

```

1 #####
2 # DCS 229 -- Linked List
3 # Linked List implementation
4 # Date: 03/13/2026
5 # Name: Benjamin Adovasio
6 # Resources Used: https://docs.python.org/3/tutorial/datastructures.html
7 #####
8 from __future__ import annotations
9
10 # https://docs.python.org/3/tutorial/errors.html#user-defined-exceptions
11 # Want to define our own custom Exception class...
12
13 class EmptyError(Exception):
14     def __init__(self, message: str) -> None:
15         super().__init__(message)
16         self.message = message
17
18 class Node[T]:
19     ''' class to implement a single node object in a
20         singly-linked
21         linked list '''
22     def __init__(self, data: T):
23         self.data = data
24         self.next = None # points to another Node
25         object
26
27     def get_data(self) -> T:
28         return self.data
29
30     def set_data(self, value: T) -> None:
31         self.data = value
32
33     def get_next(self) -> Node[T]:
34         return self.next
35
36     def set_next(self, next_node: Node[T]) -> None:

```

```

35         self.next = next_node
36
37 class LinkedList[T]:
38     ''' class to implement a singly-linked linked
    list '''
39
40     def __init__(self) -> None:
41         self.head = None    # the head pointer in the
    linked list
42         self.size = 0
43
44     def __len__(self) -> int:
45         ''' returns the number of nodes in the linked
    list
46
47         Returns:
48             int - representing the number of nodes in
    the list
49         '''
50         return self.size
51
52
53     def insert_head(self, value: T) -> None:
54         ''' adds the given T-type data value to the
    front of the linked list
55         Parameters:
56             value: a type T data item to be included
    as the data in the inserted Node
57         Returns:
58             nothing
59         '''
60
61         ## YOUR CODE HERE ##
62
63         new_node = Node(value) # create a new node
    new_node
64         new_node.set_next(self.head) # set the new
    node's next pointer to the current head of the list

```

```

65         self.head = new_node # update the head
        pointer to point to the new node
66         self.size += 1 # increase the size of the
        list by 1
67
68     def insert_tail(self, value: T) -> None:
69         ''' adds the given T-type data value to the
        end of the linked list
70         Parameters:
71             value: a type T data item to be included
        as the data in the inserted Node
72         Returns:
73             nothing
74         '''
75
76         ## YOUR CODE HERE ##
77         """
78         The insert_tail method creates a new node
        with the given value and adds it to the end of the
        linked list. If the list is empty, it sets the head
        pointer to the new node. Otherwise, it traverses the
        list until it reaches the last node and updates its
        next pointer to point to the new node. Finally, it
        increments the size of the list by 1.
79         """
80         new_node = Node(value) # create a new node
        new_node
81
82         if self.head is None:
83             self.head = new_node #if the list is
        empty, set the head pointer to the new node
84         else: #else go through list until we reach
        end and set the last node's next pointer to the new
        node
85             current = self.head
86             while current.get_next() is not None:
87                 current = current.get_next()
88             current.set_next(new_node)

```

```

89
90         self.size += 1 #increase the size of the
    list by 1
91
92     def remove_head(self) -> T:
93         ''' removes the first Node in the linked
    list, returning the data item
94             inside that Node... Remember to handle
    the special case of an
95             empty list (what should the head
    pointers be in that case?)
96             and remember to update the head pointer
    when appropriate.
97             Returns:
98                 a T type data item extracted from the
    removed Node
99             Raises:
100                 EmptyError exception if list is empty
101             '''
102
103             # raise an error if list is empty
104             # https://docs.python.org/3/tutorial/errors.
    html#user-defined-exceptions
105             if self.size == 0:
106                 raise EmptyError("Cannot remove from an
    empty list")
107
108
109             ## YOUR CODE HERE ##
110
111             removed_node = self.head # store the current
    head node in a variable removed_node
112             self.head = self.head.get_next() # update
    the head pointer to point to the next node in the
    list (which could be None if there was only one node
    )
113
114             self.size -= 1 #decrease the size of the

```

```

114 list by 1
115         return removed_node.get_data() # return the
           data item from the removed node
116
117     def remove_tail(self) -> T:
118         ''' removes the last Node in the linked list
           , returning the data item
119             inside that Node... Remember to handle
           the special case of an
120             empty list
121
122             Returns:
123                 a T type data item extracted from the
           removed Node
124             Raises:
125                 EmptyError exception if list is empty
126             ...
127             # raise an error if list is empty
128             if self.size == 0:
129                 raise EmptyError("Cannot remove from an
           empty list")
130
131             ## YOUR CODE HERE ##
132
133             """
134             If the list has only one node, we can simply
           remove the head and return its data. Otherwise, we
           need to traverse the list to find the second-to-last
           node, update its next pointer to None, and return
           the data from the last node.
135             """
136             if self.head.get_next() is None: #continue
           if the list has only one node
137                 removed_data = self.head.get_data() #
           return the data
138                 self.head = None #set the head to none
139                 self.size -= 1 #decrease the size of the
           list by 1

```

```

140         return removed_data #return the data
        from the removed node
141
142         current = self.head #start from the head
        node
143
144         while current.get_next().get_next() is not
        None: #while the next node's next pointer is not
        None, keep going
145             current = current.get_next() #move
        current to the next node
146
147         removed_data = current.get_next().get_data()
148         current.set_next(None)
149
150         self.size -= 1 #decrease the size of the
        list by 1
151         return removed_data #return the data from
        the removed node
152
153
154     def __str__(self):
155         ''' returns a str representation of the
        linked list data
156         Returns:
157             an str representation of the linked list
        , showing head pointer
158             and data tiems
159         '''
160         str_ = "head->"
161
162         # start out at the head Node, and walk
        through Node by Node until we
163         # reach the end of the linked list (i.e.,
        the ._next entry is None)
164         ptr_ = self.head
165         while ptr_ is not None:
166             str_ += "[" + str(ptr_.get_data()) +

```

```
166 "]"->"
167         ptr_ = ptr_.get_next()  # move ptr_ to
    the next Node in the linked list
168
169         if self.head != None:
170             str_ = str_[:-2]  # remove trailing "->"
171             str_ += "<-tail"
172             return str_
173
174 def main():
175     # create a LinkedList and try out some various
    adds and removes
176     ll = LinkedList()
177     try:
178         ll.remove_head()
179     except EmptyError as err:
180         print(err)
181
182     ll.insert_head(8)
183     assert len(ll) == 1
184     removed = ll.remove_head()
185     assert removed == 8
186     assert len(ll) == 0
187
188     ll.insert_tail(6)
189     ll.insert_tail(7)
190     ll.insert_tail(5)
191     print(ll)
192     assert len(ll) == 3
193
194     removed = ll.remove_tail()
195     assert removed == 5
196     assert len(ll) == 2
197
198     # ADD MORE TESTS
199
200     # test inserting multiple at head
201     ll.insert_head(1)
```

```
202     ll.insert_head(2)
203     ll.insert_head(3)
204     assert len(ll) == 5
205     print(ll)
206
207     # test removing head repeatedly
208     assert ll.remove_head() == 3
209     assert ll.remove_head() == 2
210     assert len(ll) == 3
211
212     # test removing tail until empty
213     assert ll.remove_tail() == 7
214     assert ll.remove_tail() == 6
215     assert ll.remove_tail() == 1
216     assert len(ll) == 0
217
218     # test removing from empty again
219     try:
220         ll.remove_tail()
221     except EmptyError:
222         print("Correctly caught empty list error")
223
224 if __name__ == "__main__":
225     main()
```