



北京大学

第二章 线性表

宋国杰

gjsong@pku.edu.cn

北京大学信息科学技术学院

内容概要

线性表

- 未施加任何约束的线性结构
- 存储结构：向量和链表

栈与队列

- 操作受限的线性表
- 栈：一端受限；链表：两端受限

字符串

- 内容受限的线性表，内容仅允许为字符
- 典型运算：KMP算法

2.1 线性表

➤ 线性表的概念

➤ 线性表的存储结构

➡ 顺序表—向量

➡ 链表

➤ 线性表的典型操作

线性表的概念

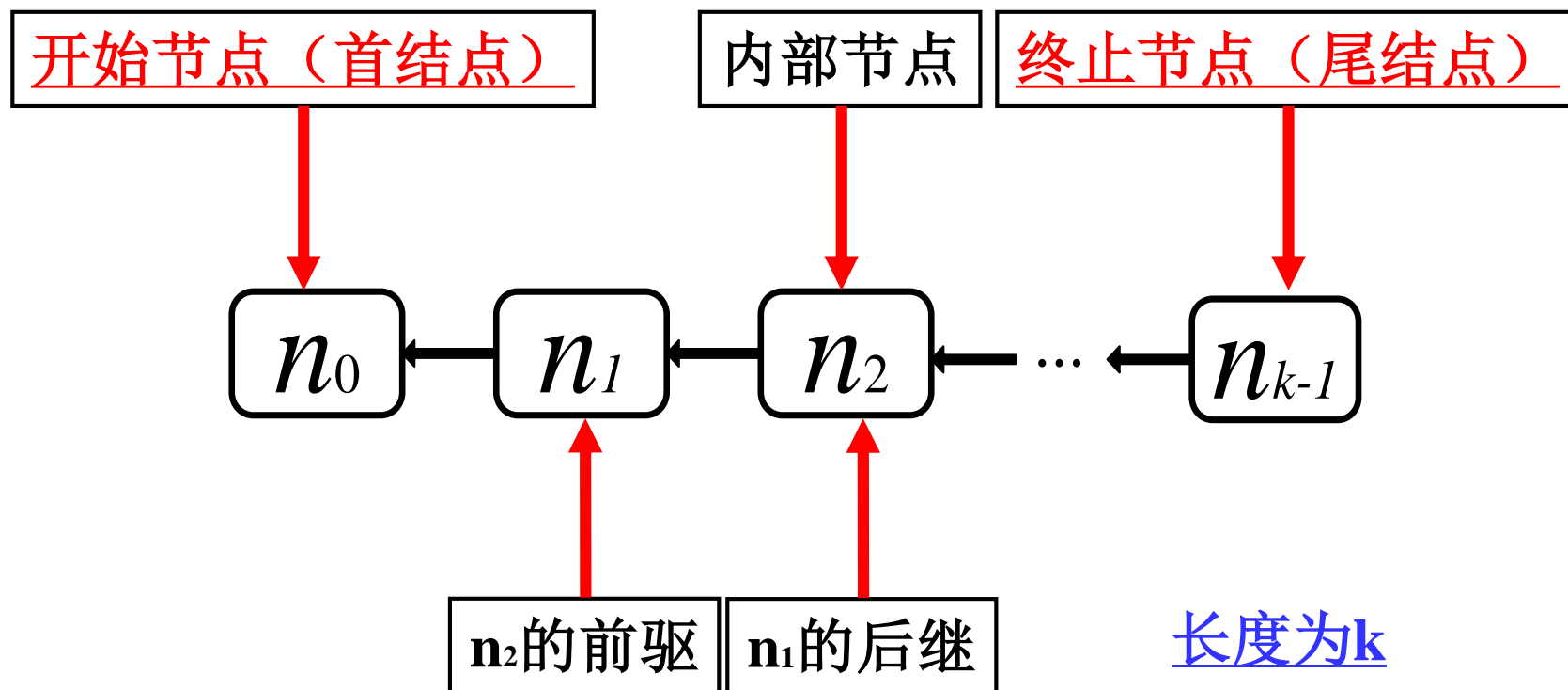
➤ 线性表

➡ 二元组 (K, R) , $K = \{a_0, a_1, \dots, a_{n-1}\}$ $R = \{r: \text{线性关系}\}$

➤ 线性表 (N, r) 的数学定义

- ➡ 唯一的开始结点: 没有前驱, 但有一个唯一的直接后继
- ➡ 唯一的终止结点: 没有后继, 但有一个唯一的直接前驱
- ➡ 内部结点: 有唯一的直接前驱, 也有一个唯一的直接后继
- ➡ 线性表的长度(包含的结点个数)为0的线性表称为空表
- ➡ 线性表的关系 r 是前驱关系, 应具有反对称性和传递性

举例



要求：内部结点具有相同的数据类型

每个元素都有自己的位置【0， n-1】

线性表运算分类

- list(): 创建线性表的一个实例（即构造函数）
- ~list(): 线性表消亡（即析构函数）
- 获取有关当前线性表的信息
 - ➡ 位置寻内容，内容找位置
- 访问线性表并改变线性表的内容或结构
 - ➡ 插入、删除、更改等
- 线性表的辅助性管理操作
 - ➡ 游标、当前长度

