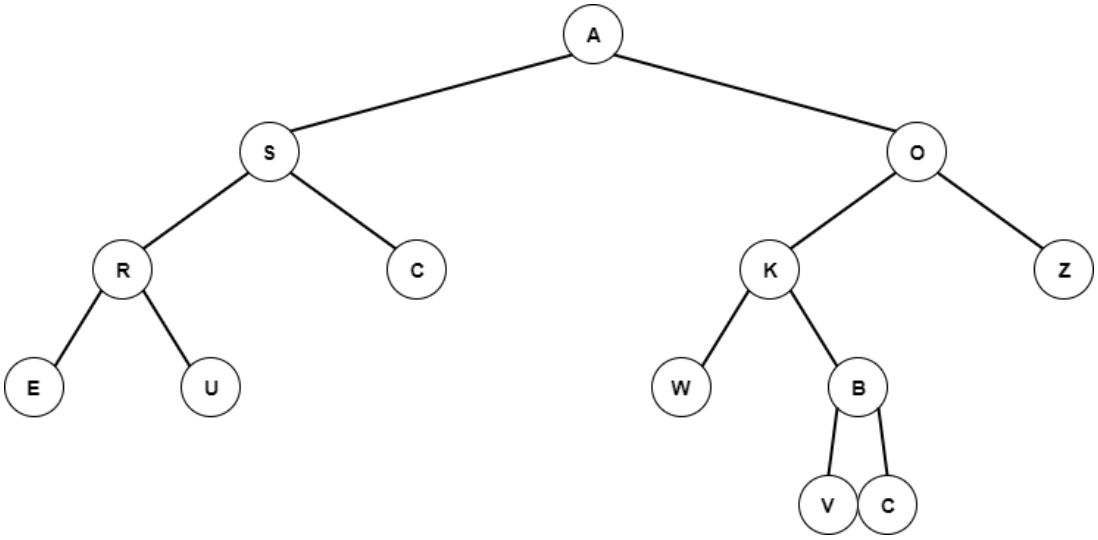
Tutorial (week 6)

All the answers can be given with pseudocode or with Python. Remember that most of the time, many solutions are possible.

1. For the following tree, give
   - the number of nodes
   - the root of the tree
   - the height of the tree
   - the depth of the node with letter W
   - the parent and descendants of the node with letter K
   - the sibling of the node with letter K
   - the external nodes
   - the internal nodes



Solution:

- the number of nodes: 13
- the root of the tree: the node with letter A
- the height of the tree: 4
- the depth of the node with letter W: 3
- the parent and descendants of the node with letter K: O - W, B, V and C
- the sibling of the node with letter K: Z
- the external nodes: E, U, C, W, V, C, Z
- the internal nodes: R, S, A, O, K, B

2. For the previous question's binary tree, give the ordered list of nodes visited

   - during a preorder traversal
   - during an inorder traversal
   - during a postorder traversal
   - during an Euler tour traversal

   Solution:

   - Preorder traversal: A, S, R, E, U, C, O, K, W, B, V, C, Z
   - Inorder traversal: E, R, U, S, C, A, W, K, V, B, C, O, Z
   - Postorder traversal: E, U, R, C, S, W, V, C, B, K, Z, O, A
   - Euler Tour traversal: A, S, R, E, E, E, R, U, U, U, R, S, C, C, C, S, A, O, K, W, W, W, K, B, V, V, V, B, C, C, C, B, K, O, Z, Z, Z, O, A

3. Let T be a proper binary tree such that all the external nodes have the same depth. Let D_e be the sum of the depths of all the external nodes of T, and let D_i be the sum of the depths of all the internal nodes of T. Find constants a and b such that

$$D_e + 1 = aD_i + bn$$

   where n is the number of nodes of T.

   Solution:
   If the binary tree is proper, then all internal nodes have exactly 2 children. Since all the external nodes have the same depth, it must be be that the tree is complete (that is all levels of the tree have the maximum number of nodes). Assume that the max depth of the tree is d, then there are 2^x nodes at each level $x \leq d$ of the tree.

