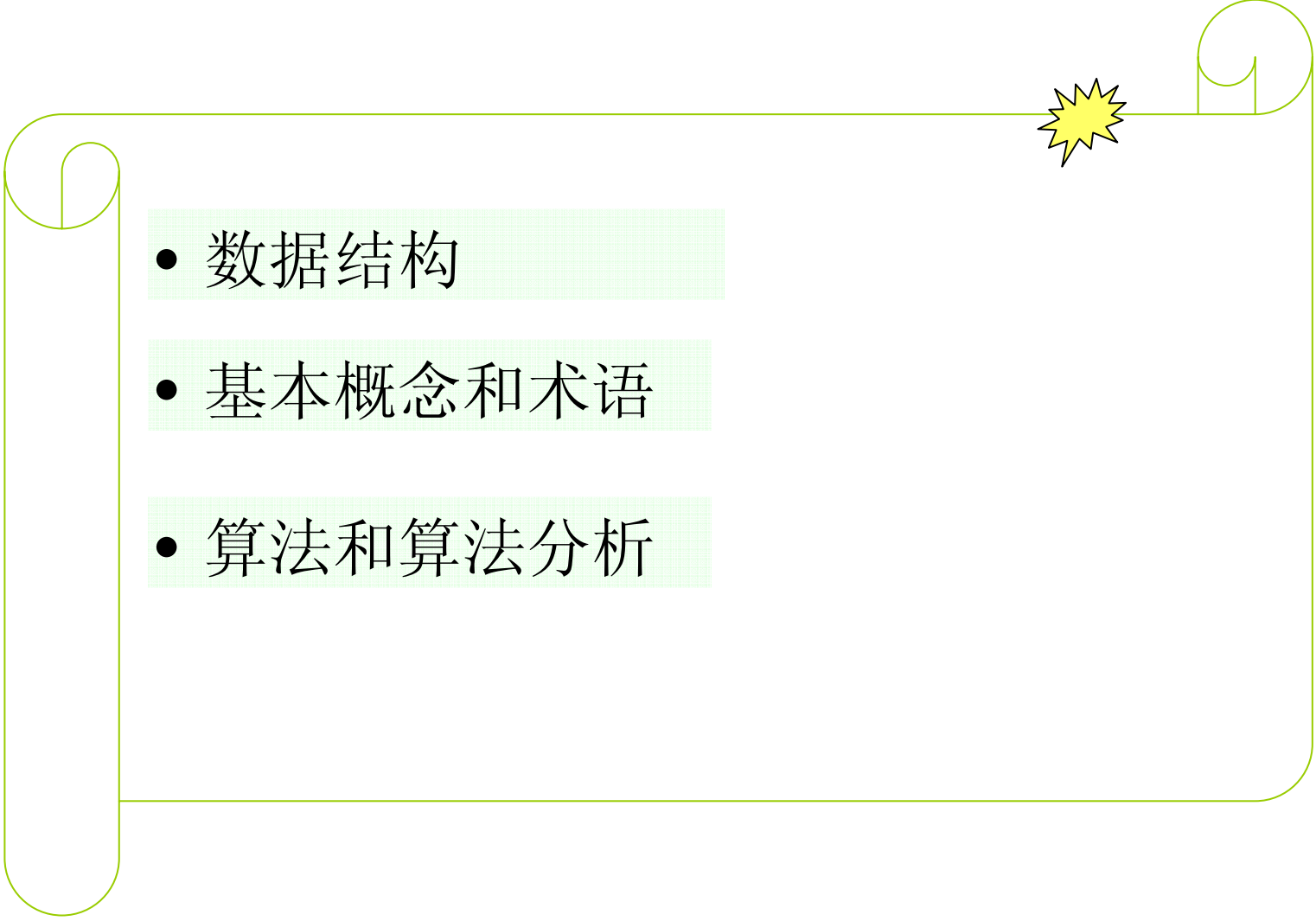


# 数据结构

## 第二讲



# 上讲主要内容

- 
- 数据结构
  - 基本概念和术语
  - 算法和算法分析

# 线 性 表

## 本讲主要内容

**线性表的基本概念**

**顺序表**

# 线性表的举例

[实例1]

线性表



英文字母表 (A, B, C, ..., Z)

数据元素



表中的每一个英文字母

# 线性表的举例

## [实例2]

线性表



学生成绩登记表。

学号	姓名	性别	成绩
02042001	尚雷雷	男	
02042002	倪亮	男	
02042003	李思默	男	
...	...	...	...

