Binary Trees – Python Implementations

Nathalie *Junior* Bouquet

February 2018

Definition:

A *binary tree* is either an empty tree, or $B = <o, L, R>$, where o is the *root* node and L and R are two distinct subtrees.

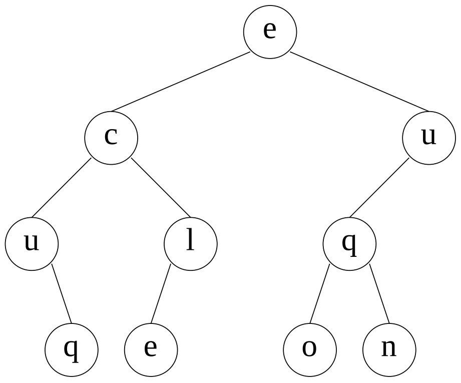


Figure 1: Binary tree

1 Abstract Data Type

TYPES

binarytree

USES

node, element

OPERATIONS

emptytree	:	$\rightarrow$ binarytree
$<\_, \_, \_>$	:	node $\times$ binarytree $\times$ binarytree $\rightarrow$ binarytree
root	:	binarytree $\rightarrow$ node
l	:	binarytree $\rightarrow$ binarytree
r	:	binarytree $\rightarrow$ binarytree
content	:	node $\rightarrow$ element

PRECONDITIONS

$root(B)$ **is-defined-iaoi** $B \neq emptytree$

$l(B)$ **is-defined-iaoi** $B \neq emptytree$

$r(B)$ **is-defined-iaoi** $B \neq emptytree$

AXIOMS

$root(<o, L, R>) = o$

$l(<o, L, R>) = L$

$r(<o, L, R>) = R$

WITH

L, R	:	binarytree
o	:	node

The operation *content* allows to associate an element to each node. Binary trees in which nodes contain elements are called *labeled trees*.

We often called $<r, L, R>$ the tree whose root contains the element r.

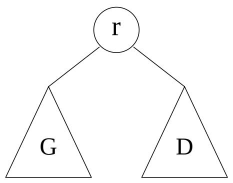


Figure 2: Binary tree: a recursive structure

2 Implementations

2.1 Python Class

The most natural representation reproduces the recursive structure (see figure 2).

Each node is represented by an object (that simulates the structure of the dynamic implementation) that contains the links to the left and right subtrees (`left` and `right` "fields").
A non-empty tree is represented by the reference on the root node. The empty tree is represented by the `None` value.

Mostly, we use labeled trees. An additional field `key` represents the information contained in the node.

2.1.1 The Class definition

```python
class BinTree:
    def __init__(self, key, left, right):
        """
        Init Tree
        """
        self.key = key
        self.left = left
        self.right = right
```

Example of a non empty tree, with `B1` and `B2` two trees (empty or not):

```python
B = BinTree(42, B1, B2)
```

Similarity with the dynamic implementation seen in lecture:

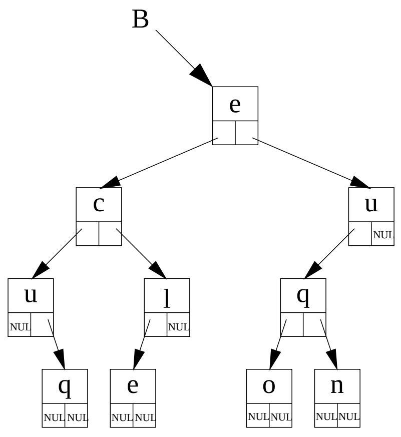


Figure 3: Dynamic representation of tree from figure 1

2.1.2 Operation implementations

Abstract type: binarytree	**Implémentation**
B : binarytree	`B`
emptytree	`None`
content(root(B))	`B.key`
l(B)	`B.left`
r(B)	`B.right`

2.2 "Static" implementation: hierachical numbering

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.

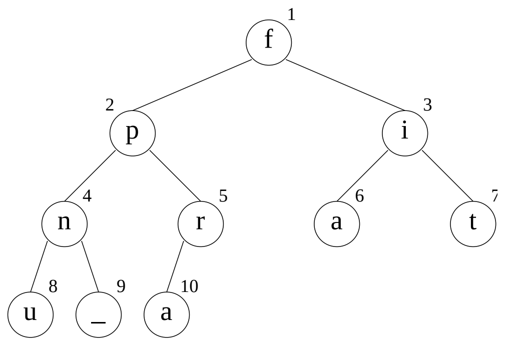


Figure 4: Complete binary tree + hierarchical numbering

The tree in figure 4 is represented by the following array:

1	2	3	4	5	6	7	8	9	10
f	p	i	n	r	a	t	u	_	a

This static representation is a good implementation for *complete* trees (or *perfect* trees, as in figure 8), which only needs a vector that has the same size as the tree's. However, with any other kind of trees, the needed size might be less "optimized".

With a Python list (or in any language that works in base 0), we can simplify by ignoring the first place. The question that remains is the length that has to be provided...

Representation of the tree in figure 1:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
	e	c	u	u	l	q			q	e		o	n	...

The space used is higher as the height of the tree increases. A *degenerate* (*filiform*) tree that has n nodes needs a $(2^n - 1)$-size array!

Representation of the *filiform* tree in figure 5:

0	1	2	3	...	7	...	14	...	29	...	58	...
	f		i	...	l	...	i	...	f	...	e	...

