数 据 结 构

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

http://staff.ustc.edu.cn/~xiaomj

李达天
联系电话： 18078908299
Email:
1137013891@qq.com

赵昱
联系电话： 13061366552
Email:
1070002758@qq.com

朱煜
联系电话： 18395951230
Email: zhuyuzy@mail.ustc.edu.cn

QQ群号: 783415818

课程简介

- **先修课程及条件**

  程序设计的经验、C、离散数学、概率分析

- **教材：**

  数据结构(C语言版)，严蔚敏，清华大学出版社

- **考核：**考试+作业+上机

- **参考书**

  C++ 数据结构，William Ford 清华影印版

  数据结构和程序设计， Robert,Kruse.2nd版

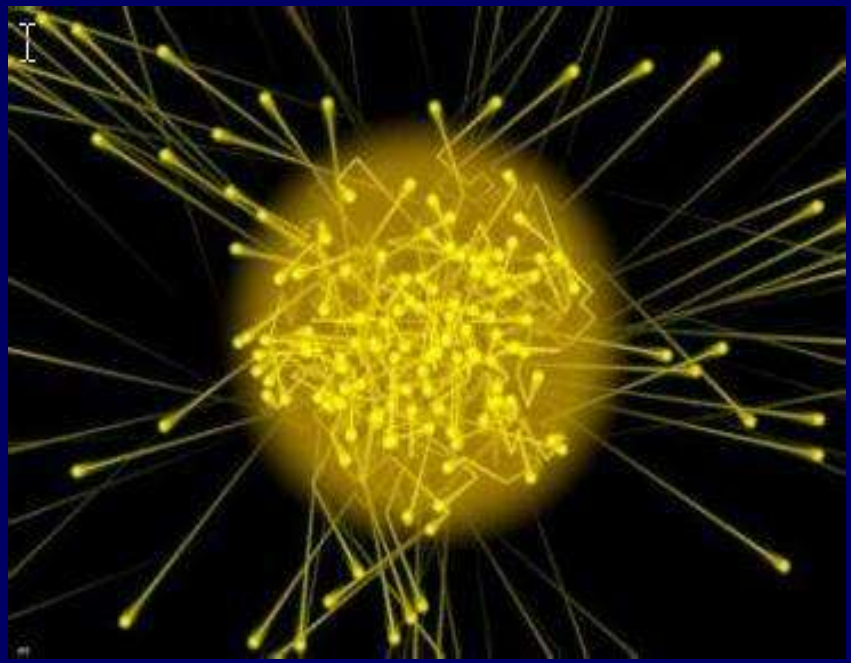
§ Ch.1 绪论

重要性

- 人类社会已步入信息社会
- 计算机不再仅仅局限于科学计算，已深入到社会生活的各个领域
- 当前热点领域
  - 智慧城市、智能工厂、物联网
  - 移动计算、云计算、边缘计算
  - 大数据与人工智能
  - 量子计算、并行与分布式计算


1100110110100101
10110100101
10100101
10100101
10111010101
10111011101
10100101
10100101
10100101
H


Revolution in Egypt
Joe Klein: What the U.S. should do
On the Street: Hope meets anxiety
Muslim Brotherhood: What it wants
Oscars: Portraits of star power
TIME
2045
The Year Man Becomes Immortal*
BY LEV GROSSMAN



§ Ch.1 绪论

重要性

- 计算机：硬件+软件

  算法+数据结构=程序设计（N.Wirth,84年图灵奖得主）

- 两个问题：信息的表示和处理

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

§ Ch.1 绪论

编写解决实际问题的程序的一般过程：

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

上面所列举的问题基本上由数据结构这门课程来回答。

例1：电话号码查询系统

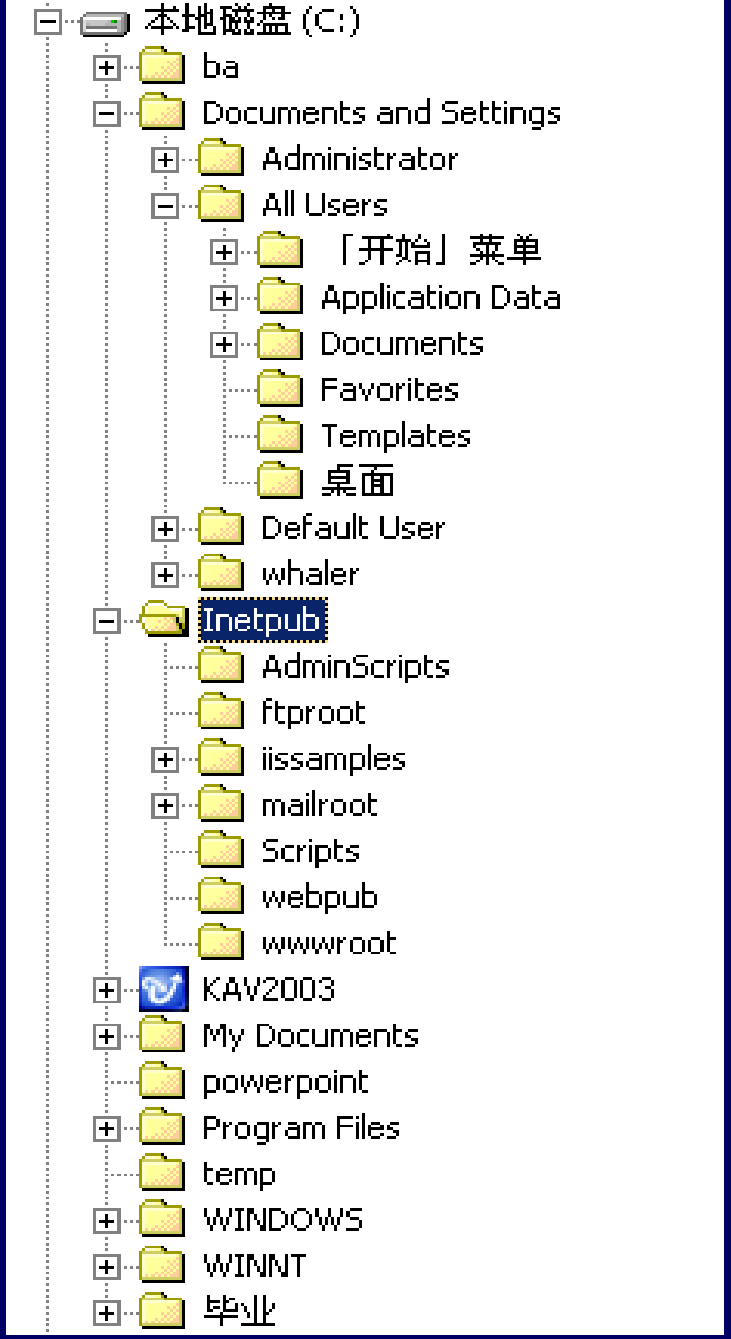
设有一个电话号码薄，它记录了N个人的名字和其相应的电话号码，假定按如下形式安排：（a_1，b_1），（a_2，b_2），...（a_n，b_n），其中a_i，b_i（i=1，2...n）分别表示某人的名字和电话号码。 本问题是一种典型的表格问题。如表所示，数据与数据成简单的一对一的线性关系。

姓名	电话号码
张三	13612345588
李四	13056112345
∘ ∘ ∘	∘ ∘ ∘

例2：磁盘目录文件系统

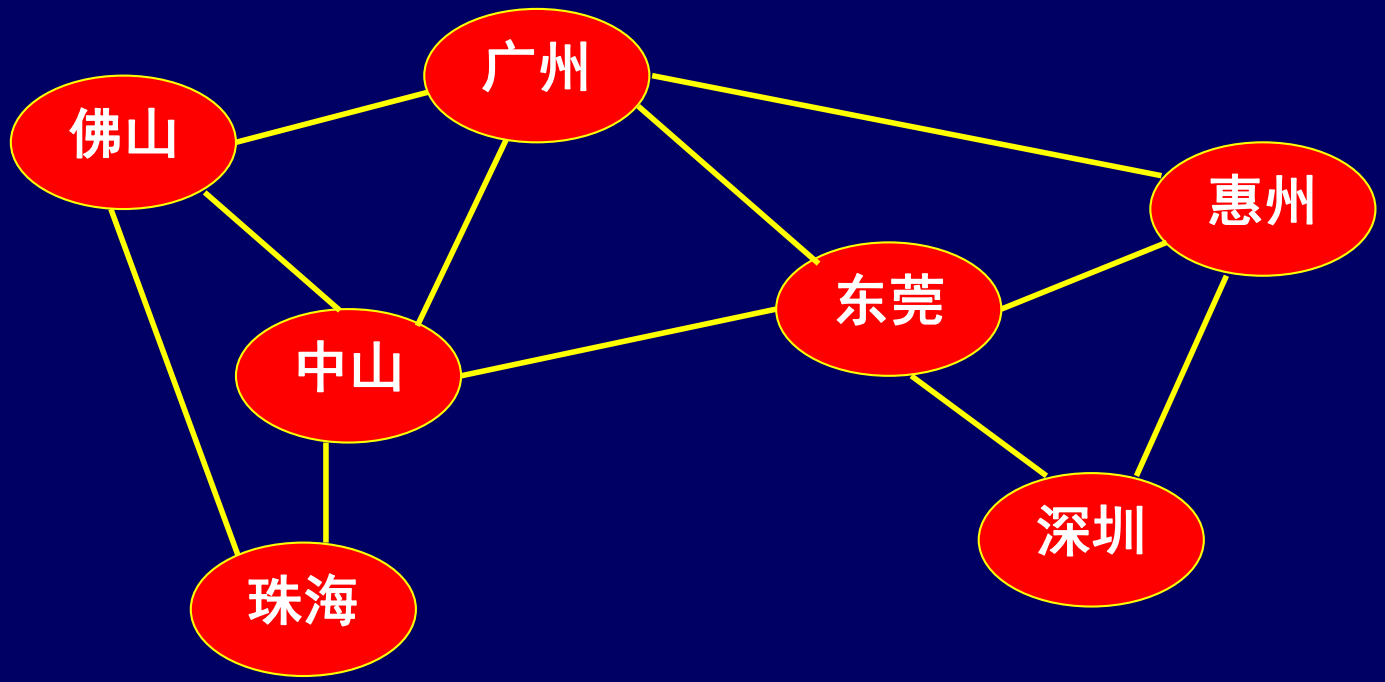
磁盘根目录下有很多子目录及文件，每个子目录里又可以包含多个子目录及文件，但每个子目录只有一个父目录，依此类推：

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



例3：交通网络图

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



网状结构

§1.1 基本概念和术语

- **数据：信息载体**

  客观事物的符号表示，能由计算机程序识别、存储和加工处理的符号集合

  所有能够数字化的信息均可认为是数据

- **数据元素：数据的基本单位**，在程序中通常作为一个整体来进行考虑和处理
  同义词：元素、结点、顶点、记录、对象、元组等

- **数据项：具有独立含义的最小标识单位**，客观事物某一方面特性的数据描述
  同义词：字段、域、属性等

§1.1 基本概念和术语

- **数据结构：** 相互之间存在一种或多种特定关系的数据元素的集合，即数据的组织形式
  - **数据的逻辑结构：** 数据元素之间的逻辑关系
  - **数据的存储结构：** 数据元素及其关系在计算机存储器内的表示
  - **数据的运算：** 对数据施加的操作

§1.1 基本概念和术语

数据结构举例

系别	姓名	职称	SCI	EI	经费
1	张明	教授	5	1	20
1	王华	教授	6	3	15
…					
23	李立	教授	1	6	60

行：结点(对象、记录、元组等)；

列：数据项(属性、域、字段等)

- 逻辑结构：开始结点？终端结点？内部结点？

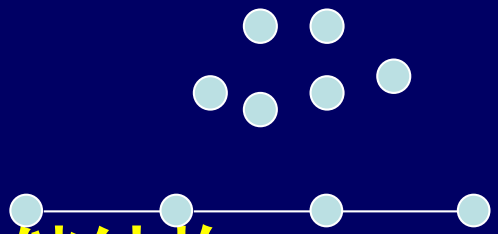
  结点之间的逻辑关系：有且仅有1个开始结点和1个终端结点，表中任1结点最多只有1个直接前驱和1个直接后继-----线性关系

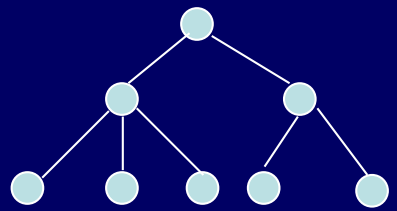
- 存储结构：用数组实现，还是用指针实现？
- 运算：创建、插入、删除、查找等

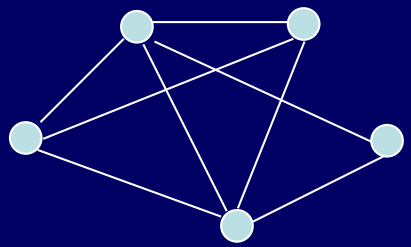
§1.1 基本概念和术语

逻辑结构

- 线性结构：任一结点最多只有1个直接前驱和1个直接后继
- 非线性结构：一结点可有多个直接前驱和多个直接后继
- 集合结构：元素间无关系，只有元素是否属于集合的关系







存储结构

(1) 顺序存储方法

逻辑上相邻的结点存储在物理位置上相邻的存储单元里

结点间的逻辑关系由存储单元的邻接关系来体现

借助于数组描述

应用于线性的数据结构，非线性的数据结构的线性化

§ 1.1 基本概念和术语

存储结构

(2) 链接存储方法

该方法不要求逻辑上相邻的结点在物理位置上亦相邻

借助于指针来表示结点间的逻辑关系

(3) 索引存储方法

在存储结点信息的同时，建立附加的索引表。

表中每项称为索引项 (关键字，地址)

稠密索引：每个结点对应1个索引项

稀疏索引：一组结点对应1个索引项

(4) 散列存储方法

根据结点的Key直接计算出该结点的存储地址。

§1.1 基本概念和术语

■数据结构的主要运算

(1) 建立(Create)一个数据结构;

(2) 消除(Destroy)一个数据结构;

(3) 从一个数据结构中删除(Delete)一个数据元素;

(4) 把一个数据元素插入(Insert)到一个数据结构中;

(5) 对一个数据结构进行访问(Access);

(6) 对一个数据结构(中的数据元素)进行修改(Modify);

(7) 对一个数据结构进行排序(Sort);

(8) 对一个数据结构进行查找(Search)。

§1.1 基本概念和术语

■ 数据结构3方面之联系

- ❖同一逻辑结构的不同存储结构，则用不同名称称谓

  例如，

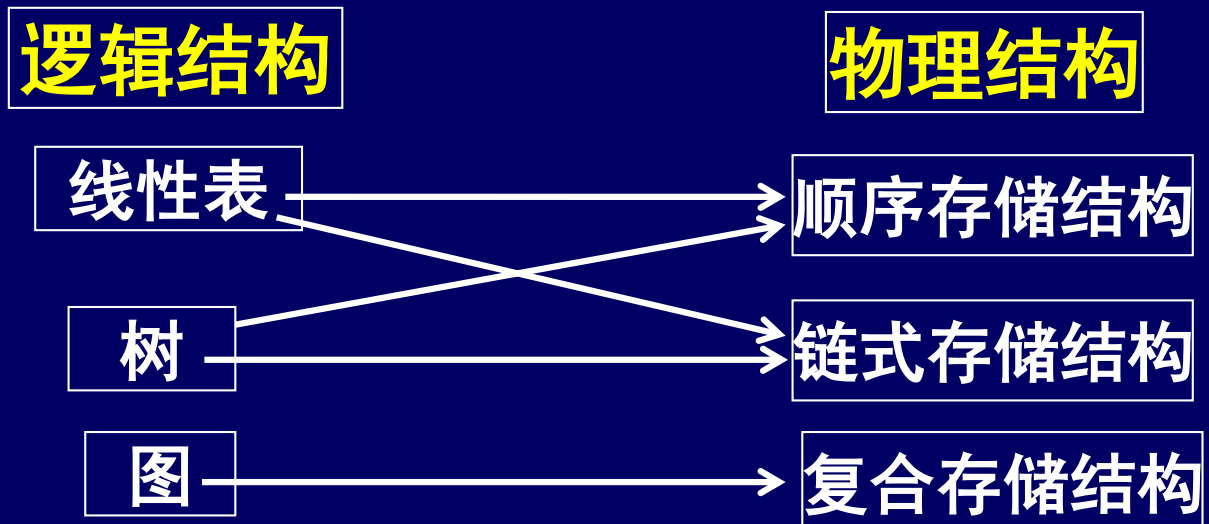
  线性表⇨顺序表，链表，散列表

- ❖运算不同，称谓也不同

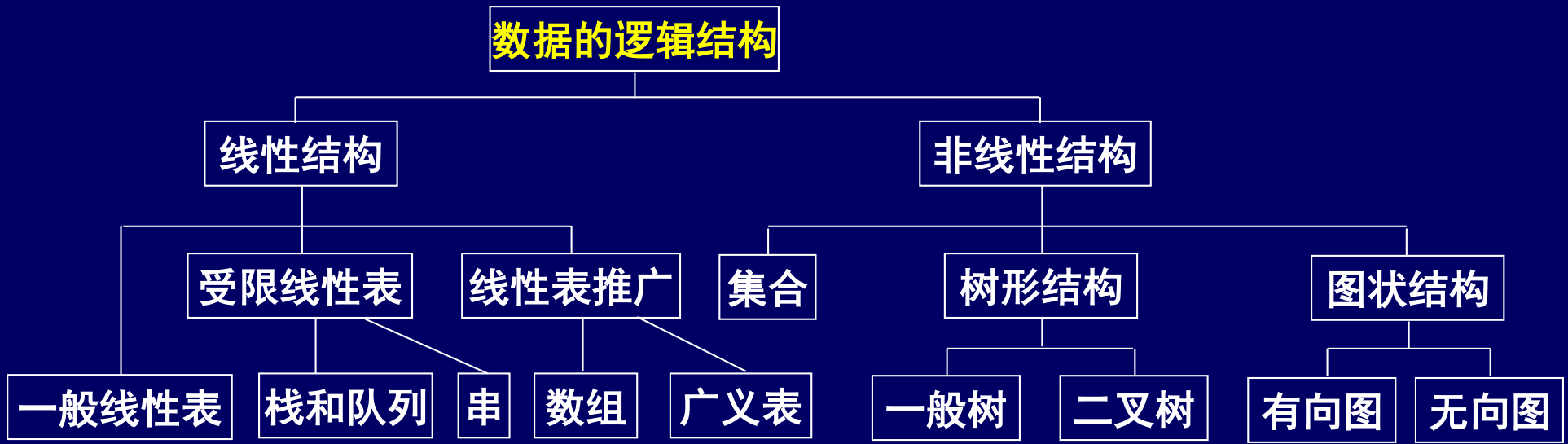
  线性表⇨栈、队列⇨ 顺序栈、顺序队列

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

- ❖在C语言中，用**一维数组**表示顺序存储结构；用**结构体类型**表示链式存储结构。



逻辑结构与所采用的存储结构



数据逻辑结构层次关系图

§1.1 基本概念和术语

- **数据类型：** 是一个值的集合以及定义在这些值上的一组操作的总称，可看作是高级语言提供的数据结构
  - **原子类型：** 值不可分，一般是基本的预定义类型

    int，char等，定义了“+”，“-”等
  - **结构类型：** 可供用户定义的新类型，构造类型、导出类型、派生类型等

    由基本类型组织而成，如数组、struct等

§1.1 基本概念和术语

抽象数据类型（Abstract Data Type）

抽象数据的组织和与之相关的操作，可看作是数据的逻辑结构及其在逻辑结构上定义的抽象操作

❖特点

①封装与隐藏：将数据和操作封装在一起，内部结构和实现细节对外屏蔽，实现信息隐藏

②抽象：用户只能通过ADT里定义的接口和说明来访问和操作数据。ADT的概念反映了程序设计的两层抽象：

- ☞概念层（抽象）---类定义
- ☞实现层---类实现
- ☞应用层----如main(){...}，通过操作对象解决实际问题

§1.1 基本概念和术语

抽象数据类型（Abstract Data Type）

```
ADT ADT_Name{
    Data: 数据结构(数据对象和数据关系)说明
    Operations：
        Constructor：//构造函数，创建对象实例
        Operation1：//用C++或C函数原型描述
            input：
            Preconditions：//初始条件，执行本操作前系统
                           //需满足的状态
            process：//对数据执行的操作
            output：//对返回数据的说明
            Postconditions：//执行本操作后系统的状态
        Operation2：
}
```

§1.1 基本概念和术语

例：抽象数据类型复数的定义

ADT Complex {

数据对象： D＝{e1,e2 | e1,e2∈RealSet }

数据关系： R1＝{<e1,e2> | e1是复数的实数部分,e2 是复数的虚数部分}

基本操作：

InitComplex(&Z, v1, v2)

操作结果：构造复数Z,其实部和虚部分别被赋以参数v1和v2的值。

DestroyComplex(&Z)

操作结果：复数Z被销毁。

GetReal(Z, &realPart)

初始条件：复数已存在。

操作结果：用realPart返回复数Z的实部值。

GetImag(Z, &ImagPart)

初始条件：复数已存在。

操作结果：用ImagPart返回复数Z的虚部值。

Add(z1,z2, &sum)

初始条件：z1, z2是复数。

操作结果：用sum返回两个复数z1、z2的和值。

} ADT Complex

§1.1 基本概念和术语

■抽象数据类型的表示与实现

通过程序设计语言中的类型来实现

❖C

- ➢抽象数据类型
- ➢数据对象　　结构体
- ➢基本操作　　函数

❖C++，Java

- ➢抽象数据类型　类class
- ➢数据对象　　数据成员
- ➢基本操作　　成员函数(方法)

ADT复数的C描述

```
typedef struct {
        double realpart;
        double imagpart;
}Complex;
boolean assign(Complex *pSrc, Complex *pDes){
        if (pSrc ==NULL || pDes==NULL ) return ERROR;
        pDes->realpart = pSrc->realpart;
        pDes->imagpart = pSrc->imagpart;
        return TRUE;
}
Complex *add(Complex *pZ1, Complex *pZ2){
        Complex *pSum = (Complex *)malloc(sizeof(Complex));
        if ( pSum==NULL )return NULL;
        pSum->realpart = pZ1->realpart + pZ2->realpart;
        pSum->imagpart = pZ1->imagpart + pZ2->imagpart;
        return pSum;
}
```

ADT复数的C++描述

```
class Complex{ // 类的声明
    protected:
        double realpart;
        double magpart;
    public:
        Complex( );
        Complex(double realVal，double imagVal);
        Complex(Complex& z){ assign(z) ;}
        ~Complex( );
        void assign(Complex& z);
        double getReal(void) const { return realpart;}
        double getImag(void) const { return imagpart;}
        friend Complex add(Complex& z1, Complex& z2);
};
```

```
// 类的实现部分
Complex::Complex(double realVal, double imagVal){
        realpart = realVal;
        imagpart= imagVal;
}
void Complex::assign(Complex& z){
        realpart = z.realpart;
        imagpart= z.imagpart;
}
Complex add(Complex& z1, Complex& z2){
        Complex sum(z1);
        sum.realpart += z2.realpart；
        sum.imagpart += z2.imagpart；
        return sum;
}
```

§1.2 算法描述与分析

算法

- **重要性**：数据的运算是通过算法描述的
- **定义**：非形式地说，算法是任意一个良定义的计算过程，它以一个或多个值作为输入，并产生一个或多个值作为输出。因此，一个算法就是一系列将输入转换为输出的计算步骤。
- **5要素**

输入、**输出**、**有穷性**（有穷步骤，每步时间有限）**确定性**（算法只有唯一执行路径）、**可行性**（所有操作可通过已经实现的基本运算有限次实现之）

算法与程序的联系与区别

§1.2 算法描述与分析

- **输入实例**

  一个问题的输入实例是由满足问题陈述中的限制、并能计算出该问题解的所有输入构成的。

- **算法描述**

  自然语言、数学语言、伪语言、程序语言等均可

  本课程以C为主，但不拘泥于细节

- **算法评价**

  正确性、可读性、健壮性、 **时空性能**

§1.2 算法描述与分析

■ 算法分析

算法效率的度量

- ❖事后统计：利用计算机内部的计时功能

缺陷

- ➢必须先运行依据算法编制的程序
- ➢时间统计量依赖于计算机的软硬件环境

```
double start, stop;
time (&start);
main process(n, …);
time (&stop);
double runTime = stop -start;
printf ("%d%d\n" , n, runTime );
```

§1.2 算法描述与分析

- 事前分析估算(时间)

  求出该算法的一个时间界限函数

  一些影响因素:

  - 算法的策略
  - 问题的规模
  - 书写程序的语言
  - 编译器产生的机器代码的质量
  - 机器执行指令的速度

§1.2 算法描述与分析

- **算法分析**

  算法的时间是每语句执行时间的总和

  每语句的执行时间=该语句执行次数（频度）

  X该语句执行1次的时间

  假定：每语句执行1次的时间为1个时间单位

  则：算法的执行时间=∑各语句频度

- **问题的规模（Size）n**

  输入量的大小，如…

- **时间复杂度：**算法的运行时间，是问题规模的函数

§1.2 算法描述与分析

时间复杂度

例1：矩阵乘法

```
for ( i=0; i＜n; i++)                                  //n+1
    for ( j=0; j＜n; j++) {                            //n(n+1)
        C[i] [j]=0;                                    //n²
        for ( k=0;k＜n; k++)                           //n²(n+1)
            C[i][j]=C[i][j]+A[i][k]*B[k][j];           //n³
    }
```

- 详细分析： $T(n)=2n^3+3n^2+2n+1$
- 简单分析：频度最大的语句

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}1 \quad =n^3$$

- **渐近时间复杂度**（简称时间复杂度）

从宏观上评价时间性能，只关心当n趋向无穷大时，T(n)的数量级(阶)

$$\lim_{n\to\infty} T(n)/n^3 =$$
$$\lim_{n\to\infty}(2n^3+3n^2+2n+1)/n^3$$
$$= 2$$

即：T(n) 和n^3同阶，数量极相同

记作：$T(n)=O(n^3)$

大O的数学定义

若T(n)和f(n)是定义在正整数集合上的两个函数，则T(n)=O(f(n))表示存在两个正的常数c和n_0，

使得当$n \geq n_0$时都满足$0 \leq T(n) \leq c \cdot f(n)$。

例２ {++x; s=0 ;}

将x自增看成是基本操作，则语句频度为１，即时间复杂度为O(1)。

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

§1.2 算法描述与分析

例３　for(i=1; i<=n; ++i)

{ ++x; s+=x ; }

语句频度为：2n，其时间复杂度为：O(n)，即为线性阶。

例４　for(i=1; i<=n; ++i)

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

{ ++x; s+=x ; }

语句频度为：$2n^2$，其时间复杂度为：$O(n^2)$，即为平方阶。

§1.2 算法描述与分析

例５　for(i=2;i<=n;++i)

　　for(j=2;j<=i-1;++j)

　　{++x; a[i,j]=x; }

语句频度为：(1+2+3+…+n-2)×2=(1+n-2) ×(n-2)

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

∴时间复杂度为O(n^2)，即此算法的时间复杂度为平方阶。

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

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

§1.2 算法描述与分析

以下六种计算算法时间的多项式是最常用的。其关系为：

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

- 指数时间的关系为：

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

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

- 有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。

例1：素数的判断算法。

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

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

例2：冒泡排序法。

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

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

§1.2 算法描述与分析

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

该存储空间一般包括三个方面：

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

一般地，算法的空间复杂度指的是辅助空间。

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

作 业

1　数据的逻辑结构？数据的物理结构？逻辑结构与物理结构的区别和联系是什么？

2　分析以下程序段的时间复杂度，请说明分析的理由或原因。

(1)

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

(2)

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

(3) 递归函数

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

3. 按增长率由小至大的顺序排列下列各函数：

$$2^{100}, (3/2)^n, (2/3)^n, n^n, \sqrt{n}, n!, 2^n, \lg n, n^{\lg n}, n^{3/2}$$

4. 有时为了比较两个同数量级算法的优劣，须突出主项的常数因子，而将降低次项用大“O”记号表示。例如，设 $T_1(n)=1.39n\lg n+100n+256=1.39n\lg n+O(n)$，$T_2(n)=2.0n\lg n-2n=2.0n\lg n+O(n)$，这两个式子表示，当n足够大时$T_1(n)$优于$T_2(n)$，因为前者的常数因子小于后者。请用此方法表示下列函数，并指出当n足够大时，哪一个较优，哪一个较劣？

(1) $T_1(n)=5n^2-3n+60\lg n$　　(2) $T_2(n)=3n^2+1000n+3\lg n$

(3) $T_3(n)=8n^2+3\lg n$　　(4) $T_4(n)=1.5n^2+6000n\lg n$

数 据 结 构

Ch.2 线性表

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

http://staff.ustc.edu.cn/~xiaomj

§2.1 线性表的逻辑结构

- **线性表：** 由n(n≥0)个结点a_1, …, a_n组成的有限序列
  - 记作：L = (a_1, a_2, …, a_n),
  - 属性：长度----结点数目n，n=0时为空表

    a_i----一般是同一类型

§2.1 线性表的逻辑结构

逻辑特征

当L≠Φ时，

① 有且仅有1个开始结点a_1和1个终端结点a_n，a_1仅有一直接后继，a_n仅有一直接前驱

② 内部结点$a_i(2\le i\le n-1)$均有一直接前驱a_{i-1}和一直接后继a_{i+1}

结点之间的**逻辑关系**是**由位置次序决定**的，a_i出在表的第i个位置。该关系是线性的(全序的)，故称L为线性表。

§2.1 线性表的逻辑结构

- **举例：**
  - 英文字母表(A, B, …, Z)
  - 扑克牌点数(2, 3, … , 10, J, Q, K, A)
  - 学生成绩表 等

a_i----抽象符号。具体应用可能是一个结构类型的对象

线性表是一种相当灵活的数据结构，其长度可根据需要增长或缩短

- **基本运算：**

InitList(L), ListLength(L), GetNode(L, i) …等

复杂运算可通过基本运算去构造

§ Ch.2 线性表

线性表的抽象数据类型定义

ADT List{

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

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

基本操作：

InitList(&L)

操作结果：构造一个空的线性表L；

ListLength(L)

初始条件：线性表L已存在；

操作结果：返回线性表L中的节点个数，即表长；

§ Ch.2 线性表

….

GetElem(L, i, &e)

初始条件：线性表L已存在，1≦i≦ListLength(L)；

操作结果：用e返回L中第i个数据元素的值；

ListInsert (L, i, &e)

初始条件：线性表L已存在，1≦i≦ListLength(L) ；

操作结果：在线性表L中的第i个位置插入元素e；

…

} ADT List

§2.2 线性表的顺序存储结构

§2.2.1 顺序表

- **定义：**将线性表中结点按逻辑次序依次存储在一组地址连续的存储单元里。由此存储的L称为顺序表。

- **地址计算：**

设结点大小为ℓ个单元，其中第i个单元为该结点地址

$$Loc(a_i) = Loc(a_1) + (i-1) * \ell$$

↑

L的基地址

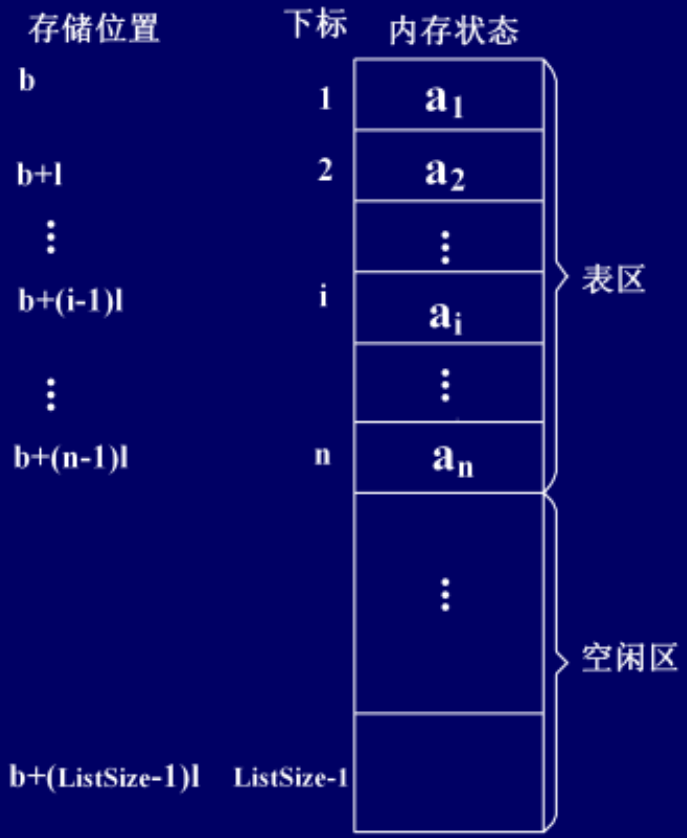
§2.2.1 顺序表

■特征：

① 随机存取 (直接存取)

② 只需存储结点。无需用辅助空间存储逻辑关系。但空闲大小不易确定，易浪费空间

③ 静态分配（当用向量实施时）亦可动态申请空间，但copy成本大（扩充表时）。



§2.2.1 顺序表

- **实现：用向量实现**

```
#define ListSize 100; //假定
typedef int DataType; //依具体使用定
typedef struct {
DataType data[ListSize]; //存放结点
int length; //当前表长n
}Seglist;
```

§2.2.2 顺序表上实现的基本运算

1.插入

- **基本思想：**

在L的第i个位置上插入一新结点x($1 \le i \le n+1$)。

$$(a_1, \ldots, a_{i-1}, \underset{\downarrow}{a_i}, a_{i+1}, \ldots, a_n) \rightarrow (a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_n)$$

（x 插入在 a_i 之前）

为保持结点的物理顺序和逻辑顺序一致，须：

① 将$a_n, \ldots, a_i$依次后移1位，空出ith位置

② 插入x

③ 表长加1

§2.2.2 顺序表上实现的基本运算

算法：

```
void InsertList(SegList *L, DataType x, int i){
    int j; //注意c数组1st位置的下标为0. ai的位置是data[i-1].
    if (i<1 || i>L->length+1)   //1≤i≤n+1是合法插入位置
       Error("Position error!");
    if (L->length >= ListSize) Error ("Overflow");    //溢出
    for (j = L->length-1; j>=i-1; j--)
       L->data[j+1] = L->data[j];    //结点后移
    L->data[i-1] = x;    //插入x
    L->length++;    //长度加1
};
```

§2.2.2 顺序表上实现的基本运算

■ 分析：

循环后移，移动节点数为n-i+1，时间与 {规模n, 插入位置i} 相关

① **最好情况：** 当i=n+1，不移动　O(1)

常数时间　$O(n^0)$与n无关

② **最坏情况：** 当i=1，全部后移　O(n)

③ **平均性能：**

设任何合法位置上插入的概率相等：$Pi = \frac{1}{n+1}$ (位置i插入的概率)

当位置i确定是，后移次数亦确定：n-i+1. 故表中平均移动结点为：

$$E_{IS}(n) = \sum_{i=1}^{n+1} Pi(n-i+1) = \sum_{i=1}^{n+1} \frac{n-i+1}{n+1} = n/2 = O(n)$$

即插入式，平均移动表中一半结点

§2.2.2 顺序表上实现的基本运算

2.删除

- **基本思想：**

在L中删除ith结点 ($1 \le i \le n$)。

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

为保持物理与逻辑顺序一致，须：

① 前移：将$a_{i+1}, \ldots, a_n$依次前移1位，填补删除a_i留下的空间

② 表长减1

§2.2.2 顺序表上实现的基本运算

算法：

```
void DeleteList(SegList *L, int i){
    int j,  n=L->length;
    if (i<1 || i>n) Error("Position error!");    //非法位置
    for (j = i; j<=n-1; j++)
      L->data[j-1] = L->data[j];    //结点后移
    L->length--;    //长度减1
};
```

§2.2.2 顺序表上实现的基本运算

分析：

前移次数与n和i相关，移动节点数n-i。

① 最好情况：当i=n，不移动　O(1)

② 最坏情况：当i=1，前移n-1次　O(n)

③ 平均性能：

$$E_{DE}(n)=\sum_{i=1}^{n}Pi(n-i)=\sum_{i=1}^{n}\frac{n-i}{n}=\frac{n-1}{2}=O(n)$$

§2.3 线性表的链式存储结构

顺序表

- 缺点：移动节点，不利于动态插入和删除
- 优点：随机存取，得到第i个结点的时间为O(1)与表长和位置无关

§2.3.1 单链表（线性链表）

存储方法：

用一组任意存储单元来存放结点（可连续，亦可不连续）；

为表示逻辑关系，须存储后继或前驱信息

§2.3.1 单链表（线性链表）

■结点结构

数据域 指针域（链域）

next指向该结点的后继（的地址）

■表示

① 开始结点无前驱，用头指针表示

② 终端结点无后继，next域为空（NULL）

§2.3.1 单链表（线性链表）

逻辑状态

头指针唯一确定一个单链表

存储状态　见图2.5

■特征

① 非顺序存取

② 用指针表示结点间逻辑关系

③ 动态分配

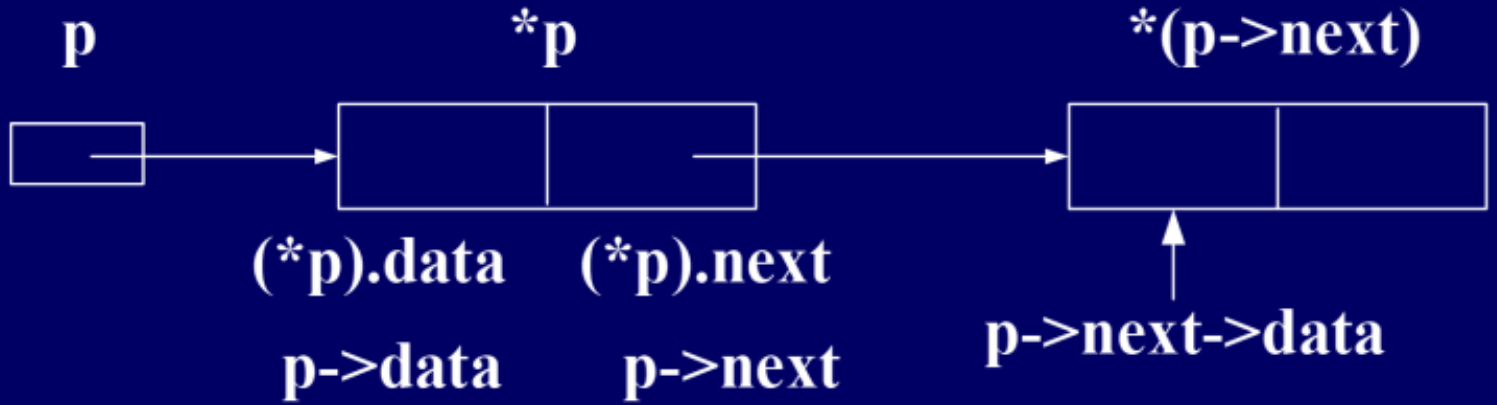
§2.3.1 单链表（线性链表）

- 实现：

```
typedef struct node{    //结点类型定义
    DataType data;    //数据域
    struct node *next;    //指针域
}ListNode;
typedef ListNode *LinkList;    //链表类型定义
ListNode *p;    //结点定义
LinkList head;    //链表头指针定义
```

§2.3.1 单链表（线性链表）

❖结点变量和指针变量



指针变量p：值为结点地址

结点变量*p：值为结点内容

动态申请，垃圾收集

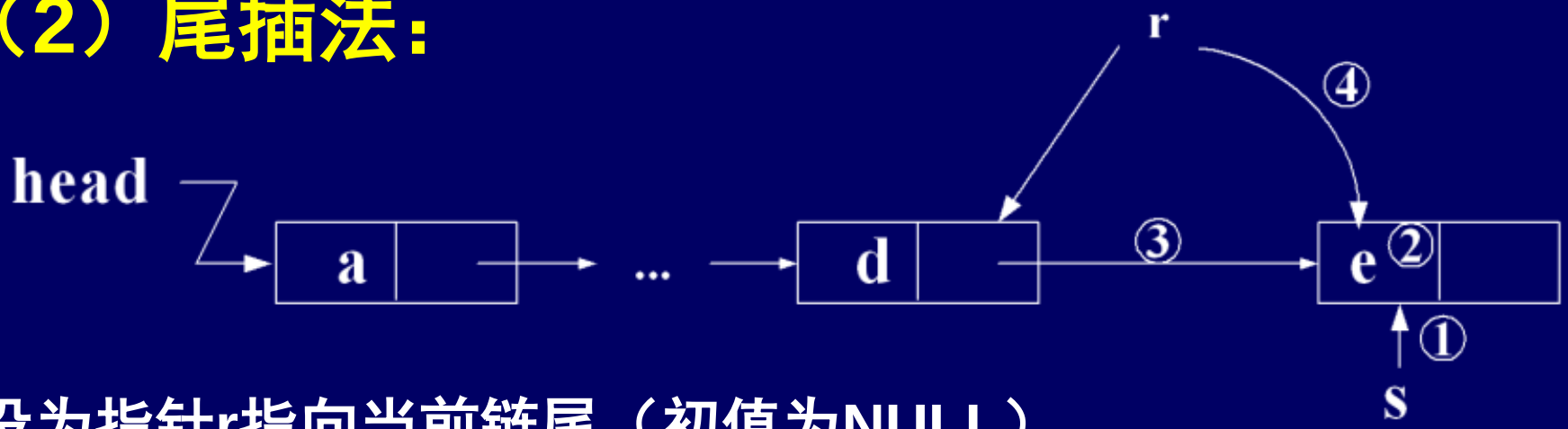
§2.3.1 单链表（线性链表）

1.建立单链表

- （1）头插法：
  - 从空表开始，重复读入数据：申请新结点、插入表头，直至输入结束。
  - 表次序与输入次序相反。
  - 处理自然简单

§2.3.1 单链表（线性链表）

（2）尾插法：



设为指针r指向当前链尾（初值为NULL）

① 申请新结点s

② 将读入数据写入

③ 新结点链到表尾（应注意边界条件）

④ 修改尾指针r

§2.3.1 单链表（线性链表）

为简便起见，设结点数据类型DataType为char，输入字符，换行符结束

```
LinkList  CreateList(void){
    //ch, head, r为局部量
    head = r = NULL;    //开始为空表
    while ((ch = getchar()) != '\n'){
      s = (ListNode*)malloc(sizeof(ListNode));    // ①
      s->data = ch;    // ②
      if (head == NULL)    //插入空表
          head = s;    //新结点插入空表（边界），r为空
```

§2.3.1 单链表（线性链表）

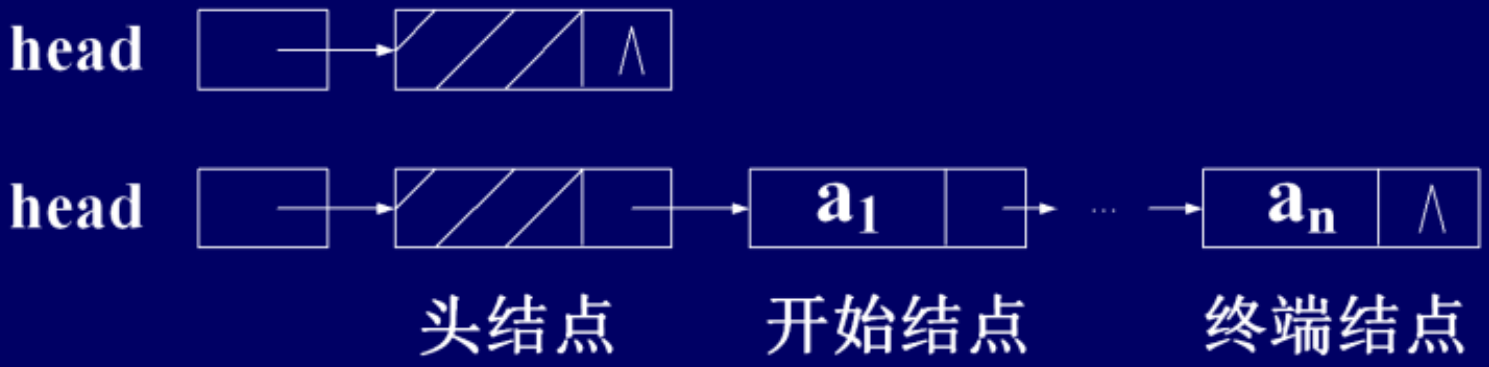
```
        else
            r->next = s;    // ③
        r = s;    // ④
        }
        if (r != NULL) r->next = NULL;    //非空表，终端结点指针为空无后继
        return head;
}
```

❖边界条件处理：

空表和非空表处理不一致

简化方法：引入头结点（哨兵），其中data域可做它用（如表长度）

§2.3.1 单链表（线性链表）




```
LinkList  CreateList(void){
   //用尾插法建立带头结点的单链表
   char ch;
   LinkList  head = (Linklist)malloc(sizeof(ListNode));   //生成头结点
   ListNode *s, *r;
   r = head;   //为指针初始指向头结点
   while ((ch = getchar()) != '\n') {
      s = (ListNode*)malloc(sizeof(ListNode));
      s->data = ch;
```

§2.3.1 单链表（线性链表）

```
        r->next = s;
        r = s;
    }
    r->next = NULL;    //终端结点指针置空，或空表时头结点指针置空
    return head;
}
```

Note：为简化起见，，申请结点时不判是否申请到

- 时间： O(n)

§2.3.1 单链表（线性链表）

2.查找

① 按值查找

找链表中关键字为k的结点

② 按序号查找

- 合法位置 1≤i≤n.　头结点可看作第0个结点
- 返回第i个结点的地址，即找到第i-1个结点，返回其next域
- 思想：

  顺链扫描：p----当前扫描到的结点，初始指向头结点

  j----计数器。累加扫描到的结点，初始值为0

  每次加1，直至 j=i为止，p指向ith结点

- 算法

§2.3.1 单链表（线性链表）

```
ListNode  *GetNode(LinkList head, int i){
    //在链表（有头结点）中查找ith结点    找到(0≤i≤n)则返回该结点的存储
    //位置，否则返回NULL
    int j;    ListNode *p;
    p = head;    j = 0;    //头结点开始扫描
    while (p->next && j<i) {//顺链扫描，至p->next为空或j=i为止
        p = p->next;    j++;
    }
    if (i == j) return p;    //找到
    else return NULL;    //当i<0或i>n时未找到
}
```

- 时间分析

循环终止条件是搜索到表尾或j=i。复杂度最多为i，与查找位置相关。

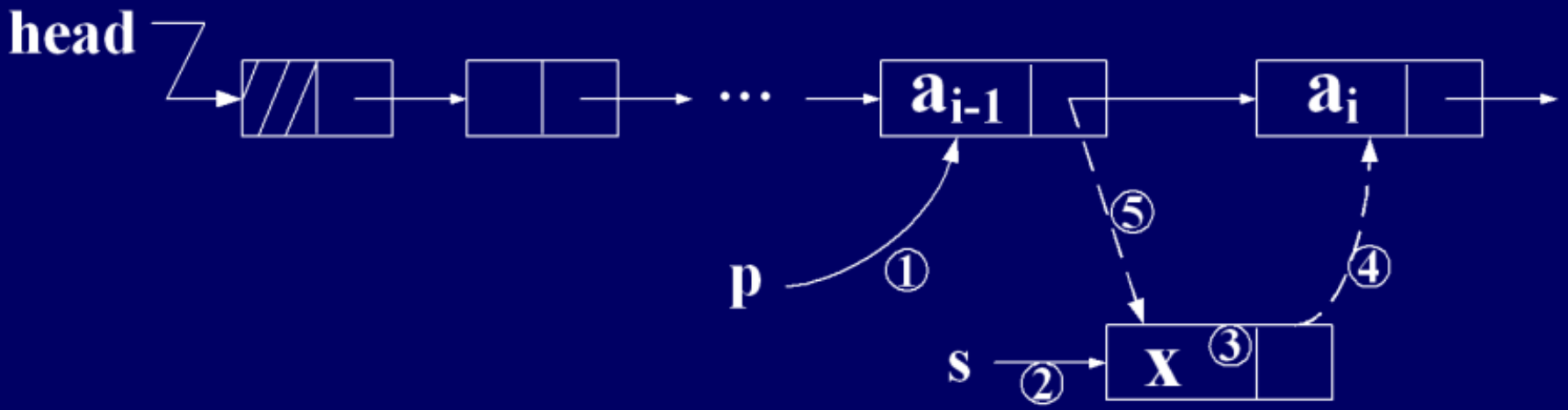
$$\sum_{i=0}^{n} i/(n+1) = n/2 = O(n)$$

//i=0，找头结点

§2.3.1 单链表（线性链表）

3.插入

- **问题：**将值为x的结点插到表的第i个结点位置上。即在a_{i-1}和a_i之间插入。故须找到第a_{i-1}个结点的地址p，然后生成新结点*s，将其链到a_{i-1}之后，a_i之前。



§2.3.1 单链表（线性链表）

```
void InsertList (LinkList head, DataType x, int i) {
//带头结点 1≤i≤n+1
    ListNode *p;
    p = GetNode(head, i-1);    //寻找第i-1个结点①
    if (p== NULL)       // i<1或i>n+1时插入位置i有错
       Error("position error");
    s = (ListNode *)malloc(sizeof (ListNode));     //②
    s->data=x;        //③
    s->next=p->next;    //④
    p->next=s;        //⑤
}
```

思考：若无头结点，边界如何处理？

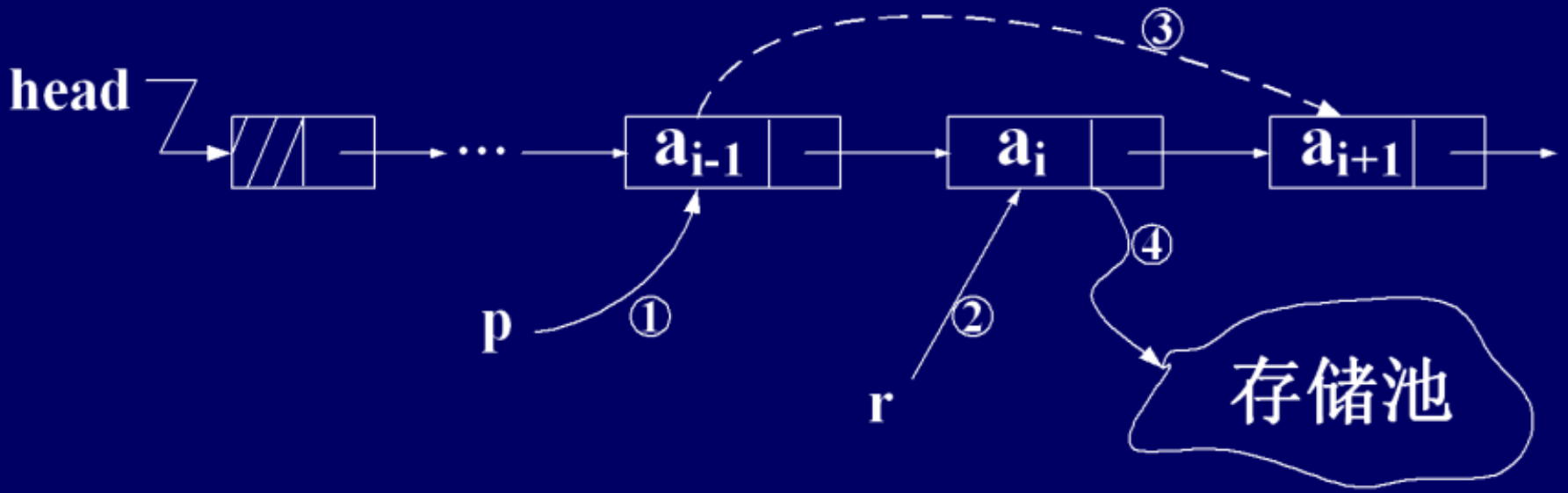
- 时间：主要是GetNode O(n) 合法位置　1≤i≤n+1

GetNode：　0≤i≤n

§2.3.1 单链表（线性链表）

4.删除

删ith结点。首先找a_{i-1}。



§2.3.1 单链表（线性链表）

```
void DeleteList (LinkList head, int i){
    //合法位置是1 ≤ i ≤ n
    p = GetNode (head, i-1);    //①
    if (!p || !(p->next))    // i<1或i>n时删除位置有错
        Error("position error");
    r = p->next;    //② 令r指向被删结点a_i
    p->next = r->next;     // ③ 将a_i从链上摘下
    free(r);           // ④ 释放a_i
}
```

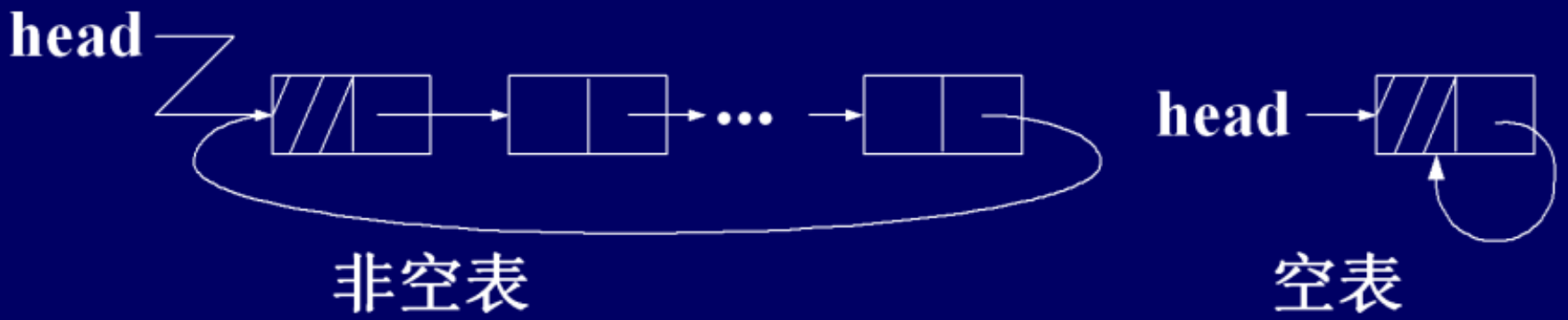
§2.3.1 单链表（线性链表）

- 正确性分析
  - ① i<1时，GetNode中的参数<0。此时返回p为空
  - ② i=n+1时，GetNode中参数=n。返回终端结点，此时p->next为空
  - ③ i>n+1时，GetNode返回p为空
- 时间 O(n)

结论：链表上插、删无须移动结点，仅需修改指针

§2.3.2 循环链表（单）

- **首尾相接：**不增加存储量，只修改连接方式



- **优点：**从任一结点出发可访问到表中任一其它结点，如找前驱（O(n)）
- **遍历表的终止方式：**

p为空或p->next为空 ⟶ p=head或p->next=head

§2.3.2 循环链表（单）

- **仅设尾指针更方便：**访问开始结点和终端结点的时间均为O(1)。

  例如：合并两个单循环链表的时间为O(1)，但用头指针表示时：

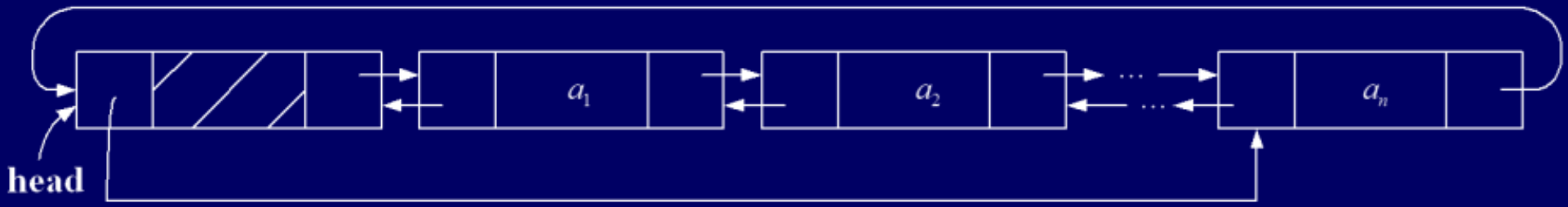
  T(m,n)=O(max(m,n))或O(min(m,n))

§2.3.3 双链表（循环）

单链表只有后向链，找一结点的后继时间为O(1)，但找前驱费时，若经常涉及找前驱和后继，则可选双链表。

（a） 结点结构

（b） 空的双循环链表



（c） 非空的双循环链表

§2.3.3 双链表（循环）

结构特征

对称结构：若p不空，则

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

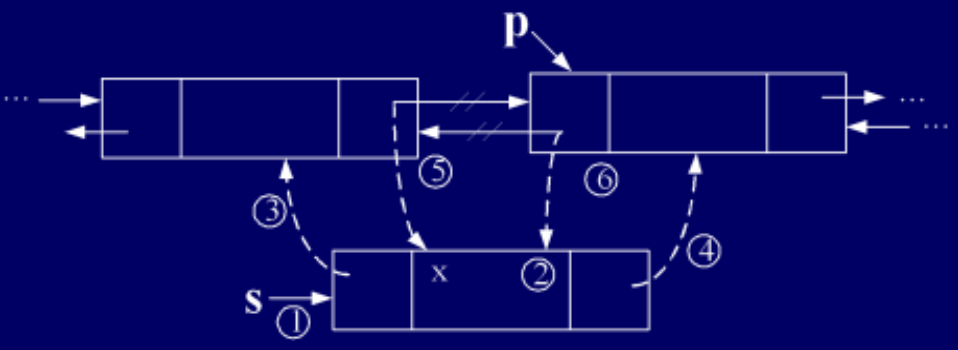
优点：

① 找一结点前驱和找一结点后继同样方便

② 删*p本身和删*p的后继同样方便

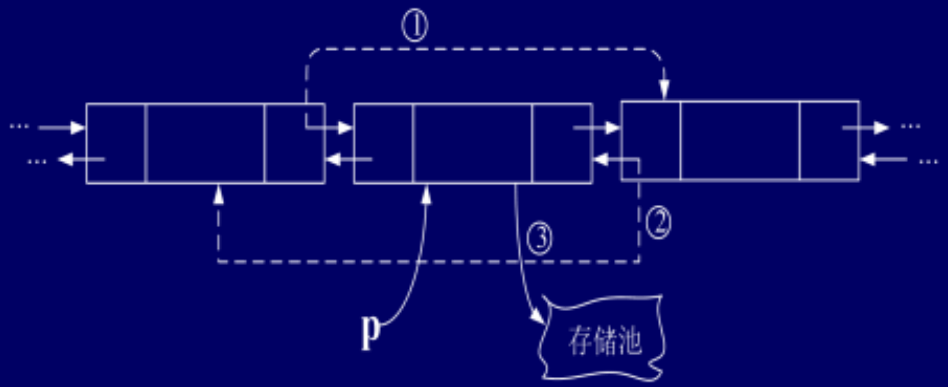
§2.3.3 双链表（循环）

例1：前插




```
s->data=x;          // ②
s->prior=p->prior;  // ③
s->next=p;          // ④
p->prior->next=s;   // ⑤
p->prior=s;         // ⑥
```

例2：删*p




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

§2.3.4 静态链表

上述链表是由指针实现的，其中结点分配和回收是由系统实现的，系统提供了malloc和free动态执行，故称为**动态链表**。

- **动态**----程序执行时系统动态分配和回收
- **静态链表**----
  - 先分配大空间作为备用结点空间（即存储池）
  - 用下标（亦称游标cursor）模拟指针，自定义分配和释放结点函数

§2.3.4 静态链表

1.定义

```
#define nil 0  //空指针
#define MaxSize 1000  //可容纳的结点数
typedef struct {
   DataType data;
   int next;
}node, NodePool[MaxSize];
typedef int StaticList;
```

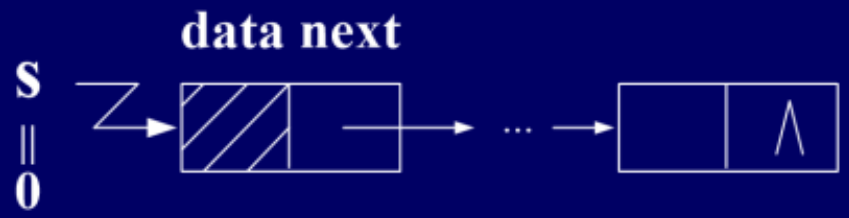
§2.3.4 静态链表

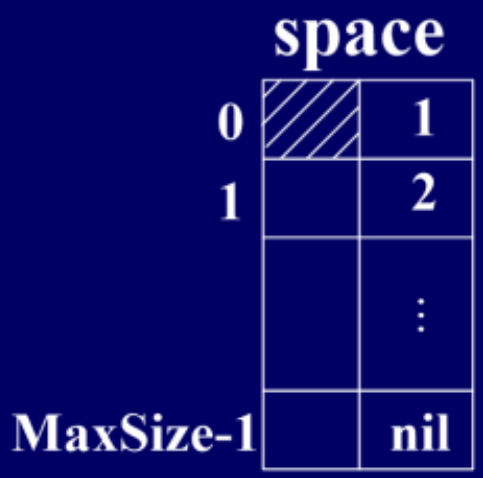
2.基本操作（模拟动态链表）

① 初始化

将整个表空间链成一个备用链表，只做一次。

```
void Initiate(NodePool space) {
    //备用链表的头结点位置为space[0]
    for (i=0; i<MaxSize-1; i++)  space[i].next = i+1;
    space[MaxSize-1].next = nil;
}
```





§2.3.4 静态链表

② 分配结点

在备用链表上申请一个结点，返回非空（不等于0）成功，否则失败。

```
int AllocNode(NodePool space) {
    i = space[0].next;
    //取备用链表上开始结点，位置为i，即space的第i个分量
    if (i)    //开始结点非空
        space[0].next = space[i].next;
        //将开始结点从备用链表上摘下
    return i;    //为nil时失效
}
```

§2.3.4 静态链表

③ 释放结点

将释放结点收还到备用链表的头上。

```
void FreeNode(NodePool space, int ptr) {
    //将ptr所指结点回收到备用链表头结点之后
    space[ptr].next = space[0].next;
    space[0].next = ptr;
}
```

§2.3.4 静态链表

3.一般操作

① 插入

```
void Insert(NodePool space, StaticList L, DataType x, int i) {
    // 在带头结点的静态链表L的第i个结点a_i之前插入新结点x
    p = Locate(space, L, i-1);    //找L中a_i的前驱①
    if (p == nil)
        Error("Position error!");    //位置错
    S = AllocNode(space);    //申请结点②
    if (s == nil)
        Error("Overflow");    //申请失败，空间不足判定
    space[s].data = x;    // ③
    space[s].next = space[p].next;    // ④
    space[p].next = s;    // ⑤
}
```

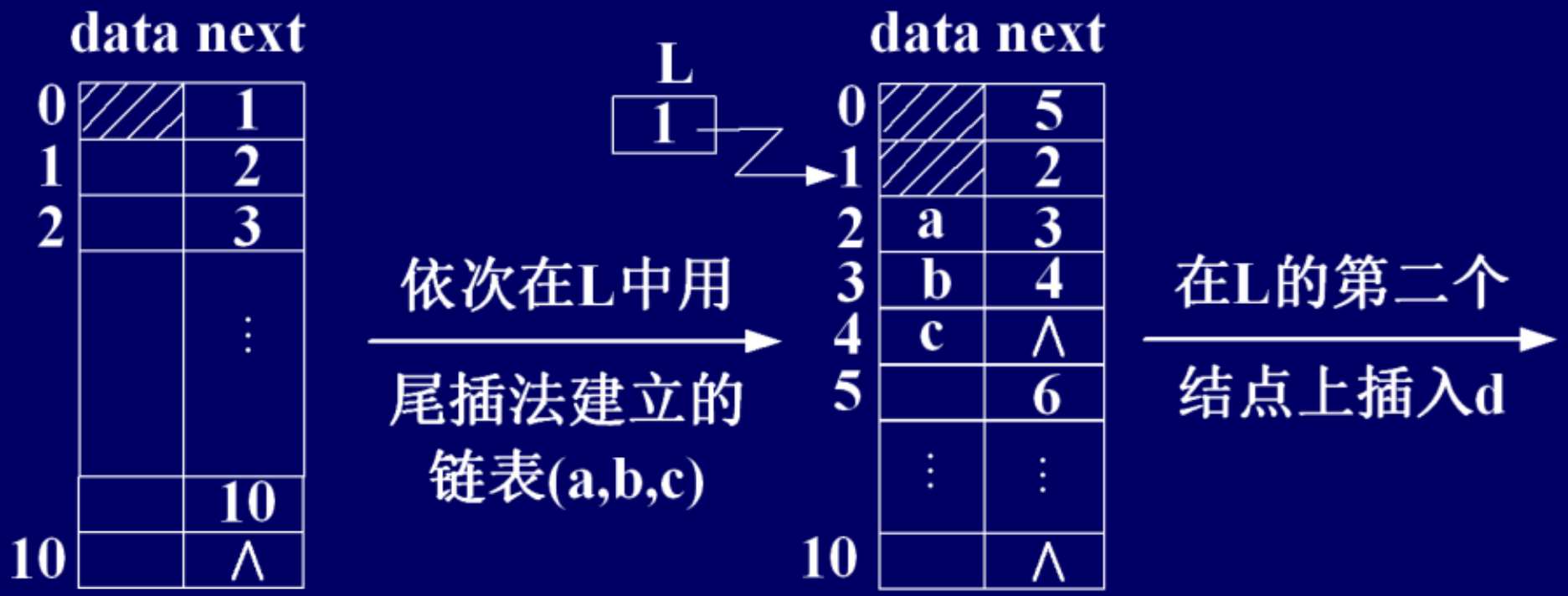
§2.3.4 静态链表

② 删除

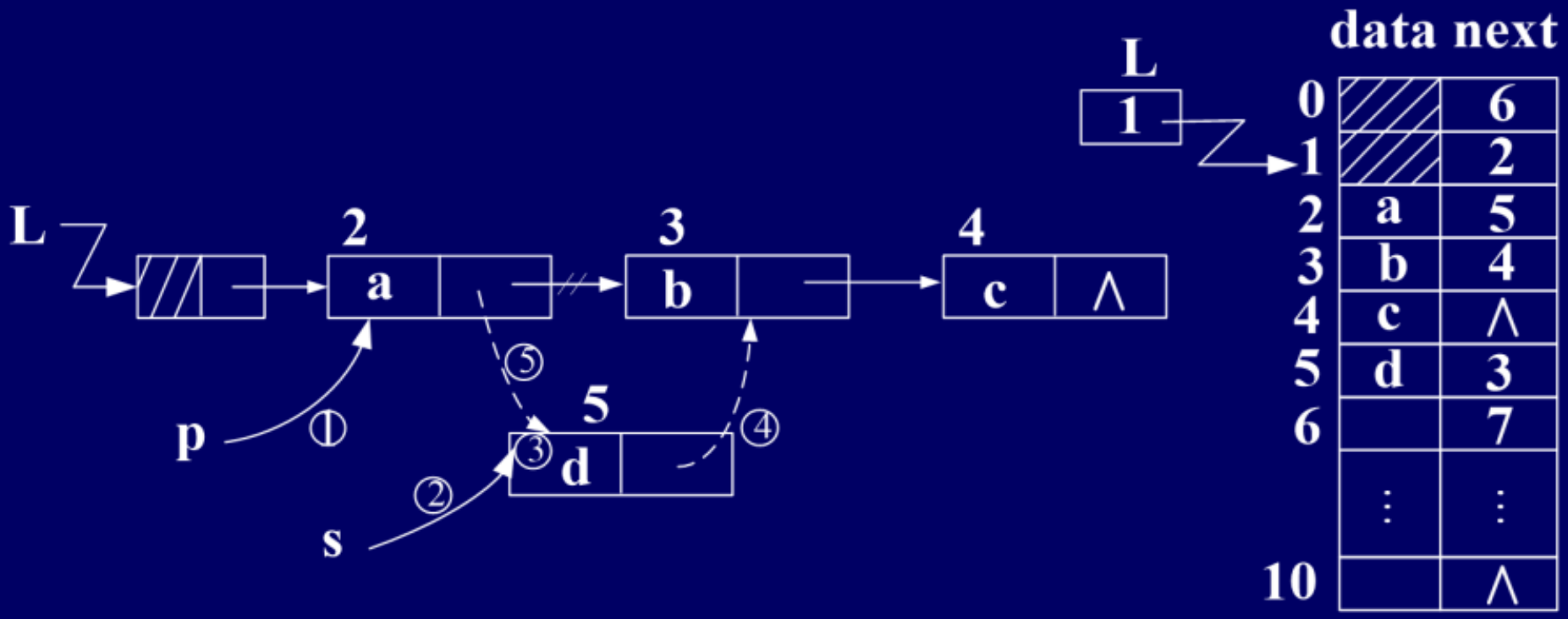
```
void Delete(NodePool space, StaticList L, int i) {
    p = Locate(space, L, i-1);    // 找L中a_i的前驱 ①
    if (p == nil || space[p].next == nil)
   Error("Position error!");    // 位置错
    q = space[p].next;    // q指向被删结点a_i ②
    space[p].next = space[q].next;    // 将a_i摘下 ③
    FreeNode(space, q);    // ④
}
```

注意：链表头指针实际上整数，即space的下标

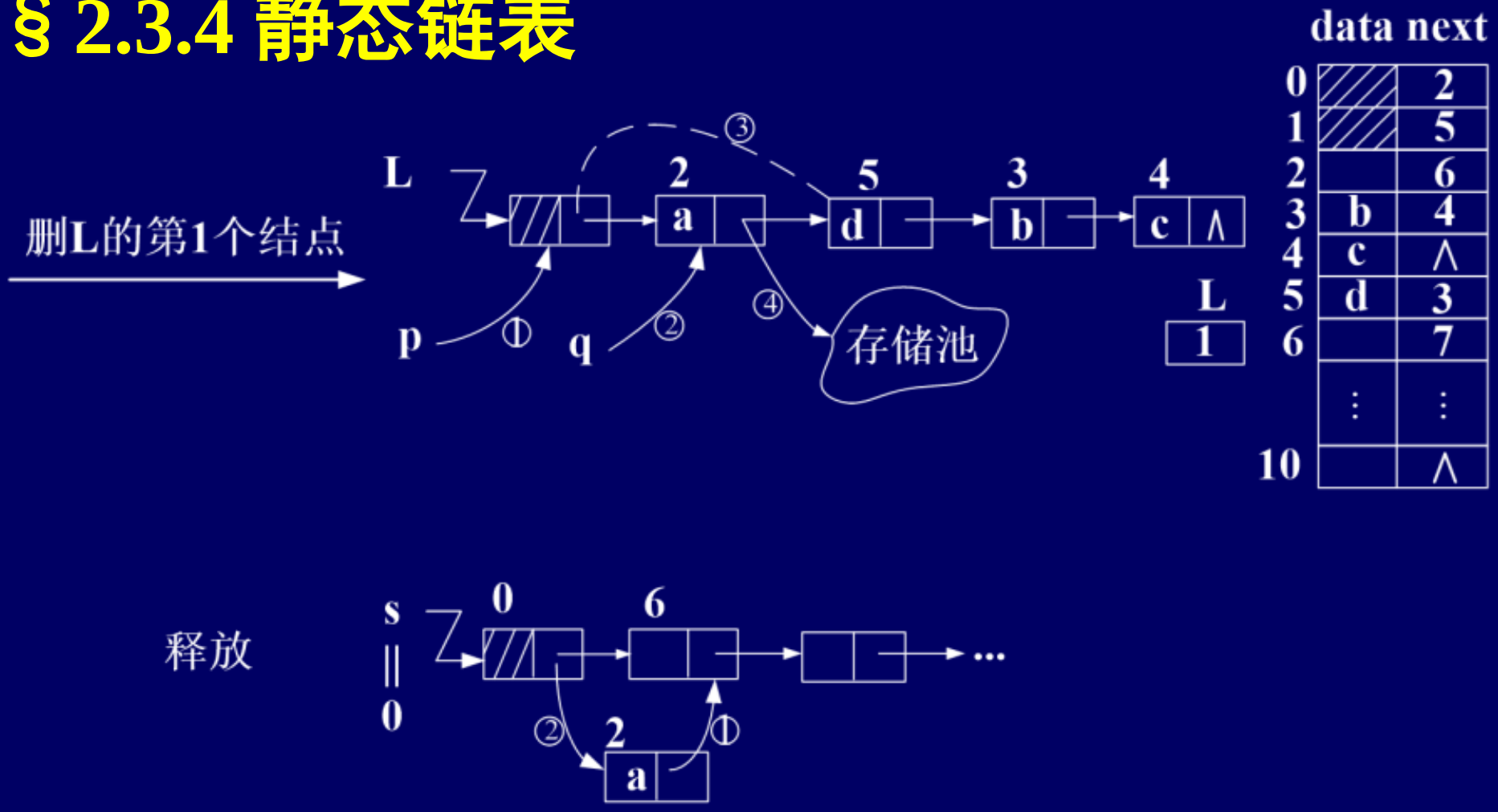
§2.3.4 静态链表


data next
0
1
2
10
1
2
3
10
∧
L
1
依次在L中用
尾插法建立的
链表(a,b,c)
data next
0
1
2
3
4
5
10
5
2
a
3
b
4
c
∧
6
∧
在L的第二个
结点上插入d

§2.3.4 静态链表



§2.3.4 静态链表



在spce中可能有多个链表，若再建立一个链表head，头结点在哪个位置？

§2.3.4 静态链表

静态链表特点：

没有指针类型的语言可以使用。亦可有双链表。循环链表等。

对比：

动态链表	静态链表
指针变量ptr	下标ptr
结点变量*ptr	结点space[ptr]
结点后继位置ptr->next	结点后继位置space[ptr].next

§2.4 顺序表和链表之比较

存储密度=结点数据所占的存储量/结点结构所占的存储量

作业

1. 为什么在单循环链表中设置尾指针比设置头指针更好？
2. 如何判断单链表是否存在环?如何判断两个单链表是否相交，以及相交点?请给出算法设计的思想。
3. 写一个算法将单链表中值重复的结点删除，使所得的结果表中各结点值均不相同。

数 据 结 构

Ch.3 栈和队列

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

http://staff.ustc.edu.cn/~xiaomj

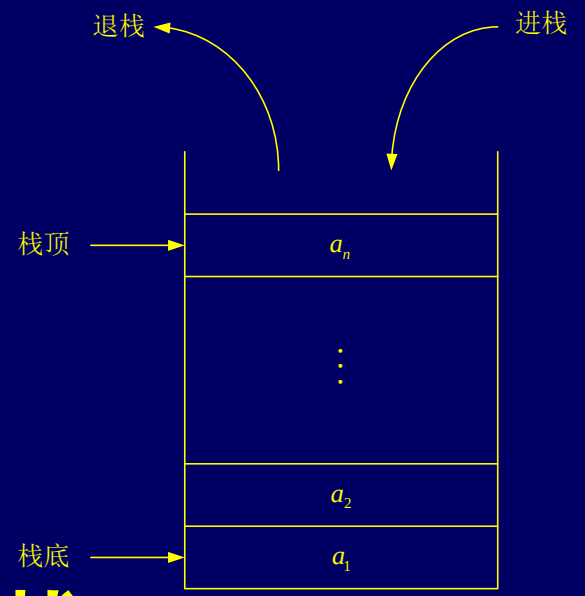
§3.1 栈

定义和运算

- 栈 —— 仅在表的一端插、删的(操作受限)线性表

  插入——进（入）栈、删除——出（退）栈

- 栈顶 —— 插删的一端
- 栈底 —— 另一端
- 结构特征 --“后进先出”
  - 修改原则：退栈者总是最近入栈者
  - 服务原则：后来者先服务 （LIFO表）

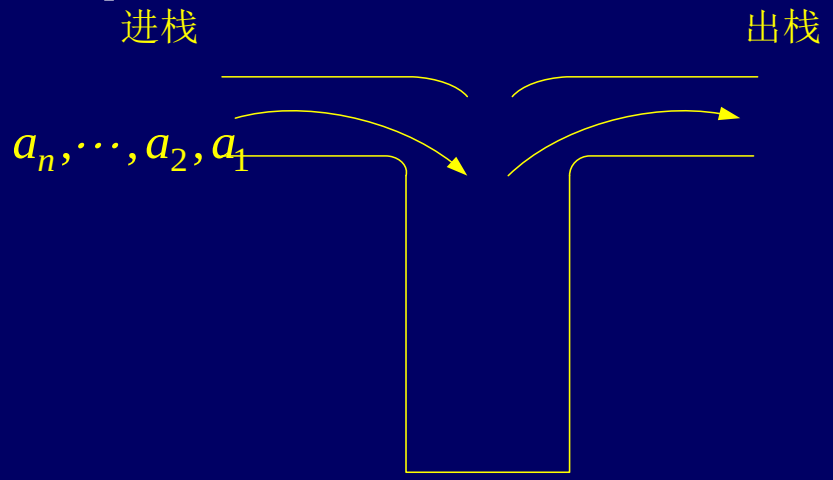


例：$a_1, a_2, \cdots, a_n$ 入栈

$a_n, a_{n-1}, \cdots, a_1$ 出栈

§3.1 栈

Note:后入栈者先出栈，但不排除后者未进栈，先入栈者先出栈。



基本运算：① 判栈空 ②入栈 ③出栈 ④判栈满 ⑤读栈顶 ⑥置空栈

§3.1.1 顺序栈

底相对固定

- 可设在向量的任一端
- Top指针为下标类型 (整型量)

```
#define StackSize 100
typedef struct {
        DataType  data[StackSize];
        int top;
}SeqStack;
```

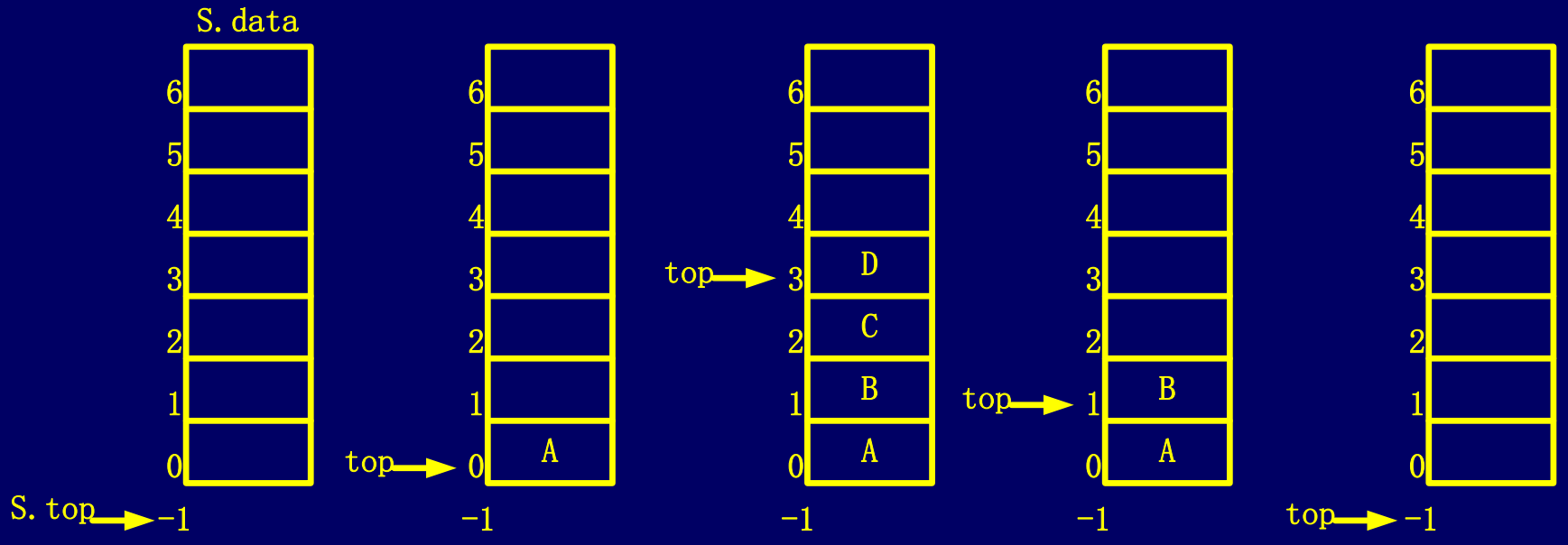
§3.1.1 顺序栈

- 设栈底在向量低端data[0]，则：
  - 入栈 ： top+1
  - 出栈：top-1
  - 栈空：top<0
  - 栈满：top=StackSize-1
  - 上溢：满时入栈
  - 下溢：空时出栈(不一定是错误)

§3.1.1 顺序栈

SeqStack S

设 S.top 指向当前栈顶元素



（a） 空栈　（b） A进栈　（c） B、C、D依次进栈　（d） D、C退栈　（e） B、A退栈

Note：top指针不是指物理地址，与C的指针变量含义不同。可理解为相对地址，是指示元素的所在位置

§ 3.1.1 顺序栈

实现：

初始化 （置空栈）

```
void InitStack(SeqStack &S) { //必须引用
        S.top=-1; }
```

判栈空

```
int StacEmpty(SeqStack &S) { //亦可用非引用
        return S.top<0;}
```

判栈满

```
int StackFull (SeqStack &S) {
        return S.top==StackSize-1;}
```

§ 3.1.1 顺序栈

- 入栈

```
void Push(SeqStack &S, DataType x) {
        if(StackFull(S))        Error("overflow");
        S.data[++S.top]=s; // 指针先加1，后x入栈
}
```

- 出栈

```
DataType  Pop(SeqStack &S) {
        if(StackEmpty(S))    Error("UnderFlow"); //下溢
        return S.data[S.top--];  //返回栈顶后指针减1
}
```

- 读栈顶

- 两个栈共享空间

§ 3.1.2 链栈

- 只在表头操作，故头指针即为top，无需头结点
- 定义

```
typedef  struct  snode {
        DataType  data;
        struct  snode *next;
} StackNode;
typedef  struct {
        StackNode   *top;
} LinkStack;
```

- 链栈动态分配结点，无需考虑上溢

§ 3.1.2 链栈

```
void  InitStack(LinkStack &S) {
        S.top=Null;
    }
int   StackEmpty (LinkStack &S) {
        return  S.top==NULL;
    }
void Push(LinkStack &S, DataType x) {
        StackNode  *p=(StackNode*) malloc (sizeof(StackNode));
        p->data=x;
        p->next=s.top;
        s.top=p;
    }
```

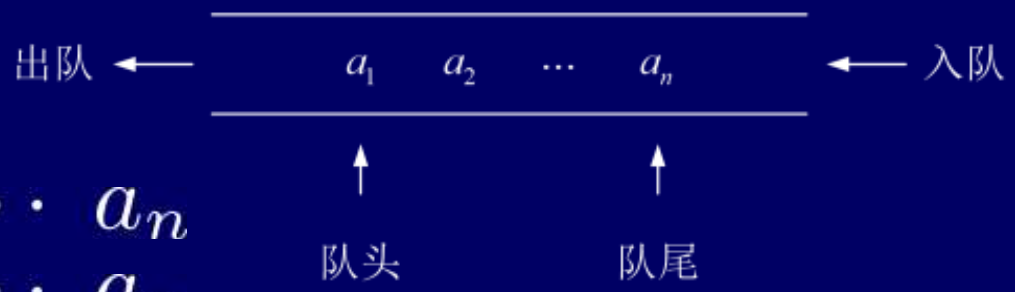
§ 3.1.2 链栈

```
DataType   Pop(LinkStack  &S) {
        DataType   x;
        StackNode  *p=S.top;
        if(StackEmpty(S))
                Error (“underflow”); // 下溢
        x=p->data;
        S.top=p->next;
        free(p);
        return  x;
   }
```

§3.2 队列

1. 定义

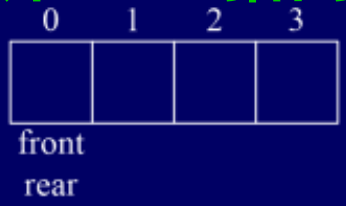
- 队列：运算受限的线性表，一端插入(队尾)，另一端删除(队头)。
- 结构特征：
  - 先进先出，先来先服务，FIFO表
- 例子：排队现象
- 操作：
  - 入队(插入)序列：$a_1 \quad \cdots \quad a_n$
  - 出队(删除)序列：$a_1 \quad \cdots \quad a_n$



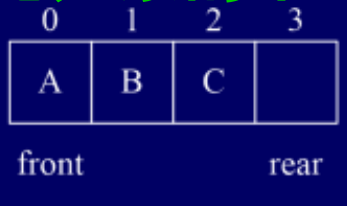
§ 3.2 队列

2. 顺序队列 —— 队列的顺序存储结构

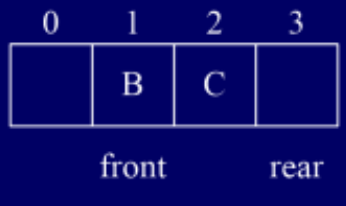
- 空队列：front = rear； 初值为0
- 入队：插入rear位置，然后加1
- 出队：删去front所指元素，然后加1
  - 头指针 front 指向实际队头元素
  - 尾指针 rear 指向实际队尾元素的下一个位置



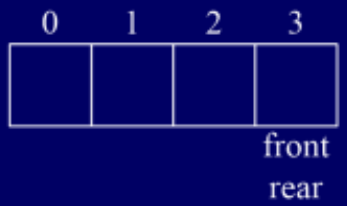
（a） 初始队列为空



（b） A、B、C入队



（c） A出队



（d） B、C出队，队为空

§ 3.2 队列

- 上溢：队满时入队
- 下溢：队空是出队
- 伪上溢：队非满但不能插入

  原因：f，r 只增不减

  例如：入，出，入，出，……

尽管任一时刻，队中最多只有1个元素，但无论事先定义多大的空间，均会产生指针越界

§ 3.2 队列

- 怎样消除伪上溢？
  - 重复利用已删除的结点空间，将向量视为首尾相接的环。这种用循环向量实现的队列称为循环队列。

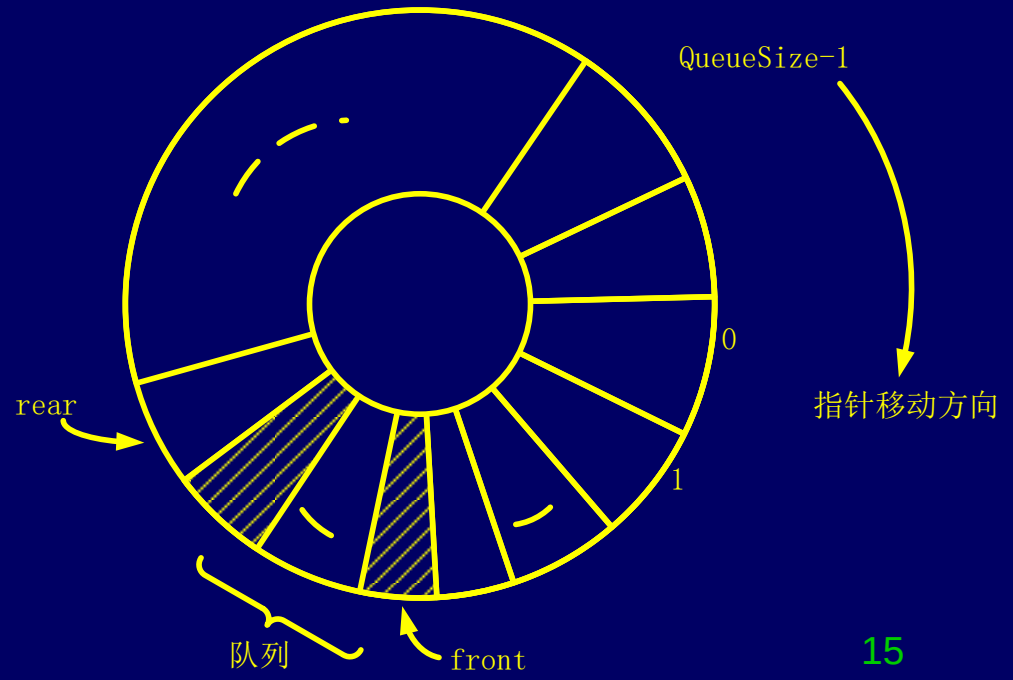
    f，r在循环定义下的加1动作：

    越界时，令其指向下界0

    i = (i+1==QueueSize) ? 0:i+1; // i为front或rear

    可用模运算简化：

    i=(i+1)%QueueSize;



- 循环队列：

  实用顺序队列多为循环队列

§ 3.2 队列

❖边界问题

队满和队空时，front=rear，如何区分？解决方法：

① 设一布尔量以示区分

② 留一个结点不用。约定

入队前，测试尾指针+1 (循环定义下) 是否等于头指针。若相等则为满

③ 使用1个计数器，记录队列长度 （用此法）

```
#define  QueueSize  100
typedef  struct {
        int front;
        int rear;
        int count;
        DataType  data [QueueSize];
} CirQueue;
```

§ 3.2 队列

❖操作实现

➢置空队

```
void InitQueue (CirQueue  &Q) {
        Q.front = Q.rear = 0;
        Q.count = 0; // 队空
}
```

➢判队空

```
int     QueueEmpty (CirQueue  &Q) {
        return Q.count == 0;
}
```

➢判队满

```
int     QueueFull (CirQueue  &Q) {
        return  Q.count ==QueueSize;
}
```

§ 3.2 队列

- 入队

```
void EnQueue (CirQueue   &Q, DataType  x) {
        if (QueueFull (Q) )
                Error(“overflow”);  //上溢
        Q.count++;
        Q.data [Q.rear] = x; // 插入
        Q.rear = (Q.rear+1)%QueueSize; // 尾指针加1
}
```

- 出队

```
DataType  DeQueue (CirQueue  &Q) {
        if(QueueEmpty (Q) )
                Error (“underflow”); //下溢
        temp = Q.data[Q.front];
        Q.count--;
        Q.front= (Q.front+1)%QueueSize;
        return  temp;
}
```

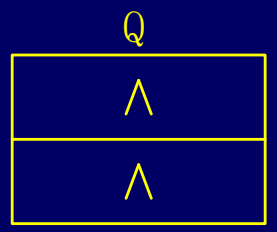
§ 3.2 队列

3. 链队列

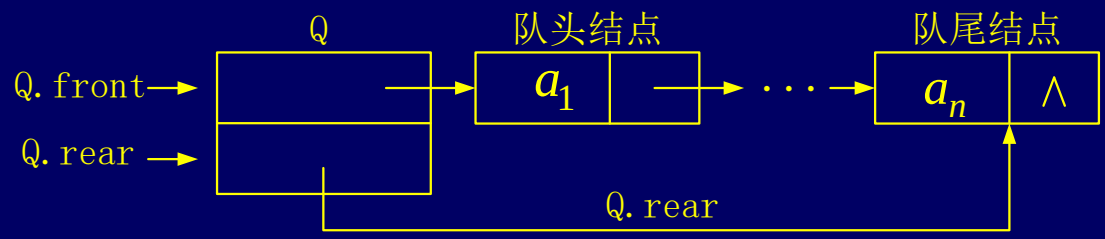
仅限于在表头和尾插删的单链表　　不带头节点：

```
typedef struct qnode{
        DataType  data;
        struct  qnode  *next;
}QNode;

typedef   struct  {
        QNode   *front;
        QNode  *rear;
} LinkQueue;
```



（a） 空队列



（b） 非空队列

§ 3.2 队列

```
void InitQueue (LinkQueue  &Q)  {
        Q.front = Q.rear = NULL; //有头结点是不同
}
int  QEmpty (LinkQueue  &Q) { //无上溢，故不判满队
        return  Q.front  == NULL; // 头尾指针均为空，有头结点时 f = r
}
void  EnQueue (LinkQueue  &Q, DataType  x) {
        QNode *p = (QNode*) malloc( sizeof(QNode) );
        p->data = x;  p->next = NULL;  // 新结点作为新的队尾
        if (Q.Empty(Q) )  // 若有头结点无需判此项
            Q.front = Q.rear = p; // 插入空队
        else { // 插入非空队尾，有无头结点均要做此项
            Q.rear->next = p;  // *p链到原尾结点之后。
            Q.rear = p; // 指向新尾*p
        }
}
```

§ 3.2 队列

```
DataType  DeQueue (LinkQueue  &Q) {
    if ( QEmpty(Q) )
        Error (“underflow”);  //下溢
    p = Q.front;  // 指向队头
    x = p->data;
    Q.front = p->next;  // 队头摘下
    if (Q.rear == p) // 原队中只有一个结点，删去后队变空
        Q.rear = NULL;
    free (p) ;
    return  x;
}
```

§ 3.3 栈和队列的应用

§ 3.3.1 栈的应用

1. 数据转换

- 问题： 一非负十进制整数N ⟹ 基为B的B进制数

例：

$$28_{10} = 3 \cdot 8^1 + 4 \cdot 8^0 = 34_8 \qquad 28_{10} = 34_8$$

$$72_{10} = 1 \cdot 4^3 + 0 \cdot 4^2 + 2 \cdot 4^1 + 0 \cdot 4^0 = 1024_4$$

规律：$N = \sum_{i=0}^{\lfloor log_B N \rfloor} b_i \cdot B^i \qquad 0 \le b_i \le B-1$ （3.1）

其中 b_i 表示B进制数的第 *i* 位数字

十进制数N可用长度为 $\lfloor log_B N \rfloor + 1$ 位B进制数表示为

$$b_{\lfloor log_B N \rfloor} \cdots b_2 b_1 b_0$$

§3.3.1 栈的应用

- 如何确定 b_i？

  令 $j = \lfloor log_B N \rfloor$ 则（3.1）式为：

$$N = b_j B^j + b_{j-1} B^{j-1} + \cdots + b_1 B + b_0$$
$$= (b_j B^{j-1} + b_{j-1} B^{j-2} + \cdots + b_2 B + b_1) B + b_0 \quad （3.2）$$

  (3.2)式整除B，则余数为 b_0 商为括号内的和式。故 (3.2)式可表达为：

$$N = (N/B) \cdot B + N\%B \quad // \text{“/”为整除}$$

- 算法思想：

  ① 模除求余数： $N\%B \Rightarrow b_0$

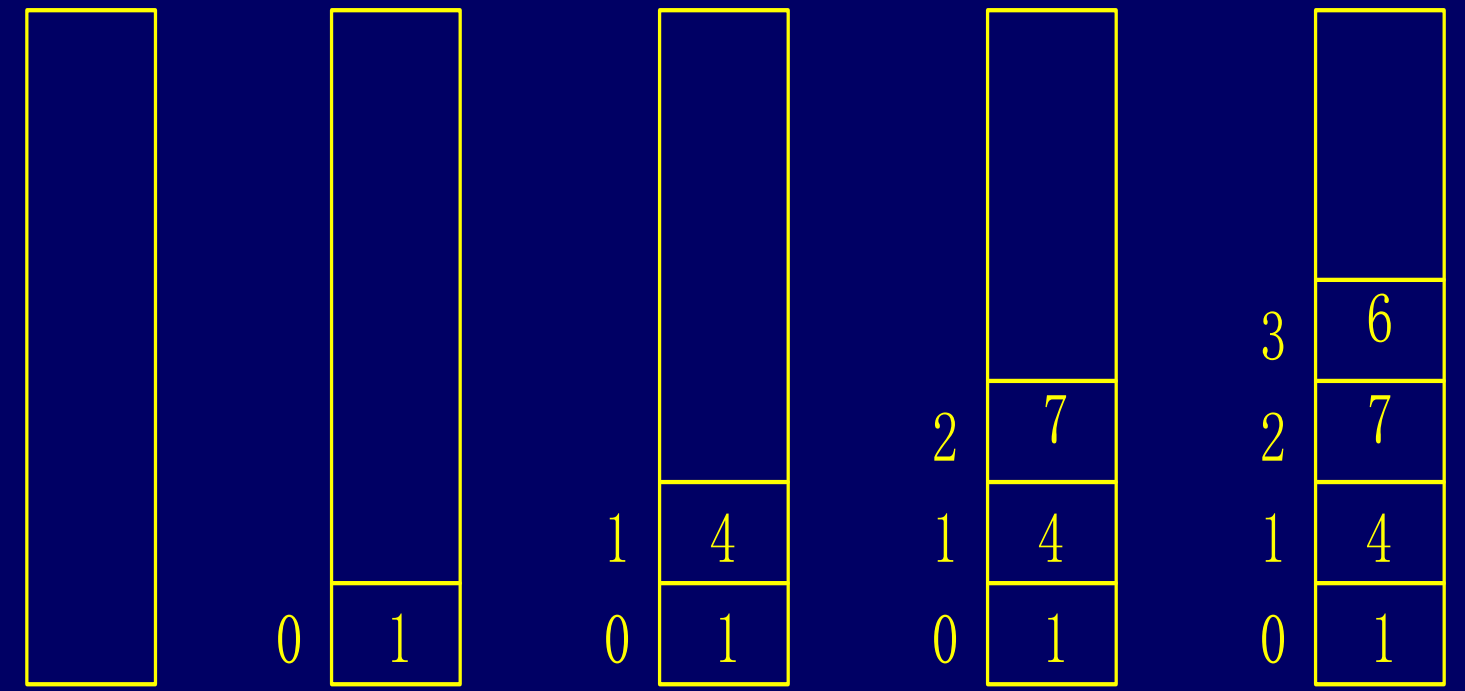
  ② 整除求商：N/B，令此为新的N，重复①求 b_1，反复至某N为0时结束

  上述过程依次求出的B进制各位为(从低位至高位): $b_0, b_1, \cdots, b_j$

  用栈保存，退栈输出 $b_j, b_{j-1}, \cdots, b_2, b_1, b_0$ 即为所求。

§ 3.3.1 栈的应用

例如： $N = 3553 \quad \Rightarrow \quad 6741_8$



空栈
N=3553

N%8=1
N/8=444
新N为444

444%8=4
444/8=55
N=55

55%8=7
55/8=6
N=6

6%8=6
6/8=0
N=0

§ 3.3.1 栈的应用

- 实现

```
typedef int DataType;
void  MultiBaseOutput  (int N, int B) {
        // N为非负十进制整数，  B为基
        int  i;   SeqStack   S;  //顺序栈S
        InitStack(S);  //置空栈
        while (N) { //从右向左产生b_i , i=0,1, …, j
                push(S, N%B);  // 将 b_i  入栈
                N=N/B;

         }

        while( !StackEmpty(S)) { // 栈S非空
                i = Pop(S);
                printf( “%d” ,i);
        }
 }
```

时间复杂度 $O(log_B N)$

§ 3.3.1 栈的应用

2.栈与递归

- 递归是一种强有力的数学工具，可使问题的描述和求解变得简洁与清晰
- 递归：若一函数、过程或数据结构定义的内部又直接或间接使用定义本身，则称它们是递归的，或递归定义的

§ 3.3.1 栈的应用

➢递归算法设计（分治法）分解、求解、组合

step1： 将原问题分解为一个或多个规模更小，但与原问题特性类似的子问题 (递归步骤) // 解子问题为调用语句

step2：确定一个或多个无须分解，可直接求解的最小子问题 (终结条件) // 归纳基础

例 1：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

//递归终结条件

//递归步骤 ⬅ (n-1)!规模比n!小1

```
int  F (int  n) { //设n为非负整数
     if (n==0)
        return 1;
     else
        return  n*F(n-1) ;
}
```

⬅ 至少有一个直接求解的最小子问题，保证递归终止，否则无限循环

⬅ 分解为一个子问题，若F(n)是解n!，则F(n-1)可解(n-1)!

§ 3.3.1 栈的应用

例2：

有些问题表面上不是递归定义的，但可通过分析，抽象出递归的定义。

写一个就地生成n个元素 $a_1, \cdots, a_{n-1}, a_n$ 全排列 (n!) 的算法，要求算法终止时保持 $a_1, \cdots, a_{n-1}, a_n$ 原状

解： 设 A[0 ..n-1] 基类型为char，“就地 (in place)”不允许使用 A 以外的数组

① 生成 $a_1, \cdots, a_{n-1}, a_n$ 全排列 $\xrightarrow{\text{分解}}$ n个子问题

求n-1个元素的全排列 + nth个元素

1st子问题	$a_1, \cdots, a_{n-2}, a_{n-1}$	a_n
2nd	$a_1, \cdots, a_{n-2}, a_n$	a_{n-1} //A[n-1] ↔ A[n]
3rd	$a_1, \cdots, a_n, a_{n-1}$	a_{n-2} //A[n-2] ↔ A[n]
⋮	⋮	⋮
*i*th	$a_1, \cdots, a_n, \cdots, a_{n-1}$ (↖ A[i])	a_i // A[i] ↔ A[n]
⋮	⋮	⋮
*n*th	$a_n, \cdots, a_{n-2}, a_{n-1}$	a_1 // A[1] ↔ A[n]

§ 3.3.1 栈的应用

② 递归终结分支

当 n=1 时， 一个元素全排列只有一种，即为本身。实际上无须进一步递归，可直接打印输出A

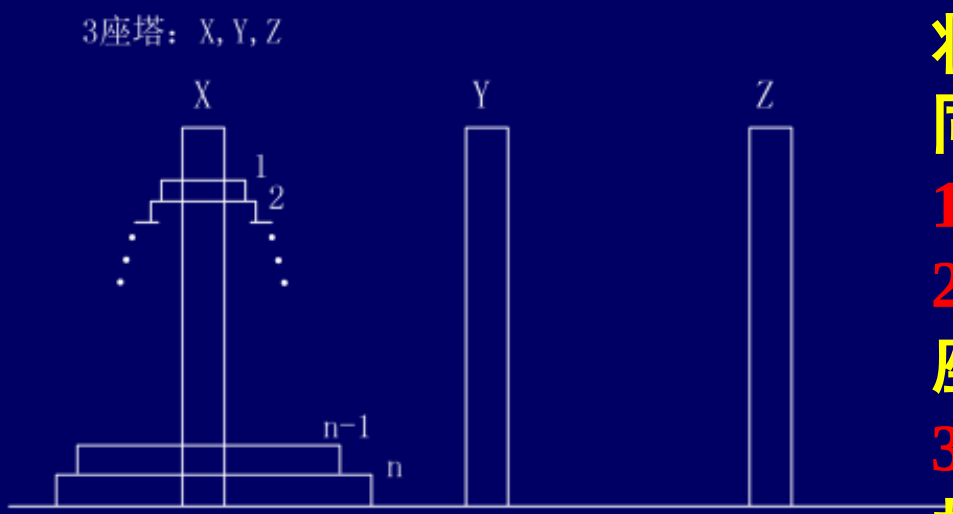
```
# define N  8 // A[0..7]
void  permute (char  A[], int  n) { //外部调用时令 n=7
      if(n==0)
         print (A);  // 打印A[0..7]
      else {
        Permute(A,n-1); //求A[0..n-1]的全部排列。1st子问题不用交换
       for (i=n-1; i>=0; i--) {
            Swap(A[i], A[n]); // 交换 a_n 和 a_i 内容，说明为引用
            Permute(A,n-1); // 求A[0..n-1] 全排列
            Swap(A[i],A[n]); //交换
          }
} }
```

§ 3.3.1 栈的应用

算法时间分析：

$$O(2^n) < n! < O(n^n)$$ 所以实验时，n不能太大

例3：n阶Hanoi塔问题



将X上的圆盘移到Z上，要求按同样次序排列，且满足：

1.每次只能移动一片

2.圆盘可插在X，Y，Z任一塔座上

3.任一时刻不能将大盘压在小盘上

§ 3.3.1 栈的应用

① 分解

设 n>1

原问题：将n片从X移到Z，Y为辅助塔，可分解为：

I. 将上面n-1 个盘从X移至Y，Z为辅助盘

II. 将 nth 片从X移至 Z

III. 将Y上n-1个盘子移至Z，X为辅助盘

//子问题特征属性与原问题相同
//规模小1，参数不同

② 终结条件

n = 1时，直接将编号为1的盘子从 X 移到Z

```
void   Hanoi (int n, char x, char y, char z ) {
    // n个盘子从 X 移至 Z，Y 为辅助
    if ( n==1 )  move(X,1,Z); // 将1号盘子从 X 移至 Z, 打印
    else {
     Hanoi (n-1,x,z,y);  //源X，辅Z，目Y
    move (x,n,z);
    Hanoi (n-1,y,x,z);  //源Y，辅X，目Z
    }
}
```

$$
\begin{aligned}
T(n) &= 2T(n-1) \\
&= 2^2 T(n-2) \\
&\vdots \\
&= 2^{n-1} T(1) \\
&= O(2^n)
\end{aligned}
$$

§ 3.3.1 栈的应用

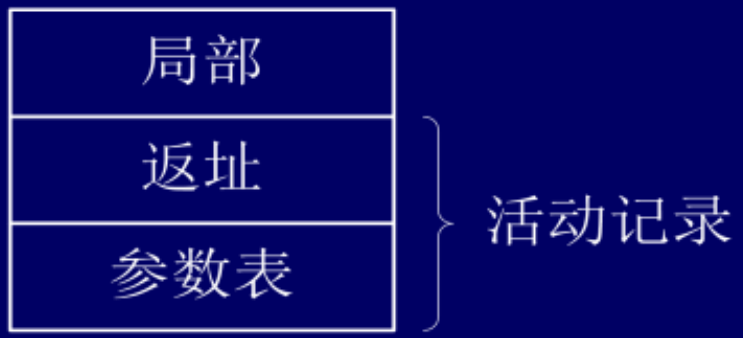
➢ 递归的内部实现

① 调用

调用一个函数时，系统将为调用者构造一个**活动记录**，并将其压入栈的顶，然后将程序控制权转移到被调用函数。若被调用函数有局部变量，则在栈顶也要为其分配空间，形成一个**活动结构**。实际上所有的递归或非递归函数都是这样实现的

活动结构： 局部变量

活动记录： 参数表的内容为实参
返址为函数调用语句的下一指令的位置



Activation Frame （活动结构）

§ 3.3.1 栈的应用

② 返回

当被调用函数执行完毕时，系统将活动结构退栈，并根据退栈返回地址将程序控制权转移给调用者继续执行

例： F(4)为例

```
void  main(void) {
        int   n;
        n = F(4);  //调用引起压栈
Ret  L1 ←
      }
```

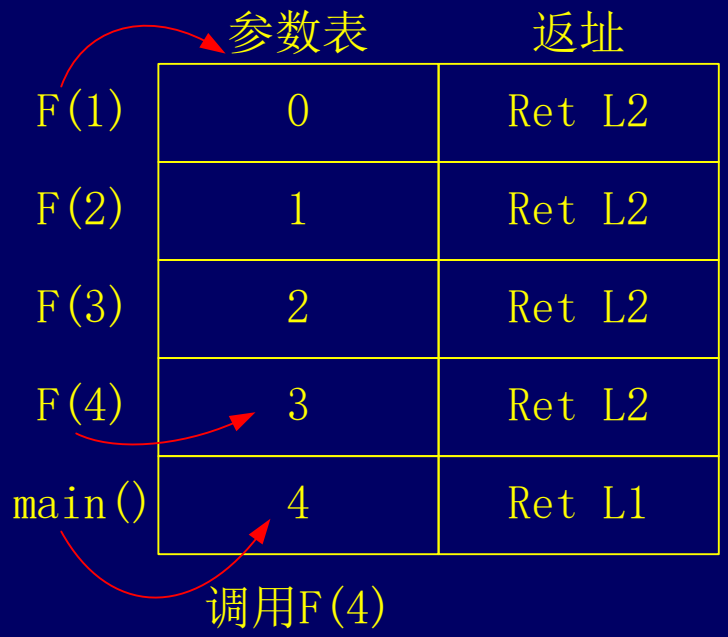
改写：

```
int  F (int n) {
     int  temp;
     if (n==0)
         return  1; //返回语句引起退栈
     else {
     //调用F(n-1) 引起入栈
       temp = n*F(n-1);
Ret  L2 ←
       return   temp;  // 退栈
     }
```

Ret L1: 赋值语句的地址
Ret L2: 计算乘法的地址

为简单起见，假设局部变量不入栈

§ 3.3.1 栈的应用



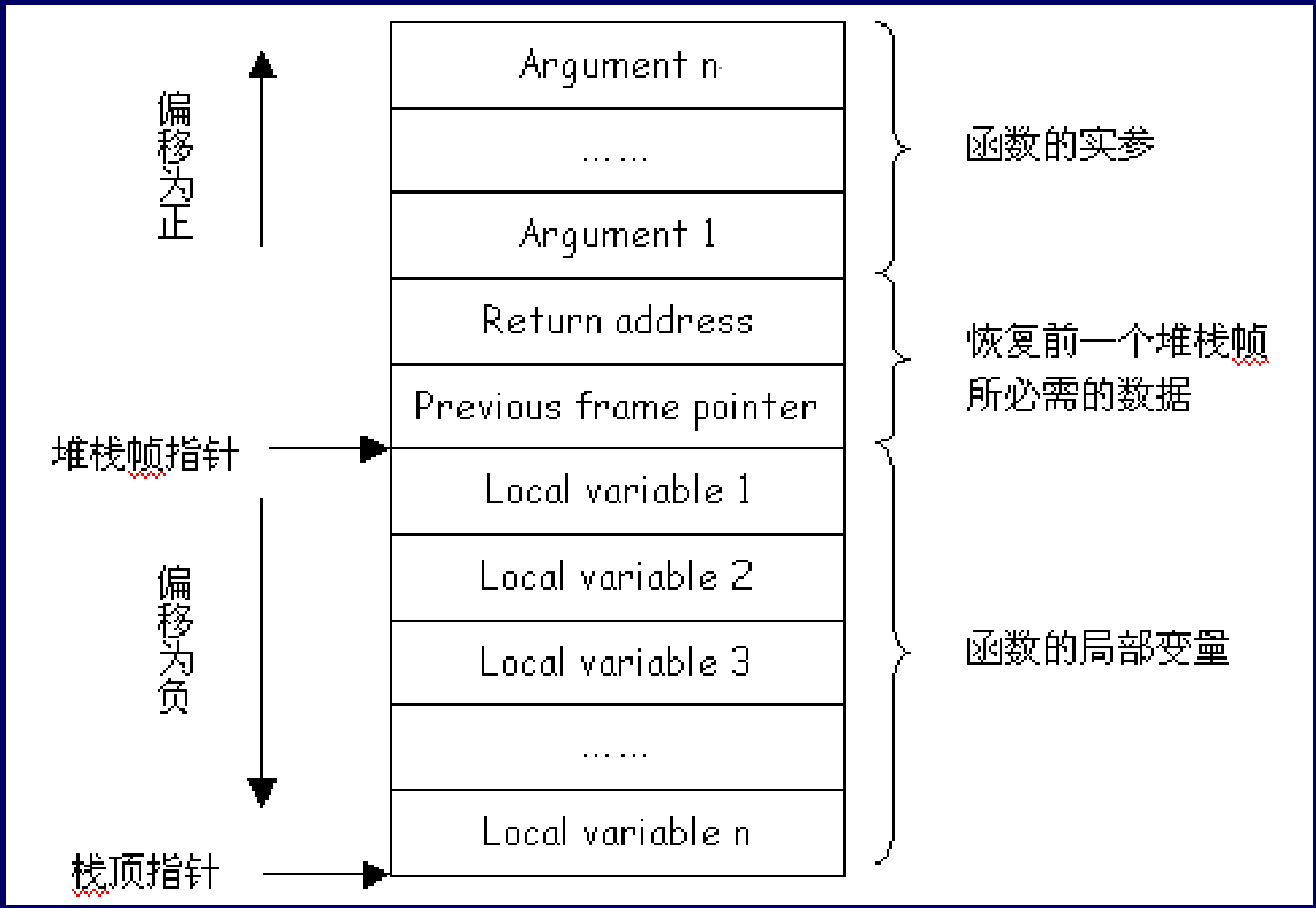
执行返回指令
Ret L2: temp=1*1; //从F(0）返回

＊ 0! = 1 是递归终结条件，故执行F(0)引起返回(return 1)

退栈 | 0 | Ret L2 | **，返回到F(1)的Ret L2处，继续执行temp = 1*1；**

按着执行return temp又引起 | 1 | Ret L2 | **退栈，返回到F(2)的Ret L2处，执行temp = 2*1，…**

典型的堆栈帧结构如图所示。堆栈中存放的是与每个函数对应的堆栈帧。当函数调用发生时，新的堆栈帧被压入堆栈；当函数返回时，相应的堆栈帧从堆栈中弹出。



典型的堆栈帧结构

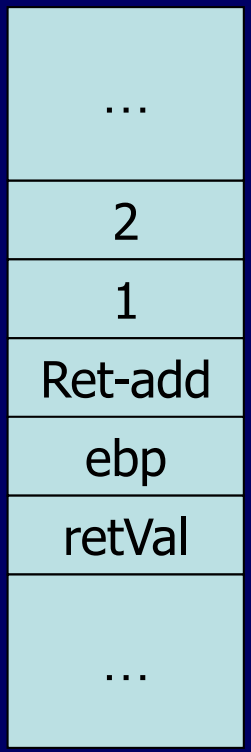
函数调用示例

- 函数：
  - int func(int a, int b){
  - int retVal = a + b;
  - return retVal;
  - }
  - int main(int argc, char* argv[])
  - {
  - int result = func(1, 2);
  - printf("Hello World!\n"
  - return 0;
  - }

```
00401000   push        ebp
00401001   mov         ebp,esp
00401003   push        ecx
00401004   mov         eax,dword ptr [ebp+8]
00401007   add         eax,dword ptr [ebp+0Ch]
0040100A   mov         dword ptr [ebp-4],eax
0040100D   mov         eax,dword ptr [ebp-4]
00401010   mov         esp,ebp
00401012   pop         ebp
00401013   ret
```

```
0040101C   push        2
0040101E   push        1
00401020   call        00401000
00401025   add         esp,8
```

Stack frame



- EIP(指令指针)、EBP(基址指针)、ESP指针(堆栈指针)
- Call DST: SP=SP-2, (SP+1,SP)=IP, IP=IP+D_{16}
- RET EXP: IP=(SP+1,SP), SP=SP+2, SP=SP+D_{16}

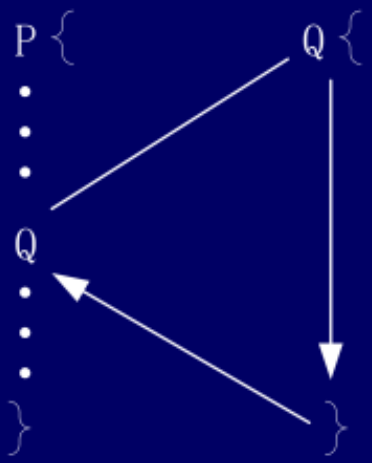
§ 3.3.1 栈的应用

递归算法的正确性

初学者很难相信递归程序的正确性

原因：一个函数尚未完成 (即对本函数的正确性还未确信) 时又调用了自身，故对递归函数正确性缺乏信心

例：非递归函数或过程



假设Q正确的情况下，证明了P正确，则一旦证明了被调过程Q是正确的，我们就对P的正确性深信不疑！

由于P、Q各自独立，独立于P来证明Q的正确性很容易，这大概是对自己写非递归程序较有信心的缘故

§ 3.3.1 栈的应用

若P是递归过程，则不可能独立于P来证明被调过程 (亦自身) 是否正确。因此，我们只能假设过程P内部所有递归的调用是正确的 (不考虑其内部实现细节)，才能由此证明P的正确性。其理论依据是数学归纳法：

(1) 证明参数满足递归终结条件时函数或过程正确 (相当于归纳基础)。

(2)假设过程函数内部的所有递归调用语句正确 (相当于归纳假设)，由此证明过程正确或函数是正确的 (相当于归纳步骤)

Note: 函数内的递归调用语句的参数应比函数本身的参数更接近递归终结条件参数，这样才能确保递归是可终止的

§ 3.3.1 栈的应用

3.表达式求值

算符优先法:

先乘除，后加减；从左到右计算；先括号内后括号外

4+2*3-10/5 = 4+6-10/5 = 10-10/5 =10-2 = 8

操作数(operand): 变量、常量，进OPND栈

操作符(operator): 算术、关系、逻辑运算符，进OPTR栈

界限符(delimiter): #，（，）

算符间的优先关系:

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Precede: 判定运算符栈的栈顶运算符θ_1与读入的运算符θ_2之间的优先关系的函数.

Operate: 进行二元运算a θ b的函数.

OperandType **EvaluateExpression()**

```
{InitStack(OPTR); Push(OPTR, '#');
  InitStack(OPND); c = getchar();
  While(c!='#' || GetTop(OPTR)!='#'){
    If(!In(c,OP)){ Push(OPND,c); c = getchar();}  //  不是运算符则进栈
    else
      switch(Precede(GetTop(OPTR),c)){
        case '<':   // 栈顶元素优先权低
              Push(OPTR,c); c = getchar();              break;
        case '=':   // 脱括号并接受下一个字符
              Pop(OPTR,x); c = getchar();               break;
        case '>':   // 退栈并将运算结果入栈
            Pop(OPTR,theta);   Pop(OPND,b);   Pop(OPND,a);
              Push(OPND,Operate(a,theta,b));            break;
        default: printf("Expression error!"); return(ERROR);
      } // switch
  } // while
  return GetTop(OPND);
} // EvaluateExpression
```

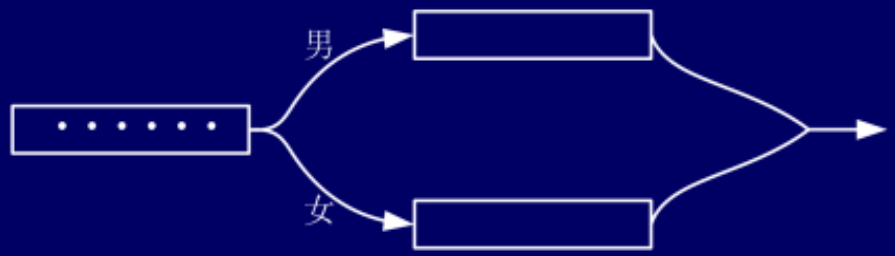
对算术表达式3*(7-2)求值

步骤	OPTR栈	OPND栈	输入字符	主要操作
1	#		3 * (7 - 2) #	Push(OPND,'3')
2	#	3	* (7 - 2) #	Push(OPTR,'*')
3	# *	3	(7 - 2) #	Push(OPTR,'(')
4	# * (	3	7 - 2) #	Push(OPND,'7')
5	# * (	3 7	- 2) #	Push(OPTR,'-')
6	# * (-	3 7	2) #	Push(OPND,'2')
7	# * (-	3 7 2	) #	Operate('7','-','2')
8	# * (	3 5	) #	POP(OPTR)
9	# *	3 5	#	Operate('3','*','5')
10	#	15	#	Return(GetTop(OPND))

§3.3.2 队列的应用

例： 周末舞会，男、女各排成一队，跳舞时，依次从男、女队的头上各出一人配成舞伴，若两队人数不同，较长的队中未配对者等下一轮舞曲

```
typedef  struct {
     char  name[20];
     char  sex; // M—男，F—女
} Person;
typedef  person  DataType; //将队列定义中的数据类型改为Person
void DancePartners（Person dancer[ ], int num）{
   int  i;  Person  P;
   CirQueue  Mdancers,  Fdancers;
   InitQueue(Mdancers); // 男士队列
   InitQueue(Fdancers);
```



§3.3.2 队列的应用

```
  for( i=0; i<num; i++ ) {
        P = dancer[ i ];
        if (P.sex == 'M')
            EnQueue (Mdancers, P); // 入男队
        else
            EnQueue (Fdancers, P); // 入女队
 }
 printf ("The dancing partners are:\n\n");
 while  (!QueueEmpty(Fdancers) && !QueueEmpty(Mdancers)) {
      // 男女队列均非空
        P=DeQueue(Fdancers);  // 女士出队
        printf("%s     ", P.name);  // 女士姓名
        P=DeQueue(Mdancers);  //男士出队
        printf("%s\n", P.name);
}
```

§3.3.2 队列的应用

```
    if (!QueueEmpty(Fdancers)) { //女队非空，输出剩余人数及队头者名字
        printf("\n There are %d women waiting for the next
                    round.\n", Fdancers.count);// count是队列属性，长度
        P=QueueFront(Fdancers); // 取队头
        printf ("%s will be the first to get a partner.\n", P.name);
    } else{
        // 男队类型处理
    }
}
```

时间复杂度：$O(n)$

作业

1. 已知从1至n的数字序列，按顺序入栈，每个数字入栈后即可出栈，也可在栈中停留，请设计算法验证给定的出栈的数字队列是否合法

2. 为了保障社会秩序，保护人民群众生命财产安全，警察同志需要与罪犯斗智斗勇，因而需要经常性地进行体力训练和智力训练！某批警察同志正在进行智力训练：

作业

1 2 3 4 5 6 7 8 9 = 110;

请看上边的算式，为了使等式成立，需要在数字间填入加号或者减号（可以不填，但不能填入其它符号）。之间没有填入符号的数字组合成一个数，例如：12+34+56+7-8+9 就是一种合格的填法；123+4+5+67-89 是另一个可能的答案。请你利用计算机的优势，帮助警察同志快速找到所有答案。每个答案占一行。形如：

12+34+56+7-8+9

123+4+5+67-89

数 据 结 构

Ch.4 串

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

http://staff.ustc.edu.cn/~xiaomj

Ch.4 串

字符串是一种特殊的线性表，它的每个结点仅有一个字符组成

早期，串只能出现在输入输出中，以直接量形式出现，不参与运算

§4.1 串定义及运算

基本概念

- 串：零个或多个字符组成的有限序列

  记为s=“$a_1a_2\cdots\cdots a_n$”（n≥0）

  s为串名，引号中为串值

  a_i：字母，数字等字符

- 串长度：n，n=0为空串

- 空白串：“ ”和“”之差别

§4.1 串定义及运算

- 子串：串中任意个连续字符组成的子序列称为该串的子串，包含子串的的串称为主串

  特别地：空串是任意串的子串

  任意串是其自身的子串

- 串常量：只能引用，不能改变，一般用直接量表示

  有的语言允许对其命名，如C++；

  const char path[]=“dir/bin/appl”

- 串变量：可读写，一般有名字

§4.1 串定义和运算

■基本运算

①求串长 ②复制 ③拼接 ④比较 ⑤子串定位

C中<string.h>

❖比较

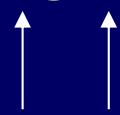
- ➢相等：长度相等，且各对应位置上字符亦相等
- ➢大小：字典序　　“axy”＜“ba”
  “baker”＞“Baker”

§4.1 串定义及运算

❖子串定位

子串在主串中首次出现时，该子串首字符对应主串中的位置序号

A=“This is a string” B=“is” 序号为3



3 6

❖其它复杂操作：均可用基本操作构成

C中有丰富的函数

§4.2 串的存储结构

串是特殊线性表，节点是单个字符，故有特殊技巧

1. 静态存储分配的顺序串

直接用定长字符向量来表示，上界预先给定

```
#define MaxStrSize 256   //用户定义
Typedef char SeqString[MaxStrSize];
SeqString S;  //S是可容纳255个字符的顺序串
```

- 终结符：‘\0’是串终结符，放在串值尾部
- 串长属性：若不设终结符

```
typedef struct {
        char ch[MaxStrSize];  //可容纳256字符
        int length;
}SeqString
```

§4.2 串的存储结构

2. 动态存储分配的顺序串

用C的malloc和free动态申请和释放向量空间，有两种形式：

① typedef char *string;　//C中串库相当于使用此类似定义

② typedef struct {

　　char *ch;　//若串非空，则按串实际长度分配，否则为NULL

　　int length;

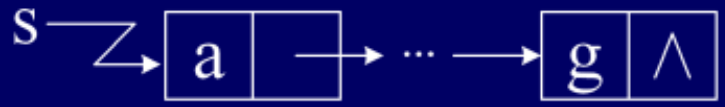
　}Hstring

§ 4.2 串的存储结构

3. 链串

块链存储

- 结点大小
  - 大小为1：



  - 大小为3：

- 存储密度d
  - d=数据大小/结点大小

§4.3 串的模式匹配算法（串定位运算）

1. 朴素的定位算法（串匹配）

- 主串：目标串（正文串Target，Text）
- 子串：模式（串）（子串，Pattern）

设T="$t_0t_1\cdots t_{n-1}$"

P="$p_0p_1\cdots p_{m-1}$" （$0\le m\le n$） 通常$m<<n$

- 应用：文本编辑，基因匹配等

§4.3 串的模式匹配算法

❖思想：

对合法的位置$0 \le i \le n-m$（合法位移），依次将目标串中的子串T[i..i+m-1]和模式串P[0..m-1]进行比较

若T[i .. i+m-1]=P[0 .. m-1](即：“$t_i t_{i+1} \cdots t_{i+m-1}$”=“$p_0 p_1 \cdots p_{m-1}$”)

则称从位置i开始的匹配成功；

否则，存在某个$0 \le j \le m-1$，使‘t_{i+j}’≠‘p_j’，则称从位置i开始的匹配失败

§4.3 串的模式匹配算法

- **有效位移：** 若T[i..i+m-1]=P[0..m-1]，则i称为有效位移
- **无效位移：** 若T[i..i+m-1]≠P[0..m-1]，则i称为无效位移
- **总结：** 朴素的串匹配算法是对合法位移检查其是否为有效位移

  有时需在T中找到所有有效位移

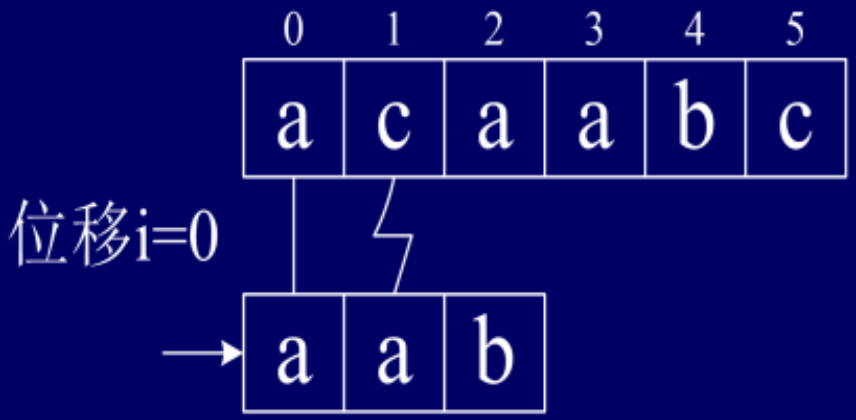
  通常找首次出现的有效位移

§4.3 串的模式匹配算法

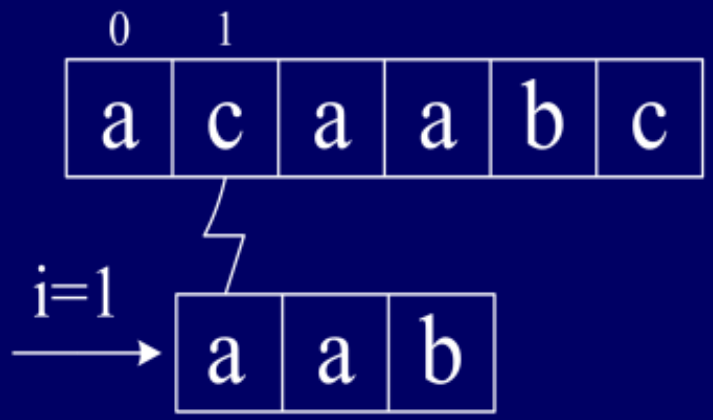
```
int NaiveStrMatch ( SeqString T, SeqString P ) { //顺序串上实现
    int i, j, k;
   int m = P.length, n = T.length;
   for (i = 0; i<=n-m; i++) { //i为合法位移，检查其是否为有效位移
      j = 0; k=i; //j指向模式，k指向目标
      while(j<m && T.ch[k] == P.ch[j]){ //检查i是否为有效位置
           k++; j++; //依次检查相应位置是否匹配
      }
      if (j == m) //即T[i..i+m-1]=P[0..m-1]，i为有效位移
           return i; //返回首次出现的有效位移i，匹配成功
   } //end for
   return -1; //匹配失败，所有合法位移均不是有效位移
}
```

§4.3 串的模式匹配算法

该算法是将模式串看作是在目标串上向右滑动的模板



（a）失败 右移



（b）失败 右移

§4.3 串的模式匹配算法

- **时间：**

  最坏情况：对所有合法位移，均要比较到模式的最后一个字符，才能确定位移无效

  即：$O((n-m+1)m) \approx O(n*m)$ // n>>m

  最坏情况发生在：

  目标串是 $a^{n-1}b$

  模式串是 $a^{m-1}b$ // 最后一次成功

- **用链串表示时定位运算类似**

§4.3 串的模式匹配算法

2. KMP算法（不带回溯）

下标仍从1算

- 原因分析

原因：没有利用部分匹配的结果

若能将模式串右移一段距离，则速度更快

§4.3 模式匹配算法

失败时有： $p_1=t_1$，$p_2=t_2$，$p_3\neq t_3$

①若将P右移1位，则p_1？t_2，因为

$p_1\neq p_2$，$p_2=t_2 \longrightarrow p_1\neq t_2$，故右移1位后仍失败

②若将P右移2位，则p_1？t_3，因为

$p_1=p_3$，$p_3\neq t_3 \longrightarrow p_1\neq t_3$，故右移2位后仍失败

结论： 上图比较失败时应直接将P右移3位（即i=1，2，3均为无效位移），即直接比较p_1和t_4，比较点不回溯。

Note： 上述观察告诉我们，分析模式本身，就可知道潜在的有效位移

§4.3 模式匹配算法

若使比较点不回溯（i不回溯），则当

$t_i \neq p_j$（T[i-j+1..i-1]=P[1..j-1],

即P的前j-1个字符与相应T上字符相等：

$t_{i-j+1} \cdots t_{i-1} = p_1 \cdots p_{j-1}$）时，P中哪个字符应与$t_i$继续比较？

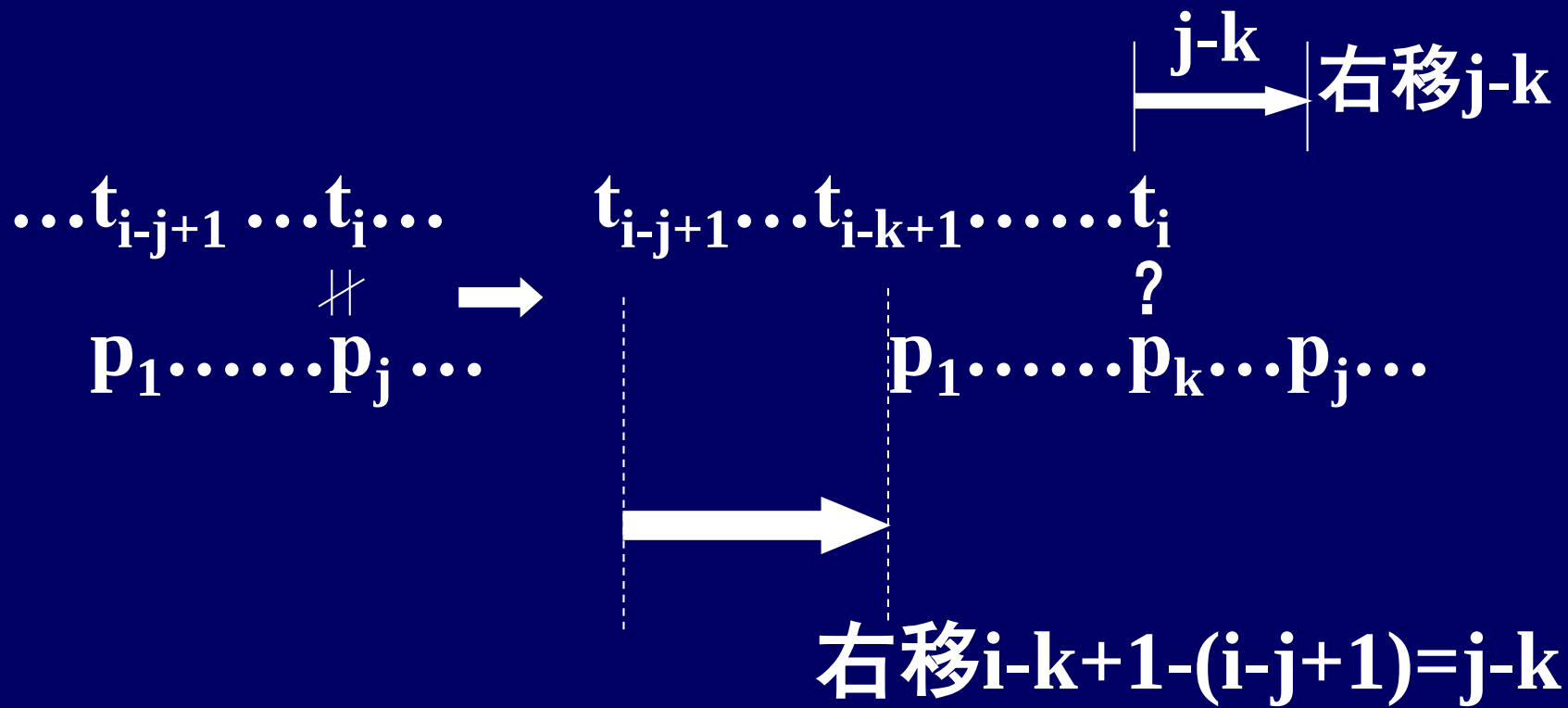
因为i不动时，必定是将模式右移一段距离，所以不妨设p_k（k<j）应与t_i 继续比较。

Knuth发现，k值仅依赖于P的前j个字符的构成，与目标T无关。

- 采用next[1..m]数组，用next[j]=k表示当$t_i \neq p_j$时，下一个应与t_i比较的是p_k
- 若P中任何字符均无需与t_i比较，则将p_1与t_{i+1}比较，此时令next[j]=0。P右移最大

§4.3 模式匹配算法

P的右移位数为： $j-next[j]$



- 算法　　若已知next[1…m],则匹配算法如下：

```
int KMPMatch ( char T[] ,char P[] ) { //T[0]和P[0]分别表示长度
  n=T[0] ; m=P[0] ; //长度
  i=j=1; //开始 t1~p1
  while (i<=n && j<=m)
      if (T[i]==P[j]) {
          i++;  j++; //继续比较下一位
      } else { //ti≠ pj
          k=next[j];
          if (k>0)
              j=k ;  //比较ti和pk: ti~pk
          else {
              j=1; i++;
          } //比较ti+1和p1:ti+1~p1
      } //endif，endwhile
```

§4.3 模式匹配算法

算法

```
    if (j>m) //匹配成功
        return i-m; //注意成功时，i和j均多加了1
    else
        return 0; //匹配失败
}
```

- 时间：循环主要取决于i只增不减，因为n>>m，所以时间为O(n)

§4.3 模式匹配算法

■ next数组的性质

① 整数数组：$0 \le next[j] < j$，即$0 \le k < j$

② 当$t_i \neq p_j$时，若$next[j]=k>0$，则比较t_i和p_k，此时有：

T[i-k+1..i-1]=P[1..k-1] //长度为k-1

T[i-j+1..i-1] =P[1..j-1] //失败前部分匹配，长度为j-1

⬇

P[j-k+1..j-1]=P[1..k-1] // “$p_1 \cdots p_{k-1}$”=“$p_{j-k+1} \cdots p_{j-1}$”

当$t_i \neq p_j$时，k的值应等于串P[1…j-1]中前后缀相等的子串的长度加1

T： …t_{i-j+1}… t_{i-k+1} … t_{i-1} t_i…

p_1……p_{j-k+1}…$p_{j-1}$$p_j$… // 前j-1个字符相等

P右移j-k位：

p_1……$p_{k-1}$$p_k$… // 前k-1个字符相等

③ 当满足性质2的k有多个（即前后缀相等的子串有多个）时，应取最大的k值。

原因：k最大，模式P右移j-k最少，不丢失任何匹配机会。

例：

j	1	2	3	4				
T	a	a	a	a	b	b	b	b
P	a	a	a	b				

P右移1位，匹配成功

P右移2位、3位均失败

在P[1..3]中，k=3最大，j-k=1位右移最少，k=2，k=1时失去匹配机会

即P[1..3]中，前后缀相等的最大真子串为P[1..2]=P[2..3]，长度+1=3=k

§4.3 模式匹配算法

真子串不包括自身，但包括空串

真子串：”aa”, “a”, “”

长度： 2 1 0

k： 3 2 1

§4.3 模式匹配算法

④若P[1…j-1]不存在首尾相同的字符串，或者说仅存在长度为零的相同前后缀（空串）子串，则k=1，即p_1与t_i继续比较

特别地，若j=1（即$t_i \neq p_1$），则P中任何字符无法与t_i继续比较，P右移1位，将p_1和t_{i+1}继续比较。按next定义，可取next[1]=0（对任何P成立）。

综上所述，当$t_i \neq p_j$时

$$next[j]=\begin{cases}0 & j=1\\ P[1\ldots j-1]\text{中前后缀相等的最大真子串的长度加1(包括空串),即：} & \\ Max\{k|1\le k<j \text{ 且” } p_1\cdots p_{k-1}\text{”=“}p_{j-k+1}\cdots p_{j-1}\text{”}\} \text{ //k=1时,为空串} & \end{cases}$$

例：

j	1	2	3	4	5	6	7	8
P	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

§4.3 模式匹配算法

next数组生成（递推法）

设next[j]=k已知，求next[j+1] (j≥1)

由性质4告知我们，对任何模式P，总有next[1]=0成立，给出了递推基础

∵next[1]~next[j]已知，且已知next[j]=k

∴P[1…j-1]中有：

“$p_1 \cdots p_{k-1}$”=“$p_{j-k+1} \cdots p_{j-1}$” //最大真子串长度为k-1

扩充一个字符p_j后，比较p_k~p_j

①若$p_j=p_k$，则P[1…k]=P[j-k+1…j]

即P[1…j]中前后缀的最大真子串长度为k，故

next[j+1]=k+1 或者 next[j+1]=next[j]+1

§4.3 模式匹配算法

②若$p_j \neq p_k$，则将求next值的问题看成是模式匹配问题，即P既为主串又为模式串

将P右移，用next[k]=h作下标，比较$p_h \sim p_j$

即：令k←next[k]，如此反复比较$p_j \sim p_k$

至$p_j=p_k$（情况①）或者

k=0，令next[j+1]=1为止 //没有一个字符与p_j比较

例子自己分析

目标串： $p_1 \cdots p_{j-k+1} \cdots p_{j-h+1} \cdots p_{j-1}\ p_j \cdots$

模式串： $p_1 \cdots\cdots p_{k-h+1} \cdots p_{k-1}\ p_k \cdots$

$p_1 \cdots\cdots p_{h-1}\ p_h$

§4.3 模式匹配算法

```
void GetNext (char p[] , int next[]) {
    //求模式串P的next数组（递推法）j-主串指针
    j=1; k=0; next[1]=0;
    m=P[0]; //模式串长度
    while (j<m)    //求next[j+1]
        if (k==0) next[++j]=++k; //next[j+1]=1
        else //k>0
            if (P[j]==P[k])
                next[++j]=++k; //next[j+1]=k+1
            else //pj≠pk
                k=next[k];
} //可改进为书上一样的算法
```

§4.3 模式匹配算法

时间：O(m)

KMP算法的时间加上求next数组后为O(n+m)

当n>>m时，它远远优于朴素匹配，尤其是模式串中存在很多“部分匹配”时

但当n≈m时，朴素匹配可能更好

- next数组的改进

next性质5：若$t_i \neq p_j$时，设next[j]=k>0，应比较$t_i \sim p_k$

若已知$p_k=p_j$，则必有$t_i \neq p_k$，此时应使用next[k]=k’

(k’>0)为下标继续比较：$t_i \sim p_{k'}$

§4.3 模式匹配算法

即可用下述方式节省时间：

当$p_j=p_k$时，将next[j]置为next[k]

此过程可重复！

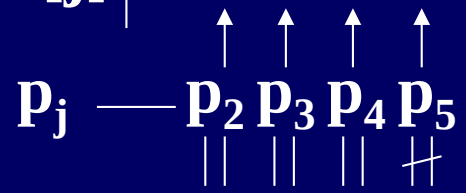
例：

j	1	2	3	4	5
P	a	a	a	a	b
next[j]	0	1	2	3	4
改进 nextval[j]	0	0	0	0	4

j	P	next[j]	nextval[j]
1	a	0	0
2	a	1	0
3	a	2	0
4	a	3	0
5	b	4	4

$t_i \neq p_2$,比较$t_i \sim p_{next[2]}$
$\because p_2=p_1$,next[2] ←next[1]

$t_i \neq p_3$,$t_i \sim p_{next[3]}$,
$\because p_3=p_2 \therefore$next[3]←next[2]



§4.3 模式匹配算法

求nextval算法

与书上不一样的算法，请比较

```
void GetNextVal (char P[], int nextval[]) { //求nextval
        j=1; k=0;
        nextval[1]=0; m=P[0];
        while (j<m) { //已知nextval[j]，求nextval[j+1]
                while (k>0 && P[j] != P[k])
                        k=nextval[k]; // 出循环时k=0或P[j]=P[k]
                j++; k++;
                if (P[j] ==P[k])
                        nextval[j] = nextval[k]; // 性质5
                else
                        nextval[j] = k; // nextval[j+1] = k+1
        }
}
```

时间仍为O(m)

作业

1. 已知模式串t = ‘abcaab*abc’，其中“*”为单一字符的通配符，请写出用KMP算法所得的next数组（数组下标从1开始）。

数 据 结 构

Ch.5数组和广义表

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

http://staff.ustc.edu.cn/~xiaomj

§5.1 多维数组

多维数组

是最易处理的非线性结构。因为各元素类型一致，各维上下界固定，所以它最容易线性化，故可看做是线性表的拓广。例如：二维数组可以看做是由列向量组成的线性表。

§5.1 多维数组

1. 结构特性

例：二维数组 $A_{mn}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$

$\forall a_{ij} \in A_{mn}$，它属于两个向量；$i$ th行和j th列。

除边界外，每个a_{ij}恰有两个直接前驱和两个直接后继。

§5.1 多维数组

- 非线性特征

  i th行：前驱a_{ij-1}，后继a_{ij+1}

  j th列：前驱a_{i-1j}，后继a_{i-1j}

  仅有一个开始结点：a_{11}

  仅有一个终端节点：a_{mn}

  其他边界上的结点只有一个直接前驱或一个直接后继，类似的m维数组$A_{n_1 n_2 \cdots n_m}$的每一个元素都属于m个向量。

§5.1 多维数组

2. 存储

一般均采用顺序方式存储，原因是结构中的结点不变动，内存是一维的，必须将多维数组线性化。

① 行优先（行主序）方式：

将$(i+1)$th行向量紧接在i th行向量之后：

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

特点：列下标变化快。Pascal、C等均是此方法。先排最右下标（多维）。

§5.1 多维数组

② 列优先（列主序）方法

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

特点行下标变化最快，先排最左下标（可推广至多维）。Fortan是用此方法。

③ 地址计算

$a_{ij}\rightarrow$一维存储地址（内部实现）。

- 基地址——开始结点存储地址Loc(a_{11}).
- 维数——每维的上下界（若下界固定，则只须知道维长度）
- 每个元素占用的单元数（元素大小）：L

§5.1 多维数组

例：行主序 $A_{m\times n}$ 。 A[1..m,1..n]

原理：a_{ij}的地址=基址+排在a_{ij}之前的元素个数×每个元素的大小

$$\mathrm{Loc}(a_{ij})=\mathrm{Loc}(a_{11})+[\underbrace{(i\text{-}1)\times n}_{\text{前}i-1\text{行结点总数}}+\underbrace{(j\text{-}1)}_{\text{第}i\text{行上}a_{ij}\text{之前的结点数}}]\times L$$

在C语言中，$A_{m\times n}$是$A[0..m\text{-}1\ ,\ 0..n\text{-}1]$，故有：

$$\mathrm{Loc}(a_{ij})=\mathrm{Loc}(a_{00})+[i\times n+j]\times L$$

§5.1 多维数组

- 多维推广：以C为例，行主序。

$$A_{d_1 d_2 \cdots d_n} \Rightarrow A[0..a_1-1, 0..d_2-1, \ldots, 0..d_n-1]$$

$$Loc(a_{j_1 j_2 \cdots j_n}) = Loc(a_{00..0}) + (j_1 \times d_2 \times d_3 .. \times d_n + j_2 \times d_3 \times .. \times d_n + .. + j_{n-1} \times d_n + j_n) \times L$$

- 思考：$A[c_1..d_1, c_2..d_2]$

$$Loc(a_{ij}) = Loc(a_{c1c2}) + [\underline{(i-c_1)} \times \underline{(d_2-c_2+1)} + \underline{(j-c_2)}] \times L$$

i th行前行数　　第2维长度　　i th行a_{ij}之前结点数

- 特点：随机存取。

§5.2 矩阵的压缩存储

- **用二维数组表示的特点：** 随机存取，存储密度为1。但对特殊和稀疏矩阵的存储则浪费空间，尤其是大规模科学与工程计算。

§5.2.1 特殊矩阵

- **有规律：** 压缩后可找到地址变换公式，保持随机存取功能。

{重复元素
零元素} 分布有规律

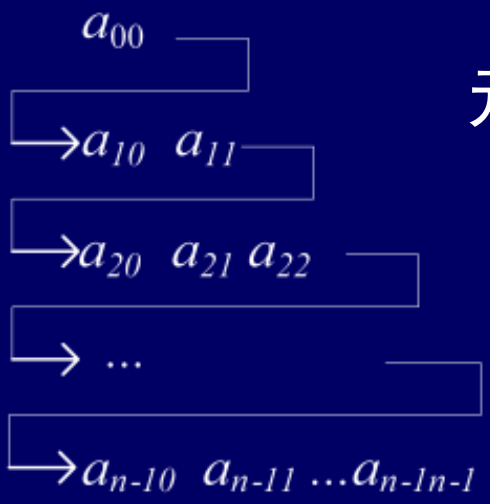
§5.2 矩阵的压缩存储

1. 对称阵

n阶方阵A，若$a_{ij} = a_{ji}\ \ 0 \le i, j \le n-1$，则称$A$为对称阵。

因为矩阵元素关于主对角线对称，故只存上三角或下三角元素即可，节约近一半空间。

不失一般性，存储下三角（包括主对角线），以行主序存储：



元素个数 $= \sum_{i=0}^{n-1}(i+1) = n(n+1)/2$

§5.2 矩阵的压缩存储

- **压缩存储：**

  将其存于向量sa[0..$n(n+1)/2-1$]中。

  如何访问a_{ij}？必须将其与sa[k]的对应关系找出来。

- **地址计算：**

  ① 下三角中有$j \le i$.

  a_{ij}之前有i行（0 ~ i-1）

  $$\text{元素个数} = \sum_{t=0}^{i-1}(t+1) = i(i+1)/2$$

  第i行上a_{ij}之前元素个数为j(0 ~ j-1).

  $$\therefore k = i(i+1)/2 + j$$

§ 5.2 矩阵的压缩存储

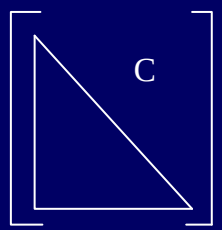
② 上三角中有$i < j$

$\because a_{ji} = a_{ij}$,只需交换上式的j和i即可得：

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

令I=max (i , j)， J=min (i , j)，则k和i，j之关系可统一表示为：$k = I(I+1)/2 + J$

2. 三角矩阵



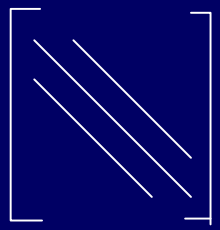
压缩方式同上，在sa中多增加一个单元：

sa[0..n(n+1)/2]

将C存于最后一个分量中。

§ 5.2 矩阵的压缩存储

3. 对角矩阵



总结：这类矩阵压缩存储后能找到地址计算公式，使其保持随机存取的功能。

§5.2 矩阵的压缩存储

§ 5.2.2 稀疏矩阵

- **定义：** 设A_{mn}中有t个非零元素，若$t << m \times n$，称A为稀疏矩阵。
- 稀疏因子：$\delta = \dfrac{t}{m \times n} \leq 0.05$ 一般非零元素分布无规律性
- 压缩存储：

只存储非零元，故须存储辅助信息，才能确定其位置。

三元组(i, j, a_{ij})：（行号，列号，非零元的值）唯一确定一个非零元。

当用三元组表示非零元时，有两种压缩方式：顺序和链式。

§5.2 矩阵的压缩存储

1. 三元组顺序表（三元组表）

以行主序（或列主序）的顺序存储非零元，跳过零元。得到一个其节点均是三元组的线性表，称此顺序存储结构为三元组表。

```
#define MaxSize 10000
typedef int DataType
typedef struct{//三元组
    int i, j;
    DataType v;
}TripleNode;
```

§5.2 矩阵的压缩存储

```
typedef struct{//三元组表
    TripleNode data[MaxSize];
    int m, n, t; //行数，列数，非零元总数
}TripleTable;
```

设a，b是TripleTable型变量。

$$A_{4\times5}=\begin{bmatrix}0&5&0&0&8\\1&0&3&0&0\\0&-2&0&0&0\\6&0&0&0&0\end{bmatrix}$$

$$B_{5\times4}=\begin{bmatrix}0&1&0&6\\5&0&-2&0\\0&3&0&0\\0&0&0&0\\8&0&0&0\end{bmatrix}$$

- **转置运算**

$$A_{m\times n}\Rightarrow B_{n\times m}$$

$$A[i][j]=B[j][i]\quad 0\le i\le m-1, 0\le j\le n-1$$

§ 5.2 矩阵的压缩存储

$$A_{4\times5}=\begin{bmatrix}0&5&0&0&8\\1&0&3&0&0\\0&-2&0&0&0\\6&0&0&0&0\end{bmatrix}$$

$$B_{5\times4}=\begin{bmatrix}0&1&0&6\\5&0&-2&0\\0&3&0&0\\0&0&0&0\\8&0&0&0\end{bmatrix}$$

a.data

	i	j	v
o	0	1	5
1	0	4	8
2	1	0	1
3	1	2	3
4	2	1	-2
a.t→5	3	0	6
	⋮		
MaxSize-1			

b.data

	i	j	v
o	0	1	1
1	0	3	6
2	1	0	5
3	1	2	-2
4	2	1	3
5	4	0	8
	⋮		
MaxSize-1			

§5.2 矩阵的压缩存储

① **方法一：**按B的次序或按A的列序转置。

∵A的列是B的行，故按A的列序转置，所得B是按行主序存放的。

✓ **基本思想：**对A中每列，从头至尾扫描a.data，找出所有列号为*col*的三元组($0 \le col \le n-1$)，将它们的行、列号互换后依次放入b.data，即可得行主序表示的B的三元组。

✓ **正确性：**∵按A的列号递增序转置，保证B按行号增序排列，B中同一行号的三元组，扫描A时所得三元组 $(i,col,v1),(j,col,v2)$必有$i<j$，转置后保证按B的列号增序排列。

✓ 例，上图。

§5.2 矩阵的压缩存储

```
void TransMatrix（TripleTable &a, TripleTable &b）{//A=>B
        int p, q, col;
        if (a.t == 0) Error(“A is empty”);
        b.m  = a.n; b.n = a.m; b.t = a.t ; //行列数互换
        q=0; //指示转置过的三元组
        for( col = 0; col < a.n ; col++)//对A的每一列号
            for( p = 0; p < a.t; p++)//扫描A的三元组表
                if (a.data[p].j == col) {//找A的列号为col的三元组
                    b.data[q].i = a.data[p].j ;
                    b.data[q].j = a.data[p].i ;
                    b.data[q].v = a.data[p].v ;
                    q++;
                }
}
```

时间O(n*t) 一般矩阵转置时间为O(mn).而t>m,故时间一般大于普通转置，当t~nm时为$O(n^2m)$

§ 5.2 矩阵的压缩存储

① 方法二：按A的行序转置。

若简单的变换a.data的行和列，则b.data为列主序存储，要重排。但若预先确定A中每一列（即B中每一行）的第一个非零元在b.data中应有的位置，则可正确转置。

位置向量：

$$\begin{cases} pot[0]=0 & //\ A\text{的第0列起址} \\ pot[col]=pot[col-1]+\text{第}col-1\text{列非零元个数} & 0\le col\le a.n-1 \end{cases}$$

§5.2 矩阵的压缩存储

- 思想

先求出A中每一列的非零元个数，可将第*col*列的非零元个数记入pot[*col*+1]中。

- ✓ step1：初始化将所有pot中元素清0. //O(*n*)
- ✓ step2：扫描a.data，将列号为*col*的非零元个数累加到pot[*col*+1]中。 //O(*t*)
- ✓ step3：令pot[col]=pot[col-1]+pot[col] $1 \le col \le$ a.n-1 //O(*n*)
- ✓ step4：扫描a.data，将(*i*,*col*,*v*)转置后放于b.data[pot[*col*]]中，pot[*col*]++. //O(*t*)

时间O(n+t)，快速。

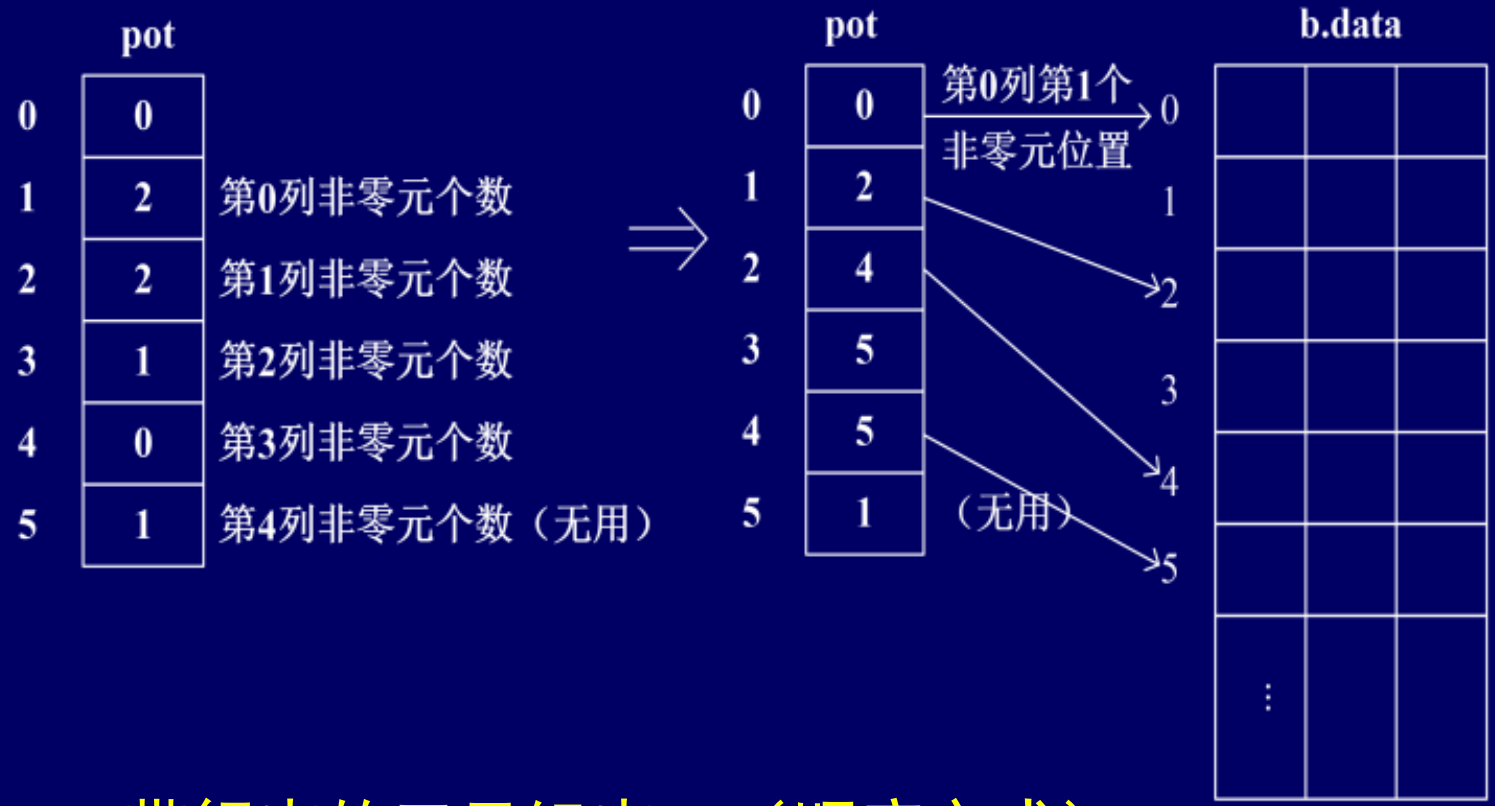
key：pot[1..a.n]=第0~a.n-1列的非零元个数。

§5.2 矩阵的压缩存储

```
void FastTransMatrix(TripleTable &a , Tripletable &b)  {//pot[0..a.n]，比n多1
    if (a.t == 0) Error("…");//A全零
    b.m = a.n; b.n  = a.n; b.t = a.t;
    for ( col = 0; col<=a.n ; col++) pot[col] = 0; //step1初始化
    for ( p = 0; p < a.t; p++) // step2扫描a.data
            pot[a.data[p].j + 1]++; //设a.data[p].j = col
    for ( col = 1; col < a.n; col++)//step3. pot[a.n]无用
            pot[col] = pot[col – 1] + pot[col];
    for ( p = 0; p < a.t; p++)  {//step4
            col = a.data[p].j; //当前三元组列号.
             q = pot[col]++;
             b.data[q].i = a.data[p].j;
             b.data[q].j = a.data[p].i;
             b.data[q].v = a.data[p].v;
    }
}
```

§5.2 矩阵的压缩存储

以上图为例，$A_{4\times5}$



2. 带行表的三元组表。（顺序方式）

在行优先存储的三元组表中，加入一个行表来记录稀疏矩阵压缩后每行非零元在三元组表中的起始位置。

§ 5.2 矩阵的压缩存储

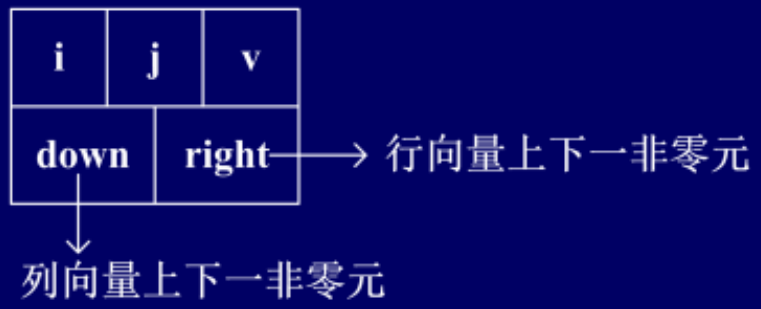
3. 十字链表

上两种方式是顺序存储，若非零元位置或个数经常发生变化，会引起结点移动，效率降低。此时宜用链式存储。

例：A←A+B

稀疏矩阵的链接存储方式有多种，这里仅介绍十字链表

- 结点结构



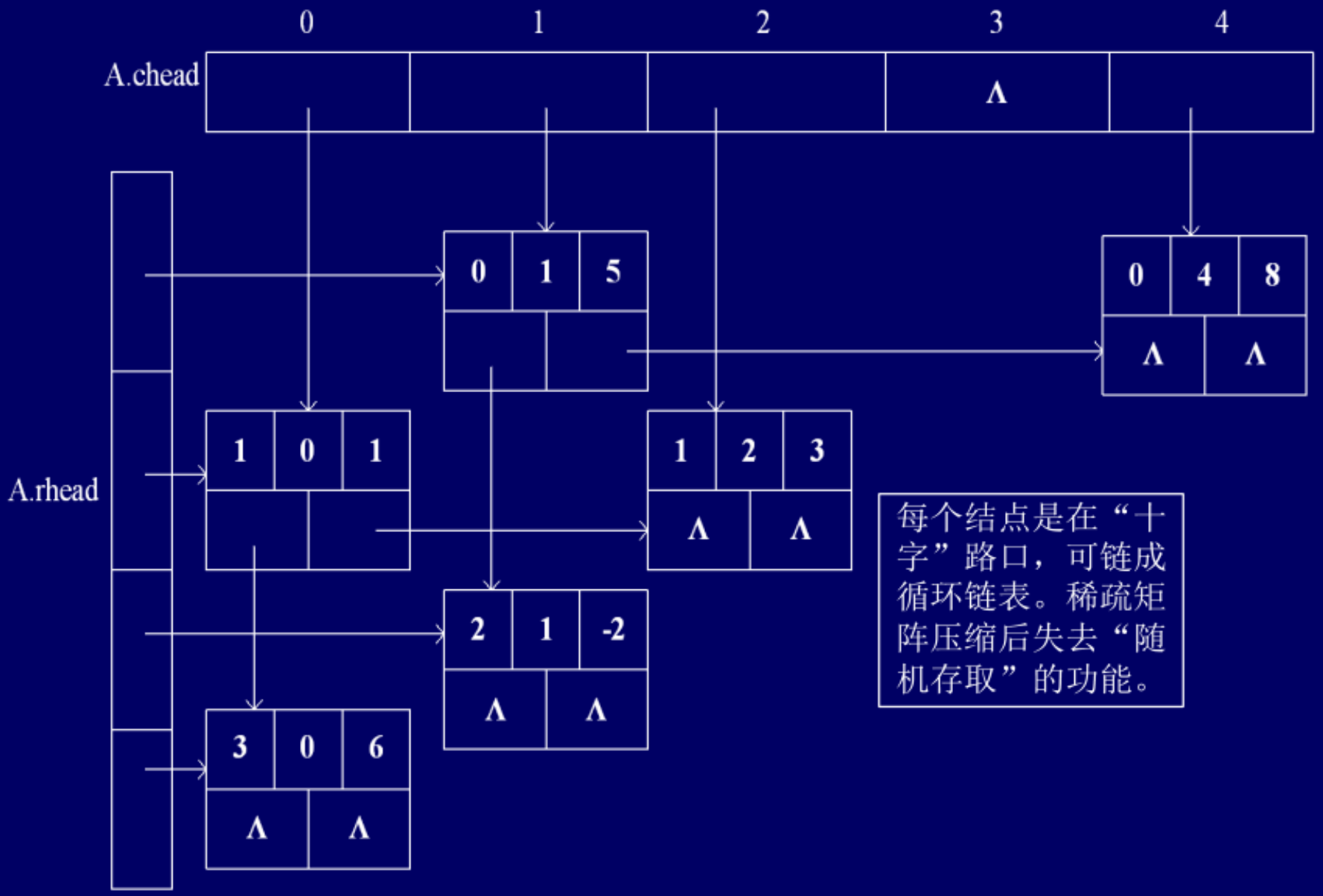
- 存储结构

分别设两个指针数组作为各行、列单链表的头指针。

§5.2 矩阵的压缩存储

```
typedef struct CLNode{
    int i, j ;
    DataType v;
    struct CLNode * right, *down;
}CLNode;
typedef struct {
    CLNode *rhead[MaxRow]; //行链表头指针，MaxRow在前定义
    CLNode *chead[MaxCol];  //列…
    int  m,n, t;
}CrossList;
CrossList A;
```

§5.2 矩阵的压缩存储


0
1
2
3
4
A.chead
Λ
0 1 5
0 4 8
Λ Λ
A.rhead
1 0 1
1 2 3
Λ Λ
2 1 -2
Λ Λ
3 0 6
Λ Λ
每个结点是在“十字”路口，可链成循环链表。稀疏矩阵压缩后失去“随机存取”的功能。

§5.3 广义表（Lists）

1. 概念

是线性表的推广，如将线性表中元素a_i放宽到可以是自身的结构。

- 定义：$LS=(a_1,a_2,\cdots,a_n),n\geq 0$，它由n个元素构成的有限序列，其中$a_i$或是原子，或是广义表（子表）。

  LS-名字，n-长度，n=0为空表。

  一般用小写字母表示原子，大写字母表示子表。

- 表头、表尾、深度

  若 $LS\neq\Phi$，则a_1成为表头（首），剩余元素构成的表 $(a_2,\cdots,a_n)$ 为表尾。广义表是递归定义的，展开到每一元素均为原子时括号的最大层数为深度。

§5.3 广义表（Lists）

- 例：

E=( ) ——空表，长度n = 0，深度d = 1.

L=(a, b) ——n = 2，d = 1.（线性表）

A=(x, L)=(x, (a, b)) ——n=2, d=2. a_1为原子，a_2为子表

B=(A, y)=((x, (a, b)), y) ——n = 2, d = 3.

C=(A, B)=((x, (a, b)), ((x, (a, b)), y)) ——n = 2, d = 4.

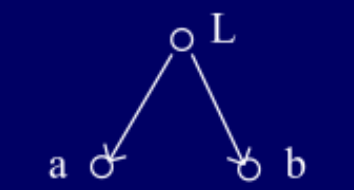
D=(a, D)=(a, (a, (a, (…)))) ——n = 2, d = ∞ .

- 若规定任何表都有名字，则可在每个表前冠名。

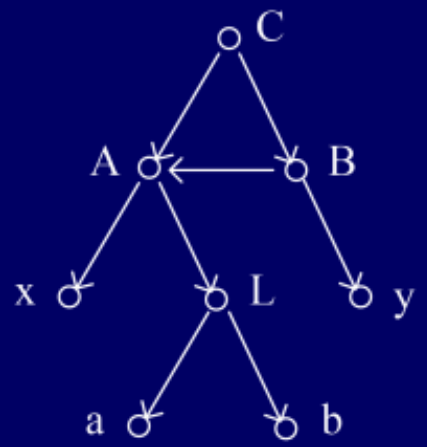
  E( )　L(a, b)　A(x, L(a, b))

§5.3 广义表（Lists）

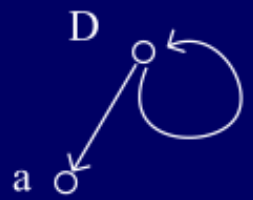
- 图示



纯表（与树对应）



再入表（允许结点共享）



递归表（允许递归）

- 各种表之关系

递归表 ⊃ 再入表 ⊃ 纯表 ⊃ 线性表

§5.3 广义表（Lists）

- 运算

求表头、表尾。

head(A) = x,　tail(A) = ((a, b))　//表尾是表，表头不一定

head(tail(A)) = (a, b) ——表

$$\begin{cases} \text{head}(\text{head}(\text{tail}(A))) = a \text{——原子} \\ \text{tail}(\text{head}(\text{tail}(A))) = (b) \text{——表} \end{cases}$$

$$\begin{cases} \text{head}(\text{tail}(\text{head}(\text{tail}(A)))) = b \text{——原子} \\ \text{tail}(\text{tail}(\text{head}(\text{tail}(A)))) = ( ) \text{——表} \end{cases}$$

Note：$( )$　和　$(( ))$ 不同。

$( )$　为空表n=0，不能求表头和表尾。

$(( ))$　为非空表，n=1．可求表头和尾：

$\text{head}[(( ))] = ( ),\quad \text{tail}[(( ))] = ( )$

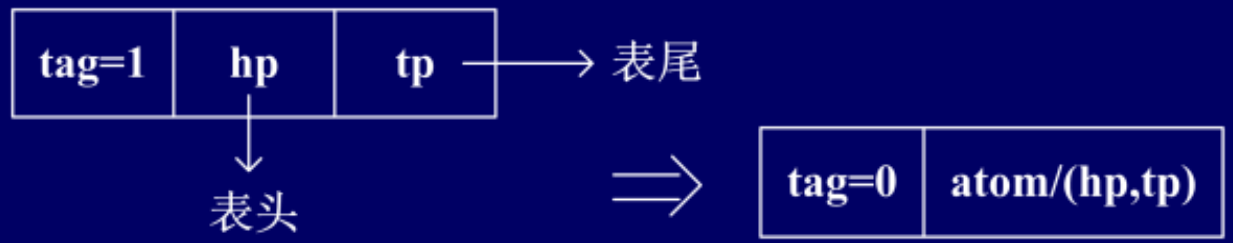
§5.3 广义表（Lists）

2. 存储结构

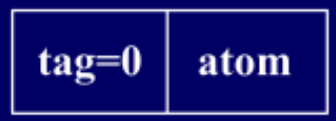
因为广义表数据元素可具有不同结构，故难以用顺序方式存储。一般用链接方式存储，称之为广义链表。

(1) 广义表的头尾链表表示方法。

- 表结点：



- 原子结点：



$$tag = \begin{cases} 1 & \text{表结点，使用hp和tp} \\ 0 & \text{原子结点，使用atom} \end{cases}$$

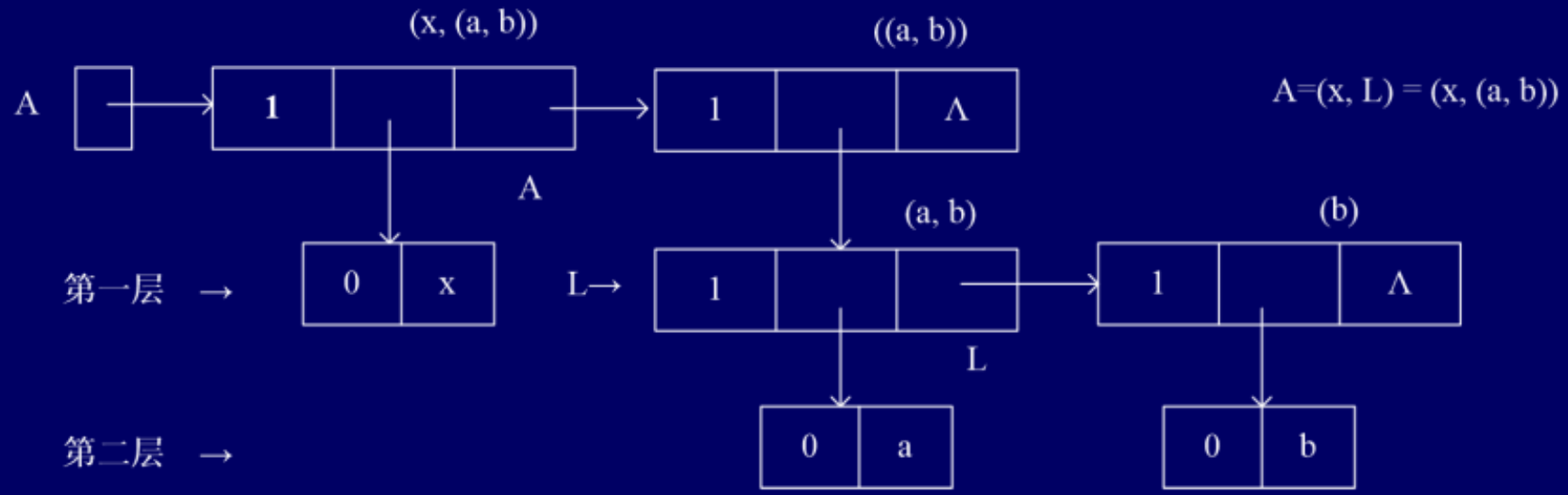
使用Union说明

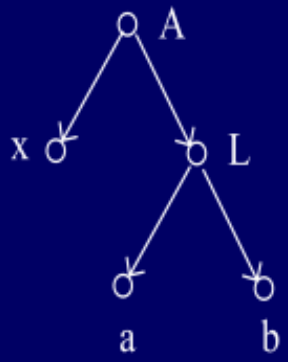
存储结构见书上说明

§5.3 广义表（Lists）

- 图示

  E=NIL





§5.3 广义表（Lists）

- 特点

① 除空表的表头指针为空外，头指针均指向表结点。

② 任一表结点的hp均指向表头部（原子结点或表结点）

任一表结点的tp均指向表尾部（当表尾部为空表时，tp=NIL，否则必指向表结点）

③ 易分清表中原子和子表所在层次。

如x、L在第一层，a、b在第二层。

④ 最高层的表结点数即为广义表的长度。

⑤ 浪费空间，易求表长和表深度。

§5.3 广义表（Lists）

(2) 单链表示法

模仿线性表的单链表结构，当所有元素为原子时，等价于单链表。

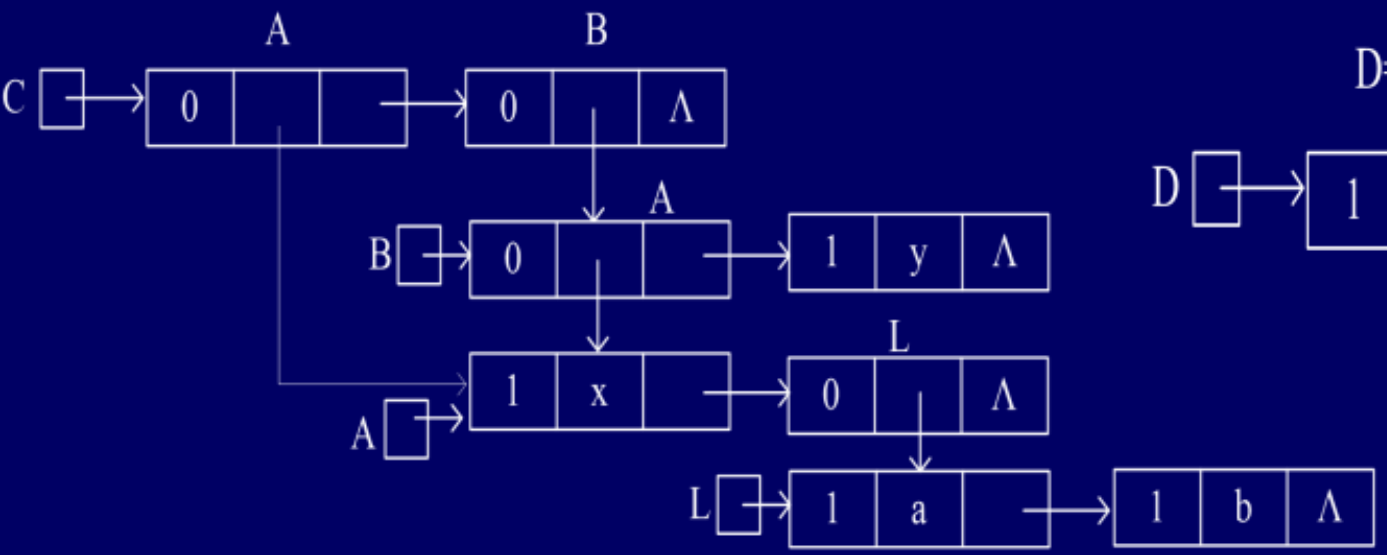
- 结点结构：

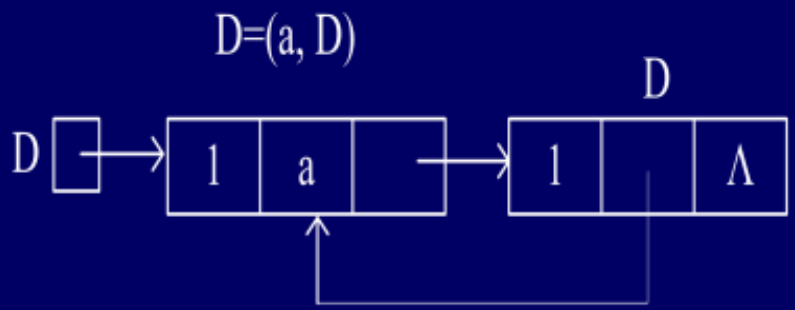
$$atom = \begin{cases} 0 & \text{本结点为子表（slink指向子表的第一个结点）} \\ 1 & \text{本结点为原子（data）} \end{cases}$$

- 图示：E=NIL

C=(A, B)=((x, (a, b)), ((x, (a, b)), y))

=(A(x, L(a, b)), B(A(x, L(a, b)), y))





§5.3 广义表（Lists）

- 存储结构说明

```
typedef struct GLNode{
    int atom; //亦可定义为枚举类型，标志域
    struct GLNode *slink; //指向同层后继
    union {
        struct GLNode *slink; //指向子表的第一个结点
        DataType data；  //原子结点数据域
    }optval;   //候选值
} *GList;
```

§5.3 广义表（Lists）

- 缺点：

① 若要在某一表中开始处插入或删除一个结点，则要找出所有指向该结点的指针，逐一加以修改。

例如，上图中，删除A表中的x，修改A的头指针外，须修改来自于B、C表的指针，引用来源点不易知道。

② 删除一个表可能导致错误。

例如，删除A表时，A的所有结点释放后，会引起B、C表的错误。

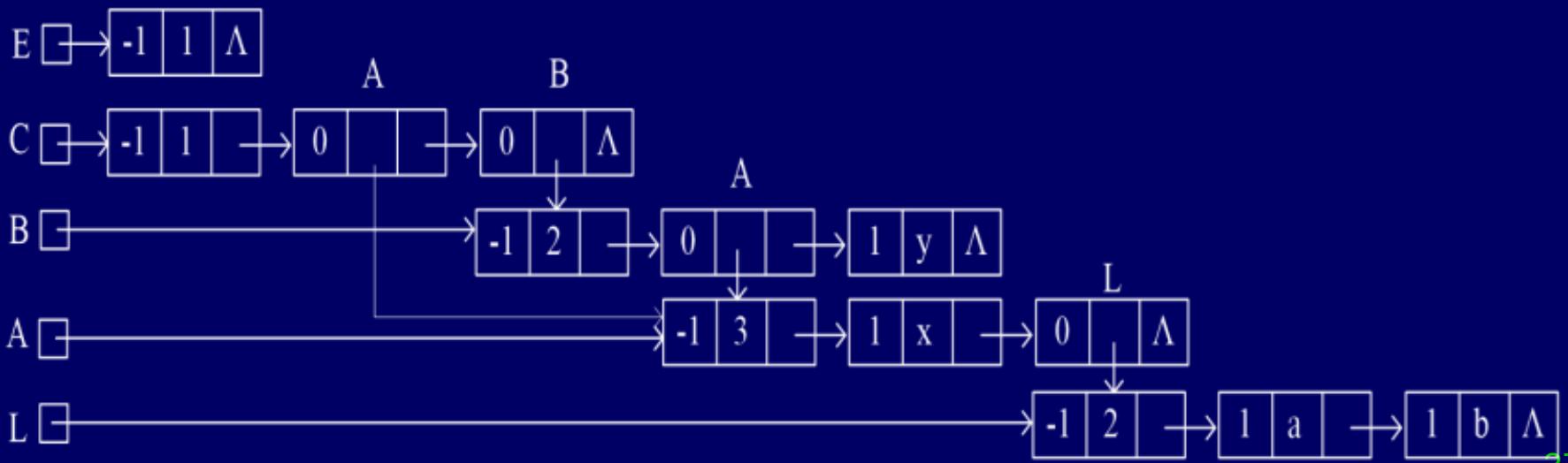
§5.3 广义表（Lists）

- 改进

引入表头结点，使子表内部的变化不会涉及外部元素的变化，插删第一个结点方便。头结点和其他结点结构相同，只是以示区别：

$$\text{atom}=\begin{cases}-1 & \text{表头结点，link域指向表中第一个结点；data域为引用计数}\\ 0 & \text{本结点为子表}\\ 1 & \text{本结点为原子}\end{cases}$$

删除表时，头结点引用计数减1，仅当引用计数为0时，才释放表中所有结点。



§5.3 广义表（Lists）

(3) 双链表示法

该方法类似于第6章的二叉链表。

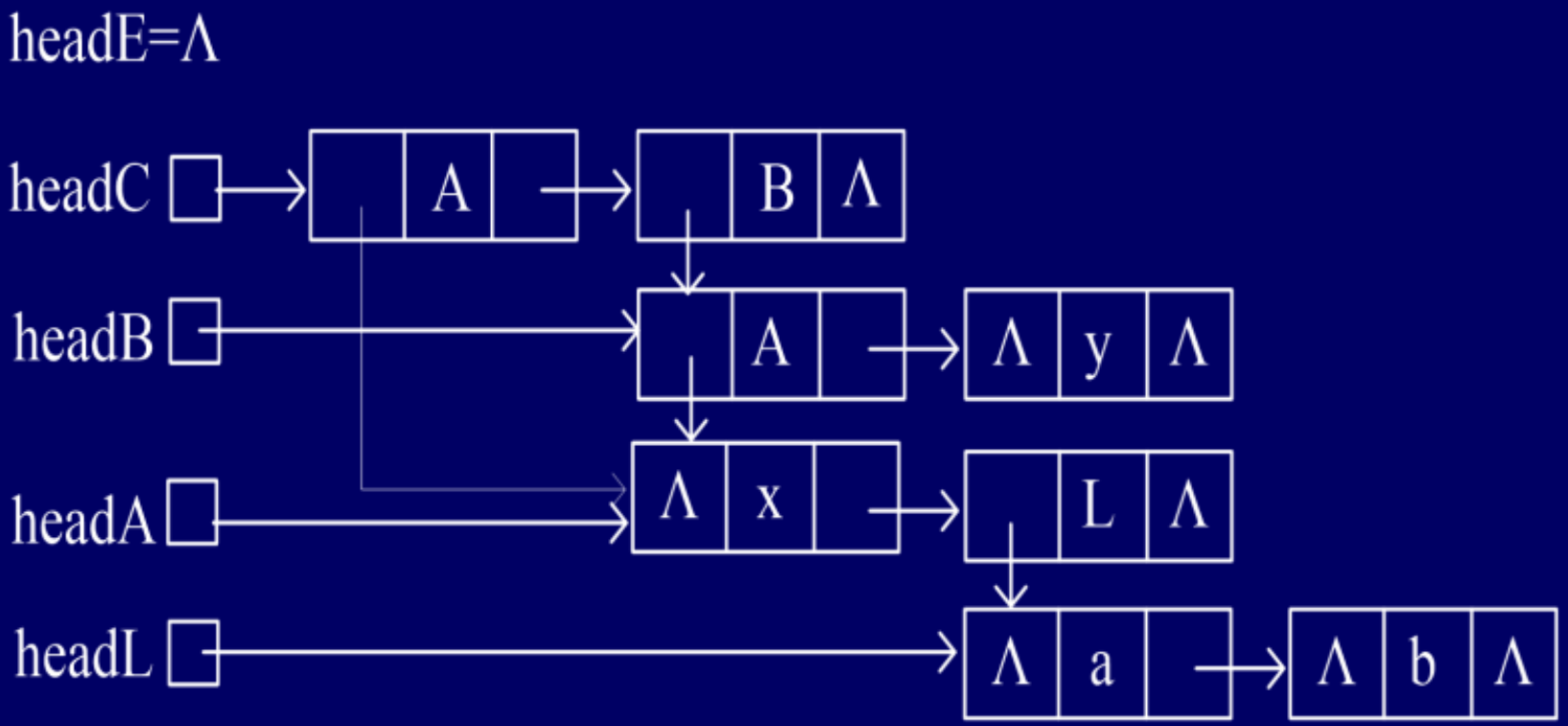
- 结点结构

- 存储说明

```
typedef struct GLNode{
        DataType data; //子表名字或原子数据
        struct GLNode *link1, *link2;
} *GList;
```

§5.3 广义表（Lists）

- 图示



- 特点

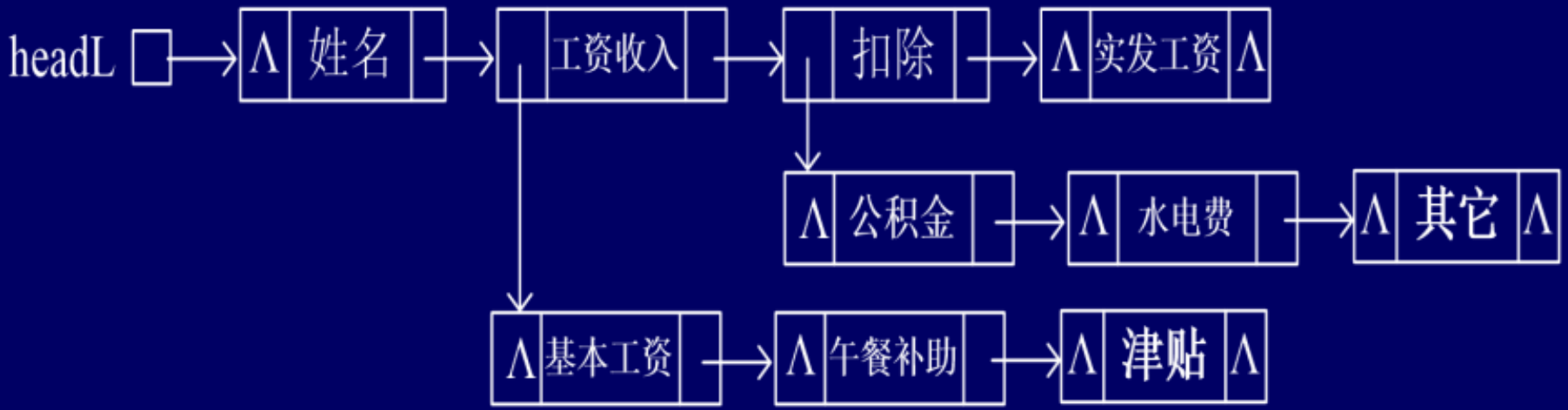
① 简洁方便，类似于二叉链表，可借助于二叉链表表示处理，易求长度深度。

② 在表结点中保存了子表的名字信息，有时子表名字和原子信息同样重要。

§5.3 广义表（Lists）

- 例子

(姓名,工资收入(基本工资,午餐补助,津贴),扣除(公积金,水电费,其它),实发工资)



3. 运算：略

数 据 结 构

Ch.6 树

计 算 机 学 院　　肖明军

Email: xiaomj@ustc.edu.cn

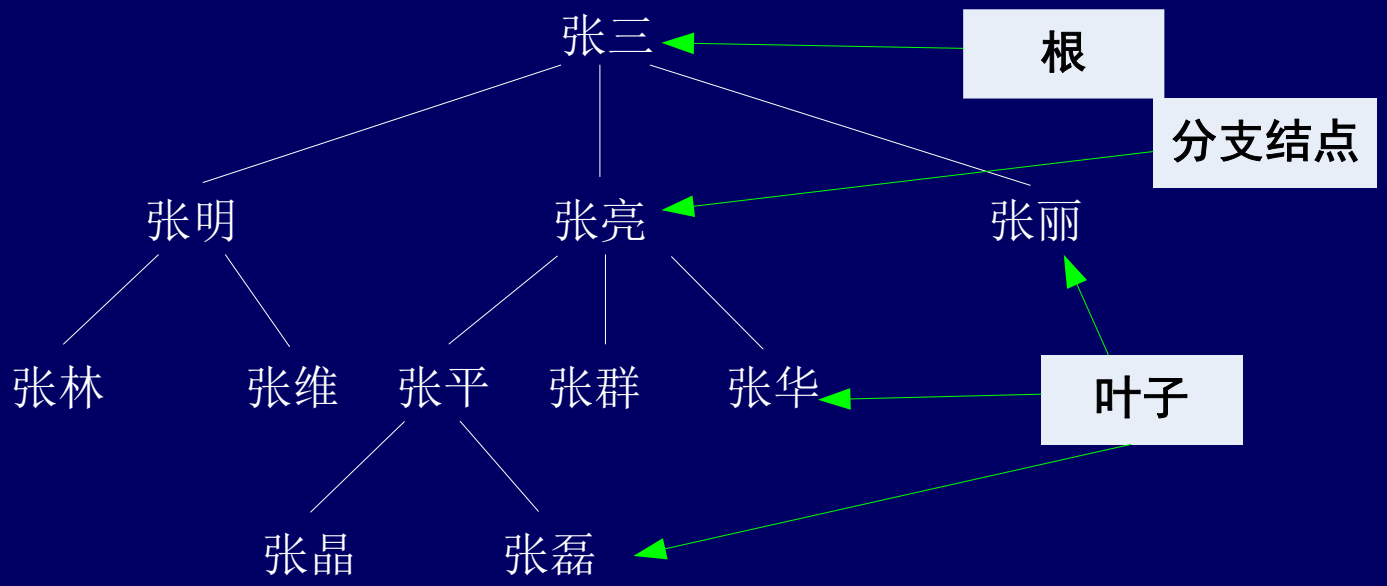
http://staff.ustc.edu.cn/~xiaomj

Ch.6 树

- **树形结构：**
  - 二叉树，树，森林等
  - 结点间有分支，具有层次关系
- **特征：**
  - 每个结点最多只有一个直接前驱，但可有多个直间后继.
  - 开始结点 —— 根
  - 终端结点 —— 叶
  - 其余结点 —— 内部结点
- **应用：家谱、行政架构等，计算机系统中的文件目录等**

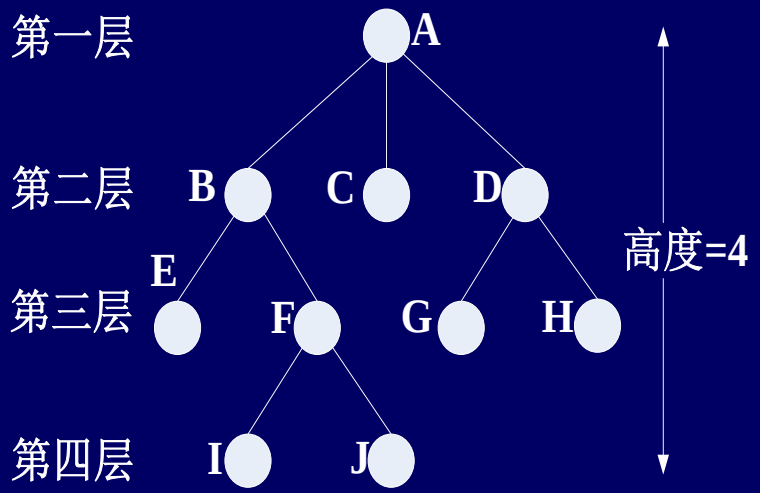
§6.1 树的概念

- **Def: 树是n (n>=0) 个结点的有限集T， T为空时称为空树，否则它满足：**
  - **有且仅有一个特定的称为根的结点；**
  - **其余结点可分为m (m>=0) 个互不相交的子集T_1， T_2，…T_m，其中每个子集本身又是一棵树，并称之为根的子树.**

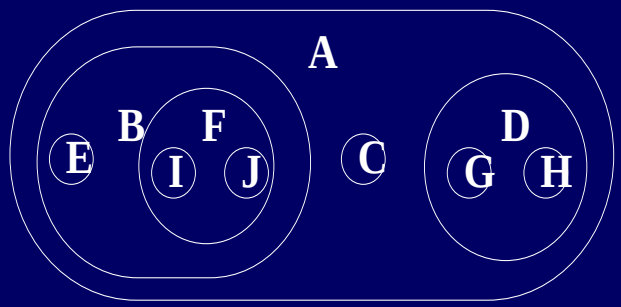


§6.1 树的概念

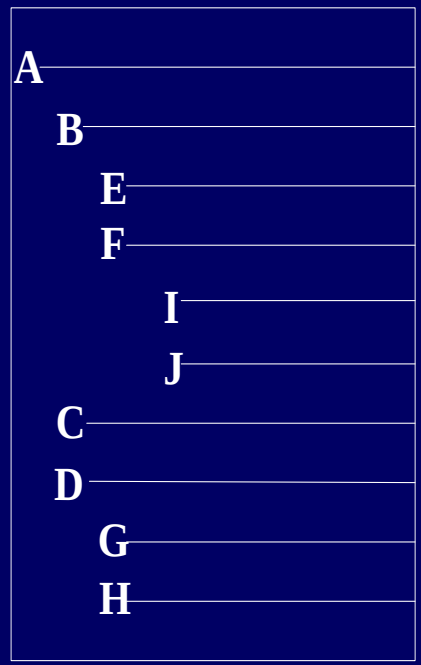
- 递归定义：刻画了树的固有特性：非空树由若干棵子树构成，子树由较小子树构成.
- 表示：



（a）树形表示法



（b）嵌套集合表示法



（c）凹入表表示法

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

（d）广义表表示法

§6.1 树的概念

- 术语：
  - ① 结点的度：结点拥有的子树数目（树的度）
  - ② 叶子：终端结点，度为0的结点
  - ③ 分支结点：非终端结点，度＞0
  - ④ 内部结点：根之外的分支结点
  - ⑤ 根：开始结点
  - ⑥ 孩子、双亲：某结点的子树的根称为该结点的孩子，该结点为孩子的双亲

    直接前驱（双亲） 直接后继（孩子）
  - ⑦ 兄弟：同一双亲的孩子互为兄弟

§6.1 树的概念

- 术语：

⑧ 路径：道路（自上而下）

若存在一结点序列$k_1,k_2,\ldots,k_j$，使得k_i是k_{i+1}的双亲（$1<=i<=j-1$），则称该序列是从k_1到k_j的一条路径或道路，路径长度为边数j-1.

⑨ 祖先和子孙：若k到k_s有一路径，则k是k_s的祖先，k_s是k的子孙.

- ✓ 结点 A 的祖先是从根到 A 的路径上所经过的所有结点.
- ✓ 结点 A 的子孙是以 A 为根的子树中的所有结点.
- ✓ 真祖先和真子孙不包含自身

⑩ 层数从根起算（为 1 或0），其余结点的层数是其双亲的层数＋ 1

§6.1 树的概念

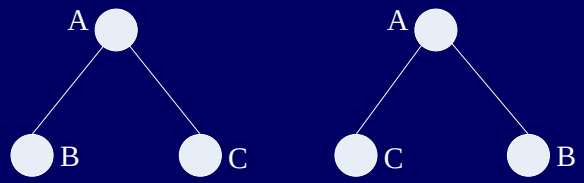
术语：

⑪ 高（深）度：树中结点的最大层数

⑫ 堂兄弟：双亲在同一层的结点互为堂兄弟

⑬ 有序树、无序树：

若每结点的各子树看成是从左到右有次序（不能互换）的，则称为有序树，否则为无序树。



不同的有序树，一般讨论有序树

⑭ 森林：m (m>=0) 棵互不相交的树的集合

树和森林非常接近：删去树根 ＝＞ 森林

森林加上一根 ＝＞ 树

§6.1 树的概念

- **逻辑特征：**
  - ✓ **父子关系**（非线性关系）：任一结点至多有一直接前驱（双亲）结点，但可有多个直接后继（子女）结点.
  - ✓ **开始结点**：根
  - ✓ **终端结点**：叶

  } 其余结点为内部结点.
  - ✓ **祖先与子孙关系**：是对父子关系的延拓，它定义了树中结点的**纵向关系**.
  - ✓ **横向关系**：有序树定义了同一组兄弟间的从左到右的长幼关系，可将其延拓到结点间的横向次序：k_1和k_2是兄弟，k_1在左，则k_1的任一子孙在k_2的任一子孙的左边.

§6.2二叉树

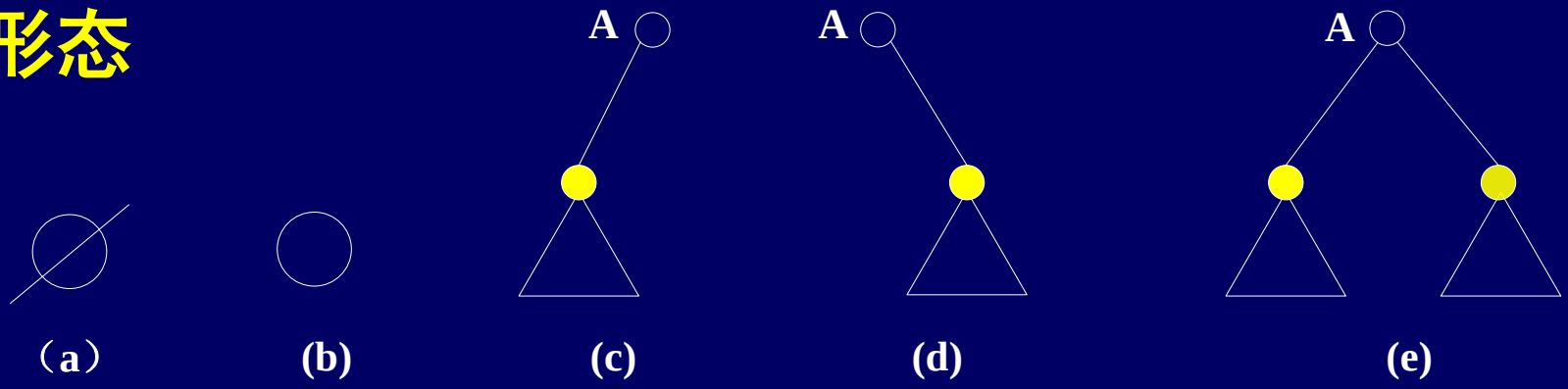
是一种特殊的树型结构，每个结点至多只有二棵子树，一般树可转换为二叉树，计算机用途甚广.

§6.2.1二叉树的定义

- Def: 二叉树是n(n>=0)个结点的有限集，它或者是空集，或由一个根结点及两棵互不相交的，分别称作根的左子树和右子树的二叉树组成

§6.2.1 二叉树的定义

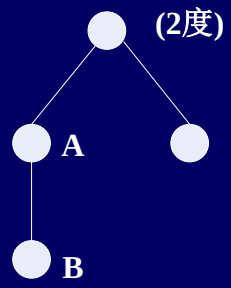
- **形态**



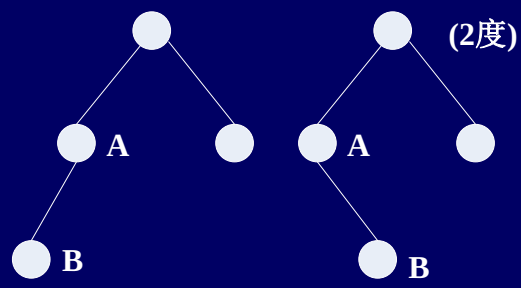
（a） 空二叉树； （b）仅有一个根节点的二叉树； （c）右子树为空的二叉树；
（d）左子树为空的二叉树； （e）左、右子树均非空的二叉树。

- **与度为２的有序树区别：**

当某一结点只有一孩子时，有序树中它理所当然为长子，但二叉树中，一个结点只有一个孩子亦需分出其左右.



一棵普通的有序树



二棵不同的二叉树

§6.2.2 二叉树的性质

1. 性质 1：二叉树的第i层上至多有2^{i-1}个结点（i≥1，根为第1层）

   pf：归纳法

   ① 归纳基础：i=1, $2^{i-1}=1$, 第1层上只有根，故成立.

   ② 归纳假设：设所有的j（1 ≤ j<i）命题成立. 即：

   第j层结点数≤ 2^{j-1}

   ① 归纳步骤：j=i时，第i−1层结点数≤ 2^{i-2}（由归纳假设）

   因为，每个结点至多有两个孩子.

   所以，第i层上结点数≤ $2*2^{i-2}=2^{i-1}$.

§6.2.2 二叉树的性质

2. **性质2.** 深度为h的二叉树至多有2^h-1个结点（h≥1）.

Pf: 深度一定时，仅当每层上结点达到最大时，该树结点最多.

利用性质1知，深度为h的二叉树至多有：

$$2^0+2^1+\ldots+2^{h-1} = 2^h-1$$

§6.2.2 二叉树的性质

3. **性质3.** 在任一二叉树T中，设叶子数为n_0，度为2的结点数为n_2. 则$n_0=n_2+1$.

Pf: 设n_1为度为1的结点总数，则结点总数n等于0度、1度和2度结点数之和

$$n = n_0+n_1+n_2 \quad // \text{二叉树} \qquad (6.1)$$

另一方面，除根外，其余结点均是其双亲的孩子，树中：

孩子结点总数 = n_1+2n_2 ⇨ $n-1 = n_1+2n_2$

$$n = n_1+2n_2+1 \qquad (6.2)$$

由6.1和6.2：$n_0=n_2+1$

§6.2.2 二叉树的性质

4. 满二叉树：深度为h的具有2^h-1个结点的二叉树称为满二叉树.

5. 完全二叉树：若一二叉树至多只有最下两层上结点的度数可小于2，且最下一层上的结点都集中在该层最左边的若干位置，则称为完全二叉树.

某结点无左孩子，则它必为叶子

满二叉树是完全二叉树，但反之未必成立.

§6.2.2 二叉树的性质

6. 性质4．具有n个结点的完全二叉树高为 $\lfloor \lg n \rfloor + 1$ 或 $\lceil \lg(n+1) \rceil$

pf: ∵树高为h，则前h-1层是高为h-1的满二叉树，结点总数 $= 2^{h-1}-1$.

∴ $2^{h-1}-1 < n \le 2^h-1$ ($2^{h-1} < n+1 \le 2^h$) // 第h层上至少有一个结点

$2^{h-1} \le n < 2^h$ //整数

$h-1 \le \lg n < h$ ($h-1 < \lg(n+1) \le h$)

∵ h-1和h是相邻的整数

∴ $h-1 = \lfloor \lg n \rfloor$ ($h = \lceil \lg(n+1) \rceil$)

§6.2.3 二叉树的存储结构

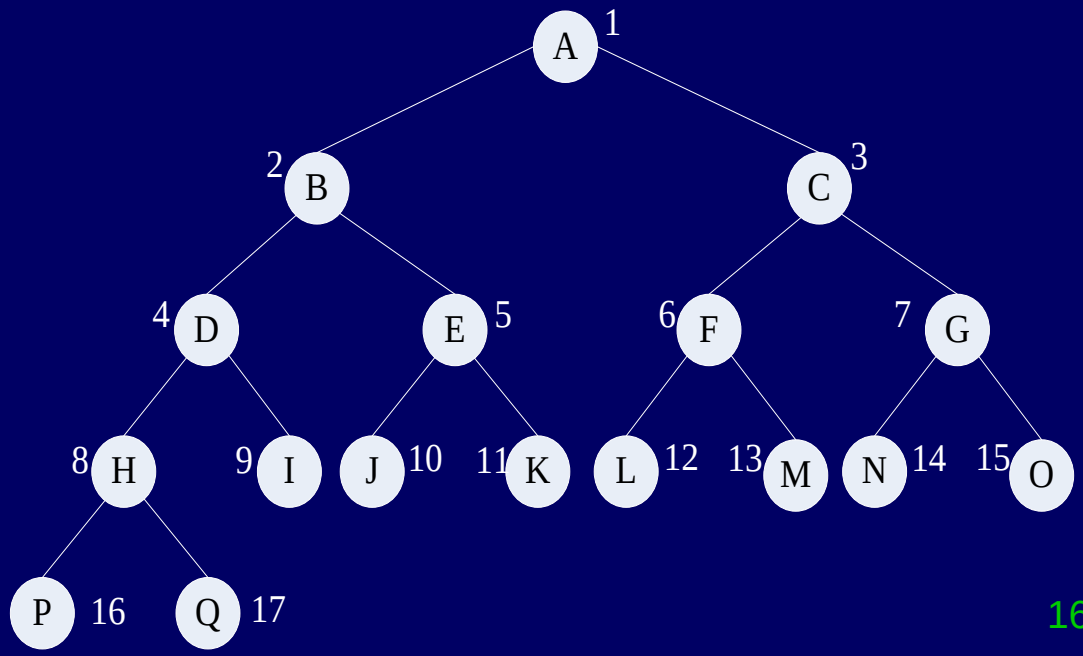
1. 顺序存储结构

如何将结点线性化，使得在线性序列中的相互位置能反映出结点间的逻辑关系？

若对完全二叉树自上而下，每层自左到右给所有n个结点编号，就能到一个足以反映整个二叉树结构的线性序列。

∵ 完全二叉树除最下层外，各层都充满了结点，每层结点数恰为上层的2倍.

∴ 从一结点编号可推出其双亲，左右孩子，兄弟结点和编号.



