Stacks, Queues, and Deques

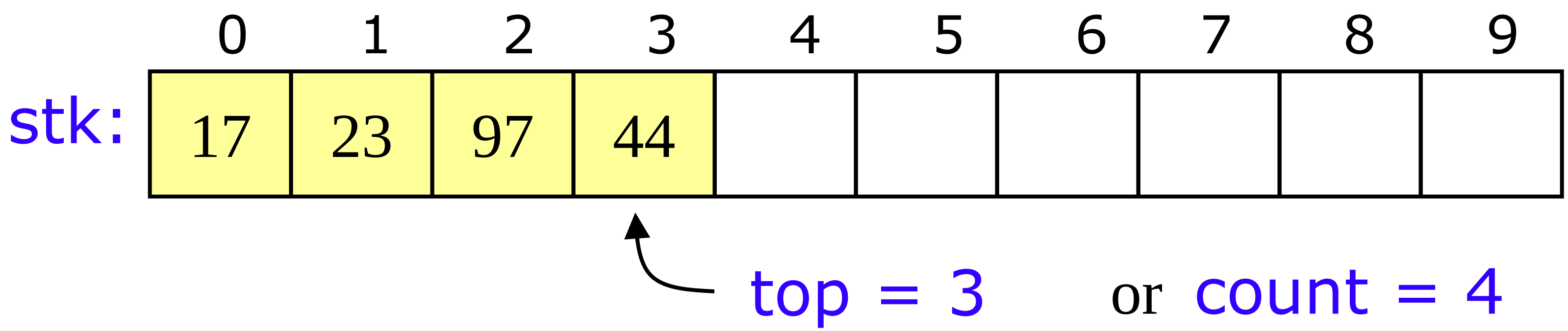
Stacks, Queues, and Deques

- 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

Array implementation of stacks

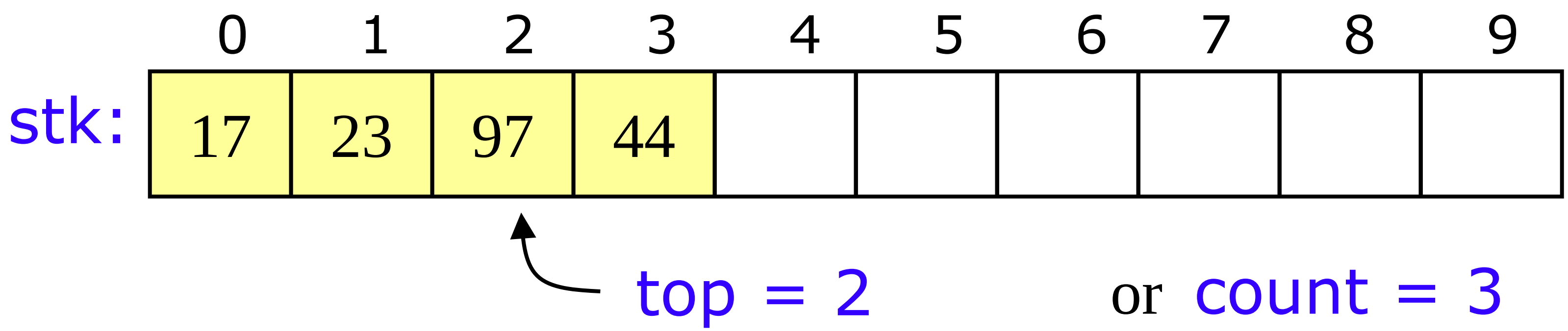
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - 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]

