Code Design Review: The Bag ADT

- A bag is a container for a group of data items.
  - Analogy: a bag of candy

- Some characteristics of a bag:
  - item positions don’t matter (unlike a sorted list).
    - ✓{3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
  - items do *not* need to be unique (unlike a set).
    - ✓{7, 2, 10, 7, 5} is a bag but not a set
  - Could have a limited size

The Bag Operations

- Operations supported by the Bag ADT:
  - **add**(item): add item to the bag
  - **remove**(item): remove one occurrence of item (if any) from the bag
  - **contains**(item): check if item is in the bag
  - **grab**(): get an item at random, without removing it
  - **numItems**(): get the number of items in the bag
  - **toArray**(): get an array containing the current contents of the bag
  - others: isFull? weight?
- The operations are provided to the programmer.

- **Note**: we don't specify *how* the bag will be implemented.

The Bag Interface

- In Java, we can specify an ADT using an interface:

```
public interface Bag {
    boolean add(int item);
    boolean remove(int item);
    boolean contains(int item);
    …
}
```

- An interface specifies a set of methods.
  - it includes only the method headers
  - it *cannot* include the actual method definitions (i.e. bodies)

- To implement the ADT, we define a class that implements the interface:

```
public class MyBag implements Bag {
      …
}
```

  - Need data structures and algorithms (procedures/functions)

The Bag Implementation: Design Choices

- Bags can contain integers only.
  - We will consider a more general implementation later.
- We will use an array to store the items.
  - The array is *not* mandated by the ADT.
  - Other design choices are possible (e.g., linked list).

Implementation of IntBag in Java

```
public class IntBag implements Bag { // bag of ints for now
// instance variables (also known as fields, members, attributes, properties)
    private int[] items;  // items is a reference
    private int numItems;
    …
    // methods (also known as functions)
    public boolean add(int item) {
        if (numItems == items.length)
            return false; // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

Accessing Private Fields

```
public class IntBag implements Bag {
  //  instance variables (also known as fields,
      members, attributes, properties)

  private int[] items; //items is a reference
  private int numItems;
  …
  //  methods (also known as functions)
  public boolean add(int item) {
     if (numItems == items.length)
          return false; // no more room!
        else {
           items[numItems] = item;
           numItems++;
           return true;
        }
     }
     …
}
```

- Private fields are for the internal use by the implementation
  - Not exposed to ADT users.
  - Collectively form the data structures.
- A method can access a private field of its own object

```
public boolean addAll(IntBag other) {
    for (int i = 0; i < other.numItems; i++)
        add(other.items[i]);
}
```

- … or *other* objects *from the same class* that are passed to the method as parameters.

Encapsulation

- *Encapsulation* is one of the key principles of object-oriented programming.
  - Also known as *information hiding*
    - It refers to “hiding” the implementation of a class from users of the class.
  - Prevents *direct* access to the internals of an object
  - Provides controlled, limited, *indirect* access through a set of methods

- Black Box Analogy
  - We treat many objects in our lives as “black boxes.”
  - We know *what* operations they can perform, but we don’t know *how* they perform those operations
  - We can’t see inside the box, so we interact with them using a well-defined *interface*

- Power of abstraction!

Encapsulation (cont.)

- Java uses *private* instance variables (and occasionally private helper methods) to hide the implementation of a class.
  - these private members can only be accessed inside methods that are part of the same class

```
class MyClass {
    …
    void myMethod() {
        IntBag b = new IntBag();
        b.items[0] = 17; // not allowed!
    …
```

- Users are limited to the *public* methods of the class, as well as any public variables (usually limited to constants – why?).
  - public members can be accessed inside methods of *any* class

Benefits of Encapsulation

- It prevents inappropriate changes to the state of an object:

```
class MyClass {
    …
    void myMethod() {
        IntBag *b = new IntBag();
        b->addItem(7);
        b->addItem(22);
        b->numItems = 0; // not allowed
    …
```

- Can you think of another benefit?

- Please make sure to use proper encapsulation in the classes that you write for this course!

Example from Last Lecture: Code Reuse

