Illinois Institute of Technology
Department of Computer Science

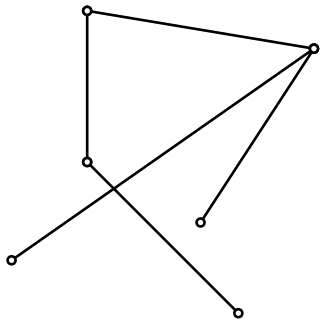
Lecture 1: January 11, 2016

CS 430 Introduction to Algorithms
Spring Semester, 2016

How to Draw a Binary Tree

1 Defining the problem

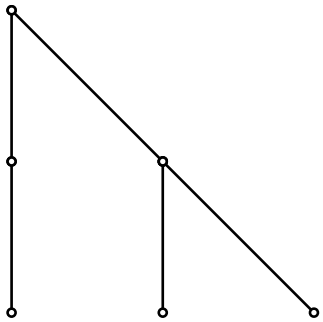
The first algorithm we examine is one to determine a clean way to draw a tree. For instance, we could draw a tree in this fashion:



This is undesirable. We can give an *aesthetic* to define properties that we would like to see in the output:

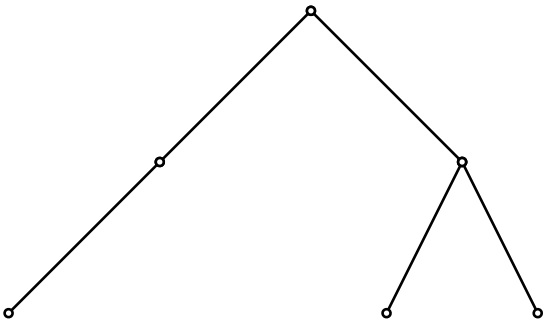
Aesthetic 1: Nodes on the same level should be on a straight line, and lines defining levels should be parallel and evenly spaced. As well, nodes on a level should be in the same left-to-right order as in the level-order traversal.

These observations lead to this drawing:



This is better but still not great. Let us insist on a second aesthetic:

Aesthetic 2: A left child must be to the left of its parent; a right child must be to the right of its parent.



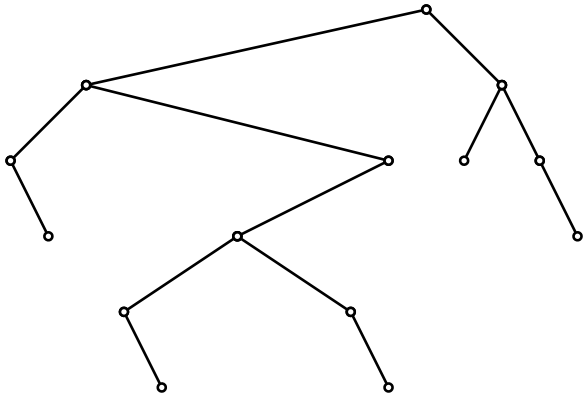
How can we accomplish this algorithmically?

2 Knuth's algorithm (1971)

Knuth's algorithm is a simple recursive algorithm for drawing a tree. To understand the order of the operations, think of the tree being printed on a line printer *on its side*, so the right part of the tree comes out first.

1. Go down a level and print the *right subtree* recursively.
2. Go up a level and to the left and print the *root*.
3. Go down a level and to the right and print the *left subtree* recursively.

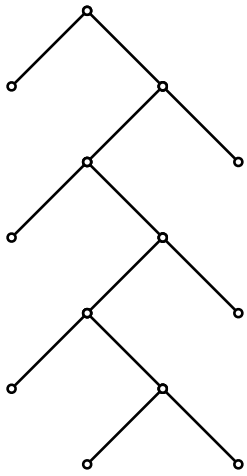
This algorithm yields results like this:



3 Redefining the problem: additional aesthetics

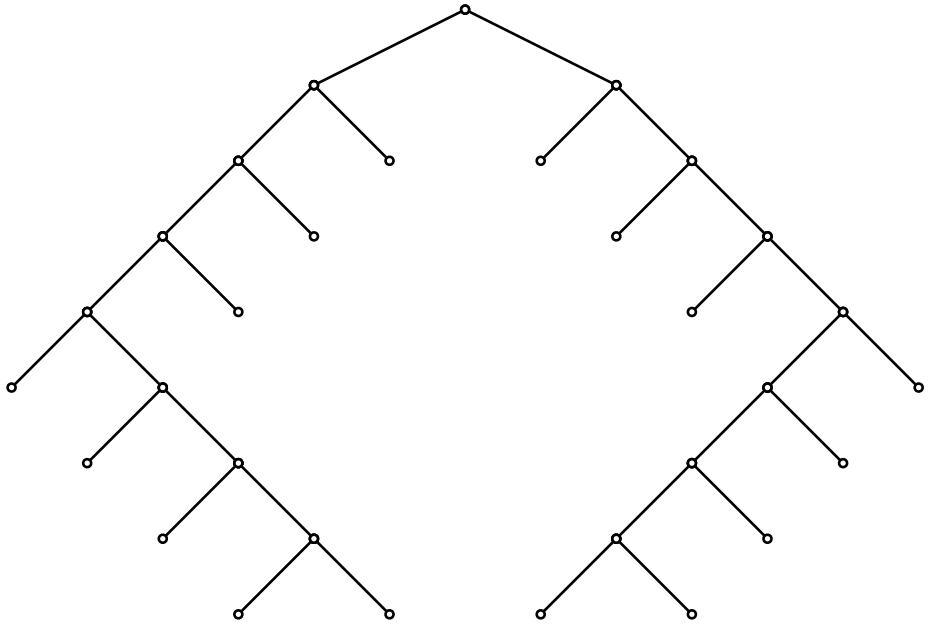
This is reasonable but still not quite what we'd like. We can propose a third aesthetic:

Aesthetic 3: A parent should be centered over its children.

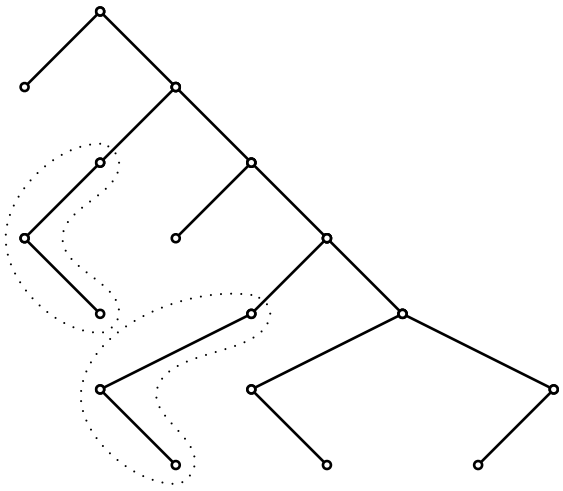


A fourth aesthetic that we propose demands some form of symmetry:

Aesthetic 4: A tree and its mirror image should yield drawings that are mirror images. More generally, a subtree should be drawn in the same way, wherever it occurs in the tree.



Note that this aesthetic requires that certain trees be drawn in more space than absolutely necessary. Notice that this tree, drawn as narrowly as possible, requires that two identical subtrees be drawn differently, violating aesthetic 4:



4 A new algorithm

1. Recursively place the left subtree.
2. Recursively place the right subtree.
3. Put the two rigidly formed subtrees as close together as possible. That is, place the two subtrees so their roots overlap, and then move them apart just enough so no part of the left subtree overlaps a part of the right subtree.[1]

How can we follow the inner contours of the subtrees? We can use the notion of a *threaded tree* to help us. Nodes whose "contour successors" are not their children must be leaves and thus have must each have a null pointer. If, in the place of these null pointers, we insert auxiliary pointers, or threads, we can follow the contour of a tree beyond a leaf.

5 Analysis of the algorithm

We can say that no work is involved in drawing the empty tree or in drawing the tree with only one node—that is, $F(\text{empty tree}) = F(\circ) = 0$. How much work is involved in drawing a more complex tree T? Say that T is made up of some parent with two children, a left child T_l and a right child T_r. Then the work involved in drawing T is simply the sum of the work involved in drawing T_l, the work involved in drawing T_r, and the work involved in combining the results. The first and second steps are applied recursively, and our analysis follows suit. The third step is somewhat more interesting. We must examine the top level of each tree, the next level down of each tree, and so on, until we have reached the bottom of one of the trees. If $H(T)$ is defined as the *height* of T in nodes (that is, the number of levels of nodes in the tree), the third step requires the *minimum* of $H(T_l)$ and $H(T_r)$ steps. Thus:

$$F(T) = F(T_l) + F(T_r) + \min\{H(T_l), H(T_r)\}$$

[1]For a more detailed treatment of this algorithm, see E. Reingold and J. Tilford, "Tidier Drawings of Trees," in *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 2, March 1981.

