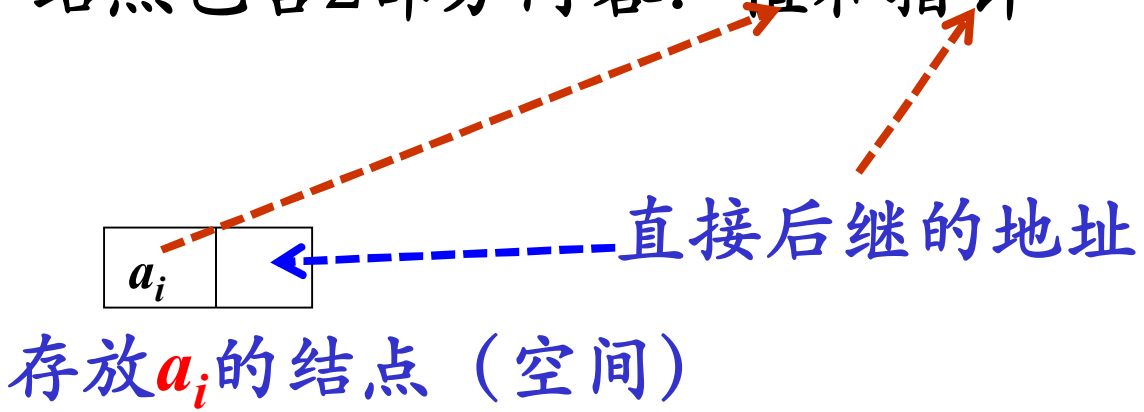
2.3 线性表的链式表示和实现

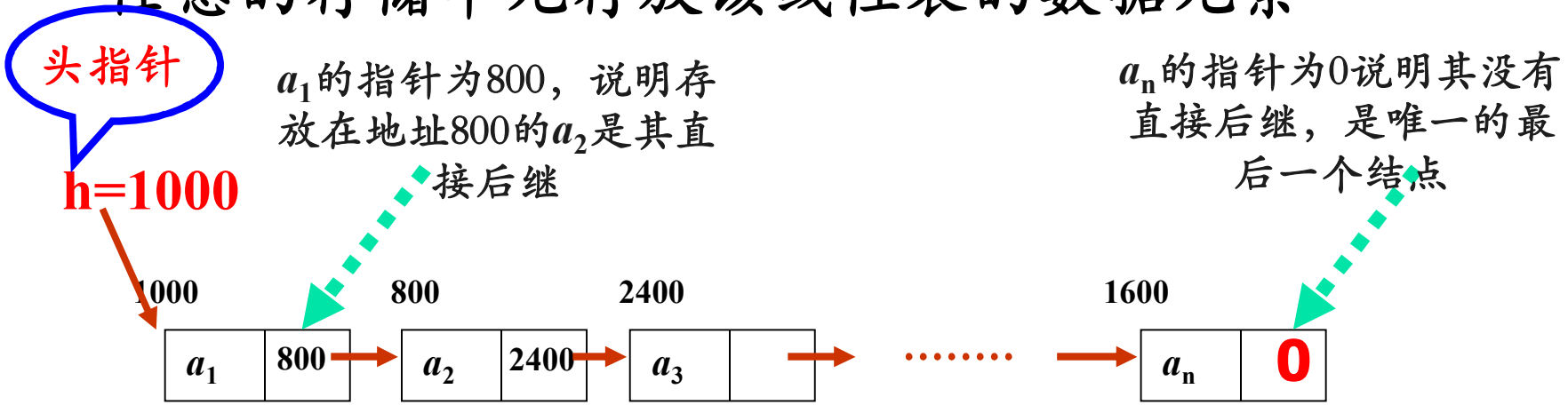
线性表的链式存储结构定义

- 用一组地址任意的存储单元存放线性表的数据元素。
- 每个数据元素存放于一个结点
- 每个结点包含2部分内容：值和指针



线性表的链式存储结构定义

- 例如：线性表List=(a_1,a_2,……,a_n)，用一组地址任意的存储单元存放该线性表的数据元素

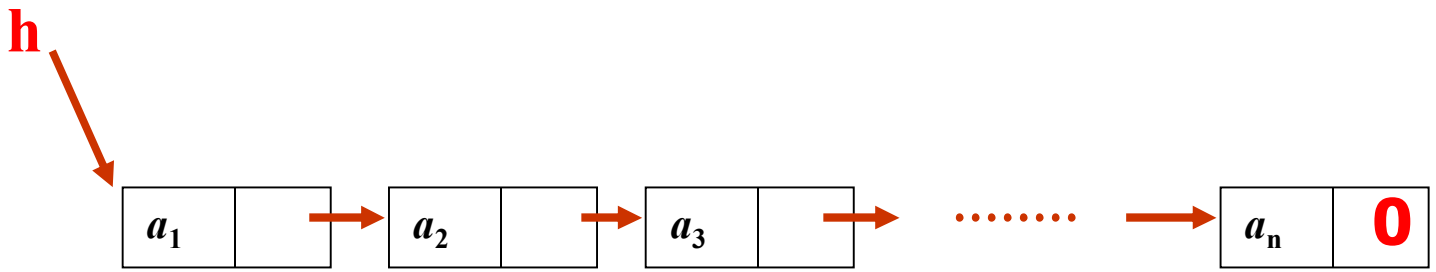


特点：逻辑相邻不一定物理相邻，只能**顺序存取**

访问第i个数据元素，必须从第一个数据元素开始，沿着每个结点的指针顺次找到第i个数据元素

线性表的链式存储结构定义

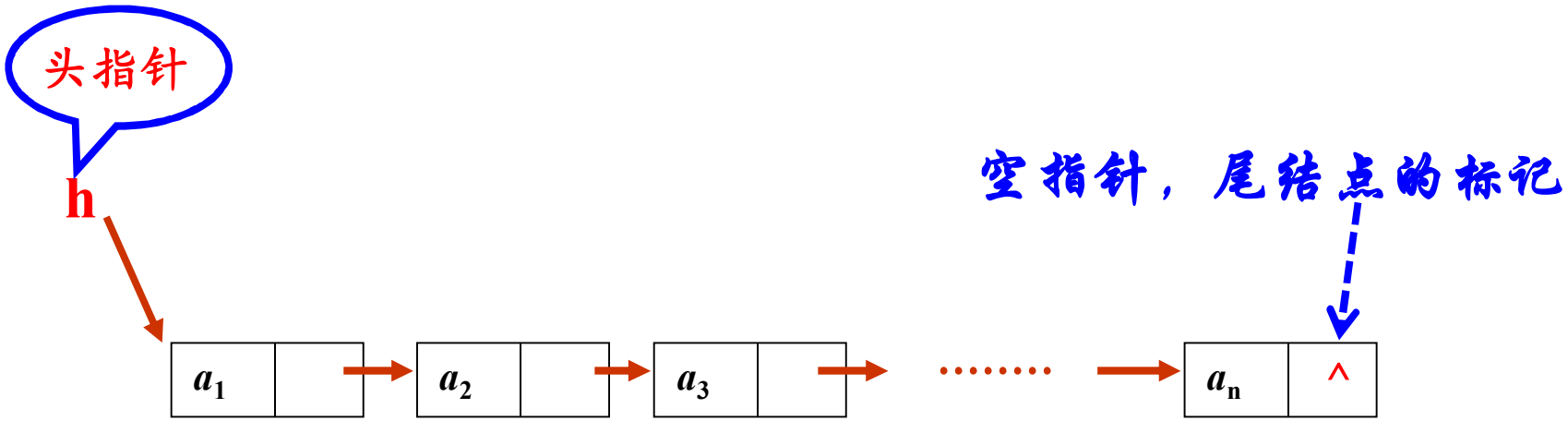
- List=(a_1,a_2,……,a_n)



线性单链表的简图—指针表示数据元素的直接后继

线性表的链式存储结构定义

- List=(a_1,a_2,……,a_n)

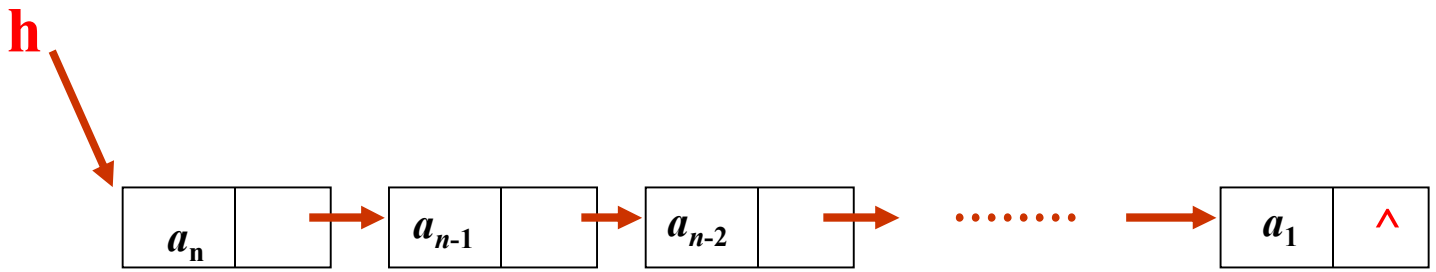


线性单链表的简图—指针表示数据元素的直接后继

•List=(a_1,a_2,……,a_n)的表示与建立要素：保存头指针，设置尾结点

线性表的链式存储结构定义

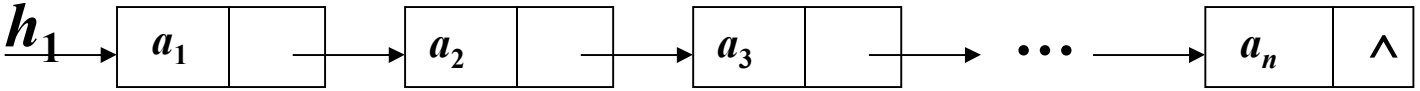
- List=(a_1,a_2,……,a_n)
- 说明：结点中的指针部分存放逻辑关系，也可以存直接前驱的地址



线性单链表的简图—指针表示数据元素的直接前驱

具体实现时：两种单链表

不带表头结点的线性单链表

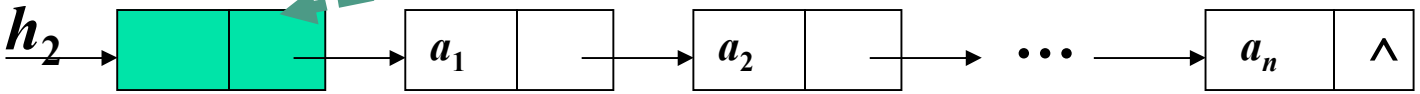