线性表ADT

```
template<class T>
```

```
class List { //线性表类模板list, 模板参数T
```

```
    void clear() ;                //置空线性表
```

```
    bool isEmpty();              //线性表为空返回true;
```

```
    void append(ELEM value) //表尾添加元素value, 表长加1;
```

```
    void insert(int p, T value) ; //在p处插入value, 表长加1;
```

```
    void delete(int p);          //删去第p元素, 表长减1;
```

```
    bool getPos(int &p, T value) //查找value, 并返回其位置;
```

```
    bool getvalue(const int p, T & value); //把p位置的值返到value
```

```
    bool setvalue(const int p, T & value); //用value修改p处值;
```

```
}
```

存储结构

➤ 定长、静态的存储结构

- ➡ 又称为向量型的一维数组结构
- ➡ 地址相邻表达线性关系，存储在连续的地址空间，随机访问，但长度固定

➤ 变长、动态的存储结构

- ➡ 链式存储结构
 - 指针指向表达线性关系
- ➡ 动态数组
 - 提供空间表管理，为长度变化提供方法，长度增大，可申请大空间

2.2 顺序表—向量

- 顺序表(Sequential list), 又称向量(Vector)
- 采用定长的一维数组存储结构
- 主要特性:
 - ➡ 元素的类型相同
 - ➡ 存储在连续的空间中, 每个元素唯一的索引值 (下标), 读写元素方便
 - ➡ 使用常数作为向量长度, 程序运行时保持不变

逻辑和存储结构

数据元素	k_0	K_1	...	k_i	...	k_{n-1}
逻辑地址	0	1	...	i	...	n-1	...	maxSize-1

□

(a) 线性表的逻辑结构

数据元素	k_0	k_1	...	K_i	...	k_{n-1}	...	
存储地址	b	b+L	...	b+i*L	...	b+(n-1)*L	...	b+(maxSize-1)*L

(b) 线性表的顺序存储结构

$$Loc(k_i) = b + L \times i,$$

其中：基地址： $b = Loc(k_0)$ ；偏移量： $L = sizeof(ELEM)$

向量的类定义

```
enum Boolean {False,True};
```

```
const int Max_length = 100;
```

```
Template <class ELEM> //假定顺序表的元素类型T为ELEM
```

```
class list { //顺序表，向量
```

```
private :
```

```
    T* nodelist; //私有变量，存储顺序表实例的向量
```

```
    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:
```


```
    list(const int size); //构造算子，实参是表实例的最大长度
```

```
    ~list(); //析构算子，用于将该表实例删去
```

```
arrList(const int size) { // 创建一个新顺序表，参数为表实例的最大长度
    maxSize = size;
    aList = new T[maxSize];
    curLen = position = 0;
}

~arrList() { // 析构函数，用于消除该表实例
    delete [] aList;
}

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];
}
```



```
void clear();           //将顺序表存储的内容清除，成为空表
int length();          //返回此顺序表的当前实际长度
bool append(const T value); //表尾增一新元素，表长加1
bool insert(const int p, const T value); //在p位置插入值value，表长加1
bool delete(const int p); //删去位置p的元素，表长减1;
bool setValue(int p, const T value); //用value修改位置p的元素值
bool getValue(const int p, T & value); //把p位置值返回到变量value中
// 查找值为value的元素，并返回第1次出现的位置
bool getPos(int &p, const T value);
}
```

查找元素

➤ 目的: 查找某个位置的值或者某个值的位置

```
template <class T>    // 假定顺序表的元素类型为T
```

```
bool arrList<T> :: getPos (int p, const T value) {
```

```
    int i;                // 元素下标
```

```
    for (i = 0; i < n; i++)    // 依次比较
```

```
        if (value == aList[i]) { // 下标为i的元素与value相等
```

```
            p = i;                // 将下标由参数p返回
```

```
            return true;
```

```
        }
```

```
    return false; // 顺序表没有元素值为value的元素
```

```
}
```

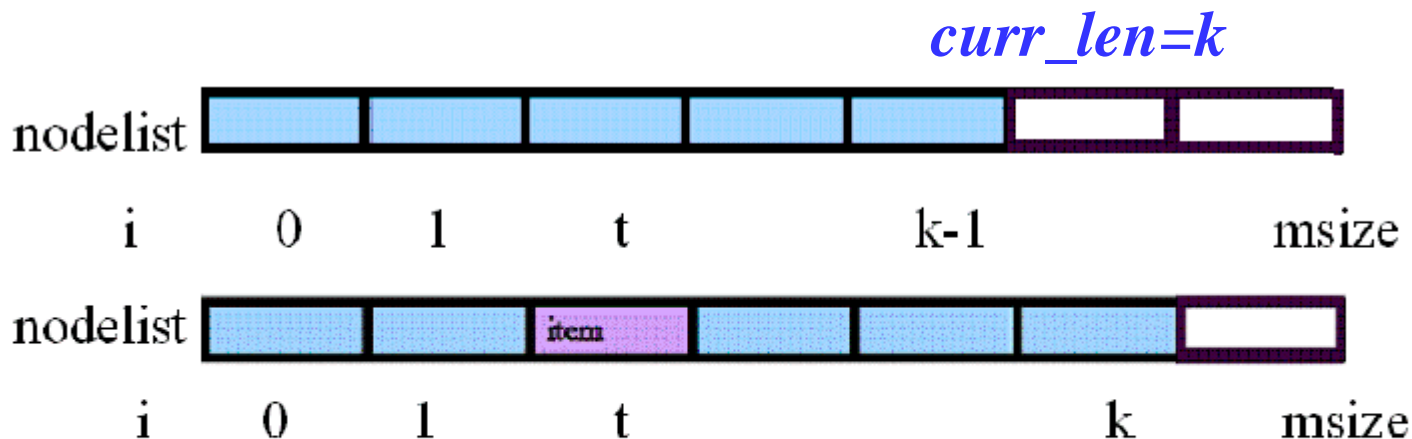
$$\sum_{i=1}^n p \times i = \frac{1}{n} (1 + 2 + \dots + n) = \frac{n+1}{2}$$

$O(n)$

插入元素运算

➤ `bool insert(const int p, const T vlaue)`

➡ 在当前下标 $p=t$ 位置插入元素新值value



(a) 在 t 位置插入元素item

- 条件判断:
- 1、当前下标 $[0, curr_len]$; (是否越界?)
 - 2、当前长度 ($< msize$) (是否溢出?)
 - 3、要先移动, 腾出空间, 再插入!

