1. Binary Tree Traversals. Consider the following tree.

```
        +----+
       | 2  |
        +----+
          /   \
         +----+   +----+   +----+
        | 7  |     | 6  |     | 1  |
       /   \
      +----+   +----+   +----+
     /      \    /      \    /      \ 
    +----+    +----+    +----+    +----+ 
   | 9  |     | 0  |     | 1  |     | 8  |
```

Fill in each of the traversals below:

Preorder traversal ________________________________

Inorder traversal ________________________________

Postorder traversal ________________________________

2. Binary Search Tree. Draw a picture below of the binary search tree that would result from inserting the following words into an empty binary search tree in the following order: Legolas, Frodo, Sam, Merry, Pippin, Aragorn, Gimli, Boromir.
3. Collections Mystery. Consider the following method:

```java
public List<Integer> mystery(int[][] data) {
    List<Integer> result = new LinkedList<Integer>();
    for (int i = 0; i < data.length; i++) {
        int sum = 0;
        for (int j = 0; j < data[i].length; j++) {
            sum = sum + j * data[i][j];
        }
        result.add(sum);
    }
    return result;
}
```

In the left-hand column below are specific two-dimensional arrays. Indicate in the right-hand column what values would be stored in the list returned by method mystery if the array in the left-hand column is passed as a parameter to mystery.

<table>
<thead>
<tr>
<th>Two-Dimensional Array</th>
<th>Contents of List Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[1, 2, 3], [4, 5, 6]]</td>
<td>_________________________</td>
</tr>
<tr>
<td>[[3, 4], [1, 2, 3], [], [5, 6]]</td>
<td>_________________________</td>
</tr>
<tr>
<td>[[1, 2, 3], [4, 5, 6], [7, 8, 9]]</td>
<td>_________________________</td>
</tr>
</tbody>
</table>
4. Details of Inheritance.

Assuming that the following classes have been defined:

```java
public class Gorge extends Cliff {
    public void method2() {
        System.out.println("Gorge 2");
    }
    public void method3() {
        System.out.println("Gorge 3");
    }
}

public class Hill extends Peak {
    public void method2() {
        System.out.println("Hill 2");
    }
    public void method3() {
        System.out.println("Hill 3");
    }
}

public class Peak {
    public void method1() {
        System.out.println("Peak 1");
        method3();
    }
    public void method3() {
        System.out.println("Peak 3");
    }
}

public class Cliff extends Peak {
    public void method3() {
        System.out.println("Cliff 3");
        super.method3();
    }
}
```

And assuming the following variables have been defined:

```java
Peak var1 = new Cliff();
Gorge var2 = new Gorge();
Peak var3 = new Hill();
Peak var4 = new Gorge();
Peak var5 = new Peak();
Object var6 = new Cliff();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a/b/c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with either the phrase "error".

<table>
<thead>
<tr>
<th>Statement</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var2.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var3.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var4.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var5.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var6.method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var1.method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var2.method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var3.method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var1.method3();</td>
<td>__________________________</td>
</tr>
<tr>
<td>var2.method3();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Gorge)var6).method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Cliff)var3).method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Gorge)var4).method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Gorge)var3).method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Hill)var3).method2();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Gorge)var1).method1();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Cliff)var4).method3();</td>
<td>__________________________</td>
</tr>
<tr>
<td>((Peak)var6).method3();</td>
<td>__________________________</td>
</tr>
</tbody>
</table>
5. Binary Trees. Write a toString method for a binary tree of integers. The method should return "empty" for an empty tree. For a leaf node, it should return the data in the node as a String. For a branch node, it should return a parenthesized String that has three elements separated by commas: the data at the root followed by a String representation of the left subtree followed by a String representation of the right subtree. For example, if a variable t stores a reference to the following tree:

```
+----+
| 2  |
+----+
    /   \
---+   +---+
| 8  |   | 1  |
+---+   +---+
    /     /\
---+   ---+
| 0  |   | 7  |   | 6  |
+---+   +---+
      /    /\
   ---+   ---+
| 4  |   | 9  |
   +---+   +---+
```

then the call t.toString() should return the following String:

"(2, (8, 0, empty), (1, (7, 4, empty), (6, empty, 9)))"

The quotes above are used to indicate that this is a String but should not be included in the String that you return.

You are writing a public method for a binary tree class defined as follows:

```java
public class IntTreeNode {
    public int data;          // data stored in this node
    public IntTreeNode left;  // reference to left subtree
    public IntTreeNode right; // reference to right subtree

    <constructors>
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not define any auxiliary data structures to solve this problem.
6. Collections Programming. Write a method called sumStrings takes a map whose keys are strings and whose values are points and that returns a map that associates each point with the sum of the lengths of the strings it is associated with in the first map.

For example, suppose that a map called data has the following associations:

{a=[x=1,y=3], apple=[x=7,y=7], be=[x=4,y=7], bear=[x=7,y=4], carpet=[x=2,y=19],
cat=[x=1,y=3], dog=[x=2,y=18], specialty=[x=7,y=4], student=[x=1,y=3],
umbrella=[x=42,y=8]}

Then the call sumStrings(data) should return the following map:

{[x=7,y=7]=5, [x=42,y=8]=8, [x=2,y=18]=3, [x=1,y=3]=11, [x=2,y=19]=6,
[x=7,y=4]=13, [x=4,y=7]=2}

Notice that the point [x=7,y=7] maps to 5 in the result because it was associated with a string of length 5 in the original ("apple"). Notice that [x=1,y=3] maps to 11 because it was associated with three strings ("a", "cat", "student") whose lengths add up to 11 (1, 3, 7).

Your method should construct the new map and can construct iterators but should otherwise not construct any new data structures. It should also not modify the map passed as a parameter and it should be reasonably efficient.

7. Comparable class. Define a class called TimeSpan that keeps track of an amount of time. A time span can be thought of in two ways. You can think of it as being a certain number of hours, minutes, and seconds where each hour is composed of 60 minutes and each minute is composed of 60 seconds. Or you can think of it as the total number of seconds. The class has the following public methods:

- **TimeSpan(hrs, min, sec)** constructs a TimeSpan object with the given hours, minutes, and seconds
- **hours()** returns the number of hours
- **minutes()** returns the number of minutes (0 to 59)
- **seconds()** returns the number of seconds (0 to 59)
- **totalSeconds()** returns the total number of seconds including minutes and hours components
- **add(other)** returns a new TimeSpan object formed by adding TimeSpan to the other TimeSpan
- **toString()** returns a string in the form "hhh:mm:ss"

For example, the following code constructs three TimeSpan objects:

```java
TimeSpan t1 = new TimeSpan(18, 3, 54);
TimeSpan t2 = new TimeSpan(95, 58, 7);
TimeSpan t3 = t1.add(t2);
```

The first TimeSpan object is expressed as 18 hours, 3 minutes, and 54 seconds. The totalSeconds method would report this as 65034 seconds. The call t1.toString() should produce "18:03:54". Notice that the minutes are reported as two digits even when it is a single digit. The same should be true of the seconds, so t2.toString() should produce the string "95:58:07". Your class should make sure that minutes and seconds are always reported as being between 0 and 59. When t1 and t2 are added together to produce t3, the resulting time should be reported as 114 hours, 2 minutes, and 1 second, with t3.toString() returning "114:02:01".

Your constructor should throw an IllegalArgumentException if any value passed to it is negative. It should fix values of minutes and seconds that are higher than 59, adjusting hours and minutes appropriately. For example, given the following call:

```java
TimeSpan t4 = new TimeSpan(4, 65, 100);
```

the resulting TimeSpan object should be reported as 5 hours, 6 minutes, and 40 seconds with t4.toString() returning "5:06:40". The TimeSpan class should implement the Comparable interface, where a shorter amount of time is considered less than a longer amount of time.
8. Binary Trees. Write a method called add that takes as a parameter a reference to a second binary tree and that adds the values in the second tree to this tree. If the method is called as follows:

```java
    tree1.add(tree2);
```

it should add all values in tree2 to the corresponding nodes in tree1. In other words, the value stored at the root of tree2 should be added to the value stored at the root of tree1 and the values in tree2's left and right subtrees should be added to the corresponding positions in tree1's left and right subtrees. The values in tree2 should not be changed by your method.

```
Initial tree1            Initial tree2             Final tree1
+----+                  +----+                   +----+
| 1  |                    | 2  |                    | 3  |
+----+                  +----+                   +----+
| 4  |                    | 3  |                    | 1  |
+----+                  +----+                   +----+
```

If tree1 has a node that has no corresponding node in tree2, then that node is unchanged. For example, if tree2 is empty, tree1 is not changed at all.

