STRUCTURES DE DONNÉES : ARBRES

I . Généralités sur les arbres

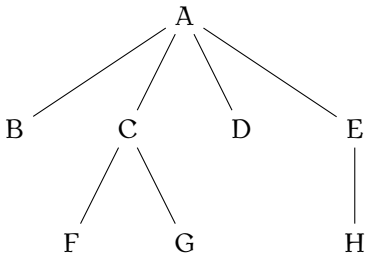
II . Représentation des arbres en Python à l'aide de listes

On peut donner d'un arbre la définition récursive suivante : un arbre est une liste formée de

- une *racine*, qui contient une information spécifique, qui dépend du problème considéré (la *clé*)
- et d'une suite, éventuellement vide, d'arbres ;

On en déduit immédiatement une manière de représenter un arbre en Python : une liste formée d'une donnée (la racine) suivie (éventuellement) d'une suite de listes bâties selon le même principe, qui représentent les arbres-fils de la racine.

Par exemple, l'arbre :



sera représenté par la liste :

```
['A', ['B'] , ['C',['F'],['G']] , ['D'] , ['E',['H']]]
```

■ *Exercice 1 :*

Écrire une fonction `afficher(arbre,marge)` qui affiche l'arbre sous une forme plus facile à lire que l'expression précédente : chaque information est sur une ligne, avec une marge à gauche qui reflète la position hiérarchique (on dit la profondeur) de l'élément.

Plus précisément, cette fonction affiche l'information portée par la racine de l'arbre en laissant à gauche la marge indiquée, puis se rappelle elle-même sur chaque sous-arbre avec une marge augmentée.

Solution :

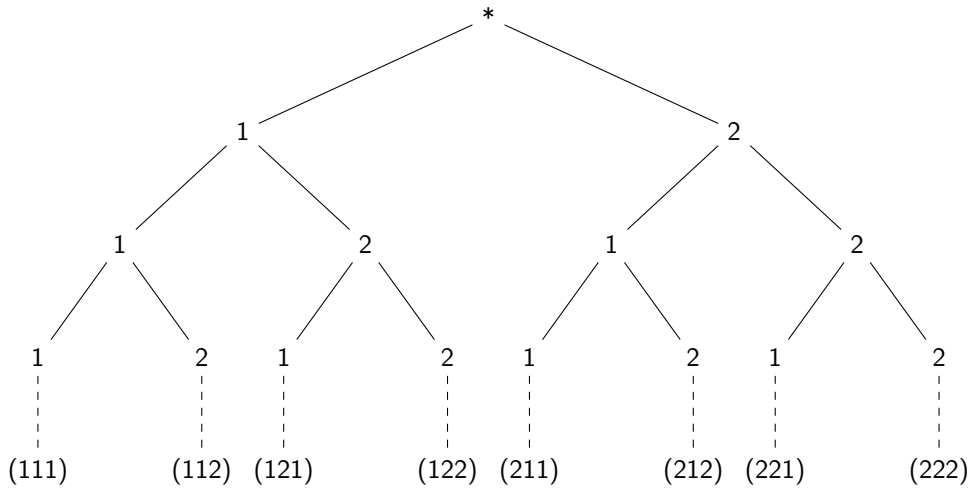
```
1  def print_arbre(liste,marge):
2      print (marge, liste[0])
3      if len(liste) == 1: return()
4      for i in range(1, len(liste)):
5          print_arbre(liste[i], marge + marge1)
6
7  arbre = ['A', ['B'] , ['C',  ['F'] ,  ['G']] , ['D'] , ['E', ['H']]]
8  marge1="   "
9  print_arbre(arbre, marge1)
```

```
    A
       B
       C
          F
          G
       D
       E
          H
```

■ *Exercice 2 :*

On considère un ensemble de n variables entières qui peuvent prendre chacune les valeurs de 1 à k. Il s'agit d'énumérer les k^n uplets représentant toutes les configurations possibles des valeurs de ces variables.

Pour cela, on construira l'arbre suivant (ici $n = 3$ et $k = 2$) :



puis on réalisera un affichage convenable de cet arbre.

