```
1 !date
```

```
Fri Apr  1 06:39:14 UTC 2022
```

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

CSCI-SHU 210 Data Structures

Recitation 8 Linked List

Name: Peter Yuncheng Yao

NetID: yy4108

- For students who have recitation on Wednesday, you should submit your solutions by Friday Apr 1 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by Saturday Apr 2 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by Sunday Apr 3 11:59pm.

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

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?

Your Answer: because we only push and pop at one end of the linked list, so there is no need to used a double ended ll. and since it is a linked list, the push and pop operation will always take o(1) time, without amortization.

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

Your Answer: the FIFO feature of the queue requires operation on both ends of the LL, so we used a double ended SLL.

Q3. Implement class LinkedDeque.

```
 1 class LinkedDeque:
 2     """Deque implementation using a doubly linked list
 3
 4     #------------------------- nested _Node class ----
 5     class _Node:
 6         """Lightweight, nonpublic class for storing a d
 7         __slots__ = '_element', '_next', '_prev'
 8
 9         def __init__(self, element, prev, next):
10             self._element = element
11             self._prev = prev
12             self._next = next
13
14
15
16     #------------------------------ queue methods ----
17     def __init__(self):
18         """Create an empty deeue."""
19         self._head = self._Node(None, None, None)
20         self._tail = self._Node(None, None, None)
21         self._head._next = self._tail
22         self._tail._prev = self._head
23         self._size = 0                          # number of
24
25     def __len__(self):
26         """Return the number of elements in the queue."
27         return self._size
28
29     def is_empty(self):
30         """Return True if the queue is empty."""
31         return self._size == 0
32
33     def _insert_between(self, e, predecessor, successor
34         """Add element e between two existing nodes and
35         newest = self._Node(e, predecessor, successor)
36         predecessor._next = newest
37         successor._prev = newest
38         self._size += 1
39         return newest
40
```

```
41      def _delete_node(self, node):
42          """Delete nonsentinel node from the list and re
43          predecessor = node._prev
44          successor = node._next
45          predecessor._next = successor
46          successor._prev = predecessor
47          self._size -= 1
48          element = node._element
49          node._prev = node._next = node._element = None
50          return element
51
52      def first(self):
53          """Return (but do not remove) the element at th
54
55          Raise Empty exception if the deque is empty.
56          """
57          # Your code
58          # pass
59          assert self._size!=0, "Empty exception"
60          return self._head._next
61
62
63      def last(self):
64          """Return (but do not remove) the element at th
65
66          Raise Empty exception if the deque is empty.
67          """
68          # Your code
69          assert self._size!=0, "empty exception"
70          return self._tail._prev
71
72
73      def delete_first(self):
74          """Remove and return the first element of the d
75
76          Raise Empty exception if the queue is empty.
77          """
78          # Your code
79          assert self._size!=0, "empty exception"
80          temp=self._head._next
81          self._head._next=self._head._next._next
```

```
 82         self._head._next._prev=self._head
 83         ret=temp._element
 84         temp._prev=temp._next=temp._element=None
 85         self._size-=1
 86         return ret
 87 
 88     def delete_last(self):
 89         """Remove and return the last element of the de
 90 
 91         Raise Empty exception if the queue is empty.
 92         """
 93         # Your code
 94         assert self._size!=0, "empty exception"
 95         temp=self._tail._prev
 96         before=temp._prev
 97         before._next=self._tail
 98         self._tail._prev=before
 99         ret=temp._element
100         temp._prev=temp._next=temp._element=None
101         self._size-=1
102 
103         return ret
104 
105 
106     def add_first(self, e):
107         """Add element e to the front of deque."""
108         # Your code
109         temp= self._Node(e, self._head, self._head._nex
110         after=self._head._next
111         after._prev=temp
112         self._head._next=temp
113         self._size+=1
114 
115         return
116 
117 
118     def add_last(self, e):
119         """Add an element to the back of deque."""
120         # Your code
121         temp=self._Node(e, self._tail._prev, self._tail
122         before=self._tail._prev
```

```
123         before._next=temp
124         self._tail._prev=temp
125         self._size+=1
126 
127         return
128 
129 
130 
131     def __str__(self):
132         result = ["head <--> "]
133         curNode = self._head._next
134         while (curNode._next is not None):
135             result.append(str(curNode._element) + " <--
136             curNode = curNode._next
137         result.append("tail")
138         return "".join(result)
139 
140 
141 
142 def main():
143     deque = LinkedDeque()
144     for i in range(3):
145         deque.add_first(i)
146     for j in range(3):
147         deque.add_last(j + 4)
148 
149     print(deque) # head <--> 2 <--> 1 <--> 0 <--> 4 <--
150     print("deleting first: ", deque.delete_first())   #
151     print("deleting last: ", deque.delete_last())     #
152     print(deque) # head <--> 1 <--> 0 <--> 4 <--> 5 <--
153 
154 if __name__ == '__main__':
155     main()
156 
157 
```

```
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 linkedli
```

