Stacks, Queues, and Deques

Slides adapted from David Matuszek, UPENN

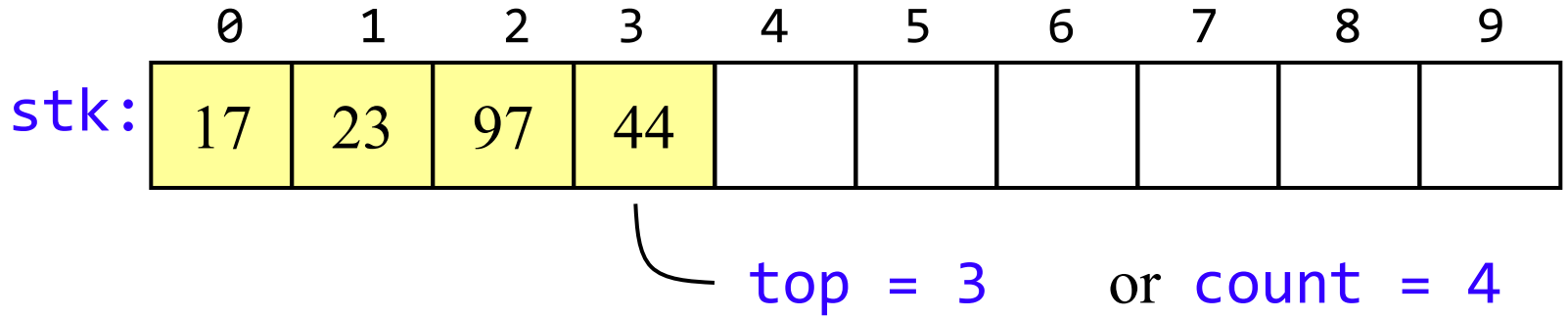
Stacks, Queues, and Deques

- 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

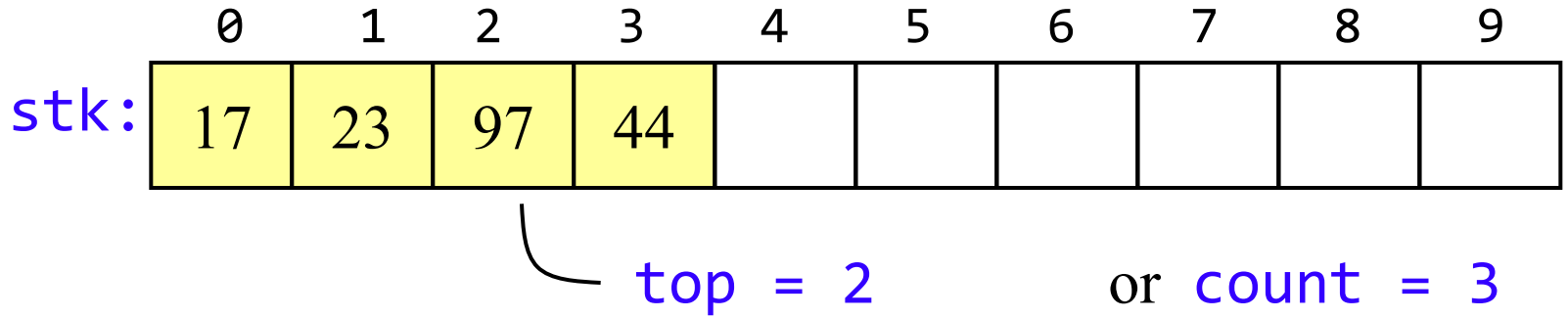
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
 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

