Lecture 10: Queues

Lecturer: Dr. Madhusanka Liyanage *Scribe: Rebecca Quinn, Peter Murphy*

1 Overview

In this document, we will explore the topic of Queue ADT. We will begin with an introduction of the queue as a concept and discuss the ways in which queues operate. We will also investigate coded examples of queues along with some asymptotic analysis, to illustrate the inner workings of the structure and to ascertain the most efficient implementation of a queue.

2 Queues - Introduction

A queue is a type of data structure known as an Abstract Data Type. Abstract Data Types, or ADTs, refer to sets of objects whose behaviour is defined by a set of value and a set of operations. An important feature of ADTs is their separation from real-world implementation. This allows ADTs such as queues to be discussed independent of any constraints, in order to be incorporated into different programming languages with ease.

Queues are like lists of objects, but with specific rules attached to the ways in which objects are added and removed from the data structure. Queues are designed in such a way that the least recently added item is dealt with first. Usually, insertion of a new element can only occur at one end of the queue, also known as the rear or tail, and removal of an element occurs only at the other end, known as the front or head of the queue. These operations can only be carried out one at a time, meaning that only one element may be inserted or removed in one operation. Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed. This means that queues abide by the rule of first-in-first-out or FIFO, which, like a queue of people in real life, means that whatever element is entered into the queue first, must also the first to be removed.

3 Operations

Below is a summary of the main operations associated with queues.

3.1 Key Operations

- enqueue(x), adds x to the back of the queue (returns None)
- dequeue(), returns and removes the front element of the queue (returns obj)

3.2 Support Operations

- size(), returns the size of the queue (returns int)
- empty(), returns if the size is 0 (returns boolean)
- full(), returns if the size has reached it's maximum value (returns boolean)
- front(), returns the front element of the queue (returns obj)

4 Implementation

A queue can be implemented using multiple data structures, e.g. array, linked list
The O(1) access to any element in an array is not needed for a queue, and the array would need to be modified with every enqueue and dequeue, so a linked list is better suited for queues, as this has O(1) access to the first and last elements, which is all a queue needs.
Every element in a linked list is a node with a value and a pointer to the next element in the list, e.g. element1.value = 5, element1.next = element2.
A queue has at least two variables;

- head - points to the first node.
- tail - points to the last node.

4.1 Pseudocode

For a queue of size n,

```
enqueue(x):
        if the queue is full;
                return 'Error: Queue is full.';
        else;
                if the queue is empty;
                        point head and tail to x
                else;
                        point tail to x
dequeue():
        if the queue is not empty;
                store value of first node in temp variable
                point head to second node
                return temp variable
        else;
                return 'Error: Queue is empty.'
size():
        return n
```

```
front():
        if the queue is not empty;
                return head.value
empty():
        if n == 0;
                return True
        else;
                return False
full():
        if n is less than the max size of the queue;
                return False
        else;
                return True
```

5 Asymptotic Analysis of Operations

	Linked List	Array
enqueue()	O(1)	O(1)
dequeue()	O(1)	O(n)
size()	O(1) or O(n)	O(1)
front()	O(1)	O(1)
empty()	O(1)	O(1)
full()	O(1)	O(1)

5.1 Comparisons

The main advantage of using array-like structures in any context, is the ability to access elements by their index. But the way a queue is required to function is that items can only be access from the front of the queue, therefore rendering this feature useless. The time complexity of deleting an item from a queue (dequeue), when implemented as an array is O(n) and is the major disadvantage if implementing a queue in this way. Since the item must be deleted from the front of the queue (item at index of 0), all items left in the queue will now have incorrect indexes. Each item must be individually moved up one space and have their index reassigned in order to form a comprehensive array filled with items with updated, accurate indexes.

The dequeue operation in a linked list implementation has a time complexity of O(1), due to the structure of a linked list. The pointer system of a linked list keeps track of the front of the queue with a head pointer. This head pointer is kept up to date whenever the first node is added, or an item is dequeued, and will always reflect the element which is currently first in line in the queue. The dequeue operation access the stored head pointer, redirect it to point at the next item in the

queue and then remove the front item. This eliminates the need to reorder and move all other items to accommodate an item deletion, making linked lists a more efficient method of implementation.

