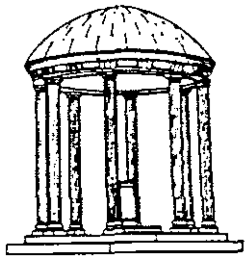
# A Node-Positioning Algorithm for General Trees

*TR89-034*

*September, 1989*

*John Q. Walker II*



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

## Abstract

Drawing a tree consists of two stages: determining the position of each node, and actually rendering the individuals nodes and interconnecting branches. The algorithm described in this paper is concerned with the first stage: given a list of nodes, an indication of the hierarchical relationship among them, and their shape and size, where should each node be positioned for optimal aesthetic effect?

This algorithm determines the positions of the nodes for any arbitrary general tree. It is the most desirable positioning with respect to certain widely-accepted heuristics. The positioning, specified in x, y coordinates, minimizes the width of the tree. In a general tree, there is no limit on the number of offspring per node; this contrasts with binary and ternary trees, for example, which are trees with a limit of 2 and 3 offspring per node. This algorithm operates in time O(N), where N is the number of nodes in the tree.

Previously, most tree drawings have been positioned by the sure hand of a human graphic designer. Many computer-generated positionings have been either trivial or contained irregularities. Earlier work by Wetherell and Shannon (1979) and Tilford (1981), upon which this algorithm builds, failed to correctly position the interior nodes of some trees. Radack (1988), also building on Tilford's work, has solved this same problem with a different method which makes four passes. The algorithm presented here correctly positions a tree's nodes using only two passes. It also handles several practical considerations: alternate orientations of the tree, variable node sizes, and out-of-bounds conditions.

## Keywords

Aesthetics, computer graphics, drawing methods, tree drawing, tree structures

# Contents

**Introduction** . . . . . 1
What Is A General Tree? . . . . . 1
Aesthetic Rules . . . . . 2
Application Areas . . . . . 3

**How the Algorithm Works** . . . . . 4

**The Algorithm** . . . . . 6

**An Example** . . . . . 17
Nodes Visited in the First Traversal . . . . . 17
Nodes Visited in the Second Traversal . . . . . 20

**Changing the Orientation of the Root** . . . . . 21

**Previous Work** . . . . . 24

**Acknowledgements** . . . . . 27

**References** . . . . . 28

**An Example Underlying Tree Structure** . . . . . 29

# Introduction

This algorithm addresses the problem of drawing tree structures. Trees are a common method of representing a hierarchically-organized structure. In computer science, trees are used in such areas as searching, compiling, and database systems; in non-computer applications, they are commonly used to construct organizational charts or to illustrate biological classifications. Visual displays of trees show hierarchical relationships clearly; they are often more useful than listings of trees in which hierarchical structure is obscured by a linear arrangement of the information.[11]

A key task in tree drawing is deciding where to place each node on the display or output page. This task is accomplished by a node-positioning algorithm that calculates the x and y coordinates for every node of the tree. A rendering routine can then use these coordinates to draw the tree. A node-positioning algorithm must address two key issues. First, the resulting drawing should be aesthetically pleasing. Second, the positioning algorithm should make every effort to conserve space. Each of these two issues can be handled straightforwardly by itself, but taking them together poses some challenges.

Several algorithms for the positioning of general trees have been published; in the works of Sweet[10] and Tilford[11], however, the authors describe anomalies with their algorithms that can cause drawings with less-than-desirable results. The algorithm presented here corrects the deficiencies in these algorithms and produces the most desirable positioning for all general trees it is asked to position. Radack[7] has published a node-positioning algorithm that uses a different solution technique, but which produces results identical to those presented here.

## What Is A General Tree?

This paper deals with rooted, directed trees, that is, trees with one root and hierarchical connections from the root to its offspring. No node may have more than one parent.

A general tree is a tree with no restriction on the number of offspring each node has. A general tree is also known as an *m-ary* tree, since each node can have m offspring (where m is 0 or more). The common terms *binary tree* and *ternary tree* are restrictive examples of the general case; binary and ternary trees allow no more than 2 and 3 offspring per node, respectively. As a class, binary trees, in particular, differ from general trees in the following respect:

> An offspring of a node in a binary tree must be either the left offspring or the right offspring. It is common practice in drawing binary trees to preserve this left-right distinction. Thus, a single offspring is placed under its parent node either to the left or right of its parent's position.

This left-right distinction does not apply in a general tree; if a node has a single offspring, the offspring is placed directly below its parent.

This algorithm positions a binary tree by ignoring the distinction above. That is, it does not preserve left or right positioning of the offspring under the parent; if a node has exactly one offspring, it is positioned directly below its parent. Supowit and Reingold[9] noted that it is **NP**-hard to optimally position minimum-width binary trees (while adhering to the distinction above) to within a factor of less than about four percent.

## Aesthetic Rules

In their paper, Wetherell and Shannon[13] first described a set of aesthetic rules against which a good positioning algorithm must be judged. Tilford[11] and Supowit and Reingold[9] have expanded that list in an effort to produce better algorithms.

***Tidy*** drawings of trees occupy as little space as possible while satisfying certain aesthetics:

1. Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.[13]

   In parse trees, one might want all leaves to lie on one horizontal line; for that application Aesthetic 1 is not desirable. In this case, though, the width of the placement is fixed and so the minimum width placement problem for such parse trees is not interesting. We therefore restrict our attention to the wide class of applications for which Aesthetic 1 is desirable.[9]

2. A parent should be centered over its offspring.[13]

3. A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree. In some applications, one wishes to examine large trees to find repeated patterns; the search for patterns is facilitated by having isomorphic subtrees drawn isomorphically.[8]

   This implies that small subtrees should not appear arbitrarily positioned among larger subtrees.

   a. Small, interior subtrees should be spaced out evenly among larger subtrees (where the larger subtrees are adjacent at one or more levels).

   b. Small subtrees at the far left or far right should be adjacent to larger subtrees.

The algorithm described in this paper satisfies these aesthetic rules.

## Application Areas

In the past, general trees displayed on a computer screen or in print have had one of the following characteristics:

1. they were positioned by hand by a graphic artist,
2. they were small, trivial, or special-case trees, able to be positioned by one of the existing algorithms, or
3. they had areas of irregularity within them where the algorithmic positioning was not aesthetically desirable.

With this algorithm, a computer can reliably generate tree drawings equivalent to those done by a skilled human. Below are some of the applications that often use tree-drawings.

- Drawings of B-trees and 2-3 trees
- Structure editors that draw trees
- Flow charts without loops
- Visual LISP editors
- Parse trees
- Decision trees
- Hierarchical database models
- Hierarchically-organized file systems (for example, directories, sub-directories, and files)
- Depth-first spanning trees (graph theory)
- Organizational charts
- Table of contents in printed matter
- Biological classification

## How the Algorithm Works

This algorithm initially assumes the common practice among computer scientists of drawing trees with the root at the top of the drawing.[2] Node-positioning algorithms are concerned only with determining the x-coordinates of the nodes; the y-coordinate of a node can easily be determined from its level in the tree, due to Aesthetic 1 and the natural convention of a uniform vertical separation between consecutive levels. "Changing the Orientation of the Root" on page 21 presents a variation of the algorithm for altering the relationship of the x- and y-coordinates.

This algorithm utilizes two concepts developed in previous positioning algorithms. First is the concept of building subtrees as rigid units. When a node is moved, all of its descendants (if it has any) are also moved--the entire subtree being thus treated as a rigid unit. A general tree is positioned by building it up recursively from its leaves toward its root.

Second is the concept of using two fields for the positioning of each node. These two fields are:

- a preliminary x-coordinate, and
- a modifier field.

Two tree traversals are used to produce the final x-coordinate of a node. The first traversal assigns the preliminary x-coordinate and modifier fields for each node; the second traversal computes the final x-coordinate of each node by summing the node's preliminary x-coordinate with the modifier fields of all of its ancestors. This allows the simple moving of a large subtree and allows the algorithm to operate in time O(N). For example, to move a subtree 4 units to the right, increment both the preliminary x-coordinate and the modifier field of the subtree's root by 4. As another example, the modifier field associated with the apex node of the tree is used in determining the final position of all of its descendants. (The term *apex node* is used here to distinguish the root of the entire tree from the roots of individual internal subtrees.)

The first tree traversal is a postorder traversal, positioning the smallest subtrees (the leaves) first and recursively proceeding from left to right to build up the position of larger and larger subtrees. Sibling nodes are always separated from one another by at least a predefined minimal distance (the *sibling separation*); adjacent subtrees are separated by at least a predefined *subtree separation*. Subtrees of a node are formed independently and placed as close together as these separation values allow.

As the tree walk moves from the leaves to the apex, it combines smaller subtrees and their root to form a larger subtree. For a given node, its subtrees are positioned one-by-one, moving from left to right. Imagine that its newest subtree has been drawn and cut out of paper along its contour. Superimpose the new subtree atop its neighbor to the left, and move them apart until no two points are touching. Initially their roots are separated by the sibling separation value; then at the next lower level, they are pushed apart until the subtree separation value is established between the adjacent subtrees at the lower level. This process continues at successively lower levels until we get to the bottom of the shorter subtree. Note that the new subtree being placed may not always bump against a descendant of its nearest sibling to the left; siblings much farther to the left, but with many offspring, may cause the new subtree to be pushed to the right. At some levels no movement may be necessary; but at no level are the subtrees moved closer together. When this process is complete for all of the offspring of a node, the node is centered over its leftmost and rightmost offspring.

When pushing a new, large subtree farther and farther to the right, a gap may open between the large subtree and smaller subtrees that had been previously positioned correctly, but now appear to be bunched on the left with an empty area to their right. This produces an undesirable appearance; this characteristic of *left-to-right gluing* was the failing of the algorithms by Sweet, Wetherell and Shannon, and Tilford.

The algorithm presented here produces evenly distributed, proportional spacing among subtrees. When moving a large subtree to the right, the distance it is moved is also apportioned to smaller, interior subtrees, satisfying Aesthetic 3. The moving of these subtrees is accomplished as above--by adding the proportional values to the preliminary x-coordinate and modifier fields of the roots of the small interior subtrees. For example, if three small subtrees are bunched at the left because a new large subtree has been positioned to the right, the first small subtree to shifted right by ¼ of the gap, the second small subtree is shifted right by ½ of the gap, and the third small subtree is shifted right by ¾ of the gap.

The second tree traversal, a preorder traversal, determines the final x-coordinate for each node. It starts at the apex node of the tree, summing each node's x-coordinate value with the combined sum of the modifier fields of its ancestors. It also adds a value that guarantees centering of the display with respect to the position of the apex node of the drawing.

# The Algorithm

Since the algorithm operates by making two recursive walks of the tree, several variables are taken to be *global* for the sake of runtime efficiency. These variables are described below, alphabetically. All other variables are local to their respective procedures and functions.

| **Variable** | **Description** |
|---|---|
| *LevelZeroPtr* | The algorithm maintains a list of the previous node at each level, that is, the adjacent neighbor to the left. LevelZeroPtr is a pointer to the first entry in this list. |
| *xTopAdjustment* | A fixed distance used in the final walk of the tree to determine the absolute x-coordinate of a node with respect to the apex node of the tree. |
| *yTopAdjustment* | A fixed distance used in the final walk of the tree to determine the absolute y-coordinate of a node with respect to the apex node of the tree. |

The following global values must be set before the algorithm is called; they are not changed during the algorithm. They can be coded as constants.

| **Constant** | **Description** |
|---|---|
| *LevelSeparation* | The fixed distance between adjacent levels of the tree. Used in determining the y-coordinate of a node being positioned. |
| *MaxDepth* | The maximum number of levels in the tree to be positioned. If all levels are to be positioned, set this value to positive infinity (or an appropriate numerical value). |
| *SiblingSeparation* | The minimum distance between adjacent siblings of the tree. |
| *SubtreeSeparation* | The minimum distance between adjacent subtrees of a tree. For proper aesthetics, this value is normally somewhat larger than SiblingSeparation. |

The algorithm is invoked by calling function POSITIONTREE, passing it a pointer to the apex node of the tree. If the tree is too wide or too tall to be positioned within the coordinate system being used, POSITIONTREE returns the boolean FALSE; otherwise it returns TRUE.

