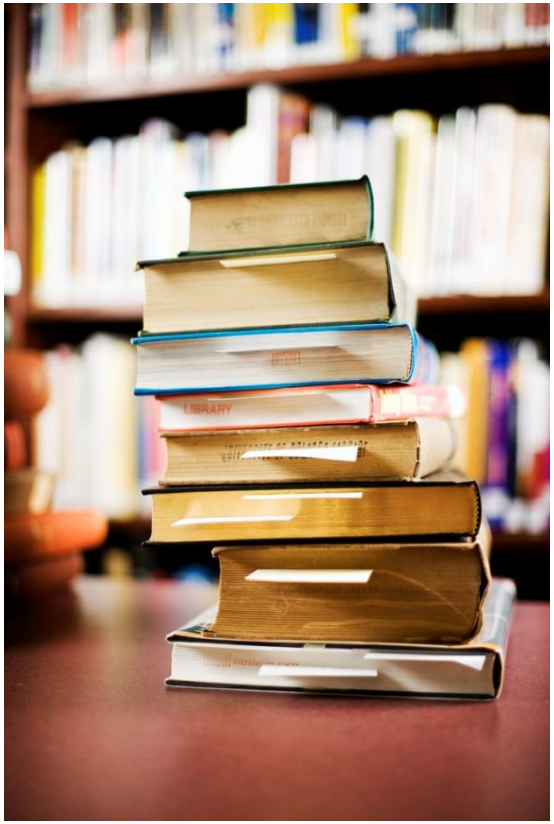
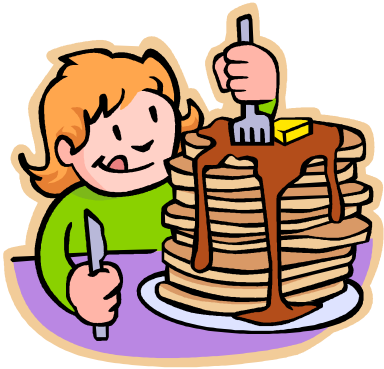
More Data Structures (Part 1)

