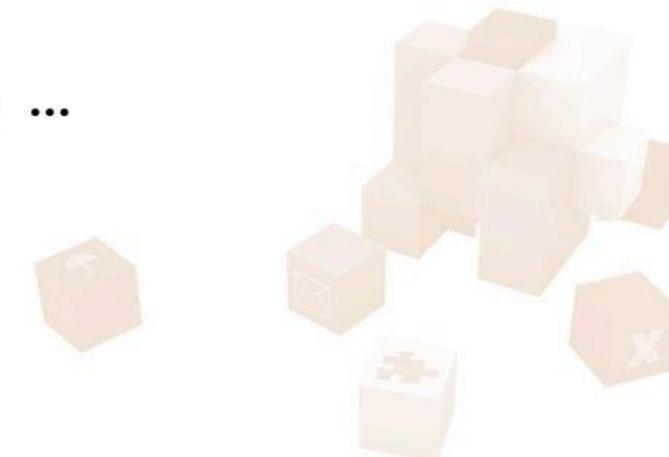
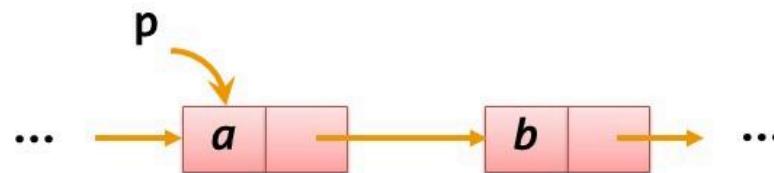


## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### ➤ 单链表的特点

当访问过一个结点后，只能接着访问它的后继结点，而无法访问它的前驱结点。





## 5 “数据结构”的学习方法



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### ④ 演绎和归纳相结合。



培根：方法是旧的，问题是新的





## 1.1 什么是数据结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 1.1.1 数据结构的定义

#### 数据结构中的几个概念

##### ➤ 数据：

所有能够输入到计算机中，且能被计算机处理的符号的集合。





## 1.1 什么是数据结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 1.1.1 数据结构的定义

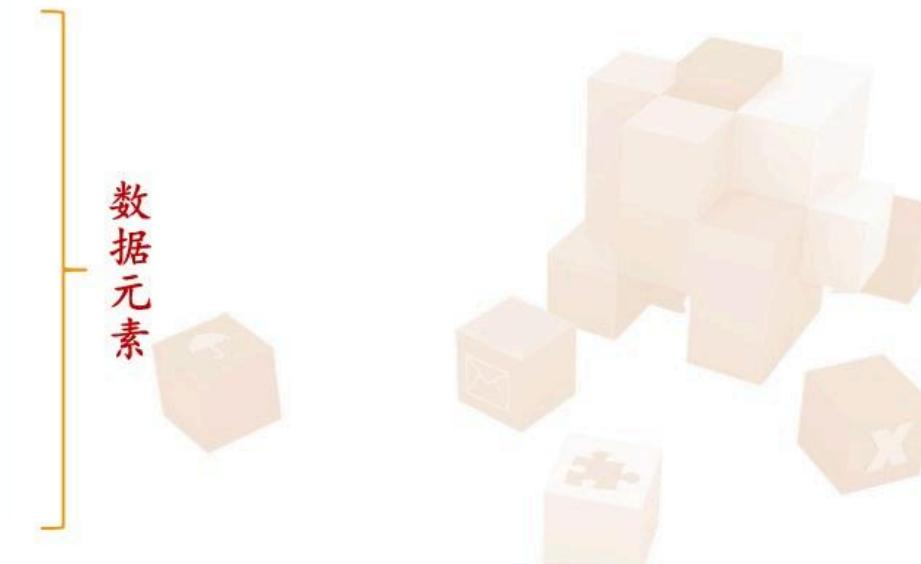
结构化数据示例

一个学生表

学号	姓名	性别	班号
1	张斌	男	9901
8	刘丽	女	9902
34	李英	女	9901
20	陈华	男	9902
12	王奇	男	9901
26	董强	男	9902
5	王萍	女	9901

← 数据项(用于描述数据元素)

数据元素



## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

#### 数据结构中的几个概念

- **数据元素：**是数据（集合）中的一个“个体”，它是数据的基本单位。
- **数据项：**数据项是用来描述数据元素的，它是数据的最小单位。
- **数据对象：**具有相同性质的若干个数据元素的集合，如整数数据对象是所有整数的集合。

默认情况下，数据结构中讨论的数据都是**数据对象**。



## 1.1 什么是数据结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 1.1.1 数据结构的定义

#### 数据结构中的几个概念

- **数据结构：**是指带结构的数据元素的集合。

$$\text{数据结构} = \text{数据对象} + \text{结构}$$

↑   ↑  
相同性质                                  数据元素之  
的数据元                                  间的关系构  
素的集合                                  成结构

## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

一个数据结构的构成



## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

#### 1、数据的逻辑结构表示

每个关系的用若干个序偶来表示：

- ◆ 序偶 $\langle x, y \rangle$  ( $x, y \in D$ )  $\Rightarrow$   $x$ 为第一元素， $y$ 为第二元素。
- ◆  $x$ 为 $y$ 的（直接）前驱元素。
- ◆  $y$ 为 $x$ 的（直接）后继元素。
- ◆ 若某个元素没有前驱元素，则称该元素为开始元素；若某个元素没有后继元素，则称该元素为终端元素。

序偶 $\langle x, y \rangle$ 表示 $x$ 、 $y$ 是有向的，序偶 $(x, y)$ 表示 $x$ 、 $y$ 是无向的

## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

#### 1、数据的逻辑结构表示

例如，如下数据为一个矩阵：

2	6	3	1
8	12	7	4
5	10	9	11

对应的二元组表示为  $B=(D, R)$ ，其中：

$$D=\{2, 6, 3, 1, 8, 12, 7, 4, 5, 10, 9, 11\}$$

$R=\{r1, r2\}$  其中， $r1$ 表示行关系， $r2$ 表示列关系

$r1=\{\langle 2, 6 \rangle, \langle 6, 3 \rangle, \langle 3, 1 \rangle, \langle 8, 12 \rangle, \langle 12, 7 \rangle, \langle 7, 4 \rangle,$   
 $\langle 5, 10 \rangle, \langle 10, 9 \rangle, \langle 9, 11 \rangle\}$

$r2=\{\langle 2, 8 \rangle, \langle 8, 5 \rangle, \langle 6, 12 \rangle, \langle 12, 10 \rangle, \langle 3, 7 \rangle, \langle 7, 9 \rangle,$   
 $\langle 1, 4 \rangle, \langle 4, 11 \rangle\}$

## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

#### 2、数据的存储结构表示

##### ① 学生表存储结构1— 结构体数组

数据在计算机存储器中的存储方式就是存储结构。它是面向程序员的。



设计存储结构的这种映射应满足两个要求：

- 存储所有元素
- 存储数据元素间的关系

## 1.1 什么是数据结构

### 1.1.1 数据结构的定义

#### 2、数据的存储结构表示

##### ② 学生表存储结构2—链表

存放学生表的链表的结点类型**StudType**声明如下：

```
typedef struct studnode
{
    int no;                                //存储学号
    char name[8];                           //存储姓名
    char sex[2];                            //存储性别
    char class[4];                           //存储班号
    struct studnode *next;                 //存储指向下一个学生的指针
} StudType;
```



## 1.1 什么是数据结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 1.1.1 数据结构的定义

#### 结论

- 同一逻辑结构可以对应多种存储结构。
- 同样的运算，在不同的存储结构中，其实现过程是不同的。

## 1.1 什么是数据结构

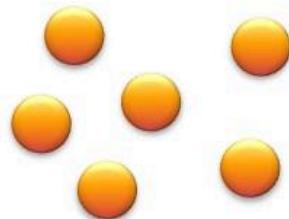
### 1.1.2 逻辑结构类型

各种各样的数据呈现出不同的逻辑结构，归纳为**4种**。

#### 1、集合

元素之间关系：无。

特点：数据元素之间除了“属于同一个集合”的关系外，别无其他逻辑关系。是最松散的，不受任何制约的关系。



## 1.1 什么是数据结构

### 1.1.2 逻辑结构类型

#### 2、线性结构

元素之间关系：一对一。

特点：开始元素和终端元素都是唯一的，除此之外，其余元素都有且仅有一个前驱元素和一个后继元素。



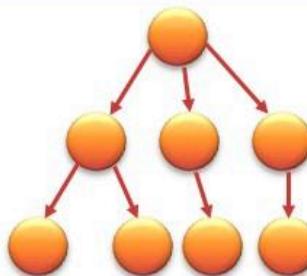
## 1.1 什么是数据结构

### 1.1.2 逻辑结构类型

#### 3、树形结构

元素之间关系：一对多。

特点：开始元素唯一，终端元素不唯一。除终端元素以外，每个元素有一个或多个后继元素；除开始元素外，每个元素有且仅有一个前驱元素。



## 1.1 什么是数据结构

### 1.1.2 逻辑结构类型

【例1.3】有一种数据结构  $B2 = (D, R)$ ，其中

$$D = \{48, 25, 64, 57, 82, 36, 75\}$$

$$R = \{r_1, r_2\}$$

$$\begin{aligned} r_1 = & \{\langle 25, 36 \rangle, \langle 36, 48 \rangle, \langle 48, 57 \rangle, \langle 57, 64 \rangle, \\ & \langle 64, 75 \rangle, \langle 75, 82 \rangle\} \end{aligned}$$

$$\begin{aligned} r_2 = & \{\langle 48, 25 \rangle, \langle 48, 64 \rangle, \langle 64, 57 \rangle, \langle 64, 82 \rangle, \\ & \langle 25, 36 \rangle, \langle 82, 75 \rangle\} \end{aligned}$$

画出其逻辑结构表示，指出是什么类型？



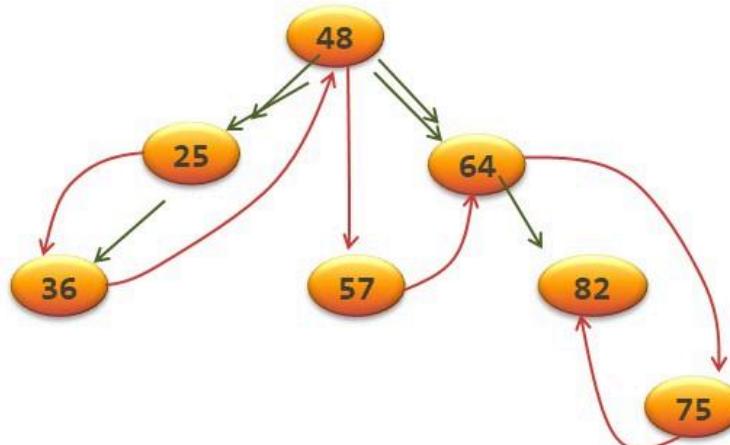
## 1.1 什么是数据结构

### 1.1.2 逻辑结构类型

解

B2的逻辑结构图如下。

$r_1$ 关系表示  $\rightarrow r_1$ 为线性结构  
 $r_2$ 关系表示  $\rightarrow r_2$ 为树形结构



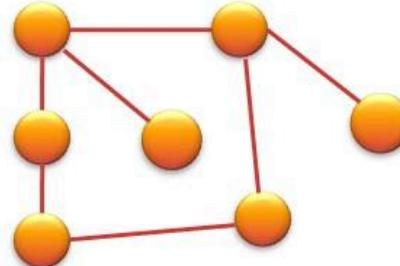
## 1.1 什么是数据结构

### 1.1.2 逻辑结构类型

#### 4、图形结构

元素之间关系：多对多。

特点：所有元素都可能有多个前驱元素和多个后继元素。



## 1.1 什么是数据结构

### 1.1.3 存储结构类型

在软件开发中，人们设计了各种存储结构。

归纳为**4**种基本的存储结构。

- 顺序存储结构
- 链式存储结构
- 索引存储结构
- 哈希（散列）存储结构

## 2.2 线性表的顺序存储结构

### 2.2.1 线性表的顺序存储—顺序表

#### ➤ 顺序表类型定义

这里，假设**ElemType**为**char**类型

```
typedef struct
{ ElemType data[MaxSize];
  int length;
} SqList; //顺序表类型
```

其中**data**成员存放元素，**length**成员存放线性表的实际长度。

说明：注意逻辑位序和物理位序相差1。



## 1.3 算法分析

### 1.3.1 算法分析概述

分析算法占用的资源



**算法分析目的：**分析算法的时空效率以便改进算法性能。

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

**【例1-6】**求两个 $n$ 阶方阵的相加 $C=A+B$ 的算法如下，分析其时间复杂度。

```
#define MAX 20      //定义最大的方阶
void matrixadd(int n, int A[MAX][MAX], int B[MAX][MAX],
               int C[MAX][MAX])
{
    int i, j;
    for (i=0;i<n;i++)                      //①
        for (j=0;j<n;j++)                  //②
            C[i][j]=A[i][j]+B[i][j];          //③
}
```

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

```
#define MAX 20 //定义最大的方阶

void matrixadd(int n, int A[MAX][MAX],
               int B[MAX][MAX], int C[MAX][MAX])
{
    int i, j;

    for (i=0; i<n; i++) //①
        for (j=0; j<n; j++) //②
            C[i][j] = A[i][j] + B[i][j]; //③
}
```

解：除变量定义语句外，该算法包括3个可执行语句①、②和③。

——频度为 $n+1$ ，循环体执行 $n$ 次

——频度为 $n(n+1)$

——频度为 $n^2$

所有语句频度之和为：

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

## 1.3 算法分析

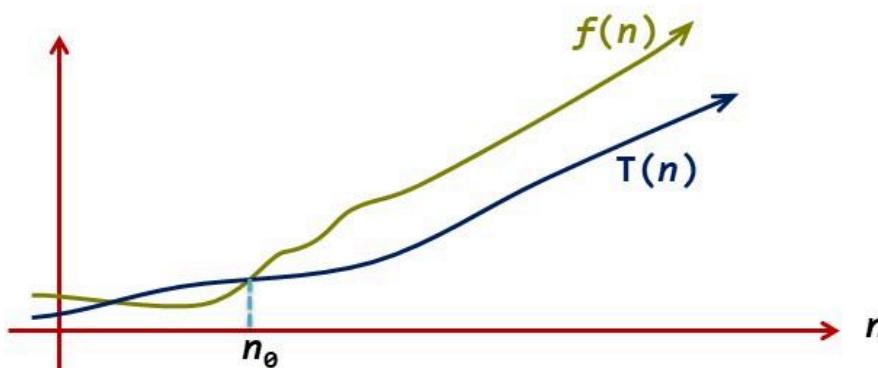
### 1.3.2 算法时间复杂度分析

#### ② 算法的执行时间用时间复杂度来表示

算法中执行时间  $T(n)$  是问题规模  $n$  的某个函数  $f(n)$ ，记作：

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

记号 “ $O$ ” 读作 “**大O**”，它表示随问题规模  $n$  的增大算法执行时间的增长率和  $f(n)$  的增长率相同。  $\Rightarrow$  趋势分析



## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

#### 简化的算法时间复杂度分析

算法中的基本操作一般是最深层循环内的原操作。

算法执行时间大致 = 基本操作所需的时间 × 其运算次数。



在算法分析时，计算  $T(n)$  时仅仅考虑基本操作的运算次数。

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

**【例1.6】**求两个 $n$ 阶方阵的相加 $C=A+B$ 的算法如下，分析其时间复杂度。

```
#define MAX 20 //定义最大的方阶
void matrixadd(int n, int A[MAX][MAX], int B[MAX][MAX],
               int C[MAX][MAX])
{
    int i, j;
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            C[i][j]=A[i][j]+B[i][j];
}
```

基本操作

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

```
void matrixadd(int n, int A[MAX][MAX], int B[MAX][MAX],
               int C[MAX][MAX])
{
    int i, j;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            C[i][j]=A[i][j]+B[i][j];
}
```

**解：**该算法中的基本操作是两重循环中最深层的语句

$C[i][j]=A[i][j]+B[i][j]$ ，分析它的频度，即：

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n \sum_{i=0}^{n-1} 1 = n * n = n^2 \\ &= O(n^2) \end{aligned}$$

这种简化的时间复杂度分析方法得到的结果相同，但分析过程更简单。

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

示例

下列程序段的时间复杂度是（ ）。

```
count=0;  
for(k=1;k<=n;k*=2)  
    for(j=1;j<=n;j++)  
        count++;
```

基本操作

A. $O(\log_2 n)$

B. $O(n)$

C. $O(n \log_2 n)$

D. $O(n^2)$

说明：本题为2014年全国考研题

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

#### 最好、最坏和平均时间复杂度分析

**定义：**设一个算法的输入规模为 $n$ ,  $D_n$ 是所有输入的集合，任一输入 $I \in D_n$ ,  $P(I)$ 是 $I$ 出现的概率，有  $\sum_{I \in D_n} P(I) = 1$ ,  $T(I)$ 是算法在输入 $I$ 下的执行时间，则算法的**平均时间复杂度**为：

$$E(n) = \sum_{I \in D_n} P(I) \times T(I)$$

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

例如，10个1~10的整数序列递增排序 ( $n=10$ )：

$$I_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$I_2 = \{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\}$$

...

$$I_m = \{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$$

构成  $D_n$



所有可能的初始序列有  $m$  个， $m=10!$   $\Rightarrow P(I)=1/m$

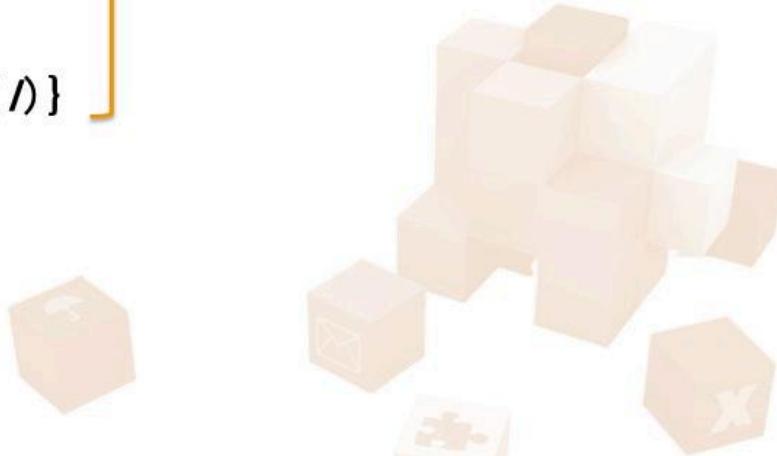
## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

算法的最坏时间复杂度为： $W(n) = \underset{I \in D_n}{\text{MAX}} \{ T(I) \}$

算法的最好时间复杂度为： $B(n) = \underset{I \in D_n}{\text{MIN}} \{ T(I) \}$

一种或几种特殊情况



## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

**【例1.8】** 以下算法用于求含 $n$ 个整数元素的序列中前 $i$  ( $1 \leq i \leq n$ ) 个元素的最大值。分析该算法的最好、最坏和平均时间复杂度。

```
int fun(int a[], int n, int i)
{
    int j, max=a[0];
    for (j=1;j<=i-1;j++)
        if (a[j]>max) max=a[j];
    return(max);
}
```



## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

解：算法的主要时间花费在元素比较上：

- $i$ 的取值范围为 $1 \sim n$ , 共 $n$ 种情况。
- 在等概率情况（每种情况的概率为 $1/n$ ）。
- 求前 $i$ 个元素的最大值时,  $\text{max}=a[0]$ , 将其与 $a[1..i-1]$ 元素进行比较。比较次数 $=(i-1)-1+1=i-1$ 次。

```
int fun(int a[], int n, int i)
{
    int j, max=a[0];
    for (j=1;j<=i-1;j++)
        if (a[j]>max) max=a[j];
    return(max);
}
```

$$T(n) = \sum_i^n \frac{1}{n} \times (i-1) = \frac{1}{n} \sum_i^n (i-1) = \frac{n-1}{2}$$

$= O(n)$  ← 平均时间复杂度

## 1.3 算法分析

### 1.3.2 算法时间复杂度分析

```
int fun(int a[], int n, int i)
{
    int j, max=a[0];
    for (j=1;j<=i-1;j++)
        if (a[j]>max) max=a[j];
    return(max);
}
```

- $i=1$ 时，比较次数为0  $\Rightarrow$  算法的最好复杂度  $B(n)=O(1)$ 。
- $i=n$ 时，比较次数为  $n-1$   $\Rightarrow$  算法的最坏复杂度  $W(n)=O(n)$ 。

**单选题 1分**

某算法的时间复杂度为 $O(n^2)$ ，表明该算法的（ ）。

- A 问题规模是 $n^2$
- B 执行时间等于 $n^2$
- C 执行时间与 $n^2$ 成正比
- D 问题规模与 $n^2$ 成正比

## 2.2 线性表的顺序存储结构

### 2.2.1 线性表的顺序存储—顺序表

线性表的顺序存储结构：

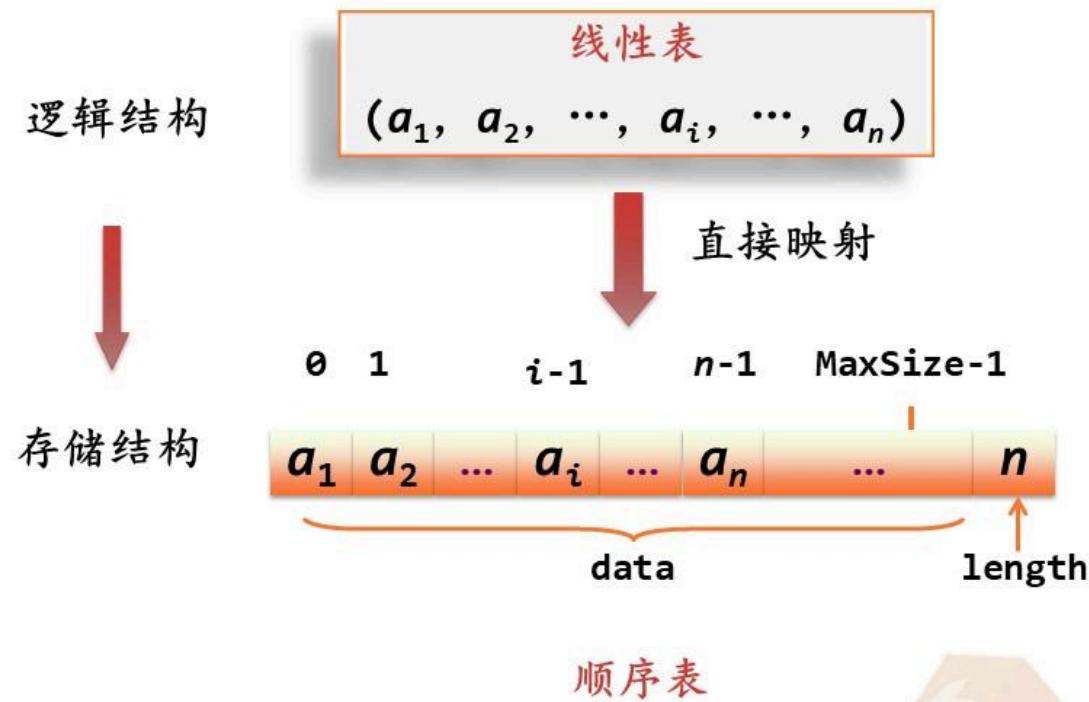
把线性表中的所有元素按照顺序存储方法进行存储。



按逻辑顺序依次存储到存储器中一片连续的存储空间中。

## 2.2 线性表的顺序存储结构

### 2.2.1 线性表的顺序存储—顺序表



## 2.2 线性表的顺序存储结构

### 2.2.1 线性表的顺序存储—顺序表

#### ➤ 顺序表类型定义

这里，假设**ElemType**为**char**类型

```
typedef struct
{ ElemType data[MaxSize];
  int length;
} SqList; //顺序表类型
```

其中**data**成员存放元素，**length**成员存放线性表的实际长度。

说明：注意逻辑位序和物理位序相差1。



## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### 1、建立顺序表

$a[0..n-1]$   $\Leftrightarrow$  顺序表 L - 整体创建顺序表。

```
void CreateList(SqList * &L, ElemType a[], int n)
{ int i=0,k=0;
  L=(SqList *)malloc(sizeof(SqList));
  while (i<n)           //i扫描a中元素
  { L->data[k]=a[i];
    k++; i++;           //k记录插入到L中的元素个数
  }
  L->length=k;
}
```

传递顺序  
表指针

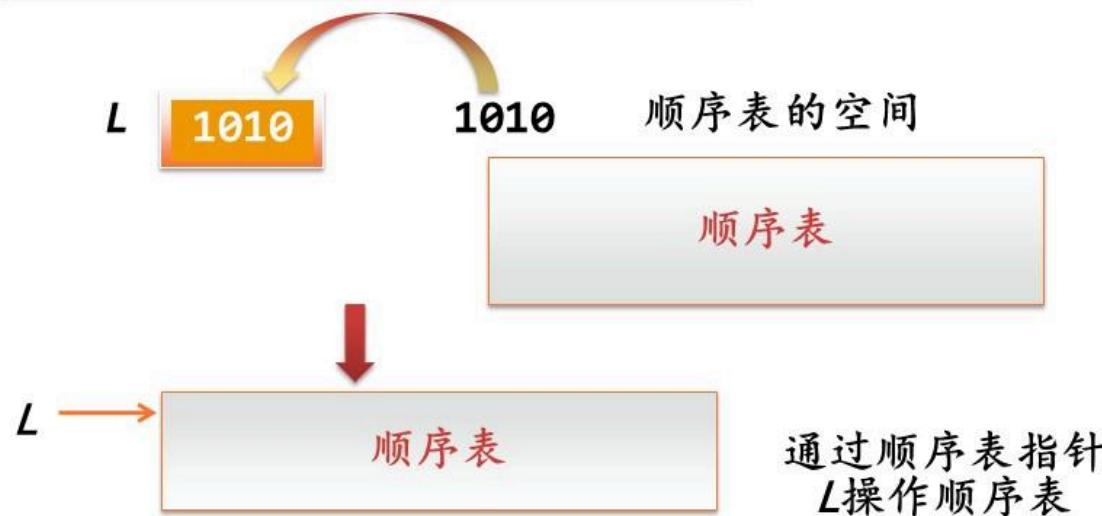
## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### 算法参数说明

##### ① 顺序表指针的含义

```
SqList *L;  
L=(SqList *)malloc(sizeof(SqList));
```



## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### ② 顺序表指针引用

```
void CreateList(SqList *&L, ElemType a[], int n)
```



引用参数：将执行结果回传给实参

- 引用符号“`&`”放在形参`L`的前面。
- 输出型参数均为使用“`&`”，不论参数值是否改变。

## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### 2、顺序表基本运算算法

##### (1) 初始化线性表 `InitList(L)`

构造一个空的线性表 L。实际上只需将 `length` 成员设置为 0 即可。

```
void InitList(SqList *L)
{
    L=(SqList *)malloc(sizeof(SqList));
    //分配存放线性表的顺序表空间
    L->length=0;
}
```

## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### (2) 销毁线性表DestroyList(L)

释放线性表L占用的内存空间。

```
void DestroyList(Sqlist *&L)
{
    free(L);
}
```

$L \rightarrow$

顺序表

顺序表采用指针传递，有两个优点：

- 更清楚看到顺序表创建和销毁过程（malloc/free）。
- 在算法的函数之间传递更加节省空间（在函数体内不必创建值形参即整个顺序表的副本）。

free(L)释放L所指向的空间

## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### (3) 判定是否为空表 `ListEmpty(L)`

回一个值表示L是否为空表。若L为空表，则返回**true**，否则返回**false**。

```
bool ListEmpty(SqList *L)
{
    return(L->length==0);
}
```



## 2.2 线性表的顺序存储结构

### 2.2.2 顺序表运算的实现

#### (4) 求线性表的长度 **ListLength(L)**

返回顺序表L的长度。实际上只需返回**length**成员的值即可。

```
int ListLength(SqList *L)
{
    return(L->length);
}
```



## 2.3 线性表的链式存储结构

### 2.3.1 线性表的链式存储—链表

线性表中每个结点有唯一的前驱结点和后继结点。

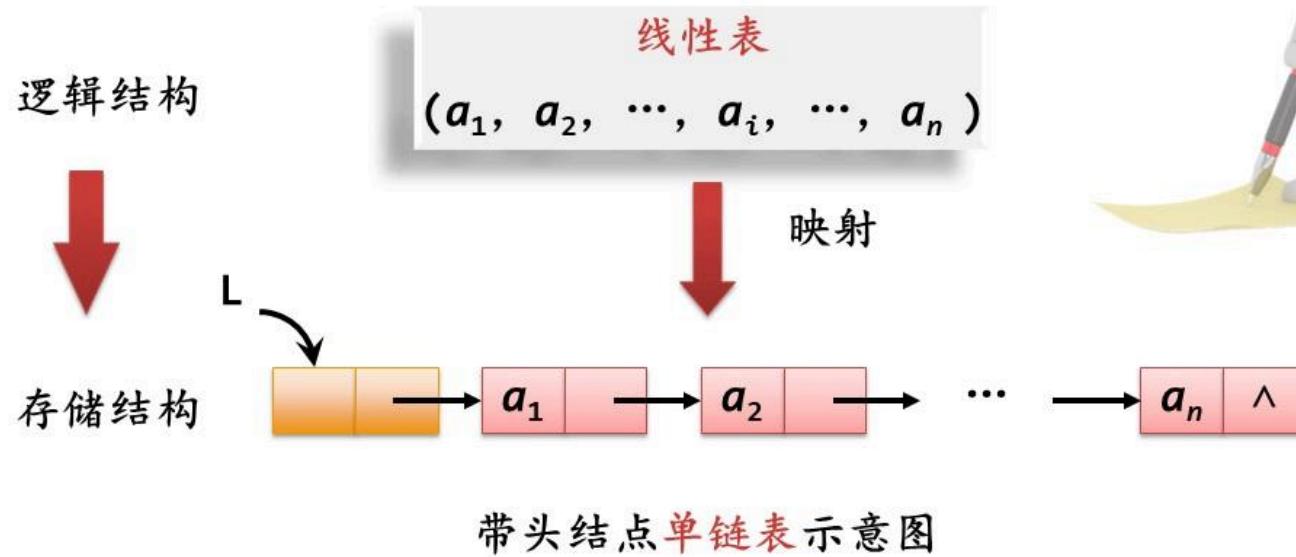


设计链式存储结构时，每个逻辑结点存储单独存储，为了表示逻辑关系，增加指针域。

- 每个物理结点增加一个指向后继结点的指针域  $\Leftrightarrow$  单链表。
- 每个物理结点增加一个指向后继结点的指针域和一个指向前驱结点的指针域  $\Leftrightarrow$  双链表。

## 2.3 线性表的链式存储结构

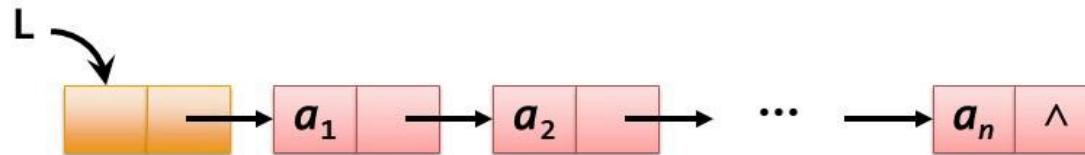
### 2.3.1 线性表的链式存储—链表



## 2.3 线性表的链式存储结构

### 2.3.1 线性表的链式存储—链表

带头结点单链表



单链表增加一个头结点的**优点**如下：

- 首结点的操作和表中其他结点的操作相一致，无需进行特殊处理；
- 无论链表是否为空，都有一个头结点，因此**空表和非空表的处理也就统一了。**



## 2.3 线性表的链式存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 2.3.1 线性表的链式存储—链表



思考题：

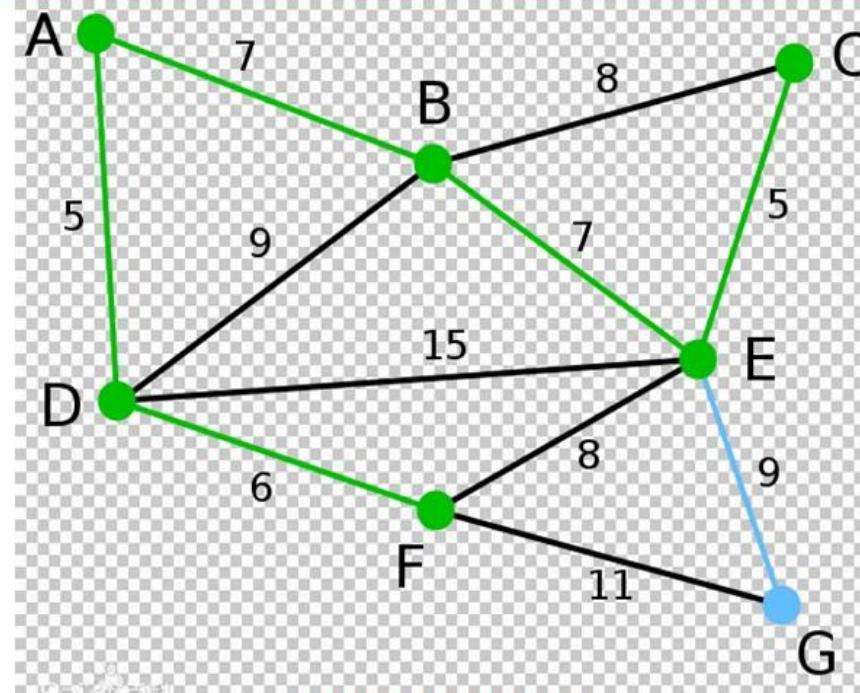
线性表的顺序存储结构和链式存储结构的差异？



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$$U=\{D, A, F, B, E, C\}$$

普里姆算法求解最小生成树的过程

## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 1、插入结点和删除结点操作

##### (1) 插入结点

**插入操作：**将值为 $x$ 的新结点 $s$ 插入到 $p$ 结点之后。

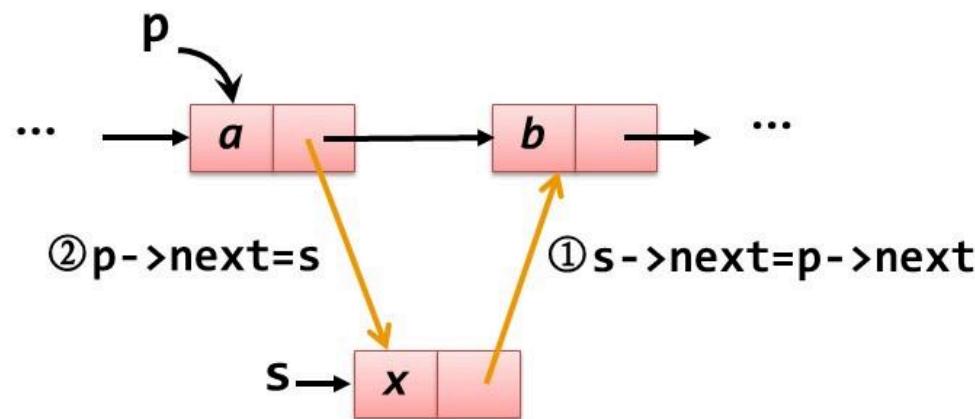
**特点：**只需修改相关结点的指针域，不需要移动结点。



## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 单链表插入结点演示



插入操作语句描述如下：

```
s->next = p->next;  
p->next = s;
```

## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 1、插入结点和删除结点操作

##### (2) 删除结点

**删除操作：**删除p结点之后的一个结点。

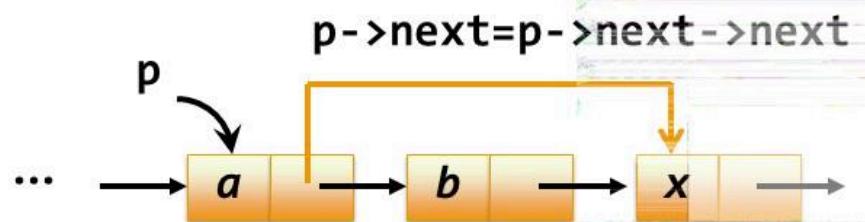
**特点：**只需修改相关结点的指针域，不需要移动结点。



## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 单链表删除结点演示



删除操作语句描述如下：

*p->next = p->next->next;*

## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 2、建立单链表

先考虑如何整体建立单链表。



建立单链表的常用方法有两种。

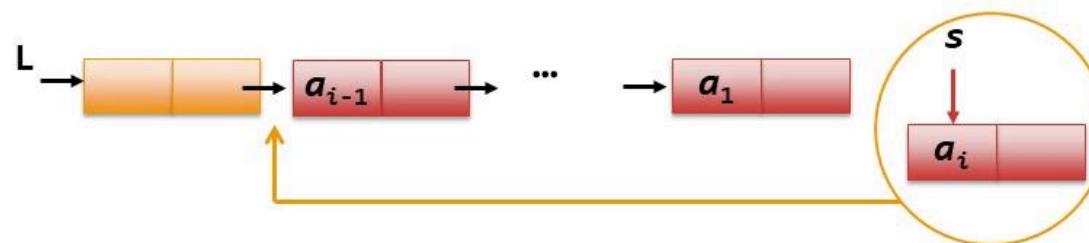
## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 2、建立单链表

##### (1) 头插法建表

- 从一个空表开始，创建一个头结点。
- 依次读取字符数组 $a$ 中的元素，生成新结点
- 将新结点插入到当前链表的表头上，直到结束为止。



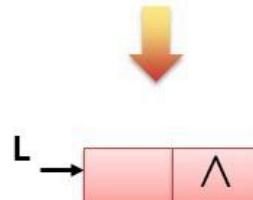
注意：链表的结点顺序与逻辑次序相反。

## 2.3 线性表的链式存储结构

### 2.3.2 单链表

头插法建表算法如下：

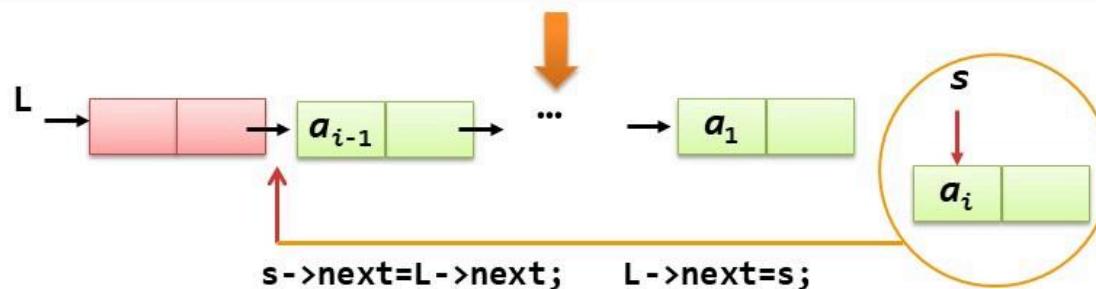
```
void CreateListF(LinkNode *&L, ElemType a[], int n)
{
    LinkNode *s;
    int i;
    L=(LinkNode *)malloc(sizeof(LinkNode));
    L->next=NULL;           //创建头结点，其next域置为NULL
```



## 2.3 线性表的链式存储结构

### 2.3.2 单链表

```
for (i=0;i<n;i++)          //循环建立数据结点
{
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=a[i];           //创建数据结点s
    s->next=L->next;        //将s插在原开始结点之前，头结点之后
    L->next=s;
}
}
```



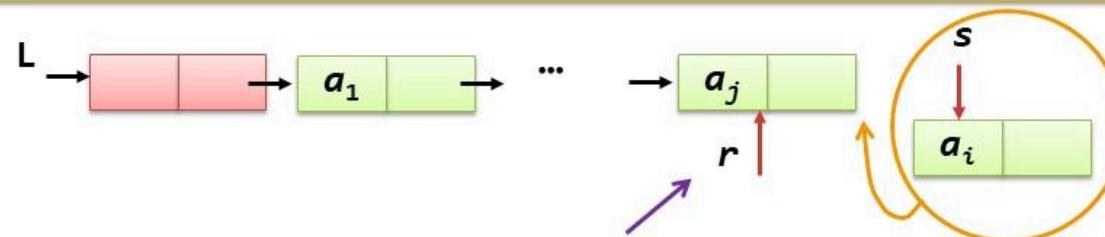
## 2.3 线性表的链式存储结构

### 2.3.2 单链表

#### 2、建立单链表

##### (2) 尾插法建表

- 从一个空表开始，创建一个头结点。
- 依次读取字符数组 $a$ 中的元素，生成新结点
- 将新结点插入到当前链表的表尾上，直到结束为止。



增加一个尾指针  $r$ ，使其始终指向当前链表的尾结点

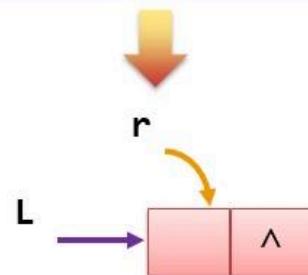
注意：链表的结点顺序与逻辑次序相同。

## 2.3 线性表的链式存储结构

### 2.3.2 单链表

尾插法建表算法如下：

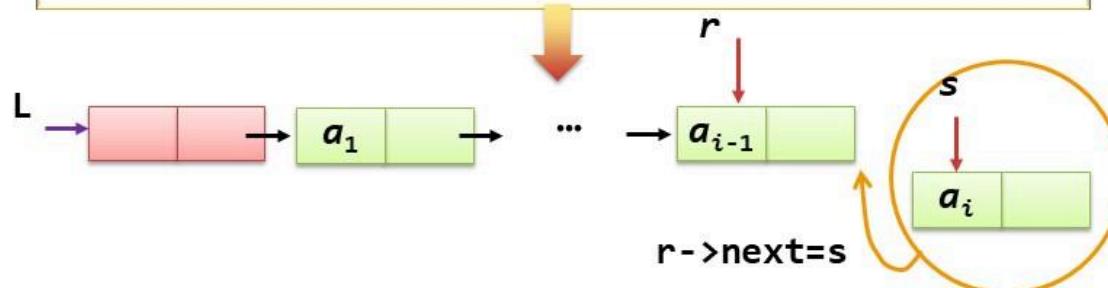
```
void CreateListR(LinkNode *&L, ElemType a[], int n)
{ LinkNode *s, *r;
  int i;
  L=(LinkNode *)malloc(sizeof(LinkNode)); //创建头结点
  r=L; //r始终指向尾结点，开始时指向头结点
```



## 2.3 线性表的链式存储结构

### 2.3.2 单链表

```
for (i=0;i<n;i++)           //循环建立数据结点
{
    s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=a[i];           //创建数据结点s
    r->next=s;              //将s插入r之后
    r=s;
}
r->next=NULL;               //尾结点next域置为NULL
}
```



## 3.1 栈

### 3.1.1 栈的定义

栈是一种只能在一端进行插入或删除操作的线性表。



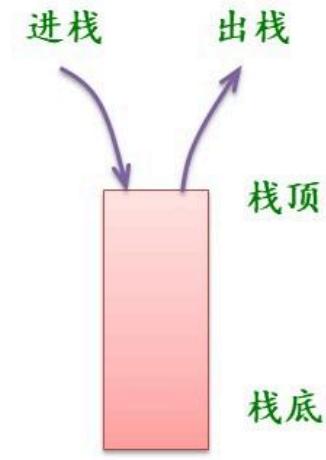
栈只能选取同一个端点进行插入和删除操作

## 3.1 栈

### 3.1.1 栈的定义

#### 栈的几个概念

- 允许进行插入、删除操作的一端称为**栈顶**。
- 表的另一端称为**栈底**。
- 当栈中没有数据元素时，称为**空栈**。
- 栈的插入操作通常称为**进栈或入栈**。
- 栈的删除操作通常称为**退栈或出栈**。



栈示意图

## 3.1 栈

### 3.1.1 栈的定义

栈的主要特点是“**后进先出**”，即后进栈的元素先出栈。栈也称为**后进先出表**。

例如：



假设死胡同的宽度  
恰好只够正一个人

走进死胡同的5人  
要按相反次序退出



死胡同就是一个栈！



## 3.1 栈

### 3.1.1 栈的定义

**示例** 设一个栈的输入序列为  $a, b, c, d$ , 则借助一个栈所得到的输出序列不可能是 ( )。

A.  $c, d, b, a$

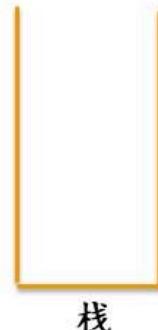
B.  $d, c, b, a$

C.  $a, c, d, b$

D.  $d, a, b, c$

选项D是不可能的?

