```
!date
```

```
Sun Mar 27 07:34:20 UTC 2022
```

Please run the above line to refresh the date before your submission.

CSCI-SHU 210 Data Structures

Recitation 8 Linked List

Name: MJ

NetID: mg7502

- For students who have recitation on Wednesday, you should submit your solutions by Friday 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by Saturday 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by Sunday 11:59pm.

No late submission is permitted. All solutions must be from your own work. Total points of the assignment is 100. (newdoc)

Part 1: Implement Deque (Double ended queue) using Double ended doubly linked list.

Q1. We've already implemented stack using Single ended singly linked list. Why?

````
```
# This is formatted as code
```

Your Answer: It is easy to add or pop the head and tail of the linked
````

```
# This is formatted as code
```

Your Answer: It is easy to add or pop the head and tail of the linked list

Q2. We've also implemented queue using Double ended singly linked list. Why?

Your Answer:

Q3. Implement class LinkedDeque.

```
class LinkedDeque:
    """Deque implementation using a doubly linked list for storage."""

    #-------------------------- nested _Node class --------------------------
    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        __slots__ = '_element', '_next', '_prev'         # streamline memory usage

        def __init__(self, element, prev, next):
            self._element = element
            self._prev = prev
            self._next = next



    #------------------------------- queue methods -------------------------------
    def __init__(self):
        """Create an empty deeue."""
        self._head = self._Node(None, None, None)
        self._tail = self._Node(None, None, None)
        self._head._next = self._tail
        self._tail._prev = self._head
        self._size = 0                      # number of elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def _insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return new node."""
        newest = self._Node(e, predecessor, successor)      # linked to neighbors
        predecessor._next = newest
        successor._prev = newest
        self._size += 1
        return newest

    def _delete_node(self, node):
        """Delete nonsentinel node from the list and return its element."""
        predecessor = node._prev
        successor = node._next
        predecessor._next = successor
        successor._prev = predecessor
        self._size -= 1
        element = node._element                             # record deleted element
        node._prev = node._next = node._element = None      # deprecate node
        return element                                      # return deleted element

    def first(self):
        if self.is_empty():
            raise Exception("empty")
        return self._head._next
    def last(self):
        if self.is_empty():
            raise Exception("empty")
        return self._tail._prev

    def delete_first(self):
        if self.is_empty():
            raise Exception("empty")
        self._size -= 1
        return self._delete_node(self.first())


    def delete_last(self):
        if self.is_empty():
            raise Exception("empty")
        self._size -= 1
        return self._delete_node(self.last())

    def add_first(self, e):
        head = self._head
        first = self._head._next
        self._insert_between(e, head, first)
```

```
        self._size += 1


    def add_last(self, e):
        last = self._tail._prev
        tail = self._tail
        self._insert_between(e, last, tail)
        self._size += 1



    def __str__(self):
        result = ["head <--> "]
        curNode = self._head._next
        while (curNode._next is not None):
            result.append(str(curNode._element) + " <--> ")
            curNode = curNode._next
        result.append("tail")
        return "".join(result)



def main():
    deque = LinkedDeque()
    for i in range(3):
        deque.add_first(i)
    for j in range(3):
        deque.add_last(j + 4)

    print(deque) # head <--> 2 <--> 1 <--> 0 <--> 4 <--> 5 <--> 6 <--> tail
    print("deleting first: ", deque.delete_first())   # 2
    print("deleting last: ", deque.delete_last())     # 6
    print(deque) # head <--> 1 <--> 0 <--> 4 <--> 5 <--> tail

if __name__ == '__main__':
    main()
```

```
head <--> 2 <--> 1 <--> 0 <--> 4 <--> 5 <--> 6 <--> tail
deleting first:  2
deleting last:  6
head <--> 1 <--> 0 <--> 4 <--> 5 <--> tail
```

Part 2: Single Linked List Exercises.

