Trees

Zack Garza

May 7, 2014

Abstract

Contents

Chapter 1

Background and Definitions

1.0.1 Background

So, today I'm going to talk to you guys about trees, and hopefully give you guys some insight into why they're a particularly nice object to work with.

- To start off, we won't actually define a tree at first, but we'll talk a bit about why they were "invented" in order to address the question of why we would even need something like a tree in the first place.
- Then, we'll talk about what a tree really is, plus when and where they might be useful.
- Then we'll go over a few of the interesting properties that trees have, as well as a few different variants of them.
- And lastly, we'll cover some of the ways trees can be applied to problems.

So first, for a little bit of background – trees were first studied in the late 1800s by Arthur Cayley, who made contributions to geometry and analysis. For those of you that might go on to take further courses in math, you'll very likely come across Cayley's Theorem in a class that covers groups (which is a topic in abstract algebra). You'll spend a lot of time studying things called "permutation groups", and Cayley's Theorem showed that theorems which were true for permutation groups could be extended to be true for all groups in general.

Much of Cayley's work revolved around studying which properties of objects were invariant under some condition – that is, after applying some kind of transformation to the object, the property remained the same and was in a somehow fundamental to that object's structure.

For example, consider rotating a circle in the plane – this does not change the set of points contained within the circle. Or consider switching rows and columns of a matrix – the set of elements inside of the matrix is the same.

So where do trees come into this? Well, Cayley invented trees while working with chemical structures – in particular, he was trying to find a way to mathematically generate the structure of hydrocarbons. (See Figure).

