A TREE REPRESENTATION IN APL

by Seth R. Alpert

This paper describes a scheme for representing trees as integer vectors and a set of functions for processing these trees. The technique evolved during the programming of a large management information system for one of our time sharing customers. In this system, the tree structure represented the organizational hierarchy associated with the firm's many subsidiaries, areas, regions, and divisions. In practice, this tree contained approximately 1500 nodes and 9 levels. The APL representation and functions described here were able to handle all system requirements conveniently and efficiently. For publication purposes, some of the functions have been modified to improve their readability.

Before discussing this representation scheme, it is useful to review some terminology. A directed graph, or digraph, G consists of a set V of nodes and a set E of distinct ordered pairs of elements of V called edges. An edge (u,v) of a directed graph has initial node u and terminal node v. Any sequence of edges of a digraph such that the terminal node of any edge in the sequence is the initial node of the next edge (if any) appearing in the sequence is called a path of the graph. A path originates at the initial node of its first edge and terminates at the terminal node of its last edge. A path with the same initial and terminal node is called a cycle. The indegree of any node v is the number of edges for which v is the terminal node; the outdegree is the number of edges for which v is the initial node.

A directed tree is a directed graph that has no cycles and has one node of indegree 0 called the root and all other nodes of indegree 1. A node of a directed tree that has outdegree 0 is called a terminal node, or a leaf. The level of any node is the number of edges in its path from the root. Thus, the level of the root is 0. If there is an edge with initial node u and terminal node v, then u is the father of v, and v is a son of u. Similarly, if there is a path with initial node u and terminal node v, then v is a descendant of its ancestor u. The path descends from u to v and ascends from v to u. Note that (perhaps perversely) descending means going from lower to higher level numbers, with the converse true for ascending. Finally, a forest is a collection of disjoint trees.

The following simple scheme may be used to represent any forest, although the discussion will be in terms of trees. The nodes of the tree are represented by the integers ιN, where N is the number of nodes. The tree itself is represented by an integer vector T, where $T[I]$ is the father of node I; and if K is a root node, then $T[K]$ is 0.

For example, consider the tree represented in Figure 1.

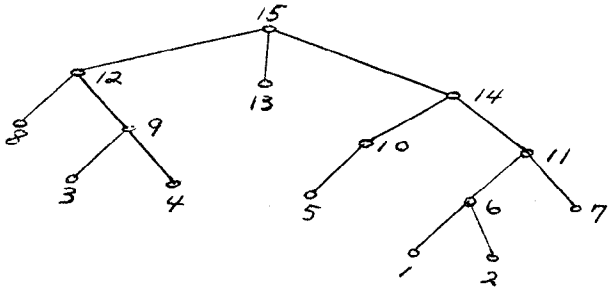

Figure 1.

Under the present scheme, the APL representation for this tree is

```
T←6 6 9 9 10 11 11 12 12 14 14 15 15 15 0
```

The simplicity of this idea may bely its power. The most common tree operations, namely, ascending and descending, are accomplished by APL primitives. Thus, the father of node *I* is *T*[*I*] and the sons of node *I* are (*T*=*I*)/⍳⍴*T*. Using this and the fact that root nodes are represented by a zero, we may write a series of tree utilities to find fathers, sons, all descendants, and all ancestors:

```
    ∇ R←T FATHERS N
[1]  ⍝FOR DIRECTED TREE <T>, FIND FATHERS
[2]  ⍝OF NODES <N>.  RESULT IS EMPTY
[3]  ⍝IF <N> IS A ROOT NODE.
[4]   R←T[N] ◇ R←(×R)/R
    ∇
```

```
    ∇ R←T SONS N
[1]  ⍝FOR DIRECTED TREE <T>, FIND SONS OF
[2]  ⍝NODES <N>.  RESULT IS EMPTY IF ALL
[3]  ⍝NODES IN <N> ARE TERMINAL.
[4]   R←(T∈N)/⍳⍴T
    ∇
```

```
    ∇ R←T DESCENDANTS N
[1]  ⍝FOR DIRECTED TREE <T>, FIND THE
[2]  ⍝NODES OF THE ENTIRE SUBTREE
[3]  ⍝ATTACHED TO NODE <N>.
[4]   →0 IF 0=⍴R←T SONS N
[5]   R←R,T DESCENDANTS R
    ∇
```

```
    ∇ R←T ANCESTORS N  (parent nodes)
[1]  ⍝FOR DIRECTED TREE <T>, FIND NODES
[2]  ⍝ON PATH FROM NODE <N> TO THE ROOT.
[3]  ⍝ASSUMES THAT <N> IS A SINGLETON.
[4]   →0 IF 0=R←T FATHER N
[5]   R←R,T ANCESTORS N
    ∇
```

