高级程序设计训练

SPT-02 线性表

本节内容

- 2.1 线性表的概念和特点
- 2.2 顺序表的结构体定义及基本操作算法
- 2.3 链表的结构体定义及基本操作算法
- 2.4 小结

2.4 线性表链式存储结构及操作

线性表的链式存储结构

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素，这组存储单元可以是连续的，也可以是不连续的。这就意味着，这些数据元素可以存在内存未被占用的任意位置（如图 3-6-1 所示）。

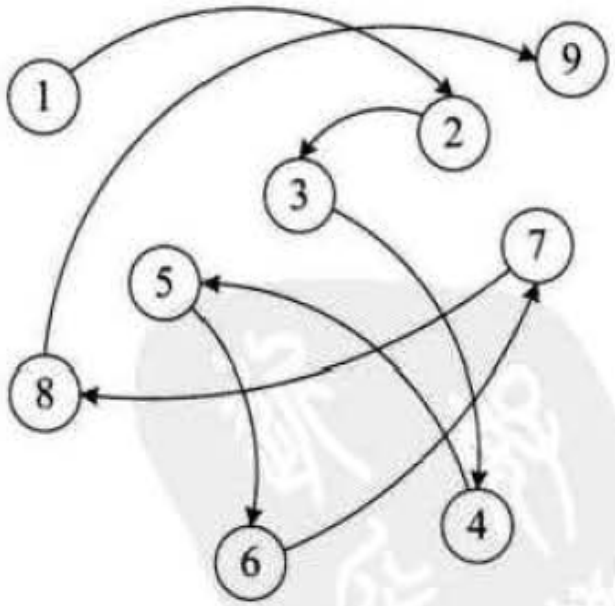
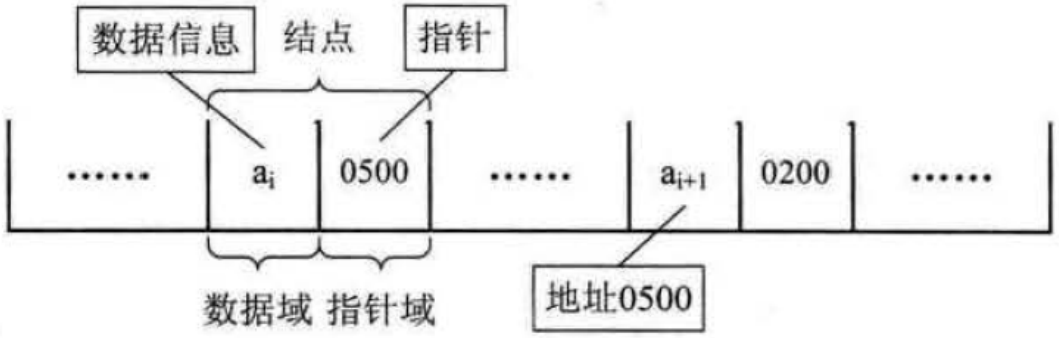


图 3-6-1

2.4 线性表链式存储结构及操作

- 为了表示每个数据元素a_i与其相邻元素如a_{i+1}之间的逻辑关系，还需要存储一个指示其相邻元素的信息。
- 也就是说线性表中每一个 数据元素的存储映像 包括两个部分：

  称为结点（Node）
  - 数据域：存储数据元素的部分；
  - 指针域：存储相邻元素位置的部分。其中存储的数据称为指针或链。
- 线性表的这种链式存储结构也称为链表。如果链表的每一个结点只包含一个指针域，就称为单链表或线性链表。



例 1 百家姓

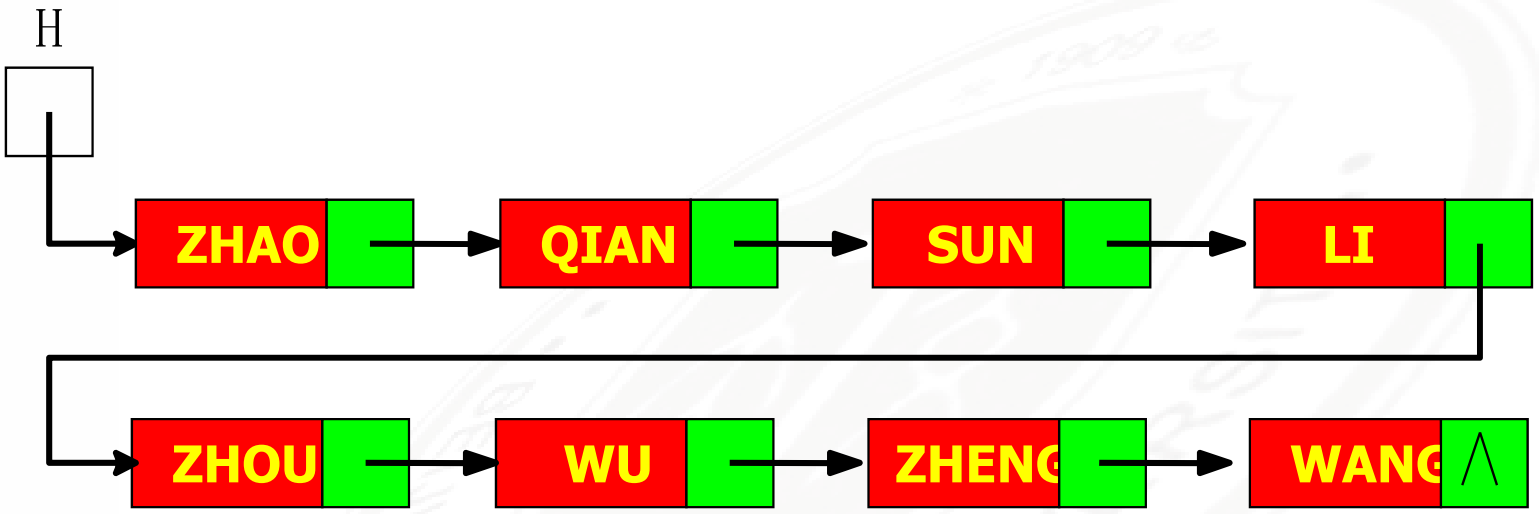
(ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)

头指针
31

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

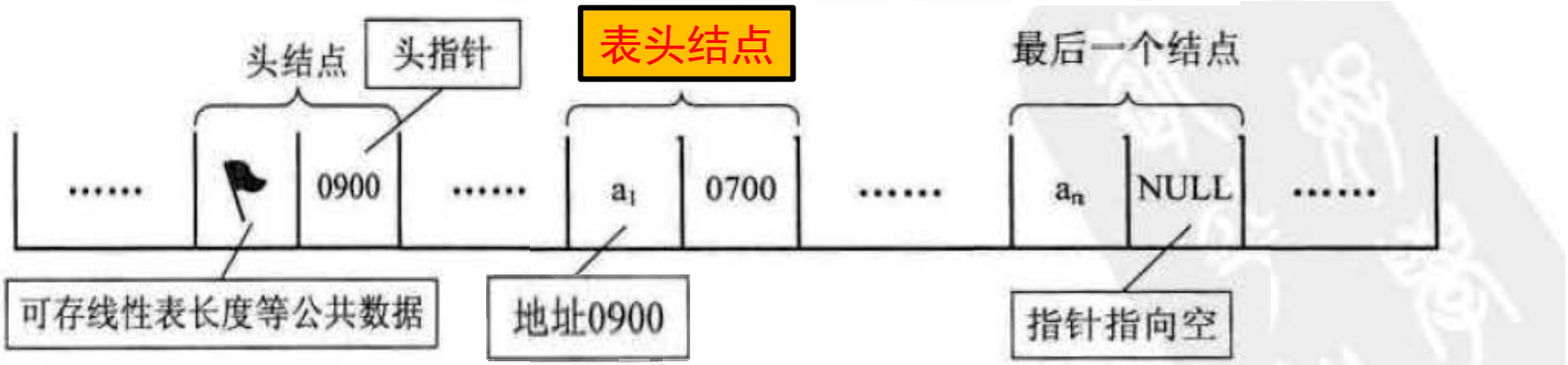
例 1 百家姓

- 线性链表的逻辑表示



2.4 线性表链式存储结构及操作

- 链表中：表头结点 的存储位置称为头指针。
- 为了操作方便，通常会在链表第一个结点之前再增加一个头结点。
  - 头结点的数据域可以不存放任何数据，也可以存放链表长度
  - 头结点的指针域存储第一个结点的位置



- 最后一个结点的直接后继不存在，其指针域为

2.4 线性表链式存储结构及操作

头指针

- 头指针是指向链表表头结点的指针，若链表有头结点，则是头结点指针域的指针。
- 头指针具有标志作用，所以常用头指针冠以链表名。

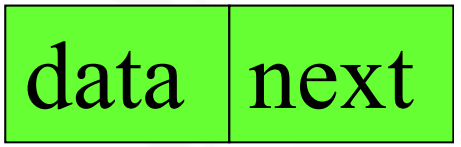
头结点

- 头结点是为了操作的统一和方便而设立的，放在第一元素的结点之前，其数据域一般无意义（也可以存放链表的长度）。
- 有了头结点，对在第一元素结点前插入结点和删除第一节点，其操作与其他结点的操作就统一了。
- 头结点不是链表必须要素。

2.4 线性表链式存储结构及操作

- 带头结点的单链表的存储结构定义

```
typedef struct Node{
    ElemType      data;      //数据域
    struct Node  *next;      //指针域
}LNode, *LinkList;
```




```
LinkList  p;

LNode  *p;
```

