CSE 143, Winter 2012
Final Exam
Tuesday, March 13, 2012

Personal Information:

Name: _____

Section: _____ **TA:** _____

Student ID #: _____

- You have 110 minutes to complete this exam.
  You may receive a deduction if you keep working after the instructor calls for papers.
- This exam is open-book for the *Building Java Programs* textbook, but otherwise it is closed-book/notes.
  You may not use any paper resources or any computing devices including calculators.
- Code will be graded on proper behavior/output and not on style, unless otherwise indicated.
- Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

  The only abbreviations that *are* allowed for this exam are:
  - `S.o.p` for `System.out.print, and`
  - `S.o.pln` for `System.out.println.`
- You do not need to write `import` statements in your code.
- If you enter the room, you must turn in an exam before leaving the room.
- You must show your Student ID to a TA or instructor for your exam to be accepted.

Good luck!

Score summary: (for grader only)

Problem	Description	Earned	Max
1	Inheritance/Polymorphism Mystery		15
2	Inheritance/Comparable Programming		15
3	Collection Mystery		8
4	Linked List Programming		15
5	Searching and Sorting		7
6	Binary Search Trees		10
7	Binary Tree Programming		15
8	Binary Tree Programming		15
X	Extra Credit		+1
TOTAL	**Total Points**		**100**

1. Inheritance and Polymorphism

Consider the following classes:

```
public class Eddie extends Kurt {
    public void b() {
        a();
        System.out.println("Eddie 2");
    }

    public void c() {
        System.out.println("Eddie 3");
    }
}

public class Kurt {
    public void a() {
        System.out.println("Kurt 1");
        c();
    }

    public void c() {
        System.out.println("Kurt 3");
    }
}

public class Chris extends Jerry {
    public void b() {
        System.out.println("Chris 2");
        super.c();
    }

    public void c() {
        System.out.println("Chris 3");
    }
}

public class Jerry extends Kurt {
    public void c() {
        System.out.println("Jerry 3");
        super.c();
    }
}
```

and that the following variables are defined:

```
Kurt   var1 = new Jerry();
Chris  var2 = new Chris();
Kurt   var3 = new Eddie();
Kurt   var4 = new Chris();
Object var5 = new Jerry();
```

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 **"error"** to indicate this.

Statement	**Output**
`var1.a();`	_____
`var1.c();`	_____
`var2.a();`	_____
`var2.b();`	_____
`var3.a();`	_____
`var3.b();`	_____
`var4.a();`	_____
`var5.a();`	_____
`((Chris) var5).a();`	_____
`((Jerry) var1).a();`	_____
`((Jerry) var4).a();`	_____
`((Chris) var3).b();`	_____
`((Eddie) var3).b();`	_____
`((Jerry) var4).c();`	_____
`((Kurt) var5).c();`	_____

2. Inheritance and `Comparable` Programming

You have been asked to extend an existing class `Dice` representing a set of 6-sided dice that can be rolled by a player.

Member	Description
`private int[] diceValues`	private data of each `Dice` object
`public` **`Dice`**`(int count)`	constructs a dice roller to roll the given # of dice; all dice initially have the value of 6
`public int` **`count`**`()`	returns # of dice as passed to constructor
`public int` **`getValue`**`(int index)`	returns the die value (1-6) at the given 0-based index
`public void` **`roll`**`(int index)`	rolls the given die to give it a new random value (1-6)
`public int` **`total`**`()`	returns sum of all current dice values in this dice roller
`public String` **`toString`**`()`	returns string of dice values, e.g. `"[4, 1, 6, 5]"`

Define a new class called `RiggedDice` that extends `Dice` through inheritance. Your class represents dice that let a player "cheat" by ensuring that they will always roll a value that is greater than or equal to a given minimum value.

You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	Description
`public` **`RiggedDice`**`(int count, int min)`	constructs a rigged dice roller to roll the given # of dice, using the given minimum value for all future rolls; all dice initially have the value 6 (if the min value is not between 1-6, throw an `IllegalArgumentException`)
`public int` **`getMin`**`()`	returns minimum roll value as passed to constructor

A `RiggedDice` should behave like a `Dice` object except for the following differences.
You may need to override or replace existing behavior in order to implement these changes.

