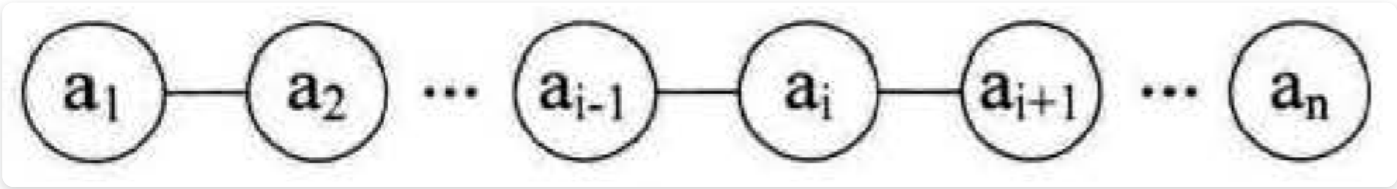
一、线性表

线性表(List):零个或多个数据元素的有限序列。所谓序列，也就是说元素之间是有序的。若元素存在多个，则第一个元素无前驱，最后一个元素无后继，其他每个元素都有且只有一个前驱和后继。同时，线性表总是强调为有限的。



enter description here

1. 线性表的顺序存储结构

线性表的顺序存储结构，指的是用一段地址连续的存储单元依次存储线性表的数据元素。线性表(a1,a2,...an)的顺序存储示意图：

线性表中的每个数据元素类型都相同，可以用一维数组实现顺序存储结构，即把第一个数据元素存到数组下标为0的位置中，接着把线性表相邻的元素存储在数组中相领的位置。

```
# define   MAXSIZE 20
typedef  int ElemType
typedef  struct{
      ElemType data[MAXSIZE]; //数组存储最大值
      int length;//线性表当前长度
}
```

