Chapitre 11 - Les arbres binaires

Objectifs :

- ▷ Connaître les algorithmes sur les arbres binaires
- ▷ Calculer la taille et la hauteur d'un arbre binaire
- ▷ Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur ou profondeur d'abord).

1 Les arbres binaires

Parmi la forêt d'arbres possibles, on s'intéressera essentiellement aux arbres dit **binaires** :

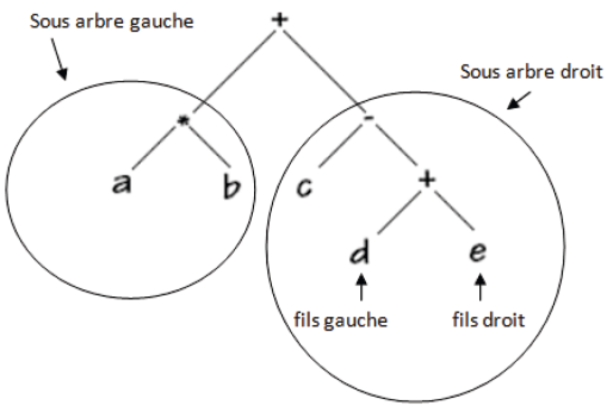
1.1 Définition

Définition

Un arbre binaire est un arbre de **degré 2** dont les noeuds sont de degré 2 **au plus**.

Vocabulaire :
Les enfants d'un noeud sont lus **de gauche à droite** et sont appelés : **fils gauche** et **fils droit**.

Par exemple, l'expression $a \times b + c - (d + e)$ est représentée par l'arbre binaire ci-dessous :



Question 1 : Parmi les arbres du cours sur les arbres (chapitre 10), lesquels sont binaires ?

Réponses :

1.2 Remarques

Les arbres binaires forment une structure de données qui peut se définir de façon **récursive**.

A retenir !

Un arbre binaire est :

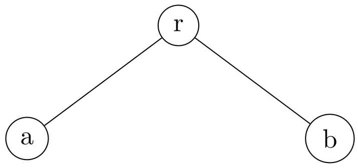
- ▷ Soit **vide**.
- ▷ Soit composé d'une racine portant une étiquette (clé) et d'une paire d'arbres binaires, appelés **sous-arbre gauche** et **sous-arbre droit**.

2 Représentations d'un arbre

Les arbres binaires sont utilisés dans de très nombreuses activités informatiques. Mais comment cette structure de données est-elle **implémentée** ?

2.0.1 Première méthode : avec un tableau

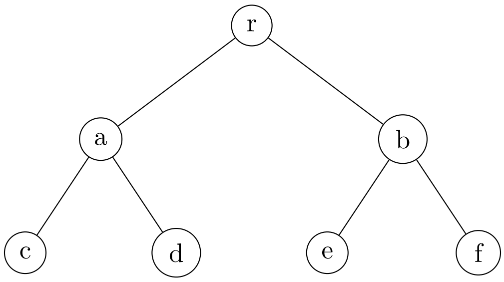
Par exemple :



L 'idée est de représenter l 'arbre avec un tableau :

r	a	b

La racine suivie de ses fils gauche et droit.



Puis on rajoute dans l'ordre les fils gauche et droit de `a`, puis ceux de `b`.

r	a	b	c	d	e	f

Remarques :

- Chaque noeud se repère par son indice n dans la liste, son fils gauche se trouvant alors à l'indice $2n + 1$ et son fils droit à l'indice $2n + 2$.
- b est à l'indice 2 son fils gauche se trouve alors à l'indice 5 et son fils droit à l'indice 6.
- Si un noeud n'a pas de fils on le précise en mettant `None` à sa place.

Notre arbre est alors représenté par le tableau :

```
[r,a,b,c,d,e,f,None,None,None,None,None,None,None,None]
```

Quelle taille doit avoir le tableau ?

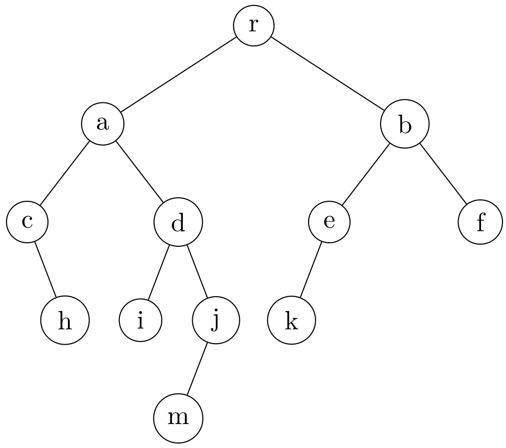
Cet arbre est **complet** (tous les noeuds internes ont deux fils).

Il possède 3 niveaux, sa hauteur ou profondeur est donc de 2.

Donc la taille du tableau sera de : $2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1 = 15$

Il faut compter le nombre de noeuds, y compris les noeuds "fantômes" des feuilles.

Exercice 1 : *Tableau*



1. Quelle est la taille du tableau qui permet de représenter cet arbre ?

   Réponse :

2. Écrire le tableau représentant cet arbre :

   Réponse :

3. Quelle propriété ont les indices des fils gauches et droits ?

   Réponse :

Exercice 2 : *Tableau bis*

Voici un tableau représentant un arbre binaire :

```
['*','-',5,2,6,None,None,None,None,None,None,None,None,None,None]
```

1. Dessiner cet arbre. Que peut-il représenter ?

   Réponse :

Voici un code Python qui crée la liste représentant l'arbre de l'exercice 1 :

```
def creation_arbre(r,profondeur):
    ''' r : la racine (str ou int). la profondeur de l'arbre (int)'''
    Arbre = [r]+[None for i in range(2**(profondeur+1)-2)]
    return Arbre

def insertion_noeud(arbre,n,fg,fd):
    '''Insere les noeuds et leurs enfants dans l'arbre'''
    indice = arbre.index(n)
    arbre[2*indice+1] = fg
    arbre[2*indice+2] = fd

# creation de l'arbre
arbre = creation_arbre("r",5)

# ajout des noeuds par niveau de gauche a droite
insertion_noeud(arbre,"r","a","b")
insertion_noeud(arbre,"a","c","d")
insertion_noeud(arbre,"b","e","f")
insertion_noeud(arbre,"c",None,"h")
insertion_noeud(arbre,"d","i","j")
insertion_noeud(arbre,"e","k",None)
insertion_noeud(arbre,"f",None,None)
insertion_noeud(arbre,"h",None,None)
insertion_noeud(arbre,"i",None,None)
insertion_noeud(arbre,"j","m",None)
insertion_noeud(arbre,"k",None,None)
insertion_noeud(arbre,"m",None,None)

#pour verifier
print(len(arbre))
print(arbre)
```

Voici une fonction qui retourne le parent d'un noeud s'il est dans l'arbre :

```
def parent(arbre,p):
    if p in arbre:
        indice = arbre.index(p)
    if indice%2 == 0:
        return arbre[(indice-2)//2]
    else:
        return arbre[(indice-1)//2]
```

2. Créer les fonctions suivantes :

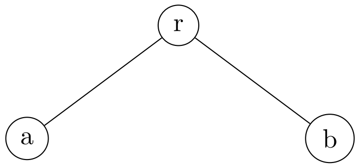
    - ▷ Une fonction qui retourne vrai si l'arbre est vide.
    - ▷ Une fonction qui retourne les enfants d'un noeud.
    - ▷ Deux fonctions qui retournent le fils gauche d'un noeud et son homologue le fils droit s'ils existent.
    - ▷ Une fonction qui retourne vrai si le noeud est la racine de l 'arbre.
    - ▷ Une fonction qui retourne vrai si le noeud est une feuille.
    - ▷ Une fonction qui retourne vrai si le noeud a un frère gauche ou droit.

2.0.2 Deuxième méthode : avec un tableau de tableaux

Comme vous l 'avez sans doute constaté, il est assez fastidieux de représenter un arbre avec un unique tableau surtout pour un arbre très profond.

L 'idée est de représenter l'arbre avec **un tableau contenant des tableaux**.

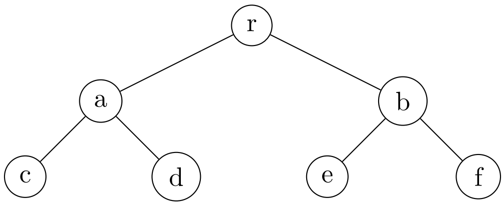
Par exemple :



Cet arbre se représente par le tableau suivant :

```
['r',['a',[],[]],['b',[],[]]]
```

Exercice 3 : *Avec la récursivité*



1. Écrire le tableau représentant cet arbre :

Réponses :

Le code Python ci-dessous, construit l'arbre de l'exercice 1 de manière **récursive**.

- ▷ Les noeuds sont représentés par un **dictionnaire**.
- ▷ L'arbre se construit depuis la racine en construisant les sous-arbres des fils gauche et droit.

```
def noeud(nom, fg = None, fd = None) :
    return {'racine':nom,'fg':fg,'fd':fd}

# creation des noeuds
k = noeud('k')
f = noeud('f')
e = noeud('e', k, None)
b = noeud('b', e, f)
m = noeud('m')
j = noeud('j', m, None)
i = noeud('i')
d = noeud('d', i, j)
h = noeud('h')
c = noeud('c', None, h)
a = noeud('a', c, d)
racine = noeud('r', a, b)

# creation de l'arbre
def construit(arbre) :
    if arbre == None :
        return []
    else:
```

```python
            return [arbre['racine'],construit(arbre['fg']),construit(arbre['fd'])]

arbre1=construit(racine)
print(arbre1)
```

2. Avec la structure de données récursive ci-dessus, écrire :

    ▷ Une fonction qui retourne vrai si l'arbre est vide.
    ▷ Une fonction qui retourne les enfants d'un noeud.
    ▷ Deux fonctions qui retournent le fils gauche d'un noeud et son homologue le fils droit s'ils existent.
    ▷ Une fonction qui retourne vrai si le noeud est la racine de l 'arbre.
    ▷ Une fonction qui retourne vrai si le noeud est une feuille.
    ▷ Une fonction qui retourne vrai si le noeud a un frère gauche ou droit.

3 Algorithmes de parcours des arbres

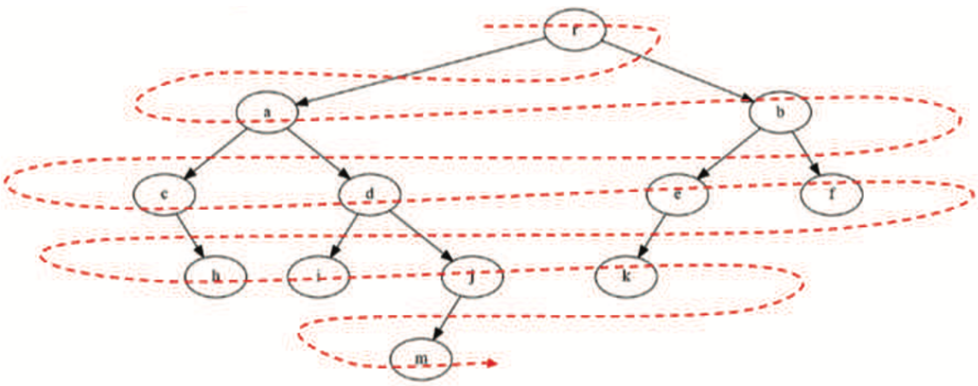
Le **parcours** des arbres, en informatique, désigne les différentes méthodes permettant d'explorer et de visiter les nœuds d'une structure arborescente. Il existe plusieurs types de parcours :

3.1 Le parcours en largeur

Le parcours d'un arbre en largeur consiste à :

— Partir de la racine.
— On visite ensuite son fils gauche.
— Puis, on visite son fils droit.
— Puis, on visite le fils gauche du fils gauche.
— Puis, on visite le fils droit du fils gauche.
— ..., *etc.*

Comme le montre le schéma ci-dessous :



L'idée est la suivante :

On utilise la structure de données **File**.

1. On met l'arbre dans la file.
2. Puis tant que la file n'est pas vide :
    ▷ On défile la file.
    ▷ On récupère sa racine.
    ▷ On enfile son fils gauche s'il existe.
    ▷ On enfile son fils droit s'il existe.

Voici l'algorithme :

Algorithme 1 : Fonction parcours en largeur

Données : Arbre binaire
Sorties : liste
$f \leftarrow$ file vide
liste $\leftarrow$ liste vide
tant que f *non vide* **faire**
  tmp $\leftarrow$ défiler f
  On ajoute *tmp*[0] à *liste*
  si *tmp*[1] *n'est pas vide* **alors**
    On enfile *tmp*[1]
  si *tmp*[2] *n'est pas vide* **alors**
    On enfile *tmp*[2]
retourner *liste*

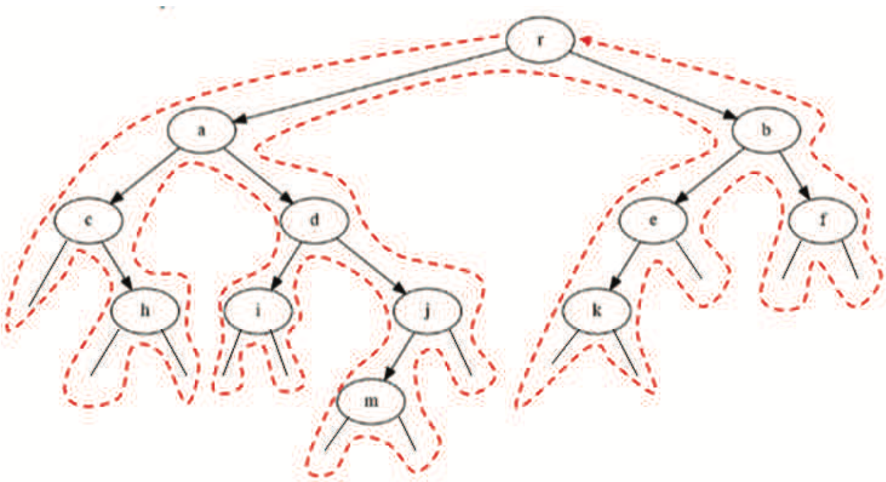
Exercice 4 : Implémenter cette fonction et tester avec l'arbre de l'exercice 1.

Vous devez obtenir : `['r','a','b','c','d','e','f' 'h','i','j','k','m']`

Vous pouvez utilise la classe `File` utilisée dans un précédent chapitre.

3.2 Les parcours en profondeur

On se balade autour de l'arbre en suivant les pointillés :



Dans le schéma ci-dessus, on a rajouté des *noeuds fantômes* pour montrer que l'on peut considérer que chaque noeud est visité **3 fois** :

- Une fois par la gauche
- Une fois par en dessous
- Une fois par la droite

A retenir !

On peut définir les parcours comme suit :

- Dans un parcours **préfixe**, on liste le noeud la **première fois** qu'on le rencontre.
- Dans un parcours **infixe**, on liste le noeud la **seconde fois** qu'on le rencontre. Ce qui correspond à : on liste chaque noeud ayant un fils gauche la seconde fois qu'on le voit et chaque noeud sans fils gauche la première fois qu'on le voit.
- Dans un parcours **suffixe**, on note le noeud la **dernière fois** qu'on le rencontre.

Exercice 5 : Ecrire les sommets de l'arbre ci-dessus dans un parcours *préfixe*, *infixe* et *suffixe*.

Réponses :

Exercice 6 : Voici 3 algorithmes récursifs, dire pour chacun d'entre eux à quel parcours il correspond.

Algorithme 2 : Fonction parcours(arbre)

Données : Arbre binaire

si *l'arbre n'est pas vide* **alors**
- parcours(sous-arbre gauche)
- parcours(sous-arbre droit)
- Afficher la racine de l'arbre

Réponse :

Algorithme 3 : Fonction parcours(arbre)

Données : Arbre binaire

si *l'arbre n'est pas vide* **alors**
- Afficher la racine de l'arbre
- parcours(sous-arbre gauche)
- parcours(sous-arbre droit)

Réponse :

Algorithme 4 : Fonction parcours(arbre)

Données : Arbre binaire

si *l'arbre n'est pas vide* **alors**
- parcours(sous-arbre gauche)
- Afficher la racine de l'arbre
- parcours(sous-arbre droit)

Réponse :

Exercice 7 : Implémenter les trois fonctions ci-dessus : `parcours_prefixe()`, `parcours_infixe()` et `parcours_suffixe()` et confirmer les réponses de l'exercice 5.

4 Représentation avec une classe

Nous allons créer une classe `Noeud` dont les attributs d'instances seront :

- ▷ Le nom (ou valeur) de la racine.
- ▷ Son fils gauche (vide par défaut).
- ▷ Son fils droit (vide par défaut).

De plus on rajoute une méthode spécifique, qui permet d'afficher la racine du noeud avec la fonction `print()`

Ci-dessous la classe `Noeud` et la représentation de l'arbre :

```python
class Noeud:

# Le constructeur
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

# Methode qui permet d'afficher la valeur
# de la racine avec la fonction print
    def __str__(self):
        return str(self.value

k = Noeud('k')
f = Noeud('f')
e = Noeud('e',k,None)
b = Noeud('b',e,f)
m = Noeud('m')
j = Noeud('j',m,None)
i = Noeud('i')
d = Noeud('d',i,j)
h = Noeud('h')
c = Noeud('c',None,h)
a = Noeud('a',c,d)

arbre = Noeud('r',a,b)
print(arbre)                         # affiche r
```

Remarque :

On aurait pu écrire la représentation de l'arbre sur une seule ligne mais c'est assez compliqué de ne pas se perdre...

```python
arbre=Noeud('r',Noeud('a',Noeud('c',None,Noeud('h')),Noeud('d',Noeud('i'),
Noeud('j',Noeud('m'),None))),Noeud('b',Noeud('e',Noeud('k',None,None),None),
    Noeud('f')))
```

Exercice 8 : Écrire une méthode `estFeuille()` qui renvoie vrai si le noeud est une feuille et faux sinon.

Exercice 9 : Écrire une méthode `hauteur()` qui renvoie la hauteur de l'arbre.

Exercice 10 : Écrire les trois méthodes de parcours en profondeur de l'arbre :

- — `parcours_prefixe()`
- — `parcours_infixe()`
- — `parcours_suffixe()`

Confirmer les réponses de l'exercice 5.