CMPSCI 187: Programming With Data Structures

Lecture #18: Implementing Queues With Arrays
David Mix Barrington (guest lecturer: James Allan)
19 October 2012

Implementing Queues With Arrays

- A First Idea: Fixing the Front of the Queue
- A Better Idea: A Circular Queue
- Code for `ArrayBndQueue`
- An Unbounded Array-Based Class
- Testing for Palindromes
- (if time) Another Application: The Game of War

A First Idea: Fixing the Front of the Queue

- Since a queue is a list of elements with a fixed order just like a stack, our natural first thought is to place the elements in an initial segment of an array.
- If we place the front element in location 0 of the array, then enqueuing works exactly like pushing did in `ArrayStack`. A new element can go in location whose index is equal to the former size, and we increase the size by one.
- But now what happens when we dequeue an element? We have a gap in location 0, and restoring our condition, that the elements be in an initial segment, takes O(n) operations to move each element forward by one.
- What if we put the elements in a *final* segment of the array? Now we can dequeue in O(1) time, but enqueuing means that we have to make room for the new element in the last location, taking O(n) time again.

A Better Idea: A Circular Queue

- So fixing either the front or the back of the queue in the array is a bad idea. What if we allow both to move? We keep a front index and a rear index in our array. To enqueue, we add a new item at the rear and increment the rear index. To dequeue, we remove the front item and increment the front index.

- This would be fine except that the array is finite. Even if the number of items in the queue stays below some bound, if there is a lot of traffic into and out of the queue we will use up our array space.

- The trick that makes this work is to think of the array as being **circular**. Instead of incrementing the array with an instruction `index++`, we instead say `index = (index + 1) % capacity`. Remember that the Java `%` operator gives us the remainder when its first argument is divided by the second. So in this way location `capacity - 1` is followed by location `0`.

Code For `ArrayBndQueue`

- There are various ways to define the variables `front` and `rear` to make the circular queue work. We choose to have `rear` point to the last occupied location, which doesn't immediately tell us what to do when the queue is empty. The right invariant is that the next location for enqueuing is `rear + 1` (with wraparound), so we want to start `rear` at location "-1". Note that a full queue and an empty queue both have `rear` just before `front`. But we keep track of `numElements`, which tells these two cases apart.

```
public class ArrayBndQueue<T> implements BQI<T> {
   protected final int DEFCAP = 100;
   protected T[ ] queue;
   protected int numElements = 0;
   protected int front = 0, rear;
   public ArrayBndQueue(int maxSize) {
      queue = (T[ ]) new Ocject(maxSize); // compiler will warn
      rear = maxSize - 1;} // virtual position "-1"
   public ArrayBndQueue( ) {
      this (DEFCAP);}
```

Operations for `ArrayBndQueue`

```
public void enqueue (T element) {
   if (isFull( ))
      throw new QueueOverflowException("add to full queue");
   else {rear = (rear + 1) % queue.length;
      queue[rear] = element;
      numElements++;}}

public T dequeue ( ) {
   if (isEmpty( ))
      throw new QueueUnderflowException("dequeue from empty");
   else {T toReturn = queue[front];
      queue[front] = null;
      front = (front + 1) % queue.length;
      numElements++;
      return toReturn;}}

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

public boolean isFull( ) {
   return (numElements == queue.length;}
```

An Unbounded Array-Based Queue Class

- As we've mentioned for ArrayList, we can increase the size of array as needed, at the cost of O(n) time each time we enlarge. (They don't double.)

```
public class ArrayUnbndQueue<T> implements UQI<T> {
   protected final int DEFCAP = 100;
   protected T[ ] queue;
   protected int origCap, numElements = 0; front = 0; rear = -1;
   public ArrayUnbndQueue(int origCap) {
      queue = (T[ ]) new Object[origCap]; // compiler will warn
      rear = origCap - 1; this.origCap = origCap;}
   public ArrayUnbndQueue( ) {
      this(DEFCAP);}
   private void enlarge( )
   // make new array of size queue.length + origCap, copy into it
   public void enqueue(T element) {
      if (numElements == queue.length) enlarge( );
      rear = (rear + 1) % queue.length;
      queue[rear] = element;
      numElements++;}
   // dequeue, isEmpty do not change, there is no isFull
```

Testing for Palindromes

- A **palindrome** is a string that is the same when written backward, like "Hannah", "Able was I ere I saw Elba", or "Madam, I'm Adam". Note that we ignore spaces, case and punctuation.

- There are word-level palindromes, like the photo caption "Girl, bathing on Bikini, eyeing boy eyeing bikini on bathing girl". But let's do letter-level.

- Here's a simple way to test whether a string is a palindrome. Create a stack and a queue (of Characters) and read the string, putting a copy of each *letter* (not spaces or punctuation) into both the stack and queue. Then pop and dequeue one character at a time, checking that they match. If we empty both the stack and queue at the same time without finding a mismatch, the original string was a palindrome.

Code for `Palindrome`

```
public class Palindrome {
   public static boolean test (String candidate) {
      char ch, fromStack, fromQueue;
      int length = candidate.length, numLetters = charCount = 0;
      boolean stillOK = true;
      BSI<Character> stack = new ArrayStack<Character>(length);
      BQI<Character> queue = new ABQ<Character>(length);
      for (int i = 0; i < length; i++) {
         ch = candidate.charAt[i];
         if (Character.isLetter(ch)) {
            numLetters++;
            ch = Character.toLowerCase(ch);
            stack.push(ch); queue.enqueue(ch);}}
      while (stillOK && (charCount < numLetters)) {
         fromStack = stack.top( ); stack.pop( );
         fromQueue = queue.dequeue( );
         if (fromStack != fromQueue) stillOK = false;
         charCount++;}
      return stillOK;}
```

A Palindrome Application

- DJW provide an application that allows a user to provide as many strings as they like and have the palindromosity of each evaluated in turn. The code is not particularly different from other console applications. The point to remember is that we separate the user interface from the central calculation.

- This is certainly an illustration of the difference between a stack and a queue, but there are other ways to decide palindromosity. We could place two “fingers” (int variables) on the string and walk the first one forward and the second one backward, as long as the characters under the two match, until they meet in the middle.

- If we knew the length of the string, we could push the first half onto a stack and compare it to the second half as we read it out backward.

- We can define a palindrome recursively as a letter, followed by a palindrome, followed by the same letter, with non-letter characters allowed anywhere. The base case is the empty string. This can be made into a recursive algorithm.

The Card Game of War

- War is a two-player card game that calls for no decision-making by either player. You shuffle the deck and play deterministically until the game ends. Each player has a “hand”, originally half the deck.

- A basic play is for each player to turn up the first card in their hand. The player with the higher rank card wins both cards and puts them at the end of their hand. If the two cards have the same rank, each player deals out three cards face-down from their hand and turns up the next card in their hand. The player with the higher-ranked of these wins all ten cards now on the table. If those two cards have equal rank, each player plays four more cards for a second “war”, and so on until there is a resolution or one player runs out of cards.

- We’ll simulate this game on random decks to determine how long it usually takes to finish. It will be easy to treat the hands as queues.