Lecture 15

Queues

- Queues
  - Metaphors
- Array Implementations
  - Linear Indexing
  - Circular Indexing
- Linked-List Implementations
  - Singly-Linked
  - Singly-Linked with Tail (Circular Singly-Linked)

Queues

Queue

In a *queue* the only element that can be accessed is the earliest added element.
We refer to it using the acronym *FIFO* for *First-In First-Out*.

A queue of some human objects or elements.

Elements are *added* to the back of a queue. It is also known as *enqueue*. It is `add` in `Linear`.
Elements are *removed* from the front of a queue. It is also known as *dequeue*. It is `remove` in `Linear`.

Implementations in Pseudocode

In this lecture, we'll consider various implementations of queues.

1. We'll start with the most obvious implementations involving arrays and linked lists, then we will show how additional thought can lead to more efficient and/or concise implementations.

2. When examining stacks, we considered Java code in `structures` package. In this lecture, we'll instead focus on pseudocode. There is no single "best" overall format for pseudocode. For example, if A is an array, then "*A.size*" or "*|A|*" or "*n = size of A*" are all reasonable choices. However, you should keep in mind the following goal:

   - A programmer should be able to translate pseudocode into their language of choice without having to know the intricacies of a particular language, and without having to think too hard.

   For example, "*A = new Vector<Integer>(10)*" is too language dependent, while "*m = the number of distinct values in A*" is probably too high-level for this course (i.e., how is m computed?).

We'll also consider several different queueing metaphors that can be helpful when comparing different implementations.

Metaphors

Metaphors for Queues

In each situation below, the people are served in the order that they arrive, so they are examples of queues. However, there are differences between the situations.

Waiting for cheese at Big Y using a ticket server.

A typical line-up that is issued at the movies or a grocery store.

Taking and making orders at a restaurant bar.

These situations are suggestive of different types of implementations.

- What is moved or updated after each enqueue or dequeue?

Array-Based Implementations

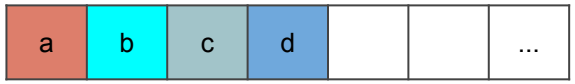
Array Implementation #1

Linear Indexing

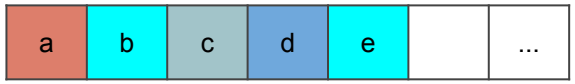
Array Implementation (Line-Up)

The most obvious implementation with an array `A` is similar to a line-up queue (e.g., for movies).

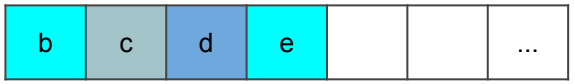
- The front of the queue is in `A[0]`. The back of the queue is in `A[num-1]`.
- When dequeuing, all of the values move to the left in the array.



Maximum size is `max = 1000`.
Number of elements is `num = 4`.



Enqueue adds element to the back.
Number of elements is `num = 5`.



Dequeue removes element from the front.
Number of elements is `num = 4`.

What happens when the maximum size `max` is reached?

Let's decide to not resize the array (since a Vector-based implementation will do that).

Pseudocode: Enqueue and Dequeue in an Array with Linear-Indexing

Below is pseudocode for adding and removing an element from a queue using an array in which the indices are used linearly.

```
// A is an array A[0]...A[max-1] of size max.
// num is the number of values in the queue.

function enqueue(newValue)

  // Error if there is no room in the array.
  if num == max then
    return error

  // Add the value in the next space.
  A[num] = newValue

  // Increment num.
  num += 1
```

Adding a value to the queue.

```
// A is an array A[0]...A[max-1] of size max.
// num is the number of values in the queue.

function dequeue()

  // Error if there are no values in the array.
  if num == 0 then
    return error

  // Remember the first value and then move
  // all of the values forward in the array.
  firstValue = A[0]
  for i = 0, 1, …, num-2 do
    A[i] = A[i+1]

  // Decrement num and return the old first value.
  num -= 1
  return firstValue
```

Removing a value from the queue.

