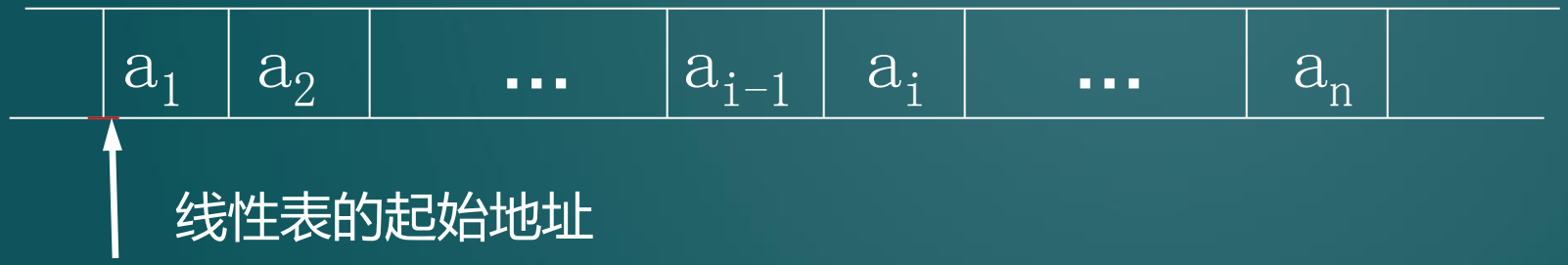
2.2 线性表的顺序表示和实现

顺序映象：以 x 的存储位置和 y 的存储位置之间某种关系表示逻辑关系<x,y>。

线性表的顺序表示指的是用一组地址连续的存储单元依次存放线性表的数据元素。



以“**存储位置相邻**”表示有序对<a_{i-1}，a_i>

即：LOC（a_i）= LOC（a_{i-1}）+ C

其中：C 是一个数据元素所占存储量

所有数据元素的存储位置均取决于第一个数据元素的存储位置。即：

LOC（a_i）= **LOC（a_1）** +（i-1）×C

↑ **基地址**

顺序映像的 C 语言描述

```
#define  LIST_INIT_SIZE   100
        // 线性表存储空间的初始分配量
#define  LISTINCREMENT  10
        // 线性表存储空间的分配增量

typedef  struct {
    ElemType    *elem；     // 存储空间基址
    int              length；   // 当前长度
    int              listsize；   // 当前分配的存储容量
                  //（ 以sizeof（ElemType）为单位）
  } SqList;  // 俗称 顺序表
```

线性表的基本操作在顺序表中的实现：

InitList（&L）　// 结构初始化

LocateElem（L, e, compare()）　// 查找

ListInsert（&L, i, e）　// 插入元素

ListDelete（&L, i）　// 删除元素

```
Status InitList_Sq( SqList& L ) {
  // 构造一个空的线性表
L.elem = (ElemType*) malloc ( LIST_
            INIT_SIZE*sizeof (ElemType) ) ;
L.length = 0 ;                  //空表长度为0
L.listsize = LIST_INIT_SIZE   //初始存储容量
} // InitList_Sq
```

