链表

链表和顺序表统称为线性表：**物理储存**

区别是：

- 顺序表：连续的储存
- 链表：离散的储存

为什么需要链表

顺序表的构建需要预先知道数据大小来申请连续的存储空间，而在进行扩充时又需要进行数据的搬迁，所以使用起来并不是很灵活。

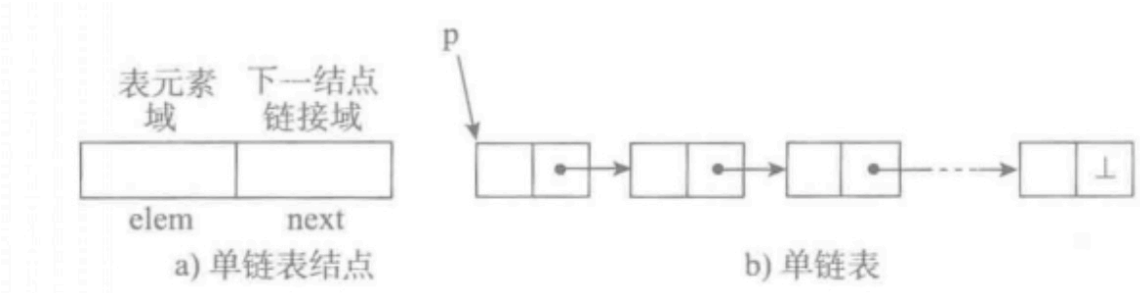
链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。

链表的定义

链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是不像顺序表一样连续存储数据，而是在每一个节点（数据存储单元）里存放下一个节点的位置信息（即地址）。

单向链表

单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个**信息域（元素域）**和一个**链接域**。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。



P表示第一个节点的位置

- 变量p指向链表的头节点（首节点）的位置，从p出发能找到表中的任意节点。
- 表元素域elem用来存放具体的数据。
- 链接域next用来存放下一个节点的位置（python中的标识）

单链表的操作

> is_empty() 链表是否为空
>
> length() 链表长度
>
> travel() 遍历整个链表
>
> add(item) 链表头部添加元素
>
> append(item) 链表尾部添加元素
>
> insert(pos, item) 指定位置添加元素
>
> remove(item) 删除节点
>
> search(item) 查找节点是否存在

注意:Python 中的变量名保存的是对象的地址 而不是对象本身

```
# 节点实现

class SingleNode(object):
  """单链表的结点"""
  def __init__(self, item):
    # item存放数据元素
    self.item = item
    # next是下一个节点的标识,因未知所以设为空None
    self.next = None
```

```
#单链表的实现

class SingleLinkList(object):
  "单链表"
  def __init__(self, node=None):
    "链表头"
    self.__head = None
  #_表示私有

  def is_empty(self):
    "链表是否为空"
    return self.__head == None
```

```
    #将head是否指向None作为return的结果

  def length(self):
    "链表长度"
    # cur初始时指向头节点，用来移动遍历节点
    cur = self.__head
    count = 0
    # 尾节点指向None，当未到达尾部时
    while cur != None:
        count += 1
        # 将cur后移一个节点
        cur = cur.next
    return count


  def travel(self):
    "遍历整个链表"
    cur = self.__head
    while cur != None:
      print(cur.item, end=" ")
      cur = cur.next
    print("") #换行

  def append(self, item):
    "链表尾部添加元素"
    node = SingleNode(item)
    # 先判断链表是否为空，若是空链表，则将_head指向新节点node
    if self.is_empty():
      self.__head = node
    # 若不为空，则找到尾部，将尾节点的next指向新节点
    else:
      cur = self.__head
      while cur.next != None:
        cur = cur.next
      cur.next = node

  def add(self, item):
    "链表头部添加元素"
    # 先创建一个保存item值的节点
    node = SingleNode(item)
    # 将新节点的链接域next指向头节点，即_head指向的位置
    node.next = self.__head
    # 将链表的头_head指向新节点
```

```
    self.head = node

  def insert(self, pos, item):
    "指定位置添加元素"
    if pos <= 0:
      self.add(item)
    elif pos > (self.length()-1):
      self.append(item)
    else:
      node = SingleNode(item)
      # pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置
      pre = self.__head
      count = 0
      while count < (pos-1):
        count += 1
        pre = pre.next
      # 先将新节点node的next指向插入位置的节点
      node.next = pre.next
      # 将插入位置的前一个节点的next指向新节点
      pre.next = node
      #不是pre = node，是pre节点的next区域变成node

  def search(self,  item):
    "查找节点是否存在，并返回True或者False"
    node = SingleNode(item)
    cur = self.__head
    while cur != None:
      if cur == node:
        return True
      else:
        cur = cur.next
    return False

  def remove(self, item):
    "删除节点"
    cur = self.__head
    pre = None

    while cur != None:
      if cur.item == item:
        # 如果第一个就是删除的节点
        if cur == self.__head:
          # 将头指针指向头节点的后一个节点
```

```
            self.__head = cur.next
          else:
            # 将删除位置前一个节点的next指向删除位置的后一个节点
            pre.next = cur.next
          break
        else:
          pre = cur
          cur = cur.next