描述顺序存储结构需要三个属性：

- 存储空间的起始位置:数组data，它的存储位置就是存储空间的存储位置
- 线性表的最大存储容量：数组长度MaxSize
- 线性表的当前长度：length

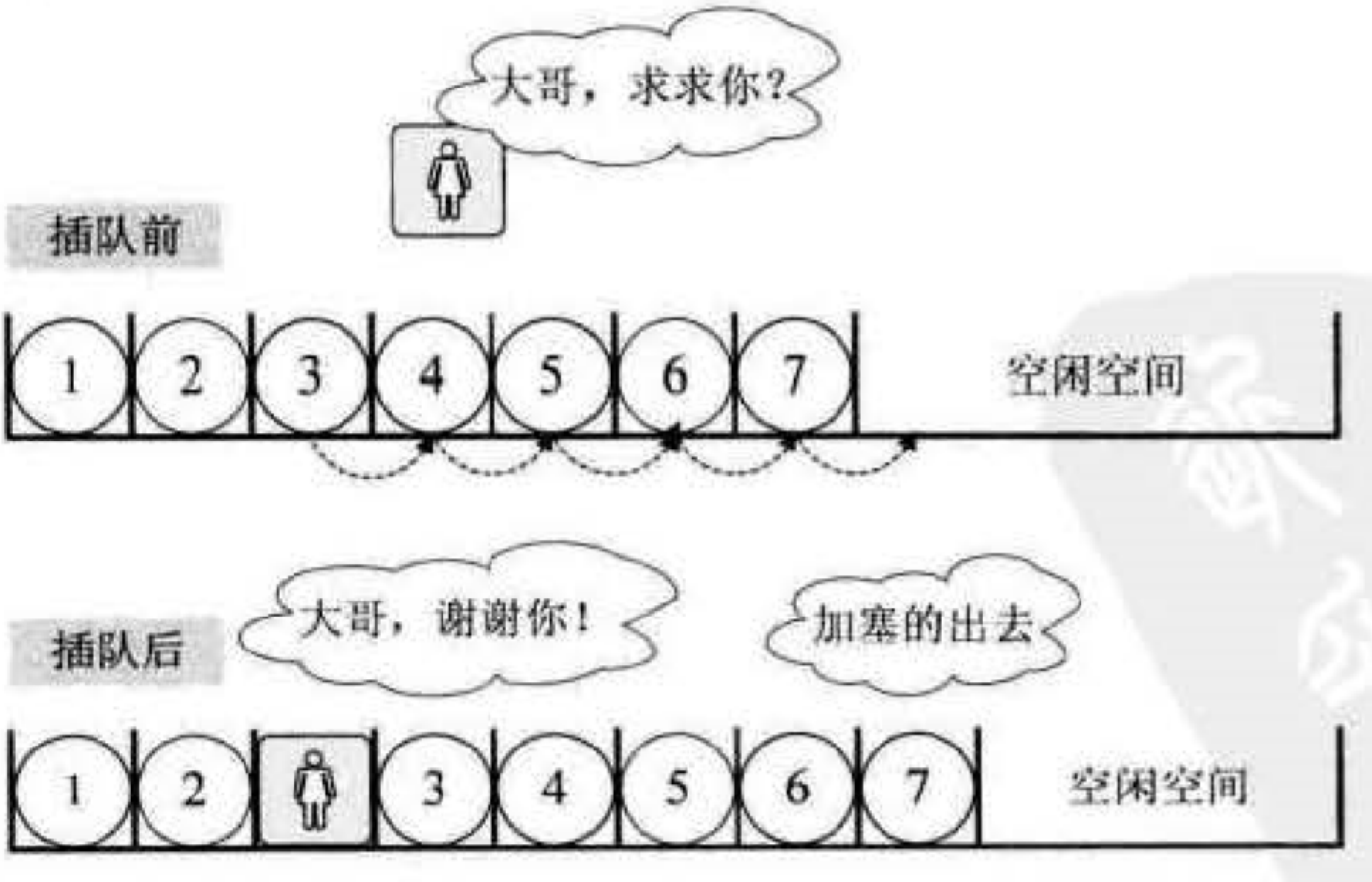
1-1. 顺序存储结构的插入与删除

1-1-1. 获得元素操作

对于线性表的顺序存储结构来说，获得元素的操作只需要将线性表L中的第i个位置的元素返回即可。就程序而言，只要i的数值在数组下标范围内，就是把数组第i-1个下标的值返回即可。

