# 第二章 线性表

张铭 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008.6 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

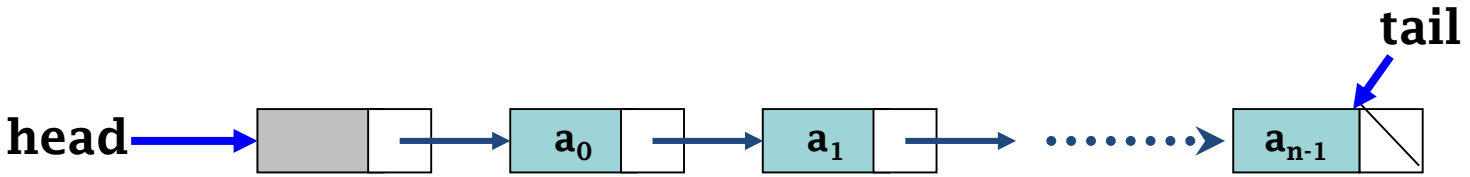
# 第二章 线性表

- 线性结构
- 顺序表、链表
- 线性表实现方法的比较
- 线性表的应用

# 第二章 线性表

- 2.1 线性表 $\{a_0, a_1, \ldots, a_{n-1}\}$
- 2.2 顺序表


- 2.3 链表

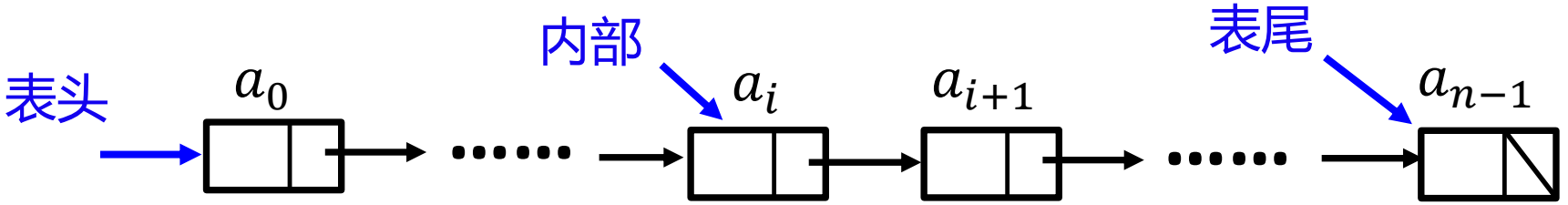



- 2.4 顺序表和链表的比较

# 线性结构

- 二元组$B=(K,R)$ $K=\{a_0,a_1,\ldots,a_{n-1}\}$ $R=\{r\}$
  - 有一个唯一的**开始结点**，它没有前驱，有一个唯一的后继
  - 一个唯一的 **终止结点**，它有一个唯一的前驱而没有后继
  - 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前驱，也有一个唯一的后继

    $<a_i,a_{i+1}>$ $a_i$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**



# 线性表中的相关概念

- 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$k_0$，$k_1$，…，$k_{n-1}$（$n \geq 1$）
  - 表目：线性表中的元素（可包含多个数据项，记录）
  - 文件：含有大量记录的线性表又称为文件
  - 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)

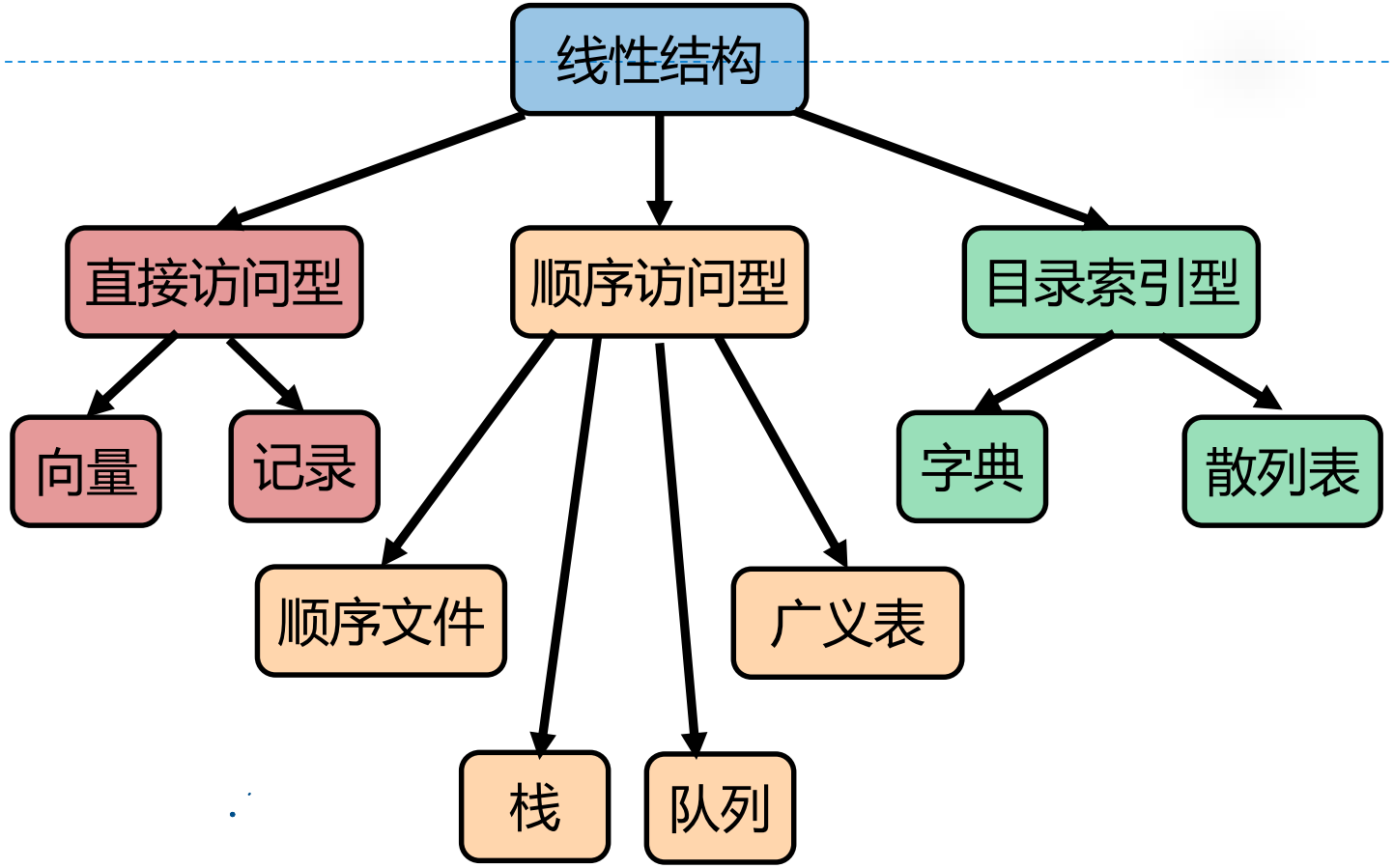
## 2.1 线性表

# 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)



# 标准模板库 STL

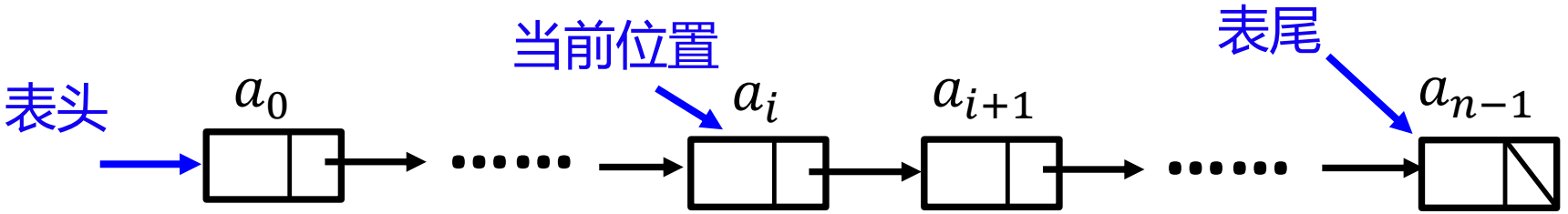
- STL是一个C++预设的容器类的模板和算法库。
- 它提供了主要的几种数据结构的容器类模板，例如vector、stack、queue、map等，使用起来非常方便。
- 当然，它也提供诸如排序和搜索这样的基本算法。

# 线性表的三个方面

- 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**
  - **表头** (head)
  - **表尾** (tail)
  - **当前位置** (current position)