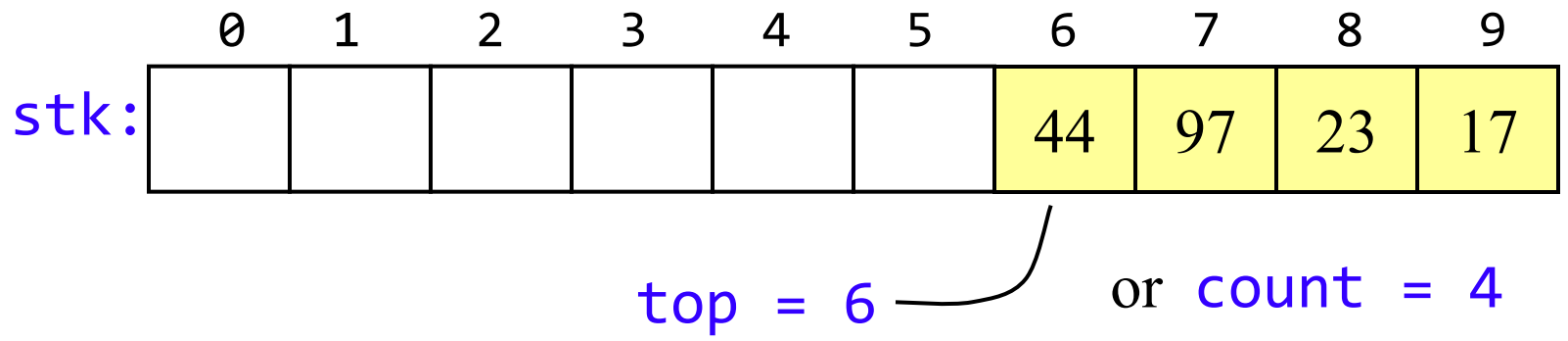
After popping



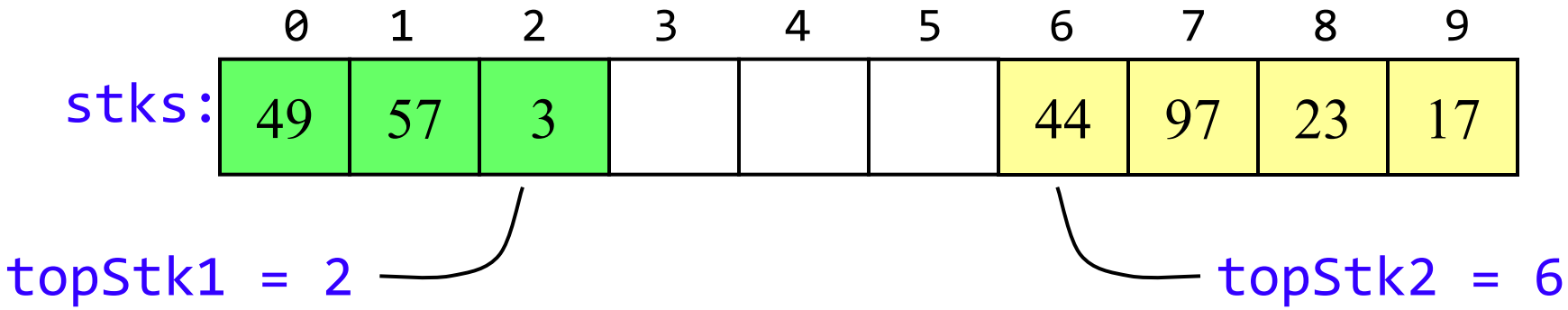
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and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

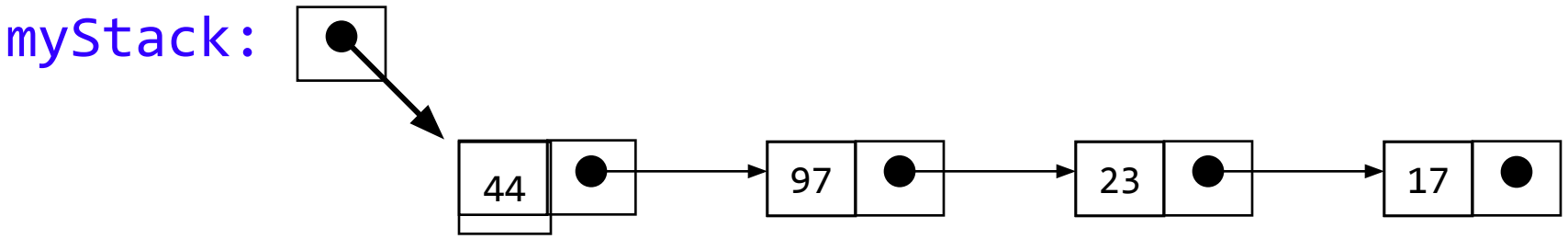


Error checking

- 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

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



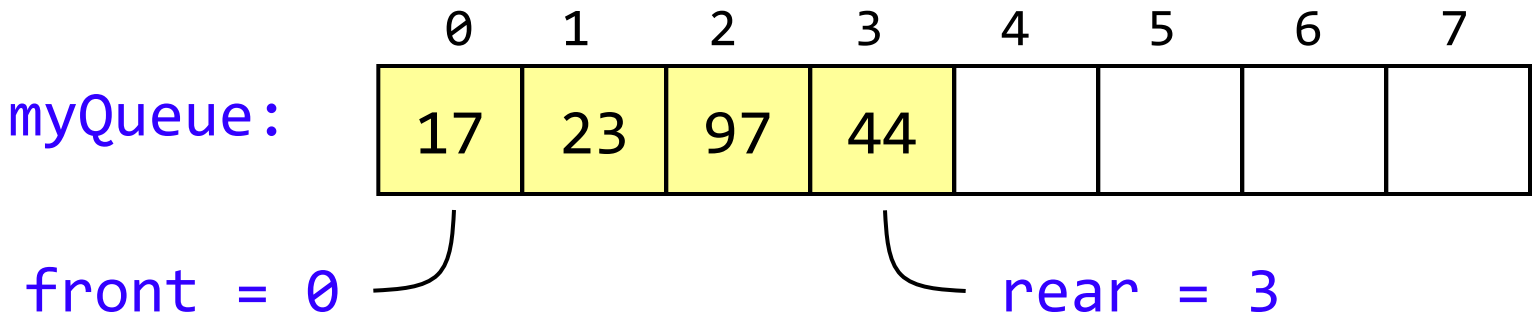
- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