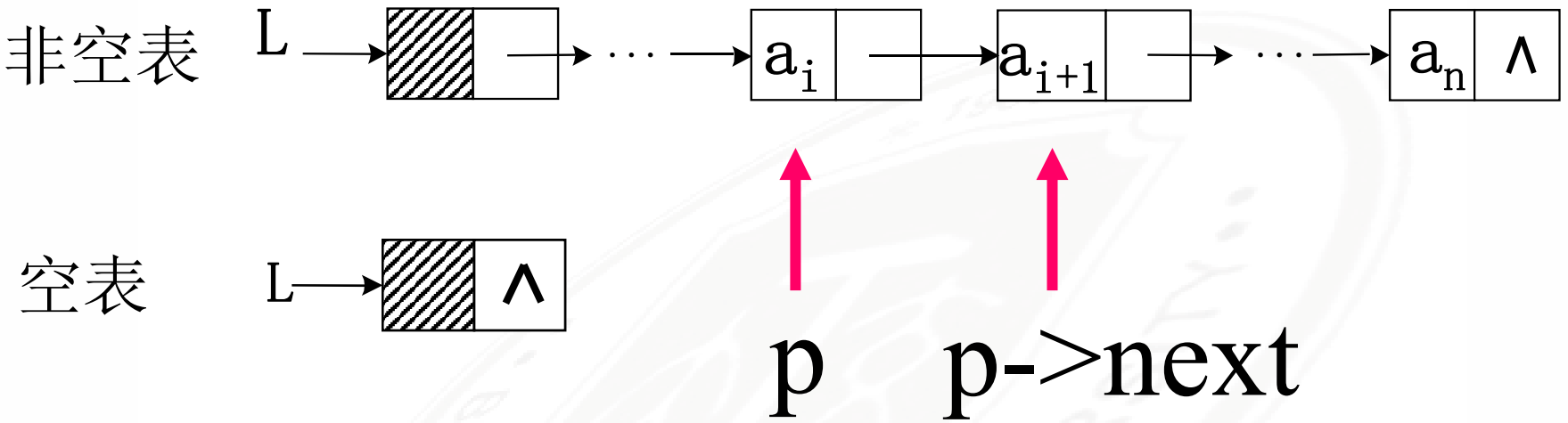
从顺序表到链表

- 请在顺序表结构体定义的基础上，编写链表的结构体定义。

```c
#include <stdlib.h>
#include <stdio.h>

#define OK 1
#define ERROR 0

typedef int Status;        /* Status是函数的类型,其值是函数结果状态代码，如OK等 */
typedef int ElemType;      /* ElemType类型根据实际情况而定，这里假设为int */

/*定义结点结构体*/
typedef struct Node
{
    ElemType data;         /*单链表中的数据域 */
    struct Node *next;     /*单链表的指针域*/
}LNode;
/*定义链表类型*/
typedef  LNode * LinkList;
```

2.4 线性表链式存储结构及操作

带头结点的单链表的存储结构定义（续）



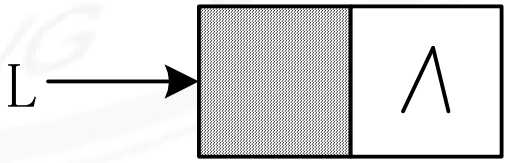
❖ **单链表的特点：**

- 若**p->data=a_i, 则p->next->data=a_{i+1}**
- 非随机存取结构 (顺序存取)

带头结点单链表的操作

(1) 初始化单链表

```
Status InitList_L(LinkList &L){
        L=(LinkList) malloc(sizeof(LNode));
        if(!L) return ERROR;
        L->next=NULL;
        return OK;
}// InitList_L
```



T(n)=O(1)

```
/*单链表的初始化*/
Status InitList_L(LinkList *L)
{
    (*L)=(LinkList)malloc(sizeof(LNode)); /* 产生头结点,
    if((*L) == NULL) /* 存储分配失败 */
        return ERROR;
    (*L)->next=NULL; /* 指针域为空 */

    return OK;
}
```

带头结点单链表的操作

(2) 单链表判空

```
Status ListEmpty_L(LinkList L){
    return (L->next==NULL);
}// ListEmpty_L
```

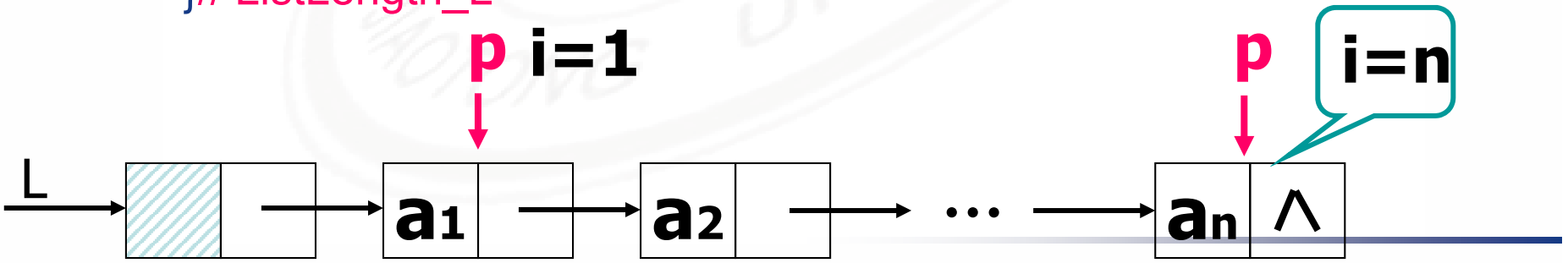
T(n)=O(1)

带头结点单链表的操作

(3) 求单链表的长度

```
int  ListLength_L(LinkList L){
    p=L->next;        //p指向第一个结点
    i=0;
    while(p){         //遍历单链表,统计结点数
                      i++;
                      p=p->next;
            }
    return i;
    }// ListLength_L
```

T(n)=O(n)



带头结点单链表的操作

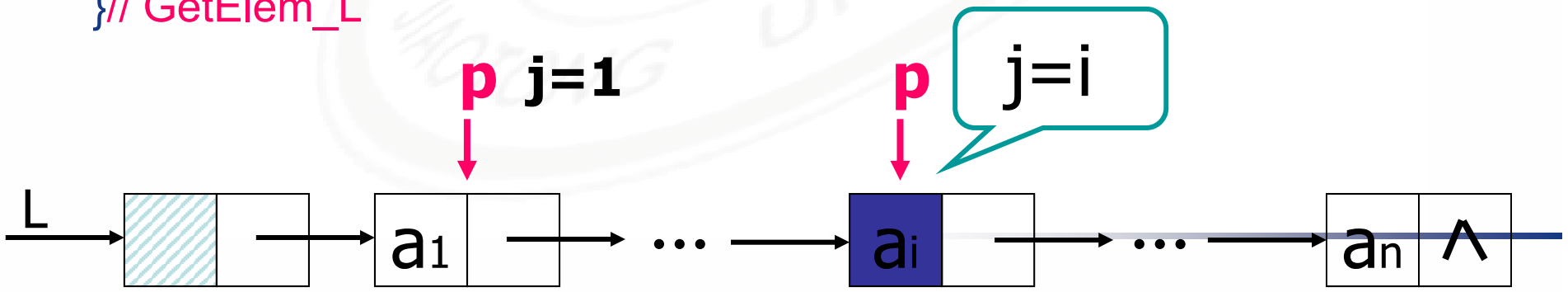
(4) 读取元素

```
Status GetElem_L (LinkList L, int  i,  Elemtype *e){
    p=L->next;                          //初始化，p指向第一个结点
```

获得链表第 i 个数据的算法思路：

1. 声明一个结点 p 指向链表第一个结点，初始化 j 从 1 开始；
2. 当 j<i 时，就遍历链表，让 p 的指针向后移动，不断指向下一结点，j 累加 1；
3. 若到链表末尾 p 为空，则说明第 i 个元素不存在；
4. 否则查找成功，返回结点 p 的数据。

```
    return OK;
}// GetElem_L
```



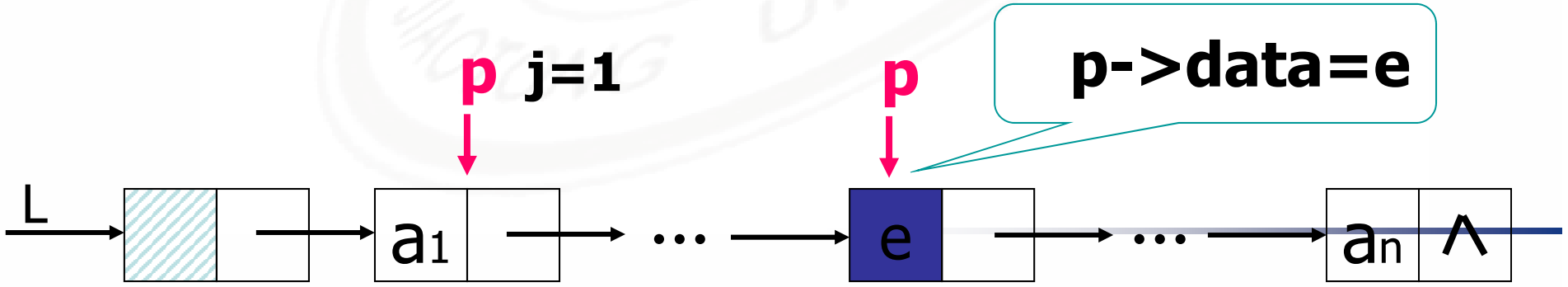
```
/* 初始条件：顺序线性表L已存在，1≤i≤ListLength(L) */
/* 操作结果：用e返回L中第i个数据元素的值 */
Status GetElem_L(LinkList L,int i,ElemType *e)
{
    int j=1;              /*  j为计数器 */
    LinkList p=NULL;          /* 声明一结点p */

    p = L->next;          /* 让p指向链表L的第一个结点 */

    while (p && j<i)  /* p不为空或者计数器j还没有等于i时，循环继续 */
    {
        p = p->next;  /* 让p指向下一个结点 */
        ++j;
    }

    if ( !p && j<i )
        return ERROR;  /*  第i个元素不存在 */

    *e = p->data;    /*  取第i个元素的数据 */

    return OK;
}
```

