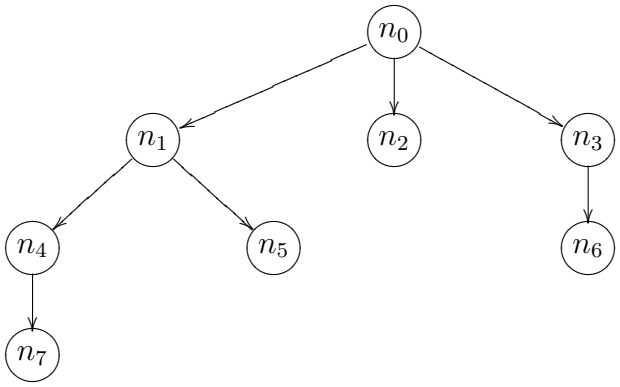
CS1Ah Lecture Note 24

Trees

This lecture examines the *tree* data structure. Trees are widely used in computer science and we have seen several examples already, including the Unix filesystem and the class hierarchies of object-oriented programming. Here we will go into more detail, considering the most common special case of binary trees, and their implementation in Java.

24.1 Tree terminology

A *tree* consists of a collection of *nodes* connected by directed *edges*; at most one edge joins any two nodes. Tree terminology is a mix of the genealogical and the arboreal. An edge connects a *parent* to its *child*. A tree has a distinguished node called the *root* node. Contrary to nature's trees but like family trees, Computer Science trees grow downwards, and are illustrated with the root at the top:



Here the nodes are shown as circles, and given names $n_0 \ldots n_7$. The root is node n_0. The edges are drawn as arrows. When we draw trees like this, the arrow heads on the edges aren't really necessary because parents are always drawn above their children.

A *path* in a tree is a sequence of connected nodes: the *length* of the path is the number of edges which appear in it. The final defining characteristic of a tree is that there is exactly one path from the root to any other node. This forbids structures with circular paths, structures where a child has more than one parent node, and structures which are not fully connected. (The more general structures are called *graphs*.)

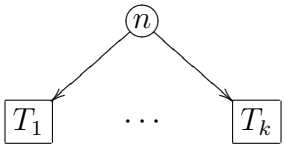
Nodes without children ($n_7, n_5, n_2.n_6$) are called *leaf* nodes. The *height* of a tree is the length of the longest path from the root to a leaf. The tree shown above has height 3.

24.2 Trees and recursion

The definition of tree given above is a classical one, which specifies trees by placing restrictions on connected sets of nodes and edges: *cutting down* the set of graphs to get the set of trees. But the set of trees has a much more natural *inductive* definition:

- An unconnected node n is a tree, with root n and leaf n:

- If $T_1, \ldots, T_k$ are trees, for $k \geq 1$, then we can form a tree by connecting a new node n above them:



  The root of the new tree is n, and the leaves are the leaves of $T_1, \ldots, T_k$.

An inductive definition specifies how we *build up* some collection. The two rules above are the only way we are allowed to construct trees. The definition has a base case (the unconnected node) and an inductive step (connecting several trees with a new node). The inductive step specifies a new larger tree by putting together smaller ones.

The inductive definition also shows the *recursive* nature of trees: the second rule specifies a tree in terms of some *direct subtrees*. Since we know that every tree is either unconnected or has subtrees, algorithms for processing trees can often be described recursively. Here is a recursive way of defining the height function on a tree T:

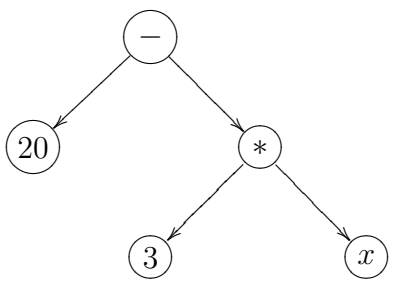
- If T is an unconnected node, $height(T) = 0$
- If T has subtrees $T_1, \ldots, T_k$, then $height(T) = 1 + max\{height(T_1), \ldots, height(T_k)\}$

Since this recursive definition follows the inductive definition of trees, it is guaranteed to be well-defined. You should convince yourself that this definition is equivalent to the one given before: the length of the longest path from the node to a leaf.