For each node, the algorithm uses nine different functions. These might be stored in the memory allocated for each node, or they might be calculated for each node, depending on the internal structure of your application.

| Function | Description |
|---|---|
| *PARENT(Node)* | The current node's hierarchical parent |
| *FIRSTCHILD(Node)* | The current node's leftmost offspring |
| *LEFTSIBLING(Node)* | The current node's closest sibling node on the left |
| *RIGHTSIBLING(Node)* | The current node's closest sibling node on the right |
| *XCOORD(Node)* | The current node's x-coordinate |
| *YCOORD(Node)* | The current node's y-coordinate |
| *PRELIM(Node)* | The current node's preliminary x-coordinate |
| *MODIFIER(Node)* | The current node's modifier value |
| *LEFTNEIGHBOR(Node)* | The current node's nearest neighbor to the left, at the same level |

Upon entry to POSITIONTREE, the first four functions--the hierarchical relationships--are required for each node. Also, XCOORD and YCOORD of the apex node are required. Upon its successful completion, the algorithm sets the XCOORD and YCOORD values for each node in the tree.

```
function POSITIONTREE (Node): BOOLEAN;
begin
    if Node ≠ ϕ then
        begin
            (* Initialize the list of previous nodes at each level.   *)
            INITPREVNODELIST;

            (* Do the preliminary positioning with a postorder walk.  *)
            FIRSTWALK(Node, 0);

            (* Determine how to adjust all the nodes with respect to  *)
            (* the location of the root.                              *)
            xTopAdjustment ← XCOORD(Node) - PRELIM(Node);
            yTopAdjustment ← YCOORD(Node);

            (* Do the final positioning with a preorder walk.         *)
            return SECONDWALK(Node, 0, 0);
        end;
    else
        (* Trivial: return TRUE if a null pointer was passed.         *)
        return TRUE;
end.
```

Figure 1. Function POSITIONTREE. This function determines the coordinates for each node in a tree. A pointer to the apex node of the tree is passed as input. This assumes that the x and y coordinates of the apex node are set as desired, since the tree underneath it will be positioned with respect to those coordinates. Returns TRUE if no errors, otherwise returns FALSE.

```
procedure FIRSTWALK (Node, Level):
begin
    (* Set the pointer to the previous node at this level.        *)
    LEFTNEIGHBOR(Node) ← GETPREVNODEATLEVEL(Level);
    SETPREVNODEATLEVEL(Level, Node);    (* This is now the previous.  *)
    MODIFIER(Node) ← 0;      (* Set the default modifier value.       *)
    if (ISLEAF(Node) or Level = MaxDepth) then
        begin
            if HASLEFTSIBLING(Node) then
                (* Determine the preliminary x-coordinate based on:   *)
                (*  the preliminary x-coordinate of the left sibling, *)
                (*  the separation between sibling nodes, and         *)
                (*  the mean size of left sibling and current node.   *)
                PRELIM(Node) ← PRELIM(LEFTSIBLING(Node)) +
                               SiblingSeparation +
                               MEANNODESIZE(LEFTSIBLING(Node), Node);
            else
                (* No sibling on the left to worry about.             *)
                PRELIM(Node) ← 0;
        end;
    else
        (* This Node is not a leaf, so call this procedure            *)
        (* recursively for each of its offspring.                     *)
        begin
            Leftmost ← Rightmost ← FIRSTCHILD(Node);
            FIRSTWALK(Leftmost, Level + 1);
            while HASRIGHTSIBLING(Rightmost) do
                begin
                    Rightmost ← RIGHTSIBLING(Rightmost);
                    FIRSTWALK(Rightmost, Level + 1);
                end;
            Midpoint ← (PRELIM(Leftmost) + PRELIM(Rightmost)) / 2;
            if HASLEFTSIBLING(Node) then
                begin
                    PRELIM(Node) ← PRELIM(LEFTSIBLING(Node)) +
                                   SiblingSeparation +
                                   MEANNODESIZE(LEFTSIBLING(Node), Node);
                    MODIFIER(Node) ← PRELIM(Node) - Midpoint;
                    APPORTION(Node, Level);
                end;
            else
                PRELIM(Node) ← Midpoint;
        end;
end.
```

Figure 2. Procedure FIRSTWALK. In this first postorder walk, every node of the tree is assigned a preliminary x-coordinate (held in field PRELIM(Node)). In addition, internal nodes are given modifiers, which will be used to move their offspring to the right (held in field MODIFIER(Node)).

```
function SECONDWALK (Node, Level, Modsum): BOOLEAN;
begin
    if Level ≤ MaxDepth then
        begin
            xTemp ← xTopAdjustment + PRELIM(Node) + Modsum;
            yTemp ← yTopAdjustment + (Level * LevelSeparation);
            (* Check to see that xTemp and yTemp are of the proper    *)
            (* size for your application.                             *)
            if CHECKEXTENTSRANGE(xTemp, yTemp) then
                begin
                    XCOORD(Node) ← xTemp;
                    YCOORD(Node) ← yTemp;
                    if HASCHILD(Node) then
                        (* Apply the Modifier value for this node to  *)
                        (* all its offspring.                         *)
                        Result ← SECONDWALK(FIRSTCHILD(Node),
                                            Level + 1,
                                            Modsum + MODIFIER(Node));
                    if (Result = TRUE and
                        HASRIGHTSIBLING(Node)) then
                        Result ← SECONDWALK(RIGHTSIBLING(Node),
                                            Level + 1,
                                            Modsum);
                end;
            else
                (* Continuing would put the tree outside of the       *)
                (* drawable extents range.                            *)
                Result ← FALSE;
        end;
    else
        (* We are at a level deeper than what we want to draw.        *)
        Result ← TRUE;

    return Result;
end.
```