数据元素



表中的每一个学生记录

# 线性表的基本概念

线性表的逻辑结构定义

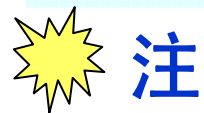
表的长度



线性表是由 $n$  ( $n \geq 0$ ) 个数据元素  
 $a_1, a_2, \dots, a_n$  构成的有限序列。

记作:  $(a_1, a_2, \dots, a_n)$   
 $(n > 0)$

# 线性表的基本概念



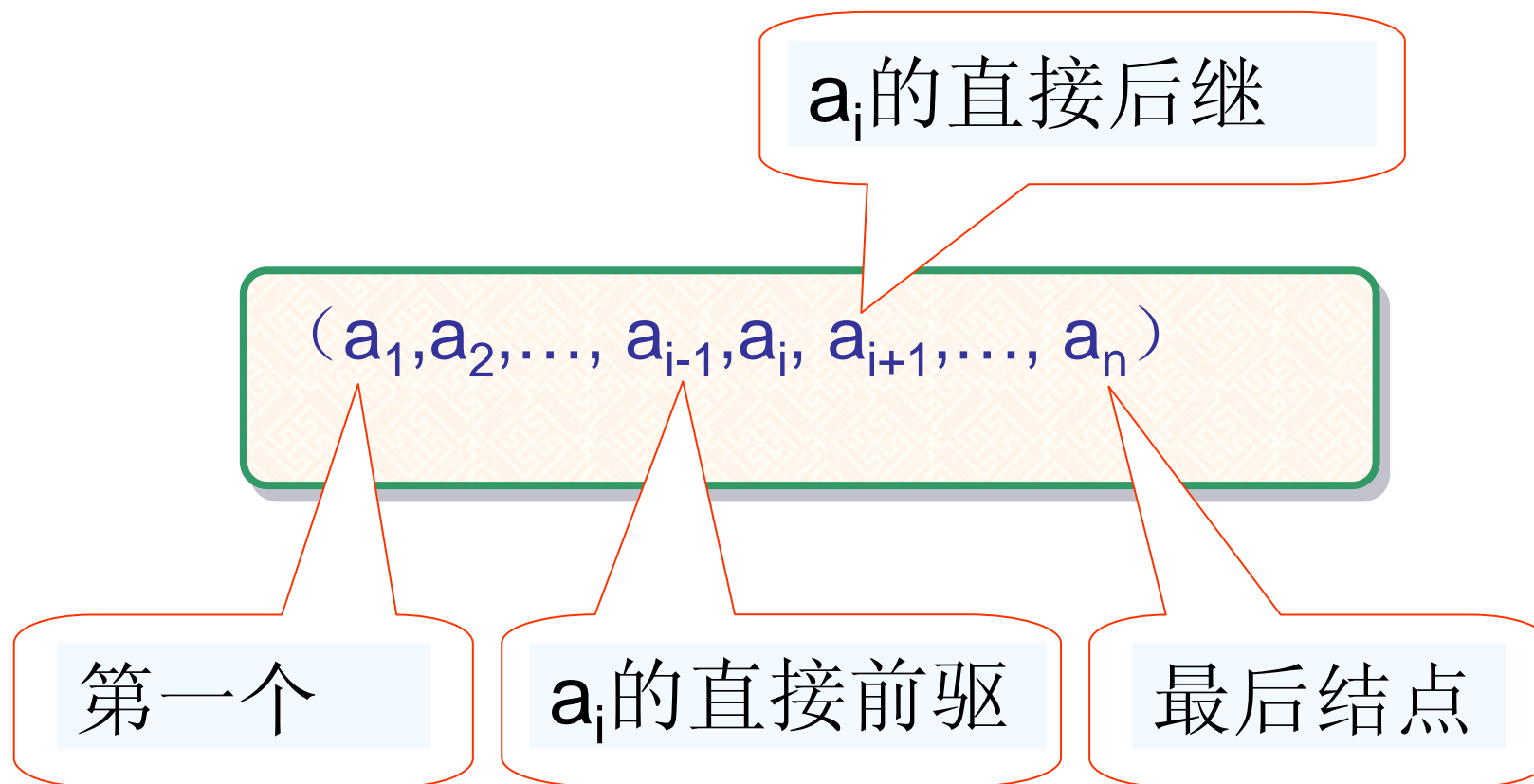
- ✓ 当 $n=0$ 时称为空表
- ✓  $a_i$  是第 $i$ 个数据元素。
- ✓  $i$ 为数据元素在线性表中的位置。
- ✓ 数据元素 $a_i$ 的具体含义，在不同的情况下可以不同。
- ✓ 同一线性表中的元素必定具有相同的特性。