```
public interface StrIterator{
        char next();
        boolean hasNext();
        ...
}

public interface StrList {
        void add(int idx, char x);
        void remove(int idx);
        StrIterator iterator();

        ...
}
```

```
public class StrArrayList implements StrList {
        public add(int idx, char x){ ... }
        ...
        public StrIterator iterator(){...}
        private static class StrArrIterator
                        implements StrIterator{
            private StrArrIterator(){...}
            ...
        }
}

public class StrLinkedList implements StrList {
        public add(int idx, char x){...}
        ...
        public StrIterator iterator(){...}
        private static class StrLLIterator
                        implements StrIterator{
            private StrLLIterator(){...}
            ...
        }
}
```

```
public class Demo {
    public static int numOccur(StrList str, char ch) {
        int numOccur = 0;
        StrList.StrIterator iter = str.iterator();
        while (iter.hasNext()) {
            char ch = iter.next();
            if (c == ch)
                numOccur++;
        }
        return numOccur;
    }

}
```

```
public static void main( String [] arg){
    StrArrayList string1 = new StrArrayList(“cat”};
    StrLinkedList string2 = new StrLinkedList(“puppy”);
    if (numOccur(string1, “c”) < numOccur(string2, “p”){
        ....
    }
}
```

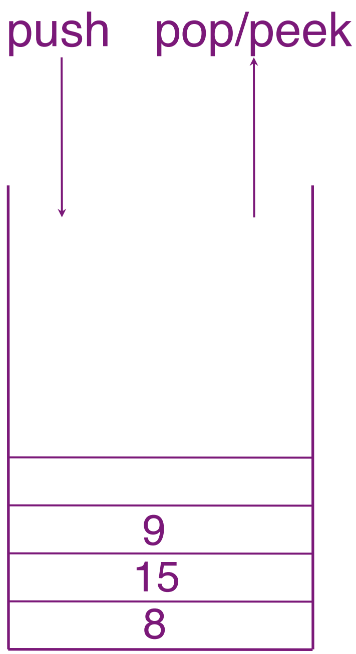
Stacks and Queues

EECS 233

Last Week: Lists

- How to represent a sequence of objects? Lists
- Data structures:
  - Array implementation
  - Linked lists
- Operations or methods:
  - Insert
  - Remove
  - Traversal
  - etc

Stack ADT

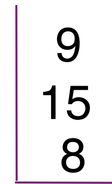


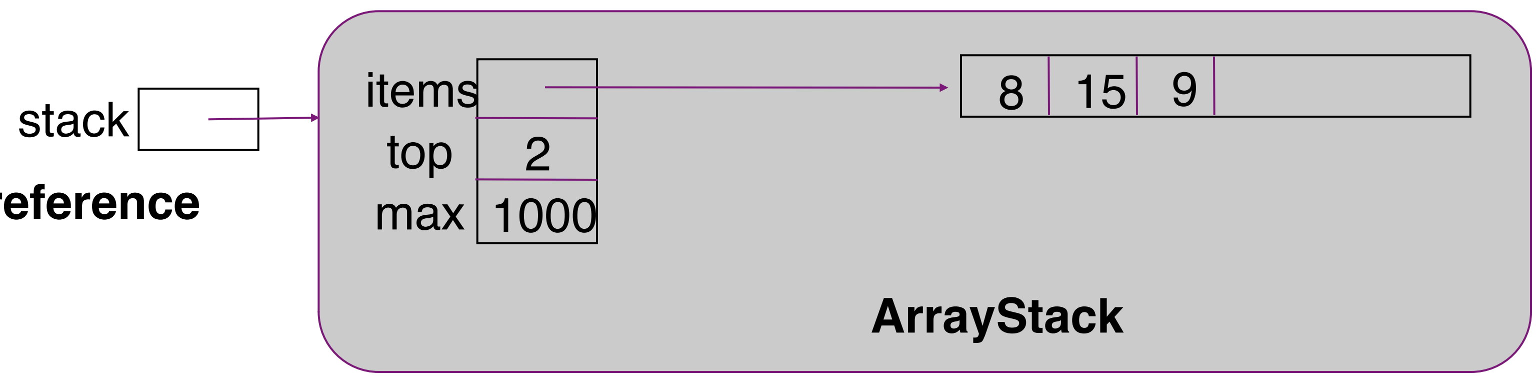
- A stack is a special sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top
- Operations:
  - boolean push(ItemType i); add an item to the top of the stack
  - ItemType pop(); remove the item at the top of the stack
  - ItemType peek(); get the item at the top of the stack, but don't remove it
  - boolean isEmpty();
  - boolean isFull();
- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - Enforced by encapsulation

push(8); push(15); push(9); pop() - returns 9

Array Implementation of Stacks

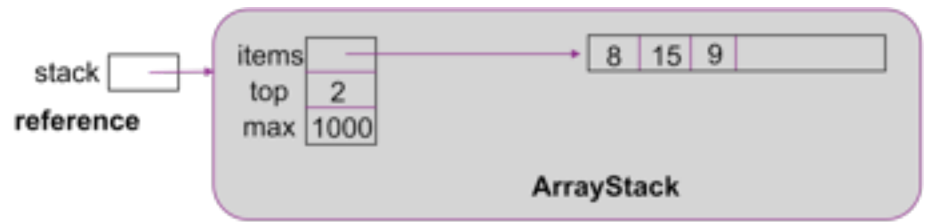
- Example: the integer stack would be represented as follows:





Array Implementation: Constructors and Methods

