Lecture 13

Queues

Queues are FIFO data structures (First In, First Out). They are the equivalent of waiting in line. Values are added to back of the queue, and are removed from the front.

We previously talked about the example of keyboard input, and the producer-consumer model. This is essentially a queue. Input come into the buffer (queue) and are consumed in the order they arrived.

Queues use these standard operations:

- `enqueue(E value)` – add value to end of the queue
- `dequeue()` – remove a value from front of the queue
- `peek()` – look at, but don't remove, first item in queue
- `empty()` – returns true iff queue has no values

Bailey's implementation of the code starts with the `Queue.java` interface. Like the `Stack` interface, it implements the `Linear` interface, and so we must implement several different methods.

Like the `Stack` example, Bailey handles the methods with duplicate functionality by implementing some of them in the `AbstractQueue.java` abstract class.

specifically:

- enqueue is just a call to add
- dequeue is just a call to remove
- peek is again a call to get

We can implement a queue as a linked list, as in `QueueList.java`. Here we use a circularly linked list. The circularly linked list maintains a `tail` pointer, which allows it to add to the back of the queue, and also to remove and get from the front of the queue easily (using `tail.next()` to refer to the head of the queue). Knowing this, the queue method implementations are mostly trivial. For example, in the abstract class `enqueue(E value)` just calls `add(E value)`, which is implemented in the `QueueList` class and just calls the cirular list's `addLast(E value)`.

The array implementation is a bit trickier, and requires some work.

It uses three instance variables, the array, the head of the queue, and the count or size of the queue.

So, when we add, we will want to add to the back of the queue. This is represented by `head` (front of queue) + `count`. Of course, we need to update count. Notice that we don't update `head`. The front of the queue doesn't change when we add a value.

To remove, we want to remove from the head of the list, which is just the value at index `head`. Of course, we will also need to update `head` and `count`.

If we examine a few enqueue and dequeue commands, we see that the items on the queue tend to drift towards the end of the array. This will be a problem over time, because we will run out of space, even though the array may not be empty.

Notice that as we remove values, we are freeing up space at the front fo the array. We can reuse this space by using modular arithmetic to calculate the indices.

So, instead of adding to index $head + count$, we will add to $(head + count)$ % $data.length$.

When we remove, we still remove the value at index `head`, but we need to use modular arithmetic to update `head`. Instead of $head = head + 1$, we use $head = (head + 1)$ % $data.length$.

One last problem. Since we are wrapping around, we have the possibility that we may smash our array. That is, add a new item on top of (and destroying) an existing and valid item in the queue.

Similarly, we are just maintaining indices of head, so we are leaving removed values in the queue, knowing they will get overwritten.

We must test for empty and full before doing removes and adds, respectively!

Look at run time efficiency:

	QueueList	QueueArray
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
peek	O(1)	O(1)
empty	O(1)	O(1)

For `QueueList`, `enqueue`, `dequeue`, and `peek` are $O(1)$ because we have tail pointer and circular list. `empty` can be implemented by testing `tail` against `null`.

For `QueueArray`, we use simple modular math and addition to find the location to `enqueue`, or the item to `dequeue` or `peek`. `empty` can simply examine the value of `count`, and `full` (not listed above, but implied by our discussion) can be obtained in constant time by comparing `count` to the length of the array.

Deques and Steques

A `Deque` is a variant of a Queue for which items can be added and removed from either end.

A textttSteque is a variant of a Queue for which items can be added and removed from one end, but only added to the other end.