# 线性表的特点

- 对于非空的线性表，有且仅有一个“第一个”的数据元素 $a_1$ ，有且仅有一个“最后一个”的数据元素 $a_n$ 。
- 当 $i=1,2,\dots,n-1$ 时， $a_i$ 有且仅有一个直接后继 $a_{i+1}$ ；当 $i=2,3,\dots,n$ 时， $a_i$ 有且仅有一个直接前趋 $a_{i-1}$ 。

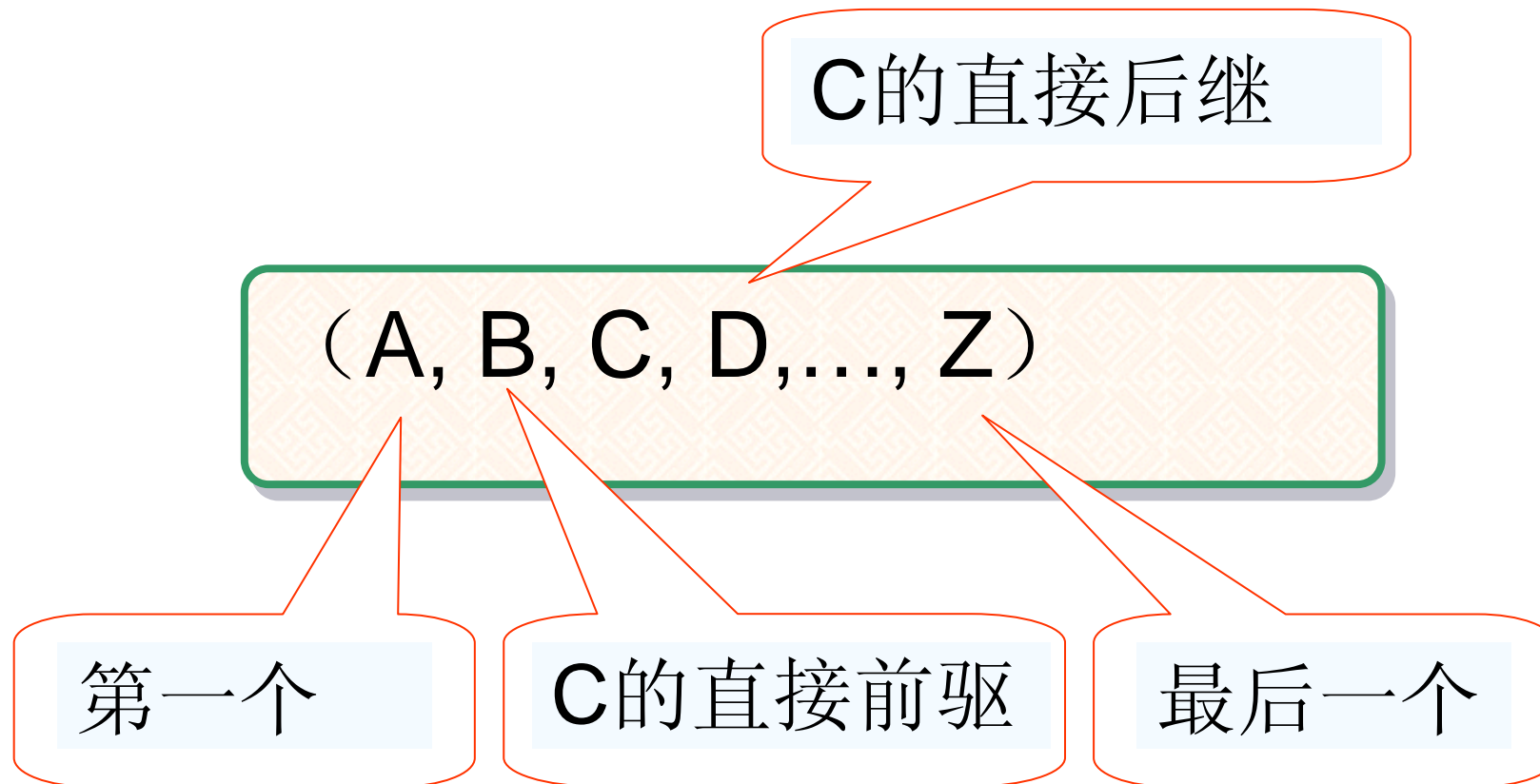


# 线性表的举例



# 线性表的举例

## [实例1]



# 线性表的举例

[实例2]

