# Reverse polish notation

## Constructing a binary search tree

Suppose the following list of numbers is to be inserted into a binary tree, in the order given, in such a way that the tree can be quickly searched

17, 8, 4, 12, 22, 19, 14, 5, 30, 25

The algorithm is:

* Place the first item at the root
* Then for each item in the list: If the value of the new data item is less than the value in the current node then branch left, otherwise branch right.
* Keep repeating this process until you come to an ‘empty’ branch, then put the new value in the node at the end of the branch

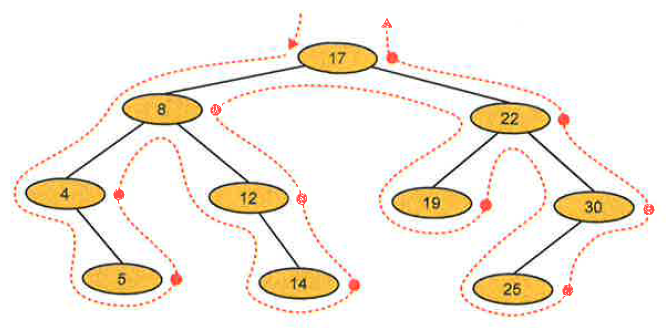
### Question

Using the algorithm and list of numbers above write the list of numbers in their correct node

## Traversing a tree (3 ways) In-order, Pre-order and Post order.

### Post order

Draw an outline around the tree, starting to the left of the root. As you pass to the right of a node (red dot), output the data in that node



Use: **Reverse Polish Notation**

## What is reverse polish notation?

It is a method of writing arithmetic expressions that is particularly suited to computerised methods of evaluation. For example, the expression 5+3 becomes 5 3 + (i.e. the “+” operator moves to the end and the operands are moved to the beginning). The advantages are:

* It eliminates the need for brackets
* It produces expressions in a form suitable for evaluation using a stack
* It puts the expression in a sequence that is more convenient for an interpreter

In primary and secondary school maths, you are taught “BODMAS” to figure out how to work out a normal arithmetic equation better known as **infix** notation e.g. (a+b)\*C. It is not easy for computers to evaluate an infix expression. The sequence you are taught in school (using BODMAS) to interpret this infix expression is:

1. Get a
2. Get b
3. Add them together and store the immediate result
4. Get c
5. Multiple by the result of step 3

A computer really needs the operands and operator in this sequence: **a b + c \***

## Question

What would be the computer sequence of instructions needed to evaluate the expression b \* c?

## Question

What would be the computer sequence of instructions needed to evaluate the expression b \* (c+d)?

## Infix and postfix expressions

An expression such as (9\*6)+5 is known as an **infix** expression, because the operator is written between the operands. The equivalent RPN form, 9 6 \* 5 + is known as a **postfix** expression, as the operator follows the operands.

## Converting expressions from infix to RPN

A computer will use a stack to translate from infix to RPN. A human can do it manually using these rules:

1. Starting from the left-hand side of the expression, allocate numbers 1,2,3 ... to operands and operators as follows:

* If the next symbol is an operand, allocate the next number (1,2,3...) to it. If it is an operator, move to the next operator
* Ignore brackets except if they affect the order of the calculation
* Bearing in mind the rules of precedence, decide which is the next operation that should be performed and as soon as its operands have been allocated numbers, back up and allocate it the next number.

e.g. a+b\*c

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| a | + | b | \* | c | This is the infix notation |
| 1 | 5 | 2 | 4 | 3 | This is the order of the postfix expression |

Working from the left, allocate 1 to a and 2 to b. Multiplication is done before addition, so keep going and allocate 3 to c. Then back up and allocate 4 to \*, and finally 5 to +.

