Trees

Zack Garza

May 2, 2014

Abstract

Contents

1 Background and Definitions **1**
1.0.1 Background 1
1.0.2 The Basics 1
1.0.3 Examples of Trees 3
1.0.4 Motivation 4

2 Rooted Trees **6**
2.0.5 m-ary Trees 7
2.0.6 Special Case: Ordered Binary Trees 8

Chapter 1

Background and Definitions

1.0.1 Background

A little bit of background: trees were first studied in the late 1800s by Arthur Cayley, made contributions geometry, analysis, and group theory. Cayley's Theorem, for example, showed that every group is isomorphic to a corresponding permutation group. He invented trees while trying to mathematically generate hydrocarbon isomers.

1.0.2 The Basics

A tree is a mathematical object that can be viewed as either a graph or a data structure.
At a bare minimum, it consists of elements that we refer to as *nodes* or *vertices* that are connected in some way. These connections are usually represented as paths or *edges* between the nodes. The edges themselves can have additional structure imparted upon them, such as specifying a direction between nodes, or assigning a numerical value or *weight* to each edge.
Definition: A tree is a connected graph that contains no simple circuits.
Note that G_4 does in fact contain two separate trees – appropriately enough, G_4 is referred to as a forest.

Notice that our definition of what constitutes a tree imparts a few guarantees about its structure. This is codified in our first theorem:

Theorem: An undirected graph is a tree $\iff$ there is a unique, simple path between any two vertices.
Proof: For the theorem to hold, it must be shown that

- If T is a tree, then there is a unique simple path between every two vertices.
- If G is an undirected graph with a unique, simple path between every two vertices, it is a tree.

(1) Assume we have a tree T. Then by definition, it is a graph, and it must be connected and contain no simple circuits.

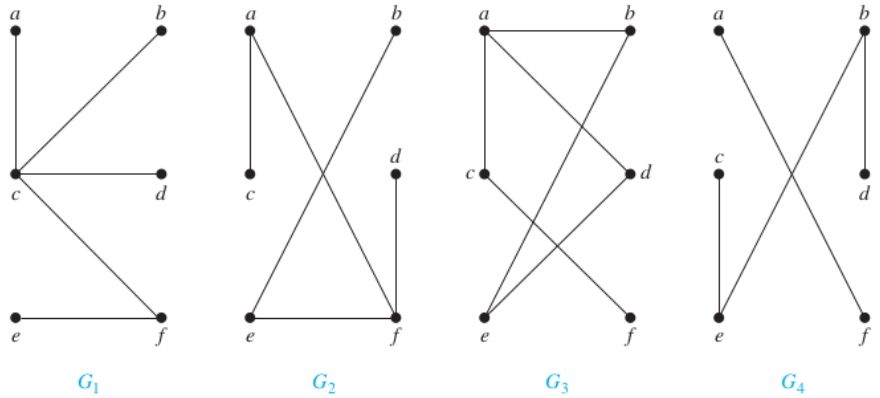

Figure 1.1: Several undirected graphs. G_1, G_2 are trees. G_3 contains a simple circuit, G_4 is not connected.

Pick two nodes, x and y. Because T is connected, there will always exist a simple path P_1 between x and y. But since T is a tree, there can be no simple circuits.

This means that the path between x and y must be unique.
Proof by contradiction: Assume there was another path, P_2 – if this path existed, it could be combined with P_1 and create a simple circuit, which contradicts the assumption that T was a tree.
So, If T is a tree, then there is a unique simple path between every two vertices.

(2) Assume we have a graph G such that G has a unique simple path between any two vertices.
Then G must be connected, because there exists some path between every node and every other node, and the first part of our definition of a tree is satisfied.

It follows that G must also contain no simple circuits.
Proof by contradiction: Suppose G did have a simple circuit containing nodes x and y. This implies that there are at least two simple paths between x and y. But this contradicts our assumption that G has only unique simple paths between any two vertices.
So, G is connected and contains no simple circuits, meaning G is a tree.