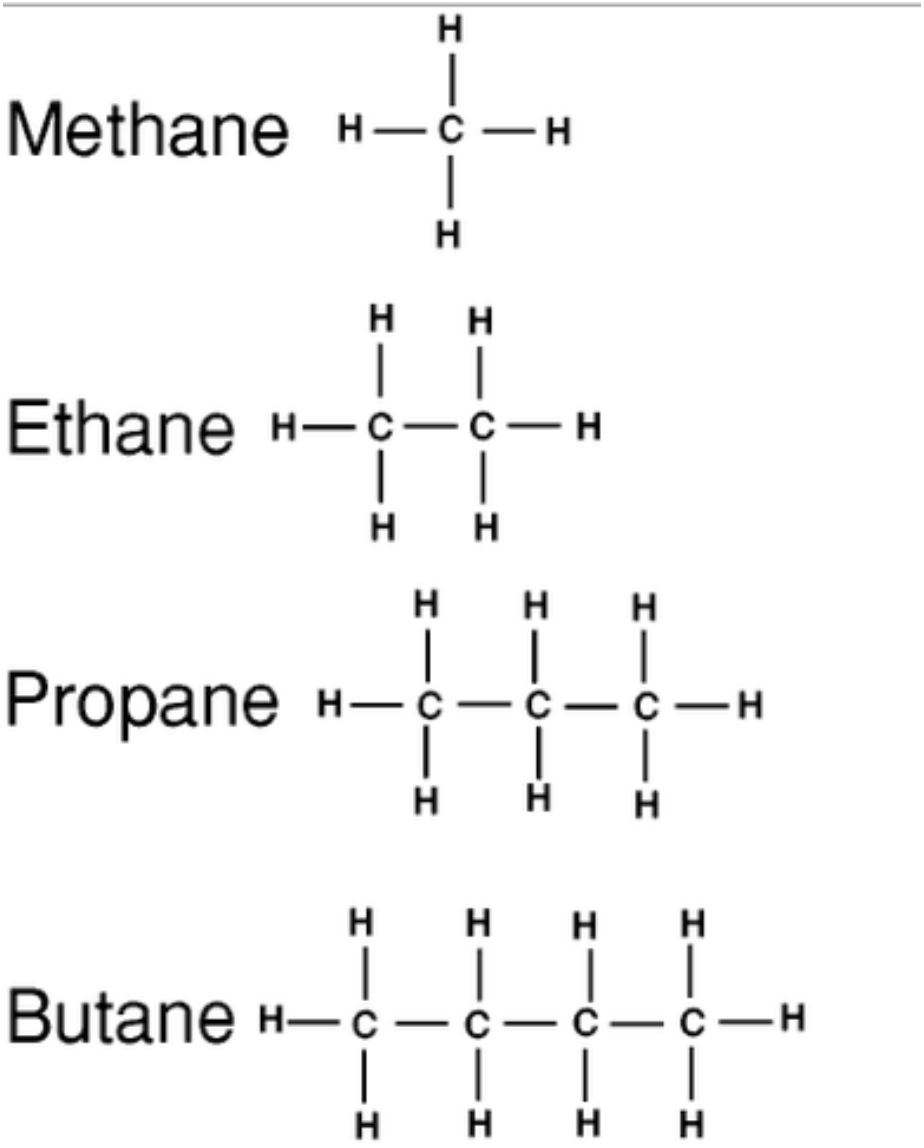

Figure 1.1: A series of Alkanes - one C always bonds to 4 other atoms, and the structure always conforms to the relationship $H_n - C_{2n+2}$

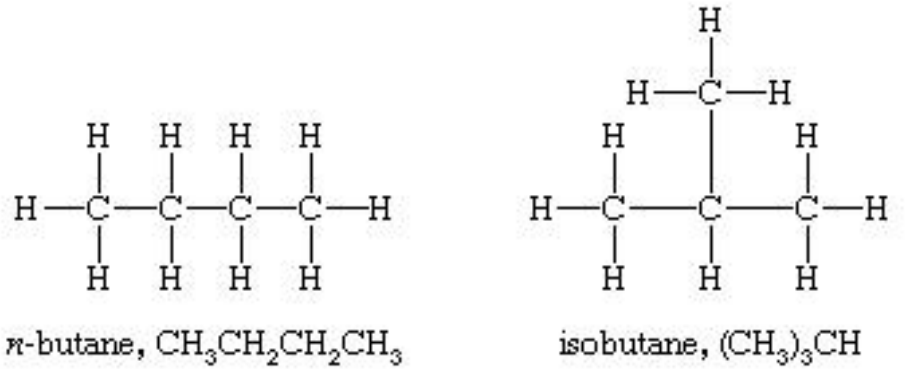

Figure 1.2: Applying a transformation (in this case, relocating a bond to create an isomer) does not affect the number of H and C atoms, nor their relationships.

One thing to notice about the alkane series above is that for $n \leq 3$, no matter how we many ways we "swap" the atoms, we can always find a way to make our 'swapped" version look like one of the given versions. But for $n \geq 4$, something interesting happens – now, we have a way of swapping atoms that really has changed the structure. And if you're doing chemistry with these compounds, this is something you might worry about – although the number of H and C atoms is the same, we know that these kinds of structural changes can cause the chemical to react in an entirely different way!

In chemistry, we might call these two forms "isomers", but Cayley's observation was that something more general was going on. If these could be formulate as mathematical objects, then it might be said that isobutane was a permutation that wasn't *isomorphic* to the rest of the permutations on 4 atoms. And hydrocarbon chains can easily extend into 100s of atoms - then what could we say about an an arbitrary set of n atoms? And so these were the kinds of problems that "trees" were invented to address, but we'll see in a moment that their use has spread far beyond that.

1.0.2 The Basics

So now we need to define what exactly we mean when we talk about trees. We'll have to go through a bunch of definitions first, so let's get to it!

A tree is a mathematical object that can be viewed as a graph, or in computer science we might consider it an "abstract data structure".

At a bare minimum, it consists of elements that we refer to as *vertices* that are connected, and these connections are usually represented as paths or *edges* between the vertices. The edges themselves can have additional structure, such as specifying a direction between vertices, or assigning a numerical value or *weight*

to each edge.

