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 `key`s are equal, regardless of their `value`s.

      True, because the `equals()` method for an `Association` behaves this way.

   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)

   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.

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

      False. Since there is no direct access to the middle of the list, we cannot apply a binary search. We must walk through the list linearly.

   e. A method with no precondition should return with its postcondition true every time it is called.

      True. No precondition implies that the method should work correctly for all inputs and terminate with its postcondition true.

   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.

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 CharSetInterface {

  public void insert(char myString);
  // Pre: none. (OR Pre: myString is not in the set)
  // Post: newString is added to the set

  public boolean contains(char 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(SinglyLinkedListElement<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 {
           SinglyLinkedListElement<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. Prove by induction that `reverse()` behaves correctly. (Hint: focus on `recReverse(current)`) Include a *brief* explanation of each step labeled with the "Explain" comment. (8 points).

We proceed by induction on the length of the list passed to `recReverse()`.

- **Base: `recReverse(current)` when `current.next() == null` corresponds to the one-element case, and clearly works.**
- **Inductive hypothesis: Suppose the method works for lists up to size $n-1$.**
- **Inductive step: We first make the recursive call `recReverse(current.next())` which, by the inductive hypothesis, correctly reverses items 2 through n. It remains to place `current` in its correct position, at the end of the reversed list. `current.next()` is a reference to the former second element of the list,**

now the last element of the reversed list. So we set that element's `next` reference to be `current`, and set `current`'s `next` reference to `null`, and the list of size n is now reversed.

b. What is the running time of `reverse()` (2 points)?

$O(n)$**.**

c. Prove using mathematical induction that your answer to part b is correct. (10 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.