Computer Laboratory 13

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- **This is the last lab of the semester.**
- This assignment is due **TUESDAY MAY 06 th at 12:00pm (noon)** and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

In this lab you will build a simple unstructured binary tree to store a real-world tree-organized concept, a "call tree". This will give you practice with binary trees, binary tree traversals, and modifying binary trees in structured ways. This practice should hopefully help you understand the binary tree fundamentals we will be building upon with binary search trees.

3 Files

This lab will involve the following **provided** files.

- `CallTreeNodeTests.java`
- `CallTreeTests.java`

This lab also involves the following files **you will create**:

- `CallTreeNode.java`
- `CallTree.java`

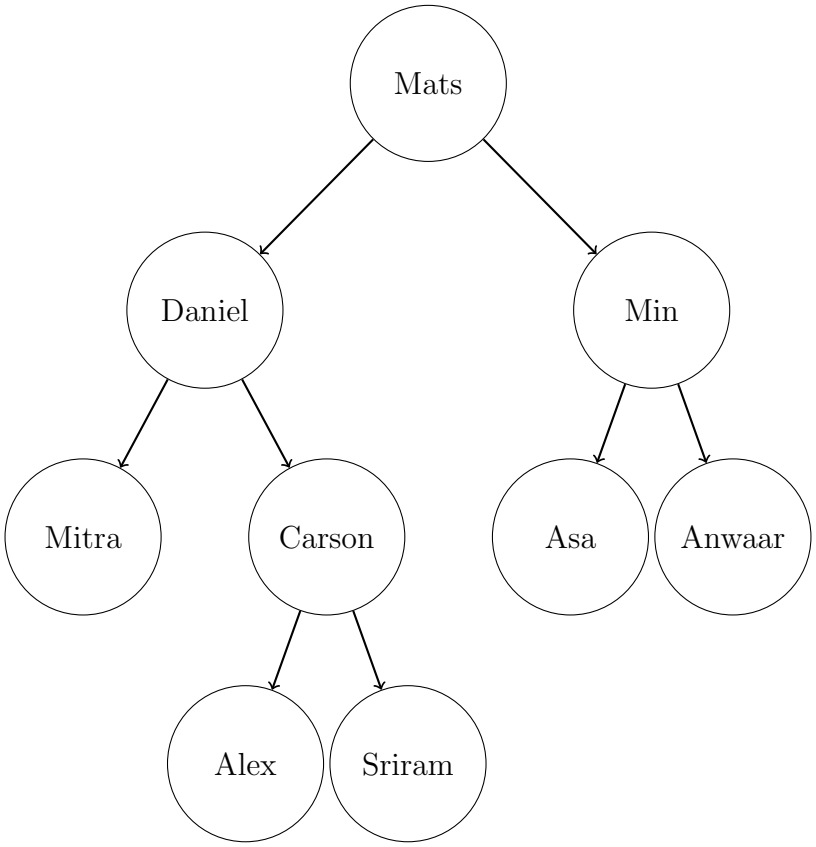
4 Theory: Call Trees

A call tree is a real-world thing used to spread information in emergency situations. It happens to be called a tree for the same reason that we call our data structure concept a tree (both ideas come from the mathematical concept of a "Tree"). In a call tree each person in a company or community organization would have a short list of people to call in case of emergency. Call trees rely on a "pass it on" principal – when you hear about the emergency information, your job is then to pass it on to the people on your list. The people you call would then pass it on to more people, who pass it on to still more people until everyone has been notified.

At each level the number of people getting called grows exponentially, as more and more people get roped into this "pass it on" principal of information sharing. This allows an emergency messages to be spread incredibly quickly, needing only $O(log(n))$ time to contact an entire group. (obviously n calls will still be made, but many will happen at the same time) This compares favorably to a centralized solution in which a few people must call everyone in the group, taking $O(n)$ time to make all the calls. Call trees can also be modified to have redundancy. This will ensure extra robustness, because the system will never rely on any one person to make sure messages spread fully. The only major downside of this process is that it really does become *everyone's* job to make sure information is shared accurately.