§ 6.2.3 二叉树的存储结构

- **性质5. 设完全二叉树中编号为i的结点简称k_i(1≤ i≤ n), 则有：**

(1) 若i=1，则结点k_i为根，无双亲；若i>1，则k_i的双亲是 $k_{\lfloor i/2 \rfloor}$

(2) 若2i≤n, 则k_i的左孩子为k_{2i}；否则k_i无左孩子，即它必为叶子。//因此，完全二叉树中编号i>$\lfloor n/2 \rfloor$ 的结点必为叶子；

(3) 若2i+1≤n, 则k_i的右孩子为k_{2i+1}，否则k_i无右孩子；

(4) 若i为奇数且大于1，则k_i的左兄弟为k_{i-1}，否则k_i无左兄弟；

(5) 若i为偶数且小于n，则k_i的右兄弟为k_{i+1}，否则k_i无右兄弟。

证明从略。

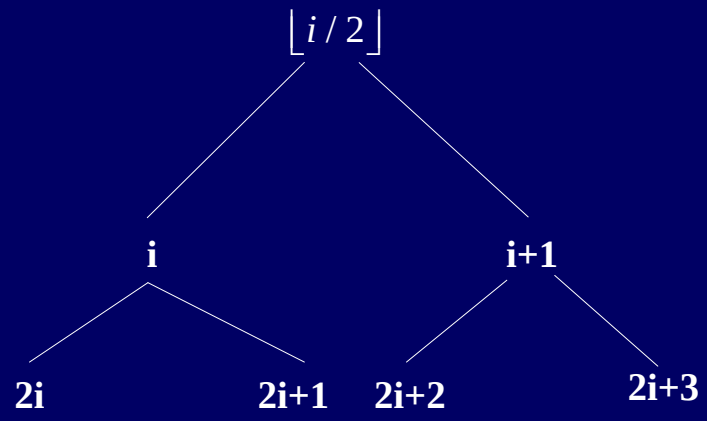
§6.2.3 二叉树的存储结构

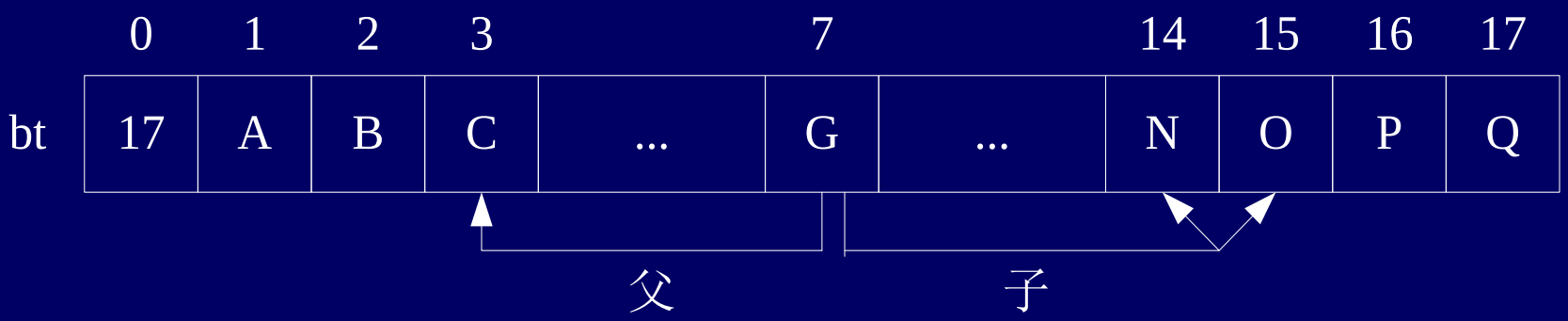
∵ 上述关系中，编号足以反映结点间的逻辑关系

∴ 可将n个结点存储在向量bt[0..n]中，其中：

bt[0] —— 不用，或存储n

bt[1..n] —— 存储编号为1至n的结点



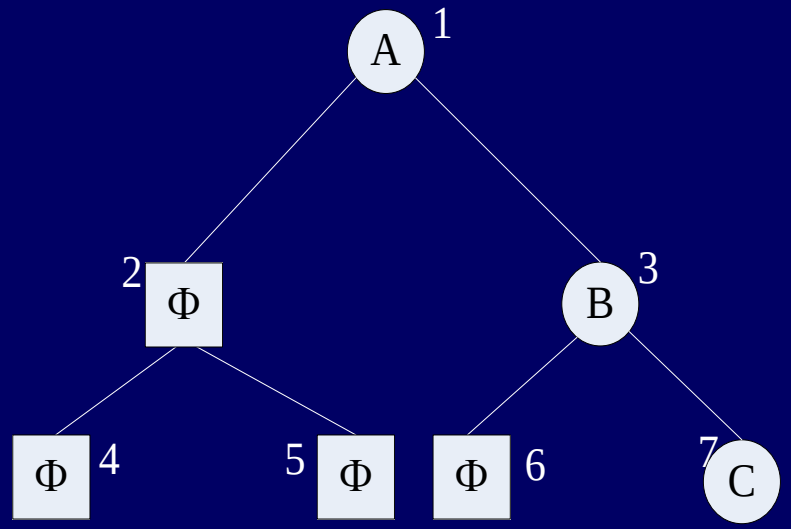


§ 6.2.3 二叉树的存储结构

- **缺点**

对一般的二叉树，须按完全二叉树的编号来存储，浪费空间，最坏情况是右单支树，k个结点需2^k-1个结点空间。

	0	1	2	3	4	5	6	7
bt	3	A	Φ	B	Φ	Φ	Φ	C



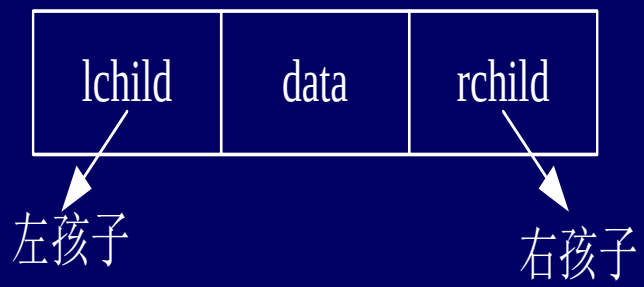
- **结论**

这种结构只适用于存储完全二叉树，且插入和删除不便。

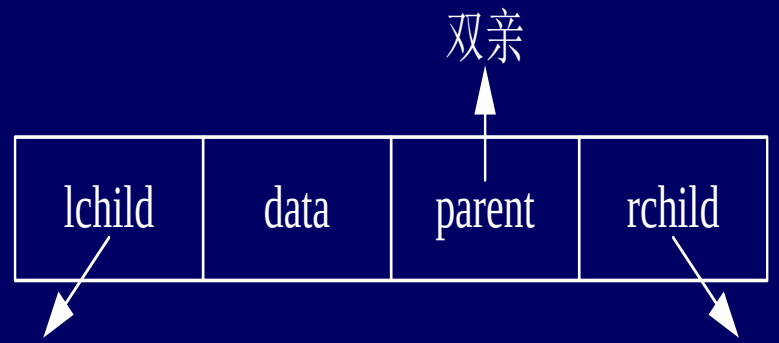
§ 6.2.3 二叉树的存储结构

2. 链式存储结构

- 结点结构



(a) 二叉链表中结点



(b) 带双亲的二叉链表

- 类型定义

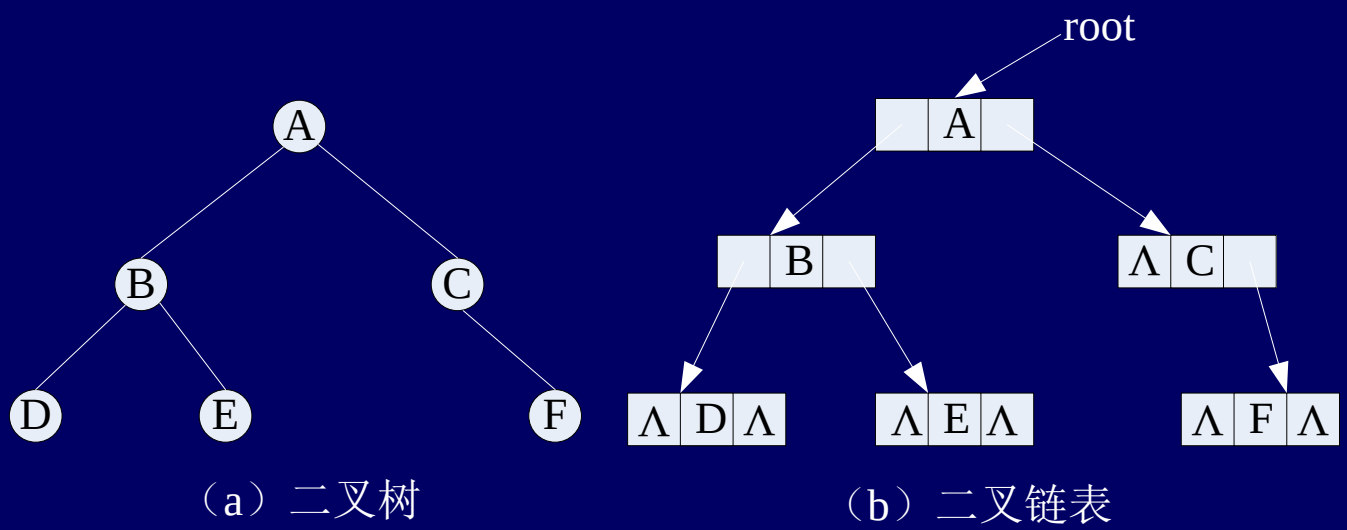
```
typedef struct node {
        DataType data;
        struct node *lchild, *rchild;
 } BinTNode; // 结点类型
 typedef BinTNode *BinTree; //二叉树类型
```

§6.2.3 二叉树的存储结构

在二叉树中，所有类型为BinTNode的结点，加上一个指向根的BinTree型头指针root，就构成了二叉树的链式存储结构，称之为**二叉链表**

显然，二叉链表由根指针root唯一确定。

- 例子



（a）二叉树　　（b）二叉链表

- 特性

空树 ⇔ root = NULL

叶子 ⇔ 左右指针为空

空指针数：n个结点，共有2n个指针域，但只有n-1个结点是别人的孩子，故空指针数为n+1

§6.3 遍历二叉树

1. 概念

- **定义**：沿某搜索路线，依次对树中每个结点均做一次且仅做一次访问。
- **重要性**：是其它运算的基础，很多树上操作均依赖于遍历操作，只是访问结点所做的操作不同。
- **如何遍历？**

  遍历**线性结构**很容易：从开始结点出发，依次访问当前结点的后继，直至终端结点为止。遍历路线只有一条(如单链表，从头指针开始)。

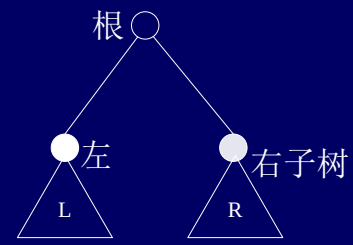
  但二叉树中每个结点可能有**两个后继**，故**遍历路线不唯一**，须找到**适用于每个结点**的相同的遍历规则。

$\lim_{n\to\infty} T(n)/n^3 = \lim_{n\to\infty}(2n^3+3n^2+2n+1)/n^3 = 2$

§6.3 遍历二叉树

∵ 在二叉树的递归定义中，非空树组成为：

D、L、R



∴ 在任一结点上，可按某种次序执行三个操作：

访问根结点(D)，遍历该结点的左子树(L),遍历该结点的右子树(R)

显然有六种执行次序：

$$\left.\begin{array}{l}\text{1. 从左到右：}\quad \text{DLR，LDR，LRD}\\ \text{2. 从右到左：}\quad \text{DRL，RDL，RLD}\end{array}\right\}\text{二者对称，只讨论前3种}$$

遍历规则(从左到右)

DLR,LDR,LRD的差别是访问根的先后次序不同

① 前序(先序，先根)遍历：DLR

② 中序(中根)遍历：LDR

③ 后序(后根)遍历：LRD

§6.3 遍历二叉树

2. 遍历算法

以中根为例，遍历二叉树定义为：

```
if 二叉树非空 then {
        (1) 遍历左子树 // 即遍历二叉树
        (2) 访问根        // 将(1)(2)和(2)(3)对调后为先根和后根遍历
        (3) 遍历右子树 //即遍历二叉树
} // 否则为空操作 (递归结束条件)

void Inorder(BinTree T) { // T为二叉树的头指针
        if (T) { // T非空，T为空时为空操作
                Inorder(T->lchild);        // 递归遍历左子树
                printf("%c", T->data);  // 访问根结点，具体问题，此
                                                // 操作不同
                Inorder(T->rchild);       // 递归遍历右子树
        }
}
```

- 时间：O(n)

§6.3 遍历二叉树

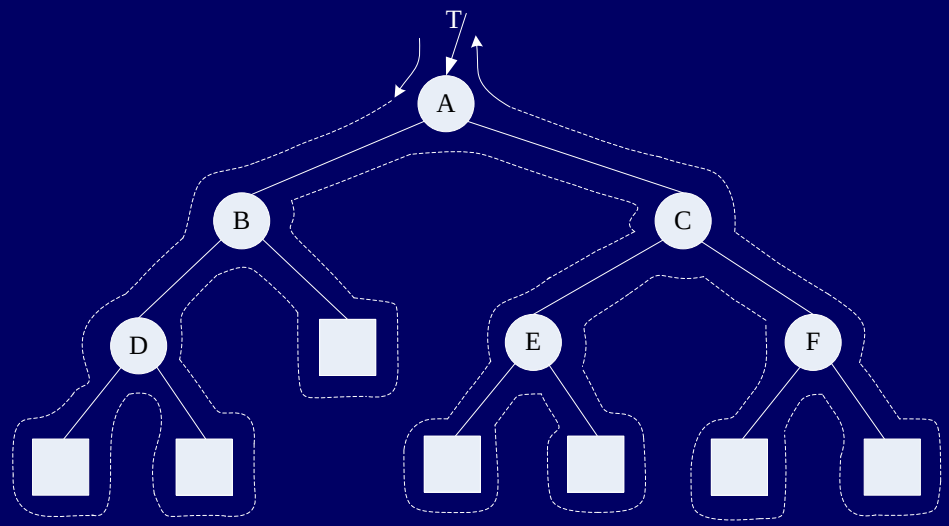
3. 遍历序列

包络线是递归遍历路线

向下：表示递归调用，更深一层

向上：表示递归结束，返回一层

每个结点经过3次，第1次经过时访问所得结点序列为前序，第2次经过时访问所得结点序列为中序，第3次经过时访问所得结点序列为后序。



前序序列：ABDCEF

中序序列：DBAECF

后序序列：DBEFCA

} 线性序列：

1个开始结点，1个终端结点，其余结点均有一个直接前驱和一个直接后继，为区别3种次序在前面冠以

前序
中序
后序 } + { 前驱
后继

▲ 叶子的相对次序相同

§6.3 遍历二叉树

4. 通用的遍历算法

因为访问结点的操作依赖于具体问题，故可将它作为一个函数指针参数放于遍历算法的参数表中，调用时，使其指向具体的访问结点的应用函数

```
void Inorder( BinTree T, void (*Visit)(DataType x) ) {
    if (T) {
        Inorder(T->lchild, Visit);
        (*Visit)(T->data);
        Inorder(T->rchild, Visit);
    }
}
```

其中Visit是一函数指针，它指向形如void f(DataType x)的函数，故可将访问结点的操作写在函数f中，通过调用语句：

```
Inorder(root, f);
```

将f的地址传给Visit。

§6.3 遍历二叉树

4. 通用的遍历算法

例如，可将打印操作作为

```
void print(DataType x) {
    print("%c", x);
}
```

调用Inorder(root, print)，即可完成前述算法。

▲ 函数名：函数代码的起址。

§6.3 遍历二叉树

5. 建立二叉链表

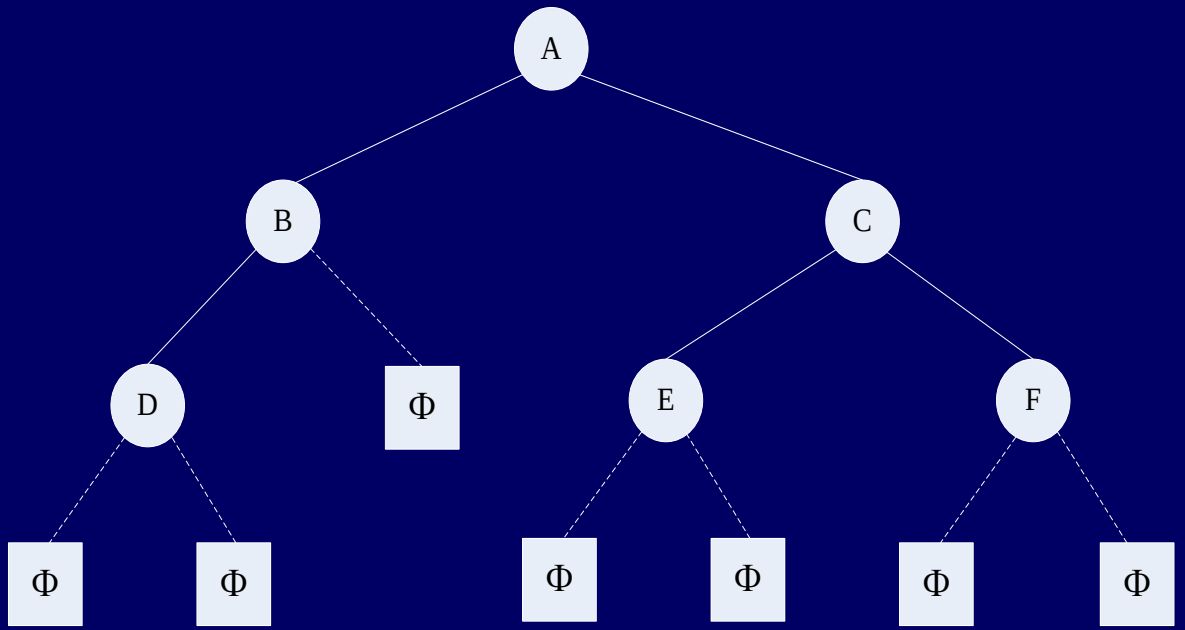
上述算法假定二叉链表已建立

建立二叉树对应的二叉链表方法很多

- **先序遍历构造法**

输入先序序列，加入虚结点(输入时用空格符“ ”)以示空指针的位置，例如前述的二叉树，输入为：

ABDΦΦΦCEΦΦFΦΦ



§6.3 遍历二叉树

```
void CreateBinTree(BinTree *T) {   // 注意T为指针的指针
        char ch;
        if ((ch = getchar()) == '\n')  return; // 回车结束输入
        if (ch == ' ') // 读入空格
                *T = NULL;  // 将相应的指针置空
        else { // 读入的是结点数据
                *T = (BinTNode*) malloc(sizeof(BinTNode));
                (*T)->data = ch; // 生成新结点，相当于访问根节点
                CreateBinTree(&(*T)->lchild); // 遍历左子树
                CreateBinTree(&(*T)->rchild); // 遍历右子树
         }
 }
```

建树时调用 CreateBinTree(&root)，将root(BinTree类型)的地址复制给T，故修改*T就相当于修改了实参root本身。

时间：O(n)

§6.4 线索二叉树

1. 基本概念

在一基本数据结构上常常需要扩充，增加辅助信息，其目的是：

①开发新操作；②加速已有操作。

- 线索 —— 利用空指针域(n+1个)存放指向结点在某种遍历次序下的前驱和后继指针
- 线索链表 —— 加上线索的二叉链表
- 线索二叉树 —— 相应的二叉树称为线索二叉树
- 目的：加速遍历操作

  加速查找任一结点在某种遍历次序下的前驱和后继操作
- 如何区别结点的指针域

  孩子指针：指向孩子？

  线索指针：指向其某种遍历次序下的前驱和后继的线索？

§6.4 线索二叉树

1. 基本概念

- 结点结构

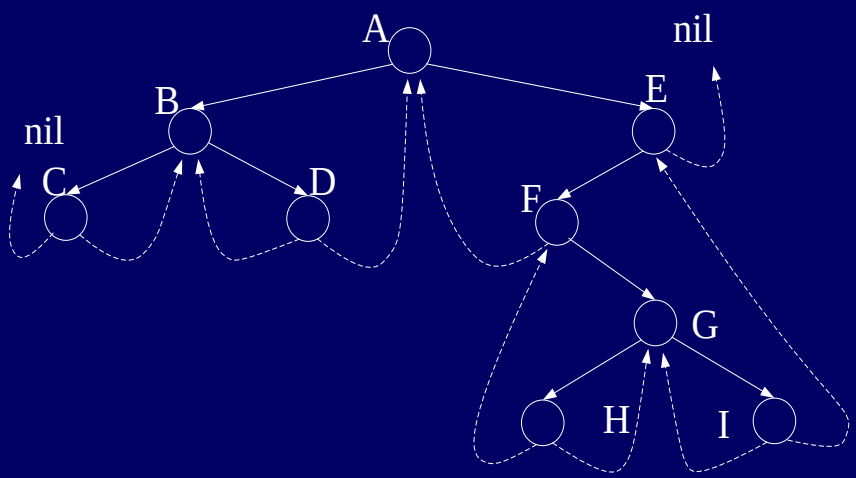
lchild	ltag	data	rtag	rchild

- 线索标志

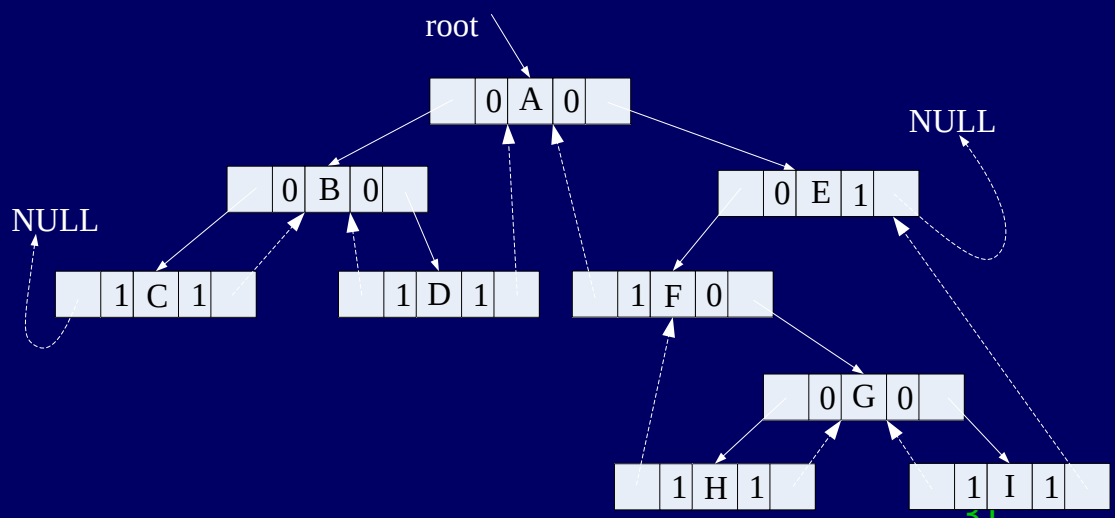
$$\text{左标志ltag} = \begin{cases} 0: \text{lchild为左指针，指向孩子} \\ 1: \text{lchild为左线索，指向前驱} \end{cases}$$

$$\text{右标志rtag} = \begin{cases} 0: \text{rchild为右指针，指向孩子} \\ 1: \text{rchild为右线索，指向后继} \end{cases}$$

- 中序线索树和中序线索链表　**中序序列：CBDAFHGIE**



（a）中序线索二叉树



（b）中序线索链表

§6.4 线索二叉树

1. 基本概念

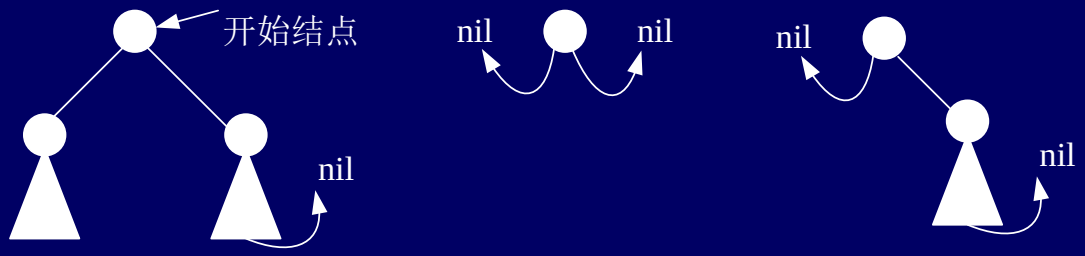
- 中序线索树中必有两个指针域为空

中序序列{开始结点，左线索为NULL；终端结点，右线索为NULL}中序为对称序

中序序列{开始结点为最左下的结点；终端结点为最右下的结点}

- 前序线索树中，有几个指针域为空？

前序序列的开始结点为根，故当它的左子树非空时，其指针域指向左指子树，此时前序序列开始结点左指针非空。



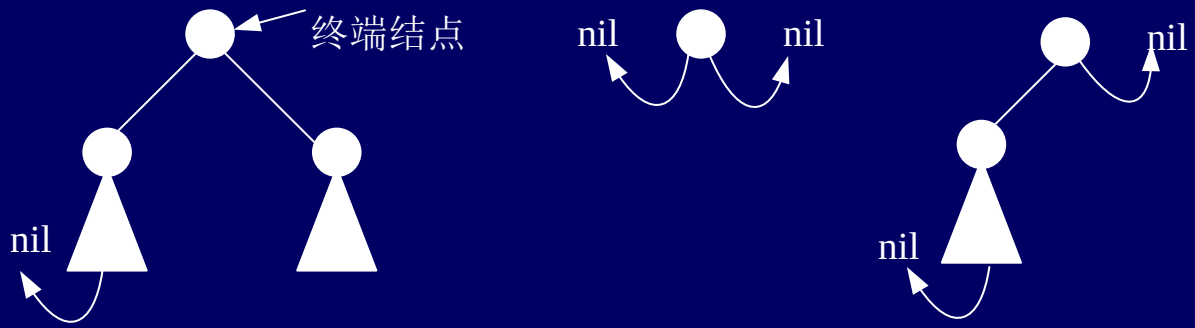
但是，前序序列的终端结点的右指针必为空。

仅当只有1个根结点或根的左子树为空时有两个空指针。

§6.4 线索二叉树

1. 基本概念

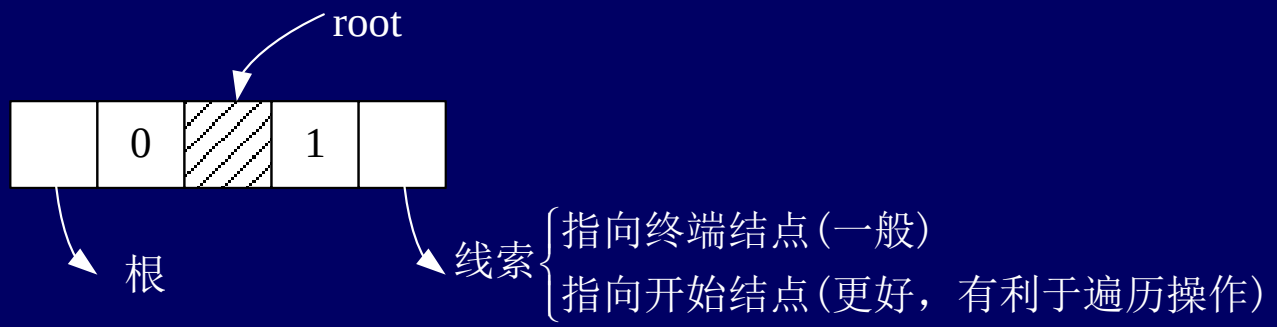
- 后序线索树中，开始结点的左指针必为空，仅当只有1个根时或根的右子树为空时有两个空指针。



与前序线索树对称

- 带头结点的线索链表

书上6.11图(b). 令空指针也指向此哨兵



§6.4 线索二叉树

1. 基本概念

- 线索树中，如何判定结点是否为叶子？

  ltag = rtag = 1 (适用于三种线索树)

- 有时线索树只有左线索或右线索之一

2. 线索化

将二叉树变为线索树的过程

按某种次序遍历，在遍历过程中用线索取代空指针。

```
typedef enum {Link, Thread} PointerTag; // 0为Link，1为Thread
typedef struct node {
        DataType data;
        PointerTag ltag, rtag;
        struct node *lchild, *rchild;
  } BinThrNode, *BinThrTree;
```

设p和pre分别指向遍历过程中当前访问结点和其前驱，即*pre和*p是前驱和后继的关系，其中pre为全局量，在遍历过程中建立线索。

以中序为例，pre初始为NULL，因为中序前驱对开始结点是NULL。

§6.4 线索二叉树

```
void InorderThreading(BinThrTree p) {
   // pre为全局量，初值为NULL
   if (p) {
      InorderThreading(p->lchild); // 左子树线索化
      p->ltag = (p->lchild) ? Link : Thread; // 左指针非空，置为
                                             // Link，否则为线索。
      p->rtag = (p->rchild) ? Link : Thread;
      if (pre) { // 若*pre存在
         if (pre->rtag == Thread) // 当前结点*p的前驱右标志为线索
            pre->rchild = p;      // 令*pre的右线索指向中序后继*p
         if (p->ltag == Thread)   // 当前结点的左线索已建立
            p->lchild = pre;      //令当前节点的左线索指向中序前驱
      }
      pre = p; //使*pre为*p的前驱，循环不变量
      InorderThreading(p->rchild); // 右子树线索化
   }
}
```

访问根结点

- 时间：和遍历相同O(n).　　后序(前序)线索化类似于此。

§6.4 线索二叉树

3. 操作(加速)

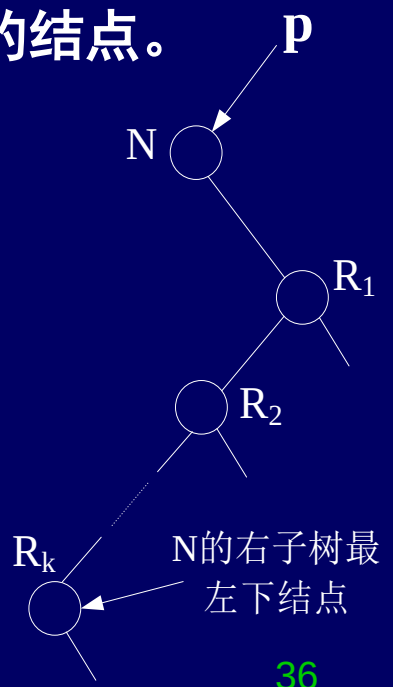
(1) 找某结点*p(中序线索树中)中序前驱和后继结点

① 找*p的中序后继

i） 若*p的右子树空，则p->rchild为右线索，直接指向*p的中序后继

ii）若*p的右子树非空(rtag为0)，则*p的中序后继必是其右子树中第一个中序遍历到的结点，其特征是：

从*p的右孩子开始，沿其左链往下找，直至找到一个没有左孩子的结点为止，不妨称其为右子树中“最左下”的结点。



这里k≥ 1, R_k不一定是叶子，其右子树可为空，可非空。

- 算法

请自己给出找给定结点*p的中序后继算法。时间O(h)，快于无线索的二叉树。

§6.4 线索二叉树

- **加速作用**

在普通二叉树中，找*p中序后继：

对于ii）同样有效

对于 i）就必须从根开始遍历。

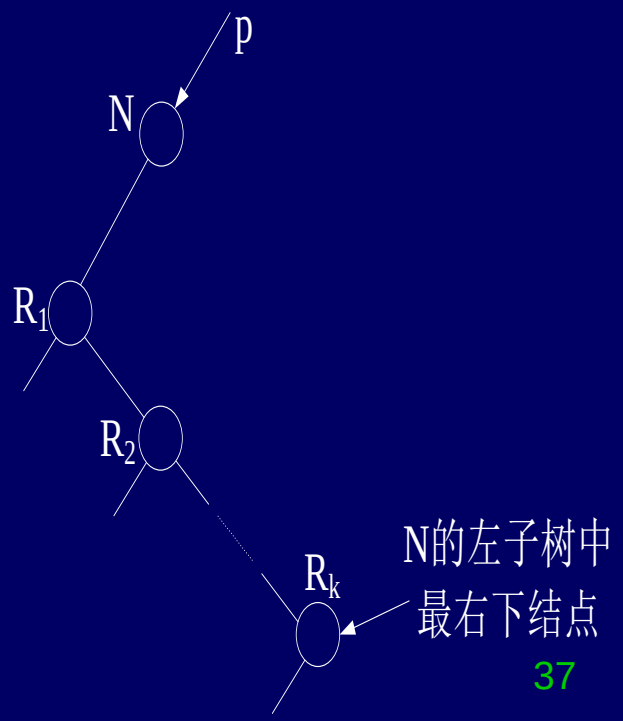
∵有线索是直接从线索找到O(1)。但线索一般“向上”指向其祖先，而二叉树中无向上的链接，只能从根开始遍历得到，最坏情况O(n)。

② **找*p的中序前驱**

∵ 中序是对称序，其方法与(1)完全对称

i） 若*p的ltag = 1，则中序前驱为lchild

ii）若*p的ltag = 0，则中序前驱是*p的左子树中 “最右下”的结点。



§6.4 线索二叉树

(2) 找*p的后序前驱和后序后继(在后序线索树中)

① 找*p的后序前驱(易)

i） 若*p的ltag = 1(左子树为空)，则后序前驱为p->lchild

ii）若*p的ltag = 0(左子树非空)，则后序前驱为p的左孩子或右孩子

∵根是在遍历左右子树L和R之后被访问

∴*p的前驱必是L和R中最后一个遍历到的结点

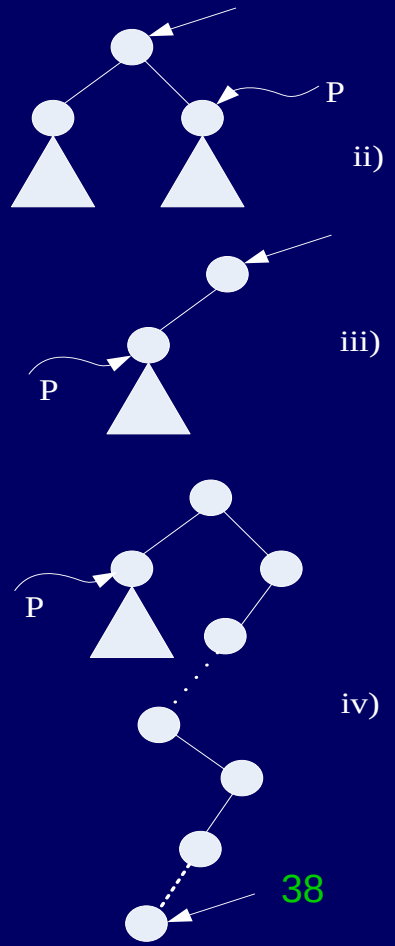
```
if (p的右孩子非空) then
        return p->rchild; // 后序前驱为p的右孩子
else
        return p->lchild; //后序前驱为p的左孩子
```

② 找*p的后序后继(难)(见右图)

i） 若*p为根，无后继，返回NULL

ii） 若*p是其双亲右子，则*p的后序后继是双亲

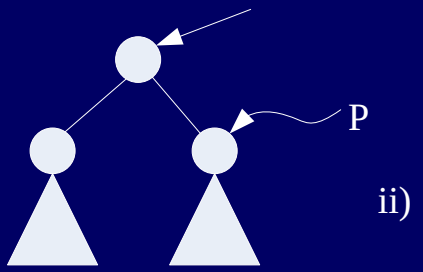
iii）若*p是其双亲左子，但*p无右兄弟，则*p的后序后继亦为双亲



§6.4 线索二叉树

② 找*p的后序后继(续)

iv）若*p是其双亲左子，但*p有右兄弟，则*p的后序后继是其右兄弟子树中1st后序遍历到的结点，它是该子树中“最左下的叶结点”

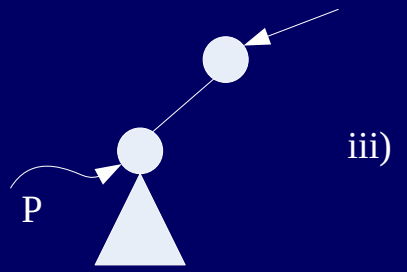


结论：找后序前驱易

找后序后继难，因为：

只有*p的右子树为空(rtag = 1)时，p->rchild是线索，可直接找到；

否则，一般须涉及*p的双亲，故仅给出*p时，须从根遍历。

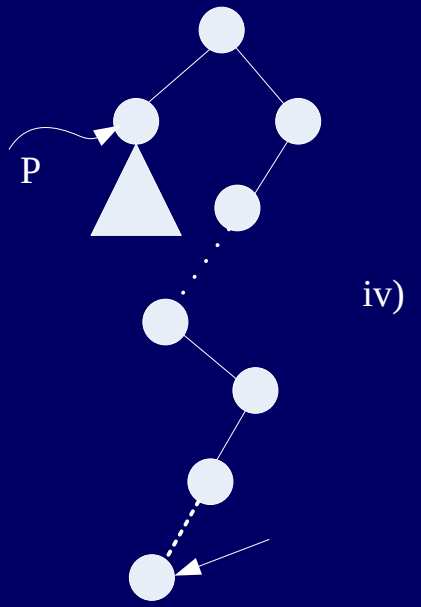


(3) 找*p的前序前驱和前序后继

类似于后序的情况分析

讨论：找前序前驱难(涉及双亲)

找前序后继易



§6.4 线索二叉树

(4) 遍历线索二叉树

遍历某次序的线索二叉树，只要从该次序下的开始结点出发，反复找其后继直至终端结点为止。

- 中序

  找开始结点(最左下结点)，找当前结点中序后继，直至终端结点(p->rchild = NULL)　头结点的右指针指向开始结点较方便

- 前序

  找开始结点(根)，找当前结点的前序后继，直至终端结点(p->rchild = NULL)　头结点的右指针指向终端结点较方便

- 后序

  找终端结点(根)，找当前结点的后序前驱，直至开始结点(p->lchild = NULL)，得到的是后序序列的逆序列

  头结点的右指针指向开始结点较方便

- 时间：仍为O(n)，但因为非递归，略快于递归的方法

- 对遍历而言

  前序线索树中，只需保留右线索树即可

  中序线索树中，保留左、右线索之一均可

  后序线索树中，只需保留左线索即可

§6.5 树和森林

1. 树、森林与二叉树的转换

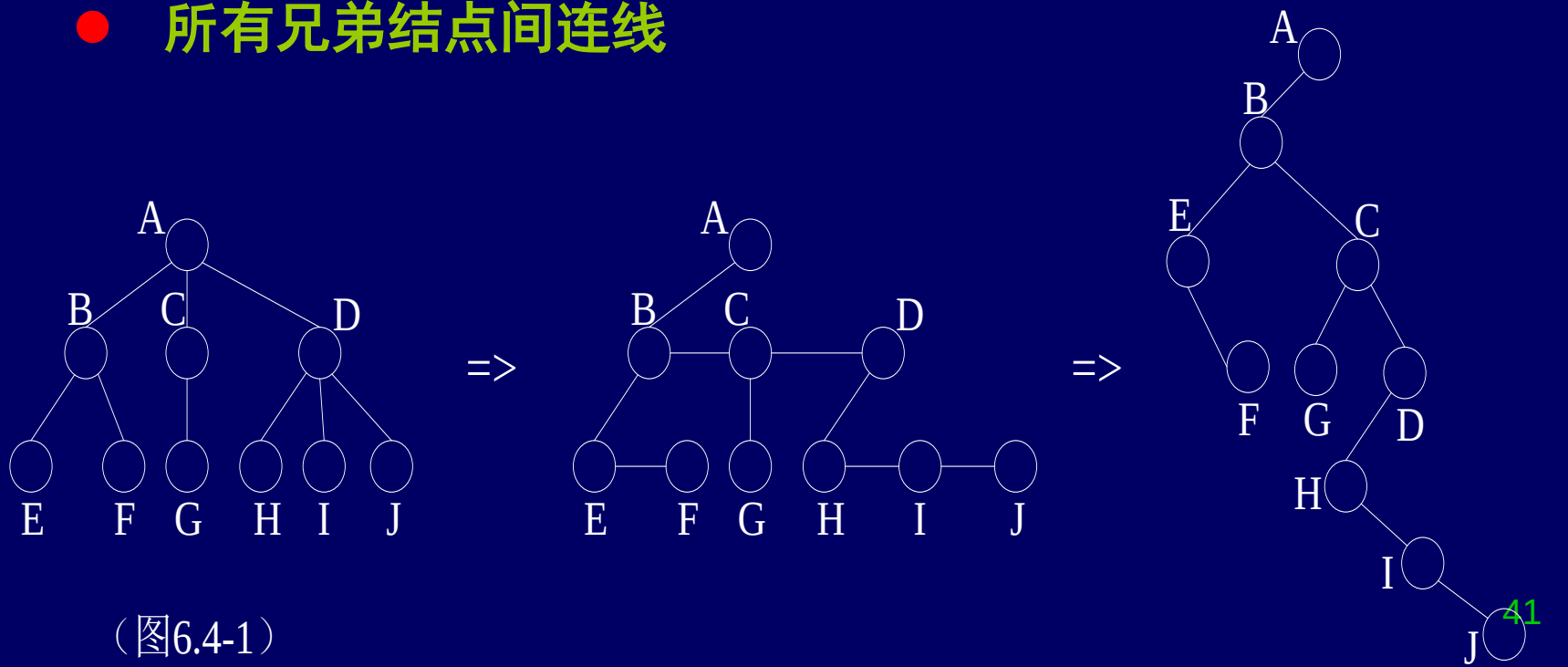
① 树 => 二叉树

树中每个结点有多个孩子 => 二叉树只有两个孩子

长子及右邻兄弟 => 二叉树的左右孩子

节点X的长子是其左子，X的右兄弟是其右子

- 每个结点仅保留与长子的连线
- 所有兄弟结点间连线



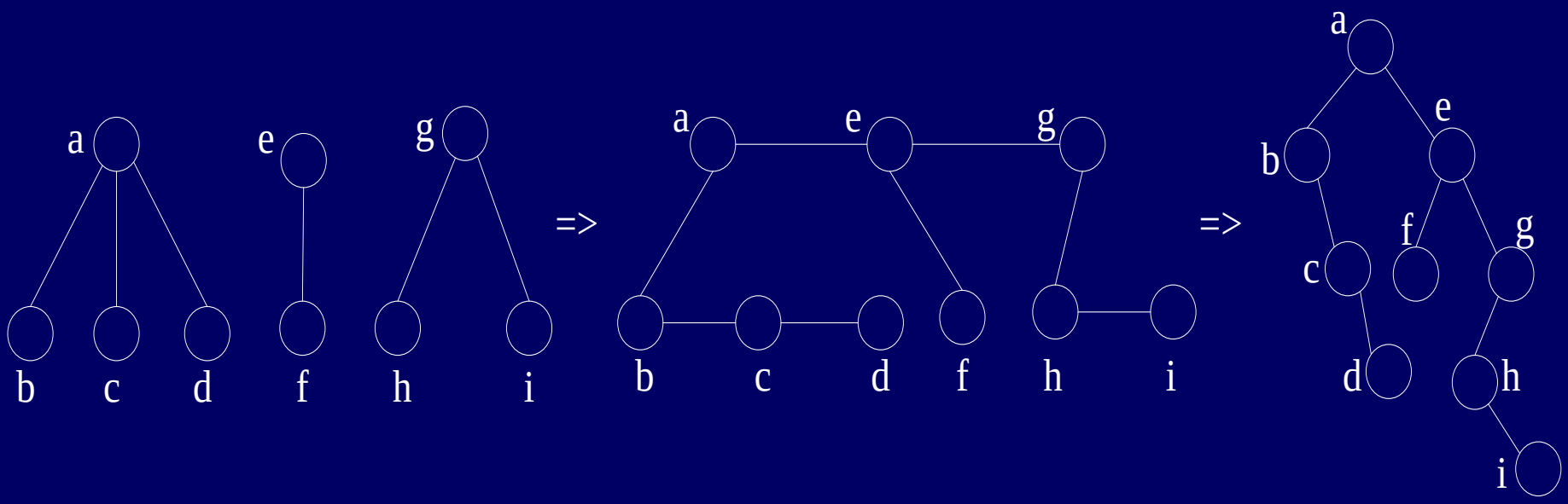
（图6.4-1）

§6.5 树和森林

1. 树、森林与二叉树的转换

② 森林 => 二叉树

- 将各树转换为二叉树(根无右兄弟，所以无右子)
- 将各根作为兄弟互连

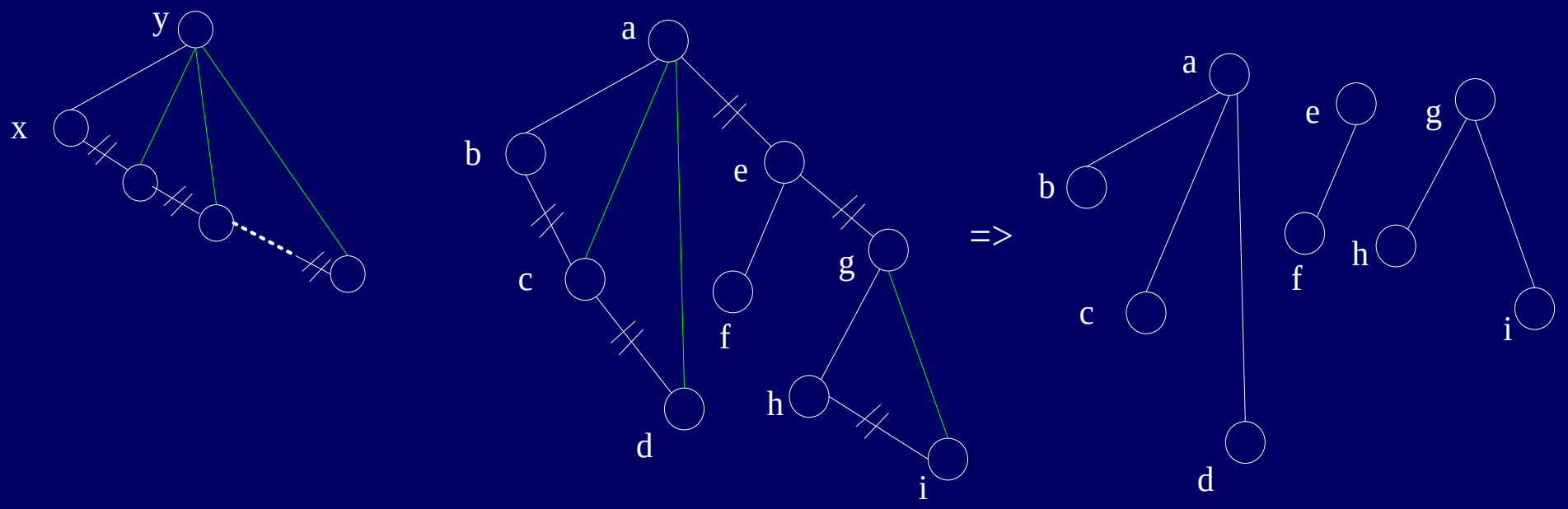


§6.5 树和森林

1. 树、森林与二叉树的转换

③ 二叉树 => 树或森林

- 设x是y的左孩子，则将x的右孩子，右孩子的右孩子，都与y相连
- 去掉所有双亲到右孩子的连线



§6.5 树和森林

2. 树的存储结构

① 双亲链表表示法

∵每结点双亲唯一，故存储结点时，增加一个parent域指向双亲，用向量表示较方便

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

Fig6.4-1对应

- 特点：向上链接，根的parent为-1
  求指定结点双亲(O(1))及祖先(O(h))方便
  求指定结点孩子及后代须遍历数组O(n)
- 类型说明(略)

§6.5 树和森林

2. 树的存储结构

② 孩子链表表示法

- 若k叉树用k叉链表表示，会导致浪费空间

∵ 树边n-1条

∴ 空指针kn-(n-1)=n(k-1)+1

- 设度数域，结点不等长、运算不便
- 孩子链表：每结点设一孩子链表，将结点及相应孩子链表的头指针放在一向量中。

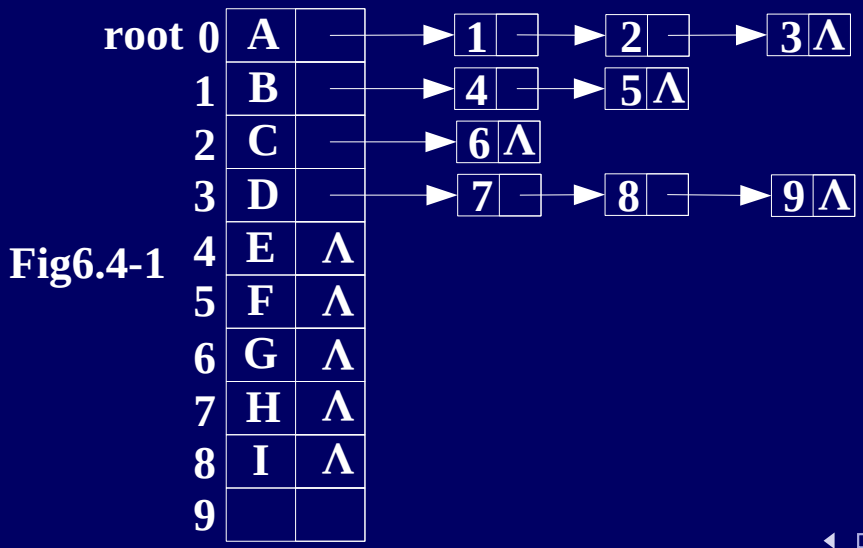


Fig6.4-1

§6.5 树和森林

2. 树的存储结构

② 孩子链表表示法（续）

- 特点：易实现找结点的孩子及子孙(向下查找易)
  难实现找结点的双亲及祖先(向上查找难)
- 双亲孩子链表表示法

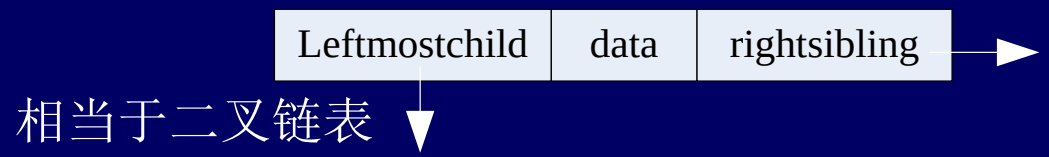
在孩子链表中，增加parent域

此方法结合了双亲链表和孩子链表的优点，向上向下查找均方便

- 类型说明：略

③ 孩子兄弟链表表示法

树 => 二叉树时，结点关系由：最左孩子、右邻兄弟表示



§6.5 树和森林

3. 树和森林的遍历

① 先序遍历树

先访问树的根；然后依次先序遍历根的每棵子树

② 后序遍历树

先依次后序遍历根的每棵子树；然后访问树的根；

③ 先序遍历森林

先访问森林中第一棵树的根；然后先序遍历第一棵树中根结点的各子树所构成的森林；最后先序遍历除第一棵外其它树构成的森林

④ 后序遍历森林

后序遍历森林中第一棵树的根结点的各子树所构成的森林；然后访问第一棵树的根；最后后序遍历除第一棵外其它树构成的森林

先序遍历恰好等价于先序遍历对应的二叉树，后序遍历恰好等价于中序遍历对应的二叉树。

§6.6 Huffman树及其应用

§6.6.1 最优二叉树

1. 概念

- **结点路径长度**：根到该结点所经过的边数
- **树的路径长度**：所有结点的路径长度之和

  (结点数相同时，完全二叉树的路径长度最短)
- **结点的带权路径长度**：

  结点的权值W_i × 结点的路径长度l_i
- **树的带权路径长度**(实际上是加权外部路径长度)

  所有叶子的加权路径长度之和

$$WPL=\sum_{i=1}^{n} w_i l_i \qquad w_i:\text{第i个叶子的权(实数)}$$

$$l_i:\text{第i个叶子的路径长度}$$

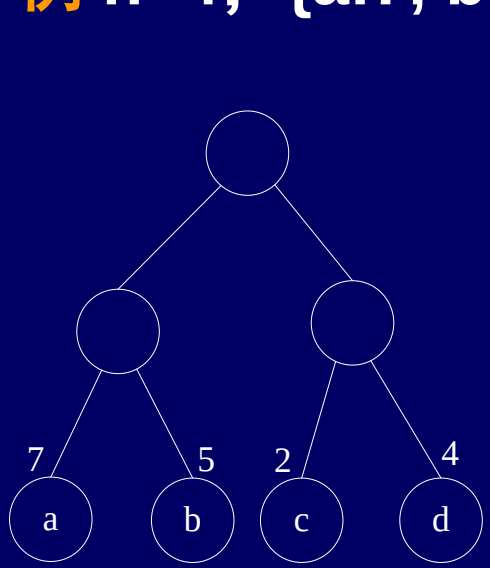
§6.6 Huffman树及其应用

§6.6.1 最优二叉树

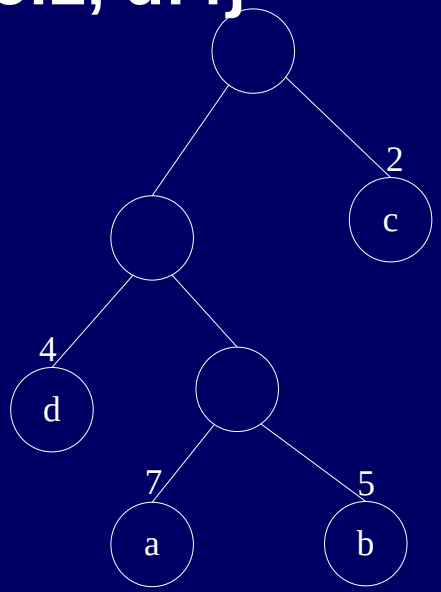
- 最优二叉树（Huffman树）

  在权为$w_1,w_2,\ldots,w_n$的叶子所构成的所有的二叉树，WPL最小的二叉树称为最优二叉树

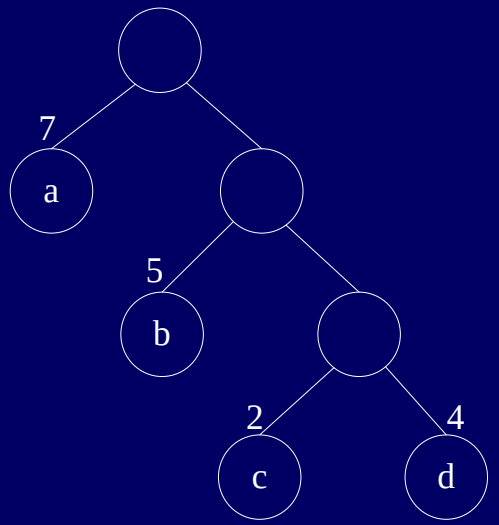
  例 n=4，{a:7, b:5, c:2, d:4}



WPL=36



WPL=46



WPL=35

- 若叶子权值相同，完全二叉树一定是最优二叉树，否则不一定

§6.6 Huffman树及其应用

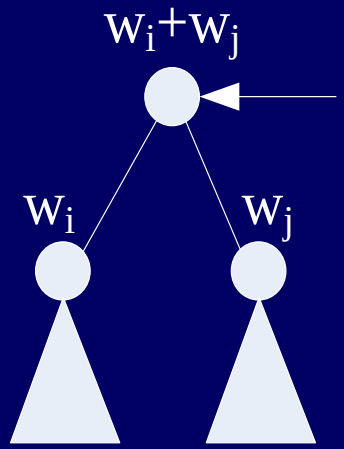
§6.6.1 最优二叉树

2. 构造最优二叉树

显然，权越大应离根越近，Huffman首先提出了构造方法：

1）**初始森林**：根据给定的n个权$\{w_1,w_2,\ldots,w_n\}$构成一个包含n棵二叉树的森林$F=\{T_1,T_2,\ldots,T_n\}$. 其中T_i只有一个根(亦为叶子)结点，其权为w_i;

2）**合并**：在F中选两棵根的权最小的二叉树(若不止两棵，则任选)作为左、右孩子，将其合并为一棵新树，新根权值为这两个结点的权值之和；



Note：合并后F中少了1棵树

3）对新森林F重复2)，直至只剩下一棵树为止。

§6.6 Huffman树及其应用

§6.6.1 最优二叉树

2. 构造最优二叉树

- **算法特点**

① 初始有n棵树，每棵树仅有一个孤立结点

② 合并：每合并一次，少一棵树，故只需合并n-1次

每合并一次产生1新结点，且新结点度为2，故最终的Huffman树有2n-1个结点：

$$2n-1\text{个结点}\begin{cases}\text{n个叶子}\\ \text{n}-1\text{个度为2的结点}\end{cases}\text{无1度结点(严格的2叉树)}$$

实际上任何严格二叉树中，叶子数为n时，总结点数为2n-1。

§6.6 Huffman树及其应用

§6.6.1 最优二叉树

2. 构造最优二叉树

- 存储结构

```
#define    n     100        // 叶子数
#define    m     2*n-1      // 结点总数
typedef struct {
     float weight;                  // 权，不妨设≥0
      int lchild, rchild, parent;   // 指针
 } HTNode;

 typedef HTNode HuffmanTree[m];
```

parent域作用：找双亲结点

为-1时表示根，区分根与非根

- 构造算法

§6.6 Huffman树及其应用

```
void CreateHuffmanTree (HuffmanTree T) { // 构造最优树，根为T[m-1]
    int i, p1, p2;
    InitHuffmanTree(T);  // 初始化, 将2n-1个结点的3个指针置空(-1),权置为0
    InputWeight(T);      // 输入n个叶子(根)的权存于T[0..n-1]中，初
                         //始化森林F0中的根权
    for (i = n; i < m; i++){ // 对F中的树合并n-1次，新根依次放入T[i]中，最终
                         //根为T[m-1]
        SelectMin(T, i-1, &p1, &p2); // 在当前森林T[0..i-1]的所有结点中，选权
        //最小和次小的两个根结点T[p1]和T[p2]作为合并对象，0≤p1,p2≤i-1
        T[p1].parent = T[p2].parent = i;  // 合并产生的新根为T[i]
        T[i].lchild = p1;
        T[i].rchild = p2;
        T[i].weight = T[p1].weight + T[p2].weight
    }
}
```

算法中用到的三个函数略，实例（教材）

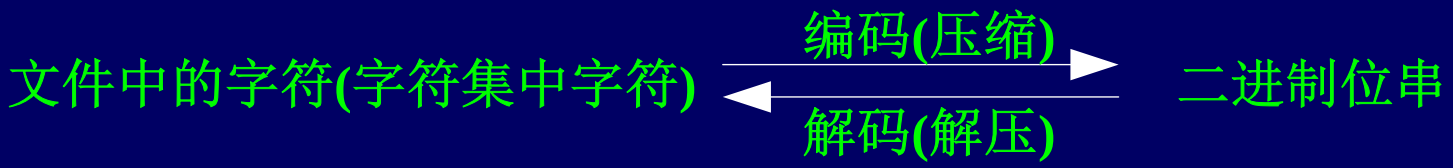
§ 6.6.2 Huffman编码

1. 概念

- **数据压缩**

  可将数据文件压缩20%～90%,压缩效率取决于文件特征

- **编解码**



- **编码方案(对字符集编码)**

  例 C = {a, b, c, d, e, f}, 设数据文件有10万字符， |C|=6

	a	b	c	d	e	f	编码总长
频度(万)	4.5	1.3	1.2	1.0	0.9	0.5	
定长	000	001	010	011	100	101	30万
变长	0	101	100	111	1101	1100	22.4万

节约25%空间

§ 6.6.2 Huffman编码

- **定长编码**：码长 $\lceil \lg |C| \rceil$
- **变长编码**：问题是解码可能有二义性

  例如：设E,T,W编码为00, 01, 0001

  解码时对0001串有两种方式：ET, W

  问题的原因 —— E的编码是W的编码的前缀
- **前缀编码**：要求字符集中，任一字符的编码皆不是其他字符编码的前缀。 显然，定长编码是前缀编码
- **最优的前缀码 (Huffman编码)**：压缩效果最佳的前缀码

① **动态编码**

对给定文件，先统计字符集C = {c_1,c_2,…,c_n} 中各字符出现的**频率**f_i, 据此设计编码，设c_i的码长为l_i，则**编码文件总长度**：

$\sum_{i=1}^{n} f_i l_i$—— 使文件编码总长最短的前缀码是最优前缀码

对同一字符集表示的不同文件，编码方案不同，即根据文件特征动态编码。

特点： 费时，效果最佳

§ 6.6.2 Huffman编码

② 静态编码

无需每次压缩前均统计C_i的频度，而是对定义在相同字符集上的大量文件进行统计，得出每个字符C_i出现的概率p_i，据此编码，则平均码长为：

$\sum_{i=1}^{n} p_i l_i$ —— 平均码长最短的前缀码为最优前缀码

例：上例中a～f 出现的概率为：0.45, 0.13, …, 0.05, 平均码长为2.24，优于定长编码 (平均码长为3)。

特点
- 所有文件使用同一编码，省时
- 对不同的文件，效果不尽相同

2. 编码算法

- 算法思想

利用Huffman树求最优前缀码

step1：用字符c_i作为叶子，p_i(或f_i)做c_i的权，构造Huffman树

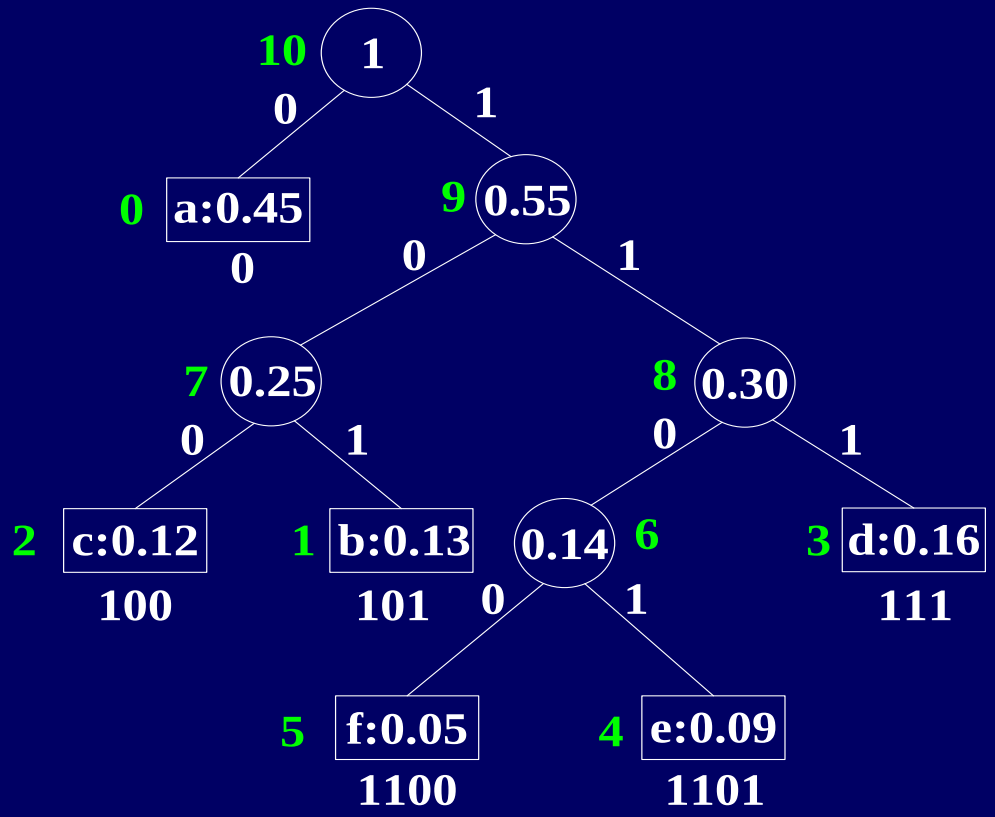
step2：将树的左右分支分别标记0和1，将根到叶子的路径上的标号依次相连，作为该叶子(字符)的编码。

§6.6.2 Huffman编码

- 例子

0 1 2 3 4 5 6 7 8 9 10

T: a:0.45, b:0.13, c:0.12, d:0.16, e:0.09, f:0.05, 0.14, 0.25, 0.3, 0.55, 1.0



（a）哈夫曼编码树

下标	ch	bits
0	a	0
1	b	101
2	c	100
3	d	111
4	e	1101
5	f	1100

（b）哈夫曼编码表

按算法建立的Huffman树唯一

§ 6.6.2 Huffman编码

- 上述编码是最优的前缀吗？

② 最优

∵ 叶子的码长 = 叶子的路径长度li

∴ $\sum_{i=1}^{n} p_i l_i$ 既是平均码长，又是二叉树的WPL

即 Huffman树是WPL最小的二叉树 => 平均码长最小

② 前缀码

∵ 树中任一叶子不可能是其它叶子的祖先

∴ 每叶子编码不可能是其它叶子编码的前缀

3. 算法实现 (对字符集编码，即叶子集编码)

```
typedef struct {
    char ch;              // 放字符
    char bits[n+1];       // 放位串 '\0'结束，码长不会超过n
} CodeNode;               // 存放Huffman编码的结点
typedef CodeNode HuffmanCode[n];
```

§ 6.6.2 Huffman编码

```
void HuffmanCoding (HuffmanTree T, HuffmanCode H) {
    // 根据哈夫曼树T求哈夫曼编码表H
    int c, p, i;        // c和p分别指示树T中孩子和父亲的位置
    char cd[n+1]; // 临时存放编码
    int start;          // 指示编码在cd中的起始位置
    cd[n] = '\0';     // 从后往前放编码
    for (i = 0; i < n; i++){ //依次求叶子T[i]的编码 0 ≤ i ≤ n-1
      H[i].ch = getchar();//读入叶子T[i]对应的字符,若建树时有字符则无需读入
      start = n;
      c = i;                      //从叶子T[i]上溯至根，逆向求编码
      while((p = T[c].parent) >= 0) { // 到根为止
        if (T[p].lchild == c) cd[--start] = '0'; // 若T[c]是T[p]的左子树生成代码0
        else cd[--start] = '1';                  // 否则生成代码1
        c = p;                                   // 继续上溯
       }
       strcpy(H[i].bits, &cd[start]);   // 复制编码位串
    } // endfor
}   //    时间: O(n.h)
```

§6.6.2 Huffman编码

4. 应用 (数据文件的压缩与解压)

① 压缩数据文件 (对文件编码)

```
for (依次从f1中读入字符c) {
    在Huffman编码表H中，找H[i].ch = c
    将c转换为H[i].bits写入压缩文件f2中；
    // 按bits0，1串写入“位”，设f2是二进制文件
}
```

② 加压译码 (对压缩文件解码)

```
for (依次读入f2中的位串) { // 直至文件结束
    从Huffman树根T[m-1]出发
    若当前读入0，走向左孩子，否则走向右孩子；
    若到达叶子T[i]，便译出字符H[i].ch写入还原文件中，然后
    重新从根出发译码
}
```

Note: 实际压缩与解压时，编码的0/1位串，不是字符串，即写入压缩文件中的是“位”

§ 6.7 回溯法与搜索树

1. 回溯法思想

有一类问题，需要找出它的解集合，或要求找出满足某些约束条件下的最优解，最简单的方法是回溯法。

所谓**回溯**就是走回头路，即在一定的约束条件下试探地搜索前进，若前进中受阻，则回头另择道路继续搜索(搜索路线是一棵树)

2. N皇后问题(Gauss 1777-1855 德国数学家)

- 高斯8后问题 (1850提出)，即在8×8国际象棋棋盘上，安放8个皇后，要求彼此互不攻击，有多少个解，这些解的格局如何？

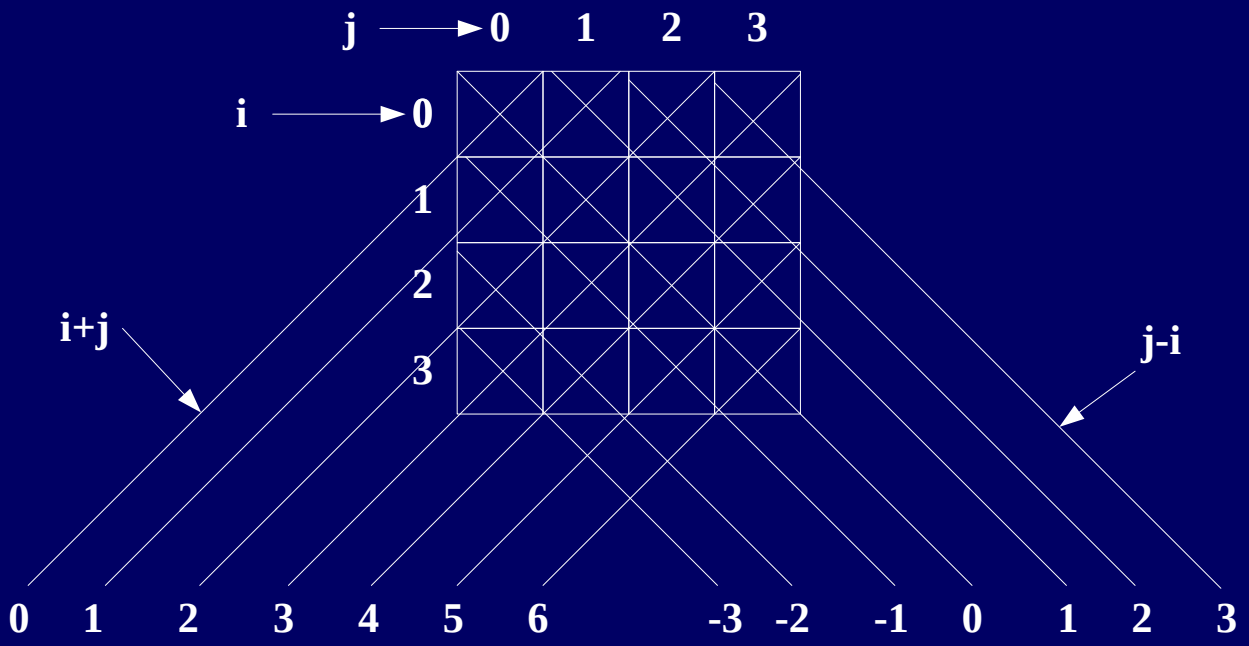
  他本人未解决此问题

  原因：$C_{64}^{8}=\frac{64!}{8!56!}=2^{32}$ 种格局，92个解(包括对称解)

§ 6.7 回溯法与搜索树

- 攻击(约束条件)

  同一行、列，同一对角线的两皇后互相攻击



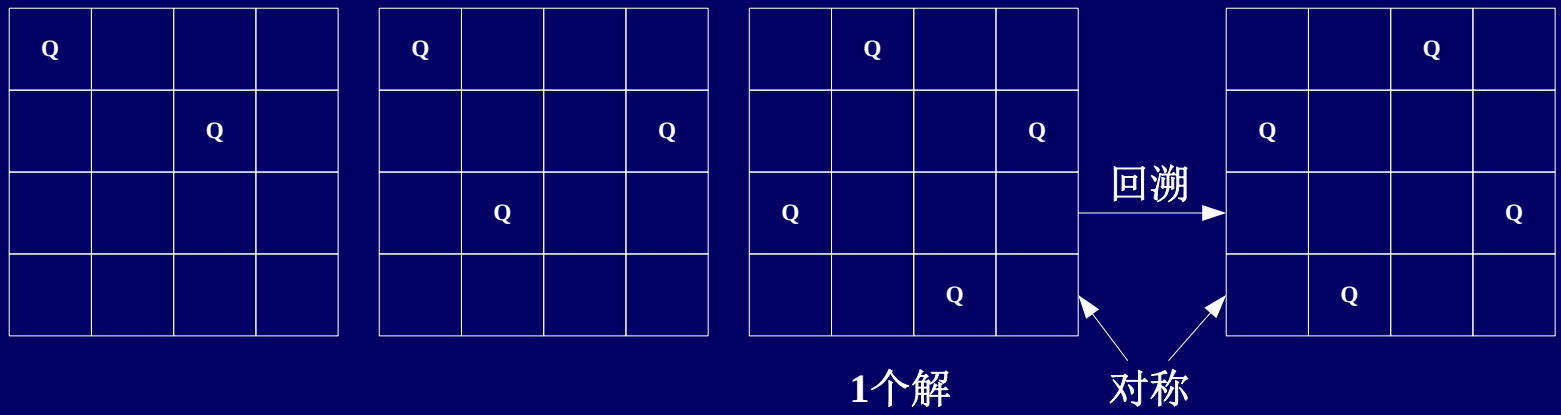
i+j: 135度对角线：同一对角线上元素行列号之和相等(2n-1条)
0～2n-2

j-i: 45度对角线：同一对角线上元素行列号之差相等(2n-1条)
–(n-1),…,0,…,n-1

§ 6.7 回溯法与搜索树

- 回溯法

从第1行开始依次放置皇后，每行从第1列开始试探位置是否安全，安全则放置，若某行所有位置均不安全，则回溯到上一行，重新放置。



将上述求解过程中棋盘状态的每一步变化用树来表示，则可用4叉树表示(如书上Fig6.29),该树反映了状态空间中搜索过程，不满足约束条件的结点不再生长(即被剪枝)。

先序遍历该树

§ 6.7 回溯法与搜索树

- N皇后算法

设try[0..n-1]存放解，下标为行，值为列，即：设try[i]=j，则(i, j)表示棋盘上第i行，第j列存一皇后，逐行放置，每行只有一皇后故可不考虑行冲突,只要考虑列和2条对角线冲突。

```
void Queens(i, col, diag45, diag135){ //i, col, diag45, diag135为值参
  // 全局量try进入此处时，部分解try[0..i]已求出，col,diag45,diag135
  // 是集合，初始调用为Queens(-1, Φ,Φ, Φ)
  if (i == n-1)
      print try[0..n-1]; // 输出一个解
  else // 试求部分解try[0..i+1]，即在(i+1)行上放皇后
      for (j = 0; j < n; j++) // 试探在第(i+1)行上放皇后
          if (j∉col && j-(i+1)∉diag45 && (i+1)+j∉diag135) {
              // （i+1, j）位置安全
              try[i+1] = j;
              Queens(i+1, col∪{j}, diag45∪{j-i-1}, diag135∪{i+j+1});
          } // endif
} // 回溯过程要跟踪执行过程才能发现
```

§ 6.7 回溯法与搜索树

- N皇后算法

上述算法的执行过程就是Fig6.29的状态树(先序遍历)

- 改进

实际算法可设置布尔数组(初始值均为false)来测试安全性

② 放置皇后(i+1, j)

令：col[j] = true, 表示第j列已有皇后

diag45[(n-1)+j-(i+1)]=true,表示该对角线上已有皇后

// 移位n-1

diag135[(i+1)+j]=true,表示该对角线上已有皇后

② 测试安全性

```
if (!col[j] && !diag45[n-i+j-2] && !diag135[i+1+j])
```

Note: 用数组要注意它们不是值参，因此可将其说明为结构，内含数组

§ 6.8 树的计数

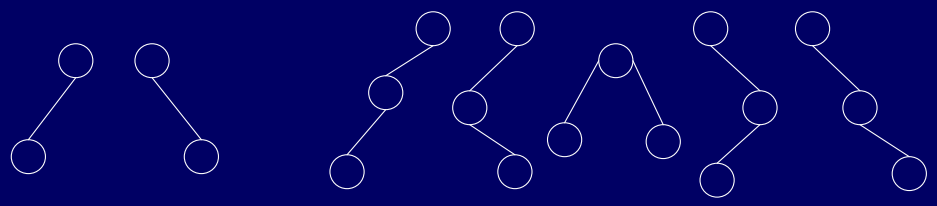
- 问题：一棵具有n个结点的二叉树有多少种不同形态？
- 二叉树相似：

T和T'相似：①二者皆空，否则
②二者不空时，它们的左右子树相似

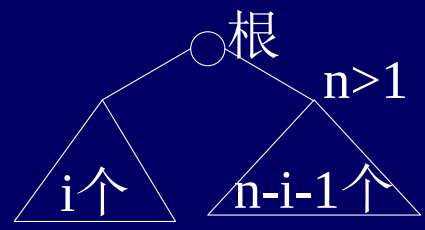
指形态相同，不考虑结点中数据是否相同，否则为树等价

- 二叉树的计数问题：求n个结点互不相似的二叉树数目b_n.

