CUS 1126: Introduction to Data Structures

Lecture : Stacks and Queues

Instructor: Nikhil Yadav

BUS STOP
6
9
13
15
60
96
291
294
297

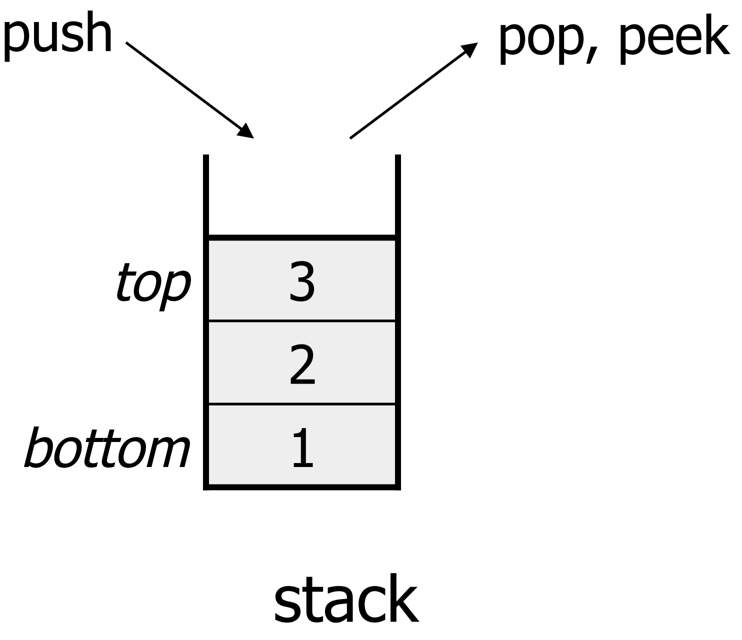
Stacks, Queues, and Dequeus

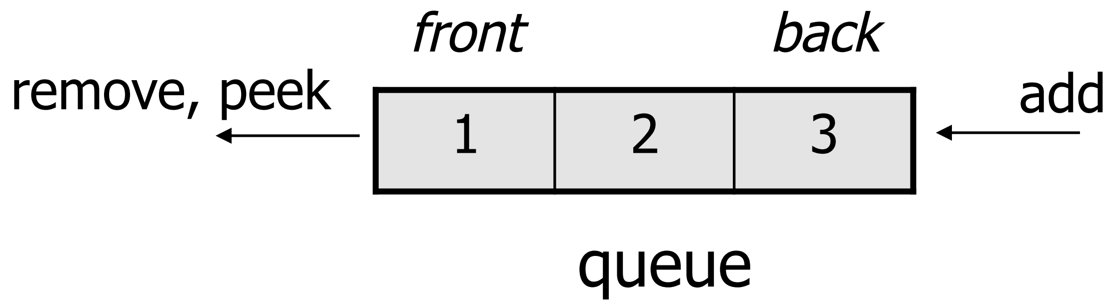
- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted

Stacks, Queues, and Dequeus

- 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

Stacks, Queues, and Dequeus





Stacks, Queues, and Dequeus

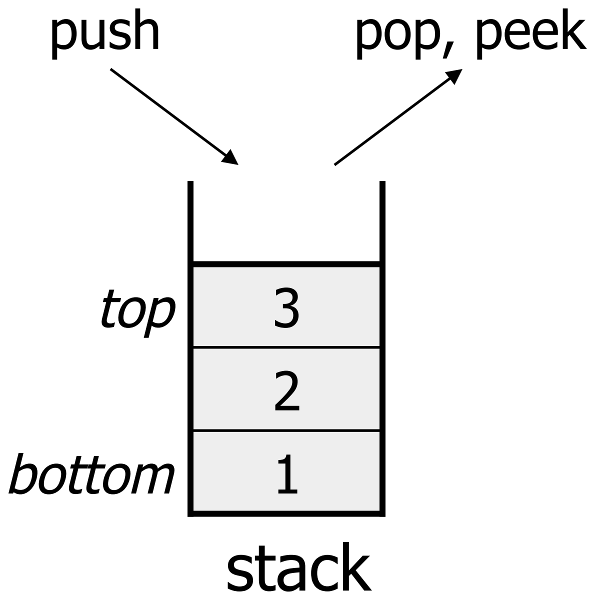
- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- A dequeu is a double-ended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

Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").

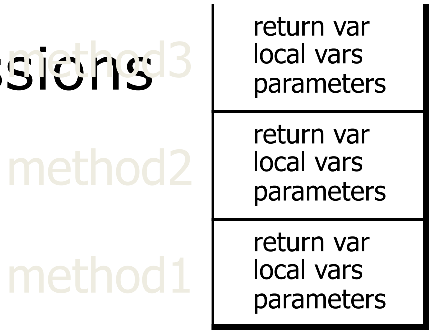
Stacks

- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



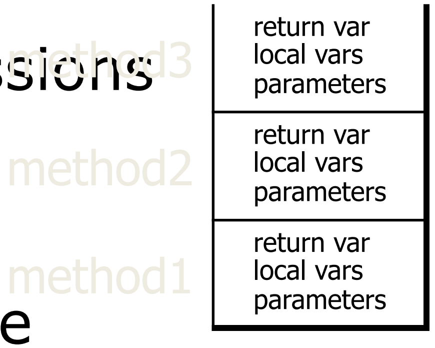
Stacks in computer science

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions



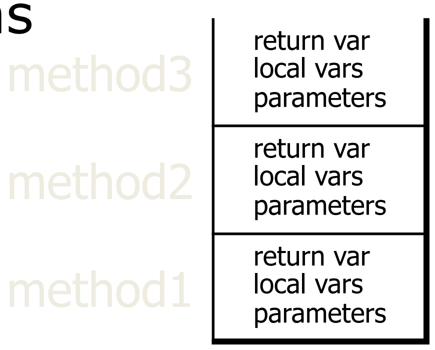
Stacks in computer science

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces `{ }` match
  - convert "infix" expressions to pre/postfix