We also impose two other conditions, but we'll need a few definitions from graph theory to do so:

Definition. Define a graph G to be a set of nodes and vertices.
Definition. Given a graph G, a path $(v_1, v_2, v_2, \cdot v_n)$ is called a *cycle* if $v_1 = v_n$. A graph that does not contain a cycle is said to be *acyclic*.
Definition. A graph G is called *connected* if for every pair of vertices, (v, w) there exists a path between them.
Definition: A graph G is a tree if G is connected and acyclic. This is equivalent to saying that for every pair of vertices (v, w), there exists a unique path between them.

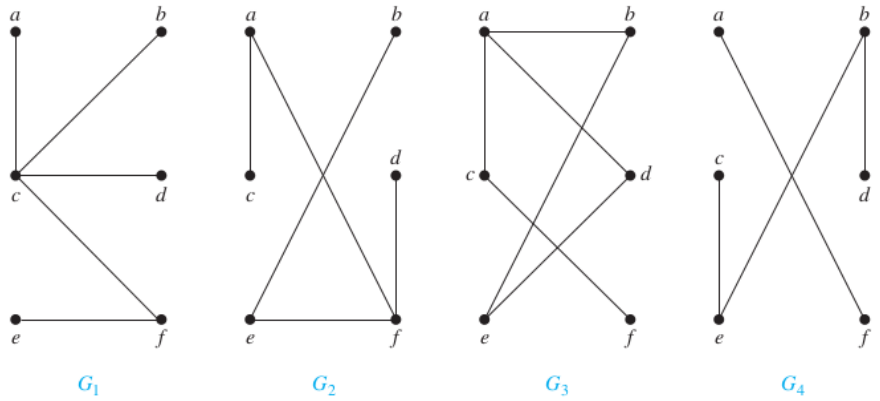

Figure 1.3: A few undirected graphs: which are trees? G_1, G_2 are, but G_3 contains a simple circuit and G_4 is not connected.

Note that G_4 does in fact contain two separate trees – appropriately enough, G_4 is referred to as a forest.

1.0.3 Examples of Trees

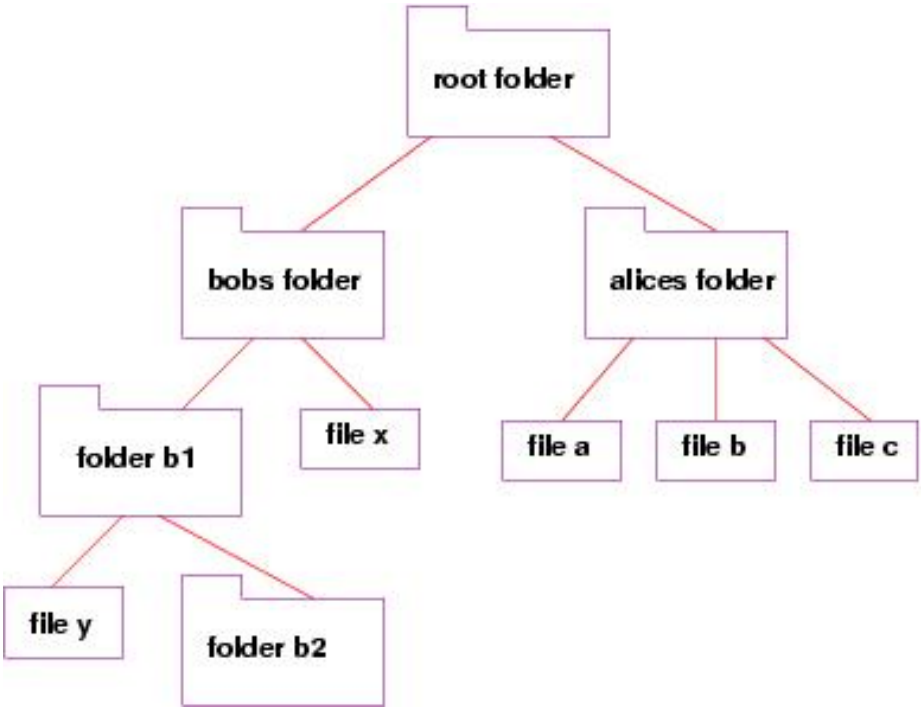

Figure 1.4: Tree representation of a directory structure, corresponding to an undirected graph.