1-1-2. 插入元素操作

对于线性表的顺序存储结构来说，插入元素的操作的最坏时间复杂度为O(N)，即需要挪动N个元素。

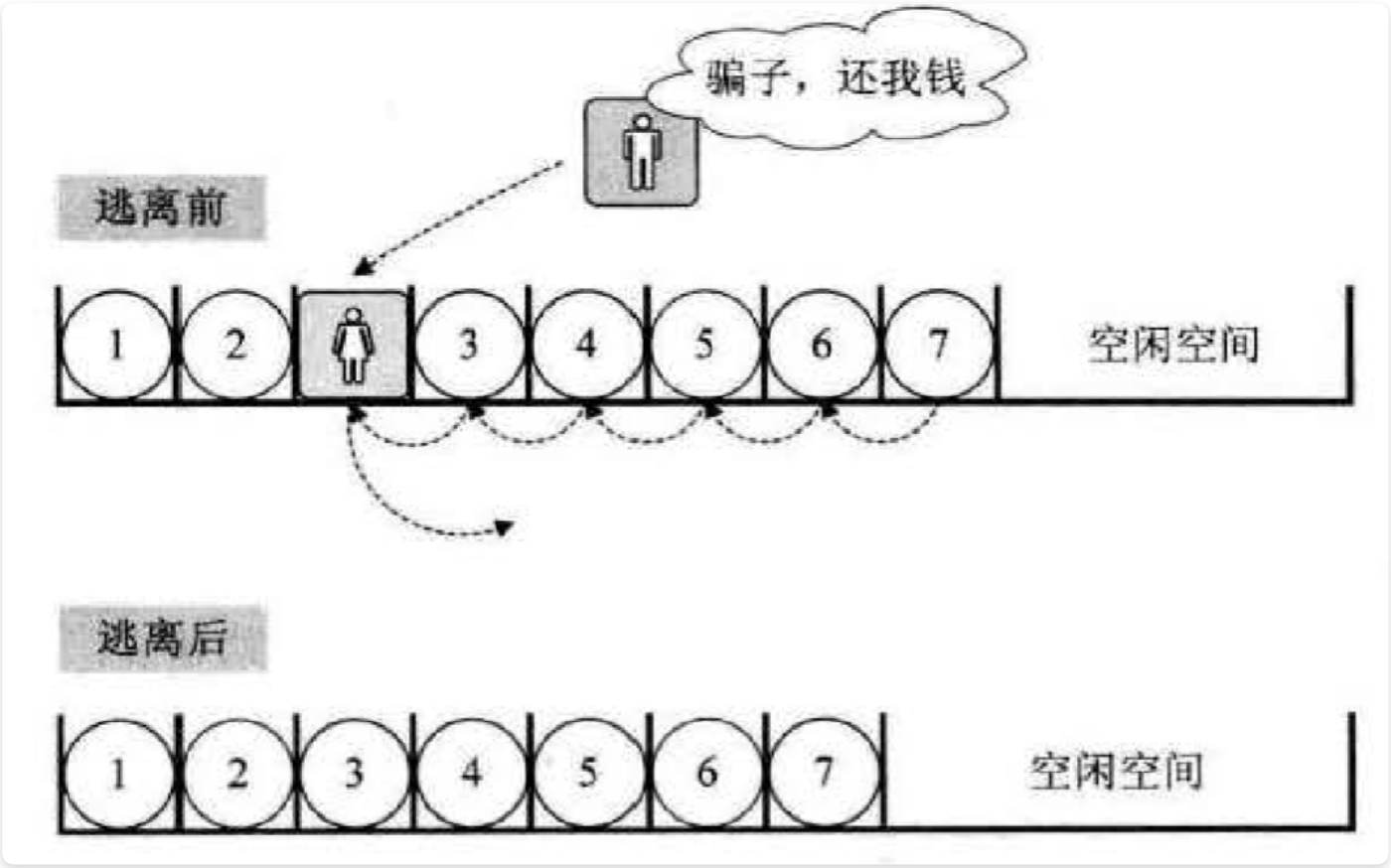


插入算法的思路：

- 如果插入位置不合理，抛出异常；
- 如果线性表的长度大于数组长度，抛出异常或动态增加容量

- 从最后一个元素向前遍历到第i个位置，分别将它们都向后移动一个位置(arr[i] = arr[i-1])
- 将要插入元素填入位置i处
- 表长+1

1-1-3. 删除操作



enter description here

删除算法的思路：

- 如果删除的位置不合理，抛出异常
- 取出删除元素
- 从删除元素位置开始遍历到最后一个元素，分别将它们都向前移动一个位置
- 表长-1

1-1-4. 分析插入和删除的时间复杂度

先来看最好的情况，如果元素要插入到最后一个元素，或者删除最后一个元素，此时时间复杂度为O(1)。
最坏的情况，如果元素要插入到第一个位置或者删除第一个位置，此时时间复杂度为O(n)

至于平均的情况，由于元素插入到第i个位置，或删除第i个位置，需要移动n-i个元素。根据概率原理，每个位置插入或删除元素的可能性是相同的，也就是位置靠后，移动元素少；位置靠前，移动位置多/最终平均移动次数和最中间的那个元素的移动次数相等，为(n-1)/2