$d \ c \ b \ a$



下一步不可能出栈  $a$

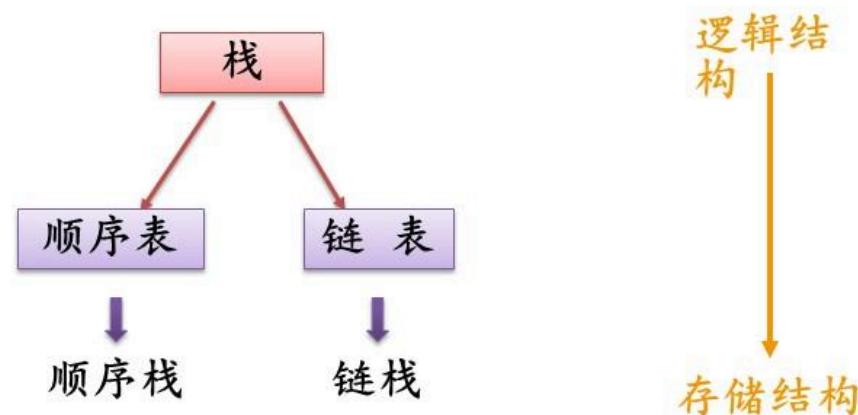
答案为 D



## 3.1 栈

### 3.1.2 栈的顺序存储结构及其基本运算实现

栈中元素逻辑关系与线性表的相同，栈可以采用与线性表相同的存储结构。



## 3.1 栈

### 3.1.2 栈的顺序存储结构及其基本运算实现

假设栈的元素个数最大不超过正整数**MaxSize**, 所有的元素都具有同一数据类型**ElemType**, 则可用下列方式来声明**顺序栈**类型**SqStack**:

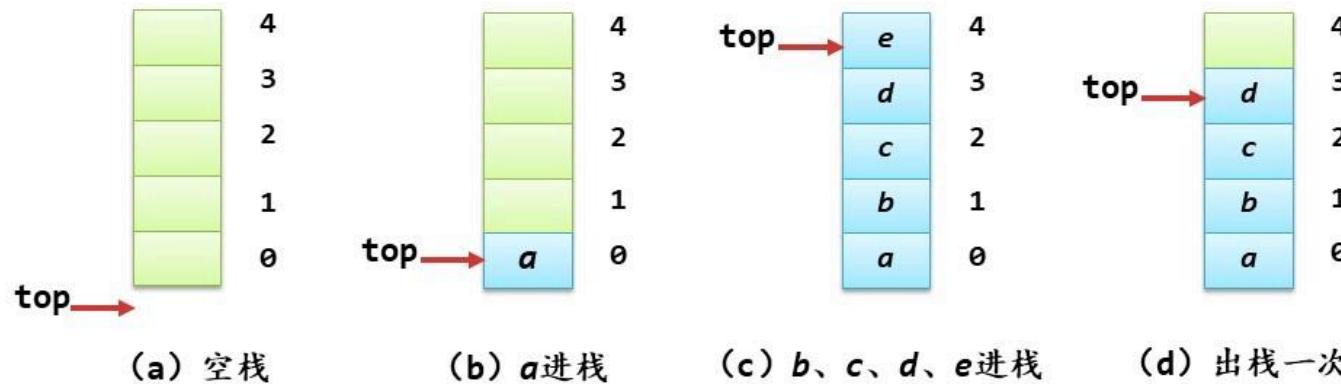
```
typedef struct
{
    ElemType data[MaxSize];
    int top;           //栈顶指针
} SqStack;
```



## 3.1 栈

### 3.1.2 栈的顺序存储结构及其基本运算实现

顺序栈的各种状态



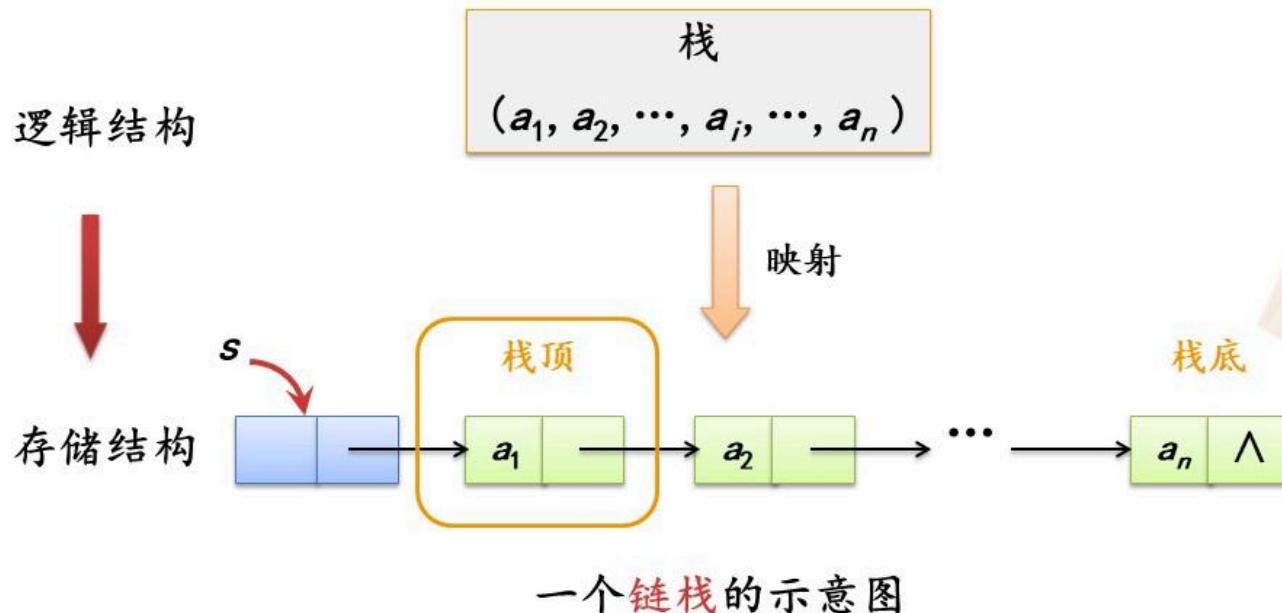
顺序栈4要素：

- 栈空条件： $\text{top} = -1$
- 栈满条件： $\text{top} = \text{MaxSize} - 1$
- 进栈e操作： $\text{top}++$ ; 将e放在top处
- 退栈操作：从top处取出元素e;  $\text{top}--$ ;

## 3.1 栈

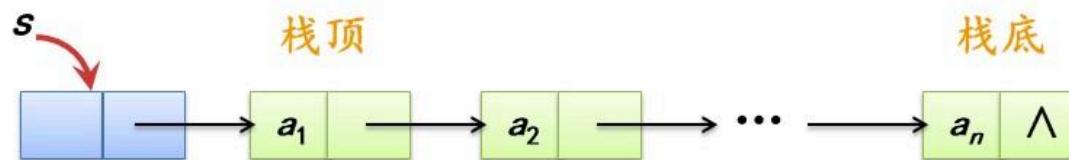
### 3.1.3 栈的链式存储结构及其基本运算的实现

采用链表存储的栈称为**链栈**，这里采用带头结点的单链表实现。



## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现



链栈的4要素：

- 栈空条件： $s \rightarrow \text{next} = \text{NULL}$
- 栈满条件：不考虑
- 进栈e操作：将存放e的结点插入到头结点之后
- 退栈操作：取出头结点之后结点的元素并删除之

## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

链栈中数据结点的类型**LinkStNode** 声明如下：

```
typedef struct linknode
{
    ELEMTYPE data;           // 数据域
    struct linknode *next;   // 指针域
} LinkStNode;
```



## 3.1 栈

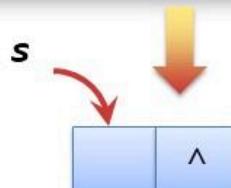
### 3.1.3 栈的链式存储结构及其基本运算的实现

在链栈中，栈的基本运算算法如下。

#### (1) 初始化栈 initStack (&s)

建立一个空栈 s。实际上是创建链栈的头结点，并将其next域置为 NULL。

```
void InitStack(LinkStNode *&s)
{
    s=(LinkStNode *)malloc(sizeof(LinkStNode));
    s->next=NULL;
}
```

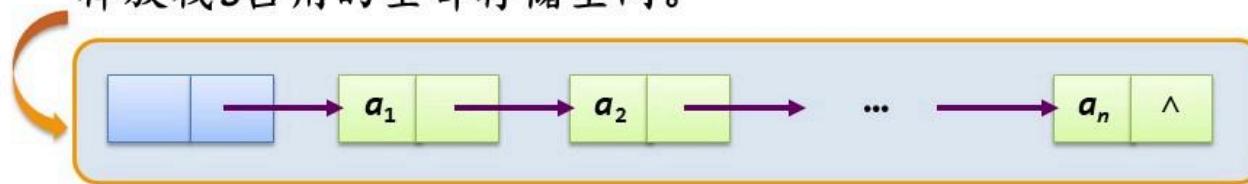


## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

#### (2) 销毁栈DestroyStack (&s)

释放栈s占用的全部存储空间。



```
void DestroyStack(LinkStNode *&s)
{
    LinkStNode *p=s, *q=s->next;
    while (q!=NULL)
    {
        free(p);
        p=q;
        q=p->next;
    }
    free(p); //此时p指向尾结点，释放其空间
}
```

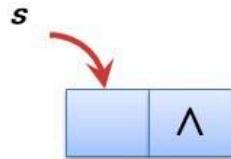
## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

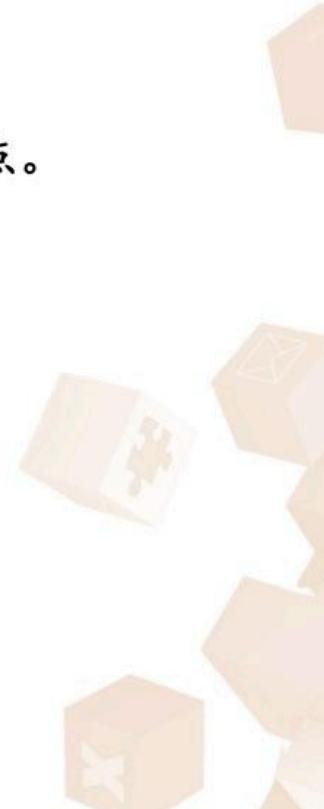
#### (3) 判断栈是否为空 StackEmpty(s)

栈S为空的条件是  $s \rightarrow \text{next} == \text{NULL}$ , 即单链表中没有数据结点。

```
bool StackEmpty(LinkStNode *s)
{
    return(s->next==NULL);
}
```



空栈的情况



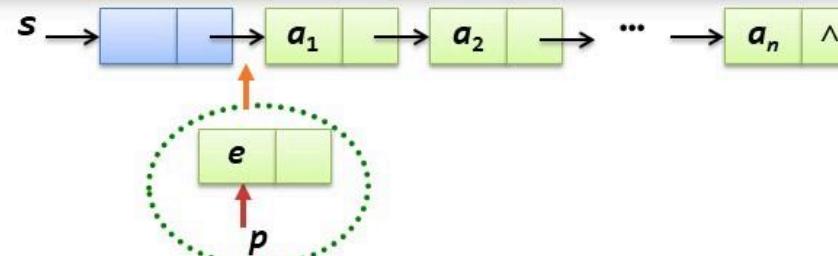
## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

#### (4) 进栈Push(&s, e)

将新数据结点插入到头结点之后。

```
void Push(LinkStNode *&s, ElemtType e)
{
    LinkStNode *p;
    p=(LinkStNode *)malloc(sizeof(LinkStNode));
    p->data=e;          //新建元素e对应的结点p
    p->next=s->next;   //插入p结点作为开始结点
    s->next=p;
}
```



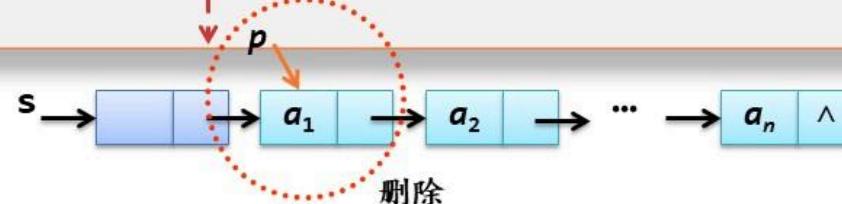
## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

#### (5) 出栈Pop (&s, &e)

栈非空时，将头结点后继数据结点的数据域赋给e，然后将其删除。

```
bool Pop(LinkStNode *&s, ElemtType &e)
{
    LinkStNode *p;
    if (s->next==NULL)          // 栈空的情况
        return false;
    p=s->next;                  // p指向开始结点
    e=p->data;
    s->next=p->next;           // 删除p结点
    free(p);                    // 释放p结点
    return true;
}
```



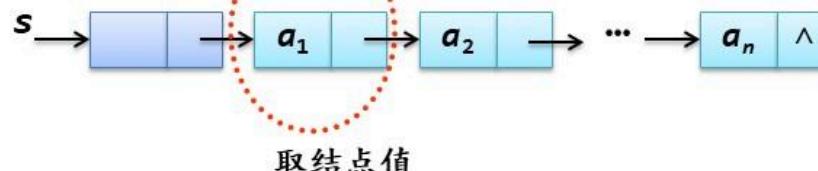
## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

#### (6) 取栈顶元素GetTop(s, e)

在栈不为空的条件下，将头结点后继数据结点的数据域赋给e。

```
bool GetTop(LinkStNode *s, ElemtType &e)
{  if (s->next==NULL)          //栈空的情况
    return false;
  e=s->next->data;
  return true;
}
```



## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

#### 链栈的应用算法设计

**【例3.5】** 编写一个算法判断输入的表达式中括号是否配对（假设只含有左、右圆括号）。

#### 算法设计思路

一个表达式中的左右括号是按最近位置配对的。所以利用一个栈来进行求解。这里采用链栈。

## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

表达式括号不配对情况的演示

例如：exp=“( ( ) ) ”

↑↑↑↑↑

(  
()

- ① ‘(‘进栈
- ② ‘(‘进栈
- ③ 遇到’)’，栈顶为’(‘，退栈
- ④ 遇到’)’，栈顶为’(‘，退栈
- ⑤ 遇到’)’，栈为空，返回false



## 3.1 栈

### 3.1.3 栈的链式存储结构及其基本运算的实现

表达式括号配对情况的演示

例如：exp=“( ( ) )”

↑↑↑↑

(  
()

- ① ‘(‘进栈
- ② ‘(‘进栈
- ③ 遇到’)’，栈顶为’(‘，退栈
- ④ 遇到’)’，栈顶为’(‘，退栈



栈空且exp扫描完，返回true



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



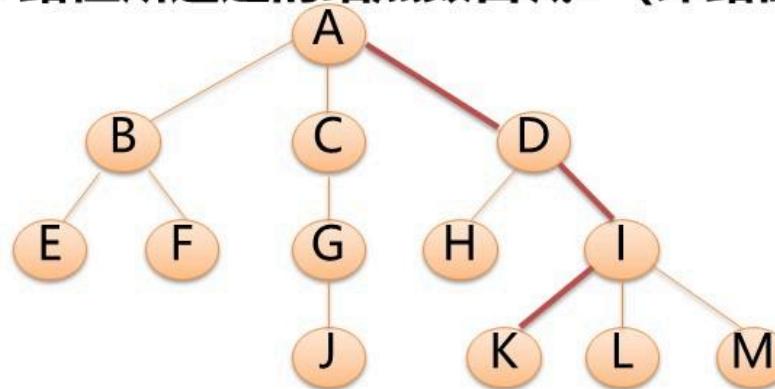
### 7.1.3 树的基本术语

#### 3、路径与路径长度

两个结点 $d_i$ 和 $d_j$ 的结点序列  $(d_i, d_{i_1}, \dots, d_j)$  称为路径。

其中 $\langle d_x, d_y \rangle$ 是分支。

路径长度等于路径所通过的结点数目减1（即路径上分支数目）。



A到K的路径为A, D, I, K,

其长度为3

## 3.2 队列

### 3.2.1 队列的定义

队列的几个概念

- 进行插入的一端称做**队尾** (rear)。
- 进行删除的一端称做**队首或队头** (front)。
- 向队列中插入新元素称为**进队或入队**，新元素进队后就成为新的队尾元素。
- 从队列中删除元素称为**出队或离队**，元素出队后，其后继元素就成为队首元素。



## 3.2 队列

### 3.2.1 队列的定义

队列的主要特点是先进先出，所以又把队列称为先进先出表。

例如：



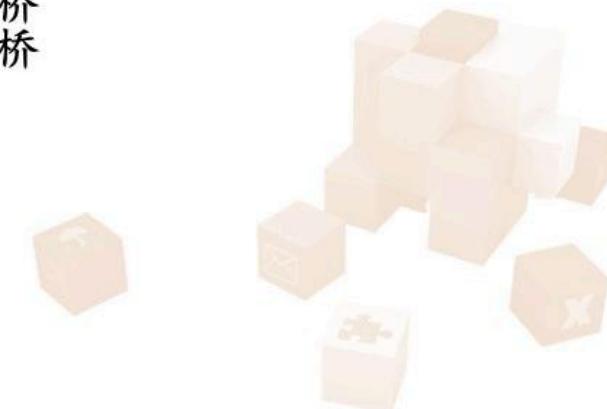
假如5个人过  
独木桥



只能按上桥  
的次序过桥



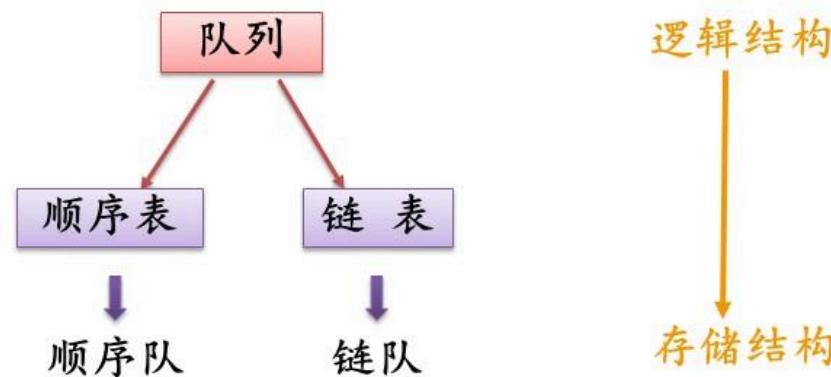
这里独木桥就是一个队列



## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

队列中元素逻辑关系与线性表的相同，队列可以采用与线性表相同的存储结构。



## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

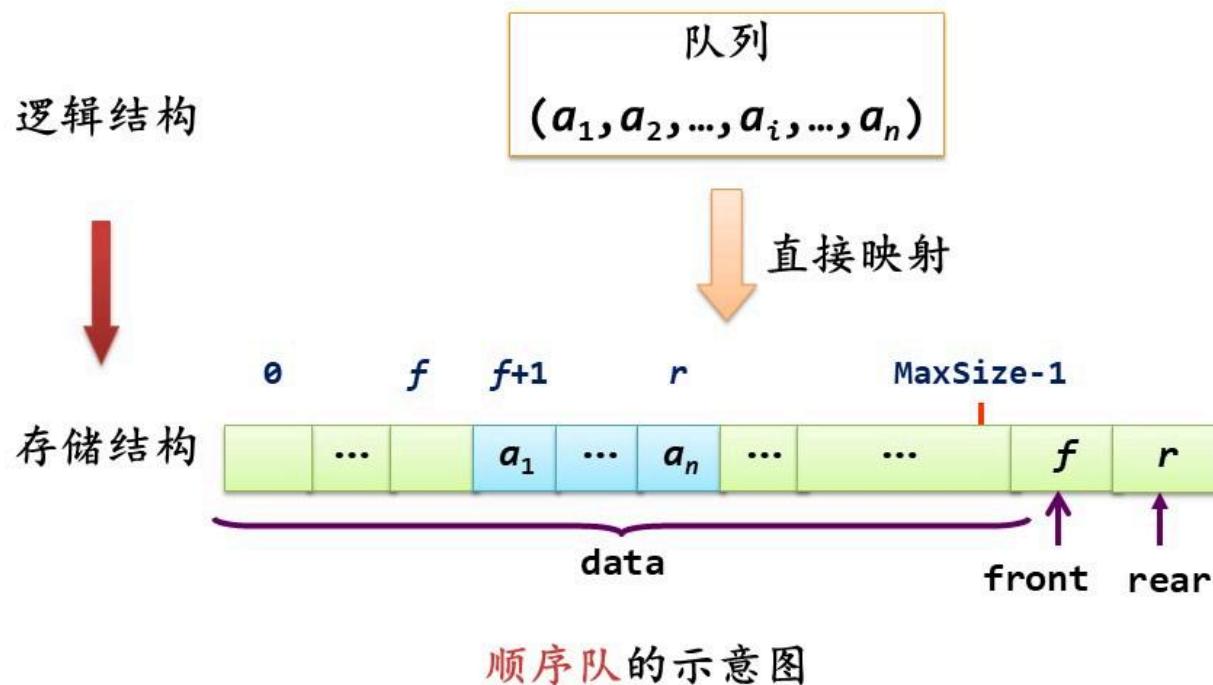
顺序队类型**SqQueue**声明如下：

```
typedef struct
{  ELEMTYPE data[MaxSize];
    int front, rear;           //队首和队尾指针
} SqQueue;
```

因为队列两端都在变化，所以需要两个指针来标识队列的状态。

## 3.2 队列

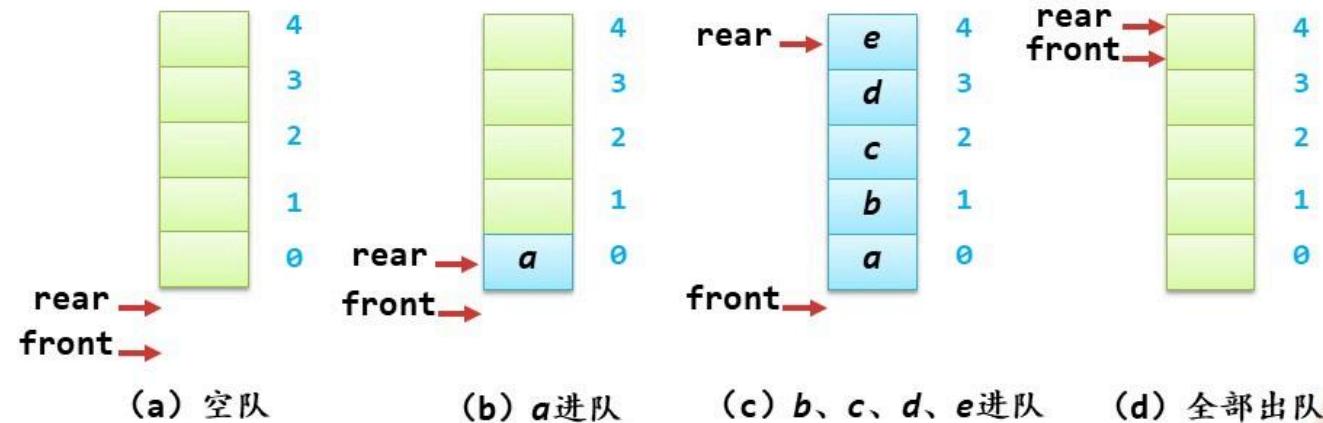
### 3.2.2 队列的顺序存储结构及其基本运算实现



## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

队列的各种状态



顺序队的4要素（初始时 $\text{front}=\text{rear}=-1$ ）：

- 队空条件： $\text{front} = \text{rear}$
- 队满条件： $\text{rear} = \text{MaxSize}-1$
- 元素 $e$ 进队： $\text{rear}++; \text{data}[\text{rear}] = e;$
- 元素 $e$ 出队： $\text{front}++; e = \text{data}[\text{front}];$

注意： $\text{rear}$ 指向队尾元素； $\text{front}$ 指向队头元素的前一个位置。

## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

#### 2、环形队列（或循环队列）中实现队列的基本运算



这是因为采用 `rear==MaxSize-1` 作为队满条件的缺陷。当队满条件为真时，队中可能还有若干空位置。

这种溢出并不是真正的溢出，称为假溢出。

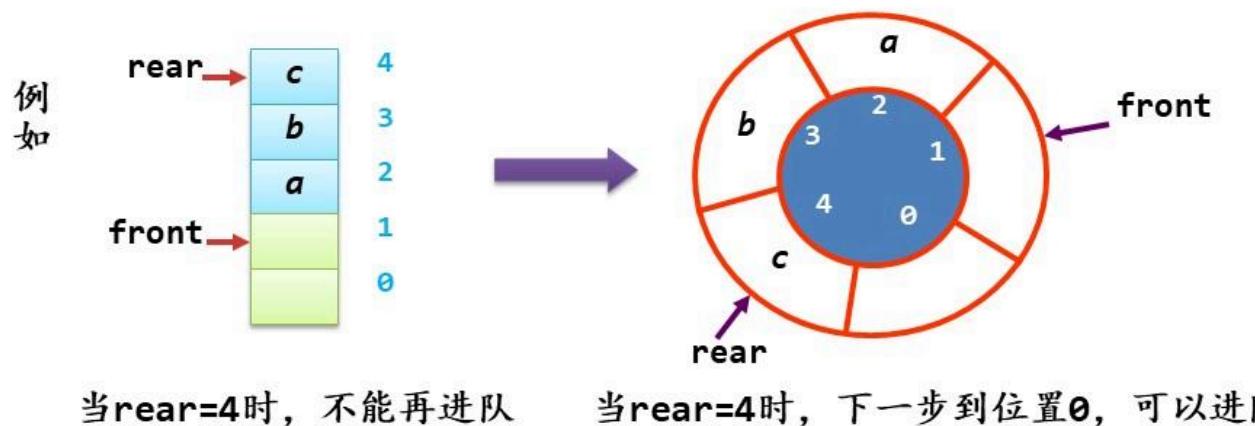


## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

#### ➤ 解决方案

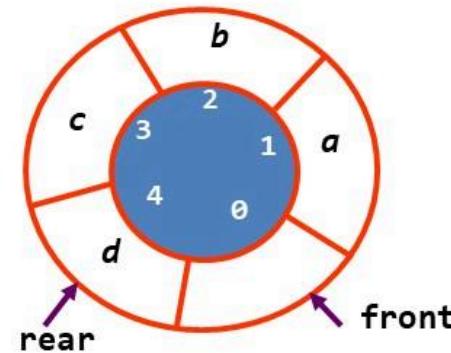
把数组的前端和后端连接起来，形成一个环形的顺序表，即把存储队列元素的表从逻辑上看成一个环，称为**环形队列或循环队列**。



## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

➤ 环形队列（循环队列）：



实际上内存地址一定是连续的，不可能是环形的，这里是通过逻辑方式实现环形队列，也就是将**rear++**和**front++**改为：

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

## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

环形队列的4要素：

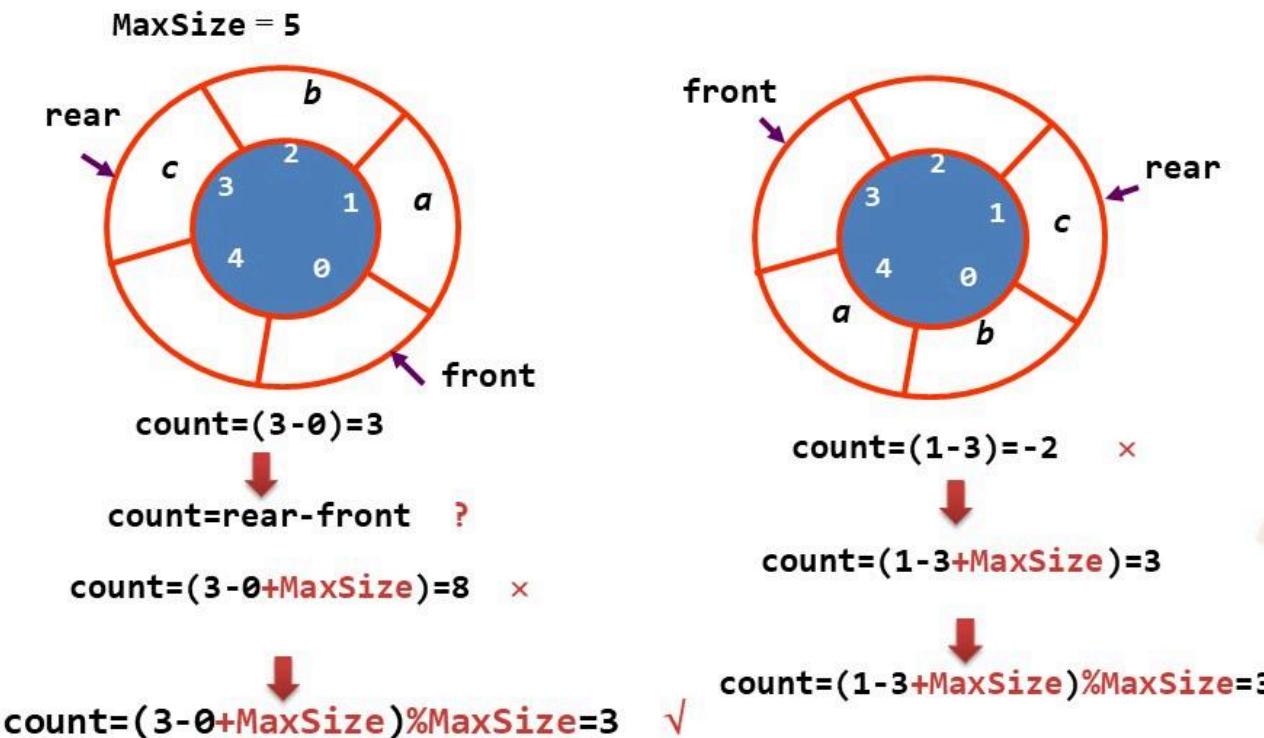
- 队空条件：`front = rear`
- 队满条件：`(rear+1)%MaxSize = front`
- 进队e操作：`rear=(rear+1)%MaxSize;` 将e放在`rear`处
- 出队操作：`front=(front+1)%MaxSize;` 取出`front`处元素e;

在环形队列中，实现队列的基本运算算法与非环形队列类似，只是改为上述4要素即可。

## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现

已知**front**、**rear**，求队中元素个数**count = ?**



## 3.2 队列

### 3.2.2 队列的顺序存储结构及其基本运算实现



已知**front**、**rear**，求队中元素个数**count**:

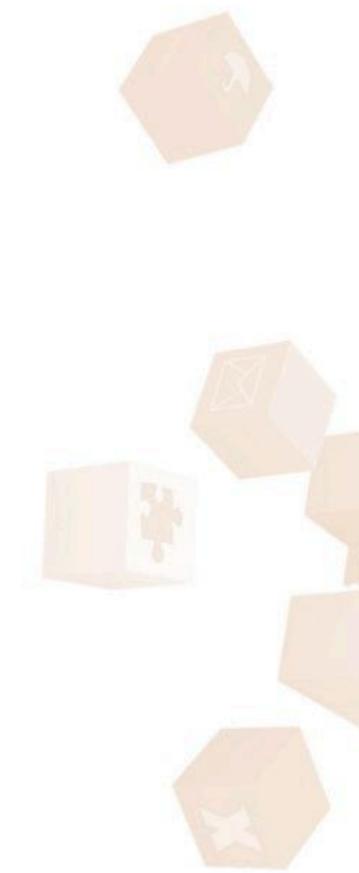
$$\text{count} = (\text{rear} - \text{front} + \text{MaxSize}) \% \text{MaxSize}$$

已知**front**、**count**，求**rear**:

$$\text{rear} = (\text{front} + \text{count}) \% \text{MaxSize}$$

已知**rear**、**count**，求**front**:

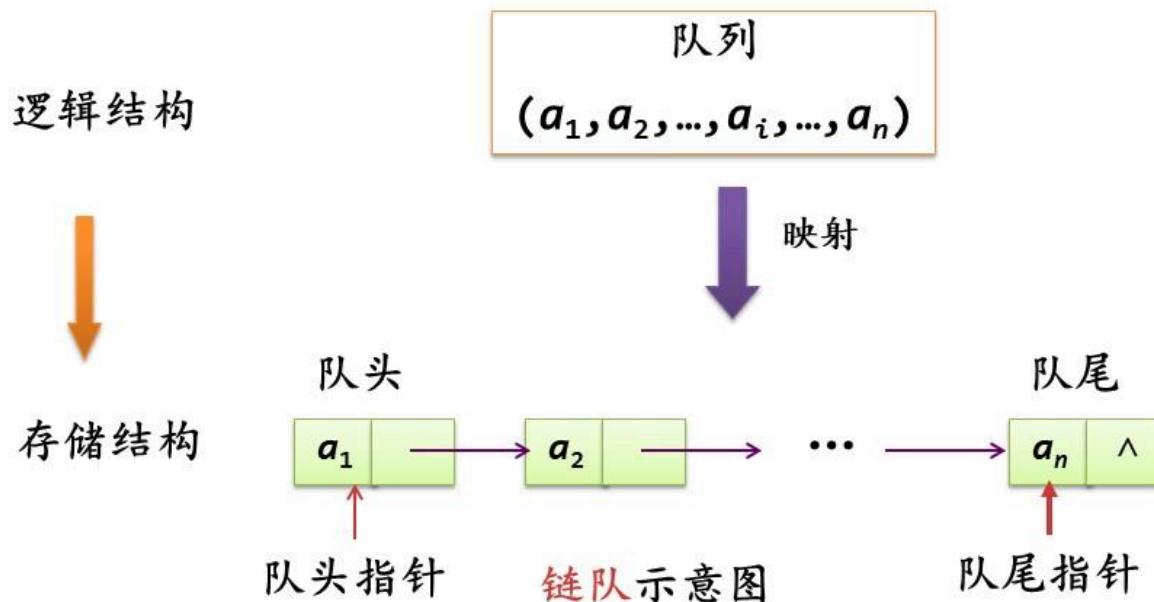
$$\text{front} = (\text{rear} - \text{count} + \text{MaxSize}) \% \text{MaxSize}$$



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

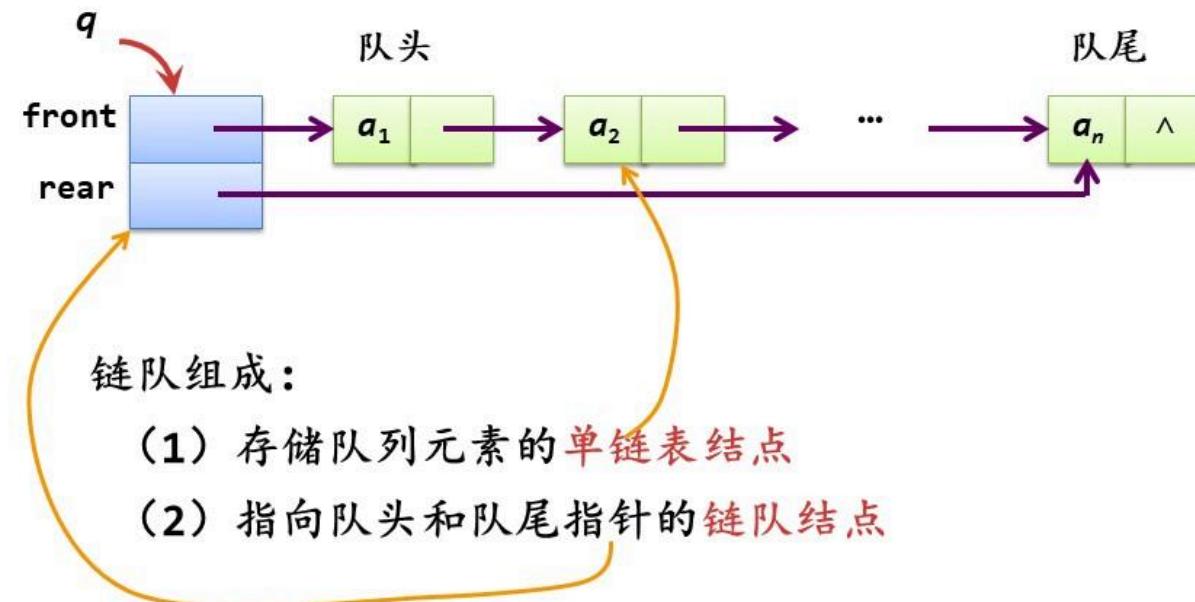
采用链表存储的队列称为**链队**，这里采用**不带头结点的单链表**实现。



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

通常将队头和队尾两个指针合起来：



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

单链表中数据结点类型**DataNode**声明如下：

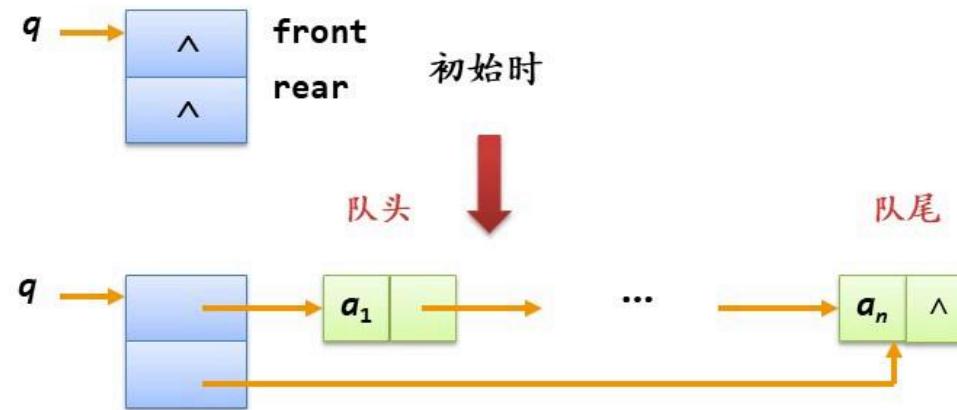
```
typedef struct qnode
{ ElemType data;      //数据元素
  struct qnode *next;
} DataNode;
```

链队结点类型**LinkQuNode**声明如下：

```
typedef struct
{ DataNode *front;    //指向单链表队头结点
  DataNode *rear;     //指向单链表队尾结点
} LinkQuNode;
```

## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现



链队的4要素：

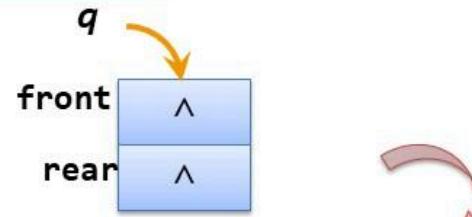
- 队空条件： $front=rear=NULL$
- 队满条件：不考虑
- 进队e操作：将包含e的数据结点插入到单链表表尾
- 出队操作：删除单链表首数据结点

## 3.2 队列

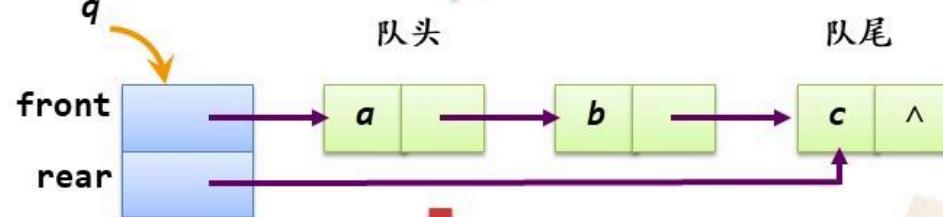
### 3.2.3 队列的链式存储结构及其基本运算的实现

#### 链队的进队和出队操作演示

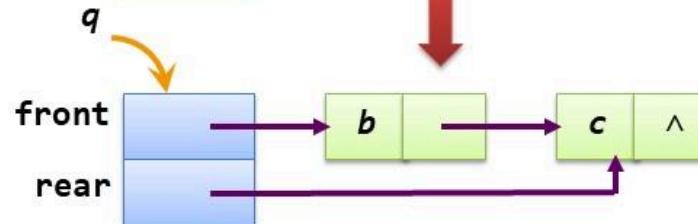
(a) 空队



(b)  $a$ 、 $b$ 、 $c$ 进队



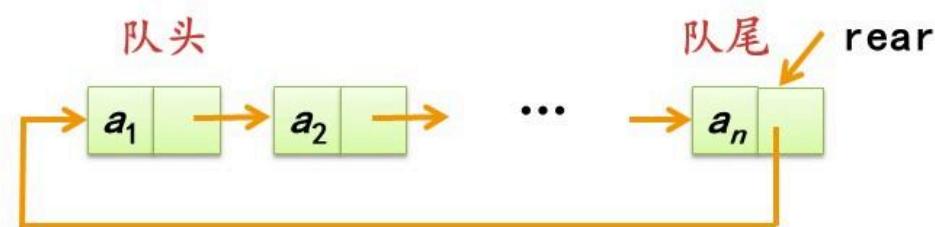
(c) 出队一次



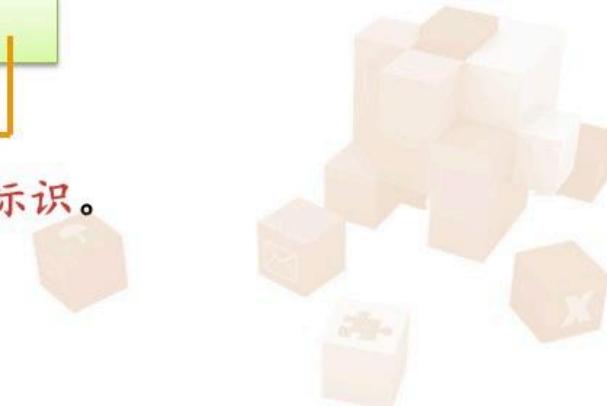
## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

**【例3.8】** 采用一个**不带头结点**只有一个尾结点指针**rear**的循环单链表存储队列，设计队列的初始化、进队和出队等算法。

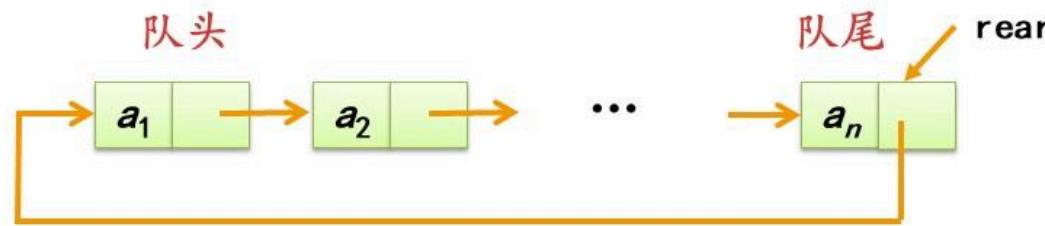


这样的链队通过尾结点指针**rear**唯一标识。



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现



这样的链队通过尾结点指针**rear**唯一标识。

链队的**4**要素：

- 队空条件：**rear=NULL**
- 队满条件：不考虑
- 进队e操作：将包含e的结点插入到单链表表尾
- 出队操作：删除单链表首结点

## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

```
void InitQueue(LinkList *&rear) //初始化队运算算法
{
    rear=NULL;
}

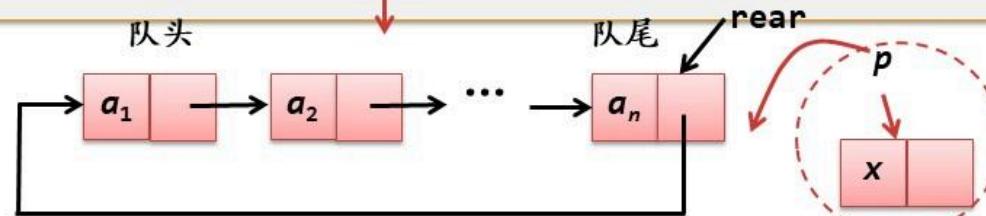
bool queueEmpty(LinkList *rear) //判队空运算算法
{
    return(rear==NULL);
}
```



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

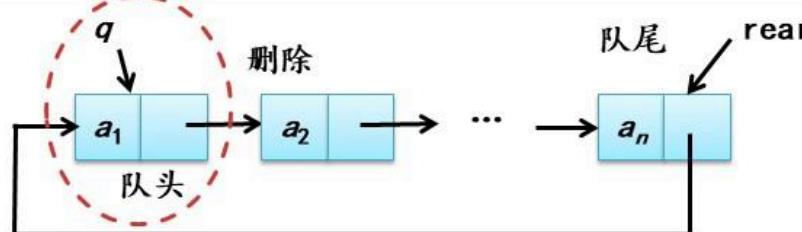
```
void enQueue(LinkList *&rear, ElemtType x) //进队运算算法
{
    LinkList *p;
    p=(LinkList *)malloc(sizeof(LinkList)); //创建新结点
    p->data=x;
    if (rear==NULL) //原链队为空
    {
        p->next=p; //构成循环链表
        rear=p;
    }
    else
    {
        p->next=rear->next; //将p结点插入rear结点之后
        rear->next=p; //让rear指向新插入的结点
        rear=p;
    }
}
```



## 3.2 队列

### 3.2.3 队列的链式存储结构及其基本运算的实现

```
bool deQueue(LinkList *&rear, ElemenType &x) //出队运算算法
{
    LinkList *q;
    if (rear==NULL) return false; //队空
    else if (rear->next==rear) //原队只有一个结点
    {
        x=rear->data;
        free(rear);
        rear=NULL;
    }
    else //原队有两个或以上的结点
    {
        q=rear->next;
        x=q->data;
        rear->next=q->next;
        free(q);
    }
    return true;
}
```





## ④ 分析递归求Fibonacci数列时，栈的变化情况？

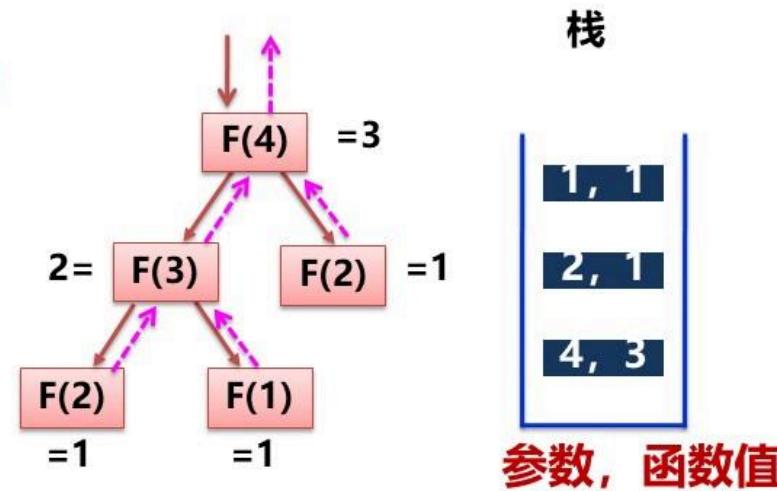
$$F(1)=1$$

$$F(2)=1$$

$$F(n)=F(n-1)+F(n-2)$$

$n>2$

求 $F(4) = ?$



求出 $F(4)=3$





## 4.1 串的基本概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



串（或字符串）是由零个或多个字符组成的有限序列。

串  $\subset$  线性表

- 串中所含字符的个数称为该串的长度（或串长）。
- 含零个字符的串称为空串，用 $\Phi$ 表示。





## 4.1 串的基本概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



串的逻辑表示：

“ $a_1a_2\cdots a_n$ ”



双引号不是串的内容，起标识作用



串的逻辑表示， $a_i$  ( $1 \leq i \leq n$ ) 代表一个字符。





## 4.1 串的基本概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



**串相等：**两个串的长度相等并且各个对应位置上的字符都相同时。

如：

“abcd” ≠ “abc”

“abcd” ≠ “abcde”

所有空串是相等的。





## 4.1 串的基本概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



子串：

一个串中任意个连续字符组成的子序列（含空串）称为该串的子串。

例如，“**abcde**”的子串有：“”、“a”、“ab”、“abc”、“abcd”和“**abcde**”等

真子串是指不包含自身的所有子串。





## 4.2 串的存储结构

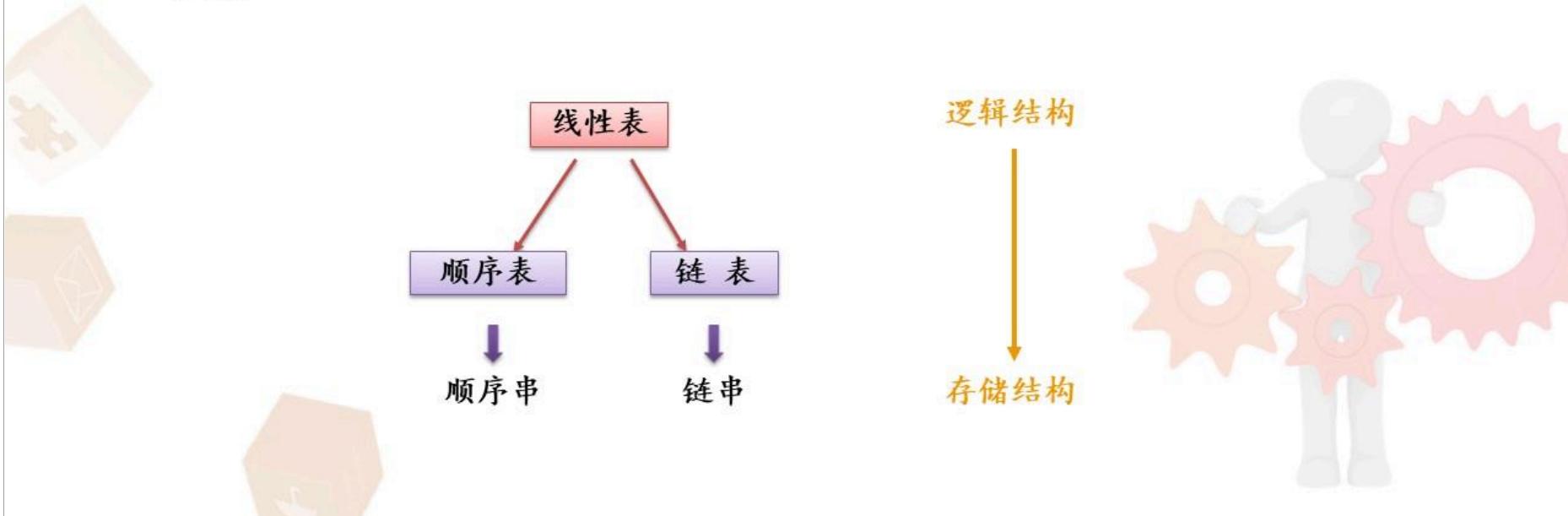


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.1 串的顺序存储及其基本操作实现

串中元素逻辑关系与线性表的相同，串可以采用与线性表相同的存储结构。





## 4.2 串的存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.1 串的顺序存储及其基本操作实现

**【例4.1】**设计顺序串上实现串比较运算 **Strcmp(s, t)** 的算法。例如：

"ab" < "abcd"

"abcd" < "abd"

解

(1) 比较s和t两个串 **共同长度范围内** 的对应字符：

① 若s的字符>t的字符，返回1； → "ba">"abc"

② 若s的字符<t的字符，返回-1； → "aba" < "bc"

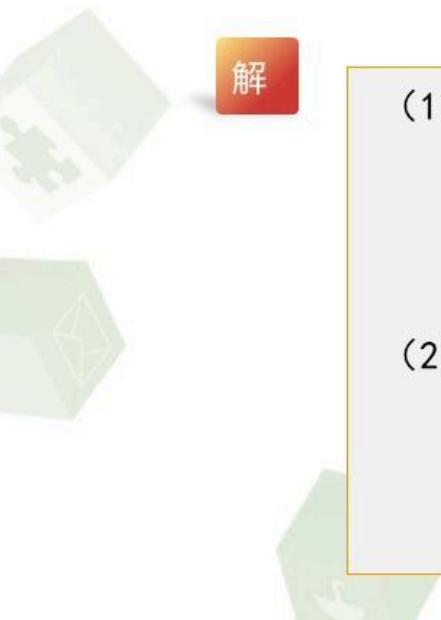
③ 若s的字符=t的字符，按上述规则继续比较。

(2) 当(1)中对应字符均相同时，比较s和t的长度：

① 两者相等时，返回0； → "abc" = "abc"

② s的长度>t的长度，返回1； → "abc" > "ab"

③ s的长度<t的长度，返回-1。 → "ab" < "abd"





## 4.2 串的存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.1 串的顺序存储及其基本操作实现

```
int Strcmp(SqString s, SqString t)
{
    int i, comlen;
    if (s.length < t.length)
        comlen = s.length;           //求s和t的共同长度
    else
        comlen = t.length;
    for (i = 0; i < comlen; i++)      //在共同长度内逐个字符比较
        if (s.data[i] > t.data[i])
            return 1;
        else if (s.data[i] < t.data[i])
            return -1;
    if (s.length == t.length)          //s==t
        return 0;
    else if (s.length > t.length)     //s>t
        return 1;
    else                            //s<t
        return -1;
}
```





## 4.2 串的存储结构

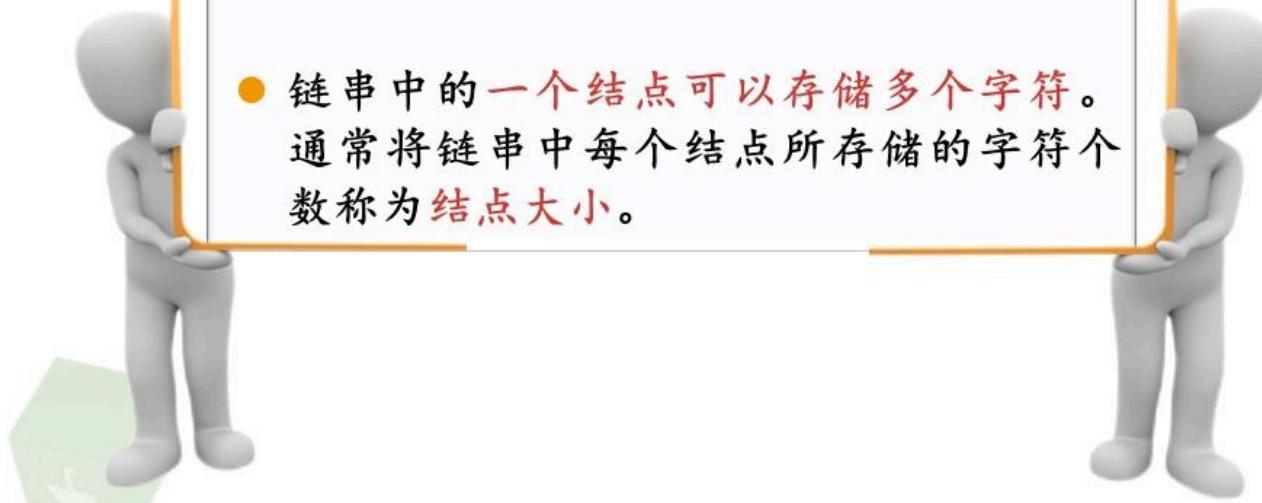


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

- 链串的组织形式与一般的链表类似。
- 链串中的一个结点可以存储多个字符。  
通常将链串中每个结点所存储的字符个数称为**结点大小**。





## 4.2 串的存储结构



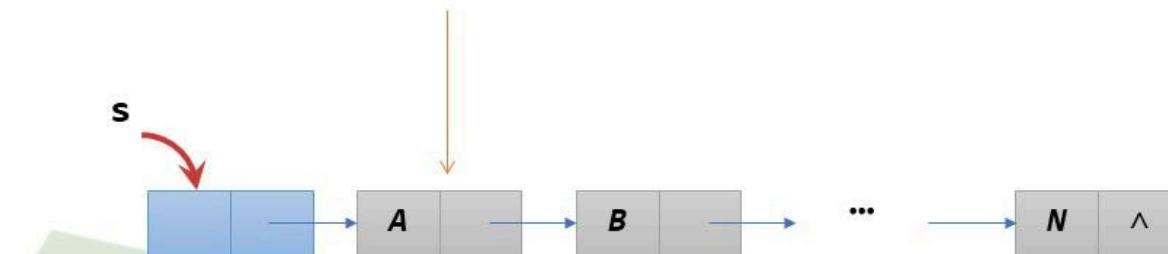
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

链串结点大小1时，链串的结点类型声明如下：

```
typedef struct snode
{
    char data;
    struct snode *next;
} LinkStrNode;
```





## 4.2 串的存储结构



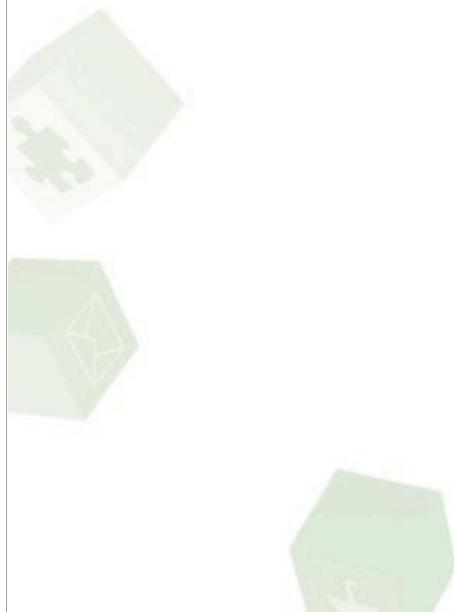
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

链串中实现串的基本运算

与单链表的基本运算类似。





## 4.2 串的存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

**【例4.3】**在链串中，设计一个算法把最先出现的子串“ab”改为“xyz”。

解

① 查找:  $p \rightarrow \text{data} = 'a' \quad \&\& \quad p \rightarrow \text{next} \rightarrow \text{data} = 'b'$





## 4.2 串的存储结构

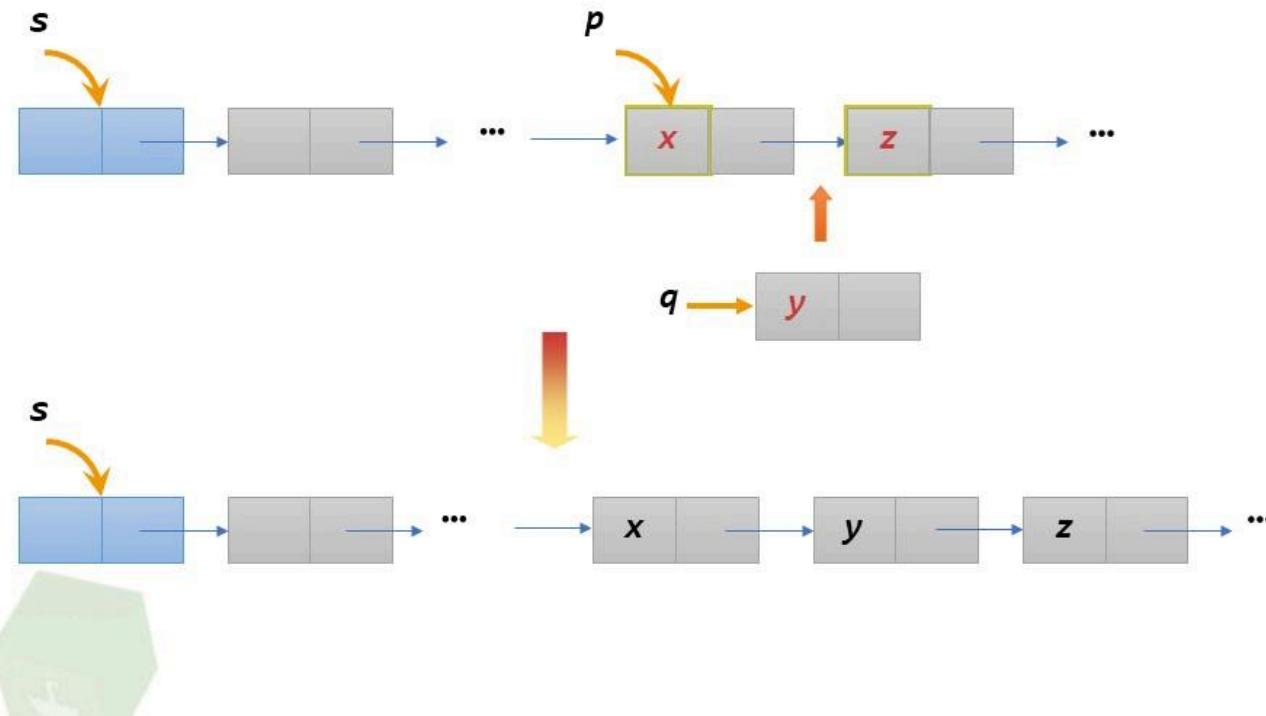


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

#### ② 替换





## 4.2 串的存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.2.2 串的链式存储及其基本操作实现

```
void Repl(LinkStrNode *&s)
{
    LinkStrNode *p=s->next, *q;
    int find=0;
    while (p->next!=NULL && find==0)          //查找ab子串
    {
        if (p->data=='a' && p->next->data=='b')
        {
            p->data='x'; p->next->data='z';
            q=(LinkStrNode *)malloc(sizeof(LinkStrNode));
            q->data='y'; q->next=p->next; p->next=q;
            find=1;
        }
        else p=p->next;
    }
}
```

算法的时间复杂度为 $O(n)$ 。

替换为xyz



## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

目标串  $s$



是子串吗?