- 头指针指向第一个数据元素的结点空间，插入删除略复杂，会涉及头指针的修改，例如删除第一个数据元素

带表头结点的线性单链表

表头结点

- 表头结点通常空着或存放特殊的信息，比如线性表的长度



- 指针指向的第一个结点是表头结点，插入删除操作不会修改头指针；但多用一个结点空间
- 表头结点的直接后继才是线性表的第一个数据元素

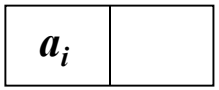
线性单链表的实现

- 线性单链表的实现有2种方式：
  - 动态链表—指针数据类型←主要介绍
  - 静态链表—数组 ←自己看 

- 动态单链表定义：

数据元素的值　指针—存放逻辑关系

data　next



存放每个数据元素的结点空间

```
typedef struct node{//定义单链表中存放每个数据元素的结点类型
        ElemType  data ;
        struct node *next;}Node,  *LinkList;
LinkList h,p;//定义指针类型变量
Node *q; //定义指针类型变量
```

定义指针类型变量没有指向实际的结点空间，必须初始化

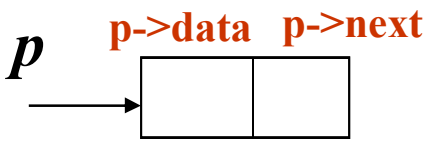
指针类型变量的初始化操作

2种初始化方法：申请空间malloc和赋值语句

1. malloc( )

p=(LinkList)malloc(sizeof(Node));

p->data　　p->next

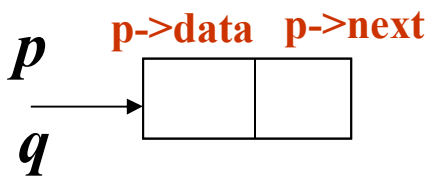


指针类型变量的初始化操作

2. 赋值语句

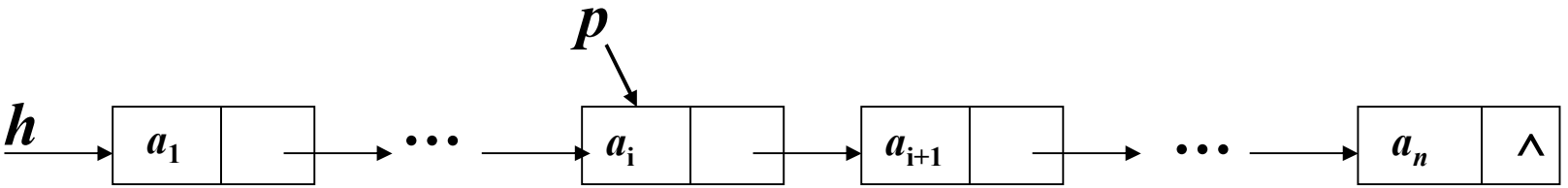
q=p;