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

```
Generate a forward iteration of the elements from self linkedlist. Remember to use keyworc
```

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 s
        __slots__ = '_element', '_next'         # strea

        def __init__(self, element, next):      # initi
            self._element = element               # ref
            self._next = next                     # ref

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

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

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

    def top(self):
```

```
25          """Return (but do not remove) the element at th
26
27          Raise Empty exception if the linkedlist is empt
28          """
29          if self.is_empty():
30              raise Exception('list is empty')
31          return self._head._element                  # head
32
33      def insert_from_head(self, e):
34          """Add element e to the head of the linkedlist.
35          # Create a new link node and link it
36          new_node = self._Node(e, self._head)
37          self._head = new_node
38          self._size += 1
39
40      def delete_from_head(self):
41          """Remove and return the element from the head
42
43          Raise Empty exception if the linkedlist is empt
44          """
45          if self.is_empty():
46              raise Exception('list is empty')
47          to_return = self._head._element
48          self._head = self._head._next
49          self._size -= 1
50          return to_return
51
52      def __str__(self):
53          result = []
54          curNode = self._head
55          while (curNode is not None):
56              result.append(str(curNode._element) + "-->"
57              curNode = curNode._next
58          result.append("None")
59          return "".join(result)
60
61      def return_max(self):
62          """
63          return the maximum element stored with in self.
64
65          For example, 9 --> 5 --> 21 --> 1 --> None shou
```

```
66          :return: The maximum element.
67          """
68          if self.is_empty():
69              raise Exception("the list is empty")
70          temp_max=-float("inf")
71          walk=self._head
72          while walk:
73              if walk._element>temp_max:
74                  temp_max=walk._element
75              walk=walk._next
76          return temp_max
77
78
79
80      def __iter__(self):
81          """
82          generate a forward iteration of the elements fr
83
84          In other words, for each in SingleLinkedList ob
85
86          :return: No return. Use yield instead.
87          """
88          # yield "Not working, change this"
89          walk=self._head
90          while walk:
91              yield walk._element
92              walk=walk._next
93
94      def insert_after_kth_index(self, k, e):
95          """
96          :param k: Int -- insert after this indexed node
97          :param e: Any -- the value we are storing
98
99          Insert element e (as a new node) after kth inde
100         (index start from zero)
101
102         L1: 11-->22-->33-->44-->None
103         L1.insert_after_kth_index(2, "Hi")
104         L1: 11-->22-->33-->”Hi”-->44-->None
105
106         :return: Nothing.
```

```
107         """
108         count=0
109         walk=self._head
110         while count<k:
111             walk=walk._next
112             count+=1
113         temp=walk._next
114         walk._next=self._Node(e, temp)
115 
116         return
117 
118 
119 def main():
120     import random
121     test_list = SingleLinkedList()
122     for i in range(8):
123         test_list.insert_from_head(random.randint(0, 20
124     print("Test list length 8, looks like:")
125     print(test_list)
126     print("-----------------------------------------
127     print("Maximum value within test list:", test_list.
128     print("-----------------------------------------
129     print("Testing __iter__ ............")
130     for each in test_list:
131         print(each, end = " ")
132     print()
133     print("-----------------------------------------
134     print("Testing insert_after_kth_index ............"
135     test_list.insert_after_kth_index(3, "Hi")
136     print(test_list)
137     print("-----------------------------------------
138 
139 if __name__ == '__main__':
140     main()
141 
142 
```

```
Test list length 8, looks like:
20-->15-->12-->12-->19-->2-->17-->17-->None
-----------------------------------------------------
Maximum value within test list: 20
-----------------------------------------------------
Testing __iter__ ............
20 15 12 12 19 2 17 17
```

```
------------------------------------------------------
Testing insert_after_kth_index .............
20-->15-->12-->12-->Hi-->19-->2-->17-->17-->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
```

```
 1 class DoubleLinkedList:
 2
 3     class _Node:
 4         """Lightweight, nonpublic class for storing a d
 5         __slots__ = '_element', '_next', '_prev'
 6
 7         def __init__(self, element, prev, next):      #
 8             self._element = element                # ref
 9             self._prev = prev                      # ref
10             self._next = next                      # ref
11
12
```

```
12
13
14
15      def __init__(self):
16          """Create an empty linkedlist."""
17          self._head = self._Node(None, None, None)
18          self._tail = self._Node(None, None, None)
19          self._head._next = self._tail
20          self._tail._prev = self._head
21          self._size = 0
22
23
24      def __len__(self):
25          """Return the number of elements in the list.""
26          return self._size
27
28      def is_empty(self):
29          """Return True if the list is empty."""
30          return self._size == 0
31
32      def _insert_between(self, e, predecessor, successor
33          """Add element e between two existing nodes and
34          newest = self._Node(e, predecessor, successor)
35          predecessor._next = newest
36          successor._prev = newest
37          self._size += 1
38          return newest
39
40      def _delete_node(self, node):
41          """Delete nonsentinel node from the list and re
42          predecessor = node._prev
43          successor = node._next
44          predecessor._next = successor
45          successor._prev = predecessor
46          self._size -= 1
47          element = node._element
48          node._prev = node._next = node._element = None
49          return element
50
51      def first(self):
52          """Return (but do not remove) the element at th
53          Raise Empty exception if the list is empty.
54          """
```

