SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(8), 817–827 (AUGUST 1993)

# Aesthetic Layout of Generalized Trees

ANTHONY BLOESCH
*Department of Computer Science, University of Queensland, St. Lucia, Queensland 4072, Australia*

SUMMARY

**Research on the aesthetic layout of trees has been largely concerned with the special case of binary trees with small constant-sized nodes. Yet, without otherwise requiring the power of general graph-layout algorithms, many layout problems involve *n*-ary trees with variable-sized nodes. This paper addresses the general issue of the aesthetic layout of such trees. Two algorithms are presented for the layout of generalized trees, and general issues, such as appropriate aesthetics, are discussed. The algorithms described are suitable for such tasks as the layout of class hierarchies, directory trees and tableau-style proofs.**

KEY WORDS Trees Graph layout Tree layout Aesthetic layout

## INTRODUCTION

Several algorithms[1–5] have been developed for the aesthetic layout of binary trees with small constant-sized nodes. The more general problem of the aesthetic layout of *n*-ary trees (i.e. trees where each node has 0–*n* children) with variable-sized nodes (generalized trees) has not been addressed adequately. A common *ad hoc* solution, where binary trees with variable-sized nodes are involved, is to use a small node algorithm but increase the space between layers in the tree (see Figure 1). Another common *ad hoc* solution, where generalized trees are involved, is to use an algorithm where sub-branches may never overlap horizontally (see Figure 2). Both of these algorithms are easy to implement, but they waste space and are unaesthetic.

In practice, many of the trees we may want to lay out automatically contain variable-sized nodes and have arities greater than two. For example, the proof technique known as the tableau method[6] produces trees with nodes containing lists of formulae, and thus nodes with varying width and height. Also, class hierarchies in object-oriented languages may be represented by *n*-ary trees with variable-width nodes containing the corresponding class's name. Increasingly, such trees are being used to represent information interactively on bit-mapped displays; however, the lack of suitable algorithms has led to the use of poor quality *ad hoc* algorithms. The requirement that tree-drawing algorithms support interactive use (even at the cost of increased code complexity or space requirements) suggests that algorithms proposed for the generalized task should be fast and their performance should degrade gracefully as problem size increases. Since such algorithms may be applied

0038–0644/93/080817–11$10.50


*Received 18 June 1991*
*Revised 21 January 1993*

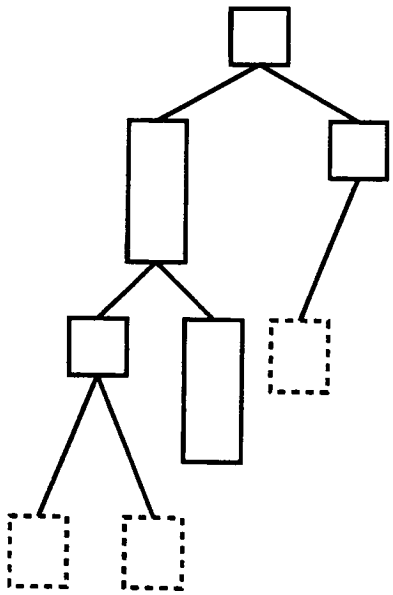

*Figure 1. An example of the tree layout generated by an algorithm that places all nodes at the same depth in the same horizontal layer. The algorithm has located the dashed boxes at unaesthetic positions*

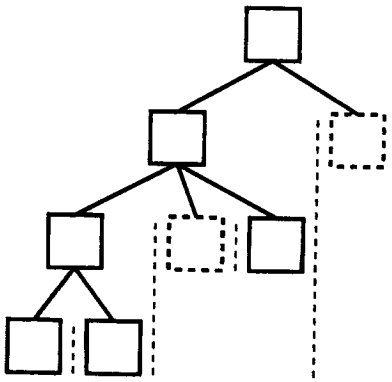

*Figure 2. An example of the tree layout generated by an algorithm that places children of a node to the right of already-placed children. The algorithm has located the dashed boxes at unaesthetic positions. The dashed lines indicate the notional columns imposed by this algorithm*

to a variety of tasks they should be inherently flexible, i.e. they should, even at the cost of some time or space efficiency, lend themselves to specialization for a given problem domain.

