Arbres et structure de données

Introduction :

Les arbres sont des structures de données hiérarchiques. Les cas d'usage des arbres peuvent être très variés, et de nombreux utilisateur·rice·s des outils numériques interagissent avec des arbres sans nécessairement en avoir conscience.
Nous allons, dans une première partie, caractériser ces structures, en illustrer différents usages, et porter une attention plus particulière aux arbres binaires, sur lesquels nous étudierons des implémentations algorithmiques dans les parties suivantes. Nous effectuerons, en deuxième partie, des mesures et parcours sur ces arbres binaires, et nous implémenterons en troisième partie des recherches et ajouts de valeurs dans des arbres binaires de recherche.

1 Caractérisation des arbres

Un **arbre** est une structure de données liant entre eux des **nœuds** par l'intermédiaire d'**arêtes** formant des **branches**. Contrairement aux graphes , qui font l'objet d'un cours séparé dans ce chapitre, l'organisation des nœuds d'un arbre comporte une dimension **hiérarchique**.

Définition

Arbre :

Un arbre est une structure de données composée de nœuds reliés entre eux par des branches, selon une organisation hiérarchique, à partir d'un nœud racine.

Commençons par présenter les différents éléments constitutifs d'un arbre.

a. Éléments constitutifs d'un arbre

Un arbre est constitué d'un élément de base : son **nœud racine**.
De ce nœud racine peuvent ensuite partir des arêtes reliant d'autres nœuds, lesquels peuvent eux-mêmes relier d'autres nœuds pour former des branches.

- Dans le sens descendant depuis la racine, les nœuds reliés aux niveaux supérieurs sont appelés **nœuds enfants** ou **nœuds fils** du nœud dont ils dépendent.
- Inversement un nœud enfant est rattaché dans le sens ascendant à un **nœud père** ou **nœud parent**.

Astuce

> Seul le nœud racine n'a pas de père.

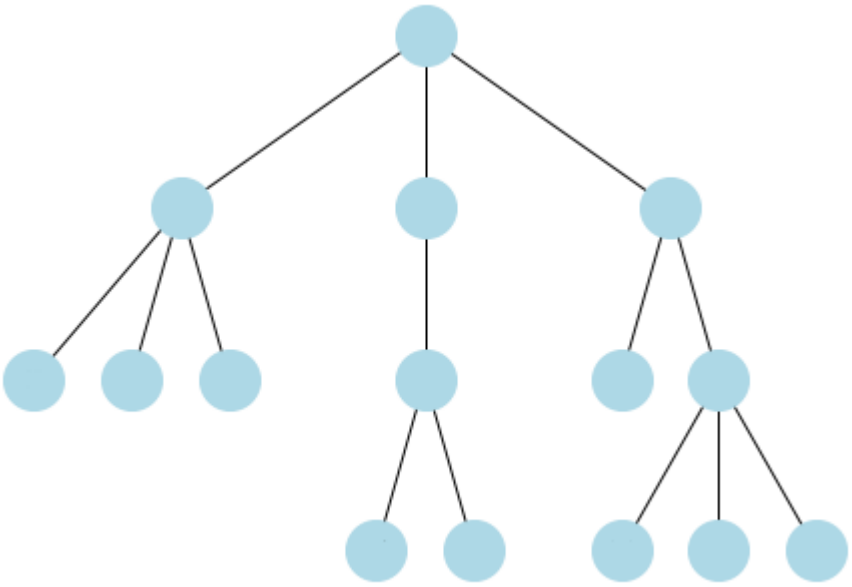
Un nœud n'ayant pas de nœuds enfants est dit **nœud terminal**. Situé à l'extrémité d'une branche, il est également appelé « feuille ».

Chaque nœud peut stocker une information, appelée **valeur** ou **clé** du nœud.

Un **sous-arbre** est une portion d'arbre à partir d'un nœud quelconque qui constitue la racine de ce sous-arbre.