We will be building a call tree object to represent a real-world call tree. Unsurprisingly, we will be using a tree structure to represent this call tree (it's kind of the natural choice). Our call tree will be a binary tree, modeling a simplified call tree in which each person has two other people to notify. We will call these people "first" and "second" Ideally one should always contact their first contact before contacting their second. (The first person should be more important to contact, often having more people in their side of the tree or more responsibilities). Building this data model would be the first (and most important) step in building a tool for automatically creating, maintaining, and updating call trees for a real world social group.

The image below depicts a simple call tree: Mats calls Daniel then Min. Daniel Calls Mitra and Carson while Min calls Asa and Anwaar. Finally Carson calls Alex and Sriram.



From a tree like this, we could answer a few questions:

- How many people are in the call tree (9)
- How many rounds of calls before we reach everyone (3–Mats does his calls, then Min and Daniel do theirs, then Carson does the final round)
- Would Asa get called if Mats sent a message (yes)
- Would Lily get called if Mats sent a message (no, Lily isn't in the tree!)
- Is Alex a "caller" (I.E. does Alex call anyone?) No.

5 Formal Requirements

You are required to write the two following classes:

1. CallTreeNode
2. CallTree

5.1 CallTreeNode

This class should be pretty straightforward to you. The real work will be happening in the other class.

Each `CallTreeNode` represents one person in the call tree, and the zero, one, or two other people they are expected to call, labeled "first" and "second" respectively. The `CallTreeNode` class will have:

- a private String variable to track the name of the person who this node represents.
- two private `CallTreeNode` variables to track the people who this person is supposed to call.
- A constructor that takes initial values for name, first, and second. The first and second variables will be null if the person represented by this node is not expected to call anyone else. It's possible for only one of these to be null, in which case the person represented by this node is only expected to call one other person.
- You may find it useful to create an additional constructor that takes only a name, setting first and second to null. This is allowed, but not required.
- the following get and set methods, which do what you might expect (get and set the related private variables):
  - `public String getName()`
  - `public CallTreeNode getFirst()`
  - `public CallTreeNode getSecond()`
  - `public void setFirst(CallTreeNode)`
  - `public void setSecond(CallTreeNode)`
- a method: `public boolean isCaller()` that returns true if the person represented by this node is a "caller" i.e. someone who is expected to call other people in the call tree.
- all methods on this class are expected to run in $O(1)$ time.

You are allowed to add other methods if you wish, but these are the only ones we would be expecting.

5.2 CallTree

The `CallTree` class represents the call tree as a whole. This class should allow the person using it to simply manipulate the call tree without worrying about the node structure we are using to track the call tree data.

The `CallTree` class must have:

- a single private variable – a CallTreeNode. I'll be calling this root in my code, but you can name it was you want. No other instance variables are expected.
- a constructor that takes a single String parameter. The string variable represents the name of the base of the call tree (think of this as the first caller or first reporter). In the earlier example, for example, it was Mats. This should create a new node representing this person and assign it as the root. The person should initially not be calling anyone.
- A private method `private CallTreeNode find(String name, CallTreeNode node)`. This function will return null if the given name is not part of the CallTree at or under the given node, if the given person is part of the call tree it should return the call tree node for that person. You often won't call this function directly, not even in your own code, the next function exists for that.
  This function should implement a recursive traversal like we discussed in lecture at the end of Monday (and we will see more of on Wednesday). Remember – to design these functions you must think recursively, so not "loop over all nodes to find the name". Instead, it's "is this node the node we are looking for? is the node we're looking for in the left sub-tree? Is the node we're looking for in the right sub-tree?"
- A private method: `private CallTreeNode find(String name)`. This function will actually be very simple, serving only to call the two parameter find function with the root node to "start" the recursion. If the recursive function returns null it's safe to say that the given name isn't in this call tree. If it returns a node, that node should be returned as it is the CallTreeNode for the given person.
  These find methods should be private. The concept of a CallTreeNode isn't something the person using this class should have access to at all. However as these functions serve to make every other function easier, you may still wish to test it. No worries. Here's a main method you can add to CallTree that can help you begin testing the find functions.

```
public static void main(String[] args) {
  CallTreeNode ctn1 = new CallTreeNode("Coffee", null, null);
    CallTreeNode ctn2 = new CallTreeNode("Flute", null, null);
    CallTreeNode ctn3 = new CallTreeNode("Ready", null, null);
    CallTreeNode ctn4 = new CallTreeNode("Magic", ctn1, ctn2);
    CallTreeNode ctn5 = new CallTreeNode("Cookies", ctn3, null);
    CallTreeNode ctn6 = new CallTreeNode("Roast", ctn4, ctn5);

    CallTree ct = new CallTree("whatever");
    ct.root = ctn6;

    // all true.
    System.out.println(ctn1 == ct.find("Coffee"));
    System.out.println(ctn2 == ct.find("Flute"));
    System.out.println(ctn3 == ct.find("Ready"));
    System.out.println(ctn4 == ct.find("Magic"));
    System.out.println(ctn5 == ct.find("Cookies"));
    System.out.println(ctn6 == ct.find("Roast"));
    System.out.println(null == ct.find("Fire"));
}
```

