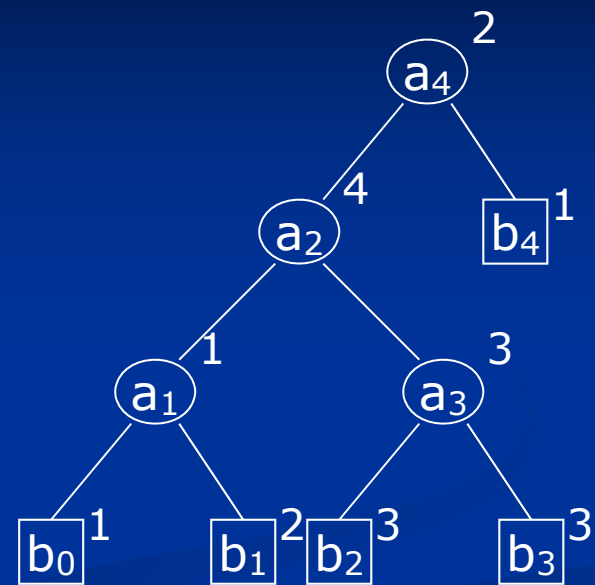
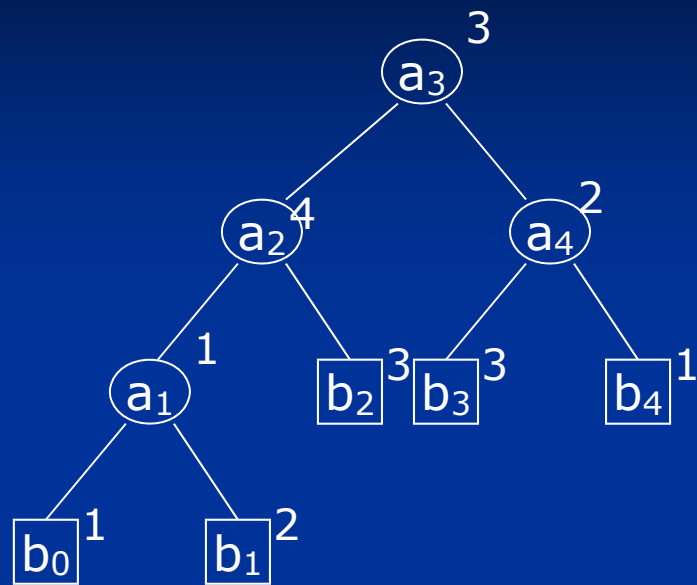
(c) 包括三个内部结点的最佳二叉排序树



花费	53
总权	20



42
20



花费 $C[0, 4] = C[0, 2] + C[3, 4] + 20 = \underline{41}$

50

总权 20

20

(d) 包括四个内部结点的最佳二叉排序树

习题

9.8 结点关键字 k_1, k_2, k_3, k_4, k_5 为一个有序序列，它们的相对使用频率分别为 $p_1=6, p_2=8, p_3=12, p_4=2, p_5=16$ ，外部结点的相对使用频率分别为 $q_0=4, q_1=9, q_2=8, q_3=12, q_4=3, q_5=2$ 。试构造出有序序列 k_1, k_2, k_3, k_4, k_5 所组成的最优查找树。

对于包括n个关键码的集合，构造最佳二叉排序树过程中需要进行C[i, j] (0≤i<j≤n) 的计算次数为：

$$\sum_{j=1}^n \sum_{i=0}^{j-1} (j-i+1) \approx \frac{n^3}{6} = O(n^3)$$

9.5.3 Huffman树

给定n个结点 k_1, k_2, \dots, k_n ，它们的权分别是 $W(k_i)$ ($1 \leq i \leq n$)，现要利用这n个结点作为外部结点（叶子结点）去构造一棵扩充二叉树，使得带权外部路径长度

$$WPL = \sum_{i=1}^n w_{k_i} \cdot (\lambda k_i)$$

达到最小值，其中 w_{k_i} 为外部结点 k_i 的权值， λk_i 是从根结点到达外部结点 k_i 的树枝长度。具有最小带权外部路径长度的二叉树称为Huffman树。

例如4个结点 k_1, k_2, k_3, k_4 ，分别带权10, 16, 20, 6。利用它们作为外部结点分别构造了以下三棵扩充二叉树（还有其它形式的扩充二叉树）如图9.13所示。

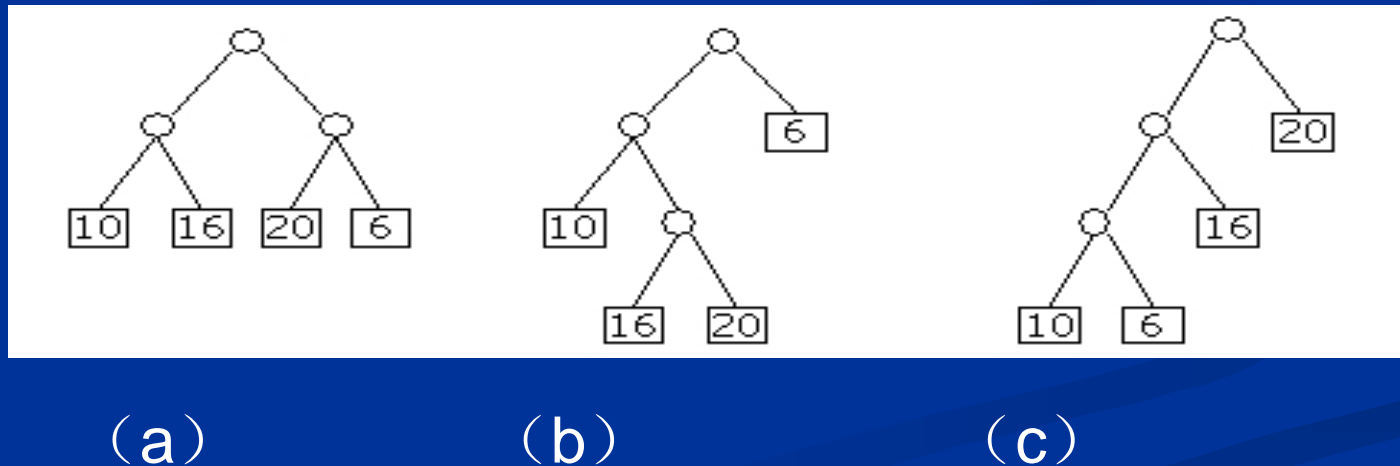


图9.13 具有不同带权路径长度的扩充二叉树

其带权路径长度分别为：

(a) $WPL=52\times 2=104$

(b) $WPL=(16+20)\times 3+20\times 2+6\times 1=154$

(c) $WPL=(10+6)\times 3+16\times 2+20\times 1=100$

其中图9.13 (c) 所示的扩充二叉树带权外部路径长度最小，可以验证，它恰为Huffman树。一般情况下，权越大的叶子离根越近，那么二叉树的带权外部路径长度就越小。在实际的应用中，分支程序的判断流程可用一棵Huffman树来表示，如果出现概率越大的分枝（条件语句）离根越近，那么所需执行的判断语句就越少，这样便可提高程序的执行效率。

Huffman给出了求具有最小带权外部路径长度的扩充二叉树的方法，通常称为Huffman算法，该算法可描述如下：

(1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构造 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 中只有一个带权为 w_i 的根结点，其左、右子树均为空。

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

(3) 在 F 中用新得到的二叉树代替这两棵树。

(4) 重复(2)、(3)过程，直到 F 中只含有一棵树为止。

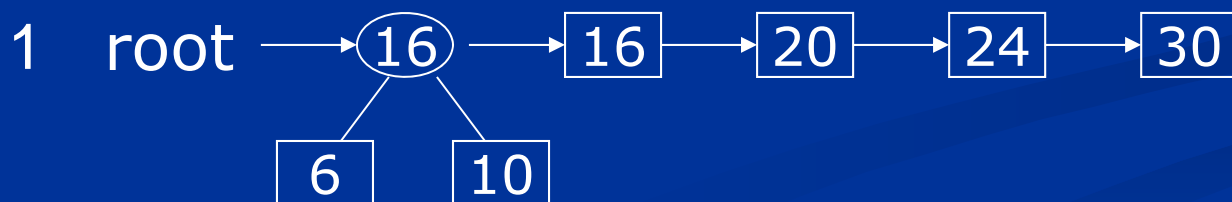
具体实现可以采用有序链表存储结构

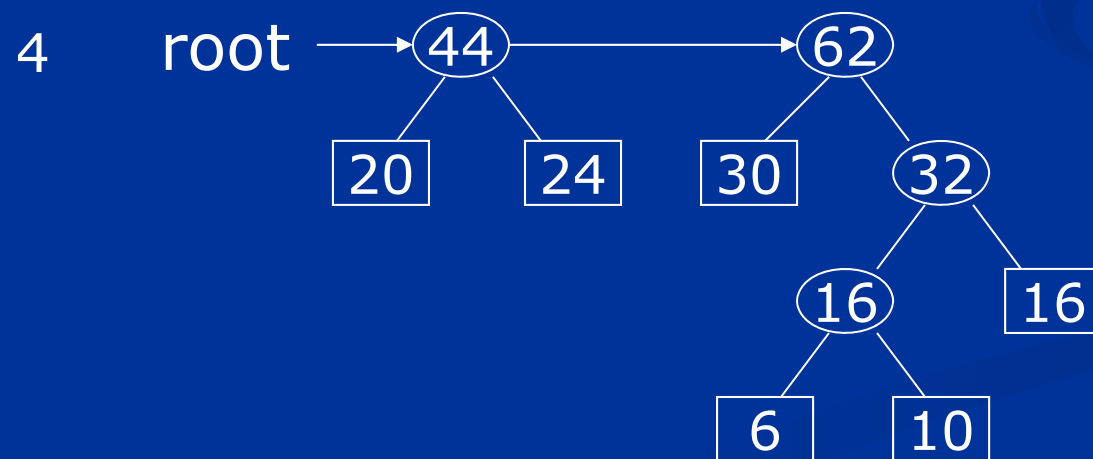
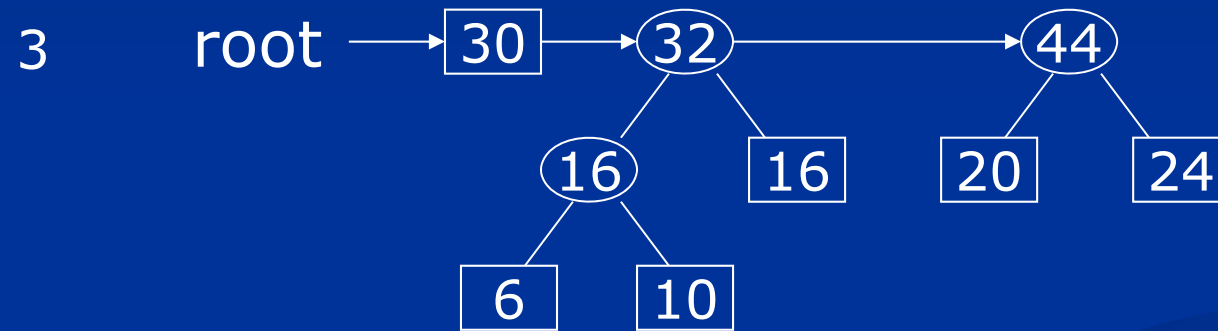
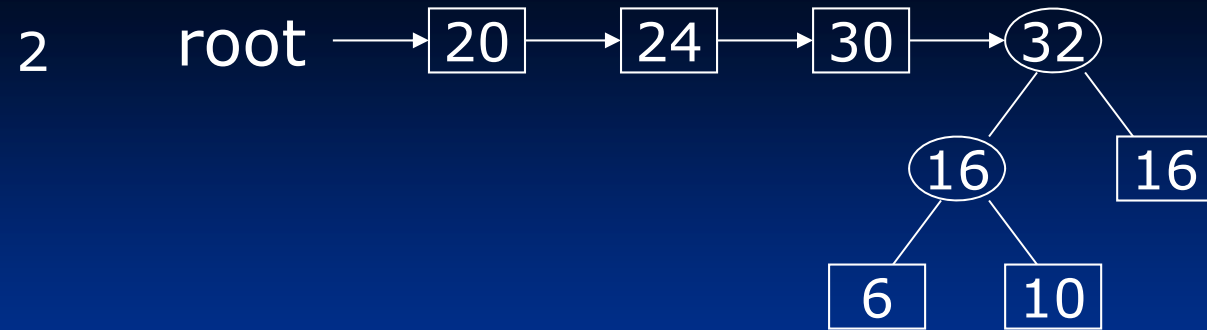
例如：对于结点序列10、16、20、6、30、24，构造huffman树的过程如下：

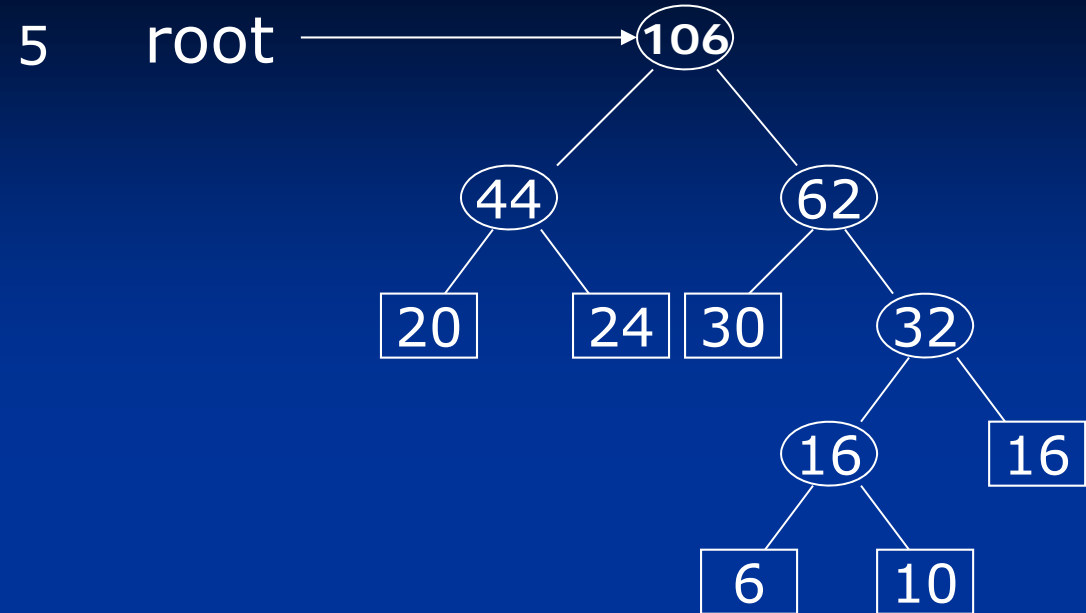
(1) 建立有序链表

root → 6 → 10 → 16 → 20 → 24 → 30

(2) 从链表中截取前面二棵根结点权值最小的树作为左右子树，生成一棵新的子树，并将新子树按其根的权值插入有序链表中去，如此循环，直到只剩一棵树为止。

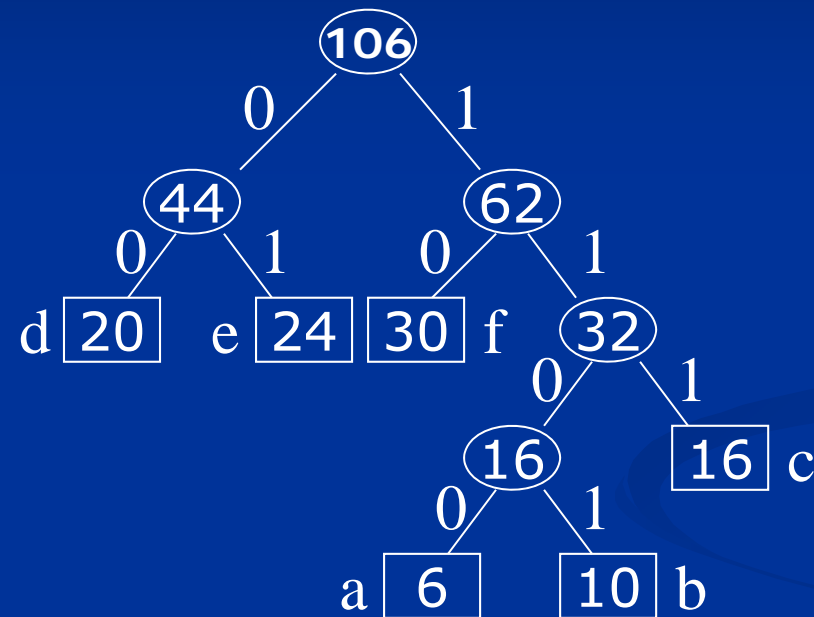






Huffman 树的应用

1、Huffman编码



各字符的二进制编码为:

a: 1100 b: 1101 c: 111 d: 00 e: 01 f: 10

2、huffman译码

从二叉树的根开始，用需要译码的二进制位串中的若干个相邻位与二叉树边上标的0、1相匹配，确定一条到达树叶的路径，一旦到达树叶，则译出了一个字符，再回到树根，从二进制位串的下一位开始继续译码。