Stacks in computer science

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions



- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

Class `Stack`

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<String> s = new Stack<String>();
s.push("a");
s.push("b");
s.push("c");          // bottom ["a", "b", "c"] top
System.out.println(s.pop()); // "c"
```

– `Stack` has other methods that are off-limits (not efficient)

Array implementation of stacks

- To implement a stack, items are inserted and removed at the same end (called the top)

Array implementation of stacks

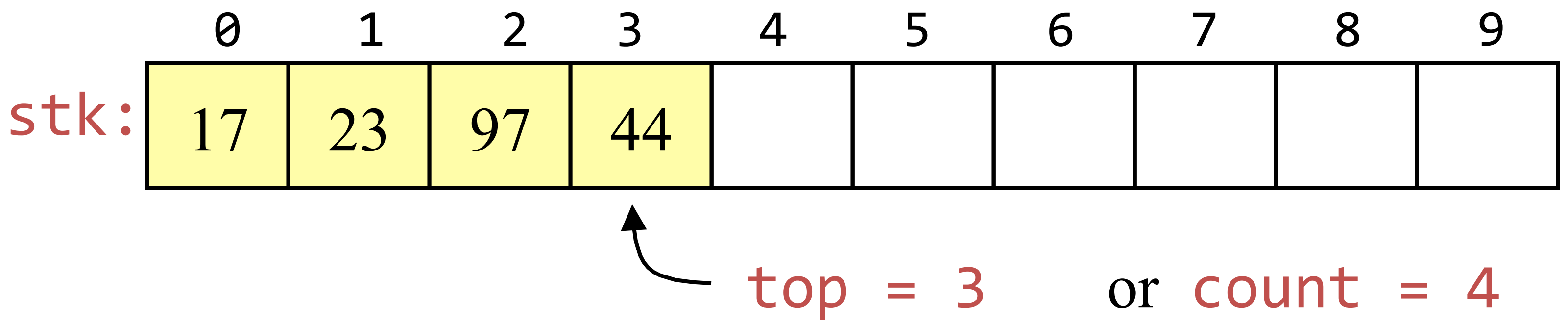
- To implement a stack, items are inserted and removed at the same end (called the top)

- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end

Array implementation of stacks

- 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

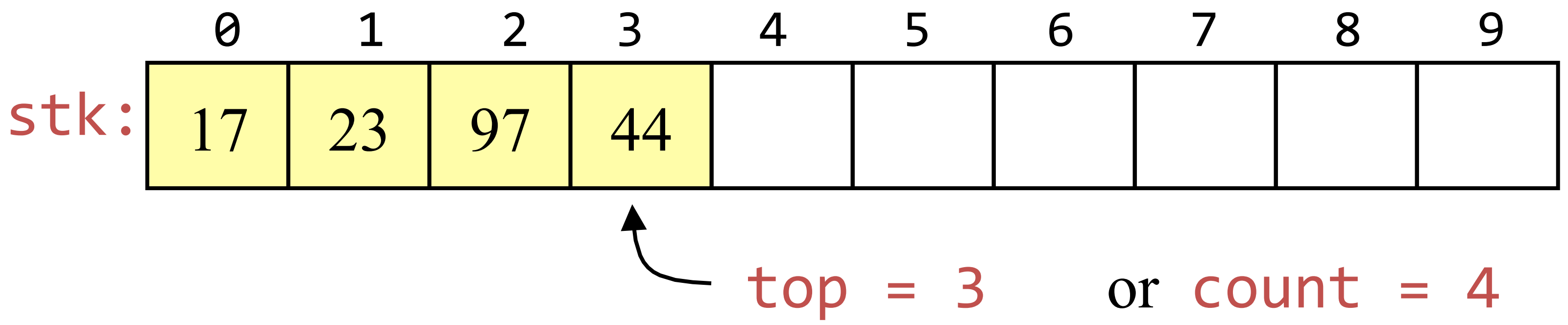
Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by

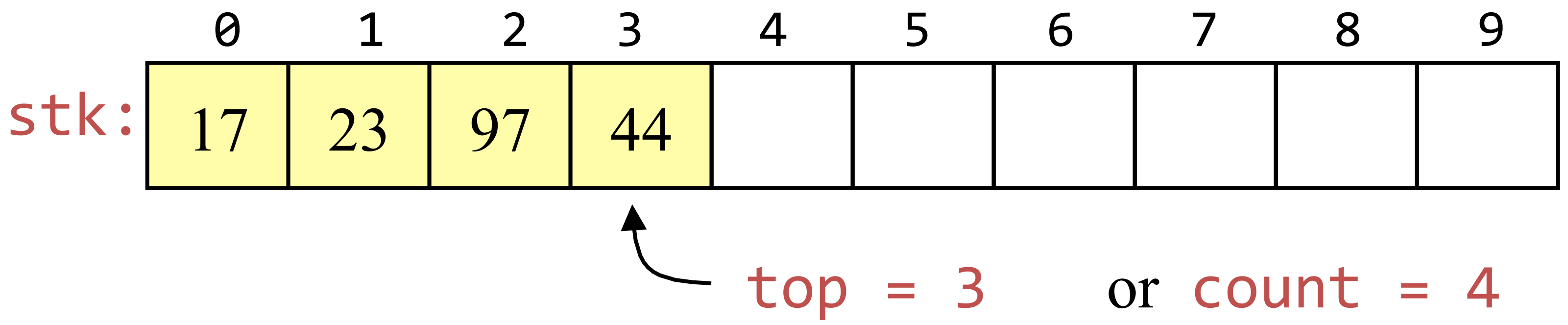
  `top = -1` or `count = 0`

Pushing and popping



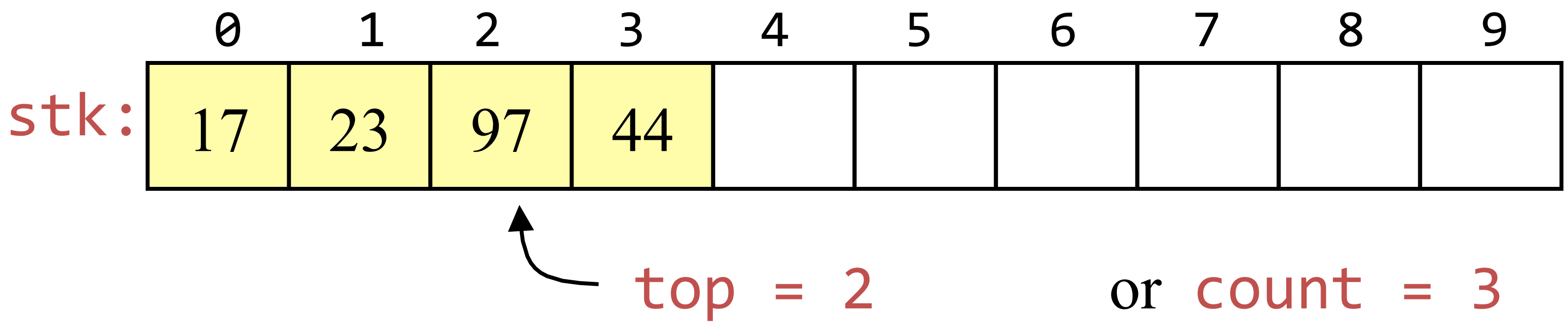
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`

Pushing and popping



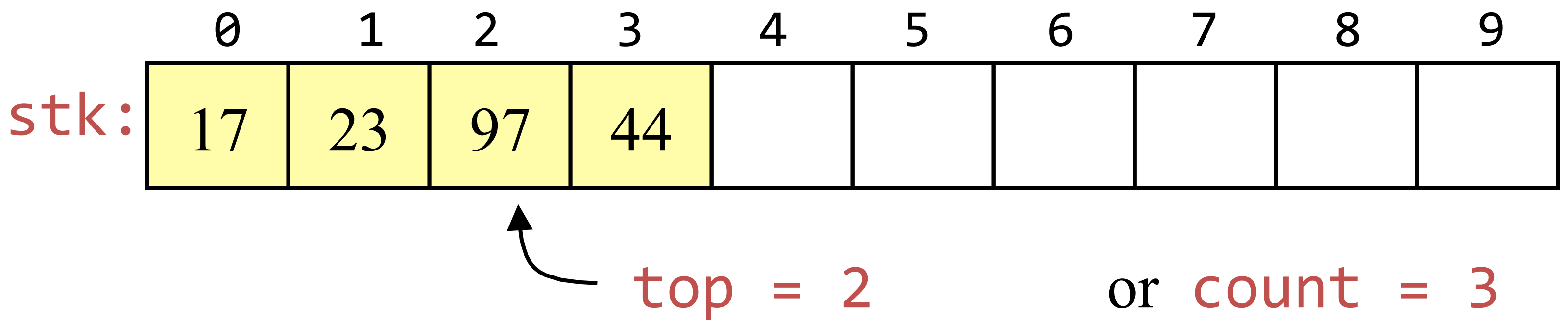
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

