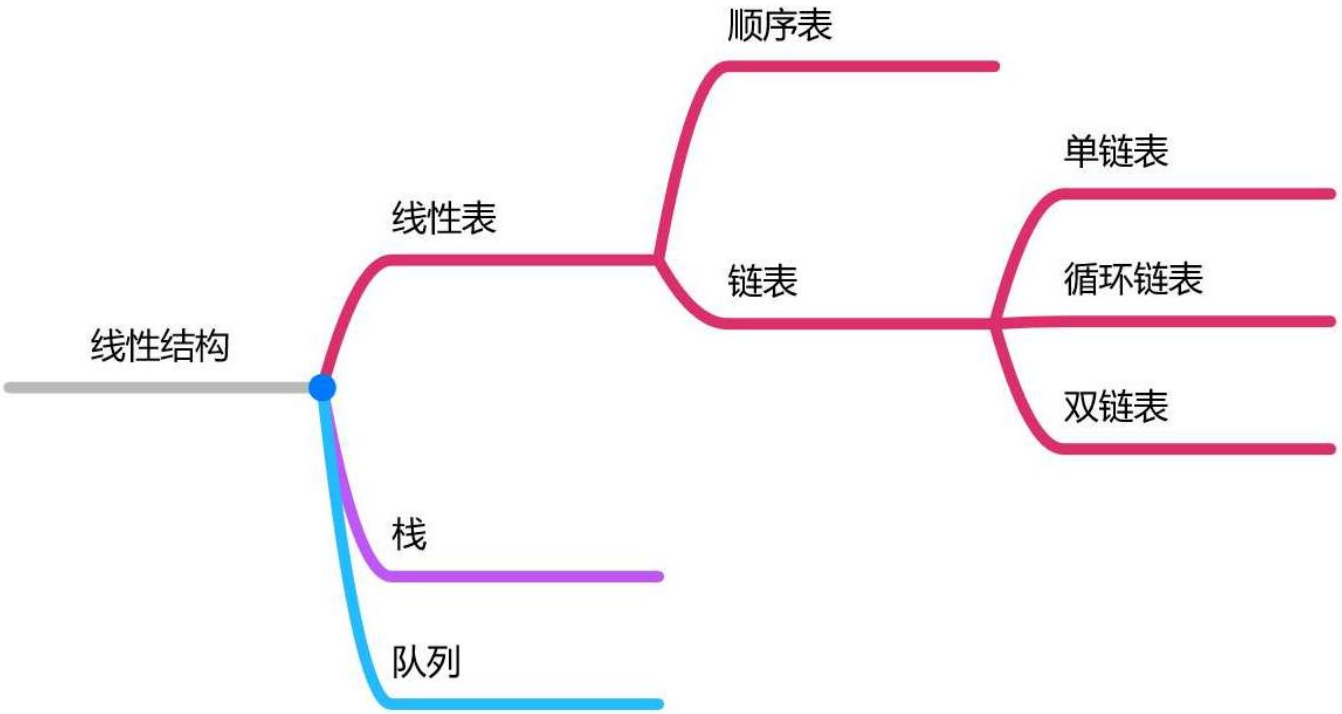
线性结构

—— 线性表

主讲教师：杨晓波

248133074@qq.com

学习要点



4.1 线性表

定义： 线性表L是n个数据元素$a_0,a_1,\ldots a_{n-1}$的有限序列,记作

$L=(a_0,a_1,\ldots a_{n-1})$。

- 其中元素个数$n(n\geq 0)$定义为表L的长度。
- 当n=0时，L为空表，记作( )。

特性：

- 在表中,相邻元素存在序偶关系$<a_{i-1},a_i>$
  - 除第一个元素a_0外，其他每一个元素a_i有且仅有一个直接前驱a_{i-1}
  - 除最后一个元素a_{n-1}外，其他每一个元素a_i有且仅有一个直接后继a_{i+1}。
  - a_0为第一个元素, 又称为表头元素;
  - a_{n-1}为最后一个元素, 又称为表尾元素。

线性表的抽象数据类型(ADT)

```
template <typename E> class List {   //类模板、抽象类
public:
  List() {}
  virtual ~List() {}
  virtual void clear()=0;   //纯虚函数
  virtual void insert(const E& item)=0;
  virtual void append(const E& item)=0;
  virtual E remove()=0;
  virtual void moveToStart()=0;
  virtual void moveToEnd()=0;
  virtual void prev()=0;
  virtual void next()=0;
  virtual int  length() const=0;
  virtual int  currPos() const=0;
  virtual void moveToPos(int pos)=0;
  virtual bool SetValue(Elem&) const=0;
  virtual const E& getValue() const=0;
};
```

- E是类型名，为模板参数；
- 参数前加const，说明变量或者对象的值不能被更新；
- 拥有纯虚函数的类是抽象类，不能直接实例化，仅提供一个接口，常作为子类的父类，子类需要重载这些纯虚函数，才能实例化

