CIS 351-Data Structure-Stack-Queue
Mar 10, 2020

Dr. Farzana Rahman

Syracuse University

Why bother with Stack/Queue

- The operations are a subset of List
- We could always just use an AList or LList
- Code is more clear and more safe
- Gives some confidence that the implementation is efficient
  - tuned to support add and remove at the same end of the collection

Stacks, Queues, and Deques

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- 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

Stacks

- Stack: a data structure in which elements are added and removed from one end only
  - Addition/deletion occur only at the top of the stack
  - Last in first out (LIFO) data structure
- Operations:
  - Push: to add an element onto the stack
  - Pop: to remove an element from the stack

Stack Operations

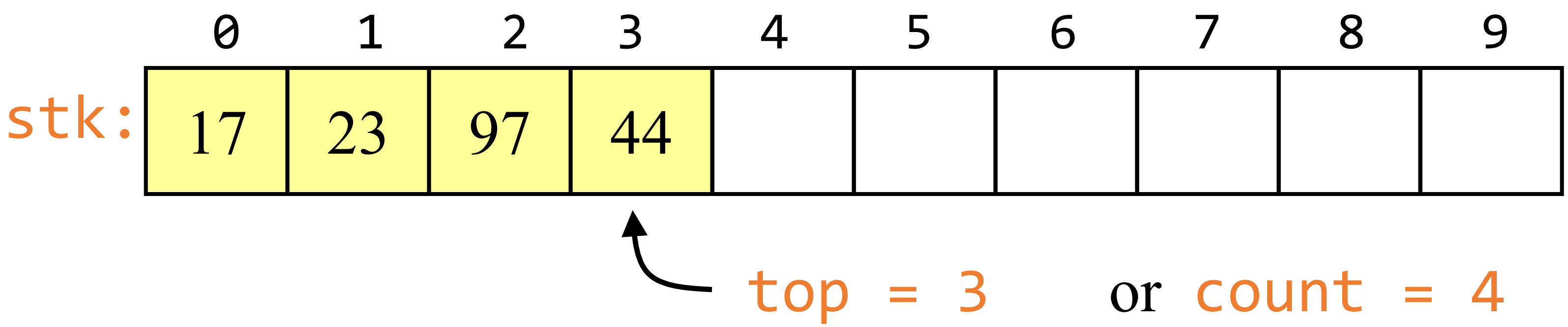
- initializeStack
- isEmptyStack
- isFullStack
- push
- top
- pop

stackADT<Type>
+*initializeStack*(): void +*isEmptyStack*(): boolean +*isFullStack*(): boolean +*push*(Type): void +*top*(): Type +*pop*(): void

Array implementation of stacks

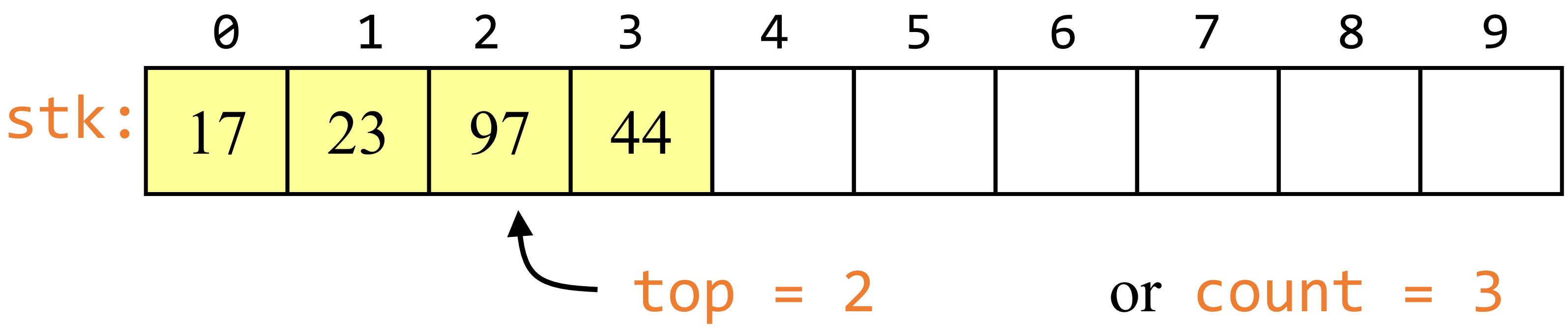
- To implement a stack, items are inserted and removed at the same end (called the top)
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

After popping



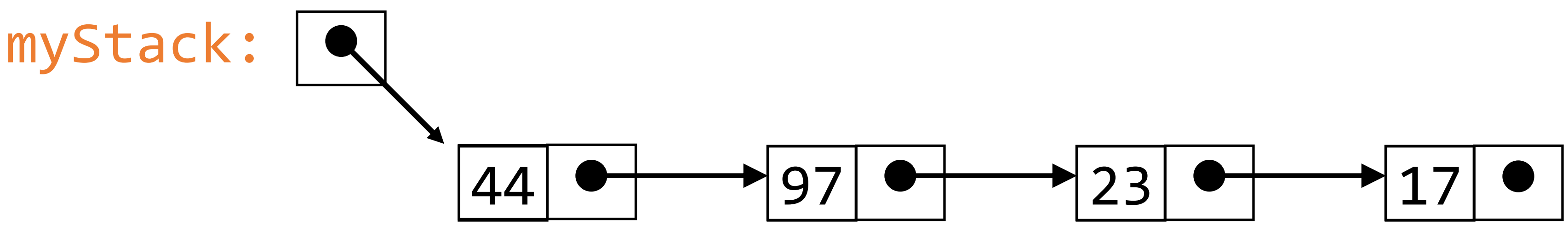
- When you pop an element, do you just leave the “deleted” element sitting in the array?
- The surprising answer is, *“it depends”*
  - If you are programming in Java, and the array contains objects, you should set the “deleted” array element to `null`
  - Why? To allow it to be garbage collected!

