数据结构与算法

线性表

张晓平

数学与统计学院

Email: xpzhang.math@whu.edu.cn

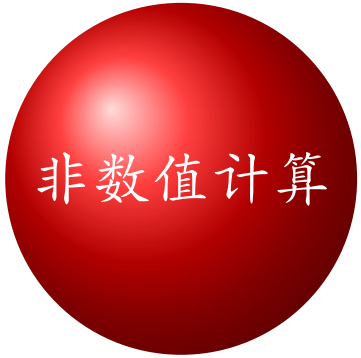
Homepage: http://staff.whu.edu.cn/show.jsp?n=Zhang%20Xiaoping

目录

If you give someone a program, you will frustrate them for a day; if you teach them how to program, you will frustrate them for a lifetime.

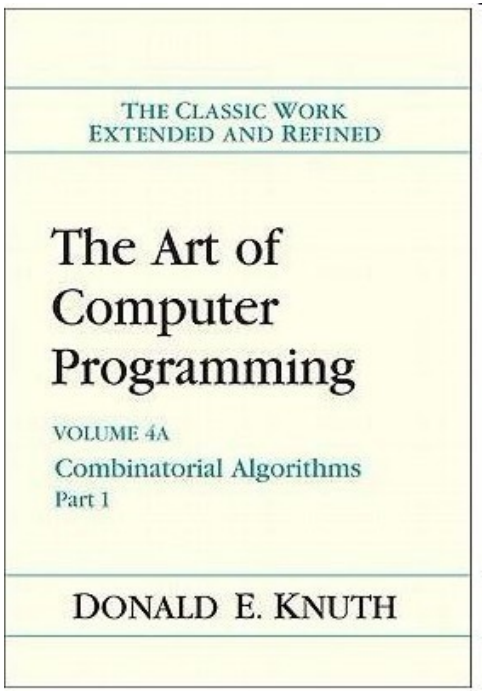
图：参考用书

1 线性表的链式存储



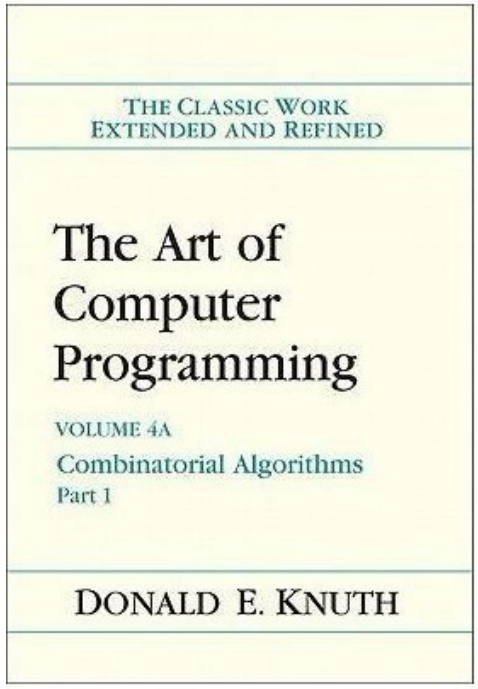
数据结构是一门研究非数值计算的程序设计问题中的操作对象, 以及它们之间的关系和操作等相关问题的学科.

(a) Donald Knuth(1938-)



(b) The Art of Computer Programming

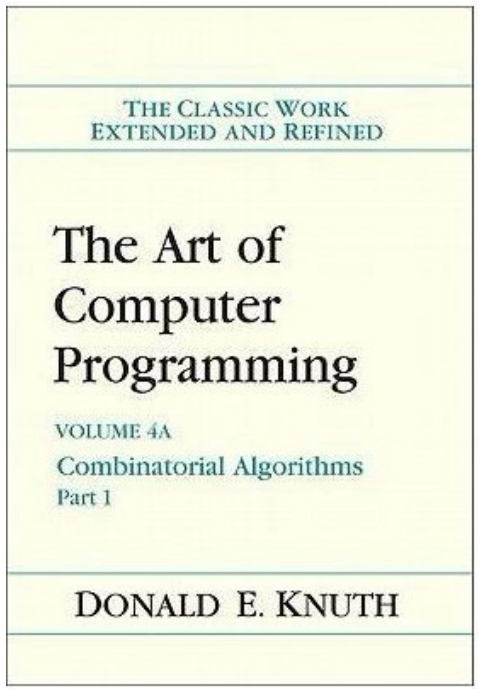
(a) Donald Knuth(1938-)



(b) The Art of Computer Programming

1968年, Donald教授较系统地阐述了数据的逻辑结构和存储结构及其操作, 开创了数据结构的课程体系.

(a) Donald Knuth(1938-)



(b) The Art of Computer Programming

1968年, Donald教授较系统地阐述了数据的逻辑结构和存储结构及其操作, 开创了数据结构的课程体系.

程序设计 = 好的结构 + 好的算法

描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合.

描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合.

数据包括

- 整型、实型等数值类型;
- 字符及声音、图像、视频等非数值类型.

数据

描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合.

数据包括

- 整型、实型等数值类型；
- 字符及声音、图像、视频等非数值类型.

数据就是符号，必须满足：

- 可以输入到计算机中；
- 能被计算机程序处理.

组成数据的、有一定意义的基本单位，在计算机中通常作为整体来处理.

数据元素

组成数据的、有一定意义的基本单位，在计算机中通常作为整体来处理.

- 在人类中，数据元素就是人;
- 在畜类中，数据元素为牛、马、猪、狗、羊等.

一个元素可由若干个数据项组成.

一个元素可由若干个数据项组成.

如: 人可以有耳、鼻、眼、嘴、手、脚等这些数据项，也可以有姓名、年龄、性别、出生地址、联系电话等数据项.

一个元素可由若干个数据项组成.

如: 人可以有耳、鼻、眼、嘴、手、脚等这些数据项，也可以有姓名、年龄、性别、出生地址、联系电话等数据项.

数据项是数据不可分割的最小部分.

性质相同的数据元素的集合，它是数据的子集。

数据对象

性质相同的数据元素的集合，它是数据的子集。

所谓性质相同指的是数据元素具有相同数量和类型的数据项。

实际应用中，处理的数据元素通常具有相同性质，在不产生混淆的情况下，将数据对象简称为数据。

相互之间存在一种或多种特定关系的数据元素的集合.

数据结构

相互之间存在一种或多种特定关系的数据元素的集合.

计算机中，数据元素并不是孤立、杂乱无序的，而是具有内在联系的;

数据元素之间存在的一种或多种特定关系，就是数据的组织形式.

1 线性表的链式存储

数据对象中数据元素之间的相互关系。

逻辑结构

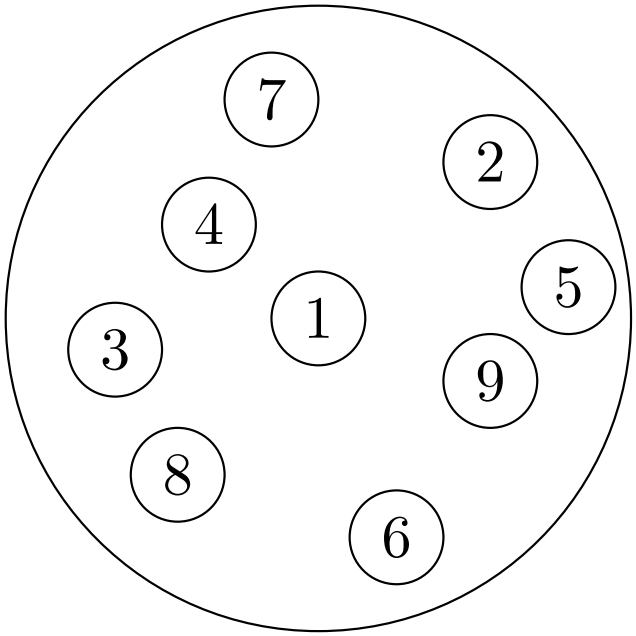
数据对象中数据元素之间的相互关系。

- 集合结构
- 线性结构
- 树形结构
- 图形结构

其中的元素除了同属于一个集合外，之间没有其他关系.

集合结构

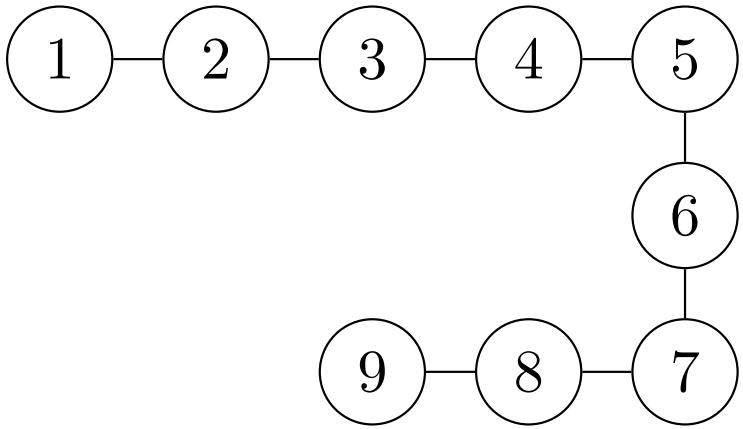
其中的元素除了同属于一个集合外，之间没有其他关系.



其中的数据元素之间是一对一的关系.

线性结构

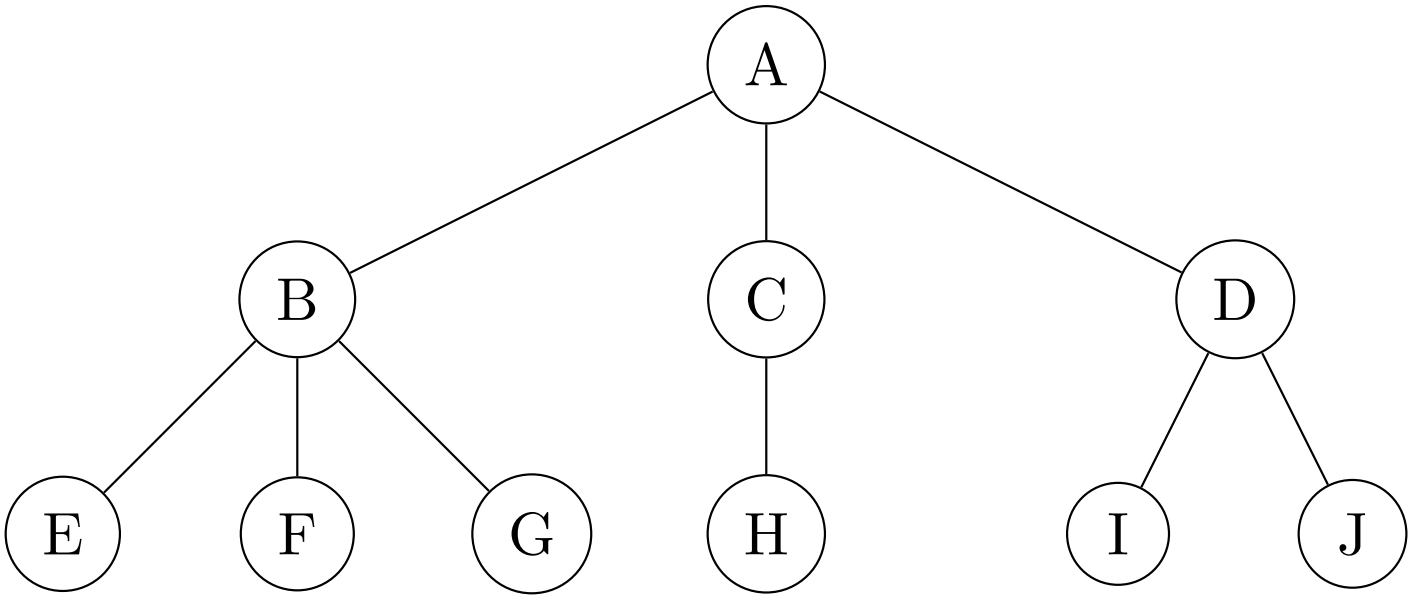
其中的数据元素之间是一对一的关系.



其中的数据元素之间是一对多的层次关系.

