CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #12

John Ridgway

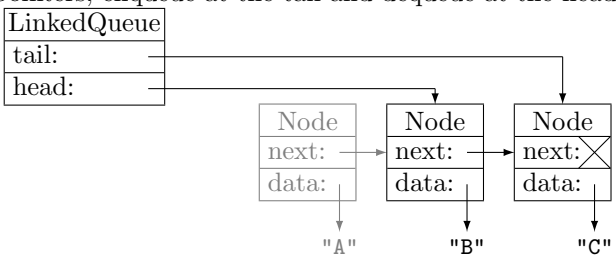
March 10, 2015

1 Implementations of Queues

1.1 Linked Queues

A Linked Queue

Implementing a queue with a linked list is very natural. We keep head and tail pointers, enqueue at the tail and dequeue at the head.



A Linked Queue (continued)

- Could we have done this the other way round, with the front of the queue at the tail of the list?
- Enqueuing would have been easy, but dequeuing would have meant removing an element from the tail of the list, which means setting the tail pointer to the penultimate node.

Code for the Linked Queue

We make our generic queue from `Node<T>` objects.

```
public class LinkedUnbndQueue<T>
    implements UnboundedQueueInterface<T>
{
  private Node<T> head;
```

```
private Node<T> tail;

public LinkedUnbndQueue() {
  head = null;
  tail = null;
}
```

Enqueuing in the Linked Queue

To enqueue an element, we must make a new node and splice it into the list at the tail.

```
public void enqueue(T element) {
  Node<T> newNode = new Node<T>(element);
  if (tail == null) {
    head = newNode;
  } else {
    tail.setNext(newNode);
  }
  tail = newNode;
}
```

The dequeue Operation

To dequeue, we remember the contents of the front node, cut it out of the list, and return the contents.

```
public T dequeue() {
  if (isEmpty()) {
    throw new QueueUnderflowException();
  } else {
    T element = head.getData();
    head = head.getNext();
    if (head == null) {
      tail = null;
    }
    return element;
  }
}
```

Clicker Question #1

In the code below, the last line has replaced `tail = newNode;`. What's wrong with this?

```
public void enqueue(T element) {
  Node<T> newNode = new Node<T>(element);
  if (tail == null) {
    head = newNode;
  } else {
    tail.setNext(newNode); }
  tail = tail.getNext(); }
```

A. the old tail node gets a pointer to itself

B. there's an NPE if the queue is empty

C. the old tail node gets garbage-collected

D. the tail node shouldn't change for `enqueue`

1.2 Applications: Palindromes

Applications: 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". Here's a simple way to test whether a string is a palindrome:

- Create a stack and a queue (of `Character`s) and read the entire string, putting a copy of each letter (not spaces or punctuation) into both the stack and queue;
- Pop and dequeue one character at a time, checking that they match. If we empty both the stack and queue without finding a mismatch, the original string was a palindrome.

Palindrome Code: Part I

```
public class Palindrome {
  public static boolean test(String candidate) {
    USI<Character> stack =
        new LinkedStack<Character>();
    UQI<Character> queue =
        new LinkedQueue<Character>();

    for (int i = 0; i < candidate.length; i++) {
      char ch = candidate.charAt(i);
      if (Character.isLetter(ch)) {
        char lowerCase = Character.toLowerCase(ch);
        stack.push(lowerCase);
        queue.enqueue(lowerCase);
      }
    }
```

Palindrome Code: Part II

```
    boolean stillOK = true;
    while (stillOK && !stack.isEmpty()) {
      char fromStack = stack.pop();  // unboxing
      char fromQueue = queue.dequeue();  // unboxing
      if (fromStack != fromQueue) {
        stillOK = false;
      }
    }
    return stillOK;
  }
}
```

Clicker Question #2

Here's a way to test whether a string is a palindrome recursively. If the first letter and last letter do not match, return false. Otherwise recurse on the string without those two letters. What is the base case of this recursion?

A. it doesn't matter as the recursion is invalid

B. return true on a string with no letters

C. return true if there is zero or one letter

D. return false if there is one letter, true if 0

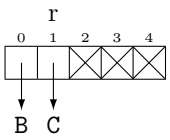
1.3 Array-based Queues

The Array Queues Way

Since a queue is a list of elements with a fixed order, a natural thought is to place the elements in an array:

r=rear

```
enqueue A
enqueue B
enqueue C
dequeue A
```

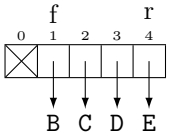


Avoiding Shifting the Array

We had to shift them, which gets expensive, so why not keep track of the front too?

f=front
r=rear

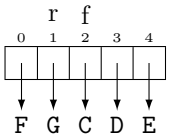
```
enqueue A
enqueue B
enqueue C
dequeue A
enqueue D
enqueue E
```



Circular Queues

But now we run out of space. So wrap-around and put enqueued elements at beginning of the array:

f=front
r=rear




```
enqueue A
enqueue B
enqueue C
dequeue A
enqueue D
enqueue E
enqueue F
dequeue B
enqueue G
```

Implementing a Circular Queue

- Instead of incrementing the index into the array with an instruction like `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`.

Clicker Question #3

The elements in the queue start at location `front` and continue to location `rear`. Which of these expressions gives the number of elements in the queue, assuming the queue is not empty?

A. `(rear - front) % capacity`

B. `(rear - front + 1) % capacity`

C. `(front - rear + 1) % capacity`

D. `(rear+capacity+1 - front) % capacity`

Coding the Circular Queue

- 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.

Circular Queue Attributes and Constructors

```
public class ArrayBndQueue<T> implements BQI<T> {
  private final int DEFCAP = 100;
  private T[] queue;
  private int numElements = 0;
  private int front = 0;
  private int rear = -1;

  public ArrayBndQueue(int maxSize) {
    queue = (T[]) new Object[maxSize];
  }
```

```
public ArrayBndQueue() {
  this(DEFCAP);
}
```

Circular Queue Transformers

```
public void enqueue(T element) {
  if (isFull()) { throw new QOE(); }
  rear = (rear+1) % queue.length;
  queue[rear] = element;
  numElements += 1; }

public T dequeue() {
  if (isEmpty()) { throw new QUE(); }
  T toReturn = queue[front];
  queue[front] = null;
  front = (front + 1) % queue.length;
  numElements -= 1;
  return toReturn; }
```

Circular Queue Observers

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

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

Application: The 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.

The Game of War

- 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.

Simulating the Game of War

- We first need to create and shuffle a virtual deck of cards – DJW's code is worth keeping in mind for future reference though we won't reproduce it here
- Our deck is an array of ints in the range from 0 through 12, since we don't care about suits
- We form an array with four of each rank, then shuffle using a Random object. For each position in turn, we choose a random position after it and swap those two cards

Simulating the Game of War

- DJW's WarGame class sets up two queues of Integers for the players' hands, and a third queue prize for the cards that will be won 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.

Simulating the Game of War

- 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.
- 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.

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", of course, but "at most different by a constant factor").
- The constructor for a linked queue is also $O(1)$, but the constructor for an array with initial size N takes $O(N)$ time.

Comparing the Implementations

- 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. Individual enqueues might be slower, but we have the guarantee about the total time

Comparing the Implementations

- 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 $O(N)$ space for a queue whose maximum size over time is N elements.