Solution :

```python
n = 2 ; k = 3
arbre = [[i + 1] for i in range(n)]
for j in range(1, k):
    arbre = [[i + 1] + arbre for i in range(n)]
arbre=['*'] + arbre
print(arbre) # pour vérifier

liste = []
# variable globale qui contient le résultat à afficher et qui est gérée comme une pile
# chaque fois que l'on passe sur un noeud on empile la valeur de ce noeud
# chaque fois que l'on remonte dans l'arbre ou que l'on passe à un frère, on dépile
def parcours_arbre(arbre):
    liste.append(arbre[0]) # on empile la valeur de la racine
    if len(arbre) == 1: # on est arrivé à une feuille
        print(liste[1:]) # on n'affiche pas le '*' de la racine
    else:
        for i in range(1, len(arbre)) :
            parcours_arbre(arbre[i])
    liste.pop() #  on dépile

parcours_arbre(arbre)
```

```
['*', [1, [1, [1], [2]], [2, [1], [2]]], [2, [1, [1], [2]], [2, [1], [2]]]]
[1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[1, 2, 2]
[2, 1, 1]
[2, 1, 2]
[2, 2, 1]
[2, 2, 2]
```

La solution précédente est en fait très mauvaise : en effet, elle oblige à construire l'arbre entier (qui comporte k^n noeuds), alors qu'il suffit d'afficher les combinaisons cherchées une par une, ce qui nécessite juste l'emploi d'une liste de longueur k. Nous verrons comment écrire un programme optimisé un peu plus loin.

■ *Exercice 3 : Algorithme de Huffman*

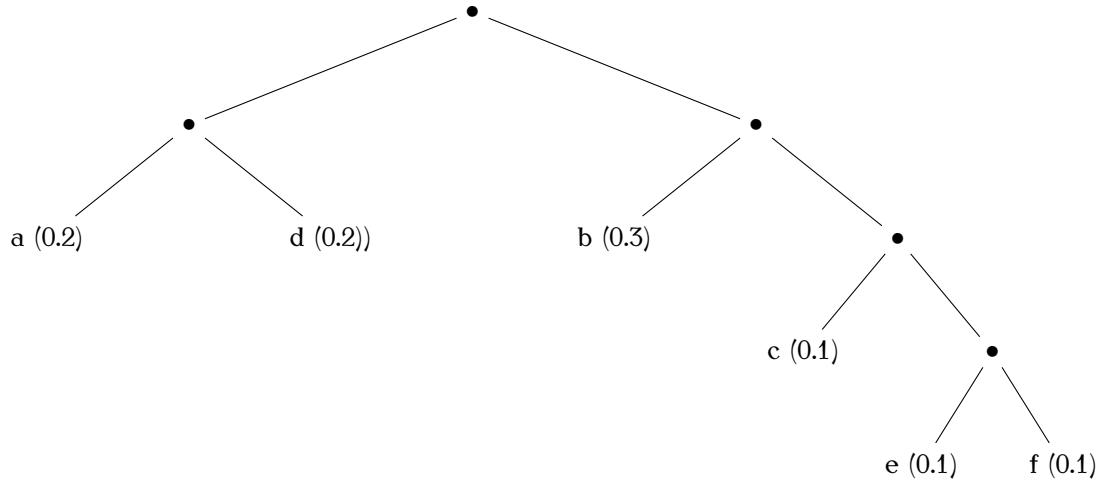
Soient n objets $x_1,\dots,x_n$ dont les probabilités d'apparition $p_1,\dots,p_n$ sont connues, avec $\sum_{i=1}^{n} p_i = 1$.

Il s'agit de construire un arbre binaire dont ces n objets sont les feuilles, de telle sorte que la distance totale des objets à la racine, pondérée par les probabilités d'apparition de chaque objet, soit minimale (si ℓ_i

est la distance de l'objet n° i à la racine, cette distance pondérée est égale à $\sum_{i=1}^{n} p_i \ell_i$).

Exemple :

Supposons que l'ont ait 6 objets a, b, c, d, e, f avec les probabilités d'apparition respectives $0.2, 0.3, 0.1, 0.2, 0.1, 0.1$. L'arbre construit pourra être :



La longueur de ce codage est alors égale à :

$$2\times 0.2 \ + \ 4\times 0.05 \ + \ 4\times 0.05 \ + \ 3\times 0.15 \ + \ 2\times 0.4 + 2\times 0.1 \ =$$

Algorithme :

Le principe de l'algorithme de Huffman est assez simple. Cet algorithme travaille sur un ensemble d'arbres (on appelle ce genre d'ensemble une *forêt* !). Chaque arbre, dont les feuilles sont des objets parmi $x_1, \ldots, x_n$, possède un poids : si cet arbre est réduit à une feuille, ce poids est la probabilité d'apparition de l'objet correspondant ; un arbre non réduit à une feuille a pour poids la somme des poids de ses deux fils.