ADT的定义法

- 教材使用C++的抽象类表示法

- 抽象类
  - 其成员函数都被声明为纯虚的（pure virtual），即函数声明的最后有"=0"的符号

- 图4.1的list定义的纯虚函数，继承该类的任何线性表实现都必须支持这样的实现，并使用函数所规定的参数和返回类型

线性表的物理实现

顺序表（顺序存储）

链表（链式存储）

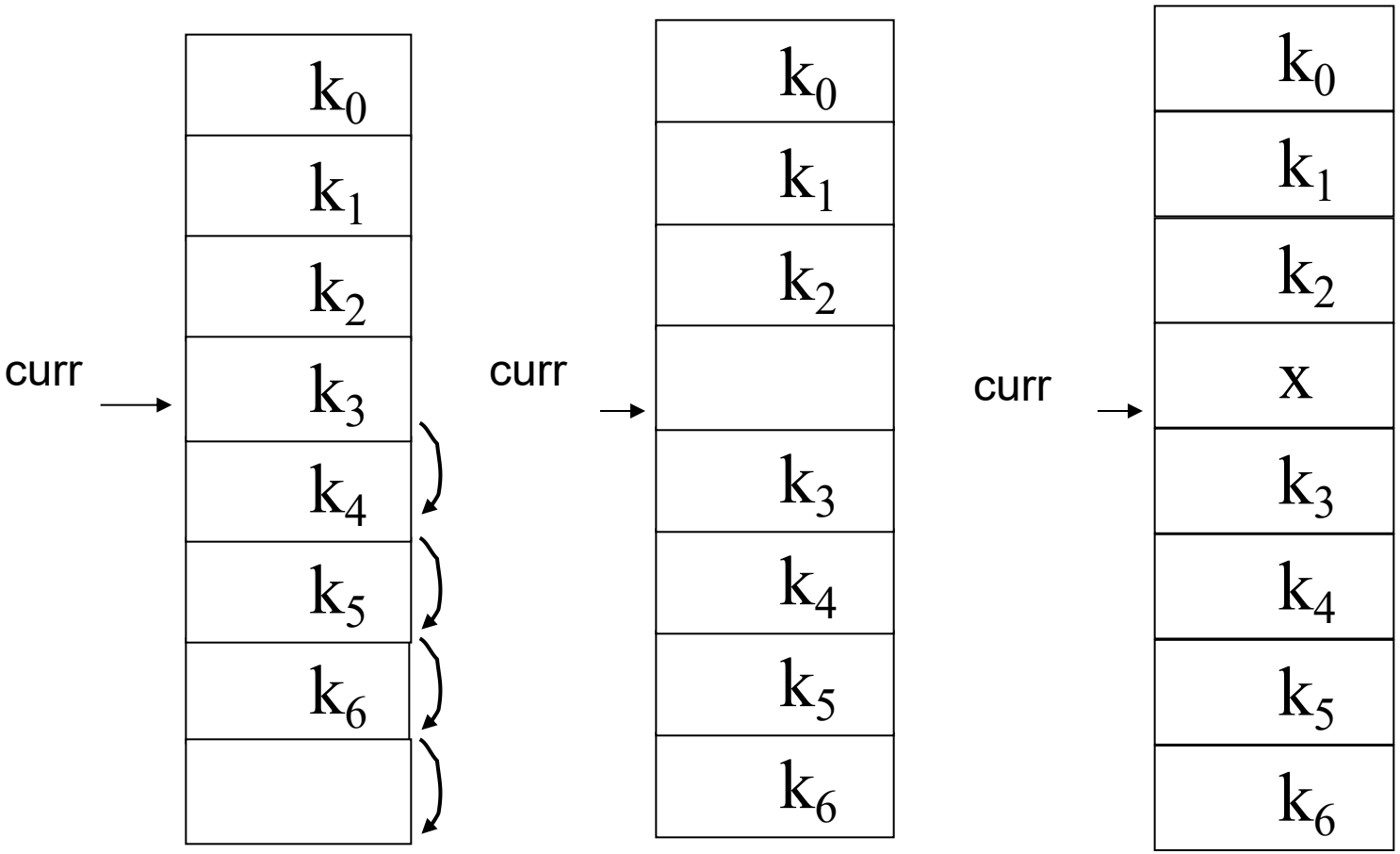
4.1.1 顺序表的实现

- 采用连续的存储单元依次存储线性表中各元素。
- 这种存储方式称为顺序存储方式，按这种存储方式所得到的线性表叫顺序表。
- 顺序表具有特点：逻辑上相邻的元素在物理上一定相邻。

顺序表类的定义

```
template <typename E>
class AList :public List<E> {
private:
  int maxSize;     //  最大长度
  int listSize;   //线性表的实际长度
  int curr; //栅栏位置，指示当前元素位置
  E* listArray; //存储数据元素的数组
public:
…
```

顺序表的插入图示
——insert（x）

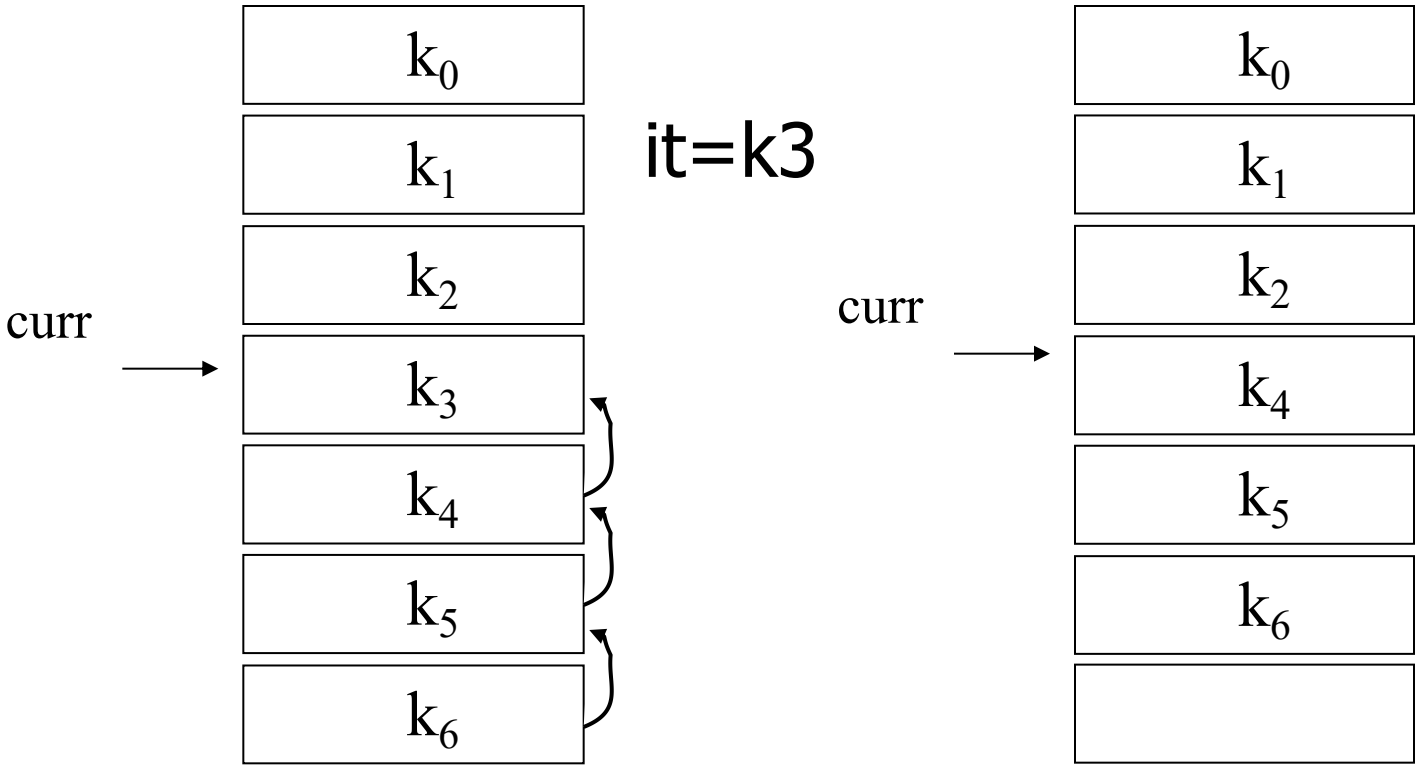


顺序表类成员函数的实现

- 顺序表结点插入操作

```
void insert(const E& it)
{  Assert (listSize<maxSize,"List capacity exceeded") ;
        //通过断言进行边界检查，当条件为假时输出提示信息，参见附录
   for (int i=listSize;i>curr;i--)    //移
        listArray[i]=listArray[i-1];
   listArray[curr]=it;//改
   listSize++;//加
}
```