} 模式匹配

模式串  $t$

- 成功是指在目标串  $s$  中找到一个模式串  $t$ — $t$  是  $s$  的子串，  
返回  $t$  在  $s$  中的位置。
- 不成功则指目标串  $s$  中不存在模式串  $t$ — $t$  不是  $s$  的子串，  
返回 -1。



## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

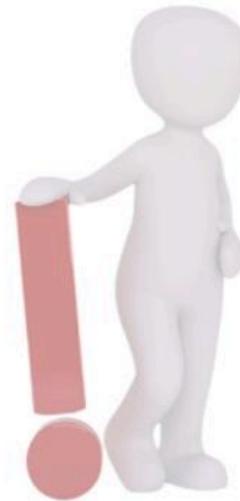
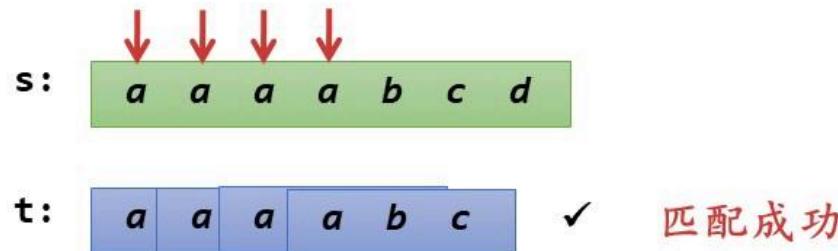


### 4.3.1 Brute-Force算法

Brute-Force简称为**BF算法**，亦称简单匹配算法。采用穷举的思路。

BF是指暴力的意思！

例如， $s = "aaaabcd"$ ， $t = "abc"$ 。





## 4.3 串的模式匹配



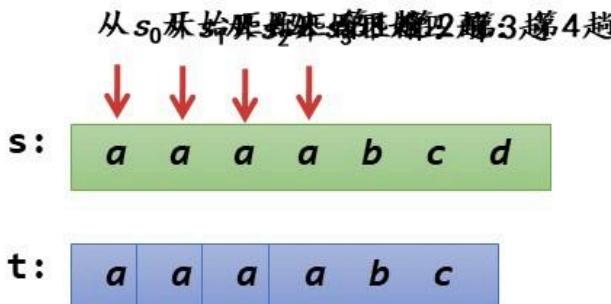
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

BF算法思路：从s的每一个字符开始依次与t的字符进行匹配。

过程



- $i, j$ 分别扫描字符串s和t
- $i$ 从0到 $s.length-t.length$ 循环。
- 对于每个 $s[i]$ ,  $k=i, j=0$ 开始比较
- 如果t扫描完毕，则返回 $i$
- 若s全部比较完毕都没有返回，则返回-1



## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



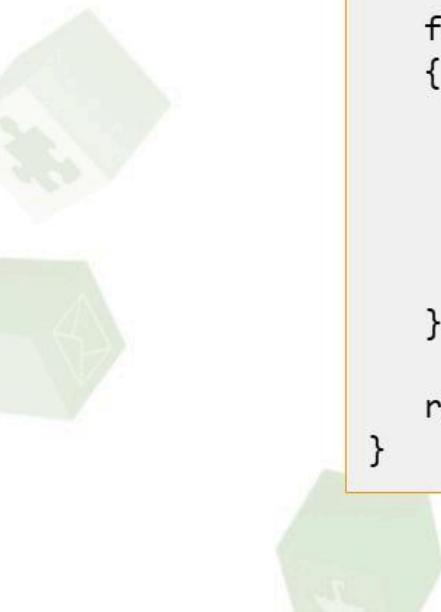
### 4.3.1 Brute-Force算法

```
int index(SqString s,SqString t)
{
    int i,j,k;

    for (i=0;i<=s.length-t.length;i++)
    {
        for (k=i,j=0; k<s.length && j<t.length &&
            s.data[k]==t.data[j]; k++,j++);

        if (j==t.length)
            return(i);
    }

    return(-1);
}
```





## 4.3 串的模式匹配

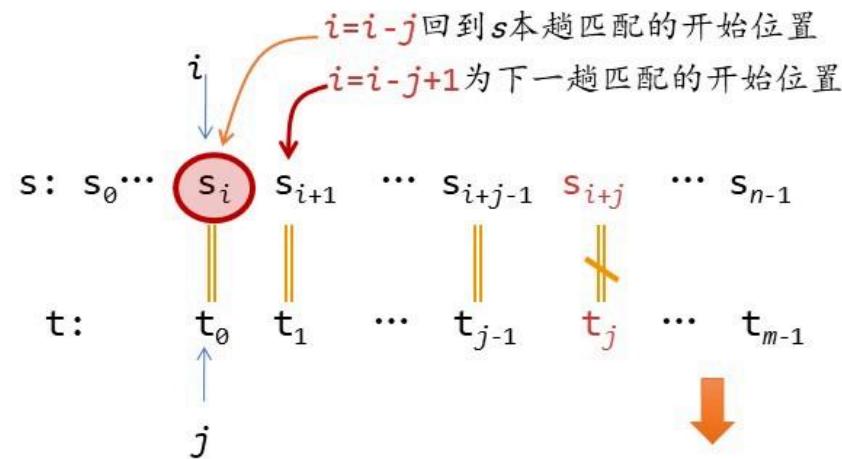


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

实际上，可以只采用两个变量  $i$ 、 $j$ ，通过  $i$  回退实现：



从  $s_i$  开始匹配：

- 若  $s_i = t_j \Rightarrow i++, j++$
- 若  $s_i \neq t_j \Rightarrow i = i - j + 1, j = 0$



## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

对应的BF算法如下：

```
int BF(SqString s, SqString t)
{
    int i=0, j=0;
    while (i<s.length && j<t.length)
    {
        if (s.data[i]==t.data[j])
        {
            i++;                      //主串和子串依次匹配下一个字符
            j++;
        }
        else
        {
            i=i-j+1;                //主串、子串指针回溯重新开始下一次匹配
            j=0;                     //主串从下一个位置开始匹配
        }
        if (j>=t.length)           //或者if (j==t.length)
            return(i-t.length);    //返回匹配的第一个字符的下标
        else
            return(-1);             //模式匹配不成功
    }
}
```





## 4.3 串的模式匹配



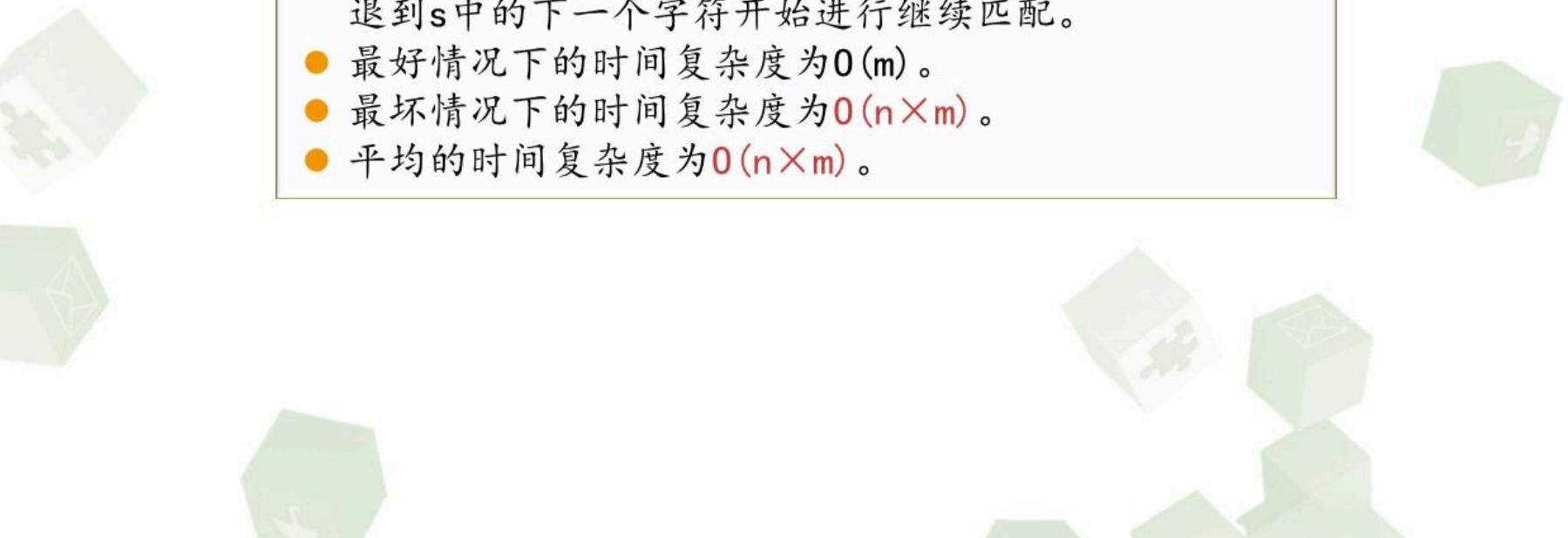
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

#### BF算法分析

- 算法在字符比较不相等，需要回溯（即  $i=i-j+1$ ）：即退到  $s$  中的下一个字符开始进行继续匹配。
- 最好情况下的时间复杂度为  $O(m)$ 。
- 最坏情况下的时间复杂度为  $O(n \times m)$ 。
- 平均的时间复杂度为  $O(n \times m)$ 。





## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

**示例** 假设串采用顺序串存储结构。设计一个算法求串 $t$ 在串 $s$ 中出现的次数，如果不是子串返回0。采用BF算法求解。

例如"aa"在"aaaab"中出现2次。

解

- 用count累计t在串s中出现的次数（初始值为0）。
- 采用BF方法，在s中找到子串t后不是退出，而是count增加1，i置为k并继续查找，直到整个字符串查找完毕。





## 4.3 串的模式匹配



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 4.3.1 Brute-Force算法

```
int StrCount1(SqString s,SqString t)//利用BF算法求t在s中出现的次数
{
    int i=0,j,k,count=0;
    while (i<=s.length - t.length)
    {
        for (k=i,j=0; k<s.length && j<t.length &&
            s.data[k]==t.data[j]; k++,j++);
        if (j==t.length)          //找到一个子串
        {
            count++;           //累加次数
            i=k;                //i从j开始
        }
        else
            i++;               //i增加1
    }
    return count;
}
```

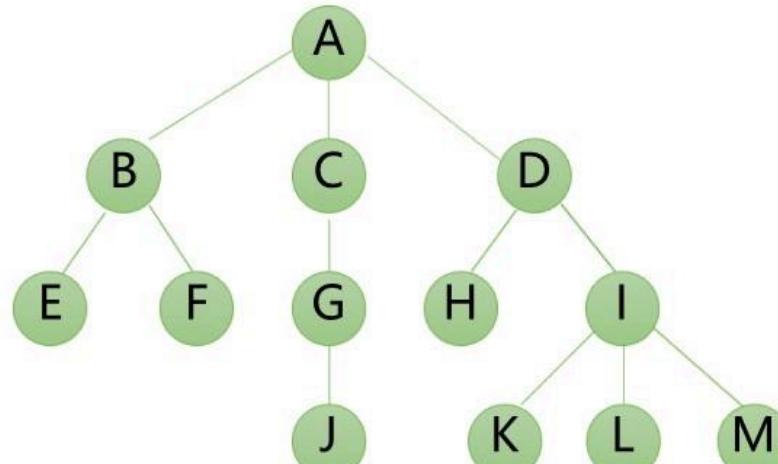


## 7.1 树的概念

### 7.1.2 树的（逻辑）表示

**(1) 树形表示法：**

使用一棵**倒置的树**表示树结构，非常直观和形象。



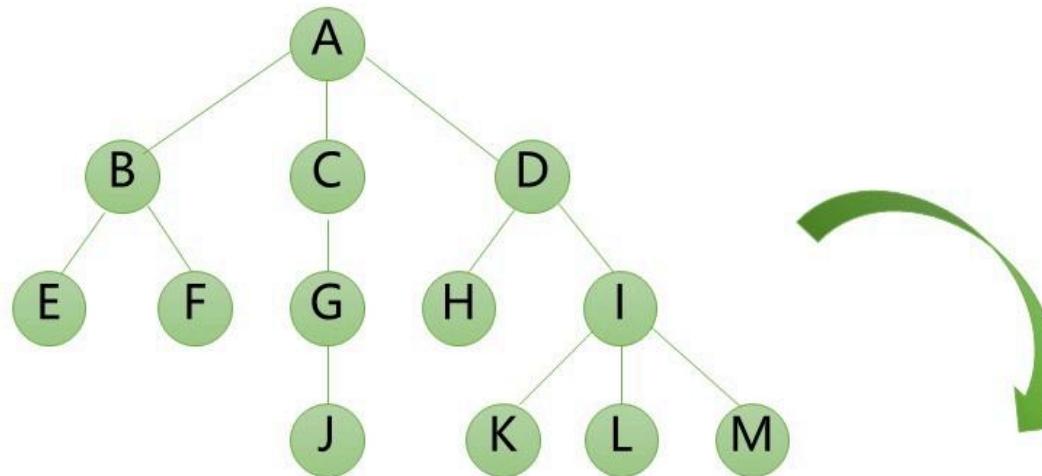
**逻辑结构表示1**

## 7.1 树的概念

### 7.1.2 树的（逻辑）表示

(4) 括号表示法: 用一个字符串表示树。

基本形式: 根(子树1, 子树2, ..., 子树 $m$ )



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



## 7.1 树的概念



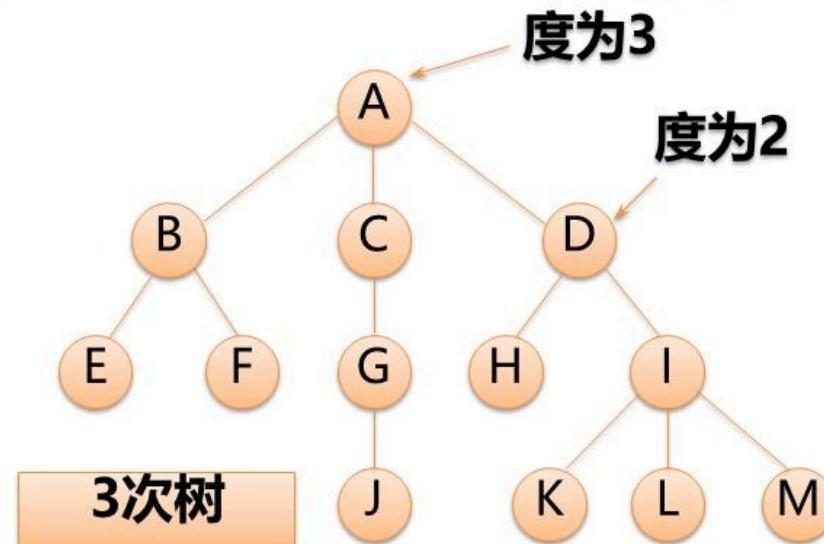
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.3 树的基本术语

#### 1、结点的度与树的度：

树中一个结点的子树的个数称为该结点的度。树中各结点的度的最大值称为树的度，通常将度为m的树称为m次树或者m叉树。





## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

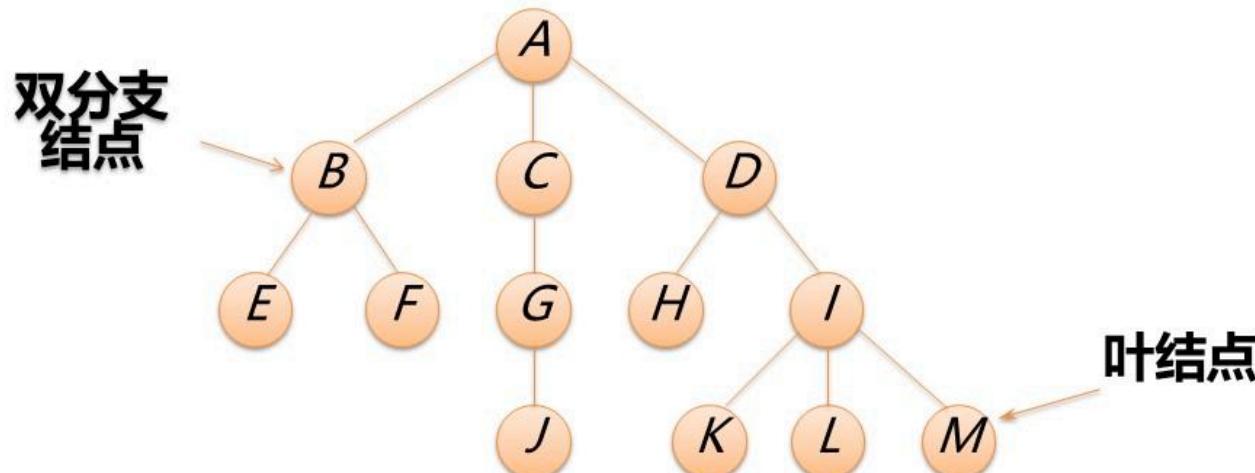


### 7.1.3 树的基本术语

#### 2、分支结点与叶结点：

度不为零的结点称为非终端结点，又叫**分支结点**。度为零的结点称为终端结点或**叶结点**（或**叶子结点**）。

度为1的结点称为**单分支结点**；度为2的结点称为**双分支结点**，依此类推。





## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

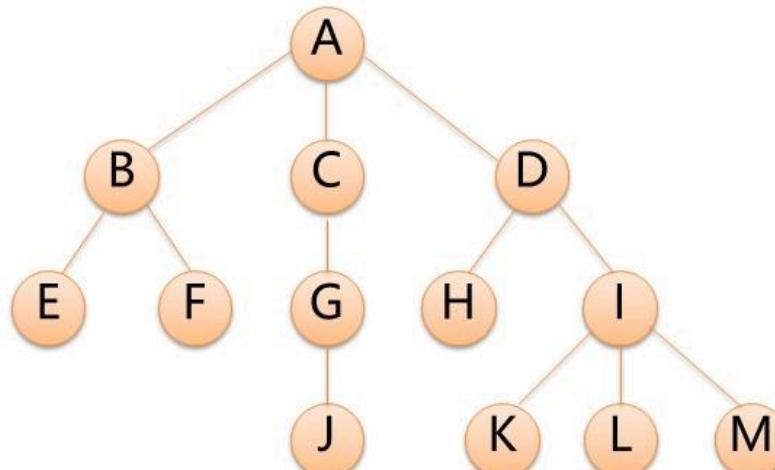


### 7.1.3 树的基本术语

#### 4、孩子结点、双亲结点和兄弟结点

在一棵树中，每个结点的后继，被称作该结点的**孩子结点**（或**子女结点**）。相应地，该结点被称作**孩子结点的双亲结点**（或**父母结点**）。

具有同一双亲的孩子结点互为**兄弟结点**。



**A**的孩子结点有**B、C、D**

**B、C、D**的双亲结点为**A**

**B、C、D**的互为兄弟结点



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



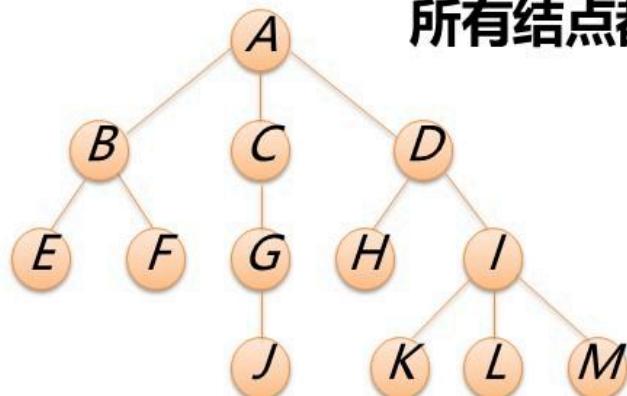
### 7.1.3 树的基本术语

#### 5、子孙结点和祖先结点：

在一棵树中，一个结点的所有子树中的结点称为该结点的**子孙结点**。

从根结点到达一个结点的路径上经过的所有结点被称作该结点的**祖先结点**。

所有结点都是A的子孙结点



L的祖先结点为**A、D、I**





## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



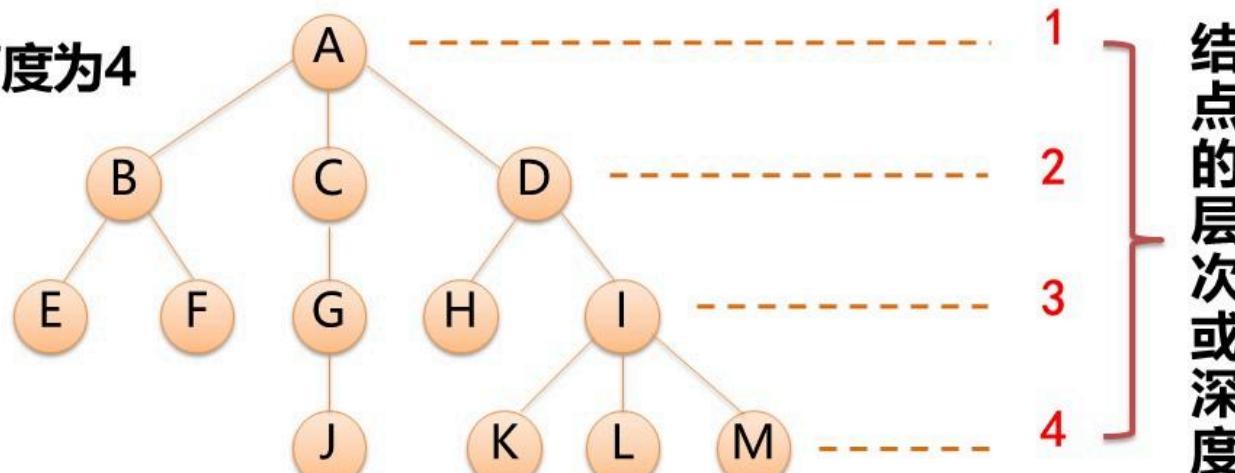
### 7.1.3 树的基本术语

#### 6、结点的层次和树的高度：

树中的每个结点都处在一个层次上。结点的层次从树根开始定义，根结点为第1层，它的孩子结点为第2层，以此类推，一个结点所在的层次为其双亲结点所在的层次加1。

树中结点的最大层次称为树的高度（或树的深度）。

树的高度为4





## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

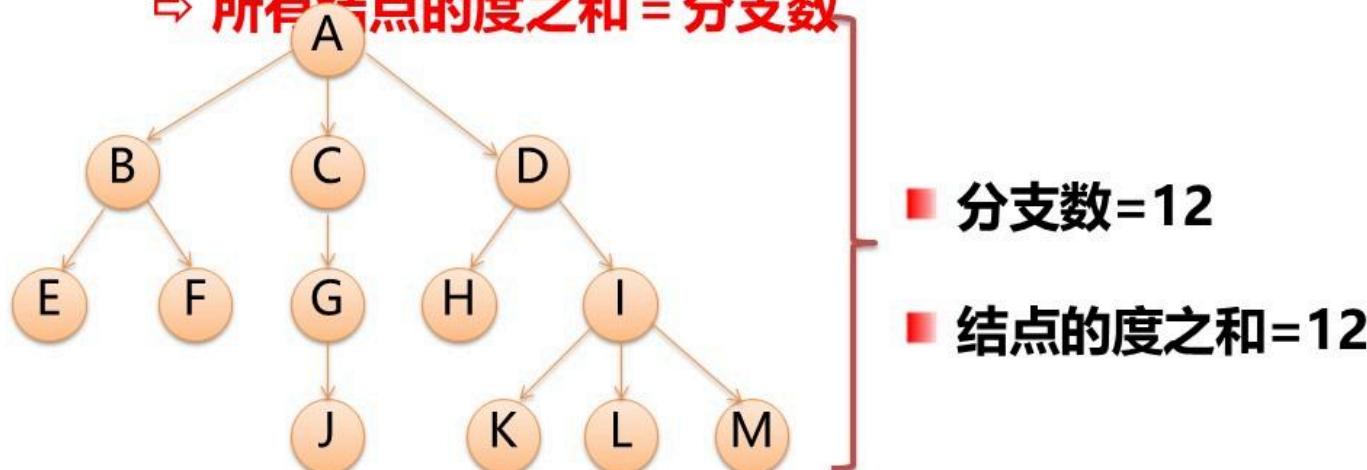
**性质1：**

**树中的结点数等于所有结点的度数之和加1。**

**证明**

① 树中每个分支计为一个结点的度（每条分支线从一个结点引出来的）

⇒ **所有结点的度之和 = 分支数**





## 7.1 树的概念



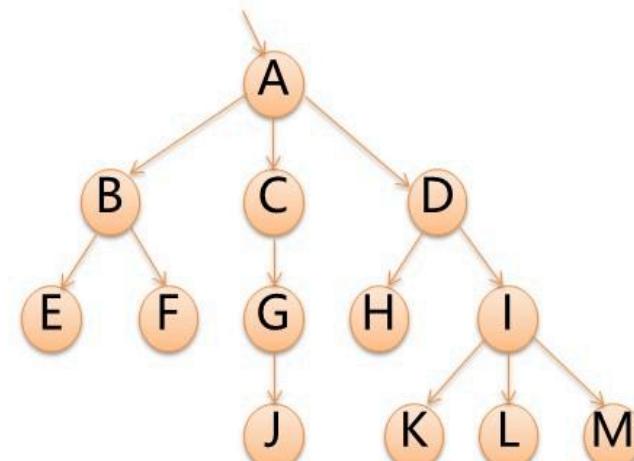
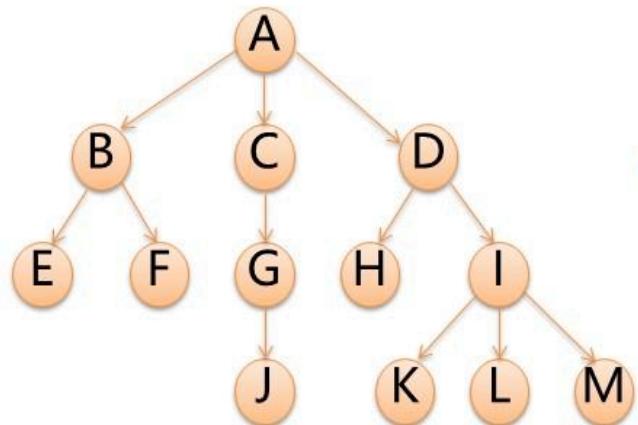
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

证明：

② 除了根结点，每条分支线都指向一个结点。给根结点加上一个分支：



这样，此时的分支数与结点数相同

所有结点的度之和 = 分支数

$\Rightarrow$  实际分支数 =  $n-1$

$n = \text{度之和} + 1$



## 7.1 树的概念



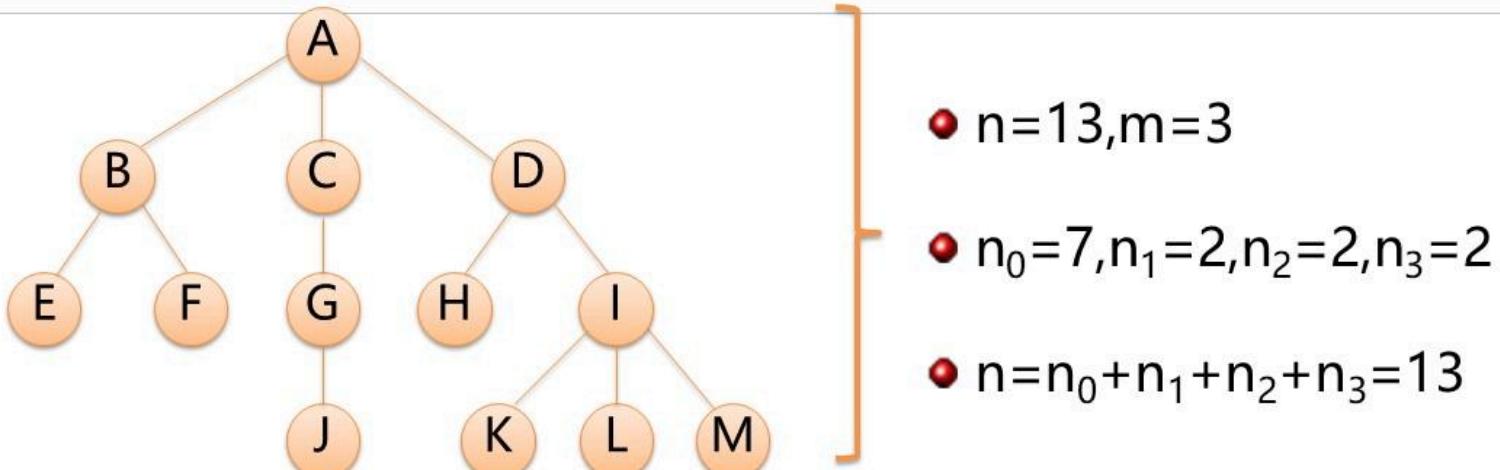
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

#### 度为 $m$ 的树的其他重要特性 (1/2) :

- 结点个数表示:  $n$ 为总结点个数,  $n_i$ 为度为 $i$  ( $0 \leq i \leq m$ ) 的结点个数
- $n = n_0 + n_1 + \dots + n_m$





## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

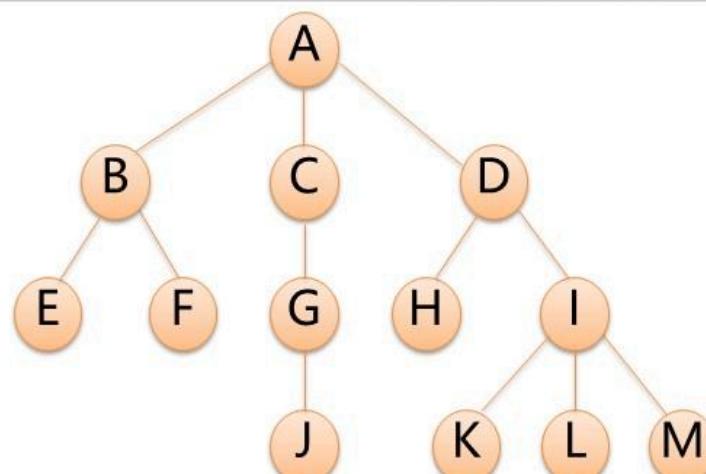


### 7.1.4 树的性质

#### 度为m的树的其他重要特性 (2/2) :

- 在所有结点度之和中，一个度为1的结点贡献1个度，...，一个度为 $m$ 的结点贡献 $m$ 个度，...，一个度为 $m$ 的结点贡献 $m$ 个度

$$\Rightarrow \text{所有结点度之和} = n_1 + 2n_2 + \dots + mn_m = n-1$$



- $n=13, m=3$
- $n_1=2, n_2=2, n_3=2$
- 结点的度之和  
 $=n_1+2n_2+3n_3=2+4+6=12=n-1$



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

#### 示例

一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶子结点个数是( )。

- A.41
- B.82
- C.113
- D.122

注：本题为2010年全国考研题

$$m = 4, n = n_0 + n_1 + n_2 + n_3 + n_4 = n_0 + 10 + 1 + 10 + 20 = n_0 + 41.$$

$$n - 1 = \text{度之和} = n_1 + 2n_2 + 3n_3 + 4n_4 = 122, \text{ 得 } n = 123.$$

$$n_0 = n - 41 = 123 - 41 = 82.$$

答案为B。



## 7.1 树的概念



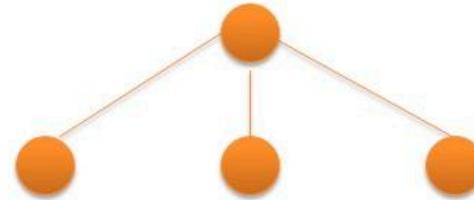
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

**性质2：**

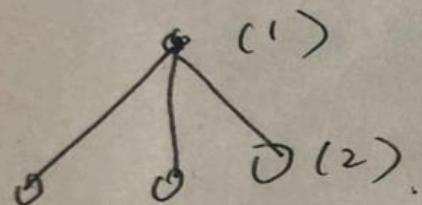
度为 $m$ 的树中第 $i$ 层上至多有 $m^{i-1}$ 个结点 ( $i \geq 1$ )。



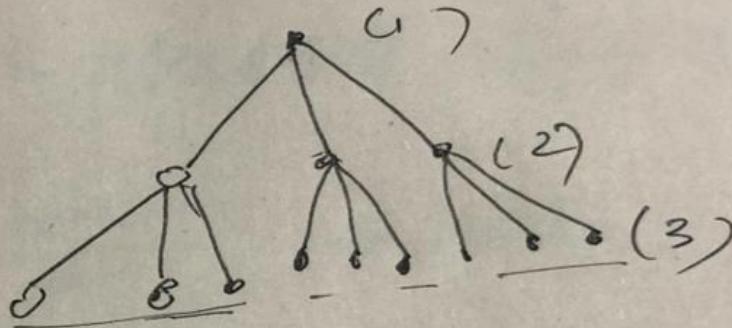
度为3的树第2层至多有3个结点



度为3的树(树中最大结点数)



$$\underline{3^{(2-1)} = 3}.$$



$$3^{(3-1)} = 9.$$

第4层吧

欸，是不是等比数列？



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

性质3：

高度为 $h$ 的 $m$ 次树至多有  $\frac{m^h - 1}{m - 1}$  个结点。

$m$ 次树每层最多结点数：

- 第1层： 1
  - 第2层：  $m^1$
  - 第3层：  $m^2$
  - ...
  - 第 $h$ 层：  $m^{h-1}$
- $\left. \begin{matrix} m^h - 1 \\ \hline m - 1 \end{matrix} \right\}$



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

**性质4：**

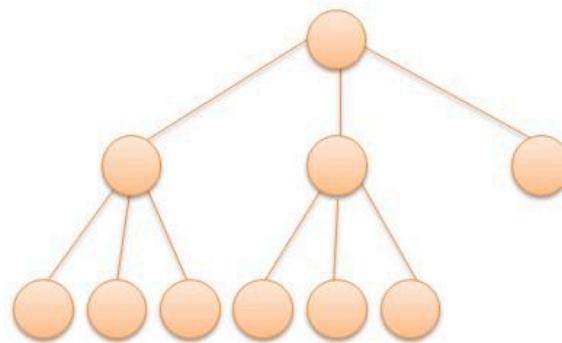
具有 $n$ 个结点的 $m$ 次树的**最小高度**为 $\lceil \log_m(n(m-1)+1) \rceil$ 。

$$\begin{array}{l} n=10, \\ m=3 \end{array}$$

$$\text{最小高度} = \lceil \log_3(10 \times (3-1)+1) \rceil$$

$$= \lceil \log_3 21 \rceil$$

$$= 3$$





高度为 h 的 m 次树至多有  $\frac{m^h - 1}{m - 1}$  个结点.

$$n \leq \frac{m^h - 1}{m - 1} \quad (n \text{ 结点数})$$

$$\Rightarrow n(m-1) \leq m^h - 1$$

$$\Rightarrow n(m-1) + 1 \leq m^h$$

取对数  $\log_m [n(m-1) + 1] \leq h$ .

因为 h 要是 正整数, 取上限(吧)?

PP  $h \geq \lceil \log_m [n(m-1) + 1] \rceil$



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

性质4：

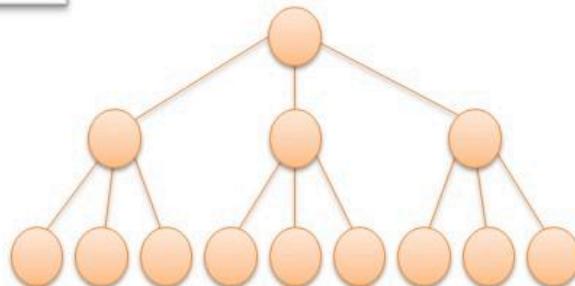
示例

含 $n$ 个结点的3次树的最小高度是多少？最大高度是多少？

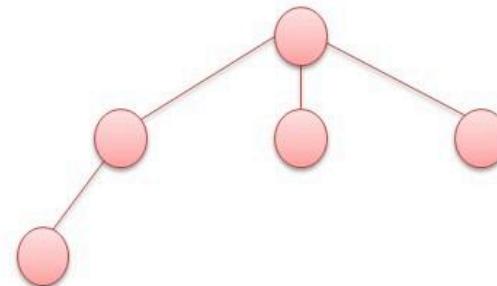
⇒ 看成是性质4的证明过程。

解

设含 $n$ 个结点的3次树的**最小高度为 $h$** ：



$m=3, h=3$ : 最多结点情况



$m=3, h=3$ : 最少结点情况



## 7.1 树的概念



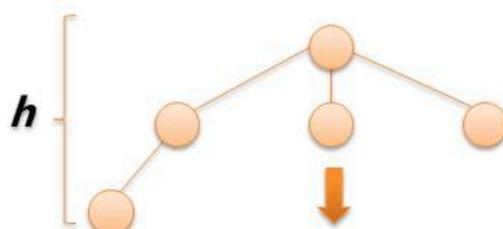
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

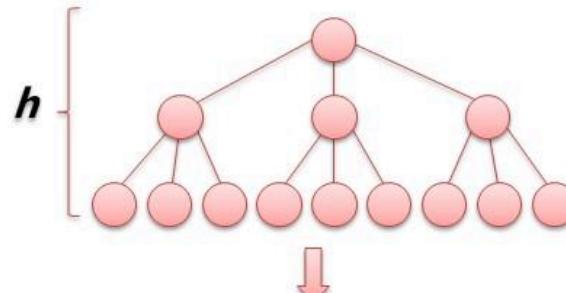
#### 性质4：

推广



最少结点情况，结点个数：

$$1 + 3^1 + 3^2 + \dots + 3^{h-2} + 1$$



最多结点情况，结点个数：

$$1 + 3^1 + 3^2 + \dots + 3^{h-1}$$

则有：

$$1 + 3^1 + 3^2 + \dots + 3^{h-2} < n \leq 1 + 3^1 + 3^2 + \dots + 3^{h-1}$$

$$(3^{h-1}-1)/2 < n \leq (3^{h-1})/2$$

$$3^{h-1} < 2n+1 \leq 3^h$$

$$\text{即: } h = \lceil \log_3(2n+1) \rceil.$$



## 7.1 树的概念



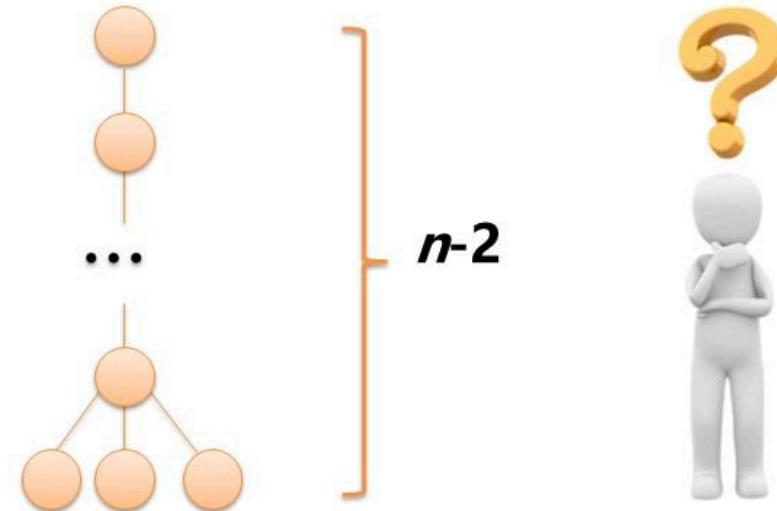
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

性质4:

3次数的最大高度?



最大高度为  $n-2$  (某一层有3个结点，其他每层只有一个结点)。



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.4 树的性质

#### 性质4：

**【例7.2】若一棵三次树中度为3的结点为2个，度为2的结点为1个，度为1的结点为2个，则该三次树中总的结点个数和叶子结点个数分别是多少？**

#### 解

- $m=3$
- $n_1=2, n_2=1, n_3=2$
- 所有结点度数之和 $=1 \times n_1 + 2 \times n_2 + 3 \times n_3 = 1 \times 2 + 2 \times 1 + 3 \times 2 = 10$ 。
- 所有结点度数之和 $=n-1 \Rightarrow n=11$
- $n_0=n-n_1-n_2-n_3=11-2-1-2=6$



## 7.1 树的概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.5 树的基本运算

**树的遍历运算是指按某种方式访问树中的每一个结点且每一个结点只被访问一次。**

**主要的遍历方法：**

- **先根遍历：**

若树不空，则先访问根结点，然后依次先根遍历各棵**子树**。

- **后根遍历：**

若树不空，则先依次后根遍历各棵**子树**，然后访问根结点。

- **层次遍历：**

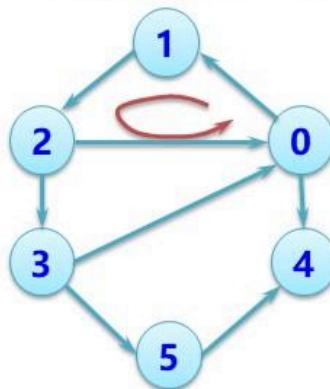
若树不空，则自上而下、自左至右访问树中每个结点。

**注意：先根和后根遍历算法都是递归的。**

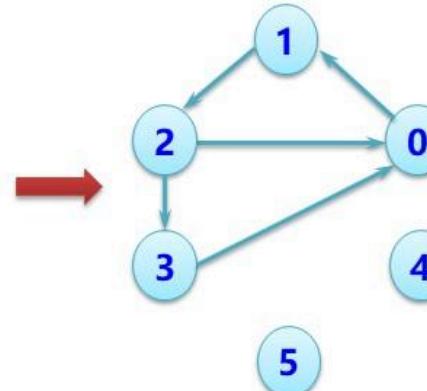


## 在一个非强连通中找强连通分量的方法：

- ① 在图中找有向环。
- ② 扩展该有向环：如果某个顶点到该环中任一顶点有路径，并且该环中任一顶点到这个顶点也有路径，则加入这个顶点。



一个非强连通图



3个强连通分量





## 7.1 树的概念

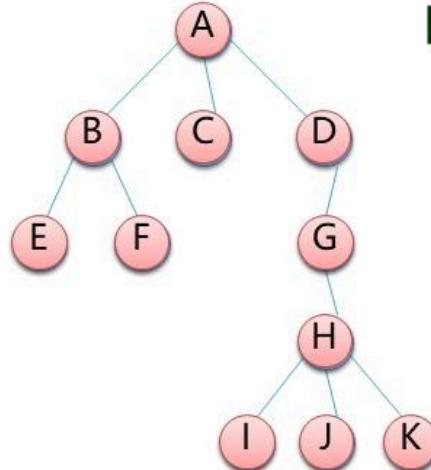


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.5 树的基本运算 树的后根遍历示例

若树不空，则先依次后根遍历各棵子树，然后访问根结点。



后根遍历的结点访问次序：

E F B C I J K H G D A

遍历完毕



## 7.1 树的概念

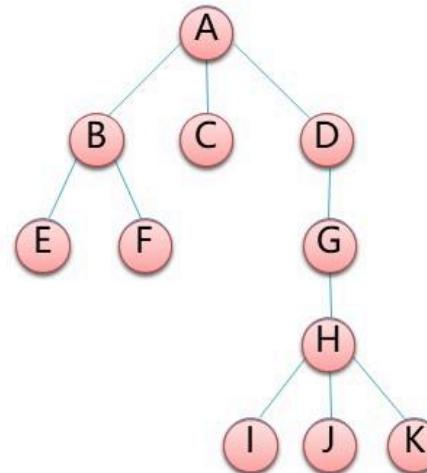


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.1.5 树的基本运算

#### 树的层次遍历示例



层次遍历的结点访问次序:

A B C D E F G H I J K

遍历完毕



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



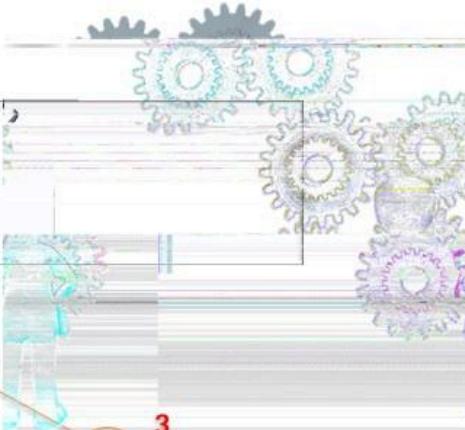
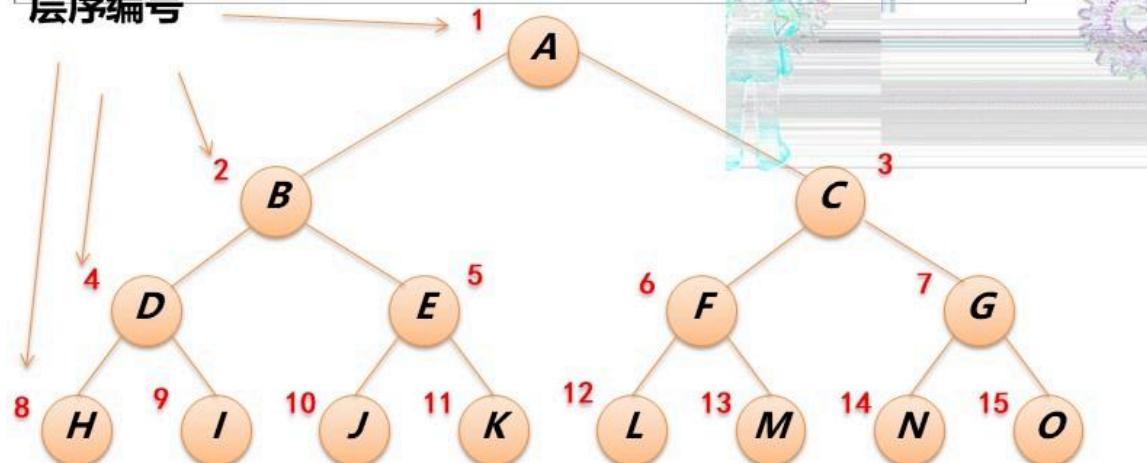
### 7.2.1 二叉树的定义

#### 两种特殊的二叉树

##### ① 满二叉树：在一棵二叉树中

- 如果所有分支结点都有双分结点；
- 并且叶结点都集中在二叉树的最下一层。

层序编号





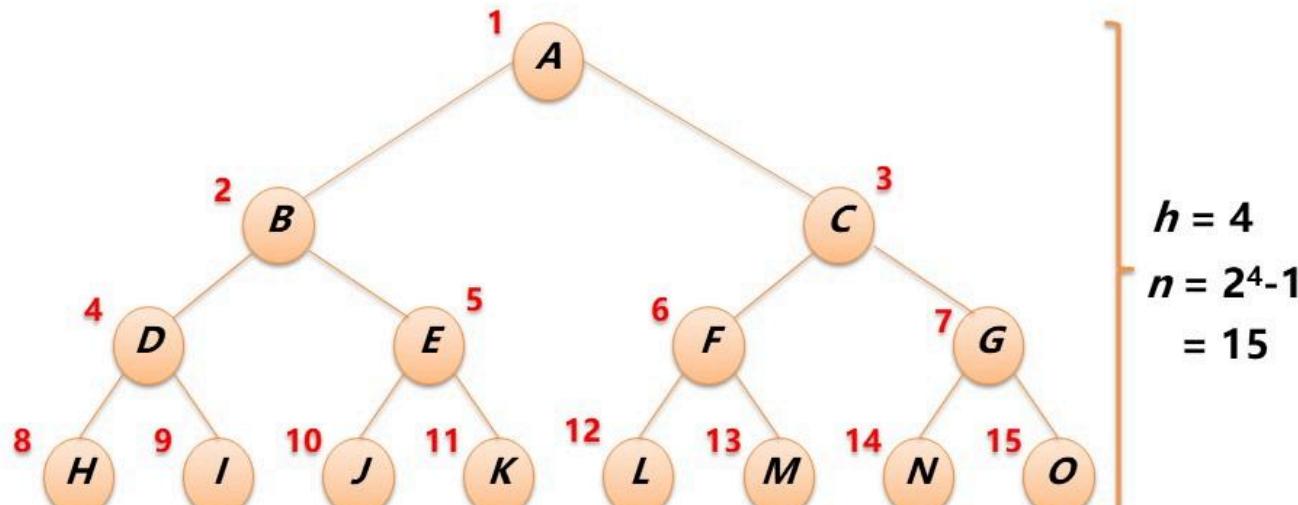
## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.1 二叉树的定义



