Programming assignment 2: Queues, stacks and singly-linked lists

20% Implement the *singly-linked list* class ***LinkedList*** including the following operations:

- push_back
  - Takes a parameter and adds its value to the back of the list
- push_front
  - Takes a parameter and adds its value to the front of the list
- pop_front
  - Removes the item from the front of the list and ***returns*** *its value*
    - *If the list is empty, return None*
- pop_back
  - Removes the item from the back of the list and ***returns*** *its value*
    - *If the list is empty, return None*
- get_size
  - Returns the number of items currently in the list
- __str__
  - Returns a string with all the items in the list, separated by a single space

*For full marks, implement all these operations (*apart from __str__ and pop_back*) with time complexity O(1)*

40% Stack and Queue

You are given a completed implementation of the class **ArrayDeque**
Implement the abstract data type(**ADT**) classes **Stack** and **Queue** using **ArrayDeque** and **LinkedList** for the underlying implementations of **Stack** and **Queue**.
You must use both **ArrayDeque** and **LinkedList** when implementing **Stack** and **Queue** but can select which class to use for which **ADT** *(either use LinkedList for Stack and ArrayDeque for Queue OR use ArrayDeque for Stack and LinkedList for Queue)*

Stack

The class should own (as an instance variable) an instance of ***ArrayDeque*** *or* ***LinkedList*** *and implement its own operations* ***only*** *with forwarding calls to the operations of the encapsulated container.*
Implement the class ***Stack,*** including the following operations:

- push
  - Takes a parameter and adds its value onto the stack
- pop
  - Removes the item off the top of the stack and ***returns*** *its value*
    - *If the stack is empty, return None*
- get_size
  - Returns the number of items currently on the stack

Queue

*The class should own (as an instance variable) an instance of **ArrayDeque** or **LinkedList** and implement its own operations **only** with forwarding calls to the operations of the encapsulated container.*

Implement the class ***Queue,*** including the following operations:

- add
    - Takes a parameter and adds its value to the back of the queue
- remove
    - Removes the item off the front of the queue and ***returns*** *its value*
        - *If the queue is empty, return None*
- get_size
    - Returns the number of items currently in the queue

Bonus 5% will be given for solutions where *push, pop, add* and *remove* are all implemented with the best time complexities possible.

40% SLL Recursion

- **(10%) get_size(head)**
    - Takes in the head of a list as a parameter
    - Returns the size of the list
- **(15%) reverse_list(head)**
    - Takes in the head of a list as a parameter
    - returns a node, the head of a list that has the same items as the previous list, but in reverse order.
- **(15%) palindrome(head)**
    - Takes in the head of a list as a parameter
    - Returns **True** if the list is a palindrome
        - A list is a palindrome if it is the same reading it forwards and backwards.
            - Example: *abba*, *level* and *radar* are palindromes
            - While *adba* is not a palindrome
        - Since we are using lists instead of strings imagine that every node in the list holds a single character
    - Otherwise returns **False**
    - This can be done with more than one separate recursive calls that may initialize new instances of Node, or move data around. As long as all runs through the list/lists are ***recursive***, and the ***original list*** sent in is not broken in any way, full marks will be given.

Bonus 5% will be given for palindrome solutions that only do **one** recursive iteration through the list.

Points can be deducted for unnecessarily complex code or memory allocation.
Solutions that put the data into a different type of data structure to solve it do not count!