Syrian Private University

Algorithms & Data Structure I

Instructor: Dr. Mouhib Alnoukari

8

QUEUES AND DEQUES

Queues, and Deques

- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- A deque is a double-ended queue—items can be inserted and removed at either end

Abstract Queue

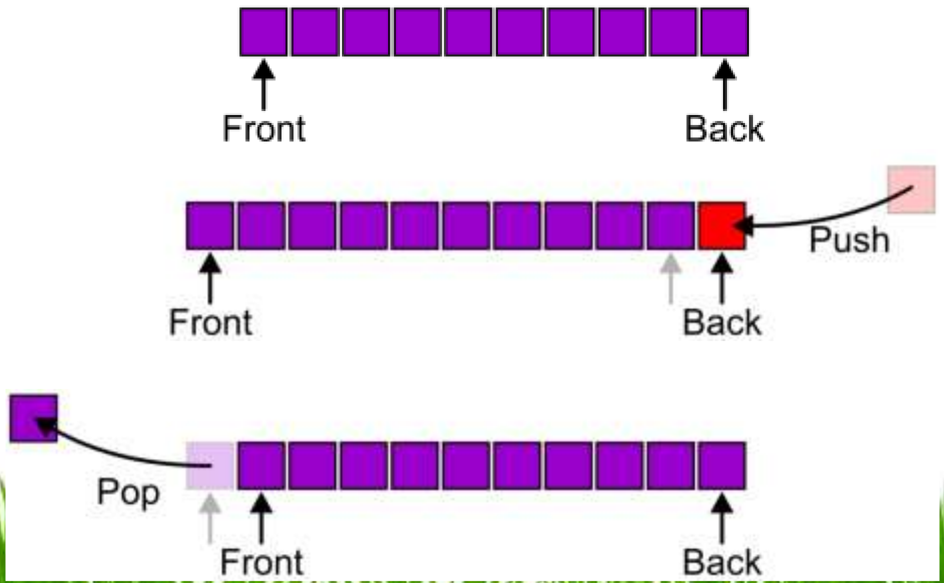
An Abstract Queue (Queue ADT) is an abstract data type that emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- There are no restrictions on objects inserted into (*pushed onto*) the queue—that object is designated the back of the queue
- The object designated as the *front* of the queue is the object which was in the queue the longest
- The remove operation (*popping* from the queue) removes the current *front* of the queue

Abstract Queue

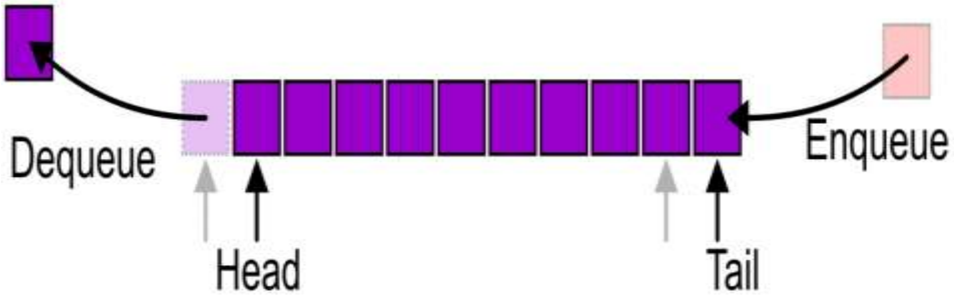
Also called a *first-in–first-out* (FIFO) data structure

– Graphically, we may view these operations as follows:



Abstract Queue

Alternative terms may be used for the four operations on a queue, including:



Abstract Queue

There are two exceptions associated with this abstract data structure:

– It is an undefined operation to call either pop or front on an empty queue

Applications

The most common application is in client-server models

- Multiple clients may be requesting services from one or more servers
- Some clients may have to wait while the servers are busy
- Those clients are placed in a queue and serviced in the order of arrival

