# Below, we show an expression in its usual inorder form, with parentheses being used to indicate the order in which operations are to be performed. Below that, we show the preorder and postorder versions of the expression. Note that parentheses are not needed.

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

# Why would we ever use the preorder and postorder versions? We discuss this below!

# Uses of preorder, or Polish Notation (PN)

# Preorder notation was invented by the Polish logician Jan Łukasiewicz, who invented it (and also postorder) in 1924. Therefore, it is sometimes called *Polish notation*, or simply *PN*. He came upon the idea of prefix notation and like it because it’s parenthesis-free —parentheses are not needed.

# Most method calls are written in prefix notation. For example, we write in Java Math.max(b, c). Here, Math.max is the operator and b and c are the operands. So, you have been using prefix notation for most of you programming life.

# Some languages, like Lisp and Scheme, use prefix notation for *everything* because the syntax is simple and easy to describe. Moreover, building a “syntax tree” from a program is almost trivial compared to a language like Java.

# To the right is a definition of the factorial function written in Lisp. Lisp was developed by John McCarthy, who is known as the father of AI (artificial intelligence) in 1958. It is a functional language —almost everything is done without the assignment statement. Basically, a program can be considered to be a bunch of recursive functions. The if-expression shown to the right is the equivalent of the Java conditional expression b ? e1 : e2 .

(defun factorial (n)

(if (= n 0)

1

(\* n (factorial (- n 1)))))

# “Lisp” is short for *LISt Processor*. Some view “Lisp” as an acronym for *Lisp Is Simply Perfect*, but its use of parentheses around everything has inspired other interpretations, like *Lost In Stupid Parentheses*, *Lots of Isolated Silly Parentheses*. But don’t knock Lisp, it was a tremendous invention, highly significant, and ideas in it form the backbone of many other languages.

# Uses of postorder, or Reverse Polish Notation (RPN)

# 

# 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.

# 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 preorder).

# 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.

# 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.

# 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.

6

7

3

5

1

9

# 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.

–

+

+

2

3

7

6

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

A

tree with

preorder B, D, E

inorder D, B, E

tree with

preorder C, F

inorder 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.

B

A

D

E

C

F

# 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.