Evaluation of postfix expressions

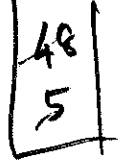
Example:	infix: 5 * 4 * 12 - 6	postfix: 5 4 12 * + 6 -

Algorithm: stack to hold operands.

for each token in input stream do
    if operand: push operand into stack
    else /* operator */: pop enough operands to perform operation, execute it and push result into stack.
stack should have exactly one operand = result.

Example:

4*12 = 48



5+48 = 53

53-6=47

Result = 47

Note: Algorithms for infix → postfix
postfix evaluation
infix → expression tree

do not handle errors.
Error handling is possible.

Infix to Expression tree

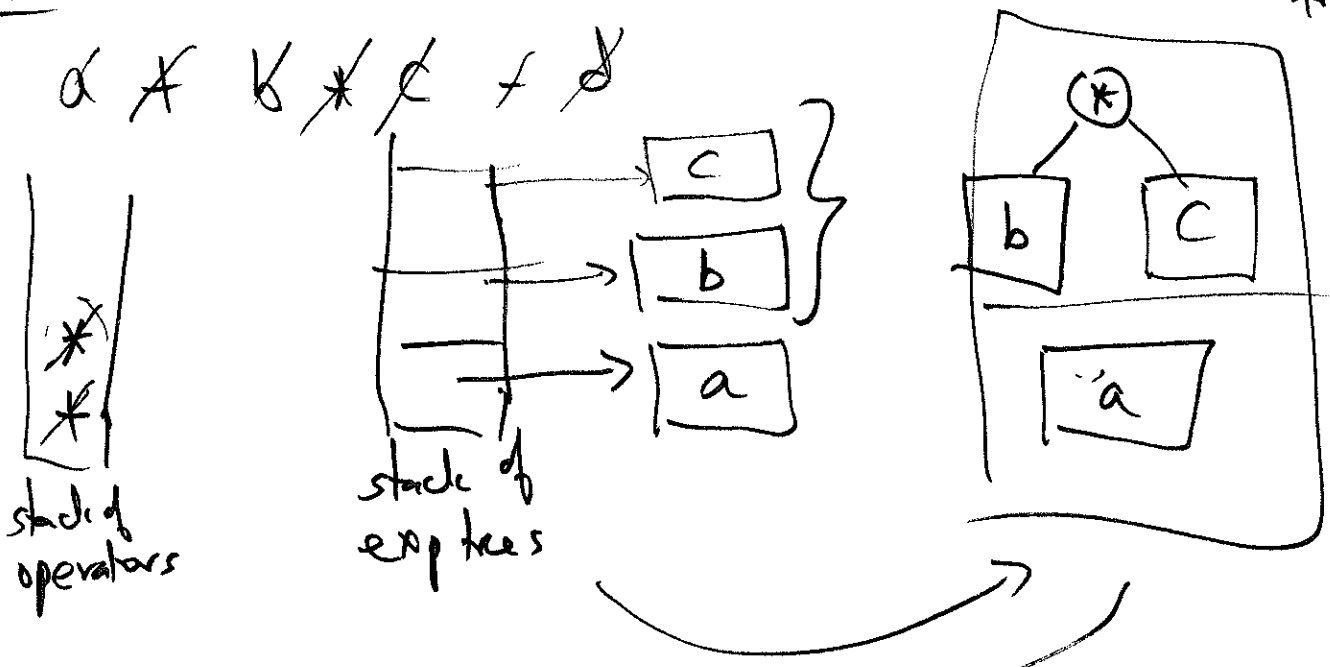
Similar to Infix to Postfix conversion.

