CS 10, Winter 2017
Problem Solving via Object Oriented Programming
Hierarchies

We have dealt with a couple of organizations of data so far — grid-based (images) and linear (lists). Today we'll deal with data that's hierarchical in nature. Hierarchically organized things that come to my mind: the tree of life, many tournaments, XML and HTML documents (take a look at document structure in browser), folders nested inside folders nested inside ... on computers.

We can represent hierarchical relationships using a data structure called a *tree*. A tree is built up from *nodes*. Tree terminology is taken from family trees. The top node is called the *root*. Each node has zero or more *children* (unlike a biological tree, by convention in computer science, the tree grows *downward* from the root). An *edge* connects a node and a child. Nodes having no children are called *external nodes* (or *leaves*) and nodes with children are called *internal nodes*. A child has exactly one *parent* (except for the root, which has no parent). Nodes with the same parent are *siblings*. *Ancestors* and *descendents* are what you would expect from family trees. A *subtree* of a node consists of all descendents of that node (including the node itself).

We'll build up some tree representations and related algorithms for dealing with some different types of hierarchical data. The textbook provides additional reading, but note that they use a somewhat different (more powerful, but more complicated) representation of trees. So read it to gain additional description and examples, but don't be thrown by the code itself.

- General-purpose binary trees
- Expression trees
- Tree traversal
- Java notes

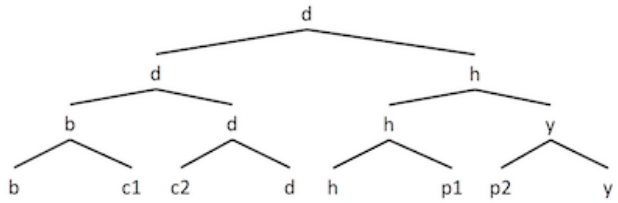
All the code files for today: Addition.java; BinaryTree.java; Expression.java; ExpressionTest.java; Multiplication.java; Number.java

Slides from class

Binary trees

There is a special, and very common, kind of tree called a *binary* tree, where each node has 0, 1, or 2 children. The tree of life we're looking at has that structure. (And so does an ancestor tree, where a node's "children" in the tree structure are its parents in real life. There are always two, except in the case of cloning, etc.)

One example of a binary tree is a tournament, e.g.:



So the "final" is between "d" and "h", and "d" won. The semifinals are between "b" and "d" ("d" won) and between "h" and "y" ("h" won). Etc. Can you guess what the strings in this tree stand for? Hint: I could have a corresponding tree represented by colors instead of strings, and the color corresponding to the string "d" would be "green". In this binary tree, every parent has two children. But say if "p2" dropped out before the tournament started, then "y" would get a bye and advance to the next round — the "y" parent there would have only the one "y" child rather than both "p2" and "y".

Some binary tree code: BinaryTree.java. Let's first look at the data structure itself. A tree has a "left" and a "right", which could be children or could be null. It also has "data" stored at the node (e.g., the name of the organism in the tree of life). We don't really care what type the data is, so we make it a generic parameter: "BinaryTree<E>". Simple accessors determine whether the tree has leaves, and whether it is an inner node or a leaf. This version doesn't include a parent, but that's an easy extension, and would allow one to go back up a tree from a node. In general, though, tree code will start at the top (the root) and just work its way down the tree, much like linked list code starts at the beginning and works forward.

The obvious difference between trees and linked lists is key: a node in a linked list has at most one "next" (maybe 0), while a node in a binary tree could have two children (maybe 0 or 1). So we can't just proceed "forward" in a loop. Consequently, most tree code is recursive, so this is a chance to polish up those skills.

size (# nodes in a tree)

The number of nodes a tree is just the number in the left subtree plus the number in the right subtree plus 1 (the node itself). So we ask the left child for its size, the right child for its size, and add those plus 1. You can think about the recursion if you want to (the left child will then ask its left and right children, ...). Or (my preference) you can just abstract that away — you call a method that gives the size of the left subtree and one that gives the size of the right subtree. They happen to be recursive calls to the same method, but what really matters is that they do what we need them to — we just need the sizes of the subtrees, and that's what the method is advertised to do!

If you think back to the kind of recursion you've probably seen before, you might be wondering about the "base case". (If not, no worries, I think this kind of recursion is actually easier to see once you're used to it.) Here, that's where we hit a leaf, and thus no more recursive calls are made. But it's kind of implicit, right? We could make it explicit by saying `if (isLeaf()) return 1;`, but the same thing gets computed anyway. More generally, this type of "data structure recursion" proceeds through a data structure as long as there is further to go. You can do exactly the same thing with a linked list.

height (longest path to a leaf)

The longest path from a node is the longer of the two paths down the left side vs. the right side, plus 1. Here I put the explicit base case in, because the height of a leaf really is considered to be 0, which is not what would be computed.

equals (same structure and data)

Two trees are "equal" to each other if they have the same data, and their left children are equal and their right children are equal. (Note that this doesn't allow for "mirror image" testing, with left==right and right==left — could you code that version?) **Important**: to check if they have the same data, we use the "equals" method. The "==" operation only tests if two things are actually the same thing — 3 == 3, 2+2 == 4, etc. In general, there's no way for Java to know if two different objects "mean" the same thing, so "==" will return false unless they are actually the same object / piece of memory (via "="). But there's a method of Object called "equals" that lets us define what it means for two different objects to be basically the same. So here we invoke that method to test if the data of this tree "equals" that of t2. `String` does the right thing for us in the example. (And for our own classes, we'd have to override it appropriately; e.g., if we wanted to allow two different blobs to be considered equal, we'd have to specify when that's true.)

