EXERCICE 1 (10 points)

Cet exercice porte sur les structures de données (listes, p-uplets et dictionnaires).
On dispose de la liste `jours` suivante et du dictionnaire `mois` suivant :

```
jours=["dimanche", "lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi"]

mois={1 :("janvier",31) , 2 :("février",28) , 3 :("mars",31),
4 :("avril",30) , 5 :("mai",31) , 6 :("juin",30) ,
7 :("juillet",31) , 8 :("aout",31) , 9 :("septembre",30) ,
10 :("octobre",31) , 11 :("novembre",30) ,
12 :("décembre",31)}
```

1.

a. A partir de la liste `jours`, comment obtenir l'élément `"lundi"` ?

b. On rappelle que l'opérateur % (« modulo ») renvoie le reste de la division entière (division euclidienne).

Exemple : 7%3 renvoie 1 qui est le reste de la division de 7 par 3 :

```
 7 | 3
-6 | 2
---
(1)|
```

Que renvoie l'instruction `jours[18%7]` ?

2. On rappelle que `jours.index[element]` renvoie l'indice de `element` dans la liste `jours` par exemple `jours.index["mercredi"]` renvoie 3.

Le nom du jour actuel est stocké dans une variable `j` (par exemple : `j = "mardi"`).
Recopier et compléter l'instruction suivante permettant d'obtenir le numéro du jour de la semaine `n` jours plus tard :
`numero_jour =(jours.index[ … ] + … )% …`

3.

a. A partir du dictionnaire `mois`, comment obtenir le nombre de jours du mois de mars ?

b. Le numéro du mois actuel est stocké dans une variable `numero_mois`, écrire le code permettant d'obtenir le nom du mois qu'il sera `x` mois plus tard à partir du dictionnaire `mois`.
Par exemple :
si `numero_mois = 4 et x = 5`, on doit obtenir `"septembre"`
si `numero_mois = 10 et x = 3`, on doit obtenir `"janvier"`

4. On définit une date comme un tuple :
`(nom_jour,numero_jour, numero_mois,annee).`

a. Sachant que `date = ("samedi", 21, 10, 1995),` que renvoie `mois[date[2]][1] ?`

b. Ecrire une fonction `jour_suivant(date)` qui prend en paramètre une date sous forme de tuple et qui renvoie un tuple désignant la date du lendemain.
Par exemple :
`jour_suivant( ("samedi", 21, 10, 1995) )` renvoie `("dimanche", 22, 10, 1995)`

`jour_suivant( ("mardi", 31, 10, 1995) )` renvoie `("mercredi", 1, 11, 1995)`

On ne tient pas compte des années bissextiles et on considère que le mois de février comporte toujours 28 jours.

Exercice 2 (10 points)

Cet exercice porte sur les structures de données (listes, piles et files).

On cherche ici à mettre en place des algorithmes qui permettent de modifier l'ordre des informations contenues dans une file. On considère pour cela les structures de données abstraites de Pile et File définies par leurs fonctions primitives suivantes :

Pile :

- `creer_pile_vide()` renvoie une pile vide ;
- `est_pile_vide(p)` renvoie `True` si la pile `p` est vide, `False` sinon ;
- `empiler(p, element)` ajoute `element` au sommet de la pile `p` ;
- `depiler(p)` renvoie l'élément se situant au sommet de la pile `p` en le retirant de la pile `p` ;
- `sommet(p)` renvoie l'élément se situant au sommet de la pile `p` sans le retirer de la pile `p`.

File :

- `creer_file_vide()` renvoie une file vide ;
- `est_file_vide(f)` renvoie `True` si la file `f` est vide, `False` sinon ;
- `enfiler(f, element)` ajoute `element` dans la file `f` ;
- `defiler(f)` renvoie l'élément à la tête de la file `f` en le retirant de la file `f`.

On considère de plus que l'on dispose d'une fonction permettant de connaître le nombre d'éléments d'une file :

- `taille_file(f)` renvoie le nombre d'éléments de la file `f`.

On représentera les files par des éléments en ligne, l'élément de droite étant la tête de la file et l'élément de gauche étant la queue de la file. On représentera les piles en colonnes, le sommet de la pile étant le haut de la colonne.

