Lists, Stacks, and Queues

Computer Science S-111
Harvard University
David G. Sullivan, Ph.D.

Representing a Sequence: Arrays vs. Linked Lists

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
  - Can represent any sequence using an array or a linked list

<table>
<thead>
<tr>
<th></th>
<th>array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>representation in memory</td>
<td>elements occupy consecutive memory locations</td>
<td>nodes can be at arbitrary locations in memory; the links connect the nodes together</td>
</tr>
<tr>
<td>advantages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>disadvantages</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The List ADT

- A list is a sequence in which items can be accessed, inserted, and removed at any position in the sequence.

- The operations supported by our List ADT:
  - `getItem(i)`: get the item at position i
  - `addItem(item, i)`: add the specified item at position i
  - `removeItem(i)`: remove the item at position i
  - `length()`: get the number of items in the list
  - `isFull()`: test if the list already has the maximum number of items

- Note that we don’t specify how the list will be implemented.

Our List Interface

```java
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- Recall that all methods in an interface are assumed to be public.
- The actual interface definition includes comments that describe what each method should do.
Implementing a List Using an Array

```java
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```

Recall: The Implicit Parameter

- All non-static methods have an implicit parameter (this) that refers to the called object.
- In most cases, we're allowed to omit it!
  - we'll do so in the remaining notes
Omitting The Implicit Parameter

```java
public class ArrayList implements List {
    private Object[] items;
    private int length;
    public ArrayList(int maxSize) {
        items = new Object[maxSize];
        length = 0;
    }
    public int length() {
        return length;
    }
    public boolean isFull() {
        return (length == items.length);
    }
    ...
}
```

- In a non-static method, if we use a variable that
  - isn't declared in the method
  - has the name of one of the fields
    Java assumes that we're using the field.

Adding an Item to an ArrayList

- Adding at position i (shifting items i, i+1, ... to the right by one):
  ```java
  public boolean addItem(Object item, int i) {
      if (i < 0 || i > length) {
          throw new IndexOutOfBoundsException();
      } else if (isFull()) {
          return false;
      }
      // make room for the new item
      for (int j = length - 1; j >= i; j--) {
          items[j + 1] = items[j];
      }
      items[i] = item;
      length++;
      return true;
  }
  ```

Example for i = 3:
```
items  0 1 2 3 4 5 6 7 8
length 6
```
Adding an Item to an ArrayList

• Adding at position i (shifting items i, i+1, ... to the right by one):

```java
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

**example for i = 3:**

![Diagram showing the process of adding an item to an ArrayList at position 3](image)

Removing an Item from an ArrayList

• Removing item i (shifting items i+1, i+2, ... to the left by one):

```java
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }

    Object removed = items[i];
    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        items[j + 1] = items[j];
    }

    items[length - 1] = null;
    length--;
    return removed;
}
```

**example for i = 1:**

![Diagram showing the process of removing an item from an ArrayList at position 1](image)
Getting an Item from an ArrayList

• Getting item i (without removing it):

```java
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    return items[i];
}
```

toString() Method for the ArrayList Class

```java
public String toString() {
    String str = "{";
    if (length > 0) {
        for (int i = 0; i < length - 1; i++) {
            str = str + items[i] + ", ";
        }
        str = str + items[length - 1];
    }
    str = str + "}"
    return str;
}
```

• Produces a string of the following form:

```
{items[0], items[1], … }
```

• Why is the last item added outside the loop?

• Why do we need the if statement?
Implementing a List Using a Linked List

```java
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```

- Differences from the linked lists we used for strings:
  - we "embed" the linked list inside another class
    - users of our LLList class won't actually touch the nodes
  - we use non-static methods instead of static ones
    - `myList.length()` instead of `length(myList)`
  - we use a special `dummy head node` as the first node

Using a Dummy Head Node

- The dummy head node is always at the front of the linked list.
  - like the other nodes in the linked list, it's of type `Node`
  - it does not store an item
  - it does not count towards the length of the list

- Using it allows us to avoid special cases when adding and removing nodes from the linked list.

- An empty LLList still has a dummy head node:
An Inner Class for the Nodes

```java
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;
        private Node(Object i, Node n) {
            item = i;
            next = n;
        }
    }...
}
```

- We make Node an `inner class`, defining it within LLList.
  - allows the LLList methods to directly access Node's private fields, while restricting access from outside LLList
  - the compiler creates this class file: LLList$Node.class
- For simplicity, our diagrams may show the items inside the nodes.