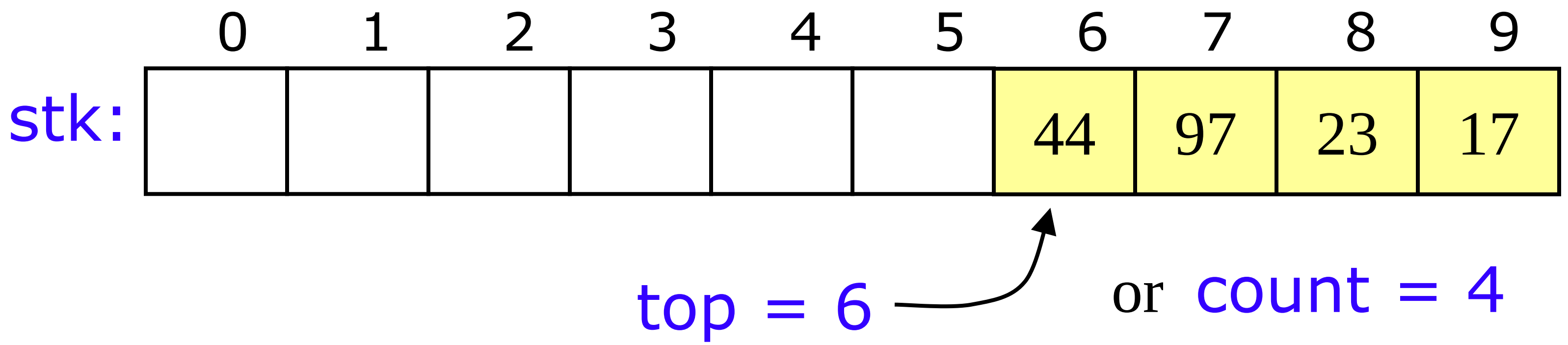
After popping



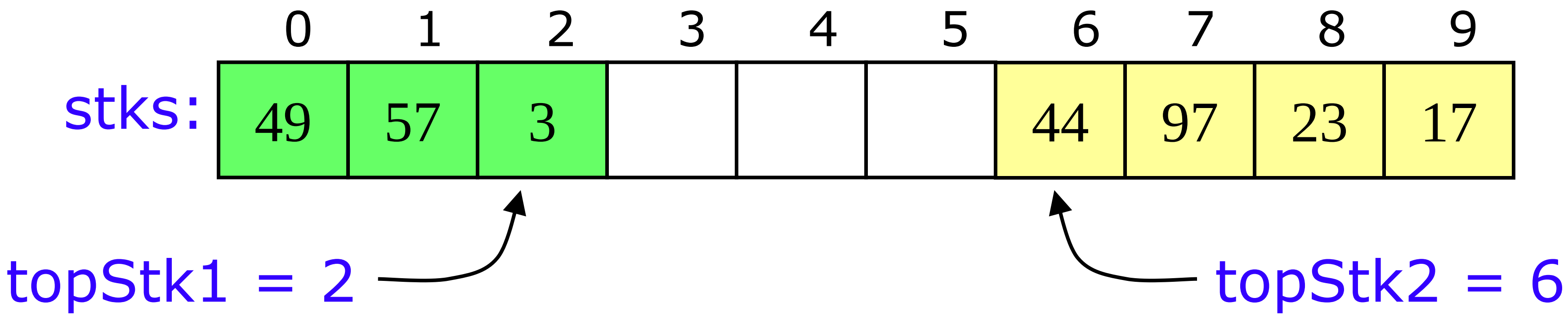
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and 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
- 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

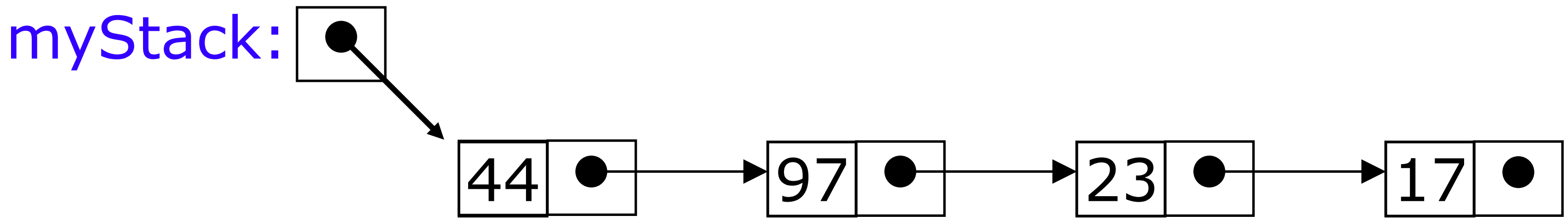
Pointers and references

- In C and C++ we have “pointers,” while in Java we have “references”
  - These are essentially the same thing
    - The difference is that C and C++ allow you to modify pointers in arbitrary ways, and to point to anything
  - In Java, a reference is more of a “black box,” or ADT
    - Available operations are:
      - dereference (“follow”)
      - copy
      - compare for equality
    - There are constraints on what kind of thing is referenced: for example, a reference to an `array of int` can *only* refer to an `array of int`

