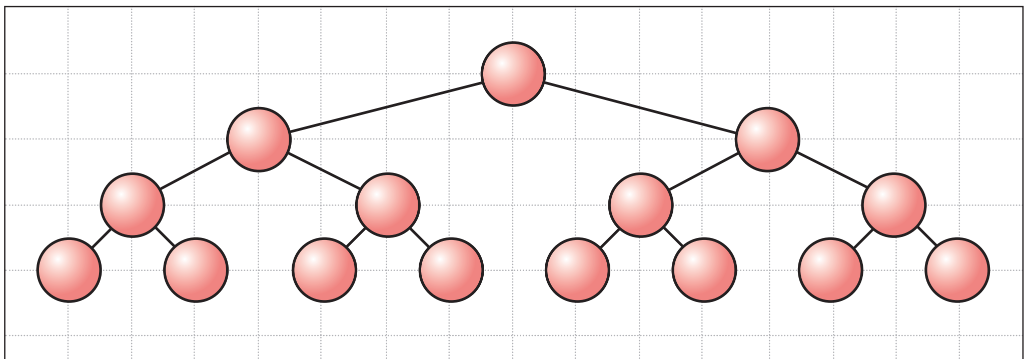
Drawing binary trees

Sometimes the easiest way to visualise an algorithm or a data structure is to draw it


▲ **Figure 1: A complete binary tree.**

In this issue...

▶ WHAT'S COVERED
Stay on top of your algorithms by charting them out. We explain the basics of nodes and trees, and the benefits of keeping them in order.

Nugget

The problem with the algorithm described here is that the end-result tends to be too spread out horizontally. To produce a more aesthetic image, we can constrain the nodes to their level, but then apply an iterative algorithm that first mimics the links as springs (so that children are drawn to their parents) and second mimics electrical repulsion between nodes on the same level. After a few cycles, the electrical repulsion will exactly counter the string-like attraction, and the nodes will be at their 'sweet' spot, resulting in a better image. ■

Sometimes the easiest way to visualise an algorithm or a data structure is to draw it. Instead of using a vector-based drawing program, you can achieve more consistent results by writing a specific program to draw the diagrams. This article shows a simple algorithm for drawing binary trees, a two-dimensional data structure which we view in a top-down manner, but which is easier to draw bottom-up.

Recently on my personal blog, I've been writing about binary trees. In order to illustrate the points I was making, I wanted to be able to create images of such trees very easily. Although drawing data structures such as trees or graphs can get very complicated very quickly, there is an algorithm for drawing a binary tree that is fast and produces a pretty good rendition of the tree on the screen. This means that choosing to use trees as examples need not be a painful decision.

Binary trees

First, let's define what we mean by a binary tree. I'll use a recursive definition, that is, a definition which invokes itself, because it's succinct and will help when drawing the tree. A binary tree can either be empty, or consist of a node, called the root, that has links to two other binary trees.

The two sub-trees hanging off the root are normally called the 'left child' and the 'right child'.

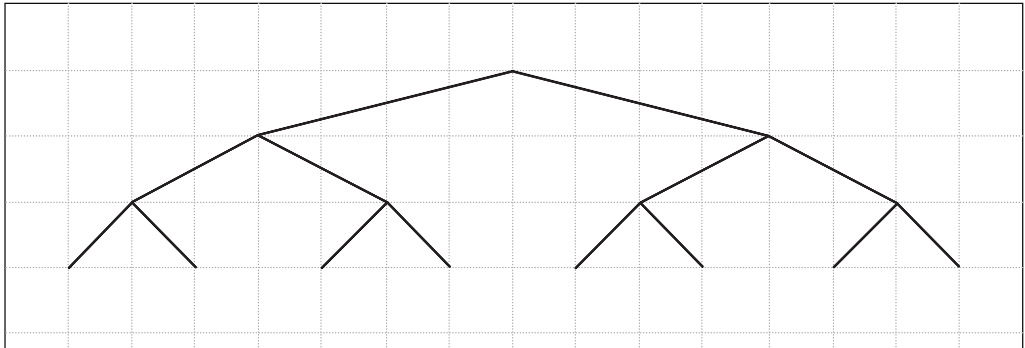
When we draw a tree, we usually draw the root at the top (see Figure 1). Its children are drawn to its left and to its right (it is between them) and underneath the root, so that we see a visual representation of the terms 'left child' and 'right child'. The children of the root are drawn in the same manner underneath their parents. Determining exactly where to place each node on the surface we're drawing on is difficult: we certainly can't draw the root first, since its horizontal position depends on how many nodes there are in the root's left child.

