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

Lecture 10: Queues and Circular Arrays

Lecturer: Dr Andrew Hines *Scribes: Alan Young, Nimisha D.B. Raju, Sean Keyes*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

10.1 Introduction

10.1.1 Recap

The lecture began with a visual demonstration of the fixed size data structure and it's potential limitations, and how joining the ends to form a circular data structure could prove more efficient.

We also began with a recap of the array and linked list data types that we covered in previous lectures; array is fast to search but removing or adding elements is more arduous.

In a queue you can only access the front and back and can only remove from one direction.

10.1.2 Queue concept

Queues are a way to impose arbitrary order on disordered information. Queues follow specific rules when it comes to adding and removing elements from the queue. Queues work on the basis of a first-come-first-served or First-in-first-out/ First-on-first-off methodology.

10.1.3 Real World Queue Examples

- Bank payments; payments happen in the order that they are received by the bank
- Ferries; roll on roll off ferry
- Stack is an example of the opposite of a queue

Consider the following real life example. An online ticket booking system were the payment where the transactions have to be processed in the correct order (First in first out). For a more in depth view check out this link! Click here!

10.2 Abstract Data Type (ADT)

10.2.1 Fundamental methods

If we consider a queue in a shopping center. It can be of variable length, and customers are only dealt with as they reach the front of the queue. Customers can only be added to the queue from the back. This follows a "first in, first out" (FIFO) methodology.

When considering a queue as an ADT, the same principles apply and make up the fundamental methods of the Queue Data Type of Class.

10.2.2 Method: Enqueue

This method takes an object inputted and adds it to the rear of the queue. This method does not produce an output, only alters the elements within the queue. An error should occur if the queue is full

Algorithm 1: Enqueue

Input : An object a
Output: None
1 $Object \leftarrow$ the object a is added to the queue
2 return $None$

10.2.3 Method: Dequeue

This method takes the first element in the queue, and removes it from the queue. It has no input, as it always removes the first element in the queue. The output is the object itself. If a queue is empty an error should occur.

Algorithm 2: Dequeue

Input : None
Output: Content of the first element of the queue b
1 $Object \leftarrow$ the object b is removed from the queue
2 return b

10.3 Support methods

These are not key to the functionality but are useful quality of life features for the ADT.

10.3.1 Size

Takes no input and returns the number of elements are in the queue.

Algorithm 3: size

Input : None
Output: The integer length of the queue s
1 $s \leftarrow$ iterate over the queue, count the elements and store it in a variable
2 return $length(s)$

10.3.2 Is Empty

Takes no input, but returns a Boolean that tells us if the queue has any elements in it. This is useful for error handling and dequeueing.

Algorithm 4: is_empty

Input : None
Output: Boolean True or False
1 $Check \leftarrow$ Does the queue contain a first element
2 return $Boolean$

10.3.3 Front

Takes no input. Inspects the front element. Maybe you want to see if you want to remove this element or not. Consider a situation where you remove the first element in the queue, but find it was not the element you were looking for. The element cannot be placed back at the front position and must be added to the back of the queue.

Algorithm 5: front

Input : None
Output: Content of the first element of the queue b

10.4 Linked list implementation

The linked list is a good way of implementing the concept of a queue as an abstract data type. Like a queue a linked list has pointers to the first and last elements. A linked list is inefficient when it comes to accessing the middle elements; the queue does not require access to any element that is not the first or last, making this pairing a good fit.

10.4.1 Enqueueing step by step

1) We take the queue, and we take our new element as an input
2) We find the tail of the queue and change the pointer in the rear element to point to the new element.
3) We change the pointer of the new rear element to point to null
4) We change the tail pointer to the new rearmost element

10.4.2 SPECIAL CASE: The empty queue

When adding to an empty queue, both head and tail point to null. In this case we need both head and tail pointers at the new and only element in the queue. Then we direct the pointer of this solitary element to null.

10.4.3 Dequeueing step by step

1) We find the head of the queue
2) We move the head pointer to reference the next element in the queue.
3) We move the pointer of the former first element of the queue to reference null
4) The dequeue method will then return the element for use
5) If not in use the element will be removed from memory by automatic process in some languages such as java (garbage collector)

Here is an example of the linked list algorithm from the lecture notes, coded in python.

