

LES ARBRES

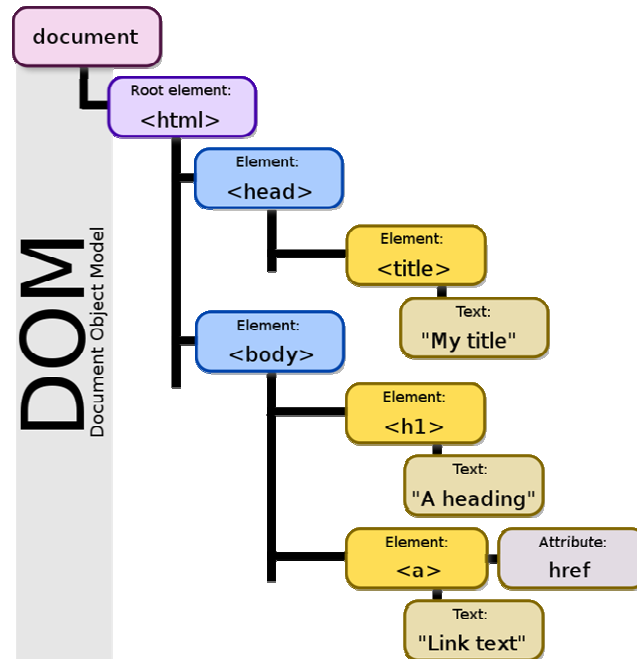
Après les **types abstraits de données (TAD) linéaires** nous allons étudier les arbres qui sont des structures de données hiérarchiques : **types abstraits de données (TAD) hiérarchiques**. Un arbre permet, contrairement aux listes, aux piles et aux files, de représenter des relations non séquentielles.

1. LES ARBRES

Voici quelques exemples d'arbres :

1.1. LE DOM (DOCUMENT OBJECT MODEL) D'UNE PAGE WEB

Structure d'une page Web :

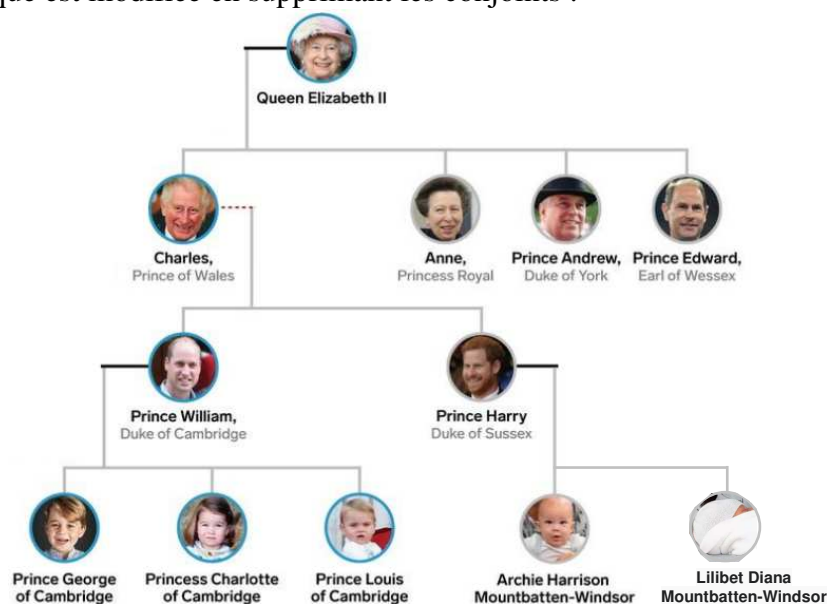


1.2. ARBRE GENEALOGIQUE DE LA FAMILLE ROYALE DU ROYAUME UNIS

La version originale

<https://i.pinimg.com/originals/e8/d1/c7/e8d1c7b2834ce2c368848cf7fc91a057.jpg>

de cet arbre généalogique est modifiée en supprimant les conjoints :



1.3. DEFINITIONS ET VOCABULAIRE

Un **arbre** est un ensemble de nœuds, reliés par des **arcs** ou des **arêtes**.

Les **nœuds** contiennent les données, aussi appelées les valeurs.

La **racine** est le nœud le plus haut dans la hiérarchie, elle est unique.

Les **feuilles** sont des nœuds terminaux.

Une **branche** est une suite de nœuds consécutifs de la racine vers une feuille

Les **nœuds internes** sont les nœuds qui ne sont ni des feuilles ni des racines.

Le **chemin** d'un nœud est la suite des nœuds qu'il faut traverser de la racine vers le noeud.

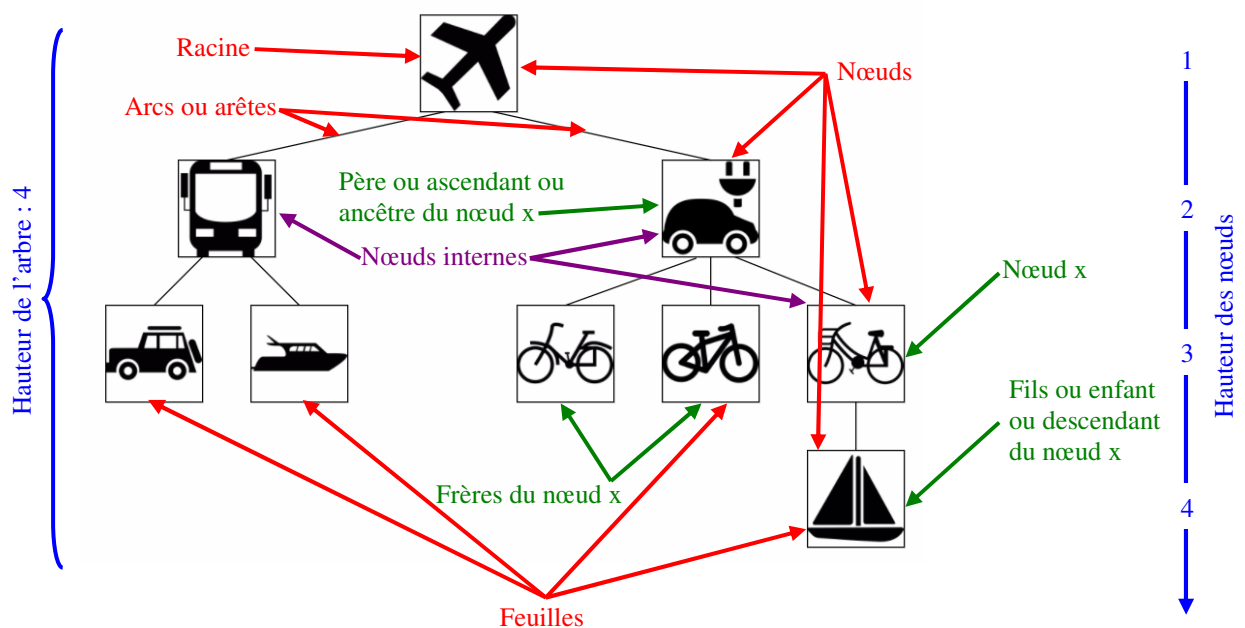
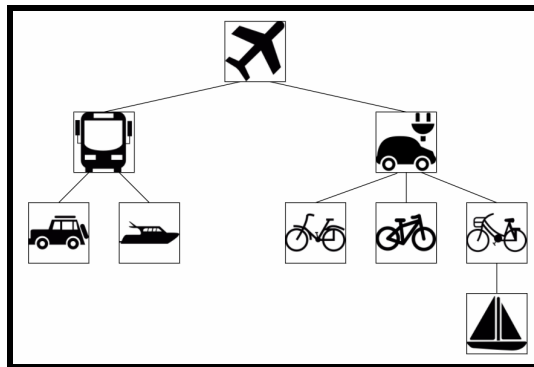
La **hauteur** (ou **profondeur**) d'un **nœud** est le nombre de nœuds de son chemin. La hauteur de la racine est 1.

La **hauteur de l'arbre** est la hauteur du nœud le plus haut, c'est à dire, le nombre de nœuds du chemin le plus long entre une feuille et la racine.

Attention : il n'existe pas de définition universelle pour la hauteur d'un arbre ou d'un nœud, dans certains cas la profondeur des nœuds est comptée à partir de 0 pour la racine.

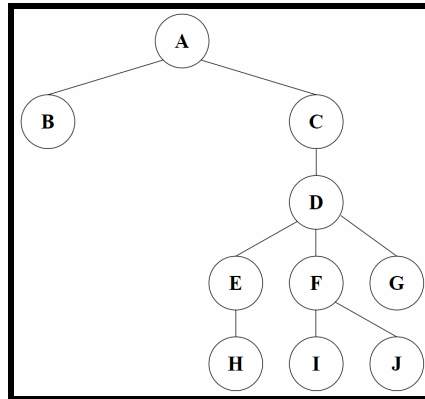
L'**arité** d'un nœud est le nombre de descendants qu'il a. L'arité d'un arbre est l'arité maximale de ses nœuds.

La taille d'un arbre est le nombre de nœuds qui le compose.



