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 use 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.

Number of nodes of a binary tree

How many nodes can a binary tree have at each level? The root has one node. Level 1 has two nodes (the two children of the root). Level 2 has four nodes, namely each child at level 1 can have two children. You can easily see that the maximum number of nodes doubles at each level, and so level l can have 2^l nodes. For a binary tree of height h, the maximum number of nodes is thus:

$$n_{max} = \sum_{l=0}^{h} 2^l = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1.$$

You have seen this geometric series several times, and you will see it again...

The minimum number of nodes in a binary tree of height h is of course $h+1$, namely each node has at most one child (the sole leaf has no children). It follows that

$$h + 1 \leq n \leq 2^{h+1} - 1$$

We can rearrange this equation to give

$$\log(n + 1) - 1 \leq h \leq n - 1.$$

These inequalities be very useful in the coming lectures, when we use binary trees to solve many problems.

Binary tree nodes: Java

Last lecture we looked at the first-child/next-sibling data structure for general trees. For binary trees, one can use a (conceptually) simpler data structure:

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

One can have a `parent` reference too, if necessary, but we don't use it now. A `parent` reference is analogous to a `prev` reference in a doubly linked list.

Binary tree traversal

A binary tree is a special case of a tree, so the algorithms we have discussed for computing the depth or height of a tree node and for traversing a tree apply to binary trees as well. We saw two simple depth-first search algorithms for general trees, namely pre- and post-order, and for binary trees they can be written as follows:

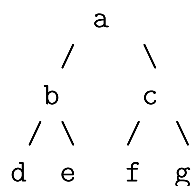
```
preorder(root){
    if (root is not null){              // base case
       visit root
       preorder(root.left)
       preorder(root.right)
    }
}

postorder(root){
    if (root is not null){              // base case
       postorder(root.left)
       postorder(root.right)
       visit root
    }
}
```

For binary trees, there is one further traversal algorithm to be considered, which is called *in-order traversal*.

```
inorder(root){
    if (root is not null){              // base case
       inorder(root.left)
       visit root
       inorder(root.right)
    }
}
```

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. But such inorder traversals are typically do not done for general trees.

Example



```
level order:  a b c d e f g   (breadth first)
pre-order:    a b d e c f g   (depth first)
post-order:   d e b f g c a       "
in-order:     d b e a f c g       "
```

Expressions

You are familiar with forming expressions using *operators* such as `+,-,*, /, ^`. (The operator `^` 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*. Let's define a set of simple expressions recursively as follows, where the symbol `|` means *or*:

```
baseExpression  =  digit | letter
operator        =  + | - | * | / |  ^
expression      =  baseExpression |  expression operator expression
```

Examples[1] of a `baseExpression` are "3", "x", "a", namely a base expression can be either a number (one digit) or a variable (one letter). An `operator` is a binary operator. An `expression` can consist of either a base expression, or it can consist of one expression followed by an operator followed by an expression. *Notice that expressions are defined recursively, and that we have a base case.*

Such formal definitions are extremely useful and common in computer science. Indeed every programming language you will likely ever work with is defined in this way. You will start learning this stuff in COMP 330. You can google "context free grammar" if you want to have a peek ahead.

Expression trees

You can represent these expressions using trees, called *expression trees*. For example, the expression `x + 4 * y` could be defined either of these two trees:

Typically though, when we have an expression with multiple operators, there is a particular order in which the operators are supposed to be applied. You learned these precedence orderings in grade school. For example "`x + 4 * y`" is to be interpreted as "`x + (4 * y)`" shown on the left, rather than "`(x + 4) * y`" shown on the right. (There is also a convention that `6 ^ z ^ 8` means `6 ^ (z ^ 8)` rather than `(6 ^ z) ^ 8`, although some may use just the opposite convention (it is arbitrary).

The above precedence order implies that an expression such as

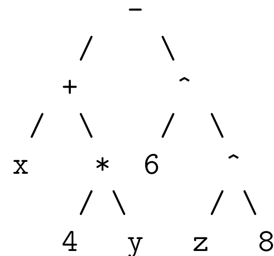
$$x + 4 * y - 6 \,\hat{}\, z \,\hat{}\, 8$$

[1] In an earlier version of the notes, I also wrote "-16" is an example. That was a typo.

can be uniquely interpreted as if there were a nesting of brackets:

$$(\mathtt{x} + (\mathtt{4} * \mathtt{y})) - (\mathtt{6} \mathbin{\hat{}} (\mathtt{z} \mathbin{\hat{}} \mathtt{8}))$$

and the expression can be represented as a tree:



Expression trees can be evaluated recursively as follows.

```
evaluateET(root){
   if (root is a leaf)   // can be determined by checking if it has
                         // any children
      return value
   else{ //  the root is an operator
      firstOperand = evaluateET(left child of root)
      secondOperand = evaluateET(right child of root)
      evaluate 'firstOperand op secondOperand' and return result
   }
```

We may think of this algorithm as performing a *postorder* traversal of the tree in the sense that, to evaluate the expression defined by a tree, you *first* need to evaluate the left and right child of the root, and *then* you can apply the operator at the root.

[**The following material was presented at the beginning of the following lecture. I am keeping it here so that the lecture notes are grouped by topic.**]

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

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. For infix expressions, the order of evaluation is determined by precedence rules.

An alternative way to write an expression is to use *prefix* notation. Here the operator comes *before* the two operands. For example,

```
- + x * 4 y ^ 6 ^ z 8
```

which is interpreted as

```
(- (+ x (* 4 y))(^ 6 (^ z 8))) .
```

Notice that a prefix expression gives the ordering of elements visited in a preorder traversal of the expression tree.

An second alternative is a *postfix* expression, where the operators comes *after* the two operands, so

```
x 4 y * + 6 z 8 ^ ^ -
```

is interpreted as

```
((x (4 y *) + ) (6 (z 8 ^) ^ ) - ) .
```

Notice that the ordering of elements is the visit order in a post-order traversal of the expression tree.

One can formally define a set of *in, pre, and post*fix expressions recursively as follows:

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
infixExpression   =  baseExpression |  infixExpression operator infixExpression
```

which is just what we saw earlier in the lecture (when all expressions were infix).

One modifies the definition slightly for prefix expressions:

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
prefixExpression   =  baseExpression |  operator prefixExpression prefixExpression
```

and for postfix expressions

```
baseExpression    =  digit | letter
operator          =  + | - | * | / |  ^
postfixExpression =  baseExpression |  postfixExpression  postfixExpression operator
```

ASIDE: Prefix notation is sometimes called “Polish” notation – it was invented by a Polish logician, Jan Lukasiewicz (about 100 years ago). Postfix notation is sometimes called “reverse Polish notation”.

The advantage of postfix and prefix expressions over infix expressions is that you do not need a precedence rule to define the order of operations. It is easiest to see the usefulness of this with postfix expressions. Here is a stack-based algorithm for evaluating a postfix expression. The algorithm assumes the expression is a list of variables, numbers, and operators, specifically, binary operators i.e with two operands. It does not need to know about precedence orderings, since these are “built in”.

See the slides for an example of how the stack evolves over time.

```
s = empty stack
cur = head;
while (cur != null){
   if (cur.element is a variable or number)
      s.push(cur.element)
   else{ //  cur is an operator
     operand1 = s.pop()
     operand2  = s.pop()
     operator  = cur.element
     s.push( evaluate( operand1 operator operand2 ) )
   }
   cur = cur.next
}
```

ASIDE: In the 1970's Hewlett-Packard introduced the first desktop calculators. These required that users enter expressions using postfix! Ask your parents (or grandparents?).