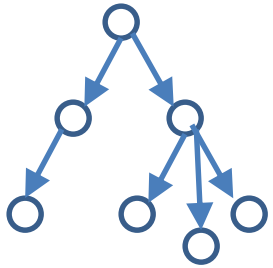
Trees

See Chapter 8 of the text

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

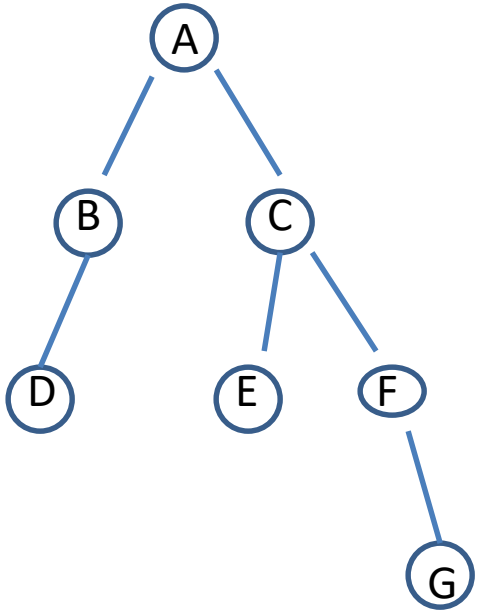
We will think about trees in two different ways.

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 parents can have multiple children, but children have exactly one parent.

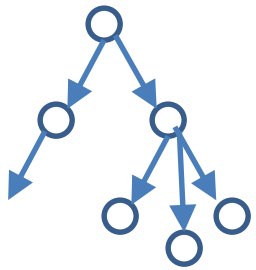
We measure the length of a path in the tree by the number of edges it contains (not the number of nodes). The height of a node is the longest path from it to a leaf. The height of the overall tree is the height of its root. The depth of a node is the length of the path from it to the root. The depth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

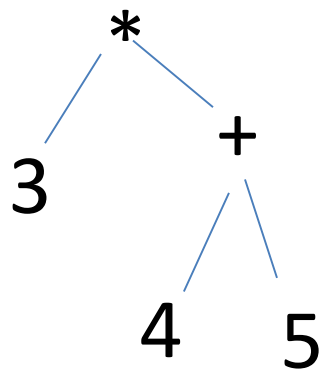


A binary tree is one in which each node can have at most two children. The children are often referred to as the *left* and *right* children.

Trees are used in many situations:

- Anything hierarchical, like file systems or administrative structures or Java class structures can be represented by trees.
- Whenever a program is compiled the compiler builds a tree representation of the program, guided by a grammar for the programming language.
- 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.

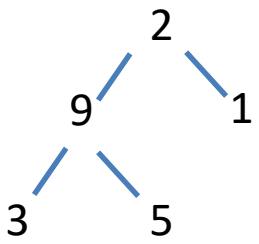
Since arithmetic operators take two arguments, one use of binary trees is in representing expressions. You might represent 3*(4+5) as



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]

Clicker Question: The children of the node at index n are at indices 2n+1 and 2n+2. So in the following array representation of a tree what are the children of the node whose *value* is 7?

0	1	2	3	4	5	6	7	8	9	10	11
3	4	9	2	1	7	15	11	5	12	8	17

A. 15 and 16
B. There are none because 15 and 16 are off the array
C. There is only a left child and its value is 17
D. There is only a right child and its value is 17

Another clicker question: What is the problem of storing a tree in an array?

A. Arrays take up too much space
B. The tree has to be *complete*: every node except for the bottom row has to have 2 children
C. It is too hard to compute where the parent of a node is stored.
D. This doesn't allow a node to have 2 parents.

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
}
```