习题

9.10 假设通讯电文中只用到A, B, C, D, E, F六个字母, 它们在电文中出现的相对频率分别为: 8, 3, 16, 10, 5, 20, 试为它们设计Huffman编码。

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
- B-树
 - 散列表检索

9.6 B-树

前面所讨论的查找算法都是在内存中进行的，它们适用于较小的文件，而对较大的、存放在外存储器上的文件就不合适了。

1972年R.Bayer和E.M.McCreight提出了一种称为B-树的多路平衡查找树，它适合在磁盘等直接存取设备上组织动态的查找表。

9.6.1 B-树的定义

B-树是一种平衡的多路查找树，在文件系统中，已经成为索引文件的一种有效结构，并得到广泛的应用。在此先介绍这种树的结构及其基本运算。

一棵 m 阶 ($m \geq 3$) B-树, 或为空树, 或为满足下列特性的 m 叉树:

- (1) 树中每个结点至多有 m 棵子树;
- (2) 若根结点不是叶子结点, 则至少有两棵子树;
- (3) 所有的非终端结点中包含下列信息

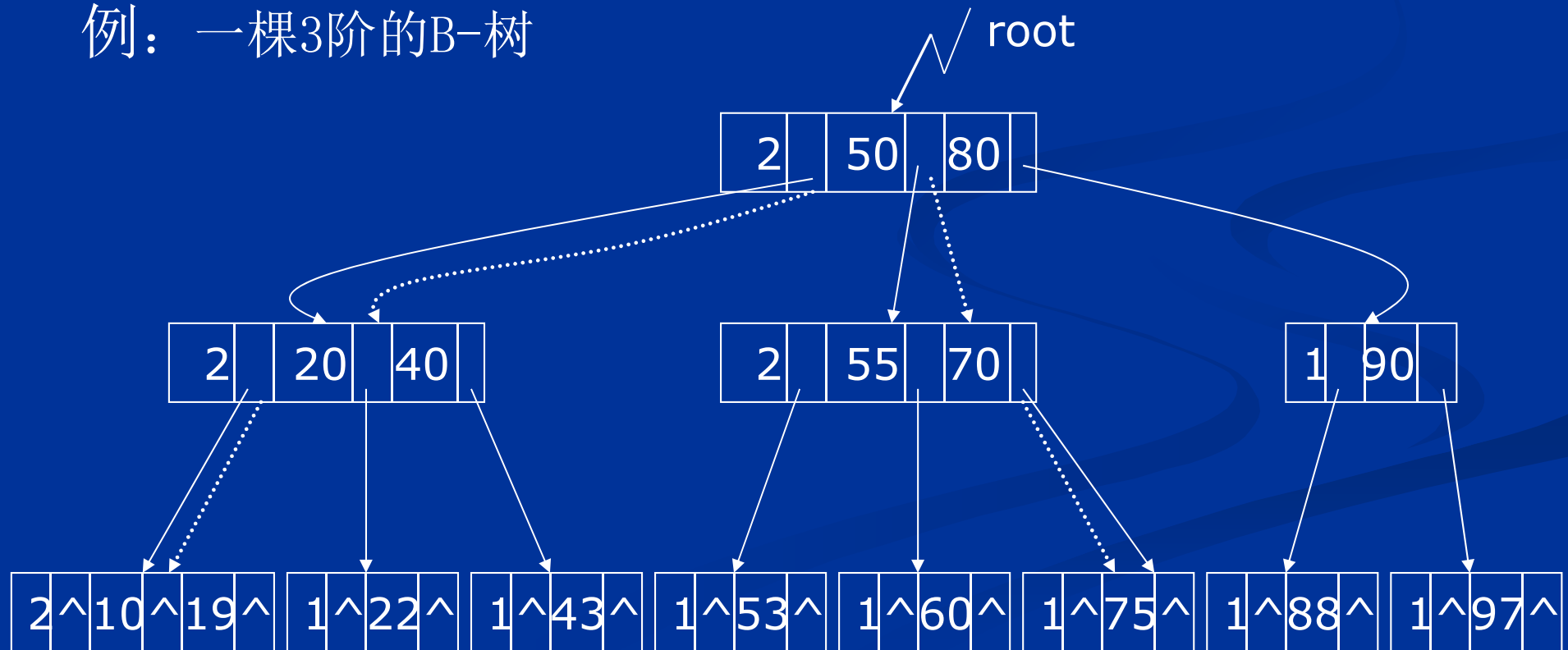
$(n, p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$

其中: k_i ($1 \leq i \leq n$) 为关键字, 且 $k_i < k_{i+1}$ ($1 \leq i < n$); p_j ($0 \leq j \leq n$) 为指向子树根结点的指针, 且 p_j ($0 \leq j < n$) 所指子树中所有结点的关键字均小于 k_{j+1} , p_n 所指子树中所有结点的关键字均大于 k_n , n ($\lceil m/2 \rceil - 1 \leq n \leq m-1$) 为关键字的个数 ($n+1$ 为子树个数)。

(4) 除根结点之外所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，也即每个非根结点至少应有 $\lceil m/2 \rceil - 1$ 个关键字；

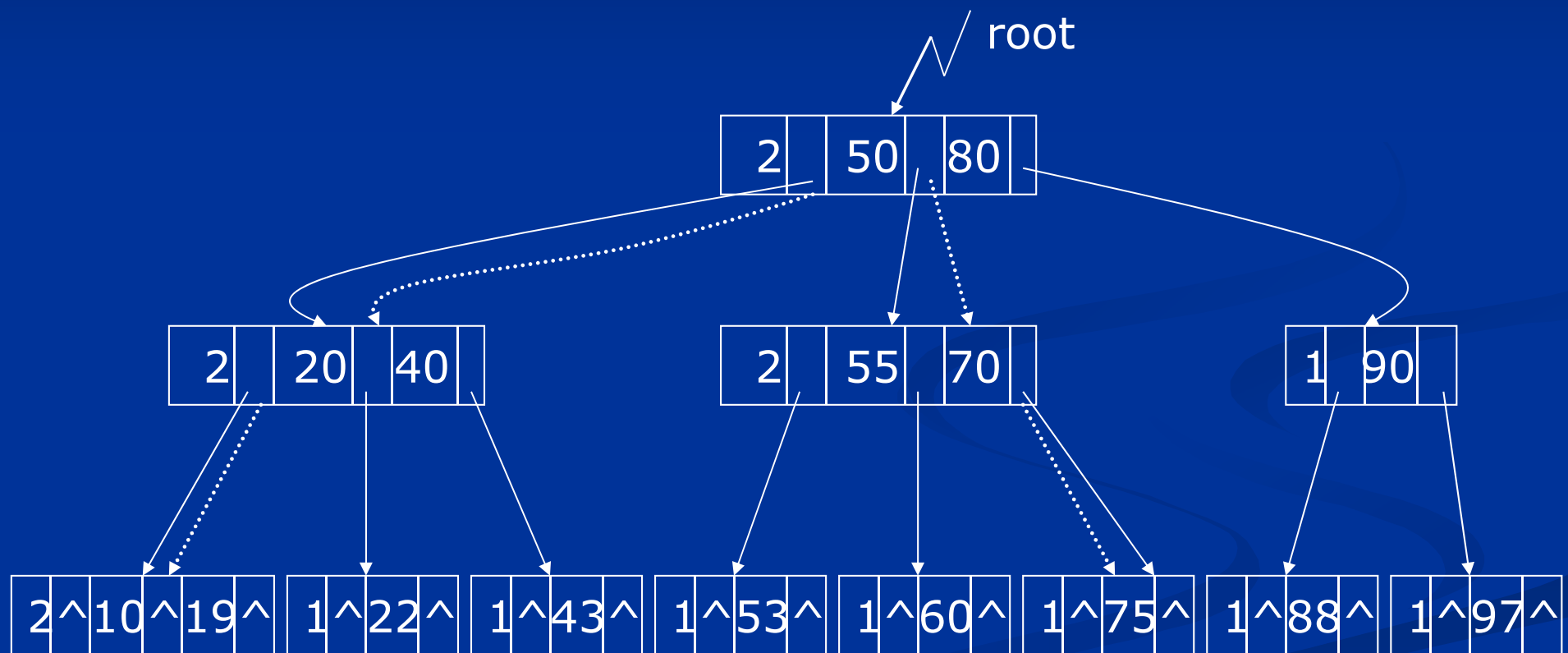
(5) 所有的叶子结点都出现在同一层上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

例：一棵3阶的B-树



9. 6. 2 B-树的基本操作

一、基于B-树的查找运算



二、基于B-树的插入运算

在B-树中插入关键字 k 的方法是：首先在树中查找 k ，若找到则直接返回（假设不处理相同关键字的插入）；否则查找操作必失败于某个叶子结点上，利用函数**btree_search**（）的返回值 $*p$ 及 $*pos$ 可以确定关键字 k 的插入位置，即将 k 插入到 p 所指的叶结点的第 pos 个位置上。若该叶结点原来是非满（结点中原有的关键字总数小于 $m-1$ ）的，则插入 k 并不会破坏B-树的性质，故插入 k 后即完成了插入操作，例如，在图9.17（a）所示的5阶B-树的某结点（假设为 p 结点）中插入新的关键字150时，可直接得到图9.17（b）所示的结果。

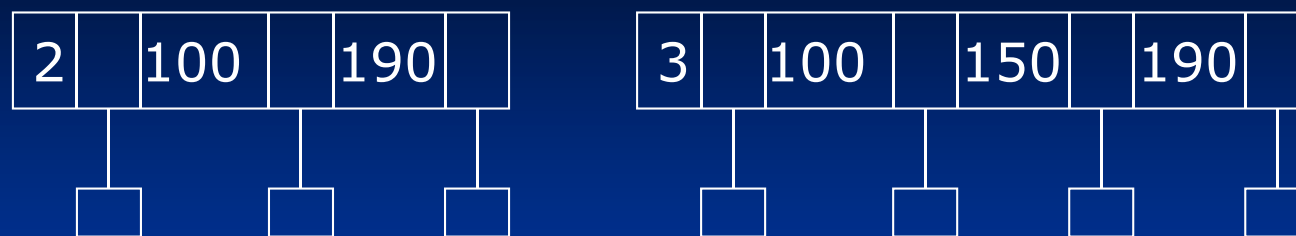
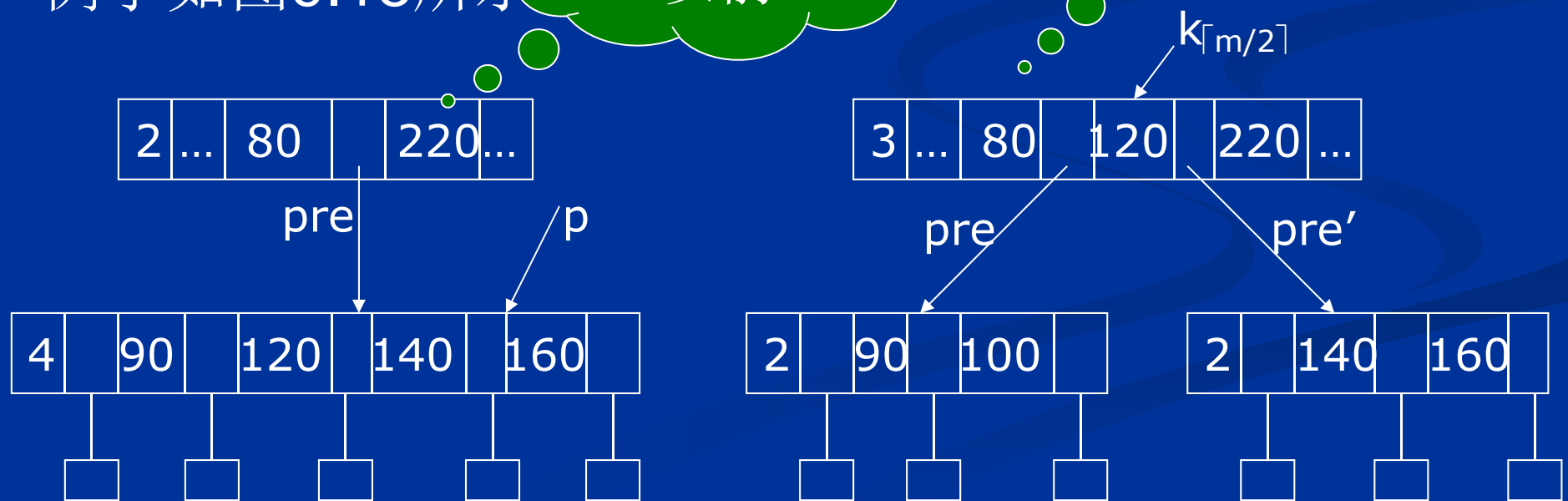