带头结点单链表的操作

(5) 找元素位置

```
int LocateElem_L (LinkList L, ElemType e) {
   p=L->next;                          //初始化,p指向第一个结点
   j=1;                                // j为计数器
   while(p && (p->data !=e)){          //顺指针向后查找
            p=p->next;  ++j;           //直到p指向第i个元素
   }
   if(p) return j;
   else return 0;
} //LocateElem_L;
```

T(n)=O(n)

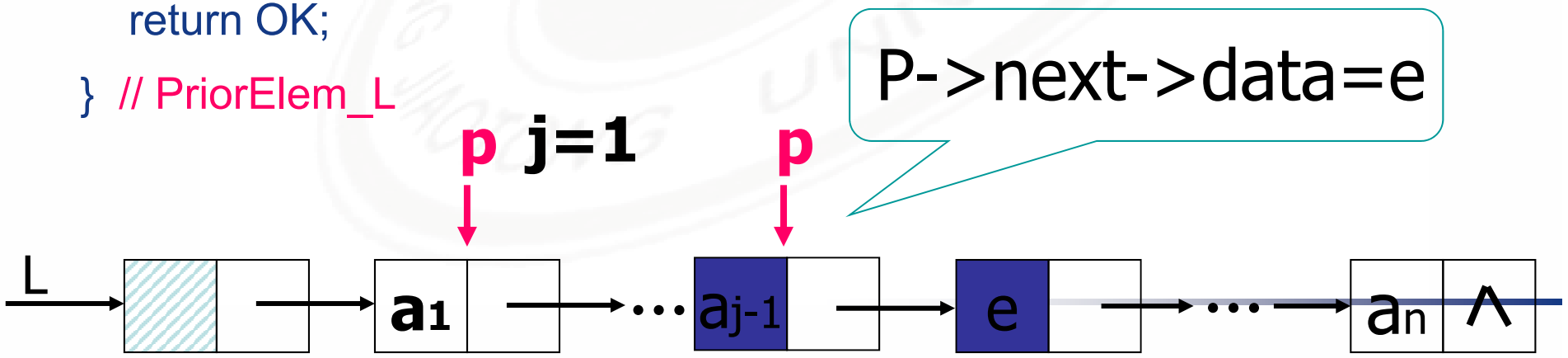


带头结点单链表的操作

(6) 找元素的前驱

```
Status PriorElem_L (LinkList L, ElemType cur_e){
    p=L; j=1;
    while(p->next && p->next->data != cur_e))
        p=p->next; j++;
    }
    if(!p->next || j==1)  return ERROR;
    pre_e=p->data;
    return OK;
}  // PriorElem_L
```

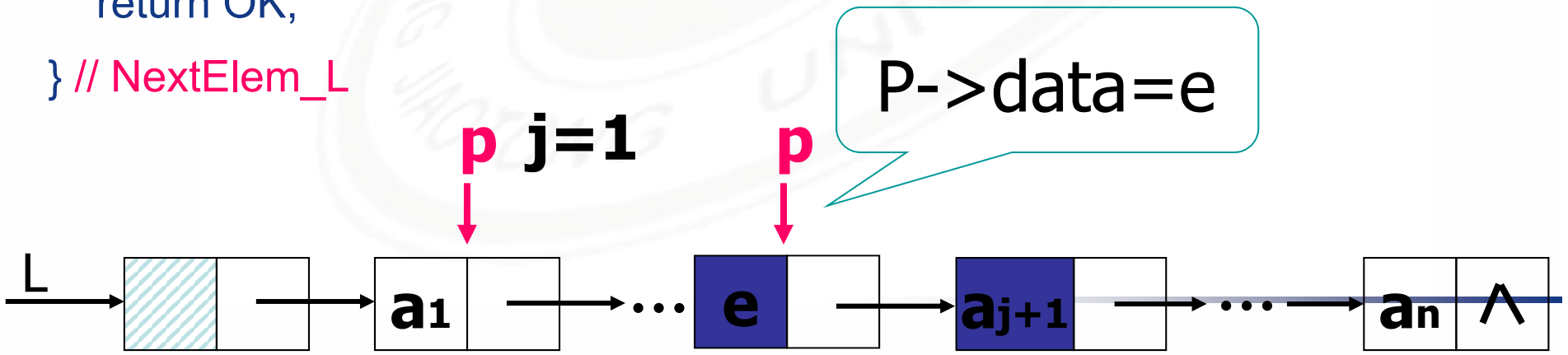
T(n)=O(n)



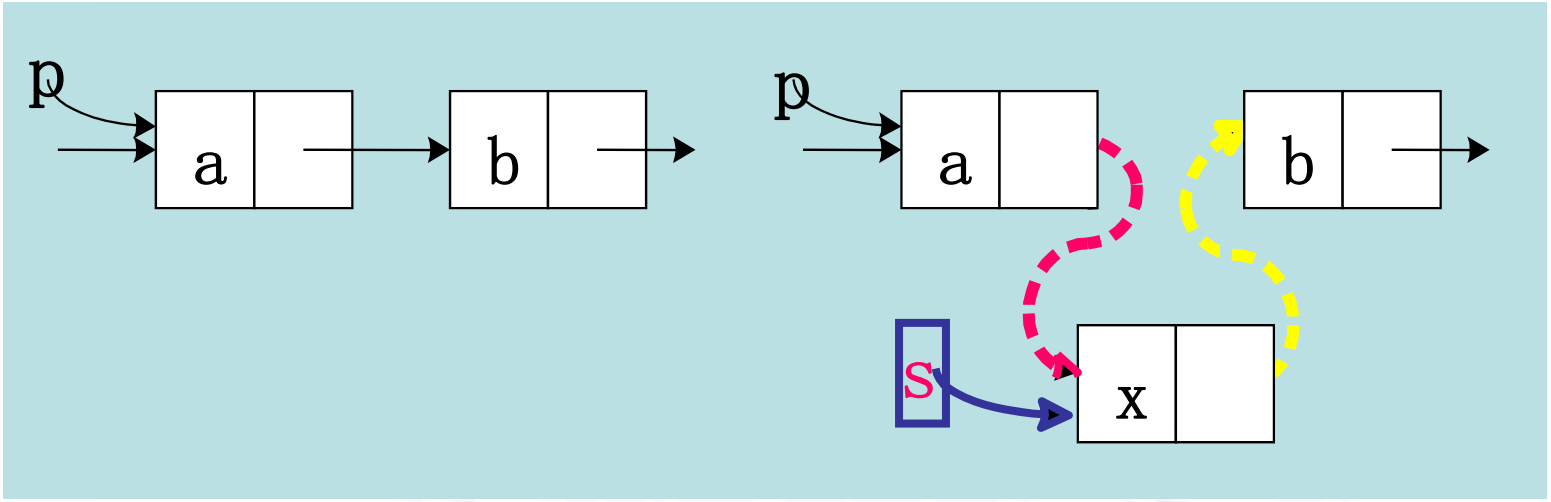
(7) 找元素的后继

```
Status NextElem_L (LinkList L, ElemType cur_e){
    p=L->next;
    while(p && p->data != cur_e))
        p=p->next;
    }
    if(!p || !p->next )  return ERROR;
    next_e=p->next->data;
    return OK;
} // NextElem_L
```

T(n)=O(n)



