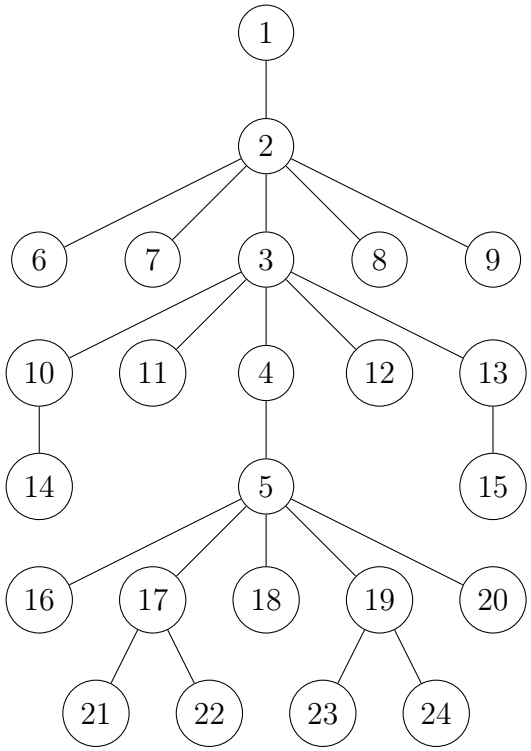
Arbres

1. Définitions

Exercice 1
On donne l'arbre suivant :



1. Déterminer pour cet arbre, sa racine, sa taille, sa hauteur, ses nœuds internes et ses feuilles.
2. Pour le nœud `4`, déterminer son père, ses frères, sa hauteur, sa profondeur.

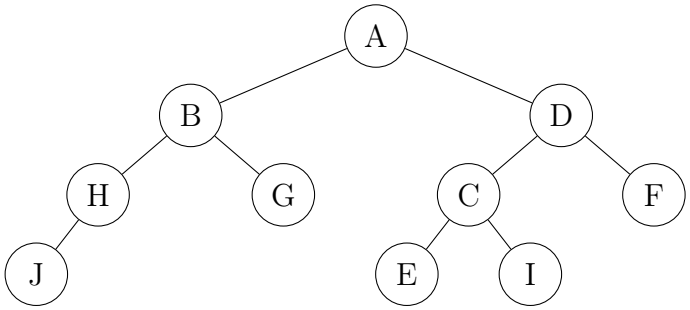
2. Arbres binaires

Exercice 2
Dessiner tous les arbres binaires ayant respectivement 3 et 4 nœuds.

Exercice 3
Sachant qu'il y a 1 arbre binaire vide, 1 arbre binaire contenant 1 nœud, 2 arbres binaires contenant 2 nœuds, 5 arbres binaires contenant 3 nœuds et 14 arbres binaires contenant 4 nœuds, calculer le nombre d'arbres binaires contenant 5 nœuds.
On ne cherchera pas à les construire tous mais seulement à les dénombrer.

Exercice 4
Donner cinq arbres de taille 3, différents, dont les nœuds contiennent les valeurs 1, 2 et 3 et pour lesquels un parcours infixe traite les nœuds dans l'ordre : `1 2 3`

Exercice 5
On considère l'arbre ci-dessous :



Donner l'ordre dans lequel les nœuds sont traités dans chacun des cas suivants :

1. Avec un parcours en profondeur d'abord suivant l'ordre préfixe.
2. Avec un parcours en profondeur d'abord suivant l'ordre infixe.
3. Avec un parcours en profondeur d'abord suivant l'ordre postfixe.
4. Avec un parcours en largeur d'abord.