After popping



- When you pop an element, do you just leave the "deleted" element sitting in the array?

After popping

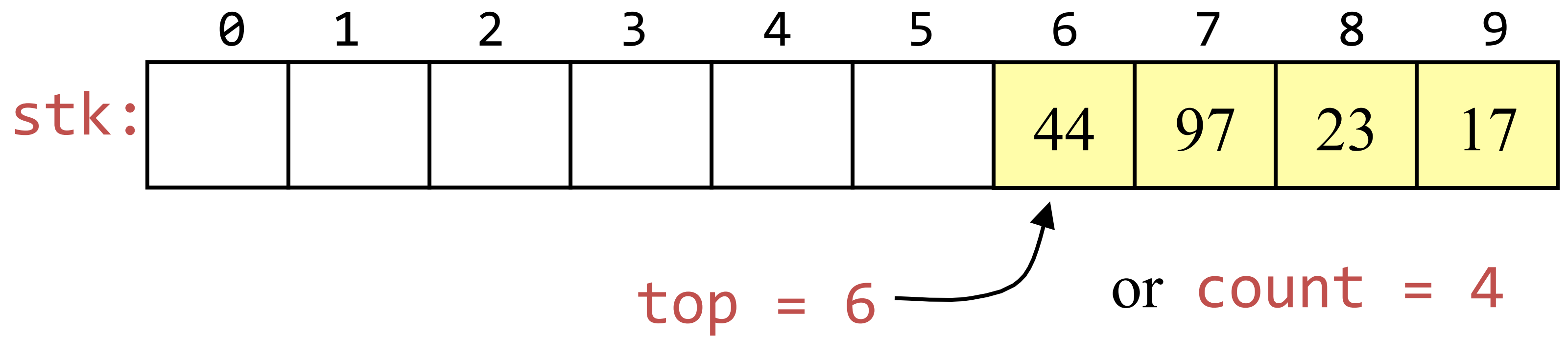


- When you pop an element, do you just leave the "deleted" element sitting in the array?

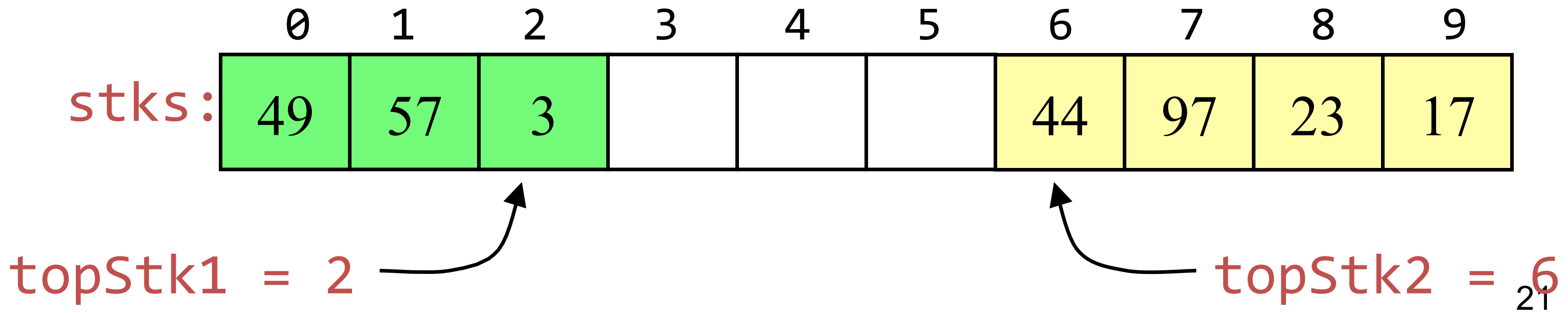
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to `null`
  - Why? To allow it to be garbage collected!

Sharing space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*



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

Error checking

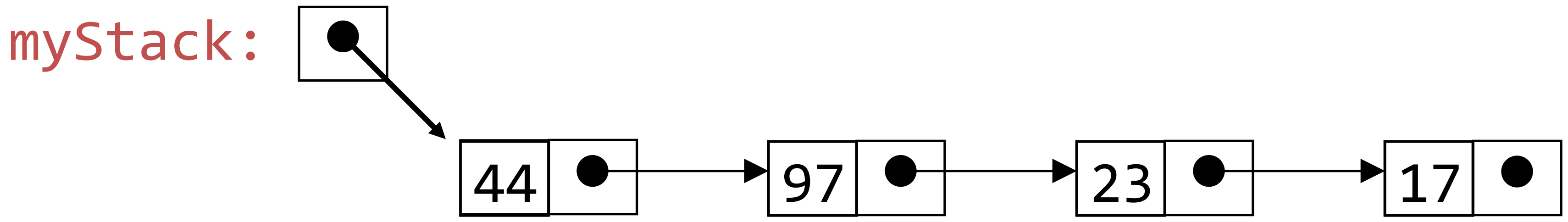
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
  - You could create your own, more informative exception

Error checking

- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

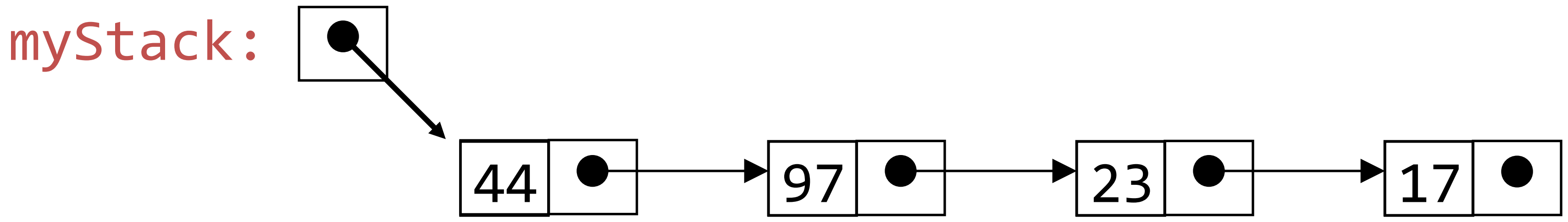
Linked-list implementation of stacks

- 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



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)

Linked-list implementation details

- Underflow can happen, and should be handled the same way as for an array implementation

Linked-list implementation details

- 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

Stack limitations/idioms

- You cannot loop over a stack in the usual way.