算法**时间复杂度**：**O(1)**

算法：2.3

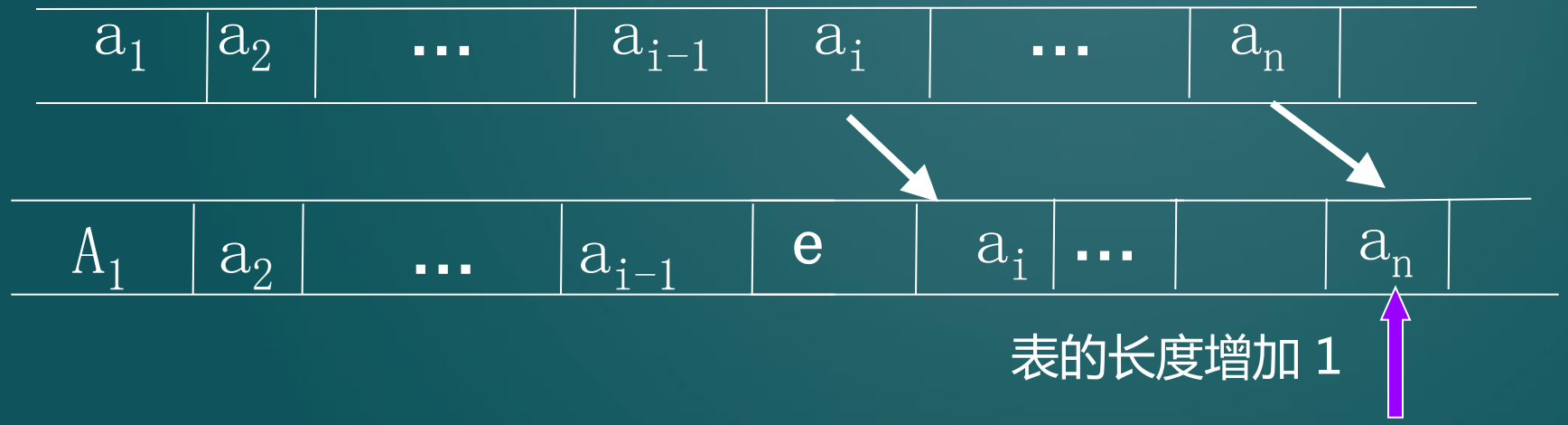
线性表操作：**ListInsert(&L, i, e)**的实现：

首先分析：插入元素时，线性表的逻辑结构发生什么变化？

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

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

$\langle a_{i-1}, a_i \rangle$ → $\langle a_{i-1}, e \rangle, \langle e, a_i \rangle$



```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
 // 在顺序表L的第 i 个元素之前插入新的元素e，
// i 的合法范围为  1≤i≤L.length+1
      ……//见下页//
 q = &(L.elem[i-1])；              // q 指示插入位置
 for (p = &(L.elem[L.length-1])； p >= q；  --p)
    *(p+1) = *p；         // 插入位置及之后的元素右移
 *q = e；         // 插入e
 ++L.length；    // 表长增1
 return OK；
} // ListInsert_Sq
```

```
if (i < 1 || i > L.length+1) return ERROR；
                    // 插入位置不合法
if (L.length >= L.listsize) {
                    // 当前存储空间已满，增加分配
    newbase = (ElemType *)realloc(L.elem,
        (L.listsize+LISTINCREMENT)*sizeof (ElemType))；
    if (!newbase) exit(OVERFLOW)；
                    // 存储分配失败
    L.elem = newbase；                  // 新基址
    L.listsize += LISTINCREMENT；// 增加存储容量
}
```

算法时间复杂度为:O(ListLength(L))

