Name: ________________________

Computer Science II — Fall 2021

Exam #2

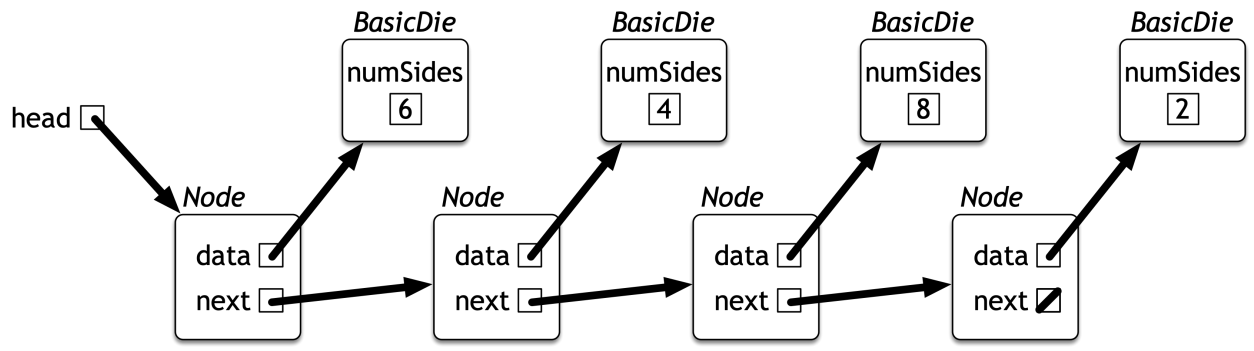
This exam should have five pages. Closed book and notes.
No computers or calculators allowed.

Problem 1: [20 points]

Below is some mysterious code that makes use of a stack. Describe *briefly* in English what the `mystery` method does. What would be a good, descriptive name for this method?

```
public static void mystery(Stack<Integer> s) {
    if (!s.isEmpty()) {
        int top = s.pop();
        mystery(s);
        if (top > 0) {
            s.push(top);
        }
    }
}
```

Problem 2: [26 points]



a) The linked structure above uses the node and die classes developed in class, and used on Lab #5. Write code below that will remove the node containing the eight-sided die from the list by linking around it. You may declare additional variables of type `Node` if you wish, but assignments to them must be made through `head`.

b) On the diagram above, draw in the changes that would result from executing the following assignment statements. (Start with the original structure, rather than the modified list produced by your changes in part a.)

```
Node temp = head.next.next;
temp.next.data = head.data;
temp.next = head;
```

Problem 3: [26 points]

The `collide` method you implemented on Assignment #4 returned a queue of integers representing asteroids that survived the simulated collisions. (Negative values represented left-moving asteroids, and positive values were right moving.) If the method worked correctly, there should never be asteroids in its output that would collide. Below, define a static method called `noCollisions` that could be used to verify this. It should take a `Queue` of `Integer`s as its input and return true if there are no collisions, or false if a collision would occur between asteroids in the queue. A portion of the Queue documentation is at the end of the exam. Feel free to remove that page for reference. (Hint: What must be true of neighboring items in the queue if there *would* be a collision?)

```
public static List<Integer> evens(List<Integer> nums) {
    if (nums.size() == 0) {
        return new LinkedList<Integer>();
    }
    else {
        int firstItem = nums.get(0);
        nums.remove(0);
        List<Integer> evensFromTail = evens(nums);
        if (firstItem % 2 == 0) {
            evensFromTail.add(0, firstItem);
        }
        return evensFromTail;
    }
}
```

Problem 4: [28 points]

We wrote the recursive method above in class. It takes a list of integers and returns a new list containing just the even numbers from the input list.

a) What kind of input list does it take? ArrayList? LinkedList? Other?

b) Assume that you could choose what kind of list it could take as input. Would the Big-O complexity be different if it took an ArrayList versus a LinkedList? If so, which would be more efficient? Explain your answer.

c) What kind of output list does it produce? ArrayList? LinkedList? Other?

d) Assume that you could choose what kind of output list was produced. Would the Big-O complexity be different if it produced an ArrayList versus a LinkedList? If so, which would be more efficient? Explain your answer.

java.util

Interface Queue<E>

Type Parameters:
`E` - the type of elements held in this collection

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingDeque<E>, BlockingQueue<E>, Deque<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

```
public interface Queue<E>
extends Collection<E>
```

A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations; in most implementations, insert operations cannot fail.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

The `offer` method inserts an element if possible, otherwise returning `false`. This differs from the `Collection.add` method, which can fail to add an element only by throwing an unchecked exception. The `offer` method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The `remove()` and `poll()` methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The `remove()` and `poll()` methods differ only in their behavior when the queue is empty: the `remove()` method throws an exception, while the `poll()` method returns `null`.

The `element()` and `peek()` methods return, but do not remove, the head of the queue.

The `Queue` interface does not define the *blocking queue methods*, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the `BlockingQueue` interface, which extends this interface.

`Queue` implementations generally do not allow insertion of `null` elements, although some implementations, such as `LinkedList`, do not prohibit insertion of `null`. Even in the implementations that permit it, `null` should not be inserted into a `Queue`, as `null` is also used as a special return value by the `poll` method to indicate that the queue contains no elements.

`Queue` implementations generally do not define element-based versions of methods `equals` and `hashCode` but instead inherit the identity based versions from class `Object`, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

This interface is a member of the Java Collections Framework.

Since:
1.5

See Also:
`Collection`, `LinkedList`, `PriorityQueue`, `LinkedBlockingQueue`, `BlockingQueue`, `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`

Method Summary

`boolean`	**`add`**`(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
`E`	**`element`**`()` Retrieves, but does not remove, the head of this queue.
`boolean`	**`offer`**`(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
`E`	**`peek`**`()` Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
`E`	**`poll`**`()` Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
`E`	**`remove`**`()` Retrieves and removes the head of this queue.