PoP Report 8g

[Niels Dreesen, Mustafa Hekmat Al Abdelamir, Jakup Højgaard Lützen]

December 3, 2022

Introduction and problem formulation

This report is about imperative programming. Our task was to create a cyclic queue with fixed storage space allocated for it. The challenge we were faced with was, to implement the variables first and last in order to point at the position containing an element in the queue. Further, we had to write functions for adding a queue space, adding a value in that position and lastly dequeuing an element.
The next challenge was to make this program cyclical, in order to run through all our queue positions and make it start over after it dequeued the last position of our queue with a fixed amount of spaces.

Problem analysis and design

First we need to think about how our program would be able to "simulate" a circular memory array. For that we develop the function iterate, which iterates the index of the array by 1 except at the last element where it is reset. Other than that we need to check if the queue is full, which is a bit trickier than usual, because it has to be done with the pointers, rather than checking the actually data in the memory array. The rest of the program should be able to work similarly to a vanilla mutable queue.

Program description (a)

In this section we describe the different functions of our cyclicqueue.fs file.

create

The create function creates an empty array with a length of n. This presents the first problem. Being that this function should return unit, the array needs to be defined outside the array as a mutable array, and then redefined inside the function with the appropriate length.

iterate

The iterate function checks for two cases. One is where the current position is at the end of the array, where the new position is then the start of the array, index 0. Otherwise, the new position is simply the old one, iterated by one.

```
Some x when x = (queue.Length - 1) -> Some 0
      | Some (x: int) -> Some (x+1)
      | None -> None
```

isfull

The isfull function calculates the index after the last-pointer, and checks whether it is the same position as the first-pointer position in the queue. The result is true, when it is the same position since when the last-pointer is right behind the first-pointer, the queue is full.

enqueue

The enqueue function iterates the last-pointer and overrites the index of the last-pointer with the new elment to be queued. If the queue has no position, it means that the queue is empty, in which case both the first and last pointer point are initialized at the same position.
If the queue is full, then the enqueue function returns false, and does not modify the queue.

dequeue

This function de-queues an element if the queue is not empty. If there is only one element left, it reinitializes the pointers as None. Once an element has been de-queued it is returned.

isEmpty

This function simply evaluates the first pointer. When it is none the queue is empty, otherwise this assertion is false.

length

This function simply subtracts the pointer indexes to find the length of the list. If the last-pointer is behind the first-pointer (because the last-pointer has looped around) it is the length of the queue subtracted by the index of the first-pointer which is added to the index of the last pointer. This means we leave out the positions in between them to count the amount of elements in the queue.

toString

This function extracts the elements in between the pointers, and converts each element to a string with a map function, and then adds all the strings together with a fold function.

Testing and experiments (b)

Our testing is implemented in the cyclicQueueApp. We initialize a queue of size 10. First we test our enqueue function by enqueuing elements from space 1-11. We test at once, for whether the enqueing is working correctly and whether it is responding correctly to trying to enqueu more elements than there is space in the queue (Since the queue only has 10 spaces while we are enquing 11).
Afterwards we dequeue the elements from space 1-5 to see if that works as well. After that we enqueue 1-5 again. This will repopulate our array to be exactly the same as before, except where the pointers are pointing differently. This test will show whether our module is correctly handling the order of the array elements according to the first- and last- pointer. We read the order by dequing the entire list.
We then repopulate the queue to test whehter our reinitialization is working correctly. And lastly we test the toString function.

The program will print the error count and terminates by returning an ineteger representing the error count. So if the program returns 0, it means that all the unit tests where sucessfull

Discussion and conclusion

This assignment presented some challenges with how to build a mutable and cyclic queue with a fixed amount of spaces. But overall the task was similar to the previous expereince we have amassed related to queues. In practical terms, the only difference is using the pointers to read the array rather than using the built in indexes.

(c)

We believe we have satisfied the program specifications. We have also handled exceptions that might happen while trying to enqueue, dequeue etc. when it is not possible to do so. We have not specifically handled exceptions where the user input data is bad. For example when a user tries to make a list with a negative amount of spaces, or add an invalid element to the queue.

(d)

Our final implementation is a very intuitive and was relatively fast to implement because we were using mutable variables. But on the other hand, because our queue exists globally, it can be difficult to know the exact sequence of what happens when we have an error. Also any changes made to the queue have to be relevant to the entire program. If a specific function needs to work with the queue in a destructive way, it needs to first clone it locally.