Figure 3. Function SECONDWALK. During a second preorder walk, each node is given a final x-coordinate by summing its preliminary x-coordinate and the modifiers of all the node's ancestors. The y-coordinate depends on the height of the tree. If the actual position of an interior node is right of its preliminary place, the subtree rooted at the node must be moved right to center the sons around the father. Rather than immediately readjust all the nodes in the subtree, each node remembers the distance to the provisional place in a modifier field (MODIFIER(Node)). In this second pass down the tree, modifiers are accumulated and applied to every node. Returns TRUE if no errors, otherwise returns FALSE.

```
procedure APPORTION (Node, Level):
begin
    Leftmost ← FIRSTCHILD(Node);
    Neighbor ← LEFTNEIGHBOR(Leftmost);
    CompareDepth ← 1;
    DepthToStop ← MaxDepth - Level;

    while (Leftmost ≠ ϕ and
           Neighbor ≠ ϕ and
           CompareDepth ≤ DepthToStop) do
        begin
            (* Compute the location of Leftmost and where it should   *)
            (* be with respect to Neighbor.                           *)
            LeftModsum ← 0;
            RightModsum ← 0;
            AncestorLeftmost ← Leftmost;
            AncestorNeighbor ← Neighbor;
            for i ← 0
              until CompareDepth do
                begin
                    AncestorLeftmost ← PARENT(AncestorLeftmost);
                    AncestorNeighbor ← PARENT(AncestorNeighbor);
                    RightModsum ← RightModsum +
                                  MODIFIER(AncestorLeftmost);
                    LeftModsum  ← LeftModsum  +
                                  MODIFIER(AncestorNeighbor);
                end;

            (* Find the MoveDistance, and apply it to Node's subtree. *)
            (* Add appropriate portions to smaller interior subtrees. *)
            MoveDistance ← (PRELIM(Neighbor) +
                             LeftModsum +
                             SubtreeSeparation +
                             MEANNODESIZE(Leftmost, Neighbor)) -
                            (PRELIM(Leftmost) + RightModsum));

            if MoveDistance > 0 then
                begin
                    (* Count interior sibling subtrees in LeftSiblings*)
                    TempPtr ← Node;
                    LeftSiblings ← 0;
                    while (TempPtr ≠ ϕ and
                           TempPtr ≠ AncestorNeighbor) do
                        begin
                            LeftSiblings ← LeftSiblings + 1;
                            TempPtr ← LEFTSIBLING(TempPtr);
                        end;
```

Figure 4. Procedure APPORTION, part 1 of 2

```
                if TempPtr ≠ ϕ then
                    (* Apply portions to appropriate leftsibling  *)
                    (* subtrees.                                   *)
                    begin
                        Portion ← MoveDistance / LeftSiblings;
                        TempPtr ← Node;
                        while
                            TempPtr = AncestorNeighbor do
                            begin
                                PRELIM(TempPtr) ←
                                    PRELIM(TempPtr) + MoveDistance;
                                MODIFIER(TempPtr) ←
                                    MODIFIER(TempPtr) + MoveDistance;
                                MoveDistance ← MoveDistance -
                                    Portion;
                                TempPtr ← LEFTSIBLING(TempPtr);
                            end;
                    end;
                else
                    (* Don't need to move anything--it needs to   *)
                    (* be done by an ancestor because             *)
                    (* AncestorNeighbor and AncestorLeftmost are  *)
                    (* not siblings of each other.                *)
                    return;
            end; (* of MoveDistance > 0 *)

        (* Determine the leftmost descendant of Node at the next  *)
        (* lower level to compare its positioning against that of *)
        (* its Neighbor.                                          *)

        CompareDepth ← CompareDepth + 1;
        if ISLEAF(Leftmost) then
            Leftmost ← GETLEFTMOST(Node, 0, CompareDepth);
        else
            Leftmost ← FIRSTCHILD(Leftmost);

    end; (* of the while *)
end.
```

Figure 5. Procedure APPORTION, part 2 of 2. This procedure cleans up the positioning of small sibling subtrees, thus fixing the "left-to-right gluing" problem evident in earlier algorithms. When moving a new subtree farther and farther to the right, gaps may open up among smaller subtrees that were previously sandwiched between larger subtrees. Thus, when moving the new, larger subtree to the right, the distance it is moved is also apportioned to smaller, interior subtrees, creating a pleasing aesthetic placement.

```
function GETLEFTMOST (Node, Level, Depth): NODE;
begin
    if Level ≥ Depth then
        return Node;
    else if ISLEAF(Node) then
        return ϕ;
    else begin
        Rightmost ← FIRSTCHILD(Node);
        Leftmost ← GETLEFTMOST(Rightmost, Level + 1, Depth);
        (* Do a postorder walk of the subtree below Node.              *)
        while (Leftmost = ϕ and
               HASRIGHTSIBLING(Rightmost)) do
            begin
               Rightmost ← RIGHTSIBLING(Rightmost);
               Leftmost ← GETLEFTMOST(Rightmost, Level + 1, Depth);
            end;
        return Leftmost;
    end;
end.
```

Figure 6. Function GETLEFTMOST. This function returns the leftmost descendant of a node at a given Depth. This is implemented using a postorder walk of the subtree under Node, down to the level of Depth. Level here is not the absolute tree level used in the two main tree walks; it refers to the level below the node whose leftmost descendant is being found.

```
function MEANNODESIZE (LeftNode, RightNode): REAL;
begin
    NodeSize ← 0;

    if LeftNode ≠ ϕ then
        NodeSize ← NodeSize + RIGHTSIZE(LeftNode);
    if RightNode ≠ ϕ then
        NodeSize ← NodeSize + LEFTSIZE(RightNode);
    return NodeSize;
end.
```

Figure 7. Function MEANNODESIZE. This function returns the mean size of the two passed nodes. It adds the size of the right half of lefthand node to the left half of righthand node. If all nodes are the same size, this is a trivial calculation.

```
function CHECKEXTENTSRANGE (xValue, yValue): BOOLEAN;
begin
    if (xValue is a valid value for the x-coordinate) and
       (yValue is a valid value for the y-coordinate) then
        return TRUE;
    else
        return FALSE;
end.
```

Figure 8. Function CHECKEXTENTSRANGE. This function verifies that the passed x- and y-coordinates are within the coordinate system being used for the drawing. For example, if the x-and y-coordinates must be 2-byte integers, this function could determine whether xValue and yValue are too large.

```
procedure INITPREVNODELIST:
begin
    (* Start with the node at level 0--the apex of the tree.        *)
    TempPtr ← LevelZeroPtr;
    while TempPtr ≠ ∅ do
        begin
            PREVNODE(TempPtr) ← ∅;
            TempPtr ← NEXTLEVEL(TempPtr);
        end;
end.
```

Figure 9. Initialize the list of previous nodes at each level. Three list-maintenance procedures, GETPREVNODEATLEVEL, SETPREVNODEATLEVEL, and INITPREVNODELIST, maintain a singly-linked list. Each entry in the list corresponds to the node previous to the current node at a given level (for example, element 2 in the list corresponds to the node to the left of the current node at level 2). If the maximum tree size is known beforehand, this list can be replaced with a fixed-size array, and these procedures become trivial.