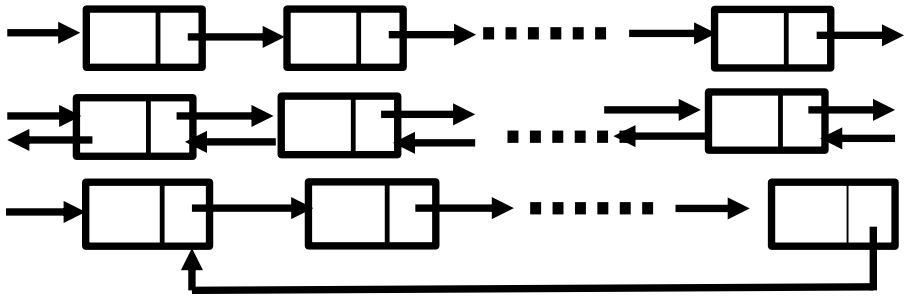


# 线性表的存储结构

$$\text{存储密度} = \frac{\text{数据本身所占存储}}{\text{整个数据结构所占存储}}$$

- **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

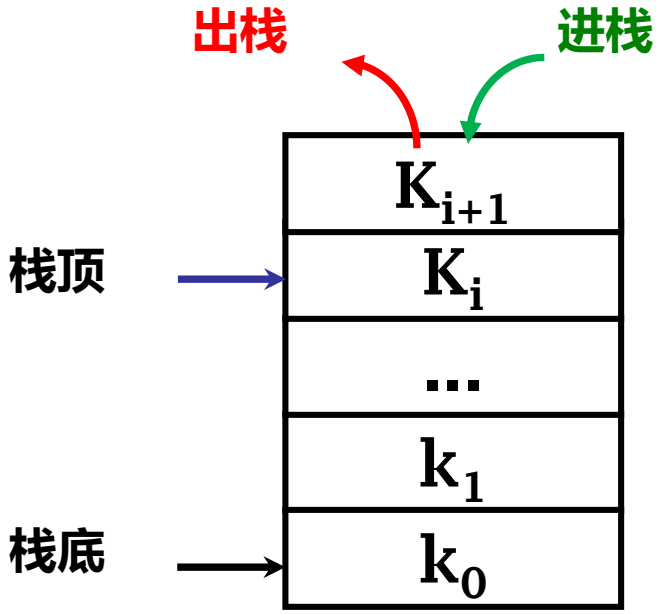


# 线性表分类（按操作）

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

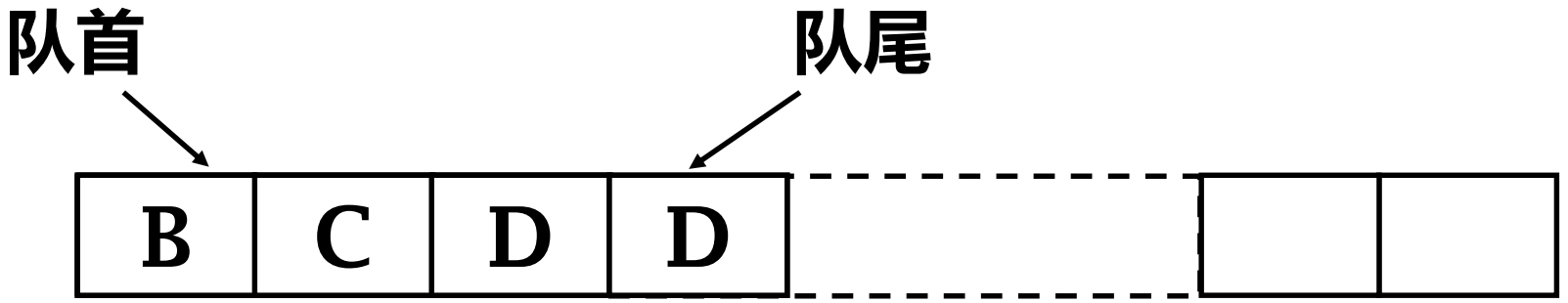
# 线性表分类（按操作）

- 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



# 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法**
  - 各种分类中的数据结构的区别和联系
- **顺序表、链表、栈、队列等线性表**
  - 哪一些是跟存储结构相关?
  - 哪一些是跟运算相关的?

# 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。固定长度的一维数组

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

| 逻辑地址(下标) | 数据元素 |
|---|---|
| 0 | $k_0$ |
| 1 | $k_1$ |
| … | … |
| i | $k_i$ |
| … | |
| n-1 | $k_{n-1}$ |

| 存储地址 | 数据元素 |
|---|---|
| $Loc(k_0)$ | $k_0$ |
| $Loc(k_0)+c$ | $k_1$ |
| … | … |
| $Loc(k_0)+i*c$ | $k_i$ |
| … | |
| $Loc(k_0)+(n-1)*c$ | $k_{n-1}$ |

# 顺序表结点定义

```
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:
    arrList(const int size) {         // 创建新表，设置表实例的最大长度
        maxSize = size;  aList = new T[maxSize];
            curLen = position = 0;
    }
    ~arrList() {                      // 析构函数，用于消除该表实例
            delete [] aList;
    }
```

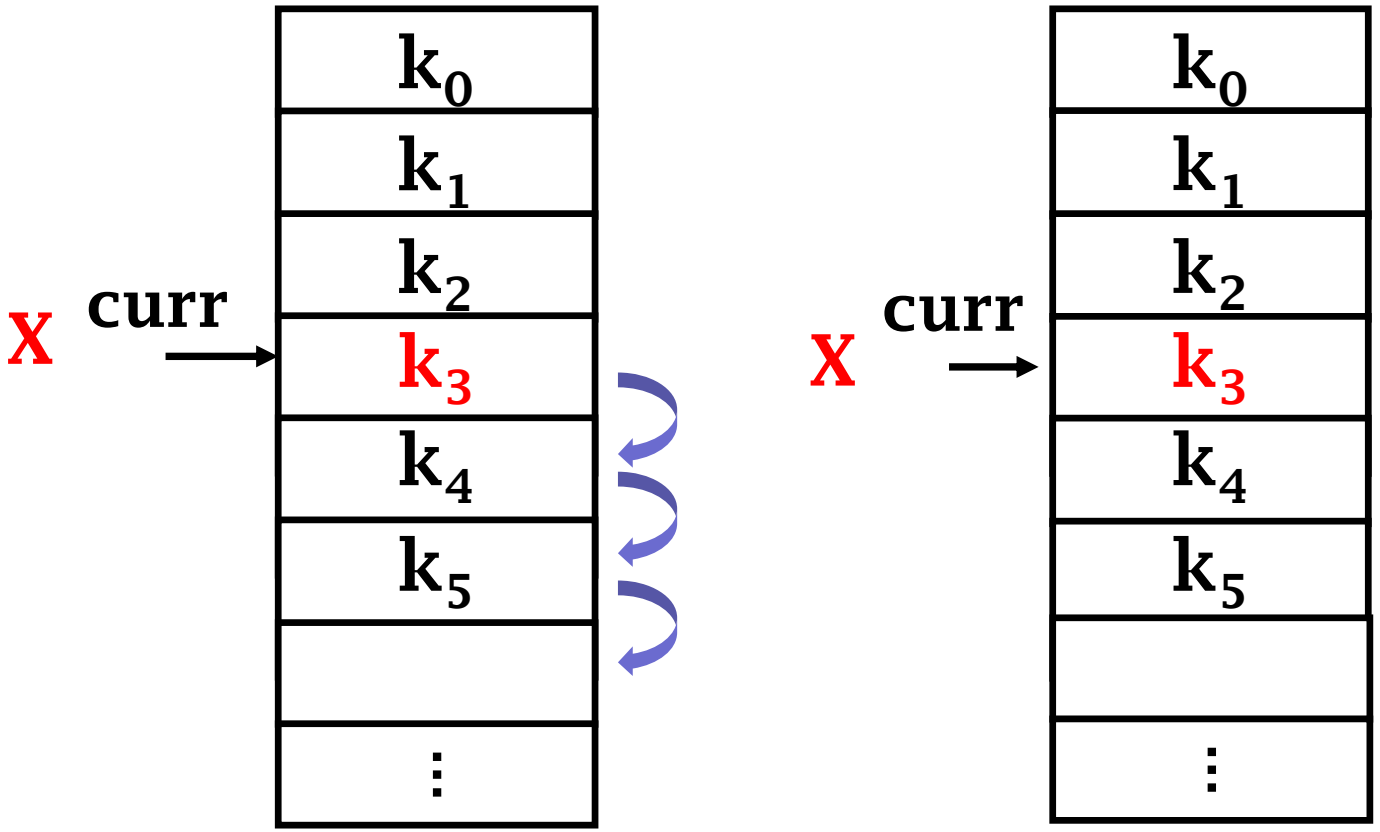
# 顺序表类定义

```
    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);
- 其他运算请大家查阅教材配套代码包
  - http://media.openjudge.cn/upload/DSMooc/DSCode_ZhangWangZhao2008_06.rar
  - http://jpk.pku.edu.cn/course/sjjg/

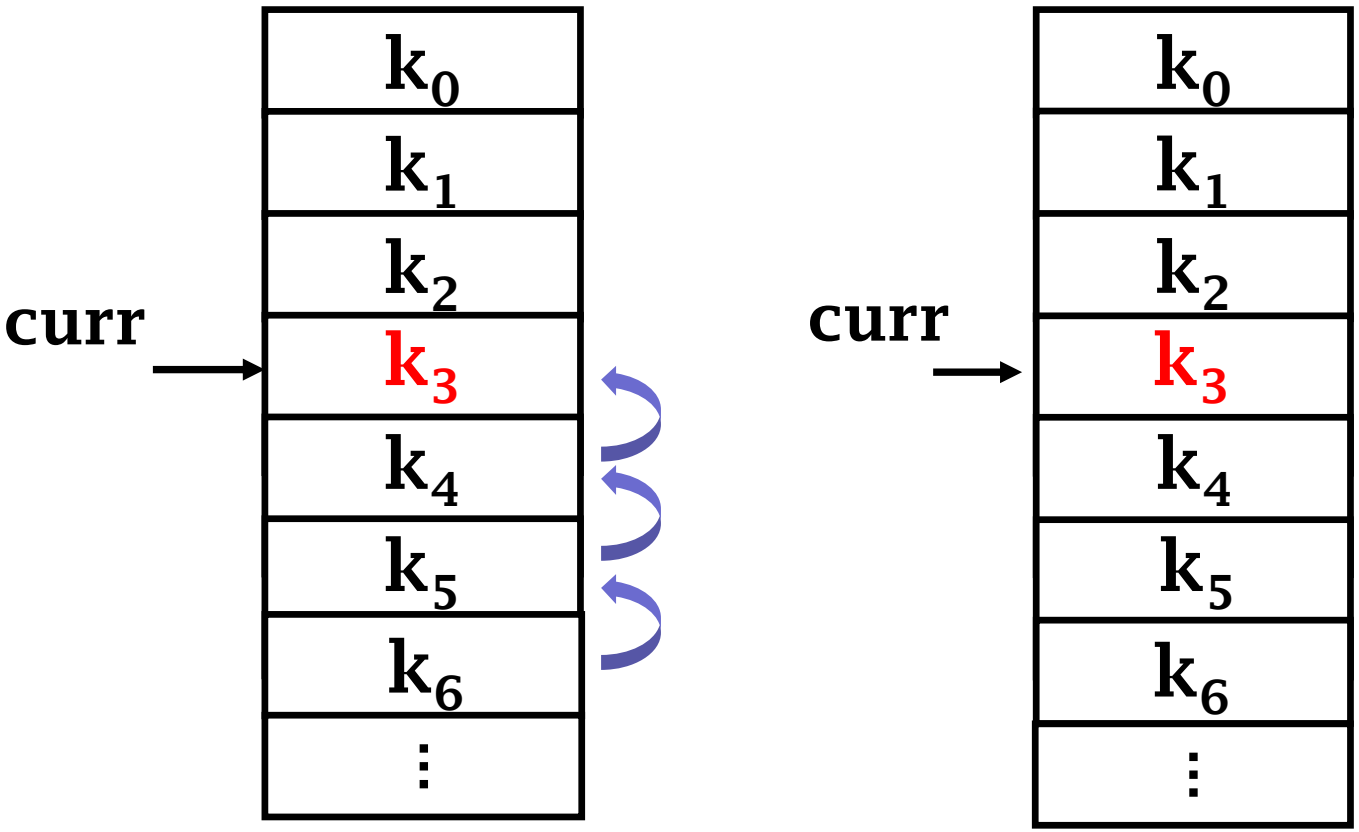
# 顺序表的插入图示



# 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;
}
```

