## Trees

## Recap

- So far we've seen “sequence containers” and touched on “Sets”
- For sets there are no releationships between elements (internally we sorted stuff, but none of the public methods exposed those relationships)
- Sequence containers defined “comes before + comes after” relationships between elements
- That's too limiting for some types of data

## Hierarchical Data

- Data is often described hierarchically
    - Folders on your filesystem (higher folders contain lower folders + files)
    - Managers manage workers, VPs manage managers, etc
    - Dogs are a type of mammal, mammals are a type of animal, etc

## Trees

- Trees are a linked data structure used for representing hierarchical data
- We'll look mostly at an implementation using a linked data structure, but that's not the only day to store trees
- Like linked lists, we'll define a node type for our trees
- Unlike linked lists, each node may have multiple links
- Trees may not have “loops” (if there are loops it's a “graph”)

## Terminology

- The top of a tree is called the “root” node (CS folks draw trees upside down)
- The nodes directly below a node are its “children”
- Nodes anywhere below a node are its “descendents”
- The node (one node, singular) directly above a node is its “parent”
- Any node above a node is an “ancestor”
- Nodes with the same parent are “siblings”
- Nodes without children are “leaf nodes”
- Nodes with children that aren't the root are “internal nodes”

## More terminology

- Any node and its descendents is a “subtree.” It's a tree that's part of another tree. We'll often talk about “the subtree rooted at n” which means to look at the tree from that node and its descendents
- The depth of a node is the number of ancestors it has (same as the number of links to trace to get to the root). The root has depth 0
- the height of a tree (or subtree) is the maximum depth of any node
- An “ordered” tree is one in which there is some sort of ordering between siblings. We'll see lots of ordered trees
- An “unordered tree” is the opposite. An “org chart” at a company is unordered there is no ordering between employees with the same manager

## Pointer based tree implementation

```
class Tree:
    Node root

    class Node:
        Data
        Node(s) children
        //could be an array, some other collection, etc
        Node parent // often we don't explicitly store this!
```

## Binary Trees

- Trees where each node has 0, 1, or 2 children are extremely common (we've seen a bunch already!)
- We call these binary trees, and we call a node's children “left” and “right”
- If a child is “missing” we set the corresponding pointer to null
- Leaf nodes have left == right == null
- A “Full” tree is one where every node except for the leaves has 2 children
- A “Complete” tree is one where every “level” is full except for the lowest. The last row must be filled in “from the left”

## Traversals

- Often we want to “visit” every node of a tree, and we call this operation a “traversal”
- Traversing a binary tree involves performing 3 steps at each node:
  - Visit this node
  - Visit its left child
  - Visit its right child
- We can order the 3 to get different traversal orders

## Pre-Order Traversal

- At each node we do : visit(this), visit(left), visit(right)
- Pre means “this before its children”
- Useful is we want to pass information from the root down

## Post-Order Traversal

- Opposite of preorder: visit(left), visit(right), visit(this)
- An example of when we'd do this is to compute the height of a tree:
- height of this node = max(left height, right height) + 1

## In-Order Traversal

- visit(left), visit(this), visit(right)
- Why this is called “in order” will become clear Monday

## Example: Expression Trees

- Leaf nodes are values
- Internal nodes are binary operations acting on values (*, -, +, etc)
- What traversal should we use to “evaluate” an expression tree?