算法：2.4

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：

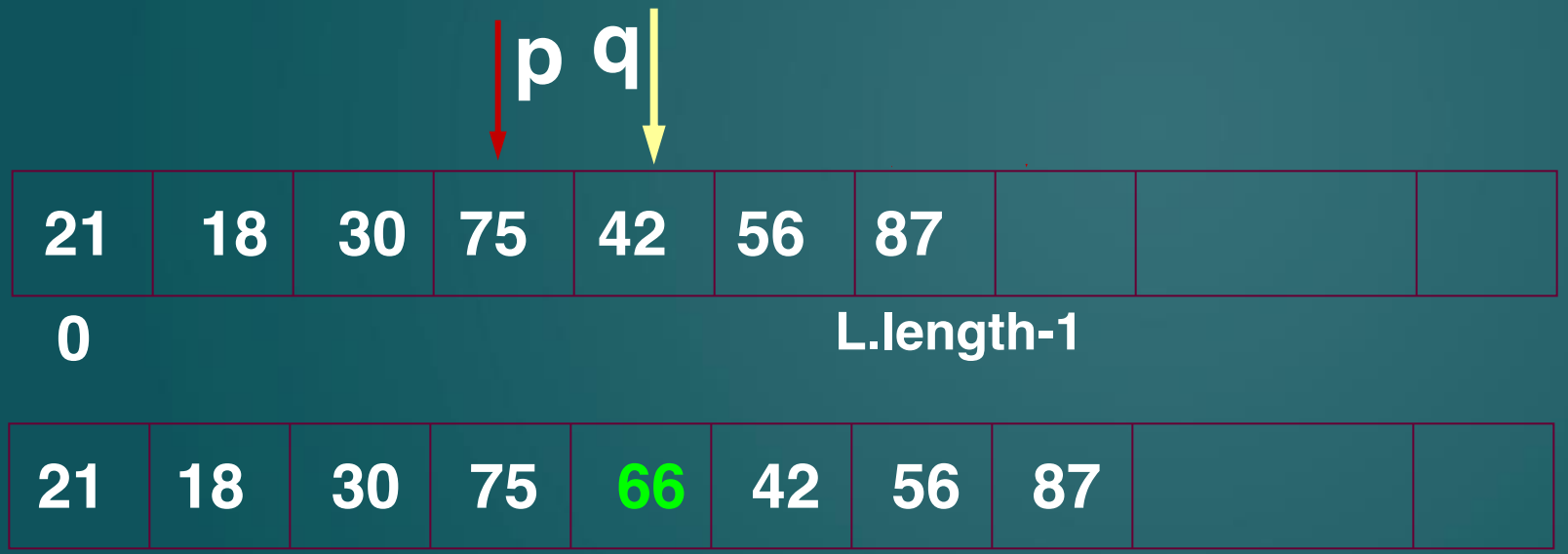
$$E_{is}=\sum_{i=1}^{n+1} p_i(n-i+1)$$

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：

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

例如：ListInsert_Sq (L, 5, 66)

```
q = &(L.elem[i-1]) ;       // q 指示插入位置
for (p = &(L.elem[L.length-1]); p >= q; - -p)
    *(p+1) = *p ;
```



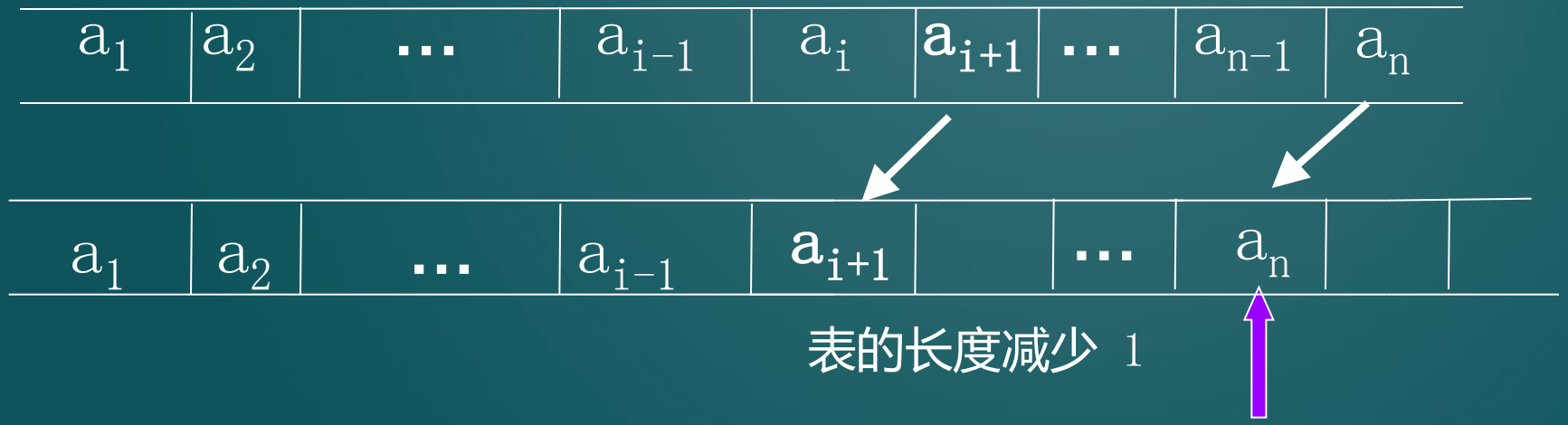
线性表操作: ListDelete(&L, i, &e)的实现：

首先分析：删除元素时，线性表的逻辑结构发生什么变化？

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

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

$\langle a_{i-1}, a_i \rangle, \langle a_i, a_{i+1} \rangle \Rightarrow \langle a_{i-1}, a_{i+1} \rangle$



```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    // 在顺序表L中删除第 i 个元素，并用e返回其值，
    // i 的合法范围为 1≤i≤L.length
    if ((i < 1) || (i > L.length))  return ERROR；
                    // i值不合法
    p = &(L.elem[i-1])；    // p 为被删除元素的位置
    e = *p;                 // 被删除元素的值赋给 e
    q = L.elem+L.length-1;  // 表尾元素的位置
    for (++p；  p <= q；  ++p)  *(p-1) = *p；
            // 被删除元素之后的元素左移
    --L.length   // 表长减1
    return OK；
} // ListDelete_Sq
```

