



# 数据结构与算法（二）

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写  
高等教育出版社，2008.6（“十一五”国家级规划教材）

<https://pkumooc.coursera.org/bdsalgo-001/>

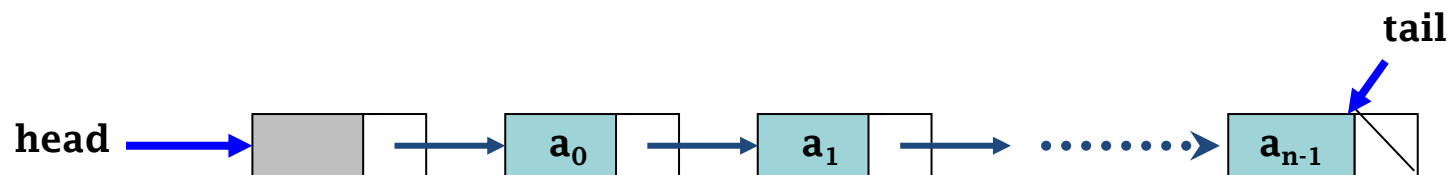
## 第二章 线性表

• 2.1 线性表  $\{a_0, a_1, \dots, a_{n-1}\}$

• 2.2 顺序表 

$a_0$	$a_1$	$a_2$	$\dots \dots$		$a_{n-1}$
-------	-------	-------	---------------	--	-----------

• 2.3 链表



• 2.4 顺序表和链表的比较

## 2.1 线性表

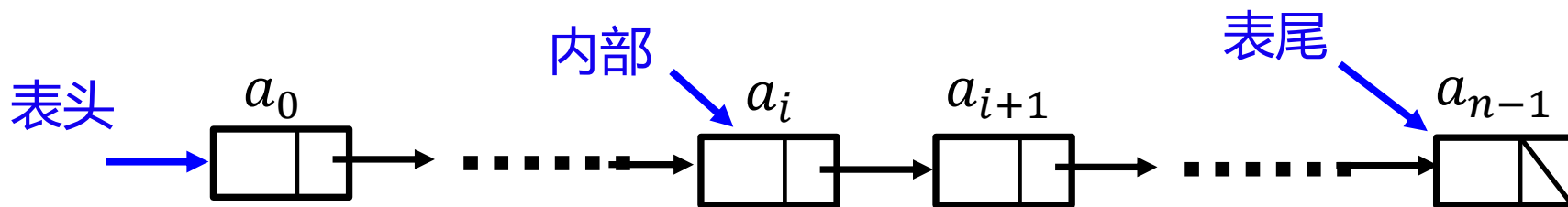
## 线性表的概念

- 线性表简称表，是零个或多个元素的有穷序列，通常可以表示成  $k_0, k_1, \dots, k_{n-1}$  ( $n \geq 1$ )
  - 表目：线性表中的元素（可包含多个数据项，记录）
  - 索引（下标）： $i$  称为表目  $k_i$  的“索引”或“下标”
  - 表的长度：线性表中所含元素的个数  $n$
  - 空表：长度为零的线性表 ( $n = 0$ )
- 线性表特点：
  - 操作灵活，其长度可以增长、缩短



# 线性结构

- 二元组  $B = (K, R)$   $K = \{a_0, a_1, \dots, a_{n-1}\}$   $R = \{r\}$ 
  - 有一个唯一的**开始结点**，它没有前驱，有一个唯一的直接后继
  - 一个唯一的**终止结点**，它有一个唯一的直接前驱，而没有后继
  - 其它的结点皆称为**内部结点**，每一个内部结点都有且仅有一个唯一的直接前驱，也有一个唯一的直接后继
- $\langle a_i, a_{i+1} \rangle$   $a_i$ 是 $a_{i+1}$ 的前驱， $a_{i+1}$ 是 $a_i$ 的后继
- 前驱/后继关系 $r$ ，具有**反对称性**和**传递性**



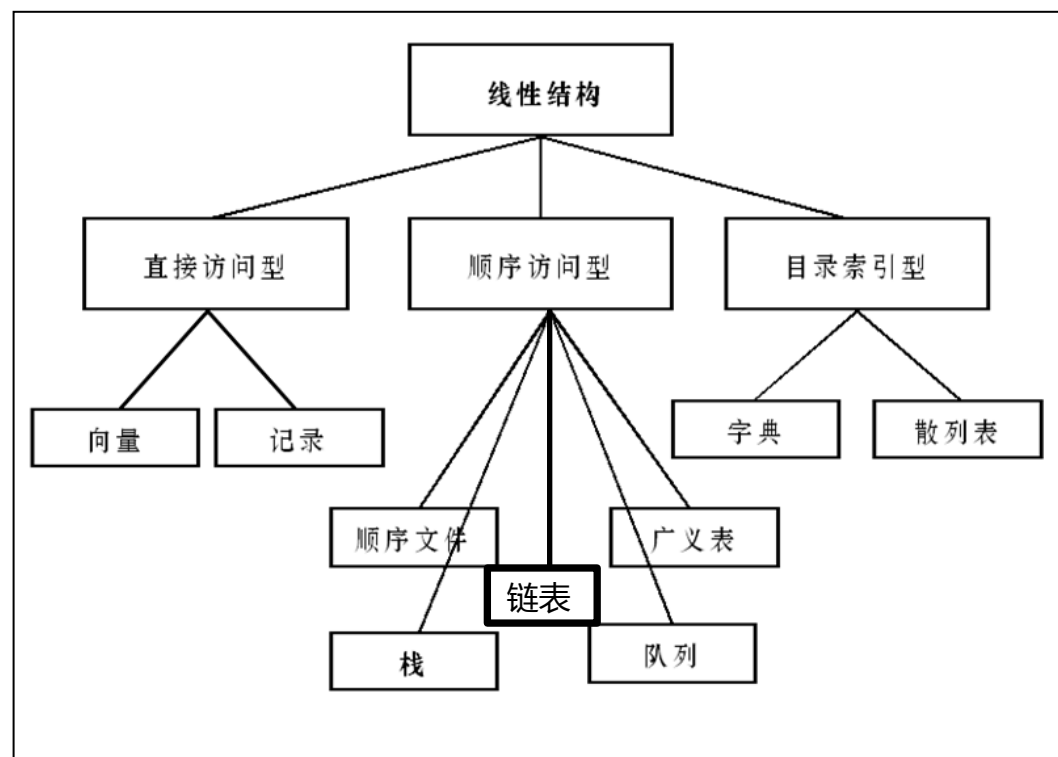
# 线性结构

- 特点

- 均匀性：虽然不同线性表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有**相同的数据类型**和**长度**
- 有序性：各数据元素在线性表中都有自己的位置，且数据元素之间的**相对位置**是**线性的**

