# Trees

See Chapter 18 of Weiss

By the way, I am

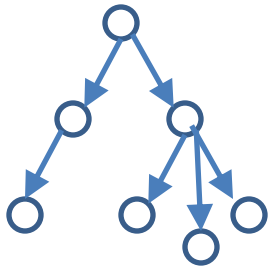
Bob Geitz
www.cs.oberlin.edu/~bob/cs151

Ben is at a conference in Kansas City (where, according to Rogers and Hammerstein, everything is up to date and they've gone about as far as they can go....). He'll be back on Monday.

Trees are the second-most important data structure in programming, following only lists.

We will think about trees in two different ways.

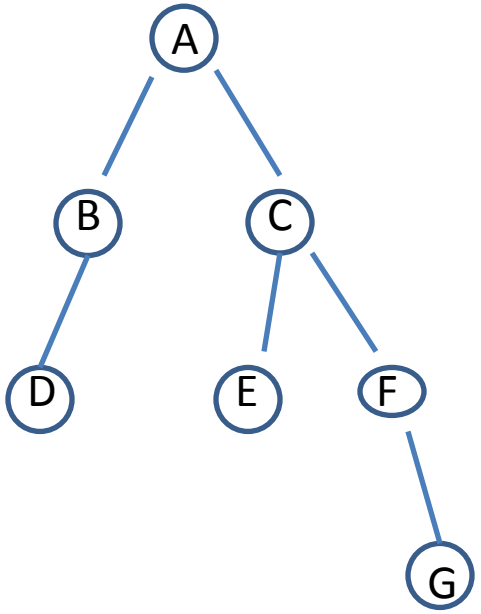
**Graph definition**: A tree is a kind of directed graph, so it has a set of nodes and a set of edges connecting the nodes. There is one special node called the root. Each node except for the root in a tree has an incoming edge from one other node. The root has no incoming edges. There is a path of edges from the root to every other node. Nodes that have no outgoing edges are called leaves.



We call the node at the start of an edge a parent node. The node at the end of this edge is the parent's child.

In this terminology a node can have multiple children, and every node but the root has exactly one parent.

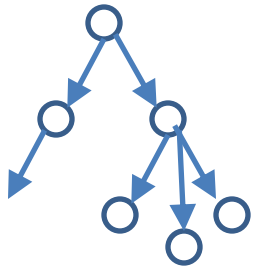
We measure the length of a path in the tree by the number of edges it contains (not the number of nodes). The height of a node is the longest path from it to a leaf. The height of the overall tree is the height of its root. The depth of a node is the length of the path from it to the root. The depth of the root itself is 0.




| Node | Height | Depth |
|---|---|---|
| A | 3 | 0 |
| B | 1 | 1 |
| C | 2 | 1 |
| D | 0 | 2 |
| E | 0 | 2 |
| F | 1 | 2 |
| G | 0 | 3 |

Here is **a recursive definition** of a tree:

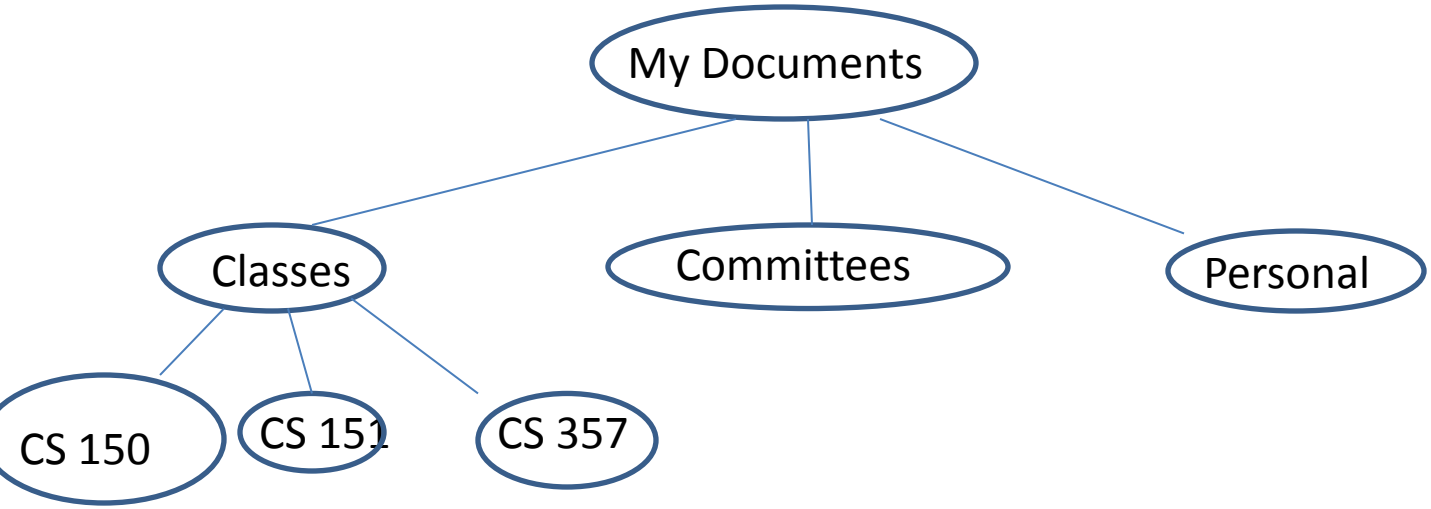
A tree is either empty or it is a root r and 0 or more non-empty subtrees connected to r by an edge.



