第二章 线性表（二）

顺序表回顾

- 线性表：由同类型有限个数据元素构成有序序列

- 顺序表的优点
  - 逻辑上相邻的两个元素在物理位置上也相邻
  - 可随意存取
  - 它的存储位置可有一个简单直观的公式来表示

- 顺序表的缺点
  - 初始化时需预分较大空间
  - 插入、删除需移动大量元素
  - 表的容量难以扩充

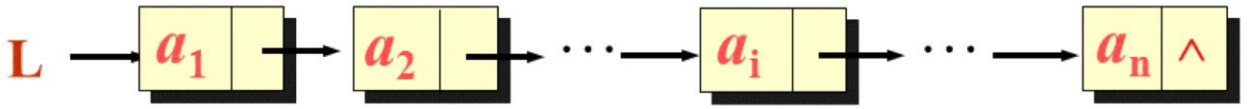
链表

- 链表是线性表的链式存储表示
- 链表中逻辑关系相邻的元素不一定在存储位置上相连，用一个链（指针）表示元素之间的邻接关系

❖ 线性表的链式存储表示主要有三种形式：

- 线性链表（单链表）
- 循环链表
- 双向链表

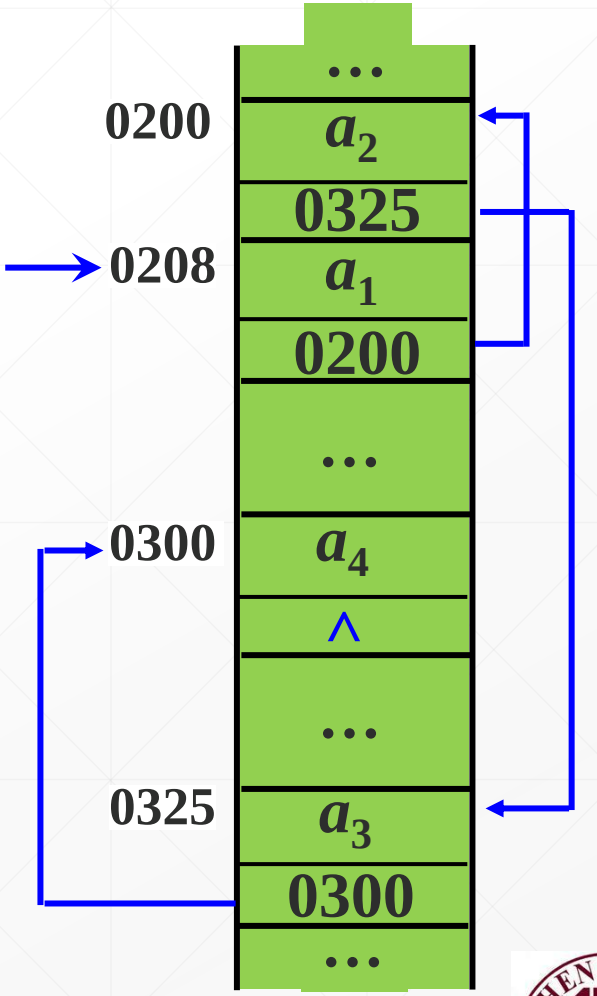
线性链表



- 线性链表的元素称为结点(node)
- 每个结点均含有两个域：存放元素的**数据域**和存放其后继结点的**指针域**

data	next

- 例： $(a1, a2, a3, a4)$的存储示意图

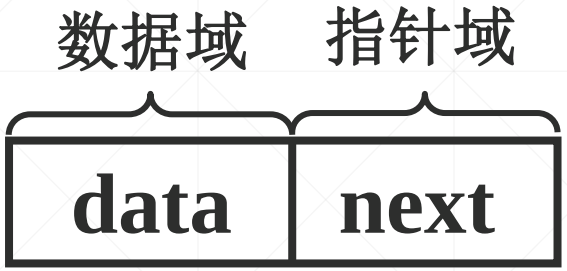


- 存储结构特点
  - ✓ 逻辑次序和物理次序不一定相同；
  - ✓ 元素之间的逻辑关系用指针表示；
  - ✓ 需要额外空间存储元素之间的关系
  - ✓ 非随机访问存取结构（顺序访问）