层序编号:  $1 \sim 2^h - 1$

**满二叉树:** 在一棵二叉树中:

- 高度为  $h$  的二叉树恰好有  $2^h - 1$  个结点。



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

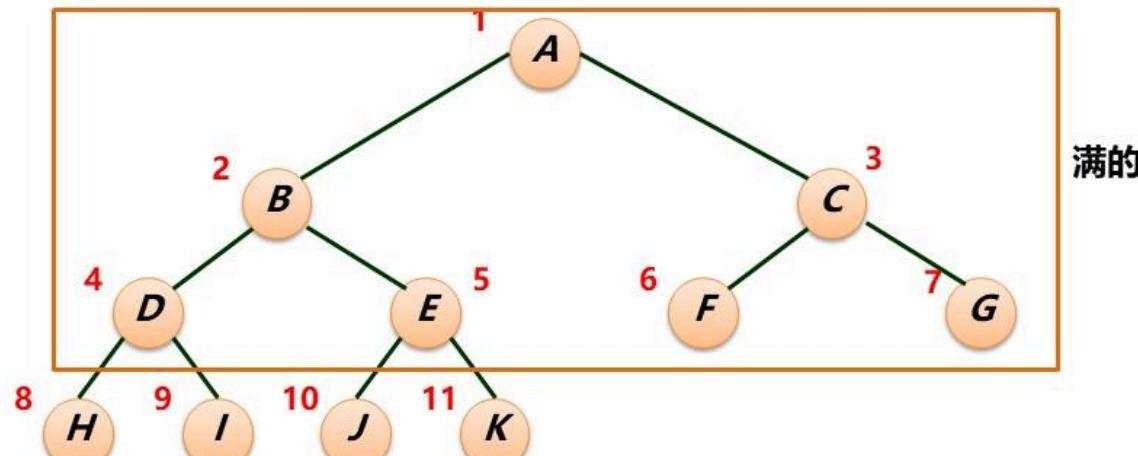


### 7.2.1 二叉树的定义

#### 两种特殊的二叉树

② 完全二叉树：在一棵二叉树中

- 最多只有下面两层的结点的度数小于2
- 并且最下面一层的叶结点都依次排列在该层最左边的位置上。



完全二叉树实际上是对应的满二叉树删除叶结点层最右边若干个结点得到的。



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

**性质1：**

**非空二叉树上叶结点数等于双分支结点数加1。即： $n_0 = n_2 + 1$ 。**

**度之和=分支数**

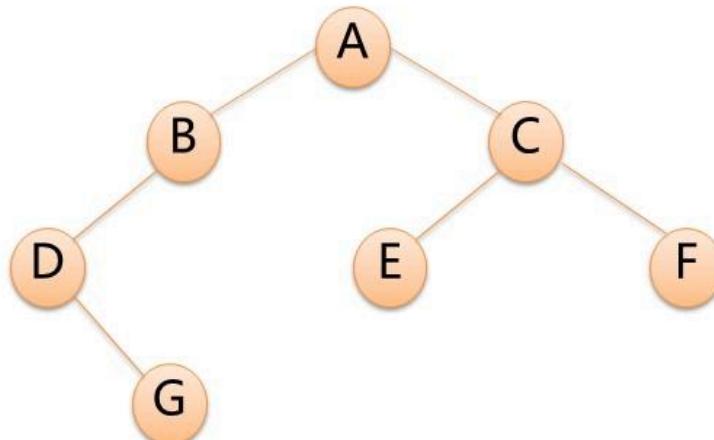
**分支数= $n-1$**

**$n = n_0 + n_1 + n_2$**

**度之和  
 $= n_1 + 2n_2$**

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

$$\downarrow$$
  
$$n_0 = n_2 + 1$$





## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

非空二叉树上叶结点数等于双分支结点数加1。即： $n_0 = n_2 + 1$ 。

#### 求解一般二叉树结点个数方法归纳

通常利用二叉树的性质1，即 $n_0 = n_2 + 1$ 来求解这类问题，常利用以下关系求解：

$$n = n_0 + n_1 + n_2$$

$$\text{度之和} = n - 1$$

$$\text{度之和} = n_1 + 2n_2$$

所以有：  $n = n_1 + 2n_2 + 1$



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

非空二叉树上叶结点数等于双分支结点数加1。即： $n_0 = n_2 + 1$ 。

示例

一棵二叉树中有7个度为2的结点和5个度为1的结点，其总共  
有（ ）个结点。

- A.16    B.18    C.20    D.30

■  $n_2 = 7, n_1 = 5 \quad \square \quad n_0 = n_2 + 1 = 8$

■ 结点总数  $n = n_0 + n_1 + n_2 = 20$ 。

■ 答案为C。



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

**性质2 非空二叉树上第*i*层上至多有 $2^{i-1}$ 个结点 ( $i \geq 1$ )。**

由树的性质2可推出。

**性质3 高度为*h*的二叉树至多有 $2^h - 1$ 个结点 ( $h \geq 1$ )。**

由树的性质3可推出。





## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

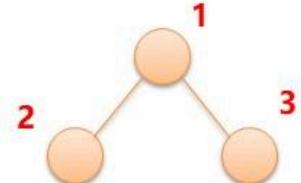


### 7.2.2 二叉树性质

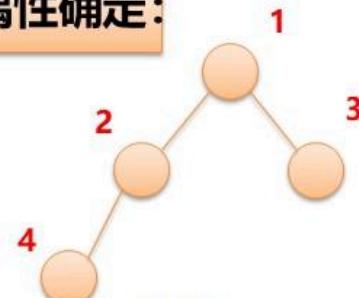
性质4：

完全二叉树性质（含n为结点）：

①  $n_1=0$  或者  $n_1=1$ 。 $n_1$  可由  $n$  的奇偶性确定：



$n$  为奇数   $n_1=0$



$n$  为偶数   $n_1=1$

② 若  $i \leq \lfloor n/2 \rfloor$ ，则编号为  $i$  的结点为分支结点，否则为叶结点。

•  $n=3$    $\lfloor n/2 \rfloor = 1$ ，编号为 1 的是分支结点；编号为 2、3 的是叶结点

•  $n=4$    $\lfloor n/2 \rfloor = 2$ ，编号为 1、2 的是分支结点；编号为 3、4 的是叶结点



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

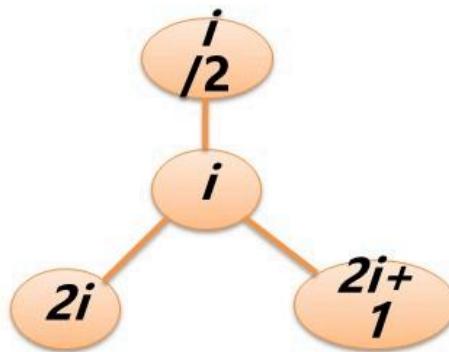


### 7.2.2 二叉树性质

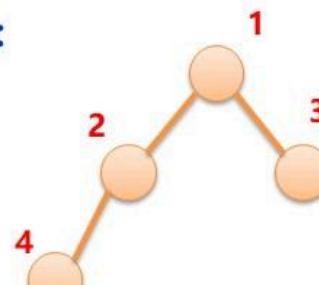
性质4：

完全二叉树性质（含n为结点）：

- ③ 除树根结点外，若一个结点的编号为*i*，则它的双亲结点的编号为 $\lfloor i/2 \rfloor$ 。
- ④ 若编号为*i*的结点有左孩子结点，则左孩子结点的编号为 $2i$ ；若编号为*i*的结点有右孩子结点，则右孩子结点的编号为 $2i+1$ 。



例如：





## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

具有 $n$ 个结点的 $m$ 次树的最小高度为 $\lceil \log_m(n(m-1)+1) \rceil$ 。

#### 求解完全二叉树结点个数方法归纳

对于完全二叉树：

由 $n$ 的奇偶性确定 $n_1$  ( $n$ 为奇数  $n_1=0$ ,  $n$ 为偶数  $n_1=1$ )

$$n_0 = n_2 + 1$$

$$n = n_0 + n_1 + n_2 = 2n_2 + n_1 + 1 \quad \square \quad n_2 = (n - n_1 - 1)/2$$

当 $n$ 确定时,  $n_0$ 、 $n_1$ 、 $n_2$ 都是确定的, 其树形也可以确定!

$$h = \lceil \log_2(n+1) \rceil \quad \square \quad \text{或者} \quad \lfloor \log_2 n \rfloor + 1$$



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

示例

已知一棵完全二叉树的第6层（设根为第1层）有8个叶子结点，则该完全二叉树的结点个数最多是（ ）。

- A. 39      B. 52      C. 111      D. 119

2009年全国计算机专业硕士学位研究生考试题目

- 完全二叉树的叶子结点只能在最下两层，对于本题，结点最多的情况是第6层为倒数第二层，即1~6层构成一个满二叉树，其结点总数为 $2^6 - 1 = 63$ 。
- 其中第6层有 $2^5 = 32$ 个结点，含8个叶子结点，则另外有 $32 - 8 = 24$ 个非叶子结点，它们中每个结点有两个孩子结点（均为第7层的叶子结点），计48个叶子结点。这样最多的结点个数 $= 63 + 48 = 111$ 。
- 答案为C。



## 7.2 二叉树的概念和性质



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.2.2 二叉树性质

示例

一棵完全二叉树中有8个叶子结点，则高度至多是（ ）。

- A.3    B.4    C.5    D.不确定

- 该完全二叉树， $n_0=8$ ， $n_2=n_0-1=7$ ，则  
 $n=n_0+n_1+n_2=15+n_1$ 。
- 完全二叉树中 $n_1=0$ 或 $n_1=1$ ，则 $n_1=1$ 时结点个数最多，此时  
 $n=16$ 。
- 最大高度 $h=\lceil \log_2(n+1) \rceil = 5$ 。
- 答案为C。



## 7.3 二叉树的存储结构



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.3.2 二叉树的链式存储结构

**借鉴树的孩子链存储结构 □ 二叉树的链式存储结构。**

**在二叉树的链式存储中，结点的类型声明如下：**

```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;
```



**指向的都是二叉树：递归性**



## 7.3 二叉树的存储结构

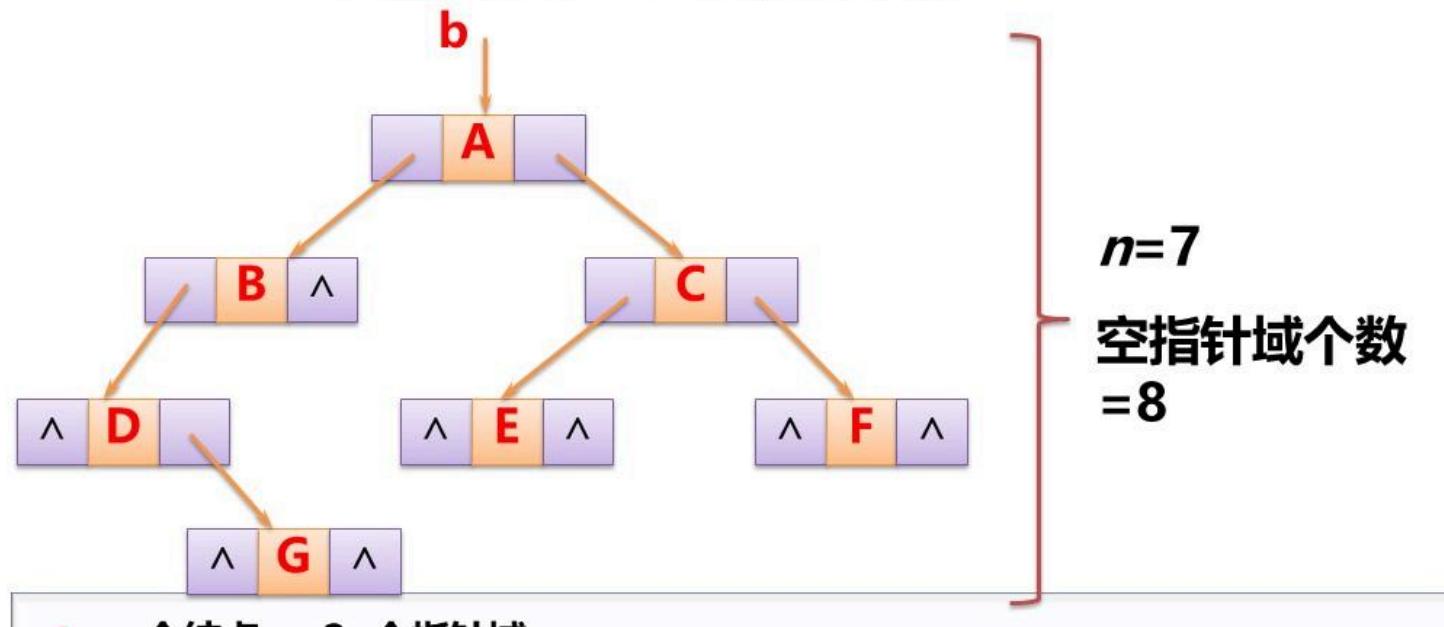


清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.3.2 二叉树的链式存储结构

在二叉链中，空指针的个数？



- $n$ 个结点  $2n$ 个指针域
- 分支数为  $n-1$  非空指针域有  $n-1$ 个
- 空指针域个数 =  $2n-(n-1) = n+1$



## 7.4 二叉树基本运算及其实现



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.4.1 二叉树的基本运算概述

归纳起来，二叉树有以下基本运算：

- ① 创建二叉树**CreateBTree(\*b, \*str)**：根据二叉树括号表示法字符串str生成对应的二叉链存储结构b。
- ② 销毁二叉链存储结构**DestroyBTree(\*b)**：销毁二叉链b并释放空间。
- ③ 查找结点**FindNode(\*b, x)**：在二叉树b中寻找data域值为x的结点，并返回指向该结点的指针。

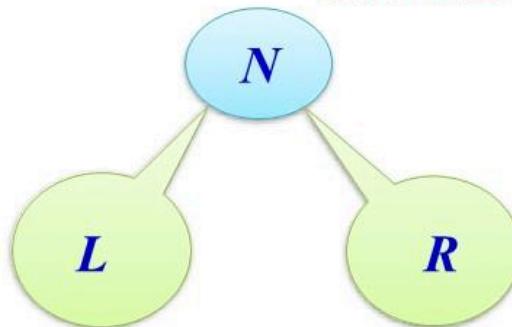
### (1) 创建二叉树CreateBTNode(\*b,\*str)

由正确的二叉树括号表示串  $\Rightarrow$  二叉链存储结构  
逻辑结构  $\xrightarrow{\text{映射}}$  存储结构

正确的二叉树括号表示串中只有4类字符：

- 单个字符：节点的值
- (：表示一棵左子树的开始
- )：表示一棵子树的结束
- ,：表示一棵右子树的开始

## 算法设计：



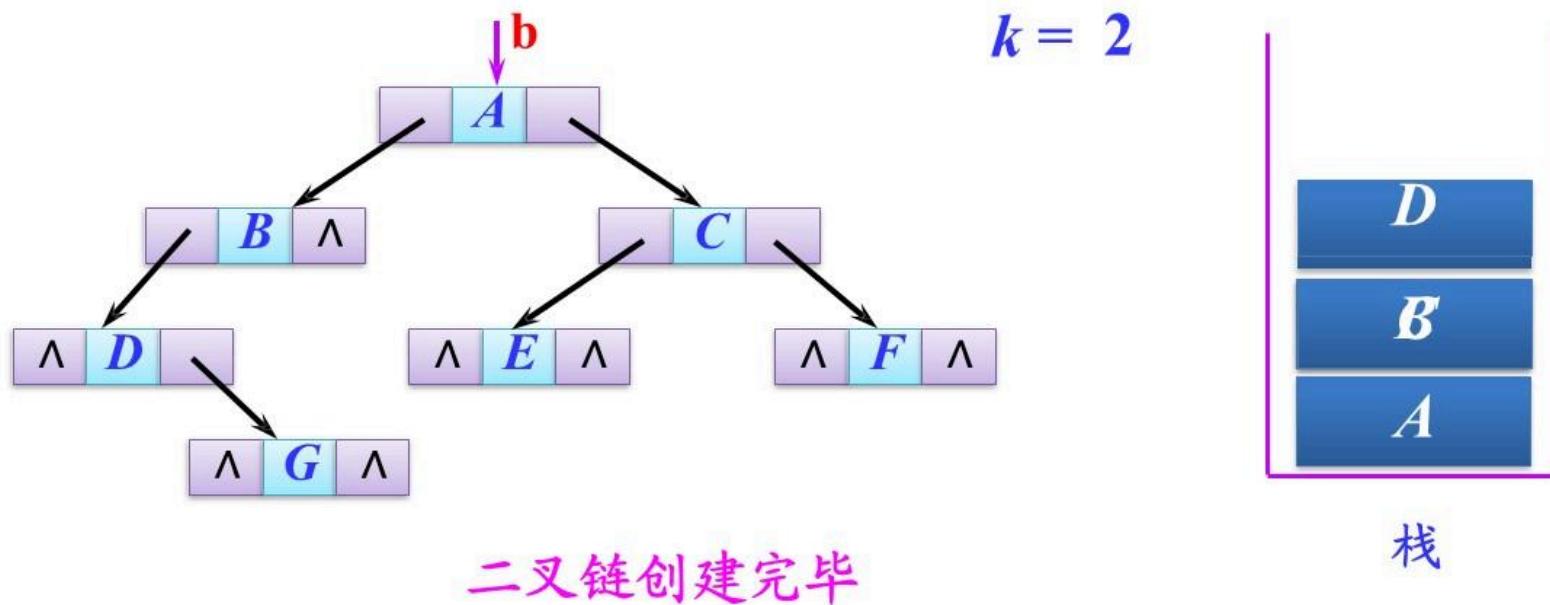
- 先构造根节点**N**，再构造左子树**L**，最后构造右子树**R**
- 构造右子树**R**时，找不到**N**了，所以需要保存**N**
- 而节点是按最近原则匹配的，所以使用一个**栈**保存**N**

## 用ch扫描采用括号表示法表示二叉树的字符串：

- ① 若 $\text{ch}='('$ ：则将前面刚创建的节点作为双亲节点进栈，并置 $k=1$ ，表示开始处理左孩子节点；
- ② 若 $\text{ch}=')'$ ：表示栈顶节点的左、右孩子节点处理完毕，退栈；
- ③ 若 $\text{ch}=','$ ：表示开始处理右孩子节点，置 $k=2$ ；
- ④ 其他情况（节点值）：
  - 创建 $*\text{p}$ 节点用于存放 $\text{ch}$ ；
  - 当 $k=1$ 时，将 $*\text{p}$ 节点作为栈顶节点的左孩子节点；当 $k=2$ 时，将 $*\text{p}$ 节点作为栈顶节点的右孩子节点。

## 根据括号表示法字符串构造二叉链的演示

$A ( B ( D ( , G ) ) , C ( E , F ) )$





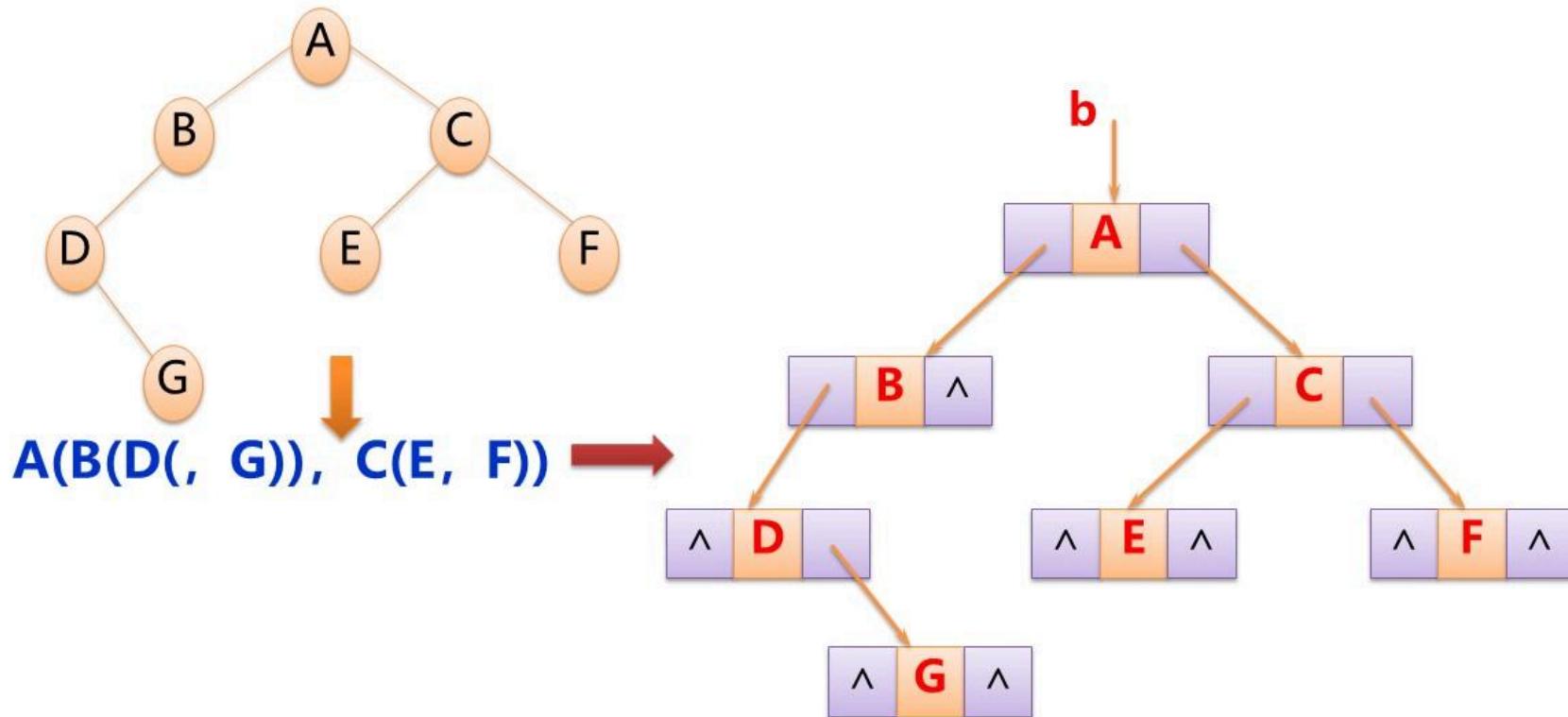
## 7.4 二叉树基本运算及其实现



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.4.1 二叉树的基本运算概述



```
void CreateBTNode(BTNode * &b,char *str)
{
    //由str⇒二叉链b
    BTNode *St[MaxSize], *p;
    int top=-1, k ,j=0;
    char ch;

    b=NULL;           //建立的二叉链初始时为空
    ch=str[j];
    while (ch!='\0') //str未扫描完时循环
    {
        switch(ch)
        {
            case '(': top++; St[top]=p; k=1; break;      //可能有左孩子节点，进栈
            case ')': top--; break;                      //后面为右孩子节点
            case ',': k=2; break;
        }
    }
}
```

```
default:          //遇到节点值
    p=(BTNode *)malloc(sizeof(BTNode));
    p->data=ch; p->lchild=p->rchild=NULL;

if (b==NULL)      //p为二叉树的根节点
    b=p; else
    {
        switch(k)
        {
            case 1: St[top]->lchild=p; break; case 2:
            St[top]->rchild=p; break;
        }
    }

j++; ch=str[j];      //继续扫描str
}
```



## 7.4 二叉树基本运算及其实现



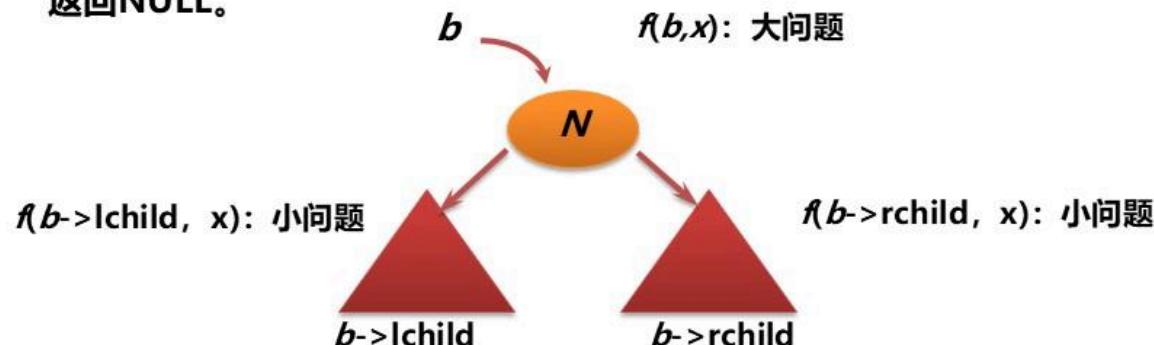
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.4.1 二叉树的基本运算概述

#### (3) 查找结点FindNode(\*b, x)

设 $f(b, x)$ 在二叉树 $b$ 中查找值为 $x$ 的结点（唯一）。找到后返回其指针，否则返回NULL。



递归模型如下：

$f(b, x) = \text{NULL}$	若 $b = \text{NULL}$
$f(b, x) = b$	若 $b->\text{data} == x$
$f(b, x) = p$	若在左子树中找到了，即 $p = f(b->\text{lchild}, x)$ 且 $p \neq \text{NULL}$
$f(b, x) = f(b->\text{rchild}, x)$	其他情况



## 7.4 二叉树基本运算及其实现



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 7.4.1 二叉树的基本运算概述 对应的递归算法如下：

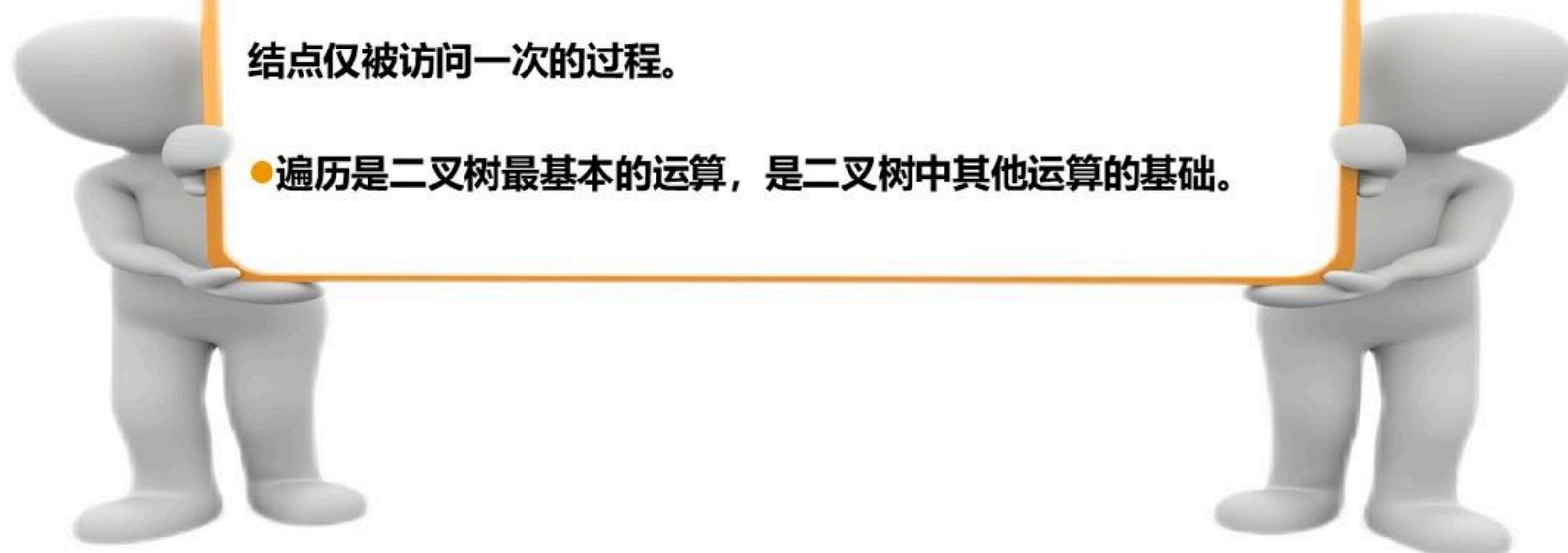
```
BTNode *FindNode(BTNode *b, ElemtType x)
{ BTNode *p;
  if (b==NULL)
    return NULL;
  else if (b->data==x)
    return b;
  else
  { p=FindNode(b->lchild, x);
    if (p!=NULL)
      return p;
    else
      return FindNode(b->rchild, x);
  }
}
```



### 7.5.1 二叉树遍历的概念

```
typedef struct node  
{ ElemtType data;  
    struct node *lchild, *rchild;  
} BTNode;
```

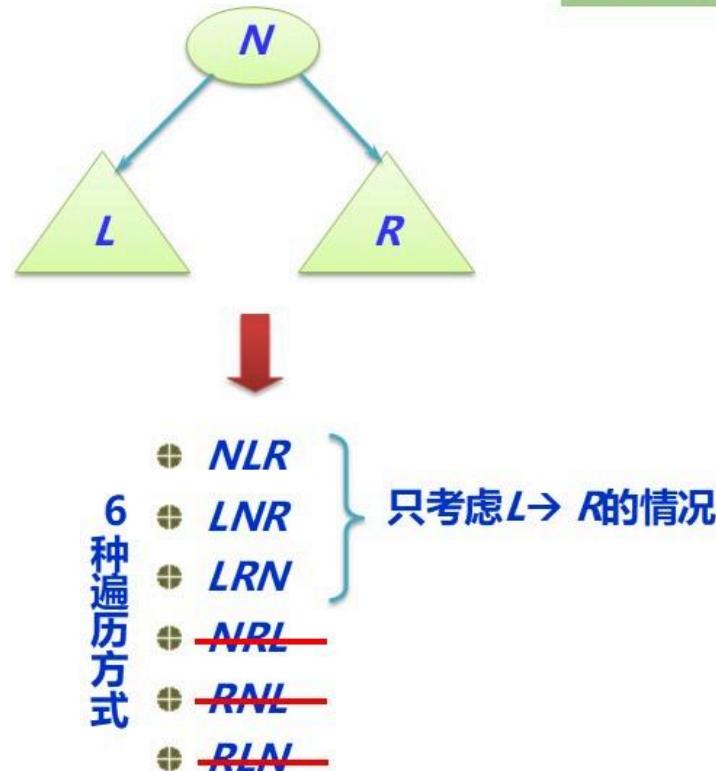
- 二叉树遍历是指按照一定次序访问树中所有结点，并且每个结点仅被访问一次的过程。
- 遍历是二叉树最基本的运算，是二叉树中其他运算的基础。





### 7.5.1 二叉树遍历的概念

二叉树的组成：



```
typedef struct node  
{ ElemType data;  
    struct node *lchild, *rchild;  
} BTNode;
```



### 7.5.1 二叉树遍历的概念

#### 1. 先序遍历

先序遍历NLR二叉树的过程是：

```
typedef struct node
{
    ElemType data;
    struct node *lchild, *rchild;
} BTNode;
```



- 访问根结点；
- 先序遍历左子树；
- 先序遍历右子树。



先序序列的第一个结点是根结点



### 7.5.1 二叉树遍历的概念

#### 2. 中序遍历

中序遍历LNR二叉树的过程是：

```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;
```

- 
- 中序遍历左子树；
  - 访问根结点；
  - 中序遍历右子树。



中序序列的根结点左边是左子树的结点，右边是右子树的结点。



### 7.5.1 二叉树遍历的概念

#### 3. 后序遍历

后序遍历LRN二叉树的过程是：

```
typedef struct node
{
    ElemenType data;
    struct node *lchild, *rchild;
} BTNode;
```

- 后序遍历左子树；
- 后序遍历右子树；
- 访问根结点。



后序序列的最后一个结点是根结点

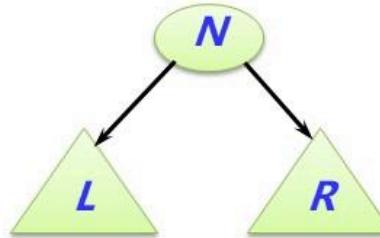


以若一颗二叉树的先序序列和后序序列正好相反。该二叉树的形态是什么？

示例



先序序列



后序序列： $L\ R\ N$   
后序序列的反序

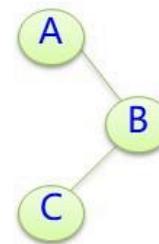
$N\ L\ R$

$L$ 为空或者 $R$ 为空时成立

$N\ R\ L$

这样的二叉树每层只有一个结点，即二叉树的形态是其高度等于结点个数。

例如





### 7.5.1 二叉树遍历的概念

#### 4. 层次遍历

层次遍历二叉树的过程是：

若二叉树非空（假设其高度为 $h$ ），则：

- (1) 访问根结点（第1层）；
- (2) 从左到右访问第2层的所有结点；
- (3) 从左到右访问第3层的所有结点、...、第 $h$ 层的所有结点。

层次序列的第一个结点是根结点

```
typedef struct node  
{ ElemType data;  
    struct node *lchild, *rchild;  
} BTNode;
```





## 7.5.2 先序、中序、后序遍历的递归算法

```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;
```

由二叉树的三种遍历过程直接得到3种递归算法。

**先序遍历**的递归算法：

```
void PreOrder(BTNode *b)
{ if (b!=NULL)
  { printf("%c ", b->data);      //访问根结点
    PreOrder(b->lchild);
    PreOrder(b->rchild);
  }
}
```





## 7.5.2 先序、中序、后序遍历的递归算法

```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;
```

**中序遍历**的递归算法：

```
void InOrder(BTNode *b)
{ if (b!=NULL)
  { InOrder(b->lchild);
    printf("%c ", b->data);      //访问根结点
    InOrder(b->rchild);
  }
}
```





## 7.5.2 先序、中序、后序遍历的递归算法

```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;
```

### 后序遍历的递归算法：

```
void PostOrder(BTNode *b)
{ if (b!=NULL)
  { PostOrder(b->lchild);
    PostOrder(b->rchild);
    printf("%c ", b->data);      //访问根结点
  }
}
```



```
typedef struct node  
{ ElementType data;  
    struct node *lchild, *rchild;  
} BTNode;
```



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



遍历完毕

以先序为例：

PreOrder(A)

→ 访问A

PreOrder(B)

→ 访问B

PreOrder(C)

PreOrder(D)

PreOrder(NUL L)

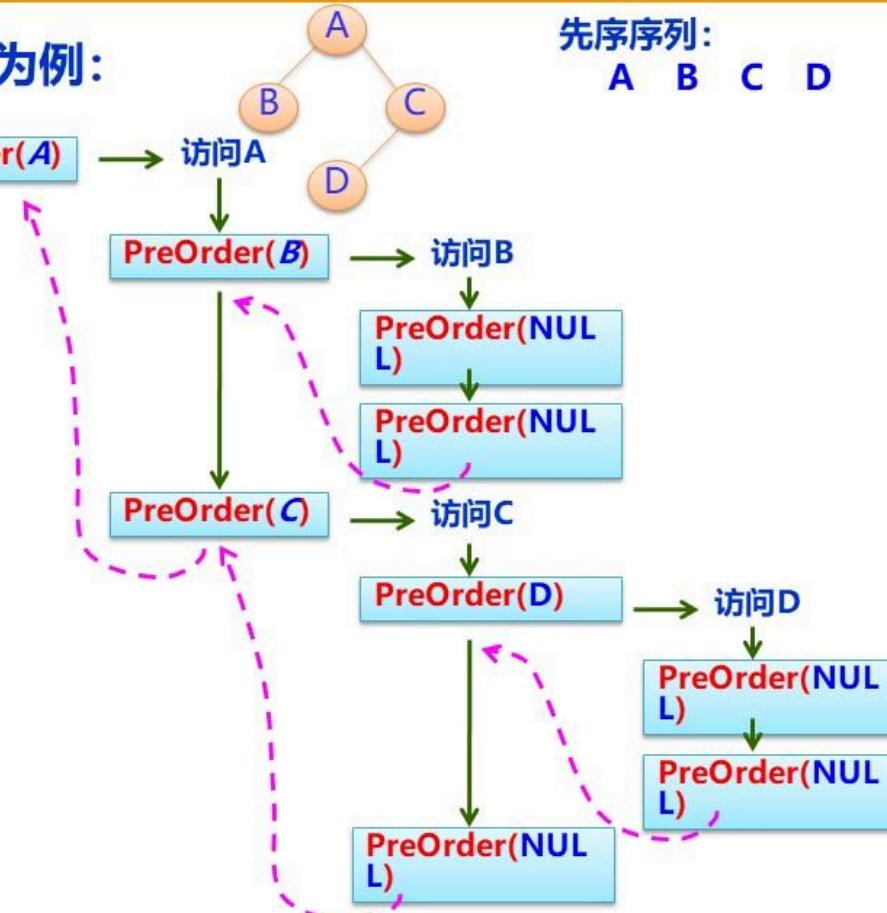
PreOrder(NUL L)

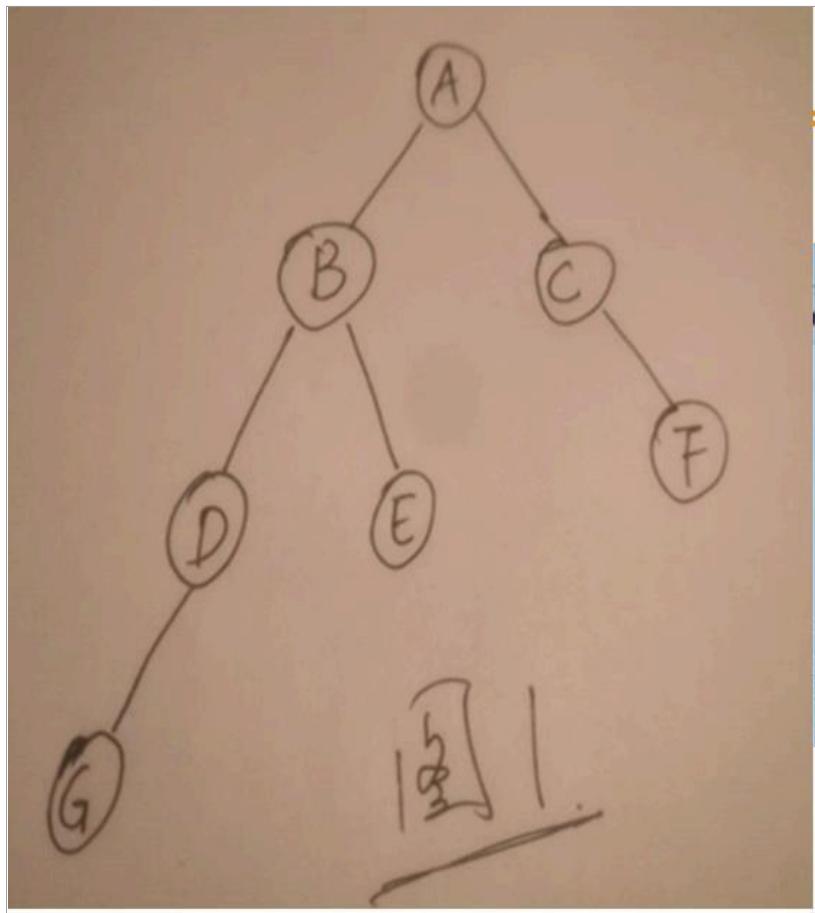
PreOrder(NUL L)

PreOrder(NUL L)

先序序列：

A B C D





```
typedef struct node  
{ ElemtType data;  
    struct node *lchild, *rchild;  
} BTNode;
```

先序遍历:

A B D G E C F

中序遍历:

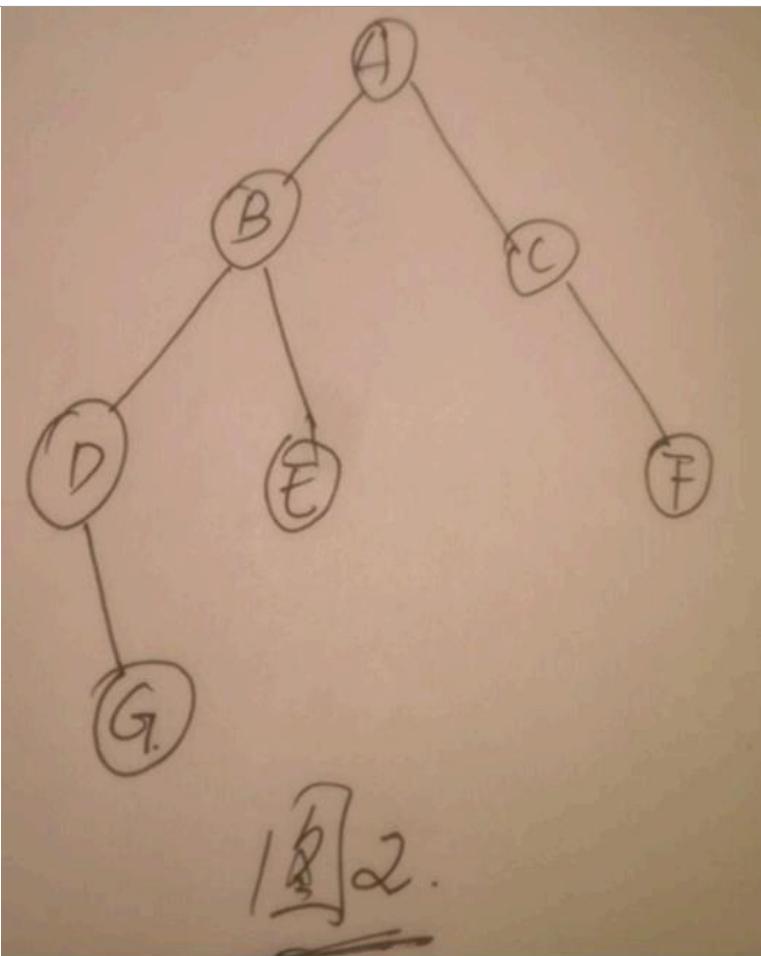
G D B E A C F

后序遍历:

G D E B F C A

Process exited after 0.4806 seconds with return value 0

请按任意键继续. . .



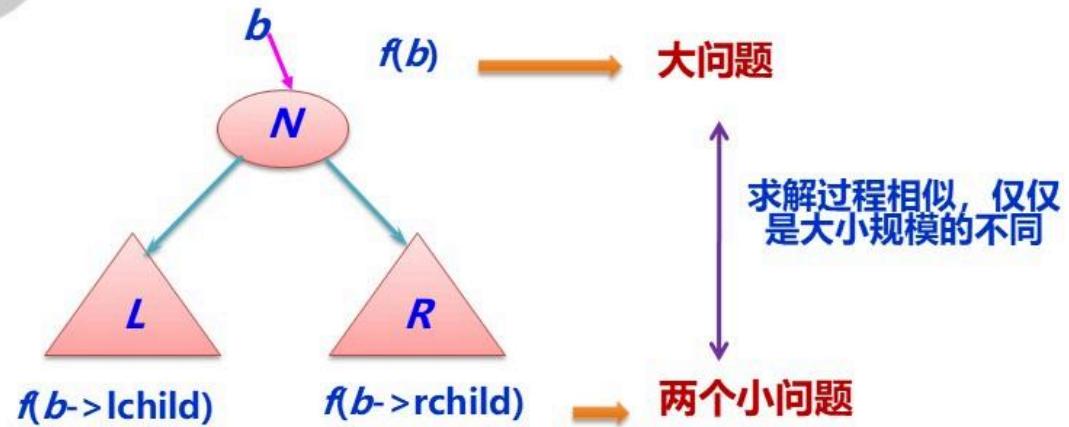
```
typedef struct node
{ ElemType data;
  struct node *lchild, *rchild;
} BTNode;

先序遍历:  
A B D G E C F  
中序遍历:  
D G B E A C F  
后序遍历:  
G D E B F C A  
-----  
Process exited after 0.7459 seconds with return value 0  
请按任意键继续. . .
```



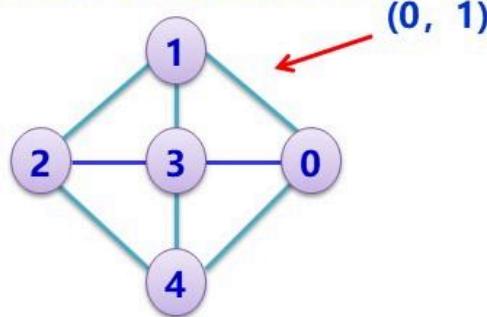
## 二叉树3种递归遍历算法的应用

基本思路

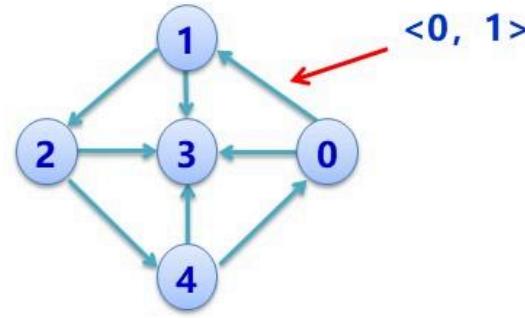




在图G中，如果代表边的顶点对是无序的，则称G为**无向图**。用圆括号序偶表示无向边。



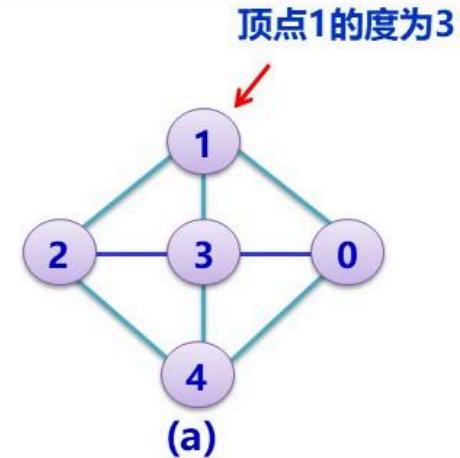
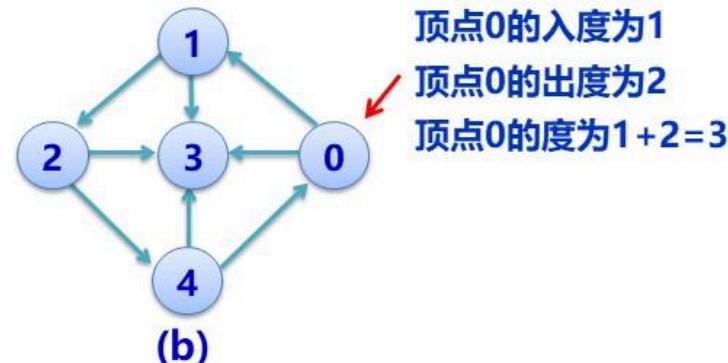
如果表示边的顶点对是有序的，则称G为**有向图**。用尖括号序偶表示有向边。





## 2、顶点的度、入度和出度

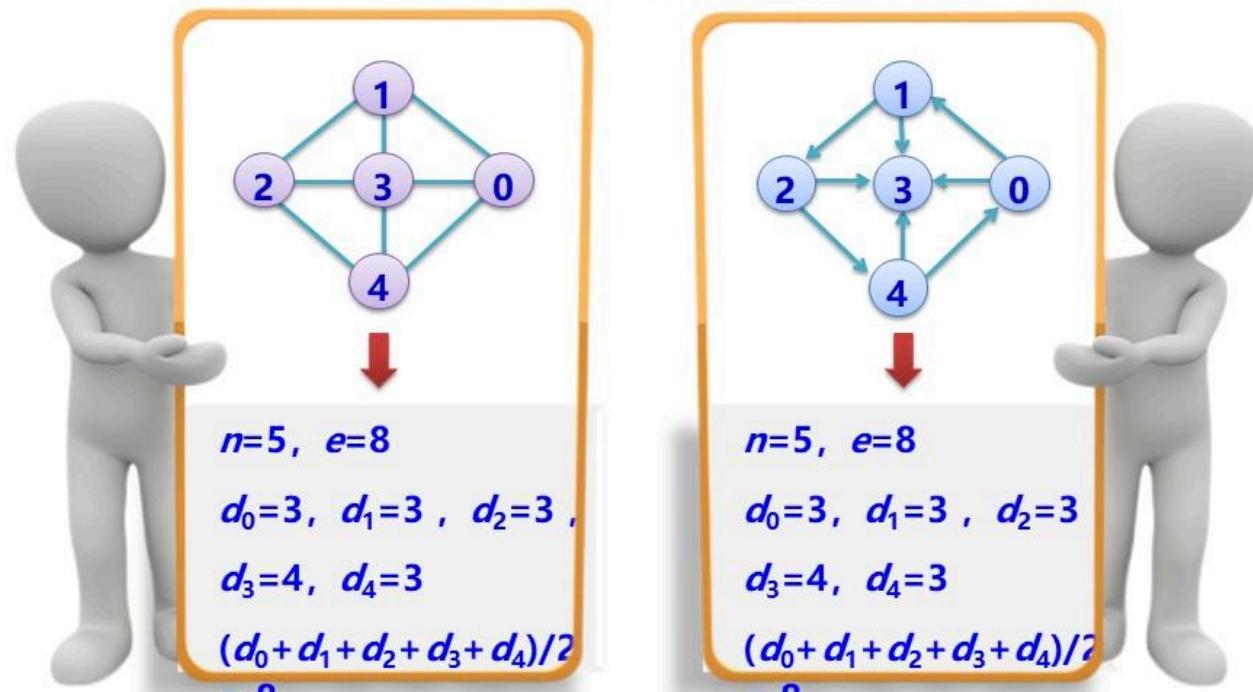
- 无向图中，以顶点*v*为端点的边数称为该顶点的度。
- 有向图中，以顶点*v*为终点的入边的数目，称为该顶点的入度。以顶点*v*为始点的出边的数目，称为该顶点的出度。一个顶点的入度与出度的和为该顶点的度。





若一个图中有 $n$ 个顶点和 $e$ 条边，每个顶点的度为 $d_i$  ( $0 \leq i \leq n-1$ )，  
则有：

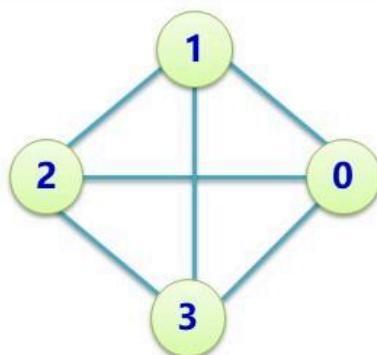
$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i$$



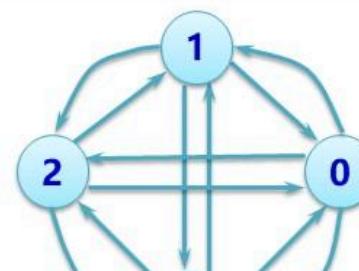


### 3、完全图

- 无向图中，每两个顶点之间都存在着一条边，称为**完全无向图**，包含有 $n(n-1)/2$ 条边。
- 有向图中，每两个顶点之间都存在着方向相反的两条边，称为**完全有向图**，包含有 $n(n-1)$ 条边。



完全无向图：  $n=4$ ,  
 $e=n(n-1)/2=6$



完全有向图：  $n=4$ ,  
 $e=n(n-1)=12$



示例

设G是一个含有6个顶点的无向图，该图至多有（ ）条边。

- A. 5      B. 6    C. 7      D. 15

边数最多时为完全无向图，有 $n(n-1)/2=15$ 条边。

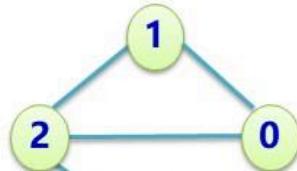
答案为D。



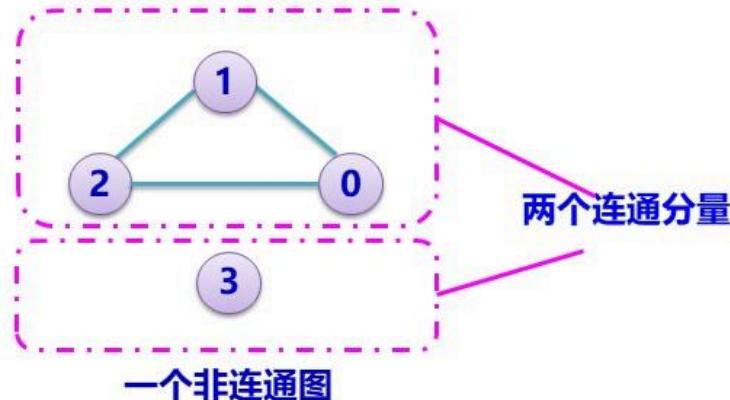
## 8、连通、连通图和连通分量

无向图中：

- 若从顶点到顶点有路径，则称顶点和是连通的。
- 若图中任意两个顶点都连通，则称为连通图，否则称为非连通图。
- 无向图G中的极大连通子图称为G的连通分量。显然，任何连通图的连通分量只有一个，即本身，而非连通图有多个连通分量。