![Diagram showing the items inside the nodes]

Other Details of Our LLList Class

```java
public class LLList implements List {
    private class Node {
        // see previous slide
    }
    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
    ...}
```

- Unlike ArrayList, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.
- The linked list can grow indefinitely, so the list is never full!
Getting a Node

- Private helper method for getting node i
- to get the dummy head node, use i = -1

```java
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = null;
    int travIndex = -1;
    while (travIndex < length) {
        travIndex++;
    }
    return trav;
}
```

Example for i = 1:

- head: null
- length: 3
- item: ["how", "are", "you"]

Getting an Item

```java
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }

    Node n = getNode(i);
    return ________;
}
```

Example for i = 1:

- head: null
- length: 3
- item: ["how", "are", "you"]
Adding an Item to an LList

```java
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    }
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i - 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;
    length++;
    return true;
}
```

- This works even when adding at the front of the list (i = 0):

```
• item  "how"
  prevNode
  newNode
  node  "hi!"

head ——— item ——— null ——— “how” ——— “are” ——— “you” ——— null
  length  4
  next
```

addItem() Without a Dummy Head Node

```java
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOf BoundsException();
    }
    Node newNode = new Node(item, null);

    if (i == 0) {                // case 1: add to front
        newNode.next = head;
        head = newNode;
    } else {                     // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }
    length++;
    return true;
}
```

(Gray code shows what we would need to add if we didn’t have a dummy head node)
Removing an Item from an LLList

```java
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Node prevNode = getNode(i - 1);
    Object removed = prevNode.next.item;
    length--;
    return removed;
}
```

- This works even when removing the first item (i = 0):

```
toString() Method for the LLList Class
```

```java
public String toString() {
    String str = "{";
    // what should go here?
    str = str + "}";
    return str;
}
```
### Efficiency of the List ADT Implementations

- **n** = number of items in the list

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LList</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>getItem()</td>
<td>only one case:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addItem()</td>
<td><strong>best:</strong></td>
<td><strong>best:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>worst:</strong></td>
<td><strong>worst:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>average:</strong></td>
<td><strong>average:</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Efficiency of the List ADT Implementations (cont.)

- **n** = number of items in the list

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LList</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>removeItem()</td>
<td><strong>best:</strong></td>
<td><strong>best:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>worst:</strong></td>
<td><strong>worst:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>average:</strong></td>
<td><strong>average:</strong></td>
<td></td>
</tr>
<tr>
<td>space</td>
<td>efficiency</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counting the Number of Occurrences of an Item

```java
class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        for (int i = 0; i < l.length(); i++) {
            Object itemAt = l.getItem(i);
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    }
}
```

- This method works fine if we pass in an `ArrayList` object.
  - time efficiency (as a function of the length, n) = ?
- However, it's **not** efficient if we pass in an `LLList`.
  - each call to `getItem()` calls `getNode()`
  - to access item 0, `getNode()` accesses 2 nodes (dummy + node 0)
  - to access item 1, `getNode()` accesses 3 nodes
  - to access item i, `getNode()` accesses i+2 nodes
  - $2 + 3 + ... + (n+1) = ?$

One Solution: Make `numOccur()` an `LLList` Method

```java
class LLList {
    public int numOccur(Object item) {
        int numOccur = 0;
        Node trav = head.next;  // skip the dummy head node
        while (trav != null) {
            if (trav.item.equals(item)) {
                numOccur++;
            }
            trav = trav.next;
        }
        return numOccur;
    }
}
```

- Each node is only visited once, so we get $O(n)$ efficiency.
- Problem: we can’t anticipate all types of operations that clients may wish to perform.
- We’d like to provide the general ability to iterate over the list.
A Better Solution: Provide an Iterator

```java
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        while (iter.hasNext()) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    }
}
```

- We add an `iterator()` method to the `List` interface.
  - it returns a separate `iterator object` that can efficiently iterate over the items in the list
- The iterator has two key methods:
  - `hasNext()`: tells us if there are items we haven't seen yet
  - `next()`: returns the next item and advances the iterator

An Interface for List Iterators

- Here again, the interface only includes the method headers:
  ```java
  public interface ListIterator { // in ListIterator.java
      boolean hasNext();
      Object next();
  }
  ```
- We can then implement this interface for each type of list:
  - `LLListIterator` for an iterator that works with `LLLists`
  - `ArrayListIterator` for an iterator for `ArrayLists`
- We use the interfaces when declaring variables in client code:
  ```java
  public class MyClass {
      public static int numOccur(List l, Object item) {
          int numOccur = 0;
          ListIterator iter = l.iterator();
          ...
      }
  }
  ```
  - doing so allows the code to work for any type of list!
Using an Inner Class for the Iterator

public class LLList {
    private Node head;
    private int length;

    private class LLListIterator implements ListIterator {
        private Node nextNode;  // points to node with the next item
        public LLListIterator() {
            nextNode = head.next;  // skip over dummy head node
        }
        ...
        public ListIterator iterator() {
            return new LLListIterator();
        }
        ...
    }

