Programming assignment 3: Doubly-linked lists

Bonus 5% for a correct implementation that has no unnecessary repetition of code
Make the class ***DLL*** which uses a doubly-linked list to implement the following operations:

- ***__str__(self) - (5%)***
  - Prints the items in the list with a single space between them
- ***__len__(self) - (5%)***
  - Returns the number of items in the list
- ***insert(value) - (5%)***
  - Inserts an item with that value *in front of* the node at the current position
    - *The new node is now in the current position*
- ***remove() - (5%)***
  - Removes the node at the current position if there is one (otherwise does nothing)
    - *The node behind the removed node is now in the current position*
- ***get_value() - (5%)***
  - Returns the value of the item at the current position in the list *(**None** if not item)*
- ***move_to_next() - (5%)***
  - Moves the current position one item closer to the tail/trailer
    - *Do nothing if at end*
- ***move_to_prev() - (5%)***
  - Moves the current position one item closer to the head/header
    - *Do nothing if at beginning*
- ***move_to_pos(position) - (5%)***
  - Moves the current position to item #*position* in the list
    - *The first actual data item is #0*
    - *Do nothing if position not between beginning and end (including both)*
- ***remove_all(value) - (20%)***
  - Remove all instances of the ***value*** from the list
  - The node at the current position could be removed during this operation
    - *If so, reset the current position to the beginning of the list*
- ***reverse() - (20%)***
  - Reverses the order of items in the list
  - After reversing reset the current position to the beginning of the list
  - **Bonus 5% for a O(1) implementation**
    - *Note that this may be implemented not only in the reverse operation itself, but in many or all of the other operations (read information on next page)*
    - *<u>It is definitely better to have a O(n) implementation that works than trying to get the bonus and ending up with something that doesn't work or breaks other functionality</u>*
- ***sort() - (20%)***
  - Order the items in the list with any method <u>that uses only your DLL structure</u>
    - *No moving everything to another structure, sorting and then moving back!*
  - After sorting reset the current position to the beginning of the list

Implementing the functionality of an ADT

Sometimes, in order to implement the description of an abstract data type, the implementation doesn't have to actually physically do what is described. It's enough to make sure that all output from the functions is correct, given the description of the ADT. It's nobody's business what is actually done in the implementation (but we're always looking for more efficiency).

Example:
The ADT for ***Set*** expects there to not be duplicates (or it doesn't care about duplicates) in the underlying data structure. I can ***add("A")*** five times, and then ask if it ***contains("A")***. It will return **True**. If I then ***remove("A")*** and again ask if it ***contains("A")*** it returns **False**, as ***"A"*** has been removed from the set. It doesn't matter how often I added it.

One might think that there is always only one *instance* of ***"A"*** in the data structure, but it might be better to implement it so that the item is always added, and there is never a check if it's already there. This way the ***add()*** function is **O(1)**, while looking for the item to see if it's already there is **O(n)** (or **O(log n)** if the structure is ordered). So you get a better time complexity on ***add()*** by just adding the duplicates. ***contains()*** will then just find the first item with the correct value and return **True** (or **False** if nothing is found). This means that ***remove()*** will have to go through the entire list and remove every single instance of the value, in order for the value not to be there anymore, but ***remove()*** would be **O(n)** either way (or **O(log n)** if the structure is ordered).

So there are two ways to implement this functionality: make sure there are no duplicates, or let there be duplicates and then just remove all of them later.

Faking it
In some cases an operation can, instead of performing some functionality, simply *pretend* that it did (set a boolean, stating the change) and then deal with the actual implications of the change later, as needed.

In all of these cases it's important that the output from every single operation that the ADT describes is correct, but it's not always the most obvious way that is the most efficient.