CS 240 Midterm Exam

Alex Vondrak

October 31, 2011

1. Prove or disprove the following conjectures.
   (a) **Conjecture.** $f \in O(g) \implies g \in O(f)$
   (b) **Conjecture.** $2^{2+n} \in O(2^n)$
   (c) **Conjecture.** $2^{2n} \in O(2^n)$

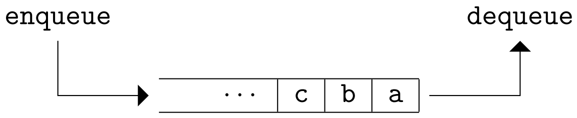
2. For each of the following inputs, illustrate the stack-based algorithm for checking if a string of parentheses is balanced.
   (a) `()())`
   (b) `((()()))`
   (c) `((()()((`

3. (a) Recall the $O(1)$ implementation of **`public int`** `size()` we had for the `ArrayStack<E>` class. Write a version of `size` that performs at least n primitive operations, where n is the number of elements on the stack at the time `size()` is invoked.
   (b) Notice that 3a didn't say "write an $O(n)$ version of `size`". What's the difference, and how could it have changed your answer?

For problems 4–6, we introduce the *queue* data structure. Unlike stacks, which are first-in, last-out, queues are *first-in, first-out* (FIFO). It works much like standing in line at a store. It is defined by the following methods:

- `enqueue` inserts an element at the *back* of the queue.
- `dequeue` removes the element at the *front* of the queue.
- `peek` is used to look at the front element of the queue (without removing it).
- `size` returns the number of elements in the queue.
- `isEmpty` tells us whether there are any elements in the queue.

For example, the following queue has had the elements `a`, `b`, and `c` `enqueue`d in order, so `a` is at the front, followed by `b`, then by `c`.



The Java **`interface`** for a generic queue can be written as

```
interface QueueInterface<E> {
   public void enqueue(E item);
   public E peek() throws EmptyQueueException;
   public E dequeue() throws EmptyQueueException;
   public int size();
   public boolean isEmpty();
}
```

See other side

4. Assume you have two initially empty queues of `Integers`, `q1` and `q2`. What do they look like after each of the following operations? If the operation returns a value, write that value in the **Output** column. If the operation triggers an exception, write “error” in the **Output** column and continue down the table as if the error hadn’t happened.

Operation	**Output**	**q1’s contents**	**q2’s contents**
		(back, . . . , front)	**(back, . . . , front)**
`q2.size()`			
`q1.isEmpty()`			
`q1.dequeue()`			
`q1.enqueue(3)`			
`q1.size()`			
`q1.peek()`			
`q1.peek()`			
`q1.enqueue(1)`			
`q1.dequeue()`			
`q1.size()`			
`q2.enqueue(4)`			
`q2.enqueue(q2.peek())`			
`q1.enqueue(q1.dequeue())`			
`q2.enqueue(q1.dequeue())`			
`q1.isEmpty()`			

5. Write a generic class `Queue<E>` that implements `QueueInterface<E>` by using two instances of `ArrayStack<E>` to store your data internally.

6. What are the running times of the `enqueue`, `dequeue`, `peek`, `size`, and `isEmpty` methods you implemented in problem 5? Give your answers in terms of O of a function of n, where n is the size of the queue.

See other side