把已经存在的结点地址p赋给一个指针变量q, 这样p和q指向同一个结点空间



算法—插入操作

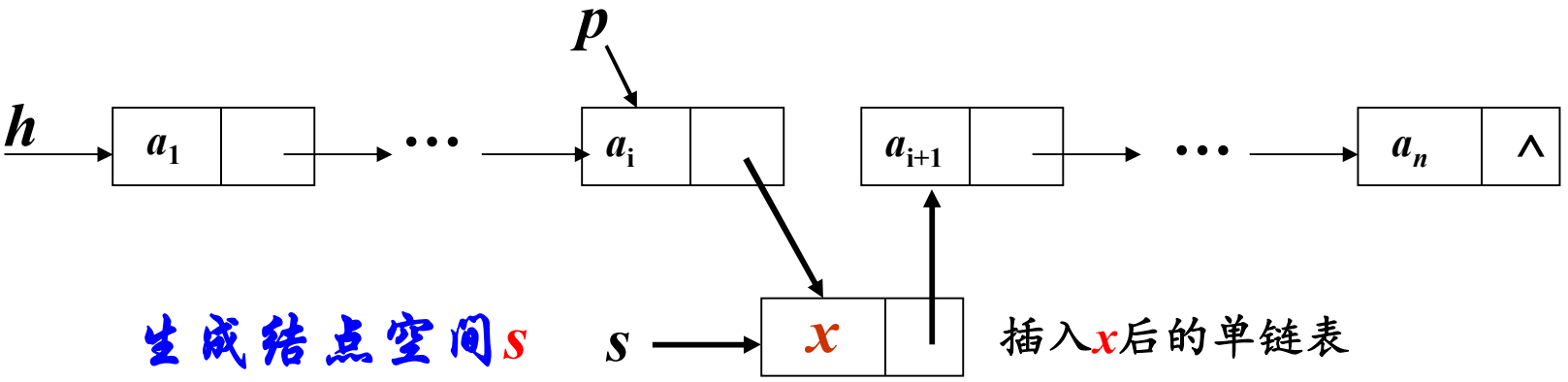
- 在线性单链表的*p*结点之后插入一个新的结点*x*。
- **假设：线性单链表已经建好**



插入*x*前单链表

算法—插入操作—不需要移动数据

- 在线性单链表的*p*结点之后插入一个新的结点*x*。
- **假设：线性单链表已经建好**



生成结点空间*s*

插入的数据*x*放入结点空间*s*

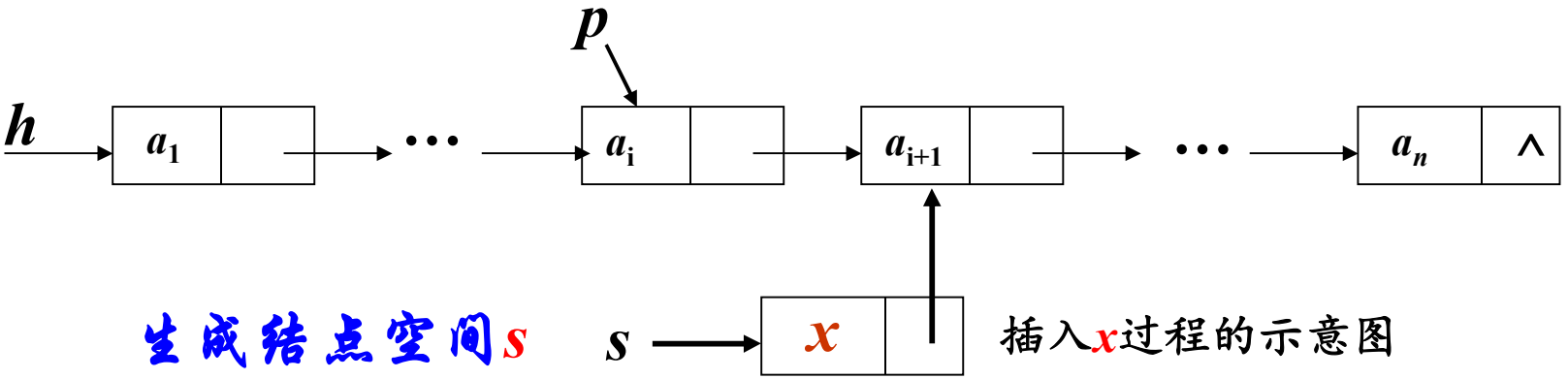
p的直接后继为新插结点*s*的直接后继

新插结点*s*为p的直接后继

s=(LinkList)malloc(sizeof(Node));
s→data=x;
s→next=p→next;

算法—插入操作

- 在线性单链表的*p*结点之后插入一个新的结点*x*。
- **假设：线性单链表已经建好**



插入x过程的示意图

生成结点空间s

插入的数据x放入结点空间s

p的直接后继改为新插结点s的直接后继

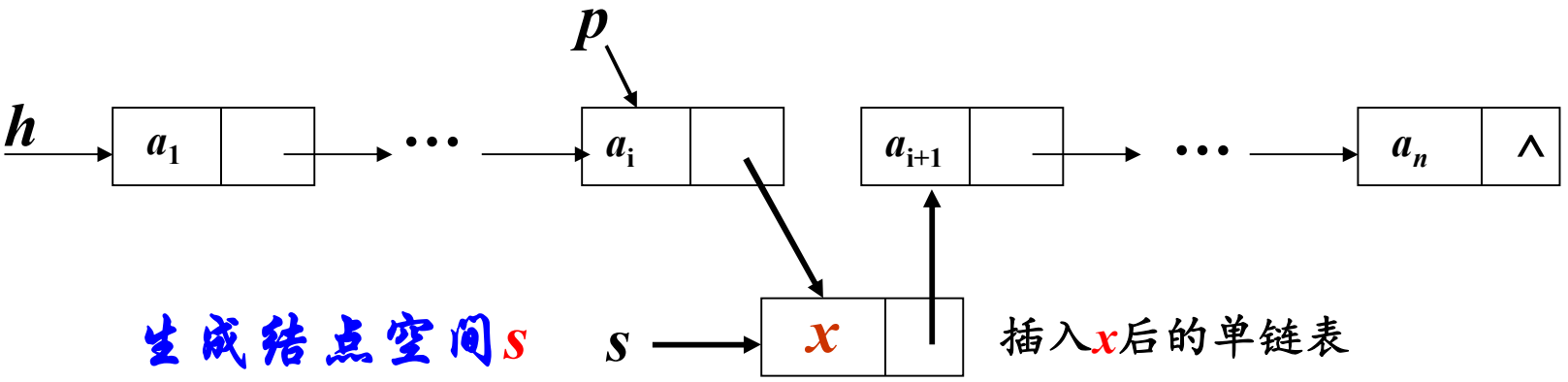
新插结点s为p的直接后继

s=(LinkList)malloc(sizeof(Node));
s→data=x;
s→next=p→next;

算法—插入操作

p→next=s;

- 在线性单链表的*p*结点之后插入一个新的结点*x*。
- **假设：线性单链表已经建好**



生成结点空间s

插入的数据x放入结点空间s

p的直接后继改为新插结点s的直接后继

新插结点s为p的直接后继

算法—插入操作

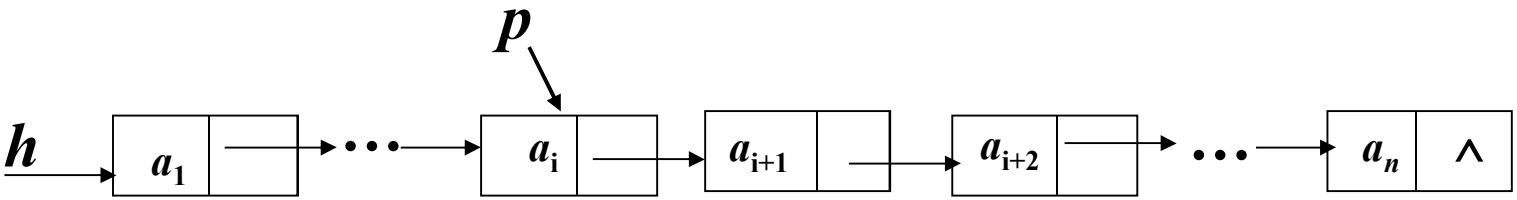
```
void insert(LinkList &p;int x)
//在线性单链表的p结点之后插入一个新的结点x
{  LinkList s;
   s=(LinkList)malloc(sizeof(Node));//生成结点空间s
   s→data=x;//插入的数据x放入结点空间s
   s→next=p→next; //p的直接后继为新插结点s的直接后继
   p→next=s; //新插结点s为p的直接后继
}
```

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*结点的直接后继结点是否存在?

假设：线性单链表已经建好



删除p结点的直接后继前的单链表

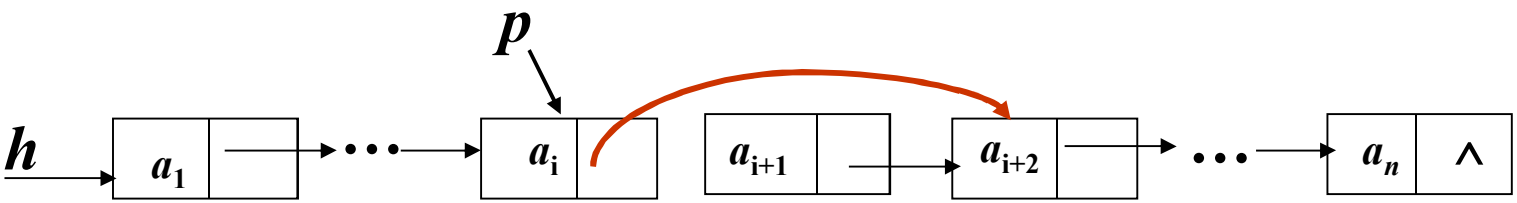
$a_1, a_2, \ldots\ldots, a_{n-1}$存在直接后继， a_n不存在直接后继

