PAGE 0 (on this page we may even list all available tutorials)

DATA STRUCTURES AND ITERATORS TUTORIAL (with code written in PHP)

In this tutorial we will explain what some discrete data structures such as graphs, trees, and linked lists are. We will then move on to explore how code can be developed to address the need of traversing these structures and accessing their elements in a systematic manner, possibly also modifying them in place or copying them to newly created and possibly altered data structures along the way. In doing so, we will learn about and put into practice the concept of code refactoring, and as part of our incremental approach to code development will also stumble across the iterator design pattern.

[Proceed to Tutorial]

---

PAGE 1

What is a graph?

A graph G is a data structure made up of a set V of nodes (also known as vertices) and a set E of edges each of which uniquely connects any given pair of distinct nodes together. A node is simply any object in our data space that we want to model. Diagrammatically, a node can be represented as a circle. Edges, on the other hand, represent the existence of a relationship of any given kind between any two nodes in a graph. Diagrammatically, edges are represented as a (straight or equivalently bent) line segment connecting any two such nodes.

[GRAPH DIAGRAMS HERE 1]

Among other things, graph edges can also have a direction, that is, they may begin at one of the nodes (the beginning node or head node) and end at the other one of the nodes it connects (the ending node or tail node). When all edges in a graph follow this pattern we have what is called a directed graph D (also known as a digraph), and its edges are known as directed edges (or arcs). In this case the uniqueness of each edge is to be understood as existing with respect an edge’s direction so that between any two nodes there may be no arcs, one arc, or two arcs, one in each direction.

[DIRECTED GRAPH DIAGRAMS HERE 2]

What is a tree?

A tree is a special case of a graph where whenever you start from any node, and walk along an untreaded relationship edge from such node to a node at the other end, and continue doing so for as long as you can or wish to do so, you will never encounter a node which had already been visited. Whenever a graph has such property it is also known as having no cycles, so a tree is simply a graph with no cycles.

[DIAGRAM OF SOME PATHS IN A TREE 3]

Whenever we have a tree, we can designate any node as the root node, and redraw the tree so that the root node is at the top, at level zero, the nodes the root node is connected to are one level below it, at level one, the nodes connected to the nodes at level one are just below these, at level two, and so on, with the existence of edge relationships between the nodes remaining unmodified as in the following diagram:

[DIAGRAM OF A TREE WITH DRAWN WITH A ROOT NODE AND LEVELS]

Trees in practice.

Note by the very definition of a tree, since there are no cycles in a tree, once we designate a tree node as a root node we are automatically considering a directed graph where the root node at level zero has its directed edges pointing to the nodes at level one, each of whose nodes has directed edges pointing to nodes at level two, and so on. Also, while the child nodes of theoretical trees are usually unordered, in order to store these as part of our code we will normally need to impose an order on such children so that for each parent node there is a first child node, a second child node, a third child node, and so on.

[]

Linked lists.

As a special case of a directed tree, we can have a linked list, which is simply a head node which is linked to its next node which is linked to the next node and so on until the tail node is reached. A linked list may also have backlinks so that it may be traversed from the tail node to the head node in reverse order. Then of course, there is also the case of a circular linked list, which is not a tree, because the tail node connects back to the head node, so that there is no actual head node and tail node, only a current node.

[]

[WALKS, CIRCUITS, AND PATHS]

Let us now cover some more terminoogy which may help us assign corresponding meaningful names to elements in our code corresponding to core concepts. Whenever we have a graph data structure, we can designate any node as the current node and traverse the graph from an initial node (also known as the source node), stepping along edges linking the current node to the node on the other side of the edge which then becomes the new current node (respecting the direction of arcs in the case of a directed graph as this is carried out), and continuing in this manner, finishing at the final node (also known at the destination node). Whenever we do this we call the sequence of vertices and edges so traversed a walk. When the edges don’t repeat we call the walk a circuit. When the vertices don’t repeat we call the walk a path. When a walk repeats only the initial and final edges it is known as a cycle. Note that whenever in a walk edges are repeated vertices must repeat as well, but a circuit can have repeated vertices, so a path is a circuit but a circuit may not be a path.

[EMPTY AND TRIVIAL CASES]

As pathological cases, a data structure can also be made up of a single element, in which case it is said to be trivial, or even no elements at all, in which case it is said to be empty.

[DIAGRAM HERE]

What other properties can nodes and edges possess?

Nodes can possess any number of properties. Typical properties may include a name (also known as a label), a value, a color, or any other custom data we may want to attach to it. In the cases of edges, besides an edge direction and edge could also have any other custom data attached to it. When writing code it is convenient to start with some generic class describing the graph, linked list, or tree nodes, and then use specialization to subclass these to create less generic entities entailing everything needed to solve the problem in the given problem domain at hand.

[PSEUDOGRAPHS, MULTIGRAPHS, PSEUDOMULTIGRAPHS, and DESIGNS]

Occasionally we might need to deal with situations where there is more than one distinguishable (distinguishable as in more than one edge for regular graphs and more than one directed edge in each direction for digraphs) edge between any of two nodes (thus obtaining what is called a pseudograph) or nodes with edges joining the same node to itself (thus obtaining what is called a multigraph), or both (this obtaining what is called a pseudomultigraph).