必须从头指针出发寻找

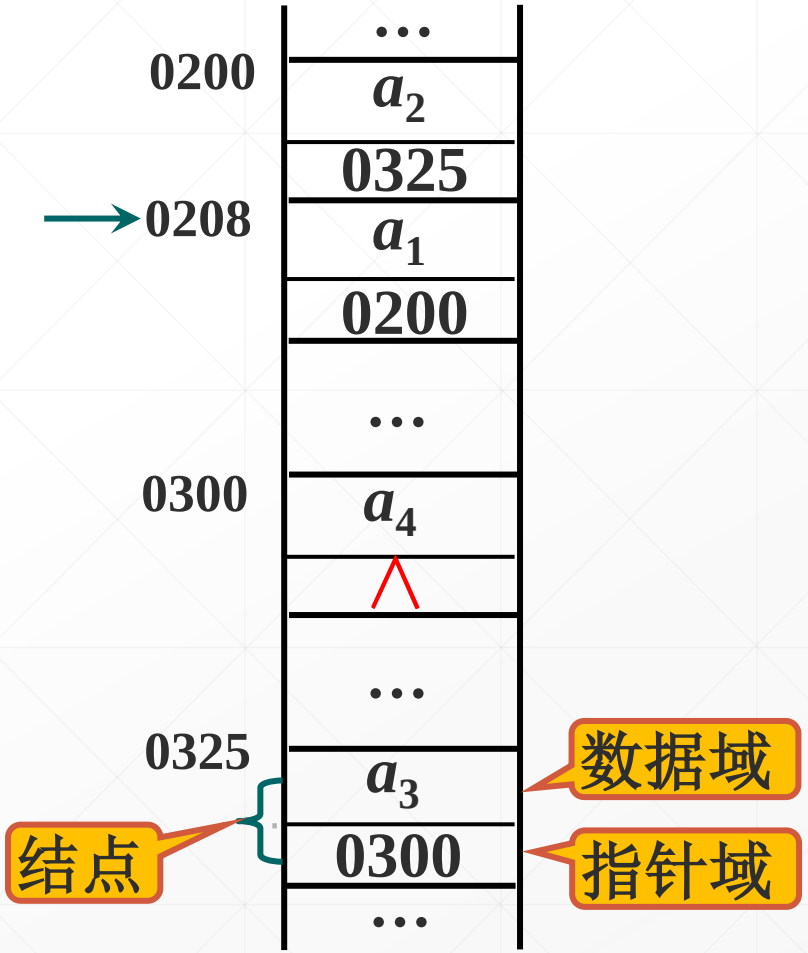
存储结构的定义

- 结点结构：



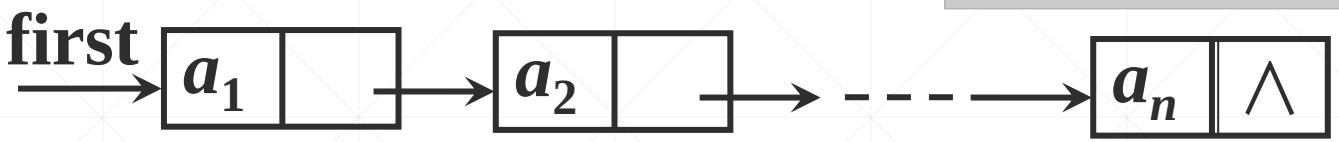
- 存储结构类型定义

```
struct LNode{
    ElemType data;     // 数据域
    LNode *next;       // 后继指针
};  /*结点型*/
 LNode *LinkList;      // 定义单链表
```



存储结构的定义（cont.）

- 单链表图示：
  - 不带头结点的单链表 - 头指针

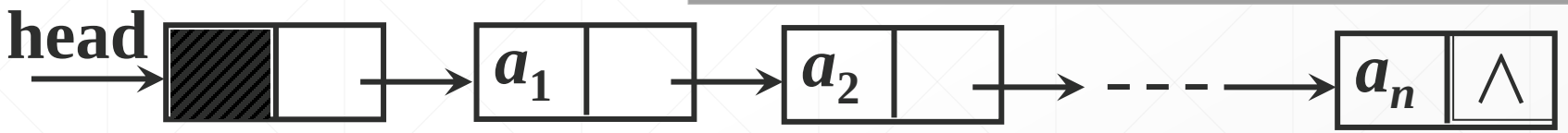


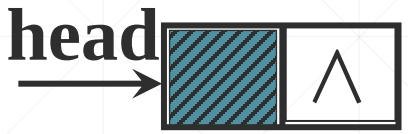
空表： first==NULL;

- ✓ 线性链表可由头指针唯一确定
- ✓ 最后一个结点没有后继，它的指针域为空（NULL）

  - 带头结点的单链表

第一个结点之前附设一个头结点，其数据域可以为空，也可以为线性链表的长度信息





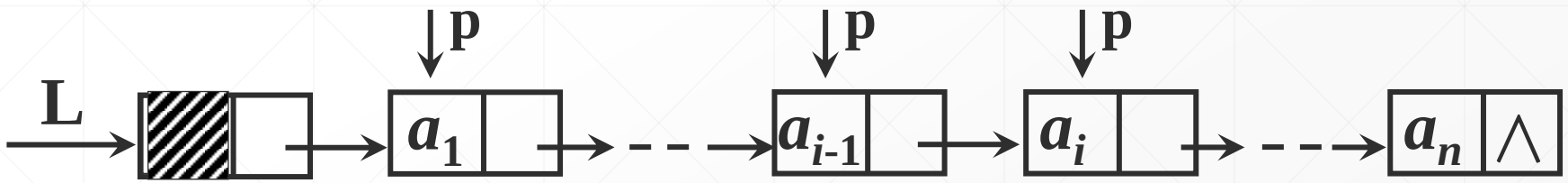
空表： head->next=NULL

- 头结点的作用：
  - ✓ 空表和非空表表示统一
  - ✓ 在任意位置的插入或者删除的代码统一

线性链表的查找

- 在线性链表中找第 *i* 个元素
- 由于线性链表中元素的存储位置具有随机性，因此只有从头结点开始，顺链一步步查找

```
Status searchLinkList(LNode *L, int i){
LNode *p=L →next; // p指向第一个结点
int j=1; //计数器
while(p!=NULL&&j<i){
p=p-next;  j++;
}
// 顺链向后寻找，直到第i个结点或p为空
if (!p||j>i) return ERROR; //第i个元素不存在
e=p->data; return OK;
}
```

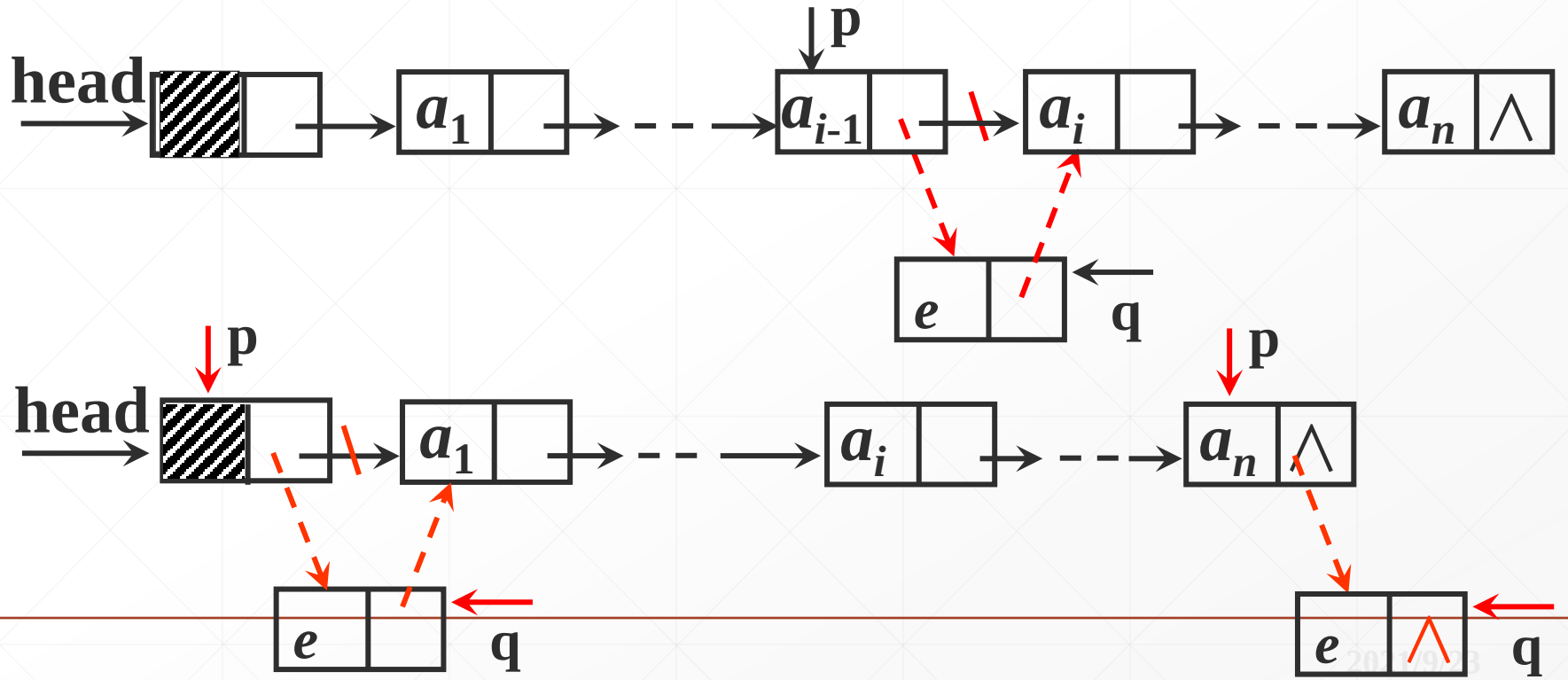


线性链表的查找（cont.）

- 算法的时间复杂度主要取决于while循环中的语句频度
- 频度与被查找元素在单链表中的位置有关
- 若$1 \leq i \leq n$, 则频度为 i-1，否则为 n
- 因此时间复杂度为 $\boldsymbol{O(n)}$

线性链表的插入

- q = new LNode;
- q -〉 data = e ;
- q -〉 next = p-〉 next ;
- p -〉 next = q ;



线性链表的插入（cont.）

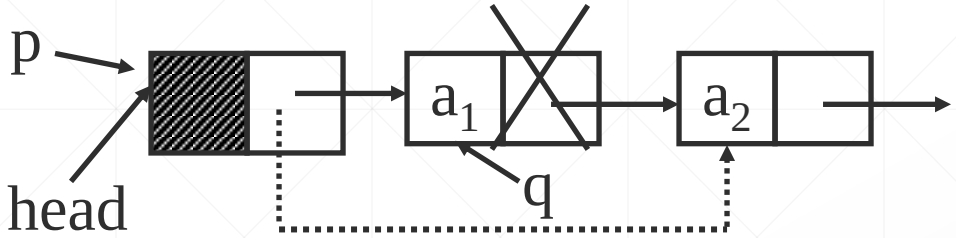
- 在已知（指定）结点后插入新结点，时间复杂度为$\boldsymbol{O(1)}$
- 在线性链表的第$i-1$元素与第i元素之间插入一个新元素：

✓需要先找到链表的第$i-1$个结点，再插入新结点

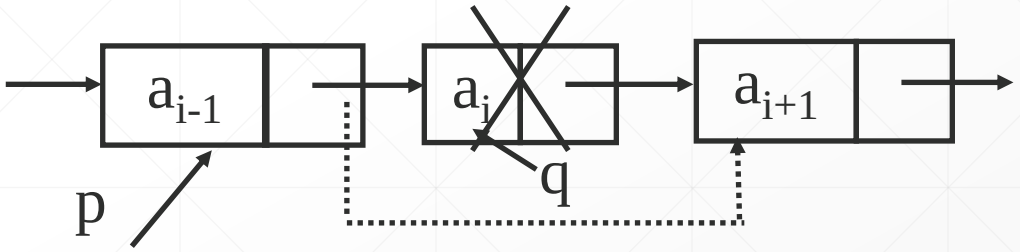
✓算法的时间复杂度主要取决于while循环中的语句频度，频度与在线性链表中的元素插入位置有关，因此线性链表的插入的时间复杂度为 $\boldsymbol{O(n)}$

