Computer Science 201
Fall 2012
Midterm #2

Your Name: ____________________________

Your NetID: ____________________________

Sign your name on the line below to confirm that you have completed this test in accordance with the Duke Community Standard.

____

This is a 75-minute, 75-point test: each point should take about one minute. Perfect Java syntax is not required: get your point across without worrying about perfect semicolon, curly-brace, and indentation hygiene. It should still look like Java: pseudocode and English aren't good enough. *Write carefully: if we can't read it, we won't grade it.*

Question	Possible	Score
0	6	
1	8	
2	10	
3	4	
4	4	
5	16	
6	4	
7	9	
8	13	
9	1	
Total	75	

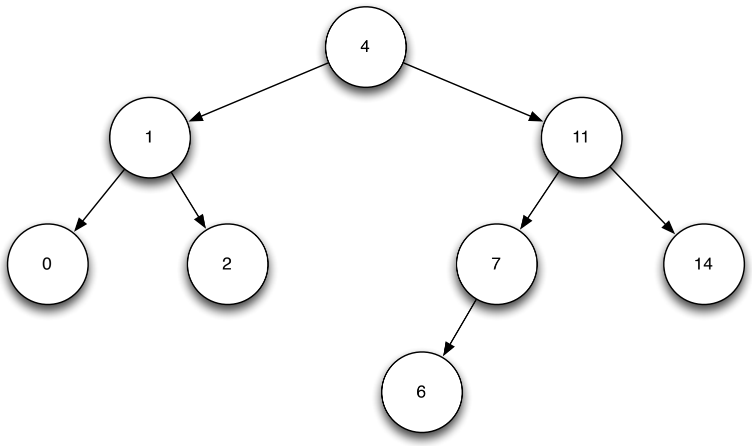
0 Binary Search Trees (6 points total)

0.1 Building Trees (2 Points)

Consider adding the values `3 8 1 7 0 6 5 2` to an (initially empty) *binary search tree*. Assume that you use the basic insertion algorithm that adds the value in the correct location, but makes no effort to balance the tree. Draw the resulting tree.

0.2 Traversals (4 Points)

Given the following tree:



Write out the pre-order traversal.

Write out the in-order traversal.

1 Trees (8 points total)

1.1 Sum a Binary Tree (4 Points)

The TreeNode class is on the cheat sheet.

Implement the method `sumTree`, which computes the sum of the elements in a given tree.

```
int sumTree(TreeNode root) {









}
```

1.2 Binary Search Trees (4 Points)

You would like to insert the integers 1 through N into a binary search tree. Complete the following code such that your resulting tree height is $O(\log N)$ where N is the number of nodes in the tree.

The `BinarySearchTree` class is not provided. Assume that it implements a binary search tree with the method `.add(value)` that will add `value` in the correct place.

```
public void addNodes(BinarySearchTree t, int num){
    ArrayList<Integer> toAdd = new ArrayList<Integer>();

        for(int i = 0; i < num; i++){
            toAdd.add(i);
        }









}
```

2 Linked Lists (10 points total)

Given a singly linked list with only a head pointer (see below), complete the function `doubleEveryOther` that doubles every other element in the list, starting at the second element in the list. For example, if your linked list was $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, `doubleEveryOther` would change the list to $1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 4$.

```
public class ALinkedList {
    // Inner class that looks like ListNode, but stores Strings.
    public class Node {
        public String myValue;
        public Node myNext;
        public Node(String value, Node next) {
            myValue = value;
            myNext = next;
        }
    }
    public Node myHead;

    // You may assume that the list has at least one Node.
    public void doubleEveryOther(){
```

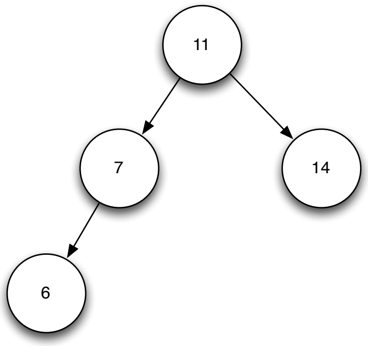
3 Stacks & Queues (4 points total)

The TreeNode class is on the cheat sheet.