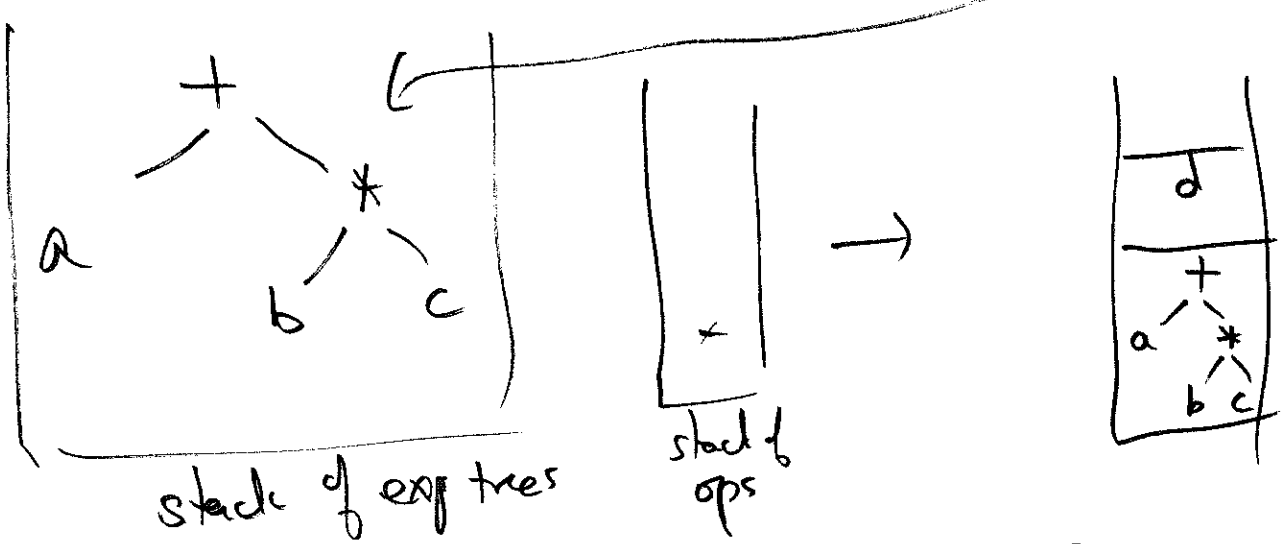
stack <operators> ~~Queue for Output~~

stack for expression trees.

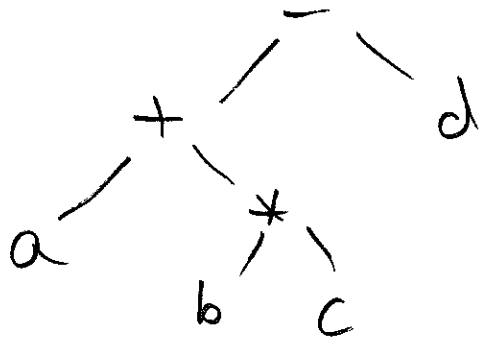
Ex:

a * b * c + d





Output expression. →



Evaluation of expression trees:

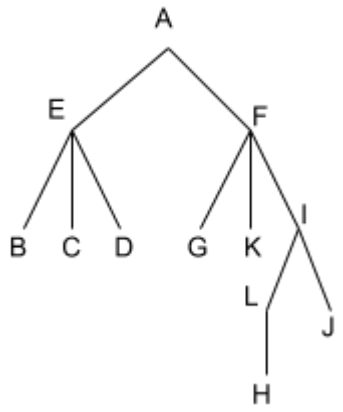
```
eval (tree):
    if tree is a leaf node (no children)
        return tree.element.number
    else  // operator node
        l ← eval (tree.left)
        r ← eval (tree.right)
        result ← l ⊕ r
                 ↑ operation specified by tree.element token.

        return result.
```

tree.element.getValue()

Trees: acyclic, connected graph (directed or undirected).

1. Undirected, unrooted tree: represented as a graph. Output of algorithms of Kruskal and Boruvka are list of edges forming MSTs. This can be converted into a rooted tree, by running DFS/BFS on this graph (induced by the edges of this tree).
2. Undirected, rooted tree: represented as binary tree or arbitrary tree. A special node is designated as root. Edges connect parent node to its children, and it takes O(1) time to iterate per child. Nodes with no children are called “leaves” or “leaf nodes”. Other nodes are “internal” nodes. It is optional to store information about a node’s parent.
3. Up tree: Each node knows its parent. A node does not store its children.
4. Directed tree (also known as branching or arborescence). Outgoing or incoming tree.

Rooted trees: Defined recursively as root node attached to zero or more subtrees.



Root = A. Leaves = { B, C, D, G, K, H, J }.
Children(A) = { E, F }. E and F are siblings. Parent(C) = E.
Descendants(F) = { F, G, K, I, L, J, H }. Ancestors(I) = { I, F, A }.
Proper descendants(I) = { L, J, H }. Proper ancestors(I) = { F, A }.

```
class Tree<T> {
   class Entry<T> {
      T element;
      List<Entry<T>> children;
   }
   Entry<T> root;
   int size;
}
```

Concepts:

Path: Sequence of edges to go from one node to another: F → I → L → H is a path of length 3 from F to H. The **length** of a path is the number of its edges.

Depth of a node is the length of the path from root of the tree to that node. dep(A): 0, dep(F): 1, dep(L): 3.

Height of a node is a maximum length of a path from the node to any of its descendants, i.e., length of path from the node to a farthest leaf node that is its descendant. So, height(G) = 0, height(F) = 3.

Subtree rooted at a node X is the part of the tree induced by X and its descendants. So, subtree of E is the tree rooted at E, connecting it to its descendants B, C, and, D.

$$\text{depth}(u) = \begin{cases} 0, & \text{if } u \text{ is root,} \\ 1 + \text{depth}(u.\text{parent}), & \text{otherwise.} \end{cases}$$

$$\text{height}(u) = \begin{cases} 0, & \text{if } u \text{ is leaf,} \\ 1 + \max_{c \in u.\text{children}} \text{height}(c), & \text{otherwise.} \end{cases}$$