若*p*->next==NULL则*p*是线性单链表的尾结点(a_n)不存在直接后继

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作

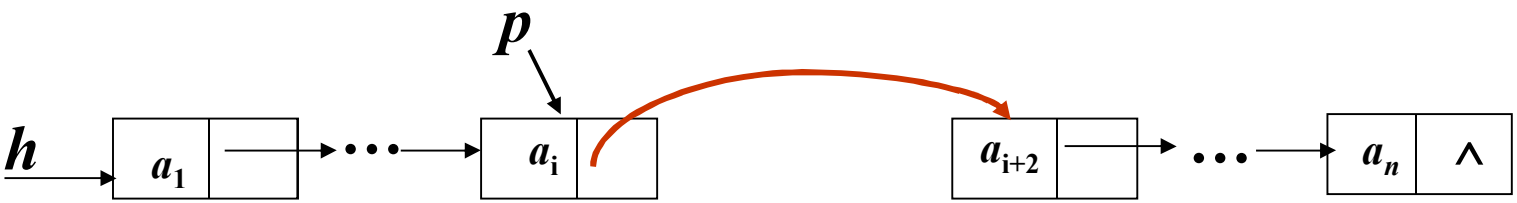


删除p结点的直接后继的过程示意图

算法—删除操作—不需要移动数据

删除线性单链表中p结点的直接后继结点。

p指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作

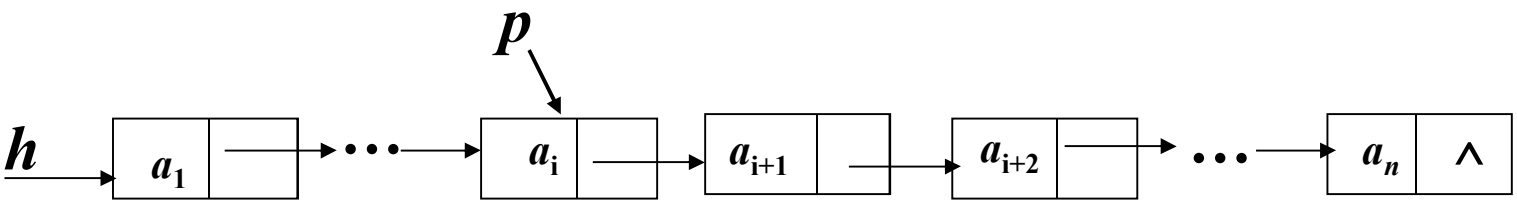


删除p结点的直接后继后的单链表

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作

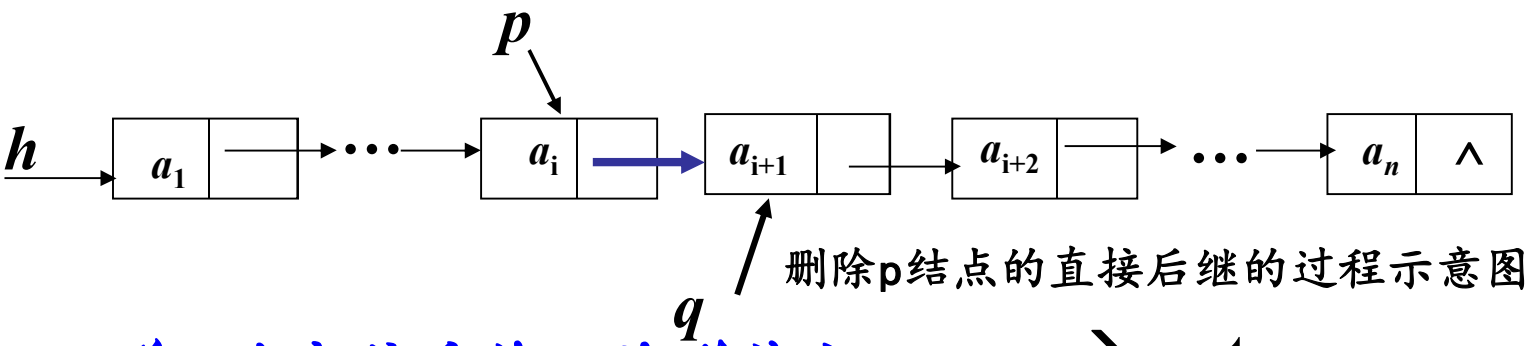


删除p结点的直接后继前的单链表

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作



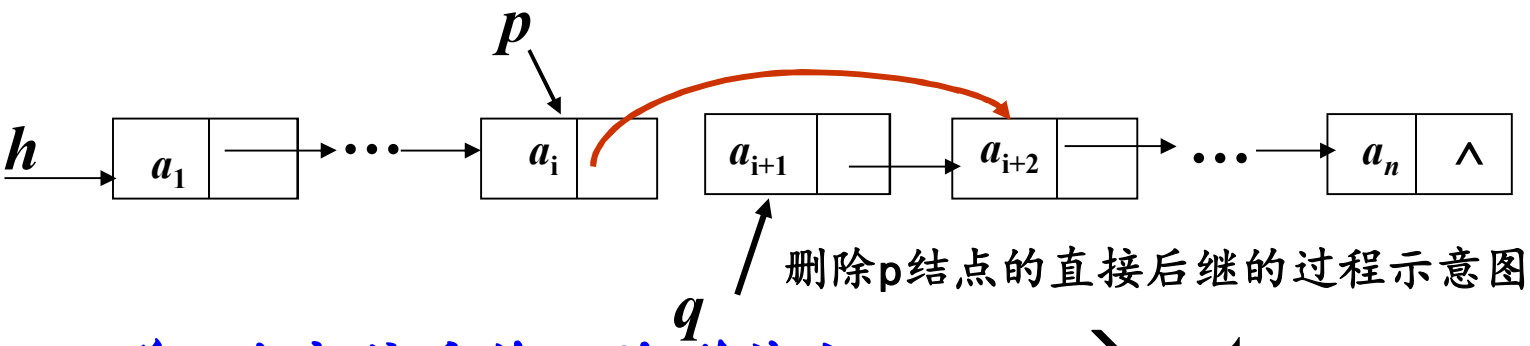
删除p结点的直接后继的过程示意图

*q*为p的直接后继—被删结点　　q=p→next;

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作



删除p结点的直接后继的过程示意图

*q*为p的直接后继—被删结点　　q=p→next;

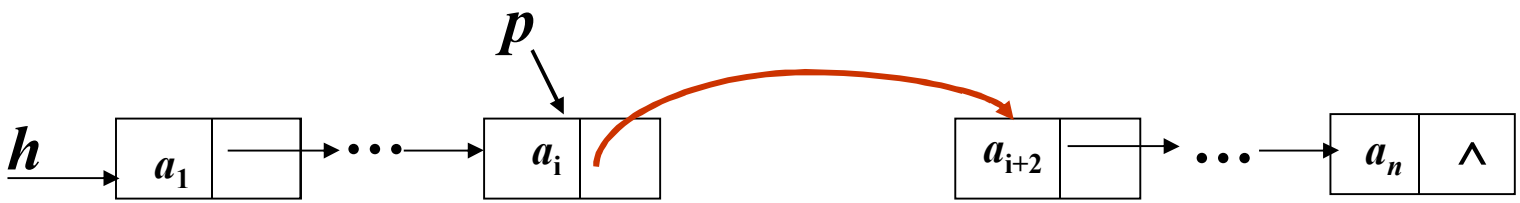
*q*的直接后继改为p的直接后继　　p→next=q→next;

释放*q*的空间　　free(q);

算法—删除操作

删除线性单链表中*p*结点的直接后继结点。

*p*指向$a_1,a_2,\ldots\ldots,a_{n-1}$存在直接后继，进行删除操作



删除p结点的直接后继后的单链表

*q*为p的直接后继—被删结点　　q=p→next;

*q*的直接后继改为p的直接后继　　p→next=q→next;

释放*q*的空间　　free(q);

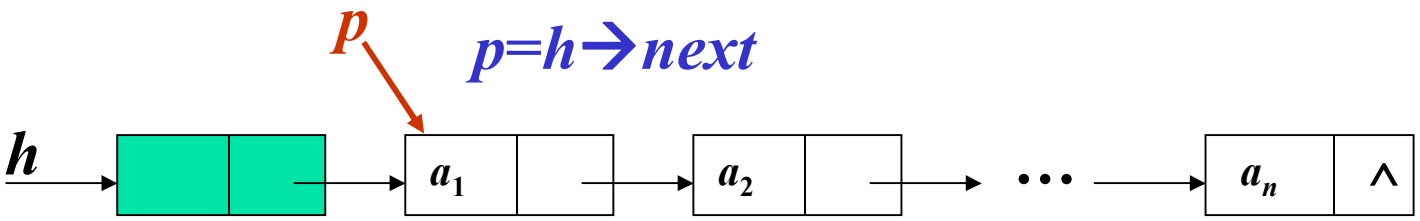
算法—删除操作

```
void delete(LinkList &p)
//删除线性单链表中p结点的直接后继结点
{  LinkList q;
   if(p→next)//p结点的直接后继结点是否存在？
   {   q=p→next;//q为p的直接后继—被删结点
       p→next=q→next;//q的直接后继改为p的直接后继
       free(q);//释放q的空间
   }
}
```

