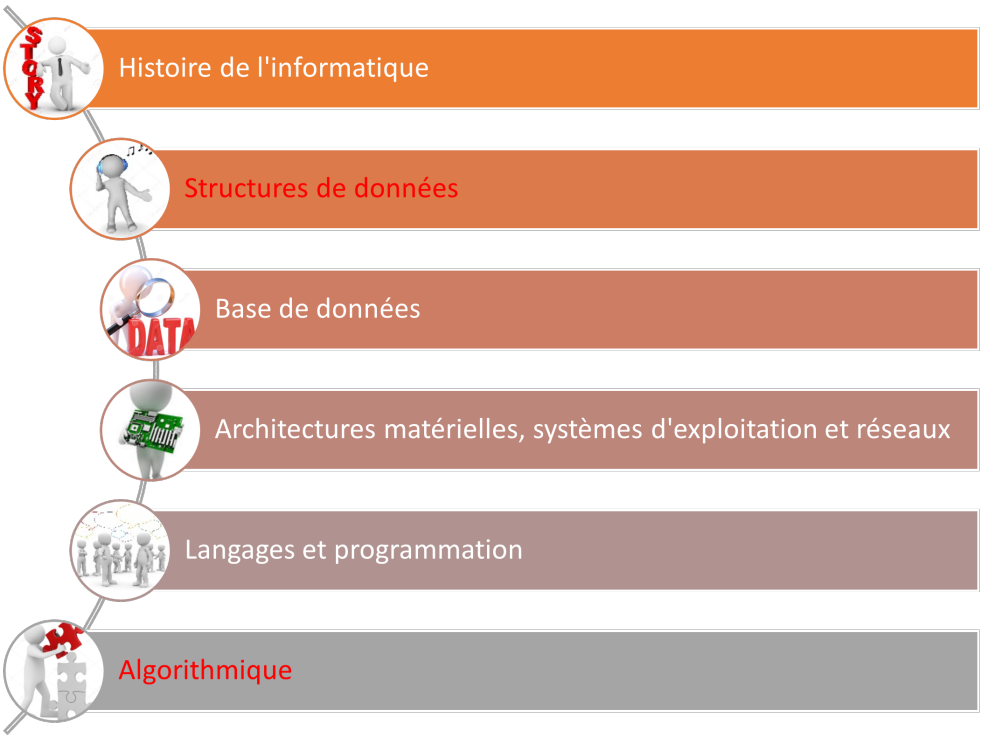


Table des matières

Les listes, les piles et files sont des structures de données linéaires. Des structures non linéaires peuvent aussi être utilisées dans la représentation de données lorsque celles-ci sont hiérarchisées et surtout si leur nombre est important.

Ainsi nous évoquerons dans un prochain chapitre la notion de graphe, et dans celui-ci nous abordons la notion d'arbre et en particulier la notion d'arbre binaire.

1 Arbres binaires

1.1 Définitions

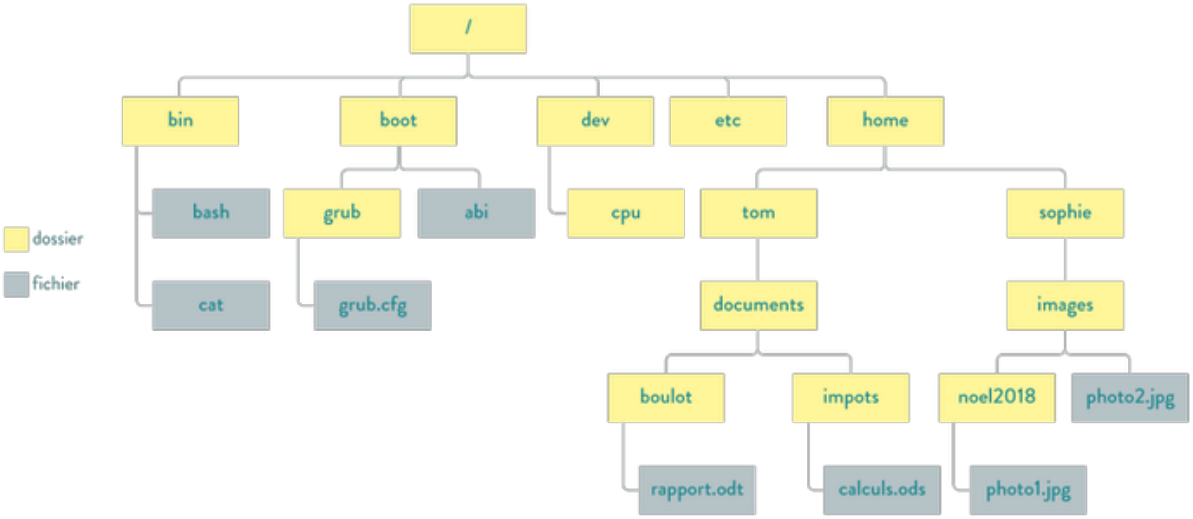
De nombreuses données peuvent être représentées à l'aide d'un arbre :

- Un arbre généalogique ;
- Un organigramme dans une société ;
- La représentation des fichiers en mémoire par un système d'exploitation ;
- La représentation des différentes pages qui constituent un site internet et les liens pour naviguer entre elles....

On peut définir les arbres de plusieurs manières :

- ▶ par divers éléments qui le caractérisent : nœuds, feuilles, racine, branches...
- ▶ Un arbre est un graphe particulier, il pourrait être défini avec le vocabulaire (sommet arête...) des graphes
- ▶ Une autre façon de définir un arbre, c'est de dire qu'il est composé de nœuds, et que chaque nœud (mise à part la racine) a un seul parent p. Un nœud qui admet pour parent p est un enfant de p. Une feuille est un nœud qui n'a pas d'enfant.
- ▶ Nous définirons les arbres binaires de manière récursive.

