Yizhou Qiu

Clay Moughon

CSCI 241

Instructor: James Deverick

Project 3 Write-up

1. a description of any issues your team encountered during the implementation:

When we build up Deque, first, we want to insert\_element\_at(val,0) method to implement push\_front method. However, after we tested it, it seemed that our push\_front method did not work. Then we realize that the linked list class can only insert things when the list is not empty. So, we divide the push\_front method into two situation: one is when the deque is empty and another is when the deque already has elements inside. When it’s empty, we use append\_element method, and after it already have an element, we can use insert\_element\_at(val, 0).

Another problem we encounter is when we are writing the code for Stack, we could not print multiple stacks at the same time. After examination, we realized that when it becuase we were in Python 2 instead of Python 3. After we changed to Python 3, everything worked properly.

When we were working on Deque’s \_\_str\_\_method, first, we want to write a string method which including the full description. Then we realize that since the linked list already has a string method, we can use str() directly and don’t need to write another on for Deque.

We had difficulty in writing the Towers of Hanoi code. Recursively, it becomes somewhat difficult to think of moving objects from stack to stack, especially because instead of naming specific stacks and moving objects in them from location to location, they’re only considered in the context of what purpose they serve at that point. Therefore, linguistically, having to explain that the source stack has now become auxiliary or the destination is now the source stack was a barrier to understanding how the algorithm worked, especially because we had considered each stack as A, B, and C before.

2. a justification for each of your test cases, and an explanation of why those test cases are complete:

For Deque class, we have test code to test:

-If we can push element at the back if the element is empty

-if push front method works and if it can only push the element to the top of the deque

-if the len() method works and return the right length of the deque

-if pop\_front() and pop\_back() work and if it only pop the top element in the deque

-if peek \_front() and peek() back work and if it won’t change anything in the deque

-if pop\_front and pop\_back() don’t pop anything from the deque and return None when the deque is empty

\_If peek\_front and peek\_back will return None and won’t do and changes to the deque when the deque is empty.

-If the string function works and allow us to print the deque

For the stack, our code tests a few things, many of which are similar to previous tests.

-does the push function work

-does pushing more values maintain the FIFO philosophy?

-does the len() function return how many values are in the stack?

-does our pop function return the top value on the stack?

-does our peek function not modify the stack?

-do peek and pop not do anything to an empty stack and return None?

-does our print() work on a stack object?

For the queue, our test codes include testing things listed below:

-does the enqueue method works and only enqueue element at the back

-does the dequeue method works and only dequeue elements from the top

-does the length method always return the right length of the queue

-does the string method work and allow us to print the queue

3. a brief explanation of how you could use an array as the storage medium instead of a Linked List or Deque for each of the three structures:

We could have easily used an array to simulate each data type - just make sure that the ways you insert new values/objects into the array follow the stack/queue philosophy of LIFO/FIFO. There are a few problems with that approach, however. In a linked list, there is no determined size for how large a list of linked nodes will be. However, in an array, you must establish array size prior to working on it. There are append functions in an array, but those are very taxing computationally. Therefore, you must assume, almost beyond all doubt, that the amount of objects placed into your data structure will not be larger than the size of the array you establish to mimic that data structure. There are ways around this, however. In queues, values/objects are likely to be removed from the queue eventually after they are dealt with. This means that there could, theoretically, be infinite space to insert new “jobs” into the queue. To do this, we would use modular arithmetic to create a circular array such that whenever the front of the queue moves indexes, it does not reach a bound of indexes, and rather returns to what would be front/back of the array.