Attendance

Creţa Florin	Ciorba Rares-Nicolaie
David Catalin Ioan	Cioroga Rares - Ioan
Cirtorosan Dragos	Demian Ana-Maria
Cosma Eduard	Carare Claudiu
Capra Paul Ovidiu	Custura Stefan - Octavian
Craiu Constantin-Tiberiu	Darida Razvan
Copîndean Alexandru	Chis Matei
Curila Sebastian	
Cimpean Andreea	Chis Sergiu
Deiac David-Mihai	
Comsa Filip-Emanuel	
Colta Paul Stefan	
Craciun Ioan Flaviu	
Caravia Andrei	Cheran Bianca Paula
Clapou Alexandru	
Comănac Dragoş-Mihail	Ciupe Sergiu-Calin

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
   => ab+c*def*+-g+

   Algorithm:
   - We need to use an auxiliary stack which contains pointers to nodes
   - We parse the postfix expression
     - if we find an operand (a, b ,c, etc.) => create a new node with this operand and push it to the stack
     - if we find an operator (+, -, etc.) => we pop two elements from the stack, we create a new node with the operator and the two popped nodes and push it to the stack
   - When the expression is over, the stack contains the root of the resulting tree.

Representation of the binary tree:

```
Node:
        info: TElem
        left, right: ↑Node

BT:
        root: ↑Node
```

We assume that the stack (queue) is already implemented and it has the following interface:

- init(st)
- push(st, e)
- pop(st) : returns and removes the element from the top of the stack
- top(st): returns the element from the top of the stack
- isEmpty(st): true/false

```
subalgorithm buildTree(postExpr, tree) is:
//tree is an output parameter
        init(st)
        for every e in postExpr execute:
                if e is an operand then
                        allocate(newNode)
                        [newNode].info <- e
                        push(st, newNode)
                else
                        allocate(newNode)
                        [newNode].info <- e
                        [newNode].right <- pop(st)
                        [newNode].left <- pop(st)
```

```
            push(st, newNode)
        end-if
    End-for
    Tree.root <- pop(st)
end-subalgorithm
```

2. Generate a table with information from a binary tree. Assign numbers to the nodes according to the levels.
    a. assign a number to every node and return the number of nodes - assignNumbers - iterative
    b. fill in the table with the information - fillInTable - recursive

- assume that the structure Node has a field/attribute *nr,* where we can store the number assigned to a node.

```
function assignNumbers(tree) is:
//returns the number of nodes as well
    init(q)

    if tree.root != NIL then

        push(q, tree.root)
    end-if
    currentNumber ← 1
    while isEmpty(q) = false execute
        currentNode ← pop(q)
        [currentNode].nr ← currentNumber
        currentNumber ← currentNumber + 1
        if [currentNode].left != NIL then
            push(q, [currentNode].left)
        end-if
        if currentNode].right != NIL then
            push(q, [currentNode].right)
        end-if

    end-while
    assignNumbers ← currentNumber - 1
end-function


subalgorithm fillinTable(node, T) is
//T is the table/matrix that we fill in. T[i, 1] - represents the information from a node, T[i, 2] is the index of the left child and T[i,3] - is he index of the right child
```

```
// can node be NIL? YES
    If node!=NIL then
        pos← [node].nr
        T[pos, 1]<--[node].info
        If [node].left!=NIL then
            T[pos, 2]<--[[node].left].nr
        Else
            T[pos,2]<-- -1
        End-if
        if  [node].right!=NIL then
            T[pos, 3]<--[[node].right].nr
        Else
            T[pos,3]<-- -1
        end-if
        fillinTable([node].left,T)
        fillinTable([node].right,T)
    end-if
end-subalgorithm

subalgorithm table(tree) is:
    nrNodes ← assignNumbers(tree)
    @create T as a matrix with nrNodes lines and 3 columns
    fillinTable(tree.root, T)
end-subalgorithm
```

What if?

- what if we want fillinTable to not be recursive?
  - use queue or a stack
- what if we want to have one single function?
  - add in the assignNumber function a part where you fill in the table (you either need access to the parents or need to assign numbers before pushing to the queue)
- what if Node does not have a field for nr (but we still have two functions)?
  - build a map in assignNumbers
    - <node/info, number>

3. We are given a binary tree which represents the ancestors of a person up to the nth generation. We know that the left subtree represents the maternal line and the right subtree represents the paternal line.
   a. display all the females from the tree
      i. recursively
         1. have a node as parameter and print the left child of the node (if exists), but have recursive calls for boh children

2. have a node as parameter and a boolean flag showing if this node is male or female. Print info if node is female and have two recursive calls with the flags set accordingly

ii. non-recursively
   1. use a stack or queue and when popping an element print its left child (if exists)
   2. if we can assume that the tree is a complete one then in a level order traversal we can print every second node (special case: the root)

b. display all ancestors of degree k (root is degree 0)
   i. recursively
      1. have a function with 3 parameters: node, k and currentLevel. When having recursive calls increment currentLevel. When currentLEvel = k print the info from the node
      2. have a function with 2 parameters: node and k. When having a recursive call decrement k. When k is 0 print the info from the node
   ii. non-recursively
      1. use a stack or a queue and push <node, level> pairs
      2. use two queues, one for currentLevel and one for the children of the nodes from the currentLevel
      3. use a queue and a special value to mark the end of a level
      4. if we know that the tree is complete we can compute how many nodes we need to pass in a level order traversal to get to level k.