### 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>
A List as an Abstract Data Type

• list = a sequence of items that supports at least the following functionality:
  • accessing an item at an arbitrary position in the sequence
  • adding an item at an arbitrary position
  • removing an item at an arbitrary position
  • determining the number of items in the list (the list’s length)
• ADT: specifies what a list will do, without specifying the implementation

Review: Specifying an ADT Using an Interface

• Recall that in Java, we can use an interface to specify an ADT:
  ```java
  public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    int length();
    ...
  }
  ```
• We make any implementation of the ADT a class that implements the interface:
  ```java
  public class MyList implements List {
    ...
  }
  ```
• This approach allows us to write code that will work with different implementations of the ADT:
  ```java
  public static void processList(List l) {
    for (int i = 0; i < l.length(); i++) {
      Object item = l.getItem(i);
      ...
    }
  }
  ```
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();
}
```

- We include an `isFull()` method to test if the list already has the maximum number of items.
- 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) {
        items = new Object[maxSize];
        length = 0;
    }

    public int length() {
        return length;
    }

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

    //... ...
}
```

- Sample list:

```
list: [null, null ...]

items: [null, null ...

length: 2
```

`list` is an `ArrayList` object, and `items` is an `ArrayList` object. The `length` variable holds the number of items in the list.
Adding an Item to an \textit{ArrayList}

• Adding at position \(i\) (shifting items \(i, i+1, \ldots\) to the right by one):
  
  ```java
  public boolean addItem(Object item, int i) {
      if (i < 0 || i > length)
          throw new IndexOutOfBoundsException();
      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;
  }
  ```

Other \textit{ArrayList} Methods

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

• Removing item \(i\) (shifting items \(i+1, i+2, \ldots\) to the left by one):
  
  ```java
  public Object removeItem(int i) {
      if (i < 0 || i >= length)
          throw new IndexOutOfBoundsException();
  }
  ```
Converting an ArrayList to a String

• The `toString()` method is designed to allow objects to be displayed in a human-readable format.

• This method is called implicitly when you attempt to print an object or when you perform string concatenation:
  ```java
  ArrayList l = new ArrayList();
  System.out.println(l);
  String str = "My list: " + l;
  System.out.println(str);
  ```

• A default version of this method is inherited from the `Object` class.
  • returns a `String` consisting of the type of the object and a hash code for the object.

• It usually makes sense to override the default version.

```
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;    // dummy head node
    private int length;
    ...}
```

- Sample list:

- Differences from the linked list we used for strings:
  - we “embed” the linked list inside another class
    - users of our LLList class will never actually touch the nodes
    - users of our StringNode class hold a reference to the first node
  - we use a dummy head node
  - we use instance methods instead of static methods
    - `myList.length()` instead of `length(myList)`

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

- An empty LLList still has a dummy head node:

- Using a dummy head node allows us to avoid special cases when adding and removing nodes from the linked list.
An Inner Node Class

```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;
        }
    }

    private Node head;
    private int length;

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

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

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

Other Details of Our `LLList` Class

```java
public class LLList implements List {
    private class Node {
        ...
    }

    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++ < i) {} // or travIndex++ < 0;
    return trav;
}
```

**Adding an Item to an LLList**

```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 (\( i == 0 \)):
addItem() Without a Dummy Head Node

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

    Node newNode = new Node(item, null);
    if (i == 0) { // case 1: add to front
        newNode.next = first;
        first = newNode;
    } else { // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

(instead of a reference named head to the dummy head node, this implementation maintains a reference named first to the first node, which does hold an item).

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;

    prevNode.next = prevNode.next.next;
    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;
}
```

Counting the Number of Occurrences of an Item

• One possible approach:
  ```java
  public 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;
      }
  }
  ```

• Problem: for LLList objects, each call to getItem() starts at the head of the list and traverses to item i.
  • to access item 0, access 1 node
  • to access item 1, access 2 nodes
  • to access item i, access i+1 nodes
  • if length = n, total nodes accessed = 1 + 2 + ... + n = O(n²)
Solution 1: Make `numOccur()` an `LLList` Method

```java
public 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 the # of accesses = n = O(n)
- Problem: we can’t anticipate all of the types of operations that users may wish to perform.
- We would like to give users the general ability to iterate over the list.

Solution 2: Give Access to the Internals of the List

- Make our private helper method `getNode()` a public method.
- Make `Node` a non-inner class and provide getter methods.
- This would allow us to do the following:

```java
public class MyClass {
    public static int numOccur(LLList l, Object item) {
        int numOccur = 0;
        Node trav = l.getNode(0);
        while (trav != null) {
            Object itemAt = trav.getItem();
            if (itemAt.equals(item))
                numOccur++;
            trav = trav.getNext();
        }
        return numOccur;
    }
}
```

- What’s wrong with this approach?
Solution 3: Provide an Iterator

- An iterator is an object that provides the ability to iterate over a list without violating encapsulation.

- Our iterator class will implement the following interface:
  ```java
  public interface ListIterator {
      // Are there more items to visit?
      boolean hasNext();

      // Return next item and advance the iterator.
      Object next();
  }
  ```

- The iterator starts out prepared to visit the first item in the list, and we use `next()` to access the items sequentially.

