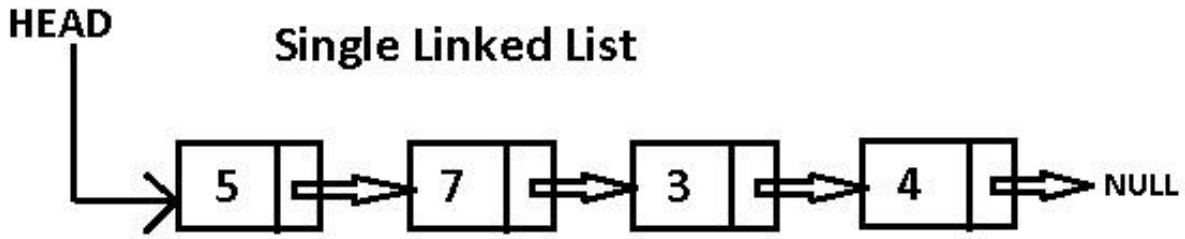
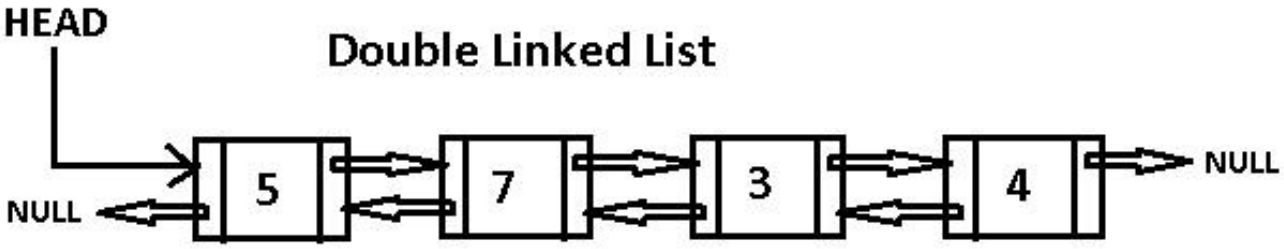
作业

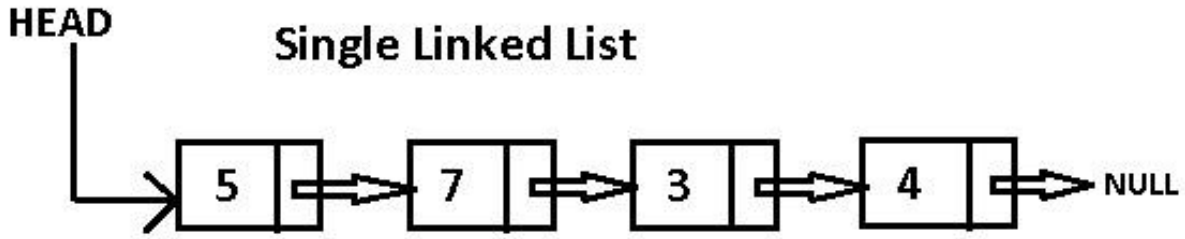
用面向对象实现LinkedList链表
单向链表实现append、iternodes方法
双向链表实现append、pop、insert、remove、iternodes方法

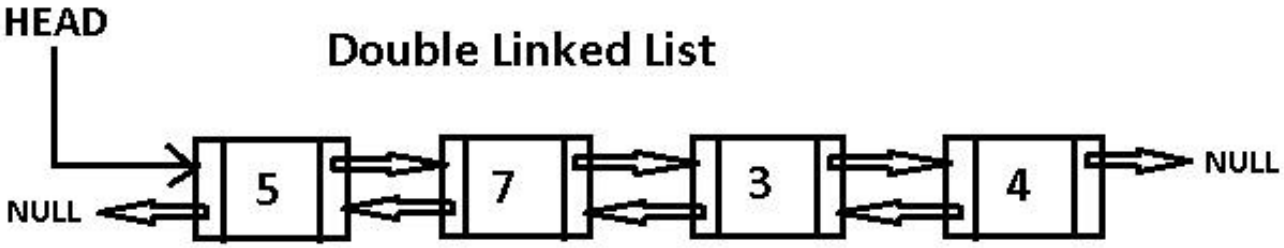




参考

实现LinkedList链表
链表有双向单向链表





对于链表来说，每一个结点是一个独立的对象，结点自己知道内容是什么，下一跳是什么。
而链表则是一个容器，它内部装着一个个结点对象。
所以，建议设计2个类，一个是结点Node类，一个是链表LinkedList类。

