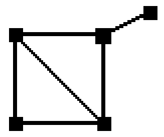
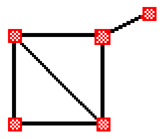
Graph Theory and Trees

Graphs

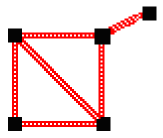
A *graph* is a set of nodes which represent objects or operations, and vertices which represent links between the nodes. The following is an example of a graph because is contains nodes connected by links.



Node / Vertex A *node* or *vertex* is commonly represented with a dot or circle. The node itself often represents something else, such as data, or an operation or activity. In the example here, nodes are highlighted in red.



Link / Edge A *link* or *edge* connects nodes. A link represents some kind of reference between nodes. In the example here, links are highlighted in red.

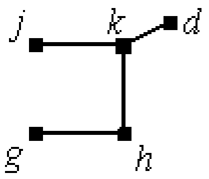


Trees

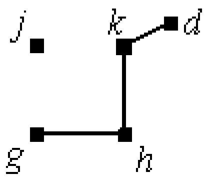
A *tree* is a special kind of graph follow a particular set of rules.

Connected A graph can be a tree if is *connected*. That is, if each of the nodes is connected with a link to at least one other node. If a node is not connected to some other node, then the assembly is not a tree.

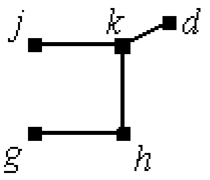
Here is an example of tree because it is connected.



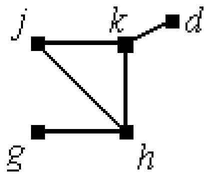
Here is an example of a graph that is not a tree because it is not connected. (Notice that node *j* is not connected to any other node.)



Acyclic A graph can be a tree if is *acyclic*. That is, if there is one and only one route from any node to any other node. Here is an example of a tree because it is acyclic:

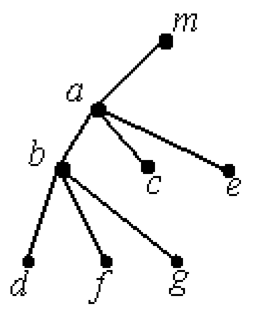


Below is an example of a graph that is not a tree because it is not acyclic. Notice that there is more than one route from node *g* to node *k*.



A tree does not have a specific direction. Depending on how it is to be used, the tree may branch outward while going upward (like a real tree – the growing kind), or it can branch down like the roots of a real tree.

Root The term *root* commonly refers to a top-most node (when the tree extends downward). A root node may be terminal (as defined below), or it may be internal (also defined below). Node *m* in the example is the root node.



Example Tree

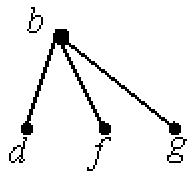
Descendant A *descendant* is a node which is further away from the root than some other node. The term descendant is always in reference to another node. In the example here, node *g* is a descendant of nodes *b, a*, and *m*.

Parent A *parent* is a node which closer to the root node by one link or vertex. In the example, node *b* is the parent of node *f*. Similarly, node *a* is the parent of node *b*.

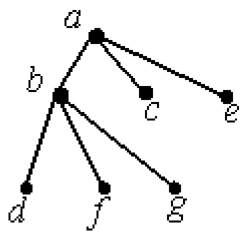
Sibling A *sibling* is a node which shares the same parent. In the example, nodes *b* and *c* are siblings because they both have node *a* as their parent.

Ancestor An ancestor is any node between a given node and the root, including the root. In the example, the ancestors of node f are nodes *b*, *a*, and *m*.

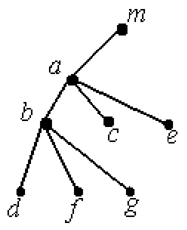
Sub-tree A sub-tree is (usually) a smaller portion of a tree starting at some specified node. In the example, the sub-tree rooted at *b* would look like this:



The sub-tree rooted at node *a* would look like this:



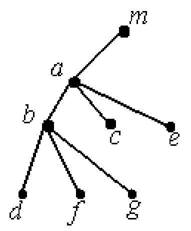
Terminal node/vertex A node is said to be terminal if it has no children. In the example below, nodes *d, f, g, c* and *e* are terminal nodes.



Internal node/vertex A node is said to be internal if it is not a terminal node. In the example, nodes *m, a*, and *b* are internal nodes.

The total number of nodes is the number of internal nodes plus the number of terminal nodes.

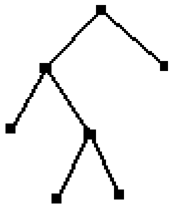
Height The height of a tree is defined as the number of vertices or edges traversed to get to the most distant node. In the example, the height of the tree is three (3).



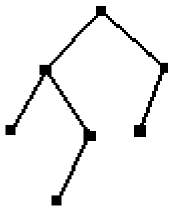
Binary Trees

Up until now, a tree could have any number of branches (a node could have any number of children). A binary tree is a special tree which limits the number of children to a maximum of two.

Full Binary Tree A binary tree is said to be full if each node has either zero or two children. The example here is a full binary tree.



Here is an example of a binary tree that is not full. Notice the nodes with only one child.

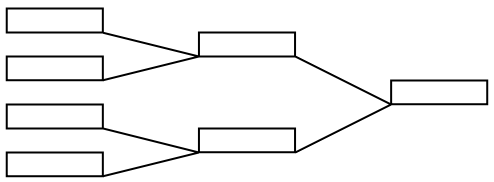


There is a relationship between the *tree height* and the *maximum* number of terminal nodes. That relationship is:

Maximum terminal nodes = $2^{\text{tree height}}$

Thus, the maximum number of terminal nodes of a binary tree with a height of 4 would be 16 (=2^4).

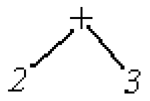
Information in trees can be just about anything we can think of. Terminal nodes may be data items, while internal nodes are simply used as connectors. When using binary trees it is not unusual to use the internal nodes as events, which produce new results. An example is a tournament tree (yet another direction for a tree to travel – sideways).



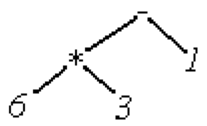
Each internal node creates a new winner which meets the winner of the adjacent contest. The root, then, is the tournament winner.

Traversing Binary Trees

Yet another use for a binary tree is to represent arithmetic operations, such as 2+3, or 6*3-1. We can represent these in a binary tree and then traverse and interpret the results in a variety of ways. The notations we will examine place values at terminal nodes and operators in internal nodes. For example, we can represent the expression 2+3 using a binary tree like this:



Similarly, we can represent the expression 6*3-1 using a binary tree this way.



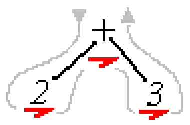
In this case, we are going to process the multiplication first. Like a tournament, the last operation goes in the root node.

In order to make sense of the tree we must establish how we are going to examine and process the information in the nodes. There are three methods.

In-fix / In-order[1]

Traversing a binary tree using infix or in-order produces a sequence of values and operators almost identical to what we started with. Start from the upper left side of the root node, and completely trace the outside edge of the tree and the nodes. With in-fix / in-order, the node content is recognized / processed as you pass to the right, going underneath the node. You can stop when you get to the right side of the root node.

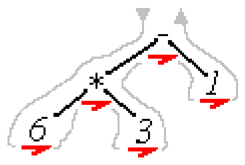
In the case of the first tree, we get this:



2 + 3

In the second tree, we get this sequence:

[1] Spelling of the traversing methods, especially with reference to hyphenation, vary. There does not appear to be a commonly accepted standard. Other texts and references may or may not use hyphens.



6 * 3 - 1

Notice that the operator can be directionally sensitive (which is to say, as in subtraction, *what gets subtracted from what* changes the answer).

Parentheses and Precedence

We got lucky in the last example. If we were to simply reorder the operators in the expression to 1-6*3, we would get a very different result. As you can see, we would be subtracting 6 from 1, and then multiplying by 3. So how can we be sure to get the solution method we are looking for?

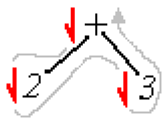
The answer is precedence. That is, some operators get processed before others. In this case, multiplication and division get processed before addition and subtraction.