    • Using an inner class gives the iterator access to the list’s internals.
    • The iterator() method is an LLList method.
        • it creates an instance of the inner class and returns it
        • its return type is the interface type
        • so it will work in the context of client code

Full LLListIterator Implementation

private class LLListIterator implements ListIterator {
    private Node nextNode;  // points to node with the next item
    public LLListIterator() {
        nextNode = head.next;  // skip over the dummy head node
    }
    public boolean hasNext() {
        return (nextNode != null);
    }
    public Object next() {
        // throw an exception if nextNode is null
        Object item = _______________;  // how
        nextNode = _______________;  // are
        return item;
    }
}
Stack ADT

- A stack is a 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:
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
  - peek: get the item at the top of the stack, but don’t remove it
  - isEmpty: test if the stack is empty
  - isFull: test if the stack is full
- Example: a stack of integers

```
start: 15 7
push 8: 15 7
pop: 15 7
pop: 7 7
push 3: 7 7
```

A Stack Interface: First Version

```java
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

- push() returns false if the stack is full, and true otherwise.
- pop() and peek() take no arguments, because we know that we always access the item at the top of the stack.
  - return null if the stack is empty.
- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - encapsulation allows us to ensure that our stacks are only manipulated in appropriate ways.
Implementing a Stack Using an Array: First Version

```java
public class ArrayStack implements Stack {
    private Object[] items;    // index of the top item
    private int top;            // index of the top item

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

    ...
}
```

- Example: the stack

```
<table>
<thead>
<tr>
<th></th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>7</td>
</tr>
</tbody>
</table>
```

- Items are added from left to right (top item = the rightmost one).
- `push()` and `pop()` won't require any shifting!

Collection Classes and Data Types

```java
public class ArrayStack implements Stack {
    private Object[] items;    // index of the top item
    private int top;            // index of the top item

    ...
}
```

- So far, our collections have allowed us to add objects of any type.
  - `ArrayStack s1 = new ArrayStack(4);
    s1.push(7);    // 7 is turned into an Integer object for 7
    s1.push("hi");
    String item = s1.pop();           // won't compile
    String item = (String)s1.pop();   // need a type cast

- We'd like to be able to limit a given collection to one type.
  - `ArrayStack<String> s2 = new ArrayStack<String>(10);
    s2.push(7);               // won't compile
    s2.push("hello");
    String item = s2.pop();   // no cast needed!
```
Limiting a Stack to Objects of a Given Type

- We can do this by using a *generic* interface and class.
- Here's a generic version of our stack interface:
  ```java
  public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
  }
  ```
- It includes a *type variable* `T` in its header and body.
- used as a placeholder for the actual type of the items

A Generic ArrayStack Class

```java
public class ArrayStack<T> implements Stack<T> {
  private T[] items;
  private int top;     // index of the top item
  public boolean push(T item) {
    ...}
  }
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.
- When we create an `ArrayStack`, we specify the type of items that we intend to store in the stack:
  ```java
  ArrayStack<String> s1 = new ArrayStack<String>(10);
  ArrayStack<Integer> s2 = new ArrayStack<Integer>(25);
  ```
- We can still allow for a mixed-type collection:
  ```java
  ArrayStack<Object> s3 = new ArrayStack<Object>(20);
  ```
Using a Generic Class

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

public class ArrayStack<String> {
    private String[] items;
    private int top;
    ...
    public boolean push(String item) {
        ...
    }
}

public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;
    ...
    public boolean push(Integer item) {
        ...
    }
}

ArrayStack<String> s1 = new ArrayStack<String>(10);
ArrayStack<Integer> s2 = new ArrayStack<Integer>(25);

ArrayStack Constructor

- Java doesn't allow you to create an object or array using a type variable. Thus, we cannot do this:
  ```java
  public ArrayStack(int maxSize) {
      items = new T[maxSize]; // not allowed
      top = -1;
  }
  ```

- To get around this limitation, we create an array of type Object and cast it to be an array of type T:
  ```java
  public ArrayStack(int maxSize) {
      items = (T[])new Object[maxSize];
      top = -1;
  }
  ```

- The cast generates a compile-time warning, but we'll ignore it.
- Java's built-in ArrayList class takes this same approach.
Testing if an ArrayStack is Empty or Full

• Empty stack:

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

• Full stack:

```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an ArrayStack

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

```java
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

• peek just returns `items[top]` without decrementing `top`.

Implementing a Generic Stack Using a Linked List

```java
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    ...