La file suivante est appelée `f` :

4	3	8	2	1

La pile suivante est appelée `p` :

5
8
6
2

1. Les quatre questions suivantes sont indépendantes. Pour chaque question, on repartira de la pile `p` et de la file `f` initiales (présentées ci-dessus)

a. Représenter la file `f` après l'exécution du code suivant.

```
enfiler(f, defiler(f))
```

b. Représenter la pile `p` après l'exécution du code suivant.

```
empiler(p, depiler(p))
```

c. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

```
for i in range(2):
  enfiler(f, depiler(p))
```

d. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

```
for i in range(2):
    empiler(p, defiler(f))
```

2. On donne ici une fonction `mystere` qui prend une file en argument, qui modifie cette file, mais qui ne renvoie rien.

```
def mystere(f):
    p = creer_pile_vide()
    while not est_file_vide(f):
        empiler(p, defiler(f))
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
    return p
```

Préciser l'état de la variable `f` après chaque boucle de la fonction `mystere` appliquée à la file | 1 | 2 | 3 | 4 | Indiquer le contenu de la pile renvoyée.par la fonction.

3. On considère la fonction `knuth(f)` suivante dont le paramètre est une file :

```
def knuth(f):
    p=creer_pile_vide()
    N=taille_file(f)
    for i in range(N):
        if est_pile_vide(p):
            empiler(p, defiler(f))
        else:
            e = defiler(f)
            if e >= sommet(p):
                empiler(p, e)
            else:
                while not est_pile_vide(p) and e < sommet(p):
                    enfiler(f, depiler(p))
                empiler(p, e)
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
```

a. Recopier et compléter le tableau ci-dessous qui détaille le fonctionnement de cet algorithme étape par étape pour la file `2,1,3`.
Une étape correspond à une modification de la pile ou de la file.
Le nombre de colonnes peut bien sûr être modifié.

f	2,1,3	2,1							
p	(pile vide)	3							

b. Que fait cet algorithme ?

EXERCICE 3 (10 points)

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

On manipule ici les arbres binaires avec trois fonctions :

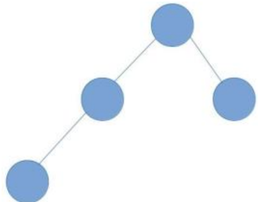
- `est_vide(A)` qui renvoie `True` si l'arbre binaire `A` est vide, `False` s'il ne l'est pas ;
- `sous_arbre_gauche(A)` qui renvoie le sous-arbre gauche de l'arbre binaire `A` ;
- `sous_arbre_droit(A)` qui renvoie le sous-arbre droit de l'arbre binaire `A`.

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.

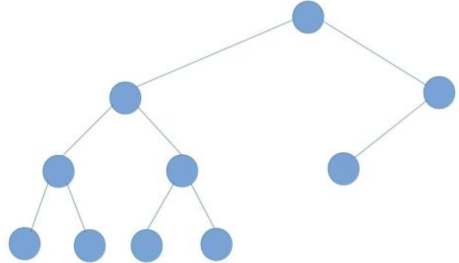
   c. Expliquer comment construire un arbre binaire de hauteur h quelconque ayant $2^{h+1}-1$ nœuds.
   Indication : $2^{h+1}-1 = 1+2+4+...+2^h$.

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 1 (10 points)

Cet exercice porte sur les structures de données (listes, p-uplets et dictionnaires).
On dispose de la liste `jours` suivante et du dictionnaire `mois` suivant :

```
jours=["dimanche", "lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi"]

mois={1 :("janvier",31) , 2 :("février",28) , 3 :("mars",31),
4 :("avril",30) , 5 :("mai",31) , 6 :("juin",30) ,
7 :("juillet",31) , 8 :("aout",31) , 9 :("septembre",30) ,
10 :("octobre",31) , 11 :("novembre",30) ,
12 :("décembre",31)}
```

1.

a. A partir de la liste `jours`, comment obtenir l'élément `"lundi"` ?

On obtient lundi avec jours[1]

b. On rappelle que l'opérateur % (« modulo ») renvoie le reste de la division entière (division euclidienne).

Exemple : 7%3 renvoie 1 qui est le reste de la division de 7 par 3 :

```
 7 | 3
-6 | 2
 1 |
```

Que renvoie l'instruction `jours[18%7]` ?

numero_jour =(jours.index(j) + n)% 7

2. On rappelle que `jours.index(element)` renvoie l'indice de `element` dans la liste `jours` par exemple `jours.index["mercredi"]` renvoie 3.

Le nom du jour actuel est stocké dans une variable `j` (par exemple : `j = "mardi"`).
Recopier et compléter l'instruction suivante permettant d'obtenir le numéro du jour de la semaine `n` jours plus tard :
`numero_jour =(jours.index[ … ] + … )% …`

numero_jour =(jours.index(j) + n)% 7

3.

a. A partir du dictionnaire `mois`, comment obtenir le nombre de jours du mois de mars ?

