Midterm Solutions

October 11, 2001. 5:00–6:15 PM

There are six problems each worth five points for a total of 30 points. Show all your work, partial credit will be awarded. When there is not enough room on the test page itself, write in the provided blue books and write and sign your name on each one. No notes, no collaboration.

Name: ______________________________

Problem	Credit
1	
2	
3	
4	
5	
6	
Total	

1. Fill in the method bodies of addToCounter and getCounter to finish the following program.

```
class ProblemOne
{
   public static void main(String [] args)
   {
       MyObjectOne moo = new MyObjectOne() ;
       moo.addToCounter(5) ;
       moo.addToCounter(2) ;
       System.out.println(moo.getCounter()) ;
   }
}

class MyObjectOne
{
    int counter = 0 ;
    void addToCounter( int numberToAdd )
    {
        // increment counter by numberToAdd
        counter = counter + numberToAdd ; /* PROBLEM SOLN */
    }

    int getCounter()
    {
        // return the value of counter
        return counter ;  /* THE PROBLEM SOLUTION */
    }
}
```

2. Suppose we have a LinkedList ADT which is required to efficiently return the current length of the list, where length of a linked list is the number of nodes on the list. It is too time-consuming to count afresh the number of nodes each time getLength is invoked, so you get the great idea of keeping a “hidden” variable which is always updated with the current linked list length. Complete the following two methods inside the class ProblemTwo to implement this idea.

```
class ProblemTwo
{
    private int count = 0 ;
    private Node root = null ;

    void insertAtHead(String s)
    {
        Node n = new Node() ;
        n.content = s ;
        n.next = root ;
        root = n ;
        // add code to keep count variable current
        count++ ; /* THE PROBLEM SOLUTION */
     }

     int getLength()
     {
        // return the length of the linked list
        return count ; /* THE PROBLEM SOLUTION */
     }
}
```

3. In this version of a linked list, rather than insert the same string twice, we first try to find the string on the linked list, and if it is on the list, instead we increment its count. Finish the code below.

```
class ProblemThree {

    private int count = 0 ;
    private NodeThree root = null ;

    void insertAtHead(String s) {
        NodeThree n = findOnList(s) ;
        if ( if n!=null ) {
            // ASSERT n.contents.equals(s)==true
            // don't re-insert, adjust count
            n.count ++ ; /* THE PROBLEM SOLUTION */
        }
        else {
           // insertAtHead, as Problem Threee
           // code omitted, assume it's here
        }
    }

    NodeThree findOnList(String s) {
        // returns n such that n.content.equals(s)==true
        // or null, if no such n exists.

        // code omitted, assume it's here and it works!
    }
}

class NodeThree {

   NodeThree next = null ;
   String content = null ;
   int count = 1 ;
}
```

4. So then you have another great idea. To do a delete, rather than removing the element from the list, because this is a pain, you simply decrement its count. Assume class ProblemThree is reproduced below, you just have to fill in the deleteFromList method.

```
class ProblemFour
{
   // instance variables and methods as in class ProblemThree

   void deleteFromList(String s)
   {
      // do the find, if found,  check count>0, and if so
      // decrement count
      /* PROBLEM SOLUTION START */
      NodeThree nt = findOnList(s) ;
      if ( (nt!=null) && (nt.count>0) )
         nt.count-- ;
      /* PROBLEM SOLUTION END */
   }
}
```

5. Now you are troubled. You imagine your list full of elements with count zero. So you decide to write a method, cleanUpList, which runs the list and deletes nodes with count zero. Write it.

```
class ProblemFive
{
   //  instance variables and methods as in class ProblemFour

   void cleanUpList() {
   /* PROBLEM SOLUTION BEGIN HERE */

        // deal with zero-count nodes at head of list
        while ( (root!=null) && (root.count==0) )
            root = root.next ;
        // Postcondition: either list is empty or first
        // element in list has count>0

        NodeThree nt = root ;
        // LOOP INVARIANT: (nt==null) || (nt.count>0)
        while ( nt!=null ) {
           if ( (nt.next!=null) && (nt.next.count==0))
               nt.next = nt.next.next ;
           else nt = nt.next ;
        }
        // Argue termination: each time through the loop
        // nt gets one nearer the end of the list.
        // L.I. + Termination == GOAL
        // nt has pointed to each thing that has remained
        // on the list, and by invariant, if nt!=null then
        // nt.count>0.
   /* PROBLEM SOLUTION END HERE */
   }
}
```

6. Then you begin to wonder about who will call cleanUpList? So you decide that the linked list object should take care of cleaning up after itself. Also, since the user of the linked list object needn't know about count zero nodes, they should be "hidden" from the public version of the find method. That is, the post-condition on the return value n of the public find method is:

   ```
   ASSERT: (n==null) || (n.count>0)
   ```

   Discuss how you decided to do all this and produce Java code for the affected methods.

   Justify why this "lazy deletion" (marking a record as deleted and really deleting it at some later time, when it is more convenient) is just as efficient as non-lazy (immediate) deletion.

   Solution: Oen approach is to include list clean-up during list search. List print can remain the same, it may be important for debugging to see the zero-count nodes. It depends what the purpose of the print routine is. The result is nearly as efficient as immediate deletion since a deleted node is handled just one extra time, during the search step that truely deletes it.

6. (extra workspace)

```
NodeThree findOnList(String s)
{
   // clean up root
   while ( (root!=null) && (root.count==0) )
       root = root.next ;
   // ASSERT: list is empty or root.count>0.

   NodeThree nt = root ;
   // L.I.: (nt==null) || (nt.count>0)
   while ( nt!=null )
   {
       // clean-up element following nt in list
       if ( (nt.next!=null) && (nt.next.count===0) )
          nt.next = nt.next.next ;
       else {
          // when clean, check nt for find or move on
          if ( nt.content.equals(s) ) break ;
          nt = nt.next ;
       }
   }
   return nt ;
}
```