TREES

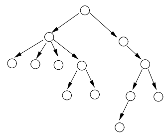
Trees

A tree is a restricted form of graph.

It has a finite set of nodes and edges. But the edges have a direction (from "parent" node to "child" node), and a non-empty tree must satisfy the following rules:

1. Each node has exactly one parent, except the root, which has none.
2. Each node is connected to the root by a path from the root to that node.
3. There are no "cycles", that is no paths that form loops through two or more nodes.

Also, the children in our trees are ordered; some people call such trees "ordered trees".



About the diagram:

- One draws trees upside-down.
- The arrows indicate the parent-child relationship: they aren't necessarily object references.

Crunchy nugget: The first two properties are sufficient to define a tree, since together they imply the third.

Questions:

- Can you prove this?
- Are any other subsets of these properties sufficient?

Definitions

- A node with no children is called a **leaf**.
- A node that is neither a leaf nor the root is an **internal** node. (alternatively: a node that has both a parent and children).
- Nodes with the same parent are **siblings**.
- The tree formed from a node together with all its "descendants" and the edges among them is a **subtree**.
- If m_1 is the parent of m_2, m_2 is the parent of m_3, ..., and m_{k-1} is the parent of m_k, then the sequence of nodes $m_1, m_2, \ldots, m_k$ is a **path** in the tree.
- The **length** of such a path is k, i.e., the number of nodes on the path.
- The **height of a tree** is the length of its longest path from the root to a leaf.
- The **depth of a node** is its distance from the root: the number of *edges* on the path from the root to that node. The depth of the root is 0.

Special Kinds of Trees

A tree may have a fixed **branching factor**: a maximum number of children for any node.

A **binary tree** has branching factor 2.

Operations on trees include

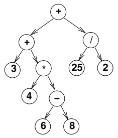
- inserting a new node
- removing a node
- "traversing" a tree: visiting the nodes in some order and doing something at each, e.g., printing its label
- attaching a new subtree at a given node
- removing a subtree

A tree can be implemented many ways, not necessarily with node objects and references. For now, think of it abstractly.

Uses of Trees

Trees can be used to represent things that are in some hierarchical relationship:

- some family relationships, e.g. father-of. **Question:** what about parent-of?
- The class inheritance hierarchy in Java.
- Which directories occur inside which others in a computer file system.
- The relationship between method calls and their arguments. Example: `a( b(c()), d(), g(h(), i()) )`
- The relationship between operations and operands in an arithmetic expression. Example: $3 + (4 * (6 - 8)) + (25/2)$.



Binary Search Trees

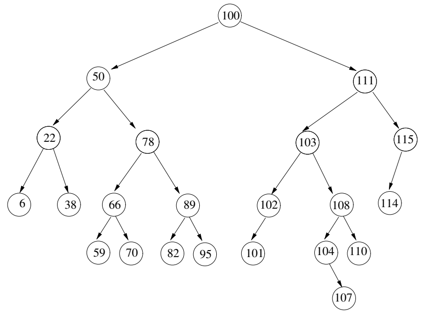
A **binary search tree** (or BST) is a binary tree where the nodes are labelled, and each node's label is:

- greater than the labels of all the nodes in its left subtree, and
- less than the labels of all the nodes in its right subtree.

Any subtree of a binary search tree is a binary search tree.

Question: Is it sufficent to only require that a node's label is greater than the left *child*'s and less than the right *child*'s?

An Example of a BST



Question: Is the above BST the only way to store those elements in a BST?

Question: How would you search in a BST?

A BST combines the best features of an array and a linked list:

- Because it is ordered, one can use an efficient search strategy analogous to binary search in a sorted array.
- But it is dynamic like a linked list, so it can grow and shrink in size easily.

Question: Why not just use a sorted linked list?

Efficiency of BST Operations

BST search seems to be efficient because, like binary search, it allows us to disregard a lot more of the problem every time we go down a level in the tree.

But we may get to disregard a lot or a little, depending on the shape of the tree. Think of some examples.

In fact, the efficiency of the operations depends on the height of the tree.

What do we know about a tree's height? ...

Height of Binary Trees

What is the maximum height of a binary tree with n nodes?

What is the minimum?