It may be of interest to know whether or not a node is terminal, and this is easily accomplished, as follows:

```
      ∇ R←T TERMINAL∆OR∆NOT N
[1]  ⍝FOR DIRECTED TREE <T>, RETURN 1 IF
[2]  ⍝N IS A TERMINAL NODE, 0 OTHERWISE.
[3]   R←~N∈T
    ∇
```

Then, finding all terminal nodes that are descendants of node *I* amounts to using this simple sequence:

```
R←T DESCENDANTS I

R←(T TERMINAL∆OR∆NOT R)/R
```

The reader may note that the above functions have the additional property that the nodes are treated as being in increasing numeric order on each level. We can, of course, choose to ignore this ordering. In applications requiring ordered trees, however, this becomes a happy by-product of the representation, provided that we follow the convention of assigning increasing node numbers on each level.

It is also worthwhile to briefly consider the two iterative functions introduced thus far -- *DESCENDANTS* and *ANCESTORS*. In each function one iteration corresponds to a single level of the tree. Thus, a tree with a large number of levels will force many iterations with a concomitant adverse effect on run times. In such a situation (one that I have not yet encountered in practice), the present scheme may prove to be unsuitable.

The function *DESCENDANTS* provides a means of traversing the tree, that is, a well-defined progression through all of its nodes. Specifically, for any node *I*, *T DESCENDANTS I* simply lists all nodes in the subtree with root node *I*, with the result being built up one level at a time from the top down, and within each level nodes are listed in increasing numeric order.

Many other types of tree traversal have been defined, and I would like to consider here a particular one that arose in actual practice. The following rule defines what is called the <u>preorder traversal</u> of a tree [1]: List the root node; process the subtrees in left-to-right order. For the tree given in Figure 1, a preorder traversal would yield the nodes in the following order:

15 12 8 9 3 4 13 14 5 11 6 1 2 7

If the tree represents an organizational hierarchy, then the preorder traversal of the tree provides the basis for an organization table. To see this, look at the preorder traversal of the above tree with node names written out, and each

name indented a number of spaces equal to five times the level of that node:

```
Fifteen
   Twelve
      Eight
      Nine
         Three
         Four
   Thirteen
   Fourteen
      Ten
         Five
      Eleven
         Six
            One
            Two
         Seven
```

The following function provides the data necessary to build an organization chart of this type.

```
    ∇ R←T PREORDER N
[1]  ⍝FOR DIRECTED TREE <T>, PERFORM
[2]  ⍝PREORDER TRAVERSAL OF THE SUBTREE
[3]  ⍝ATTACHED TO NODE <N>.  ASSUMES THAT
[4]  ⍝1=⍴N.  RETURNS VECTOR OF NODES IN
[5]  ⍝PREORDER TRAVERSAL ORDER.
[6]   →NEXTL IF 0≠⍴N ◇ R←⍳0 ◇ →0
[7]   →NEXTL IF ~∧/T TERMINAL∆OR∆NOT N ◇
       R←N ◇ →0
[8]   R←((1↑N),T PREORDER T SONS 1↑N),
       T PREORDER 1↓N
    ∇
```

The idea used here for counting levels is the same as that mentioned above -- level number corresponds to iteration number. That idea could easily be applied to produce functions that compute the level of any node or the set of all nodes on a given level.

Up to now, all the ideas we have discussed deal with a fixed tree structure. The tree representation in question proves to be well suited to modifications of the tree, as well. Let us consider three fundamental kinds of modifications of tree structure: deletion of a node, addition of a node, and change to existing structure. Deletion of a node is simple -- it means removing the node and all edges incident upon it. This may, of course, result in more trees in the forest, but that does not affect our ability to represent the result.