1.4. EXERCICES**1.4.1. EXERCICE 1**

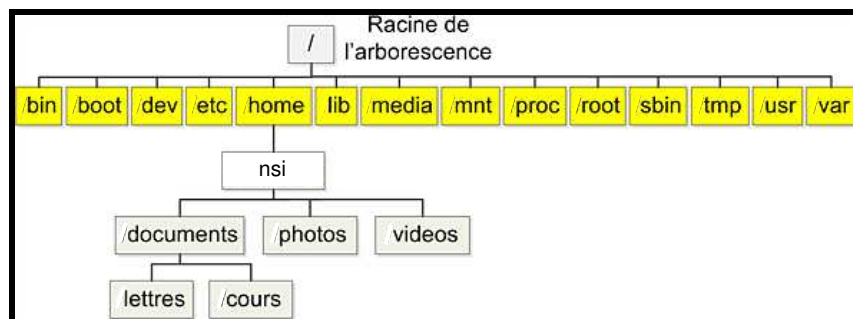
Soit l'arbre suivant :



1. Quelle est la racine de cet arbre ?
2. Combien y a t il de nœuds ? Citez les.
3. Combien y a t il de feuilles ? Citez les.
4. Combien y a t il de branches ?
5. Combien y a t il de nœuds internes ? Citez les.
6. Quels sont les descendants de D ?
7. Quel est l'ascendant de D ?
8. Quels sont les frères de G ?
9. Quelle est la hauteur de l'arbre si celle de la racine est 1 ?
10. Quelle est la profondeur de l'arbre si celle de la racine est 1 ?
11. Quelle est la hauteur du nœud F si celle de la racine est 1 ?
12. Quelle est la profondeur du nœud E si celle de la racine est 1 ?
13. Quelle est l'arité du nœud F ?
14. Quelle est l'arité du nœud C ?
15. Quelle est l'arité de cet arbre ?
16. Quelle est la taille de cet arbre ?
17. Quel est le chemin du nœud J ?

1.4.2. EXERCICE 2

Le système de fichiers de Linux est organisé ainsi :



1. Quelle est la racine de cet arbre ?
2. Combien y a t il de nœuds ? Citez les.
3. Combien y a t il de feuilles ? Citez les.
4. Combien y a t il de branches ?
5. Combien y a t il de nœuds internes ? Citez les.

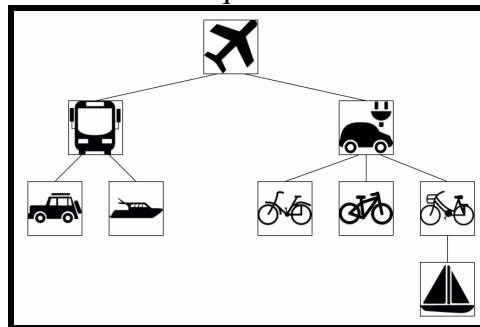
6. Quels sont les descendants de « documents » ?
7. Quel est l'ascendant de « documents » ?
8. Quels sont les frères de « documents » ?
9. Quelle est la hauteur de l'arbre si celle de la racine est 1 ?
10. Quelle est la profondeur de l'arbre si celle de la racine est 1 ?
11. Quelle est la hauteur du nœud « documents » si celle de la racine est 1 ?
12. Quelle est la profondeur du nœud « lettres » si celle de la racine est 1 ?
13. Quelle est l'arité du nœud « NSI » ?
14. Quelle est l'arité du nœud « documents » ?
15. Quelle est l'arité de cet arbre ?
16. Quelle est la taille de cet arbre ?
17. Quel est le chemin du nœud « cours » ?

2. LES ARBRES BINAIRES

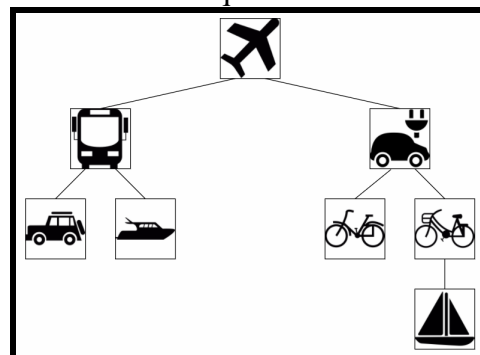
2.1. DEFINITIONS ET VOCABULAIRE

Un **arbre binaire** est un arbre pour lequel un père a au plus deux fils (arbre d'arité inférieure ou égale à 2, les nœuds ont 0, 1 ou 2 enfants).

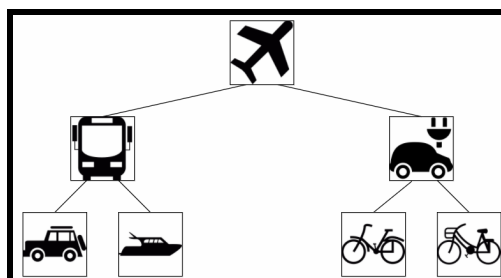
L'arbre suivant est-il un arbre binaire ? Pourquoi ?



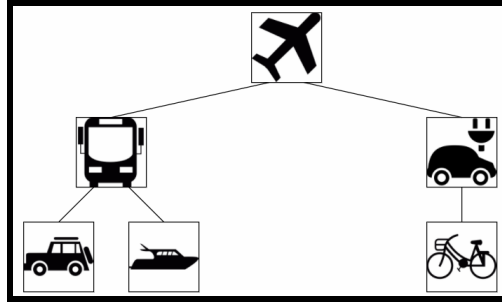
L'arbre suivant est-il un arbre binaire ? Pourquoi ?



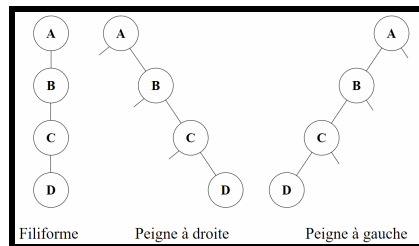
Un **arbre parfait** est un arbre dont tous les nœuds, sauf les feuilles, possèdent exactement 2 fils.



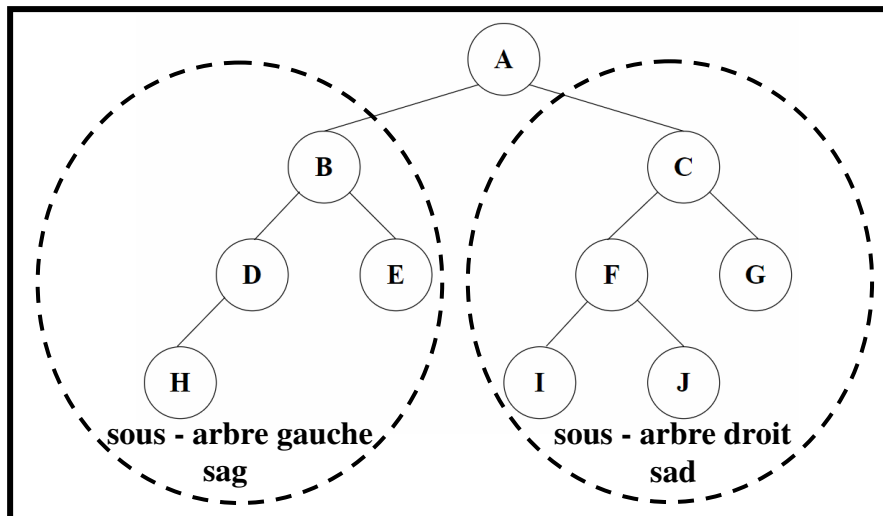
Un **arbre équilibré** est un arbre dont toutes les feuilles ont la même profondeur (hauteur).



Un **arbre filiforme, dégénéré, peigne à gauche ou peigne à droite** est un arbre dont tous les noeuds, sauf les feuilles, ne possède qu'un et un seul fils. Cet arbre est donc tout simplement une liste chaînée.



Soit A un noeud d'un arbre binaire, soient B et C ses deux fils. Le **sous - arbre gauche** de A est le sous - arbre dont B est la racine. Le **sous - arbre droit** de A est le sous - arbre dont C est la racine.



L'**encadrement de la hauteur h** d'un arbre binaire permet d'estimer sa hauteur h connaissant sa taille n (avec la définition de la hauteur qui correspond à la hauteur de la racine valant 1) :

- Sa hauteur maximale h_{\max} serait celle d'un arbre binaire filiforme de taille n est vaudrait donc : $h_{\max} = n$
- Sa hauteur minimale h_{\min} serait celle d'un arbre binaire parfait de taille n est vaudrait donc : $h_{\min} = \lceil \log_2(n+1) \rceil$ avec $\lceil \rceil$ signifiant entier immédiatement supérieur.

Casio : MATH - logab ou OPTN – CALC - > - logab

TI : math – A – baseLOG

$$\log_2(n) = \frac{\ln n}{\ln 2}$$

L'encadrement de la hauteur h d'un arbre binaire de taille n est donc :

$$h_{\min} \leq h \leq h_{\max}$$

$$\lceil \log_2(n+1) \rceil \leq h \leq n$$

Exemples :

- Donnez l'encadrement de la hauteur d'un arbre binaire de taille $n = 7$.
- Donnez l'encadrement de la hauteur d'un arbre binaire de taille $n = 55$.