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

Linked-list implementation details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `null`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate

Complexity Analysis

- Linked List based

operation	cost
push()	O(1)
pop()	O(1)
topValue()	O(1)
length()	O(1)

- Array based

operation	cost
push()	O(1)
pop()	O(1)
topValue()	O(1)
length()	O(1)

Queues

- **Queue**: set of elements of the same type
  - A Queue is another name for waiting line
  - Example: Submit jobs to a network printer
    - What gets printed first?
      - Print the one first entered the job list
      - Add new print jobs at the end of the queue
- Elements are:
  - Added at one end (the back or rear)
  - Deleted from the other end (the front)
- **First In First Out (FIFO)** data structure
  - Middle elements are inaccessible
- **Example:** Waiting line in a bank

Queue Operations

- Queue operations include:
  - `initializeQueue`
  - `isEmptyQueue`
  - `isFullQueue`
  - `Front/enqueue`
  - `Back/dequeue`
  - `addQueue`
  - `deleteQueue`

Implementation of Queues as Arrays

- Need at least four (member) variables:
  - Array to store queue elements
  - `queueFront` and `queueRear`
    - To track first and last elements
  - `maxQueueSize`
    - To specify maximum size of the queue

The Java Queue class?

- Some languages have a Queue class or queue is part of a library that works with the language
  - Java 1.4 used class LinkedList to offer FIFO functionality by adding methods addLast(Object) and Object(getFirst)
  - Java 1.5 added a Queue interface and several collection classes: ArrayBlockingQueue<E> and LinkedBlockingQueue<E>

Designing a Queue Interface

- Queues typically provide these operations
- *add* adds an element at the end of the queue
- *peek* returns a reference to the element at the front of the queue
- *remove* removes the element from the front of the queue and returns a reference to the front element
- *isEmpty* returns false if there is at least one element in the queue

Specify an interface

- We will use an interface to describe a queue ADT
  - The interface specifies method names, return types, the type of elements to add, and hopefully comments
- interface **`OurQueue`** declares we must be able to **`add`** and **`remove`** any type of element
  - Collection class must have <E> to make it a generic type

Interface to specify a FIFO Queue

```
import java.util.NoSuchElementException;

public interface OurQueue<E> {
  // Return true if this queue has 0 elements
  public boolean isEmpty();

  // Store a reference to any object at the end
  public void add(E newEl);

  // Return a reference to the object at the
  // front of this queue
  public E peek() throws NoSuchElementException;

  // Remove the reference to the element at the front
  public E remove() throws NoSuchElementException;
}
```

Let SlowQueue implement the Queue interface

- We need to store an Object[] *an array of Object objects*
  - avoids having queue of int *and* people *and* cars, *and*...

```
public class SlowQueue<E> implements OurQueue<E> {
private int back;
private Object[] data;
  // ...
```

- Now implement all methods of the **OurQueue** interface as they are written
  - plus a constructor with the proper name

Bad array type queue

- Queue as an array *could* have
  - the front of the queue is *always* in **`data [0]`**

```
public SlowQueue(int max) {
  data = new Object[max];
  back = -1;
}
```

data[0]	data[1]	data[2]	data[3]
null	null	null	null

`back == -1` *So far so good. An empty queue*

First version of add

```java
public void add(E element) {
   // This method will be changed later
   back++;
   data[back] = element;
}
```

- Send an add message

aQueue.add("a");

data[0]	data[1]	data[2]	data[3]
"a"	null	null	null

back == 0

So far so good. A queue of size 1

add another

```java
public void add(E element) {
   back++;
   data[back] = element;
}
```

- Send two more add messages

aQueue.add("b");

aQueue.add("c");