We can also force precedence by grouping an expression in a set of parentheses. The expression 3*(6-1) forces us to process the subtraction before the multiplication. Precedence functions this way by implying grouping. Thus, the way most of us learned arithmetic, higher precedent functions like multiplication and division are automatically recognized, but computers require that we be more explicit.

For this reason, one thing that some expression processors do is parenthesize based on precedence. This allows another part of the program to ignore the problem of precedence completely and just pay attention to grouping with parentheses. Thus, our original expression, 6*3-1, would be rewritten as ((6*3)-1). While the last set of parentheses may seem excessive, it also doesn't hurt[2].

Prefix / Pre-order

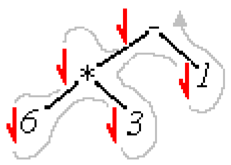
Traversing a binary tree using prefix or pre-order means to trace the outline of the tree, again starting from the upper left next to the root, identifying nodes as you travel downward on the left side of the node. Thus, the first node is always the root node. Here is an example of the tree using 2+3:



+ 2 3

[2] It is not uncommon to add an additional programming layer in a numeric or symbolic parser that looks for excessive parentheses pairs, such as ((5+2)) and removes them. It can add time on the front end of a compiler, but trim program runtime.

A way to interpret this is linguistically; add the values of two and three. Obviously things get more complicated (and un-earthly?) as the expression gets more complicated. Here is our other example:

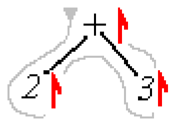


- * 6 3 1

"Subtract the result of multiplying 6 and 3 by 1". (I didn't say it would make sense to us, but it might make sense to a computer.)

Polish Notation
This notation was discovered in the 1920's by Jan Lukasiewicz, a Polish mathematician. One of the useful properties of this notation is that it can be processed by a computer or other computing machine. How do you think it might work?

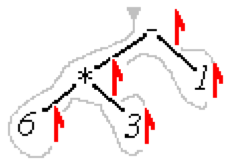
Postfix / Post-order
Traversing a binary tree using postfix or post-order means to trace the outline of the tree, again starting just to the left of the root node, and going downward. You identify node contents when the trace passes upward on the right side of the node. The root node is always the last node. Here is our example of 2+3:



2 3 +

One way to interpret this is: Using 2 and 3, add them together.

This produces a result which at first looks like just the opposite of prefix/pre-order, but it is not. Notice that the sequence of the values is the same, but the placement of the operator has changed. This becomes more obvious when we perform the same operation on our other expression tree:



6 3 * 1 -

One way to interpret this is: using 6 and 3, multiply them, then using 1, subtract. (Still potentially confusing, but it gets better.)

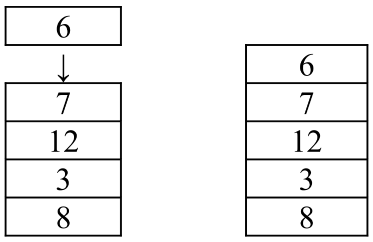
Reverse Polish Notation (RPN)

Similar to Polish notation, Reverse Polish Notation was invented in the 1950's by Charles Hamblin. It uses Polish notation, and reverses part of the process, placing operators *after* the values or terms. "So what?", you might ask.

Stack Based Processing

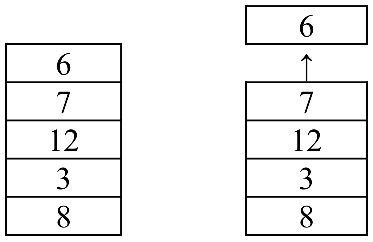
It turns out that one of the most important operational structures in modern computers is something called a *stack*. In a stack, values can be placed or removed only from the "top". There are commonly two operations which access the stack: push and pop[3].

The *push* operator places a value on the top of the stack, adjusting a memory pointer to the address. Usually, the address is just the next one in line, so the system only has to increment or decrement (depending on the system) to calculate the next storage location.



Push the value 6 onto a stack.

The *pop* operator uses the address defined by the push operator, and gets the value currently at that location. As part of the operator, the address is adjusted to its previous value by reversing the arithmetic of the push operation.



Pop the value from a stack.