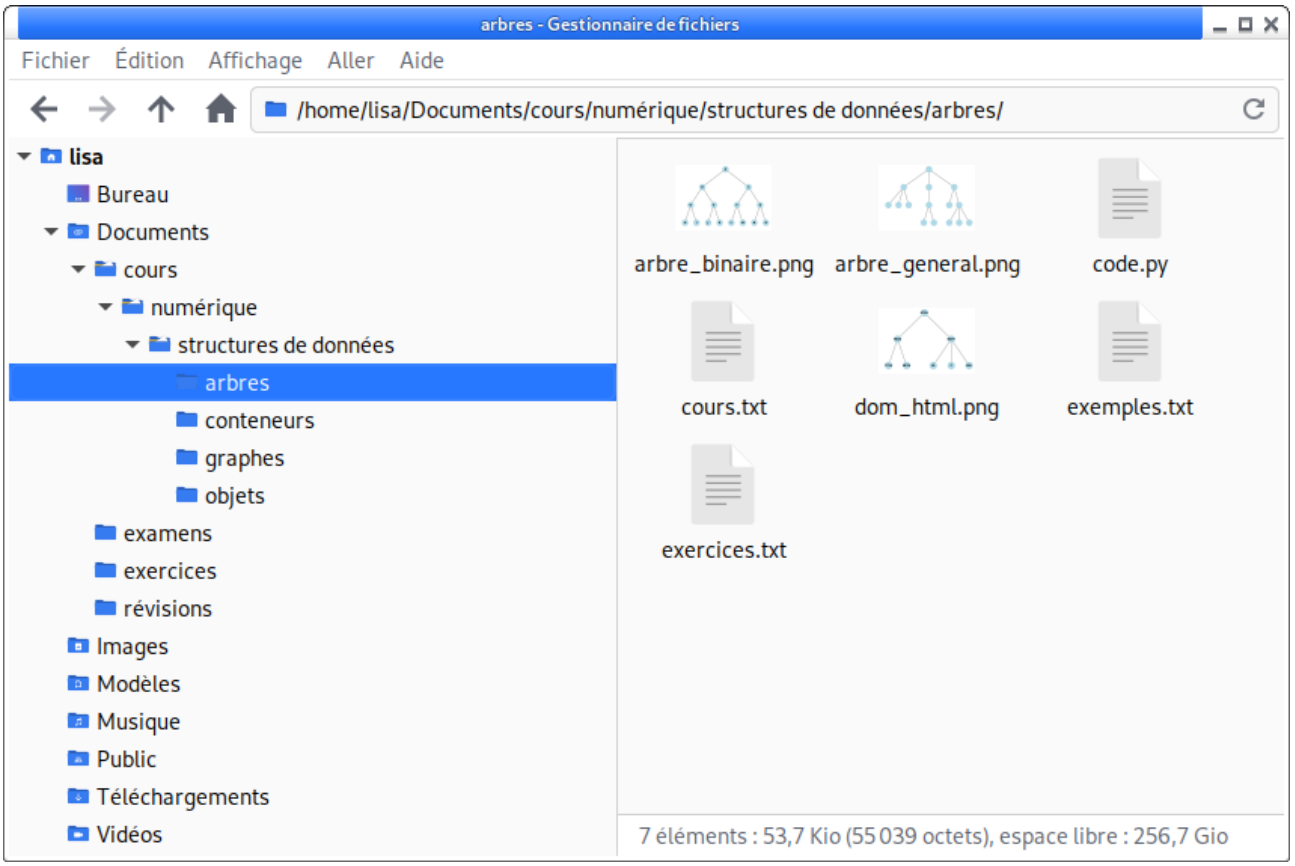
→ Une structure de données organisée comme un arbre est dont dite « arborescente ».

Sa représentation graphique évoque la forme d'un arbre végétal, à ceci près que les arbres informatiques sont généralement représentés avec la racine tout en haut du schéma, pour mieux traduire cette notion de hiérarchie descendante depuis le nœud racine.



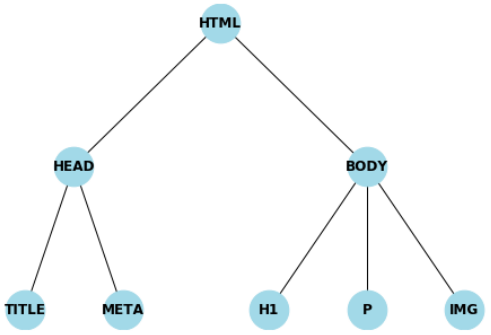
b. Contextes d’emploi des arbres

Les structures de type arbre sont très présentes en informatique. L’organisation du système de fichiers d’un ordinateur est arborescente. Les fichiers sont organisés de manière hiérarchique à partir d’une racine, à laquelle peuvent être rattachés des répertoires (parfois appelés dossiers) et des fichiers, les répertoires pouvant eux-mêmes de manière récursive contenir d’autres répertoires et/ou fichiers.



On trouve également des arbres dans le contexte du web.
Le DOM (*Document Object Model*) ou « modèle objet de document » du langage HTML représente un document web de manière structurée sous forme d'un arbre hiérarchique.

```html
<!DOCTYPE html>
<html lang=fr dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Bienvenue</title>
  </head>
  <body>
    <h1>Bonjour</h1>
    <p>Bienvenue sur notre site internet</p>
    <img src="logo.png" alt="notre_logo">
  </body>
</html>
```



C. Types d'arbres

Il existe différentes sortes d'arbres :

- les **arbres généraux** peuvent posséder un nombre variable de branches ;

- les **arbres binaires** et les **arbres binaires de recherche** sont caractérisés par des propriétés particulières.

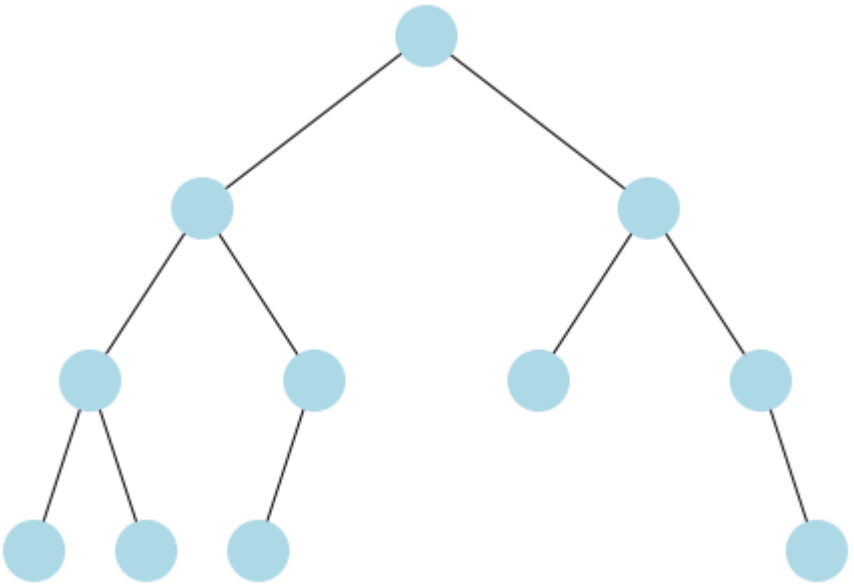
1 Arbre binaire

Définition

> **Arbre binaire :**
>
> Un arbre binaire est un arbre dont chaque nœud comporte au plus deux sous-arbres enfants.

