COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 10: Queues

Lecturer: Dr. Andrew Hines *Scribes: Colin Beagan, Abhishek Sinha*

10.1 Outline

This session introduced the Queue data structure. We reviewed the concept of a Queue, defined it based on a few simple rules and gave a number of real world examples. We then looked at the Queue as an abstract data type (ADT) and the two fundamental methods it supports, enqueue() and dequeue(). Finally we reviewed two implementations of a Queue ADT, one based on arrays (lecture 7) and the other linked lists (lecture 9). This involved an analysis of the relative performance of both implementations, drawing on lecture 3 (Run time analysis/Big(O).

10.2 Queue Concept/Definition

Using the supermarket analogy we can evaluate the basic operations of a Queue data structure;

- You queue up at a given checkout aisle
- You join the back of a queue (tail)
- You move through the queue and get to the front of the queue (head)
- You can occupy both the front and back of the queue at the same time
- There is only one entry point and one exit point
- You cannot leave for a shorter queue

A queue follows the "First in First out principle" (FIFO). The longest member in the queue is the only one allowed to leave. Joining of a queue is called enqueuing. Leaving a queue is called dequeuing.

10.2.1 Real World Examples

Some examples include;

- Queue of cars on single lane road
- Queue of planes on an airport runway
- Queue of people at an airport security check
- Queue of packets in data communication

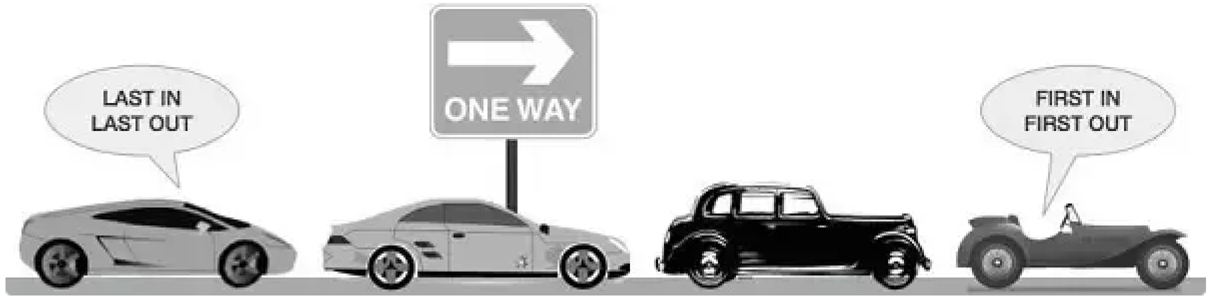

Figure 10.1 Traffic representation of a queue [1]

10.3 Queue as an abstract data type

Fundamental methods;

- enqueue() - Insert an object at the rear of queue
- dequeue() - Remove an object from the head of queue and return it

Support methods;

- size() - Return the number of objects in the queue
- `is_empty()` - Return boolean true if the queue is empty
- front() - Return an object at the head of the queue, but do not remove from queue

A queue data structure can be implemented using one of two basic data structures, linked lists and arrays. We will explore both below.

10.4 Implementation - Linked list

Drawing on our analysis of linked lists from lecture 9 we can describe why a linked list implementation may be a good choice. Linked lists can accommodate any number of elements. They are dynamic, where as arrays are static. Queues need to be able to handle a varying number of elements. Also queues by nature do not access elements outside of the head or tail. For this purpose we do not need an arrays fast element access. For linked lists all of the methods above in section 10.3 have runtime of $\mathcal{O}(1)$.

Enqueue: Consider the queue example below. Currently Denise is pointing to "null". Eric wants join the queue. The enqueue() method is called. Denise now points to Eric and Eric points to null. The "rear" is moved from Denise to Eric. Nothing is returned.

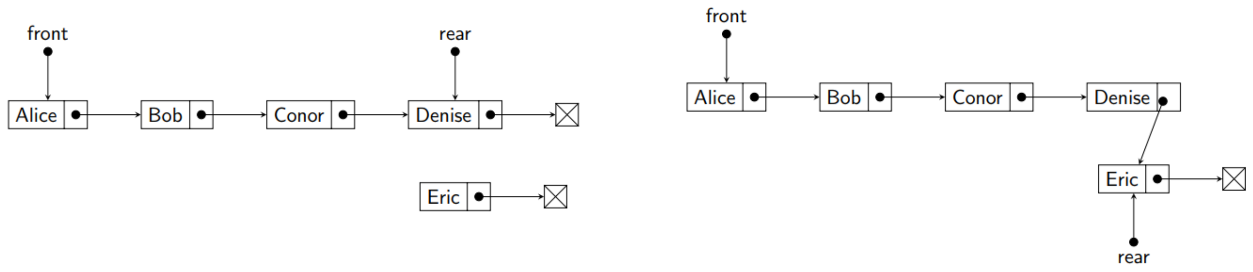

Figure 10.2 Adding a member to a queue

If there are no members in the queue then the "front" and "rear" will be pointing to null. Consider the example of adding Alice to the empty queue. Here we point both the "head" and "tail" to Alice and set Alice to be the queue.

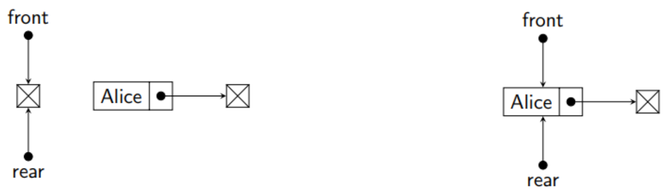

Figure 10.3 Adding a member to a empty queue

Dequeue: Consider the original queue from the first example. Alice is now finished in the queue. The dequeue() method is called. The "head" pointer is moved from Alice to Bob. Alice then detaches from the queue and points to null. The method returns Alice.

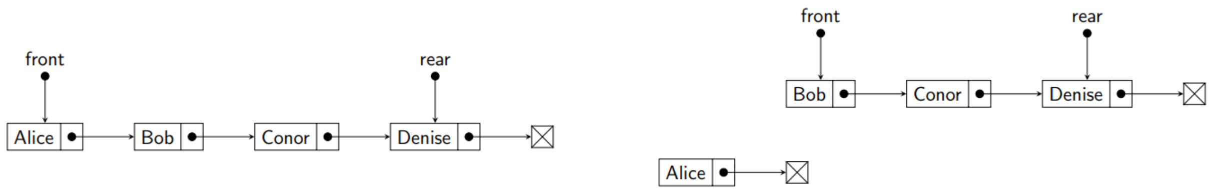

Figure 10.4 Removing member from a queue

10.5 Implementation - Circular Array

Drawing on our analysis of arrays from lecture 7 we can describe how it is possible to implement queues using a traditional array. Here an arrays fast access, $\mathcal{O}(1)$, is not useful. Each time we make a change to the queue, calling the dequeue() or enqueue() methods, we need to modify the array size and re-index it. This results in a runtime complexity of $\mathcal{O}(n)$. It is computationally expensive.

An alternative to get around re-indexing is to use a circular based array, where the end of the array points to the beginning. See example below. Here we set the array to a fixed size i.e. a **capacity**, that we are sure will be enough for the task. We then monitor the "head" and "tail" pointers as they move around the circle. The relative positions of the "head" and "tail" define the current **size** of the array. For a circular array base queue all of the methods above in section 10.3 have runtime of $\mathcal{O}(1)$. See figure 10.4 below;

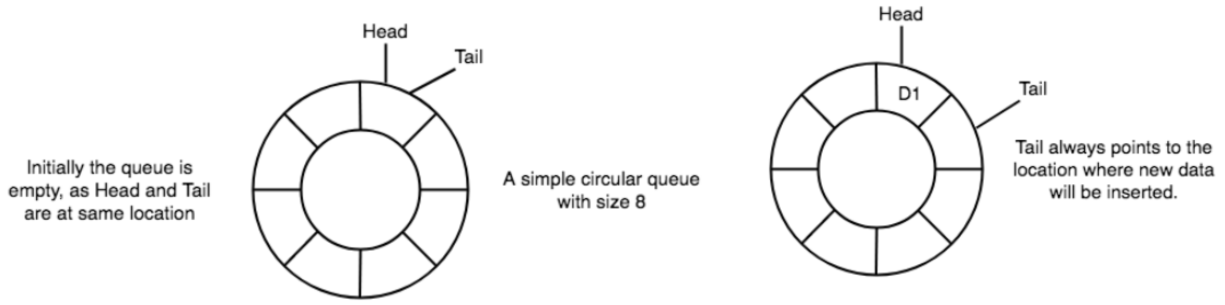
Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8
Head
D1
Tail
Tail always points to the location where new data will be inserted.

Figure 10.4 Add an element to the queue

The "head" will remain static until an element is dequeued. In our representation the "tail" will always point to the next available free space and move clockwise as elements are added. Once the array is full the "tail" will have circled back to the "head" and will be pointing to an existing element i.e. there is no free space left **(size == capacity)**. In the example below when an element is dequeued the "head" will move clockwise to the next element in line. The "tail" will now point to a free space **(size < capacity)**.