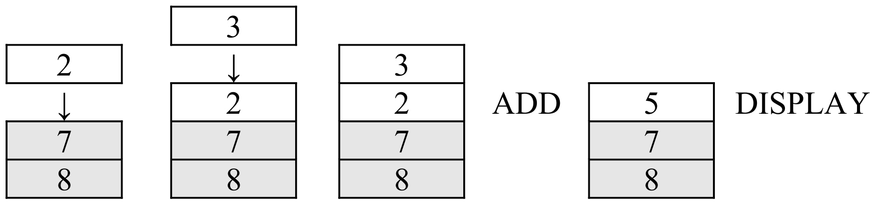
Arithmetic operations are performed by popping values from the stack, performing the calculation, and then pushing the result back onto the stack. A final operation pops the value on the stack for storage or display.

[3] For more information on stack-based processing and RPN, you may wish to check Wikipedia at http://en.wikipedia.org/wiki/Stack_(computing) .

Thus, our operations for 2 3 + could be translated into code…

```
PUSH 2
PUSH 3
ADD
DISPLAY
```

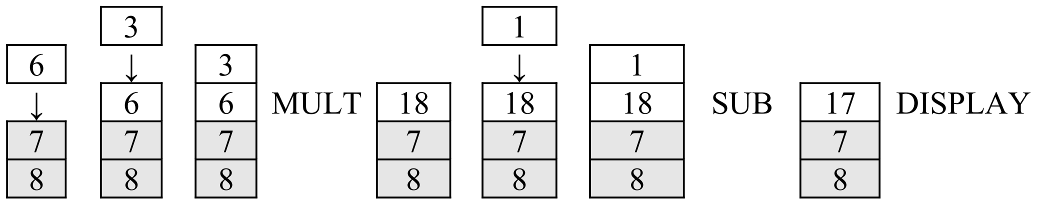
…and could look something like this[4]:



The operations for 6 3 * 1 – could look like this:

```
PUSH 6
PUSH 3
MULT
PUSH 1
SUBTRACT
DISPLAY
```

…and could look something like this:



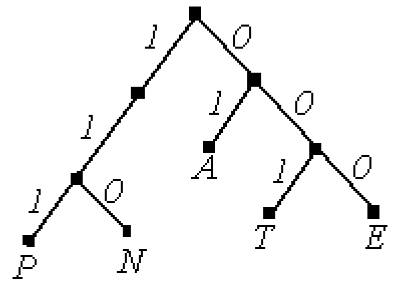
Reverse Polish Notation and stack based processing are very important ideas and will effect the way your programs work. By taking advantage of them you can make sure that you programs run the way you expect them to.

[4] The values 8 and 7 have been added to the stack as a way of showing previous values on the stack. We don't actually care about the previous values themselves, but wish to show that they are preserved. This is a significant and extremely useful property of stacks.

Huffman Code

Most data is stored in a binary system using fixed length binary string. ASCII is a fixed length format, using 7 (or 8) bits to identify specific characters. *Huffman codes* are special binary trees intended to encode data. By using a string of 1's and 0's, a user or program can locate a specific data element. The primary purpose is to allow paths of varying lengths.

The following is an example of a Huffman tree. Each node contains a character of the alphabet. Each link represents an option: either 0 or 1.



The letter 'P' can be reached by using the binary code 111. The letter 'N' can be reached using the binary code 110. In Huffman coding, not all nodes have the same length. In general, more common characters would have shorter binary or bit codes.

Words can be spelled using sequences of bit codes. For example, then word "PEN" can be represented by the bit sequence

$$\text{PEN} = 111000110$$

The bit sequence is scanned from left to right.

$$\text{PEN} = \underbrace{111}_{\text{P}}000110$$

$$\text{PEN} = 111\underbrace{000}_{\text{E}}110$$

$$\text{PEN} = 111000\underbrace{110}_{\text{N}}$$

The word "PAN" would be represented by the bit sequence 11101110. Notice that the 'A' only uses 2 bits in the sequence.

To encode a word, get the path for each letter, and string them in sequence. For example, the word "TEN" would be constructed as follows:

T = 001 E = 000 N = 110 TEN = 001000110 (= 001 000 110)