A binary tree is one in which each node can have at most two children. The children are often referred to as the *left* and *right* children.

Trees are used in many situations:

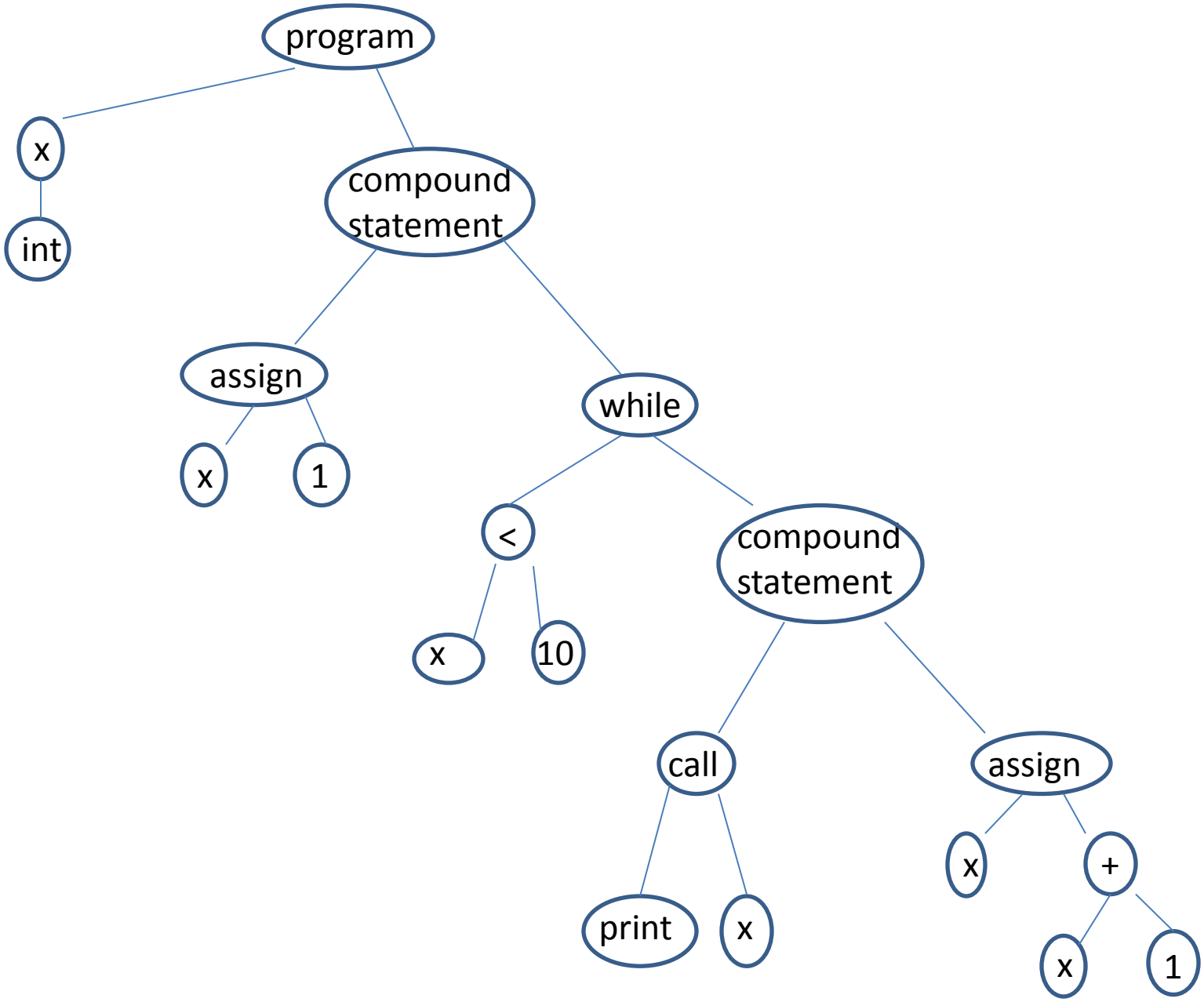
- Anything hierarchical, like file systems or administrative structures or Java class structures can be represented by trees.