# 顺序表各运算的算法分析

- 插入和删除操作的主要代价体现在表中**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$p_i = \frac{1}{n+1}$，$p_i' = \frac{1}{n}$

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

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

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

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

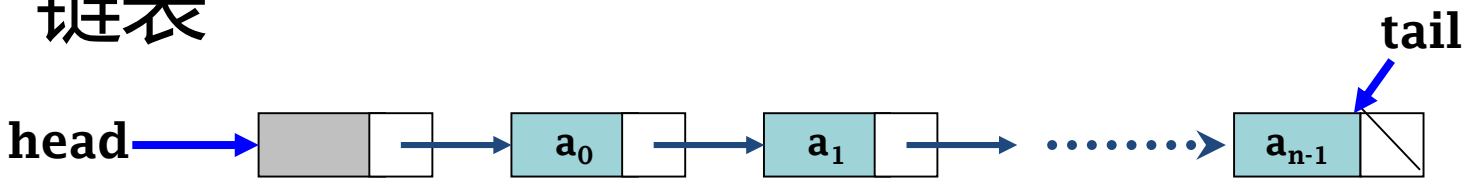
时间代价为$\boldsymbol{O(n)}$

# 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

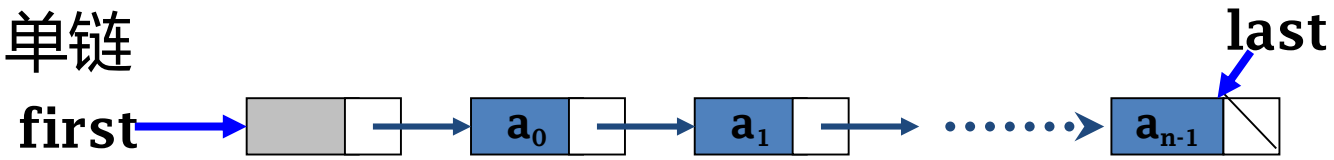



- 2.4 顺序表和链表的比较

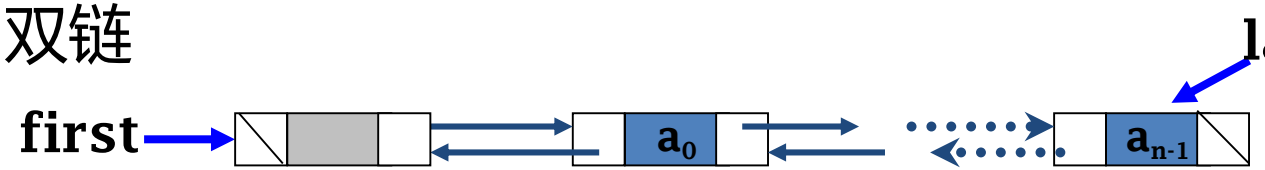
# 2.3 链表

```
template <class T> class Link {
    public:
    T          data;    // 数据域
    Link<T> * next;  // 后继指针
}
```

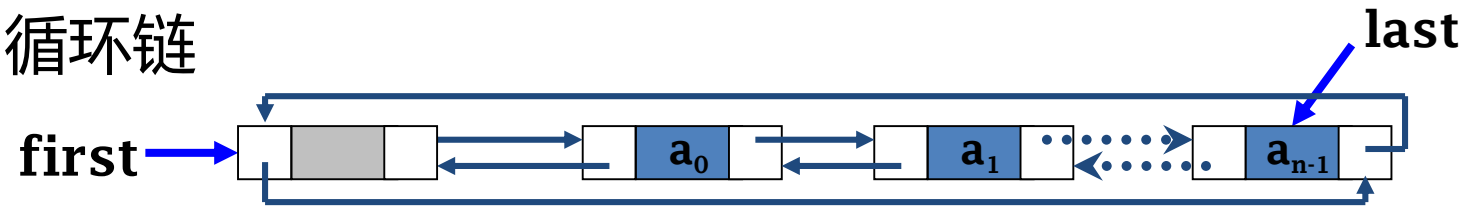
- **存储利用指针**
  - **动态**地为表中新的元素**分配存储空间**
- **分类**
  - 单链




  - 双链

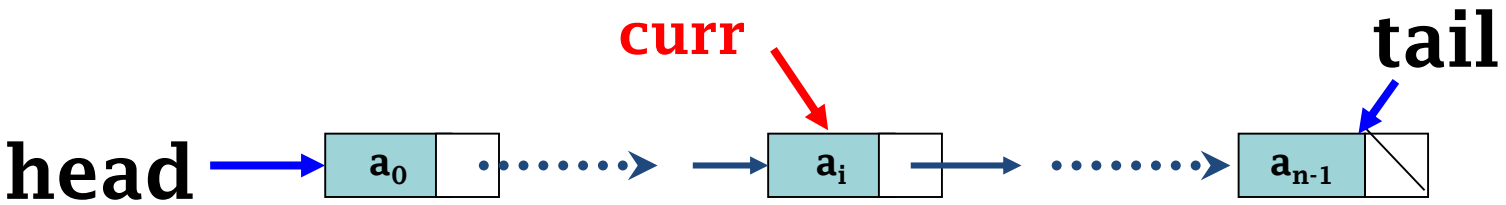



  - 循环链



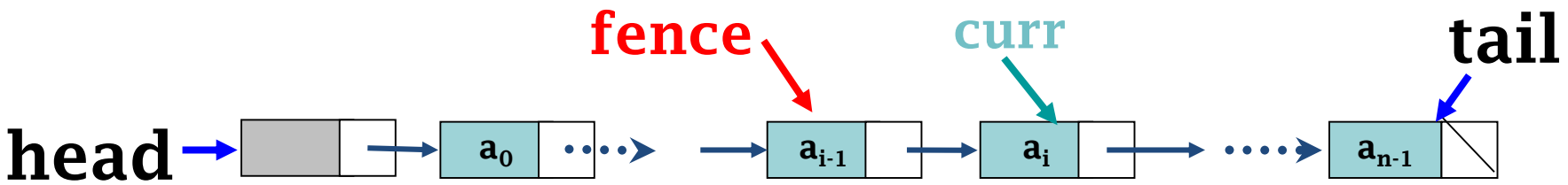
# 单链表

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



# 单链表

- **带头结点的单链表**
  - 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$a_i$：fence->next (curr 隐含)