根据时间复杂度的推导，不考虑系数和常数，平均时间复杂度还是O(n)
这说明线性表的顺序存储结构更适合读取元素，不适合对元素进行插入或删除操作。

2. 线性表的链式存储结构

线性表的链式存储结构通过链表进行实现。

3. 单链表

3-1. 单链表的读取

获得链表第i个数据的算法思路：

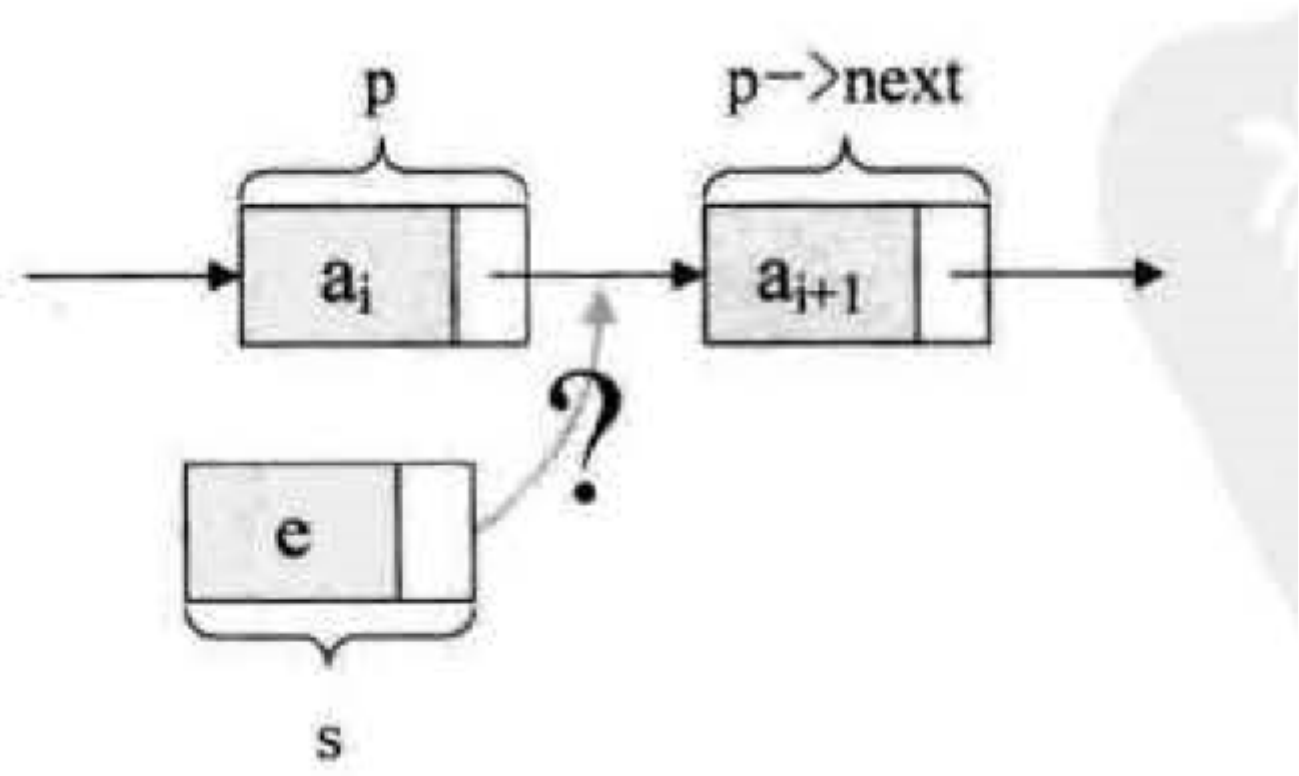
1. 声明一个结点current指向链表第一个结点，初始化j从1开始
2. 当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j+1
3. 若到链表末尾p为空，则说明第i个元素不存在
4. 否则查找成功，返回结点p的数据

```
  private Node find(Node head,int i){
          int j = 1;
          Node current = head;
          while(current != null && j < i){
                current =current.next;
                j++;
          }
          if( current == null || j < i){
                return null;
          }
          return current;
    }
```

3-2. 单链表的插入与删除

3-2-1. 单链表的插入

假设存储元素e的结点为s，要实现结点s插入结点p之后，只需将结点s插入到结点p和结点p.next之间即可




```
s.next = p.next;
p.next = s;
```

单链表第i个数据插入结点的算法思路：

1. 遍历链表，若到链表末尾p为空，说明第i个元素不存在
2. 否则查找成功，在系统中生成一个空结点s
3. 将数据元素e赋值给s.data
4. 单链表的插入标准语句：s.next = p.next; p,next = s;

