数据结构与算法

第2章 线性表

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

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

内容提要

- 线性结构
- 顺序表
- 链表
- 线性表实现方法的比较

线性结构

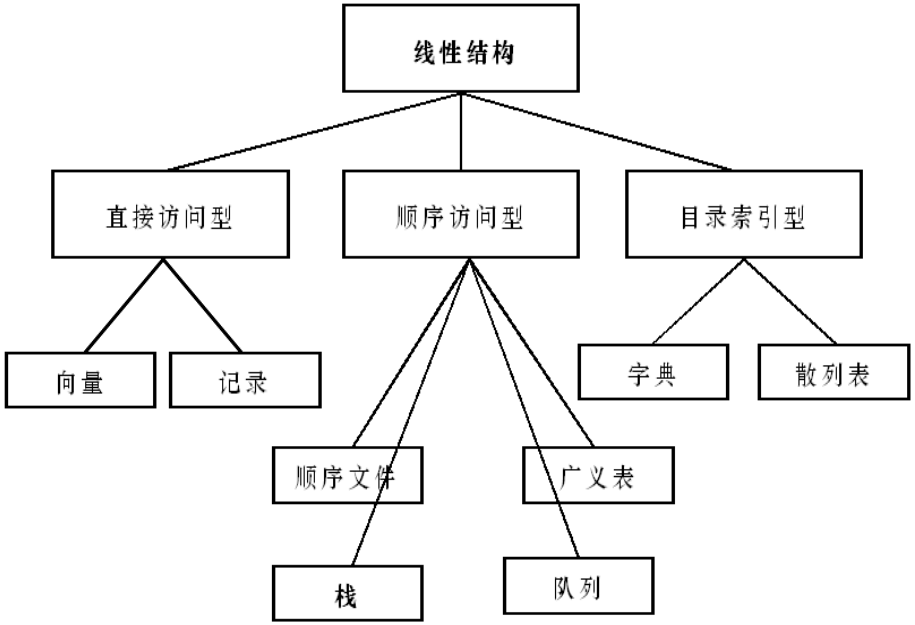
- 二元组B = (K, R) , K = {a_0, a_1, ⋯, a_{n-1}}，R = {r}
  - K中有唯一的开始结点，没有前驱但有一个唯一的后继
  - K存在唯一的终止结点，有一个唯一的前驱而无后继
  - 其它的结点皆称为内部结点，每一个内部结点都有且仅有一个唯一的前驱，也有一个唯一的后继

    a_0, a_1, ⋯, a_{n-1}

    $< a_i, a_{i+1}>$ a_i是a_{i+1}的前驱， a_{i+1}是a_i的后继
- 元素间具有线性关系
  - “1-to-1”的关系，所有元素排成一个线性序列
  - 也称前驱/后继关系，具有 反对称性 和 传递性

线性结构分类

- 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)



线性结构分类

- 按操作划分
  - 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

线性结构

- 简单线性结构
  - 特点
    - **均匀性**：同一线性结构中各数据元素具有相同的数据类型和长度
    - **有序性**：各数据元素在线性结构各有其位，且数据元素间的相对位置是线性的
  - e.g., 线性表、栈、队列、散列表
- 高级线性结构
  - e.g., 广义表、多维数组、文件
- 特点
  - 操作灵活，长度可增长或缩短

线性表（Linear list）

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

线性表的逻辑结构

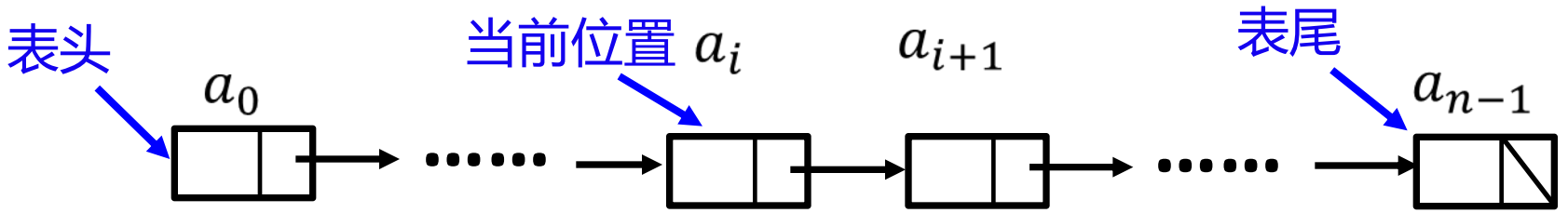
- 由称为元素的数据项组成的一种有限且有序的序列，元素也称为结点或表目
  - 表示方式
    - 元素间由逗号隔开， 并由括号括起

      $(a_0, a_1, \cdots, a_n)$
    - 字符串等的表示不必明确写出逗号和括号

      Example string
  - 子表
    - 线性表 $(a_0, a_1, \cdots, a_n)$， 对任意的 i 和 j，若有 $0 \leqslant i \leqslant j \leqslant n$，则 $(a_i, a_{i+1}, \cdots, a_j)$ 称为前者的子表

线性表逻辑结构

- 主要属性
  - 长度：包含的结点个数
    - 长度为 0 的线性表称为空表
  - 表头(head)
  - 表尾 (tail)
  - 当前位置(current position)



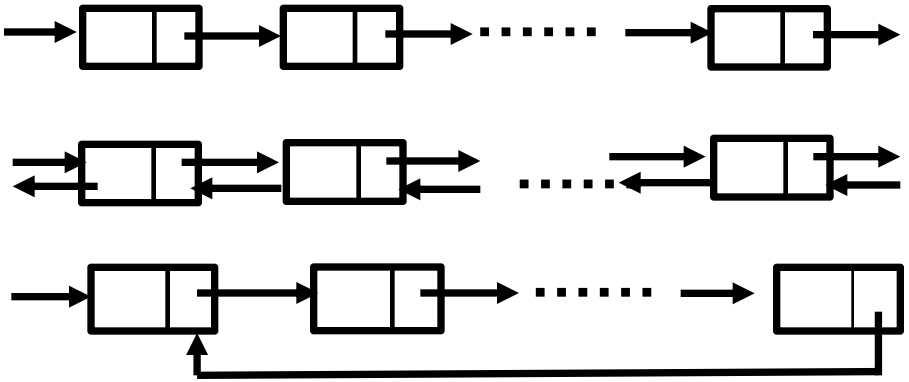
线性表存储结构

- **定长**的一维数组结构
  - 又称 向量型的顺序存储结构
- **变长**的存储结构
  - 链接式存储结构
  - 串结构、动态数组、顺序文件

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



线性表的运算

- **表整体相关**
  - 建立线性表
  - 清除线性表
  - 辅助性管理操作
  - 排序
  - 检索
- **表中元素相关**
  - 插入一个新元素
  - 删除某个元素
  - 修改某个元素