- Every time a die is **rolled**, you must ensure that the value rolled is greater than or equal to the minimum value passed to your constructor. Do this by re-rolling the die if the value is too small, as many times as necessary.
- The rigged dice should return a **total** that lies and claims to be 1 higher than the actual total. For example, if the sum of the values on the dice add up to 13, your rigged dice object's `total` returned should be 14.
- When a rigged dice object is printed or used as a string, it should display that the dice are rigged, then the dice values, then the minimum dice value, such as, `"rigged [4, 3, 6, 5] min 2"`

Also **make `RiggedDice` objects comparable to each other using the `Comparable` interface**. `RiggedDice` are compared by total dice value in ascending order, breaking ties by minimum roll value in ascending order. In other words, a `RiggedDice` with a lower total dice value is considered to be "less than" one with a higher total. If two objects have the same total, the one with a lower min value passed to its constructor is "less than" one with a higher course count. If the two objects have the same total and the same min, they are considered to be "equal."

The majority of your grade comes from implementing the correct behavior. Part of your grade also comes from appropriately utilizing the behavior you have inherited from the superclass and not re-implementing behavior that already works properly in the superclass. You may assume that the superclass already checks all arguments passed to its constructor and methods to make sure that they are valid.

(Write your answer on the next page.)

---

2. Inheritance and `Comparable` Programming (writing space)

3. Collection Mystery

Write the output that is printed when the given method below is passed each of the following maps as its parameter. Recall that maps print in a *key=value* format. Your answer should display the right keys and values in the right order. The maps shown below are displayed with one key/value pair per line to save space, but no line breaks actually appear in the keys or values stored in any of the maps.

```
public static void mystery(Map<String, String> map) {
    Map<String, String> result = new TreeMap<String, String>();
    for (String key : map.keySet()) {
        if (key.compareTo(map.get(key)) < 0) {
            result.put(key, map.get(key));
        } else {
            result.put(map.get(key), key);
        }
    }
    System.out.println(result);
}
```

Map	Output
a) {two=deux, five=cinq, one=un, three=trois, four=quatre}	
b) {skate=board, drive=car, program=computer, play=computer}	
c) {siskel=ebert, girl=boy, heads=tails, ready=begin, first=last, begin=end}	
d) {cotton=shirt, tree=violin, seed=tree, light=tree, rain=cotton}	

4. Linked List Programming

Write a method `sortPairs` that could be added to the `LinkedIntList` class from lecture and section. The method compares each pair of consecutive elements in the list (the pair at indexes 0 and 1, and then the pair at indexes 2 and 3, and then the pair at indexes 4 and 5, etc.), and swaps the order of the two nodes if necessary so that the smaller data value of the two comes first. If the list is of odd size, the last element is left unmodified.

Suppose a `LinkedIntList` variable named `list` stores the following values. Notice that for some pairs, the first element in the pair is larger than the second in the pair; those elements are underlined in the example.

```
index   0   1   2   3   4   5  6  7    8    9  10  11  12
      [20, 10, 15, 11, 32, 28, 5, 7, 314, 100, 17, 29, 12]
```

The call of `list.sortPairs();` would change the list to store the elements in the following order. Notice that whenever there is a pair where the larger value came first (such as 15 and 11 at indexes 2 and 3), they have swapped places so that the smaller of the two now comes first.

```
[10, 20, 11, 15, 28, 32, 5, 7, 100, 314, 17, 29, 12]
```

If the list is empty or contains only a single element, it is unchanged by a call to your method.
For full credit, obey the following restrictions in your solution:

- The method should run in no worse than O(*N*) time, where *N* is the length of the list.
  For full credit, you must solve the problem by making a single pass over the list.
- Do not call any methods of the linked list class to solve this problem.
  Note that the list does not have a `size` field, and you are not supposed to call its `size` method.
- Do not use auxiliary data structures such as arrays, `ArrayList`, `Queue`, `String`, etc.
- Do not modify the `data` field of any nodes; you must solve the problem by changing the links between nodes.
- You may not create new `ListNode` objects, though you may create as many `ListNode` variables as you like.