Consider the following piece of code, which prints every node in a tree.

```
// You may assume that node is non-null.
void search(TreeNode node) {
  // Assume there's a Queue class that stores TreeNodes.
  // It has .push(), .pop(), and .size(). (Note that this is slightly
  // different than Java's built-in Queue.)
  Queue q = new Queue();
  q.push(node);

  while (q.size() > 0) {
    TreeNode n = q.pop();
    if (n.myLeft != null) {
      q.push(n.myLeft);
    }
    if (n.myRight != null) {
      q.push(n.myRight);
    }
    System.out.print(n.myValue + " ");
  }
}
```

3.1 (2 Points)



Consider running this code on the tree above with `search(root)`, where `root` is the root node of the tree. Write down what would be printed.

Question continues on the next page.

3.2 (2 Points)

Now, consider replacing the line

```
Queue q = new Queue();
```

with

```
Stack q = new Stack();
// Assume that the Stack class exists, and implements a stack, with
// push, pop, and size. Note the re-use of the name 'q' so that the
// code will run with no other changes.
```

Write out what would be printed if this version of the code was run by calling `search(root)`.

4 Fundamentals (4 points total)

4.1 Trees vs. Linked Lists (2 Points)

What is the difference between a tree node and a linked-list node? (Hint: Think about the node inner classes when implementing a tree and a linked list)

4.2 Binary Search Trees vs. Heaps (2 Points)

What is the difference between a binary search tree and a min-heap?

5 Recurrence Relations (16 points total)

For each of the following methods, write down its recurrence relation *and the corresponding Big-O running time. Don't forget the base case!*

5.1 maximumOfList (4 points)

`maximumOfList` finds the largest value in a list. As our `ListNodes` do not store a length, `computeLength(ListNode node)` runs in $O(n)$. (Note that the code for `computeLength` is not provided, as you don't need it.) You may assume that the passed-in list has length at least 1.

```
int maximumOfList(ListNode node) {
  if (computeLength(node) == 1) {
    return node.myValue;
  }
  return Math.max(node.myValue, maximumOfList(node.myNext));
}
```

5.2 hasPartialLeaf (4 points)

You may assume that the tree is height-balanced.

```
// A "partial leaf" is a node with exactly one child tree.
boolean hasPartialLeaf(TreeNode node) {
  if (node == null) {
    return false;
  }
  if (node.myLeft == null && node.myRight != null) {
    return true;
  }
  if (node.myLeft != null && node.myRight == null) {
    return true;
  }
  return (hasPartialLeaf(node.myLeft) || hasPartialLeaf(node.myRight));
}
```

Question continues on the next page.

5.3 moveMinToRoot (4 points)

You may assume the tree is height-balanced.

```
// (Recursively) move the minimum value in a BST to the root.
// The resulting tree is not necessarily a BST.
void moveMinToRoot(TreeNode node) {
  if (node == null) {
    return;
  }
  if (node.myLeft == null && node.myRight == null) {
    return;
  }
  moveMinToRoot(node.myLeft);
  int temp = node.myValue;
  node.myValue = node.myLeft.myValue;
  node.myLeft.myValue = temp;
}
```

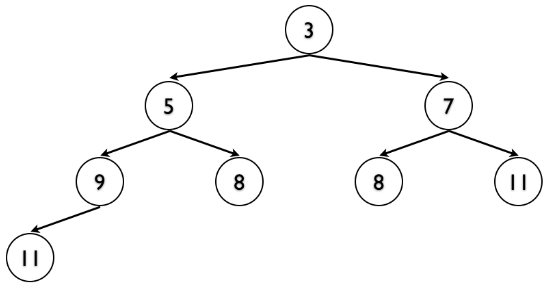
5.4 leafHeights (4 points)

You may assume the tree is balanced.

```
// Compute the height of every leaf in the tree.
ArrayList<Integer> leafHeights(TreeNode node) {
  ArrayList<Integer> result = new ArrayList<Integer>();
  if (node == null) {
    return result;
  }
  if (node.myLeft == null && node.myRight == null) {
      result.add(1);
      return result;
  }
  ArrayList<Integer> lefts = leafHeights(node.myLeft);
  ArrayList<Integer> rights = leafHeights(node.myRight);
  for (int i : lefts) {
    result.add(i + 1);
  }
  for (int i : rights) {
    result.add(i + 1);
  }
  return result;
}
```

6 Heaps (4 Points)

The image below is a min-heap. Draw the resulting min-heap after adding the value 4.



