数据结构和算法
（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

线性表

线性表

- 线性表是一个元素构成的序列
- 该序列有唯一的头元素和尾元素，除了头元素外，每个元素都有唯一的前驱元素，除了尾元素外，每个元素都有唯一的后继元素
- 线性表中的元素属于相同的数据类型，即每个元素所占的空间必须相同。
- 分为顺序表和链表两种

顺序表

河北草原天路

顺序表

- 即Python的列表，以及其它语言中的数组
- 元素在内存中连续存放
- 每个元素都有唯一序号(下标），且根据序号访问（包括读取和修改）元素的时间复杂度是O(1)的 --- 随机访问
- 下标为i的元素前驱下标为i-1，后继下标为i+1

顺序表支持的操作

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表，元素值随机	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	length()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	find(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	remove(i)	删除下标为i的元素	O(n)

顺序表的append的O(1)复杂度的实现

- 总是分配多于实际元素个数的空间(容量大于元素个数)
- 元素个数小于容量时，append操作复杂度O(1)
- 元素个数等于容量时，append导致重新分配空间，且要拷贝原有元素到新空间，复杂度O(n)

顺序表的append的O(1)复杂度的实现

- 重新分配空间时，新容量为旧容量的k倍(k>1且固定)，可确保append操作的平均复杂度是O(1)。Python的list取k=1.2左右

假定第一次分配空间时，容量为1。由于元素个数每达到k的幂加1(向上取整)时就需要重新分配存储空间并进行元素的复制，执行k^m次append(x)操作，元素个数达到k^m个时，总共复制过的元素个数是：

$$1 + k + k^2 + \cdots + k^{m-1} = \frac{k^m - 1}{k - 1}$$

在元素总数$n=k^m$的情况下，平均每次append(x)操作，需要复制的元素个数是：

$$\frac{(k^m - 1)}{k^m(k - 1)} = \frac{n - 1}{n(k - 1)} < \frac{1}{k - 1}$$

因k是常量，所以append(x)操作的平均复杂度是O(1)的。

链表概述

宁夏中卫沙坡头

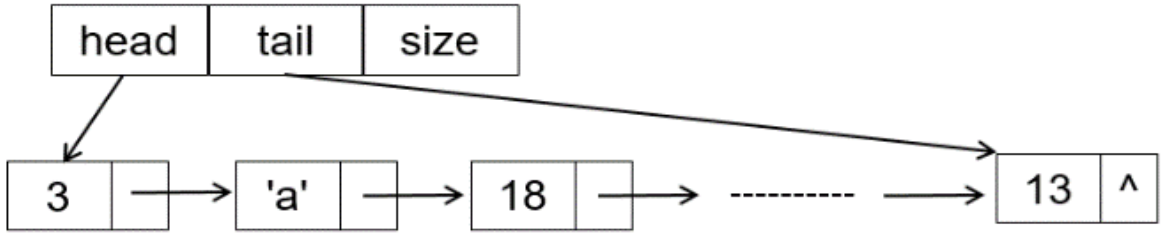
链表

- 元素在内存中并非连续存放，元素之间通过指针链接起来
- 每个结点除了元素，还有next指针，指向后继
- 不支持随机访问。访问第i个元素，复杂度为O(n)
- 已经找到插入或删除位置的情况下，插入和删除元素的复杂度O(1),且不需要复制或移动结点
- 有多种形式：
  - 单链表
  - 循环单链表
  - 双向链表
  - 循环双向链表

单链表

张掖冰沟丹霞

单链表




```
class LinkList:
        class Node: #表结点
                def __init__(self, data, next=None):
                        self.data, self.next = data, next
        def __init__(self):
                self.head = self.tail = None
                self.size = 0
```