To make the tree-layout problem tractable, a suitable discrete representation of the drawing medium is required. Previous algorithms give each node a $y$ co-ordinate equal to its depth in the tree, thus effectively partitioning the drawing medium into horizontal bands. All these algorithms, and the approach of Reingold and Tilford[1]

in particular, are highly dependent on this property, making departures from it untenable. However, if nodes are permitted to have differing heights, the preservation of this property often leads to unaesthetic trees like Figure 1. These algorithms are difficult to extend directly to generalized trees and (even where thus extended) they would lead to unaesthetic generalized trees. The approach taken here is to use the best ideas of these algorithms to develop more general-purpose algorithms for tree layout. The question of how the drawing medium should be represented remains. The representation chosen here is the obvious one of equally-spaced rasters of discrete points such as, for example, the pixels of a bit-mapped display. Since we do not insist that the vertical and horizontal separations of points be equal, this representation is suitable for both lines of text and conventional raster displays.

Now that we have developed a representation for the drawing medium, we can state the problem more formally: given a finite $n$-ary tree where each rectangular node has a distinct predefined width and height, find a position in $\mathbf{Z} \times \mathbf{Z}$ for each node such that the following aesthetics are preserved:

1. Sibling nodes should have their top edges aligned horizontally.
2. Sibling nodes should be drawn in the same left-to-right order as their logical order.
3. Parent nodes should be centred over the centre of their leftmost and rightmost children.
4. A tree and its logical mirror image should be drawn as reflections of each other, and a subtree should be drawn in the same way no matter where it appears in a tree.
5. No edge joining the centre of the bottom of a node with the centre of the top of a child should cross any other such edge or node.
6. All nodes that share a raster should be separated horizontally by at least a distance $p > 0$. Note: for the purposes of this aesthetic a node is considered to extend a distance $q > 0$ above its top edge.
7. Each node should be separated vertically from its parent by exactly a distance $q$. If nodes are composed of lines of text on a bit-mapped display, then $q$ should be a multiple of the line height.

Aesthetics 1–3 have been generalized from Wetherell and Shannon,[5] aesthetic 4 is from Reingold and Tilford[1] and 5–6 are based on aesthetics from Supowit and Reingold.[2] To facilitate the display of trees on physical devices, the following physical limitation, generalized from Wetherell and Shannon,[4] may be added to the above aesthetics: tree drawings should occupy as little width and height as possible. Aesthetic 3 is interesting in that it causes even binary trees to be drawn differently from previous algorithms, since a single child will be drawn directly below its parent. Previous algorithms, however, place the logical left branch of a node strictly to the left of its parent and the right branch strictly to the right of its parent—even when there is only one branch. Since it does not make sense to have such a restriction in the case of $n$-ary trees, we must find an alternative aesthetic. The obvious alternatives are to centre the parent above its leftmost and rightmost children or to place the parent at the average of the branch positions.

An experiment by the author shows that most people (9 out of 10 subjects significant at the criterion level of 5 per cent by a sign test) prefer trees where parents are centred over children. In the experiment, each subject was asked which

of the two layouts (one with centred and one with averaged parent positions) of a 5-ary tree looked best. Each subject was shown a different randomly-generated tree of between 100 and 200 nodes. None of the other aesthetics was tested, since they all represent generalizations of commonly-accepted aesthetics and no good alternatives were apparent.

Below are two algorithms for the aesthetic layout of generalized trees. The first of these algorithms (algorithm 1) uses $\Theta(h)$ space and $O(nh)$ time, where the tree is $h$ rasters high and has $n$ nodes. Algorithm 1 is very fast in practice, producing good-quality output (Figure 3). For example, algorithm 1 can lay out a randomly-generated 5-ary tree of about 1000 nodes, each 5–10 rasters high, in an average of about 0·1 s on a Sun 3/60; by comparison, algorithm 2 takes 0·5 s on average to perform the same task. Algorithm 1 does not, however, always produce trees that respect aesthetic 4. Moreover, algorithm 1 also produces trees that are wider than they need be; even so, the effect is barely noticeable even in large trees. The second algorithm (algorithm 2) is guaranteed to meet all of the above requirements (Figure 3), but at the cost of $O(nh)$ space and $O(nh)$ time.

## ALGORITHM 1

The aesthetic-layout algorithm of Vaucher[3] carries out a postorder traversal of a binary tree. For each level in the tree it stores the leftmost position at which a node has been placed, and it positions a node either at some position defined by its parent's nominal position or the rightmost available position. To store the node positions and the rightmost available positions the algorithm uses a complicated data structure that consumes $O(n)$ space. By improving the data structure and generalizing the algorithm of Vaucher[3] it is possible to arrive at algorithm 1.