(8) 单链表的插入



（a）插入前　　（b）插入后

s->next=p->next;　　p->next=s

p->next=s;　　s->next=p->next　　X

带头结点单链表的操作

- (8) 单链表的插入

s->next=p->next;　　p->next=s

对于单链表的表头和表尾的特殊情况，操作是相同的，如图 3-8-4 所示。

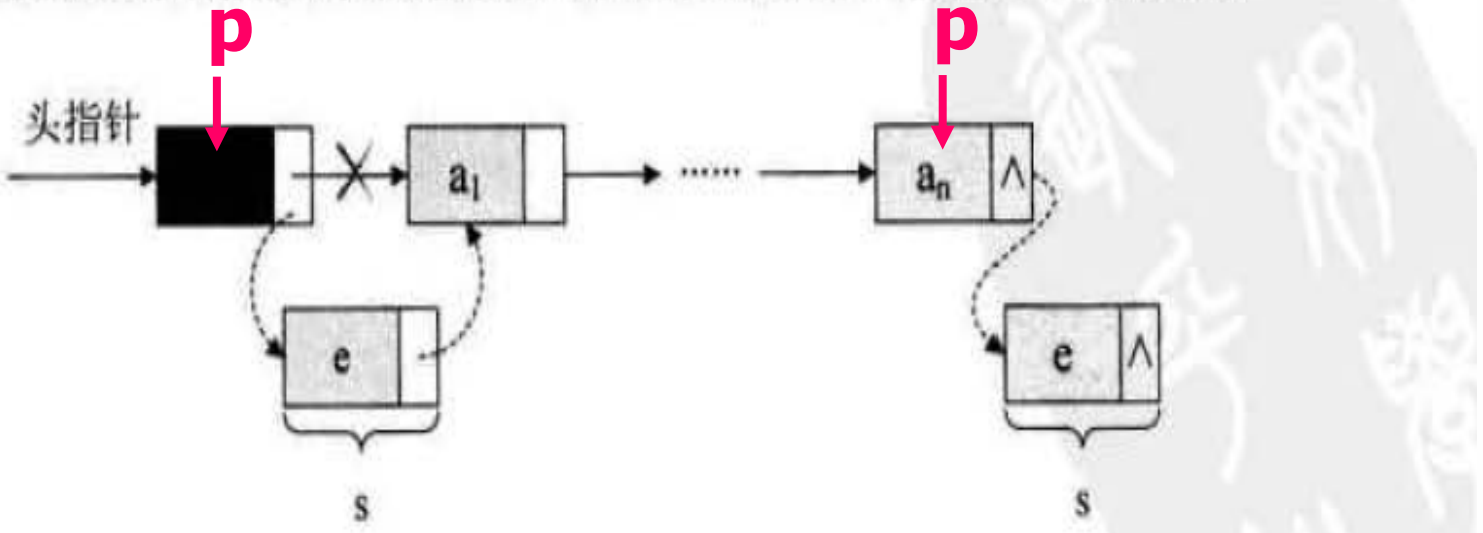


图 3-8-4

(8) 单链表的插入

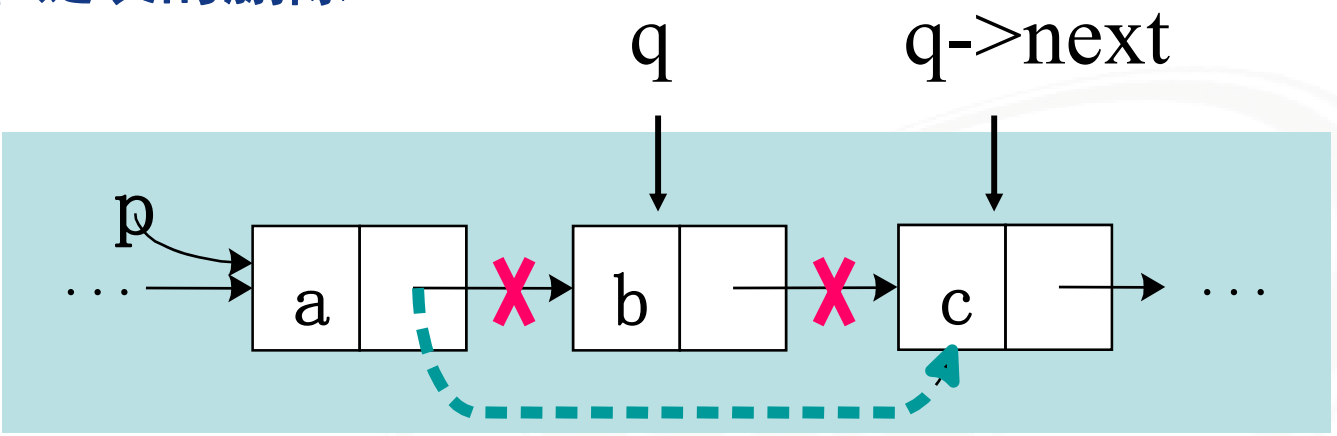
```
Status ListInsert_L(LinkList &L,int i,ElemType e){
  p=L;  j=1;
  while(p && j<i){p=p->next;++j;}      //寻找第i-1个元素
  if(!p && j<i) return ERROR;                  //i小于1或大于表长
  s=(LinkList)malloc(sizeof(LNode));       //生成新结点
  s->data=e;
  s->next=p->next;                              //插入L中
  p->next=s;
  return OK;
}//ListInsert_L
```

T(n)=O(n)

```c
/*单链表的插入，在链表的第i个位置插入e的元素*/
/* 初始条件：顺序线性表L已存在,1≤i≤ListLength(L)， */
/* 操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1 */
Status ListInsert_L(LinkList *L,int i,ElemType e)
{
    int j=1;
    LinkList p=NULL;
    LinkList s=NULL;

    p = *L;

    while (p && j < i)      /* 寻找第i-1个元素 */
    {
        p = p->next;
        ++j;
    }

    if (!p && j < i)
        return ERROR;    /* 第i个元素不存在 */

    s = (LinkList)malloc(sizeof(LNode));  /*  生成新结点(C语言标准函数) */
    s->data = e;
    s->next = p->next;       /* 将p的后继结点赋值给s的后继  */
    p->next = s;           /* 将s赋值给p的后继 */

    return OK;
}
```

带头结点单链表的操作

(9) 单链表的删除



q = p->next

p->next=q->next

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

带头结点单链表的操作

(9) 单链表的删除

```
Status ListDelete_L(LinkList L,int i,ElemType *e){
  p=L; j=1;
  while(p->next && j<i){            //寻找第i-1个元素
    p=p->next;++j;
  }
  if(!(p->next) && j<i) return ERROR;  //删除位置不合理
  q=p->next; p->next=q->next;          //删除并释放结点
  *e=q->data; free(q);
  return OK;
}//ListDelete_L
```

$T(n)=O(n)$

```c
/*单链表的删除，在链表中删除第e个元素*/
/* 初始条件：顺序线性表L已存在，1≤i≤ListLength(L) */
/* 操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1 */
Status ListDelete_L(LinkList *L,int i,ElemType *e)
{
    int j=1;
    LinkList p=NULL;
    LinkList q=NULL;

    p = *L;


    if(i<1)
        return ERROR;

    while (p->next && j < i)  /* 遍历寻找第i-1个元素 */
    {
        p = p->next;
        ++j;
    }

    if (!(p->next) && j < i)
        return ERROR;           /* 第i个元素不存在 */

    q = p->next;
    p->next = q->next;      /* 将q的后继赋值给p的后继 */
    *e = q->data;               /* 将q结点中的数据给e */
    free(q);                    /* 让系统回收此结点，释放内存 */

    return OK;
}
```

带头结点单链表的操作

单链表和顺序表的插入和删除算法对比

分析一下刚才我们讲解的单链表插入和删除算法，我们发现，它们其实都是由两部分组成：第一部分就是遍历查找第 i 个元素；第二部分就是插入和删除元素。

从整个算法来说，我们很容易推导出：它们的时间复杂度都是 $O(n)$。如果在我们不知道第 i 个元素的指针位置，单链表数据结构在插入和删除操作上，与线性表的顺序存储结构是没有太大优势的。但如果，我们希望从第 i 个位置，插入 10 个元素，对于顺序存储结构意味着，每一次插入都需要移动 n−i 个元素，每次都是 $O(n)$。而单链表，我们只需要在第一次时，找到第 i 个位置的指针，此时为 $O(n)$，接下来只是简单地通过赋值移动指针而已，时间复杂度都是 $O(1)$。显然，**对于插入或删除数据越频繁的操作，单链表的效率优势就越是明显。**

带头结点单链表的操作

(10) 单链表的遍历

```
void  ListTraverse_L (LinkList L) {
      p=L->next;
      while(p){
        vist(p->data); p=p->next;
        }
 }// ListTraverse_L

 Status visit(ElemType e){
    printf("The value of the element is %d\n", (int) e);
    return OK;
}
```

T(n)=O(n)

```c
/* 初始条件：顺序线性表L已存在 */
/* 操作结果：依次对L的每个数据元素输出 */
Status visit(ElemType c)
{
    printf("%d ",c);
    return OK;
}

Status ListTraverse_L(LinkList L)
{
    LinkList p=L->next;
    while(p)
    {
        visit(p->data);
        p=p->next;
    }
    printf("\n");
    return OK;
}
```

(11) 销毁单链表

```
Status DestroyList_L(LinkList &L){
  cp=L->next;
  while(cp!=NULL){          //遍历单链表,向系统交回每一个结点
    np=cp->next;            //保存下一个结点的指针
    free(cp);               //删除并释放当前结点
    cp=np;                  //使下一个结点成为当前结点
  }
  free(L);                  //删除并释放头结点
  return OK;
}// DestroyList_L
```

T(n)=O(n)

