Homework 2: Stacks and Queues

Your first data structures

Goals for this assignment

- Implement the Stack and Queue Classes
  - These are going to follow the Stack and Queue interfaces provided in the starter code
    - You must abide by the rules of the interface, otherwise your implementation is not “right”
  - Helper functions are good, won’t interfere with the interface but functions defined by the interface are the core of your assignment
- First time using cons, very powerful and useful!
  - Builds “list” and in this case, a **linked list**, we’re not doing the array implementation of Stacks and Queues
- Get you used to thinking of abstractly and efficiently
  - This is the first assignment where it may not be fully clear what it is you’re doing, a lot of this is going to get you to think abstractly about what your code is doing, why it’s doing that, and why this implementation is optimal (efficient)

The Cons Struct

- Can be a bit confusing at first, but practice makes perfect
  - Note: This isn't the cons library, we're not working with that, so no need to look through the documentation for it yet (Until HW4), but you're welcome to do so!
- Cons Struct builds "list"
  - We'll walk through an example so it makes sense
  - When you keep adding elements to this list, it builds a **Linked-List**
- We're going to be doing a bit more with cons so we know **what to get rid of at certain points**

The ListStack Class

Recall: Stacks are LIFO! *"Last in, First Out"*

- If at first you're struggling with visualizing what a stack is doing, you can think of a stack to have the same behavior as a python list
  - Yes! Python list are stacks!
    - .append() -> adds element to the back of a list
    - .pop() -> removes an element from the back of a list
- How do you what element to pop? Use a "head" pointer
  - Keep track of what element was last added to the Stack
- **Your Job:**
  - push(self, element: T) -> NoneC
  - pop(self) -> T ***This just means it's going to return an element***
  - empty?(self) -> bool? ***This just means that you return a boolean, True/False***

Example Expected Behavior:

head = None
Let example_stack = ListStack()

head = 0, next = None
example_stack.put(0)

head = 1, Next = 0
example_stack.put(1)

head = "214", Next = 1, Next Next = 0
example_stack.put("214")

"214" gets removed from Linked-List
head = 1, next = 0
example_stack.pop()

1 gets removed from Linked-List
head = 0
example_stack.pop()

0 gets removed from Linked-List
head = None
example_stack.pop()

.pop(self) -> T

- Removes an element from the Stack list
  - More specifically, it removes the “head” of the list (whatever is at the front)
- **Special Case:** if popping from an empty stack -> error

General Approach:

- Make sure that you’re not popping from an empty stack
- Get the current head
- Reassign the head (so whatever element is after the front)
- Make sure to return the previous head!

The ListQueue Class

Recall: Stacks are FIFO! *“First in, First Out”*

- Python list can also be Queues!
  - Append elements to the back
  - pop() from the 0th index, .pop(0)
- Now you need to use both a Head and Tail pointer!
  - Your head is going to keep track of what element is at the front of the list
    - This is the element that gets pushed back when you enqueue
  - Your tail is going to keep track of what element is at the back
    - This is the element that was last added to the queue
- Both enqueue and dequeue should be O(1) time
- **Your Job:**
  - enqueue(self, element: T) -> NoneC ***This one can be tricky!***
  - def dequeue(self) -> T
  - def empty?(self) -> bool?

Example Expected Behavior:

head = None, tail = None
Let example_queue = ListQueue()

head = 0, tail = 0
example_queue.enqueue(0)

head = 0,1, tail = 1
example_queue.enqueue(1)

head = 0,1,”214”, tail = “214”
example_queue.enqueue(“214”)

0 gets removed from Linked-List
head = 1,”214”, tail = “214”
example_queue.dequeue()

1 gets removed from Linked-List
head = “214”, tail = “214”
example_queue.dequeue()

“214” gets removed from Linked-List
head = None, tail = None
example_queue.dequeue()

.enqueue(self, element: T) -> NoneC

- Need to keep track of the ends of the list!
  - Need a hail and a tail
  - Make sure you assign the head correctly!

General Approach:

- If this is the first element that you're adding to the Queue, need to assign **both the head to this element!**
- Otherwise, if this is not first enqueued element, need to add this element to the list and assign the tail to the new element

.dequeue(self) -> T

- Now we use the ends of the list
- **Special cases:**
  - if the queue is empty -> error
  - **Make sure what when you dequeue the last element, that both the head and tail are None!**

General Approach:

- Make sure we can actually dequeue (queue is not empty)
- Get the current head
- Assign to the head to whatever element is next
- Return previous heads data field

Consider This:

When you dequeue the last element in the queue, both the head and tail should be None, make sure your implementation does this!

Managing Playlist

- This is just meant to show you what queues can be used for!
  - “Real life” use cases are going to be used for future homework assignments
- ***These are supposed to be extremely straightforward, don’t overthink them!!***
- Might wanna comment out the Queue interface for this one
  - Ran into issues with this last quarter
- To make a RingBuffer
  - Let x = RingBuffer(y) where y is the size of the RingBuffer

That’s all for HW2!