Rather than use the data structure of Vaucher,[3] algorithm 1 stores the absolute node positions in the node data structure itself and uses an auxiliary array to store the leftmost node positions raster by raster (note: algorithm 1 traverses the tree left to right, whereas Vaucher[3] traverses the tree right to left). Algorithm 1 (Appendix I contains a C version of the algorithm, Figure 3 an example of its output) proceeds

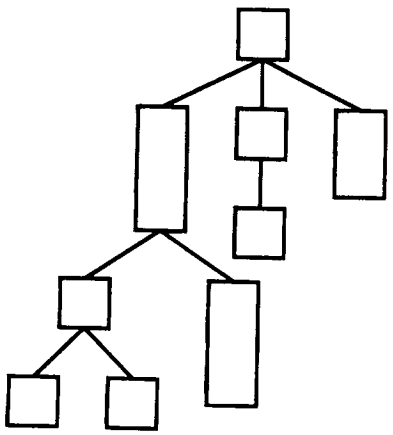

*Figure 3. An example of the tree layout generated by either algorithm 1 or 2*

as follows: a postorder traversal of the tree is carried out; if the nominal position of a node is such that the node would intersect a previously-placed node, then the nominal position of the node is changed so that it lies at a distance $p$ to the right of the rightmost node placed so far. Its children are placed with nominal positions relative to the parent appropriate for their width and number.

In outline, algorithm 1 proceeds as follows:

```
doShapeTree(tree, yPosition, newPosition, position)
If tree is empty then
  newPosition←position
else
  Force position to be to the right, by distance p, of any nodes already placed.
  If there are any branches in tree
    Set branchPosition to the rightmost position that the left-hand branch can
      occupy and allow the parent to be centred over its children.
  Position the leftmost branch at branchPosition.
  Position the other subbranches at branchPosition, each time incrementing
    branchPosition first so that the nodes do not overlap.
Adjust the last array to reflect the placement of this node.
Set the node's x and y values to the calculated values.
```

Figure 4 contains the data structure used by algorithms 1 and 2 to represent trees. The array branches is intended to be filled left to right with no gaps. Since clarity of exposition was the goal for both algorithms 1 and 2, the C code of Appendices I and II contain several redundancies. For example, the $y$ position of the nodes need not be recorded, since it can be calculated from the node widths and heights as the tree is drawn. In addition, the node data structure need not be a constant size.

## ALGORITHM 2

The aesthetic-layout algorithm of Reingold and Tilford[1] carries out a postorder traversal of a binary tree, placing subtrees of a node so that they are some predefined distance $p$ apart. In order to access the outline of the tree efficiently, the tree is threaded as the algorithm carries out its traversal. Unfortunately such expediencies are not applicable to generalized trees, since nodes at the same level in a tree need not share the same output raster. Thus, although algorithm 2 uses the idea of placing branches as closely together as possible and synthesizing the tree layout from the

```
typedef struct TreeNode {
  int              height;
  int              width;
  int              x;
  int              y;
  int              nrBranches;
  struct TreeNode *branches[NR_BRANCHES]} TreeNode;
```

*Figure 4. The data structure used to represent n-ary trees in both algorithm 1 and algorithm 2*

bottom up, it does not (and indeed cannot) take advantage of the threading techniques of Reingold and Tilford.[1]

Algorithm 2 (Appendix II) proceeds as follows: a postorder traversal of the tree is carried out where branches are placed at a nominal $x$ offset, from the parent, of 0. Each branch has associated with it left and right outlines; once all the sub-branches have been placed, proceed left to right, positioning branches so that they are $p$ units apart. New left and right outlines are accumulated as branches are added, and finally the positions of the branches are adjusted relative to the parent node ($x = 0$). Unlike algorithm 1, algorithm 2 uses $x$ co-ordinates relative to parent nodes (root at 0), thus facilitating the necessary movement of branches. By keeping each tree outline as two lists where each offset (from the centre of the root) is relative to the previous offset and constructing them relative to the leftmost branch, the algorithm can be simplified and the cost of maintaining the tree outlines reduced. Note also that, for efficiency reasons, the algorithm allocates space for the tree outline proportionally to the height of the tree in rasters. At a small cost in code complexity the amount of space required by the data structure, on average, could be reduced by storing the offsets in reverse order and extending the arrays as required with calls to, say, realloc; however, this could reduce the performance of the algorithm.