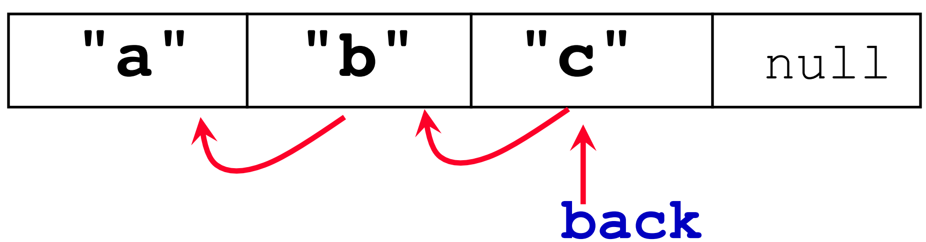
data[0]	data[1]	data[2]	data[3]
"a"	"b"	"c"	null

back == 2

So far so good. A Queue of size 3

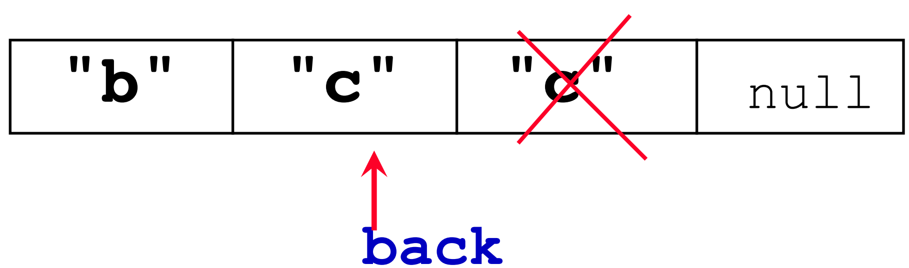
Array Implementation of a Queue

Before remove

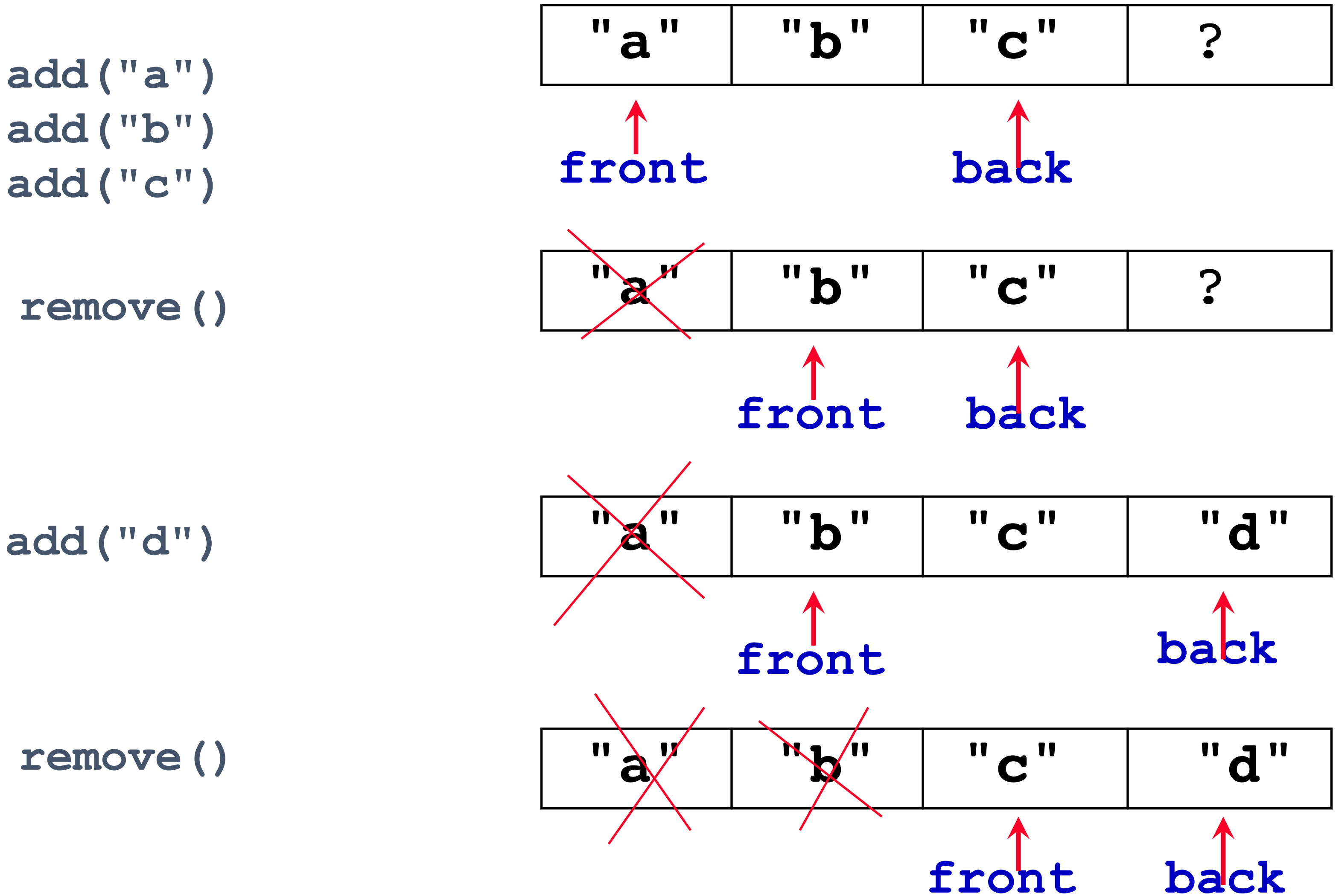


A poor remove algorithm

After remove

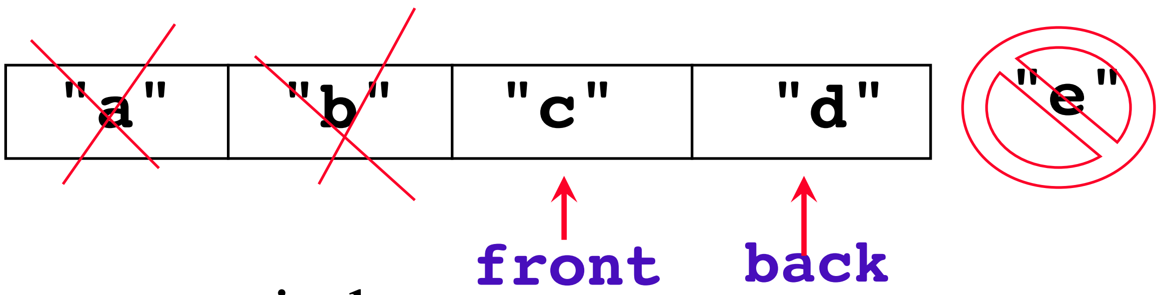


Effect of queue operation using an array with a "floating" front



What happens next when back equals array.length?