Pour obtenir le nombre de jours au mois de mars, on doit écrire : mois[3][1]

b. Le numéro du mois actuel est stocké dans une variable `numero_mois`, écrire le code permettant d'obtenir le nom du mois qu'il sera `x` mois plus tard à partir du dictionnaire `mois`.

Par exemple :

si `numero_mois = 4 et x = 5`, on doit obtenir `"septembre"`
si `numero_mois = 10 et x = 3`, on doit obtenir `"janvier"`

```
def nom_mois(numero_mois,x):
    num_mois = (numero_mois+x)%12
    if num_mois == 0 :
        return mois[12][0]
    else :
    return mois[num_mois][0]
```

4. On définit une date comme un tuple :
`(nom_jour,numero_jour, numero_mois,annee).`

a. Sachant que `date = ("samedi", 21, 10, 1995)`, que renvoie `mois[date[2]][1] ?`

mois[date[2]][1] renvoie 31

b. Ecrire une fonction `jour_suivant(date)` qui prend en paramètre une date sous forme de tuple et qui renvoie un tuple désignant la date du lendemain.

Par exemple :

`jour_suivant( ("samedi", 21, 10, 1995) )` renvoie `("dimanche", 22, 10, 1995)`

`jour_suivant( ("mardi", 31, 10, 1995) )` renvoie `("mercredi", 1, 11, 1995)`

On ne tient pas compte des années bissextiles et on considère que le mois de février comporte toujours 28 jours.

```
def jour_suivant(date):
    annee = date[3]
    no_mois = date[2]
    no_jour = date[1]
    jour_semaine = date[0]
    nombre_jours_du_mois = mois[no_mois][1]
    if no_jour == nombre_jours_du_mois:
        no_jour_suivant = 1
        if no_mois == 12:
            no_mois_suivant = 1
            annee_suivante = annee + 1
        else:
            no_mois_suivant = no_mois
            annee_suivante = annee
    else:
        no_jour_suivant = no_jour + 1
        no_mois_suivant = no_mois
        annee_suivante = annee
    jour_semaine_suivant = jours[(jours.index(jour_semaine) + 1) % 7]
    return (jour_semaine_suivant,
no_jour_suivant,no_mois_suivant,annee_suivante)
```

Exercice 5 8 6 2 2 (10 points)

Cet exercice porte sur les structures de données (listes, piles et files).

On cherche ici à mettre en place des algorithmes qui permettent de modifier l'ordre des informations contenues dans une file. On considère pour cela les structures de données abstraites de Pile et File définies par leurs fonctions primitives suivantes :

Pile :

- `creer_pile_vide()` renvoie une pile vide ;
- `est_pile_vide(p)` renvoie `True` si la pile `p` est vide, `False` sinon ;
- `empiler(p, element)` ajoute `element` au sommet de la pile `p` ;
- `depiler(p)` renvoie l'élément se situant au sommet de la pile `p` en le retirant de la pile `p` ;
- `sommet(p)` renvoie l'élément se situant au sommet de la pile `p` sans le retirer de la pile `p`.

File :

- `creer_file_vide()` renvoie une file vide ;
- `est_file_vide(f)` renvoie `True` si la file `f` est vide, `False` sinon ;
- `enfiler(f, element)` ajoute `element` dans la file `f` ;
- `defiler(f)` renvoie l'élément à la tête de la file `f` en le retirant de la file `f`.

On considère de plus que l'on dispose d'une fonction permettant de connaître le nombre d'éléments d'une file :

- `taille_file(f)` renvoie le nombre d'éléments de la file `f`.

On représentera les files par des éléments en ligne, l'élément de droite étant la tête de la file et l'élément de gauche étant la queue de la file. On représentera les piles en colonnes, le sommet de la pile étant le haut de la colonne.

