CSE143 Final, Spring 2021

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.

In general the exam is not graded on style and you do not need to include comments, although you are required to declare all data fields and helper methods as private, to use generics properly, to declare variables and parameters using interfaces when possible, and to avoid the use of break statements, the diamond operator, and return from a void method. You do not have to include any import statements. Do not abbreviate any code that you write (e.g., S.o.p versus System.out.print).

For standard Java classes such as Math and String, you are limited to the methods listed on the cheat sheet. You are not allowed to use the Arrays or Collections classes or other standard classes and methods that aren't included on the cheat sheet.

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.

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

Fill in each of the traversals below:

Preorder traversal ____

Inorder traversal   ____

Postorder traversal ____

2. Binary Search Tree, 4 points. 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: Picard, Riker, Worf, Crusher, Troi, LaForge, Data. Assume the search tree uses alphabetical ordering to compare words.

3. Collections Mystery, 5 points. Consider the following method:

```
     public Set<Integer> mystery(int[][] data, int pos) {
         Set<Integer> result = new TreeSet<Integer>();
         for (int i = 0; i < data.length; i++) {
             if (i % 2 == 0) {
                 result.add(data[pos][i]);
             } else {
                 result.add(data[i][pos]);
             }
         }
         return result;
     }
```

Suppose that a variable called grid has been declared as follows:

```
    int[][] grid = {{5, 7, 2, 9, 5, 8}, {1, 2, 4, 9, 5, 4},
                    {9, 2, 4, 6, 6, 3}, {9, 5, 8, 8, 4, 9},
                    {6, 5, 6, 2, 6, 9}, {8, 3, 4, 1, 8, 9}};
```

which means it will store the following 6-by-6 grid of values:

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

For each call below, indicate what value is returned. If the method call results in an exception being thrown, write "exception" instead.

Method Call	Contents of Set Returned
mystery(grid, 1)	_________________________________
mystery(grid, 2)	_________________________________
mystery(grid, 4)	_________________________________

4. 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.).

5. Binary Trees, 10 points. Write a method called oddPathSum that returns the number of nodes in a tree that have an odd path sum. The path sum of a node is the sum of all values from the overall root to that node. For example if the variable t refers to the following tree:

```
                              +----+
                              |  5 |
                              +----+
                           /          \
                   +----+                +----+
                   |  3 |                | 12 |
                   +----+                +----+
                  /      \                     \
             +----+       +----+               +----+
             |  1 |       |  2 |               |  7 |
             +----+       +----+               +----+
            /      \                          /      \
       +----+      +----+                +----+      +----+
       |  6 |      | 11 |                | 10 |      |  4 |
       +----+      +----+                +----+      +----+
```

Then t.oddPathSum() should return 4 indicating that 4 of these nodes have an odd path sum (the nodes storing 5, 12, 1, and 6)

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.

6. 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.

7. Comparable class, 20 points. Define a class called ClockTime that stores information about time of day using a standard clock. Each ClockTime object keeps track of hours, minutes, and a String to indicate "am" or "pm". It has the following public methods:

ClockTime(hours, minutes, amPm)	constructs a ClockTime with given hours, minutes and amPm setting
getHours()	returns the hours
getMinutes()	returns the minutes
getAmPm()	returns the am/pm setting
toString()	returns a String representation of the time

Assume that the values passed to your constructor are legal. In particular, hours will be between 1 and 12 inclusive, minutes will be between 0 and 59 inclusive, and the am/pm parameter will be either the String "am" or the String "pm". These values should be returned by the various "get" methods.

The toString method should return a String composed of the hours followed by a colon followed by the minutes (2 digits) followed by a space followed by the am/pm String. For example, given these declarations:

```
ClockTime time1 = new ClockTime(8, 31, "am");
ClockTime time2 = new ClockTime(12, 7, "pm");
```

time1.toString() should return "8:31 am" and time2.toString() should return "12:07 pm". You must exactly reproduce the format of these examples.

Your class should implement the Comparable<E> interface with earlier times considered "less." The earliest time is 12:00 am and the latest time is 11:59 pm. In between the time increases as it would in a standard clock. Keep in mind that 12:59 am is followed by 1:00 am, that 11:59 am is followed by 12:00 pm, and that 12:59 pm is followed by 1:00 pm.

Write your solution to ClockTime on the next page.

Write your solution to ClockTime below.

8. Binary Trees, 20 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 "morphing" in assignment 8) and you are not to replace any existing nodes. 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.

9. Linked Lists, 20 points. Write a method of the LinkedIntList class called removeAlternatePairs that removes alternate pairs from a list of numbers, constructing and returning another LinkedIntList that contains the values removed. For example, suppose that a variable called list stores the following values:

```
[1, 2, 3, 4, 5, 6, 7, 8]
 |  |  |  |  |  |  |  |
 +--+  +--+  +--+  +--+
 pair  pair  pair  pair
```

As indicated, this list has four pairs. If the following call is made:

```
LinkedIntList result = list.removeAlternatePairs();
```

then after the call, list and result would store the following values:

```
list:   [3, 4, 7, 8]
result: [1, 2, 5, 6]
```

Notice that the first pair has been moved to the new list, the second pair has not been moved, the third pair has been moved, and the fourth pair has not been moved. This pattern should be repeated with every other pair from the original list being moved to the new list.

This example purposely used sequential integers to make the rearrangement clear, but you should not expect that the list will store sequential integers. It might end with a pair that is to be moved or, as in the example above, it might end with a pair that is not supposed to be moved. It also might end with an extra value that is not part of a pair. Only complete pairs should be moved to the new list. For example, if the list had stored this sequence of values:

```
[5, 17, 8, 2, 9, 42, 37, 23, 19, -6, -5, 0, 4]
 |   |  |  |  |   |   |   |   |   |   |  |
 +---+  +--+  +---+   +---+   +---+   +--+
 pair   pair  pair    pair    pair    pair
```

then after the call list and result would store the following values:

```
list:   [8, 2, 37, 23, -5, 0, 4]
result: [5, 17, 9, 42, 19, -6]
```

Notice that the final pair of -5 and 0 is not moved but the final value of 4 is also not moved because it is not part of a complete pair. If no pairs are removed, the method should return an empty list.

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>
}
```

<continued on next page>

You are writing a method that will become part of the LinkedIntList class. You will need to call the zero-argument LinkedIntList constructor to construct the list to be returned and 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, you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc), and your solution must run in O(n) time where n is the number of nodes in the list. You also may not change any data fields of the nodes. You MUST solve this problem by rearranging the links of the list.

CSE143 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

Two-dimensional Arrays (7.5)

construct a rectangular array with 4 rows and 6 columns:

```
int[][] data = new int[4][6];
```

construct a jagged array with different numbers of columns in each row (3 rows that have 2, 3, and 5 columns):

```
int[][] data = new int[3][];
data[0] = new int[2];
data[1] = new int[3];
data[2] = new int[5];
```

Example values:

`data`	entire array
`data[2]`	row 2
`data[2][3]`	value in row 2 and column 3
`data.length`	number of rows
`data[2].length`	number of columns in row 2

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 thecharacter 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)