树形结构

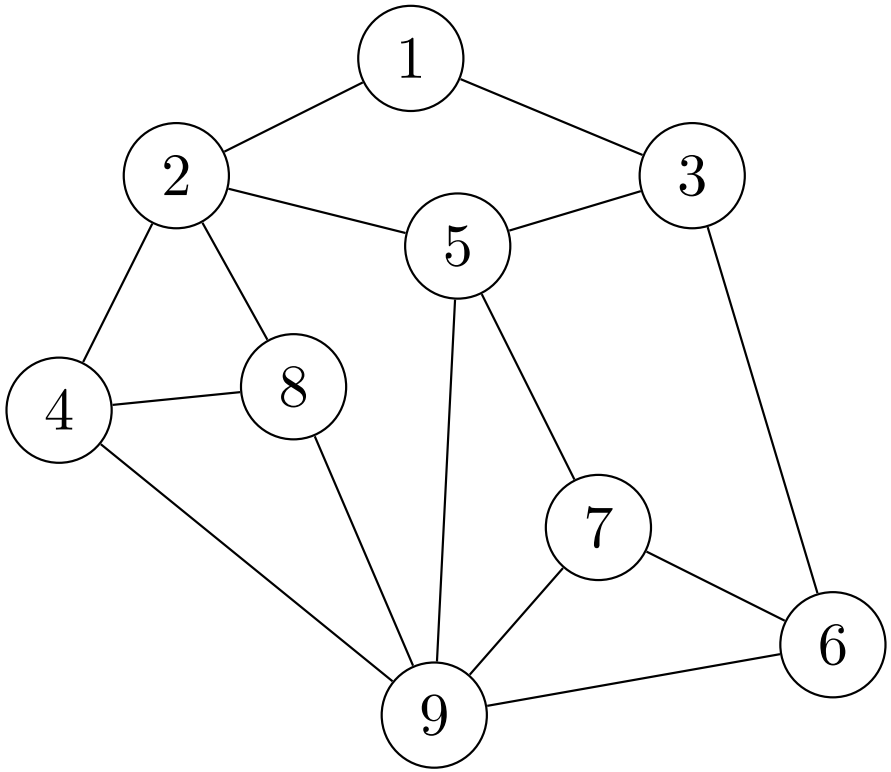
其中的数据元素之间是一对多的层次关系.



其中的数据元素之间是多对多的关系.

图形结构

其中的数据元素之间是多对多的关系.



在用示意图表示数据的逻辑结构时，请注意：

- 将每一个数据元素看做一个结点，用圆圈表示；
- 元素之间的逻辑关系用结点之间的连线表示，如果这个关系是有方向的，那么用带箭头的连线表示。

在用示意图表示数据的逻辑结构时，请注意：

- 将每一个数据元素看做一个结点，用圆圈表示；
- 元素之间的逻辑关系用结点之间的连线表示，如果这个关系是有方向的，那么用带箭头的连线表示。

逻辑结构是针对具体问题的，是为了解决某个问题，在对问题理解的基础上，选择一个合适的数据结构表示数据元素之间的逻辑关系。

数据的逻辑结构在计算机中的存储方式.

物理结构

数据的逻辑结构在计算机中的存储方式.

数据的存储结构应正确反映数据元素之间的逻辑关系。如何存储数据元素之间的逻辑关系，是实现物理结构的重点和难点。

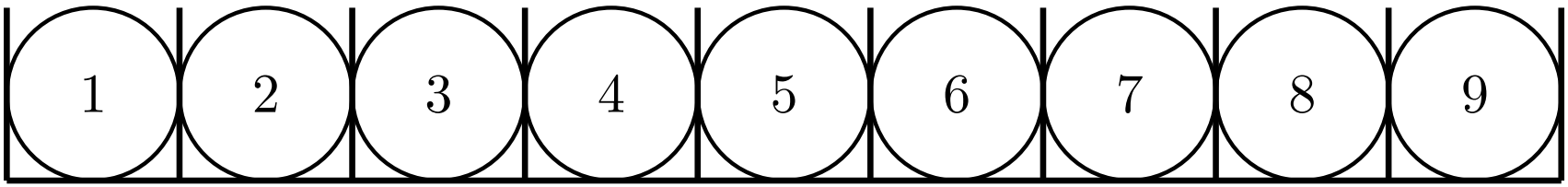
数据的逻辑结构在计算机中的存储方式.

数据的存储结构应正确反映数据元素之间的逻辑关系。如何存储数据元素之间的逻辑关系，是实现物理结构的重点和难点。

把数据元素存放在连续的存储单元里，其数据间的逻辑关系与物理关系一致。

顺序存储结构

把数据元素存放在连续的存储单元里，其数据间的逻辑关系与物理关系一致。

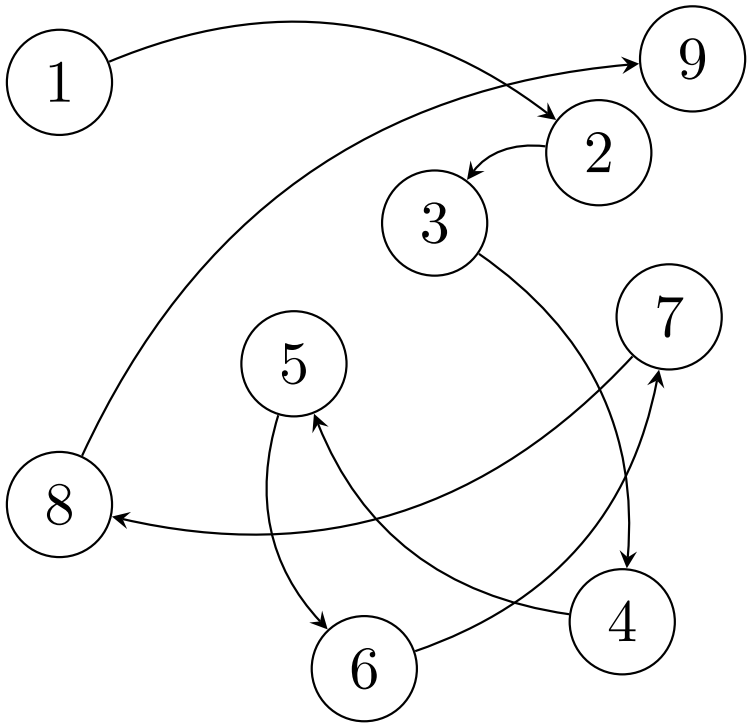


链式存储结构

把数据元素存放在任意的存储单元里，这组存储单元可以连续，也可以不连续。

链式存储结构

把数据元素存放在任意的存储单元里，这组存储单元可以连续，也可以不连续。



数据元素的存储关系并不能反映其逻辑关系，需要用一个指针存放数据元素的地址，这样就可以通过地址找到相关联数据元素的位置。

逻辑结构是面向问题的，而物理结构是面向计算机的，我们的目标是将数据及其逻辑关系存储到计算机的内存中。

1 线性表的链式存储

指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

数据类型

指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

在C语言中，按照取值的不同，数据类型可分为：

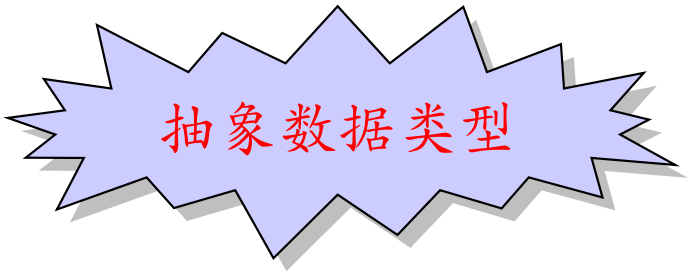
- 原子类型：是不可再分解的基本类型，包括整型、浮点型、字符型等；
- 由若干个类型组合而成，是可以再分解。如整型数组由若干个整型数据组成。

抽取出事物具有的普遍性的本质。

抽取出事物具有的普遍性的本质。

提炼出问题的特征，忽略非本质的细节，对具体事物做一个概括。

抽象是一种思考问题的方式，它隐藏了繁杂的细节，只保留实现目标所必须的信息。



(Abstract Data Type, ADT)
指一个数学模型及定义在该模型上的一组操作。

抽象数据类型

(Abstract Data Type, ADT)
指一个数学模型及定义在该模型上的一组操作。

ADT的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关。

抽象的意义在于数据类型的数学抽象特性。

例

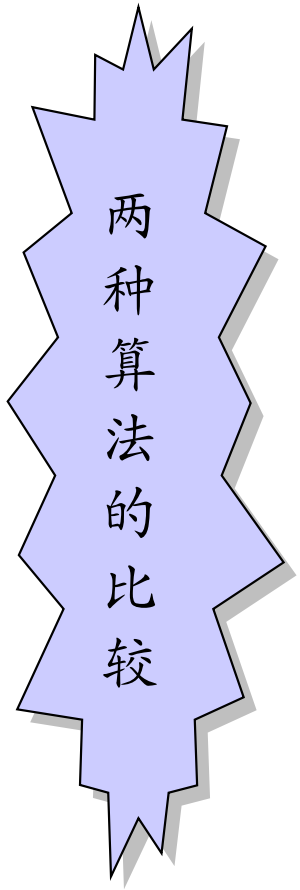
编写计算机绘图软件时，经常会用到坐标，总会出现成对的x和y，在3D系统中还有z出现。我们不妨定义一个叫point的抽象数据类型，它有x, y, z三个变量。这样我们可以方便地操作一个point数据变量就能知道这一点的坐标。

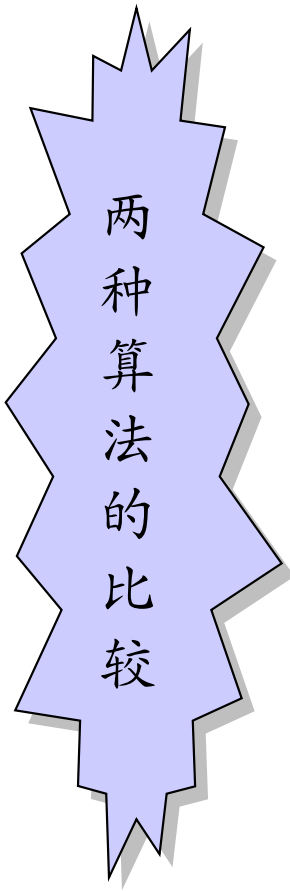
图: 超级玛丽


ADT标准格式


```
ADT 抽象数据类型名
Data
    数据元素之间逻辑关系的定
    义
Operation
    操作1
        初始条件
        操作结果描述
    操作2
        ------
    操作3
        ------
```

1 线性表的链式存储


两种算法的比较




```
int i,sum=0,n=100;
for (i=1;i<=n;i++)
  sum=sum+i;
printf("%d",sum);
```

```
int i,sum=0,n=100;
sum=(n+1)*n/2;
printf("%d",sum);
```

算法

算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作.

算法

算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作.