Stacks

Stack

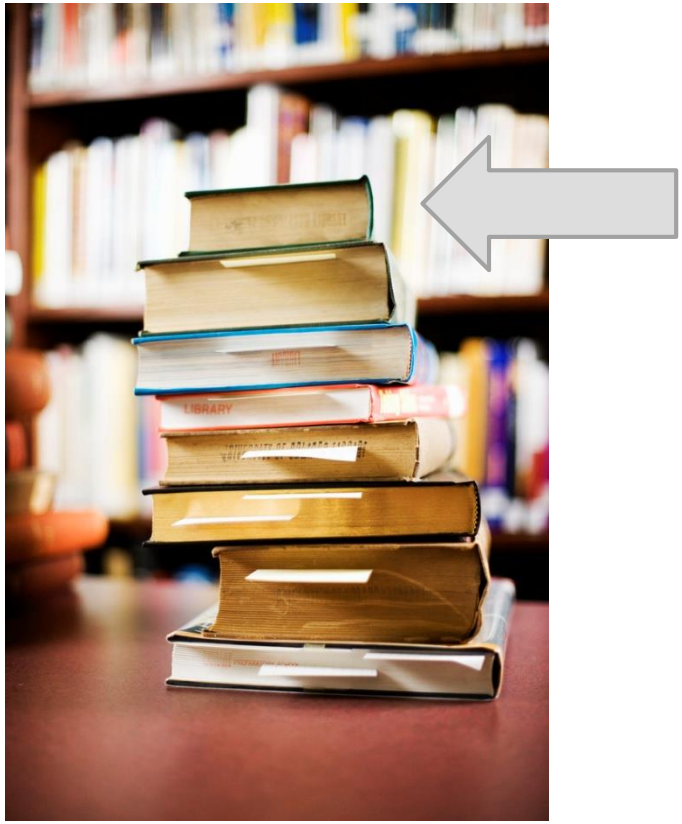
- examples of stacks





Top of Stack

- top of the stack

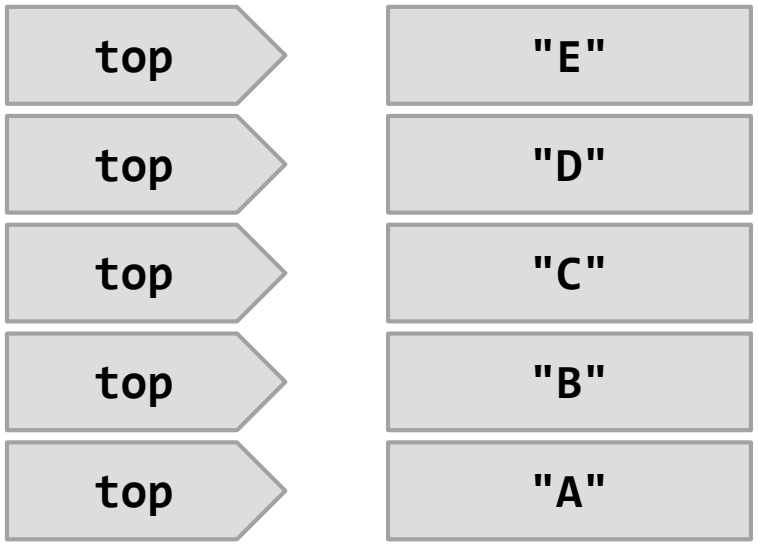


Stack Operations

- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack
     - throws an exception if there is nothing on the stack

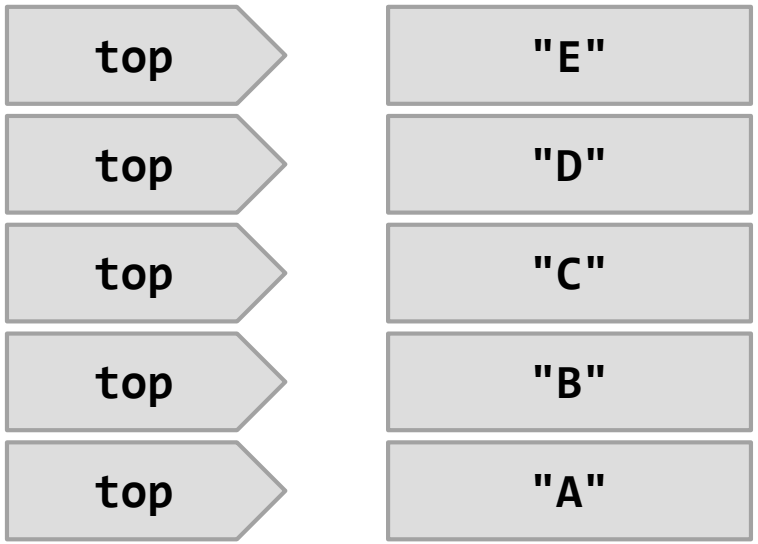
Push

1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



Pop

1. `String s = st.pop()`
2. `s = st.pop()`
3. `s = st.pop()`
4. `s = st.pop()`
5. `s = st.pop()`



Applications

- stacks are used widely in computer science and computer engineering
  - undo/redo
  - widely used in parsing
  - a call stack is used to store information about the active methods in a Java program
  - convert a recursive method into a non-recursive one

Example: Reversing a sequence

- a silly and usually inefficient way to reverse a sequence is to use a stack

Don't do this

```
public static <E> List<E> reverse(List<E> t) {
  List<E> result = new ArrayList<E>();
  Stack<E> st = new Stack<E>();
  for (E e : t) {
    st.push(e);
  }
  while (!st.isEmpty()) {
    result.add(st.pop());
  }
  return result;
}
```

Implementation with ArrayList

- **`ArrayList`** can be used to efficiently implement a stack
- the end of the list becomes the top of the stack
  - adding and removing to the end of an **`ArrayList`** usually can be performed in O(1) time

```
public class Stack<E> {
  private ArrayList<E> stack;

  public Stack() {
    this.stack = new ArrayList<E>();
  }

  public void push(E element) {
    this.stack.add(element);
  }

  public E pop() {
    return this.stack.remove(this.stack.size() - 1);
  }
}
```

Implementation with Array

- the ArrayList version of stack hints at how to implement a stack using a plain array
  - however, an array always holds a fixed number of elements
    - you cannot add to the end of the array without creating a new array
    - you cannot remove elements from the array without creating a new array
- instead of adding and removing from the end of the array, we need to keep track of which element of the array represents the current top of the stack
  - we need a field for this index

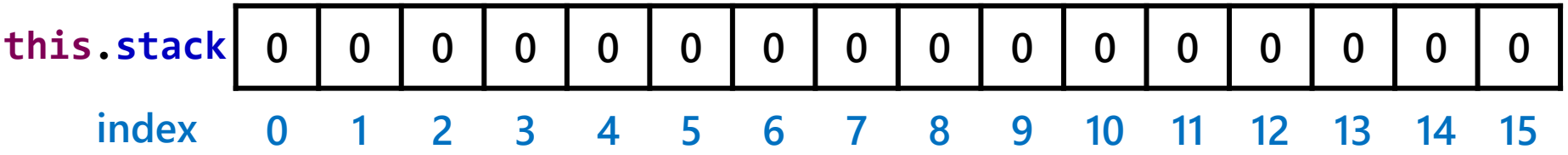
```
import java.util.Arrays;
import java.util.EmptyStackException;

public class IntStack {
  // the initial capacity of the stack
  private static final int DEFAULT_CAPACITY = 16;

  // the array that stores the stack
  private int[] stack;

  // the index of the top of the stack (equal to -1 for an empty stack)
  private int topIndex;
```

```java
/**
 * Create an empty stack.
 */
public IntStack() {
  this.stack = new int[IntStack.DEFAULT_CAPACITY];
  this.topIndex = -1;
}
```

Implementation with Array

```
IntStack t = new IntStack();
```

`this.topIndex == -1`



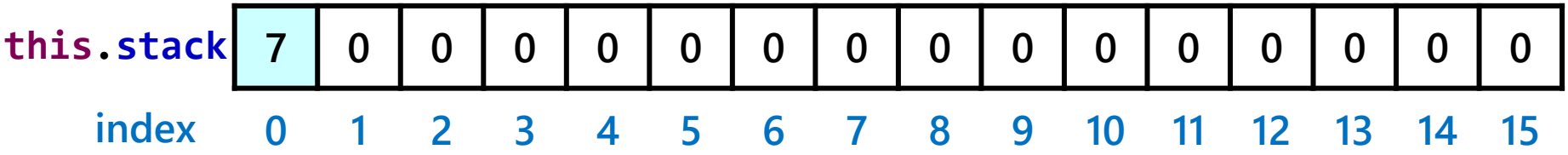
Implementation with Array

- pushing a value onto the stack:
  - increment `this.topIndex`
  - set the value at `this.stack[this.topIndex]`

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
```

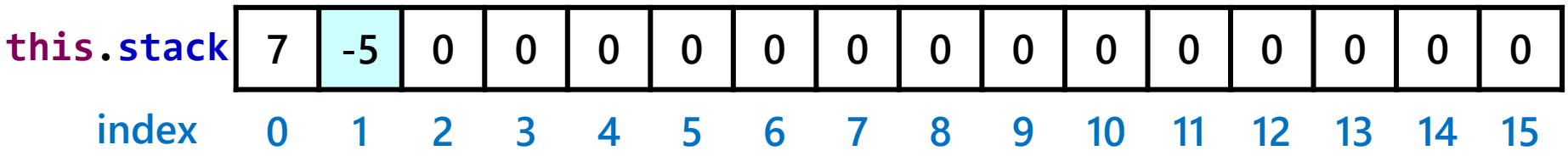
`this.topIndex == 0`



Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
```

