The Ultimate Linked Structure, Reading

Every linked structure we've studied so far has had limits. A singly-linked list node can only link to the next node after it. A doubly-linked node can link to the nodes before and after. Trees nodes can link only to *their* children, which cannot be shared with other nodes. All tree nodes have just a single parent. Tree nodes cannot link to nodes at their own level or higher. BST nodes can have only two children, and their left child has to be <= it, and the right >=.

What if we wanted to have a data structure to represent interconnected cities on a map, or interrelated tasks in a project chart, or turns in a maze? None of the above allow that, and it's just because they have restrictions on the numbers of links they can have in their **struct Node** and on who else they can link to. What we need is:

1. a **struct Node** that allows *any* number of links, and
2. freedom to link to *any* other node, including one's self.

There's a data structure for that -- it's called a "graph".

Graphs

Graphs are linked structures of interconnecting nodes. Unlike the data structures we've studied so far, they are typically developed for specific applications, and as such are not templated, nor do they have to be classes. But they do represent nodes as objects, with their data contents and links. We'll continue to use **struct Node** for this, but it will be in the main program, and not as a private member of a templated class.

Nodes

Nodes actually contains three things:

1. their data value(s)
2. their links to other nodes
3. tracking variable(s) *new*

We already mentioned the first two. The third is not so obvious, but in our development in this module the need for it will become clear and we'll add them as we go. For now, just remember that nodes may have overhead in addition to links.

The node data depends on the application. We'll use a road map example for our development, were the nodes are cities, interconnected by freeways. For the data value(s), we just need the city name. We could include other information, like elevation and population, if those were useful to our application, but for our development we just need a name.

The links have been specific **Node*** pointers with specific names, like "next" and "prev" and "left" and "right". Naming them in this way limits how many links that we can have. So we need another way to store links -- why not an array of links? An array that self-adjusts its capacity as needed? We have all kinds of options for implementing those!

We could have stacks or queues, but it's hard to "see inside" of those. We could use C++ STL templates like vector or deque or list, or our own DynamicArray. Or maybe an associative array. Since we'll only be iterating over the whole set of links when we use it, and not searching for any by index or key, a good choice is the STL list, like we used in our chained hash table -- a list of **Node***'s. We'll call the collection of connected nodes "neighbors". So:

```
struct Node
{
  string city;
  list<Node*> neighbors; // to be modified below
};
```

No need for any "tracking variables" yet, so we'll just leave it like this for now. But don't get too attached to the way links are represented, because that's going to change (below).

The Database

At this point we have just a loose collection of nodes, linked any way we want. But this means that our application has to declare individual nodes as named objects. That's okay, but it's going to make it hard to search them. Say we want to use our road map application to find a route from San Francisco to Phoenix. it's going to be hard to look through the whole interconnected set of nodes for a name match.

A better way than individually named and declared Node objects is to put them into an array. Then we can use a simple for-loop to search this "database" of nodes. If we know the exact number of nodes ("N"), then a regular array is perfect:

```
const int N = ...
Node database[N];
```

Or if we don't know, and we're reading the cities and their connections from an external database, we could use an STL vector and "push_back" Node objects to it as they are created.

```
vector<Node> database;
```

No matter which we choose, a node is still referred to in code as **database[i]**.

The Node, Redefined

The links to neighboring nodes are their **Node*** memory locations. We did it this way because that's how we've *always* done it. Neighbor nodes of the node stored at index "i" would be traversed like this:

```
list<Node*>::const_iterator it;
for (it = database[i].neighbors.begin(); it != database[i].neighbors.end(); it++)
  **it is a Node
```

...a doubly dereferenced pointer. But since we put all the nodes into an array, why couldn't we just use *index* of the neighbor's position in the database instead of the memory location of that indexed position? Like this:

```
struct Node
{
  string city;
  list<int> neighbors; // the neighbors' indexes in the database array
};
```

Then neighbors are traversed like this instead:

```
list<int>::const_iterator it;
for (it = database[i].neighbors.begin(); it != database[i].neighbors.end(); it++)
  database[*it] is a Node
```

In any case, don't get too attached to this representation either, because it will change again by the time we're done. But it will serve us for now.