Computer Science 136 Midterm Exam

Possible Answers

1. True/false statements (2 points each). Justify each answer with a sentence or two.

a. Two instances of `class Association` in the `structure` package are equal if and only if their `keys` are equal, regardless of their `values`.

> True, because the `equals()` method for an `Association` is implemented with this behavior. We can verify this using the online structure5 documentation, the structure5 source code, or the description on page 16 in the textbook.

b. An instance variable declared as `protected` can be accessed by any non-static method of the class in which it is declared.

> True. All instance variables can be accessed by any method of the class in which it is declared. (Protected variables can also be accessed by classes that are subclasses of the class in which it is declared). Chapter 1.3 in the textbook describes the ways we often use the `public`, `private`, `protected`, and `static` keywords within our classes (espeically in refence to the encapsulation), and Appendix B.8 gives more details about the specifics.

c. A binary search can locate a value in a sorted `Vector` in $O(\log n)$ time.

> True. We can apply the "divide and conquer" approach to halve the search size at each step. Since the list is sorted, we can rule out at least half of the values in our search space simply by examining the middle value:
>
> - if our target value is greater than (less than) the number, we know it can't be in the leftmost (rightmost) half of the list. These computations can be done in $O(1)$ time.
> - The number of times you can divide a number n by the a number b until you get to 1 is $\log_b(n)$. Once we have a list of size 1, the base case gives the solution in $O(1)$ time.
> - (In computer science, we commonly use 2 as the base in our logs, so when we say "$\log(n)$", we mean $\log_2(n)$ by default. If we want to describe something other than $\log_2$, then we must specify the base of the log explicitly.)

d. A binary search can locate a value in a sorted `SinglyLinkedList` in $O(\log n)$ time.

False. Since list elements are not "random access" (i.e., there is no direct reference to the list's middle node without traversing all of the links between neighboring nodes starting at the head of the list), we cannot apply a binary search effectively: we must walk through the list linearly in order to arrive at the "middle". Even though we still only perform $O(\log n)$ comparisons, we must peform $O(n)$ work to traverse the linked list in order to make each comparison.

e. If a method that has no preconditions is called, all of that method's postconditions should be guaranteed to be true when the method returns.

True. Having no precondition implies that the method should work correctly for *all* inputs, and the method should terminate with its postcondition true.

Preconditions, postconditions, and assertions are covered in Chapter 2. Note that preconditions and postconditions are simply expressed as comments, and they therefore cannot be enforced by the language. It is our job as programmers to provide accurate preconditions and postconditions. Assertions, on the other hand, let us verify that certain assumptions hold true at program *runtime*.

f. The Unix command `cp /path/to/directory` changes your current working directory to `/path/to/directory`.

False. `cp` is used to copy files and directories, `cd` changes your current working directory.

g. Instance variables are specified in an interface file.

False. Instance variables are specified in the class definition, not in the interface. It is not possible in Java to specify an instance variable in an interface.

An interface is like a contract. It provides a list of methods that an implementing class *must* complete, but it does not specify any details as to how.

An abstract class may provide some implementation.

Interfaces, abstract classes, and ways to use them in program design are covered in Chapter 7.

2. Consider the following Java program:

```
class Container {
    protected int count;
    protected static int staticCount;

    public Container(int initial) {
```

```
        count = initial;
        staticCount = initial;
    }

    public void setValue(int value) {
        count = value;
        staticCount = value;
    }

    public int getCount() {
        return count;
    }

    public int getStaticCount() {
        return staticCount;
    }
}

class WhatsStatic {

    public static void main(String[] args) {
        Container c1 = new Container(17);
        System.out.println("c1 count=" +c1.getCount()+
                           ", staticCount=" + c1.getStaticCount());

        Container c2 = new Container(23);
        System.out.println("c1 count=" +c1.getCount()+
                           ", staticCount=" + c1.getStaticCount());
        System.out.println("c2 count=" +c2.getCount()+
                           ", staticCount=" + c2.getStaticCount());

        c1.setValue(99);
        System.out.println("c1 count=" +c1.getCount()+
                           ", staticCount=" + c1.getStaticCount());
        System.out.println("c2 count=" +c2.getCount()+
                           ", staticCount=" + c2.getStaticCount());

        c2.setValue(77);
        System.out.println("c1 count=" +c1.getCount()+
                           ", staticCount=" + c1.getStaticCount());
        System.out.println("c2 count=" +c2.getCount()+
                           ", staticCount=" + c2.getStaticCount());
```

```
    }
}
```

a. What will the output be when the program is run (`java WhatsStatic`)? Assume no exceptions occur. (4 points)

```
c1 count=17, staticCount=17
c1 count=17, staticCount=23
c2 count=23, staticCount=23
c1 count=99, staticCount=99
c2 count=23, staticCount=99
c1 count=99, staticCount=77
c2 count=77, staticCount=77
```

b. What memory is allocated for `Containers` `c1` and `c2` at the time `c1.setValue(99)` is executed? Show any existing local variables and instance variables. (6 points)

The references to our two objects are stored in `c1` and `c2`, which are local variables to the `main` method. The two objects themselves exist, each with its own copy of instance variable `count`. They share one copy of `staticCount`.