```

```
#----测试-----
# 节点
class SingleNode(object):
  """单链表的结点"""
  def __init__(self, item):
    # item存放数据元素
    self.item = item
    # next是下一个节点的标识,因未知所以设为空None
    self.next = None


# 单链表
class SingleLinkList(object):
  "单链表"
  def __init__(self, node=None):
    "链表头"
    self.__head = None
  #_表示私有

  def is_empty(self):
    "链表是否为空"
    return self.__head == None
    #将head是否指向None作为return的结果

  def length(self):
    "链表长度"
    # cur初始时指向头节点，用来移动遍历节点
    cur = self.__head
    count = 0
    # 尾节点指向None，当未到达尾部时
    while cur != None:
        count += 1
        # 将cur后移一个节点
```

```
        cur = cur.next
    return count

  def travel(self):
    "遍历整个链表"
    cur = self.__head
    while cur != None:
      print(cur.item, end=" ")
      cur = cur.next
    print("")

  def append(self, item):
    "链表尾部添加元素"
    node = SingleNode(item)
    # 先判断链表是否为空，若是空链表，则将_head指向新节点node
    if self.is_empty():
      self.__head = node
    # 若不为空，则找到尾部，将尾节点的next指向新节点
    else:
      cur = self.__head
      while cur.next != None:
        cur = cur.next
      cur.next = node

  def add(self, item):
    "链表头部添加元素"
    # 先创建一个保存item值的节点
    node = SingleNode(item)
    # 将新节点的链接域next指向头节点，即_head指向的位置
    node.next = self.__head
    # 将链表的头_head指向新节点
    self.__head = node

  def insert(self, pos, item):
    "指定位置添加元素"
    if pos <= 0:
      self.add(item)
    elif pos > (self.length()-1):
      self.append(item)
    else:
      node = SingleNode(item)
      # pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置
      pre = self.__head
```

```
        count = 0
        while count < (pos-1):
            count += 1
            pre = pre.next
        # 先将新节点node的next指向插入位置的节点
        node.next = pre.next
        # 将插入位置的前一个节点的next指向新节点
        pre.next = node
        #不是pre = node，是pre节点的next区域变成node

    def search(self,  item):
        "查找节点是否存在，并返回True或者False"
        cur = self.__head
        while cur != None:
            if cur.item == item:
                return True
            else:
                cur = cur.next
        return False

    def remove(self, item):
        "删除节点"
        cur = self.__head
        pre = None

        while cur != None:
            if cur.item == item:
                # 如果第一个就是删除的节点
                if cur == self.__head:
                    # 将头指针指向头节点的后一个节点
                    self.__head = cur.next
                else:
                    # 将删除位置前一个节点的next指向删除位置的后一个节点
                    pre.next = cur.next
                break
            else:
                pre = cur
                cur = cur.next


#测试
if __name__ == "__main__":
    ll = SingleLinkList()
```

```
    print(ll.is_empty())
    print(ll.length())

    ll.append(1)
    print(ll.is_empty())
    print(ll.length())
    ll.add(9)
    ll.append(2)
    ll.append(3)
    ll.append(4)
    ll.insert(-1,100)
    ll.travel()
    ll.insert(100,200)
    ll.travel()
    ll.insert(3,8)
    ll.travel()
    ll.search(100)
    ll.remove(100)
    ll.travel()
    ll.remove(1)
    ll.travel()
    ll.remove(200)
    ll.travel()

```

```
True
0
False
1
100 9 1 2 3 4
100 9 1 2 3 4 200
100 9 1 8 2 3 4 200
9 1 8 2 3 4 200
9 8 2 3 4 200
9 8 2 3 4
```