Pour la suite on reprend la classe `Noeud` vue en cours :

```
class Noeud:
    '''un nœud d'un arbre binaire'''
    def __init__(self, v, g, d):
        self.valeur = v
        self.gauche = g
        self.droit = d
```

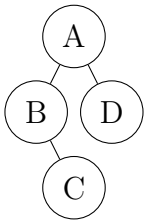
Exercice 6

1. Écrire une fonction `est_feuille(arb)` qui prend en paramètre un arbre binaire et retourne `True` si l'arbre est réduit à une feuille et `False` sinon.
2. Écrire une fonction `nb_feuilles(arb)` qui prend en paramètre un arbre binaire et retourne le nombre de feuilles de l'arbre.
3. Écrire une fonction `nb_noeud_int(arb)` qui prend en paramètre un arbre binaire et retourne le nombre de nœuds internes de l'arbre.
4. Écrire une fonction `parcours_feuilles(arb)` qui prend en paramètre un arbre binaire et affiche les valeurs des feuilles.

Exercice 7
Écrire une fonction `appartient(arb,v)` qui retourne `True` si la valeur `v` appartient à l'arbre binaire `arb` et `False` sinon. Quelle est la complexité de cet algorithme ?

Exercice 8

1. Écrire la fonction `affiche(arb)` qui imprime un arbre binaire sous la forme suivante : pour un arbre vide, on n'imprime rien ; pour un nœud, on imprime une parenthèse ouvrante, son sous-arbre gauche (récursivement), sa valeur, son sous-arbre droit (récursivement), puis enfin une parenthèse fermante. Ainsi, pour l'arbre ci-dessous on doit afficher `((B(C))A(D))`.



2. Dessiner l'arbre binaire sur lequel la fonction `affiche` produit la sortie `(1((2)3))`.
3. Écrire la fonction `chaine(arb)` qui **retourne** la chaîne de caractère au lieu de l'afficher.

Exercice 9

1. Ajouter à la classe `Noeud` une méthode `__eq__` permettant de tester l'égalité entre deux arbres binaires à l'aide de l'opération ==.
2. Ajouter également une méthode `__str__` qui retourne la chaîne de caractère décrite dans l'exercice 8. On rappelle que cette fonction permet l'utilisation de la fonction `str`, mais aussi de `print`.

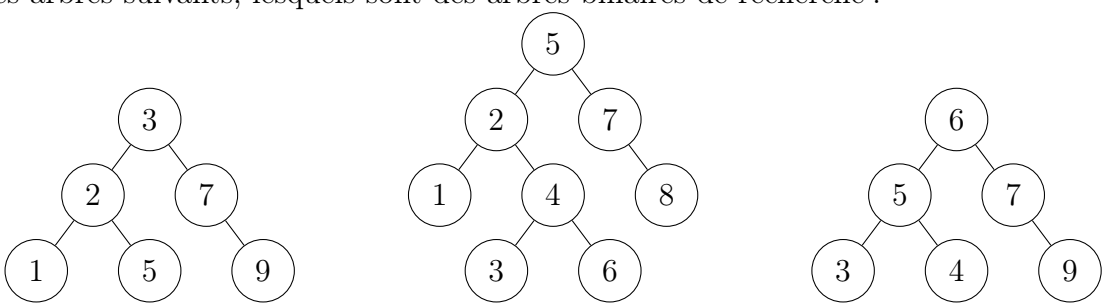
Exercice 10

Écrire une fonction récursive `parfait(h)` qui reçoit en argument un entier positif `h` et retourne un arbre binaire parfait de hauteur `h`.

3. Arbres binaires de recherche

Exercice 11

Parmi les arbres suivants, lesquels sont des arbres binaires de recherche ?



Exercice 12

Donner tous les arbres binaires de recherche formés de trois nœuds et contenant les entiers 1, 2 et 3.

Exercice 13

1. Dans un arbre binaire de recherche, où se trouve le plus petit élément ? en déduire une fonction `minimum(arb)` qui retourne le plus petit élément de l'ABR `arb` et `None` si cet arbre est vide.
2. Écrire une fonction `maximum(arb)` qui retourne le plus grand élément de l'ABR `arb` et `None` si cet arbre est vide.

Exercice 14

Écrire une variante de la fonction `ajout(arb, v)` qui n'ajoute pas `v` à `arb` si `v` est déjà présent dans `arb`.