You are using the `LinkedIntList` and `ListNode` class as defined in lecture and section. (See cheat sheet.)

---

4. Linked List Programming (writing space)

5. Searching and Sorting

(a) Suppose we are performing a **binary search** on a sorted array called `numbers` initialized as follows:

```
// index          0    1   2   3   4   5   6   7   8    9  10  11  12  13
int[] numbers = {-23, -5,  9,  14, 15, 18, 23, 24, 25, 29, 34, 62, 85, 87};

int index = binarySearch(numbers, 19);
```

Write the indexes of the elements that would be examined by the binary search (the `mid` values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: _____
- Value Returned: _____

(b) Consider the following array:

```
int[] numbers =   {29, 17, 3, 94, 46, 8, -4, 12};
```

After **one pass** of selection sort, what would be the contents of the array? **Circle** the correct answer.

choice #1: `{-4, 29, 17, 3, 94, 46, 8, 12}`

choice #2: `{29, 17, 3, 94, 46, 8, 12}`

choice #3: `{-4, 29, 17, 3, 94, 46, 8, -4, 12}`

choice #4: `{-4, 17, 3, 94, 46, 8, 29, 12}`

choice #5: `{3, 17, 29, 94, -4, 8, 46, 12}`

(c) Exactly **one** of the following statements about sorting and Big-Oh is true. **Circle** the true statement.

#1: Selection sort could sort the array `numbers` from part **(b)** in O(N) time.

#2: Merge sort achieves an O($N \log_2 N$) runtime by dividing the array in half at each step and then recursively sorting and merging the halves back together.

#3: Merge sort runs faster than selection sort because it is recursive, and recursion is faster than `for` loops.

#4: Selection sort runs in O(N) time if the array is already sorted to begin with, or O(N^2) if it is not.

#5: Sorting algorithms that rely on comparing elements can only be used with type `int`, because values from other types of data cannot be compared to each other.

6. Binary Search Trees

(a) Write the binary search tree that would result if these elements were added to an empty tree in this order:

- Dodo, Eaglet, Rabbit, Cat, Alice, Jabberwock, Hatter, Tweedledee, Queen, Bill

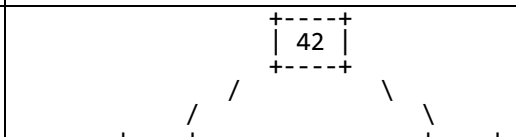
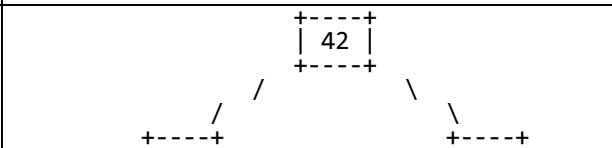
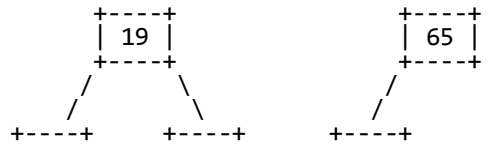
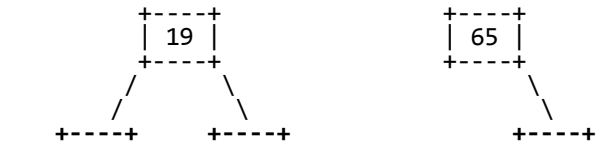
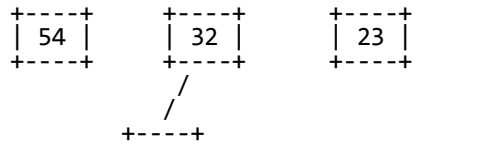
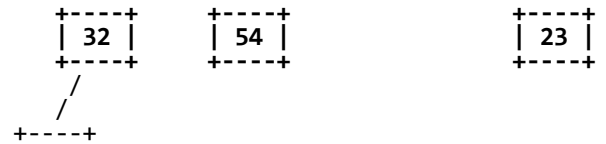
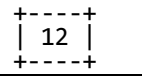
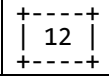
(b) Write the elements of your above tree in the order they would be visited by each kind of traversal:

- Pre-order: _____
- In-order: _____
- Post-order: _____

7. Binary Tree Programming

Write a method **`swapChildrenAtLevel`** to be added to the `IntTree` class from class (see cheat sheet). Your method should accept an integer `n` as a parameter and swap the left and right children of all nodes at level `n`. In other words, after your method is run, any node at level `n + 1` that used to be its parent's left child should now be its parent's right child and vice versa. For this problem, the overall root of a tree is defined to be at level 1, its children are at level 2, etc. The table below shows the result of calling this method on an `IntTree` variable `tree`.

```
IntTree tree = new IntTree();
...
tree.swapChildrenAtLevel(2);
```

Level	Before Call	After Call
1		
2		
3		
4		

If `n` is 0 or less than 0, your method should throw an `IllegalArgumentException`. If the tree is empty or does not have any nodes at the given level or deeper, it should not be affected by a call to your method.

For efficiency, your method should not traverse any parts of the tree that it does not need to traverse. Specifically, you should not access any nodes lower than level `n + 1`, because there is nothing there that would be changed.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the tree class nor create any data structures such as arrays, lists, etc. You should not construct any new node objects or change the `data` of any nodes. For full credit, your solution must be recursive.

(Write your answer on the next page.)

---

7. Binary Tree Programming (writing space)

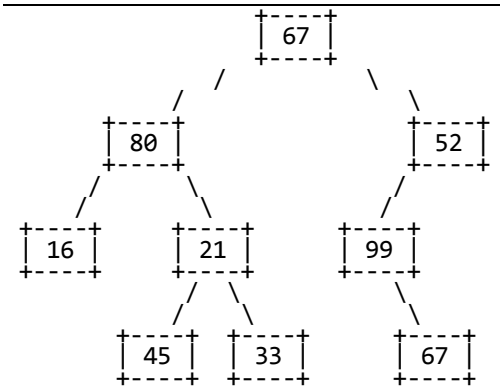
8. Binary Tree Programming

Write a method `makePerfect` that could be added to the `IntTree` class from lecture and section. The method should add nodes until the binary tree is a "perfect" tree. A perfect binary tree is one where all leaves are at the same level. Another way of thinking of it is that you are adding dummy nodes to the tree until every path from the root to a leaf is the same length. A perfect tree's shape is exactly triangular and every branch node has exactly two children. Each node you add to the tree should store the value 0.

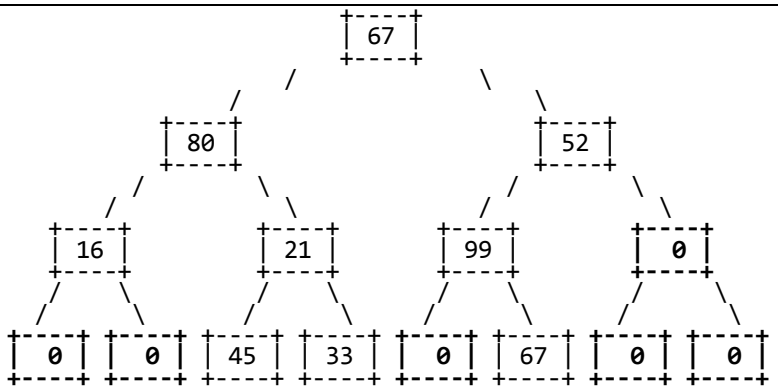
The following table shows the results of a call of your method on a particular tree:

```
IntTree tree = new IntTree();
...
tree.makePerfect();
```

`tree` before call



`tree` after call



For this problem, you may assume the existence of the following helper method, which you are allowed to call at most once during an entire call to your method:

```
public int height() { ... }   // returns height of tree from root to lowest leaf
```

For example, calling `height` on the tree above would return `4` because it is 4 levels tall.

You may define private helper methods, but aside from `height` you may not call other methods of the class nor create data structures. For full credit, your solution must be recursive and properly utilize the *x = change(x)* pattern.

(Write your answer on the next page.)

8. Binary Tree Programming (writing space)

X. Extra Credit

If your TA's name were an acronym (an abbreviation formed from taking the initial letters of each word), what would the acronym be? For example, the name “JOE” might stand for the acronym, “Jealous Of Everyone”.