A useful consequence of this theorem is that the number of edges and vertices in a tree are always related:
Theorem: A tree with n vertices has $n - 1$ edges, where $n \in \mathbb{N}$.
Proof: We utilize choice in this proof to distinguish nodes and pick an arbitrary root for the tree, and then proceed by induction.

Let P_n be the statement of the theorem.
Basis: When $n = 1$, P_1 states that a tree with 1 vertex has 0 edges, which is trivially true.

Induction: Assume P_n. Then, P_{n+1} states that a tree with $n+1$ vertices has n edges.
Consider such a tree, T. Since T is finite, it must have a leaf. Remove this leaf to produce a tree T', which must still be a tree because it remains connected and contains no simple circuits.
Since T' now has n nodes, by the inductive hypothesis, it then has $n-1$ vertices. Adding one vertex requires adding one edge, which acts on T' to transform it back to T, which must have $n+1$ vertices and n edges.

Thus, $P_n \rightarrow P_{n+1}$.

1.0.3 Examples of Trees

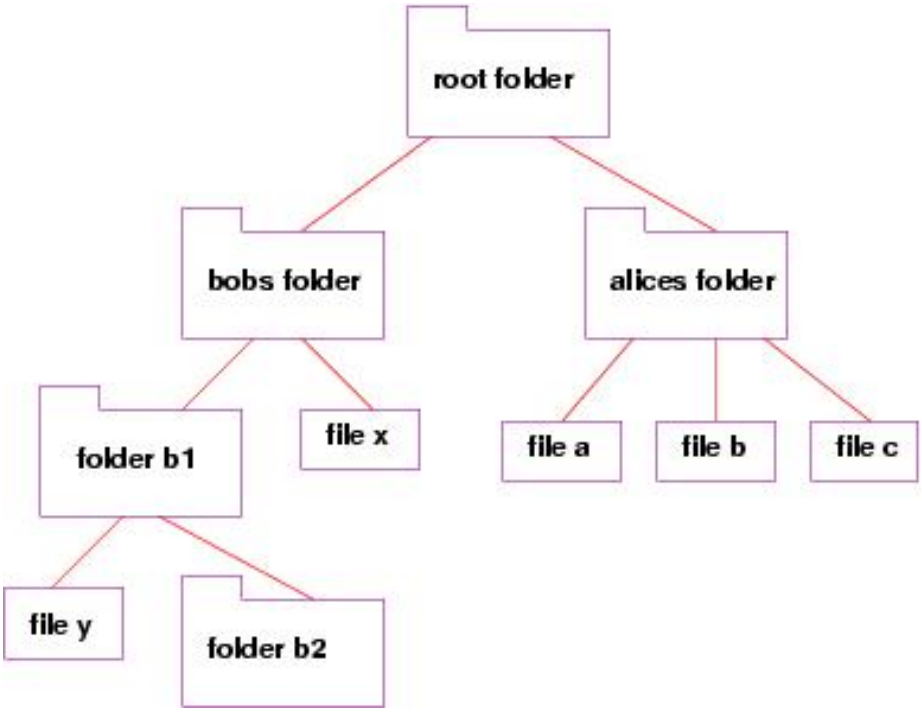

Figure 1.2: Tree representation of a directory structure, corresponding to an undirected graph.