- La forêt initiale est constituée des arbres-feuilles $[p_i, x_i]$, déjà triés par ordre décroissant ($p_i \geqslant p_{i+1}$).
- Tant que la forêt n'est pas réduite à un élément, on fait :
  - chercher les deux éléments a_1 et a_2 de plus faibles poids dans la forêt (ce sont donc forcément les deux derniers de la liste) ;
  - les retirer de la forêt et créer un nouvel arbre dont le poids sera la somme des poids de a_1 et a_2, et dont les fils gauche et droit seront a_1 et a_2 ; ce nouvel arbre sera inséré dans la forêt de façon à respecter l'ordre.
- à la fin, la forêt ne comporte plus qu'un arbre, qui est celui cherché.

Une solution pour implémenter cet algorithme est d'utiliser une liste triée par poids décroissants. Les éléments de cette liste seront des arbres ; chaque arbre sera représenté sous la forme d'une liste [*poids*, *fils gauche*, *fils droit*], où *fils gauche* et *fils droit* sont aussi des arbres. Dans le cas d'une feuille, *fils gauche* contiendra l'objet x_i et *fils droit* sera la liste vide [ ] (en effet, dans un arbre de Huffman, tous les noeuds sauf les feuilles ont nécessairement deux fils).

Programme :

```
from copy import *

def insere(liste,arbre):
    # insère un nouvel arbre dans la forêt liste déjà triée en ordre décroissant
    liste.append(arbre) # on l'insère à la fin
    x = arbre[0] # le poids
    i = len(liste) - 2
    while (i >=0 ) and (x > liste[i][0]):
# on remonte la liste en échangeant les éléments jusqu'à ce que
# le bon emplacement ait été trouvé
        liste[i+1] = liste[i]
        liste[i] = arbre
        i -= 1

```

```python
def construit(arbre1,arbre2):
    # construit un nouvel arbre binaire de fils arbre1 et arbre2
    nouv = [0,0,0]
    nouv[0] = arbre1[0] + arbre2[0] # somme des poids
    nouv[1] = arbre1
    nouv[2] = arbre2
    return(nouv)

def arbre_huffman(foret) :
    arbre = deepcopy(foret)
    while len(arbre) > 1:
        arbre1 = arbre.pop()
        arbre2 = arbre.pop()
        insere(arbre, construit(arbre1, arbre2))
    return(arbre[0])

liste_feuilles = [ [0.3, ['b'], [] ], [0.2, ['d'], [] ], [0.2, ['a'], [] ],\
         [0.1, ['c'], [] ] , [0.1, ['f'], [] ] , [0.1, ['e'], [] ]]

arbre = arbre_huffman(liste_feuilles)

print(arbre[0])
print(arbre[1])
print(arbre[2][0])
print(arbre[2][1])
print(arbre[2][2])
```

```
1.0
[0.4, [0.2, ['a'], []], [0.2, ['d'], []]]
0.6000000000000001
[0.3, ['b'], []]
[0.30000000000000004, [0.1, ['c'], []], [0.2, [0.1, ['e'], []], [0.1, ['f'], []]]]
```

Un joli affichage de l'arbre de Huffman obtenu serait à faire, mais ce n'est pas ce qui nous importe pour la suite !

■ *Exercice 4 : Codage de Huffman*

Le codage de Huffman est un algorithme de compression des données basé sur les fréquences d'apparition des caractères apparaissant dans le document initial. Il a été développé par un étudiant de la MIT, David A. Huffman, en 1952. Cette technique est largement utilisée car elle est très efficace et on observe selon le type de données des taux de compression allant de 20% à 90% mais plus généralement entre 30% et 60%.

Ce principe de compression est utilisé dans le codage d'image TIFF (Tagged Image Format File). La méthode JPEG (Join Photographic Experts Group) utilise un algorithme similaire.

Le principe consiste à attribuer des codes plus courts pour des valeurs fréquentes et des codes plus longs pour les valeurs moins fréquentes. Cela est plus efficace que la représentation de base qui consiste à utiliser une longueur fixe pour chaque symbole (exemple : un octet par caractère dans le code ASCII).