# 线性结构

- 按复杂程度划分
  - 简单的：线性表、栈、队列、散列表
  - 高级的：广义表、多维数组、文件.....
- 按访问方式划分
  - 直接访问型 (direct access)
  - 顺序访问型 (sequential access)
  - 目录索引型 (directory access)





# 线性结构

- 按操作划分（详见后）
  - 线性表
    - 所有表目都是同一类型结点的线性表
    - 不限制操作形式
    - 根据存储的不同分为：顺序表，链表
  - 栈 (LIFO, Last In First Out)
    - 插入和删除操作都限制在表的同一端进行
  - 队列 (FIFO, First In First Out)
    - 插入操作在表的一端，删除操作在另一端



## 2.1 线性表

# 2.1 线性表

- 三个方面
  - 线性表的逻辑结构
  - 线性表的存储结构
  - 线性表运算



# 线性表逻辑结构

- 主要属性包括

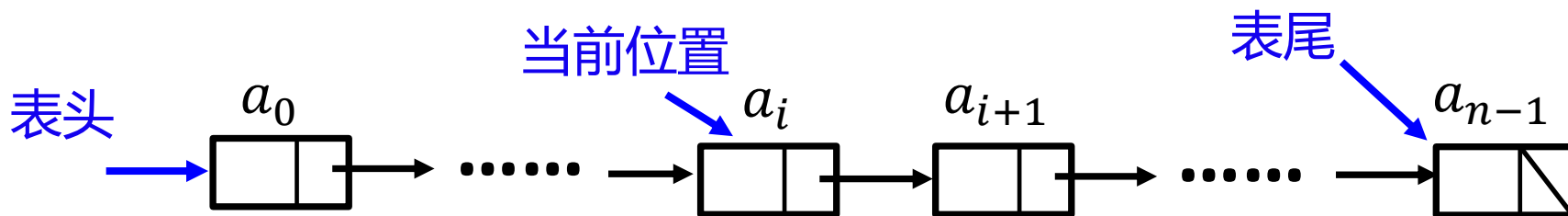
- 线性表的长度 预留存储, 数据规模

- 表头 (head)

- 表尾 (tail)

- 当前位置 (current position)

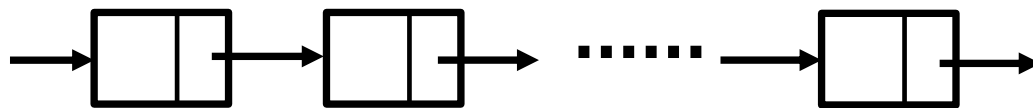
先定位, 再操作



## 线性表分类（按存储）

### · 线性表

- 所有表目都是同一类型结点的线性表
- 不限制操作形式
- 根据存储的不同分为：**顺序表**，**链表**



# 线性表的存储结构

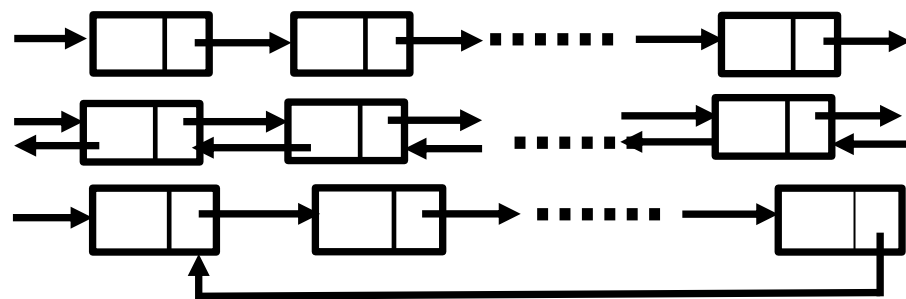
## · 顺序表

- 按索引值从小到大存放在一片相邻的连续区域
- 紧凑结构，存储密度为 1



## · 链表

- 单链表
- 双链表
- 循环链表

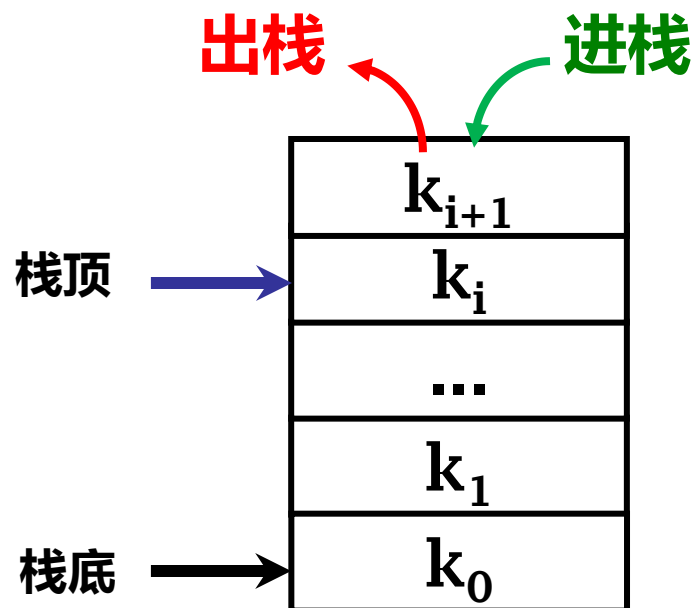


## 线性表分类（按操作）

- 线性表
  - 不限制操作
- 栈
  - 在同一端操作
- 队列
  - 在两端操作

## 线性表分类（按操作）

- 栈 (LIFO, Last In First Out)
  - **插入和删除**操作都限制在表的**同一端**进行



## 线性表分类（按操作）

- 队列 (FIFO, First In First Out)
  - **插入**操作在表的**一端**，**删除**操作在**另一端**
- rear实指

删除

front front rear rear

插入



# 线性表的运算

- 建立线性表
- 清除线性表
- 插入一个新元素
- 删除某个元素
- 修改某个元素
- 排序
- 检索

## 2.1 线性表

## 线性表类模板

```
template <class T> class List {  
    void clear();           // 置空线性表  
    bool isEmpty();        // 线性表为空时，返回 true  
    bool append(const T value);  
                           // 在表尾添加一个元素 value，表的长度增 1  
    bool insert(const int p, const T value);  
                           // 在位置 p 上插入一个元素 value，表的长度增 1  
    bool delete(const int p);  
                           // 删除位置 p 上的元素，表的长度减 1  
    bool getPos(int& p, const T value);  
                           // 查找值为 value 的元素并返回其位置  
    bool getValue(const int p, T& value);  
                           // 把位置 p 元素值返回到变量 value  
    bool setValue(const int p, const T value);  
                           // 用 value 修改位置 p 的元素值  
};
```



