Application for Trees

MyComp:
  ├── Project
  │     ├── C
  │     │     ├── Docs
  │     │     │     ├── CS201
  │     │     │     │     └── HW1
  │     │     │     ├── Lab1
  │     │     │     ├── CS202
  │     │     │     └── Pic1
  │     └── D
  │         ├── Pics
  │         │     └── Pic2
  │         └── Back-Up
  └── BDay

Want to print as:

MyComp:
  ├── Project
  │     └── C:
  │         ├── Docs
  │         │     └── CS201
  │         │         └── HW1
  │         │     └── Lab1
  │         ├── CS202
  │         └── Pics
  │             ├── Baby
  │             │     └── Pic1
  │             └── Pic2
  └── BDay
  └── Backup
void printAll (int d)

print (d spaces)
print name of node at depth d
print ("\n")
if node at depth d is directory:
for each node of this directory:
printAll (d+1)

initial call: listAll (0):
listAll (1) (Project)
listAll (2) Docs
listAll (3) CS 201
listAll (4) HW1
listAll (4) Lab1

Notice: for every node, we printed (processed) the node before moving on to children.

This is just one possible way to traverse the tree.

Tree traversals: order in which nodes are processed
Pre-order traversal:

Start at root

For every node:
1. Process the node
2. Recursively process all children

(Actual recursive code after discussing implementation)

Run-time in terms of \( n = \# \) nodes? \( \Rightarrow O(n) \)
(must process every node!)

Another example:

pre-order: \( abefjçghid \)

post-order: \( ejfbghicda \)

Another kind of traversal:

Post-order: Process children before parent

Start at root

For every node:
1. Process all children
2. Process the node
When would this be useful?

Back to directory structure example: Deleting files.

In many systems, can't delete a non-empty directory. Must delete files/sub-directories first.

Post-order:

Start at root
For every node:
  Delete children
  Delete node

ex:

pre-order: 

a b d e h i f c g

post-order:

d i h c f b g c a
So far we've looked at general trees. Every node can have many children.

Now focus on specific tree structures:

**Binary Tree** — (tree in which) each node has at most 2 children

Specific type of Binary Tree:

**Binary Search Tree (BST)** — (binary tree where)

for every node \( x \):

- all elements in the left subtree of \( x \) are \(< x\)
- all elements in the right subtree of \( x \) are \(> x\)

[Diagram of valid BST]

[Diagram of invalid BST]
**BST Implementation**

"Tree is made up of Binary Nodes."

```java
class BinaryNode {
  Object element;
  BinaryNode left;
  BinaryNode right;
  <Something else we'll add later>
}
```

```
// Object I duplicates
```

```java
class BinarySearchTree {
  BinaryNode root;

  // Methods
  contains(...);
  insert(...); // (we will see what purpose)
  remove(...); // they are when we write the code)
  preorder(...);
  postorder(...);
  inorder(...);
```
With BST, we have another type of traversal:
Recurisvely process left subtree, then node, then right subtree.

In-order traversal: left, node, right.
Start at root, for each node:
1. Recurisvely process left subtree
2. Process node
3. Recurisvely process right subtree.

```
void inorder (Node n)
if (n != null)
    inorder (n.left)
    process (n)
    inorder (n.right)
```

```
(a)
  (b) (c)
  |   |
  (d) (e) (f)
  |   |
  (g) (h)
```

d be a h f c g

d be a h c f g