CMPSCI 187:Programming With Data Structures

Lecture #19: Linked Queues
David Mix Barrington
22 October 2012

Linked Queues

- Review: Array-Based (Circular) Queues
- The Game of War
- The War Simulation
- A Link-Based Queue Implementation
- The `enqueue` Operation
- The `dequeue` Operation
- A Circular Linked Queue Design

Review: Array-Based (Circular) Queues

- A **queue** is a linear data structure that is **first-in-first-out**. Objects may be added at the **rear** of the queue by the **enqueue** operation, or removed from the **front** of the queue by the **dequeue** operation. DJW's `QueueInterface` interface has the `dequeue` and `isEmpty` methods -- they have two extensions of this interface with the `enqueue` method (one, `BoundedQueueInterface`, also has an `isFull` method).

- We saw last week how to implement a queue with an array. *Both* the front and rear of the queue change their position in the array as items enter and leave the queue. We think of the array as **circular**, in that the position coming after `size - 1` is position `0`. In general, the position after position `i` is position `(i+1) % size`. The variable `rear` is the index before the place to put the next enqueued item (initially -1), and the variable `front` is the index of the next item to be dequeued. We also keep a variable `numElements`, not least to be able to distinguish a full array from an empty one.

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.

The `RankCardDeck` Class

- Each card has a rank from 0 to 12. The shuffle chooses cards independently.

```
public class RankCardDeck {
   private static final numCards = 52;
   protected int[ ] carddeck = new int[numCards];
   protected int curCardPos = 0;
   protected Random rand = new Random( );
   public RandCardDeck ( ) {
      for (int i = 0; i , numCards; i++) carddeck[i] = i/4;
   public void shuffle( ) {
      int randLoc, temp;
      for (int i = (numCards - 1); i > 0; i--) {
         randLoc = rand.nextInt(i); // 0 <= randLoc < i
         temp = carddeck[randLoc];
         carddeck[randLoc] = carddeck[i];
         carddeck[i] = temp;}
      curCardPos = 0;}
   public boolean hasMoreCards( ) {return curCardPos != numCards;}
   public int nextCard( ) {
      curCardPos++; return (carddeck[curCardPos - 1]);}}
```

The War Simulation

- The `WarGame` class sets up two queues of `Integers` for the players' hands, and a third queue for the cards that will be one in the current battle, if any. A variable, set by the user, limits the number of battles that may happen before the game is abandoned.

- The `play` method in `WarGame` deals the cards into the two player's hands and then runs battles until the game ends. The `battle` method puts the players' next cards into the prize queue, then checks the ranks of those two cards. If they are different, it puts the cards in the prize queue into the winner's hand. If they are the same, it puts three cards from each player into the prize queue, then recursively calls itself. Note that DJW's code is intended to analyze the average number of battles in a game -- it does not report the winner.

- If a player runs out of cards, a `QueueUnderflowException` is thrown and caught by the `play` method, which declares the game over and keeps going.

A Link-Based Queue Implementation

- Implementing a queue with a linked list is very natural. We will keep the front of the queue at the head of the list, where we can dequeue easily by cutting out the head node and returning its contents. We need to enqueue elements at the rear of the list, and we can save time by maintaining a pointer to the last element of the list.

- Could we have done this the other way round? That would have meant removing an element from the end of the list, which is problematic for a singly-linked list because we must reset the `rear` pointer to the penultimate node.

```
public class LinkedUnbndQueue<T>
      implements UnboundedQueueInterface<T> {
   protected LLNode<T> front, rear;
   public LinkedUnbndQueue ( ) {
      front = rear = null;}
```

The `enqueue` Operation

- To enqueue an element, we must make a new node and splice it into the list at the rear. Normally this will just mean making the old rear node point to the new one, and updating the rear pointer. We have a special case if the queue is currently empty, and our new node will be the only element of the queue.

- In that case the new node will become both the front and rear element. We can bring the statement `rear = newNode` out of the `if-else` block, however, since we want it to be executed in either case.

```
public void enqueue (T element) {
   LLNode<T> newNode = new LLNode<T>(element);
   if (rear == null)
      front = newNode;
   else
      rear.setLink (newNode);
   rear = newNode;}
```

The `dequeue` Operation

- To dequeue, we remember the contents of the front node, cut it out of the list, and return the contents. Of course if the list is empty we must throw an exception.

- The one complication occurs if the node we are removing is the only one in the queue. We detect this situation as `front == null`, and rectify it by setting `rear` to `null` as well. If we didn't do this, the `rear` pointer would still point to the abandoned former node.

```
public T dequeue ( ) {
   if (isEmpty( ))
      throw new QueueUnderflowException ("Dequeue from empty");
   else {
      T element = front.getInfo( );
      front = front.getLink( );
      if (front == null)
         rear = null;
      return element;}}
```

A Circular Linked Queue Design

- Since a linked list is assembled from nodes with one pointer each, there is no reason it could not be circular instead of linear.

- We can keep just one pointer into the circular list, for the rear element of the queue. Instead of having a null link, the rear node can point to the front node. That way the front node can be referenced as `rear.getLink( )`, and we can dequeue the front element with the statement `rear.setLink(rear.getLink( ).getLink( ))`, of course after remembering its contents.

- The important special case is when there is exactly one node in the circular list, so that we must set rear to null if we dequeue. We can recognize this situation because the read node then links to itself, as it is also the front node. Of course we also need special code to enqueue onto an empty queue.

Comparing the Implementations

- We’ve been careful in both implementations to make the enqueue and dequeue operations each O(1) time (not “always the same time”, as many of you said on the test, but “at most a constant”). The constructor for a linked queue is also O(1), but the constructor for an array with initial size N takes O(n).

- An exception is the unbounded array implementation, which might take O(N) time with N items in the queue if it has to resize the array. As we’ve noted, if we double the array size each time we change it, the total time to resize the array up to size N turns out to be O(N), or an amortized O(1) per enqueue.

- The linked structure uses an extra pointer in each node, doubling the space needed for the pointer to the element object. The array implementation uses the space for the unused entries in the array. Thus if the array is always at least half full, the array implementation uses less space, but both use only O(N) for a queue whose maximum size over time is N elements.