Write down the tokens (operators and operands) in the order of the numbers you have allocated. The RPF of the expression is abc\*+

Example 1

Convert the following expression to RPN: 8+((7+1)\*2)-6

Following the rules above, taking into account order of precedence and brackets where they affect this:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 7 | 2 | 4 | 3 | 6 | 5 | 9 | 8 | infix |
| 8 | + | ((7 | + | 1) | \* | 2) | - | 6 | postfix |

Note: Taking into account, 7+1 is the first thing to be calculated and then this is multiplied by 2. Keep backing up everytime you have one or two operands which need to be evaluated next

The RPF of the expression is 871+2\*+6

## Convert the following infix expression to RPN

4 - 3 The answer is: 4 3 -

4 + 2

9 / 4

(4 + 3) \*2

(7+5) / (2+1)

Translate into RPN

(a+b) – x ∧ y\* 3 Where ∧ means “to the power of”

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  | infix |
|  |  |  |  |  |  |  |  |  | postfix |

Reverse Polish form is:

Translate into RPN

X = - a + (c – d) / e

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  | infix |
|  |  |  |  |  |  |  |  |  | postfix |

Reverse Polish form is:

## Translation from Reverse Polish to infix

Convert the following to infix: **25 16 18 + \* 12 -**

Visually scan the operands writing down until you find two operands followed by an operator. The unpaired operand **25** is written down.

Bracket the next two operands with the operator between them and add them to the expression hat is building up. We now have **25 (16+18)**.

Continue writing down operands until you find the next operator, which will operate on the two preceding operands, in this case \* operates on 25 and (16 + 18). This gives us **25\*(16+18)**.

The next symbol is an operand, so the following operator will operate on the two operands

25\*(16+18) and 12, giving the final result: **25\*(16+18) - 12**

### Convert the following RPN expressions to infix

2 4 / The answer is: 2/4

3 4 /

7 2 +

3 5 2 + \*

3 4 + 9 \*

6 3 + 7 2 - \*

## Evaluation of RPN expression using a stack

Once a compiler has translated an arithmetic expression into RPN, each symbol in the expression may be held in a string or array. The expression may then be evaluated using a stack, scanning the elements of the string (or array) from left to right as follows:

* If the next token is an operand, place it on the stack
* If the next token is an operator, remove the required number of operands from the stack, perform the operation and put the result on the stack.

Convert the following expression to RPN and show how the resulting expression may be evaluated using a stack.

(7 + 10/5) + (6 \* 2)

First convert to Reverse Polish:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 2 | 4 | 3 | 9 | 6 | 8 | 7 | infix |
| (7 | + | 10 | / | 5) | + | (6 | \* | 2) | postfix |

i.e. 7 10 5 / + 6 2 \* +

Using a stack to evaluate the expression, the contents of the stack will change as follows:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 5 |  |  |  | 2 |  |  |
|  | 10 | 10 | 2 |  | 6 | 6 | 12 |  |
| 7 | 7 | 7 | 7 | 9 | 9 | 9 | 9 | 21 |

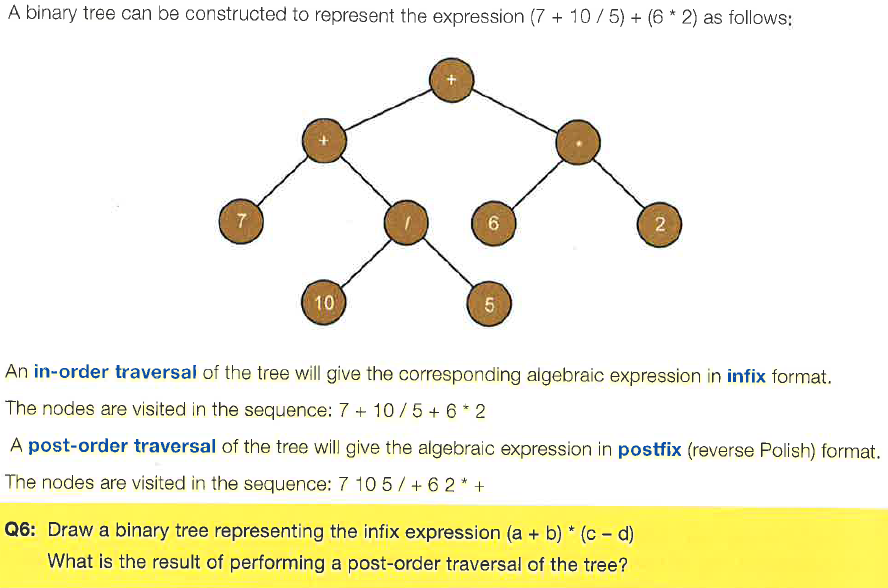
### Covert the expression (5+9) / 2 (2\*3) to RPN and show how it may be evaluated using a stack:

Reverse Polish form is

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

## A binary expression tree

(algorithm available on page one above)



The post-order traversal is: