Problem 1
You were to add exactly three assignment statements (one per blank) to a destructive reverseHelper method to complete the List reverse method. (“Destructive” was added during the exam.) The code was as follows:

```java
public void reverse ( ) {
    if (myHead != null) {
        myHead = reverseHelper (myHead);
    }

    private static ListNode reverseHelper (ListNode head) {
        ListNode soFar = null;
        for (ListNode p=head; p!=null; /* no increment */ ) {
            ListNode temp = p.myRest;
            _____________________________________________;
            _____________________________________________;
            _____________________________________________;
            p.myRest = soFar;
            soFar = p;
            p = temp;
        }
        return soFar;
    }
}
```

The solution:

```java
p.myRest = soFar;
soFar = p;
p = temp;
```
Problem 2

This problem asked you to analyze the following code.

```java
public void removeNulls () {
    if (myHead == null) {
        return;
    } else if (myHead.myItem == null) {
        myHead = myHead.myNext;
        removeNulls ();
    } else {
        helper (myHead);
    }
}

private void helper (ListNode predecessor) {
    if (predecessor.myNext == null) {
        return;
    } else if (predecessor.myNext.myItem == null) {
        predecessor.myNext = predecessor.myNext.myNext;
        removeNulls ();
    } else {
        helper (predecessor.myNext);
    }
}
```

In part a, you were to determine the number of comparisons made with myItem entries when this list contains exactly one null element, at position k. There are two cases:

- **k = 0.** There is one comparison in removeNulls, then one more comparison in removeNulls (in the recursive call), and then N–2 comparisons in helper, totalling \( N = N+k \).
- **k > 0.** There is one comparision in removeNulls, then k comparisons in helper, then one more in removeNulls, then N–2 more in helper, totalling \( N+k \).

Part b involved finding the approximate worst case comparison count. A recursive call to removeNulls is made for each null element; thus there are at most N+1 calls to removeNulls, and these contribute N+1 comparisons. A call to removeNulls that does not immediately encounter a null instead calls helper, which can each make up to N comparisons. Thus the number of comparisons is at most proportional to \( N^2 \).

Finally, part c asked for a list for which the number of comparisons was proportional to \( N^2 \). Two such lists appear below. (X stands for a nonnull and O stands for a null element.)

- XOXOXOXOXO ...
- XXX ... XXX000 ... 000
  (alternating nonnull and null)
  (first half nonnull, second half null)
Problem 3

This problem was to consider the redefinition of NumberElement and StringElement from project 1 to use inheritance by extending a revised Queryable class given below.

```java
public class Queryable implements Comparable {
    private Comparable myItem; // the value of the queryable object
    private String myName; // e.g. "number", "string"

    public Queryable (Comparable q, String name) {
        myItem = q;
        myName = name;
    }

    public int compareTo (Object arg) {
        return myItem.compareTo (((Queryable) arg).myItem);
    }

    public String toString ( ) {
        return myItem.toString ( );
    }

    public Queryable toQueryable (String s) {
        return this;
    }

    public String name ( ) {
        return myName;
    }

    public boolean isTotalable ( ) {
        return false;
    }

    public Queryable plus (Queryable x) {
        throw new IllegalStateException ("wrong type for plus");
    }

    public static boolean isLegalValue (String s) {
        return true;
    }
}
```

Before discussion of a solution and point allocation, it makes sense to compare this implementation to its counterpart in project 1. The NumberElement, StringElement, and DateElement classes in project 1 each had its own instance variable that stored the number, the string, or the date respectively. Here, the instance variable that stores the number, the string, or the date is in the superclass Queryable; it's named myItem. Moreover, myItem is private and therefore not directly accessible from the subclasses. Initializing myItem from, say, the NumberElement constructor must be done by calling the Queryable constructor.
On to the questions. Parts a and b required that you identify which of a given set of methods would need to override those provided in the Queryable class. The answers are:

<table>
<thead>
<tr>
<th>method</th>
<th>override Queryable method?</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringElement.compareTo</td>
<td>no; polymorphism ensures that the StringElement compareTo method is used</td>
</tr>
<tr>
<td>StringElement.name</td>
<td>no; same reason</td>
</tr>
<tr>
<td>NumberElement.toQueryable</td>
<td>yes; NumberElement.toQueryable must return an object of type NumberElement, not Queryable</td>
</tr>
<tr>
<td>NumberElement.toString</td>
<td>no</td>
</tr>
</tbody>
</table>

In part c, you were to supply the StringElement constructor. As noted above, it must call the superclass constructor:

```java
public StringElement (String s) {
    // s's value will be copied to myItem, and the type of value will be "string".
    super (s, "string");
}
```

Part d asked for the NumberElement value method. (This method is specific to NumberElements; it was used to handle the total, max, and min commands.) Here's a solution:

```java
public int value ( ) {
    return Integer.parseInt (toString ( ));
}
```

**Problem 4**

This problem involved computing the “width” of a binary tree. Here is a solution:

```java
public int width ( ) {
    return helper (myRoot);
}

private static int helper (TreeNode t) {
    if (t == null) {
        return 0;
    } else if (t.myLeft == null) {
        return Math.max (t.myHeight, helper (t.myRight));
    } else if (t.myRight == null) {
        return Math.max (t.myHeight, helper (t.myLeft));
    } else {
        return Math.max (t.myLeft.myHeight + t.myRight.myHeight + 1,
                         Math.max (helper (t.myLeft), helper (t.myRight)));
    }
}
```
Problem 5

Here, you were to identify places in the exprTree code to throw exceptions for illegal exceptions. The code appears below.

```java
1. private TreeNode exprTree (String expr) {
2.     if (expr.charAt (0) != '(') {
3.         return new TreeNode (expr);
4.     } else {
5.         int nesting = 0;
6.         int opPos = 0;
7.         for (int k=1; k<expr.length()-1; k++) {
8.             char c = expr.charAt(k);
9.             if (c == '(') {
10.                nesting++;
11.             } else if (c == ')') {
12.                nesting--;
13.             } else if (nesting == 0 && (c == '+' || c == '*')) {
14.                 opPos = k;
15.                 break;
16.             }
17.         }
18.         String opnd1 = expr.substring(1, opPos);
19.         String opnd2 = expr.substring(opPos+1, expr.length()-1);
20.         String op = expr.substring(opPos, opPos+1);
21.         return new TreeNode (op, exprTree(opnd1), exprTree(opnd2));
22.     }
23. }
```

There were four situations a perfect solution was to guard against:

1. The call to charAt in line 2 will crash if expr is empty. This requires a test for a nonempty string prior to line 2.

2. The call to substring in line 18 will crash if opPos ≤ 1; similarly, the call to substring in line 19 will crash if opPos ≥ expr.length – 2. This happens when the loop ends normally, that is, if the break statement at line 15 isn’t executed.

3. The test at line 2 checks for an opening parenthesis, but nothing checks for a closing parenthesis. This check could be made in several places (basically anywhere prior to the return in line 21).

4. Nothing checks that a variable is a single digit or letter. This check is required prior to line 3.