CPE 202 Lab 4 Winter 2020

Lab4: Doubly Linked List

Implement an ordered list using doubly linked list

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

Implement the following operations for an **ordered list of integers ordered in ascending order** using a **doubly linked list**. Create a Node class with two pointers, next and prev. The "**head**" of the list be where the "smallest item is and let "**tail**" be where the largest item is.

Use the following data definition to implement doubly linked lists:

```
class Node:
    """ A node of a list
    Attributes:
        val (int): the payload
        next (Node): the next item in the list
        prev (Node): the previous item in the list
    """
```

OrderedList () creates a new ordered list that is empty. It needs no parameters and returns an empty list. The following functions belong to OrderedList, thus they are methods. The first argument, self, is omitted in the descriptions below, but you need to have it as the first argument. **It is recommended to use recursive helper functions to implement most of the functions.** For example, create add_helper(head, tail, item)->(Node, Node) function which returns new head and tail.

Use the following data definition for the OrderedList:

```
class OrderedList:
    """an ordered list
    Attributes:
        head (Node): a ponter to the head of the list
        tail (Node): a pointer to the tail of the list
        num_items (int): the number of items stored in the list
    """
```

Download a starter file, ordered_list.py from Polylearn.

- **add(item)** adds a new item (int) to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- **remove(item)** removes the item (int) from the list. It needs the item and modifies the list. Return the position of removed item if it is in the list, otherwise **raise ValueError**.
- **search_forward(item)** searches for the item (int) in the list. It needs the item and returns the boolean value True if the item is in the list and False if the item is not in the list.
- **search_backward(item)** searches for the item (int) in the list starting from the tail of the list. It needs the item and returns the boolean value True if the item is in the list and False if the item is not in the list.
- **is_empty ()** tests to see whether the list is empty. It needs no parameters and returns a boolean value. True if the list is empty and False if any items are in the list.
- **size ()** returns the number of items in the list. It needs no parameters and returns an integer.
- **index (item)** returns the position of item (int) in the list. It needs the item and returns the index. If it is not in the list, **it raises LookupError**.
- **pop (pos=None)** takes an optional argument pos and removes and returns the last item in the list if the argument is not passed. If the argument is passed, it removes and returns the item at position pos. If there is no item in that position (the position is out of bound), **it raises IndexError**. **pop(pos) should compare pos to the size of the list and search from the head if pos <= size/2 and from the rear if pos > size/2. Think about the advantage of doing this as opposed to always traversing from the head.**

An example:

```
    def search_forward(self, item):
        """searches a specified item in the list starting from the head.
        Args:
```

```
        item (int): the value to be searched in the list
    Returns:
        bool: True if found, False otherwise.
    """
    return self.search_forward_helper(self.head, item)


def search_forward_helper(self, node, item):
    """a helper function for searching an item forward
    Args:
        node (Node): a node
        int (item): a value to search for in the list
    Returns:
        bool: True if found, False otherwise.
    """
    if node is None:
        return False
    if node.val == item:
        return True
    return self.search_forward_helper(node.next, item)
```

Write separate test for each function.

Submit your work to the grader and then submit two files to PolyLearn

Zip two files, ordered_list.py (containing all your implementations including OrderedList and Node classes) and ordered_list_tests.py, into one zip file and submit it to polylearn. Do not include a directory structure in your zip file.

Follow the 6 steps of the design recipe and do not forget to provide docstrings for class and function definitions. For each public method (meaning not including boilerplate methods and helper functions) in OrderedList class, consider its time complexity in Big-O notation. All your classes need to have the three boilerplate methods(__init__, __repr__, and __eq__). Pylint (for Python3) will be used to rate your coding style.