CMPSCI 187: Programming With Data Structures

Lecture 14: The Queue ADT and Breadth-First Search
11 October 2011

The Queue ADT and Breadth-First Search

- Stacks, Queues, and Priority Queues
- Using Queues For Timesharing
- Other Uses for Queues
- The Queue ADT's: L&C's `QueueADT` and Java's `Queue` Interface
- General Search
- Project 4: Breadth-First Search in the Maze
- Recovering the Path in Project 4

Stacks, Queues, and Priority Queues

- When I hold office hours, many students come to see me and I can concentrate on only one at a time. This is typical of real-world computing situations -- I am a **server** with multiple **clients**, and I need a protocol telling which ones to serve and when.

- A **queue** is the most typical way to handle such a situation. ("Queue" is the British usage for what Americans more often call a "line", as in a supermarket or a bank -- it is the French word for "tail".) If I keep the waiting students in a queue, when I finish with one student I turn to the one who has been waiting the longest.

- If I instead turned to the student with the most urgent problem, or the student I liked the best, I would be operating a **priority queue**. If I turned to the student who had been waiting the least time, I would be operating a stack. (Suppose, for example, that I put the current student on hold whenever a new one came in, and went back to the latest-interrupted student when I finished.)

Using Queues For Timesharing

- What I actually do in my office hours is similar to the practice of **timesharing**, the most common way to have a server handle multiple clients.
- I keep a queue, and when I have been talking to a particular student for long enough, I send them to the back of the current queue and start with the next student.
- Before personal computers, timesharing was used by a single large computer to give a number of different users the illusion that they had their own computer. It would rotate among the different users, each time using its CPU to carry out that user's tasks. Maintaining the context for all the users, so as to be able to restart each one's job quickly, was a challenging problem.
- A modern operating system has to timeshare between many processes.

Other Uses For Queues

- The **buffers** of a computer network are organized as queues. A given node in the network receives packets and holds them until it is ready to send them out. A given buffer normally has a fixed capacity, and when packets come in that will not fit into the buffer, they are dropped.

- Modern networks do not try to eliminate packet drop in general, but instead send enough multiple copies of packets so that messages can usually be reconstructed. A branch of mathematics called **queuing theory** deals with how often the drops happen, given assumptions about when packets will arrive. Analysis of networks relies heavily on probability theory.

- A major use of queue data structures is to **simulate** queues in the real world, to observe their range of behavior under different assumptions. L&C give an example of a simulation of a set of bank tellers, which we will look at later.

The Queue ADT's, in L&C and in the Java Library

- As they did for the stack, L&C define a Java interface for the queue, which they call `QueueADT`. It has six methods: `enqueue`, `dequeue`, `first`, `isEmpty`, `size`, and `toString`.

- In `java.util`, as you can see in the API, there is an interface called `Queue`, which has a number of methods including exact analogs of five of the six above: `add`, `remove`, `element`, `isEmpty`, and `size`. There are also variants of the first three: `offer`, `poll`, and `peek`.

- The difference in names is unfortunate but understandable -- L&C want to explain clearly what a queue is, while the production Java code is more concerned with compatibility with other classes in the Collections framework.

- You'll want to use the Java names in Project #4 but we'll use L&C's here.

The Queue Operations

- The `enqueue` (Java `add`) method puts a new element at what we call the **rear** of the queue. It has one parameter of type T, the element added. The Java `add` method returns a boolean that is true if the add succeeds. It throws an exception if the queue is full, while the Java `offer` method just returns false.

- The `dequeue` (Java `remove`) method removes and returns the element from the front of the queue, and throws an `EmptyCollectionException` if the queue is empty. The Java `poll` method returns `null` from an empty queue rather than throwing the exception.

- The `first` (Java `element`) method returns the first element from the queue without removing it. It throws an exception from an empty queue, while the Java variant `peek` instead returns `null`.

- The `isEmpty` and `size` methods are just the same as in `StackADT`.

General Search

- Our maze search in Project #2 can be thought of as an example of a general search pattern.
- We had a data structure that held cells that were (or at least might have been) not yet completely investigated. When we took a node out and investigated it, we put any open unseen neighbors into the data structure. When we were done with a node, we removed it (but kept it marked as “seen”).
- If we find the goal node in our search, we report that a path exists. If we empty out the data structure, we know that no path exists because we have tried all possibilities.
- In Project #2, of course, this data structure was a stack. In Project #4 it will be a queue, and in CMPSCI 250 you will see about using a priority queue.

Project #4: Breadth-First Search in the Maze

- Let's consider what happens when we carry out the general search with a queue. We begin with the source node in the queue, then take it out and enqueue all of its open unseen neighbors.

- We then process each of those neighbors in turn, placing *their* open unseen neighbors in the queue. These newly enqueued nodes have two-step paths from the source, but not one-step paths. (If they had one-step paths they would have been neighbors, and would now have been seen.)

- We then enqueue all the distance-3 nodes, then all the distance-4 nodes, and so on, until or unless we find the destination node. If we run out of open unseen nodes, we give up on finding a path.

- This guarantees that we find not only a path but the *shortest path*.

Recovering the Path in Project #4

- Or does it? In Project #2, if we found the destination, the nodes on the stack at the moment we found it themselves constituted a path. This was because each node went on the stack because it extended a path from the source.

- Now, if we find the destination as distance 7, for example, the nodes on the queue at that time are a mixture of distance-6 and distance-7 nodes, and not a path at all.

- But we can recover the path as long as each node remembers *how it was found* in the search -- which node was it found to be a neighbor of, and how long is its shortest path from the source.

- We will define a new class `QCell` extending `SCell`, whose objects have an `int` field for the distance and a `QCell` field for their parent cell.