Question 1. The code in Figure 1 defines a StringCollection class that contains some useful, but very poorly named, methods. Suppose that we have executed the statements:

```java
StringCollection words = new StringCollection(10);
words.add( "parts" );
words.add( "the" );
words.add( "all" );
words.add( "than" );
words.add( "greater" );
```

Indicate the String or int value that would be produced by each of the following expressions. Also provide a brief description of what the first method invoked in each expression does in general to a StringCollection.

a) `words.toString()`
   
   **Answer:** The String returned would be

   greater than all the parts

   In general, the toString method builds its result by concatenating together all the words in the array contents in the reverse order compared to how they were added to the collection. That is, the last word added appears first in the string returned (and so on).

   **Explanation:** The Strings added to a class are initially stored in the contents array in the order in which they were added. The first string appears in contents[0], the second and contents[1] and so on. The loop in toString also processes the Strings in the order that they are positioned in the array. The statement

   ```java
   result = contents[ i ] + " " + result;
   ```

   however, puts the i\text{th} element of contents before the elements that are already in result. Therefore, the last element processed by the loop which is the same as the last element added to the collection, will appear first in result.

b) `words.stern()`

   **The String returned would be**

   greater

   In general, the stern method returns the last element held in the array contents.

c) `words.cubits()`

   **The int returned would be**

   22

   In general, the cubits method returns total number of letters in all the Strings stored in the array contents.

d) Now, suppose we execute the statement:

   ```java
   words.deport( "all" );
   ```

   Describe the string that would be returned by evaluating the expression

   ```java
   words.toString()
   ```

   after the statement that invokes deport and describe the general operation performed by the deport method.

   **The String returned would be**

   han greater the parts

   In general, the deport method removes one copy of the word provided as a parameter from the array contents and reorders the words in contents by moving the last word into the place previously occupied by the word removed.
public class StringCollection {

    private int size = 0;
    private String[] contents;

    public StringCollection(int maxSize) {
        contents = new String[maxSize];
    }

    public void add(String element) {
        if (size < contents.length) {
            contents[size] = element;
            size = size + 1;
        }
    }

    public String toString() {
        String result = "";
        for (int i = 0; i < size; i = i + 1) {
            result = contents[i] + " " + result;
        }
        return result;
    }

    public void deport(String target) {
        int i = 0;
        while (i < size - 1 && !contents[i].contains(target)) {
            i = i + 1;
        }
        if (contents[i].contains(target)) {
            contents[i] = contents[size - 1];
            size = size - 1;
        }
    }

    public String stern() {
        return contents[size - 1];
    }

    public int cubits() {
        int result = 0;
        for (int i = 0; i < size; i = i + 1) {
            result = contents[i].length() + result;
        }
        return result;
    }
}

Figure 1: Code for StringCollection class
Question 2. Consider the following set of class definitions involving inheritance:

```java
public class Child {
    public Child() { }
    public int descendants() {
        return children();
    }
    public int children() { return 0; }
}

public class Parent extends Child {
    private final int MAX_FAMILY = 70;
    private int familySize = 0;
    private Child[] children = new Child[ MAX_FAMILY ];
    public Parent() { }
    public void addChild( Child newBorn ) {
        if ( familySize < MAX_FAMILY ) {
            children[ familySize ] = newBorn;
            familySize++;
        }
    }
    public int children() { return familySize; }
    public Child getChild( int birthOrder ) {
        if ( birthOrder < familySize ) {
            return children[ birthOrder ];
        } else {
            return null;
        }
    }
}

public class GrandParent extends Parent {
    public GrandParent() { }
    public int descendants() {
        int total = 0;
        for ( int r = 0; r < children(); r++ ) {
            total = total + getChild( r ).descendants() + 1;
        }
        return total;
    }
}
```
Suppose that the following code using this classes has been executed in some method:

```java
Child john2 = new Child();
Child jim = new Child();
Parent joan = new Parent();
joan.addChild(jim);
joan.addChild(john2);

Child johnjr = new Child();

GrandParent mary = new GrandParent();
mary.addChild(joan);
mary.addChild(johnjr);
```

a) What value would be produced by the invocation

```
mary.descendants()
```

if it occurred after the code shown was executed?

**Answer:** The expression would produce the value

4

**Explanation:** Invoking `descendants` on `mary` leads to the execution of the version of `descendants` that appears in the `GrandParent` class since the object associated with the name `mary` is a `GrandParent` object. The loop in this version of the method will add 1 to total for each of the objects added to `mary` (i.e. `joan` and `johnjr`) and also add in the results of invoking `descendants` on these two objects. Both of these invocations of `descendants` will lead to the execution of the code for the version of the `descendants` method in the `Child` class since `johnjr` is a `Child` object and `joan` belongs to the `Parent` class which inherits the version of `descendants` from `Child` since it extends `Child` and does not provide a new definition of `descendants`. Both of these invocations of `descendants` decide what to return by invoking `children`. Since `johnjr` is a `Child` object it will use the version of `children` defined in `Child` which returns 0. The invocation of `children` on `joan` however, will use the version of `children` defined in `Parent` since `joan` is a `Parent` object. This will return 2.

