CMPSCI 187: Programming With Data Structures

Lecture #17: Queues as an Abstract Type
David Mix Barrington (guest lecturer: James Allan)
17 October 2012

Queues as an Abstract Type

- The Queue Operations
- Using Queues: Repeating Strings
- Searching With a Stack
- Searching With a Queue
- Finding Distance on a Grid
- The Queue Interfaces
- Queues in `java.util`

The Queue Operations

- A **queue** is a data structure holding a collection of objects. Like a stack, it allows insertion of new elements and removal of existing elements. Unlike a stack, it allows removal of only the *earliest-inserted element*, rather than the most recently inserted one. This is called **first-in-first-out** or **FIFO**.

- In DJW's vocabulary, we add an element by **enqueueing** it, whereupon it is at the **rear** of the queue. We can remove the element at the **front** of the queue by **dequeueing** it. "Queue" is another word for "waiting line", particularly in British English, and our queues operate like those at a supermarket -- you enter at the rear and stay there until you reach the front and are served. The queue classes in java.util have a different vocabulary which we'll review later.

- DJW's queue classes do not have a way to look at the item at the front of the queue without removing it, and this time they do not appear to care that dequeue is both an observer and a transformer.

Using Queues: Repeating Strings

- DJW's sample program here allows the user to input exactly three lines of text, then outputs them in the same order they were input. If we had put them into a stack rather than a queue, they would have come out in the reverse order. "BQI" abbreviates "BoundedQueueInterface".

```
public class RepeatStrings {
   public static void main (String [ ] args) {
      Scanner conIn = new Scanner (System.in);
      BQI<String> queue = new ArrayBndQueue<String> (3);
      String line;
      for (int i = 1, i <= 3; i++) {
         System.out.println ("Enter a line of text > ");
         line = conIn.nextLine( );
         queue.enqueue(line);}
      System.out.println ("Order is:\n");
      while (!queue.isempty( ))
         System.out.println (queue.dequeue( ));}}
```

Searching With a Stack

- A **graph** is a general structure where items called **nodes** are linked by **edges**, where an edge goes from one node to another. In Project #2 we could imagine a graph where the nodes represented board positions, and the edges represented the result of a move. We wanted to know whether there was a path from the start position to a winning position, using only nodes for valid positions.

- Our strategy then was a **backtrack search** -- we tried moves forward, then rejected and replaced them if the turned out not to lead to winning positions. In Project #2 we kept track of our search with an explicit stack, which stored the moves we were considering at any given time.

- Our search was **greedy** in that it took a move forward and stayed with that move until or unless it was proved to be not useful. We go *deeper* into the search wherever possible -- this general strategy is called **depth-first search**.

Searching With a Queue

- In the Grid class, our markBlobs method is supposed to search out all possible paths from the target square across other land squares, so that it can mark all the squares on the target square's continent. It operated recursively, marking the target and then calling itself on any unseen land neighbors. If we look at the call stack of the recursion, we can describe this strategy as "pop a square, mark it, then push all its unseen land neighbors".

- What if we try "dequeue a square, mark it, then enqueue all its unseen land neighbors"? We mark the target, then the target's neighbors, then the neighbors of those neighbors, and so forth. We visit all the squares at a particular distance from the target before visiting any at a greater distance. This is called a **breadth-first search** because we make a broad coverage of the search space at one depth before going any deeper.

- A breadth-first search of the space of Sudoku boards would not have worked well at all. Can you see why?

Finding Distance in a Grid

- We define the **distance** from one land square (i, j) to another square (i', j') in a Grid object to be the length of the shortest path from (i, j) to (i', j') if there is such a path, and "undefined" (sometimes called "infinite") otherwise. Here a path must move north, east, south or west (not diagonally) on each move, and may use only land squares.

- We could compute the distance from (i, j) to (i', j') by starting a queue-based search at (i, j), and remembering the distance to each square when we put it on the queue. We enqueue a square because it is adjacent to an existing square -- if the existing square is at some distance k from (i, j), then the enqueued node is at distance k + 1. If we find (i', j') and assign it, that is our answer. If we empty the queue without finding it, there is no path to it at all.

- In Project #3 your task will be a bit easier. You have no goal square -- you just run the search until the queue empties, and return the distance to the last square in the queue. This is the longest distance from the start of the search to a square on the continent -- we will look for a capital that minimizes this.

The Queue Interfaces

- As they did for stacks, DJW define three separate interfaces for the Queue abstract data type. One is for bounded queues that could conceivably become full, and the other is for unbounded queues that can always find more memory. Remember that since the queue exceptions are unchecked (they extend `RuntimeException`), the throws clauses for them are advisory rather than required. Also note that the central meaning of a Queue, that dequeuing returns the element that has been in the queue the longest, is not guaranteed by the interface -- it only defines the possible interactions with it.

```
public interface QueueInterface<T> {
   T dequeue throws QueueUnderflowException;
   boolean isEmpty;}

public interface BoundedQueueInterface<T> extends QI<T> {
   void enqueue (T element) throws QueueOverflowException;
   boolean isFull;}

public interface UnboundedQueueInterface<T> extends QI<T> {
   void enqueue (T element);}
```

Queues in `java.util`

- In `java.util`, `Queue` is an interface that is part of the Collections Framework. It is implemented by various classes such as `ArrayQueue`. It usually acts as a FIFO queue, but can be defined to work in other ways.
- Enqueuing can be done with either of two methods: `add` which throws an exception if the queue is full, and `offer` which returns `false` if it is full. Both methods return a boolean which is true if the enqueuing succeeds.
- Dequeuing can be done with either of two methods as well: `remove`, which throws an exception if the queue is empty, and `poll` which returns `null` (not `false`, as DJW say on page 305) if it is empty.
- There are also two methods to return the front element without removing it. If the queue is empty, `element` throws an exception but `peek` returns `null`.