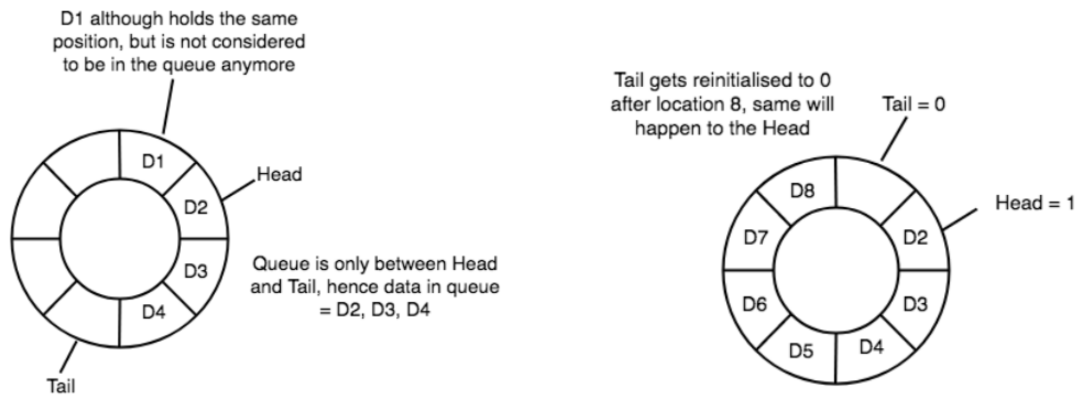
D1 although holds the same position, but is not considered to be in the queue anymore
D1
D2
D3
D4
Head
Tail
Queue is only between Head and Tail, hence data in queue = D2, D3, D4
Tail gets reinitialised to 0 after location 8, same will happen to the Head
Tail = 0
Head = 1
D8
D7
D2
D6
D3
D5
D4

Figure 10.5 Removing member from queue

Note:
1. The bounds of the circular array index index is controlled by: Index = (Index + 1) % **capacity**. This

ensures the index count is never greater than the number of spaces as the ”head” and ”tail” loops around.
2. If a dequeue operation is performed on an empty queue the queue will remain unchanged.
3. If an enqueue operation is performed on an full queue no an extra element will be added.

10.6 Circular Array Example

Below is a walk through of the enqueue() and dequeue() operations on a circular array. Observe the movement of the ”head” and ”tail” pointers during each operation.

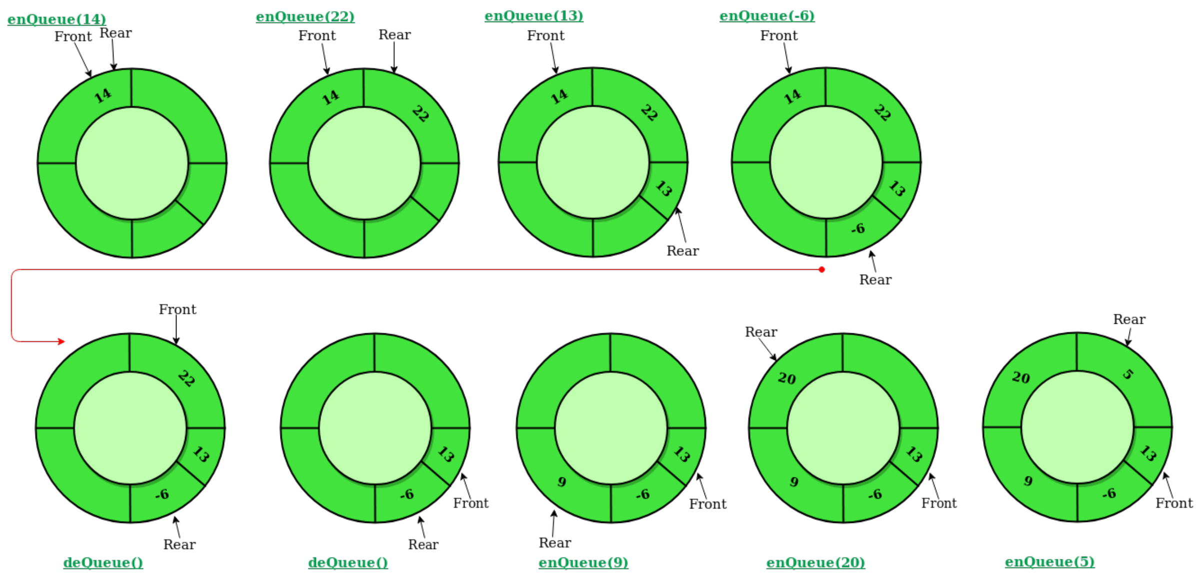

Figure 10.6 Example of circular based array in operation [2]

10.7 Past exam questions

See below past exam questions that you should now be able to answer;

- What is a queue? List the operations commonly associated with a queue (2015/16, 2016/17 5 marks).
- Describe the output of any series of queue operations on a single, initially empty queue (similar to figure 10.6) (2015/16, 2016/17 5 marks).
- Describe how to implement a queue using a circular array/Linked list. Indicate, in words or diagrams as appropriate, how operations enqueue() and dequeue() are implemented. All queue operations should run in O(1) time (2015/16, 2016/17 10 marks).
- Describe briefly the “circular-array” representation of the ADT queue that we studied in the module and indicate, in words or diagrams as appropriate, how operations enqueue() and dequeue() are implemented ((2018/17 10 marks).

10.8 Bibliography

[1] https://www.sitesbay.com/data-structure/c-queue
[2] https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/