If one of a tree's N nodes is deleted and we wish to represent the new structure, then it will be necessary to renumber the nodes using the integers $\iota N-1$. The following utility does this.

```
    ∇ R←T DROPNODE N
[1]  ⍝EXCISE NODE <N> FROM DIRECTED TREE
[2]  ⍝<T>.  RESULT IS A FOREST WITH
[3]  ⍝ONE LESS NODE THAN <T> HAS.
[4]  ⍝ASSUMES THAT <N> IS A SINGLETON.
[5]   R←(⍴T)⍴1 ◇ R[N]←0 ◇ R←R/T
[6]  ⍝REMOVE EDGES FROM <N>.
[7]   R[(R=N/⍳⍴R]←0←¯1+R[(R>N)/⍳⍴R]
[8]  ⍝RENUMBER REMAINING NODES
[9]   R[(R>N)/⍳⍴R]
    ∇
```

Adding a node could be somewhat complex if we tried to add all its new edges at the same time. Instead, let us restrict our attention to the edge that links the node to its father. For our purposes, then, adding a node amounts to catenating an integer to our tree vector, as follows:

```
    ∇ R←ADDNODE T
[1]   'NEW NODE WILL BE NUMBER: ',⍕1+⍴T
[2]   'NEW NODE REPORTS TO WHICH NODE?'
[3]   R←T,□
[4]   'DONE'
    ∇
```

Finally, we consider changing the existing relationships among nodes. Any set of such changes can be obtained by a sequence of changes at a single node. Of such changes, there are two basic types -- changing the node's father and changing its sons. Each is readily accomplished within the present framework.

```
    ∇ R←T CHANGEFATHER N
[1]  ⍝FOR DIRECTED TREE <T>, CHANGE FATHER OF
[2]  ⍝NODE 1↑N TO 1↓N.  ASSUME THAT 2=⍴N.
[3]   R←T ◇ R[1↑N]←1↓N
    ∇
```

```
    ∇ R←T CHANGESONS N
[1]  ⍝FOR DIRECTED TREE <T>, CHANGE SONS
[2]  ⍝OF NODE 1↑N TO 1↓N.  OLD SONS
[3]  ⍝ARE MADE SONS OF <1↑N>'S FATHER.
[4]  ⍝WARNING: MAY RESULT IN NONTREE.
[5]   R←T ◇ R[(T=1↑N)/⍳⍴T]←T[1↑N]
[6]   R[1↓N]←1↑N
    ∇
```

The reader will note that a drastic rearrangement in the tree, such as that which might result from changing a node's father, is accomplished by altering one element in the vector representing the tree.

These utilities point out the need for one additional utility, for there is nothing to guarantee that the result of *CHANGEFATHER*, for example, still represents a tree. The question becomes how do we know if a given integer vector represents a tree. The algorithm below answers that question.

```
    ∇ R←TESTTREE T;A;B
[1]  ⍝RETURN 1 IF <T> IS A VALID REPRESENTATION
[2]  ⍝OF A DIRECTED TREE; ELSE RETURN 0.
[3]   R←0
[4]   →(1≠⍴⍴T)⍴0 ⍝IS T A VECTOR?
[5]   →(0≠1↑0⍴T)⍴0 ⍝IS T NUMERIC?
[6]   →(∨/(0>T),T>⍴T)⍴0 ⍝IS 0≤T≤⍴T?
[7]   →(∨/T≠⌈T)⍴0 ⍝IS T INTEGER?
[8]  ⍝TEST FOR CYCLES AND REACHABILITY FROM ROOTS.
[9]   A←(T=0)/⍳⍴T ◇ B←A
[10] ⍝GO TO L2 IF ALL NODES IN B ARE TERMINAL.
[11] L1:B←T SONS B ◇ (→0=⍴B)⍴L2
[12]  →(∨/B∈A)⍴0 ⍝HAVE WE RETURNED TO ANY PRIOR NODES?
[13]  A←A,B ◇ →L1 ⍝CONTINUE DESCENDING.
[14] ⍝HAVE ALL NODES BEEN REACHED FROM THE ROOTS?
[15] L2:→((⍴T)≠⍴A)⍴0
[16]  R←1 ⍝T IS A TREE.
    ∇
```

Reference

1. Knuth, D. E. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. 2nd Ed. Addison-Wesley, Reading, Mass., 1973.

Seth R. Alpert
Scientific Time Sharing Corporation
747 Third Avenue
New York, New York 10017