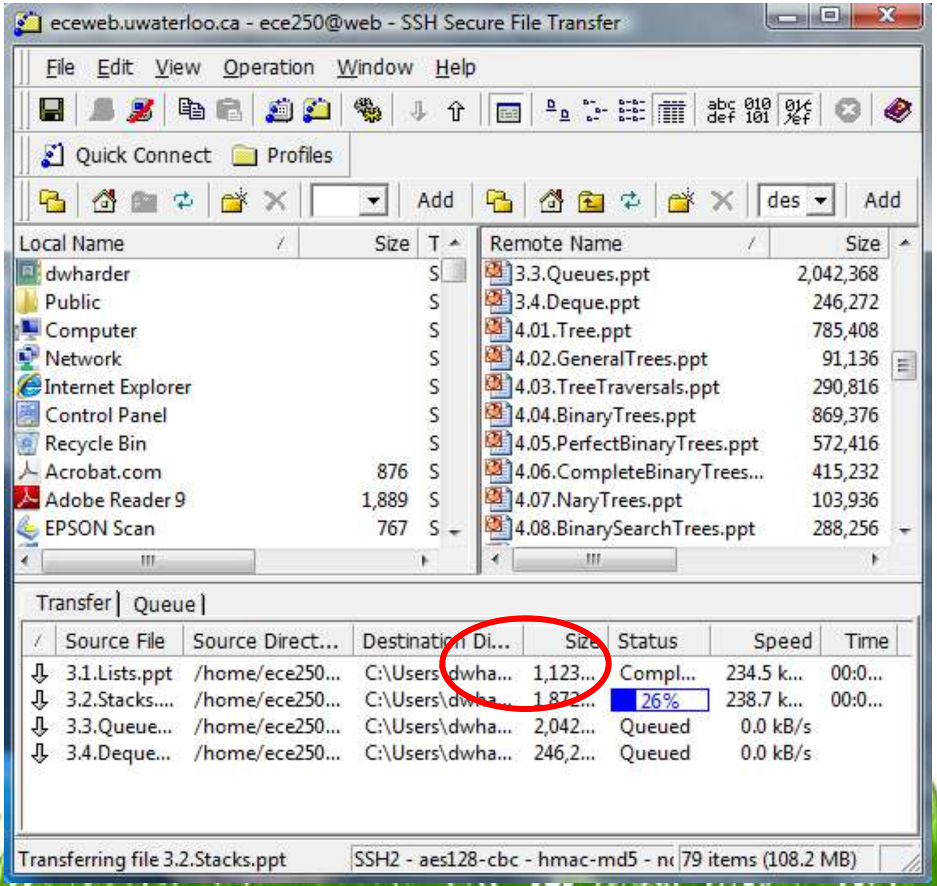
Grocery stores, banks, and airport security use queues

Most shared computer services are servers:

- Web, file, ftp, database, mail, printers, WOW, *etc*.

Applications

For example, in downloading these presentations from the ECE 250 web server, those requests not currently being downloaded are marked as "Queued"



Implementations

We will look at two implementations of queues:

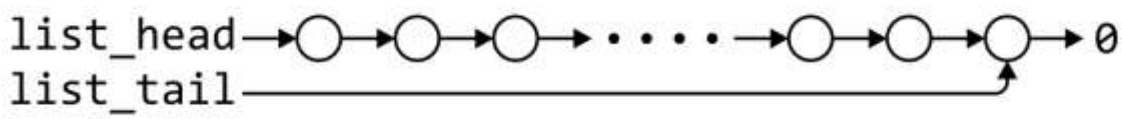
- Singly linked lists
- Circular arrays

Requirements:

- All queue operations must run in $\Theta(1)$ time

Linked-List Implementation

Removal is only possible at the front with $\Theta(1)$ run time

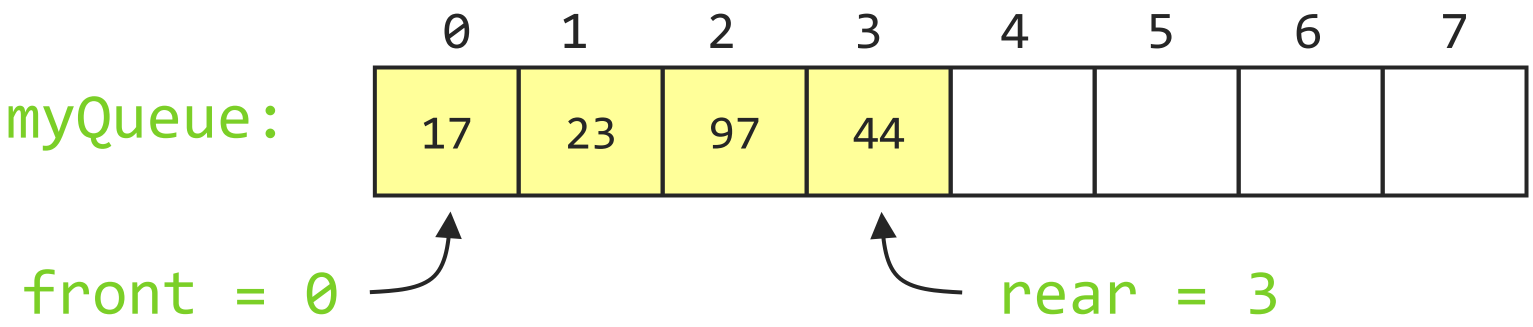


	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behaviour of an Abstract Queue may be reproduced by performing insertions at the back

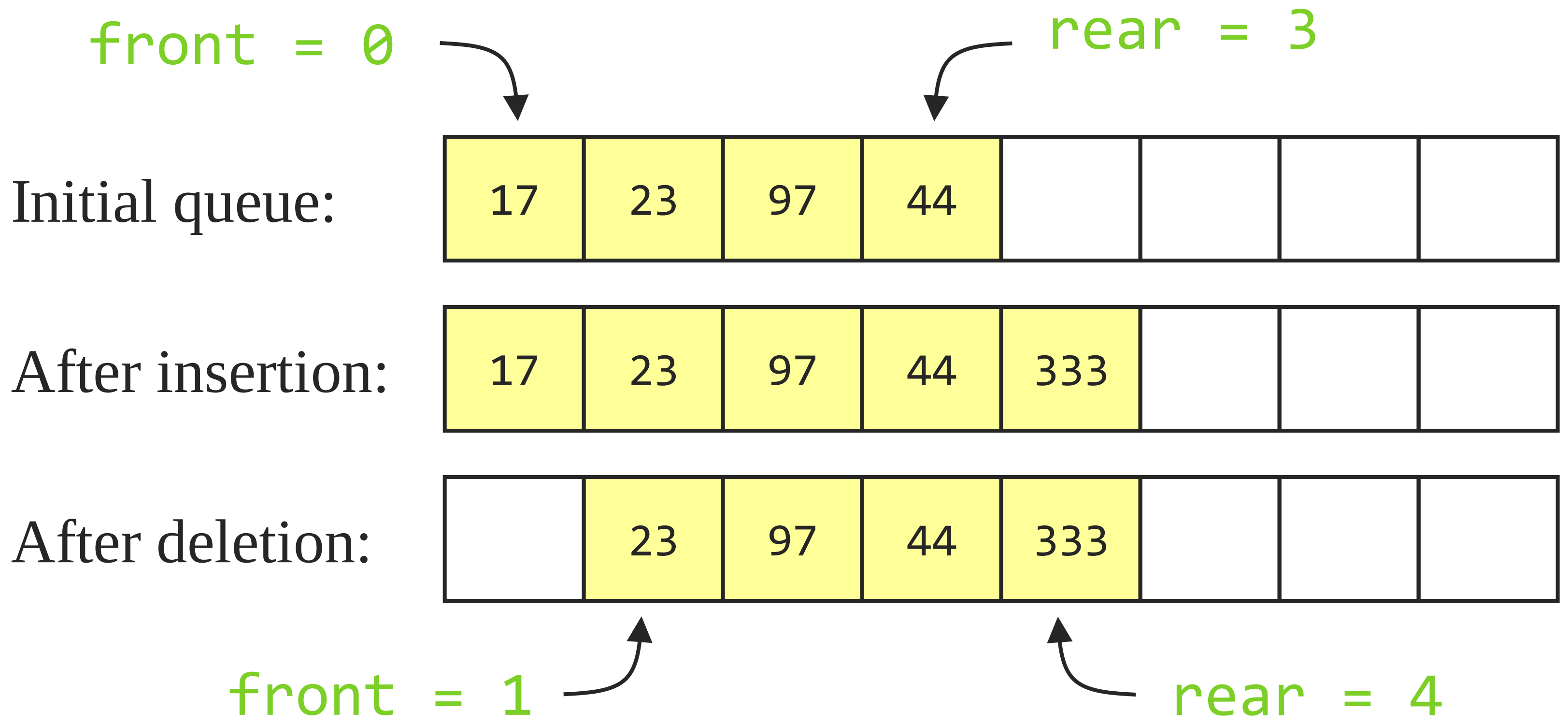
Array implementation of queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

