## Computer Science 136

Data Structures<br>
Lecture #19 (November 1, 2021)

1. Announcements.
   (a) Pre-registration is this week. Pre-registration is necessary to get into CS classes.
   (b) Comments on the current lab.
   (c) Questions?
2. Trees.
   (a) A *tree* is a recursively defined structure: dataless end-nodes, or a structure that contains a single data element and points to an ordered list (a *forest*) of other trees (called *subtrees*).
   (b) Not cyclic.
   (c) Terminology: *root*, *leaf*, *interior node*, *ancestor*, and *descendant*.
   (d) Terminology: *degree* (or *arity*), *full node*, *binary tree*, *height*, *depth* (or *level*), *full tree*, and *complete tree*.
3. Binary Tree implementation.
   (a) Not a `Structure`.
   (b) First, notion of a *dummy node*, or *sentinel*. Empty trees are empty nodes – nodes with no data – so that we may call methods on them. The other option: null references for empty trees, but you can't call methods on null pointers, and this leads to significant numbers of tests for null pointers.
   (c) Each node maintains a data value (`null` in empty trees), a parent, and two children (left and right).
   (d) Three constructors: no parameters (empty tree), one parameter (leaf), two parameters (interior node).
   (e) Methods: `isEmpty`, `value`, `setValue`, `left/right`, `setLeft/Right`, `isLeft/RightChild`, `parent`, `setParent`.
   (f) N.B. `setLeft/Right` re-parent the new child's parent pointers.
   (g) Is an iterable (has an `iterator` method). How would you traverse a tree's nodes?
4. Example: Infinite questions.
5. Since the structure is recursive, many methods are recursive as well.
   (a) `size` – count of nodes in tree.
   (b) `height` – length of longest path.
   (c) `root` – root of this tree.
   (d) `depth` – length of path to root.
   (e) `isLinear` – (yet to be written) is degree always less than 2?
   (f) `isFull` – is it "triangular".
   (g) `isComplete` – is it "almost triangular".
6. Traversals – a basis for iteration.
   (a) Inorder. The root appears after everything in left subtree and before right.
   (b) Preorder. The root appears before left, which appears before right.
   (c) Postorder. The root appears last, after left then right.
   (d) Levelorder. Top to bottom, left to right.
7. Iterators – Tricky. It's all in the choice of underlying data structure.
   (a) Inorder. At every stage, the current node (top on stack) and its left subtree have been traversed. A stack keeps track of roots of all trees not yet fully traversed.
   (b) Preorder. At every stage, top item of stack is current. Popping pushes right subtree, then left.
   (c) Postorder. At every stage, top item is current (subtrees have been done), and lower items are ancestors.
   (d) Levelorder. Current node is at head of queue. When dequeuing, add subtrees to queue.

---

**Notes:**