Programming Problems

For each of these problems (unless otherwise specified), write the needed code directly in the Python file, in the corresponding function definition.

All programming problems may also be checked by running 'Run File As Script' on the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `findParents(t, name)` - 10pts

Can attempt after Search Algorithms II lecture

Write the function `findParents(t, name)` which takes a genealogical family tree in the form of a binary tree (as discussed in class) and a string, the name of a person in that family tree. The function returns a list containing the named person's parents as shown by the tree.

You'll need to use if statements to separate the four possible arrangements of parents.

- Return an empty list if the parents are unknown
- Return a singleton list in the two cases where only one parent is known
- Return a two-element list if both parents are known.

You are guaranteed that every name in the tree only shows up once (to avoid conflicting information) and that the person being searched for is in the tree.

(cont'd on next page)

For example, consider the following family tree (as shown in the course slides):

```
t = { "contents" : "Arya",
      "left"  : { "contents" : "Ned",
                  "left"  : { "contents" : "Rickard",
                              "left"  : None, "right" : None },
                  "right" : { "contents" : "Lyarra",
                              "left"  : None, "right" : None } },
      "right" : { "contents" : "Catelyn",
                  "left"  : { "contents" : "Hoster",
                              "left"  : None, "right" : None },
                  "right" : { "contents" : "Minisa",
                              "left"  : None, "right" : None } } }
```

If we called `findParents(t, "Ned")`, the function would return the list `["Rickard", "Lyarra"]`.

If we called `findParents(t, "Rickard")`, the function would return `[ ]`.

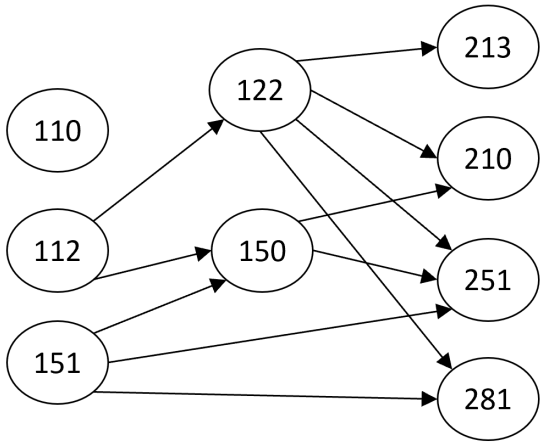
Hint 1: treat this like a recursive search problem (in fact, you might want to reference tree linear search and binary search if you get stuck). You'll need to make **two** base cases - one for when you find the person, one for when you reach a leaf or an empty tree. For this problem, it may be easier to set up the base case as an empty tree.

Hint 2: unlike recursive binary search, you need to check **both** branches of the tree to see if the person occurs in either branch. How can you combine their results? If the person is found in one of the branches, you should get a list with the parents; if they are *not* found, you'll get an empty list. Just concatenate the two results together to get the final result.

#2 - `getPrereqs(g, course)` - 10pts

Can attempt after Graphs lecture

College course prerequisites are notoriously complicated. However, we can make them a little easier to understand by representing the course dependency system as a **directed graph**, where the nodes are courses and an edge leads from course A to course B if A is a prerequisite of B. For example, the core Computer Science courses (almost) produce the following prereq graph:



Which would be represented in code as:

```
g = { "110" : [],
      "112" : ["122", "150"],
      "122" : ["213", "210", "251", "281"],
      "151" : ["150", "251", "281"],
      "150" : ["210", "251"],
      "213" : [],
      "210" : [],
      "251" : [],
      "281" : [] }
```

Write the function `getPrereqs(g, course)` that takes a directed graph (in our adjacency list dictionary format, without weights) and a string (a course name) and returns a list of all the immediate prerequisites of the given course. If we called `getPrereqs` on our graph above and `"210"`, for example, the function should return `["122", "150"]`.

Hint: you can't just return the neighbors of the course, because the edges are going in the opposite direction! Instead, iterate over all the nodes to find those that have the course as a neighbor. Construct a new list out of these nodes as the result.