第一个

学号	姓名	性别	成绩
02053001	尚雷雷	男	
02042002	倪亮	男	
02042003	李思默	男	
02042004	雷冰	男	
...	...	...	...
02042093	王敏旻	男	

003的直接前驱

003的直接后继

最后一个

# 抽象类型的线性表的定义

ADT list{

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

数据关系:  $R1=\{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作:

InitList(&L)

操作的结果: 构成一个空的线性表L。

DestroyList(&L)

初始条件: 线性表L已存在。

操作的结果: 销毁线性表L。

# 抽象类型的线性表的定义

## ClearList(&L)

初始条件：线性表L已存在。

操作的结果：将L置为空表。

## ListEmpty(&L)

初始条件：线性表L已存在。

操作的结果：若是L是空表，则返回TRUE，否则返回FALSE。

## List Length(&L)

初始条件：线性表L已存在。

操作的结果：返回L中数据元素的个数。

# 抽象类型的线性表的定义

**GetElem(L, i, &e)**

初始条件：线性表L已存在，

$1 \leq i \leq \text{ListLength}(L)$ 。

操作的结果：用e返回L中第i个元素的值。

**LocateElem(L, e, compare())**

初始条件：线性表L已存在，

**compare()**是数据元素判定函数。

操作的结果：若是L中第一个与e满足关系**compare()**的数据元素的位序。若这样的数据元素不存在，则返回值为0。

# 抽象类型的线性表的定义

**PriorElem(L, cur\_e, &pre\_e)**

初始条件：线性表L已存在。

操作的结果：若cur\_e是L的数据元素，且不是第1个元素，则用pre\_e返回它的前驱，否则操作失败，pre\_e 无意义。

**NextElem(L, cur\_e, &next\_e)**

初始条件：线性表L已存在。

操作的结果：若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e 无意义。

# 抽象类型的线性表的定义

**ListInsert(&L, i, e)**

初始条件：线性表L已存在，

$1 \leq i \leq \text{ListLength}(L) + 1$  。

操作的结果：在L的第i个位置之前插入新的数据元素e，L的长度加1。

**ListDelete(&L, i, &e)**

初始条件：线性表L已存在，

$1 \leq i \leq \text{ListLength}(L)$  。

操作的结果：删除L的第i个数据元素，并用e返回其值，L的长度减1。



# 抽象类型的线性表的定义

**ListTraverse(&L, visit())**

初始条件：线性表L已存在。

操作的结果：依次对L的每个数据元素调用函数visit()，一旦visit()失败，则操作失败。

## 算法2.1

### 合并线性表

```
void union(List &La,List Lb)
```

```
{ la_len=listLength(La);
```

```
  lb_len=listLength(Lb);
```

```
  for (i=1;i≤lb_len;i++)
```

```
    { GetLem(Lb,i,e);
```

```
      if (!LocateElem(La,e,equal))
```

```
        ListInsert(La,++la_len,e);
```

```
    }
```

```
}
```

```
/* 将所有在线性表Lb中但不在  
   La中的数据元素插入到La  
   中*/
```

```
/*求线性表表长*/
```

```
//取Lb的第i个数据元素赋给e  
/*La中不存在和e相同的数据元  
   素，则插入之*/
```

```
/*Union */
```

时间复杂度：  $O(\text{Listlength}(la) * \text{listlength}(lb))$

## 算法2.2

### 合并有序线性表

<pre>void Mergelist(List La, List Lb, List &amp;Lc)  {  InitList(Lc);    i=j=1; k=0;    la_len=listLength(La);    lb_len=listLength(Lb);    while ((i&lt;=la_len)&amp;&amp;( j&lt;=lb_len))    {  GetLem(La,i,ai);       GetLem(Lb,j,bj);</pre>	<pre>/* 已知线性表la和lb中的数据    元素按值非递减排列。归并    la和lb得到新的线性表Lc,    Lc的数据元素也按值非递    增排列。*/  /*求线性表表长*/  /*La和lb都未合并完*/</pre>
---	---

## 算法2.2

### 合并有序线性表

```
if (ai<=bj)
    { ListInsert(Lc,++k, ai);
      ++i;
    }
else
    { ListInsert(Lc,++k, bj);
      ++j;
    }
}
```

## 算法2.2

### 合并有序线性表

```
while (i<=la_len)
{ GetElem(La,i++,ai);
  ListInsert(Lc,++k,ai);
}
while (i<=lb_len)
{ GetElem(La,j++,bj);
  ListInsert(Lc,++k,bj);
}
}
```