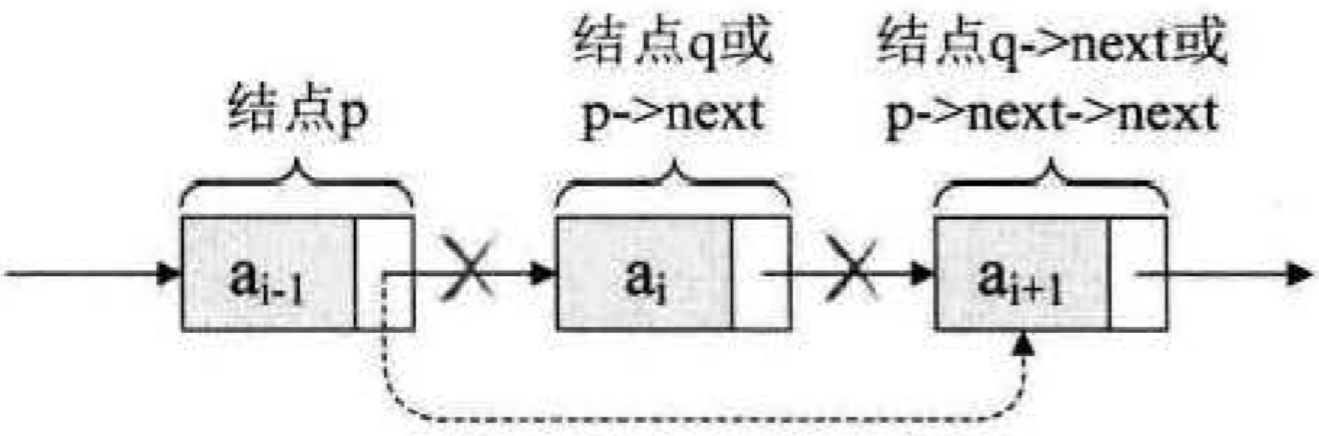
```
/**
     * 从任意位置添加
     * @param data
     * @param index
     * @return
     */
    public boolean insert (T data , int index){
        if(index < 0 || index >size){
            throw new IllegalArgumentException("index error");
```

```
            }
            //即将添加的新节点
            Node<T> node = new Node<>(data);
            Node<T> temp = null;
            //index为0,则是添加到链表头部
            if(index == 0){
                  //先判断head是否存在
                  if(head == null){
                        head = node;
                        return true;
                  }else{
                        //将节点置为新的头部
                        node.next = head;
                        head = node;
                        return true;
                  }
            }else{
                  if(index == size){
                        this.insert(data);
                        return true;
                  }else{
                        //遍历到要插入所在位置的指针
                        temp = head;
                        for(int i=1; i<index ;i++){
                            temp = temp.next;
                        }
                        //标准插入
                        node.next = temp.next;
                        temp.next = node;
                        return true;
                  }
            }

      }
```

3-2-2. 单链表的删除

设存储元素ai的结点为q，要实现将结点q删除单链表的操作，其实就是将它的前继结点的指针绕过，指向它的后继结点即可。



删除实际实际上就是，p.next = p.next.next；用q来取代p.next,即是让p的后继的后继结点改成p的后继结点

```
q = p.next;
p.next = q.next;
```

单链表第i个数据删除结点的算法思路：

1. 遍历链表到要删除结点的前一个结点，若到链表末尾为空，则说明第i个元素不存在
2. 否则查找成功，将欲删除的结点的前一个结点设为p，即将被删除结点为q，将p.next赋值给q
3. 单链表的删除标准语句p.next = q.next
4. 将q结点中的数据返回
5. 释放q结点

```
public boolean remove(int index){
        if(index < 0 || index > size){
            throw new IllegalArgumentException("index error");
        }
        Node<T> preNode = null;
        Node<T> current = null;
        //删除头节点
        if(index == 0){
            //判断链表长度
            if(head.next == null){
                head = null;
            }else{
                head = head.next;
            }
            return true;
        }

        int i = 1;
        preNode = head;
        current = preNode.next;
        while(current != null){
            if(i == index){
                //略过将删除的节点
                preNode.next = current.next;
                return true;
            }
            preNode = current;
            current = current.next;
            i++;
        }
        return false;
    }
```

