

Problem 1 (4 points)

Add exactly three assignment statements (one per blank) to the destructive reverseHelper method to complete the List reverse method. Don't change any of the existing code, and don't use new.

```
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;
            _____ ;
            _____ ;
            _____ ;
        }
        return soFar;
    }
}
```

Problem 2 (8 points)

In a recent quiz, you were asked to write a void List method named `removeNulls` that removed all null elements of the list to which it was applied. Consider the following solution to this question.

```
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);
    }
}
```

Part a

Suppose that this List contains N elements. Complete the following sentence.

The number of comparisons involving `myItem` entries resulting from a call to `removeNulls` when this List contains exactly one null list element and it appears in position k (positions start at 0) is _____

Briefly explain your answer.

Part b

Suppose again that this List contains N elements (with any number of null list elements). Complete the following sentence.

The maximum number of comparisons involving `myItem` entries that result from a call to `removeNulls` is proportional to _____

Briefly explain your answer.

Part c

Describe a list of N elements for which the number of comparisons of `myItem` values resulting from a call to `removeNulls` is proportional to your answer for part b.

Problem 3 (8 points)

In project 1, the various elements of the data base were instances of classes that implemented the Queryable interface. For this problem, you will consider redefinition of the StringElement and NumberElement classes to use inheritance.

Given below is a redefinition of the Queryable class.

```
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;
    }

    // Return the result of comparing this Queryable to arg.
    public int compareTo (Object arg) {
        return myItem.compareTo (((Queryable) arg).myItem);
    }

    // Return a printable representation of this Queryable.
    public String toString ( ) {
        return myItem.toString ( );
    }

    // Return the Queryable object whose printable representation is s.
    public Queryable toQueryable (String s) {
        return this;
    }

    // Return the name of this Queryable.
    public String name ( ) {
        return myName;
    }

    // Return true if this Queryable can be totalled; return false otherwise.
    public boolean isTotalable ( ) {
        return false;
    }

    // Returns the result of adding this Queryable to x.
    public Queryable plus (Queryable x) {
        throw new IllegalStateException ("wrong type for plus");
    }

    // Return true if s is the printable representation of some Queryable;
    // return false otherwise.
    public static boolean isLegalValue (String s) {
        return true;
    }
}
```

Part a

For each of the `StringElement` methods below, indicate whether or not it should be defined to override the corresponding `Queryable` method.

<i>method</i>	<i>override corresponding Queryable method?</i>
<code>StringElement.compareTo</code>	
<code>StringElement.name</code>	

Part b

For each of the `NumberElement` methods named below, indicate whether or not it should be defined to override the corresponding `Queryable` method.

<i>method</i>	<i>override corresponding Queryable method?</i>
<code>NumberElement.toQueryable</code>	
<code>NumberElement.toString</code>	

Part c

Write the `StringElement` constructor. Use methods defined in the `Queryable` class wherever possible; assume where necessary that they have been correctly overridden in the `StringElement` class.

```
public StringElement (String s) {
```

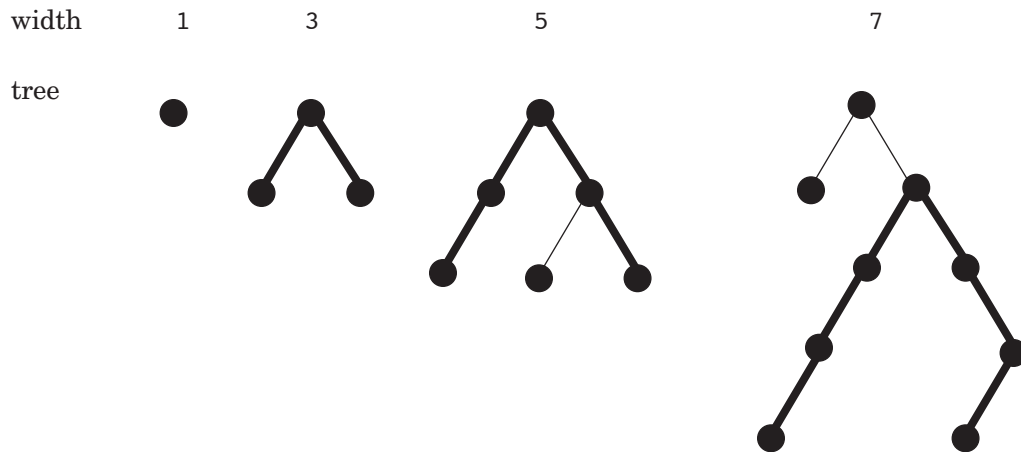
Part d

Write the `NumberElement` value method. It returns the `int` value of this `NumberElement`. You may assume that this `NumberElement` is properly initialized. Use methods defined in the `Queryable` class wherever possible; assume where necessary that they have been correctly overridden in the `NumberElement` class.

```
public int value ( ) {
```

Problem 4 (7 points)

The *width* of a tree is the maximal number of nodes on a path from one node in the tree to another. (The width of an empty tree is 0.) Here are some examples, with the relevant path indicated in bold. Note in particular that the relevant path does not necessarily include the root of the tree.



Write a `BinaryTree` method named `width` that returns the width of the tree. Assume that each tree node contains its height in a field named `myHeight` in an instance variable named `myHeight`. (The heights of the trees above are 1, 2, 3, and 5, respectively. The height of a tree is the height of its root.) You may find the method `Math.max` helpful; it returns the larger of its two `int` arguments.

```
public int width ( ) {
```

Problem 5 (8 points)

Given below is the `BinaryTree.exprTree` method you worked with in lab.

```
private TreeNode exprTree (String expr) {
    if (expr.charAt (0) != '(') {
        return new TreeNode (expr);
    } else {
        // expr is a parenthesized expression.
        // Strip off the beginning and ending expression,
        // find the main operator (an occurrence of + or * not nested
        // in parentheses), and construct the two subtrees.
        int nesting = 0;
        int opPos = 0;
        for (int k=1; k<expr.length()-1; k++) {
            char c = expr.charAt (k);
            if (c == '(') {
                nesting++;
            } else if (c == ')') {
                nesting--;
            } else if (nesting == 0 && (c == '+' || c == '*')) {
                opPos = k;
                break;
            }
        }
        String opnd1 = expr.substring (1, opPos);
        String opnd2 = expr.substring (opPos+1, expr.length()-1);
        String op = expr.substring (opPos, opPos+1);
        return new TreeNode (op, exprTree(opnd1), exprTree(opnd2));
    }
}
```

In lab, you assumed that the `String` argument was free of errors. For this problem, you are to supply the tests necessary to verify that assumption. A legal argument is a fully parenthesized arithmetic expression in which only the operators `+` and `*` are used, and in which operands are either legal expressions, single-character variable names, or single-digit nonnegative numbers. (This is slightly more restrictive than what you assumed in lab.)

In the table on the next page, list the following:

- the line number where you would place a test for some aspect of legality, and throw an exception if it failed;
- a description of the test you would make;
- the message you would supply as argument to the command `throw new IllegalArgumentException (...)`.

An example is provided to start you off; we think there are four more. Your tests should run in constant time where possible.

Problem 5, continued

```

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.}

```

<i>line number</i>	<i>test</i>	<i>error message</i>
before 2	expr == null	can't build tree from null string