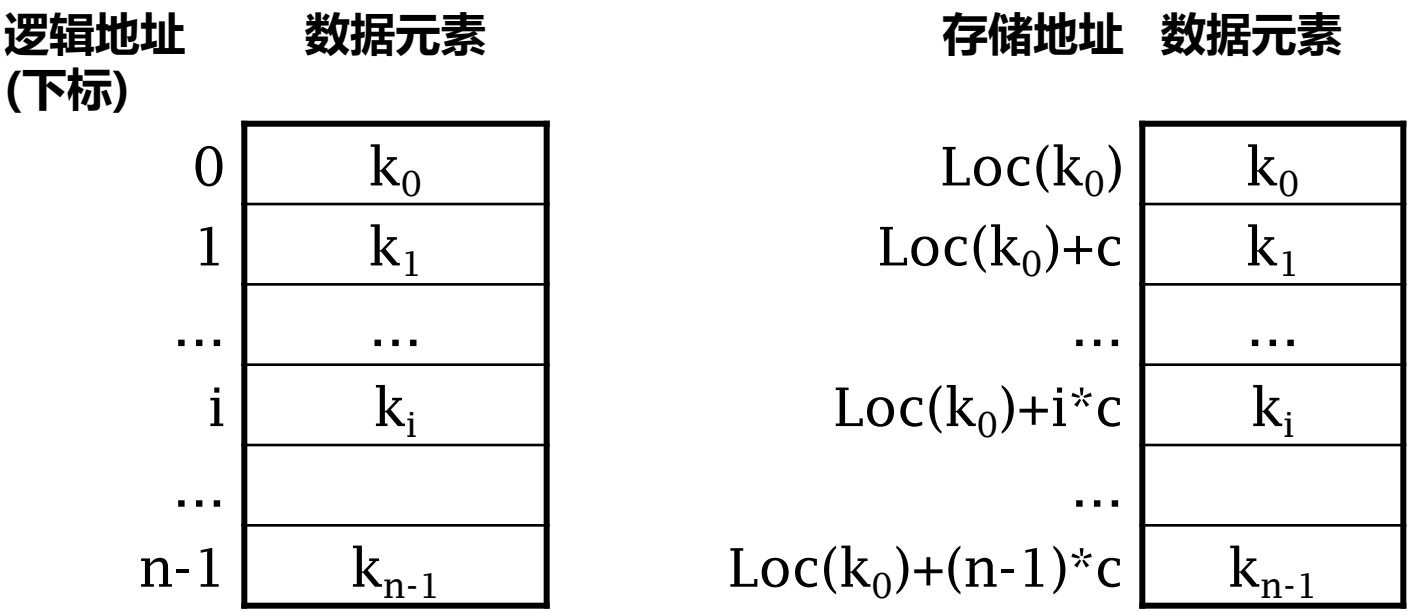
线性表类模板

```
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
};
```

顺序表（Array-based list）

- 定长的一维数组，也称向量
  - 元素顺序存储在一块连续存储空间中，每个元素有唯一的索引值，可随机访问

$$\mathrm{Loc}(k_i) = Loc(k_0) + c \times i, c = sizeof(ELEM)$$



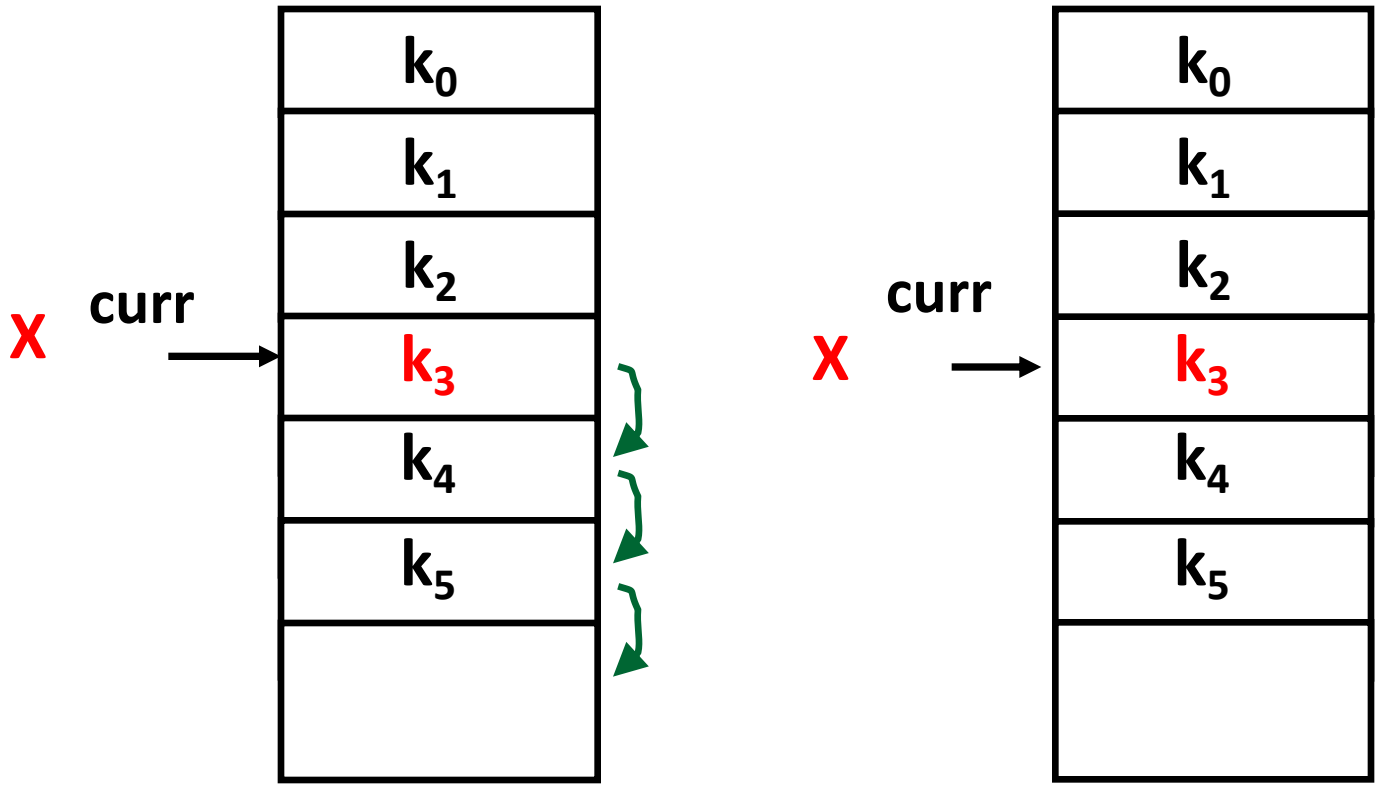
顺序表类定义

```
class arrList : public List<T> {  // 顺序表，向量
private:                          // 线性表的取值类型和取值空间
   T * aList ;                    // 私有变量，存储顺序表的实例
   int maxSize;                   // 私有变量，顺序表实例的最大长度
   int curLen;                    // 私有变量，顺序表实例的当前长度
   int position;                  // 私有变量，当前处理位置
public:
   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
};
```

