Trees

Trees are the first **non linear** data structure that we look at.

Note 0.1

Characteristics of Trees

1. No cycles, so you can't reach a node using itself
2. Highly recursive
3. Usually implemented with linked lists rather than arrays (arrays get messy)

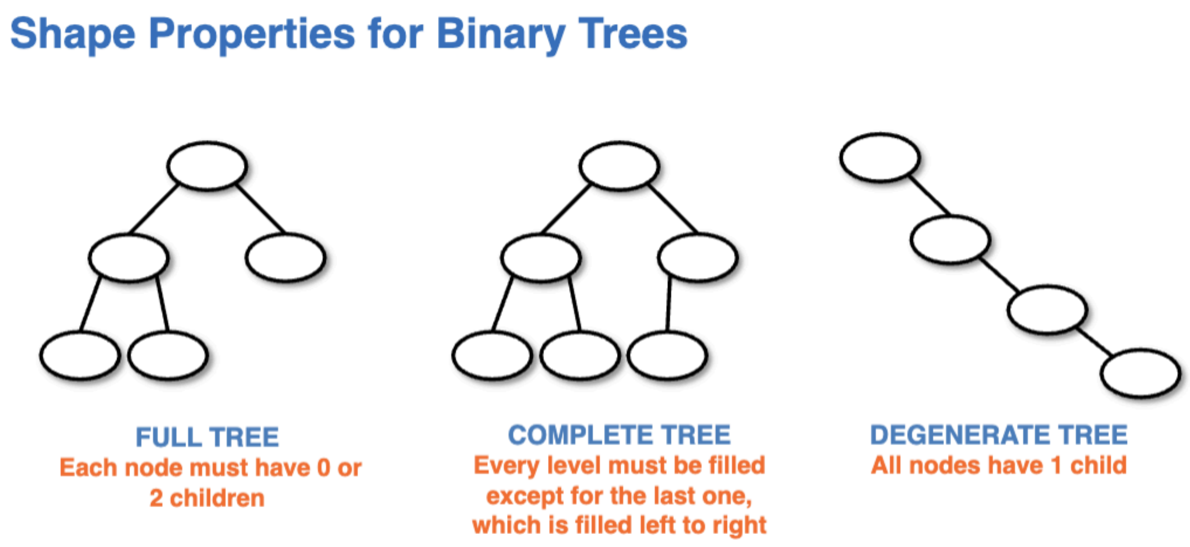
A tree is **full** if each node has exactly 0 or 2 kids, it cannot have just 1.

A tree is **complete** if each level except the last has the maximum number of nodes (2^n w/ root has n = 0)

- The last level must be filled left to right, but doesn't need to be full

A node is **balanced** if its children heights differ by only 0 or 1 (height of missing children = −1). Thus, a tree is **balanced** if all of its nodes are balanced.

- Leaf nodes are always balanced


Shape Properties for Binary Trees
FULL TREE
Each node must have 0 or 2 children
COMPLETE TREE
Every level must be filled except for the last one, which is filled left to right
DEGENERATE TREE
All nodes have 1 child

Theorem 0.1

Important Terminology

Root: the head/entry point of the tree

Children: what the nodes point towards (can have multiple children, each node can only have one parent though)

External/Leaf Nodes: nodes without children

Internal Nodes: nodes with children

Definition 0.1

The **depth** of a node is the distance of it from the root, or how many nodes away it is(?)

The **height** of a node is the distance of a node from the furthest leaf node
- $\text{height} = 1 + \max(\text{height}(\text{children}))$

Binary Trees

Note 0.2

Characteristics of Binary Trees
- Shape: each node can have at most 2 children
- Children are labeled left and right (left precedes right)

Theorem 0.2

Iterating through a binary tree

```
public void traverse(Node node) {
  if (node != null) {
    traverse(node.left);
    traverse(node.right);
  }
}
```

Binary Search Trees

Definition 0.2

Binary Search Trees are Binary Trees with the given rule: any child node to its left must be less than that node, and any child note to the right must be greater than it. This applies for all children nodes, not just direct children.

The motivation behind BST's are the **binary search algorithm**, because we are able to split the search space in half each operation, making searching **O(logn)**.

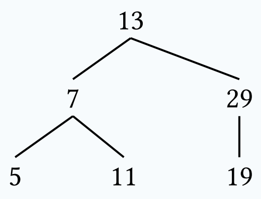
Pre Order Traversal

Definition 0.3

In **preorder traversal**, we look at each node and then traverse left, then right. If you draw a "glove" around the tree, you would print a node every time you touch the left side of the node

```
preorder (Node node):
  if node is not null:
    look at the data in node
    recurse left
    recurse right
```

Example 0.1



In this case, our traversal would print *13 7 5 11 29 19*

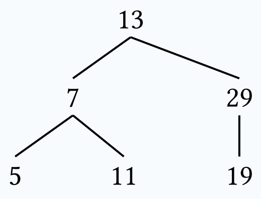
Post Order Traversal

```
ppreorder (Node node):
  if node is not null:
    recurse left
    recurse right
    look at the data in node
```

Definition 0.4

In **postorder traversal** we first recurse left and right before looking at the data in the node. So if you wrap around the tree like a glove, every time you reach the right side of a node, you print.

Example 0.2



In this case, our traversal would print *5, 11, 7, 19, 29, 13*

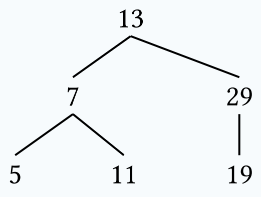
In Order Traversal

Definition 0.5

In **inorder traversal**, we first recurse left, then look at node, then recurse right. So if you wrap around the tree like a glove, every time you reach the bottom of a node you print it.

```
preorder (Node node):
  if node is not null:
    recurse left
    look at the data in node
    recurse right
```

Example 0.3



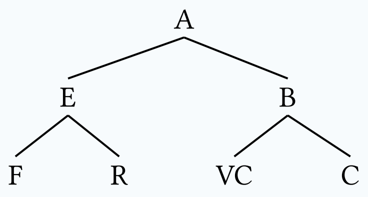
In this case, our traversal would print *5, 7, 11, 13, 19, 29*

Level Order Traversals

Definition 0.6

Level Order Traversal: print all the nodes at depth 0, then depth 1, then depth 2, etc

Example 0.4



In this example, we would print *A E B F VC C* in that order