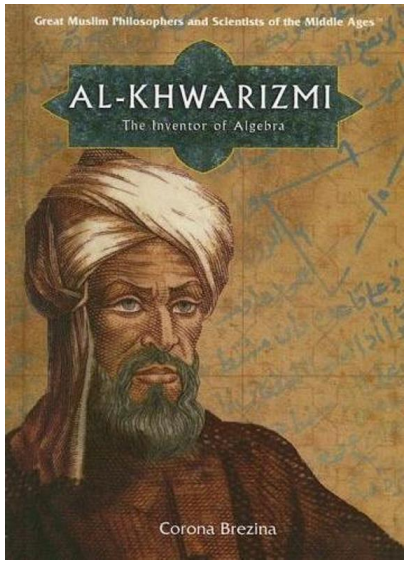
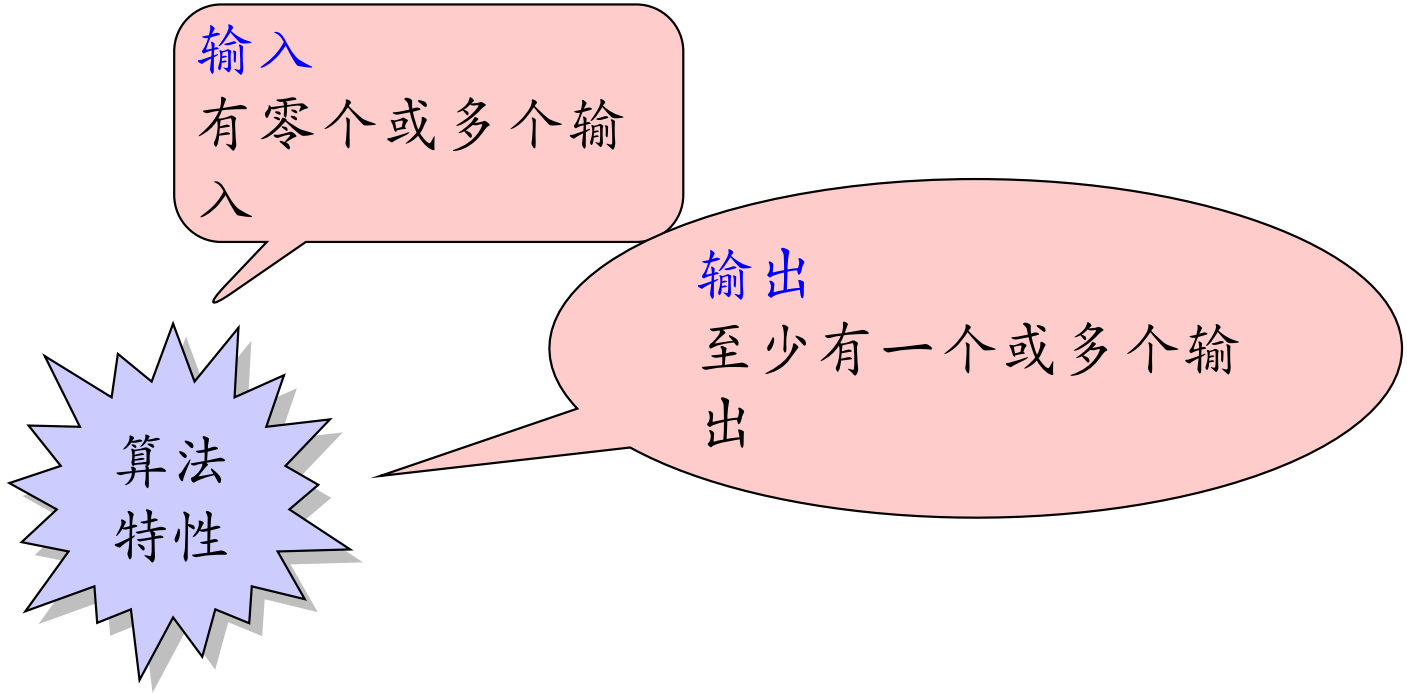


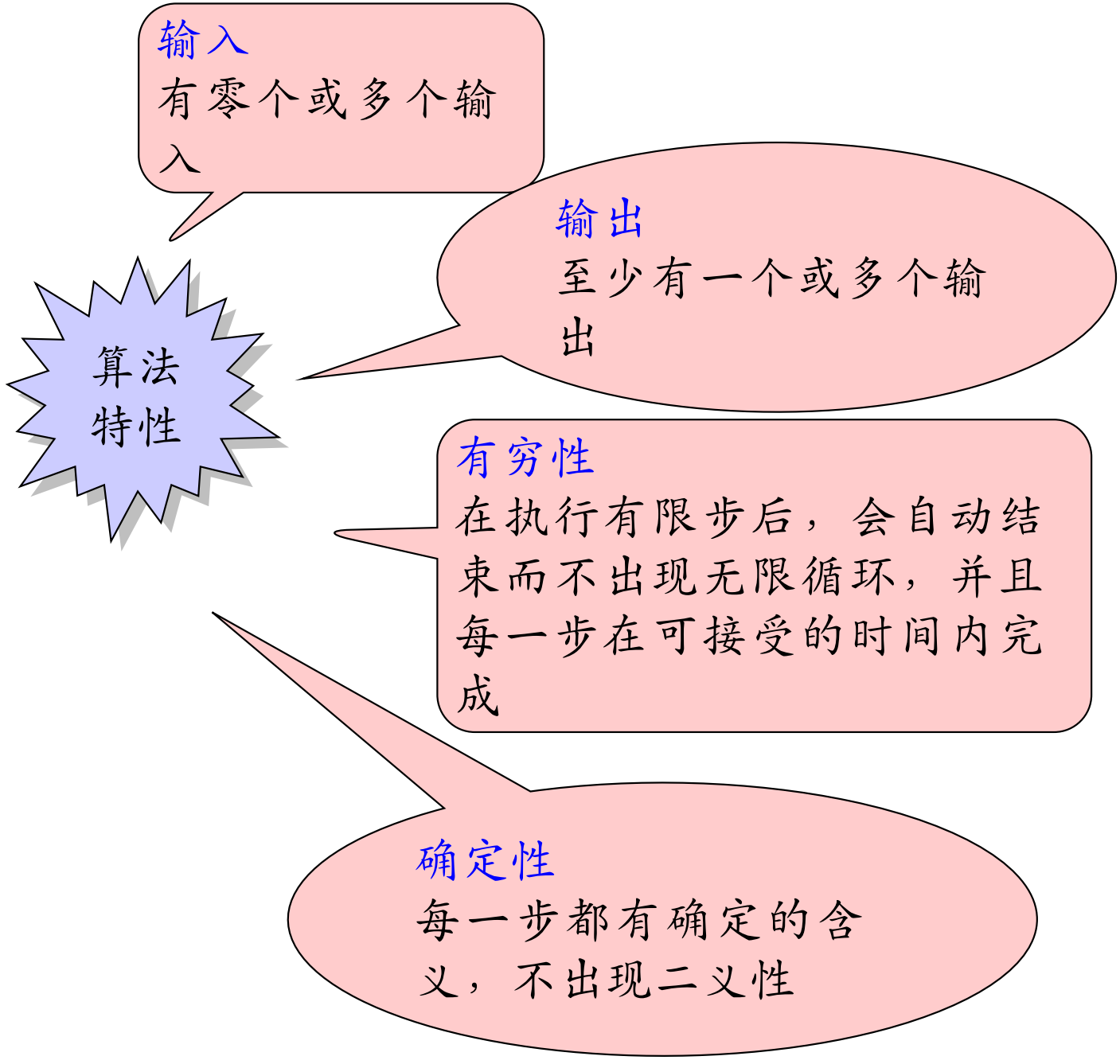
图: 阿勒.花剌子密(约780～约850，波斯数学家)

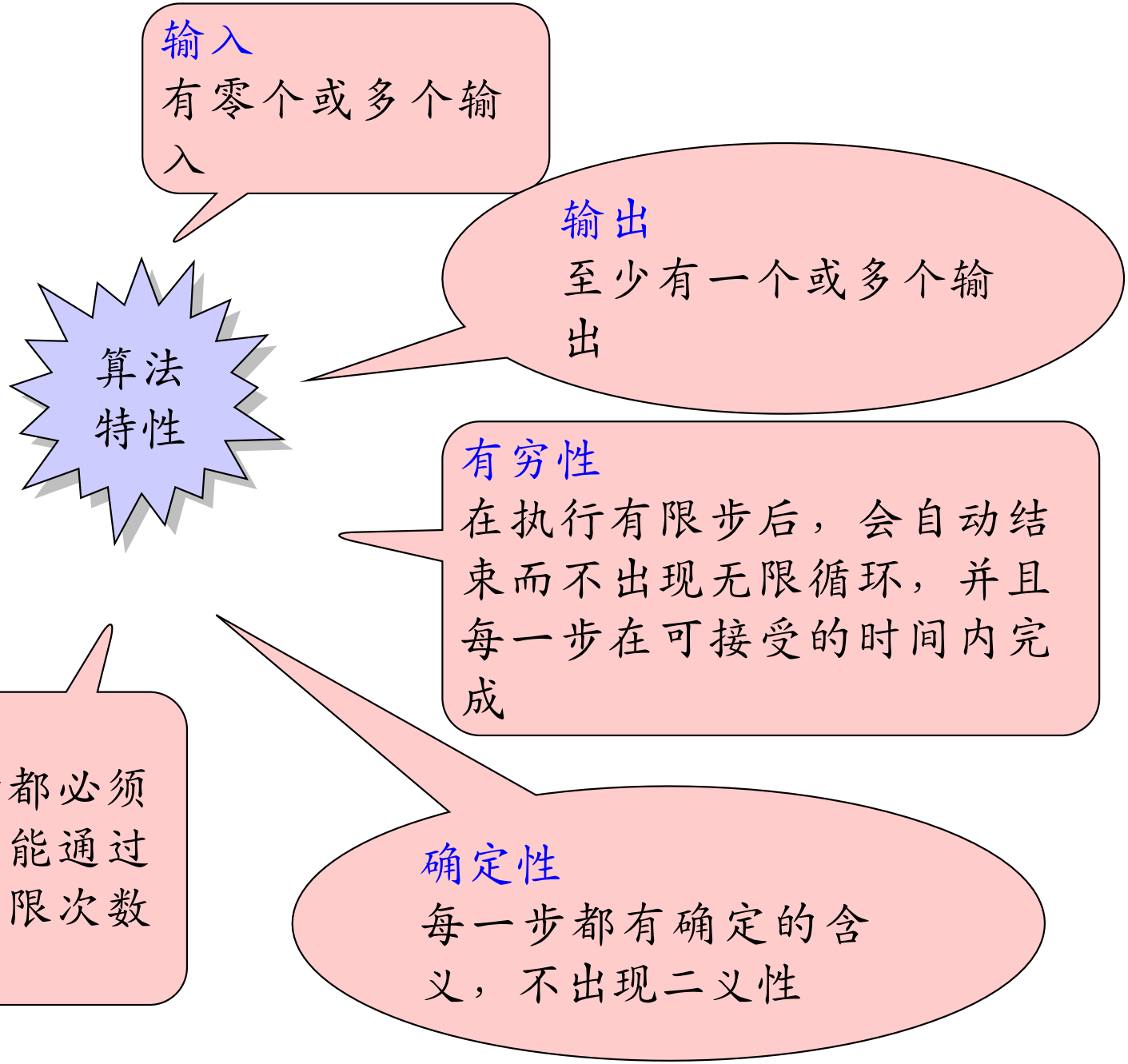

算法
特性


输入
有零个或多个输入
算法
特性


输入
有零个或多个输入
输出
至少有一个或多个输出
算法特性


输入
有零个或多个输入
输出
至少有一个或多个输出
算法特性
有穷性
在执行有限步后，会自动结束而不出现无限循环，并且每一步在可接受的时间内完成


输入
有零个或多个输入
输出
至少有一个或多个输出
算法特性
有穷性
在执行有限步后，会自动结束而不出现无限循环，并且每一步在可接受的时间内完成
确定性
每一步都有确定的含义，不出现二义性


输入
有零个或多个输入
输出
至少有一个或多个输出
算法特性
有穷性
在执行有限步后，会自动结束而不出现无限循环，并且每一步在可接受的时间内完成
可行性
每一步都必须可行，能通过执行有限次数完成
确定性
每一步都有确定的含义，不出现二义性


算法
设计
要求

正确性

算法至少应该具有输入、输出和加工处理无歧义性，能正确反映问题的需求、能得到问题的正确答案。

- 无语法错误
- 对合法输入能产生满足要求的输出
- 对非法输入能给出满足规格的说明
- 对精心选择的甚至是刁难的测试数据都有满足要求的输出

可读性

便于阅读、理解和交流

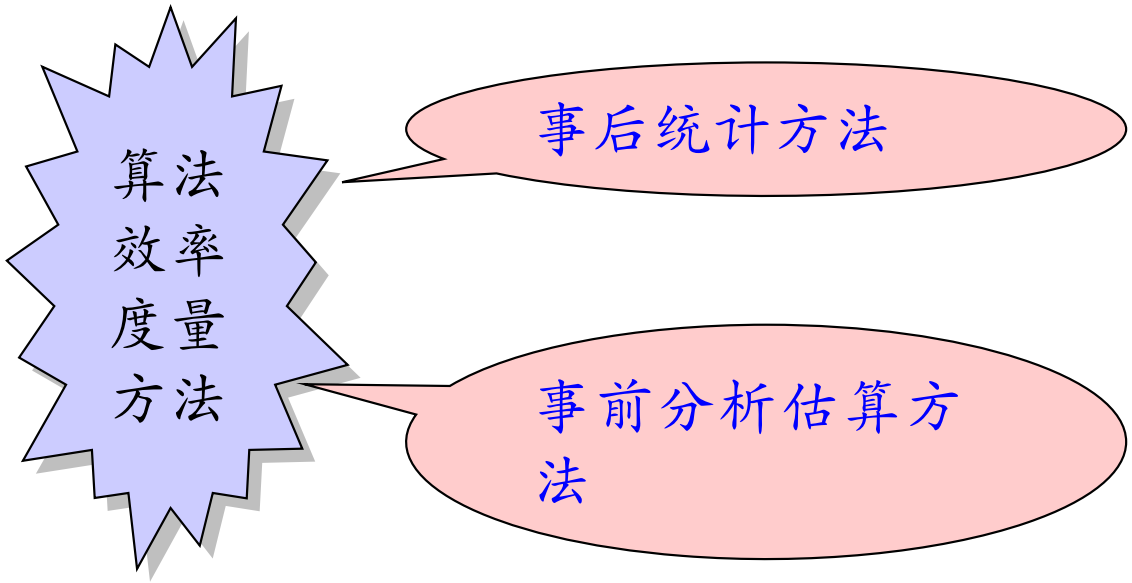
健壮性(鲁棒性)