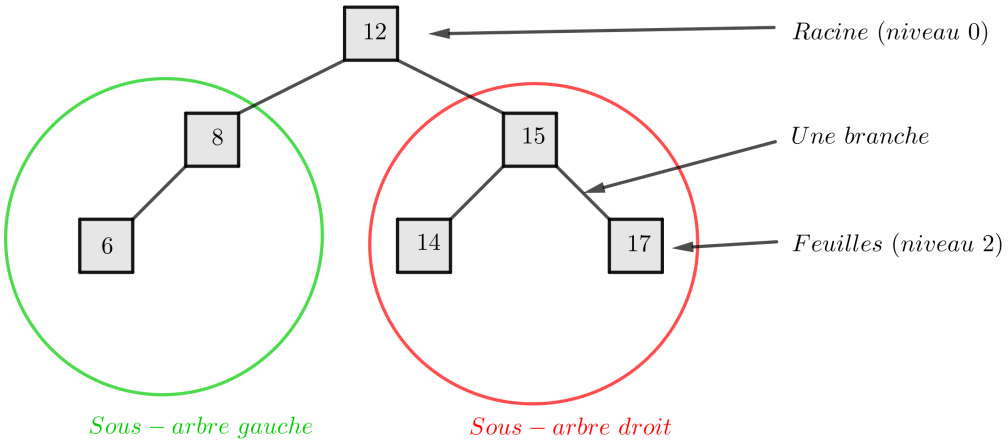
Voici un exemple d'arbre avec l'arborescence des fichiers sous linux :



Définition :

Un arbre binaire est un ensemble fini de nœuds correspondant à l'un des deux cas suivants :

- soit l'arbre binaire est vide, c'est à dire qu'il ne contient aucun nœud ;
- soit l'arbre binaire n'est pas vide, et ses nœuds sont structurés de la manière suivante :
  - ▶ un nœud est appelé la **racine** de l'arbre binaire ;
  - ▶ tous les autres nœuds forment récursivement deux sous-ensembles, appelés respectivement **sous-arbre gauche** et **sous-arbre droit**, qui sont deux arbres binaires.
  - ▶ la racine de l'arbre binaire est reliée à ces deux sous-arbres gauche et droit.



Arbre binaire n°1

1.2 Notions de taille et de hauteur d'un arbre binaire

Définition :
On appelle **taille** d'un arbre binaire le nombre de nœuds de l'arbre

Définition :
On appelle **hauteur** d'un arbre binaire le plus grand nombre de nœuds en allant de la racine à une feuille (ou jusqu'à un sous-arbre vide), racine et feuille comprises.

Remarque :
Si on considère que la racine a un niveau 0 et que pour tout nœud de niveau k, ses enfants ont pour niveau $k+1$. La **hauteur** d'un arbre binaire correspond au nombre de niveaux de nœuds de l'arbre binaire.

Propriété :
Un arbre binaire dont tout parent a exactement 2 enfants est dit complet.
Si la hauteur d'un arbre binaire complet est h, alors sa taille sera : $2^h - 1$

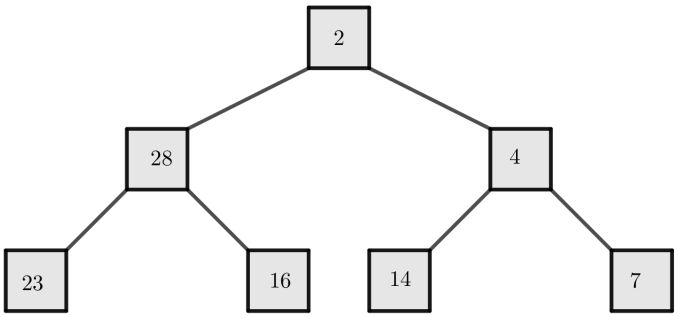
Preuve :
L'arbre étant complet au niveau 0, on a 1 nœud
au niveau 1, on a 2 nœuds
...
au niveau $h-1$ (dernier niveau puisque l'on commence à compter à 0) on a 2^{h-1} nœuds
Soit en tout : $1 + 2 + \ldots + 2^{h-1} = \dfrac{2^h - 1}{2 - 1} = 2^h - 1$
(Somme des h premiers termes d'une suite géométrique de raison 2 de premier terme 1)

Propriété :
Si on considère un arbre binaire de hauteur h, alors sa taille T vérifie :

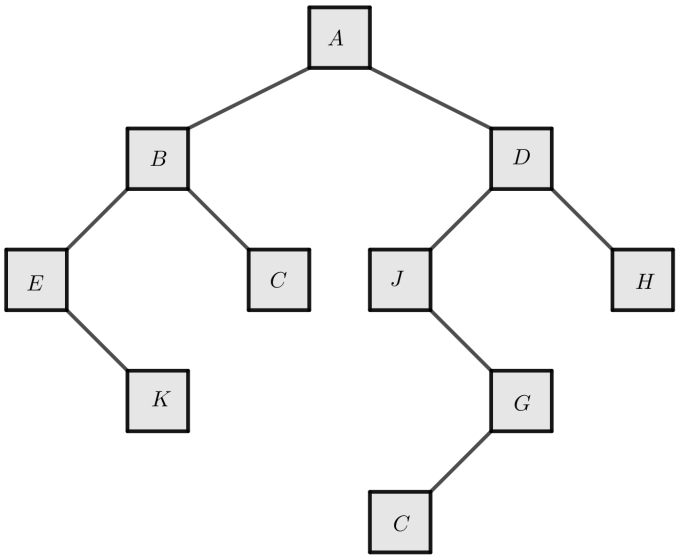
$$h \leqslant T \leqslant 2^h - 1$$

Preuve : La taille minimale d'un arbre binaire de hauteur h est h, cela correspond à un arbre binaire dont chaque parent a un et un seul enfant.
La taille maximale d'un arbre binaire de hauteur h est celle d'un arbre binaire complet de hauteur h.
D'où la double inégalité.

Des exemples :



Arbre binaire n°2 :



Arbre binaire n°3 :

1.3 Implémentation en Python

1.3.1 Avec des listes

On peut utiliser une liste pour implémenter une structure d'arbres binaires :

- une liste vide pour un arbre binaire vide ;
- une liste contenant 3 éléments : la valeur, le sous-arbre gauche et le sous-arbre droit pour un arbre binaire non vide

Il est nécessaire de disposer des fonctions pour :

- ▶ créer un arbre vide ou un arbre dont la racine est donnée ;
- ▶ savoir si un arbre est vide ou pas ;
- ▶ obtenir le sous-arbre gauche ou le sous-arbre droit à partir de la racine ;
- ▶ insérer un nouveau nœud.