**/\*MergeList \*/**

时间复杂度：  $O(\text{Listlength}(la) + \text{listlength}(lb))$

# 顺序表

用一组地址连续的存储单元依次存储线性表的元素

存储地址    内存状态    元素序号

<b>b</b>	<b>a<sub>1</sub></b>	<b>1</b>
<b>b+c</b>	<b>a<sub>2</sub></b>	<b>2</b>
<b>⋮</b>	<b>...</b>	<b>⋮</b>
<b>b+(i-1)*c</b>	<b>a<sub>i</sub></b>	<b>i</b>
<b>⋮</b>	<b>...</b>	<b>⋮</b>
<b>b+(n-1)*c</b>	<b>a<sub>n</sub></b>	<b>n</b>
<b>b+n*c</b>		备用区

线性表 ( $a_1, a_2, \dots, a_n$ )

起始位置或基地址:  $\text{Loc}(a_1)$

每个元素占用:  $c$ 个空间

线性表的第 $i$  ( $1 \leq i \leq n$ ) 个  
元素  $a_i$  的存储位置为

$$\text{Loc}(a_i) = \text{Loc}(a_1) + (i-1)*c$$

# 顺序表

✓线性表的这种机内表示称做线性表的**顺序存储结构**或**顺序映象**。

✓线性表的顺序存储结构是一种**随机存取**的存储结构。



**顺序表：**线性表的顺序存储结构，即用向量实现的线性表

# Typedef

- ❖ typedef为C语言的关键字，作用是作为一种数据类型定义一个新名字。
- ❖ 在编程中使用typedef目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。
- ❖ 格式    typedef 旧类型 新类型名1， 新类型名2；
- ❖ 例1： typedef long byte\_4;

❖ 例2：

```
typedef struct tagNode
{
    char *pItem;
    struct tagNode *pNext;
} *pNode;
```

```
struct tagNode
{
    char *pItem;
    struct tagNode *pNext;
}
Typedef tagNode *pNode
```



# Malloc和Realloc

- ❖ Malloc是在申请动态内存的函数。使用前需包含头文件`malloc.h`
  - ❖ 格式 类型 `malloc` (大小)
  - ❖ 例 `(int *) malloc(n*sizeof(int))`
- 
- ❖ `realloc` 可以对给定的指针`p`所指的空间进行扩大或者缩小，无论是扩张或是缩小，原有内存中的内容将保持不变。
  - ❖ 例 `(int *) realloc (p, sizeof(int) *m);`

# 线性表的顺序存储结构

```
#define LIST_INIT_SIZE 100
    /* 线性表存储空间的初始分配量*/
#define LISTINCREMENT 10
    /* 线性表存储空间的分配增量*/
typedef struct
{ ElemType *elem; /* 存储空间的基址*/
  int length;      /* 当前长度*/
  int listsize;    /* 当前分配的存储容量（以sizeof
    (ElemType) 为单位）*/
} SqList;
```

## 算法2.3 顺序表初始化

```
Status InitList-Sq(SqList &L) /* 构造一个空的线性表*/  
  
{ L.elem=(ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType))  
  if (!L.elem) exit(OVERFLOW);  
  L.length=0;  
  L.listsize=LIST_INIT_SIZE;  
  return OK;  
} //InitList_Sq
```

