Section Handout 8

Problem One: Breadth-First Search

All trees are graphs, though not all graphs are trees. Therefore, it is possible to use breadth-first search to traverse a tree structure.

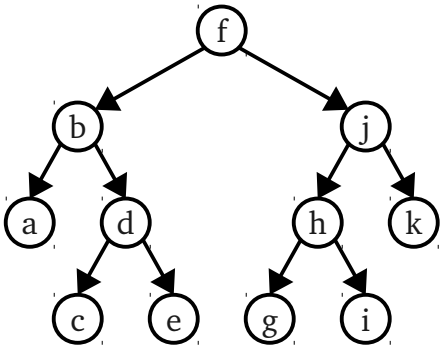
Suppose that you have a binary search tree where nodes are represented using the following **`struct`**:

```
struct Node {
    string value;
    Node* left;
    Node* right;
};
```

Write a function

```
void breadthFirstSearch(Node* root);
```

that accepts as input a pointer to the root of a binary search tree, then prints out all the nodes in the tree using a breadth-first search. What will the output of your function be on the following tree?



Under what circumstances will your function list all the nodes in a BST in sorted order?

Problem Two: Functions as Data

The function that you wrote for Problem One allows you to visit every node in a binary search tree, performing some simple operation (namely, printing out the node contents) at every node. Let's suppose that you want to modify this function so that instead of printing out the contents of each node, you want to (say) write the contents of that node to a file. Currently, you would have to copy the entire function and replace the line that prints out the node contents so that it instead writes the node contents to a file.

More generally, over the course of the quarter, we have written many functions (most of them recursive) that generated or visited all objects of some type and performed some action on them. If we want to change what action we perform on those objects once we generate or visit them, we have to duplicate the entire function just so that we can change that action. This seems wasteful and complicated.

Fortunately, C++ offers a simple way to solve this. It is possible to pass C++ functions as arguments to other C++ functions. For example, consider the following function:

```
void tabulateFunction(double start, double stop, double step,
                      double function(double arg)) {
    for (double val = start; val <= stop; val += step) {
        cout << function(val) << endl;
    }
}
```

This function accepts four parameters. The first three parameters are **double** s indicating a range of **double**s and a step size over that range. The final parameter is itself a function that takes in a **double** and returns a **double**. The **tabulateFunction** function then iterates over the specified range, calling the requested function over each point in the range and printing the result out. We could then use this function in the following ways to tabulate the values of sine, cosine, and tangent over the range of values $[0, \pi / 4]$ with a step size of 0.01:

```
const double kPi = 3.141592653589793238462643; // Close enough.

tabulateFunction(0, kPi / 4, 0.01, sin);
tabulateFunction(0, kPi / 4, 0.01, cos);
tabulateFunction(0, kPi / 4, 0.01, tan);
```

Here, the functions **sin**, **cos**, and **tan** are defined in the **<cmath>** header file.

Using the fact that you can pass functions as input to other functions, rewrite your **breadthFirstSearch** function from Problem One so that it has the following signature:

```
void breadthFirstSearch(Node* root, void processFn(Node* node));
```

This function should process the nodes of the tree in breadth-first order by calling the function **processFn** on each of them. This lets the client of the function control what should be done to each node in the tree.

Problem Three: Depth-First Search

In lecture, we chose to represent graphs as a **Map<string, Vector<string>>** where each key is the name of a node and each value is the list of all outgoing edges from that node. However, there are many other ways that we could conceivably represent a graph. One alternative approach is to represent a graph as a function that accepts as input a **string** representing a node and produces as output a **Vector<string>** representing all the nodes connected to that node. This representation is useful when there are a large number of edges in the graph that follow a predictable pattern, since the edges can be computed as necessary rather than stored in memory at all times.

Write a function

```
Vector<string> depthFirstSearch(string start, string end,
                                Vector<string> edgeFunction(string nodeName));
```

that accepts as input a start node, an end node, and a graph represented as a set of all the nodes in the graph and a function mapping nodes to their outgoing edges, then runs a depth-first search over the graph to find a path from **start** to **end**. If a path is found between them, your function should return a **Vector<string>** containing the nodes in that path. If no path is found, your function should return an empty **Vector** to signal this.