链表与顺序表的对比

操作	链表	顺序表
访问元素 search	O(n)	O(1)
在头部插入/删除 add	**O(1)**	**O(n)**
在尾部插入/删除 append	**O(n)**	**O(1)**
在中间插入/删除 insert	O(n)	O(n)

while循环：O(n), 链表只记录头节点，访问需要遍历

注意虽然表面看起来复杂度都是 O(n)，但是链表和顺序表在插入和删除时进行的是完全不同的操作。链表的主要耗时操作是遍历查找，删除和插入操作本身的复杂度是O(1)。顺序表查找很快，主要耗时的操作是拷贝覆盖。因为除了目标元素在尾部的特殊情况，顺序表进行插入和删除时需要对操作点之后的所有元素进行前后移位操作，只能通过拷贝和覆盖的方法进行。

顺序表：

- 优点：存取元素的时候复杂度O(1)，一次性并类
- 缺点：需要连续的整块内存
- 在中间插入/删除 insert O(n):花费在数据搬迁

链表：

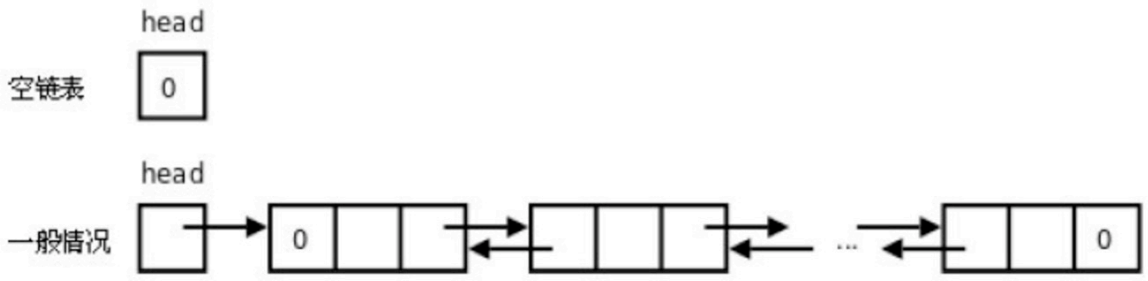
- 优点：可以使用分散的可用内存
- 缺点：储存空间需要更大，除了原有数据还要存地址指针，存取元素的时候复杂度O(n)
- 在中间插入/删除 insert O(n):花费在遍历

双向链表

双向链表结构分析

一种更复杂的链表是“双向链表”或“双面链表”。每个节点有两个链接：一个指向前一个节点，当此节点为第一个节点时，指向空值；而另一个指向下一个节点，当此节点为最后一个节点时，指向空值。