    • Example: the stack

    • Things worth noting:
      • our LLStack class needs only a single field: a reference to the first node, which holds the top item
      • top item = leftmost item (vs. rightmost item in ArrayStack)
      • we don’t need a dummy node
        • only one case: always insert/delete at the front of the list!
```
Other Details of Our LLStack Class

```java
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
    }
    private Node top;
    public LLStack() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public boolean isFull() {
        return false;
    }
}
```

- The inner Node class uses the type parameter T for the item.
- We don’t need to preallocate any memory for the items.
- The stack is never full!

**LLStack.push**

```
public boolean push(T item) {
}
```
public T pop() {
    if (isEmpty())
        return null;
    T removed = __________________;
}

public T peek() {
    if (isEmpty())
        return null;
    return top.item;
}
Applications of Stacks

• The runtime stack in memory
• Converting a recursive algorithm to an iterative one
  • use a stack to emulate the runtime stack
• Making sure that delimiters (parens, brackets, etc.) 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: $5 \times [3 + \{(5 + 16 - 2)\}$

Queue ADT

• A queue is a 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)
    • you can only access the item that is currently at the front
  • Operations:
    • insert: add an item at the rear of the queue
    • remove: remove the item at the front of the queue
    • peek: get the item at the front of the queue, but don’t remove it
    • isEmpty: test if the queue is empty
    • isFull: test if the queue is full
  • Example: a queue of integers
    start: 12 8
    insert 5: 12 8 5
    remove: 8 5
Our Generic Queue Interface

```java
public interface Queue<T> {
    boolean insert(T item);
    T remove();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `insert()` returns `false` if the queue is full, and `true` otherwise.
- `remove()` and `peek()` take no arguments, because we know that we always access the item at the front of the queue.
  - `return null` if the queue is empty.
- Here again, we will use encapsulation to ensure that the data structure is manipulated only in valid ways.

Implementing a Queue Using an Array

```java
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;
    ...
}
```

- Example:
  ```
  ArrayQueue queue = new ArrayQueue();
  queue.insert(73);
  queue.insert(25);
  queue.insert(51);
  System.out.println(queue.remove()); // 73
  System.out.println(queue.remove()); // 25
  System.out.println(queue.remove()); // 51
  ```

- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue
Avoiding the Need to Shift Items

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

  example: a queue of integers:
  \[
  \begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
  \text{front} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} \\
  54 & 4 & 21 & 17 & 89 & 65 & \phantom{6} & \phantom{6} & \phantom{6} \\
  \end{array}
  \]

  the same queue after removing two items and inserting one:
  \[
  \begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
  \text{front} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} \\
  & 21 & 17 & 89 & 65 & 43 & \phantom{6} & \phantom{6} & \phantom{6} \\
  \end{array}
  \]

  to insert two or more additional items, would need to shift items left

- Solution: maintain a circular queue. When we reach the end of the array, we wrap around to the beginning.

  the same queue after inserting two additional items:
  \[
  \begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
  \text{rear} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} \\
  5 & 21 & 17 & 89 & 65 & 43 & 81 & \phantom{6} & \phantom{6} \\
  \end{array}
  \]

A Circular Queue

- To get the front and rear indices to wrap around, we use the modulus operator (%).

- \( x \% y = \) the remainder produced when you divide \( x \) by \( y \)
  - examples:
    - \( 10 \% 7 = 3 \)
    - \( 36 \% 5 = 1 \)

- Whenever we increment front or rear, we do so modulo the length of the array.
  \[
  \text{front} = (\text{front} + 1) \% \text{items.length};
  \]
  \[
  \text{rear} = (\text{rear} + 1) \% \text{items.length};
  \]

- Example:
  \[
  \begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
  \text{front} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} & \phantom{6} \\
  & 21 & 17 & 89 & 65 & 43 & 81 & \phantom{6} & \phantom{6} \\
  \end{array}
  \]

  items.length = 8, rear = 7

  before inserting the next item: \( \text{rear} = (7 + 1) \% 8 = 0 \)

  which wraps \( \text{rear} \) around to the start of the array
**Testing if an ArrayQueue is Empty**

