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) {
        // code to check for invalid maxSize goes here...
        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

```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);
    }
    ...
}
```

- 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 (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } 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:
  ```plaintext
<table>
<thead>
<tr>
<th>items</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
</tbody>
</table>
  ```
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 (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } 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++;  // set the length
    return true;
}
```

element for i = 3:

<table>
<thead>
<tr>
<th>items</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>items</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

example for i = 3: 0 1 2 3 4 5 6 7 8

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

element for i = 1:

<table>
<thead>
<tr>
<th>items</th>
<th>removed</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>null null null null</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>items</th>
<th>removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>&quot;Dave&quot; &quot;Caitlin&quot; &quot;Cody&quot; &quot;Kylie&quot; &quot;Libby&quot;</td>
</tr>
</tbody>
</table>

example for i = 1: 0 1 2 3 4 5 6 7 8
Getting an Item from an ArrayList

• Getting item i (without removing it):
  public Object getItem(int i) {
    if (i < 0 || i >= length) {
      throw new IndexOutOfBoundsException();
    }
    return items[i];
  }

toString() Method for the ArrayList Class

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;
        }
    }
    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 in 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.

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 item item item
length 3 1 2
```

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 item item item
length 3 1 2
```
Adding an Item to an LLList

```java
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    }
    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 (\(i = 0\)):

```
<table>
<thead>
<tr>
<th>i</th>
<th>length</th>
<th>head</th>
<th>item</th>
<th>prevNode</th>
<th>newNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>4</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>null</td>
<td>&quot;how&quot;</td>
<td>null</td>
<td>&quot;hi!&quot;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>null</td>
<td>&quot;are&quot;</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>null</td>
<td>&quot;you&quot;</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
```

addItem() Without a Dummy Head Node

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

(The 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
class LLList {
    public Object removeItem(int i) {
        if (i < 0 || i >= length) {
            throw new IndexOutOfBoundsException();
        }
        Node prevNode = getNode(i - 1);
        Object removed = prevNode.next.item;
        length--; // what line goes here?
        return removed;
    }

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

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

```
Removed Item: "are you"
```

toString() Method for the LLList Class

```java
class LLList {
    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>Function</th>
<th>ArrayList</th>
<th>LList</th>
<th>best:</th>
<th>worst:</th>
<th>average:</th>
</tr>
</thead>
<tbody>
<tr>
<td>getItem()</td>
<td></td>
<td></td>
<td>only one case:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addItem()</td>
<td><em>best:</em></td>
<td></td>
<td></td>
<td>worst:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>average:</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>ArrayList</th>
<th>LList</th>
<th>best:</th>
<th>worst:</th>
<th>average:</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeItem()</td>
<td></td>
<td></td>
<td><em>best:</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>worst:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>average:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

*n = number of items in the list*

<table>
<thead>
<tr>
<th>Function</th>
<th>ArrayList</th>
<th>LList</th>
<th>best:</th>
<th>worst:</th>
<th>average:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**space efficiency**
Counting the Number of Occurrences of an Item

```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 != null && 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
  - \(2 + 3 + \ldots + (n+1) = ?\)

One Solution: 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 != null && 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 != null && 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

```java
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

```java
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;
    }
    ...
}
```

[Diagram of LLList with iterator]
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:  8  15  7
pop:  15  7
pop:  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 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) {
        // code to check for invalid maxSize goes here...
        items = new Object[maxSize];
        top = -1;
    }
    ...

    • Example: the stack
      15
      7

    • 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
    ...                    
}

s1 ———— items ———— top
   |       |       | 1
    | variable of type ArrayStack
                      ArrayStack object

s1

• 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) {
        ... // code
    }
    ... // code
}
```

- 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) {
        ...
    }
}
```

```
ArrayStack<String> s1 = new ArrayStack<String>(10);
```

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

```
ArrayStack<Integer> s2 = new ArrayStack<Integer>(25);
```

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

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) {
      // code to check for invalid maxSize goes here...
      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) {
      // code to check for invalid maxSize goes here...
      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:

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

- Full stack:

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

Pushing an Item onto an ArrayStack