Exercice 15

Écrire une fonction `est_ABR(arb)` qui retourne `True` si `arb` est un arbre binaire de recherche et `False` sinon.

Exercice 16

1. Écrire une fonction `remplir(arb, tbl)` qui ajoute tous les éléments de l'ABR `arb` dans le tableau `tbl` dans l'ordre infixe. Utiliser la méthode `append` pour ajouter les éléments au tableau.
2. On reprend la définition de la classe `ABR` vue en cours, avec la fonction spécifique qui teste l'appartenance à un arbre binaire de recherche :

```
def appartient(arb, v):
  '''détermine si la valeur v appartient à l'ABR arb'''
  if arb is None:
    return False
  elif arb.valeur == v:
    return True
  elif v < arb.valeur:
    return appartient(arb.gauche, v)
  else:
    return appartient(arb.droite, v)

class ABR:
    '''un arbre binaire de recherche'''
    def __init__(self):
        self.racine = None
    def ajouter(self, v):
        self.racine = ajout(self.racine, v)
    def contient(self, v):
        return appartient(self.racine, v)
```

Pour la fonction `ajout` on pourra reprendre le code défini dans l'exercice 14.

Ajouter alors une méthode `lister(self)` à la classe `ABR` qui retourne un nouveau tableau contenant tous les éléments de l'ABR `self` dans l'ordre croissant.

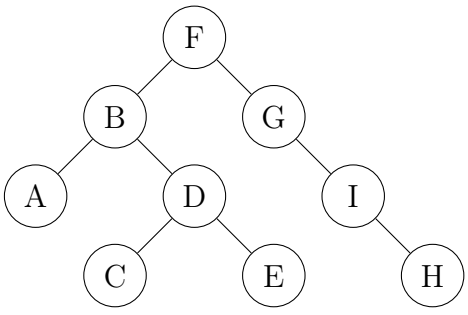
3. Écrire finalement une fonction `trier(tbl)` qui reçoit en argument un tableau d'entiers et retourne un tableau trié contenant les mêmes éléments.

4. Épreuve pratique

Exercice 17

Dans cet exercice, un arbre binaire de caractères est stocké sous la forme d'un dictionnaire où les clefs sont les caractères des nœuds de l'arbre et les valeurs, pour chaque clef, la liste des caractères des fils gauche et droit du nœud.

Par exemple, l'arbre



est stocké dans

```
a = {'F':['B','G'], 'B':['A','D'], 'A':['',''], 'D':['C','E'], \
     'C':['',''], 'E':['',''], 'G':['','I'], 'I':['','H'], 'H':['','']}
```

Écrire une fonction récursive `taille` prenant en paramètres un arbre binaire `arbre` sous la forme d'un dictionnaire et un caractère `lettre` qui est la valeur du sommet de l'arbre, et qui renvoie la taille de l'arbre à savoir le nombre total de nœud.

On pourra distinguer les 4 cas où les deux « fils » du nœud sont `''`, le fils gauche seulement est `''`, le fils droit seulement est `''`, aucun des deux fils n'est `''`.
Exemple :

```
>>> taille(a, 'F')
9
```

5. Pour aller plus loin

Voir et travailler le fichier `Ex08_arbres_Huffman.ipynb` sur le codage de Huffman.

6. Annales

EXERCICE 5 (4 points)

Cet exercice traite du thème « algorithmique », et principalement des algorithmes sur les arbres binaires.

On manipule ici les arbres binaires avec trois fonctions :

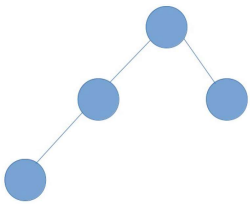
- `est_vide(A)` qui renvoie `True` si l'arbre binaire `A` est vide, `False` s'il ne l'est pas ;
- `sous_arbre_gauche(A)` qui renvoie le sous-arbre gauche de l'arbre binaire `A` ;
- `sous_arbre_droit(A)` qui renvoie le sous-arbre droit de l'arbre binaire `A`.