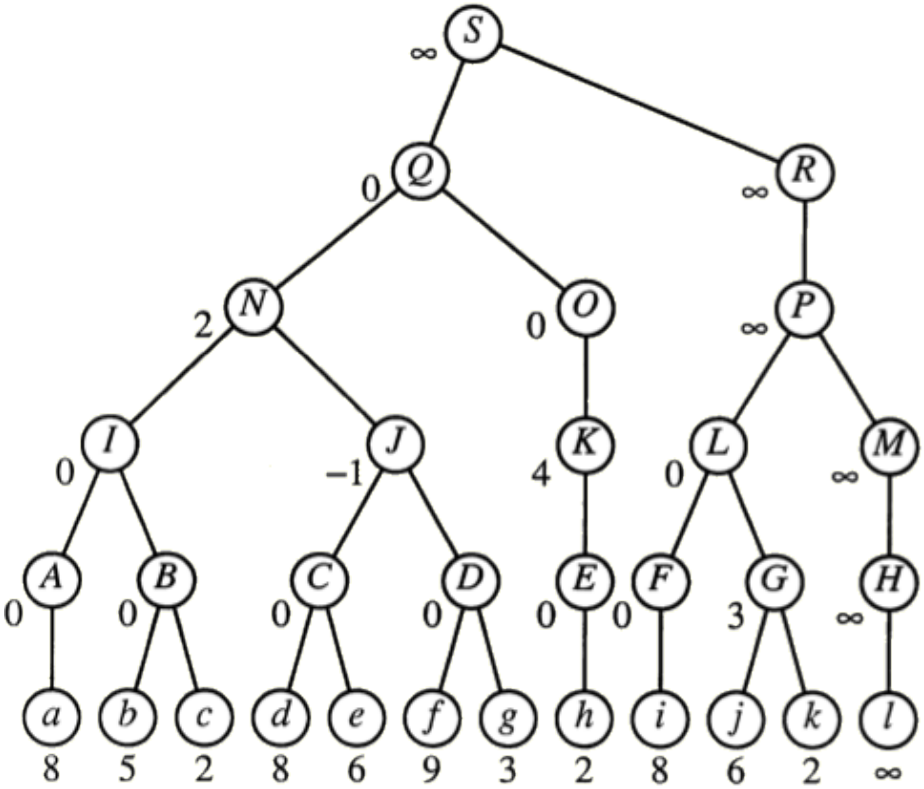

Figure 1.5: Tree with weighted edges, again undirected.

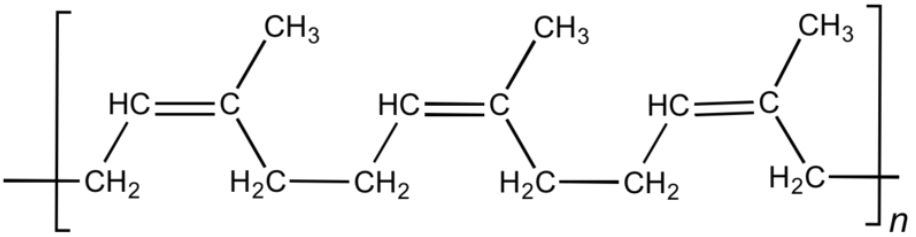

Figure 1.6: Cis-polyisoprene, the main constituent of natural rubber, and many other hyrdocarbons can be represented as connected graphs, and thus trees as well. Nonisomorphic trees with the same number of vertices correspond to isomers.

