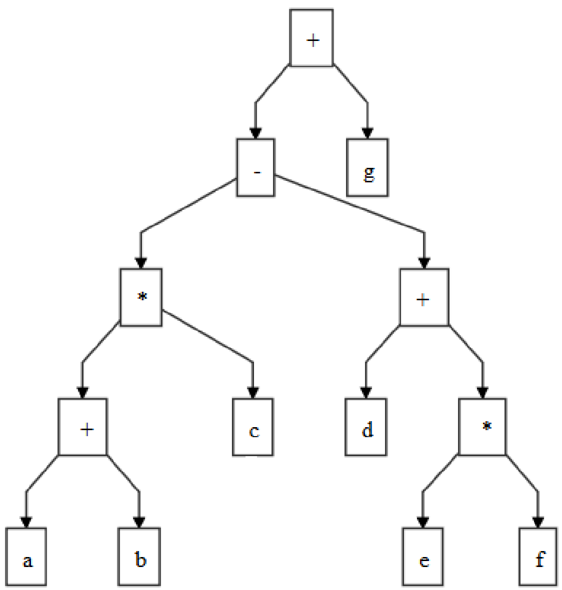
DSA - Seminar 7

1. Build the binary tree for an arithmetic expression that contains the operators +, -, *, /. Use the postfix notation of the expression.
   Ex: (a + b)* c – (d + e * f) + g =>
   Postfix notation: ab+c*def*+-g+

   The corresponding binary tree is:

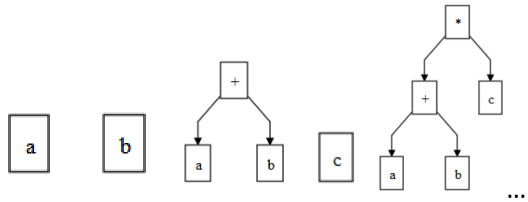


If we traverse the tree in postorder, we will get the postfix notation.

Algorithm:

1. Use an auxiliary stack that contains the address of nodes from the tree
2. Start building the tree from the bottom up.
3. Parse the postfix expression
4. If we find an operand -> push it to the stack
5. If we find an operator->
   a. Pop an element from the stack – left child
   b. Pop an element from the stack – right child
   c. Create a node containing as information the operator and the left and right child
   d. Push this new node to the stack
6. The root of th tree will be the last element from the stack.

Stack:



Assume we have a binary tree with dynamically allocated nodes.

<u>Node</u>:
e: TElem
left, right: ↑Node

<u>BT</u>:
root: ↑Node

The stack will contain elements of type ↑Node and we will only use the interface of the stack

- Init
- Push
- Pop
- Top

```
Subalgorithm buildTree (postE, tree) is:
    init(s)
    for every e in postE execute:
        if e is an operand then:
            alocate (newNode)
            [newNode].e ← e
            [newNode].left ← NIL
            [newNode].right ← NIL
            push (s, newNode)
        else
            pop(s, p1)
            pop(s, p2)
            alocate (newNode)
            [newNode].e ← e
            [newNode].left ← p2
            [newNode].right ← p1
            push (s, newNode)
        end-if
    end-for
    pop(s, p)
    tree.root ← p
end-subalgorithm
```

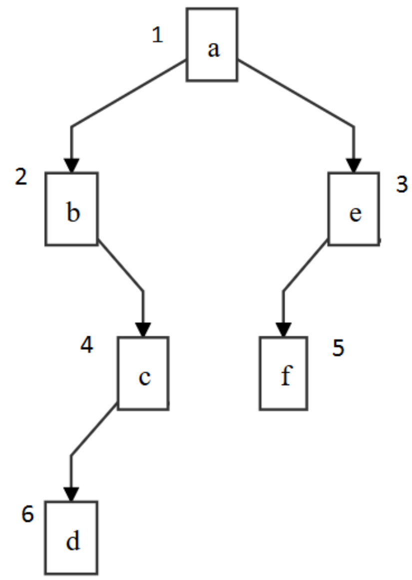
<u>If, instead of node, we want to use binary trees:</u> Stack will contain elements of type AB and we will call operations from AB's interface.

initLeaf(bt, e)
push(s, bt)

initTree(bt, p2, e, p1)
push(s, bt)

pop(s, tree)

2. Generate the table with information from a binary tree. Node numbering is done according to levels.



	1	2	3
	Info	Index Left	Index Right
1	a	2	3
2	b	0	4
3	e	5	0
4	c	6	0
5	f	0	0
6	d	0	0