Linked List Implementation

```python
class Node:
    def __init__(self, data=None, nextNode=None):
        self.data = data
        self.setNext(nextNode)

    def getData(self):
        return self.data

    def getNext(self):
        return self.nextNode

    def setNext(self, newNext):
        self.nextNode = newNext
```

```python
class LinkedList:
    def __init__(self, head=None, tail=None):
        self.head = head
        self.tail = tail

    def enqueue(self, data):
        newNode = Node(data)
        if self.is_empty() == False:
            self.tail.setNext(newNode)
        else:
            self.head = newNode
        self.tail = newNode

    def dequeue(self):
        if self.is_empty() == False:
            dequeuedNode = self.head
            self.head = self.head.getNext()
            dequeuedNode.setNext(None)
            return dequeuedNode
        else:
            print("Error: empty queue")

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count += 1
            current = current.getNext()
        return count

    def is_empty(self):
        if self.size() == 0:
            return True
        else:
            return False

    def front(self):
        return self.head

    def printList(self):
        node = self.head
        while node != None:
            print(node.getData())
            node = node.getNext()
```

10.5 Array based queues

10.5.1 static arrays

We should note that the greatest strength of an array is the ability to access any element quickly and efficiently via the index (order BigO(1)). This is not valuable within the context of a queue as you are only inspecting elements at the first position and are only adding elements at the rearmost position.
Compared to the linked list where we can adjust the pointers as required to re-size the Data Structure, in the case of an array the dequeue and enqueue methods both require costly operations in terms of processor time.
A traditional array is not a suitable data structure when we have these sort of requirements. One solution is to use a circular array.

When we examine the Big-O notation of the various methods required for our queue class, we find that in the case of a linked list, all of the essential operations (enqueue and dequeue) are of order Big-O(1). However in the case of the Array these methods have order Big-O(N).

Static Array Algorithm

```
1  def enqueueArray(array, element):
2      if isinstance(array, list):
3          array.append(element)
4      else:
5          print(array, " is not a list")
6
7  def dequeueArray(array):
8      if isinstance(array, list):
9          if len(array) > 0:
10             element = array[0]
11             for i in range(1, len(array)-1):
12                 array[i-1] = array[i]
13             del array[-1]
14             return element
15     else:
16         print(array, " is not a list")
```

Figure 10.1: Here is an example of the Static Array algorithm from the lecture notes implemented as a list in python.

10.6 Deques

Double ended queues; we can work on these from both ends. This allows us to enqueue or dequeue from either end of the queue as desired. An example of this is the palindrome. We can dequeue from the front and back of the deque and compare the elements. In a palindrome these will always contain the same value. Another example is trains; instead of turning around the entire train and all the carriages you just take the engine off the front (now end) of the train and add an engine onto the former rear, now front, of the train. Internet history is also stored as a deque; elements are added to the front of the list and older elements are then removed from the tail end of the list.

10.7 Circular array

The Circular Array is a similar concept to a deque. We take an array and virtually "turn it around" making the head and tail point to the same element. This is an alternative to the linked list, used when an array size is fixed. An example is the size of the class. As long as you know the capacity of the classroom you can use a circular array. We need to wrap around and keep a track of tail and head. While empty, both head and tail point to the same place. The new data is added and the tail is moved to point to the location where the next element will be inserted. To remove the data (or dequeue), we move the head. Once the array's capacity is reached the tail index re-initializes to zero. We use the modulus operator to reset the overlapping index.

```
head = (head+1) \% maxSize
```

```
tail = (tail+1) \% maxSize
```

10.8 Conclusion

A Circular array works only for a fixed capacity set when creating the array, and does not allow you to exceed this capacity. Since this ADT is meant to simulate a queue, it is by definition unidirectional. Elements are always added to the tail even when "earlier" locations are available.

Links to helpful information

- Geeks for Geeks
- Interactive Python
- Nathaniel G Martin
- Study Tonight
- Python Central