$b_0=b_1=1$ $b_2=2$ $b_3=5$



n>3?



§6.8 树的计数

一般情况：$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} \quad n \geq 1 \end{cases}$

利用生成函数可求出此递推公式的解为（教材）：$b_n = \frac{C_{2n}^n}{n+1}$

- **树的计数问题**

  n个结点的树的形态数目 $t_n = b_{n-1}$，∵树转换为二叉树后，根无右子树

- **遍历序列能否唯一确定一棵二叉树？**

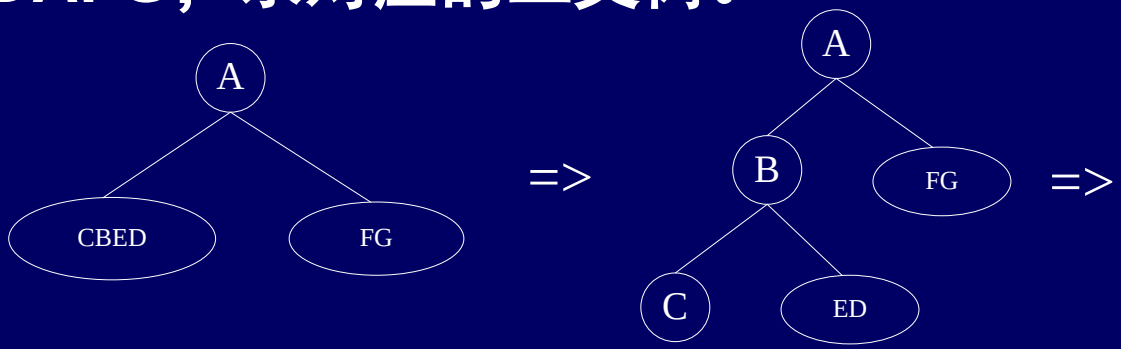
  二叉树确定后，其三种遍历序列唯一确定

  反之，能唯一确定一棵二叉树吗？

§6.8 树的计数

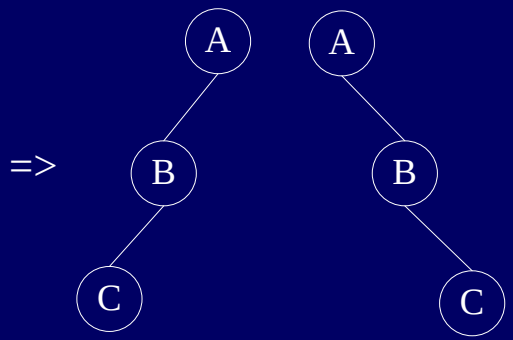
① 由一棵二叉树的前序序列(或后序序列)+中序序列可唯一确定该二叉树

例：已知某二叉树的前序序列：ABCDEFG，中序序列：CBEDAFG，求对应的二叉树。



② 由前序序列+后序序列不能唯一确定一棵二叉树

例：前：ABC
后：CBA



③ 由一个遍历序列更不能唯一确定一棵二叉树。

作业

- 已知一棵二叉树的先序遍历序列和中序遍历序列分别为ABDGHCEFI和GDHBAECIF，请画出这棵二叉树对应的中序线索树，然后给出该树的后序遍历序列。
- 假设用于通信的电文是由字符集{a, b, c, d, e, f, g, h}中的字符构成， 这8个字符在电文中出现的概率分别为{0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10} 。
  ① 请画出对应的huffman树（按左子树根结点的权小于等于右子树根结点的权的次序构造）。
  ② 求出每个字符的huffman编码。

Ch.7 图

- 图是一种复杂的非线性结构
- 应用：AI、工程、数学、生物、计算机
- 结点间的逻辑关系：任两个结点都可能相关

§7.1 概念

- **Def：** 图由两集合组成G=(V, E)

  V(G):顶点集——顶点的有穷非空集

  E(G):边集——V中顶点偶对的有穷集

- **无向图：** 边由顶点的无序对构成。

  (V_i, V_j) 和 (V_j, V_i) 表示同一条边，称为无向边 

- **有向图：** 边由顶点的有序对构成。

  $\langle V_i, V_j \rangle$ 和 $\langle V_j, V_i \rangle$ 表示不同的有向边

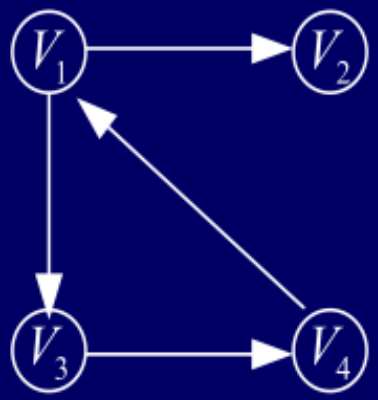
  

弧尾 V_i ——起点

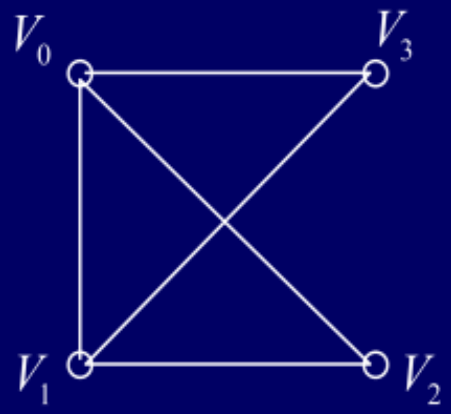
  弧头 V_j ——终点

§7.1 概念

- 例子：



G_1



G_2

$$G_1=(V_1,E_1) \qquad V_1=V(G_1)=\{v_1,v_2,v_3,v_4\}$$

$$E_1=E(G_1)=\{<v_1,v_2>,<v_1,v_3>,<v_3,v_4>,<v_4,v_1>\}$$

$$G_2=(V_2,E_2) \qquad V_2=V(G_2)=\{v_0,v_1,v_2,v_3\}$$

$$E_2=E(G_2)=\{(v_0,v_1),(v_0,v_2),(v_0,v_3),(v_1,v_2),(v_1,v_3)\}$$

- 约定：只讨论简单图

① 不允许有反身边：即 (V_i,V_j) 或 $\langle V_i,V_j\rangle \in E$ 则 $V_i \neq V_j$

② 不允许平行边：$E(G)$ 中不允许有重复元素

§7.1 概念

- **顶点和边的关系：** 设 $|V|=n,|E|=e$

① **无向图：** $0 \le e \le n(n-1)/2$

当 $e=0$ 时，则为**零图** $(E=\Phi)$

当 $e=n(n-1)/2$，则称之为（无向）**完全图**

完全图中任一顶点到其他顶点均有边

② **有向图：**

$0 \le e \le n(n-1)$

当 $e=n(n-1)$，则称之为**有向完全图**

§7.1 概念

③ **稀疏图、稠密图.**

边少、边多. $e < n\lg n$?

- **邻接与关联（依附）**

若$e_1 = (v_i, v_j) \in E$，则 v_i 和 v_j 互为邻接点

若$e_1 = \langle v_i, v_j \rangle \in E$，则 v_i 邻接到 v_j，v_j 邻接自 v_i

} v_i和v_j相邻接

边 e_1 和 e_2 关联（依附）于顶点 v_i 和 v_j

不好定义前驱和后继

§7.1 概念

- **顶点的度**

无向图：关联于顶点的边数 $D(v)$。例 $\forall v \in V(G_3),\quad D(v)=4$

有向图：出度—以 v 为起点的边数 $OD(v)$

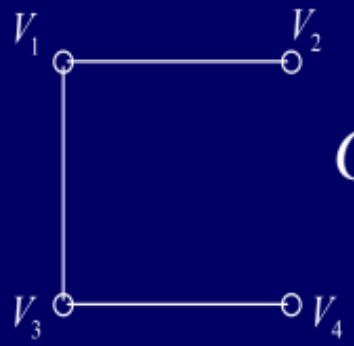
入度—以 v 为终点的边数 $ID(v)$

$$D(v)=ID(v)+OD(v)$$

$$e=\frac{1}{2}\sum_{i=1}^{n}D(v_i)$$ ——对有向或无向图均成立

- **子图**

设 $G=(V,E)$ 是图，$V'\subseteq V, E'\subseteq E$，且 E' 中边所关联的顶点均在 V' 中，则 $G'=(V',E')$ 亦为图，称之为G的子图。



$G'=(\{v_1,v_2\},\{(v_1,v_2),(v_3,v_4)\})$不是G的子图，因为它不是图

§7.1 概念

- **路径**

  若存在一顶点序列 $v_p, v_{i1}, v_{i2} \cdots v_{im}, v_q$ 使

  $(v_p, v_{i1}), (v_{i1}, v_{i2}), \cdots, (v_{im}, v_q) \in E(G)$

  则称从 v_p 到 v_q 存在一条**路径**。

  所经过的边数称为路径**长度**。

  有向路径类似定义。

- **简单路径：**

  除起点和终点外，其余顶点均不相同的路径.

- **简单回路（环）：**

  起点和终点相同的简单路径。

§7.1 概念

- **有根图：**

  在有向图G中，若存在$v \in V$，从v到其他顶点均有路径可达，则v称为根，G为有根图。

- **连通、连通图、连通分量**

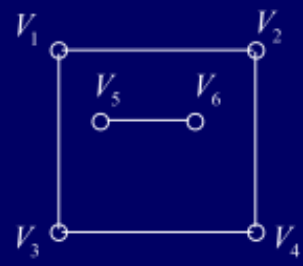
  设G为无向图

  ① G中，若v_i和v_j有路径，则称v_i和v_j连通。

  ② 若$\forall v_i, v_j \in V(G)$，$v_i$和$v_j$均连通，则G为连通图。

  ③ 无向图中极大连通子图称为连通分量。

  连通图只有一个连通分量，即自身。



§7.1 概念

- **强连通图**

  设G是有向图，$\forall v_i, v_j \in V(G)$，若 v_i 到 v_j 及 v_j 到 v_i 都有路径，则是强连通图

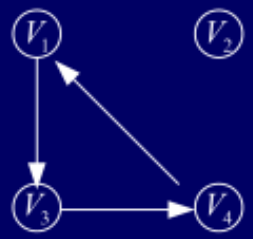
  n个顶点的强连通图至少有几条边？

- **强连通分量：**

  有向图的极大强连通子图，称为强连通分量。

  强连通图只有一个强连通分量。

  例：G_1不是强连通图，它有两个强连通分量。



- **加权图：**

  顶点、边上带权。

§7.2 图的存储结构

没有前驱和后继的关系，任两结点均可能有“邻接”关系

- 无向图中，邻接点互为对称，分不清谁是前驱和后继。
- 有向图中，边的起点为前驱，终点为后继，但有向环又如何表达？

设G中，$V(G)=\{v_0,\cdots,v_{n-1}\}$

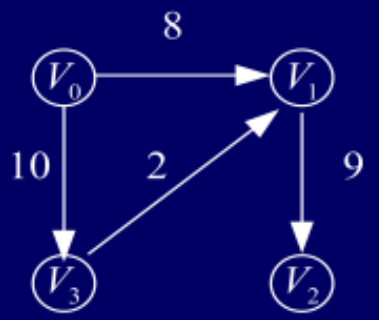
多种表示法，重点介绍常用的两种。

§7.2.1 邻接矩阵表示法

- **邻接矩阵**：表示顶点（数据元素）相邻关系的矩阵。

  若顶点信息无关紧要，则用二维数组 $A_{n\times n}$ 表示，$n=|V(G)|$

$$A[i,j]=\begin{cases}1 & \text{若}(v_i,v_j),\text{或}<v_i,v_j>\in E(G)\\ 0 & \text{否则}\end{cases}$$



对于加权图：

$$A[i,j]=\begin{cases}w_{ij} & \text{若}(v_i,v_j),\text{或}<v_i,v_j>\in E(G)\\ 0\text{或}\infty & \text{否则}\end{cases}$$

无向图的邻接矩阵是对称的

$$\begin{array}{c} \\ 0\\ 1\\ 2\\ 3\end{array}\begin{array}{c}\begin{array}{cccc}0 & 1 & 2 & 3\end{array}\\ \begin{bmatrix}\infty & 8 & \infty & 10\\ \infty & \infty & 9 & \infty\\ \infty & \infty & \infty & \infty\\ \infty & 2 & \infty & \infty\end{bmatrix}\end{array}$$

- **邻接矩阵表示法**：

  若顶点信息重要，则应将其组织成一个顺序表：

  边表（邻接矩阵）

  顶点表

  → 邻接矩阵表示法

§7.2.1 邻接矩阵表示法

- 类型说明

```
#define MaxVertexNum 100   //最大顶点数
typedef int EdgeType;  //边类型
typedef char VertexType; //顶点类型
typedef struct{
    VertexType vexs[MaxVertexNum];  //顶点表
    EdgeType edges[MaxVertexNum][MaxVertexNum];
                                        //边表，邻接矩阵
    int n, e; //顶点数，边数
}MGraph;
```

§7.2.1 邻接矩阵表示法

- 建立无向图的邻接矩阵表示

step1：输入顶点数、边数 $O(1)$

step2：输入顶点表 $O(n)$

step3：初始化邻接矩阵 $O(n^2)$

step4：输入边（及权值） $O(e)$

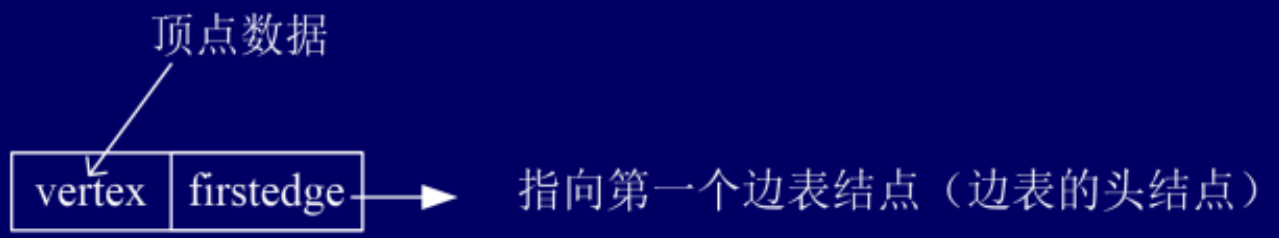
$O(n^2)$

§7.2.2 邻接表

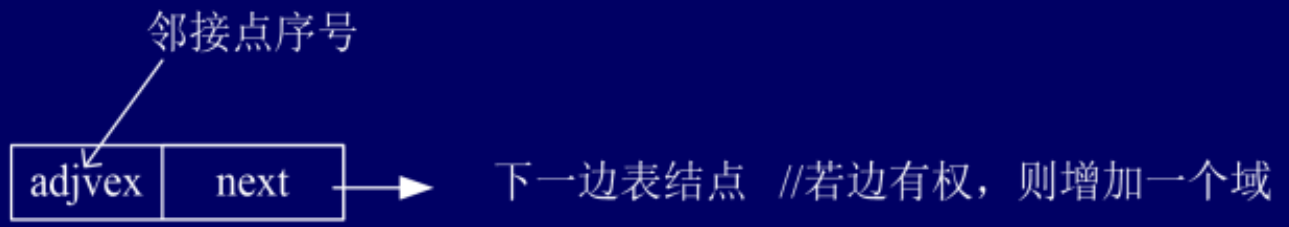
- **链式存储结构**

为每个顶点的邻接关系建立一个单链表，将头结点放在一个数组中，形成一个顶点表和一个边表。

顶点表结点：



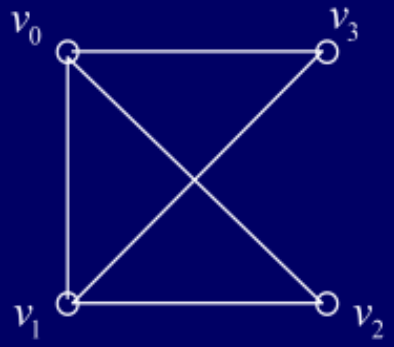
边表结点：

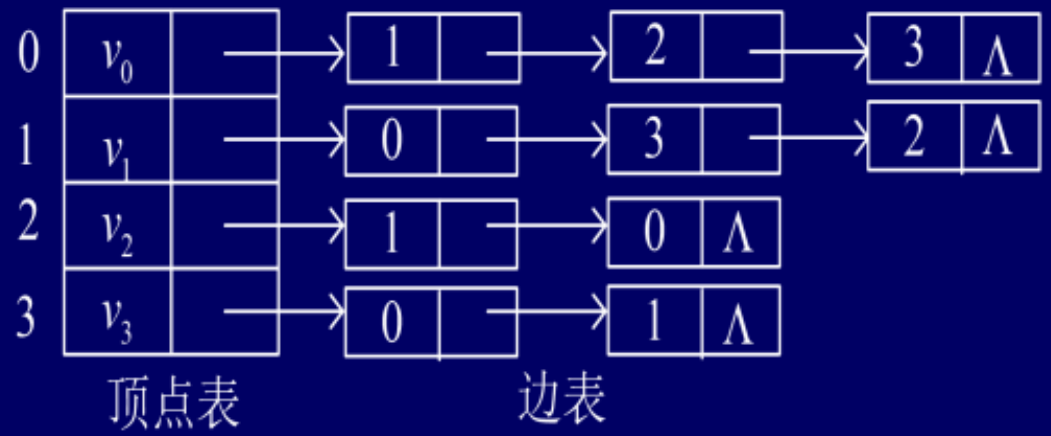


边表示：(v_i, v_j)，因为 v_i 和 v_j 分别存储在顶点表的*i*th和*j*th分量中，故只需在 v_i 的边表中存放序号j即可。

§7.2.2 邻接表

- 例：





- 说明

```
typedef struct node{//边表结点
    int adjvex;  //邻接点序号
    struct node * next; //指向下一个边表结点
    //若有权，则增加一域
}EdgeNode;
```

§7.2.2 邻接表

```
typedef struct vnode{//顶点表结点
    VertexType vertex;  //顶点数据域
    EdgeNode * firestedge; //边表头指针
}VertexNode, AdjList[MaxVertexNum];  //邻接表类型

typedef struct {
    AdjList adjlist; //邻接表
    int n, e; //顶点数和边数
}ALGraph;
```

§ 7.2.2 邻接表

- **无向图的邻接表**

$(v_i, v_j) \in E$，则在 v_i 的邻接表中有一 $adjtex = j$ 的边表结点。

在 v_j 的邻接表中有一 $adjtex = i$ 的边表结点。

每条边表示了两次，所以一共有2e个边表结点。

空间复杂度为 $O(n+e)$，邻接矩阵为 $O(n^2)$。

稀疏图邻接表节省空间，稠密图邻接矩阵为宜。

§7.2.2 邻接表

- **有向图的邻接表**

  $<v_i,v_j>\in E$，在 v_i 的邻接表中有一个 $adjtex=j$ 的边表结点。

  每边表示了1次，所以共有e个边表结点。

  此时的边表称为出边表。

  出边表中求出度易，求入度难。

  逆邻接表：$<v_i,v_j>\in E$，在 v_j 的邻接表中有一个 $adjtex=i$ 的边表结点。

  此时的边表称为入边表。

  入边表中求入度易，求出度难。

§7.2.2 邻接表

- **建立邻接表**
  - ① 时间：$O(n+e)$，邻接矩阵 $O(n^2)$
  - ② 唯一性：不唯一，不同输入次序，结果不同。但邻接矩阵唯一。
- **运算**
  - ① 判 (v_i, v_j) 是否为边？　邻接矩阵 $O(1)$，邻接表 $O(n)$
  - ② 求顶点度数：　二者差不多 $O(n)$
  - ③ 检测边数：邻接矩阵 $O(n^2)$，邻接表 $O(n+e)$

§7.3 图的遍历

- 是图上运算的基础
- 没有开始结点，如何访问？任两点可能相邻，故访问某点后可能顺回路又回到该点，为避免重复访问，须标记已访问过的顶点
- Visited[0…n-1]布尔数组，初值为false
- 常用的方法有两种

§7.3.1 深度优先遍历（dfs遍历）

类似于树的前序遍历

- **基本思想**

设G初态是所有顶点均未曾访问，∀v∈V(G)为初始出发点（源点），则深度优先遍历可定义为：

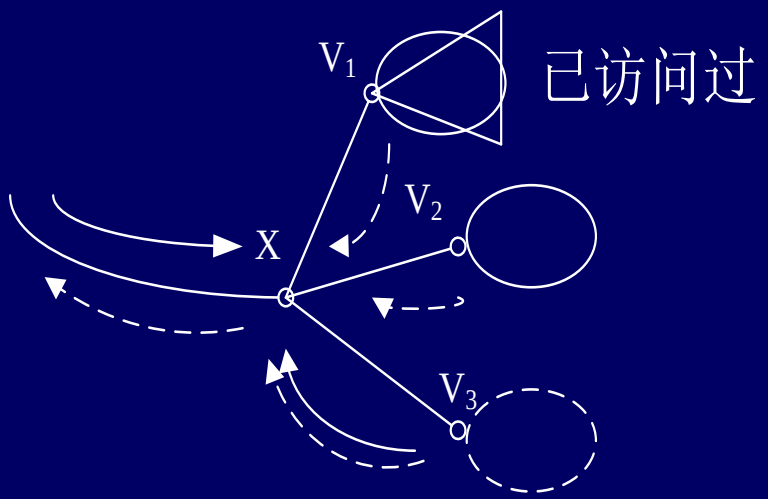
首先访问出发点v，将其标记为已访问；

然后依次从v出发搜索v的每个邻接点w，若w未曾访问过，则以w为出发点继续深度优先遍历，直至图中所有和v有路径相通的顶点均已被访问为止；

若此时图中仍有未访问过的顶点，则另选一尚未访问过的顶点作为新源点重复上述过程，直至图中所有顶点均已访问为止。

§7.3.1 深度优先遍历（dfs遍历）

- **特点**：遍历定义是递归的，特点是尽可能先对纵深方向搜索，故称为深度优先搜索（dfs），搜索过程：



回溯“- - - - →”

碰壁回头（典型的回溯法）

§7.3.1 深度优先遍历（dfs遍历）

设x是当前访问顶点：

① 若v_1，v_2，v_3均未被访问过，则以v_1为出发点向纵深搜索，不能前进后回溯到x，继续从v_2，v_3出发，当v_2，v_3为出发点搜索完成后回溯至x，因为x的所有邻接点均已访问过，故继续回溯至x之前被访问的顶点；

② 若v_1，v_2，v_3均访问过，表示一定是从x之前被访问的顶点出发的搜索曾到达过v_1，v_2，v_3，故访问x之后返回（回溯）至x之前的结点。

§7.3.1 深度优先遍历（dfs遍历）

- 算法实现

```
typedef enum {FALSE, TRUE} Boolean;
Boolean Visited[MaxVertexNum]; //全局量
void DFSTraverse(ALGraph *G) { //以邻接表表示图
    for (i=0; i<G->n; i++)
        Visited[i] = FALSE; //初始化
    for (i=0; i<G->n; i++)
        if (! Visited[i]) //vi未访问过
            DFS(G,i); //以vi为源点开始dfs搜索
                      //图连通仅有1次外部调用
}
```

§7.3.1 深度优先遍历（dfs遍历）

```
void DFS (ALGraph *G, int i) { //以v_i为出发点对G进行dfs
    EdgeNode *p;
    printf ("%c", G->adjlist[i].vertex); //访问顶点v_i
    Visited[i]=TRUE; //标记v_i已访问过
    p=G->adjlist[i].firstedge; //取v_i边表的头指针
    while (p) { //依次搜索v_i的邻接点v_j
        if (!Visited[p->adjvex]) //若v_j未访问过，j=p->adjvex
            DFS(G,p->adjvex); //以v_j为出发点向纵深搜索
        p=p->next; //若v_j已访问过，或从v_j出发的dfs完成，则
    //找v_i的下一邻接点
    }
}
```

以邻接矩阵为存储结构自己写出DFS

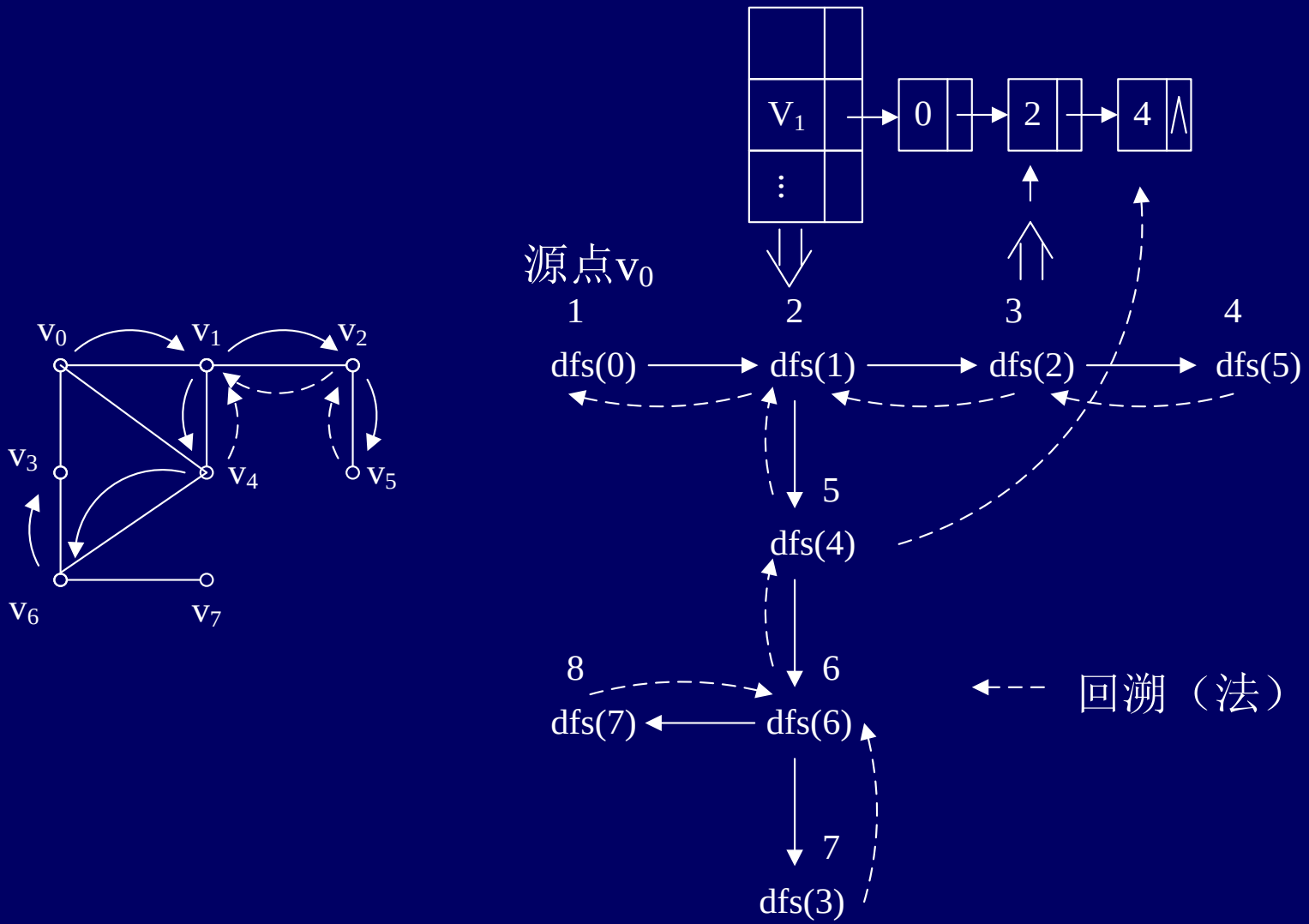
§7.3.1 深度优先遍历（dfs遍历）

- 深度优先遍历序列（DFS序列）

  遍历过程的顶点访问序列

  G的DFS序列不唯一，但初始源点给定，存储结构给定时所得序列唯一

§7.3.1 深度优先遍历（dfs遍历）



dfs序列：$v_0,v_1,v_2,v_5,v_4,v_6,v_3,v_7$（假设邻接表的边表递增有序）

§7.3.1 深度优先遍历（dfs遍历）

- 时间

① 对G中每个顶点恰做一次访问（外部+内部调用dfs），∴共做n次访问

②当访问顶点v_i时，时间主要花费在搜索v_i的邻接点上

合计：邻接表表示：O(n+e)，各个边表结点均搜索到

邻接矩阵：O(n^2)，各个元素均检索到

§7.3.2 广度优先遍历

类似于树的层次遍历

- **基本思想**

①选一顶点v为源点访问之

②依次访问v的所有邻接点$w_1,w_2\cdots w_t$

③然后依次访问$w_i(1\le i\le t)$的所有未曾访问过的邻接点

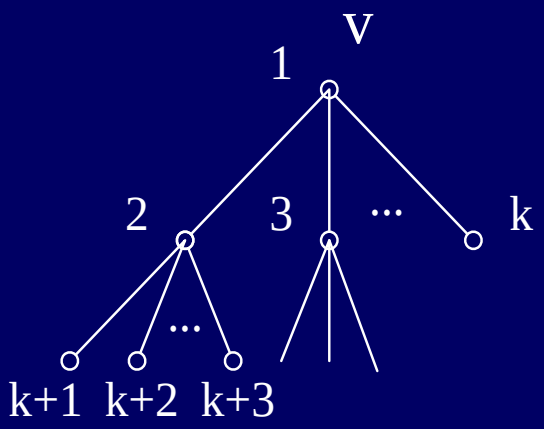
依次类推，直至图中所有和源v有路径连通的顶点均已访问过为止

此时，若G是连通图，则遍历完成；否则选G的另一未访问过的顶点做新源点继续上述搜索过程，直至G中所有顶点均已访问过为止

§7.3.2 广度优先遍历

- 特点

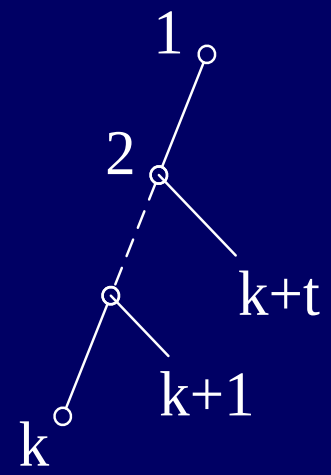
尽可能先对横向搜索，故称之为广度优先搜索（BFS）



象波 Bfs

非递归，用队列作为中间数据结构

先访问的顶点其邻接点亦被先访问（FIFO）



Dfs

递归、回溯，用栈保存访问过的顶点

后访问的顶点其邻接点被先访问（LIFO）

§7.3.2 广度优先遍历

设x和y是两个相继访问的顶点，其邻接点分别记为$x_1,\cdots,x_s$和$y_1,\cdots,y_t$

∵x在y之前访问，故$x_1,\cdots,x_s$中未访问顶点亦先于$y_1,\cdots,y_t$中未访问被访问到

∴可用FIFO队列

①保存已访问过的顶点 ➡顶点入队时对其访问

②保存尚未访问过的顶点 ➡顶点出队时对其访问

第一种方法较好，下面用此方法实现算法

§7.3.2 广度优先遍历

- 算法

遍历算法类似于DFSTraverse

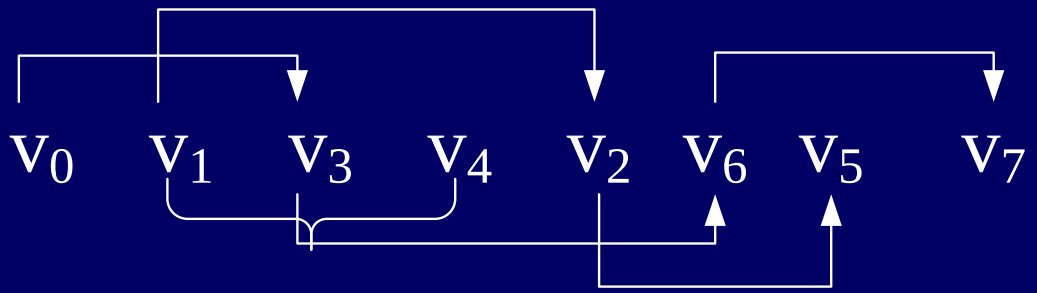
```
void BFS(ALGraph *G, int k) { //以vk为源
    InitQueue(&Q); //队列Q初始化
    printf("%c", G->adjlist[k].vertex); //访问源vk
    Visited[k]=TRUE;
    EnQueue(&Q, k); //相当于vk入队
```

§7.3.2 广度优先遍历

```
while ( !QueueEmpty(&Q) ) {
        i=DeQueue(&Q); //vi出队
        p=G->adjlist[i].firstedge; //取vi边表头指针
        while(p) { //依次搜索vi的邻接点vj
            if(!Visited[p->adjvex]) //vj未访问过，设p->adjvex=j
                    printf("%c",G->adjlist[p->adjvex].vertex); //访问vj
                    Visited[p->adjvex]=TRUE;
                    EnQueue(&Q, p->adjvex); //vj入队
            }
            p=p->next; //在下一边表结点中找vi下一个邻接点
        }
    }
}
```

§7.3.2 广度优先遍历

- **BFS队列**



- **时间**

  每个顶点入队一次，每个边表结点搜索一次

  时间与dfs相同

§7.4 图的连通性问题

§7.4.1 无向图的连通分量和生成树

1.求连通分量

每外部调用一次DFS或BFS，可求一连通分量的顶点集

2.生成树和生成森林

- **生成树**

连通图G的极小连通子图，但包含G的所有顶点（支撑树），不唯一

n个顶点的连通图的生成树一定有n-1条边

§7.4.1 无向图的连通分量和生成树

- **生成森林**：各连通分量的生成树集合
- **求生成树和生成森林**（使用DFS和BFS）

①设G是无向图，$\forall v \in V(G)$做出发点

若图连通，则做一次DFS或BFS，必将G中n个顶点都访问到，且只做一次访问

两种搜索方法中，从v_i搜索到v_j时，须经过边(v_i,v_j)，故只需添加输出边的操作即可：

§7.4.1 无向图的连通分量和生成树

在dfs和bfs中，均在

```
while(p) {
    if( !Visited[p->adjvex] ) {
        加入打印：（i，p->adjvex）
        //dfs须在递归前加入
    }
```

若G连通则求出的生成树，否则为生成森林

②若G为有向图，仅当源点为有根图的根时，才能求得生成树，否则为生成森林

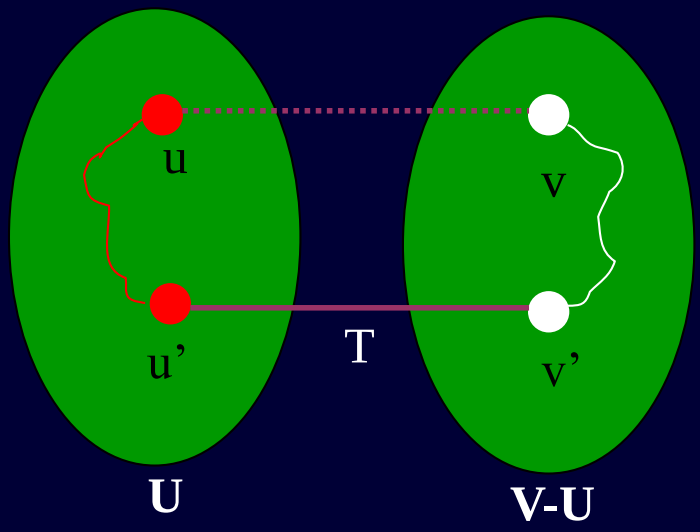
§7.4.2 最小生成树（Minimum Spanning Tree）

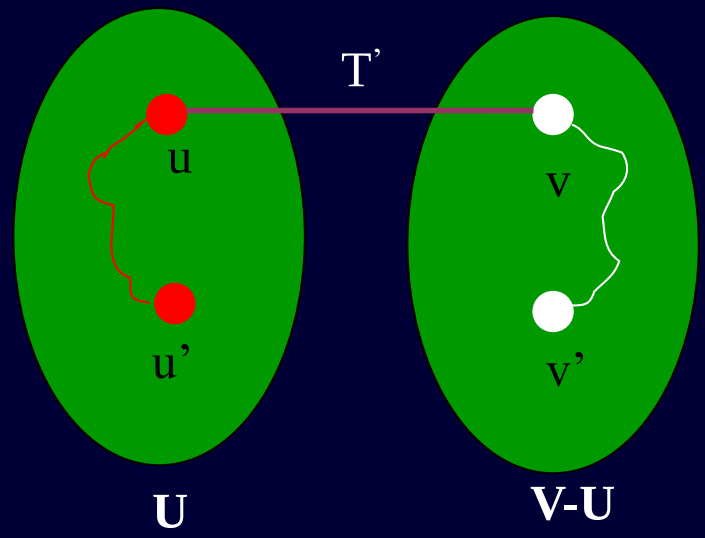
- 设G是连通图，G的生成树不唯一
- MST：权最小的生成树，树的权是各边上的权值之和
- 应用
  - n个城市之间的通信网，可构建n(n-1)/2条线路
  - n个城市连通至少要n-1条线路，G的生成树是1个可行的方案
  - 最小生成树是最经济的可行方案

MST性质—大多数算法都利用了此性质

设G＝(V，E)是一连通图，U是V的真子集，若（u，v）是所有连接U和V-U的边中权最小的边（轻边），则一定存在G的一棵最小生成树包括此边。

Pf：设G的任何一棵最小生成树均不包括(u，v)；

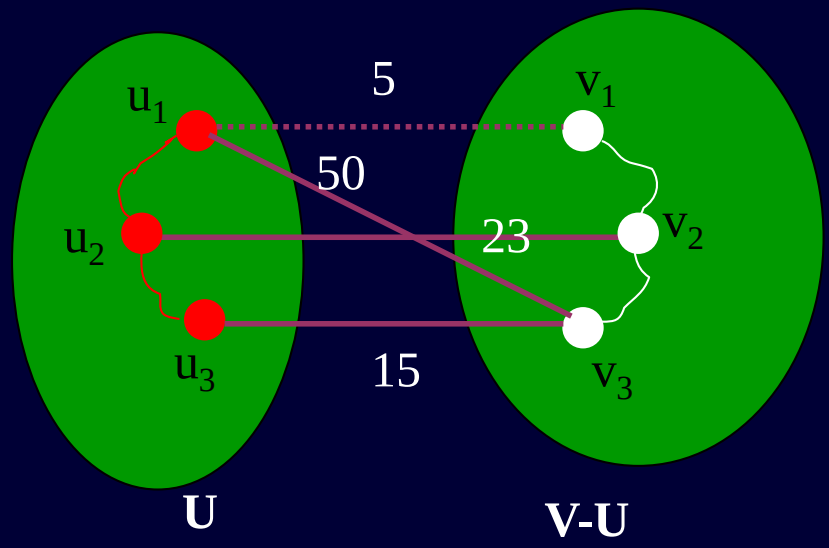




构造MST：

就是找轻边扩充到当前生成的树T＝（U，TE）中

- U－红点集、红边集，构成T
- V-U－白点集、白边集
- 紫边集－连接红点和白点的边
- 轻边－权最轻的紫边，或最短紫边(若权为长度)：（u_1，v_1）



1、Prim算法

- 特点
  - 当前形成的集合T＝（U，TE）始终是一棵树
  - T中的顶点和边均为红色
- 基本思想（贪心法）
  - 设V(G)={0,1,…,n-1}
  - 算法的每一步均是在连接红、白点集的紫边中选一轻边扩充到T（贪心），T从任意一点r开始（r为根），直至U＝V为止。MST性质保证了贪心选择策略的正确性。

1、Prim算法

- 如何找轻边？
  - ❖可能的紫边集

    设红点集|U|=k，白点集|V-U|=n-k，则可能的紫边数为：k(n-k)。

    在此紫边集中选择轻边效率太低。

  - ❖构造候选轻边集

    构造较小的紫边集，但保证轻边在其中。

    因为，∀v∈白点集，从v到各红点的紫边中，只有最短的那一条才可能是轻边，所以只须保留所有n－k个白点所关联的最短紫边作为轻边候选集即可。

    显然，轻边是该候选集中权最轻的边。

    可以针对红点集来构造候选轻边集吗？

1、Prim算法

- 如何维护候选轻边集？

当把轻边（u，v）扩充至T中时，

v由白点变为红点，紫边（u，v）变为红边；

∵对每个剩余白点j，边（v，j）由白变紫，此新紫边的长度可能小于白点j原来所关联的最短紫边。

∴须调整候选轻边集，用更短的新紫边（v，j）取代原来关联于j的最短紫边。

1、Prim算法

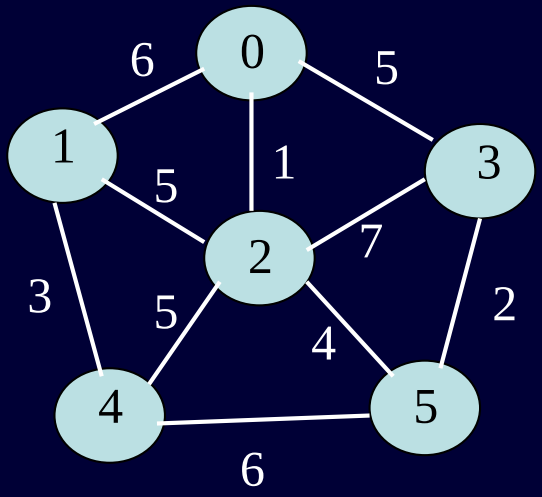
算法梗概

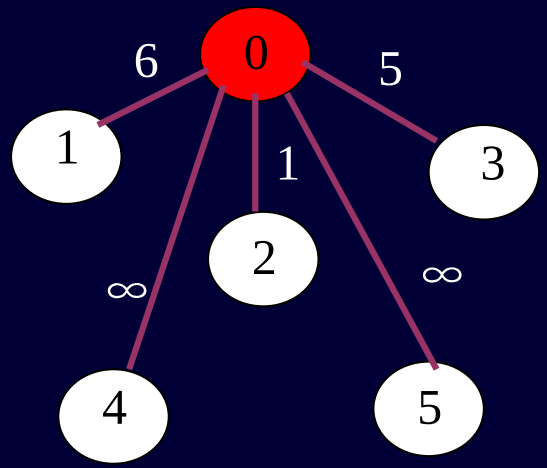
```
PrimMST(G,T,r) { //求以r为根的MST
    InitCandidateSet(…);
     //初始化，置初始的候选轻边集，置T＝（{r }, ϕ ）
    for (k=0; k<n-1; k++) { //求T的n-1条树边
        (u,v)=SelectLightEdge(…); //选轻边，可能不唯一
        TE＝TE∪{(u, v)};
        //将(u,v)涂红加入树中，白点v加入红点集
        ModifyCandidateSet(…);
       //根据新红点v调整候选轻边集
    }
}
```

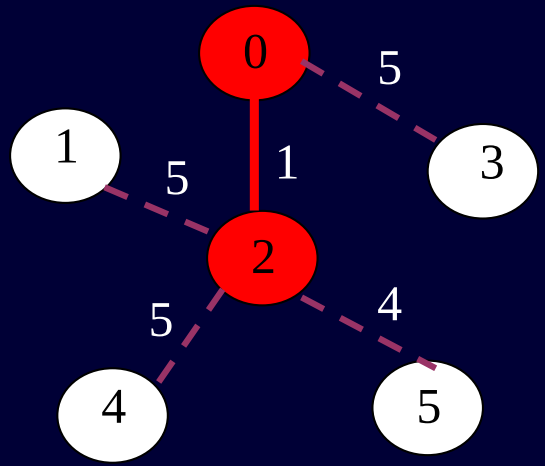
算法终止时U＝V，T=(V,TE)

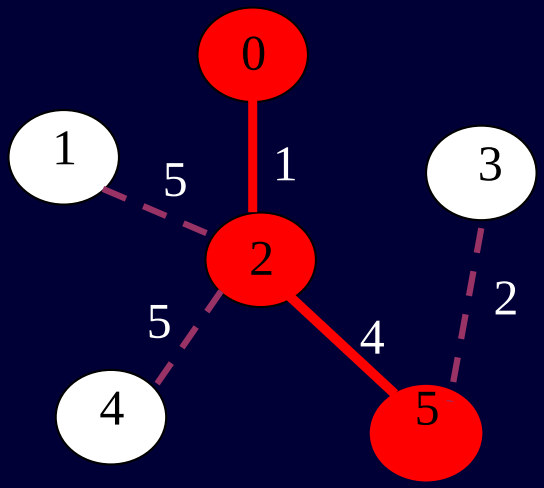
1、Prim算法

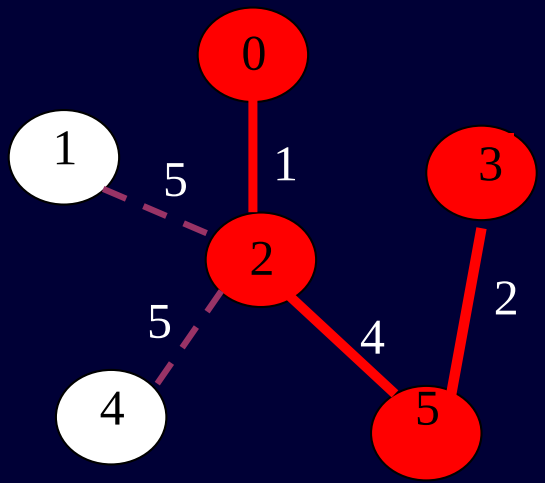
实例

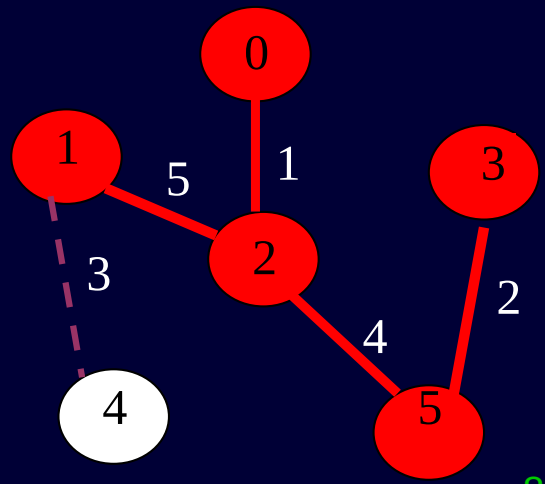

0
1
2
3
4
5
6
5
5
1
7
3
5
2
4
6


0
1
2
3
4
5
6
5
1
∞
∞


0
1
2
3
4
5
5
5
1
5
4


0
1
2
3
4
5
5
1
5
4
2


0
1
2
3
4
5
5
1
5
4
2


0
1
2
3
4
5
5
1
3
4
2

1、Prim算法

存储结构

```
# define Infinity INT_MAX //表示最大整数
# define n 100
typedef int AdjMatrix[n][n]; //邻接矩阵
typedef struct { //树边
        int  fromvex, tovex; //起点、终点
        int  len; //边长度，权值
} MST[n-1];
```

设邻接矩阵初值：不存在的边其权值为Infinity

1、Prim算法

算法求精－初始化

将根r涂红加入红点集U，TE＝φ。

对每个白点i (0≤i ≤n-1, i≠r)，i所关联的最短紫边（r,i）的长度为G[r][i]，这n-1条最短紫边构成了初始的候选轻边集。

因为树边为空，故将T[0..n-2]全部用来存放候选轻边集。

```
void InitCandidateSet (AdjMatrix G, MST T, int r) {
   int i, k=0;
   for (i=0; i<n; i++) //依次形成白点i初始的最短紫边存放在T[k]中
       if  ( i != r ) {
            T[k].fromvex=r; //紫边起点为红点
            T[k].tovex=i; //紫边终点为白点
            T[k++].len=G[r][i]; //紫边长度，权值
   }
} ;
```

1、Prim算法

算法求精一选轻边

在当前候选轻边集T[k..n-2]中，选长度最短的紫边。(Note:T[0..k-1]是红边集，T也是树，为什么针对白点构造候选集更好？)

```
void SelectLightSet ( MST T, int k) {
   int i, minpos, min=Infinity;
   for (i=k; i<n-1; i++) //遍历候选集找轻边
         if  ( T[i].len < min ) {
              min=T[i].len; //紫边起点为红点
              minpos=i; //记录当前最短紫边的位置
         }
   if (min==Infinity )  error( “G不连通”）;
   return minpos; //轻边为T[minpos]
} ;
```

1、Prim算法

算法求精—调整候选轻边集

设轻边（u,v）涂红后加入到树边中，T[k..n-2]是待调整的候选轻边集，则须根据新红点v调整T[k..n-2]。

```
void ModifyCandidateSet ( AdjMatrix G, MST T, int k, int v) {
    int i, d; //v是新红点
    for (i=k; i<n-1; i++) { //遍历候选集
        d=G[v][T[i].tovex]; //T[i]的终点是白点，d是新紫边的长度
        if  ( d<T[i].len ) {//d小于白点T[i].tovex原关联最短紫边长度
            T[i].len=d;
            T[i].fromvex=v; //新紫边取代T[i].tovex原来的最短紫边
        }
    }
};
```

1、Prim算法

算法求精一最终算法

```
void PrimMST(AdjMatrix G, MST T, int r) {
    int k,m,v;
    InitCandidateSet(G, T, r);//初始候选集T[0..n-2]
    for (k=0; k<n-1; k++) { //求n-1条树边中的k^th条
        m＝SelectLightEdge(T,k);
        // 在T[k..n-2]选轻边T[m]
        T[m]↔T[k];//轻边和紫边T[k]交换，将其扩充至树中
        v＝T[k].tovex; //交换后红边集为T[0..k],v是新红点
        ModifyCandidateSet(G,T,k+1,v);
        //T[k+1..n-2]是新候选集，根据新红点v调整候选轻边集
    }
}
```

1、Prim算法

时间分析

初始化：O(n)

在k循环中：

Select中循环次数为n-k-1 // 在T[k..n-2]选轻边T[m]

Modify中循环次数为n-k-2

//T[k+1..n-2]是新候选集，根据新红点v调整候选轻边集

因此：

时间为$O(n^2)$，与边无关，适合稠密图。

2、Kruskal算法

- **特点：**当前形成的集合T除最终结果外，始终是森林，无环。
- **算法：**

```
KruskalMST(G){
    T=(V, φ)；//包含n个顶点，不含边的森林；
    依权的增序对E(G)中边排序，设结果在E[0..e-1];
    for (i=0; i<e; i++) {
        取E中第i条边(u,v);
        if （u和v分属森林T中两棵不同的树） then
            T＝T∪{（u,v）}; //否则产生回路，放弃此边
        if （T已是一棵树）then  return T；
    }//endfor
}
```

2、Kruskal算法（续）

- 例子：略
- 时间：
  - 对边排序O(elge)
  - for循环中：

    检测每条边的两个顶点是否在同一连通分量（树）上

    只要采用合适的数据结构(6.5节求等价类)，检测时间为O(lge)

    因此，此时间亦为O(elge)。
  - 总计：

    O(elge)
- 结论：时间性能主要取决于边数，适合稀疏图。

§ 7.5 拓扑排序

有向无环图DAG（Directed Acyclic Graph）的应用

1、相关概念

集合与关系

- 笛卡儿积：设A、B是两个集合，所有序偶(x,y)构成的集合（x∈A，y∈B），称为A，B的笛卡儿积，记为A×B。
- 二元关系：集合A×B的一个子集R，称为集合A与B上的一个二元关系，简称关系。若B＝A，则R称为A上的一个二元关系，他刻画了集合A中两个元素之间的关系。若(x，y)∈R，则称x和y有关系R，亦可记为xRy，否则x和y没有关系R，记为x$\not R$ y。
- 集合A上的关系R是自反的（反身的），若对∀x ∈A，都有xRx。
- 集合A上的关系R是对称的，若xRy，则yRx。其中，x，y ∈A。
- 集合A上的关系R是反对称的，若xRy，yRx,则必有x＝y.其中x,y ∈A。
- 集合A上的关系R是传递的，若xRy，yRz，则xRz。其中x,y,z ∈A。
- 例子：

数之间的相等关系，具有自反性，对称性，传递性；

数之间的小于关系，具有反自反性，传递性，反对称性；

数之间的小于等于关系，具有自反性，传递性，反对称性；

1、相关概念

偏序（部分序）

设R是集合A上的一个关系，若R具有自反性，反对称性和传递性，则称R是A上的**偏序关系**，A是**偏序集**（对于R）。

偏序关系R一般记为≤，若将关系看作是比较，则偏序关系是指集合中仅有**部分**元素是可以比较的。

全序（线性序）

设R是集合A上的一个偏序关系，若对

$\forall x,y \in A$，必有xRy或yRx，

则称R是A上的**全序关系**，A是**全序集**（对于R）。

数集合上的大小关系是全序关系

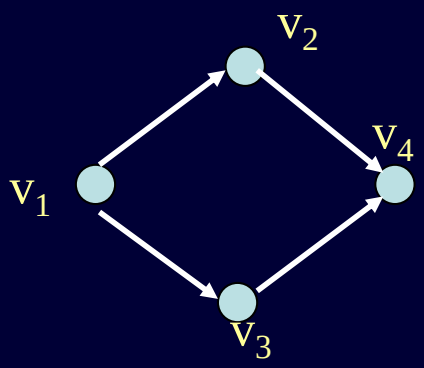
全序关系是指集合中**全体成员**之间均可比较。

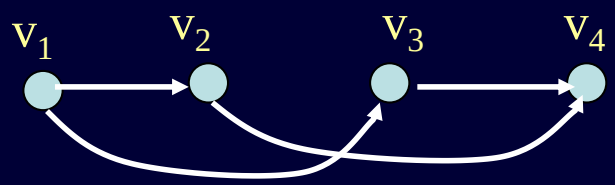
1、相关概念

拓扑排序

将一个dag图G＝（V，E）中的所有顶点排成一个线性序列，使得对G中任意一对顶点u和v，若<u,v> ∈E，则在线性序列中u在v之前。这种给顶点定序的操作称为拓扑排序。

- 拓扑（有序）序列：上述顶点的线性序列称为拓扑序列。唯一吗？
- 几何意义：将G中顶点按拓扑序列的次序排成一行，则G中所有的有向边的指向均为从左到右
- 例子：

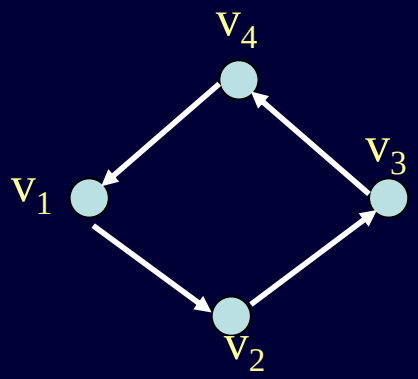


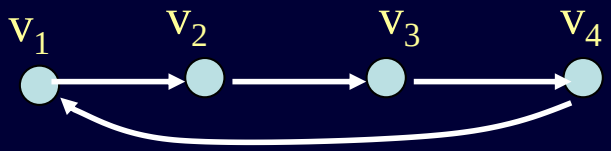


1、相关概念

拓扑排序

- **拓扑排序成功的充要条件：** 无环。
- **例子：**





- **应用背景** 有向图G可表示事件之间的先后关系，顶点表示事件，边表示事件之间的依赖关系：

<u,v> ∈E(G)表示u先于v发生，u完成后才能开始v。

若G表示施工图、生产流程图、学习计划安排，则在只能串行工作时，拓扑序列是一种可行的方案或计划。

2、求拓扑序列?

(1) 无前驱的顶点优先

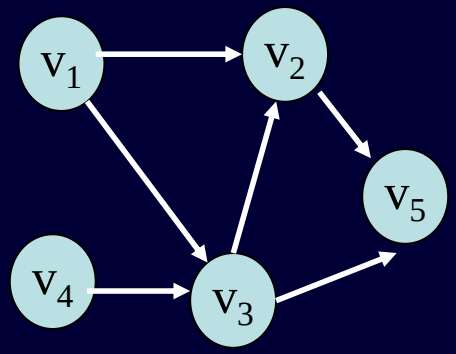
- 算法思想：输出当前入度为0的顶点。

```
NonPreFirstTopSort（G）{
    while( G中有入度为0的顶点 ) do {
        从G中选1个入度为0的顶点v输出之;
        从G中删v及其出边，出边终点的入度减1;
    }
    if ( 输出的顶点数 < | V(G) | ) then
        Error ( “G有环，排序失败” );
}
```

2、求拓扑序列？

(1) 无前驱的顶点优先

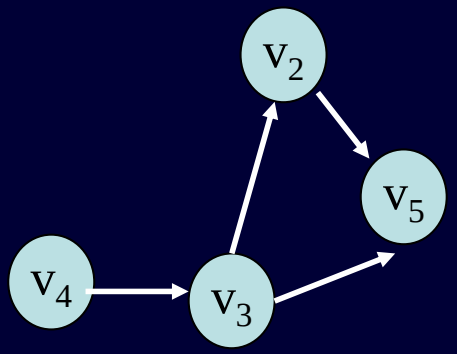
- 例子：输出v_1，v_4，v_3，v_2，v_5



Step1:

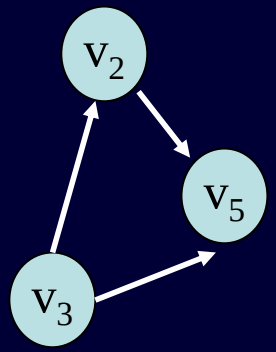
删v_1或v_4

不妨设删v_1



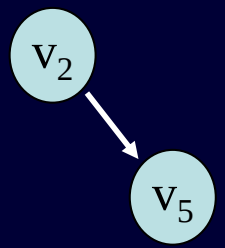
Step:

删v_4



Step3:

删v_3



Step4:

删v_2

Step5:

删v_5

2、求拓扑序列？

❖算法实现

增加一局部向量indegree[0..n]保存各顶点的当前入度

或者在邻接表的顶点表中增加入度域

用栈（或队列）来保存所有入度为0的顶点，以免每次选入度为0的顶点时扫描整个indegree向量

```
void NonPreFirstTopSort（ALGraph G）{ //以下v_i简称为i
    int indegree[MaxVertexNum]，i，j，count=0；
    SeqStack S;
    EdgeNode *p;

    for( i=0; i<G.n; i++ ) indegree[i]=0;
    for( i=0; i<G.n; i++ ) //对每个顶点i
        for( p=G.adjlist[i].firstedge; p; p=p->next)//扫描i的出边表
            indegree[p->adjvex]++; //将<i,j>的终点j入度加1，j=p->adjvex
    InitStack(&S);
    for (i=0; i<G.n; i++) if ( !indegree[i] ) Push(&S,i);//入度为0的顶点入栈
```

2、求拓扑序列？

```
while( !StackEmpty(&S) ){//栈非空时,图中仍有入度为0的顶点
    i=Pop(&S); //删除无前驱的顶点i
    printf("%c\t",G.adjlist[i].vertex); //输出顶点i
    count++; //输出顶点计数
     for (p=G.adjlist[i].firstedge; p; p==p->next){ //扫描顶点i的出边表
          j=p->adjvex; //j是<i,j>的终点
          indegree[j]--;  //j的入度减1，相当于删出边<i,j>
          if ( ! indegree[j] ) Push(&S, j); //j的入度为0则进栈
      }
  }
 if ( count<G.n ) printf( "\nG is not a DAG\n");
}
```

❖时间：初始化O(n+e)

排序成功时最大：每个顶点入出栈各1次，每个边表节点被检查1次O(n+e)

2、求拓扑序列?

(2) 无后继的顶点优先

- 算法思想：输出当前出度为0的顶点。

```
NonSuccFirstTopSort（G）{ //应选G的逆邻接表
    while( G中有出度为0的顶点 ) do {
        从G中选1个出度为0的顶点v输出之；
        从G中删v及其入边，入边起点的出度减1；
    }
    if ( 输出的顶点数 < | V(G) | ) then
        Error ( “G有环，排序失败” );
}
```

- 输出结果：逆拓扑序列
- 算法实现：略

2、求拓扑序列?

