CMPSCI 187: Programming With Data Structures

Lecture #15: Implementing Queues
12 October 2011

Implementation of Queues

- Review the Queue ADT (in L&C's language)
- Linked List Implementation
- Circular Array Implementation
- Modular Arithmetic
- The Java Queue Classes

Review of the Queue ADT (in L&C's language)

- We want to be able to add elements to the queue (with `enqueue`), remove and return the element of the queue that has been there longest (with `dequeue`), look at that longest-present element without removing it (with `first`), test whether the queue is empty (with `isEmpty`), and return how many elements are in the queue (with `size`).

- The only kind of exception we want to throw from these methods is an `EmptyCollectionException`, in the case where we try to dequeue or look at the first element and the queue is empty.

- L&C's `QueueADT` interface also includes the `toString` method. This isn't fundamental to the operation of the queue, but it's a useful reminder that wherever we put the data in our implementation, we must be able to find it.

- Naturally `toString` is going to take O(n) time on a queue of n elements, but we we would like the other operations to take only O(1) time.

The Linked List Implementation

- As with the stack, we can reasonably hope for O(1) implementations of all five basic methods, counting on the flexibility of linked lists to avoid the need for time-consuming resizing.

- Our basic linear linked list of `LinearNode` objects has a start, which we'll now call the **front**, from which we can reach all the nodes in the list by following `next` pointers. But we need to take actions at both ends of the list, so we will also keep a pointer to the **rear**, the last element.

- It's not immediately obvious whether the "front" or the "rear" of the list should be the place where we enqueue new elements. In fact it's pretty easy to add an element in either place. But what about dequeuing? We know how to cut out the first element easily enough. But if we cut out the rear element, we have to reset the rear pointer to its predecessor, and in a singly linked list we don't have any way to get to that node without taking O(n) time.

The `LinkedQueue` Class

- We have three fields and a simple constructor.

- The `enqueue` method has a special case for an empty node. The new node is at the rear.

```
public class LinkedQueue<T> implements QueueADT<T> {
   private int count;
   private LinearNode<T> front, rear;
   public LinkedQueue ( ) {count = 0; front = rear = null;}

   public void enqueue (T element) {
      LinearNode<T> node = new LinearNode<T> (element);
      if (isEmpty( )) front = node;
      else rear.setNext (node);
      rear = node;
      count++;}
```

The Rest of the `LinkedQueue` Methods

- We dequeue from the front -- the other three methods are very simple.

```
public T dequeue( ) throws EmptyCollectionException {
   if (isEmpty())
      throw new EmptyCollectionException("queue");
   T result = front.getElement( );
   front = front.getNext( );
   count--;
   if (isEmpty( )) rear = null;
   return result;}

public T first( ) throws EmptyCollectionException {
   if (isEmpty( ))
      throw new EmptyCollectionException("queue");
   return front.getElement( );}

public boolean isEmpty( ) {return (count == 0);}

public int size( ) {return count;}
```

The Circular Array Implementation of a Queue

- As we saw with the dropout stack of Project #3, we can have an array where the area used is a continuous segment of entries, possibly wrapping around from item n-1 to item 0, where we can alter this segment at both ends.

- Following L&C, we will call the beginning of this segment `front`, the entry after the end of this segment `rear`, and the size of the segment `count`. Along with these three `int` fields, we have an array `queue`. We'll refer to `queue.length` as the **capacity**.

- If we get a request to enqueue and the queue is full, we expand capacity by copying the array into a new array of twice the size. Thus in the worst case enqueuing takes O(n) rather than O(1) time, though a series of O(n) enqueue operations will also take O(n) time because there will only be O(1) resizings. We call this "O(1) amortized time" per enqueue operation.

Code for the `CircularArrayQueue` Class

- We have four fields (including a constant, which I would have made `static`) and two constructors that produce empty queues.

```
public class CircularArrayQueue<T> implements QueueADT<T> {
   private final int DEFAULT_CAPACITY = 100;
   private int front, rear, count;
   private T[ ] queue;

   public CircularArrayQueue( ) {
      front = rear = count = 0;
      queue = (T[ ]) (new Object[DEFAULT_CAPACITY]);}

   public CircularArrayQueue(int initialCapacity) {
      front = rear = count = 0;
      queue = (T[ ]) (new Object[initialCapacity]);}
```

More Code for `CircularArrayQueue`

- Here's three of the five -- `isEmpty` and `size` are obvious. We'll also leave out `resize`.

```
public void enqueue (T element) {
   if (size( ) = queue.length)
      expandCapacity( );
   queue[rear] = element;
   rear = (rear+1) % queue.length;
   count++;}

public T dequeue( ) throws EmptyCollectionException {
   if (isEmpty( )) throw new EmptyCollectionException("queue");
   T result = queue[front];
   front = (front+1) % queue.length;
   count--;
   return result;}

public T peek( ) throws EmptyCollectionException {
   if (isEmpty( )) throw new EmptyCollectionException("queue");
   return queue[front];}
```

Modular Arithmetic

- In our circular array, we needed to turn the linear array into a circular one. We did this by replacing the “next entry” operation, which normally takes i to i+1, with a “wraparound next entry” operation, taking x to (x+1)%n. (Here n is the size of the array.)

- If we always apply the operation “%n” at the end of every arithmetic operation (at least +, -, and *), we turn the set {0, 1, 2,..., n-1} into a different number system, called the **integers modulo n**.

- We’ll use addition modulo n next lecture when we talk about certain substitution ciphers (following L&C’s example in section 5.2) and we’ll see it again later in the course when we do hashing.

- We do a lot more with modular arithmetic in CMPSCI 250.

The Java Queue Classes

- Recall that the standard Java `Queue` interface has different names for most of the basic methods, and has variant versions that don't throw exceptions.
- Collections has lots of linked and array classes that implement Queue.
- Some are "blocking queues", which refers to the possibility that multiple threads will try to modify the same data structure -- we ignore this issue.
- Some are double-ended queues or **deques** (pronounced "decks") like those we saw in today's discussion. The class `ArrayDeque` is a good one to use.
- In fact priority queues still also implement the Queue interface even though they don't necessarily return the same elements as a real queue would.