In outline, algorithm 2 proceeds as follows:

```
doShapeTree(tree, yPosition, newPosition, left, right)
if tree has no branches then
   Create left and right
else
   Position each branch at x = 0 remembering their outlines.
   Set left and right to the left and right outline of the first branch
   Push the branches apart so that they do not overlap, updating left and right
      as branches are added.
Move the branches so they centre about x = 0.
Update left and right to reflect the position of the root of the tree.
```

If we are prepared to ignore aesthetic 2, we can often make a tree narrower by changing the order of siblings. However, it would be computationally costly to find the narrowest possible tree by exhaustive search. A more reasonable approach that eliminates many bad sibling orders, but at a low computational cost, is to use a greedy optimization algorithm that optimizes each group of siblings independently. Algorithm 2 can easily be adjusted to do this in one of two ways. First, as we proceed left to right we test to see if the overlap between the sub-branches would be smaller if the current sub-branch were placed to the left of all the nodes placed so far; if so, we reorder the branches appropriately (Figures 5 and 6). Secondly, we could reorder the branches if the result would actually be narrower but at the cost of extra computation and extra storage (Figures 5 and 6). The second technique, however, suffers from diminishing returns due to the biases it introduces into subtrees. Both optimization techniques are sensitive to the overall shape of the tree—preferring unbalanced trees. Also, if the arity of the tree is low enough then it may pay to try, at each parent node, all possible child-node orderings.

Suppose we wish to lay a tree out on pieces of paper, but the tree is too large

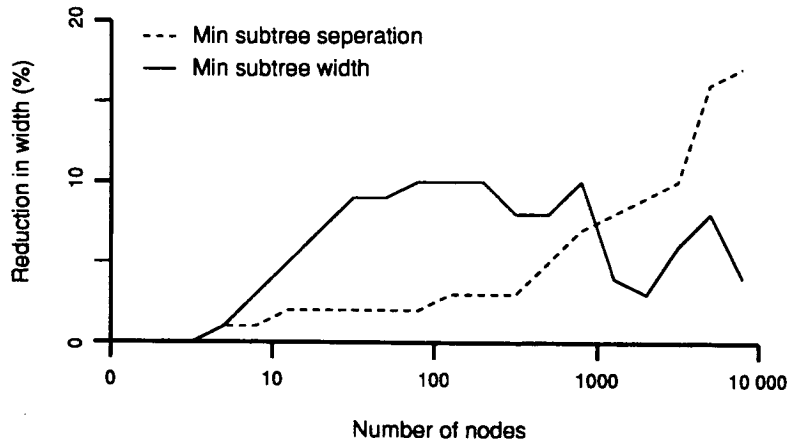


*Figure 5. A plot of the mean percentage reduction in the width of a randomly-generated binary tree if the optimizing versions of algorithm 2 are used instead of the non-optimizing version. The corresponding curves for higher-arity trees are very similar*

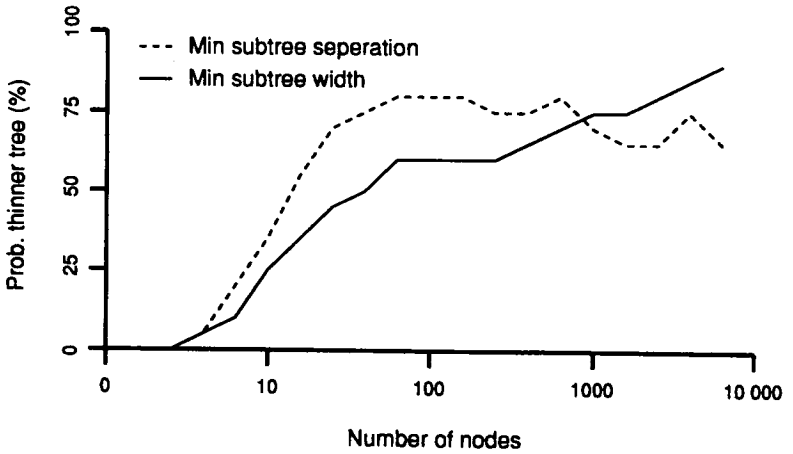


*Figure 6. A plot of the probability that a randomly-generated binary tree will be narrower if laid out with the optimizing versions of algorithm 2 instead of the non-optimizing version. The corresponding curves for higher-arity trees are very similar*

to fit on a single piece of paper. One possibility is to modify algorithm 1 as follows. Whenever a subtree would be too wide or high to fit on a page we mark the largest subtree as a reference node and reconstruct its outline accordingly. When drawing a tree we replace marked nodes with a reference to the appropriate page and add the marked subtree to the list of subtrees to be drawn. Provided we draw trees using a preorder traversal, they will appear in an appropriate order. This technique may waste some space, but it is very simple to implement.

## CONCLUSION AND DISCUSSION

We have given two algorithms (algorithms 1 and 2) for the layout of general trees. Although algorithm 1 is faster than algorithm 2, for common problem sizes the speed difference is negligible, thus making algorithm 2's better aesthetics of primary concern. Algorithm 1 uses a much smaller amount of storage and is recommended where the trees are likely to be high (in rasters). As given, both algorithms lend themselves to non-rectangular nodes. If the nodes are in fact rectangular then run-line[7] encoding of the outline of a tree may lead to efficiency improvements. Modifications to algorithm 2 have been suggested that allow trees to be rearranged so that they are narrower. Also, a modification to algorithm 2 has been suggested that allows trees to be laid out over several sheets of paper. Both algorithms 1 and 2 produce the shortest possible trees.

### ACKNOWLEDGEMENTS

I would like to thank Terry Halpin and Peter Eades for their comments on an earlier draft of this paper. This research was supported by an Australian Postgraduate Award and a Key Centre for Software Technology Research Fellowship.

### REFERENCES

1. Edward M. Reingold and John S. Tilford, 'Tidier drawings of trees', *IEEE Transactions Software Engineering*, **7**, (2), 223–228 (1981).
2. Kenneth J. Supowit and Edward M. Reingold, 'The complexity of drawing trees nicely', *Acta Informatica*, **18**, 377–392 (1983).
3. Jean G. Vaucher, 'Pretty-printing of trees', *Software—Practice and Experience*, **10**, 553–561 (1980).
4. Charles Wetherell and Alfred Shannon, 'Tidy drawings of trees', *IEEE Trans. Software Engineering*, **5**, (5), 514–520 (1979).
5. Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
6. Raymond M. Smullyan, *First-Order Logic*, Springer-Verlag, Berlin, 1968.
7. Thomas J. Lynch, *Data Compression: Techniques and Applications*, Dickenson, Belmont, California, 1985.

## APPENDIX I: C CODE FOR ALGORITHM 1

```
void doShapeTree(tree, yPosition, last, newPosition, position)
  int   last[]; /* Last position occupied by a node at each raster. */
  int  *newPosition; /* Where the node was actually placed. */
  int   position; /* Leftmost legal position for the node. */
  Tree  tree; /* The tree to draw. */
  int   yPosition; /* The raster the top of the node lies on. */
{
  int branchPosition;
  int i;
  int leftPosition;
  int rightPosition;
  int width;

  if (tree == (Tree)NULL)
    *newPosition = position;
```

```
  else { /* Place subtree. */
    /* Ensure the nominal position of the node is right of any other node. */
    for (i = yPosition-Y_SEPARATION; i < yPosition+tree->height; i++)
      position = max(position,
                     last[i] + MIN_X_SEPARATION + tree->width/2);

    if (tree->nrBranches >= 1) { /* Place branches if they exist. */
      if (tree->nrBranches > 1) {
        width = (tree->branches[0]->width +
                 tree->branches[tree->nrBranches-1]->width)/2 +
                (tree->nrBranches-1)*MIN_X_SEPARATION;
        for (i = 1; i < tree->nrBranches-1; i++)
          width += tree->branches[i]->width; }

        else
          width = 0;

        branchPosition = position - width/2;
        /* Position far left branch. */
        doShapeTree(tree->branches[0], yPosition+tree->height+Y_SEPARATION,
                    last, &leftPosition, branchPosition);

        /* Position the other branches if they exist. */
        rightPosition = leftPosition;
        for (i = 1; i < tree->nrBranches; i++) {
          branchPosition += MIN_X_SEPARATION +
                        (tree->branches[i-1]->width +
                         tree->branches[i]->width)/2;
        doShapeTree(tree->branches[i], yPosition+tree->height+Y_SEPARATION,
                    last, &rightPosition, branchPosition);
      } /* for */

      position = (leftPosition+rightPosition)/2;

    } /* tree->nrBranches >= 1 */

    /* Add node to last. */
    for (i = yPosition-Y_SEPARATION; i < yPosition+tree->height; i++)
      last[i] = position + (tree->width+1)/2;

    tree->x = position;
    tree->y = yPosition;

    *newPosition = position;

  } /* if */
  /* doShapeTree */;
```

## APPENDIX II: C CODE FOR ALGORITHM 2

```
typedef struct edge {
  int yPosition;
  int offset[1];
} edge;

void doShapeTree(tree, height, yPosition, left, right)
  int    height; /* The height of the original tree in rasters. */
  edge **left; /* The left outline of the placed tree. */
  edge **right; /* The right outline of the placed tree. */
  Tree   tree; /* The tree to draw. */
  int    yPosition; /* The raster the top of node lies on. */

{
  int    centre;
  int    i;
  int    j;
  edge **leftOutline; /* The left outlines of the subbranches */
  int    overlap; /* The amount of overlap between two branches. */
  edge **rightOutline; /* The right outlines of the subbranches */

  if (tree->nrBranches == 0) {
    *left  = newEdge(height);
    *right = newEdge(height);
    (*left)->yPosition = yPosition+tree->height-1;
    (*right)->yPosition = yPosition+tree->height-1; }

  else {
    leftOutline = (edge **)malloc((unsigned)(tree->nrBranches*sizeof(edge *)));
    rightOutline = (edge **)malloc((unsigned)(tree->nrBranches*sizeof(edge *)));

    for (i = 0; i < tree->nrBranches; i++)
      doShapeTree(tree->branches[i], height,
                  (int)(yPosition+tree->height+Y_SEPARATION),
                  &leftOutline[i], &rightOutline[i]);

    /* Set up left and right. */
    *left = leftOutline[0];
    *right = rightOutline[0];

    /* Position branches relative to the left branch. */
    tree->branches[0]->x = 0;
    for (i = 0; i < tree->nrBranches - 1; i++) {
      /* Calculate maximum overlap. */
      overlap = 0;
      for (j = yPosition+tree->height+Y_SEPARATION;
```

```
            j <= min(leftOutline[i+1]->yPosition, (*right)->yPosition);
            j++)
          overlap = max(overlap, leftOutline[i+1]->offset[j] +
                                 (*right)->offset[j]);

        /* Push branches apart. */
        tree->branches[i+1]->x = overlap+MIN_X_SEPARATION;

        /* Adjust left outline. */
        for (j = (*left)->yPosition+1; j <= leftOutline[i+1]->yPosition; j++)
          (*left)->offset[j] = leftOutline[i+1]->offset[j] -
                               tree->branches[i+1]->x;
        (*left)->yPosition = max((*left)->yPosition,
                               leftOutline[i+1]->yPosition);

        /* Adjust right outline */
        for (j = yPosition; j <= rightOutline[i+1]->yPosition; j++)
          (*right)->offset[j] = rightOutline[i+1]->offset[j] +
                               tree->branches[i+1]->x;
        (*right)->yPosition = max((*right)->yPosition,
                               rightOutline[i+1]->yPosition);
      } /* for */

      if (tree->nrBranches > 1) {
        /* Position branches relative to the centre. */
        centre = tree->branches[tree->nrBranches-1]->x/2;
        for (i = 0; i < tree->nrBranches; i++)
          tree->branches[i]->x -= centre;

        for (i = yPosition; i <= (*left)->yPosition; i++)
          (*left)->offset[i] += centre;

        for (i = yPosition; i <= (*right)->yPosition; i++)
          (*right)->offset[i] -= centre;
       } /* if */

      /* Free the old outlines. */
      for (i = 1; i < tree->nrBranches; i++) {
        (void)free((char *)leftOutline[i]);
        (void)free((char *)rightOutline[i]);
      } /* for */
    } /* if */

    for (i = yPosition - Y_SEPARATION; i < yPosition+tree->height; i++) {
      (*left)->offset[i] = tree->width/2;
      (*right)->offset[i] = (tree->width+1)/2;
    } /* for */

    tree->y = yPosition;

} /* doShapeTree */;
```