当输入不合法时，也能做出相应处理，而不是产生异常或莫名其妙的结果

时间效率高和存储量低


算法效率度量方法


算法效率度量方法
事后统计方法


算法效率度量方法
事后统计方法
事前分析估算方法

事后统计方法

通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定效率的高低。

事后统计方法

通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定效率的高低。

缺点

须依据算法事先编制好程序

时间的比较依赖于软硬件等环境因素

- 不同性能的机器上算法的表现不尽相同;
- 不同操作系统、编译器等也会影响算法的运行结果;
- CPU使用率和内存占用情况也会造成微小差异。

测试数据设计困难，且程序运行时间还与测试数据的规模有很大关系， 效率高的算法在小的测试数据面前往往得不到体现。

事前分析估算方法

在编制程序前，依据统计方法对算法进行估算。

事前分析估算方法

在编制程序前，依据统计方法对算法进行估算。

程序运行时间取决于
(1) 算法采用的策略、方法 (算法好坏的根本)
(2) 编译产生的代码质量 (软件支持)
(3) 问题的输入规模
(4) 机器执行指令的速度 (硬件性能)

事前分析估算方法

在编制程序前，依据统计方法对算法进行估算。

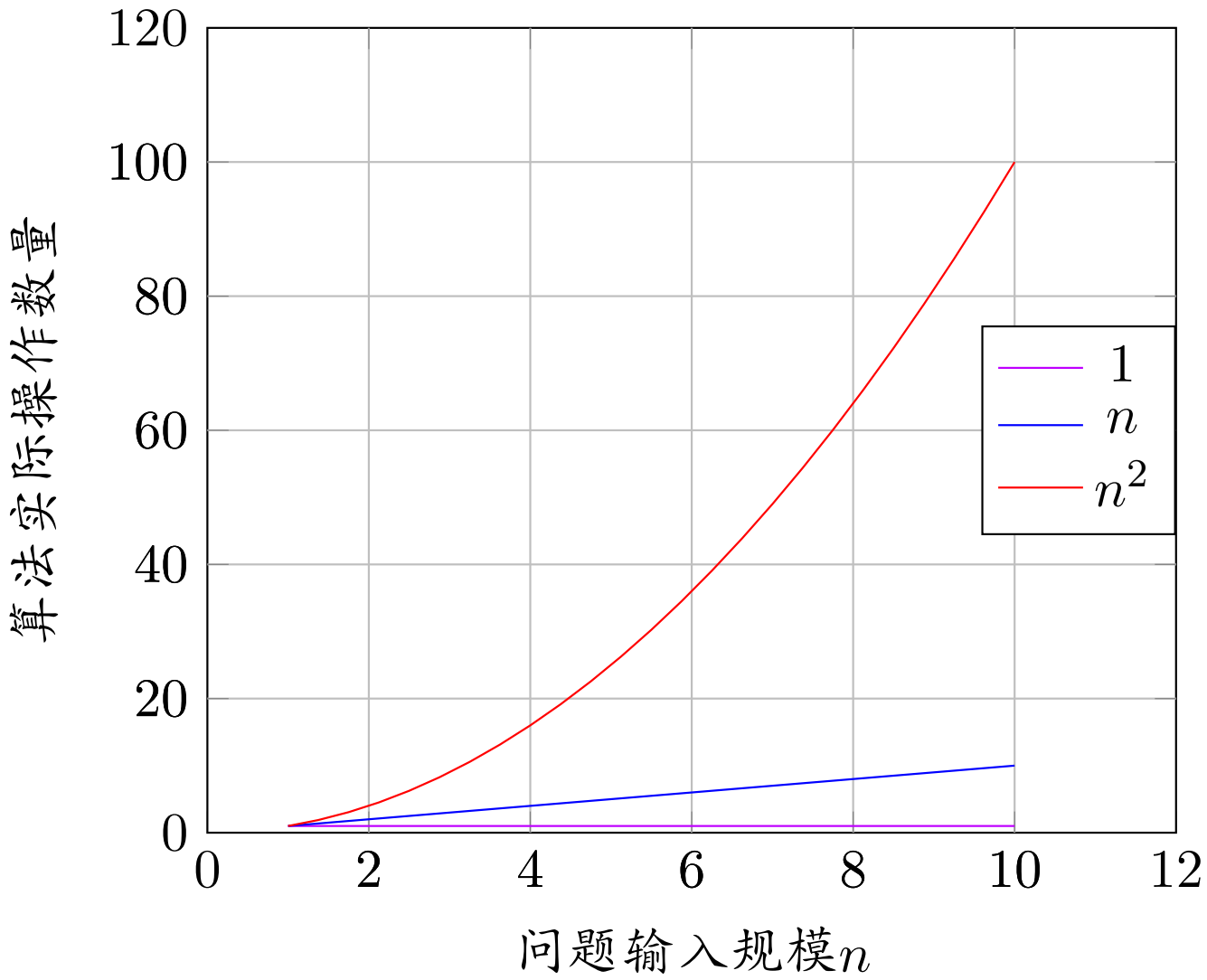
程序运行时间取决于
(1) 算法采用的策略、方法 (算法好坏的根本)
(2) 编译产生的代码质量 (软件支持)
(3) 问题的输入规模
(4) 机器执行指令的速度 (硬件性能)

程序的运行时间，依赖于算法的好坏和问题的输入规模。

```
for(i=1;i<=n;i++)
   sum+=i;                    //执行n次
```

```
sum=(n+1)*n/2;                //执行1次
```

```
for(i=1;i<=n;i++)
   for(j=1;j<=n;j++){
      x++;                    //执行nxn次
      sum+=x;
   }
```


120
100
80
60
40
20
0
0
2
4
6
8
10
12
算法实际操作数量
问题输入规模n
1
n
n^2

函数的渐近增长

给定两个函数$f(n)$和$g(n)$，
若$\exists N \in \mathbb{N}, \text{s.t. } \forall n > N$, $f(n)$总是比$g(n)$大，我们就说$f(n)$的增长渐近快于$g(n)$.

函数的渐近增长

给定两个函数$f(n)$和$g(n)$，
若$\exists N \in \mathbb{N}, \text{s.t. } \forall n > N$, $f(n)$总是比$g(n)$大，我们就说$f(n)$的增长渐近快于$g(n)$.

次数n	算法A $(2n+3)$	算法A' $(2n)$	算法B $(3n+1)$	算法B' $(3n)$
1	5	2	4	3
2	7	4	7	6
3	9	6	10	9
10	23	20	31	30
100	203	200	301	300

次数n	算法C $(4n+8)$	算法C' (n)	算法D $(2n^2+1)$	算法D' (n^2)
1	12	1	3	2
2	16	2	9	4
3	20	3	19	9
10	48	10	201	100
100	408	100	20 001	10 000
1000	4 008	1 000	2 000 001	1 000 000

次数n	算法C $(4n+8)$	算法C' (n)	算法D $(2n^2+1)$	算法D' (n^2)
1	12	1	3	2
2	16	2	9	4
3	20	3	19	9
10	48	10	201	100
100	408	100	20 001	10 000
1000	4 008	1 000	2 000 001	1 000 000

函数的渐近增长可忽略加法常数，并且最高次项的系数也不重要。

次数n	算法E $(2n^2+3n+1)$	算法E' (n^2)	算法F $(2n^3+3n+1)$	算法F' (n^3)
1	6	1	6	1
2	15	4	23	8
3	28	9	64	27
10	231	100	2 031	1 000
100	20 301	10 000	2 000 301	1 000 000

次数n	算法E $(2n^2+3n+1)$	算法E' (n^2)	算法F $(2n^3+3n+1)$	算法F' (n^3)
1	6	1	6	1
2	15	4	23	8
3	28	9	64	27
10	231	100	2 031	1 000
100	20 301	10 000	2 000 301	1 000 000

最高次项的指数越大，随着n的增长，函数结果也会变得增长特别快。

次数n	算法G $(2n^2)$	算法H $(3n+1)$	算法I $(2n^2+3n+1)$
1	2	4	6
2	8	7	15
5	50	16	66
10	200	31	231
100	20 000	301	20 301
1,000	2 000 000	3 001	2 003 001
10,000	200 000 000	30 001	200 030 001
100,000	20 000 000 000	300 001	20 000 300 001
1,000,000	2 000 000 000 000	3 000 001	2 000 003 000 001

次数n	算法G $(2n^2)$	算法H $(3n+1)$	算法I $(2n^2+3n+1)$
1	2	4	6
2	8	7	15
5	50	16	66
10	200	31	231
100	20 000	301	20 301
1,000	2 000 000	3 001	2 003 001
10,000	200 000 000	30 001	200 030 001
100,000	20 000 000 000	300 001	20 000 300 001
1,000,000	2 000 000 000 000	3 000 001	2 000 003 000 001

注

当n越来越大时，$3n+1$的结果与$2n^2$相比，几乎可以忽略不计。也就是说，随着n的不断增大，算法G其实很接近于算法I.

判断一个算法的效率时，函数中的常数与其他次要项可以忽略，而更应该关注主项(最高阶项)的阶数。

算法时间复杂度

在进行算法分析时，语句总的执行次数$T(n)$是关于问题规模n的函数，进而分析$T(n)$随n的变化情况并确定$T(n)$的数量级。算法的时间复杂度，也就是算法的时间量度，记作

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

它表示随着n的增大，算法执行时间的增长率和$f(n)$的增长率相同，称为算法的渐近时间复杂度，简称为时间复杂度，其中$f(n)$是关于n的某个函数。

算法时间复杂度

在进行算法分析时，语句总的执行次数$T(n)$是关于问题规模n的函数，进而分析$T(n)$随n的变化情况并确定$T(n)$的数量级。算法的时间复杂度，也就是算法的时间量度，记作

大O记法

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

它表示随着n的增大，算法执行时间的增长率和$f(n)$的增长率相同，称为算法的渐近时间复杂度，简称为时间复杂度，其中$f(n)$是关于n的某个函数。

如何分析一个算法的时间复杂度?
即如何推导大O阶?

如何分析一个算法的时间复杂度?
即如何推导大O阶?