7 Priority Queues (9 points total)

You are registering for spring classes and there is a very popular course. The professor is a little crazy and has decided to admit people into the class based on the month they were born. People in January will be admitted first, then February, then March, and so on. However, the professor is ignoring the day within the month. She has decided that whoever registered first with a January birthday will be the first person admitted to the class.

That is, if the following students, with their birth month, registered in the following order: `Joe (Feb.), Sam (Mar.), Jill (Jan.), John (Mar.), Jane (Feb.)` then they will be admitted into the class in the following order: `Jill, Joe, Jane, Sam, John`.

Complete the following code to determine the students who will be admitted to the class.

```
public class AdmitStudents {

  private class Student implements Comparable<Student>{
    private String myName;
    private int myBirthMonth;
    public Student(String name, int birthMonth){
      myName = name;
      myBirthMonth = birthMonth;
    }  // end of Student constructor.
```

Question continues on the next page.

7.1 2 Points

Complete `compareTo` for your Student inner class.

```
  public int compareTo(Student other) {







    }
} // closes the Student inner class
```

7.2 3 points

Add any instance variables that `AdmitStudents` needs. Then, complete the `add` method (it may be helpful to read the description of `getStudents`, below, before you do this).

```
// Instance variables go here, before add().


void add(Student s) {



}
```

7.3 4 Points

Complete `getStudents` which gets the next k students to be admitted to the class and returns them in a String array. You may assume that at least k students have been added using your `.add` method. `getStudents` should run in $O(k \log n)$, where n is the number of students that have been added.

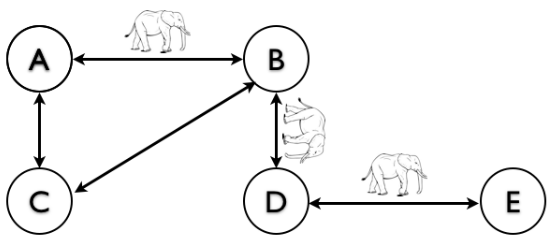
```
public String[] getStudents(int k) {









}
```

8 Recursive Backtracking (13 points)

You're going on vacation! Due to a gasoline shortage, you have chosen to travel by elephant. Because elephants are *big*, elephant housing is expensive, so you want to visit as few cities as possible. You'll be writing code to compute a route from where you are to where you want to be while visiting as few cities as possible. *Conveniently, using recursive backtracking to find a path will automatically find you the shortest path, so you don't need to worry about "shortest" for this problem. We'll talk about this more in the weeks to come.*

You have a city map which gives each city a name, and an array of neighboring cities, as seen below. Your job is to write the method `travelRoute` (see the next page) that returns the route you should take. Your code should fill in the `myPath` variable with the path you should take, city-by-city, in order. If no such path exists, `myPath` should end up empty.

Consider the city map below. It has five cities (A–E). If you called `travelRoute` with `City` objects representing A and E, `myPath` would contain `City` objects representing A, B, D, and E, in that order.



```
public class ElephantVacation {
  // Your code will fill in the values in this list.
  public List<City> myPath;

  public ElephantVacation() {
    myPath = new LinkedList<City>();
  }

  // Inner class that represents a city.
  public class City implements Comparable<City> {
    public String myName;
    public City[] myNeighbors;

    public City(String name) {
      myName = name;
    }

    public void setNeighbors(City[] neighbors) {
      myNeighbors = neighbors;
    }

    // You may assume that City has correct
    // implementations of equals(), compareTo(),
    // and hashCode().

  }  // End of City.
```

Question continues on the next page.

```
public void travelRoute(City start, City finish) {
  // TODO: Fill this in!
  // Start with your base case!
```

One more question!

9 Catch your breath. (1 point)

Draw a picture that will entertain your grader!

The next three pages are the cheat sheet.

The complete cheat sheet from Midterm #1 is on the last page.

