- This lab will cover review material for upcoming Midterm 2.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

---

1. Draw the state of the linked list object as the following code executes:

```
from DoublyLinkedList import *

L = DoublyLinkedList()
L.add_first(1)
L.add_last(3)
L.add_last(5)
L.add_after(L.first_node(),2)
L.add_before(L.last_node(),4)
L.delete_node(L.last_node())
L.add_first(0)

print(L)
```

What is the output of the code?

2. Draw the state of the deque object as the following code executes:

```
from ArrayDeque import *

D = ArrayDeque()
D.add_first(3)
D.add_last(4)
D.add_first(2)
D.add_last(5)
D.add_first(1)
D.delete_last()
D.add_first(0)
```

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Write a function that takes a positive integer value and returns the prime factorization of the integer as a linked list.

   ```
   def prime_factorization(int):
       '''
       : return-value type: DoublyLinkedList
       '''
   ```

   For example, a call to `prime_factorization(24)` would return a DoublyLinkedList consisting of `[2<-->2<-->2<-->3]`.

2. Implement the *boost queue ADT* again. This time, use a `DoublyLinkedList` instead of a circular array. A *boost queue* is like a queue, but it also supports a special "boost" operation, that moves the element in the back of the queue a few steps forward.

   The operations of the *boost queue ADT* are:

   - `__init__(self)`: creates an empty `BoostQueue` object.
   - `__len__(self)`: returns the number of elements in the queue
   - `is_empty(self)`: returns `True` if and only if the queue is empty
   - `enqueue(self, elem)`: adds `elem` to the back of the queue.
   - `dequeue(self)`: removes and returns the element at the front of the queue. If the queue is empty an exception is raised.
   - `first(self)`: returns the element in the front of the queue without removing it from the queue. If the queue is empty an exception is raised.
   - `boost(self, k)`: moves the element from the back of the queue k steps forward. If the queue is empty an exception is raised. If k is too big (greater or equal to the number of elements in the queue) the last element will become the first.

For example, your implementation should provide the following behavior:

```
>>> boost_q = BoostQueue()
>>> boost_q.enqueue(1)
>>> boost_q.enqueue(2)
>>> boost_q.enqueue(3)
>>> boost_q.enqueue(4)
>>> boost_q.boost(2)
>>> boost_q.dequeue()
1
>>> boost_q.dequeue()
4
>>> boost_q.dequeue()
2
>>> boost_q.dequeue()
3
```

Implementation requirements:

- All queue operations should run in θ(1) worst case, besides the `boost(k)` operation, which should run in θ(k) worst case.
- You should use a `DoublyLinkedList` in your implementation.

3. Give an implementation of a function to flatten a doubly-linked list. For this function, we will work with a *nested doubly linked lists of integers*. That is, each element is an integer or a nested doubly linked list of integers.

   The function is given a nested doubly linked list of integers, `lnk_lst`, and returns a new, flattened, version of it. Here is its prototype:

```
def flattened_linked_lst(lnk_lst):
    '''
    : lnk_lst type: DoublyLinkedList
    : return-value type: DoublyLinkedList
    '''
```

   Write two implementations of this method, one using recursion and non-recursive.

   For example, after implementing `flattened_linked_lst`, you should expect the following behavior when loading your file using `python3 -i flatten_linked_list.py`:

```
>>> lnk_lst1 = DoublyLinkedList()
>>> elem1 = 1
>>> lnk_lst1.add_last(elem1)
>>> elem2 = DoublyLinkedList()
>>> elem2.add_last(2)
>>> elem2.add_last(3)
>>> lnk_lst1.add_last(elem2)
>>> elem3 = 4
>>> lnk_lst1.add_last(elem3)
>>> lnk_lst2 = flatten_linked_lst(lnk_lst1)
>>> lnk_lst1
[1<-->[2<-->3]<-->4]
>>> lnk_lst2
[1<-->2<-->3<-->4]
```

3. Implement a `PlayList` class for an audio player. The playlist will store songs as strings using the title of the song. You should use a circular array in your implementation. You can modify the `ArrayQueue` class to solve this problem. Note that you won't be able to implement the functionality of a `Playlist` by simply using the `ArrayQueue` as a data member. Also note that the index position of a song in the playlist can be calculated from its track number using the front_index position and number of songs in the list.

   Implement the following methods for a Playlist object:

   - `__len__(self)`: returns the number of songs in the playlist
   - `is_empty(self)`: returns `True` if and only if the playlist is empty
   - `add_to_playlist(self, str_title)`: adds the song to the end of the playlist
   - `print_playlist(self)`: returns a representation of the playlist, with the tracks numbered. For example, a call to `print_playlist()` could return:

     ```
     Track 1: Stairway to Heaven
     Track 2: Help
     Track 3: I Will Survive
     Track 4: Lean On Me
     Track 5: Livin' On A Prayer
     ```

   - `play(self, track_no)`: returns the title of the song corresponding to the track number given. Throws an exception if the playlist is empty or there is no song at the specified track.

- `move_up(self, track_no):` moves the song at the specified track up one position in the playlist. Throws an Exception if there is no song at the specified track or the song is already the first item in the playlist.

- `move_down(self, track_no):` moves the song at the specified track back one position in the playlist. Throws an Exception if there is no song at the specified track or the song is already the last item in the playlist

- `remove_song(self, track_no):` removes the song at the specified track from the playlist. Throws an exception if the playlist is empty or there is no song at the specified track. Keep in mind that all index positions in your circular array between the front index and (front index + number of elements -1 % len(self.data)) must contain data.

Determine the runtime of your methods. If the implementation used a `DoublyLinkedList`, how would the runtime of the methods differ?