# 单链表的结点类型

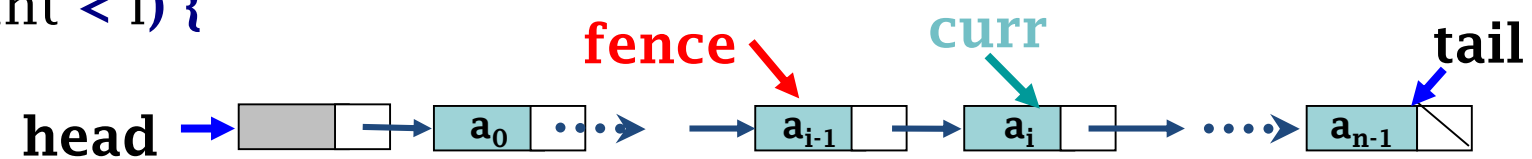
```
template <class T> class Link {
    public:
    T    data;                    // 用于保存结点元素的内容
    Link<T> * next;               // 指向后继结点的指针

    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

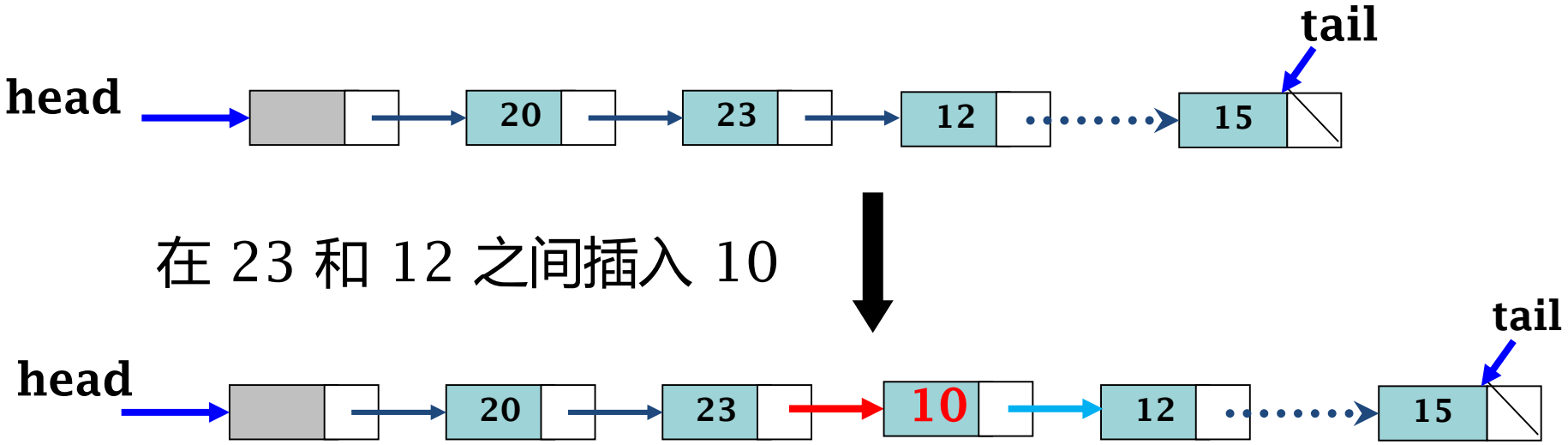
```
template <class T> class lnkList : public List<T> {
    private:
    Link<T>  * head, *tail;                    // 单链表的头、尾指针
    Link<T>  *setPos(const int p);             // 第p个元素指针
     public:
    lnkList(int s);                            // 构造函数
    ~lnkList();                                // 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(cosnt T value);                // 表尾添加一个元素 value，表长度增 1
    bool insert(cosnt int p, cosnt T value); // 位置 p 上插入一个元素
    bool delete(cosnt int p);                  // 删除位置 p 上的元素，表的长度减 1
    bool getValue(cosnt 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，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;
}
```

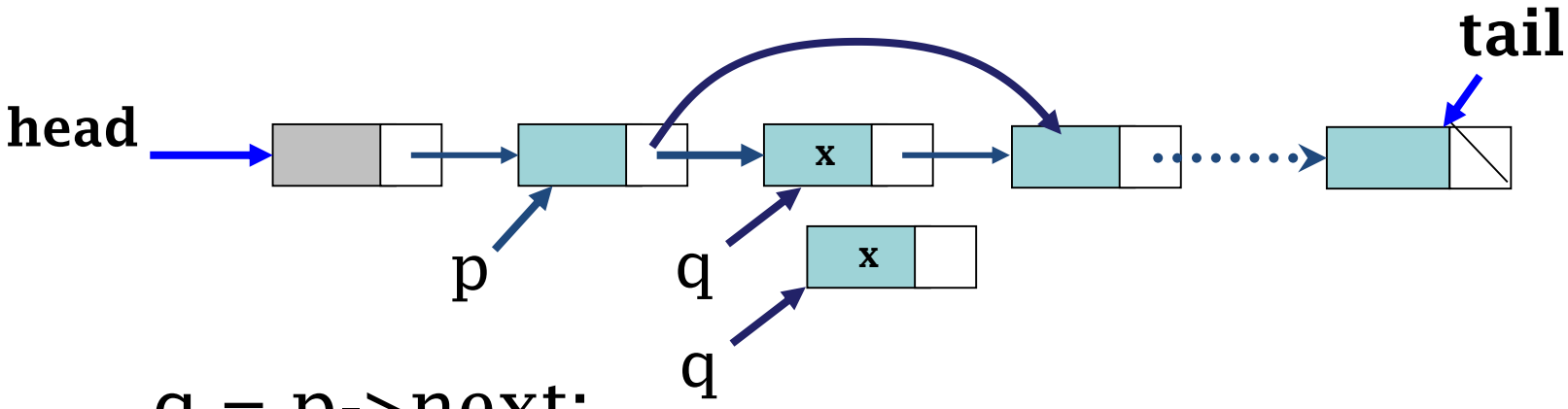
# 单链表的删除

- 从链表中删除结点 x
  - 1. 用 p 指向元素 x 的**前驱结点**

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

  - 2. 删除元素为 x 的结点
  - 3. 释放 x 占据的空间

# 删除值为 x 的结点