插入和删除均不需要移动数据

算法—查找操作

在头指针为*h*的**带表头**结点的单链表中查找是否存在值为*x*的结点。**线性单链表已经建好**



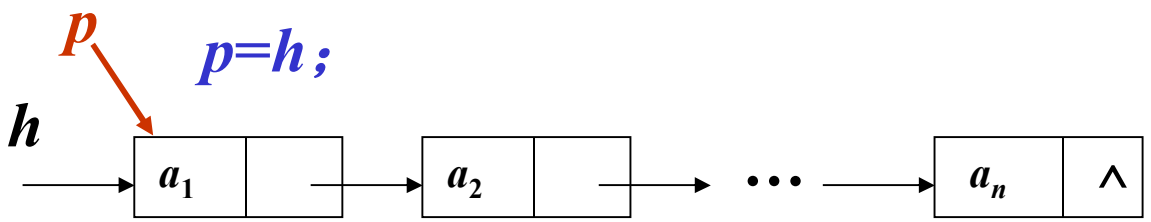
从表中第一个数据元素开始顺次比较直到找到*x*，或找到表尾

算法—查找操作

```
LinkList search(LinkList h, int x)
//在头指针为h的带表头结点的单链表中查找是否存在值为x的结点
{ LinkList p;
  p=h→next;//线性表第一个数据元素结点地址
  while(p!=NULL)
    if(p→data==x) return p;//查找成功，返回x所在结点的地址
    else p=p→next; //尚未找到，在下一个结点继续查找
  return NULL; //查找失败，返回空指针
}
```

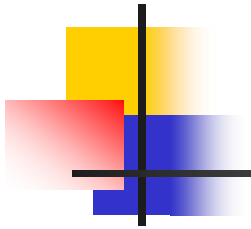
算法—查找操作

在头指针为h的**不带表头**结点的单链表中查找是否存在值为x的结点。



算法—查找操作

```
LinkList search(LinkList h,int x)
//在头指针为h的不带表头结点的单链表中查找是否存在值为x的结点
{ LinkList p;
  p=h; //线性表第一个数据元素结点地址
  while(p!=NULL)
    if(p→data==x) return p;//查找成功，返回x所在结点的地址
    else p=p→next; //尚未找到，在下一个结点继续查找
  return NULL; //查找失败，返回空指针
}
```

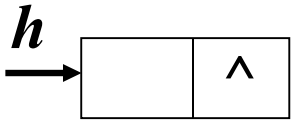


算法—建立单链表

- 单链表的建立可以从一个空表开始，通过插入操作完成，通常2种方法：
  - 首插法
  - 尾插法

算法—首插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继

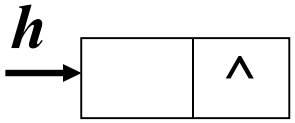


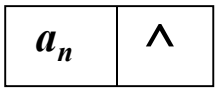
建立一个带表头结点的空链表

```
h =(LinkList)malloc(sizeof(Node));
   h→next=NULL;
```

算法—首插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继



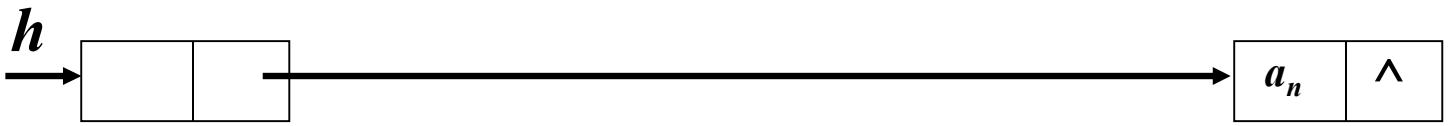


读入a_n,建立结点存放其值

将其插做表头结点的直接后继

算法—首插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继



读入a_n,建立结点存放其值

将其插做表头结点的直接后继

算法—首插法建立

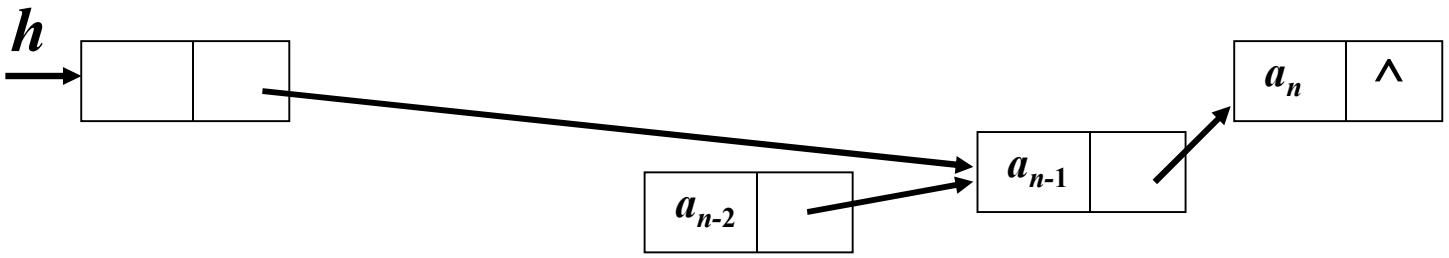
- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继

读入a_{n-1},建立结点存放其值

将其插做表头结点的直接后继

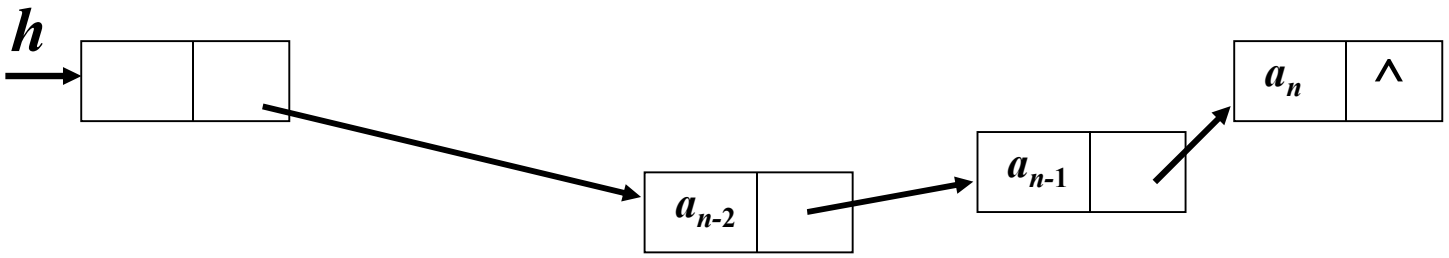
算法—首插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继



算法—首插法建立

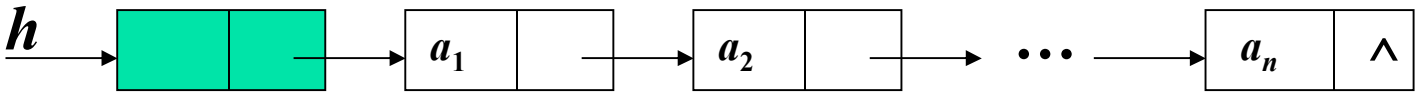
- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表中，做表头结点的直接后继



算法—首插法建立

输入数据顺序：

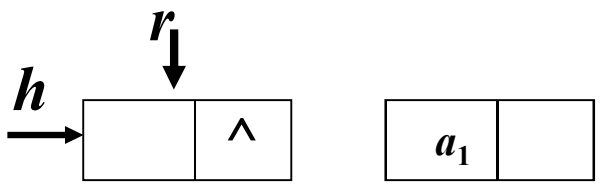
$a_n, a_{n-1}, \cdots, a_{i+1}, a_i, a_{i-1}, \cdots, a_2, a_1$



```
void creat1(LinkList &h)
//首插法建立头指针为h的带表头结点的单链表
{ LinkList p;  int x, int i, n;
   h =(LinkList)malloc(sizeof(Node));//申请表头结点空间
   h→next=NULL;//建立一个带表头结点的空表
   scanf("%d",&n); //读入要建立的单链表的数据元素个数
   for(i=1; i<=n; i++)//依次读入n个数据元素插入表中
   {  scanf("%d",&x);//读入数据元素
      p=(LinkList)malloc(sizeof(Node));//申请存放数据元素的空间
      p→data=x;//读入的值放入所申请空间的data域
      p→next=h→next;//插入链表-原先表头结点的直接后继作为新插结点的直接后继
      h→next=p;//新插结点作为表头结点的直接后继
   }
}
```

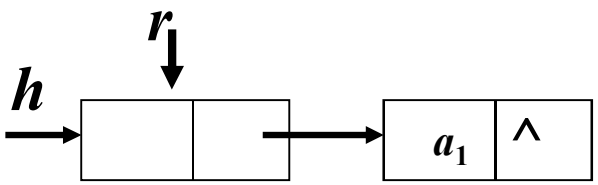
算法—尾插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表尾



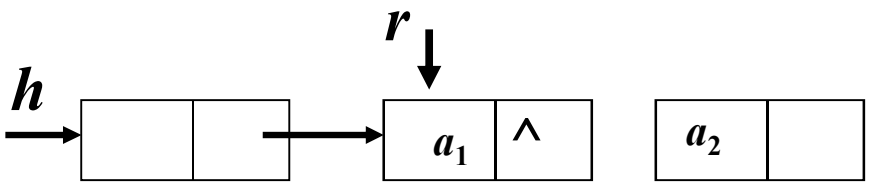
算法—尾插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表尾



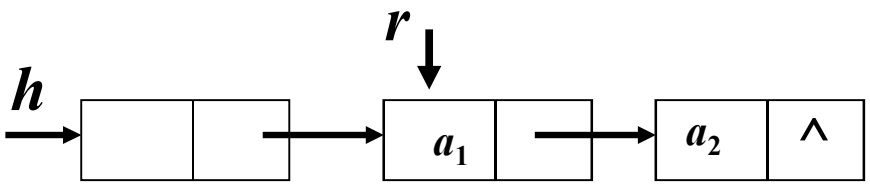
算法—尾插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表尾



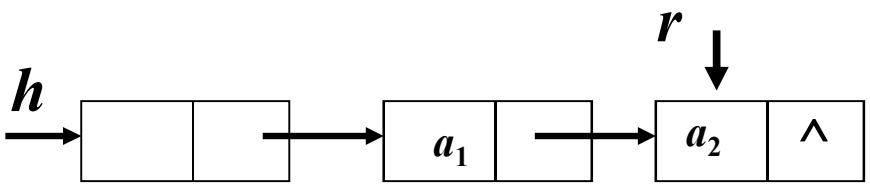
算法—尾插法建立

- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表尾



算法—尾插法建立

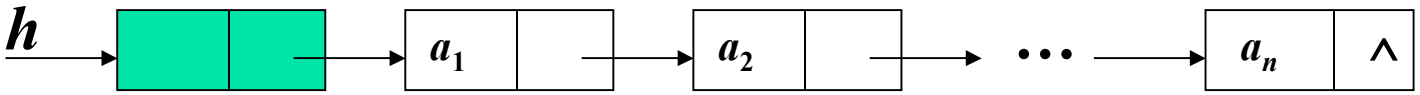
- 建立一个带表头结点的空链表
- 依次读入线性表中的数据元素，将其插入到表尾



算法—尾插法建立

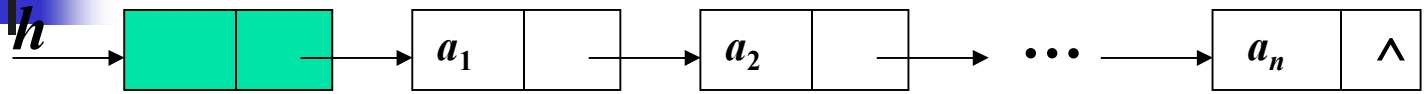
输入数据顺序：

$a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_{n-1}, a_n$

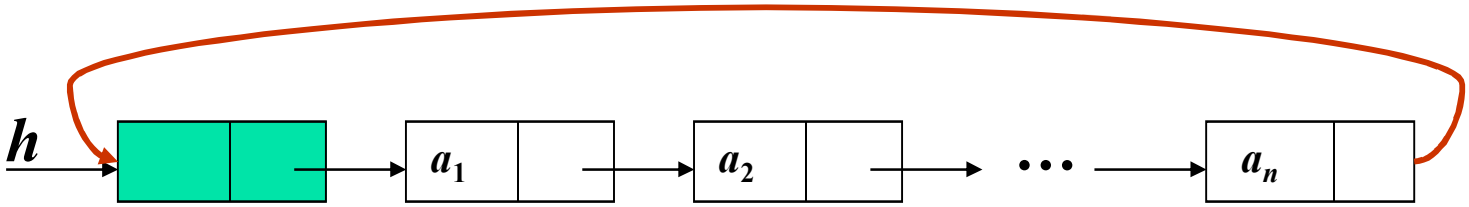


```
void creat2(LinkList &h)
//尾插法建立头指针为h的带表头结点的单链表
{ LinkList p,r;  int x, i, n;
   h =(LinkList)malloc(sizeof(Node)); //申请表头结点空间
   h→next=NULL; //建立一个带表头结点的空表
   r=h;//设置单链表的尾节点r为当前链表的头结点，因为空表只有表头结点
  scanf("%d",&n); //读入要建立的单链表的数据元素个数
   for(i=1;i<=n;i++) //依次读入n个数据元素插入表中
   {   scanf("%d",&x); //读入数据元素
       p=(LinkList)malloc(sizeof(Node)); //申请存放数据元素的空间
        p→data=x; /读入的值放入所申请空间的data域
        p→next=NULL; //新插入的数据作为链表尾结点，加尾结点标记
        r→next=p; //新插入链表为原先尾结点r的直接后继
        r=p;//新插入的结点尾插入后的尾结点
    }
}
```