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

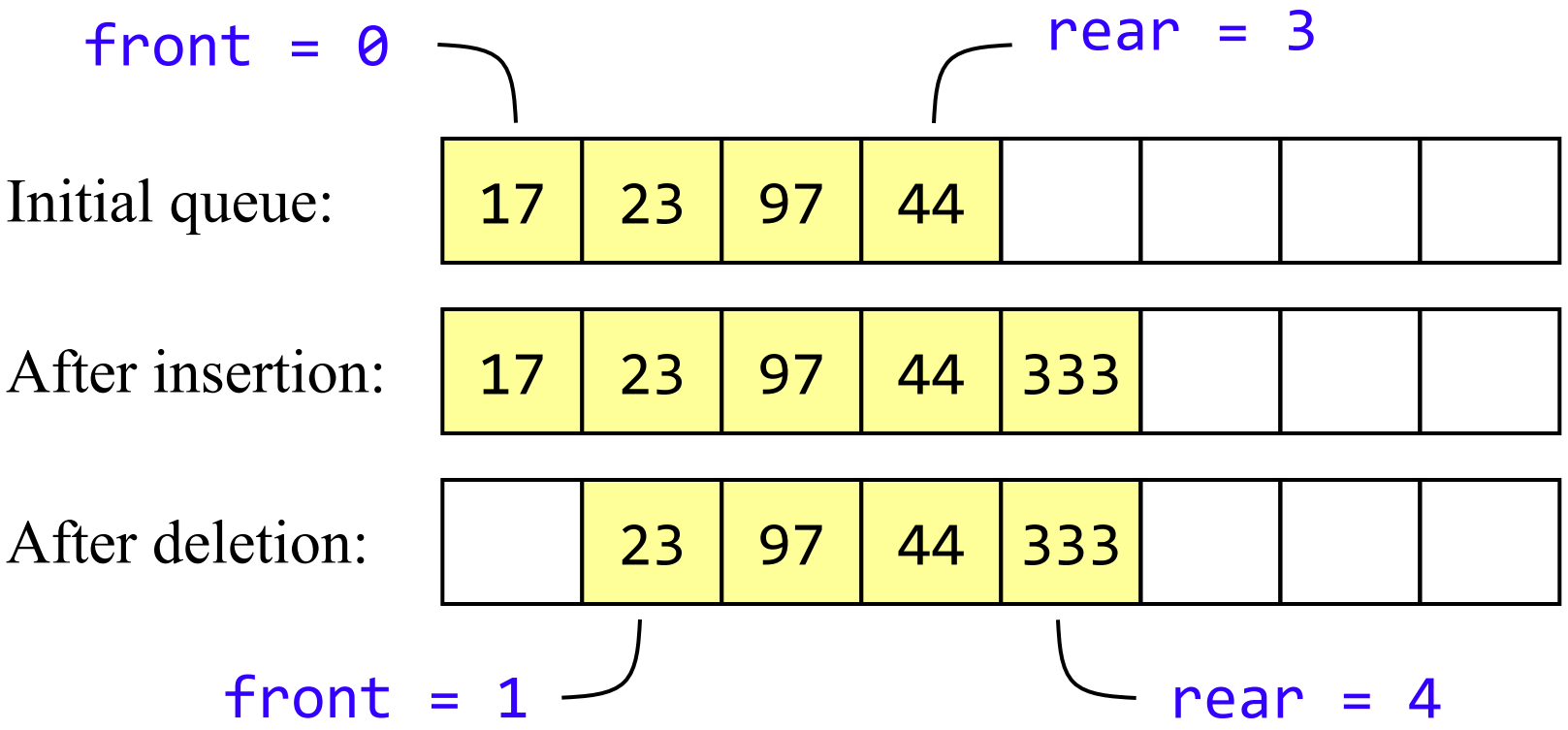
Array implementation of queues

- 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

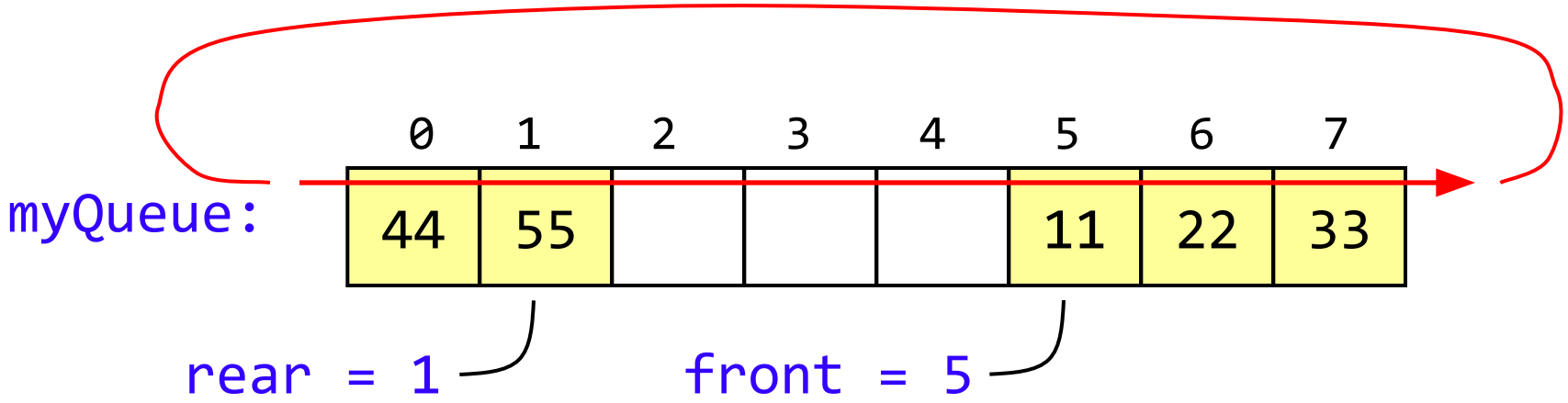
Array implementation of queues



- 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

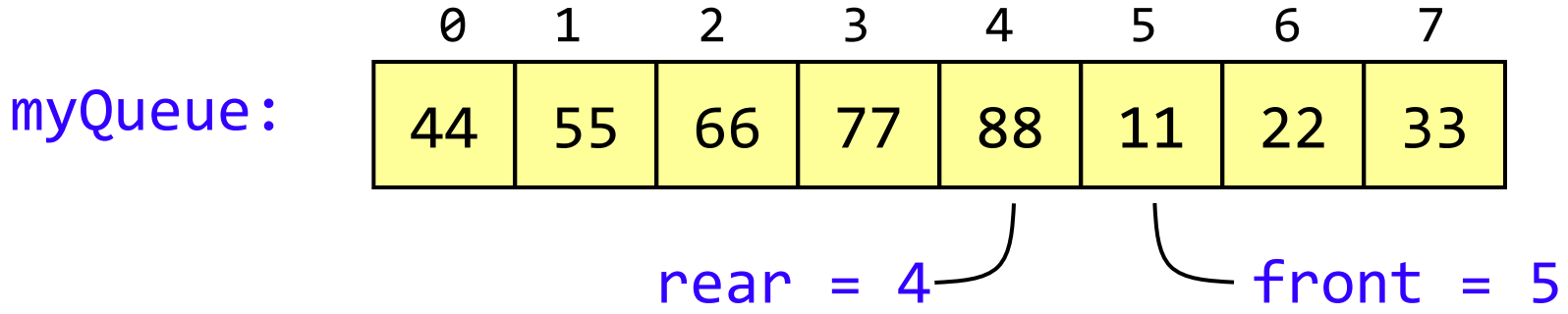
- We can treat the array holding the queue elements as circular (joined at the ends)