除了后继节点：.next, 多一个前驱节点：.prev



双链表操作

> is_empty() 链表是否为空
>
> length() 链表长度
>
> travel() 遍历链表
>
> add(item) 链表头部添加
>
> append(item) 链表尾部添加
>
> insert(pos, item) 指定位置添加
>
> remove(item) 删除节点
>
> search(item) 查找节点是否存在

```
# 节点实现

class Node(object):
  """双链表的结点"""
  def __init__(self, item):
    # item存放数据元素
    self.item = item
    # 后继节点 next
    self.next = None
    #前驱节点 prev
    self.prev = None
```

```
# 双向链表的实现

class DoubleLinkList(object):
```

```
  "双链表"
  def __init__(self, node=None):
    "链表头"
    self.__head = None

  def is_empty(self):
    "链表是否为空--一样"
    return self.__head is None
    #将head是否指向None作为return的结果

  def length(self):
    "链表长度--一样"
    # cur初始时指向头节点，用来移动遍历节点
    cur = self.__head
    count = 0
    # 尾节点指向None，当未到达尾部时
    while cur != None:
        count += 1
        # 将cur后移一个节点
        cur = cur.next
    return count

  def travel(self):
    "遍历整个链表--一样"
    cur = self.__head
    while cur != None:
      print(cur.item, end=" ")
      cur = cur.next
    print("")

  def search(self,  item):
    "查找节点是否存在，并返回True或者False--一样"
    cur = self.__head
    while cur != None:
      if cur.item == item:
        return True
      else:
        cur = cur.next
    return False

#-----有变化----

  def append(self, item):
```

```
    "链表尾部添加元素"
    node = Node(item)
    # 先判断链表是否为空，若是空链表，则将_head指向新节点node
    if self.is_empty():
      self.__head = node
    # 若不为空，则找到尾部，将尾节点的next指向新节点
    else:
      cur = self.__head
      while cur.next != None:
        cur = cur.next
      cur.next = node
      node.prev = cur

  def add(self, item):
    "链表头部添加元素"
      # 先判断链表是否为空，若是空链表，则将_head指向新节点node
    if self.is_empty():
      self.__head = node
    else:
      # 先创建一个保存item值的节点
      node = Node(item)
      # 将新节点的链接域next指向头节点，即_head指向的位置
      node.next = self.__head
      # 将链表的头_head指向新节点
      self.__head = node
      node.next.prev = node

#---也可以是这样：2-3注意循序
# node.next = self.__head
# self.__head.prev= node
# self.__head = node

#def add(self, item):
#    """头部插入元素"""
#    node = Node(item)
#    if self.is_empty():
#        # 如果是空链表，将_head指向node
#        self.__head = node
#    else:
#        # 将node的next指向_head的头节点
#        node.next = self.__head
#        # 将_head的头节点的prev指向node
#        self.__head.prev = node
```

```
90  #         # 将_head 指向node
91  #         self.__head = node
92
93    def insert(self, pos, item):
94      "指定位置添加元素"
95      if pos <= 0:
96        self.add(item)
97      elif pos > (self.length()-1):
98        self.append(item)
99      else:
100       node = Node(item)
101       cur = self.__head
102       #cur指向pos位置
103       count = 0
104       while count < pos:
105         count += 1
106         cur = cur.next
107       # 将node的next指向cur
108       node.next = cur
109       # 将node的prev指向cur的prev
110       node.prev = cur.prev
111       # 将cur的prev的指向node
112       cur.prev = node
113       # 将cur的prev的下一个节点指向node
114       cur.prev.next = node
115
116
117   def remove(self, item):
118     "删除节点"
119     cur = self.__head
120
121     while cur != None:
122       # 找到了要删除的元素
123       if cur.item == item:
124         # 先判断此结点是否是首节点
125         if cur == self.__head:
126           # 删除首节点
127           self.__head = cur.next
128           if cur.next:
129             # 判断链表是否只有一个节点，即cur.next != None
130             cur.next.prev = None
131
132         else:
```

```
            # 删除非首节点（一般情况）
            cur.prev.next = cur.next
            if cur.next:
              # 判断链表是否是最后一个节点
              cur.next.prev = cur.prev
          break
        else:
          cur = cur.next
```

```
#----测试----

# 节点实现

class Node(object):
  """双链表的结点"""
  def __init__(self, item):
    # item存放数据元素
    self.item = item
    # 后继节点 next
    self.next = None
    #前驱节点 prev
    self.prev = None

# 双向链表的实现

class DoubleLinkList(object):
  "双链表"
  def __init__(self,node=None):
    "链表头"
    self.__head = None

  def is_empty(self):
    "链表是否为空--一样"
    return self.__head is None
    #将head是否指向None作为return的结果

  def length(self):
    "链表长度--一样"
    # cur初始时指向头节点，用来移动遍历节点
```

```
        cur = self.__head
        count = 0
        # 尾节点指向None，当未到达尾部时
        while cur != None:
            count += 1
            # 将cur后移一个节点
            cur = cur.next
        return count

    def travel(self):
        "遍历整个链表--一样"
        cur = self.__head
        while cur != None:
            print(cur.item, end=" ")
            cur = cur.next
        print("")

    def search(self,  item):
        "查找节点是否存在，并返回True或者False--一样"
        cur = self.__head
        while cur != None:
            if cur.item == item:
                return True
            else:
                cur = cur.next
        return False

#-----有变化----

    def append(self, item):
        "链表尾部添加元素"
        node = Node(item)
        # 先判断链表是否为空，若是空链表，则将_head指向新节点node
        if self.is_empty():
            self.__head = node
        # 若不为空，则找到尾部，将尾节点的next指向新节点
        else:
            cur = self.__head
            while cur.next != None:
                cur = cur.next
            cur.next = node
            node.prev = cur