Neither find method should change the structure of the tree.

- a public method: `public boolean inCallTree(String person)`. This should return true or false to indicate if the given person is in the call tree. This function should not change the structure of the tree.
- a public method: `public boolean shouldCall(String person, String first, String second)`. This should modify the tree structure so that the given person is recorded as calling the two listed people (in the provided order.) (In tree terminology, this should change a leaf node in the tree into an internal node by adding two new leafs as children). The function returns a boolean to indicate if this was successful (true) or couldn't happen (false).

  The function should return false if any of the following is true:

  - The person listed (first parameter) is not currently in the call tree
  - The person listed (first parameter) is currently a caller (I.E. already expected to call one or more people)
  - The people listed for first and second are currently in the call tree (are already being called)

  If any of these errors occur you shouldn't modify the data and should just return false. If none of the above error conditions happen then the node representing the person (first parameter) should be modified to have a new first and second child nodes representing the listed first and second people. The first and second people should initially not be calling anyone.
- a method `public String getFirstCall(String name)` this should look up who the provided person (name parameter) will call, returning the name of the person they should call first. If the named person is not a caller, or is not in the tree, return null.
- a method `public String getSecondCall(String name)` this should look up who the provided person (name parameter) will call, returning the name of the person they should call second. If the named person is not a caller, or is not in the tree, return null.
- a method `public int getCallCount()` that returns the number of rounds of calls before everyone has been contacted. This is (conveniently) just the height of the tree – I.E. the maximum number of edges between the root and one of it's leafs.
- a private method `private int getHeight(CallTreeNode node)` This should implement a recursive traversal to compute the height of a given node. Remember – the height is the maximum length of any path from this node to a leaf. (This function has an easy recursive definition since this is your first function we will give the recursive structure to you as a hint. If given null (I.E. if not given a node) return -1. Otherwise, return 1 plus the maximum height of the first and second child-nodes of the provided node.)
- a method `public int terminalContactCount()` that returns the count of "terminal contacts" (I.E. non-callers, or leafs in the tree)
- a method `private int terminalContactCountHelper(CallTreeNode node)` This function implements the recursive traversal to compute the number of leaf nodes accessible from a given node. We will let you think about how best to define this recursive function. Think it through using the ideas described above and you should be fine (It does not

need to be complicated).

6 Submission

For this lab you need to turn in two files:

- `CallTree.java`
- `CallTreeNode.java`

You can submit other files if you wish, but we do not promise to look at them during grading.

7 Grading

This assignment will be partially manually graded, and partially automatically graded. 50 points will be graded automatically – based directly on the provided tests. An additional 50 points will be set aside for manual review and partial credit.

- 15 autograder points for the call tree node class
- 35 autograder points for the call tree class
- 10 points code style – see past assignments for expectations here.
- 10 points manual grading for the tree node class
- 30 points manual grading for the tree class