Les sous-arbres d'un arbre binaire sont conventionnellement appelés « sous-arbre gauche » et « sous-arbre droit », ou encore « branche gauche » et « branche droite ». Voire, plus simplement, **gauche** et **droit**.

Chaque nœud de l'arbre binaire comporte au maximum deux sous-arbres.



Astuce

> - Certains nœuds non terminaux peuvent n'avoir qu'une seule branche.

- Les branches d’un arbre ou d’un sous-arbre binaire ne sont pas nécessairement de longueurs égales.

2 Arbre binaire de recherche

Un arbre binaire de recherche est un cas particulier d’arbre binaire qui se distingue par le caractère ordonné des nœuds qui le composent.

Définition

Arbre binaire de recherche :

Un arbre binaire de recherche est un arbre binaire dont les clés sont ordonnées.

Définition

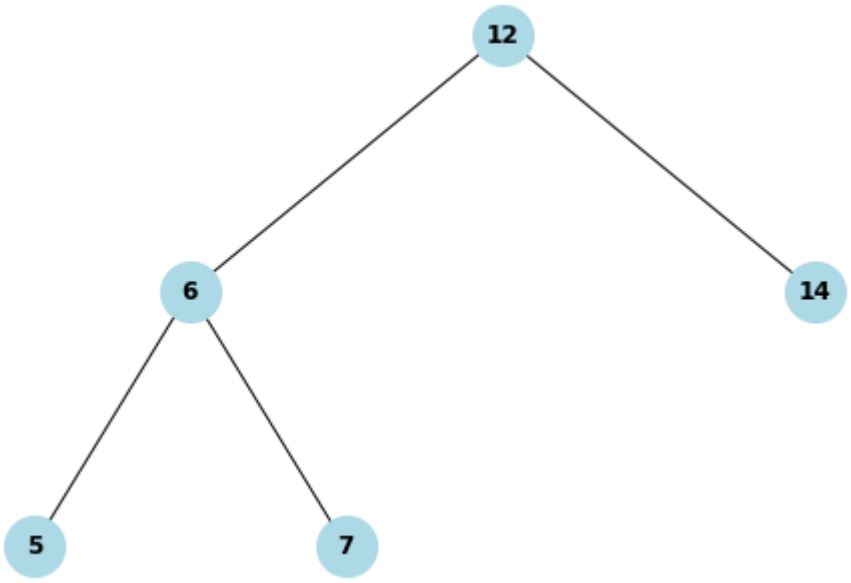
Clé (d’un nœud) :

La clé d'un nœud correspond à la valeur qui lui est affectée.

→ Par exemple, la clé de la racine de l'arbre binaire de recherche donné ci-dessous est 12.

Le placement des clés est effectué de manière à toujours maintenir le caractère ordonné de l’arbre binaire de recherche :

- les valeurs situées ou ajoutées dans le sous-arbre gauche d’un arbre binaire de recherche sont nécessairement inférieures à celle de la clé du nœud considéré ;
- les valeurs situées ou ajoutées dans le sous-arbre droit d’un arbre binaire de recherche sont nécessairement supérieures à celle de la clé du nœud considéré.



Exemple

Pour ajouter 13, on doit nécessairement l'ajouter dans le sous-arbre droit en considérant la racine car $13 > 12$. Dans le sous-arbre droit, 14 étant la racine, 13 sera alors dans le sous-sous-arbre gauche puisque $13 < 14$. Sachant qu'il n'existe pas encore ces sous-sous-arbre gauche, 13 en sera la racine.

d. Mesures sur les arbres

Les arbres peuvent comporter plus ou moins de nœuds et de branches de longueurs variables. On peut mesurer la taille et la hauteur des arbres.

Définition

Taille d'un arbre :

La taille d'un arbre est le nombre de nœuds qu'il comporte (incluant bien sûr le nœud racine).

Exemple

La taille de l'arbre binaire de recherche ci-avant est de 5.

Définition

Hauteur d'un arbre :

La hauteur, parfois appelée « profondeur », correspond au nombre de nœuds parcourus sur le plus long chemin possible depuis la racine de l'arbre, sans revenir sur ses pas.

Exemple

La hauteur de l'arbre binaire de recherche ci-avant est de 3.

Les structures de type arbre peuvent être implémentées de différentes manières. Le paradigme de programmation orientée objet est généralement employé, avec un recours fréquent à l'approche récursive pour les méthodes.
Nos implémentations dans les parties suivantes du cours suivront ces pratiques usuelles, étant précisé qu'il est également possible d'employer le paradigme fonctionnel et des approches itératives.

2 Implémentations sur des arbres binaires

Cette deuxième partie est consacrée aux implémentations sur des arbres binaires.
Après avoir conçu des classes adaptées à ces implémentations, nous calculerons taille et hauteur, avant d'effectuer différents parcours des arbres binaires.

a. Conception des classes

Notre implémentation des arbres binaires repose sur deux classes distinctes :

- une classe modélisant des nœuds ;

- une classe modélisant des arbres binaires à partir des nœuds.