# 插入运算

逻辑关系

原来线性表

$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

$a_i$ 的直接前驱

插入后线性表

$(a_1, a_2, \dots, a_{i-1}, x, a_i, a_{i+1}, \dots, a_n)$

$a_i$ 的直接前驱

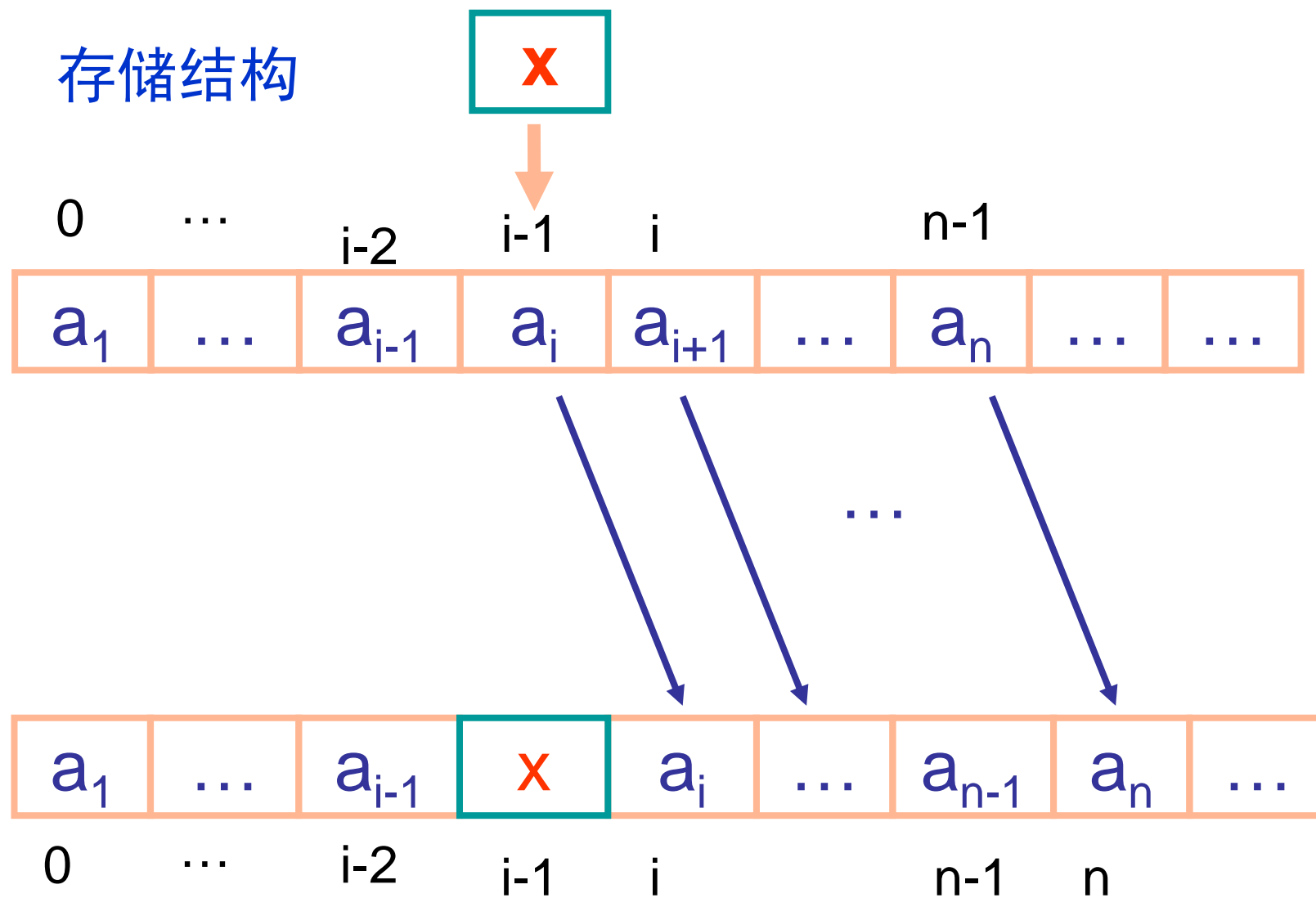
$a_{i-1}$

及后继

$a_{i-1}$ 的直接前驱

# 插入运算

存储结构



# 顺序表的基本运算

## 1. 插入运算

先移动后插入

	插入前	移动后	插入后
0	$a_1$	$a_1$	$a_1$
1	$a_2$	$a_2$	$a_2$
	$\vdots$	$\vdots$	$\vdots$
$i-2$	$a_{i-1}$	$a_{i-1}$	$a_{i-1}$
插入 $x$	$a_i$		$x$
$i$	$a_{i+1}$	$a_i$	$a_i$
	$\vdots$	$\vdots$	$\vdots$
	$a_n$	$a_{n-1}$	$a_{n-1}$
		$a_n$	$a_n$

## 算法2.4 插入运算的实现

```
Status ListInsert_Sq(SqList &L, int i, ElemType e)
```

```

{ If ((i<1)||i>L.length+1)) return ERROR; /* 非法位置 */
  if (L.length>=L.listsize) //当前存储空间已满，增加分配
  { newbase=(ElemType *)realloc(L.elem,
    (L.Listsize+LISTINCREMENT)*sizeof(ElemType));
    if (!newbase) exit(OVERFLOW); //存储分配失败
    L.elem=newbase; //新基址
    L.listsize+=LISTINCREMENT; //增加存储容量
  }
  q=&(L.elem[i-1]); //q为插入位置
  for (p=&(L.elem[L.length-1]); p>=q;--p) //插入位置后的元素后移
    *(p+1)=*p;
  *q=e; //插入e
  ++L.length; //表长加1
  return OK;
} //ListInsert_Sq