```
public ArrayStack(int max) {
      items = new int[max];          // a stack of integers
      top = -1;
      maxSize = max;
}

public boolean isEmpty() {
      return (top == -1);
}

public boolean isFull() {
      return (top == maxSize – 1);
}
```



Java Generics

- Consider the Bag class again.
- Suppose we want to implement a bag for any type of object
- A bag of candy, a bag of apples, a bag of baseballs, … a bag of integers, a bag of floating-point numbers, …
  - A bag of objects
  - the *logical operation* is independent of the types of objects

- Code reuse for different bags (various types of objects), rather than recode the same (or almost identical) logic (e.g. same algorithms) for different types

- Type-independent data structures and algorithms can be used more widely
  - Linked list of integers, or linked list of (name, phone#) pairs, …
  - tree of different types
  - …

- Accomplished through Java *Generics*

Using a Superclass to Implement General Classes

```
public class Bag {
     // instance variables (also known as fields, members,
        attributes, properties)
     private Object[] items;  // items is a reference
     private int numItems;
     …
     // methods (also known as functions)
     public boolean add(Object item) {
          if (numItems == items.length)
            return false; // no more room!
          else {
            items[numItems] = item;
            numItems++;
            return true;
          }
     }
     …
     public Object getFirst() {
          if (numItems == 0)
            return false; // nothing there
          else {
            return items[0];
          }
     }
     …
}
```

Using the Generic Class

- Type downcast for access generic class objects

```
public class Test1 {
    public static void main( String [ ] args ) {
        Bag b = new Bag( );

        b.add( “37C” );
        String bodyTemp = (String) b.getFirst( );
        System.out.println( ”Temperature is: " + bodytemp );
    }
}
```

- The “add” method passes a string, so the actual object is String.
- The “getFirst” method cannot tell by itself what should be the return type.
  - A typecast is necessary!

Using the Generic Class (cont.)

- No restrictions on object types in a Bag

```
public class Test2 {
    public static void main( String [ ] args )     {
        Bag b = new Bag( );

        b.add( “37C” );
        b.add(new Integer(96)); // Wrapper class needed! Why?
        BodyTemperature temp = (BodyTemperature) m.getFirst( );
        // Run-time error!
     }
}
```

Because the primitive types are not objects. The wrapper class stores the primitive type and adds operations that it doesn’t implement.

Java’s primitive types: byte, short, int, long, float, double, boolean, char

Implementing General Classes (Java >=5)