- Initial configuration:
  
  \[
  \begin{array}{c|c}
  \text{rear} & \text{front} \\
  \hline
  -1 & 0 \\
  \end{array}
  \]

- We increment rear on every insertion, and we increment front on every removal.
  
  \[
  \begin{array}{c|c|c}
  \text{rear} & \text{front} & \text{after one insertion:} \\
  \hline
  15 & & \\
  \end{array}
  \]

  \[
  \begin{array}{c|c|c}
  \text{rear} & \text{front} & \text{after two insertions:} \\
  \hline
  15 & 32 & \\
  \end{array}
  \]

  \[
  \begin{array}{c|c}
  \text{rear} & \text{front} \\
  \hline
  32 & \\
  \end{array}
  \]

  \[
  \begin{array}{c|c}
  \text{rear} & \text{front} \\
  \hline
  & \\
  \end{array}
  \]

- The queue is empty when rear is one position "behind" front:
  
  \[((\text{rear} + 1) \mod \text{items.length}) == \text{front}\]

**Testing if an ArrayQueue is Full**

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

  \[
  \begin{array}{c|c|c|c|c|c|c|c|c}
  \text{rear} & \text{front} & \text{after one removal:} \\
  \hline
  5 & 21 & 17 & 89 & 65 & 43 & 81 & \\
  \end{array}
  \]

- This is why we maintain numItems!

  ```java
  public boolean isEmpty() {
      return (numItems == 0);
  }
  
  public boolean isFull() {
      return (numItems == items.length);
  }
  ```
Constructor

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

Inserting an Item in an ArrayQueue

- We increment rear before adding the item:

    before:  
    
    rear  

    after:  
    
    rear

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

```java
public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _________________;
    items[front] = null;
    front = (front + 1) % items.length;
    numItems--;
    return removed;
}
```

Implementing a Queue Using a Linked List

```java
public class LLQueue<T> implements Queue<T> {
    private Node front;    // front of the queue
    private Node rear;     // rear of the queue
...
}
```

- Example:

```
queue
  front
  rear
  variable of type LLQueue
  LLQueue object
  item
  "hi"
  "how"
  "are"
  "you"
  null
```

- Because a linked list can be easily modified on both ends, we don’t need to take special measures to avoid shifting items, as we did in our array-based implementation.
Other Details of Our LLQueue Class

```java
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;...
    }

    private Node front;
    private Node rear;

    public LLQueue() {
        front = null;
        rear = null;
    }

    public boolean isEmpty() {
        return (front == null);
    }

    public boolean isFull() {
        return false;
    }
    ...}

• Much simpler than the array-based queue!
```

Inserting an Item in an Empty LLQueue

```
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        rear.next = newNode;
        rear = newNode;
    }
    return true;
}
```

The next field in the newNode will be null in either case. Why?
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty()) {
    } else {
        return true;
    }
}

Inserting an Item in a Non-Empty LLQueue

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _________________;
    if (front == rear) {     // removing the only item
        } else {
            } 
    return removed;
}

Removing from an LLQueue
Removing from an LLQueue with One Item

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _________________;
    if (front == rear) {     // removing the only item
        return removed;
    }
    else {
        return removed;
    }
}

Removing from an LLQueue with Two or More Items

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _________________;
    if (front == rear) {     // removing the only item
        return removed;
    }
    else {
        return removed;
    }
}
Efficiency of the Queue Implementations

<table>
<thead>
<tr>
<th></th>
<th>ArrayQueue</th>
<th>LLQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>peek()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>space efficiency</td>
<td>$O(m)$ where $m$ is the anticipated maximum number of items</td>
<td>$O(n)$ where $n$ is the number of items currently in the queue</td>
</tr>
</tbody>
</table>

Applications of Queues

- first-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- simulations of banks, supermarkets, airports, etc.
Lists, Stacks, and Queues in Java's Class Library

- Lists:
  - interface: `java.util.List<T>`
  - slightly different methods, some extra ones
  - array-based implementations: `java.util.ArrayList<T>`
    `java.util.Vector<T>`
    - the array is expanded as needed
    - `Vector` has extra non-List methods
  - linked-list implementation: `java.util.LinkedList<T>`
    - `addLast()` provides $O(1)$ insertion at the end of the list

- Stacks: `java.util.Stack<T>`
  - extends `Vector` with methods that treat a vector like a stack
  - problem: other `Vector` methods can access items below the top

- Queues:
  - interface: `java.util.Queue<T>`
  - implementation: `java.util.LinkedList<T>`.