L'encadrement de la taille n d'un arbre binaire permet d'estimer sa taille n connaissant sa hauteur h (avec la définition de la hauteur qui correspond à la hauteur de la racine valant 1) :

- Sa taille minimale n_{\min} serait celle d'un arbre binaire filiforme de taille n est vaudrait donc : $n_{\min} = h$
- Sa taille maximale n_{\max} serait celle d'un arbre binaire parfait de hauteur h est vaudrait donc : $n_{\max} = 2^h - 1$

L'encadrement de la taille n d'un arbre binaire de hauteur h est donc :

$$n_{\min} \leq n \leq n_{\max}$$

$$h \leq n \leq 2^h - 1$$

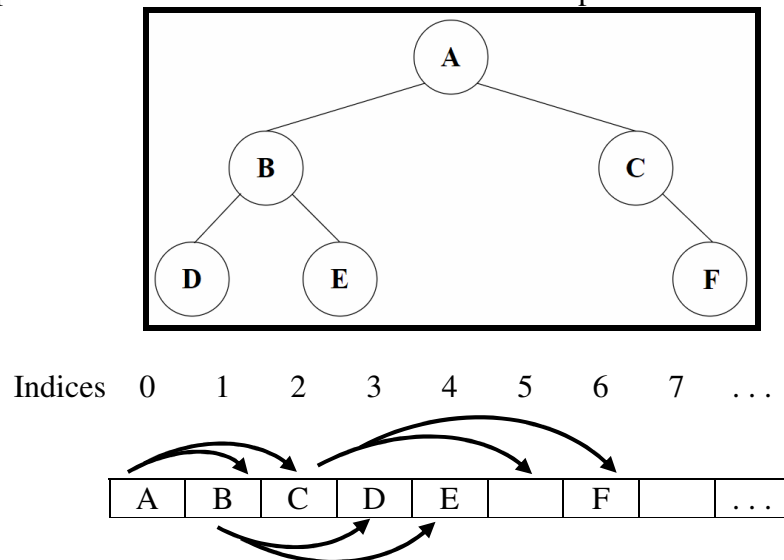
Exemples :

- Donnez l'encadrement de la taille d'un arbre binaire de hauteur $h = 5$.
- Donnez l'encadrement de la taille d'un arbre binaire de hauteur $h = 7$.

3. IMPLEMENTATIONS DES ARBRES BINAIRES

3.1. IMPLEMENTATIONS DES ARBRES BINAIRES AVEC DES LISTES

Il existe une méthode pour implémenter un arbre binaire (qui est une structure hiérarchique) avec une liste (qui est une structure linéaire). Ceci peut se faire par le biais d'une astuce sur les indices. Cette méthode est connue sous le nom de numérotation Eytzinger ou Stradonitz / Sosa, et est utilisée pour numéroter facilement les individus d'un arbre généalogique : les fils du nœud d'indice i sont placés aux indices $2i+1$ et $2i+2$ si la racine est placée à l'indice 0 de la liste :



$$\text{indice}_A = 0$$

$$\text{indice}_B = 2 * \text{indice}_A + 1 = 2 * 0 + 1 = 1$$

$$\text{indice}_C = 2 * \text{indice}_A + 2 = 2 * 0 + 2 = 2$$

$$\text{indice}_D = 2 * \text{indice}_B + 1 = 2 * 1 + 1 = 3$$

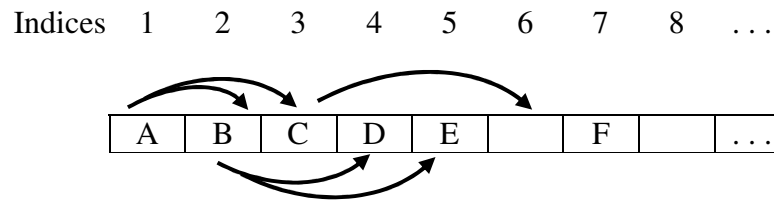
$$\text{indice}_E = 2 * \text{indice}_B + 2 = 2 * 1 + 2 = 4$$

$$\text{indice}_{\ll \gg} = 2 * \text{indice}_C + 1 = 2 * 2 + 1 = 5$$

$$\text{indice}_F = 2 * \text{indice}_C + 2 = 2 * 2 + 2 = 6$$

...

Si la racine de l'arbre est placée à l'indice 1, les fils du nœud d'indice i sont placés aux indices $2i$ et $2i+1$.



$$\text{indice}_A = 1$$

$$\text{indice}_B = 2 * \text{indice}_A = 2 * 1 = 2$$

$$\text{indice}_C = 2 * \text{indice}_A + 1 = 2 * 1 + 1 = 3$$

$$\text{indice}_D = 2 * \text{indice}_B = 2 * 2 = 4$$

$$\text{indice}_E = 2 * \text{indice}_B + 1 = 2 * 2 + 1 = 5$$

$$\text{indice}_{\ll \gg} = 2 * \text{indice}_C = 2 * 3 = 6$$

$$\text{indice}_F = 2 * \text{indice}_C + 1 = 2 * 3 + 1 = 7$$

...

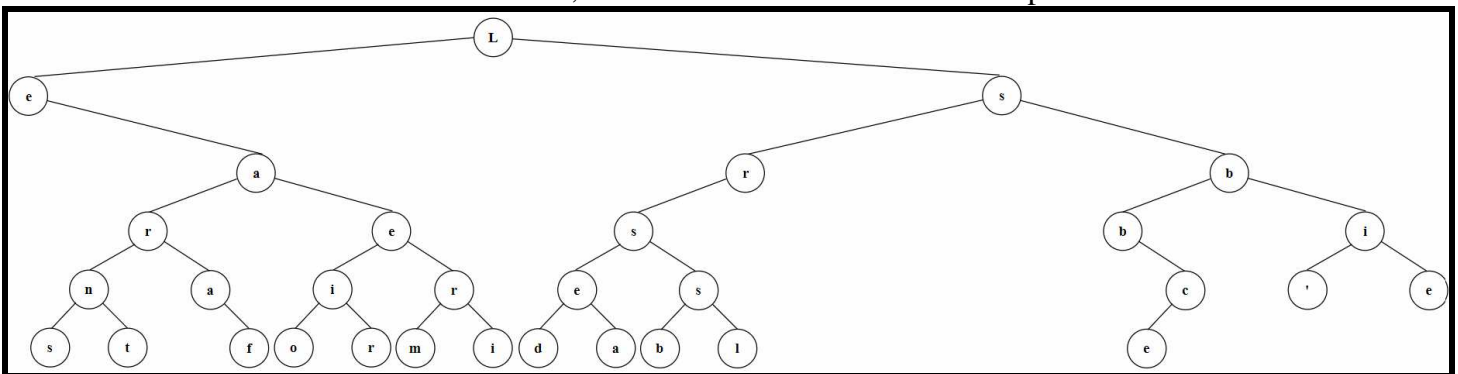
Le gros inconvénient d'une telle représentation : si l'arbre a beaucoup d'absence de fils droit ou de fils gauche, la liste comporte beaucoup d'emplacements vides.

3.1.1. EXEMPLE 1

Sachant q'un sous arbre vide est noté « _ », dessinez l'arbre représenté par la liste suivante : **exemple1** = [J, ', a, i, m, e, _, l, a, _, N, S, I, _, _].

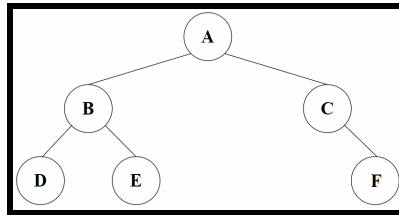
3.1.2. EXEMPLE 2

Soit l'arbre binaire suivant, donnez le contenu de la liste exemple2

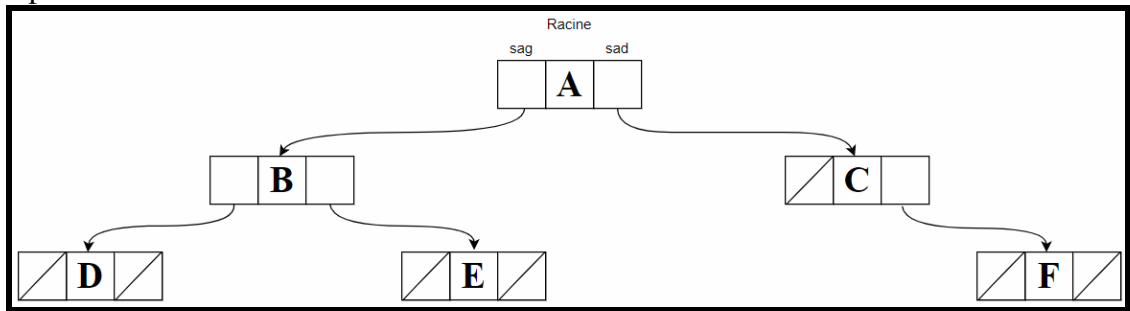


3.2. IMPLEMENTATIONS DES ARBRES BINAIRES AVEC DES TUPES IMBRIQUES

Un arbre peut se représenter par un tuple de la forme (sous - arbre gauche (sag), valeur du noeud, sous - arbre droit (sad)). Ainsi l'arbre suivant :



sera représenté :



par le tuple :

```
tup1 = ((None, D, None), B, (None, E, None)), A, ((None), C, (None, F, None))
```

