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*.

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.

How many nodes can a binary tree have at each level? The root has one node. Level 1 can have two nodes (the two children of the root). Level 2 can have 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 if node has at most one child and there is one leaf, which by definition has no children. It follows that

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

ASIDE: We can rearrange this equation to give

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

Binary tree nodes: Java

We have seen the first-child/next-sibling data structure for general trees. For binary trees, one typically uses the following data structure instead:

```
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. Recall that 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:

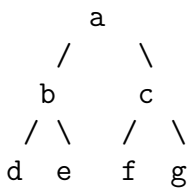
```
preorderBT(root){
    if (root is not null){              // base case
       visit root
       preorderBT(root.left)
       preorderBT(root.right)
    }
}

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

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

```
inorderBT(root){
    if (root is not null){              // base case
       inorderBT(root.left)
       visit root
       inorderBT(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 not done for general trees.

Example (different from one given in slides)




```
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 *binary 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*. (This is similar to the "grammar" we saw in Assignment 2 Question 1.)

```
baseExpression  =  variable | integer      (e.g. definition from Assignment 2)
operator        =  + | - | * | / | ^
expression      =  baseExpression |  expression operator expression
```

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.

Expression trees

[In the slides, I used slightly different examples from what I use below.]

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:

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, although as I mentioned in the lecture, my MS Windows calculator ignores the precedence ordering of * over +. There is also a convention that `6 ^ z ^ 8` means `6 ^ (z ^ 8)` rather than `(6 ^ z) ^ 8`. See the example that I gave in the slides.

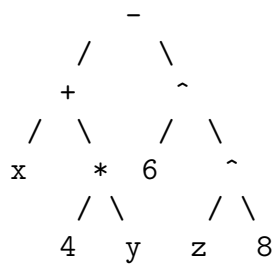
The above precedence order implies that an expression such as

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

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

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

and the expression can be represented as a tree:



You may be tempted to think that the + operator should be in the root rather than the - operator being in the root because the + operator is to the left and therefore has precedence (and this question came up during the lecture too). In fact, it works the other way: higher precedence means it is evaluated first, which means it is deeper in the 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)
      return evaluate(firstOperand, root, secondOperand)
   }
```

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.

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
prefixExpression  =  baseExpression |  operator prefixExpression prefixExpression
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" or RPN. Many calculators, in particular, Hewlett-Packard calculators require that users enter expressions using RPN. Apparently one learns very quickly how to do it.]

For computer science, the advantage of postfix (and prefix) expressions over infix expressions is that with postfix expressions you do not need a precedence rule to define the order of operations. In particular, below is a simple stack-based algorithm for evaluating a postfix expression. The algorithm does not need to know about precedence orderings, since these are "built in" to the postfix expression. See the slides for an example of how the stack evolves over time for a particular expression.

```
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
     operand2 = s.pop()    // opposite order to push
     operand1 = s.pop()    //   "
     operator = cur.element
     s.push( evaluate( operand1 operator operand2 ) )
   }
   cur = cur.next
}
```