线性链表的删除

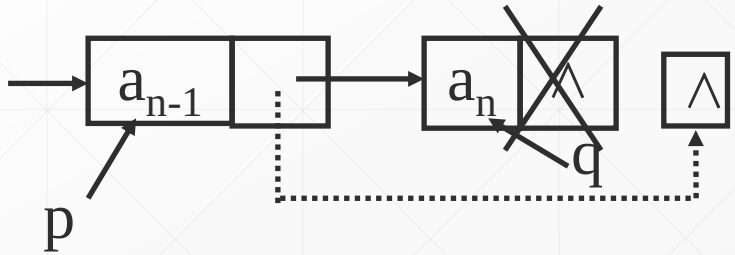
- q = p→next ;
- p→next = q→next ;
- delete q ;



(a) 删除第一个元素



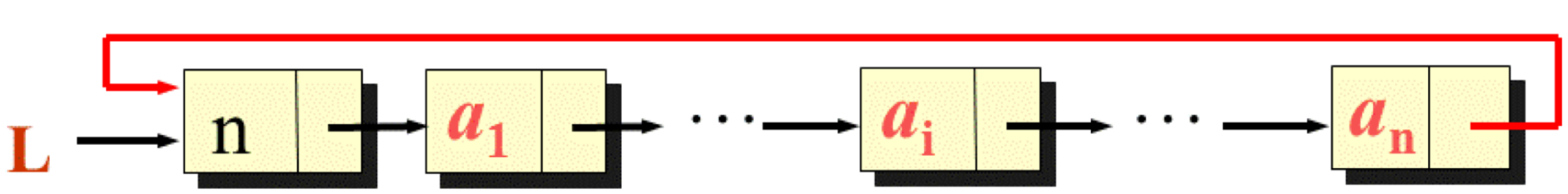
(b) 删除中间元素



(c) 删除表尾元素

循环链表

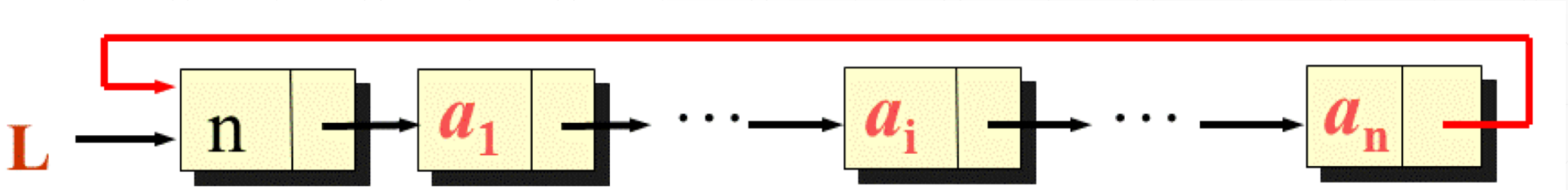
- 循环链表是一种特殊的线性链表
- 循环链表中最后一个结点的指针域指向头结点，整个链表形成一个环。



循环链表

- 查找、插入和删除

- 在循环链表中查找指定元素，插入一个结点或删除一个结点的操作与线性链表基本一致
- 差别仅在于算法中的循环条件不是 p->next或p是否为空(^), 而是它们是否等于头指针(L)

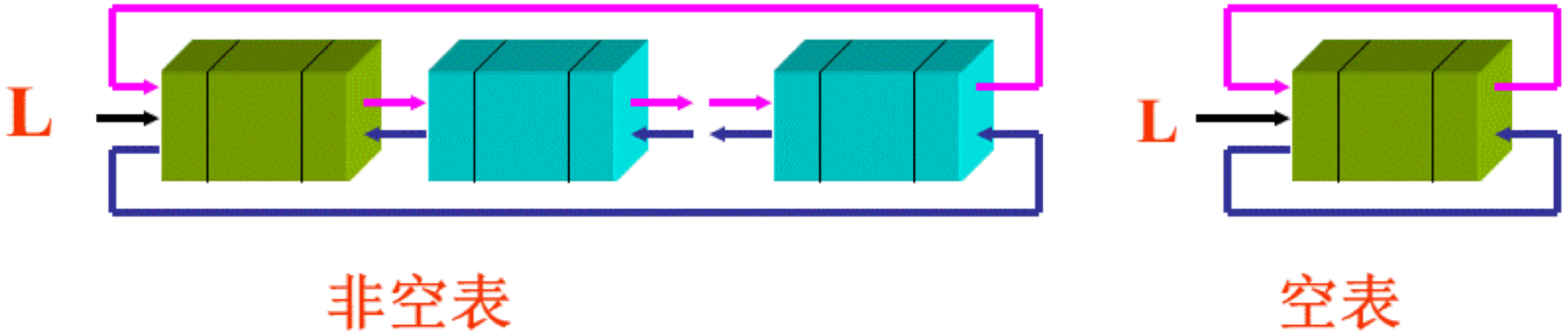


双向链表

- 双向链表也是一种特殊的线性链表
- 双向链表中每个结点有两个指针，一个指针指向直接后继（next），另一个指针指向直接前驱（prior）
- 优点：实现双向查找（单链表不易做到）
- 缺点：空间开销大