3.2.1. EXEMPLE 1

Tracez l'arbre binaire correspondant au tuple suivant :

```
tup11 = ((None, 6, None), 8, (None, 9, None)), 2, ((None, 7, None), 1, None)
```

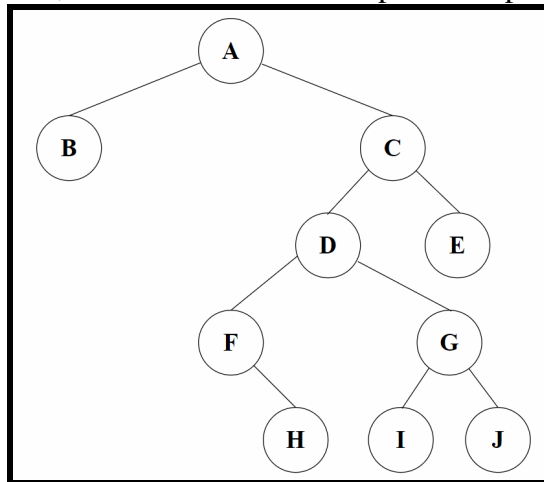
Dans la console python, saisissez les lignes de codes suivantes et commentez les résultats :

```

tup11 = ((None, 6, None), 8, (None, 9, None)), 2, ((None, 7, None), 1, None)
tup11[1]
tup11[0]
tup11[2]
tup11[0][1]
tup11[0][0]
tup11[0][2]
tup11[0][0][1]
tup11[0][2][1]
tup11[0][0][2]
print(tup11[0][0][2])
tup11[0][2][2]
print(tup11[0][2][2])
tup11[0][0][0]
print(tup11[0][0][0])
tup11[2]
tup11[2][1]
tup11[2][0]
tup11[2][0][0]
print(tup11[2][0][0])
tup11[2][0][1]
tup11[2][0][2]
print(tup11[2][0][2])
  
```


3.2.2. EXEMPLE 2

Soit l'arbre binaire suivant, donnez le contenu du tuple correspondant :

**3.3. IMPLEMENTATIONS DES ARBRES BINAIRES AVEC LA POO**

La classe **Arbre**, qui contiendra 3 attributs :

- **_racine** : la valeur du nœud (de type **Int**)
- **_gauche** : le sous - arbre gauche : sag (de type **Arbre**)
- **_droite** : le sous - arbre droite : sad (de type **Arbre**).

Par défaut, les attributs **_gauche** et **_droite** seront à **None**, qui représenteront le sag et le sad vides (ce n'est pas très rigoureux, car **None** n'est pas de type **Arbre**).

un « **_** » (underscore, trait de soulignement) devant les noms, dans une classe, permet simplement d'indiquer aux autres programmeurs que l'attribut ou la méthode est destiné à être privé. On dit que les données sont **protégées** mais elles sont accessibles normalement comme pour des données **publiques**.

Afin de respecter le paradigme de la Programmation Orientée Objet, nous allons jouer totalement le jeu de l'**encapsulation** en nous refusant d'accéder directement aux attributs.

Nous allons donc construire des méthodes permettant d'accéder à ces attributs (avec des **getters** / **accesseurs**) ou de les modifier (avec des **setters** / **mutateurs**).

Commentez, exécutez et vérifiez la validité du script suivant :

```

class Arbre:                                     # Arbre binaire en POO
    def __init__(self, valeur):
        self._racine = valeur
        self._gauche = None
        self._droite = None

    def set_gauche(self, sousarbre):
        self._gauche = sousarbre

    def set_droite(self, sousarbre):
        self._droite = sousarbre

    def get_gauche(self):
        return self._gauche

    def get_droite(self):
        return self._droite
  
```

```

def get_racine(self):
    return self._racine

def __str__(self):
    return "({}, {}, {})".format(self._gauche, self._racine,
                                   self._droite)

arbr = Arbre(4)
print(arbr)
arbr.set_gauche(Arbre(3))
print(arbr)
arbr.set_droite(Arbre(1))
print(arbr)
arbr.get_droite().set_gauche(Arbre(2))
print(arbr)
arbr.get_droite().set_droite(Arbre(7))
print(arbr)
arbr.get_gauche().set_gauche(Arbre(6))
print(arbr)
arbr.get_droite().get_droite().set_gauche(Arbre(9))
print(arbr)

print("La racine du sag du sad de l'arbre est : ",
      arbr.get_droite().get_gauche().get_racine())
print("La racine du sag du sad de l'arbre est : ",
      arbr.get_gauche().get_gauche().get_racine())
print("La racine du sag du sad de l'arbre est : ",
      arbr.get_droite().get_droite().get_gauche().get_racine())

```

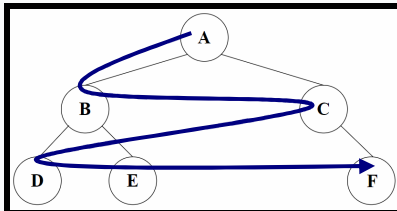
Dessinez l'arbre créé par le script précédent.

3.3.1. PARCOURS DES ARBRES BINAIRES

Les arbres ayant une structure hiérarchique, leur utilisation implique la nécessité d'un **parcours** des valeurs stockées pour toutes les récupérer dans un certain ordre ou pour en chercher une en particulier. L'objectif est de visiter tous les nœuds d'un arbre, sans visiter deux fois le même nœud. Il existe plusieurs manières de parcourir un arbre.

3.3.1.1. PARCOURS EN LARGEUR D'ABORD DES ARBRES BINAIRES

Le **parcours en largeur d'abord** (Breadth First Search(BFS) : recherche en largeur d'abord) consiste, simplement, à lire les nœuds dans le sens de la lecture.



Si l'arbre est représenté par une liste, cela revient simplement à lire les cases de celle-ci, les unes après les autres :

Indices	0	1	2	3	4	5	6	7	...
	A	B	C	D	E		F		...

L'ordre des lettres de cet arbre binaire lors d'un parcours en largeur d'abord est : A, B, C, D, E et F

Pour parcourir en largeur d'abord un tel arbre, on explore la racine, puis les successeurs non explorés (sag puis sad), puis les successeurs non explorés des successeurs, . . .

Il faut une file dans laquelle sera enfilé l'arbre et une liste dans laquelle seront stockés les nœuds de l'arbre et des sous – arbres. La méthode est :

Enfiler l'arbre dans une file nommée « file »

Créer une liste vide nommée « resultat »

Tant que la file n'est pas vide :

 Défiler le nœud de la file pour le mettre dans la liste en supprimant la file

 Enfiler le sag

 Enfiler le sad

Renvoyer la liste « resultat »

Commentez, exécutez et vérifiez la validité du script suivant :

```
from queue import Queue
. . .
def BFS(arbre):
    file = Queue()
    file.put(arbre)
    resultat = []
    while file.empty() is False :
        element = file.get()
        if element is not None :
            print("L'arbre ou le sous arbre est : ",element)
            resultat.append(element.get_racine())
            print(resultat)
            file.put(element.get_gauche())
            file.put(element.get_droite())
    return resultat
. . .
print("----- C'est la BFS (Breadth First Search), parcours
                                en largeur d'abord -----")
print("Le parcours en largeur d'abord de l'arbre donne la liste
                                suivante : ",BFS(arbr))
```

3.3.1.2. PARCOURS EN PROFONDEUR D'ABORD DES ARBRES BINAIRES

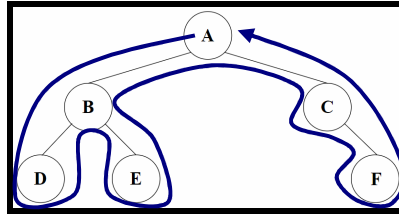
Les **parcours en profondeur d'abord** (Depth First Search (DFS) : recherche en profondeur d'abord) consiste à parcourir totalement un des deux sous - arbres avant que l'exploration du deuxième ne commence (explorer un chemin jusqu'au bout, avant de revenir en arrière pour emprunter un autre chemin).

3.3.1.2.1. PARCOURS PRÉFIXE OU PRÉORDRE DES ARBRES BINAIRES

Le parcours **préfixe** ou **préordre** est un parcours en **profondeur d'abord**.

Parcours préfixe :

- Chaque nœud est visité avant que ses fils le soient.
- On part de la racine, puis on visite son fils gauche (et éventuellement le fils gauche de celui-ci, . . .) avant de remonter et de redescendre vers le fils droit.



Lors du parcours préfixe chaque noeud est « traité » au premier « passage ».