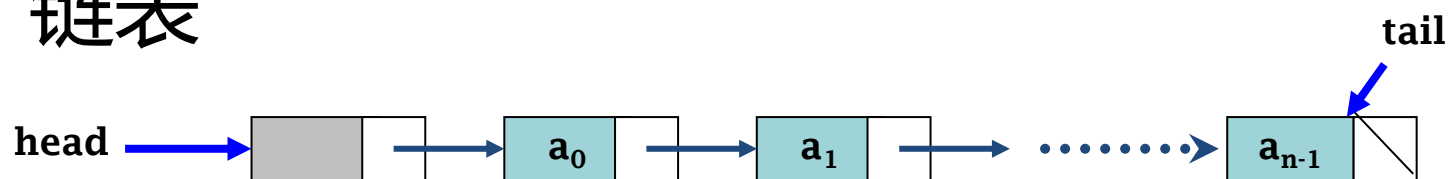


## 思考

- 线性表有哪些分类方式？
- 各种线性表名称中，哪些跟存储结构相关？哪些跟运算相关？

## 第二章 线性表

- 2.1 线性表
- 2.2 顺序表
- 2.3 链表

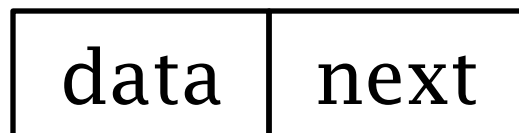


- 2.4 顺序表和链表的比较



## 链表 ( linked list )

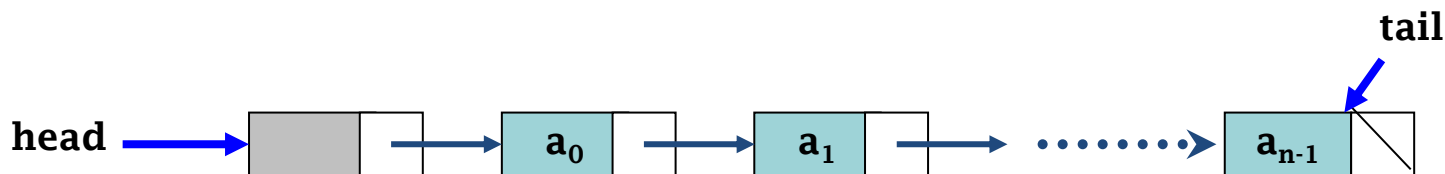
- 通过指针把它的一串存储结点链接成一个链
- 存储结点由两部分组成：
  - 数据域 + 指针域 ( 后继地址 )



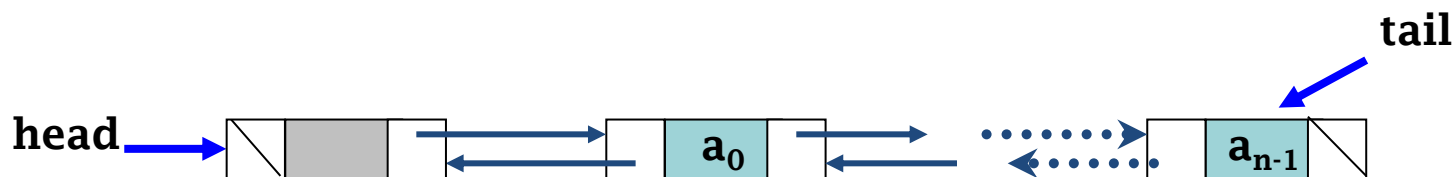
## 2.3 链表

### · 分类（根据链接方式和指针多寡）

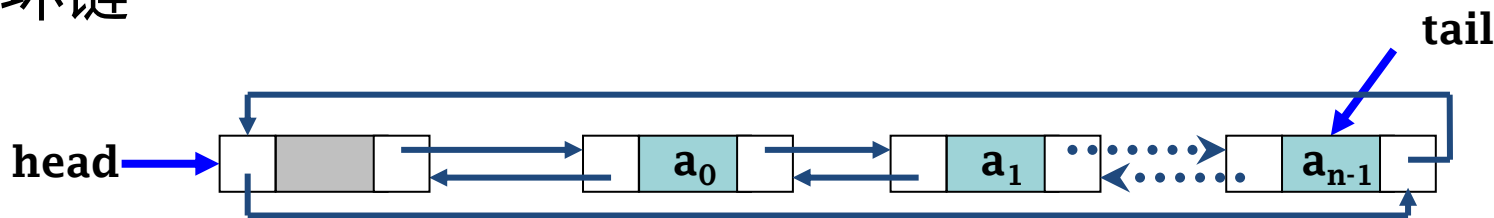
#### – 单链



#### – 双链



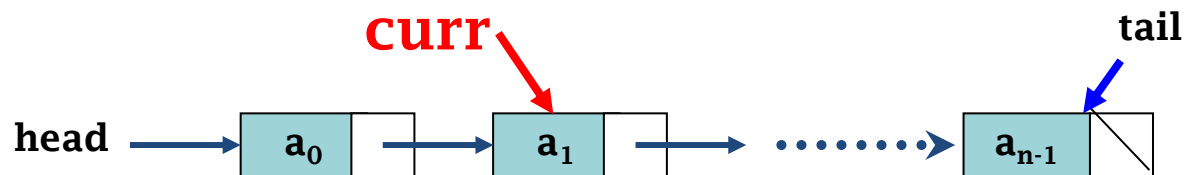
#### – 循环链



# 单链表 ( singly linked list )

## · 简单的单链表

- 整个单链表 : head
- 第一个结点 : head
- 空表判断 : head == NULL
- 当前结点  $a_1$  : curr

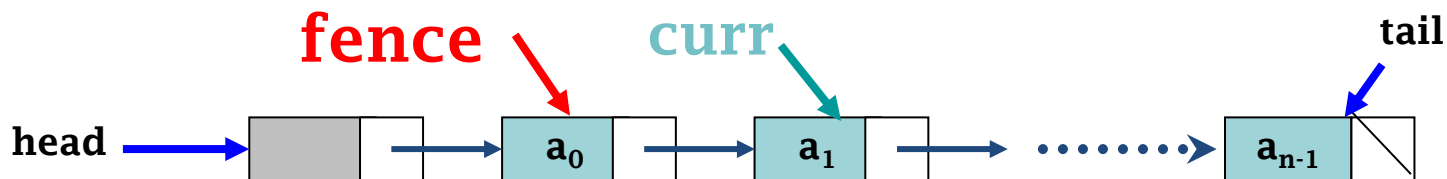


# 单链表 ( singly linked list )

## · 带头结点的单链表

- 整个单链表 : head
- 第一个结点 : head->next , head  $\neq$  NULL
- 空表判断 : head->next == NULL

当前结点 $a_1$  : fence->next (curr 隐含)





## 单链表的结点类型

```
template <class T> class Link {  
    public:  
        T    data;                // 用于保存结点元素的内容  
        Link<T> * next;           // 指向后继结点的指针  
  
    Link(const T info, const Link<T>* nextValue =NULL) {  
        data = info;  
        next = nextValue;  
    }  
    Link(const Link<T>* nextValue) {  
        next = nextValue;  
    }  
};
```