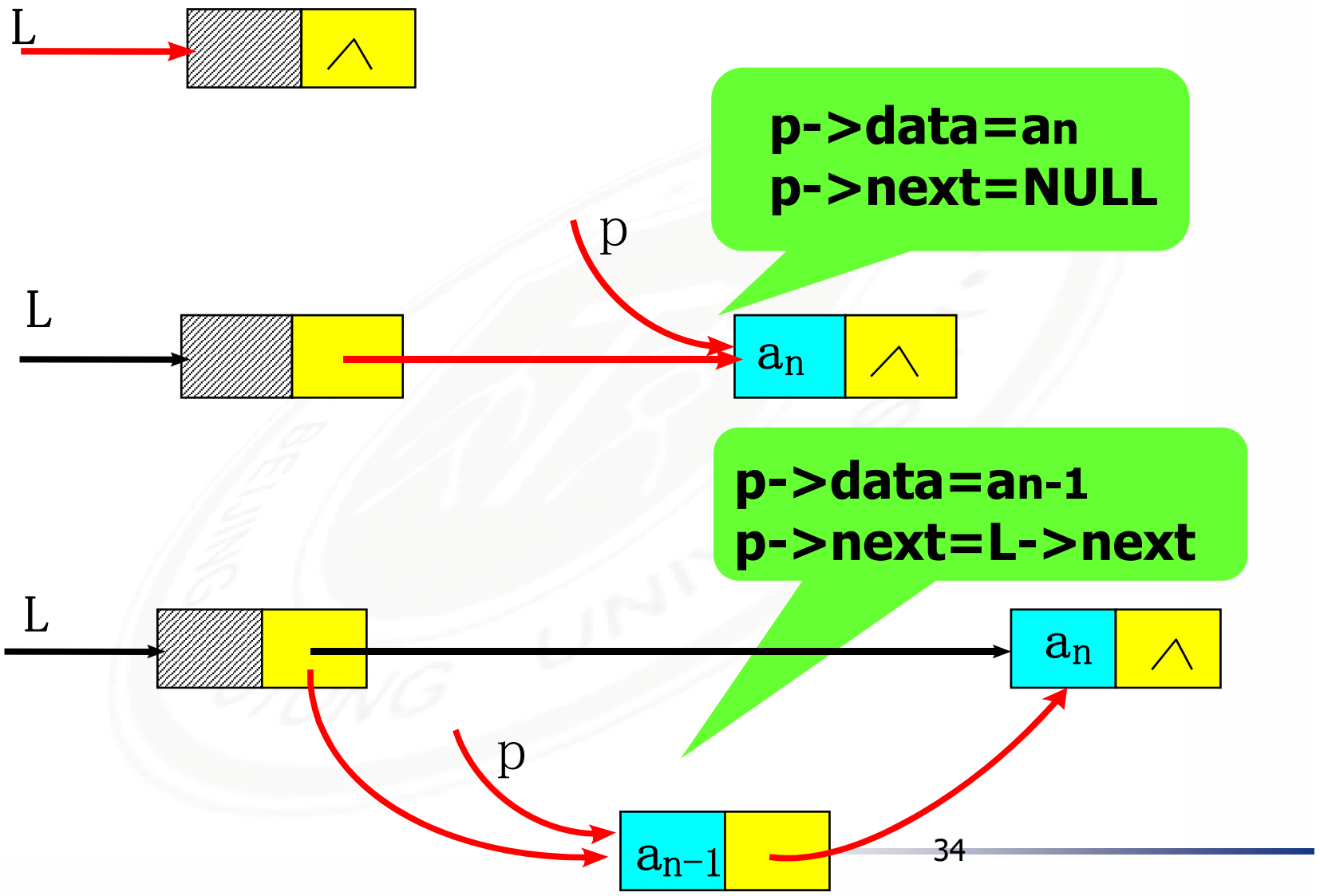
```
/*销毁链表*/
Status DestroyList_L(LinkList *L)
{
    LNode * cp = (* L)->next;
    LNode * np = NULL;

    while(cp!=NULL){              //遍历单链表,向系统交回每一个结点
        np=cp->next;              //保存下一个结点的指针
        free(cp);                 //删除并释放当前结点
        cp=np;                    //使下一个结点成为当前结点
    }

    free(*L);                     //删除并释放头结点
    return OK;
}
```

(12) 清空单链表

```
Status ClearList_L(LinkList &L){
  cp=L->next;
  while(cp!=NULL){          //遍历单链表,向系统交回每一个结点
    np=cp->next;            //保存下一个结点的指针
    free(cp);               //删除当前结点
    cp=np;                  //使下一个结点成为当前结点
  }
  L->next=NULL;             //置单链表为空
  return OK;
}
```

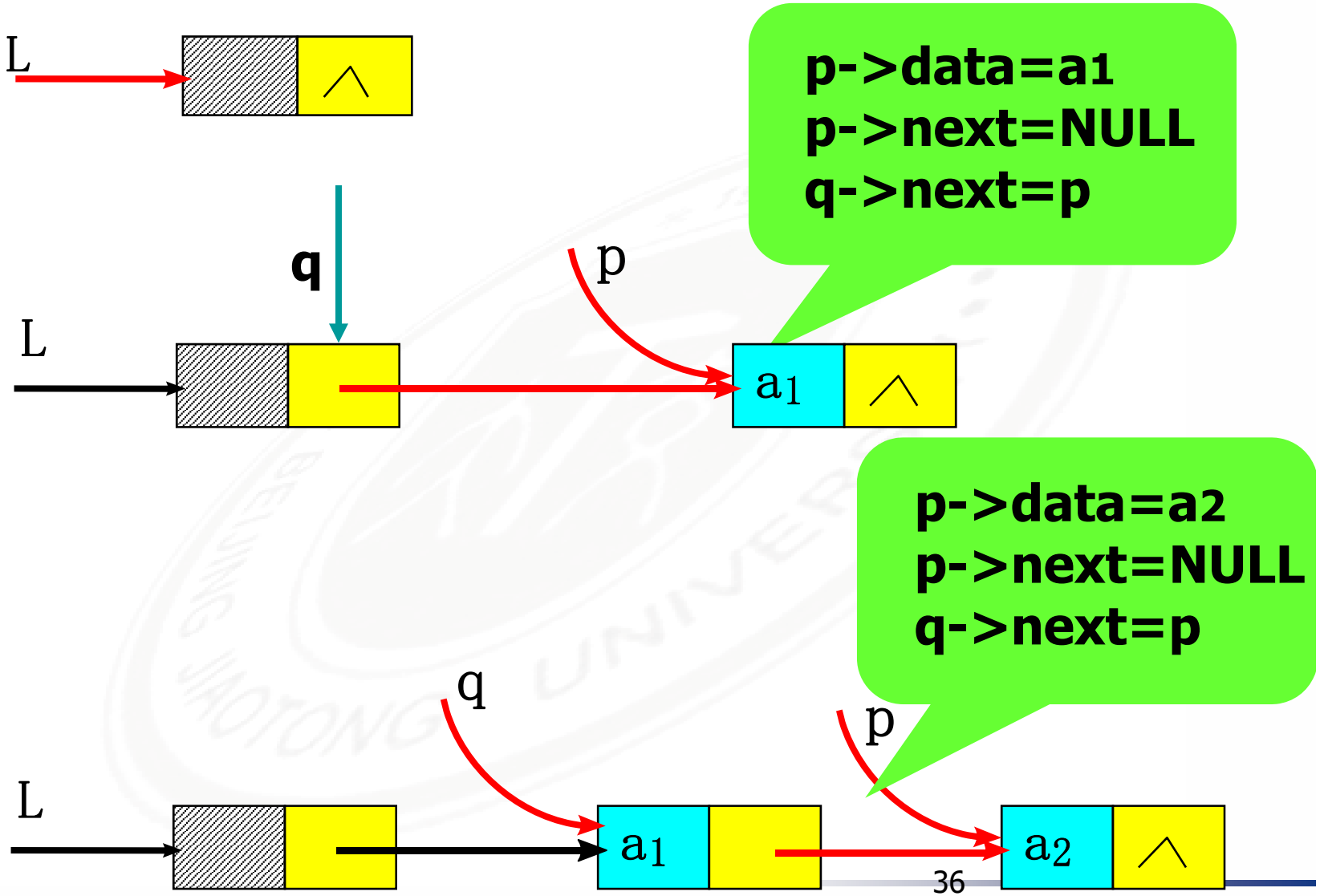
T(n)=O(n)

(7) 单链表的建立（头插法）



```
Status CreatList_L(LinkList &L,int n){
  //逆位序输入n个元素的值,建立带表头结点的单链线性表L
  L=(LinkList) malloc(sizeof(LNode));
  if(!L) return ERROR;
  L->next=NULL;                          //先建立一个带头结点的单链表
  for(i=n;i>0;--i){
    p=(LinkList) malloc(sizeof(LNode));    //生成新结点
    if(!p) return ERROR;
    scanf(&p->data);                       //输入元素值
    p->next=L->next; L->next=p;            //插入到表头
  }
  return OK;
}//CreatList_L
```

(7) 单链表的建立（尾插法）


L
p->data=a1
p->next=NULL
q->next=p
q
p
L
a1
p->data=a2
p->next=NULL
q->next=p
q
p
L
a1
a2

```
Status CreatList_L(LinkList &L, int n){
  //正序输入n个元素的值,建立带表头结点的单链线性表L
   L=(LinkList) malloc(sizeof(LNode));
   if(!L) return ERROR;
   L->next=NULL; //先建立一个带头结点的单链表
   q=L;
   for(i=0;i<n;i++){  //建立n个结点
       p=(LinkList) malloc(sizeof(LNode));       //生成新结点
       if(!s) return ERROR;
       scanf(&p->data);                          //输入元素值
       q->next=p;
       q=p;
   }
    q->next=NULL;
    return OK;
}//CreatList_L
```

引用方式

```c
/*  随机产生n个元素的值，建立带表头结点的单链线性表L（尾插法） */
void CreateListTail(LinkList * L)
{
    LinkList p = NULL;
    LinkList tail = NULL;
    int e=0;
    char c=0;

    tail= *L;                                   /* r为指向尾部的结点 */
    while(1)
    {
        scanf("%d", &e);

        p = (LNode *)malloc(sizeof(LNode)); /*  生成新结点 */
        p->data = e;              /*  随机生成100以内的数字 */
        tail->next=p;                           /* 将表尾终端结点的指针指向新结点 */
        tail = p;                               /* 将当前的新结点定义为表尾终端结点 */

        scanf("%c", &c);
        if(c =='\n')
            break;
    }

    tail->next = NULL;                          /* 表示当前链表结束 */
}
```

单链表与顺序表的对比

	顺序表	单链表
存储分配方式	用一段连续的存储单元，依次存储线性表的元素。	用一组任意的存储单元存储线性表的元素。
时间复杂度	• 查找操作顺序表的时间复杂度为O(1) • 插入和删除操作顺序表平均移动一般的元素，复杂度为O(n)	• 查找操作单链表的时间复杂度为O(n) • 插入和删除操作单链表找到位置i的指针后，插入和删除操作的复杂度仅为O(1)
空间复杂度	需要考虑溢出问题。顺序表的最大长度的难以设置。（套餐）	只要内存还有空间，随用随取，不限个数。（自助餐）