顺序表的删除图示



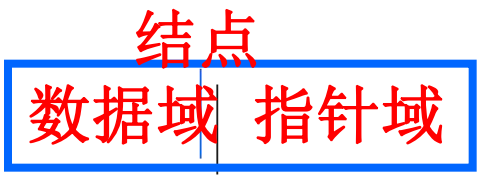
顺序表结点删除操作

```
E remove()
 {  Assert ((curr>=0) &&(curr<listSize), “No element”);
        //边界检查
   E it=listArray[curr]; //存
   for (int i=curr;i<listSize-1;i++)    //移
        listArray[i]=listArray[i+1];
   listSize--;//减
   return it;//返
 }
```

链表

特点：

- 用一组任意的存储单元存储线性表的数据元素
- 利用指针实现了用不相邻的存储单元存放逻辑上相邻的元素
- 每个数据元素a_i，除存储本身信息外，还需存储其直接后继的信息
- 结点
  - 数据域：元素本身信息
  - 指针域：指示直接后继的存储位置



链表实现——链表结点

```
template <typename E>
class Link {//链表节点
public:
  E element; // value for this node
  Link *next; // Pointer to next node in list
  Link(const E& elemval, Link* nextval=NULL)
   { element=elemval; next=nextval; }
  Link(link* nextval=NULL) { next=nextval; }
}
```

单链表类

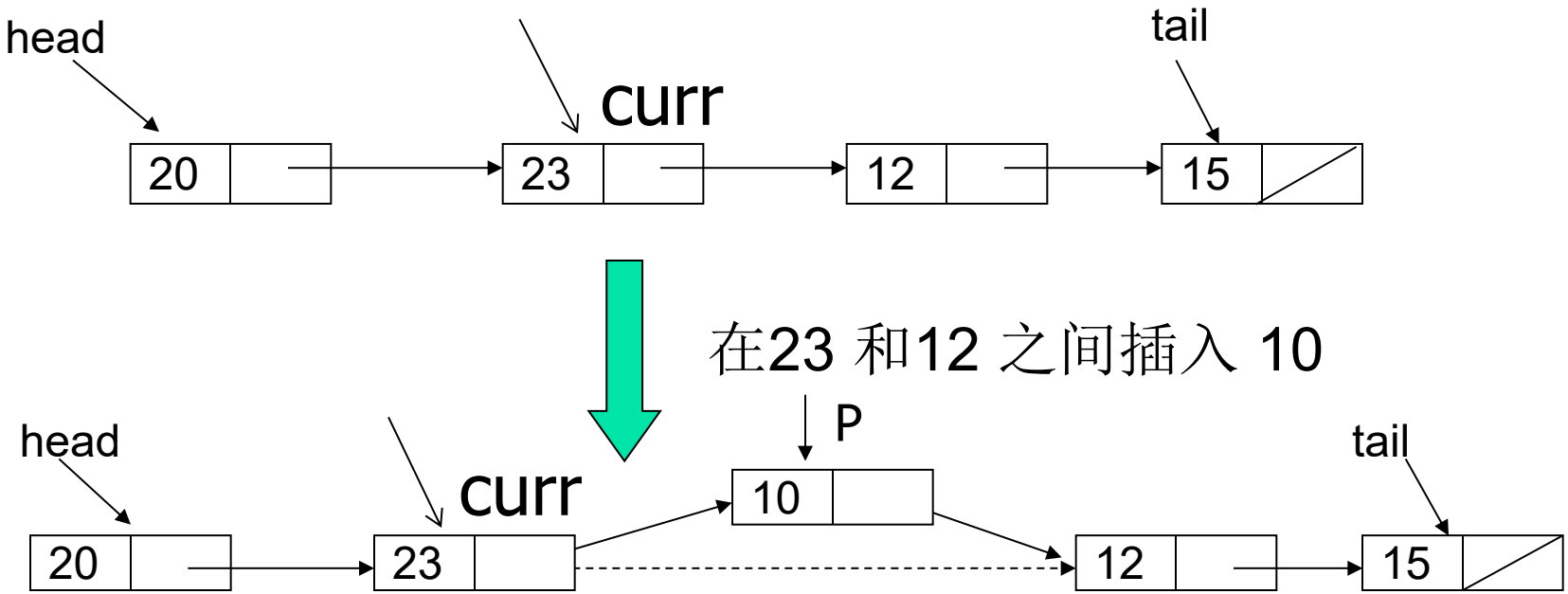
```
template <typename E>
class LList :public List<E> {
private:
  Link<E>* head;//头指针
  Link<E>* tail; //尾指针
  Link<E>* curr;//当前节点
    的前驱节点
  int cnt;//节点数
```

```
void init(){
  //初始化只有一个头节点的
  单链表
   curr=tail=head=new
  Link<E>;
    cnt=0;
  }
  void removeall(){
        //删除所有结点
    while (head!=NULL) {
     curr=head;
     head=head->next;
        //记住下一节点的位置
     delete curr;
    }//从头节点至尾节点逐一
  删除节点，释放空间    }
```

单链表类

```
public:
  LLlist(int size=DefaultListSize){init();}
  ~LList(){removeall();}
  void clear(){removeall();init();}
  bool insert(const E& it);
  bool append(const E& it);
  E remove();
  void moveToStart() {curr=head;}
  void moveToEnd()  {curr=tail;}
  void prev();
  void next();
  int  length() const {return cnt;}
  int  currPos() const;
  void moveToPos(int pos);
  const E& getValue() const;
  }
```