Q1. Implement function return_max(self) in class SingleLinkedList.

```
Traverse the single linked list and return the maximum element stored with in the linkedlist.
```

Q2. Implement function **iter**(self) in class SingleLinkedList.

```
Generate a forward iteration of the elements from self linkedlist. Remember to use keyword “yield”!
```

Q3. Implement function insert_after_kth_index(self, k, e) in class SingleLinkedList.

```
Insert element e (as a new node) after kth indexed node in self linkedlist.
For example,
    L1: 11-->22-->33-->44-->None
    L1.insert_after_kth_position(2, “Hi”) # 33 is the index 2.
    L1: 11-->22-->33-->”Hi”-->44-->None
```

```
class SingleLinkedList:

    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'         # streamline memory usage

        def __init__(self, element, next):      # initialize node's fields
            self._element = element               # reference to user's element
            self._next = next                     # reference to next node

    def __init__(self):
        """Create an empty linkedlist."""
        self._head = None
        self._size = 0

    def __len__(self):
        """Return the number of elements in the linkedlist."""
        return self._size

    def is_empty(self):
        """Return True if the linkedlist is empty."""
        return self._size == 0

    def top(self):
        """Return (but do not remove) the element at the top of the linkedlist.

        Raise Empty exception if the linkedlist is empty.
        """
        if self.is_empty():
            raise Exception('list is empty')
        return self._head._element              # head of list

    def insert_from_head(self, e):
        # Create a new link node and link it
        new_node = self._Node(e, self._head)
        self._head = new_node
        self._size += 1

    def delete_from_head(self):

        if self.is_empty():
            raise Exception('list is empty')
        to_return = self._head._element
        self._head = self._head._next
        self._size -= 1
        return to_return

    def __str__(self):
        result = []
        curNode = self._head
        while (curNode is not None):
            result.append(str(curNode._element) + "-->")
            curNode = curNode._next
        result.append("None")
        return "".join(result)

    def return_max(self):

        curr = self._head._next
        max = self._head._element
        while curr is not None:
            if curr._element > max:
                max = curr._element
            curr= curr._next
        return max

    def __iter__(self):

        curr = self._head
        while curr is not None:
            yield curr._element
            curr = curr._next

    def insert_after_kth_index(self, k, e):

        """
        L1: 11-->22-->33-->44-->None
```

```
        L1.insert_after_kth_index(2, "Hi")
        L1: 11-->22-->33-->"Hi"-->44-->None
        """
        prev = self._head
        for i in range(k-1):
            prev = prev._next
        next = prev._next
        el = self._Node(e,next)
        prev._next = el
        self._size += 1


def main():
    import random
    test_list = SingleLinkedList()
    for i in range(8):
        test_list.insert_from_head(random.randint(0, 20))
    print("Test list length 8, looks like:")
    print(test_list)
    print("---------------------------------------------------------")
    print("Maximum value within test list:", test_list.return_max())
    print("---------------------------------------------------------")
    print("Testing __iter__ ...........")
    for each in test_list:
        print(each, end = " ")
    print()
    print("---------------------------------------------------------")
    print("Testing insert_after_kth_index ...........")
    test_list.insert_after_kth_index(3, "Hi")
    print(test_list)
    print("---------------------------------------------------------")

if __name__ == '__main__':
    main()
```

```
Test list length 8, looks like:
17-->0-->5-->16-->6-->7-->5-->18-->None
---------------------------------------------------------
Maximum value within test list: 18
---------------------------------------------------------
Testing __iter__ ...........
17 0 5 16 6 7 5 18
---------------------------------------------------------
Testing insert_after_kth_index ...........
17-->0-->5-->Hi-->16-->6-->7-->5-->18-->None
---------------------------------------------------------
```

Part 3: Double Linked List Exercises.

Q1. Implement function split_after(self, index) in class DoubleLinkedList.

```
After called, split self DoubleLinkedList into two separate lists.
Self list contains first section, return a new list that contains the
second section.

For example,
    L1: head<-->1<-->2<-->3<-->4-->tail
    L2 = L1.split_after(2)
    L1: head<-->1<-->2<-->3-->tail
    L2: head<-->4-->tail
```

