Pattern: Searching for a Tree node with a certain property.

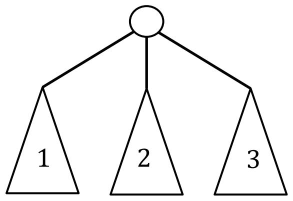
```
/** An instance is node of a tree. */
public class Node {
   public int val;  // the value in the node
   public Set<Node> children; // children
                                        // Never null
}
```

We work with class *Node* as in the previous video.

We develop a method that can serve as a model for recursive methods that look for a node with a certain property.

That property will be the value of the node, but there are other possibilities. We could search for any leaf, a leaf with a certain value, or the parent of a node with a certain value.

To the right, we show a tree with three children, numbered 1, 2, 3. There could be fewer children or more.



The specification of the method appears to the right. Using this specification and header, we write the method body.

```
/** Return a node of tree t with value v.
  * If no node contains v, return null.
  * Note: t = null denotes the empty tree. */
public static Node getNode(Node t, int v)
```

Looking at the specification, the first step is to take care of the base case that *t* could be null:

if (*t* == **null**) **return null**;

The second step: Determine whether root *t* is the desired node:

if (*t.val* == *v*) **return** *t*;

The third step: Check each child in turn to see whether the subtree with the child as root contains a node with value *v*. If so, return it. This requires a loop that, at each iteration, processes a child.

for (*Node ch*: *t.children*) { Process subtree *ch* }

How is subtree *ch* to be processed? The function specification says that if the function is called with *ch* as the first parameter, it will return the node within subtree *ch* that has value *v*, or **null** if such a node doesn't exist. Therefore, we write the body of the loop to *Process subtree ch* like this:

if (*getNode*(*ch, v*) != **null**) **return** *getNode*(*ch*, *v*);

But look! There are *two* calls *getNode*(*ch*, *v*). In the worst case, all parts of the tree will be traversed twice, and for a large tree that can be very expensive. When writing methods that may traverse large data structures, avoid duplicate work wherever possible. So we introduce a local variable and write the loop body like this:

Node n= *getNode*(*ch*, *v*);
if (*n* != *null*) **return** *res*;

Finally, if no child contains a node with value *v*, the loop terminates normally. Therefore, after the loop, **null** must be returned:

This yields the function in the textbox to the right.

```
/** Return a node of tree t with value v.
  * If no node contains v, return null.
  * Note: t = null denotes the empty tree. */
public static Node getNode(Node t, int v) {
   if (t == null) return null;
   if (t.val == v) return t;
   for (Node ch : t.children) {
       Node n= getNode(ch, v);
       if (n != null) return n;
   }
   return null;
}
```

The model, the design pattern

Use this method as a model for any function that looks for a node with a certain property. It is a design pattern consisting of four steps: (1) Base case: process an empty tree. (2) Base case: check the root, (3) Check each child in turn, returning a suitable node if the property is found, and (4) Take care of the case that no node has the property.

There will be differences, of course. For example, a precondition might be that parameter t is not null. And, testing whether a node has a certain property may be more complicated.

Discussion

We make several points about the development and the resulting method.

1. **Avoid extra unnecessary case analysis**. Some people put this statement before the foreach loop:

   if (t.children.size() == 0) **return null**;

This statement is unnecessary and makes the method *less* efficient. The foreach loop processes the case of 0 children appropriately: The loop terminates immediately, executing 0 iterations,; after that, the statement **return null**; is executed.

2. **Avoid the use a break statement to terminate the loop**. If a break statement is used in the case that a child subtree contains the desired property, the code after the loop becomes more complicated.
3. **Don't traverse the tree unnecessarily**. A function contains(t, v) is likely to be available, and the tendency is to insert the statement **if** (!contains(t, v)) **return null;** before testing the root.
4. **Avoid unnecessary structure**. We did *not* write the method body as shown to the right. Doing this introduces more complexity —an extra, unnecessary, level of nesting. Further, if the code to process a non-empty tree is long, it will become more difficult to remember why the final statement returns **null**. Handle the cases as we did, one at a time, with each case doing what is necessary.

```
if (t != null) {
  process non-empty tree
}
return null;
```