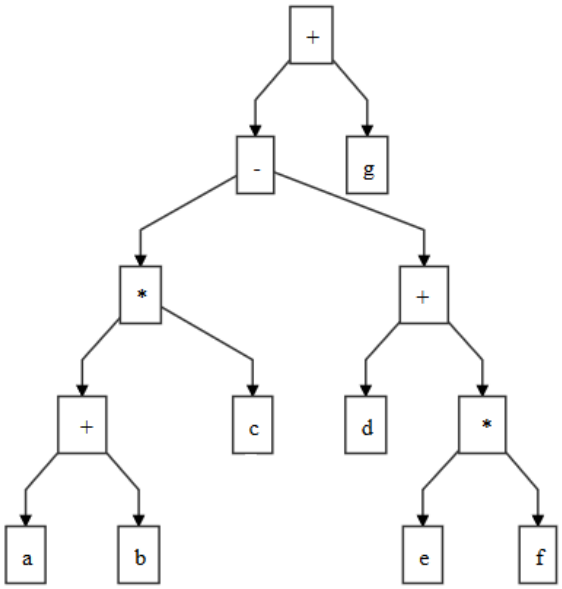
DSA - Seminar 7

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:

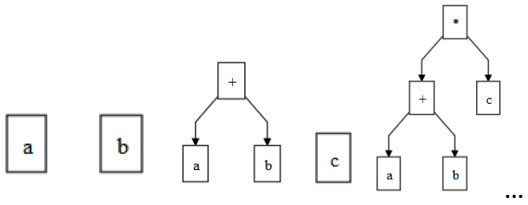


   If we traverse the tree in postorder, we will get the postfix notation.

   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 – lright eft 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.

   Stack:



...

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

Node:
e: TElem
left, right: ↑Node

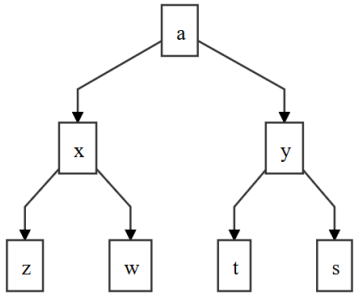
BT:
root: ↑Node

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. 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 (unless you push some special value together with the node)). 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(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 NOT isEmpty(tree) then
        if k = 0 then
            insert(v,root(tree))
        else
            if NOT isEmpty(left(tree)) then
                level(left(tree), k-1, v)
            end-if
```

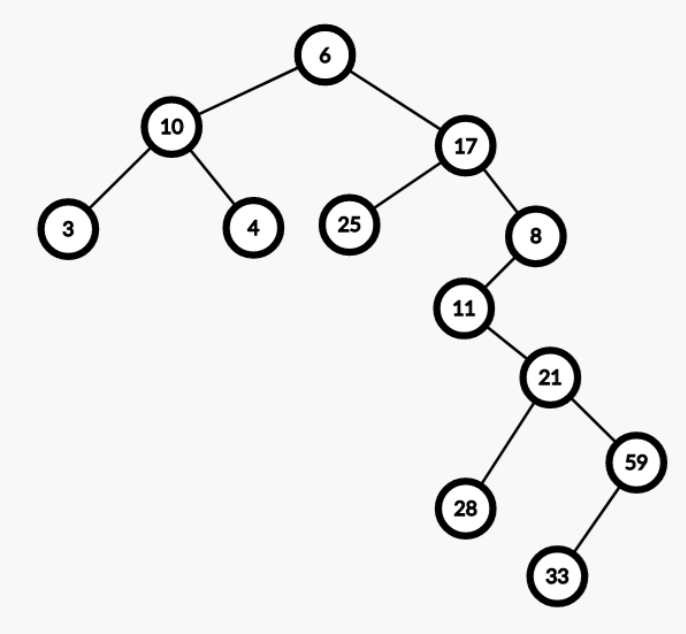
```
            if NOT isEmpty(right(tree)) then
                level(right(tree), k-1, v)
            end-if
        end-if
    end-if
end-subalgorithm