```
Stack<Integer> s = new
Stack<Integer>();
...
for (int i = 0; i < s.size(); i++)
{
    do something with s.get(i);
}
```

Stack limitations/idioms

- Instead, you pull elements out of the stack one at a time.
  - common idiom: Pop each element until the stack is empty.

```
// process (and destroy) an entire
stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

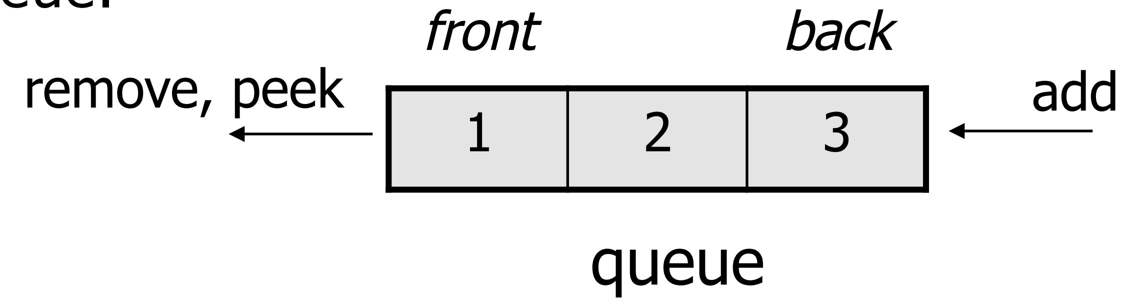
Stack Implementation

```
class Stack {
      Node top; //maintain top of stack
      Node pop() {
            if (top != null) { //if list is not empty
                  Node item = top;
                   top = top.next;
                   return item;
             }
            return null;
      }

      void push(Node item) {
            Node t = new Node(item);
             t.next = top;
             top = t;
      }
 }
```

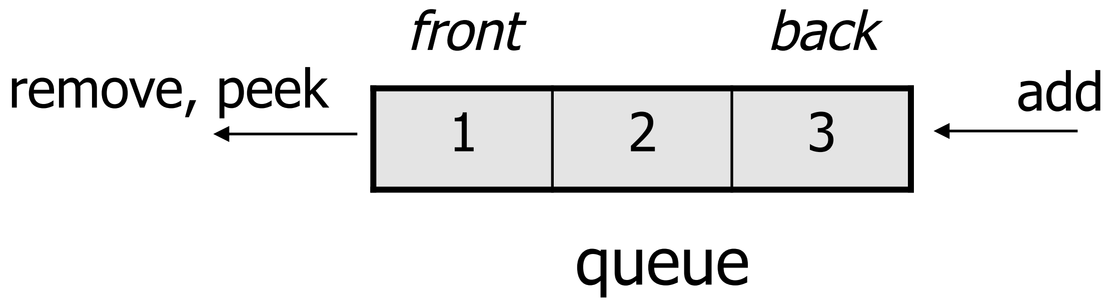
Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.

Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send

- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Programming with Queues

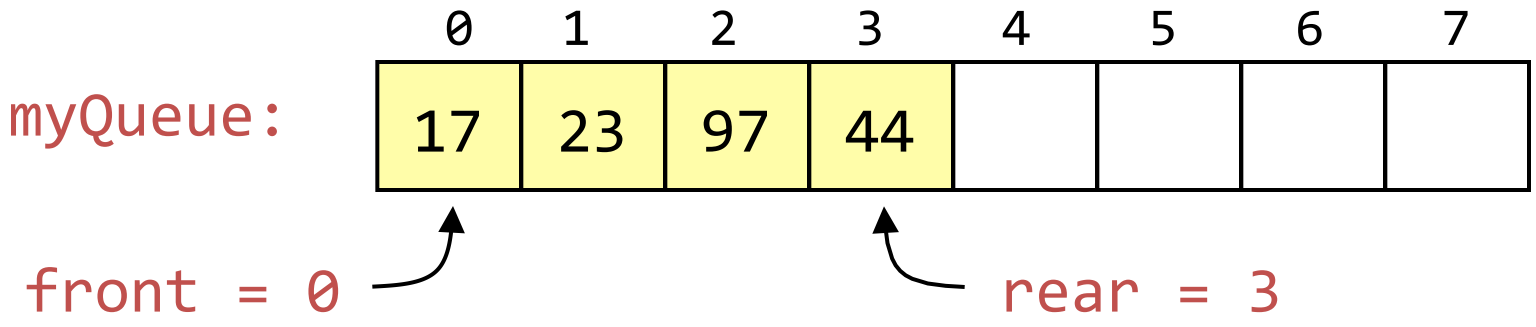
`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);           // front [42, -3, 17] back
System.out.println(q.remove());    // 42
```

- **IMPORTANT**: When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object.
  - This has to do with a topic we'll discuss later called *interfaces*.

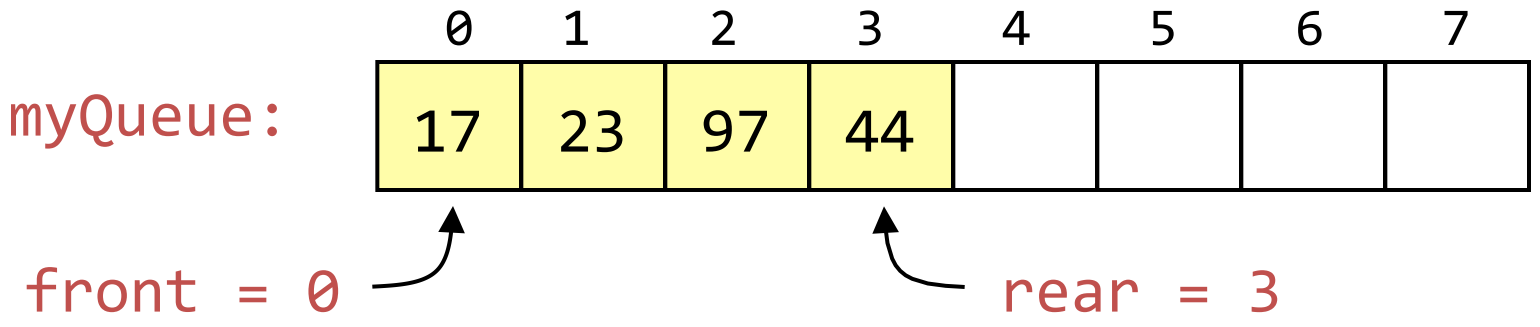
Array implementation of queues

- 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)



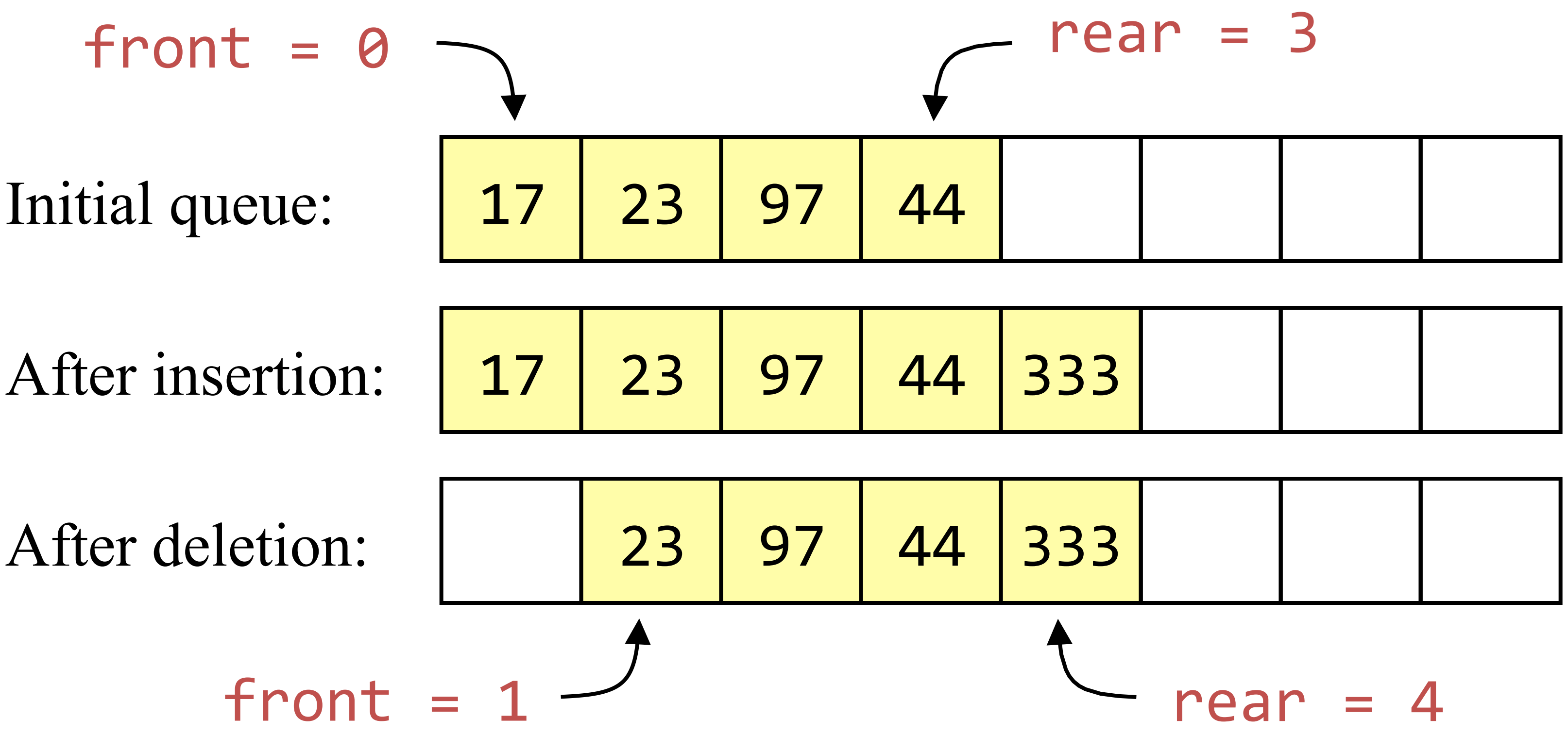
Array implementation of queues

- 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