- Many compilers build a tree representation of the program they are compiling, guided by a grammar for the programming language.

The next slide shows a tree representing

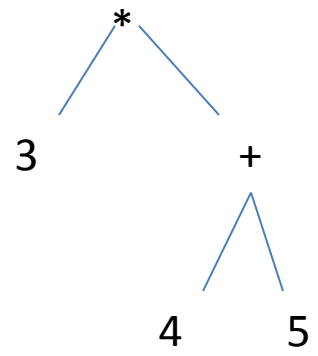
```
int x;
x = 1;
while (x < 10) {
        print(x);
        x = x + 1;
}
```


program
x
int
compound statement
assign
x
1
while
<
x
10
compound statement
call
print
x
assign
x
+
x
1

- Games are often represented by trees, where the root represents the current state of the game and children represent possible moves.

- Indexes in a database are built on tree structures.

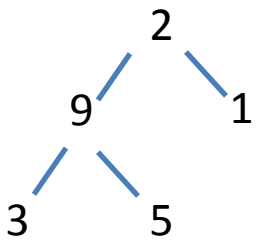
- Since arithmetic operators take two arguments, one use of binary trees is in representing expressions. You might represent 3*(4+5) as




Such a tree has operators in the interior nodes and numbers in the leaves. There is an easy recursive algorithm to compute its value -- evaluate the left child, evaluate the right child and apply the operator to those values.

So how do we represent trees?

You can use arrays. If there are at most 2 children of each node, you can put the root at index 0, its children at index 1 and 2, the children of the node at index 1 could be at indices 3 and 4, and so forth. The children of the node at index n are at index 2n+1 and 2n+2. The parent of node at index k is at index (k-1)/2. So the array
[2 9 1 3 5] represents the tree




If a node could have 3 children the kids of node [n] would be at [3n+1] [3n+2] and [3n+3]

A more flexible scheme is to use a linked structure. In Lab 5 we will use the following 3 classes:

```
abstract class BinaryTree<T> {
        //  methods we want the tree classes to have
}

class EmptyTree<T> extends BinaryTree<T> {
        // no data and just trivial methods
}

class ConsTree<T> textends BinaryTree<T> {
        T data;
        BinaryTree<T> left;
        BinaryTree<T> right;
        // non-trivial methods
}
```

For example, suppose you want to add a height( ) method to the BinaryTree class. You do this in three steps:

First, you need to add an abstract height( ) method to the abstract BinaryTree<T>  class:

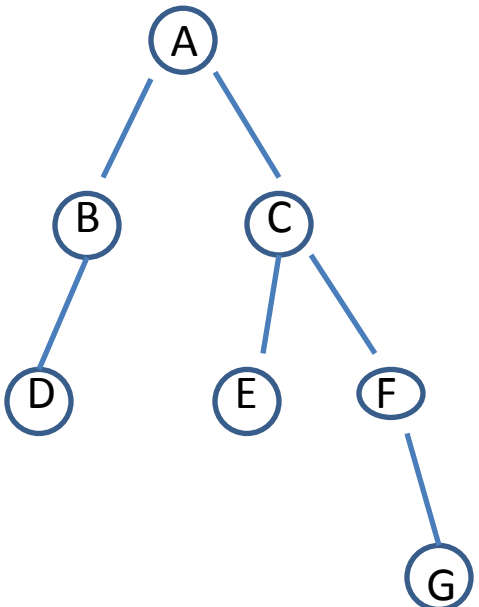
```
public abstract int height( );
```

Next, add a height( ) method to the EmptyTree<T> class. The EmptyTree methods are almost always trivial. In this case

```
public int height( ) {
        return -1;
}
```

Finally, add a height method to the ConsTree<T> class. This is usually the only step that has any substance. Because the BinaryTree structure is recursive, the ConsTree methods are usually recursive.

In this case we want to find the height of a tree in terms of the heights of its children.



The method for the ConsTree<T> class is

```
int height( ) {
    return 1+ Math.max(left.height(), right.height() );
}
```

Note that we defined the height of an empty tree as -1 to make this work. A leaf node has height 0 and 2 empty trees as children, so the empty trees need to have height -1.

Note also that our recursive tree structure has empty trees as the base cases at the bottom of our trees. ConsTrees always have children, so every downward path in the tree is terminated by an EmptyTree. This means that the base cases for our recursions are the methods in the EmptyTree class.