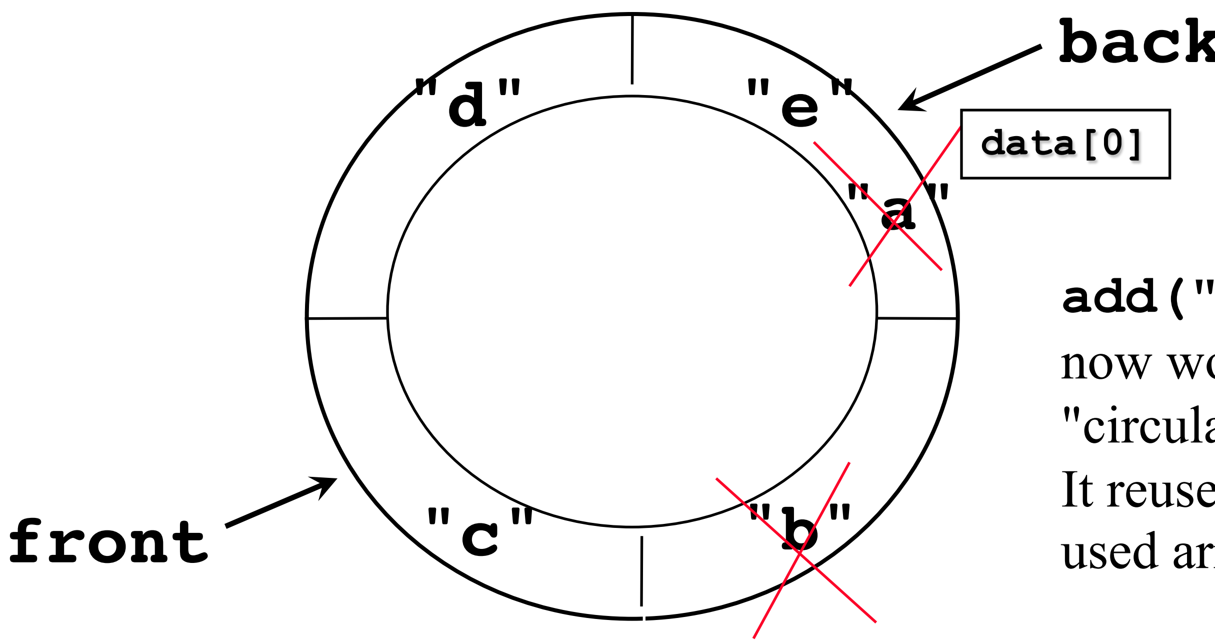
`add("e")`



- `back` indexes the last array index
- However, this queue is *not* full
- Where do you place "e"?
  - data[0] is available

The Circular Queue

- A "circular queue" implementation uses *wraparound* The queue has "c" "d" "e"
  - either increase **`back`** by 1
  - or set **`back`** to 0



`add("e")`
now works in this "circular" queue.
It reuses previously used array indexes