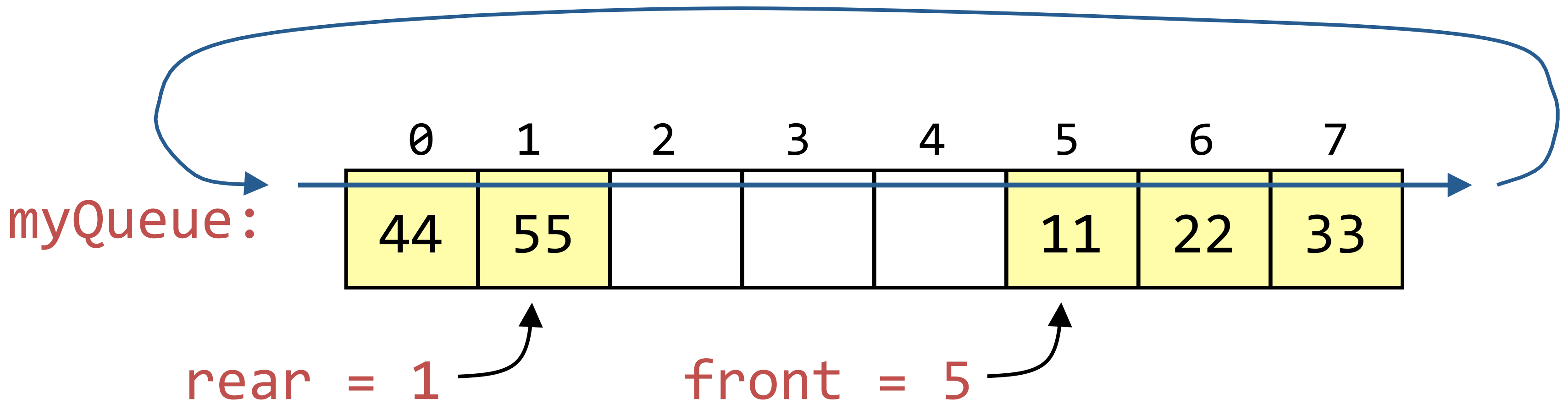
Array implementation of queues



- 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

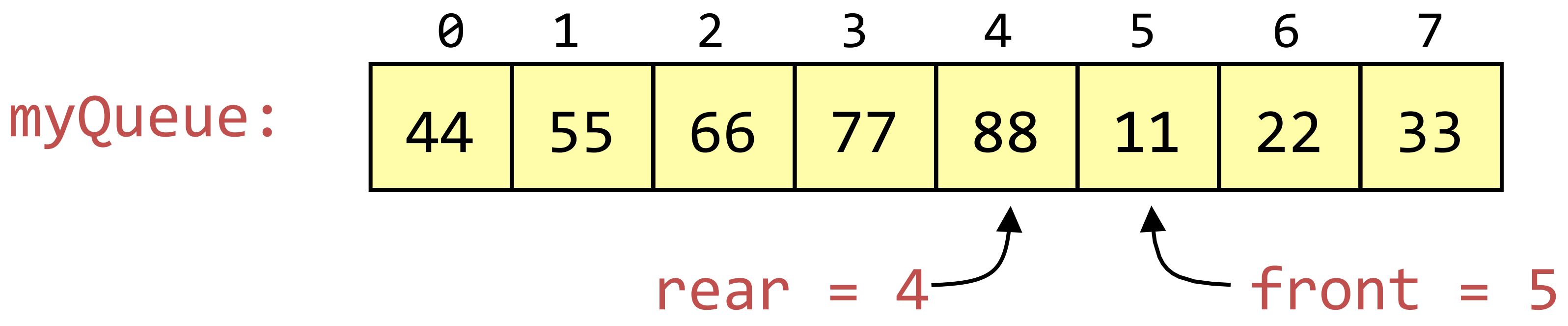
- We can treat the array holding the queue elements as circular (joined at the ends)



- 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;`

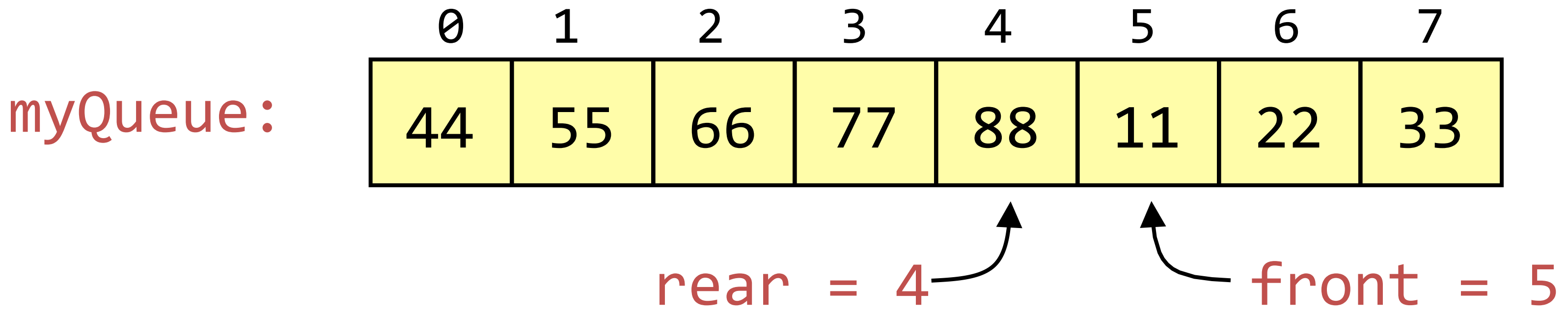
Full and empty queues

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

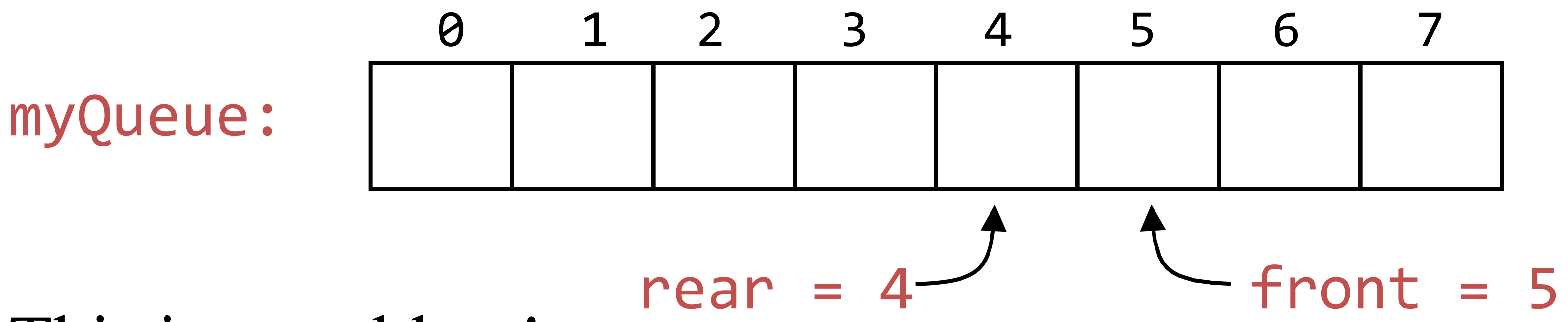


Full and empty queues

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



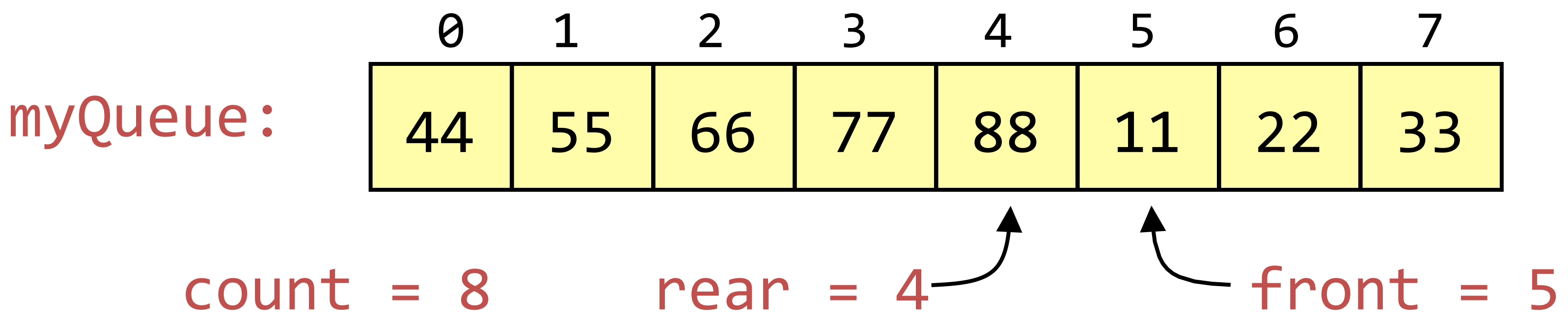
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



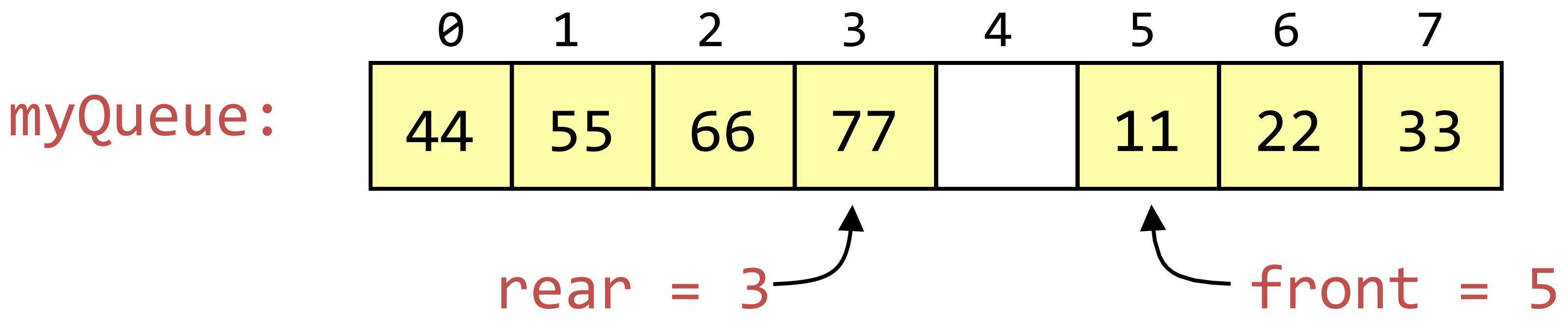