Creating references

- The keyword `new` creates a new object, but also returns a *reference* to that object
- For example, `Person p = new Person("John")`
  - `new Person("John")` creates the object and returns a reference to it
  - We can assign this reference to `p`, or use it in other ways

Creating links in Java

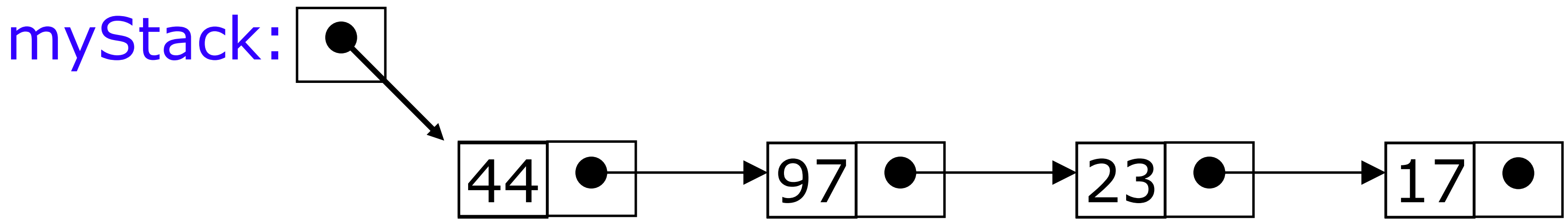



```
class Cell { int value; Cell next;
    Cell (int v, Cell n) { value = v; next = n; }
}

Cell temp = new Cell(17, null);
temp = new Cell(23, temp);
temp = new Cell(97, temp);
Cell myStack = new Cell(44, temp);
```

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



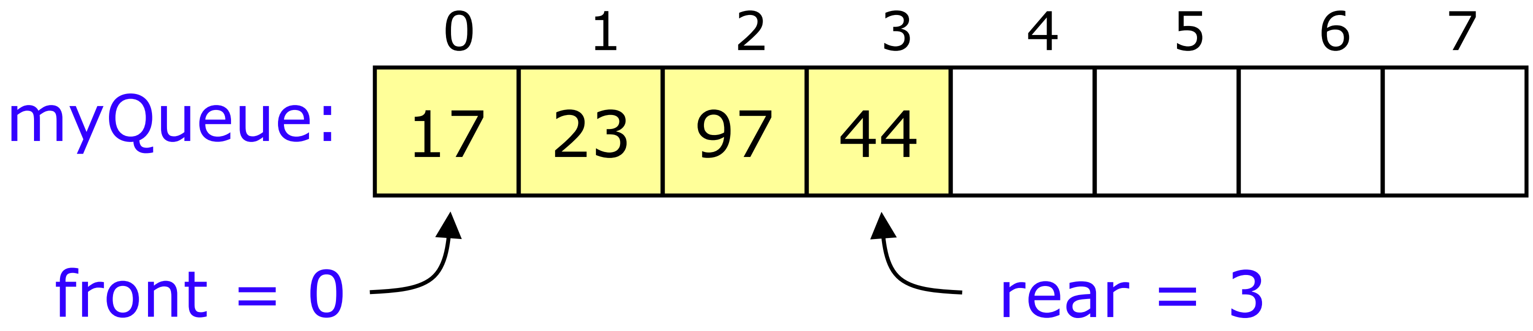
- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

