

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

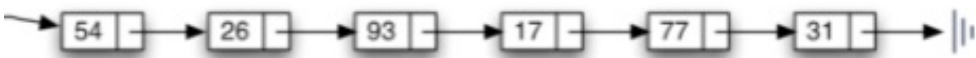
为什么需要链表

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。

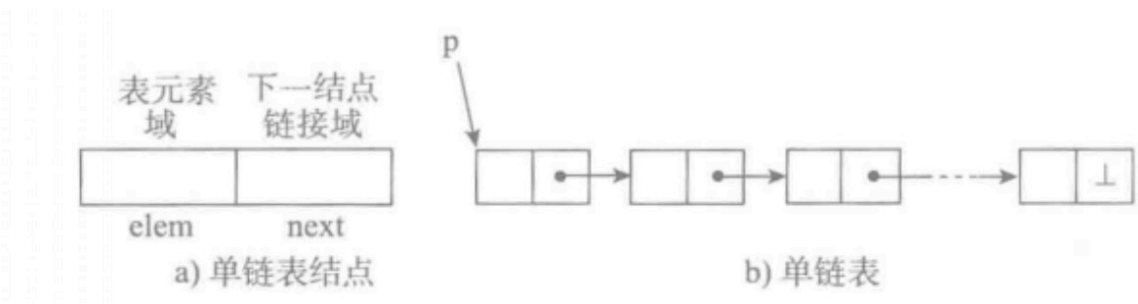
链表的定义

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

```
1  # 节点实现
2
3  class SingleNode(object):
4      """单链表的结点"""
5      def __init__(self, item):
6          # item存放数据元素
7          self.item = item
8          # next是下一个节点的标识,因未知所以设为空None
9          self.next = None
```

```
1  #单链表的实现
2
3  class SingleLinkList(object):
4      "单链表"
5      def __init__(self, node=None):
6          "链表头"
7          self.__head = None
8          #_表示私有
9
10     def is_empty(self):
11         "链表是否为空"
12         return self.__head == None
```

```
13     #将head是否指向None作为return的结果
14
15     def length(self):
16         "链表长度"
17         # cur初始时指向头节点，用来移动遍历节点
18         cur = self.__head
19         count = 0
20         # 尾节点指向None，当未到达尾部时
21         while cur != None:
22             count += 1
23             # 将cur后移一个节点
24             cur = cur.next
25         return count
26
27
28     def travel(self):
29         "遍历整个链表"
30         cur = self.__head
31         while cur != None:
32             print(cur.item, end=" ")
33             cur = cur.next
34         print("") #换行
35
36     def append(self, item):
37         "链表尾部添加元素"
38         node = SingleNode(item)
39         # 先判断链表是否为空，若是空链表，则将_head指向新节点node
40         if self.is_empty():
41             self.__head = node
42         # 若不为空，则找到尾部，将尾节点的next指向新节点
43         else:
44             cur = self.__head
45             while cur.next != None:
46                 cur = cur.next
47             cur.next = node
48
49     def add(self, item):
50         "链表头部添加元素"
51         # 先创建一个保存item值的节点
52         node = SingleNode(item)
53         # 将新节点的链接域next指向头节点，即_head指向的位置
54         node.next = self.__head
55         # 将链表的头_head指向新节点
```

```

56     self.head = node
57
58     def insert(self, pos, item):
59         "指定位置添加元素"
60         if pos <= 0:
61             self.add(item)
62         elif pos > (self.length()-1):
63             self.append(item)
64         else:
65             node = SingleNode(item)
66             # 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
67             pre = self.__head
68             count = 0
69             while count < (pos-1):
70                 count += 1
71                 pre = pre.next
72             # 先将新节点node的next指向插入位置的节点
73             node.next = pre.next
74             # 将插入位置的前一个节点的next指向新节点
75             pre.next = node
76             #不是pre = node, 是pre节点的next区域变成node
77
78     def search(self, item):
79         "查找节点是否存在, 并返回True或者False"
80         node = SingleNode(item)
81         cur = self.__head
82         while cur != None:
83             if cur == node:
84                 return True
85             else:
86                 cur = cur.next
87         return False
88
89     def remove(self, item):
90         "删除节点"
91         cur = self.__head
92         pre = None
93
94         while cur != None:
95             if cur.item == item:
96                 # 如果第一个就是删除的节点
97                 if cur == self.__head:
98                     # 将头指针指向头节点的后一个节点