单链表与顺序表的对比

- 通过上述对比，我们可以得出一些经验性结论：
  - 若线性表需要频繁查找，很少进行插入和删除操作时，宜采用顺序存储结构；
    - 例如，用户账户管理，除了注册时是插入，大多数情况是读取。
  - 反之，若插入和删除操作频繁，查找少，宜采用单链表结构。
    - 例如，游戏中玩家的武器装备。
  - 当线性表中元素个数变化大，或者不知道有多大时，宜采用单链表结构
- 总之，要视情况选择。

《实验二：单链表的实现及基本操作》

- 通过键盘创建有若干个元素（可以是整型数值）的单链表，实现对单链表的初始化，对已建立的顺序表插入操作、删除操作、查找操作、遍历输出单链表。
- 要求各个操作均以函数的形式实现，并且在主函数中调用各个函数实现以下操作：

① 键盘输入单链表x、x、x、x、x、x，并输出显示，其中x为任意整数。

② 在单链表的第4个位置插入67，并输出单链表中的各元素值。

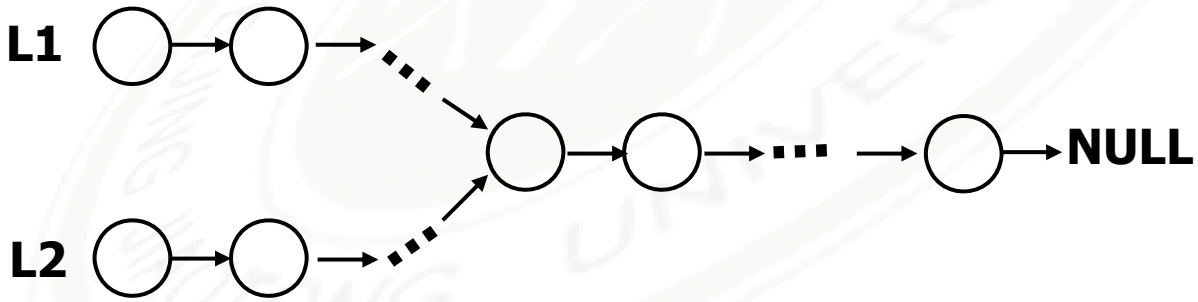
③ 删除单链表中的第2个数据元素，并输出单链表中的各元素值。

④ 查找单链表中的第5个元素并输出该元素的值。

```
int main()
{
    int e1;
    LinkList queue = NULL;

    InitList_L(&queue);

    CreateListTail(& queue);
    ListTraverse_L(queue);

    ListInsert_L(&queue,4,67);
    ListTraverse_L(queue);

    ListDelete_L(&queue,2,&e1);
    ListTraverse_L(queue);

    GetElem_L(queue, 5, &e1) ;
    printf("第5个元素是 %d\n",e1);

    DestroyList_L(&queue);

    return OK;
}
```

练习题

1. 已知顺序表**L**中的数据元素为**int**。设计算法将其调整为左右两部分，左边的元素（即排在前面的）均为奇数，右边所有元素（即排在后面的）均为偶数，并要求算法的时间复杂度为**O(n),**空间复杂度为**O**（**1**）。
2. 假设有一个没有头指针的单链表。一个指针指向该单链表中间的一个结点（不是第一个，也不是最后一个结点），请将该结点从单链表中删除。
3. 给出两个单向链表的头指针，如**L1**和**L2**，判断这两个链表是否相交。为简化问题，假设两个链表均不带环。



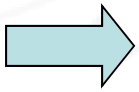
例3 合并有序表

问题描述：

假设头指针为La和Lb的单链表分别为线性表LA和LB的存储结构，归并La和Lb得到单链表Lc

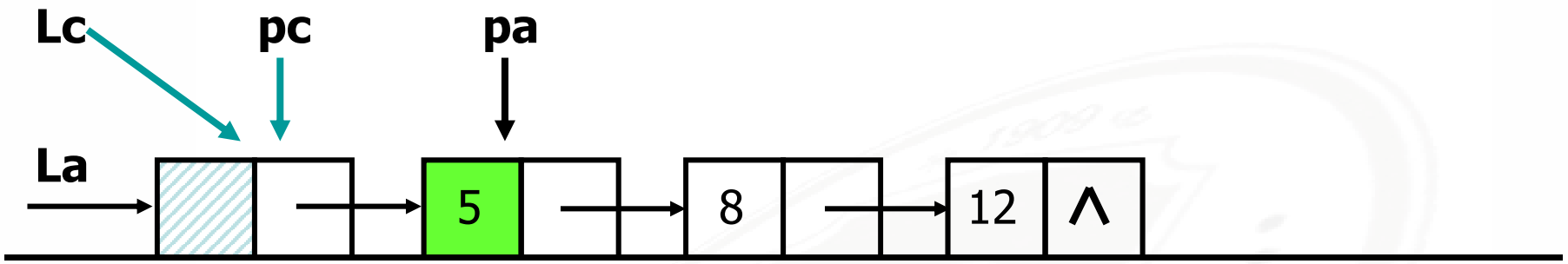
LA=(5, 8, 12)

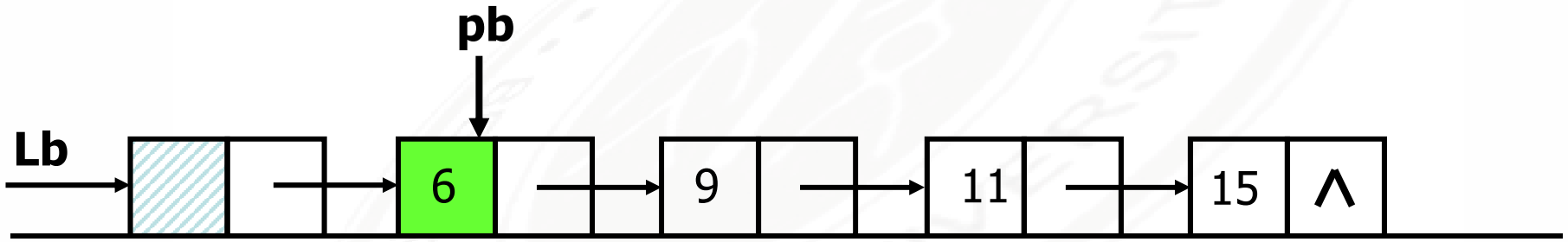
LB=(6, 9, 11, 15)



LC=(5, 6, 8, 9, 11, 12, 15)