1 Classe `Noeud`

La classe `Noeud` définit des objets composés d'une valeur (la clé du nœud) et de deux branches, la gauche et la droite, identifiées par leur nœud racine .

```
class Noeud:
    def __init__(self, cle):
        self.cle = cle
        self.gauche = None
        self.droite = None
```

À retenir

> Les modalités de création d'un nœud sont précisées dans la définition de la méthode spéciale `__init__` :
>
> - seule la valeur de la clé doit être précisée au moment de la création d'un nœud ;
> - ses sous-branches sont initialisées avec l'objet `None`, qui exprime l'absence de valeur.

La création d'un nœud s'effectue ainsi :

```
noeud = Noeud('A')
```

Affichons les attributs du nœud.

```
print(noeud)
print(noeud.cle)
print(noeud.gauche)
print(noeud.droite)
```

affiche successivement :

```
__main__.Noeud object at 0x10b215278> # (adresse mémoire attribuée dynamiquement à la création de l'objet)
A
None
None
```

Les attributs de l'objet peuvent ensuite être modifiés.

 Exemple

Ajout de deux nœuds enfants aux sous-branches du nœud initial.

```
noeud.gauche = Noeud('B')
noeud.droite = Noeud('C')

print(noeud)
print(noeud.cle)
print(noeud.gauche)
print(noeud.droite)
```

affiche successivement:

```
<__main__.Noeud object at 0x10b215278>
A
<__main__.Noeud object at 0x10b1ec588>
<__main__.Noeud object at 0x10b1ec630>
```

→ Les nœuds B et C sont bien affectés aux deux sous-branches du nœud A.

Notons que pour afficher la valeur des clés gauche et droite, il suffit d'employer les commandes suivantes :

```
print(noeud.gauche.cle)
print(noeud.droite.cle)
```

Ce qui affichera successivement

```
B
C
```

2 Classe `ArbreBinaire`

La classe `ArbreBinaire` définit des objets composés initialement d'un nœud racine, auquel pourront être rattachés des nœuds aux différentes sous-branches.

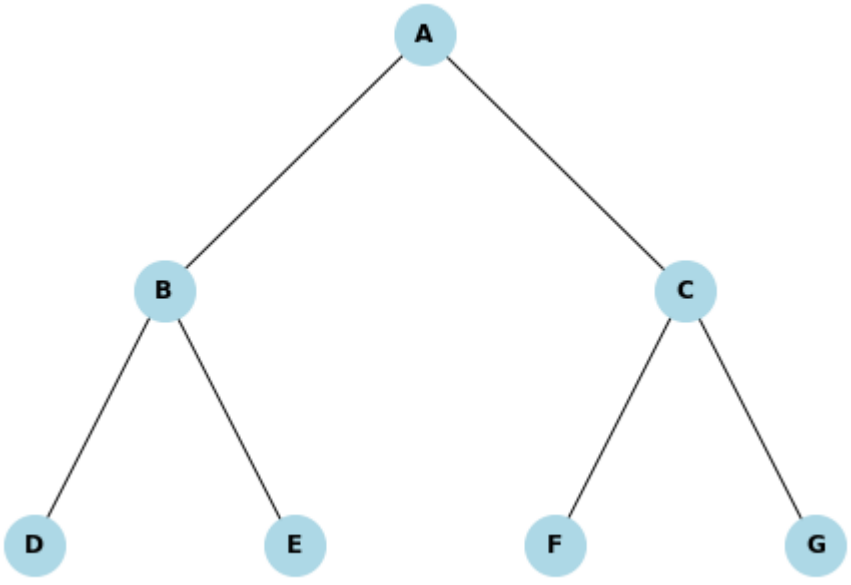
```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)
```

La création d'un arbre binaire s'effectue par initialisation de l'arbre avec un nœud racine.

```
ab = ArbreBinaire('A')
```

On peut ensuite ajouter des nœuds aux branches du nœud racine, puis aux nœuds qui y sont rattachés, sans limitation de profondeur.

```
ab.racine.gauche = Noeud('B')
ab.racine.droite = Noeud('C')

ab.racine.gauche.gauche = Noeud('D')
ab.racine.gauche.droite = Noeud('E')

ab.racine.droite.gauche = Noeud('F')
ab.racine.droite.droite = Noeud('G')
```



Une implémentation plus complète pourrait intégrer la possibilité d'ajout, de modification et de suppression de nœuds de l'arbre par des méthodes dédiées. Nous implémenterons l'ajout d'un élément sur un arbre binaire de recherche dans la troisième partie du cours.
Dans cette partie, nous nous intéressons en priorité aux mesures de taille et de profondeur, ainsi qu'aux modalités de parcours des arbres binaires.

b. Taille d'un arbre binaire

Déterminer la taille d'un arbre revient à compter l'ensemble des nœuds composant cet arbre. Nous effectuons ce compte à partir du nœud racine, en explorant récursivement tous les sous-arbres de chacun des nœuds.

```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)

    def taille(self, noeud='racine'):
        if noeud == 'racine':
            noeud = self.racine
        if not noeud:
            return 0
        else:
            return 1 + self.taille(noeud.gauche) + self.taille(noeud.droite)
```

À retenir

> La méthode `taille` est définie avec comme paramètres :
>
> - `self` qui désigne l'instance de la classe ;
> - le nœud à traiter.