How much time do these operations take?

- Enqueue is O(1)-time.
- Dequeue is O(n)-time where n is the number of values in the queue (i.e., n = `num`).

Discussion: Improving the Array-Based Implementation

Our first array-based implementation had the following run-times.

- O(1)-time for enqueue (so long as there is space in the array).
- O(n)-time for dequeue.

Think about this for 1 minute.
Then discuss it with your neighbor for 2 minutes.

Can we create an improved array-based implementation that has the following run-times:

- O(1)-time for enqueue (so long as there is space in the array).
- O(1)-time for dequeue.

Hint: Allow the front of the queue to move away from `A[0]` and/or think about a ticket taker.

Array Implementation #2

Circular Indexing

Array Implementation (Ticket Server)

A ticket server keeps track of a queue in a different way.

- When enqueuing, it adds values to the back (except when wrapping around; see note below).
- When dequeing, it increments the starting position, and the other values don't move.

	a	b	...	x		
...	452	453	...	853	854	...

Maximum size is 1000.
Number of values in queue is 403.

	a	b	...	x	y	
...	452	453	...	853	854	...

Add element to the end.
Number of values in queue is 404.

	(a)	b	...	x	y	
...	452	453	...	853	854	...

Remove element from the front.
Number of values in queue is 403.

What happens when the numbers run out?

- In the ticket server, the numbers wrap around and the next ticket number is 1.
- In the array, the new values are now added to the front of the array.

We refer to this as *circular indexing*.

Pseudocode: Enqueue and Dequeue in an Array with Circular-Indexing

Below is pseudocode for adding and removing an element from a queue using an array in which the indices are used circularly.

```
// A is an array A[0]...A[max-1] of size max.
// num is the number of values in the queue,
// and they are stored in A[first]...A[last]
// where last = (first + num - 1) % max.

function enqueue(newValue)

  // Error if there is no room in the array.
  if num == max then
    return error

  // Increment last circularly, then
  // add the value in the new last position.
  last = (last + 1) % max
  A[last] = newValue

  // Increment num.
  num += 1
```

Adding a value to the queue.

```
// A is an array A[0]...A[max-1] of size max.
// num is the number of values in the queue,
// and they are stored in A[first]...A[last]
// where last = (first + num - 1) % max.

function dequeue()

  // Error if there are no values in the array.
  if num == 0 then
    return error

  // Remember the first value and then
  // increment the first index circularly.
  firstValue = A[first]
  first = (first + 1) % max

  // Decrement num and return the old first value.
  num -= 1
  return firstValue
```

Removing a value from the queue.

How much time do these operations take?

- Enqueue is O(1)-time.
- Dequeue is O(1)-time.

Efficiency for Queue Operations with Arrays

The efficiency of the two main queue operations for our two array-based implementations.

	Array (linear indices)	Array (circular indices)
enqueue	O(1)-time	O(1)-time
dequeue	O(n)-time	O(1)-time

Big-O measurements for a queue containing n elements.

Recall that we do not allow the array to be resized in these implementations.
If we did, then the enqueue times would need to be adjusted.
Alternatively, we could use a Vector instead of an array.

Linked-List Implementations

Metaphor for Linked List Based Queues

In this situation, the bartender and/or patrons keep track of who arrives after whom, so it suggests an implementation using a linked list.

- The `head` is the front of the list (i.e. first person).

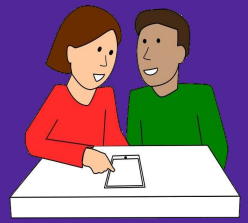
Taking orders at a restaurant bar.

Which type of linked list should we use when implementing a queue?

Discussion: Linked List-Based Implementations

Let's aim for the simplest implementation that has the following run-times.

- O(1)-time for enqueue.
- O(1)-time for dequeue.