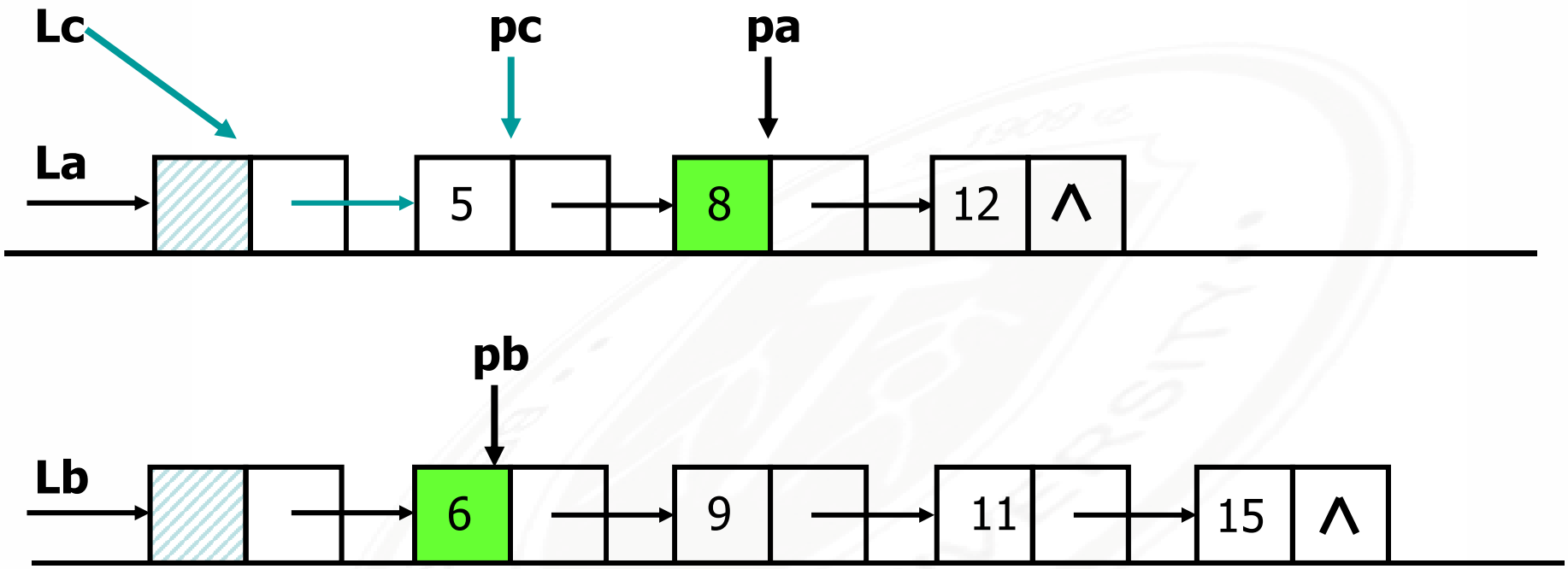
例3 合并有序表





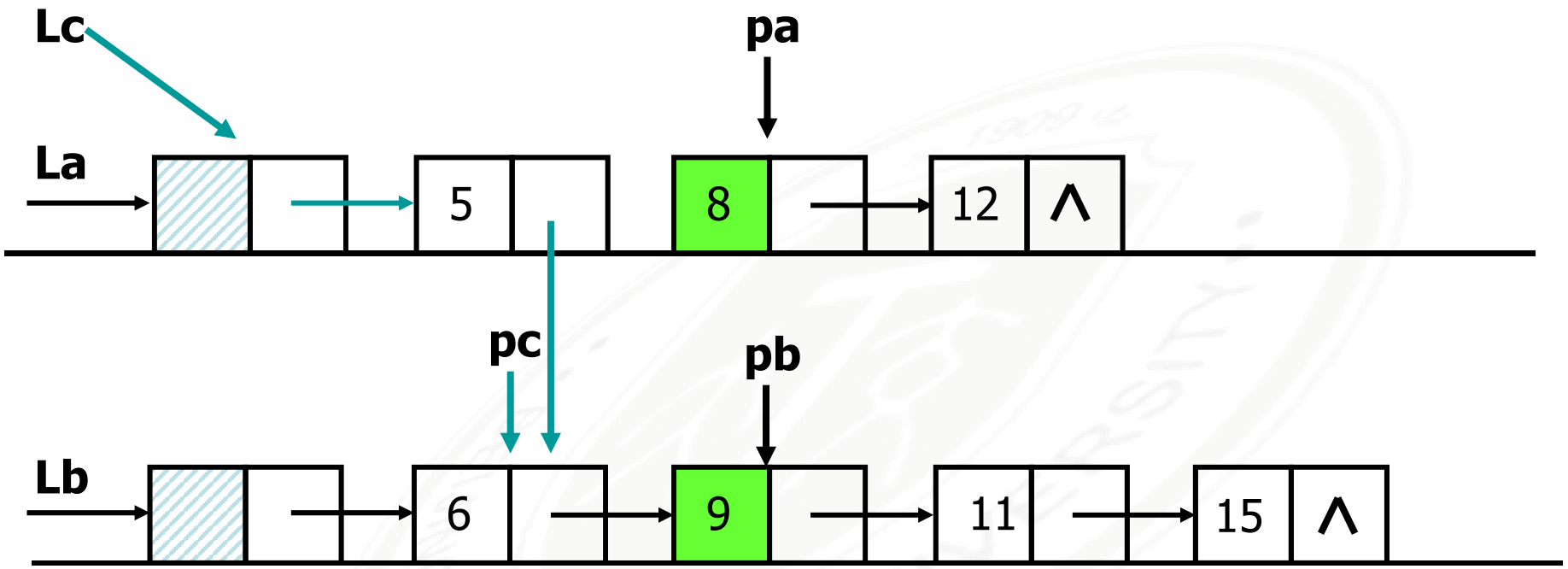

```
pa=La->next;
pb=Lb->next;
pc=La=Lc;
```

例3 合并有序表



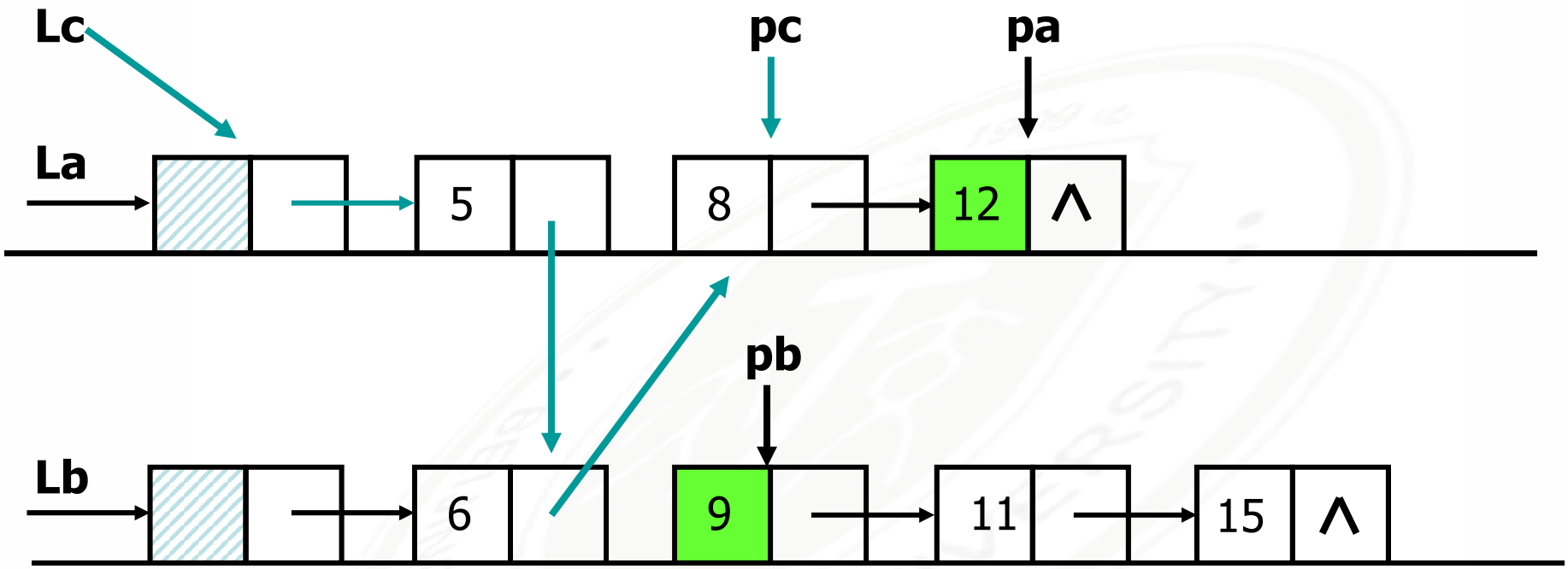

```
pc->next=pa;
pc=pa
pa=pa->next;
```

例3 合并有序表



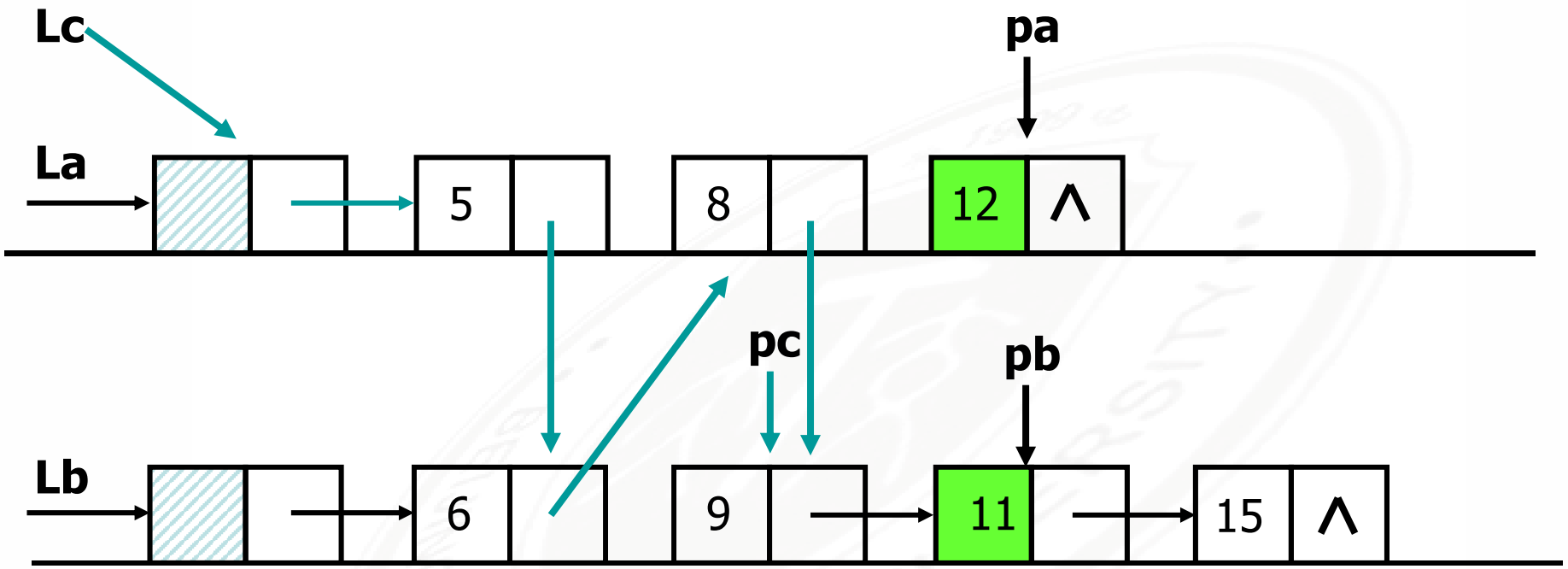

```
pc->next=pb;
pc=pb
pb=pb->next;
```