一个连通图



一个非连通图

两个连通分量



示例

设G是一个非连通无向图，有15条边，则该图至少有( )个顶点。

- A. 5    B. 6    C. 7    D. 15

- 要使顶点个数最少并且为非连通无向图，图由两个连通分量构成：完全无向图+单个顶点。
- 完全无向图： $n(n-1)/2=15 \Rightarrow n=6$ 。
- 单个顶点：1个顶点。
- 对应的图有 $6+1=7$ 个顶点。

答案为C。

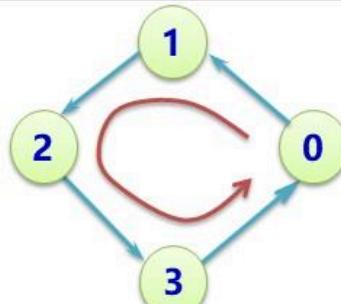




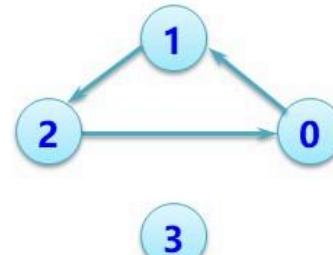
## 9、强连通图和强连通分量

有向图中：

- 若从顶点 $i$ 到顶点 $j$ 有路径，则称从顶点 $i$ 到 $j$ 是连通的。
- 若图G中的任意两个顶点 $i$ 和 $j$ 都连通，即从顶点 $i$ 到 $j$ 和从顶点 $j$ 到 $i$ 都存在路径，则称图G是强连通图。



一个强连通图

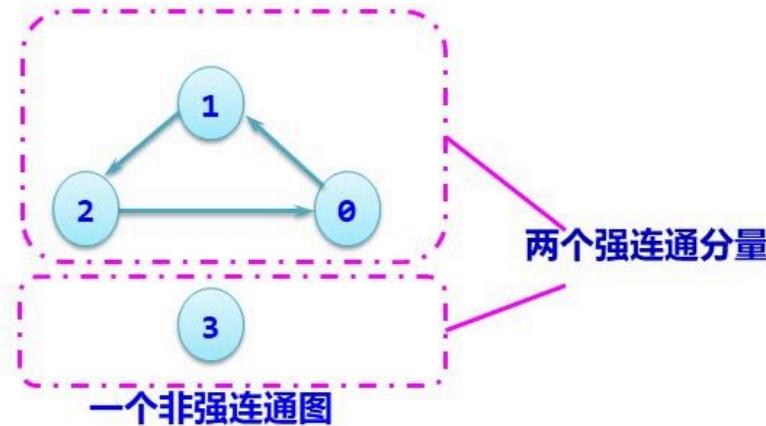


一个非强连通图





- 有向图G中的极大强连通子图称为G的**强连通分量**。
- 强连通图只有一个强连通分量，即本身。非强连通图有多个强连通分量。





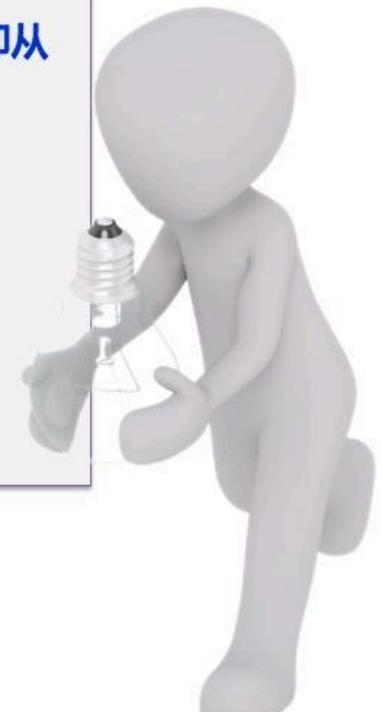
**【例8.1】**  $n$  ( $n > 2$ ) 个顶点的强连通图至少有多少条边？这样的有向图是什么形状？

根据强连通图的定义可知，图中的任意两个顶点和都连通，即从顶点到顶点和从顶点到顶点都存在路径。

这样，每个顶点的度  $d \geq 2$ ，设图中总的边数为  $e$ ，有：

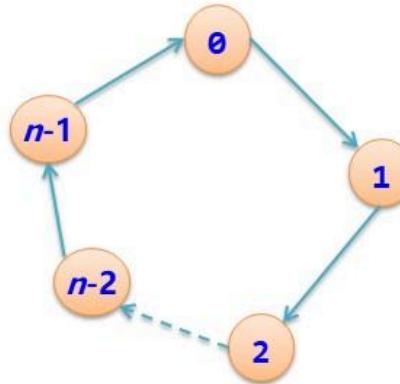
$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i \geq \frac{1}{2} \sum_{i=0}^{n-1} 2 = n$$

即： $e \geq n$ 。因此， $n$  个顶点的强连通图至少有  $n$  条边。





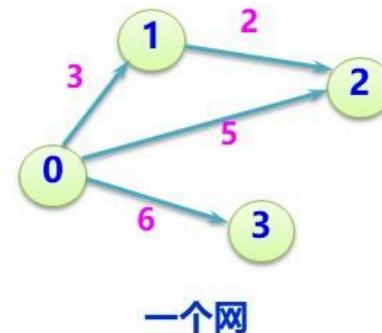
而只有 $n$ 条边的强连通图是环形的，即顶点0到1有一条有向边，顶点1到2有一条有向边，...，顶点 $n-1$ 到0有一条有向边，如下图所示。





## 10、权和网

- 图中每一条边都可以附带有一个对应的数值，这种与边相关的数值称为**权**。权可以表示从一个顶点到另一个顶点的距离或花费的代价。
- 边上带有权的图称为**带权图**，也称作**网**。



## 8.2 图的存储结构和基本运算算法



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图的两种主要存储结构：

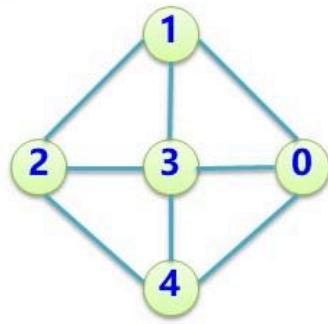
- 邻接矩阵
- 邻接表

- 存储每个顶点的信息
- 存储每条边的信息





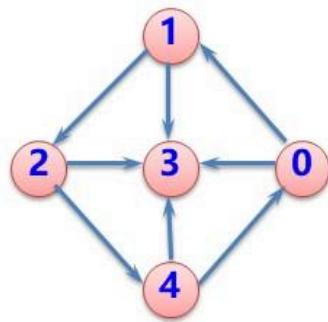
## 邻接矩阵演示



$$A_1 = \begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \left[ \begin{matrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{matrix} \right] \end{matrix}$$

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \left[ \begin{matrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{matrix} \right] \end{matrix}$$

对称



$$A_2 =$$

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \left[ \begin{matrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{matrix} \right] \end{matrix}$$

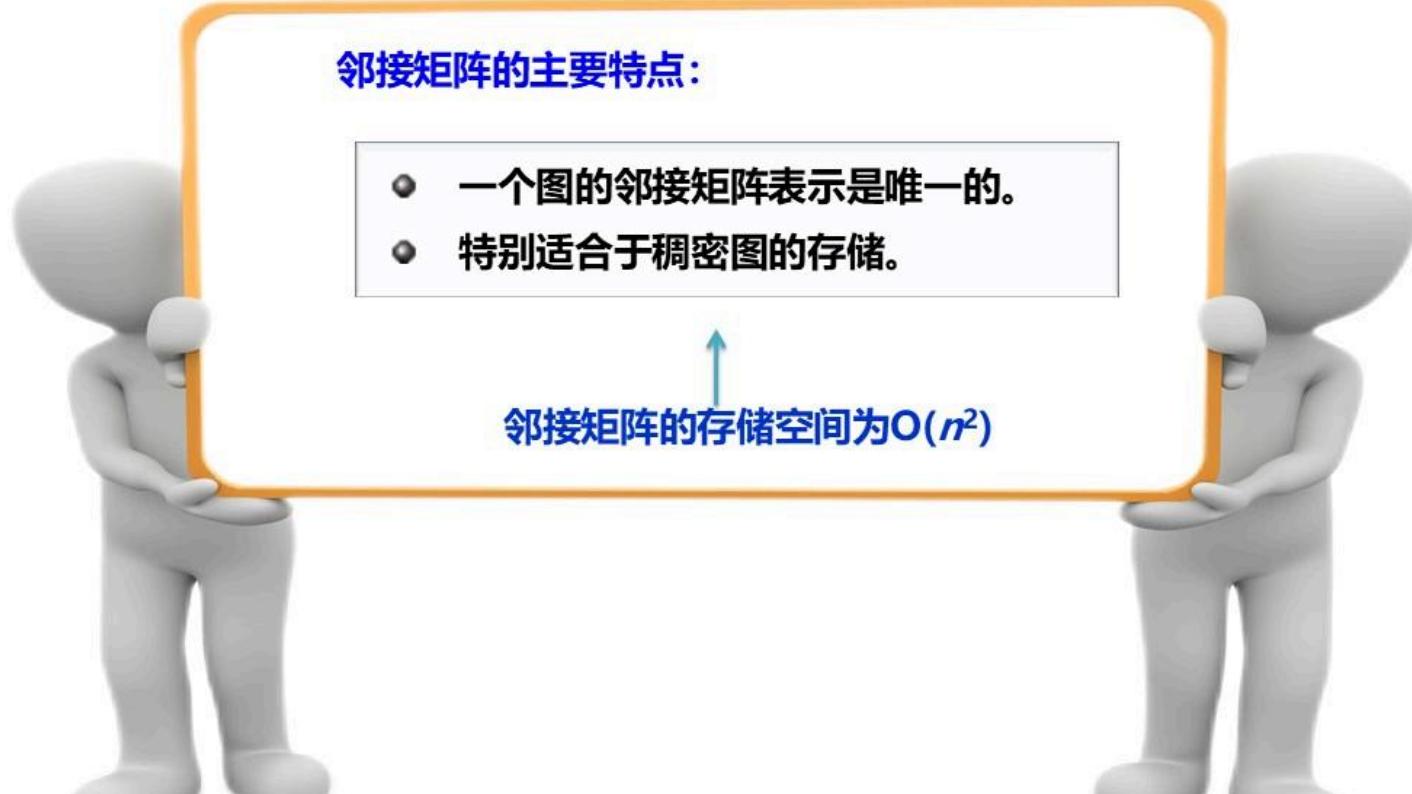
不对称



### 邻接矩阵的主要特点：

- 一个图的邻接矩阵表示是唯一的。
- 特别适合于稠密图的存储。

邻接矩阵的存储空间为 $O(n^2)$





## 图的邻接矩阵存储类型定义如下：

```
#define MAXV <最大顶点个数>
typedef struct
{ int no;           //顶点编号
  InfoType info;    //顶点其他信息
} VertexType;

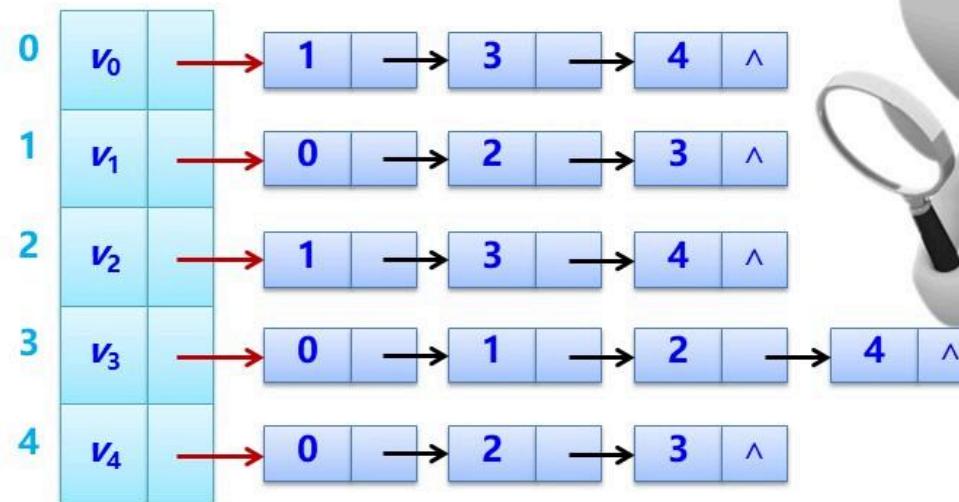
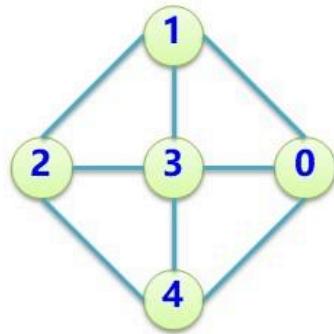
typedef struct      //图的定义
{ int edges[MAXV][MAXV]; //邻接矩阵
  int n, e;          //顶点数, 边数
  VertexType vexs[MAXV]; //存放顶点信息
} MatGraph;
```

声明顶点  
的类型

声明的邻接  
矩阵类型

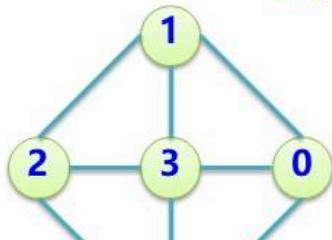


- 每个单链表上添加一个表头结点（表示顶点信息）。并将所有表头结点构成一个数组，下标为 $v$ 的元素表示顶点 $v$ 的表头结点。



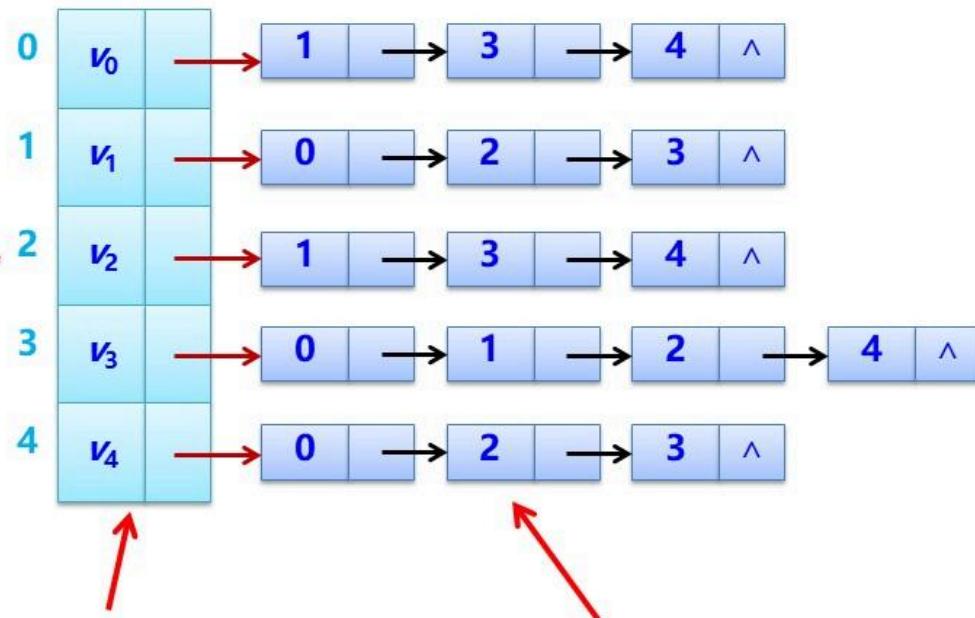


图的邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。



找顶点2的边

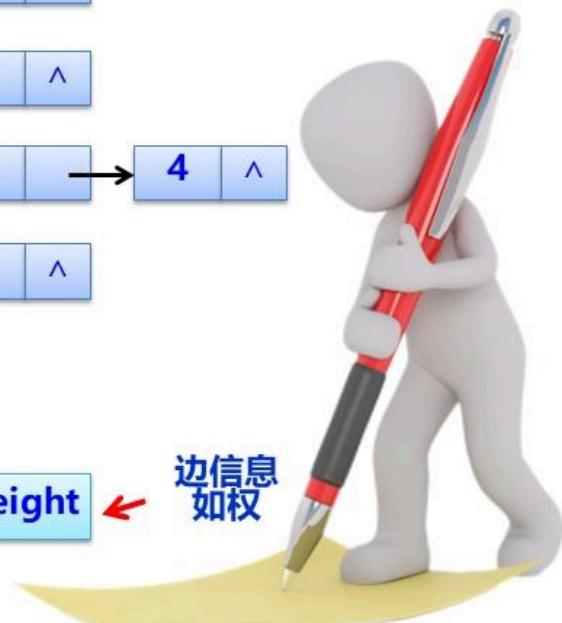
两类结点



头结点

边结点

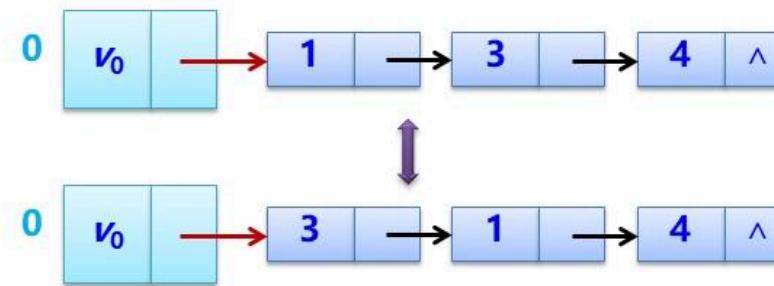
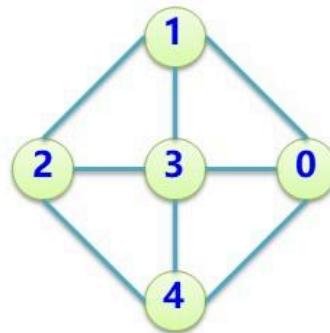
边信息  
如权





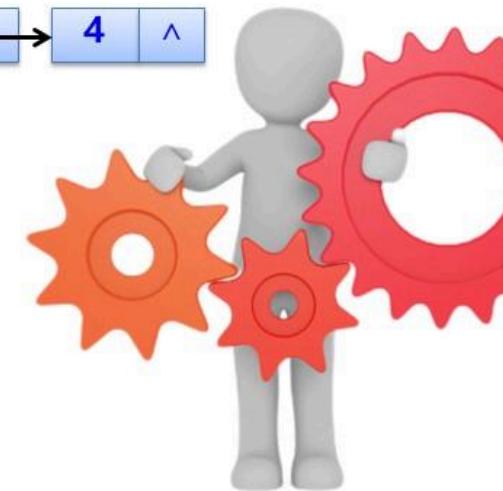
## 邻接表的特点如下：

- 邻接表表示不唯一。



- 特别适合于稀疏图存储。

邻接表的存储空间为  $O(n+e)$





图的邻接表存储类型定义如下：

```
typedef struct ANode
{ int adjvex;           //该边的终点编号
  struct ANode *nextarc; //指向第一条边的指针
  InfoType weight;       //该边的权值等信息
} ArcNode;

typedef struct Vnode
{ Vertex data;          //顶点信息
  ArcNode *firstarc;    //指向第一条边
} VNode;

typedef struct
{ VNode adjlist[MAXV];   //邻接表
  int n, e;              //图中顶点数n和边数e
} AdjGraph;
```

声明边结点类型

声明邻接表头结点类型

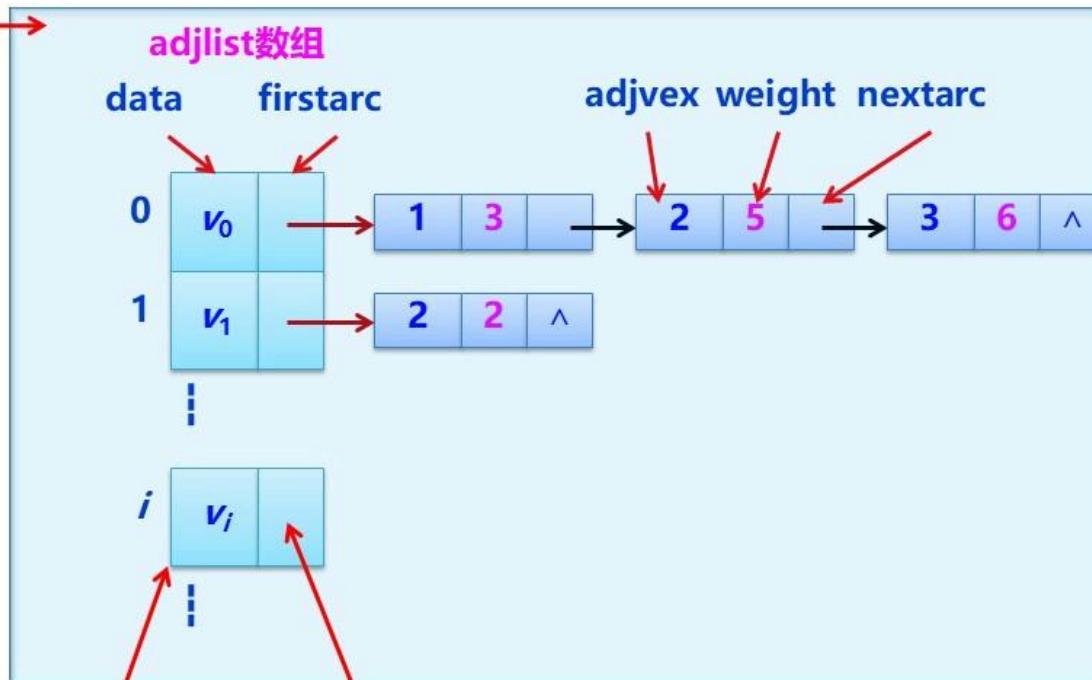
声明图邻接表类型





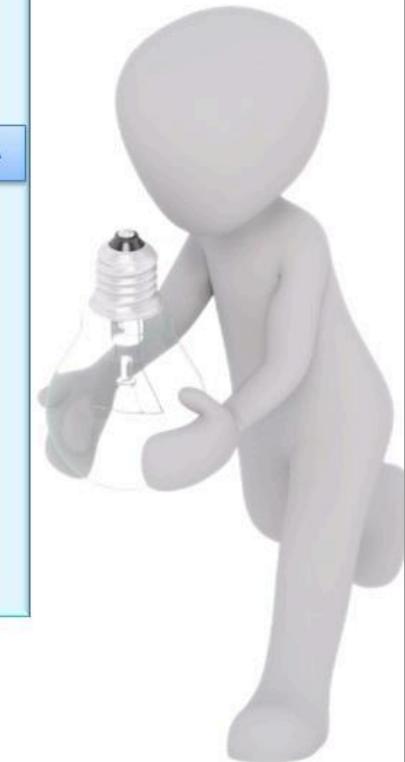
## 一个邻接表通常用指针引用：

G →



引用头结点：G->adjlist[ $\lambda$ ]

引用头结点的指针域：G->adjlist[ $\lambda$ ].firstarc





### 8.3.1 图的遍历的概念

- 从给定图中任意指定的顶点（称为初始点）出发，按照某种搜索方法沿着图的边访问图中的所有顶点，使每个顶点仅被访问一次，这个过程称为图的遍历。
- 图的遍历得到的顶点序列称为图遍历序列。



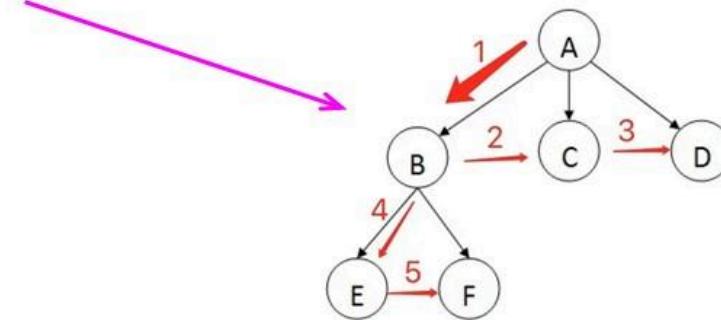
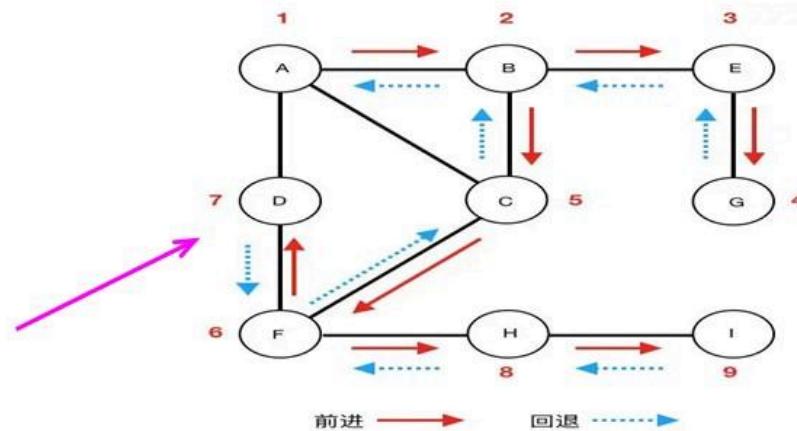
根据搜索方法的不同，图的遍历方法有两种：



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



- 深度优先遍历 (DFS)。
- 广度优先遍历 (BFS)。





## 深度优先遍历过程：

- (1) 从图中某个初始顶点  $v$  出发，首先访问初始顶点  $v$ 。
- (2) 选择一个与顶点  $v$  相邻且没被访问过的顶点  $w$ ，再从  $w$  出发进行深度优先搜索，直到图中与当前顶点  $v$  邻接的所有顶点都被访问过为止。





- 深度优先遍历的过程体现出后进先出的特点：用栈或递归方式实现。

- 如何确定一个顶点是否访问过？设置一个**visited[] 全局数组**，**visited[i]=0** 表示顶点*没有*访问过；**visited[i]=1**表示顶点*已经*访问过。





图的邻接表存储类型定义如下：

```
typedef struct ANode
{ int adjvex;           //该边的终点编号
  struct ANode *nextarc; //指向下一条边的指针
  InfoType weight;      //该边的权值等信息
} ArcNode;

typedef struct Vnode
{ Vertex data;          //顶点信息
  ArcNode *firstarc;    //指向第一条边
} VNode;

typedef struct
{ VNode adjlist[MAXV];   //邻接表
  int n, e;              //图中顶点数n和边数e
} AdjGraph;
```

声明边结点类型

声明邻接表头结点类型

声明图邻接表类型



## 采用邻接表的DFS算法：



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



```
void DFS(AdjGraph *G, int v)
{ ArcNode *p; int w;
    visited[v]=1;           //置已访问标记
    printf("%d ", v);      //输出被访问顶点的编号
    p=G->adjlist[v].firstarc; //p指向顶点v的第一条边的边头结点

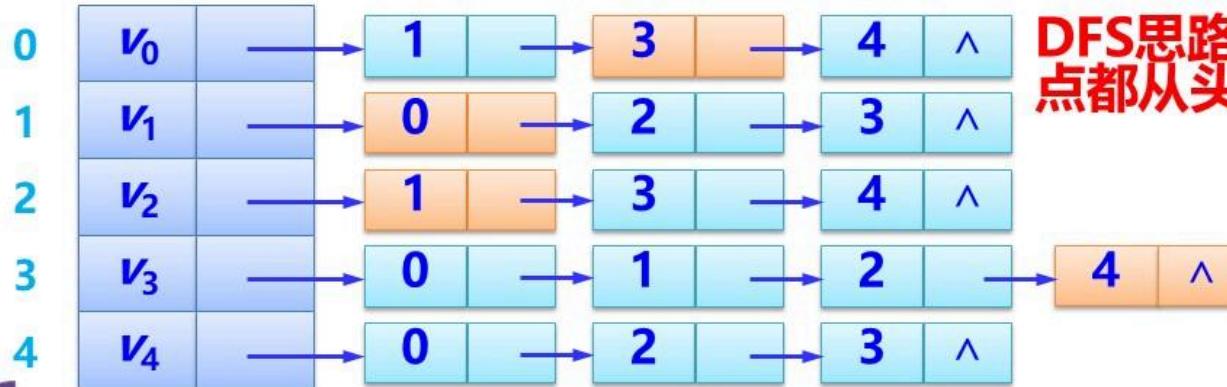
    while (p!=NULL)
    { w=p->adjvex;
        if (visited[w]==0)
            DFS(G, w);          //若w顶点未访问，递归访问它
        p=p->nextarc;         //p指向顶点v的下一条边的边头结点
    }
}
```

该算法的时间复杂度为 $O(n+e)$ 。

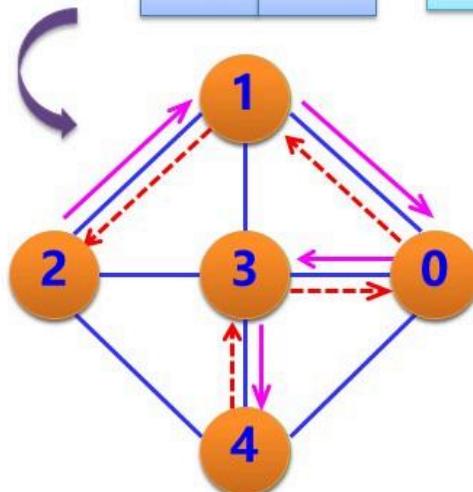
## 深度优先遍历过程演示



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



DFS思路：递归思想，每个顶点都从头结点数组重新“开始”



$v=2$ 的DFS序列：

2 1 0 3 4

遍历过程结束

DFS思路：距离初始顶点越远越优先访问！



示例

无向图  $G = (V, E)$ ，其中

$$V = \{a, b, c, d, e, f\},$$

$$E = \{(a, b), (a, e), (a, c), (b, e), (c, f), (f, d), (e, d)\}$$

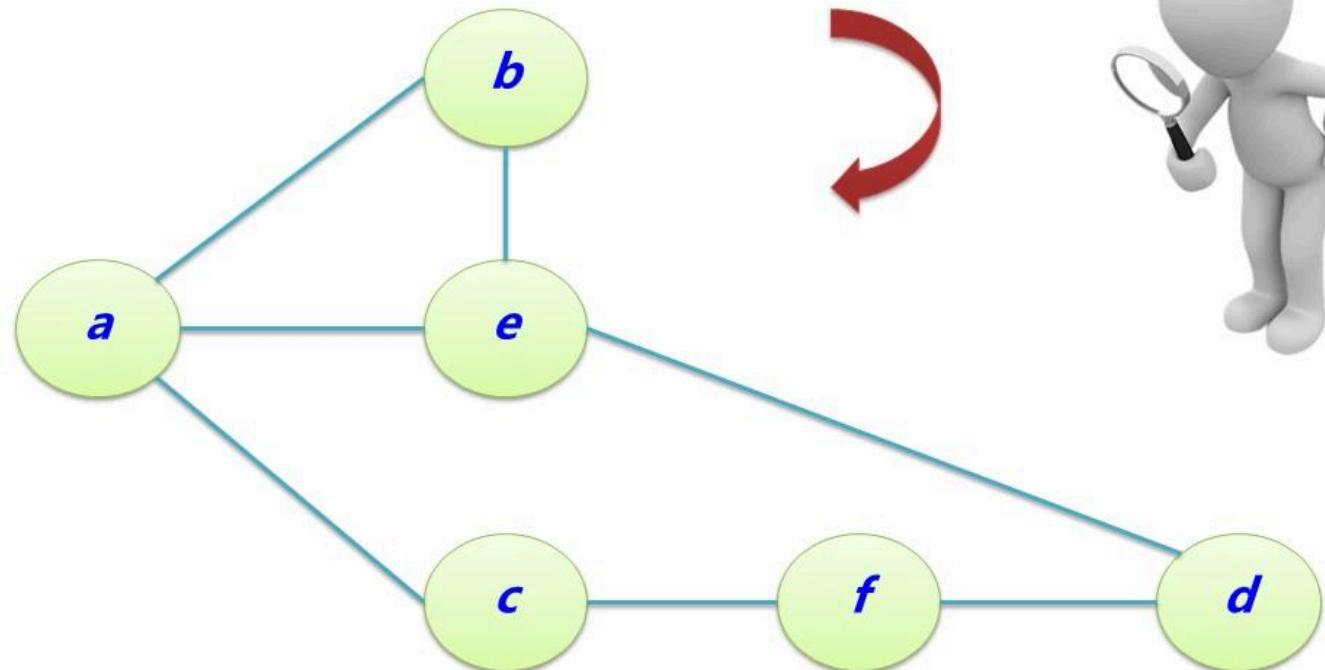
对该图进行深度优先排序，得到的顶点序列正确的是（ ）。

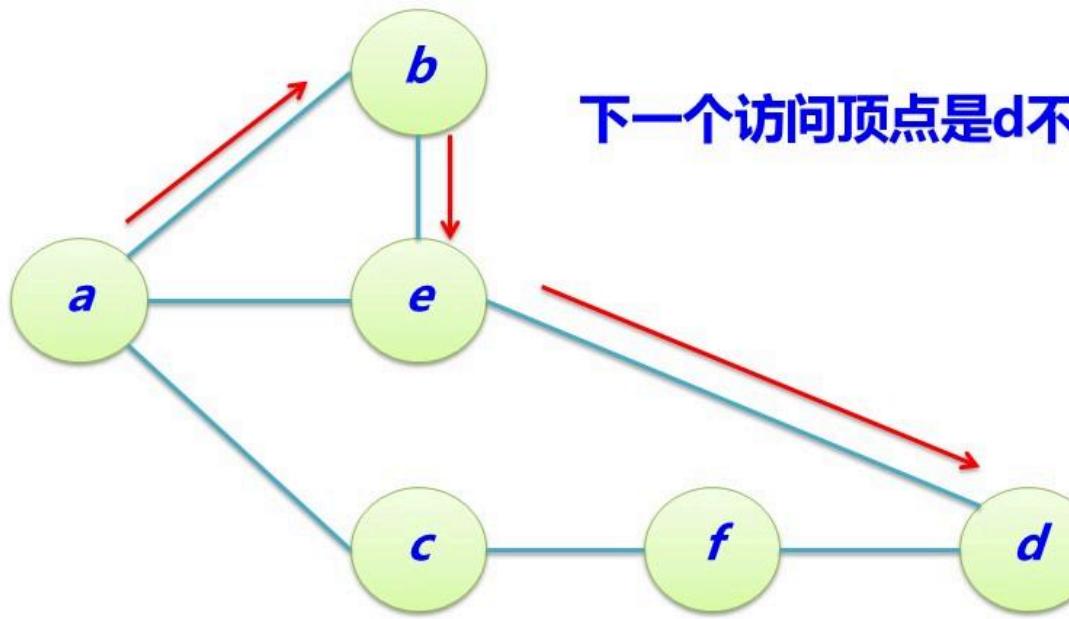
- A. a, b, e, c, d, f
- B. a, c, f, e, b, d
- C. a, e, b, c, f, d
- D. a, e, d, f, c, b



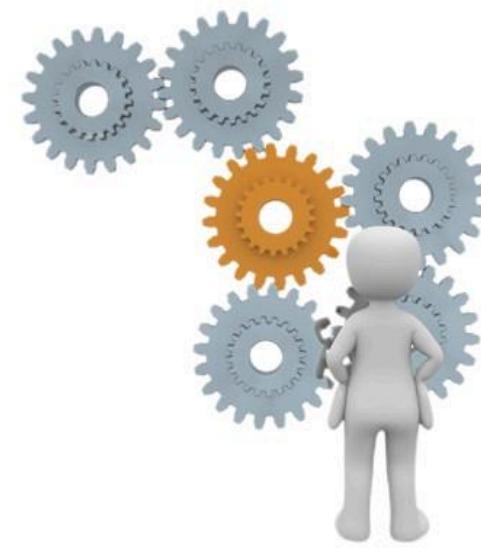


$$E = \{(a,b), (a,e), (a,c), (b,e), (c,f), (f,d), (e,d)\}$$





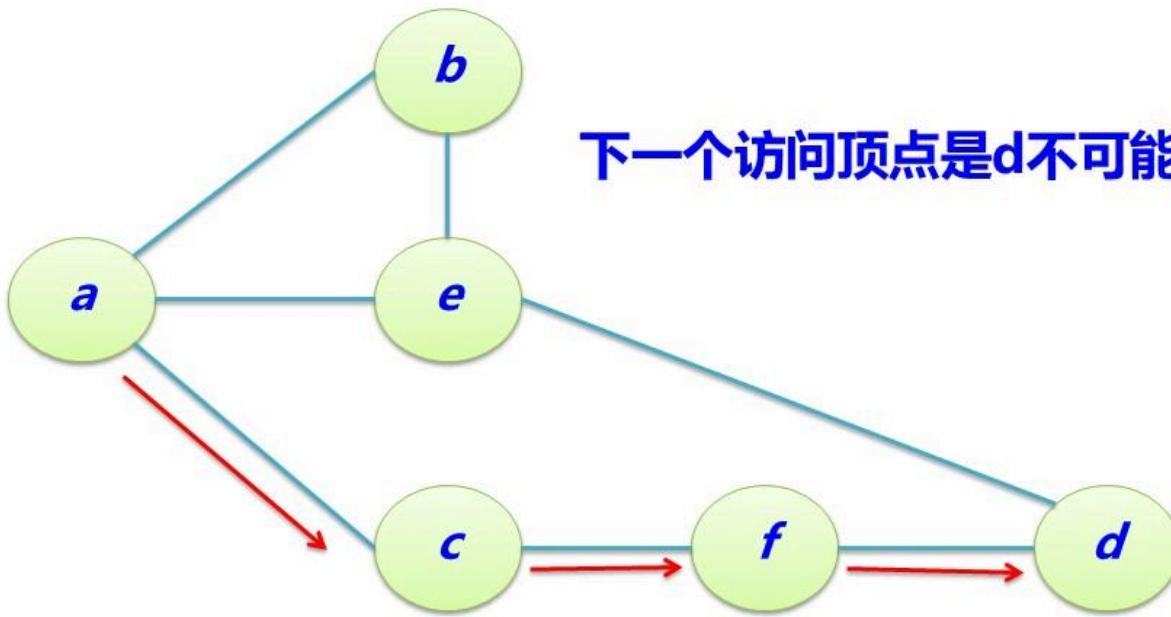
下一个访问顶点是d不可能是c



A. a, b, e, c, d, f

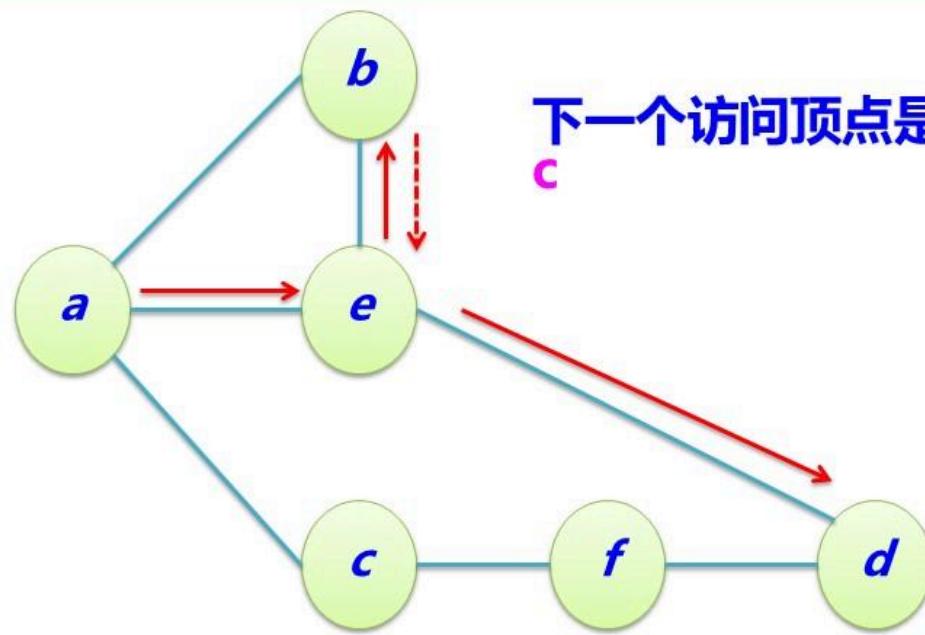


下一个访问顶点是d不可能是e



B. a, c, f, e, b, d

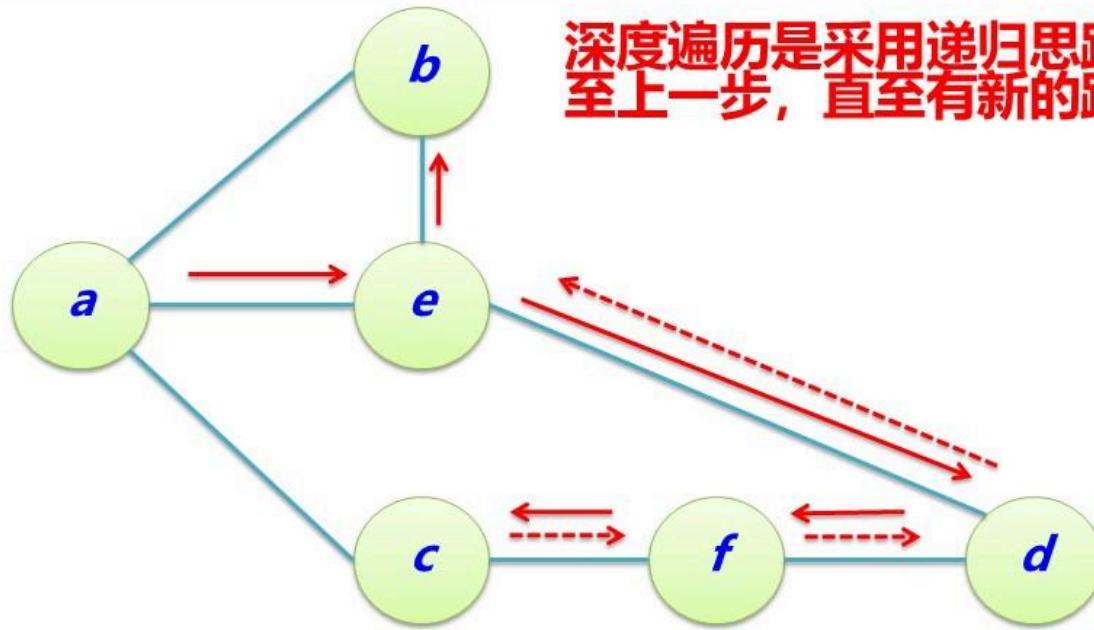




C. a, e, b, c, f, d



深度遍历是采用递归思路，遇路不通时，回溯至上一步，直至有新的路可走，否则就结束



D. a, e, d, f, c, b

答案为D





#### 广度优先遍历的过程：

- (1) 访问初始点  $v$ ，接着访问  $v$  的所有未被访问过的邻接点  $v_1, v_2, \dots, v_t$ 。
- (2) 按照  $v_1, v_2, \dots, v_t$  的次序，访问每一个顶点的所有未被访问过的邻接点。
- (3) 依次类推，直到图中所有和初始点  $v$  有路径相通的顶点都被访问过为止。



- 广度优先搜索遍历体现**先进先出**的特点，用**队列**实现。

(1) 访问初始点  $v$ ，接着访问  $v$  的所有未被访问过的邻接点  $v_1, v_2, \dots, v_t$ 。  
(2) 按照  $v_1, v_2, \dots, v_t$  的次序，访问每一个顶点的所有未被访问过的邻接点。  
(3) 依次类推，直到图中所有和初始点  $v$  有路径相通的顶点都被访问过为止。

- 如何确定一个顶点是否访问过？设置一个 **visited[]** 数组， $\text{visited}[i]=0$  表示顶点  $i$  没有访问； $\text{visited}[i]=1$  表示顶点  $i$  已经访问过。





## 采用邻接表的BFS算法：



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



```
void BFS(AdjGraph *G, int v)
{ int w, i;
ArcNode *p;
SqQueue *qu;           //定义环形队列指针
InitQueue(qu);         //初始化队列
int visited[MAXV];     //定义顶点访问标记数组
for (i=0;i<G->n;i++)
    visited[i]=0;       //访问标记数组初始化
    printf("%2d", v);   //输出被访问顶点的编号
    visited[v]=1;        //置已访问标记
    enQueue(qu, v);
```

思考题：为什么visited数组不需要设置为全局变量？



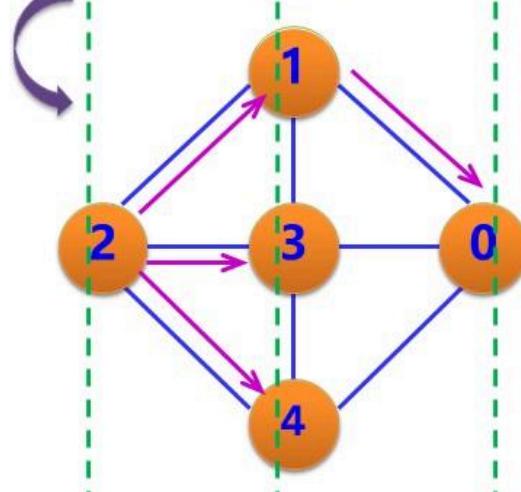
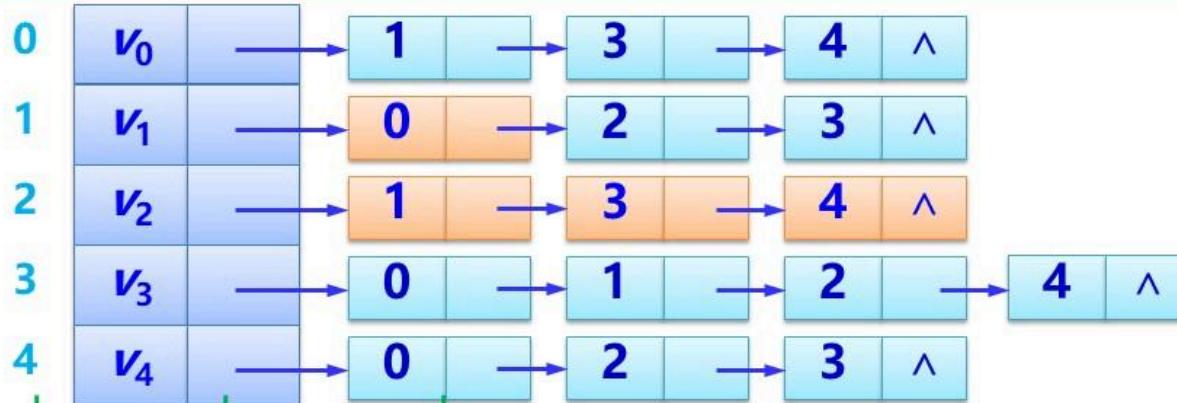
```
while (!QueueEmpty(qu))          //队不空循环
{ deQueue(qu, w);               //出队一个顶点w
  p=G->adjlist[w].firstarc;    //指向w的第一个邻接点
  while (p!=NULL)               //查找w的所有邻接点
  { if (visited[p->adjvex]==0)  //若当前邻接点未被访问
    { printf("%2d", p->adjvex); //访问该邻接点
      visited[p->adjvex]=1;     //置已访问标记
      enqueue(qu, p->adjvex);   //该顶点进队
    }
    p=p->nextarc;             //找下一个邻接点
  }
}
printf("\n");
```

该算法的时间复杂度为 $O(n+e)$ 。

## 广度优先遍历过程演示



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$v=2$ 的BFS序列:

2 1 3 4 0

遍历过程结束

BFS思路: 距离初始顶点越近越优先访问!