L'appel initial est effectué avec une valeur par défaut correspondant au nœud racine de l'arbre. Les appels suivants sont effectués de manière récursive sur les sous-arbres, dont le nombre de nœuds est ajouté au nœud courant. La récursion s'arrête en l'absence de nœud en sous-arbre.

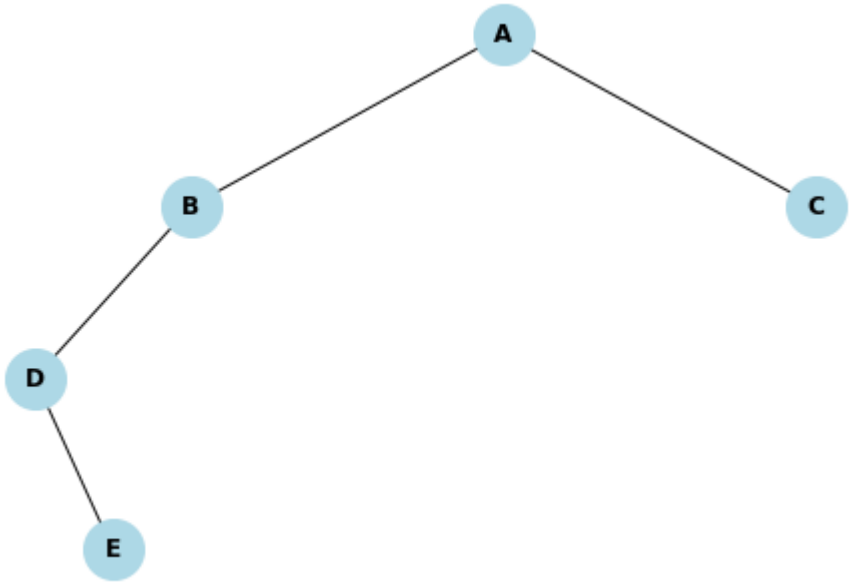
Exemple

Appel de la méthode sur l'arbre binaire nommé `ab` :

```
print(ab.taille())
# affiche 7
```

Vérifions que notre méthode de détermination de taille fonctionne également sur un arbre binaire dont toutes les branches ne sont pas pareillement développées.

```
# arbre binaire incomplet
abi = ArbreBinaire('A')
abi.racine.gauche = Noeud('B')
abi.racine.droite = Noeud('C')
abi.racine.gauche.gauche = Noeud('D')
abi.racine.gauche.gauche.droite = Noeud('E')
```




```
print(abi.taille())
# affiche 5
```

Implémentons maintenant une méthode déterminant la hauteur d'un arbre binaire.

c. Hauteur d'un arbre binaire

Nous employons la même approche récursive que pour le calcul de la taille.

```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)

    def hauteur(self, noeud='racine'):
        if noeud == 'racine':
            noeud = self.racine
        if not noeud:
            return 0
        else:
            return 1 + max(self.hauteur(noeud.gauche), self.hauteur(noeud.droite))
```

À retenir

> Comme précédemment, la méthode `hauteur` est définie avec comme paramètres :
>
> - `self` qui désigne l'instance de la classe ;
> - le nœud à traiter.

Les retours des appels récursifs sur les sous-branches gauches et droites sont passés en arguments à la fonction `max`, laquelle retourne la plus grande des valeurs fournies, de manière à pouvoir déterminer la hauteur de tout type d'arbre binaire.

Exemple

> Appel de la méthode `hauteur` sur les deux arbres binaires précédents `ab` et `abi` :
>
> ```
> print(ab.hauteur())
> # affiche 3
> ```

```
print(abi.hauteur())
# affiche 4
```

La hauteur est correctement déterminée pour les deux arbres.

→ L'arbre `ab` présente la particularité que toutes ses branches sont de longueur égale.

→ Les sous-branches de l'arbre `abi` sont en revanche de longueurs inégales, mais c'est bien la plus grande des branches qui est prise en compte.

Intéressons-nous maintenant aux parcours des arbres binaires.

d. Parcours des arbres binaires

Définition

Parcours d'un arbre :

Parcourir un arbre consiste à explorer l'ensemble de ses nœuds.

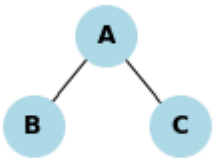
Il existe différentes méthodes pour effectuer cette exploration. Nous implémentons ici trois modes de parcours des arbres binaires :

- le parcours **préfixe** ;
- le parcours **infixe** ;
- le parcours **suffixe**.

Les trois modes se distinguent par l'ordre dans lequel le parcours est effectué sur les nœuds et leurs branches. Nous l'illustrons avec l'exemple ci-après :

Exemple

Considérons l'arbre suivant :



- Le parcours préfixe consiste à parcourir d'abord le nœud racine, puis le nœud gauche, et ensuite le nœud droit. Le parcours préfixe de l'arbre s'effectue dans l'ordre A → B → C.
- Le parcours infixe consiste à parcourir d'abord le nœud gauche, puis le nœud racine, et ensuite le nœud droit. Le parcours infixe de l'arbre s'effectue dans l'ordre B → A → C.

Le parcours suffixe consiste à parcourir d'abord le nœud gauche, puis, le nœud droit et ensuite le nœud racine. Le parcours suffixe de l'arbre s'effectue dans l'ordre B → C → A.