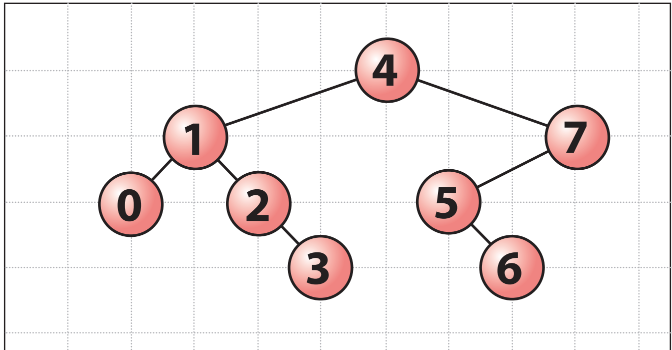
Let's imagine that we have a grid overlaying the surface on which we're going to draw the tree. The grid is fairly coarse, and we are only going to draw a node on an intersection of the horizontal grid line and a vertical one. Furthermore we shall assume that the horizontal grid lines are counted from the top of the surface, and the vertical grid lines are counted from the left. In other words, the grid acts as a coordinate space whose origin (0,0) is at the top left of the drawing surface. (This is upside down from the usual way we learnt drawing graphs at school, where the origin was at the bottom-left corner.)

This means that the root will be drawn on the horizontal grid line 0 – its y-coordinate is 0 – on an unspecified vertical grid line. Its x-coordinate still needs to be calculated.

Node coordinates

When talking about a tree, we use the term 'level' to describe where a node appears vertically. The level of a node in a binary tree is given


▲ **Figure 2: The same binary tree as shown above with only the links visible.**



▲ **Figure 3: A binary tree showing the order that the nodes were drawn.**

by the number of links you have to travel down from the root in order to reach it. The root is therefore at level 0 (no links need to be followed), and its children nodes are at level 1. Their children are at level 2, and so on. The level of a node is therefore the same as the y-coordinate where it will be drawn in our grid system.

But what about a node's x-coordinate? To make it easier on ourselves, and to avoid having to detect whether one node bumps into or overlays another, we shall institute a rule that there will only be one node per vertical grid line. This may mean, as we shall see, that the tree is a little too spread out horizontally, but it'll be a whole lot easier to deal with. What this means in practice is that the root's x-coordinate will be equal to the total number of nodes in its left subtree.

To suit the recursive nature of binary trees, the algorithm itself is recursive. Given the root of a binary tree and its level, we draw its left subtree starting one level down, draw its right subtree one level down, and then draw the root at its level.

It may appear that we can perform the algorithm by first drawing the left subtree, then the root, and lastly the right subtree, because after drawing the left subtree we have enough information to draw the root (in essence, we'll know the number of nodes in the left subtree). This supposition would indeed be the case if the nodes did not need linking together.