示例

对下图所示的无向图，从顶点0开始进行广度优先遍历，不可能得到顶点访问序列是（ ）。

A.0231645

B.0163254

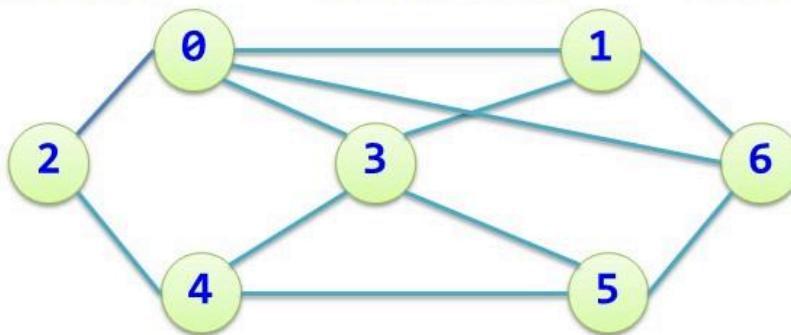
C.0613254

D.0243165

D.0243165

顶点0到2、3、1、6的距离均为1。一定是0，后面为2316的排列

答案为D。





## 找起点 $\Rightarrow$ 终点的一条路径

- 看到一条可以走（没有走过）路径就直接走下去，没有道路时回退  $\Rightarrow$  DFS
- 考虑每一条可以走（没有走过）的路径，尝试走每条道路  $\Rightarrow$  BFS

记下路径的所有路口位置：确定一条路径

DFS和BFS试探的所有路口数是差不多的！



### 概念

- 对于带权连通图G（每条边上的权均为大于零的实数），可能有多棵不同生成树。
- 每棵生成树的所有边的权值之和可能不同。
- 其中权值之和最小的生成树称为图的最小生成树。



### 8.4.3 普里姆算法



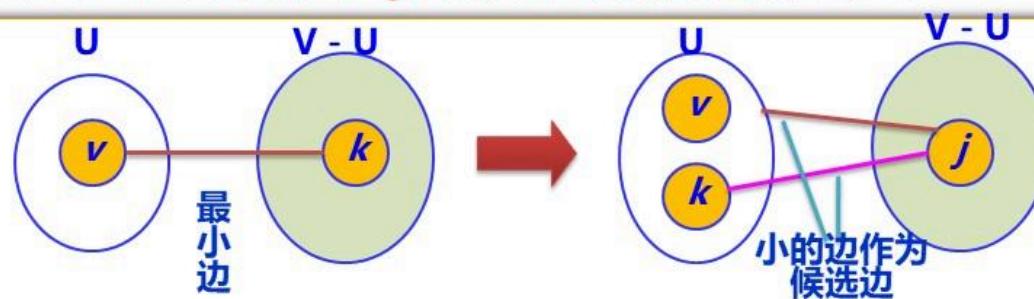
清华大学出版社  
TSINGHUA UNIVERSITY PRESS

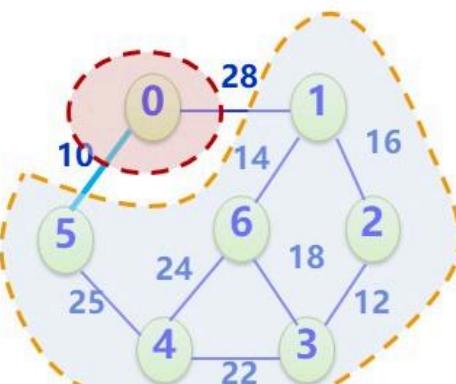


普里姆 (Prim) 算法是一种构造性算法，  
用于构造最小生成树。过程如下：

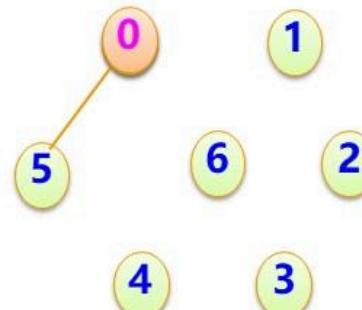


- (1) 初始化  $U = \{v\}$ 。 $v$ 到其他顶点的所有边为候选边；
- (2) 重复以下步骤  $n-1$  次，使得其他  $n-1$  个顶点被加入到  $U$  中：
  - ①从候选边中挑选权值最小的边输出，设该边在  $V-U$  中的顶点是  $k$ ，将  $k$  加入  $U$  中；
  - ②考察当前  $V-U$  中的所有顶点  $j$ ，修改候选边：若  $(j, k)$  的权值小于原来和顶点  $k$  关联的候选边，则用  $(k, j)$  取代后者作为候选边。





图G



$U=\{0\}$



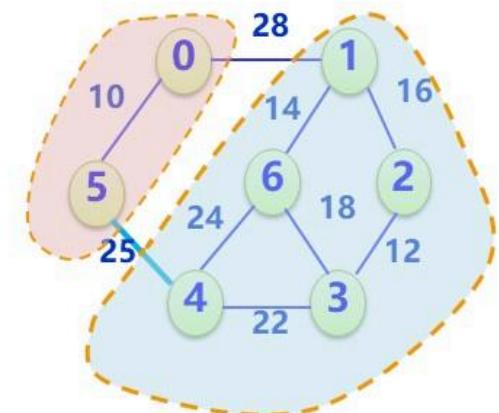
普里姆算法求解最小生成树的过程



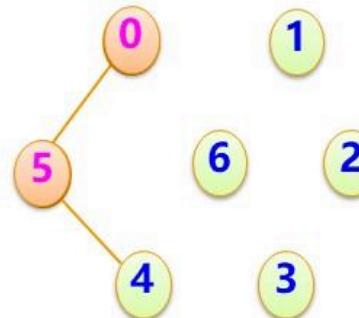
## Prim算法示例演示 (起点0)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{0, 5\}$



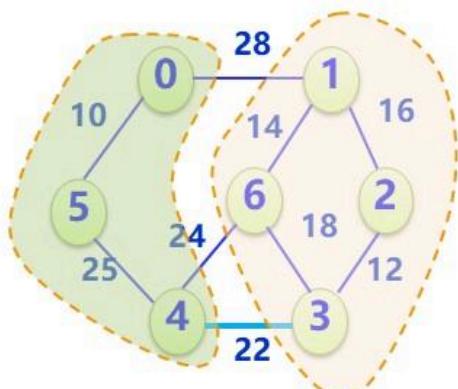
普里姆算法求解最小生成树的过程



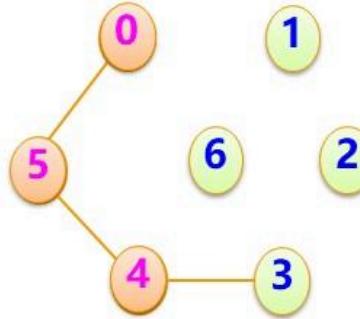
## Prim算法示例演示 (起点0)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{0, 5, 4\}$



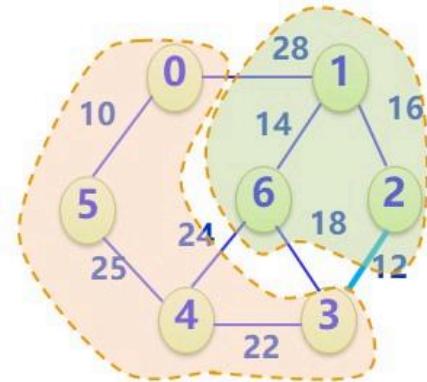
普里姆算法求解最小生成树的过程



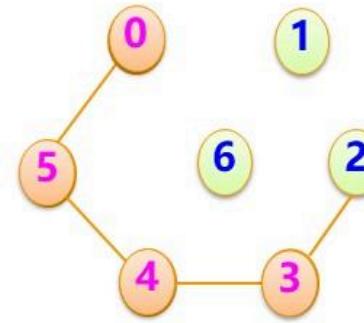
## Prim算法示例演示 (起点0)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{0, 5, 4, 3\}$



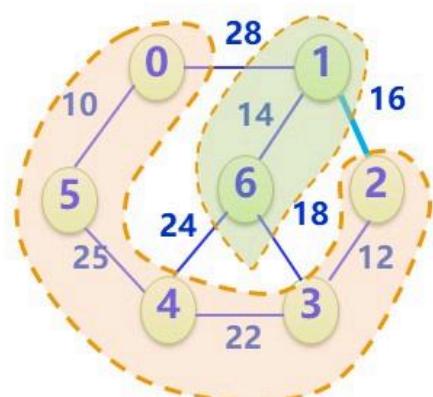
普里姆算法求解最小生成树的过程



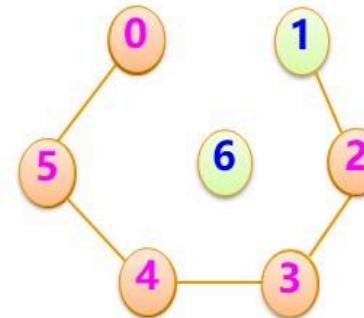
## Prim算法示例演示 (起点0)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{0, 5, 4, 3, 2\}$



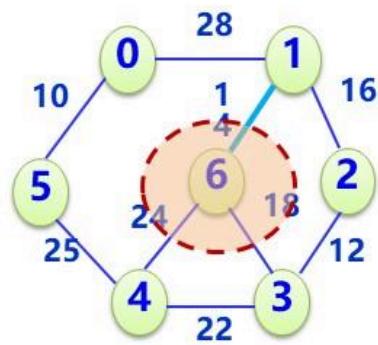
普里姆算法求解最小生成树的过程



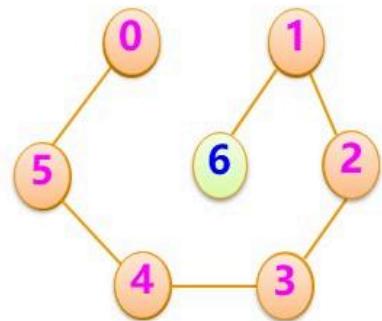
## Prim算法示例演示 (起点0)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{0, 5, 4, 3, 2, 1, 6\}$

最小生成树



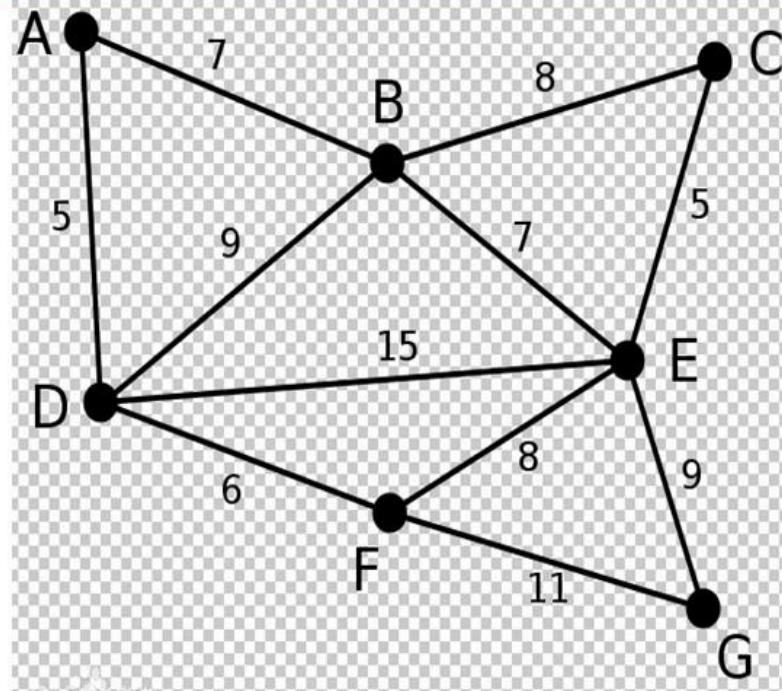
普里姆算法求解最小生成树的过程



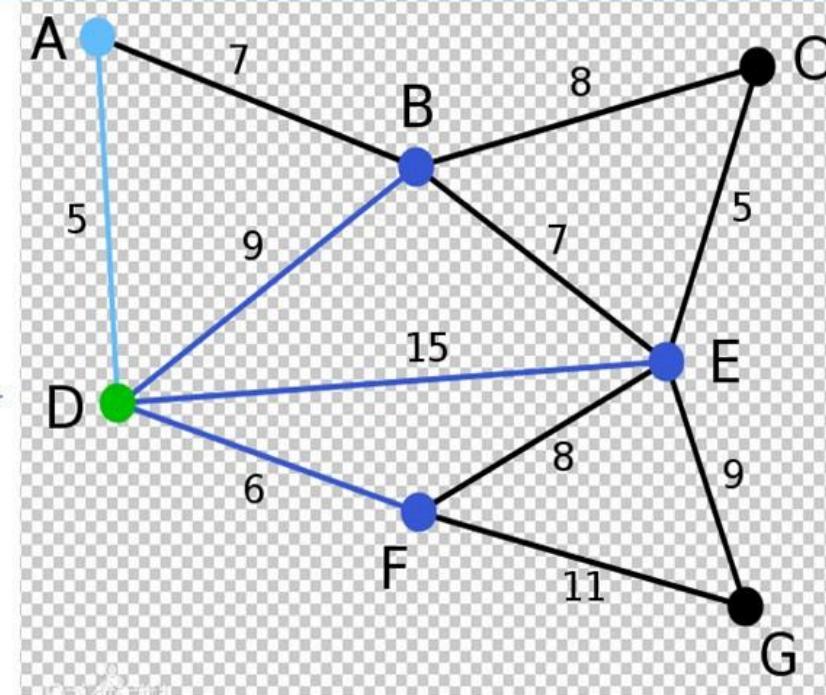
## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图G



$U=\{D\}$

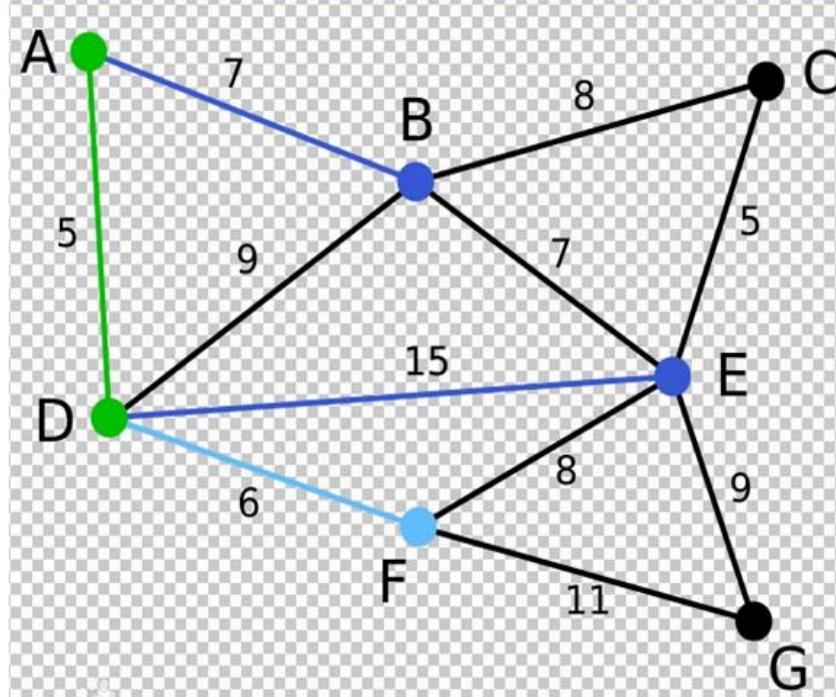
普里姆算法求解最小生成树的过程



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$$U=\{D, A\}$$

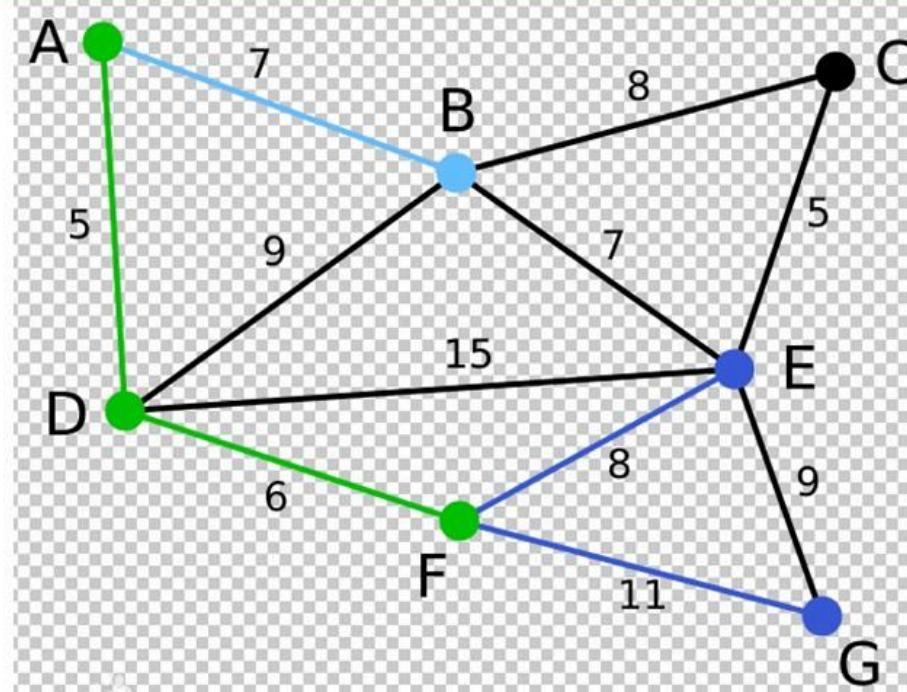
普里姆算法求解最小生成树的过程



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$$U=\{D, A, F\}$$

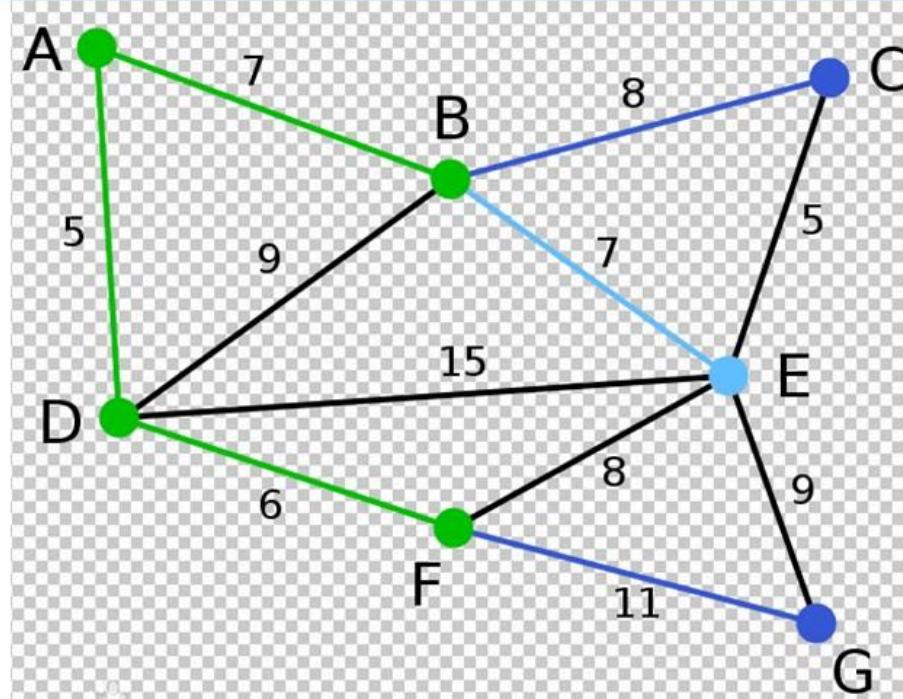
普里姆算法求解最小生成树的过程



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$$U=\{D, A, F, B\}$$

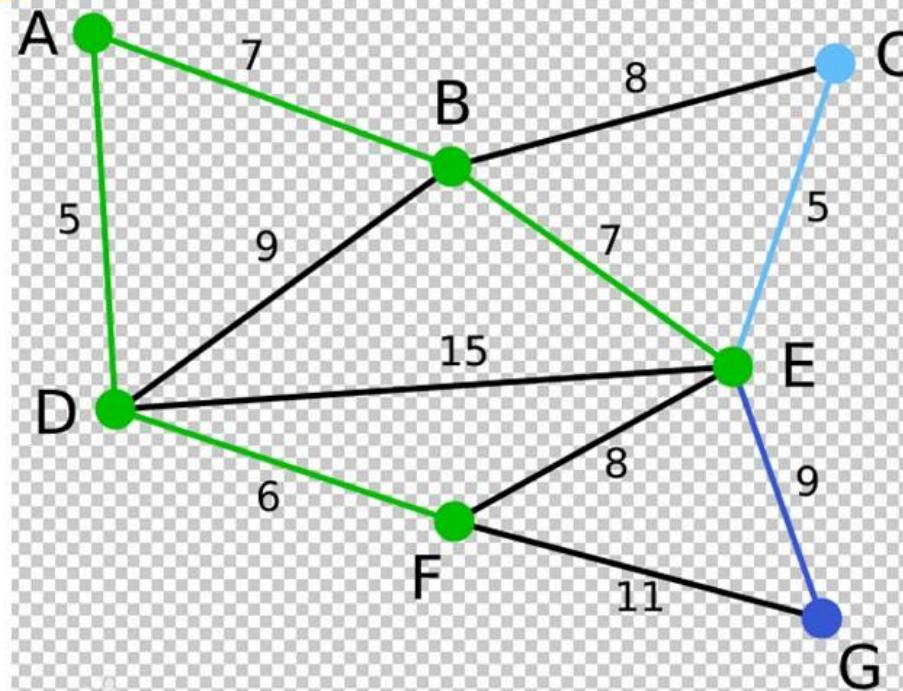
普里姆算法求解最小生成树的过程



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



$$U=\{D, A, F, B, E\}$$

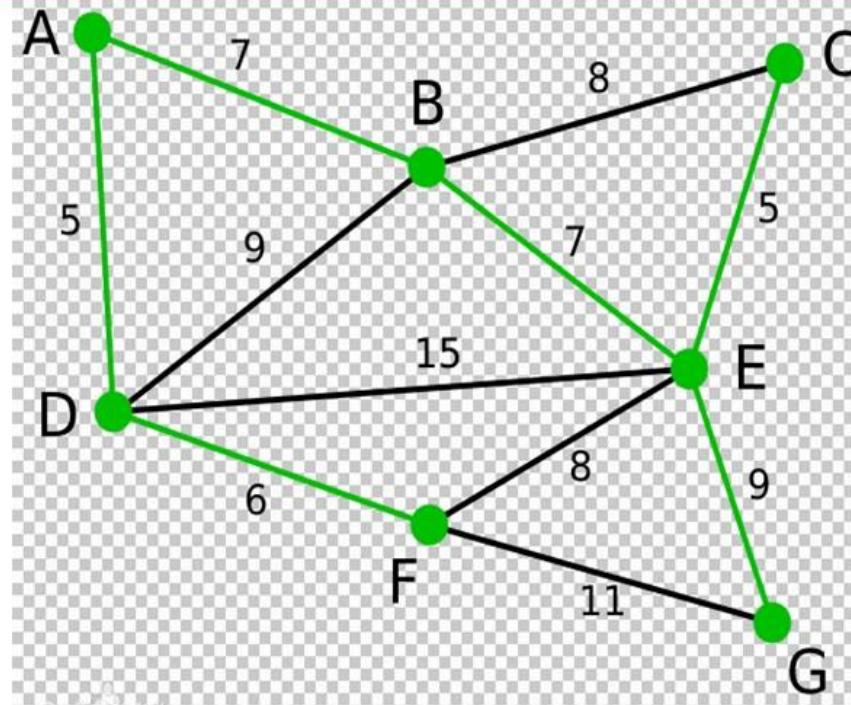
普里姆算法求解最小生成树的过程



## Prim算法示例演示 (起点D)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



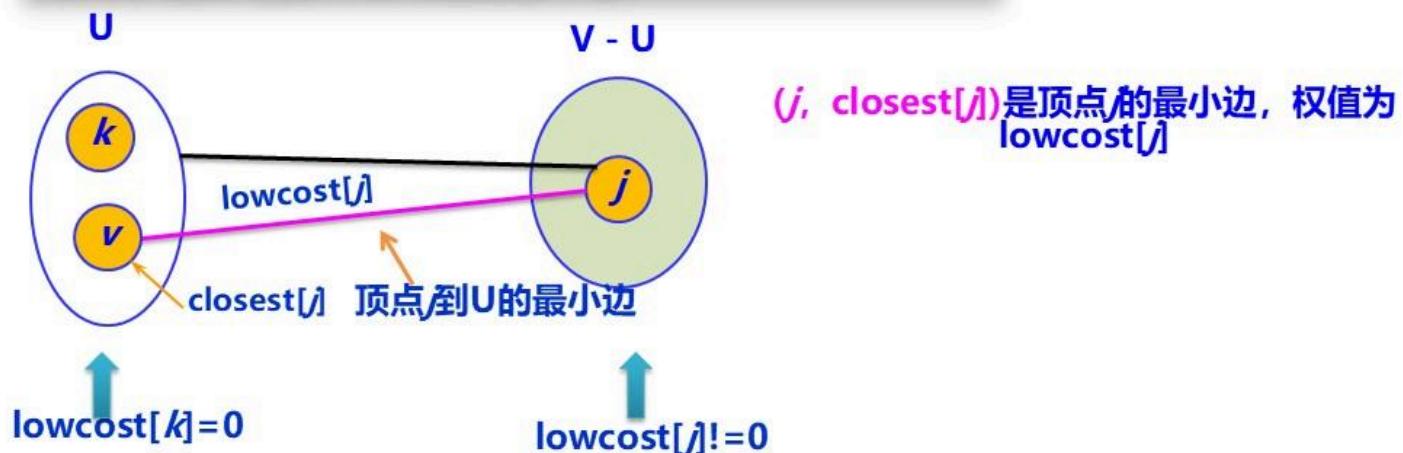
$$U=\{D, A, F, B, E, C, G\}$$

普里姆算法求解最小生成树的过程



如何求U、V-U两个顶点集之间的最小边？（只求一条）

只考虑V-U中顶点j到U顶点集的最小边（无向图），比较来找最小边  
如何存储顶点j到U顶点集的最小边？



一个顶点属于哪个集合？

图采用哪种存储结构更合适？ 邻接矩阵



## 8.4.4 克鲁斯卡尔算法



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 克鲁斯卡尔 (Kruskal) 算法过程：构造最小生成树 (U,TE)

- (1) 置U的初值等于V (即包含有G中的全部顶点)， TE的初值为空集 (即图T中每一个顶点都构成一个连通分量)。
- (2) 将图G中的边按权值从小到大的顺序依次选取：
- ① 若选取的边未使生成树T形成回路，则加入TE；
  - ② 否则舍弃，直到TE中包含 $(n-1)$ 条边为止。

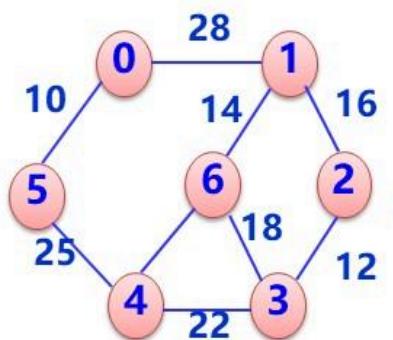




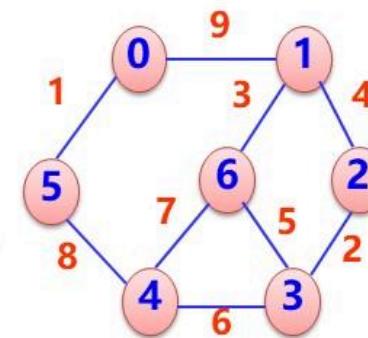
## Kruskal算法示例的演示



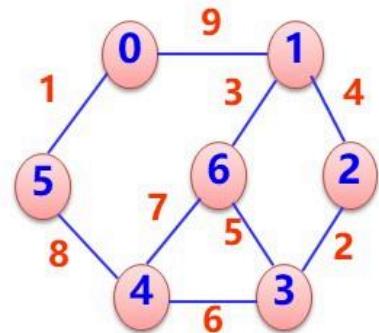
清华大学出版社  
TSINGHUA UNIVERSITY PRESS



按边大小递增排序

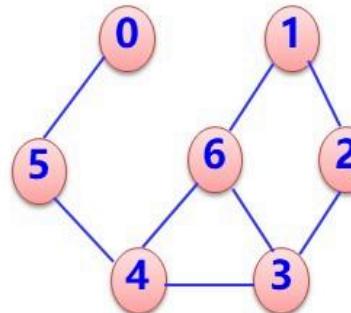


克鲁斯卡尔算法求解最小生成树的过程



操作  
取8号边

选取了 $n-1$ 条边



最小生成树

克鲁斯卡尔算法求解最小生成树的过程



## 算法设计（解决3个问题）：



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



图采用哪种存储结构更合适？

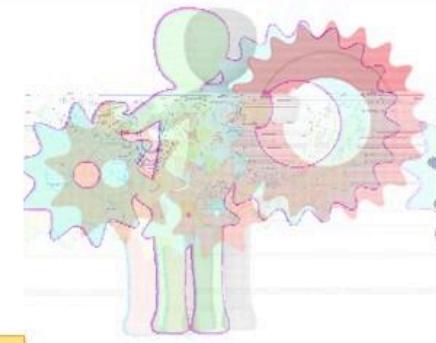
邻接矩阵

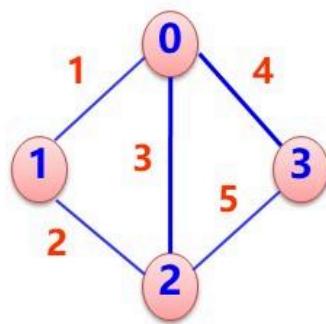
边的排序问题？

这里采用直接插入排序算法

如何解决加入一条边后是否出现回路？

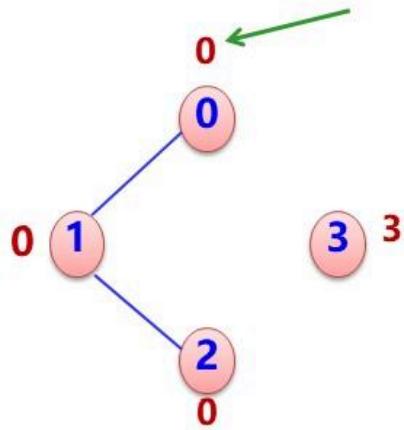
采用连通分量编号或顶点集合编号





操作  
取3号边

vset[0]: 连通分量编号



3号边的两个顶点的vset

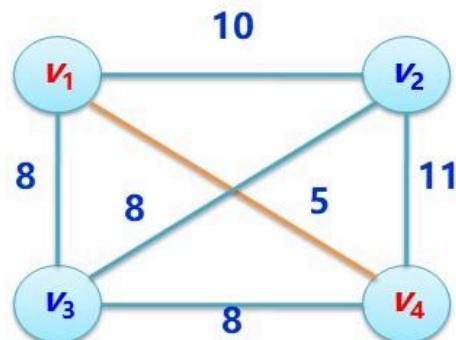
值相同，不能添加！



**【例8.12】**求下面带权图的最小（代价）生成树时，可能是克鲁斯卡（kruskal）算法第二次选中但不是普里姆（Prim）算法（从 $v_4$ 开始）第2次选中的边是（ ）。

- A.  $(v_1, v_3)$    B.  $(v_1, v_4)$    C.  $(v_2, v_3)$    D.  $(v_3, v_4)$

注：2015年全国考研题



**kruskal:**

1:  $(v_1, v_4)$ , 2:  $(v_1, v_3)$  或  $(v_3, v_4)$   
或  $(v_2, v_3)$  , ...

**Prim (从 $v_4$ 开始) :**

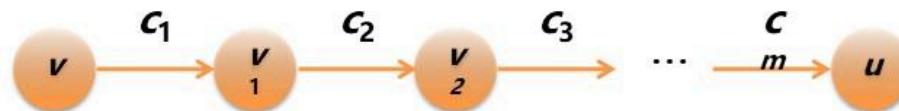
1:  $(v_1, v_4)$ , 2:  $(v_1, v_3)$  或  $(v_3, v_4)$  ,  
不可能是  $(v_2, v_3)$

答案为C



## 8.5.1 路径的概念

考虑**带权有向图**，把一条路径（仅仅考虑简单路径）上所经边的权值之和定义为该路径的**路径长度**或称**带权路径长度**。



$$\text{路径长度} = c_1 + c_2 + \dots + c_m$$

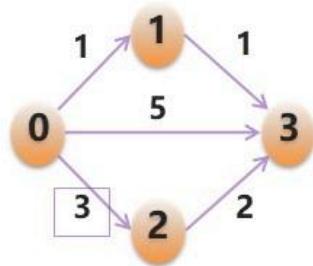
$$\text{路径: } (v, v_1, v_2, \dots, u)$$

从源点到终点可能不止一条路径，把路径长度最短的那条路径称为**最短路径**。



### 8.5.1 路径的概念

#### 带权有向图



顶点0到3：

- **最短路径： $0 \rightarrow 1 \rightarrow 3$**
- **最短路径长度：2**

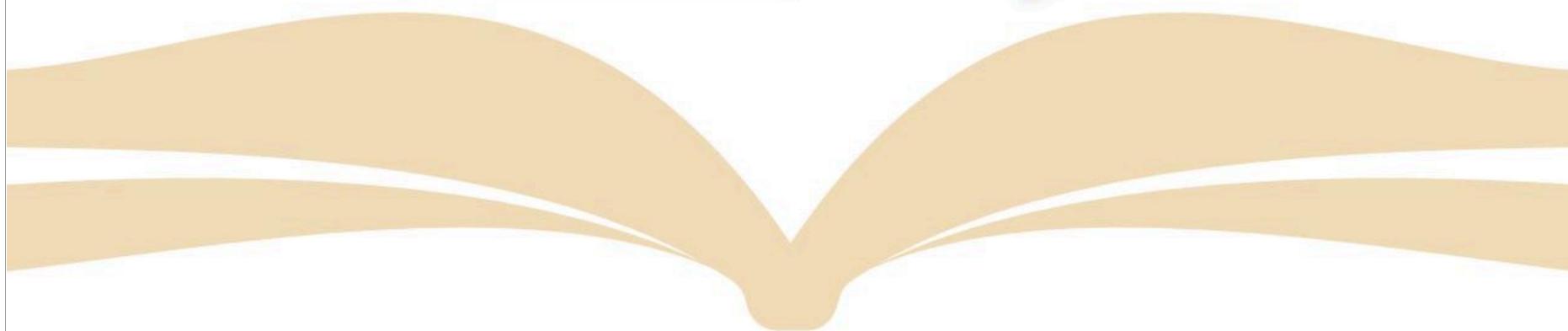
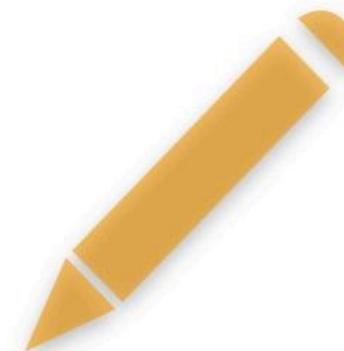
很多情况下，两个顶点的最短路径不一定唯一，但最短路径长度一定是唯一的。



### 8.5.1 路径的概念

如何求图中的最短路径？

- Dijkstra算法
- Floyd算法





### 8.5.2 从一个顶点到其余各顶点的最短路径



**问题描述：**给定一个带权有向图G与源点 $v$ ，求从 $v$ 到G中其他顶点的最短路径，并限定各边上的权值大于或等于0。

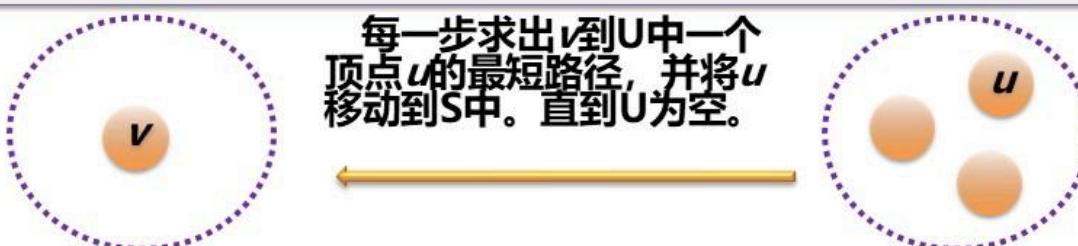
**单源最短路径问题：  
Dijkstra算法**



## 狄克斯特拉 (Dijkstra) 求解思路

设 $G=(V, E)$ 是一个带权有向图，把图中顶点集合 $V$

- 第1组为已求出最短路径的顶点集合（用 $S$ 表示，初始时 $S$ 中只有一个源点，以后每求得一条最短路径 $v, \dots, u$ ，就将 $u$ 加入到集合 $S$ 中，直到全部顶点都加入到 $S$ 中，算法就结束了）。
- 第2组为其余未求出最短路径的顶点集合（用 $U$ 表示）。

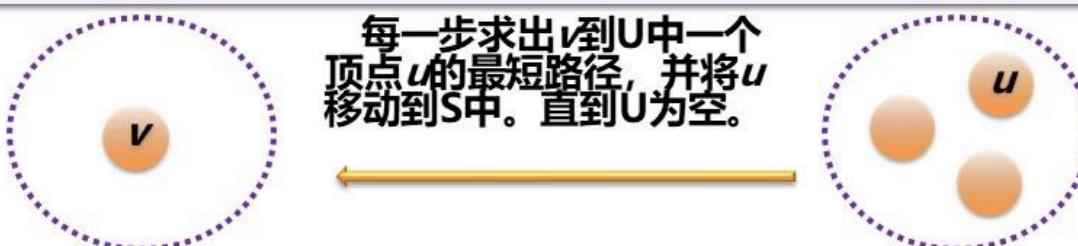




## 狄克斯特拉 (Dijkstra) 求解思路

设 $G=(V, E)$ 是一个带权有向图，把图中顶点集合 $V$

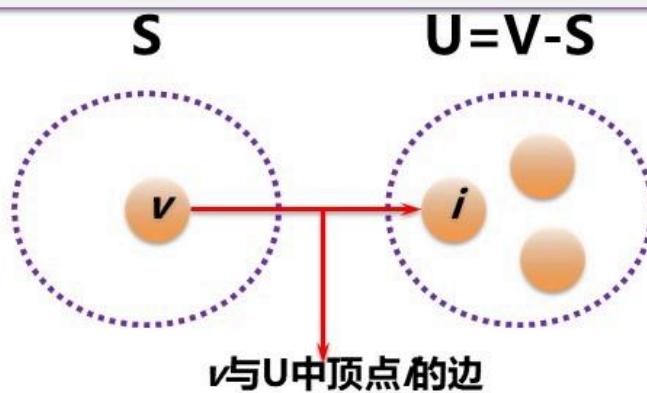
- 第1组为已求出最短路径的顶点集合（用 $S$ 表示，初始时 $S$ 中只有一个源点，以后每求得一条最短路径 $v, \dots, u$ ，就将 $u$ 加入到集合 $S$ 中，直到全部顶点都加入到 $S$ 中，算法就结束了）。
- 第2组为其余未求出最短路径的顶点集合（用 $U$ 表示）。





## 狄克斯特拉算法的过程

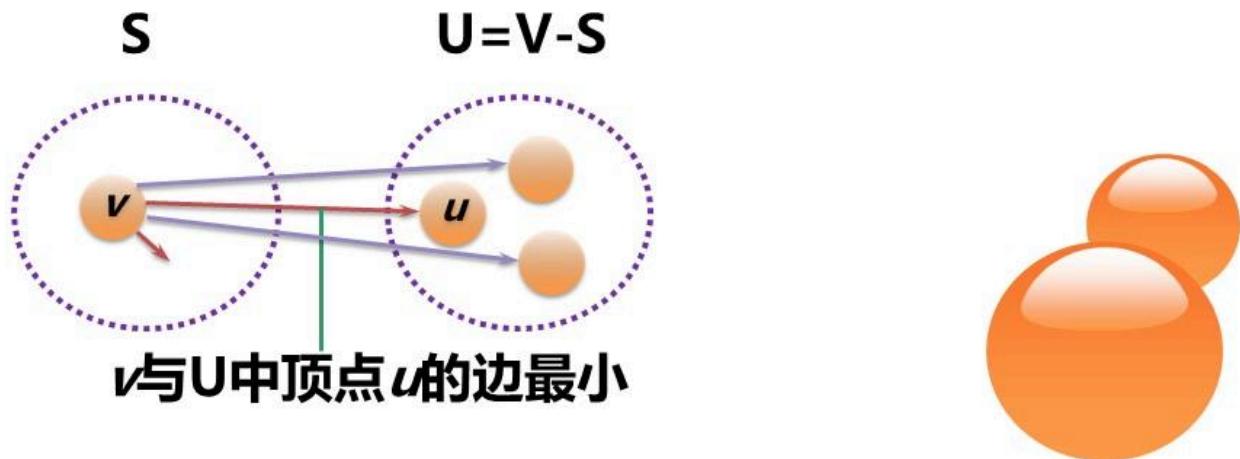
(1) **初始化:**  $S$ 只包含源点即 $S=\{v\}$ ,  $v$ 的最短路径为0。  $U$ 包含除 $v$ 外的其他顶点,  $U$ 中顶点距离为边上的权值 (若 $v$ 与 $i$ 有边 $\langle v, i \rangle$ ) 或 $\infty$  (若 $i$ 不是 $v$ 的出边邻接点)。





## 狄克斯特拉算法的过程

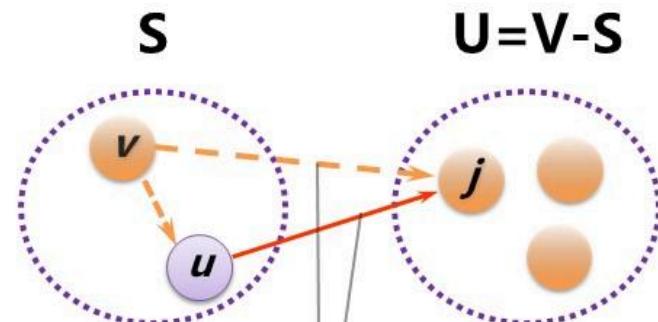
(2) 从 $U$ 中选取一个距离 $v$ 最小的顶点 $u$ , 把 $u$ 加入 $S$ 中 (该选定的距离就是  $v \rightarrow u$  的最短路径长度)。





## 狄克斯特拉算法的过程

(3) 以 $u$ 为新考虑的中间点，修改 $U$ 中各顶点 $j$  的最短路径长度：  
若从源点 $v \rightarrow j$  ( $j \in U$ ) 的最短路径长度 (经过顶点 $u$ ) 比原来最短路径长度 (不经过顶点 $u$ ) 短，则修改顶点 $j$  的最短路径长度。

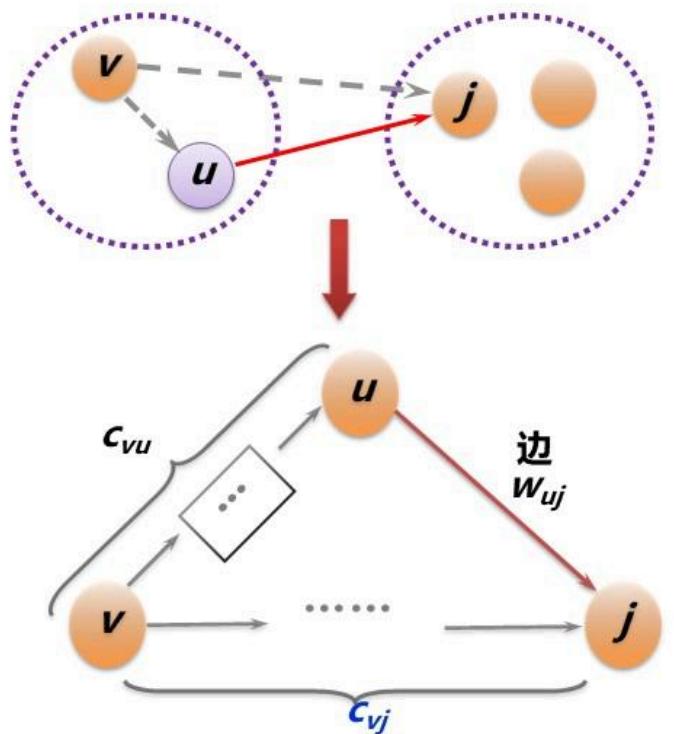


两条路径进行比较：

若经过 $u$ 的最短路径长度更短，则修正



## 修改方式



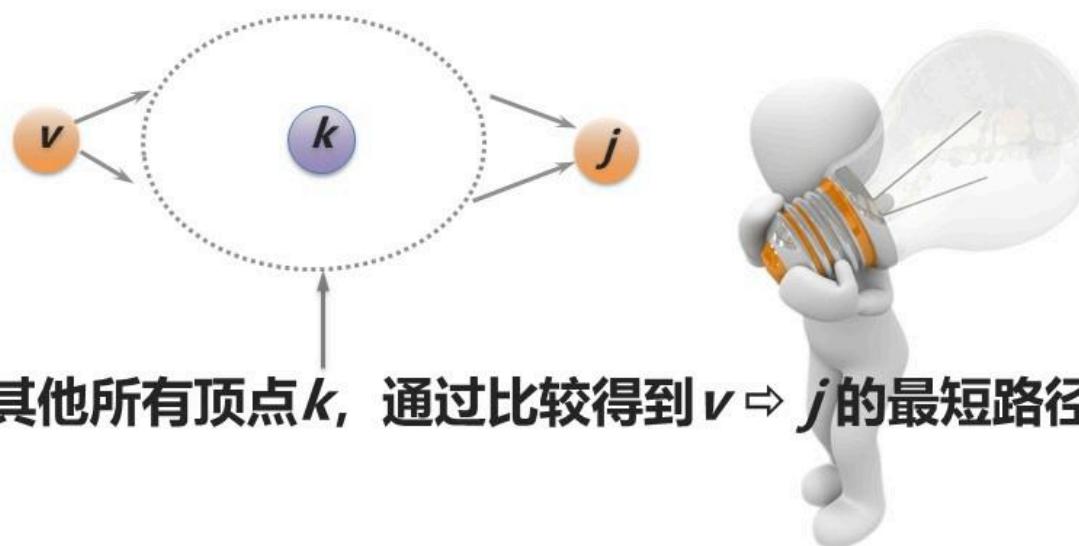
$v \Rightarrow j$  的路径:

- 不经过顶点  $u$
- 经过顶点  $u$

$$\text{顶点 } v \Rightarrow j \text{ 的最短路径长度} = \text{MIN}(c_{vu} + w_{uj}, c_{vj})$$



(4) 重复步骤 (2) 和 (3) 直到所有顶点都包含在S中。





## 算法设计（解决2个问题）

## ● 如何存放最短路径长度：

用一维数组 $dist[j]$ 存储！

源点 $v$ 默认， $dist[j]$ 表示源点 $\Rightarrow$ 顶点 $j$ 的最短路径长度。如  
 $dist[2]=12$ 表示源点 $\Rightarrow$ 顶点2的最短路径长度为12。

## ● 如何存放最短路径：

从源点到其他顶点的最短路径有 $n-1$ 条，一条最短路径用一个一维数组表示，如从顶点 $0 \Rightarrow 5$ 的最短路径为 $0, 2, 3, 5$ ，表示为

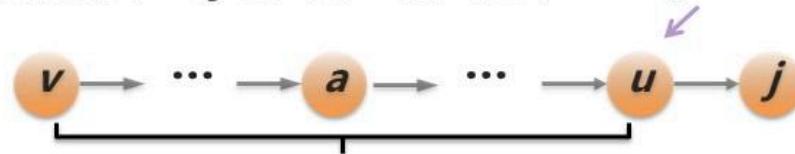
$path[5]=\{0,2,3,5\}$ 。

所有 $n-1$ 条最短路径可以用二维数组 $path[][]$ 存储？



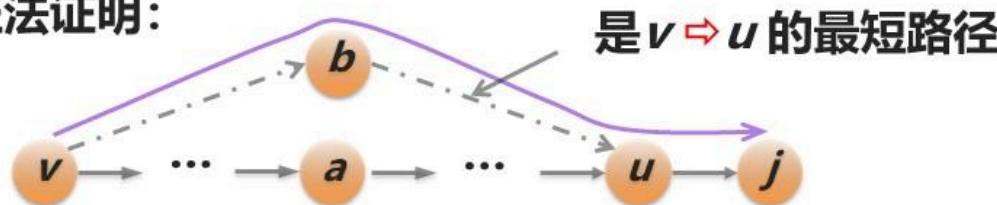
改进的方法是采用一维数组path来保存：

若从源点 $v \Rightarrow j$ 的最短路径如下： $v \Rightarrow j$ 最短路径中 $j$ 的前一个顶点



则一定是从源点 $v \Rightarrow u$ 的最短路径？

反证法证明：

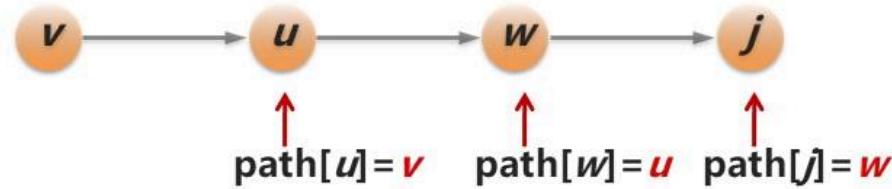


而通过 $b$ 的路径更短，则 $v \rightarrow \dots a \rightarrow \dots u \rightarrow j$ 不是最短路径

与假设矛盾，问题得到证明。



$v \rightarrow j$  的最短路径:



- 从path[j]推出的逆路径:  $j, w, u, v$
- 对应的最短路径为:  $v \rightarrow u \rightarrow w \rightarrow j$

dist[i] : 源点v至顶点i的最短路径长度

path[i] : 源点v至顶点i的最短路径上顶点i的前一个顶点的编号

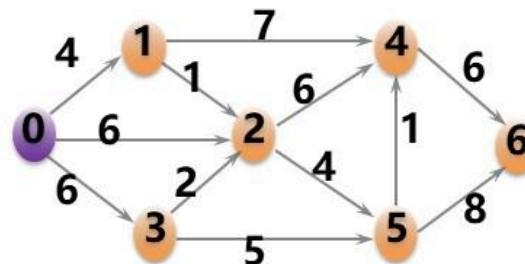


## 8.4 最短路径



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### Dijkstra算法示例演示



S

U

{0}

{1,2,3,4,5,6}

dist[]

0 1 2 3 4 5 6

path[]

0 1 2 3 4 5 6

{0, 4, 6, 6,  $\infty$ ,  $\infty$ ,  $\infty$ }

{0, 0, 0, 0, -1, -1, -1}

↓ U中dist最小的顶点: 1

{0, 1}

{2,3,4,5,6}

{0, 4, 5, 6, 11,  $\infty$ ,  $\infty$ }

{0, 0, 1, 0, 1, -1, -1}

↓ U中dist最小的顶点: 2

{0, 1, 2}

{3,4,5,6}

{0, 4, 5, 6, 11, 9,  $\infty$ }

{0, 0, 1, 0, 1, 2, -1}

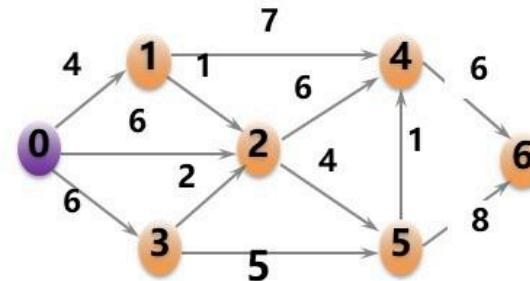
dist[i] :源点v至顶点i的最短路径长度  
path[i] :源点v至顶点i的最短路径上顶点i的前一个顶点的编号

## 8.4 最短路径



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### Dijkstra算法示例演示



$dist[i]$  : 源点v至顶点i的最短路径长度

$path[i]$  : 源点v至顶点i的最短路径上顶点i的前一个顶点的编号

S	U	dist[]	path[]
{0,1,2}	{3,4,5,6}	{0, 4, 5, 6, <u>11</u> , <u>9</u> , $\infty$ }	{0, 0, 1, 0, 1, 2, -1}
{0,1,2,3}	{4,5,6}	{0, 4, 5, 6, <u>11</u> , <u>9</u> , $\infty$ }	{0, 0, 1, 0, 1, 2, -1}
{0,1,2,3,5}	{4,6}	{0, 4, 5, 6, <u>10</u> , 9, <u>17</u> }	{0, 0, 1, 0, 5, 2, 5}

U中dist最小的顶点: 3

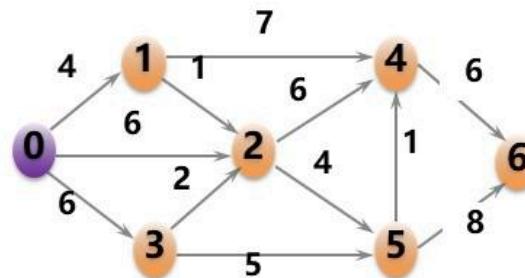
U中dist最小的顶点: 5

## 8.4 最短路径



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### Dijkstra算法示例演示



**dist[i]** : 源点v至顶点i的最短路径长度  
**path[i]** : 源点v至顶点i的最短路径上顶点i的前一个顶点的编号

S	U	dist[]	path[]
{0,1,2,3,5}	{4,6}	0 1 2 3 4 5 6	0 1 2 3 4 5 6
		{0, 4, 5, 6, <u>10</u> , 9, <u>17</u> }	{0, 0, 1, 0, <b>5</b> , 2, <b>5</b> }
		U中dist最小的顶点: 4	
{0,1,2,3,5,4}	{6}	0, 4, 5, 6, 10, 9, <u>16</u>	{0, 0, 1, 0, 5, 2, 4}
		U中dist最小的顶点: 6	
{0,1,2,3,5,4,6}	{}	0, 4, 5, 6, 10, 9, 16	{0, 0, 1, 0, 5, 2, 4}
		最终结果	



## 5. 查找方法的性能指标



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### ● 5. 查找方法的性能指标

- 查找运算时间主要花费在关键字比较上，通常把查找过程中执行的关键字平均比较个数（称为平均查找长度）作为衡量一个查找算法效率优劣的标准。
- 平均查找长度ASL (Average Search Length) 定义为：

$$ASL = \sum_{i=1}^n p_i c_i$$

- $n$ 是查找表中元素的个数。
- $p_i$ 是查找第  $i$  个元素的概率，一般地，认为每个元素的查找概率相等，即  $p_i=1/n$  ( $1 \leq i \leq n$ )。
- $c_i$ 是找到第  $i$  个元素所需进行的比较次数。



#### 平均查找长度ASL分为

- 成功情况下的平均查找长度  
——**ASL<sub>成功</sub>**
- 不成功情况（失败）下的平均查找长度  
——**ASL<sub>不成功</sub>**