1.0.4 Motivation

What are trees useful for?
Used widely in computer science (which explains the abundance of arboreal puns.)

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

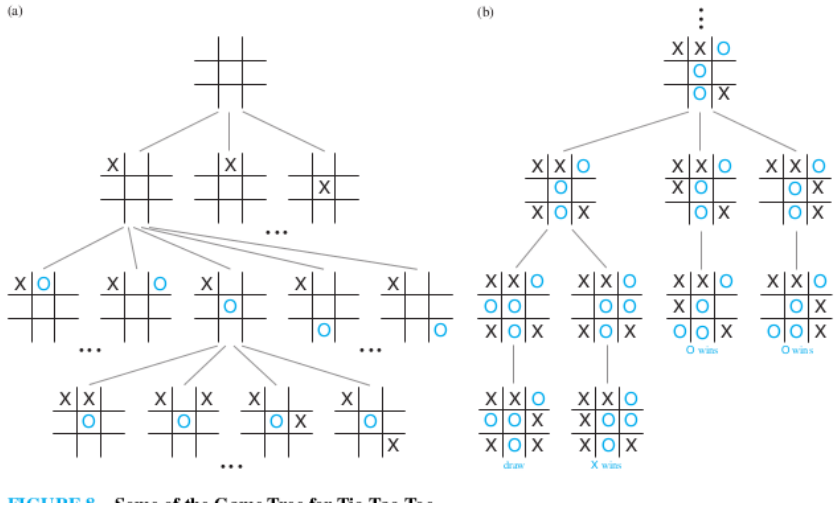

Figure 1.7: Portion of a game state tree from Tic-Tac-Toe

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 vertices. They are also amenable to proofs by induction.
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.
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.

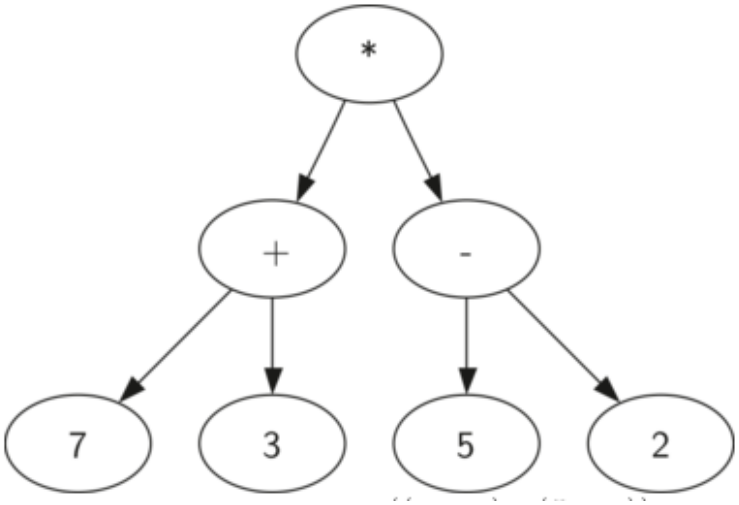

Figure 1.8: A tree that parses mathematical operators that preserves their ordering.

- 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.
Pick two vertices, 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 vertex and every other vertex, 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 vertices 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 vertices 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 vertices, 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, $P_n \to P_{n+1}$.

In practice, when we talk about and use trees, we give them a little more structure. We usually define a *partial ordering*. We define this with the binary relation $v \leq w$ if v is on a path between the root and w.

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.