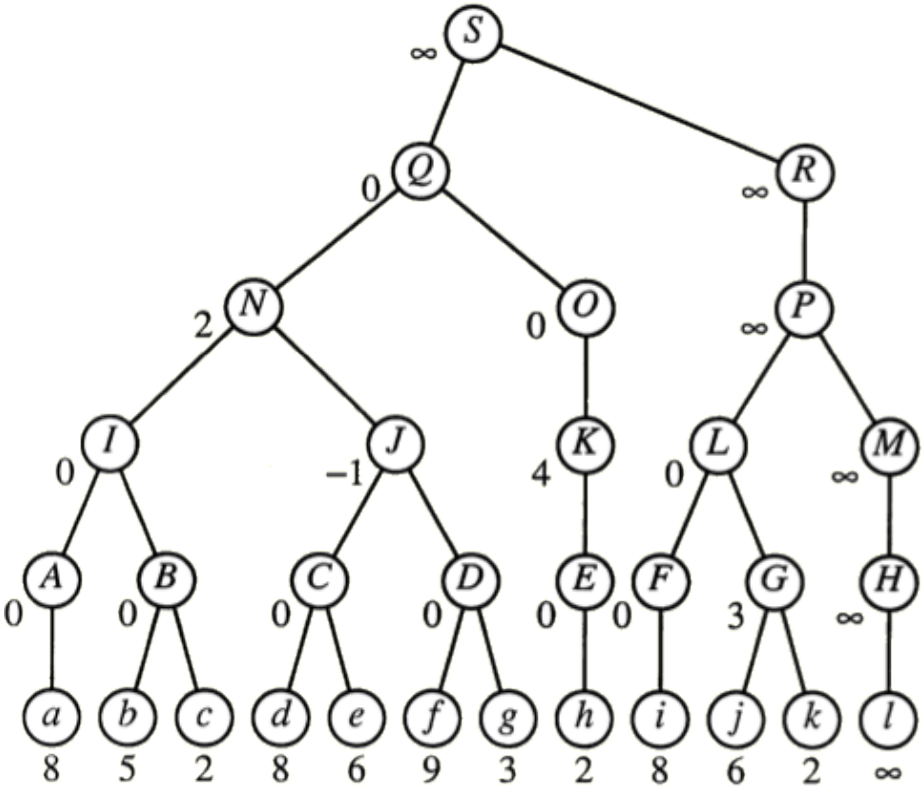

Figure 1.3: Tree with weighted edges, again undirected.

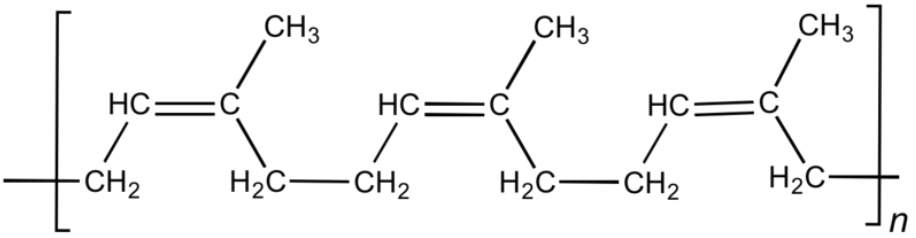

Figure 1.4: Cis-polyisoprene, the main constituent of natural rubber, and many other hyrdocarbons can be represented as connected graphs, and thus trees as well. Nonisomorphic trees with the same number of nodes correspond to isomers.

1.0.4 Motivation

What are trees useful for?

1. Game state trees (checkers, chess, etc.) - nodes represent states, and edges represent valid moves.

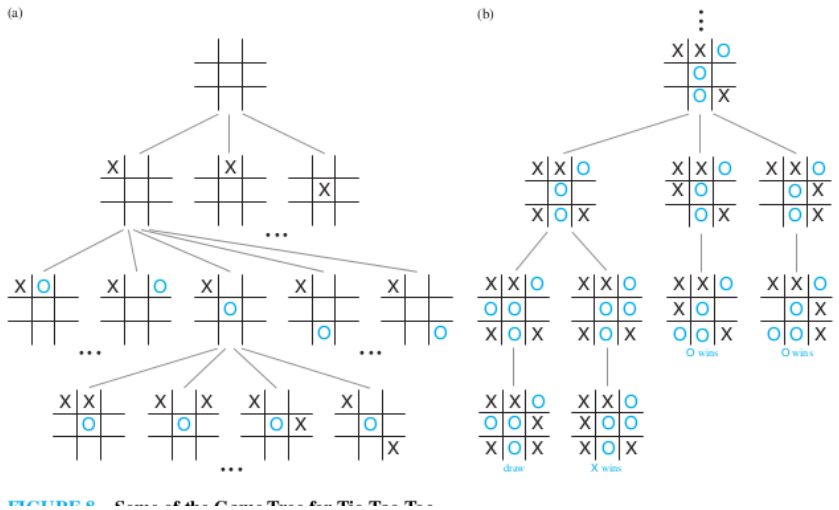

Figure 1.5: ?????