Each list element contains two fields: PREVNODE--the previous node at this level, and NEXTLEVEL--a forward pointer to the next list element. The list is does not need to be cleaned up between calls to POSITIONTREE, for performance.

```
function GETPREVNODEATLEVEL (Level): NODE;
begin
    (* Start with the node at level 0--the apex of the tree.        *)
    TempPtr ← LevelZeroPtr;
    i ← 0;
    while TempPtr ≠ ϕ do
        begin
            if i = Level then
                return PREVNODE(TempPtr)
            TempPtr ← NEXTLEVEL(TempPtr);
            i ← i + 1;
        end;
    (* Otherwise, there was no node at the specific level.          *)
    return ϕ;
end.
```

Figure 10. Get the previous node at this level. See Figure 9.

```
procedure SETPREVNODEATLEVEL (Level, Node):
begin
    (* Start with the node at level 0--the apex of the tree.          *)
    TempPtr ← LevelZeroPtr;
    i ← 0;
    while TempPtr ≠ ϕ do
        begin
            if i = Level then
                begin
                    (* At this level, replace the existing list       *)
                    (* element with the passed-in node.               *)
                    PREVNODE(TempPtr) ← Node;
                    return;
                end;
            else if NEXTLEVEL(TempPtr) = ϕ then
                (* There isn't a list element yet at this level, so   *)
                (* add one.  The following instructions prepare the   *)
                (* list element at the next level, not at this one.   *)
                begin
                    NewNode ← ALLOCATE_A_NODE;
                    PREVNODE(NewNode) ← ϕ;
                    NEXTLEVEL(NewNode) ← ϕ;
                    NEXTLEVEL(TempPtr) ← NewNode;
                end;

            (* Prepare to move to the next level, to look again.      *)
            TempPtr ← NEXTLEVEL(TempPtr);
            i ← i + 1;
        end;

    (* Should only get here if LevelZeroPtr is nil.                   *)
    LevelZeroPtr ← ALLOCATE_A_NODE;
    PREVNODE(LevelZeroPtr) ← Node;
    NEXTLEVEL(LevelZeroPtr) ← ϕ;
end.
```

Figure 11. Set an element in the list. See Figure 9 on page 14. Function "ALLOCATE_A_NODE" (not shown here) requests a pointer to a block of memory, to be used to represent a node in the list.

## An Example

The operation of the algorithm during these two walks can be best illustrated with an example. At least three levels are needed to illustrate its operation, since a small subtree must be centered between larger sibling subtrees. The following figure is an example tree positioned by this algorithm. Its fifteen nodes have been lettered in the order that they are visited in the first postorder traversal. For this example, the mean size of each node is 2 units and the sibling separation and subtree separation values are the same: 4 units.

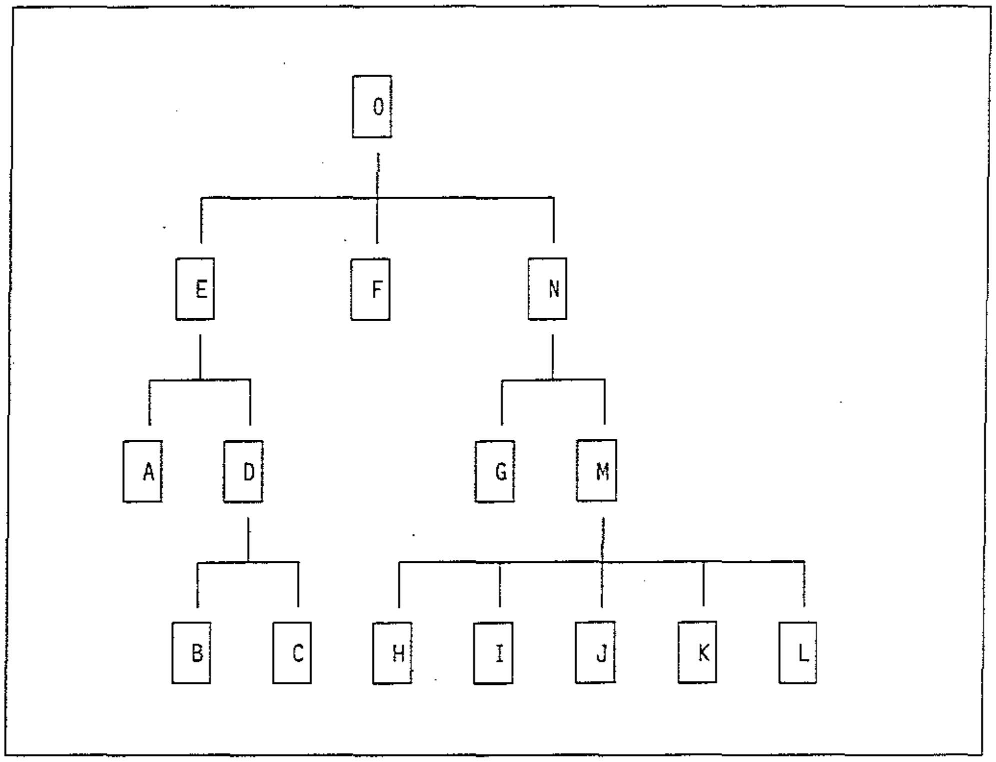


Figure 12. An example general tree, with 15 nodes

## Nodes Visited in the First Traversal

The nodes are visited in a postorder walk. Their preliminary x-coordinate value and modifier values are calculated in this traversal.

**Node** **Preliminary X-coordinate and Modifier**

**A** is a leaf with no left sibling.

```
PRELIM(A) = 0
MODIFIER(A) = 0
```

**B** is also a leaf with no left sibling.

```
PRELIM(B) = 0
MODIFIER(B) = 0
```

**C** is the right sibling of node B. It is separated from it by the sibling separation value plus the mean size of the two nodes.

```
PRELIM(C) = 0 + 4 + 2 = 6
MODIFIER(C) = 0
```

**D** is the parent of nodes B and C, and the right sibling of node A. It is separated from node A by the sibling separation value plus the mean size of the two nodes. Its modifier is set so that when it is applied to nodes B and C, they will appear centered underneath it. The modifier is determined by taking PRELIM(D) and subtracting the mean of the PRELIM(its mostly widely-separated offspring) values.

```
PRELIM(D) = 0 + 4 + 2 = 6
MODIFIER(D) = 6 - (0 + 6)/2 = 3
```

**E** is the parent of nodes A and D. It is centered over nodes A and D.

```
PRELIM(E) = (0 + 6)/2 = 3
MODIFIER(E) = 0
```

**F** is a right sibling of node E. It is separated from it by the sibling separation value plus the mean size of the two nodes. That would place it directly over node C. We can see now that node N's subtree will later be placed much further to the right, leaving the spacing between nodes E and F smaller, and hence different, than the spacing between nodes F and N. When node N is finally positioned, the position of node F will be adjusted. But for now,

```
PRELIM(F) = 3 + 4 + 2 = 9
MODIFIER(F) = 0
```

**G** is a leaf with no left sibling.

```
PRELIM(G) = 0
MODIFIER(G) = 0
```

**H** is a leaf with no left sibling.

```
PRELIM(H) = 0
MODIFIER(H) = 0
```

**I** is the right sibling of node H. It is separated from it by the sibling separation value plus the mean size of the two nodes.

```
PRELIM(I) = 0 + 4 + 2 = 6
MODIFIER(I) = 0
```

**J** is the right sibling of node I. As above, it is separated by the standard spacing from node I.

```
PRELIM(J) = 6 + 4 + 2 = 12
MODIFIER(J) = 0
```

**K** is the right sibling of node J.

```
PRELIM(K) = 12 + 4 + 2 = 18
MODIFIER(K) = 0
```

**L** is the right sibling of node K.

```
PRELIM(L) = 18 + 4 + 2 = 24
MODIFIER(L) = 0
```

**M** is the parent of nodes H through L, and the right sibling of node G. It is separated from node G by the sibling separation value plus the mean size of the two nodes. Its modifier is set so that when it is applied to nodes H through L, they will appear centered underneath it.

```
PRELIM(M) = 0 + 4 + 2 = 6
MODIFIER(M) = 6 - (0 + 24)/2 = -6
```

**N** is the parent of nodes G and M, and the right sibling of node F. It is first of all given its standard positioning to the right of node F, with a modifier that reflects the centering of its offspring beneath it.

```
PRELIM(N) = 9 + 4 + 2 = 15
MODIFIER(N) = 15 - (0 + 6)/2 = 12
```

Now we have to verify that node E's subtree and node N's subtree are properly separated.

Moving down one level, the leftmost descendant of node N, node G, currently has a positioning of 0 + 12 = 12 (PRELIM(G) plus the MODIFIER(N), its parent). The rightmost descendant of node E, node D is positioned at 6 + 0 = 6 (PRELIM(D) plus the MODIFIER(E), its parent). Their difference is 12 - 6 = 6, which is equal to the minimum separation (subtree separation plus mean node size), so N's subtree does not need to be moved, since there is no overlap at this level.

Moving down one more level, the leftmost descendant of node N is node H. It is positioned at 0 + -6 + 12 = 6 (PRELIM(H) plus MODIFIER(M) and MODIFIER(N)). The rightmost descendant of node E, node C, is positioned at 6 + 3 + 0 = 9 (PRELIM(C) plus MODIFIER(D) and MODIFIER(E)). Their difference is 6 - 9 = -3; it should be 6, the minimum subtree separation plus the mean node size. Thus node N and its subtree need to be moved to the right a distance of 6 - -3 = 9.

```
PRELIM(N) = 15 + 9 = 24
MODIFIER(N) = 12 + 9 = 21
```

This opens a gap of size 9 between sibling nodes E and N. This difference needs to be evenly distributed to all contained sibling nodes, and node F is the only one. Node F is moved to the right a distance of 9/2 = 4.5.

```
PRELIM(F) = 9 + 4.5 = 13.5
MODIFIER(F) = 0 + 4.5 = 4.5
```

**O** is the parent of nodes E, F, and N. It is positioned halfway between the position of nodes E and N.

```
PRELIM(O) = (3 + 24)/2 = 13.5
MODIFIER(O) = 0
```

## Nodes Visited in the Second Traversal

The nodes are all visited a second time, this time in a preorder traversal. Their final x-coordinates are determined by summing their preliminary x-coordinates with the modifier fields of all of their ancestors.

| **Node** | **Final X-coordinate (preliminary x-coordinate + modifiers of ancestors)** |
|---|---|
| **O** | 13.5 |
| **E** | 3 + 0 = 3 |
| **A** | 0 + 0 + 0 = 0 |
| **D** | 6 + 0 + 0 = 6 |
| **B** | 0 + 3 + 0 + 0 = 3 |
| **C** | 6 + 3 + 0 + 0 = 9 |
| **F** | 13.5 + 0 = 13.5 |
| **N** | 24 + 0 = 24 |
| **G** | 0 + 21 + 0 = 21 |
| **M** | 6 + 21 + 0 = 27 |
| **H** | 0 + -6 + 21 + 0 = 15 |
| **I** | 6 + -6 + 21 + 0 = 21 |
| **J** | 12 + -6 + 21 + 0 = 27 |
| **K** | 18 + -6 + 21 + 0 = 33 |
| **L** | 24 + -6 + 21 + 0 = 39 |

## Changing the Orientation of the Root

The algorithm illustrates tree positioning where the apex of the tree is at the top of the drawing. Some simple modifications allow other common positionings, such as where the root is on the left and the siblings are to its right. Four such orientations of the root can be readily identified; these will be the values taken by a new global constant, **RootOrientation**, to be set before the algorithm is called.

| | |
|---|---|
| **NORTH** | root is at the top, as shown in the preceding algorithm |
| **SOUTH** | root is at the bottom, its siblings are above it |
| **EAST** | root is at the left, its siblings are to its right |
| **WEST** | root is at the right, its siblings are to its left |

The ability to accommodate a change in orientation involves some minor changes to three functions: POSITIONTREE, SECONDWALK, and MEANNODESIZE. These changes are shown below with change bars, |.

```
function POSITIONTREE (Node): BOOLEAN;
begin
    if Node ≠ ∅ then
        begin
            (* Initialize the list of previous nodes at each level.   *)
            INITPREVNODELIST;

            (* Do the preliminary positioning with a postorder walk.  *)
            FIRSTWALK(Node, 0);

            (* Determine how to adjust all the nodes with respect to  *)
            (* the location of the root.                              *)
|           if RootOrientation = (NORTH or SOUTH) then
|               begin
                    xTopAdjustment ← XCOORD(Node) - PRELIM(Node);
                    yTopAdjustment ← YCOORD(Node);
|               end;
|           else if RootOrientation = (EAST or WEST) then
|               begin
|                   xTopAdjustment ← XCOORD(Node);
|                   yTopAdjustment ← YCOORD(Node) + PRELIM(Node);
|               end;
            .
            .
            .
```