Array implementation of queues



- Notice how the array contents “crawl” to the right as elements are inserted and deleted
- This will be a problem after a while!

Array Implementation

A one-ended array does not allow all operations to occur in $\Theta(1)$ time

	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Array Implementation

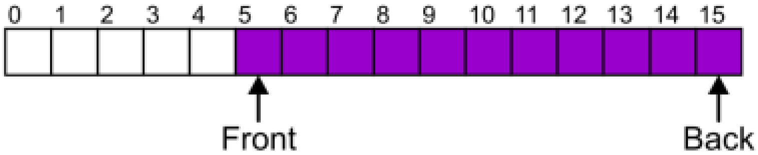
Using a two-ended array, $\Theta(1)$ are possible by pushing at the back and popping from the front

	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Remove	$\Theta(1)$	$\Theta(1)$

Member Functions

Suppose that:

- The array capacity is 16
- We have performed 16 pushes
- We have performed 5 pops
  - The queue size is now 11



- We perform one further push

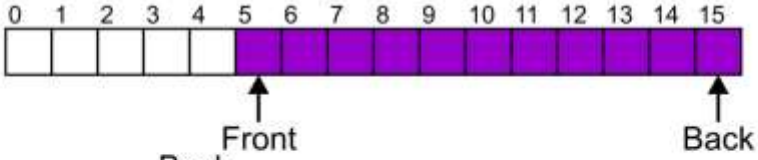
In this case, the array is not full and yet we cannot place any more objects in to the array

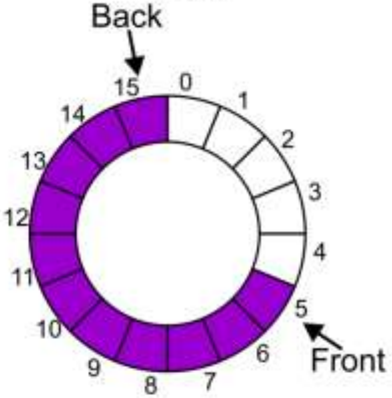
Member Functions

Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

…, 15, 0, 1, …, 15, 0, 1, …, 15, 0, 1, …

This is referred to as a *circular array*

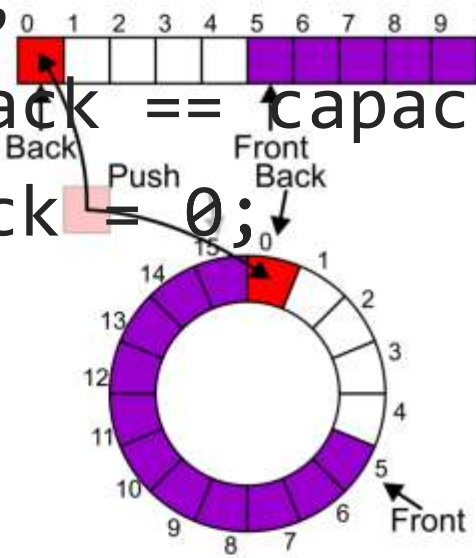




Member Functions

Now, the next push may be performed in the next available location of the circular array:

```
++iback;
if ( iback == capacity() ) {
    iback = 0;
}
```



Exceptions

As with a stack, there are a number of options which can be used if the array is filled

If the array is filled, we have five options:

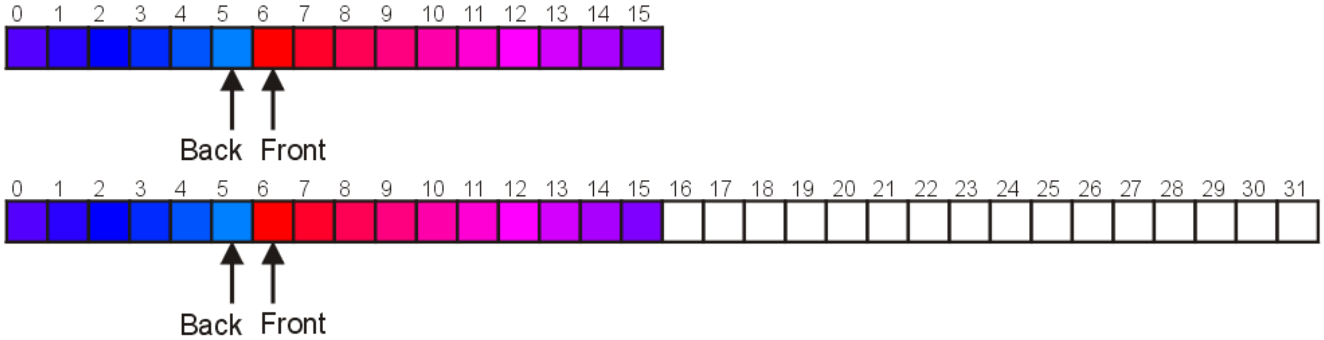
– Increase the size of the array
– Throw an exception
– Ignore the element being pushed
– Put the pushing process to “sleep” until something else pops the front of the queue

Include a member function **`bool full()`**

Increasing Capacity

Unfortunately, if we choose to increase the capacity, this becomes slightly more complex

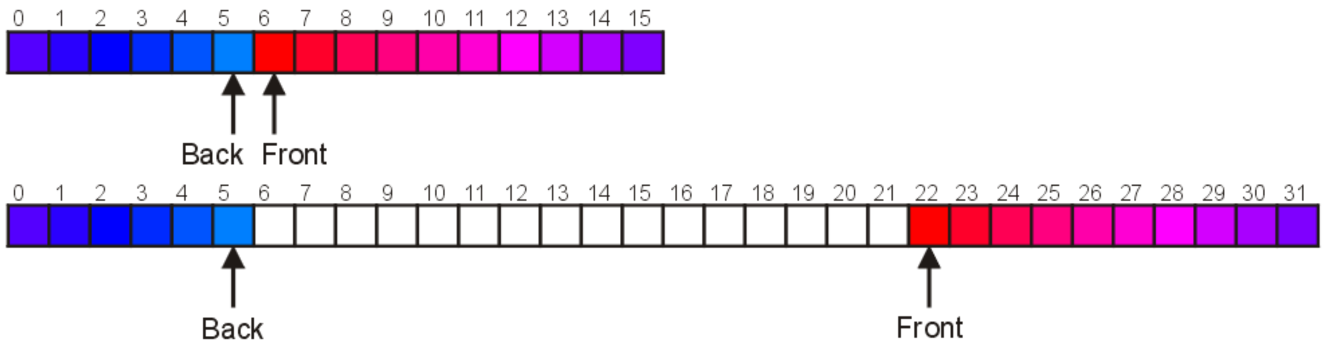
– A direct copy does not work:



Increasing Capacity

There are two solutions:

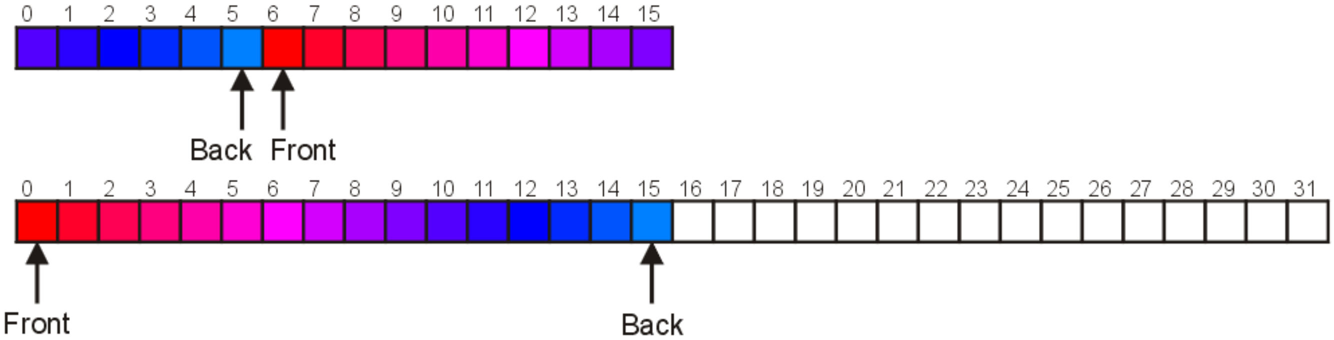
- Move those beyond the front to the end of the array
- The next push would then occur in position 6



Increasing Capacity

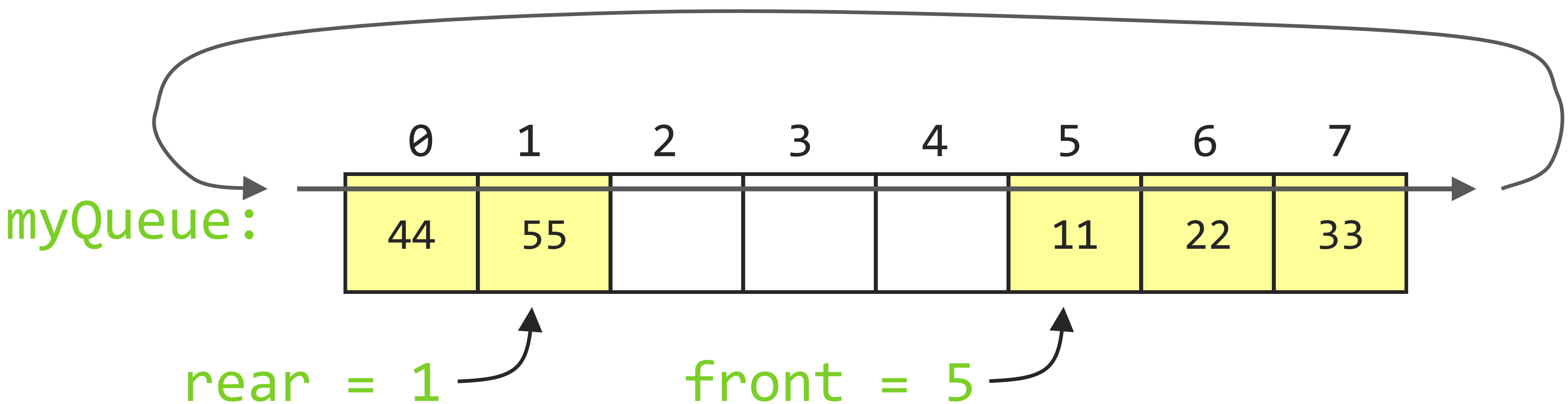
An alternate solution is normalization:

- Map the front back at position 0
- The next push would then occur in position 16



Circular arrays

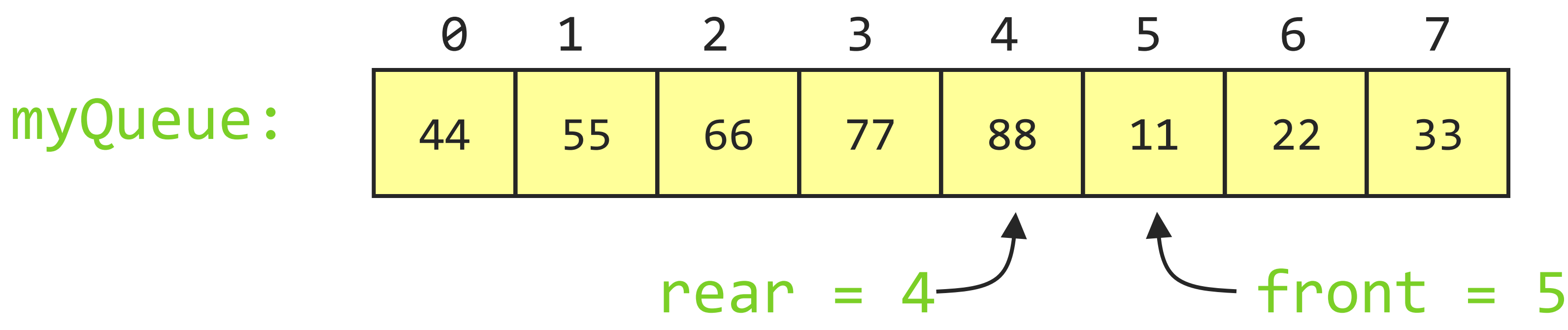
- We can treat the array holding the queue elements as circular (joined at the ends)