(3) 利用dfs遍历算法

- 原理：因为当从某顶点v出发的dfs搜索完成时，v的所有后继必定均已访问过（想象他们均已被删除），此时v相当于是无后继的顶点，所以可在dfs算法返回前输出顶点v，即可得到DAG的逆拓扑序列。
- 算法：

```
DFSTopSort（G，i，T）{ //在DFSTraverse中调用此算法，T是栈
        visited[i]=TRUE; //访问i
        for (所有i的邻接点j）//即< i, j> ∈E(G)
            if ( !visited[j] )
                DFSTopSort(G,j,T)；
        Push（&T, i）//从i出发的搜索已完成，输出i
}
```

- 特点：与NonSuccFirstTopSort算法类似；若G有环，则不能正常工作

§7.6 最短路径

- **应用背景：交通咨询、导航**
- **约定**

  有向图

  设V＝{0,1,…,n-1}，边上的权值非负（长度）

- **分类**

①单源最短路径：1个源点到其余顶点的最短路径

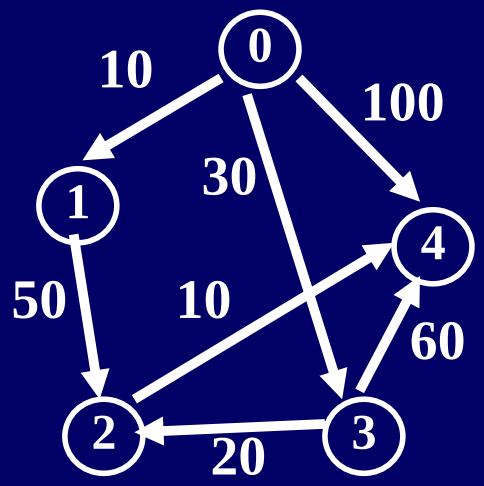
②单目标最短路径：将各边反向，即为问题1

③单点对间最短路径：可用①来解，但二者渐近时间相同

④所有点对间最短路径：亦可用①来解，即每个顶点作为源点调用①

§7.6.1 单源最短路径问题

- 观察



源点	中间顶点	终点	长度
0		1	10
0		3	30
0	3	2	50
0	3,2	4	60

上表是按路径长度递增序产生的从源点到其余顶点的最短路径

0到4的路径：<0,4>, <0,3,4>, <0,1,2,4>, <0,3,2,4>

长度： 100， 90， 70， 60

- 规律：当按长度增序生成从源s到其它顶点的最短路径时，则当前正在生成的最短路径上除终点外，其余顶点的最短路径均已生成
- 例子：当求0到2的最短路径时，则该路径<0,3,2>上顶点0,3的最短路径在此前已生成

§7.6.1 单源最短路径问题

- **约定**

  从源s到终点v的最短路径简称为v的**最短路径**，SP(v)

  s到v的最短路径长度简称为v的**最短距离**，SD(v)

  红点集S：最短距离已确定的顶点集合

  白点集V-S：最短距离尚未确定的顶点集合

- **算法思想- Dijkstra（1972图灵奖得主）**

  基于上述观察

  - **初始化**：仅已知源s的最短距离SD(s)=0，故红点集S={s};
  - **扩充红点集**：算法的每一步均是在当前白点集中选一最短距离最小的白点来扩充红点集，以保证算法是按**长度增序来产生各顶点的最短路径**;
  - **结束**：当前白点集空或仅剩下最短距离为∞的白点为止。注：若s到某白点的路径不存在，可假设该白点的最短路径是一条长度为∞的虚拟路径。

§7.6.1 单源最短路径问题

- 如何扩充红点集？

∵白点k的最短路径上除终点外，其余顶点的最短路径均已生成，故它们均为红点

∴设置向量D[0..n-1]，对每一个白点v∈V-S，用D[v]记录从源点s到达v，且除v外中间不经过任何白点的“最短”路径长度。初始时每个白点v的D[v]值是边<s,v>上的权。

Note：从s到v的中间不经过其他白点的路径可能不止1条，但只需将其中最短的那条的长度记录在D[v]中。

D[v]=SD[v]? 即D[v]是v最终的最短距离吗？不一定，因为s到v可能存在包含其它白点作为中间点的更短路径。

D[v]只是v当前估计的最短距离（简称估计距离），即：D[v]≥SD[v]

- 如何在当前白点集中选一最短距离最小的白点k来扩充红点集？

§7.6.1 单源最短路径问题

- 如何扩充红点集？

Th.7.6.1 若k是白点集中估计距离最小的顶点，则k的估计距离就是最短距离。
即：若D[k]=min{ D[i]：∀i∈V-S }，则D[k]=SD[k]

Pf（反证法）

设D[k]不是k的最短距离，则必存在一条路径P：s ⇝ k，其长度

Length(p)<D[k] （式1）

由D[k]定义知，P至少包含1个白点作为中间点，不妨设x是P上第1个白点，则P可分解为： s ⇝ x， x ⇝ k。其中P1中仅有x为白点，由D[x]定义知length[P1]≥D[x]，又因权为非负，故length[P2]≥0，所以

length(P)=length(P1)+length(P2)≥ D[x] （式2）

由式1,2得：D[k]>length(P)≥D[x]，这与k是当前白点集中估计距离最小的顶点矛盾！

k是最短距离最小的白点吗？

定理保证了k加入红点集的正确性

§7.6.1 单源最短路径问题

- 如何调整白点集中白点的估计距离？

由于新红点k可能导致剩余白点的估计距离变小，使之离源点更近，故需调整。

设∀j∈V-S，若D[j]由于k加入红点集而变小，则新路径P必是

s ⇝p1 k ⇝p2 j，且P1中只有红点，P2必是边<k,j>，即：

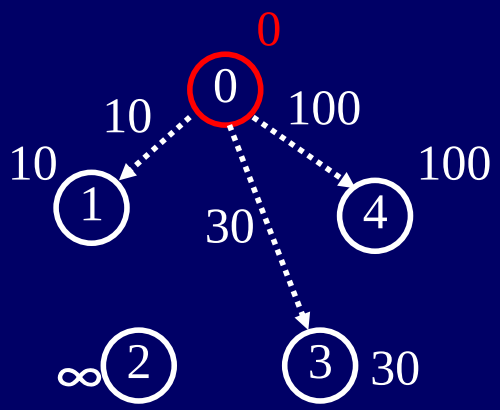
Length(p)= D[k] + w<k, j>.　证明：略

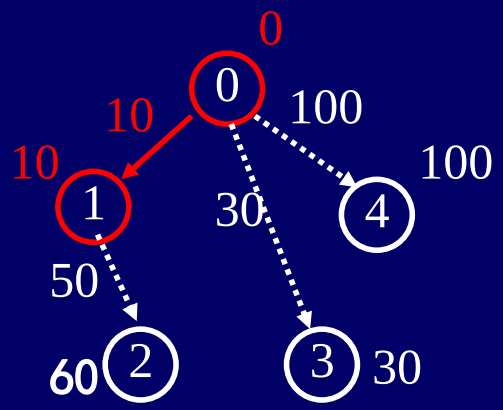
❖调整方法

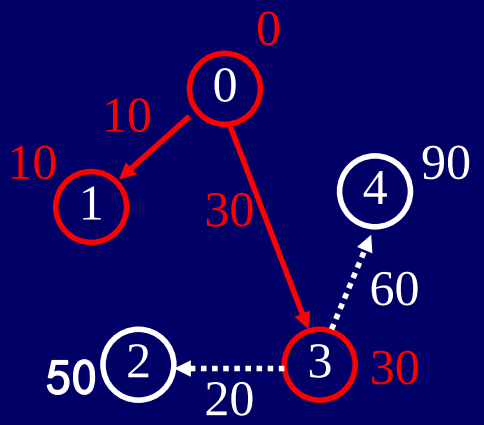
若length(P)<D[j]，则用length(P)来修正D[j]。

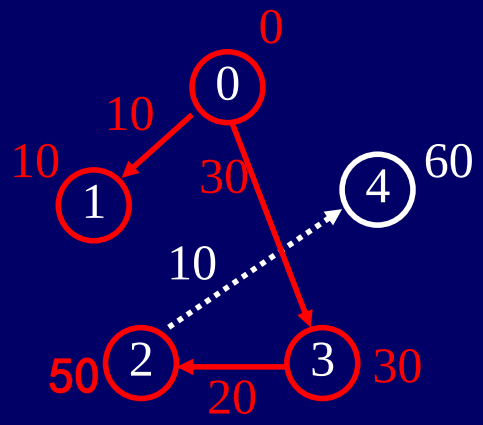
§7.6.1 单源最短路径问题

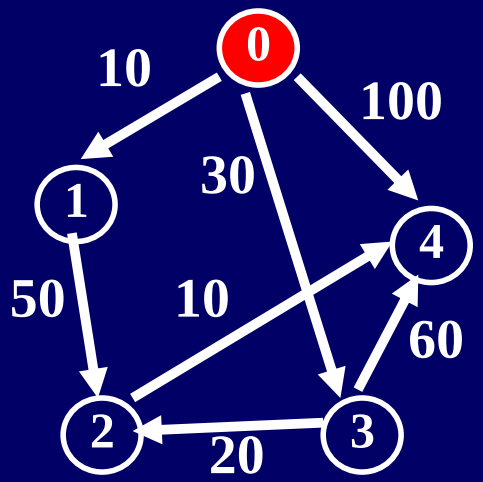
■ 例子











最短距离：红色

估计距离：白色

依次求出的最短距离为：

1) D[0]=0
2) D[1]=10，调整顶点2
3) D[3]=30，调整顶点2，4
4) D[2]=50，调整顶点4
5) D[4]=60

- **最短路径树：** 各顶点的最短路径（实线）总是一棵以源点为根的树，称之为最短路径树。

§7.6.1 单源最短路径问题

- 如何构造最优解

因为D向量只记录了最优解的值，但不能得到最优解。因此，要记录最优解则须引入附加信息。

因为最优解是最短路径树，故只需增加一个向量P[0..n-1]，用P[i]记录顶点的双亲，由双亲的唯一性知，顶点i的最短路径可从P[i]反复上溯至根（源点）即可求得最优解。

- 算法实现

$$G[i][j]=\begin{cases}\infty & \text{if } <i,j>\text{不是边}\\ w(<i,j>) & \text{otherwise}\end{cases}$$

§7.6.1 单源最短路径问题

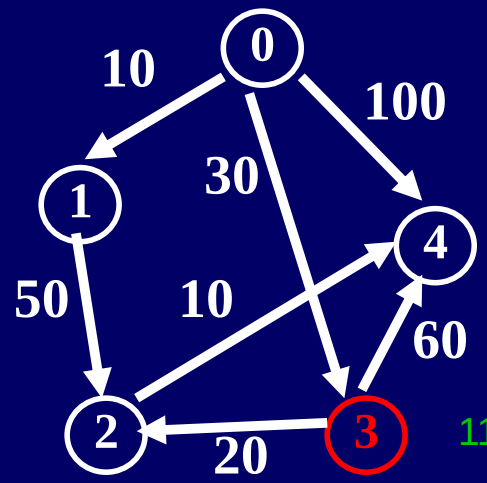
```
void Dijkstra ( AdjMatrix G, Distance D, Path P, int s ){
    //0≤s ≤n-1,若<i,j>不是边，则G[i][j]=Infinity
    Boolean S[n]；//S是红点集。S[i]为真表示j为红点，否则为白点
    for ( i=0; i<n; i++) { //初始化
        S[i]=FALSE;
        D[i]=G[s][i]; //置初始的估计距离
        if ( D[i]<Infinity ) P[i]=s; //s是i的前驱(双亲)
        else  P[i]=-1; // i无前驱，注意P[s]亦无前驱
    }
   S[s]=TRUE; D[s]=0；//红点集仅有源点s
```

```
    for ( i=0; i<n-1; i++) { //向红点集S扩充n－1个红点
        min=Infinity;
        for ( j=0; j<n; j++ ) //选估计距离最小的白点k（离s最近）
            if ( !S[ j ]&&D[ j ]<min ) {
                min=D[ j ]; k=j;
            }
        if (min==Infinity) return; // 白点集为空或只剩下无最短路径的点
        S[k]=TRUE; // k加入红点集
        for ( j=0; j<n; j++ ) //调整剩余白点的估计距离
            if ( !S[ j ]&&D[ j ]>D[k]+G[k][ j ] ) {
                D[ j ] = D[k]+G[k][ j ]; //修改白点j的估计距离，使之离s更近
                P[ j ] = k; // k是j的前驱
            }
     }//endfor
}
```

- **算法执行过程**

源点s＝3

循环i	红点集S	k	D[0]	D[1]	D[2]	D[3]	D[4]	P[0]	P[1]	P[2]	P[3]	P[4]
初始化	{3}	—	∞	∞	20	0	60	-1	-1	3	-1	3
1	{3,2}	2	∞	∞	20	0	30	-1	-1	3	-1	2
2	{3,2,4}	4	∞	∞	20	0	30	-1	-1	3	-1	2
3	同上	—	白点集{0,1}中所有点的估计距离为∞，退出循环									



- **时间：** 时间$O(n^2)$
- **构造解**

构造解

输出路径及其长度

```
void PrintPath（Path P，Distance D）{ //路径是逆向的
   int i，pre;
   for ( i=0; i<n; i++ ) {  //依次打印点i的最短路径及长度
        printf("\nD:%d, P:%d", D[i], i); //输出终点i
        pre=P[i]; //找终点i的前驱
        while ( pre != -1 ) {
             printf( ", %d", pre );
             pre=P[pre]; //上溯找前驱
        }
   }
}
```

§7.6.2 所有点对间的最短路径问题

- **解法一**

  以每一顶点为源点，调用Dljkstra算法，时间$O(n^3)$

- **解法二**

  Floyd（1978年图灵奖得主）算法更简洁，但是时间仍为$O(n^3)$

  - 假设：加权邻接矩阵C（n×n）

$$C[i][j]=\begin{cases}0 & \text{if } i=j\\ \infty & \text{if } <i,j>\text{不是边}\\ w(<i,j>) & \text{otherwise}\end{cases}$$

  - 思想：

    ∀i,j∈V，若<i,j>∈E，则从i到j有一条路径，长度为C[i][j]。

    但它不一定是最短路径，因为可能存在一条从i到j中间包含其他顶点的更短路径。因此，必须依次考虑能否在i和j之间加入顶点0,1…,n-1,而得到更短的路径。

§ 7.6.2 所有点对间的最短路径问题

Floyd算法的基本步骤

定义n×n的方阵序列$D_{-1}, D_0, \ldots D_{n-1}$,

- 初始化： $D_{-1}=C$

  $D_{-1}[i][j]$＝边<i,j>的长度，表示初始的从i到j的最短路径长度，即它是从i到j的中间不经过其他中间点的最短路径。

- 迭代：设D_{k-1}已求出，如何得到D_k（$0 \leqslant k \leqslant n-1$）？

  $D_{k-1}[i][j]$表示从i到j的中间点不大于k－1的最短路径p：i…j,

  考虑将顶点k加入路径p得到顶点序列q：i…k…j,

  若q不是路径或q是长度大于p长度的路径，则当前的最短路径仍是上一步结果：$D_k[i][j]= D_{k-1}[i][j]$；否则用q取代p作为从i到j的最短路径。

  因为q的两条子路径i…k和k…j皆是中间点不大于k－1的最短路径，所以从i到j中间点不大于k的最短路径长度为：

  $$D_k[i][j]=\min\{ D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j] \}$$

  矩阵序列$D_{-1}, D_0, \ldots D_{n-1}$可在同1个矩阵上迭代求得，为什么？

§ 7.6.2 所有点对间的最短路径问题

Floyd算法的基本步骤

❖解矩阵：

Path[n][n]：记录路径构造

在第k次跌代中，P[i][j]记录从i到j的中间点序号不大于k的最短路径上顶点i的后继顶点。

当算法结束时，可由Path[i][j]得到从i到j的最短路径，其方法是从顶点i开始反复找其后继，直至找到顶点j或确认i到j没有路径为止。

§ 7.6.2 所有点对间的最短路径问题

Floyd算法实现

```
void Floyd( AdjMatrix D, AdjMatrix C, int Path[n][n] ) {
    for ( i=0; i<n; i++ )
        for ( j=0; j<n; j++ ) { //初始化
            D[ i][ j]=C[ i][ j];
            if ( C[ i][ j]=Infinity ) Path[i][j]=-1;
            else  Path[ i][ j]=j; //j是i的后继
        }//endfor
     for ( k=0;k<n;k++ ) //将k扩充到从i到j的最短路径上
          for ( i=0;i<n;i++ )
               for ( j=0;j<n;j++ )
                    if ( D[ i][ k]+D[ k][ j]<D[ i][ j] ) {
                         D[ i][ j]=D[ i][ k]+D[ k][ j]; //修改路径长度
                         Path[ i][ j]=Path[ i] [ k]; //修改路径
                    }
}
```

§ 7.6.2 所有点对间的最短路径问题

- Floyd算法实现

```
void PrintPath( AdjMatrix D,  int Path[n][n] ) {//不妨设D是整型矩阵
    for ( i=0; i<n; i++ )
        for ( j=0; j<n; j++ ) {
            printf( "%d", D[ i][ j] ); //i到j的最短路径长度
            next=Path[i][j]；//next为起点i的后继
            if （ next == -1 ）//i无后继表示i到j无路径
                printf（ "%d to %d no path.\n", i, j ）；
            else {
                printf(  "%d", i ); //输出起点
                while ( next != j ){
                    printf( "->%d", next ); //输出后继顶点
                    next=Path[next][j]; //继续找后继顶点
                } //endwhile
                printf( "%->%d", j ); //输出终点
            } //endif
        } //endfor
}
```

Ch.9 查找

§9.1 基本概念

查找和排序是两个重要的运算

- **对象：**表、文件等。其中每个结点（记录）由多个数据项构成，假设每个结点有1个能唯一标识该结点的key。
- **定义：**给定1个值K，在含有n个结点的表中找出关键字等于K的结点，若找到（查找成功），则返回该结点信息或它在表中的位置；否则（查找失败），返回相关指示信息。
- **分类：**
  - 查找过程中是否修改表？动态查找、静态查找
  - 查找过程是否均在内存中进行？内部查找、外部查找
- **效率：**与存储结构、文件状态（有序、无序）有关

  平均查找长度(ASL)，即平均的比较次数，作为衡量查找效率的标准：

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

P_i：查找第i个结点的概率

C_i：查找第i个结点的比较次数

设$P_i = 1/n$, $1 \leqslant i \leqslant n$

- **约定：**typedef int KeyType;

§ 9.2 线性表的查找

- **对象：**用线性表作为表的组织形式。
- **分类：**静态查找、内部查找
- **方式：**顺序查找、二分查找、分块查找

define n 100//

§ 9.2.1 顺序查找

- **基本思想**

  从表的一端开始，顺序扫描线性表，依次将扫描到的结点的Key与给定值K进行比较，若当前扫描到结点Key＝K，则查找成功返回；若扫描完整个表后，仍未找到，则查找失败。

- **适用范围：**顺序表、链表
- **算法：**

§ 9.2.1 顺序查找

```
typedef struct{
        KeyType key;
        InfoType otherinfo; //应用相关
}NodeType, SeqList[n+1]; //0号单元作为哨兵

int SeqSearch( SeqList R, KeyType K) {
     //在R[1..n]中查找，成功时返回结点位置，失败时返回0
     int n;
     R[0].key=K; //设置哨兵
     for ( i=n; R[i].key != K; i-- ); //从后往前找
     return i; //若i为0，则失败
}
```

§9.2.1 顺序查找

- 哨兵的作用

  for中省略了下标越界i>=1判定,节约了约一半时间

- 时间分析

  成功：$ASL_{ss}=(n+1)/2$，key的平均比较次数约为表长的一半

  失败：n+1次比较

- 优缺点

  优点：简单，对存储结构、Key之间的关系均无特殊要求

  缺点：效率低，当n较大时不宜用

§ 9.2.2 二分（折半）查找

- 适用范围：顺序表、有序
- 基本思想（分治法）

（1）设R[low..high] 是当前查找区间，首先确定该区间的中点位置：mid= (low+high)/2 //整除

（2）将待查的K值与R[mid]比较，

① K＝R[mid].key：查找成功，返回位置mid

② K＜R[mid].key：则左子表R[low..mid-1]是新的查找区间

③ K＞R[mid],key：则右子表R[mid+1..high]是新的查找区间

初始的查找区间是R[1..n]，每次查找比较K和中间点元素，若查找成功则返回；否则当前查找区间缩小一半，直至当前查找区间为空时查找失败。

§ 9.2.2 二分（折半）查找

- 算法：

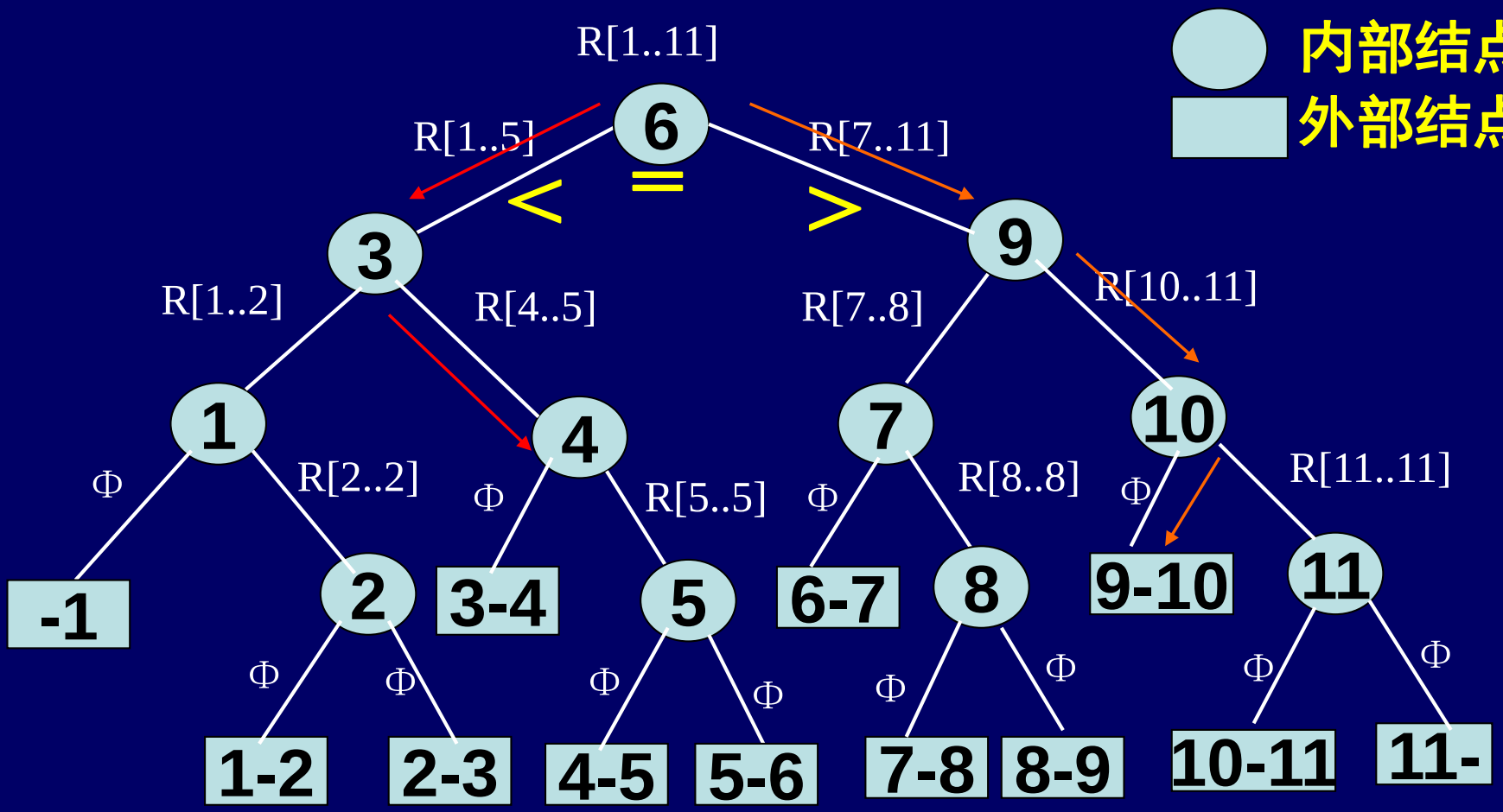
```
int BinSearch( SeqList R, KeyType K ) {
    int mid, low=1, high=n;
    while ( low < high ) { //当前查找区间R[low..high]非空
        mid= (low+high)/2;  //整除
        if ( R[mid].key==K ） return mid; //成功返回位置mid
        if ( K＜R[mid].key ) //两个子问题求解其中的一个
            high=mid-1; //在左区间中查找
        else
            low=mid+1;  //在右区间中查找
    } // endwhile
    return 0; //当前查找区间为空时失败
}
```

§ 9.2.2 二分（折半）查找

- **查找过程**　　**判定(比较)树**：分析查找过程的二叉树，形态只与结点数相关，与输入实例的取值无关（为什么？）。例如n＝11的形态如下图所示。

1 2 3 4 5 6 7 8 9 10 11

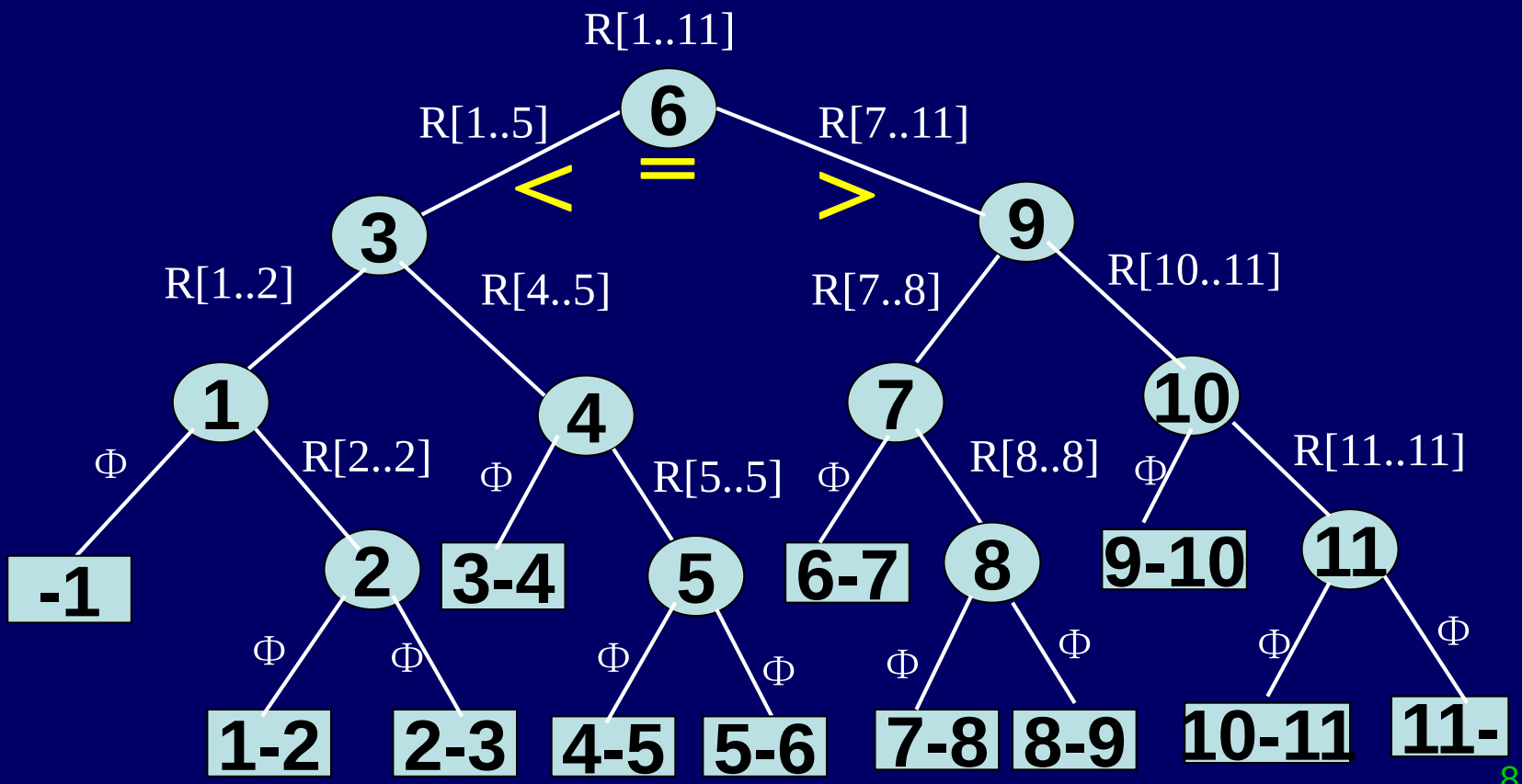
（05,13,19,21,37,56,64,75,80, 88, 92），找21成功，找85失败。



§ 9.2.2 二分（折半）查找

时间分析

- 查找R[6]：比较1次；查找R[3]、R{9]：比较2次；查找R[1]、R[4]、R[7]、R[10]：比较3次；查找R[2]、R[5]、R[8]、R[11]：比较4次



§ 9.2.2 二分（折半）查找

时间分析

- **总结**：查找过程走了一条从判定树的根到被查结点的路径。

  成功：终止于一个内部结点，所需的Key比较次数恰为该结点在树中的层数；

  失败：终止于一个外部结点，所需的Key比较次数为该路径上内部结点总数。（层数－1）

- **平均查找长度（ASL）**：

  设$n=2^h-1$, 则树高$h=\lg(n+1)$ //不计外部结点的满二叉树

  第k层上结点数为2^{k-1}，找该层上每个结点的比较次数为k，在等概率假设下：

$$ASL_{bs}=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\sum_{k=1}^{h}k2^{k-1}=\frac{n+1}{n}\lg(n+1)-1\approx\lg(n+1)-1$$

- **最坏时间**：查找失败，不超过树高 $\lceil\lg(n+1)\rceil$
- 在链表上可做折半查找吗？

§ 9.2.2 分块查找（索引顺序查找）

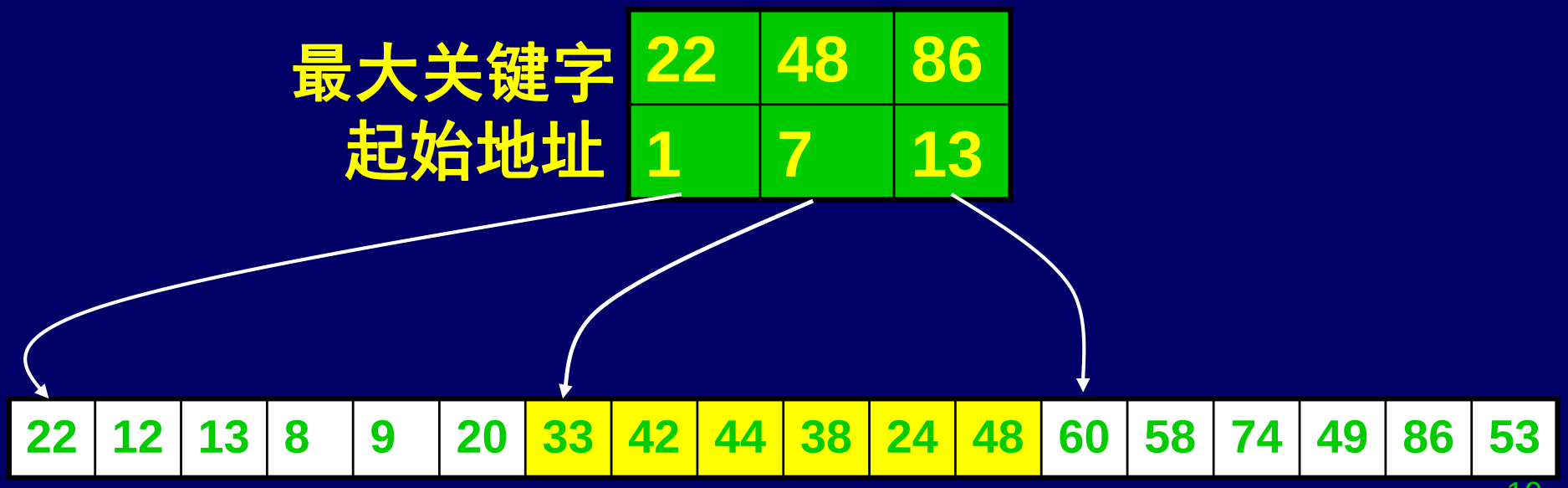
存储结构

将R[1..n]均分成b块，前b－1块中结点数为s＝⌈n/b⌉，最后一块中的结点数可能小于s，引入索引表标记块。

关键字状态

分块有序：块间有序，块内不一定有序;

例子：n＝18，b＝3，s＝6



§ 9.2.2 分块查找（索引顺序查找）

- 查找过程（两步查找）
  - 索引表查找：确定块。n较大时用折半查找，n较小时用顺序查找。
  - 块内查找：只能顺序查找。
- 性能分析
  - 以折半查找确定块：

    $ASL = ASL_{bs} + ASL_{ss} = \lg(b+1) - 1 + (s+1)/2 \approx \lg(n/s+1) + s/2$

  - 以顺序查找确定块：

    $ASL = ASL_{ss} + ASL_{ss} = (b+1)/2 + (s+1)/2 = (s^2+2s+n)/(2s)$

    当$s = \sqrt{n}$ 时，ASL取极小值$\sqrt{n}+1$
  - 块不一定等分

§ 9.3 树上的查找（动态查找）

折半查找效率最高，但它不适应插删操作。本节讨论适用于动态查找，即查找过程中动态维护表结构。

§ 9.3.1 二叉查找树（BST）

- **定义**：二叉查找树或是空树，或满足下述**BST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keys均小于根的key

  ②若它的右子树非空，则右子树上所有结点的keys均大于根的key

  ③左右子树又都是二叉查找树

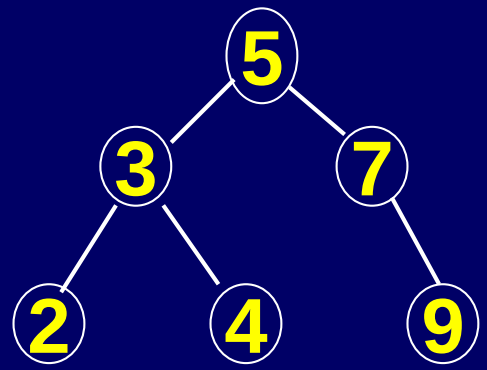
  Note：性质1或2中，可以将“＜”改为“≤”；或“＞”改为“≥”

- **BST性质**

  二叉查找树的中序序列是一个递增有序序列

§ 9.3.1 二叉查找树（BST）

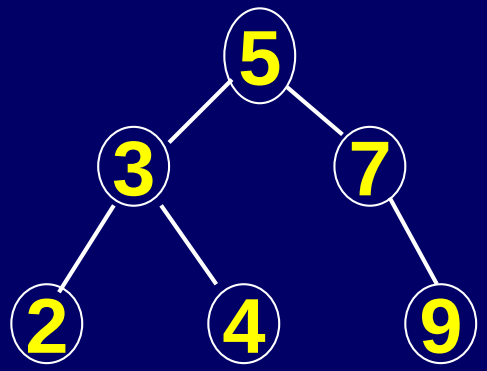
- 举例



- 存储结构：

```
typedef struct node {
    KeyType key;
    InfoType otherinfo;
    struct node *lchild, *rchild;
} BSTNode, *BSTree;
```

§ 9.3.1 二叉查找树（BST）



1.查找

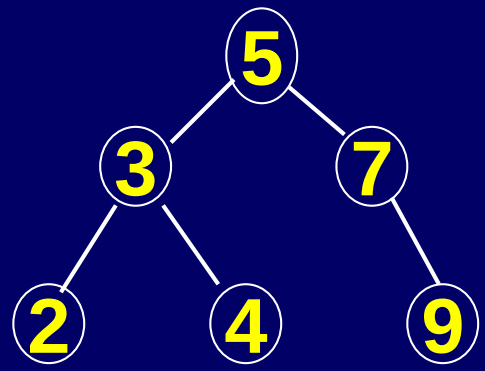
- **基本思想**

从根结点开始比较，若当前结点的key与待查的key相等，则查找成功，返回该结点的指针；否则在左子树或右子树中继续查找。若树中没有待查结点，则失败于某个空指针上。

- **算法**

```
BSTNode *SearchBST( BSTree T, KeyType key ){
  //在T上查找key=K的节点，成功时返回该节点，否则返回NULL
  if  ( T==NULL || key==T->key ) return T; //若T为空，查找失败；否则成功
  if ( key < T->key )
      return SearchBST ( T->lchild, key ); //查找左子树
  else
      return SearchBST ( T->rchild, key ); //查找右子树
}
```

§ 9.3.1 二叉查找树（BST）



1.查找

- 时间

若成功，则走了一条从根到待查节点的路径

若失败，则走了一条从根到叶子的路径

❖上界：O(h)

❖分析：与树高相关

①最坏情况：单支树，ASL=(n+1)/2, 与顺序查找相同

②最好情况：ASL≈lgn，形态与折半查找的判定树相似

③平均情况：假定n个keys所形成的n!种排列是等概率的，则可证明由这n!个序列产生的n!棵BST（其中有的形态相同）的平均高度为O(lgn)，故查找时间仍为O(lgn)

2、BST的插入和生成

■ 插入

❖ 算法思想　保证新结点插入后满足BST性质，基本思想如下：

- ➢ 若T为空，则为待插入的Key申请一个新结点，并令其为根；
- ➢ 否则，从根开始向下查找插入位置，直到发现树中已有Key，或找到一个空指针为止；
- ➢ 将新结点作为叶子插入空指针的位置。

查找过程是一个关键字的比较过程，易于写出递归或非递归算法，也可以调用查找操作。

❖ 算法实现

2、BST的插入和生成

```
void InsertBST( BSTree *T,  KeyType key ) { //*T是根
    BSTNode *f,  *p=*T; //p指向根结点
    while( p ) { //当树非空时，查找插入位置
        if ( p->key==key ) return; //树中已有key，不允许重复插入
        f＝p；// f和p为父子关系
        p＝( key < p->key ) ? p->lchild; p->rchild;
    } //注意：树为空时，无须查找
    p = (BSTNode *) malloc(sizeof(BSTNode));
    p->key=key;  p->lchild=p->rchild=NULL; //生成新结点
    if ( *T ==NULL )  * T=p; //原树为空,新结点为根
    else //原树非空, 新结点作为*f的左或右孩子插入
        if ( key < f->key )  f->lchild=p;
        else  f->rchild=p;
} //时间为O（h）
```

2、BST的插入和生成

生成

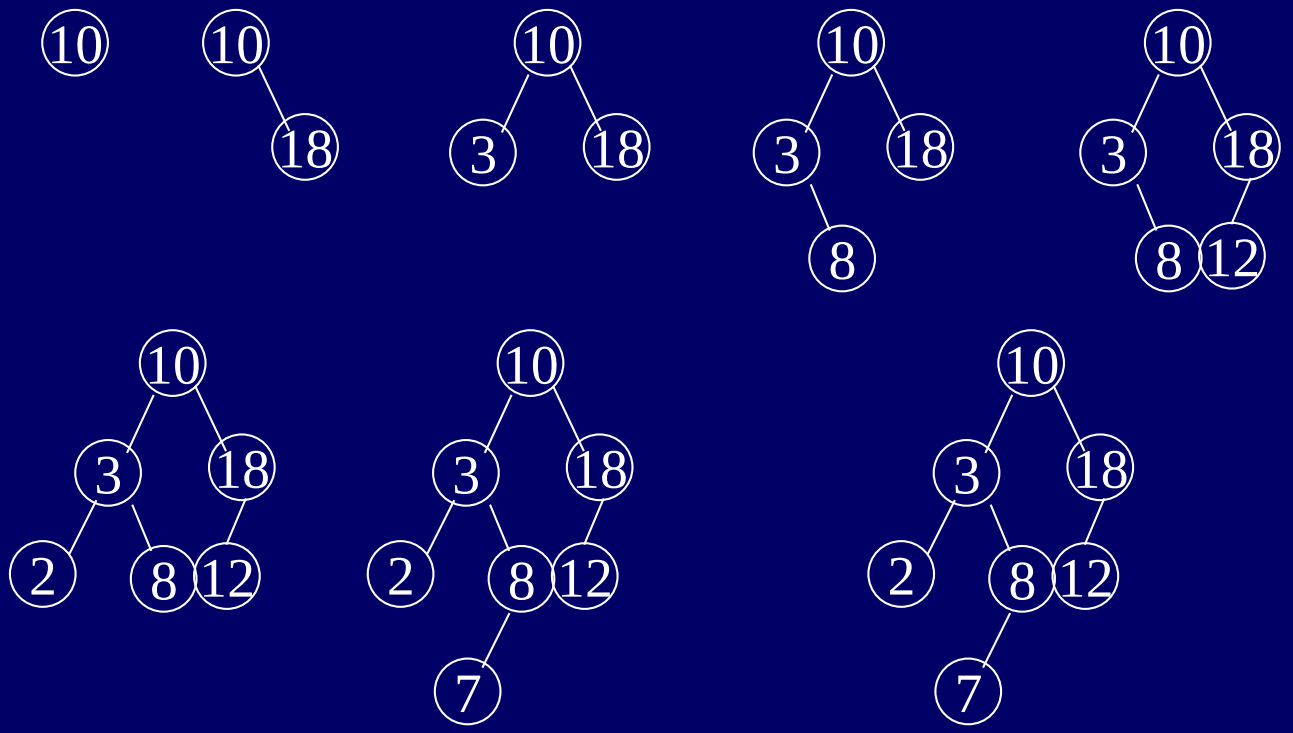
- 算法： 从空树开始，每输入一个数据就调用一次插入算法。

```
BSTree CreateBST( void ) {
    BSTree  T＝NULL; //初始时T为空
    KeyType key;
    scanf( “%d”, &key );
    while( key ) {  //假设key＝0表示输入结束
        InsertBST ( &T, key ) ; //将key插入树T中
        scanf( “%d”, &key )；// 读入下一个关键字
    }
    return T；// 返回根指针
}
```

2、BST的插入和生成

举例

输入实例 {10， 18， 3， 8， 12， 2， 7， 3}



2、BST的插入和生成

- **一般情况**

不同的输入实例（数据集不同、或排列不同），生成的树的形态一般不同。对n个结点的同一数据集，可生成n！棵BST。

- **例外情况**

但有时不同的实例可能生成相同的BST，例如：（2，3，7，8，5，4）和（2，3，7，5，8，4）可构造同一棵BST。

- **排序树名称的由来**

因为BST的中序序列有序，所以对任意关键字序列，构造BST的过程，实际上是对其排序。

生成n个结点的BST的平均时间是O(nlgn)，但它约为堆排序的2—3倍，因此它并不适合排序。

3、BST的删除

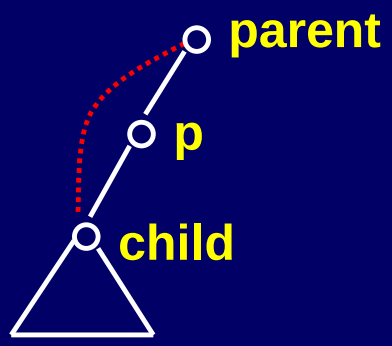
保证删一结点不能将以该结点为根的子树删去，且仍须满足BST性质。即：删一结点相当于删有序序列中的一个结点。

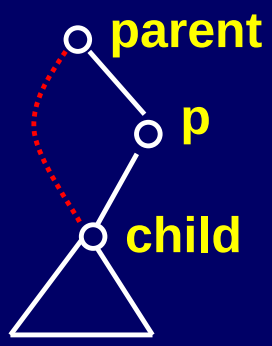
基本思想

- 查找待删结点*p，令parent指向其双亲（初值NULL）；若找不到则返回，否则进入下一步。
- 在删除*p时处理其子树的连接问题，同时保持BST性质不变。

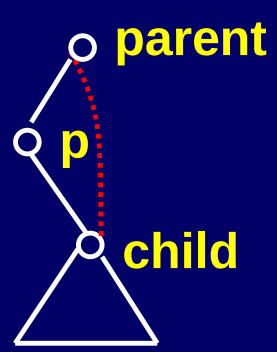
case1:*p是叶子，无须连接子树，只需将*parent中指向*p的指针置空

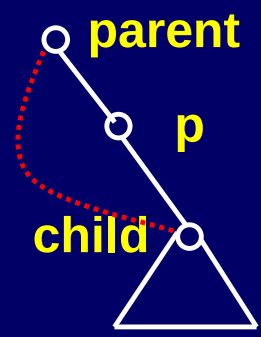
case2:*p只有1个孩子，只须连接唯一的1棵子树，故可令此孩子取代*p与其双亲连接（4种状态）





*p只有左孩子





*p只有右孩子

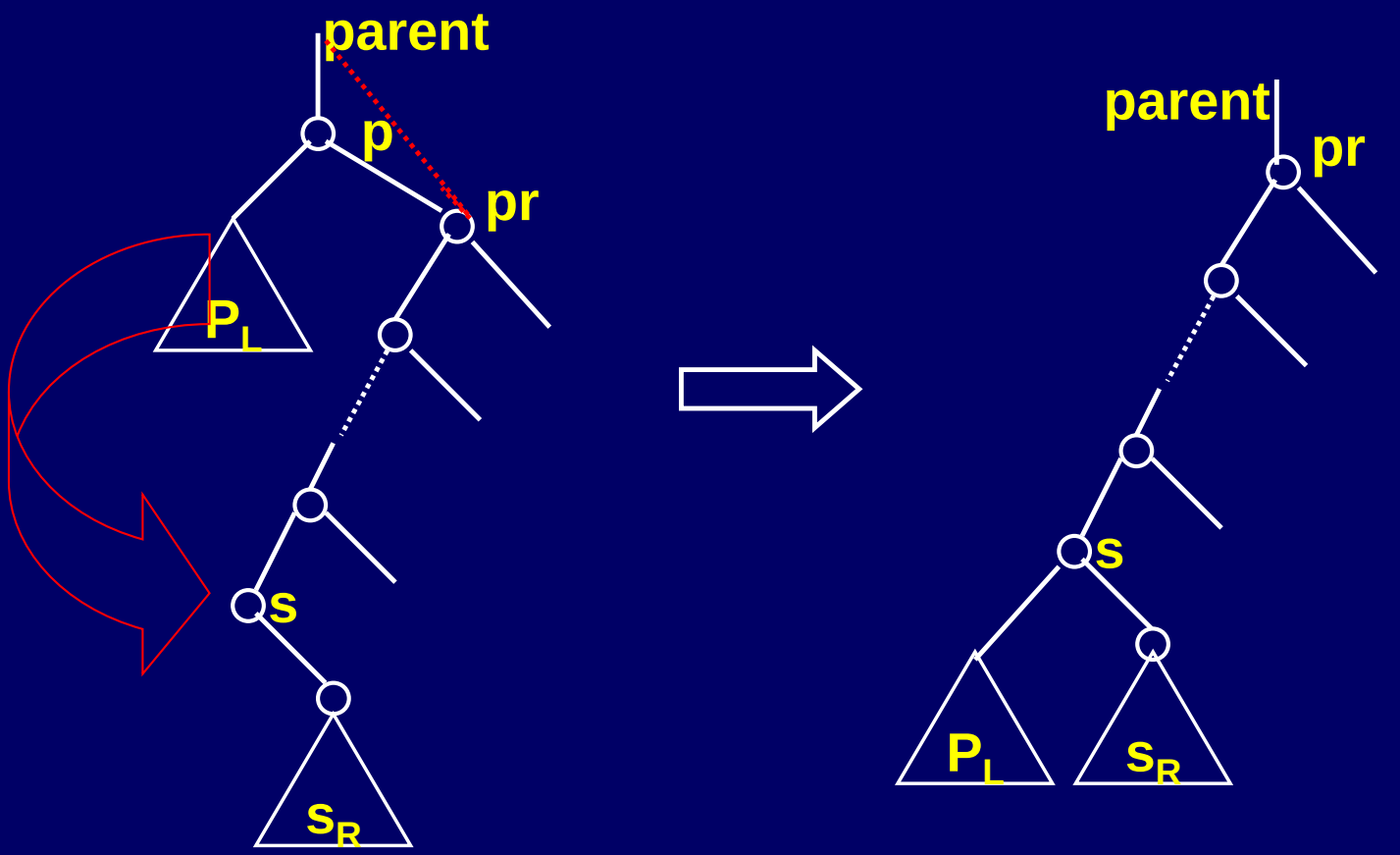
3、BST的删除

基本思想

case3: *p有2个孩子，有2种处理方式：

①找到*p 的中序后继(或前驱)*s，用*p的右(或左)子树取代*p与其双亲*parent连接；而*p的左(或右)子树P_L则作为*s的左(或右)子树与*s连接。

缺点：树高可能增大。

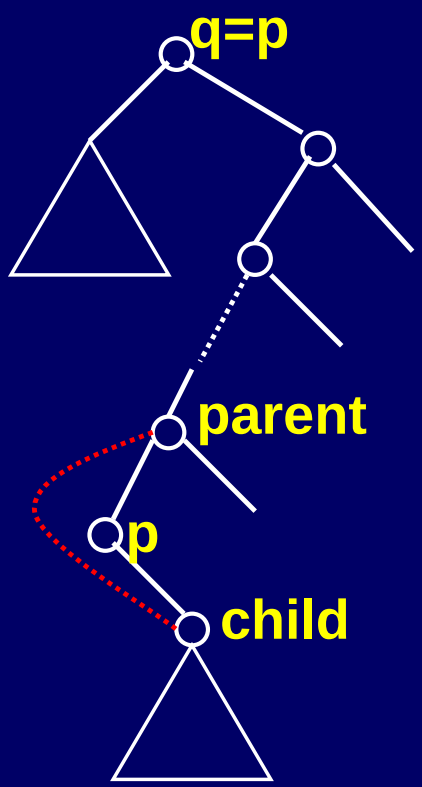


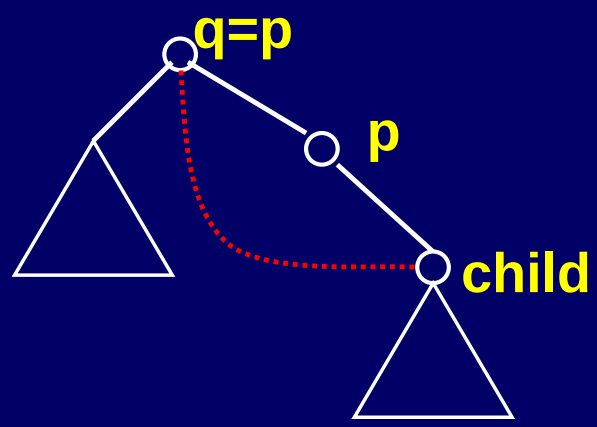
3、BST的删除

- 基本思想

②令q=p，找*q的中序后继*p，并令parent指向*p的双亲，将*p的右子树取代*p与其双亲*parent连接。将*p的内容copy到*q中，相当于删去了*q，将删*q的操作转换为删*p的操作。对称地，也可找*q的中序前驱

因为*p最多只有1棵非空的子树，属于case2。实际上case1也是case2的特例。因此，case3采用该方式时，3种情况可以统一处理为case2。





*q的中序后继就是其右孩子

3、BST的删除

- 算法

```
void  DelBSTNode( BSTree *T,  KeyType key) {  //*T是根
    BSTree *q, *child, *parent=NULL, *p=*T;
    while ( p ) { //找待删结点
          if  ( p->key ==key ) break;  //已找到
          parent=p; //循环不变式是*parent为*p的双亲
          p＝( key < p->key ) ? p->lchild; p->rchild;
    }
    if ( !p ) return;  //找不到被删结点，返回
    q=p; //q记住被删结点*p
    if (q->lchild && q->rchild ) //case3，找*q的中序后继
        for(parent=q,p=q->rchild; p->lchild; parent=p, p=p->lchild);
```

3、BST的删除

```
//现在3种情况已统一到情况2，被删结点*p最多只有1个非空的孩子
    child=(p->lchild)? p->lchild; p->rchild; //case1时child空，否则非空
    If ( !parent ) //*p的双亲为空，说明 *p是根，即删根结点
        *T=child; //若是情况1，则树为空；否则*child取代*p成为根
    else { //*p非根，它的孩子取代它与*p的双亲连接，即删 *p
        if ( p==parent->lchild )  parent->lchild=child;
        else   parent->rchild=child;
        if ( p != q ) //情况3，将*p的数据copy到*q中
            q->key=p->key; //若有其他数据亦需copy
    }
    free( p ); //删除*p
} //时间O(h)
```

4、平衡二叉树(AVL)

BST是一种查找效率比较高的组织形式，但其平均查找长度受树的形态影响较大，形态比较均匀时查找效率很好，形态明显偏向某一方向时其效率就大大降低。因此，希望有更好的二叉排序树，其形态总是均衡的，查找时能得到最好的效率，这就是平衡二叉排序树。

平衡二叉排序树(Balanced Binary Tree或Height-Balanced Tree)是在1962年由Adelson-Velskii和Landis提出的，又称AVL树。

平衡二叉树的定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树。

⑴：左子树和右子树深度之差的绝对值不大于1；

⑵：左子树和右子树也都是平衡二叉树。

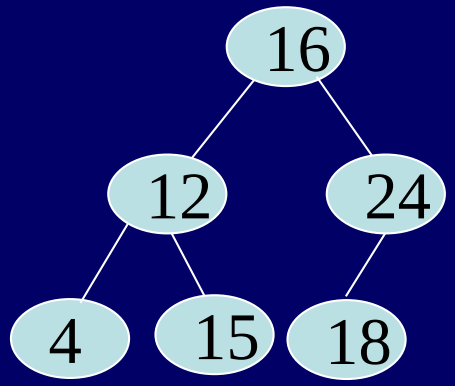
平衡因子(Balance Factor)：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。

因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1， 该二叉树就不是平衡二叉树。

如果一棵二叉树既是二叉排序树又是平衡二叉树，称为平衡二叉排序树(Balanced Binary Sort Tree)。

结点类型定义如下：

```
typedef  struct  BNode
  {  KeyType  key ;    /*  关键字域  */
     int  Bfactor ;    /*  平衡因子域  */
     …      /*  其它数据域  */
     struct  BNode  *Lchild , *Rchild ;
  }BSTNode ;
```



平衡二叉树

在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，则在AVL树上执行查找时，和给定的K值比较的次数不超过树的深度。

设深度为h的平衡二叉排序树所具有的最少结点数为N_h，则由平衡二叉排序树的性质知：

$N_0=0$，$N_1=1$，$N_2=2$，…，$N_h=N_{h-1}+N_{h-2}+1$

该关系和Fibonacci数列相似。根据归纳法可证明，当$h\geq 0$时，$N_h=F_{h+2}-1$，…而

$$F_h\approx\frac{\Phi^h}{\sqrt{5}}\quad 其中\ \Phi=\frac{1+\sqrt{5}}{2}\qquad 则N_h\approx\frac{\Phi^h}{\sqrt{5}}-1$$

这样，含有n个结点的平衡二叉排序树的最大深度为

$$h\approx\log_{\Phi}(\sqrt{5}\times(n+1))-2$$

则在平衡二叉排序树上进行查找的平均查找长度和$\log_2 n$是一个数量级的，平均时间复杂度为$O(\log_2 n)$。

平衡化旋转

一般的二叉排序树是不平衡的，若能通过某种方法使其既保持有序性，又具有平衡性，就找到了构造平衡二叉排序树的方法，该方法称为平衡化旋转。

在对AVL树进行插入或删除一个结点后，通常会影响到从根结点到插入（或删除）结点的路径上的某些结点，这些结点的子树可能发生变化。以插入结点为例，影响有以下几种可能性

- 以某些结点为根的子树的深度发生了变化；
- 某些结点的平衡因子发生了变化；
- 某些结点失去平衡。

沿着插入结点上行到根结点就能找到某些结点，这些结点的平衡因子和子树深度都会发生变化，这样的结点称为失衡结点。

1 LL型平衡化旋转

(1) 失衡原因

在结点a的左孩子的左子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后的平衡因子是2。设b是a的左孩子，b在插入前的平衡因子只能是0，插入后的平衡因子是1(否则b就是失衡结点或者a不会成为失衡节点)。

(2) 平衡化旋转方法

通过顺时针旋转操作实现。

用b取代a的位置，a成为b的右子树的根结点，b原来的右子树作为a的左子树。

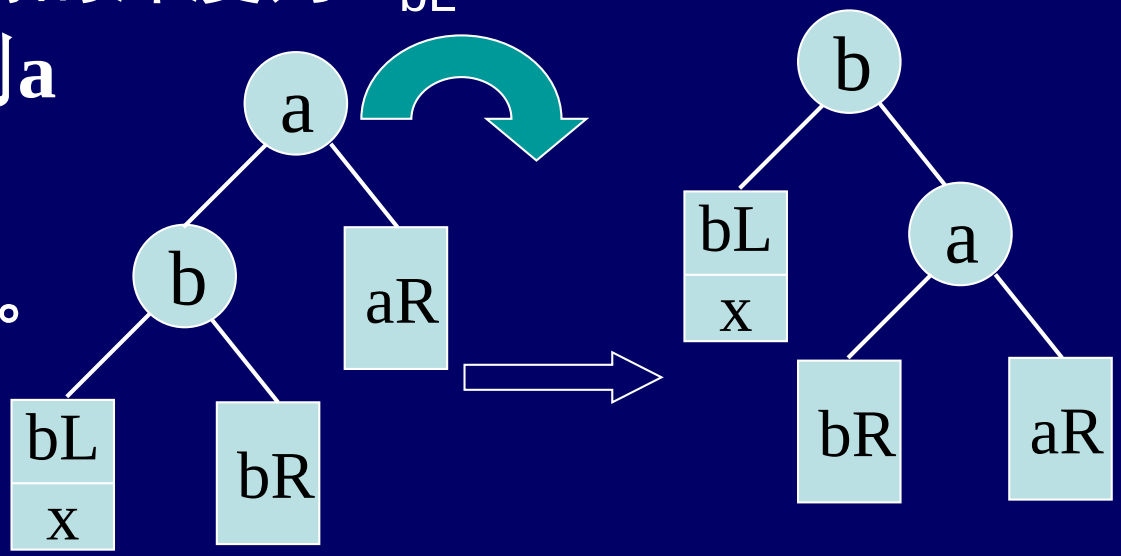
(3) 插入后各结点的平衡因子分析

① 旋转前的平衡因子

设插入后b的左子树的深度为H_{bL}，则其右子树的深度为H_{bL}-1； a的左子树的深度为H_{bL}+1。

a的平衡因子为2，则a的右子树的深度为：

$$H_{aR}=H_{bL}+1-2=H_{bL}-1。$$



LL型平衡化旋转示意图

② 旋转后的平衡因子

a的右子树没有变，而左子树是b的右子树，则平衡因子是：$H_{aL}- H_{aR}=(H_{bL}-1)-(H_{bL}-1)=0$

即a是平衡的，以a为根的子树的深度是H_{bL}。

b的左子树没有变化，右子树是以a为根的子树，则平衡因子是： $H_{bL}-H_{bL}=0$

即b也是平衡的，以b为根的子树的深度是$H_{bL}+1$，与插入前a的子树的深度相同，则该子树的上层各结点的平衡因子没有变化，即整棵树旋转后是平衡的。

(4) 旋转算法

```
BBSTNode * LL_rotate(BBSTNode *a)
   {  BBSTNode *b ;
      b=a->Lchild ; a->Lchild=b->Rchild ;
      b->Rchild=a ;
      a->Bfactor=b->Bfactor=0 ;
      return b;
   }
```

2 LR型平衡化旋转

(1) 失衡原因

在结点a的左孩子的右子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后a的平衡因子是2。设b是a的左孩子，c为b的右孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是-1；c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

①插入后c的平衡因子是1：即在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}-1；b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2。

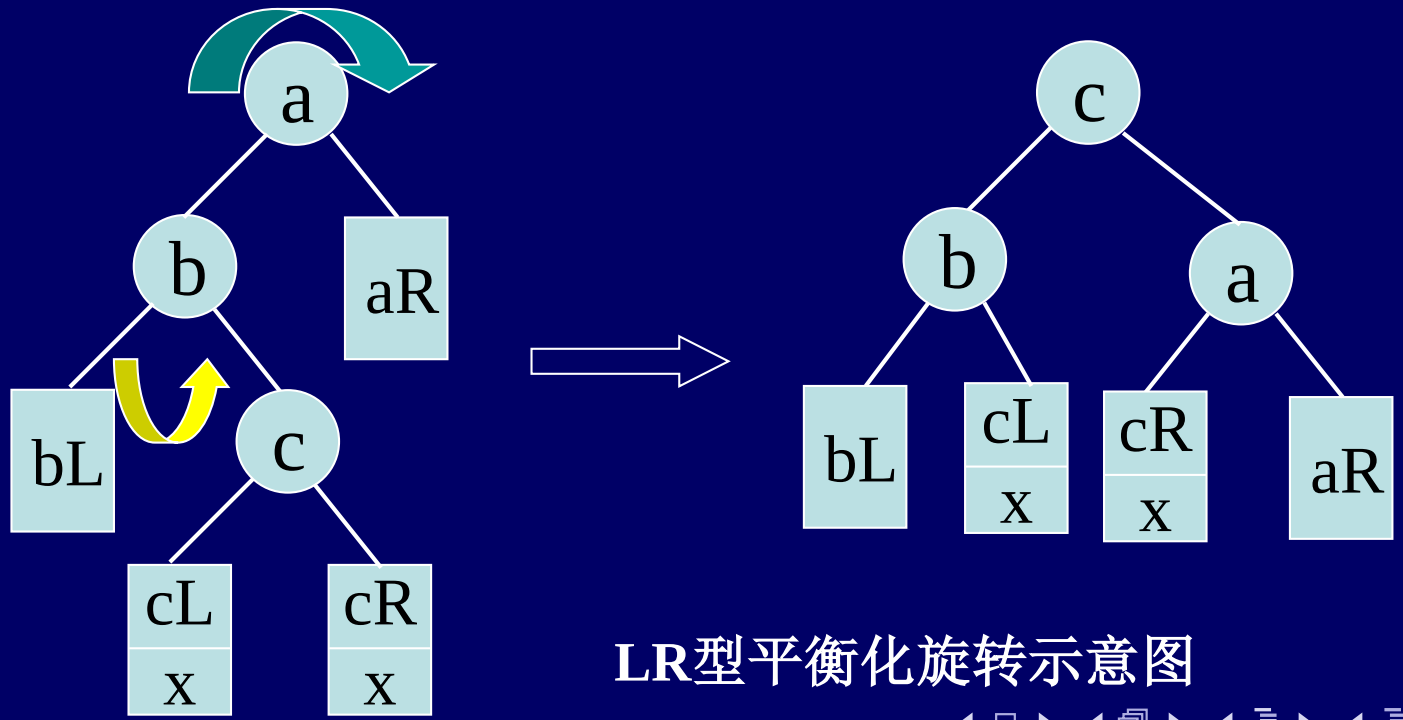
因插入后a的平衡因子是2 ，则a的右子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；插入后a的平衡因子是2 ，则a的右子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：即在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}+1 ，以c为根的子树的深度是H_{cL}+2；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}+1，以b为根的子树的深度是H_{cL}+3；则a的右子树的深度是H_{cL}+1。

(3) 平衡化旋转方法

先以b进行一次逆时针旋转(将以b为根的子树旋转为以c为根)，再以a进行一次顺时针旋转，如图所示。将整棵子树旋转为以c为根，b是c的左子树，a是c的右子树；c的右子树移到a的左子树位置， c的左子树移到b的右子树位置。



LR型平衡化旋转示意图

(4) 旋转后各结点(a,b,c)平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树深度为H_{cL}-1，其右子树没有变化，深度是H_{cL}，则a的平衡因子是-1；b的左子树没有变化，深度为H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则b的平衡因子是0； c的左、右子树分别是以b和a为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是0，1，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
BBSTNode * LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Lchild ; c=b->Rchild ;      /*  初始化  */
     a->Lchild=c->Rchild ;  b->Rchild=c->Lchild ;
     c->Lchild=b ;  c->Rchild=a ;
     if (c->Bfactor==1) {a->Bfactor=-1 ;b->Bfactor=0 ; }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
     else {   a->Bfactor=0 ; b->Bfactor=1 ;   }
     c->Bfactor=0; return c;
  }
```

3 RL型平衡化旋转

(1) 失衡原因

在结点a的右孩子的左子树上进行插入，插入使结点a失去平衡，与LR型正好对称。对于结点a，插入前的平衡因子是-1，插入后a的平衡因子是-2。设b是a的右孩子，c为b的左孩子，b在插入前的平衡因子只能是0，插入后的平衡因子是1；同样，c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}-1。

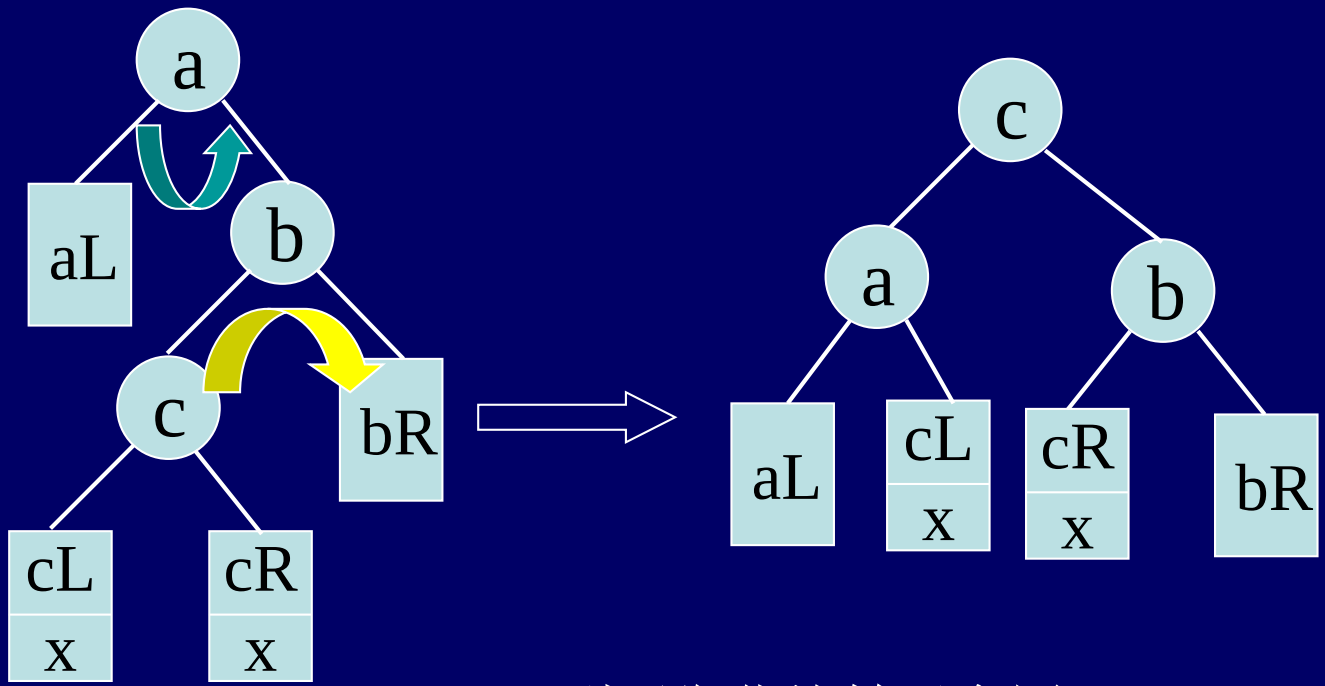
因b插入后的平衡因子是1，则其右子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}+1，以c为根的子树的深度是H_{cL}+2；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}+1，以b为根的子树的深度是H_{cL}+3；则a的右子树的深度是H_{cL}+1。

(3) 平衡化旋转方法

先以b进行一次顺时针旋转，再以a进行一次逆时针旋转，如图所示。即将整棵子树(以a为根)旋转为以c为根，a是c的左子树，b是c的右子树；c的右子树移到b的左子树位置，c的左子树移到a的右子树位置。



RL型平衡化旋转示意图

(4) 旋转后各结点(a, b, c)的平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树没有变化，深度是H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则a的平衡因子是0；b的右子树没有变化，深度为H_{cL}，左子树是c旋转前的右子树，深度为H_{cL}-1，则b的平衡因子是-1； c的左、右子树分别是以a 和b为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是1，0，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
BBSTNode * LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Rchild ; c=b->Lchild ;      /*  初始化  */
     a->Rchild=c->Lchild ;  b->Lchild=c->Rchild ;
     c->Lchild=a ;  c->Rchild=b ;
     if (c->Bfactor==1){ a->Bfactor=0 ; b->Bfactor=-1 ; }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
     else {   a->Bfactor=1 ;b->Bfactor=0 ;   }
     c->Bfactor=0; return c;
  }
```

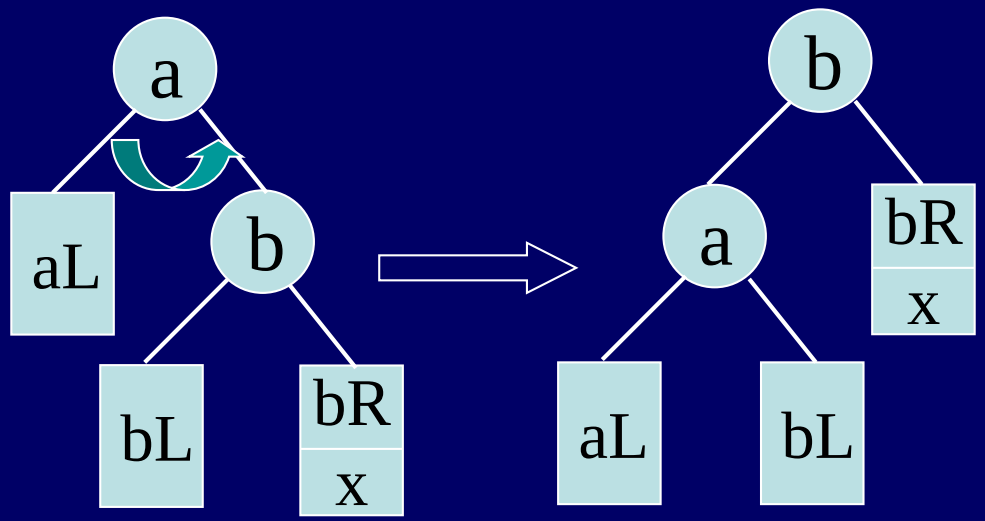
4 RR型平衡化旋转

(1) 失衡原因

在结点a的右孩子的右子树上进行插入，插入使结点a失去平衡。要进行一次逆时针旋转，和LL型平衡化旋转正好对称。

(2) 平衡化旋转方法

设b是a的右孩子，通过逆时针旋转实现，如图所示。用b取代a的位置，a作为b的左子树的根结点，b原来的左子树作为a的右子树。



RR型平衡化旋转示意图

(3) 旋转算法

```
BBSTNode  *RR_rotate(BBSTNode *a)
  {  BBSTNode *b ;
     b=a->Rchild ; a->Rchild=b->Lchild ; b->Lchild=a ;
     a->Bfactor=b->Bfactor=0 ; return b ;
  }
```

对于上述四种平衡化旋转，其正确性容易由“遍历所得中序序列不变”来证明。并且，无论是哪种情况，平衡化旋转处理完成后，形成的新子树仍然是平衡二叉排序树，且其深度和插入前以a为根结点的平衡二叉排序树的深度相同。所以，在平衡二叉排序树上因插入结点而失衡，仅需对失衡子树做平衡化旋转处理。

平衡二叉排序树的插入

平衡二叉排序树的插入操作实际上是在二叉排序插入的基础上完成以下工作：

(1)：判别插入结点后的二叉排序树是否产生不平衡?