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

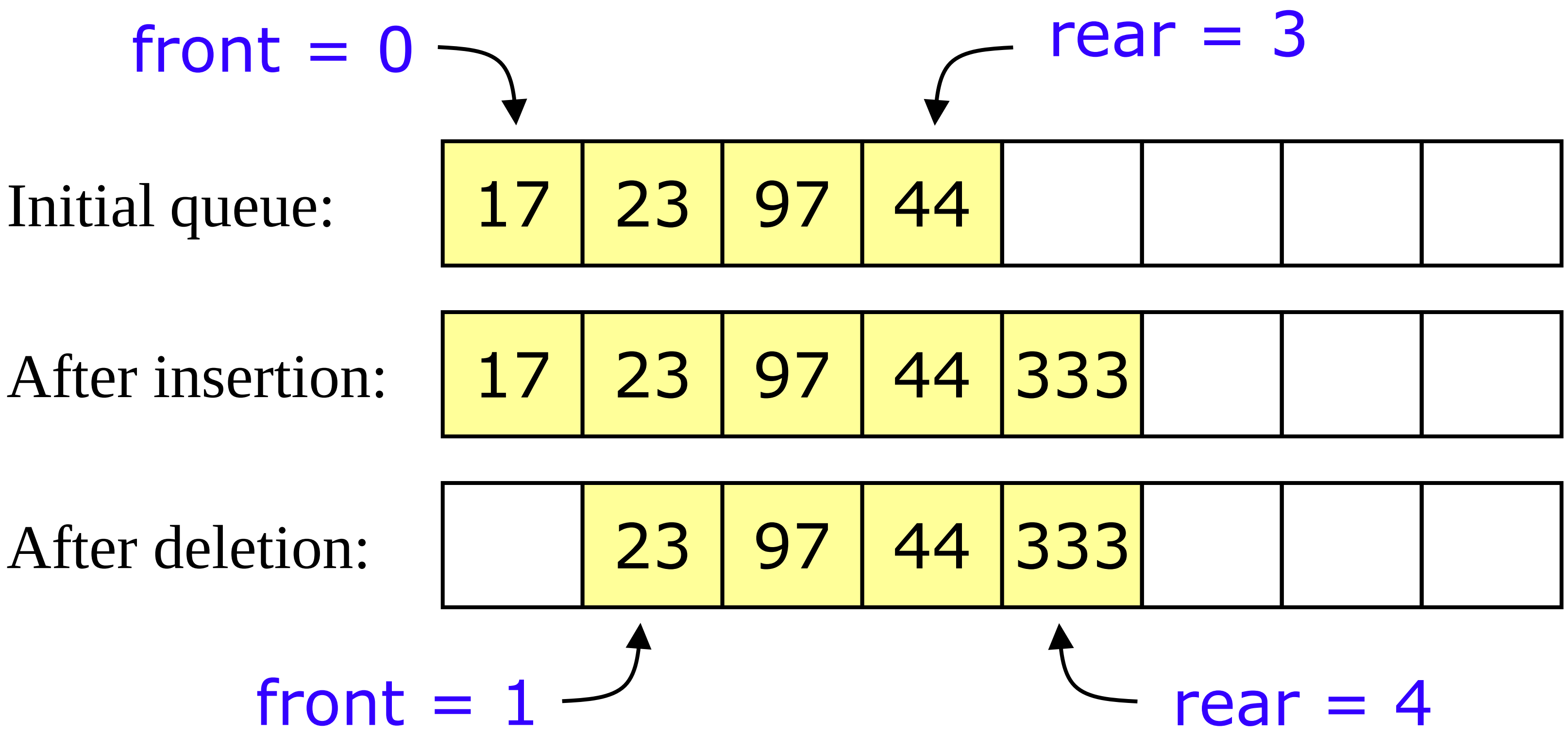
Array implementation of queues

- 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

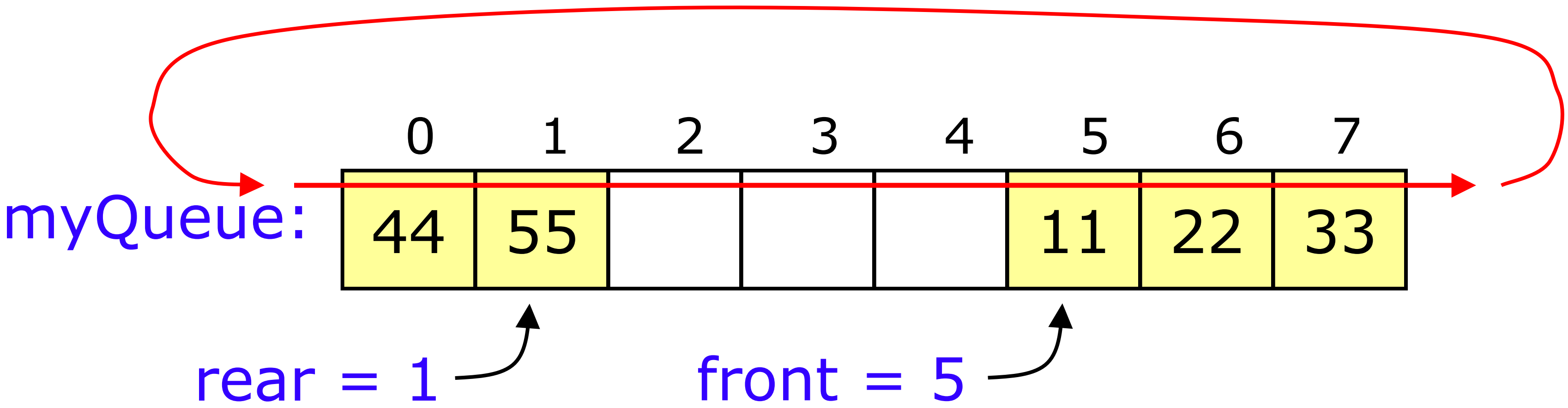
Array implementation of queues



- 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

