Results

Shang Xiao — 1st Attempt

100%

10
Out of 10 points

282:42:08
Time for this attempt

This assessment has unlimited attempts. Take Now

Attempt History

Results	Points	Score	(Highest score is kept)
Attempt 1	10 of 10	100%	(Highest score)
Attempt 2	0 of 10	0%	

Your Answers:

1 2 / 2 points

Describe 2 differences between Abstract Data Types and Data Structures.

Per instruction on the module video, an ADT is:

description of an organized way to store data

tells what kinds of things can be stored

tells what operations are provided to store, manipulate, and extract

Unlike data structures, ADTs do not tell you HOW to implement certain tasks, rather they tell you what kind of things and opertions can be completed. ADTs are not implementation and can be immediately used. Think ADTs as outlines for data structures can be the best way to distinguish these two.

Another difference between these two is ADTs do not care about actual data that we will be using in it, the data type can rather be strings, objects, or something else, we only care about these data types when we get to data structures.

Correct

2 2 / 2 points

What are two expected functions for Stack and what do they do?

push() that pushes things/new elements onto the top of the stack, as more and more new elements being pushed onto the stack, the earliest element that was pushed onto the stack go the the bottom of the stack and the latest element that was pushed onto the stack go the top of the stack. For example, if we push 4, 5, 6, 7 the stack would look like this:

7

6

5

4

pop() that pulls the top/the last pushed element off the stack. For example, if we pop() the above stack, the pop() function would return 7.

now if we pop() again, the pop function would return 6.

now if we dump(), the dump() function would return:

5

4

We need to keep note that, as per our class discussion, although the original version pop function PULLS OFF 7 and 6 simultaneously, it does not REMOVE 7 and 6, because if we were to use a debugger, we could see that 7 and 6 are still there is just that the pop() function pulled off 7 and 6 from the stack. In fact, 7 and 6 will remain there 'hidden' until we overwrite them with something else. We could, however; modify our code so that one line holds the data that is being poped and another line sets self.data[self.end] = 0 and then return the poped data.

Other functions related to Stack are:

peek(): take a look at what is currently at the top of the stack without taking it off

count(): sum up how many things are on the stack

dump(): print out all current elements that are on the stack

Correct

3 2 / 2 points

What are two expected functions of a Queue and what do they do?

enqueue(), similar to push(), enters an element into the line

dequeue(), similar to pop(), gets the earliest enqueued element out of the line

Other functions related to Queue are:

is_full(): check if the queue has space

is_empty(): check if the queue has something

peek(): take a look at what is currently at the front of the line

count(): count the number of elements in the line

Correct

4 1 / 1 point

What is the order in which data is accessed in a Stack?

Last in, first out (FIFO), often can be seen as a pile/stack of paper/plates in offices/restaurants.

Correct

5 1 / 1 point

What is the order in which data is accessed in a Queue?

First in, first out (LIFO), often can be seen as a line-up at bus stops/checkouts in superstores.

Correct

6 2 / 2 points

What is a potential problem with respect to memory usage in a Queue and how could you fix it?

In our code we set my_queue = Queue(10), that is, no matter how many elements you dequeue after you enqueue, you can never have the 11th element enqueue into the queue. In short, quoting from the class slide: "empty space at front of queue is not reclaimed", or quoting from professor John Park :"we have not ran out of the room, we just hit the end of the list".

This is because although the value referred to has been dequeued, the value itself, which is empty, still takes the place within the queue. To fix this, we introduce two method, one is shifted queue and another is circular buffers.

Shifted queue: move everything forward like an actual queue. We can shift things down when we dequeue elements. The disadvantage of this approach is that we have to down shift everything once, so when we have a significant large amout of elements in our queue, this shifting method can gets very annoything. A code implementation for this shifting method is shown below:

Before shifting, the dequeue function:

```
def dequeue(self):

    if self.start == self.end:
        print("Empty!")
        return
    item = self.data[self.start]
    self.data[self.start] = "<EMPTY>"
    self.start += 1
    return item
```

After shifting, the dequeue function:

```
def dequeue(self):

    if self.start == self.end:
        print("Empty!")
        return
    item = self.data[0]
    self.end -= 1
    for i in range(0, self.end):
        self.data[i] = self.data[i + 1]
    return item
```

Circular buffer: effectively use up the empty space back at the low end when we reach the end of the high end (wrap around the high end). Quoting from professor John Park: "we want to be able to keep writing around a loop, possibly many times, as long as there's spare space to be recycled". We achieve this type of logic by using the modulo division operator, which has the exact same behaviour we needed here. An implementation of this logic is below:

len = 10

end = 8

end = (end +1) % len

end returns 9

end = (end+1) % len

now end returns 0

This is because no matter how big the number is, whichever number % 10 will always return the last digit of that number.

The shifting method enqueue() function:

```
def enqueue(self, item):

    if self.end >= len(self.data):
        print("Full!")
        return
    self.data[self.end] = item
    self.end += 1
```

The circular buffer method enqueue() function:

```
def enqueue(self, item):

    if (self.end + 1) % len(self.data) == self.start:
        print("Full!")
        return
    self.data[self.end] = item
    self.end = (self.end + 1) % len(self.data)
```

The shifting method dequeue() function:

```
def dequeue(self):

    if self.start == self.end:
        print("Empty!")
        return
    item = self.data[0]
    self.end -= 1
    for i in range(0, self.end):
        self.data[i] = self.data[i + 1]
    return item
```

The circular buffer method dequeue() function:

```
def dequeue(self):

    if self.start == self.end:
        print("Empty!")
        return
    item = self.data[self.start]
    self.data[self.start] = "<EMPTY>"
    self.start = (self.start + 1) % len(self.data)
    return item
```

Correct