It is also possible that tree2 will have one or more nodes that have no corresponding node in tree1. For each such node, create a new node in tree1 in the corresponding position with the value stored in tree2's node. For example:

```
Initial tree1            Initial tree2             Final tree1
+----+                  +----+                   +----+
| 6  |                    | 1  |                    | 7  |
+----+                  +----+                   +----+
| 2  |                    | 4  |                    | 5  |
+----+                  +----+                   +----+
| 8  |                    | 2  |                    | 8  |
+----+                  +----+                   +----+
```

You are writing a method that will become part of the IntTree class. You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available. You are NOT to replace any of the existing nodes in the tree. You will, however, construct new nodes to be inserted into the tree as described above.
9. Linked Lists. Write a method called `rearrange` that rearranges the order of a list of integers so that all of the values in even-numbered positions appear in reverse order followed by all of the values in odd-numbered positions in forward order. We are using zero-based indexing, as with Java arrays and lists, where the first element is considered to be at position 0. For example, if a variable called `list` stores these values:

\[0, 1, 2, 3, 4, 5, 6, 7, 8, 9\]

and you make the following call:

```java
list.rearrange();
```

the list should store the following values after the call:

\[8, 6, 4, 2, 0, 1, 3, 5, 7, 9\]

In this example the values in the original list were equal to their positions and there were an even number of elements, but that won't necessarily be the case. For example, if the list had instead stored:

\[3, 8, 15, 9, 4, 42, 5\]

then after a call on `rearrange` it would store:

\[5, 4, 15, 3, 8, 9, 42\]

If the list has fewer than two elements, it should be unchanged by a call on `rearrange`.

You are writing a public method for a linked list class defined as follows:

```java
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // link to next node in the list
    <constructors>
}

public class LinkedIntList {
    private ListNode front;
    <methods>
}
```

You are writing a method that will become part of the `LinkedIntList` class. You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available. You are allowed to define your own variables of type ListNode, but you may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve this problem by rearranging the links of the list. Your solution must run in O(n) time where n is the length of the list.
1. Preorder traversal 2, 7, 9, 0, 5, 3, 6, 1, 4, 8
   Inorder traversal 9, 7, 5, 0, 3, 2, 6, 4, 1, 8
   Postorder traversal 9, 5, 3, 0, 7, 4, 8, 1, 6, 2

2. 
   +------------+
   | Legolas     |
   +------------+
   /              /
   /              /
   +------------+  +------------+
   | Frodo      |  | Sam         |
   +------------+  +------------+
   /              /
   /              /
   +------------+  +------------+
   | Aragorn    |  | Merry       |
   +------------+  +------------+
   /              /
   /              /
   +------------+  +------------+
   | Boromir    |  | Pippin      |
   +------------+  +------------+

3. Two-Dimensional Array
   Contents of List Returned
   --------------------------------------------
   
   [[1, 2, 3], [4, 5, 6]]                 [8, 17]
   [[3, 4], [1, 2, 3], [], [5, 6]]        [4, 8, 0, 6]
   [[1, 2, 3], [4, 5, 6], [7, 8, 9]]      [8, 17, 26]

4. Statement Output
   var1.method1();                  Peak 1/Cliff 3/Peak 3
   var2.method1();                  Peak 1/Gorge 3
   var3.method1();                  Peak 1/Hill 3
   var4.method1();                  Peak 1/Gorge 3
   var5.method1();                  Peak 1/Peak 3
   var6.method1();                  error
   var1.method2();                  error
   var2.method2();                  Gorge 2
   var3.method2();                  error
   var1.method3();                  Cliff 3/Peak 3
   var2.method3();                  Gorge 3
   var3.method3();                  Hill 3
   ((Gorge)var6).method1();         error
   ((Cliff)var3).method2();         error
   ((Gorge)var4).method2();         Gorge 2
   ((Gorge)var3).method2();         error
   ((Hill)var3).method2();          Hill 2
   ((Gorge)var1).method1();         error
   ((Cliff)var4).method3();         Gorge 3
   ((Peak)var6).method3();          Cliff 3/Peak 3
5. One possible solution appears below.

```java
public String toString() {
    return toString(overallRoot);
}

private String toString(IntTreeNode root) {
    if (root == null)
        return "empty";
    else if (root.left == null && root.right == null)
        return "+" + root.data;
    else
        return "(" + root.data + ", " + toString(root.left)
    + ", " + toString(root.right) + ")";
}
```