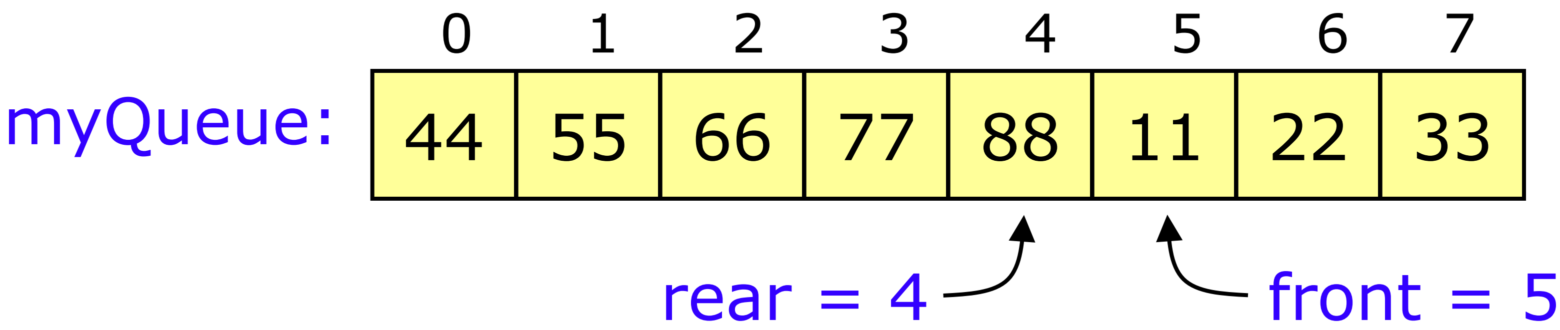
- We can treat the array holding the queue elements as circular (joined at the ends)



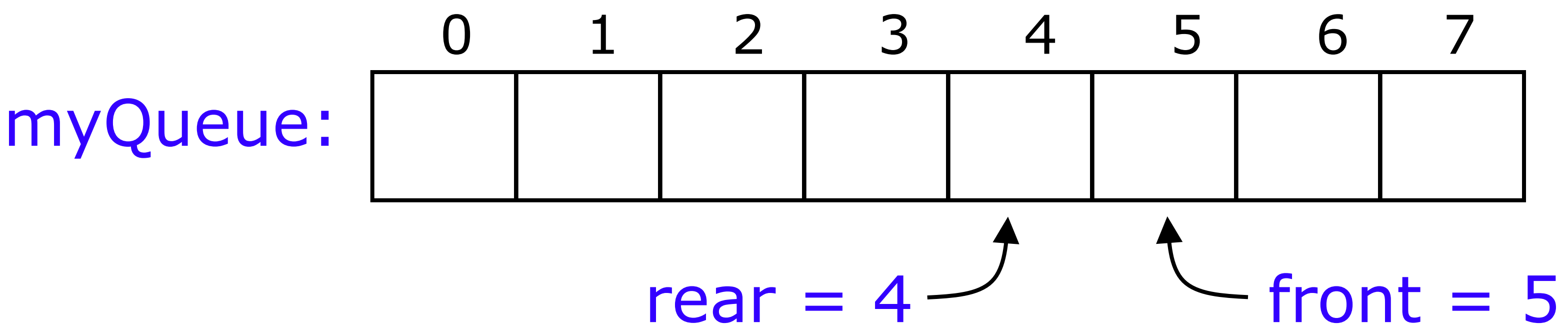
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: front = (front + 1) % myQueue.length;
  and: rear = (rear + 1) % myQueue.length;