```
q = p->next;
p->next = q->next;
// x元素的q结点被剥离

delete 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;
     p = setPos(i-1);
    if  (p == NULLl) {// 待删结点不存在
        cout << " 非法删除点 " << endl;
        return false;
    }
    q = p->next;                      // q 是真正待删结点
    p->next = q->next;                // 删除结点 q 并修改链指针
    if (q == tail)                    // 待删结点为尾结点，则修改尾指针
        tail = p;
    delete q;
    return true;
}
```

# 单链表上运算的分析

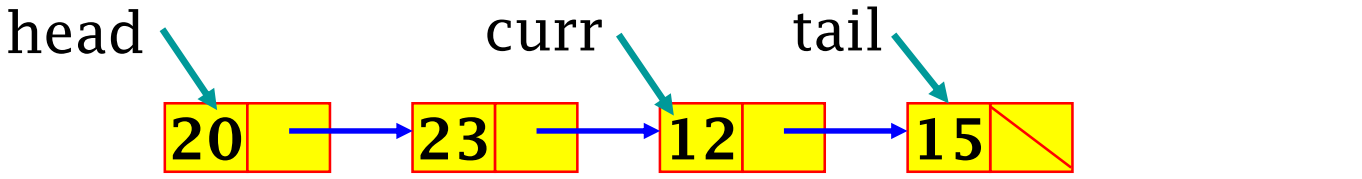
- 对一个结点操作，必先找到它，即用一个指针指向它
- 找单链表中任一结点，**都必须从第一个点开始**

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

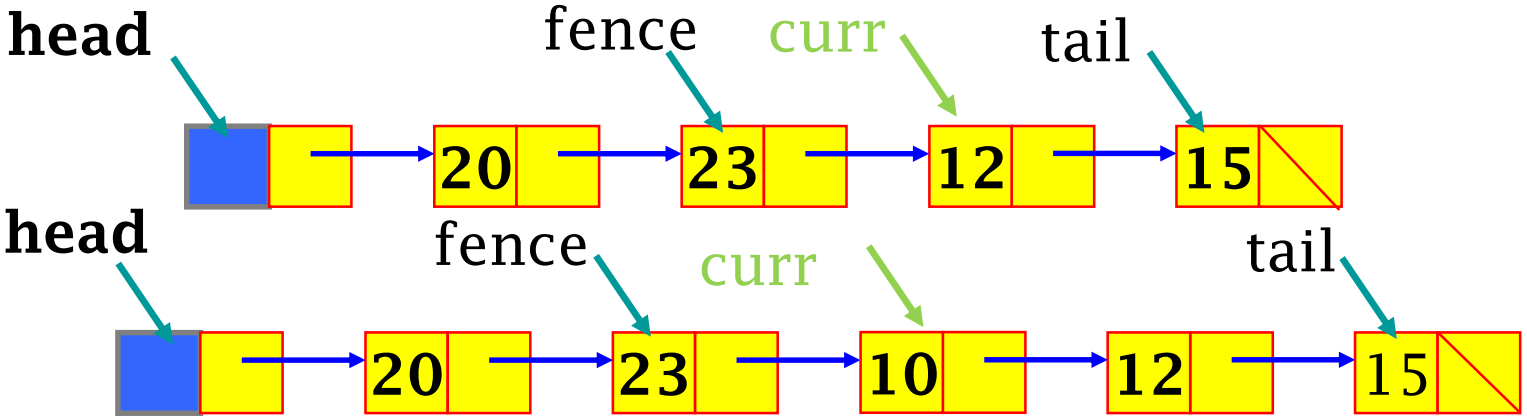
- 单链表运算的时间复杂度 $O(n)$
  - **定位：$O(n)$**
  - 插入：$O(n)+O(1)$
  - 删除：$O(n)+O(1)$

# 不带头结点 vs 带头结点

不带头结点

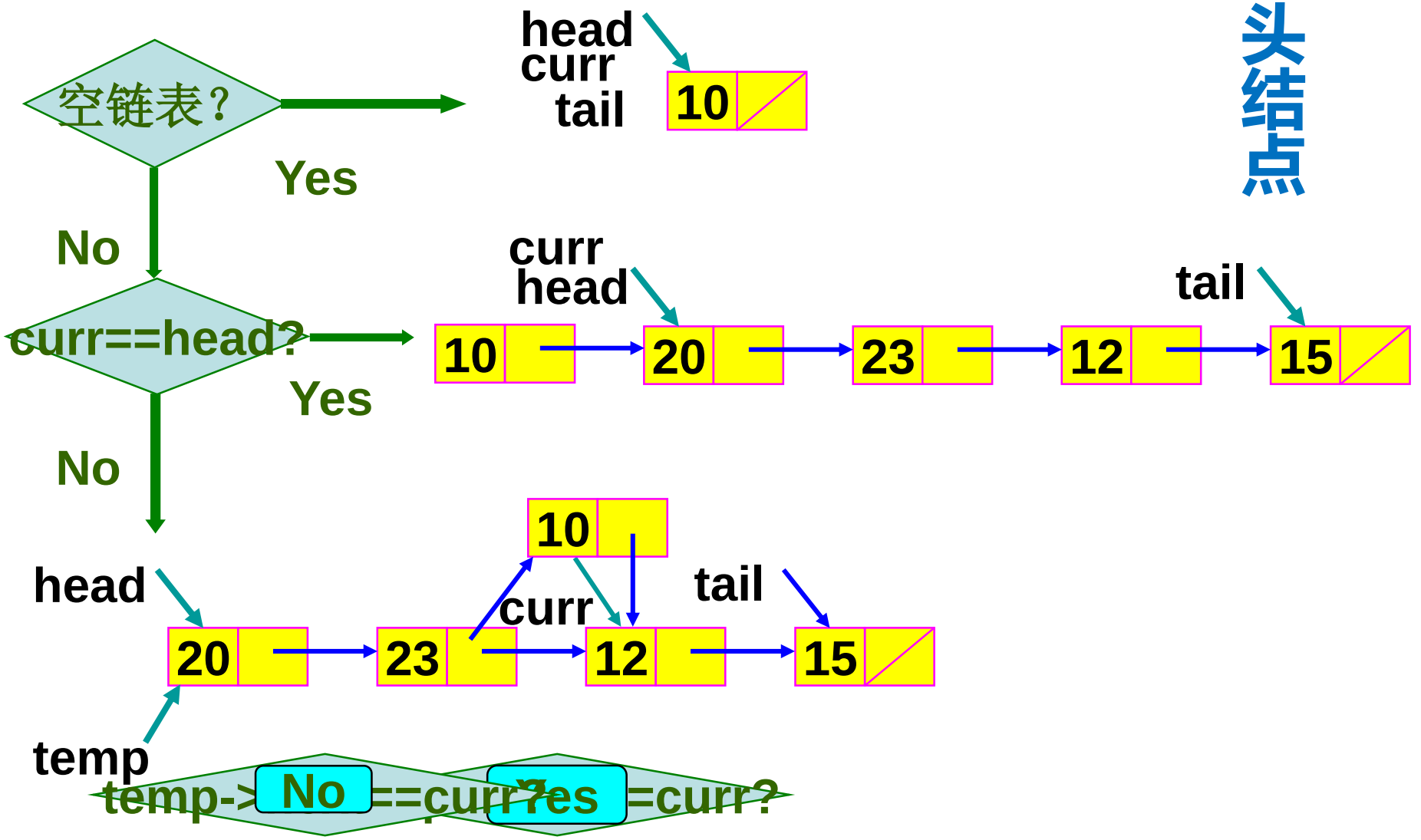



带头结点

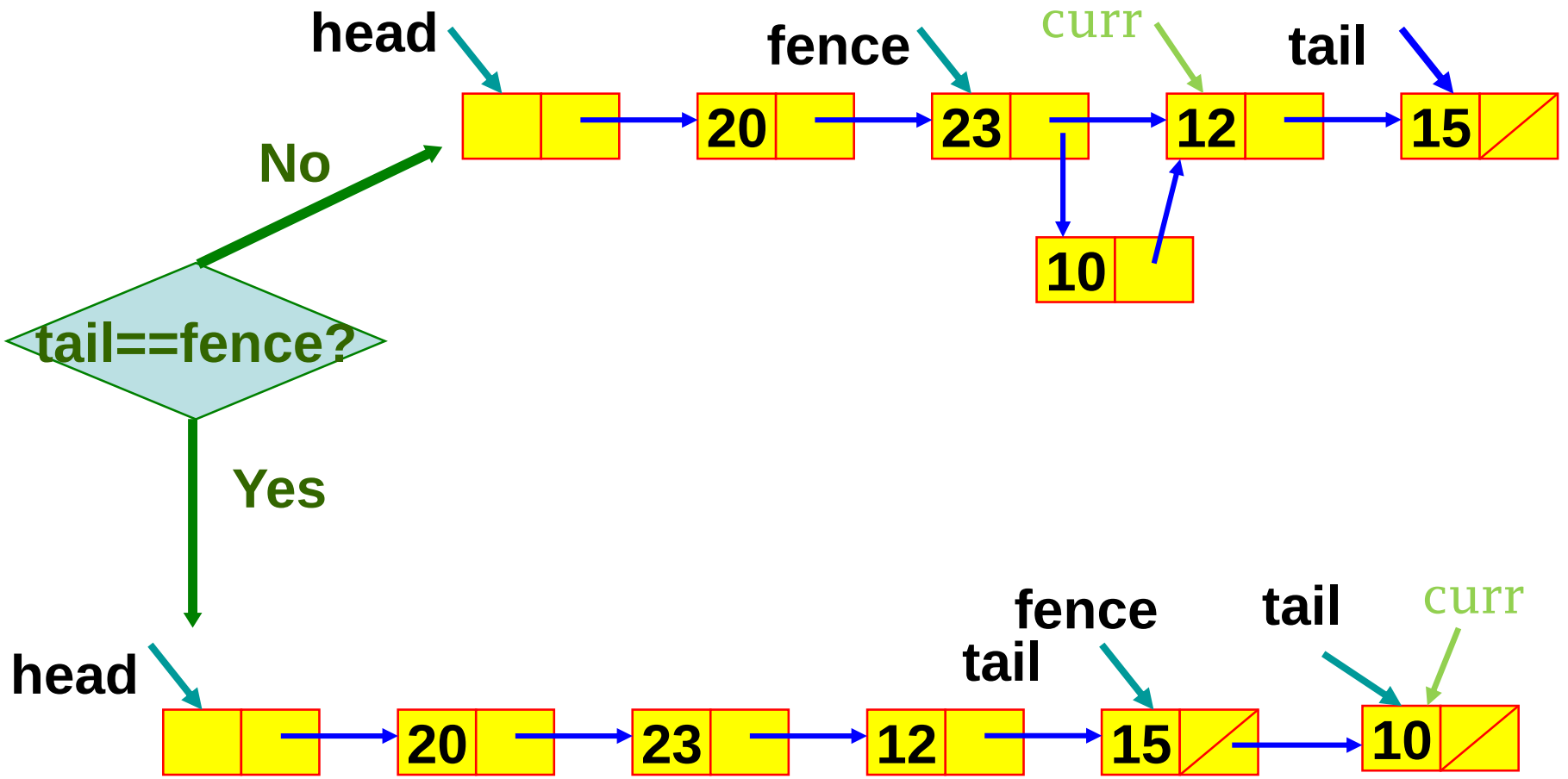


# 在指定位置前插入数值10

不带头结点



# 带头结点，当前位置前插入结点10



// 带头结点