- Ex: position of the iterator is shown by the cursor symbol (|) after the iterator is created:
  ```plaintext
  after the iterator i is created: | "do" "we" "go" ...
  after calling i.next(), which returns "do": "do" | "we" "go" ...
  after calling i.next(), which returns "we": "do" "we" | "go" ...
  ```

numOccur() Using 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;
    }
}
```

- The `iterator()` method returns an iterator object that is ready to visit the first item in the list. (Note: we also need to add the header of this method to the `List` interface.)

- Note that `next()` does two things at once:
  - gets an item
  - advances the iterator.
Using an Inner Class for the Iterator

```java
public class LLList {
    public ListIterator iterator() {
        return new LLListIterator();
    }

    private class LLListIterator implements ListIterator {
        private Node nextNode;
        private Node lastVisitedNode;

        public LLListIterator() {
            nextNode = head.next;    // skip over head node
            lastVisitedNode = null;
        }
        ...
    }
}
```

- Using a inner class gives the iterator access to the list’s internals.
- Because `LLListIterator` is a private inner class, methods outside `LLList` can’t create `LLListIterator` objects or have variables that are declared to be of type `LLListIterator`.
- Other classes use the `interface name` as the declared type, e.g.:
  ```java
  ListIterator iter = l.iterator();
  ```

**LLListIterator Implementation**

```java
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;

    public LLListIterator() {
        nextNode = head.next;    // skip over head node
        lastVisitedNode = null;
    }
    ...
}
```

- Two instance variables:
  - `nextNode` keeps track of the next node to visit
  - `lastVisitedNode` keeps track of the most recently visited node
    - not needed by `hasNext()` and `next()`
    - what iterator operations might we want to add that `would` need this reference?
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;
    public LLListIterator() {
        nextNode = head.next;  // skip over dummy node
        lastVisitedNode = null;
    }
    public boolean hasNext() {
        return (nextNode != null);
    }
    public Object next() {
        if (nextNode == null)
            throw new NoSuchElementException();
        Object item = nextNode.item;
        lastVisited = nextNode;
        nextNode = nextNode.next;
        return item;
    }
}

More About Iterators

• In theory, we could write list-iterator methods that were methods of the list class itself.

• Instead, our list-iterator methods are encapsulated within an iterator object.
  • allows us to have multiple iterations active at the same time:
    ListIterator i = l.iterator();
    while (i.hasNext()) {
        ListIterator j = i.iterator();
        while (j.hasNext()) {
            ...
        }
    }

• Java’s built-in collection classes all provide iterators.
  • LinkedList, ArrayList, etc.
  • the built-in Iterator interface specifies the iterator methods
    • they include hasNext() and next() methods like ours
Efficiency of the List Implementations

<table>
<thead>
<tr>
<th></th>
<th>ArrayList</th>
<th>LList</th>
</tr>
</thead>
<tbody>
<tr>
<td>getItem()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addItem()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeItem()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n = number of items in the list

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: 15 7
push 3: 3 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 manipulated only in ways that are consistent with what it means to be a stack.

Implementing a Stack Using an Array: First Version

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

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

    // ... (other methods)
}
```

- Example: the stack would be represented as follows:

```
stack: [null, null, null, 7, 15, 15, ...]
items: [null, null, null, 7, 15, 15, ...]
```

- Items are added from left to right. The instance variable `top` stores the index of the item at the top of the stack.
Limiting a Stack to Objects of a Given Type

- We can do this by using a *generic* interface and class.
- Here is 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.
  - This type variable is used as a placeholder for the actual type of the items on the stack.

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 object) {
    }
}
```

- 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<Integer> s1 = new ArrayStack<Integer>(10);
  ArrayStack<String> s2 = new ArrayStack<String>(5);
  ArrayStack<Object> s3 = new ArrayStack<Object>(20);
  ```
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;
}
```

(This doesn't produce a `ClassCastException` at runtime, because in the compiled version of the class, `T` is replaced with `Object`.)

- The cast generates a compile-time warning, but we'll ignore it.
- Java's built-in `ArrayList` class takes this same approach.

More on Generics

- When a collection class uses the type `Object` for its items, we often need to use casting:

```java
LLList list = new LLList();
list.addItem("hello");
list.addItem("world");
String item = (String)list.getItem(0);
```

- Using generics allows us to avoid this:

```java
ArrayStack<String> s = new ArrayStack<String>;
s.push("hello");
s.push("world");
String item = s.pop(); // no casting needed
```
### Testing if an ArrayStack is Empty or Full

- **Empty stack:**
  ```java
def isEmpty() {
    return (top == -1);
}
```

- **Full stack:**
  ```java
def isFull() {
    return (top == items.length - 1);
}
```

### Pushing an Item onto an ArrayStack

- **We increment top before adding the item:**
  ```java
def push(T item) {
    if (isFull())
      return false;
    top++;
    items[top] = item;
    return true;
}
```
ArrayStack pop() and peek()

- pop: need to get items[top] before we decrement top.

