

# Computer Science 136

Sample 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 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 (especially in reference 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 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.
- The number of times you can divide a number  $n$  by the base  $b$  until you get to 1 is  $\log_b(n)$ .
- (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 perform  $O(n)$  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.

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 the line `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.
- `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 preconditions and postconditions for the methods. (6 points)

```
public interface StringSetInterface {

    public void insert(String myString);
    // Pre: none. (OR Pre: myString is not in the set)
    // Post: myString is present in the set

    public boolean contains(String myString);
    // Pre: none.
    // Post: return value indicates if myString was found in the set
}
```

**It is up to you if you want to make `insert` of a `String` already in the set an error condition or just have it return.**

b. Suppose we decide to implement `StringSetInterface` by a class in which a singly-linked list 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 pre- and post- conditions from part a. Please call your class `StringSet`. (10 points)

```

public class StringSet implements StringSetInterface {

    protected List stringList;

    public StringSet() {
        stringList = new SinglyLinkedList();
    }

    public void insert(String myString) {
        // this corresponds to the no precondition answer above
        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  $O(n)$ , which is more significant than the  $O(1)$  list insertion.**

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  $O(n)$ , which is more significant than the cost to add at the end of the Vector. 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  $O(n)$  in the worst case.**

4. (20 points) Consider the following class, `ReversibleList`, that extends the `SinglyLinkedList` class by adding a method for reversing the list.

```

public class ReversibleList extends SinglyLinkedList {

    public ReversibleList() {
        super();
    }
}

```

```

public void reverse() {
    // Post: list is reversed.
    if (head != null)
        head = recReverse(head);
}

private static SinglyLinkedListNode<E> recReverse(SinglyLinkedListNode<E> current) {
    // Pre: current is not null.
    // Post: list headed by current is reversed; and first Node in that list is returned.
    if (current.next() == null) { // Single-node list
        return current;
    } else {
        SinglyLinkedListNode<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. Justify your answers. (3 points each, 12 total)

- a.  $f(x) = x^2 + 17x + 2001$   
 $O(x^2)$ , since we only care about the dominant term.
- 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  $\frac{x}{7}$  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 it only matters what happens as  $x$  gets large.

6. Searching and sorting.

- a. SelectionSort and Insertion both take  $O(n^2)$  in the worst case. However, they have different best-case running times. Explain why this difference occurs; include a description of examples that have best-case performance.

**SelectionSort** assumes the whole array is unsorted. It iteratively scans the unsorted portion of the array for the largest element, then places the largest element at the end of the unsorted portion. This reduces the unsorted region's size by 1. This algorithm is data independent, so it is always  $O(n^2)$ .

**Insertion** also assumes the whole array is unsorted. It iteratively searches the unsorted portion of the array for the first element that is out of place, then places it in the correct location in the sorted portion. If the array is already sorted, then there are no elements to move, reducing the runtime of InsertionSort to  $O(n)$ : a single pass through the array detects that it is sorted.

- b. When applied to an array, a MergeSort has three phases:

**Split:** Find the middle element of the array

**Recursively Solve:** MergeSort each half of the array

**Combine:** Merge the two sorted halves of the array into a single sorted array

As we've seen, the Split phase takes  $O(1)$  time while the Combine phase takes  $O(n)$  time. Suppose we want to implement MergeSort for a SinglyLinkedList data structure (with tail pointers). Describe what would be involved in implementing the Split and Combine phases and how much time (in the  $O()$  worst-case sense) each phase would take. Would such a MergeSort still take  $O(n \log n)$  time? Why?

**Unlike arrays, SLLs are not random-access data structures: to find an element in a SLL, you must traverse the list element by element, which is  $O(n)$ . But once you have a pointer to an element, splitting the list is  $O(1)$ , since you just unlink a node from its predecessor, and add it the head of a new list. Thus, splitting takes  $O(n)$ .**

**Merging two SLLs is still  $O(n)$ : you must walk through the list nodes**

in order, compare nodes, and add the smallest to a sorted list.

The runtime is still  $O(n \log n)$  because you have  $O(\log n)$  levels and you do  $O(n)$  work at each level. The difference is that you do  $O(n)$  work at each level on the “way down” (splitting) *and* on the “way back up” (merging). Since we ignore constants, doubling the work does not affect the big-O runtime: the factor of 2 is independent of  $n$ .