# 单链表类定义

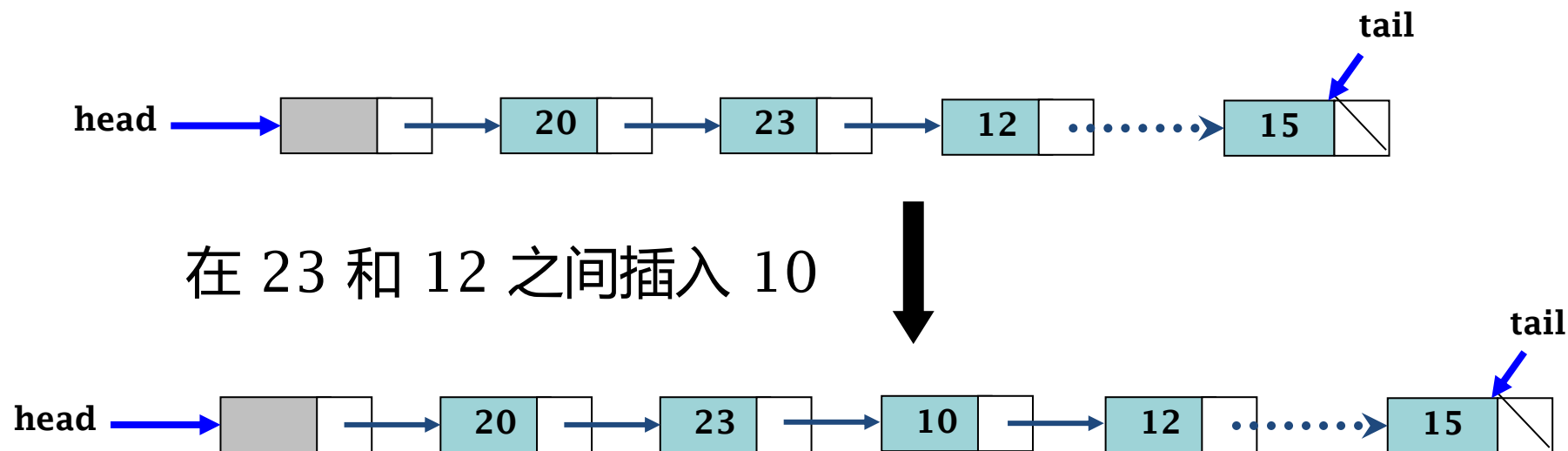
```
template <class T> class lnkList : public List<T> {  
private:  
    Link<T> * head, *tail;           // 单链表的头、尾指针  
    Link<T> *setPos(const int p);     // 第p个元素指针  
public:  
    lnkList(int s);                   // 构造函数  
    ~lnkList();                       // 析构函数  
    bool isEmpty();                   // 判断链表是否为空  
    void clear();                     // 将链表存储的内容清除，成为空表  
    int length();                     // 返回此顺序表的当前实际长度  
    bool append(const T value);        // 表尾添加一个元素 value，表长度增 1  
    bool insert(const int p, const T value); // 位置 p 上插入一个元素  
    bool delete(const int p);          // 删除位置 p 上的元素，表的长度减 1  
    bool getValue(const int p, T& value); // 返回位置 p 的元素值  
    bool getPos(int &p, const T value); // 查找值为 value 的元素  
}
```



# 查找单链表中第 $i$ 个结点

```
// 函数返回值是找到的结点指针
template <class T>          // 线性表的元素类型为 T
Link<T> * lnkList <T>:: setPos(int i) {
    int count = 0;
    if (i == -1)             // i 为 -1 则定位到头结点
        return head;
    // 循链定位, 若i为0则定位到第一个结点
    Link<T> *p = head->next;
    while (p != NULL && count < i) {
        p = p-> next;
        count++;
    };
    // 指向第 i 结点, i = 0,1,..., 当链表中结点数小于 i 时返回 NULL
    return p;
}
```

# 单链表的插入



- 创建新结点
- 新结点指向右边的结点
- 左边结点指向新结点

# 单链表插入算法

```
// 插入数据内容为value的新结点作为第 i 个结点
template <class T>                                // 线性表的元素类型为 T
bool lnkList<T> :: insert(const int i, const T value) {
    Link<T> *p, *q;

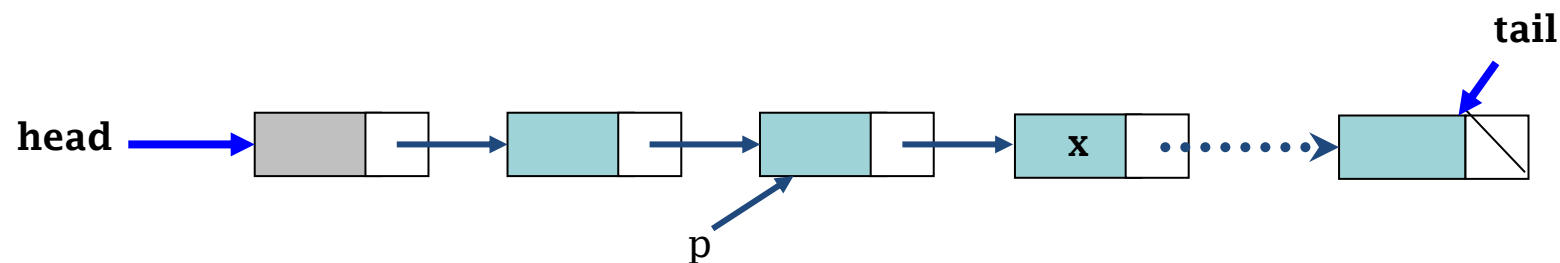
    if ((p = setPos(i - 1)) == NULL) {           // p 是第 i 个结点的前驱
        cout << " 非法插入点" << endl;
        return false;
    }
    q = new Link<T>(value, p->next);
    p->next = q;
    if (p == tail)                                // 插入点在链尾，插入结点成为新的链尾
        tail = q;
    return true;
}
```



## 单链表的删除

- 从链表中删除结点  $x$ 
  - 1. 用  $p$  指向元素  $x$  的结点的前驱结点
  - 2. 删除元素为  $x$  的结点
  - 3. 释放  $x$  占据的空间

