Binary Trees (Arbres binaires)

1 Measures

Exercise 1.1 (Size (Taille))

1. Using the abstract data type `binary tree`, give the axioms that define the *size* operation.
2. Write a function that calculates the size of a binary tree.

Exercise 1.2 (Height (Hauteur))

1. Using the abstract data type `binary tree`, give the axioms that define the *height* operation.
2. Write a function that calculates the height of a binary tree.

Exercise 1.3 (Path Lengths and Average Depths)

1. Consider the following functions:

```
function frec(binarytree B, integer h,  ref integer n) : integer
begin
    if B = emptytree then
        return 0
    else
        n ← n + 1
        return h + frec(l(B), h+1, n) + frec(r(B), h+1, n)
    end if
end
```

```
function fun(binarytree B) : real
variables
   integer  nb
begin
    if B = emptytree then
        /* Exception */
    else
        nb ← 0
        return (frec (B, 0, nb) / nb)
    end if
end
```

   (a) What is the result of the application of the function `fun` to the tree in figure 1?

   (b) What mesure does this function calculate?

2. Write a function that calculates the external path length of a binary tree.
3. What has to be modified in this function to calculate the external average depth?

2 Traversals

Exercise 2.1 (Depth-first Traversal (Parcours profondeur))

1. Give the three orders induced by the depth-first traversal (beginning on the left side) of the tree in figure 1.

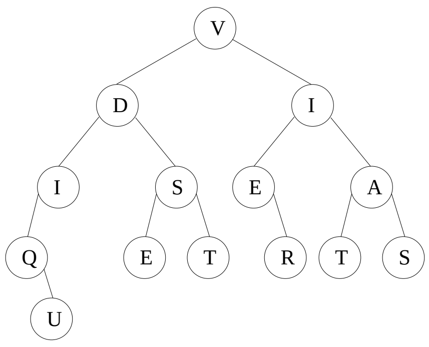

Figure 1: Binary tree for traversals

2. Give the $<r, G, D>$ form of the tree in figure 1 with _ to represent the empty tree.
3. Write a function that displays the $<r, G, D>$ form of a binary tree with _ to represent the empty tree.

Exercise 2.2 (Breadth-first Traversal (Parcours largeur))

1. Run through the breadth-first traversal algorithm on the tree in figure 1.
2. Write a function that calculates the width of a binary tree.

3 Applications

Exercise 3.1 (Search for the path)

1. Write a function that searches for a value x in a binary tree. It returns the tree whose root contains x if found, the value `None` otherwise.
2. The previous function does not give the path to x from the root. Write another version of the search that returns this path.

Another solution?

Exercise 3.2 (Parent and children)

Below, the class `BinTreeParent` permits to represent binary trees where each node has links to its children but also a link to its parent.

```
class BinTreeParent:
    def __init__(self, key, parent, left, right):
        self.key = key
        self.parent = parent
        self.left = left
        self.right = right
```

Write a function that builds from a "classic" binary tree (`BinTree`) the equivalent tree (with same values at same places) with in each node a link to its parent (`BinTreeParent`).

Exercise 3.3 (Degenerate, complete, perfect)

1. **Degenerate tree (*dégénéré*):**
   (a) What is a degenerate tree?
   (b) How to verify if a tree is degenerate knowing its size and height?
   (c) Write an algorithm that tests if a tree is degenerate without using `size` neither `height`.
2. **Perfect tree (*complet*) :**
   (a) Give several definitions of a perfect tree.
   (b) How to verify if a tree is perfect knowing its size and height?
   (c) Write a function that tests if a tree is complete (you can use the function `height`, only once).
   (d) Write again the test function without using `height`.
3. **Complete tree (*parfait*) :**
   (a) What is a complete tree?
   (b) How to modify the previous functions, that test if a tree is perfect, to test if a tree is complete?

Exercise 3.4 (Tree serialization)

Here is the list of values encountered in prefix order in the depth-first traversal of the tree from previous tutorial:

```
L_tutoPref = ['V', 'D', 'I', 'Q', 'U', 'S', 'E', 'T', 'I', 'E', 'R',
              'A', 'T', 'S']
```

It is not possible to reconstruct the tree form this single list: there are several trees that give this preorder.

Howerver, it is possible with this list:

```
L_tuto = ['V', 'D', 'I', 'Q', '#', 'U', '#', '#', '#', 'S', 'E', '#', '#',
          'T', '#', '#', 'I', 'E', '#', 'R', '#', '#', 'A', 'T', '#', '#',
          'S', '#', '#']
```

It is called "serialize" a tree. (Preorder traversal serialization of a binary tree.)

1. Write a fonction that serializes a tree. It returns a list as the one presented above.
2. Write a function that reconstructs the tree form its serialization.

Bonus : Given a list, verify whether it is a correct serialization of a binary tree without reconstructing the tree.

4 Expressions and trees - midTerm 2016

An expression can be represented by a tree: internal nodes contain the operators and external ones contain the operands. Here we work with arithmetical expressions that use the binary operators +, -, * and /.

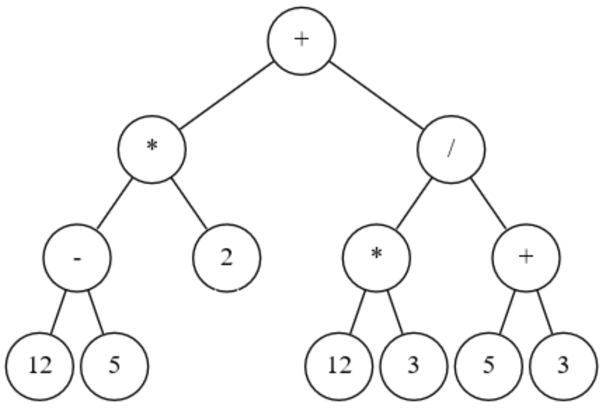

Figure 2: Tree for the expression $(12 - 5) * 2 + (12 * 3)/(5 + 3)$

Exercise 4.1 (Draw me)

Let B_1, B_2, B_3 and B_4 be binary trees representing expressions.