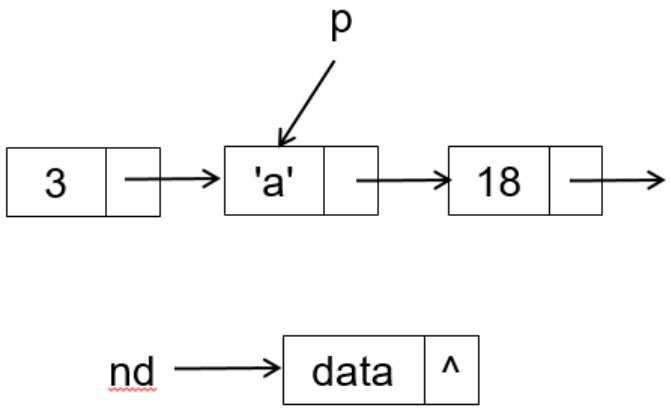
单链表

```
def printList(self): #打印全部结点
        ptr = self.head
        while ptr is not None:
                print(ptr.data, end=",")
                ptr = ptr.next
```

单链表插入元素

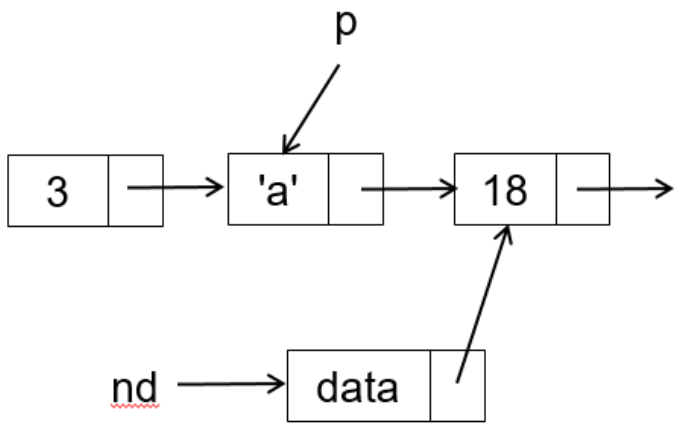
```
def insert(self,p,data): #在结点p后面插入元素
        nd = LinkList.Node(data,None)
        if self.tail is p:   # 新增的结点是新表尾
                self.tail = nd
        nd.next = p.next
        p.next = nd
        self.size += 1
```