1. 用常数1取代运行次数中的所有加法常数;
2. 在修改后的运行次数函数中，只保留最高阶项;
3. 如果最高阶项存在且不是1，则去除最高阶项的系数。

得到的结果就是大O阶。

常数阶$O(1)$

```
int  sum=0,n=100;    //执行1次
sum=(n+1)*n/2;       //执行1次
printf("%d",sum);    //执行1次
```

常数阶$O(1)$

```
int  sum=0,n=100;    //执行1次
sum=(n+1)*n/2;       //执行第1次
sum=(n+1)*n/2;       //执行第2次
sum=(n+1)*n/2;       //执行第3次
sum=(n+1)*n/2;       //执行第4次
sum=(n+1)*n/2;       //执行第5次
printf("%d",sum);    //执行1次
```


```
int  i;
for  (i=0;i<n;i++}
     //时间复杂度为O(1)的语句块
```

对数阶$O(\log n)$

```
int count=1;
while (count<n){
   count*=2;
   //时间复杂度为O(1)的语句块
}
```

对数阶$O(\log n)$

```
int count=1;
while (count<n){
   count*=2;
   //时间复杂度为O(1)的语句块
}
```

$$2^x = n \implies x = \log_2 n = \frac{\log n}{\log 2}.$$

平方阶$O(n^2)$

```
int i,j;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

```
int i,j;
for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

```
int i,j;
for (i=0;i<n;i++)
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

平方阶$O(n^2)$

```
int i,j;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

$O(n^2)$

```
int i,j;
for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

```
int i,j;
for (i=0;i<n;i++)
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

平方阶$O(n^2)$

```
int i,j;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

$O(n^2)$

```
int i,j;
for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

$O(m \times n)$

```
int i,j;
for (i=0;i<n;i++)
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

平方阶$O(n^2)$

```
int i,j;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

$O(n^2)$

```
int i,j;
for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    //时间复杂度为O(1)的语句块
```

$O(m \times n)$

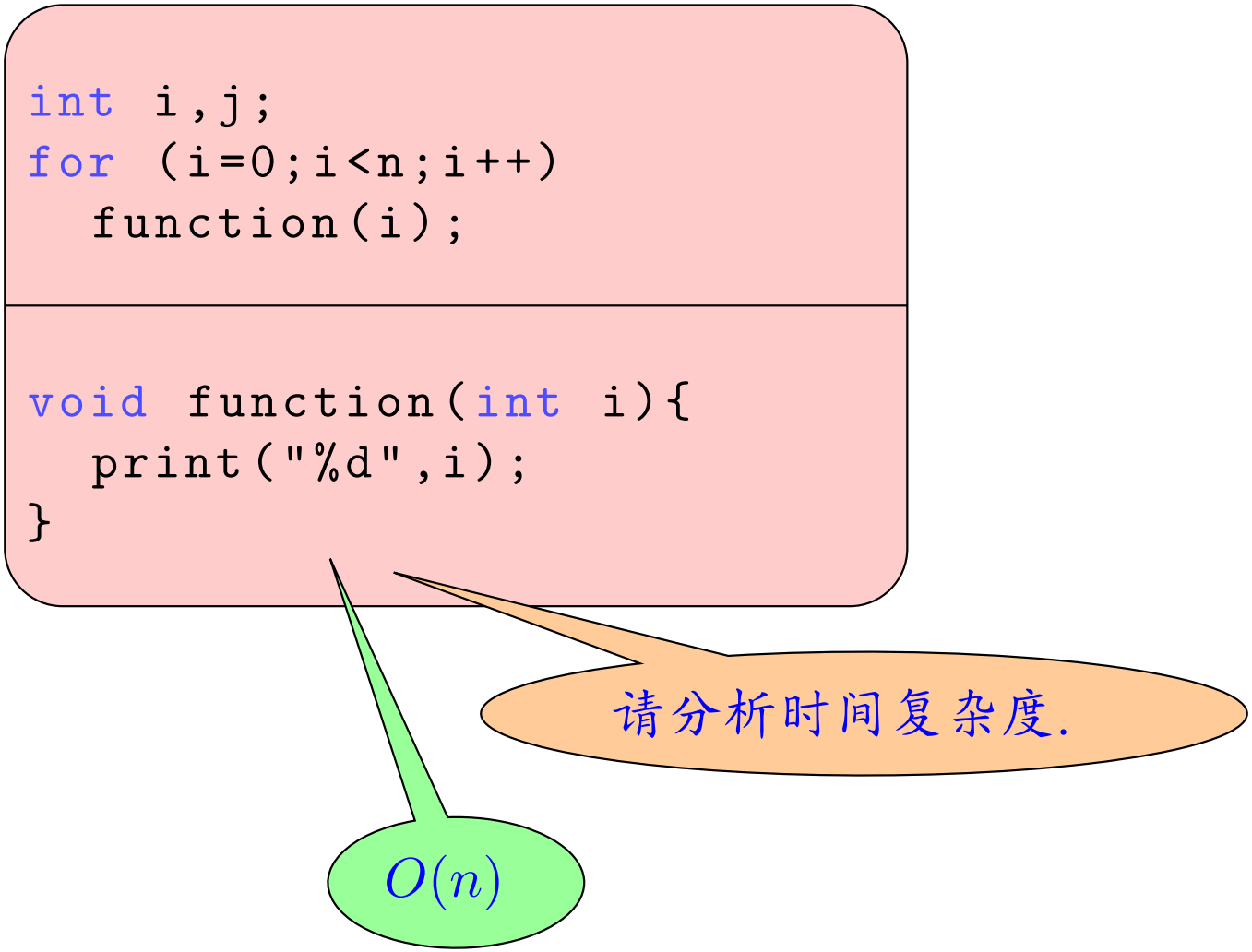
```
int i,j;
for (i=0;i<n;i++)
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

因$f(n) = n + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$,
故时间复杂度为$O(n^2)$.

```
int i,j;
for (i=0;i<n;i++)
  function(i);
```

```
void function(int i){
  print("%d",i);
}
```

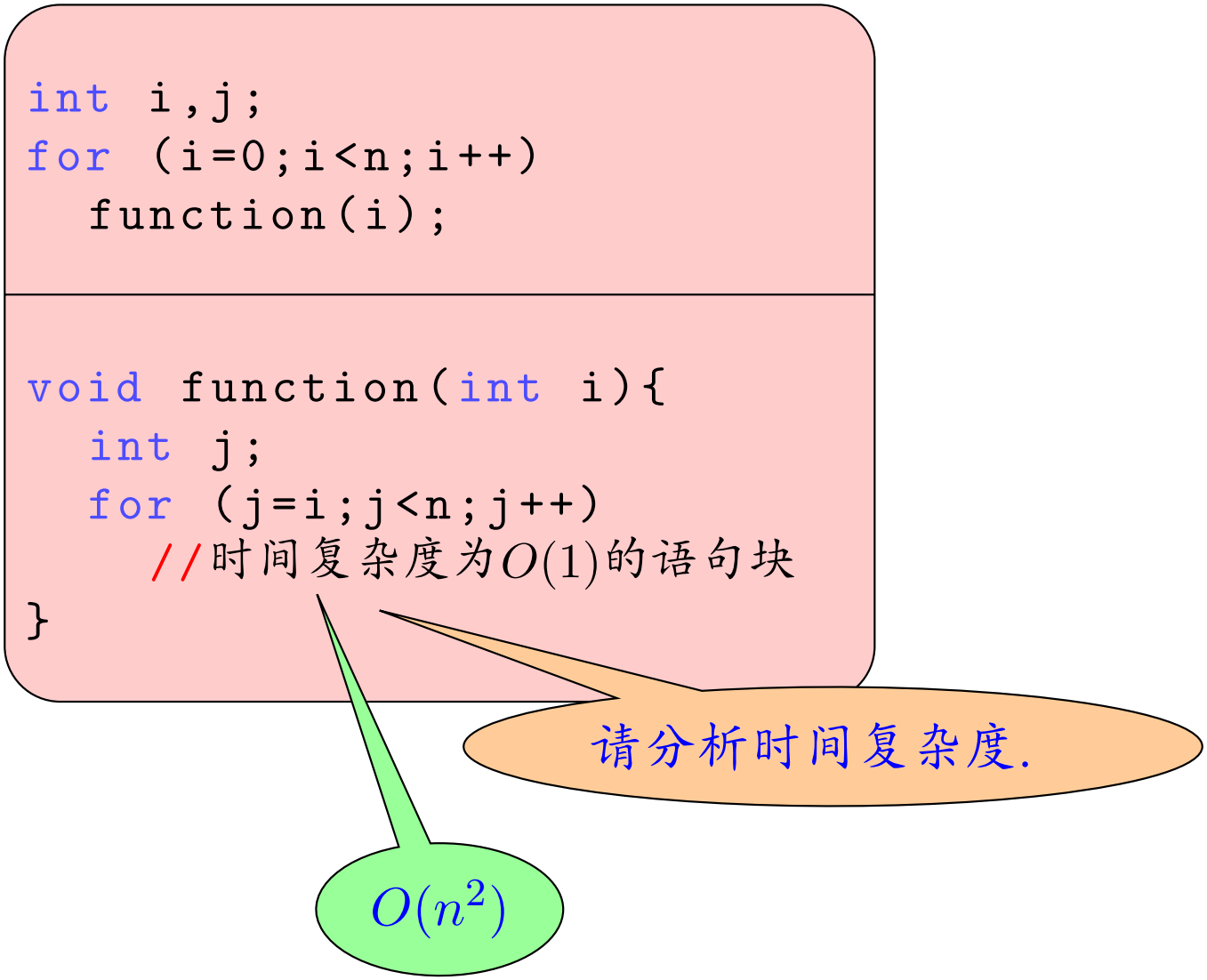
请分析时间复杂度.


int i,j;
for (i=0;i<n;i++)
function(i);
void function(int i){
print("%d",i);
}
请分析时间复杂度.
$O(n)$

```
int i,j;
for (i=0;i<n;i++)
  function(i);
```

```
void function(int i){
  int j;
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
}
```

请分析时间复杂度.


int i,j;
for (i=0;i<n;i++)
function(i);
void function(int i){
int j;
for (j=i;j<n;j++)
//时间复杂度为O(1)的语句块
}
请分析时间复杂度.
$O(n^2)$

```
n++;                                    //执行次数为1
function(n);                            //执行次数为n
int i,j;
for (i=0;i<n;i++)                       //执行次数为n^2
  function(i);
for (i=0;i<n;i++)                     //执行次数为n(n+1)/2
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

请分析时间复杂度.

```
n++;                                    //执行次数为1
function(n);                            //执行次数为n
int i,j;
for (i=0;i<n;i++)                       //执行次数为n^2
  function(i);
for (i=0;i<n;i++)                     //执行次数为n(n+1)/2
  for (j=i;j<n;j++)
    //时间复杂度为O(1)的语句块
```

请分析时间复杂度.

执行次数$f(n)=1+n+n^2+\frac{n(n+1)}{2}=\frac{3}{2}n^2+\frac{3}{2}n+1$，故时间复杂度为$O(n^2)$.

Table: 常见时间复杂度

执行次数函数	阶	非正式术语
12	$O(1)$	常数阶
$2n+3$	$O(n)$	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	$O(\log n)$	对数阶
$2n+3n\log_2 n+19$	$O(n\log n)$	$n\log n$阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

Table: 常见时间复杂度

执行次数函数	阶	非正式术语
12	$O(1)$	常数阶
$2n+3$	$O(n)$	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	$O(\log n)$	对数阶
$2n+3n\log_2 n+19$	$O(n\log n)$	$n\log n$阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

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

最坏情况

在n维随机数组中查找某个数字，

- 最好情况是出现在第一个位置，时间复杂度为$O(1)$;
- 最坏情况是出现在最后一个位置，时间复杂度为$O(n)$.

最坏情况

在n维随机数组中查找某个数字，

- 最好情况是出现在第一个位置，时间复杂度为$O(1)$;
- 最坏情况是出现在最后一个位置，时间复杂度为$O(n)$.