This is a problem!

Full and empty queues: solutions

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



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



Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end

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

Linked-list implementation of queues

- 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

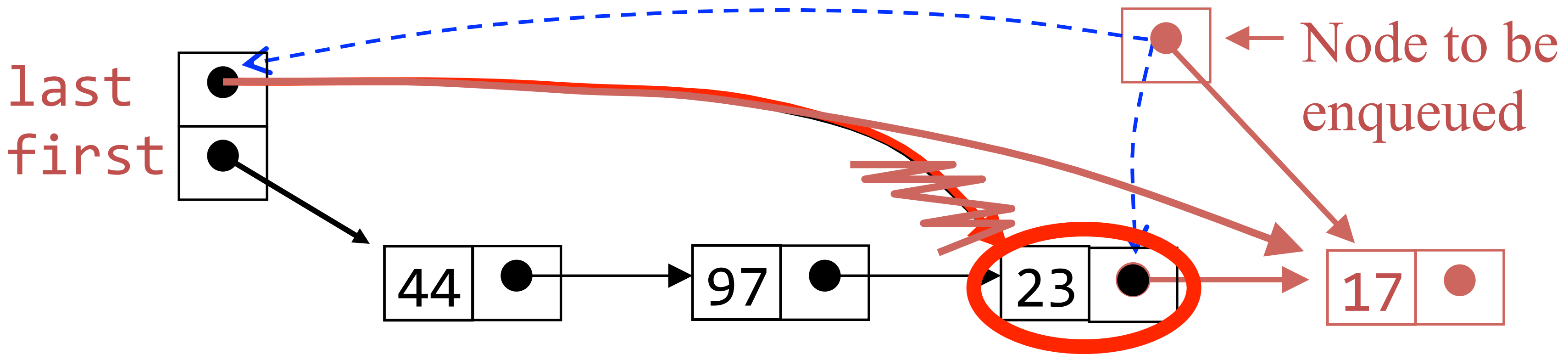
SLL implementation of queues

- 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

SLL implementation of queues

- 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

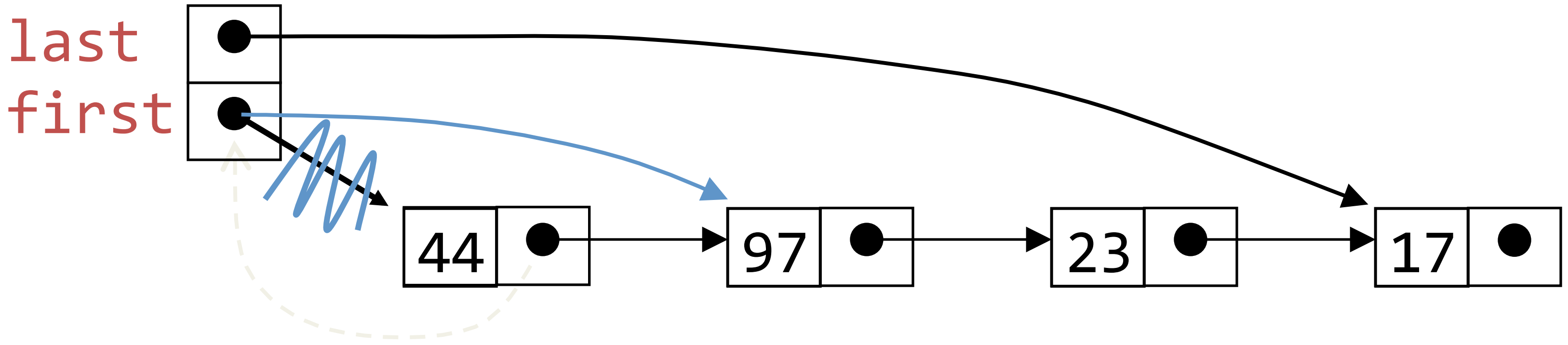
Enqueueing a node



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