```

```
  def add(self, item):
    "链表头部添加元素"
      # 先判断链表是否为空，若是空链表，则将_head指向新节点node
    if self.is_empty():
      self.__head = node
    else:
      # 先创建一个保存item值的节点
      node = Node(item)
      # 将新节点的链接域next指向头节点，即_head指向的位置
      node.next = self.__head
      # 将链表的头_head指向新节点
      self.__head = node
      node.next.prev = node

#---也可以是这样：2-3注意循序
# node.next = self.__head
# self.__head.prev= node
# self.__head = node

#def add(self, item):
#    """头部插入元素"""
#    node = Node(item)
#    if self.is_empty():
#        # 如果是空链表，将_head指向node
#        self.__head = node
#    else:
#        # 将node的next指向_head的头节点
#        node.next = self.__head
#        # 将_head的头节点的prev指向node
#        self.__head.prev = node
#        # 将_head 指向node
#        self.__head = node

  def insert(self, pos, item):
    "指定位置添加元素"
    if pos <= 0:
      self.add(item)
    elif pos > (self.length()-1):
      self.append(item)
    else:
      node = Node(item)
      cur = self.__head
      #cur指向pos位置
```

```
        count = 0
        while count < pos:
            count += 1
            cur = cur.next
        # 将node的next指向cur
        node.next = cur
        # 将node的prev指向cur的prev
        node.prev = cur.prev
        # 将cur的prev的指向node
        cur.prev = node
        # 将cur的prev的下一个节点指向node
        cur.prev.next = node


    def remove(self, item):
        "删除节点"
        cur = self.__head

        while cur != None:
            # 找到了要删除的元素
            if cur.item == item:
                # 先判断此结点是否是首节点
                if cur == self.__head:
                    # 删除首节点
                    self.__head = cur.next
                    if cur.next:
                        # 判断链表是否只有一个节点，即cur.next != None
                        cur.next.prev = None

                else:
                    # 删除非首节点（一般情况）
                    cur.prev.next = cur.next
                    if cur.next:
                        # 判断链表是否是最后一个节点
                        cur.next.prev = cur.prev
                break
            else:
                cur = cur.next


#测试

if __name__ == "__main__":
```

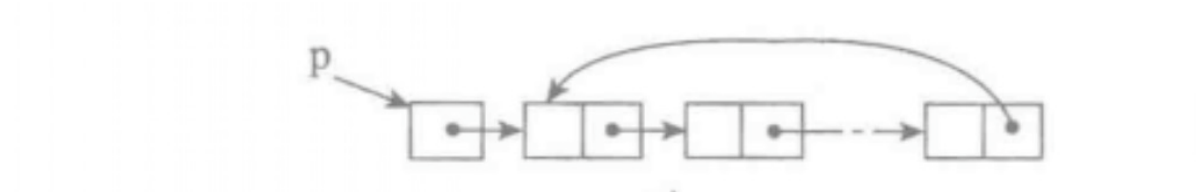
```
    dll = DoubleLinkList()
    print(dll.is_empty())
    print(dll.length())

    dll.append(1)
    print(dll.is_empty())
    print(dll.length())
    dll.add(9)
    dll.append(2)
    dll.append(3)
    dll.append(4)
    dll.insert(-1,100)
    dll.travel()
    dll.insert(100,200)
    dll.travel()
    dll.insert(3,8)
    dll.travel()
    dll.search(100)
    dll.remove(100)
    dll.travel()
    dll.remove(1)
    dll.travel()
    dll.remove(200)
    dll.travel()
```

```
True
0
False
1
100 9 1 2 3 4
100 9 1 2 3 4 200
100 9 1 2 3 4 200
9 1 2 3 4 200
9 2 3 4 200
9 2 3 4
```

单向循环链表

单向循环链表的结构

单链表的一个变形是单向循环链表，链表中最后一个节点的next域不再为None，而是指向链表的头节点。



单向循环链表的操作

> is_empty() 链表是否为空
>
> length() 链表长度
>
> travel() 遍历整个链表
>
> add(item) 链表头部添加元素
>
> append(item) 链表尾部添加元素
>
> insert(pos, item) 指定位置添加元素
>
> remove(item) 删除节点
>
> search(item) 查找节点是否存在

```
#----code含测试-----
# 节点
class SingleNode(object):
  def __init__(self, item):
    self.item = item
    self.next = None


# 单向循环链表
class SinCycLinkedList(object):
  "单向循环链表"
  def __init__(self, node=None):
    self.__head = None
    if node:
```

```
    # 判断node是否是None
      node.next = node

  def is_empty(self):
    "链表是否为空--一样"
    return self.__head == None
    # 将head是否指向None作为return的结果

  def length(self):
    "链表长度"
    # 如果链表为空，返回长度0
    if self.is_empty():
      return 0

    cur = self.__head
    count = 1
    while cur.next != self.__head:
        count += 1
        cur = cur.next
    return count

  def travel(self):
    "遍历整个链表"
    if self.is_empty():
      # 空链表，退出不做任何操作
      return
    cur = self.__head
    while cur.next != self.__head:
      print(cur.item, end=" ")
      cur = cur.next
    # 退出循环，cur指向尾节点，但是未被打印
    print(cur.item)
    print("")

  def add(self, item):
    "链表头部添加元素"
    node = SingleNode(item)
    if self.is_empty():
      self.__head = node
      node.next = node
    else:
      cur = self.__head
      while cur.next != self.__head:
```

```
58          cur = cur.next
59        # 将新节点的链接域next指向头节点，即_head指向的位置
60        node.next = self.__head
61        # 将链表的头_head指向新节点
62        self.__head = node
63        cur.next = node
64
65    def append(self, item):
66      "链表尾部添加元素"
67      node = SingleNode(item)
68      if self.is_empty():
69        self.__head = node
70        node.next = node
71      else:
72        cur = self.__head
73        while cur.next != self.__head:
74          cur = cur.next
75        cur.next = node
76        node.next = self.__head
77
78    def insert(self, pos, item):
79      "指定位置添加元素--一样"
80      if pos <= 0:
81        self.add(item)
82      elif pos > (self.length()-1):
83        self.append(item)
84      else:
85        node = SingleNode(item)
86        # pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置
87        pre = self.__head
88        count = 0
89        while count < (pos-1):
90          count += 1
91          pre = pre.next
92        # 先将新节点node的next指向插入位置的节点
93        node.next = pre.next
94        # 将插入位置的前一个节点的next指向新节点
95        pre.next = node
96        #不是pre = node，是pre节点的next区域变成node
97
98    def search(self,  item):
99      "查找节点是否存在，并返回True或者False"
100     if self.is_empty():
```

```
        return False
    cur = self.__head
    if cur.item == item:
        return True
    while cur.next != self.__head:
        if cur.item == item:
            return True
        else:
            cur = cur.next
    # 退出循环，cur指向尾节点
    return False

def remove(self, item):
    "删除节点"
    if self.is_empty():
        return

    cur = self.__head
    pre = None

    while cur.next != self.__head:
        if cur.item == item:
            if cur == self.__head:
                # 头节点
                rear = self.__head #找尾节点
                while rear.next != self.__head:
                    rear = rear.next
                self.__head = cur.next
                rear.next = self.__head
            else:
                # 中间节点
                pre.next = cur.next
            return # 不是break
            # break是跳出一层循环，continue是结束一趟循环
        else:
            pre = cur
            cur = cur.next
    # 退出循环，cur指向尾节点
    if cur.item == item:
        if cur == self.__head:
        # 链表只有一个节点
            self.__head = None
        else:
```

```
        pre.next = cur.next
        # 等于 pre.next = self.__head


#测试
if __name__ == "__main__":
    ll = SinCycLinkedList()
    print(ll.is_empty())
    print(ll.length())

    ll.append(1)
    print(ll.is_empty())
    print(ll.length())
    ll.add(9)
    ll.append(2)
    ll.append(3)
    ll.append(4)
    ll.insert(-1,100)
    ll.travel()
    ll.insert(100,200)
    ll.travel()
    ll.insert(3,8)
    ll.travel()
    ll.search(1000)

    ll.remove(100)
    ll.travel()
    ll.remove(1)
    ll.travel()
    ll.remove(200)
    ll.travel()

```

```
True
0
False
1
100 9 1 2 3 4

100 9 1 2 3 4 200

100 9 1 8 2 3 4 200

```

```
9 1 8 2 3 4 200

9 8 2 3 4 200

9 8 2 3 4
```

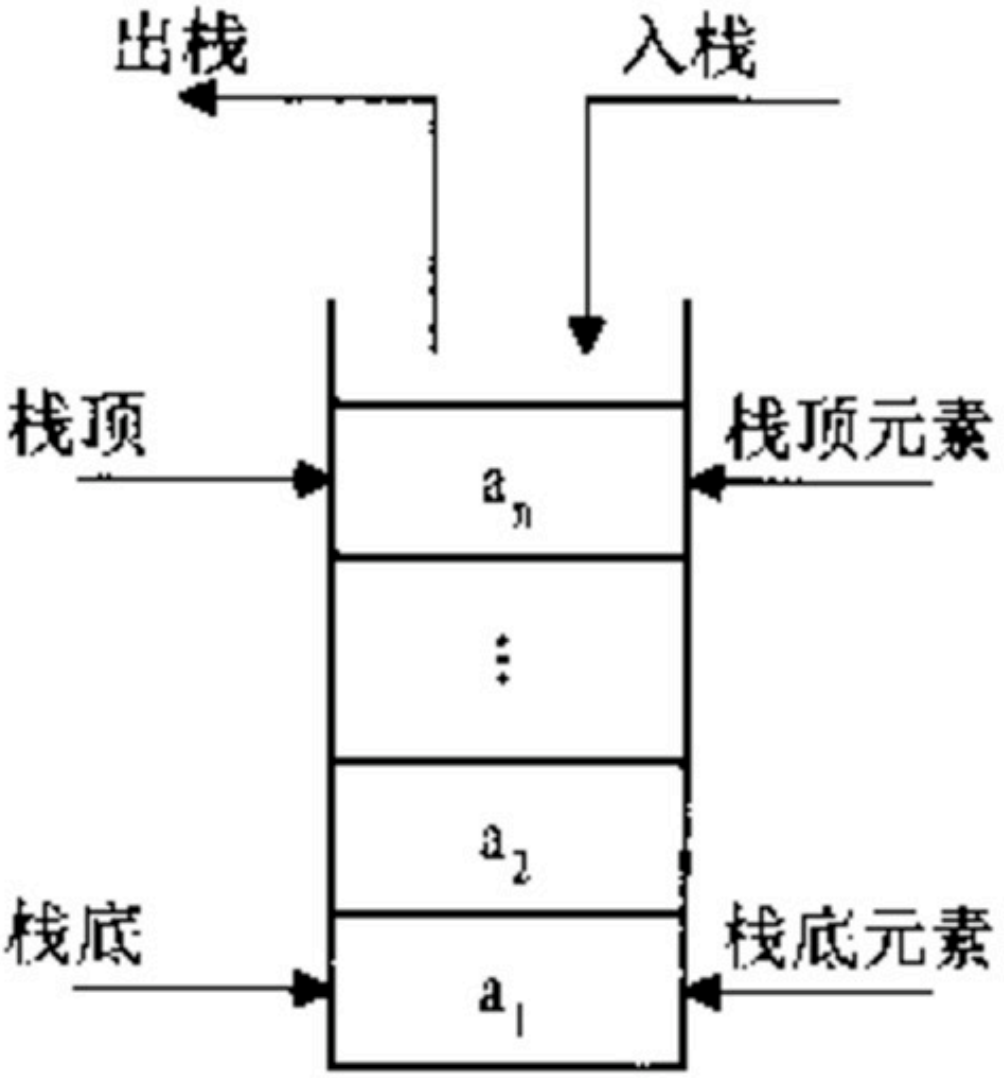
栈线性表的区别：栈描述怎么操作，线性表描述怎么存放

栈和队列（下一节）：这两种数据结构不用考虑他们在物理上是怎样储存的， 只需要关心**支持什么样的操作，操作有什么特点**

栈 Stack

栈（stack），有些地方称为堆栈，是一种容器，可存入数据元素、访问元素、删除元素，它的特点在于只能允许在容器的一端（称为栈顶端指标，英语：top）进行加入数据（英语：push）和输出数据（英语：pop）的运算。没有了位置概念，保证任何时候可以访问、删除的元素都是此前最后存入的那个元素，确定了一种默认的访问顺序。

由于栈数据结构只允许在一端进行操作，因而按照后进先出“最后一个进来的第一个先出去”（LIFO, Last In First Out）的原理运作。



像是一个杯子，只有一个口：先加的跑到杯底，倒出的时候杯口先出来

栈结构和实现

栈的操作

> Stack() 创建一个新的空栈
>
> push(item) 添加一个新的元素item到栈顶
>
> pop() 弹出栈顶元素
>
> peek() 返回栈顶元素

> is_empty() 判断栈是否为空
>
> size() 返回栈的元素个数

```
class Stack(object):
    "栈"
    def __init__(self):
         self.items = []

    def is_empty(self):
        "判断是否为空"
        return self.items == []
        # return not self.item
#逻辑上为假: "", [], {},(),

    def push(self, item):
        "加入元素"
        self.items.append(item)

    def pop(self):
        "弹出元素"
        return self.items.pop()

    def peek(self):
        "返回栈顶元素"
        if self.items:
          return self.items[len(self.items)-1]
        else:
          return None
    def size(self):
        "返回栈的大小"
        return len(self.items)

if __name__ == "__main__":
    stack = Stack()
    stack.push("hello")
    stack.push("world")
    stack.push("itcast")
    print(stack.size())
    print(stack.peek())
```

```
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())

```

```
3
itcast
itcast
world
hello
```

队列

队列（queue）是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

队列是一种先进先出的（First In First Out）的线性表，简称FIFO。允许插入的一端为队尾，允许删除的一端为队头。队列不允许在中间部位进行操作！假设队列是q=（a1，a2，……，an），那么a1就是队头元素，而an是队尾元素。这样我们就可以删除时，总是从a1开始，而插入时，总是在队列最后。这也比较符合我们通常生活中的习惯，排在第一个的优先出列，最后来的当然排在队伍最后。

操作

> Queue() 创建一个空的队列
>
> enqueue(item) 往队列中添加一个item元素
>
> dequeue() 从队列头部删除一个元素
>
> is_empty() 判断一个队列是否为空
>
> size() 返回队列的大小

```
class Queue(object):

```

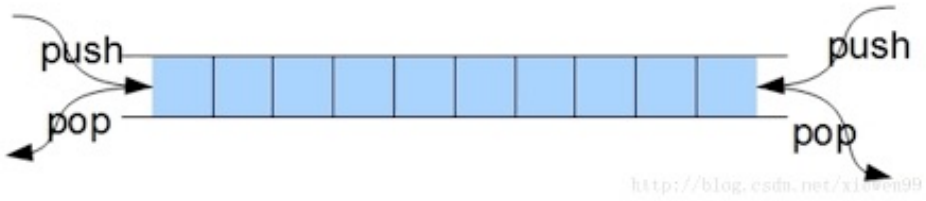
```
    "队列"
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        "进队列"
        self.items.insert(0,item)# 尾部添加
        #self.item.append(item) # 头部添加
    def dequeue(self):
        "出队列"
        return self.items.pop() # 尾部弹出
        # return self.items.pop(0) # 头部弹出
    def size(self):
        "返回大小"
        return len(self.items)

if __name__ == "__main__":
    q = Queue()
    q.enqueue("hello")
    q.enqueue("world")
    q.enqueue("itcast")
    print(q.size())
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())
```

```
3
hello
world
itcast
```

双端队列

双端队列（deque，全名double-ended queue），是一种具有队列和栈的性质的数据结构。

双端队列中的元素可以从两端弹出，其限定插入和删除操作在表的两端进行。双端队列可以在队列任意一端入队和出队。



双端队列：相当于两个栈底部合到一起

操作

> Deque() 创建一个空的双端队列
>
> add_front(item) 从队头加入一个item元素
>
> add_rear(item) 从队尾加入一个item元素
>
> remove_front() 从队头删除一个item元素
>
> remove_rear() 从队尾删除一个item元素
>
> is_empty() 判断双端队列是否为空
>
> size() 返回队列的大小

return 会直接另函数返回，函数就运行结束了，所有该函数体内的代码都不再执行了，所以该函数体内的循环也不可能再继续运行。

如果你需要让循环继续执行，就不能return函数，而应该选用break或者continue。

break：跳出所在的当前整个循环，到外层代码继续执行。

continue：跳出本次循环，从下一个迭代继续运行循环，内层循环执行完毕，外层代码继续运行。

return：直接返回函数，所有该函数体内的代码（包括循环体）都不会再执行。

```
class Deque(object):
    "双端队列"
    def __init__(self):
        self.items = []

    def is_empty(self):
```

```
        "判断队列是否为空"
        return self.items == []

    def add_front(self, item):
        "在队头添加元素"
        self.items.insert(0,item)

    def add_rear(self, item):
        "在队尾添加元素"
        self.items.append(item)

    def remove_front(self):
        "从队头删除元素"
        return self.items.pop(0)

    def remove_rear(self):
        "从队尾删除元素"
        return self.items.pop()

    def size(self):
        "返回队列大小"
        return len(self.items)


if __name__ == "__main__":
    deque = Deque()
    deque.add_front(1)
    deque.add_front(2)
    deque.add_rear(3)
    deque.add_rear(4)
    print(deque.size())
    print("deque:", 2, 1, 3, 4)
    print(deque.remove_front())
    print(deque.remove_front())
    print(deque.remove_rear())
    print(deque.remove_rear())
```

```
4
deque: 2 1 3 4
2
1
4
3
```

```
```