CSE143X Final, Fall 2020

This is a closed-book/closed-note exam. There is a "cheat sheet" at the end. You are not allowed to access the internet or other sources during the exam.

You are NOT to use any electronic devices while taking the test, including calculators.

Give yourself 110 minutes to complete the exam and then scan it (preferably as a pdf) and upload it to the course web page.

1. Binary Tree Traversals, 6 points. Consider the following tree.

```
                                 +---+
                                 | 7 |
                                 +---+
                               /       \
                             /           \
                       +---+               +---+
                       | 2 |               | 9 |
                       +---+               +---+
                      /     \             /     \
                     /       \           /       \
                  +---+     +---+     +---+     +---+
                  | 3 |     | 4 |     | 6 |     | 0 |
                  +---+     +---+     +---+     +---+
                       \         \             /
                        \         \           /
                       +---+     +---+     +---+
                       | 8 |     | 1 |     | 5 |
                       +---+     +---+     +---+
```

Fill in each of the traversals below:

Preorder traversal ____

Inorder traversal ____

Postorder traversal ____

2. Recursive Programming, 9 points. Write a recursive method called digitMatch that takes two nonnegative 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 relative position starting from 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, digitMatch(1072503891, 62530841) 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:

```
     Method call                Result       Method call                Result
     --------------------------------       --------------------------------
     digitMatch(38, 34)            1        digitMatch(5, 5552)            0
     digitMatch(892, 892)          3        digitMatch(1234567, 67)        2
     digitMatch(298892, 7892)      3        digitMatch(380, 0)             1
     digitMatch(123456, 654321)    0        digitMatch(0, 4)               0
     digitMatch(42, 24)            0        digitMatch(0, 0)               1
```

Your method should throw an IllegalArgumentException if either of the two parameters is negative. You are not allowed to construct any structured objects to solve this problem (no array, String, StringBuilder, ArrayList, etc) and you may not use a while loop, for loop or do/while loop to solve this problem; you must use recursion.

3. Details of inheritance, 10 points. Assuming that the following classes have been defined:

```
public class Fork extends Pot {
    public void method2() {
        System.out.println("Fork 2");
        super.method2();
    }
}

public class Pot {
    public void method2() {
        System.out.println("Pot 2");
    }

    public void method3() {
        System.out.println("Pot 3");
        method2();
    }
}

public class Bowl extends Fork {
    public void method1() {
        System.out.println("Bowl 1");
    }

    public void method2() {
        System.out.println("Bowl 2");
    }
}

public class Spoon extends Pot {
    public void method1() {
        System.out.println("Spoon 1");
    }

    public void method2() {
        System.out.println("Spoon 2");
    }
}
```

And assuming the following variables have been defined:

```
        Pot var1 = new Spoon();
        Bowl var2 = new Bowl();
        Pot var3 = new Bowl();
        Pot var4 = new Pot();
        Object var5 = new Bowl();
        Pot var6 = new Fork();
```

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 "compiler error" or "runtime error" to indicate when the error would be detected.

Statement	Output
`var1.method2();`	____________________________
`var2.method2();`	____________________________
`var3.method2();`	____________________________
`var4.method2();`	____________________________
`var5.method2();`	____________________________
`var6.method2();`	____________________________
`var1.method1();`	____________________________
`var2.method1();`	____________________________
`var3.method1();`	____________________________
`var1.method3();`	____________________________
`var2.method3();`	____________________________
`var3.method3();`	____________________________
`var4.method3();`	____________________________
`((Spoon)var1).method1();`	____________________________
`((Bowl)var3).method1();`	____________________________
`((Fork)var3).method3();`	____________________________
`((Fork)var5).method1();`	____________________________
`((Spoon)var5).method1();`	____________________________
`((Fork)var6).method2();`	____________________________
`((Bowl)var6).method3();`	____________________________

4. Stacks/Queues, 10 points.   Write a method called alternatingReverse that takes a stack of integers as a parameter and that rearranges the values so that every other value starting from the bottom of the stack is reversed in order. For example, if a variable s stores these values:

```
     bottom [1, 2, 3, 4, 5, 6, 7, 8] top
             ^     ^     ^     ^
             |     |     |     |
             +-----+-----+-----+
             sequence to reverse
```

Starting from the bottom of the stack and looking at every other value, we find the sequence of numbers 1, 3, 5, 7. This sequence should be reversed while the other values should stay in the same positions. If we make the following call:

```
     alternatingReverse(s);
```

the stack should store the following values after the call:

```
     bottom [7, 2, 5, 4, 3, 6, 1, 8] top
             ^     ^     ^     ^
             |     |     |     |
             +-----+-----+-----+
              reversed sequence
```

This example uses sequential integers to make it easier to see the sequence, but you should not assume anything about the sequence. For example, if s instead stored this sequence:

```
     bottom [7, 1, 4, 18, 23, 0, -5, 12] top
```

then after the method is called, it would store this sequence:

```
     bottom [-5, 1, 23, 18, 4, 0, 7, 12] top
```

Your method should throw an IllegalArgumentException if the number of elements in the stack is not an even number.