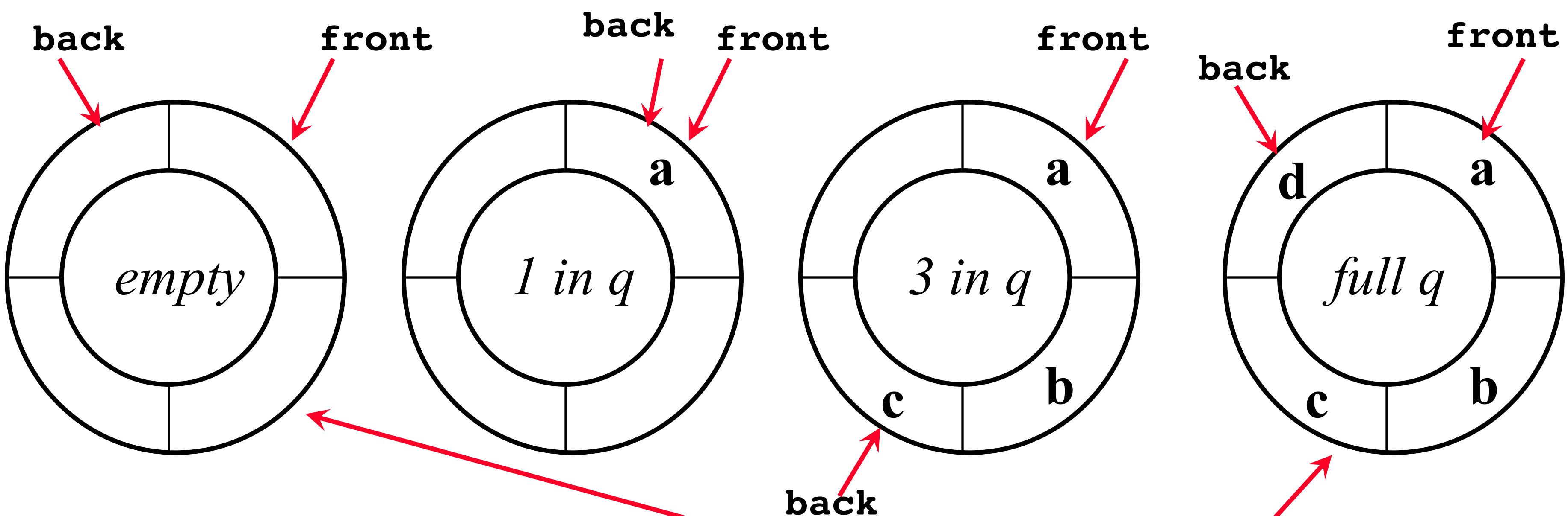
Implementing a Circular Queue

- Still have to work with arrays, not circles
  - In order for the first and last indices to work in a circular manner:
    - increase by one element at a time
    - after largest index in the array, go to zero.

      back = 0 1 2 3 0 1 2 3 0 1 2 3 0 ...
  - could contain code you just wrote on previous slide
- But what is an empty queue?
  - What values should be given to front and back when the queue is constructed?

Problem: A full queue can not be distinguished from an empty queue

One option is to have the constructor place **`back`** one index before **`front`** then increment **`back`** during **`add`**



What does **`back == front`** imply? An empty or full queue?

Corrected Circular Queue

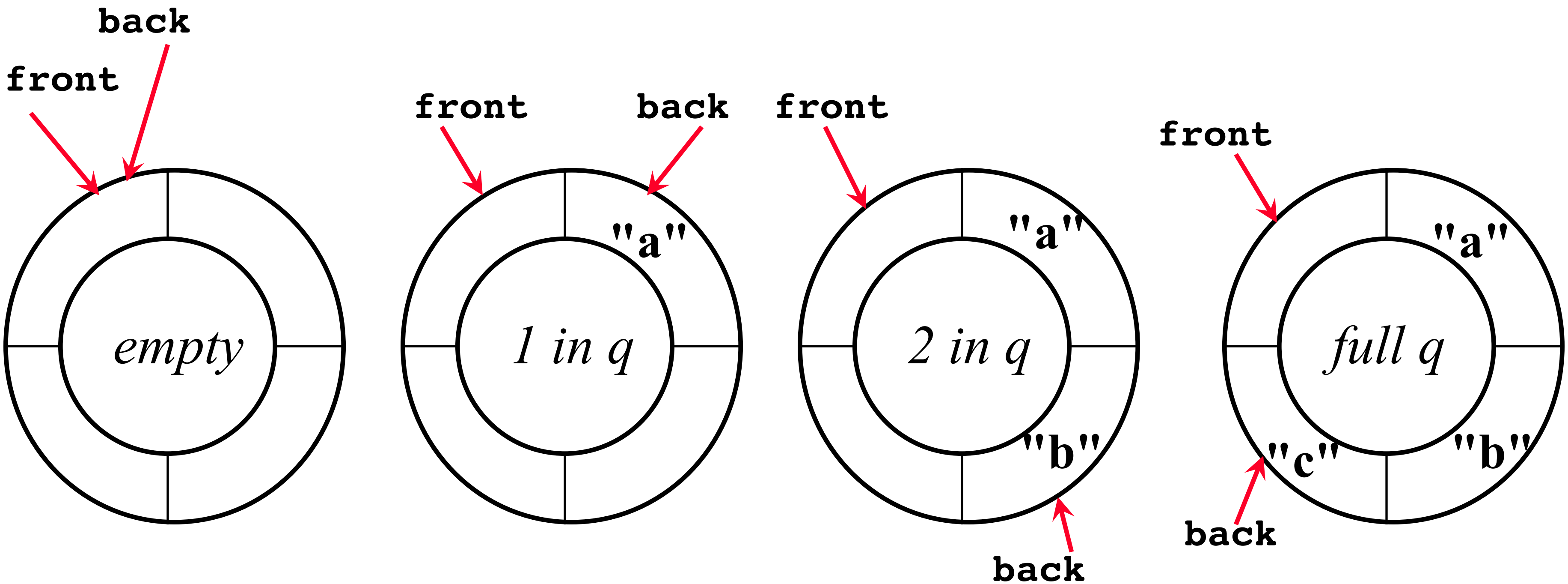
- Use this trick to distinguish between full and empty queues
  - The element referenced by **front** never indexes the front element— the "real" front is located at **`nextIndex(front)`**

```
private int nextIndex(int index) {
  // Return an int to indicate next position
  return (index + 1) % data.length;
}
```