6. One possible solution appears below.

```java
public static Map<Point, Integer> sumStrings(Map<String, Point> data) {
    Map<Point, Integer> result = new HashMap<Point, Integer>();
    for (String s : data.keySet()) {
        Point p = data.get(s);
        if (!result.containsKey(p)) {
            result.put(p, s.length());
        } else {
            result.put(p, result.get(p) + s.length());
        }
    }
    return result;
}
```

7. Two possible solutions appear below.

```java
public class TimeSpan implements Comparable<TimeSpan> {
    private int totalSeconds;

    public TimeSpan(int hours, int minutes, int seconds) {
        if (hours < 0 || minutes < 0 || seconds < 0) {
            throw new IllegalArgumentException();
        }
        totalSeconds = seconds + 60 * (minutes + 60 * hours);
    }

    public int hours() {
        return totalSeconds / 3600;
    }

    public int minutes() {
        return totalSeconds % 3600 / 60;
    }

    public int seconds() {
        return totalSeconds % 60;
    }

    public int totalSeconds() { return totalSeconds;
    }
```
public String toString() {
    return hours() + ":" + minutes() / 10 + minutes() % 10 + ":" +
    seconds() / 10 + seconds() % 10;
}

public TimeSpan add(TimeSpan other) {
    TimeSpan result = new TimeSpan(0, 0, 0);
    result.totalSeconds = totalSeconds + other.totalSeconds;
    return result;
}

public int compareTo(TimeSpan other) {
    return this.totalSeconds - other.totalSeconds;
}

public class TimeSpan implements Comparable<TimeSpan> {
    private int hours, minutes, seconds;
    public TimeSpan(int hours, int minutes, int seconds) {
        if (hours < 0 || minutes < 0 || seconds < 0) {
            throw new IllegalArgumentException();
        }
        int total = seconds + 60 * (minutes + 60 * hours);
        this.seconds = total % 60;
        this.minutes = total / 60 % 60;
        this.hours = total / 3600;
    }

    public int hours() {
        return hours;
    }

    public int minutes() {
        return minutes;
    }

    public int seconds() {
        return seconds;
    }

    public int totalSeconds() {
        return seconds + 60 * (minutes + 60 * hours);
    }

    public String toString() {
        return hours + ":" + minutes / 10 + minutes % 10 + ":" +
        seconds / 10 + seconds % 10;
    }

    public TimeSpan add(TimeSpan other) {
        return new TimeSpan(hours + other.hours,
            minutes + other.minutes,
            seconds + other.seconds);
    }

    public int compareTo(TimeSpan other) {
        if (hours != other.hours)
            return hours - other.hours;
        else if (minutes != other.minutes)
            return minutes - other.minutes;
        else
            return seconds - other.seconds;
    }
}
8. One possible solution appears below.

```java
public void add(IntTree other) {
    overallRoot = add(overallRoot, other.overallRoot);
}

private IntTreeNode add(IntTreeNode root1, IntTreeNode root2) {
    if (root2 != null) {
        if (root1 == null)
            root1 = new IntTreeNode(0);
        root1.data += root2.data;
        root1.left = add(root1.left, root2.left);
        root1.right = add(root1.right, root2.right);
    }
    return root1;
}
```

9. One possible solution appears below.

```java
public void rearrange() {
    if (front != null && front.next != null) {
        ListNode current = front.next;
        while (current != null && current.next != null) {
            ListNode temp = current.next;
            current.next = current.next.next;
            temp.next = front;
            front = temp;
            current = current.next;
        }
    }
}
```