单链表的插入



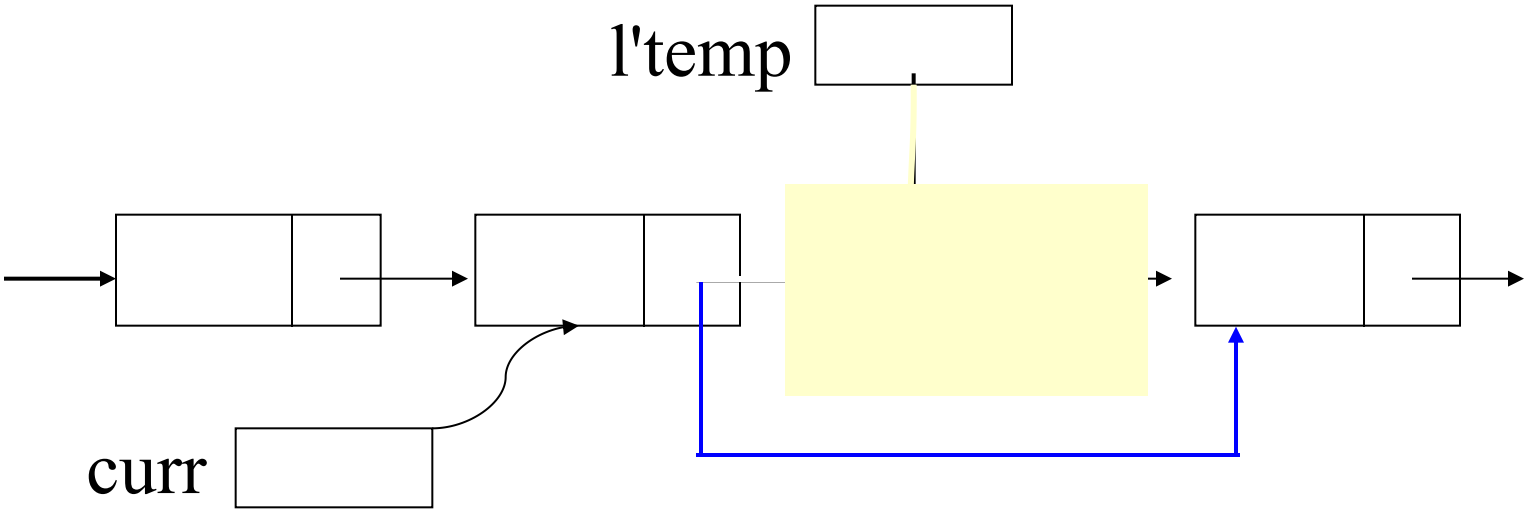
1. 创建新结点，用指针记录其位置 （新）
2. 新结点的后继指针指向当前结点 （插）
3. 当前结点的前驱结点的后继指针指向新结点 （改）

单链表的结点插入

```
void insert(const E& it) {
  curr->next=new link<E>(it, curr->next);
  if (tail == curr) tail =curr->next;//在表尾插入
  cnt++;
 }
```

- 创建新的结点并且赋给新值，其next指针指示当前节点。
  - new link<E>(it, curr->next);
- 当前结点元素前驱的next 域要指向新插入的结点。
  - curr->next=new link<E>(it, curr->next);

删除当前结点



ltemp = curr→next; //记录待删除结点位置

curr→ next = ltemp→ next; //修改前驱的后继为其后继

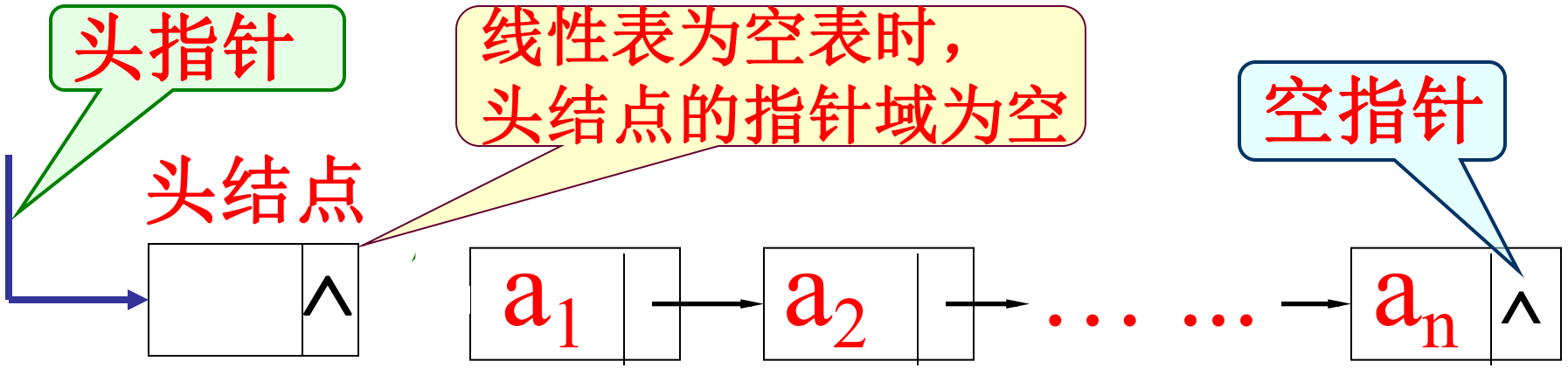
free(ltemp);

19

单链表结点的删除

```
E remove() {
  Assert(curr->next!=NULL,” No element”);
  E it =curr->next->element;//取出当前数据元素
  link<E>* ltemp =curr->next;//记下删除元素位置
  if (tail == ltemp) tail =curr;
     //删除最后一个数据元素，修改尾指针
  curr->next =curr->next->next;
  delete ltemp;
  cnt--;
  return it;
}
```

