Problem 1: digitMatch

8. **Recursive Programming**. Write a recursive method `digitMatch` that accepts two non-negative integers as parameters and that returns the number of digits that match between them. Two digits match if they are equal and have the same position relative to the end of the number (i.e., starting with the ones digit). In other words, the method should compare the last digits of each number, the second-to-last digits of each number, the third-to-last digits of each number, and so forth, counting how many pairs match. For example, for the call of `digitMatch(1072503891, 62530841)`, the method would compare as follows:

```
1 0 7 2 5 0 3 8 9 1
      | | | | | | | |
      6 2 5 3 0 8 4 1
```

The method should return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). Below are more examples:

Call	Value Returned
`digitMatch(38, 34)`	1
`digitMatch(5, 5552)`	0
`digitMatch(892, 892)`	3
`digitMatch(298892, 7892)`	3
`digitMatch(380, 0)`	1
`digitMatch(123456, 654321)`	0
`digitMatch(1234567, 67)`	2

Your method should throw an `IllegalArgumentException` if either of the two parameters is negative. You are not allowed to construct any structured objects other than `Strings` (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.

Problem 2: switchPairs

Write a method `switchPairs` that switches the order of elements in a linked list of integers in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if the list initially stores these values:

```
[3, 7, 4, 9, 8, 12]
```

Your method should switch the first pair (`3`, `7`), the second pair (`4`, `9`), the third pair (`8`, `12`), etc. to yield this list:

```
[7, 3, 9, 4, 12, 8]
```

If there are an odd number of values, the final element is not moved. For example, if the list had been:

```
[3, 7, 4, 9, 8, 12, 2]
```

It would again switch pairs of values, but the final value (`2`) would not be moved, yielding this list:

```
[7, 3, 9, 4, 12, 8, 2]
```

Assume that we are adding this method to the `LinkedIntList` class as shown below. You may not call any other methods of the class to solve this problem, you may not construct any new nodes, and you may not use any auxiliary data structures to solve this problem (such as an array, `ArrayList`, `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.

```
    public class LinkedIntList {
        private ListNode front;
        ...
    }

    public class ListNode {
        public int data;
        public ListNode next;
        ...
    }
```

Problem 3: trim

Write a method `trim` that could be added to the `IntTree` class. The method accepts minimum and maximum integers as parameters and removes from the tree any elements that are not within that range, inclusive. For this method you should assume that your tree is a binary search tree (BST) and that its elements are in valid BST order. Your method should maintain the BST ordering property of the tree.

For example, suppose a variable of type `IntTree` called `tree` stores the following elements:

tree

```
                             +----+
                             | 50 |
                     _____   +----+   _____
                    /                      \
              +----+                        +----+
              | 38 |                        | 90 |
       _____  +----+                   ___  +----+
      /              \                /
+----+                +----+   +----+
| 14 |                | 42 |   | 54 |
+----+                +----+   +----+
 /    \                            \
+----+ +----+                       +----+
|  8 | | 20 |                       | 72 |
+----+ +----+                       +----+
           \                        /    \
            +----+            +----+    +----+
            | 26 |            | 61 |    | 83 |
            +----+            +----+    +----+
```

The table below shows what the state of the tree would be if various `trim` calls were made. The calls shown are separate; it's not a chain of calls in a row. You may assume that the minimum is less than or equal to the maximum.

```
tree.trim(25, 72);

              +----+
              | 50 |
           _  +----+
          /          \
     +----+           +----+
     | 38 |           | 54 |
     +----+           +----+
     /    \                \
+----+    +----+            +----+
| 26 |    | 42 |            | 72 |
+----+    +----+            +----+
                            /
                       +----+
                       | 61 |
                       +----+
```

```
tree.trim(54, 80);

+----+
| 54 |
+----+
    \
     +----+
     | 72 |
     +----+
     /
+----+
| 61 |
+----+
```

```
tree.trim(18, 42);

          +----+
          | 38 |
       _  +----+
      /          \
+----+           +----+
| 20 |           | 42 |
+----+           +----+
    \
     +----+
     | 26 |
     +----+
```

```
tree.trim(3, 7);
```

Hint: The BST ordering property is important for solving this problem. If a node's data value is too large or too small to fit within the range, this may also tell you something about whether that node's left or right subtree elements can be within the range. Taking advantage of such information makes it more feasible to remove the correct nodes.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc.

Assume that you are adding this method to the `IntTree` class as defined below:

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
}
```