`this.topIndex == 1`



Implementation with Array

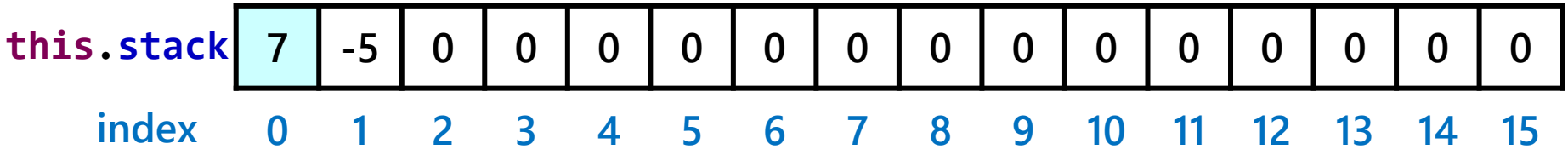
- popping a value from the stack:
  - get the value at `this.stack[this.topIndex]`
  - decrement `this.topIndex`
  - return the value

- notice that we do not need to modify the value stored in the array

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
int value = t.pop();   // value == -5
```

`this.topIndex == 0`

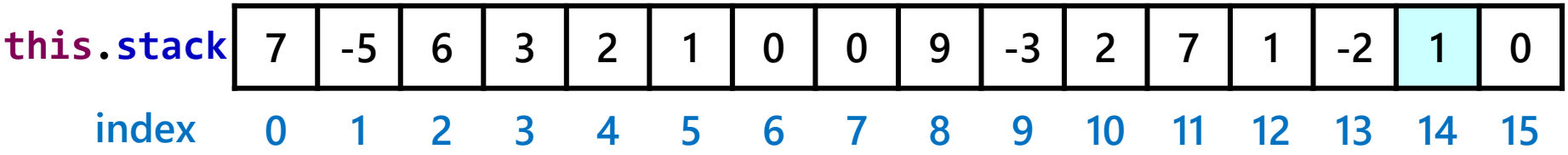


```
  /**
   * Pop and return the top element of the stack.
   *
   * @return the top element of the stack
   * @throws EmptyStackException if the stack is empty
   */
  public int pop() {
    // is the stack empty?
    if (this.topIndex == -1) {
      throw new EmptyStackException();
    }
    // get the element at the top of the stack
    int element = this.stack[this.topIndex];

    // adjust the top of stack index
    this.topIndex--;

    // return the element that was on the top of the stack
    return element;
}
```

Implementation with Array

// stack state when we can safely do one more push

`this.topIndex == 14`



```java
/**
 * Push an element onto the top of the stack.
 *
 * @param element the element to push onto the stack
 */
public void push(int element) {
  // is there capacity for one more element?
  if (this.topIndex < this.stack.length - 1) {
    // increment the top of stack index and insert the element
    this.topIndex++;
    this.stack[this.topIndex] = element;
  }
  else {
```

Adding Capacity

- if we run out of capacity in the current array we need to add capacity by doing the following:
  - make a new array with greater capacity
    - how much more capacity?
  - copy the old array into the new array
  - set `this.stack` to refer to the new array
  - push the element onto the stack

```
  else {
    // make a new array with double previous capacity
    int[] newStack = new int[this.stack.length * 2];

    // copy the old array into the new array
    for (int i = 0; i < this.stack.length; i++) {
      newStack[i] = this.stack[i];
    }

    // refer to the new array and push the element onto the stack
    this.stack = newStack;
    this.push(element);
  }
}
```

Adding Capacity

- when working with arrays, it is a common operation to have to create a new larger array when you run out of capacity in the existing array
- you should use **`Arrays.copyOf`** to create and copy an existing array into a new array

