CSE 143, Summer 2012
Final Exam – Part 2
Friday, August 17, 2012

Personal Information:

Name: _____

Section: _____ **TA:** _____

Student ID #: _____

- You have 60 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	Searching and Sorting		8
2	Collection Mystery		10
3	Linked List Programming		15
4	Binary Tree Programming		15
X	Extra Credit		+0
TOTAL	**Total Points for Part 2**		**48**
TOTAL	**Total Points**		**100**

1. 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 = {-42, -3,  1,  14, 22, 29, 30, 31, 35, 37, 38, 62, 85, 87};

int index = binarySearch(numbers, 36);
```

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 =   {14, 17, 3, 23, 46, 8, -5, 12};
```

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

choice #1: `{14, 17, 3, 23, 46, 8, 12}`

choice #2: `{-5, 17, 3, 23, 46, 8, 14, 12}`

choice #3: `{-5, 14, 17, 3, 23, 46, 8, 12}`

choice #4: `{-5, 14, 17, 3, 23, 46, 8, -5, 12}`

choice #5: `{3, 17, 14, 23, -5, 8, 46, 12}`

(c) Trace the complete execution of the **merge sort** algorithm when called on the array below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-arrays that are created by the algorithm and show the merging of sub-arrays into larger sorted arrays.

```
int[] numbers = {33, 14, 3, 95, 47, 9, -42, 13};
mergeSort(numbers);
```

2. Collection Mystery

Write the output that is printed when the given method below is passed each of the following maps as its parameter. Your answer should display the right 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. You can assume that a for-each loop over the map will produce the elements in the order they are shown in the question.

```
public static void mystery (Map<String, String> m) {
      Set<String> s = new TreeSet<String>();

      for (String key : m.keySet()) {
            if (!m.get(key).equals(key)) {
                  s.add(m.get(key));
            } else {
                  s.remove(m.get(key));
            }
      }
      System.out.println(s);
}
```

Map	Output
a) {sheep=wool, house=brick, cast=plaster, wool=wool}	
b) {munchkin=blue, winkie=yellow, corn=yellow, grass=green, emerald=green}	
c) {pumpkin=peach, corn=apple, apple=apple, pie=fruit, peach=peach}	
d) {lab=IPL, lion=cat, terrier=dog, cat=cat, platypus=animal, nyan=cat}	

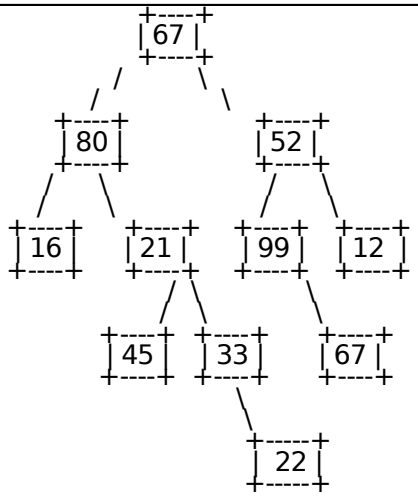
3. Binary Tree Programming

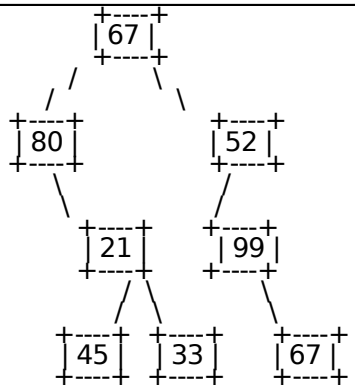
Write a method `evenLevels` that could be added to the `IntTree` class from lecture and section. The method should make sure that all branches end on an even level. If a leaf node is on an odd level it should be removed from the tree. We will define the root as being on level 1.

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

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

level	**tree** before call	**tree** after call
1	67	67
2	80, 52	80, 52
3	16, 21, 99, 12	21, 99
4	45, 33, 67	45, 33, 67
5	22	





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 and properly utilize the *x = change(x)* pattern.

(Write your answer on the next page.)

---

3. Binary Tree Programming (writing space)

4. Linked List Programming

Write a method `removeRange` that could be added to the `LinkedIntList` class from lecture and section. The method takes two integer parameters, min and max, and removes all elements from the list whose values are between min and max, inclusive. It should return the number of elements removed.

Suppose a `LinkedIntList` variable named `list` stores the following values:

```
[4, 2, 1, 10, 15, 8, 7, 4, 20, 36, -3, 40, 5]
```

The call of `list.removeRange(4, 20)` would return `8` and change the list to store the following elements:

```
[2, 1, 36, -3, 40]
```

If the value of the max parameter is less than that of the min parameter you should throw an `IllegalArgumentException`. If the list is empty or does not contain any elements between or equal to min and max, it should return `0` and leave the list unchanged after a call to `removeRange`.

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 should make only one 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. 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)

X. For fun

Draw a picture of your TA.

(This is just for fun. You will not receive any points for this.)

^_^ 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
}
```