You are to use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You also may not solve the problem recursively. Your solution must run in O(n) time where n is the size of the stack. Use the Stack and Queue structures described in the cheat sheet and obey the restrictions described there (recall that you can't use the peek method or a foreach loop or iterator).

Space is provided on the next page for your answer.

Please write your answer to alternatingReverse below.

5. Collections Programming, 5 points. Write a method called acronymFor that takes a list of strings as a parameter and that returns the corresponding acronym. You form an acronym by combining the capitalized first letter of a series of words. For example, the list [laughing, out, loud] produces the acronym "LOL". The list [Computer, Science and, Engineering] produces the acronym "CSE". You may assume that all of the strings are nonempty. Your method is not allowed to change the list passed to it as a parameter. If passed an empty list, your method should return the empty string.

   You may construct iterators and strings, but you are not allowed to construct other structured objects (no set, list, stack, queue, etc.).

6. Binary Trees, 10 points. Write a method called hasPathSum that takes an integer n as a parameter and that returns true if there is some nonempty path from the overall root of a tree to a node of the tree in which the sum of the data stored in the nodes adds up to n (returning false if no such path exists). For example if the variable t refers to the following tree:

```
                                 +----+
                                 |  5 |
                                 +----+
                              /          \
                      +----+                +----+
                      |  1 |                | 21 |
                      +----+                +----+
                     /      \                     \
                +----+       +----+               +----+
                | -9 |       |  2 |               | 20 |
                +----+       +----+               +----+
               /      \                          /      \
          +----+      +----+                +----+      +----+
          |  3 |      | 30 |                | 13 |      |  4 |
          +----+      +----+                +----+      +----+
```

Below are various calls and an explanation for the value returned:

```
     t.hasPathSum(8) returns true because of the path (5, 1, 2)
     t.hasPathSum(26) returns true because of the path (5, 21)
     t.hasPathSum(0) returns true because of the path (5, 1, -9, 3)
     t.hasPathSum(5) returns true because of the path (5)
     t.hasPathSum(1) returns false because no path with that sum exists
```

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

```
     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 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 call any other methods of the class. You may not construct any extra data structures to solve this problem.

7. Collections Programming, 10 points. Write a method called acronyms that takes a set of word lists as a parameter and that returns a map whose keys are acronyms and whose values are the word lists that produce that acronym. Acronyms are formed from each list as described in problem 4. Recall that the list [laughing, out, loud] produces the acronym "LOL". The list [League, of, Legends] also produces the acronym "LOL". Suppose that a variable called lists stores this set of word lists:

```
[[attention, deficit], [Star, Trek, Next, Generation],
 [laughing, out, loud], [International, Business, Machines],
 [League, of, Legends], [anno, domini], [art, director],
 [Computer, Science and, Engineering]]
```

Each element of this set is a list of values of type String. You may assume that each list is nonempty and that each string in a list is nonempty.

Your method should construct a map whose keys are acronyms and whose values are sets of the word lists that produce that acronym. For example, the call acronyms(lists) should produce the following map:

```
{AD=[[attention, deficit], [anno, domini], [art, director]],
 CSE=[[Computer, Science and, Engineering]],
 IBM=[[International, Business, Machines]],
 LOL=[[laughing, out, loud], [League, of, Legends]],
 STNG=[[Star, Trek, Next, Generation]]}
```

Notice that there are 5 unique acronyms produced by the 8 lists in the set. Each acronym maps to a set of the word lists for that acronym. Your method should not make copies of the word lists; the sets it constructs should store references to those lists. As in the example above, the keys of the map that you construct should be in sorted order. You may assume that a working version of acronymFor as described in problem 4 is available for you to use no matter what you wrote for problem 4. Your method is not allowed to change either the set passed as a parameter or the lists within the set.

8. Comparable class, 10 points. Write a class called ItemOrder that stores information about an item being ordered. Each ItemOrder object keeps track of its item number (a string), an integer quantity, and a price per item expressed as an integer number of pennies. For example:

```
     ItemOrder item = new ItemOrder("007", 3, 36);
```

Notice that the quantity is passed as a parameter before the price in the constructor, so this indicates an order for item number 007 with a quantity of 3 at 36 cents each. The total price for this order would be 108 cents. The class should include the following public methods:

```
    getPrice()   returns the total price for this order in pennies
    toString()   returns a String representation of the order
```

Below is a pattern for formatting the toString result.