L'ordre des lettres de cet arbre binaire lors d'un parcours préfixe est : A, B, D, E, C et F

L'algorithme du parcours préfixe est :

- **Visiter la racine du sous - arbre**
- **Parcourir le sous - arbre gauche (appel récursif du parcours préfixe du sag)**
- **Parcourir le sous - arbre droite (appel récursif du parcours préfixe du sad)**

Il faut une liste dans laquelle seront stockés les nœuds de l'arbre et des sous - arbres.
Il faut :

- **Créer une liste vide nommée « resultat »**
- **Ajouter la racine du sous - arbre à la liste « resultat »**
- **Si sag n'est pas vide :**
 - **Descendre dans le sag par appel récursif**
- **Si sad n'est pas vide :**
 - **Descendre dans le sad par appel récursif**
- **Renvoyer la liste « resultat »**
- **Aplatir la liste « resultat » (passer d'une liste de liste à une liste)**

Commentez, exécutez et vérifiez la validité du script suivant :

```

def prefixe(arbre) :
    resultat = []
    resultat.append(arbre._racine)
    if arbre._gauche is not None:
        resultat.append(prefixe(arbre._gauche))
    if arbre._droite is not None:
        resultat.append(prefixe(arbre._droite))
    return resultat

def aplatirListe(liste):
    for element in liste:
        if type(element) == list:
            aplatirListe(element)
        else:
            listePlate.append(element)
    return listePlate

. . .
print("----- C'est la DFS (Depth First Search), parcours
  
```

```

                                préfixe en profondeur d'abord -----")
print("Le parcours préfixe en profondeur de l'arbre donne la
                                liste suivante : ",prefixe(arbr))
listePlate = []
print("Le parcours préfixe en profondeur de l'arbre donne la
                                liste aplatie suivante : ",aplatirListe(prefixe(arbr)))

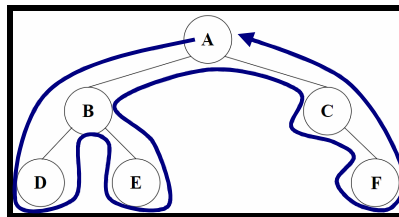
```

3.3.1.2.2. PARCOURS POSTFIXE DES ARBRES BINAIRES

Le parcours **postfixe** ou **suffixe** ou **post ordre** est un parcours **en profondeur d'abord**.

Parcours postfixe :

- Chaque nœud est visité après que ses fils le soient.
- On part de la feuille la plus à gauche, et on ne remonte à un nœud père que si ses fils ont tous été visités.



Lors du parcours postfixe chaque nœud est « traité » au dernier passage.

L'ordre des lettres de cet arbre binaire lors d'un parcours postfixe est : D, E, B, F, C et A

L'algorithme du parcours postfixe est :

- **Parcourir le sous - arbre gauche (appel récursif du parcours préfixe du sag)**
- **Parcourir le sous - arbre droite (appel récursif du parcours préfixe du sad)**
- **Visiter la racine du sous - arbre**

Il faut une liste dans laquelle seront stockés les nœuds de l'arbre et des sous - arbres.
Il faut :

- **Créer une liste vide nommée « resultat »**
- **Si sag n'est pas vide :**
 - **Descendre dans le sag par appel récursif**
- **Si sad n'est pas vide :**
 - **Descendre dans le sad par appel récursif**
- **Ajouter la racine du sous - arbre à la liste « resultat »**
- **Renvoyer la liste « resultat »**
- **Aplatir la liste « resultat » (passer d'une liste de liste à une liste)**

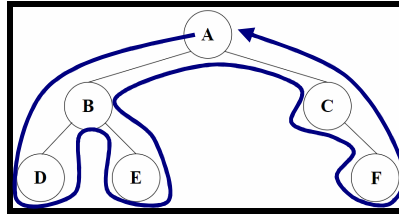
En utilisant le script du parcours préfixe ci-dessus, écrivez celui du parcours postfixe puis, commentez le, exécutez le et vérifiez sa validité.

3.3.1.2.3. PARCOURS INFIXE DES ARBRES BINAIRES

Le parcours **infixe** est un parcours **en profondeur d'abord**.

Parcours infixe :

- Chaque nœud est visité après son fils gauche mais avant son fils droit.
- On part de la feuille la plus à gauche, et on ne remonte par vagues successives. Un nœud ne peut pas être visité si son fils gauche ne l'a pas été.



Lors du parcours infixe chaque nœud :

- ayant un fils gauche est « traité » au deuxième passage.
- n'ayant un fils gauche est « traité » au premier passage.

L'ordre des lettres de cet arbre binaire lors d'un parcours infixe est : D, B, E, A, C et F

L'algorithme du parcours infixe est :

- **Parcourir le sous - arbre gauche (appel récursif du parcours préfixe du sag)**
- **Visiter la racine du sous - arbre**
- **Parcourir le sous - arbre droite (appel récursif du parcours préfixe du sad)**

Il faut une liste dans laquelle seront stockés les nœuds de l'arbre et des sous - arbres.

Il faut :

- **Créer une liste vide nommée « resultat »**
- **Si sag n'est pas vide :**
 - **Descendre dans le sag par appel récursif**
- **Ajouter la racine du sous - arbre à la liste « resultat »**
- **Si sad n'est pas vide :**
 - **Descendre dans le sad par appel récursif**
- **Renvoyer la liste « resultat »**
- **Aplatir la liste « resultat » (passer d'une liste de liste à une liste)**

En utilisant le script des parcours préfixe et postfixe ci-dessus, écrivez celui du parcours infixe puis, commentez le, exécutez le et vérifiez sa validité.

3.3.1.2.4. PARCOURS PRÉFIXE, INFIXE ET POSTFIXE DES ARBRES BINAIRES

Afin de ne pas mélanger les parcours préfixe, infixe et postfixe :

- « pré » signifie « avant »
- « in » signifie « au milieu »
- « post » signifie « après »

Ces trois mots-clés précisent la place du **père** par rapport à ses **fils** et le fils gauche est toujours traité avant le fils droit. Donc pour les parcours :

- préfixe : le père doit être le premier par rapport à ses fils.
- infixe : le père doit être entre son fils gauche (traité en premier) et son fils droit.
- postfixe : le père ne doit être traité que quand ses deux fils (gauche d'abord, droite ensuite) l'ont été.

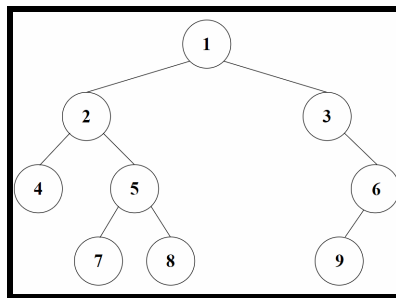
Un parcours préfixe commencera toujours par la racine, alors qu'un parcours postfixe finira toujours par la racine. Dans un parcours infixe, la racine sera « au milieu ».

3.3.1.3. EXERCICES PARCOURS DES ARBRES BINAIRES

3.3.1.3.1. EXERCICE 1

Soit l'arbre binaire suivant, donnez la liste des nœuds lors des parcours :

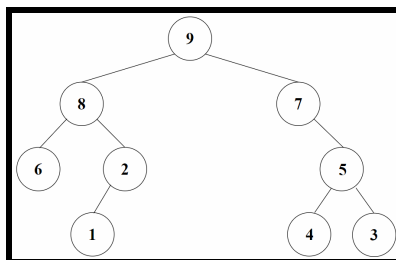
- En largeur
- Préfixe
- Infixe
- Postfixe



3.3.1.3.2. EXERCICE 2

Soit l'arbre binaire suivant, donnez la liste des nœuds lors des parcours :

- En largeur
- Préfixe
- Infixe
- Postfixe



3.3.2. CALCUL DE LA TAILLE DES ARBRES BINAIRES

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10), écrivez une fonction **taille** récursive multiple de paramètre **arbre**, permettant de calculer la taille d'un arbre binaire (nombre de nœuds contenus dans l'arbre binaire) puis, commentez la, exécutez la et vérifiez sa validité.

3.3.3. CALCUL DE LA HAUTEUR DES ARBRES BINAIRES

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10), écrivez une fonction **hauteur** récursive multiple de paramètre **arbre**, permettant de calculer la hauteur d'un arbre binaire (hauteur du nœud le plus haut, c'est à dire, le nombre de nœuds du chemin le plus long entre une feuille et la racine. Nous utiliserons le fait que la hauteur de la racine est 1 et qu'un arbre vide a pour hauteur 0) puis, commentez la, exécutez la et vérifiez sa validité.

