Trees and structural induction

Margaret M. Fleck

25 October 2010

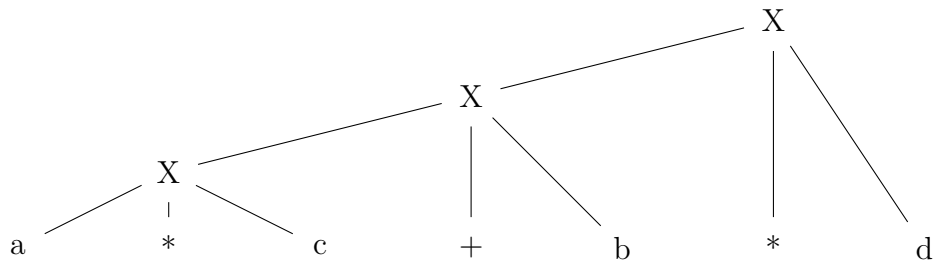
These notes cover trees, tree induction, and structural induction. (Sections 10.1, 4.3 of Rosen.)

1 Why trees?

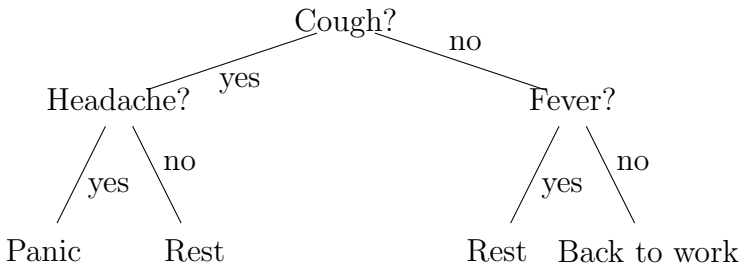
Computer scientists are obsessed with trees, because trees of various sorts show up in a wide range of contexts. Originally, of course, trees are a familiar form of plant. Oddly, computer scientists normally draw ours upside down. [smile] Trees in computer science show up as

- Organization of real-world data: family/geneaology trees, taxonomies (e.g. animal subspecies, species, genera, families)
- 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.



2 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. That is, for each edge, we know which end is the "parent" and which end is the "child." In drawings, the direction is sometimes indicated by arrows on the edges (e.g. on the child end) but more often it is indicated implicitly by drawing parents higher on the page. This is how you think about trees when you are designing data structures for computer programs.

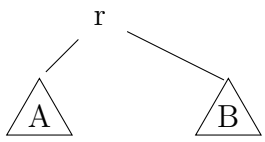
Researchers in graph theory start with a general graph, i.e. a bunch of nodes strewn all over space in no particular pattern, joined together by edges with no specific direction to each edge. A "tree" is a graph with two properties. First, it must be connected, i.e. there's a way to get from any node 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. Recursive definitions will be important when we come to write proofs involving trees. A simple recursive definition might look 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 new root node r and attaching the roots of A and B onto r as its children

Here's a picture of the inductive case:



This definition only creates trees in which each node has either zero or two children. If we wanted to allow nodes with a single child, we'd need another inductive rule:

> Induction: if A is a tree, then you can make a bigger tree by taking a new root node r and attaching the root of A onto r as its child

This idea can be extended to allow 3 children, or 4 children, or n children (where n is any finite integer).

3 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. We will treat x as an ancestor/descendent of itself. [1] The ancestors/descendents of x other than x itself are its "proper" ancestors/descendents.

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.

[1] Some reputable authors don't. This is a place where you always double-check what convention the author is using.

4 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, because the most useful definitions depend on the application. So always check which definitions 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, nodes can have zero or three children, but not one child or 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.

For restricted types of trees like this, there are strong relationships between the numbers of different types of notes. for example:

Claim 1 *A full m-ary tree with i internal nodes has $mi + 1$ nodes total.*

To see why this is true, notice that there are two types of nodes: nodes with a parent and nodes without a parent. A tree has exactly one node with no parent. We can count the nodes with a parent by taking the number of parents in the tree (i) and multiplying by the branching factor m.

Therefore, the number of leaves in a full m-ary tree with i internal nodes is $(mi + 1) - i = (m - 1)i + 1$.

5 Height vs number of nodes

Recall that the level of a node is the number of edges in the path from it to the root. That is, the root has level 0. The height of a tree is the maximum level of any (leaf) node.