Figure 13. Function POSITIONTREE. The final position of the tree's apex depends on the RootOrientation value.

```
function SECONDWALK (Node, Level, Modsum): BOOLEAN;
begin
    if Level ≤ MaxDepth then
        begin
            if RootOrientation = NORTH then
                begin
                    xTemp ← xTopAdjustment + (PRELIM(Node) + Modsum);
                    yTemp ← yTopAdjustment + (Level * LevelSeparation);
                end;
            else if RootOrientation = SOUTH then
                begin
                    xTemp ← xTopAdjustment + (PRELIM(Node) + Modsum);
                    yTemp ← yTopAdjustment - (Level * LevelSeparation);
                end;
            else if RootOrientation = EAST then
                begin
                    xTemp ← xTopAdjustment + (Level * LevelSeparation);
                    yTemp ← yTopAdjustment - (PRELIM(Node) + Modsum);
                end;
            else if RootOrientation = WEST then
                begin
                    xTemp ← xTopAdjustment - (Level * LevelSeparation);
                    yTemp ← yTopAdjustment - (PRELIM(Node) + Modsum);
                end;
            (* Check to see that xTemp and yTemp are of the proper    *)
            (* size for your application.                             *)
            if CHECKEXTENTSRANGE(xTemp, yTemp) then
            .
            .
            .
```

Figure 14. Function SECONDWALK. The values of xTemp and yTemp now depend on the RootOrientation value.

```
function MEANNODESIZE (LeftNode, RightNode): REAL;
begin
    NodeSize ← 0;

    if RootOrientation = (NORTH or SOUTH) then
        begin
            if LeftNode ≠ ϕ then
                NodeSize ← NodeSize + RIGHTSIZE(LeftNode);
            if RightNode ≠ ϕ then
                NodeSize ← NodeSize + LEFTSIZE(RightNode);
        end;
    else if RootOrientation = (EAST or WEST) then
        begin
            if LeftNode ≠ ϕ then
                NodeSize ← NodeSize + TOPSIZE(LeftNode);
            if RightNode ≠ ϕ then
                NodeSize ← NodeSize + BOTTOMSIZE(RightNode);
        end;

    return NodeSize;
end.
```

Figure 15. Function MEANNODESIZE. This function now returns the mean width of the two nodes if the RootOrientation is NORTH or SOUTH; if the RootOrientation is EAST or WEST, it returns the mean height of the two nodes.

## Previous Work

Early algorithms concentrated on drawing binary trees. Knuth[1] is generally credited with the first published algorithm for drawing binary trees. Its positioning of nodes in the drawing was sometimes rather crude; later algorithms tried to improve upon it.[8, 12, 13, 14]

The drawing of optimally-positioned general trees, because of their distinctions from binary trees (as noted above), is not an **NP**-hard problem. However, the problem was approached somewhat later than the problem of drawing binary trees. In the algorithms of Sweet[10] and Tilford,[11] the authors both note their algorithm's irregular behavior under certain conditions.

Sweet[10] published his algorithm as an appendix to his dissertation and noted the following:

> The principal shortcoming of the algorithm is somewhat difficult to demonstrate in a small tree. Figure B.3 is a somewhat contrived example that points out the problem. Consider the sons of node D: they are E, F, and G. Since the subtree E is "shallow," it is placed quite far to the left. The nodes F and G must be placed considerably farther to the right in order to make room for their subtrees. With larger, wider trees, a shallow subtree such as E can be so far from its brothers that it gets "lost." Shallow subtrees in son positions other than the first lead to uneven spacing of the sons. One could probably add a pass to the algorithm that, once having established the rightmost subtree of a node, then reformats the other subtrees for compactness and even spacing.

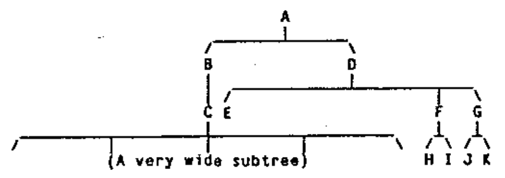


**Figure B.3. Example tree showing a shortcoming of the printing algorithm.**

Figure 16. Sweet's illustration showing deficiencies in his algorithm. From page 96 of Sweet.[10]

Tilford's Masters thesis, *Tree Drawing Algorithms*,[11] included algorithms for both binary and general tree drawings. He stated:

> Consider the three trees in Figure 5.1; in the first, subtrees were glued from left to right; in the second, from right to left. Of course, the most desirable positioning is given by the third drawing, which cannot be produced by Algorithm 5.1 [for drawing general trees] regardless of the

gluing order, because Algorithm 5.1 always puts a pair of subtrees as close together as possible.

The problem arises when a tree has the general shape shown in Figure 5.2, in which two non-adjacent subtrees are large, and the intervening ones are small enough that there is freedom in deciding where to place them. Although this will not always lead via Algorithm 5.1 to a violation of Aesthetic 4 [which states: "A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree."], it is clear that the small subtrees ought to be spaced out evenly rather than bunched up on one side or the other.

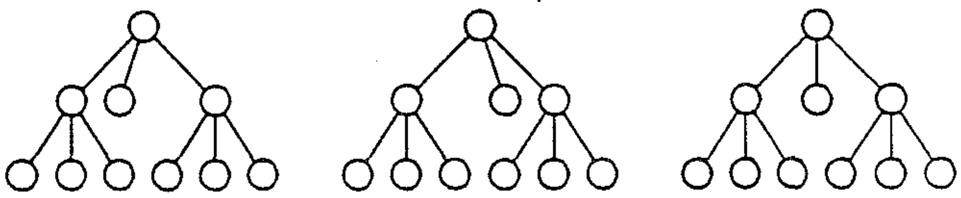


Figure 5.1. A small ternary tree positioned (a) by Algorithm 5.1, with left-to-right gluing; (b) by Algorithm 5.1, with right-to-left gluing; and (c) ideally.