- 成功情况下（概率相等）的平均查找长度 $ASL_{\text{成功}}$ 是指找  
**到T中任一元素平均需要的关键字比较次数。**
- 例如， $n=9$ ：

关键字	5	1	4	8	7	9	2	4	3
找到时的比较次数	1	2	3	4	5	6	7	8	9



$$ASL_{\text{成功}} = \frac{1+2+3+4+5+6+7+8+9}{9} = 5$$



- 不成功情况下的平均查找长度  $ASL_{\text{不成功}}$  是指 **查找失败**  
**(在T中未查找到) 平均需要的关键字比较次数。**



- 例如， $n=9$ :

关键字	5	1	4	8	7	9	2	4	3
找到时的比较次数	1	2	3	4	5	6	7	8	9

- 按顺序查找， $ASL_{\text{不成功}} = n = 9$



## 9.2.2 折半查找



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1. 基本折半查找

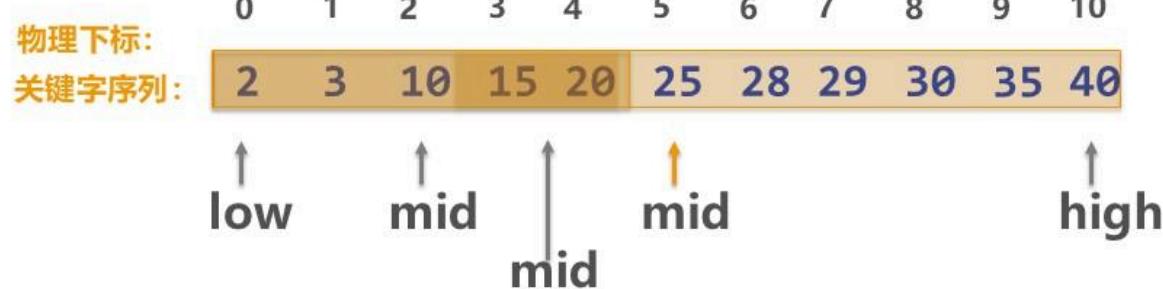
- 折半查找也称为二分查找，要求线性表中的元素必须已按关键字值有序（递增或递减）排列。
- 思路：





- 例如，在关键字有序序列：(2, 3, 10, 15, 20, 25, 28, 29, 30, 35, 40) 采用折半查找法查找关键字为15的元素。

- 找关键字为15的元素



查找成功，关键字为15的元素的逻辑序号为4  
关键字比较次数为3



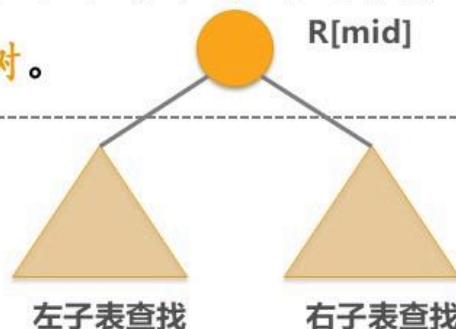
其算法如下（在有序表  $R[0..n-1]$  中进行折半查找，成功时返回元素的逻辑序号，失败时返回 0）：

```
int BinSearch(RecType R[],int n,KeyType k)
{ int low=0,high=n-1,mid;
  while (low<=high)           //当前区间存在元素时循环
  {
    mid=(low+high)/2;
    if (R[mid].key==k)         //查找成功返回其逻辑序号mid+1
      return mid+1;
    if (k<R[mid].key)          //继续在R[low..mid-1]中查找
      high=mid-1;
    else
      low=mid+1;              //继续在R[mid+1..high]中查找
  }
  return 0;
}
```



### 二分查找过程可用二叉树来描述：

- 把当前查找区间的中间位置上的元素作为根；
- 左子表和右子表中的元素分别作为根的左子树和右子树。



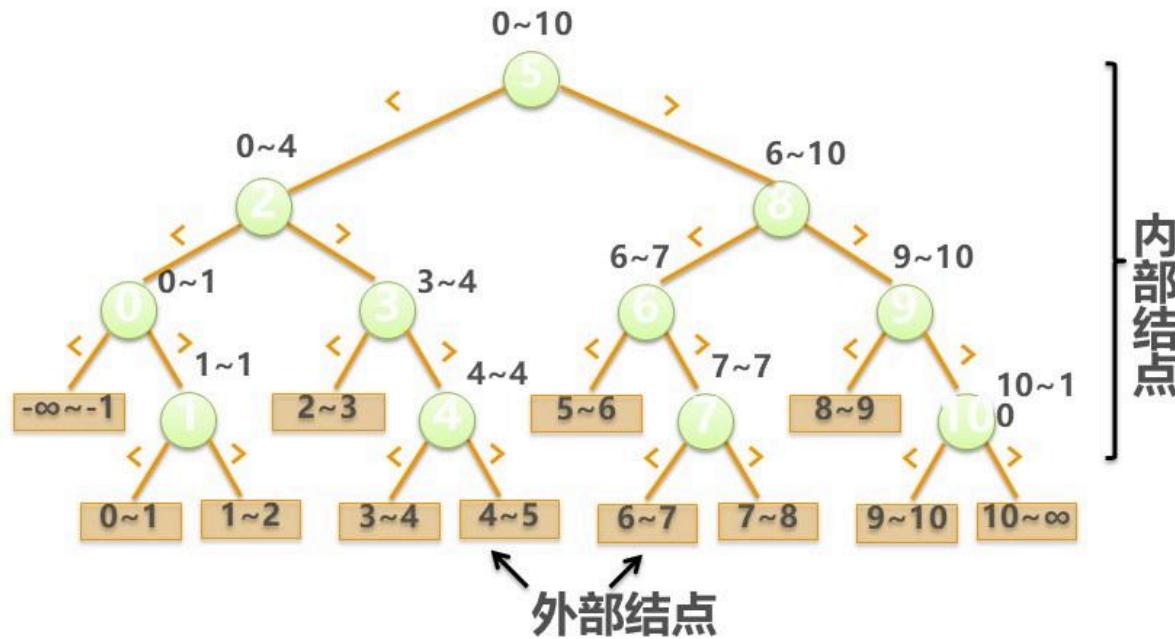
这样的二叉树称为**判定树**或**比较树**。



## R[0..10]的二分查找的判定树 (n=11)



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



外部结点即查找失败对应的结点，是虚拟的

$n$ 个关键字：内部结点为 $n$ 个，外部结点为 $n+1$ 个



### 【例9.1】对于给定11个数据元素的有序表：

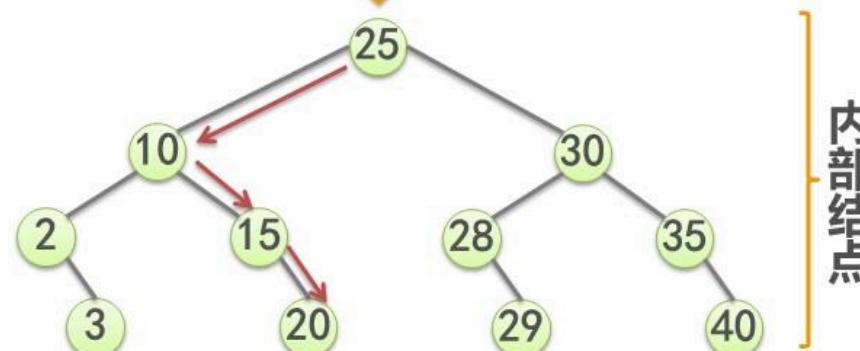
(2, 3, 10, 15, 20, 25, 28, 29, 30, 35, 40)

采用二分查找，试问：

1. 若查找给定值为20的元素，将依次与表中哪些元素比较？
2. 若查找给定值为26的元素，将依次与哪些元素比较？
3. 假设查找表中每个元素的概率相同，求**查找成功时的平均查找长度**和**查找不成功时的平均查找长度**。

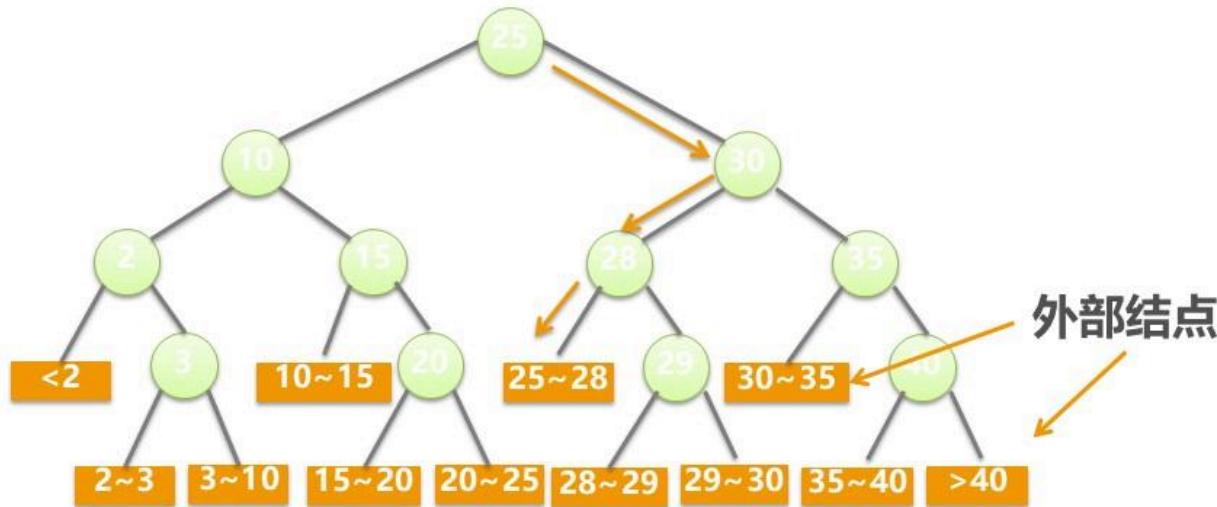


0 1 2 3 4 5 6 7 8 9 10  
(2, 3, 10, 15, 20, 25, 28, 29,  
30, 35, 40)



(1) 若查找给定值为 20 的元素，依次与表中 25、10、15、20 元素比较，共比较 4 次。

关键字比较的个数



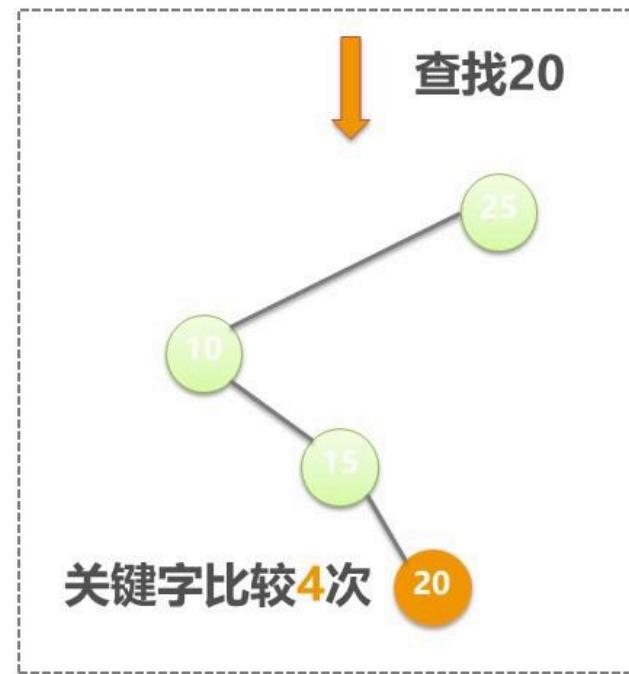
(2) 若查找给定值为26的元素，依次与25、30、28元  
素比较，共比较3次。

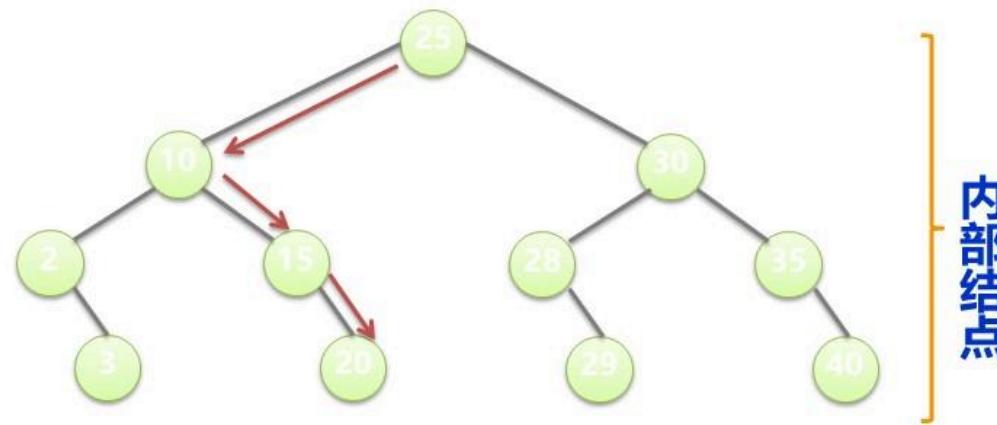


## 成功二分查找



清华大学出版社  
TSINGHUA UNIVERSITY PRESS





(3) 在查找成功时，会找到某个内部结点，则**成功**时的平均查找长度：

$$ASL_{\text{成功}} = \frac{1 \times 1 + 2 \times 2 + 4 \times 3 + 4 \times 4}{11} = 3$$

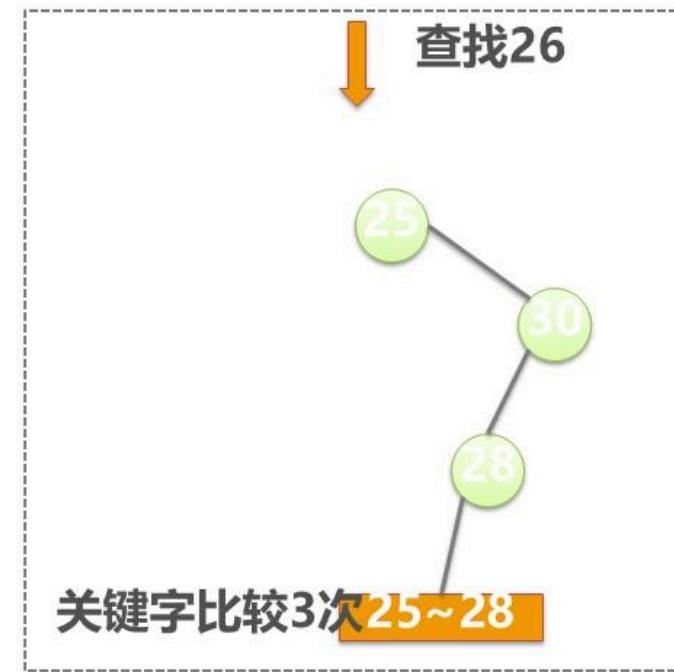


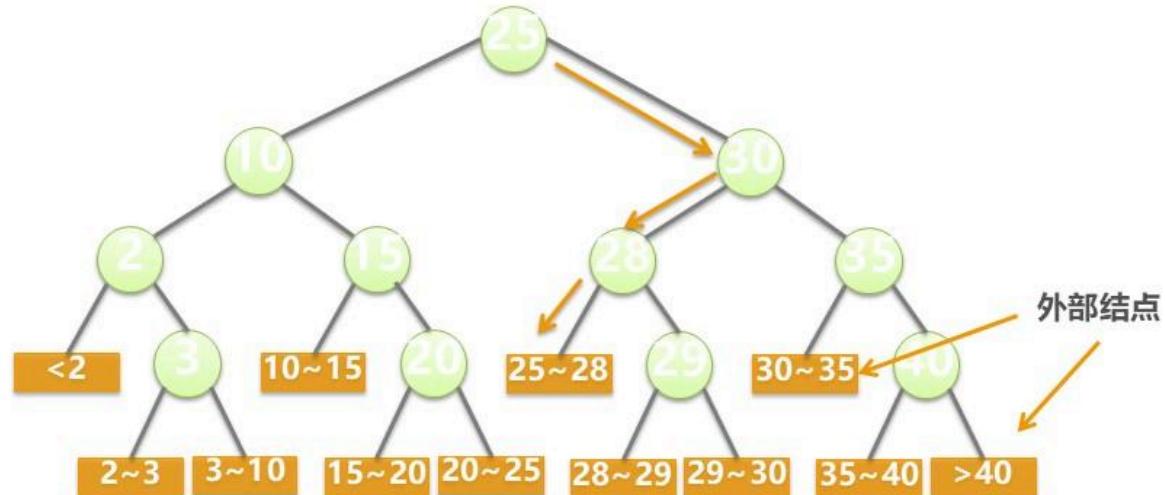
## 不成功二分查找



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

比较过程经历了一条从判定树根到某个外部结点的路径，所需的关键字比较次数是该路径上内部结点的总数，即**该外部结点的层次减1。**





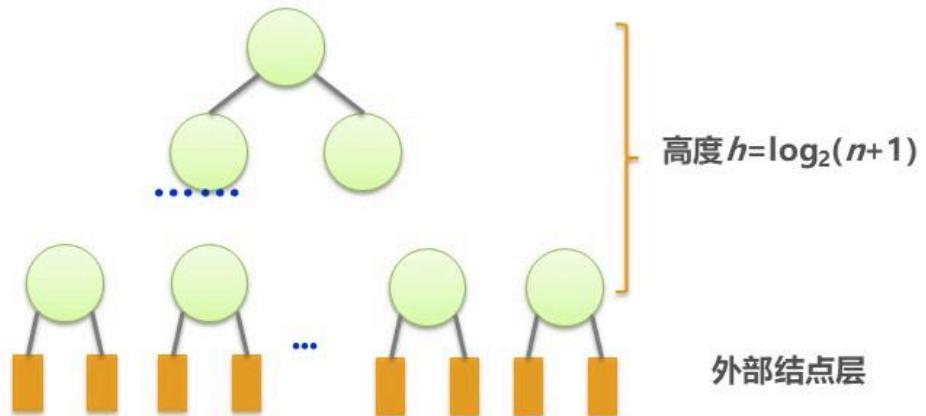
(3) 在查找不成功时，会找到某个外部结点，则**不成功**时的平均查找长度：

$$ASL_{\text{不成功}} = \frac{4 \times 3 + 8 \times 4}{12} = 3.67$$



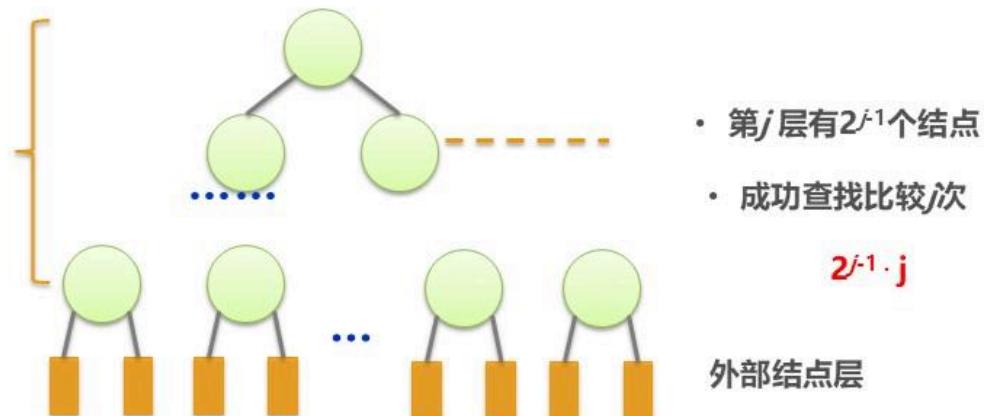
当  $n$  比较大时，将判定树看成：

- 内部结点总数  $n=2^h-1$ 、  
高度为  $h=\log_2(n+1)$  的  
**满二叉树**（高度  $h$  不计  
外部结点）。
- 树中第  $j$  层上的元素个  
数为  $2^{j-1}$ ，查找该层上  
的每个元素需要进行  $j$   
次比较。



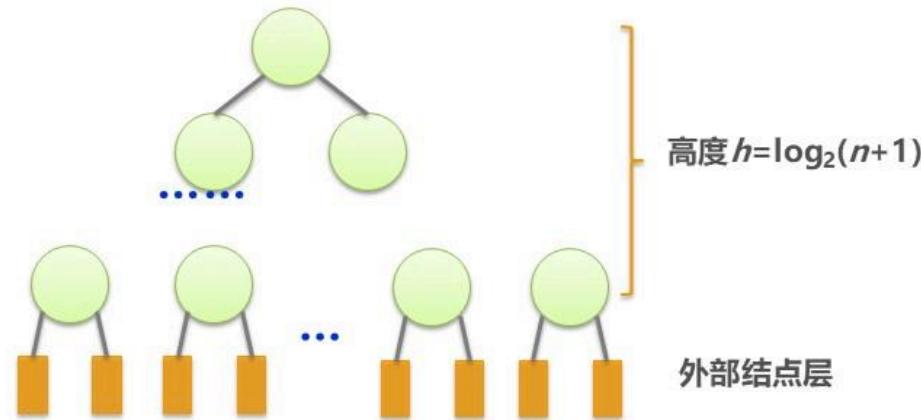


高度  
 $h = \log_2(n+1)$



在等概率假设下，二分查找成功时的平均

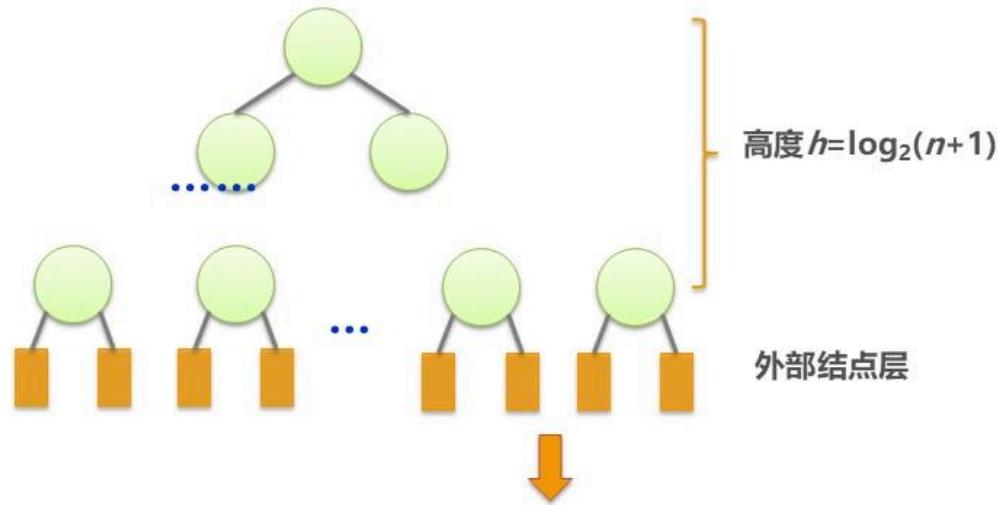
$$ASL_{bn} = \sum_{i=1}^n p_i c_i = \frac{1}{n} \sum_{j=1}^h 2^{j-1} \times j = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$



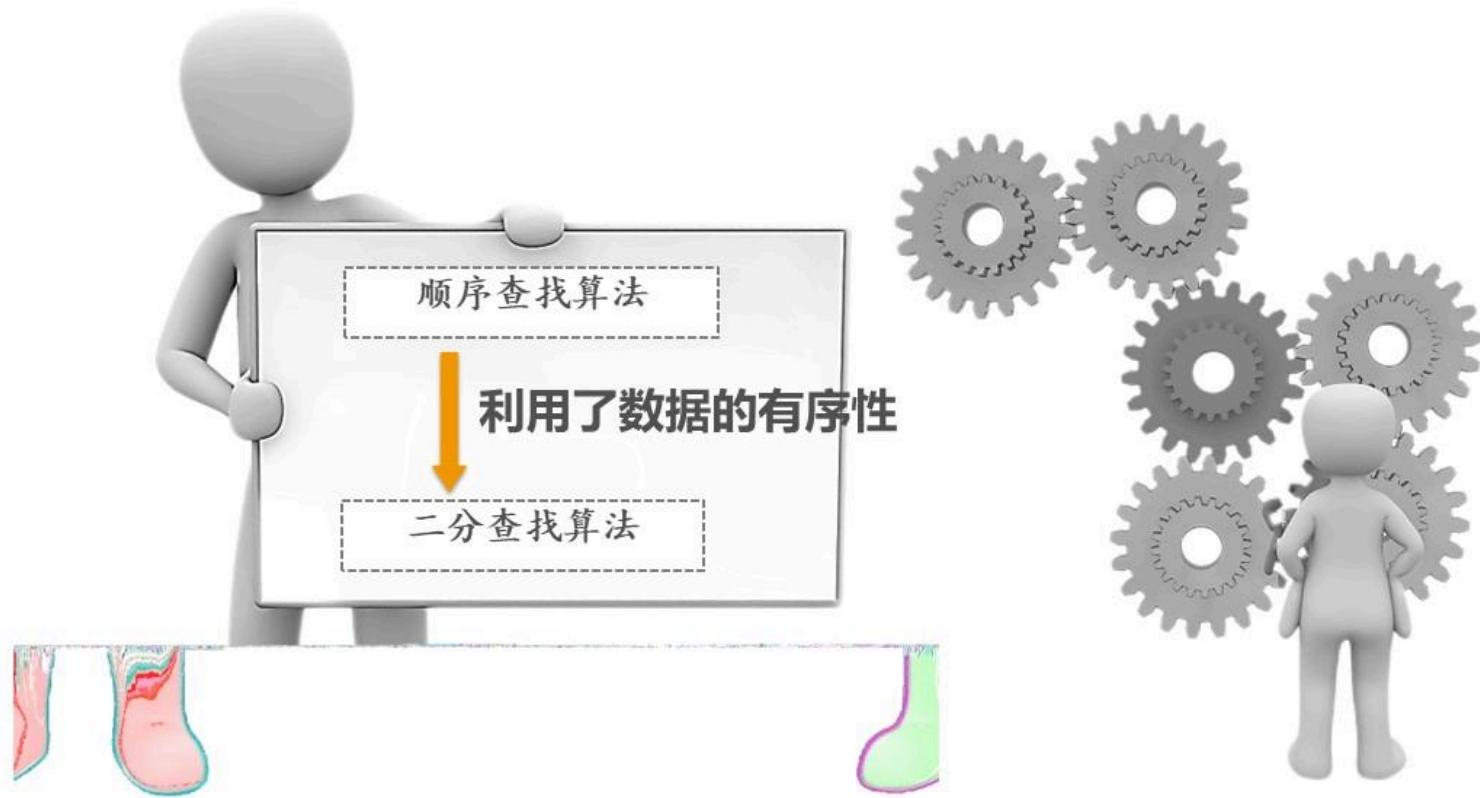
在等概率假设下，二分查找不成功时的平均查找长度为：  
 $\text{ASL}_{\text{不成功}} = h = \log_2(n+1)$



二分查找的时间复杂度为  $O(\log_2 n)$ 。



- 对于  $n$  个元素，二分查找成功时最多的关键字比较次数为： $\lceil \log_2(n+1) \rceil$
- 不成功时最多关键字比较次数为： $\lceil \log_2(n+1) \rceil$ 。





### 9.3.1 二叉排序树

**二叉排序树**（简称BST）又称**二叉查找（搜索）树**，其定义为：  
二叉排序树或者是空树，或者是满足如下性质（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值（指关键字值）均小于根结点值。
- ② 若它的右子树非空，则右子树上所有结点值均大于根结点值。
- ③ 左、右子树本身又各是一棵二叉排序树。

**注意：**二叉排序树中没有相同关键字的结点。

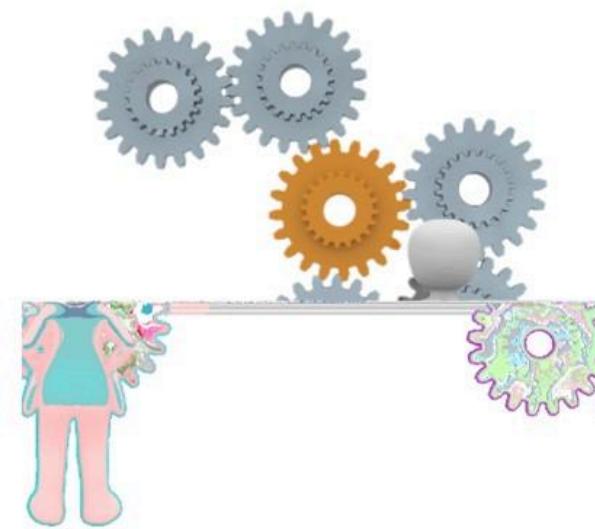
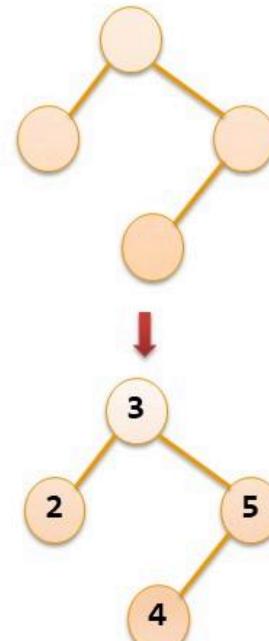


## 二叉树结构



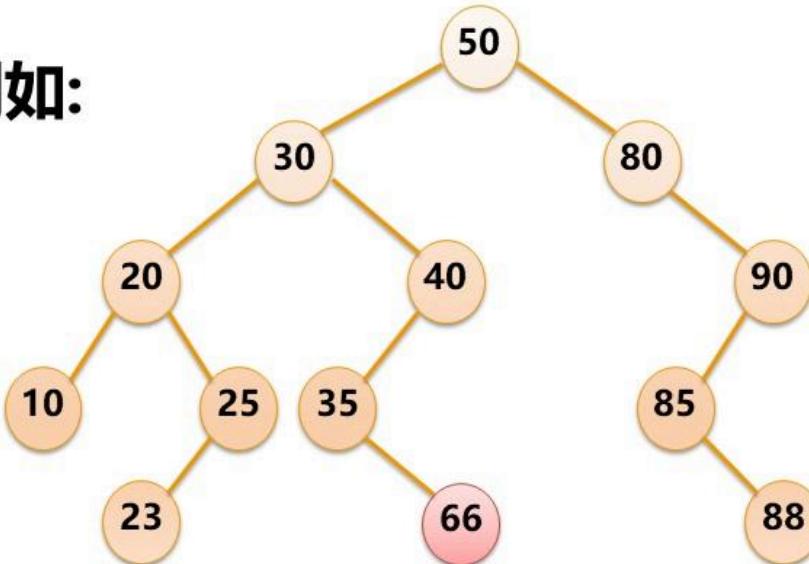
满足BST性质：  
结点值约束

## 二叉排序树





例如：

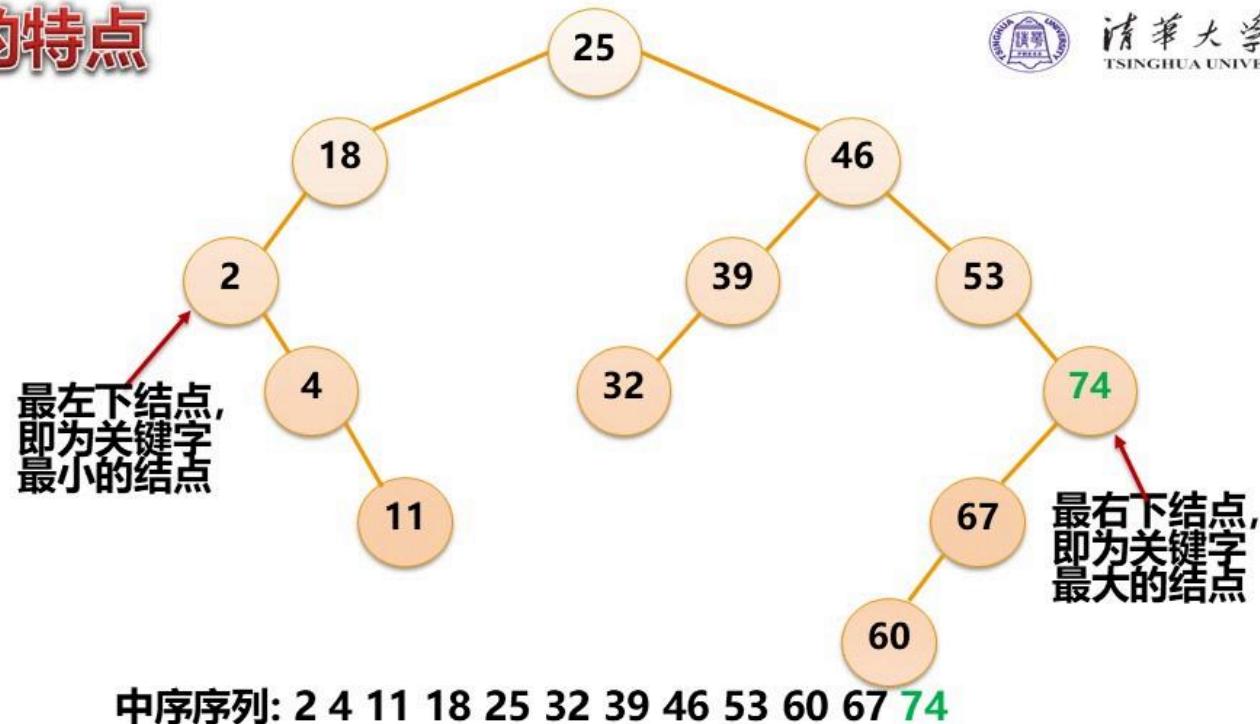


不 是二叉排序树。

## 二叉排序树的特点



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

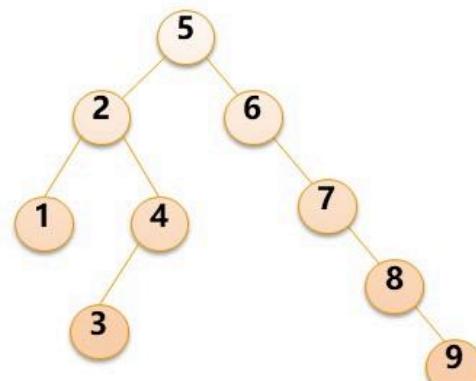


### 特点:

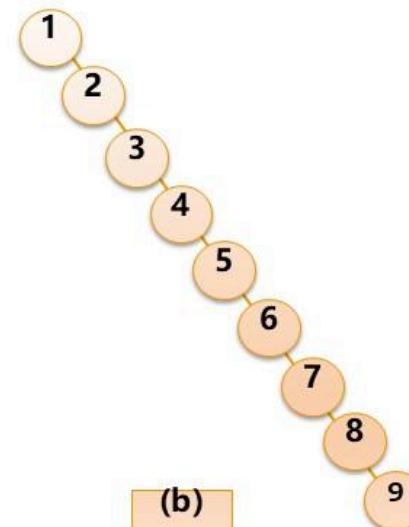
- 二叉排序树的中序序列是一个递增有序序列
- 根结点的最左下结点是关键字最小的结点
- 根结点的最右下结点是关键字最大的结点



- 一个关键字集合有多个关键字序列，不同的关键字序列采用上述创建 CreateBST 算法得到的二叉排序树可能不同。
- 例如，同样是 1 ~ 9 的关键字集合：
  - 由关键字序列 (5, 2, 1, 6, 7, 4, 8, 3, 9) 创建的二叉排序树如图(a)所示。
  - 由关键字序列 (1, 2, 3, 4, 5, 6, 7, 8, 9) 创建的二叉排序树如图(b)所示。



(a)



(b)





清华大学出版社  
TSINGHUA UNIVERSITY PRESS



示例

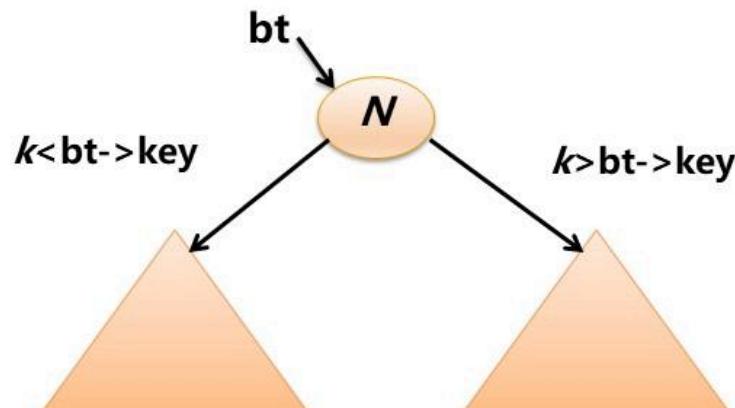
由一组关键字23、12、45、36构成的不同二叉排序树  
有（ ）棵。

- A. 1
- B. 10
- C. 14
- D. 无法确定



## 2、二叉排序树的查找

- 二叉排序树可看做是一个有序表（中序序列是递增有序序列）。
- 在二叉排序树上进行查找，和二分查找类似，也是一个逐步缩小查找范围的过程。



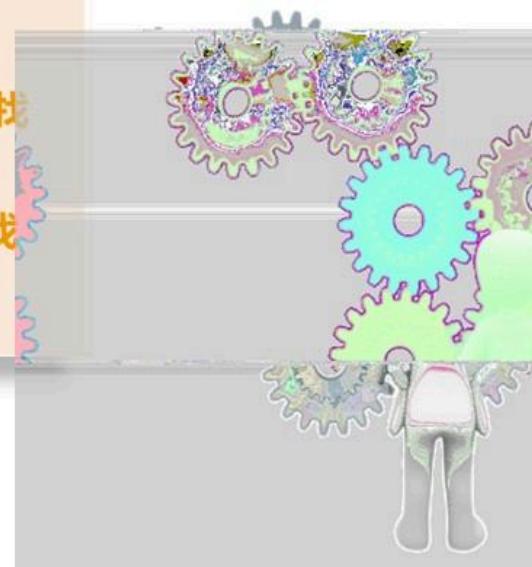
每一层只和一个结点进行关键字比较！



■ 查找算法**SearchBST(bt,k)**: 在二叉排序树bt上查找关键字为k的结点，成功时返回该结点地址，否则返回NULL。

```
BSTNode *SearchBST(BSTNode *bt, KeyType k)
{  if (bt==NULL || bt->key==k)      //递归出口
   return bt;
  if (k<bt->key)
   return SearchBST(bt->lchild, k); //在左子树中递归查找
  else
   return SearchBST(bt->rchild, k); //在右子树中递归查找
}
```

↑  
递归算法





- 查找算法**SearchBST(bt,k)**: 在二叉排序树bt上查找关键字为k的结点，成功时返回该结点地址，否则返回NULL。

```
BSTNode *SearchBST1(BSTNode *bt,KeyType k)
{ BSTNode* p=bt;
  while(p!=NULL)
  { if(p->key==k)
    break;           //找到关键字为k的结点p退出循环
    else if(k<p->key)
      p=p->lchild;
    else
      p=p->rchild;
  }
  return p;           //返回查找结果
}
```



非递归算法

## 二叉排序树查找性能分析

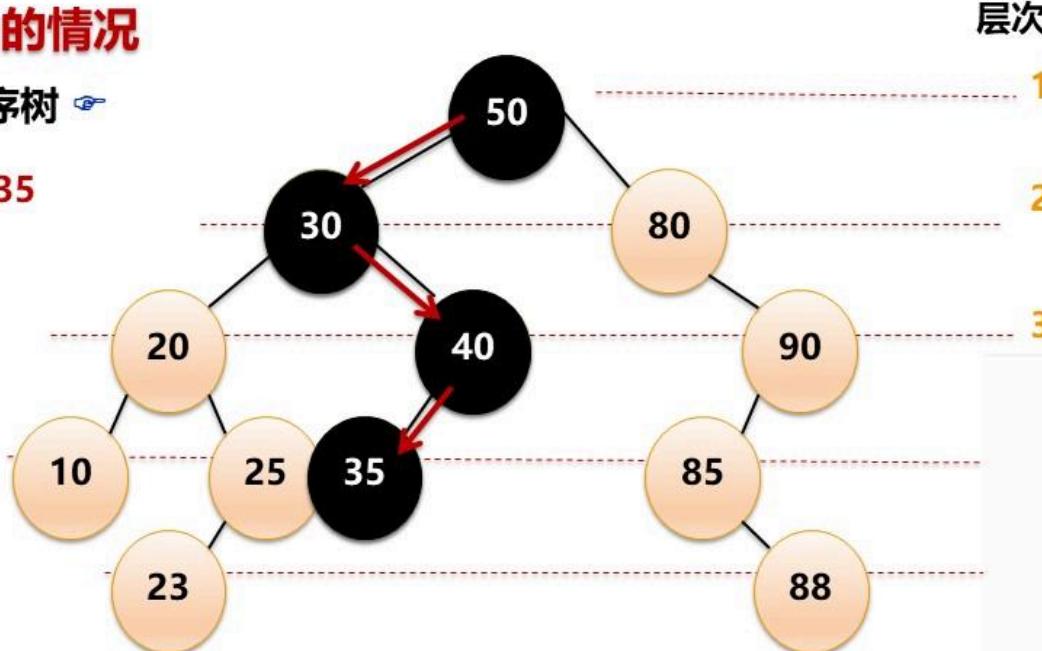


清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### ① 查找成功的情况

一棵二叉排序树

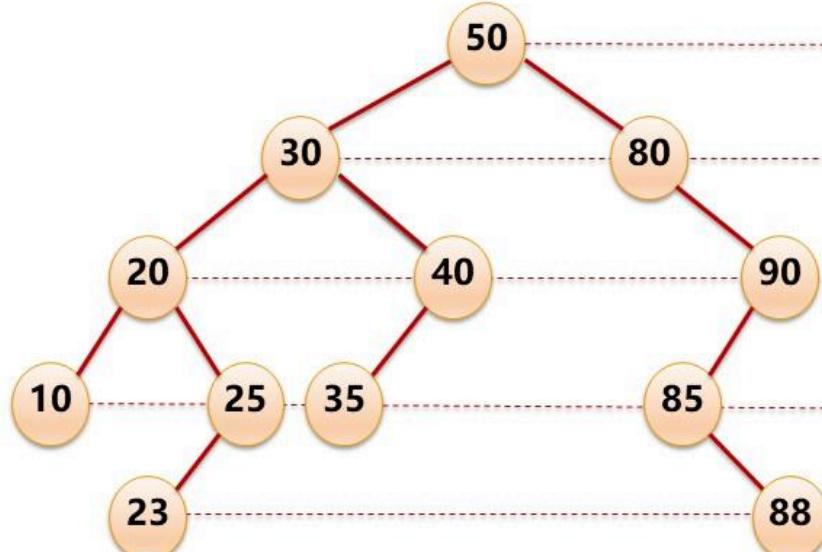
查找: 35



- 从根结点开始，查找最后落在某个**内部结点**中
- 比较次数为该内部结点的层次
- 查找路径是二叉排序树的一部分，本身也是一棵二叉排序树



## 层次



在等概率假设下，查找成功的平均查找长度

$$\text{ASL}_{\text{成功}} = \frac{1 \times 1 + 2 \times 2 + 3 \times 3 + 4 \times 4 + 2 \times 5}{12} = 3.33$$

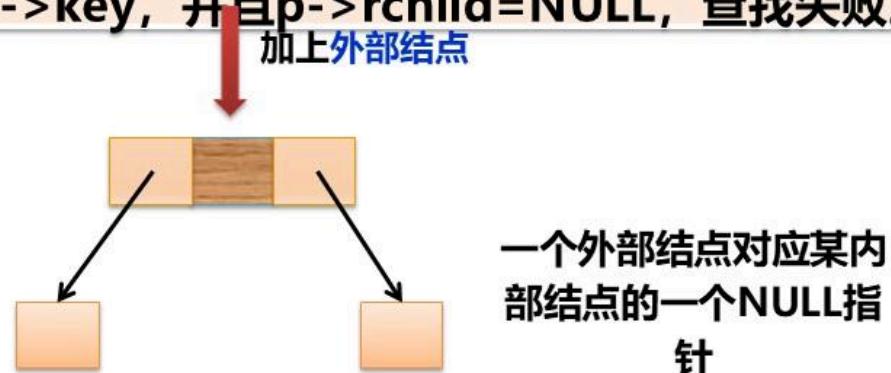




## ② 查找失败的情况



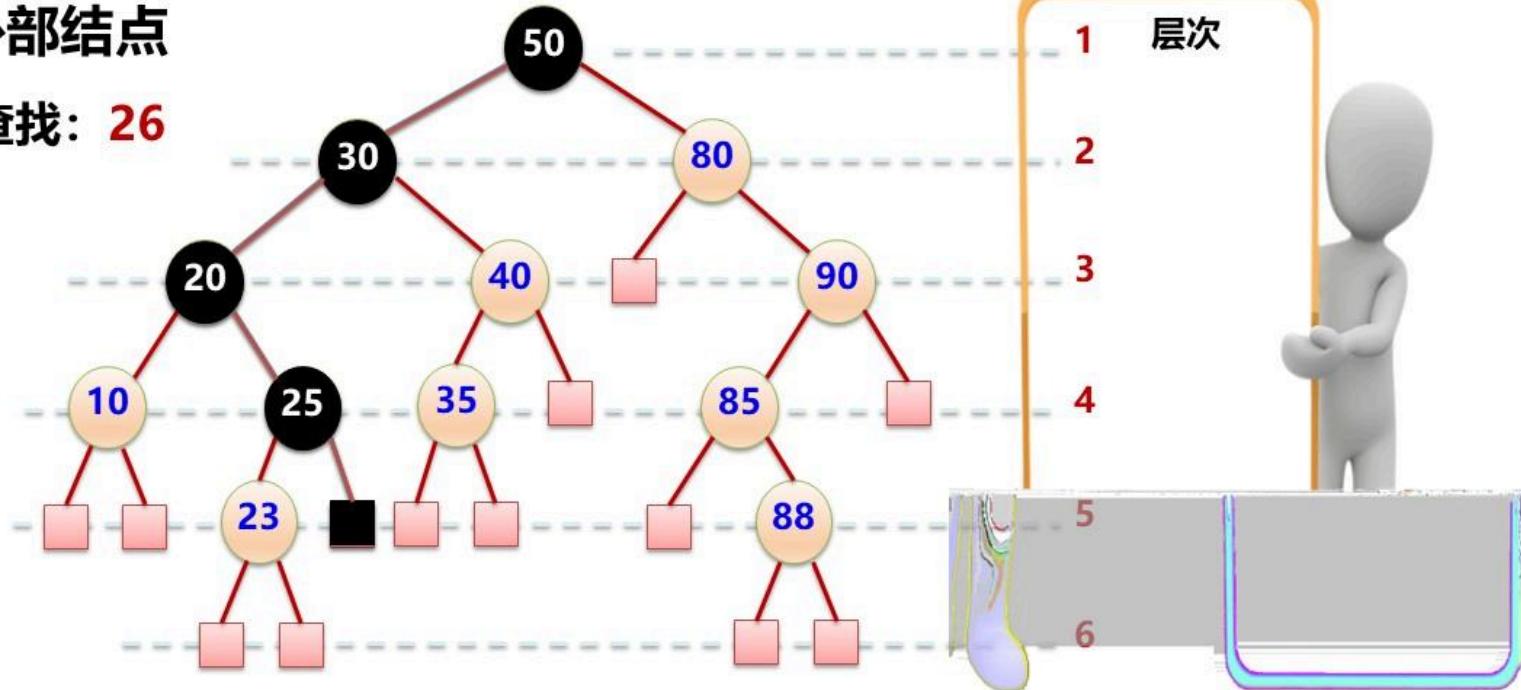
- 若  $k < p->\text{key}$ , 并且  $p->\text{lchild}=\text{NULL}$ , 查找失败。
- 若  $k > p->\text{key}$ , 并且  $p->\text{rchild}=\text{NULL}$ , 查找失败。



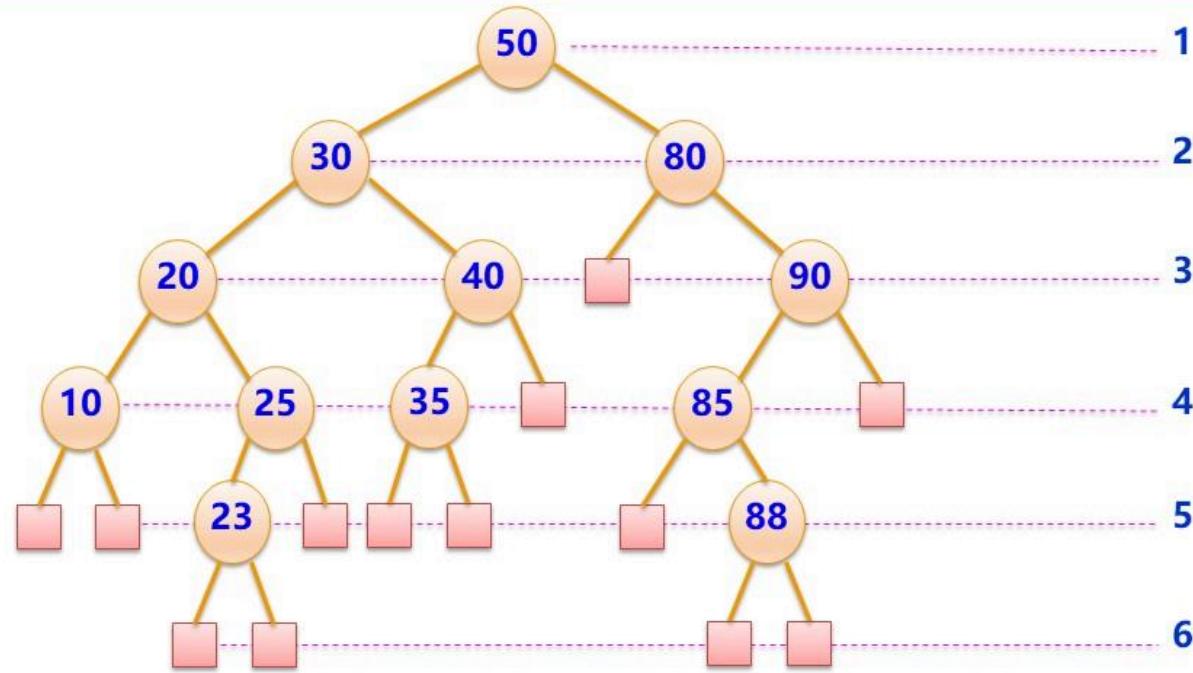


## 加上外部结点

查找: 26



- 从根结点开始，查找最后落在某个外部结点中
- 比较次数为该外部结点的层次减1

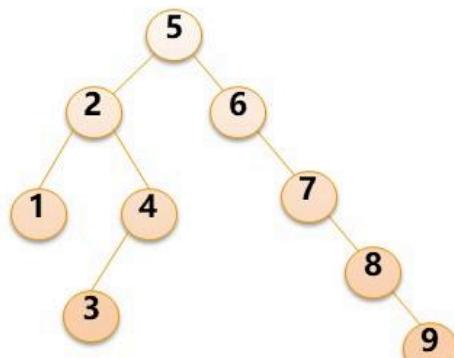


在等概率假设下，查找不成功的平均查找长度

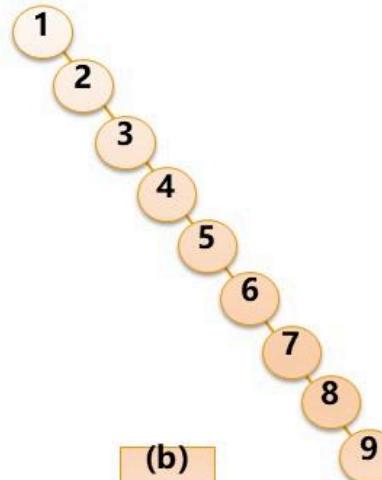
$$ASL_{\text{不成功}} = \frac{1 \times 2 + 2 \times 3 + 6 \times 4 + 4 \times 5}{13} = 4$$