## 单链表删除示意

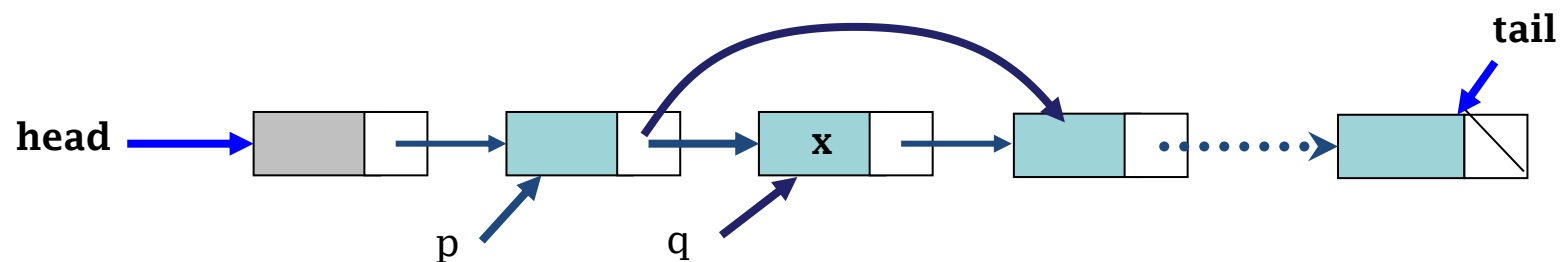


```
p = head;
```

```
while (p->next!=NULL && p->next->info!= x)
```

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

## 删除值为 $x$ 的结点



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

# 单链表删除算法

```
template <class T>                // 线性表的元素类型为 T
bool lnkList<T>::delete((const int i) {
    Link<T> *p, *q;
    // 待删结点不存在，即给定的i大于当前链中元素个数
    if ((p = setPos(i-1)) == NULL || p == tail) {
        cout << " 非法删除点 " << endl;
        return false;
    }
    q = p->next;                // q 是真正待删结点
    if (q == tail) {            // 待删结点为尾结点，则修改尾指针
        tail = p;
        p->next = NULL;
    }
    else                        // 删除结点 q 并修改链指针
        p->next = q->next;
    delete q;
    return true;
}
```



## 单链表上运算的分析

- 对一个结点操作，必先找到它，即用一个指针指向它
- 找单链表中任一结点，**都必须从第一个点开始**

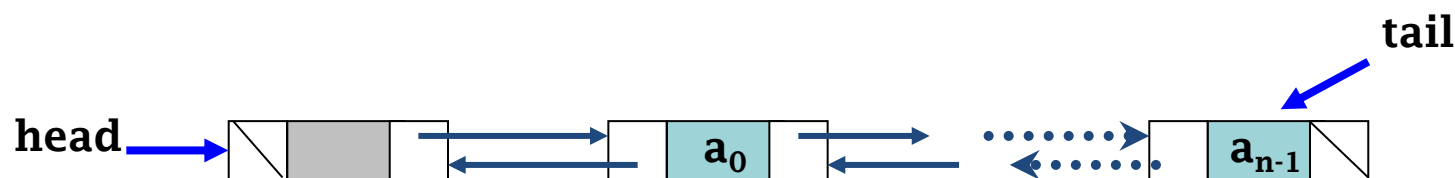
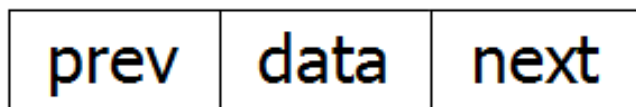
```
p = head;  
while (没有到达) p = p->next;
```

- 单链表的时间复杂度  $O(n)$ 
  - 定位： $O(n)$
  - 插入： $O(n) + O(1)$
  - 删除： $O(n) + O(1)$



# 双链表(double linked list)

- 为弥补单链表的不足,而产生双链表
  - 单链表的 next 字段仅仅指向后继结点,不能有效地找到前驱,反之亦然
  - 增加一个指向前驱的指针



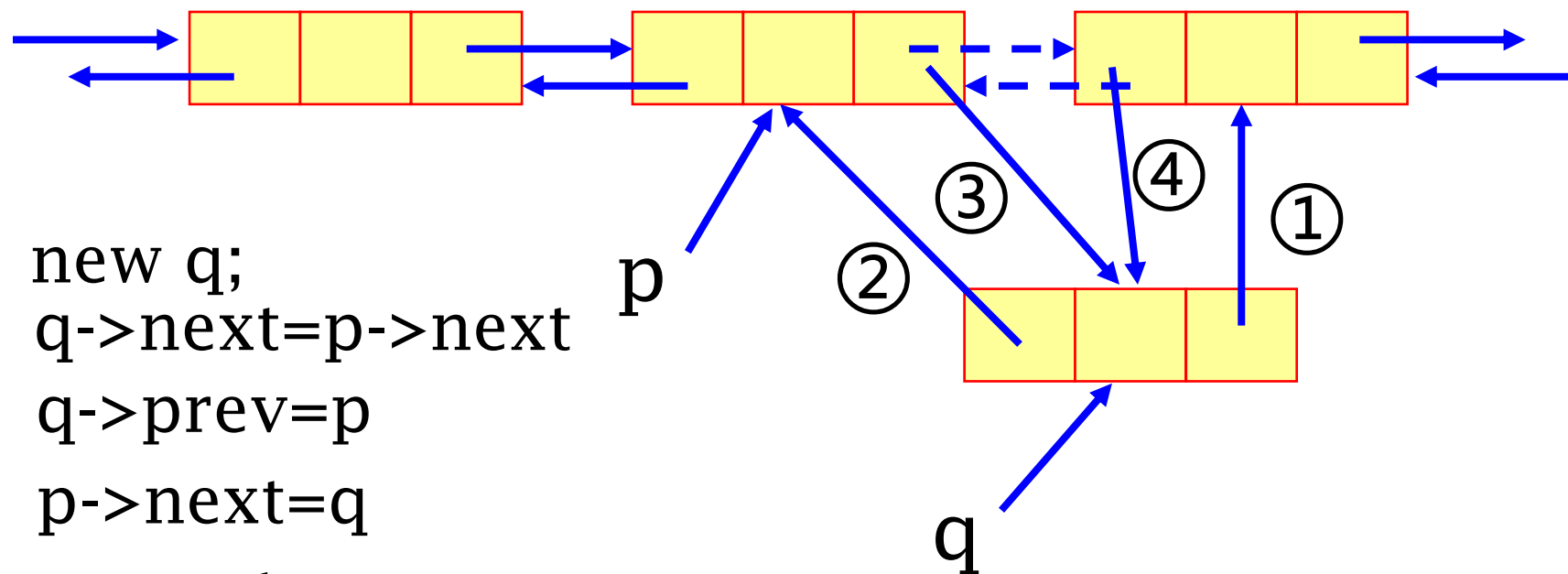


# 双链表及其结点类型的说明

```
template <class T> class Link {  
public:  
    T data;           // 用于保存结点元素的内容  
    Link<T> * next;    // 指向后继结点的指针  
    Link<T> * prev;    // 指向前驱结点的指针  
    Link(const T info, Link<T>* preValue = NULL, Link<T>* nextValue = NULL) {  
        // 给定值和前后指针的构造函数  
        data = info;  
        next = nextValue;  
        prev = preValue;  
    }  
    Link(Link<T>* preValue = NULL, Link<T>* nextValue = NULL) {  
        // 给定前后指针的构造函数  
        next = nextValue;  
        prev = preValue;  
    }  
};
```