(2)：找出失去平衡的最小子树；

(3)：判断旋转类型，然后做相应调整。

失衡的最小子树的根结点a在插入前的平衡因子不为0，且是离插入结点最近的平衡因子不为0的结点的。

1 算法思想(插入结点的步骤)

①：按照二叉排序树的定义，将结点s插入；

②：在查找结点s的插入位置的过程中，记录离结点s最近且平衡因子不为0的结点a，若该结点不存在，则结点a指向根结点；

③： 修改结点a到结点s路径上所有结点的；

④：判断是否产生不平衡，若不平衡，则确定旋转类型并做相应调整。

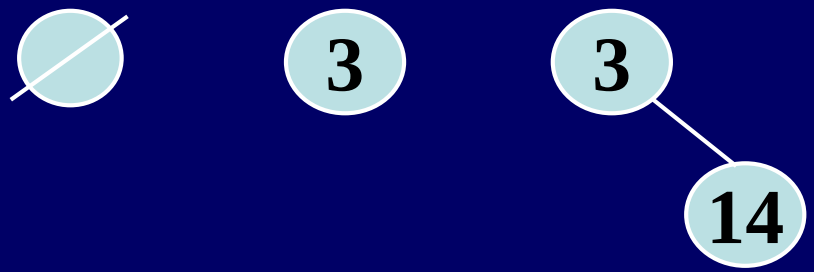
2 算法实现

```
void  Insert_BBST(BBSTNode *T, BBSTNode *S)
   {   BBSTNode *f,*a,*b,*p,*q;
       if  (T==NULL)  {   T=S ; T->Bfactor=0 ; return ;  }
       a=p=T ;     /* a指向离s最近且平衡因子不为0的结点 */
       f=q=NULL ;       /*   f指向a的父结点,q指向p父结点   */
       while (p!=NULL)
          {  if (S->key==p->key)  return ;    /*  结点已存在  */
             if (p->Bfactor!=0)  {  a=p ; f=q ;  }
             q=p ;
             if (S->key < p->key)  p=p->Lchild ;
             else p=p->Rchild ;     /*  在右子树中搜索  */
          }     /*  找插入位置  */
```

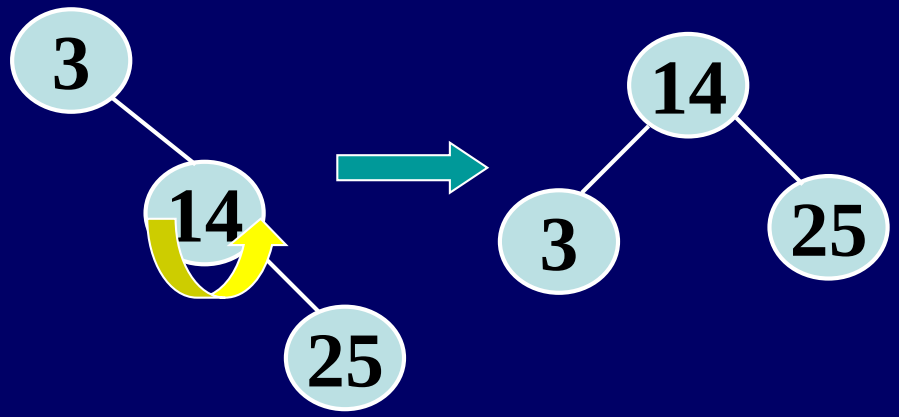
```
if (S->key < q->key)  q->Lchild=S ;/*  s为左孩子  */
else   q->Rchild=S ;      /*  s插入为q的右孩子  */
p=a ;
while (p!=S)
   {   if (S->key < p->key )
          {   p->Bfactor++ ;  p=p->Lchild ;   }
       else {   p->Bfactor-- ; p=p->Rchild ;  }
   }     /*  插入到左子树,平衡因子加1,插入到左子树,减1  */
if (a->Bfactor>-2&& a->Bfactor<2)
   return ;   /*   未失去平衡,不做调整  */
if (a->Bfactor==2)
   {   b=a->Lchild ;
```

```
        if (b->Bfactor==1)    p=LL_rotate(a) ;
        else p=LR_rotate(a) ;
    }
else
    {  b=a->Rchild ;
        if (b->Bfactor==1)    p=RL_rotate(a) ;
        else  p=RR_rotate(a) ;
    }  /*  修改双亲结点指针  */
if (f==NULL)  T=p ;      /*  p为根结点  */
else    if (f->Lchild==a) f->Lchild=p ;
        else  f->Lchild=p ;
}
```

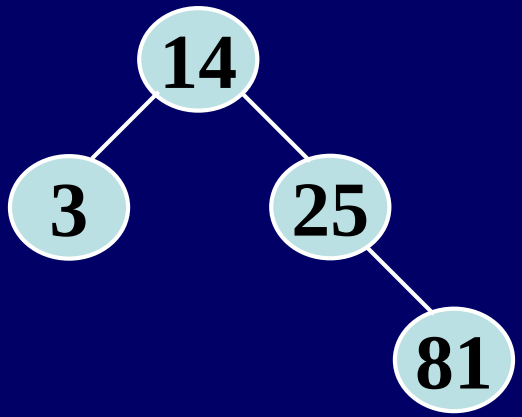
例： 设要构造的平衡二叉树中各结点的值分别是(3, 14, 25, 81, 44)， 平衡二叉树的构造过程如下。



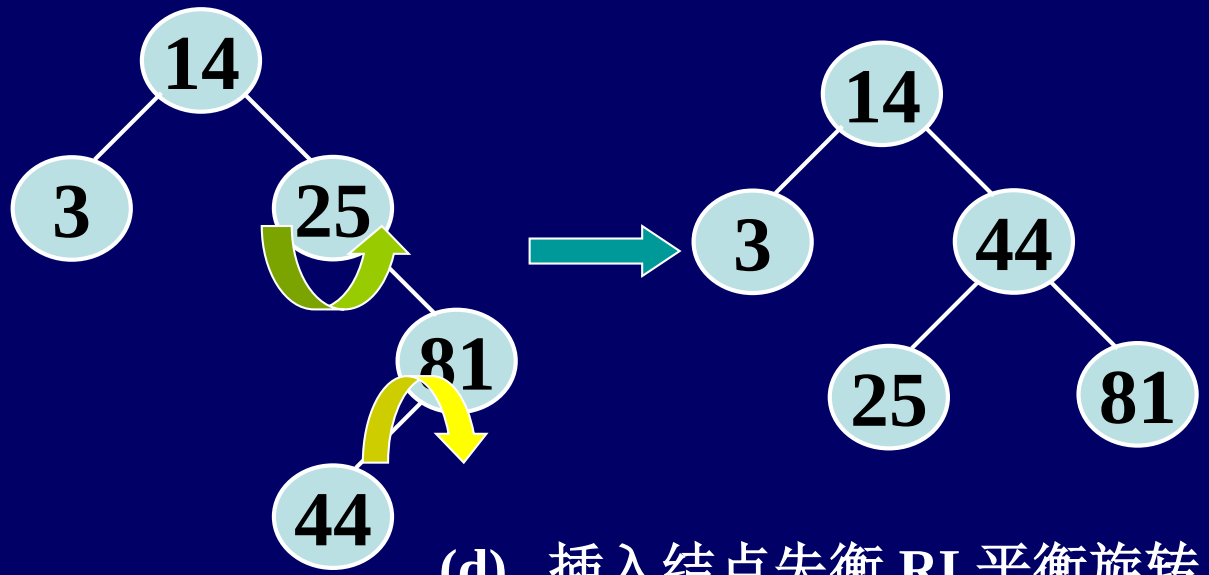
(a) 插入不超过两个结点



(b) 插入新结点失衡,RR平衡旋转



(c) 插入新结点未失衡



(d) 插入结点失衡,RL平衡旋转

平衡二叉树的构造过程

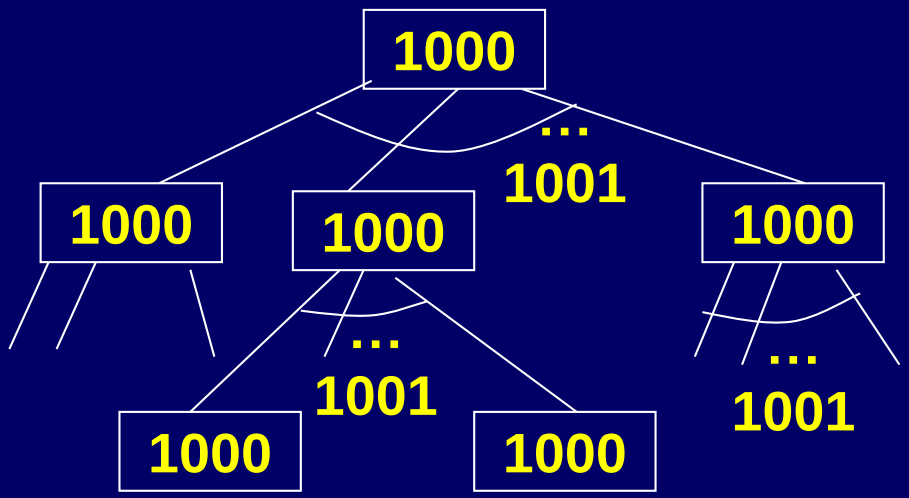
§ 9.3.2 B-树

1.概念

- B-树是一种完全平衡的多阶查找树，主要是作为磁盘文件的索引组织，用于外部查找。
- 基本想法是增大结点来降低树高，减少访问外存的次数

一棵m阶B-树，每个结点最多有m个孩子，m一般为50－2000

例如，1棵1001阶B-树，3层即可包含10亿以上的Keys，当根结点置于内存时，查找任一Key至多只要访问2次外存。



1个结点
1000个关键字

1001个结点
1,001,000个关键字

1,002,001个结点
1,002,001,000个关键字

§ 9.3.2 B-树

- **定义：一棵m(m≥3)阶的B-树是满足下述性质的m叉树：**
  - **性质1：** 每个结点至少包含下列数据域：$(j, P_0, K_1, P_1, K_2, \ldots, K_j, P_j)$

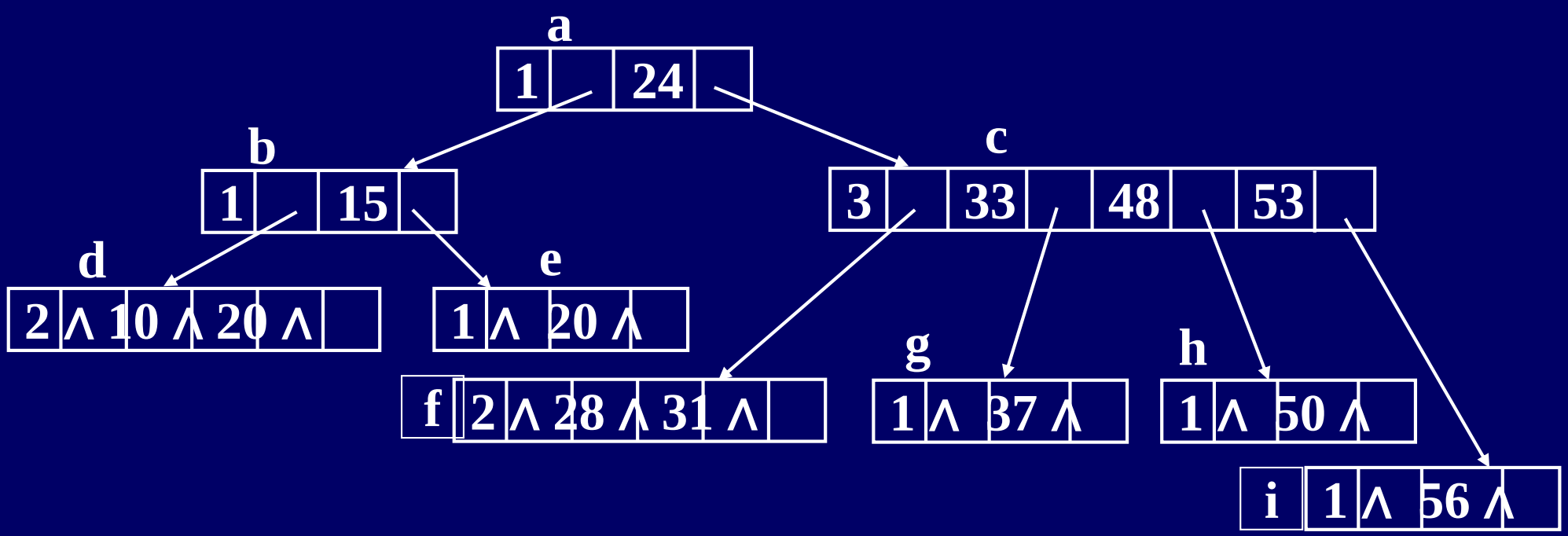
    j：keys总数， K_i：关键字， P_i：指针

    $keys(P_0) < K_1 < keys(P_1) < K_2 < \ldots < K_j < keys(P_j)$
  - **性质2：** 所有叶子在同一层上，叶子层数为树高h，叶子中的指针为空。
  - **性质3：** 每个非根结点中的关键字数目j满足：

$$\lceil m/2 \rceil - 1 \le j \le m - 1$$

    即：每个非根的内部结点的子树数在m和$\lceil m/2 \rceil$之间
  - **性质4：** 非空的B－树中，根至少有1个关键字。

    即：根不是叶子时，子树数为：2～m之间



一棵包含13个关键字的4阶B_树

§ 9.3.2 B-树

运算

- **查找**

  多路查找：先在结点内查找（折半或顺序），后在子结点中查找（读盘）

- **插入和生成**

  平衡机制：满时插入分裂结点，从叶子 ⟹ 根，树高可能长一层

- **删除**

  平衡机制：半满时删除，可能要合并结点，从叶子 ⟹ 根，树高可能减一层

根据m阶B_树的定义，结点的类型定义如下：

```
#define M    5      /*  根据实际需要定义B_树的阶数  */
typedef  struct BTNode
  {  int   keynum ;   /*   结点中关键字的个数   */
     struct BTNode  *parent ;    /*   指向父结点的指针   */
     KeyType  key[M+1] ;     /*   关键字向量,key[0]未用   */
     struct BTNode  *ptr[M+1] ; /*   子树指针向量   */
     RecType   *recptr[M+1] ;
        /*   记录指针向量,recptr[0]未用   */
  }BTNode ;
```

2 B_树的查找

由B_树的定义可知，在其上的查找过程和二叉排序树的查找相似。

(1) 算法思想

① 从树的根结点T开始，在T所指向的结点的关键字向量key[1…keynum]中查找给定值K(用折半查找)：

若key[i]=K(1≤i≤keynum)，则查找成功，返回结点及关键字位置；否则，转(2)；

② 将K与向量key[1…keynum]中的各个分量的值进行比较，以选定查找的子树：

- 若K<key[1]：T=T->ptr[0]；

- 若key[i]<K<key[i+1](i=1, 2, …keynum-1)：
  T=T->ptr[i]；
- 若K>key[keynum]：T=T->ptr[keynum]；

转①，直到T是叶子结点且未找到相等的关键字，则查找失败。

(2) 算法实现

```
int  BT_search(BTNode *T, KeyType K, BTNode *p)
  /*  在B_树中查找关键字K, 查找成功返回在结点中的位置   */
 /*  及结点指针p; 否则返回0及最后一个结点指针    */
 {   BTNode *q ; int n ;
     p=q=T ;
```

```
    while  (q!=NULL)
       {  p=q ; q->key[0]=K ;      /*  设置查找哨兵  */
          for (n=q->keynum ; K<q->key[n] ; n--);
          if (n>0&&EQ(q->key[n], K) )    return n ;
          q=q->ptr[n] ;
       }
    return 0 ;
  }
```

(3) 算法分析

在B_树上的查找有两中基本操作：

- 在B_树上查找结点(查找算法中没有体现)；

◆ 在结点中查找关键字：在磁盘上找到指针ptr所指向的结点后，将结点信息读入内存后再查找。因此，磁盘上的查找次数(待查找的记录关键字在B_树上的层次数)是决定B_树查找效率的首要因素。

根据m阶B_树的定义，第一层上至少有1个结点，第二层上至少有2个结点；除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，…，第h层上至少有$\lceil m/2 \rceil^{h-2}$个结点。在这些结点中：根结点至少包含1个关键字，其它结点至少包含$\lceil m/2 \rceil$-1个关键字，设s=$\lceil m/2 \rceil$，则总的关键字数目n满足：

$$n \geqq 1+(s-1)\sum_{i=2}^{h} 2s^{i-2}=1+\ 2(s-1)\frac{s^{h-1}-1}{s-1}=2s^{h-1}-1$$

因此有： $h \leqslant 1+\log_s((n+1)/2)=1+\log_{\lceil m/2 \rceil}((n+1)/2)$

即在含有n个关键字的B_树上进行查找时，从根结点到待查找记录关键字的结点的路径上所涉及的结点数不超过$1+\log_{\lceil m/2 \rceil}((n+1)/2)$ 。

3 B_树的插入

B_树的生成也是从空树起，逐个插入关键字。插入时不是每插入一个关键字就添加一个叶子结点，而是首先在最低层的某个叶子结点中添加一个关键字，然后有可能“分裂”。

(1) 插入思想

① 在B_树的中查找关键字K，若找到，表明关键字已存在，返回；否则，K的查找操作失败于某个叶子结点，转 ②；

② 将K插入到该叶子结点中，插入时，若：

- 叶子结点的关键字数<m-1：直接插入；
- 叶子结点的关键字数=m-1：将结点“分裂”。

(2) 结点“分裂”方法

设待“分裂”结点包含信息为：

$(m, A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m)$，从其中间位置分为两个结点：

$(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

$(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \ldots, K_m, A_m)$

并将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点中，以分裂后的两个结点作为中间关键字$K_{\lceil m/2 \rceil}$的两个子结点。

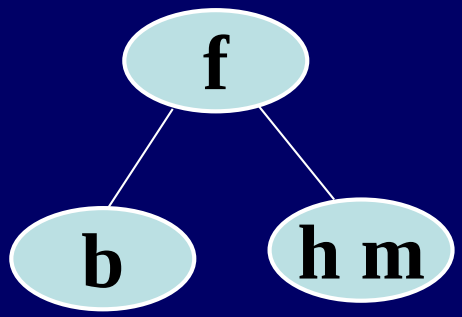
当将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点后，父结点也可能不满足m阶B_树的要求(分枝数大于m)，则必须对父结点进行“分裂”，一直进行下去，直到没有父结点或分裂后的父结点满足m阶B_树的要求。

当根结点分裂时，因没有父结点，则建立一个新的根，B_树增高一层。

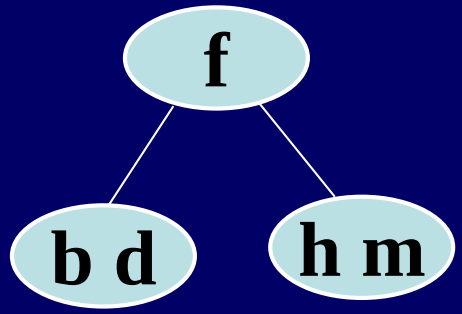
例（下页）：在一个3阶B_树(2-3树)上插入结点的过程。

(3) 算法实现

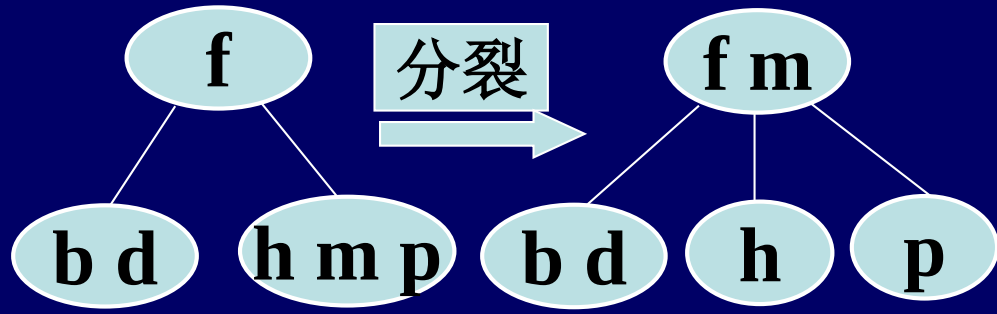
要实现插入，首先必须考虑结点的分裂。设待分裂的结点是p，分裂时先开辟一个新结点，依此将结点p中后半部分的关键字和指针移到新开辟的结点中。分裂之后，而需要插入到父结点中的关键字在p的关键字向量的p->keynum+1位置上。



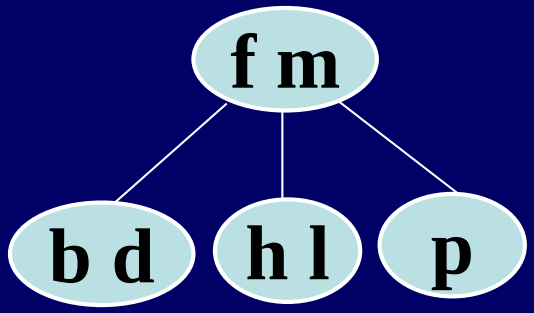
(a) 一棵2-3树



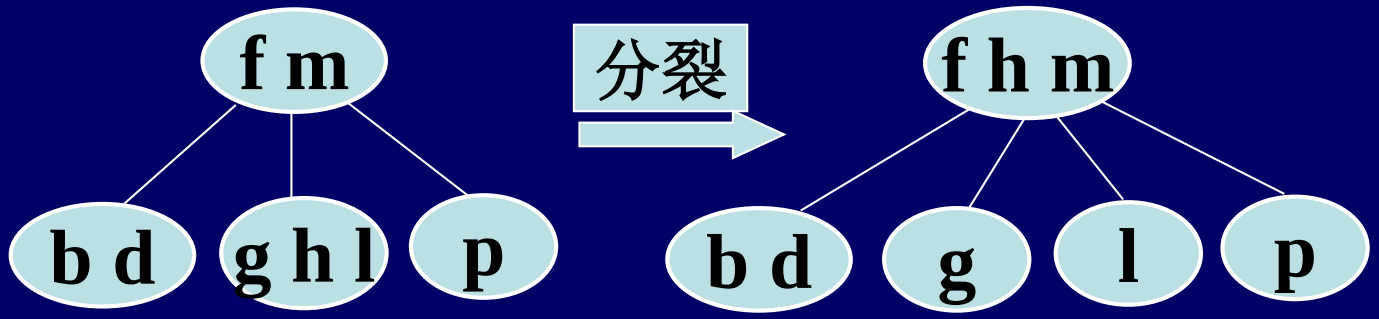
(b) 插入d后



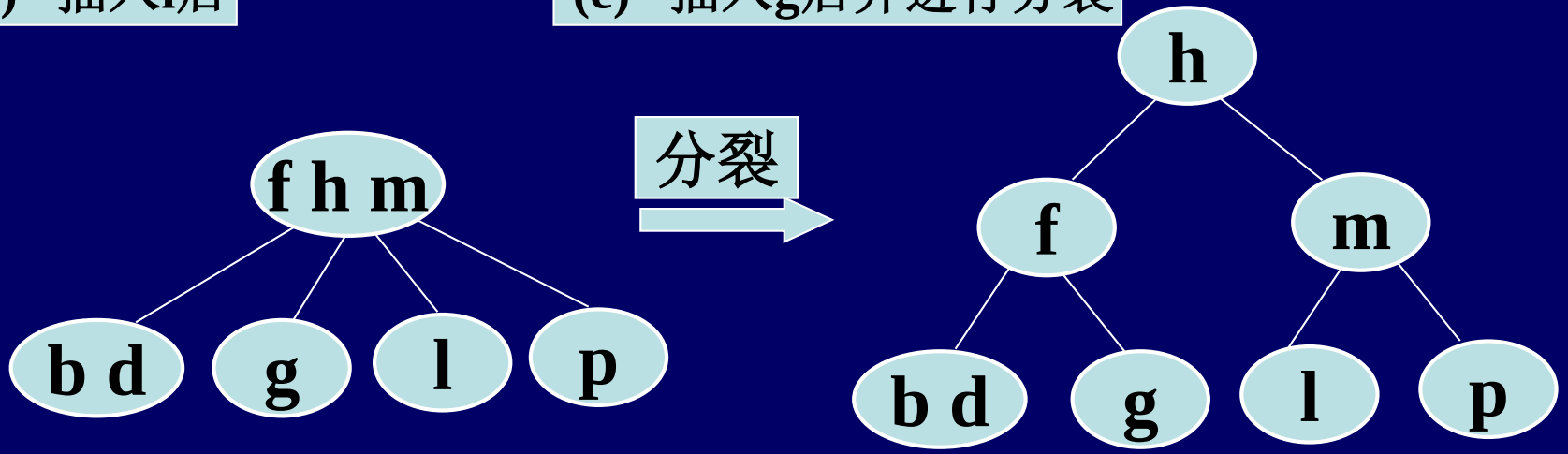
(c) 插入p后并进行分裂



(d) 插入l后



(e) 插入g后并进行分裂



(f) 继续进行分裂

在B_树中进行插入的过程

```
BTNode  *split(BTNode *p)
  /*  结点p中包含m个关键字，从中分裂出一个新的结点  */
  {  BTNode *q ;  int k, mid, j ;
     q=(BTNode *)malloc(sizeof( BTNode)) ;
     mid=(m+1)/2 ;   q->ptr[0]=p->ptr[mid] ;
     for (j=1,k=mid+1; k<=m; k++)
        {   q->key[j]=p->key[k] ;
            q->ptr[j++]=p->ptr[k] ;
        }  /*  将p的后半部分移到新结点q中  */
     q->keynum=m-mid ;  p->keynum=mid-1 ;
     return(q) ;
  }
```

```
void  insert_BTree(BTNode *T, KeyType  K)
    /*   在B_树T中插入关键字K，*/
   {   BTNode *q, *s1=NULL, *s2=NULL ;
       int n ;
       if  (!BT_search(T, K, p))     /*  树中不存在关键字K  */
          {   while (p!=NULL)
              {  p->key[0]=K ;     /*   设置哨兵   */
                 for (n=p->keynum ; K<p->key[n] ; n--)
                    {   p->key[n+1]=p->key[n] ;
                        p->ptr[n+1]=p->ptr[n] ;
                    }    /*   后移关键字和指针   */
                 p->key[n+1]=K ; p->ptr[n]=s1 ;
```

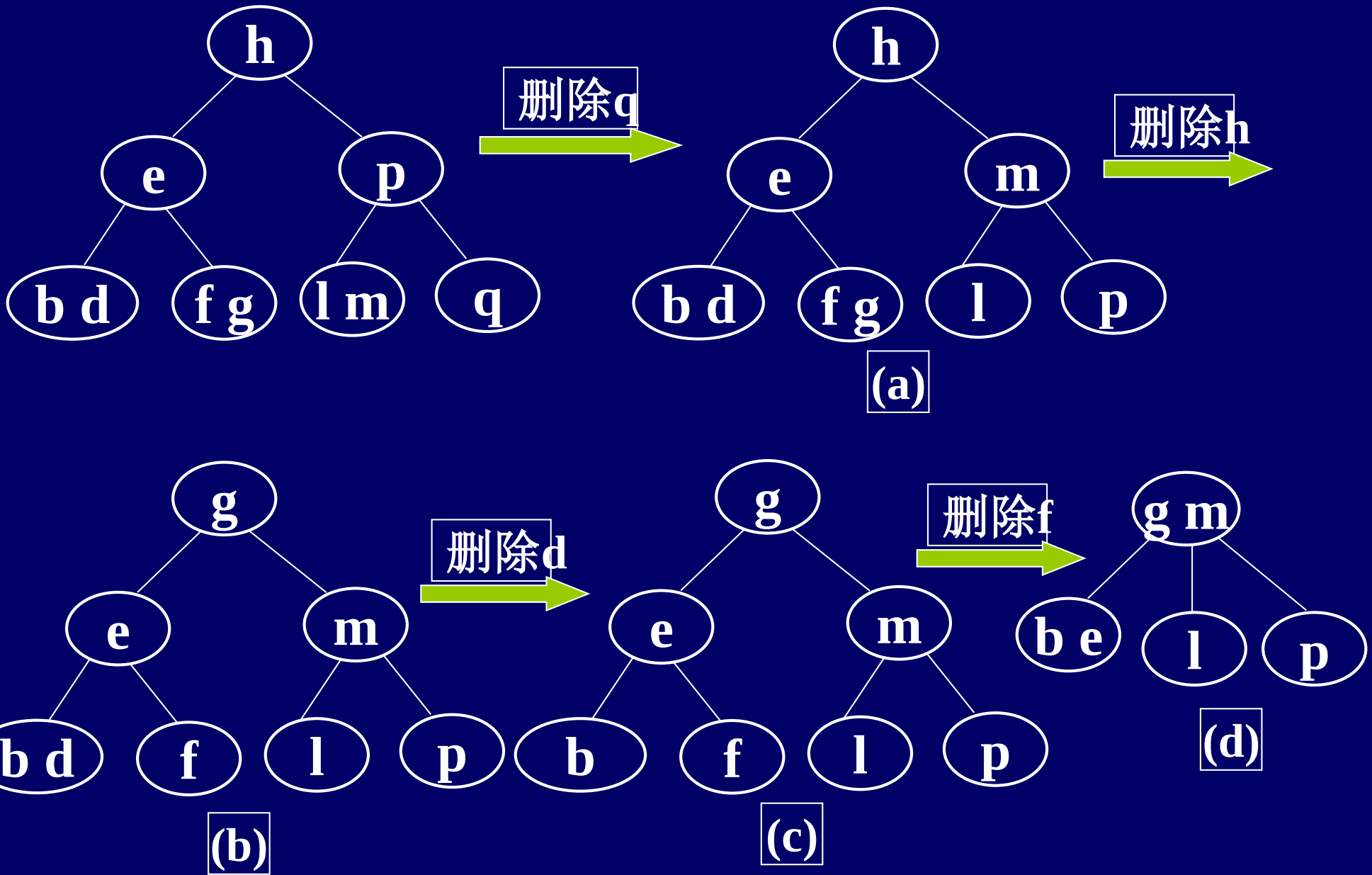
```
            p->ptr[n+1]=s2 ;   /*  置关键字K的左右指针  */
        if (++(p->keynum )<m)  break ;
        else {  s2=split(p) ; s1=p ;  /*  分裂结点p  */
                K=p->key[p->keynum+1] ;
                p=p->parent ;    /*  取出父结点*/
              }
    }
        if (p==NULL)       /*  需要产生新的根结点  */
          {  p=(BTNode*)malloc(sizeof( BTNode)) ;
             p->keynum=1 ; p->key[1]=K ;
             p->ptr[0]=s1 ; p->ptr[1] =s2 ;
           }
} }
```

利用m阶B_树的插入操作，可从空树起，将一组关键字依次插入到m阶B_树中，从而生成一个m阶B_树。

4 B_树的删除

在B_树上删除一个关键字K，首先找到关键字所在的结点N，然后在N中进行关键字K的删除操作。

若N不是叶子结点，设K是N中的第i个关键字，则将指针A_{i-1}所指子树中的最大关键字(或最小关键字)K'放在(K)的位置，然后删除K'，而K'一定在叶子结点上。如图，删除关键字h，用关键字g代替h的位置，然后再从叶子结点中删除关键字g。



在B_树中进行删除的过程

从叶子结点中删除一个关键字的情况是：

⑴ 若结点N中的关键字个数>⌈m/2⌉-1：在结点中直接删除关键字K，如图 (b)~©所示。

⑵ 若结点N中的关键字个数=⌈m/2⌉-1：若结点N的左(右)兄弟结点中的关键字个数>⌈m/2⌉-1，则将结点N的左(或右)兄弟结点中的最大(或最小)关键字上移到其父结点中，而父结点中大于(或小于)且紧靠上移关键字的关键字下移到结点N，如图 (a)。

⑶ 若结点N和其兄弟结点中的关键字数=⌈m/2⌉-1：删除结点N中的关键字，再将结点N中的关键字、指针与其兄弟结点以及分割二者的父结点中的某个关键字K_i，合并为一个结点，若因此使父结点中的关键字个数<⌈m/2⌉-1，则依此类推，如图 (d)。

算法实现

在B_树上删除一个关键字的操作，针对上述的(2)和(3)的情况，相应的算法如下：

```
int  BTNode  MoveKey(BTNode *p)
  /*  将p的左(或右)兄弟结点中的最大(或最小)关键字上移  */
  /*   到其父结点中,父结点中的关键字下移到p中  */
 {  BTNode *b , *f=p->parent ;   /*  f指向p的父结点  */
    int k, j ;
    for (j=0; f->ptr[j]!=p; j++) ;  /*  在f中找p的位置  */
    if (j>0)        /*  若p有左邻兄弟结点  */
        {  b=f->ptr[j-1] ;     /*  b指向p的左邻兄弟  */
```

```
if (b->keynum>(m-1)/2)
        /*   左邻兄弟有多余关键字  */
    {  for (k=p->keynum; k>=0; k--)
          {   p->key[k+1]=p->key[k];
              p->ptr[k+1]=p->ptr[k];
           }      /*   将p中关键字和指针后移   */
       p->key[1]=f->key[j];
       f->key[j]=b->key[keynum] ;
     /*  f中关键字下移到p, b中最大关键字上移到f  */
       p->ptr[0]= b->ptr[keynum] ;
       p->keynum++ ;
       b->keynum-- ;
```

```
        return(1) ;
    }
  if (j<f->keynum)     /*   若p有右邻兄弟结点   */
    {  b=f->ptr[j+1] ;      /*   b指向p的右邻兄弟   */
       if (b->keynum>(m-1)/2)
            /*   右邻兄弟有多余关键字  */
          {  p->key[p->keynum]=f->key[j+1] ;
             f->key[j+1]=b->key[1];
             p->ptr[p->keynum]=b->ptr[0];
   /*  f中关键字下移到p, b中最小关键字上移到f  */
             for (k=0; k<b->keynum; k++)
```

```
                    {   b->key[k]=b->key[k+1];
                        b->ptr[k]=b->ptr[k+1];
                     }      /*  将b中关键字和指针前移  */
                  p->keynum++ ;
                  b->keynum-- ;
                  return(1) ;
                }
            }
          return(0);
      }   /*  左右兄弟中无多余关键字,移动失败  */
}
```

```
BTNode  *MergeNode(BTNode *p)
   /*   将p与其左(右)邻兄弟合并,返回合并后的结点指针  */
  {   BTNode *b, f=p->parent ;
      int j, k ;
      for (j=0; f->ptr[j]!=p; j++);   /*   在f中找出p的位置   */
      if (j>0) b=f->ptr[j-1];    /*   b指向p的左邻兄弟   */
      else {  b=p; p=f->ptr[j+1];  }    /*   p指向p的右邻   */
      b->key[++b->keynum]=f->key[j] ;
      b->ptr[b->keynum]=p->ptr[0] ;
      for (k=1; k<=p->keynum ; k++)
         {   b->key[++b->keynum]=p->key[k] ;
             b->ptr[b->keynum]=p->ptr[k] ;
         }     /*   将p中关键字和指针移到b中   */
```

```
    free(p);
    for (k=j+1; k<=f->keynum ; k++)
        {   f->key[k-1]=f->key[k] ;
            f->ptr[k-1]=f->ptr[k] ;
        }     /*   将f中第j个关键字和指针前移   */
    f->keynum-- ;
    return(b) ;
}
```

```
void  DeleteBTNode(BTNode *T, KeyType K)
  {  BTNode  *p, *S ;
     int j,n ;
     j=BT_search(T, K, p) ; /* 在T中查找K的结点  */
     if (j==0)  return(T) ;
     if (p->ptr[j-1])
        {  S=p->ptr[j-1] ;
           while (S->ptr[S->keynum])
              S=S->ptr[S->keynum] ;
            /*  在子树中找包含最大关键字的结点   */
           p->key[j]=S->key[S ->keynum] ;
           p=S ; j=S->keynum ;
        }
```

```
    for (n=j+1; n<p->keynum; n++)
        p->key[n-1]=p->key[n] ;
            /*  从p中删除第m个关键字   */
    p->keynum-- ;
    while (p->keynum<(m-1)/2&&p->parent)
        {   if (!MoveKey(p) )  p=MergeNode(p);
            p=p->parent ;
        }    /*  若p中关键字树目不够,按(2)处理     */
     if (p==T&&T->keynum==0)
        {   T=T->ptr[0] ;   free(p) ;    }
}
```

5 B^+树

在实际的文件系统中，基本上不使用B_树，而是使用B_树的一种变体，称为m阶B^+树。 它与B_树的主要不同是叶子结点中存储记录。在B^+树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B^+树与m阶B_树的主要差异是：

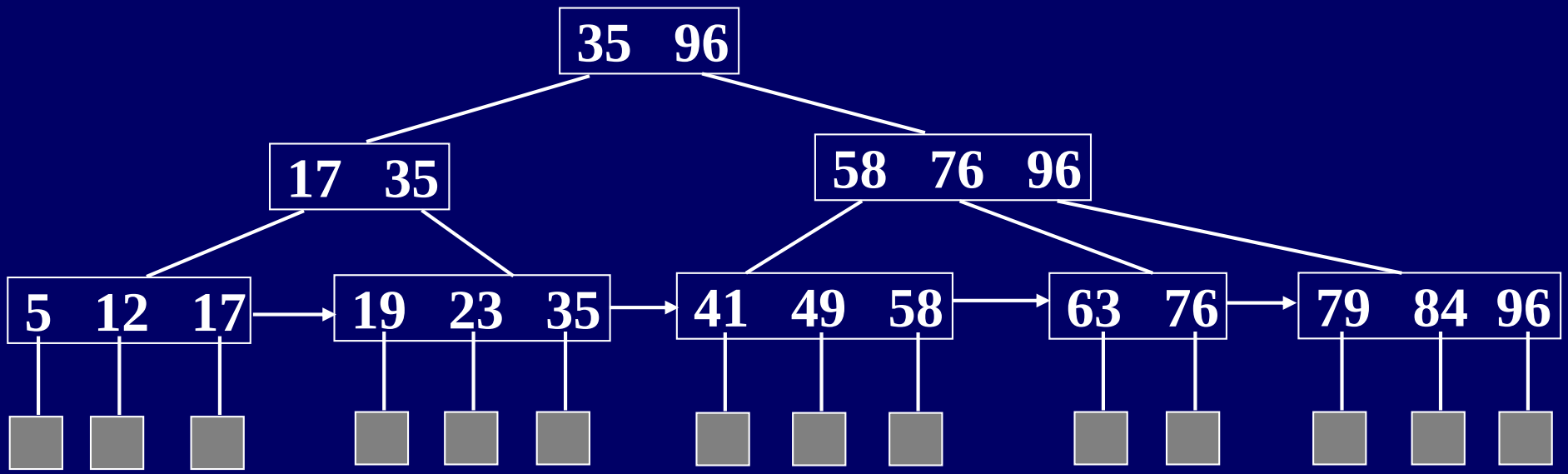
⑴ 若一个结点有n棵子树，则必含有n个关键字；

⑵ 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；

(3) 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

如图所示是一棵3阶B+树。

由于B+树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：



一棵3阶B+树

```
typedef  enum{branch, left}  NodeTag ;
typedef  struct BPNode
   {   NodeTag   tag ;     /*  结点标志  */
       int   keynum ;   /*  结点中关键字的个数  */
       struct BTNode  *parent ;    /*  指向父结点的指针  */
       KeyType  key[M+1] ;     /*  组关键字向量,key[0]未用
       */
       union pointer
          {   struct BTNode  *ptr[M+1] ; /*  子树指针向量  */
              RecType   *recptr[M+1] ;  /*  recptr[0]未用  */
          }ptrType ;    /*  用联合体定义子树指针和记录指针  */
   }BPNode ;
```

与B_树相比，对B^+树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。在B^+树上进行随机查找、插入、删除的过程基本上和B_树类似。

在B^+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录)，即无论查找成功与否，都走了一条从根结点到叶子结点的路径。

B^+树的插入仅仅在叶子结点上进行。当叶子结点中的关键字个数大于m时，“分裂”为两个结点，两个结点中所含有的关键字个数分别是$\lfloor (m+1)/2 \rfloor$和$\lceil (m+1)/2 \rceil$，且将这两个结点中的最大关键字提升到父结点中，用来替代原结点在父结点中所对应的关键字。提升后父结点又可能会分裂，依次类推。

§ 9.4 散列技术

上述查找均是基于key的比较，由判定树可证明，在平均和最坏情况下所需的比较次数的下界是：

$$\lg n + O(1)$$

要突破此界，就不能只依赖于比较来进行。

§ 9.4.1 散列表的概念

- **直接寻址**：当结点的key和存储位置间建立某种关系时，无须比较即可找到结点的存储位置。

例如，若500个结点的keys取值为1～500间互不相同的整数，则keys可作为下标，将其存储到T[1..500]之中，寻址时间为O (1)。

§ 9.4.1 散列表的概念

- 压缩映象：
  - 例子：设影像出租店

    总片数：10,000张

    借还：500人次/每天

    记录：（电话号码，姓名，住址，…），Key为7位电话号码

    直接寻址：须保留1000万个记录

    但实际只要大于500即可，最多1万个记录
  - 一般情况

    全集U：可能发生的关键字集合，即关键字的取值集合

    实际发生集K：实际需要存储的关键字集合

    ∵ |K| << |U|，当表T的规模取|U|浪费空间，有时甚至无法实现

    ∴ T的规模一般取O(K)，但压缩存储后失去了直接寻址的功能

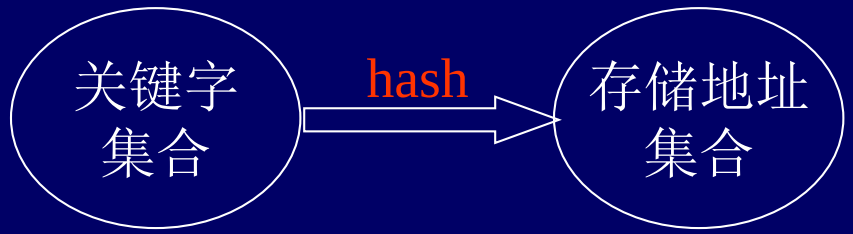
§ 9.4.1 散列表的概念

- 散列（hash）函数h

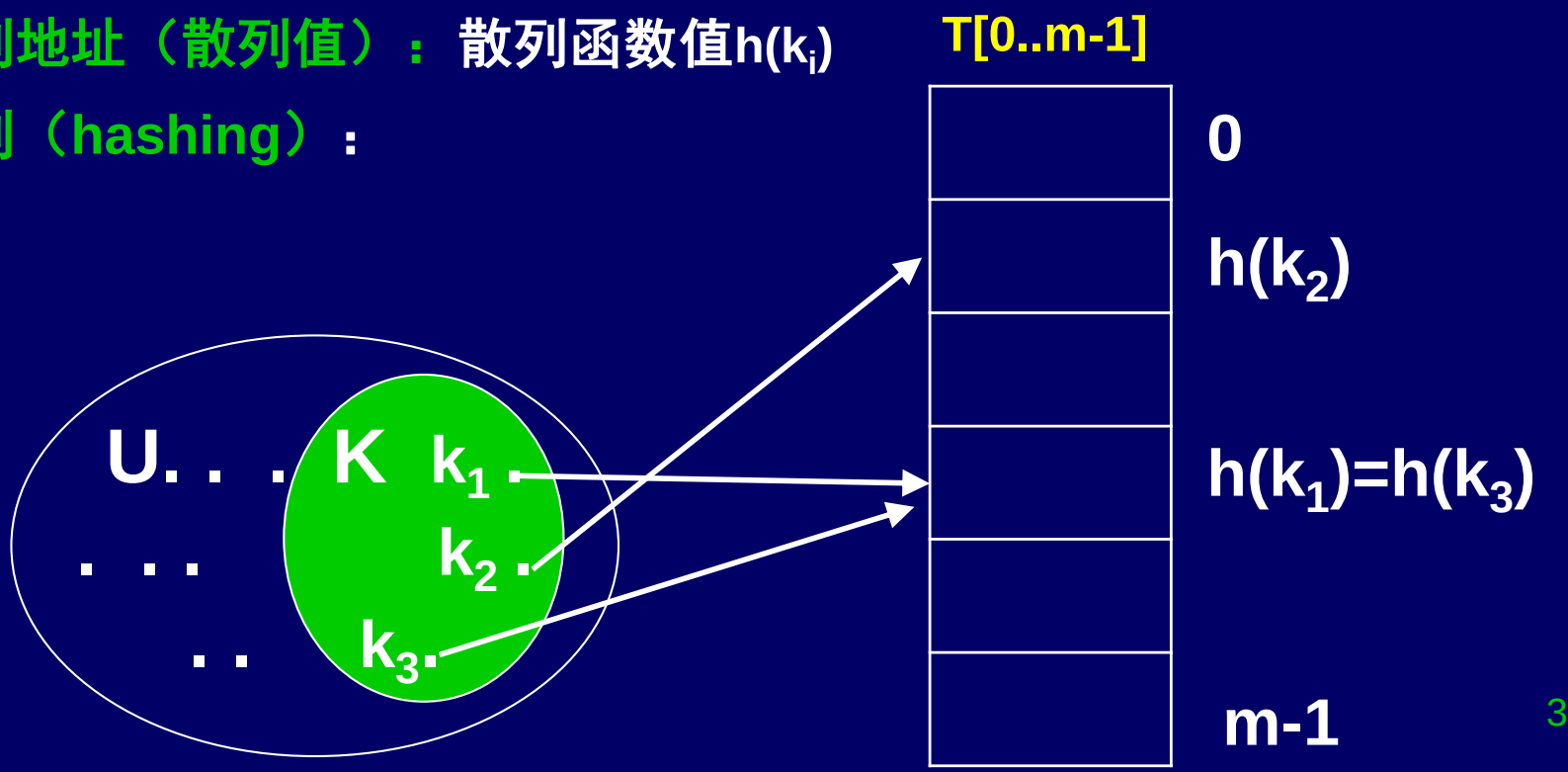
在keys和表地址之间建立一个对应关系h，将全集U映射到T[0..m-1]的下标集上：

h：U→{ 0, 1, … , m-1 }

$\forall k_i \in U$，$h(k_i) \in \{ 0,1,\ldots,m-1 \}$



- 散列表: T
- 散列地址（散列值）：散列函数值$h(k_i)$
- 散列（hashing）：



§ 9.4.1 散列表的概念

- 散列（hash）函数h
  - ❖冲突(碰撞)：若$k_i \neq k_j$，有$h(k_i) = h(k_j)$
  - ❖同义词：发生冲突的两个关键字互为同义词(相对于h)
  - ❖能完全避免冲突吗？

    须满足：① $|U| \leq m$ ② 选择合适的hash函数

    否则只能设计好的h使冲突尽可能少
  - ❖解决冲突： 须确定处理冲突的方法
  - ❖装填因子：冲突的频繁程度除了与h的好坏相关，还与表的填满程度相关

    $\alpha = n/m, \quad 0 \leq \alpha \leq 1$

§ 9.4.2 散列函数的构造方法

- 原则：简单、均匀

  简单——计算快速，均匀——冲突最小化

- 假定：关键字定义在自然数集合上，否则将其转换到自然数集合上

1、平方取中法

先通过平方扩大相近数的差别，然后取中间的若干位作为散列地址。因为中间位和乘数的每一位相关，故产生的地址较为均匀。

```
int Hash( int k ) {
        k *=k;    k /=100;  //去掉末尾两位数
        return k%1000;  //取中间3位，T的地址为0~999
}
```

§ 9.4.2 散列函数的构造方法

2、除余法

用表长除以关键字，取其余数作为散列地址

h(k)＝ k%p； p≤m，有时取p＝m

p最好取素数，或取不包含小于20的质因数的合数

优点：最简单，无需定义为函数，直接写到程序里使用

3、随机数法

h(k)＝random(k);

伪随机函数须保证函数值在0～m－1之间

- 选取哈希函数，考虑的因素
  - 计算哈希函数所需时间
  - 关键字长度
  - 哈希表长度（哈希地址范围）
  - 关键字分布情况
  - 记录的查找频率

§ 9.4.3 处理冲突的方法

1、开放地址法（闭散列）

- **基本思想：**当发生冲突时，使用某种探测技术在散列表中形成一个**探测序列**，沿此序列逐个单元查找，直到找到给定的key，或碰到1个**开放地址**（空单元）为止
  - ❖插入时，探测到开地址可将新结点插入其中
  - ❖查找时，探测到开地址表示查找失败
  - ❖开地址表示：与应用相关，如key为串时，用空串表示开地址
- **一般形式**

$h_i = (h(k) + d_i) \% m \qquad 1 \le i \le m-1$ （9.1）

探测序列：$h(k), h_1, h_2, \ldots, h_{m-1}$

开地址法要求：$\alpha < 1$， 实用中 $0.5 \le \alpha \le 0.9$

§ 9.4.3 处理冲突的方法

- **开放地址法分类：**按照形成探测序列的方法不同，可将其分为3类：线性探测、二次探测、双重散列法

（1）线性探测法

- **基本思想：**将T[0..m-1]看作一循环向量，令$d_i=i$，即

$$h_i=(h(k)+i)\%m \qquad 0\le i\le m-1$$

若令d＝h(k)，则最长的探测序列为：

d,d+1,…,m-1,0,1,…,d-1

探测过程终止于

- 探测到空单元：查找时失败，插入时写入
- 探测到含有k的单元：查找时成功，插入时失败
- 探测到T[d-1] 但未出现上述2种情况：查找、插入均失败

§ 9.4.3 处理冲突的方法

- **例1：** 已知一组keys为（26,36,41,38,44,15,68,12,06,51），用除余法和线性探测法构造散列表

解： ∵ $\alpha<1$，n＝10，不妨取m＝13，此时 $\alpha\approx0.77$

h(k)= k%13　　keys：(26,36,41,38,44,15,68,12,06,51)

对应的散列地址：(0,10, 2, 12, 5, 2, 3, 12, 6, 12)

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26		41			44					36		38
	26	12	41	15	68	44	06	51			36		38
探测次数	1	3	1	2	2	1	1	9			1		1

§ 9.4.3 处理冲突的方法

- **非同义词冲突**：上例中，h(15)=2，h(68)=3，15和68不是同义词，但是15和41这两个同义词处理过程中，15先占用了T[3]，导致插入68时发生了非同义词的冲突。
- **聚集（堆积）**：一般地，用线性探测法解决冲突时，当表中i,i+1,…,i+k的位置上已有结点时，一个散列地址为i,i+1,…,i+k,i+k+1的结点都将争夺同一地址：i＋k＋1。这种不同地址的结点争夺同一后继地址的现象称为**聚集**。

聚集增加了探测序列的长度，使查找时间增加。应**跳跃式**地散列。

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26		41			44					36		38
	26	12	41	15	68	44	06	51			36		38
探测次数	1	3	1	2	2	1	1	9			1		1

§ 9.4.3 处理冲突的方法

（2）二次探测法：

探测序列为(增量序列为$d_i = i^2$)：

h_i＝（h(k)＋i*i）％m　　0 ≤i ≤ m－1

缺点：不易探测到整个空间

（3）双重散列法：

是开地址法中最好的方法之一，探测序列为

h_i＝（h(k)＋i* h_1(k) ）％m　　0 ≤i ≤ m－1

即： $d_i = i * h_1(k)$

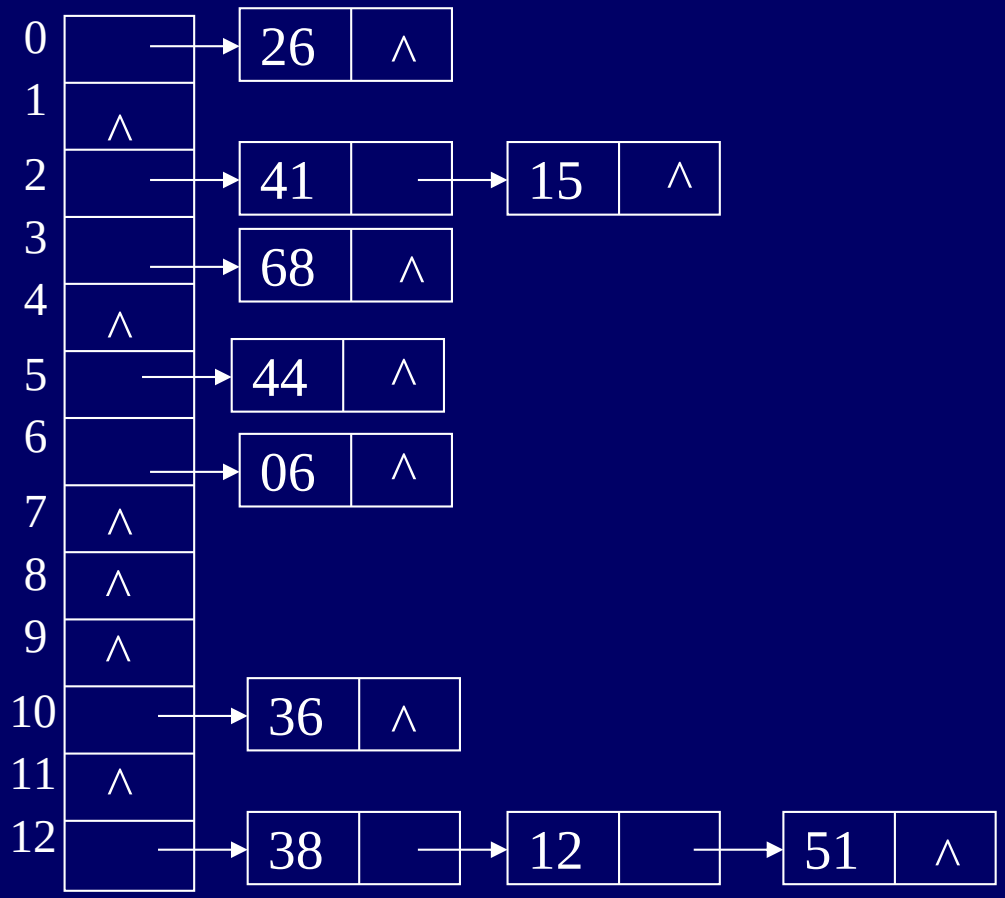
特点：使用了两个散列函数，一般须使h_1(k)的值与m互素(否则可能造成同义词地址的循环计算)

§ 9.4.3 处理冲突的方法

2、拉链法（开散列）

- **基本思想：**将所有keys为同义词的结点连接在同一个单链表中，若散列地址空间为0～m-1，则将表定义为m个头指针组成的指针数组T[0..m-1]，其初值为空。

例2：keys和hash函数同前例



比较次数：　1　　2　　3

§ 9.4.3 处理冲突的方法

- 特点
  - 无堆积现象、非同义词不会冲突、ASL较短
  - 无法预先确定表长度时，可选此方法
  - 删除操作易于实现，开地址方法只能做删除标记
  - 允许$\alpha>1$，当n较大时，节省空间

§ 9.4.4 散列表上的运算

仅给出开放定址法处理冲突时的相关运算

- 存储结构

```
#define NIL -1  //空结点标记
#define m  997 //表长，依赖应用和α
typedef struct {
    KeyType key;
    //….
}NodeType, HashTable[m];
```

1、查找运算

和建表类似，使用的函数和处理冲突的方法应与建表一致

- 开地址可统一处理

§ 9.4.4 散列表上的运算

```
int Hash( KeyType K, int i){
    return ( h(K)+ Increment( i ) )%m; //Increment相当于d_i
}
Int HashSearch( HashTable T, KeyType K, int *pos){
    int i=0; //探测次数计数器
    do {  *pos=Hash( K,i ); //求探测地址h_i
          if ( T[*pos].key==K ) return 1; //成功
          if ( T[*pos].key==NIL ) return 0; //失败
    } while (++i<m); //最多探测m次
    return -1; //可能是表满，失败
}
```

§ 9.4.4 散列表上的运算

2、插入和建表

- **插入：**先查找，找到开地址成功，否则(表满、K存在)失败

```
void HashInsert( HashTable T, NodeType new ){
    int pos, sign;
    sign=HashSearch( T, new.key, &pos ) ;
    if ( !sign ) //标记为0，是开地址
            T[pos]=new; //插入成功
    else
            if ( sign>0 ) printf("duplicate key") ; //重复，插入失败
            else  Error("overflow"); //表满，插入失败
}
```

- **建表：**将表中keys清空，然后依次调用插入算法插入结点

§ 9.4.4 散列表上的运算

3、删除

- 开地址法不能真正删除（置空），而应该置特定标记Deleted
  - 查找时：探测到Deleted标记时，继续探测
  - 插入时：探测到Deleted标记时，将其看作开地址，插入新结点
- 当有删除操作时，一般不用开地址法，而用拉链法

4、性能分析

只分析查找时间，插删取决于查找，因为冲突，仍需比较。

- 例子（见例1和例2图）
  - 成功

线性探测：ASL＝(1*6+2*2+3*1+9*1)/10=2.2 //n=10

拉链法： ASL＝(1*7+2*2+3*1)/10=1.4 //n=10

§ 9.4.4 散列表上的运算

❖ 不成功: 查找不成功时对关键字需要执行的平均比较次数

线性探测：直到探测到开地址为止

若h(K)=0，则必须依次将T[0..8]中的关键字和K比较

ASL＝(9+8+7+6+5+4+3+2+1+1+2+1+10)/13=4.54 //m=13

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26	12	41	15	68	44	06	51			36		38
探测次数	9	8	7	6	5	4	3	2	1	1	2	1	10

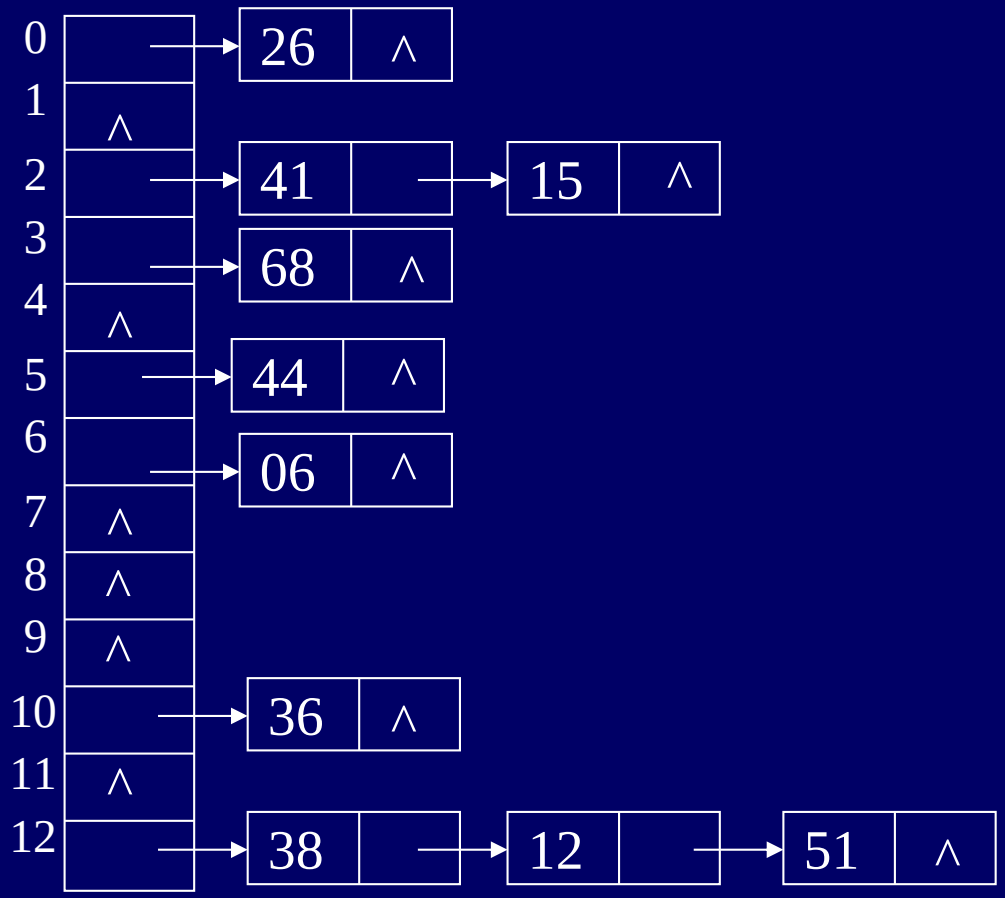
注意：当m>p（p＝|散列地址集|）时，分母应该为p

§ 9.4.4 散列表上的运算

- 不成功

拉链法：不包括空指针判定

ASL＝(1+0+2+1+0+1+1+0+0+0+1+0+3)/13=0.77 //m=13



比较次数：　1　　2　　3

§ 9.4.4 散列表上的运算

- 一般情况
  - 线性探测

    成功：ASL＝（1＋1/(1－α)）/2

    失败：ASL＝（1＋1/(1－α)²）/2

    显然，只与α相关，只要α合适，ASL＝O(1)。
  - 其他方法优于线性探测：略

5、用途

- 加速查找
- 加密等信息安全领域

Ch.10 排序

§10.1 基本概念

- 排序：
  - 输入：n个记录$R_1, R_2, \ldots, R_n$，相应的key为$K_1, K_2, \ldots, K_n$
  - 输出：$R_{i1}, R_{i2}, \ldots, R_{in}$，使得$K_{i1} \le K_{i2} \le \ldots \le K_{in}$(或≥)
- 稳定性：相同的keys经排序后相对次序不变
- 内排和外排：
- 分类：
  - ①按方法分：插入，选择，交换，归并，分配
  - ②按存贮方式分：顺序表、链表、辅助表
- 就地排序：辅助空间O(1)
- 存贮结构说明：设n预先有定义。设RecType同前一章NodeType。 SeqList为n+1个单元
- 以下只讨论增序

§10.2 插入排序

基本思想：每次将待排序的记录，依key大小插入到前面已排好序的子文件中的适当位置。

§10.2.1 直接插入排序

- **思想：**待排序记录存于R[1..n]中，排序过程中某一刻，R被分为两子区间：

当前无序区的第1个记录R[i]插入有序区R[1..i-1]中合适的位置，使R[1..i]变为有序区

初始时，因为R[1]自然有序，R[2..n]为无序区，故只须从R[2]开始插入

此过程类似于理牌

§10.2.1 直接插入排序（续）

- 怎样将R[i]插到有序区，使其扩充呢？

① 先找到正确的插入位置k（1≤k≤i）；然后将R[k..i-1]中记录后移1位，再插入

② 从有序区尾向前查找插入位置，同时做后移操作（二者交替进行），直到找到第一个不大于R[i].key的记录为止，并将R[i]插入其后。

§10.2.1 直接插入排序（续）

```
void InsertSort(SeqList R){
 int i, j;
 for (i=2; i<=n; i++)  //依次插入R[2],..R[n]，共进行n-1趟排序
    if (R[i].key<R[i-1].key) {//否则，R[i]在原位置上
        R[0] = R[i];  //R[0]既是哨兵，又起保存R[i]的作用
        j = i-1;
       do{
          //从后向前在有序区R[1..i-1]中找到第1个不大于R[i].key的记录
            R[ j+1] = R[ j ];  //大于R[ i ].key的记录后移
            j--;
        }while (R[0].key<R[ j ].key);  //哨兵防止越界
        R[ j+1] = R[0];//R[i]插入正确位置,循环终止于R[ j ].key≤R[0].key
    }//endif
}
```

§10.2.1 直接插入排序（续）

- 就地排序、稳定的排序
- 例子略
- 时间分析

① 最好情况：初始正序

每趟插入R[i]时，因为它都大于等于R[i-1]的key，无须移动，不进入内循环C_{min}=n-1，M_{min}=0

② 最坏情况：初始反序

- 比较：每趟排序均需比较R[0..i-1]中所有keys（if比较1次，do中比较i-1次）即比较i次

$$C_{\max} = \sum_{i=2}^{n} i = O(n^2)$$

§ 10.2.1 直接插入排序（续）

- **移动**：内循环中R[1..i-1]均后移，外加设置哨兵和将R[i]插到最终位置共移动i-1+2=i+1.

$$M_{\max} = \sum_{i=2}^{n} (i+1) = O(n^2)$$

③ 平均：

若记录随机分布。比较、移动平均为$n^2/4$

§10.2.2 希尔排序

希尔排序(Shell Sort，又称缩小增量法)是一种分组插入排序方法。

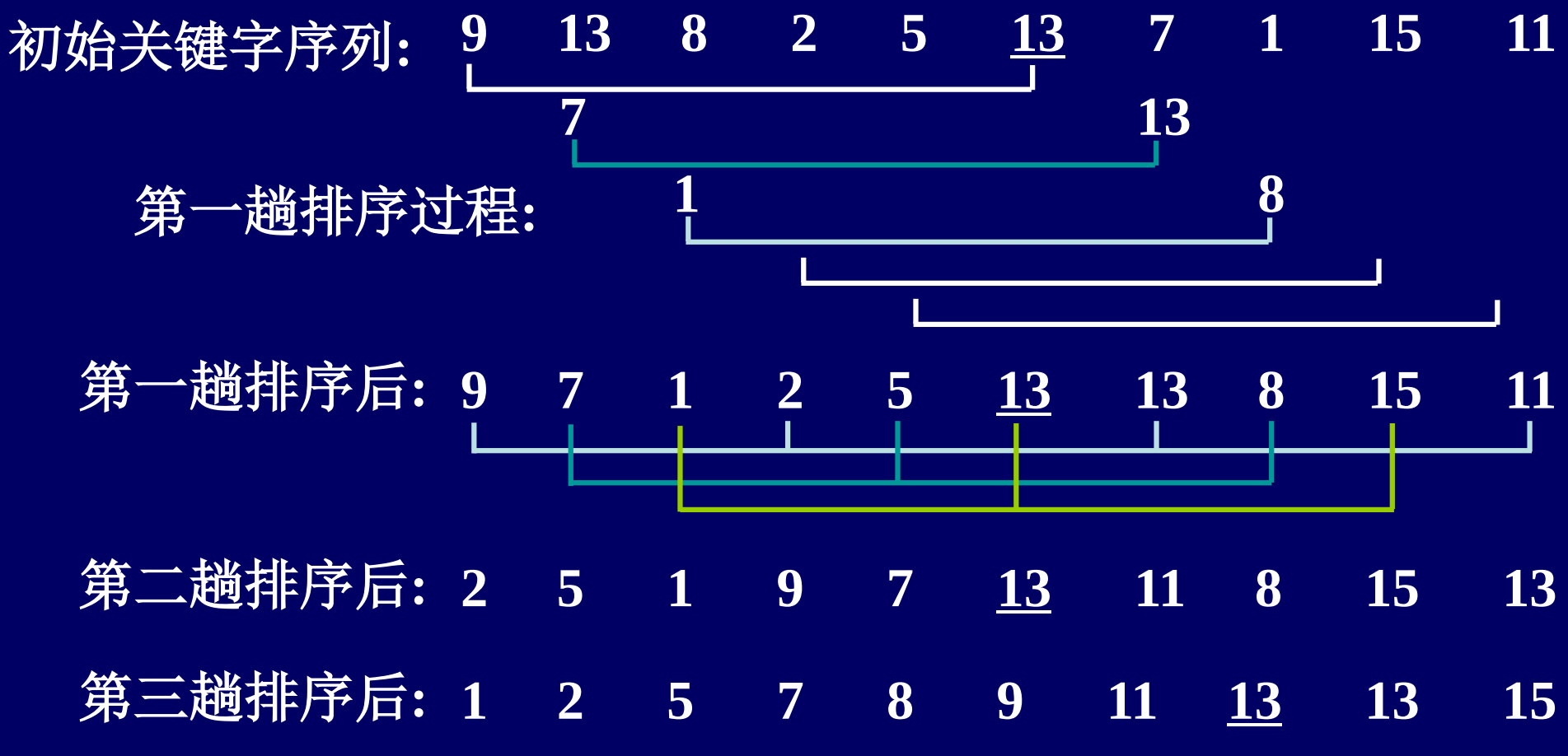
1 排序思想

① 先取一个正整数d_1(d_1<n)作为第一个增量，将全部n个记录分成d_1组，把所有相隔d_1的记录放在一组中，即对于每个k(k=1, 2，… d_1)，R[k], R[d_1+k], R[2d_1+k]，…分在同一组中，在各组内进行直接插入排序。这样一次分组和排序过程称为一趟希尔排序；

② 取新的增量d_2<d_1，重复①的分组和排序操作；直至所取的增量d_i=1为止，即所有记录放进一个组中排序为止。

2 排序示例

设有10个待排序的记录，关键字分别为9, 13, 8, 2, 5, 13, 7, 1, 15, 11，增量序列是5, 3, 1，希尔排序的过程如图所示。



希尔排序过程

3 算法实现

先给出一趟希尔排序的算法，类似直接插入排序。

```
void shell_pass(Sqlist *L, int d)
    /*  对顺序表L进行一趟希尔排序, 增量为d  */
  {  int j, k ;
     for (j=d+1; j<=L->length; j++)
        {  L->R[0]=L->R[j] ;        /*  设置监视哨兵  */
           k=j-d ;
           while (k>0&&L->R[0].key < L->R[k].key )
              {   L->R[k+d]=L->R[k] ; k=k-d ;   }
           L->R[k+d]=L->R[0] ;
        }
  }
```

然后在根据增量数组dk进行希尔排序。

```
void shell_sort(Sqlist *L, int dk[], int t)
   /*   按增量序列dk[0 … t-1],对顺序表L进行希尔排序   */
   {  int m ;
      for (m=0; m<=t; m++)
         shll_pass(L, dk[m]) ;
   }
```

希尔排序的分析比较复杂，涉及一些数学上的问题，其时间是所取的“增量”序列的函数。

希尔排序特点

子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。

希尔排序可提高排序速度，原因是：

- 分组后n值减小，n^2更小，而$T(n)=O(n^2)$,所以T(n)从总体上看是减小了；
- 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。
- **当n较大时，比较和移动次数约在$n^{1.25}$~$1.6n^{1.25}$之间**

增量序列取法

- 无除1以外的公因子；
- 最后一个增量值必须为1。

§10.3 交换排序

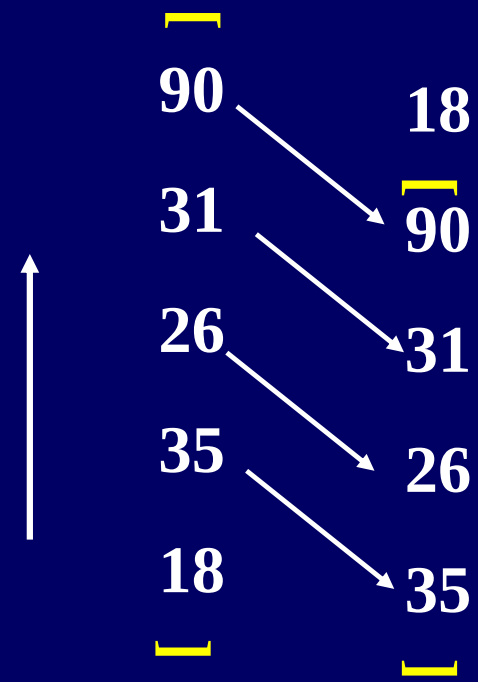
两两比较待排序记录的key，发现两个记录的次序相反时即进行交换，直到无反序的记录为止。

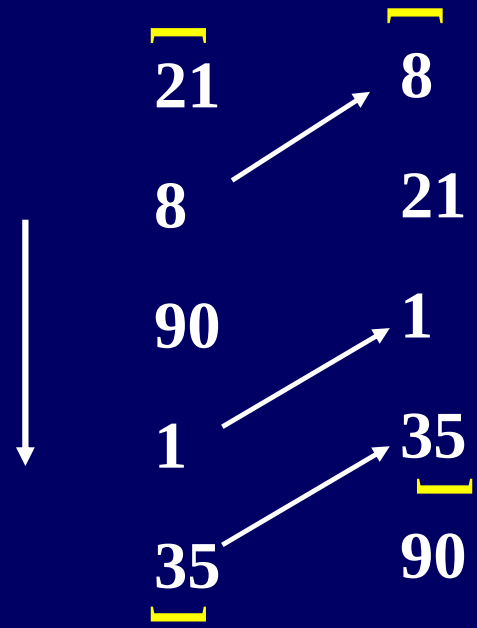
§10.3.1 冒泡排序

- 设想待排序的记录数组R[1..n]垂直竖立，将每个记录R[i]看作是重量为R[i].key的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组R，凡扫描到违反本原则的轻气泡，就使其向上“飘浮”，如此反复，直到最后任何两气泡都是轻者在上，重者在下为止。
- 从下往上和从上往下交替进行，可改善性能。

§10.3.1 冒泡排序

- 从下向上扫描：重气泡下沉一个位置，最轻气泡浮顶，最重气泡在顶时要n-1趟排序
- 从上向下扫描：请气泡上浮一个位置，最重气泡沉底，最轻气泡在底时要n-1趟排序





初始关键字序列:	23	38	22	45	23	67	31	15	41
第一趟排序后:	23	22	38	23	45	31	15	41	67
第二趟排序后:	22	23	23	38	31	15	41	45	67
第三趟排序后:	22	23	23	31	15	38	41	45	67
第四趟排序后:	22	23	23	15	31	38	41	45	67
第五趟排序后:	22	23	15	23	31	38	41	45	67
第六趟排序后:	22	15	23	23	31	38	41	45	67
第七趟排序后:	15	22	23	23	31	38	41	45	67

冒泡排序过程

```
void Bubble_Sort(Sqlist *L)
   {  int j ,k , flag ;
      for (j=1; j<L->length; j++)       /*   共有n-1趟排序   */
         {  flag=TRUE ;
            for (k=1; k<=L->length-j; k++)   /*   一趟排序   */
               if (LT(L->R[k+1].key, L->R[k].key ) )
                  {   flag=FALSE ; L->R[0]=L->R[k] ;
                      L->R[k]=L->R[k+1] ;
                      L->R[k+1]=L->R[0] ;
                   }
               if  (flag==TRUE)  break ;
         }
   }
```

算法分析

时间复杂度

- 最好情况(正序)：比较次数：n-1；移动次数：0；
- 最坏情况(逆序)：

比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

移动次数：$3\sum_{i=1}^{n-1}(n-i)=\frac{3n(n-1)}{2}$

故时间复杂度：$T(n)=O(n^2)$

空间复杂度：$S(n)=O(1)$

§10.3.2 快速排序

划分交换排序，1962年由Hoare(1980图灵奖)提出。

- **分治法**：将原问题分解为若干个规模较小但结构与原问题相似的子问题；递归地求解这些子问题，然后将这些子问题的解组合为原问题的解。

因此在用递归描述的分治算法的每一层递归上，都有如下三个步骤：

① **分解（divide）**：将原问题分解为若干个子问题，此步称为划分；

② **求解（conquer）**：递归地解各子问题，若子问题的size足够小，则直接求解；

③ **组合（combine）**：将各子问题的解组合为原问题的解。

第2步最易，表示为递归调用语句，分解和组合的难易取决于应用。

§10.3.2 快速排序（续）

- **快速排序的思想**：设待排序的无序区为R[low..high]

① **分解**：在当前的无序区选一记录作为基准，以此基准将其划分为：

R[low..pivotpos-1].keys≤R[pivotpos].key
≤R[pivotpos+1..high].keys (9.1式)

显然基准记录已在正确位置上，其余两子区间排序方法与原问题相同

② **求解**：通过递归调用快速排序对左、右子区间排序，当子区间长度小于等于1时无须排。

③ **组合**：当step②中两个递归调用结束时，其左右子区间内已有序，故R[low..high]自然有序，即组合操作为空操作。

§10.3.2 快速排序（续）

```
void QuickSort(SeqList R, int low, int high){
    //对R[low..high]快速排序
    int pivotpos;
    if (low<high){  //当区间长度大于1时须排序
        pivotpos = Partition(R, low, high);  //对R[low..high]划分
        QuickSort(R, low, pivotpos-1);  //对左区间递归排序
        QuickSort(R, pivotpos+1, high);  //对右区间递归排序
    }
}
```

为排序整个文件，只须调用QuickSort(R, 1, n)即可完成对R[1..n]的排序。

§10.3.2 快速排序（续）

- 如何划分

① 设两个指针i，j，初始时i=low, j=high

② 取排序区间第1个记录作基准：pivot=R[i]（即R[low]）

③ 令j从后往前扫描，直至找到第1个key小于pivot.key的记录R[j]，将R[j]移至i所指的位置（相当于R[j]和R[i]交换，注意交换前R[i]=pivot，交换后基准记录相当于在R[j]中，i++）

