CSC1001 Tutorial 12 - Linked List

29 November 2023 - Created by Florensia Widjaja (122040013)

```
## Code for the Node Class
class Node:
    def __init__(self, el):  # define the properties of the node
objects
        self.element = el
        self.pointer = None

class singleLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
        self.tail = None

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def insert_head(self, el):
        new = Node(el)
        self.size += 1
        if self.size == 1:  # it is the only node in this linked list
            self.tail = new
        else:
            new.pointer = self.head  # self.head is not None since
there exist other nodes
        self.head = new
        return new

    def insert_tail(self, el):
        new = Node(el)
        self.size += 1
        if self.size == 1:  # this is the first node of the linked
list --> self.tail and self.head is None
            self.head = new
        else:
            self.tail.pointer = new  # self.tail is not None since
there exist other nodes
        self.tail = new
        return new

    def delete_head(self):
        if self.size == 0:
            print("the Linked List is empty")
```

```
            return None
        else:
            delete_el = self.head.element
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.head = self.head.pointer

            return delete_el

    def iterate(self):
        current_node = self.head
        # you cannot immediately use the linked list head! otherwise,
after finishing iteration,
        # you will end up with an empty linked list
        print("The elements in the current linked list")
        while current_node is not None:
            print(current_node.element, end = ' ')
            current_node =current_node.pointer
        print()

    # ANOTHER ANSWER VERSION FOR QUESTION 1
    def concatenate(self, Mhead):  # M is the other linked list
        # remember that the question told us we only have access to L
head so imagine L tail doesn't exist
        nd = self.head
        while nd.next is not None:
            nd = nd.next
        nd.next = Mhead
        return self.head  # return the head reference of this newly
concatenated linked list
```

Remember! The time complexities:

- delete_head = O(1) # CONSTANT
- insert_head = O(1)
- insert_tail = O(1)

However, **delete_tail = O(n)** # LINEAR

```
# testing the singly linked list code
ll = singleLinkedList()
ll.insert_head(1)  # 1
ll.insert_tail(2)  # 1 2
ll.insert_head(4)  # 4 1 2
```

```
ll.insert_tail(5)  # 4 1 2 5

ll.iterate()

The elements in the current linked list
4 1 2 5
```

Circular Linked List

Difference:

- we don't necessarily have to make two different head and tail variable because... **head_node = tail_node.pointer**
- There is no difference between inserting in the head and the tail
- we can only delete from the head

```
class circularLinkedList:
    def __init__(self):  # same with Single Linked List (SLList)
        self.tail = None
        self.size = 0

    def __len__(self):  # same with Single Linked List (SLList)
        return self.size

    def is_empty(self):  # same with Single Linked List (SLList)
        return self.size == 0

    def insert(self, el):  # insert from the head and from the tail is 
equivalent!!!
        new = Node(el)
        self.size += 1
        if self.size == 1:
            new.pointer = new  # because new is simultaenously the 
head and the tail
        else:  # make the new node the head node
            new.pointer = self.tail.pointer  # new.pointer = head_node
            self.tail.pointer = new
        self.tail = new
        return new

    def delete_head(self):
        if self.size == 0:
            print("The linked list is empty")
        else:
            want_to_delete_node = self.tail.pointer
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.pointer = want_to_delete_node.pointer
```

```python
            return want_to_delete_node


    def iterate(self):
        current_head = self.tail.pointer
        node = self.tail.pointer
        print("the current element in the linked list: ")
        print(node.element)
        node = node.pointer
        while node is not current_head:
            print(node.element)
            node = node.pointer

cl = circularLinkedList()
cl.insert(1)
cl.insert(2)
dellNode = cl.delete_head()
print("Succesfully delete", dellNode.element)
cl.insert(3)
cl.insert(5)
cl.iterate()
```

```
Succesfully delete 1
the current element in the linked list:
2
3
5
```

Time complexity:

- insert: O(1)
- delete_head = O(1)
- delete_tail = O(n) --> NOT EFFICIENT

Doubly Linked List

```python
class dlNode():
    def __init__(self, element):
        self.element = element
        self.prev = None
        self.next = None

class DoubleLinkedList:
    def __init__(self):
        self.head = dlNode(None)  # header
        self.tail = dlNode(None)  # trailer
        self.head.next = self.tail
        self.tail.prev = self.head
```

```
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def insert_between(self, el, predecessorNode, succNode):
        new = dlNode(el)
        new.next = succNode
        new.prev = predecessorNode

        # don't forget to also set the prev and next of the succNode
and predecessorNode
        if succNode is not None:
            succNode.prev = new
        else:
            self.tail.prev = new
            new.next = self.tail

        if predecessorNode is not None:
            predecessorNode.next = new
        else:
            self.head.next = new
            new.prev = self.head
        self.size += 1
        return new

    def delete_element(self, el):
        # we only know the element, we need to find the location
        nd = self.head
        while nd is not None:
            if nd.element == el:
                break
            else:
                nd = nd.next

        if nd is None:
            print("this element doesn't exist in the linked list")
            return None
        else:
            predecessor = nd.prev
            successor = nd.next
            predecessor.next = successor
            successor.prev = predecessor
            self.size -= 1
            delete_node_element =nd.element
            nd.prev = nd.next = nd.element = None
```

```
            return delete_node_element


    def iterate(self):
        nd = self.head.next
        print("the elements in this doubly linked list are")
        while nd.next is not None:  # NOTICE THE ND.NEXT
            print(nd.element, end =' ')
            nd = nd.next
        print()
```

```
# test the doubly linked list code
dl = DoubleLinkedList()
node1 = dl.insert_between(3, None, None)   # 3
node2 = dl.insert_between(5, node1, None)   # 3 5
node3 = dl.insert_between(8, node1, node2)  # 3 8 5 (insert between 
node1 and node2)
node4 = dl.insert_between(4, node1, node3)  # 3 4 8 5
node4 = dl.insert_between(9, None, node1)  # 9 3 4 8 5
dl.delete_element(8)  # 9 3 4 5
dl.delete_element(3)  # 9 4 5
dl.iterate()
```

```
the elements in this doubly linked list are
9
4
5
```