Q2. Implement function merge(self, other) in class DoubleLinkedList.

```
This function adds other DoubleLinkedList to the end of self
DoubleLinkedList. After merging, other list becomes empty.
For example,
```

```
L1: head<-->1<-->2<-->3-->tail
L2: head<-->4-->tail
L1.merge(L2)
L1: head<-->1-->2-->3<-->4-->tail
L2: head<-->tail
```

```
class DoubleLinkedList:

    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        __slots__ = '_element', '_next', '_prev'         # streamline memory usage

        def __init__(self, element, prev, next):      # initialize node's fields
            self._element = element               # reference to user's element
            self._prev = prev                     # reference to prev node
            self._next = next                     # reference to next node



    def __init__(self):
        """Create an empty linkedlist."""
        self._head = self._Node(None, None, None)
        self._tail = self._Node(None, None, None)
        self._head._next = self._tail
        self._tail._prev = self._head
        self._size = 0


    def __len__(self):
        """Return the number of elements in the list."""
        return self._size

    def is_empty(self):
        """Return True if the list is empty."""
        return self._size == 0

    def _insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return new node."""
        newest = self._Node(e, predecessor, successor)      # linked to neighbors
        predecessor._next = newest
        successor._prev = newest
        self._size += 1
        return newest

    def _delete_node(self, node):
        """Delete nonsentinel node from the list and return its element."""
        predecessor = node._prev
        successor = node._next
        predecessor._next = successor
        successor._prev = predecessor
        self._size -= 1
        element = node._element                           # record deleted element
        node._prev = node._next = node._element = None      # deprecate node
        return element                                    # return deleted element

    def first(self):
        """Return (but do not remove) the element at the front of the list.
        Raise Empty exception if the list is empty.
        """
        if self.is_empty():
            raise Exception('list is empty')
        return self._head._next._element              # front aligned with head of list

    def last(self):
        """Return (but do not remove) the element at the end of the list.

        Raise Empty exception if the list is empty.
        """
        if self.is_empty():
            raise Exception('list is empty')
        return self._tail._prev._element


    def delete_first(self):
        """Remove and return the first element of the list.
```

```
        Raise Empty exception if the list is empty.
        """
        if self.is_empty():
            raise Exception('list is empty')
        return self._delete_node(self._head._next)

    def delete_last(self):
        """Remove and return the last element of the list.

        Raise Empty exception if the list is empty.
        """
        if self.is_empty():
            raise Exception('list is empty')
        return self._delete_node(self._tail._prev)


    def add_first(self, e):
        """Add an element to the front of list."""
        self._insert_between(e, self._head, self._head._next)


    def add_last(self, e):
        """Add an element to the back of list."""
        self._insert_between(e, self._tail._prev, self._tail)


    def __str__(self):
        result = ['head <--> ']
        curNode = self._head._next
        while (curNode._next is not None):
            result.append(str(curNode._element) + " <--> ")
            curNode = curNode._next
        result.append("tail")
        return "".join(result)

    def split_after(self, index):
        """
        :index: Int -- split after this indexed node.
        (index start from zero)

        split self DoubleLinkedList into two separate lists.

        ***head/tail sentinel nodes does not count for indexing.

        :return: A new DoubleLinkedList object that contains the second section.
        """
        ret = DoubleLinkedList()
        lst = self._head
        start = self._head._next
        for _ in range(index+1):
            lst = lst._next
            start = start._next
        lst._next = self._tail
        ret._head._next = start
        return ret

    def merge(self, otherlist):
        """
        :otherlist: DoubleLinkedList -- another DoubleLinkedList to merge.

        For example:
        L1: head<-->1<-->2<-->3-->tail
        L2: head<-->4-->tail
        L1.merge(L2)
        L1: head<-->1-->2-->3<-->4-->tail
        L2: head<-->tail
        :return: Nothing.
        """
        last = self._head
        while last._next != self._tail:
            last = last._next
        last._next = otherlist._head._next



def main():
    import random
```

```
    test_list = DoubleLinkedList()
    for i in range(8):
        test_list.add_first(random.randint(0, 20))
    print("Test list length 8, looks like:")
    print(test_list)
    print("-------------------------------------------------------")
    print("Split after index 5:")
    new_list = test_list.split_after(5)
    print("Original List:", test_list)
    print("The second part:", new_list)
    print("-------------------------------------------------------")
    print("Merging original list with the second part:")
    test_list.merge(new_list)
    print("Original List:", test_list)
    print("The second part:", new_list)
    print("-------------------------------------------------------")

if __name__ == '__main__':
    main()
```

```
Test list length 8, looks like:
head <--> 18 <--> 12 <--> 3 <--> 15 <--> 2 <--> 0 <--> 1 <--> 15 <--> tail
```