```
class Queue {
        Node first=null; Node last=null;
        void enqueue(Node item) {
            item = new Node();
            if (first==null) {
                    back = item;
                    first = back;
            }
            else {
                    back.next = item;
                    back = back.next;
            }
        }
        Node dequeue(Node item) {
            if (first != null) {
                 Node item = front;
                 first = first.next;
                 return item;
            }
            return null;
        }
}
```

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.Stack`

- The `Stack` ADT, as provided in `java.util.Stack`:
  - `Stack()`: the constructor
  - `boolean empty()` (but also inherits `isEmpty()`)
  - `Object push(Object item)`
  - `Object peek()`
  - `Object pop()`
  - `int search(Object o):` Returns the 1-based position of the object on this stack

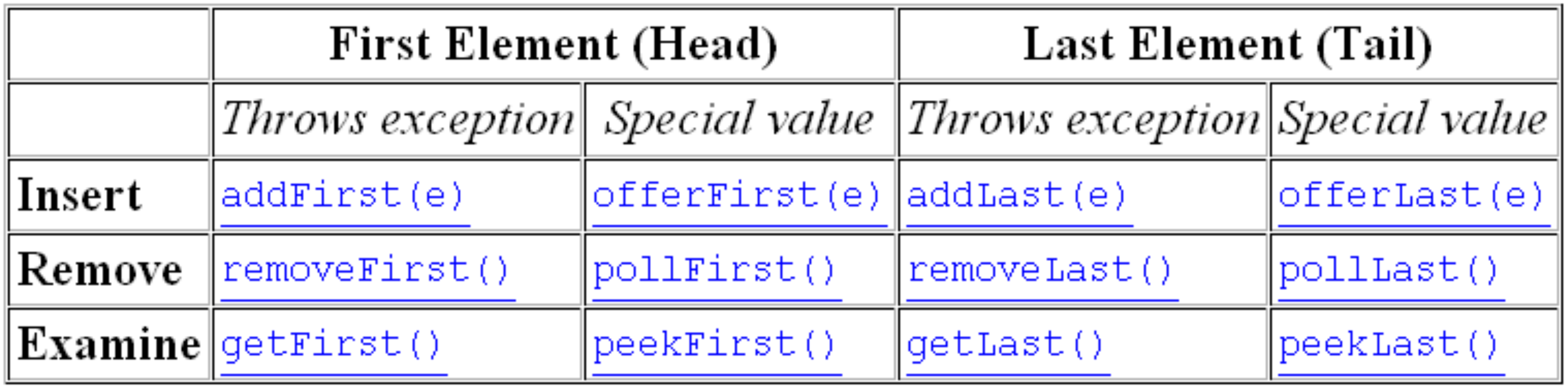
java.util `Interface Queue<E>`

- 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

Thank you