④ 令i指针从前往后扫描，直到找到第1个关键字大于pivot.key的记录R[i]，将R[i]移到j所指位置（相当于交换R[i]和基准R[j]，交换后R[i]中又相当于存放了pivot，j- -）。

⑤ 重复③④，如此交换改变扫描方向，从两端向中间靠拢，直至i=j，i便是基准pivot最终的位置，将pivot放于此。

§10.3.2 快速排序（续）

初始

j向前

第1次交换后i++

i向后

§10.3.2 快速排序（续）

2nd交换后j--

j向前扫描
位置不变交换后i++

i向后扫描
位置不变交换后j--

j向前，完成

§10.3.2 快速排序（续）

```
int Partition(SeqList R, int i, int j){  //调用时i=low, j=high
    Rectype pivot=R[i];  //区间第1个记录为基准
    while (i<j) {  //从区间两端交替向中间扫描至i=j为止
        while (i<j && R[ j ].key>= pivot.key)  //pivot相当于在R[i]
            j--;  //从后往前扫描，找第1个key小于基准关键字的记录R[ j ]
        if (i<j)  //表示找到R[ j ]
            R[i++] = R[ j ];  //相当于交换R[i]和R[j]，交换后i加1，pivot在R[ j ]
        while (i<j && R[i].key<= pivot.key)
            i++;  //从前往后扫描，找第1个key大于基准关键字的记录R[ i]
        if (i<j)  //表示找到R[ i]
            R[ j--] = R[i]; //相当于交换R[i]和R[j]，交换后j减1，pivot回到R[i]
    } //i=j时终止
    R[i] = pivot;  //基准记录已被最后定位
    return i;
}
```

§10.3.2 快速排序（续）

- **时间分析**：主要耗费在partition上。对长度为k的区间划分，需比较keys共k-1次

① **最坏情况**：每次划分都是基准恰为当前区间中关键字最小（或最大）的记录

划分结果是左子区间（或右子区间）为空，而另一子区间长度仅比原区间小了1

这时须做n-1次划分，第i次划分开始时区间长度为n-i+1，需比较的次数为n-i

$$C_{\max} = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

显然，当文件状态递增（减）有序时，正是如此。

§10.3.2 快速排序（续）

② **最好情况**：每次划分所取的基准是当前区间中的“中值”记录，划分后左、右子区间大致相等。

设C(n)表示对长度为n的文件进行快速排序所需比较次数

显然$C(n) \le n-1+2C(n/2)$

//设$n=2^k$，n-1是划分R[1..n]的比较次数

$C(n) \le n+2C(n/2)$

$\le n+2[n/2+2C(n/2^2)] = 2n+4C(n/2^2) \le \ldots$

$\le kn+2^kC(n/2^k) = n\lg n+nC(1)$

$= O(n\lg n)$ //k=lgn

§10.3.2 快速排序（续）

③ 平均情况：Tavg(n) ≈1.39nlgn+O(n)

尽管它的最坏情况$O(n^2)$，但就平均时间而言最快

- 改进

为使划分较均匀，避免取当前区间最大最小元做基准

- 三者取中法：然后将它与当前区间第1个元素交换，仍可使用上述Partition算法
- 最好方法是产生一个R[i..j]之间的随机数

K=random(i,j); //k∈ [i,j]

交换R[i]和R[k]，然后使用Partition

- 不稳定：请检查反例[2, 2, 1]
- 递归： 要使用栈，平均深度O(lgn)

§10.4 选择排序

基本思想：每一趟从待排序的记录中选出最小key的记录（简称最小元），放在已排好序的子区间最后

§10.4.1 直接选择排序（简单选择）

- **基本思想：**
  - 第1趟，无序区为R[1..n]，选最小者放在R[1]，无序区变为[2..n].
  - 第i趟，有序区为R[1..i-1]，无序区为R[i..n]

    显然R[1..i-1].keys≤R[i..n].keys

    选无序区中最小者R[k]，交换R[i]和R[k]后使R[1..i].keys≤R[i+1..n].keys //有序区长度加1，无序区长度减1
  - 第n-1趟之后，R[1..n-1].keys ≤R[n].keys，结束

§ 10.4.1 直接选择排序（简单选择）

- 算法

```
void SelectSort(SeqList R){
    int i, j, k;
    for (i=1; i<n; i++){  //第i趟排序，1≤i ≤n-1
        k = i;
        for (j=i+1; j<=n; j++)
        //在当前无序区R[i..n]中选key最小的记录R[k]
            if (R[j].key<R[k].key) k=j;
        if (k!=i) R[i]↔R[k];  //可用R[0]做交换单元
    }
}
```

§ 10.4.1 直接选择排序（简单选择）

时间

- 比较：

  无论文件状态为何，第i趟排序中需比较n-i次（内循环次数）

  $$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2=O(n^2)$$ //$C_{max}=C_{min}$

- 移动：

  状态正序：$M_{min}=0$

  状态逆序：每趟交换1次，$M_{max}=3(n-1)$

  } O(n), 较少

- 就地，不稳定，检验反例[2, 2, 1]

§ 10.4.2 堆排序（Floyd &Williams）

直接选择的比较次数多是因为后一趟未利用前一趟的比较结构，树形选择可克服此缺点，但它耗费的空间大，故实用的树形选择是堆排序。

- **思想**

将R[1..n]看作是1棵完全二叉树的顺序存储结构，利用完全二叉树的双亲和孩子之关系，在当前无序区里选择最小（大）元扩充至有序区。

- **二叉堆（快速选择最大/小元）**

n个keys序列K_1，K_2，…，K_n称为堆，当且仅当满足如下性质（**堆性质**，**堆序**）：

（1）$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases}$ 或 （2）$\begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases}$

这里，$1 \le i \le \lfloor n/2 \rfloor$。即i结点不是叶子

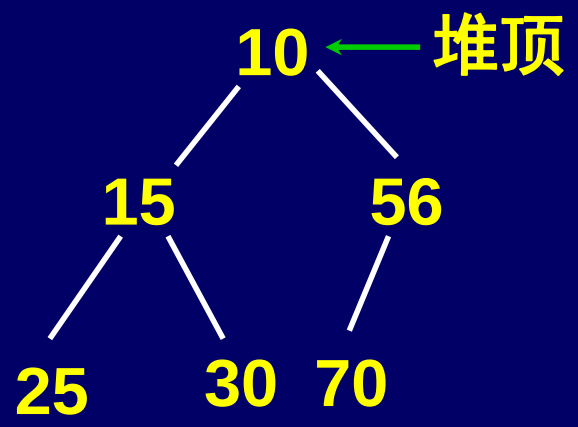
§ 10.4.2 堆排序

堆性质

将R[1..n]看作是完全二叉树的顺序存储结构时，堆性质实质上是满足如下性质的完全二叉树：

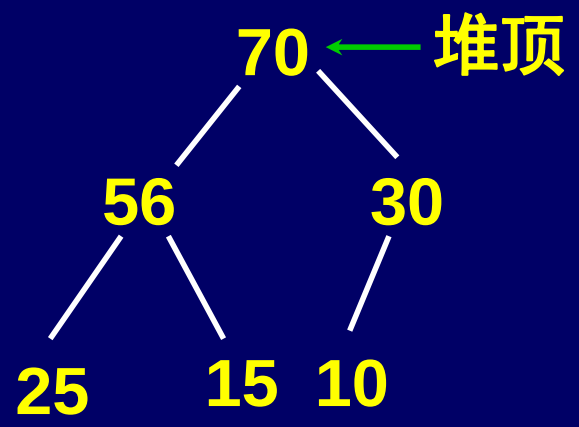
树中任一非叶结点的key均不大/小于其左右孩子（若存在）结点的key。即：从根到任一叶子的路径上的结点序列是一个有序序列，堆中任一子树仍是堆。它适合查找吗？

10	15	56	25	30	70



小根堆

70	56	30	25	15	10



大根堆

§ 10.4.2 堆排序

算法思想

1、初始化 将R[1..n]建成大根堆，即初始无序区。

2、排序

- 交换：设当前无序区是大根堆R[1..i]，交换其中的首尾记录，即最大元R[1]（堆顶）交换到无序区尾部（有序区头部），使有序区在R的尾部逐渐扩大：

  R[1..i-1].keys≤R[i..n].keys //前者为无序区，后者为有序区

  显然，i＝n,n-1,…,2，即n－1趟排序即可完成。

- 调整：将新无序区R[1..i-1]调整为堆。注意：只有R[1]可能违反堆性质。

§ 10.4.2 堆排序

- 算法实现

```
void HeapSort( SeqList  R ) {
    int i;
    BuildHeap( R ); //将R[1..n]建成初始堆
    for ( i=n; i>1; i-- ) { //进行n-1趟堆排序，当前无序区为R[1..i]
        R[1] ⟺ R[i]; //无序区首尾记录交换，R[0]做暂存单元
        Heapify( R,1,i-1 ); //将R[1..i-1]重新调整为堆
    }
}
```

- 如何调整堆和构造初始堆？

§ 10.4.2 堆排序

调整（重建）堆

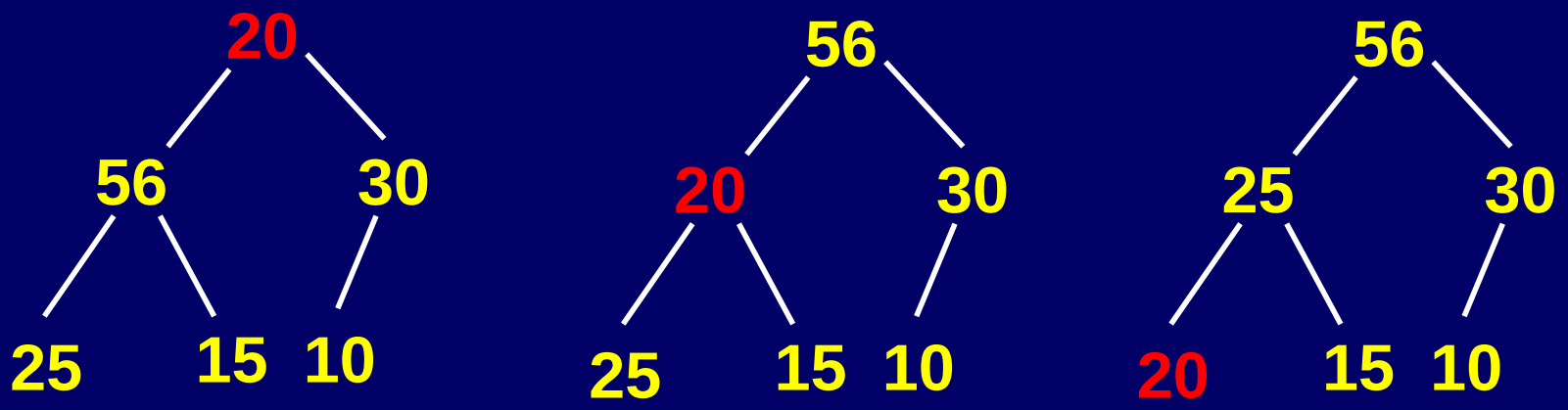
设调整区间为R[low..high]，因为只有根R[low]违反堆序，它的两子树（若存在，则根为R[2low]，R[2low+1]）均是堆。

- **无须调整** 若R[low].key不小于两孩子的Keys，则R[low]不违反堆序
- **必须调整** 将R[low]和它的两孩子中较大者交换：

  设R[large].key=max{ R[2low].key, R[2low+1].key }

  令R[low] ⟺ R[large]

交换后R[large]可能违反堆序，重复上述过程，直至被调整的结点已满足堆序，或该结点已是叶子。



§ 10.4.2 堆排序

■ 调整堆算法

```
void Heapify( SeqList  R, int low, int high ) {
    int large; //只有R[low]可能违反堆序
    RecType temp=R[low];
    for ( large=2*low; large<=high; large*=2 ) {
        //R[low]是当前调整结点，若large>high，则R[low]是叶子，结束；
        //否则，先令large指向R[low]的左孩子
        if (large<high && R[large].key<R[large+1].key )
            large++; //若R[large]有右兄弟，且右兄弟大，则令large指向右兄弟
        if ( temp.key>=R[large].key ) break; //满足堆序
        R[low]=R[large]; //交换，小的筛下
        low=large; //令low指向新的调整结点
    }
    R[low]=temp; //将被调整结点放到最终的位置
}
```

§ 10.4.2 堆排序

- **构造初始堆算法**

  将R[1..n]建成堆，须将其每个结点为根的子树均调整为堆。对叶子（i> $\lfloor n/2 \rfloor$）无须调整，只要依次将以序号为 $\lfloor n/2 \rfloor$,$\lfloor n/2 \rfloor -1$, … ,2,1的结点为根的子树均调整为堆即可。按此次序调整每个结点时，其左右子树均已是堆

```
void BuildHeap( SeqList  R ) {
    int i;
    for ( i=n/2; i>0; i-- )
        Heapify( R, i, n); //将R[i..n] 调整为堆
}
```

- **时间：** 最坏及平均皆为O(nlgn) **(2nlgn+O(n))**
- **特点：** 就地，不稳定

§ 10.5 归并排序

- **归并的基本思想：**将K（K≥2）个有序表合并成一个新的有序表。
- **二路归并：**K＝2，类似于理牌

```
void Merge( SeqList  R, int low, int m, int high ) {
  //将2个有序表R[low..m]和R[m+1..high]归并为1个有序表R[low..high]
  int i=low, j=m+1, p=0; //i,j指向输入子表头，p指向输出子表
  RecType *R1=(RecType *)malloc(high-low+1)*sizeof(RecType));//输出
  if （ !R1 ）Error( “存储分配失败” )
  while( i<=m && j<=high ) //2子表非空时，取其头上较小者输出到R1[p]上
        R1[p++]=( R[i].key<=R[j].key ) ? R[ i++]: R[ j++] ;
  while ( i<=m ) //第1子表非空，将剩余记录copy到R1中
        R1[p++]=R[ i++ ];
  while ( j<=high ) //第2子表非空，将剩余记录copy到R1中
        R1[p++]=R[ j++ ];
  R=R1; //将R1copy回R，R[low..high]⇐R1[0..high-low]
}
```

§ 10.5 归并排序

- 排序算法
  - 自底向上：见Fig10.13
  - 自上而下：分治法设计

(1)分解：将当前区间一分为二，分裂点mid＝$\lfloor (low + high)/2 \rfloor$

(2)求解：递归地对2个子表R[low..mid]和R[mid+1..high]进行归并排序，出口是当前区间长度为1。

(3)组合：将上述两有序的子表归并成1个有序表R[low..high]

```
void MergeSort( SeqList  R, int low, int high ) {
   int mid;
   if ( low<high ){ //区间长度>1
        mid=( low+high )/2; //分解
        MergeSort( R, low, mid ); //解子问题1，结束时R[low..mid]有序
        MergeSort( R, mid+1, high ); //解子问题2，结束时R[mid+1..high]有序
        Merge( R, low, mid, high ); //组合
   }//endif
}
```

§ 10.5 归并排序

- 性能分析
  - 时间：最好最坏均是O(nlgn)
  - 空间：辅助O(n)，非就地排序
- 特点
  - 稳定
  - 易于在链表上实现
  - 与快排相比：分解易、组合难

§ 10.6 分配排序

- 基于比较的排序时间下界: $\lceil \lg n! \rceil$

  由Stirling公式知：lgn!＝nlgn-1.44n+O(lgn)

  要突破此界，就不能通过keys的比较。

- 分配排序正是如此，它通过“分配”和“收集”过程实现排序，时间为O(n)。

§ 10.6.1 箱排序

- 基本思想
  - 分配：扫描R[0..n-1],将key等于k的记录全装入k^{th}箱子里
  - 收集：按序号将各非空箱子首尾连接起来
  - 多趟：每个关键字1趟，例如:扑克牌
- 时间：
  - 分配：O(n)；
  - 收集：设箱子数为m(与key相关)，时间为O(m)或O(m+n)
  - 总计：O(m+n)=O(n) if m=O(n)

§ 10.6.2 基数排序

基本思想

箱排序只适用于keys取值范围小的情况，否则浪费时间和空间。

例如，若$m = O(n^2)$, 则时间和空间均为$O(n^2)$。

基数排序是通过分析key的构成，用多趟箱排序实现的。

例子

设$n = 10$，$k_i \in [0..99]$, $1 \le i \le 10$

输入:（36,5,10,16,98,95,47,32,36,48）

将2位整数看作2个keys，先对个位，后对十位做箱排序。因此，无须100个箱子，只要10个箱子。

§10.6.2 基数排序

（36,5,10,16,98,95,47,32,36,48）

第1趟箱排序

分配：

0	10
1	
2	32
3	
4	
5	5,95
6	36,16,36
7	47
8	98,48
9	

收集：

10,32,5,95,36,16,36,47,98,48

第2趟箱排序

分配：

0	05
1	10,16
2	
3	32,36,36
4	47,48
5	
6	
7	
8	
9	95,98

收集：

05,10,16,32,36,36,47,48,95,98

- 各趟排序前要求清空箱子，分配和收集须按FIFO进行，箱子用链队列表示。
- 除第1趟外，其余各趟一定要是稳定的排序，否则结果可能有错。
- m不再在数量级上大于O(n)，故上述排序时间是O(n)

§10.6.2 基数排序

一般情况

设任一记录R[i]的key均由d个分量 $K_i^0 K_i^1 \ldots K_i^{d-1}$ 构成

- 多关键字文件：d个分量皆为独立的key
- 单关键字文件：每个分量是key中的1位，只讨论这种情况。

设每位取值范围相同：

$$C_0 \le K^j \le C_{rd-1} \quad (0 \le j < d)$$

这里，rd称为基数，d为key的位数。

若key为十进制整数，按位分解后rd＝10，C_0＝0，C_9＝9

排序算法

从低位到高位依次对K^j（j＝d-1,d-2,…,0）进行d趟箱排序，所需的箱子数为基rd。

```
#defin KeySize 4 //设d＝4
#define Radix 10 //基rd为10
typedef  RecType  DataType; //各箱子用链队列表示，其中的结点数据类型改为与本章的记录类型一致
```

§10.6.2 基数排序

```
void RadixSort( SeqList R ){
    //对R[0..n-1]做基数排序，设keys为非负整数，
    //且位数不超过KeySize
    LinkQueue B[Radix]; //10个箱子
    int i;

    for ( i=0; i<Radix; i++ ) //初始化
          InitQueue(&B[i]); //清空箱子
    for( i=KeySize-1; i>=0; i-- ) {
         //对位i箱排序，从低位到高位进行d趟箱排序
          Distribute( R,B,i ); //分配
          Collect( R,B );  //收集
    }
}
```

§10.6.2 基数排序

```
void Distribute( SeqList R, LinkQueue B[ ], int j ){
    int i，k，t；  //按关键字j^th分量分配，进入此过程时各箱子为空
    j=KeySize - j; //将 j 换算为从个位起算，个位是第1位
    for ( i=0; i<n; i++ ) { //扫描R，装箱
          k=R[i].key;
          for( t=1; t<j; t++ ) k=k/10; //去掉key的后j-1位：k=k/10^(j-1)
           k=k%10; //取key的第j位数字k
          EnQueue( &B[k],R[i] ); //将R[i]装入箱子B[k]
    }
}
void Collect( SeqList R, LinkQueue B[ ] ){
    int i=0, j; //依次连接各非空箱子，完成后使各箱子变空
    for ( j=0; j<Radix; j++ ) //将j^th箱子的内容依FIFO序收集到R中
        while ( !QueueEmpty (&B[ j ]) ) //若是链队列，只需首尾链接
            R[i++]=DeQueue( &B[ j ]);
}
```

§10.6.2 基数排序

- **链式基数排序**：文件R不是以向量形式，而是以单链表形式给出。
- **时间：线性阶**

  箱子初始化：O(rd)

  分配时间：O(n)，不计求第j位数字的时间

  收集收集：O(n+rd)，链式为O(rd)

  d趟总时间：O(d(2n+rd))，链式为O(d(n+rd))

  T(n)=O(n) if d=O(1) and rd=O(1) ～ O(n)

  - 设key是十进制整数，d是常数吗？

    若n个keys的取值范围是0 ～n^k，(常数k>1)，则key的位数是：

    $d=\log_{10}n^k=k\log_{10}n=O(k\lg n)$

    因此，基数排序的时间是O(nlgn)。但是可将其改为n进制表示：

    $rd=n$， $d=\log_n n^k=k$，$T(n)=O(k(n+n))=O(n)$

- **辅助空间：O(n+rd)**
- **对key的要求：**
- **稳定排序：要求第1趟稳定，其余各趟必须稳定。**

§10.7 各种排序方法的比较和选择

- **选择因素**：实例规模，记录大小，key的结构及初始状态，对稳定性的要求，存储结构，时间和辅助空间的要求，语言工具（指针）。
- **比较**

n		直接插入	直接选择	冒泡排序	堆排序	快速排序
随	4000	5.67	17.30	15.78	0.13	0.07
	8000	23.15	29.43	64.03	0.28	0.17
	10000	35.43	46.02	99.10	0.35	0.22
	15000	80.23	103.00	223.28	0.58	0.33
机	20000	143.67	185.05	399.47	0.77	0.47
增	20000	0.05	185.78	0.03	0.75	0.23
减	20000	286.92	199.00	584.67	0.80	0.28

- **说明**

直接选择无论k和i是否相等，均交换；快排用中间元做划分元。

§ 10.7 各种排序方法的比较和选择

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
冒泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
希尔		$O(n^{1.25})$		O(1)	不稳定
快速	O(nlgn)	O(nlgn)	$O(n^2)$	O(lgn)	不稳定
堆	O(nlgn)	O(nlgn)	O(nlgn)	O(1)	不稳定
归并	O(nlgn)	O(nlgn)	O(nlgn)	O(n)	稳定
基数	O(d*n+d*rd)	O(d*n+d*rd)	O(d*n+d*rd)	O(n+rd)	稳定

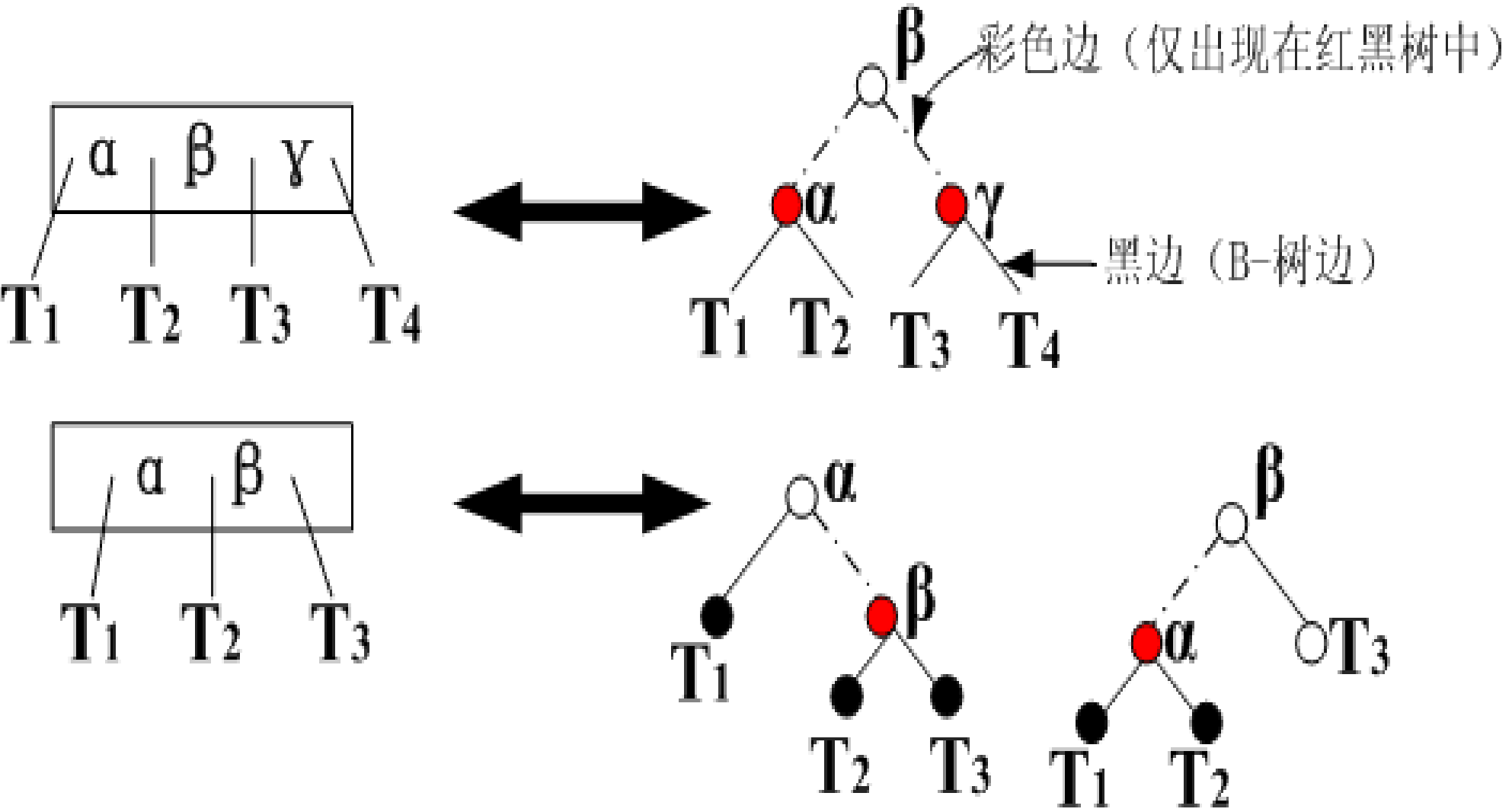
Chapter 13. 红黑树

§13 红黑树

二叉搜索树上执行查找、前趋、后继、最大/小值、插、删等动态集合。操作时间取决于树高O(h)，当它退化为单支树时，其时间和单链表上操作时间一样为O(n)

- 平衡二叉树O(lg n)
  - 完全平衡 —-满二叉树
- 平衡的二叉搜索树
  - AVL—62年　至多1.44lgn
  - 红黑树—72年　Bayer　高至多为2lg(n+1)
  - 4阶B树↔红黑树　　keys：1～3，subtrees：2～4

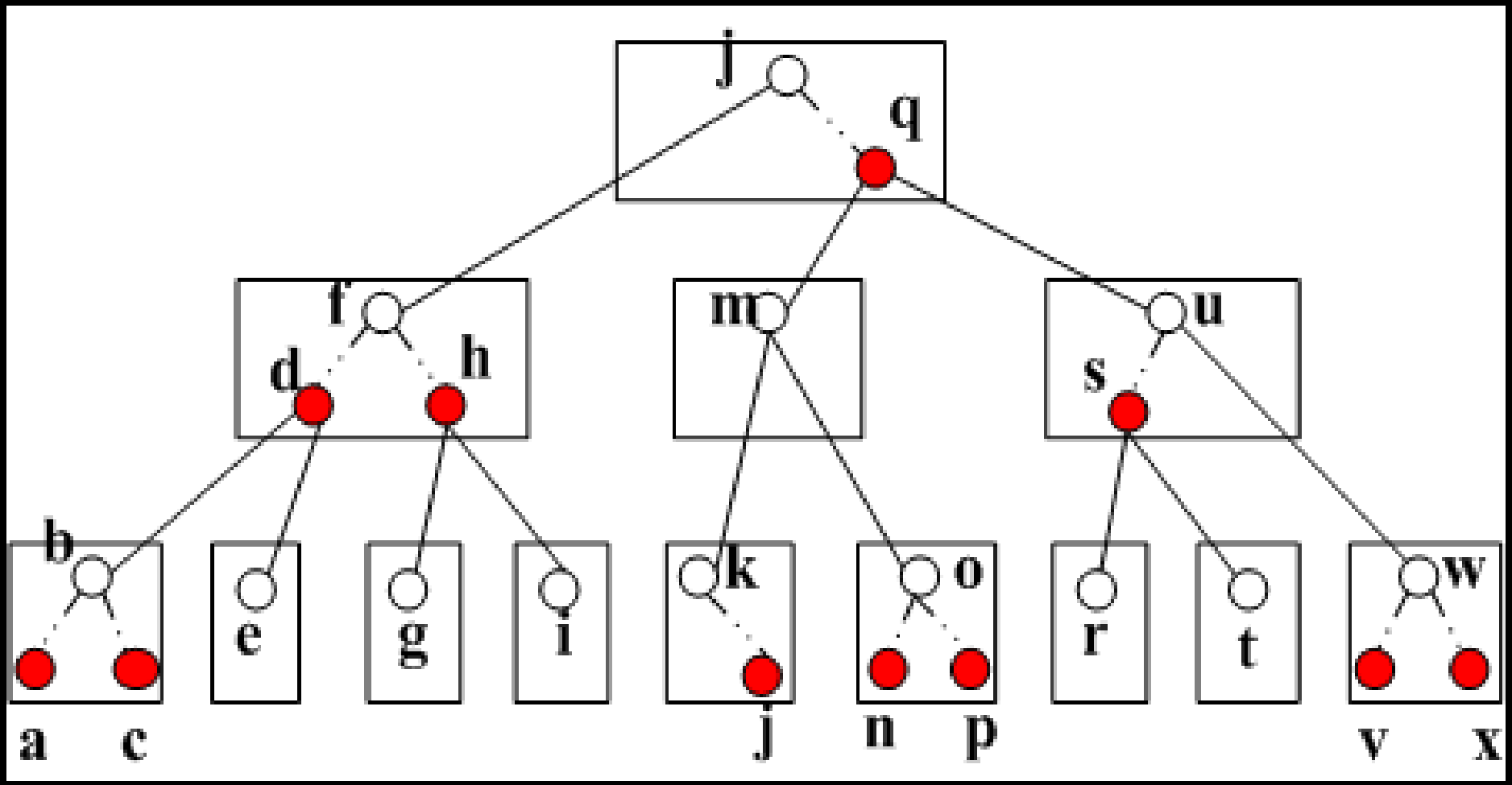
§13 红黑树



结点内只有一个关键字不变

§13 红黑树（续）

- B-树只适于外部查找，内部查找只适用于阶数较低的B-树。4阶B-树转换为红黑树，后者变为二叉搜索树，其平衡性由B-树得以保证



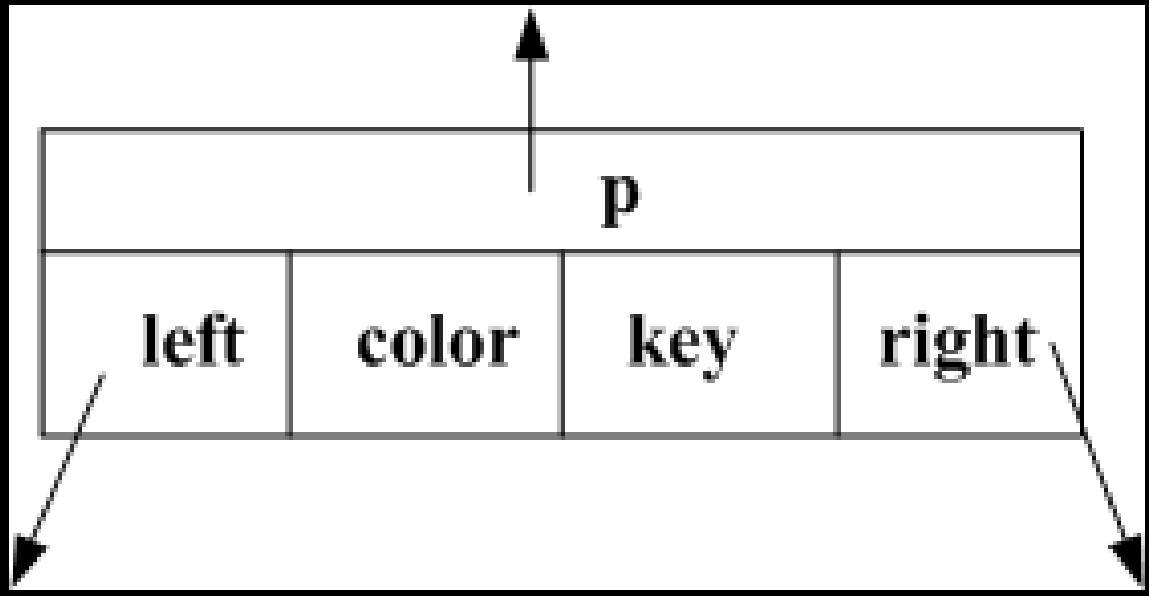
§13.1 定义及性质

- **Def 1. 红黑树是满足下述性质（简称RB性质）的二叉搜索树:**
  1. **每个结点或红或黑**
  2. **根为黑色**
  3. **每个叶子（nil）为黑（外部结点）**
  4. **若结点为红，则其两个孩子必为黑**
  5. **每一结点到其后代叶子的所有路径含有相同数目的黑结点**

§13.1 定义及性质

结点结构

- color—1位表示红或黑
- 内点：含关键字
- 外点：所有空指针域加上外结点（树叶）



§13.1 定义及性质（续）

- 例子（三种表示）

  ① Fig13.1(a)通常表示

  ② 所有nil指针域用1个哨兵NIL[T]表示（包括根的双亲指针），哨兵的color为黑。哨兵 — 简化边界条件处理，NIL[T]是一个含有5个域的对象，和通常结点一样，若每个nil域用一个哨兵表示，浪费空间，但其好处是其父亲无二义性，即每个P[NIL[T]]有确定的值。所有NIL共享一个NIL[T]时，处理稍复杂

  ③ 省略空指针表示

- Def 2: 结点x的黑高bh(x)是该结点到它的任何后代叶子路径上的黑点数（不包括x本身）。

  性质5保证此定义有效

§13.1 定义及性质（续）

- **Def 3：红黑树的黑高是根的黑高：bh(root[T])**
- **Lemma13.1：一棵n个内点的红黑树的高度至多为2lg(n+1)**

Pf:

- **方法1**

$$\because B-\text{树高为：}$$

$$h=O\left(\log_{\lceil m/2\rceil} n\right)=O(\lg n)$$

$$\therefore \text{树高为} O(2\lg n)$$

§13.1 定义及性质（续）

❖方法2

⑴∵红点下层必为黑，

∴bh(root)≥h/2，即红点最多为h/2层

(2)∵4阶B树中每个结点（方框）均有一黑点，

∴B树高=*bh*(*root*)

RB树的内点数=*B*树的（关键字数）*n*

≥高为*bh*(*root*)的*B*树结点数（方框数）

≥高为*bh*(*root*)的满二叉树的结点数

$$\geq 2^{bh(root)}-1\geq 2^{h/2}-1\Rightarrow \lg(n+1)\geq h/2$$

§13.1 定义及性质（续）

❖方法3

1. 先证任一结点x为根的子树至少有$2^{bh(x)}-1$个内点（用B树易证，∵每一B树中结点有一黑点，而B树关键字[RB树中结点]数≥相应的满二叉树关键字数）

对x的*高度*（不是黑高）用归纳法证明

❖**归纳基础**：若h(x)=0，则x为叶子（nil）bh(x)=0，$2^0-1=0$无内点，故成立。

❖设h(x)>0，则x是内点，有两孩子。x的每孩子黑高为bh(x)或bh(x)-1

x的红孩子黑高应为bh(x)，∵该红孩子没有计算在bh(x)中，而x自身也未算

x的黑孩子黑高应为bh(x)-1，∵该孩子在bh(x)中已计算一次，但该孩子在自己的黑高中不计算

§13.1 定义及性质（续）

$\because$ x的孩子高度 $< h(x)$，由归纳假设，

以x的孩子为根的子树至少有$2^{bh(x)-1}-1$个内点

$\therefore$ 以x为根的子树的内点数至少为：

$$\underbrace{2(2^{bh(x)-1}-1)}_{\text{两孩子为根...}}+1=2^{bh(x)}-1$$

↑

§13.1 定义及性质（续）

（2）根黑高至少为树高h的一半

∵ 性质4：红点必有两黑孩子．即：bh(root) ≥ h/2

//根到任何叶子的路径上至少有一半结点为黑点（不包含根）

由（1）知该树内点数n至少为$2^{h/2}-1$，i.e.，

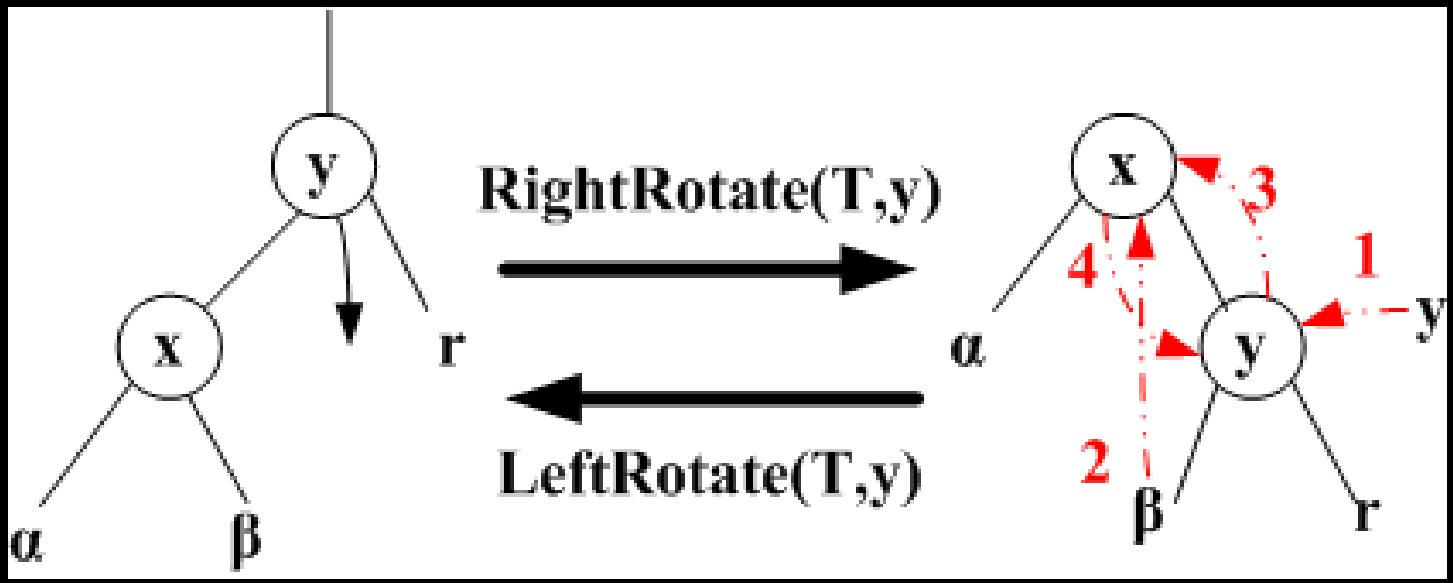
$$n \geq 2^{h/2}-1 \Rightarrow (n+1) \geq 2^{h/2} \Rightarrow \lg(n+1) \geq h/2 \Rightarrow h \leq 2\lg(n+1)$$

该引理保证树高O(lg n) ∴ 基本操作时间$O(\lg n)$

- Ex13.1-3 Ex13.1-5

§13.2 旋转

- 插入、删除等修改操作可能违反红黑树性质，恢复时须修改某些结点颜色和改变指针
- 属性：双亲P，左子L，右子R，颜色C，关键字k



- 左旋
  - Step1: 记录y
  - Step2: β连到x右
  - Step3: y连到p[x]（y是子树根）
  - Step4: x连到y左

§13.2 旋转（续）

对x做左旋时，假定x的右子y非空(y不动，x-y左旋 90°)

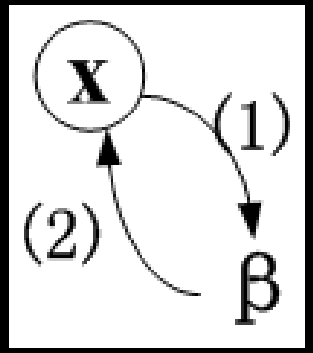
对y做右旋时，假定y的左子x非空(x不动，x-y右旋 90°)

- 旋转仍保留了排序树性质不变

  $\alpha \le x \le \beta \le y \le \gamma$ //红黑性质暂不讨论

- 右旋与之对称
- 算法

```
LeftRotate (T, x) {
    //假定right[x] ≠ nil，结点有双亲指针
        y ← right[x];   //Step1
step2   right[x] ← left[y];
β取代y      //(1)处理β  y左子树为x右子树, 即β代y
        if  left[y] ≠ nil  then
          //β非空，当用nil[T]取代nil时可省略
          P[left[y]] ← x;   //(2)
```



step3
y取代x

$p[y] \leftarrow p[x]$;

//(1)step3，y与p[x]连接 p[x]为y的双亲

if $p[x] = nil[T]$ *then* //x为根

$root[T] \leftarrow y$ //T的全局属性

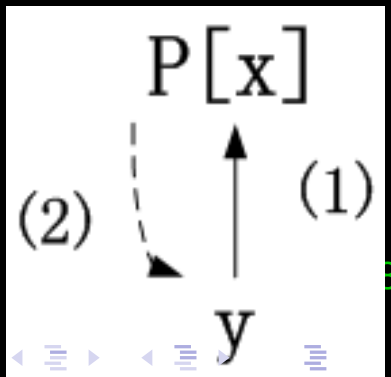
else //x非根，处理x是p[x]左还是右孩子

if $x = left[p[x]]$ *then* //(2)x为p[x]左子

$left[p[x]] \leftarrow y$ //y为p[x]左子

else

$right[p[x]] \leftarrow y$

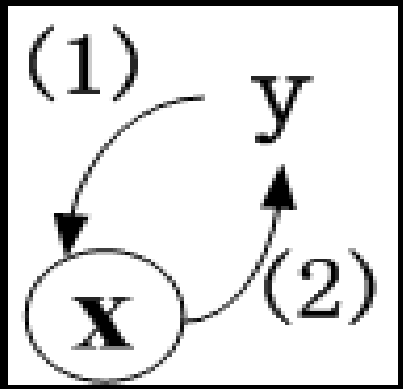


step4 $\begin{cases} left[y] \leftarrow x; & //(1)\text{x作为y的左孩子} \\ p[x] \leftarrow y; & //(2)\text{x的双亲是y} \end{cases}$

x取代β

}

$T(n) = O(1)$



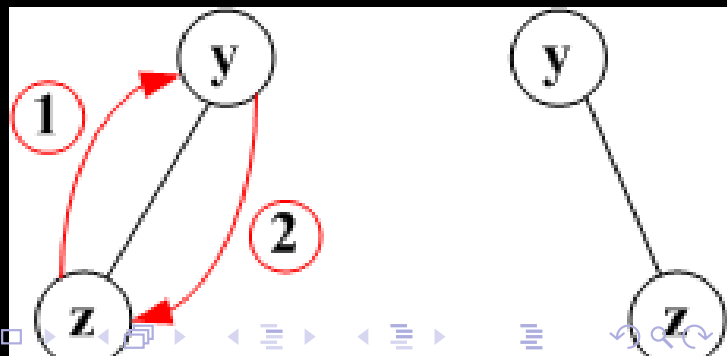
右旋与之对称

§13.3 Insertion

- 思想及步骤
  - Step1：将z插入到RB树中（与二叉搜索树相同），z总是作为叶子插入（不是指外部结点）
  - Step2：将z涂红
  - Step3：使之满足RB性质（着色其它结点及旋转），即调整
- 算法

```
RBInsert(T, z) {
    y ← nil[T];   //y初值
    x ← root[T];   //从根开始找插入位置
    while x ≠ nil[T] do {//y维持是x的双亲
        y ← x;
        if key[z] < key[x] then
            x ← left[x];   //z插入x的左子树
        else x ← right[x]; //z插入x的右子树
    }  //z作为y的孩子插入
```

```
    p[z] = y;   //z的双亲是y
    if  y = nil[T] then    //z是插入空树中
        root[T] = z;       //z是根
    else
        if  key[z] < key[y] then
            left[y] = z    //z作为y的左子插入
        else  right[y] = z;
    left[z] = right[z] = nil[T];
    //z是叶子，假定nil[T]的color是黑色
    color[z] = Red;   //着色
    RBInsertFixup(T, z);
}
```



§13.3 Insertion（续）

调整分析

- ∵z作为红点插入后，其两个外部结点作为左右孩子是黑色的，∴不违反性质1，3，5（非红即黑，叶子为黑（外部点）、黑高定义完整 **指任一结点黑高不变**）
- 只可能违反性质2（当z作为根插到空树中）和性质4（当p[z]为红点）
  1. 若z作为根插入，将其涂黑（违反性质2，恢复）
  2. 若z非根，则P[z]存在
     - ① 若p[z]为黑色，无须调整
     - ② 若p[z]为红色，则调整（违反性质4）

§13.3 Insertion（续）

∵p[z]为红，它不是根（根为黑），　∴p[p[z]]必存在，且为黑色

- 分六种情况调整，其中case1～3为z的双亲p[z]是其是其祖父p[p[z]]的左孩子（右孩子对称）
  - ❖Case1：z的叔叔y是红色

§ 13.3 Insertion（续）

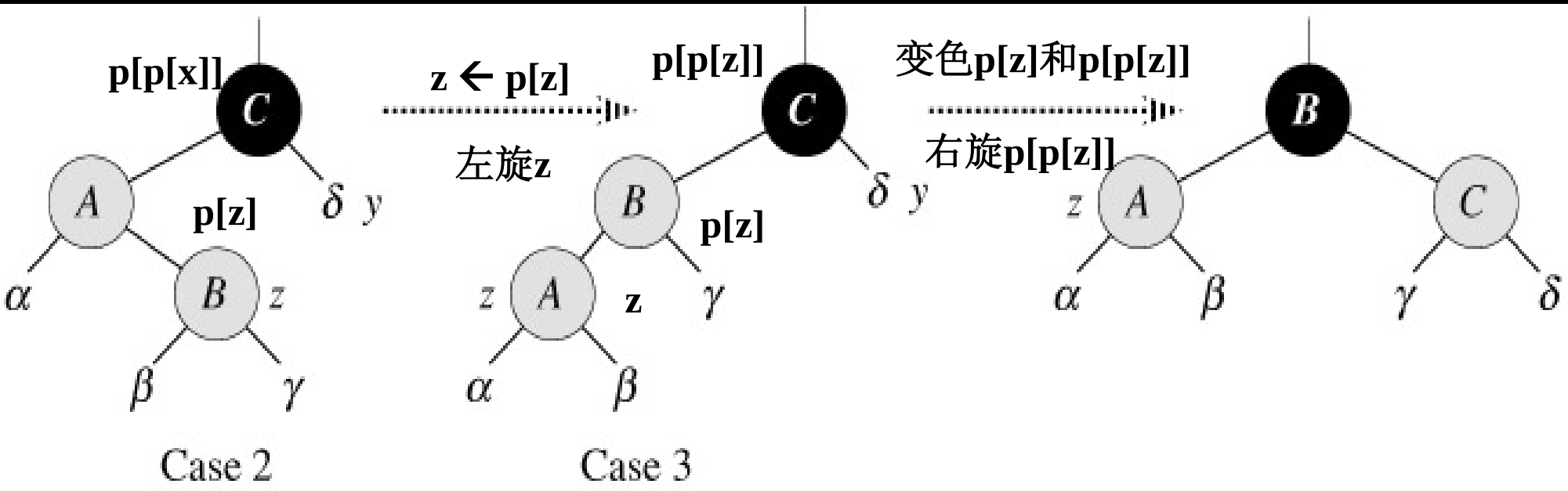
- **Note**

$\left.\begin{array}{l}\alpha,\beta,\gamma,\varepsilon,\delta\text{均有黑根和同样的黑高，}\\ p[p[z]]\text{由黑变红，它的两子由红变黑}\end{array}\right\}\Rightarrow$ 性质5,1,3不变

- 现在可能违反性质4的是p[p[z]]和它的双亲 ∴令它是新的z，继续保持性质5的基础上向上调整，最多至根（若p[p[z]]与其双亲不违反性质4，则终止）
- ***当红色向上传播至根时***，（违反性质2），将其涂为黑色，故树的***黑高最多加1***，但性质5不变

§13.3 Insertion（续）

- Case2：当z的叔叔y是黑色，且z是双亲p[z]的右孩子
- Case3：当z的叔叔y是黑色，且z是双亲p[z]的左孩子



- 上述变换仍保持性质5不变，旋转后又保持性质4. ∴调整终止
- Note：α，β，γ，δ根为黑色
- Case1和Case2，Case3互斥

```
RBInsertFixup (T, z) {
    while (color[p[z]] = Red) do {
//若z为根，则p[z]是nil[T]，其颜色为黑，不进入循环
//若p[z]为黑色，与红z不冲突，亦无须进行调整
      if p[z] = left[p[p[z]]] then {  //z的双亲是其祖父的左孩子
         y = right[p[p[z]]];   //y是x的叔叔
        if color[y] = Red then {   //case1，y为红色
           color[p[z]] = Black;
           color[y] = Black;   //z的父亲及叔叔变黑
           color[p[p[z]]] = Red;   //z的祖父变红
           z ← p[p[z]];  //z上溯至祖父，红色向上传播，新z，循环
      } else {    //case2 or case3，y为黑色
```

```
    if z = right[p[z]] then {//case2，z是双亲右子⇒case3
      z ← p[z];     //z上溯至双亲
      LeftRotate(T, z);    //z左旋
    }; //以下是case3，z是双亲的左子
    color[p[z]] = Black;   //变色
    color[p[p[z]]] = Red; //变色
    RightRotate(T, p[p[z]]);  //p[z]变为黑，退出循环
  }   //end if
 } else {  //case4～6与此对称，right，left交换}
} //end while
color[root[T]] ← Black;
//有两处需要:1).z插入空树，2)case1，使红传到根
}//T（n）=O（lg n）
```

§13.3 Insertion（续）

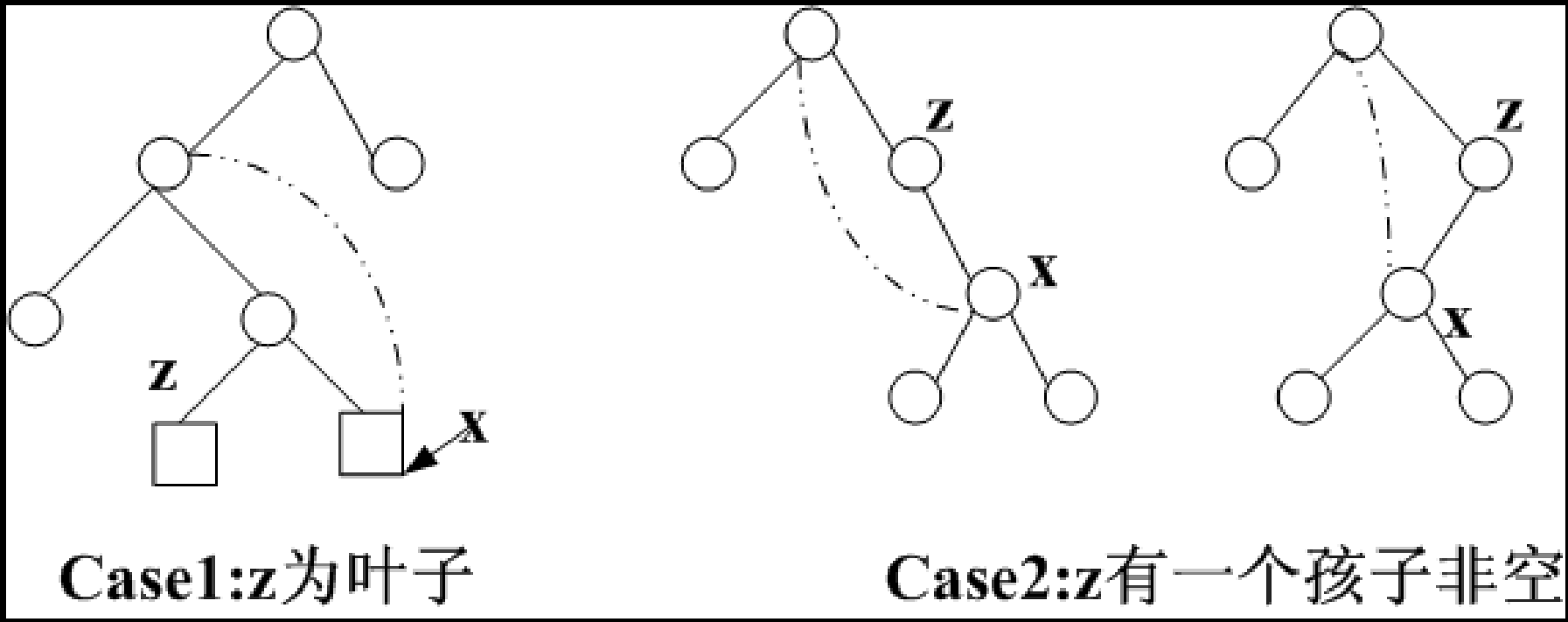
- 时间分析

∵RB树高O(lgn)，插入为O(lgn)，调整为O(lgn)，2个旋转O(1)（至多2个旋转）

总的时间： O(lgn)

§ 13.4 Deletion

- 二叉搜索树上的删除，设被删结点为z



只要短路z即可，（实际上case1是case2的特例），即用z的唯一（可能非空的）孩子x取代之

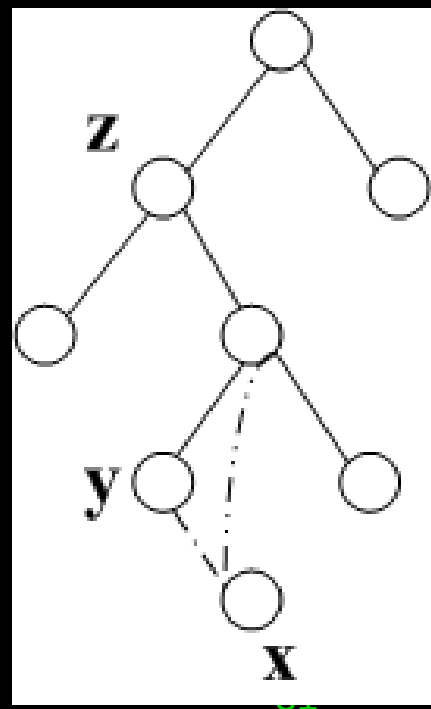
§13.4 Deletion（续）

Case3：z的两子均非空

① 找z的中序后继y，即右子树中最左下结点y，y的左孩子为空；或z的中序前趋，即左子树中最右下结点y，y的右孩子为空

② 因为y最多只有一个非空孩子x（可能两子均空），故可用x取代y，与y的双亲短路，删去y

③ 将y的内容copy至z中



即：case3是将删z的操作变为删y的操作，而删y的操作与case1和case2相同。物理上删y，逻辑上删z。y的内容copy至z*不破坏排序树的性质，三种情况统一到case2处理*

```
RBDelete(T, z) {
  //以下先确定被删结点y
  if (left[z] = nil[T]) or (right[z] = nil[T]) then //case1 or case2
      y ← z; //z至少有一个孩子不存在，直接删z，
             //即z既是物理上，又是逻辑上被删的结点
  else   //两子非空，case3
      y ← TreeSuccessor(z); //转换为case2或case1
  if left[y] ≠ nil[T] then//转换y至多有1个非空孩子
             x ← left[y]
  else       x ← right[y];
 //以下是用x取代y与其y的双亲短路if p[y]存在
  p[x] ← p[y];   //(1).y的双亲变为x的双亲。
  // Note:若无nil[T]，则应判定x ≠ nil?
```

```
if  p[y] = nil[T]  then   //y是根
        root[T] ← x   //x为根
else   //y非根(2)
     if  y = left[ p[y]] then //y是其双亲左子
          left[ p[y]] ← x
     else  right[ p[y]] ← x
if  y ≠ z  then copy y的关键字和数据域到z;
if  color[y] = Black  then  //1)y为红点时，删去不改变黑高;
    //2)也不会使相邻红点出现;3)y不可能是根，根仍黑
    //  ∴ 不破坏RB性质, 无须调整
        RBDeleteFixup(T, x);
return y;  //若要返回被删结点的其它数据，应将z copy到 y
}
```

§13.4 Deletion（续）

- **调整方法 RBDeleteFixup（T,x）//恢复RB性质**

删y后可能破坏{ 性质2，当y是根，x是红点
性质4，x和p[y]均为红
性质5，包含y的路径上少一黑点

x={ y的唯一孩子 if y有一孩子为空 //case2 or case3
哨兵nil[T] otherwise //case1 or case3

∵被删结点y为黑色，故对于任何y的祖先结点而言，它们到叶子的路径上少了一个黑点，违反性质5

∴*想象*将y的黑色涂到x上，

若x原为红色，则x变为黑色 } 即x的“额外黑色”，使经过x的路径
若x原为黑色，则x变为双黑 } 包含的黑结点数加1，以此维护性质5

§13.4 Deletion（续）

但是*双黑结点违反性质1（非红即黑）*故调整时试图恢复性质1，消去双黑结点的一层黑

情况1： 若x是根，不影响性质5，直接移去一层黑（黑高减1）

情况2： 若x原为红，则将y的黑色加到x上（注意y只有一非空孩子），终止

情况3： 若x非根且原为黑，则通过变色和旋转将x上额外的一层黑色向树*上方移动*直至x指向某个红点或x指向根时终止，调整共分为八种情况：

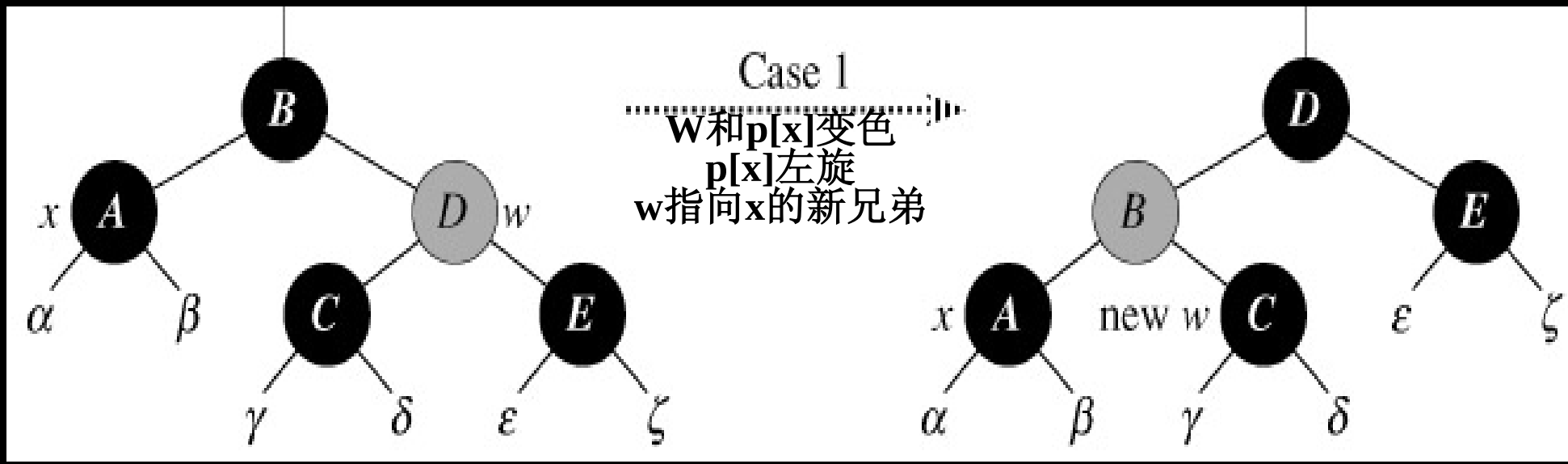
Case1～4与Case5～8对称，前者x是p[x]的左子，后者x是p[x]的右子

§13.4 Deletion（续）

❖Case1: x的兄弟w是红色

∵x非根， ∴必有兄弟w。（∵严格二叉树）又∵w不可能是nil[T]，即它可能为红点

∵w为红， ∴p[x]必为黑

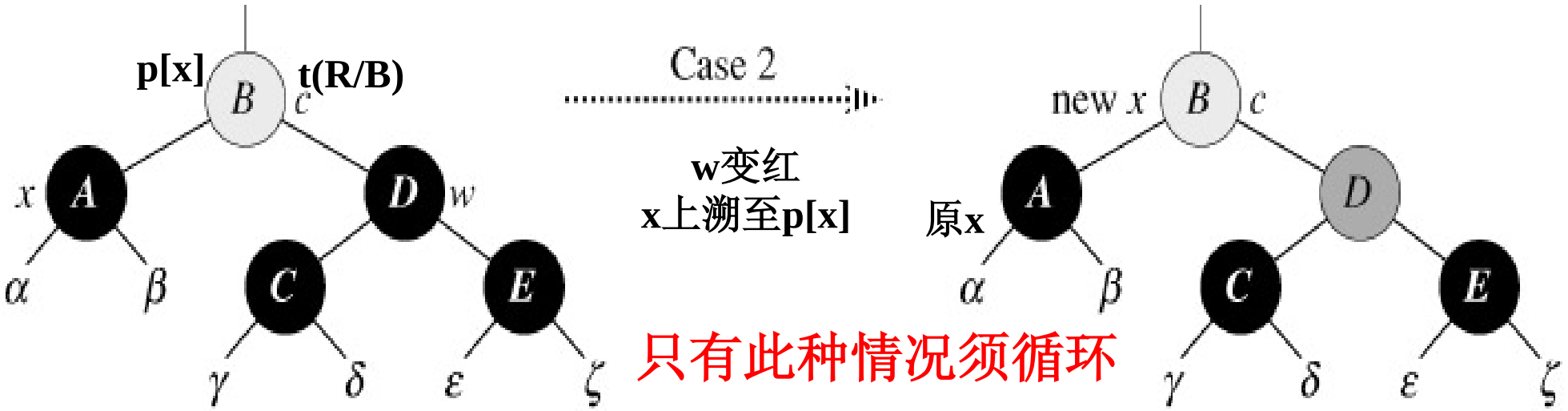


调整后子树根的黑高不变，BST性质不变

§13.4 Deletion（续）

Case2～Case4：x的兄弟w为黑色，由w的孩子颜色区别

- Case2：x黑兄弟w的两孩子为黑（w至少有一红孩子是case3，case4）



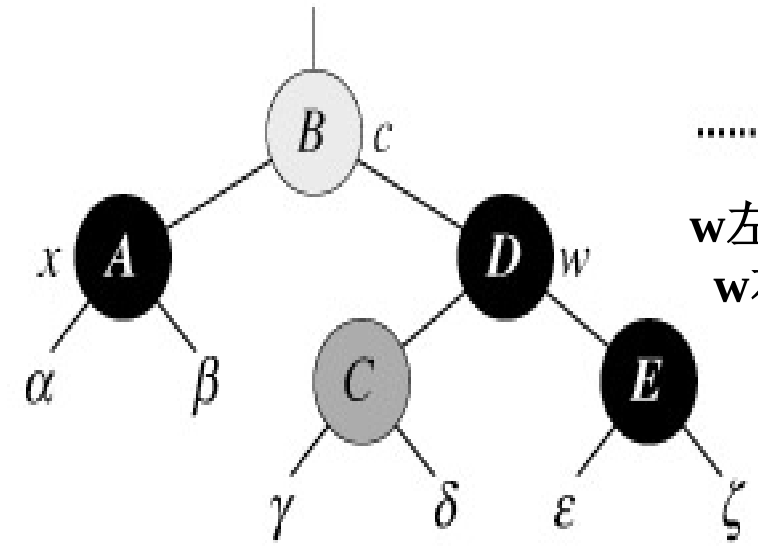
只有此种情况须循环

∵w有两黑子，故w可变红

∴令A及D的黑色涂到B上，现在A是单黑结点。若B原为红（例如从case1而来）则变为step2，终止；否则继续脱去B上的双黑

§13.4 Deletion（续）

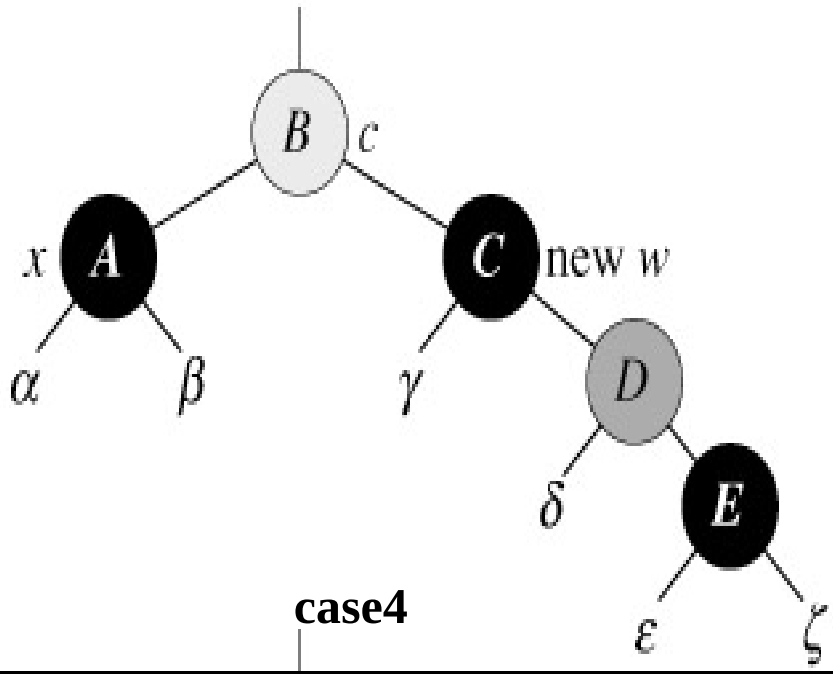
- Case3：x黑兄弟w的右子为黑（左子必为红）



case3

Case 3

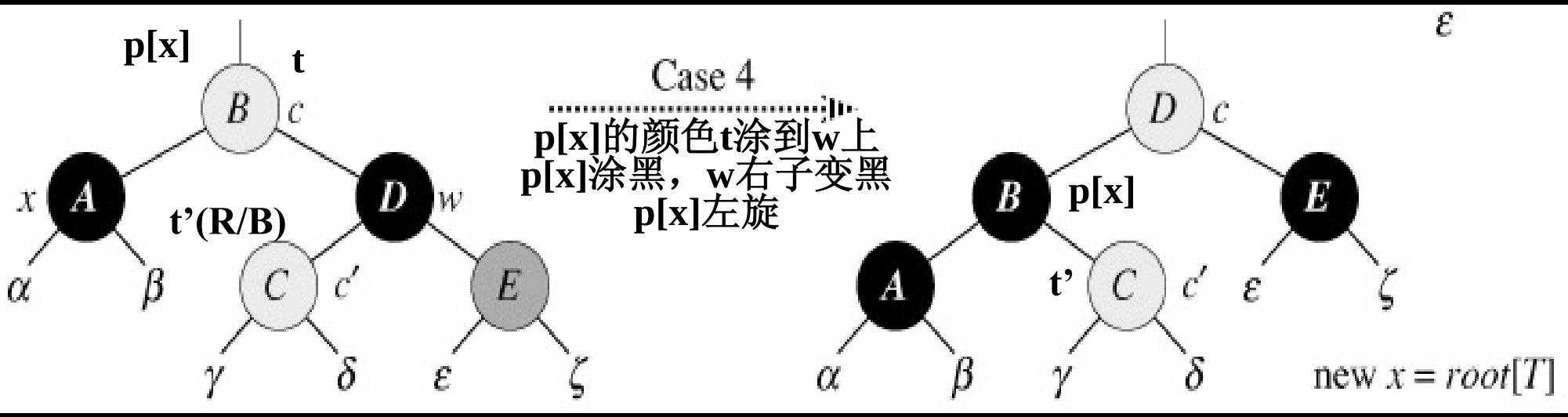
w左子c变黑，w变红
w右旋，w指向x的新兄弟



case4

§13.4 Deletion（续）

❖ Case4：x黑兄弟w的右子为红，左子可红可黑



∵γ，δ，ε，ζ之上黑点数目变换前后不变，**而α，β之上多了一个黑点**，相当于x上多余黑点计数被移去，故调整终止

- Ex13.4-5

```
RBDeleteFixup(T, x) {
    while (x ≠ root[T]) and (color[x] = Black) do {
    //x非根，双黑，被删结点y的黑色涂到x上
        if   x = left[p[x]]  then {//x是双亲左子，case1～case4
            w ← right[p[x]];    //w是x的兄弟
            if   color[w] = Red  then {    //case1
                    color[w] = Black;
                    color[p[x]] = Red;   //x的兄弟及双亲变色
                    LeftRotate(T, p[x]);
                    w ← right[p[x]];      //x的新兄弟w
            }  //变为case2或3,4   新w为黑
```

```
if (color[left[w]] = Black) and (color[right[w]] = Black) then {
    //case2, w两子黑
    color[w] = Red;   //x及w上的黑色涂到p[x]
    x = p[x];
   //p[x]为新x，若原p[x]为黑，它是双黑继续循环，否则终止
 } else {   //case3~4,w的两孩子至少有一个为红色
       if (color[right[w]] = Black) {
           //case3, w右子为黑，左子必为红
           color[left[w]] = Black;
           color[w] = Red;   //w及左子变色
           RightRotate(T, w);
           w ← right[p[x]];   //w指向x的新兄弟
        } //case3处理后变为case4，w右子为红，左子可红可黑
```

```
        color[w] = color[ p[x]];   //p[x]的颜色涂到w上
        color[ p[x]] = Black;      //p[x]涂黑
        color[right[w]] = Black; //w右子由红变黑
        LeftRotate(T, p[x]);    //B-D左旋，移去多余黑色
        x = root[T];   //退出循环，注意x本身是黑色
    } //end  case3 ~ 4
  } else { //x是其双亲p[x]的右子为case5 ~ case8，与上对称
 } //endwhile
 color[x] = Black;
}
```

至多3个旋转

$T(n) = O(\lg n)$