```
     item #<item>: <quantity>@$<price per> = $<total price>
```

For example, given the variable defined above, item.toString() should return "item #007: 3@$0.36 = $1.08". Notice that prices are expressed as dollars and cents in the usual format with 2 digits for pennies.

The ItemOrder class should implement the Comparable<E> interface. Item orders should be ordered first by item number (sorted alphabetically) and then by total price (with lower total-priced orders appearing earlier).

9. Binary Trees, 15 points. Write a method called makeFull that turns a binary tree of integers into a full binary tree. A full binary tree is one in which every node has either 0 or 2 children. Your method should produce a full binary tree by replacing each node that has one child with a new node that has the old node as a leaf where there used to be an empty tree. The new node should store a value that indicates the level of the tree (-1 for the first level of the tree, -2 for the second level of the tree, and so on). For example, if a tree called t stores the following:

```
               +----+
               | 12 |
               +----+
              /
         +----+
         | 29 |
         +----+
```

and we make the call:

```
        t.makeFull();
```

then the tree should store the following after the call:

```
               +----+
               | -1 |
               +----+
              /      \
         +----+      +----+
         | 29 |      | 12 |
         +----+      +----+
```

Notice that the node storing 12 that used to be at the top of the tree is now a leaf where there used to be an empty tree. In its place at the top of the tree is a new node that stores the value -1 to indicate that it was added at level 1 of the tree. Your method should perform this operation at every level of the tree. For example, if t had instead stored:

```
                                 +----+
                                 | 12 |
                                 +----+
                              /          \
                      +----+                +----+
                      | 28 |                | 19 |
                      +----+                +----+
                     /                     /
                +----+                +----+
                | 94 |                | 32 |
                +----+                +----+
               /      \                     \
          +----+      +----+                +----+
          | 65 |      | 18 |                | 72 |
          +----+      +----+                +----+
```

then after the call it would store:

```
                                  +----+
                                  | 12 |
                                  +----+
                               /          \
                      +----+                  +----+
                      | -2 |                  | -2 |
                      +----+                  +----+
                     /      \                /      \
                +----+      +----+      +----+      +----+
                | 94 |      | 28 |      | -3 |      | 19 |
                +----+      +----+      +----+      +----+
               /      \                /      \
          +----+      +----+      +----+      +----+
          | 65 |      | 18 |      | 32 |      | 72 |
          +----+      +----+      +----+      +----+
```

<continued on next page>

Notice that two nodes were added at level 2, and one at level 3.

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

```
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

    // post: constructs an IntTreeNode with the given data and links
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

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 CHANGE THE DATA FIELD OF THE EXISTING NODES IN THE TREE (what we called You will, however, construct new nodes containing negative values to be inserted into the tree (notice that there is only one constructor for nodes). You will also change the links of the tree to restructure the tree as described. Your solution must run in O(n) time where n is the number of nodes in the tree.

10. Linked Lists, 15 points. Write a method of the LinkedIntList class called switchEvens that takes a second list of integers as a parameter and that switches values in even numbered positions between the first and second lists. For example, if the variables list1 and list2 store the following:

```
    list1 = [3, 9, 5, 4, 2]
    list2 = [7, 1, 0, 6, 18, 12, 8]
```

and you make the following call:

```
    list1.switchEvens(list2);
```

The method switches the values at index 0 (3 and 7), the values at index 2 (5 and 0), and the values at index 4 (2 and 18), as indicated below:

```
    list1 = [3, 9, 5, 4, 2]
             |     |     |
    list2 = [7, 1, 0, 6, 18, 12, 8]
```

After the method is called, the lists should store the following values:

```
    list1 = [7, 9, 0, 4, 18]
    list2 = [3, 1, 5, 6, 2, 12, 8]
```

Notice that it doesn't matter whether the numbers themselves are even but whether they appear in even positions. Also notice that if one list is longer than the other, then the values that don't have corresponding entries in the shorter list are left unchanged.