双向循环链表

- 双向循环链表中存在两个环（一个是直接后继环（红），另一个是直接前驱环（蓝））



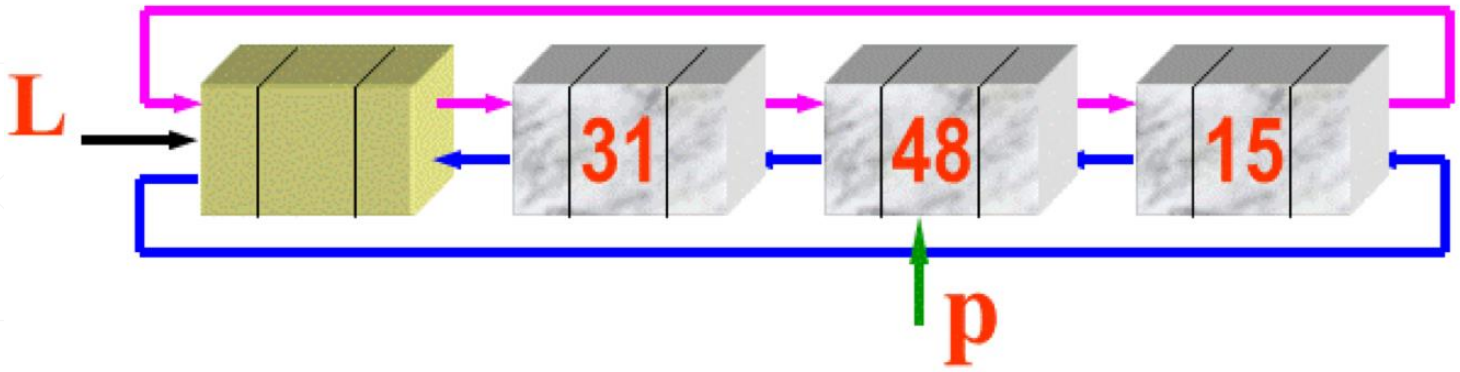
双向链表的定义

- 定义一个双向链表的结点

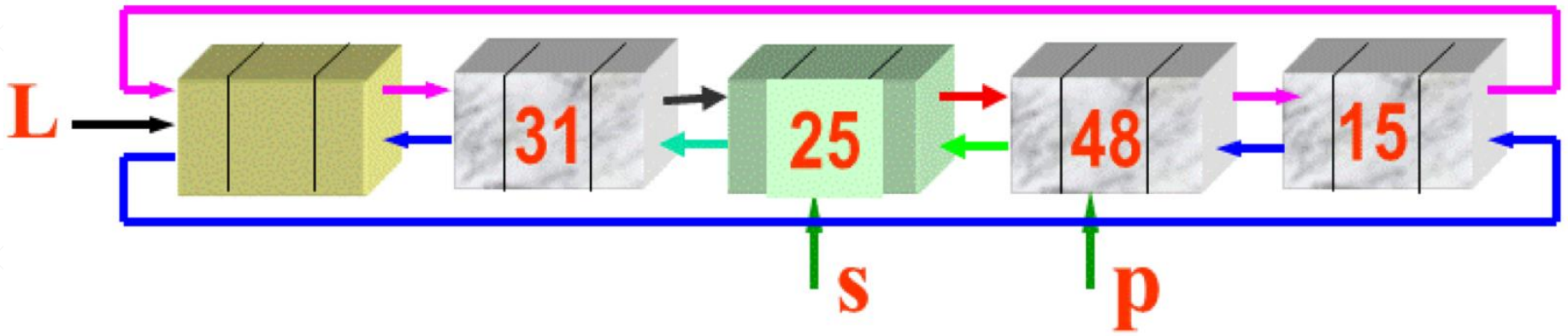
```
typedef struct DuLNode{
        ElementType data;     // 数据域
        DuLNode *prior;          // 前驱指针
        DuLNode *next;          // 后继指针
};

DuLNode *DuLinkList;          // 定义双向链表
```

双向链表的插入

- 双向链表的插入操作需要改变两个方向的指针

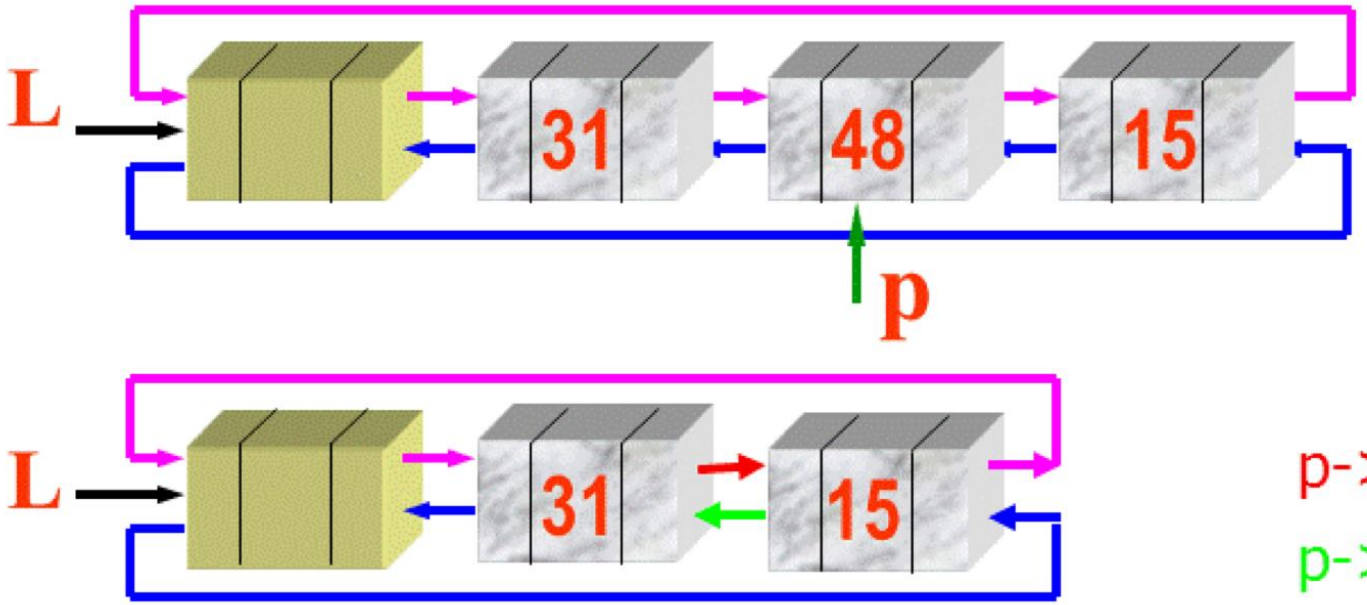


- s->prior=p->prior;
- p->prior->next=s;
- s->next=p;
- p->prior=s;



双向链表的删除

- 双向链表的删除操作需要改变两个方向的指针




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

顺序表与链表的比较

I. 基于空间的比较

- 存储分配的方式
  - 顺序表的存储空间是静态分配的
  - 链表的存储空间是动态分配的

- 存储密度=结点数据本身所占的存储量/结点结构所占的存储总量
  - 顺序表的存储密度=1
  - 链表的存储密度<1

顺序表与链表的比较

II. 基于时间的比较

- 存取方式
  - 顺序表中的元素的值可以随机存取，也可以顺序存取
  - 链表必须顺序存取［即需要沿链查找指定位置］

- 插入/删除时移动元素个数
  - 顺序表平均需要移动近一半元素
  - 链表不需要移动元素，只需要修改指针

顺序表与链表的比较

I. 基于应用的比较

- 如果线性表主要存储大量的数据，并主要用于查找时，采用顺序表较好，如数据库

- 如果线性表存储的数据元素经常需要做插入与删除操作，则采用链表较好，如操作系统中进行控制块（PCB）的管理，内存空间的管理等