- For example, use this during **`peek()`**

```
return data[nextIndex(front)];
```

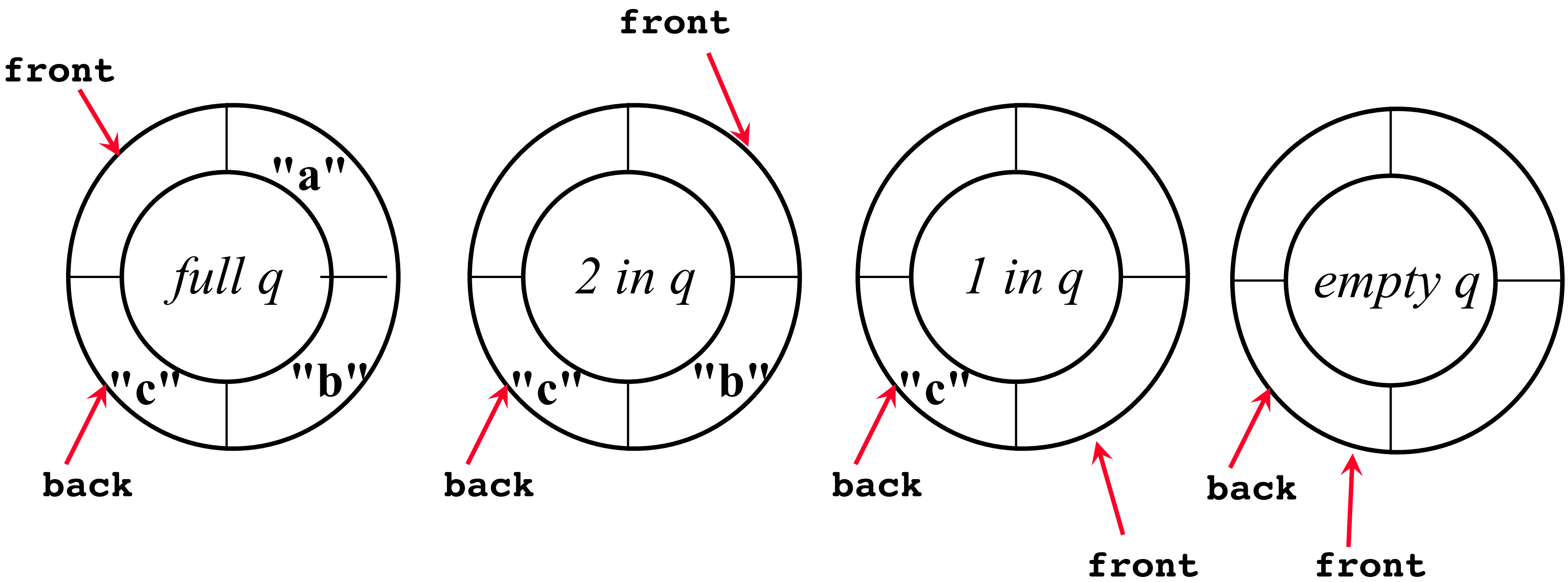
Correct Circular Queue Implementation Illustrated



The front index is always 1 behind the actual front

This wastes one array element *but it's no big deal*

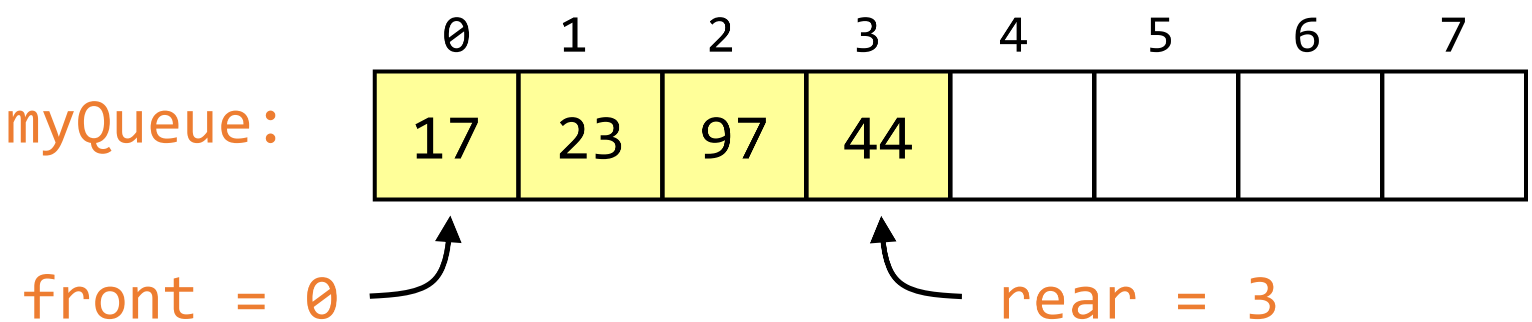
Correct Circular remove Implementation Illustrated