24.3 Storing data in a tree

Trees are usually used to store data. This is done by attaching data values to nodes and edges in different ways. The data value attached to a node or edge is sometimes called its *label*.

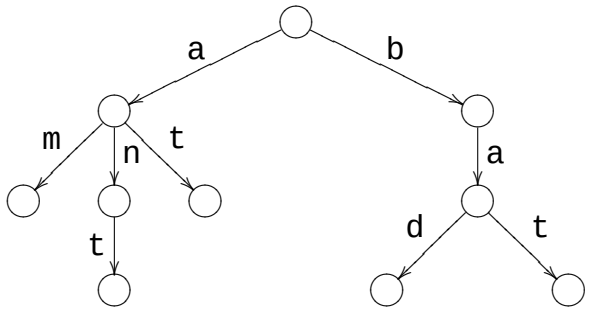
In an *expression tree*, internal (non-leaf) nodes of the tree are labelled with operators, while the leaves are labelled with variables or constants. Here is an expression tree which stores the expression $20 - 3 * x$:



Notice that we are showing the node labels instead of node names n shown before. Names of nodes are used to identify them uniquely, whereas the same label may occur many times on different nodes.

The structure of an expression tree shows how the expression has been interpreted according to precedence rules. Similar trees are used to represent not just expressions but entire programs, or in natural language processing, sentences and paragraphs in English. These are called *parse trees* (or *syntax trees*).

In a *trie*, the edges are labelled with letters from an alphabet, so a path to a leaf encodes a word. This can be an efficient way of storing a dictionary. Here is a trie which stores the words "am", "ant", "at", "bad" and "bat":



To help achieve an efficient representation, we should not use duplicate labels for edges from the same node. So to handle prefixes (e.g. storing "an" in the trie above) we would need to add a special stop letter. Given these hints, can you think of an efficient way to represent this structure?

24.4 Ordered trees and binary trees

Often we work with *ordered trees* where there is an ordering amongst the children. Pictures like those above immediately define such an ordering according to the horizontal placement of nodes: each internal node has a *leftmost* child and a *rightmost* child. A kind of ordered tree where each node has at most two children is called a *binary* tree; additionally, in a binary tree a subtree is always in a *left* or *right* position. The expression tree shown on page 2 is a binary tree.

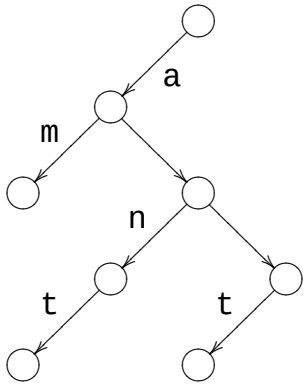
A convenient way of defining a binary trees inductively is to use the idea of an empty tree in the base case:

- The empty tree is a binary tree;
- If T_L and T_R are trees, n is a fresh node name, then the tree with n as root and T_L and T_R as subtrees is a binary tree.

The empty tree is useful here because it means all nodes in a binary tree can be treated as having *exactly* two subtrees. A node which has two empty trees as subtrees is really a leaf. This simplification makes binary trees rather easier and more efficient to represent on the computer than general trees.

Binary trees are one of the most important basic data structures. In fact, a binary tree can be used to represent any other ordered tree. This can be done by placing the

leftmost child in the left subtree, and the remaining siblings along a right-hand path. For example, the left-hand branch of the trie above becomes:



In general, we need more nodes in the binary tree representation to reflect the structure of the original tree. The edges connecting these extra nodes are not labelled.

24.5 Traversing a tree

Many tree algorithms first build up a tree in memory, and then process it somehow. One way of processing a whole tree is to *traverse* it, visiting all the nodes and edges.

There are three basic modes of traversing a binary tree, which directly give rise to recursive algorithms, beginning from the root of the tree:

pre-order We visit the current node, then the left subtree, then the right subtree. For the expression tree shown on page 2, a pre-order traversal printing out the node labels would result in the prefix notation string: $- \quad 20 \quad * \quad 3 \quad x$.

in-order We visit the left tree, then the current node, then the right tree. Printing the expression tree nodes, we get: $20 \; - \; 3 \; * \; x$.

post-order We visit the left then right subtrees first, then the current node. Printing the expression tree nodes, we get the postfix notation: $20 \quad 3 \quad x \quad * \quad -$.

