



华南师范大学  
SOUTH CHINA NORMAL UNIVERSITY

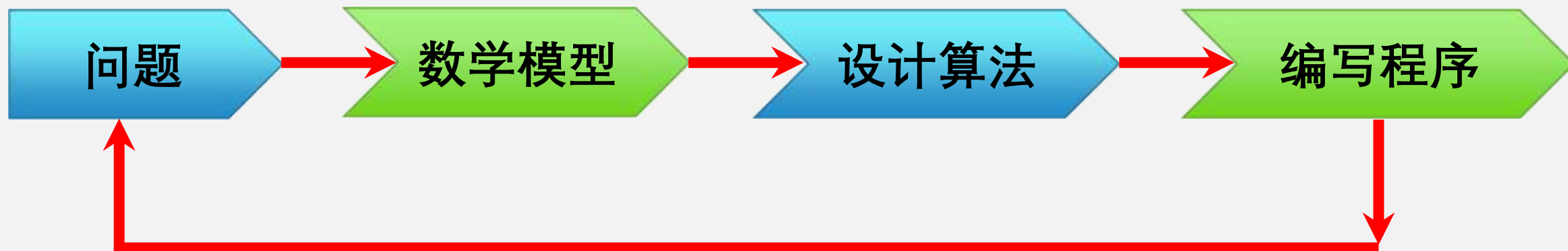
# 第3章：复杂数据的存储与管理



## 3.1 数据结构概述

## 数据结构的概念

### 计算机解决问题的一般步骤



## 数据结构

数据结构主要研究数据的组织、存储和运算。

### 数据

能被计算机程序处理的符号的集合。

### 组织

数据元素以及元素之间的关系。

### 存储

元素及其关系在计算机中的表示。

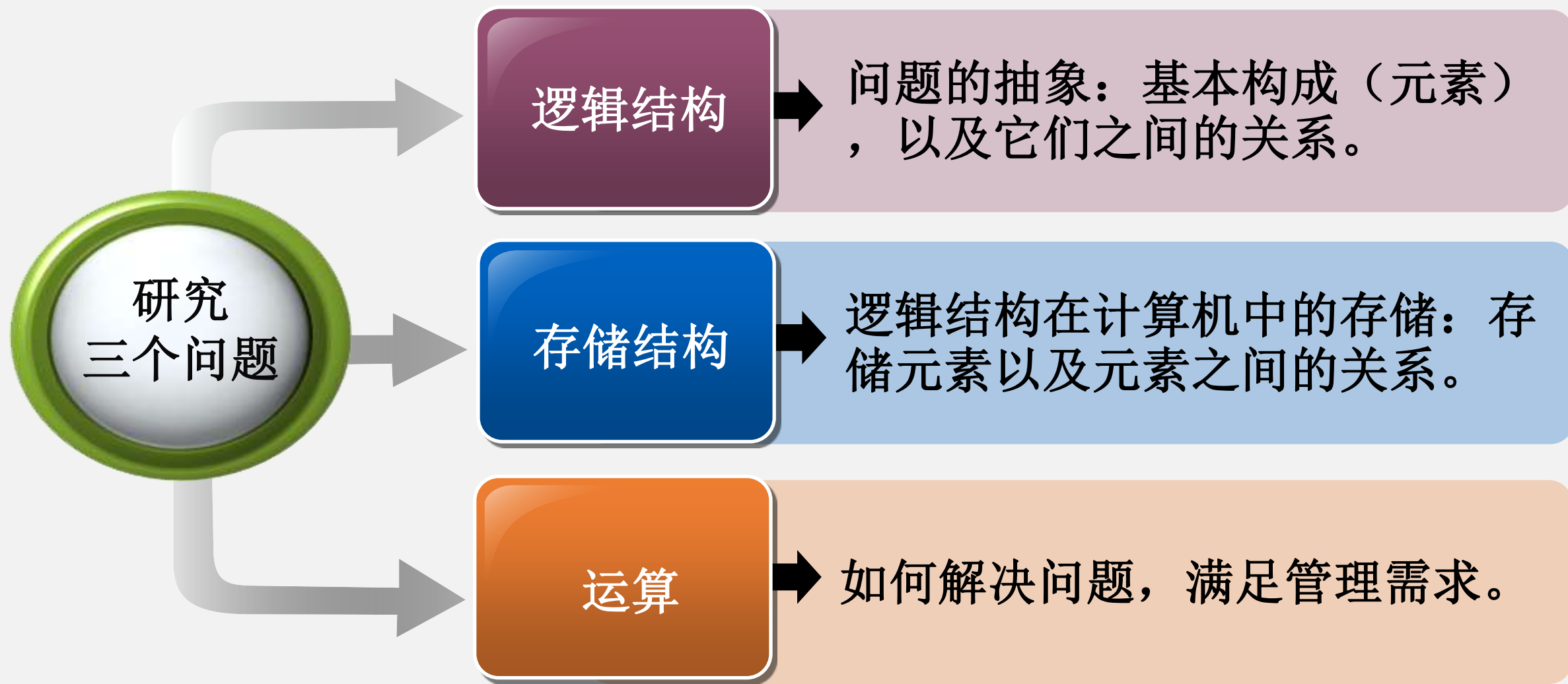
### 运算

对数据元素进行的操作处理（插入、删除、修改、查找、排序）。

## 数据结构的地位

- ◆ 计算机科学的核心内容之一。
- ◆ 介于数学、计算机硬件和计算机软件三者之间。
- ◆ 不仅是一般程序设计的基础，而且是设计和实现编译程序、操作系统、数据库系统及其他系统程序的重要基础。

## 数据结构的研究内容



## 逻辑结构

### 逻辑结构的概念

数据的逻辑结构是指反映数据元素之间逻辑关系的数据结构。

数据的逻辑结构包含：

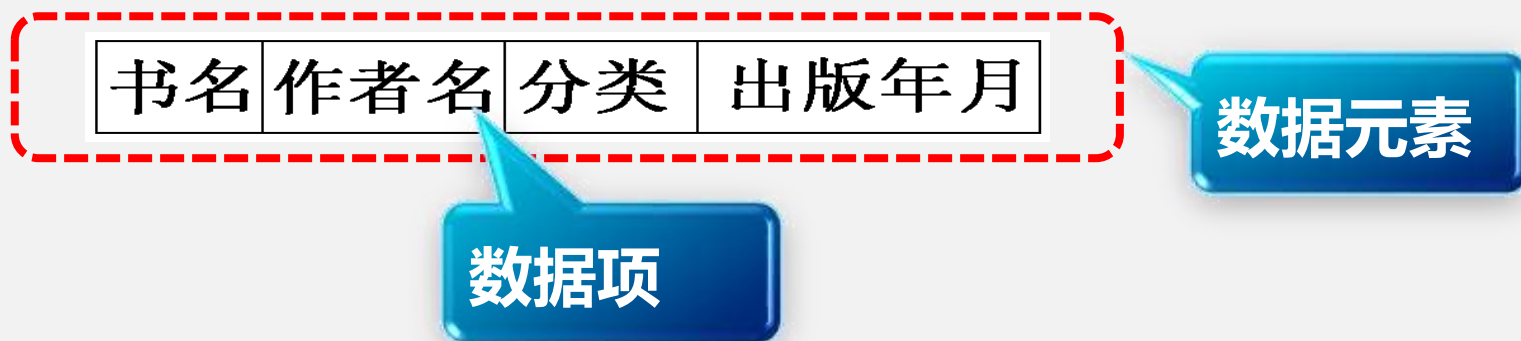
- (1) 表示数据元素的信息；
- (2) 表示各数据元素之间的关系。

**数据元素**是数据的基本单位，即数据集合中的个体。

有时一个数据元素可由若干**数据项** (Data Item) 组成。

**数据项**是数据的最小单位。

数据元素亦称结点或记录。





## 逻辑数据结构的描述

数据结构可描述为：  $\text{Group} = (D, R)$

D

→ 有限个数据元素的集合。

R

→ 数据元素之间关系的集合。

例：

## 1. 一年四季的数据结构

$$B=(D,R)$$

$$D=\{\text{春}, \text{夏}, \text{秋}, \text{冬}\}$$

$$R=\{(\text{春}, \text{夏}), (\text{夏}, \text{秋}), (\text{秋}, \text{冬})\}$$

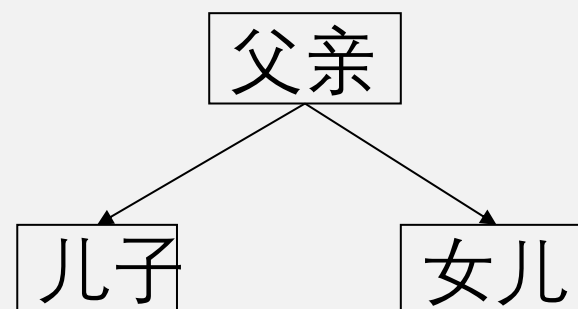
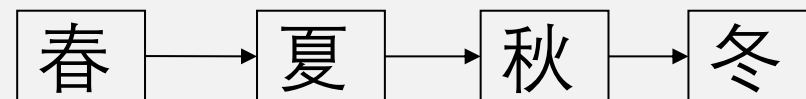
## 2. 家庭成员的数据结构

$$B=(D,R)$$

$$D=\{\text{父亲}, \text{儿子}, \text{女儿}\}$$

$$R=\{(\text{父亲}, \text{儿子}), (\text{父亲}, \text{女儿})\}$$

## 数据结构的图形表示



## 常见的逻辑结构

### (1) 线性结构

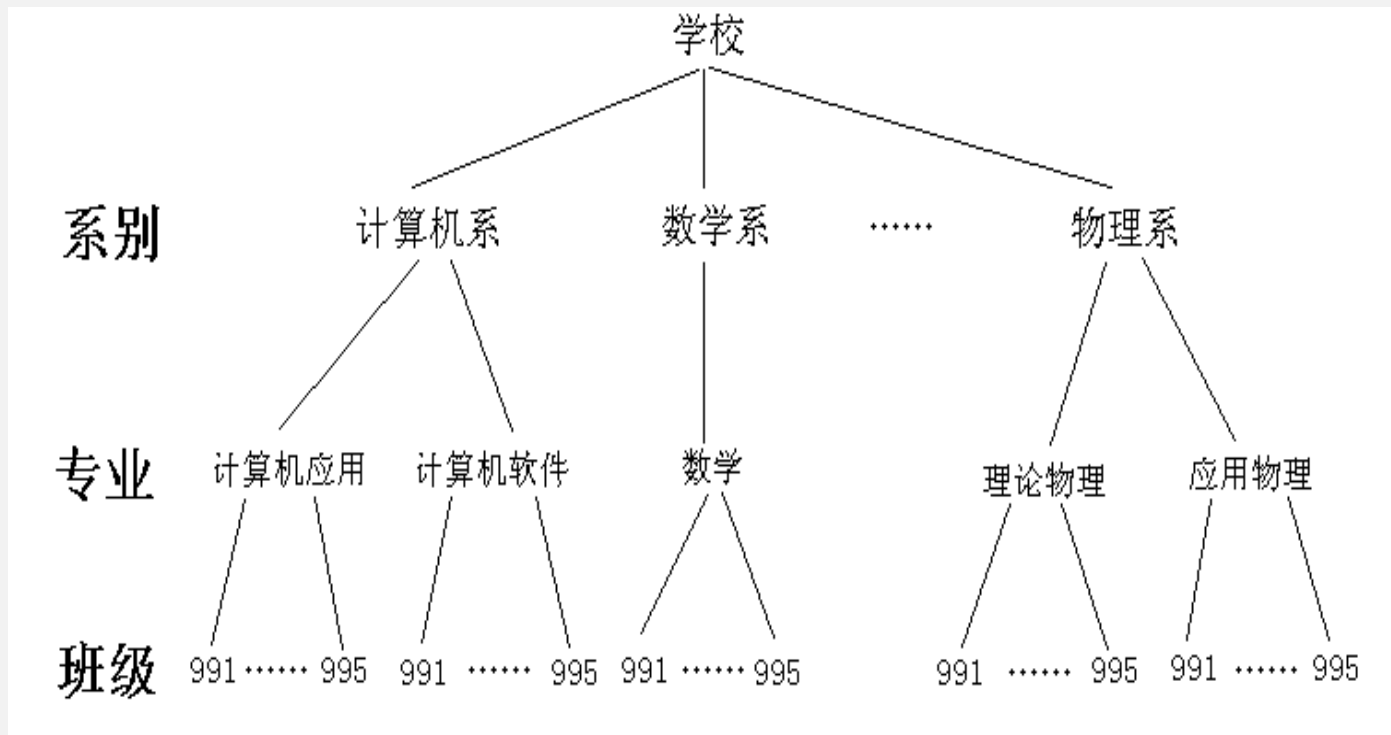
结点间是线性关系。

A, B, C, …… , X, Y, Z

学 生 成 绩 表

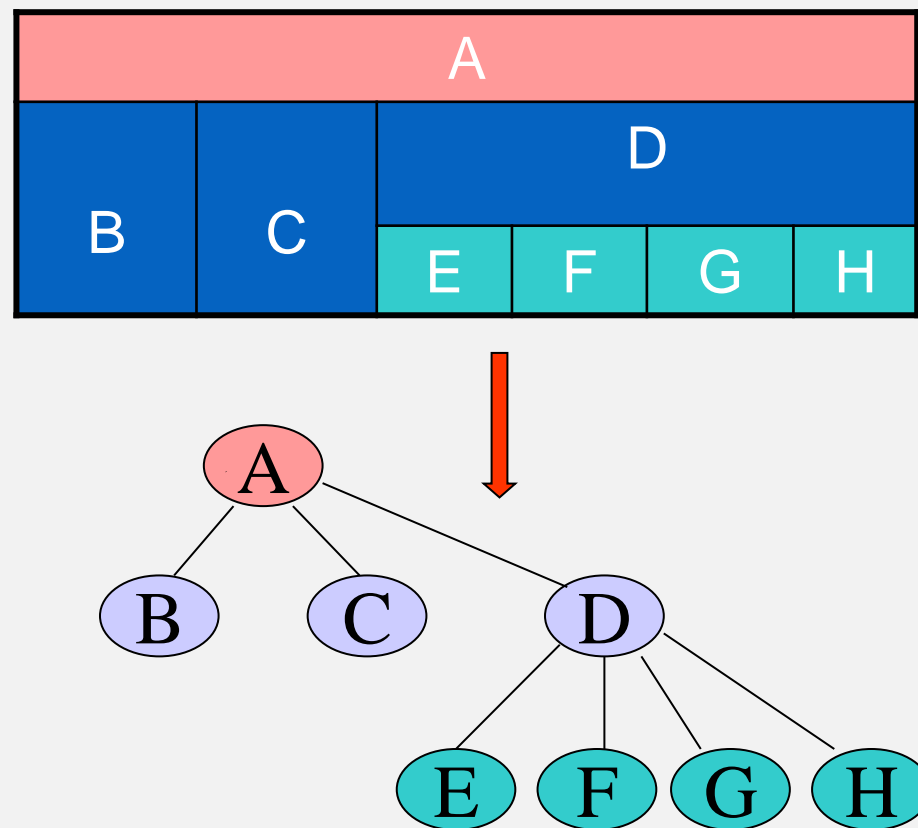
学号	姓名	成绩
2018161109	张卓	87
2018161107	刘雨涵	95
2018161103	胡敏	86

## (2) 树形结构

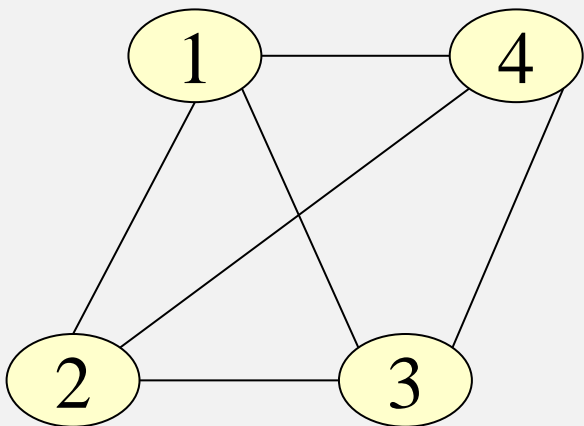


全校学生档案管理的组织方式

树形结构 —— 结点间具有分层关系

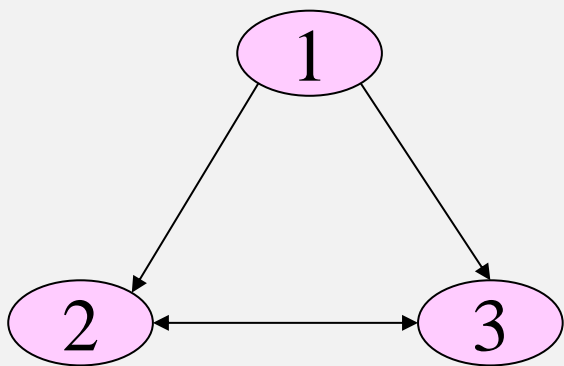


### (3) 图形结构



$$D = \{ 1, 2, 3, 4 \}$$

$$R = \{ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) \}$$



$$D = \{ 1, 2, 3 \}$$

$$R = \{ \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle, \langle 1,3 \rangle \}$$

## 存储结构

### 存储结构的概念

存储结构指逻辑结构在计算机存储空间中的具体实现。

- ◆ 存储所有元素；存储元素之间的关系
- ◆ 一种逻辑结构可以有多种存储结构。

## 常见的存储结构



一种逻辑结构：  
可以表示成一种或  
多种存储结构。

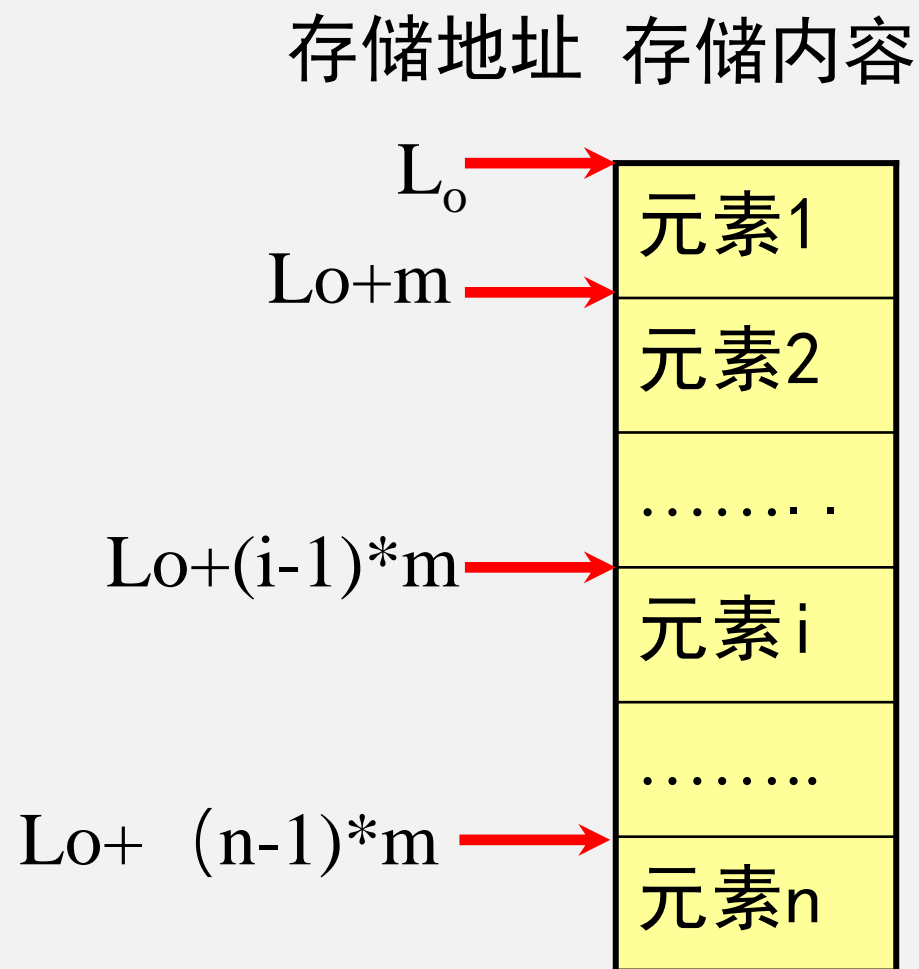


## 3.2 顺序存储与链式存储



## 顺序存储

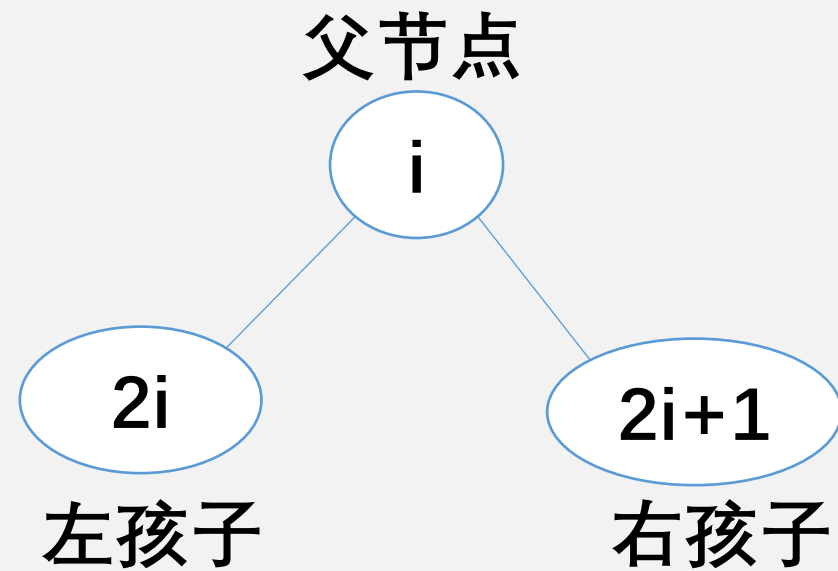
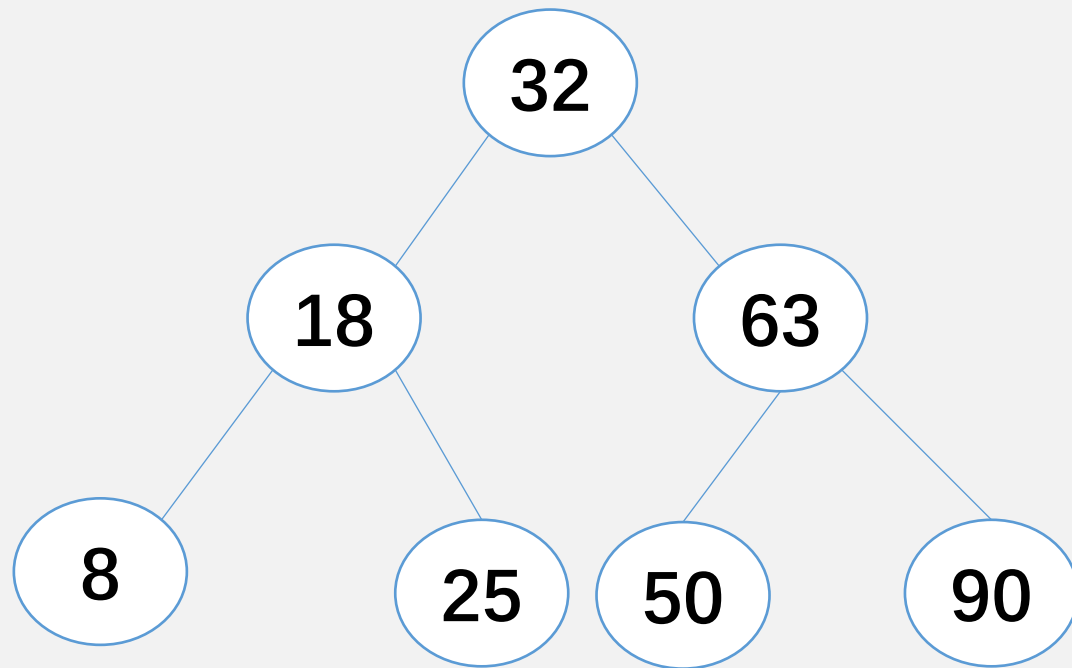
- 采用连续存储空间存储数据元素；
- 元素之间的关系通过存储位置的关系来表示。



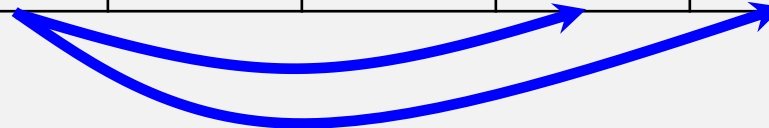
有关关键字序列： 32    75    70    63    48    94    25    36    18

a[0]   a[1]   a[2]   a[3]   a[4]   a[5]   a[6]   a[7]   a[8]   a[9]

32	75	70	63	48	94	25	36	18	
----	----	----	----	----	----	----	----	----	--



$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$
7	32	18	63	8	25	50	90		



优点：易于定位

$$\text{Loc}(a) = L_0 + (i-1) * m$$

每个元素所占用的  
存储单元个数

不足之处：

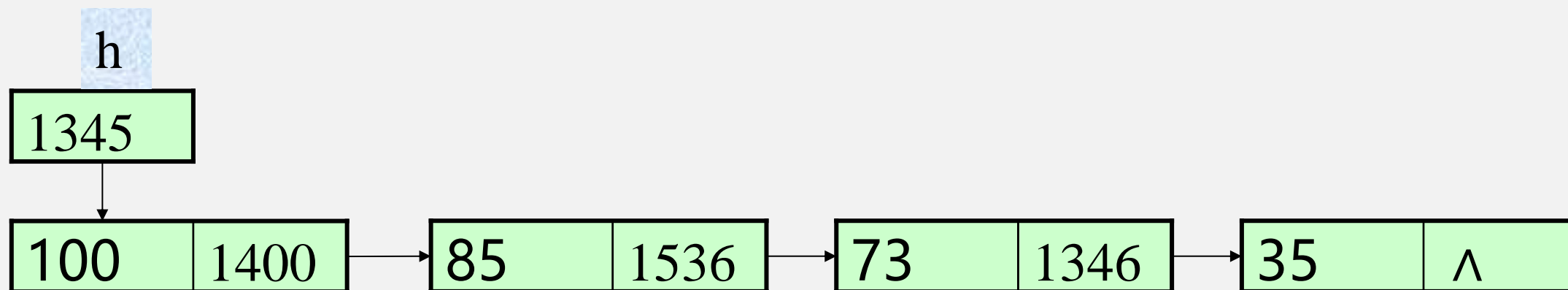
- 作插入或删除操作时，需移动大量元素。
- 长度变化较大时，需按最大空间分配。
- 容量难以扩充。

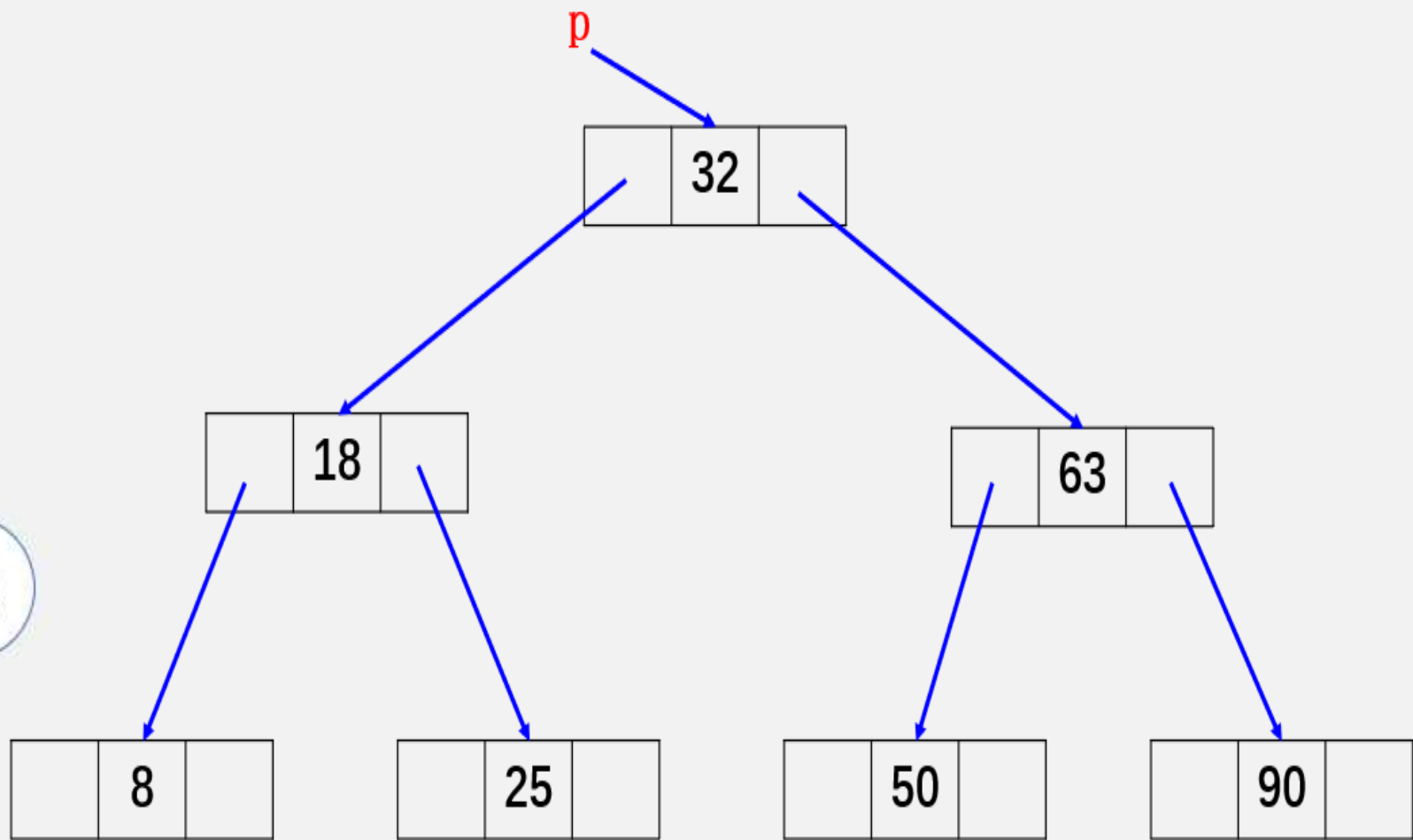
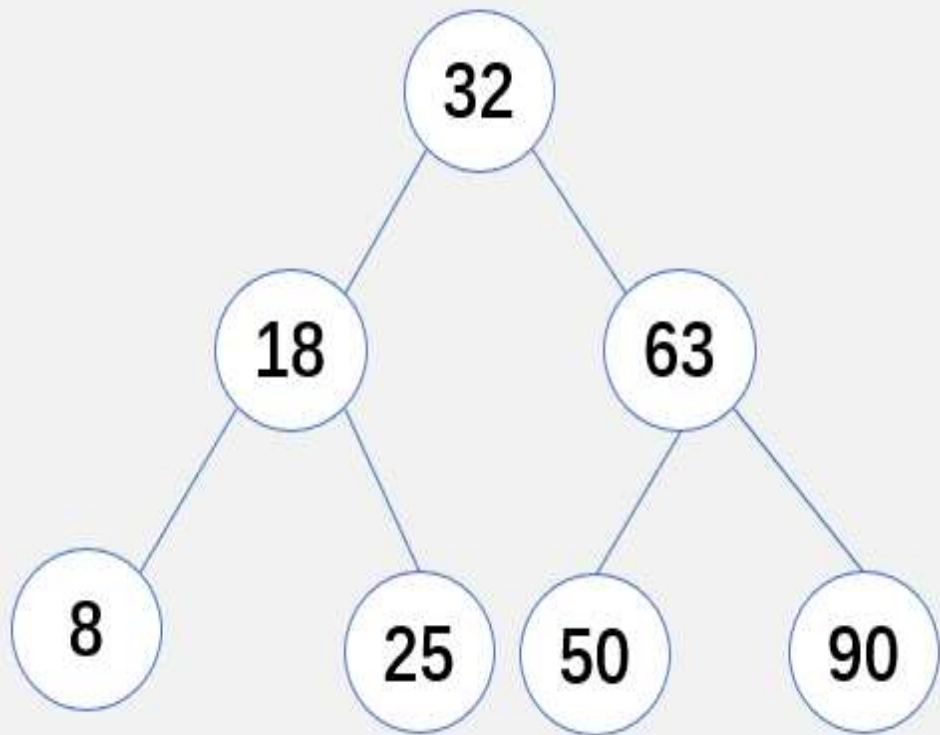
## 链式存储

每个结点都由两部分组成：**数据域**和**指针域**。

**数据域**存放元素本身的数据，**指针域**存放下一个结点的地址。

数据元素之间逻辑上的联系由指针来体现。





## 链接存储结构特点:

- ◆ 逻辑上相邻的节点物理上不必相邻。
- ◆ 插入、删除灵活（不必移动节点，只要改变结点中的指针）。
- ◆ 不能直接定位，必须从头向后遍历。



## 3.3 索引存储与散列存储



## 索引方式

存储结点信息的同时，建立附加的索引表。

索引表中的**每一项**称为一个索引项，索引项的一般形式：

关键字

地址

关键字：能唯一标识一个结点的那些数据项。

地址：同值关键字结点的起始位置。

稠密索引：每个结点在索引表中都有一个索引项。

稀疏索引：**一组结点**在索引表中只对应一个索引项

姓氏	地址
安	
白	
⋮	⋮

姓名	电话号码
安云	
安名	
⋮	
安华	
白玉明	
⋮	
白春雨	

## 散列方式

分配连续存储空间，元素的存储位置由元素的关键字计算得到。

有关键字序列： 32    75    70    63    48    94    25    36    18

存储空间大小： 11

地址计算规则：  $L = K \text{ MOD } 11$

冲突解决机制：  $L+1$

计算存储位置：

K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	?				

A	0	1	2	3	4	5	6	7	8	9	10
K					70				63	75	32




K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5				

A	0	1	2	3	4	5	6	7	8	9	10
K					70	48			63	75	32




K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5	6	3	?	

A	0	1	2	3	4	5	6	7	8	9	10
K				25	70	48	94		63	75	32



K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5	6	3	7	

A	0	1	2	3	4	5	6	7	8	9	10
K				25	70	48	94	36	63	75	32



K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5	6	3	7	?

A	0	1	2	3	4	5	6	7	8	9	10
K				25	70	48	94	36	63	75	32



K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5	6	3	7	0

A	0	1	2	3	4	5	6	7	8	9	10
K	18			25	70	48	94	36	63	75	32

A	0	1	2	3	4	5	6	7	8	9	10
K	18			25	70	48	94	36	63	75	32

K	32	75	70	63	48	94	25	36	18
L	10	9	4	8	4	6	3	3	7
A	10	9	4	8	5	6	3	7	0
N	1	1	1	1	2	1	1	5	5



华南师范大学  
SOUTH CHINA NORMAL UNIVERSITY

## 3.4 算法

## 算法概念

算法是在有限步骤内求解某一问题所使用的一组定义明确的规则。

- ◆ 算法是解题的过程。
- ◆ 形成解题思路(推理实现的算法)，编写程序(操作实现的算法)，都是在实施某种算法。

**算法是程序设计的核心**

**算法以存储结构为基础**

## 算法特征

确定性

每一步骤必须有确切的定义

有输入

有0个或多个输入，0输入是指算法本身设定了初始条件

有输出

有一个或多个输出，没有输出的算法毫无意义

有穷性

保证执行有限步之后结束

可行性

能够在有限时间、有限空间内完成

## 算法设计的要求

正确性 能够正确的解决问题。

- ①不含语法错误；
- ②对于几组输入数据，结果正确；
- ③对于精心选择的几组数据，结果正确；
- ④对一切合法的输入数据，结果正确。

### 可读性

应容易供人阅读和交流。可读性好的算法有助于对算法的理解和修改。

### 健壮性

应具有容错处理。当输入非法或错误数据时，算法应能适当地作出反应或进行处理

### 通用性

应具有一般性。即算法的处理结果对于一般的数据集合都成立。

## 算法描述

算法的表示需要使用一些语言形式。

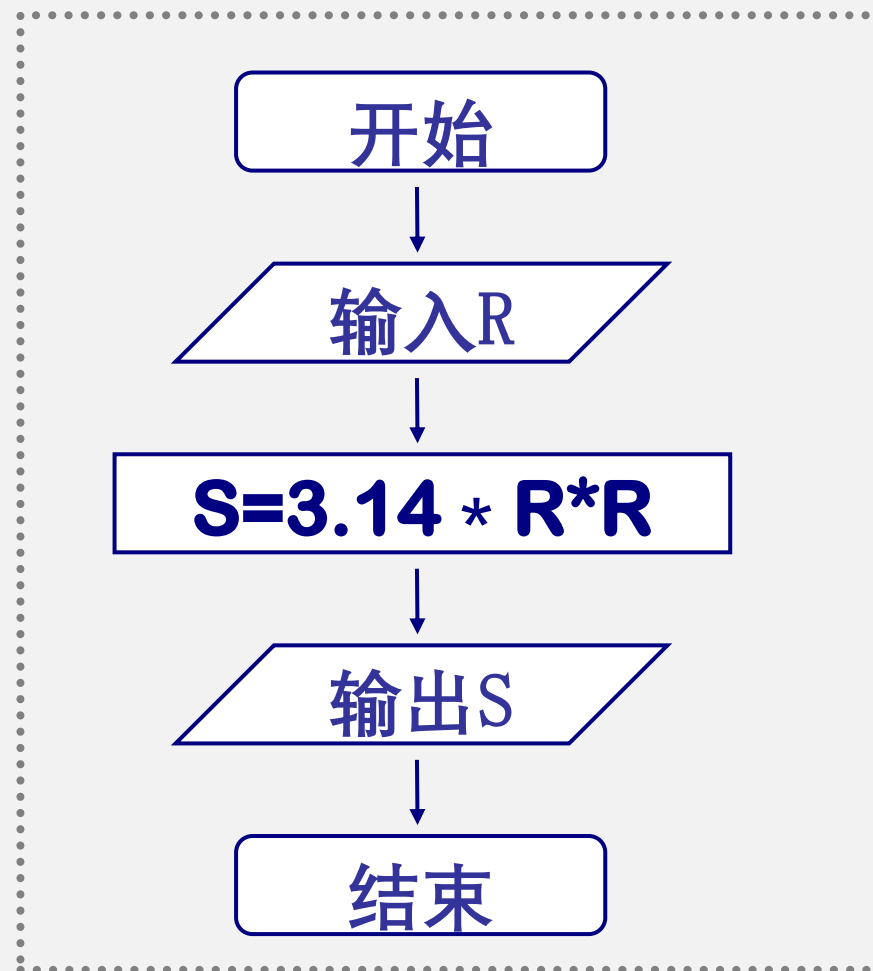
传统描述-----图形法：“流程图”和**N-S**图。

常用描述-----使用伪码描述算法。



问题：输入圆的半径，计算圆的面积。

```
INPUT  r
S=3.14 * r*r
PRINT S
```



## 算法性能分析

### 时间复杂度

执行过程中所需**基本运算的执行次数**来度量。

### 空间复杂度

执行过程中**临时占用的存储空间**来度量。

## 时间复杂度

- ◆ 基本语句执行次数，而非时间。
- ◆ 计算最坏情况。

常数阶 $O(1)$

对数阶 $O(\log_2 n)$

线性阶 $O(n)$

线性对数阶 $O(n \log_2 n)$

k次方阶 $O(n^k)$

指数阶 $O(2^n)$

随着问题规模 $n$ 的不断增大，  
时间复杂度不断增大。

```
int x=1,s=0;  
while (x <10)  
{  
    S+=X; X++;  
}
```

执行10次，时间复杂度表示是 $O(1)$ 。

```
int i,j,s=1;  
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
        { s=s+i*j }  
}
```

时间复杂度是 $O(n^2)$

## 空间复杂度

运行过程中临时占用存储空间大小的量度。

计算方法：

①忽略常数，用 $O(1)$ 表示

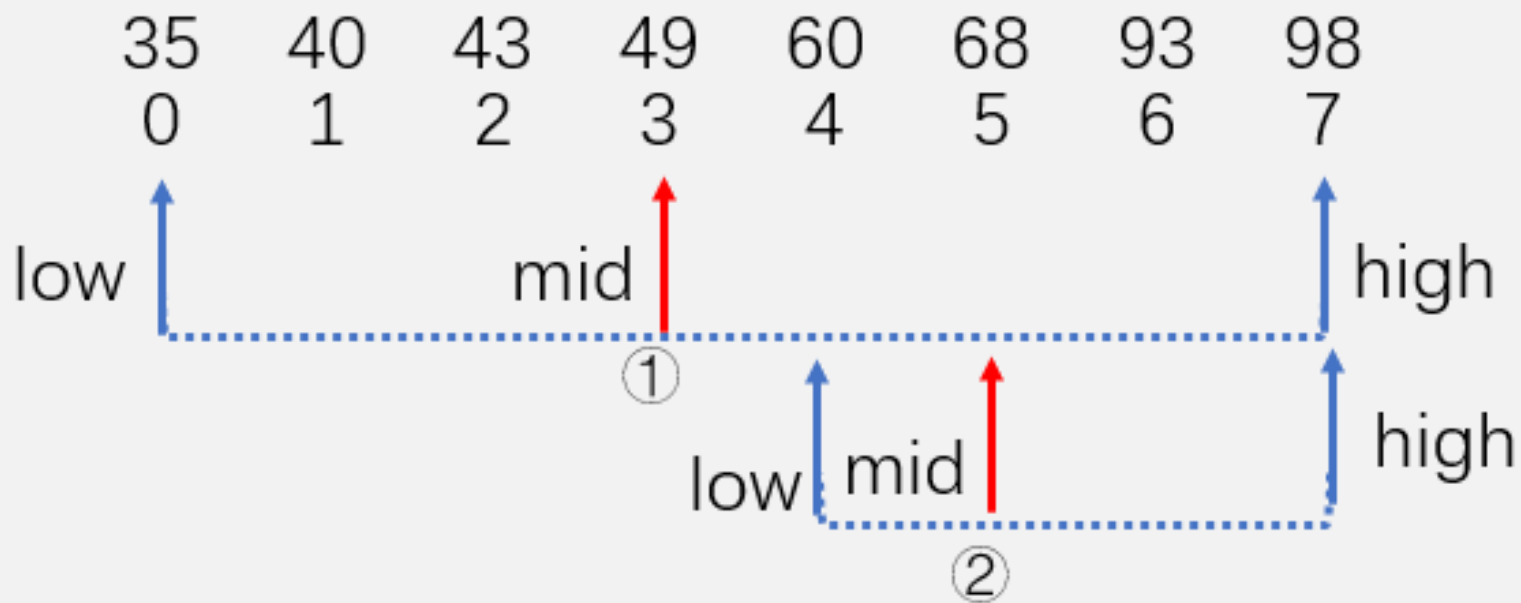
②递归算法的空间复杂度=递归深度 $N$ \*每次递归所要的辅助空间

```
int fun(int n)
{
    int k=10;
    if (n==k)
        return n;
    else
        return fun(++n);
}
```

递归实现，调用fun函数每次都创建1个变量k。  
调用n次，空间复杂度 $O(n \times 1) = O(n)$ 。

## 二分查找的时间复杂度及空间复杂度

以查找68为例：





## 二分查找的时间复杂度及空间复杂度

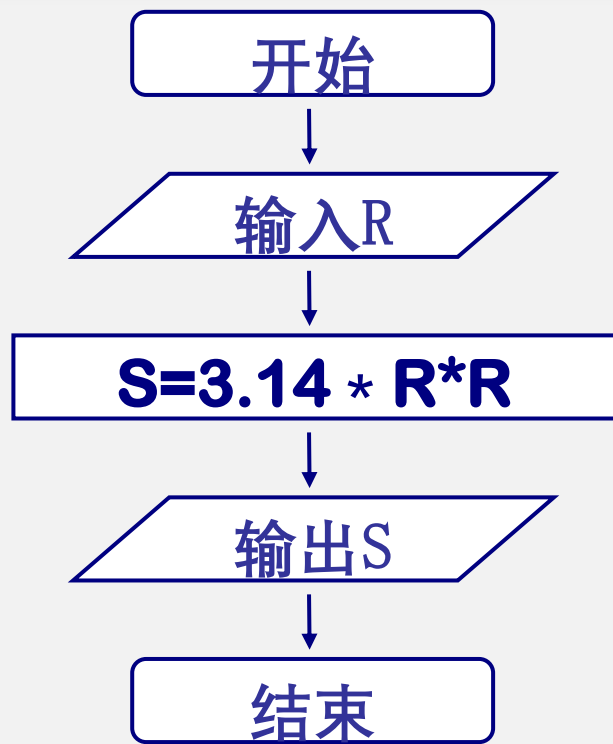
**非递归：**每次都对原查找内容进行二分，所以时间复杂度为 $O(\log_2 n)$ 。  
变量值创建一次，所以空间复杂度为 $O(1)$ 。

**递归：**时间复杂度为 $O(\log_2 n)$ 。  
每进行一次递归都会创建变量，所以空间复杂度为 $O(\log_2 n)$ 。

## 算法和程序的区别

算法：是一组逻辑步骤。

程序：用计算机语言描述的算法。



```
main()
{
    float r,s;
    scanf( "%f" ,&r);
    if(r<0)
        printf( "err,r<0" );
    else
        {s=r*r*3.14;
         printf( "s=%f" ,s);
        }
}
```

## 3.5 线性表

## 线性结构

一个非空的数据结构若满足下面的几个条件：

- ①有且仅有一个根结点；
- ②除第一个结点外，每一个结点最多有一个直接前驱结点；
- ③除最后一个结点外，每一个结点最多有一个直接后继结点。

**线性表、栈和队列都是线性结构；树、图、网属于非线性结构。**

## 线性表的概念

由 $n$  ( $n \geq 0$ ) 个数据元素 $a_1, a_2, \dots, a_i, \dots, a_n$ 组成的一个有限序列。

20	18	32	30	76	78	90	65	70	34
----	----	----	----	----	----	----	----	----	----

## 线性表的特点：

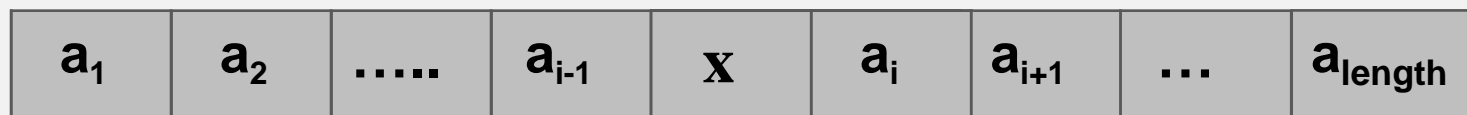
- (1) 线性表中所有元素的性质相同。
- (2) 除第一个和最后一个数据元素之外，其它数据元素有且仅有一个前驱和一个后继。第一个数据元素无前驱，最后一个数据元素无后继。
- (3) 数据元素在表中的位置只取决于它自身的序号。

## 线性表上常用的操作有：

初始化、求长度、取元素、修改、插入、删除、检索、排序。

## 线性表的存储结构

◆ 顺序存储结构



◆ 链式存储结构



## 线性表的顺序存储

分配连续存储空间，依次存储所有元素。

• 线性表的顺序存储结构称为顺序表

◆ 只存储结点的值，不存储结点间的关系；

◆ 逻辑上相邻的数据元素存储在物理上相邻的存储单元里；

◆ 结点间的关系由存储单元的邻接关系来体现。



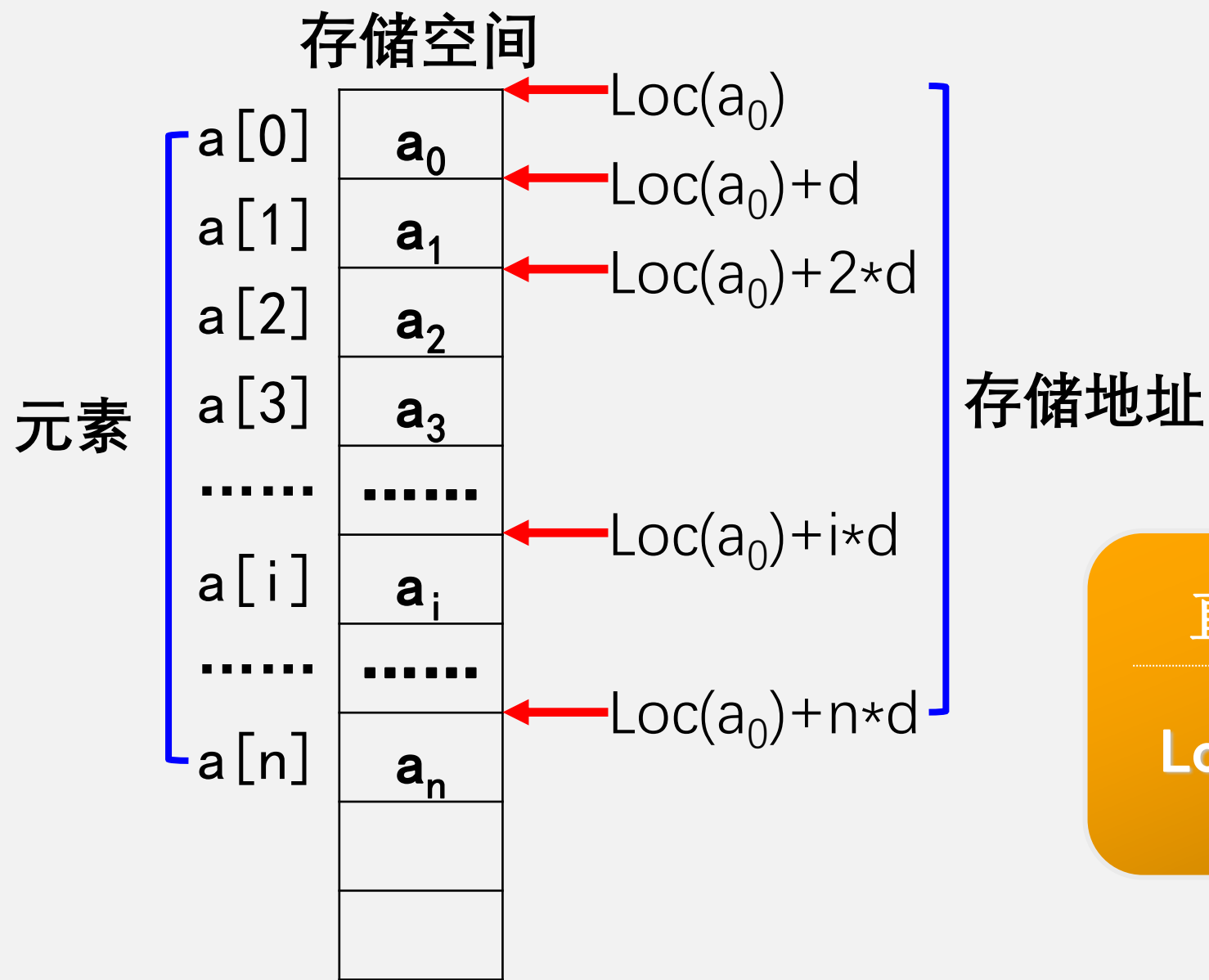
**(1) 数组中的元素间的地址是连续的;**

**(2) 数组中所有元素的数据类型相同。**

```
int a[10];
```

系统将在内存中分配连续的40个字节（VC中整数占4个字节），存放10个整数。

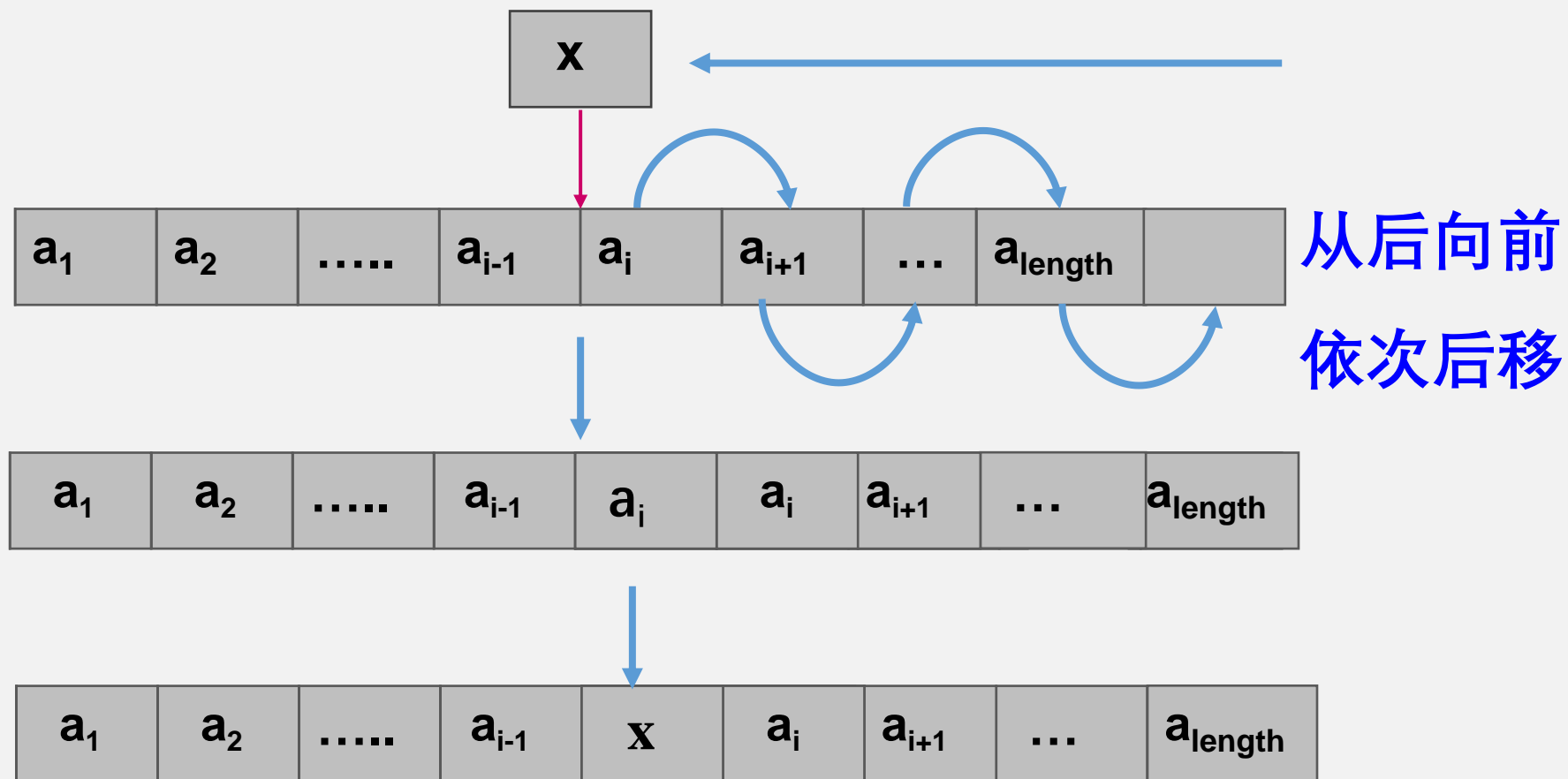
[illegible]



直接定位，实现随机存取

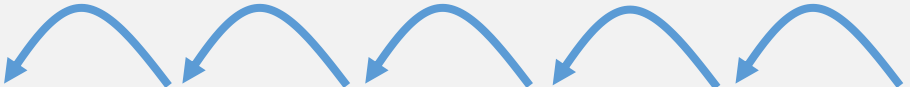
$$\text{Loc}(a_i) = \text{Loc}(a_0) + i*d$$

## 插入运算



## 删除运算

从前向后  
依次前移



4	17	15	28	30	32	42	51	63	
---	----	----	----	----	----	----	----	----	--

4	17	15	30	32	42	51	63	63	
---	----	----	----	----	----	----	----	----	--

队尾元素

## 插入算法性能分析

假设线性表中含有 $n$ 个数据元素，在进行插入操作时，若假定在 $n+1$ 个位置上插入元素的可能性均等，则平均移动元素的个数为：

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

时间复杂度为 $O(n)$ 。

## 删除算法的分析

在进行删除操作时，若假定删除每个元素的可能性均等，则平均移动元素的个数为：

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

时间复杂度为 $O(n)$ 。

## 顺序存储总结

- ◆表中数据元素类型一致，只有数据域，存储空间利用率高；
- ◆各数据元素在存储空间中按逻辑顺序依次存放，占有连续存储空间，可以直接定位；
- ◆做插入、删除时需移动大量元素，若表长为 $n$ ，则插入算法的时间复杂度都为 $O(n)$ 。
- ◆空间估计不清楚时，应按最大空间分配。