最坏情况运行时间是一种保证，亦即运行时间将不会再坏了。在应用中，这是一种最重要的需求，通常，除非特别指定，我们提到的运行时间都是最坏情况的运行时间。

平均情况

在n维随机数组中查找某个数字，

- 从概率角度讲，该数字在每个位置的可能性相同，故平均查找时间为$n/2$。

平均情况

在n维随机数组中查找某个数字，

- 从概率角度讲，该数字在每个位置的可能性相同，故平均查找时间为$n/2$。

平均运行时间是所有情况中最有意义的，因为它是期望的运行时间。

对算法的分析，

- 一种方法是计算所有情况的平均值，这种时间复杂度的计算方法称为平均时间复杂度；
- 另一种是计算最坏情况下的时间复杂度，这种方法称为最坏时间复杂度。

在没有特别说明的情况下，都指最坏时间复杂度。

算法空间复杂度

通过计算算法所需的存储空间实现，其计算公式记作

$$S(n) = O(f(n)).$$

其中n为问题的规模，$f(n)$是语句关于n所占存储空间的函数。


总结回顾

数 据

数 据

数 据 对 象

数 据

数 据 对 象

数据元素	数据元素	数据元素	数据元素

数 据

数 据 对 象

数据元素	数据元素	数据元素	数据元素

数据项1	数据项2	数据项1	数据项2	数据项1	数据项2	数据项1	数据项2

数 据

数 据 对 象

数据元素	数据元素	数据元素	数据元素

数据项1	数据项2	数据项1	数据项2	数据项1	数据项2	数据项1	数据项2

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。


总结回顾

总结回顾

- 逻辑结构
- 集合结构
- 线性结构
- 树形结构
- 图形结构

逻辑结构
集合结构
线性结构
树形结构
图形结构

物理结构
顺序存储结构
链式存储结构


总结回顾

总结回顾

算法定义：解决特定问题求解步骤的描述，在计算机中为指令的有限序列，且每条指令表示一个或多个操作。

总结回顾

算法定义：解决特定问题求解步骤的描述，在计算机中为指令的有限序列，且每条指令表示一个或多个操作。

算法特性：有穷性、确定性、可行性、输入、输出

总结回顾

算法定义：解决特定问题求解步骤的描述，在计算机中为指令的有限序列，且每条指令表示一个或多个操作。

算法特性：有穷性、确定性、可行性、输入、输出

算法设计要求：正确性、可读性、健壮性、高效率和低存储。


总结回顾

总结回顾

算法度量方法：事后统计方法(不科学、不准确)、事前分析估算方法。

总结回顾

算法度量方法：事后统计方法(不科学、不准确)、事前分析估算方法。

函数的渐近增长：给定两个函数$f(n)$和$g(n)$，若$\exists N \in \mathbb{N}, \text{s.t. } \forall n > N$, $f(n)$总是比$g(n)$大，我们就说$f(n)$的增长渐近快于$g(n)$.


总结回顾

大O阶推导过程

(1) 用常数1取代运行次数中的所有加法常数；

(2) 在修改后的运行次数函数中，只保留最高阶项；

(3) 如果最高阶项存在且不是1，则去除最高阶项的系数。

得到的结果就是大O阶。


总结回顾

总结回顾

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

总结回顾

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

算法最坏情况和平均情况，以及空间复杂度的概念。

1 线性表的链式存储

线性表的链式存储

定义 (链式存储)

用一组任意的存储单元存储线性表中的数据元素。 用这种方法存储的线性表简称线性链表。

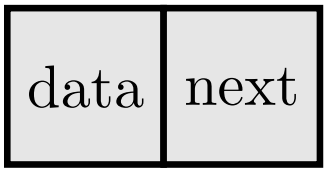
线性表的链式存储

定义 (链式存储)

用一组任意的存储单元存储线性表中的数据元素。 用这种方法存储的线性表简称线性链表。

- 存储链表中结点的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- 链表中的逻辑顺序和物理顺序不一定相同。

线性表的链式存储



data: 数据域，存放结点的值

next: 指针域，存放结点的直接后继的地址

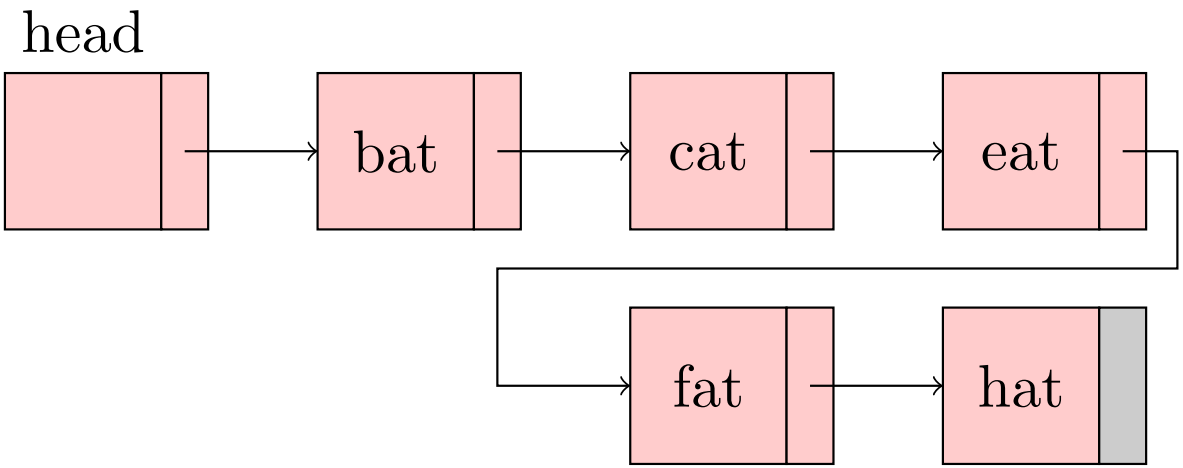
- 链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。
- 每一个结点只包含一个指针域的链表，称为单链表。
- 为操作方便，总在链表的第一个结点之前附设一个头结点（头指针）head指向第一个结点。头结点的数据域可以不存储任何信息（或链表长度等信息）。
- 单链表由表头唯一确定，因此单链表可用头指针的名字来命名。

线性表的链式存储

例1：线性表$L = (bat, cat, eat, fat, hat)$

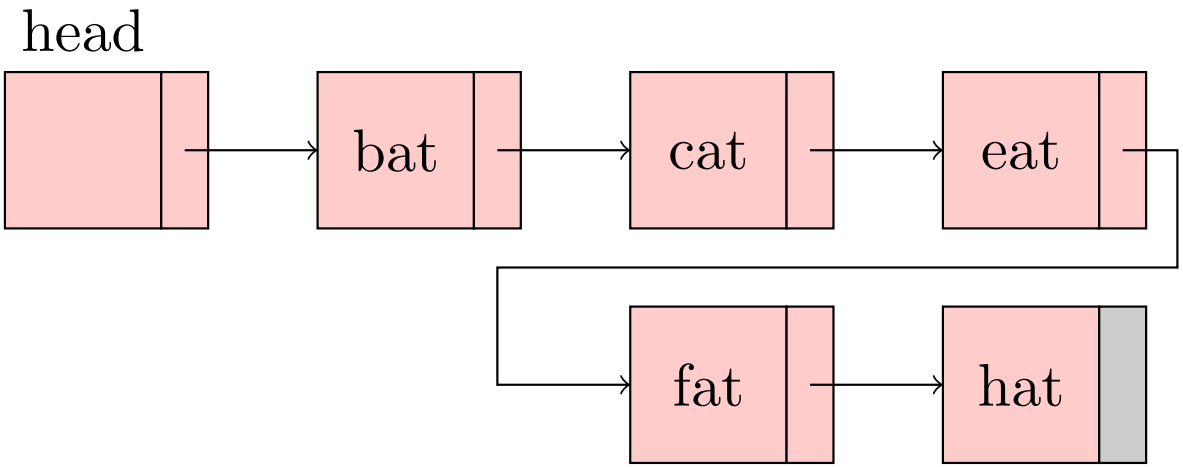
线性表的链式存储

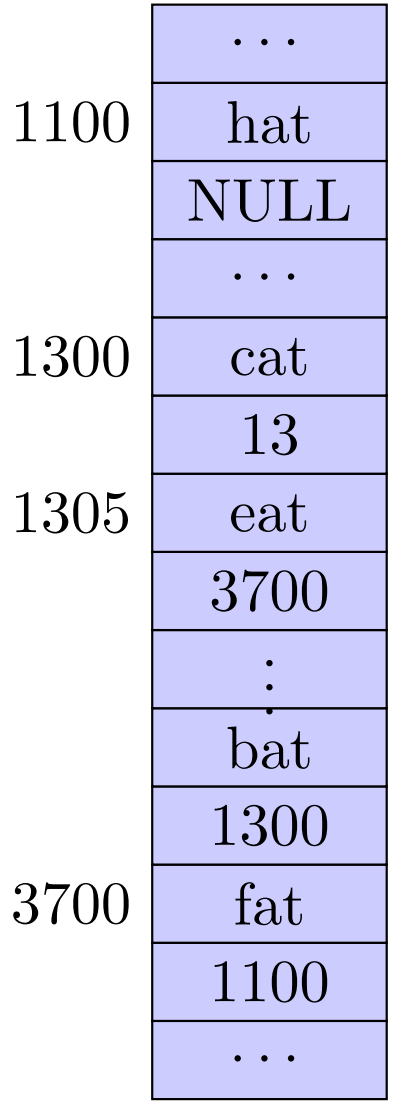
例1：线性表$L = (bat, cat, eat, fat, hat)$



线性表的链式存储

例1：线性表$L = (bat, cat, eat, fat, hat)$





线性表的链式存储结构

结点的描述与实现

```
typedef struct LNode{
   ElemType data;
   struct LNode *next;
} LNode;
typedef struct LNode *LinkList;
```

线性表的链式存储结构

结点的实现

结点是通过动态分配和释放来实现的，即需要时分配，不需要时释放。

```
malloc(), realloc(), sizeof(), free();
```

线性表的链式存储结构

结点的实现

结点是通过动态分配和释放来实现的，即需要时分配，不需要时释放。

```
malloc(), realloc(), sizeof(), free();
```

- 动态分配

  ```
  p=(LNode*)malloc(sizeof(LNode));
  ```

  分配了一个类型为LNode的结点变量的空间，并将其首地址放入指针变量p中。

- 动态释放

  ```
  free(p);
  ```

  系统回收由指针变量p指向的内存区。

(2) 常见的指针操作

图: q=p

(2) 常见的指针操作

图: q=p

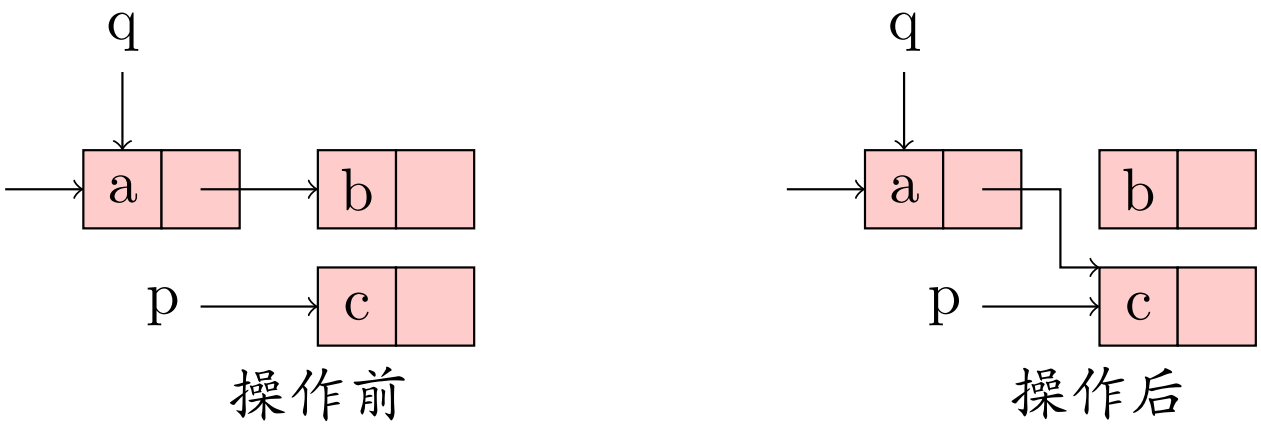
图: q=p->next

常用操作

图：p=p->next;

常用操作

图：p=p->next;



图：q->next=p;