La file suivante est appelée `f` :

4	3	8	2	1

La pile suivante est appelée `p` :

5
8
6
2

1. Les quatre questions suivantes sont indépendantes. Pour chaque question, on repartira de la pile `p` et de la file `f` initiales (présentées ci-dessus)

   a. Représenter la file `f` après l'exécution du code suivant.
   `enfiler(f, defiler(f))`

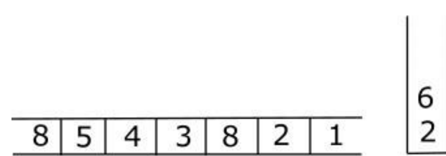
1	4	3	8	2

   b. Représenter la pile `p` après l'exécution du code suivant.
   `empiler(p, depiler(p))`

   5
   8
   6
   2

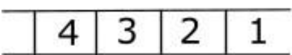
c. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

```
for i in range(2):
  enfiler(f, depiler(p))
```



d. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

```
for i in range(2):
    empiler(p, defiler(f))
```



2. On donne ici une fonction `mystere` qui prend une file en argument, qui modifie cette file, mais qui ne renvoie rien.

```
def mystere(f):
    p = creer_pile_vide()
    while not est_file_vide(f):
        empiler(p, defiler(f))
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
    return p
```

Préciser l'état de la variable `f` après chaque boucle de la fonction `mystere` appliquée à la file | 1 | 2 | 3 | 4 | Indiquer le contenu de la pile renvoyée.par la fonction.

Contenu de la file f :

4	3	2	1

3. On considère la fonction `knuth(f)` suivante dont le paramètre est une file :

```
def knuth(f):
    p=creer_pile_vide()
    N=taille_file(f)
    for i in range(N):
        if est_pile_vide(p):
            empiler(p, defiler(f))
        else:
            e = defiler(f)
            if e >= sommet(p):
                empiler(p, e)
            else:
                while not est_pile_vide(p) and e < sommet(p):
                    enfiler(f, depiler(p))
                empiler(p, e)
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
```

a. Recopier et compléter le tableau ci-dessous qui détaille le fonctionnement de cet algorithme étape par étape pour la file `2,1,3`.
Une étape correspond à une modification de la pile ou de la file.
Le nombre de colonnes peut bien sûr être modifié.

f	2,1,3	2,1	3,2	3	2,3	1,2,3
p	(vide)	3	1	2 1	1	(vide)

Que fait cet algorithme ?

Cet algorithme permet de trier dans l'ordre décroissant (plus grand élément en tête de file) une file composée de 3 éléments (pour 4 éléments ou plus cela ne fonctionne pas !?). Si on prend une file contenant au départ plus de 3 éléments, on peut dire que cet algorithme permet de mélanger cette file.

EXERCICE 3 (10 points)

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

On manipule ici les arbres binaires avec trois fonctions :

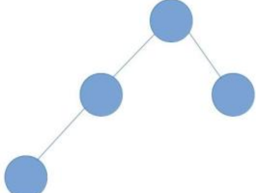
- `est_vide(A)` qui renvoie `True` si l'arbre binaire `A` est vide, `False` s'il ne l'est pas ;
- `sous_arbre_gauche(A)` qui renvoie le sous-arbre gauche de l'arbre binaire `A` ;
- `sous_arbre_droit(A)` qui renvoie le sous-arbre droit de l'arbre binaire `A`.

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 :

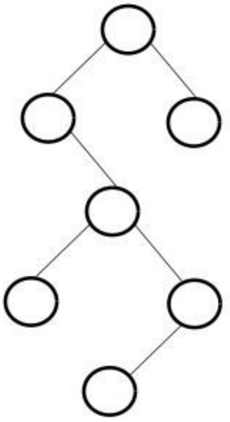
- si ses sous-arbres gauche et droit sont vides, sa hauteur est 0 ;
- si l'un des deux au moins est non vide, alors sa hauteur est égale à 1 + M, où M est la plus grande des hauteurs de ses sous-arbres (gauche et droit) non vides.

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



La hauteur de l'arbre est de 2

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



CORRECTION

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

```
12 Algorithme hauteur(A):
13   test d'assertion : A est supposé non vide
14   si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
15     renvoyer 0
16   sinon, si sous_arbre_gauche(A) vide:
17     renvoyer 1 + hauteur(sous_arbre_droit(A))
18   sinon, si ... :
19     renvoyer ...
20   sinon:
21      renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
22                       hauteur(sous_arbre_droit(A)))
```

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

```
Algorithme hauteur(A):
si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide: renvoyer 0
sinon, si sous_arbre_gauche(A) vide:
        renvoyer 1 + hauteur(sous_arbre_droit(A))
sinon, si sous_arbre_droit(A) vide :
        renvoyer 1 + hauteur(sous_arbre_gauche(A))
sinon:
        renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
        hauteur(sous_arbre_droit(A)))
```

3.