图9.17在关键字个数不满的结点中插入关键字

若 p 所指示的叶结点原为满，则 k 插入后 $keynum=m$ ，破坏了B-树的性质（1），故须调整使其维持B-树的性质不变。调整的方法是将违反性质（1）的结点以中间位置的关键字 $key[\lceil m/2 \rceil]$ 为划分点，将该结点（即 p ）

$(m, p_0, k_1, p_1, \dots, k_m, p_m)$

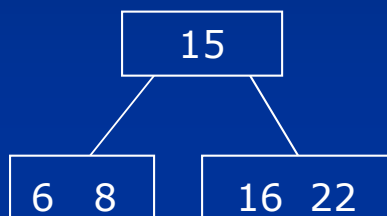
分裂为两个结点，左边结点为 $(\lceil m/2 \rceil - 1, p_0, k_1, \dots, k_{\lceil m/2 \rceil - 1}, p_{\lceil m/2 \rceil - 1})$ ，右边结点为 $(m - \lceil m/2 \rceil, p_{\lceil m/2 \rceil}, k_{\lceil m/2 \rceil + 1}, \dots, k_m, p_m)$ ，同时把中间关键字 $k_{\lceil m/2 \rceil}$ 插入到双亲结点中。于是双亲结点中指向被插入结点的指针 pre 改成 pre 、 $k_{\lceil m/2 \rceil}$ 、 pre' 三部分。指针 pre 指向分裂后的左边结点，指针 pre' 指向分裂后的右边结点。由于将 $k_{\lceil m/2 \rceil}$ 插入双亲时，双亲结点原本为满，若如此，则插入100以后



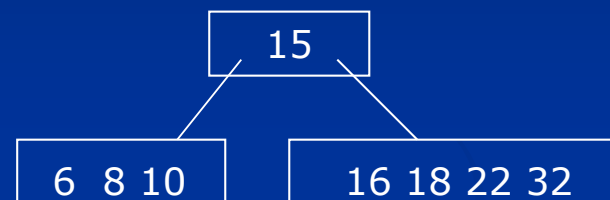
如果初始时B-树为空树，通过逐个向B-树中插入新结点，可生成一棵B-树。图9.19说明了一棵5阶B-树的生长过程。



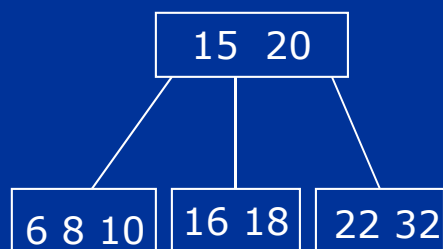
(a) 插入6、8、15、16



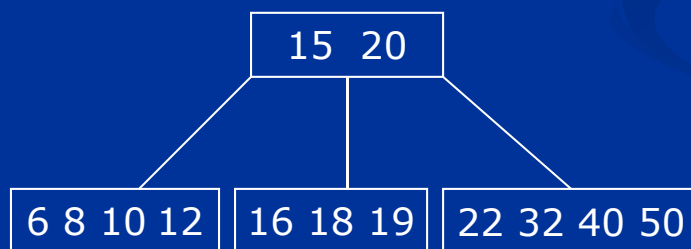
(b) 插入22



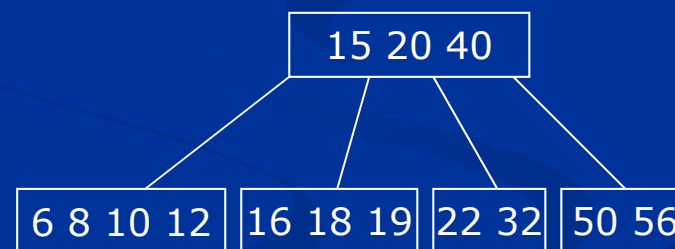
(c) 插入10、18、32



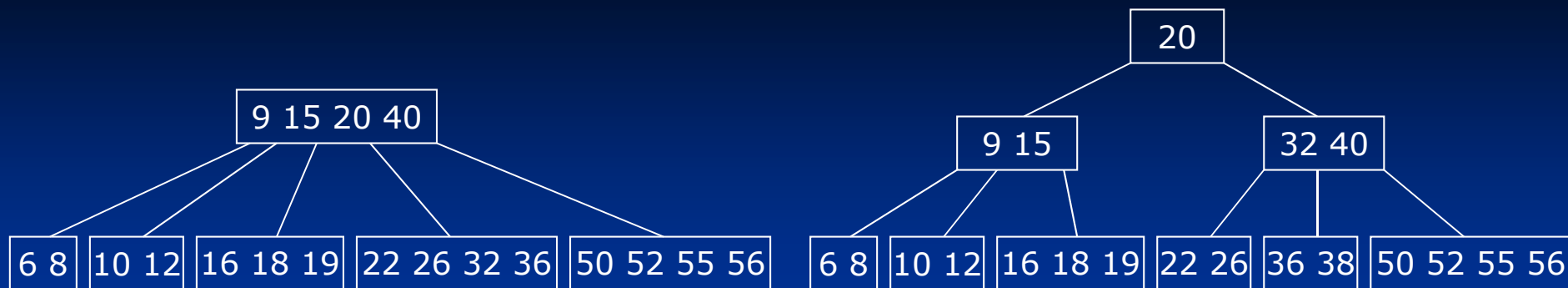
(d) 插入20



(e) 插入12、19、40、50



(f) 插入56



(g) 插入9、26、36、52、55

(h) 插入38

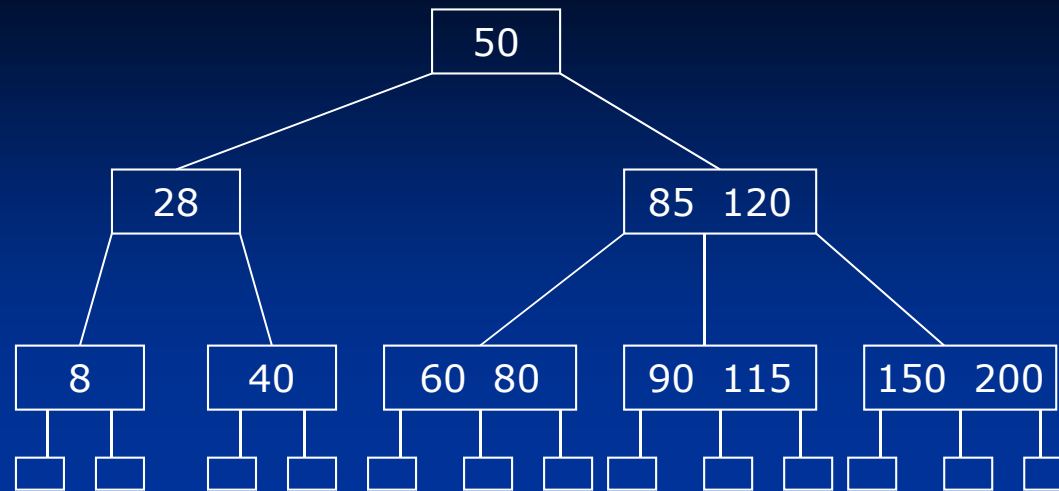
三、基于B-树的删除运算

在B-树上删除一个关键字，首先找到该关键字所在结点及其在结点中的位置。具体可分为两种情况：

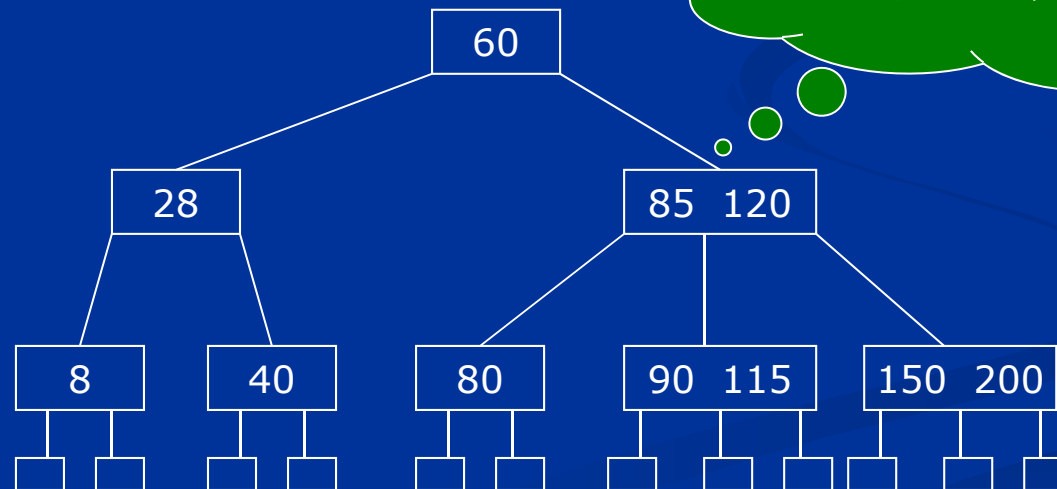
(1) 若被删除结点 k_i 在最下层的非终端结点（即叶子结点的上一层）里，则应删除 k_i 及它右边的指针 p_i 。删除后若结点中关键字数目不少于 $\lceil m/2 \rceil - 1$ ，则删除完成，否则要进行“合并”结点的操作。

(2) 假若待删结点 k_i 是最下层的非终端结点以上某层的结点, 根据B-树的特性可知, 可以用 k_i 右边指针 p_i 所指子树中最小关键字 y 代替 k_i , 然后在相应的结点中删除 y , 或用 k_i 左边指针 p_{i-1} 所指子树中最大关键字 x 代替 k_i , 然后在相应的结点中删除 x 。例如删除图9.20

(a) 所示3阶B-树中的关键字50, 可以用它右边指针所指子树中最小关键字60代替50, 尔后转化为删除叶子上面一层的结点中的60, 删除后得到的B-树如图9.20 (b) 所示。



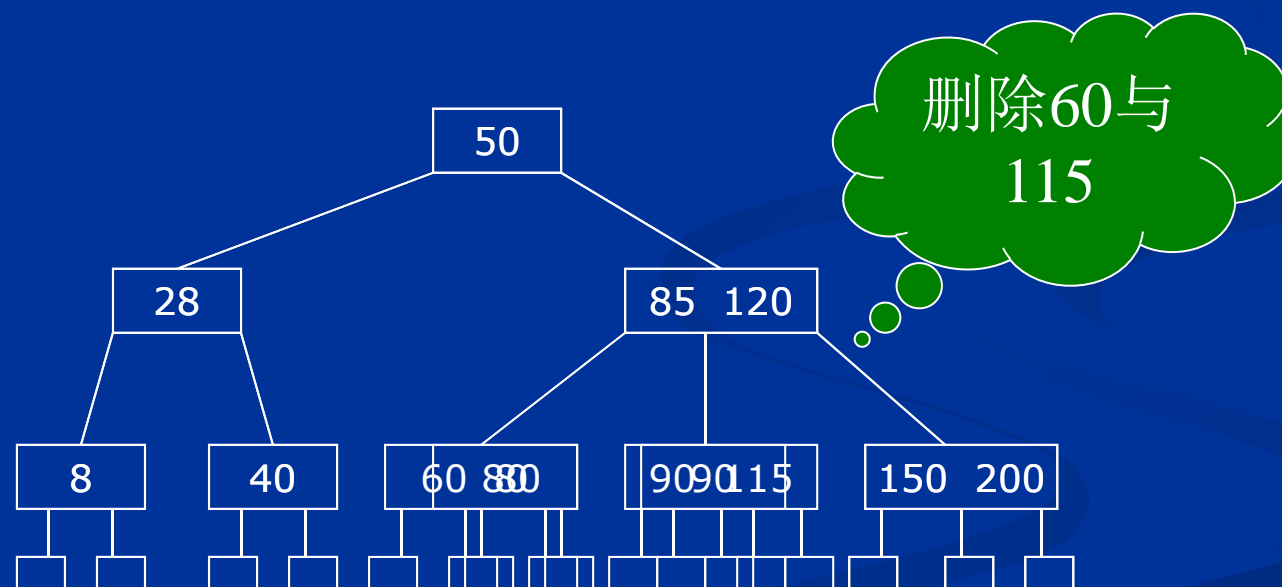
3阶B-树中删除50
以60代替50



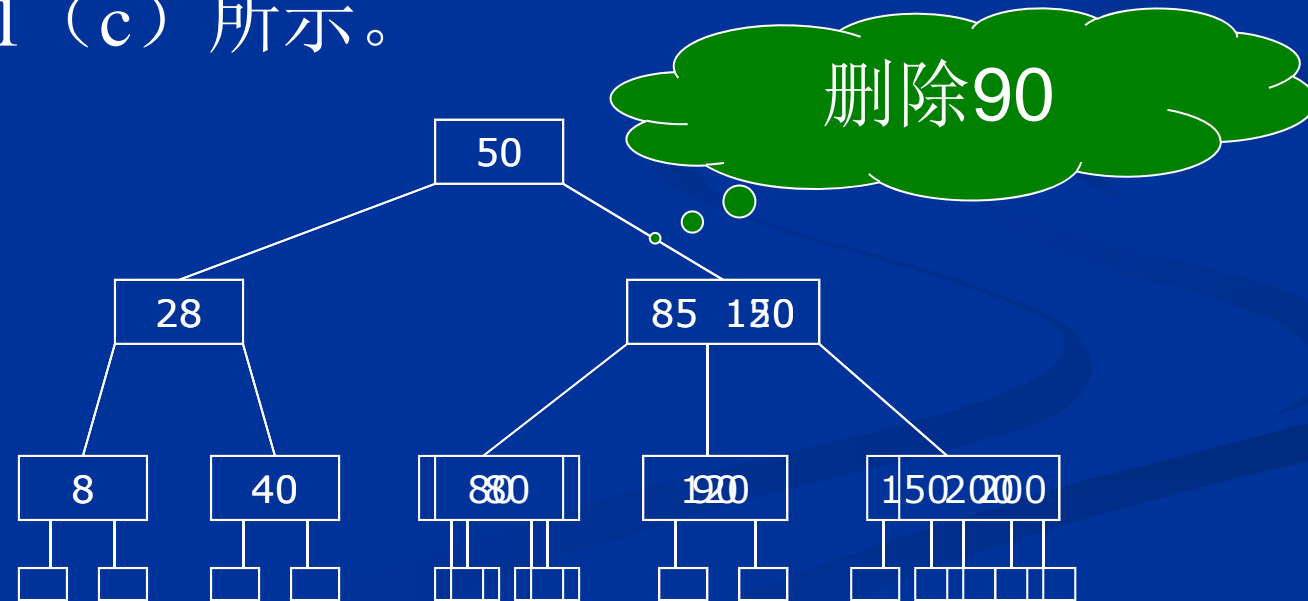
因此，下面主要讨论删除B-树叶子上面一层结点中的关键字的方法，具体分三种情形：

1) 被删关键字所在叶子上面一层结点中的关键字数目不小于 $\lceil m/2 \rceil$ ，则只需要从该结点中删去关键字 k_i 和相应的指针 p_i ，树的其它部分不变。

