Programming Component

Note: For all programming questions you must use Python 3.2.3 or higher. Please follow the Python style guide available from the course website.

Note: Correctness tests will include some marks for efficiency (that is, they will test with large inputs, causing inefficient solutions to time out)

Q1. (20 Marks Correctness + 5 Marks Style)

In this question you must implement a Sequence ADT using a **doubly linked list** (it’s the only way to meet the required time complexities)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Required Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(self)</td>
<td>Creates a new (empty) Sequence</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>getitem</strong>(self,i)</td>
<td>Returns the element at index i. Requires that i is a valid index (ie –len &lt;= i &lt; len)</td>
<td>O(i)</td>
</tr>
<tr>
<td><strong>setitem</strong>(self,i,v)</td>
<td>Assigns the value v to index i. Requires that i is a valid index (ie –len &lt;= i &lt; len)</td>
<td>O(i)</td>
</tr>
<tr>
<td><strong>contains</strong>(self,v)</td>
<td>Returns true if v is a member of self</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>len</strong>(self)</td>
<td>Returns the number of elements in self</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>iter</strong>(self)</td>
<td>Returns an Iterator that traverses self</td>
<td>O(1), and the Iterator’s <strong>next</strong> method must also be O(1)</td>
</tr>
<tr>
<td>add_front(self,v)</td>
<td>Prepends v (prepends means “places in front of”)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add_back(self,v)</td>
<td>Appends v (appends means “places after”)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove_front(self)</td>
<td>Removes the first element</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove_back(self)</td>
<td>Removes the last element</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(self,v)</td>
<td>Removes the first occurrence of v (requires: there is at least one occurrence)</td>
<td>O(n)</td>
</tr>
<tr>
<td>sort(self)</td>
<td>Sorts the Sequence</td>
<td>O(n^2) (or better)</td>
</tr>
</tbody>
</table>
Place your implementation of the Sequence class in `sequence.py`

Note: The magic methods have been documented. The rest is up to you! Please use your own words, don’t copy & paste the descriptions from the table!

**Example interaction**

```python
>>> S = Sequence()
>>> len(S)
0
>>> S.add_front(3)
>>> S.add_back(9)
>>> S[0]
3
>>> S[1]
9
>>> S[-1] = 0
>>> S[1]
0
>>> S.add_back(12)
>>> S.sort()
>>> list(S)
[0, 3, 12]
>>> S.remove(3)
>>> 3 in S
False
```

Note: The public test will use the same operations as the example interaction. Notice that it does not test all methods!!!

**IMPORTANT:** Get `add_back` and `__iter__` working first, since without them we can’t test any of your other methods! Feel free to use and modify my example code from lectures as a starting point!

**Written Component**

Place your answers in A2.pdf

Although you are not required to, we recommend you use LaTeX to create your documents. Resources and links can be found on the course website.

**Q2. (10 marks)**

In Lectures, the quicksort algorithm always selected the first element as the pivot. Suppose that instead it computed the average of the list (assuming it is a list of integers or floats). This is problematic, since the algorithm given in class assumes that the pivot is a member of the list, but in the list `[1, 11]` the average is 6, and that is not a member of the list. To fix the issue, I use the following algorithm.
Q3. (10 Marks)
As shown in class, a singly linked list can run into problems if it shares nodes in common with another linked list.

Suppose you are given two node references, head1 and head2, and told that both linked lists share at least one node in common. You must find the first node that they share in common.

Example:

In this the above lists, the node containing 16 is the first node they have in common.

To simplify your argument, you may assume that the second list is no longer than the first list (meaning, \( \text{len(head1)} \geq \text{len(head2)} \)).

You must justify that your algorithm is correct, and that it meets the time complexity requirements.

a) Give an algorithm that has \( O(nm) \) time complexity (but no better, that’s part b), where \( n \) is the length of the first list (the one starting with head1) and \( m \) is the length of the second list.

b) Give an improved algorithm that has \( O(n+m) \) time complexity.

---

1 This kind of assumption is common when defining algorithms. The usual magic phrase used is “Without loss of generality,” meaning “This assumption does not restrict the problem.”