The postfix order should be familiar as the behaviour of the Java compiler and virtual machine: expressions are compiled into instructions which evaluate them using a stack, pushing the operands and popping them off when we meet an operator. Interestingly, with either the prefix or postfix traversal of an expression tree, the resulting output is unambiguous (no precedence conventions are required). For the in-order traversal, we generally need to insert parentheses to print an unambiguous expression.

The three basic modes of traversal can be combined into one more powerful method:

Euler tour We first compute some starting value; then visit the left subtree; then compute an intermediate value; then visit the right subtree; finally compute a resulting value.

The Euler tour can be visualised as a "walk" around the outside perimeter of the tree, so that each node is actually visited (passed in the walk) three times: on the left, underneath, then on the right.

24.6 Binary trees in Java

Up to now, we have only described abstract notions of trees. To actually use trees on the computer we must consider how to represent them. A simple way of representing binary trees in Java is to define a `BinTree` class whose instances are non-empty trees, with `left` and `right` subtrees, and use the `null` reference to stand for an empty tree. The code below follows this idea, and implements an pre-order traversal method.

```
 1  class BinTree {
 2      private BinTree left;         // left subtree
 3      private Object  data;         // label for node
 4      private BinTree right;        // right subtree
 5
 6      // Construct a leaf given a label.
 7      BinTree(Object data) {
 8          this.data = data;
 9          // left and right default to null (the empty tree)
10      }
11
12      // Construct a tree given a label and two subtrees.
13      BinTree(BinTree left, Object data, BinTree right) {
14          this.data = data;
15          this.left = left;
16          this.right = right;
17      }
18
19      public void printNodesPreorder() {
20          printNodesPreorder(this);
21      }
22      private static void printNodesPreorder(BinTree tree) {
23          if (tree != null) {
24              System.out.print(tree.data);
25              System.out.print(" ");
26              printNodesPreorder(tree.left);
27              printNodesPreorder(tree.right);
28          }
29      }
30  }
```

Lines 2–4 define the three fields of the `BinTree` class. An object in this class represents a node in the tree; each node is labelled with a `data` item that can be any object. In lines 6–17 we define two constructors for `BinTree`.

The public instance method `printNodesPreorder` defined on line 19 invokes the private static method `printNodesPreorder` on line 22, on the tree object itself. (Remember that in any instance method, **`this`** stands for the object invoking the method). The static private method `printNodesPreorder` traverses the given `tree` argument recursively following the pre-order method.

An alternative recursion pattern

An alternative way to implement `printNodesPreorder` is to use the instance method itself to do the recursion, recursing via the node objects of the tree:

```
public void printNodesPreorder() {
    System.out.print(data);
    System.out.print(" ");
    if (left != null)
        left.printNodesPreorder();
    if (right != null)
        right.printNodesPreorder();
}
```

We know that the object invoking `printNodesPreorder` is non-null (a priori), but we must check the subtrees before attempting to invoke their `printNodesPreorder` methods. You might like to ponder other differences compared to the static method.

Here is a class `BinTreeTest` which contains a `main` method to test the `BinTree` class. The `main` method builds the expression tree shown on page 2.

```
1  class BinTreeTest {
2      public static void main(String[] args) {
3          BinTree exprtree =
4              new BinTree(new BinTree(new Integer(20)),
5                          "-",
6                          new BinTree(new BinTree(new Integer(3)),
7                                      "*",
8                                      new BinTree("x")));
9          exprtree.printNodesPreorder();
10         System.out.println();
11     }
12 }
```

Here's the result of running `BinTreeTest`:

```
[fraenk]da: java BinTreeTest
- 20 * 3 x
```

As an exercise, you should try to implement the other traversal methods.

References Chapter 19 of *Java How to Program, 4th Edition* has some basic material on trees (section 19.7, page 1116). A book which covers the topic well, including an implementation of the Euler tour, is *Classic Data Structures in Java* by Timothy Budd, published by Addison Wesley.

David Aspinall, 2002/11/29 12:42:07.