```
public class Bag<AnyType> {
    private AnyType[] items;
    private int numItems;

    public AnyType getFirst( )
      { … }
    public void Add(AnyType x )
      { … }

    private AnyType storedValue;
}
```

- Include one or more type parameters in <>
- Bag<String> replaces AnyType with "String" throughout – bag of strings
- Bag<BodyTemperature> -- bag of BodyTemperature objects.
- Type correctness can be checked at compile time.

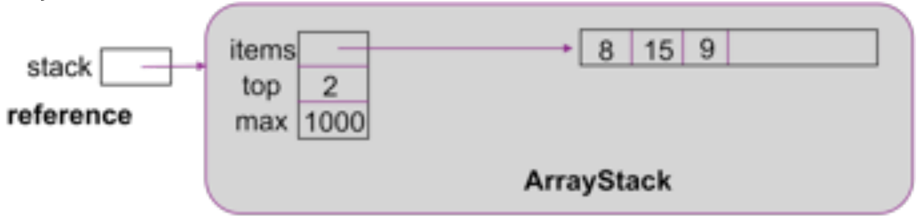
Generic ArrayStack Class

- Generic classes use *type variables T* that serve as placeholders for actual types.

```
public class ArrayStack<T> {
        private T[ ] items;
        private int top;
        private int maxSize;
        …
        public boolean push(T item) {
        …
}
```

- Constructor is rewritten as:

```
public ArrayStack(int max) {
        items = (T[ ]) new Object[max];
        top = -1;
        maxSize = max;
}
```



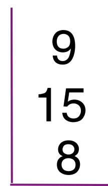
- Methods: push(), pop(), peek()

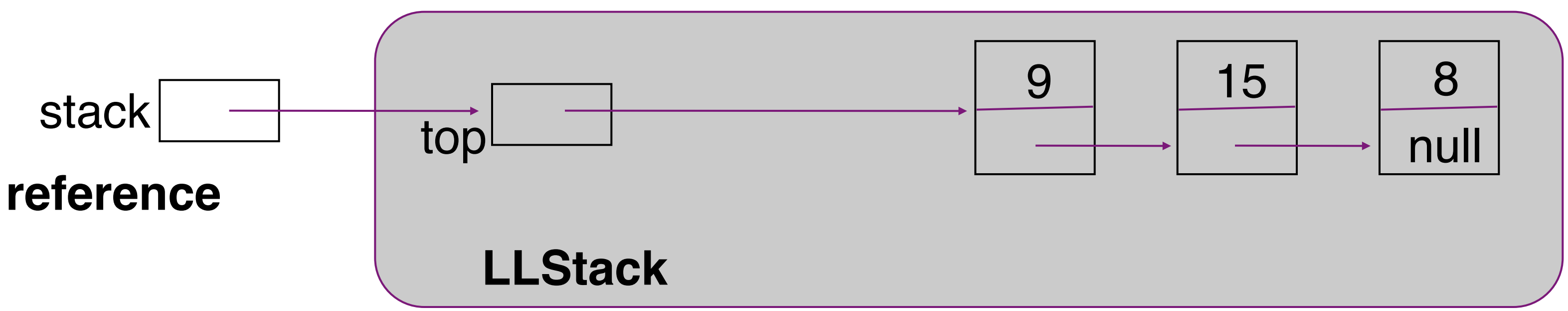
```
public boolean push(T item) {
        if (isFull())
                return false;
        top++;
        items[top] = item;
        return true;
}

public T pop() {
        if (isEmpty())
                throw exception;
        return items[top--];
}

public T peek() {
        if (isEmpty())
                throw exception;
        return items[top];
}
```

Usage:

ArrayStack<Bag> bagStack = new ArrayStack<Bag>(100);

Linked-List Implementation of Stacks

- The integer stack  in linked list:



Generic LLStack Class

```
public class LLStack<T> {
    private class Node {
        T item;
        Node next;
    }

    private Node top;
    ...