例：

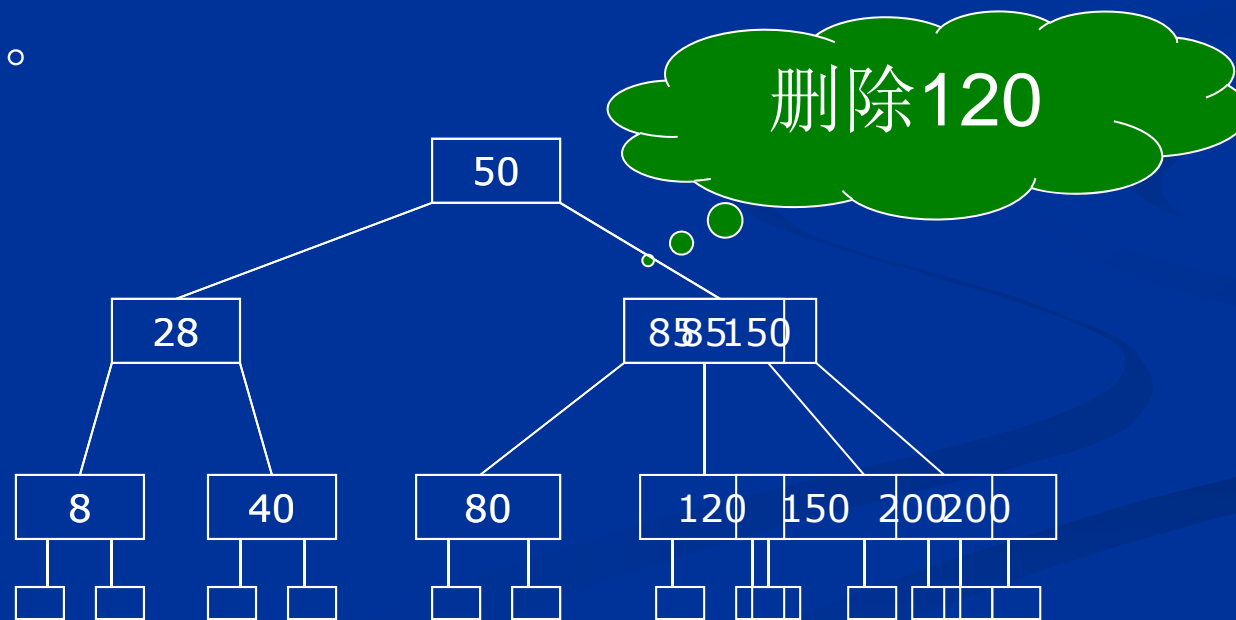


2) 被删关键字所在叶子上面一层结点中的关键字数目等于 $\lceil m/2 \rceil - 1$ ，而与该结点相邻的右兄弟结点（或左兄弟结点）中的关键字数目大于 $\lceil m/2 \rceil - 1$ ，则需要将其右兄弟的最小关键字（或其左兄弟的最大关键字）移至双亲结点中，而将双亲结点中小于（或大于）该上移关键字的关键字下移至被删关键字所在的结点中。例如从图9.21（b）中删除关键字90，结果如图9.21（c）所示。

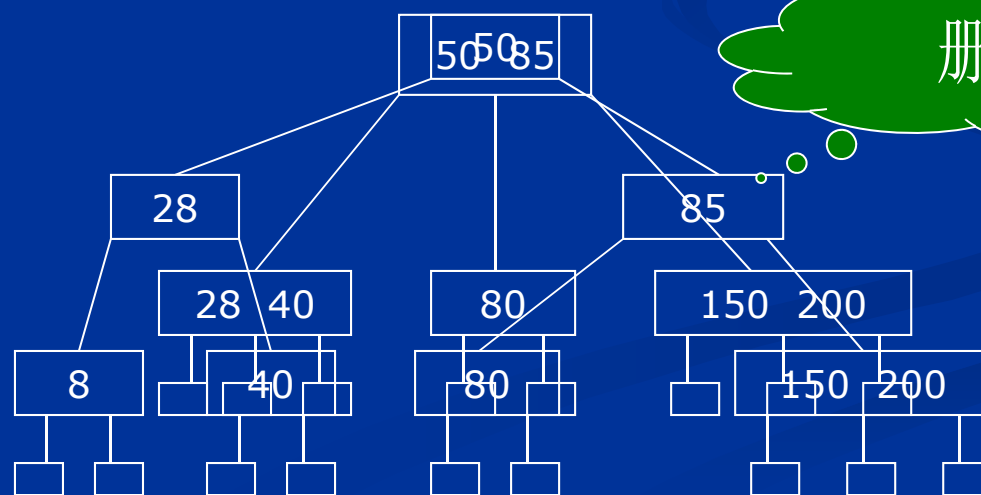


3) 被删关键字所在叶子上面一层结点中的关键字数和其相邻的兄弟结点中的关键字数目均等于 $\lceil m/2 \rceil - 1$, 则第(2)种情况中采用的移动方法将不奏效, 此时须将被删关键字所有结点与其左或右兄弟合并。不妨设该结点有右兄弟, 但其右兄弟地址由双亲结点指针 p_i 所指, 则在删除关键字之后, 它所在结点中剩余的关键字和指针加上双亲结点中的关键字 k_i 一起合并到 p_i 所指兄弟结点中(若没有右兄弟, 则合并至左兄弟结点中)。

例如，从图9.21（c）中删去关键字120，则应删去120所在结点，并将双亲结点中的150与200合并成一个结点，删除后的树如图9.21（d）所示。如果这一操作使双亲结点中的关键字数目小于 $\lceil m/2 \rceil - 1$ ，则依同样方法进行调整，最坏的情况下，合并操作会向上传播至根，当根中只有一个关键字时，合并操作将会使根结点及其两个孩子合并成一个新的根，从而使整棵树的高度减少一层。



例如，在图9.21（d）中删除关键字8，此关键字所在结点无左兄弟，只检查其右兄弟，然而右兄弟关键字数目等于 $\lceil m/2 \rceil - 1$ ，此时应检查其双亲结点关键字数目是否大于等于 $\lceil m/2 \rceil - 1$ ，但此处其双亲结点的关键字数目等于 $\lceil m/2 \rceil - 1$ ，从而进一步检查双亲结点兄弟结点关键字数目是否均等于 $\lceil m/2 \rceil - 1$ ，这里关键字28所在的结点的右兄弟结点关键字数目正好等于 $\lceil m/2 \rceil - 1$ ，因此将28和40结合成一个结点，50和85结合成一个结点，使得树变矮，删除结点8后的结果如图9.21（e）所示。



习题

9.11 含有9个叶子结点的3阶B-树中至少有多少个非叶子结点？含有10个叶子结点的3阶B-树中至少有多少个非叶子结点？

9.13 用依次输入的关键字23、30、51、29、27、15、11、17和16建一棵3阶B-树，画出建该树的变化过程示意图（每插入一个结点至少有一张图）。

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
- B-树
- 散列表检索

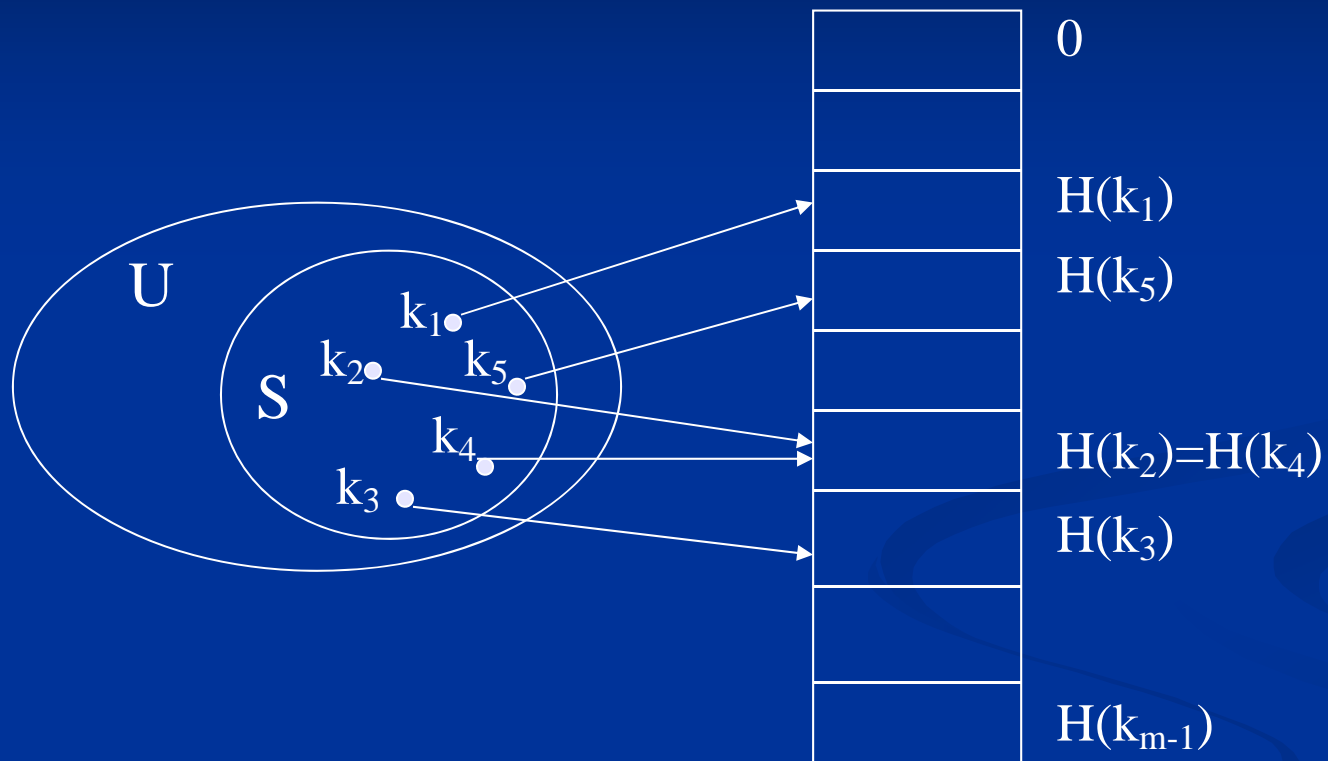
9.7散列表检索

在已经介绍过的线性表、树等数据结构中，记录存储在结构中的相对位置是随机的，因而相应的检索是通过若干次的比较以寻找指定的记录。本节将介绍一种新的存储结构——散列存储，它既是一种存储方式，又是一种常见的检索方法。

9.7.1散列存储

散列存储的基本思想是以关键码的值为自变量，通过一定的函数关系（称为散列函数，或称Hash函数），计算出对应的函数值来，以这个值作为结点的存储地址，将结点存入计算得到的存储单元里去。

图9.22 散列过程示例



散列存储中经常会出现对于两个不同关键字 x_i, x_j , 却有 $H(x_i) = H(x_j)$, 即对于不同的关键字具有相同的存放地址, 这种现象称为冲突或碰撞。碰撞的两个(或多个)关键字称为同义词(相对于函数 H 而言)。

“负载因子” α 反映了散列表的装填程度, 其定义为:

$$\alpha = \frac{\text{散列表中结点的数目}}{\text{基本区域能容纳的结点数}}$$

当 $\alpha > 1$ 时冲突是不可避免的。因此, 散列存储必须考虑解决冲突的办法。

综上所述，对于Hash方法，需要研究下面两个主要问题：

（1）选择一个计算简单，并且产生冲突的机会尽可能少的Hash函数；

（2）确定解决冲突的方法。

9.7.2散列函数的构造

（1）除余法

$$H(\text{key}) = \text{key} \% p$$

例如 $S=\{5, 21, 65, 22, 69\}$ ，若 $m=7$ 且 $H(x) = x \% 7$ ，则可以得到如表9.1所示的Hash表。

0	1	2	3	4	5	6
	21	22	65			5 69

表9.1 散列表表示例

(2) 平方取中法

取关键字平方后的中间几位为Hash地址，所取的位数和Hash地址位数相同。这是一种较常用的构造Hash函数的方法。因为通常在选定Hash函数时不一定能知道关键字的全部情况，难以决定取其中哪几位比较合适，而一个数平方后的中间几位数和数的每一位都相关，由此使随机分布的关键字得到的Hash地址也是随机的。

(3) 数字分析法

对于关键字的位数比存储区域的地址码位数多的情况，可以采取对关键字的各位进行分析，丢掉分布不均匀的位留下分布均匀的位作为Hash地址，这种方法称为数字分析法。

Key	H (key)
0 1 9 <u>4</u> 2 8 <u>3 2</u>	432
0 1 5 <u>7</u> 9 8 <u>8 5</u>	785
0 1 9 <u>5</u> 9 0 <u>1 3</u>	513
0 1 2 <u>8</u> 1 5 <u>3 8</u>	838
0 1 9 <u>2</u> 1 8 <u>2 7</u>	227
0 1 7 <u>1</u> 1 8 <u>4 6</u>	146

（4）折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为Hash地址，称为折叠法。关键字位数很多且关键字中每一位上数字分布大致均匀时，可以采用折叠法得到Hash地址。

在折叠法中数位叠加可以有移位叠加和间界叠加两种方法。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是从一端向另一端沿分割界来回折迭，然后对齐相加。如关键码为7-302-03806-6，若Hash地址取4位，则此关键字的Hash地址采用折叠法得到如图9.23所示的结果。

$$\begin{array}{r}
 8006 \\
 0203 \\
 +) 073 \\
 \hline
 8282
 \end{array}$$

$$H(\text{key}) = 8282$$

(a) 移位叠加

$$\begin{array}{r}
 8006 \\
 3020 \\
 +) 073 \\
 \hline
 11099
 \end{array}$$

$$H(\text{key}) = 1099$$

(b) 间界叠加

图9.23 由折叠法求得Hash地址

(5) 直接地址法

取关键字或关键字的某个线性函数值为哈希地址，即：

$$H(\text{key}) = \text{key} \text{ 或 } H(\text{key}) = a \cdot \text{key} + b$$

9.7.3冲突处理

1、开放定址法

开放定址法的基本做法是在发生冲突时，按照某种方法继续探测基本表中的其它存储单元，直到找到一个开放的地址（即空位置）为止。显然这种方法需要用某种标记区分空单元与非空单元。

开放定址法的一般形式可表示为：

$$H_i(k) = (H(k) + d_i) \bmod m \quad (i=1, 2, \dots, k \ (k \leq m-1))$$

其中， $H(k)$ 为键字为 k 的直接哈希地址， m 为哈希表长， d_i 为每次再探测时的地址增量。

当 $d_i=1, 2, 3, \dots, m-1$ 时, 称为线性探测再散列;
当 $d_i=1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$ ($k \leq m/2$) 时, 称为二次探测再散列; 当 d_i =随机数序列时, 称为随机探测再散列。

例如, 有数据 (654, 638, 214, 357, 376, 854, 662, 392), 现采用数字分析法, 取得第二位数作为哈希地址, 将数据逐个存放入大小为10的散列表 (此处为顺序表) 中。若采用线性探测法解决地址冲突, 则8个数据全部插入完成后, 散列表的状态如表9.2所示。