## 在等概率假设下，查找成功的平均查找长度



(a)



(b)



$$ASL_a = \frac{1 \times 1 + 2 \times 2 + 3 \times 3 + 2 \times 4 + 1 \times 5}{9} = 3$$

$$ASL_b = \frac{1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4 + 1 \times 5 + 1 \times 6 + 1 \times 7 + 1 \times 8 + 1 \times 9}{9} = 5$$



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



**【例9.3】已知关键字序列为:**

**{25, 18, 46, 2, 53, 39, 32, 4, 74, 67, 60, 11}**

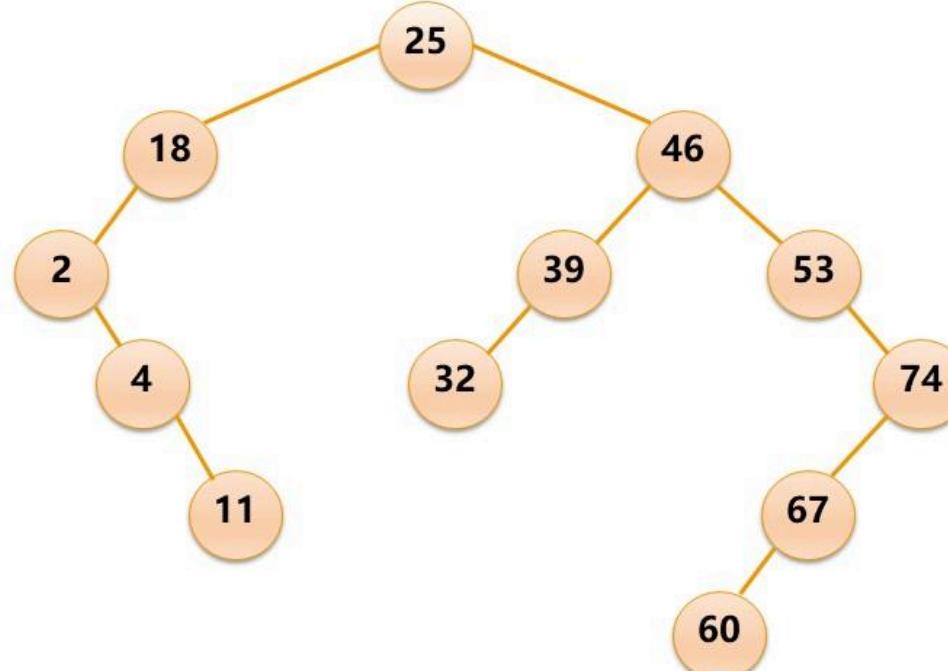
按表中的元素顺序依次插入到一棵初始为空的二叉排序树中，画出结果二叉排序树。  
求在等概率的情况下查找成功的平均查找长度和查找不成功的平均查找长度。



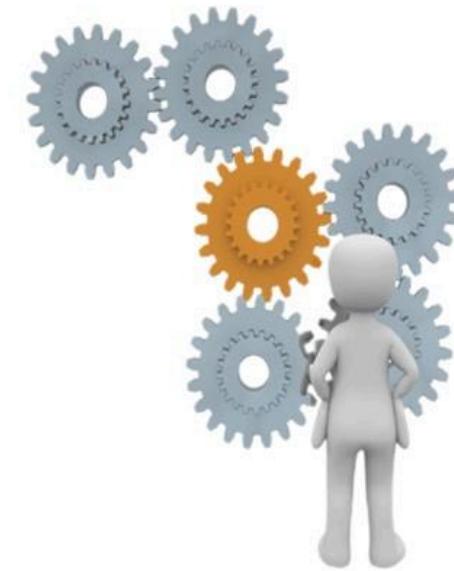
序列: 25 18 46 2 53 39 32 4 74 67 60 11



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

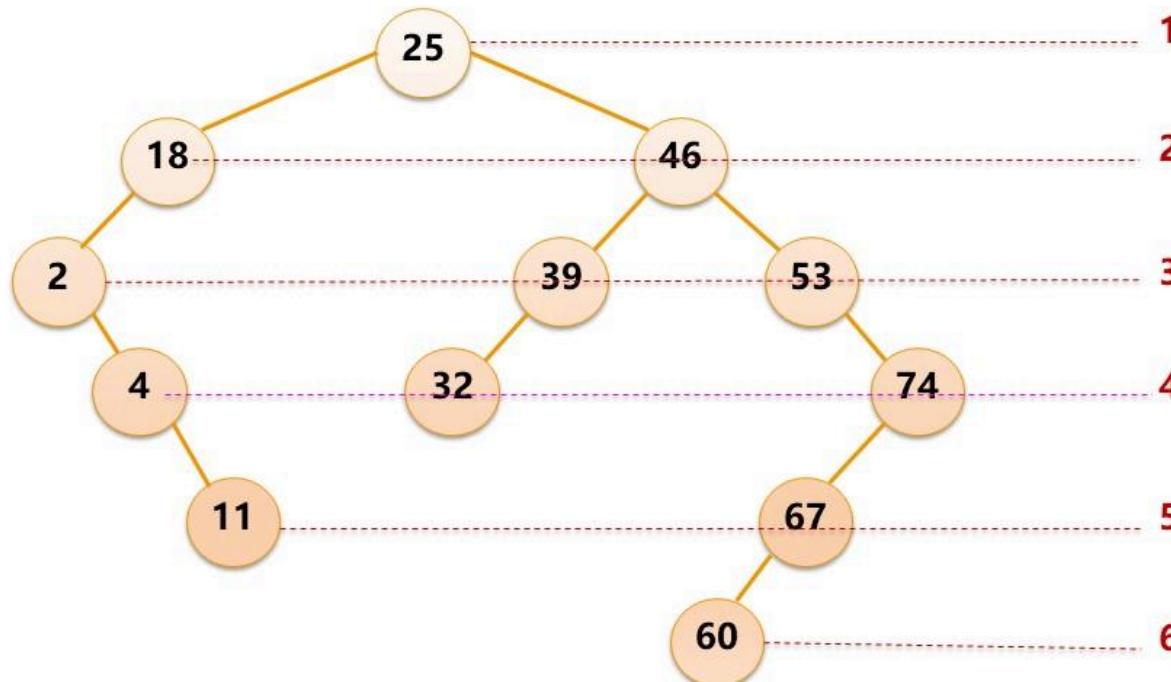


二叉排序树创建完毕





## 层次

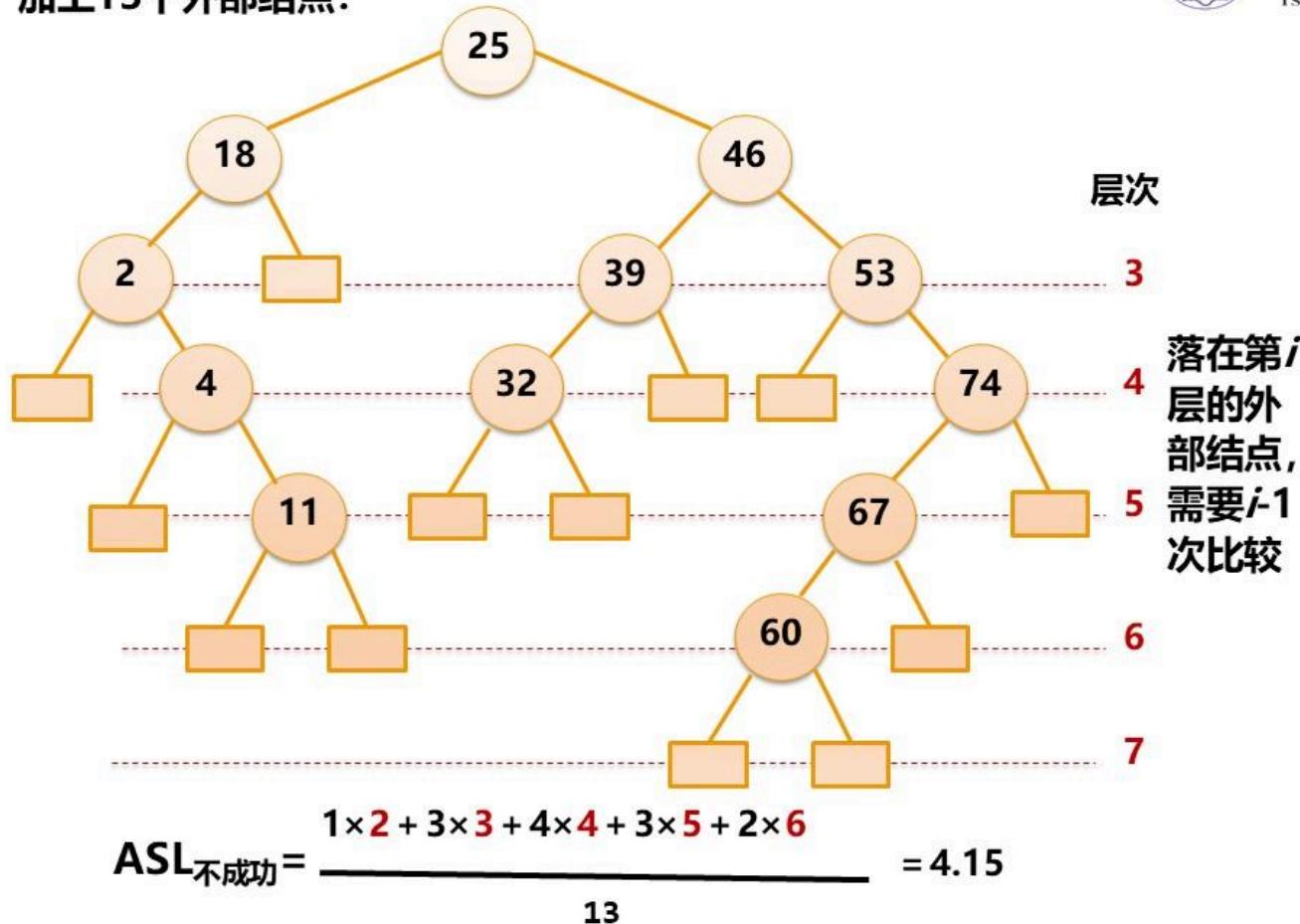


成功找到第*i*层的结点，需要*i*次比较

$$ASL_{\text{成功}} = \frac{1 \times 1 + 2 \times 2 + 3 \times 3 + 3 \times 4 + 2 \times 5 + 1 \times 6}{12} = 3.5$$



加上13个外部结点:





示例

在含有27个结点的二叉排序树上，查找关键字为35的结点，以下（ ）是可能的关键字比较序列？

- A. 28, 36, 18, 46, 35
- B. 18, 36, 28, 46, 35
- C. 46, 28, 18, 36, 35
- D. 46, 36, 18, 28, 35

**判断标准：**在二叉排序树中的查找路径是原来二叉排序树的一部分，也一定构成一棵二叉排序树。



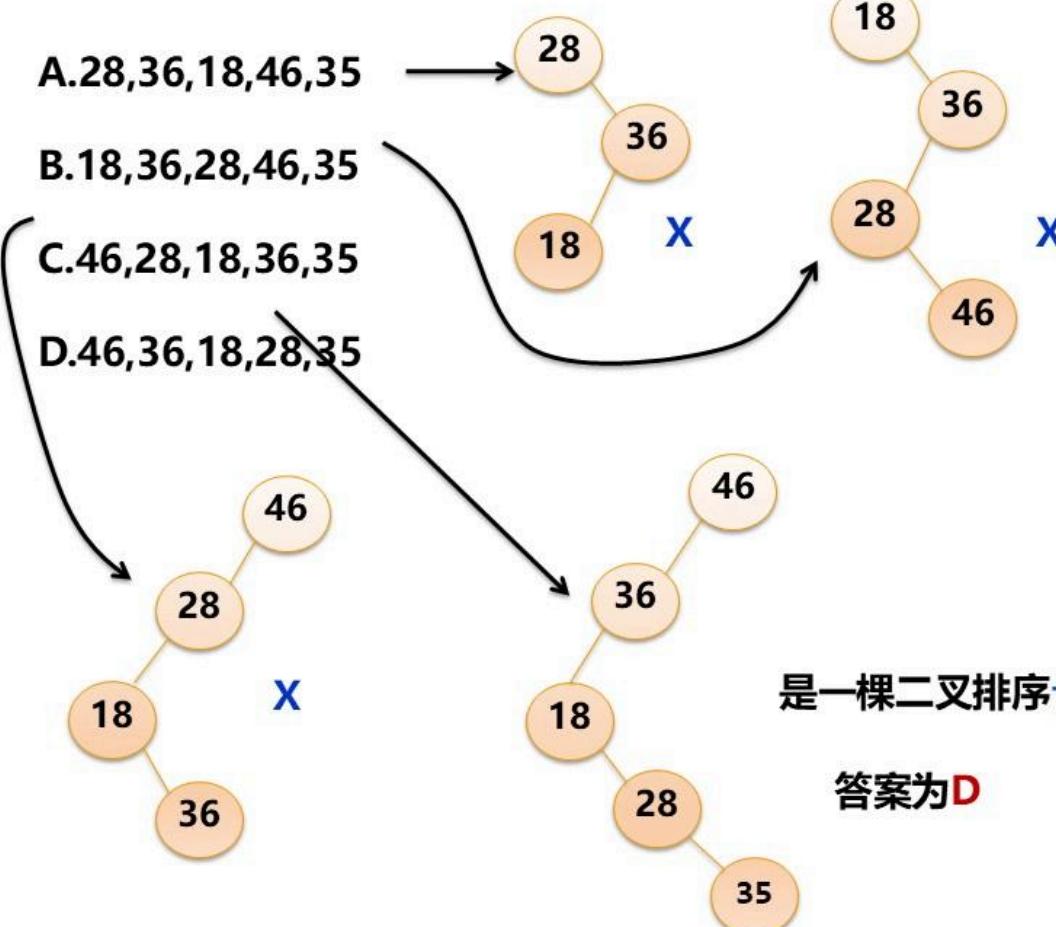


A. 28, 36, 18, 46, 35

B. 18, 36, 28, 46, 35

C. 46, 28, 18, 36, 35

D. 46, 36, 18, 28, 35





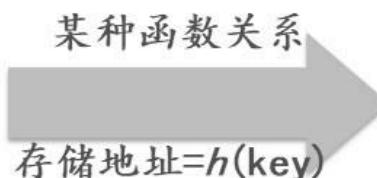
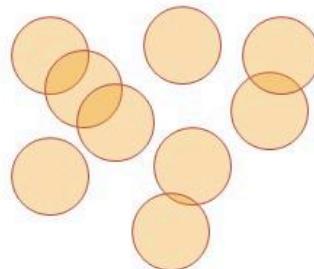
## 9.4 哈希表的查找



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### ● 9.4.1 哈希表的基本概念

#### 1、哈希表适合情况



● 注意：哈希表是一种存储结构，它并非适合任何情况，主要适合记录的**关键字与存储地址存在某种函数关系的数据**。

## 2. 几个概念



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 哈希冲突

- 对于两个关键字分别为 $k_i$ 和 $k_j$  ( $i \neq j$ ) 的记录，有 $k_i \neq k_j$ ，但 $h(k_i) = h(k_j)$ 。把这种现象叫做**哈希冲突（同义词冲突）**。
- 在哈希表存储结构的存储中，哈希冲突是很难避免的！！

### 示例

哈希表中出现同义词冲突是指（ ）。

- A. 两个元素具有相同的序号
- B. 两个元素的关键字不同，而其他属性相同
- C. 数据元素过多
- D. 两个元素的关键字不同，而对应的哈希函数值（存储地址）相同

答案：D。

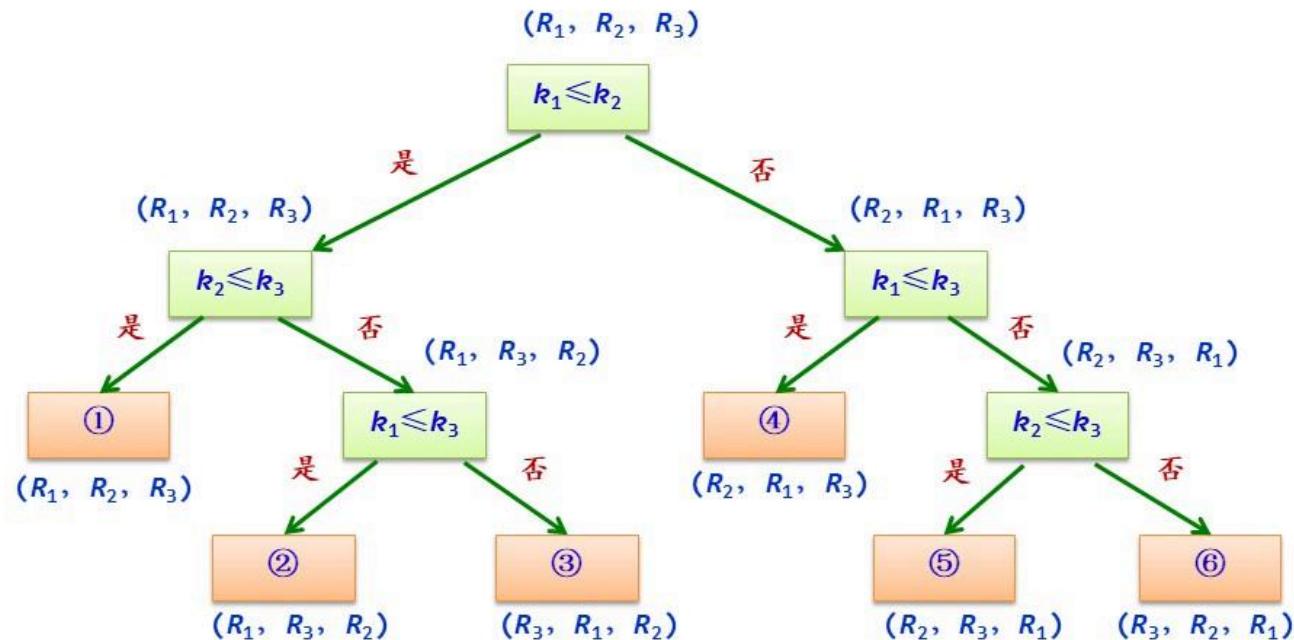


## 1.1 排序的基本概念

### 1.1.3 内排序

2

所有可能的初始序列的排序过程构成一个决策树：



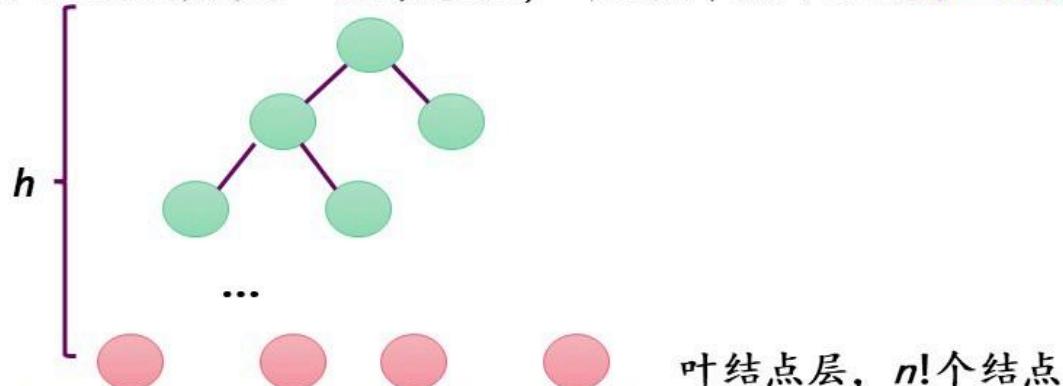
决策树是一棵有 $n!$ 个叶结点的二叉树。

316/39



### 1.1.4 基于比较的内排序算法最快有多快

③ 决策树可以近似看成是一颗高度为  $h$ , 叶结点个数为  $n!$  的满二叉树。



- 叶结点个数 =  $n!$
- 总结点个数 =  $2n! - 1$
- $h = \log_2(\text{总结点个数} + 1) = \log_2(n!) + 1$
- 平均关键字比较次数 =  $h - 1 = \log_2(n!) \approx n \log_2 n$
- 移动次数也是同样的数量级, 即这样的算法平均时间复杂度为  $O(n \log_2 n)$ 。

317/39

## 1.2 插入排序

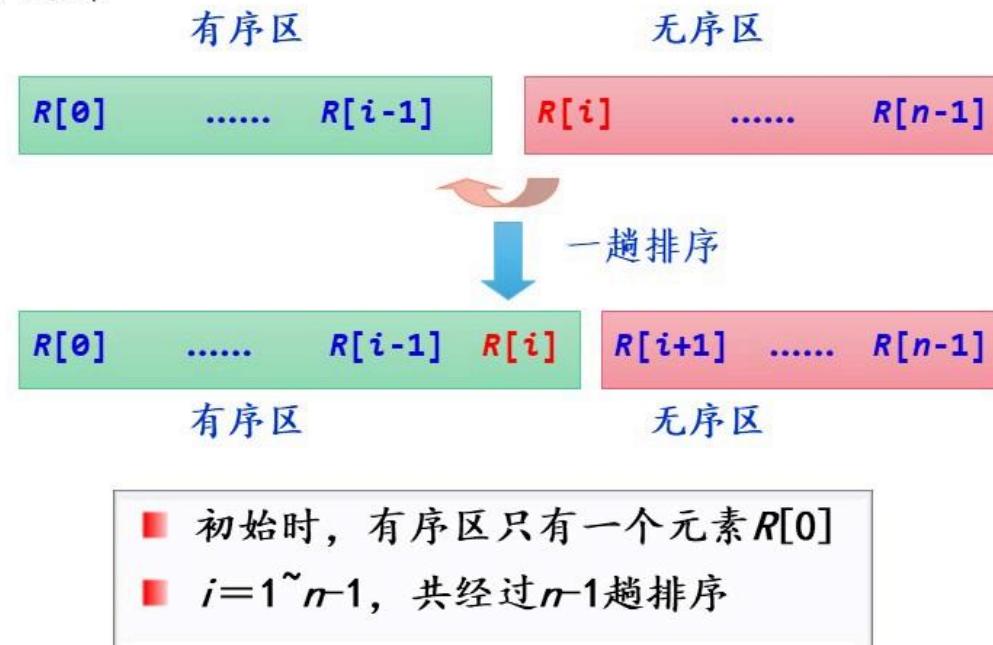


清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 1. 直接插入排序

##### 1 直接插入排序的基本思路



318/39

## 1.2 插入排序

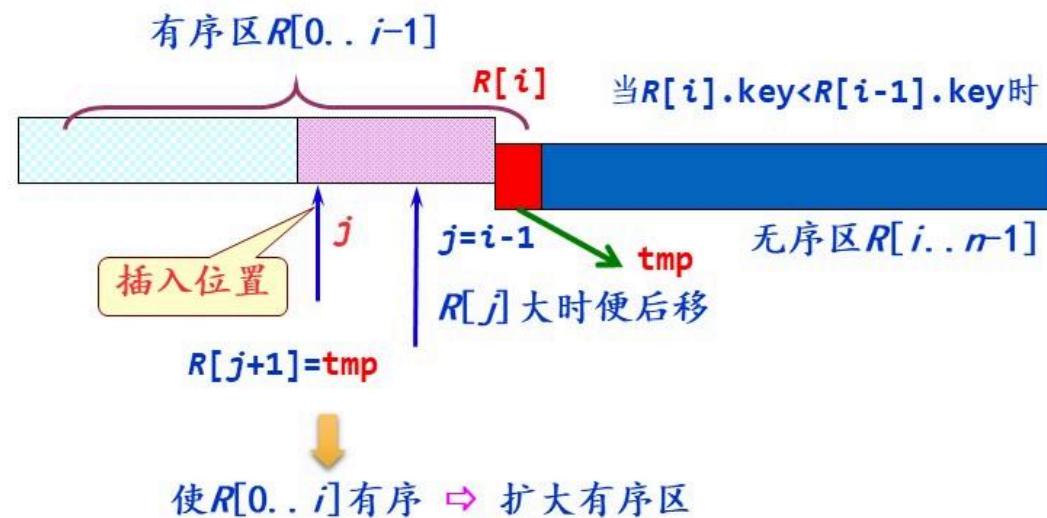


清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 1. 直接插入排序

- 2 一趟直接插入排序：在有序区中插入  $R[i]$  的过程



319/39

## 1.2 插入排序

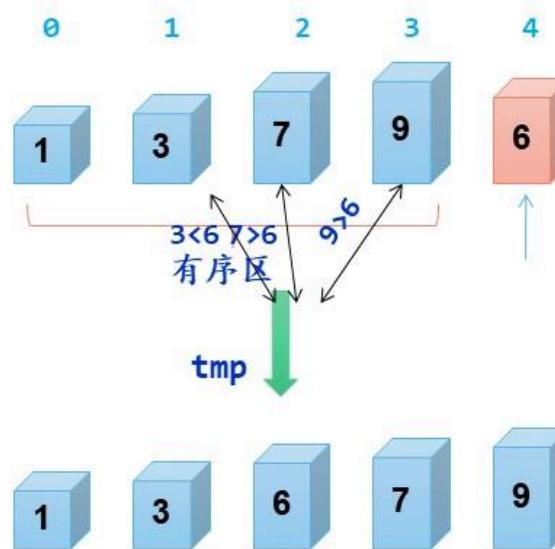


清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1. 直接插入排序

#### ③ 在无序区从后向前查找

例如



😊 在无序区从后向前查找  
 $\leq \text{tmp. key}$  的第一个  $R[j]$

320/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 1.. 直接插入排序

##### ④ 直接插入排序过程

设待排序的表有10个元素，其关键字分别为(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)。说明采用直接插入排序方法进行排序的过程。

321/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 1. 直接插入排序

#### 5 直接插入排序的算法

```
void InsertSort(RecType R[], int n)
{ int i, j; RecType tmp;
  for (i=1;i<n;i++)
  { if (R[i].key<R[i-1].key) //反序时
    { tmp=R[i];
      j=i-1;
      do //找R[i]的插入位置
      { R[j+1]=R[j]; //将关键字大于R[i].key的元素后移
        j--;
      } while (j>=0 && R[j].key>tmp.key)
      R[j+1]=tmp; //在j+1处插入R[i]
    }
  }
}
```

322/39

## 直接插入排序算法分析：

最好的情况（关键字在元素序列中顺序有序）：

“比较”的次数：

$$\sum_{i=1}^{n-1} 1 = n - 1$$

“移动”的次数：

0

最好

$O(n)$

最坏的情况（关键字在元素序列中逆序有序）：

“比较”的次数：

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

“移动”的次数：

$$\sum_{i=1}^{n-1} (i+2) = \frac{(n-1)(n+4)}{2}$$

最坏

$O(n^2)$

总的平均比较和移动次数约为

$$\sum_{i=1}^{n-1} \left( \frac{i}{2} + \frac{i}{2} + 2 \right) = \sum_{i=1}^{n-1} (i+2) = \frac{(n-1)(n+4)}{2} = O(n^2)$$

平均

$O(n^2)$

323/86

## 1.2 插入排序



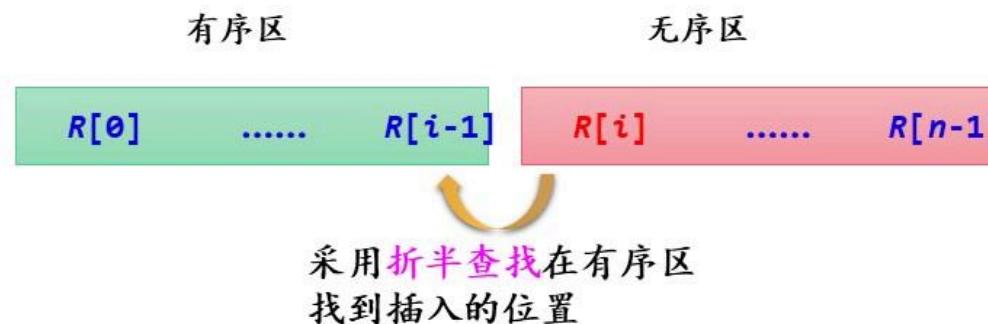
清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序

#### 2. 折半插入排序

##### ① 折半插入排序基本思路

查找采用折半查找方法，称为二分插入排序或折半插入排序。



324/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 2. 折半插入排序 ② 如何在 $R[\text{low..high}]$ 中查找插入 $R[i]$ 的位置？

在  $R[\text{low..high}]$  中查找  $\geq R[i].\text{key}$  ( $k=R[i].\text{key}$ ) 的最后的位置  $high$

① 如果  $R[\text{low..high}]$  不空：

$$\text{mid} = (\text{low} + \text{high}) / 2$$

$k < R[\text{mid}].\text{key}$

$R[\text{low}..\text{mid}-1] < k$

$k \geq R[\text{mid}].\text{key}$

$R[\text{mid}+1..\text{high}] \geq k$



折半插入排序

② 如果  $R[\text{low}..\text{high}]$  空：

$R[\text{low}..\text{high}] R[\text{high}+1..*]$

$R[i]$

325/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 2. 折半插入排序

#### 3 折半插入排序算法

```
void BinInsertSort(RecType R[], int n)
{ int i, j, low, high, mid;
  RecType tmp;
  for (i=1; i<n; i++)
  { if (R[i].key<R[i-1].key) //反序时
    { tmp=R[i]; //将R[i]保存到tmp中
      low=0; high=i-1;
      while (low<=high) //在R[low..high]中查找插入的位置
      { mid=(low+high)/2; //取中间位置
        if (tmp.key<R[mid].key)
          high=mid-1; //插入点在左半区
        else
          low=mid+1; //插入点在右半区
      }
      for (j=i-1; j>=high+1; j--) //元素后移
        R[j+1]=R[j];
      R[high+1]=tmp; //插入tmp
    }
  }
}
```

326/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 2.. 折半插入排序

- 在  $R[0..i-1]$  中查找插入  $R[i]$  的位置，折半查找的平均关键字比较次数为  $\log_2(i+1)-1$ 。
- 平均移动元素的次数为  $i/2+2$ 。

4

#### 算法分析

$$\sum_{i=1}^{n-1} [\log_2(i+1) - 1 + \frac{i}{2} + 2] = O(n^2)$$

- 折半插入排序采用折半查找，查找效率提高，即关键字比较次数减少了。
- 但元素移动次数不变，仅仅将分散移动改为集合移动。

327/39



### 1.2.2 主要的插入排序方法

#### 2. 折半插入排序

##### 示例

对同一待排序序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是（ ）。

##### 5 折半插入排序案例

- A. 排序的总趟数
- B. 元素的移动次数
- C. 使用辅助空间的数量
- D. 元素之间的比较次数

说明：本题为2012年全国考研题

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 3. 希尔排序

##### ① 希尔排序的基本思路 -- 分组插入

希尔排序  
的基本思路

- ①  $d=n/2$
- ② 将排序序列分为 $d$ 个组，在各组内进行直接插入排序
- ③ 递减 $d=d/2$ ，重复②，直到 $d=1$



算法最后一趟对所有数据进行了直接插入排序，所以结果一定是正确的。

329/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

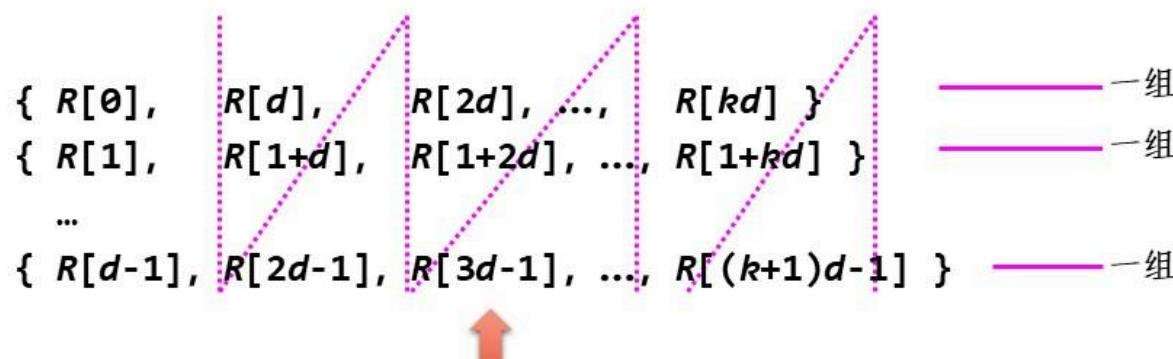
### 1.2.2 主要的插入排序方法

#### 3. 希尔排序

##### 2 希尔排序的过程

将元素序列分成若干子序列，分别对每个子序列进行直接插入排序。

例如：将  $n$  个元素分成  $d$  个子序列：



相距  $d$  个位置的元素分为一组

330/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 3. 希尔排序

初始序列	9	8	7	6	5	4	3	2	1	0
$d=5$	9	8	7	6	5	4	3	2	1	0

直接插入排序	4	3	2	1	0	9	8	7	6	5
--------	---	---	---	---	---	---	---	---	---	---

#### 2 希尔排序的过程

$d=d/2=2$	4	3	2	1	0	9	8	7	6	5
-----------	---	---	---	---	---	---	---	---	---	---

直接插入排序	0	1	2	3	4	5	6	7	8	9
--------	---	---	---	---	---	---	---	---	---	---

$d=d/2=1$	0	1	2	3	4	5	6	7	8	9
-----------	---	---	---	---	---	---	---	---	---	---

直接插入排序	0	1	2	3	4	5	6	7	8	9
--------	---	---	---	---	---	---	---	---	---	---

注意：对于 $d=1$ 的一趟，排序前的数据已将近正序！

331/39

## 1.2 插入排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS

### 1.2.2 主要的插入排序方法

#### 3. 希尔排序

##### 3 希尔排序算法

```
void ShellSort(RecType R[], int n)
{  int i, j, d;
   RecType tmp;
   d=n/2;           //增量置初值
   while (d>0)
   {  for (i=d;i<n;i++)
      { //对相隔d位置的元素组直接插入排序
          tmp=R[i];
          j=i-d;
          while (j>=0 && tmp.key<R[j].key)
          {  R[j+d]=R[j];
              j=j-d;
          }
          R[j+d]=tmp;
      }
      d=d/2;           //减小增量
   }
```

d循环：每个元素都参加排序了



```
for (i=1;i<n;i++)
{  tmp=R[i];
   j=i-1;
   while (j>=0 &&
          tmp.key<R[j].key)
   {  R[j+1]=R[j];
      j=j-1;
   }
   R[j+1]=tmp;
}
```

332/39



### 1.2.2 主要的插入排序方法

#### 3. 希尔排序

##### ④ 为什么希尔排序比直接排序好

希尔排序的时间复杂度约为 $O(n^{1.3})$ 。

为什么希尔排序比直接插入排序好?

例如：有10个元素要排序。

#### 希尔排序

##### 直接插入排序

$d=5$ : 分为5组，时间约为 $5 \times 2^2 = 20$

大约时间= $10^2=100$

$d=2$ : 分为2组，时间约为 $2 \times 5^2 = 50$

$d=1$ : 分为1组，几乎有序，时间约为10

333/39

## 10.3 交换排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.3.1 冒泡排序-算法

采用前面的冒泡排序方法对(2, 1, 3, 4, 5)进行排序

初始关键字



i=0



已经全部有序了

i=1



i=2



i=3



如何提高效率?

一旦某一趟比较时不出现元素交换，说明已排好序了，就可以结束本算法。



## 10.3 交换排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



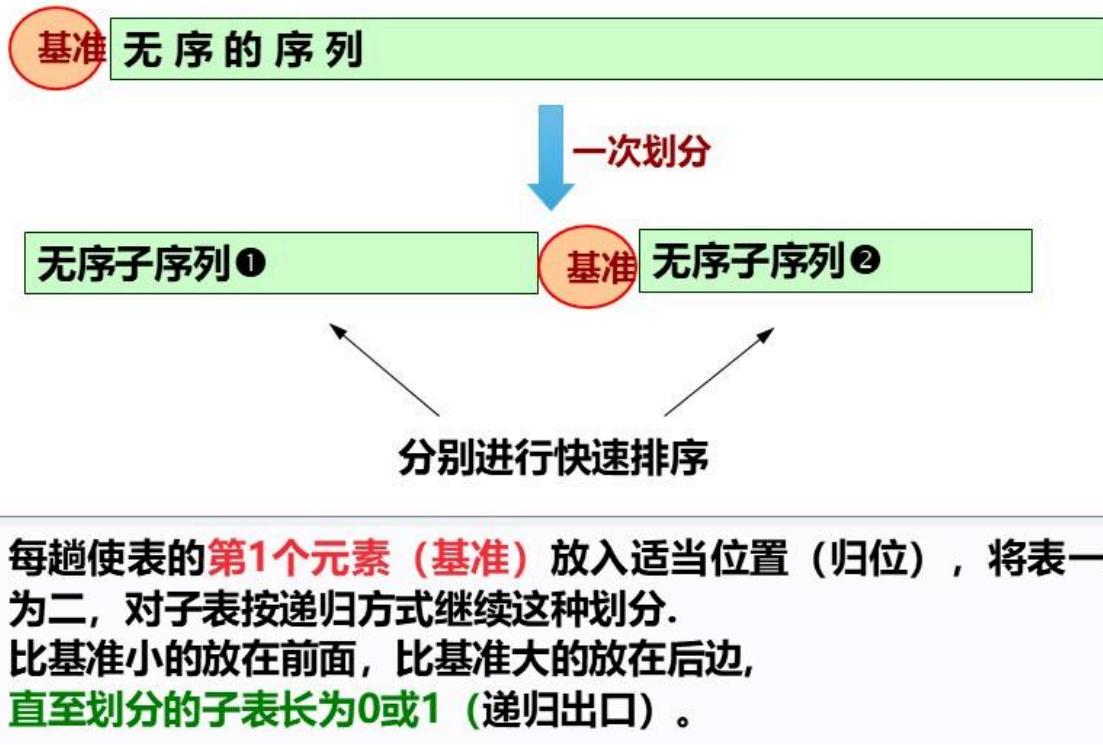
### 10.3.1 冒泡排序-算法

```
void BubbleSort(RecType R[], int n)
{ int i, j; bool exchange; RecType temp;
  for (i=0;i<n-1;i++)
  {
    exchange=false;
    for (j=n-1;j>i;j--)          //比较，找出最小关键字的元素
      if (R[j].key<R[j-1].key)   //反序⇒交换
      { swap(R[j],R[j-1]);
        exchange=true;
      }
    if (exchange==false) return; //中途结束算法
  }
}
```



## 10.3 交换排序

### 10.3.2 快速排序



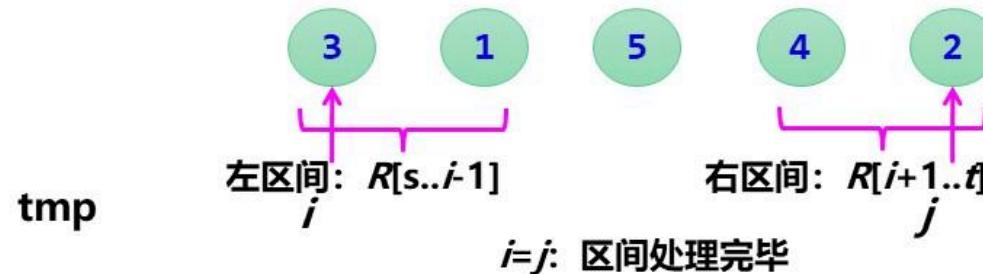
336/43

## 10.3 交换排序

### 10.3.2 快速排序

回顾划分：示例

整个区间： $R[s..t]$



划分完毕

- 令  $i$  为  $s$ ,  $j$  为  $t$
- 令  $j$  从最右侧  $t$  开始向左逐一与  $\text{tmp}$  比较，若比  $\text{tmp}$  小，将数据放置  $i$  处；然后  $i$  从  $i+1$  逐一与  $\text{tmp}$  比较，若比  $\text{tmp}$  大，将数据放置  $j$  处。
- 如此反复，直至  $i=j$ 。
- 比基准小的放在前面，比基准大的放在后边，
- 直至划分的子表长为 0 或 1 (递归出口)。

## 10.3 交换排序

### 10.3.2 快速排序-算法

```
int partition(RecType R[],int s,int t) //一趟划分
{
    int i=s,j=t;
    RecType tmp=R[i]; //以R[i]为基准
    while (i<j) //从两端交替向中间扫描,直至i=j为止
    {
        while (j>i && R[j].key>=tmp.key)
            j--; //从右向左扫描,找一个小于tmp.key的R[j]
        if(i<j)
        { R[i]=R[j]; //找到这样的R[j],放入R[i]处
            i++;
        }
        while (i<j && R[i].key<=tmp.key)
            i++; //从左向右扫描,找一个大于tmp.key的R[i]
        if(i<j)
        { R[j]=R[i]; //找到这样的R[i],放入R[j]处
            j--;
        }
    }
    R[i]=tmp;
    return i;
}
```

## 10.3 交换排序

### 10.3.2 快速排序-算法

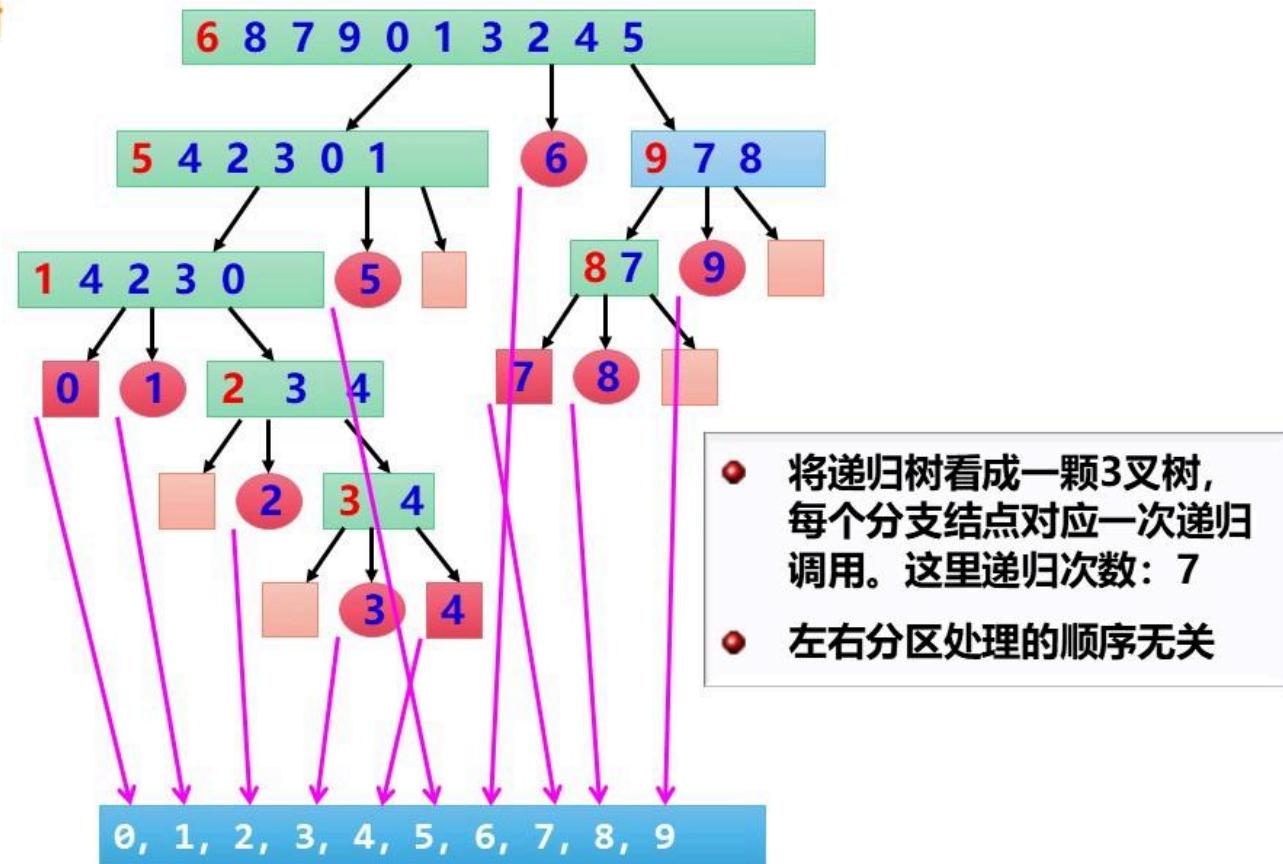


```
void QuickSort(RecType R[], int s, int t)
//对R[s..t]的元素进行快速排序
{ int i;
  if (s < t) //区间内至少存在两个元素的情况
  { i = partition(R, s, t);
    QuickSort(R, s, i-1); //对左区间递归排序
    QuickSort(R, i+1, t); //对右区间递归排序
  }
}
```

**【例10.4】** 设待排序的表有10个元素，其关键字分别为 (6, 8, 7, 9, 0, 1, 3, 2, 4, 5)。说明采用快速排序方法进行排序的过程。

## 10.3 交换排序

### 10.3.2 快速排序-递归树



## 10.3 交换排序

### 10.3.2 快速排序-递归树

示例

采用递归方式对顺序表进行快速排序，下列关于递归次数的叙述中，正确的是（ ）。

- A. 递归次数与初始数据的排列次序无关
- B. 每次划分后，先处理较长的分区可以减少递归次数
- C. 每次划分后，先处理较短的分区可以减少递归次数
- D. 递归次数与每次划分后得到的分区处理顺序无关

说明：本题为2010年全国考研题

## 10.3 交换排序

### 10.3.2 快速排序-递归树

示例

为实现快速排序法，待排序序列宜采用存储方式是（ ）。

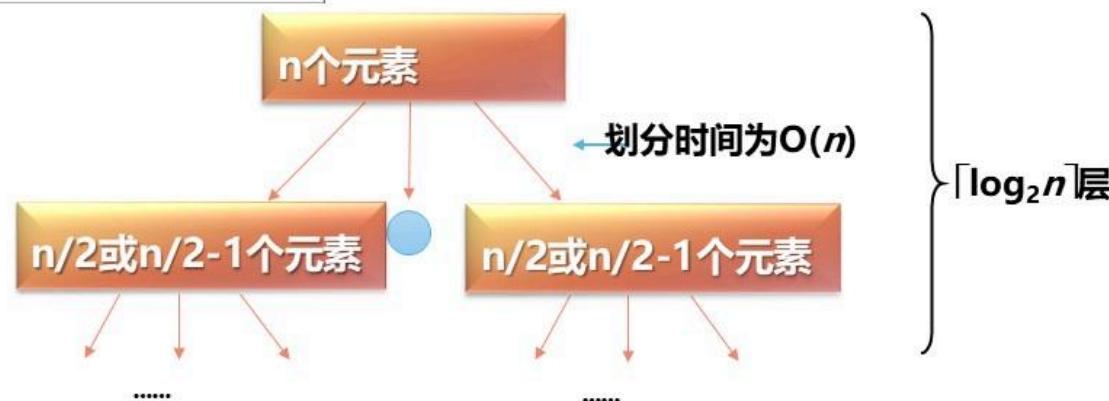
- A. 顺序存储
- B. 散列存储
- C. 链式存储
- D. 索引存储

说明：本题为2011年全国考研题

## 10.3 交换排序

### 10.3.2 快速排序-算法分析

■ 最好情况：

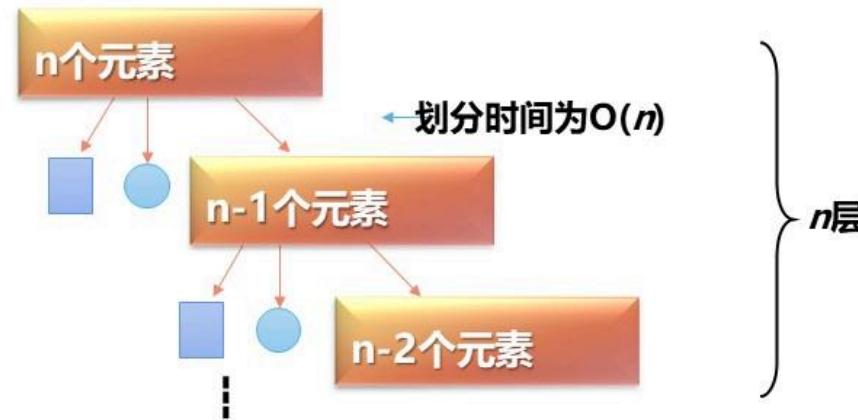


此时时间复杂度为  $O(n \log_2 n)$ , 空间复杂度为  $O(\log_2 n)$ 。

## 10.3 交换排序

### 10.3.2 快速排序-算法分析

■ 最坏情况：

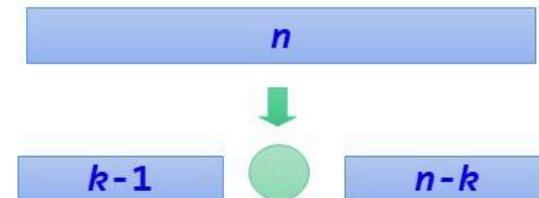


此时时间复杂度为 $O(n^2)$ , 空间复杂度为 $O(n)$ 。

平均情况：

$$T_{avg}(n) = Cn + \frac{1}{n} \sum_{k=1}^n [T_{avg}(k-1) + T_{avg}(n-k)]$$

↑  
1次划分的时间



则可得结果：

$$T_{avg}(n) = Cn \log_2 n.$$

**结论：**快速排序的时间复杂度为  $O(n \log_2 n)$

平均所需栈空间为  $O(\log_2 n)$ 。

## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



**常见的选择排序方法：**

- (1) 简单选择排序 (或称直接选择排序)
- (2) 堆排序



## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.4.1 简单选择排序

从 $a[i..n-1]$ 中选出最小元素：

```
int Min(int a[], int n, int i)
{ int k=i, j; //k保存最小元素的下标
  for (j=i+1;j<n;j++)
    if (a[j]<a[k]) k=j;
  return a[k];
}
```



简单选择  $n$ 个元素中找最小元素需要 $n-1$ 次比较

## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.4.1 简单选择排序



采用简单选择方法选出最小元素

全局有序区

无序区

- 初始时，全局有序区为空
- $i=0 \sim n-2$ ，共经过  $n-1$  趟排序



## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.4.1 简单选择排序-算法

```
void SelectSort(RecType R[], int n)
{ int i, j, k;
  for (i=0;i<n-1;i++)
    { k=i;
      for (j=i+1;j<n;j++)
        if (R[j].key<R[k].key)
          k=j;
      if (k!=i)           //R[i]↔R[k]
        swap(R[i],R[k]);
    }
}
```

## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.4.1 简单选择排序-算法

采用简单选择排序方法对(2, 1, 3, 4, 5) 进行排序

初始关键字



$i=0$



$i=1$



$i=2$



$i=3$



没有元素  
移动

任何情况下：都有做 $n-1$ 趟

## 10.4 选择排序



清华大学出版社  
TSINGHUA UNIVERSITY PRESS



### 10.4.1 简单选择排序-算法分析

- 从*n*个元素中挑选最小元素需要比较*i-1*次。
- 第*i* (*i=0 ~ n-2*) 趟从*n-i*元素中挑选最小元素需要比较*n-i-1*次。

对*n*个元素进行简单选择排序，所需进行的关键字的比较次数总计为：

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

- 移动元素的次数，正序为最小值 0，反序为最大值  $3(n-1)$ 。

简单选择排序的最好、最坏和平均时间复杂度为  $O(n^2)$ 。