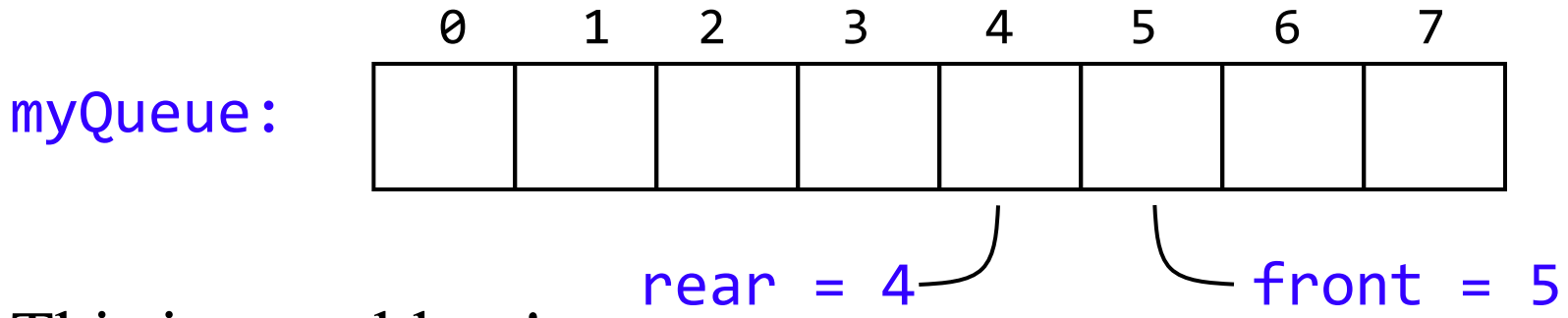
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

Full and empty queues

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



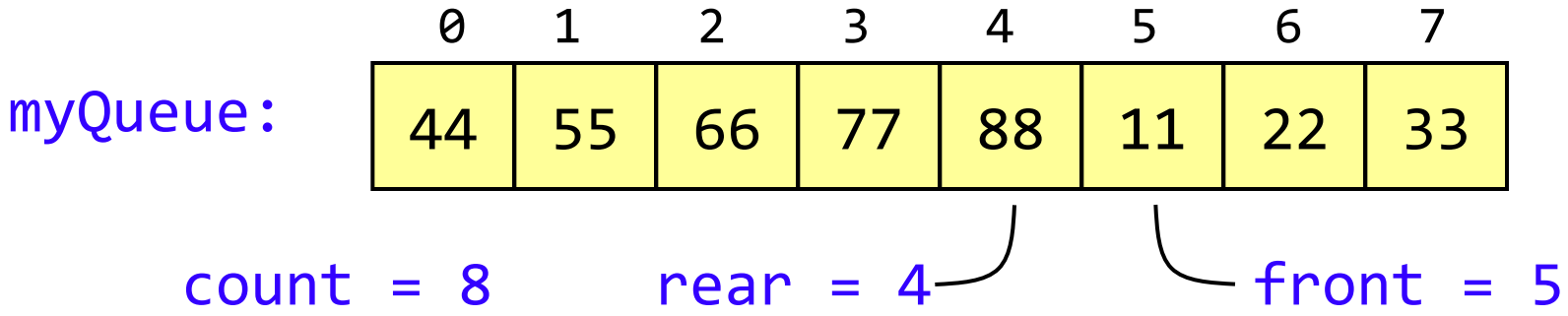
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



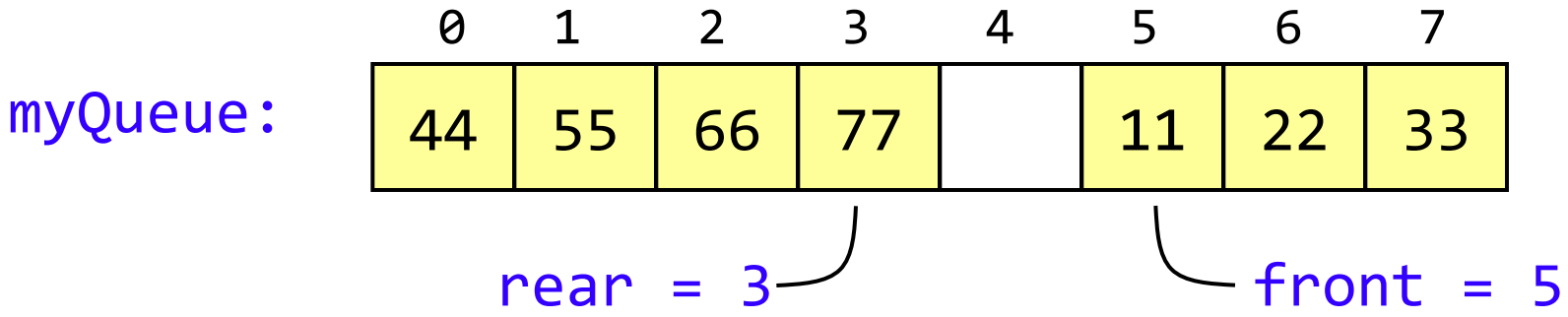
This is a problem!

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



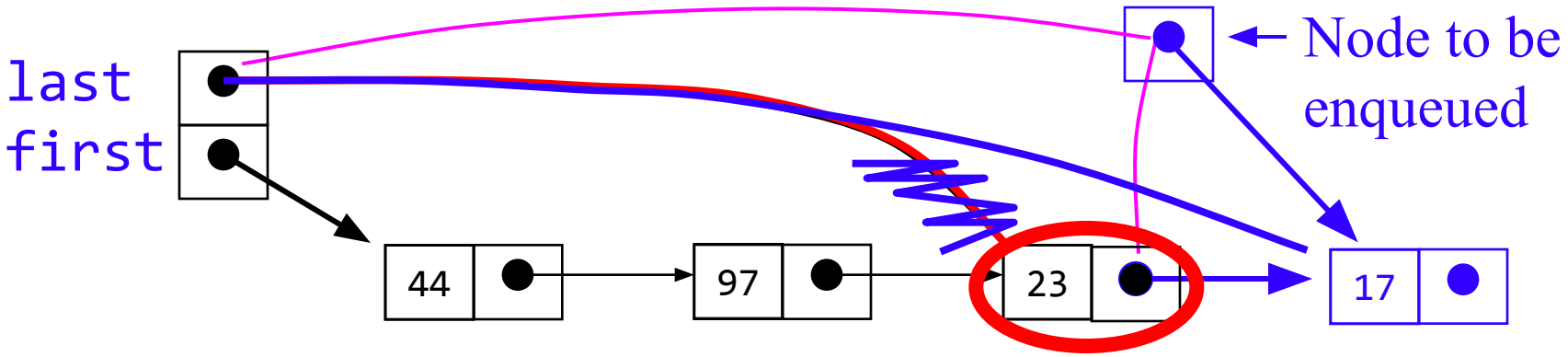
Linked-list implementation of queues

- 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



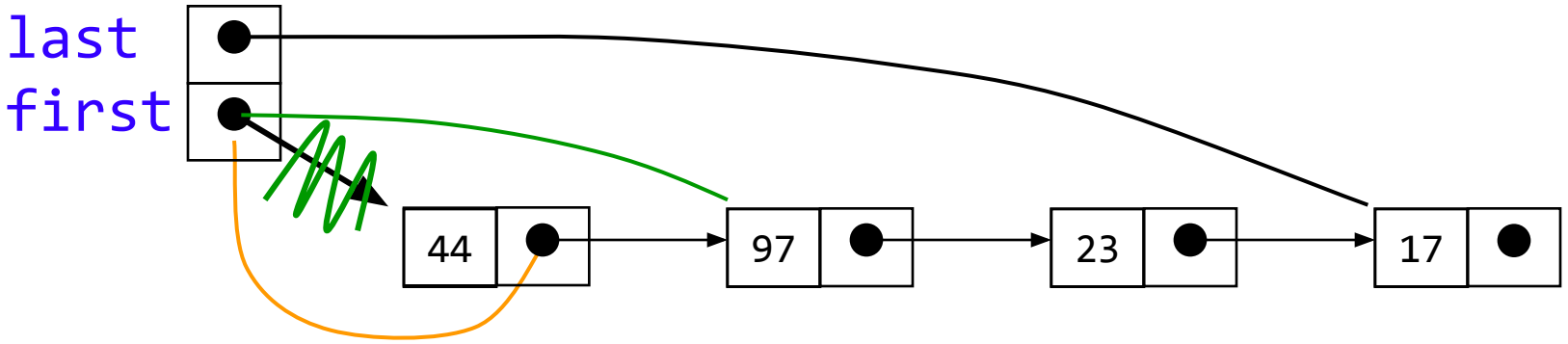
To enqueue (add) a node:

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

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 (not removes), the head of the queue /returns null if this queue is empty.
- `E poll()`
  - Retrieves and removes the head of this queue / 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()`