Discuss this with your neighbor for 3 minutes.

Consider each of the following linked list types.

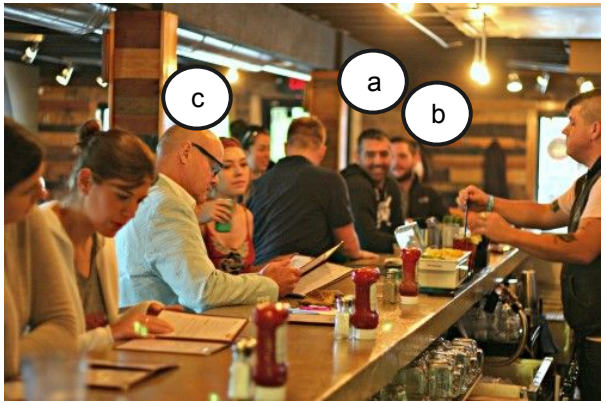
- Singly-linked with a `head` reference.
- Singly-linked with a `head` and `tail` reference. This is similar to a circular linked list.
- Doubly-linked list with `head` and `tail` reference.

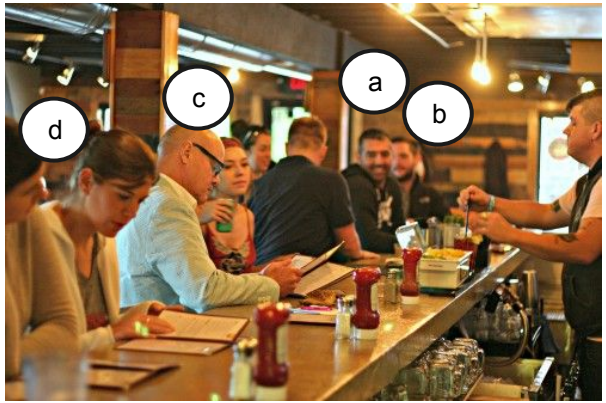
Which is the simplest that will allow for the above run-times?

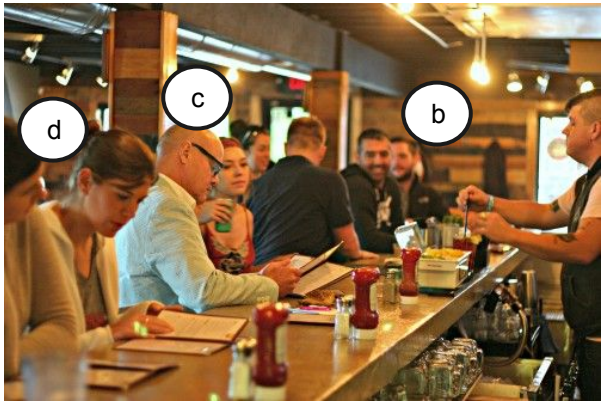
Queue in a Linked List

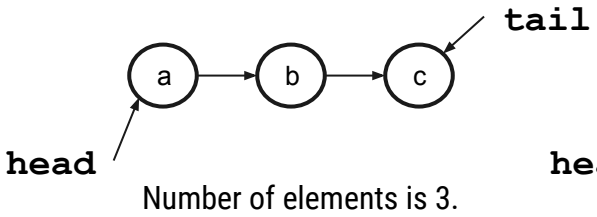
Taking orders at a restaurant bar is similar to storing a queue in a linked list.

Should we keep track of the `tail`?

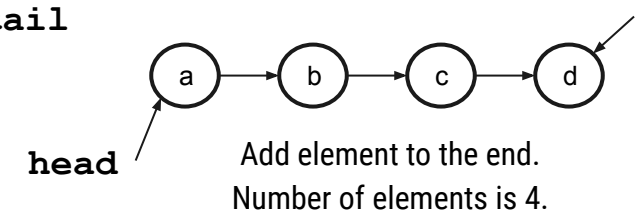




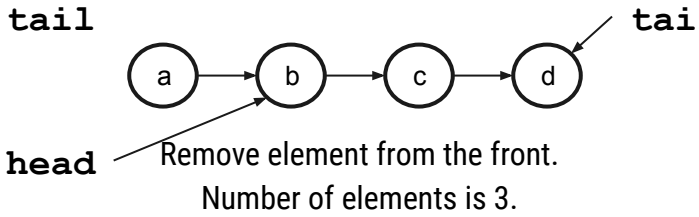




Number of elements is 3.