Full and empty queues

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



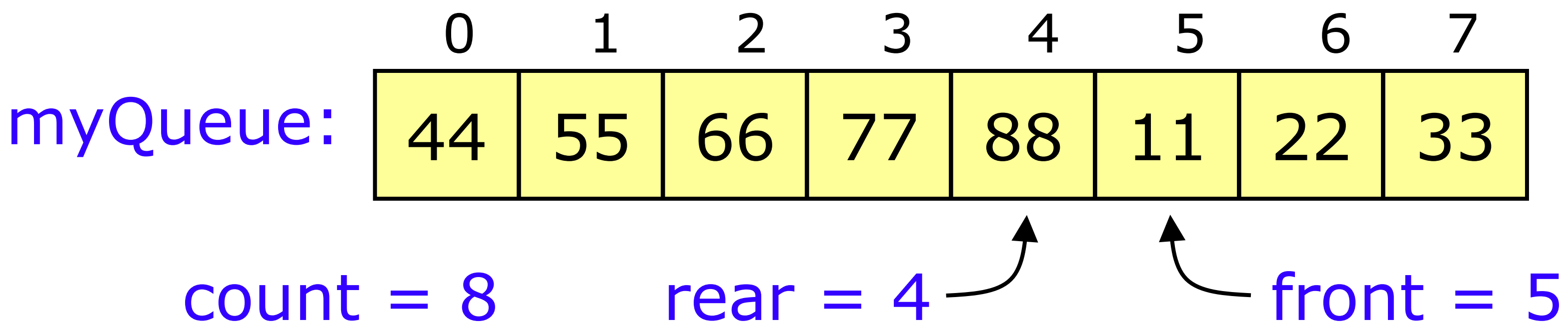
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



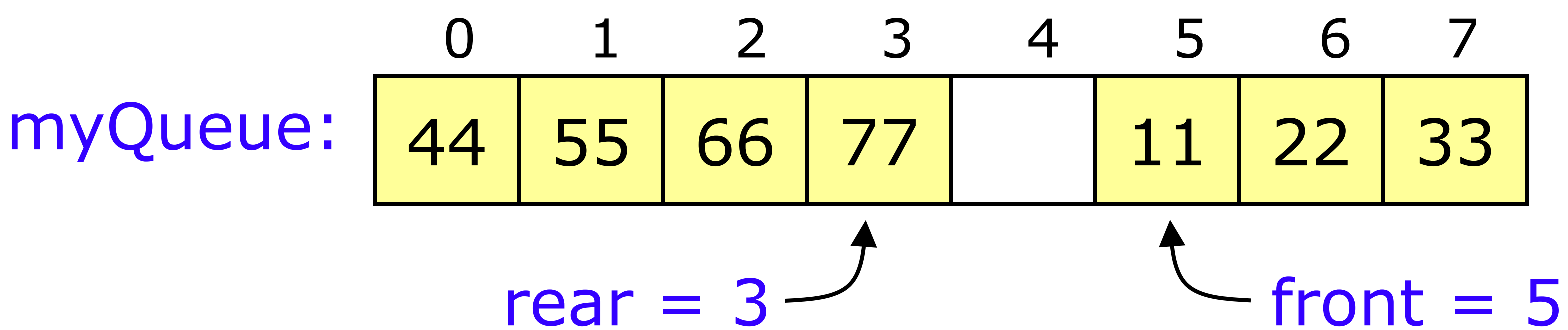
This is a problem!