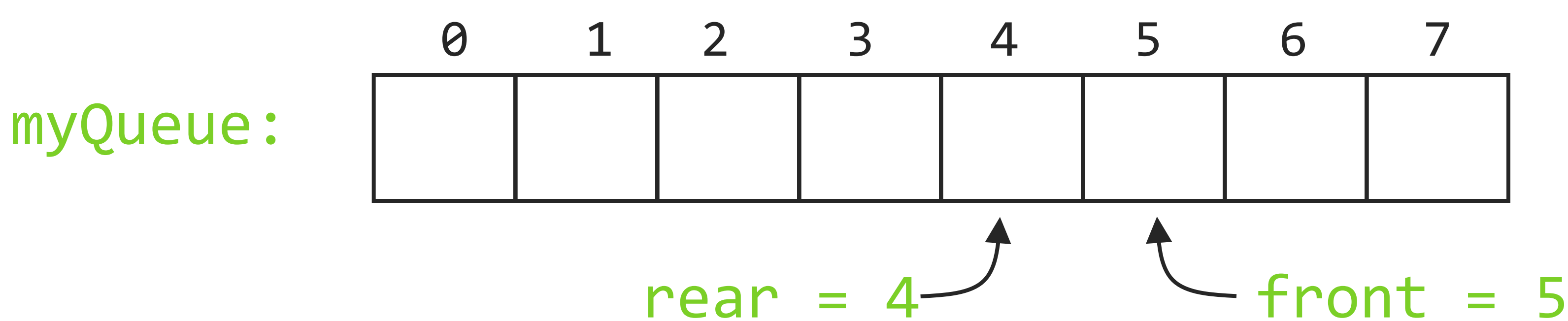
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

Full and empty queues

- If the queue were to become completely full, it would look like this:



- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



This is a problem!

Full and empty queues: solutions

- **Solution #1:** Keep an additional variable

`myQueue:`

0	1	2	3	4	5	6	7
44	55	66	77	88	11	22	33

`count = 8`   `rear = 4`   `front = 5`

- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements

`myQueue:`

0	1	2	3	4	5	6	7
44	55	66	77		11	22	33

`rear = 3`   `front = 5`

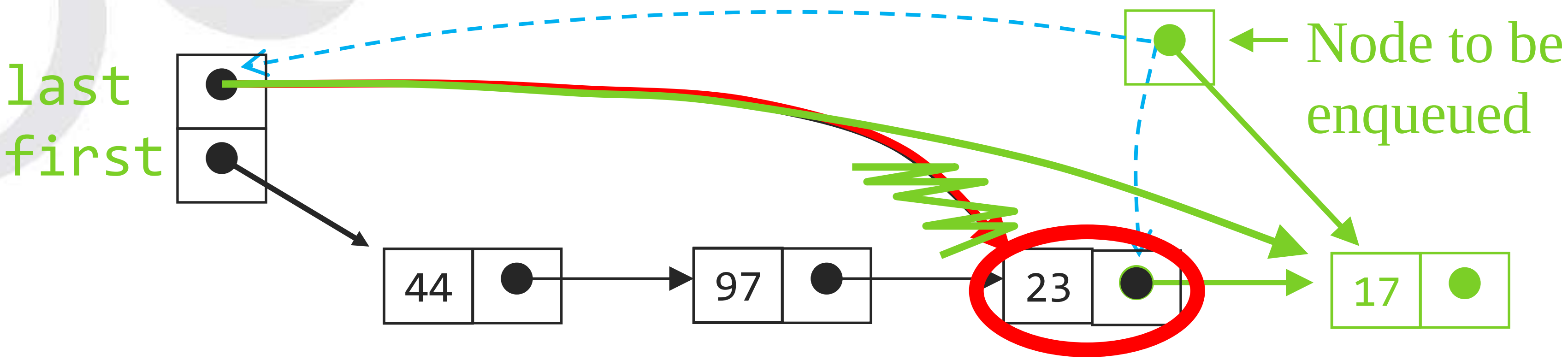
Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in O(1) time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list

SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Enqueueing a node



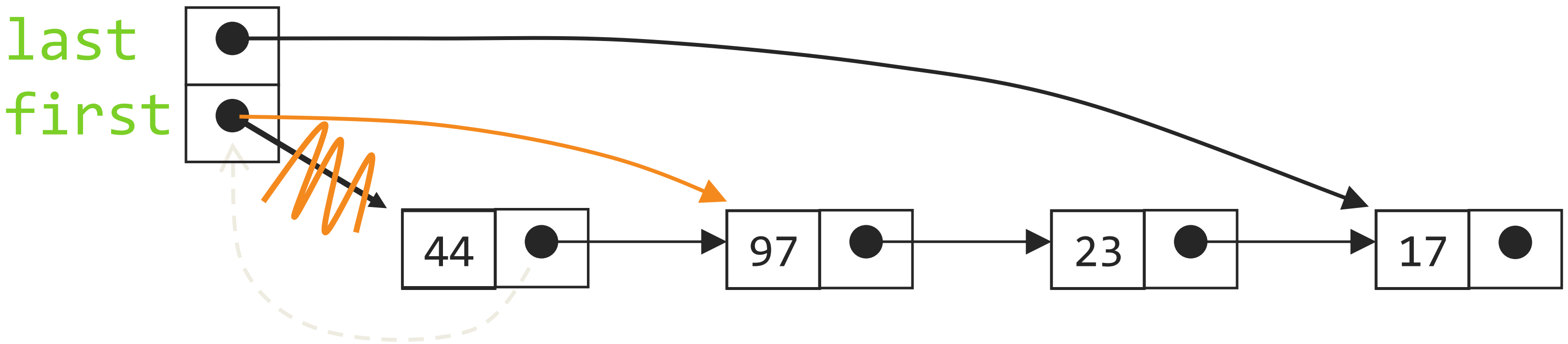
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the `last` pointer in the list header

Dequeueing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to `null`
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to `null`

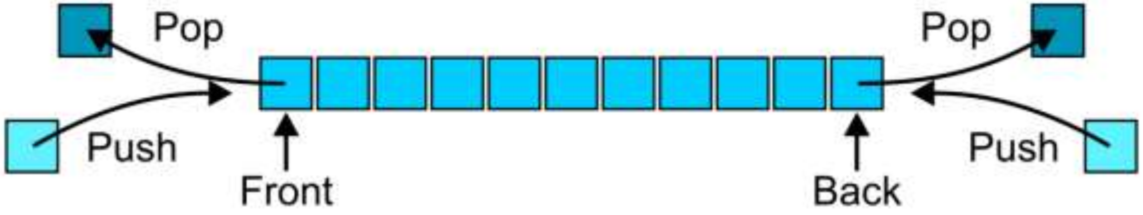
Deques

- A deque is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further

Abstract Deque

An Abstract Deque (Deque ADT) is an abstract data structure which emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Allows insertions at both the front and back of the deque



Abstract Deque

The operations will be called

`front`	`back`
`push_front`	`push_back`
`pop_front`	`pop_back`

There are four errors associated with this abstract data type:

– It is an undefined operation to access or pop from an empty deque

Applications

Useful as a general-purpose tool:

– Can be used as either a queue or a stack

Problem solving:

– Consider solving a maze by adding or removing a constructed path at the front

– Once the solution is found, iterate from the back for the solution

Implementations

The implementations are clear:

– We must use either a doubly linked list or a circular array