fringe (list of all leaf data)

This one has a somewhat different pattern. Intuitively, to get the fringe of a tree, get the fringe of its left child and that of its right child, and "add" them together. This would actually be inefficient in Java (w/o your efficient singly-linked list append method from the short assignment). Appending two ArrayLists actually ends up copying one of them, and we'd end up with a quadratic time algorithm with stuff getting copied again & again on its way up the tree. So instead, we pre-create a list and have each node adds its fringe to the end of that list. No copying, just adding new things onto a single growing list, and each thing gets added once (when it's the leaf). This list is called an "accumulator", as it accumulates the fringe going forward.

In order to use the accumulator pattern, we split our recursive call out into a separate helper method. The "main" `fringe` method just creates the accumulator, passes it to the helper, and returns it once the helper is done. The "helper" then makes the recursive calls to add leaf data to the accumulator. Note that we could actually use the same name for the helper method, since the parameter list is different, but here a different name sounded better. And I made it private since nobody else should be calling it directly.

stringification

There are many ways to print a tree; here, we use an indented representation, where at each level we indent two more spaces. Think of it like a table of contents hierarchical structure — sections / chapters / sections / subsections.... So to convert a tree, we concatenate its data (indented) and the conversion of its children (indented a bit more). Note again the use of a helper method to carry forward some information through the recursion, here how much to indent.

Building a tree from a string representation is much harder. There are many ways to represent trees as strings (with one example just above); one reasonable format is Newick. The interactive tree of life provided a Newick-format dump of the default tree: itol.txt. (Save it in folder "inputs".) Newick trees are parenthesized, with labels for the nodes and commas separating the children; the whole tree ends in a semicolon. For example, the tournament above is "(((b,c1)b,(c2,d)d)d,((h,p1)h,(p2,y)y)h)d;".

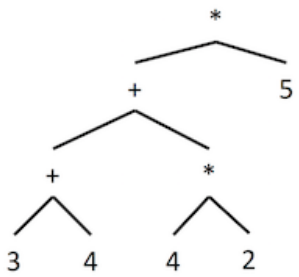
I provided a simple, completely non-robust Newick parsing method just to support testing; you can ignore the code if you like and just use it (as in the `main`method in `BinaryTree`), but if you're interested, read on. My parser uses the (now deprecated) tokenizer in Java to split the string up at parentheses and commas. When it sees an open paren, it recursively parses the left subtree, then gets the comma, then recursively parses the right subtree, then gets the close paren, then the node label. (Newick trees also allow distances from parent to child; I just throw those away.)

Expression trees

Programs are often represented as trees inside an IDE like Eclipse, as well as inside a compiler or interpreter that is generating lower-level code or running bits of code. You can see the nested structure in the indentation and other punctuation (matching pairs of braces,

parentheses, etc.) — the class has a child that's a method, the method has a child that's an "if" statement, the "if" statement has a child that's the test and another child that's the code to execute when the test passes, ...

Without going that deep into it, let's just consider simple arithmetic expressions. Each node represents an operation to be applied to its children. For example, here's ((3+4) + (4*2)) * 5



So to evaluate the tree, the root would ask its children to evaluate themselves. One child would (recursively) return 15, while the other would return 5, so then the root would multiply those to get 75. If we were to look at what went on recursively on the left, we'd see that the left child there would return 7 and the right child 8, and thus the sum would be 15.

To emphasize that trees need not use uniform classes, I've provided a simple version of an expression evaluator for just addition and multiplication. The Expression.java interface simply guarantees that everything can return a value; the Addition.java and Multiplication.java classes implement inner nodes; the class implements leaf nodes. You can do your own math with ExpressionTest.java. The implementations do exactly the recursion described in the preceding paragraph. Note also another way to `toString` a tree, here parenthesizing each part of an expression.

Tree traversal

The basic pattern of computing something in a tree is that inner nodes recurse left and right and combine results from their children, while leaf nodes provide the base cases. But there are different ways of ordering these steps. For example, we could recurse with the left, then with the right, and then do something with those results at the parent (this is called "postorder"). Or the parent could do something first, and then the two children ("preorder"). Or the left subtree, then the parent, then the right subtree ("inorder"). Or the parent, then the left, then the parent again, then the right, then the parent ("Eulerian walk"). Etc. The book goes through these in detail.

```
preorder()
  do something
  left.preorder()
  right.preorder()

postorder()
  left.postorder()
  right.postorder()
  do something

inorder()
  left.inorder()
  do something
  right.inorder()

euler()
  do something "pre"
  left.euler()
  do something "in"
  right.euler()
  do something "post"
```

Examples that we've seen of each?

Java notes

equality testing

== is really only appropriate for numbers and characters (primitive types), to see if something is "null", or to see if two object variables

are referring to exactly the same instance (not instances that have the same data). An equals() method is the correct way to see if two objects (including Strings) have the same data.