插入算法

```
template <class T>                                // 假定顺序表的元素类型为T
bool arrList<T> :: insert(int p, const T value) {
    int i;
    if (curLen >= maxSize)    // 检查顺序表是否溢出
        return false;
    if (p < 0 || p > curLen)    // 检查插入位置是否合法
        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;
}
```


算法执行时间

➤ 主要代价

➡ 元素的移动

➤ 元素总个数为 n ，各个位置插入的概率相等为 $p=1/n$

➤ 平均移动元素次数为

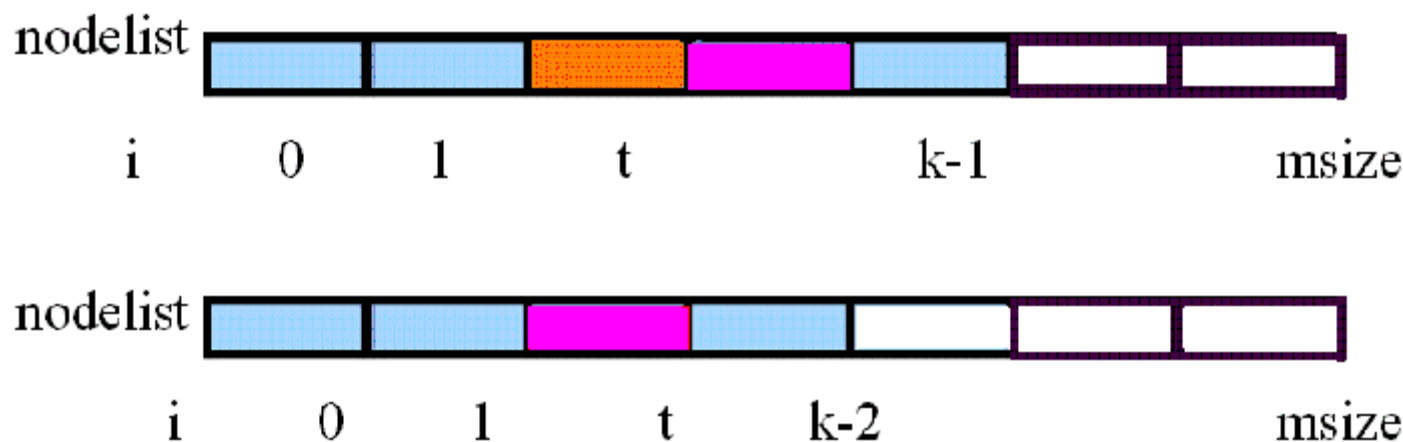
$$\sum_{i=0}^n p \times (n - i) = \frac{1}{n + 1} \sum_{i=0}^n (n - i) = \frac{n}{2}$$

➤ 总时间开销估计为 $O(n)$

删除元素运算

➤ Delete (const int p)

➡ 下标t位置值作为返回值，并删去该元素



- 条件判断：
- 1、当前下标[0, curr len) 删除位置是否有效？
 - 2、当前长度 (>0) 是否向下溢出？
 - 3、删除后，t后元素向前依次移动！

删除算法

```
template <class T>          // 顺序表的元素类型为T
```

```
bool arrList<T> :: delete(int p) {
```

```
    int i;
```

```
    if (curLen <= 0 )           // 检查顺序表是否为空
```

```
        return false ;
```

```
    if (p < 0 || p > curLen-1)   // 检查删除位置是否合法
```

```
        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;
```

```
}
```

算法时间代价

- 与插入操作相似， $O(n)$
- 顺序表读取元素方便，时间代价为 $O(1)$
- 但插入、删除操作则付出时间代价 $O(n)$

2.3 链表(Linked List)

➤ 链表(linked list)

- ➡ 指针指向保持前驱关系，节点不必物理相邻
- ➡ 动态申请/释放空间，长度动态变化（插入/删除）

➤ 在非线性结构（如树、图）中的应用

➤ 分类

- ➡ 单链表
- ➡ 双链表
- ➡ 循环链表

单链表

➤ 结点类型以及变量说明

```
struct ListNode
```

```
{
```

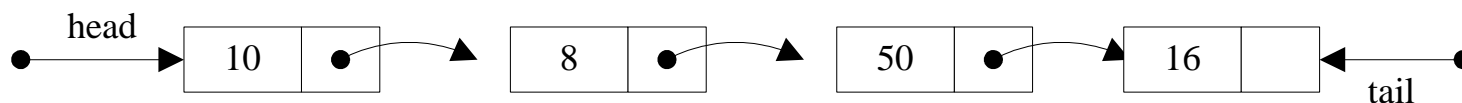
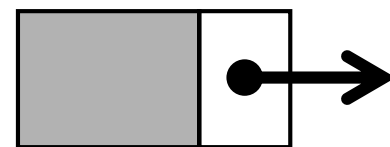
```
    ELEM data;    //存放线性表结点的数据;
```

```
    ListNode *next; //存放指向后继结点的指针;
```

```
};
```

```
typedef ListNode * ListPtr;
```

```
ListPtr head, tail; // head是分别指向单链表头、尾结点的指针;
```