- Divide the solution into 2 functions: *addNumbers* and *buildTable*
- We use a queue for storing the nodes (we need level-order traversal)
- Assume that each Node has a field *nr:Integer* (we are going to store the number of a node here).

```
Subalgorithm addNumbers (tree, k)
//pre: tree is a binary tree
//post: nr field from every node is set to the correct value, k is an integer
number, it represents the number of nodes from the tree.
    k ← 0
    init(q)
    if tree.root ≠ NIL then
        push(q, tree.root)
        k ← 1
        [tree.root].nr ← k
    end-if
    while (¬ isEmpty(q)) execute
        pop (q, p)
        if ([p].left ≠ NIL) then
            k ← k + 1
            [[p].left].nr ← k
            push(q, [p].left)
        end-if
        if ([p].right ≠ NIL) then
            k ← k + 1
            [[p].right].nr ← k
            push(q, [p].right)
        end-if
    end-while
end-subalgorithm
```

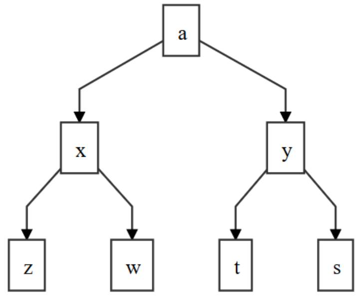
```
subalgorithm buildTable(p, T) is:
//pre: p is a pointer to a node, T is a matrix that holds the information
from the tree (column 1 node info, column 2 index of left, column 3 index of
right
    if (p ≠ NIL) then
        T[[p].nr, 1] ← [p].e
        if ([p].left ≠ NIL) then
            T[[p].nr, 2] ← [[p].left].nr
        else
            T[[p].nr, 2] ← 0
        end-if
        if ([p].right ≠ NIL) then
            T[[p].nr, 3] ← [[p].right].nr
        else
            T[[p].nr, 3] ← 0
        end-if
        buildTable([p].left, T)
        buildTable([p].right, T)
    end-if
end-subalgorithm

subalgorithm table(tree, T, k) is:
    addNumbers(tree, k)
    @define T as a matrix with k lines and 3 columns
    buildTable(tree.root, T)
end-subalgorithm
```

- How could I solve the problem if I do not want to add the *nr* field to the Node structure?
    - I can write one single function (addNumbers + buildTable together) and can add to the queue pairs of the form <node, nr>. When we pop a node from the queue we set the data for the corresponding row from T.

    - If I still want to use two function, the *addNumbers* can create a Map with elements <node, nr> or <tree, nr> and pass this map as a parameter to the *buildTable* function.

- What happens if I do not want recursive traversal for building the table?
    - Use a stack of a queue for the traversal.

3. Given a binary tree that represents the ancestors of a person up to the nth generation, where the left subtree represents the maternal line and the right subtree represents the paternal line:
    a. Display all the females from the tree (root can be either male of female)
        - a, x, z, t (assuming root is female)
    b. Display all ancestors of degree *k* (root has degree 0)
        - K = 2 – z, w, t, s



a. Traverse the tree using a queue (or stack) and print only the left subtrees

```
Subalgorithm females (tree) is:
    init(q)
    if tree.root ≠ NIL then
        push (q, tree.root)
        print [tree.root].e
    end-if
    while ¬isEmpty(q) execute
        pop(q, p)
        if ([p].left ≠ NIL) then
            print [[p].left].e
            push(q, [p].left)
        end-if
        if ([p].right ≠ NIL) then
            push(q, [p].right)
        end-if
    end-while
end-subalgorithm
```

b. Recursive version – using the tree's interface (we do not care/do not use the representation of the tree)

```
Subalgorithm level(tree, k, v) is
// v is a vector in which we will add the elements from level k, assume
it has an insert operation that adds a new element.
    if ¬isEmpty(tree) then
        if k = 0 then
            insert(v, root(tree))
        else
            if ¬ isEmpty(left (tree)) then
                level(left (tree), k-1, v)
            end-if
            if ¬ isEmpty(right (tree)) then
                level(left (tree), k-1, v)
            end-if
        end-if
    end-if
end-subalgorithm

subalgorithm ancestors (tree, k, v) is:
    init (v) // initialize an empty vector
    level (tree, k, v)
    for i ← 1, dim(v) execute
        print element(v, i)
    end-for
end-subalgorithm
```

- How can we solve the problem with a non-recursive function?
    - Put <node/tree, level> pairs in the queue
    - Or use two queues
    - Or count on the fact the tree should be complete (in real life you might have missing nodes/subtrees)