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
```

Obs. When working recursively, we need to have a stopping condition for our algorithm. Obviously, we stop when we have no more nodes, but this can be implemented in two ways:

- Assume that your recursive subalgorithm is never called for nodes whose value is NIL.
  - In this case, we can count on the fact that our parameter is never NIL, but, before doing any recursive calls, we need to check that the corresponding node is not NIL.
  - And we need to check in the main function, the one which starts the recursive call, as well, and not call the function if the tree is empty.
- Allow the recursive call to receive as parameter nodes whose value is NIL:
  - In this case the first thing that needs to be done in the function is to check if the parameter is NIL or not.
  - However, there are no restrictions when making the recursive calls on the children (or on the tree in the main function).

5. Given a binary tree return the *top view* of its nodes, in order from left-to-right. The top view of the nodes contains the nodes that are visible from the top of the tree. For example, for the tree below, the top view contains the following nodes (in this order from left to right): 3 10 6 17 8 59.



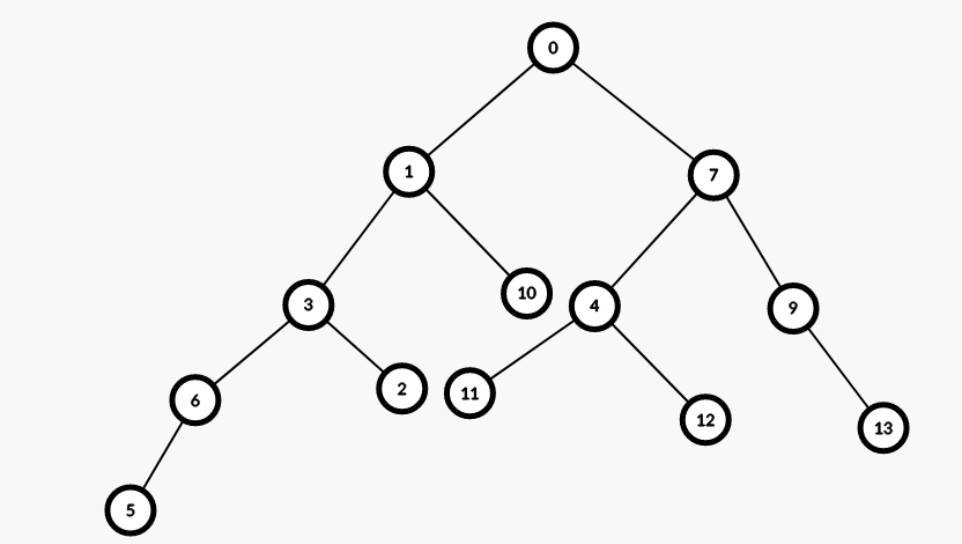
How to determine which nodes are part of the top view? We can see in the above example that the root and its children are part of the top view. However, from the grandchildren of the root, only 3 and 8 are part of the view, the other two (4 and 25) are *hidden* by the root. So which nodes are not hidden? The ones which are sufficiently to the left/right so that no other node hides them. How can we check this? We need to keep count of how much to the left or right of the root a node is. For any node which is at a "distance" *d* from the root, its left child will be at distance *d-1* while the right child will be at distance *d+1*. The root is at distance 0.

So, for example a few distances from the above tree:

6 0	4 0	8 2
10 -1	17 1	11 1
3 -2	25 0	Etc.

This suggests a traversal of the tree where we keep count of the distance of every node (we will have pairs in the stack/queue). Since we need the top view, probably a BFS would be better. For every possible distance, we need to print one single node, the first (starting from the root) found at that distance. But how do we know if a node is the first at a given distance? And how do we make sure to print them in the required order, which might not be the same as the order in which we find them? In order to make sure that we print them in the right order, we need to start printing only when all the tree was traversed. Up until that, we need to store the elements of the top view in a container, where we keep the distance as well. This will help us determine which node is the first as a given distance: if that distance is already in the container, it means that we have already found the first node. What should this container be? Since we need to store pairs, a Map seems like a good idea. But do we store node – distance or distance – node pairs? Since our main task will be ot check if a distance already exists, we need to store distance – node pairs.

Let's see how the above algorithm discovers the nodes from the top view for the following tree:



We start with node 0 and distance 0.

Then, since it is a breadth first search, we discover 1 with distance -1 and 7 with distance 1. Both of them go in the map.

Then, we visit 3 with distance -2, 10 with distance 0, 4 with distance 0 and 9 with distance 2. 3 and 9 go to the map.

Then we visit 6 (distance -3), 2 (distance -1), 11 (distance – 1), 12 (distance 1) and 13 (distance 3). 56 and 13 go to the map.

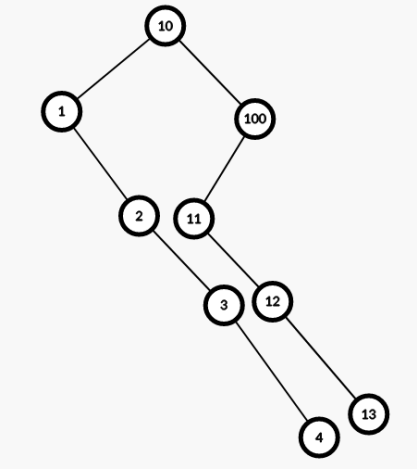
Then we visit 5 (distance -4) and add it to the map and we are done.
So for this tree, at every level excepting the root, we find two new nodes to be included in the top view, one which becomes the leftmost node (up until that moment) and one which becomes the rightmost node. While it might not be this case for every tree, we will always find at most two new nodes on a level, and we will always find either a new minimum (consecutive to the previous minimum) or a new maximum distance (consecutive to the previous maximum). So if we wanted to keep the nodes from the top view ordered by distance, something we do not have now, when we use a Map, we would at every step either add a new element at the beginning or at the end. This suggests that a deque could also be used instead of a Map to store the nodes from the top view. And since a deque can be used to simulate a queue or a stack, we could replace the queue from the breadth first traversal with a deque as well.

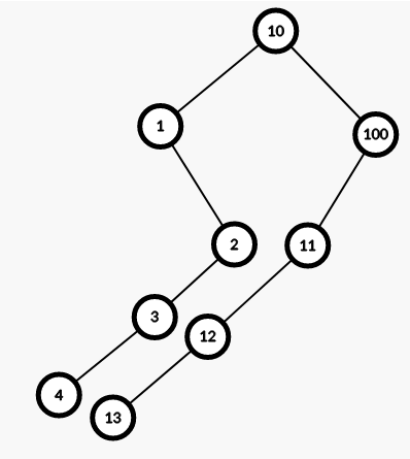
This gives us the following implementation:

```
subalgorithm topView_withDeque(tree):
      init(dq) //a deque with <node, distance> pairs (previously queue)
      init(topTree) // a deque with nodes. This is the top view
      if tree.root≠ NIL then
            push_back(dq, <tree.root, 0>)
            push_back(topTree, tree.root)
            minLeft ← 0
            maxRight ←0
      end-if
      while NOT isEmpty(dq))execute:
            <node, dist> ← pop_front(dq)
            if [node].left ≠ NIL then
                  cnr ← dist-1 // cnr - current number
                  push_back(dq, <[node].left, cnr>)
                  if cnr < minLeft then
                        push_front(topTree, [node].left)
                        minLeft ← cnr
                  end-if
            end-if
            if [node].right ≠ NIL then
                  cnr ← dist + 1 // cnr - current number
                  push_back(dq, [node].right, cnr)
                  if cnr > maxRight then
                        push_back(topTree, cnr)
                        maxRight ← cnr
                  end-if
            end-if
      end-while

      while NOT isEmpty(topTree) execute
            node ← pop_from(topTree)
            print ([node].info
      end-while
end-subalgorithm
```

Let's consider the following two trees:





For the first tree, the algorithm will provide the top view: 1 10 100 4
For the second tree, the algorithm will provide the top view: 4 1 10 100
Are these results correct?
The one for the right tree is correct, but for the tree from the left the last node of the top view should be 13, not 4.
When on the same level we find two nodes, both of which qualify as new minimum or maximum distance, we need to keep the left node if we found a new minimum and we need to keep the right node if we found a new maximum. In the current implementation we always keep the left node. So we need to change it, so that if we find a node whose distance is equal to the max distance (on right) and in the deque we already have a node with this level, we will overwrite the current node with the new one. But in order to test for this condition, we need to keep count of the current level as well, not just the current distance.

```
subalgorithm topView_withDeque(tree):

      init(dq) //deque with triples <nodes, current dist, current level>
      init(topTree)topTree // deque with <node level> pairs
      if tree.root ≠ NIL then
            push_back(dq, <tree.root, 0, 0>)
            push_back(topTree, <tree.root, 0>)
            minLeft ← 0
            maxRight ← 0
      end-if
      while NOT isEmpty(dq) execute:
            <pNode, dist, level> ← pop_front(dq)
            if [pNode].left ≠ NIL then
                  push_back(dq, <[pNode].left, dist-1, level+1>)
                  if dist-1 < minLeft then
                        push_front(topTree, <[pNode].left,dist-1>)
                        minLeft ← dist-1
                  end-if
            end-if
            if [pNode].right ≠ NIL then
                  push_back(dq, [pNode].right, dist+1, level+1>)
                  if dist+1 > maxRight then
                        push_back(topTree, <[pNode].right, level+1>)
                        maxRight ← dist +1
                  else  if dist + 1 = maxRight then
```

```
                        <lastNode, lastLevel> ← top_back(topTree)
                        if lastLevel = level+1 then
                            pop_back(topTree)
                            push_back(topTree, <[pNode].right, lastLevel>)
                        end-if
                end-if

            end-if
        end-while

        while NOT isEmpty(topTree) execute
            node ← pop_front(topTree)
            print ([node].info)
        end-while

end-subalgorithm
```

Complexity of the code: Θ(n).