类型定义

```
template <class T> class lnkList : public List<T> {
```

private:

```
    Link<T> *head, tail;
```

// 单链表的头、尾指针

```
    Link<T> *setPos(int p);
```

// 返回线性表指向第p个元素的指针值

public:

```
    lnkList(ints);
```

// 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(T value);
```

// 在表尾添加一个元素value，表的长度增1

```
    bool insert(int p, T value);
```

// 在位置p插入一个元素value，表的长度增1

```
    bool delete(int p);
```

// 删除位置p上的元素，表的长度减1

```
    bool getValue(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，并返回第1次出现的位置
```

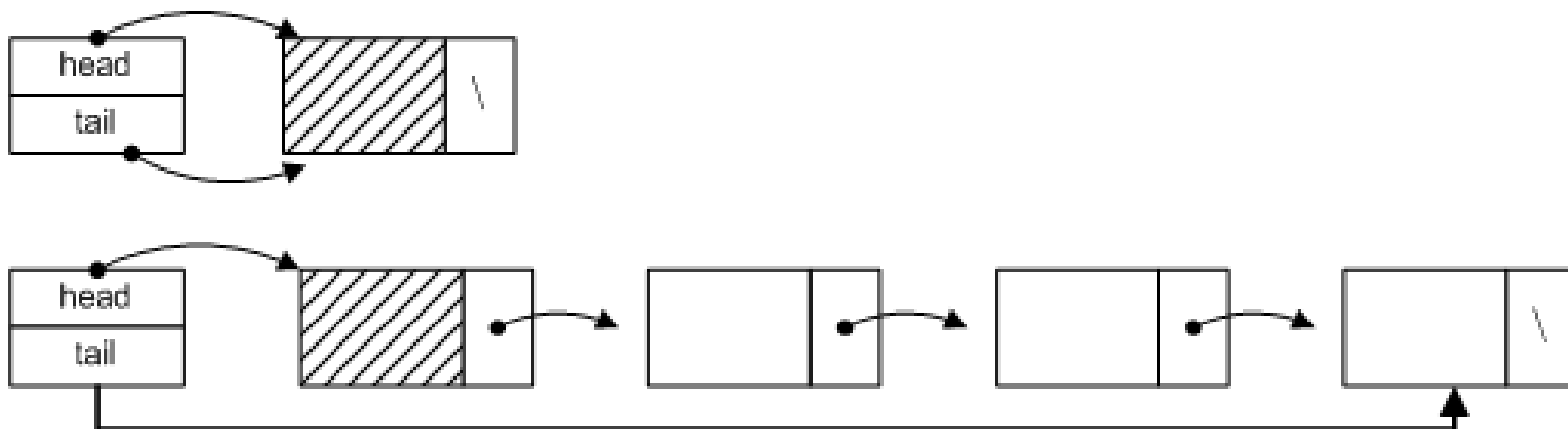
单链表头节点

➤ Header Node（或称“哨兵”）

- ➡ 不被作为表中的实际元素，值忽略
- ➡ head指向该节点

➤ 访问

- ➡ 必须从head开始查找链表中的元素



➤ 为什么要引入头结点呢？

链表检索

// 返回位置i处的结点指针

```
template <class T>
```

// 线性表的元素类型为T

```
Link<T> * InkList <T>:: setPos(int i) {
```

```
    int count = 0;
```

```
    if (i <= -1)    return head;           // i 为-1则定位到头结点
```

```
    Link<T> *p = head->next;
```

```
    while (p != NULL && count < i) { // 若i为0则定位到第1个结点
```

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

```
        count++;
```

```
    };
```

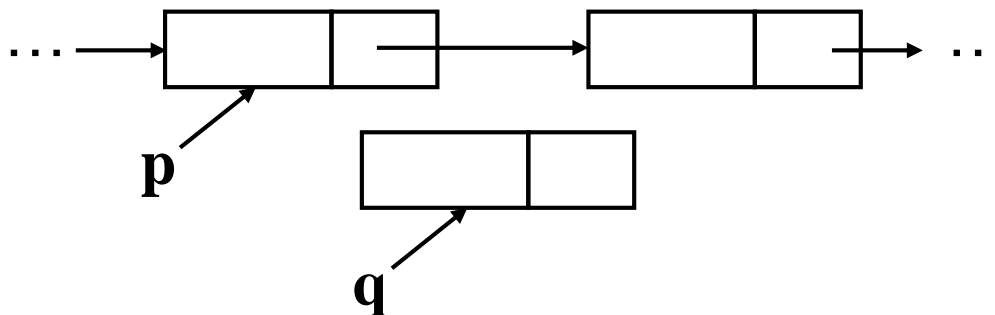
```
    return p;           // 或者为空，或者指向第i个节点！
```

