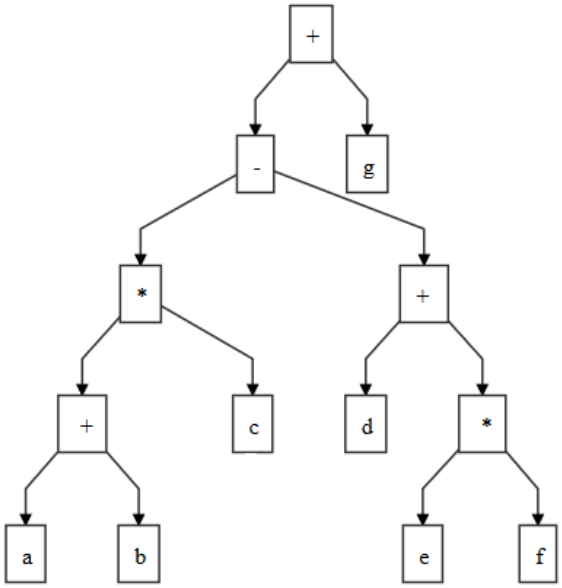
DSA - Seminar 7

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+

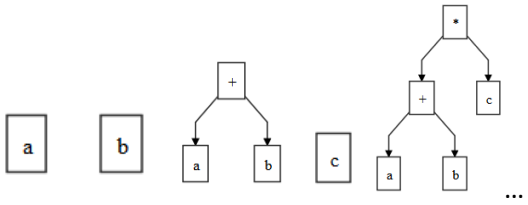
   The corresponding binary tree is:



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

   Algorithm (somehow similar to the evaluation of an arithmetic expression):
   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 the tree will be the last element from the stack.

   Stack:



...

Assume we have a binary tree with linked representation and dynamic allocation.

Node:
e: TElem
left, right: ↑Node

BT:
root: ↑Node

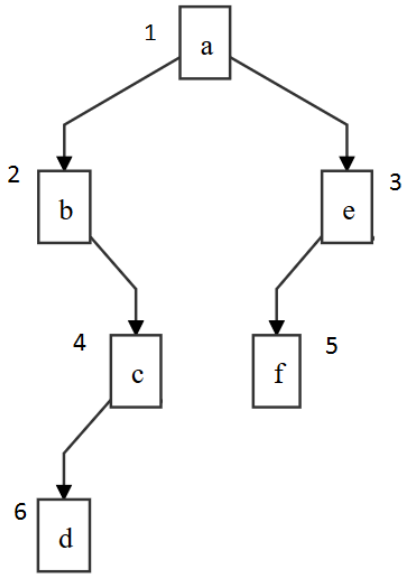
The stack will contain elements of type ↑Node and we will only use the interface of the stack

- Init
- Push
- Pop
- Top
- IsEmpty (will be needed for other problems)

```
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
            p1 ← pop(s)
            p2 ← pop(s)
            alocate (newNode)
            [newNode].e ← e
            [newNode].left ← p2
            [newNode].right ← p1
            push (s, newNode)
        end-if
    end-for
    p ← pop(s)
    tree.root ← p
end-subalgorithm
```

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

- The problem requires two things
    - Assign a number to every node (considering the levels)
    - Fill in the table with the information about the node, considering the assigned numbers



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

- Divide the solution into 2 functions: *addNumbers* (implemented non-recursively) and *buildTable* (implemented recursively)
- 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
        p ← pop (q)
        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
```

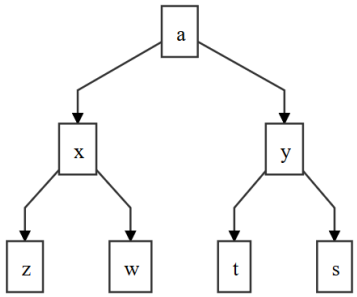
```
            push(q, [p].right)
        end-if
    end-while
end-subalgorithm

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] ← -1
        end-if
        if ([p].right ≠ NIL) then
            T[[p].nr, 3] ← [[p].right].nr
        else
            T[[p].nr, 3] ← -1
        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
```

- What happens if I do not want recursive traversal for building the table?
    - Use a stack or a queue for the traversal.
- What happens if I do not want a field for a *nr* in a node?
    - Have one single function which builds the table and use a map to store node – nr pairs. Use the level order traversal, before pushing a node to the queue add the <node, nr> pair in the map. When popping from queue, you can get the number of this node from the map, you number the children (add then to the map) and fill in the line corresponding to the node in the table (you know everything you need: info of node, number of node, numbers of children).

3. Given a binary tree that represents the ancestors of a person up to the n^{th} 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. Important to observe that when you have a node (popped from the queue, for example), you have no way of knowing whether it represents a male or a female. But you know for sure that its left child will be a female.

```
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
        p ← pop(q)
        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(node, 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 node ≠ NIL then
        if k = 0 then
            insert(v, node)
        else
            if [node].left ≠ NIL then
                level([node].left, k-1, v)
            end-if
            if [node].right ≠ NIL then
                level([node].right, 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.root, 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, level> pairs in the queue
  - Or use two queues, at every step you have nodes of one level in one queue, and push children of these nodes in the other one. When first is empty, swap them.
  - Or count on the fact the tree should be complete (in real life you might have missing nodes/subtrees) and count how many nodes you have to pass using a level order traversal to get to the nodes that are on level k.