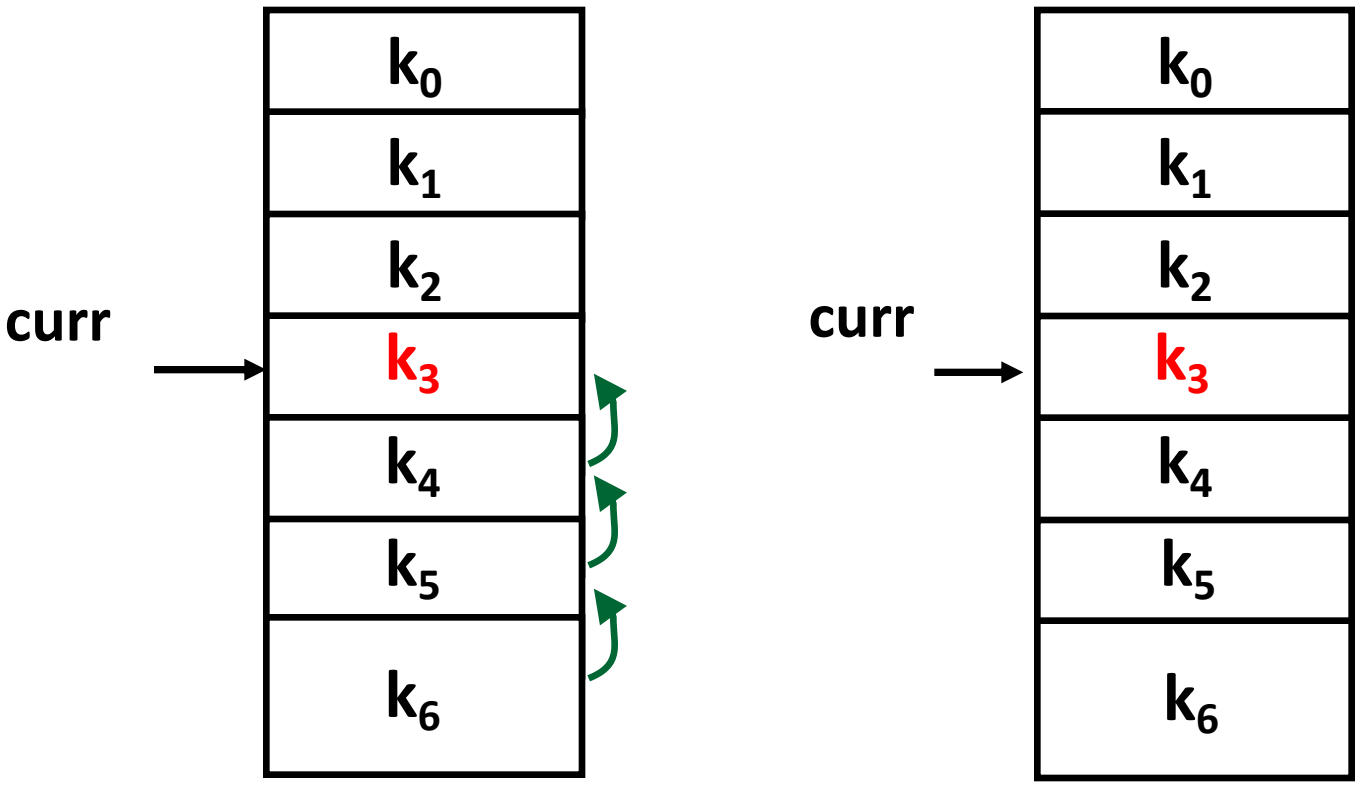
顺序表上的运算

- 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

顺序表插入图示



顺序表删除图示



顺序表代价分析

- 插入和删除的主要代价在于元素的移动，*i* 位置上
  - 插入：*n-i*
  - 删除：*n-i-1*
- 若 *i* 位置上插入和删除的概率分别为 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

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

顺序表代价分析

- 若在顺序表每个位置上插入和删除的概率相同，即，

$$p_i = \frac{1}{n+1} \qquad p'_i = \frac{1}{n}$$

$$M_i = \sum_{i=0}^{n} (n-i)p_i = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}\left(\sum_{i=0}^{n} n - \sum_{i=0}^{n} i\right)$$

$$= \frac{n(n+1)}{n+1} - \frac{n(n+1)}{2(n+1)} = \frac{n}{2}$$

$$M_d = \sum_{i=0}^{n-1} (n-i-1)p'_i = \frac{1}{n}\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{n}\left(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - n\right)$$