```
before:
0 1 ... top

after:
0 1 ...
```

```
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.

toString() Method for the ArrayStack Class

- Assume that we want the method to show us everything in the stack – returning a string of the form

```
"{top, one-below-top, two-below-top, ... bottom}"
```

```
public String toString() {
    String str = "{";
    // what should go here?
    str = str + "}"
    return str;
}
```
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

```
    15
    7
```

would be represented as follows:

- Things worth noting:
  - our `LLStack` class needs only a single instance variable—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, because we always insert at the front, and thus the insertion code is already simple

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!
public boolean push(T item) {
}

public T pop() {
    if (isEmpty())
        return null;
    T removed = __________________;
}

public T peek() {
    if (isEmpty())
        return null;
    return top.item;
}
**toString() Method for the LLStack Class**

- Again, assume that we want a string of the form
  
  "\{top, one-below-top, two-below-top, … bottom\}"

```java
public String toString() {
    String str = "\{");
    // what should go here?
    str = str + "}"
    return str;
}
```

**Efficiency of the Stack Implementations**

<table>
<thead>
<tr>
<th></th>
<th>ArrayStack</th>
<th>LLStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>pop()</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 on the stack</td>
</tr>
</tbody>
</table>
Applications of Stacks

- The runtime stack in memory
- Converting a recursive algorithm to an iterative one by using 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: $5^4 \left[ 3 + \{ (5 + 16 - 2) \} \right]$

An Example of Switching Between Implementations

- In the example code for this unit, there is a test program for each type of sequence:
  - ListTester.java, StackTester.java, QueueTester.java
- Each test program uses a variable that has the appropriate interface as its type. For example:
  ```java
  Stack<String> myStack;
  ```
- The program asks you which implementation you want to test, and it calls the corresponding constructor:
  ```java
  if (type == 1)
      myStack = new ArrayStack<String>(10);
  else if (type == 2)
      myStack = new LLStack<String>();
  ```
- This is an example of what principle of object-oriented programming?
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:**
  - 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:
      ```
      front  rear
      54  4  21  17  89  65
      ``
      - the same queue after removing two items and inserting one:
        ```
        front  rear
        21  17  89  65  43
        ```
        - 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:*
      ```
      front  rear
      5  21  17  89  65  43  81
      ```
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 \( \text{front} \) or \( \text{rear} \), 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:

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

  \( \text{items.length} = 8, \text{rear} = 7 \)

  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 Array Queue is Empty

- Initial configuration:

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

- We increment \( \text{rear} \) on every insertion, and we increment \( \text{front} \) on every removal.

  after one insertion:
  \[
  \begin{array}{c|c|c|c|c|c|c}
  \text{front} & \text{rear} \\
  \hline
  15 &  \phantom{0} \\
  \end{array}
  \]

  after two insertions:
  \[
  \begin{array}{c|c|c|c|c|c|c}
  \text{front} & \text{rear} \\
  \hline
  15 & 32 &  \phantom{0} \\
  \end{array}
  \]

  after one removal:
  \[
  \begin{array}{c|c|c|c|c|c|c}
  \text{front} & \text{rear} \\
  \hline
  32 &  \phantom{0} \\
  \end{array}
  \]

  after two removals:
  \[
  \begin{array}{c|c|c|c|c|c|c}
  \text{front} & \text{rear} \\
  \hline
  &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} \\
  \end{array}
  \]

- The queue is empty when \( \text{rear} \) is one position “behind” \( \text{front} \):

  \[
  ((\text{rear} + 1) \% \text{items.length}) = \text{front}
  \]
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!

  example: what if we added one more item to this queue?

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>89</td>
</tr>
<tr>
<td>65</td>
<td>43</td>
</tr>
<tr>
<td>81</td>
<td></td>
</tr>
</tbody>
</table>

- This is why we maintain numItems!

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

Constructor

```java
public ArrayQueue(int maxSize) {
    items = (T[]) new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```
Inserting an Item in an ArrayQueue

- We increment \( \text{rear} \) before adding the item:

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

---

ArrayQueue remove()

- remove: need to get \( \text{items[front]} \) before we increment \( \text{front} \).

```
public T remove() {
    if (isEmpty())
        return null;
    T removed = items[front];
    items[front] = null;
    front = (front + 1) % items.length;
    numItems--;
    return removed;
}
```
Implementing a Queue Using a Linked List

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

- **Example:**

A linked list can be easily modified on both ends, so 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

```
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 = 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 \texttt{LLQueue}

```java
class Node {
    T item;
    Node next;
}

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

The next field in the newNode will be \texttt{null} in either case. Why?

Inserting an Item in a Non-Empty \texttt{LLQueue}

```java
class Node {
    T item;
    Node next;
}

public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty())
        return true;
    else {
        return true;
    }
    return true;
}
```
Removing from an **LLQueue** with One Item

```java
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

```java
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>Method</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</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>
<tr>
<td>efficiency</td>
<td></td>
<td></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.
- breadth-first traversal of a graph or level-order traversal of a binary tree (more on these later)
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>.