    public boolean push(T item) {
        ...
    }
}
```

Applications: Checking for Delimiter Balancing

- Making sure delimiters (e.g., parentheses, brackets) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example: 5 * [3 + { (5 + 16 – 2)]
    - ✓ push “[“; push “{“; push “(“;
    - ✓ pop “(“ when seeing “)”
    - ✓ pop “{“ when seeing “]” ???

Queue ADT

- A queue is a special sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - we can only access the item that is currently at the front
- Operations:
  - boolean insert(T item); add an item at the rear of the queue
  - T remove(); remove the item at the front of the queue
  - T peek(); get the item at the front of the queue, but don't remove it
  - boolean isEmpty(); test if the queue is empty
  - boolean isFull(); test if the queue is full
- Example: a queue of integers
  - *Starting state:* 12 8
  - *insert 5:* 12 8 5
  - *remove:* 8 5

Array Implementation of Generic Queues

- Five instance variables:

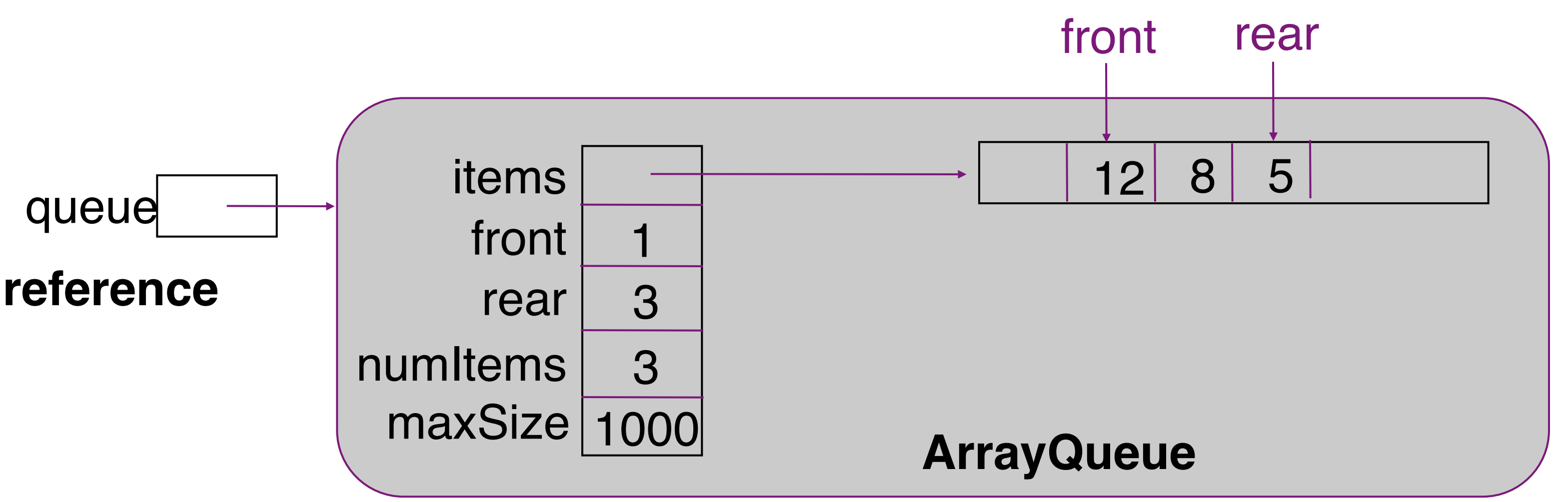