```

```

99         self.__head = cur.next
100     else:
101         # 将删除位置前一个节点的next指向删除位置的后一个节点
102         pre.next = cur.next
103         break
104     else:
105         pre = cur
106         cur = cur.next
107

```

```

1  #----测试-----
2  # 节点
3  class SingleNode(object):
4      """单链表的结点"""
5      def __init__(self, item):
6          # item存放数据元素
7          self.item = item
8          # next是下一个节点的标识,因未知所以设为空None
9          self.next = None
10
11
12  # 单链表
13  class SingleLinkList(object):
14      "单链表"
15      def __init__(self, node=None):
16          "链表头"
17          self.__head = None
18      #_表示私有
19
20      def is_empty(self):
21          "链表是否为空"
22          return self.__head == None
23          #将head是否指向None作为return的结果
24
25      def length(self):
26          "链表长度"
27          # cur初始时指向头节点, 用来移动遍历节点
28          cur = self.__head
29          count = 0
30          # 尾节点指向None, 当未到达尾部时
31          while cur != None:
32              count += 1
33          # 将cur后移一个节点

```

```
34         cur = cur.next
35     return count
36
37 def travel(self):
38     "遍历整个链表"
39     cur = self.__head
40     while cur != None:
41         print(cur.item, end=" ")
42         cur = cur.next
43     print("")
44
45 def append(self, item):
46     "链表尾部添加元素"
47     node = SingleNode(item)
48     # 先判断链表是否为空，若是空链表，则将_head指向新节点node
49     if self.is_empty():
50         self.__head = node
51     # 若不为空，则找到尾部，将尾节点的next指向新节点
52     else:
53         cur = self.__head
54         while cur.next != None:
55             cur = cur.next
56         cur.next = node
57
58 def add(self, item):
59     "链表头部添加元素"
60     # 先创建一个保存item值的节点
61     node = SingleNode(item)
62     # 将新节点的链接域next指向头节点，即_head指向的位置
63     node.next = self.__head
64     # 将链表的头_head指向新节点
65     self.__head = node
66
67 def insert(self, pos, item):
68     "指定位置添加元素"
69     if pos <= 0:
70         self.add(item)
71     elif pos > (self.length()-1):
72         self.append(item)
73     else:
74         node = SingleNode(item)
75         # 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
76         pre = self.__head
```

```

77         count = 0
78         while count < (pos-1):
79             count += 1
80             pre = pre.next
81         # 先将新节点node的next指向插入位置的节点
82         node.next = pre.next
83         # 将插入位置的前一个节点的next指向新节点
84         pre.next = node
85         #不是pre = node, 是pre节点的next区域变成node
86
87     def search(self, item):
88         "查找节点是否存在, 并返回True或者False"
89         cur = self.__head
90         while cur != None:
91             if cur.item == item:
92                 return True
93             else:
94                 cur = cur.next
95         return False
96
97     def remove(self, item):
98         "删除节点"
99         cur = self.__head
100         pre = None
101
102         while cur != None:
103             if cur.item == item:
104                 # 如果第一个就是删除的节点
105                 if cur == self.__head:
106                     # 将头指针指向头节点的后一个节点
107                     self.__head = cur.next
108                 else:
109                     # 将删除位置前一个节点的next指向删除位置的后一个节点
110                     pre.next = cur.next
111                 break
112             else:
113                 pre = cur
114                 cur = cur.next
115
116
117     #测试
118     if __name__ == "__main__":
119         ll = SingleLinkedList()

```

```

120     print(ll.is_empty())
121     print(ll.length())
122
123     ll.append(1)
124     print(ll.is_empty())
125     print(ll.length())
126     ll.add(9)
127     ll.append(2)
128     ll.append(3)
129     ll.append(4)
130     ll.insert(-1,100)
131     ll.travel()
132     ll.insert(100,200)
133     ll.travel()
134     ll.insert(3,8)
135     ll.travel()
136     ll.search(100)
137     ll.remove(100)
138     ll.travel()
139     ll.remove(1)
140     ll.travel()
141     ll.remove(200)
142     ll.travel()
143

```

```

1  True
2  0
3  False
4  1
5  100 9 1 2 3 4
6  100 9 1 2 3 4 200
7  100 9 1 8 2 3 4 200
8  9 1 8 2 3 4 200
9  9 8 2 3 4 200
10 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:

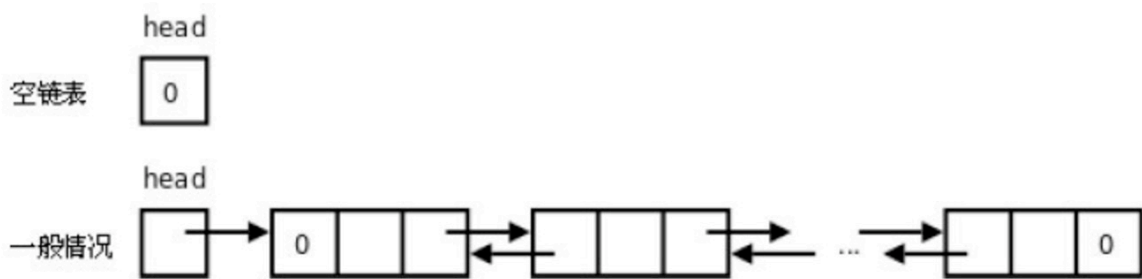
- 优点: 可以使用分散的可用内存
- 缺点: 储存空间需要更大, 除了原有数据还要存地址指针, 存取元素的时候复杂度 $O(n)$
- 在中间插入/删除 insert $O(n)$: 花费在遍历

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

```
1 # 节点实现
2
3 class Node(object):
4     """双链表的结点"""
5     def __init__(self, item):
6         # item存放数据元素
7         self.item = item
8         # 后继节点 next
9         self.next = None
10        # 前驱节点 prev
11        self.prev = None
```

```
1 # 双向链表的实现
2
3 class DoubleLinkedList(object):
```

```

4      "双链表"
5      def __init__(self, node=None):
6          "链表头"
7          self.__head = None
8
9      def is_empty(self):
10         "链表是否为空---一样"
11         return self.__head is None
12         #将head是否指向None作为return的结果
13
14     def length(self):
15         "链表长度---一样"
16         # cur初始时指向头节点，用来移动遍历节点
17         cur = self.__head
18         count = 0
19         # 尾节点指向None，当未到达尾部时
20         while cur != None:
21             count += 1
22             # 将cur后移一个节点
23             cur = cur.next
24         return count
25
26     def travel(self):
27         "遍历整个链表---一样"
28         cur = self.__head
29         while cur != None:
30             print(cur.item, end=" ")
31             cur = cur.next
32         print("")
33
34     def search(self, item):
35         "查找节点是否存在，并返回True或者False---一样"
36         cur = self.__head
37         while cur != None:
38             if cur.item == item:
39                 return True
40             else:
41                 cur = cur.next
42         return False
43
44     #-----有变化-----
45
46     def append(self, item):

```

```

47     "链表尾部添加元素"
48     node = Node(item)
49     # 先判断链表是否为空，若是空链表，则将_head指向新节点node
50     if self.is_empty():
51         self.__head = node
52     # 若不为空，则找到尾部，将尾节点的next指向新节点
53     else:
54         cur = self.__head
55         while cur.next != None:
56             cur = cur.next
57         cur.next = node
58         node.prev = cur
59
60     def add(self, item):
61         "链表头部添加元素"
62         # 先判断链表是否为空，若是空链表，则将_head指向新节点node
63         if self.is_empty():
64             self.__head = node
65         else:
66             # 先创建一个保存item值的节点
67             node = Node(item)
68             # 将新节点的链接域next指向头节点，即_head指向的位置
69             node.next = self.__head
70             # 将链表的头_head指向新节点
71             self.__head = node
72             node.next.prev = node
73
74     #---也可以是这样：2-3注意循序
75     # node.next = self.__head
76     # self.__head.prev= node
77     # self.__head = node
78
79     #def add(self, item):
80     #    """头部插入元素"""
81     #    node = Node(item)
82     #    if self.is_empty():
83     #        # 如果是空链表，将_head指向node
84     #        self.__head = node
85     #    else:
86     #        # 将node的next指向_head的头节点
87     #        node.next = self.__head
88     #        # 将_head的头节点的prev指向node
89     #        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:
```

```
133         # 删除非首节点 (一般情况)
134         cur.prev.next = cur.next
135         if cur.next:
136             # 判断链表是否是最后一个节点
137             cur.next.prev = cur.prev
138         break
139     else:
140         cur = cur.next
141
142
143
144
```

```
1  #----测试----
2
3  # 节点实现
4
5  class Node(object):
6      """双链表的结点"""
7      def __init__(self, item):
8          # item存放数据元素
9          self.item = item
10         # 后继节点 next
11         self.next = None
12         # 前驱节点 prev
13         self.prev = None
14
15     # 双向链表的实现
16
17     class DoubleLinkedList(object):
18         "双链表"
19         def __init__(self, node=None):
20             "链表头"
21             self.__head = None
22
23         def is_empty(self):
24             "链表是否为空--一样"
25             return self.__head is None
26             #将head是否指向None作为return的结果
27
28         def length(self):
29             "链表长度--一样"
30             # cur初始时指向头节点, 用来移动遍历节点
```

```

31     cur = self.__head
32     count = 0
33     # 尾节点指向None, 当未到达尾部时
34     while cur != None:
35         count += 1
36         # 将cur后移一个节点
37         cur = cur.next
38     return count
39
40     def travel(self):
41         "遍历整个链表---一样"
42         cur = self.__head
43         while cur != None:
44             print(cur.item, end=" ")
45             cur = cur.next
46         print("")
47
48     def search(self, item):
49         "查找节点是否存在, 并返回True或者False---一样"
50         cur = self.__head
51         while cur != None:
52             if cur.item == item:
53                 return True
54             else:
55                 cur = cur.next
56         return False
57
58     #-----有变化-----
59
60     def append(self, item):
61         "链表尾部添加元素"
62         node = Node(item)
63         # 先判断链表是否为空, 若是空链表, 则将_head指向新节点node
64         if self.is_empty():
65             self.__head = node
66         # 若不为空, 则找到尾部, 将尾节点的next指向新节点
67         else:
68             cur = self.__head
69             while cur.next != None:
70                 cur = cur.next
71             cur.next = node
72             node.prev = cur
73

```

```

74     def add(self, item):
75         "链表头部添加元素"
76         # 先判断链表是否为空，若是空链表，则将_head指向新节点node
77         if self.is_empty():
78             self.__head = node
79         else:
80             # 先创建一个保存item值的节点
81             node = Node(item)
82             # 将新节点的链接域next指向头节点，即_head指向的位置
83             node.next = self.__head
84             # 将链表的头_head指向新节点
85             self.__head = node
86             node.next.prev = node
87
88     #---也可以是这样：2-3注意循序
89     # node.next = self.__head
90     # self.__head.prev= node
91     # self.__head = node
92
93     #def add(self, item):
94     #     """头部插入元素"""
95     #     node = Node(item)
96     #     if self.is_empty():
97     #         # 如果是空链表，将_head指向node
98     #         self.__head = node
99     #     else:
100     #         # 将node的next指向_head的头节点
101     #         node.next = self.__head
102     #         # 将_head的头节点的prev指向node
103     #         self.__head.prev = node
104     #         # 将_head 指向node
105     #         self.__head = node
106
107     def insert(self, pos, item):
108         "指定位置添加元素"
109         if pos <= 0:
110             self.add(item)
111         elif pos > (self.length()-1):
112             self.append(item)
113         else:
114             node = Node(item)
115             cur = self.__head
116             #cur指向pos位置

```



```

117         count = 0
118         while count < pos:
119             count += 1
120             cur = cur.next
121             # 将node的next指向cur
122             node.next = cur
123             # 将node的prev指向cur的prev
124             node.prev = cur.prev
125             # 将cur的prev的指向node
126             cur.prev = node
127             # 将cur的prev的下一个节点指向node
128             cur.prev.next = node
129
130
131     def remove(self, item):
132         "删除节点"
133         cur = self.__head
134
135         while cur != None:
136             # 找到了要删除的元素
137             if cur.item == item:
138                 # 先判断此结点是否是首节点
139                 if cur == self.__head:
140                     # 删除首节点
141                     self.__head = cur.next
142                     if cur.next:
143                         # 判断链表是否只有一个节点，即cur.next != None
144                         cur.next.prev = None
145
146                 else:
147                     # 删除非首节点（一般情况）
148                     cur.prev.next = cur.next
149                     if cur.next:
150                         # 判断链表是否是最后一个节点
151                         cur.next.prev = cur.prev
152                     break
153                 else:
154                     cur = cur.next
155
156
157     #测试
158
159     if __name__ == "__main__":

```

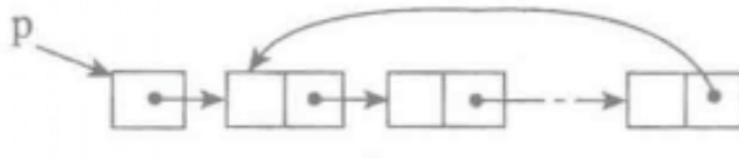
```
160     dll = DoubleLinkedList()
161     print(dll.is_empty())
162     print(dll.length())
163
164     dll.append(1)
165     print(dll.is_empty())
166     print(dll.length())
167     dll.add(9)
168     dll.append(2)
169     dll.append(3)
170     dll.append(4)
171     dll.insert(-1,100)
172     dll.travel()
173     dll.insert(100,200)
174     dll.travel()
175     dll.insert(3,8)
176     dll.travel()
177     dll.search(100)
178     dll.remove(100)
179     dll.travel()
180     dll.remove(1)
181     dll.travel()
182     dll.remove(200)
183     dll.travel()
```

```
1  True
2  0
3  False
4  1
5  100 9 1 2 3 4
6  100 9 1 2 3 4 200
7  100 9 1 2 3 4 200
8  9 1 2 3 4 200
9  9 2 3 4 200
10 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

```
1  #----code含测试-----
2  # 节点
3  class SingleNode(object):
4      def __init__(self, item):
5          self.item = item
6          self.next = None
7
8
9  # 单向循环链表
10 class SinCycLinkedList(object):
11     "单向循环链表"
12     def __init__(self, node=None):
13         self.__head = None
14         if node:
```

```
15     # 判断node是否是None
16     node.next = node
17
18     def is_empty(self):
19         "链表是否为空--一样"
20         return self.__head == None
21         # 将head是否指向None作为return的结果
22
23     def length(self):
24         "链表长度"
25         # 如果链表为空, 返回长度0
26         if self.is_empty():
27             return 0
28
29         cur = self.__head
30         count = 1
31         while cur.next != self.__head:
32             count += 1
33             cur = cur.next
34         return count
35
36     def travel(self):
37         "遍历整个链表"
38         if self.is_empty():
39             # 空链表, 退出不做任何操作
40             return
41         cur = self.__head
42         while cur.next != self.__head:
43             print(cur.item, end=" ")
44             cur = cur.next
45         # 退出循环, cur指向尾节点, 但是未被打印
46         print(cur.item)
47         print("")
48
49     def add(self, item):
50         "链表头部添加元素"
51         node = SingleNode(item)
52         if self.is_empty():
53             self.__head = node
54             node.next = node
55         else:
56             cur = self.__head
57             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():

```

```

101         return False
102     cur = self.__head
103     if cur.item == item:
104         return True
105     while cur.next != self.__head:
106         if cur.item == item:
107             return True
108         else:
109             cur = cur.next
110     # 退出循环, cur指向尾节点
111     return False
112
113 def remove(self, item):
114     "删除节点"
115     if self.is_empty():
116         return
117
118     cur = self.__head
119     pre = None
120
121     while cur.next != self.__head:
122         if cur.item == item:
123             if cur == self.__head:
124                 # 头节点
125                 rear = self.__head #找尾节点
126                 while rear.next != self.__head:
127                     rear = rear.next
128                 self.__head = cur.next
129                 rear.next = self.__head
130             else:
131                 # 中间节点
132                 pre.next = cur.next
133             return # 不是break
134             # break是跳出一层循环, continue是结束一趟循环
135         else:
136             pre = cur
137             cur = cur.next
138     # 退出循环, cur指向尾节点
139     if cur.item == item:
140         if cur == self.__head:
141             # 链表只有一个节点
142             self.__head = None
143         else:

```

```

144         pre.next = cur.next
145         # 等于 pre.next = self.__head
146
147
148     #测试
149     if __name__ == "__main__":
150         ll = SinCycLinkedList()
151         print(ll.is_empty())
152         print(ll.length())
153
154         ll.append(1)
155         print(ll.is_empty())
156         print(ll.length())
157         ll.add(9)
158         ll.append(2)
159         ll.append(3)
160         ll.append(4)
161         ll.insert(-1,100)
162         ll.travel()
163         ll.insert(100,200)
164         ll.travel()
165         ll.insert(3,8)
166         ll.travel()
167         ll.search(1000)
168
169         ll.remove(100)
170         ll.travel()
171         ll.remove(1)
172         ll.travel()
173         ll.remove(200)
174         ll.travel()
175

```

```

1  True
2  0
3  False
4  1
5  100 9 1 2 3 4
6
7  100 9 1 2 3 4 200
8
9  100 9 1 8 2 3 4 200
10

```

11	9 1 8 2 3 4 200
12	
13	9 8 2 3 4 200
14	
15	9 8 2 3 4

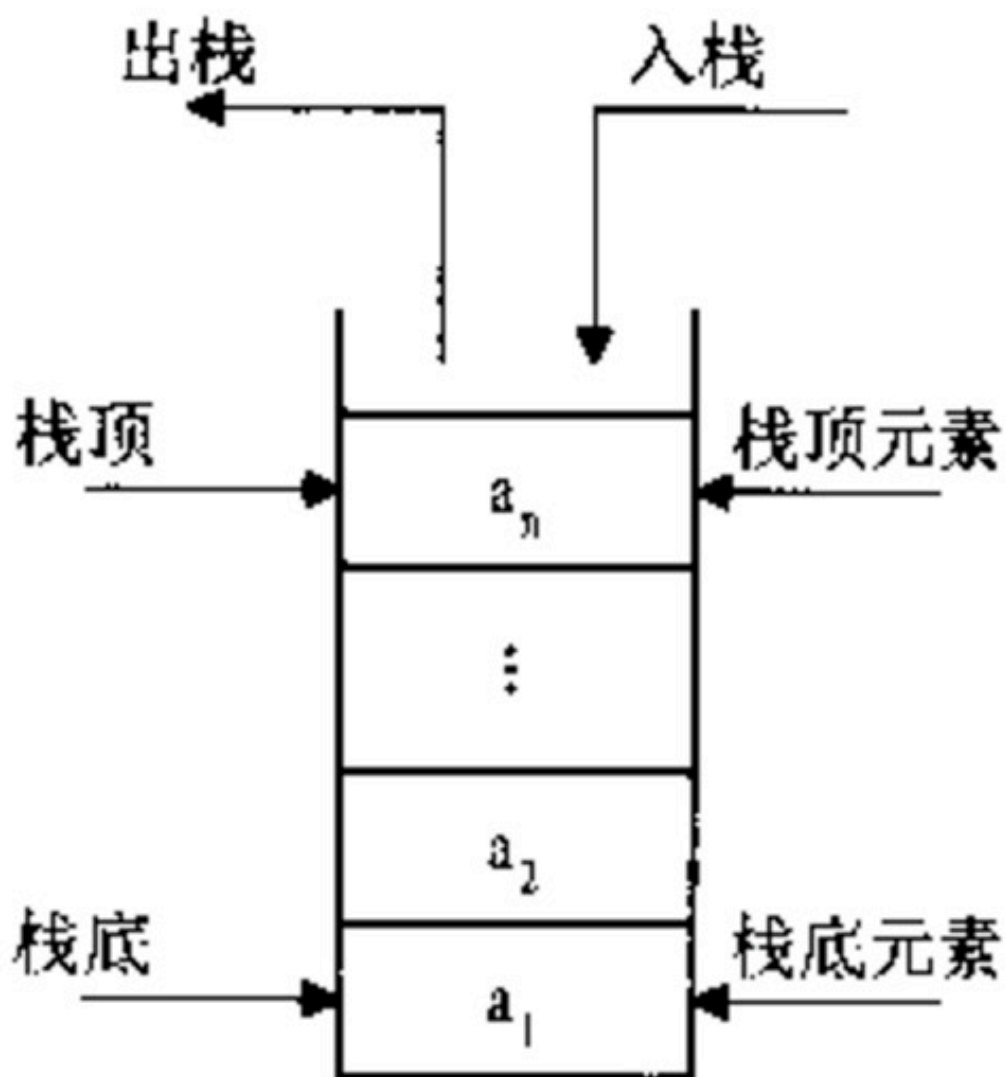
栈线性表的区别：栈描述怎么操作，线性表描述怎么存放

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

```
1 class Stack(object):
2     "栈"
3     def __init__(self):
4         self.items = []
5
6     def is_empty(self):
7         "判断是否为空"
8         return self.items == []
9         # return not self.item
10    #逻辑上为假: "", [], {}, (),
11
12    def push(self, item):
13        "加入元素"
14        self.items.append(item)
15
16    def pop(self):
17        "弹出元素"
18        return self.items.pop()
19
20    def peek(self):
21        "返回栈顶元素"
22        if self.items:
23            return self.items[len(self.items)-1]
24        else:
25            return None
26    def size(self):
27        "返回栈的大小"
28        return len(self.items)
29
30    if __name__ == "__main__":
31        stack = Stack()
32        stack.push("hello")
33        stack.push("world")
34        stack.push("itcast")
35        print(stack.size())
36        print(stack.peek())
```

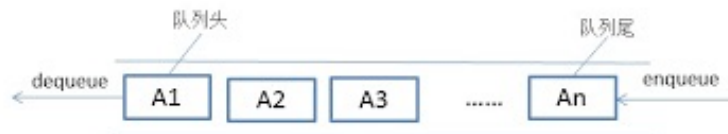
```
37 | print(stack.pop())
38 | print(stack.pop())
39 | print(stack.pop())
40 |
```

```
1 | 3
2 | itcast
3 | itcast
4 | world
5 | 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 $q = (a_1, a_2, \dots, a_n)$ ，那么 a_1 就是队头元素，而 a_n 是队尾元素。这样我们就可以删除时，总是从 a_1 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

```
1 | class Queue(object):
```

```

2     "队列"
3     def __init__(self):
4         self.items = []
5
6     def is_empty(self):
7         return self.items == []
8
9     def enqueue(self, item):
10        "进队列"
11        self.items.insert(0,item)# 尾部添加
12        #self.item.append(item) # 头部添加
13    def dequeue(self):
14        "出队列"
15        return self.items.pop() # 尾部弹出
16        # return self.items.pop(0) # 头部弹出
17    def size(self):
18        "返回大小"
19        return len(self.items)
20
21 if __name__ == "__main__":
22     q = Queue()
23     q.enqueue("hello")
24     q.enqueue("world")
25     q.enqueue("itcast")
26     print(q.size())
27     print(q.dequeue())
28     print(q.dequeue())
29     print(q.dequeue())

```

```

1 3
2 hello
3 world
4 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。

```
1 class Deque(object):
2     "双端队列"
3     def __init__(self):
4         self.items = []
5
6     def is_empty(self):
```

```
7         "判断队列是否为空"
8         return self.items == []
9
10    def add_front(self, item):
11        "在队头添加元素"
12        self.items.insert(0,item)
13
14    def add_rear(self, item):
15        "在队尾添加元素"
16        self.items.append(item)
17
18    def remove_front(self):
19        "从队头删除元素"
20        return self.items.pop(0)
21
22    def remove_rear(self):
23        "从队尾删除元素"
24        return self.items.pop()
25
26    def size(self):
27        "返回队列大小"
28        return len(self.items)
29
30
31    if __name__ == "__main__":
32        deque = Deque()
33        deque.add_front(1)
34        deque.add_front(2)
35        deque.add_rear(3)
36        deque.add_rear(4)
37        print(deque.size())
38        print("deque:", 2, 1, 3, 4)
39        print(deque.remove_front())
40        print(deque.remove_front())
41        print(deque.remove_rear())
42        print(deque.remove_rear())
```

1	4
2	deque: 2 1 3 4
3	2
4	1
5	4
6	3

1	
---	--