```
template <class Elem>
bool lnkList<Elem>::insert(const Elem& item)
    {
    fence->next = new
      Link<Elem>(item, fence->next);
    if (tail == fence)
       tail = fence->next;
    rightcnt++;
    return true;
}
```

// 不带头结点

```
template<class Elem>
bool NHList<Elem>::insert(const Elem& item)
    {
    if (head == NULL)
          head = curr = tail = new
    Link<Elem>(item, NULL);
    else {
          Link<Elem>* temp = head;
          if (temp == curr) {
              head = new
    Link<Elem>(item, head);
              curr = head;
          }
          else {
              while(temp->next != curr)
    temp = temp->next;
              temp->next = new
    Link<Elem>(item, curr);
              curr = temp->next;
          }
    }
    rightcnt++;
    return true;
}
```

# 约瑟夫问题的由来

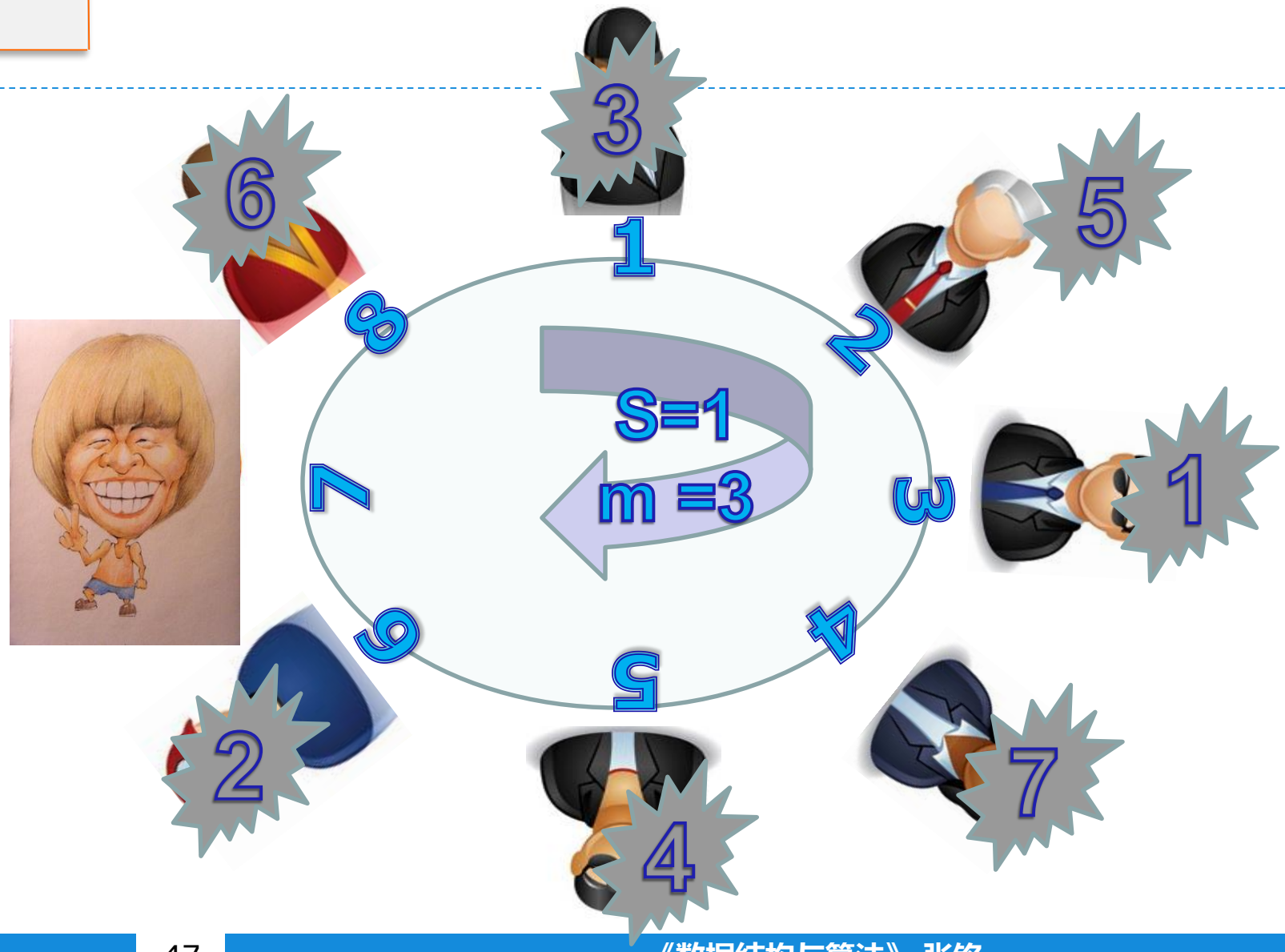
- Josephus （弗拉维奥·约瑟夫斯，一世纪的犹太历史学家，在自己的日记中写道：

  在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友（共41人）躲到一个洞中，大家决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式。

# 约瑟夫问题的由来

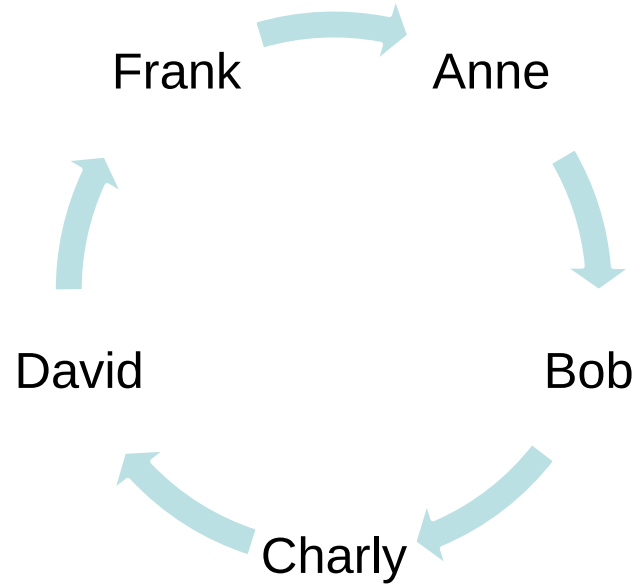
- 自杀方式的约定：
  - 41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。
  - Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

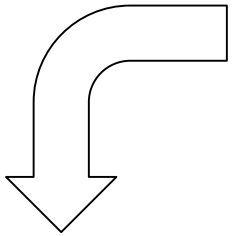
- 后传：
  - 约瑟夫斯说服了他的朋友，他们将向罗马军队投降，不再自杀。


3
1
5
2
1
4
7
5
4
2
6
9
7
3
8
S=1
m =3

# 约瑟夫问题的解答算法

- **Josephus问题描述**：对于任意给定的 n, s和m，求按出列次序得到的人员序列。
  - n：参与游戏的人数，每个人的信息
  - s：开始的人
  - m :单次计数



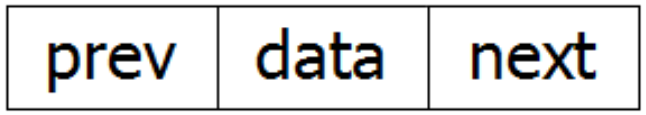



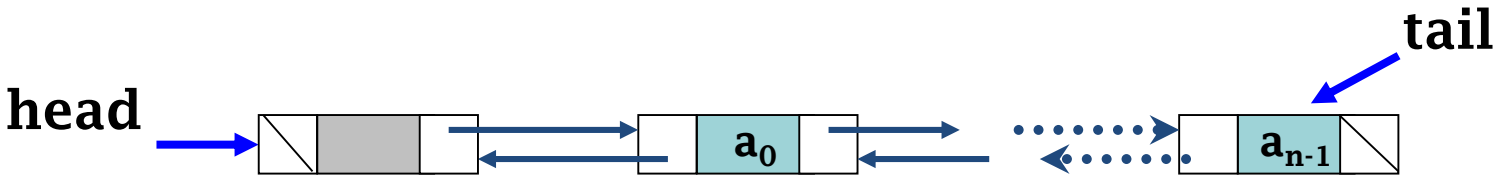
| Anne | Bob | Charly | David | Frank | … | n |
|---|---|---|---|---|---|---|

顺序表

# 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



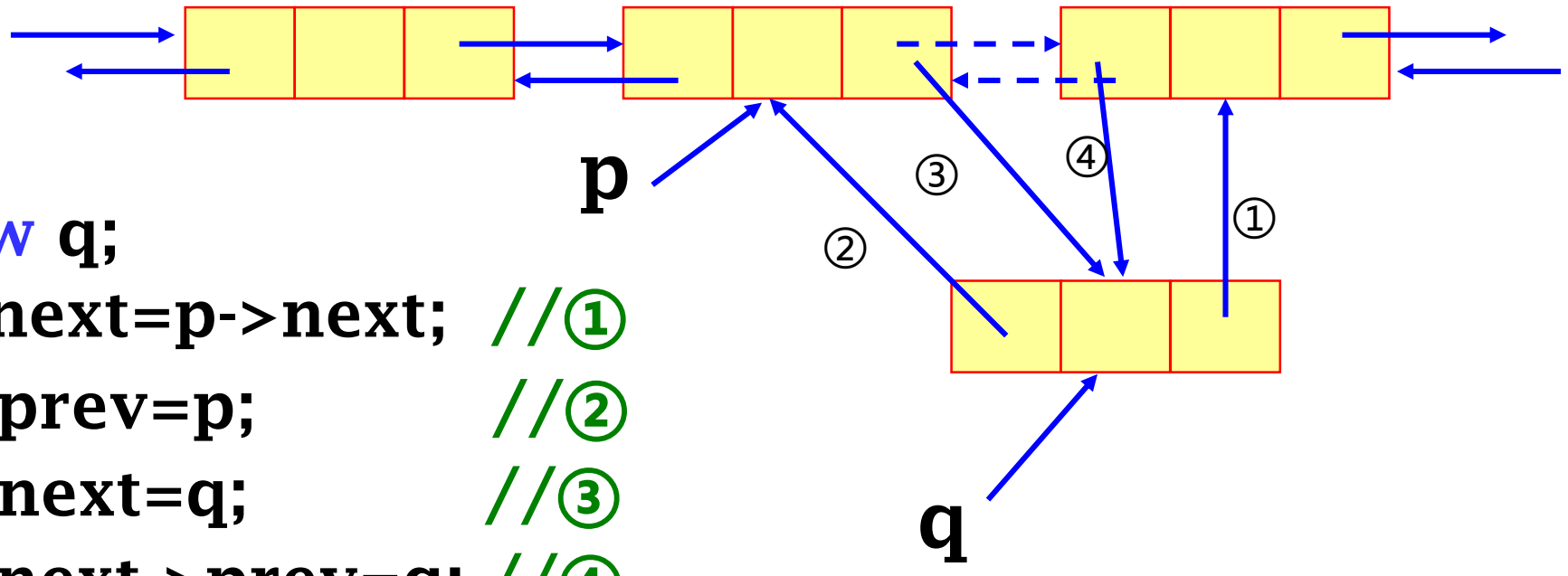



# 双链表及其结点类型的说明