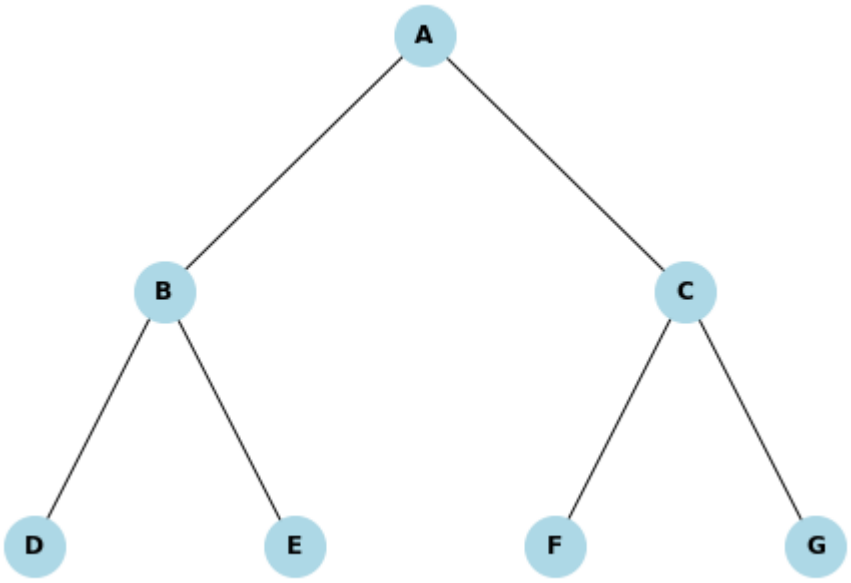
Ces modes de parcours s'appliquent récursivement.

1 Parcours préfixe

Commençons par implémenter le parcours préfixe.

```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)

    def parcours_prefixe(self, position='racine', parcours=None):
        if parcours is None:
            parcours = []
        if position == 'racine':
            position = self.racine

        if position:
            parcours += position.cle
            parcours = self.parcours_prefixe(position.gauche, parcours)
            parcours = self.parcours_prefixe(position.droite, parcours)
        return parcours
```

Le parcours est enregistré dans une liste qui est complétée au fur à mesure des appels récursifs du parcours.
L'appel de la méthode sur le nom de l'arbre retourne la liste des nœuds dans l'ordre dans lequel ils ont été visités.



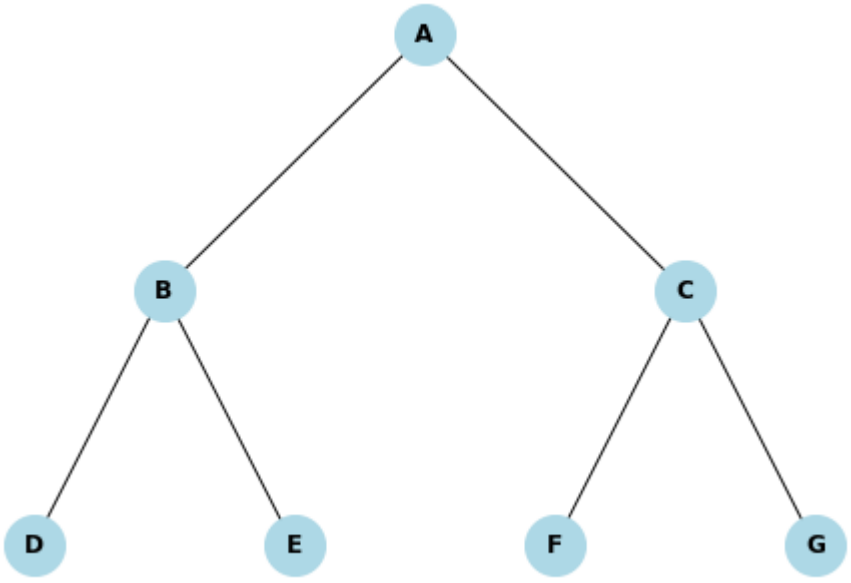

```
print(ab.parcours_prefixe())
# affiche ['A', 'B', 'D', 'E', 'C', 'F', 'G']
```

2 Parcours infixe

Implémentons maintenant la méthode de parcours infixe.

```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)

    def parcours_infixe(self, position='racine', parcours=None):
        if parcours is None:
            parcours = []
        if position == 'racine':
            position = self.racine

        if position:
            parcours = self.parcours_infixe(position.gauche, parcours)
            parcours += position.cle
```

```
        parcours = self.parcours_infixe(position.droite, parcours)
    return parcours
```

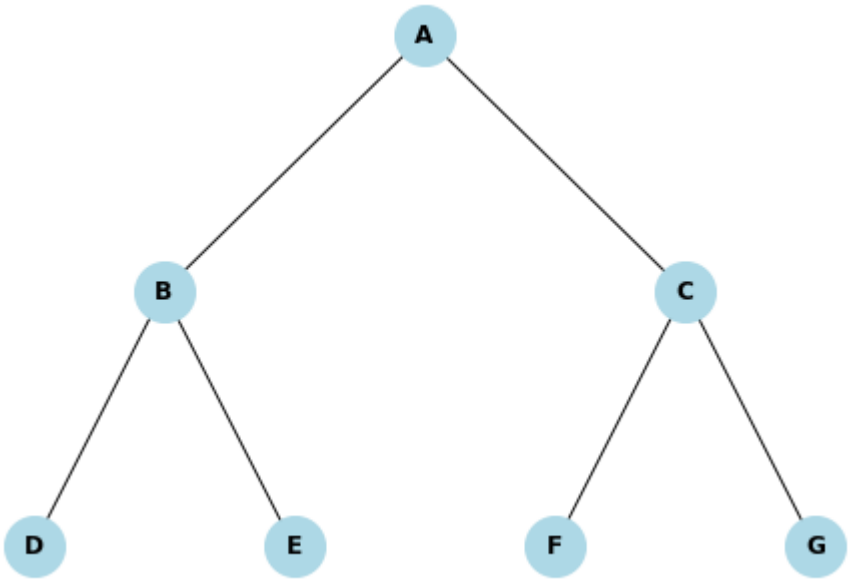



```
print(ab.parcours_infixe())
# affiche ['D', 'B', 'E', 'A', 'F', 'C', 'G']
```