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

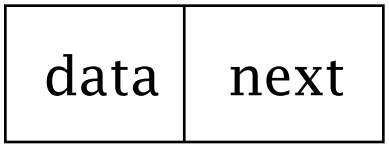
时间代价为 *O(n)*

顺序表的优缺点

- 优点
  - 不需要附加空间
  - 随机存取任一元素（根据下标）
- 缺点
  - 伊始即需分配足够大的一片连续的内存空间，难以预估所需空间大小
  - 更新操作代价大

链表（Linked List）

- 通过指针将一串存储结点链接结点
  - 逻辑上相邻的元素在物理位置上不要求相邻
  - 按需动态地为新元素分配存储空间
- 存储结点由两部分组成：数据字段 + 指针字段

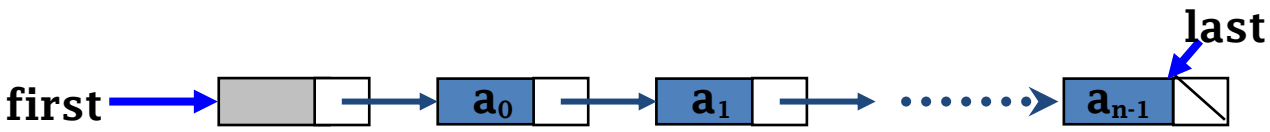


“programming the logic theory machine” By Newell and Shaw in Proc. WJCC,1957,

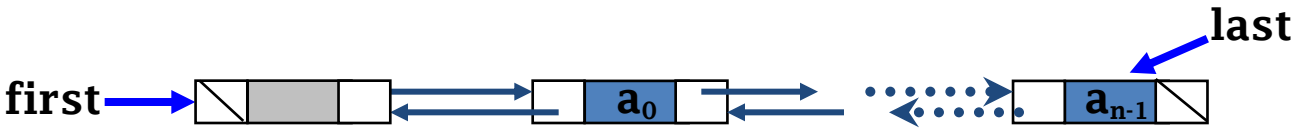
Newell and Simon were recognized with ACM Turing Award in 1975 for having “made basic contributions to artificial intelligence, the psychology of human cognition, and list processing”

链表的分类

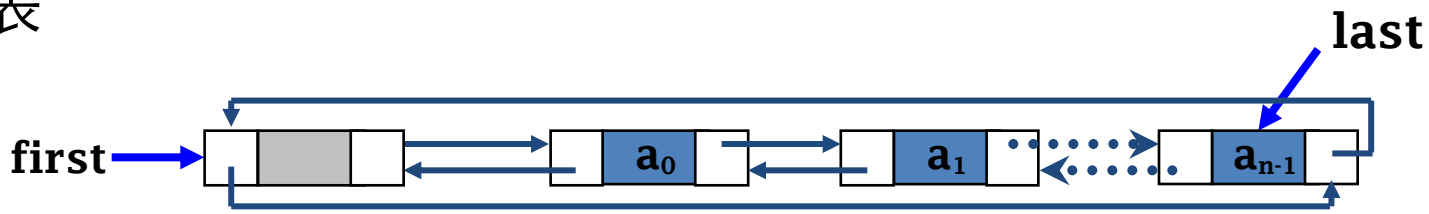
- **根据链接方式和指针多寡**
  - 单链表



  - 双链表



  - 循环链表

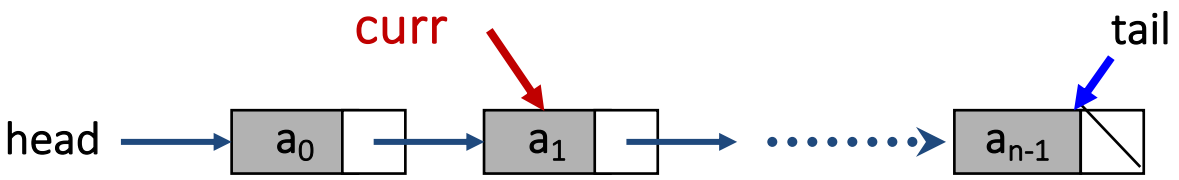


链表的运算

- 检索
  - 在链表中查找满足某种条件的元素
  - 从表头开始循链进行，不能随机访问
- 插入
  - 在链表的适当位置插入一个元素
- 删除
  - 从链表中删除一个指定元素