```
}                       // i从0开始！
```

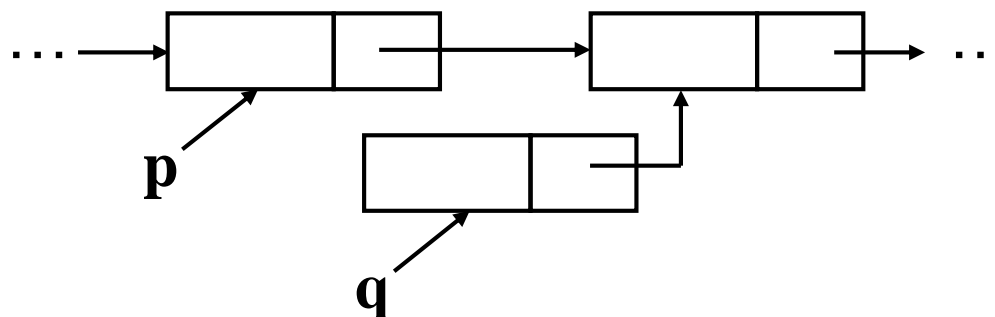
链表插入: `bool insert(int i, T value)`

`p = setPos($i-1$)`

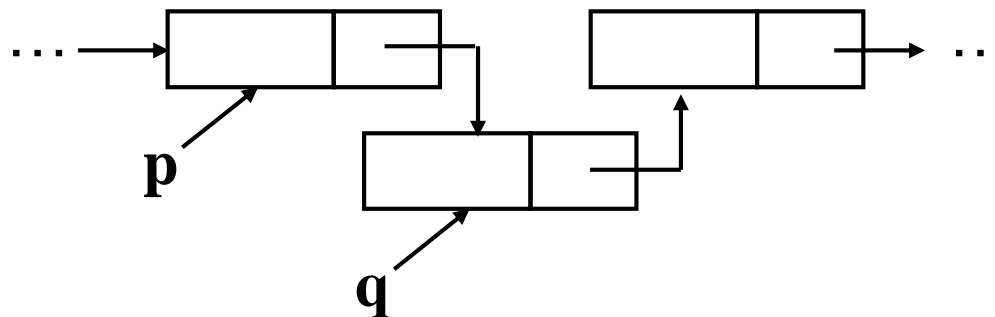
`q = new ListNode`



`q->next = p->next`



`p->next = q`



插入算法

```
ListNode * Insert(int i, T value)
```

```
{
```

```
    ListNode *p, *q;
```

```
    q = new ListNode;
```

//产生一个新结点空间q

```
    p = setPos(i-1);
```

//找到待插位置的前一个位置p

```
    if (p == NULL ) return false;
```

//位置i无效

```
    q->data = value;
```

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

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

```
    if(q->next == NULL )
```

```
        tail=q;
```

指针调整过程

//当插入元素是最后位置时维护尾指针

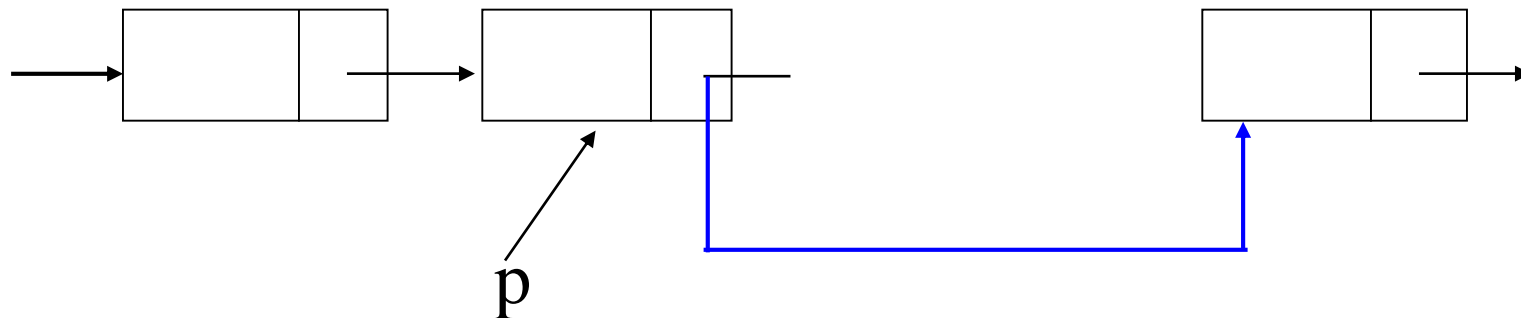
```
    return true;
```

```
}
```

链表删除: `bool delete(int i)`

`p = setPos(i-1);`

d (待删节点)



`d = p→next;` **P节点不能不存在或者为尾节点！！**

`p→next = d→next;`

`delete(d)`

删除算法

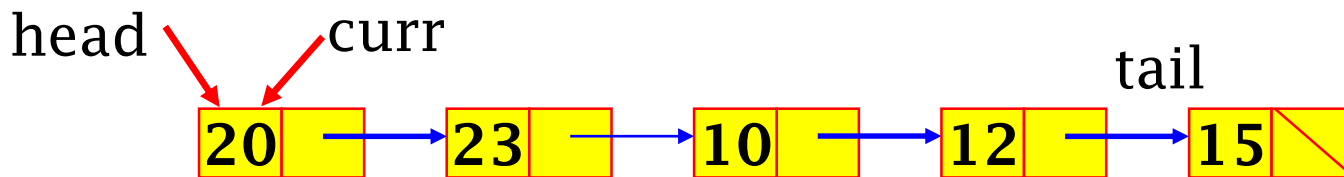
```
template <class T>                                // 线性表的元素类型为T
bool lnkList<T>::delete(const int i) {
    Link<T> *p, *d;
    if (((p = setPos(i-1)) == NULL || p == tail) { // 待删结点不存在;
        cout << " 非法删除点 " << endl;  return false;
    }
    d = p->next;                                // d是真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