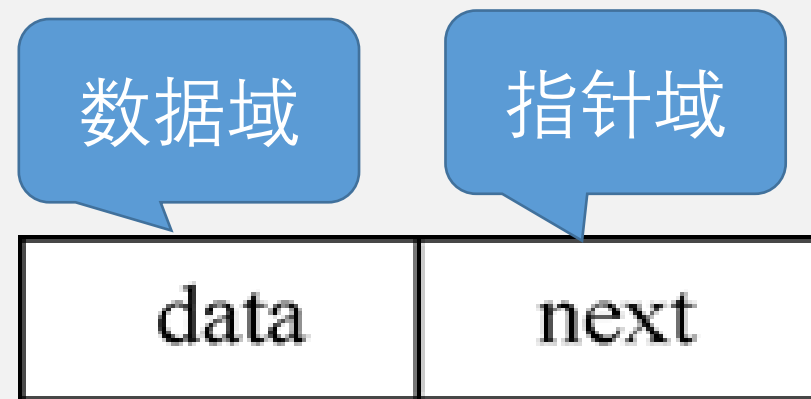
## 3.6 线性表的链式存储

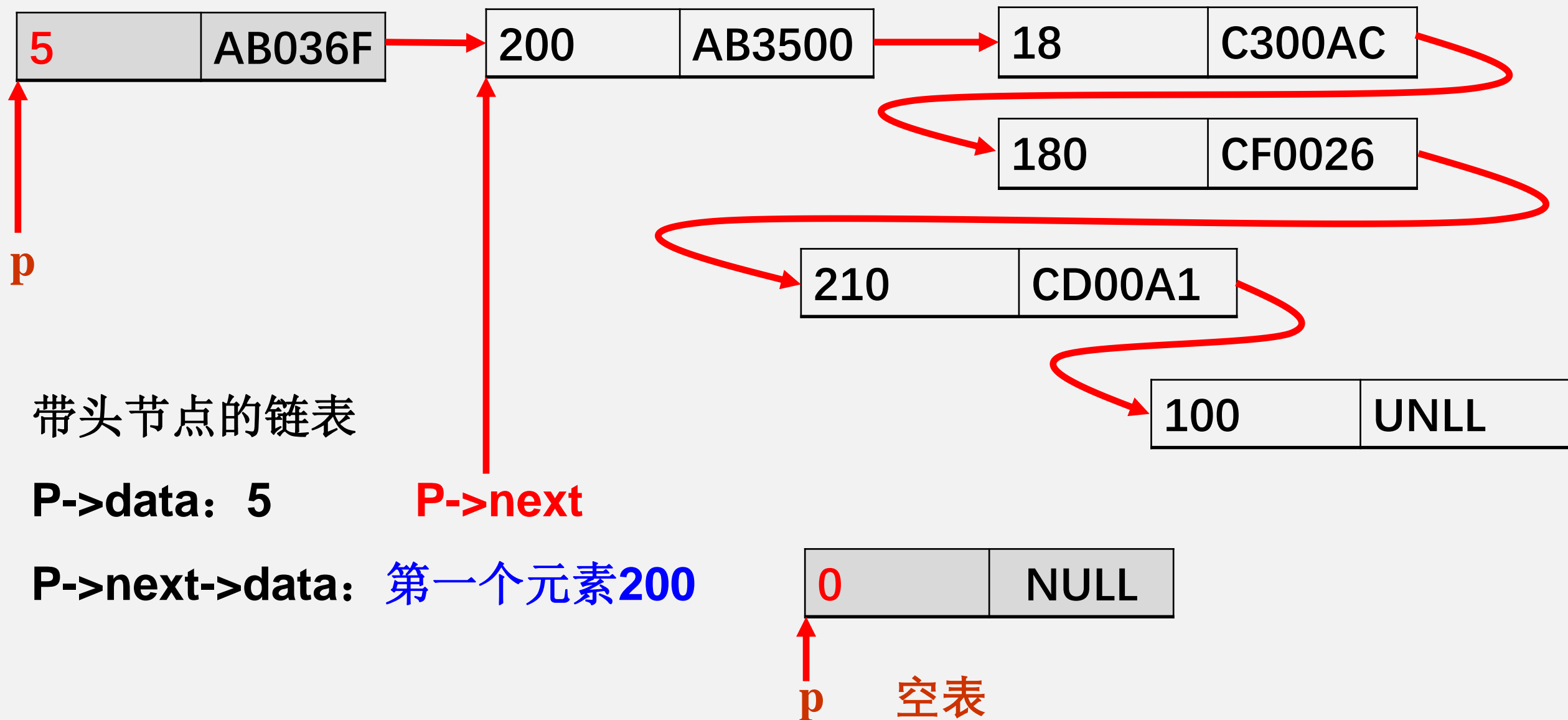


## 链式存储

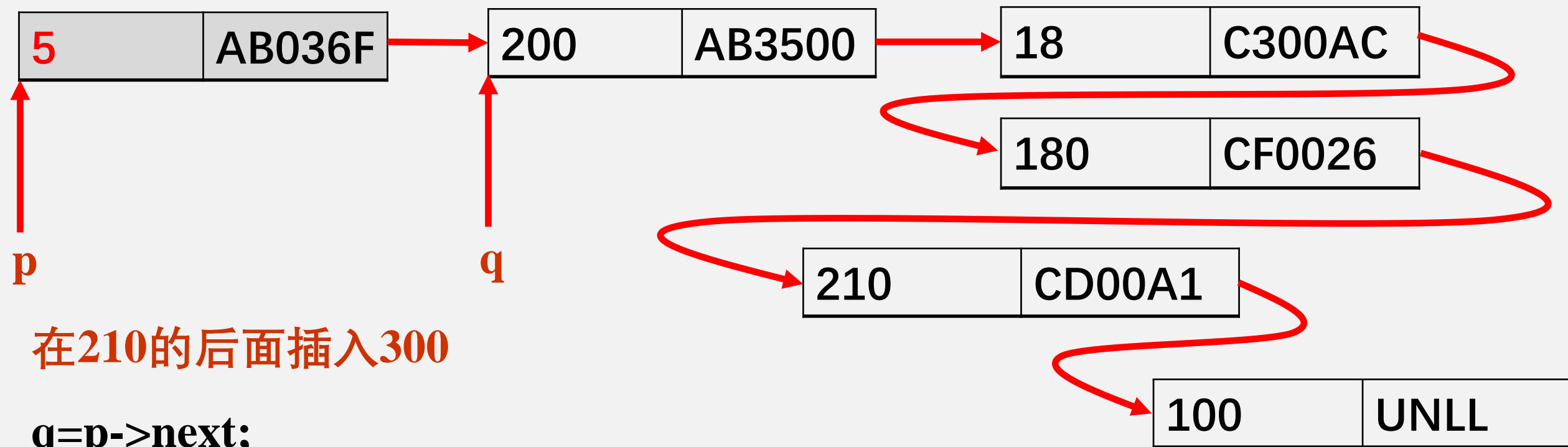
• 线性表的链式存储结构称为线性链表

- ◆ 数据存在结点中，不要求逻辑上相邻的数据元素物理位置也相邻；
- ◆ 各数据元素的存储顺序任意；
- ◆ 数据元素的先后关系是由结点的指针域指示。





## 插入运算

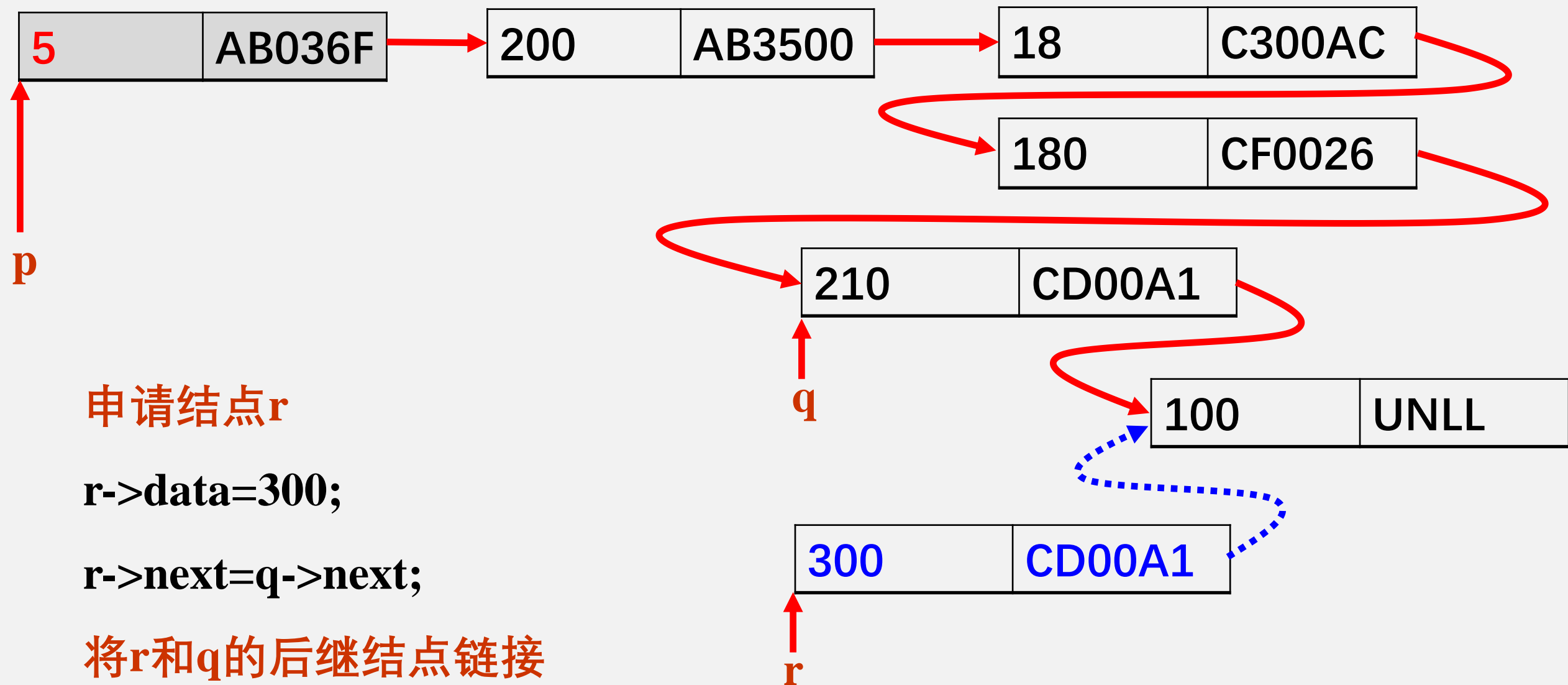


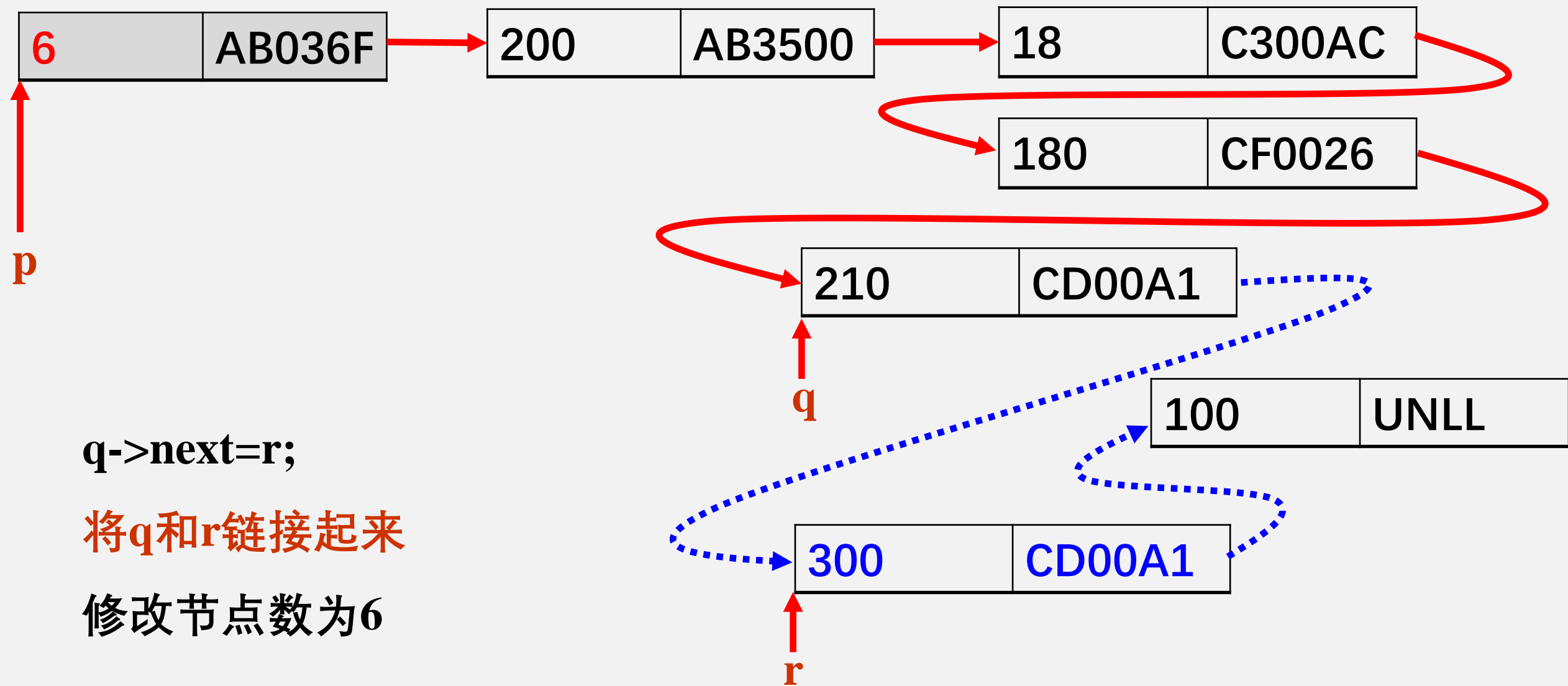
在210的后面插入300

`q=p->next;`

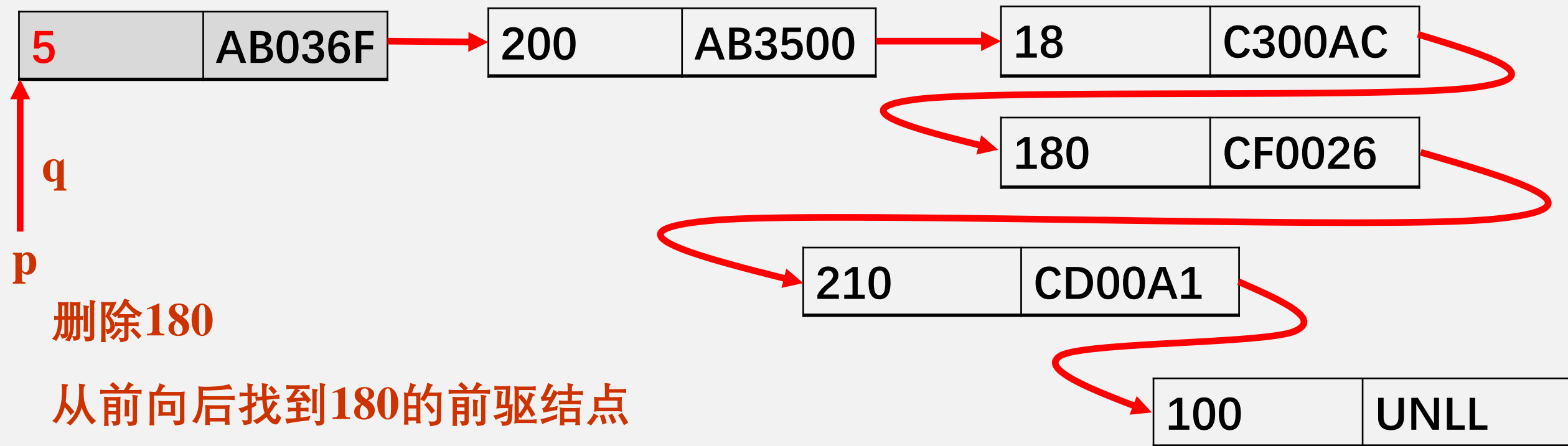
从q开始向后遍历，找210

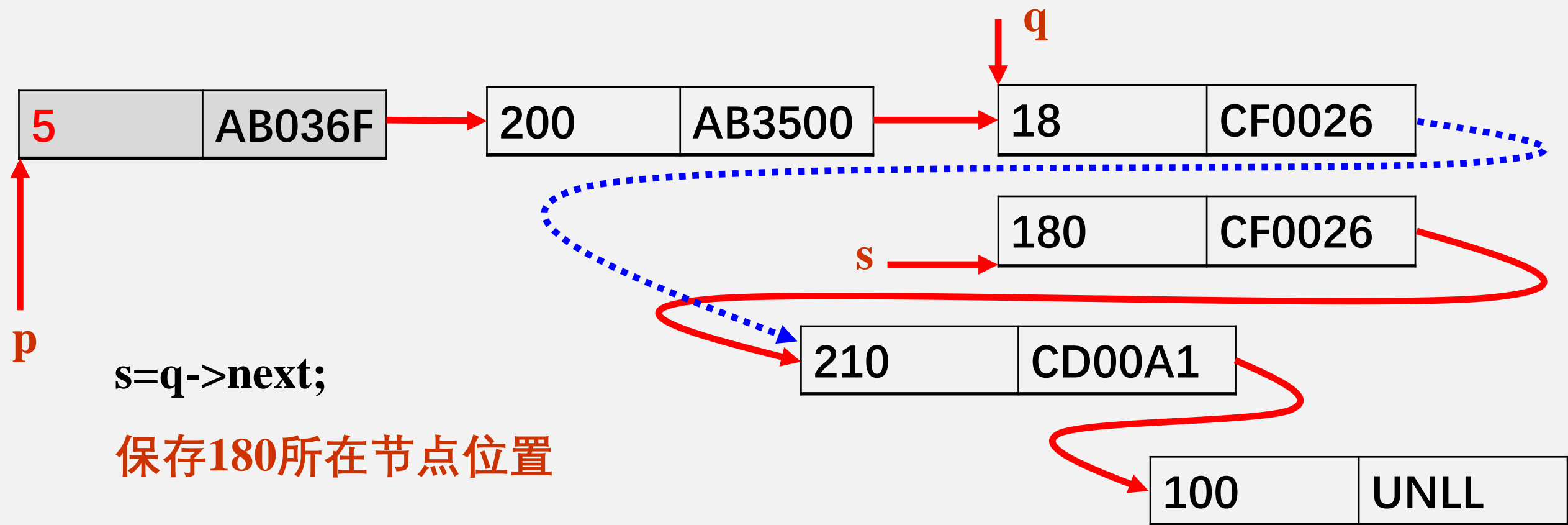
若`q->data!=210`，则 `q=q->next;`





## 删除运算



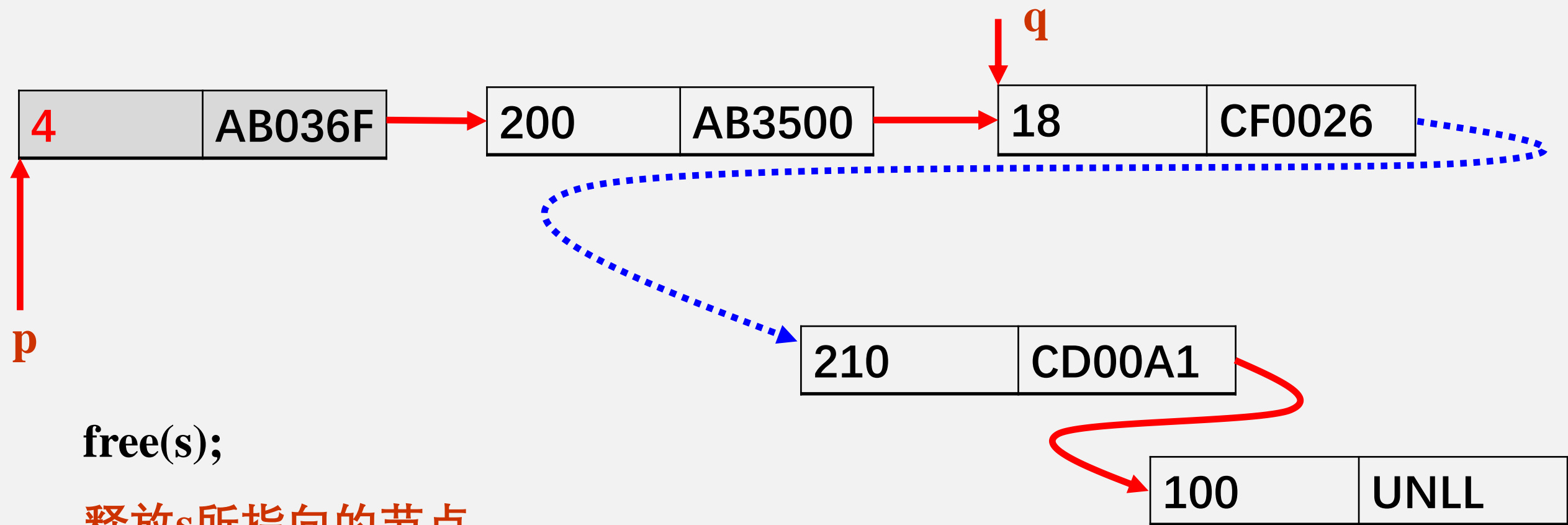


$s = q \rightarrow \text{next};$

保存180所在节点位置

$q \rightarrow \text{next} = s \rightarrow \text{next};$

将S从链中删除



`free(s);`

释放s所指向的节点

修改节点数为4

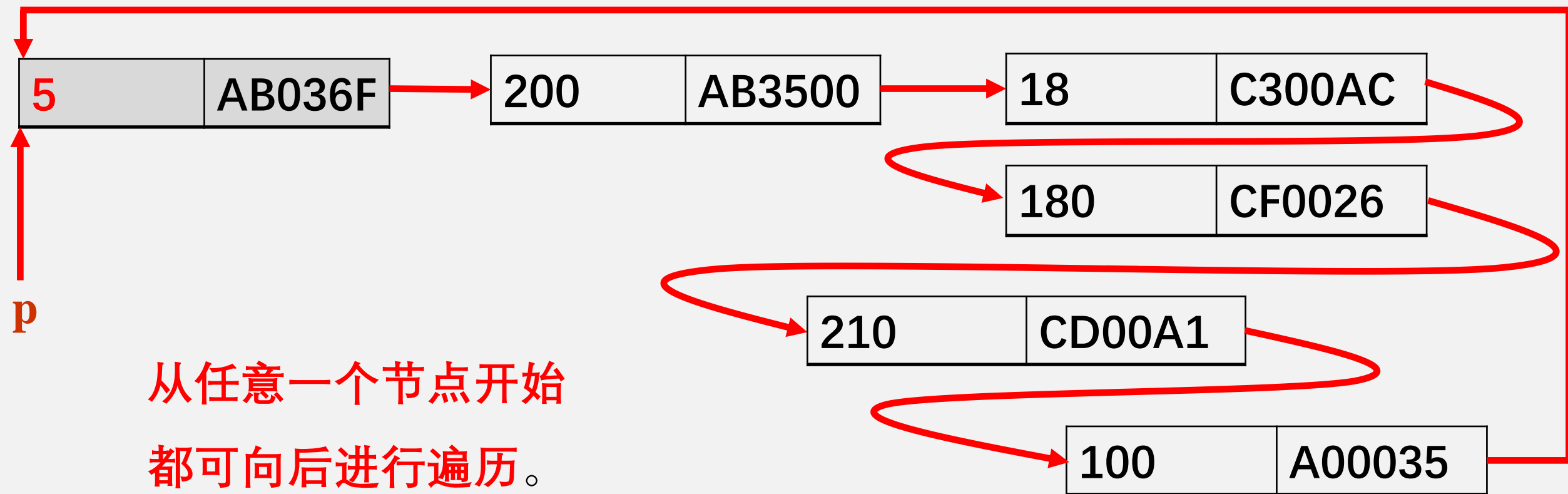


## 链式存储总结

- ◆表中数据元素类型一致，结点由数据域和指针域构成；
- ◆按需申请结点，结点空间不连续；
- ◆不可直接定位，需从头遍历；
- ◆便于插入、删除。

## 单向链表的改进

循环链表：最后一个节点的指针域指向头结点。

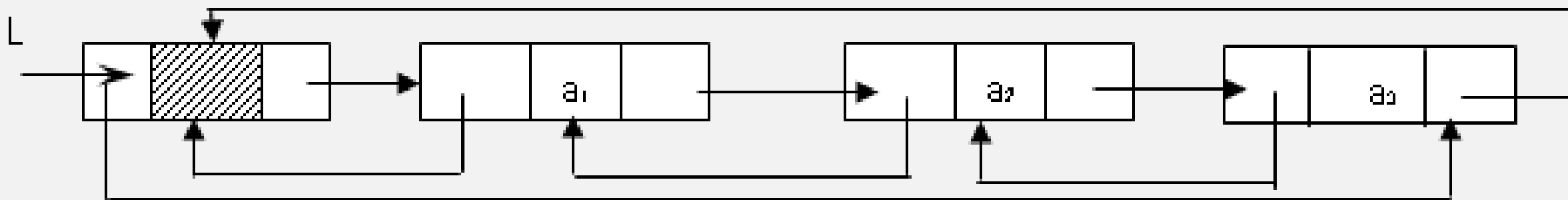


双向链表：可以克服单链表的单向性的缺点。

在双向链表的结点中有两个指针域：

其一指向直接后继，另一指向直接前趋。

prior	data	next
-------	------	------



从任意一节点开始，既可向前( $q \rightarrow \text{prior}$ )，也可向后( $q \rightarrow \text{next}$ )进行遍历。



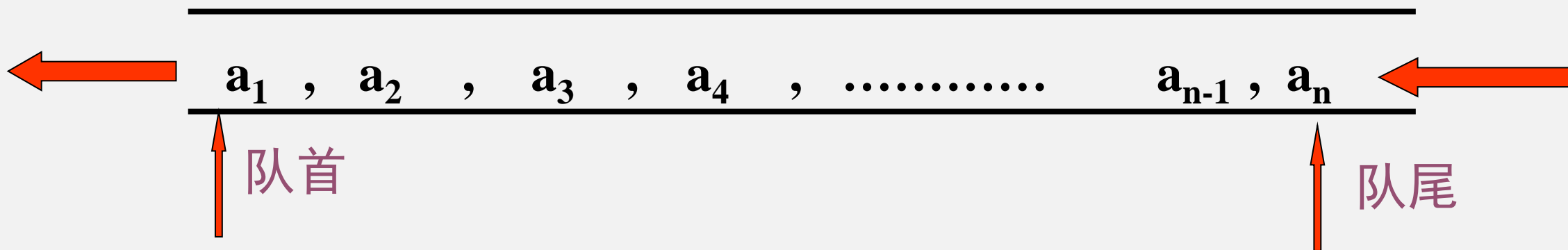
## 3.7 队列的存储与处理

## 队的概念

队列是操作位置受限的线性表

只能在表的一端插入，另一端进行删除的线性表。

此种结构称为**先进先出表**。

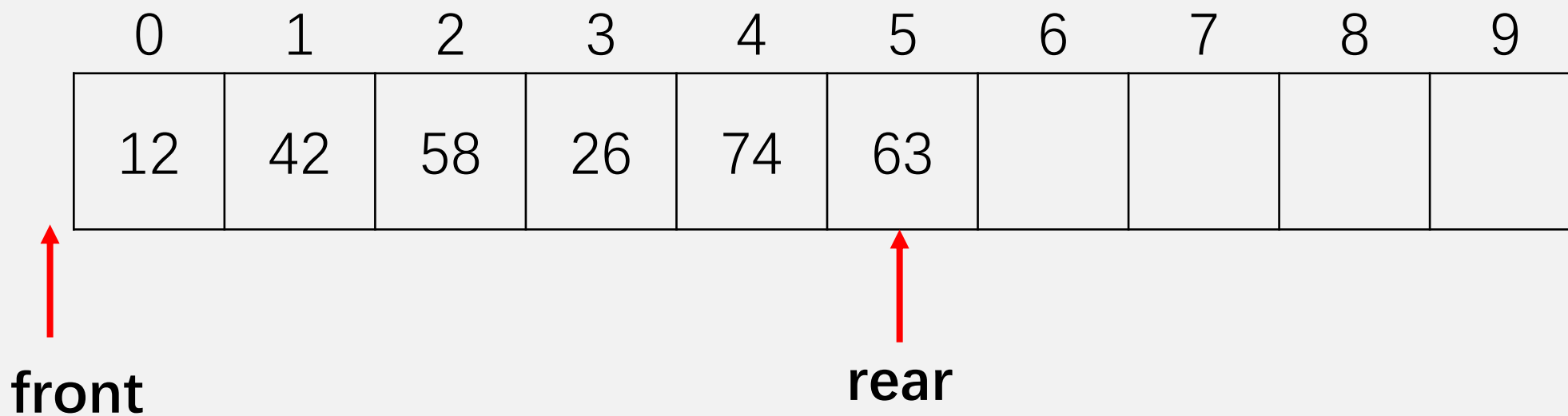


## 队的基本操作

- (1) 设置一个空队列；
- (2) 插入一个新的元素（队尾），称为入队；
- (3) 删除队头元素（队首），称为出队；
- (4) 读取队头元素；

## 队的顺序存储

用一组地址连续的存储单元依次存放从队列头到队列尾的元素，非空队列指针front和rear分别指示队头元素前一个位置和队尾元素的位置。

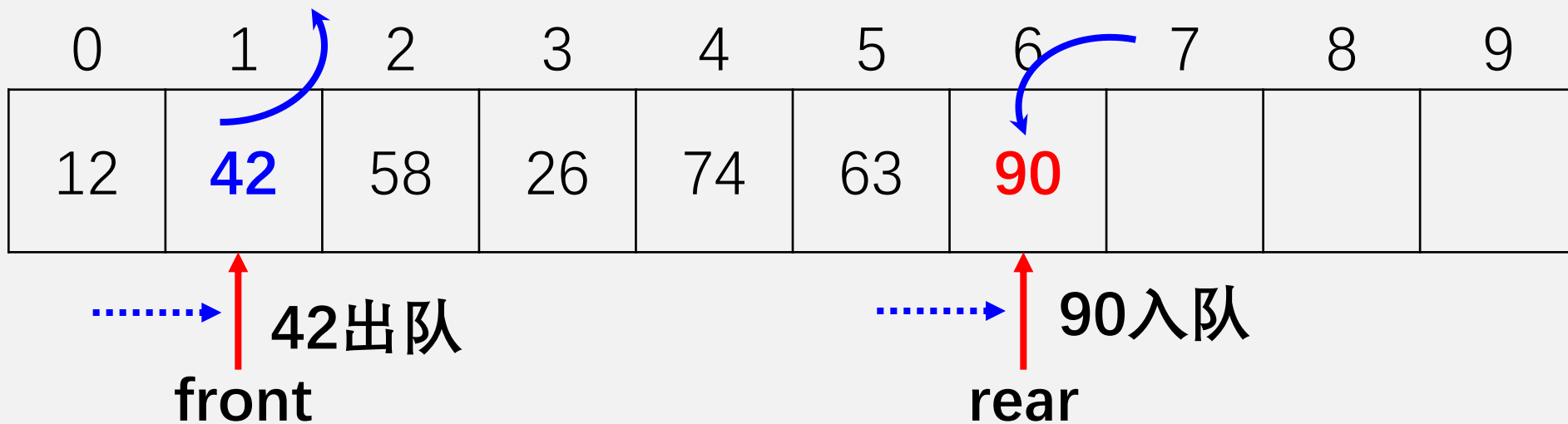


队空：令 $\text{rear}=\text{front}=-1$ ；

元素入队： $\text{rear}=\text{rear}+1$ ； $\text{a}[\text{rear}]=\text{m}$ ；

元素出队： $\text{front}=\text{front}+1$ ； $\text{m}=\text{a}[\text{front}]$ ；

故在非空队列中，**头指针**始终指向**队头元素前一个位置**，而**尾指针**始终指向**队尾元素的位置**





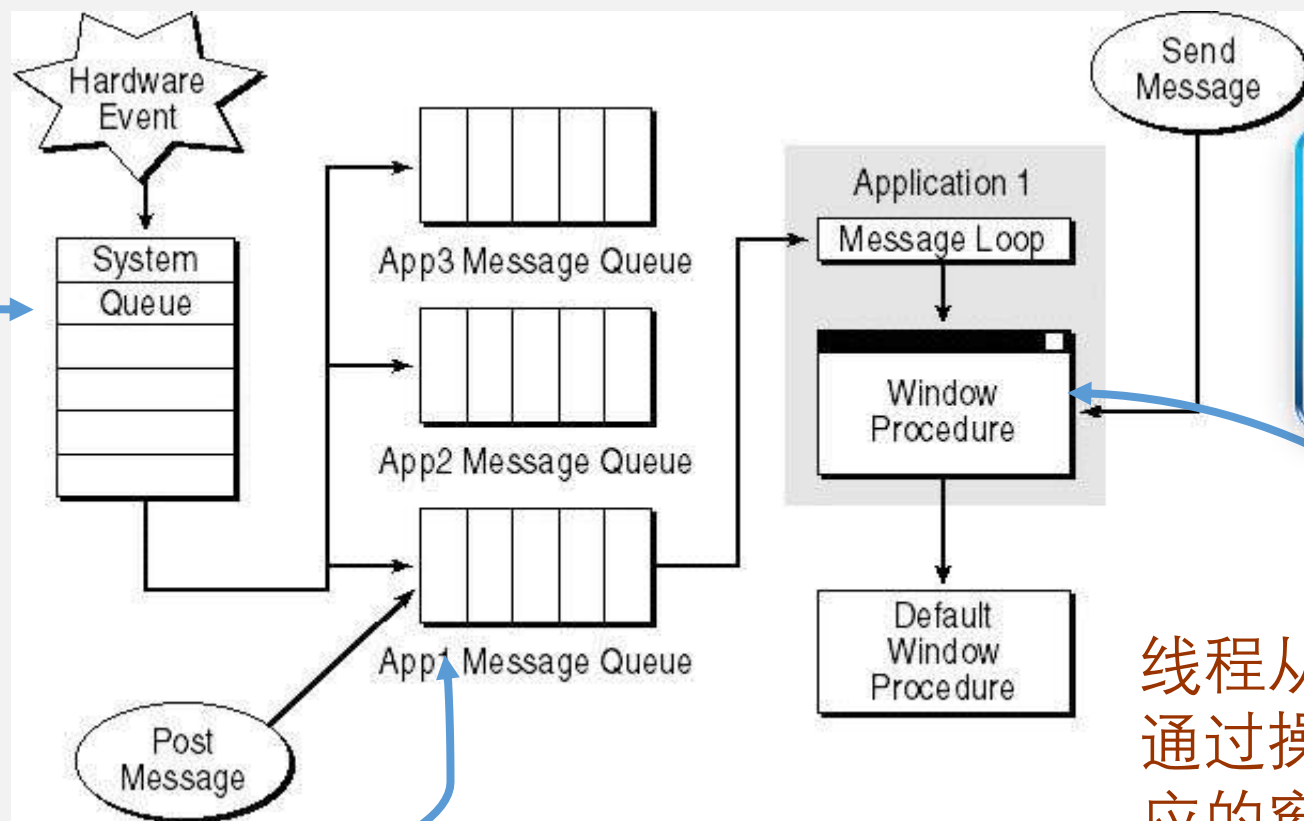
## 队列的应用

队列具有的先进先出的固有特性，使得队列成为程序设计中常用的工具。

任务调度

事件被触发，产生消息，存入系统的消息队列。

OS从系统消息队列中取出一个消息，送往处理该消息的线程消息队列。



先来先服务

对短作业不利  
改进.....

线程从队列中取出消息，通过操作系统发送到对应的窗口过程去处理。



## 3.8 栈的存储与处理

## 栈的概念

设栈  $s = (a_1, a_2, \dots, a_i, \dots, a_n)$  ,

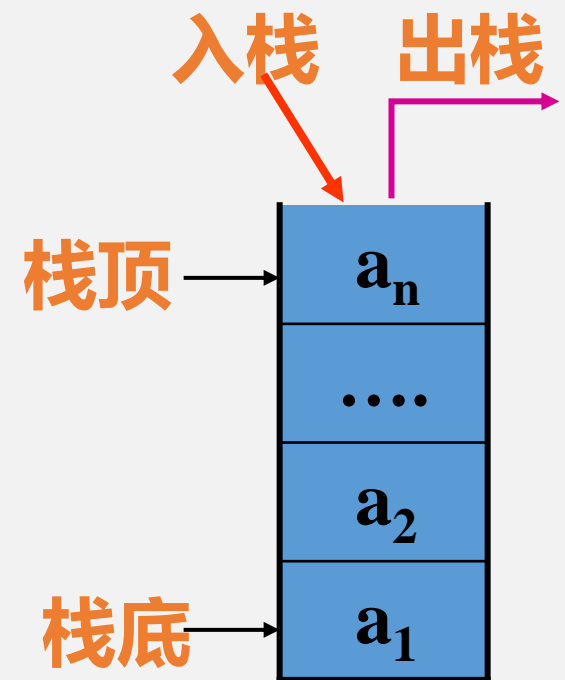
其中  $a_1$  是栈底元素,  $a_n$  是栈顶元素。

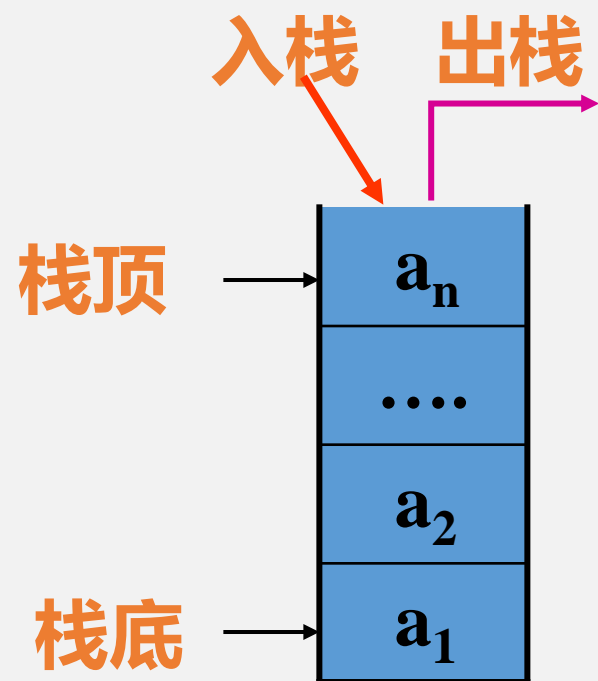
栈顶 (top): 允许插入和删除的一端;

栈底 (bottom): 不允许插入和删除的一端。

栈是操作位置受限: 只能在栈顶进行插入和删除。

栈也称为称为后进先出表





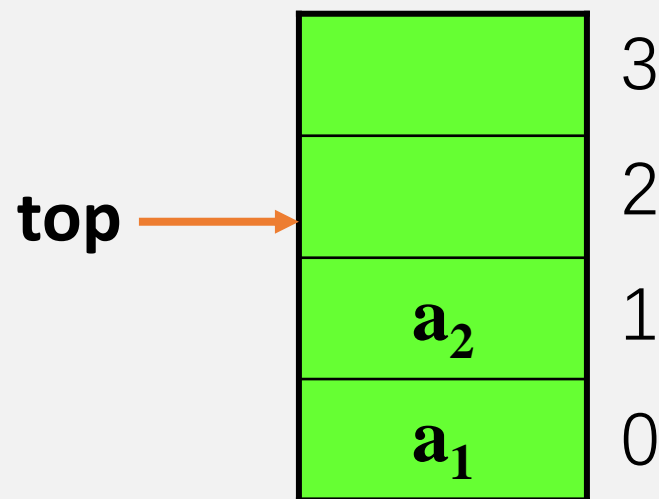
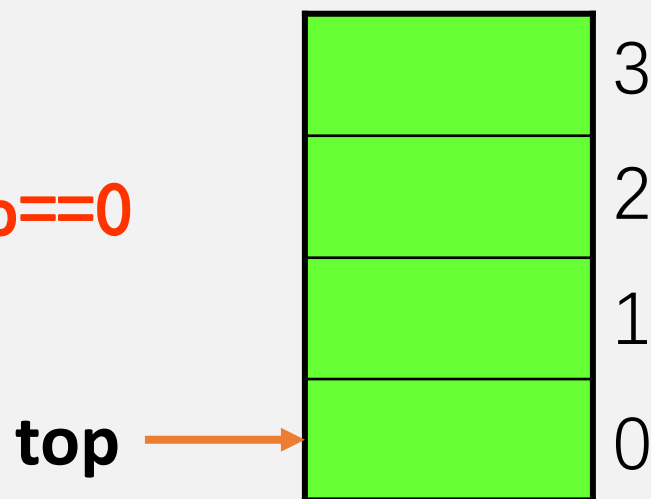
第一个进栈的元素在栈底；  
最后一个进栈的元素在栈顶；  
第一个出栈的元素为栈顶元素；  
最后一个出栈的元素为栈底元素。  
不含元素的栈称为空栈。

## 栈的顺序存储

顺序栈用一组连续的存储单元存放自栈底到栈顶的数据元素。

一般用一维数组表示，设置简单变量 $top$ 指示栈顶位置，称为**栈顶指针**，它始终指向待插入元素的位置。即：栈顶元素在 $top-1$ 处。

空栈：  $top == 0$



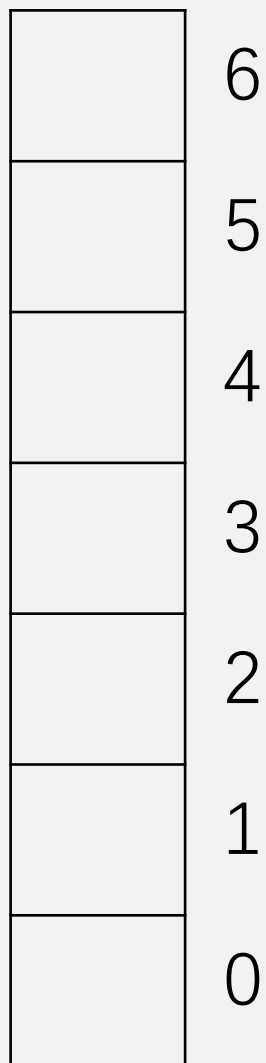
## 入栈

**int a[7],top;**

**top=0;**

空栈

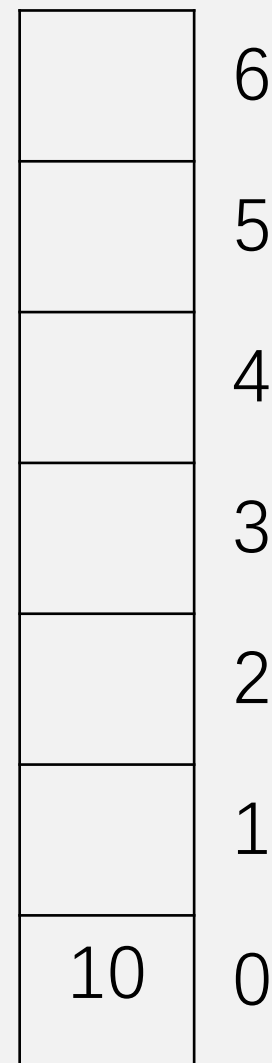
Top=0



**a[top]=m; top=top+1;**

10入栈

Top=1



20,12,34,56,78,9入栈

Top=6

	6
9	5
78	4
56	3
34	2
20	1
10	0

Top=7

26	6
9	5
78	4
56	3
34	2
20	1
10	0

26入栈

满栈

再入栈，则溢出

出栈

Top=7

26	6
9	5
78	4
56	3
34	2
20	1
10	0

$\text{top}=\text{top}-1; m=a[\text{top}];$

Top=5

26、9出栈

26	6
9	5
78	4
56	3
34	2
20	1
10	0



若有**A,B,C**三个元素进**S**栈的顺序是**A,B,C**，则可能的出栈序列有：

**ABC**

**BAC**

**ACB**

**BCA**

**CBA**

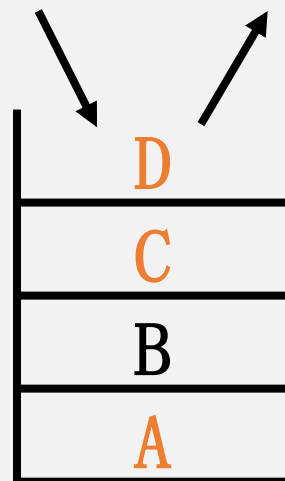
入栈、出栈可能交叉进行

## 栈的应用

栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。

### 内容逆置

ABCD-----DCBA



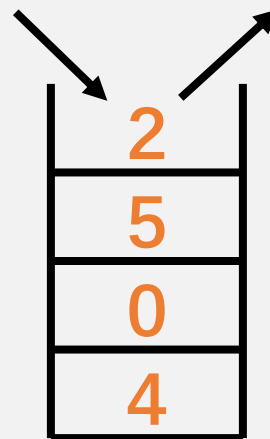
## 数制转换

原理：  $N = (N \text{ div } d) * d + N \text{ mod } d$

算法分析：

1. 当  $N \neq 0$ ，将  $N \% r$  压入栈中；
2. 用  $N / r$  代替  $N$ ；
3. 若  $N > 0$ ，则重复以上步骤，若  $N == 0$ ，则依次输出栈的内容。

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2



## 递归调用

栈的另一个重要应用是在程序设计语言中实现递归调用。

**递归调用**：一个函数(或过程)直接或间接地调用自己本身。

**递归**是程序设计中的一个强有力的工具。

◆递归函数结构清晰，程序易读，正确性很容易得到证明。

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

**递推规则(方法)，终止条件。**

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

$\text{fact}(10) = \text{fact}(9) * 10;$

$\text{fact}(9) = \text{fact}(8) * 9;$

$\text{fact}(8) = \text{fact}(7) * 8;$

.....

$\text{fact}(1) = \text{fact}(0) * 1;$

$\text{fact}(0) == 1$

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

每一层递归包含信息：参数、局部变量、上一层的返回地址构成一个“工作记录”。

每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

董卫军





## 3.9 二叉树的概念与性质

## 树的概念

由一个或多个结点组成的有限集合。仅有一个根结点，结点间有明显的层次结构关系。

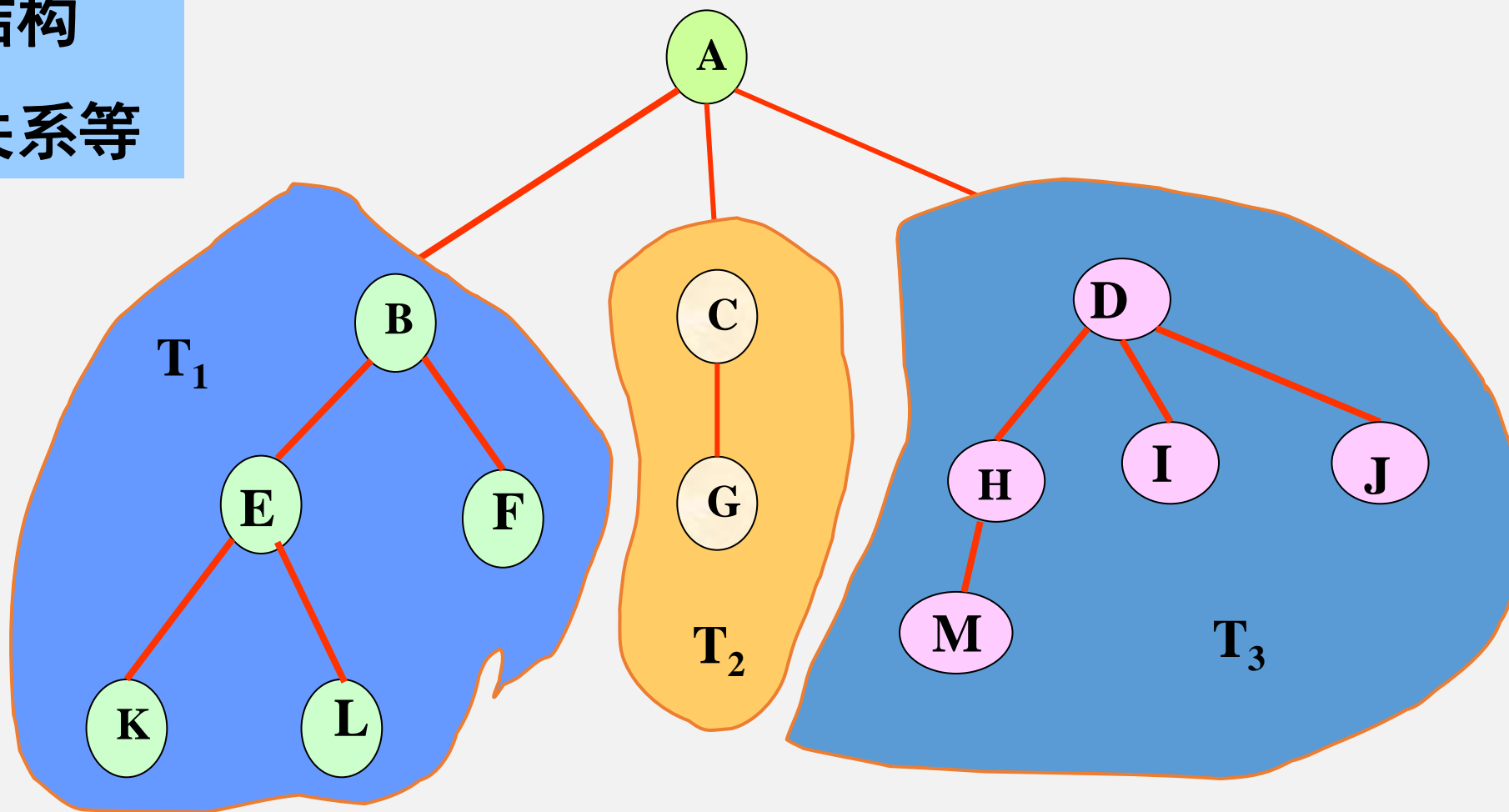
树(Tree)是 $n(n \geq 0)$ 个结点的有限集合 $T$ ，若 $n=0$ 时称为空树，否则：

- ◆ 有且只有一个特殊的称为树的根(Root)结点；
- ◆ 若 $n>1$ 时，其余的结点被分为 $m(m>0)$ 个互不相交的子集 $T_1, T_2, T_3 \cdots T_m$ ，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。

