Application for Trees

```
MyComp
   ↓
   Remote
      ↓
      Docs
         ↓
         CS201
         HW1
         Lab1
      Pics
         ↓
         CS202
         Pic1
         Pic2
```

Want to print as:

```
MyComp:
  ↓Remote
    ↓C
       ↓Docs
          ↓CS201
          HW1
          Lab1
       ↓Pics
          ↓CS202
          Pic1
          Pic2
  ↓D
     ↓Back-up
```
void printAll (int d)

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

initial call: printAll (0):

printAll (1) (Remote) Done (has no child)

printAll (1) (c: )
printAll (2) Docs
printAll (3) CS201
printAll (4) HW1
printAll (4) Lab1
printAll (3) CS202

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

This is just one possible way to traverse the

Tree traversals - order in which nodes are processed
Pre-order traversal: Recursively process parent, then each child.

```java
void preorder(Node n)
    if n != null
        process n
        for each child of n:
            preorder(chi)
```

Run-time: In terms of \( n = \# \text{ nodes} \) ? \( \mathcal{O}(n) \)

(must process every node!)

Another example:

- pre-order: \( a \ b \ e \ f \ j \ c \ g \ h \ d \)
- post-order: \( e \ j \ f \ b \ g \ h \ i \ c \ d \ a \)

Another kind of traversal:

Post-order: Recursively process each child, then parent.

```java
void postorder(Node n)
    if n != null
        for each child of n:
            postorder(chi)
        process n
```
pre-order:  
a b d e h i f c g

post-order:  
d i h c f b g c a

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.
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 $< X$

and all elements in the right subtree of $X$ are $> X$.

![Diagram of BST and non-BST trees]

valid BST

$\text{not BST}$
"Tree is made up of Binary nodes." (like Nodes in LinkedList).

```java
class BinaryNode {
    Object element;
    BinaryNode left;
    BinaryNode right;
}
```

```
            a
            |
        b     c
        |     |
      d   e   f   g
          |    |
       v    v
```

something else we'll add later

Object I duplicate

```java
class BinarySearchTree {
    BinaryNode root;

    // traversals
    preorder();
    postorder();
    inorder(); // specific to binary trees.

    // other operations
    contains(); leaving the parameters
    insert(); empty for now
    remove(); (we will see how to set them later).
```
With binary trees, we have another type of traversal. Recursively process left subtree, then node, then right subtree.

In-order traversal: recursively process left, parent, right.

Base Case: if node is null, don't process.
otherwise process left, node, right.

void inorder(Node n)
if (n != null)
inorder(n.left)
process(n) // e.g. print, delete, etc
inorder(n.right)
Operations:
// returns true if x in tree, o/w returns false
boolean contains (object x, BinaryNode t)

Idea:
contains(3) =
  6
  2  5
  1  4
  3

Start at root:
For each node, check element:
If = x, done, return true
If > x, recurse on left child
If < x, recurse on right child

Base Case: when node is null, return false
contains (object x, BinaryNode t)?

if t == null
  return false
else if t.element == x
  return true
else if t.element > x
  return contains (x, t.left)
else // t.element < x
  return contains (x, t.right)
\( \text{contains}(4, \text{root}) = \text{contains}(4, 6) \quad 6 > 4 \Rightarrow \text{left} \)

\( \Downarrow \)

\( \text{contains}(4, 2) \quad 2 < 4 \Rightarrow \text{right} \)

\( \Downarrow \)

\( \text{contains}(4, 4) \quad 4 = 4 \checkmark \text{ return true} \)

\( \text{ex(2) contains}(5, 6) \quad 6 > 5 \Rightarrow \text{left} \)

\( \Downarrow \)

\( \text{contains}(5, 2) \quad 2 < 5 \Rightarrow \text{right} \)

\( \Downarrow \)

\( \text{contains}(5, 4) \quad 4 < 5 \Rightarrow \text{right} \)

\( \Downarrow \)

\( \text{contains}(5, \text{null}) \quad t = \text{null, return false} \)

Run Time? For a tree with \( n \) nodes

Worst-case: Start at root, keep searching until we hit a leaf (traverse the depth of tree).

Worst-case depth?

Average case is \( \log(n) \)

- Top level has 1 node (root)
- Each level has \( \approx \) twice as many nodes as previous level
- Bottom level has \( \approx \frac{n}{2} \) nodes (leaves)