单链表插入元素



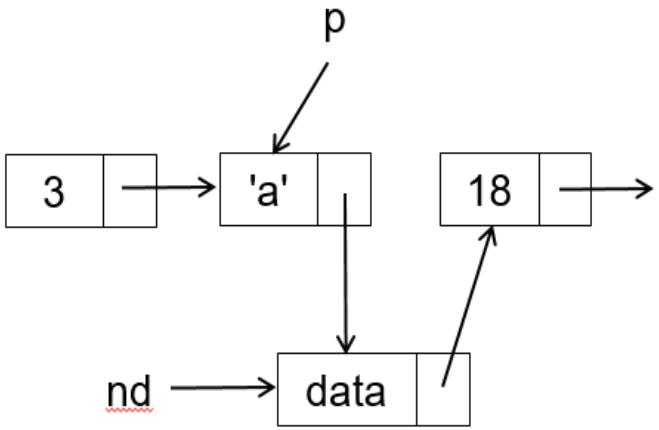
(1)执行nd = Node(data,None) 新建结点nd

单链表插入元素



(2)执行nd.next = p.next

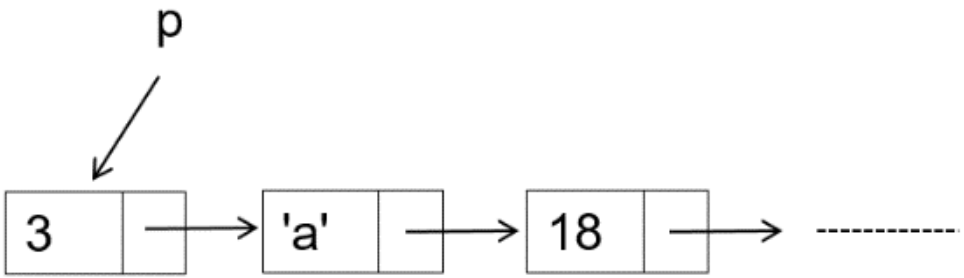
单链表插入元素



(3)执行p.next = nd，完成插入

单链表删除元素

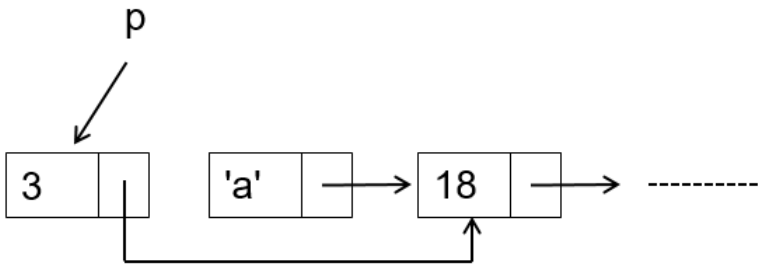
- 删除p后面的元素



(1)初始状态，将要删除'a'结点

单链表删除元素

- 删除p后面的元素



(2)执行p.next = p.next.next，完成删除

单链表删除元素

```
def delete(self,p):   #删除p后面的结点
        if self.tail is p.next:
                self.tail = p
        p.next = p.next.next
        self.size -= 1

#结点空间会被Python自动回收
```

判断变量是否为None，应写 p is None，p is not None
最好不要写 p == None，p != None

单链表

```
def popFront(self): #删除前端元素
        if self.head is None:
                raise \
          Exception("Popping front for Empty link list.")
        else:
                self.head = self.head.next
                self.size -= 1
                if self.size == 0:
                        self.head = self.tail = None
def pushBack(self,data): #在尾部添加元素
        if self.size == 0:
                self.pushFront(data)
        else:
                self.insert(self.tail,data)
```

单链表

```
def pushFront(self,data): #在链表前端插入一个元素data
        nd = LinkList.Node(data, self.head)
        self.head = nd
        self.size += 1
        if self.tail is None:
                self.tail = nd
```

单链表

```
def clear(self):
        self.head = self.tail = None
        self.size = 0
def __iter__(self):
        self.ptr = self.head
        return self
def __next__(self):
        if self.ptr is None:
                raise StopIteration()   # 引发异常
        else:
                data = self.ptr.data
                self.ptr = self.ptr.next
                return data
```

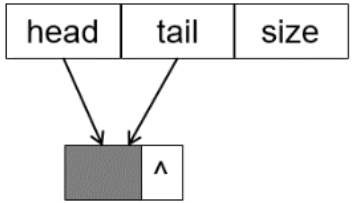
单链表

```
linkLst = LinkList()
linkLst.pushFront(0)
linkLst.pushFront(1)
for i in range(2,5):
        linkLst.pushBack(i)
for x in linkLst:    #>>1,0,2,3,4,
        print(x,end=",")
```