  T[ ] items; // array of type T (type variable)

  int front; // index of item at front of queue

  int rear; // index of item at rear of queue

  int numItems; // number of items in queue (optional for now)

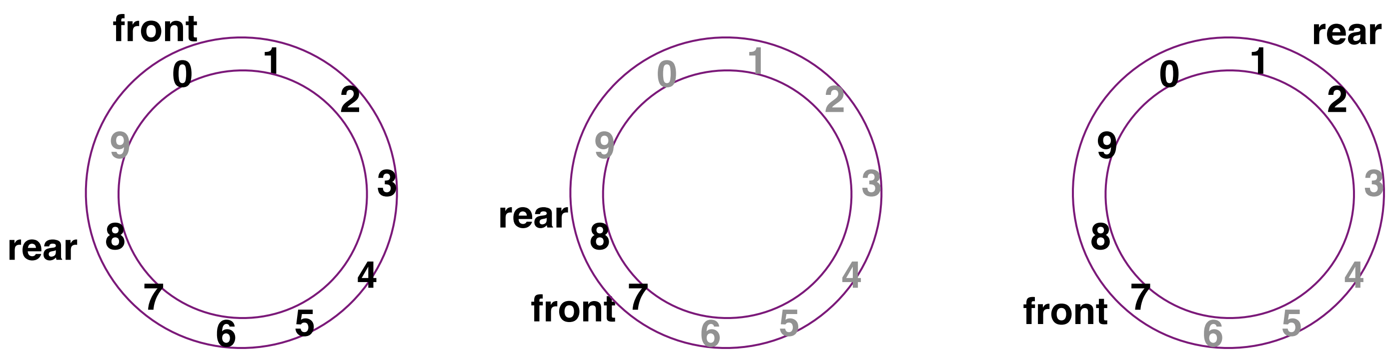
  int maxSize; // size of the array (optional - see array.length)



Problem: what do we do when we reach the end of the array?

Circular Array Implementation

- Problem: what do we do when we reach the end of the array?
- Solution: a *circular array*.
  - When we reach the end of the array, we wrap around to the beginning.

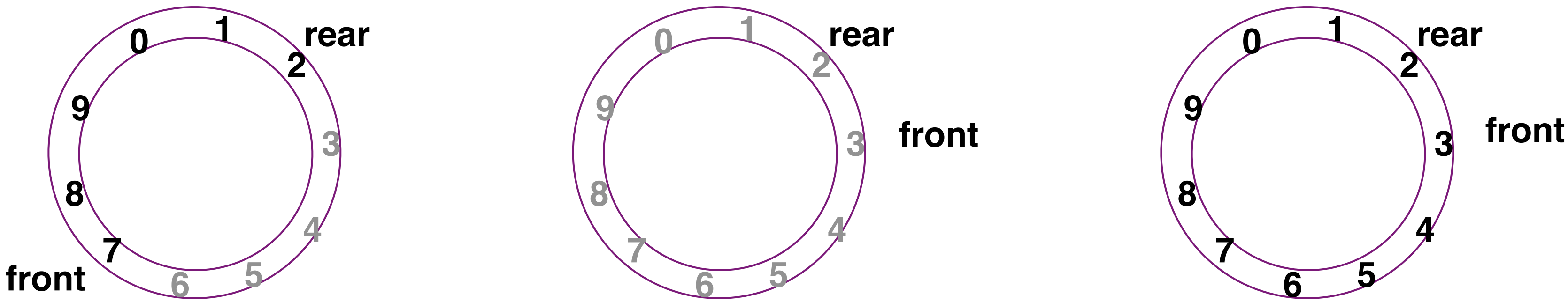


After 9 insertions

After 7 removals

After 4 insertions

Circular Array: Distinguishing Full From Empty



Previous state

After 6 more removals: empty

After 10 more insertions: full

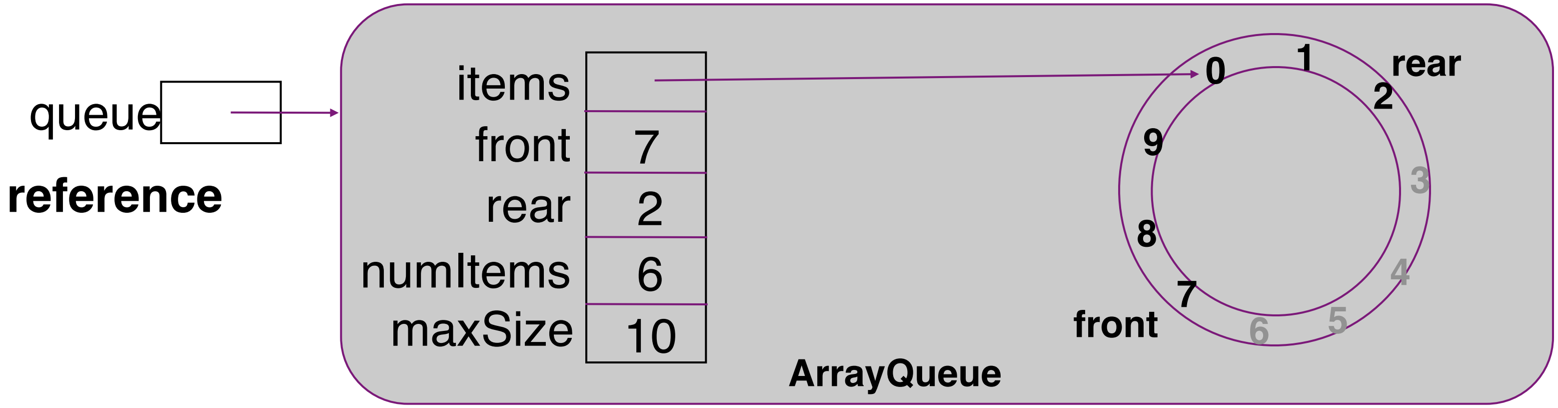
- The queue is empty when front “overcomes” rear:
  - ((rear + 1) % maxSize) == front
- But how to test if an ArrayQueue is full?
  - we maintain numItems!
  - return (numItems == maxSize);

Constructors and Methods