0	1	2	3	4	5	6	7	8	9
392	214		638		654	357	376	854	662

2、再哈希法

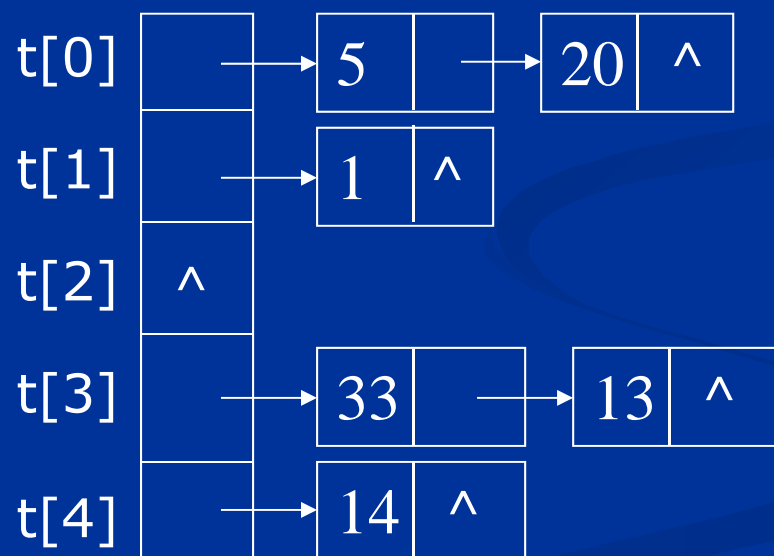
采用再哈希法解决冲突的做法是当待存入散列表的某个元素 k 在原散列函数 $H(k)$ 的映射下与其它数据发生碰撞时，采用另外一个Hash函数 $H_i(k)$ ($i=1, 2, \dots, n$) 计算 k 的存储地址 (H_i 均是不同的Hash函数)，这种计算直到冲突不再发生为止。

3、拉链法

拉链法解决冲突的做法是，将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m ，则可将散列表定义为一个由 m 个头指针组成的指针数组 $T[0..m-1]$ ，凡是散列地址为 i 的结点，均插入到以 $T[i]$ 为头指针的单链表中。

拉链法的缺点主要是指针需要用额外的空间，故当结点规模较小时，开放定址法较为节省空间。

例如，关键字集合为{1, 13, 20, 5, 14, 33}，散列表长度 $m=5$ ，现采用除余法为哈希函数并采用拉链法解决地址冲突，所创建的Hash链表如图9.24所示。



除了上述三种方法外，还有差值法可解决地址冲突。这种方法在发生冲突时，处理原则以现在的数据地址加上一个固定的差值，当数据地址超出数据大小时，则让数据地址采用循环的方式处理。另外，还可以建立一个公共溢出区的方法去解决冲突。即 m 个Hash地址用数组 $t[0..m-1]$ 表示，称此表为基本表，每一个分量存放一个关键字，另外设立一个数组 $v[0..n]$ 为溢出表。若关键字和基本表中关键字为同义词，不管它由Hash函数得到的Hash地址是什么，一旦发生冲突，都填入溢出表。

假设负载系数为 α ，则：

（1）如果用开放定址线性探测再散列法解决冲突，Hash表查找成功和查找不成功的平均查找长度 S_n 和 U_n 分别为：

$$S_n \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right) \quad U_n \approx \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2}\right)$$

（2）如果用二次探测再散列解决冲突，Hash查找成功和查找不成功的平均查找长度 S_n 和 U_n 分别为：

$$S_n \approx \frac{1}{(1-\alpha)} \quad U_n \approx -\frac{1}{\alpha} \ln(1-\alpha)$$

(2) 如果用拉链法解决冲突, Hash表查找成功和查找不成功的平均查找长度 S_n 和 U_n 分别为:

$$S_n \approx 1 + \frac{\alpha}{2} \quad U_n \approx \alpha + e^{-\alpha}$$

习题

9.14 设散列表长度为11，散列函数 $H(x) = x \% 11$ ，给定的关键字序列为：1，13，12，34，38，33，27，22。试画出分别用拉链法和线性探测法解决冲突时所构造的散列表，并求出在等概率的情况下，这两种方法查找成功和失败时的平均查找长度。

9.15 设散列表为 $T[0..12]$ ，即表的大小 $m=13$ 。现采用再哈希法（双散列法）解决冲突。散列函数和再散列函数分别为：

$$H_0(k) = k \% 13、$$

$$H_i = (H_{i-1} + \text{REV}(k+1) \% 11 + 1) \% 13; i=1, 2, \dots, m-1$$

其中，函数 $REV(x)$ 表示颠倒10进制数的各位，如 $REV(37)=73$ ， $REV(1)=1$ 等。若插入的关键码序列为{2, 8, 31, 20, 19, 18, 53, 27}。

- (1) 试画出插入这8个关键码后的散列表。
- (2) 计算检索成功的平均查找长度ASL。

第9章 检索

- 检索的基本概念
 - 线性表的检索
 - 二叉排序树
 - 丰满树和平衡树
- 最佳二叉排序树和Huffman树
 - B-树
 - 散列表检索

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.1 排序的基本概念

假设一个文件是由 n 个记录 R_1, R_2, \dots, R_n 组成, 所谓排序就是以记录中某个(或几个)字段值不减(或不增)的次序将这 n 个记录重新排列, 称该字段为排序码。能唯一标识一个记录的字段称为关键码, 关键码可以作为排序码, 但排序码不一定是关键码。

按排序过程中使用到的存储介质来分，可以将排序分成**两大类:内排序和外排序**。

内排序是指在排序过程中所有数据均放在内存中处理，不需要使用外存的排序方法。而对于数据量很大的文件，在内存不足的情况下，则还需要使用外存，这种排序方法称为**外排序**。

排序码相同的记录，若经过排序后，这些记录仍保持原来的相对次序不变，称这个排序算法是**稳定的**。否则，称为**不稳定的排序算法**。

评价排序算法优劣的标准：

首先考虑算法执行所需的时间，这主要是用执行过程中的比较次数和移动次数来度量；

其次考虑算法执行所需要的附加空间。

当然，保证算法的正确性是不言而喻的，可读性等也是要考虑的因素。

排序算法如未作特别的说明，使用的有关定义如下：

/*常见排序算法的头文件，文件名table.h*/

#define MAXSIZE 100 /*文件中记录个数的最大值*/

typedef int keytype; /*定义排序码类型为整数类型*/

typedef struct{

 keytype key;

 /*此处还可以定义记录中除

}recordtype; /*记录

typedef struct{

 recordtype r[MAXSIZE+1];

 int length; /*待排序

}table; /*待排序

为了方便，**r[0]**一般不用于存放排序码，在一些排序算法中它可以用来作为中间单元存放临时数据。**length**域是待排序的记录个数，它必须不大于**MAXSIZE**，这样，第**1~length**个记录的排序码分别存于**r[1].key~r[length].key**中

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.2 插入排序

插入排序的基本方法是：

将待排序文件中的记录， 逐个地按其排序码值的大小插入到目前已经排好序的若干个记录组成的文件中的适当位置， 并保持新文件有序。

10.2.1 直接插入排序

直接插入排序算法的思路是：初始可认为文件中的第1个记录已排好序，然后将第2个到第n个记录依次插入已排序的记录组成的文件中。在对第i个记录 R_i 进行插入时， R_1, R_2, \dots, R_{i-1} 已排序，将记录 R_i 的排序码 key_i 与已经排好序的排序码从右向左依次比较，找到 R_i 应插入的位置，将该位置以后直到 R_{i-1} 各记录顺序后移，空出该位置让 R_i 插入。

一组记录的排序码分别为:

312, 126, 272, 226, 28, 165, 123

初始时将第1个排序码作为已经排好序的, 把排好序的数据记录放入中括号[]中, 表示有序的文件, 剩下的在中括号外, 如下所示:

[312], 126, 272, 226, 28, 165, 123

设前3个记录的排序码已重新排列有序, 构成一个含有3个记录的有序文件:

[126, 272, 312], 226, 28, 165, 123

现在要将第4个排序码226插入!

[126, 272, 312], 226, 28, 165, 123

现在要将第4个排序码226插入！

将待插入的排序码226和已经有序的最后一个人排序码312比较，因为待插入的排序码226小于312，所以226肯定要置于312的前面，至于是否就是置于312的前一个位置，此时还不能确定，需要继续向左比较；

将所有大于待插入排序码226的那两个排序码312和272依次后移一个位置，在空出的位置插入待排序的排序码226，得一含有4个记录的有序文件：

[126, 226, 272, 312], 28, 165, 123

需要注意的是，当待插入排序码小于所有已排序的排序码时，如在插入第5个值28时：

[126, 226, 272, 312], 28, 165, 123

算法设计的时候如处理？

方法之一：设置“哨兵”

```
void insertsort(table *tab)
{
    int i,j;
    for(i=2;i<=tab->length;i++)/*依次插入从第2个开始的所有元素*/
    { j=i-1;
        tab->r[0].key=tab->r[i].key;/*设置哨兵，准备找插入位置*/
        while(tab->r[0].key<tab->r[j].key) /*找插入位置并后移*/
        { tab->r[j+1].key=tab->r[j].key;    /*后移*/
            j=j-1;                /*继续向前（左）查找*/
        }
        tab->r[j+1].key=tab->r[0].key; /*插入第i个元素的副本，即前面设置的哨兵*/
    }
}
```

算法10.1 直接插入排序算法

设待排序的7记录的排序码为{312, 126, 272, 226, 28, 165, 123}, 直接插入排序算法的执行过程如图10.2所示。

哨兵		排序码
		[] 312, 126, 272, 226, 28, 165, 123
初始	()	[312], 126, 272, 226, 28, 165, 123
i=2:	(126)	[126, 312], 272, 226, 28, 165, 123
i=3:	(272)	[126, 272, 312], 226, 28, 165, 123
i=4:	(226)	[126, 226, 272, 312], 28, 165, 123
i=5:	(28)	[28, 126, 226, 272, 312], 165, 123
i=6:	(165)	[28, 126, 165, 226, 272, 312], 123
i=7:	(123)	[28, 123, 126, 165, 226, 272, 312]

图10.2 直接插入排序算法执行过程示意图

直接插入排序算法执行时间的分析：

最好的情况：

即初始排序码开始就是有序的情况下，因为当插入第 i 个排序码时，该算法内循环`while`只进行一次条件判断而不执行循环体，外循环共执行 $n-1$ 次，其循环体内不含内循环每次循环要进行2次移动操作，所以在最好情况下，直接插入排序算法的比较次数为 $(n-1)$ 次，移动次数为 $2*(n-1)$ 次。

最坏情况：

即初始排序码开始是逆序的情况下，因为当插入第*i*个排序码时，该算法内循环while要执行*i*次条件判断，循环体要执行*i-1*次，每次要移动1个记录，外循环共执行*n-1*次，其循环体内不含内循环每次循环要进行2次移动操作，所以在最坏情况下，比较次数为 $(1+2+\dots+n)*(n-1)$ ，移动次数为 $(1+2+2+2+\dots+n+2)*(n-1)$ 。假设待排序文件中的记录以各种排列出现的概率相同，因为当插入第*i*个排序码时，该算法内循环while平均约要执行*i/2*次条件判断，循环体要执行 $(i-1)/2$ 次，外循环共执行*n-1*次，所以平均比较次数约为 $(2+3+\dots+n)/2*(n-1)$ ，平均移动次数为 $(n-1)*(2+1+3+1+\dots+n+1)/2$ ，也即直接插入排序算法的时间复杂度为 $O(n^2)$ 。

10.2.2 二分法插入排序

二分法插入排序的思想：

根据插入排序的基本思想，在找第 i 个记录的插入位置时，前 $i-1$ 个记录已排序，将第 i 个记录的排序码 $key[i]$ 和已排序的前 $i-1$ 个的中间位置记录的排序码进行比较，如果 $key[i]$ 小于中间位置记录排序码，则可以在前半部继续使用二分法查找，否则在后半部继续使用二分法查找，直到查找范围为空，即可确定 $key[i]$ 的插入位置。


```
void binarysort(table *tab)
{ int i,j,left,right,mid;
  for(i=2;i<=tab->length;i++) /*依次插入从第2个开始的所有元素*/
  { tab->r[0].key=tab->r[i].key; /*保存待插入的元素*/
    left=1;right=i-1; /*设置查找范围的左、右位置值*/
    while(left<=right) /*查找第i个元素的插入位置*/
    { mid=(left+right)/2; /*取中点位置*/
      if(tab->r[i].key<tab->r[mid].key) right=mid-1;
      else left=mid+1;
    } /*插入位置为left*/
    for(j=i-1;j>=left;j--) tab->r[j+1].key=tab->r[j].key; /*后移,空出插入位置*/
    tab->r[left].key=tab->r[0].key; /*插入第i个元素的副本*/
  }
} /*算法10.2 二分法插入排序算法 */
```

设待排序的7记录的排序码为{312, 126, 272, 226, 28, 165, 123}, 在前6个记录已经排序的情况下, 使用二分法插入排序算法插入第7个记录的排序码123的执行过程示意如图10.3所示 (见书本)。

二分法插入排序算法, 在查找第*i*个记录的插入位置时, 每执行一次while循环体, 查找范围缩小一半, 和直接插入排序的比较次数对比, 二分法插入的比较次数少于直接插入排序的最多比较次数, 而一般要多于直接插入排序的最少比较次数。总体上讲, 当*n*较大时, 二分法插入排序的比较次数远少于直接插入排序的平均比较次数, 但二者所要进行的移动次数相等, 故二分法插入排序的时间复杂度也是 $O(n^2)$, 所需的附加存储空间为一个记录空间。

10.2.3 表插入排序

二分法插入排序比较次数通常比直接插入排序的比较次数少，但移动次数相等。表插入排序将在不进行记录移动的情况下，利用存储结构有关信息的改变来达到排序的目的。

给每个记录附设一个所谓的指针域link，它的类型为整型，**表插入排序的思路**：在插入第 i 个记录 R_i 时， R_1, R_2, \dots, R_{i-1} 已经通过各自的指针域link按排序码不减的次序连接成一个（静态链）表，将记录 R_i 的排序码 key_i 与表中已经排好序的排序码从表头向右、或称向后依次比较，找到 R_i 应插入的位置，将其插入在表中，使表中各记录的排序码仍然有序。

```
/* 表插入排序定义的头文件，文件名为：table2.h */  
  
#define MAXSIZE 100    /*文件中记录个数的最大值*/  
  
typedef int keytype;    /*定义排序码类型为整数类型*/  
  
typedef struct {  
    keytype key;  
    int link;  
    /*此处还可以定义记录中除排序码外的其它域*/  
} recordtype;          /*记录类型的定义*/  
  
typedef struct {  
    recordtype r[MAXSIZE+1];  
    int length;          /*待排序文件中记录的个数*/  
} table2;               /*待排序文件类型*/
```