[DIAGRAMS HERE]

Another generalization of graphs involves considering edge relationships which may relate any number of vertices together. In this case, to represent a design pictorially, instead of joining any two vertices with a curve as we did formerly, we represent each edge by a doubled circle, and join all node circles of nodes related via this edge to to this double cirle edge by means of (straight or equivalently bent) line segments.

[DIAGRAMS HERE]

As an extreme and rarely useful case, we could have a pseudomultidesign, whereby any nonempty subset of nodes can be an edge (similar to what is known by the name of a discrete topology on the set of nodes, where the edges would be the open sets, with the only difference being that for such structure the entire set of vertices and the empty set must not be included as edges and any given subset may not be repeated more than once). Edges associated with the same subset of vertices may carry different associated properties. And this concludes our analysis of data structures.

[CODING CONSIDERATIONS]

To model normal multipseudodesign data structures, we would usually code a class corresponding to each node, and maintain an array of child nodes within each instance of such a class. One could ask, why not deal with the most general case and treat other cases by subclassing? This could indeed be done, but it may be more efficient or cleaner to avoid this and create unrelated classes to dealing with the most generic useful case at hand.

In practice, data strucures can be much more heterogenous than what Is usually found in mathematics books. For example, we could have a data structures whose nodes may host any number of properties as well as any number of pointers to any number of data structures which may have nothing in common (and this is indeed the case when we look at operating system code data structures).

---

PAGE 2

So much for the theory. Now let’s get down to the code. In this tutorial we will deal with the special case of trees having a nameless root node, named nonleaf nodes, and leaves having a name and a value. As applications you could think of the scenario where a corporate website may present a multilevel menu at the top of the page (here the root node is simply a reference to the menu stucture, nonleaf nodes represent named main menu and submenu entries, and leaf nodes represent web pages which may have a handle to display in the web browser URL as well as a value as the filename for the page contents to be displayed), a social web application where a container in the sidebar allows contacts to be organized in folders and subfolders (here the root node is simply a reference to the contact list, nonleaf nodes represent the folders and subfolders the contacts are organized into, and the leaf nodes are the contacts themselves), or a similarly organized cloud storage application (where the root node is simply a reference to the data stored in the cloud, intermediate nodes are named folders, and leaf nodes are files, which may have a filename as well as a value made up of other properties such as creation, last modification, and last access times, and others).

treeDescent-v1.php

In this file we begin by including "includes/treeNode-vX.php" which is where we will define our TreeNode class:

require\_once("includes/treeNode-vX.php");

We then write a function that shows us how a test tree instance could be created. For now the static function calls included therein for creating the nameless root node, other nonleaf nodes, and leaf nodes, are written, giving us an idea of how a tree can be instantiated.

function createTestTreeInstance() {

return TreeNode::createNamelessNonLeafNode(array(

TreeNode::createNonLeafNode("node1", array(

TreeNode::createNonLeafNode("node11", array(

TreeNode::createLeafNode("node111", "The"),

)),

TreeNode::createNonLeafNode("node12", array(

TreeNode::createLeafNode("node121", "fishermen"),

TreeNode::createLeafNode("node122", "know"),

)),

TreeNode::createNonLeafNode("node13", array(

TreeNode::createLeafNode("node131", "that"),

TreeNode::createNonLeafNode("node132", array(

TreeNode::createLeafNode("node1321", "the"),

TreeNode::createLeafNode("node1322", "sea"),

TreeNode::createLeafNode("node1323", "is"),

TreeNode::createLeafNode("node1324", "dangerous"),

)),

)),

)),

TreeNode::createNonLeafNode("node2", array(

TreeNode::createNonLeafNode("node21", array(

TreeNode::createNonLeafNode("node211", array(

TreeNode::createNonLeafNode("node2111", array(

TreeNode::createNonLeafNode("node21111", array(

TreeNode::createLeafNode("node211111", "and"),

TreeNode::createNonLeafNode("node211112", array(

TreeNode::createNonLeafNode("node2111121", array(

TreeNode::createLeafNode("node21111211", "the"),

)),

)),

TreeNode::createLeafNode("node211113", "storm"),

TreeNode::createNonLeafNode("node211114", array(

TreeNode::createLeafNode("node2111141", "terrible,"),

TreeNode::createLeafNode("node2111142", "but"),

)),

TreeNode::createLeafNode("node211115", "they"),

)),

)),

)),

TreeNode::createNonLeafNode("node212", array(

TreeNode::createLeafNode("node2121", "have"),

TreeNode::createLeafNode("node2122", "never"),

TreeNode::createLeafNode("node2123", "found"),

TreeNode::createLeafNode("node2124", "these"),

)),

)),

)),

TreeNode::createNonLeafNode("node3", array(

TreeNode::createNonLeafNode("node31", array(

TreeNode::createLeafNode("node311", "dangers"),

)),

TreeNode::createLeafNode("node32", "sufficient"),

TreeNode::createLeafNode("node33", "reason"),

TreeNode::createLeafNode("node34", "for"),

TreeNode::createNonLeafNode("node35", array(

TreeNode::createLeafNode("node351", "remaining"),

)),

TreeNode::createLeafNode("node36", "ashore..."),

)),

TreeNode::createNonLeafNode("node4", array(

TreeNode::createLeafNode("node41", "or"),

TreeNode::createLeafNode("node42", "so"),

TreeNode::createLeafNode("node43", "goes"),

)),

TreeNode::createLeafNode("node5", "the"),

TreeNode::createLeafNode("node6", "saying."),

));

}