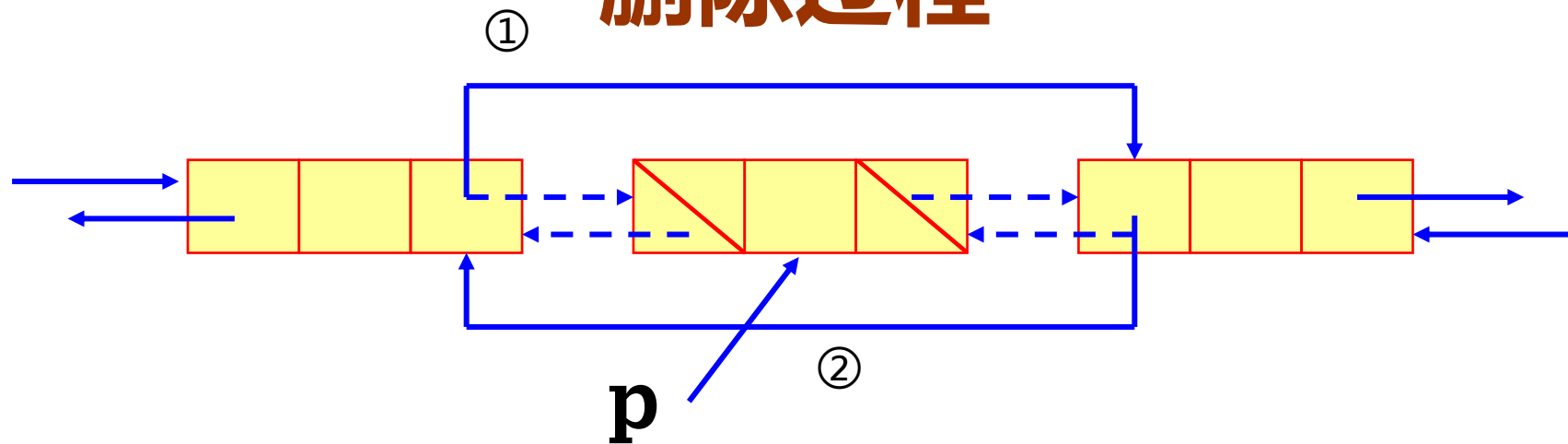
## 双链表插入过程（注意顺序）

在  $p$  所指结点后面插入一个新结点



```
new q;  
q->next=p->next  
q->prev=p  
p->next=q  
q->next->prev=q
```

## 删除过程



删除  $p$  所指的结点

$p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{next}$

$p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}$

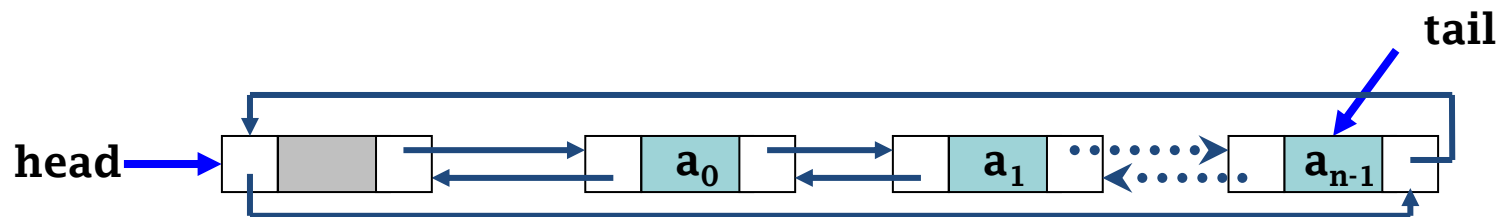
$p \rightarrow \text{next} = \text{NULL}$

$p \rightarrow \text{prev} = \text{NULL}$

- 如果马上删除  $p$ 
  - 则可以不赋空

## 循环链表 (circularly linked list)

- 将单链表或者双链表的头尾结点链接起来，就是一个循环链表
- 不增加额外存储花销，却给不少操作带来了方便
  - 从循环表中任一结点出发，都能访问到表中其他结点





## 链表的边界条件

- 几个特殊点的处理
  - 头指针处理
  - 非循环链表尾结点的指针域保持为 NULL
  - 循环链表尾结点的指针回指头结点
- 链表处理
  - 空链表的特殊处理
  - 插入或删除结点时指针勾链的顺序
  - 指针移动的正确性
    - 插入
    - 查找或遍历

## 思考

- 带表头与不带表头的单链表？
- 处理链表需要注意的问题？

## 第二章 线性表

· 2.1 线性表

· 2.2 顺序表



· 2.3 链表

· 2.4 顺序表和链表的比较





## 2.2 顺序表

- 也称**向量**，采用**定长**的一维数组存储结构
- 主要特性
  - 元素的类型相同
  - 元素**顺序**地存储在连续存储空间中，每一个元素有唯一的索引值
  - 使用常数作为向量长度
- 数组存储
- 读写其元素很方便，通过下标即可指定位置
  - 只要确定了首地址，线性表中任意数据元素都可以随机存取

## 2.2 顺序表

- 元素地址计算如下所示：
  - $Loc(k_i) = Loc(k_0) + c \times i, c = \text{sizeof}(ELEM)$

逻辑地址  
(下标)

0	$k_0$
1	$k_1$
...	...
$i$	$k_i$
...	
$n-1$	$k_{n-1}$

存储地址 数据元素

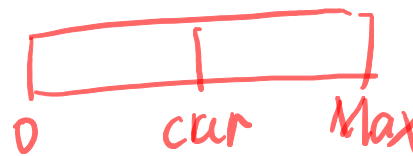
$Loc(k_0)$	$k_0$
$Loc(k_0)+c$	$k_1$
...	...
$Loc(k_0)+i*c$	$k_i$
...	
$Loc(k_0)+(n-1)*c$	$k_{n-1}$

## 2.2 顺序表

## 顺序表类定义

```
class arrList : public List<T> {  
private:  
    T * aList ;  
    int maxSize;  
    int curLen;  
    int position;  
public:  
    arrList(const int size) {  
        maxSize = size; aList = new T[maxSize];  
        curLen = position = 0;  
    }  
    ~arrList() {  
        delete [] aList;  
    }  
};
```

// 顺序表，向量  
// 线性表的取值类型和取值空间  
// 私有变量，存储顺序表的实例  
// 私有变量，顺序表实例的最大长度  
// 私有变量，顺序表实例的当前长度  
// 私有变量，当前处理位置  
// 创建新表，设置表实例的最大长度  
// 析构函数，用于消除该表实例





