> Version pdf

# Arbre binaire (Bac 🎯)

## Définition

**"Point de cours 1 : arbre binaire"**

Un **arbre binaire** est un ensemble fini de **noeuds**. Cet ensemble peut être vide.

Un **noeud** d'arbre binaire est constitué de trois champs :

- un champ d'information contient un *élément*
- un premier champ *enfant* contient un lien appelé *fils gauche* vers un autre noeud ou une valeur représentant l'absence de noeud
- un second champ *enfant* contient un lien appelé *fils droit* vers un autre noeud ou une valeur représentant l'absence de noeud

**"Un noeud d'arbre binaire a toujours deux enfants"**

Un noeud d'arbre binaire est normalement toujours représenté avec ses deux enfants même s'ils sont vides.

On peut alors définir un **arbre binaire** de façon *récursive* :

- soit c'est **l'arbre vide** s'il ne contient aucun noeud
- soit il est constitué d'un noeud spécial appelé **racine** de l'arbre dont le *fils droit* et le *fils gauche* sont des **arbres binaires**

Le fils gauche non vide d'un arbre binaire non vide est appelé *sous-arbre gauche*.

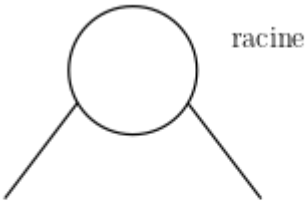
Le fils droit non vide d'un arbre binaire non vide est appelé *sous-arbre droit*.

### ⚠ "Distinction entre fils gauche et droit"

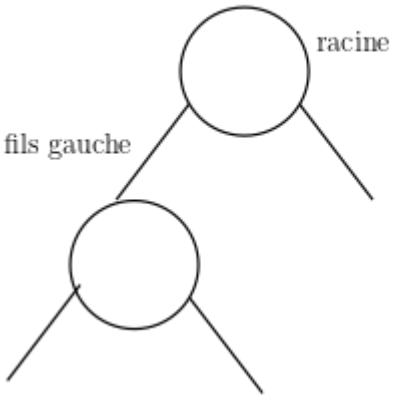
Attention, il faut bien distinguer le fils gauche et le fils droit : il existe deux structures d'arbre binaire distinctes avec deux noeuds.

Arbre binaire vide

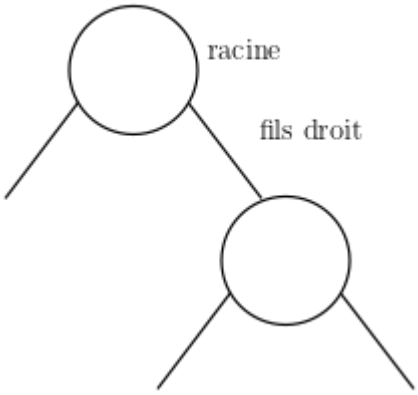
Arbre binaire avec un seul noeud




Arbre binaire 1 avec deux noeuds




Arbre binaire 2 avec deux noeuds




### ⚠ "Arbres binaires homogènes ou pas"

En général, comme pour les structures linéaires, nous construirons des arbres binaires stockant des éléments de même type. Mais ce ne sera pas le cas pour les arbres de

calcul comme celui-présenté dans l'introduction des structures arborescentes.

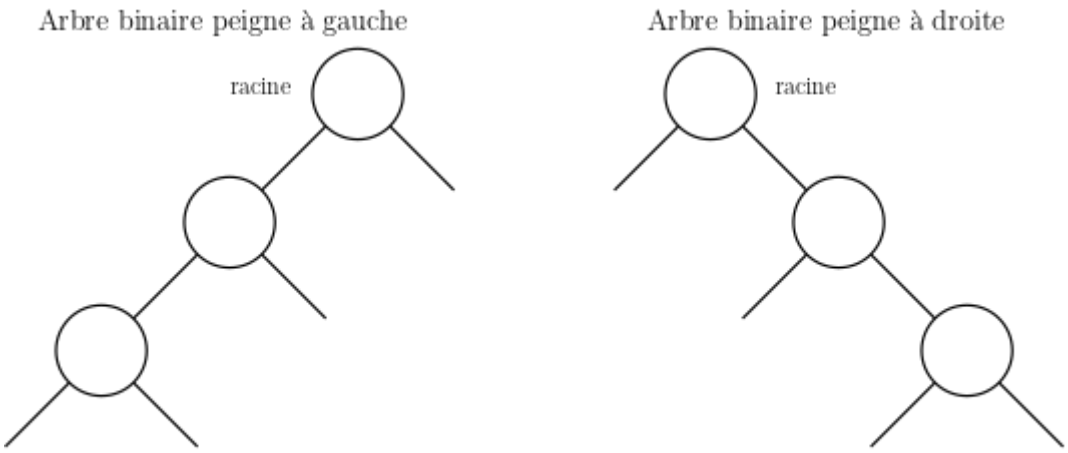
# Mesures d'un arbre binaire

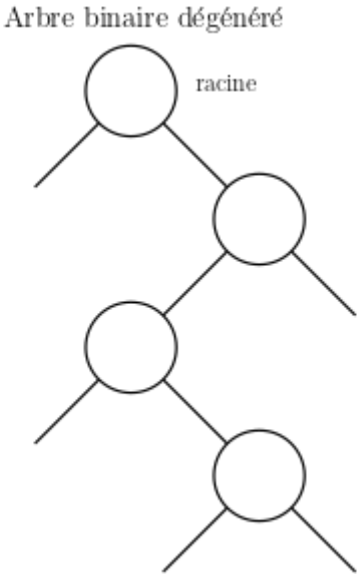
**"Point de cours 3"**

Un **arbre binaire** dont tous les *noeuds internes* ont un seul fils (non vide) est un **arbre binaire dégénéré** : il possède une unique *feuille*.

Un *arbre binaire dégénéré* peut être assimilé à une *liste chaînée*.

Un *arbre binaire dégénéré* dont tous les fils non vides sont du même côté est un **arbre peigne**.



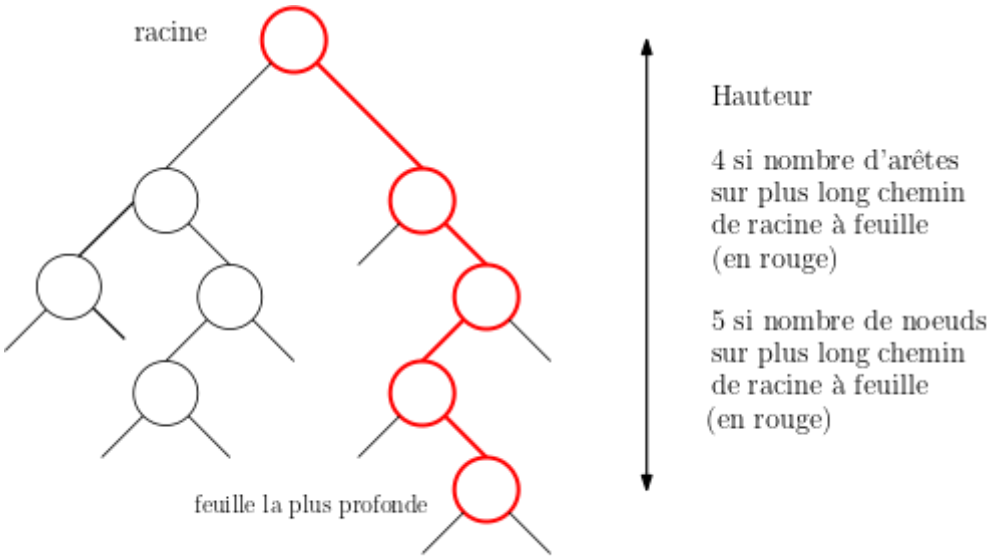




**"Point de cours 4"**

- La **taille** d'un arbre binaire est son nombre de noeuds.
- Dans un arbre binaire, il existe une unique chaîne, appelée *chemin simple*, reliant par des liens de filiation, le noeud racine à un autre noeud. Par la suite, on désigne par *arête* un lien de filiation entre deux noeuds.
    - La **profondeur** d'un noeud est le nombre d'arêtes dans le chemin simple reliant la racine à ce noeud.
    - Un **niveau** dans un arbre binaire est l'ensemble des noeuds de même profondeur.
    - La **hauteur** d'un arbre binaire peut se définir comme **le nombre de noeuds** ou comme **le nombre d'arêtes** dans le *plus long chemin simple reliant la racine à une feuille*.[1] C'est une mesure importante car elle représente le plus long chemin pour accéder à un élément depuis la racine. Nous privilégierons la définition suivante :

> **"Définition choisie pour la hauteur d'un arbre binaire"**
>
> La **hauteur** d'un arbre binaire est le nombre de noeuds dans le *plus long chemin simple reliant la racine à une feuille*. Ainsi un *arbre vide* a pour hauteur 0.

| Définition | Hauteur de l'arbre avec un noeud | Hauteur de l'arbre vide |
|---|---|---|
| Par le nombre d'arêtes | 0 | $-1$ |
| Par le nombre de noeuds | 1 | 0 |

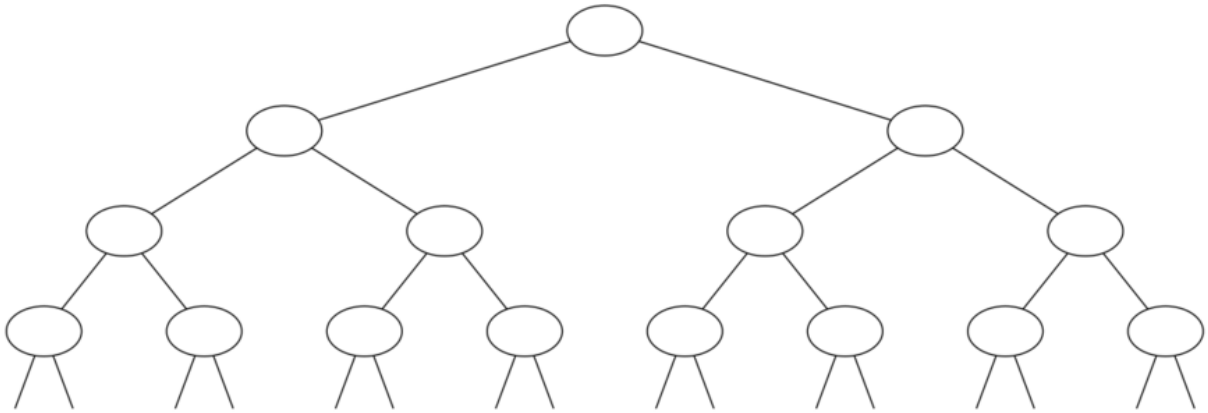



Arbre de taille 9

### "Point de cours 5"

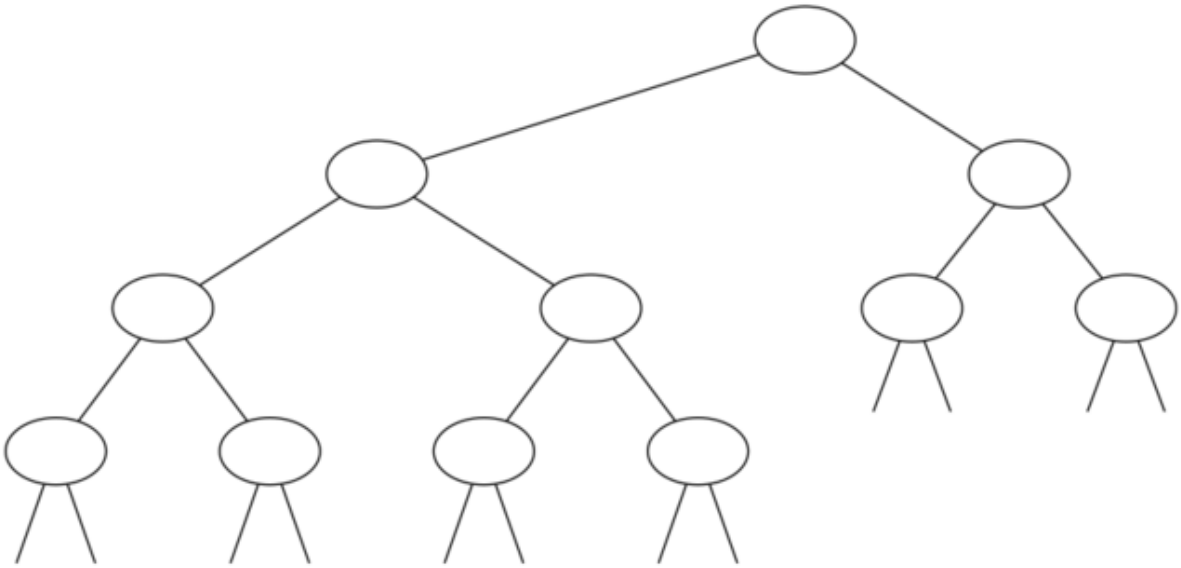
- Un **arbre binaire parfait** est un arbre binaire dans lequel toutes les feuilles sont à la même profondeur c'est-à-dire que tous les niveaux sont remplis complètement.

#### "Un arbre binaire parfait de taille 14 et de hauteur 4"



- Un **arbre binaire presque complet à gauche** est un arbre binaire dans lequel tous les niveaux sont remplis complètement sauf le plus profond où les feuilles sont serrées à gauche.

#### "Un arbre binaire presque complet à gauche de hauteur 4"



### "Point de cours 6"

Soit un **arbre binaire** de taille $n$ et de hauteur $h$ (définition avec le nombre de noeuds). On a les encadrements :

#### "Encadrement de la taille d'un arbre binaire"

$$h \leqslant n \leqslant 2^h - 1 < 2^h$$

#### "Encadrement de la hauteur d'un arbre binaire"

$$\log_2(n) < h \leqslant n$$

Un **arbre binaire** de taille $n$ est :

- de *hauteur maximale* si c'est un **arbre binaire dégénéré** et en particulier si c'est un **arbre peigne**
- de *hauteur minimale* si c'est un **arbre binaire presque complet à gauche** ou si c'est un **arbre binaire parfait**. Dans ce cas on a $h = \lfloor \log_2(n) \rfloor + 1$ qui est exactement le nombre de chiffres en base $2$ de $n$.

**"Hauteur d'un arbre et complexité des algorithmes sur les arbres binaires"**

Si on doit stocker $n$ éléments dans un arbre binaire, la propriété précédente nous permet d'estimer la hauteur de l'arbre c'est-à-dire la longueur du chemin le plus long pour accéder à un élément depuis la racine :

- Dans le *meilleur des cas*, si on peut construire un *arbre parfait* ou *presque complet à gauche*, on aura une hauteur d'arbre $h$ de l'ordre de $\log_2(n)$. C'est bien meilleur que pour une liste chaînée (complexité linéaire) !
- Dans le *pire des cas*, si on construit un *arbre peigne* ou *dégénéré* on se retrouve dans une situation équivalente à une liste chaînée .

# Interface et implémentation

**"Point de cours 7"**

Un **arbre binaire** est un ensemble fini de **noeuds** donc il faut d'abord représenter un noeud.

Un noeud est constitué d'un élément, d'un fils gauche et d'un fils droit donc on peut représenter un noeud par un objet d'une classe `Noeud` avec trois attributs : `element` , `gauche` et `droite` .

On choisit de représenter l'absence de noeud par la valeur spéciale `None`

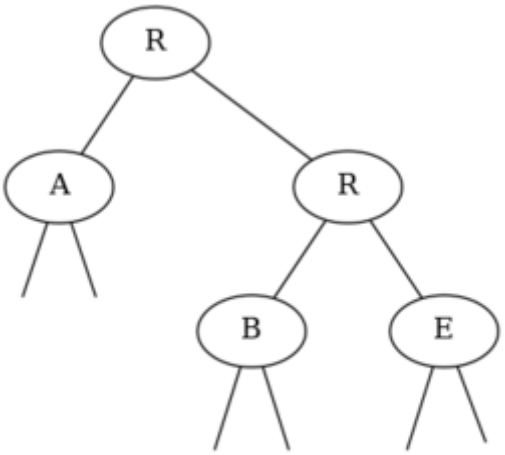
```
class Noeud:
"""Noeud pour arbre binaire"""

    def __init__(self, g, e, d):
        self.gauche = g # lien vers fils gauche g éventuellement vide (None)
        self.element = e # élément e stocké dans le noeud
        self.droit = d # lien vers fils droit d éventuellement vide (None)
```

**"Du noeud à l'arbre binaire"**

Cette classe `Noeud` permet déjà de construire des arbres binaires. Par exemple l'expression :

```
Noeud(Noeud(None, 'A', None), 'R',
       Noeud(Noeud(None, 'B', None), 'R', Noeud(None, 'E', None)))
```

permet de construire l'arbre binaire ci-dessous :




**"Cas de l'arbre binaire vide"**

Attention cette classe `Noeud` ne permet pas de représenter l'arbre binaire vide et ne suffit pour implémenter en paradigme objet (POO) un type abstrait **arbre binaire**.

**"Point de cours 8"**

Pour le type abstrait **arbre binaire** on a besoin de représenter :

- l'arbre binaire vide : on peut choisir une valeur spéciale comme `None`
- un noeud par exemple avec un objet de la classe `Noeud` définie précédemment.

On peut alors définir un ensemble de fonctions constituant une *interface* fonctionnelle minimale pour le type abstrait **arbre binaire**. Le paramètre `arbre` désigne un arbre qui est de type `Noeud` s'il est non vide ou qui vaut `None` sinon.

| Fonction | Signature | Description |
|---|---|---|
| `arbre_vide` | `arbre_vide()` | Renvoie un arbre vide représenté par `None` |
| `est_vide` | `est_vide(arbre)` | Renvoie un booléen indiquant si `arbre` est vide |
| `element_racine` | `element_racine(arbre)` | Renvoie l'élément à la racine de l'arbre s'il est non vide |
| `gauche` | `gauche(arbre)` | Renvoie le sous-arbre fils gauche de l'arbre s'il est non vide |
| `droit` | `droit(arbre)` | Renvoie le sous-arbre fils droit de l'arbre s'il est non vide |
| `creer_arbre` | `creer_arbre(g, e, d)` | Renvoie un noeud constitué de l'élément `e`, du sous-arbre gauche `g` et du sous-arbre droit `d` |

A partir de cette interface fonctionnelle, on peut donner une implémentation d'**arbre binaire immuable** c'est-à-dire que chaque fonction renvoie un nouvel arbre binaire et ne modifie jamais l'arbre binaire passé en paramètre :

**"Implémentation d'un type d'arbre binaire immuable"**

```
def arbre_vide():
    return None

def est_vide(arbre):
    return arbre is None

def gauche(arbre):
    assert not est_vide(arbre), "Arbre vide"
    return arbre.gauche

def droit(arbre):
    assert not est_vide(arbre), "Arbre vide"
    return arbre.droit

def element_racine(arbre):
    assert not est_vide(arbre), "Arbre vide"
    return arbre.element

def creer_arbre(g, e, d):
    return Noeud(g, e, d)
```

# Parcours

**"Point de cours 9"**

Un **parcours** d'un **arbre binaire** est un algorithme qui visite chaque noeud de l'arbre et lui applique un certain traitement (affichage, récupération de la valeur de l'élément stocké, de la profondeur ...). Le parcours peut traverser plusieurs fois un noeud mais le traitement est appliqué une seule fois pour chaque noeud.

Pour un **arbre binaire** on distingue deux principaux algorithmes de parcours :

- le **parcours en profondeur**
- le **parcours en largeur**

**"⏱️ Complexité"**

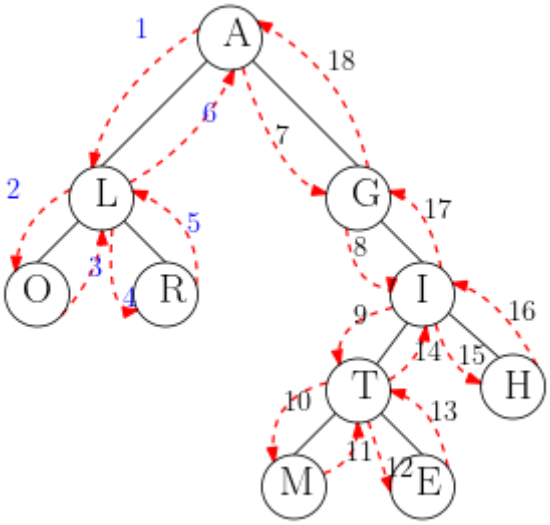
Si le nombre de fois où un algorithme de parcours traverse un noeud est borné, et si la complexité de traitement d'un noeud l'est aussi, alors un algorithme de parcours d'arbre binaire a une **complexité temporelle linéaire** par rapport à la taille de l'arbre. C'est le cas des algorithmes de **parcours en profondeur** et de **parcours en largeur**.

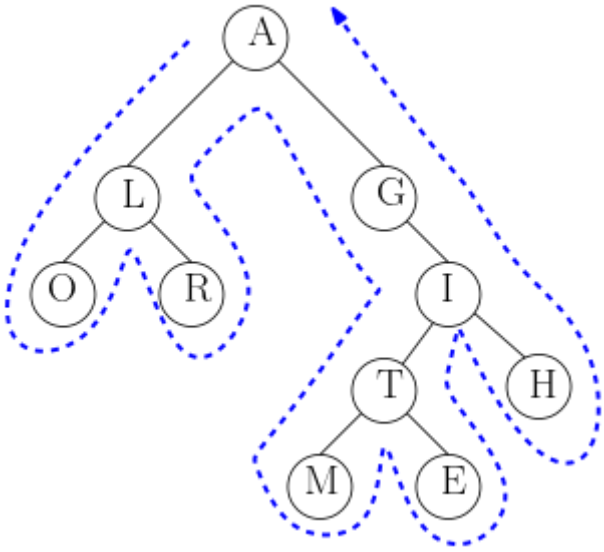
### "Point de cours 10"

Le **parcours en profondeur** d'un **arbre binaire** part de la racine et se dirige préférentiellement vers les feuilles en explorant les branches jusqu'au bout avant de remonter vers les noeuds déjà visités pour parcourir d'autres descendants. On visite les enfants d'un noeud avant ses frères.

> L'acronyme anglais pour désigner le parcours en profondeur est DFS pour Depth First Search.

#### "Exemple de parcours en profondeur d'un arbre binaire"




Le **parcours en profondeur** exploite directement la nature récursive d'un arbre binaire, il est donc naturel de l'implémenter récursivement.

Le **parcours en profondeur** peut passer jusqu'à trois fois par un même noeud : *découverte*, *remontée depuis le sous-arbre/fils gauche* et *remontée depuis le sous-arbre/fils droit*. On distingue donc trois types de parcours en profondeur selon la position du *traitement* appliqué à l'élément stocké dans le noeud par rapport à l'exploration des sous-arbres/fils (gauche puis droite).

**"Parcours en profondeur préfixe"**

Dans le **parcours en profondeur préfixe**, on traite l'élément stocké dans le noeud *avant* de parcourir les sous-arbres.

```
def parcours_prefixe(arbre):
    if arbre is None: # caas de l'arbre vide
        return
    traitement(arbre.racine) # on traite le noeud racine
    parcours_prefixe(arbre.gauche) # on parcourt récursivement le fils gau
    parcours_prefixe(arbre.droit) # on parcourt récursivement le fils droi
```

Pour l'arbre binaire donné en exemple, avec un parcours préfixe, l'ordre de traitement des éléments stockés serait :

```
A - L - O - R - G - I - T - M - E - H
```

**"Parcours en profondeur infixe"**

Dans le **parcours en profondeur infixe**, aon taite l'élément stocké dans le noeud *entre* le parcours du sous-arbre/fils gauche et le parcours du sous-arbre/fils droit.

```
def parcours_infixe(arbre):
    if arbre is None: # caas de l'arbre vide
        return
    parcours_infixe(arbre.gauche) # on parcourt récursivement le fils gauc
    traitement(arbre.racine) # on traite le noeud racine
    parcours_infixe(arbre.droit) # on parcourt récursivement le fils droit
```

Pour l'arbre binaire donné en exemple, avec un parcours infixe, l'ordre de traitement des éléments stockés serait :

```
O - L - R - A - G - M - T -  E - I - H
```

**"Parcours en profondeur suffixe ou postfixe"**

Dans le **parcours en profondeur suffixe** appelé aussi **parcours en profondeur postfixe**, on taite l'élément stocké dans le noeud *après* le parcours du sous-arbre/fils gauche et le parcours du sous-arbre/fils droit.

```
def parcours_postfixe(arbre):
    if arbre is None: # caas de l'arbre vide
        return
    parcours_postfixe(arbre.gauche) # on parcourt récursivement le fils gau
    parcours_postfixe(arbre.droit) # on parcourt récursivement le fils droi
    traitement(arbre.racine) # on traite le noeud racine
```

Pour l'arbre binaire donné en exemple, avec un parcours postfixe, l'ordre de traitement des éléments stockés serait :

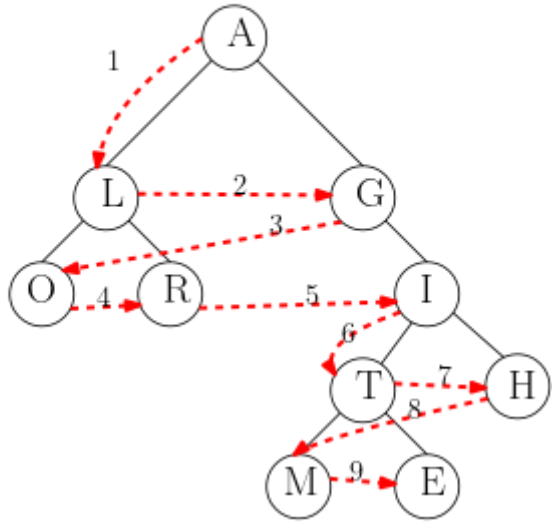
```
O - R - L - M - E - T - H - I - G - A
```

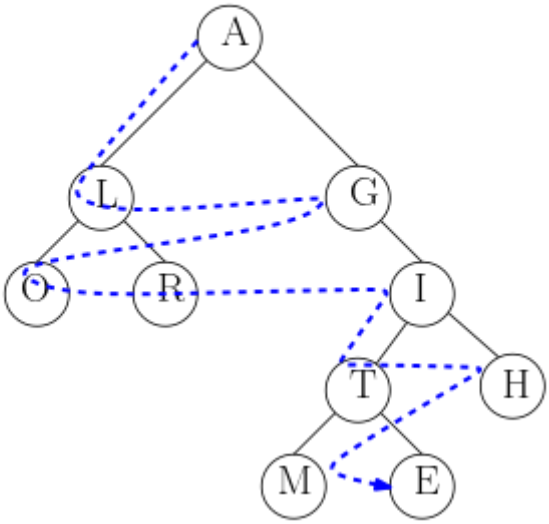
### "Point de cours 11"

Le **parcours en largeur** d'un **arbre binaire** visite les noeuds selon leur ordre de profondeur par rapport à la racine : en partant de la racine on épuise d'abord tous les noeuds d'un même niveau avant de passer au niveau suivant. On visite donc d'abord les frères ou cousins d'un noeud avant de passer à ses descendants. En général on explore un niveau de "gauche à droite".

> L'acronyme anglais pour désigner le parcours en largeur est BFS pour Breadth First Search.

#### "Exemple de parcours en largeur d'un arbre binaire"




Dans le **parcours en largeur**, on ne passe qu'une seule fois par un noeud donc on traite l'élément stocké dans le noeud dès qu'on le visite. Ainsi l'ordre de traitement par le parcours en largeur des éléments stockés dans l'arbre binaire ci-dessus sera :

```
A - L - G - O - R - I - T - H - M - E
```

**"Implémentation du parcours en largeur"**

Pour l'implémentation, on n'utilise pas la récursivité car le parcours en largeur n'explore pas d'abord les fils/sous-arbres. On va utiliser une **file** pour stocker les noeuds en attente ce qui permet de les traiter par profondeur/distance croissante par rapport à la racine.

On commence par enfiler l'arbre (supposé non vide), puis tant que la file n'est pas vide :

- on défile l'arbre en tête de file ;
- on traite l'élément stocké dans le noeud racine puis on enfile les fils/sous-arbres (gauche puis droite) s'ils existent.

```
def parcours_largeur(arbre):
    f = creer_file()
    enfiler(f, arbre)
    while not file_vide(f):
        a = defiler(f)
        traitement(a.racine)
        if a.gauche is not None:
            enfiler(f, a.gauche)
        if a.droit is not None:
            enfiler(f, a.droit)
```

1. La hauteur ne diffère que de $1$ entre les deux définitions, au Bac, la définition utilisée est toujours précisée dans le sujet. ↩