Iterating Over A Graph, Reading

Iteration is to graphs what traversal is to arrays and linked lists, and VLR, LVR, and LRV are to binary trees. It's working all paths through a graph to "visit" each node.

But it's a bit different for graphs, because unlike all the other data structures we've studied, it's possible to loop around and visit nodes more than once! Also, if there are any one-way connections, it's possible for there to be "dead ends".

Also, there is no specific starting place. Arrays have index zero, linked lists have a head, and trees have a top. But there's no start node for a graph. Or there are lots of start nodes, because *any* node could be a starting point. Certainly if we have a road map and want to use it to find a way from any city to any other, then any city can be the starting point.

There are three common ways to iterate over a graph. The easiest is simply to traverse the database's array! That gets to all nodes, but unlike any other traversal we've ever seen, it totally disregards linkages. That may be okay for some situations, like finding the index of San Francisco in a road map graph. But in case connections *are* important, there are two other ways: BFS and DFS -- both of which start at a chosen node and find all *reachable* nodes starting from there. Of course, as long as there are no one-way connections, *all* other nodes are reachable.

BFS (Breadth First Search)

The process for **BFS** (http://cpp.datastructures.net/presentations/BFS.pdf) is to start with all the nodes that are neighbors. Then for each neighbor, find *its* neighbors. Those should be 2 steps away from the start. Then find *those* neighbors' neighbors, 3 steps away from the start, and then keep going until the furthest nodes are found.

Of course, the first node is a neighbor of all *its* neighbors, so it will be important to track previously "visited" nodes so they get skipped and not double-counted. We also want to avoid endless looping! For that we'll need to *add our first tracking variable* to **struct Node**:

```
bool isVisited;
```

In this "fanning out" process, the order of nodes finds the *closest* ones first. In case of a tie (as all of the first node's neighbors are just one step away), the tie breaker is their order of appearance in the **list<int> neighbors;** attribute (for lack of anything better).

The coded solution for BFS will have to return a *list of indexes* of the reachable nodes. As it finds them, it can add them to the end of this list, and when the list gets processed later, it would start at the *front* of that list: FIFO. A good choice in that case is an STL **queue<int>**. A BFS value-returning function could return a queue, just like the **keys** functions in our associative arrays. (If fact, a BFS function would actually be a graph version of a **keys** function!)

Stepping through the graph certainly looks like a recursive solution, because as we work on the neighbors of one of the start node's neighbors, the *other* neighbors have to wait their turn to be processed. It's like the recursive loops in the BST functions that needed to visit every node. But we could write those functions easily because they had a *known* number of links, with names. But this time, it's a bit more complicated...

Recall that there are *two* ways to implement recursive loops -- function calls and ones that use "to-do" lists, as we did in the quicksort algorithm. The latter is what we'll use here: the "to-do" list. Here's how:

```
create an empty result queue of ints to return at end of function call
create another queue of ints to store the "to-do" list
initialize each node in database: set to "not visited"
mark starting node as "visited"
push start node's index onto the result queue
push start node's index onto the "to-do" list
while the "to-do" list is not empty
  peek/pop a node's index from the "to-do" list
  for each of that node's neighbors
    if neighbor has not yet been visited
      mark neighbor as visited
      push neighbor's index onto the result queue
```

```
      push neighbor's index onto the "to-do" list
return the result queue
```

DFS (Depth First Search)

While BFS fans out a step away from the start at at time, **DFS** **(http://cpp.datastructures.net/presentations/DFS.pdf)** instead takes one path from the start and probes as far as it can go. Then it backs up (using a "to-do" list) to the last node with a path as yet not taken, and probes in its direction. Here is its algorithm -- it uses a stack for it's "to-do" list, because of the "backing up to the last node".

```
create an empty result queue if ints to return at end of function call
create a stack of ints to store the "to-do" list
initialize each node in database: set to "not visited"
push start node's index onto the "to-do" list
while the "to-do" list is not empty
  peek/pop a node's index from the "to-do" list
  if that node has not yet been visited
    mark the node as visited
    push node's index onto the result queue
    for each of that node's neighbors (in reverse order)
      if neighbor has not yet been visited
        push neighbor's index onto the "to-do" list
return the result queue
```

The reason for the "in reverse order" is that we already decided that the tie-breaker should be the order that neighbors are listed in **list<int> neighbors;**. They're being pushed into a LIFO stack, so to preserve their order (and still put them all ahead of other already-waiting nodes in the "to-do" list, put them in *backwards*.

Graph Search Functions

These should be very similar to the AssociativeArray::keys function, returning a queue. But they are not templated members, and they need input -- namely, the starting node as defined by its index. The stand-alone prototypes would be:

```
queue<int> BFS(int startNodeIndex, vector<int>& database);
queue<int> DFS(int startNodeIndex, vector<int>& database);
```

Note that the database is also shared as a parameter -- either that or it has to be global, and we like to avoid global variables, even if they are objects.

Here's how the function would be called from the main program and its results used (using copy-pop):

```
int start = ... // start node's index
for (queue<int> result = BFS(start, database); !result.empty(); result.pop())
{
  int nodeIndex = result.front;
  ...database[nodeIndex]... // do something with the node
}
```