```
!date
```

```
Mon Apr  4 03:21:10 UTC 2022
```

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

CSCI-SHU 210 Data Structures

Recitation 8 Linked List

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?

Answer:

```
push() function is the same as insert_from_head() function in the Single-ended Singly Linked List,
pop() function is the same as delete_from_head() function in the Single-ended Singly Linked List,
and the runtime for both are O(1).
```

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

Answer:

```
enqueue() function is the same as add_last() function in the Double-ended Singly Linked List,
dequeue() function is the same as delete_first() function in the Double-ended Singly Linked List,
and the runtime for both are O(1).
```

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):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the deque is empty.
        """
        # Your code
        if self.is_empty():
            raise Exception('Deque is empty')
        return self._head._next._element                  # front located at head._next

        pass

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

        Raise Empty exception if the deque is empty.
        """
        # Your code
        if self.is_empty():
            raise Exception('Deque is empty')
        return self._tail._prev._element                  # front located at tail._prev
        pass

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

        Raise Empty exception if the queue is empty.
        """
        # Your code
        if self.is_empty():
            raise Exception('Deque is empty')
        return self._delete_node(self._head._next)
        pass

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

        Raise Empty exception if the queue is empty.
        """
        # Your code
        if self.is_empty():
            raise Exception('Deque is empty')
        return self._delete_node(self._tail._prev)
```

```
            pass


    def add_first(self, e):
            """Add element e to the front of deque."""
            # Your code
            self._insert_between(e, self._head, self._head._next)
            pass


    def add_last(self, e):
            """Add an element to the back of deque."""
            # Your code
            self._insert_between(e, self._tail._prev, self._tail)
            pass



    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):
        """Add element e to the head of the linkedlist."""
        # 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):
        """Remove and return the element from the head of the linkedlist.

        Raise Empty exception if the linkedlist is empty.
        """
        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):
        """
        return the maximum element stored with in self.
```

```
            For example, 9 --> 5 --> 21 --> 1 --> None should return 21.
            :return: The maximum element.
            """
            if self.is_empty():
                raise Exception()
            temp = self._head
            maximum = self._head._element
            while temp is not None:
                if maximum < temp._element:
                    maximum = temp._element
                temp = temp._next
            return maximum
            pass

        def __iter__(self):
            """
            generate a forward iteration of the elements from self list.

            In other words, for each in SingleLinkedList object will become working.

            :return: No return. Use yield instead.
            """
            temp = self._head
            while temp is not None:
                yield temp._element
                temp = temp._next
            pass

        def insert_after_kth_index(self, k, e):
            """
            :param k: Int -- insert after this indexed node.
            :param e: Any -- the value we are storing

            Insert element e (as a new node) after kth indexed node in self linkedlist.
            (index start from zero)

            L1: 11-->22-->33-->44-->None
            L1.insert_after_kth_index(2, "Hi")
            L1: 11-->22-->33-->"Hi"-->44-->None

            :return: Nothing.
            """
            # locate kth node
            if k > self._size:
                raise Exception()

            temp = self._head
            for i in range(k):
                temp = temp._next

            new = self._Node(e, None)
            new._next = temp._next
            temp._next = new
            self._size += 1
            pass


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:
15-->6-->6-->10-->7-->11-->5-->18-->None
--------------------------------------------------------
Maximum value within test list: 18
--------------------------------------------------------
Testing __iter__ ...........
15 6 6 10 7 11 5 18
--------------------------------------------------------
Testing insert_after_kth_index ...........
15-->6-->6-->10-->Hi-->7-->11-->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):
```

```python
    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):            # O(index) solution
        """
        :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.
        """
        new = DoubleLinkedList()

        # O(index loop), locate split point
        temp = self._head
        for i in range(index + 1):
            temp = temp._next

        # Looks complicated, but O(1), which is better.
        new._head._next = temp._next     # New's head sentinel's next
        temp._next._prev = new._head     # one after split's prev

        self._tail._prev._next = new._tail   # One before tail's next
        new._tail._prev = self._tail._prev   # new tail's prev

        temp._next = self._tail   # one before split's next
        self._tail._prev = temp   # self tail's prev

        new._size = self._size - index - 1
        self._size = index + 1

        return new
        pass

    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.
        """
        # Looks complicated, but O(1), which is better.
        self._tail._prev._next = otherlist._head._next   # Link self last one with other's first
        otherlist._head._next._prev = self._tail._prev   # Link self last one with other's first

        otherlist._tail._prev._next = self._tail   # Set up new tail
```

```
            self._tail._prev = otherlist._tail._prev    # Set up new tail

            self._size = self._size + otherlist._size    # Set up size

            otherlist.__init__()
            #otherlist._head._next = otherlist._tail
            #otherlist._tail._prev = otherlist._head
            #otherlist._size=0
            pass



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 <--> 11 <--> 5 <--> 13 <--> 1 <--> 13 <--> 19 <--> 3 <--> 6 <--> tail
--------------------------------------------------------
Split after index 5:
Original List: head <--> 11 <--> 5 <--> 13 <--> 1 <--> 13 <--> 19 <--> tail
The second part: head <--> 3 <--> 6 <--> tail
--------------------------------------------------------
Merging original list with the second part:
Original List: head <--> 11 <--> 5 <--> 13 <--> 1 <--> 13 <--> 19 <--> 3 <--> 6 <--> tail
The second part: head <--> tail
--------------------------------------------------------
```

✓ 0 秒 完成时间：11:21