15-110 Check4 - Written Portion

Name:

AndrewID:

#1 - Tree Vocabulary - 15pts

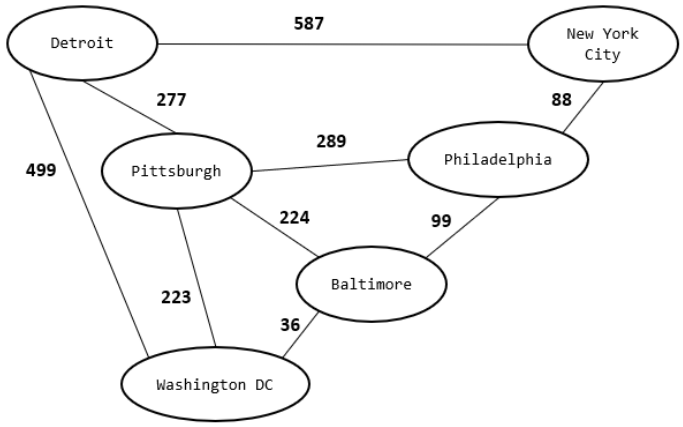
Consider the following tree, implemented in code with our dictionary implementation:

```
t = { "value" : "A",
      "left"  : { "value" : "B",
                  "left"  : None,
                  "right" : None },
      "right" : { "value" : "C",
                  "left"  : { "value" : "D",
                              "left"  : None,
                              "right" : { "value" : "E",
                                          "left"  : None,
                                          "right" : None } },
                  "right" : { "value" : "F",
                              "left"  : None,
                              "right" : None } } }
```

How many **nodes** does this tree have?	
Which nodes are **children** of the node with value "C"?	
What is the value of the **root** of the tree?	
What are the values of the **leaves** of the tree?	
If we ran the first version of the function `countNodes` from lecture on this tree (with leaf base case), how many total **recursive function calls** would be made?	

#2 - Graph Vocabulary - 15pts

Consider the graph below, shown both in visual and adjacency list (dictionary) format.




```
g = {
  "Detroit" :       [ ["New York City", 587],
                      ["Pittsburgh",    277],
                      ["Washington DC", 499] ],
  "New York City" : [ ["Detroit",       587],
                      ["Philadelphia",   88] ],
  "Pittsburgh" :    [ ["Detroit",       277],
                      ["Philadelphia",  289],
                      ["Baltimore",     224],
                      ["Washington DC", 223] ],
  "Philadelphia" :  [ ["New York City",  88],
                      ["Pittsburgh",    289],
                      ["Baltimore",      99] ],
  "Baltimore" :     [ ["Pittsburgh",    224],
                      ["Philadelphia",   99],
                      ["Washington DC",  36] ],
  "Washington DC" : [ ["Detroit",       499],
                      ["Pittsburgh",    223],
                      ["Baltimore",      36] ]
}
```

How many **nodes** does this graph have?	
How many **edges** does this graph have?	
What are the **neighbors** of the node with value "Baltimore"?	
Are the edges **weighted** or **unweighted**?	
Are the edges **directed** or **undirected**?	

#3 - Tracing Graph Code - 15pts

The function below takes a graph (in adjacency list format) and the value of a node in that graph. The graph represents a map, where nodes are cities and edge weights are distances between cities. The graph shown in #2 could be an input to this function.

```
def mystery(g, node):
    a = None
    b = 1000
    for val in g[node]:
        c = val[1]
        if c < b:
            a = val[0]
            b = c
    return a
```

In the context of the map graph (using non-code terms), what is the relationship between the variable **node** and the variable **c** in a specific loop iteration?

If you trace through the code with the graph shown in #2 and the node set to the value "Philadelphia", what values do **a** and **b** hold at the end of the function call?

In general (abstract, non-code) terms, for any given map graph and city node, what does this function return?

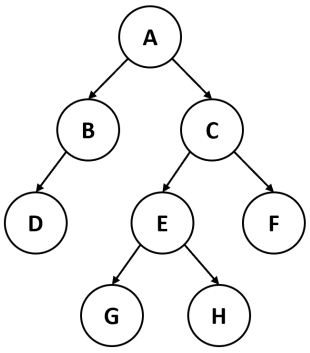
15-110 Check4 - Programming Portion

Each of these problems should be solved in the starter file available on the course website. Submit your code to the Gradescope assignment Check4 - Programming for autograding.