Claim: $F(T) = N(T) - H(T)$, where $N(T)$ is the number of nodes in the tree T.

Proof 1: We prove our claim by induction on n, the number of nodes in T. Clearly, by our definition of height, the claim holds for $n = 0$ and $n = 1$. Now we assume that the claim holds for trees of less than n nodes and show that it holds for trees of exactly n nodes. Say T_l contains k nodes; clearly $k < n$. Then, by the inductive hypothesis,

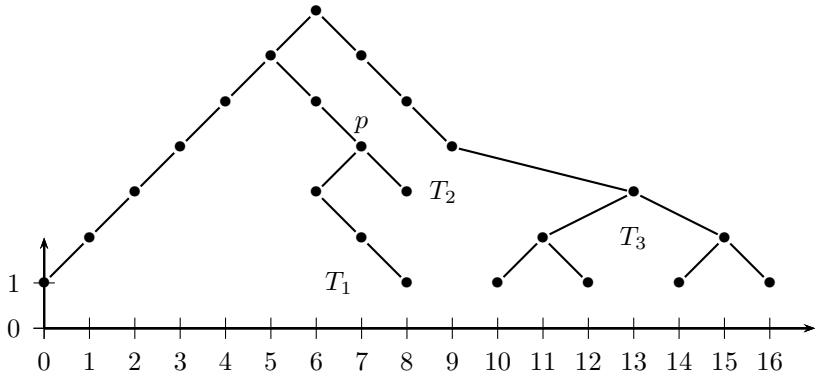
$$\begin{aligned} F(T) &= (N(T_l) - H(T_l)) + (N(T_r) - H(T_r)) + \min\{H(T_l), H(T_r)\} \\ &= (k - H(T_l)) + (n - k - 1 - H(T_r)) + \min\{H(T_l), H(T_r)\} \\ &= n - 1 - H(T_l) - H(T_r) + \min\{H(T_l), H(T_r)\} \\ &= n - (\max\{H(T_l), H(T_r)\} + 1) \\ &= n - H(T) \\ &= N(T) - H(T) \end{aligned}$$

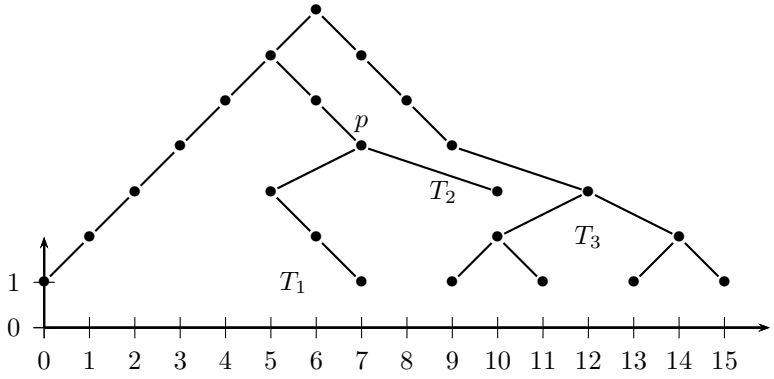
Proof 2: An alternative proof counts the number of times the algorithm combines two subtrees. How many "gluing" operations take place at each level of the tree? Clearly, if the level has k nodes, then $k - 1$ gluings must take place. Summing over all levels, there are $N(T)$ nodes in the entire tree and $H(T)$ levels, so $N(T) - H(T)$ gluing operations must occur.

Thus this is a linear-time algorithm.

6 What about narrow trees?

What happens if we modify our aesthetics and demand the narrowest output possible? The problem then becomes significantly more difficult. The difficulty lies in the idea that to find the narrowest possible drawing of a tree, its subtrees must sometimes be drawn more widely than necessary. An example of this is the pair of drawings below.





This is similar to the difficulty found in the traveling salesman problem. We will return to this problem later in the semester.