```
  else {
    int[] newStack = Arrays.copyOf(this.stack, this.stack.length * 2);
    // refer to the new array and push the element onto the stack
    this.stack = newStack;
    this.push(element);
  }
}
```

More Data Structures (Part 2)

Queues

Queue

Queue

Queue Operations

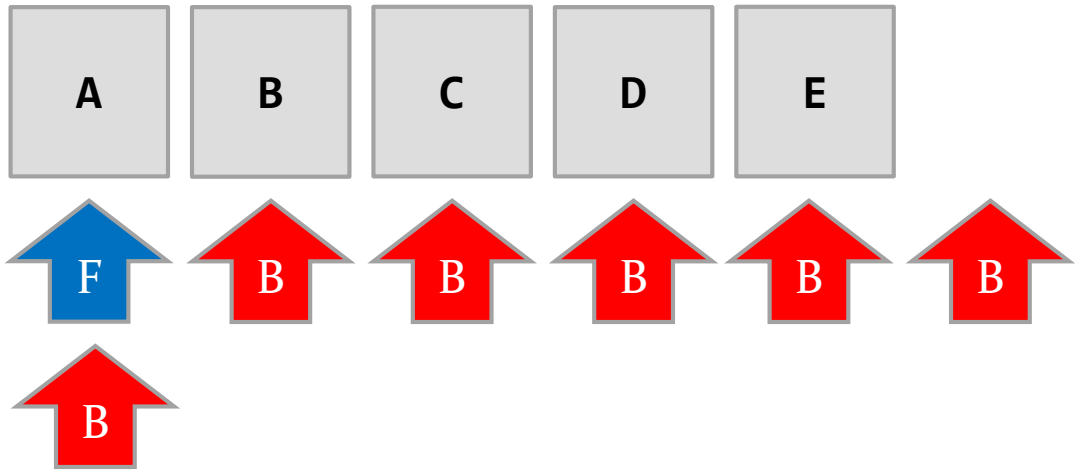
- classically, queues only support two operations
  1. enqueue
     - add to the back of the queue
  2. dequeue
     - remove from the front of the queue

Queue Optional Operations

- optional operations
  1. size
     - number of elements in the queue
  2. isEmpty
     - is the queue empty?
  3. peek
     - get the front element (without removing it)
  4. search
     - find the position of the element in the queue
  5. isFull
     - is the queue full? (for queues with finite capacity)
  6. capacity
     - total number of elements the queue can hold (for queues with finite capacity)

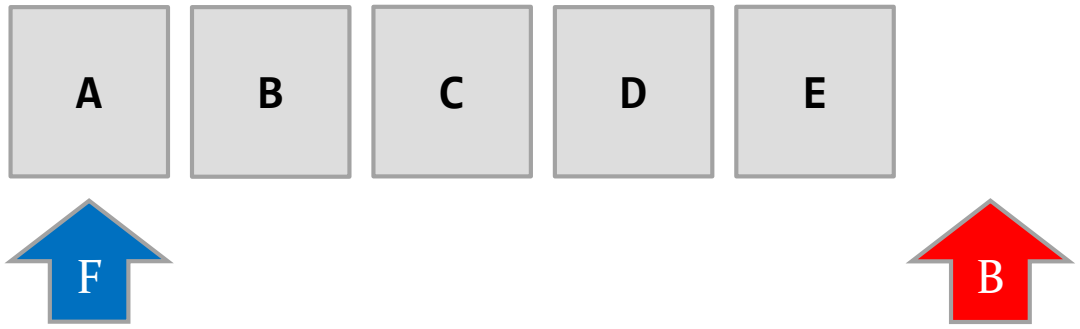
Enqueue

1. `q.enqueue("A")`
2. `q.enqueue("B")`
3. `q.enqueue("C")`
4. `q.enqueue("D")`
5. `q.enqueue("E")`



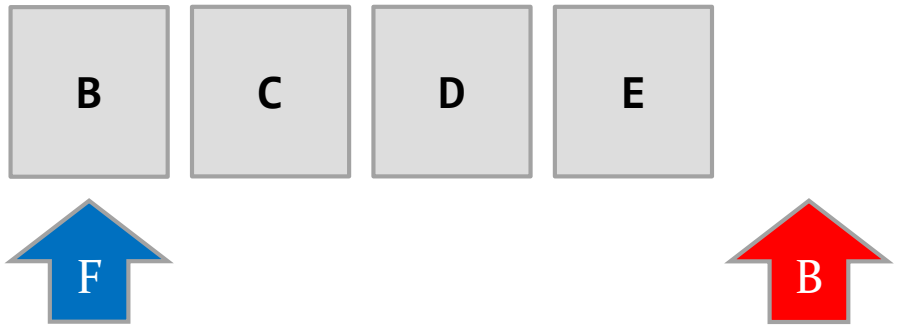
Dequeue

1. `String s = q.dequeue()`



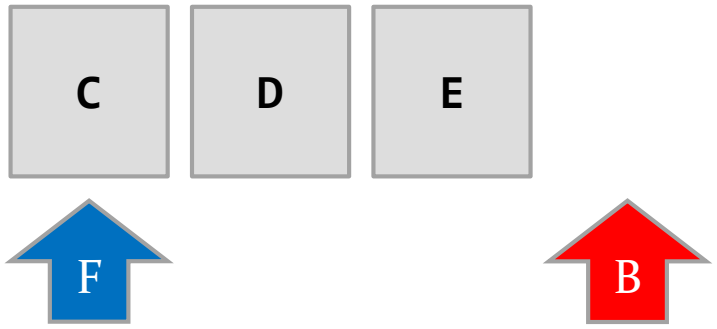
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`



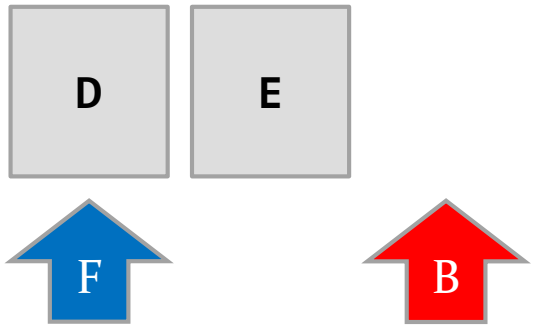
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`



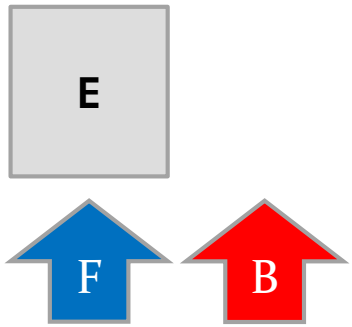
Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`



Dequeue

1. `String s = q.dequeue()`
2. `s = q.dequeue()`
3. `s = q.dequeue()`
4. `s = q.dequeue()`
5. `s = q.dequeue()`



FIFO

- queue is a First-In-First-Out (FIFO) data structure