```java
public boolean push(T item) {
    // code to check for a null item goes here
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```
ArrayStack pop() and peek()

```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;  // THIS IS WHERE THE TOP IS DECREMENTED!
    return removed;
}
```

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

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

- Example: the stack

```
   15
    7
```

- 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

```java
public boolean push(T item) {
    // code to check for a null item goes here
}
```
LLStack pop() and peek()

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

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

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
  • 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: $5 \times [3 + \{(5 + 16 - 2)\}$

• Evaluating arithmetic expressions

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
  
  \[
  \begin{align*}
  \text{start:} & \quad 12 \quad 8 \\
  \text{insert 5:} & \quad 12 \quad 8 \quad 5 \\
  \text{remove:} & \quad 8 \quad 5
  \end{align*}
  \]
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:

  
<table>
<thead>
<tr>
<th>54</th>
<th>4</th>
<th>21</th>
<th>17</th>
<th>89</th>
<th>65</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>rear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  the same queue after removing two items and inserting one:

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

  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:

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

A Circular Queue

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

  \[ x \% y = \text{the remainder produced when you divide } x \text{ by } y \]

  • examples:
    • 10 \% 7 = 3
    • 36 \% 5 = 1

• Whenever we increment front or rear, we do so modulo the length of the array.

  \[
  \begin{align*}
  \text{front} &= (\text{front} + 1) \% \text{items.length}; \\
  \text{rear} &= (\text{rear} + 1) \% \text{items.length};
  \end{align*}
  \]

• Example:

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

  items.length = 8, rear = 7
  before inserting the next item: rear = (7 + 1) \% 8 = 0
  which wraps rear around to the start of the array
Inserting an Item in an ArrayQueue

- We increment rear before adding the item:

before:

\[\begin{array}{cccc}
& & \text{front} & \text{rear} \\
\\text{after:} & & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{front} & \text{rear} & & \\
& & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{front} & \text{rear} & & \\
& & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{front} & \text{rear} & & \\
& & & \\
\end{array} \]

public boolean insert(T item) {
    // code to check for a null item goes here
    if (isFull()) {
        return false;
    }
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}

ArrayQueue remove()

before:

\[\begin{array}{cccc}
& & \text{front} & \text{rear} \\
\\text{after:} & & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{front} & \text{rear} & & \\
& & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{front} & \text{rear} & & \\
& & & \\
\end{array} \]

removed

\[\begin{array}{cccc}
\text{removed} & & & \\
& & & \\
\end{array} \]

\[\begin{array}{cccc}
\text{removed} & & & \\
& & & \\
\end{array} \]

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

```java
public ArrayQueue(int maxSize) {
    // code to check for an invalid maxSize goes here...
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```

- When we insert the first item in a newly created `ArrayQueue`, we want it to go in position 0. Thus, we need to:
  - start `rear` at `-1`, since then it will be incremented to `0` and used to perform the insertion
  - start `front` at `0`, since it is not changed by the insertion

![Queue Diagram]

Testing if an `ArrayQueue` is Empty

- **Initial configuration:**
  - `rear = -1`
  - `front = 0`

- We increment `rear` on every insertion, and we increment `front` on every removal.
  - **after one insertion:**
    - `rear = 1`
    - `front = 0`
  - **after two insertions:**
    - `rear = 2`
    - `front = 1`
  - **after one removal:**
    - `rear = 1`
    - `front = 1`
  - **after two removals:**
    - `rear = 0`
    - `front = 1`

- The queue is empty when `rear` is one position "behind" `front`:
  - 
  
```java
  ((rear + 1) % items.length) == 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?*

  ![Example](image)

- This is why we maintain `numItems`!

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

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:

  ![Diagram](image)

- 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

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

The next field in the newNode will be null in either case. Why?

front null
rear null
item —— "now"
newNode —— null

public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        ...
    } else {
        ...
    }
    return true;
}
public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
    } else {
        return true;
    }
}

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _________________;
    if (front == rear) {     // removing the only item
    } else {
        }}
    return removed;
}
Removing from an LLQueue with One Item

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