   Trying with a small complete binary tree of depth 1, we have $D_e = 2$, $D_i = 0$ and $n = 3$. Then, trying with a small complete binary tree of depth 2, we have $D_e = 8$, $D_i = 2$ and $n = 7$. Using these two equation instances, we easily deduce that $a = b = 1$.

   More generally, we have:

$$D_e = d \cdot 2^d$$
$$D_i = \sum_{i=0}^{d-1} i \cdot 2^i$$
$$n = \sum_{i=0}^{d} 2^i$$

   and the formula can be proven by induction.

4. Design algorithms for the following operations for a node `v` in a proper binary tree `T`:

   - `preorderNext(v)`: return the node visited after `v` in a preorder traversal of `T`
   - `inorderNext(v)`: return the node visited after `v` in an inorder traversal of `T`

- `postorderNext(v)`: return the node visited after `v` in a postorder traversal of `T`.

What are the worst-case running times of your algorithms ?

Solution:
In a preorder traversal of a tree `T`, the root of `T` is visited first and then the subtrees rooted at its children are traversed recursively.

```
def preorderNext(v):
    '''
    INPUT: a node v in a proper binary tree
    OUTPUT: the node next to v in a preorder traversal of the tree
    '''
    # if v is an internal node, then the next node is simply its left child
    if v.isInternal():
        return v.left

   # if v is an external node, then we keep tracking back up in the tree,
   # as long as we are a right node. Once we reach a left node, then the next
   # node to v in a preorder traversal of the tree is its sibling
    while (v.parent).right == v:
        v = v.parent
        if v.parent == None: return None
    return (v.parent).right
```

The inorder traversal of a binary tree `T` can be informally viewed as visiting the nodes of `T` "from left to right". Indeed, for every node `v`, the inorder traversal visits `v` after all the nodes in the left subtree of `v` and before all the nodes in the right subtree of `v`.

```
def inorderNext(v):
    '''
    INPUT: a node v in a proper binary tree
    OUTPUT: the node next to v in a inorder traversal of the tree
    '''

    # if v is an external node, you must go back the first node you can
    # find for which you are not the right children. This is the next node
    if not v.isInternal():
        while (v.parent).right == v:
            v = v.parent
            if v.parent == None: return None # if we reach the root
        return v.parent

    # if v is an internal node, just go to your right subtree and
    # find the leftmost node in that subtree
    v = v.right
    while v.left != None:
        v = v.left
    return v
```

The postorder traversal method will visit a node `v` after it has visited all the other nodes in the subtree rooted at `v`.

```
def postorderNext(v):
    '''
    INPUT: a node v in a proper binary tree
    OUTPUT: the node next to v in a postorder traversal of the tree
    '''

    # if v is the root, it is the last node to be visited
    if v.parent == None: return None

    # if v is a right node, then its parent is necessarily the next node
    if (v.parent).right == v:
        return v.parent

    # otherwise, we jump to v's sibling and find the leftmost node in that subtree
    v = (v.parent).right
    while v.left != None:
        v = v.left
    return v
```

During the traversal, in the worst case you might have to go from an external node back to the root, and then again to an external node. Thus, basically you won't be jumping for more than twice the height of the tree. The worst case running times for these algorithms are then all $O(h)$ where h is the height of the tree `T`.

5. Give an $O(n)$-time algorithm that computes and prints the depth of all the nodes of a tree `T`, where n is the number of nodes of `T`.

   Solution:
   This can be done by simply performing a preorder (or postorder) traversal of the tree, where the "visit" action is to report the depth of the node that is currently visited (you can keep track of the current depth using a counter).

   You can use the following pseudocode, which can be started by calling `computeDepth(T.root,0)`:

```
def computeDepth(v,current_depth):
    '''
    INPUT: a node v from a proper binary tree T
    OUTPUT: prints the depth of v and of all the nodes in his subtree
    '''
    print("node " + str(v) + "is at depth \n" % current_depth)
    if v.left != None:  computeDepth(v.left,current_depth+1)
    if v.right != None:  computeDepth(v.right,current_depth+1)
```

6. Describe in Python or pseudocode a nonrecursive method for performing an Euler tour traversal of a proper binary tree that runs in linear time and does not use a stack.