Pour l'insertion d'un nouveau nœud, on peut procéder de différentes manières, on a fait le choix d'insérer récursivement la valeur dans le sous-arbre gauche si elle est inférieure ou égale à la racine, dans le sous-arbre droit sinon.
Cela suppose que les valeurs indiquées sont toutes de même type et qu'elles puissent être comparées entre elles (des chaines de caractère avec l'ordre alphabétique ou des nombres de type int ou float avec l'ordre usuel)

Vous noterez qu'en fin de script les tests conduisent à la création de l'arbre binaire n°1.

```
def creer_arbre(r = None):
    """renvoie un arbre vide
    ou un arbre de racine r"""
    if r:
        return[r, [], []]
    else:
        return []

def arbre_vide(a):
    return a == []

def get_gauche(a):
    if not arbre_vide(a):
        return a[1]

def get_droite(a):
    if not arbre_vide(a):
        return a[2]

def insere(a, valeur):
    if arbre_vide(a):
        a.append(valeur)
        a.append([])
        a.append([])
    elif valeur <= a[0]:
        insere(a[1], valeur)
    else:
        insere(a[2], valeur)

if __name__ == "__main__":
    a = creer_arbre(12)
    insere(a, 8)
    insere(a, 6)
    insere(a, 15)
    insere(a, 14)
    insere(a, 17)
    print(a)
```

arbres_binaires_recursifs.py

1.3.2 Encapsulation : premier exemple

Nous avons construit un arbre binaire en Python à l'aide de listes dans le paragraphe précédent, définissons maintenant une classe `Noeud` dans laquelle chaque objet est caractérisé par ses trois attributs : `valeur`, `gauche` et `droit`.

Nous disposons des méthodes pour :

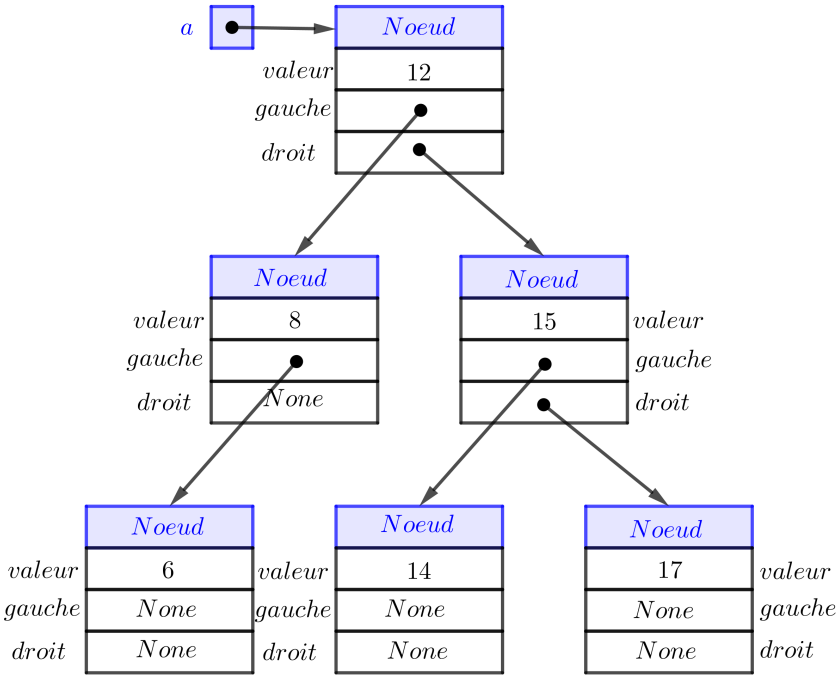
- ▶ créer un arbre vide ou un arbre dont la racine est donnée ;
- ▶ savoir si un nœud est vide ou pas ;
- ▶ obtenir le sous-arbre gauche ou le sous-arbre gauche droit à partir de la racine ;
- ▶ insérer un nouveau nœud au sous-arbre gauche ou au sous-arbre droit.

Vous noterez :

- l'utilisation de la fonction récursive `affiche()` pour permettre d'afficher l'arbre ainsi construit ;
- dans la partie test, on construit l'arbre binaire $n°1$ donné en exemple page 2.
- une représentation de l'arbre ainsi construit vous est donnée dans la figure de droite.

```
class Noeud:
    """un noeud d'un arbre binaire"""
    def __init__(self, valeur = None):
        self.valeur = valeur
        self.gauche = None
        self.droit  = None

    def ajoute_gauche(self, valeur):
        if self.valeur == None:
            self.valeur = valeur
        elif self.gauche is None:
            self.gauche = Noeud(valeur)
        else:
            self.gauche.ajoute_gauche(
    valeur)

    def ajoute_droit(self, valeur):
        if self.valeur == None:
            self.valeur = valeur
        elif self.droit is None:
            self.droit = Noeud(valeur)
        else:
            self.droit.ajoute_droit(valeur)

    def get_valeur(self):
        return self.valeur

    def get_gauche(self):
        return self.gauche

    def get_droit(self):
        return self.droit

if __name__ == '__main__':
    a = Noeud(12)
    a.ajoute_droit(15)
    a.droit.ajoute_gauche(14)
    a.ajoute_droit(17)
    a.ajoute_gauche(8)
    a.ajoute_gauche(6)
    def affiche(T):
        if T != None:
            return (T.get_valeur(),affiche(
    T.get_gauche()),affiche(T.get_droit()))
    print(affiche(a))
```

classe_noeud.py



1.3.3 Encapsulation : deuxième exemple

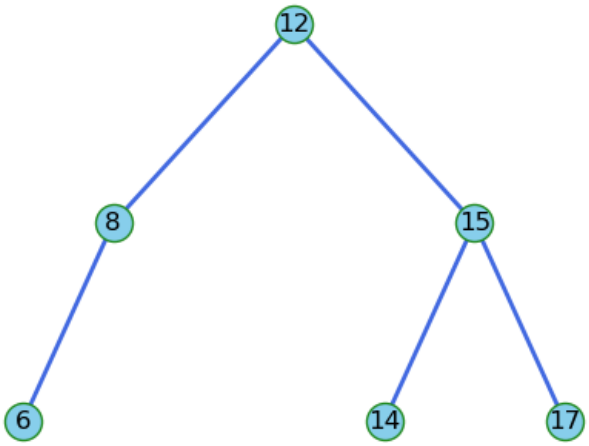
Dans l'implémentation précédente, les deux enfants d'un nœud sont des nœuds. Une autre approche récursive, consiste à dire que les deux enfants d'un nœud sont eux mêmes des arbres.
Nous disposons des méthodes pour :

- ▶ créer un arbre dont la racine est donnée ;
- ▶ insérer un arbre au sous-arbre gauche ou au sous-arbre droit.

Vous noterez :

- l'importation du module `représentation_arbre` qui permet de construire une représentation de l'arbre binaire construit ;
- dans la partie test, on construit l'arbre binaire $n°1$ donné en exemple page 2.
- une représentation de l'arbre ainsi construit vous est donnée dans la figure de droite.

```python
from representation_arbre import *

class ArbreBinaire:
    """un valeur d'un arbre binaire"""
    def __init__(self, valeur, gauche =
    None, droit  = None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit  = droit

    def ajoute_gauche(self, valeur):
        self.gauche = ArbreBinaire(valeur)

    def ajoute_droit(self, valeur):
        self.droit = ArbreBinaire(valeur)

if __name__ == '__main__':
    a = ArbreBinaire(12)
    a.ajoute_droit(15)
    a.droit.ajoute_droit(17)
    a.droit.ajoute_gauche(14)
    a.ajoute_gauche(8)
    a.gauche.ajoute_gauche(6)

    arbre = repr_graph(a)
```



classe_arbre_binaire.py

Dans la suite de ce cours, nous privilégierons l'utilisation de cette classe `ArbreBinaire`.

2 Parcours d'un arbre binaire

On peut parcourir un arbre binaire pour plusieurs raisons, calculer sa hauteur, sa taille, déterminer si une valeur appartient à l'arbre etc...
On distingue deux types de parcours :

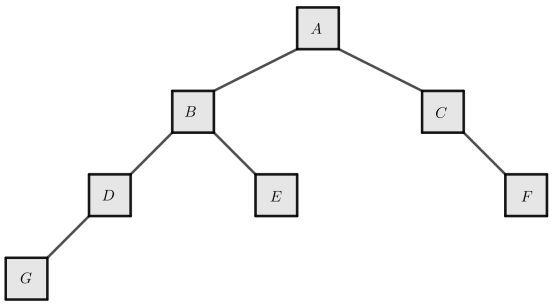
2.1 Parcours en profondeur d'abord

Pour les parcours en profondeur d'abord, en anglais on les appelle DFS (Depth-First Search), le principe est d'explorer l'arbre de la racine à chaque feuille et ce de manière récursive.

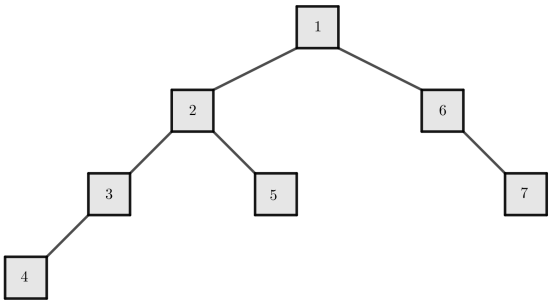
Il y a différentes manières de parcourir un arbre en profondeur selon l'ordre dans lequel on affiche les nœuds parcourus :

- ▶ Si la racine est traitée avant ses deux sous arbres, on parle d'un **ordre préfixe**.
- ▶ Si la racine est traitée entre ses deux sous arbres, on parle d'un **ordre infixe**.
- ▶ Si la racine est traitée après ses deux sous arbres, on parle d'un **ordre suffixe**.

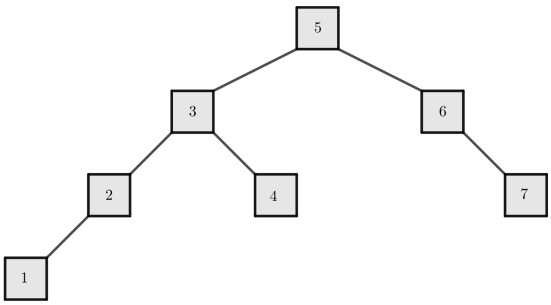
Ainsi si l'on considère l'arbre suivant, on a indiqué ci-dessous l'ordre dans lequel on visitera chaque nœud en fonction de l'ordre choisi.



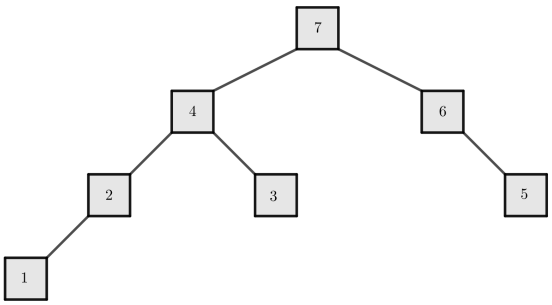
Arbre



Ordre préfixe



Ordre infixe



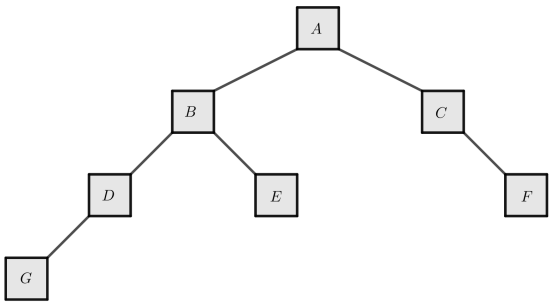
Ordre suffixe

Ainsi on peut insérer la méthode suivante qui décrit le parcours infixe dans la classe `Nœud` :

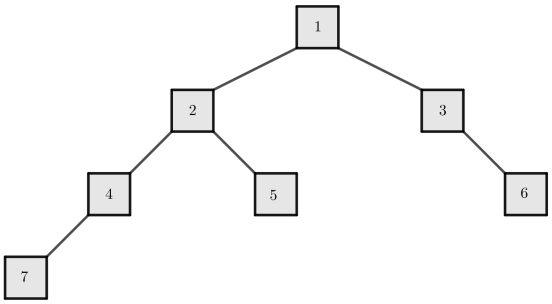
```
def dfs_infixe(self):
  if self != none:
    dfs_infixe(self.gauche)
    print(self.valeur)
    dfs_infixe(self.droite)
```

2.2 Parcours en largeur d'abord

Le parcours en largeur d'abord, BFS en anglais, pour (Breadth-First Search) d'un arbre est le plus simple. On balaye les nœuds d'un même niveau, avant de passer au niveau suivant.