算法时间复杂度为:O（ ListLength(L) ）

算法：2.5

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：

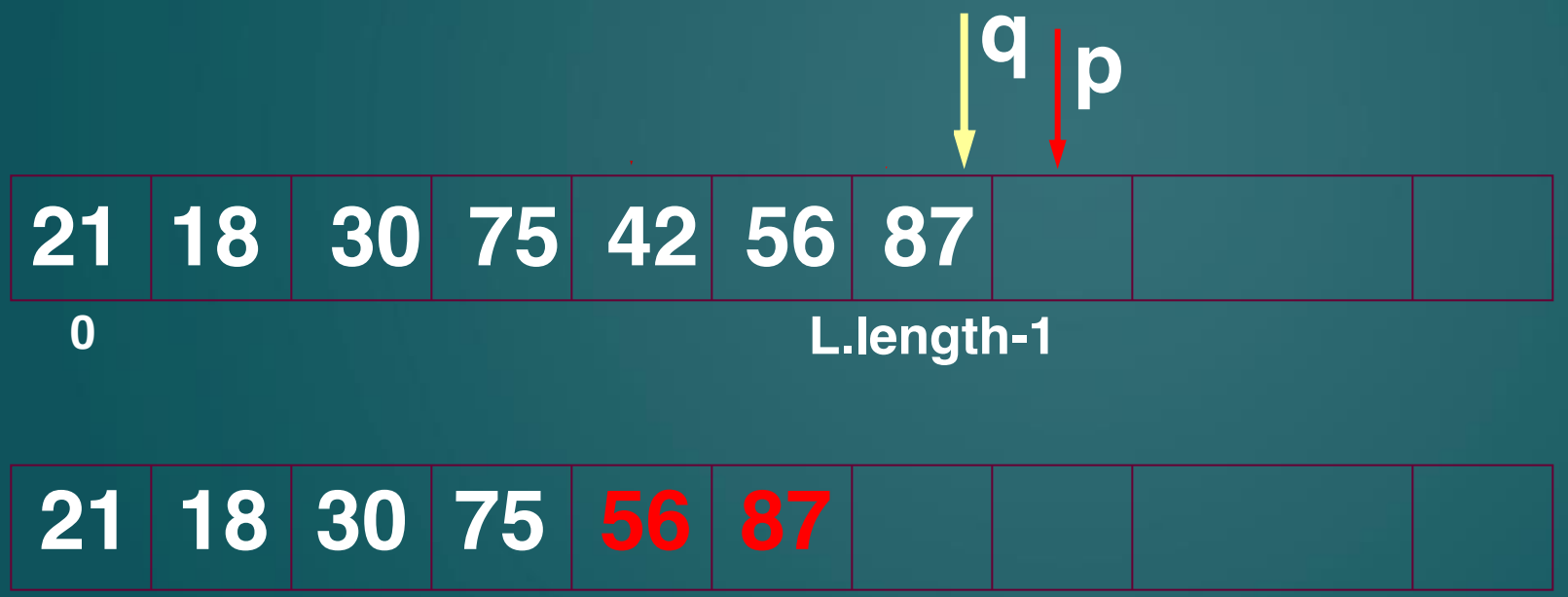
$$E_{dl}=\sum_{i=1}^{n} q_i(n-i)$$

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：

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

例如：ListDelete_Sq(L, 5, e)