组织机构关系

书的层次结构

家族血缘关系等



## 树的基本术语

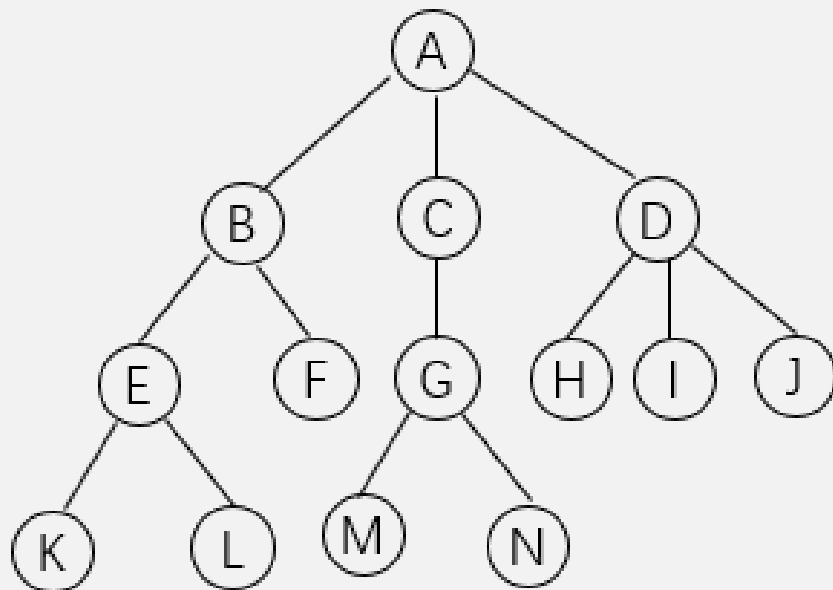
**结点**：一个数据元素及其若干指向其子树的分支。

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

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



(a) 只有根结点

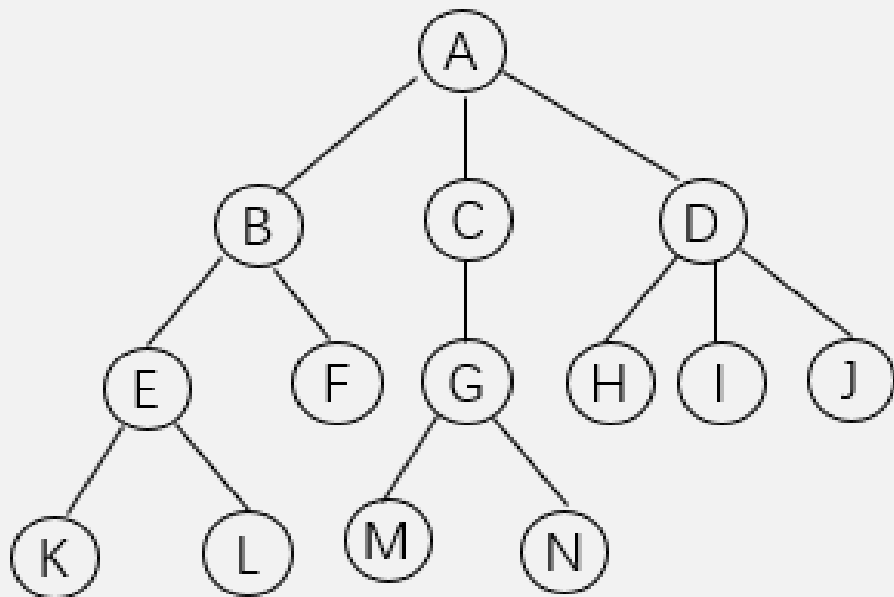


(b) 一般的树

**叶子结点、非叶子结点：**树中度为0的结点称为叶子结点（或终端结点）。

相对应地，度不为0的结点称为非叶子结点。

**孩子结点、双亲结点、兄弟结点：**一个结点的子树的根称为该结点的孩子结点（子结点）；相应地，该结点是其孩子结点的双亲结点（父结点）。



K、L、F、M、N：叶子节点

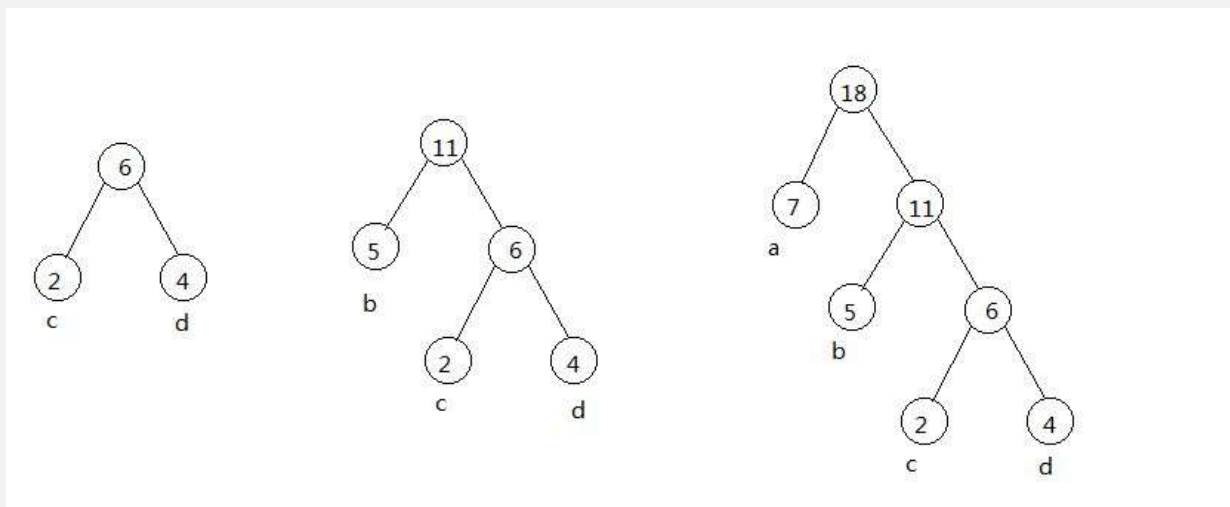
A的孩子节点：B、C、D

兄弟节点：H、I、J

**树的深度：** 树中结点的最大层次值，又称为树的高度。

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

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



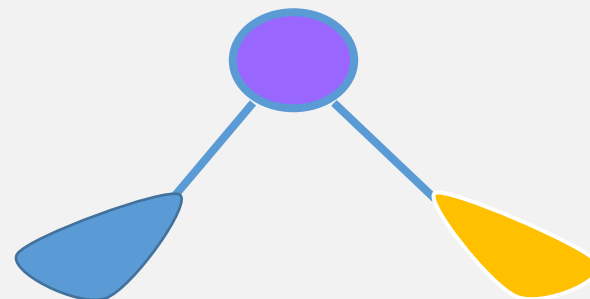
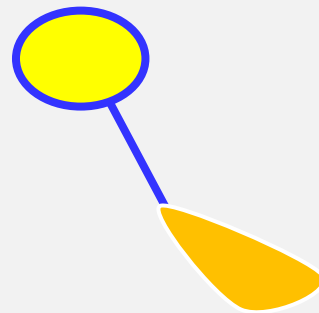
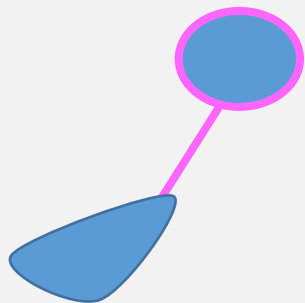
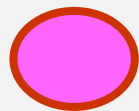
## 二叉树

### 二叉树的概念

二叉树是 $n$  ( $n \geq 0$ ) 个结点的有限集合。它或为空树 ( $n=0$ )，或由一个根结点和两棵分别称为根的左子树和右子树的互不相交的二叉树组成。

二叉树是一种树型结构。

特点：树中每个结点最多有两棵子树，且子树有左右之分。



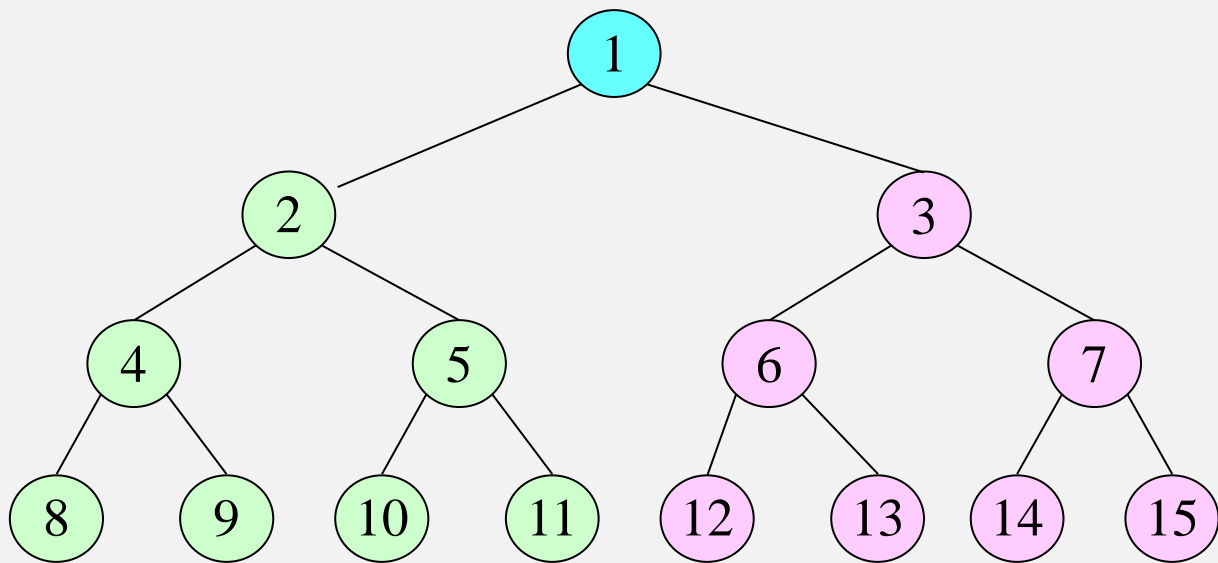
空二叉树    仅有根结点    右子树为空    左子树为空    左右子树均非空

二叉树的五种基本形态



## 二叉树的性质

- ◆ 二叉树的第 $i$ 层上至多有 $2^{i-1}$  ( $i \geq 1$ ) 个结点。
- ◆ 深度为 $h$ 的二叉树中至多含有 $2^h - 1$ 个结点。
- ◆ 若在任意一棵二叉树中，有 $n$ 个叶子结点，有 $m$ 个度为2的结点，则： $n = m + 1$



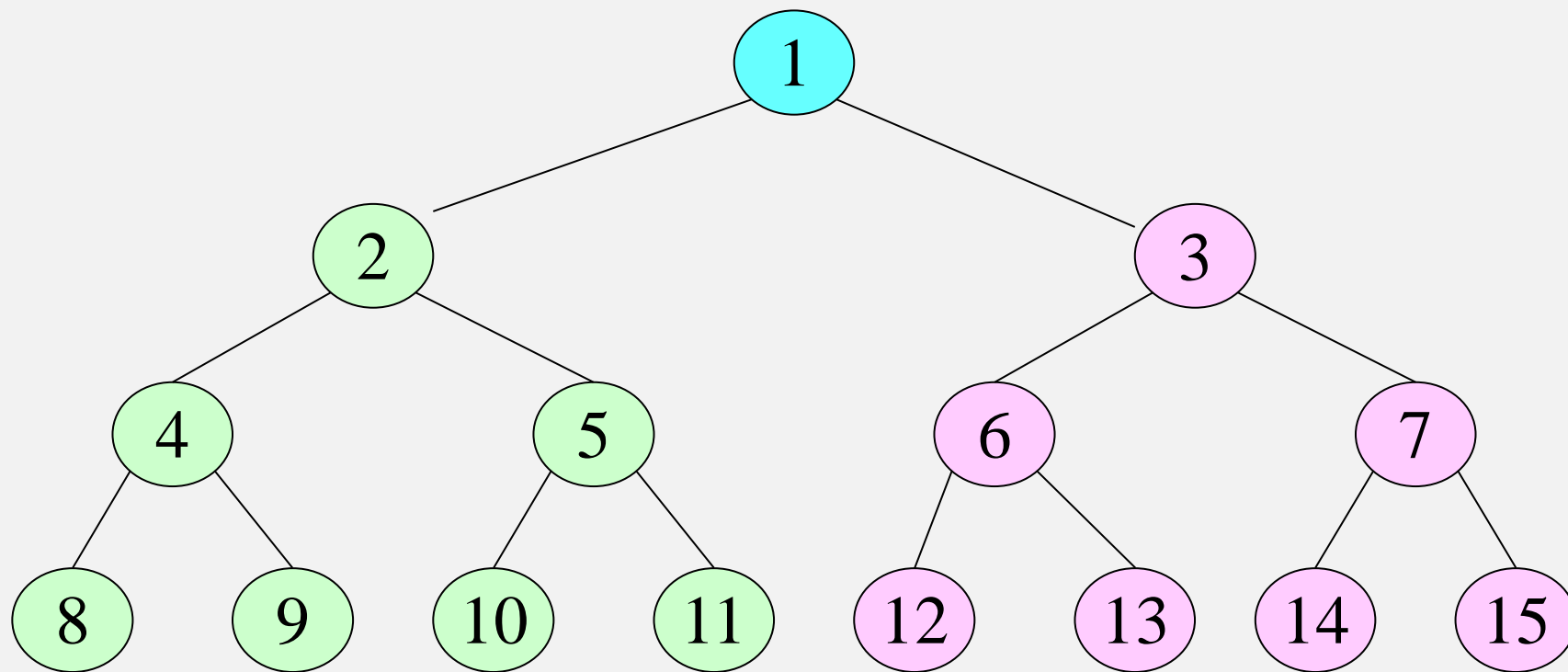
第三层上 ( $i=3$ )，有 $2^{3-1}=4$ 个节点。

第四层上 ( $i=4$ )，有 $2^{4-1}=8$ 个节点。

深度 $h=4$ ，最多有 $2^4-1=15$ 个节点。

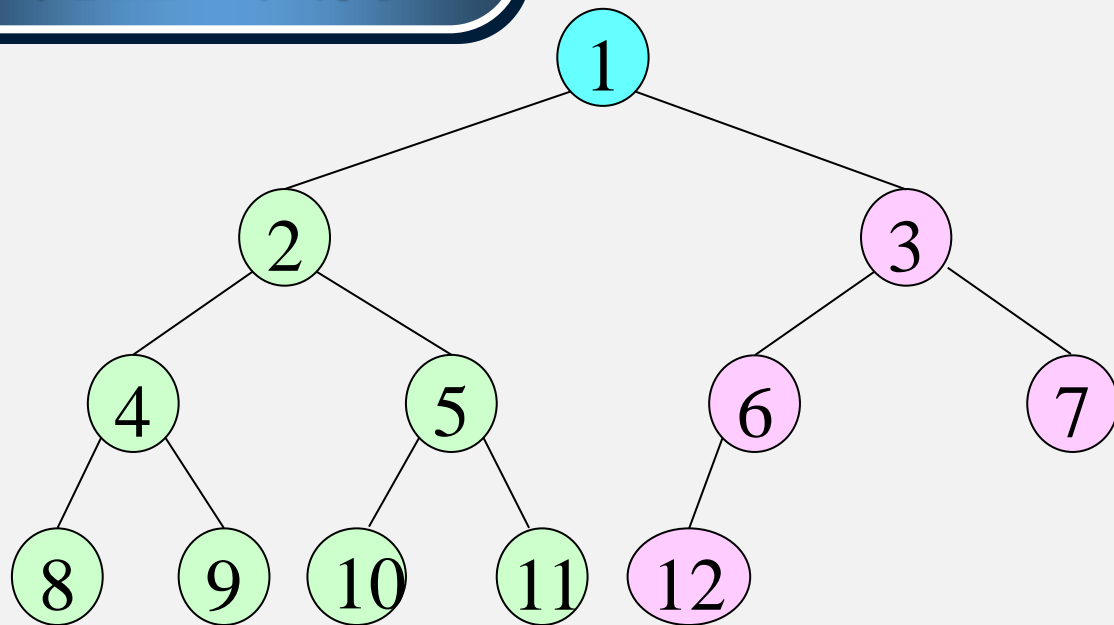
$n=8$ ， $m=7$ 。

## 满二叉树

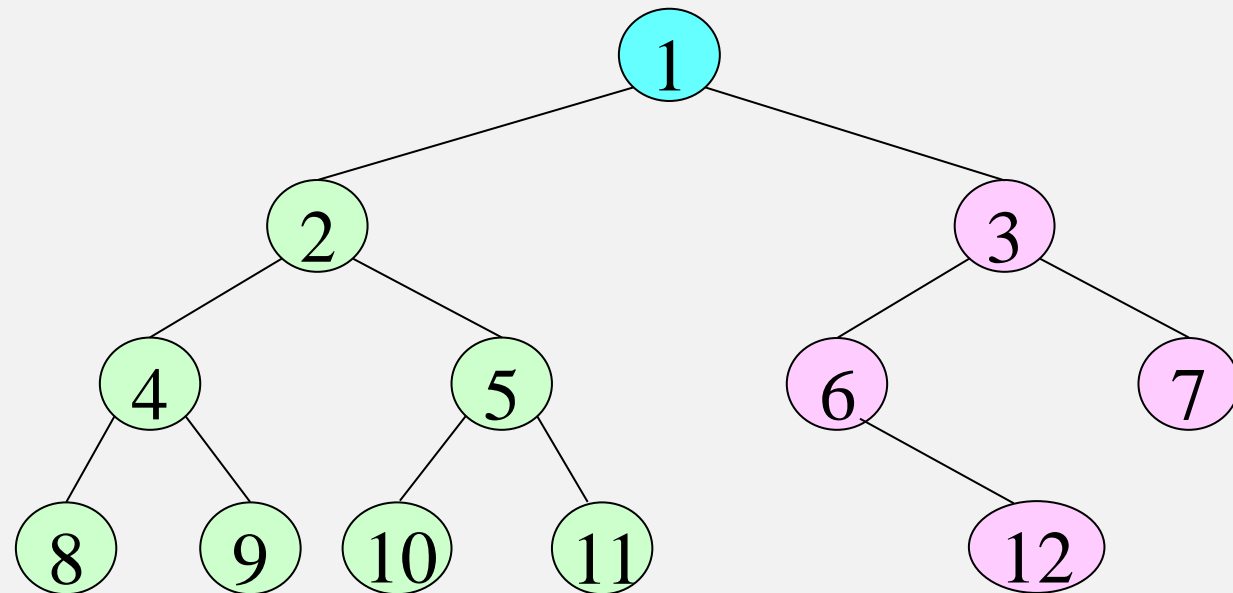


特点：每一层上都含有最大结点数。

## 完全二叉树

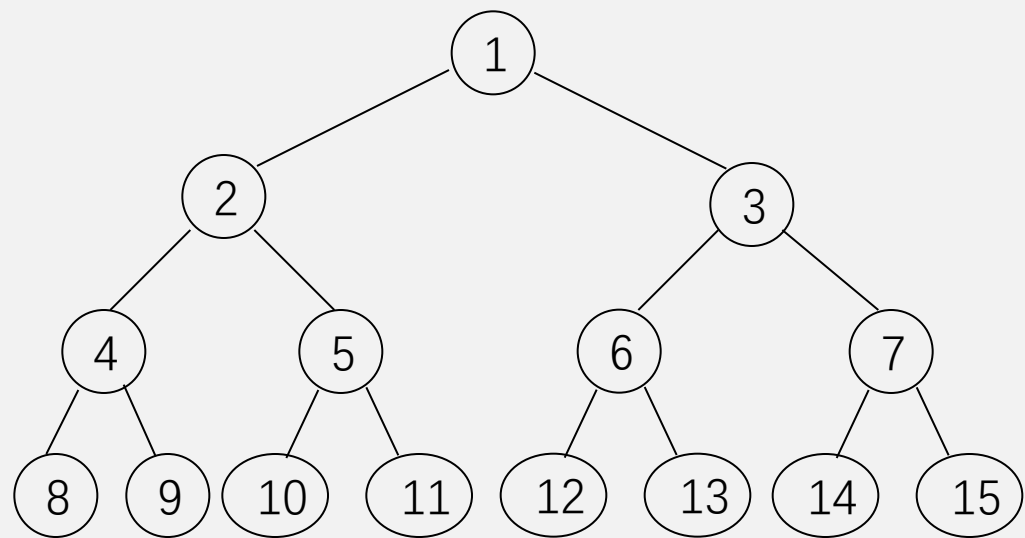


完全二叉树

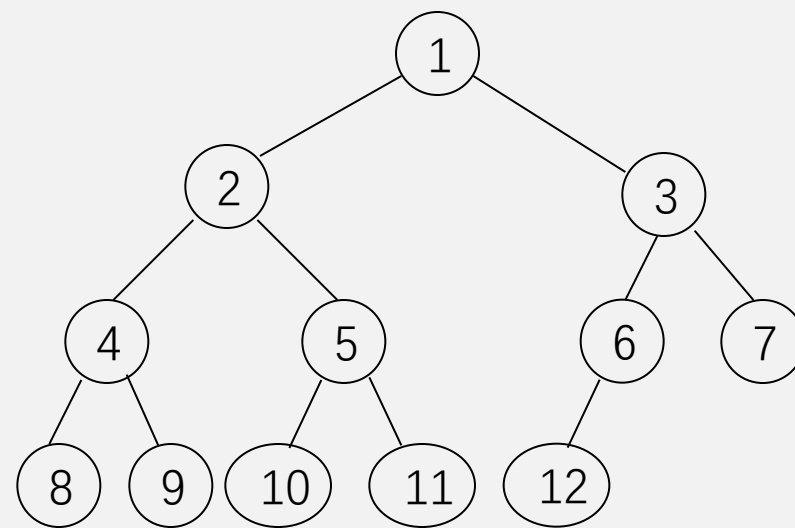


非完全二叉树

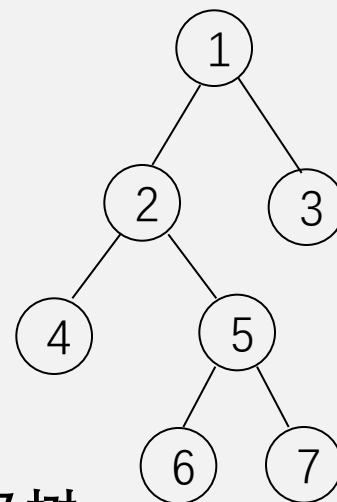
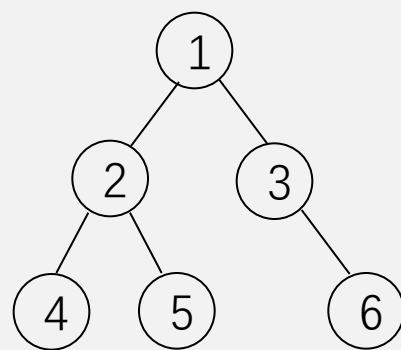
**特点：**除最后一层外，每一层都取最大结点数，  
最后一层结点都集中在该层最左边的若干位置。



(a) 满二叉树



(b) 完全二叉树



(c) 非完全二叉树

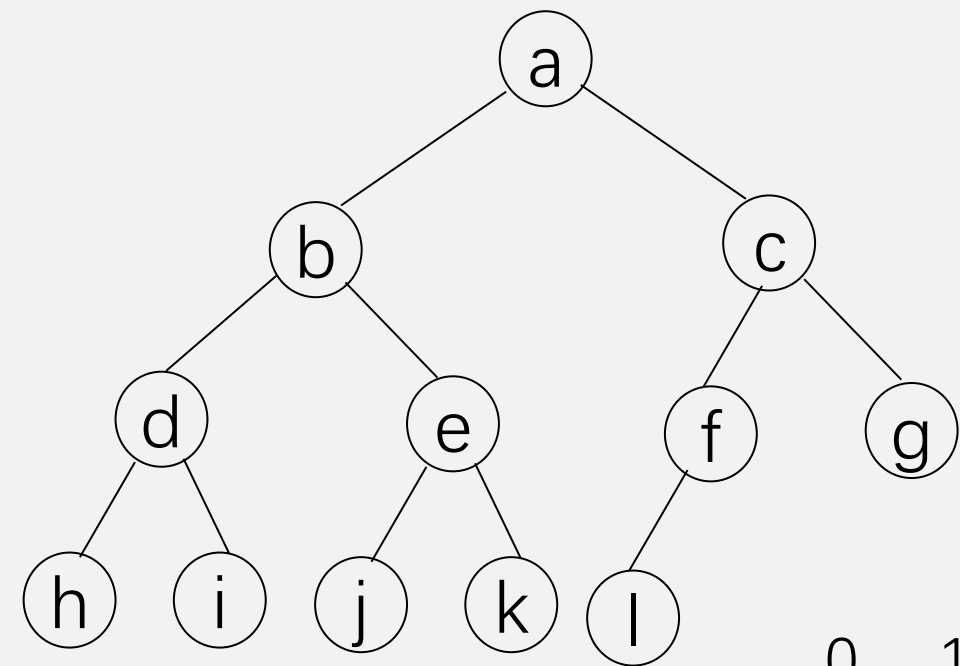


## 3.10 二叉树的存储与遍历

## 二叉树的顺序存储

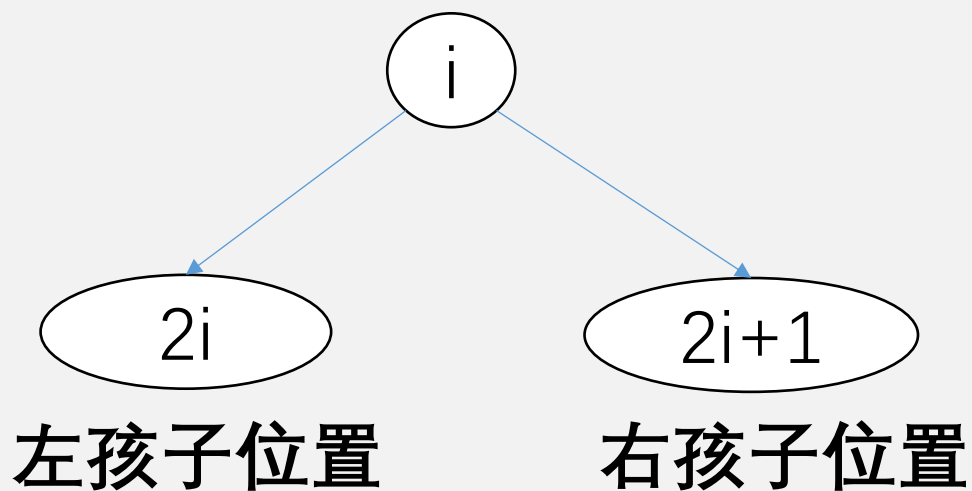
用一组地址连续的存储单元依次“**自上而下、自左至右**”存储完全二叉树的数据元素。

- ◆ 若不是完全二叉树，则补 **空** 使其变为完全二叉树，然后存储。
- ◆ 元素之间的父子、兄弟关系，通过存储位置的关系来表示。

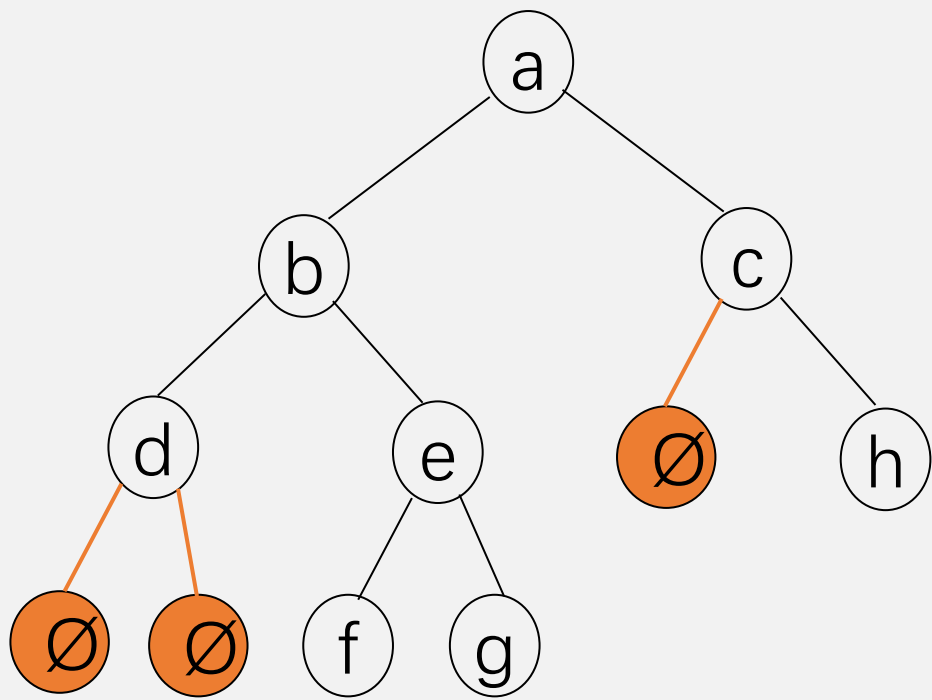


完全二叉树

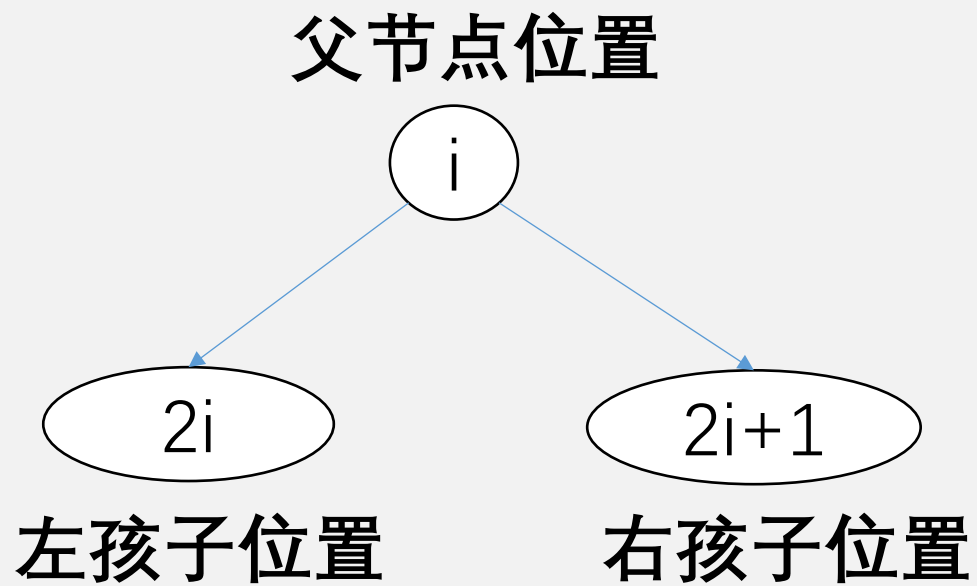
父节点位置



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	a	b	c	d	e	f	g	h	i	j	k	l		



非完全二叉树



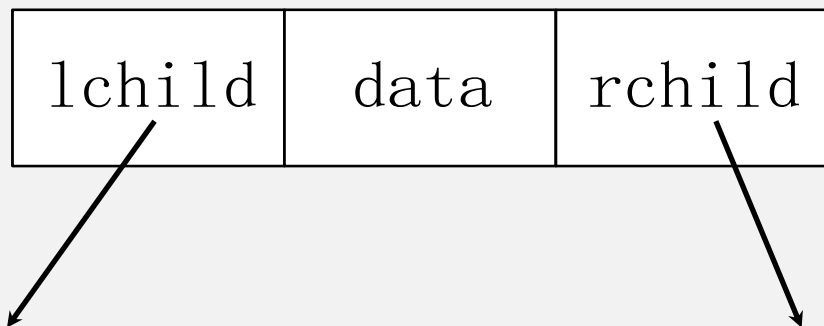
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	a	b	c	d	e	∅	h	∅	∅	f	g			

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

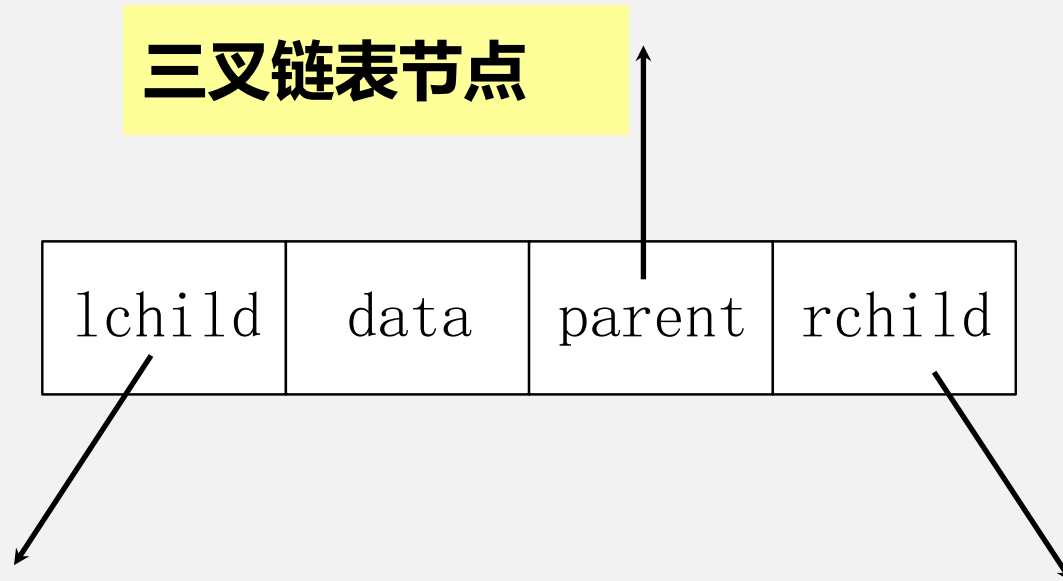


## 二叉树的链式存储

### 二叉链表节点



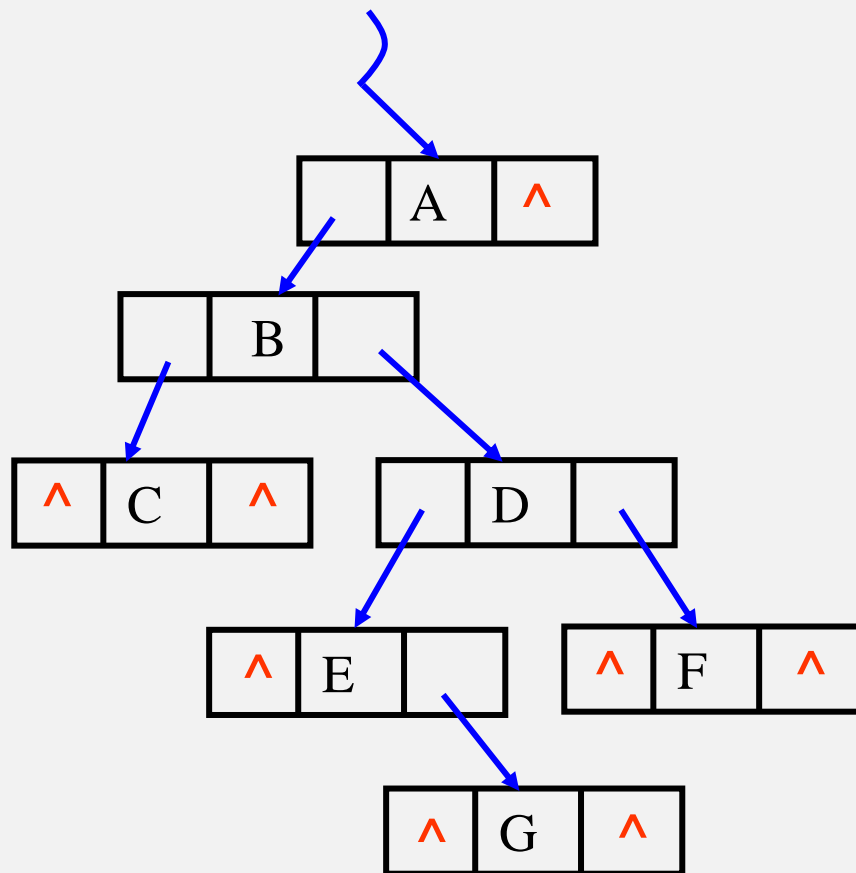
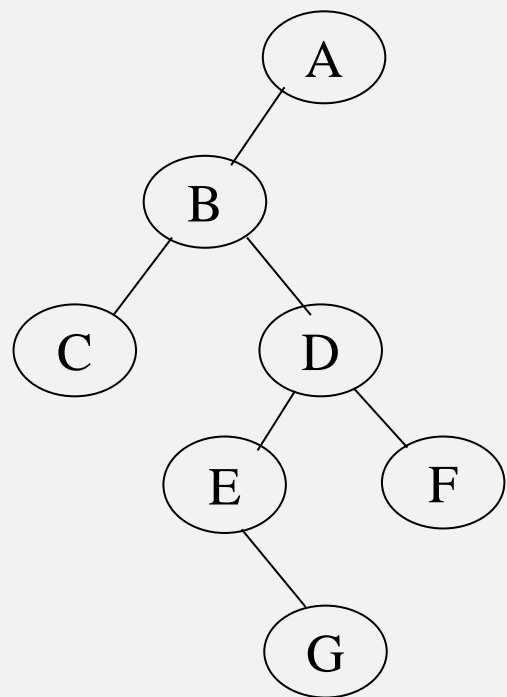
### 三叉链表节点