Now, suppose that we have a binary tree of height h. How many nodes and how many leaves does it contain? This clearly can't be an exact formula, since some trees are more bushy than others and some are more balanced

than others (all leaves at approximately the same level). But we can give useful upper and lower bounds.

To minimize the node counts, consider a tree that has just one leaf. It contains $h + 1$ nodes connected into a straight line by h edges. So the minimum number of leaves is 1 (regardless of h) and the minimum number of nodes is $h + 1$.

The node counts are maximized by a tree which is full (see above) and complete (all leaves are at the same level). In that case, the number of leaves is 2^h and the number of nodes is $\sum_{L=0}^{h} 2^L = 2^{h+1} - 1$.

So for a full, complete binary tree, the total number of nodes n is $\Theta(2^h)$. So then h is $\Theta(\log_2 n)$. If the tree might not be full and complete, this is a lower bound on the height, so h is $\Omega(\log_2 n)$. There are similar relationship between the number of leaves and the height.

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 that might not be a power of 2) while keeping all the leaves at approximately the same height. Balanced trees aren't as rigid as full binary trees, but they also have $\Theta(\log_2 n)$ height. This means that all the leaves are fairly close to the root, which leads to good behavior from algorithms trying to store and find things in the tree.

6 Tree induction

We claimed that

Claim 2 *Let T be a binary tree, with height h and n nodes. Then $n \leq 2^{h+1} - 1$.*

We can prove this claim by induction. Our induction variable needs to be some measure of the size of the tree, e.g. its height or the number of nodes in it. Whichever variable we choose, it's important that the inductive step divide up the tree at the top, into a root plus (for a binary tree) two subtrees.

> Proof by induction on h, where h is the height of the tree.
>
> Base: The base case is a tree consisting of a single node with no edges. It has $h = 0$ and $n = 1$. Then we work out that $2^{h+1} - 1 = 2^1 - 1 = 1 = n$.

Induction: Suppose that the claim is true for all binary trees of height $< h$, where $h > 0$. Let T be a binary tree of height h.

Case 1: T consists of a root plus one subtree X. X has height $h-1$. So X contains at most $2^h - 1$ nodes. And then X contains at most 2^h nodes, which is less than $2^{h+1} - 1$.

Case 2: T consists of a root plus two subtrees X and Y. X and Y have heights p and q, both of which have to be less than h, i.e. $\leq h - 1$. X contains at most $2^{p+1} - 1$ nodes and Y contains at most $2^{q+1} - 1$ nodes, by the inductive hypothesis. But this means that X and Y each contain $\leq 2^h - 1$ nodes.

So the total number of nodes in T is the number of nodes in X plus the number of nodes in Y plus one (the new root node). This is $\leq 1 + (2^p - 1) + (2^q - 1) \leq 1 + 2(2^h - 1) = 1 + 2^{h+1} - 2 = 2^{h+1} - 1$

So the total number of nodes in T is $\leq 2^{h+1} - 1$, which is what we needed to show. $\square$

7 Structural induction

Inductive proofs on trees can also be written using "structural induction." In structural induction, there is no explicit induction variable. Rather, the outline of the proof follows the structure of a recursive definition. This is sometimes simpler than trying to find an explicit integer induction variable.

Structural induction is used to prove a claim about a set T of objects which is defined recursively. Instead of having an explicit induction variable n, our proof follows the structure of the recursive definition.

- Show the claim holds for the base case(s) of the definition of T
- For the recursive cases of T's definition, show that if the claim holds for the smaller/input objects, then it holds for the larger/output objects.

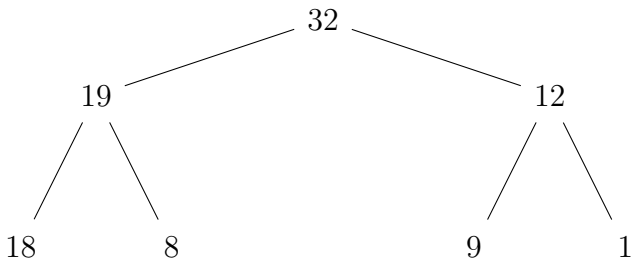
In Section 2, we saw a recursive definition for binary trees. To do structural induction on trees using this definition, we show that

- The claim holds for trees consisting of a single node.
- If the claim holds for trees A and B, it also holds for a new tree consisting of a root with A and B attached as its children.