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



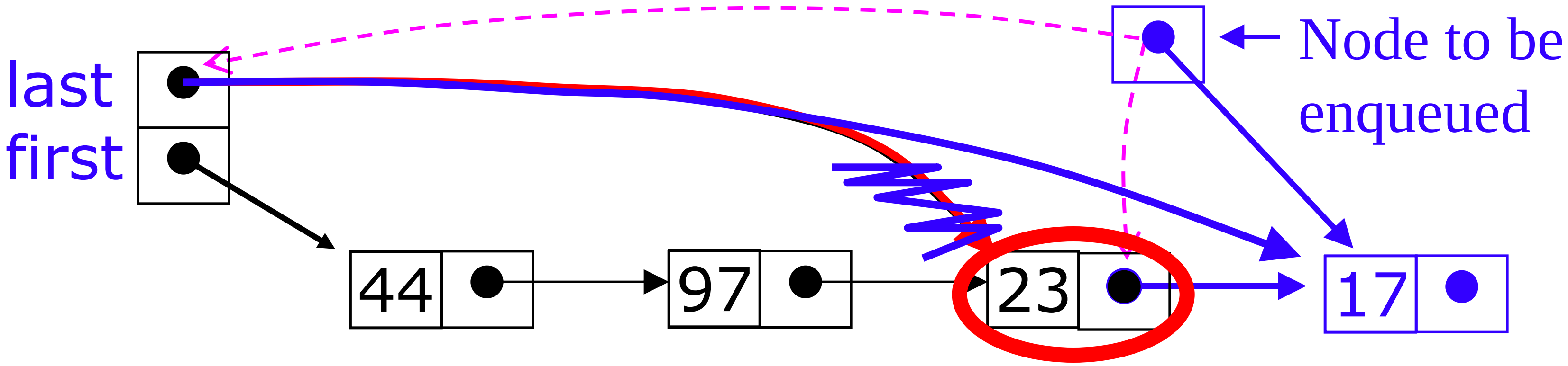
Linked-list implementation of queues

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



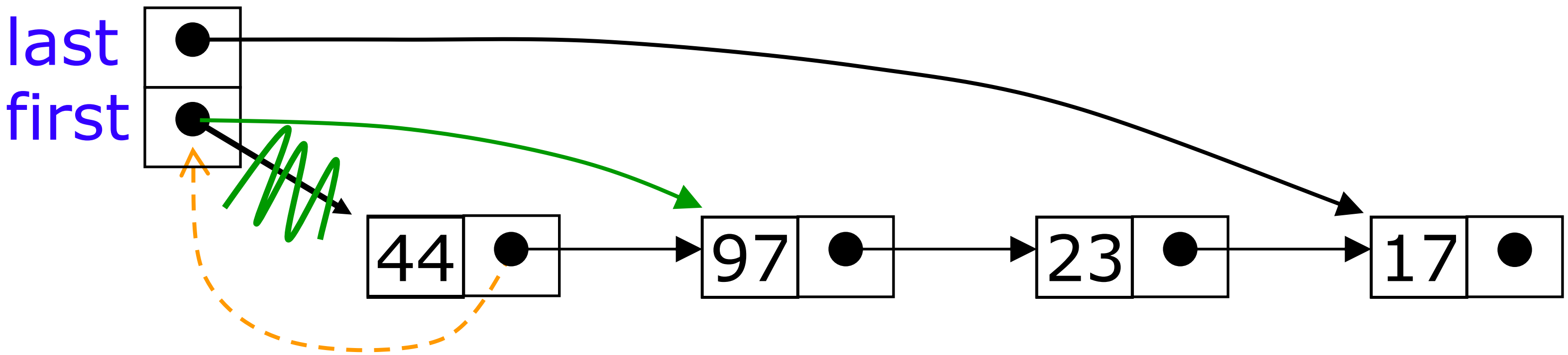
To enqueue (add) a node:

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

Stack ADT

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

A queue ADT

- Java does *not* provide a queue class
- Here is a *possible* queue ADT:
  - `Queue()`: the constructor
  - `boolean empty()`
  - `Object enqueue(Object item)`: add at element at the rear
  - `Object dequeue()`: remove an element from the front
  - `Object peek()`: look at the front element
  - `int search(Object o)`: Returns the 1-based position from the front of the queue

A deque ADT

- Java does *not* provide a deque class
- Here is a possible deque ADT:
  - Deque(): the constructor
  - boolean empty()
  - Object addAtFront(Object item)
  - Object addAtRear(Object item)
  - Object getFromFront()
  - Object getFromRear()
  - Object peekAtFront()
  - Object peekAtRear()
  - int search(Object o): Returns the 1-based position from the front of the deque

Using Vectors

- You could implement a deque with java.util.Vector:
  - addAtFront(Object) → insertElementAt(Object, 0)
  - addAtRear(Object item) → add(Object)
  - getFromFront() → remove(0)
  - getFromRear() → remove(size() – 1)
- Would this be a good implementation?
- Why or why not?

The End