```

# 算法分析

- 从上面算法可看出，顺序表的插入运算，其时间主要耗费在移动元素上，而移动元素的个数不仅依赖于表长 $n$ ，而且还与插入的位置 $i$ 有关。
- 在第 $i$ 个位置上插入一个元素的移动元素的次数为  $n - i + 1$
- 假设  $p_i$  是在第 $i$ 个位置上插入一个元素的概率，
- 则在长度为 $n$ 的线性表中插入一个元素时所需移动元素次数的期望值（平均次数）为

$$E_{IS} = \sum_{i=1}^{n+1} p_i (n - i + 1)$$

- 任何位置上插入元素的概率相等，则  $p_i = \frac{1}{n+1}$  故

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



# 删除运算

逻辑关系

原来线性表

$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

$a_{i+1}$ 的直接前驱

$a_{i-1}$ 的直接前驱

删除后线性表

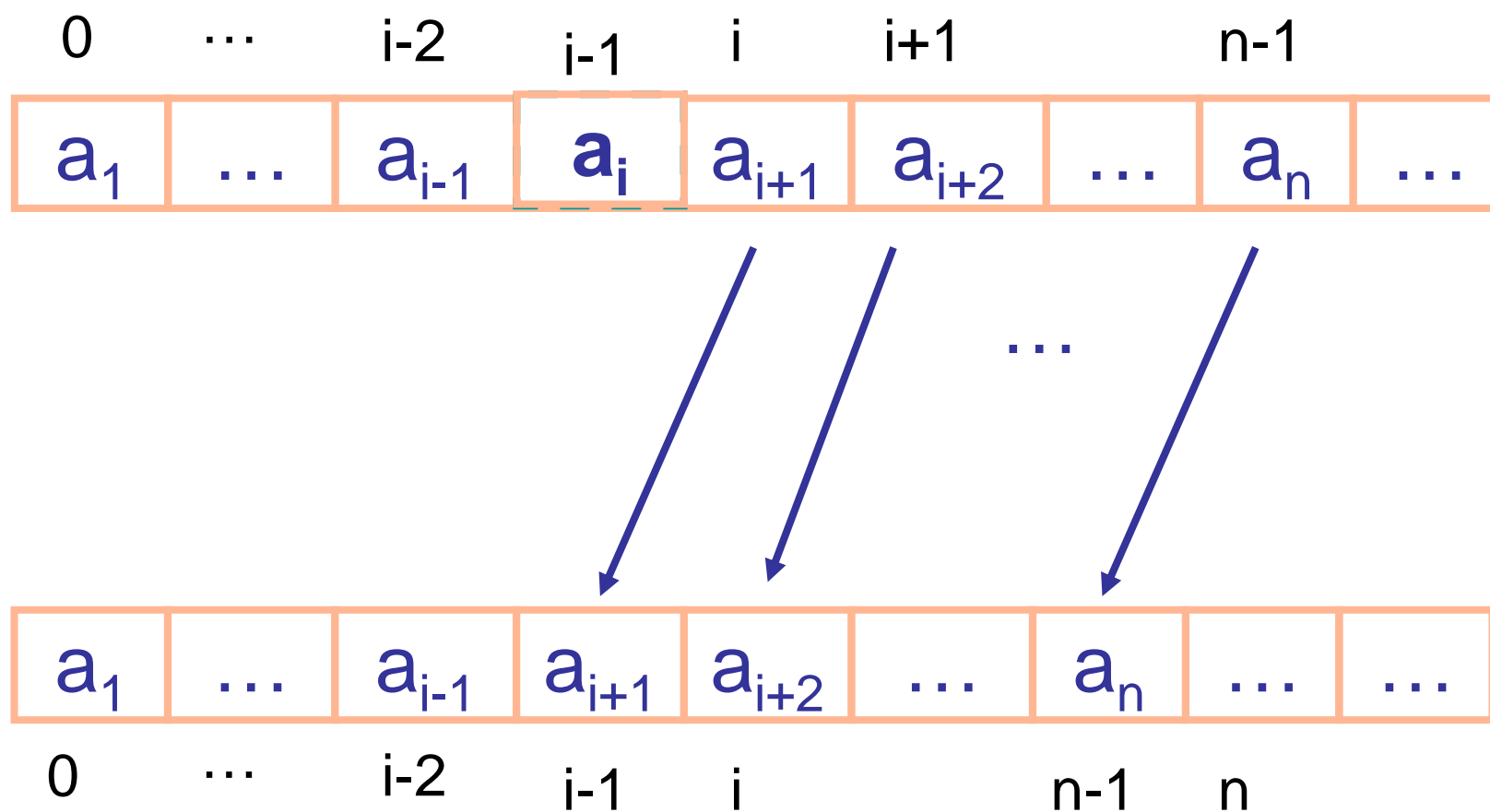
$(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

