**Day 27 – Stacks, Queues, Linked Lists**

**Introduction**

Stacks, queues, and linked lists provide ways to organize data for certain kinds of tasks. Each can be thought of as a box into which you can place data and from which you can take data. They differ in how they organize the data inside the box and how you access the data.

**Stacks**

To understand stacks, think of a **stack** of plates. To put data onto the stack, you place it on a plate and then place the plate on top of a stack of plates. As more data is added, the stack grows. When you wish to take a plate off, you must take off the plate on top before you take any other plates off. Thus, the first data to come off the stack is the last to go on … **LIFO** … **last in, first out**. This is the nature of a stack.

A common abstraction of a stack involves declaring the stack, and then performing operations (method calls) on the stack: **push**, **pop**, **peek**, and **empty**. **push** adds a new value to the top of the stack. **pop** removes one value from the stack and returns it to the caller. **peek** returns the top value to the caller without removing it. **empty** allows the user to test if the stack is empty or not.

Run **Stack.jar** to see how a stack works. It allows you to push new values on the stack, pop values off, and peek at the top item without popping.

**Stack Usage**

So, where are stacks used? Any place you might want to reverse the order of data. For example:

* In a browser … the browser keeps two stacks of the pages you have visited: a reverse stack and a forward stack.
  + The location (URL) of each new page you visit is pushed on the reverse stack.
  + The back button pops a URL from the reverse stack and displays it, and pushes it on the forward stack
  + For the forward button a URL is popped off the forward stack, displayed, and pushed on the reverse stack.
* In programming languages. Each time you call a method, that methods “activation record” is pushed on a stack. The activation record includes
  + Where to return to when we return from the method
  + The parameters to the method
  + Room for all local variables used in the method

This way, each call to a method is kept separate, with its own local variables. When it is time to return from the method, we simply pop its activation record off the stack and resume the activation record of the calling method.

**Stack Exercise**

private int[] myArray = { 5, -1, 10, 23, 4, 8, 2, 25, 7, 0 };

Stack<Integer> myStack = new Stack<Integer>();

To implement a stack for yourself, look at the Javadoc for Stack, and write a program called **TestStack.java** that pushes all of the items in myArray onto the stack, prints out the contents of the stack once all the items have been pushed onto it, and then pop all the items off the stack (print each item individually as you pop them) while testing to see if the stack is empty or not. **You should not use the size of myArray to pop items off of the stack, only to push them onto the stack**.

**Queues**

Queues implement waiting lines. People get in line at the tail of the line and are served when they get to the head of the line. The order that people are served is **FIFO … first in, first out**. This is the primary difference between Stack and Queues.

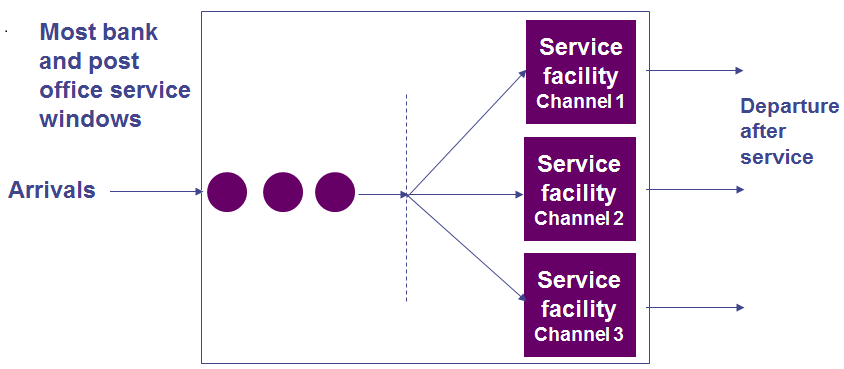
The abstraction for queues usually includes operations (methods) to *enqueue* an item (**add** it to the tail), *dequeue* an item (**remove** it from the head of the queue), **peek** at the value at the head of the queue without taking it off the queue, and testing to see if the queue is **empty**.

Run **Queue.jar** to see how a queue works. It allows you to enqueue (**add**)new values to the end of the queue, dequeue (**remove**) values from the head of the queue, and **peek** at the head without removing it.

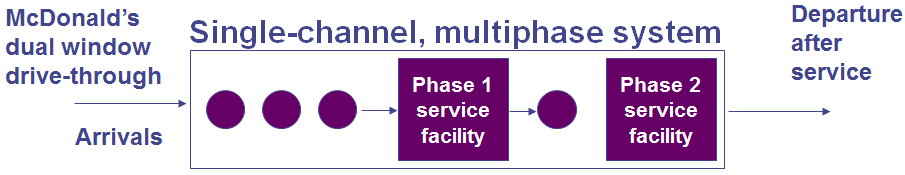
**Queue Usage**

So, where are queues used? Any place you want to keep the order of data. For example:

* Queues are often used to keep track of people in line, waiting for service. These can become more complex. For example, consider waiting in line at a bank or the post office, where there are several people who can serve you. There is one line (queue) but several servers.



There can also be more than one phase. This is what happens at some fast food drive up windows, where there are two windows. You place your order and pay at the first, then receive your order at the second.



Here we need two queues, for the two phases.

**Queue Exercise**

private int[] myArray = { 5, -1, 10, 23, 4, 8, 2, 25, 7, 0 };

Queue<Integer> myQueue = new LinkedList<Integer>();

To implement a queue for yourself, look at the Javadoc for Queue, and write a program called **TestQueue.java** that enqueues all of the items in myArray, prints out the contents of the queue once all the items have been added, and then removes all the items from the queue (print each item individually as you remove them) while testing to see if the queue is empty or not, using the queue’s **size** method. **You should not use the size of myArray to remove items from the queue, only to add them to the queue**.

Note that to create a new Queue object, you actually need to use a new LinkedList since Queue is implemented as a LinkedList. This is a little confusing, but this is how Java does it. You’ll want to look at both the Queue and LinkedList in Java Docs for details on available method calls.

**Linked Lists**

Linked lists are the “ultimate” data structure, in terms of ordering data. They are flexible and quite general. However, they come with a lot of overhead and can be quite expensive if this flexibility is not necessary.

The idea is that each piece of data has a link to the next piece of data. Each item in a linked list is stored in a Node. The item is of type T and the link to the next node is just a Node<T> reference. A **null** link indicates the end of the list.

In Java, there is an **add** method that allows the user to either insert an item at the end of the list, as in ArrayList, or at a specific index. There are also methods to insert an item at the beginning (**addFirst**) or end (**addLast**) of the list. To traverse the LinkedList, there is a **get** method or an **Iterator** object may be used.

Run **LinkedList.jar** to get an idea of how a linked list works. This program allows you to insert items at legal locations within the linked list and to remove existing indices within the linked list.

**LinkedList Exercise**

private int[] myArray = { 5, -1, 10, 23, 4, 8, 2, 25, 7, 0 };

LinkedList<Integer> myList = new LinkedList<Integer>();

To implement a linked list for yourself, look at the Javadoc for LinkedList, and write a program called **TestLinkedList.java** that adds all of the items in myArray (using either **add** or **addLast**), prints out the contents of the linked list once all the items have been added, and traverses the linked list (print each item individually as you encounter it). **You should not use the size of myArray to traverse the linked list, only to add items to the linked list**. After you’ve traversed the linked list, find the method in the Javadocs that allows you to remove all items from the linked list with a single method call. Confirm that the method does, in fact, clear the linked list by printing out its contents.