The size operation returns the logical size of the queue, meaning the number of nodes connected together to form one instance of a queue. The time complexity is O(1) in both implementation, if the size of the queue is stored as a variable inside the queue class, and can be accessed at any time. If this data is not stored, the time complexity jumps to O(n) for the linked list implementation, because each node would need to be traversed from one end of the queue to the other, in order to count them all and then return the final count.

6 Double-Ended Queues

Double-ended queues, abbreviated to deque, are a more generalised version of the queue structure, whereby the insertion and deletion of items can be done at both the front and the back of the queue. This means that the enqueue and dequeue operations of the regular queue will not be sufficient in this instance. Separate operations need to be created for adding at the front and at the back of the queue. Double-ended queues can be incorporated inside other data structures, or used to implement a priority queue, where the additions or removal of elements from the queue is not always as straight forward of a process as adding to the rear of the queue. Similar to the regular queue, deques can be implemented with a dynamic/circular array or with a doubly linked list.

6.1 Circular Array Implementation

As discussed above, queue as an array is not the most efficient option, due the re-indexing that must occur when items are removed from a single-ended queue. With arrays, enqueueing and dequeuing operations can either take up too much time or too much space. We can attempt to reduce the amount of space used, by implementing a circular array queue. Circular array queues are also useful particularly when know the size of the queue in advance. You can store pointer variables that point to the item at the front and at the back of the queue to aid this implementation. Upon initialisation of this queue, the front and rear pointers can be set to -1, to illustrate the lack of items. These will be updated as enqueue and dequeue operations occur, where rear will always point to the newly added item, and front will point at the next item to be dequeued.

6.1.1 Key Operations

The addition of the first element will trigger both pointers will point to the same place, the newly added item, at index 0. This is due to the circular structure which loops back on itself, unlike the linear queue discussed above. However, now that we have a circular structure, we need to create a different strategy to increment the rear pointer, when we add a new item after an entire rotation. For updating the rear pointer after an enqueue operation in a clockwise circular array, we can use the formula **(rear + 1) % length**. This will return the index of the next space in the array. When

it returns the index of the front pointer, we know the array is full. This allows us to enqueue items easily, while eliminating the issue of pointing to an index outside the bounds of the limited size.

For the dequeue operation in a circular array, the same logic can be applied for working with the limited indexing. We can just substitute the front pointer for the rear pointer, thereby giving the formula **(front + 1) % length**. This way we can easily update the font pointer to rotate around the circle as items are dequeued behind.

6.1.2 Other Operations

The support operations for a circular array queue will be similar to those of the regular queue (full(), empty(), front()), with some slight modifications to their internal code. But one possible addition to these support operations would be a resize method. The way storage allocation works for dynamic arrays in python for example, is that when the allotted memory is reached, python assigns another block of memory addresses to that data structure to facilitate newer items being added. We can emulate this behaviour in a resize method. We can create a newer, larger array (double the size of the old), add all the elements from the original array into it and set the front and rear pointers to 0 and the length of the old array - 1, respectively.

6.1.3 Asymptotic Analysis of Operations

- enqueue(x) - O(n)
- dequeue() - O(1)
- size() - O(1)
- front() - O(1)
- empty() - O(1)
- full() - O(1)
- resize() - O(n)

7 Real World Examples of Queues

Here are examples are queues being used in real life:

- printer
- traffic
- water dispenser (loosely)
- process scheduling in an operating system

8 Implementation - Linked List

Here there is an example of a basic queue and its key operations, as a linked list.

```
class Node():
    def __init__(self, data, next_node = None):
        self.data = data
        self.next = next_node

class MyQueue():
    __head = None
    __tail = None
    __size = 0

    def __init__(self, max_size = 0):
        self.__head = None
        self.__tail = None
        self.__size = 0

    def queue_enqueue(self, element):
        newest = Node(element)
        # If queue is empty, add node and set head pointer
        if self.__size == 0:
            self.__head = newest
        else:
            self.__tail.next = newest
        # Set tail pointer and update size
        self.__tail = newest
        self.__size += 1

    def queue_dequeue(self):
        if self.__size == 0:
            raise Exception("Cannot remove element from empty queue.")
        else:
            # Move head pointer to next element and remove front element
            temp = self.__head
            self.__head = self.__head.next
            temp.next = None
            self.__size -= 1
            return temp.data

    def to_list(self):
        if self.__size == 0:
            return "There are no elements in this queue."
        else:
            current = self.__head
            elements = []
            while current:
                elements.append(current.data)
                current = current.next
            return elements
```