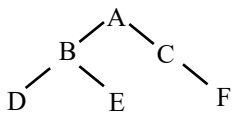
Traversing binary trees

We discuss traversing or "walking" over a binary tree, processing each node in some fashion (e.g. print its value). We use the declaration of a node of a binary tree shown to the right. In each node, field `val` contains a value of type `T`, field `left` is (a pointer to) the root of the left subtree or **null** if the left subtree is empty, and similarly for field `right`. We have made the fields public only to simplify methods we will write. An example of a binary tree appears to the right below, with the letters being the values in the nodes.

```
/** An instance is a (node of) a binary tree. */
public class Node<T> {
   /** value in this node */
   public T val;

   /** Left and right subtrees (null if empty) */
   public Node left, right;
}
```



Preorder, inorder, and postorder traversal

There are three major ways of traversing the tree, which differ only in when the root is processed. The names *pre*-order, *in*-order, and *post*-order indicate when the root is processed, as we see below. We define preorder traversal recursively like this:

preorder: process the root;
process the left subtree (in preorder);
process the right subtree (in preorder).

To the right is a concrete example of a method that does a preorder traversal to print the values in a tree. In all these methods, we assume that parameter `t` is not null. We discuss a call and show what is printed when parameter `t` is the root `A` of the tree shown above.

```
/** Print the tree values in preorder.*/
public static void printPre(Node t) {
   System.out.println(t.val);
   if (t.left != null) printPre(t.left);
   if (t.right != null) printPre(t.right);
}
```

The println statement at the beginning of `printPre`'s body prints `A`.
Next, `printPre` (`t.left`) is called. This call prints the values in subtree `B` in preorder. That is `B`, `D`, `E`.
Next, `printPre` (`t.right`) is called. This call prints the values in subtree `C` in preorder. That is: `C`, `F`.

Thus, the values printed are: `A`, `B`, `D`, `E`, `C`, `F`.

Inorder and postorder are similar and do not need much discussion.

inorder: process the left subtree (in inorder);
process the root;
process the right subtree (in inorder).

To the right is an example of a method that does an inorder traversal to print the values in a tree. Here are the values printed when parameter `t` is the root `A` of the tree should above: `D`, `B`, `E`, `A`, `C`, `F`.

```
/** Print the tree values in inorder. */
public static void printIn(Node t) {
   if (t.left != null) printIn(t.left);
   System.out.println(t.val);
   if (t.right != null) printIn(t.right);
}
```

postorder: process the left subtree (in postorder);
process the right subtree (in postorder).
process the root;

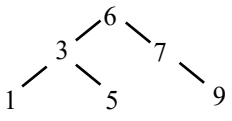
To the right is an example of a method that does a postorder traversal to print the values in a tree. Here are the values printed when parameter `t` is the root `A` of the tree should above: `D`, `E`, `B`, `F`, `C`, `A`.

```
/** Print the tree values in postorder. */
public static void printPost(Node t) {
   if (t.left != null) printPost(t.left
   if (t.right != null) printPost(t.right); );
   System.out.println(t.val);
}
```

Discussion

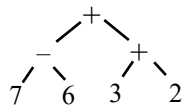
Each of these three tree traversals has its uses. Here's one use of an inorder traversal.

Look at the tree to the right. For each node `n` (say), all the values in `n`'s left subtree are less than `n`'s value, and all the values in `n`'s right subtree are greater than `n`'s value. Therefore, an inorder traversal, using method `printIn` given above, will print the values in ascending order! A tree with the property stated above is called a *Binary Search Tree*, or *BST* for short.



Traversing binary trees

Here's another example. The tree to the right represents an expression with binary operations + and – and integer operands. Below, we give the preorder, inorder, and postorder traversals of this tree. We give two alternatives for the inorder representation. The first does not have parentheses, and therefore the order of evaluation of the operations is ambiguous (except for mathematical convention). The second has parentheses inserted to indicate the order of evaluation given by the tree to the right. It's easy to rewrite the inorder-traversal method to output the parentheses.



preorder: `+ - 7 6 + 3 2`
inorder: `7 - 6 + 3 + 2`
inorder: `( 7 - 6 ) + ( 3 + 2 )`
postorder: `7 6 - 3 2 +  +`

The preorder and postorder representations of the expression look strange. Why would one ever want that? We discuss the preorder and postorder representations of expressions in another place, showing how they have been and still are in heavy use!

Using instance methods

The static methods presented above perform preorder, inorder, and postorder traversals of the tree given as a parameter. We can also write the methods as instance methods within class `Node<T>`. These instance methods don't need parameter `t` because the node (and the tree of which it is the root) is the node in which the method resides.

To the right is method `printPre`, written in class `Node`, which prints the preorder of the tree with this node (the one in which this method resides) as the root. Thus, the fields in this node are used directly.

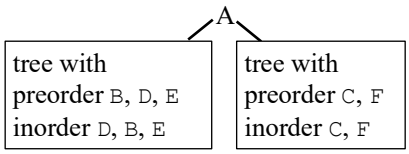
```
/** Print the values in this tree preorder. */
public void printPre() {
   System.out.println(val);
   if (left != null) left.printPre();
   if (right != null) right.printPre();
}
```

Creating a tree from two traversals

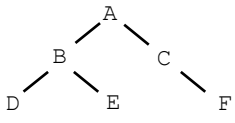
This is neat: from the preorder and inorder traversals of a tree (where nodes are uniquely identified), the tree can be unambiguously constructed. We show how, using these preorder and inorder tree traversals:

preorder: `A, B, D, E, C, F`
inorder: `D, B, E, A, C, F`

From the preorder, we know that the root is `A`. Then, from the inorder, we know that the left subtree contains everything to the left of `A` (i.e. `D, B,` and `E`), and the right subtree contains everything to the right of `A` (i.e. `C` and `F`). Therefore, the tree is as shown to the right. Now repeat the process to find the left and right subtrees from their preorders and inorders.



We show the final tree to the right.



In similar fashion, one can construct a tree from its postorder and inorder, since the postorder identifies the root as the last value. But one can't in general construct a tree from its preorder and postorder. Only the root is known, but that is all.