```
54
55          if self.is_empty():
56              raise Exception('list is empty')
57          return self._head._next._element                #
58
59      def last(self):
60          """Return (but do not remove) the element at th
61
62          Raise Empty exception if the list is empty.
63          """
64          if self.is_empty():
65              raise Exception('list is empty')
66          return self._tail._prev._element
67
68
69      def delete_first(self):
70          """Remove and return the first element of the l
71
72          Raise Empty exception if the list is empty.
73          """
74          if self.is_empty():
75              raise Exception('list is empty')
76          return self._delete_node(self._head._next)
77
78      def delete_last(self):
79          """Remove and return the last element of the li
80
81          Raise Empty exception if the list is empty.
82          """
83          if self.is_empty():
84              raise Exception('list is empty')
85          return self._delete_node(self._tail._prev)
86
87
88      def add_first(self, e):
89          """Add an element to the front of list."""
90          self._insert_between(e, self._head, self._head.
91
92
93      def add_last(self, e):
94          """Add an element to the back of list."""
95          self._insert_between(e, self._tail._prev, self.
96
```

```
 96
 97
 98     def __str__(self):
 99         result = ['head <--> ']
100         curNode = self._head._next
101         while (curNode._next is not None):
102             result.append(str(curNode._element) + " <--
103             curNode = curNode._next
104         result.append("tail")
105         return "".join(result)
106
107     def split_after(self, index):
108         """
109         :index: Int -- split after this indexed node.
110         (index start from zero)
111
112         split self DoubleLinkedList into two separate l
113
114         ***head/tail sentinel nodes does not count for
115
116         :return: A new DoubleLinkedList object that con
117         """
118         walk=self._head._next
119         count=0
120         while count<index:
121             walk=walk._next
122             count+=1
123         new_tail=self._Node(None,None,None)
124         walk._prev._next=new_tail
125         walk._prev=None
126         ret=DoubleLinkedList()
127         while walk and walk._element!=None:
128             ret.add_last(walk._element)
129             walk=walk._next
130         return ret
131
132
133
134
135
136     def merge(self, otherlist):
137         """
```

```
138         :otherlist: DoubleLinkedList -- another DoubleL
139 
140         For example:
141         L1: head<-->1<-->2<-->3-->tail
142         L2: head<-->4-->tail
143         L1.merge(L2)
144         L1: head<-->1-->2-->3<-->4-->tail
145         L2: head<-->tail
146         :return: Nothing.
147         """
148         walk=otherlist._head._next
149         # while walk and walk._element!=None:
150         #     # print(walk._element)
151         #     # temp=self._Node(walk._element,self._tai
152         #     # self._tail._prev._next=temp
153         #     # self._tail._prev=temp
154         #     # self._size+=1
155 
156         #     # print(self)
157         #     walk=walk._next
158         #     otherlist.delete_first()
159         #     self.add_last(walk._element)
160         while not otherlist.is_empty():
161             self.add_last(otherlist.first())
162             # print(str(self))
163             otherlist.delete_first()
164 
165 
166 
167 
168 
169 
170 def main():
171     import random
172     test_list = DoubleLinkedList()
173     for i in range(8):
174         test_list.add_first(random.randint(0, 20))
175     print("Test list length 8, looks like:")
176     print(test_list)
177     print("----------------------------------------
178     print("Split after index 5:")
179     new_list = test_list.split_after(5)
```

```
180     print("Original List:", test_list)
181     print("The second part:", new_list)
182     print("-----------------------------------------------
183     print("Merging original list with the second part:"
184     test_list.merge(new_list)
185     print("Original List:", test_list)
186     print("The second part:", new_list)
187     print("-----------------------------------------------
188
189 if __name__ == '__main__':
190     main()
191
192
```

```
Test list length 8, looks like:
head <--> 8 <--> 16 <--> 9 <--> 16 <--> 18 <--> 13 <--> 13 <--> 1 <--> tail
------------------------------------------------------------
Split after index 5:
Original List: head <--> 8 <--> 16 <--> 9 <--> 16 <--> 18 <--> tail
The second part: head <--> 13 <--> 13 <--> 1 <--> tail
------------------------------------------------------------
Merging original list with the second part:
Original List: head <--> 8 <--> 16 <--> 9 <--> 16 <--> 18 <--> tail
The second part: head <--> tail
------------------------------------------------------------
```