The Queue Interface

• A “Queue” is an object that contains a collection of other objects.
• A Queue maintains its contents in order.
• Elements may be “enQueued” onto the back of a Queue.
• Elements may be “deQueued” off of the front of a Queue.
• FIFO: “First in first out”.
Queue Illustration

EnQueue → Datum N → ... → Front: Datum 3 Datum 2 Datum 1 → DeQueue
public interface Queue<D> {
    public boolean isEmpty();
    public D enQueue(D item);
    public D deQueue();
    public D front();
}

Implementations of the Queue Interface

- Singly linked list. (Good!)
- Doubly linked list. (OK, but why?)
- Array. (Interesting, but tricky!)
- ...?...
public class QueueListSL<D> implements Queue<D> {

    private ListSL<D> contents;

    public QueueListSL() {
        contents = new ListSL<D>();
    }

    // ... Omitted ...
}

Implementing a queue as a singly-linked list: We define an instance variable `contents` to hold a `ListSL` object. The `QueueListSL` default constructor initializes `contents` to be an empty list.
public D enQueue(D item) {
    contents.addToTail(item);
    return item;
}

public D deQueue() {
    D temp = contents.headPeek();
    contents.removeFromHead();
    return temp;
}

To implement enQueue and deQueue, we use the methods addToTail and removeFromHead from the ListSL class. Notice what enQueue and deQueue return.
public boolean isEmpty() {
    return contents.isEmpty();
}

public D front() {
    return contents.headPeek();
}

To implement `front` and `isEmpty`, we use the methods `headPeek` and `isEmpty` from the `ListSL` class.
public class QueueArray implements Queue
{
    private static final int INITIAL_SIZE = 10;
    private Object[] contents;
    private int front;
    private int back;
    QueueArray()
    {
        contents = new Object[INITIAL_SIZE];
        front = 0;
        back = 0;
    }
    // . . . Omitted . . . }

Implementing a queue as an array: We define an instance variable contents to hold an array of objects. We also define the integer instance variable front to be the array index of the front of the queue, and define the integer instance variable back to be one array location after the back element of the queue. The QueueArray default constructor initializes contents an array of INITIAL_SIZE, and sets front and back to 0, indicating an empty queue.
Wrapping an Array Around a Circle

Each time we compute an array index, we take its value mod N (value % N), where N is the length of the array.
To enQueue an element into the queue, we first make sure the array is big enough, copying the data to a larger array if necessary. Then we increment back (mod contents.length) and store the new item at array index back. The we return the new item. (Notice that we expand the array before completely filling it. If we were to fill it completely, we could not distinguish a completely full array (front == back) from an empty queue (front == back).
To **deQueue** an element from the queue, we first store the **front** element in a temporary variable `temp`. Then we set the **front** element of the array to **null**, possibly freeing up some memory. Then we increment `top` (mod `contents.length`) recording the fact that the queue is now smaller. Then we return `temp`, which is the element that was on top of the stack.
public boolean isEmpty()
{
    return front == back;
}

public Object front()
{
    return contents[front];
}

Since the queue elements run from array indexes front … back-1, the queue is empty if front==back.

The front element is contents[front].
private void expand()
{
    Object[] newContents = new Object[2*contents.length];
    for (int i = 0; i < contents.length-1; i++)
    {
        newContents[i] = contents[(front+i)%contents.length];
    }
    front = 0;
    back = contents.length-1;
    contents = newContents;
}

The queue elements get copied from contents[front] ...
... contents[back-1], (mod contents.length).
The queue elements get copied to newContents[0] ...
... newContents[contents.length-2].
Binary Tree
Binary Tree Structure

- **Root**
  - **data**

  - **Left Subtree**
  - **Right Subtree**
Defining Tree Data

```java
BTree<String> bTree = new BTree<String>("a",
    new BTree<String>("b",
        new BTree<String>("d"),
        new BTree<String>("e")),
    new BTree<String>("c",
        new BTree<String>("f"),
        new BTree<String>("g")));
```
public class BTree<D> {
    public D data;
    public BTree<D> left, right;

    public BTree(D data) {
        this(data, null, null);
    }

    public BTree(D data, BTree<D> left, BTree<D> right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}

A **Btree** is an object with a D **data**, a BTree **left** (subtree) and a BTree **right** (subtree). Either or both subtrees could be **null**. If both are **null**, the object is a leaf of the tree.
Tree Traversal

• Pre-Order:
  – Each node comes before its children.
  – All left descendants come before right descendants.

• In-Order:
  – The left child of a node comes before the node.
  – The right child of a node comes after the node.

• Post-Order:
  – All left descendants come before right descendants.
  – Each node comes after its children.
public void preOrderPrint(BTree<String> bTree) {
    if (bTree==null) return;
    else {
        print(bTree.data+" ");
        preOrderPrint(bTree.left);
        preOrderPrint(bTree.right);
    }
}
Pre-Order:   a b d e c f g
In-Order Traversal

```java
class InOrderPrint
```
```java
public void inOrderPrint(BTree<String> bTree) {
    if (bTree == null) return;
    else {
        inOrderPrint(bTree.left);
        print(bTree.data + " ");
        inOrderPrint(bTree.right);
    }
}
```
In-Order:  d b e a f c g
public void postOrderPrint(BTree<String> bTree) {
    if (bTree == null) return;
    else {
        postOrderPrint(bTree.left);
        postOrderPrint(bTree.right);
        print(bTree.data + " ");
    }
}

Post-Order Traversal
Post-Order: d e b f g c a
Iterative Ordering Algorithms

• Stack Ordering:
  – Similar to pre-ordering.
  – But with right and left reversed.

• Queue Ordering:
  – Level by level.
  – From the root to the leaves.
Stack-Order Traversal

```java
public void stackOrderPrint(BTree<String> bTree) {
    Stack<BTree<String>> theStack = new StackListSL<BTree<String>>() {
        theStack.push(bTree);
        while (!theStack.isEmpty()) {
            BTree<String> bT = theStack.pop();
            print(bT.data + " ");
            if (bT.left != null) theStack.push(bT.left);
            if (bT.right != null) theStack.push(bT.right);
        }
    }
```
Stack-Order:   a c g f b e d
public void queueOrderPrint(BTree<String> bTree) {
    Queue<BTree<String>> theQueue = new QueueListSL<BTree<String>>();
    theQueue.enQueue(bTree);
    while (!theQueue.isEmpty()) {
        BTree bT = theQueue.deQueue();
        print(bT.data + " ");
        if (bT.left != null) theQueue.enQueue(bT.left);
        if (bT.right != null) theQueue.enQueue(bT.right);
    }
}
Queue-Order: a b c d e f g