2. Huffman encoding - Given a large amount of text, it can be compressed into a tree based on symbol frequency. Used in jpeg and mp3 compression.

3. Decision trees

4. Efficient, structured storage - certain types of trees are guaranteed to be bounded by $O(\log n)$ (as long as they are balanced, as in certain B-trees or Red-Black trees.)

5. Form the basis of heaps, which are used to implement priority queues. These are used for operating system scheduling processes, quality-of-service in routers, AI path-finding algorithms, etc.

Most things that can be represented in a hierarchy can be modeled as trees. From a practical perspective, they are also very easy to implement. They can be defined recursively, and have no theoretical upper limit to the amount of nodes.

What are trees not so good for?

Generally, any graph that contains loops. For example, a circuit that could be represented as a tree would have no current.

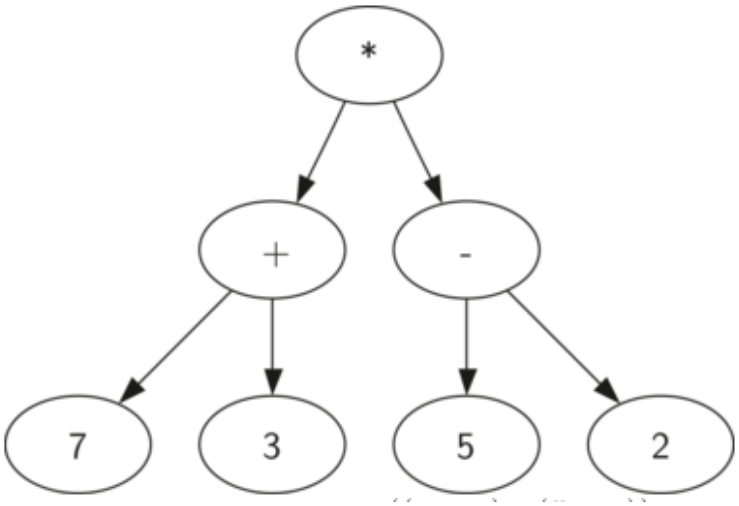

Figure 1.6: A tree that parses mathematical operators that preserves their ordering.

Chapter 2

Rooted Trees

Definition: A *rooted tree* is a tree in which one vertex is designated as the root, and every edge is directed away from the root.
This means that there is no predefined notion of what the root is, as long as there is consistency in direction.

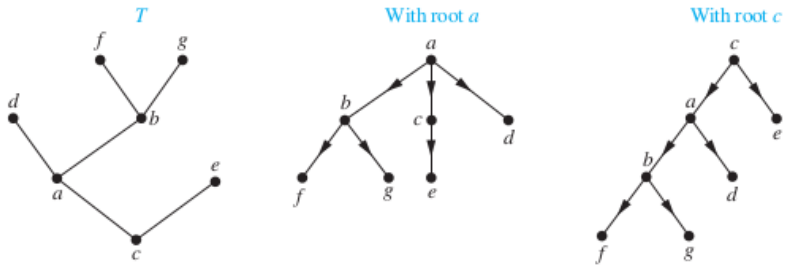

Figure 2.1: Changing the root of T from a to c. Notice that the direction between a and c is reversed when the roots are changed.

Terminology to know:

1. Parent - If v is a vertex, the parent p is the unique vertex such that $p \Rightarrow v$.
2. Child - If p is the parent of v, v is the child of p
3. Sibling - Vertices with the same parent.
4. Ancestor - Any vertex on a path between v and the root.
5. Descendant - Vertices for which v is an ancestor.
6. Leaf - Vertex with no children
7. Internal Vertex - Any vertex with children.
8. Subtree - Subgraph consisting of a as its root, including all of its descendants.

9. Level (*or height*) - Number of edges along the longest path between a node and a leaf. i.e., a local measure of how far a vertex is from a leaf.

   Leaves are defined to have a level of 0.

