Trees

Margaret M. Fleck

26 October 2009

We've seen trees already, but informally. This lecture firms up the details of trees and tree terminology (section 10.1 of Rosen).

1 Announcements

Midterm coming up a week from Wednesday (4 November). As for the first midterm, discussions that week will be exam review and office hours will be shifted to the early part of the week.

Quizzes are almost graded: will be returned in discussion sections.

There are a couple important typos on HW 8: see newsgroup and new posted version of HW 8.

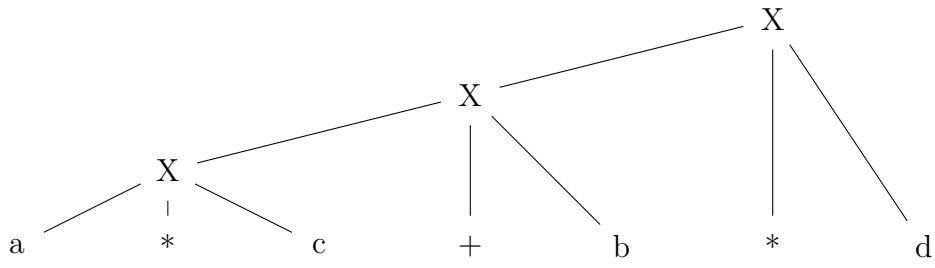
2 Why trees?

Computer scientists are obsessed with trees, because trees of various sorts show up in a wide range of contexts. Originally, of course, trees are a familiar form of plant. Oddly, computer scientists normally draw ours upside down. ¡smile¿ Trees in computer science show up as

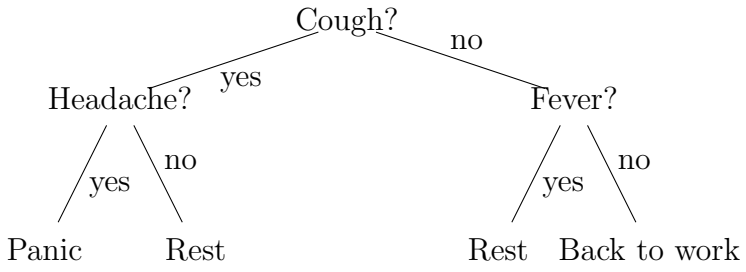
- Organization of real-world data: family/geneaology trees, taxonomies (e.g. animal species)
- Data structures for efficiently storing and retrieving data. The basic idea is the same one we saw for binary search within an array: sort the data, so that you can repeatedly cut your search area in half.

- Parse trees, which show the structure of a piece of (for example) computer program, so that the compiler can correctly produce the corresponding machine code.
- Decision trees, which classify data by asking a series of questions. Each tree node contains a question, whose answer directs you to one of the node's children.

For example, here's a parse tree for "((a * c) + b) * d)"



Here's what a medical decision tree might look like. Decision trees are also used for engineering classification problems, such as transcribing speech waveforms into the basic sounds of a natural language.



3 Defining trees

Because trees are used for such a wide range of different purposes, there are several radically-different ways to define what a tree is. Fortunately, they amount to the same thing even though they look very different.

A common feature of all definitions is that a tree is made up of a set of nodes/vertices and a set of edges that join them together. For the purposes of this class, both sets (and thus the tree) will be finite. These definitions can be extended to infinite trees, but we won't go there.

The most familiar definition of a tree involves giving directions to the edges. We then say that each node has exactly one outgoing edge, which points to its parent. Or, instead, we could say that each node has exactly one incoming edge, coming from its (one) parent. In both cases, except for one special root node, which has no parent. This is how you think about trees when you are designing data structures for computer programs.

Researchers in graph theory start with a general graph, i.e. a bunch of nodes strewn all over space in no particular pattern, joined together by edges with no specific direction to each edge. A "tree" is a graph with two properties. First, it must be connected, i.e. there's a way to get from any node to any other node via the edges. Second, it contains no cycles (loops where you go around in a circle and get back to the same node).

If you take one of these graph theory trees, choose any node to be the root, pick up the tree by this node and shake it a bit so it hangs down nicely, it will look like a normal tree. It doesn't matter which node you pick to be the root. A graph theory tree with a designated root is called a "rooted tree" in graph theory, but it's what we're just calling a "tree."

Finally, we can define trees recursively. I'll define a special case called a "full binary tree," but the idea extends to any sort of tree. Our definition looks like:

> Base: a single node with no edges is a tree
>
> Induction: if A and B are two trees, then you can make a bigger tree by taking a root node r and grafting A and B onto r as its children (using two new edges).

This recursive picture of trees is extremely important when we come to write proofs.

4 Basic tree terminology

Much of the basic terminology for trees is based on either the plant analogy or the geneaology analogy.

A tree has one special "root" node that has no parent. Every other node is joined by an edge to exactly one parent node. If p is the parent of x, then x is a child of p. The parent is always closer to the root than the child. If y is also a child of p, then x and y are siblings.

A leaf node is a node that has no children. A node that does have children is known as an internal node. The root is an internal node, except in the special case of a tree that consists of just one node (and no edges).

If you can get from x to g by following one or more parent links, then g is an ancestor of x and x is a descendent of g. x is not an ancestor/descendent of itself. At least, this is true in Rosen. It's a likely place for authors to differ, since normal mathematical convention would be the other way.

If you pick some random node a in a tree T, the subtree rooted at a consists of a, all its descendents. and all the edges linking them.

The nodes of a tree can be organized into levels, based on how many edges away from the root they are. The root is defined to be level 0. Its children are level 1. Their children are level 2, and so forth. You can also define the height of a node as the number of edges in a path from that node up to the root.

The height of a tree is the maximum level of any of its nodes. Or, equivalently, the maximum level of any of its leaves.

5 m-ary trees

Many applications restrict how many children each node can have. A binary tree (very common!) allows each node to have at most two children. An m-ary tree allows each node to have up to m children. Trees with "fat" nodes with a large bound on the number of children (e.g. 16) occur in some storage applications.

Important special cases involve trees that are nicely filled out in some sense. Here, there is considerable variation in terminology as you move from author to author, so always check which definition your author is using.

In a "full" m-ary tree, each node has either zero or m children. Never an intermediate number. So in a full 3-ary tree, you can't have a node with one

child or with two children.

In a"complete" m-ary tree, all leaves are at the same height. Normally, we'd be interested only in "full complete" m-ary trees, where this means that the whole bottom level is fully populated with leaves.

In a "balanced" m-ary tree of height h, all leaves are either at height h or at height $h-1$. Balanced trees are useful when you want to store n items (where n is some random natural number) while keeping all the leaves at approximately the same height. Balanced trees lead to good behavior from algorithms trying to find things in the tree.