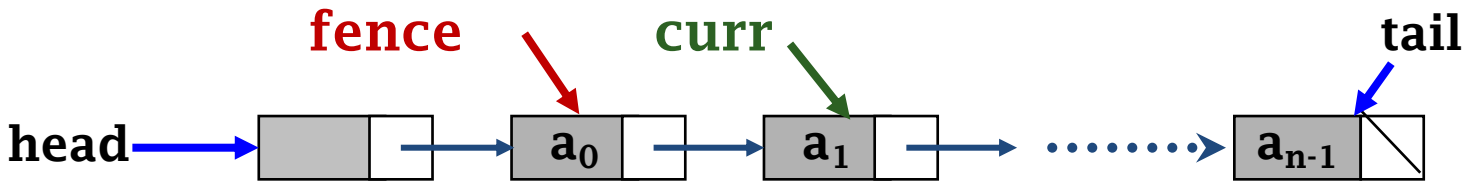
单链表

- 简单单链表
  - 整个单链表： head
  - 第一个结点： head
  - 空表判断： head == NULL
  - 当前结点 a1： curr

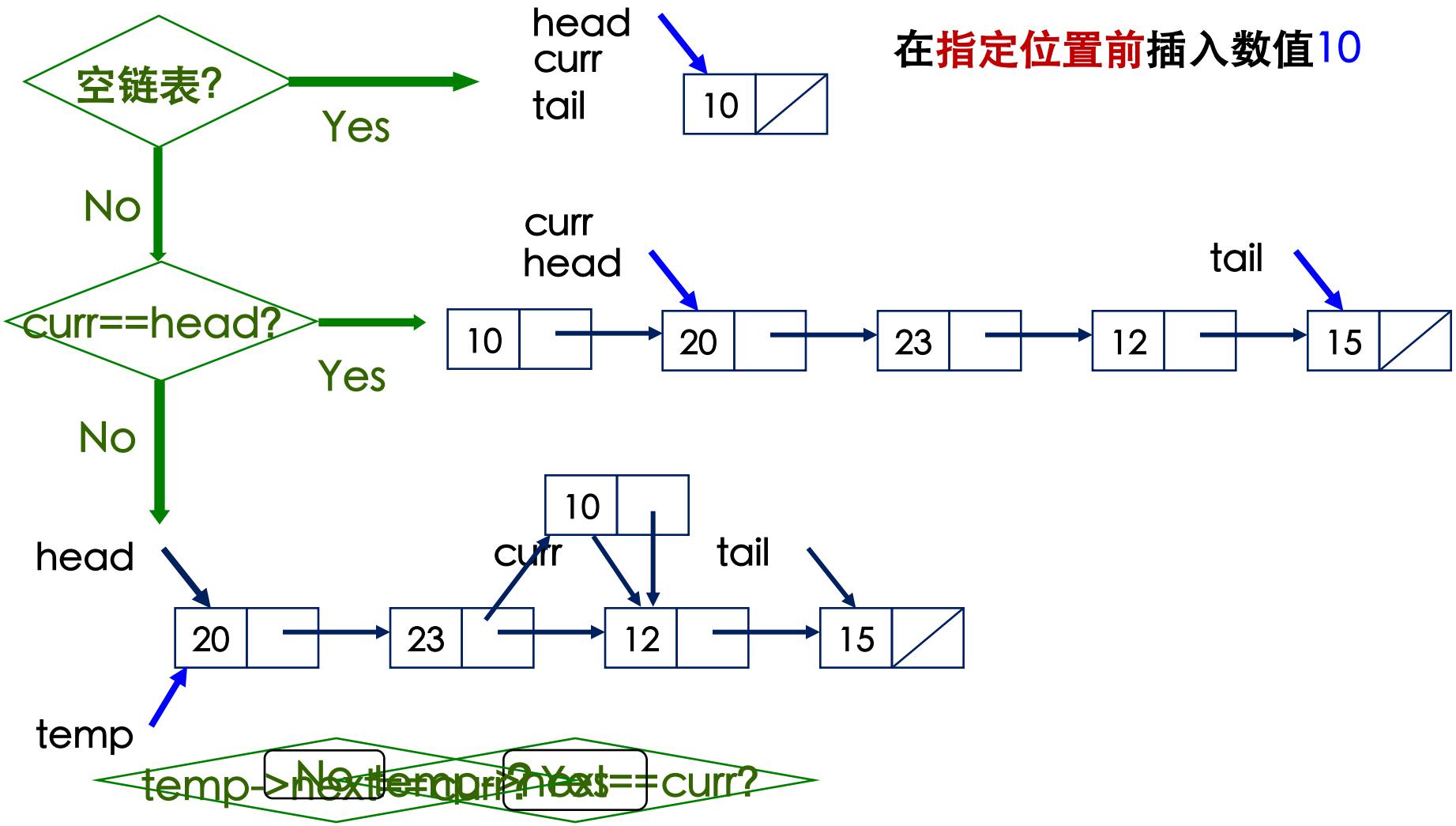


单链表

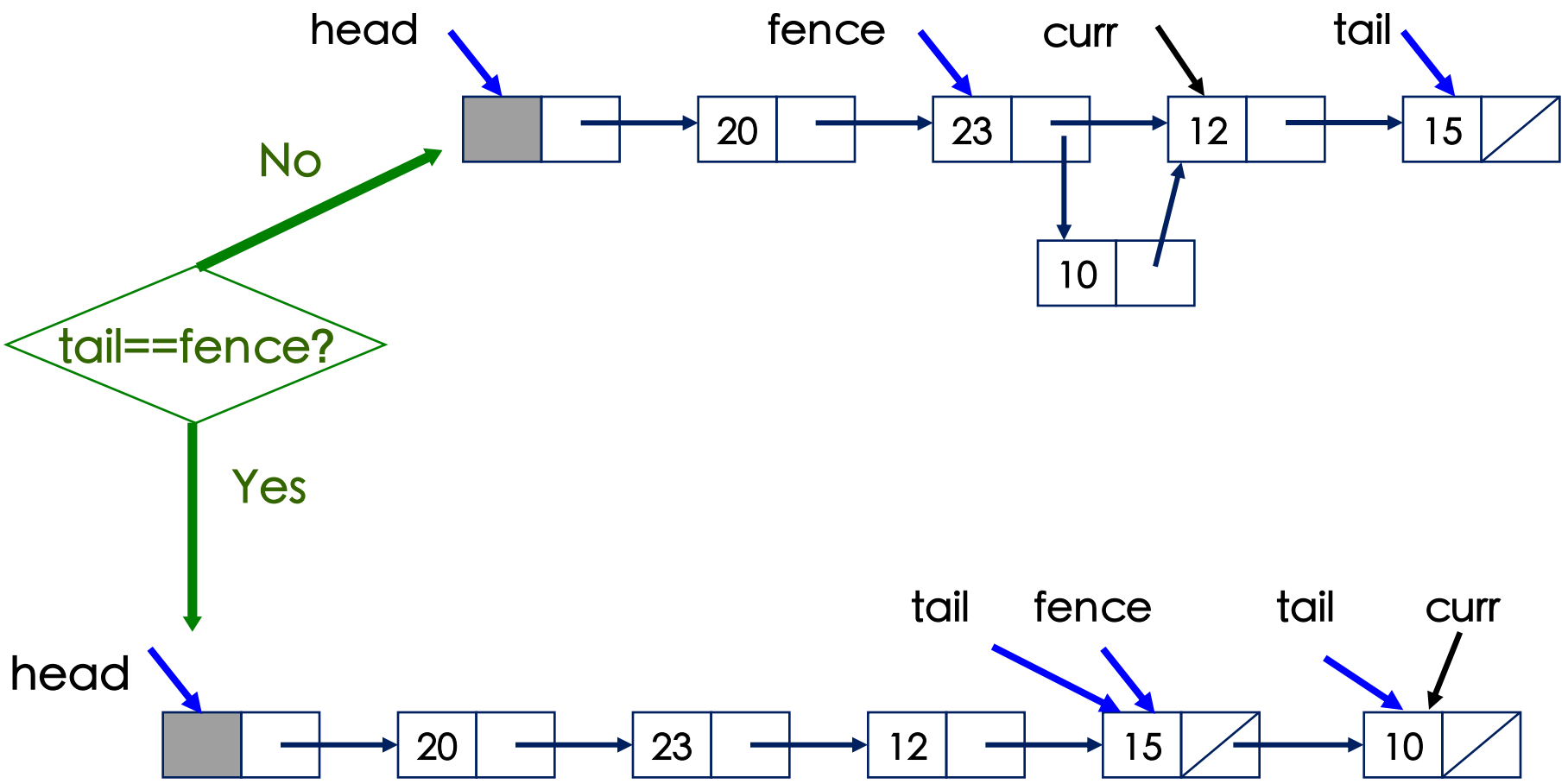
- 带头结点单链表
  - 整个单链表： head
  - 第一个结点： head->next，head ≠ NULL
  - 空表判断： head->next == NULL
  - 当前结点 a1：fence->next (curr 隐含)



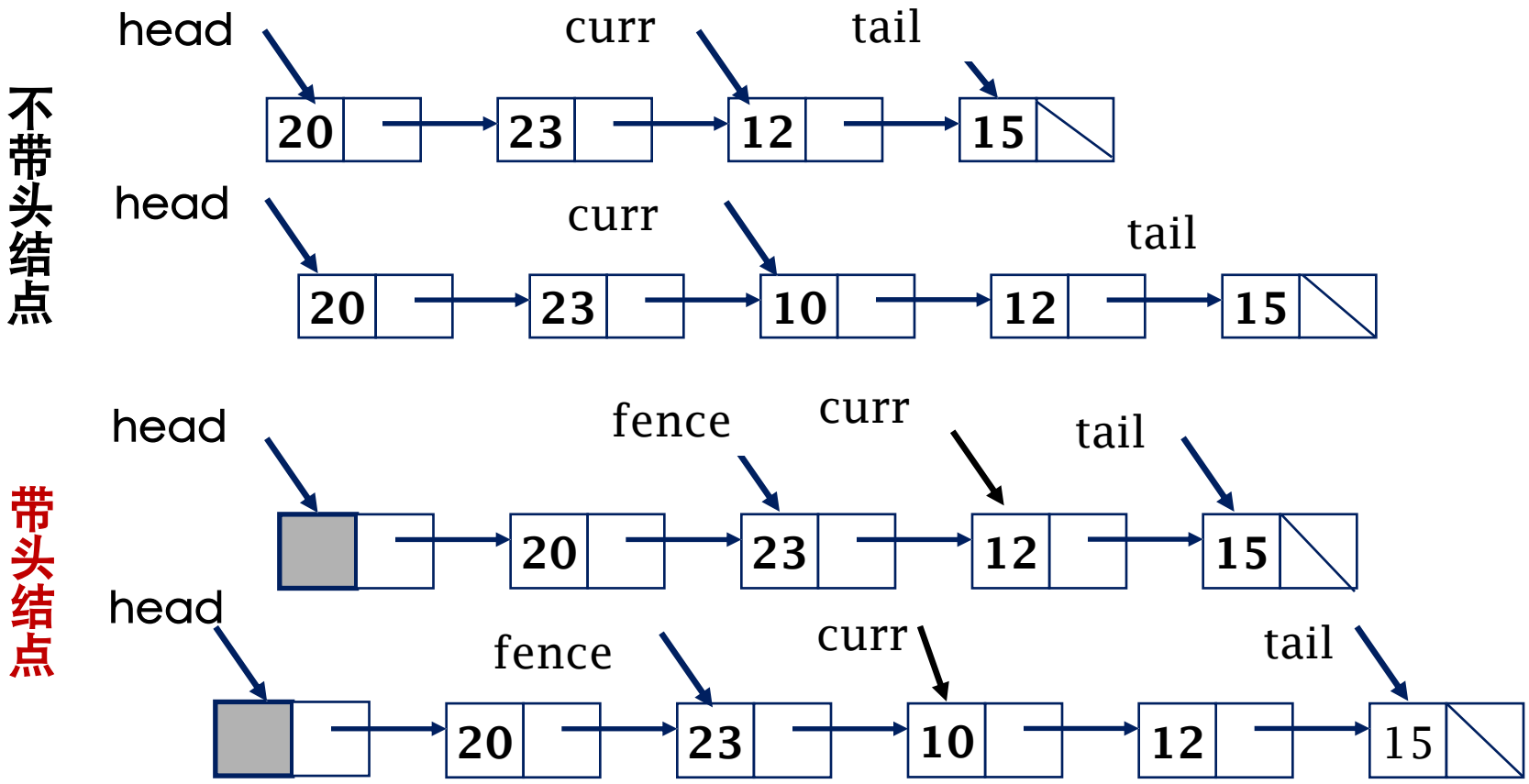
不带头结点的链表插入示意



带头结点的链表插入示意

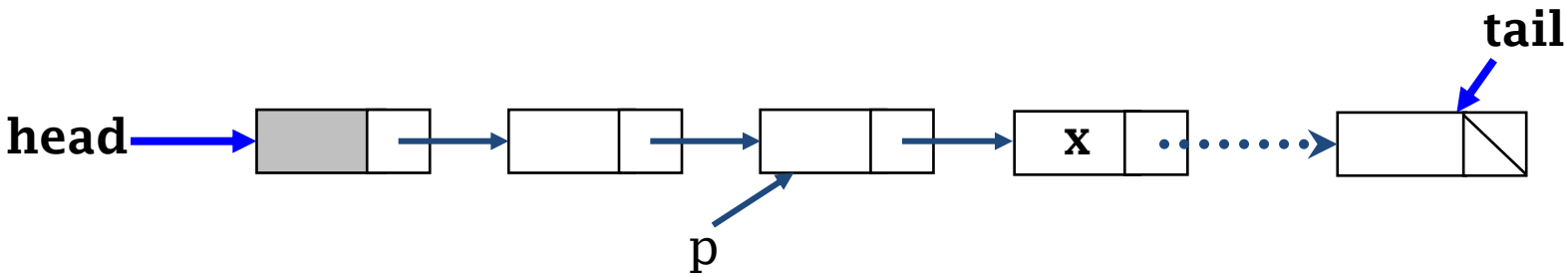


链表：是否带头结点


不带头结点
head
curr
tail
20
23
12
15
head
curr
tail
20
23
10
12
15
带头结点
head
fence
curr
tail
20
23
12
15
head
fence
curr
tail
20
23
10
12
15

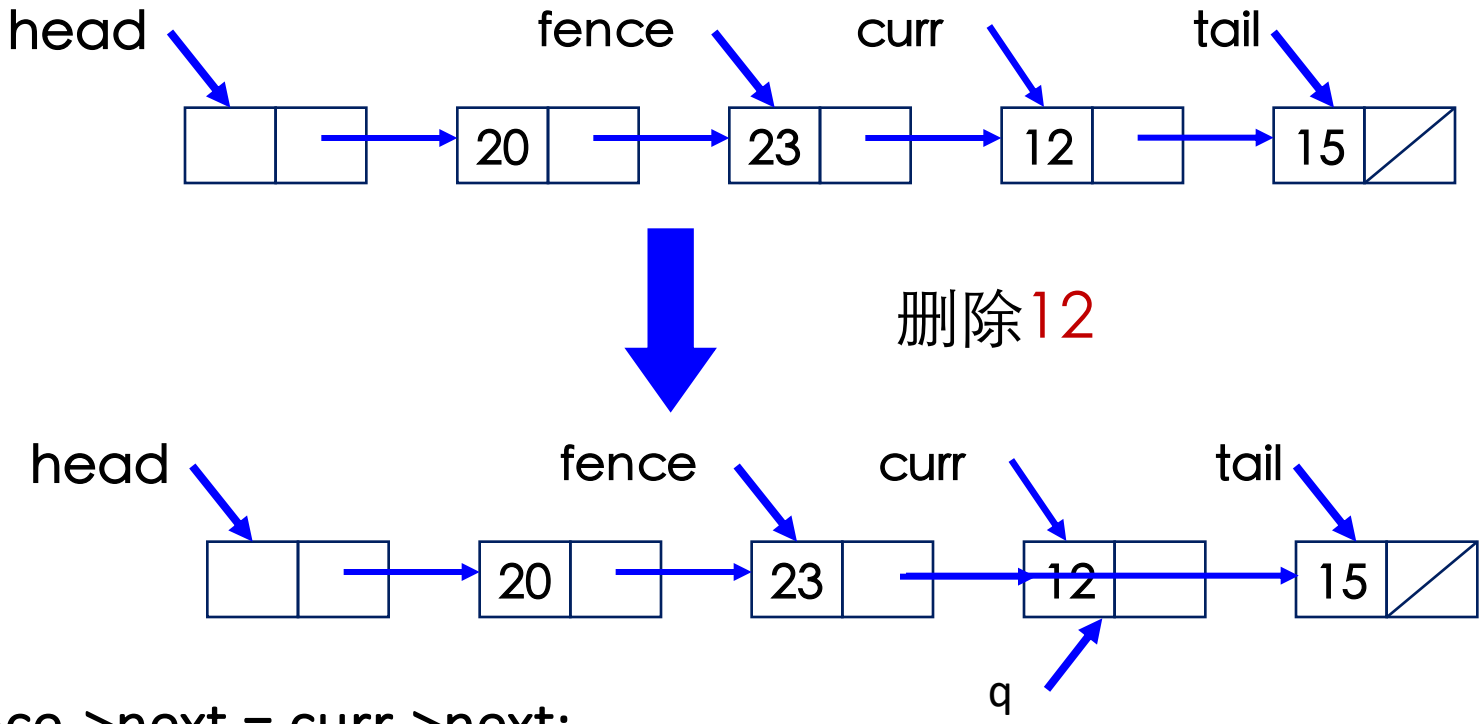
单链表的删除

- 从链表中删除结点 x
  - 用p指向元素 x 的结点的前驱结点
  - 删除元素为 x 的结点
- 用p 指向元素 x 的结点的前驱结点




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

单链表的删除示例




```
fence->next = curr->next;

q = curr;  curr = curr ->next;

free(q);
```

单链表运算的代价分析

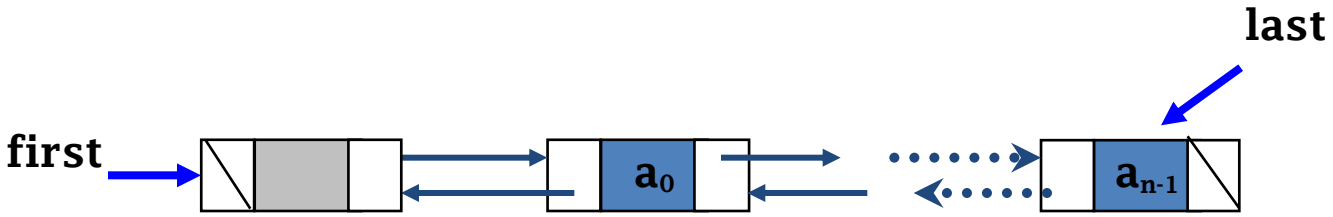
1. 对一个结点操作，必须先找到它，即须有一个指向它的指针
2. 找单链表中任一结点，都须自首结点始

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

- 时间复杂度 O(n)
  - 定位：O(n)
  - 插入：O(n) + O(1)
  - 删除：O(n) + O(1)