表插入排序算法的示意如图10.4所示（见书本）

对于将一个值为 x 的记录，插入到一个已排序（不减）的单链表`head`中，使新的单链表的结点值以不减序排列，读者容易给出解决此问题的算法。

表插入排序算法：初始时，`r[0].Link`用于存放表中第1个记录的下标，`r[0].Link`的值为1，排序结束时，`r[0].Link`中存放的是所有排序码中值最小的对应记录的下标，其它的排序码通过各自的指针域`link`按不减的次序连接成一个（静态链）表，最大的排序码对应的`link`为0。

```

void tableinsertsort(table2 *tab)
{ int i,p,q;
  tab->r[0].link=1;tab->r[1].link=0;    /*第1个元素为有序静态表*/
  for(i=2;i<=tab->length;i++)          /*依次插入从第2个开始的所有元素*/
  {
    q=0;p=tab->r[0].link;              /*p指向表中第1个元素，q指向p的前驱元素位置*/
    while(p!=0&&tab->r[i].key>=tab->r[p].key) /*找插入位置*/
    {
      q=p;    p=tab->r[p].link;          /*继续查找*/
    }
    tab->r[i].link=p;tab->r[q].link=i; /*将第i个元素插入q和p所指向的元素之间*/
  }
}

```

算法10.3 表插入排序算法

10.2.4 Shell插入排序

	312	126	272	226	28	165	123
0	1	2	3	4	5	6	7

(a) 初始状态

	123	28	165	226	126	272	312
0	1	2	3	4	5	6	7

(b) $d=2$

	28	123	126	165	226	272	312
0	1	2	3	4	5	6	7

(c) $d=1$

设待排序的7记录的排序码为{312, 126, 272, 226, 28, 165, 123}, 初始让 $d=7/2=3$, 以后每次让 d 缩小一半, 其排序过程如图所示。

```

void shellinsertsort(table *tab)
{ int i,j,d;
  d=tab->length/2;
  while(d>=1)
  { for(i=d+1;i<=tab->length;i++) /*从第d+1个元素开始,将所有元素有序插入相应分组中*/
    { tab->r[0].key=tab->r[i].key; /*保存第i个元素*/
      j=i-d; /*向前找插入位置*/
      while(tab->r[0].key<tab->r[j].key&& j>0) /*找插入位置并后移*/
      { tab->r[j+d].key=tab->r[j].key; /*后移*/ j=j-d; /*继续向前查找*/ }
      tab->r[j+d].key=tab->r[0].key; /*插入第i个元素的副本*/
    }
    d=d/2;
  }
}

```

算法10.4 Shell插入排序算法

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.3 选择排序

选择排序的基本思想是：每次从待排序的文件中选出排序码最小的记录，将该记录放于已排序文件的最后一个位置，直到已排序文件记录个数等于初始待排序文件的记录个数为止。

10.3.1 直接选择排序

首先从所有 n 个待排序记录中选择排序码最小的记录，将该记录与第1个记录交换，再从剩下的 $n-1$ 个记录中选出排序码最小的记录和第2个记录交换。重复这样的操作直到剩下2个记录时，再从中选出排序码最小的记录和第 $n-1$ 个记录交换。剩下的那1个记录肯定是排序码最大的记录，这样排序即告完成。

```

void simpleselectsort(table *tab)
{ int i,j,k;
  for(i=1;i<=tab->length-1;i++)
  { k=i;          /*记下当前最小元素的位置*/
    for(j=i+1;j<=tab->length;j++) /*向右查找更小的元素*/
      if(tab->r[j].key<tab->r[k].key) k=j; /*修改当前最小元素的位置*/
    if(k!=i) /*如果第i次选到的最小元素位置k不等于i，则将第k、i个元素交换*/
    { tab->r[0].key=tab->r[k].key; /*以第0个元素作为中间单元进行交换*/
      tab->r[k].key=tab->r[i].key;
      tab->r[i].key=tab->r[0].key;
    }
  }
}

```

算法10.5 直接选择排序算法

直接选择排序算法执行过程如图10.6所示（见书本）

10.3.2 树型选择排序（略）

10.3.3 堆排序

为了既要保存中间比较结果，减少后面的比较次数，又不占用大量的附加存储空间，使排序算法具有较好的性能，Williams和Floyd在1964年提出的称为堆排序的算法实现了这一想法。

堆是一个序列 $\{k_1, k_2, \dots, k_n\}$ ，它满足下面的条件：

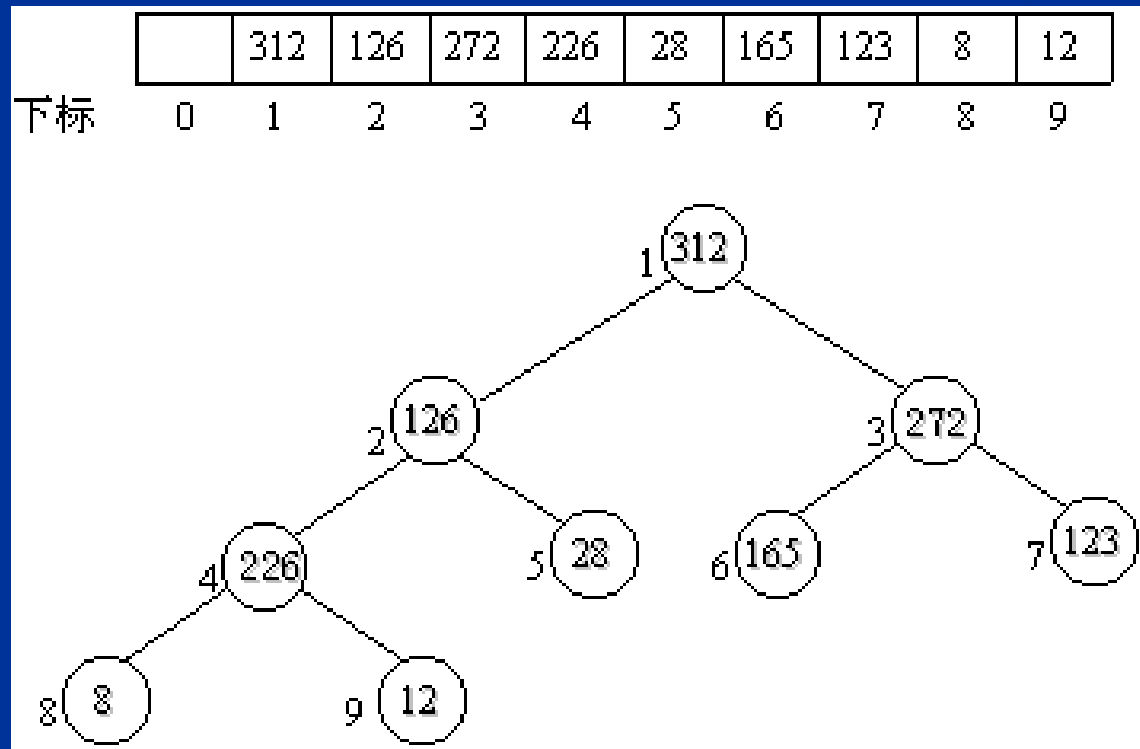
$$k_i \leq k_{2i} \text{ 并且 } k_i \leq k_{2i+1}, \text{ 当 } i=1, 2, \dots, \lfloor n/2 \rfloor$$

采用顺序方式存储这个序列，就可以将这个序列的每一个元素 k_i 看成是一颗有 n 个结点的完全二叉树的第 i 个结点，其中 k_1 是该二叉树的根结点。

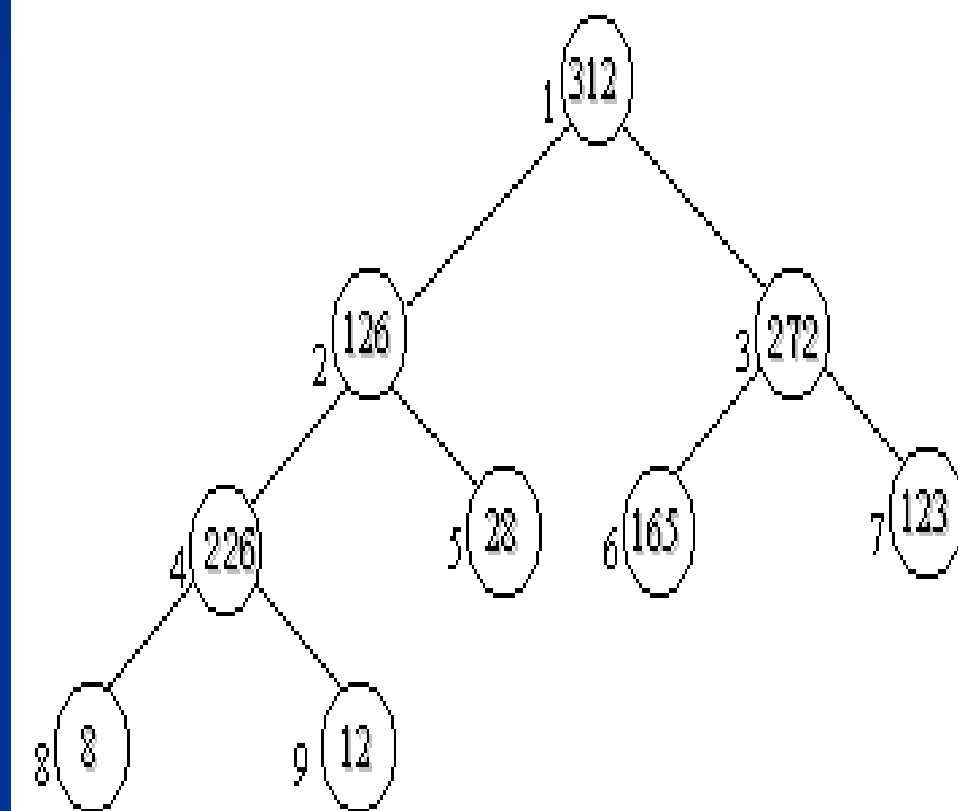
把堆对应的一维数组(即该序列的顺序存储结构)看作一棵完全二叉树的顺序存储, 那么堆的特征可解释为, 完全二叉树中任一分支结点的值都小于或等于它的左、右儿子结点的值。堆的元素序列中的第一个元素 k_1 , , 即对应的完全二叉树根结点的值是所有元素中值最小的。堆排序方法就是利用这一点来选择最小元素。

一个序列和相应的完全二叉树:

这个序列不是一个堆。堆排序的关键问题是如何将待排序记录的排序码建成一个堆。



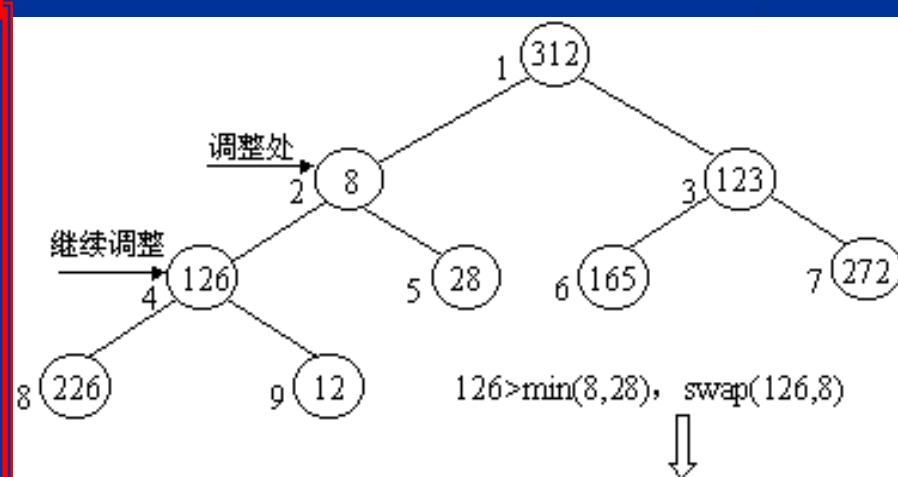
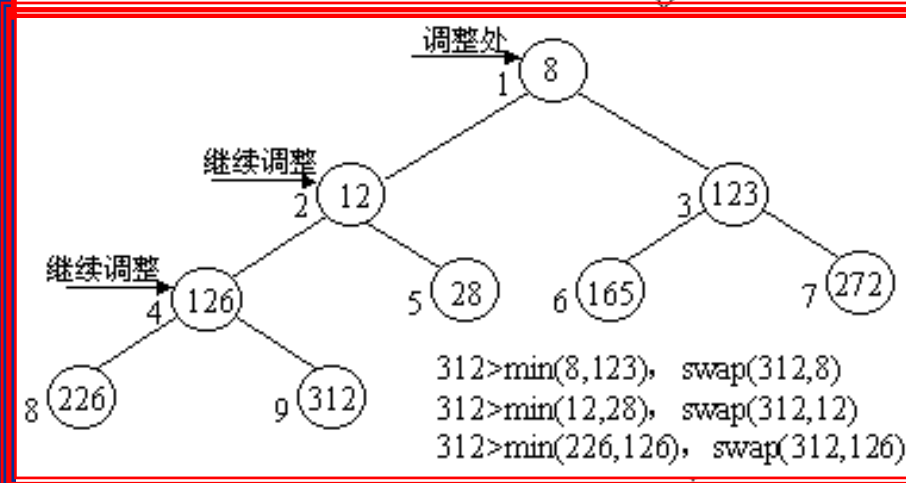
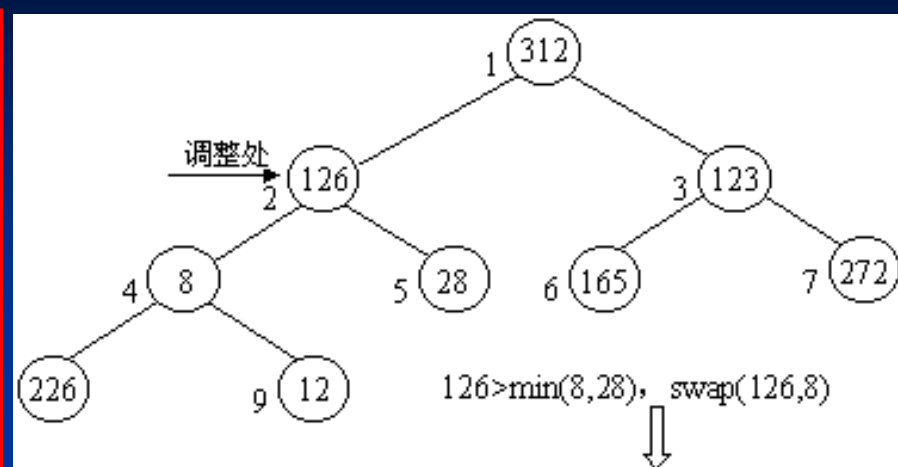
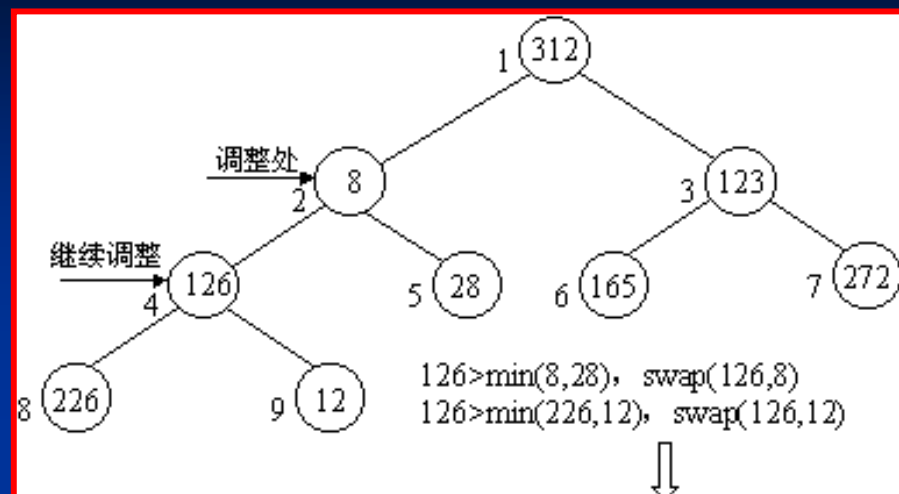
		312	126	272	226	28	165	123	8	12
下标	0	1	2	3	4	5	6	7	8	9



调整是从序号为1的结点处开始直到4($=n/2$), 还是从序号为4的结点开始, 然后对序号为3, 2, 1的结点依次进行呢?