$a_{i-1}$ 的直接后继

$a_{i+1}$ 的直接前驱

# 删除运算

## 存储结构



# 删除运算

## 2.删除运算

直接移动数据元素

	删 除 前	删 除	删 除 后 (移 动 )
0	$a_1$	$a_1$	$a_1$
1	$a_2$	$a_2$	$a_2$
	$\vdots$	$\vdots$	$\vdots$
$i-2$	$a_{i-1}$	$a_{i-1}$	$a_{i-1}$
$i-1$	$a_i$		$a_{i+1}$
$i$	$a_{i+1}$	$a_{i+1}$	$a_{i+2}$
	$\vdots$	$\vdots$	$\vdots$
	$a_{n-1}$	$a_{n-1}$	$a_n$
	$a_n$	$a_n$	

## 算法2.4 删除运算的实现

Status ListDelete\_Sq(SqList &L, int i, ElemType &e)

```
{ If ((i<1)||i>L.length)) return ERROR;    /* 非法位置 */  
  p=&(L.elem[i-1]);                          //p为被删除元素的位置  
  e=*p;                                       //被删除的元素值赋给e  
  q=L.elem+L.length-1;                     //q为表尾元素的位置  
  for (++p; p<=q;++p)                       //删除元素后元素前移  
    *(p-1)=*p;  
  - -L.length;                              //表长减1  
  return OK;  
} //ListInsert_Sq
```

# 算法分析

- 从上面算法可看出，顺序表的删除运算，其时间主要耗费在移动元素上，而移动元素的个数不仅依赖于表长 $n$ ，而且还与删除的位置 $i$ 有关。
- 在第 $i$ 个位置上删除一个元素的移动元素的次数为  $n - i$
- 假设  $p_i$  是在第 $i$ 个位置上删除一个元素的概率，
- 则在长度为 $n$ 的线性表中删除一个元素时所需移动元素次数的期望值（平均次数）为

$$E_{DE} = \sum_{i=1}^n p_i (n - i)$$

- 任何位置上插入元素的概率相等，则  $p_i = \frac{1}{n}$  故

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

## 算法2.6 定位的实现

```
int LocateElem_Sq(SqList &L, ElemType e, Status
(*compare)(ElemType a, ElemType b)) /*在顺序表L中查
找第1个值与e满足compare()的元素的位置，若找到，则
返回其在L中的序位，否则返回0*/
```

```
{ i=1;
  p=L.elem;           //p的初值为第1个元素的起始位置
  while(i<=L.length&&!(*compare(*p++,e)) ++i
    if (i<=L.length) return i;
    else
      return 0;
} //LocateElem_Sq
```

## 算法2.2

### 合并有序线性表

```
void Mergelist_Sq(List La, List Lb, List  
    &lc)
```

```
{ pa=La.elem;  
  pb=Lb.elem;  
  Lc.Listsize=Lc.length=la.length  
              +Lb.length;  
  pc=Lc.elem=(ElemType *)malloc  
    (Lc.Listsize*sizeof(ElemType));
```

```
/* 已知线性表la和lb中的数据  
   元素按值非递减排列。归并  
   la和lb得到新的线性表Lc,  
   Lc的数据元素也按值非递  
   增排列。*/
```

## 算法2.2

### 合并有序线性表

<pre>if (!Le.elem) exit(OVERFLOW); pa_last=La.elem+la.length-1; pb_last=Lb.elem+lb.length-1; while (pa&lt;=pa_last&amp;&amp;pb&lt;=pb_last)     { if(*pa&lt;=*pb)         *pc++=*pa++;       else         *pc++=*pb++;     } while (pa&lt;=pa_last) *pc++=*pa++; while (pb&lt;=pb_last) *pc++=*pb++; } //MergeList_Sq</pre>	<pre>//存储分配失败  //归并  //插入La的剩余元素 //插入Lb的剩余元素</pre>
---	--



# 归并有序表效率高的原因

1.  $L_a$ 和 $L_b$ 有序，则无需在表中从头至尾搜索；
2. 插入操作是借助复制来完成的；

# 小结

线性表

$n$ 个元素构成的有限序列  
当  $(n > 0)$ ，记  $(a_1, a_2, \dots, a_n)$

顺序表

线性表的顺序存储结构，即用  
向量实现的线性表

运算

插入

删除

定位