Expression Trees, Binary Search Trees, and Tree Traversal
Advanced Computer Programming

We now look more specifically at two special kinds of binary trees. These two types of trees are very important in computer science.

I. Expression Trees

An expression tree represents an algebraic expression in such a way that stores its structure and shows how the order of operations applies.

We're interested in a few different operators. We break these operators down into two categories:

- **Binary Operators** - operators that take two inputs
  - +
  - – (here, subtraction)
  - *
  - `/` (both integer and floating-point division)
  - `%` (modulus)
  - `^` or `**` (exponentiation)

- **Unary Operators** - operators that take one input
  - -(here, negation)

Note that we don't mention parentheses. The expression tree's structure removes the need to talk about parentheses, as the structure encodes precedence.

When we have a single expression based on a binary operator, we draw the expression tree as follows:

- The operator is the **root** of the tree.
- The operands are the **children**. Because some operations are **not** commutative, order does matter. The operand before the operator is the **left child** and the operand after the operator is the **right child**.

Thus, we get a tree with a root and two children.

**Example 1:** Expression tree for \(x + y\):

When we have a single expression based on a unary operator, we draw the expression tree as follows:

- The operator is the **root** of the tree.
- The operand is the **child**.

Thus, we get a tree with a root and one child. (It's really more of a linear structure than a tree, but it does fit the definition of a tree. We'll find that these kinds of trees are interesting when we join them together as part of more complicated expressions.)

**Example 2:** Expression tree for \(-x\):

Note that we could treat negation as multiplication by \(-1\) and eliminate the need for unary trees if we'd like to have all nodes in our tree having exactly 2 children (or no child).
When we wish to work with more complicated expressions, we invoke the recursive nature of binary trees. When an operand is an expression rather than a single variable or constant, we simply put the expression tree for that expression in lieu of the operand.

**Example 3:** Expression tree for $x + y \times z$:

**Example 4:** Expression tree for $n \times (n + 1)$:

There's no reason we can't go to more levels…

**Example 5:** Expression tree for $(n \times (n + 1)) / 2$:

**Example 6:** Expression tree for $-(x + y) - (y / z)$:

**Example 7:** Expression tree for the right-hand side of the quadratic formula: 
$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$ To simplify things, let's work with just the + case.
So, why would anyone care? Well, when expressions are coded in a programming language, the code needs to be compiled into an executable program. In this process, the compiler needs to look through the code in processes called scanning and parsing. In the parsing part of compilation, the compiler interprets any expressions in the code and stores them in an expression tree. In theory, an expression tree could be exponentially large, so this process allows the computer to handle rather complicated expressions.

**Question:** Can we store an exponentially large expression on a computer?

Now then, I still haven't told you why exactly an expression tree is useful. That requires us to make an additional distinction. An expression can be described in two different ways: in terms of its **syntax** and in terms of its **semantics**.

**Question:** What is the difference between **syntax** and **semantics**, both for natural language and for formal languages like mathematical expressions and programming languages?

When a compiler scans a program and finds expressions, it is only interested in the syntax. As long as an expression is legal syntactically, it gets past the scanning process. That doesn't mean it has any meaning yet. So the expression is stored in an expression tree (encoding the rules of operator precedence) and later meaning (the semantics) is attached to it to, for example, say that \( x+y \) means to **add** \( x \) and \( y \), as opposed to any other meaning.

Incidentally, while on the surface, specifying the semantics of programming languages doesn't seem like it's all that big of a deal, it is a far more complicated process than you might think. It has been done for "toy" languages, but expressing the semantics of programming languages that are actually used in practice remains an unsolved problem (or at least it still was when I last checked in April 2004). I did research in this area and I've dropped my attempt at the semantics of part of Java on the resources page of the course site for the curious among you. This particular area might be something someone wants to pursue as a project and I'd be happy to ramble on at length about it with you.

Enough said there. We'll leave the scanning and parsing and formal languages for later courses.

### II. Binary Search Trees

A **binary search tree**, sometimes abbreviated BST, is another special kind of binary tree that allows us to store data. We'll look at using a BST to store a list of numbers.

To build a binary search tree from a set of input numbers:
1. Make the first input the **root** of the BST.
2. For each remaining input, recursively compare the input to the root of the tree.
   a. If the input is **less than** the root, it becomes the **left child** of the root (or, recursively, it goes into the left subtree.)
   b. If the input is **greater than** the root, it becomes the **right child** of the root (or, recursively, it goes into the right subtree.)

**Example 1:** Build a BST from the following lists:
   a. 6, 4, 7  
   b. 6, 4, 7, 2, 5, 9
We can build much more complex BSTs:

**Example 2:** Build a BST from a long list of numbers generated in class.

The list of inputs:

The result BST:

**Problem:**

a. Build a BST from these inputs: 10, 20, 30, 40, 5, 8, 50, 60, 70, 15, 80

b. Comment on the shape of the BST.

Now then, it's worth noting how BSTs can be used. While we could certainly use a BST to store a list of numbers, it's really the meaning of those numbers that makes a BST useful. We really want to use a BST to store records. But, in practice, we don't really store an entire record in a node of a BST; we instead store some key to the record.

**Question:** What is a primary key in a database engine such as Microsoft Access?

So, we store keys to records in a tree and use the structure of a binary tree to locate a record easily. That's why it's called a binary search tree.
**Question:** Suppose we have a binary search tree already in place. How would we go about looking for a number (or the key to a record)?

### III. Tree Traversal

One a tree is in place, we can traverse or walk the tree to list the elements of the tree. There are three kinds of traversals. The first is called an inorder traversal of the tree.

**Algorithm: Inorder Traversal(Tree T)**

1. Do an Inorder Traversal on the left subtree of T
2. Print the root of T
3. Do an Inorder Traversal on the right subtree of T

Notice the recursive nature of this procedure.

**Example:** Let's go back and do an inorder traversal on a BST from an earlier example.

The other two kinds of traversals are called preorder and postorder. In short, here's how all three go:

- **Inorder Traversal:** left, root, right
- **Preorder Traversal:** root, left, right
- **Postorder Traversal:** left, right, root

These kinds of traversals can be used for expression trees and they define different ways of writing expressions. Inorder traversal results in what we call an *infix* expression. That is, no doubt, the most natural way to write an expression, but its weakness is that it requires parentheses. The other two kinds of traversals produce expressions that can be written without any parentheses. Preorder traversal produces a *prefix expression* and postorder traversal produces a *postfix* expression. Many calculators (not the graphing kind) use postfix notation.

**Example 1:**

Generate a prefix expression for the expression in Part I, Example 3. Also generate a postfix expression.
Example 2:
Generate a prefix expression for the expression in Part I, Example 4. Also generate a postfix expression.

IV. The Tree Sort Algorithm

Question: Suppose we had a list of numbers we wanted to sort. How could we use a BST to do this?

Question: What advantages does this method have?

Problem: Write a recursive function for a tree sort. Assume you have a Tree class with logical operations defined.

Homework: Problems from Section 13.4 of Mizrahi/Sullivan: #25-32