例3 合并有序表



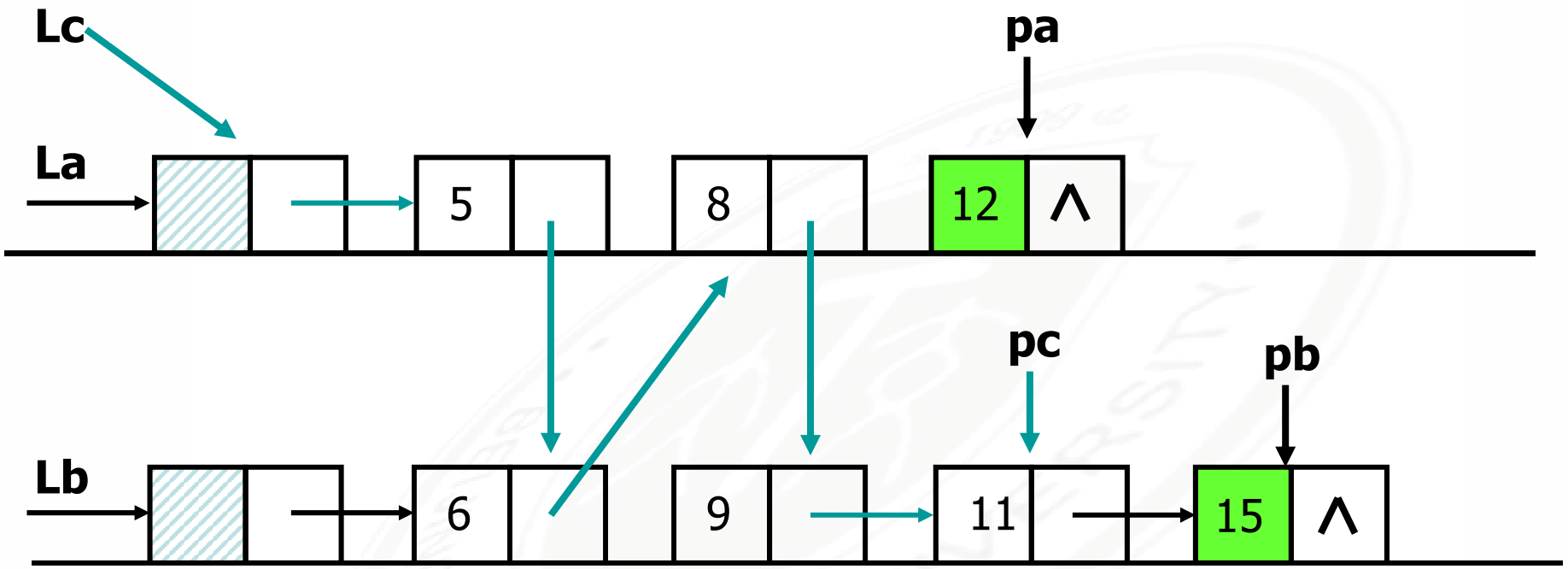

```
pc->next=pa;
pc=pa
pa=pa->next;
```

例3 合并有序表



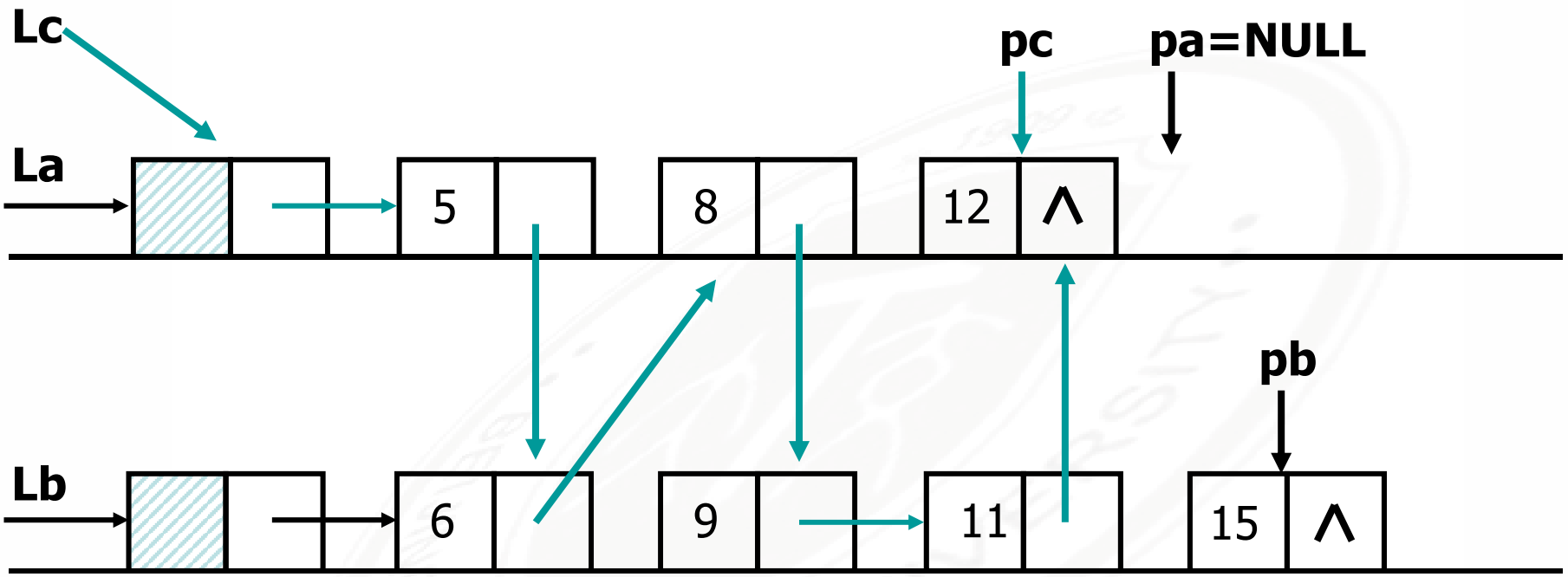

```
pc->next=pb;
pc=pb
pb=pb->next;
```

例3 合并有序表



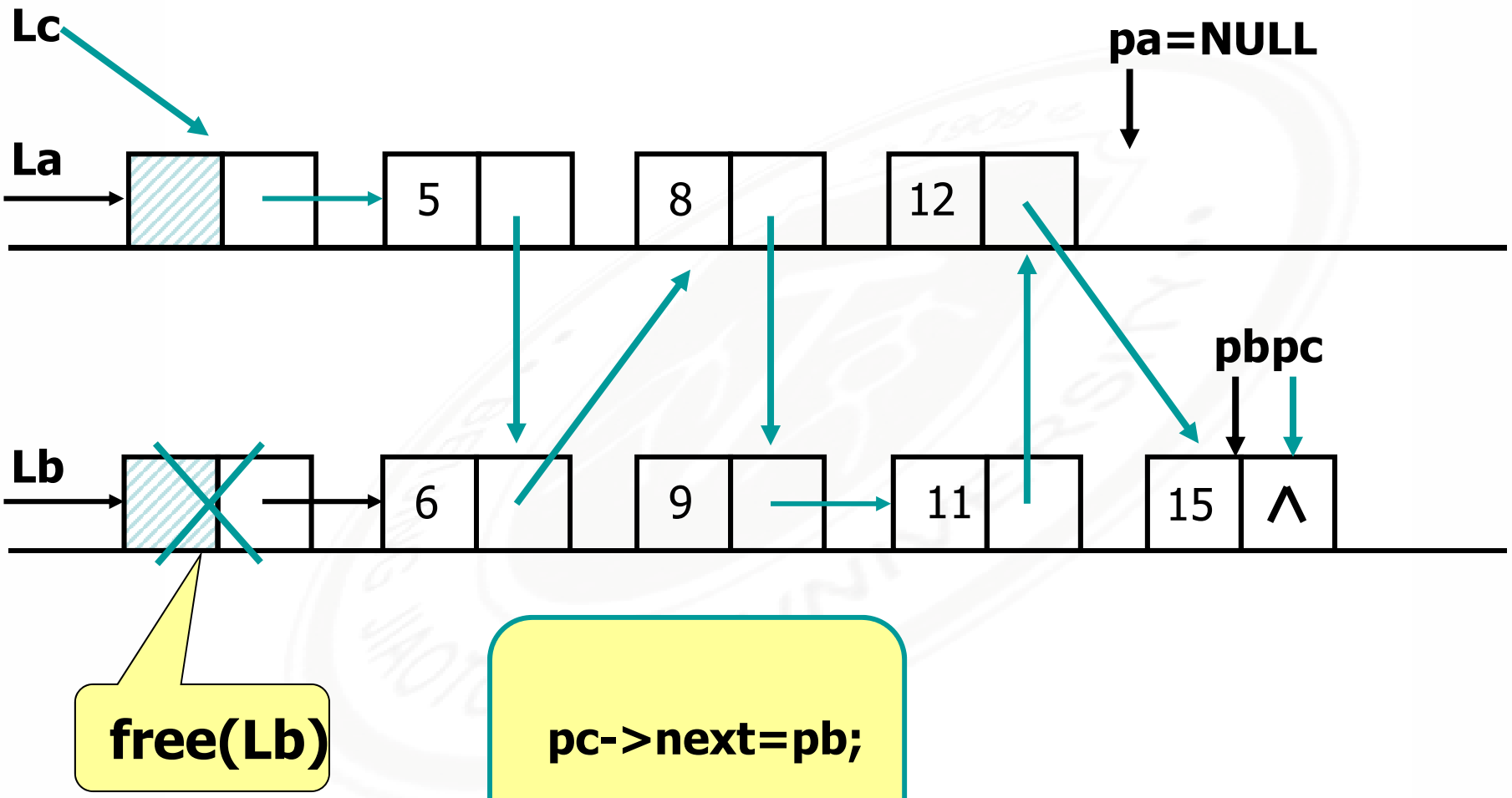

```
pc->next=pb;
pc=pb
pb=pb->next;
```

例3 合并有序表




```
pc->next=pa;
pc=pa
pa=pa->next;
```

例3 合并有序表



```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
  //已知单链线性表La和Lb的元素按值非递减排列
  //归并La和Lb得到新的单链线性表Lc,Lc的元素也按值非递减排列
  pa=La->next;  pb=Lb->next;
  Lc=pc=La;              //用La的头结点作为Lc的头结点
  while(pa && pb){
    if(pa->data<=pb->data){
      pc->next=pa; pc=pa; pa=pa->next;
    }
    else{pc->next=pb; pc=pb; pb=pb->next;}
  pc->next= pa? pa: pb;    //插入剩余段
  free(Lb);                //释放Lb的头结点
}//MergeList_L
```