# 顺序表类定义

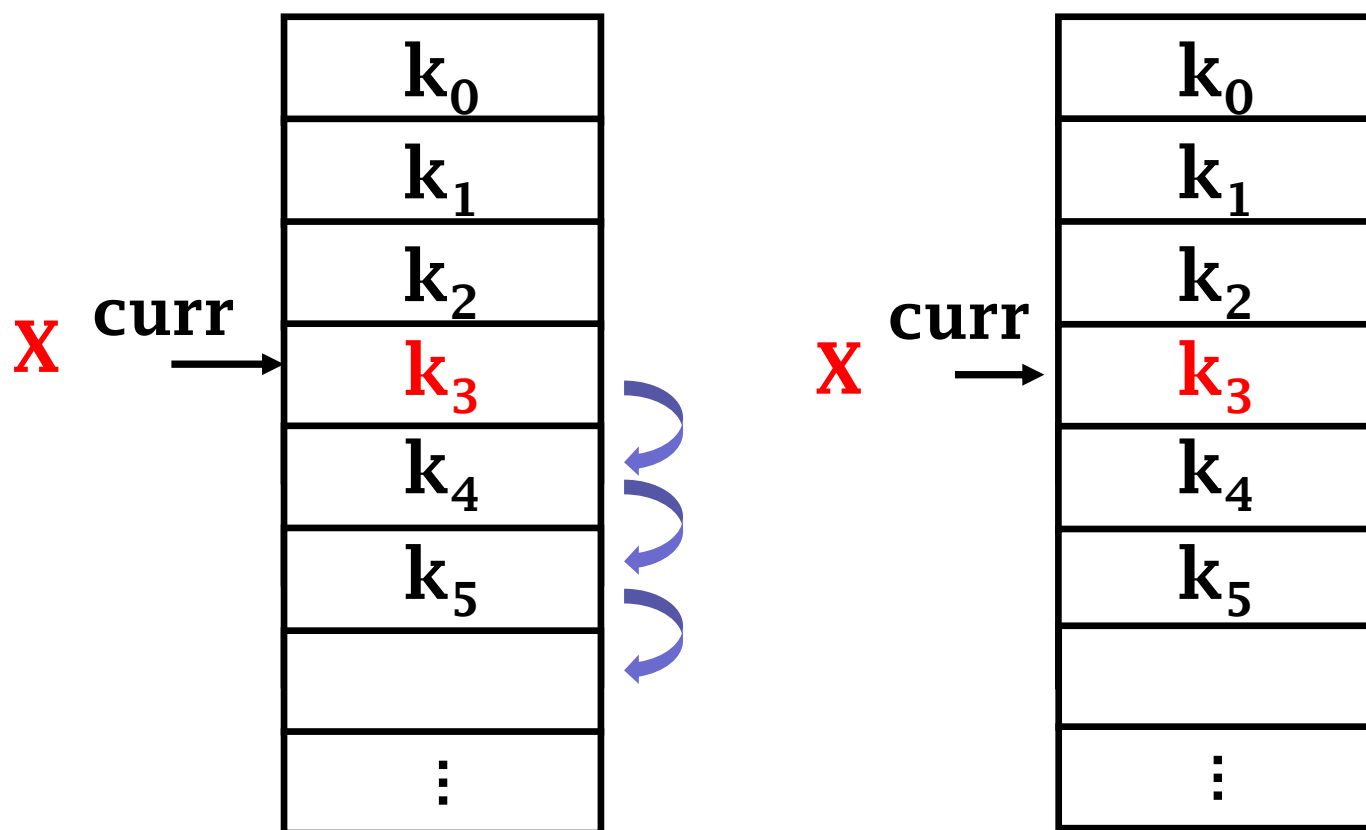
```
void clear() { // 将顺序表存储的内容清除，成为空表
    delete [] aList; curLen = position = 0;
    aList = new T[maxSize];
}
int length(); // 返回当前实际长度
bool append(const T value); // 在表尾添加元素 v
bool insert(const int p, const T value); // 插入元素
bool delete(const int p); // 删除位置 p 上元素
bool setValue(const int p, const T value); // 设元素值
bool getValue(const int p, T& value); // 返回元素
bool getPos(int &p, const T value); // 查找元素
};
```



## 顺序表上的运算

- 重点讨论
  - 插入元素运算
    - `bool insert(const int p, const T value);`
  - 删除元素运算
    - `bool delete(const int p);`
- 其他运算请大家思考.....

