Algorithmics
Correction Midterm #2 (C2)

UNDERGRADUATE 1^{st} YEAR S2 – EPITA

2 March 2020 - 10 : 00

Solution 1 **(A little coursework... –** ***4 points*****)**
The general tree **T** being as follows :

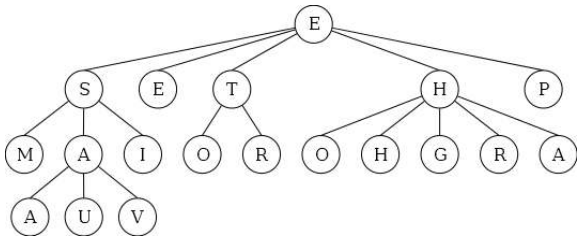

Figure 1: The general tree **T**

1. The size of thee tree `T` is: `19`
2. The height of thee tree `T` is: `3`
3. The internal path length of the tree `T` is: `5`
4. The external average depth of the tree `T` is: `29/14` $\simeq$ `2.07`
5. The list of vertices of the tree `T` encountered in postorder traversal is :
   `{M, A, U, V, A, I, S, E, O, R, T, O, H, G, R, A, H, P, E}`
6. The list of nodes of the tree `T` encountered in level order is:
   `{E, S, E, T, H, P, M, A, I, O, R, O, H, G, R, A, A, U, V}`

Solution 2 **(Magic Square –** ***4 points*****)**

Specifications:

The function `Siamese(`n`)` builds and returns a magic square of order n (n is an odd integer greater than 2).

```
        def Siamese(n):
            """
            n natural, odd
            """
            S = matrix.init(n, n, 0)
            (i, j) = (n - 1, n // 2)
            for val in range(1, n*n + 1):
                S[i][j] = val
                if val % n == 0:
                    i = i - 1
                    if i == -1:
                        i = n-1
                else:
                    (i, j) = ((i + 1) % n, (j + 1) % n)
            return S
```

Solution 3 **(Sub-List – *5 points*)**

Specifications:

The function `sub_line(M, L)` checks if the list L is included in one of the lines of the matrix M (assumed non empty).

```
1   def sub_line(M, L):
2       (lineM, colM) = (len(M), len(M[0]))
3       n = len(L)
4       if n > colM:
5           return False
6       else:
7           i = 0
8           ok = False
9           while i < lineM and not ok:
10              j = 0
11              while j < colM - n and not ok:
12                  k = 0
13                  while k < n and L[k] == M[i][j+k]:
14                      k += 1
15                  ok = (k == n)
16                  j += 1
17              i += 1
18          return ok
19
20  # two functions
21
22  def equalList(LM, L, start) :
23      (i, n) = (0, len(L))
24      while i < n and LM[start+i] == L[i] :
25          i += 1
26      return i == n
27
28  def sub_line2(M, L) :
29      (lb, cb, n) = (len(M), len(M[0]), len(L))
30      if n > cb :
31          return False
32      else:
33          i = 0
34          j = (cb - n) + 1
35          while i < lb and j > (cb-n) :
36              j = 0
37              while j <= (cb-n) and not equalList(M[i], L, j):
38                  j += 1
39              if j > (cb-n) :
40                  i += 1
41          return i < lb
```

***Solution 4* (Partially ordered tree – *3 points*)**

Specifications:

The function `priority(B)` checks if the binary tree B (whose keys are non zero naturals) is partially ordered.

```
1  def __test(B, p):
2      """
3      p: B's parent
4      """
5      if B == None:
6          return True
7      else:
8          if B.key < p:
9              return False
10         else:
11             return __test(B.left, B.key) and __test(B.right, B.key)
12
13 def priority(B):
14     return __test(B, 0)
15
16
17
18 def priority2(B, p=0):
19     """
20     p: B's parent
21     """
22     if B == None:
23         return True
24     else:
25         if B.key < p:
26             return False
27         else:
28             return priority2(B.left, B.key) and priority2(B.right, B.key)
```

Version without the parent in parameter:

```
1  def __priority3(B):
2      """
3      B not empty
4      """
5      test = True
6      if B.left != None:
7          if B.key > B.left.key:
8              test = False
9          else:
10             test = __priority3(B.left)
11     if test and B.right != None:
12         if B.key > B.right.key:
13             test = False
14         else:
15             test = __priority3(B.right)
16     return test
17
18 def priority3(B):
19     return B == None or __priority3(B)
```

***Solution 5* (Width – *4 points*)**

Specifications:

The function `width(B)` calculates the width of the binary tree B.

```
# with level change marks (None)

def width(B):
    w_max = 0
    if B:
        q = queue.Queue()
        q.enqueue(B)
        q.enqueue(None)
        w = 0
        while not q.isempty():
            B = q.dequeue()
            if B == None:
                w_max = max(w, w_max)
                if not q.isempty():
                    q.enqueue(None)
                    w = 0
            else:
                w = w + 1
                if B.left:
                    q.enqueue(B.left)
                if B.right:
                    q.enqueue(B.right)
    return w_max


# another way to manage levels, with two queues.

def width2(B):
    w_max = 0
    if B != None:
        q = queue.Queue() #current
        q.enqueue(B)
        q_next = queue.Queue() #next level
        w = 0
        while not q.isempty():
            B = q.dequeue()
            w = w + 1
            if B.left != None:
                q_next.enqueue(B.left)
            if B.right != None:
                q_next.enqueue(B.right)
            if q.isempty():
                w_max = max(w, w_max)
                (q, q_next) = (q_next, q)
                w = 0
    return w_max
```