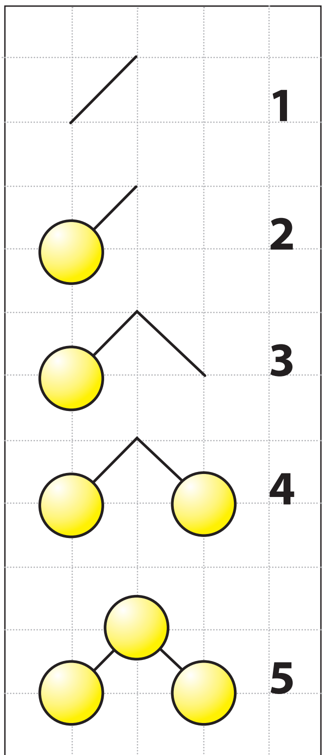
Linking the nodes

To draw a link, or in other words a line between two nodes, it's going to be orders of magnitude easier to draw the line between the grid intersections before we draw the nodes on top of the grid intersections. The filled circles representing the nodes will overwrite the portion of the lines by the intersection and give a very clean join. If we didn't do it this way, we'd have to do some funky trigonometry in order to work out where the link's line intersects the node's circle to cut it off exactly, and to be honest, it's not worth it.

Figure 2 shows the same binary tree as Figure 1, except that I've suppressed the drawing of the nodes as circles. You can see the links are drawn from grid intersection to grid intersection.

This means that a parent node can only be drawn once the links to both its children have been drawn, so that it will overlap the lines from its children.

The algorithm makes use of a counter to count the number of nodes that have been painted, up to the point we need to draw another node. Initially, it's set to 0. When, through our traversal of the tree, we identify a node that can be painted, we'll know its level, that is, its y-coordinate, from how far we've travelled down the tree (every time we recursively call the algorithm, we increment the level number being passed).



▲ **This snapshot sequence shows the drawing process.**

The x-coordinate is slightly more complicated. We draw the left subtree and now know how many nodes are in it from the node counter. The root will have this value as its x-coordinate, but we can't draw it until we have drawn the right subtree. So we make a note of the node counter's value as if we had drawn the root, increment the node counter, and then draw the right subtree. Once that's done, we can draw the root, using the saved x-coordinate.

We've already identified several routines that we need to write. The first is the one we call to draw the full tree. All it does is to initialise the node counter to 0 and call the routine to draw the children (it should pass level number 1 to the draw children routine), which returns the x-coordinate for the root, before drawing the root. We have to have a special routine for this, since the root of the whole tree does not have a link upwards; it has no parent node.

The routine to draw the children calls another routine to draw the left subtree (passing its level), saves the value of the node counter for the parent node's x-coordinate, increments it, and then calls a routine to draw the right subtree. This last routine returns the saved node counter value for the caller (who will then use it to draw the parent node).

Recursion routines

Now we get into the actual recursion. The routine that draws the left subtree first calls the routine that draws the children. This will return the x-coordinate of the root of this subtree. We then draw the link from this node's grid intersection (we know both its coordinates) to its parent. We know the parent's level (it's one less than the level of this node), and we also know the parent's x-coordinate from the current value of the node counter. We then draw the node itself.

> ### *Nugget*
> The method shown in the main article draws lines at an angle directly from child to parent node. Another common way of drawing lines is known as orthogonal drawing. Here the links are drawn as line segments along the grid lines and the nodes are usually rendered as rectangles. In general, the lines aren't kept separate, but instead are allowed to merge so that the effect is of one line segment leaving the parent and splitting into two, with one part going to the left child and the other to the right. The best example for this kind of tree drawing is an organisation chart showing the management chain in a company. ■

Finally, the routine that draws the right subtree saves the current value of the node counter (it's the x-coordinate of the subtree's parent) and calls the draw children routine, which returns the x-coordinate of the root of this subtree. We can then draw the link from this node's grid intersection to the parent's, and draw the node itself.

That's it for the algorithm itself. If you think about what's going on in the recursion, you'll see that the tree is drawn and filled in from the lowest levels up to the root, since that is the only way we can calculate the position of all the nodes. The rest of the code that you'd need to write to implement the algorithms is code to actually draw the lines and the filled circles on the graphics surface, and that depends on the graphics library API you are using.

Figure 3 shows a less balanced binary tree where the number in each node is the order in which the nodes were drawn.

The figures shown in this article were created by this algorithm and 'drawn' by emitting PostScript commands to an Encapsulated PostScript (EPS) file. The typesetter for this article could grow or shrink the images without the fuzziness or jaggedness you'd see from doing the same to a JPG or PNG image. Every little helps. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk