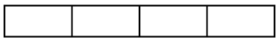
Definition of a Queue

What is a queue?

A queue is another data structure referred to as a “collection”. A queue differs from a stack in that a queue operates on the principle of adding or removing data in a FIFO (first in; first out) manner. Recall that a collection is any data structure that binds groups of objects of a given type together.

With a Queue, we add and remove items from different points generally.

In a classic model, queues are thought of like a line (at DMV, for example):



The diagram above can demonstrate the nature of how we model a queue data structure.

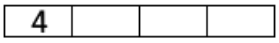
When we create a new queue, the queue is empty. Therefore, it contains no data in it. Like with stacks, we refer to a piece of data inside of the queue as an “item”.

Since queues are modelled as a line, we consider one end of the line as the “head” and the opposite end of the line as the “tail.” Whether the right or left side is considered the head or tail is generally determined by whether the implementation of the queue is done using an array or linked lists.

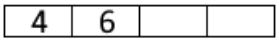
For an array implementation of a queue, we generally consider the left side to be the “head” and the right side to be the “tail.” This means that, when adding an item to a queue, we will add to the tail. When removing from a queue, we remove from the head. When a queue is empty, the head and the tail are the same.

For a linked list implementation of a queue, we usually model it as the right side being the “head” and the left side being the “tail” although this isn’t always the case. One thing that is universal with queues is that we always add to the tail and remove from the head (regardless of where those might be).

Assuming an array implementation of a queue (left: head, right: tail), assuming we added a 4 to the queue:



This is how the queue would look. Now, assume we added a 6 to the queue. This is where the item would be placed:



At this point, we can either add more items to the queue, remove an item from the queue, or peek at the head of the queue. In a basic queue, we are unable to peek at any items other than the item at the head of the queue.

If we were to add a 7 to the queue, it would be placed to the right of the 6. If we, instead, removed from the queue, the 4 would be removed and returned leaving only the 6 in the queue. If we wanted to peek at the queue instead of the first two operations, the 4 would be returned but would still remain in the queue.

Like stacks, we cannot arbitrarily access items within the queue. You can only access the head item.

Purpose of a Queue

What purpose does a Queue serve?

Queues serve the same purpose stacks do but with different ordering of items within the collection. Stacks were used for a recursive ordering of data, but Queues are useful for when we want a linear approach or a “fair” approach (every item must “wait their turn”).

Queues are useful when we want to track data in the order it was received (or prioritize by time in.)

Think of when you go to DMV (or even a busy deli). You will receive a number based on when you came in. The numbers generally are called in the order given out. You receive your number and wait for your turn. Such is the way that queues operate.

Common Operations in a Queue

What are the common operations (functions) in a Queue class?

Queues generally support all of the base operations that Stacks do, such as:

1. Adding an item to the Queue
2. Removing an item from the Queue
3. Looking at the item at the head (front) of the Queue
4. Knowing whether the Queue is empty or not
5. Knowing how many items are in the Queue
6. Resetting the Queue to empty

The terminology for adding and removing is different with Queues than for Stacks. With Stacks, we refer to adding to a stack as “pushing” the stack. With a Queue, there are two different ways this can be referred to: “adding” to the Queue or “enqueueing” an item. Therefore, most Queue classes will have the functions “add” or “enqueue” to refer to the process of placing an item into a queue.

With Stacks, we refer to removing from the stack as “popping” the stack. With a Queue, there are also two different ways to refer to removing. The functions are generally either called “remove” or “dequeue”. Whatever you name it for adding determines what you name for removing. The “add” function matches with “remove”. Likewise, the “enqueue” method matches with “dequeue”. Keep consistency regardless of what naming convention you use when creating custom Queue classes.

“Peek” works similar to how it does with a Stack, but we view the item at the head of the queue. The “isEmpty” and “size” methods work identical to how stacks work; as does the clear operation.

Creating and Using Queues using the Java API

How would we use Queues in Java?

The Java API includes Queue as part of it. Here is the key information:

```
import java.util.Queue; // This references the Queue class of the Java API
import java.util.LinkedList; // The Queue class uses the linked list class to work properly
// Inside method:
// Notice that the new object is “LinkedList” instead of “Queue”? Remember this gotcha!
Queue<String> q = new LinkedList<String>(); // Declare and initialize empty queue named “q"
q.add(“String #1”); // Java API version uses “add” instead of “enqueue”
q.add(“String #2”); // Add the second string to the queue
String s = q.remove(); // Java API uses “remove” instead of “dequeue”
String s2 = q.peek(); // Head of queue item placed into s2
boolean b = q.isEmpty(); // Works just like a Stack here
int n = q.size(); // Save number of items left in Queue
q.clear(); // Resets queue to empty (size() == 0; isEmpty() == true)
```

Creating Queues from Scratch in Java (Array Implementation)

How do we create a Queue from scratch? (Array implementation; fixed-size)

Generally, we want to implement Queues from scratch using linked lists. This is because of the weirdness of how queues work with arrays when we start to remove items from the queue. We will end up with a “wrapping” effect after a while.

Although it is possible to create a dynamically sized queue using an array, it is tricky and ugly. We will limit our example of a custom queue with an array implementation to a fixed size queue:

```
package Main;

/* This is a fixed-size (max size) array implementation of a queue */

public class Fqueue{

	// Fields

	private int[] a;

	private int head;

	private int tail;

	private int n;

	// Constructor

	public Fqueue(){

		a = new int[256];

		head = 0;

		tail = 0;

		n = 0;

	}

	// Methods

	public void enqueue(int item){

		if(size() == a.length) return; // No room left!

		if(tail >= a.length)

			tail = 0;

		a[tail] = item;

		tail++;

		n++;

	}

	public int dequeue(){
```

```
            if(isEmpty()) return -1; // Error code
            int ret = a[head];
            head++;
            n--;
            if(head >= a.length)
                  head = 0;
            return ret;
      }
      public int peek(){
            if(isEmpty()) return -1; // Error code
            return a[head];
      }
      public boolean isEmpty(){
            return size() == 0;
      }
      public int size(){
            return n;
      }
      public void clear(){
            head = 0;
            tail = 0;
            n = 0;
      }
}
```

Creating Queues from Scratch in Java (Linked List Implementation)

How do we create a Queue from scratch? (Linked List implementation)

```
package Main;

/* This is a linked-list implementation of a queue */

public class Lqueue{

	// Fields

	private Node head;

	private Node tail;

	private int n;

	private class Node{

		int item;

		Node next;

	}

	// Constructor

	public Lqueue(){

		head = null;

		tail = null;

		n = 0;

	}

	// Methods

	public void enqueue(int item){

		Node oldTail = tail;

		tail = new Node();

		tail.item = item;

		tail.next = null;

		if(isEmpty()) head = tail;

		else oldTail.next = tail;

		n++;

	}

	public int dequeue(){

		if(n == 0) return -1;
```

```
            int item = head.item;
            head = head.next;
            if(isEmpty()) tail = null;
            n--;
            return item;
      }
      public int peek(){
            if(n == 0) return -1;
            return head.item;
      }
      public boolean isEmpty(){
            return head == null;
      }
      public int size(){
            return n;
      }
      public void clear(){
            head = null;
            tail = null;
            n = 0;
      }
}
```