L'arbre binaire renvoyé par les fonctions `sous_arbre_gauche` et `sous_arbre_droit` peut éventuellement être l'arbre vide.

On définit la **hauteur** d'un arbre binaire non vide de la façon suivante :

- si ses sous-arbres gauche et droit sont vides, sa hauteur est 0 ;
- si l'un des deux au moins est non vide, alors sa hauteur est égale à 1 + M, où M est la plus grande des hauteurs de ses sous-arbres (gauche et droit) non vides.

1. a. Donner la hauteur de l'arbre ci-dessous.



b. Dessiner sur la copie un arbre binaire de hauteur 4.

La hauteur d'un arbre est calculée par l'algorithme récursif suivant :

```
1   Algorithme hauteur(A):
2     test d'assertion : A est supposé non vide
3     si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
4       renvoyer 0
5     sinon, si sous_arbre_gauche(A) vide:
6       renvoyer 1 + hauteur(sous_arbre_droit(A))
7     sinon, si ... :
8       renvoyer ...
9     sinon:
10      renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
11                       hauteur(sous_arbre_droit(A)))
```

2. Recopier sur la copie les lignes 7 et 8 en complétant les points de suspension.

3. On considère un arbre binaire R dont on note G le sous-arbre gauche et D le sous-arbre droit. On suppose que R est de hauteur 4 et G de hauteur 2.

   a. Justifier le fait que D n'est pas l'arbre vide et déterminer sa hauteur.

   b. Illustrer cette situation par un dessin.

Soit un arbre binaire non vide de hauteur h. On note n le nombre de nœuds de cet arbre. On admet que $h+1 \leqslant n \leqslant 2^{h+1}-1$.

4. a. Vérifier ces inégalités sur l'arbre binaire de la question 1.a.

   b. Expliquer comment construire un arbre binaire de hauteur h quelconque ayant h+1 nœuds.

   c. Expliquer comment construire un arbre binaire de hauteur h quelconque ayant $2^{h+1}-1$ nœuds.

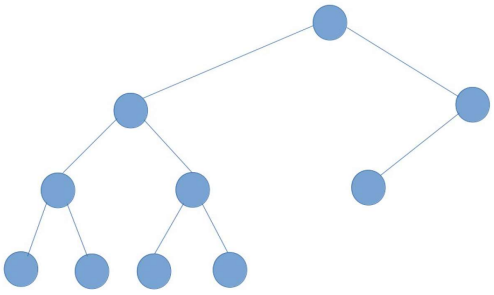
   Indication : $2^{h+1}-1 = 1+2+4+...+2^h$.

L'objectif de la fin de l'exercice est d'écrire le code d'une fonction `fabrique(h, n)` qui prend comme paramètres deux nombres entiers positifs `h` et `n` tels que $h+1 < n < 2^{h+1}-1$, et qui renvoie un arbre binaire de hauteur `h` à `n` nœuds.

Pour cela, on a besoin des deux fonctions suivantes:

- `arbre_vide()`, qui renvoie un arbre vide ;
- `arbre(gauche, droit)` qui renvoie l'arbre de fils gauche `gauche` et de fils droit `droit`.

5. Recopier sur la copie l'arbre binaire ci-dessous et numéroter ses nœuds de 1 en 1 en commençant à 1, en effectuant un parcours en profondeur préfixe.



La fonction `fabrique` ci-dessous a pour but de répondre au problème posé. Pour cela, la fonction `annexe` utilise la valeur de `n`, qu'elle peut modifier, et renvoie un arbre binaire de hauteur `hauteur_max` dont le nombre de nœuds est égal à la valeur de `n` au moment de son appel.

```
1.  def fabrique(h, n):
2.    def annexe(hauteur_max):
3.      if n == 0 :
4.        return arbre_vide()
5.      elif hauteur_max == 0:
6.        n = n - 1
7.        return ...
8.      else:
9.        n = n - 1
10.       gauche = annexe(hauteur_max - 1)
11.       droite = ...
12.       return arbre(gauche, droite)
13.   return annexe(h)
```

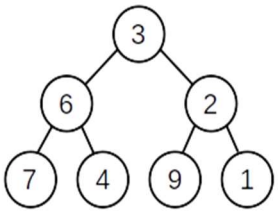
6. Recopier sur la copie les lignes 7 et 11 en complétant les points de suspension.