设置头结点的好处

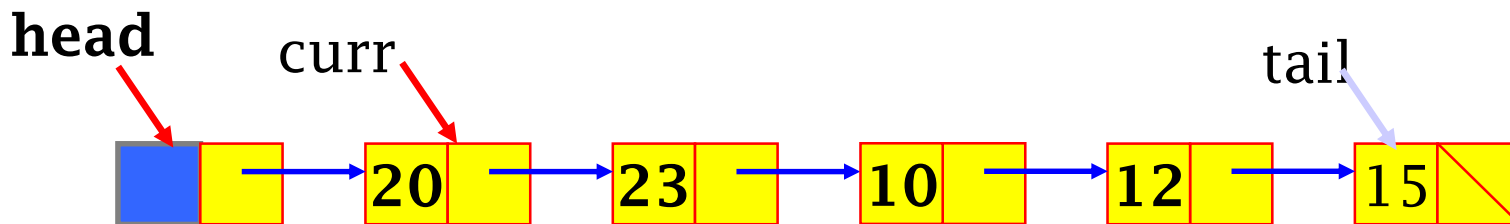
1. 由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上操作一致，无须进行特殊处理；
2. 无论链表是否为空，其头指针是指向头结点的非空指针（空表中头结点的指针域空），因此空表和非空表的处理也就统一了。

举例：不带头结点 vs 带头结点

不带头结点



带头结点



无头结点的插入算法

//无头结点，须对i为0的插入位置作特殊处理

```
ListNode * Insert(int i, T value){  
    if ( i==0){  
        s = new ListNode;           //生成新结点  
        s->data = value; s->next = head; // 插入到链表L中  
        head = s;                   // 修改链头指针L  
    }else{  
        .....                      //同有头结点算法的处理;  
    }  
    return true;  
}
```


单链表分析

➤ 单链表的不足之处

- ➡ **link**字段仅仅指向后继结点，不能有效地找到前驱

➤ 为弥补上述不足

- ➡ 引入双链表

- ➡ 节点增加一个指向前驱的指针

双链表

➤ 类型说明

struct DbListNode

{

T data;

DbListNode *prev;

DbListNode *next;

};

struct DoubleList

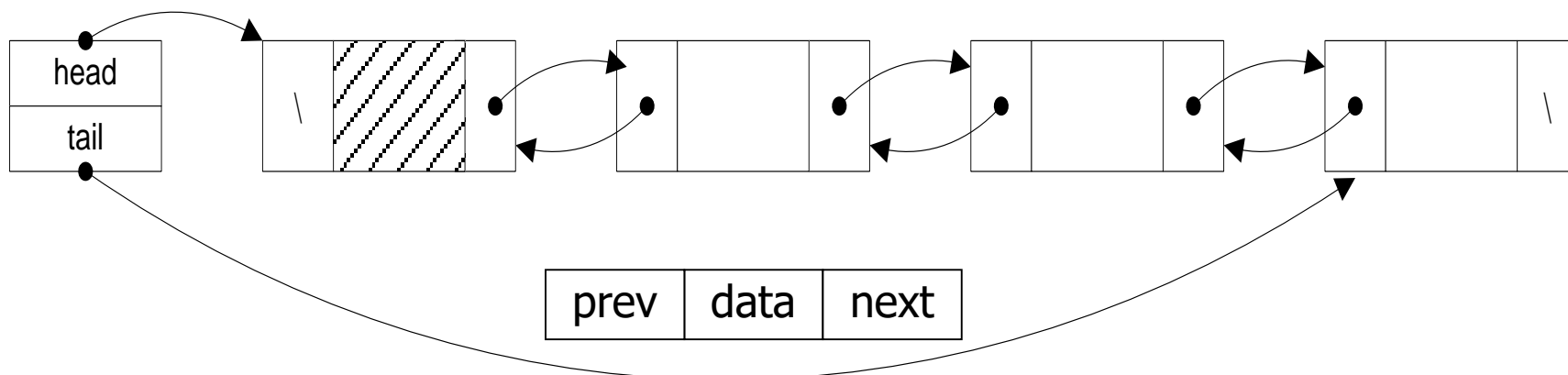
{

DbListNode *head, *tail;

};

➔ 指向前驱结点

➔ 指向后继结点



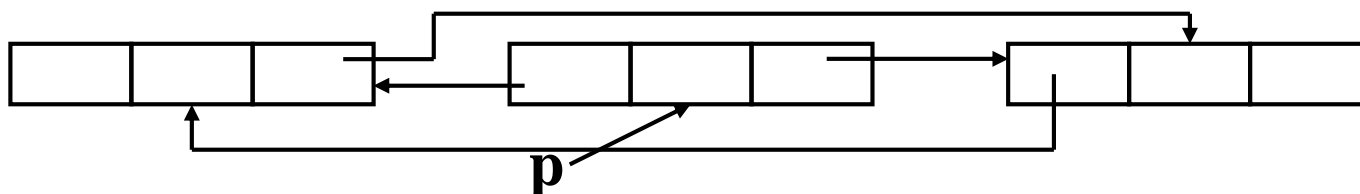
删除结点示意图

删除p所指的结点setPos(i)



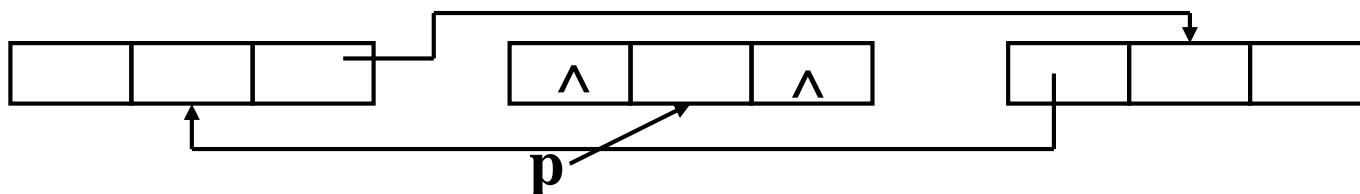
p->prev->next=p->next;

