General Trees (Arbres généraux)

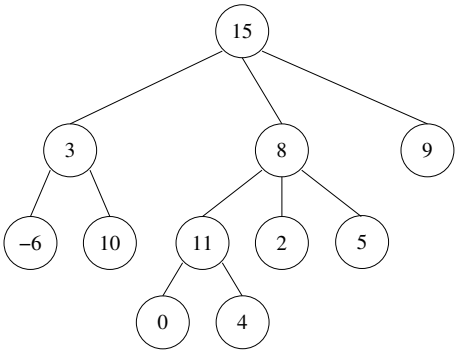

Figure 1: General Tree T_1

1 Measures

Exercise 1.1 (Size (Taille))

1. Give the definition of the size of a tree.
2. Write a function that returns the size of a tree, for both implementations:
   (a) by tuples (each node contains a tuples of children);
   (b) left child - right sibling (as a binary tree).

Exercise 1.2 (Height (Hauteur))

1. Give the definition of the height of a tree.
2. Write a function that returns the height of a tree, for both implementations:
   (a) by tuples;
   (b) left child - right sibling.

Exercise 1.3 (External Average Depth (Profondeur moyenne externe))

1. Give the definition of external average depth of a tree.
2. Write a function that returns the external average depth of a tree, for both implementations:
   (a) by tuples;
   (b) left child - right sibling.

2 Traversals

Exercise 2.1 (DFS: Depth First Search (Parcours en profondeur))

1. Give the principle of a depth-first search for a general tree.
2. List elements in prefix and suffix orders for the depth-first search of the tree in figure 1. What other action can be done when visiting a node ?
3. Write a template depth-first search algorithm (insert node actions as comments) for both implementations:
   (a) by tuples;
   (b) left child - right sibling.

Exercise 2.2 (BFS: Breadth First Search (Parcours en largeur))

1. Give the principle of a breadth-first search for a tree.
2. How can we detect level changes during the traversal?
3. Write a function that computes the width of a tree, for both implementations:
   (a) by tuples;
   (b) left child - right sibling.

3 Applications

Exercise 3.1 (Prefix - Suffix – *C3 - 2017*)

The aim here is to fill a vector with the keys of a general tree. Each key is put **twice** in the vector: at the first encounter (prefix / preorder) and at the last encounter (suffix / postorder) during the depth first search.

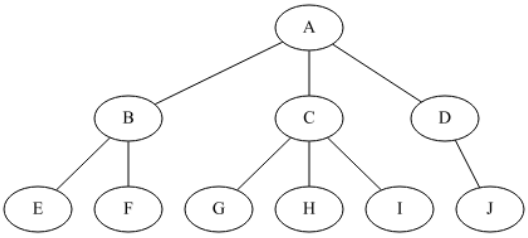

Figure 2: General Tree T

1. Give the arry after the depth first search of the tree in figure 2.
2. Write the function `prefsuff(T)` that builds and returns from the tree `T` the corresponding key vector (represented as a list in Python) according to the described order, for both implementations:
   (a) by tuples; (b) left child - right sibling.

Exercise 3.2 (*List* Representation)

Let A be the general tree $A =< o, A_1, A_2, ..., A_N >$. The following linear representation of A (o A_1 A_2 ... A_N) is called *list*.

1. (a) Give the linear representation of the tree in figure 1.
   (b) Draw the tree corresponding to the *list* `(12(2(25)(6)(-7))(0(18(1)(8))(9))(4(3)(11)))`.
2. Write the function that builds the linear representation (as a string) from a tree, for both implementations:
   (a) by tuples; (b) left child - right sibling.
   What has to be change to obtain an "abstract type" like representation ($A =< o, A_1, A_2, ..., A_N >$)?

Exercise 3.3 (Average Arity of a General Tree – *C3# - 2018*)

We now study the average arity (number of children for a node) in a general tree. We define the average arity as the sum of the number of children per node divided by the number of internal nodes.

For example, in the tree from figure 3, there's 8 internal nodes and the sum of the number of children per node is 17 (check the arrows), thus the average arity is: $17/8 = 2,125$.

Write the function `averageArity(`B`)` that returns the average arity of the a general tree T, for *first child - right sibling* implementation.

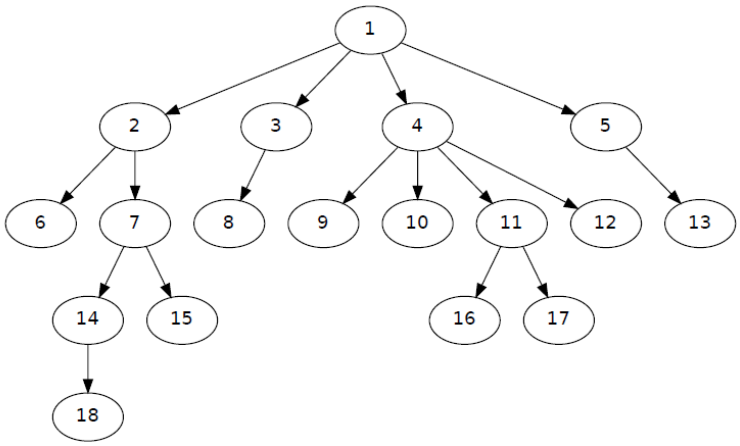

Figure 3: General tree

Exercise 3.4 (`dot` format)

A tree can be represented as a list of links (like a graph): the *dot* format.

```
graph {
    15 -- 3;
    15 -- 8;
    15 -- 9;
    3 -- -6;
    3 -- 10;
    8 -- 11;
    8 -- 2;
    8 -- 5;
    11 -- 0;
    11 -- 4;
}
```

Another possibility:

```
graph {
    15 -- {3; 8; 9};
    3 -- {-6; 10};
    8 -- {11; 2; 5};
    11 -- {0; 4};
}
```

';' can be omitted.

If you want to see the graphical representation of your tree, use "Graphviz".
Warning: according to the order of links, the result will not be the same. **The order given here is the appropriate one for a tree.**
Write the functions that allow to:

- build the `.dot` file from a tree (for both implementations)
- and in return, build a tree (in both implementations) from a `.dot` file.

Exercise 3.5 (Tuples ↔ left child - right sibling)

1. Write a function that builds, from a general tree with left child - right sibling implementation (*i.e.* a binary tree), its "by tuples" implementation.
2. Write the translation function for the other way.

Exercise 3.6 (Load trees from files)

To store the trees in text files (`.tree`) we use the representation by *lists* seen exercise 3.2 ($A =< o, A_1, A_2, ..., A_N >$ is represented by (o A_1 A_2 ... A_N).

1. Write the function that builds the tree from the *list* (type `str`), in both implementations.
2. Write a function that loads a tree from a `.tree` file.