Representation of the *right comb* in figure 6:

0	1	2	3	...	6	7	...	14	15	...	30	31	...	62	63	...
	p	d	e	...	r	i	...	o	g	...	i	n	...	t	e	...

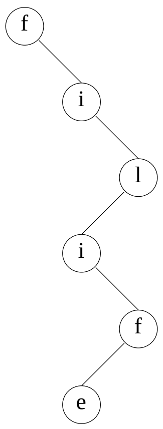


Figure 5: Filiform tree

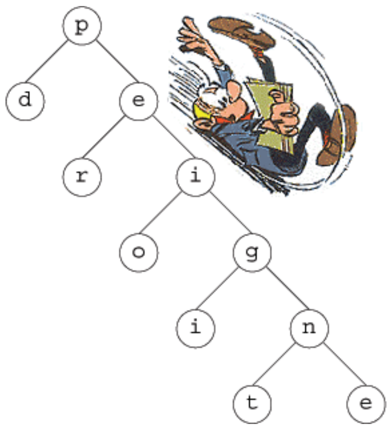


Figure 6: Right comb

Utilisation

- The root is at the first position in the vector.
- Let i be the actual node position:
  - its left child is at position $2i$
  - the right one is at position $2i + 1$
  - its parent (except for the root) is at position i div 2.

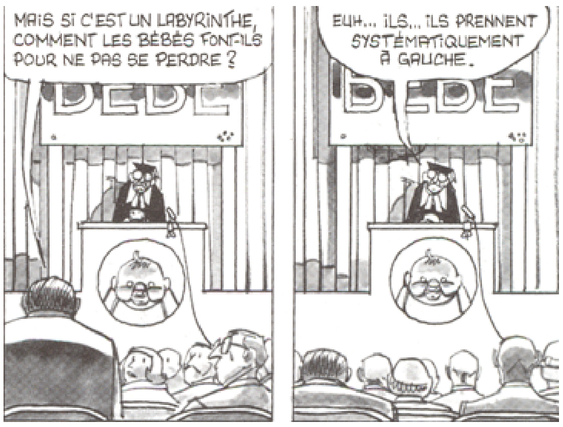
We must find a way to reference an empty tree (for instance put $\emptyset$ to the root), except for complete trees for which we just have to know the size.

3 Traverse a binary tree

3.1 Depth-First Search (Parcours profondeur)

The depth-first traversal (beginning on the left) consists of going down in the tree to the left as far as possible. When we can no longer go down, we go back one level: coming from the left, descend once to the right, otherwise we still go back. . . And so on. . .

The simplest way to figure this traversal is recursive. This is simply to traverse the left sub-tree, then the right sub-tree!


MAIS SI C'EST UN LABYRINTHE, COMMENT LES BÉBÉS FONT-ILS POUR NE PAS SE PERDRE ?
DEDE
EUH... ILS... ILS PRENNENT SYSTÉMATIQUEMENT À GAUCHE.
DEDE

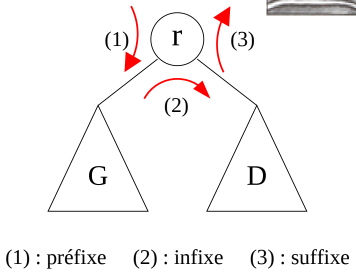


Figure 7: Binary tree depth-first traversal

During the depth-first traversal, each node is met three times:

(1) before the descent on the left sub-tree: **preorder** (préfixe),
(2) going back from the left, before descending right: **inorder** (infixe),
(3) going back from the right: **postorder** (suffixe).

Examples: the three induced orders by a depth-first traversal of a binary tree.

- The preorder on the tree in figure 8 gives: *lui_est_complet*
- The inorder on the tree in figure 4 gives: *un_parfait*
- The postorder on the tree in figure 1 gives: *quelconque*

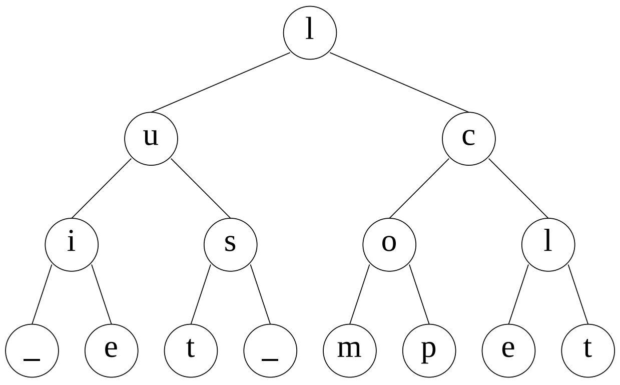


Figure 8: Perfect binary tree

3.2 Breadth-first Search (Parcours largeur)

The *breadth-first* or *level-order* traversal consists of visiting the nodes level by level (usually from left to right): it is the hierarchical order.

The algorithm uses a *queue*:

- first the root is enqueued
- while the queue is not empty:
    - the first element is dequeued
    - the existing children of the dequeued node are enqueued (first the left then the right).

With the static representation, the algorithm can be implemented without queue. It is simply a sequential traversal of the vector (ignoring the possible $\emptyset$ values). But, this is really interesting with complete trees! Otherwise, the classical implementation can be written with an integer queue that contains the hierarchical numbers of the nodes.