```
public ArrayQueue<T>(int max) {
        items = (T[ ]) new Object[max];
        maxSize = max;
        front = 0;
        rear = -1;
        numItems = 0;
}
```

```
public boolean insert(T item) {
        if (isFull())
                return false;
        rear = (rear + 1) % items.length;
        items[rear] = item;
        return true;
}
```

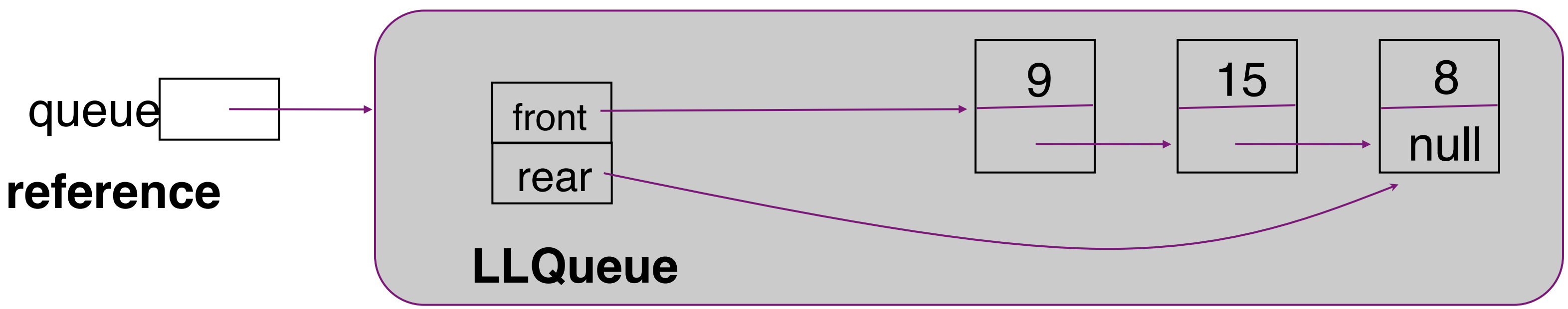
```
public T remove() {
        if (isEmpty())
                throw exception;
        T removed = items[front];
        front = (front + 1) % items.length;
        return removed;
}
```



--

Linked-List Implementation of Queues

- Two instance variables:
  - Node front; // front of the queue
  - Node rear; // rear of the queue



- No capacity issue: no need for circular buffer.

Applications of Queues

- First-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- Simulations of banks, supermarkets, airports, etc.
- Breadth-first traversal of a graph (stay tuned…)

Summary: Efficiency of Stacks and Queues

- Stack and Queue complexity
- Array and linked list implementation
  - Running time of insert (push), remove (pop), peek
  - Space complexity?
- Problem-of-the-week: emulate a queue using stacks