# L13 - Queues

November 13, 2019

## 1 Queues with Linked Lists

### 1.1 What is a queue?

A queue is a collection in which the elements are maintained in the same order in which they were added. A linear collection is the simplest kind and can be implemented as a singly-linked-list.

Generally, we make a queue as a **f**irst-**in**-**f**irst-**o**ut, or **FIFO**.

Our queue will have two important operations, which we expect to execute many times:

#### 1.1.1 `enqueue()`

- add a new element to the *back* of the queue

#### 1.1.2 `dequeue()`

- remove an element from the *front* of the queue
- return the value of the element

### 1.2 Design

We have some design options for implementing the queue structure. Consider:

- functional requirements
  - actually doing the job
- non-functional requirements
  - execution speed
  - memory footprint
  - scalability

### 1.3 Queue by singly-linked list - first go

The first node in the list represents the front of the queue. The last node in the list represents the back of the queue.

- `dequeue()` will remove and return the first node in the queue, in $O(1)$

- `enqueue()` will add the new node to the back of the list in $O(n)$
    - this is very inefficient to add a new element, as we must traverse the entire queue to get to the end.

### 1.4 Queue by singly-linked list - second go

What if we change it so the first node in the list is the *back* of the queue?

- `enqueue()` will add the new node to the back of the list in $O(1)$
- `dequeue()` will remove and return the first node in the queue, in $O(n)$

All we've done is moved our inefficiency from `dequeue()` to `enqueue()`. Nuts.

### 1.5 Queue by singly-linked list - third go

What if we add another pointer, `rear`, to point to the first node of the queue? (The first node in the list is the *back* of the queue, still.)

- `enqueue()` is $O(1)$ and following `front` is $O(1)$
- `dequeue()` is *still* $O(n)$! Why?
    - when we add to the queue, we are just going to make pointers to pointers to `front`
    - the `rear` pointer must follow all the `.next` pointers!

... and we've just accomplished the same efficiency all over again. Rats.

### 1.6 Queue by singly-linked list - fourth go

Change it back so that the node referred to by the `head` pointer is the front of the queue (the node acted upon by `dequeue()`.

- `front` and `dequeue` are $O(1)$
    - don't need to traverse anything to get from `head` to the first entry
- `enqueue()` is also $O(1)$
    - `rear` points directly to the last node in the list
    - we can change `rear->next` to point to our new node
    - then move `rear` to the *new* last node once it is created
        - this is make before break
- it's probably a good idea to define a struct like `queue_t` to store the pointers to the front and the back of the list.

**Finally**, we can efficiently add and remove elements from the list!

## 1.7 Circular Singly-Linked List

- Instead of assinging the pointer in the last node to `NULL`, make it point back to the front!
    - eliminates one pointer - only need `rear` to get into the list
    - when adding a node
        - break the old last-to-front pointer and move it to the new node
            - `rear->next` should point to the output of `malloc()`
        - point `rear` to the new node output by `malloc()`

### 1.7.1 Can we change to use only `front` and not `rear`?

- Problematic:
    - Need some way to find the end of the list without the `rear` pointer
    - Changes `enqueue()` to $O(n)$ complexity

## 1.8 A primer on Big-O Complexity Notation

https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/