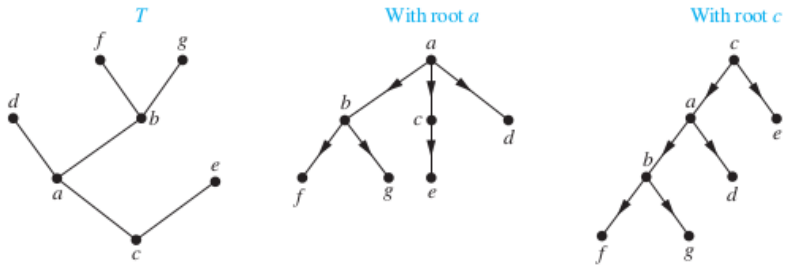

Figure 2.1: Changing the root of T from a to c. Notice that the direction between a and c is reversed when the roots are changed.

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 vertex 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 vertex, i.e. a measure of distance to the root vertex.

    Root vertices are defined to have a depth of 0

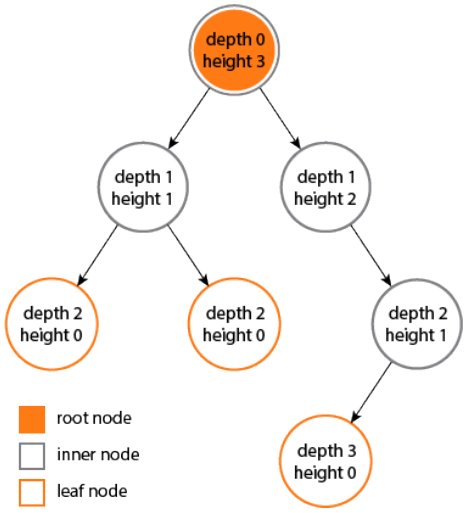

Figure 2.2: Root vertices always have a depth of 0, and leaf vertices will always have a height of zero.

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.

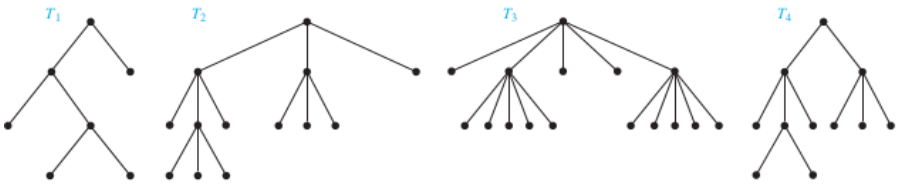

Figure 2.3: Are these trees full m-ary trees from some $m \in \mathbb{N}$?

2.0.6 Special Case: Ordered Binary Trees

Defined as an m-ary tree with $m = 2$, i.e. that every vertex 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 vertices on the bottom are on the left.