常用操作

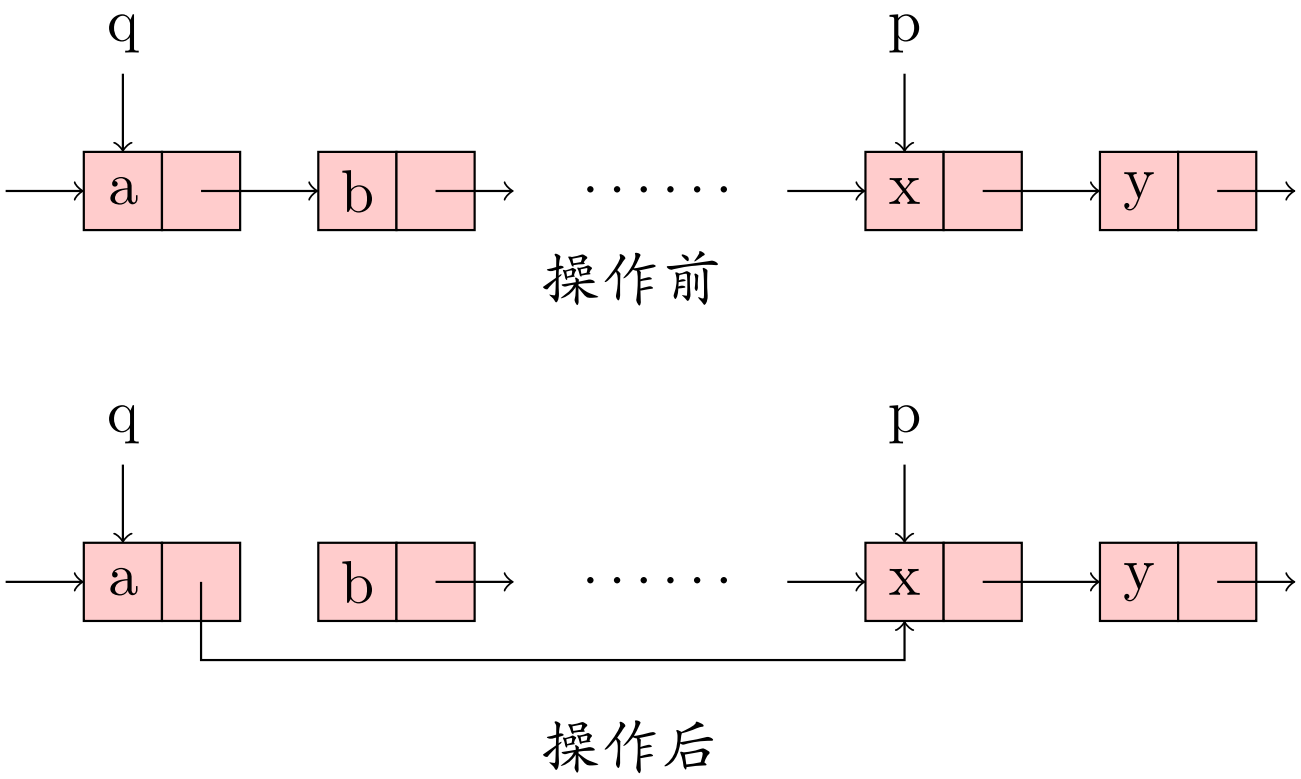
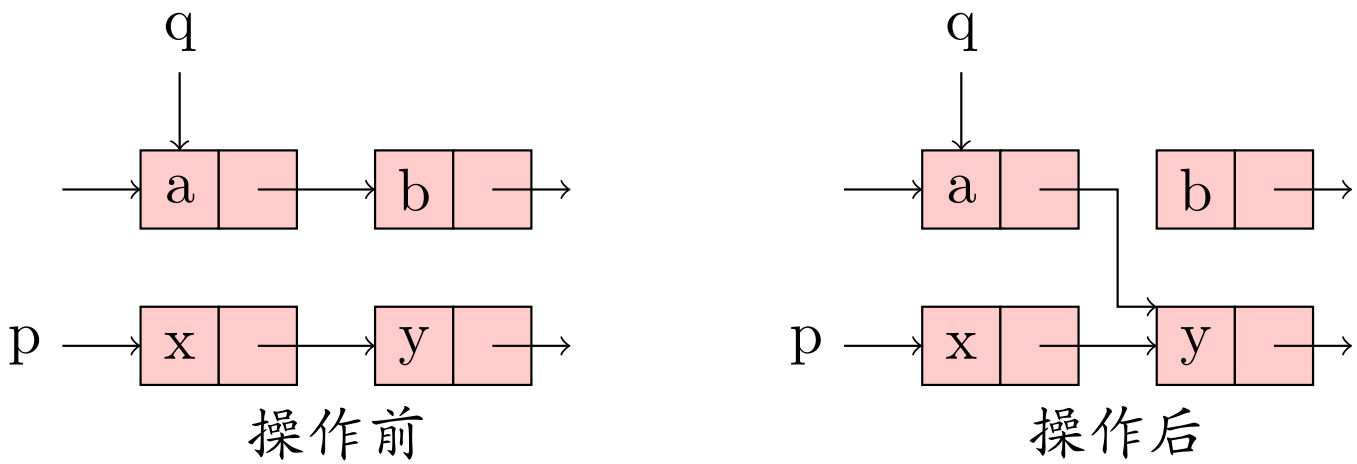


图: q->next=p;

常用操作



图：q->next=p->next;

常用操作

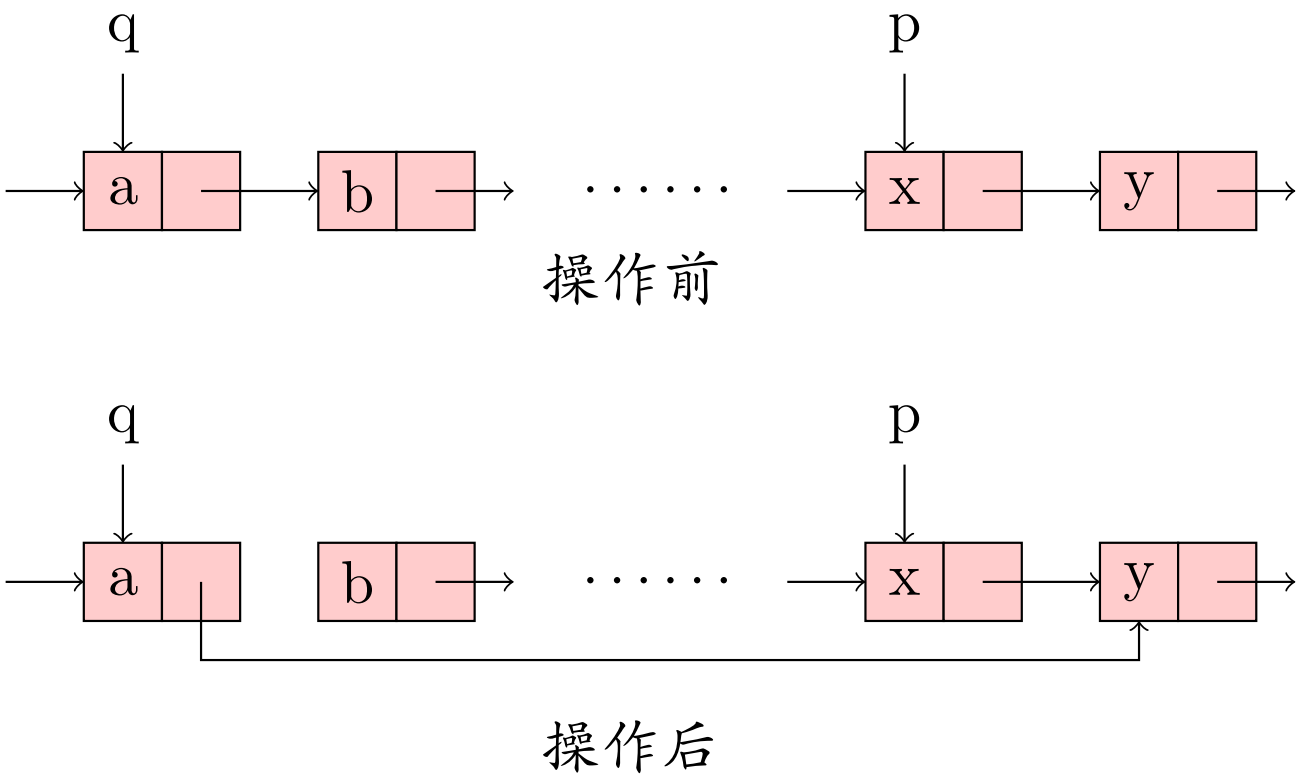


图: q->next=p->next;

单链表的整表创建

- 顺序存储结构的创建，就是一个数组的初始化。而单链表则不同，它可以很散，是一个动态结构。
- 对每个链表而言，它所占用空间的大小和位置不需要预先分配，可根据系统的情况和实际需求即时生成。

所以创建单链表的过程就是一个动态生成链表的过程，即从空表的初始状态起，一次建立各元素结点，并逐个插入链表。

动态地建立单链表的常用方法有两种：

- 头插入法
- 尾插入法

头插入法

- 声明一结点p和计数器变量i
- 初始化一空链表L
- 让L的头结点的指针指向NULL，即建立一个带头结点的单链表
- 循环
  - 生成一个新结点赋值给p
  - 随机生成一个数赋给p的数据域p->data
  - 将p插入到头结点与前一新结点之间

头插入法

```
void CreateLinkListHead(LinkList L, int n){
  LinkList p;
  int i;
  L->next=NULL;
  for(i=0;i<n;i++){
    p=(LinkList) malloc(sizeof(LNode));
    p->data=rand()%100+1;
    p->next=L->next;
    L->next=p;
  }
}
```

尾插入法

头插入法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾，使其成为当前链表的尾结点。

尾插入法

```
void CreateLinkListTail(LinkList L, int n){
  LinkList p,r;
  int i;
  r=L;
  for(i=0;i<n;i++){
    p=(LinkList) malloc(sizeof(LNode));
    p->data=rand()%100+1;
    r->next=p;
    r=p;
  }
  r->next=NULL;
}
```

单链表的查找

1. 按序号查找
2. 按值查找

按序号查找

对于单链表，不能像顺序表中那样直接按序号i访问结点，而只能从链表的头结点出发，沿指针域next逐个结点往下搜索，知道搜到第i个结点为止。因此，链表不是随机存储结构。

设单链表长度为n，要查找第i个结点，仅当$1 \leq i \leq n$时，i的值是合法的。

按序号查找

```
Status GetElem(LinkList L,int i,ElemType *e){
  int j;
  LinkList p;
  p=L->next;          /* Point to the first node */
  j=1;
  while(p!=NULL&&j<i){
    p=p->next; j++;
  }
  if(!p||j>i)
    return ERROR;     /* the i-th element DONOT exist */
  *e=p->data;         /* get data of the i-th element */
  return OK;
}
```

按序号查找

移动指针p的频度

$$\begin{cases} 0\text{次}, & i < 1; \\ i-1\text{次}, & i \in [1,n]; \\ n\text{次}, & i > n. \end{cases} \quad \Rightarrow \quad \text{时间复杂度为} O(n).$$

按值查找

按值查找是在链表中，查找是否有结点值等于给定值key的结点？

- 若有，则返回首次找到的值为key的结点的存储位置；
- 否则返回NULL。

查找时从开始结点出发，沿链表逐个将结点的值和给定值key作比较。

按值查找

```
LinkList LocateNodeKey(LinkList L,ElemType key){
  LinkList p=L->next;

  while(p!=NULL&&p->data!=key) p=p->next;
  if(p->data==key) return p;
  else{
    printf("The node you find DONOT exist!\n");
    return NULL;
  }
}
```

按值查找

平均时间复杂度

算法的执行与形参key有关，平均时间复杂度为$O(n)$。

插入结点

插入运算是指将值为e的新结点插入到表的第i个结点的位置上，即插入到a_{i-1}与a_i之间。因此，必须首先找到a_{i-1}所在的结点p，然后生成一个数据域为e的新结点q，q作为p的直接后继。

插入结点

```
void InsertLNode(LinkList L,int i,ElemType e){
  int j=0; LinkList p,q;
  p=L->next;                    /* Point to the first node */
  while(p!=NULL&&j<i-1){
    p=p->next; j++;
  }
  if(j!=i-1) printf("i too big or equal 0!\n");
  else {
    q=(LinkList) malloc(sizeof(LNode));
    q->data=e; q->next=p->next;
    p->next=q;
  }
}
```

插入结点

平均时间复杂度

设链表长度为n，合法的插入位置是$1 \leq i \leq n$。算法的时间主要耗费在移动指针p上，平均时间复杂度为$O(n)$。

删除结点

- 按序号删除：删除单链表中的第i个结点。
- 按值删除：删除单链表中值为key的第一个结点。

删除结点

按序号删除

- 为了删除第i个结点a_i，必须找到结点的存储地址。
- 该存储地址在其直接前驱结点a_{i-1}的next域中，因此必须首先找到a_{i-1}的存储位置p，然后令p->next指向a_i的直接后继结点，即把a_i从链上摘下来。
- 最后释放结点a_i的空间，将其归还给“存储池”。

设单链表长度为n，则删去第i个结点仅当$1 \leq i \leq n$时是合法的。
当$i = n + 1$时，虽然被删结点不存在，但其前驱结点却存在，是终端结点。故判断条件之一是p->next!=NULL。