## 顺序表的插入图示

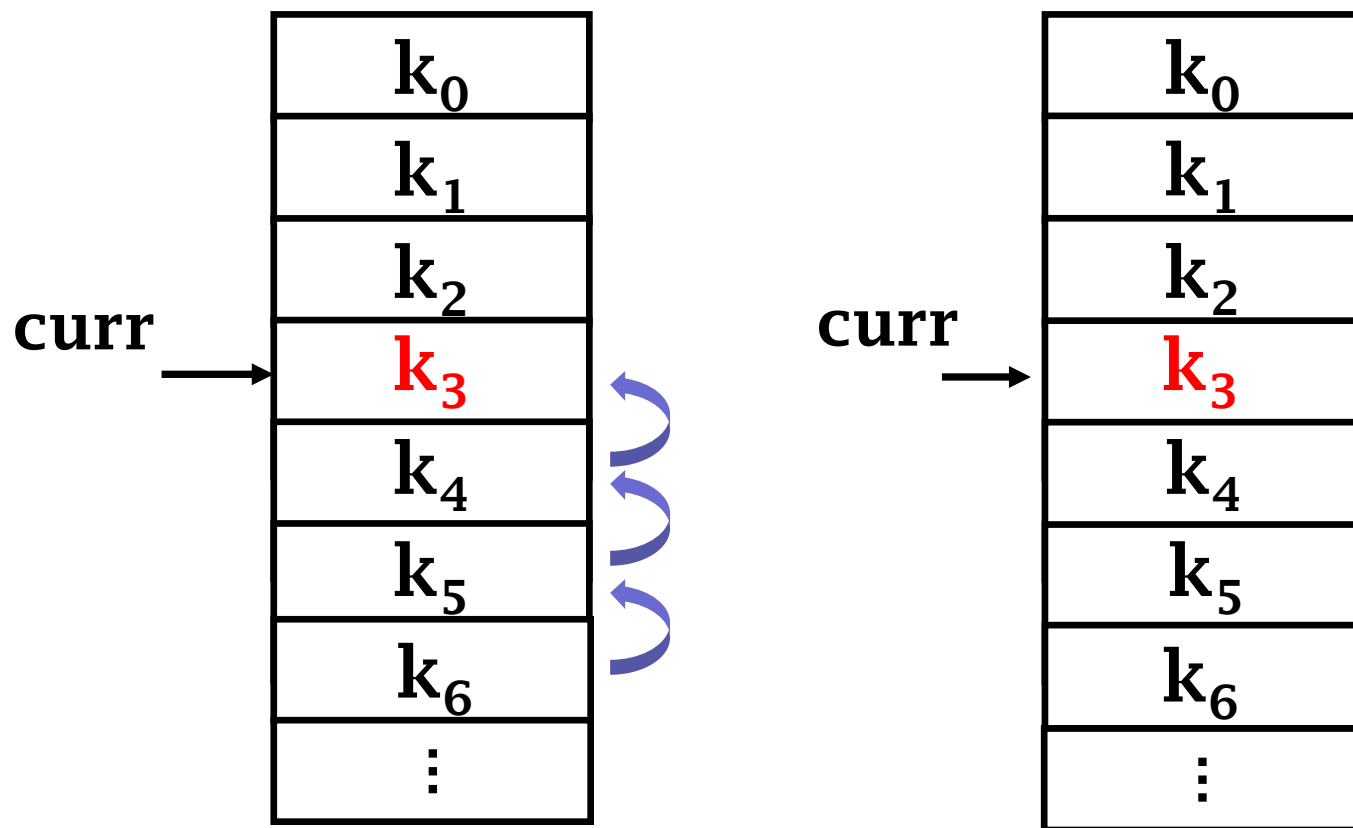




# 顺序表的插入

```
// 设元素的类型为T，aList是存储顺序表的数组，maxSize是其最大长度；  
// p为新元素value的插入位置，插入成功则返回true，否则返回false  
template <class T> bool arrList<T>::insert (const int p, const T value) {  
    int i;  
    if (curLen >= maxSize) { // 检查顺序表是否溢出  
        cout << "The list is overflow" << endl; return false;  
    }  
    if (p < 0 || p > curLen) { // 检查插入位置是否合法  
        cout << "Insertion point is illegal" << endl; return false;  
    }  
    for (i = curLen; i > p; i--)  
        aList[i] = aList[i-1]; // 从表尾 curLen - 1 起往右移动直到 p  
    aList[p] = value; // 位置 p 处插入新元素  
    curLen++; // 表的实际长度增 1  
    return true;  
}
```

# 顺序表的删除图示





# 顺序表的删除

```
// 设元素的类型为 T ; aList是存储顺序表的数组 ; p 为即将删除元素的位置
// 删除成功则返回 true , 否则返回 false
template <class T>          // 顺序表的元素类型为 T
bool arrList<T> :: delete(const int p) {
    int i;
    if (curLen <= 0 ) {      // 检查顺序表是否为空
        cout << " No element to delete \n"<< endl;
        return false ;
    }
    if (p < 0 || p > curLen-1) { // 检查删除位置是否合法
        cout << "deletion is illegal\n"<< endl;
        return false ;
    }
    for (i = p; i < curLen-1; i++)
        aList[i] = aList[i+1]; // 从位置p开始每个元素左移直到 curLen
    curLen--;                  // 表的实际长度减1
    return true;
}
```



## 顺序表插入删除运算的算法分析

- 表中元素的移动
  - 插入：移动  $n - i$
  - 删除：移动  $n - i - 1$  个
- $i$  的位置上插入和删除的概率分别是  $p_i$  和  $p_i'$ 
  - 插入的平均移动次数为

$$M_i = \sum_{i=0}^n (n - i) p_i$$

- 删除的平均移动次数为

$$M_d = \sum_{i=0}^{n-1} (n - i - 1) p_i'$$



## 算法分析

- 如果在顺序表中每个位置上插入和删除元素的概率相同，即  $p_i = \frac{1}{n+1}$ ,  $p'_i = \frac{1}{n}$

$$\begin{aligned} M_i &= \frac{1}{n+1} \sum_{i=0}^n (n-i) = \frac{1}{n+1} \left( \sum_{i=0}^n n - \sum_{i=0}^n i \right) \\ &= \frac{n(n+1)}{n+1} - \frac{n(n+1)}{2(n+1)} = \frac{n}{2} \end{aligned}$$

$$\begin{aligned} M_d &= \frac{1}{n} \sum_{i=0}^n (n-i-1) = \frac{1}{n} \left( \sum_{i=0}^n n - \sum_{i=0}^n i - n \right) \\ &= \frac{n^2}{n} - \frac{(n-1)}{2} - 1 = \frac{n-1}{2} \end{aligned}$$

时间代价  
为 $O(n)$

## 思考

- 顺序表中，插入删除操作需要考虑哪些问题？
- 顺序表有哪些优缺点？



## 第二章 线性表

- 2.1 线性表
- 2.2 顺序表
- 2.3 链表
- 2.4 顺序表和链表的比较



## 2.4 线性表实现方法的比较

- 顺序表的主要优点
  - 没有使用指针，不用花费额外开销
  - 线性表元素的读访问非常简洁便利
- 链表的主要优点
  - 无需事先了解线性表的长度
  - 允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况
- 总结
  - 顺序表是存储静态数据的不二选择
  - 链表是存储动态变化数据的良方



## 顺序表和链表的比较

- 顺序表

- 插入、删除运算时间代价  $O(n)$ ，查找则可常数时间完成
- 预先申请固定长度的连续空间
- 如果整个数组元素很满，则没有结构性存储开销

- 链表

- 插入、删除运算时间代价  $O(1)$ ，但找第 $i$ 个元素运算时间代价  $O(n)$
- 存储利用指针，动态地按照需要为表中新的元素分配存储空间
- 每个元素都有结构性存储开销



## 顺序表和链表存储密度

$n$  表示线性表中当前元素的数目，

$P$  表示指针的存储单元大小（通常为 4 bytes）

$E$  表示数据元素的存储单元大小

$D$  表示可以在数组中存储的线性表元素的最大数目

- 空间需求
  - 顺序表的空间需求为  $DE$
  - 链表的空间需求为  $n(P + E)$
- $n$  的临界值，即  $n > DE / (P+E)$ 
  - $n$  越大，顺序表的空间效率就更高
  - 如果  $P = E$ ，则临界值为  $n = D / 2$





## 应用场合的选择

- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
  - 线性表的最大长度也是一个重要因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择



## 顺序表和链表的选择

- 顺序表
  - 结点总数目大概可以估计
  - 线性表中结点比较稳定（插入删除少）
  - $n > DE / (P + E)$
- 链表
  - 结点数目无法预知
  - 线性表中结点动态变化（插入删除多）
  - $n < DE / (P + E)$

## 思考

- 顺序表和链表的选择？
  - 结点变化的动态性
  - 存储密度

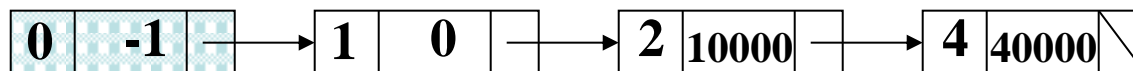
## 思考：一元多项式的表达

- 一元多项式： $P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_n x^n$
- 线性表表示： $P = (p_0, p_1, p_2, \dots, p_n)$
- 顺序表表示：只存系数（第  $i$  个元素存  $x^i$  的系数）



数据稀疏的情况： $p(x) = 1 + 2x^{10000} + 4x^{40000}$

- 链表表示： 结点结构





# 数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”

<http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/>

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。“十一五”国家级规划教材