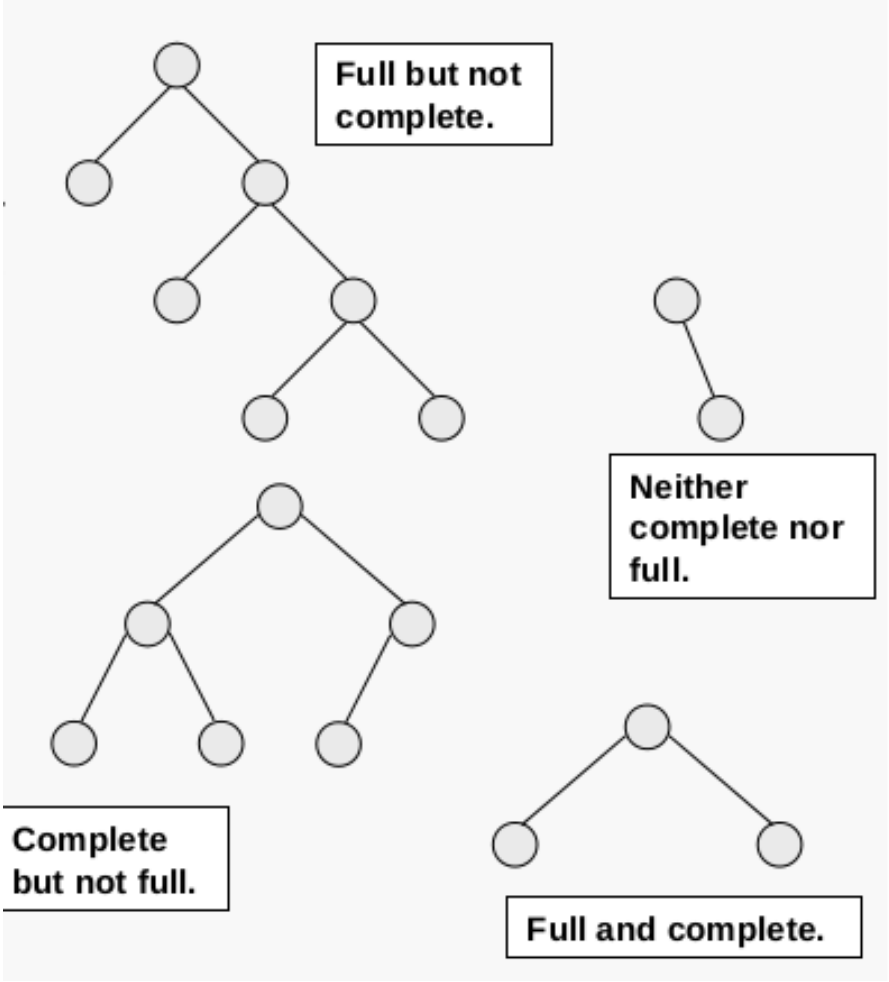

Figure 2.4: Distinction between full and complete m-ary trees (in this case, binary).

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 vertices.
To find an arbitrary vertex 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 vertices in the tree? Well, we count

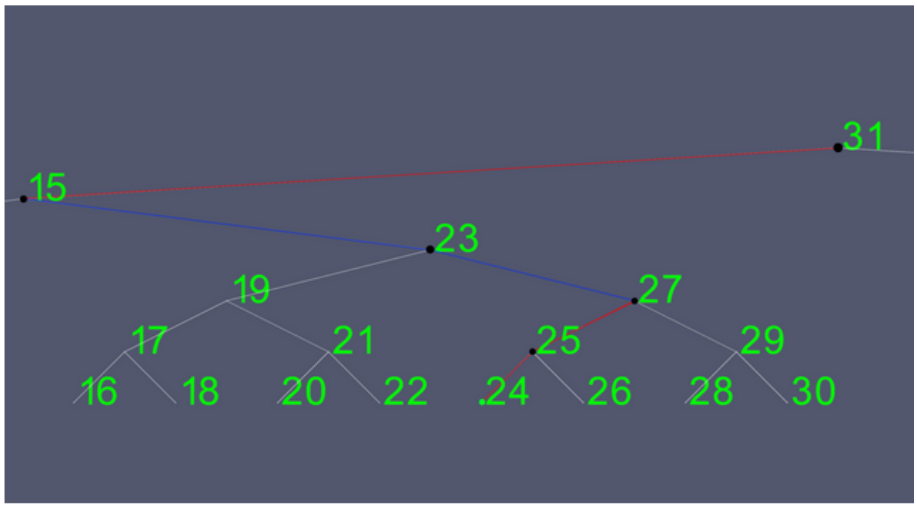

Figure 2.5: Binary trees provide $O(\log n)$ access to arbitrary elements.

the vertices 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 vertices 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

Left subtree ¡ root, right subtree ¿ root. – Enumerate types of graphs – Digraphs? – Spanning trees – Minimum spanning trees – Depth/Breadth Search – Sums of subsets – Web spiders – Traversals (Easy to define recursively with subtrees) – Preorder Root, left, right Polish notation – Postorder Left, right, root Reverse-Polish/Postfix notation – Inorder (Only for Binary trees) Left, root, right Infix notation (ambiguous, parentheses needed) Compare to pre/post - only one way to parse – Arithmetic Parsing – Infix – Prefix – Postfix –Matrix representation
http://www.geeksforgeeks.org/graph-and-its-representations/