10. Depth - Number of edges between a vertex and the root node, i.e. a measure of distance to the root node.

    Root vertices are defined to have a depth of 0

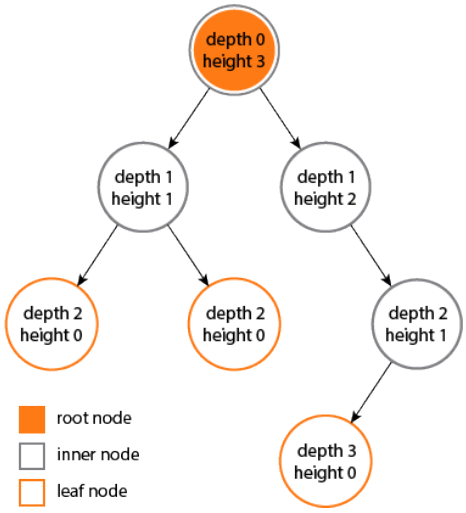

Figure 2.2: Root nodes always have a depth of 0, and leaf nodes will always have a height of zero.

2.0.5 m-ary Trees

Definition: A rooted tree is called an *m-ary tree* is every internal vertex has at most m children, and is said to be *full* if every internal vertex has exactly m children.
Note that the definition of full only applies to *internal* vertices! If we counted leaves, no finite tree would ever fit this definition.

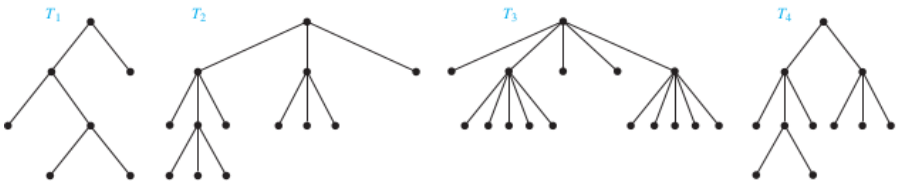

Figure 2.3: Are these trees full m-ary trees from some $m \in \mathbb{N}$?

2.0.6 Special Case: Ordered Binary Trees

Defined as an m-ary tree with $m = 2$, i.e. that every node has two children. There is also an additional restriction that Commonly used as a data structure, and yields several nice properties. **Definition:** An m-ary tree is *complete* if every level (but not necessarily the last) is full, and all nodes on the bottom are on the left.