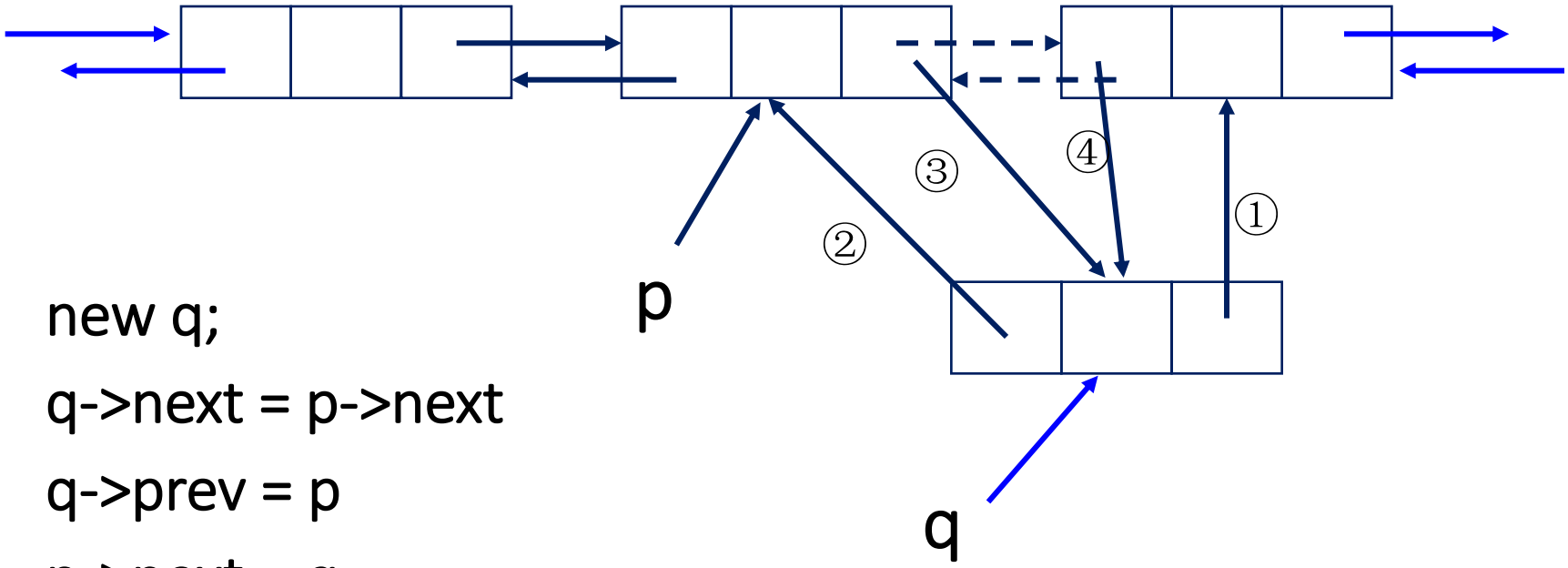
双链表(Double linked list)

- 为弥补单链表的不足，而采用双链表
  - 单链表的 next 字段指向后继结点，不能高效找到前驱，反之亦然
  - 增加一个指向前驱的指针，提高结点的存访能力



双链表插入图示

在p所指结点后面插入一个新结点



new q;

q->next = p->next

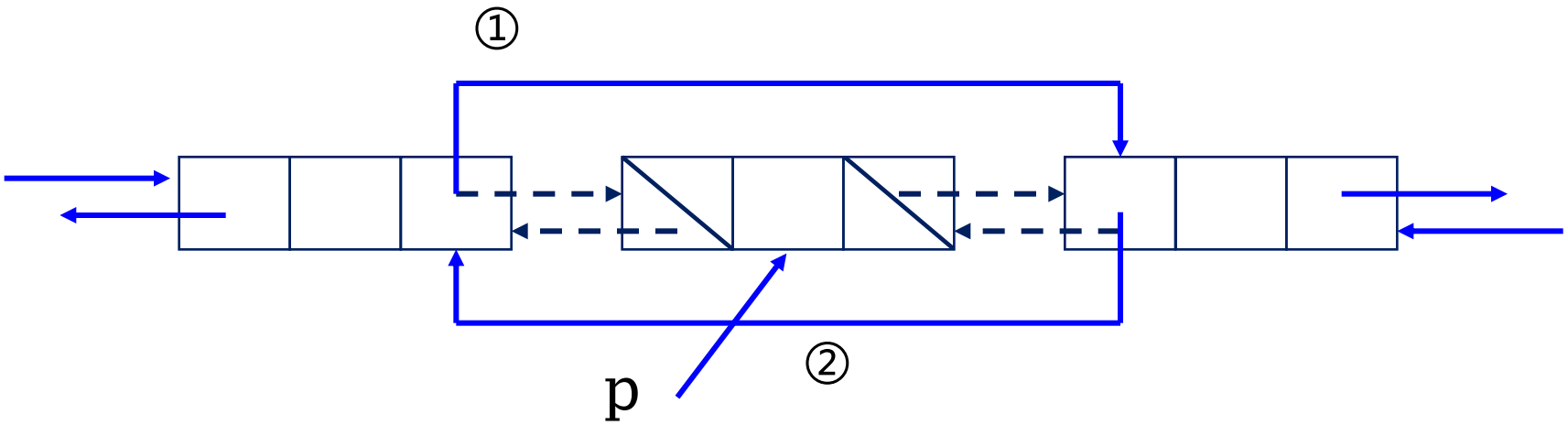
q->prev = p

p->next = q

q->next->prev = q

双链表删除图示

- 删除p所指的结点
  - 若马上删除 p，则可以不赋空



p->prev->next=p->next

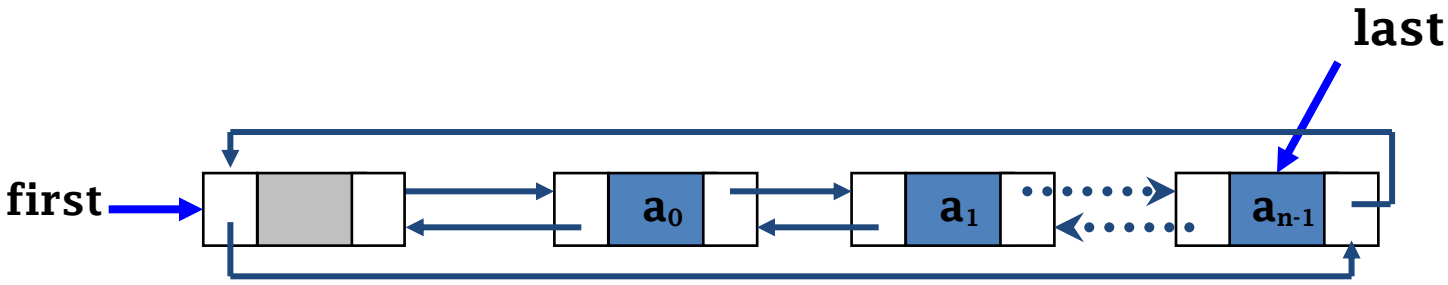
p->next->prev=p->prev

p->next=NULL

p->prev=NULL

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

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

- 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4bytes）

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

思考题

- 若没有指针支持，如何实现诸如链表的易于增删功能?

- 带表头与不带表头的单链表
- 顺序表和链表的选择
  - 结点变化的动态性
  - 存储密度