3 Parcours suffixe

Implémentons enfin la méthode de parcours suffixe.

```
class ArbreBinaire:
    def __init__(self, racine):
        self.racine = Noeud(racine)

    def parcours_suffixe(self, position='racine', parcours=None):
        if parcours is None:
            parcours = []
        if position == 'racine':
            position = self.racine

        if position:
            parcours = self.parcours_suffixe(position.gauche, parcours)
            parcours = self.parcours_suffixe(position.droite, parcours)
            parcours += position.cle
        return parcours
```




```
print(ab.parcours_suffixe())
# affiche ['D', 'E', 'B', 'F', 'G', 'C', 'A']
```

Nous observons que les méthodes des trois parcours reposent sur la même logique d'appels récursifs des différentes branches, dans un ordre spécifique à chaque parcours.

Passons maintenant à des implémentations concernant plus spécifiquement les arbres binaires de recherche.

3 Implémentations sur les arbres binaires de recherche

Cette troisième partie est consacrée à des implémentations sur des arbres binaires de recherche ; les clés de leurs nœuds ont la particularité d'être ordonnées. Nous allons tirer parti de cette spécificité dans notre implémentation de fonctionnalités de recherche et d'insertion.

a. Conception des classes

Notre implémentation des arbres binaires de recherche repose sur deux classes distinctes :

- une classe modélisant des nœuds ;
- une classe modélisant des arbres binaires de recherche à partir des nœuds.

Astuce

> Cette implémentation s'appuie sur la même classe de nœuds que celle utilisée en deuxième partie, et sur une classe spécifique aux arbres binaires de recherche.
> Nous nous abstenons volontairement de recourir à l'héritage entre classes afin de permettre l'étude des arbres binaires et des arbres binaires de recherche de manière indépendante.

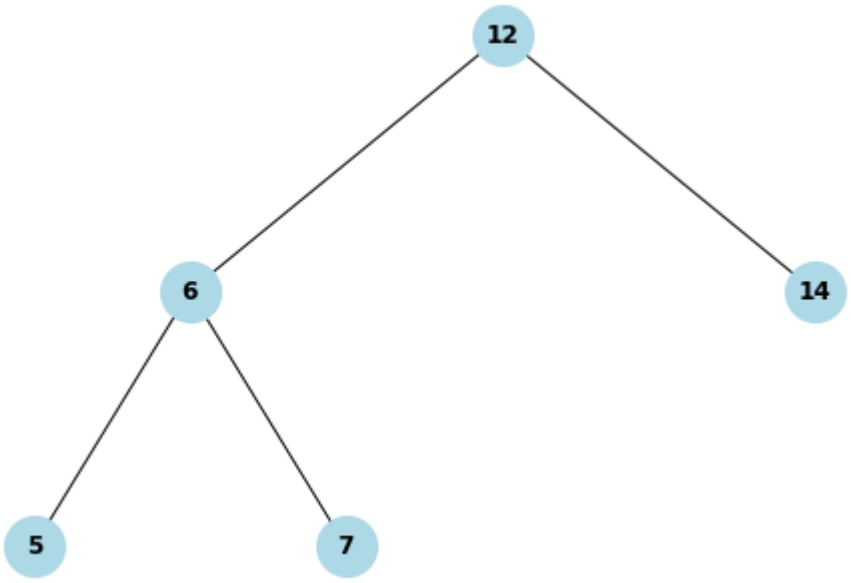
1 Classe `Noeud`

```
class Noeud:
    def __init__(self, cle):
        self.cle = cle
        self.gauche = None
        self.droite = None

class ArbreBinaireRecherche:
    def __init__(self, racine):
        self.racine = Noeud(racine)
```

Nous utilisons cette base pour instancier un arbre binaire de recherche. Nous ajoutons quelques valeurs en veillant à respecter le caractère ordonné de l'arbre. Nous utiliserons cet arbre pour effectuer nos recherches, et nous implémenterons ensuite l'ajout de clé sous forme de méthode déterminant l'emplacement d'insertion.

2 Arbre binaire de recherche

```
abr = ArbreBinaireRecherche(12)

abr.racine.gauche = Noeud(6)
abr.racine.droite = Noeud(14)

abr.racine.gauche.gauche = Noeud(5)
abr.racine.gauche.gauche.droite = Noeud(7)
```



b. Recherche de clé

La recherche de clé dans un arbre binaire de recherche exploite le caractère ordonné des clés.

Nous définissons une méthode de recherche qui détermine, de manière récursive à partir du nœud racine, si la clé du nœud courant correspond ou non à la valeur recherchée. Si ce n'est pas le cas, elle compare la clé à la valeur recherchée en évaluant si elle y est supérieure ou inférieure.