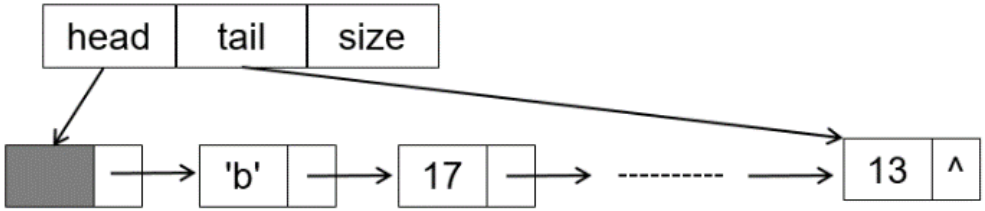
上述实现方式没有实现“隐藏”，不是很好的实现方式

带头结点的单链表

- 为避免链表为空是做特殊处理，可以为链表增加一个空闲头结点



带头结点的空单链表



带头结点的非空单链表

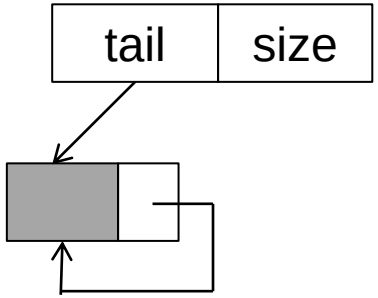
带头结点的单链表

- 为避免链表为空是做特殊处理，可以为链表增加一个空闲头结点

构造函数：

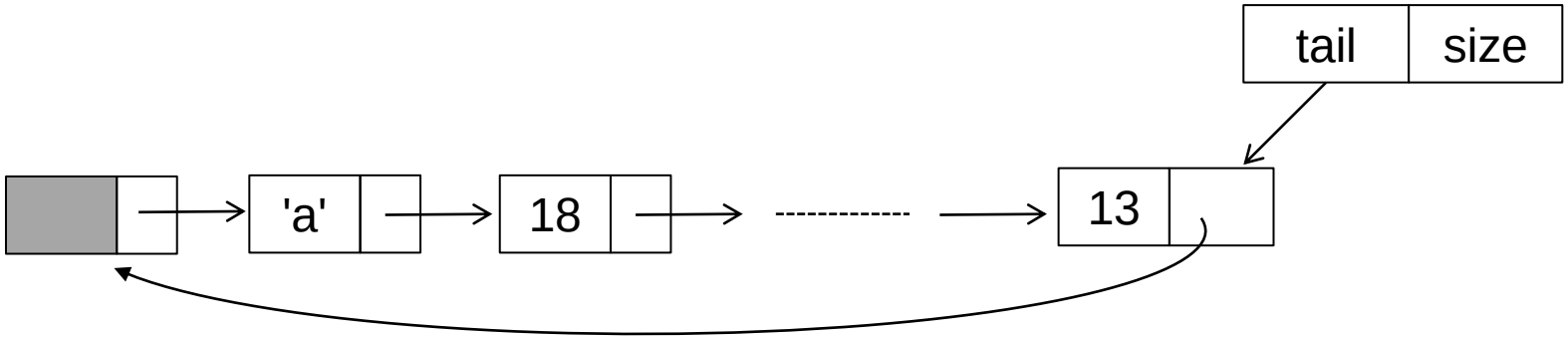
```
class LinkList:
        def __init__(self):
                self.head = self.tail =  LinkList.Node(None,None)
                self.size = 0
```

循环单链表



空表

循环单链表在表首或表尾添加元素，以及删除表首元素，复杂度都是O(1)的。



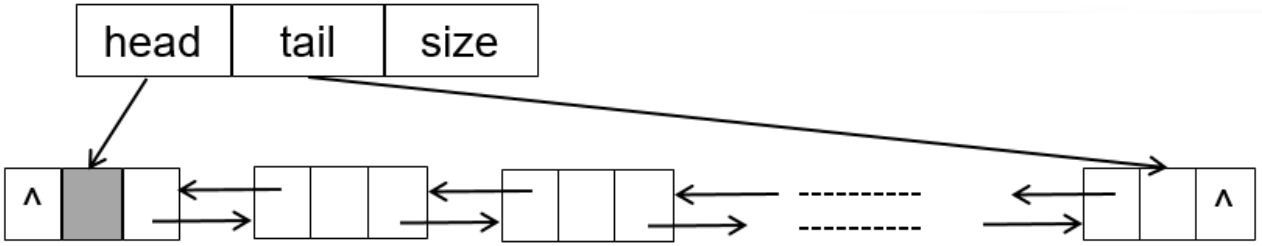
tail.next即头结点

双向链表

张掖平山湖大峡谷

双向链表(双链表)