Exercises

Q1: Concatenate Two Linked List

ANOTHER ANSWER VERSION can be seen at the singlyLinkedList class above!!

```
def concatenate(LHead, MHead):
    # we need to return a L' such that it contains elements from L and 
M
    Lprime = singleLinkedList() # L prime is the new concatenated 
linked list
    Lprime.head = LHead
    # find the tail
    nd = LHead
    while nd.pointer is not None:
        nd = nd.pointer
    nd.pointer = MHead
    return Lprime

L = singleLinkedList()
```

```
L.insert_head(10)
L.insert_tail(20)
L.insert_tail(30)

M = singleLinkedList()
M.insert_head(60)
M.insert_head(50)
M.insert_head(40)

L.iterate()
M.iterate()
concatenate(L.head,M.head).iterate()
```

```
The elements in the current linked list
10 20 30
The elements in the current linked list
40 50 60
0
The elements in the current linked list
10 20 30 40 50 60
```

Notice!

When we print the size of the new linked list, it shows that the size is **zero**!

Why?

Because, in this implementation, we are actually not really creating a linkedlist, we are just arranging the pointer of the two existing linked list to connect them together

Q2: LinkedQueue

```
class QueueNode:
    def __init__(self,e,node):
        self.element=e
        self.pointer=node

class LinkedQueue:
    def __init__(self):
        self.head=None
        self.tail=None
        self.size=0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size==0

    def first(self):
```

```
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.head.element

    def end(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.tail.element

    def dequeue(self):  # we dequeue from the head
        nd = self.head
        self.head = self.head.pointer
        self.size -= 1
        if self.size == 0:
            self.tail = None
        return nd.element

    def enqueue(self, e):  # we enqueue to the tail
        new = QueueNode(e, None)
        self.size += 1
        if self.size == 1:
            self.head = new
        else:
            self.tail.pointer= new
        self.tail = new


    def __str__(self):
        lsAns=[]
        nd=self.head
        while nd != None:
            lsAns.append(str(nd.element))
            nd=nd.pointer
        return str(lsAns)
```

Q3: LinkedStack

we only need one pointer (usually the self.head)

```
class StackNode:
    def __init__(self,e,node):
        self.element=e
        self.pointer=node

class LinkedStack:
    def __init__(self):
        self.head=None
```

```python
        self.size=0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def top(self):
        if self.is_empty():
            print('Stack is empty')
        else:
            return self.head.element

    def pop(self):
        if self.is_empty():
            print("Stack is empty")
        else:
            delete_node_element = self.head.element
            self.head = self.head.pointer
            self.size -= 1
            return delete_node_element

    def push(self, e):
        self.size += 1
        new = StackNode(e, None)
        new.pointer = self.head
        self.head = new

    def __str__(self):
        ll = []
        nd = self.head
        while nd is not None:
            ll.append(nd.element)
            nd = nd.pointer
        return str(ll[::-1])
        # OR use ll.reverse() and then return ll


## testing
s=LinkedStack()
s.push(1)
s.push(2)
s.push(3)
s.push(4)
print(s.pop())
print(s)
print(s.top())
```

```
4
[1, 2, 3]
3
```

Q4: Insertion Sort

```
## USING STANDARD LIST
def insertion(insertionList):
    for i in range(1, len(insertionList)):
        j = i-1
        x = insertionList[i]
        while j>=0 and insertionList[j]>x:
            insertionList[j+1] = insertionList[j]
            j = j-1
        # when insertionList[j] <= x or we reach the zero index
        insertionList[j+1] = x
    return insertionList

## using Doubly Linked List
def dlListInsertionSorT(myDlList):
    currentNode = myDlList.head.next.next  # because we want the first 
index, not the zero index
    while currentNode is not myDlList.tail:
        j = currentNode.prev
        x = currentNode.element
        while j is not myDlList.head and j.element > x:
            j.next.element = j.element  # do switching
            j = j.prev
        j.next.element = x
        currentNode = currentNode.next
    return myDlList

## Test our answer
L=DoubleLinkedList()
L.insert_between(1,L.head,L.tail)
L.insert_between(2,L.head,L.head.next)
L.insert_between(4,L.head,L.head.next)
L.insert_between(6,L.head,L.head.next)
L.insert_between(5,L.head,L.head.next)
L.insert_between(9,L.head,L.head.next)
L.insert_between(8,L.head,L.head.next)
L.insert_between(7,L.head,L.head.next)
L.iterate()
dlListInsertionSorT(L).iterate()

the elements in this doubly linked list are
7 8 9 5 6 4 2 1
the elements in this doubly linked list are
1 2 4 5 6 7 8 9
```