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



p->prev=NULL;

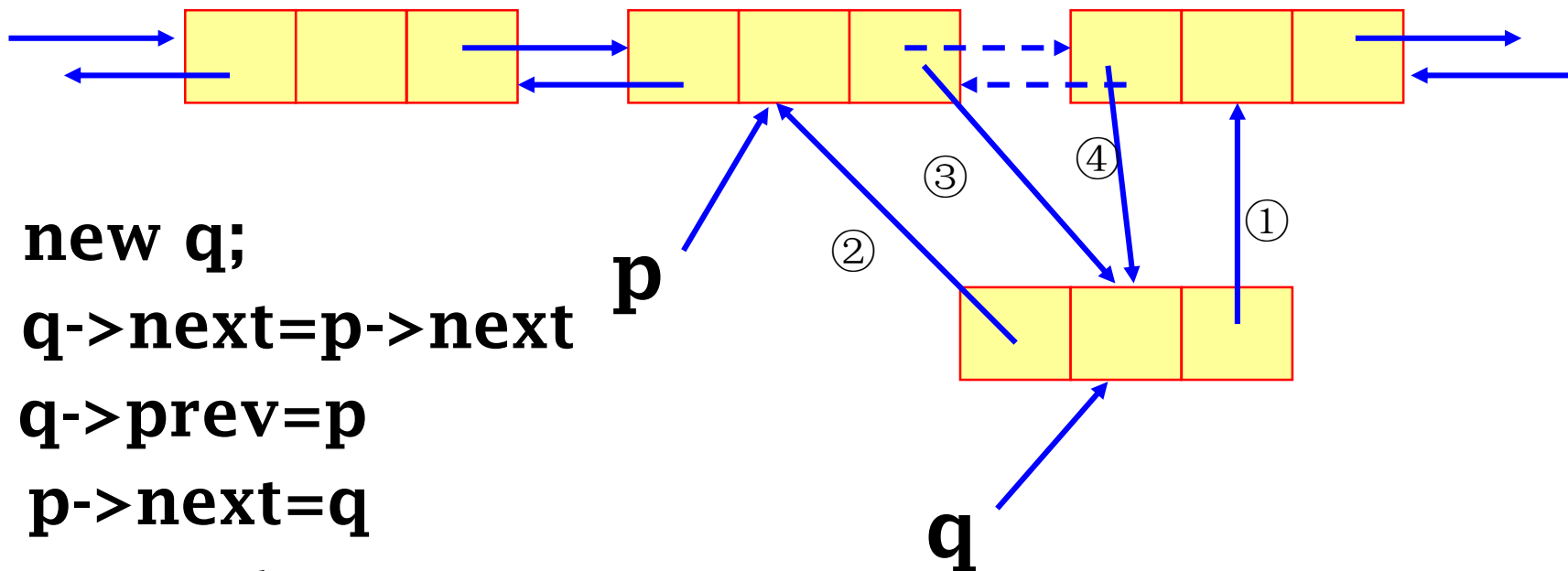
p->next=NULL;



free(p)

插入和删除都要注意边界条件的判断！

在p所指结点后插入一个新的结点setPos(i-1)

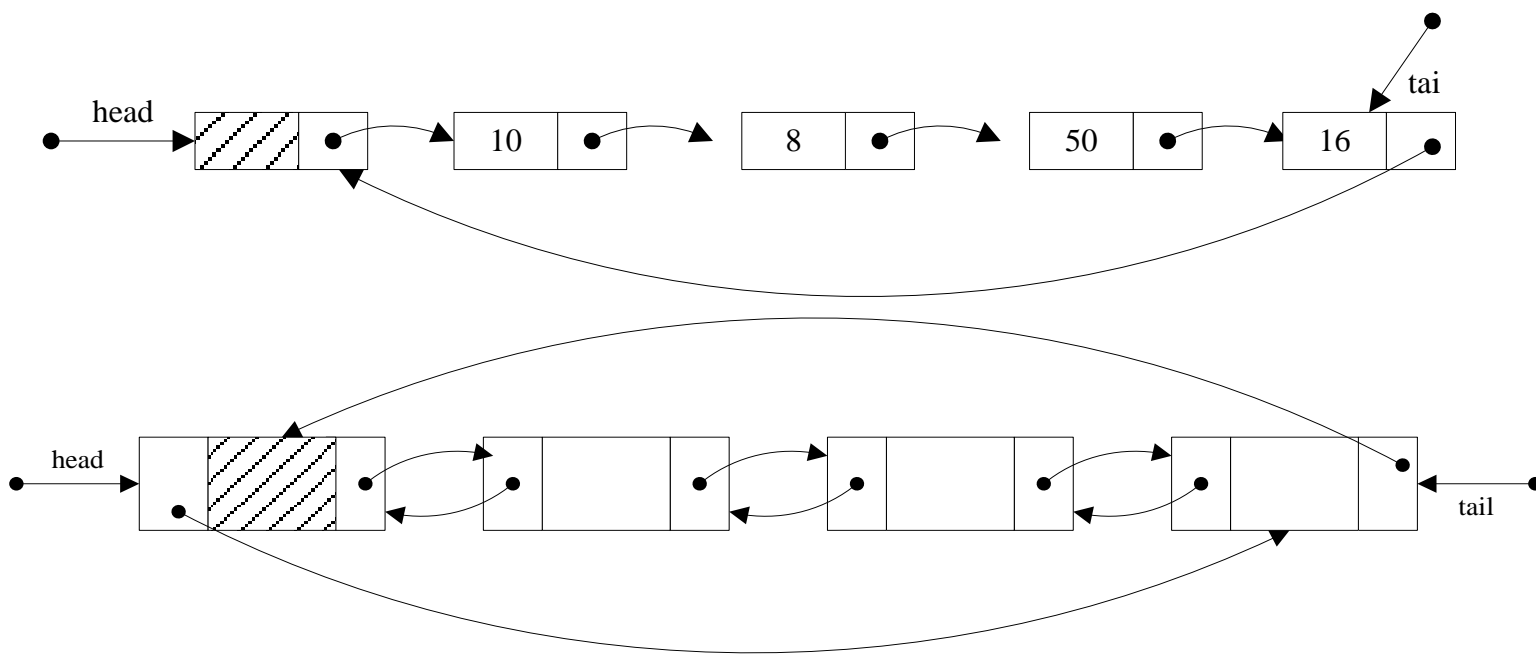


```
new q;  
q->next=p->next  
q->prev=p  
p->next=q  
q->next->prev=q
```

