Trees

Last class we introduced trees. We could, for example, define a tree node in Java as follows:

```
class  TreeNode<T>{
   T             element;
   TreeNode<T>   parent;
   TreeNode<T>   nextSibling;
   TreeNode<T>   firstChild;
}
```

The above class definition is not the only possibility. One can sometimes get away with a more basic tree node structure which has only the `firstChild` and `nextSibling` fields but not the parent field. Whether we want to drop the `parent` field or not depends on what problem we are solving. [ASIDE: the textbook introduces another possibilty, where an ArrayList is used to reference the children of a node.]

Binary Trees

The *order* of a (rooted) tree is the maximum number of children of any node. A tree of order n is called an n-ary tree. It is very common to examine trees of order 2. These are called *binary trees*. Each node of a binary tree can have two children, called the *left child* and *right child*. The terms "left" and "right" refer to their relative position when you draw the tree. Here is a typical definition of a binary tree node.

```
class  BTNode<T>{
   T             e;
   BTNode<T>   parent;
   BTNode<T>   left;
   BTNode<T>   right;
}
```

A binary tree is a special case of a tree, so the algorithms we discussed last lecture for computing the depth or height of a node and for traversing a tree apply to binary trees as well. There is one more traversal algorithm to be mentioned, though, which is used for binary trees, called *in-order traversal*.

```
inorderTraversal(tree, root){
  visit root.leftchild
  visit root
  visit root.rightchild
}
```

You could define an inorder traversal for general trees. For example, you could visit the first child, then visit the root, then visit any remaining children. The inorder traversal is not typically used for general trees, though.

Example

```
          a
        /   \
       b     c
      / \   / \
     d   e   f  g
```

```
level order:  a b c d e f g
pre-order:    a b d e c f g
post-order:   d e b f g c a
in-order:     d b e a f c g
```

Another example: Expression Trees

You are familiar with forming expressions using *operators* such as `+,-,*, /, ^` where the last one is the power operator i.e. `x ^ n` is `power(x,n)`. Each of the operators takes two arguments (called the left and right *operands*).

When you make a long expression out of operators and operands, there is a certain order in which the operators are applied. You learned this in grade school. For example "3 + 4 * 5" is interpreted as "3 + (4 * 5)" rather than "(3 + 4) * 5". There is also a common convention that "6 ^ 7 ^ 8" is interpreted as "6 ^ (7 ^ 8)" rather than "(6 ^ 7) ^ 8".

Now suppose you were given an expression such as

$$3 + 4 * 5 - 6 \,\hat{}\, 7 \,\hat{}\, 8$$

and you wanted to evaluate it. Knowing the precedence rules for the operators, it is possible (and not sooo difficult) to come with an algorithm for transforming any naked expression such as above to a *parsed* expression, in which the order of operations is expressed by the nesting of brackets:

$$(3 + (4 * 5)) - (6 \,\hat{}\, (7 \,\hat{}\, 8)).$$

In turn, it is possible to come up with an algorithm that takes such a bracketed expression as input, and outputs a binary tree. For the above example, the tree would be:

```
          -
        /   \
       +      ^
     /  \    /  \
    3    *  6    ^
        / \     / \
       4   5   7   8
```

To save time, I do not provide those algorithms here. At the end of this lecture, I will give you an algorithm for solving a problem closely related to the latter one.

"Expression trees" can be evaluated recursively as follows:

```
evaluate(expressionTree){
   if (expressionTree has a single node i.e. it is a leaf)
      return value of that node
   else{
      firstOperand = evaluate(left child of expressionTree)
      secondOperand = evaluate(right child of expressionTree)
      op = operand stored at root of expressionTree
      return  result of 'firstOperand op secondOperand' // e.g. '4 * 5'
   }
```

Note that this algorithm performs a postorder traversal of the tree. To evaluate the expression defined by a given sub-tree, you *first* need to evaluate the left and right child of that subtree, and *then* you evaluate the subtree by applying the operator at the root of the subtree to the evaluation result of the left and right child.

In-fix, pre-fix, post-fix expressions

We would like to be able to evaluate such expressions *without brackets* by scanning them left to right. As we know, however, this is non-trivial because often we are unsure whether we can apply an operator or not. For example, the expression begins with `3 + 4` but we do not evaluate this sum. The brackets make the problem easier, in that it is slighty easier to write down an algorithm for evaluating a bracketed expression. (It uses a stack.) However, this is still not so satisfactory as we would like to avoid using brackets. Is there an alternative? Yes, there is.

You are used to writing expressions as two operands separated by an operator. This representation is called *infix*, because the operator is "in" between the two operands. Alternatively, we could define a *postfix* expression, where the operators comes *after* the two operands.

```
((3 (4 5 *) + ) (6 (7 8 ^) ^ ) - )
```

or, without brackets

```
3 4 5 * + 6 7 8 ^ ^ - .
```

This is also known as reverse Polish notation.

One can of course also consider a *prefix* expression, where the operator comes *before* the two operands. e.g.

```
(- (+ 3 (* 4 5))(^ 6 (^ 7 8)))
```

or, without brackets:

```
- + 3 * 4 5 ^ 6 ^ 7 8
```

Converting a postfix expression into an expression tree

Let's now look at a simple algorithm for converting a (postfix) expression into a binary tree. The algorithm uses a stack.

```
buildExpressionTree( e ){  //  an expression with n terms
  S = new empty stack
  for i = 0 to n-1
    if e(i) is a number{          // rather than an operator
      t = new empty binary tree
      t.addRoot( e[i] )    //  makes a root node whose element is e[i]
                           //  and left and right children are null
      S.push(t)
    }
    else {                 // e(i) is an operator
        t2 = S.pop()
        t1 = S.pop()
        S.push( makeBT( e(i), t1, t2) ) //  makes a binary tree whose
      }                                 //  root node has element e(i)
  }                                     //  and whose left and right
  return S.pop()                        //  subtrees are t1, t2
}
```

Example

```
Input expression:   3 4 5 * + 6 7 8 ^ ^ -

CONTENTS OF STACK AS ALGORITHM GOES THROUGH FOR LOOP

 3
 34
 345
 3a        a is tree defined by 45*
 b         b  "  "    "         3a+
 b6
 b67
 b678
 b67c      c       "            78*
 b6d       d       "            7c^
 be        e       "            6d^
 f         f       "            be-
```