Trees & Lists
The following two classes are used in the test.

```
class TreeNode {
  public int myValue;
  public TreeNode myLeft;
  public TreeNode myRight;
  public TreeNode(int value,
                  TreeNode left,
                  TreeNode right) {
    myValue = value;
    myLeft = left;
    myRight = right;
  }
}
```

```
class ListNode {
  public int myValue;
  public ListNode myNext;
  public ListNode(int value,
                  ListNode next) {
    myValue = value;
    myNext = next;
  }
}
```

Recurrence Relations
In the relations below, $T(0) = T(1) = 1$.

$$T(n) = T\left(\frac{n}{2}\right) + O(1) \qquad O(\log n)$$
$$T(n) = T(n-1) + O(1) \qquad O(n)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) \qquad O(n)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \qquad O(n \log n)$$
$$T(n) = T(n-1) + O(n) \qquad O(n^2)$$

Random Numbers
To generate a random integer between 0 (inclusive) and n (exclusive), do this:

```
Random rand = new Random();
// r will be at least 0, and at most n-1.
int r = rand.nextInt(n);
```

Priority Queues

```
// Java's built-in priority queue is a min-heap. You can use any
// Comparable type, not just Integer.
PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

pq.add(7);                   // add 7 to pq
pq.add(3);                   // add 3 to pq
int smallest = pq.peek();    // Return the smallest element (in this case, 3)
                             // peek DOES NOT modify the queue

smallest = pq.poll();        // _Remove_ and return the smallest element.
```

Tree Traversals

```
void preOrderTraversal(TreeNode node) {
  if (node == null) {
    return;
  }
  System.out.println(node.myValue);  // Or some other operation.
  preOrderTraversal(node.myLeft);
  preOrderTraversal(node.right);
}

void inOrderTraversal(TreeNode node) {
  if (node == null) {
    return;
  }

  inOrderTraversal(node.myLeft);
  System.out.println(node.myValue);  // Or some other operation.
  inOrderTraversal(node.right);
}

void postOrderTraversal(TreeNode node) {
  if (node == null) {
    return;
  }

  postOrderTraversal(node.myLeft);
  postOrderTraversal(node.right);
  System.out.println(node.myValue);  // Or some other operation.
}
```

`String`

- `.length()` Get the length of the `String`. $O(1)$.
- `.charAt(i)` Get the `char` at index `i`. $O(1)$.
- `.substring(i, j)` Get the substring between indices `i` and `j`. Index `i` is *inclusive*, and index `j` is *exclusive*. $O(1)$. For example:

  ```
  String x = "abcdefg";
  String y = x.substring(2, 4);
  // y now has the value "cd"
  ```

`ArrayList<T>  // Where T is a type, like String or Integer`

- `.add(i, X)` Add element `X` to the list at index `i`. If no `i` is provided, add an element to the end of the list. Adding to the end runs in $O(1)$.
- `.get(i)` Get the element at position `i`. Runs in $O(1)$.
- `.set(i, X)` Set the element at position `i` to the value `X`. $O(1)$.
- `.size()` Get the number of elements. $O(1)$.

`HashSet<T>  // Where T is a type, like String or Integer`

- `.size()` Compute the size. $O(1)$.
- `.add(X)` Add the value `X` to the set. If it's already in the set, do nothing. $O(1)$.
- `.contains(X)` Return a `boolean` indicating if `X` is in the set. $O(1)$.
- `.remove(X)` Remove `X` from the set. If `X` was not in the set, do nothing. $O(1)$.

`HashMap<K, V>  // Where K and V are the key and value types, respectively.`

- `.size()` Compute the size. $O(1)$.
- `.containsKey(X)` Determines if the map contains a value for the key `X`. To get that value, use `.get()`. $O(1)$.
- `.get(X)` Gets the value for the key `X`. If `X` is not in the map, return `null`. $O(1)$.
- `.put(k, v)` Map the key `k` to the value `v`. If there was already a value for `k`, replace it. $O(1)$.
- `.keySet()` Return a `Set` containing the keys in the map. Useful for iterating over. $O(1)$.

To iterate over a `HashSet<T>`, use

```
for (T v : nameOfSet) {
  // v is the current element of the set.
}
```

This can be combined with `HashMap`'s `.keySet()` to iterate over a `HashMap`.