Add element to the end.
Number of elements is 4.



Remove element from the front.
Number of elements is 3.

If we don't maintain the `tail`, then we'll need to recompute it on every enqueue operation.

Pseudocode: Enqueueing in Linked List

Below is pseudocode for adding an element to a queue either using or not using `tail`.

```
// Each node has .value and .next
// head is the first node.

function enqueue(newValue)

  // Create the new node
  newNode = new Node()
  newNode.value = newValue
  newNode.next = null

  // Find the end of the list
  temp = head
  while temp.next != null do
    temp = temp.next

  // Add the new node after the last node
  temp.next = newNode
```

Linked list without `tail`

```
// Each node has .value and .next
// head is the first node.
// tail is the last node.

function enqueue(newValue)

  // Create the new node
  newNode = new Node()
  newNode.value = newValue
  newNode.next = null

  // Add the new node after tail
  tail.next = newNode
  tail = newNode
```

Linked list with `tail`

Notice that the `tail` variable helps us enqueue faster.

Are there any errors in the code or any edge cases that are missing?

```
// Each node has .value and .next
// head is the first node.

function enqueue(newValue)

  // Create the new node
  newNode = new Node()
  newNode.value = newValue
  newNode.next = null

  // Edge case: queue is currently empty
  if head == null then
    head = newNode
    return

  // Find the end of the list
  temp = head
  while temp.next != null do
    temp = temp.next

  // Add the new node after the last node
  temp.next = newNode
```

```
// Each node has .value and .next
// head is the first node.
// tail is the last node.

function enqueue(newValue)

  // Create the new node
  newNode = new Node()
  newNode.value = newValue
  newNode.next = null

  // Edge case: queue is currently empty
  if head == null then
    head = newNode
    tail = newNode
    return

  // Add the new node after tail
  tail.next = newNode
  tail = newNode
```

When searching for bugs, it can be helpful to consider everything that the code must accomplish. In this case, the `head` (and `tail`) reference must be updated somewhere in these functions. More specifically, when the queue is empty, the references will need to be updated.

- Dummy nodes can be used to reduce the number of edge cases in linked structures.

Pseudocode: Dequeuing in Linked List

Below is pseudocode for removing an element from a queue either using or not using `tail`.

```
// Each node has .value and .next
// head is the first node.

function dequeue()

  // Error if the queue is empty
  if head == null then
    return error

  // Get the first value
  firstValue = head.value

  // Update the head
  head = head.next

  return firstValue
```

Linked list without `tail`

```
// Each node has .value and .next
// head is the first node.
// tail is the last node.

function dequeue()

  // Error if the queue is empty
  if head == null then
    return error

  // Get the first value
  firstValue = head.value

  // Update the head (and tail)
  head = head.next
  if head == null then
    tail = null

  return firstValue
```

Linked list with `tail`

Does the `tail` variable helps us dequeue faster? No.

Does it make the implementation more complicated? Yes, a little bit.

Efficiency for Queue Operations in Linked List

The efficiency of the two main queue operations for our two linked list-based implementations.

	Linked List (no tail)	Linked List (with tail)
enqueue	O(n)-time	O(1)-time
dequeue	O(1)-time	O(1)-time

Big-O measurements for a queue containing n elements.

What would the run-times be for doubly linked lists with `head` and `tail`? O(1)-time for both. Using a singly linked list with a `tail` reference is optimal in terms of time and simplicity.