- 每个结点有next指针指向后继，有prev指针指向前驱

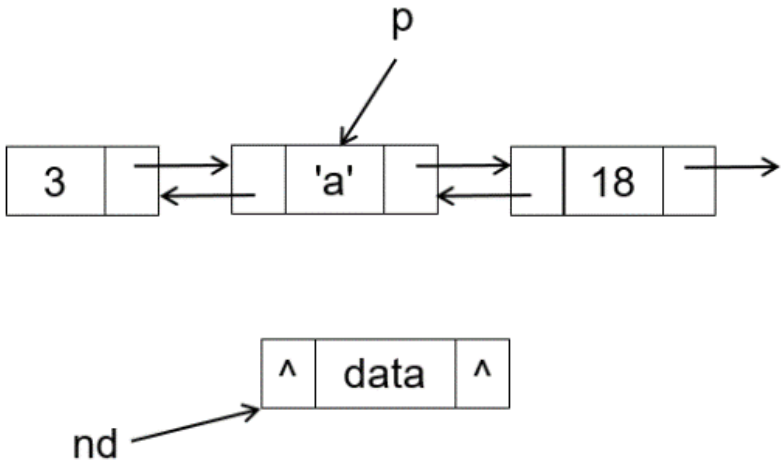


带头结点的双向链表

```
class DoubleLinkList:
    class _Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next
```