3. In this problem you are to design a Java interface and class for a data structure which represents sets of Strings. As usual for sets, no repeated elements are allowed. Thus, the collection `"Propser"`, `"Anya"`, `"Lisa"`, `"Karl"`, `"Isabella"` is a legal set, but `"Bill"`, `"Duane"`, `"Bill"` is not. This data structure will have two methods:

- `void insert(String myString)` adds `myString` to the set if it is not already in the set; if `myString` is already in the set then `insert` does nothing.
- `boolean contains(String myString)` returns a boolean value indicating if `myString` is an element of the set.

a. Write a legal Java `interface` called `StringSetInterface` for this data structure. Include javadoc-style comments for preconditions and postconditions for the methods, as well as their return values where applicable. (You do not need to create javadoc-style comments for the parameters.) (6 points)

```
public interface StringSetInterface {

  /**
  *  @pre: none
  *  @post: myString is present exactly once in the set
  */
  public void insert(String myString);

  /**
  *  @pre: none
  *  @post: none
  *  @return: return value indicates if myString was found in the set
  */
  public boolean contains(String myString);
}
```

b. Suppose we decide to implement `StringSetInterface` by a class in which a `structure5 SinglyLinkedList` holds all of the elements. Write the definition of this class. This should be a full and legal Java class definition *with all method bodies filled in.* Don't forget to declare instance variables, include a constructor, and use qualifiers such as `public` and `protected` when appropriate. You need not repeat your javadoc-style comments for each method from part a. Please call your class `StringSet`. (10 points)

```
public class StringSet implements StringSetInterface {

  protected List<String> stringList;

  public StringSet() {
    stringList = new SinglyLinkedList<String>();
  }

  public void insert(String myString) {
    if (!contains(myString)) {
       stringList.add(new String(myString));
    }
  }

  public boolean contains(char myString) {
    return stringList.contains(myString);
  }
}
```

c. If `StringSet` is implemented as in part b, what would the worst-case time complexity be for the insert operation when the set has n elements? (Use "Big O" notation.) (4 points)

$O(n)$: the contains operation is $O(n)$, and the list insertion is $O(1)$. Summing, we get $O(n)$ time overall. (Note that if `insert` did not call `contains`, this method would be $O(1)$.)

d. Suppose we design an alternative implementation in which the set is represented by a `Vector<String>`called `strVec`. What is the worse-case complexity of insert with this representation? (6 points)

It is still $O(n)$: we don't benefit from a Vector's ability to do efficient random accesses in this case. In an unsorted Vector, the `contains` operation is already $O(n)$ time by itself.

If the vector is sorted, we can use binary search to check whether the String is already in the set, but we have to move elements to make room for any new String, which is still $O(n)$ in the worst case.

4. (15 points) Consider the following class, `ReversibleList`, that extends the `SinglyLinkedList` class by adding a method for reversing the list.

```
public class ReversibleList<E> extends SinglyLinkedList<E> {

  public ReversibleList() {
    super();
  }

  /**
  * @post: list is reversed
  */
  public void reverse() {
    if (head != null)
      head = recReverse(head);
  }

  /**
  * @pre: current is not null.
  * @post: list headed by current is reversed;
  *        and first Node in that list is returned.
  */
  private Node<E> recReverse(Node<E> current) {
    if (current.next() == null) {  // Single-node list
      return current;
    } else {
      Node<E> newHead = recReverse(current.next());  // Explain
      // current.next() now points to final node in reversed list!
      current.next().setNext(current);  // Explain
      current.setNext(null); // Explain
```

```
        return newHead;
      }
    }
}
```

a. What is the running time of `reverse()` (3 points)?

$O(n)$.

b. Prove using mathematical induction that your answer to part a is correct. (12 points)

We proceed by counting calls to the `setNext()` method, and claim that there are $2(n-1)$ calls needed to reverse a list of size n. We proceed by mathematical induction on n.

- Base: For a list of size 1, there are $2(1-1) = 0$ calls.
- Inductive hypothesis: Assume it takes $2(k-1)$ calls to `setNext()`, where $k < n$.
- Inductive step: For a list of size n, we know by the inductive hypothesis that the recursive call to `recReverse()` makes $2(n-2)$ calls to `setNext()`. We make two additional calls, giving a total of $2(n-2) + 2 = 2(n-1)$, which is $O(n)$.

5. Growth of functions. Using "Big O" notation, give the rate of growth for each of these functions. Briefly justify your answers (you do not need to use the definition of Big O; just explain how you found your solution). (3 points each, 12 total)

a. $f(x) = x^2 + 17x + 2001$

$O(x^2)$, since for sufficiently large x, x^2 is much larger than $17x$ or 2001.

b. $f(x) = \cos(x^4 + \log x)$

$O(1)$, since cos always remains in $[-1, 1]$.

c. $f(x) = 7x$ when x is odd, $f(x) = \frac{x}{7}$ when x is even.

$O(x)$, since $f(x)$ is always less than $7x$, and $7x$ is $O(x)$.

d. $f(x) = 5x^3$ for $x < 23$, $f(x) = 37$ otherwise.

$O(1)$, since $f(x) \leq 5 * (23)^3$ for all x.