- curr指向待删除结点的前驱



以线性表中第一个数据元素 a_1 的存储地址作为线性表的地址，称作线性表的头指针。

有时为了操作方便，在第一个结点之前虚加一个“头结点”，以指向头结点的指针为链表的头指针。

单链表的建立

逆序输入 n 个数据元素的值，建立带头结点的单链表。

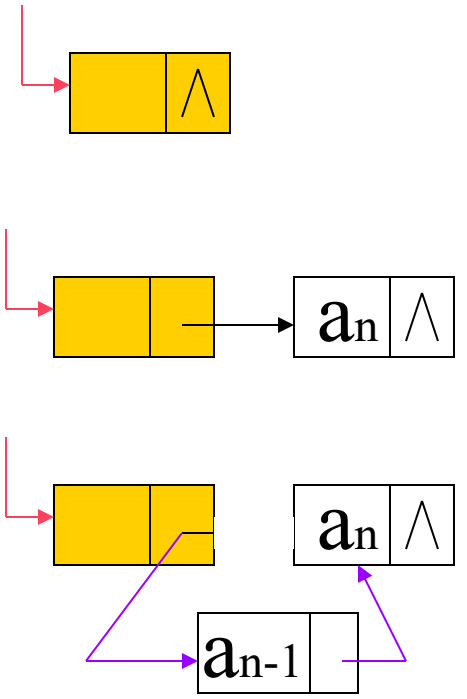
操作步骤：

一、建立一个“空表”；

二、输入数据元素a_n，
建立结点并插入；

三、输入数据元素a_{n-1}，
建立结点并插入；

四、依次类推，直至输入a_1为止。



思考：为什么要逆序来建立单链表？

应用举例：一元多项式的表示及相加

一般形式：

$$p_n(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_n x^n$$

在计算机中，可以用一个线性表来表示：

$P = (p_0, p_1, \cdots, p_n)$

空间开销：O（n）

稀疏形式：

$$S(x) = 1 + 3x^{10000} - 2x^{20000}$$

按一般方法表示，将造成大量空间浪费

一元稀疏多项式的表示及相加

一般情况下的一元稀疏多项式可写成

$$P_n(x) = p_1x^{e1} + p_2x^{e2} + \text{---} + p_mx^{em}$$

其中：p_i 是指数为e_i 的项的非零系数，

$$0 \leq e_1 < e_2 < \text{---} < e_m = n$$

可以下列线性表表示：

$$((p_1, e_1), (p_2, e_2), \text{---}, (p_m, e_m))$$

抽象数据类型一元多项式的定义

ADT Polynomial {

数据对象：

D＝{ a_i | a_i ∈TermSet, i=1,2,...,m, m≥0

TermSet 中的每个元素包含一个

表示系数的实数和表示指数的整数 **}**

数据关系：

$\mathbf{R_1}$＝{ <a_{i-1} ,a_i >|a_{i-1} ,a_i∈D, i=2,...,n

且$\mathbf{a_{i-1}}$中的指数值＜$\mathbf{a_i}$中的指数值 **}**

基本操作

CreatPolyn (&P, m)

操作结果：输入 m 项的系数和指数，建立一元多项式 P。

DestroyPolyn (&P)

初始条件：一元多项式 P 已存在。

操作结果：销毁一元多项式 P。

PrintPolyn (&P)

初始条件：一元多项式 P 已存在。

操作结果：打印输出一元多项式 P。

PolynLength(P)

初始条件：一元多项式 P 已存在。

操作结果：返回一元多项式 P 中的项数。

AddPolyn (&Pa, &Pb)

初始条件：一元多项式 Pa 和 Pb 已存在。

操作结果：完成多项式相加运算，即：

Pa = Pa＋Pb，并销毁一元多项式 Pb。

SubtractPolyn (&Pa, &Pb)

… …

} ADT Polynomial

一元多项式的实现

typedef OrderedLinkList polynomial;

// 用带表头结点的有序链表表示多项式

结点的数据元素类型定义为:

```
typedef struct {      // 项的表示
    float  coef;      // 系数
    int    expn;      // 指数
} term, ElemType;
```

```
Status CreatPolyn ( polynomail &P, int m ) {
  // 输入m项的系数和指数，建立表示一元多项式的有序链表P
    InitList (P); e.coef = 0.0;  e.expn = -1;
    SetCurElem (h, e);  // 设置头结点的数据元素
    for ( i=1; i<=m; ++i ) {  // 依次输入 m 个非零项
        scanf (e.coef, e.expn);
        if (!LocateElem ( P, e, (*cmp)()) ) //新元素
            if ( !InsAfter ( P, e ) )   return ERROR;
    }
    return OK;
} // CreatPolyn
```

注意：1. 输入次序不限；

2. 指数相同的项只能输入一次。

如何实现线性链表表示的多项式的加法运算?

一元多项式的运算法则：

指数相同的项，对应系数相加，若其和不为**0**，则构成"和多项式"的一项；

对于两个一元多项式中所有指数不同的项，则分别复抄到"和多项式"中。

```
Status AddPolyn ( polynomial &Pc,
                  polynomial &Pa, polynomial &Pb) {
  // 利用两个多项式的结点构成“和多项式” Pc = Pa + Pb
                                  … …
  if (DelAfter(Pa, e1))  a=e1.expn   else  a=MAXE;
  if (DelAfter(Pb, e2))  b=e2.expn   else  b=MAXE;
  while (!(a=MAXE && b=MAXE)) {//不是两个多项式都到最后
                  … …
   }                 … …
} // AddPolyn
```

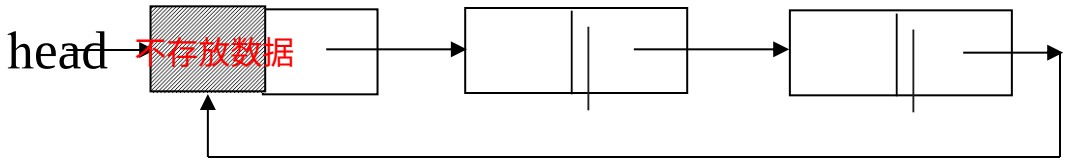
```
switch (*cmp(e1, e2)) {
   case -1: {  // 多项式PA中当前结点的指数值小
                        … …       break;  }
   case 0: {   //  两者的指数值相等
      e1.coef= a.coef + b.coef ;
      if (e.coef != 0.0 ) InsAfter(Pc, e1);
                  … …                 break;
   }
   case 1: {  //多项式PB中当前结点的指数值小
                        … …       break;  }
}
```

ADT的构成要素【多选题】

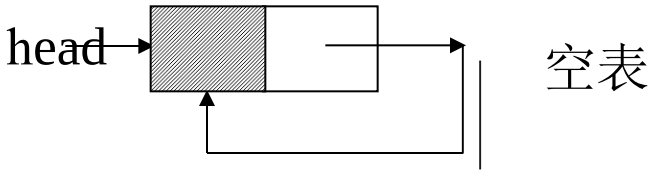
抽象数据结构（ADT）包含哪几个部分
A、
数据对象
B、
数据关系
C、
基本操作

循环链表

- 循环链表是表中最后一个结点的指针指向头结点，使链表构成环状。



- 特点：从表中任一结点出发均可找到表中其他结点，提高查找效率。
- 操作与单链表基本一致，表尾和表空的判断条件不同
  - 表尾的判定条件不同
  - 单链表p：p->next=NULL
  - 循环链表p：p->next=head



  - 空表的条件呢？ 单链表p：head->next=NULL，
  - 循环链表p：head->next=head

双链表

- 双向链表是指在前驱和后继方向都能遍历的线性链表。
- 双向链表每个结点结构：

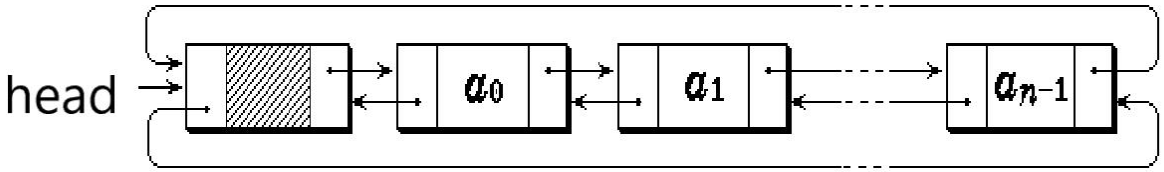
前驱方向 ← → 后继方向

- 双向链表通常采用带表头结点的循环链表形式。

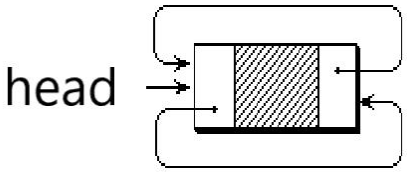
双链表

数据域为黑表示不存放数据

结点指针的指向

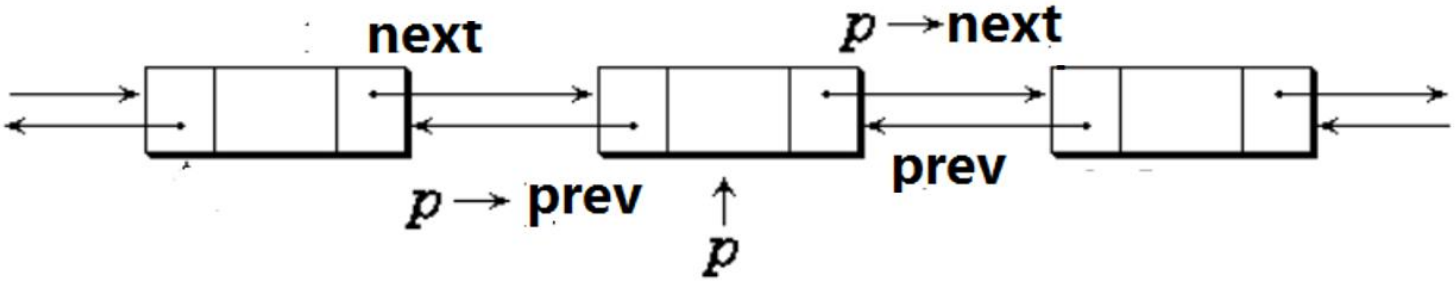


非空表



空表

空表 head->prev=head->next=head

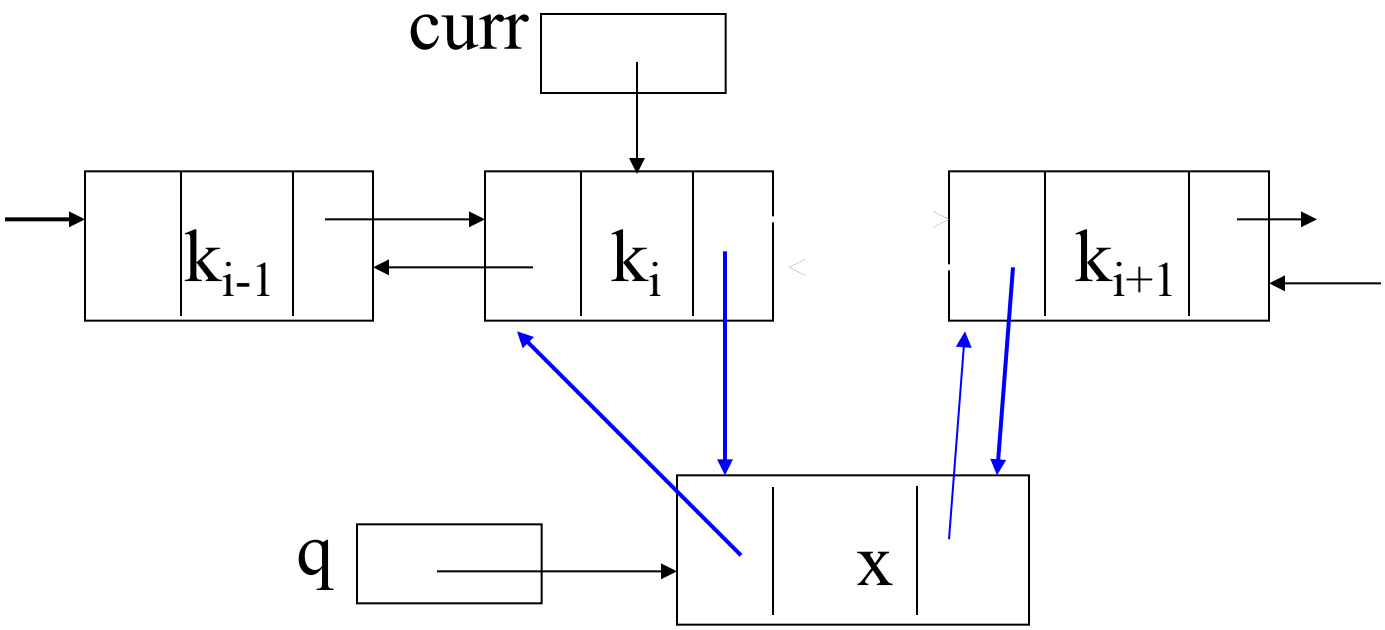


非空表 p = p→prev→next = p→next→prev

双链表结点

```
template <typename E> class link {
private:
  static Link<E> *freelist;   //指向可用空间表头
public:
  E element;
  Link* next;
  Link* prev;
  Link(const E& it, Link* prevp, Link* nextp)
  { element=it; prev=prevp; next=nextp; }
  Link(Link* prevp=NULL,Link* nextp=NULL)
  { prev=prevp;next=nextp; }
  void* operator new(size_t);
  void operator delete(void*);
}
```

双链表插入示意



(1) q=new link(x,curr,curr->next);

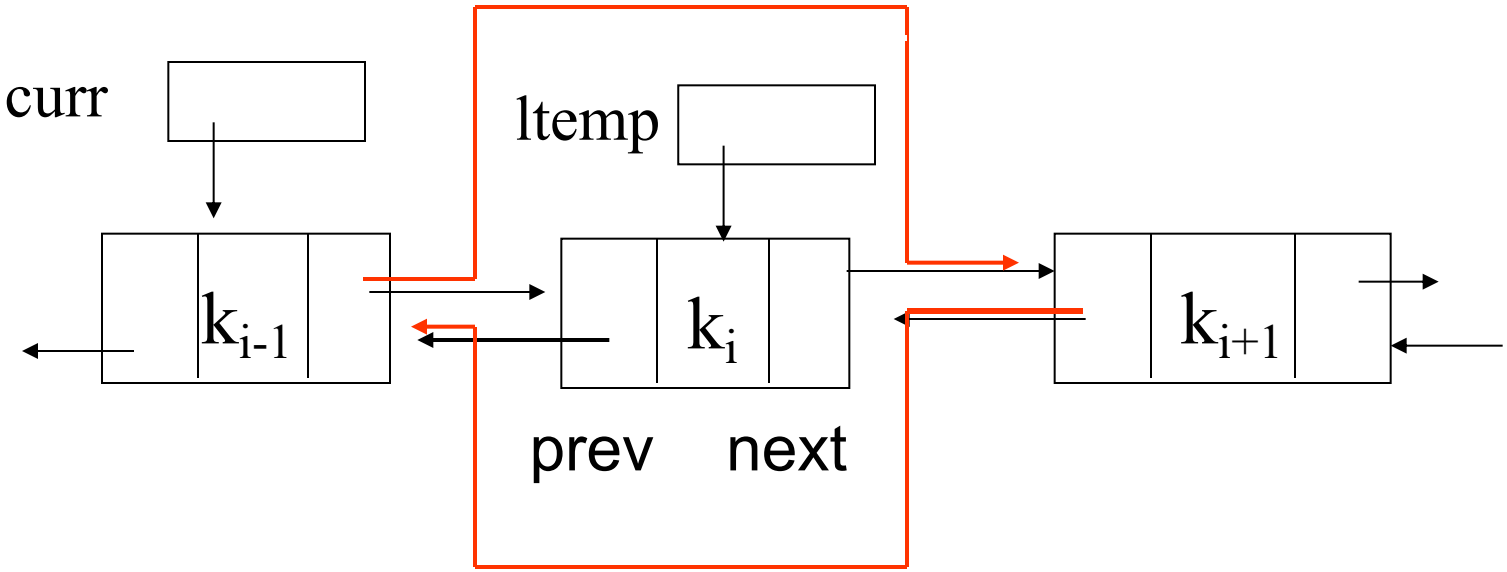
(2) curr →next → prev = q;

(3) curr → next = q;

双链表的插入

```
void insert(const E& it) {
  curr->next=curr->next->prev
          =new Link<E>(it,curr,curr->next);
  cnt++;
  }
```

双链表删除示意



ltemp= curr → next

curr→ next = ltemp→ next

curr →next → prev = ltemp→ prev

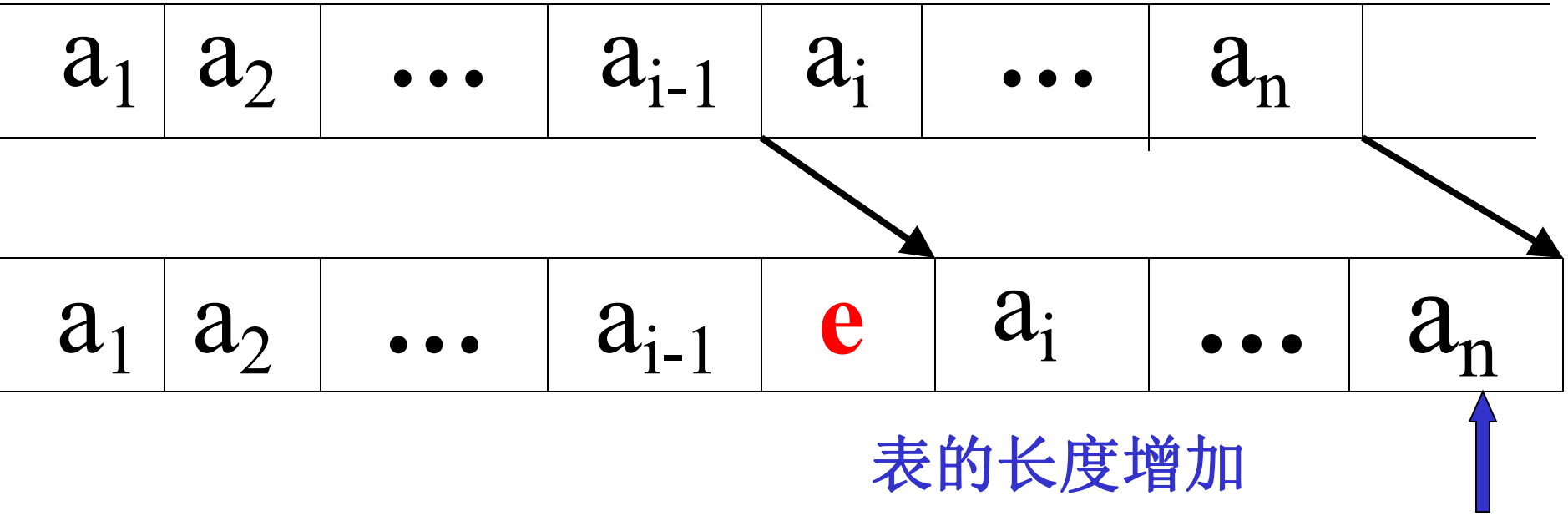
双链表的删除

```
E remove() {
  if (cnt==0||curr->next==NULL) return NULL;
          //空表或者无该结点
  E it =curr->next->element;
  link<E>* ltemp =curr->next;
  if（ltemp!=tail)      ltemp->next->prev=curr;//非尾结点
  curr->next= ltemp->next;
 delete ltemp;
  cnt--;
  return it;
}
```

线性表实现方式的比较——顺序表插入操作

$(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$ 改变为

$(a_1, \ldots, a_{i-1}, e, a_i, \ldots, a_n)$



考虑移动元素的平均情况

假设在第 i 个元素之前插入的概率为 P_i ，则在长度为 n 的线性表中插入一个元素所需移动元素次数的期望值为：

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

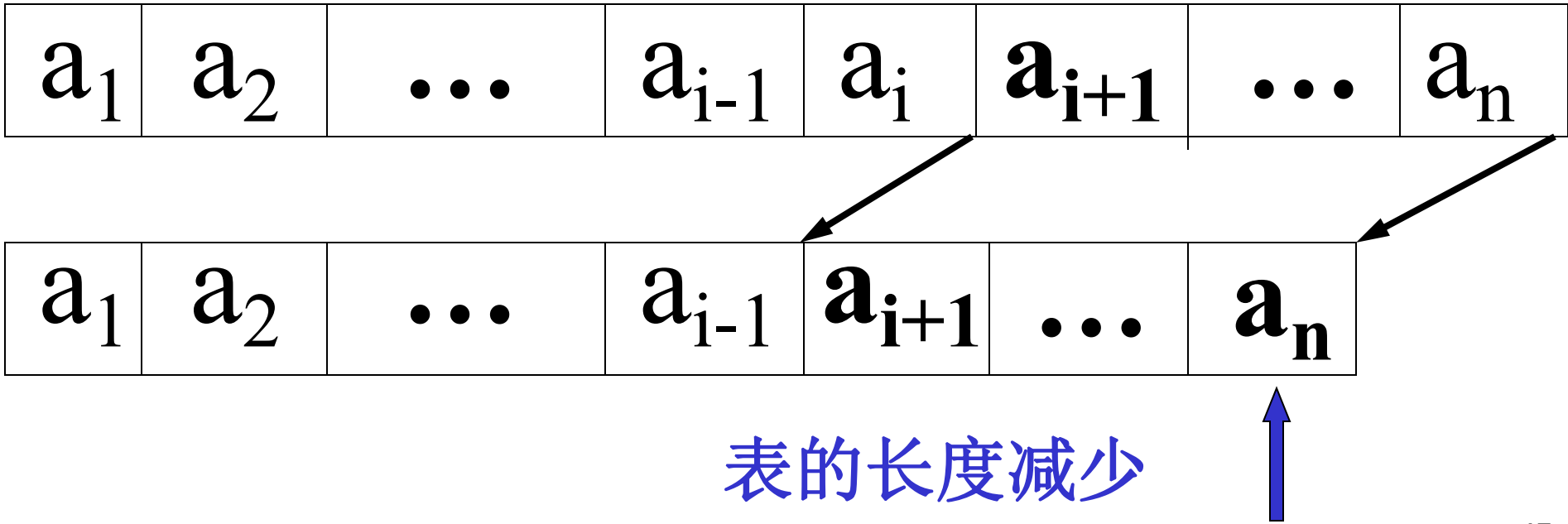
若假定在线性表中任何一个位置上进行插入的概率都是相等的，则移动元素的期望值为：

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

顺序表删除操作

$(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ 改变为

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



考虑移动元素的平均情况

假设删除第 i 个元素的概率为 q_i ，则在长度为 n 的线性表中删除一个元素所需移动元素次数的期望值为：

$$E_{dl} = \sum_{i=1}^{n} q_i (n - i)$$

若假定在线性表中任何一个位置上进行删除的概率都是相等的，则移动元素的期望值为：

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

线性表实现方法的比较

顺序表

- 插入、删除运算时间代价O(n)
- 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个元素删除运算时间代价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的临界值

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个字节)
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(按最大需求分配）
  - 链表的空间需求为n(P+E)（按需分配）
- 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

根据应用选择顺序表和链表

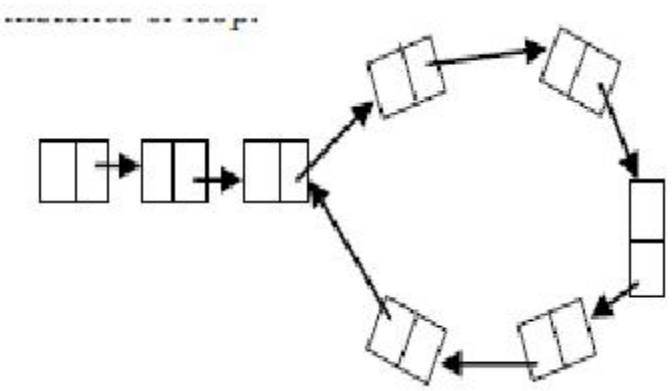
顺序表适用场景

- 结点总数目大概可以估计
- 线性表中结点比较稳定（插入删除操作少）
- $n>DE/(P+E)$

链表适用场景

- 结点数目无法预知
- 线性表中结点动态变化（插入删除多）
- $n<DE/(P+E)$

判定给定的链表是以NULL结尾，还是形成一个环。



蛮力法：例如考虑上面的链表，其中包含一个环。这个链表与常规链表的区别在于，其中有两个结点的后继结点是相同的。在常规链表中是不存在环的，每个结点的后继结点是唯一的。换言之，链表中若出现（多个结点的）后继指针重复，就表明存在环。

判定给定的链表是以NULL结尾，还是形成一个环。

- Floyd环判定算法：使用了两个在链表中具有不同移动速度的指针。一旦它们进入环便会相遇，即表示存在环。

```
boolean DoesLinkedListContainsLoop(ListNode head) {
if (head == NULL ) return FALSE;
ListNode slowPtr = head, fastPtr = head;
while (fastPtr.getNext()!= null && fastPtr.getNext().getNext()!= null ) {
    slowPtr = slowPtr.getNext();
      fastPtr = fastPtr.getNext().getNext();
      if ( slowPtr == fastPtr ) return TRUE;
    }
return FALSE;
}
```

看申请了多少空间

时间复杂度为**O(*n*)**，空间复杂度为**O(1)**。

实验

- 第4周前完成实验一准备，第4周进行测试
- 阅读课程慕课1.2实验指南
- 资料中《2023春《数据结构与算法》课程实验指导书》和《2023春《数据结构与算法》课程实验安排及评分说明》