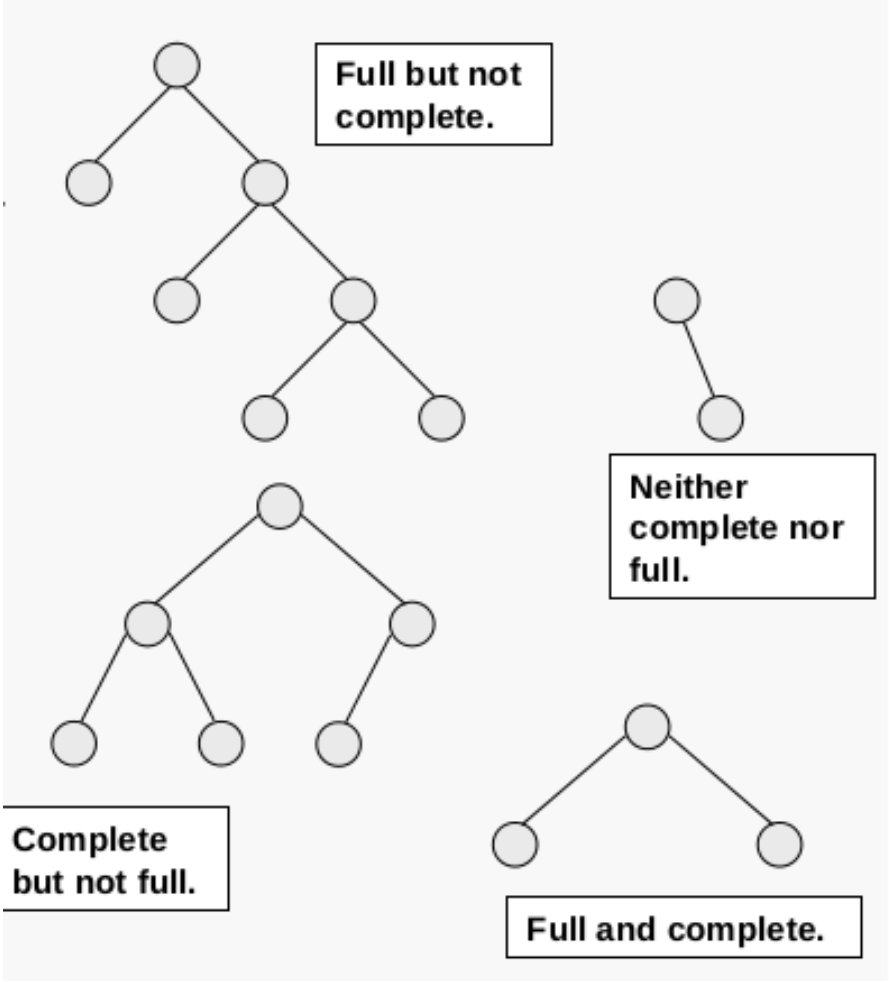

Figure 2.4: Distinction between full and complete m-ary trees (in this case, binary).

How do we know that lookup in $O(\log n)$? Well, suppose we have a binary tree B, with a maximum depth of d, and n nodes.
To find an arbitrary node in B, we need to make at most d comparisons – one at each level of the tree.
But how is this related to the number of nodes in the tree? Well, we count the

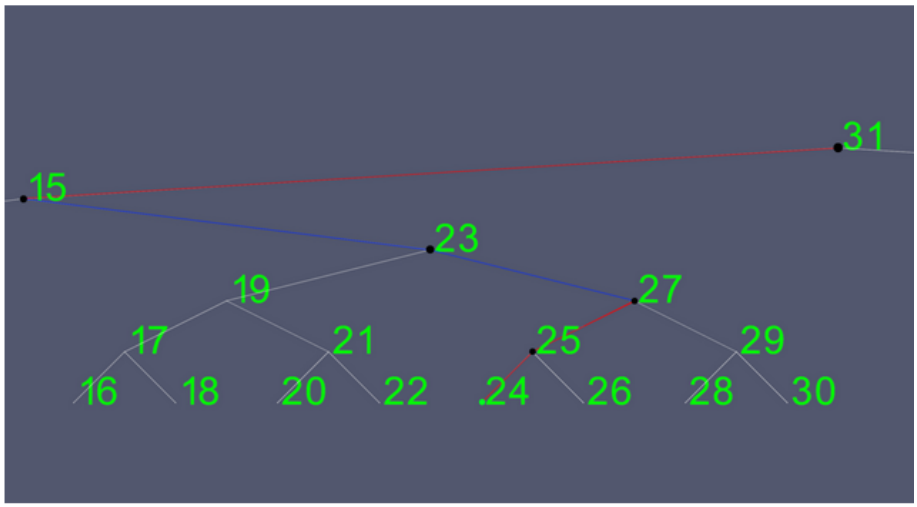

Figure 2.5: Binary trees provide $O(\log n)$ access to arbitrary elements.

nodes and see that

$$n \leq 1 + 2 + 4 + 8 + \cdots + 2^d = \sum_{k=0}^{d} (1)2^k = (1)\frac{1 - 2^{d-1}}{1 - 2} = 2^{d-1} - 1$$
$$\Rightarrow n \leq 2^{d-1} - 1.$$

That is, that the maximum number of nodes in a tree is bounded by the depth in a particular way. Solving for d, which the maximum number of comparisons needed, we see that

$$\begin{aligned} n + 1 &\leq 2^{d-1} \\ \Rightarrow \ln(n+1) &\leq (d-1)\ln 2 \\ \Rightarrow \frac{\ln(n+1)}{\ln 2} &\geq d - 1 \qquad (\text{where } \ln 2 < 0) \\ \Rightarrow \log_2(n+1) + 1 &\geq d \\ \Rightarrow d \leq \log_2(n+1) + 1 &\leq \log_2(n+1) \leq \log_2(n) \\ \Rightarrow d &\in O(\log_2 n). \qquad \square \end{aligned}$$

Chapter 3

Applications

3.0.7 Binary Search Trees