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

```
    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. Both lists are of type LinkedIntList. 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.

Write your solution to switchEvens on the next page.

Write your solution to switchEvens below.

11. Fiction, 1 point (bonus). Your TA has woken up in jail. Explain why. You may express your answer in any way you choose by writing a story, drawing a picture, writing a poem, etc. If your answer below indicates at least 1 minute of effort, you will receive full credit, although your TA would probably appreciate it if you put in a little more effort.

CSE143X Cheat Sheet

`Linked Lists (16.2)`

Below is an example of a method that could be added to the LinkedIntList class to compute the sum of the list:

```
public int sum() {
    int sum = 0;
    ListNode current = front;
    while (current != null) {
        sum += current.data;
        current = current.next;
    }
    return sum;
}
```

`Math` Methods (3.2)

mathematical operations

`Math.abs(`*value*`)`	absolute value
`Math.min(`*v1, v2*`)`	smaller of two values
`Math.max(`*v1, v2*`)`	larger of two values
`Math.round(`*value*`)`	nearest whole number
`Math.pow(`*b, e*`)`	b to the e power

`Stacks and Queues (14.2)`

(LIFO and FIFO structures)

Queues should be constructed using the Queue<E> interface and the LinkedList<E> implementation. For example, to construct a queue of String values, you would say:

```
Queue<String> q = new LinkedList<>();
```

Stacks should be constructed using the Stack<E> class (there is no interface):

```
Stack<String> s = new Stack<>();
```

For Stack<E>, you are limited to the following operations (no iterator or foreach loop):

`push(`**value**`)`	pushes the given value onto the top of the stack
`pop()`	removes and returns the top of the stack
`isEmpty()`	returns `true` if this stack is empty
`size()`	returns the number of elements in the stack

For Queue<E>, you are limited to the following operations (no iterator or foreach loop):

`add(`**value**`)`	adds the given value at the end of the queue
`remove()`	removes and returns the front of the queue
`isEmpty()`	returns `true` if this queue is empty
`size()`	returns the number of elements in the queue

`Iterator<E>` Methods (11.1)

(An object that lets you examine the contents of any collection)

`hasNext()`	returns `true` if there are more elements to be read from collection
`next()`	reads and returns the next element from the collection
`remove()`	removes the last element returned by `next` from the collection

`List<E>` Methods (10.1)

(An ordered sequence of values)

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value at given index, shifting subsequent values right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the list's size is 0
`addAll(`**collection**`)`	adds all elements from the given collection to the end of the list
`contains(`**value**`)`	returns `true` if the given value is found somewhere in this list
`remove(`**value**`)`	finds and removes the given value from this list if it is present
`removeAll(`**list**`)`	removes any elements found in the given collection from this list

`iterator()`	returns an object used to examine the contents of the list

`Set<E>` Methods (11.2)

(A fast-searchable set of unique values)

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in the set
`remove(`**value**`)`	removes the given value from the set if it is present
`clear()`	removes all elements of the set
`size()`	returns the number of elements in the set
`isEmpty()`	returns `true` if the set's size is 0
`addAll(`**collection**`)`	adds all elements from the given collection to the set
`containsAll(`**collection**`)`	returns `true` if set contains every element from given collection
`removeAll(`**collection**`)`	removes any elements found in the given collection from this set
`retainAll(`**collection**`)`	removes any elements *not* found in the given collection from this set
`iterator()`	returns an object used to examine the contents of the set

`Map<K, V>` Methods (11.3)

(A fast mapping between a set of keys and a set of values)

`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value
`get(`**key**`)`	returns the value mapped to the given key (`null` if none)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`keySet()`	returns a `Set` of all keys in the map
`values()`	returns a `Collection` of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map

`Point` Methods (8.1)

(an object for storing integer x/y coordinates)

`Point(`**x, y**`)`	constructs a new point with given x/y coordinates
`Point()`	constructs a new point with coordinates (0, 0)
`getX()`	returns the x-coordinate of this point
`getY()`	returns the y-coordinate of this point
`equals(`**other**`)`	returns true if this Point stores the same x/y values as the other
`translate(`**dx, dy**`)`	translates the coordinates by the given amount

`String` Methods (3.3)

(An object for storing a sequence of characters)

`length()`	returns the number of characters in the string
`charAt(`**index**`)`	returns the character at a specific index
`compareTo(`**other**`)`	returns how this string compares to the other
`equals(`**other**`)`	returns true if this string equals the other
`toUpperCase()`	returns a new string with all uppercase letters
`toLowerCase()`	returns a new string with all lowercase letters
`startsWith(`**other**`)`	returns true if this string starts with the given text
`substring(`**start, stop**`)`	returns a new string composed of character from start index (inclusive) to stop index (exclusive)

`Collections Implementations`

`List<E>`	`ArrayList<E>` and `LinkedList<E>`
`Set<E>`	`HashSet<E>` and `TreeSet<E>` (values ordered)
`Map<K, V>`	`HashMap<K, V>` and `TreeMap<K, V>` (keys ordered)