```
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）

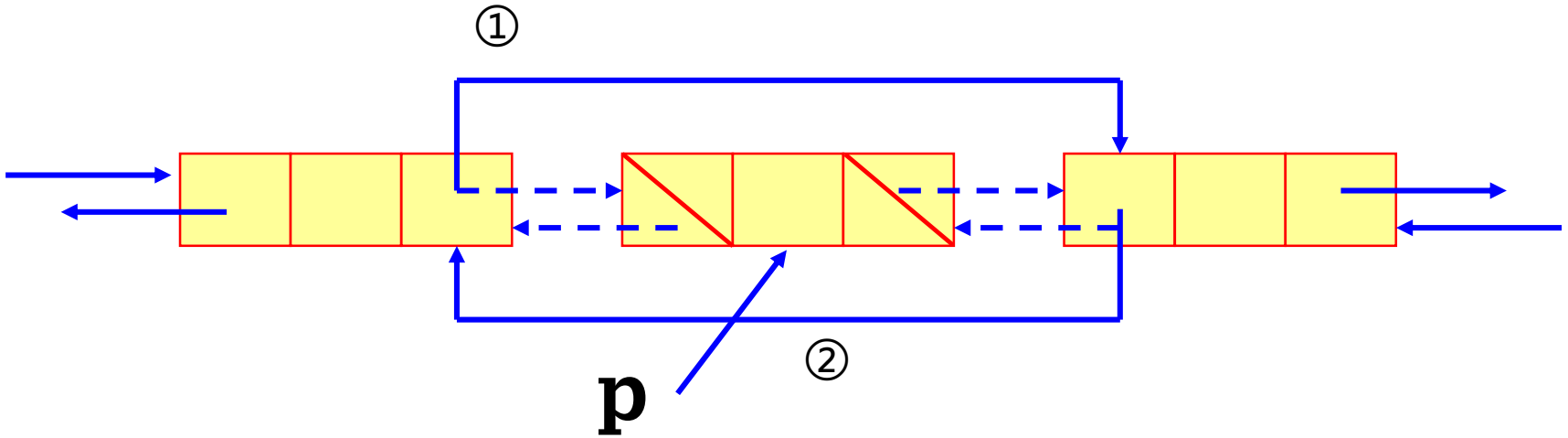
在p所指结点后面插入一个新结点




```
new q;
q->next=p->next;  //①
q->prev=p;        //②
p->next=q;        //③
q->next->prev=q;  //④
```

# 删除过程

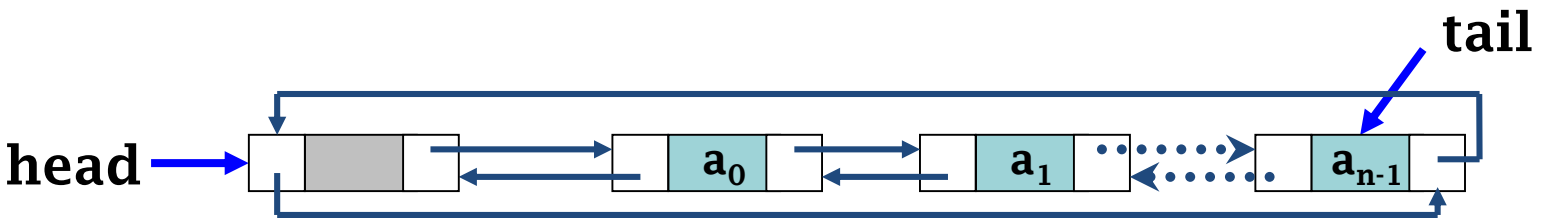
- 删除p所指的结点




```
p->prev->next=p->next;        p->next=NULL;
p->next->prev=p->prev;        p->prev=NULL;
delete p;
```

# 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

# 链表的边界条件

- 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
    - 查找或遍历

例如，

```
while (p != NULL && count < i) {
    p = p-> next;
    count++;
 };
```

# 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

# 存储

- **顺序表**为**静态**结构，而**链表**为**动态**结构
- **数组从栈**中申请空间，而**链表从堆**空间申请

```
ELEM sortarray[1000000];                    // X
ELEM *sortarray =new ELEM[1000000];         // √
```

- 单链表的**存储密度**比顺序表低

$$\text{存储密度} = \frac{\text{数据本身所占存储}}{\text{整个数据结构所占存储}}$$

- 字符串往往用字符数组来实现

# 运算

- **定位**操作
  - **顺序表**，通过定位公式可以**随机**访问任一个元素
  - **单链表**中，需要**顺链**逐个查找

- **修改**操作
  - **顺序表数据元素移动**，而**链表修改指针**
  - 在单链表里进行插入、删除运算比在顺序表容易

# 顺序表和链表的空间需求

**n**，当前元素的数目，
**P**，指针大小（通常为**4bytes**)
**E**，数据域大小
**D**，数组元素的最大数目

- 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)**

# 顺序表和链表的选择

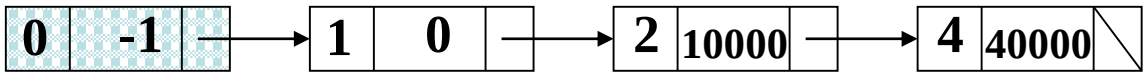
- **顺序表**
  - 结点总数目大概可以估计
  - 线性表中结点比较稳定（插入删除少）
  - **n > DE/(P+E)**
- **链表**
  - 结点数目无法预知
  - 线性表中结点动态变化（插入删除多）
  - **n < DE/(P+E)**

# 例：一元多项式的表达

- 一元多项式：$P_n(x) = p_0+p_1x+p_2x^2+\ldots+p_n\ x^n$
- 线性表表示：$P = (p_0, p_1, p_2, \ldots, p_n)$
- 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

# 一元多项式表达

- 数据定义：

```
struct linknode      {
    float  c;                    //coefficient，记录每一项的系数
    int    e;                     //power，记录每一项的幂指数
    struct linknode *link; //指向下一个节点的指针
};
struct linknode  List;
```

# 思考：多元多项式的表达

- $P(x, y)=x^5y^3 + 2x^4y^3 + 3x^4y^2 + x^4y^4 + 6x^3y^4 + 2y$

| | X0 | X1 | X2 | X3 | X4 | X5 |
|---|---|---|---|---|---|---|
| Y0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Y1 | 2 | 0 | 0 | 0 | 0 | 0 |
| Y2 | 0 | 0 | 0 | 0 | 3 | 0 |
| Y3 | 0 | 0 | 0 | 0 | 2 | 1 |
| Y4 | 0 | 0 | 0 | 6 | 1 | 0 |