3.3.4. RECHERCHER UNE CLE DANS UN ARBRE BINAIRE

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10), écrivez une fonction **recherche** récursive multiple de paramètres **arbre** et **valeur**, permettant de rechercher une clé (valeur) dans un arbre binaire puis, commentez la, exécutez la et vérifiez sa validité avec :

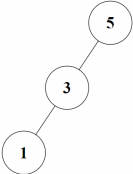
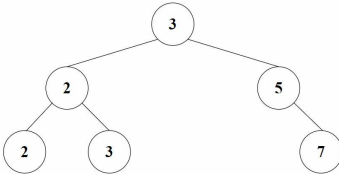
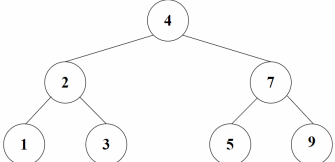
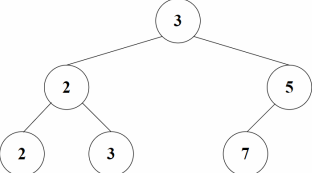
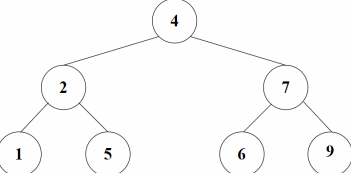
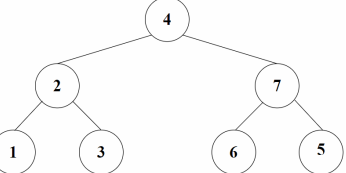
```
print("----- Recherche d'une clé (valeur) dans l'arbre -----")
for val in range(0,21):
    print("La clé",val,"est dans l'arbre : ",recherche(arbr,val))
```

4. LES ARBRES BINAIRES DE RECHERCHE (ABR)

Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire dont l'ensemble des valeurs des nœuds (étiquettes ou clés) vérifient les propriétés suivantes :

- Les valeurs du **sous - arbre gauche** sont **inférieures ou égales** à la valeur du **nœud**.
- Les valeurs du **sous - arbre droit** sont **strictement supérieures** à la valeur du **nœud**.

Le TAD (types abstraits de données) Arbre Binaire de Recherche (ABR) est une extension des arbres binaires vus dans les paragraphes précédents. Il possède donc les mêmes caractéristiques.

ABR	 <p style="text-align: center;">Arbre 1</p>	 <p style="text-align: center;">Arbre 2</p>	 <p style="text-align: center;">Arbre 3</p>
Pas ABR	 <p style="text-align: center;">Arbre 4</p>	 <p style="text-align: center;">Arbre 5</p>	 <p style="text-align: center;">Arbre 6</p>

L'arbre 4 n'est pas un ABR à cause de la feuille 7, qui fait partie du sag de 5 sans lui être inférieure.

L'arbre 5 n'est pas un ABR à cause de la feuille 5, qui fait partie du sag de 4 sans lui être inférieure.

L'arbre 6 n'est pas un ABR à cause de la feuille 5, qui fait partie du sad de 7 sans lui être supérieure.

4.1. VERIFIER SI UN ARBRE BINAIRE EST UN ARBRE BINAIRE DE RECHERCHE (ABR)

En récupérant le parcours infixe dans une liste aplatie et en testant si cette liste est ordonnée, il est possible de vérifier si un arbre binaire est ou n'est pas un ABR (arbre binaire de recherche).

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10), écrivez :

- une fonction **aplatirListe** récursive de paramètres **liste** et **listePlateParcoursInfixe**, permettant d'aplatir une liste (transformer une liste de liste en liste) obtenue par la fonction **infixe** vu précédemment. L'appel de la fonction se fera par : **aplatirListe(infixe(arbr), [])**,
- une fonction **est_ABR** de paramètres **arbre** et **list**, permettant de tester si l'arbre **arbre** est un arbre binaire de recherche (ABR) en testant si la liste **list** contenant le

parcours infixe de l'arbre est triée. L'appel de la fonction se fera par : **est_ABR(arbr, aplatirListe(infixe(arbr), []))**

Commentez ces fonctions, exécutez les et vérifiez leurs validités en vous aidant du script suivant :

```
# ARBRE BINAIRE N°1 mais pas un arbre binaire de recherche
print("----- Arbre N°1 pas ABR -----")
arbr = Arbre(4)
# Instancier / Créer l'arbre dont le nœud a pour valeur 4, les sag
# et sad sont None
print(arbr)                                # Afficher l'arbre
arbr.set_gauche(Arbre(3))
# Instancier / Créer l'arbre dont le nœud a pour valeur 3, les sag
# et sad sont None pour sag de a
print(arbr)                                # Afficher l'arbre
arbr.set_droite(Arbre(1))
# Instancier / Créer l'arbre dont le nœud a pour valeur 1, les sag
# et sad sont None pour sad de a
print(arbr)                                # Afficher l'arbre
arbr.get_droite().set_gauche(Arbre(2))
# Instancier / Créer l'arbre dont le nœud a pour valeur 2, les sag
# et sad sont None pour sag du sad de a
print(arbr)                                # Afficher l'arbre
arbr.get_droite().set_droite(Arbre(7))
# Instancier / Créer l'arbre dont le nœud a pour valeur 7, les sag
# et sad sont None pour sad du sad de a
print(arbr)                                # Afficher l'arbre
arbr.get_gauche().set_gauche(Arbre(6))
# Instancier / Créer l'arbre dont le nœud a pour valeur 6, les sag
# et sad sont None pour sag du sag de a
print(arbr)                                # Afficher l'arbre
arbr.get_droite().get_droite().set_gauche(Arbre(9))
# Instancier / Créer l'arbre dont le nœud a pour valeur 9, les sag
# et sad sont None pour sag du sag du sad de a
print(arbr)                                # Afficher l'arbre

# ARBRE BINAIRE N°2 et un arbre binaire de recherche
print("\n----- Arbre N°2 ABR -----")
arbr2 = Arbre(5)
# Instancier / Créer l'arbre dont le nœud a pour valeur 4, les sag
# et sad sont None
print(arbr2)                                # Afficher l'arbre
arbr2.set_gauche(Arbre(2))
# Instancier / Créer l'arbre dont le nœud a pour valeur 3, les sag
# et sad sont None pour sag de a
print(arbr2)                                # Afficher l'arbre
arbr2.set_droite(Arbre(7))
# Instancier / Créer l'arbre dont le nœud a pour valeur 1, les sag
# et sad sont None pour sad de a
print(arbr2)                                # Afficher l'arbre
arbr2.get_droite().set_gauche(Arbre(6))
```

```

# Instancier / Créer l'arbre dont le nœud a pour valeur 2, les sag
# et sad sont None pour sag du sad de a
print(arbr2)                                # Afficher l'arbre
arbr2.get_droite().set_droite(Arbre(8))
# Instancier / Créer l'arbre dont le nœud a pour valeur 7, les sag
# et sad sont None pour sad du sad de a
print(arbr2)                                # Afficher l'arbre
arbr2.get_gauche().set_gauche(Arbre(0))
# Instancier / Créer l'arbre dont le nœud a pour valeur 6, les sag
# et sad sont None pour sag du sag de a
print(arbr2)                                # Afficher l'arbre
arbr2.get_gauche().set_droite(Arbre(3))
# Instancier / Créer l'arbre dont le nœud a pour valeur 6, les sag
# et sad sont None pour sag du sag de a
print(arbr2)                                # Afficher l'arbre

print("\n----- Test ABR -----")
print("\n--- Test arbre binaire N°1 pas ABR ---")
print("Le parcours infixe en profondeur de l'arbre N°1 donne la
      liste suivante : ",infixe(arbr))
# Créer une liste de liste contenant le parcours infixe
print("Le parcours infixe en profondeur de l'arbre N°1 donne la
      liste aplatie suivante : ",aplatirListe(infixe(arbr),[]))
# Créer une liste aplatie contenant le parcours infixe
print("L'arbre binaire N°1 est un arbre binaire de recherche : ",
      est_ABR(arbr, aplatirListe(infixe(arbr),[])))
# Vérifier si l'arbre est une ABR en vérifiant si la liste aplatie
# est ordonnée

print("\n--- Test arbre binaire N°2 ABR ---")
print("Le parcours infixe en profondeur de l'arbre N°1 donne la
      liste suivante : ",infixe(arbr2))
print("Le parcours infixe en profondeur de l'arbre N°2 donne la
      liste aplatie suivante : ",aplatirListe(infixe(arbr2),[]))
print("L'arbre binaire N°2 est un arbre binaire de recherche : ",
      est_ABR(arbr2, aplatirListe(infixe(arbr2),[])))

```

4.2. RECHERCHER UNE CLE DANS UN ARBRE BINAIRE DE RECHERCHE (ABR)

Nous avons déjà vu cette recherche dans un paragraphe précédant pour un arbre binaire. Pour un ABR, le script sera légèrement modifié pour tenir compte des caractéristiques des ABR et afin d'être plus efficace. Dans un ABR, le fait que les valeurs soient « ordonnées » va considérablement améliorer la vitesse de recherche de cette clé, puisque la moitié de l'arbre restant sera écartée après chaque comparaison avec la clé à chercher.

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10) et l'arbre binaire N°2 ci-dessus (pages 17 et 18) étant un ABR, écrivez une fonction **rechercheABR** récursive multiple de paramètres **arbre** et **valeur**, permettant de rechercher une clé (valeur) dans un arbre binaire de recherche puis, commentez la, exécutez la et vérifiez sa validité avec :

```

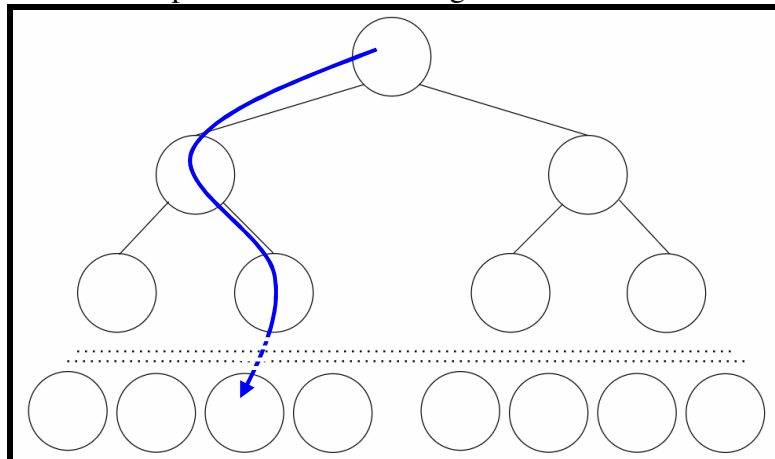
print("\n--- Test ABR ---")
print("Le parcours infixe en profondeur de l'arbre N°1 donne la
      liste suivante : ",infixe(arbr2))
print("Le parcours infixe en profondeur de l'arbre N°2 donne la
      liste aplatie suivante : ",aplatirListe(infixe(arbr2), []))
print("L'arbre binaire N°2 est un arbre binaire de recherche :
      ", est_ABR(arbr2, aplatirListe(infixe(arbr2), [])))

print("\n----- Recherche d'une clé (valeur) dans l'arbre
      binaire de recherche -----")
for val in range(0,21):
    print("La clé",val,"est dans l'arbre binaire de recherche :
          ",rechercheABR(arbr2,val))

```

4.2.1. COUT DE LA RECHERCHE D'UNE CLE DANS UN ARBRE BINAIRE EQUILIBRE DE RECHERCHE (ABR)

Soit un arbre binaire de recherche équilibré de taille n . Après chaque nœud, le nombre de nœuds restant à explorer est divisé par 2. On retrouve là le principe de recherche dichotomique, vu en classe de première. Considérons le pire des cas, où il faut parcourir tous les étages de l'arbre avant de trouver la clé recherchée (si elle est présente) ou si elle n'est pas présente. Le nombre de nœuds parcourus sera donc égal à la hauteur h de l'arbre.



Pour un arbre complet de taille n , sa hauteur h vérifie la relation $n = 2^h - 1$ et donc $2^h = n + 1$. h est donc le « nombre de puissance de 2 » que l'on peut mettre dans $n + 1$. Cette notion s'appelle le logarithme de base 2 et se note \log_2 . Par exemple, $\log_2(1024) = 10$ car $2^{10} = 1024$.

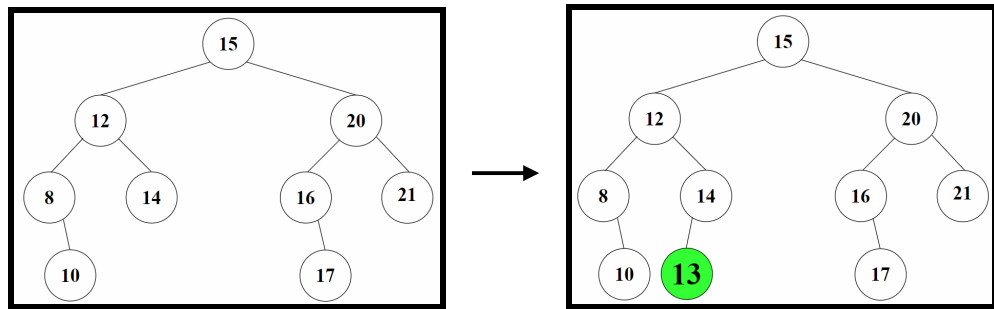
Le nombre maximal de nœuds à parcourir pour rechercher une clé dans un arbre binaire de recherche (ABR) équilibré de taille n est donc de l'ordre de $\log_2(n)$ (ceci est relativement performant). Pour ABR contenant 2000 valeurs, 11 étapes suffisent à rechercher une valeur.

Cette **complexité logarithmique** est un atout essentiel de la structure d'arbre binaire de recherche (ABR).

4.3. INSERER UNE CLE DANS UN ARBRE BINAIRE DE RECHERCHE (ABR)

L'ajout d'un élément dans un ABR se fait au niveau des feuilles. Il faut distinguer dans lequel des deux sous - arbres gauche ou droit doit se trouver la clé à insérer, et d'appeler récursivement la fonction d'insertion dans le sous - arbre concerné.

Par exemple pour insérer la clé « 13 » dans l'arbre binaire de recherche suivant :



Il faudra « naviguer » dans l'arbre binaire de recherche, en partant de la racine jusqu'à trouver un emplacement qui permettra de conserver la caractérisation d'un ABR. Nous obtenons alors l'algorithme suivant :

- Si l'arbre est vide,
 - renvoyer un nouvel objet Arbre contenant la clé.
- Sinon,
 - Si la clé est inférieure ou égale à la valeur du nœud,
 - modifier le sag par appel récursif.
 - Si la clé est supérieure à la valeur du nœud,
 - modifier le sad par appel récursif.
- Renvoyer le nouvel arbre modifié.

En utilisant le script de la classe « **Arbre** » ci-dessus (pages 9 et 10) et l'arbre binaire N°2 ci-dessus (pages 17 et 18) étant un ABR, écrivez une fonction **insertion** récursive multiple de paramètres **arbre** et **valeur**, permettant d'insérer une clé (valeur) dans un arbre binaire de recherche puis, commentez la, exécutez la et vérifiez sa validité avec :

```

print("\n----- Insérer une clé (valeur) dans l'arbre
                               binaire de recherche -----")

vale = 1
vale1 = 4
vale2 = 14
vale3 = 8

print("Insertion de la clé",vale,"dans l'arbre binaire de
                               recherche N°2 : ", insertion(arbr2, vale))
print("Insertion de la clé",vale1,"dans l'arbre binaire de
                               recherche N°2 : ", insertion(arbr2, vale1))
print("Insertion de la clé",vale2,"dans l'arbre binaire de
                               recherche N°2 : ", insertion(arbr2, vale2))
print("Insertion de la clé",vale3,"dans l'arbre binaire de
                               recherche N°2 : ", insertion(arbr2, vale3))

print("\n--- Test ABR ---")
print("Le parcours infixe en profondeur de l'arbre N°1 donne la
                               liste suivante : ",infixe(arbr2))
print("Le parcours infixe en profondeur de l'arbre N°2 donne la
                               liste aplatie suivante : ",aplatirListe(infixe(arbr2), []))
print("L'arbre binaire N°2 est un arbre binaire de recherche :
```

```

        ", est_ABR(arbr2, aplatirListe(infixe(arbr2), [])))

print("\n----- Recherche d'une clé (valeur) dans l'arbre
                               binaire de recherche -----")
for val in range(0,21):
    print("La clé",val,"est dans l'arbre binaire de recherche :
                                                ",rechercheABR(arbr2,val))

```

4.4. IMPLEMENTATIONS DES ARBRES BINAIRES DE RECHERCHE AVEC LA POO ET UNE STRUCTURE RECURSIVE POUR AJOUTER LES NOEUDS

Uniquement pour ceux qui sont très en avance.

L'arbre est une structure récursive définie par :

- Une valeur (la donnée du nœud),
- Les références vers le sag : sous - arbre gauche et le sad : sous - arbre droit, qui sont eux-mêmes représenté par un nœud, d'où la définition récursive.

La classe **Arbre**, qui contiendra 3 attributs :

- **_racine** : la valeur du nœud (de type **Int**)
- **_gauche** : le sous - arbre gauche : sag (de type **Arbre**)
- **_droite** : le sous - arbre droite : sad (de type **Arbre**).

Par défaut, les attributs **_racine**, **_gauche** et **_droite** seront à **None**, qui représenteront le nœud, le sag et le sad vides (ce n'est pas très rigoureux, car **None** n'est pas de type **Arbre**).

un « **_** » (underscore, trait de soulignement) devant les noms, dans une classe, permet simplement d'indiquer aux autres programmeurs que l'attribut ou la méthode est destiné à être privé. On dit que les données sont **protégées** mais elles sont accessibles normalement comme pour des données **publiques**.