EXERCICE 4 (4 points)

Cet exercice, composé de deux parties A et B, porte sur le parcours des arbres binaires, le principe "diviser pour régner" et la récursivité.

Cet exercice traite du calcul de la somme d'un arbre binaire. Cette somme consiste à additionner toutes les valeurs numériques contenues dans les nœuds de l'arbre.

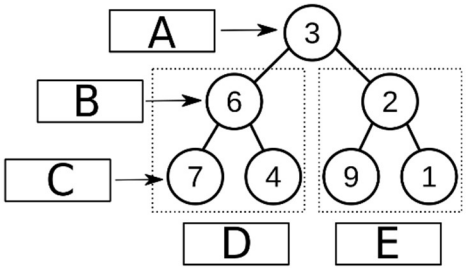
L'arbre utilisé *dans les parties A et B est le suivant :*



Partie A : Parcours d'un arbre

1. Donner la somme de l'arbre précédent. Justifier la réponse en explicitant le calcul qui a permis de l'obtenir.

2. Indiquer la lettre correspondante aux noms 'racine', 'feuille', 'nœud', 'SAG' (Sous Arbre Gauche) et 'SAD' (Sous Arbre Droit). Chaque lettre **A**, **B**, **C**, **D** et **E** devra être utilisée une seule fois.



Arbre avec les lettres à associer

3. Parmi les quatre propositions A, B, C et D ci-dessous, donnant un parcours en largeur d'abord de l'arbre, une seule est correcte. Indiquer laquelle.
Proposition A : 7 - 6 - 4 - 3 - 9 - 2 - 1
Proposition B : 3 - 6 - 7 - 4 - 2 - 9 - 1
Proposition C : 3 - 6 - 2 - 7 - 4 - 9 - 1
Proposition D : 7 - 4 - 6 - 9 - 1 - 2 – 3

4. Écrire en langage Python la fonction `somme` qui prend en paramètre une liste de nombres et qui renvoie la somme de ses éléments.
Exemple : `somme([1, 2, 3, 4])` est égale à 10.

5. La fonction `parcourir(arbre)` pourrait se traduire en langage naturel par :

```
parcourir(A):
    L = liste_vide
    F = file_vide
    enfiler A dans F
    Tant que F n'est pas vide
        défiler S de F
        ajouter la valeur de la racine de S dans L
        Pour chaque sous arbre SA non vide de S
            enfiler SA dans F
    renvoyer L
```

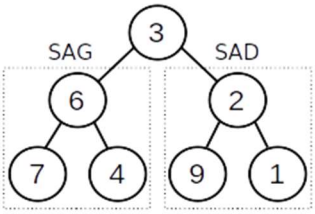
Donner le type de parcours obtenu grâce à la fonction `parcourir`.

Partie B : Méthode 'diviser pour régner'

6. Parmi les quatre propositions A,B, C et D ci-dessous, indiquer la seule proposition correcte.

En informatique, le principe diviser pour régner signifie :
Proposition A : diviser une fonction en deux fonctions plus petites
Proposition B : utiliser plusieurs modules
Proposition C : séparer les informations en fonction de leur types
Proposition D : diviser un problème en deux problèmes plus petits et indépendants.

7. L'arbre présenté dans le problème peut être décomposé en racine et sous arbres :



Indiquer dans l'esprit de 'diviser pour régner' l'égalité donnant la somme d'un arbre en fonction de la somme des sous arbres et de la valeur numérique de la racine.

8. Écrire en langage Python une fonction récursive `calcul_somme(arbre)`. Cette fonction calcule la somme de l'arbre passé en paramètre.