What kind of tree has the minimum height?

Height $\Rightarrow$ maximum number of nodes

Let $M(h)$ represent the maximum number of nodes in a binary tree of height h.

Prove: For all $h \geq 0$, $M(h) = 2^h - 1$.

Base Case: Prove that $M(0) = 2^0 - 1$.
A binary tree with height 0 has no nodes, so $M(0) = 0$.
And $2^0 - 1 = 1 - 1 = 0$.

Let $k \geq 0$ be an arbitrary integer.
Induction Hypothesis: Assume that
for all $0 \leq i \leq k$, $M(i) = 2^i - 1$.

Induction Step: Prove that $M(k+1) = 2^{k+1} - 1$.
Let T be any binary tree of height $k+1$.
Let L and R be the two (possibly empty) subtrees of the root.
Clearly, the maximum number of nodes in T, namely $M(k+1)$, is just 1 (for the root) plus the maximum number of nodes in L and R.
But T has height $k+1$, so L and R have height $\leq k$.
(The proof of this is left to reader.)
So the IH applies to both L and R.
As a result: $M(k+1) = 1 + 2 \times M(k)$
$= 1 + 2 \times (2^k - 1)$, by the IH
$= 1 + 2^{k+1} - 2$
$= 2^{k+1} - 1$.

Conclusion: For all $h \geq 0$, $M(h) = 2^h - 1$.

Number of nodes $\Rightarrow$ minimum height

Let n be the number of nodes in some binary tree of height h.

We proved that the maximum number of nodes in such a tree is $2^h - 1$. So:

$$n \leq 2^h - 1$$

$$n + 1 \leq 2^h$$

$$\log_2(n+1) \leq h.$$

Efficiency of Search in a BST

The height of a BST containing n nodes varies from $\log_2(n+1)$ to n.

So the number of steps required to search a BST with n nodes varies
from $\log_2(n+1)$ — like binary search
to n — like linear search,
depending on the shape of the tree!

We can keep a BST's height on the order of $\log_2 n$ by careful rearrangement during insertion and deletion. And there are ways to do this without making insertion and deletion take much longer. However, we don't cover this in our course.

Abstraction again

We've figured out a lot about trees without ever seeing an implementation.

We've talked completely in terms of the abstract notion of a tree.

Question: Why is this a good thing?

Now on to implementations ...

Implementing Trees

Using References

Intuitive implementation: each node is an object, and each edge is a reference.

(There are still other decisions, such as how to represent the list of edges from a node to its children.)

Using Arrays

Question: How can you represent a tree using an array?

ADT or Data Structure

The tree can be viewed as either

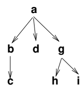
- an ADT, or
- a data structure.
  Example: A tree can be no more than an implementation of the set ADT.
  And an array, for instance, could in turn implement the tree.

Tree "Traversal"

Traversal means doing something with each node, e.g., printing it.

Standard orders in which to process a tree:

- **preorder**: For each node, process the node first, and then its subtrees, in order.
- **postorder**: For each node, process the subtrees in order first, and then the node itself.
- **inorder:** (for binary trees only) For each node, process the node inbetween processing the left and right subtrees.



preorder:
postorder:
inorder:

In each order, we weave our way through the tree in the same way ("depth-first"), passing non-leaves more than once, but differing in which pass it is when we process the node.

Question: Which traversal order represents:

- How Java evaluates method calls in an expression? The order of those calls in the source code?
- The order of values in a BST?

Implementing Tree Traversal

Notice that:

- We must remember what nodes to return to and finish later.
- In what order do we return to the previously-visited nodes?

What ADT would be useful here?

A Simpler Way

The following approach is *much* simpler and doesn't require us to manage a stack.

```
class BSTNode {
    public Comparable key;
    public BSTNode left;
    public BSTNode right;
    public BSTNode(Comparable key) {
        this.key = key;
    }
}

/** Print the tree rooted at t, in order. */
private static void inorderPrint(BSTNode t) {
    if (t != null) {
        inorderPrint(t.left);
        System.out.println(t.key);
        inorderPrint(t.right);
    }
}
```

Question: How can we make this preorder?

This method is "recursive": it calls itself.

Recursive code is often more elegant and simple than iterative code.