Complétez le script suivant en écrivant la méthode **ajouter** récursive multiple de paramètres **self** (arbre sur lequel on travaille) et **valeur**, permettant d'insérer une clé (valeur) dans un arbre binaire de recherche sans doublons (ajouter une valeur déjà existante dans l'arbre est sans effet). Commentez ce script, exécutez le et vérifiez sa validité.

```

class Arbre:
    def __init__(self):
        self._racine = None
        self._gauche = None
        self._droite = None
    def estVide(self):
        return self._racine is None
    def get_racine(self):
        assert not self.estVide(), "l'arbre est vide"
        return self._racine
    def get_sag(self):
        assert not self.estVide(), "l'arbre est vide"
        return self._gauche
    def get_sad(self):
        assert not self.estVide(), "l'arbre est vide"
        return self._droite

```

```

def __str__(self):
    return "{},{},{}".format(self._gauche, self._racine,
                               self._droite)

def ecrire(self):
    if self.estVide():
        print("arbre vide")
    else:
        self._ecrire(0)
def _ecrire(self, niveau):
    if self._droite:
        self._droite._ecrire(niveau + 1)
    print("{}{}".format(' ' * 4 * niveau, self._racine))
    if self._gauche:
        self._gauche._ecrire(niveau + 1)
def ajouter(self, valeur):

    # . . . à compléter . . .

    # Nous n'avons pas mis else pour éviter de mettre des
    # doublons dans l'arbre

    # . . . à compléter . . .

def listeToArbre(Lst):
    for i in Lst:
        arb.ajouter(i)
        print(arb)
    arb.ecrire()
arb = Arbre()
Liste = [13,26,18,5,3,4,12,18,6,7]
listeToArbre(Liste)

```

Complétez la méthode suivante **hauteur** récursive multiple de paramètre **self** (arbre sur lequel on travaille), permettant de calculer la hauteur d'un arbre binaire de recherche (définition avec la hauteur de la racine valant 1) :

```

def hauteur(self):
    if self.estVide():
        return . . .
    else:
        return . . .

def _hauteur(self):
    . . .
# . . . à compléter . . .
    . . .

```

Ajoutez ce script au précédent, commentez le, exécutez le et vérifiez sa validité avec la ligne suivante :

```

print("La hauteur de l'arbre binaire de recherche est :
      ", arb.hauteur())

```

Ajoutez la ligne suivante au début du script précédent, commentez la :

```
import matplotlib.pyplot as plt
```

Ajoutez la méthode suivante à la classe **Arbre** précédente :

```
def _draw(self):
# Méthode protégée permettant de tracer le graphique de l'ABR
plt.rcParams.update({'font.size': 7})
# Définition de la taille de police

# Parcours en largeur
an = []
# Stockage des noeuds visités
res = []
# Stockage des noeuds parcourus en largeur
posInit = 10 * self.hauteur()
# Calculer la position des noeuds
an.append((self, 0, posInit, "racine"))
# Racine affichée en (0, posInit)
# le parcours est terminé quand tous les noeuds ont été traités
while (len(an) > 0):
# Tant que les noeuds ne sont pas tous visités
n = an[0]
# Conserver le noeud courant pour gérer l'affichage
res.append(n)
# Ajouter le nouveau noeud visité à la liste res
an.pop(0)
# Retirer de la liste an des noeuds visités
h = n[2] - 10
# Calculer la position du noeud
ecart = 20 * n[0].hauteur()**2
# Améliorer l'affichage en écartant la position des noeuds

# Les fils du noeud courant sont ajoutés
# Tracer de l'arc / l'arête entre le noeud courant et les fils
# gauche et droite
if (n[0]._gauche is not None):
# Si le sag n'est pas vide
an.append((n[0]._gauche, n[1] - (ecart + h), h, "left"))
# Ajouter le sag à la liste an
plt.plot([n[1], n[1] - (ecart + h)], [n[2] - 3, h + 3],
        color='red', marker='o')
# Tracer le sur le schéma
if (n[0]._droite is not None):
# Si le sad n'est pas vide
an.append((n[0]._droite, n[1] + (ecart + h), h,
        "droite"))
# Ajouter à la liste an
plt.plot([n[1], n[1] + (ecart + h)], [n[2] - 3, h + 3],
        color='green', marker='o')
```

```

# Tracer le sur le schéma
# Une fois les noeuds positionnés, on les affiche, en calculant
# la taille de graphique nécessaire
xmin = res[0][1]
# Mettre la première valeur dans x minimum
xmax = res[0][1]
# Mettre la première valeur dans x maximum
# Calculer le décalage nécessaire pour que les noeuds ne se
# chevauchent pas
nbNoeuds = len(res)
# Nombre de noeuds est égale à la longueur de la liste des
# noeuds parcourus
for x in res:
# Pour tous les noeuds visités
    if (x[0] is not None):
# S'il y a bien des noeuds dans l'ABR
        if (x[3] == "gauche"):
# Faire un décalage adapté pour les fils gauche ou droite
            decalage = (-nbNoeuds) * 2.5
# Décaler de 2,5 * nombre de noeuds à gauche
        else:
# Sinon
            decalage = nbNoeuds
# Décaler de 1 * nombre de noeuds
        plt.annotate(str(x[0]._racine), (x[1] + decalage, x[2]))
# La valeur du noeud est affichée
        if (x[1] < xmin):
# Si la valeur à l'indice 1 est inférieur au minimum
            xmin = x[1]
# Modifier le minimum
        if (x[1] > xmax):
# Si la valeur à l'indice 1 est supérieur au maximum
            xmax = x[1]
# Modifier le maximum
plt.xlim(xmin - 100, xmax + 100)
# Fixer la taille du schéma en largeur
plt.ylim(-10, posInit + 10)
# Fixer la taille du schéma en hauteur
plt.show()
# Afficher le schéma

```

Dans la fonction **listeToArbre** ci-dessus, ajouter la ligne :

```
arb._draw()
```

Exécutez l'ensemble et indiquez ce qu'il se passe.

Complétez les méthodes suivantes **parcoursPre**, **parcoursInf** et **parcoursPost** récursive multiple de paramètre **self** (arbre sur lequel on travaille), permettant de renvoyer les parcours préfixe, infixe et postfixe de l'ABR :

```
def parcoursPre(self):
```



```

    if self.estVide():
        return . . .
    else:
        return . . .

def _parcoursPre(self, l):
    . . .
# . . . à compléter . . .
    . . .

def parcoursInf(self):
    if self.estVide():
        return . . .
    else:
        return . . .

def _parcoursInf(self, l):
    . . .
# . . . à compléter . . .
    . . .

def parcoursPost(self):
    if self.estVide():
        return . . .
    else:
        return . . .

def _parcoursPost(self, l):
    . . .
# . . . à compléter . . .
    . . .

```

Ajoutez ce script au précédent, commentez le, exécutez le et vérifiez sa validité avec les lignes suivantes :

```

print("Le parcours préfixe de l'arbre binaire de recherche est :
      ",arb.parcoursPre())
print("Le parcours infixe de l'arbre binaire de recherche est :
      ",arb.parcoursInf())
print("Le parcours postfixe de l'arbre binaire de recherche est
      : ",arb.parcoursPost())

```

Complétez la méthode suivante **parcoursLargeur** récursive multiple de paramètre **self** (arbre sur lequel on travaille), permettant de renvoyer le parcours en largeur de l'ABR :

```

def parcoursLargeur(self):
    if self.estVide():
        return . . .

```

```

else:
    return . . .

def _parcoursLargeur(self, l):
    . . .
# . . . à compléter . . .
    . . .

```

Ajoutez ce script au précédent, commentez le, exécutez le et vérifiez sa validité avec la ligne suivante :

```

print("Le parcours en largeur de l'arbre binaire de recherche
      est : ",arb.parcoursLargeur())

```

Complétez la méthode suivante **taille** récursive multiple de paramètre **self** (arbre sur lequel on travaille), permettant de renvoyer la taille (nombre de nœuds) de l'ABR :

```

def taille(self):
    if self.estVide():
        return . . .
    else:
        . . .
        # . . . à compléter . . .
        . . .

```

Ajoutez ce script au précédent, commentez le, exécutez le et vérifiez sa validité avec la ligne suivante :

```

print("La taille de l'arbre binaire de recherche est :
      ",arb.taille())

```

Complétez la méthode suivante **rechercher** récursive multiple de paramètre **self** (arbre sur lequel on travaille) et **valeur** (valeur recherchée dans l'ABR), permettant de renvoyer le booléen **True** si la valeur est présente dans l'ABR et **False** sinon :

```

def rechercher(self, valeur):
    if self._racine is None:
        return . . .
    elif:
        . . .
        # . . . à compléter . . .
        . . .

```

Ajoutez ce script au précédent, commentez le, exécutez le et vérifiez sa validité avec les lignes suivantes :

```

for val in range(0,30):
    print("La clé",val,"est dans l'arbre binaire de recherche :
          ",arb.rechercher(val))

```

Complétez les méthodes suivantes **minimum** et **maximum** récursives de paramètre **self** (arbre sur lequel on travaille), permettant de renvoyer la valeur minimum et la valeur maximum dans l'ABR :

```
def minimum(self):  
    Arbre = self  
    while . . .  
        . . .  
        # . . . à compléter . . .  
        . . .
```

```
def maximum(self):  
    Arbre = self  
    while . . .  
        . . .  
        # . . . à compléter . . .  
        . . .
```

Ajoutez ces scripts au précédent, commentez les, exécutez les et vérifiez leur validité avec les lignes suivantes :

```
print("Le maximum de l'arbre binaire de recherche est " +  
      str(arb.maximum()))  
print("Le minimum de l'arbre binaire de recherche est " +  
      str(arb.minimum()))
```