remove three times to make this queue empty

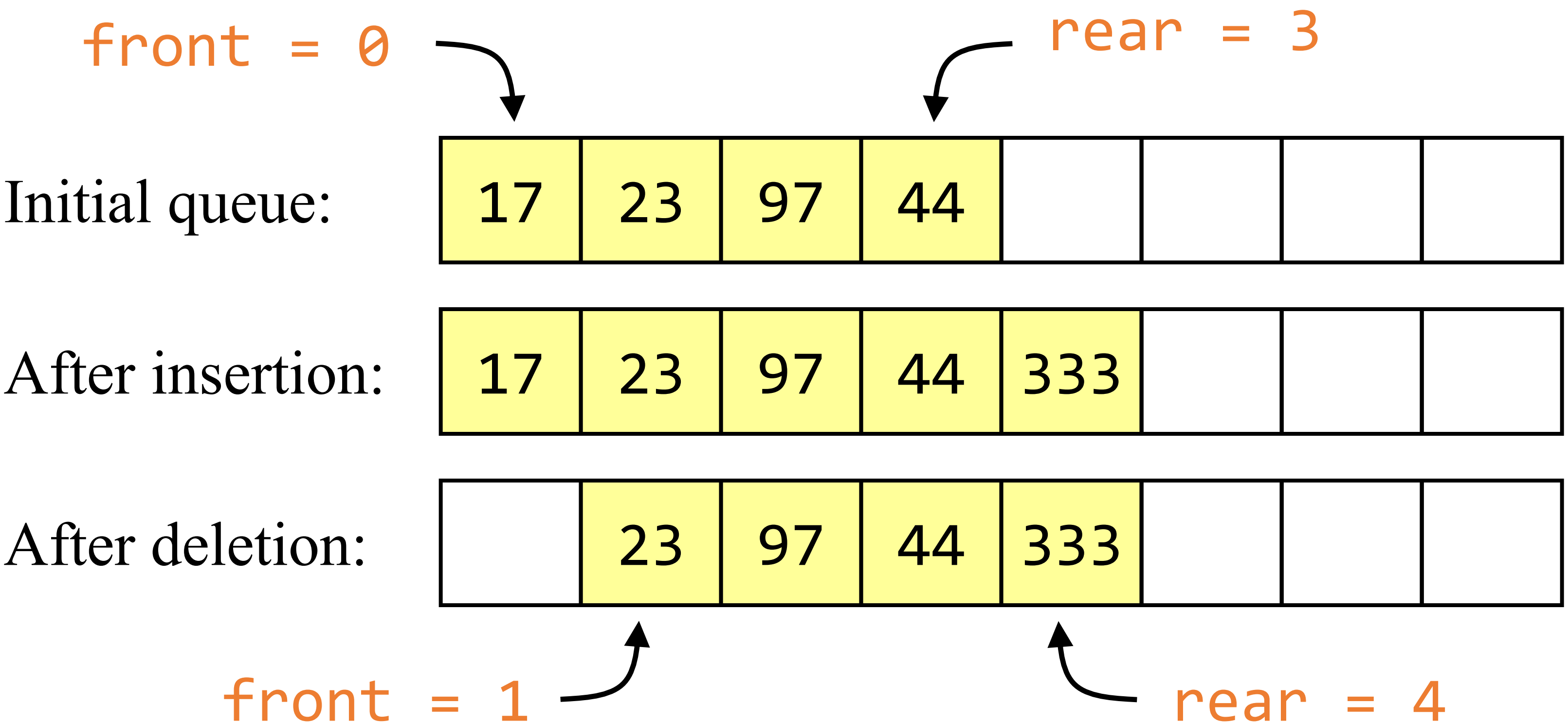
Array implementation of queues

- 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

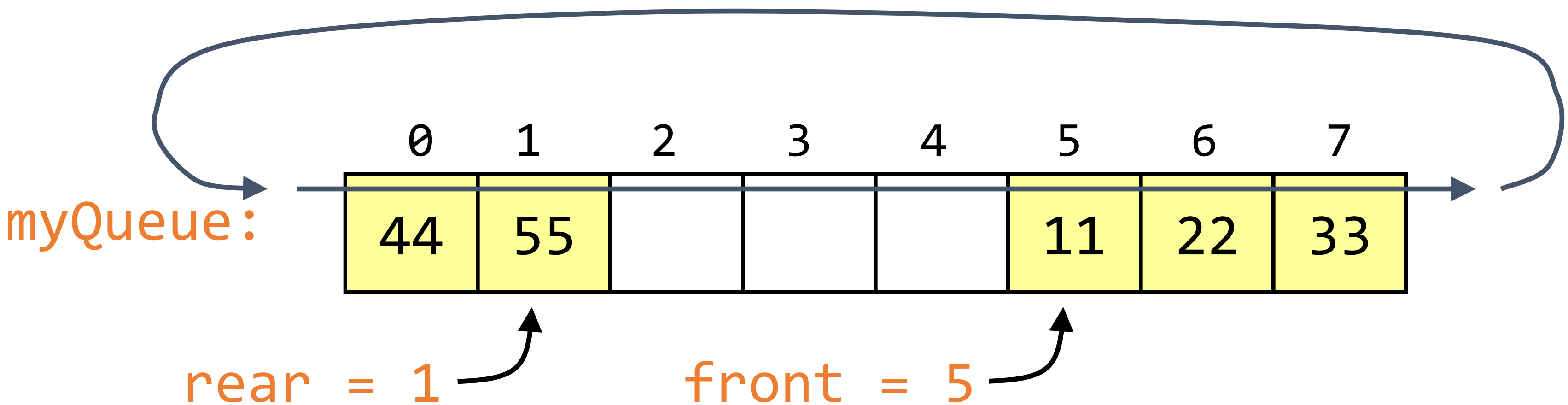
Array implementation of queues



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

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:

`myQueue:`

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

`rear = 4` `front = 5`

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

`myQueue:`

0	1	2	3	4	5	6	7

`rear = 4` `front = 5`

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`

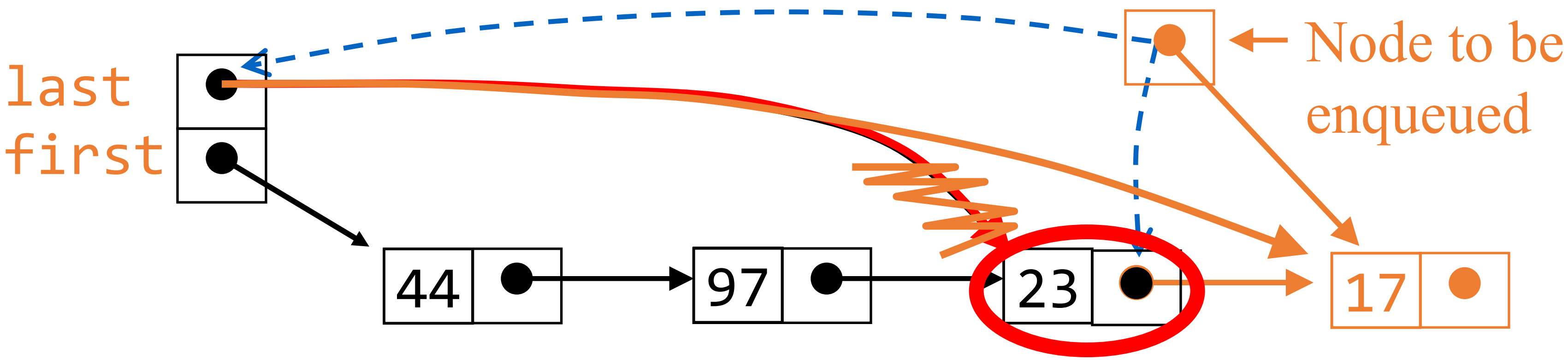
Linked-list implementation of queues

- 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



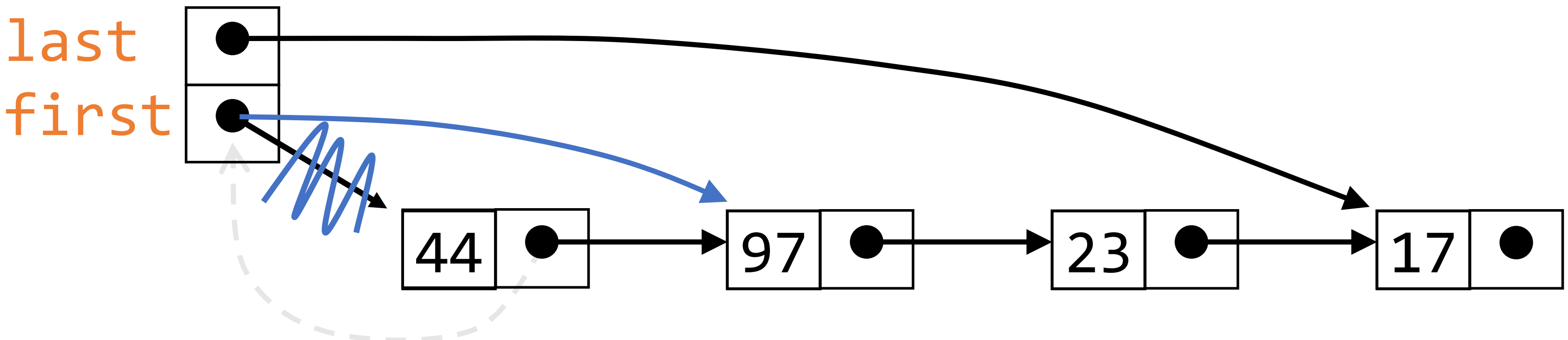
To enqueue (add) a node:

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

java.util `Interface Queue<E>`

- Java provides a queue *interface* and several implementations
- `boolean add(E e)`
  - Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
- `E element()`
  - Retrieves, but does not remove, the head of this queue.
- `boolean offer(E e)`
  - Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
- `E peek()`
  - Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- `E poll()`
  - Retrieves and removes the head of this queue, or returns null if this queue is empty.
- `E remove()`
  - Retrieves and removes the head of this queue.

Source: Java 6 API

java.util `Interface Deque<E>`

- Java 6 now has a `Deque` interface
- There are 12 methods:
  - Add, remove, or examine an element...
  - ...at the head or the tail of the queue...
  - ...and either throw an exception, or return a special value (`null` or `false`) if the operation fails

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Source: Java 6 API