应该从第4个结点开始, 依次使以第4个结点为根的子树变成堆, 直到以第1个结点为根的整个完全二叉树具有堆的性质, 则建堆完成。

建堆过程如下图所示




```

/*      筛选算法      */

void sift(table *tab,int k,int m)
{ int i,j,finished;
  i=k;j=2*i;tab->r[0].key=tab->r[k].key;finished=0;
  while((j<=m)&&(!finished))
  { if((j<m)&&(tab->r[j+1].key<tab->r[j].key)) j++;
    if(tab->r[0].key<=tab->r[j].key) finished=1;
    else { tab->r[i].key=tab->r[j].key; i=j;j=2*j; }
  }
  tab->r[i].key=tab->r[0].key;
}

```

算法10.6 筛选算法

通过筛选算法，可以将一个任意的排序码序列建成一个堆，堆的第1个元素，即完全二叉树的根结点的值就是排序码中最小的。将选出的最小排序码从堆中删除，对剩余的部分重新建堆，可以继续选出其中的最小者，直到剩余1个元素排序即告结束。

```
/*          堆排序算法          */  
  
void heapsort(table *tab)  
{ int i;  
  for(i=tab->length/2;i>=1;i--) sift(tab,i,tab->length); /*对所有元素建堆*/  
  for(i=tab->length;i>=2;i--) /* i表示当前堆的大小，即等待排序的元素的个数*/  
  { tab->r[0].key=tab->r[i].key;  
    tab->r[i].key=tab->r[1].key;  
    tab->r[1].key=tab->r[0].key;  
    /*上述3条语句为将堆中最小元素和最后一个元素交换*/  
    sift(tab,1,i-1);  
  }  
}
```

算法10.7 堆排序算法

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.4交换排序

交换排序的基本思路：

对待排序记录两两进行排序码比较，若不满足排序顺序则交换这对记录，直到任何两个记录的排序码都满足排序要求为止。

10.4.1 冒泡排序

冒泡排序

第1趟，对所有记录从左到右每相邻两个记录的排序码进行比较，如果这两个记录的排序码不符合排序要求，则进行交换，这样一趟做完，将排序码最大者放在最后一个位置；

第2趟对剩下的 $n-1$ 个待排序记录重复上述过程，又将一个排序码放于最终位置，反复进行 $n-1$ 次，可将 $n-1$ 个排序码对应的记录放至最终位置，剩下的即为排序码最小的记录，它在第1的位置处。

如果在某一趟中，没有发生交换，则说明此时所有记录已经按排序要求排列完毕，排序结束。

```
void bubblesort(table *tab)
{ int i,j,done;      i=1;done=1;
  while(i<=tab->length&&done)
    /*最多进行tab->length次冒泡，如没有发生交换则结束*/
    { done=0;
      for(j=1;j<=tab->length-i;j++)
        if(tab->r[j+1].key<tab->r[j].key)
          { tab->r[0].key=tab->r[j].key; tab->r[j].key=tab->r[j+1].key;
            tab->r[j+1].key=tab->r[0].key;  done=1;    }
        i++;
      }
    }
  /*算法10.8 冒泡排序算法*/
}
```

待排序的9个记录的排序码序列为{312, 126, 272, 226, 8, 165, 123, 12, 28}, 使用冒泡排序算法进行的排序过程如下图所示:

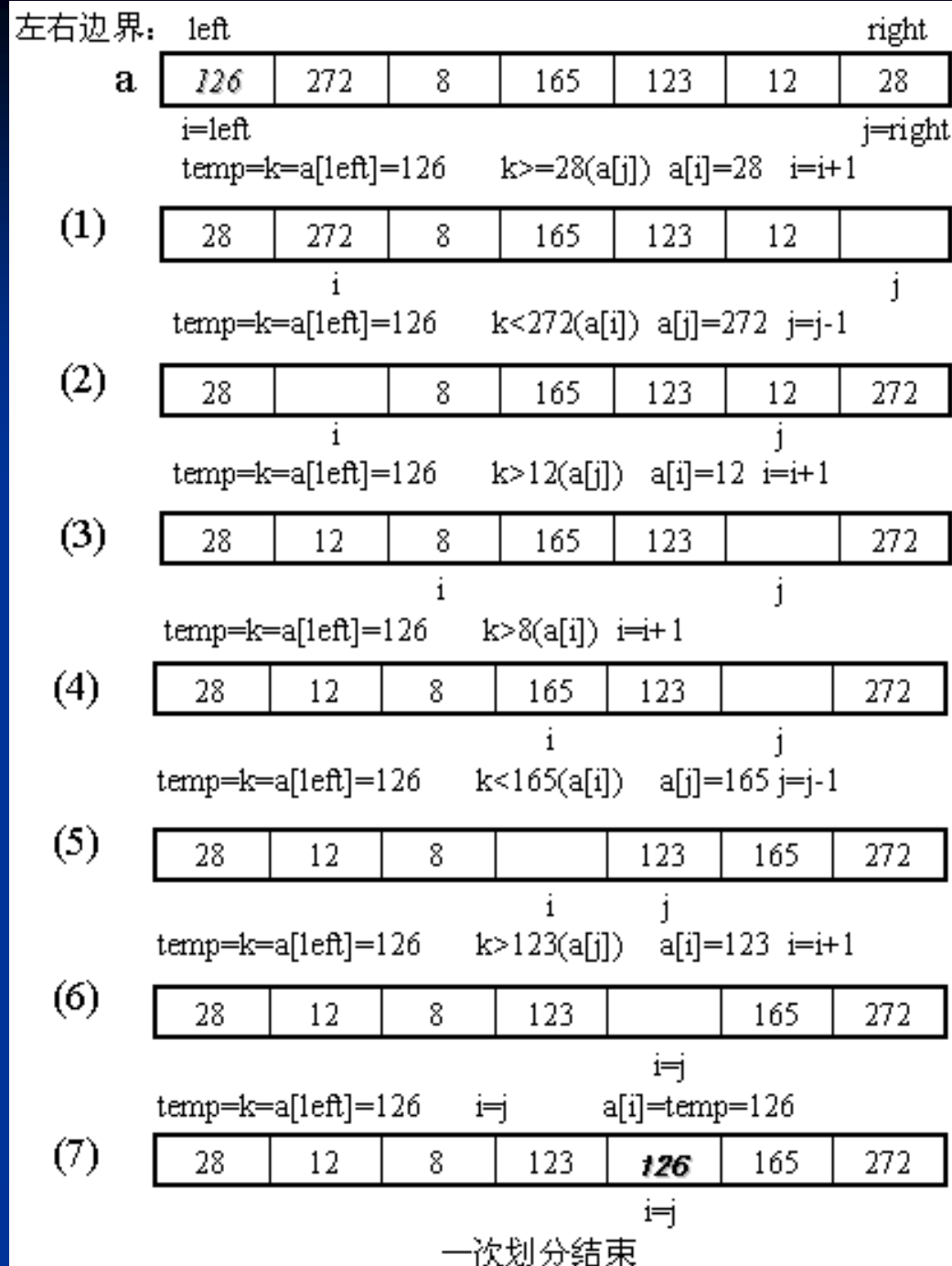
下标	排序码	第1趟后	第2趟后	第3趟后	第4趟后	第5趟后	第6趟后	没有发生 交换, 结束
1	312	126	126	126	8	8	8	8
2	126	272	226	8	126	123	12	12
3	272	226	8	165	123	12	28	28
4	226	8	165	123	12	28	123	123
5	8	165	123	12	28	126	126	126
6	165	123	12	28	165	165	165	165
7	123	12	28	226	226	226	226	226
8	12	28	272	272	272	272	272	272
9	28	312	312	312	312	312	312	312

10.4.2 快速排序

快速排序算法的基本思路是：

从 n 个待排序的记录中任取一个记录(不妨取第1个记录)，设法将该记录放置于排序后它最终应该放的位置，使它前面的记录排序码都不大于它的排序码，而后面的记录排序码都大于它的排序码，然后对前、后两部分待排序记录重复上述过程，可以将所有记录放于排序成功后的相应位置，排序即告完成。

设待排序的7个记录的排序码序列为
 {126, 272, 8, 165, 123, 12, 28}, 一次
 划分的过程如图所示



```

void quicksort(table *tab,int left,int right)
{ int i,j;
  if(left<right)
  { i=left;j=right;   tab->r[0].key=tab->r[i].key;
    do { while(tab->r[j].key>tab->r[0].key&& i<j) j--;
        if(i<j) { tab->r[i].key=tab->r[j].key;i++;}
        while(tab->r[i].key<tab->r[0].key&& i<j) i++;
        if(i<j) { tab->r[j].key=tab->r[i].key;j--;}
    }while(i!=j);
    tab->r[i].key=tab->r[0].key;
    quicksort(tab,left,i-1);   /*对标准值左边递归调用本函数*/
    quicksort(tab,i+1,right); /*对标准值右边递归调用本函数*/
  }
}

```

算法10.9 快速排序算法

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.5 归并排序

归并排序的基本思路是：一个待排序记录构成的文件，可以看作是有多个有序子文件组成的，对有序子文件通过若干次使用归并的方法，得到一个有序文件。归并是指将两个（或多个）有序子表合并成一个有序表的过程。将两个有序子文件归并成一个有序文件的方法简单，只要将两个有序子文件的当前记录的排序码进行比较，较小者放入目标——有序文件，重复这一过程直到两个有序子文件中的记录都放入同一个有序文件为止

归并排序需要调用两个操作，一个称之为**一次归并**，另一个称之为**一趟归并**。一次归并是指将一个数组中两个相邻的有序数组段归并为一个有序数组段，其结果存储于另一个数组中的操作。

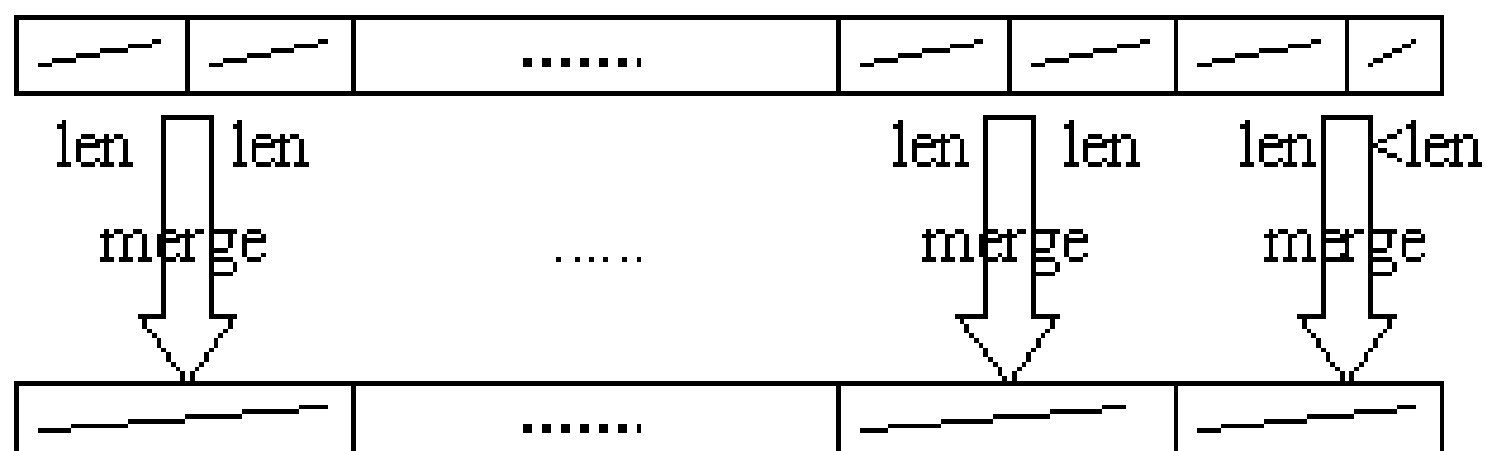
```

void merge(table *tabs, table *tabg, int u, int m, int v)
{
    int i, j, k, t;
    i=u;    /*i从第1段的起始位置开始，一直到最终位置m*/
    j=m+1;  /*j从第2段的起始位置开始，一直到最终位置v*/
    k=u;    /*k表示的是目标tabg的起始位置*/
    while(i<=m&& j<=v)
    { if(tabs->r[i].key<=tabs->r[j].key) { tabg->r[k].key=tabs->r[i].key; i++; }
      else { tabg->r[k].key=tabs->r[j].key; j++; }
      k++;
    }
    if(i<=m) for(t=i; t<=m; t++) tabg->r[k+t-i].key=tabs->r[t].key;
    else     for(t=j; t<=v; t++) tabg->r[k+t-j].key=tabs->r[t].key;
}

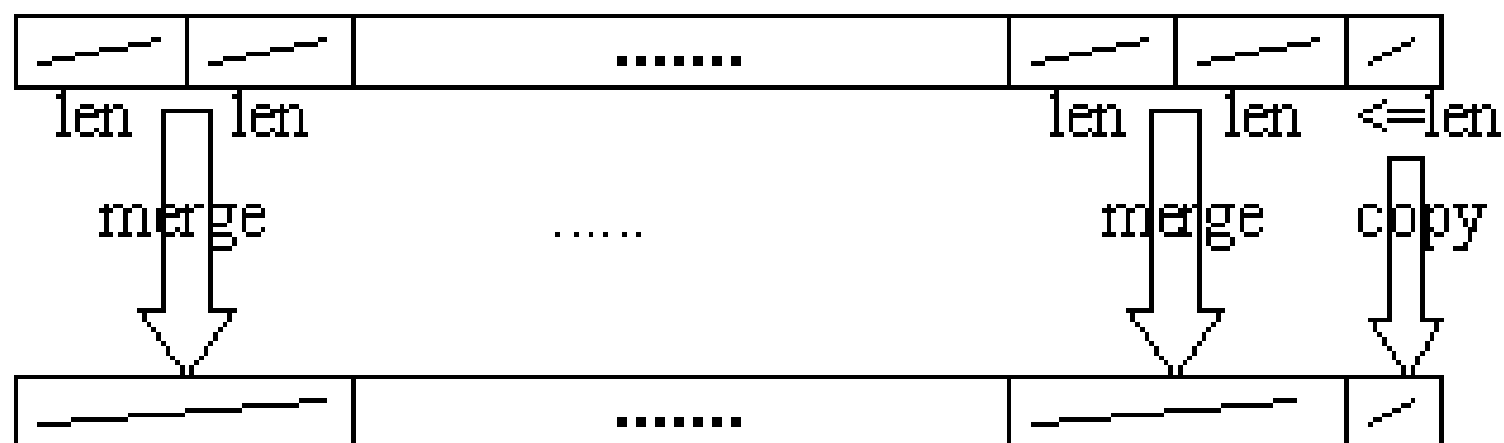
```

算法10.10 一次归并算法

一趟归并的图示：



(a) 最后对一个长度为 len ，另一个长度小于 len 的有序段归并



(b) 剩余的一个有序段元素直接拷贝到目标数组中

```

void mergepass(table *tabs, table *tabg, int len)
{ int i, j, n;
  n=tabg->length=tabs->length;
  i=1;
  while(i<=n-2*len+1)
  { merge(tabs, tabg, i, i+len-1, i+2*len-1); /*一次归并*/
    i=i+2*len; /*置下一个一次归并的起始位置*/
  }
  if(i+len-1<n)
    merge(tabs, tabg, i, i+len-1, n);
  else /*对剩下的1个长不超过len,终点为n的有序段进行处理*/
    for(j=i; j<=n; j++) tabg->r[j].key=tabs->r[j].key;
} /* 本算法结束后tabg中的有序段的长度为2*len */

```

算法10.11一趟归并算法


```
void mergesort(table *tab)
{
    int len;
    table temp;      /*中间变量*/
    len=1;           /*初始时有序段的长度为1*/
    while(len<tab->length) /*有序段的长度小于待排序元素的个数，继续归并*/
    {
        mergepass(tab,&temp,len); /*一趟归并，结果在temp中*/
        len=2*len;                /*有序段的长度翻倍*/
        mergepass(&temp,tab,len); /*一趟归并，结果在tab中*/
        len=2*len;                /*有序段的长度翻倍*/
    }
}
```

算法10.12归并排序算法

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

10.6 基数排序

基数排序（又称分配排序）是一种和前述各种方法都不相同的排序方法。前面介绍的排序方法是通过排序码的比较以及记录的移动来实现排序的，而基数排序没有作这两种操作，它不对排序码进行比较，是借助于多排序码排序的思想进行单排序码排序的方法。

10.6.1 多排序码的排序

多排序码排序的思想：一副游戏扑克牌中除大、小王之外的52张牌面的次序关系如下：

$\clubsuit 2 < \clubsuit 3 < \dots < \clubsuit A < \diamond 2 < \diamond 3 < \dots < \diamond A < \heartsuit 2 < \heartsuit 3 < \dots < \heartsuit A < \spadesuit 2 < \spadesuit 3 < \dots < \spadesuit A$

每一张牌有两个“排序码”：花色（梅花<方块<红心<黑桃）和面值（2<3<...<A），且花色的地位高于面值，即面值相等的两张牌，以花色大的为大。在比较两张牌的牌面大小时，先比较花色，若花色相同，则再比较面值，通常采用下面方法将扑克牌进行上述次序的排序：先将52张牌以花色分成为四堆，再对每一堆同花色的牌按面值大小整理有序。实际上，还可以用下面的方法对扑克牌排序：首先将52扑克牌按面值分成13堆，将这13堆牌自小至大叠在一起，然后再重新按不同花色分成4堆，最后将这4堆牌按自小至大的次序合在一起即可。这就是一个具有2个排序码的排序过程，其中分成若干堆的过程称为分配，从若干堆中自小到大叠在一起的过程称为收集。扑克牌排序使用了2次分配和2次收集操作。

10.6.2 静态链式基数排序

将扑克牌排序的第二种方法推广。可以得到对多个排序码排序算法。即若每个记录有 b 个排序码，则可采用扑克牌排序相同的思想，从最低位排序码 k^b 开始进行排序，再对高一位的排序码 k^{b-1} 进行排序，重复这一过程，直到对最高位 k^1 进行排序后便得到一个有序序列。

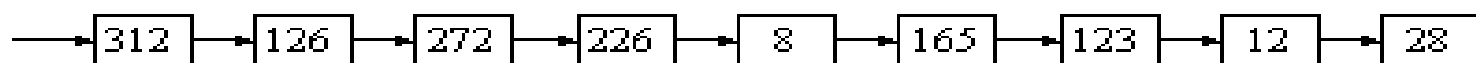
经常碰到的整数序列，可以把整数的个位数看作是最低位排序码，十位数是次低位排序码，依次类推，若待排序整数序列中最大整数序列中最大整数的位数为 b ，则整数序列的排序问题可用 b 个排序码的基数排序方法实现。在对某一位进行排序时，并不要进行比较，而是通过分配与收集来实现。

静态链式基数排序的思想是：

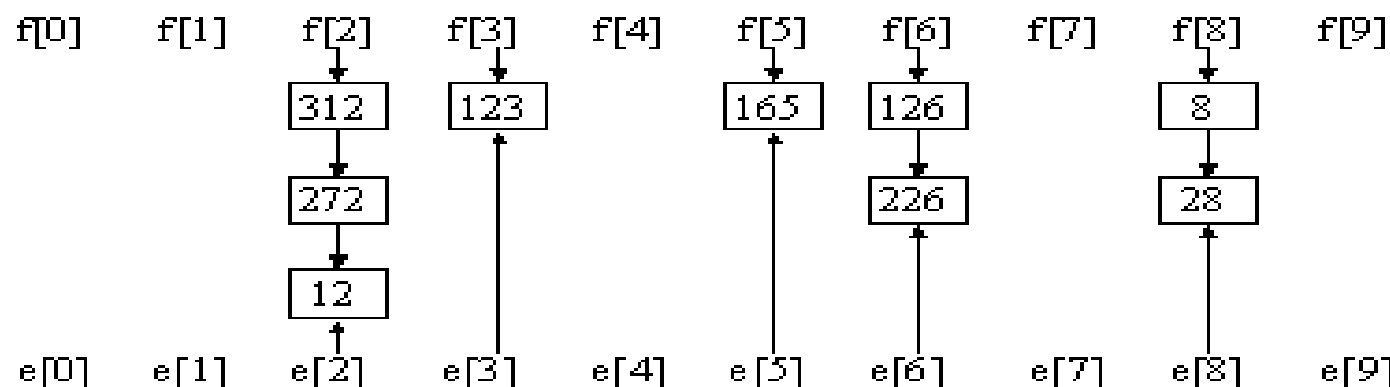
先用静态链表存储待排序文件中的 n 个记录，即建立一个静态单链表，表中每一个结点对应于一个记录，并用表头指针指向该静态单链表的表头结点。第一趟分配对最低位排序码（个位数）进行，修改静态链表的指针域，将 n 个结点分配到序号分别为0~9的10个链式队列中，其中每个队列中结点对应记录的个位数相同，用 $f[i]$ 和 $e[i]$ 分别作为第 i 个队列的队首和队尾指针；第一趟收集过程将这个10个队列中非空的队列依次合并在一起产生一个新的静态单链表，对这个新的静态单链表按十位数进行分配和收集，然后再依次对百位数、...、最高位数反复进行这样分配和收集操作，排序即可结束。

272,
数排

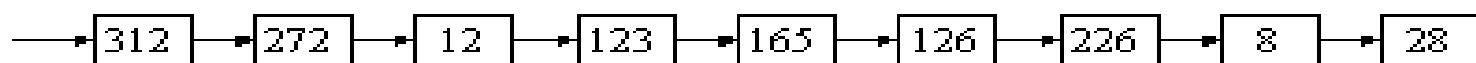
6,
式基



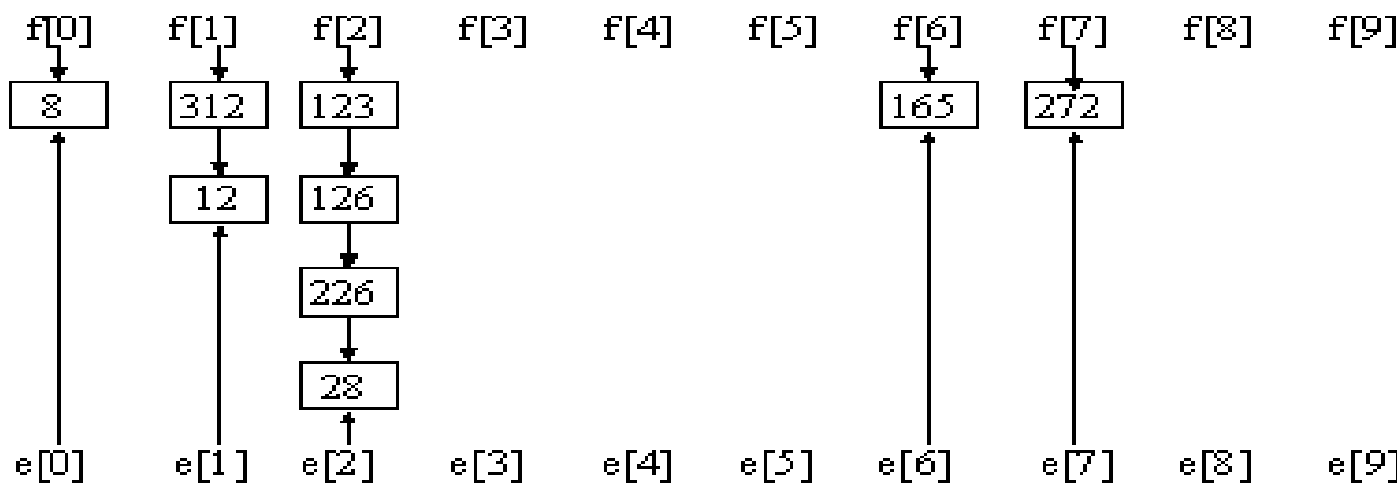
(a)初始状态



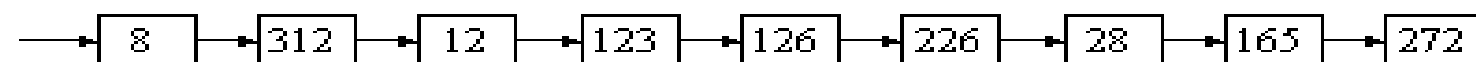
(b)第1次分配后的状态



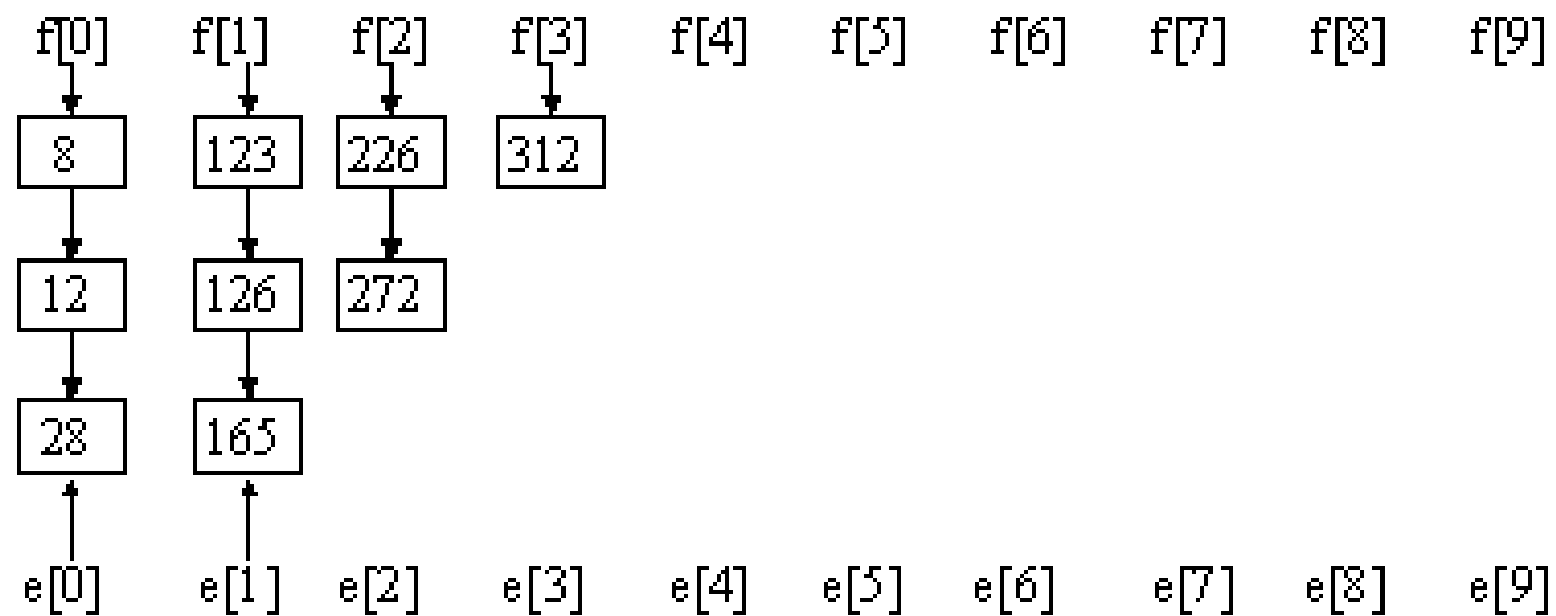
(c)第1次收集后的状态



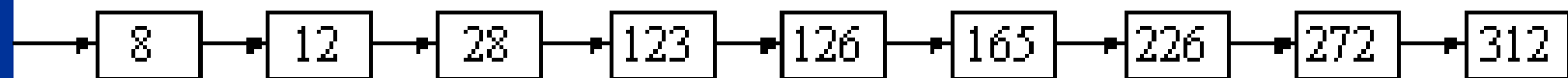
(d)第2次分配后的状态



(e)第2次收集后的状态



(f)第3次分配后的状态



(g)第3次收集后的状态，排序完成

第10章 内排序

10.1 排序的基本概念

10.2 插入排序

10.2.1 直接插入排序

10.2.2 二分法插入排序

10.2.3 表插入排序

10.2.4 Shell插入排序

10.3 选择排序

10.3.1 直接选择排序

10.3.2 树型选择排序(略)

10.3.3 堆排序

10.4 交换排序

10.4.1 冒泡排序

10.4.2 快速排序

10.5 归并排序

10.6 基数排序

10.6.1 多排序码的排序

10.6.2 静态链式基数排序

作业： 1, 3, 5, 6, 8, 10