Topics

• Exam I stuff, plus...

• Lists
  – Be familiar with the underlying implementations of the ArrayList and SinglyLinkedList.
  – Be able to justify the time complexities of commonly used List methods.
  – Understand the impact on time complexity of applying various optimizations (e.g., iterator caching, tail reference, double linking) on linked lists.
  – Be able to implement various list algorithms, and be able to state/justify their time complexities. For instance, reverse a list.

• Stacks and Queues (and Priority Queues)
  – Understand their underlying implementations (i.e., array vs. linked list) and the impact on the performance of their associated methods.
  – Know when it is appropriate to use either of these data structures in a problem you’re solving.
  – Be able to use either data structure in the design and implementation of a given algorithm.
  – Be familiar with algorithms we went through in class, including palindrome checker, parentheses checker, Dijkstra’s 2-stack infix solver, RPN calculator, infix2postfix, and airport simulator.

• Recursion
  – Be able read and debug recursive methods.
  – Be able to trace a recursive method’s execution.
  – Know how to determine the time complexity of recursive algorithms.
  – Be able to write recursive methods, including those that manipulate known data structures. For instance, how would you use write a recursive indexOf() method for a List?

• Misc.
  – Defining data structures that accept generic types (i.e., diamond notation).
  – Be able to implement new “list-based” data structures on demand given only brief descriptions of their interface. For instance, Deques, CircularLinkedLists, DoublyLinkedLists, etc.
Practice Problems

1. [Queue/Stack Programming] ** Suppose the only data structure currently implemented is the Stack (i.e., Lists don’t yet exist in our language). Define a Queue<E> class with E poll(), offer(E item), and E peek() methods, using only Stacks<E> and its associated methods (push(E item), E pop(), and E peek()). What are the complexities of E poll(), offer(E item), and E peek() using such an implementation?

2. [Queue Programming] Write a method interleave that accepts a Queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue. For example, suppose a variable q stores the following sequence of values: [4,5,6,7,8,9,10,11] (with 4 at the head of the queue). Then interleave(q) should return [4,8,5,9,6,10,7,11]. For full credit, you are only allowed to use one temporary queue in your solution, and you cannot use any methods besides those provided by java’s Queue interface. You should throw an IllegalArgumentException if the size of the given queue is odd.

3. [List Complexity] Consider the following two methods that reverses the contents in a List. Answer the following questions. (If this question appeared on the exam, I would not provide you with the SinglyLinkedList code).

```java
public void reverse1() {
    for (int i = 0; i < size(); i++) {
        add(i, remove(size() - 1));
    }
}

public void reverse2() {
    for (int i = 0; i < size(); i++) {
        add(size() - 1 - i, remove(0));
    }
}
```

(a) If the class in which these methods were implemented is a SinglyLinkedList, which of the two methods would you prefer to run if iterator (location) caching and tail reference are both enabled?

(b) Does your answer to the previous question change if the list was instead doubly linked, with iterator (location) caching and tail reference are both enabled?

(c) Would you prefer one over the other if the list was an ArrayList?
4. [Recursive Tracing] **Consider the following recursive method.

```java
public static int mystery(int[] list, int head) {
    if (head == list.length - 1) {
        if (list[head] % 2 == 0) {
            return 1;
        }
        else {
            return 0;
        }
    }
    else if (list[head] % 2 == 0) {
        return 1 + mystery(list, head+1);
    }
    else {
        return 0 + mystery(list, head+1);
    }
}
```

In a couple sentences, describe what this method does. What is the time complexity of this algorithm? Show your work.

5. [Recursive Programming] The exponentiation of a to the power of b can be expressed as follows:

\[ a^b = a \times a \times ... \times a \text{ \ b times} \]

Provide a recursive method, \( \text{exp()} \), that inputs two integers, \( a \) and \( b \), and returns the value of \( a^b \). You may assume that \( b \geq 0 \). Recall from algebra that when \( b = 0 \), then \( a^b = 1 \). What is the time complexity of this algorithm? Show your work.

6. [Recursive Programming] Write a recursive method \( \text{repeat}() \) that accepts a string \( \text{str} \) and an integer \( n \) as parameters and that returns a string consisting of \( n \) copies of \( \text{str} \). If \( n \) is negative, throw an IllegalArgumentException. For example:

- \( \text{repeat("foo ", 3)} \) returns "foo foo foo "
- \( \text{repeat("i love CS", 1)} \) returns "i love CS"
- \( \text{repeat("i love CS", 0)} \) returns ""

7. [Recursive Programming] ** Define the method \( \text{buildOddStack}() \) that creates and returns a stack full of odd numbers. It should take a single argument, the largest odd number to be placed in the stack, and return a stack containing 1 through that value, with 1 at the bottom of the stack. For example \( \text{buildOddStack(11)} \) would return a stack containing \([1,3,5,7,9,11]\). For full credit, define the method recursively, and don’t use methods other than those defined in the \( \text{Stack} \) class.

8. [Recursive Programming] Do the previous problem again, except that the method should return a stack containing the numbers in reverse order. For example \( \text{buildOddStack(11)} \) would return a stack containing \([11,9,7,5,3,1]\). For full credit, define the method recursively, and don’t use methods other than those defined in the \( \text{Stack} \) class.