单向链表1

```
class SingleNode: # 节点保存内容和下一跳
    def __init__(self, item, next=None):
        self.item = item
        self.next = next

    def __repr__(self):
        return repr(self.item)

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None # 思考tail属性的作用

    def append(self, item):
        node = SingleNode(item)
        if self.head is None: #
            self.head = node # 设置开头结点，以后不变
        else:
            self.tail.next = node # 当前最后一个结点关联下一跳
        self.tail = node # 更新结尾结点
        return self

    def iternodes(self):
        current = self.head
        while current:
            yield current
            current = current.next

ll = LinkedList()
ll.append('abc')
ll.append(1).append(2)
ll.append('def')

print(ll.head, ll.tail)

for item in ll.iternodes():
    print(item)
```

单向链表2

借助列表实现

```
class SingleNode:
    def __init__(self, item, next=None):
        self.item = item
        self.next = next

    def __repr__(self):
        return repr(self.item)

class SingleNode: # 节点保存内容和下一跳
    def __init__(self, item, next=None):
        self.item = item
        self.next = next

    def __repr__(self):
        return repr(self.item)

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None # 思考tail属性的作用
        self.items = [] # 为什么在单向链表中使用list?

    def append(self, item):
        node = SingleNode(item)
        if self.head is None: #
            self.head = node # 设置开头结点，以后不变
        else:
            self.tail.next = node # 当前最后一个结点关联下一跳
        self.tail = node # 更新结尾结点

        self.items.append(node)
        return self

    def iternodes(self):
        current = self.head
```

```
        while current:
            yield current
            current = current.next

    def getitem(self, index):
        return self.items[index]


ll = LinkedList()
ll.append('abc')
ll.append(1).append(2)
ll.append('def')

print(ll.head, ll.tail)

for item in ll.iternodes():
    print(item)

for i in range(len(ll.items)):
    print(ll.getitem(i))
```

为什么在单向链表中使用list？
因为只有结点自己知道下一跳是谁，想直接访问某一个结点只能遍历。
借助列表就可以方便的随机访问某一个结点了。

双向链表

实现单向链表没有实现的pop、remove、insert方法

```
class SingleNode: # 节点保存内容和下一跳
    def __init__(self, item, prev=None, next=None):
        self.item = item
        self.next = next
        self.prev = prev # 增加上一跳

    def __repr__(self):
        #return repr(self.item)
        return "({} <== {} ==> {})".format(
            self.prev.item if self.prev else None,
            self.item,
```

```
            self.next.item if self.next else None)

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None # 思考tail属性的作用
        self.size = 0 # 以后实现

    def append(self, item):
        node = SingleNode(item)
        if self.head is None: #
            self.head = node # 设置开头结点，以后不变
        else:
            self.tail.next = node # 当前最后一个结点关联下一跳
            node.prev = self.tail # 前后关联
        self.tail = node # 更新结尾结点
        return self

    def insert(self, index, item):
        if index < 0: # 不接受负数
            raise IndexError('Not negative index {}'.format(index))

        current = None
        for i, node in enumerate(self.iternodes()):
            if i == index: # 找到了
                current = node
                break
        else: # 没有break，尾部追加
            self.append(item)
            return

        # break，找到了
        node = SingleNode(item)
        prev = current.prev
        next = current

        if prev is None: # 首部
            self.head = node
        else: # 不是首元素
            prev.next = node
```

```
            node.prev = prev
        node.next = next
        next.prev = node

    def pop(self):
        if self.tail is None: # 空
            raise Exception('Empty')

        node = self.tail
        item = node.item
        prev = node.prev
        if prev is None: # only one node
            self.head = None
            self.tail = None
        else:
            prev.next = None
            self.tail = prev
        return item

    def remove(self, index):
        if self.tail is None: # 空
            raise Exception('Empty')

        if index < 0: # 不接受负数
            raise IndexError('Not negative index {}'.format(index))

        current = None
        for i, node in enumerate(self.iternodes()):
            if i == index:
                current = node
                break
        else: # Not Found
            raise IndexError('Wrong index {}.'.format(index))

        prev = current.prev
        next = current.next

        # 4种情况
        if prev is None and next is None: # only one node
            self.head = None
```

```
            self.tail = None
        elif prev is None: # 头部
            self.head = next
            next.prev = None
        elif next is None: # 尾部
            self.tail = prev
            prev.next = None
        else: # 在中间
            prev.next = next
            next.prev = prev

        del current

    def iternodes(self, reverse=False):
        current = self.tail if reverse else self.head
        while current:
            yield current
            current = current.prev if reverse else current.next

ll = LinkedList()
ll.append('abc')
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)
ll.append(5)
ll.append('def')
print(ll.head, ll.tail)

for x in ll.iternodes(True):
    print(x)

print('===============')

ll.remove(6)
ll.remove(5)
ll.remove(0)
ll.remove(1)

for x in ll.iternodes():
```

```
    print(x)

print('~~~~~~~~~~~~~')

ll.insert(3, 5)
ll.insert(20, 'def')
ll.insert(1,2)
ll.insert(0,'abc')
for x in ll.iternodes():
    print(x)
```