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

#1 - `getLeftmost(t)` - 15pts

Write the function `getLeftmost(t)` that takes a binary tree in our dictionary format and returns the contents of the **leftmost** child of that tree. This is the child we reach if we keep moving down and left from the root node until we cannot go left any further. For example, in the tree:



Which is represented as the dictionary:

```
t = { "contents" : "A",
      "left" : { "contents" : "B",
          "left" : { "contents" : "D", "left" : None, "right" : None},
          "right" : None },
      "right" : { "contents" : "C",
          "left" : { "contents" : "E",
              "left" : { "contents" : "G", "left" : None, "right" : None },
              "right" : { "contents" : "H", "left" : None, "right" : None } },
          "right" : { "contents" : "F", "left" : None, "right" : None } } }
```

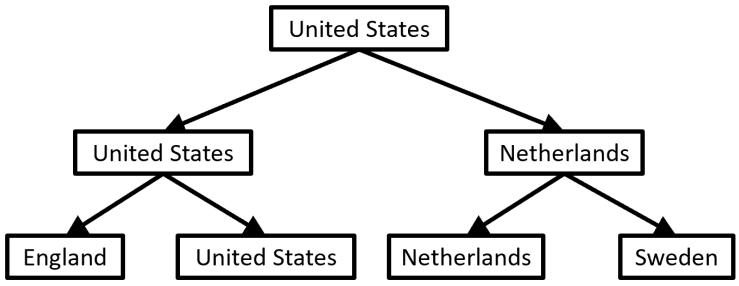
We go from A to B, then from B to D, then we can't go left any further. `"D"` is the contents of the leftmost node and is returned when we call the function on `t`.

Hint: you can solve this using recursion, or you can just use a while loop.

#2 - `getInitialTeams(bracket)` - 25pts

We can represent a tournament bracket from a sports competition as a binary tree. To do this, store the winning team as the root node. Its children are the winning team again, as well as the second-place team. In general, every node represents the winner of a match, and its two children are the two teams that competed in that match.

For example, the following bracket represents the last two rounds of the Women's World Cup in 2019.



In our binary tree dictionary format, this would look like:

```
t1 = { "contents" : "United States",
       "left" : { "contents" : "United States",
           "left"  : { "contents" : "England", "left" : None, "right" : None },
           "right" : { "contents" : "United States", "left" : None, "right" : None}},
       "right" : { "contents" : "Netherlands",
           "left"  : { "contents" : "Netherlands", "left" : None, "right" : None },
           "right" : { "contents" : "Sweden", "left" : None, "right" : None } }
    }
```

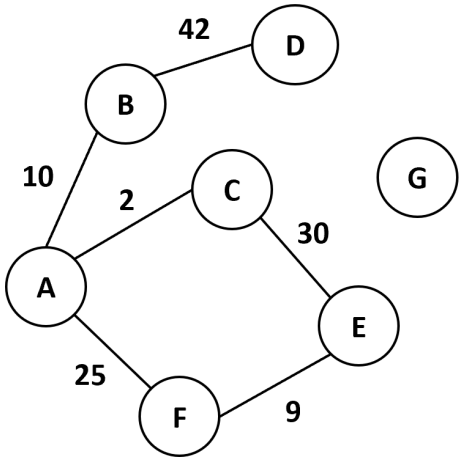
Write the function `getInitialTeams(bracket)` which takes a tournament bracket and returns a list of all the teams that participated in that tournament. For example, if the function is called on the tree above it might return `[ "England", "United States", "Netherlands", "Sweden" ]`. You will need to implement this function **recursively** to access all the nodes. We recommend that you start by looking at the `sumNodes` and `listValues` examples from the slides.

Hint 1: how can we get all of the teams to show up in the list exactly once? Every team occurs at the very beginning of the tournament, in the first set of matches. In the tree, this is represented by the **leaves,** so you should not include values on non-leaf nodes.

Hint 2: make sure the **type** you return is the same in both base and recursive cases!

#3 - `largestEdge(g)` - 15pts

We often want to find the **largest edge weight** in a graph. This can help us identify useful information, like the most congested street in a city or the two gas stops that are farthest apart on a highway. Write the function `largestEdge(g)` that takes a weighted graph in our dictionary format and returns a list holding two elements - the two endpoints of the edge with the largest weight in the graph. For example, in the graph:



Which is represented as the dictionary:

```
g = { "A" : [ [ "B", 10 ], [ "C",  2 ], [ "F", 25 ] ],
      "B" : [ [ "A", 10 ], [ "D", 42 ] ],
      "C" : [ [ "A",  2 ], [ "E", 30 ] ],
      "D" : [ [ "B", 42 ] ],
      "E" : [ [ "C", 30 ], [ "F",  9 ] ],
      "F" : [ [ "A", 25 ], [ "E",  9 ] ],
      "G" : [ ] }
```

The largest edge has the weight `42`. That edge is between the nodes B and D, so if we call the function on that graph, it will return `[ "B", "D" ]` (or `[ "D", "B" ]` - the order doesn't matter).

To find the largest edge, modify the find-most-common/find-largest-item pattern we've discussed several times in class. Iterate over each of the nodes in the graph, then for each node iterate over each of that node's neighbors to visit each edge.

Note: to make this easier, you are guaranteed that all edge weights will be **positive** and there will be at least one edge in the graph.