循环单链表



带表头结点的单链表

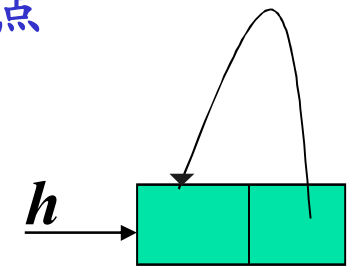


带表头结点的循环单链表

将单链表的尾结点的指针强行指向单链表的头结点

特点：

1. 从表中任一结点出发均能找到表中所有结点
2. *p*结点为尾结点的条件：*p→next ==h*
3. 循环单链表为空表的判断条件：*h→next ==h*



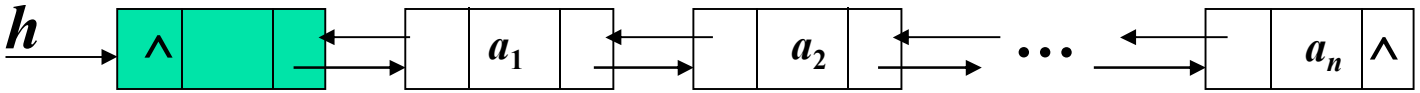
空的循环单链表

双向链表

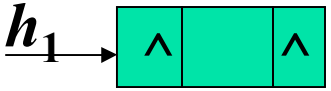
双向链表：单链表的每个结点包含2个指针，分别指向结点的直接前驱和直接后继

data：存放数据元素的值
next：存放直接后继的地址
prior：存放直接前驱的地址

双向链表结点结构示意图



带表头结点的双向链表



带表头结点的空的双向链表

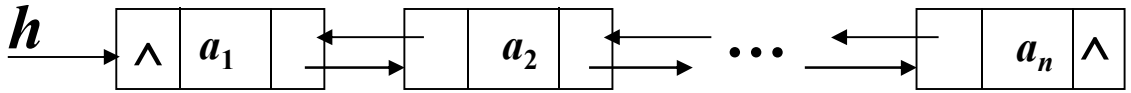
特点：1.从表中任一结点p出发均能找到表中所有结点，设p结点存放的是线性表的数据元素a_i，沿着*next*指针能找到a_{i+1}，a_{i+2}，…，a_n。沿着*prior*指针能找到a_{i-1}，a_{i-2}，…，a_1. 2. 插入和删除操作，结点的2个指针均要连接上

双向链表

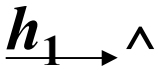
双向链表：单链表的每个结点包含2个指针，分别指向结点的直接前驱和直接后继

data：存放数据元素的值
next：存放直接后继的地址
prior：存放直接前驱的地址

双向链表结点结构示意图



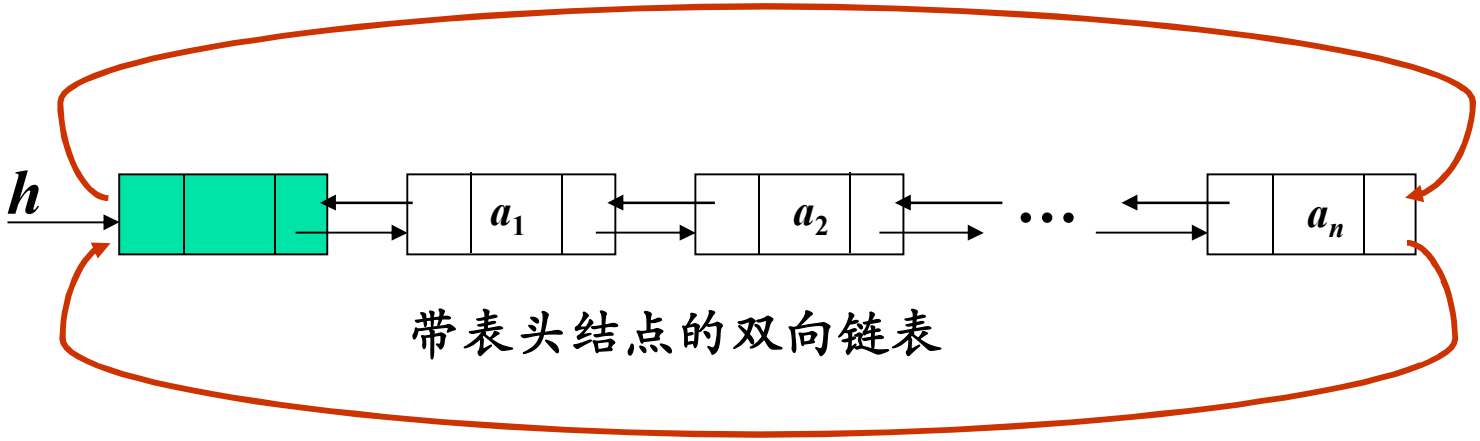
不带表头结点的双向链表



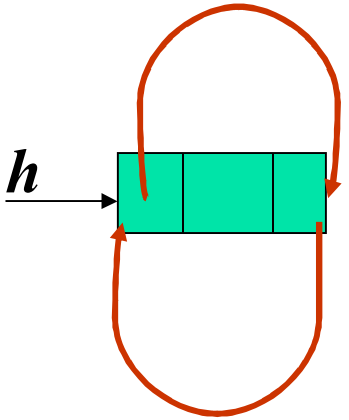
特点：1.从表中任一结点p出发均能找到表中所有结点，设p结点存放的是线性表的数据元素a_i，沿着*next*指针能找到a_{i+1}，a_{i+2}，…，a_n。沿着*prior*指针能找到a_{i-1}，a_{i-2}，…，a_1. 2. 插入和删除操作，结点的2个指针均要连接上

不带表头结点的空的双向链表

双向循环链表



带表头结点的双向链表



h→next ==h，h→prior ==h

带表头结点的空的双向链表

链式存储结构小结

- 逻辑相邻不一定物理相邻
- 只能顺序存取
- 插入和删除操作不需要移动数据
- 按值查找O(n),和顺序存储结构的按值查找速度相同
- 按数据元素的位置查找O(n),比顺序存储结构的按位置查找速度慢