Figure 17. Examples of centering vs. two types of gluing. From page 37 of Tilford.[11]

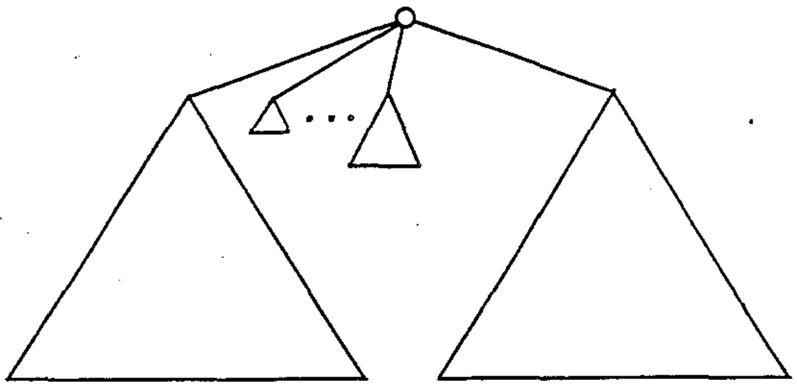


Figure 5.2. The general shape of a tree for which Algorithm 5.1 produces an unsatisfactory positioning.

Figure 18. Tilford's illustration showing deficiencies in his algorithm. From page 38 of Tilford.[11]

Radack[7] follows directly from Tilford's work, solving the left-to-right-gluing problem by essentially running his algorithm twice. The first time, the subtrees are agglutinated left to right; the second time, from right to left. A node is positioned at the average of the two assigned positions. Radack's algorithm positions the nodes in four passes.

Andy Poggio of SRI International, describing an unpublished algorithm used on their CCWS system[5] noted, "As their display is not a central aspect of our research, we developed a simple, expedient algorithm for that purpose." [6] It adheres to three aesthetic rules:

1. nodes at the same level are displayed on the same horizontal level,
2. all successors of a node are displayed below the node in an area bounded by the midpoint distance to adjacent nodes, and
3. the display root node is always centered at the top of the display area.

Trivial algorithms also exist for drawing general trees in an outline-like form, where the apex node is positioned to the left of the display and not centered above its offspring (for example, Petzold[4]).

The algorithms by Manning and Atallah[3] are examples of the class of algorithms that do node positioning with a different set of aesthetic rules; their primary goal was to highlight the symmetry inherent in hierarchical relationships.

## Acknowledgements

I appreciate the written correspondence I received from the experts on this topic (listed alphabetically): Brad A. Myers at the Computer Systems Research Institute in Toronto, Andy Poggio at SRI International, Edward Reingold at the University of Illinois, Bob Tarjan at AT&T Bell Laboratories, and C.S. Wetherell at AT&T Information Systems.

Thanks to Jane Munn, Jim Staton, and Dr. Bill Wright at IBM who reviewed an earlier version of this paper. Bob Gibson and John Broughton have also given me a lot of help.

## References

[1] Knuth, D.E. Optimum Binary Search Trees. *Acta Informatica* **1** (1971) 14-25.

[2] Knuth, D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1973.

[3] Manning, J.B. and M.J. Atallah. Fast Detection and Display of Symmetry in Trees. *Congressus Numerantium* **64** (November 1988) 159-169.

[4] Petzold, C. A Sight Better Than TREE. *PC Magazine* **4**, 22 (October 29, 1985) 199-206.

[5] Poggio, A.A., *et al*. CCWS: A Computer-Based, Multimedia Information System. *IEEE Computer* **18**, 10 (October 1985) 92-103.

[6] Poggio, A.A., personal communication, October 21, 1985.

[7] Radack, G.M. *Tidy Drawing of M-ary Trees*. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, Report CES-88-24, November 1988.

[8] Reingold, E.M. and J.S. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering* **SE-7**, 2 (March 1981) 223-228.

[9] Supowit, K.J. and E.M. Reingold. The complexity of drawing trees nicely. *Acta Informatica* **18**, 4 (January 1983) 377-392.

[10] Sweet, R.E. *Empirical estimates of program entropy*. Department of Computer Science, Stanford University, Stanford, CA, Report STAN-CS-78-698, November 1978. Also issued as Report CSL-78-3, Xerox PARC, Palo Alto, CA, September 1978.

[11] Tilford, J.S. *Tree drawing algorithms*. M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, Report UIUCDCS-R-81-1055, April 1981. Available as document UILU-ENG-81-1711 from the College of Engineering Document Center at the Univ. of Illinois.

[12] Vaucher, J.G. Pretty-Printing of Trees. *Software - Practice and Experience* **10** (1980) 553-561.

[13] Wetherell, C.S. and A. Shannon. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering* **SE-5**, 5 (September 1979) 514-520.

[14] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

## An Example Underlying Tree Structure

In this example, I use the internal tree notation described by Knuth [Reference 2, section 2.3.3] for a triply-linked tree. Each node consists of three pointers, FATHER, LSON, and RLINK, and its information in field INFO. FATHER points to the parent of the node. LSON points to the leftmost offspring of a node. RLINK points to the right sibling of a node. Thus, if node T is the root of a binary tree, the root of its left subtree is LSON(T) and the root of its right subtree is RLINK(LSON(T)).

This node structure is illustrated below, using the syntax of the C programming language.

```
struct position {
   /*-----------------------------------------------------------------------*/
   /* This structure contains the node positioning information.             */
   /* I've used floating point values here; use integer values, if          */
   /* necessary, but you may need to handle rounding errors.                */
   /*-----------------------------------------------------------------------*/
   float x_coordinate;        /* the value identified as XCOORD(Node)      */
   float y_coordinate;        /* the value identified as YCOORD(Node)      */
   float preliminary;         /* the value identified as PRELIM(Node)      */
   float modifier;            /* the value identified as MODIFIER(Node)    */
};

struct information {
   /*-----------------------------------------------------------------------*/
   /* This structure contains whatever node information your application    */
   /* requires. Here I show a fixed-length, 80-character label for the node. */
   /* If the sizes of the nodes differ, a node's width or height could be    */
   /* included here (see function MEANNODESIZE).                             */
   /*-----------------------------------------------------------------------*/
   char node_label[80];
};

struct node {
   struct node *father;        /* pointer to the parent of this node        */
   struct node *lson;          /* pointer to this node's leftmost offspring */
   struct node *rlink;         /* pointer to the right sibling of this node */
   struct node *left_neighbor; /* pointer to the adjacent node to the left  */
   struct position pos;        /* positioning values, as defined above      */
   struct information info;    /* node information, as defined above        */
};
```