把 P(x，y) 重新写作:

$P(x, y)=((x^4+6x^3)y^4+(x^5+2x^4)y^3 +3x^4y^2+2y$

# 思考：多元多项式的表达

- $P(x, y, z)=x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^3y^4z+2yz$

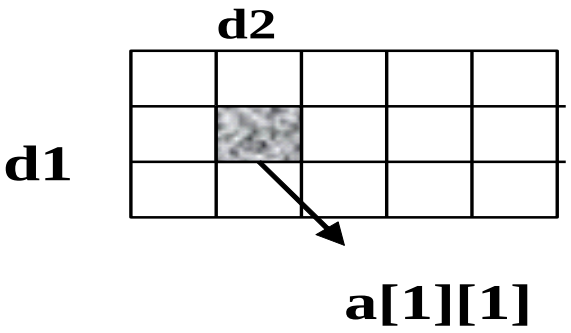
把P(x，y，z)重新写作:

$P(x, y, z)=((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$

**多维数组、广义表**

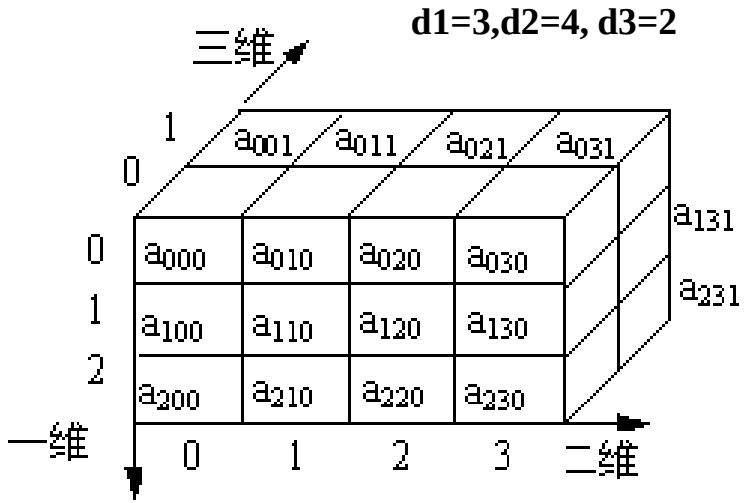
# 数组的空间结构

d1=3,d2=5




二维数组

d1=3,d2=4, d3=2

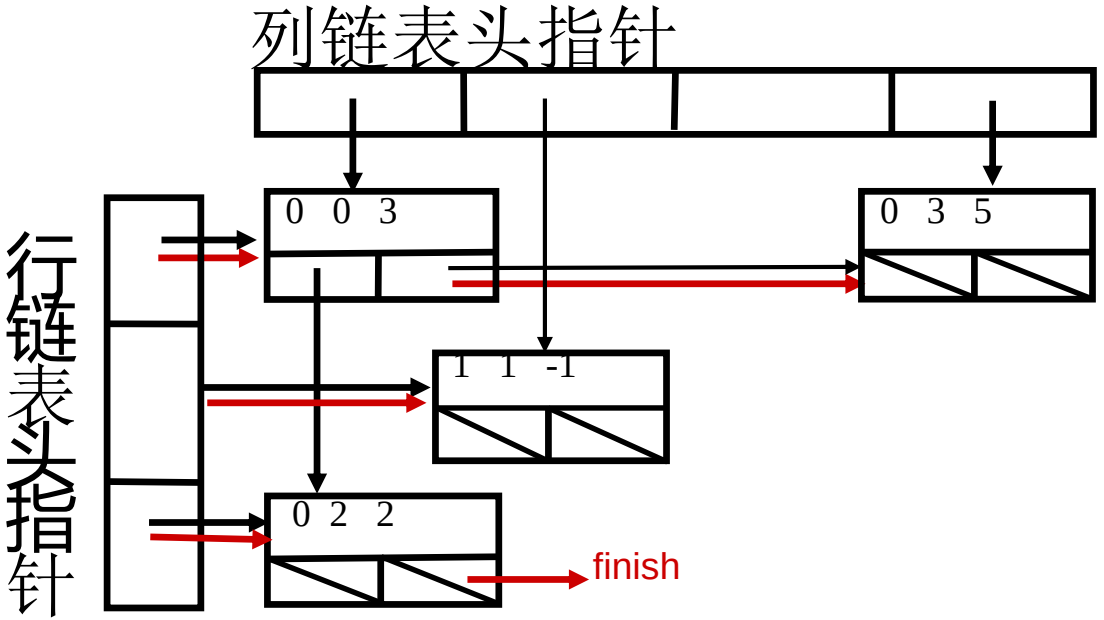


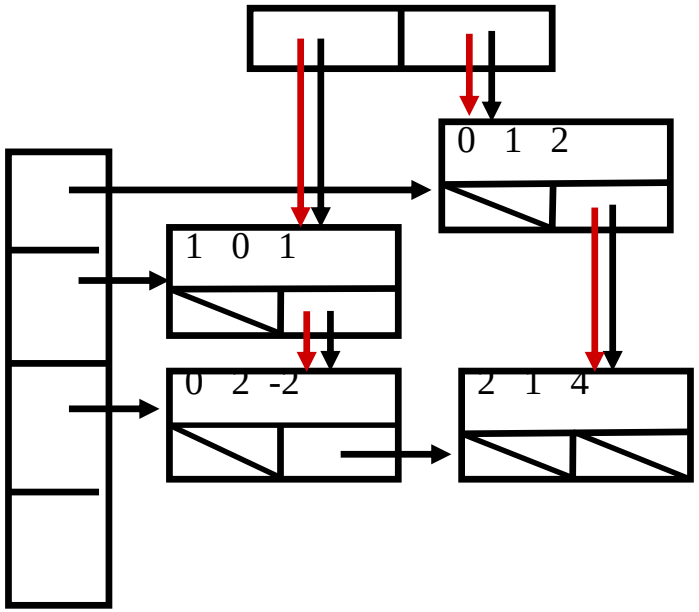

三维数组

d1[0..2],d2[0..3],d3[0..1]分别为3个维

# 稀疏矩阵乘法

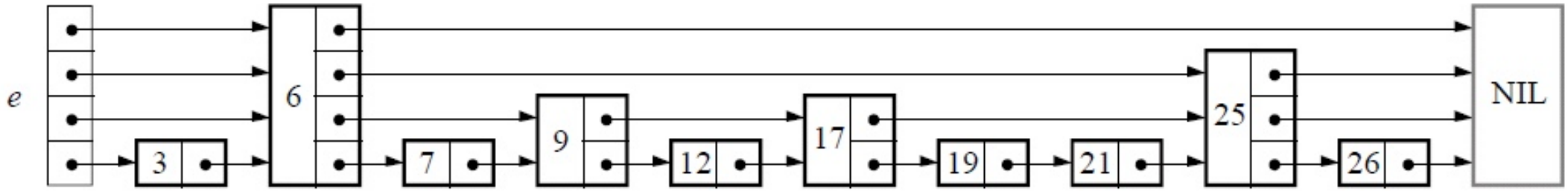
$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$



# 补充：跳表SkipList

跳表是平衡树的一种替代的数据结构
跳表基于一种随机化的算法
跳表的插入和删除比较简单



# 数组的应用

- 1. 如何判断单链表中是否有回路?
- 2. 求数组最大子数组之和

  1, -2, 3, 10, -4, 7, 2, -5

  - 和最大的子数组为【3, 10, -4, 7, 2】
  - 输出为该子数组的和18
- 3. 求矩阵的最大子区域和

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | -10 | -3 | 26 | -11 | -1 | -6 | 12 | 17 | 6 | 28 | 4 |
| 20 | -13 | -20 | -13 | -15 | -254 | 5 | 8 | 9 | -4 | -9 | 29 |
| -11 | 18 | −25 | **9** | **12** | **-9** | **-2** | **23** | 8 | -1 | 3 | -14 |
| -16 | -7 | 0 | **201** | **-1** | **309** | **3** | **6** | −18 | 11 | 24 | -8 |
| -1 | -7 | 11 | **100** | **21** | **292** | **-2** | **2** | -18 | -8 | -10 | 9 |
| 26 | -11 | -19 | -18 | 20 | -981 | 2 | -14 | 12 | -14 | 1 | 27 |
| 9 | -20 | 5 | 28 | -15 | 26 | -20 | -8 | -16 | 30 | 3 | 20 |
| -6 | -7 | -5 | -9 | -16 | -15 | 5 | -16 | 22 | -17 | 11 | -18 |

# 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?**
- **线性表实现方法的比较?**
  - 插入、删除、查找等代价
- **顺序表和链表的选择?**
  - 结点变化的动态性
  - 存储密度

# 数据结构与算法

**谢谢聆听**

国家精品课“数据结构与算法”

http://jpk.pku.edu.cn/course/sjjg/

https://www.icourse163.org/course/PKU-1002534001

**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**