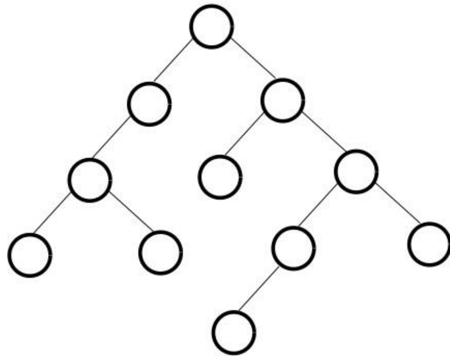
On considère un arbre binaire R dont on note G le sous-arbre gauche et D le sous-arbre droit. On suppose que R est de hauteur 4 et G de hauteur 2.

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

Si D est l'arbre vide, la hauteur de R est égale à 1 + hauteur(G) soit 1+2=3, comme la hauteur de R est 4, D ne peut pas être l'arbre vide .

Comme ni D et ni G sont des arbres vides, hauteur(R) = 1+max(hauteur(G), hauteur(D)), d'où hauteur(R) = 1+max(2,hauteur(D)) avec hauteur(R) = 4, hauteur(D) est obligatoirement égale à 3.

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 $h+1 \leqslant n \leqslant 2^{h+1}-1$.

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

L'arbre de la question 1.a. possède 4 nœuds d'où n = 4 Ce même arbre a une hauteur de 2 d'où h = 2
D'où h+1 = 3 et $2^{h+1} - 1 = 2^3 - 1 = 7$
Nous avons bien 4 qui est compris entre 3 et 7, les inégalités sont donc vérifiées.

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

Pour créer un arbre binaire de hauteur h quelconque ayant h+1 nœuds, il suffit de créer un arbre où chaque nœud, a au maximum un enfant.

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

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

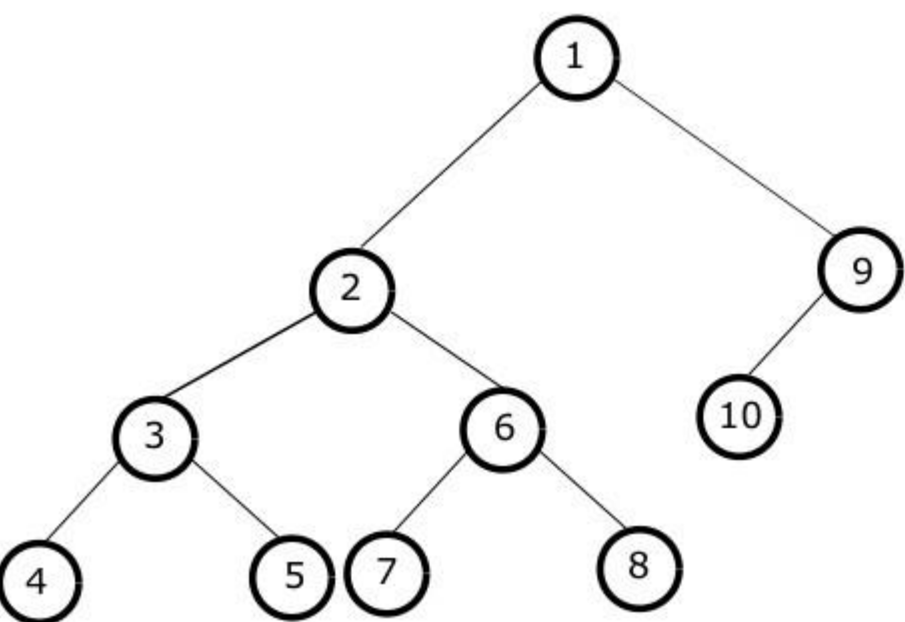
Pour créer un arbre binaire de hauteur h quelconque ayant $2^{h+1} - 1$ nœuds, il suffit de créer un arbre où chaque nœud, qui n'est pas une feuille, a 2 enfants.

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.

```
3.  def fabrique(h, n):
4.    def annexe(hauteur_max):
3.    if n == 0 :
6.        return arbre_vide()
7.      elif hauteur_max == 0:
6.      n = n - 1
9.        return ...
10.     else:
9.        n = n - 1
14.       gauche = annexe(hauteur_max - 1)
15.       droite = ...
16.       return arbre(gauche, droite)
17.   return annexe(h)
```

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

```
def fabrique(h, n):
  def annexe(hauteur_max): if n
    == 0 :
      return arbre_vide() elif
    hauteur_max == 0:
      n=n-1
      return arbre(arbre_vide(), arbre_vide()) else:
      n=n-1
      gauche = annexe(hauteur_max - 1) droite
      =   annexe(hauteur_max   -   1)   return
      arbre(gauche, droite)
  return annexe(h)
```