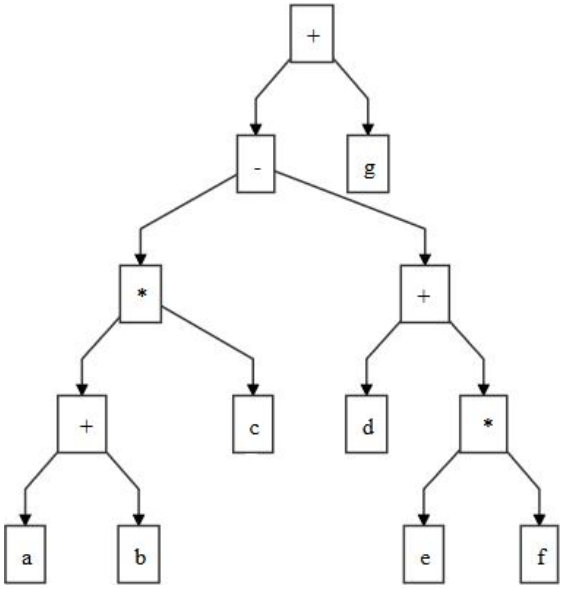
DS - Seminar 6

1. Build the binary tree for an arithmetic expression that contains the operators +, -, *, /. Use as input the postfix notation of the expression.

   Ex (in the infix form): (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 – right child
      b. Pop an element from the stack – left 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.

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

Node:
e: TElem
left, right: ↑Node

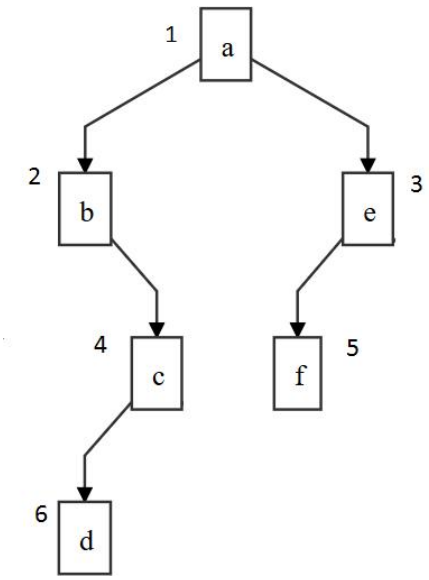
BT:
root: ↑Node

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 a table with information from a binary tree. Number the nodes ofthe tree 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. This table is actually a matrix with 3 columns and as many rows as the number of nodes in the tree. On a row *r* we will put data about the node whose assigned number is *r*.



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

- We will divide the solution in 2 functions:
  - *addNumbers*
    - Needs to do level order traversal of the nodes => we need to use a queue.
    - For storing the assigned numbers we will change the representation of the node, we will assume that besides *e, left* and *right* every nodes has a field *nr:Integer*, in which we can store these numbers.
  - *buildTable*
    - Assumes that the numbers were already assigned
    - Does not need to go through the tree in any specific order, we could use any traversal: with a queue, with a stack or recursive. We will do a recursive implementation

```
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
```

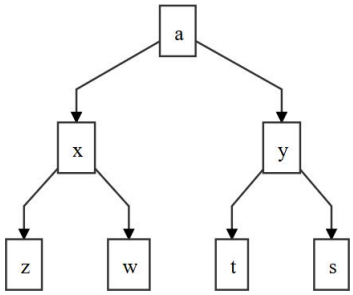
```
            k ← k + 1
            [[p].right].nr ← k
            push(q, [p].right)
        end-if
    end-while
end-subalgorithm

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 a field for a *nr* in a node?
    - If we do not want to split the implementation in two functions anymore, we can do a function similar to addNumbers, where we do a traversal with the queue. In this function we also have the table T as parameter. In the queue we can push <*node, nr*> pairs. When we pop from the queue, we pop a pair, a node with its number, *nr*. At this point we know the numbers that are going to be assigned to the children, so we can fill in row *nr* from T completely.
        - Disadvantage: we do not know how big T should be (we will only know the total number of nodes from the tree when the traversal is complete, but then it is too late to fill in T (we have lost the numbers when we popped from the queue).
    - If we want to keep the implementation in two functions, addNumbers need to find a way to store the numbers externally (i.e., not in the tree nodes). Since we need to store node – assigned number associations, the most natural way is to keep a map. When a node is *numbered*, we actually add a node – nr pair to the map. AddNumbers returns this map which is passed as parameter to T. When in the buildTable subalgorithm we need the number assigned to a node, we just do a search in the map.

3. We are 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 is 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 //assume root is female
    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

```
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.