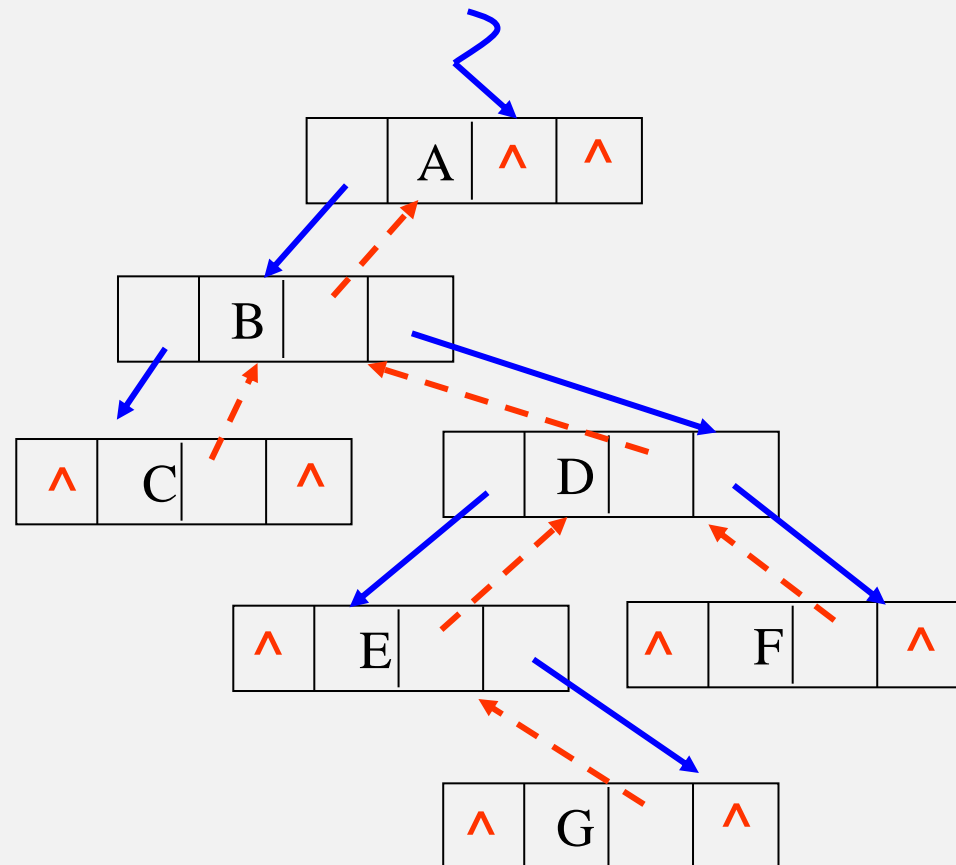
**二叉链表结点：**一个数据域，两个分别指向左右子结点的指针域。

**三叉链表结点：**除二叉链表的三个域外，再增加指向父结点的指针域。

## 二叉链表



## 三叉链表



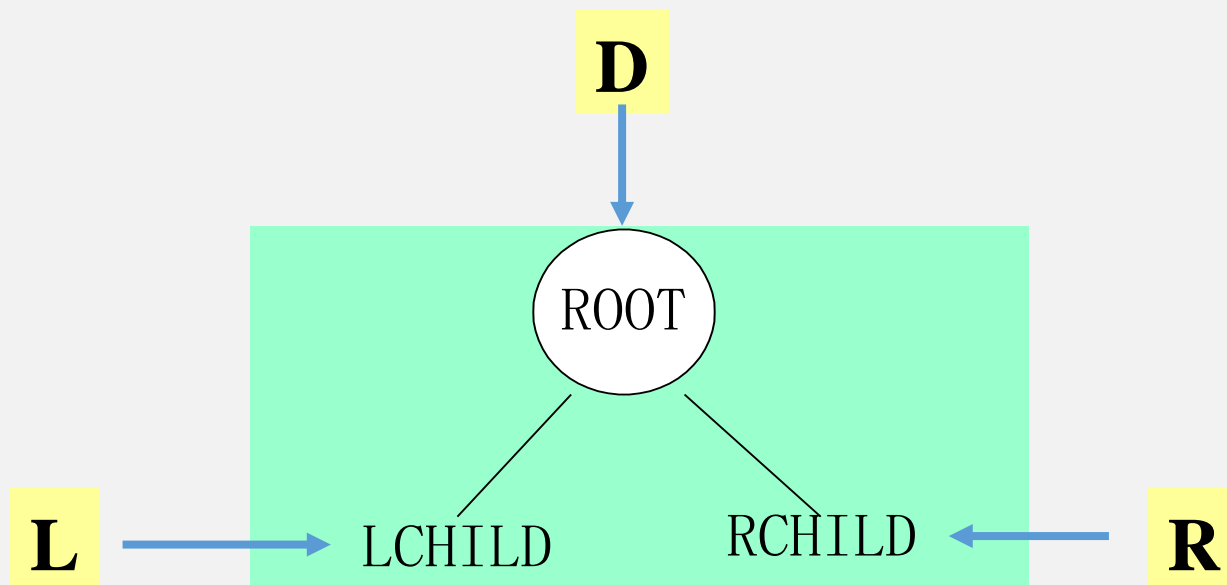
## 遍历的概念

**遍历：**指按指定的规律对二叉树中的每个结点访问一次且仅访问一次。

**用途：**它是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。

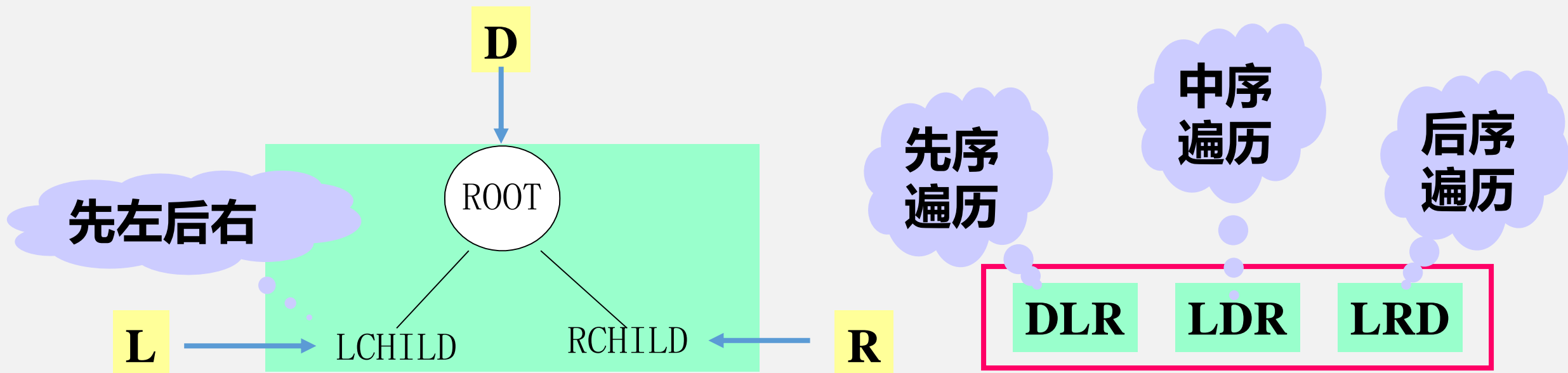
二叉树是一种非线性结构，每个结点都可能都有左、右两棵子树。

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



若以L、D、R分别表示遍历左子树、遍历根结点和遍历右子树，则有六种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD。

若规定先左后右，则只有三种情况。



## 先序递归遍历

二叉树遍历的递归算法是由系统通过使用堆栈来实现控制，结构清晰。

算法的递归定义：

若二叉树为空，则遍历结束；否则

(1) 访问根结点；

(2) 先序遍历左子树(递归调用本算法)；

(3) 先序遍历右子树(递归调用本算法)。

```
void PrTraverse(BTNode *T)
```

```
{ if (T!=NULL)
```

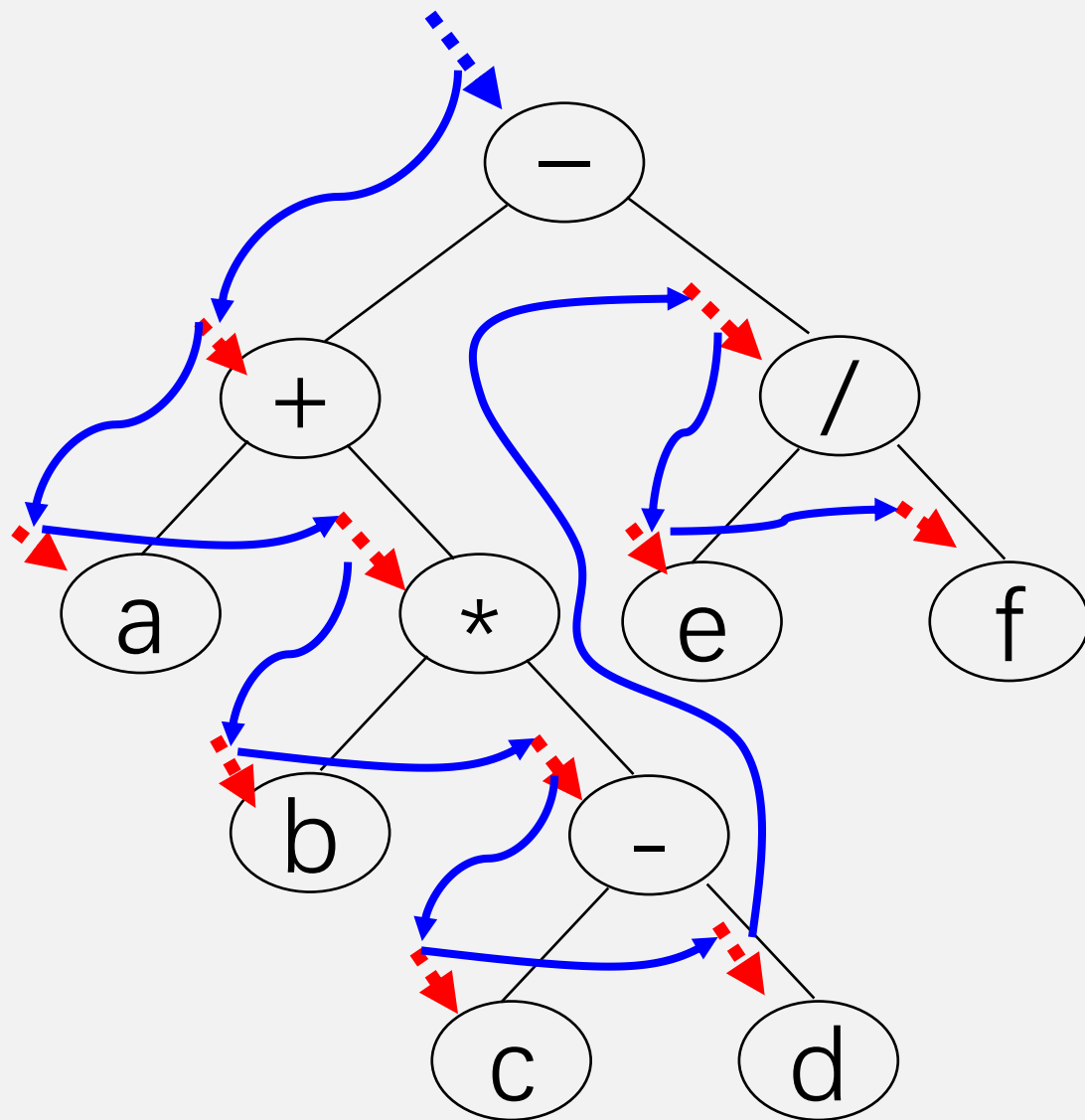
```
{ visit(T->data)
```

```
PrTraverse(T->Lchild);
```

```
PrTraverse(T->Rchild);
```

```
}
```

```
}
```



先序序列  
- + a \* b - c d / e f

## 中序递归遍历

算法的递归定义：

若二叉树为空，则遍历结束；否则

(1) 中序遍历左子树(递归调用本算法)；

(2) 访问根结点；

(3) 中序遍历右子树(递归调用本算法)。

```
void ITraverse(BTNode *T)
```

```
{ if (T!=NULL)
```

```
{ ITraverse(T->Lchild) ;
```

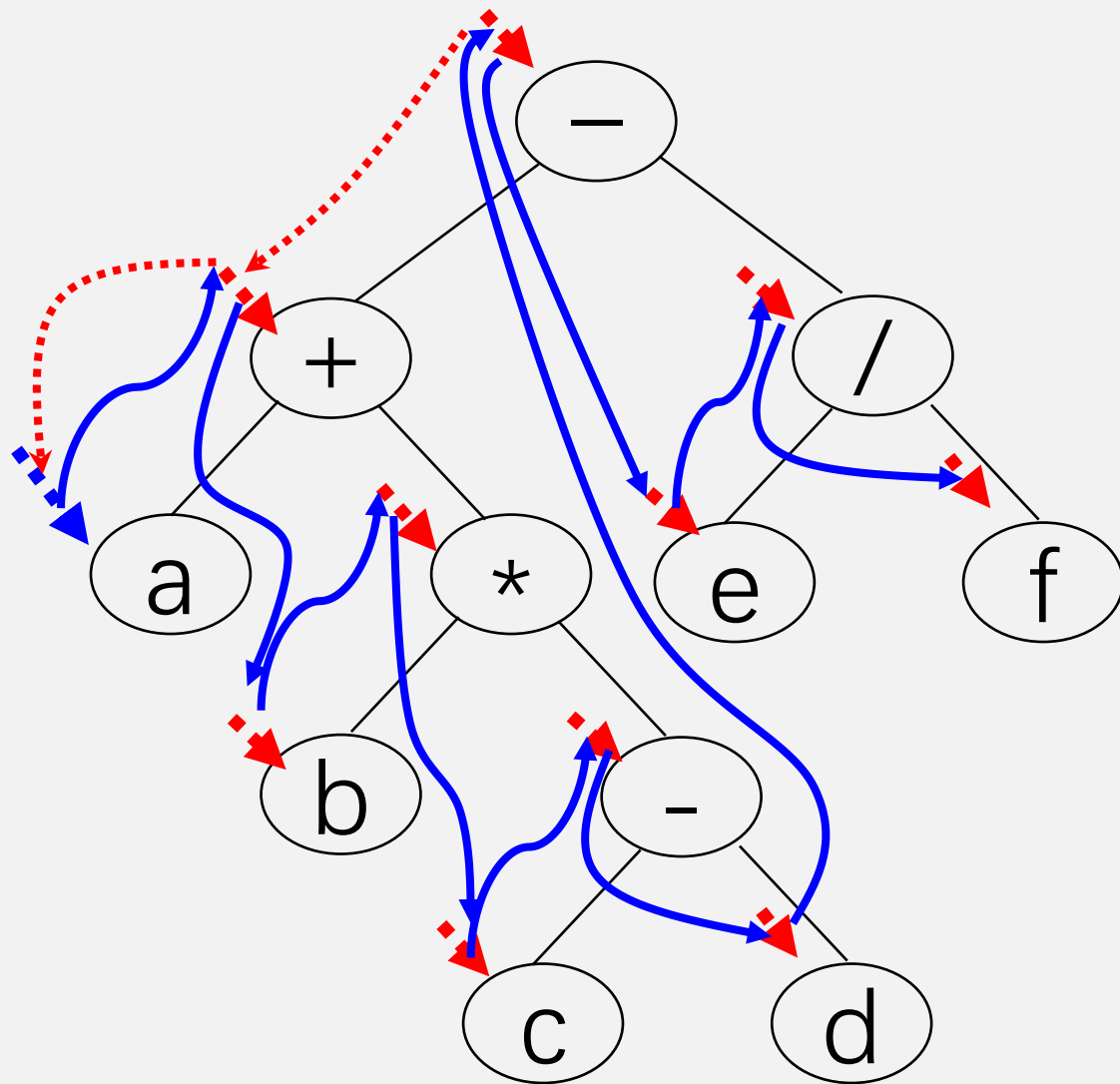
```
  visit(T->data) ;
```

```
  ITraverse(T->Rchild) ;
```

```
}
```

```
}
```





中序序列  
 $a+b*c-d-e/f$



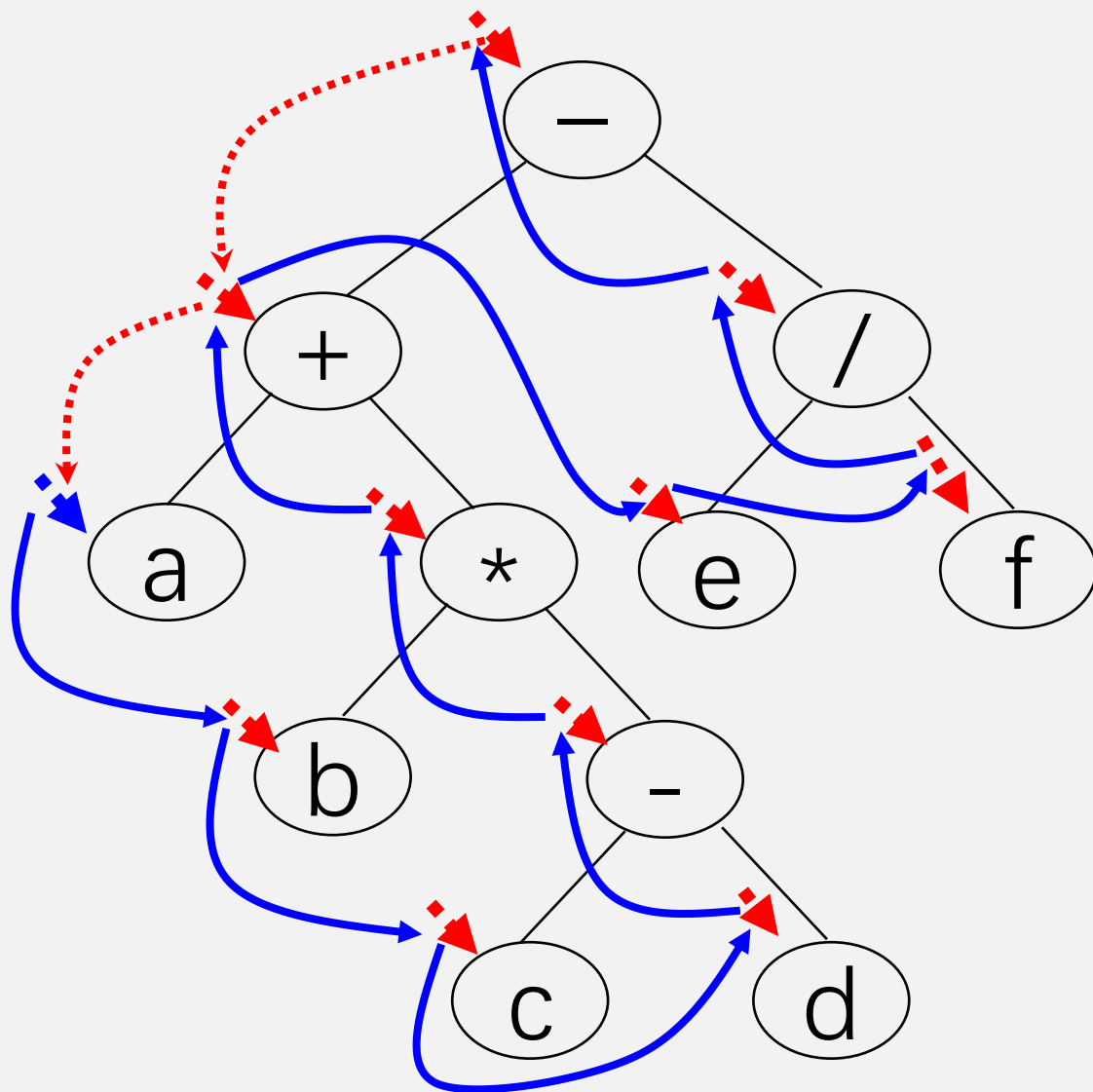
## 后序递归遍历

算法的递归定义：

若二叉树为空，则遍历结束；否则

- (1) 后序遍历左子树(递归调用本算法)；
- (2) 后序遍历右子树(递归调用本算法)。
- (3) 访问根结点；

```
void PoTraverse(BTNode *T)
{ if (T!=NULL)
  { PoTraverse(T->Lchild) ;
    PoTraverse(T->Rchild) ;
    visit(T->data) ;
  }
}
```



后序序列  
 $abcd-*+ef/-$



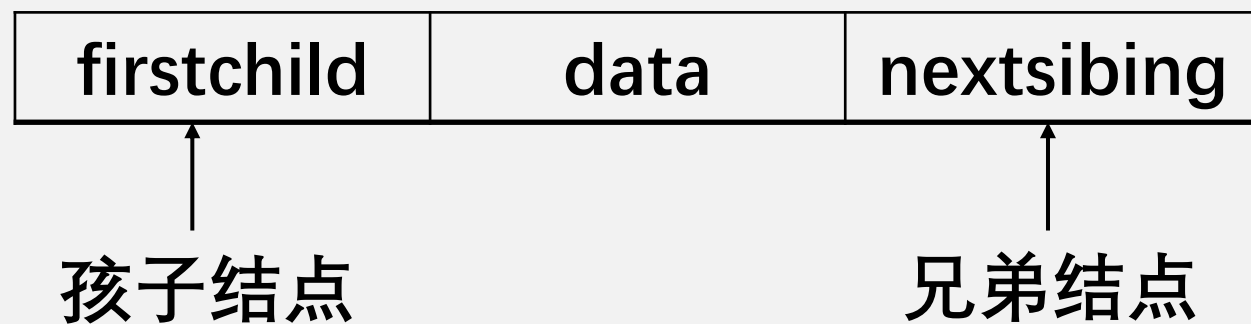
华南师范大学  
SOUTH CHINA NORMAL UNIVERSITY

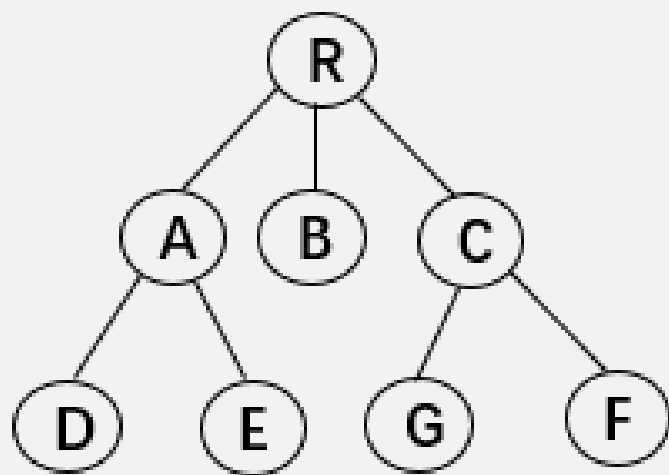
## 3.11 二叉树与树

## 树和二叉树的转换

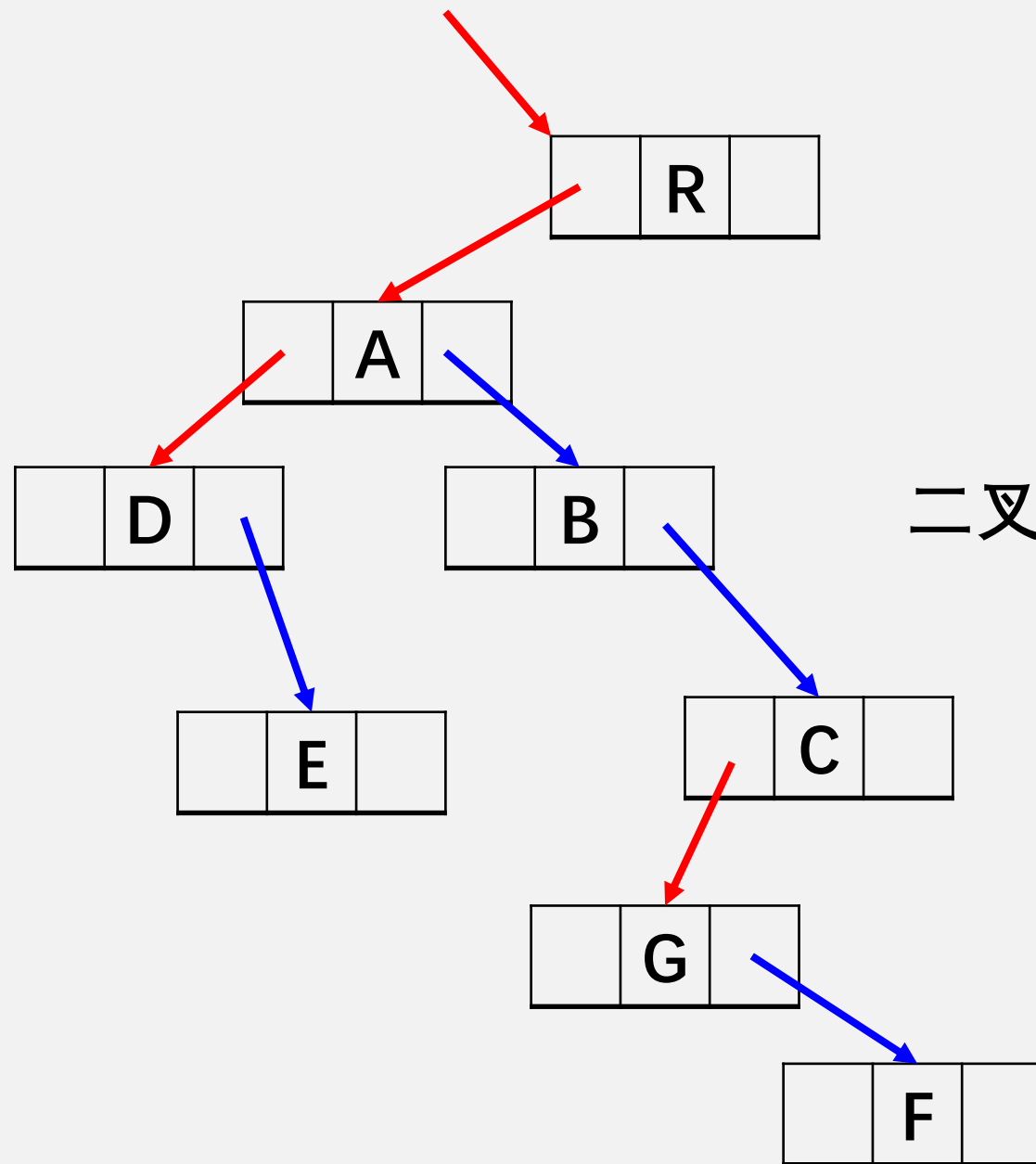
### 树的二叉链表存储

以二叉链表作为树的存储结构：孩子兄弟法。



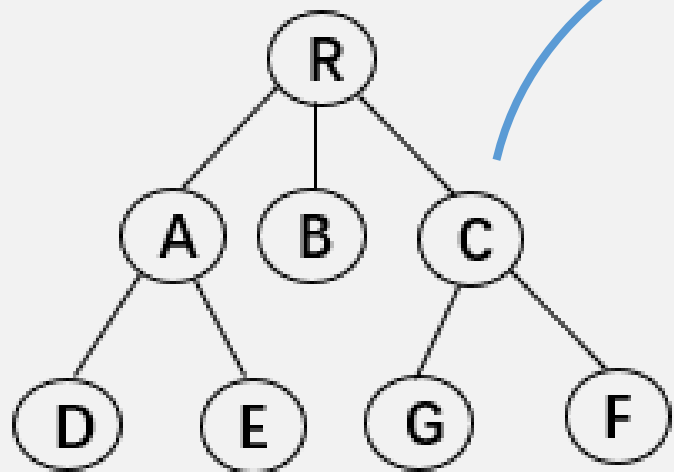


树



二叉链表存储

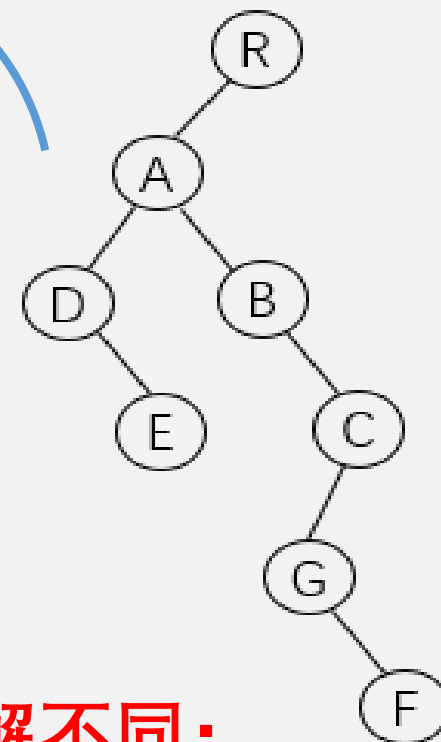
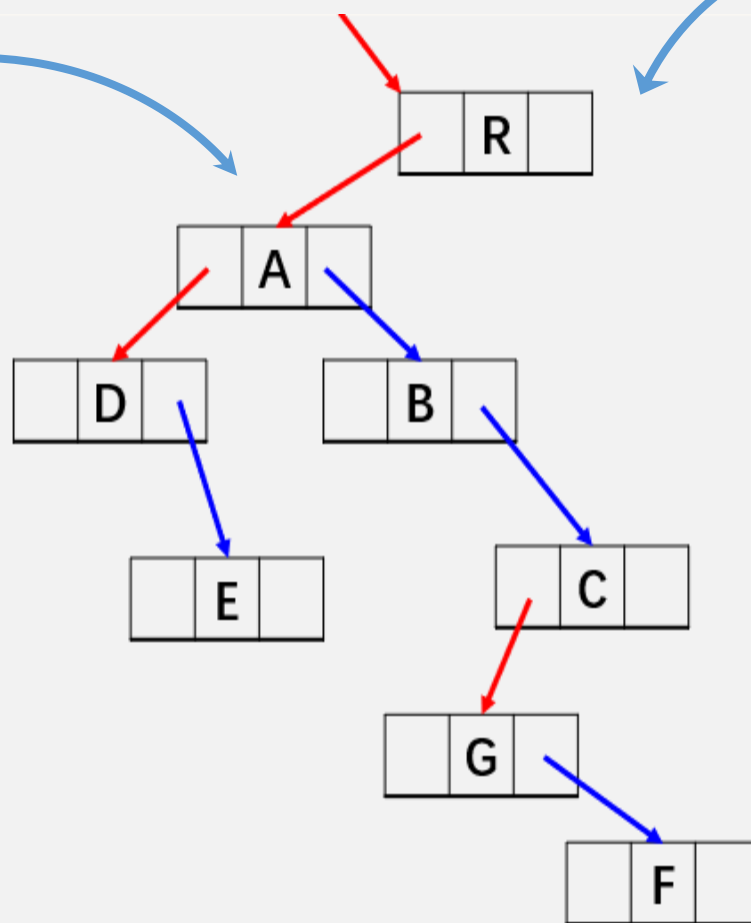
## 树与二叉树存储关系



理解不同：

左指针指向孩子；

右指针指向兄弟。



理解不同：

左指针指向左孩子；

右指针指向右孩子。

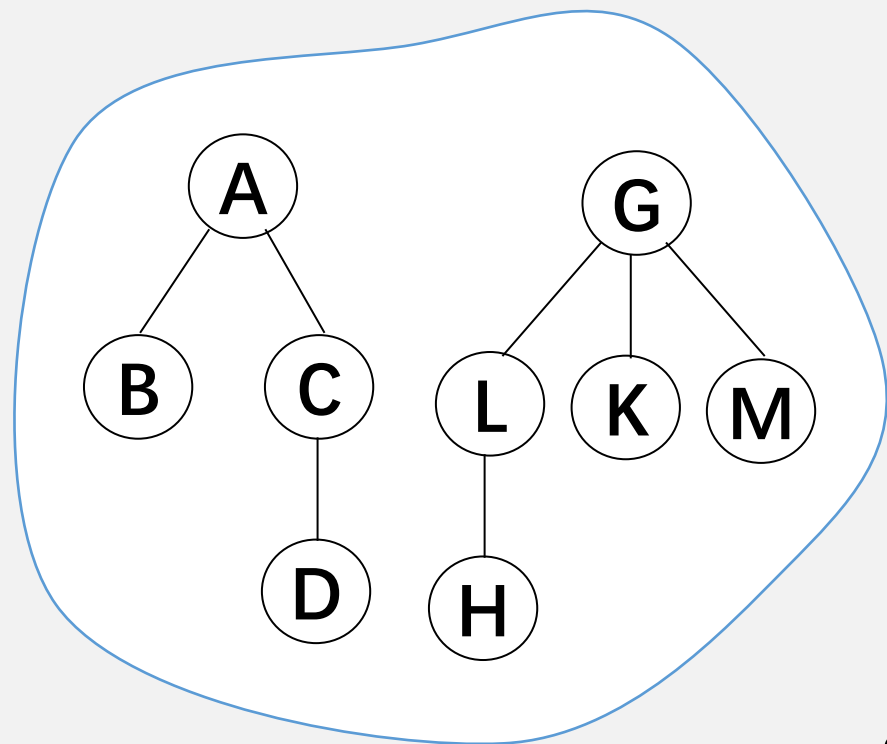
## 森林到二叉树的转换

森林转换成二叉树

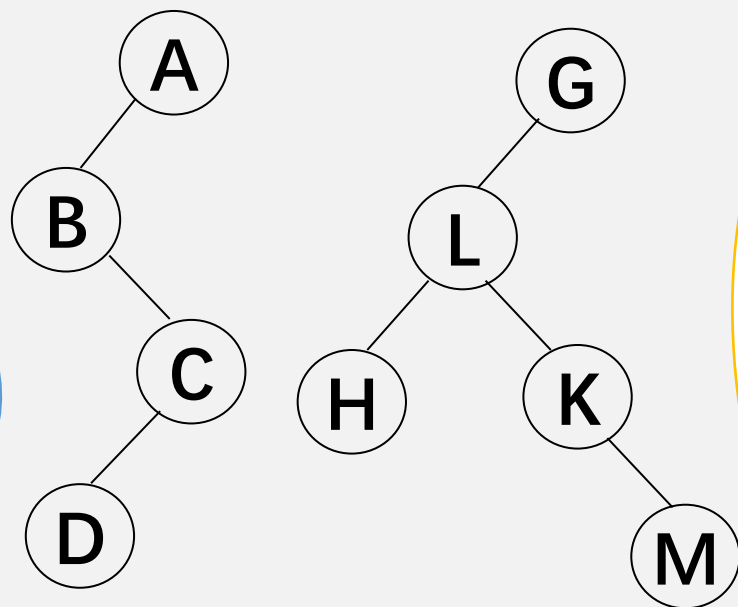


孩子兄弟表示法

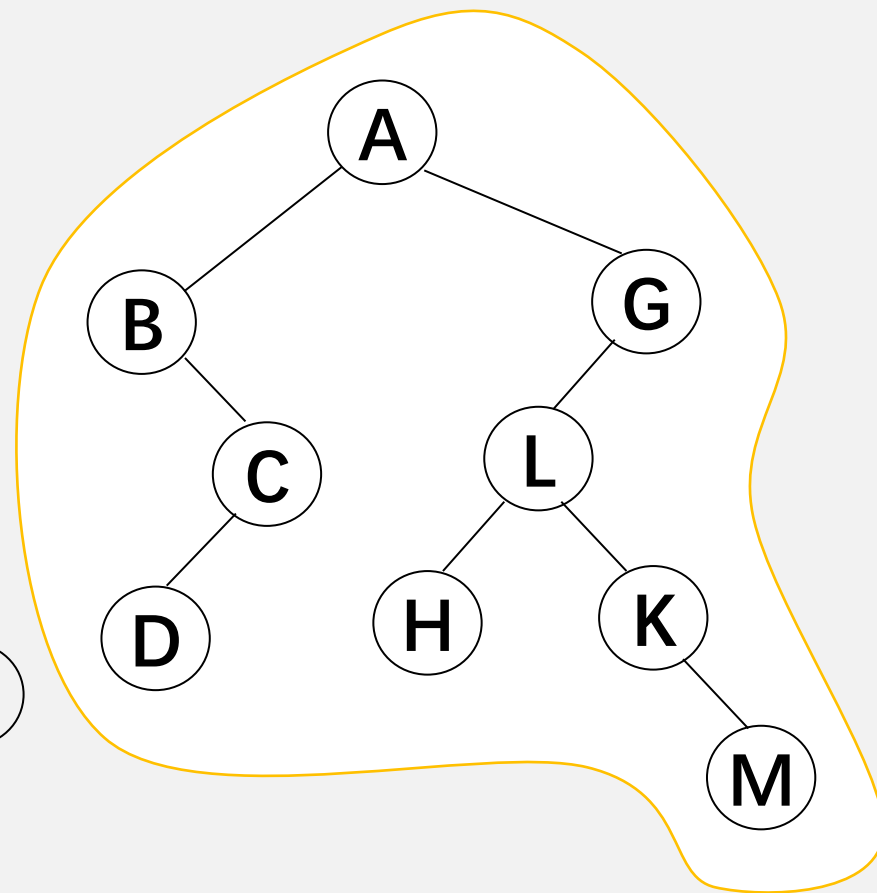




(a) 森林



(b) 森林中每棵树  
对应的二叉树

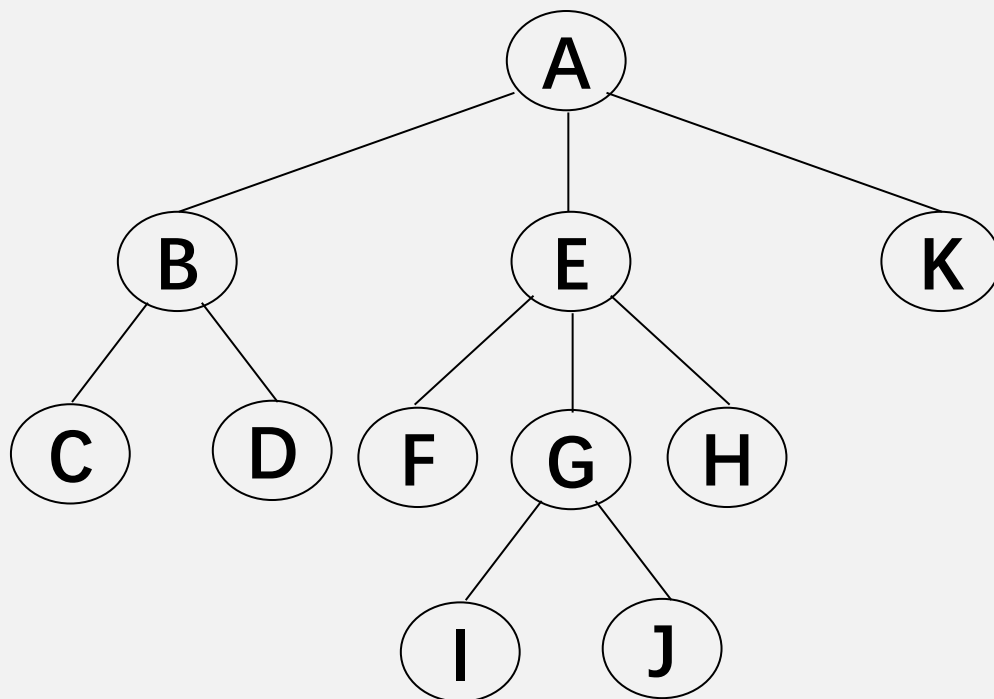


(c) 森林对应的二叉树

## 树的遍历

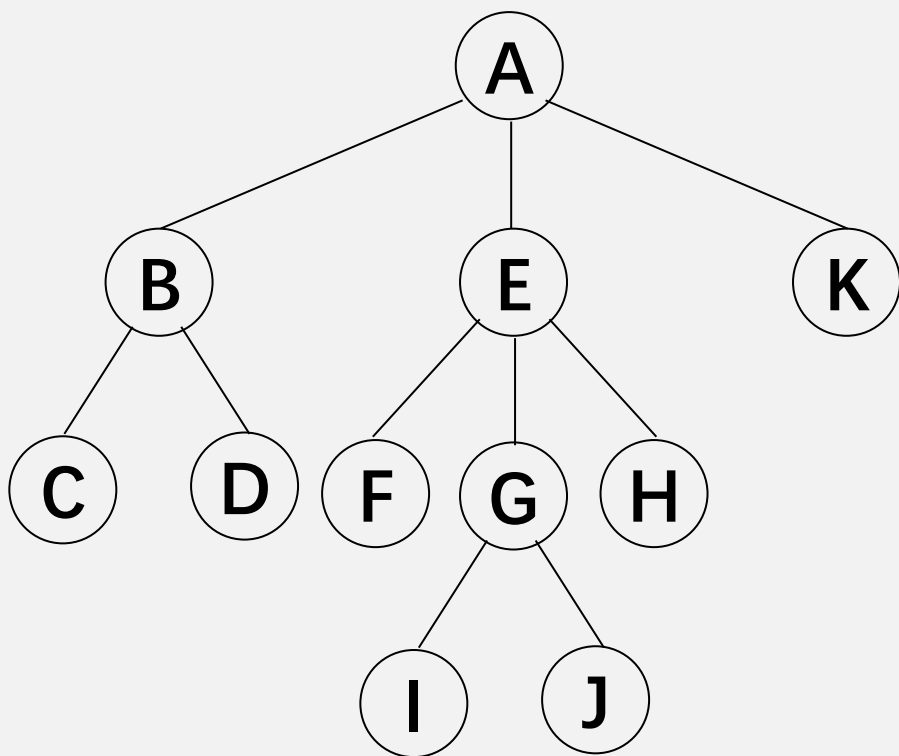
树的遍历有二种方法。

**先序遍历**：先访问根结点，然后依次先序遍历完每棵子树。

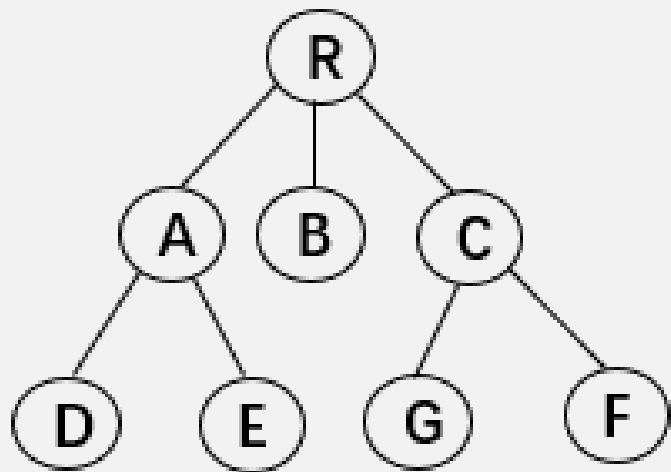


**A**BCDEFGIJK

后序遍历：先依次后序遍历完每棵子树，然后访问根结点。

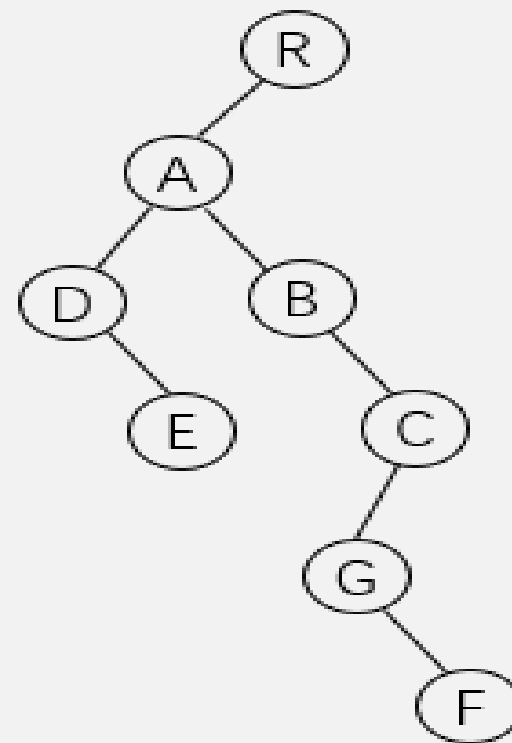


CDBFIJGHEKA



先序遍历: RADEBCGF

后序遍历: DEABGFCR



先序遍历: RADEBCGF

中序遍历: DEABGFCR

树的先序遍历



与将树转换成二叉树后对二叉树的先序遍历相同

树的后序遍历



与将树转换成二叉树后对二叉树的中序遍历相同

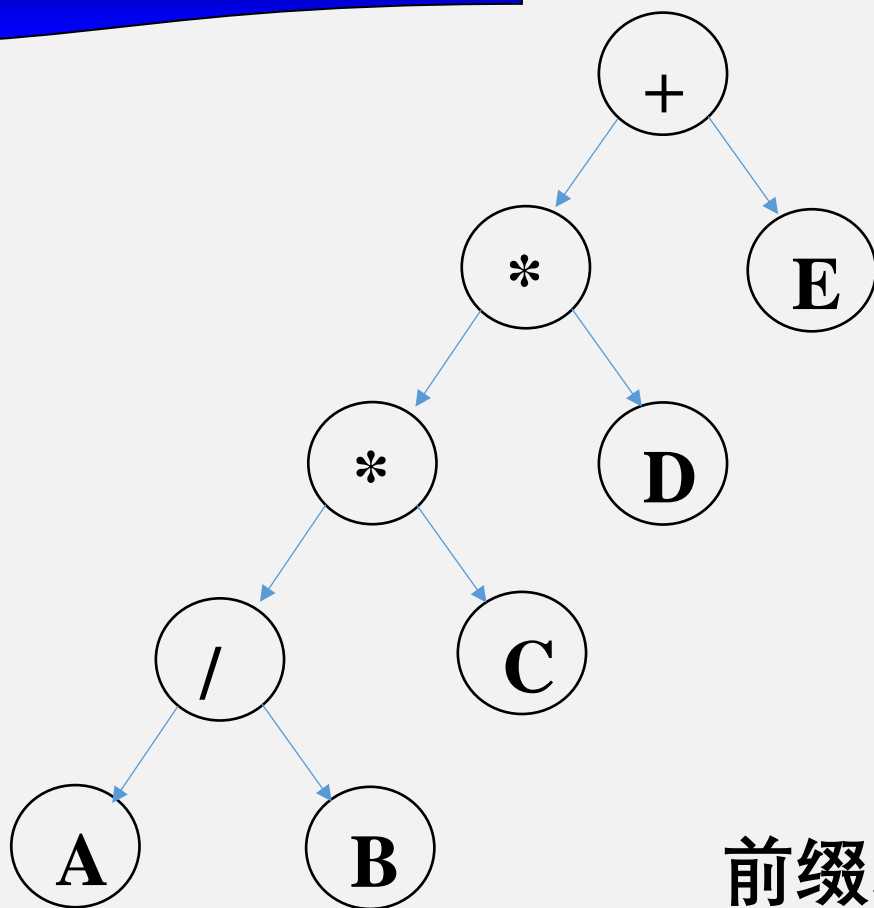
树采用二叉链表存储：简单、规范

对树的操作可以通过对相应二叉树的操作来实现



## 3.12 二叉树的应用

## 二叉树表示算术表达式



**先序遍历**

$+ * * / A B C D E$

**前缀表示**

**中序遍历**

$A / B * C * D + E$

**中缀表示**

**后序遍历**

$A B / C * D * E +$

**后缀表示**

前缀表达式、中缀表达式、后缀表达式都是四则运算的表达方式。

## 前缀表达式的计算：

从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（栈顶元素  $op$  次顶元素），并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果。

**先序遍历**  
+ \* \* / A B C D E  
**前缀表示**

EDCBA入栈，AB出栈；  
计算 A/B 记做R1 结果入栈；  
计算 R1\*C 记做R2 结果入栈；  
计算 R2\*D 记做R3 结果入栈；  
计算 R3+E 记做R4 结果入栈；  
结果 R4 出栈。



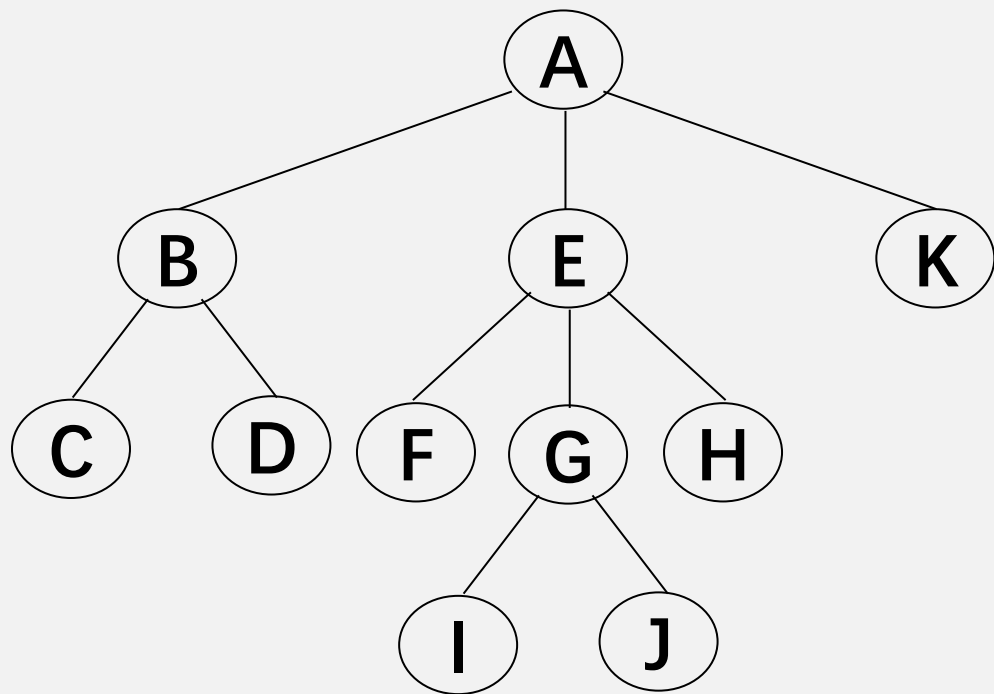
## 哈夫曼树

哈夫曼树又称最优树，是一类带权路径长度最短的树。

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

**路径长度：**结点路径上的分支数目称为路径长度。

**树的路径长度：**从树根到每一个结点的路径长度之和。



**A到F：** 结点路径 AEF ；

**A到F路径长度：** 2 ；

**树的路径长度：**  $3 \times 1 + 5 \times 2 + 2 \times 3 = 19$

**结点的带权路径长度：**从结点到树根之间的路径长度与结点权(值)的乘积。

**权(值)：**各种开销、代价、频度等的抽象称呼。

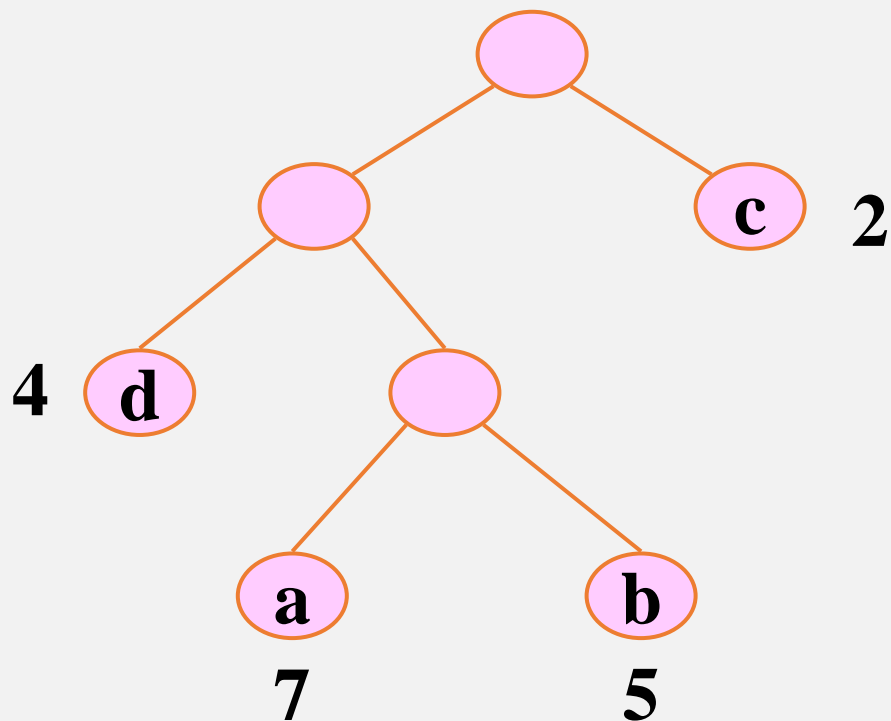
**树的带权路径长度：**树中所有叶子结点的带权路径长度之和，记做：

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

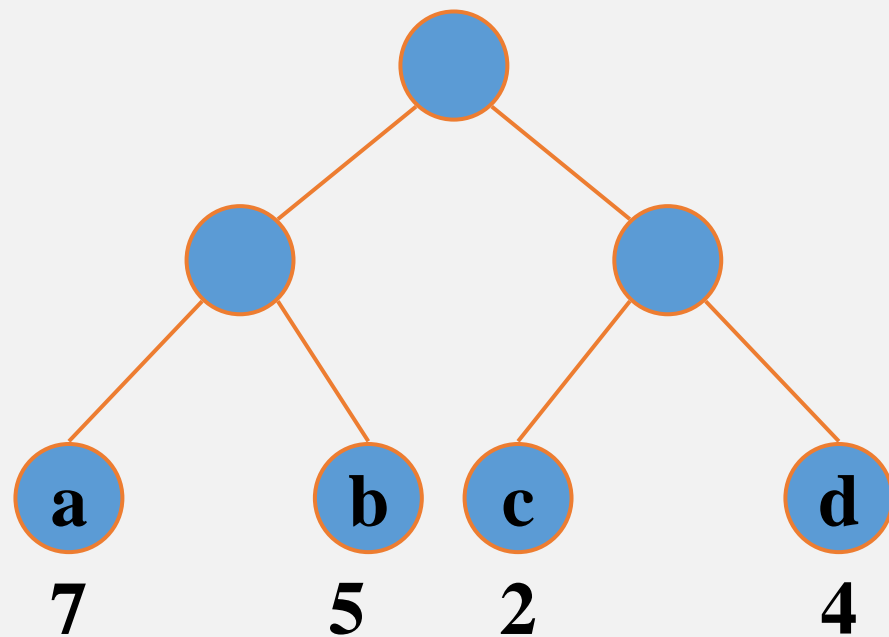
- ◆  $n$ 为叶子结点个数；
- ◆  $w_i$ 为第 $i$ 个叶子结点的权值；
- ◆  $l_i$ 为第 $i$ 个叶子结点的路径长度。

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

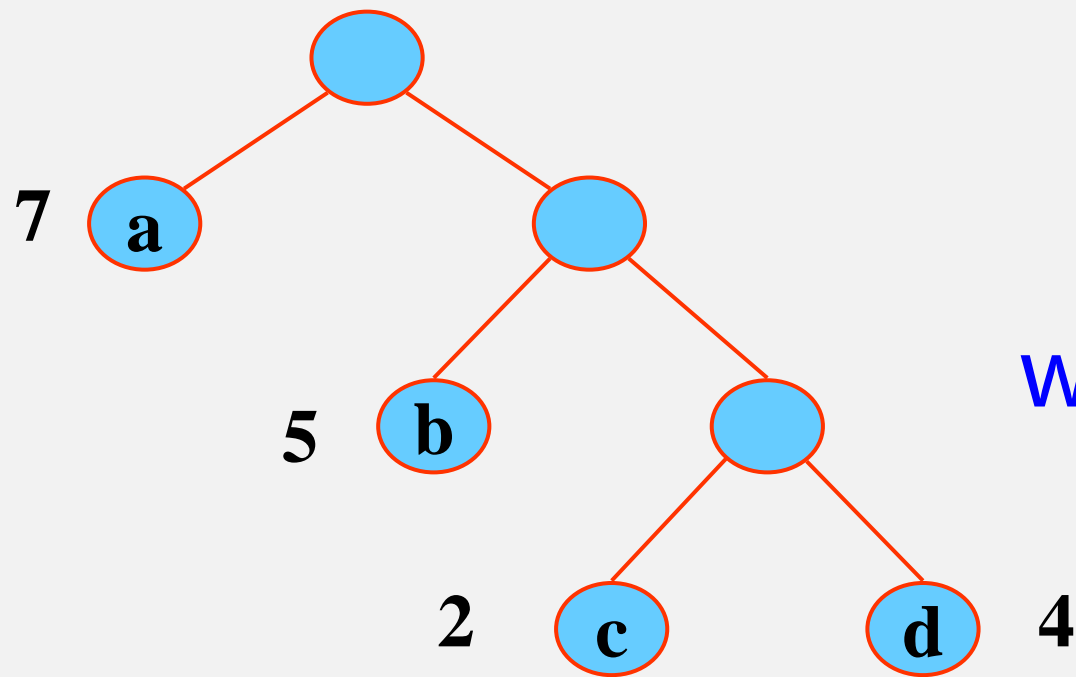
例如：有4个节点a,b,c,d，对应的权值分别为7、5、2、4。以这4个节点作为叶子节点构造二叉树如下。



$$WPL=7*3+5*3+2*1+4*2=46$$



$$WPL=7*2+5*2+2*2+4*2=36$$



WPL值最小，可以证明是哈夫曼树。

$$WPL=7*1+5*2+2*3+4*3=35$$

## 哈夫曼编码

在数据通讯中，常需要将传送的文字转换成01代码传输。为提高速度要求电文编码要尽可能地短，就要设计长短不等的编码：

- ◆ 高概率数据编码短，低概率数据编码长。
- ◆ 还必须保证任意字符的编码都不是另一个字符编码的前缀。

哈夫曼树可以用来构造编码长度不等且译码不产生二义性的编码

## 构造哈夫曼树

以字符集 $C$ 作为叶子结点，频度集 $W$ 作为结点的权值来构造哈夫曼树。

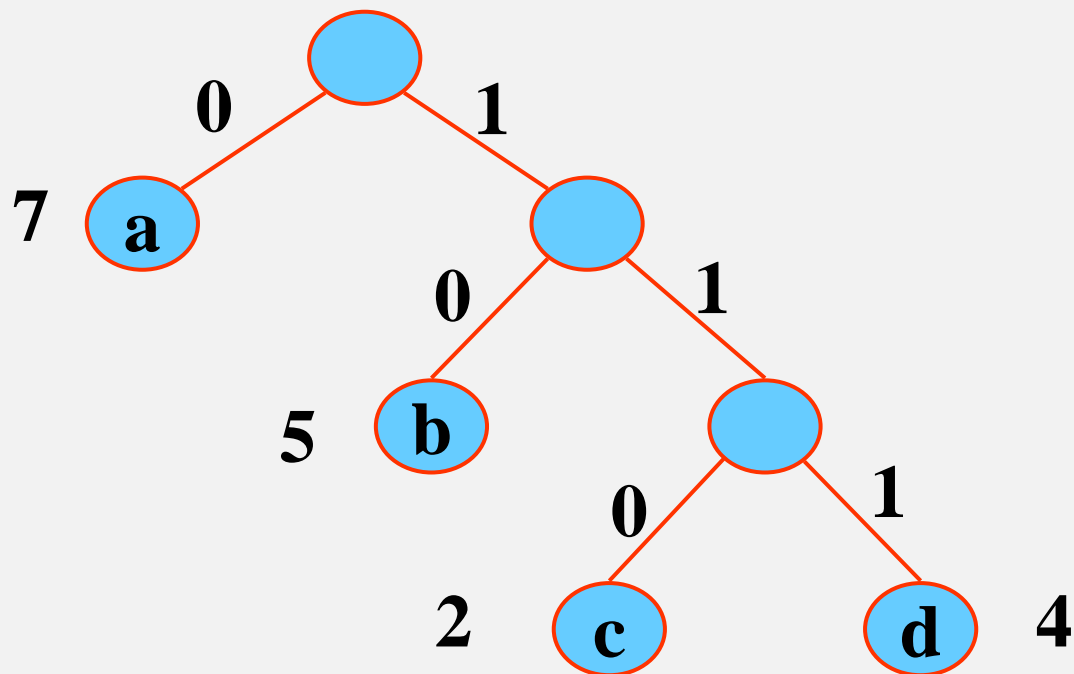
## 规定分支编码

规定哈夫曼树中左分支代表“0”，右分支代表“1”。

## 产生编码

从根结点到每个叶子结点所经历路径分支上的“0”或“1”所组成的编码，为该结点的哈夫曼编码。





哈夫曼编码

a: 0

b: 10

c: 110

d: 1111

$$WPL = 7 * 1 + 5 * 2 + 2 * 3 + 4 * 3 = 35$$

高概率数据编码短，低概率数据编码长。  
任意字符的编码都不是另一个字符编码的前缀。

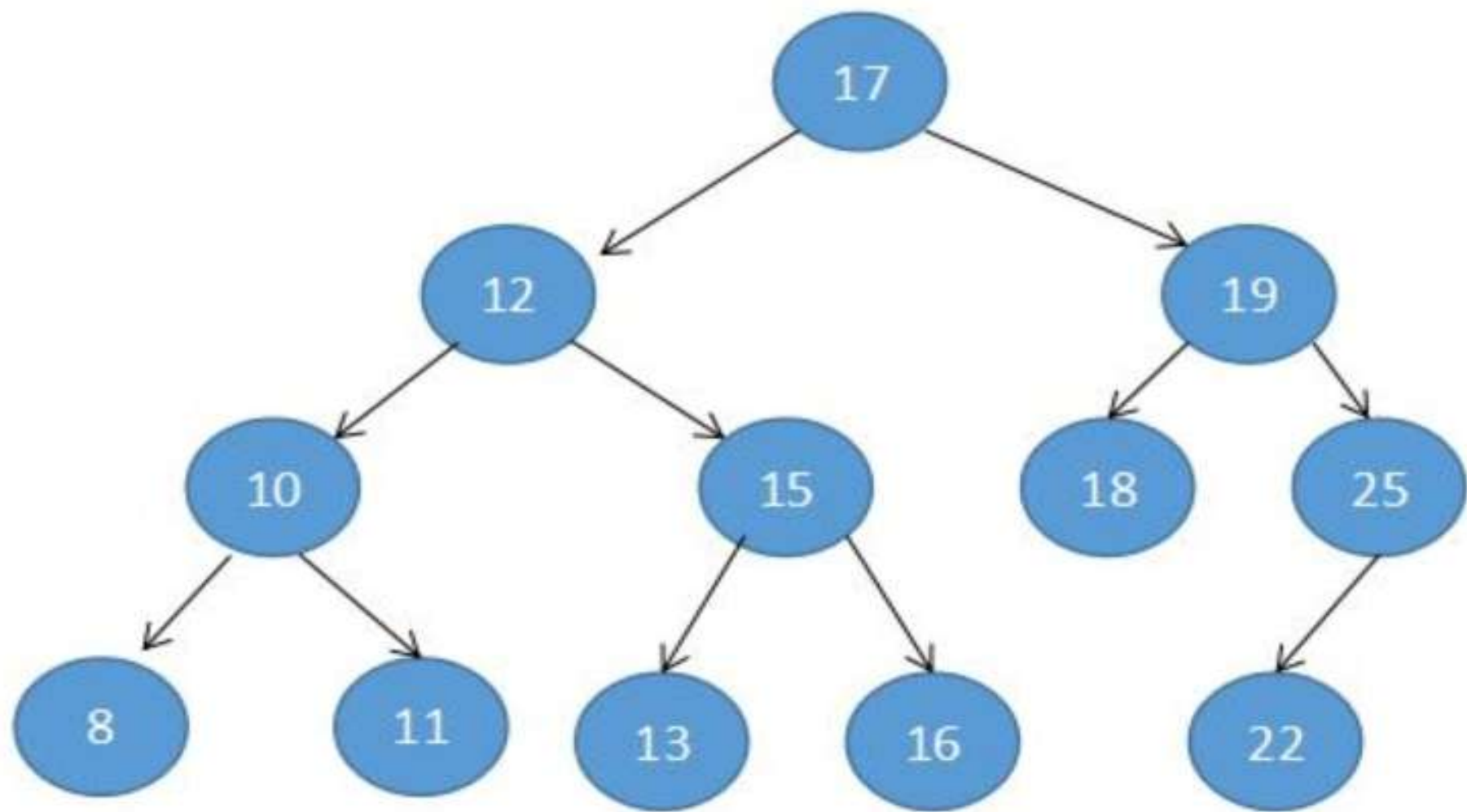


## 3.13 二叉排序树

## 二叉排序树的概念

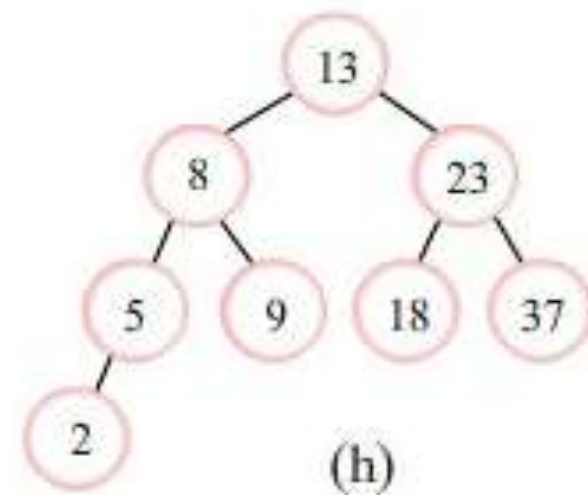
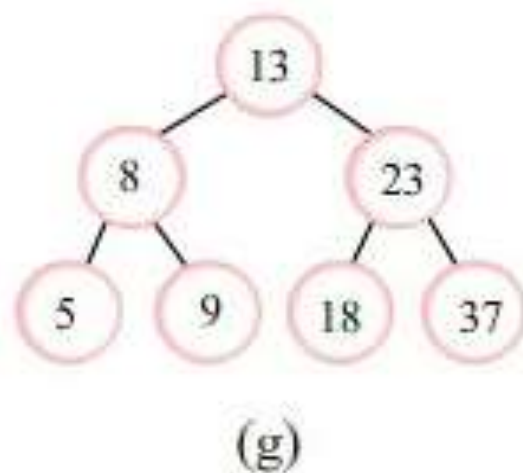
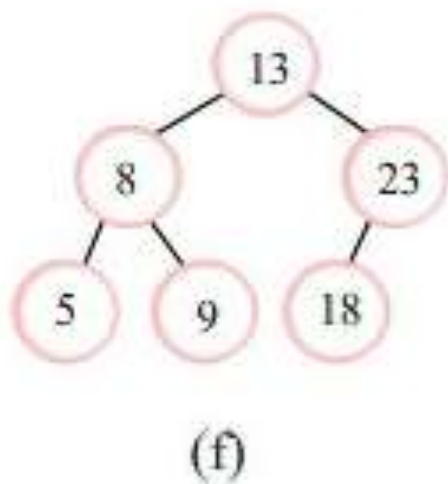
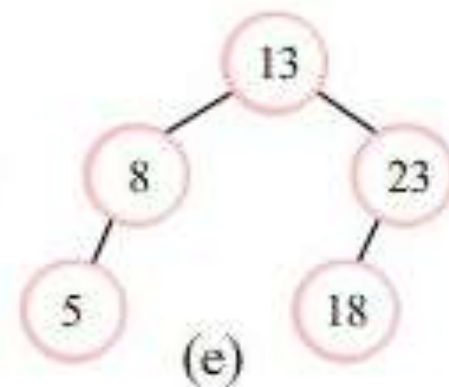
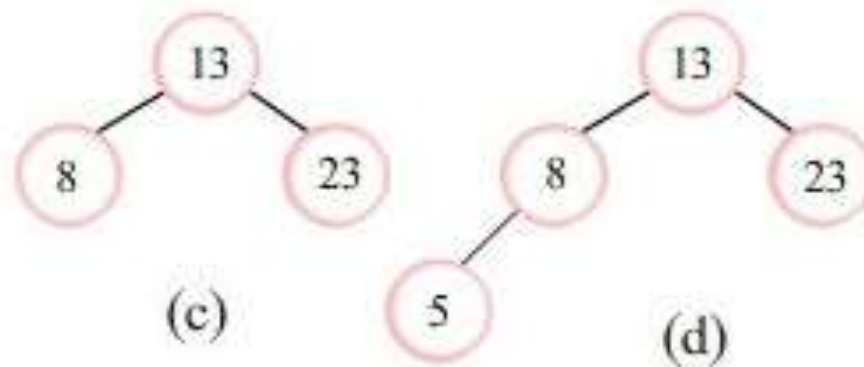
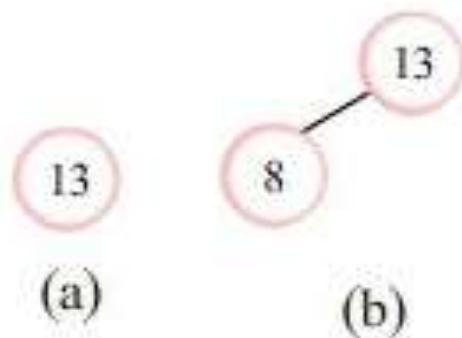
二叉排序树是具备以下特点的二叉树：

- ◆ 若它的左子树不空，则左子树上所有节点的值均小于它根节点的值；
- ◆ 若它的右子树不空，则右子树上所有节点的值均大于其根节点的值。



## 二叉排序树的构造

序列13、8、23、5、18、9、37、2，构造二叉树。

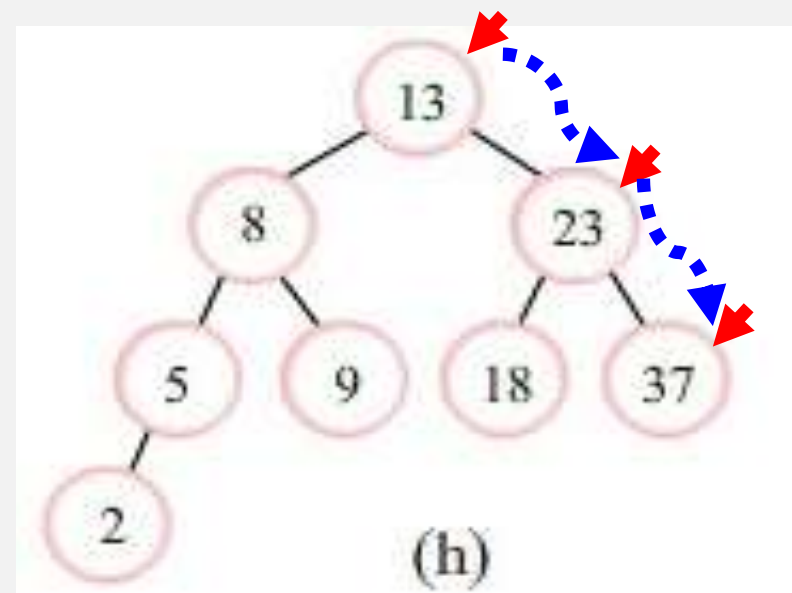
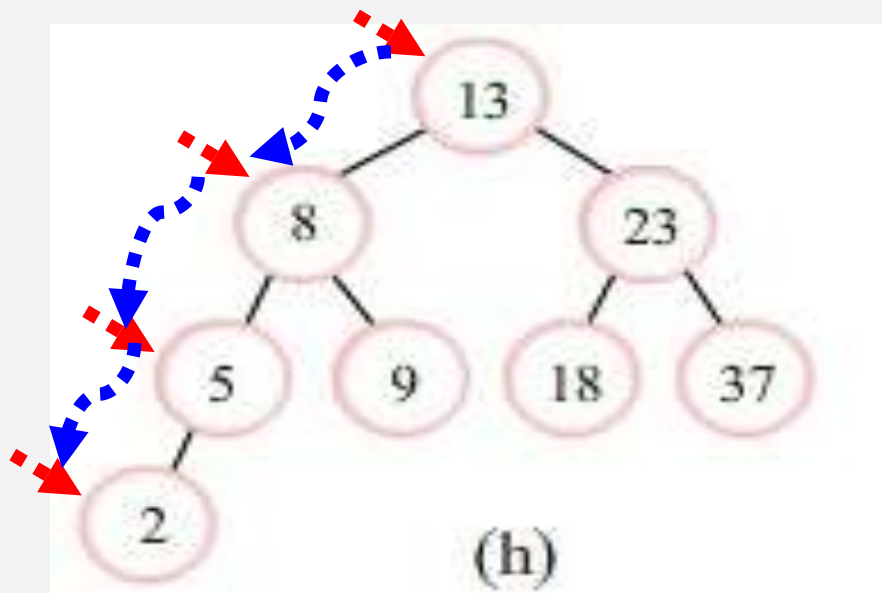


## 二叉排序树的应用

### (1) 数据查找

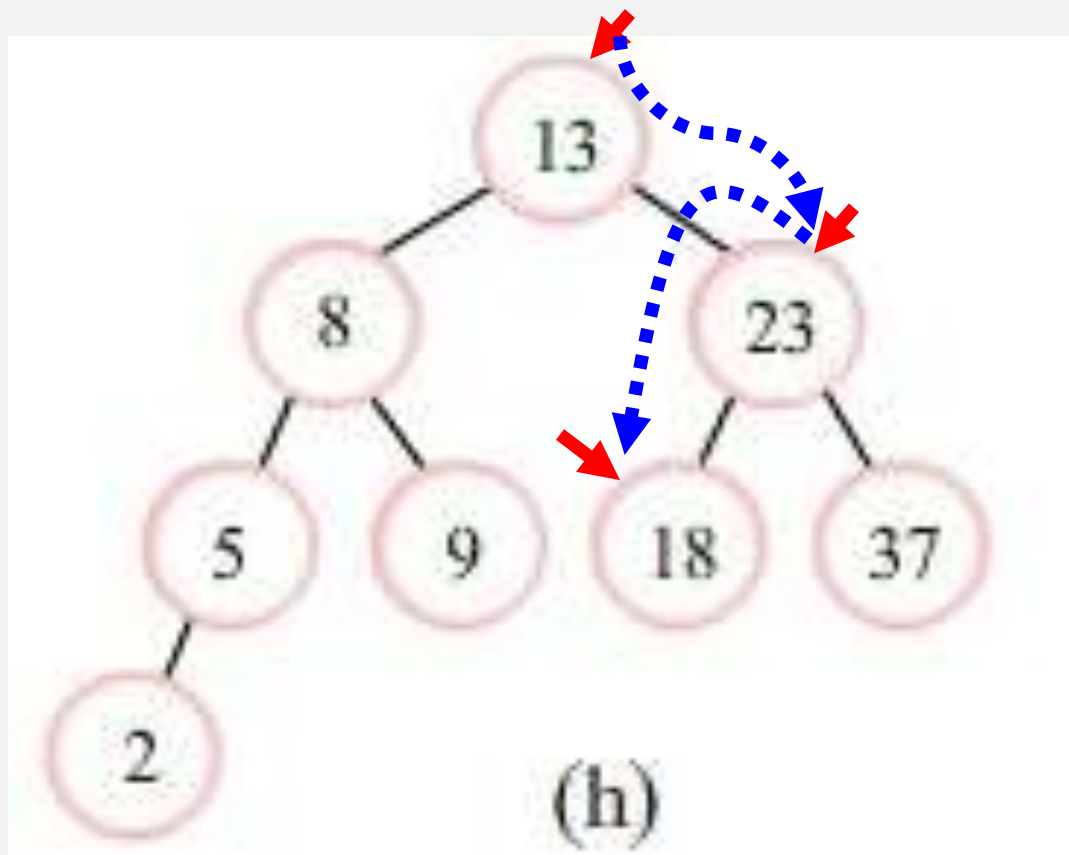
最小元素：从根节点沿左子树，直到找到叶子节点就可得到最小值；

最大元素：从根节点沿右子树，直到找到叶子节点就可得到最大值。



**特定元素：**从根节点开始，若比根节点小，则在左子树查找；若比根节点大，则在右子树查找；以此类推，直到成功或失败。

例如：查找18。

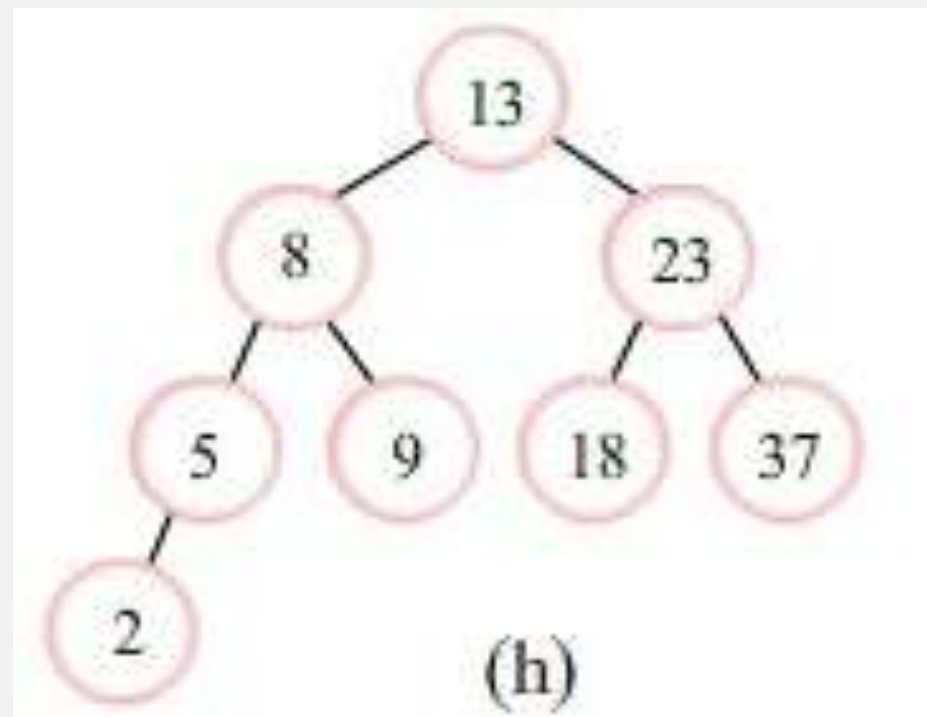


## (2) 数据排序

构造二叉排序树；

中序遍历二叉排序树即得到升序序列。

序列13、8、5、23、18、9、37、2。



中序遍历：2、5、8、9、13、18、23、37。



### (3) 数据统计

统计一本书中，出现过多少个不同的单词，以及每个单词的出现次数。

LC	NUM	WORD	RC
----	-----	------	----

LC：指向左孩子； RC：指向右孩子；

NUM：存储单词出现的次数；

WORD：存储单词。

## 具体过程

(1) 从书中依次读入单词，对每个单词进行如下操作：

在二叉排序树中查找该单词是否存在：

- ◆ 若存在，修改该单词对应节点的NUM域： $NUM=NUM+1$ ；
- ◆ 若不存在，则将该单词插入二叉树，使结果仍为二叉排序树，  
同时 将节点的NUM域设置为1；

(2) 重复 (1) 直到所有单词均处理完；

(3) 按中序遍历输出二叉排序树中每个节点的WORD域和NUM域。



## 3.14 数据查找概述

## 查找的概念

在给定的查找表中找出满足某种条件的结点；  
若存在这样的结点，则查找成功；否则，查找失败。

**查找表：** 是一组待查数据元素的集合。

**性能衡量指标：** 平均查找长度（与关键字进行比较的平均次数）

## 基本形式

两种基本形式：静态查找和动态查找

### 静态查找

查找时只对数据元素进行查询或检索，表内容不变。查找表称为静态查找表

### 动态查找

在查找时，在表中插入中不存在的记录，或删除已存在的某个记录，表内容会发生变化。查找表称为动态查找表。

## 查找表的组织

查找表是记录的集合，元素之间是一种完全松散的关系；

- ◆ 查找表结构灵活，可用多种方式来存储。
- ◆ 存储结构的不同决定了查找方法的不同。

## 四种基本存储结构

根据给定的K值**直接访问**查找表，从而找到要查找的记录。



将给定的K值与查找表中记录的关键字**逐个比较**，找到要查找的记录。

首先根据索引确定待查找记录所在的块，然后再从块中找到要查找的记录。

## 基本查找方法

### 顺序查找

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；若扫描完整个表，仍没有找到相应的记录，则查找失败。

静态查找

顺序表、链表



## 折半查找

折半查找又称为二分查找，要求查找表中的所有记录按关键字有序(升序或降序)。查找时，先和查找表最中间位置的元素进行比较，确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

静态查找

顺序表

## 分块查找

分块查找又称索引顺序查找，是顺序比较和折半查找方法的综合。

- ◆ 将查找表分成若干块。块间有序，块内无序。即第 $i+1$ 块的所有记录关键字均大于(或小于)第 $i$ 块记录的关键字；
- ◆ 在查找表的基础上附加一个索引表，索引表按关键字有序。

先确定待查记录所在块（顺序或折半），再在块内查找(顺序查找)。

静态查找

索引查找表

## 二叉排序树查找

将给定的K值与二叉排序树根结点的关键字进行比较：

- ◆ 若**相等**： 则查找成功；
- ◆ 若给定的K值**小于**二叉排序树根结点的关键字： 继续在该结点的**左子树**上进行查找； 查找失败，插入该节点（**仍是二叉排序树**）。
- ◆ 若给定的K值**大于**二叉排序树根结点的关键字： 继续在该结点的**右子树**上进行查找。 查找失败，插入该节点（**仍是二叉排序树**）。

动态查找

二叉链表

## 散列表查找

设计一种记录存储地址和它关键字之间的确定对应关系，根据该关系构造查找表（散列表）；查找时，直接定位元素的存储位置。

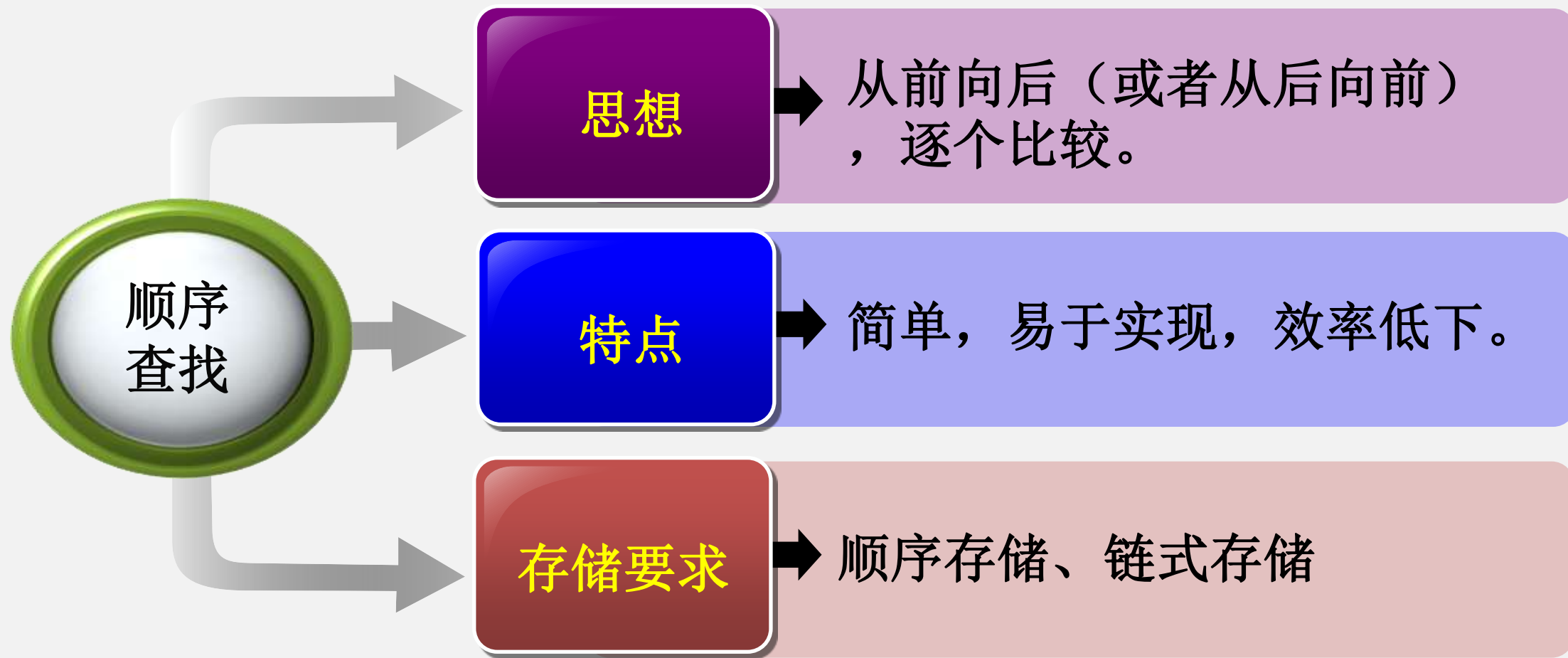
静态查找

散列表

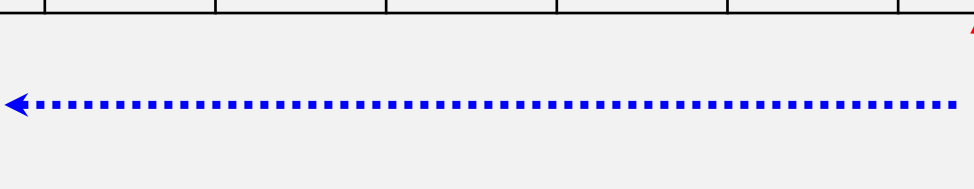


## 3.15 顺序比较与折半查找

## 顺序查找



0	1	2	3	4	5	6	7	8	9
	45	6	38	72	16	23	87	65	39



**特点：**

**n个元素，所需空间n+1;**

**元素存于a[1]-----a[n];**

**a[0]为监视哨，为查找失败标志；**

**查找时，从后向前比较；**

```
int p ;
```

```
a[0]=key ;
```

```
/* 设置监视哨兵,失败返回0 */
```

```
for (p=9; a[p]!=key; p--);
```

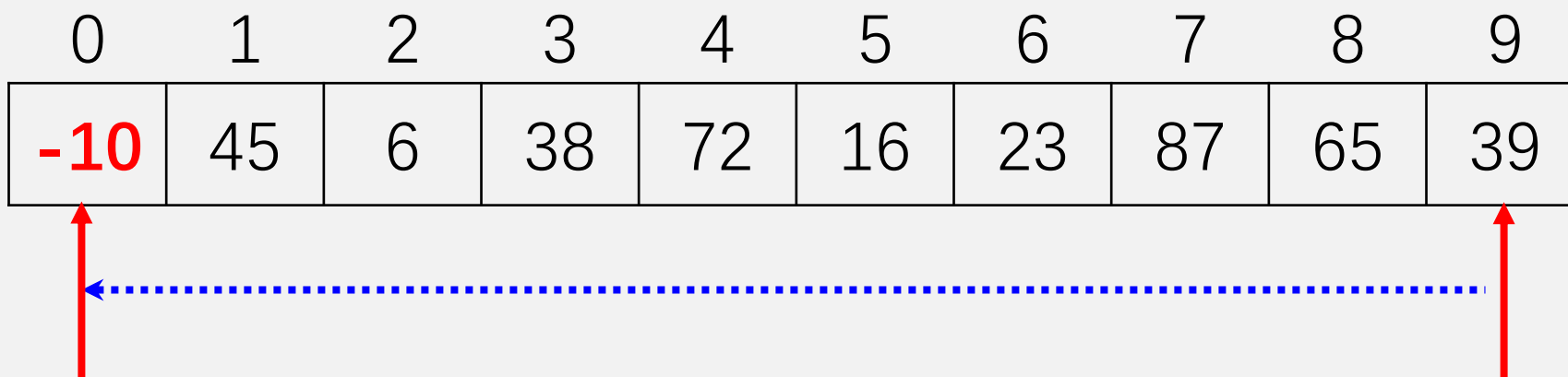
```
return(p) ;
```

0	1	2	3	4	5	6	7	8	9
38	45	6	38	72	16	23	87	65	39

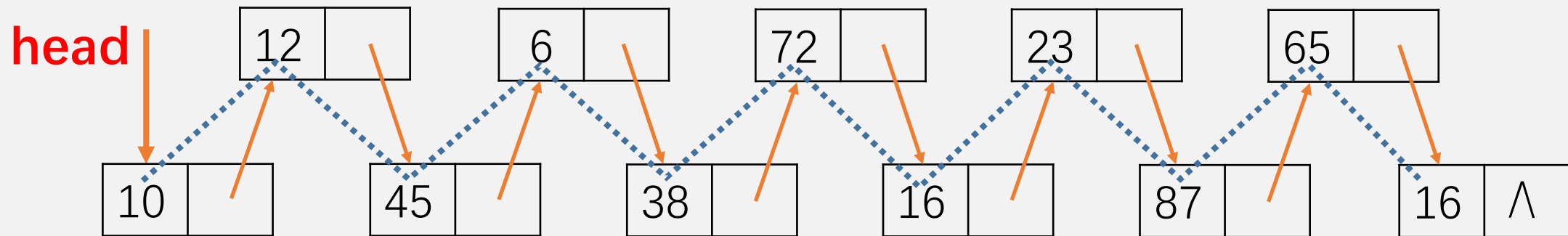
查找38，结果为3；



0	1	2	3	4	5	6	7	8	9
-10	45	6	38	72	16	23	87	65	39



查找-10，结果为0；



特点：

链式存储

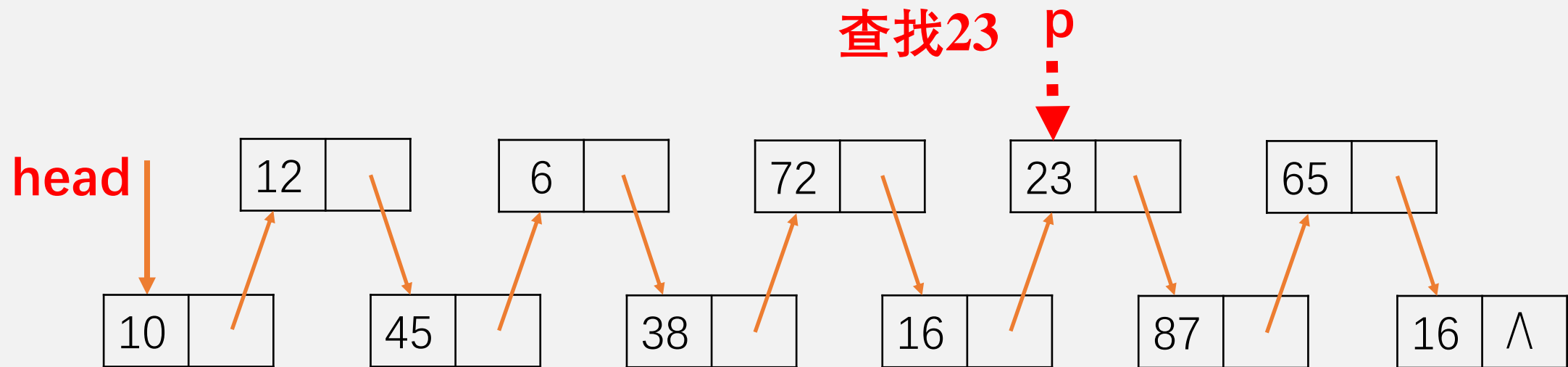
从前向后比较。

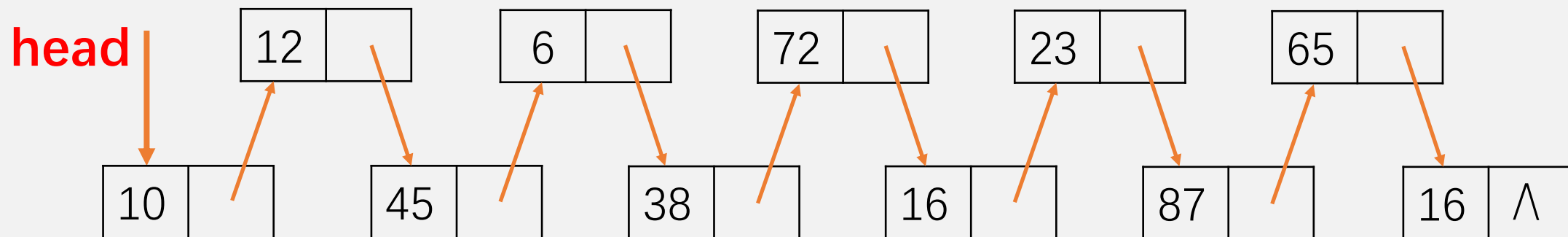
```
p=head;
```

```
While(p!=NULL&& p->data!=key)
```

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

```
return(p) ;
```





查找-10 P:NULL

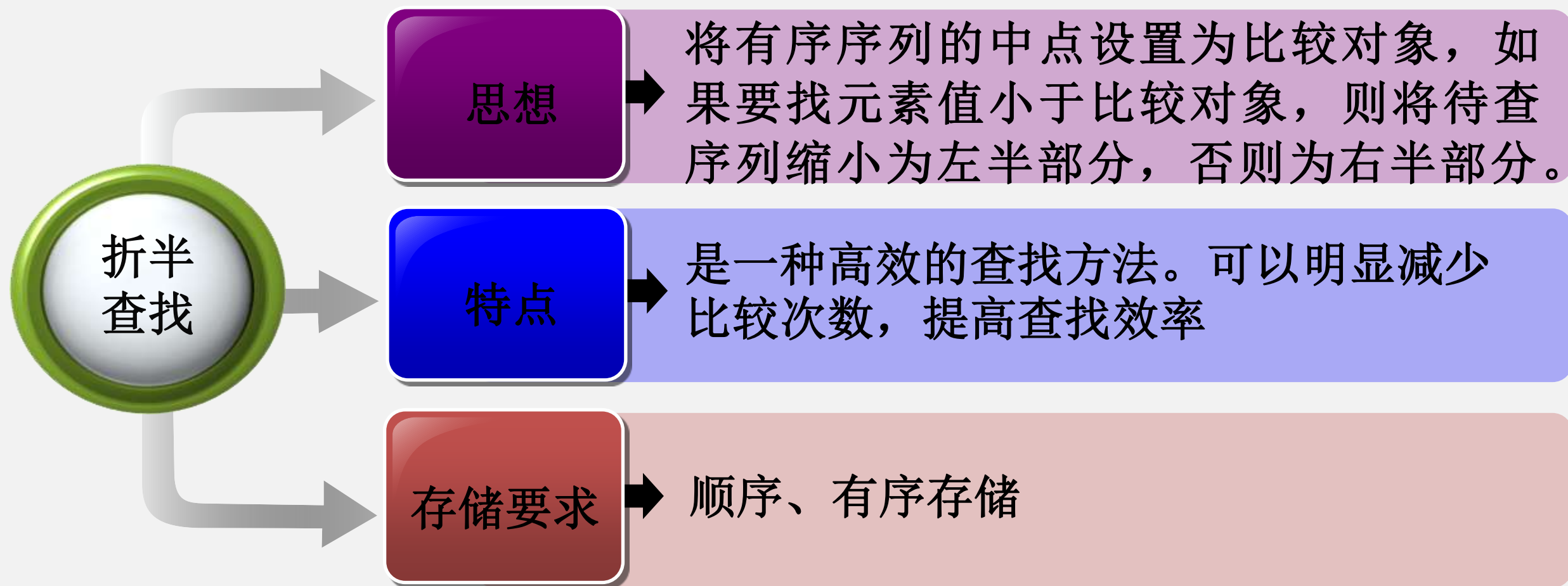
不失一般性，设查找每个记录成功的概率相等，即 $P_i=1/n$ ;

$$ASL = \sum_{i=1}^n P_i \times C_i = \frac{1}{n} \sum_{i=1}^n (n-i+1) = \frac{n+1}{2}$$

若表长为10亿，ASL约为5亿。

时间复杂度为 $O(n)$ ;

## 折半查找



## 算法步骤

- (1) 首先确定整个查找区间的中间位置;  $\text{mid} = (\text{left} + \text{right}) / 2$
- (2) 用待查关键字值与中间位置的关键字值进行比较:
  - ◆ 若相等, 则查找成功;
  - ◆ 若大于, 则确定查找范围为后半区域  $[\text{mid} + 1, \text{right}]$ ;
  - ◆ 若小于, 则确定查找范围为前半区域  $[\text{left}, \text{mid} - 1]$ 。
- (3) 对确定的缩小区域重复 (1)、(2) 步骤;

查找关键字值为87的数据元素

0	1	2	3	4	5	6	7	8	9
12	45	6	38	72	16	23	87	65	39

↑ left                      ↑ mid                      ↑ right

**$87 > 72$**

缩小查找范围为右半部分  **$\text{mid}+1$**  至 **right**



## 查找关键字值为87的数据元素

(1)

0	1	2	3	4	5	6	7	8	9
12	45	6	38	72	16	23	87	65	39

mid

right

2次比较  
成功

(2)

5	6	7	8	9
16	23	87	65	39

left

mid

right

查找时每经过一次比较，查找范围就缩小一半。

一般情况下，表长为  $n$  的折半查找的判定树的深度和含有  $n$  个结点的完全二叉树的深度相同。

$$ASL_{bs} \approx \log_2(n+1) - 1$$

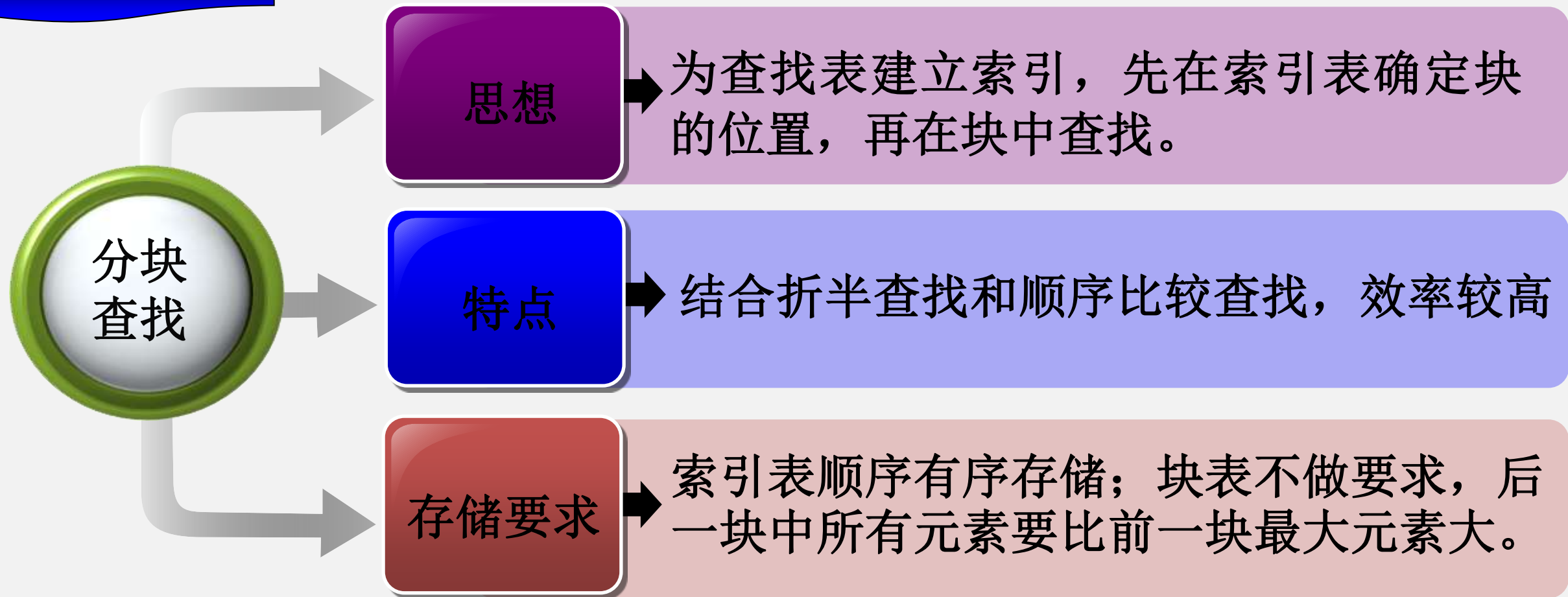
若表长为10亿，ASL约为29。

时间复杂度为  $O(\log_2 n)$



## 3.16 索引查找与散列查找

## 分块查找



12	23	5	6	7	29	43	57	87	36	40	90	120	95	112	93	95	219	223	567	350
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

查找表分成若干块；

块内可无序也可有序，块间有序。

例中，21个元素，每块6个，分4块（最后一块一般不满）

为查找表建立索引表  
每块一个索引项

索引表  
块最大关键字有序

29	90	219	567
0	6	12	18

索引项的构成

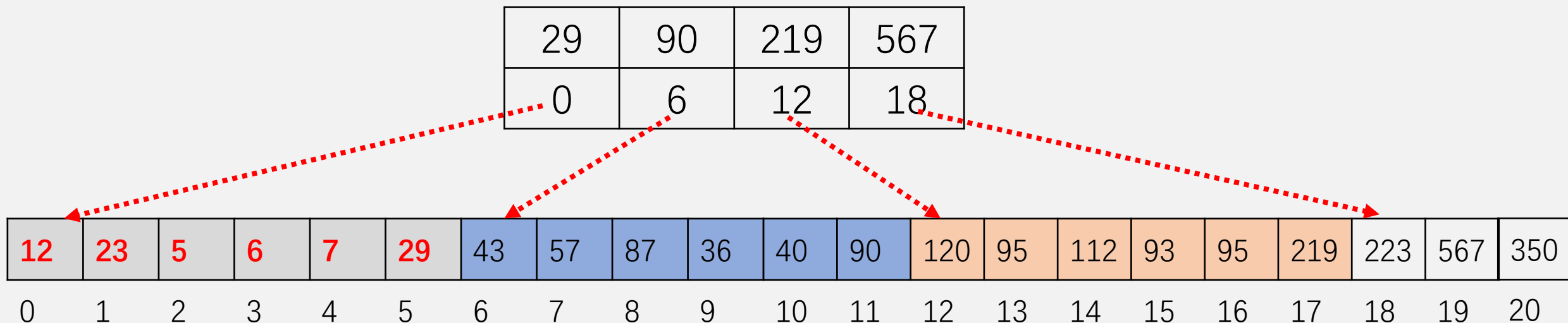
块最大关键字

块起始地址

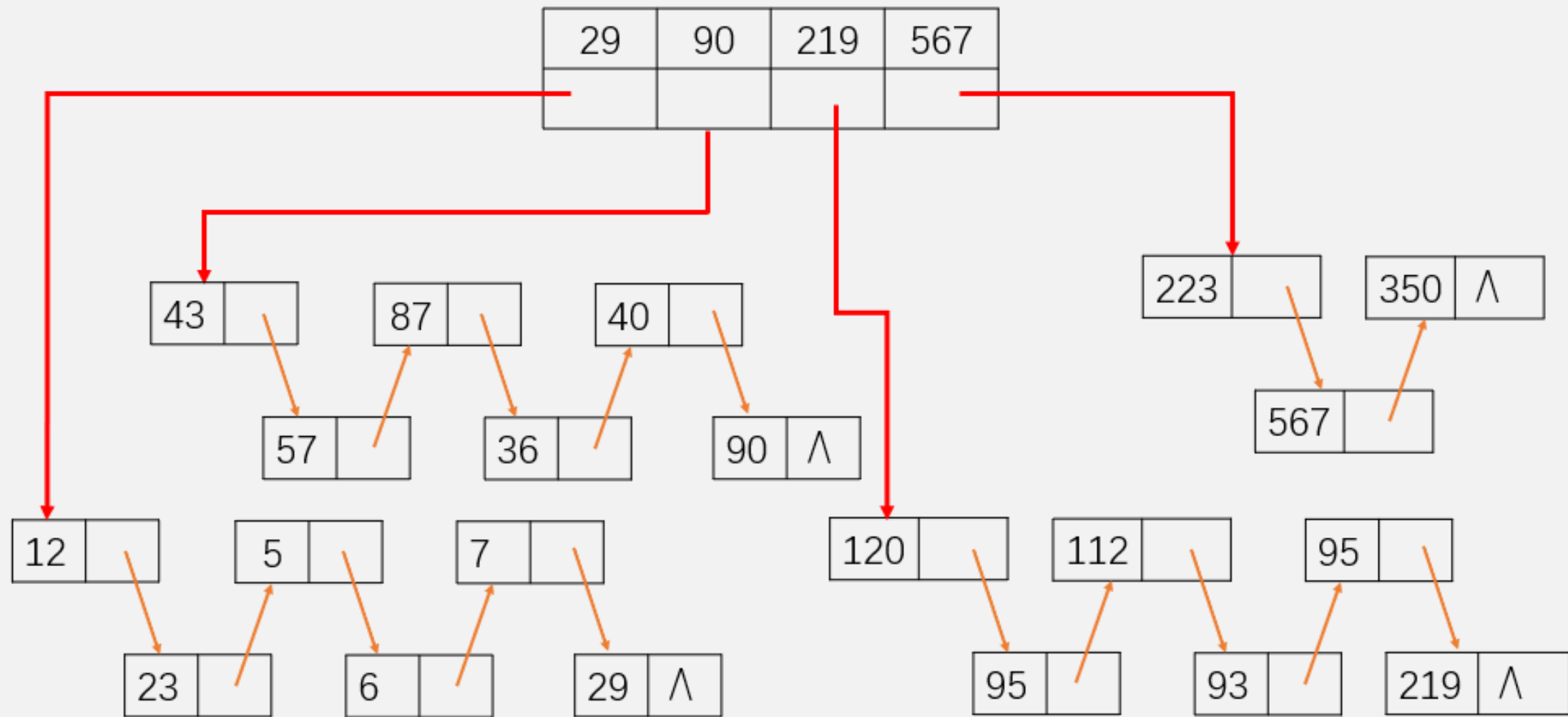
12	23	5	6	7	29	43	57	87	36	40	90	120	95	112	93	95	219	223	567	350
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

## 两种组织方式

有序列 12, 23, 5, 6, 7, 29, 43, 57, 87, 36, 40, 90, 120, 95, 112, 93, 95, 219, 223, 567, 350



查找表采用顺序存储



查找表采用链式存储



## 算法步骤

- (1) 先选取各块中的最大关键字构成一个索引表;
- (2) 查找

分两步:

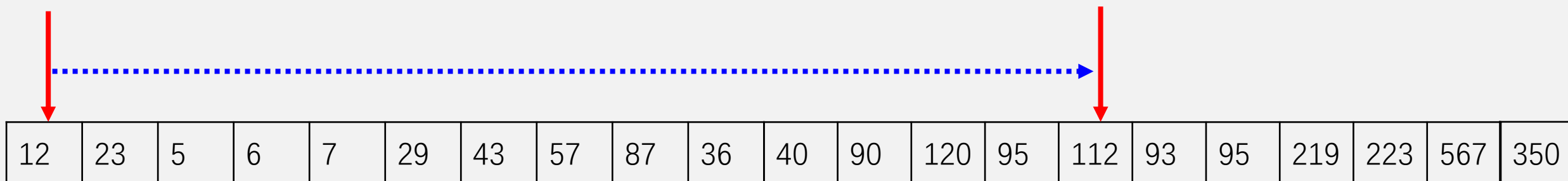
- ①对索引表进行二分查找或顺序查找, 确定待查记录在哪一块;
- ②在已确定的块中用顺序法进行查找。

有序列 12, 23, 5, 6, 7, 29, 43, 57, 87, 36, 40, 90, 120, 95, 112, 93, 95, 219, 223, 567, 350

查找112

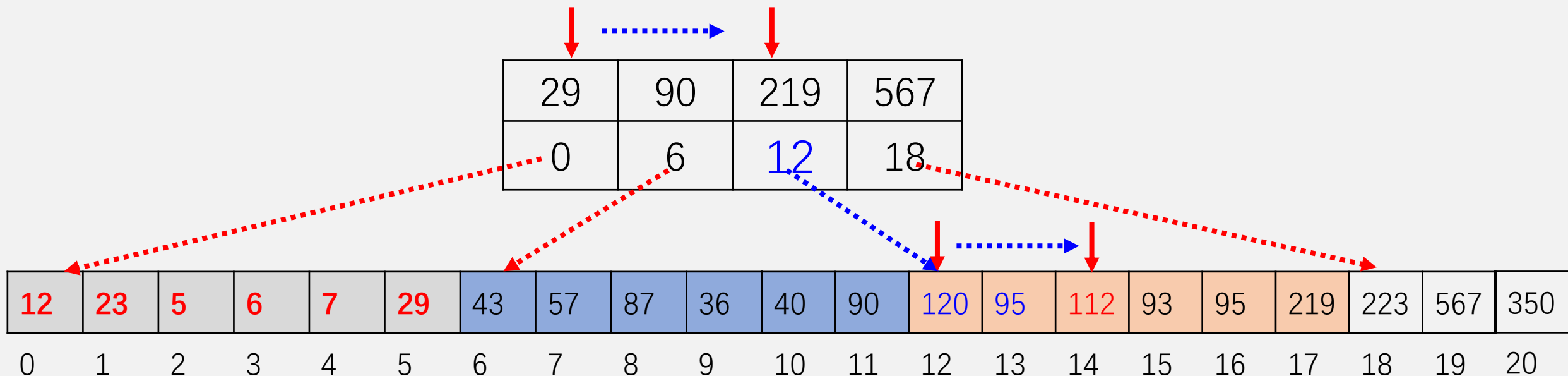
顺序  
比较

从前向后依次比较，比较15次



## 分块查找112

- (1) 先在索引表中找块,  $<219$ , 比较3次;
  - (2) 再在块中顺序比较, 比较3次。合计6次。
- 当问题规模 $n$ 较大时, 效率比顺序比较明显提高。



## 散列表查找

散列表  
查找

思想

在记录存储地址和关键字之间建立一个确定的对应关系，直接定位存放位置。

特点

依据关键字直接求取存放位置。

存储要求

顺序存储，元素存储位置由其关键字决定。

## 散列表的构造

### 哈希函数

**哈希函数**：确定记录关键字与记录存储地址之间对应关系的函数。

**哈希表**：应用哈希函数，由记录的关键字确定记录在表中的地址，并记录存入该地址，采用此方法构造的表叫**哈希表**，也叫**散列表**。

## 哈希函数的选择

哈希函数设定灵活，只要使**任何关键字的哈希函数值都落在表长允许的范围之内**即可。

**直接定址法：**取关键字或关键字的某个线性函数作哈希地址。

**平方取中法：**将关键字平方后取中间几位作为哈希地址。

**除留余数法：**取关键字被某个数除后所得余数作哈希地址。

## 冲突的解决

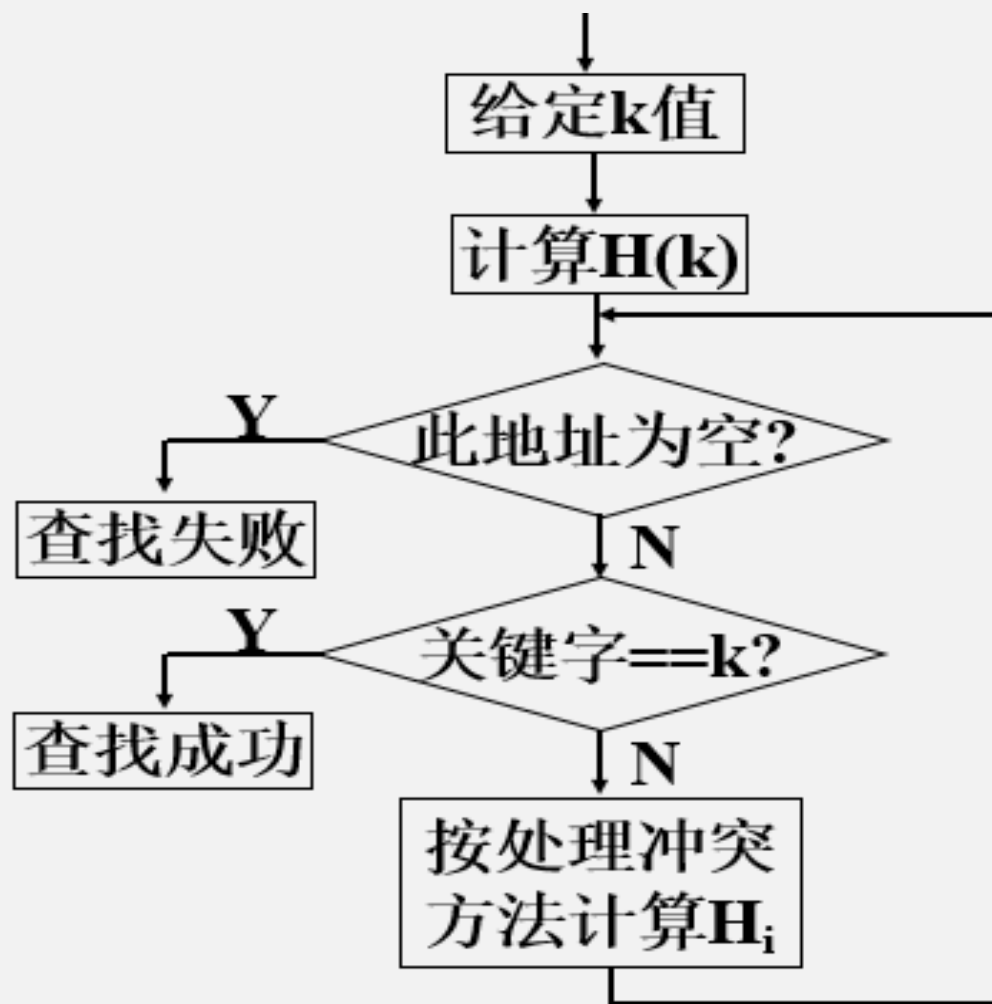
两个关键字的哈希地址相等，或空间已被占用，称为冲突。

**开放定址法：**当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到一个空地址为止。

**线性探测法**是最常用的方法之一（**探测下一个相邻地址**）。

**再哈希法：**构造若干个哈希函数，当发生冲突时，利用不同的哈希函数计算新的哈希地址，直到不发生冲突为止。

## 散列表查找过程



散列表的查找过程



例如，有关键字序列： 67 81 52 73 44 91 27 36 20 39。

存储空间大小： 13

地址计算规则：  $L = K \text{ MOD } 13$ ； 冲突解决机制： 线性探测

	0	1	2	3	4	5	6	7	8	9	10	11	12
散列表	52	91	67	81	27	44	39	20	73		36		
	1	2	1	1	4	1	4	1	1		1		

查找时比较次数

查找成功的  $ASL = 17/10 = 1.7$



## 3.17 插入排序

## 排序的概念

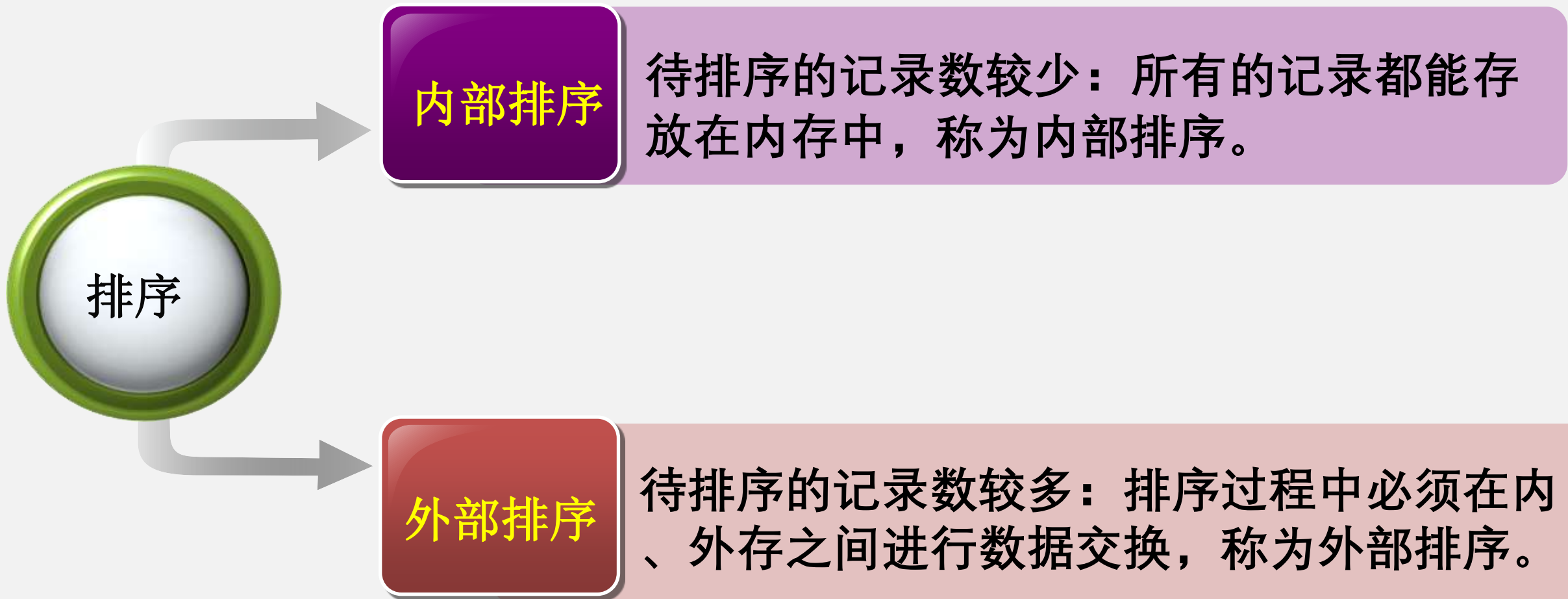
设  $n$  个记录的序列为  $\{ R_1, R_2, R_3, \dots, R_n \}$

相应的关键字序列为  $\{ K_1, K_2, K_3, \dots, K_n \}$

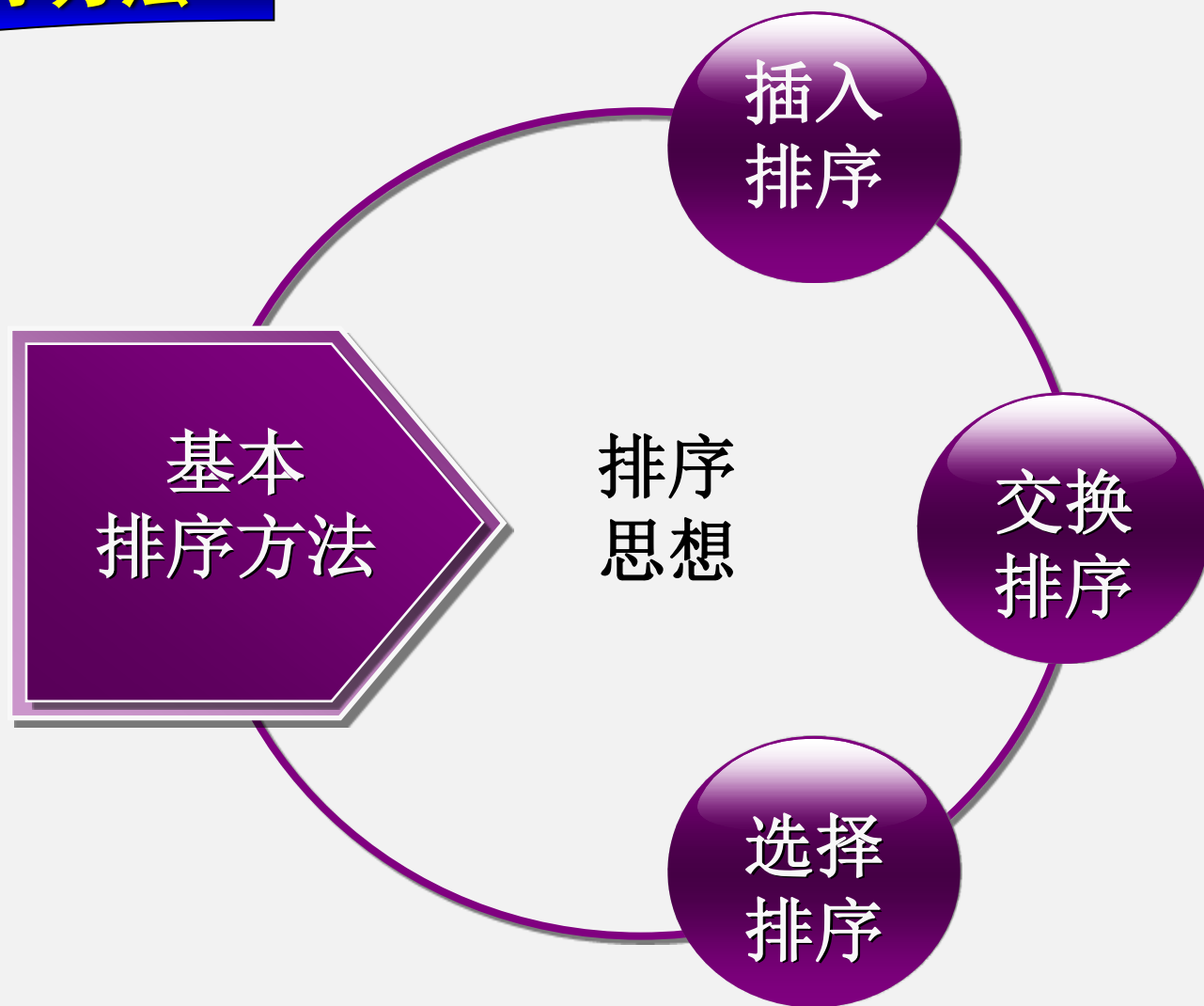
若一个排列  $p_1, p_2, p_3, \dots, p_n$ ，使得相应的关键字满足非递减关系。

将待排序列中元素重新调换位置，使其按关键字有序的过程。

## 排序的分类



## 常见内部排序方法



**稳定排序：**同值关键字排序前后相对位置绝对不变

**不稳定排序：**同值关键字排序前后相对位置可能改变

待排序列	3	15	8	8	6	9	
排序后：	3	6	8	8	9	15	稳定的
排序后：	3	6	8	8	9	15	不稳定

## 插入排序

将关键字的插入到已排序序列的适当位置完成排序。

### 直接插入排序

稳定排序算法

将待排序的记录 $R_i$ ，插入到已排好序的记录表 $R_1, R_2, \dots, R_{i-1}$ 中，得到一个新的、记录数增加1的有序序列。需要 $N-1$ 趟。

### 希尔排序

不稳定排序算法

缩小增量排序：待排序列按增量分组，每组进行直接插入排序；缩小增量后重复该过程，直至有序。

## 直接插入排序

### 思想

每一趟将待排序的记录，按其关键字的大小插入到已排序的适当位置； $N-1$ 趟。

### 特点

方法简单，易于实现

### 存储要求

顺序存储

直接插入排序



## 算法步骤

初始，令第 **1** 个元素作为初始有序表；

依次插入第 **2** , **3** , ..., **k** 个元素，构造新的有序表；

直至最后一个元素；

直接插入排序算法主要应用**比较**和**移动**两种操作。

例如，有待排序列：37，26，30，56，78，28，**37**。

初始：有序序列{37}                      待排序列{**26**，30，56，78，28，**37**}

第1趟：有序序列{**26**，37}                      待排序列{**30**，56，78，28，**37**}

第2趟：有序序列{26，**30**，37}                      待排序列{**56**，78，28，**37**}

第3趟：有序序列{26，30，37，**56**}                      待排序列{78，28，**37**}



第3趟：有序序列{26, 30, 37, 56}                      待排序列{78, 28, 37}

第4趟：有序序列{26, 30, 37, 56, 78}                      待排序列{28, 37}

第5趟：有序序列{26, 28, 30, 37, 56, 78}                      待排序列{37}

第6趟：有序序列{26, 28, 30, 37, 37, 56, 78}                      待排序列{}

每趟有序序列增1，待排序列减1；

N个关键字，需要 $n-1$ 趟；

稳定的排序算法， $O(n^2)$



## 3.18 交换排序和选择排序

## 交换排序

将比较关键字大小，交换位置完成排序

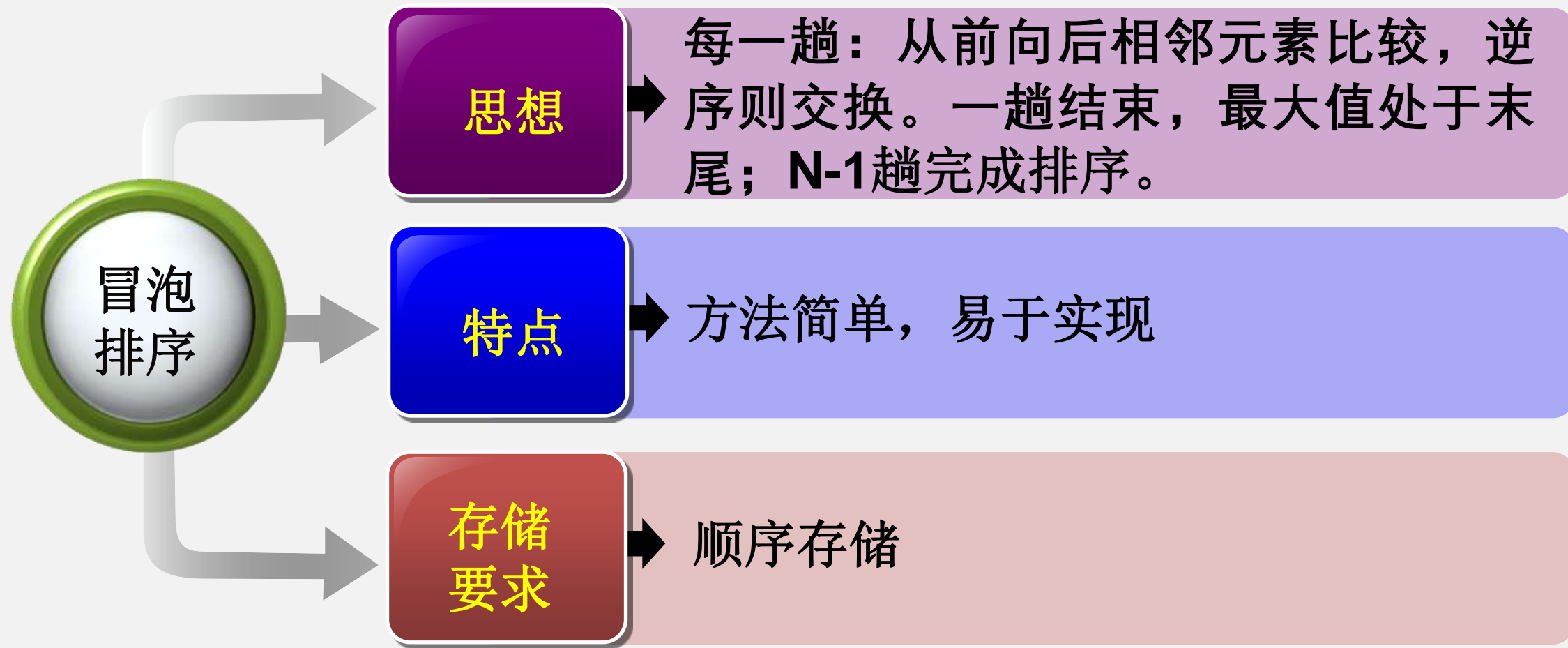
冒泡排序：相邻元素比较交换位置， $O(n^2)$

稳定排序算法

快速排序：一趟排序将待排记录分成两部分，一部分的关键字均比另一部分关键字小。

不稳定排序算法

## 冒泡排序



## 算法步骤

### 第1趟排序：

首先将第一个元素与第二个元素比较大小，若为逆序，则交换；

然后比较第二个元素与第三个元素的大小，若为逆序，则交换；

直至比较第  $n-1$  个元素与第  $n$  个元素的大小，若为逆序，则交换；

结果：

关键字最大的记录被交换至最后一个元素位置上。

重复以上步骤，直到有序。



例如，有序列 23, 14, 78, 65, 26, 98, 73, 18, 14，冒泡排序。

第1趟：

①

23	14	78	65	26	98	73	18	14
----	----	----	----	----	----	----	----	----

②

14	23	78	65	26	98	73	18	14
----	----	----	----	----	----	----	----	----

③

14	23	78	65	26	98	73	18	14
----	----	----	----	----	----	----	----	----

④

14	23	65	78	26	98	73	18	14
----	----	----	----	----	----	----	----	----

⑤

14	23	65	26	78	98	73	18	14
----	----	----	----	----	----	----	----	----

⑥

14	23	65	26	78	98	73	18	14
----	----	----	----	----	----	----	----	----

⑦

14	23	65	26	78	73	98	18	14
----	----	----	----	----	----	----	----	----


⑧

14	23	65	26	78	73	18	98	14
----	----	----	----	----	----	----	----	----

14	23	65	26	78	73	18	14	98
----	----	----	----	----	----	----	----	----


9个数，从前向后8次比较，  
最大值处于末尾。

第2趟:



14	23	26	65	73	18	14	78	98
----	----	----	----	----	----	----	----	----

第3趟:



14	23	26	65	18	14	73	78	98
----	----	----	----	----	----	----	----	----

第4趟:




14	23	26	18	14	65	73	78	98
----	----	----	----	----	----	----	----	----

第5趟:




14	23	18	14	26	65	73	78	98
----	----	----	----	----	----	----	----	----

第6趟：



14	18	14	23	26	65	73	78	98
----	----	----	----	----	----	----	----	----

第7趟：



14	14	18	23	26	65	73	78	98
----	----	----	----	----	----	----	----	----

第8趟：序列有序



14	14	18	23	26	65	73	78	98
----	----	----	----	----	----	----	----	----

**N个数，需要n-1趟才能完成。**

## 时间复杂度

◆ **最好情况（正序）**：比较次数： $n-1$ ；移动次数： $0$ ；

◆ **最坏情况（逆序）**：

比较次数：
$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

移动次数：
$$3 \sum_{i=1}^{n-1} (n-i) = \frac{3n(n-1)}{2}$$

故时间复杂度： $T(n)=O(n^2)$       空间复杂度： $S(n)=O(1)$

## 选择排序

从待排序列中选出极值，存入特定位置。

简单选择排序：选择最小值，放于已排序序列末尾， $O(n^2)$

不稳定排序算法

堆排序：基于完全二叉树，通过构建堆-输出堆顶-调整堆实现排序。

不稳定排序算法

## 简单选择排序



## 算法步骤

第 1 趟:从  $1—n$  个记录中选择关键字最小的记录, 并和第 1 个记录交换。

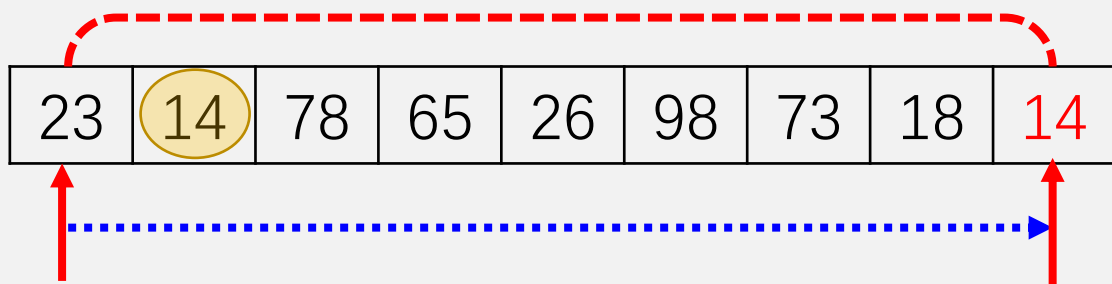
第 2 趟:从  $2—n$  个记录中选择关键字最小的记录, 并和第 2 个记录交换。

第 $n-1$ 趟:

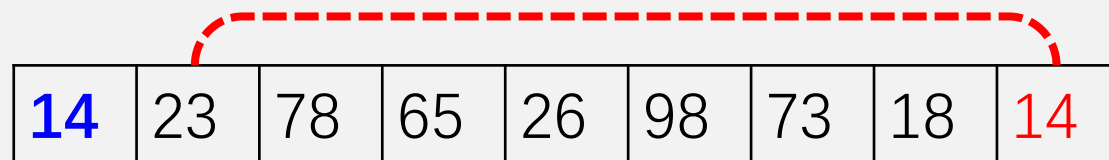
从  $n-1—n$  个记录中选择关键字最小的记录, 并和第  $n-1$  个记录交换。

以序列 23, 14, 78, 65, 26, 98, 73, 18, 14 为例。

第1趟:



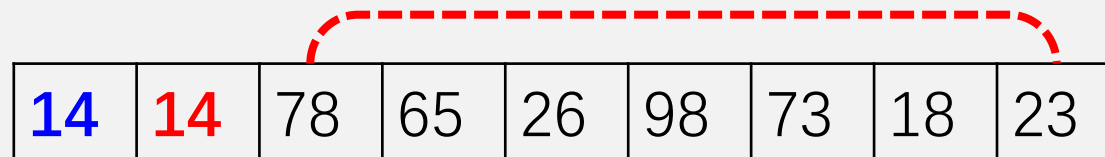
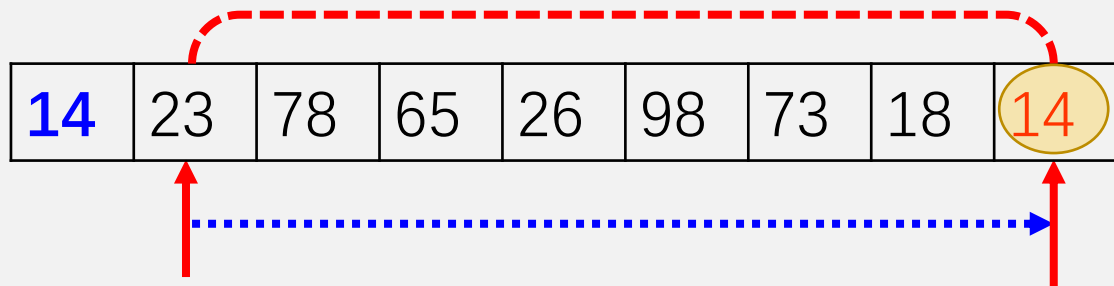
在长度为 $n$ 的待排序列中，从前向后找最小值，需要 $n-1$ 次比较。



找到后交换。待排序列长度-1

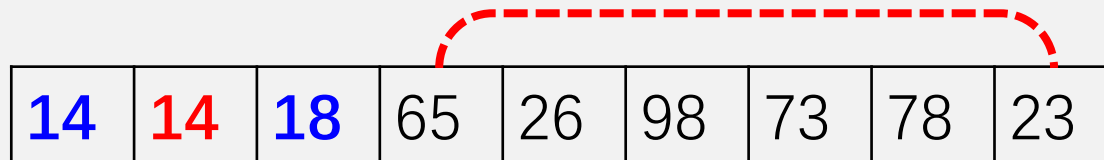
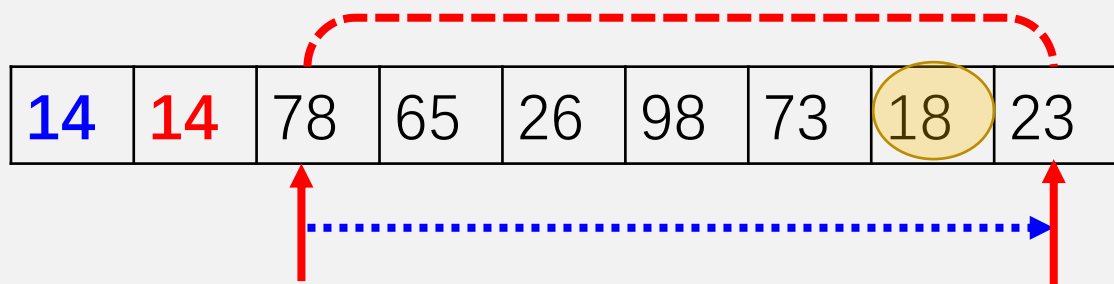


第2趟:



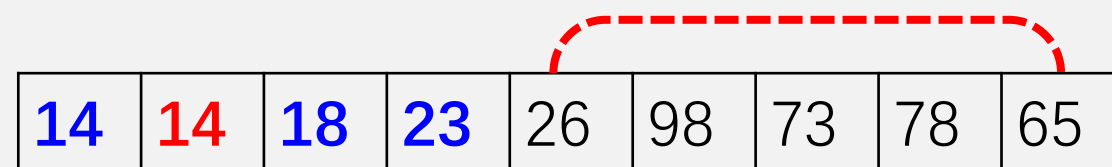
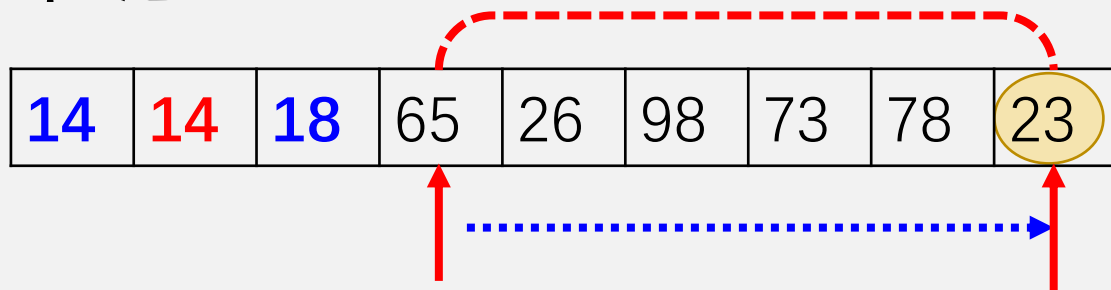
找到后交换。待排序列长度-1

第3趟:



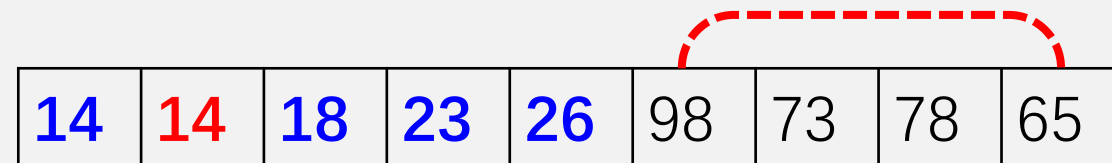
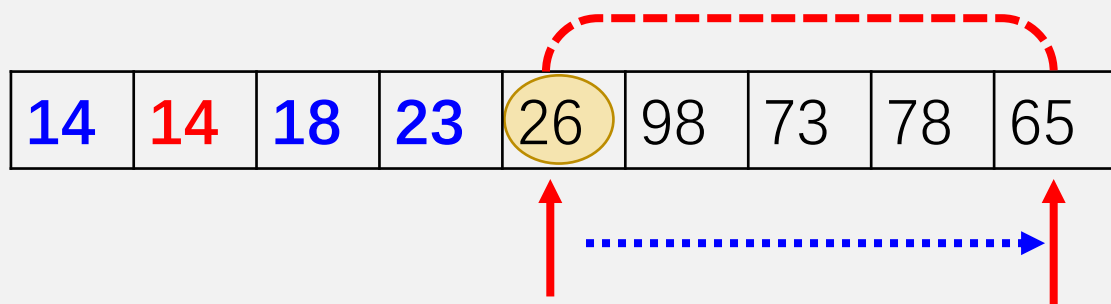
找到后交换。待排序列长度-1

第4趟:



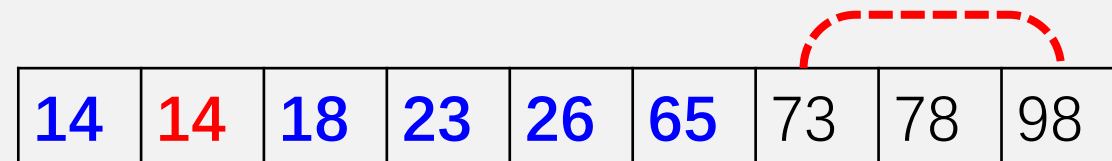
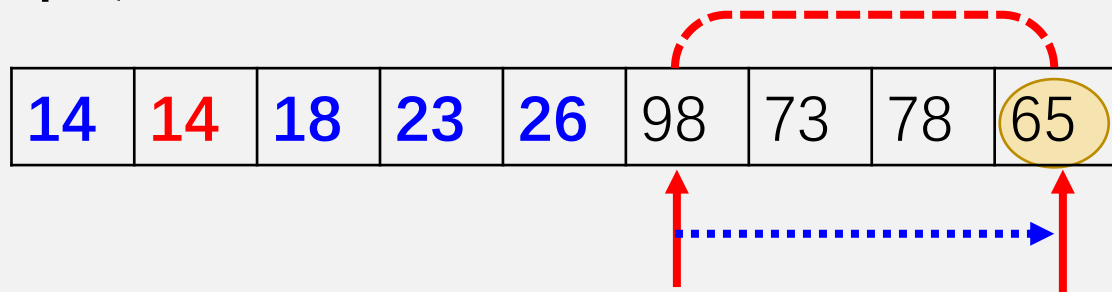
找到后交换。待排序列长度-1

第5趟:



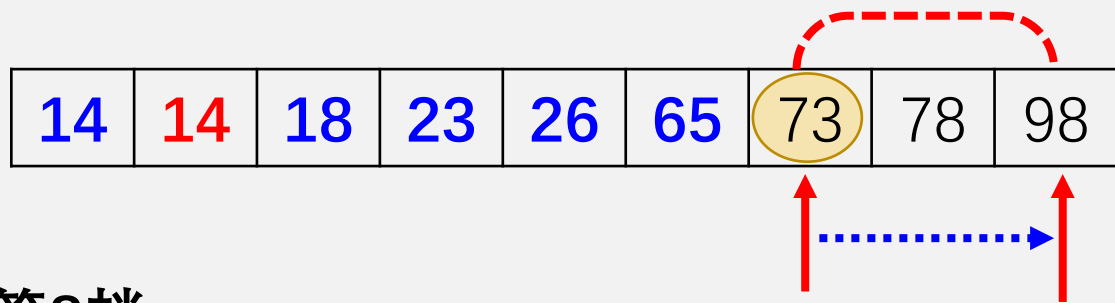
找到后不用交换

第6趟:

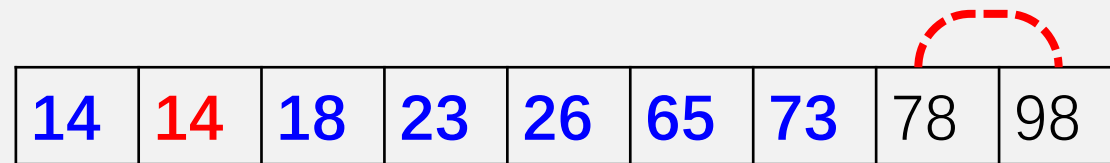
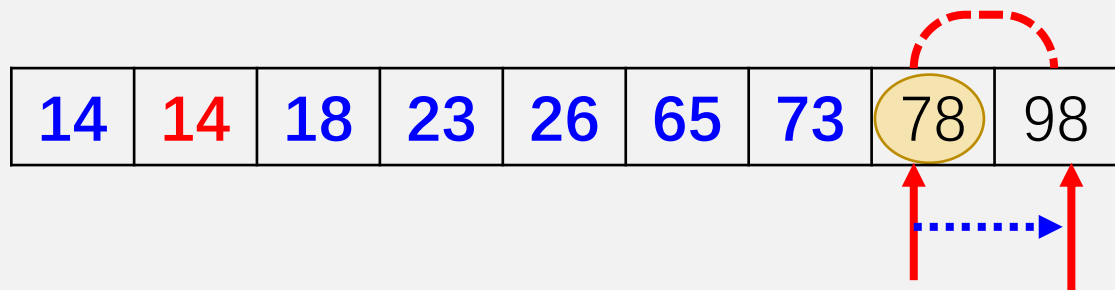


找到后交换。待排序列长度-1

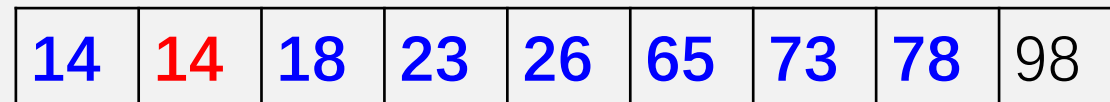
第7趟:



第8趟:



找到后不用交换



找到后不用交换

N个数，需要n-1趟才能完成。

对n个记录进行排序的趟数为n-1趟,进行第i趟排序时,关键字的比较次数为n-i, 则:

$$\text{比较次数: } \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

时间复杂度是:  $T(n)=O(n^2)$ , 空间复杂度是:  $S(n)=O(1)$