→ Les clés de l'arbre étant ordonnées, cela permet de déterminer dans quel sous-arbre la valeur se situe nécessairement, si elle est présente dans l'arbre.

```
class ArbreBinaireRecherche:
    def __init__ (self, racine):
        self.racine = Noeud(racine)

    def recherche(self, valeur_recherchee, noeud=None):
        if noeud is None:
            noeud = self.racine

        contenu = noeud.cle

        if contenu == valeur_recherchee:
            return True
```

```
        elif contenu < valeur_recherchee:
            if not noeud.droite:
                return False
            return self.recherche(valeur_recherchee, noeud.droite)
        else:
            if not noeud.gauche:
                return False
            return self.recherche(valeur_recherchee, noeud.gauche)
```

→ La fonction retourne `True` quand la valeur est trouvée, et `False` quand elle est absente de l'arbre.

```
print(abr.recherche(12))
# affiche True

print(abr.recherche(9))
# affiche False
```

Le caractère ordonné de l'arbre permet d'éliminer un sous-arbre sur deux à chaque étape. Nous avons utilisé des nombres pour les clés des nœuds, mais notre code fonctionne également avec des caractères alphabétiques, l'ordre appliqué étant dans ce cas l'ordre lexicographique (A est inférieur à B qui est lui-même inférieur à C etc.).

La complexité de cet algorithme de recherche dans un arbre binaire de recherche est d'ordre logarithmique, ce qui le rend efficace, y compris avec des volumes importants de données.

Nous allons nous appuyer sur cette même logique pour l'implémentation de l'insertion d'une nouvelle valeur dans l'arbre.

c. Ajout de clé

L'ajout d'une clé dans un arbre binaire de recherche doit s'effectuer de manière à préserver le caractère ordonné des clés. On effectue donc une démarche similaire à celle de la recherche, pour déterminer l'emplacement auquel la nouvelle clé doit être insérée.

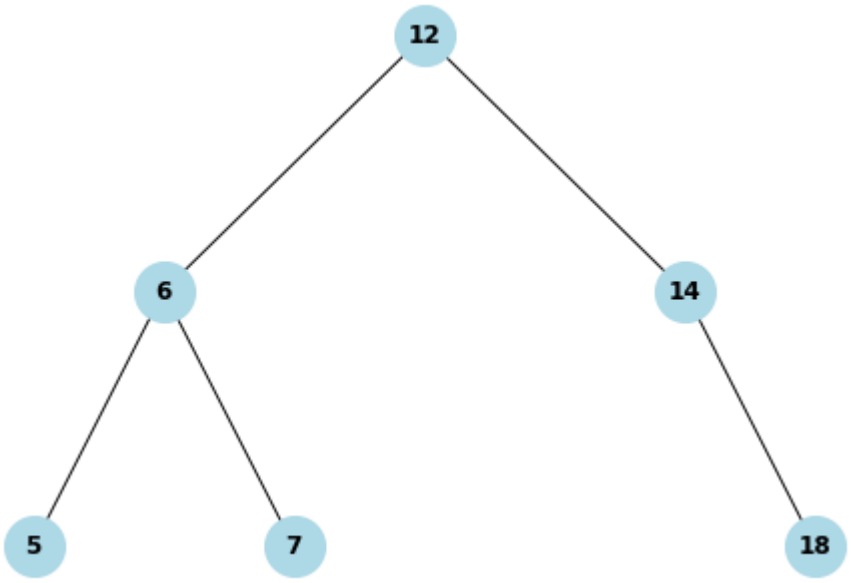
```
class ArbreBinaireRecherche:
```

```
def __init__(self, racine):
    self.racine = Noeud(racine)

def ajoute(self, nouvelle_valeur, noeud=None):
    if noeud is None:
        noeud = self.racine

    contenu = noeud.cle

    if contenu == nouvelle_valeur:
        return False
    elif contenu < nouvelle_valeur:
        if not noeud.droite:
            noeud.droite = Noeud(nouvelle_valeur)
            return True
        return self.ajoute(nouvelle_valeur, noeud.droite)
    else:
        if not noeud.gauche:
            noeud.gauche = Noeud(nouvelle_valeur)
            return True
        return self.ajoute(nouvelle_valeur, noeud.gauche)
```

Notre implémentation s'assure par ailleurs que la clé qu'on cherche à ajouter ne s'y trouve pas déjà.

→ La méthode retourne `True` en cas d'insertion réussie, et `False` dans le cas contraire, c'est-à-dire le cas où la valeur est déjà présente dans l'arbre.

```
print(abr.ajoute(18))
# affiche True

print(abr.ajoute(7))
# affiche False
```



La position d'insertion de la nouvelle valeur est dictée par la propriété ordonnée de l'arbre binaire de recherche, elle n'est donc pas spécifiée par l'utilisateur·rice mais déterminée par l'algorithme. Dans le cas d'un arbre binaire ou d'un arbre général, l'emplacement serait en revanche choisi par l'utilisateur·rice.

> Conclusion :
>
> Nous avons caractérisé les structures de données hiérarchiques que sont les arbres, dont les cas d'usage peuvent être très variés. Nous avons porté une attention plus particulière aux arbres binaires, et parmi eux aux arbres binaires de recherche. Nous avons ensuite proposé l'implémentation de la détermination de la taille et de la hauteur d'un arbre binaire, puis des méthodes pour leurs parcours de manière préfixe, infixe et suffixe. Nous avons ensuite implémenté des fonctionnalités de recherche et d'insertion de clé dans des arbres binaires de recherche.