   Solution:

```
def iterative_euler(v):
```

```
    '''
    INPUT: a node v that is root from a tree T
    OUTPUT: an Euler tour traversal of T
    '''

    print(v)  # print the root node once


    # continue until we are finished priting the entire Euler tour traversal
    while True:

        # keep going through left nodes and print each node encountered
        while v.left != None:
            v = v.left
            print(v)

        # when you reach an external node, you have to print twice more
        # in order to take into account left/bottom/right actions
        print(v)
        print(v)

        # as long as you are coming from a right node, you can continue backtracking
        # and printing
        while (v.parent).right == v:
            v = v.parent
            print(v)
            if v.parent == None: return   # if we are back to the root from the right
                , we are done


        # go back to the parent and go visit the next subtree on the right
        v = v.parent
        print(v)
        v = v.right
        print(v)
```

7. Describe in Python or pseudocode a nonrecursive method for performing an inorder traversal of a proper binary tree in linear time.

Solution:

```
def inorder_nonrecursive(v):
    '''
    INPUT: a node v, root of a proper binary tree T
    OUTPUT: prints the nodes of T in a inorder traversal
    '''

    s = Stack() # create an empty stack

    while True:  # repeat the process until we are done

        # continue pushing the successive left nodes to the stack
        while v != None:
            s.push(v)
            v = v.left

        # otherwise we go back to the two nodes in the stack and we explore the right
            node
        v = s.pop()
```

```
        print(v.data)
        if s.isEmpty():
            return
        v = s.pop()
        print(v.data)
        v = v.right
```

8. The *path length* of a tree T is the sum of the depths of all the nodes in `T`. Describe a linear-time method for computing the path length of a tree `T` (which is not necessarily binary).

Solution:

```
def explore_pathLength(v,current_depth):

    # if we reach an external node, we just return its depth
    if v.children() == None: return current_depth

    # we sum the path lengths of all the children and we add it to the node's depth
    current_pathLength = current_depth
    for i in v.children():
        current_pathLength += explore_pathLength(i,current_depth+1)

    return current_pathLength


def pathLength(v):
    '''
    INPUT: a node v from a tree T
    OUTPUT: the pathlength of T
    '''

    return explore_pathLength(v,0)
```

9. Define the *internal path length*, $I(T)$, of a tree `T` to be the sum of the depths of all the internal nodes in `T`. Likewise, define the external path length, $E(T)$, of a tree `T` to be the sum of the depths of all the external nodes in `T`. Show that if `T` is a binary tree with $n(T)$ internal nodes, then $E(T) = I(T)+2n(T)$.

Solution:
You can prove it by induction. First, the simple tree T_0 with a single root node indeed verifies the formula (as a single root node is an external node). Now, any proper binary tree can be built from T_0 by successively replacing external nodes into an internal node with two leaf children.

From a proper binary tree `T`, assume that we expand an external node that had depth d with two children at depth $d+1$ to obtain a new tree `T'`. Then, from `T` to `T'`, such a transformation has the following effects:

- a internal node at depth d is gained, thus $I(T') = I(T) + d$ and $n(T') = n(T) + 1$
- an external node at depth d is lost, but two external nodes at depth $d+1$ are gained, thus $E(T') = E(T) - d + 2(d+1) = E(T) + d + 2$

Assuming that the formula is verified for `T`, we can easily show that it is also the case for `T'`:

$$E(T') = E(T) + d + 2 = I(T) + 2n(T) + d + 2 = I(T') + 2n(T')$$

Hints

- **Question 3:** try to gain some intuition by drawing a few different proper binary trees such that all the external nodes have the same depth.
- **Question 4:** think about what could be the worst case number of nodes that would have to be traversed to answer each of these queries.
- **Question 6:** you can tell which visit action to perform at a node by taking note of where you are coming from.
- **Question 7:** use a stack.
- **Question 8:** modify an algorithm for computing the depth of each node so that it computes path lengths at the same time.
- **Question 9:** use the fact that we can build `T` from a single root node via a series of $n(T)$ operations that expand an external node into an internal node with two leaf children.