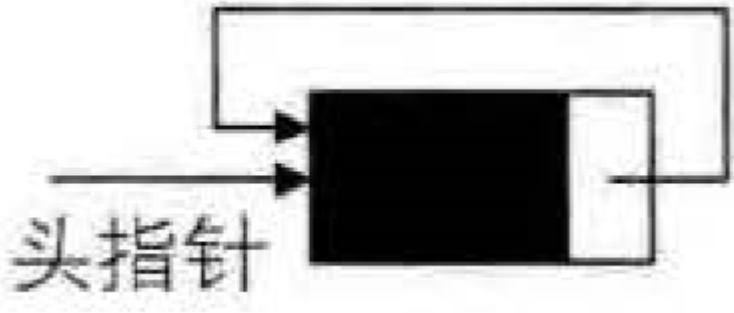
3-2-3. 分析插入和删除的时间复杂度

对于线性表的链式存储结构,插入和删除操作都是由两部分组成的：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。它们的时间复杂度都是O(n）。如果我们不知道第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，对于顺序存储结构来说，每一次插入到需要移动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)，接下老只是简单地通过赋值移动指针，时间复杂度都是O(1). 显然，对于插入和删除数据越频繁地操作，单链表的效率优势越明显。

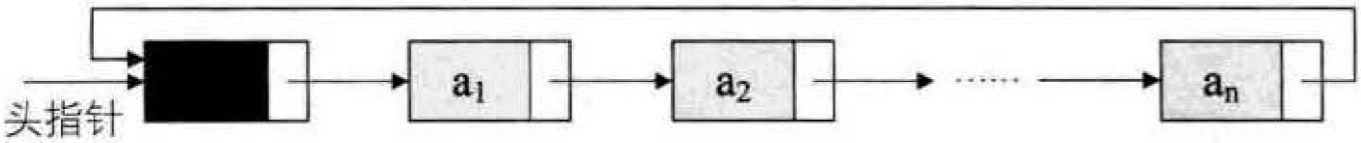
4. 循环链表

将单链表中终端结点的指针端由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表称为单循环链表，简称循环链表。
循环链表解决了一个很麻烦的问题：如何从当中一个结点出发，访问到链表的全部结点，而不是每次都要从链表头结点开始才能访问全部结点