(This is just for fun. Any answer that writes any set of words that start with the letters of your TA's name or makes a close attempt to do so will get the point.)

^_^ CSE 143 FINAL EXAM CHEAT SHEET ^_^

Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<String>();
Set<String> set = new HashSet<String>();
Map<String, Integer> map = new TreeMap<String, Integer>();
```

Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

`clear()`	removes all elements of the collection
`equals(`**`collection`**`)`	returns `true` if the given other collection contains the same elements
`isEmpty()`	returns `true` if the collection has no elements
`size()`	returns the number of elements in the collection
`toArray()`	returns an array of the elements in this collection
`toString()`	returns a string representation such as `"[10, -2, 43]"`

Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**`value`**`)`	adds value to collection (appends at end of list)
`contains(`**`value`**`)`	returns `true` if the given value is found somewhere in this collection
`iterator()`	returns an Iterator object to traverse the collection's elements
`remove(`**`value`**`)`	finds and removes the given value from this collection

List<E> Methods (10.1)

`add(`**`index, value`**`)`	inserts given value at given index, shifting subsequent values right
`indexOf(`**`value`**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	returns the value at given index
`lastIndexOf(`**`value`**`)`	returns last index where given value is found in list (-1 if not found)
`remove(`**`index`**`)`	removes/returns value at given index, shifting subsequent values left
`set(`**`index, value`**`)`	replaces value at given index with given value
`subList(`**`from, to`**`)`	returns sub-portion at indexes **from** (inclusive) and **to** (exclusive)

Stack<E> Methods

`peek()`	returns the top value from the stack without removing it
`pop()`	removes the top value from the stack and returns it; `peek`/`pop` throw an `EmptyStackException` if the stack is empty
`push(`**`value`**`)`	places the given value on top of the stack

Queue<E> Methods

`add(`**`value`**`)`	places the given value at the back of the queue
`peek()`	returns the front value from the queue without removing it; returns `null` if the queue is empty
`remove()`	removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty

Map<K, V> Methods (11.3)

`clear()`	remove all keys and values from the map
`containsKey(`**`key`**`)`	`true` if the map contains a mapping for the given key
`get(`**`key`**`)`	the value mapped to the given key (`null` if none)
`keySet()`	returns a `Set` of all keys in the map
`put(`**`key, value`**`)`	adds a mapping from the given key to the given value
`remove(`**`key`**`)`	removes any existing mapping for the given key
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`
`values()`	returns a `Collection` of all values in the map

String Methods (3.3, 4.4)

`charAt(`**`i`**`)`	the character in this String at a given index
`compareTo(`**`str`**`)`	compare strings by ABC order; returns <0 (less), 0 (=), or >0 (greater)
`contains(`**`str`**`)`	`true` if this String contains the other's characters inside it
`endsWith(`**`str`**`)`	`true` if this String ends with the other's characters
`equals(`**`str`**`)`	`true` if this String is the same as *str*
`equalsIgnoreCase(`**`str`**`)`	`true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**`str`**`)`	first index in this String where given String begins (-1 if not found)
`lastIndexOf(`**`str`**`)`	last index in this String where given String begins (-1 if not found)
`length()`	number of characters in this String
`startsWith(`**`str`**`)`	`true` if this String begins with the other's characters
`substring(`**`i, j`**`)`	characters in this String from index *i* (inclusive) to *j* (exclusive)
`toLowerCase(), toUpperCase()`	a new String with all lowercase or uppercase letters

Random Methods (5.1)

`nextBoolean()`	random `true`/`false` result
`nextDouble()`	random real number between 0.0 and 1.0
`nextInt()`	random integer
`nextInt(`**`max`**`)`	random integer from 0 to *max* - 1, inclusive

```
public class ListNode {
    public int data;
    public ListNode next;
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
public class LinkedIntList {
    private ListNode front;
    methods
}

public class IntTreeNode {
    public int data;
    public IntTreeNode left;
    public IntTreeNode right;
    public IntTreeNode(int data) { ... }
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {...}
}
public class IntTree {
    private IntTreeNode overallRoot;
    methods
}
```