  - the first element enqueued in the queue is the first element that can be accessed from the queue

Implementation with LinkedList

- a linked list can be used to efficiently implement a queue as long as the linked list keeps a reference to the last node in the list
  - required for enqueue
- the head of the list becomes the front of the queue
  - removing (dequeue) from the head of a linked list requires O(1) time
  - adding (enqueue) to the end of a linked list requires O(1) time if a reference to the last node is available

- **`java.util.LinkedList`** is a doubly linked list that holds a reference to the last node

```
public class Queue<E> {
  private LinkedList<E> q;

  public Queue() {
    this.q = new LinkedList<E>();
  }

  public enqueue(E element) {
    this.q.addLast(element);
  }

  public E dequeue() {
    return this.q.removeFirst();
  }
}
```

Implementation with LinkedList

- note that there is no need to implement your own queue as there is an existing interface
  - the interface does not use the names enqueue and dequeue however

java.util.Queue

```
public interface Queue<E>
  extends Collection<E>
```

`boolean`	**`add(E e)`** Inserts the specified element into this queue…
`E`	**`remove()`** Retrieves and removes the head of this queue…
`E`	**`peek()`** Retrieves, but does not remove, the head of this queue…

- plus other methods
  - http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

java.util.Queue

- **`LinkedList`** implements **`Queue`** so if you ever need a queue you can simply use:
  - e.g. for a queue of strings

```
Queue<String> q = new LinkedList<String>();
```