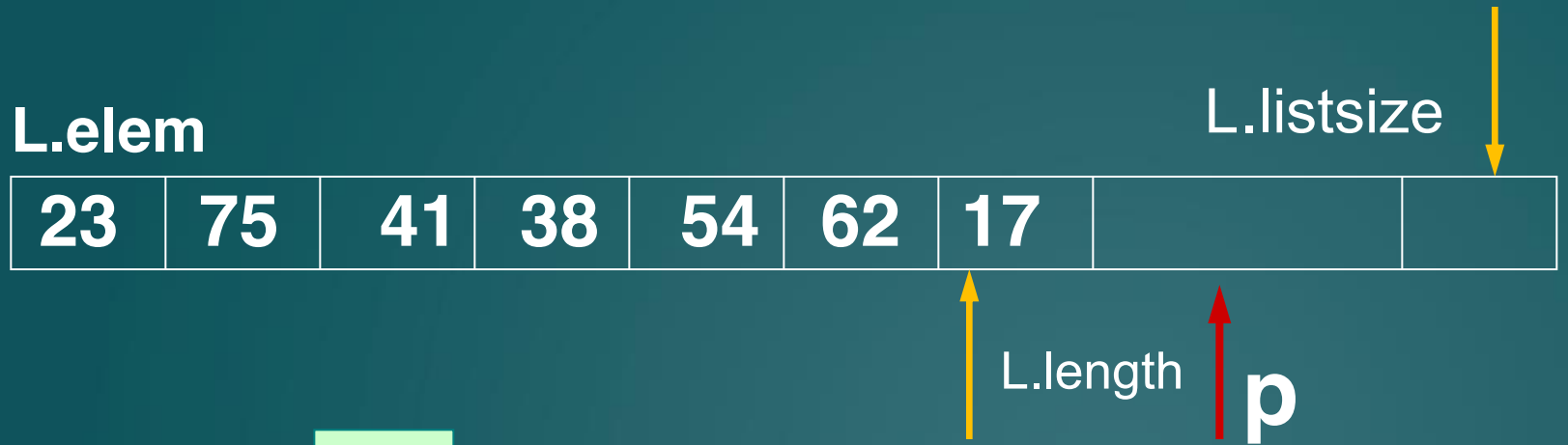
```
p = &(L.elem[i-1]);
q = L.elem+L.length-1;
for (++p; p <= q; ++p)  *(p-1) = *p;
```



讨论例2.1和例2.2在顺序存储结构的线性表中的实现方法和时间复杂度的分析。

算法2.1的执行时间主要取决于查找函数LocateElem的执行时间。在顺序表L查访是否存在和e相同的数据元素的最简便的方法是，令e和L中的数据元素逐个比较之，如算法2.6所示。从2.6中可见，基本操作是“进行两个元素之间的比较”，若L中存在和e相同的数据元素a_i，则比较次数为i(1≤i≤L.length),否则为L.length，即算法LocateElem_Sq的时间复杂度为O（L.length）。由此，对于顺序表La和Lb而言，union的时间复杂度为O（La.length×Lb.length）。

例如：顺序表



i 8 1

e = 38 50

可见，基本操作是：将顺序表中的元素逐个和给定值 e 相比较。

```
int LocateElem_Sq(SqList L, ElemType e,
  Status (*compare)(ElemType, ElemType)) {
  // 在顺序表中查询第一个满足判定条件的数据元素，
  // 若存在，则返回它的位序，否则返回 0
    i = 1；          // i 的初值为第 1 元素的位序
    p = L.elem；     // p 的初值为第 1 元素的存储位置
    while (i <= L.length &&  !(*compare)(*p++, e))
++i；
      if (i <= L.length)  return i；
      else  return 0；
} // LocateElem_Sq
```

算法的时间复杂度为： O（ListLength(L)）

算法：2.6

```
void MergeList _Sq(SqList La, SqList Lb, SqList &Lc) {
  //已知线性表La 和Lb中数据元素按值非递减有序排列
  //归并La 和Lb得到新的线性表Lc，Lc中的数据元素也按值
 //非递减排列。
  pa = La.elem ; pb = Lb.elem ;
  Lc.listsize =  Lc.length =  La.length +
  Lb.length ;
  pc= Lc.elem = (ElemType*) malloc
  ( Lc.listsize*sizeof ( ElemType ))；
  if (! Lc.elem)exit(OVERFLOW)；    //存储分配失败
```

```
    pa_last = La.elem +  La.length-1 ;
   pb_last = Lb.elem +  Lb.length-1 ;
    while ( pa<= pa_last && pb<= pb_last ) {  //归并
       if ( *pa<= *pb )  *pc++ = *pa++;
        else *pc++ = *pb++;
    }
    while ( pa<= pa_last ) *pc++ = *pa++;
    //插入La的剩余元素
    while ( pb<= pb_last ) *pc++ = *pb++;
    //插入Lb的剩余元素
} // MergeList _Sq
```

算法：2.7