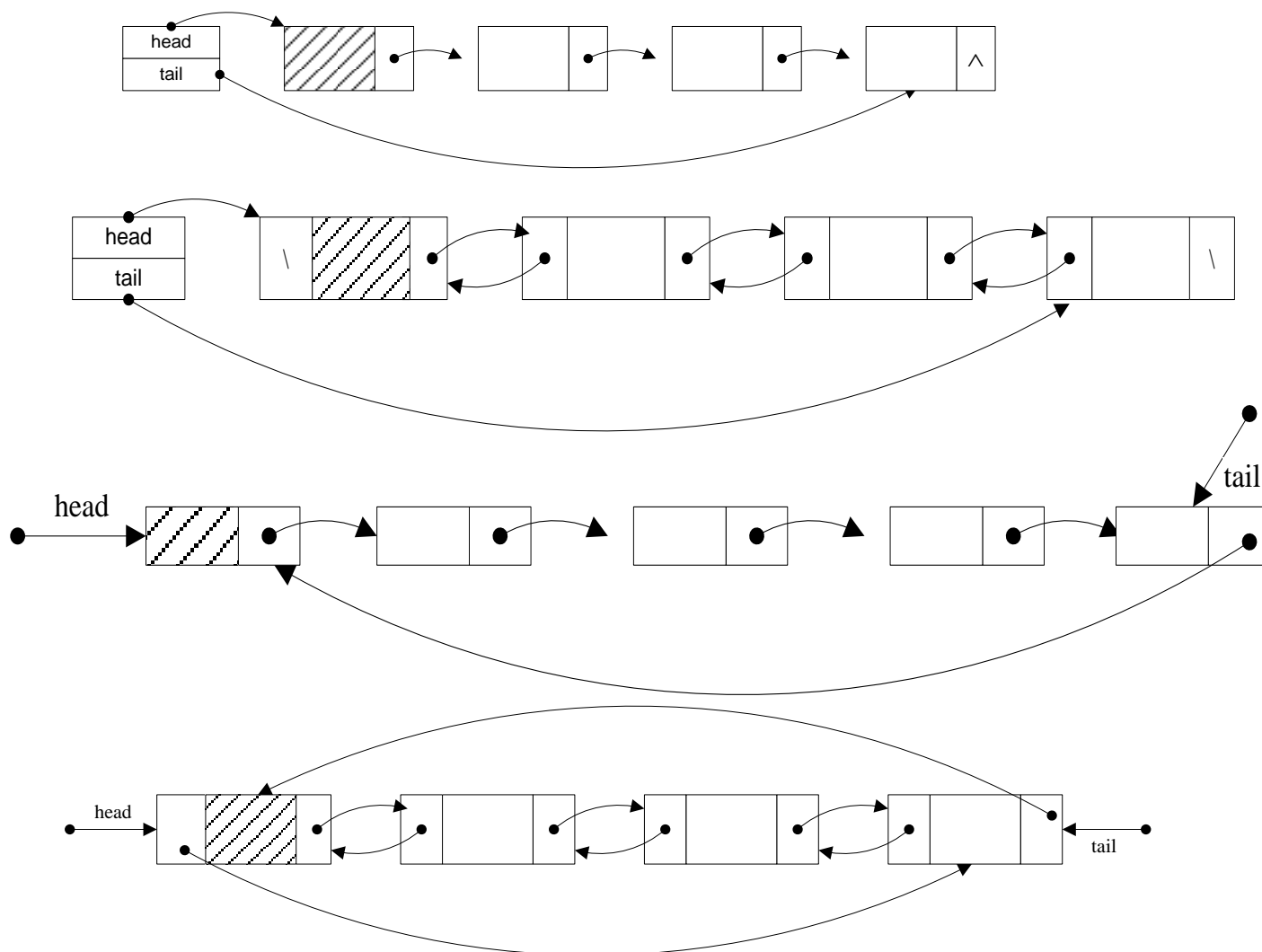
要注意操作的次序！

循环链表

- 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。



几种主要链表比较



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
- ➡ 线性表元素的访问非常便利

➤ 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

➤ 顺序表适合存储静态数据，链表适合动态数据

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数组
- ➡ 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*个元素时间代价 $O(n)$
- ➡ 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

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



再见…

联系信息：

电子邮件：**gjsong@pku.edu.cn**

电 话：**62754785**

办公地点：**理科2号楼2307室**