During the depth-first traversal:

- values of B_1 in preorder give the list: + - / 8 2 5 + 4 * 2 3
- values of B_2 in postorder give the list: 8 20 - 2 2 * 2 + /
- the traversals of B_3 and B_4 give the same inorder list: 8 + 7 / 5 - 4 * 2
- B_3 and B_4 are different but the two expressions they represent have the same result : -5.

Draw the trees B_1, B_2, B_3 and B_4. Give the values of the expressions from B_1 and B_2.

Exercise 4.2 (Display me)

Write the function `exp2str` that returns a string of the fully parenthesized expression represented by a tree.

Application example with the tree in figure 2:

```
>>> exp2str(B)
'(((12-5)*2)+((12*3)/(5+3)))'
```

Exercise 4.3 (Count me)

Write the function `nodes` that count the number of operators and operands of an expression represented by a binary tree.

Application example with the tree in figure 2:

```
>>> nodes(B)
(6, 7)
```

Binary Trees (Arbres binaires) Occurrences - Hierarchical numbering

5 Occurrences

Exercise 5.1 (Occurrence display)

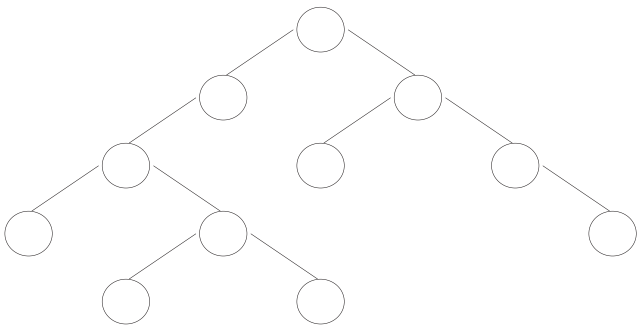

Figure 3: Binary tree for occurrences

1. Give the occurrence list representation of the tree in figure 3.
2. Write the function that displays the occurrence representation of a binary tree.

Exercise 5.2 (Binary tree and prefix code – *Final 2015*)

Compressing text files can be done by encoding characters with binary words. We use here a variable-length code: each character can be encoded using a different number of bits. Obviously we want frequent characters to use short codes, while long codes should be reserved for infrequent characters.
Also our encoding must have the prefix property: no code should be prefix of another code. If we do not respect this property, for instance encoding 'a' with 11, and 'b' with 111, then we cannot tell if 11111 encodes "ab" or "ba".

1. Consider the following encoding:

letter	a	b	c	d	e	f
code	0	101	100	111	1101	1100

   Decode 11011100110001001101.

2. The encoding is represented by a tree whose leaves are the letters to encode (the field `key` contains the letter). The code of a letter can be deduced from the path from the root to the leaf that contains this letter: each left branch represents a 0, and each right branch is a 1.
   What does the code of a letter correspond to?
3. Draw the tree representing the encoding of question 1.
4. The tree representing the encoding is a full (proper) tree in which each leaf is used. Write a function that displays the code of a given letter if it exists in the tree (type is given in appendix). For instance, with the encoding of question 1, if the letter is 'e', the function displays "1101".

   Two possible versions for this function:

   - The "simple" one, but not so optimal: it traverses the whole tree.
   - The more optimal (but...): it stops as soon as possible. If this version is chosen, it will also have to display "no code found" if the letter is not found in the tree.

   In both cases, you have to write a recursive function and the one that calls it.

   Choose the version that suits you knowing that it is obviously the second version that will bring the most points.[1]

[1]Sometimes a few points are better than none.

6 Hierarchical Numbering

Exercise 6.1 (Occurrence ↔ Hierarchical number)

1. Write a function that calculates the hierarchical number of a node given its occurrence (a string).
2. Write the function that builds the occurrence of a node from its hierarchical number.

Exercise 6.2 (Hierarchical implementation)

Reminder:

A simple array (a list in Python) can be used to represent a binary tree. The hierarchical numbering of a node is the position of its value in the vector. All "unused" cells have here the value $\emptyset$ (`None` in Python).

1. Give the array that represents the tree in figure 4.
2. What has to be modified in the traversals (both depth and width) when the tree is given as a list ("hierarchical" representation)?

Exercise 6.3 (Hierarchical implementation test – *Final Test 2014*)

Write a function that tests whether the two trees in parameters, one in classic implementation ("object") the other in list one, are identical.

Exercise 6.4 (Object ↔ List)

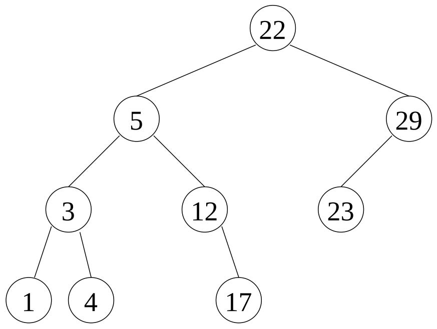

Figure 4: Tree for hierarchical implementation

1. Write a function that builds the list implementation using hierarchical numbering of a tree ("classic" implementation). The value `None` will be used to reference an empty tree.

   The representation is built here for any tree. What can be modified if the tree is complete?

2. Write a function that builds the "object" representation of a tree from its hierarchical form.