双向链表插入结点

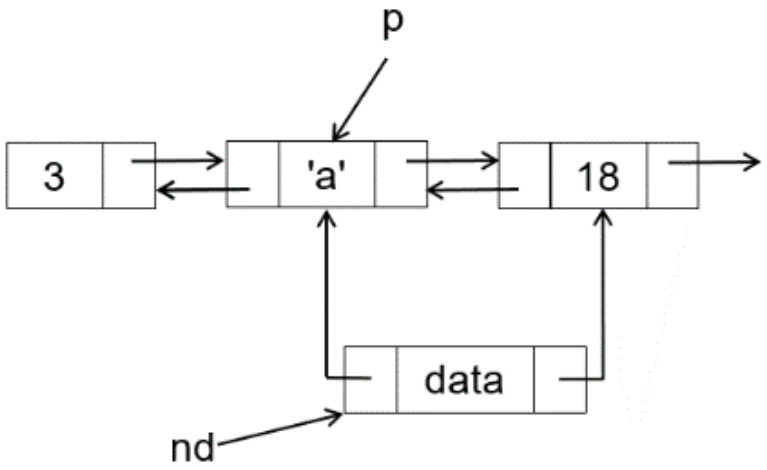
- 在结点p后面插入新结点nd



(1)执行nd = Node(data,None,None) 新建结点nd

双向链表插入结点

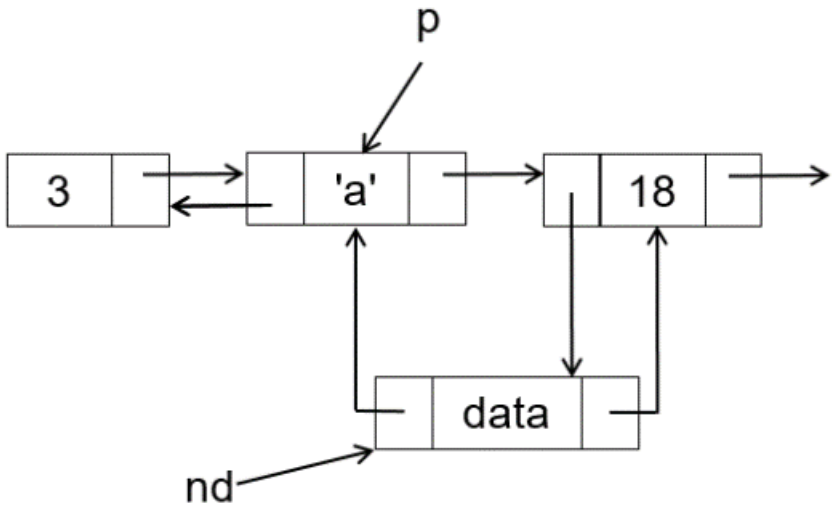
- 在结点p后面插入新结点nd



(2) 执行 nd.prev, nd.next = p, p.next

双向链表插入结点

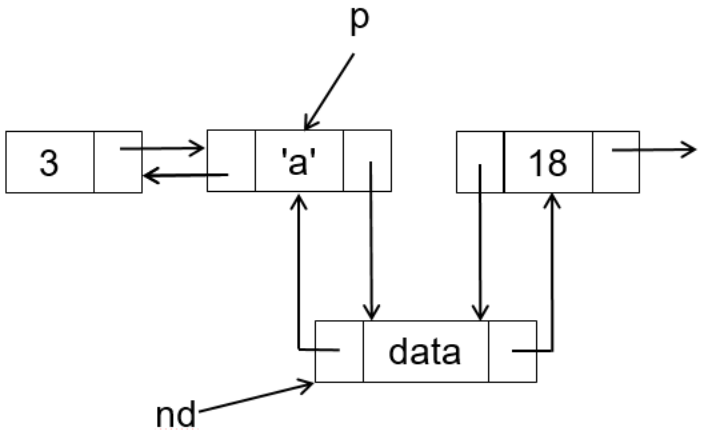
- 在结点p后面插入新结点nd



(3)执行p.next.prev = nd

双向链表插入结点

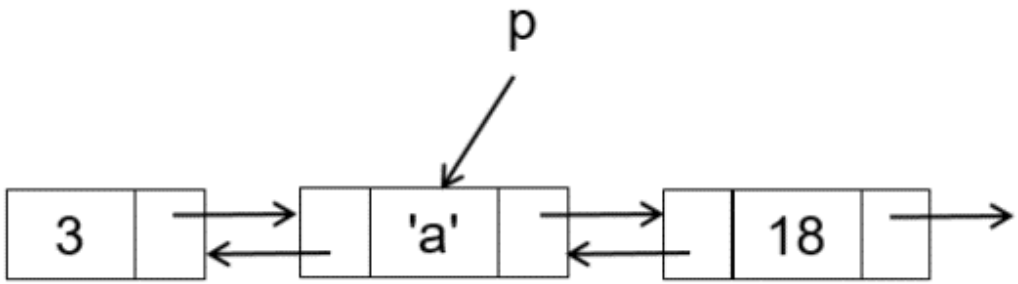
- 在结点p后面插入新结点nd



(4)执行p.next = nd，插入完成

双向链表删除结点

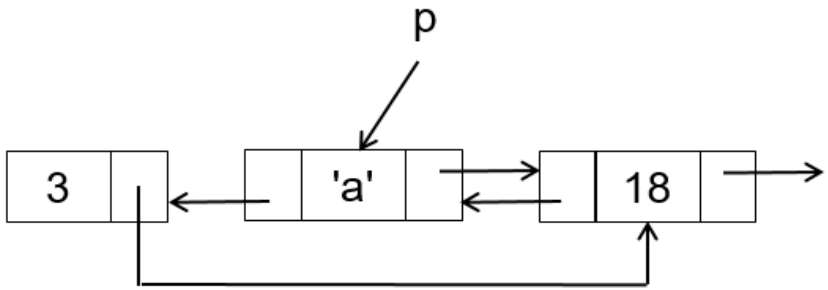
- 删除结点p



(1)初始状态，将要删除'a'结点

双向链表删除结点

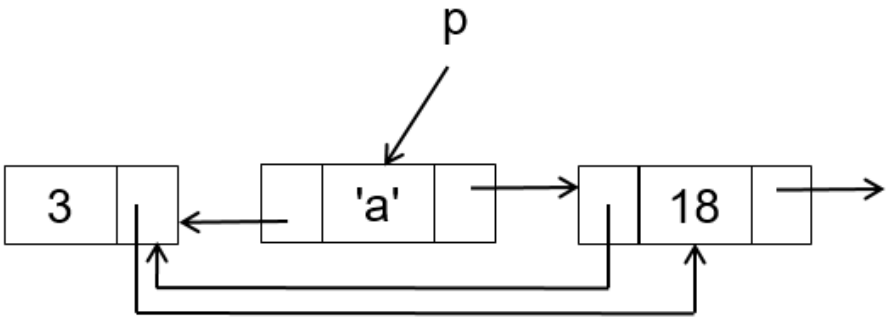
- 删除结点p



(2)执行p.prev.next = p.next

双向链表删除结点

- 删除结点p



(3)执行 p.nex.prev = p.prev, 完成删除

双向链表实现

```
class DoubleLinkList:
    class _Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next
    class _Iterator:
        def __init__(self,p):
            self.ptr = p
        def getData(self):
            return self.ptr.data
        def setData(self,data):
            self.ptr.data = data
        def __next__(self):
            self.ptr = self.ptr.next
            if self.ptr is None:
                return None
            else:
                return DoubleLinkList._Iterator(self.ptr)
```

双向链表实现

```
        def prev(self):
            self.ptr = self.ptr.prev
            return DoubleLinkList._Iterator(self.ptr)

    def __init__(self):
        self._head = self._tail = \
            DoubleLinkList._Node(None,None,None)
        self._size = 0
    def _insert(self,p,data):
        nd = DoubleLinkList._Node(data,p,p.next)
        if self._tail is p:   # 新增的结点是新表尾
            self._tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self._size += 1
```

双向链表实现

```
    def _delete(self,p):   #删除结点p
        if self._size == 0 or p is self._head:
            raise Exception("Illegal deleting.")
        else:
            p.prev.next = p.next
            if p.next: #如果p有后继
                p.next.prev = p.prev
            if self._tail is p:
                self._tail = p.prev
            self._size -= 1
    def clear(self):
        self._tail = self._head
        self._head.next = self._head.prev = None
        self.size = 0
    def begin(self):
        return DoubleLinkList._Iterator(self._head.next)
    def end(self):
        return None
```

双向链表实现

```
def insert(self,i,data): #在迭代器i指向的结点后面插入元素
    self._insert(i.ptr,data)
def delete(self, i):  # 删除迭代器i指向的结点
    self._delete(i.ptr)
def pushFront(self,data): #在链表前端插入一个元素
    self._insert(self._head,data)
def popFront(self):
    self._delete(self._head.next)
def pushBack(self,data):
    self._insert(self._tail,data)
def popBack(self):
    self._delete(self._tail)
def __iter__(self):
    self.ptr = self._head.next
    return self
```

双向链表实现

```
def __next__(self):
    if self.ptr is None:
        raise StopIteration()   # 引发异常
    else:
        data = self.ptr.data
        self.ptr = self.ptr.next
        return data
def find(self,val): #查找元素val，找到返回迭代器，找不到返回None
    ptr = self._head.next
    while ptr is not None:
        if ptr.data == val:
            return DoubleLinkList._Iterator(ptr)
        ptr = ptr.next
    return self.end()
```

双向链表实现

```
    def printList(self):
        ptr = self._head.next
        while ptr is not None:
            print(ptr.data,end=",")
            ptr = ptr.next

linkLst = DoubleLinkList()
for i in range(5):
    linkLst.pushBack(i)
i = linkLst.begin()
while i != linkLst.end(): #>>0,1,2,3,4,
    print(i.getData(),end = ",")
    i = next(i)
print()
i = linkLst.find(3)
i.setData(300)
linkLst.printList()   #>>0,1,2,300,4,
```

双向链表实现

```
print()
linkLst.insert(i,6000) #在i后面插入6000
linkLst.printList() #>>0,1,2,300,6000,4,
print()
linkLst.delete(i)
linkLst.printList() #>>0,1,2,6000,4,
```

链表和顺序表的选择

祁连牛心山

链表和顺序表的选择

- 顺序表

  中间插入太慢

- 链表

  访问第i个元素太慢
  顺序访问也慢(现代计算机有cache，访问连续内存域比跳着访问内存区域快很多)
  还多费空间

  结论：尽量选用顺序表。比如栈和队列，都没必要用链表实现
  基本只有在找到一个位置后反复要在该位置周围进行增删，才适合用链表
  实际工作中几乎用不到链表

链表结合顺序表

- collections.deque：

结合链表和顺序表的特点。

是一张双向链表，每个结点是一个64个元素的顺序表。

```
class Node:
    def __init__(self ,prev=None,next = None):
        self.data = [0 for i in range(64)]
        self.data[0],self.data[-1] = prev,next
```