- If the claim holds for tree A, it also holds for a new tree consisting of a root with A attached as its child.

8 Heap example

To have a nice claim to prove by structural induction, suppose we store numbers in the nodes of a full binary tree. The numbers obey the *heap property* if, for every node X in the tree, the value in X is at least as big as the value in each of X's children. For example:



Notice that the values at one level aren't uniformly bigger than the values at the next lower level. For example, 18 in the bottom level is larger than 12 on the middle level. But values never decrease as you move along a path from a leaf up to the root.

Trees with the heap property are convenient for applications where you have to maintain a list of people or tasks with associated priorities. It's easy to retrieve the person or task with top priority: it lives in the root. And it's easy to restore the heap property if you add or remove a person or task.

I claim that:

Claim 3 *If a tree has the heap property, then the value in the root of the tree is at least as large as the value in any node of the tree.*

To keep the proof simple, let's restrict our attention to full binary trees:

Claim 4 *If a full binary tree has the heap property, then the value in the root of the tree is at least as large as the value in any node of the tree.*

Let's let $v(a)$ be the value at node a and let's use the recursive structure of trees to do our proof.

> Proof by structural induction.
>
> Base: If a tree contains only one node, obviously the largest value in the tree lives in the root!
>
> Induction: Suppose that the claim is true for trees X and Y. We need to show that the claim is also true for the tree T that consists of a root node plus subtrees X and Y.
>
> Let r be the root of the whole tree T. Suppose p and q are the children of r, i.e. the root nodes of X and Y. Since T has the heap property, $v(r) \geq v(p)$ and $v(r) \geq v(q)$.
>
> Suppose that x is any node of T. We need to show that $v(r) \geq v(x)$. There are three cases:
>
> Case 1: $x = r$. This is obvious.
>
> Case 2: x is any node in the subtree X. By the inductive hypothesis $v(p) \geq v(x)$. But we know that $v(r) \geq v(p)$. So $v(p) \geq v(x)$.
>
> Case 3: x is any node in the subtree Y. Similar to case 2.
>
> So, for any node x in T, $v(r) \geq v(x)$.$\square$

In the inductive step, notice that we split up the big tree (T) at its root, producing two smaller subtrees (X) and (Y). Some students try to do induction on trees by grafting stuff onto the bottom of the tree. This frequently does not work, especially as you get to examples in more advanced courses. Therefore, we will take off points if you do it on homework or tests.

9 Structural induction with 2D points

Structural induction is not limited to trees. It can be used on any class of objects with a recursive definition. For example, consider the following recursive definition of a set S of 2D points:

1. $(3, 5) \in S$
2. If $(x, y) \in S$, then $(x + 2, y) \in S$

3. If $(x, y) \in S$, then $(-x, y) \in S$

4. If $(x, y) \in S$, then $(y, x) \in S$

What's in S? Starting with the pair specified in the base case $(3, 5)$, we use rule 3 to add $(-3, 5)$. Rule 2 then allows us to add $(-1, 5)$ and then $(1, 5)$. If we apply rule 2 repeatedly, we see that S contains all pairs of the form $(2n + 1, 5)$ where x is a natural number. By rule 4, $(5, 2n + 1)$ must also be in S for every natural number n.

We then apply rules 2 and 3 in the same way, to show that $(2m+1, 2n+1)$ is in S for every natural numbers m and n.

We've now shown, albeit somewhat informally, that every pair with odd coordinates is a member of S. But does every member of S have odd coordinates?

To show that all members of S have odd coordinates, we use structural induction.

> Proof by structural induction that all elements of S have both coordinates odd.
>
> Base: Both coordinates of $(3, 5)$ are odd.
>
> Induction: Suppose that both coordinates of (x, y) are odd. We need to show that both coordinates of $(x + 2, y)$, $(-x, y)$, and (y, x) are odd. But this is (even in the context of this course), obvious.

This proof would be complicated to write with standard induction. It's possible to find a standard induction variable n, but it's done in a somewhat obscure way: the "size" of an element x in S is the number of times you have to apply the recursive rules in S's definition in order to show that x is in S.