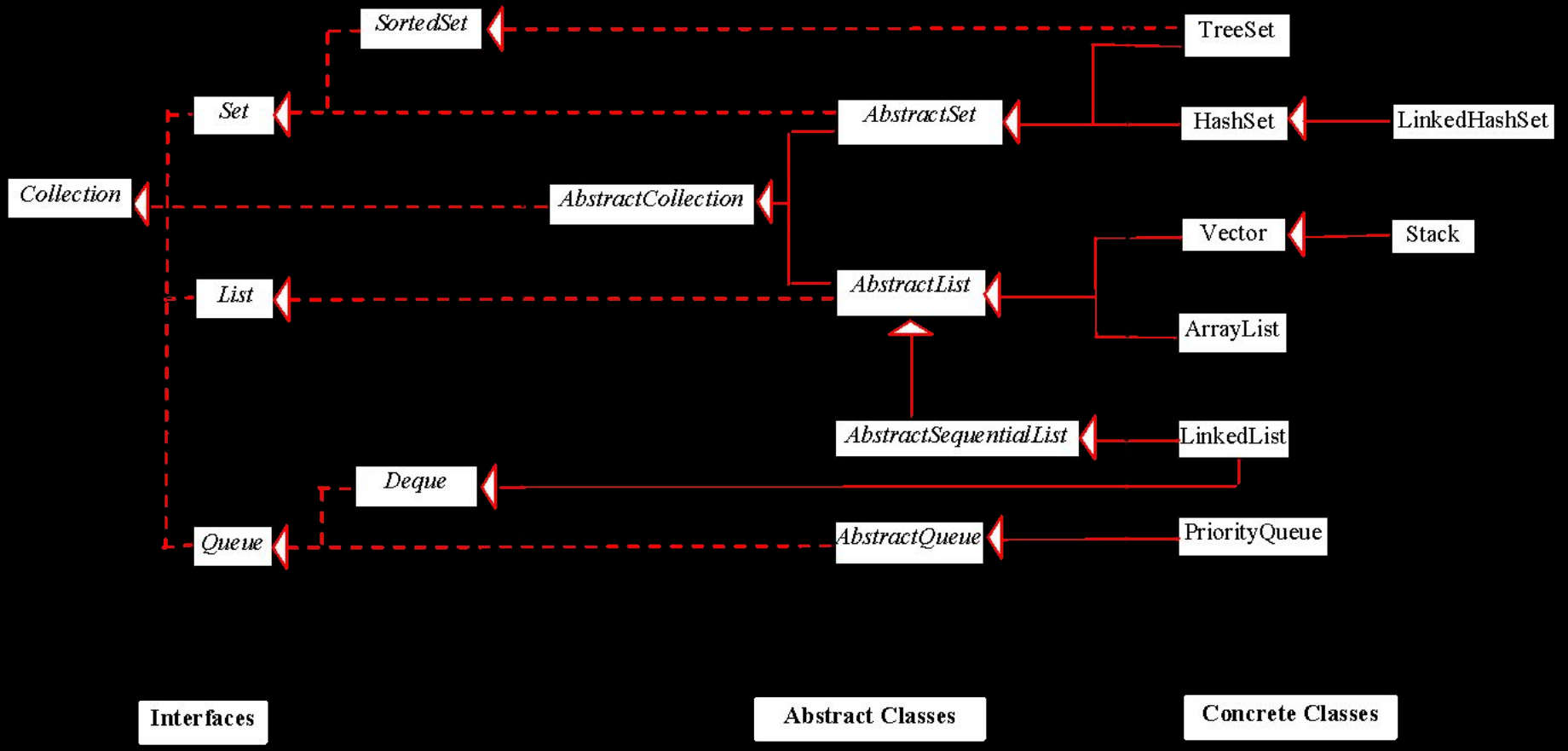
Source: Java 6 API

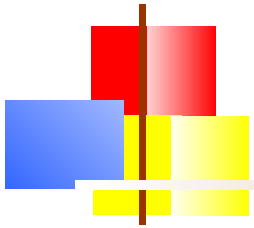
Java Collection Framework hierarchy

A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *sets, lists,* and *maps*.

Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.

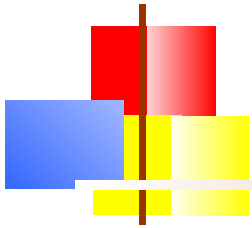




Collections as containers

- Containers can be studied under three points of views.
  - How to access the container elements
    - Arrays: Accessing is done using array index
    - Stacks: Done using LIFO (Last In First Out)
    - Queue: Done using FIFO (First In First Out)
  - How to store container elements in memory
    - Some containers are finite and some are infinite
  - How to traverse (visit) the elements of the container

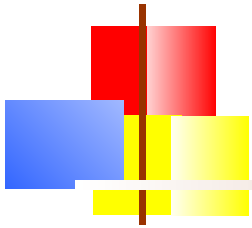
Source: http://en.wikipedia.org/wiki/Collection_class



Methods for container classes

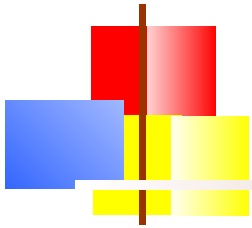
- Container classes are expected to implement methods to do the following:
  - Create a new empty container (constructor),
  - Report the number of objects it stores (size),
  - Delete all the objects in the container (clear),
  - Insert new objects into the container,
  - Remove objects from it,
  - Provide access to the stored objects.
- Containers are sometimes implemented in conjunction with iterators.

Source: http://en.wikipedia.org/wiki/Collection_class



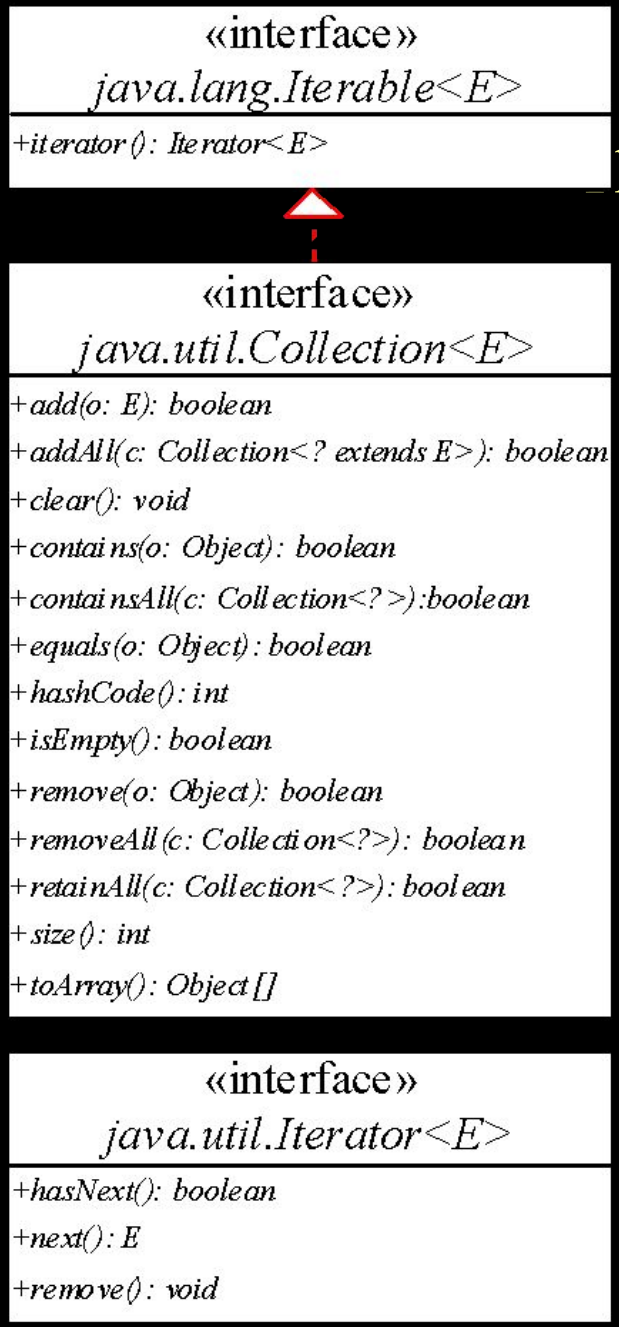
The Collection interface

- Much of the elegance of the Collections Framework arises from the intelligent use of interfaces
- The Collection interface specifies (among many other operations):
  - boolean add(E o)
  - boolean contains(Object o)
  - boolean remove(Object o)
  - boolean isEmpty()
  - int size()
  - Object[] toArray()
  - Iterator<E> iterator()
- You should learn *all* the methods of the Collection interface--all are important



The Iterator interface

- An iterator is an object that will return the elements of a collection, one at a time
- interface Iterator<E>
  - boolean hasNext()
    - Returns true if the iteration has more elements
  - E next()
    - Returns the next element in the iteration
  - void remove()
    - Removes from the underlying collection the last element returned by the iterator (optional operation)



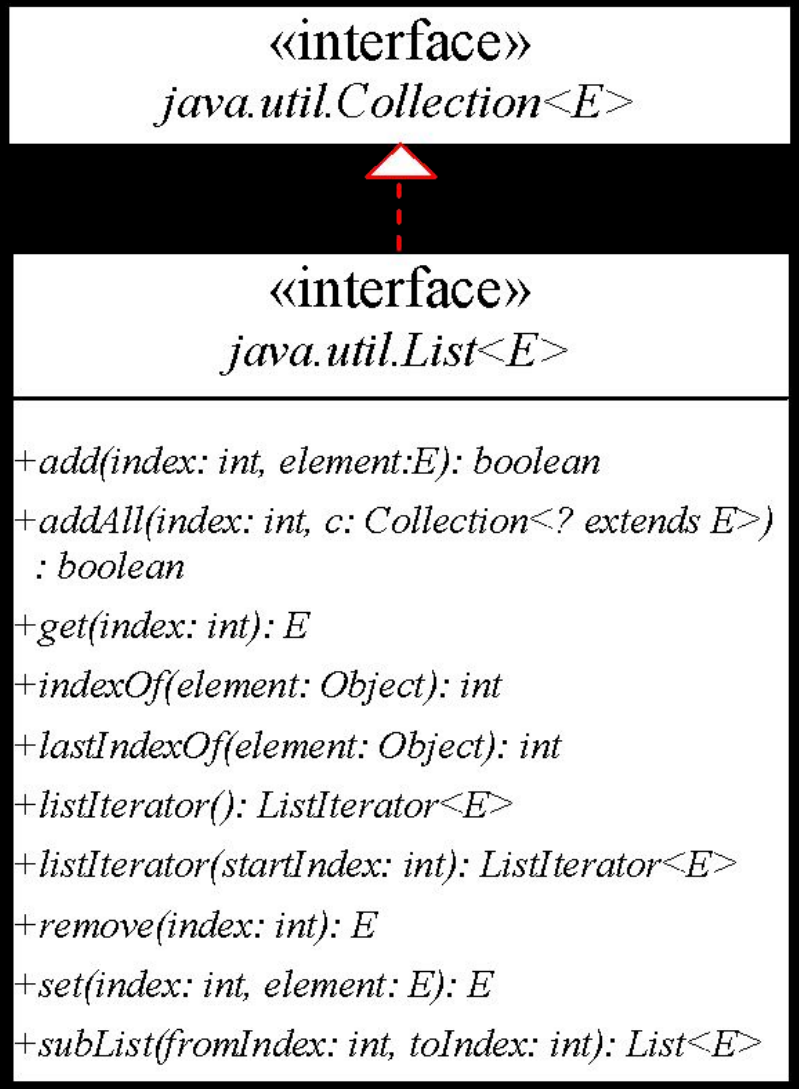
The Collection interface is the root interface for manipulating a collection of objects.

*

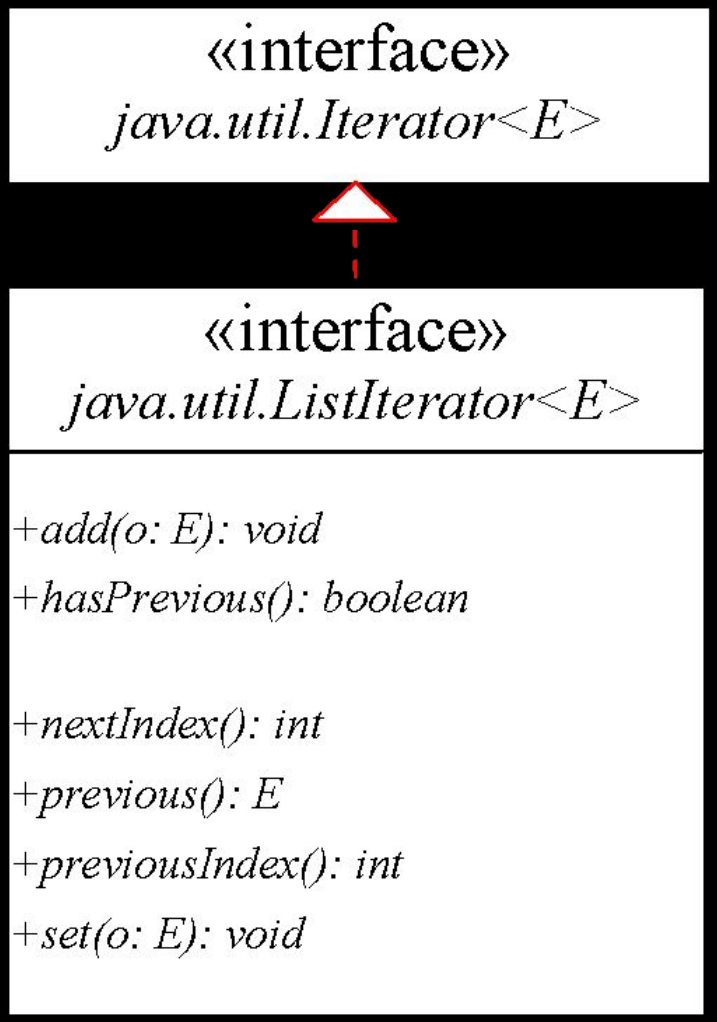
The List Interface

A set stores non-duplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list can not only store duplicate elements, but can also allow the user to specify where the element is stored. The user can access the element by index.

The List Interface, cont.



The List Iterator



Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

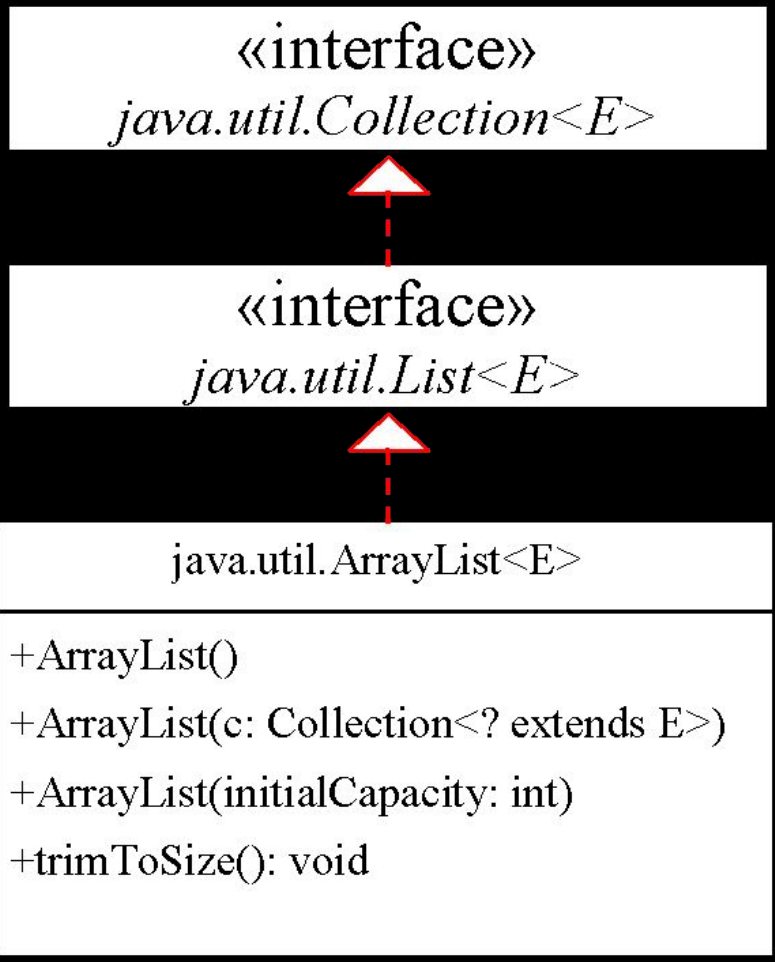
Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

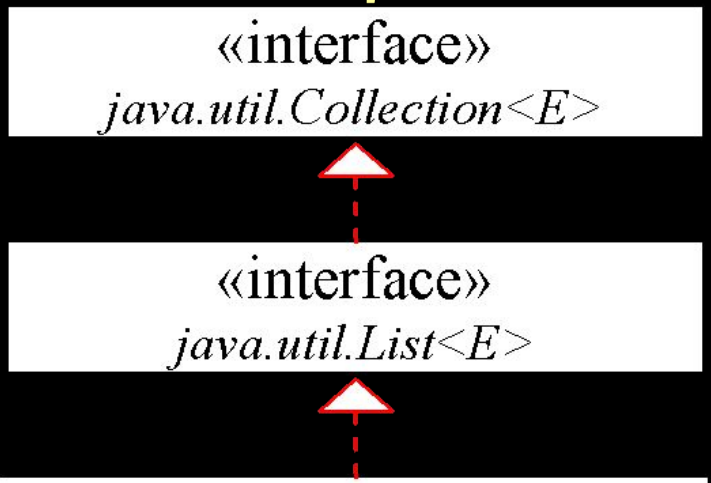
ArrayList and LinkedList

The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection. If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList



java.util.LinkedList



+LinkedList()	Creates a default empty linked list.
+LinkedList(c: Collection<? extends E>)	Creates a linked list from an existing collection.
+addFirst(o: E): void	Adds the object to the head of this list.
+addLast(o: E): void	Adds the object to the tail of this list.
+getFirst(): E	Returns the first element from this list.
+getLast(): E	Returns the last element from this list.
+removeFirst(): E	Returns and removes the first element from this list.
+removeLast(): E	Returns and removes the last element from this list.

The End