(worth 2 points)

b) Would the assignment

```
Child newKid = mary;
```

be legal in this context. If so, what value would be produced by the invocation

```
newKid.descendants()
```

if it occurred after the code shown was executed?

**Answer:** The assignment would be legal since the value of `mary` belongs to a class that extends `Child`, the type of the variable `newKid`. After the assignment, the expression `newKid.descendants()` will produce the value

4

since `newKid` refers to the same object a `mary` did in part (a).

(worth 2 points)

c) Would the assignment

```
GrandParent grandMa = joan;
```

be legal in this context. If so, what value would be produced by the invocation
grandMa.descendants()

if it occurred after the code shown was executed?

Answer: The assignment would be illegal since the variable joan is only known to refer to a value of type Parent. Although this value could also be a Grandparent, this is not certain so the assignment would not be allowed.  
(worth 2 points)

Question 3.

Table 1 shows the results of executing the main loop of Dijkstra’s shortest path algorithm several times to build a forwarding table for the router named A. Unfortunately, the table is incomplete and much of the information about the neighbors of the routers disappeared during a freak transcription accident involving cod liver oil and a small rodent.

Please follow the steps of Dijkstra’s Algorithm to complete the entries in the Best Route Length and First Step columns of the table. The rows in which the connections to neighboring routers rows are completely empty are the places where the information disappeared. You should leave these table entries blank. You will discover you do not need this information to complete the Best Route Length and First Step columns of the table if you simply follow algorithm 1 as described below. Assume that all edges have positive integer lengths.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J:2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>L</td>
<td>Adjacent</td>
<td>B: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 2</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>E</td>
<td>Adjacent</td>
<td>C: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 2</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td>Distant</td>
<td>L: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J: 2</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>L</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>10</td>
<td>C</td>
<td>Adjacent</td>
<td>I: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 2</td>
</tr>
<tr>
<td>I</td>
<td>10</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G: 3</td>
</tr>
<tr>
<td>J</td>
<td>5</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>C</td>
<td>Adjacent</td>
<td>F: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L: 2</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
<td>L</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>7</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 1 DISJKSTRA’S SHORTESTPATH

1: Mark starting point as \textit{KNOWN} with length 0
2: Identify each neighbor of start as \textit{ADJACENT}
3: Set first step of each neighbor of start to itself
4: Set route length of each neighbor to first step distance
5: While you don’t know the shortest path to all the cities:
   6: Select adjacent city with shortest ‘best route’ value
   7: Identify that adjacent city with shortest best route as \textit{KNOWN}
   8: Mark neighbors of new \textit{KNOWN} city that were \textit{DISTANT} as \textit{ADJACENT}
   9: Update path lengths and record first steps to \textit{ADJACENT} neighbors of new \textit{KNOWN} city.

Answer: We show the remaining un-‘known’ rows for each iteration of the loop. Vertex E is selected and made ‘Known’.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>11</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>E</td>
<td>Adjacent</td>
<td>B: 1</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>E</td>
<td>Known</td>
<td>C: 3</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td>Distant</td>
<td>L: 3</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>E</td>
<td></td>
<td>I: 1</td>
</tr>
<tr>
<td>I</td>
<td>10</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 3</td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>C</td>
<td>Adjacent</td>
<td>F: 1</td>
</tr>
</tbody>
</table>

Next H is selected and made Known.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>11</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>E</td>
<td>Adjacent</td>
<td>B: 1</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td>Distant</td>
<td>L: 3</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>E</td>
<td>Known</td>
<td>I: 1</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>E</td>
<td>Adjacent</td>
<td>K: 3</td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>C</td>
<td>Adjacent</td>
<td>F: 1</td>
</tr>
</tbody>
</table>

Next D (could have been I) is selected and made Known.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>10</td>
<td>E</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>E</td>
<td>Known</td>
<td>B: 1</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td>Distant</td>
<td>L: 3</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>E</td>
<td>Adjacent</td>
<td>K: 3</td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>C</td>
<td>Adjacent</td>
<td>F: 1</td>
</tr>
</tbody>
</table>

Next I is selected and made Known.
Next B is selected and made Known.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>10</td>
<td>E</td>
<td>Known</td>
<td>K: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J: 2</td>
</tr>
</tbody>
</table>

| F      | -                 |            | Distant| L: 3      |
|        |                   |            |        | J: 2      |

| K      | 11                | E          | Adjacent| F: 1      |
|        |                   |            |         | C: 3      |
|        |                   |            |         | L: 2      |

Next K is selected and made Known.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>12</td>
<td>E</td>
<td>Adjacent</td>
<td>L: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J: 2</td>
</tr>
</tbody>
</table>

| K      | 11                | E          | Known  | F: 1      |
|        |                   |            |         | C: 3      |
|        |                   |            |         | L: 2      |

Finally F is selected and made Known.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>12</td>
<td>E</td>
<td>Known</td>
<td>L: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J: 2</td>
</tr>
</tbody>
</table>

At this point all vertices are Known and so all shortest path lengths and first steps are correct.