Les fonctions suivantes sont disponibles :

- `est_vide(arbre)` : renvoie `True` si `arbre` est vide et renvoie `False` sinon ;
- `valeur_racine(arbre)` : renvoie la valeur numérique de la racine de `arbre` ;
- `arbre_gauche(arbre)` : renvoie le sous arbre gauche de `arbre` ;
- `arbre_droit(arbre)` : renvoie le sous arbre droit de `arbre`.

EXERCICE 1 (4 points)

Cet exercice porte sur les arbres binaires de recherche, la programmation orientée objet et la récursivité.

Dans cet exercice, la taille d'un arbre est le nombre de nœuds qu'il contient. Sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles (nœuds sans sous-arbres). On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et la hauteur de l'arbre vide vaut 0.

1. On considère l'arbre binaire représenté ci-dessous:

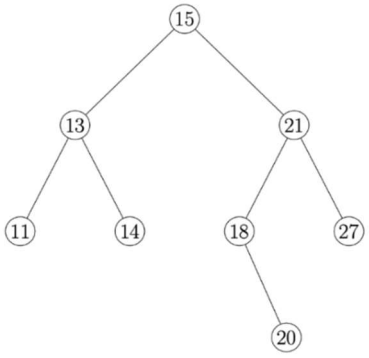


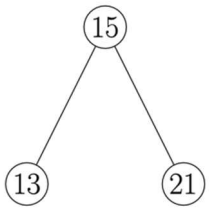
Figure 1

a. Donner la taille de cet arbre.
b. Donner la hauteur de cet arbre.
c. Représenter sur la copie le sous-arbre droit du nœud de valeur 15.
d. Justifier que l'arbre de la figure 1 est un arbre binaire de recherche.
e. On insère la valeur 17 dans l'arbre de la figure 1 de telle sorte que 17 soit une nouvelle feuille de l'arbre et que le nouvel arbre obtenu soit encore un arbre binaire de recherche. Représenter sur la copie ce nouvel arbre.

2. On considère la classe `Noeud` définie de la façon suivante en Python :

```
1  class Noeud:
2     def __init__(self, g, v, d):
3        self.gauche = g
4        self.valeur = v
5        self.droit = d
```

a. Parmi les trois instructions **(A)**, **(B)** et **(C)** suivantes, écrire sur la copie la lettre correspondant à celle qui construit et stocke dans la variable `abr` l'arbre représenté ci-contre.



(A) `abr=Noeud(Noeud(Noeud(None,13,None),15,None),21,None)`

(B) `abr=Noeud(None,13,Noeud(Noeud(None,15,None),21,None))`

(C) `abr=Noeud(Noeud(None,13,None),15,Noeud(None,21,None))`

b. Recopier et compléter la ligne 7 du code de la fonction `ins` ci-dessous qui prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et qui renvoie l'arbre obtenu suite à l'insertion de la valeur `v` dans l'arbre `abr`. Les lignes 8 et 9 permettent de ne pas insérer la valeur `v` si celle-ci est déjà présente dans `abr`.

```
1  def ins(v, abr):
2      if abr is None:
3          return Noeud(None, v, None)
4      if v > abr.valeur:
5          return Noeud(abr.gauche,abr.valeur,ins(v,abr.droit))
6      elif v < abr.valeur:
7          return ........................................
8      else:
9          return abr
```

3. La fonction `nb_sup` prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et renvoie le nombre de valeurs supérieures ou égales à la valeur `v` dans l'arbre `abr`.

Le code de cette fonction `nb_sup` est donné ci-dessous :

```
1  def nb_sup(v, abr):
2      if abr is None:
3          return 0
4      else:
5          if abr.valeur >= v:
6              return 1+nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
7          else:
8              return nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
```

a. On exécute l'instruction `nb_sup(16, abr)` dans laquelle `abr` est l'arbre initial de la figure 1. Déterminer le nombre d'appels à la fonction `nb_sup`.

b. L'arbre passé en paramètre étant un arbre binaire de recherche, on peut améliorer la fonction `nb_sup` précédente afin de réduire ce nombre d'appels. Écrire sur la copie le code modifié de cette fonction.