删除结点

按序号删除

```
void DeleteLNodeIndex(LinkList L,int i){
  int j=1; LinkList p,q;
  p=L; q=L->next;
  while(p->next!=NULL&&j<i){
    p=q; q=q->next; j++;
  }
  if(j!=i) printf("i too big or equal 0!\n");
  else {
    p->next=q->next;
    free(q);
  }
}
```

删除结点

按序号删除

时间复杂度

时间复杂度为$O(n)$。

删除结点

按值删除

与按值查找相类似，首先要查找值为key的结点是否存在？

- 若存在，则删除；
- 否则返回NULL。

删除结点

按值删除

```
void DeleteLNodeKey(LinkList L,ElemType key){
  LinkList p=L,q=L->next;
  while(q!=NULL&&q->data!=key){
    p=q; q=q->next;
  }
  if(q==NULL) {
    printf("The Node you delete DONOT exist!\n");
    return;
  }
  if(q->data==key){
    p->next=q->next; free(q);
  }
  else {
    printf("The Node you delete DONOT exist!\n");
  }
}
```

删除结点

按值删除

时间复杂度

时间复杂度为$O(n)$。

单链表的删除

链表实现插入和删除运算，无需移动结点，仅需修改指针，解决了顺序表的插入或删除操作需要移动大量元素的问题。

按值删除多个结点

变形之一

删除单链表中值为key的所有结点。

按值删除多个结点

变形之一

删除单链表中值为key的所有结点。

基本思想

- 从单链表的第一个结点开始，对每个结点进行检查，若结点的值为key，则删除之，
- 然后检查下一个结点，直到所有的结点都检查。

按值删除多个结点

```
void Delete_LinkList_Node(LinkList L, int key) {
  LinkList p=L, q=L->next;
  while (q!=NULL){
    if (q->data!=key){
      p->next=q->next; free(q); q=p->next;
    } else {
      p=q; q=q->next;
    }
  }
}
```

删除重复结点

变形之二

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

删除重复结点

变形之二

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

基本思想

从单链表的第一个结点开始，对每个结点进行检查：

- 检查链表中该结点的所有后继结点，只要有值和该结点的值相同，则删除之；
- 然后检查下一个结点，直到所有的结点都检查。

删除重复结点

```
void Delete_LinkList_Value(LNode *L) {
  LinkList p=L->next, q, ptr;
  while (p!=NULL){
    *q=p; *ptr=p->next;
    while(ptr!=NULL){
      if (ptr->data==p->data){
        q->next=ptr->next; free(ptr); ptr=q->next;
      } else {
        q=ptr; ptr=ptr->next;
      }
    }
    p=p->next;
  }
}
```

单链表的合并

设有两个有序的单链表，它们的头指针分别为La、Lb，将它们合并为以Lc为头指针的有序链表。

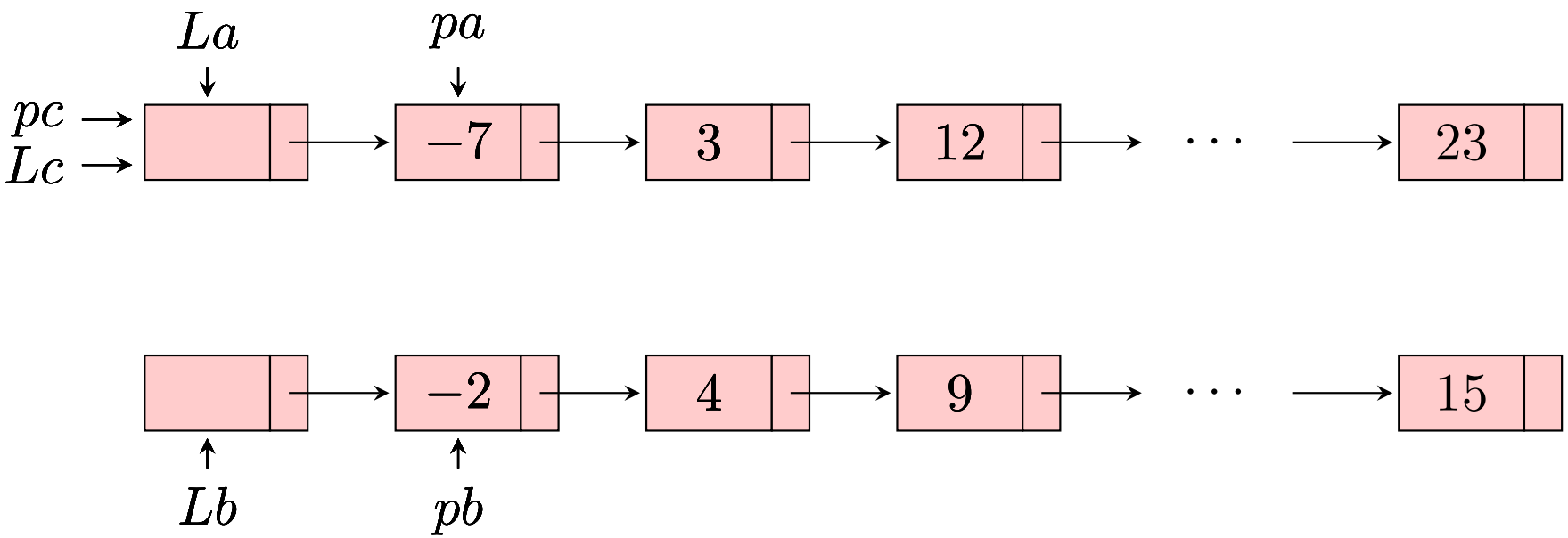


图: 合并前的示意图

单链表的合并

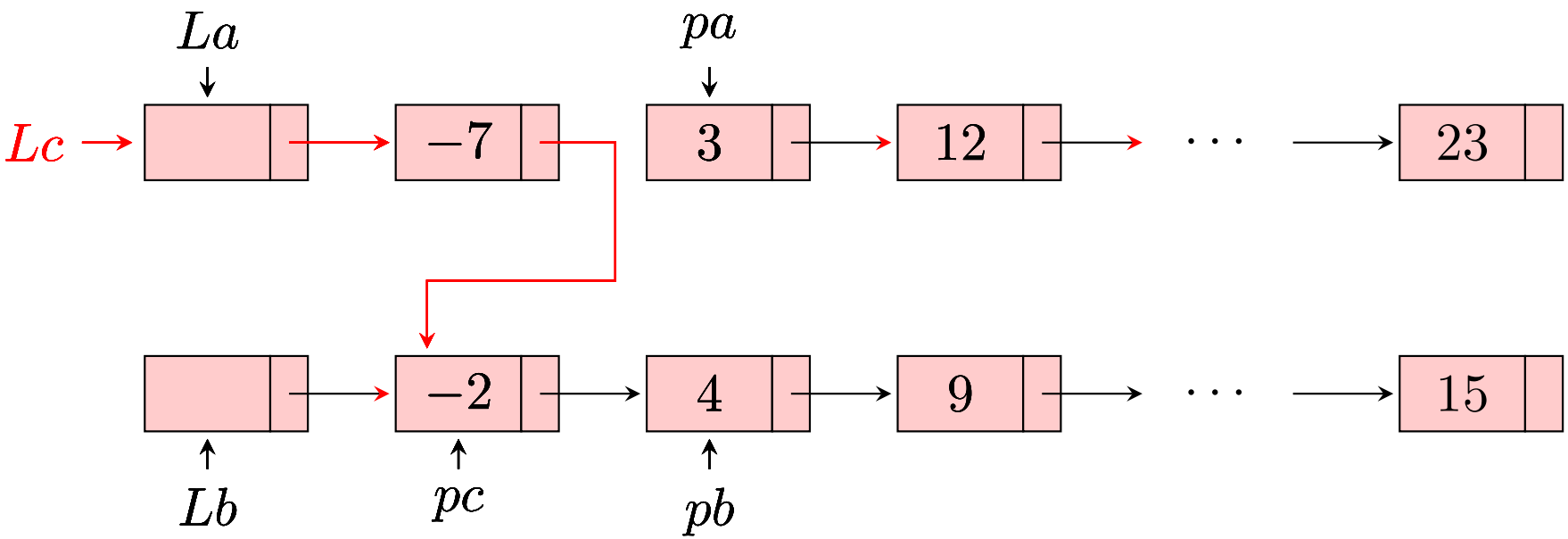


图: 合并了值为$-7,-2$的结点后的示意图

单链表的合并

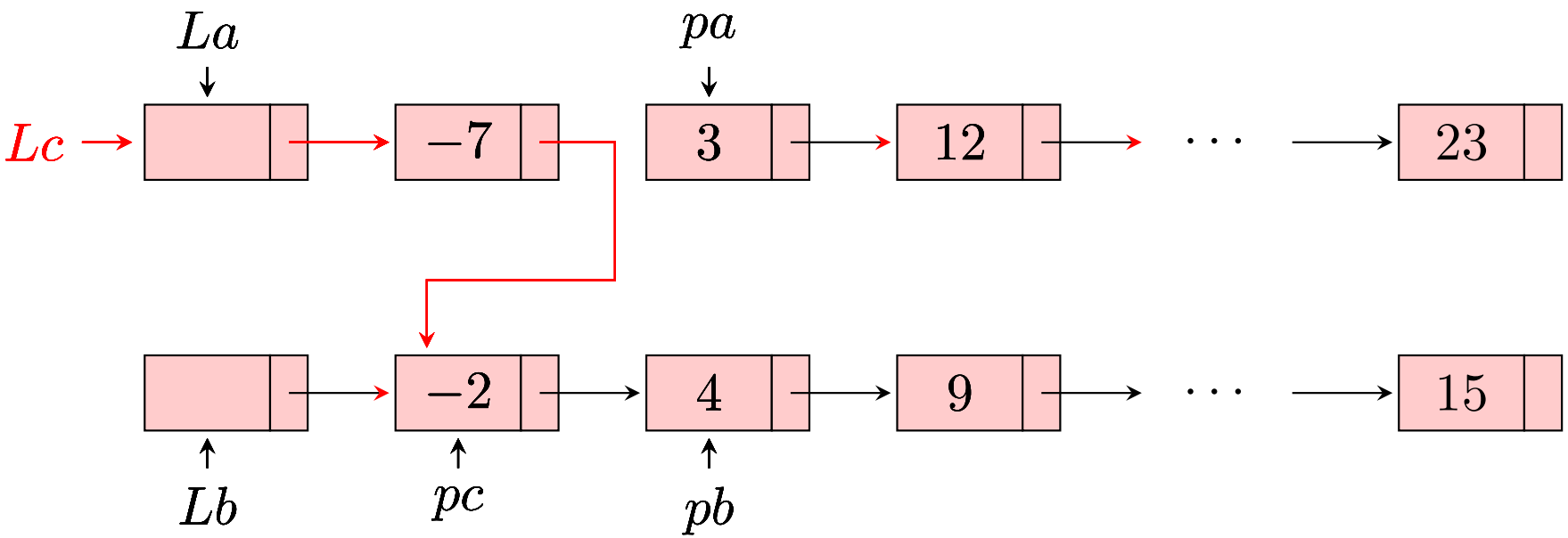


图: 合并了值为$-7,-2$的结点后的示意图

- pa, pb分别是待考察的两个链表的当前结点；
- pc是合并过程中合并的链表的最后一个结点。

单链表的合并

```
LinkList Merge_Linklist(LinkList La, LinkList Lb){
  LinkList Lc,pa,pb,pc,ptr;
  Lc=La; pc=La; pa=La->next; pb=Lb->next;
  while(pa!=NULL && pb!=NULL){
    if (pa->data<pb->data){
      pc->next=pa; pc=pa; pa=pa->next; }
    if (pa->data<pb->data){
      pc->next=pb; pc=pb; pb=pb->next; }
    if (pa->data==pb->data){
      pc->next=pa; pc=pa; pa=pa->next;
      ptr=pb; pb=pb->next; free(ptr); }
  }
  if(pa!=NULL) pc->next=pa;
  else pc->next=pb;
  free(Lb);
  return(Lc);
}
```

单链表的合并

时间复杂度

若La, Lb两个链表的长度分别为m, n，则链表合并的时间复杂度为$O(m+n)$。