Trees Day 3 (probably a little behind)

Goals for today:

* Illustrate diff btw MAP and LIST (get class to pick between with examples/situations)
* Re-introduce linked node structure of a binary tree.
* Illustrate examples of recursive code that traverses a binary tree.
  + But first do this with linked lists (recursive code to say print out LL forwards and backwards, add up all elements).
  + With arrays and lists, traversing the list is relatively straightforward. (forwards/backwards) for trees, there are 3 common ways.
  + Preorder, inorder, postorder with operator tree.
* Introduce BST as a way of representing a map.

**Diff btw MAP / LIST**

* Memphis is a pretty big hub for shipping and freight because of our presence on the Mississippi river and all the freight trains that come through town.
* Each car in a train can contain a variety of things, like petroleum, or chemicals, or grain, or coal, or something in bulk like that. Every morning I need to send various goods to various places.
* Suppose the thing I care about most is the order of the cars on the train. In other words I need to know that my first car carries coal, my second car carries petroleum, and so on.
* What is the best data structure to represent the collection of cars that make up a train? (list). List<String>
* Now suppose I say all the cars on a train must carry the same kind of cargo. So there's now one train for coal, one train for petroleum, etc. I give every train a unique three digit ID number.
* I want to be able to type in a train ID number and quickly find out what kind of cargo the train is carrying. What data structure should I use? Map<int, string>
* Change it up: Suppose I want to type in a type or cargo and get back the Train ID number that is carrying that cargo (assume there's only one train that is carrying every kind of commodity). Ans: Map<string, int>
* What if there are multiple trains carrying each kind of cargo. I.e., trains 100 and 400 carry grain, trains 200, 900, 600 carry petroleum, and train 500 carries coal.
  + Map<string list<int>>
* Now suppose I want to make a up a schedule for the order in which the trains leave Memphis in the morning. So assume all we care about is the 3-dig ID number for each train, and the order in which they leave in the morning. List<Int>
* Now suppose I care about the actual times they leave.
  + different ideas: maybe Map<Time, Integer> (lets us quickly lookup train id given departure time)
  + maybe Map<Integer, Time> (lets us quickly lookup departure time given train id)
  + maybe List<Train> where the Train object stores the Id number and departure time.

**Linked Node Struct for BT:**

Class Node{

int/string/object data;

Node left;

Node right;

}

show example with "expression tree" – this is what programming langs use under the hood to represent math in a compiler or interpreter.

Show example with (3 \* 2) + (15 / 3)

* First show 3 \* 2 as BT with explicit nodes.
* Then 15/3 as BT with explicit nodes.
* Then join together with a root node.
* Then draw as a simpler tree with just circles.
* Note the hierarchy: in this kind of tree, the LEAVES are the operands (numbers), and the INTERNAL NODES are the operators. The higher you go in the tree, the later the operations are done.

RECURSION

* Most algorithms for binary trees are recursive.
* Reason: because we have to process both a left and a right child, and each of those children has two children, there's no simple linear way to walk through a tree anymore if you want to visit every node.
* So to practice with recursion, I want to show some examples with linked lists.
* Write psuedocode:
* LLnode
* void func(Node n){
* if n==null: return  
   else   
   print(n.data)  
   func(n)  
    
  Reverse order of print/func.
* Sum up all elts in a list.

**Traversals:**

* + Preorder(node):
  + IF NOT NULL:  
     Visit(node.data)
    - Preorder(node.left)
    - Preorder(node.right)
  + Do simp example with small tree

**BST Introduction:**

* So we've talked about general "trees" (any number of children per node permitted), "binary trees" (0, 1, or 2 children permitted). Now we're going to introduce a further restriction of a tree, which is called a "binary search tree." To be clear, we haven't made any sort of connection between the MAP ADT we saw the other day and any of these tree concepts. So if you're not seeing the connection, you're not supposed to.
* However, now we're going to introduce a specific kind of tree called a BST that we will use to implement a map.
* BST definition:
  + A BST is a kind of binary tree (so 0, 1, or 2 children) where the specific items stored in each node restrict the possible values of all the descendent nodes.
  + Definition:
  + A BT is a BST if, for every node N in the tree,
    - the left child of N and all further descendent nodes have values that are less than N's value.
    - the right child of N and all further descendent nodes have values that are all greater than N's value.
* Examples:
  + 50  
     25 75
  + 22 47 70 88  
     44 49 99->100
* Things to point out.
  + Circle entire left subtree---everything less than 50.
  + Circle entire right subtree ---everything > 50.
  + But now look at 25.
    - left branch – everything less than 25.
    - right brand --- everything > than 25.
      * But also < 50.
  + This is a recursive structure.
    - The whole structure is a BST.
    - But also each subtree is also a BST.
* So why are BST's so cool?
  + They are faster than lists (arrays/LL) for adding items, removing items, and searching for items.
* Searching:
  + Suppose I want to know whether 44 is in this tree.
  + Suppose all I am given is a pointer to the ROOT of this tree (the 50).
  + What would this search algorithm look like?
  + Well if all I have is the root, I can compare 44 against the root. Now 44 is smaller than the root. So WHAT DOES THIS TELL ME? 44 if it is in the tree, must be in the left subtree.
* pseudocode:
  + Boolean search(node, lookFor) {
  + if lookfor == node.data return true  
     else if lookfor < node.data return search(node.left)  
     else return search(node.right)
* Introduce this as a way of representing a map, with bank account numbers and balances.
  + Show example, say we have 3 bank accounts, with numbers 123 ($50), 456 ($25), and 789 ($30).
  + Show a BST with 456 as the root.
* INSERT
* handwavy version
* Same as search algorithm, but whenever we get to null, make a new node at that location.
* Implementing this is slightly tricky, because similarly to inserting at the tail of a linked list, we actually need to hold onto the pointer BEFORE it becomes null.
* Cover insert code.
* REMOVE
* Cover handwavy version.
* Three cases:
* No children.
* One child.
* Two children.
* Alg: Replace value to be deleted with its INORDER SUCCESSOR,
* then delete the successor.
* **DAY 4**
* Go through remove.
* Do worksheet.
* Talk about bad insertion orders—all sorted forwards or backwards. Best is a “random” order.
* Talk about height of a tree. Height is logarithmic in # of nodes in “perfect” case, linear in worst case.