为了使空链表与非空链表处理一致，我们通常设一个头结点。
循环链表带有头结点的空链表：



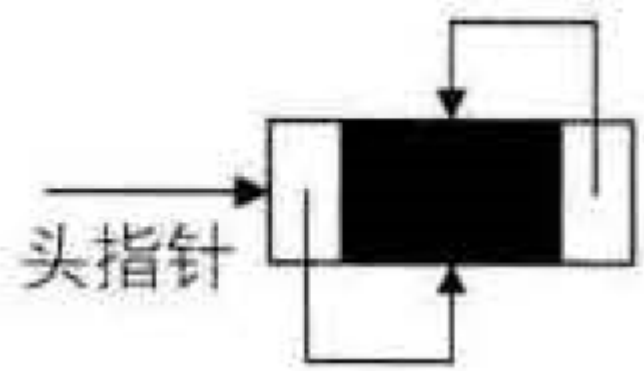
对于非空的循环链表：



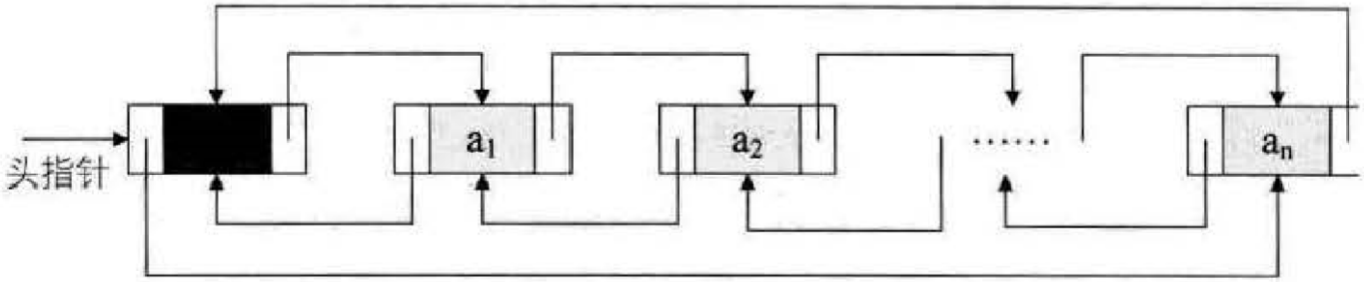
其实循环链表和单链表的主要差异在于循环的判断条件上，原来是判断p.next是否为空，现在则是判断p.next是否等于头结点

5. 双向链表

双向链表是在单链表的每个结点中，再设置一个指向其前驱结点的指针。双向链表中的结点都有两个指针域，一个指向后继，另一个指向前驱。
双向链表中也可以是循环链表。
双向链表的循环带头结点的空链表：

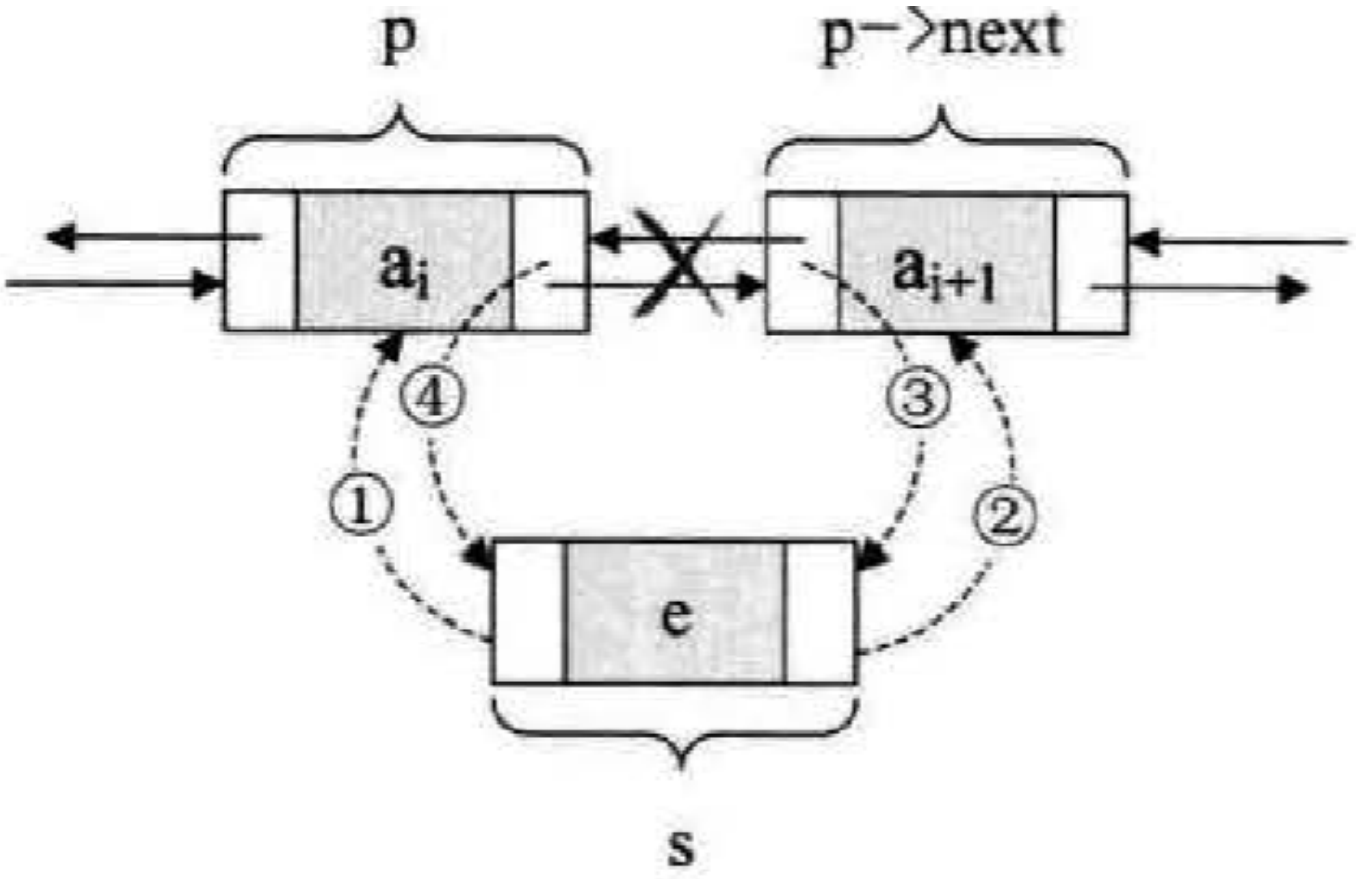


非空的循环的带头结点的双向链表：



5-1. 插入操作

双向链表的插入操作要注意顺序：
假设存储元素e的结点为s，要实现将结点s插入到结点p和p.next：

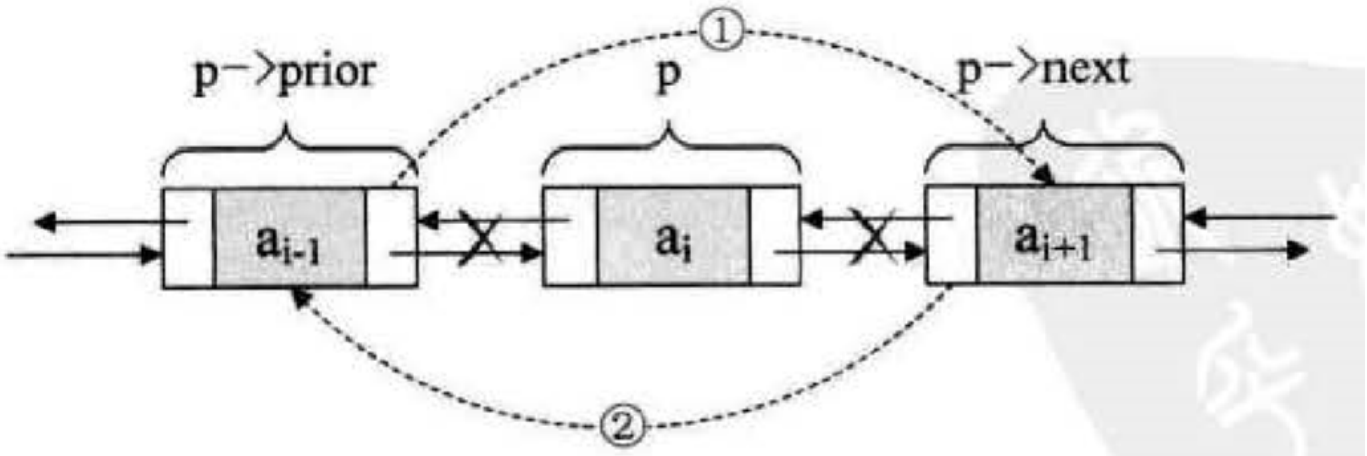



```
s.pre = p; // 1
s.next = p.next; // 2
p.next .pre = s;// 3
p.next = s;// 4
```

顺序是先搞定s的前驱和后继，再搞定后结点的前驱，最后解决前结点的后继 (由于第二步和第三步都使用了p.next，如果第四步先执行,则会使得p.next提前变成s)

5-2. 删除操作

删除结点p，只需要下面两步骤:



```
p.pre.next  =  p.next;
p.next.pre  =  p.pre;
```