Arbre



Parcours BFS

Dans cet algorithme, contrairement au parcours en profondeur, on n'utilisera pas de fonction récursive. Afin de mémoriser les nœuds déjà visités, on utilisera une file (FIFO). Voici l'algorithme décrit en langage naturel :

```
Fonction parcours_largeur(racine)
    file : file de type FIFO
    ajouter racine a file
    tant que file est non vide :
        defiler le noeud de debut de file
        Afficher le noeud
        Ajouter le noeud de gauche puis de droite a la file.
```

3 Arbres binaires de recherche ABR

3.1 Définition

Nous allons définir ici la notion d'arbre binaire de recherche, on suppose donc que l'on dispose d'un "outil" pour comparer 2 à 2 les valeurs indiquées dans un tel arbre : en mathématiques on appelle cela une *relation d'ordre sur l'ensemble des valeurs*. Ainsi si les valeurs sont des chaines de caractères, Python les comparera avec l'ordre alphabétique et pour les nombres de type int ou float on disposera de l'ordre usuel.

Définition :
Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout nœud :
- les enfants à gauche d'un nœud ont des valeurs inférieures à lui ;
- les enfants à droite d'un nœud ont des valeurs supérieures à lui.

On suppose que les valeurs sont distinctes

L'arbre binaire $n°1$ présenté plus haut est un ABR, en revanche pas les $n°2$ et 3.

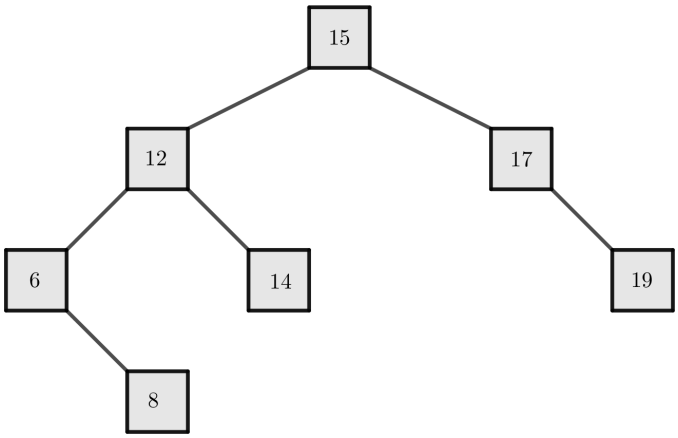
3.2 Implémentation en Python

Voici un exemple de classe Python pour définir un ABR. Chaque objet dispose de trois attributs, `valeur`, `gauche` et `droit`, en outre nous avons implémenter les méthodes `get_valeur()`, `get_gauche()` et `get_droit()` déjà présentées.
De plus pour insérer une nouveau nœud dans l'ABR on a codé une méthode `inserer(valeur)`. On compare `valeur` avec la valeur de la racine, puis avec celle de chaque nœud rencontré ; si cette valeur est inférieure, on continue en descendant à gauche, sinon en descendant à droite. Le dernier nœud rencontré est le parent du nouveau nœud.

Vous noterez :
- l'importation du module `représentation_arbre` qui permet de construire une représentation de l'arbre binaire construit ;
- dans la partie test, on construit l'ABR rerpésenté ci-dessous dans la figure de droite.
- dans la partie test, l'usage d'une liste et d'une boucle `for` pour construire l'ABR à l'aide la fonction `insérer()`.

```python
from representation_arbre import *

class ABR:
    def __init__(self, valeur):
        self.valeur = valeur
        self.gauche = None
        self.droit = None

    def get_valeur(self):
        return self.valeur

    def get_gauche(self):
        return self.gauche

    def get_droit(self):
        return self.droit

    def inserer(self, valeur):
        if valeur < self.valeur:
            if self.gauche == None:
                self.gauche = ABR(valeur)
            else:
                self.gauche.inserer(valeur)
        else:
            if self.droit == None:
                self.droit = ABR(valeur)
            else:
                self.droit.inserer(valeur)

if __name__ == '__main__':
    a = ABR(15)
    liste = [12, 17, 6, 8, 19, 14]
    for elt in liste:
        a.inserer(elt)
    arbre = repr_graph(a)
```

classe_ABR.py



Un ABR

Il est important de comprendre que l'ordre dans lequel on insère les valeurs dans l'ABR est essentiel, dans l'exemple précédent on construit un ABR de hauteur 5, alors que les instructions suivantes conduiraient à un ABR de hauteur 7 (contenant les mêmes valeurs !) :

```
a =  ABR(6)
liste = [8, 12, 14, 15, 17, 19]
for elt in liste:
  a.inserer(elt)
print(affiche(a))
```

4 Algorithmes sur les arbres binaires de recherche ABR

4.1 Recherche dans un ABR

Pour la recherche d'une clé, autrement dit d'une valeur dans un ABR, nous précéderons comme pour la méthode `inserer()` :

```
class ABR:
  ...
  def recherche(self, cle):
        if cle < self.valeur:
            if self.gauche is None:
                return False
            else:
                return self.gauche.recherche(cle)
        elif cle > self.valeur:
            if self.droit is None:
                return False
            else:
                return self.droit.recherche(cle)
        return True
```

Avec l'arbre a défini plus haut, on obtiendrait :

```
>>> print(a.recherche(19))
True
>>>  print(a.recherche(18))
False
```

Le principe utilisé est le même que celui de l'algorithme de recherche dichotomique. suivant la valeur de la clé on poursuit la recherche dans le sous arbre gauche ou dans le sous arbre droit.
Toutefois, comme déjà dit, si l'on insère dans l'arbre `a = ABR(6)` dans cet ordre les valeurs `8, 12, 14, 15, 17, 19` on créé un arbre de hauteur 7, alors qu'en insérant ans l'arbre `b = ABR(15)` dans cet ordre les valeurs `12, 17, 6, 8, 19, 14`, on créé un arbre de hauteur 4. Il existe des méthodes pour équilibrer les arbres si nécessaire, mais on peut retenir :

Propriété :
Si un ABR est équilibré et possède `n` nœuds, alors la recherche d'une valeur dans cet ABR a un coup de l'ordre de $\log_2(n)$, c'est à dire de l'ordre du nombre de chiffres utilisés dans l'écriture binaire de `n`

4.2 Ajout dans un ABR

L'insertion dans un ABR a été présentée initialement avec la classe ABR.
De manière générale, comme pour la recherche d'une clé, l'insertion d'un élément dans un ABR de hauteur `h` ayant `n` nœuds est de l'ordre de `h` et de $\log_2(n)$.

5 Exercices

Exercice n° 1.

Dessiner tous les arbres binaires ayant 3 ou 4 nœuds.

Exercice n° 2.

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé. De plus, un arbre binaire de recherche limité à un nœud a une hauteur de 1. On considère l'arbre binaire de recherche représenté ci-dessous (figure 1), où val représente un entier :

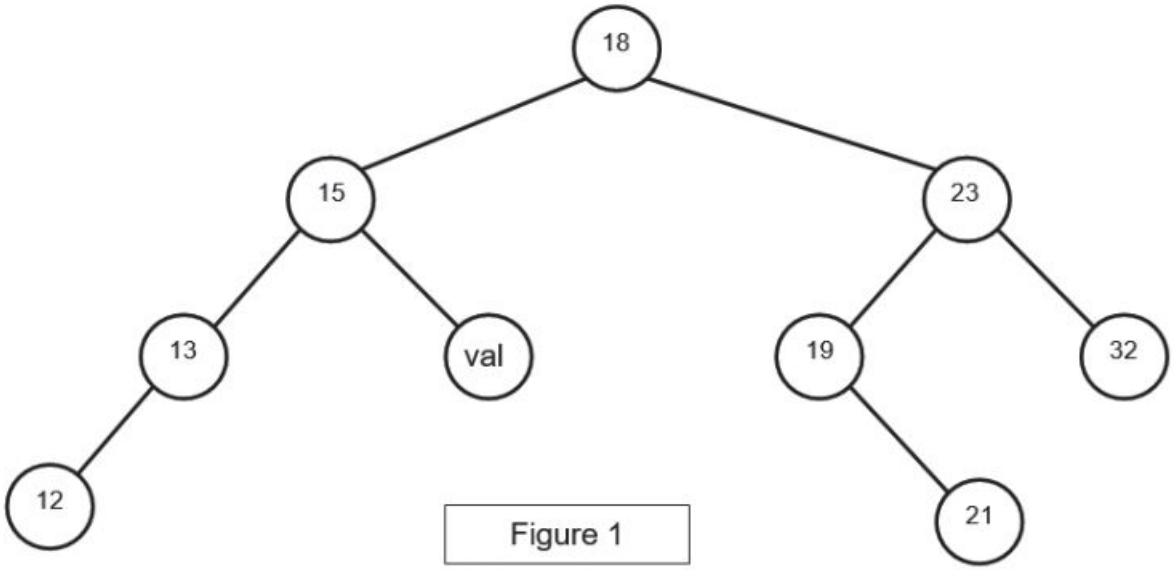


Figure 1

1. Donner le nombre de feuilles de cet arbre et préciser leur valeur (étiquette).
2. Donner le sous arbre-gauche du nœud 23.
3. Donner la hauteur et la taille de l'arbre.
4. Donner les valeurs entières possibles de val pour cet arbre binaire de recherche
   On suppose, pour la suite de cet exercice, que val est égal à 16
   Rappel :
   - Le parcours en profondeur infixe d'un arbre binaire consiste à parcourir son sous-arbre gauche, puis sa racine, puis son sous-arbre droit. - Le parcours suffixe consiste à parcourir l'arbre suivant l'ordre : fils g.→ fils d. → racine
5. Donner les valeurs d'affichage des nœuds dans le cas du parcours infixe de l'arbre.
6. Donner les valeurs d'affichage des nœuds dans le cas du parcours suffixe de l'arbre

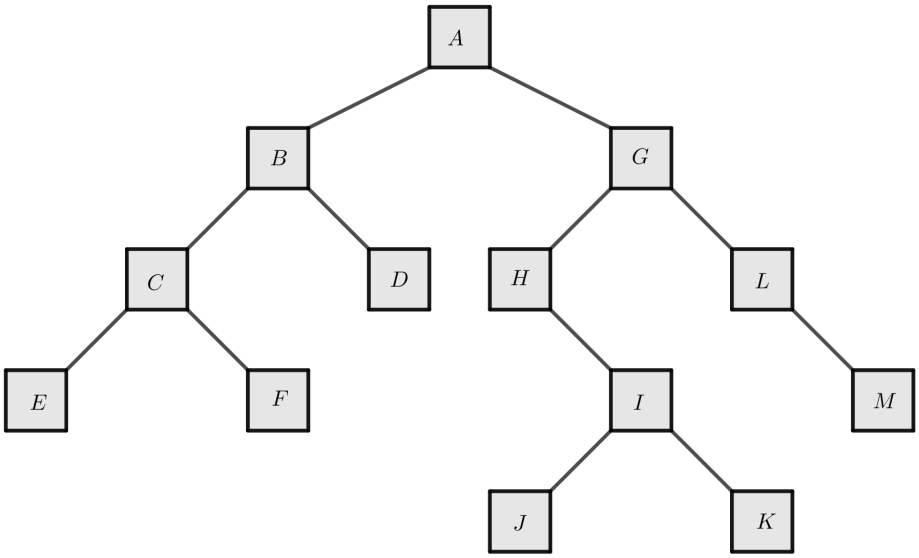
Exercice n° 3.

Ajouter à la classe `Noeud` une méthode `__eq__` qui permet de tester l'égalité entre deux arbres binaires à l'aide de l'opération `==`.

Exercice n° 4.

Écrire une fonction récursive `parfait(h)` qui reçoit en argument un entier naturel `h` et qui renvoie un arbre binaire complet quelconque de hauteur `h`.
Vous noterez que pour `h = 0`, votre fonction renverra l'arbre vide qui est complet.

L'arbre ci-dessous servira d'exemple, pour les exercice 5, 6 et 7 :



Exercice n° 5.

1. Quelle est la taille de l'arbre ci-dessus ?
2. Quelle est la taille de son sous-arbre gauche ? de son sous-arbre droit ?
3. Quelle relation y a t-il entre la taille d'un arbre et la taille de ses sous-arbres gauche et droit ?
4. En déduire un algorithme récursif permettant de calculer la taille d'un arbre binaire.
5. Implémenter votre algorithme en Python en créant une fonction `taille(a)` prenant en argument un arbre binaire `a` et renvoyant sa taille.

Exercice n° 6.

1. Quelle est la hauteur de l'arbre de l'exercice précédent ?
2. Quelle est la hauteur de son sous-arbre gauche ? de son sous-arbre droit ?
3. Quelle relation y a t-il entre la hauteur d'un arbre et la hauteur de ses sous-arbres gauche et droit ?
4. En déduire un algorithme récursif permettant de calculer la hauteur d'un arbre binaire.
5. Implémenter votre algorithme en Python en créant une fonction `hauteur(a)` prenant en argument un arbre binaire `a` et renvoyant sa hauteur.

Exercice n° 7.

1. Si l'on parcourt en profondeur d'abord l'arbre ci-dessus dans l'ordre préfixe, indiquer l'ordre dans lequel les nœuds sont visités.
2. Si l'on parcourt en profondeur d'abord l'arbre ci-dessus dans l'ordre infixe, indiquer l'ordre dans lequel les nœuds sont visités.
3. Si l'on parcourt en profondeur d'abord l'arbre ci-dessus dans l'ordre suffixe, indiquer l'ordre dans lequel les nœuds sont visités.
4. Si l'on parcourt en largeur d'abord l'arbre ci-dessus, indiquer l'ordre dans lequel les nœuds sont visités.

Exercice n° 8.

1. Écrire une fonction `peigne_gauche(h)` qui reçoit en argument un entier naturel `h` et qui renvoie un arbre binaire de hauteur `h` tel que chaque nœud a un sous arbre droit vide.
2. Écrire une fonction `est_peigne_gauche(a)` qui reçoit en argument un arbre binaire `a` et qui renvoie `True` si et seulement si chaque nœud de `a` a un sous arbre droit vide.

Exercice n° 9.

1. Écrire une méthode itérative de recherche d'une clé dans un ABR.
2. Ajouter deux méthodes `min` et `max` à la la classe `ABR` qui renvoie respectivement la valeur minimale et maximale présente dans l'ABR.

Exercice n° 10.

On reprend la classe `ABR` du cours en ajoutant un nouvel attribut `parent` :

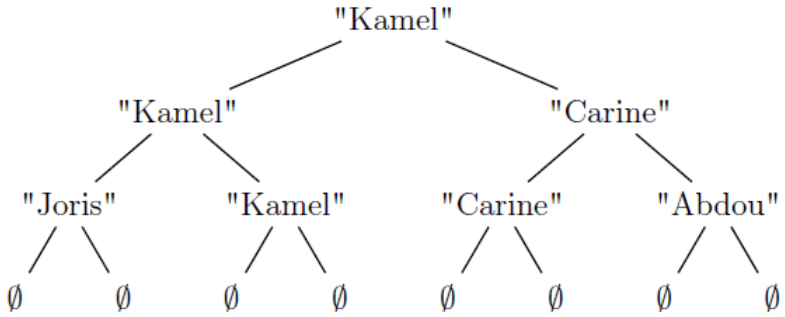
```
class ABR:
    def __init__(self, valeur):
        self.valeur = valeur
        self.gauche = None
        self.droit = None
        self.parent = None
```

1. Modifier la méthode `insere()` en conséquence.
2. Un nœud est un enfant gauche s'il a un parent et qu'il est la racine du sous arbre gauche de son parent. Écrire une méthode `enfant_gauche` qui renvoie `True` si le nœud concerné est un enfant gauche et `False` sinon.
   Écrire de même une méthode `enfant_droit`.
   Les valeurs contenues dans un ABR peuvent être rangées dans un ordre croissant. Toute valeur autre que le minimum et le maximum a donc une valeur qui la précède et une valeur qui lui succède.
3. Écrire une méthode `successeur` qui renvoie le successeur d'un nœud.
   On utilisera la méthode `min` de l'exercice précédent.

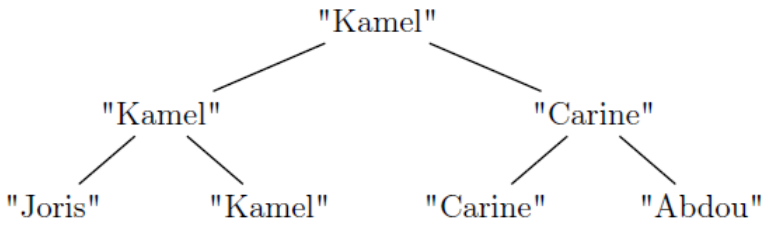
4. Écrire une méthode `predecesseur` qui renvoie le prédécesseur d'un nœud.
   On utilisera la méthode `max` de l'exercice précédent.

Exercice n° 11.

La fédération de badminton souhaite gérer ses compétitions à l'aide d'un logiciel.
Pour ce faire, une structure arbre de compétition a été définie récursivement de la façon suivante : un arbre de compétition est soit l'arbre vide, noté ∅, soit un triplet composé d'une chaîne de caractères appelée valeur, d'un arbre de compétition appelé sous-arbre gauche et d'un arbre de compétition appelé sous-arbre droit.
On représente graphiquement un arbre de compétition de la façon suivante :

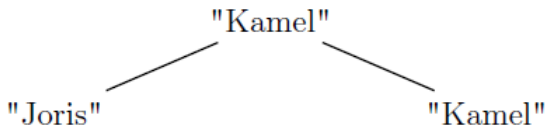


Pour alléger la représentation d'un arbre de compétition, on ne notera pas les arbres vides, l'arbre précédent sera donc représenté par l'arbre A suivant :

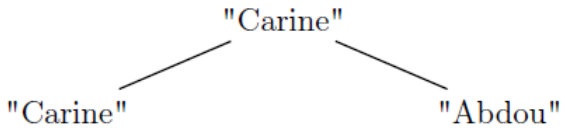


Cet arbre se lit de la façon suivante :
- 4 participants se sont affrontés : Joris, Kamel, Carine et Abdou. Leurs noms apparaissent en bas de l'arbre, ce sont les valeurs de feuilles de l'arbre.
- Au premier tour, Kamel a battu Joris et Carine a battu Abdou.
- En finale, Kamel a battu Carine, il est donc le vainqueur de la compétition.

Pour s'assurer que chaque finaliste ait joué le même nombre de matchs, un arbre de compétition a toutes ces feuilles à la même hauteur.
Les quatre fonctions suivantes pourront être utilisées :
- La fonction `racine` qui prend en paramètre un arbre de compétition `arb` et renvoie la valeur de la racine.
  Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, `racine(A)` vaut "Kamel".
- La fonction `gauche` qui prend en paramètre un arbre de compétition `arb` et renvoie son sousarbre gauche.
  Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, `gauche(A)` vaut l'arbre représenté graphiquement ci-après :



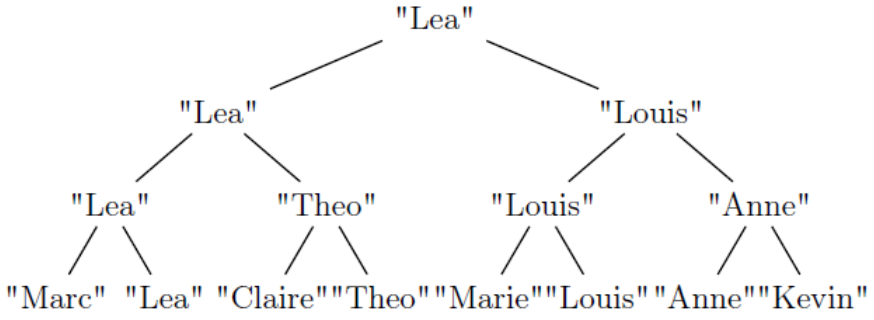
- La fonction `droit` qui prend en argument un arbre de compétition `arb` et renvoie son sous-arbre droit.
  Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, `droit(A)` vaut l'arbre représenté graphiquement ci-dessous :



– La fonction `est_vide` qui prend en argument un arbre de compétition et renvoie `True` si l'arbre est vide et `False` sinon.
Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, `est_vide(A)` vaut `False`.

Pour toutes les questions de l'exercice, on suppose que tous les joueurs d'une même compétition ont un prénom différent.

1. On considère l'arbre de compétition B suivant :



(a) Indiquer la racine de cet arbre puis donner l'ensemble des valeurs des feuilles de cet arbre.

(b) Proposer une fonction Python `vainqueur` prenant pour argument un arbre de compétition `arb` ayant au moins un joueur.
Cette fonction doit renvoyer la chaîne de caractères constituée du nom du vainqueur du tournoi.
Exemple : `vainqueur(B)` vaut "Lea"

(c) Proposer une fonction Python `finale` prenant pour argument un arbre de compétition `arb` ayant au moins deux joueurs. Cette fonction doit renvoyer le tableau des deux chaînes de caractères qui sont les deux compétiteurs finalistes.
Exemple : `finale(B)` vaut `["Lea", "Louis"]`

2. (a) Proposer une fonction Python `occurrences` ayant pour paramètre un arbre de compétition `arb` et le nom d'un joueur `nom` et qui renvoie le nombre d'occurrences (d'apparitions) du joueur `nom` dans l'arbre de compétition `arb`.
Exemple : `occurences(B, "Anne")` vaut 2.

(b) Proposer une fonction Python `a_gagne` prenant pour paramètres un arbre de compétition `arb` et le nom d'un joueur `nom` et qui renvoie le booléen `True` si le joueur `nom` a gagné au moins un match dans la compétition représenté par l'arbre de compétition `arb`.

3. On souhaite programmer une fonction Python `nombre_matchs` qui prend pour arguments un arbre de compétition `arb` et le nom d'un joueur `nom` et qui renvoie le nombre de matchs joués par le joueur `nom` dans la compétition représentée par l'arbre de compétition `arb`.
Exemple : `nombre_matchs(B, "Lea")` doit valoir 3 et `nombre_matchs(B, "Marc")` doit valoir 1.

(a) Expliquer pourquoi les instructions suivantes renvoient une valeur erronée. On pourra pour cela identifier le nœud de l'arbre qui provoque une erreur.

```
1 def nombre_matchs(arb, nom ):
2   """ arbre_competition, str -> int """
3   return occurrences(arb, nom )
```

(b) proposer une correction pour la fonction `nombre_matchs`.

4. Recopier et compléter la fonction `liste_joueurs` qui prend pour argument un arbre de compétition `arb` et qui renvoie un tableau contenant les participants au tournoi, chaque nom ne devant figurer qu'une seule fois dans le tableau.
L'opération `+` à la ligne 8 permet de concaténer deux tableaux.
Exemple : Si `L1 = [4, 6, 2]` et L2 = `[3, 5, 1]`, l'instruction `L1 + L2` va renvoyer le tableau `[4, 6, 2, 3, 5, 1]`.

```
1 def liste_joueurs(arb):
2   """ arbre_competition -> tableau """
3   if est_vide(arb):
```

```
4       return ..............................
5     elif ................................ and ................................ :
6       return [racin (ar )]
7     else :
8       return ................................ + liste_joueurs (droit(arb))
```