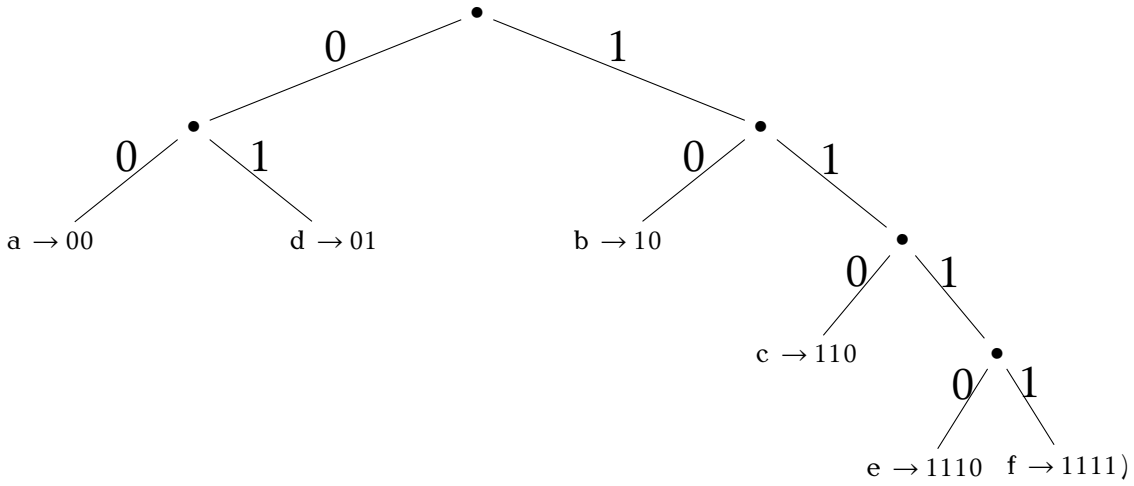
L'algorithme de Huffman utilise une table contenant les fréquences d'apparition de chaque caractère pour établir une manière optimale de les représenter par une chaîne binaire (cela reprend le principe du code Morse qui tend à minimiser le nombre de symboles à utiliser pour les lettres les plus fréquemment employées).

On peut décomposer la procédure en plusieurs parties :

- Tout d'abord, la création de la table de fréquence d'apparition des caractères dans le texte initial (dans la pratique, on utilise plutôt une table de fréquences prédéfinie, ce qui est plus rapide).
- Ensuite la création de l'arbre de Huffman, selon le principe vu auparavant, en utilisant la table précédente.
- Puis coder chaque symbole en représentation binaire, en affectant à chaque caractère un symbole d'autant plus court qu'il est proche de la racine dans l'arbre de Huffman. Pour cela, chaque branche de l'arbre sera codée par 0 (fils gauche d'un noeud) ou 1 (fils droit) ; les chemins depuis la racine jusqu'aux feuilles épèlent alors les codes binaires des caractères.
- Enfin, coder le texte initial en entier. Le fichier de sortie comprendra comme en-tête la liste des codes, afin que le destinataire puisse recomposer le message.

On peut montrer que le code obtenu à la 3ème étape est un *code préfixe*, c'est-à-dire qu'aucun code obtenu n'est préfixe d'un autre code. Cela permet d'assurer le décodage du document sans ambiguïté.

Ainsi, dans l'exemple précédent, on pourra étiqueter l'arbre de la façon suivante :



La procédure pour le codage des caractères consiste simplement à parcourir l'arbre en profondeur, en empilant l'étiquette 0 quand on parcourt la branche gauche, l'étiquette 1 pour la branche droite, et en affectant au caractère la pile obtenue lorsque l'on arrive à une feuille. Comme dans l'exercice 2 (`parcours_arbre`), on dépile chaque fois que l'on remonte dans l'arbre ou que l'on passe au frère.

```
from Arbre_Huffman  import *

liste_feuilles = [ [0.3, ['b'], [] ], [0.2, ['d'], [] ], [0.2, ['a'], [] ],\
         [0.1, ['c'], [] ] , [0.1, ['f'], [] ] , [0.1, ['e'], [] ]]

def codage_huffman(liste_feuilles):

    def codage(arbre):
        if arbre[2] != []: # si on n'est  pas sur une feuille
            temp.append("0")
            codage(arbre[1])
            temp.pop()
            temp.append("1")
            codage(arbre[2])
            temp.pop()
        else:
            liste_codes[arbre[1][0]] = "".join(temp)

    arbre = arbre_huffman(liste_feuilles)
    liste_codes = {} # codage des caractères dans un dictionnaire
    temp = []
    codage(arbre)
    return(liste_codes)

print(codage_huffman(liste_feuilles))
```

```
{'a': '00', 'c': '110', 'b': '10', 'e': '1110', 'd': '01', 'f': '1111'}
```

Il reste maintenant à écrire trois procédures : la première permet, à partir d'un texte ici représenté par une chaîne de caractères (ce sera un fichier d'entrée dans la pratique), de construire la table de fréquence d'apparition de chaque caractère puis d'en déduire la forêt `liste_feuilles` ; le second permettra ensuite l'encodage du texte sous forme d'une suite de 0 et de 1 ; enfin la tropisième permettra le décodage d'un texte formé de 0 et de 1 connaissant la forêt `liste_feuilles` (dans la pratique, cette variable forme le *header* du fichier compressé).

```
from Codage_Huffman import *

def liste_feuilles(texte):
```

```
    table = {}
    # on compte d'abord le nombre d'occurences de chaque caractère
    for caractere in texte:
        if caractere in table:
            table[caractere] += 1
        else:
            table[caractere] = 1
    # puis on remplace ce nombre par sa fréquence
    n = len(texte)
    for caractere in table:
        table[caractere] /= n
    # à partir du dictionnaire, création de la liste d'abord non triée
        liste = []
    for caractere in table:
        liste.append([table[caractere], [caractere], []])
    # on la trie selon l'ordre décroissant des fréquences
    liste.sort(key = lambda l: l[0], reverse = True)
    return(liste)

texte="abracadabra"

dict_codage = codage_huffman(liste_feuilles(texte))
print(dict_codage)

def encodage(texte,code):
    texte_binaire = ""
    for c in texte:
        texte_binaire += code[c]
    return(texte_binaire)

texte_binaire = encodage(texte,dict_codage)
print(texte_binaire)

def decodage(texte_binaire, code):
    code_inv = dict((code[b], b) for b in code)
    # construit le dictionnaire inversé
    texte = ""
    temp = ""
    for b in texte_binaire:
        temp += b
        if temp in code_inv:
            texte += code_inv[temp]
            temp = ""
    return(texte)

texte_decode = decodage(texte_binaire,dict_codage)
print(texte_decode)
```

```
{'a': '0', 'c': '1101', 'b': '10', 'd': '1100', 'r': '111'}
01011101101011000101110
abracadabra
```

Dans l'exemple (très simpliste !) ci-dessus, le texte initial comporte 11 caractères, donc occupe 11 octets (ou 22 selon le codage utilisé pour les chaînes de caractères) soit 88 (ou 176) bits en mémoire. La chaîne compressée, elle, n'en occupe que 23, soit un taux de compression de presque 74% (voire 87%) ! (dans la pratique il faut ajouter à la taille de la chaîne encodée celle du *header* ; c'est-à-dire du dictionnaire permettant le décodage).

III . Représentation des arbres binaires à l'aide de listes contigües

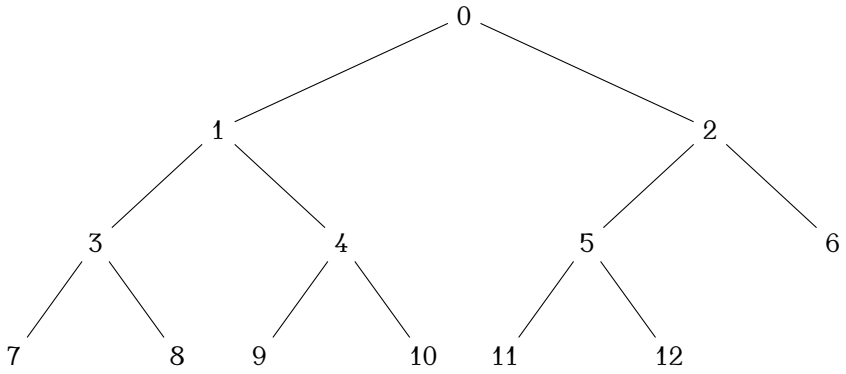
Les exemples précédents montrent que l'implémentation d'un arbre à l'aide de listes est relativement complexe ; de plus, elle ne permet pas de faire aisément certaines opérations courantes sur les arbres.

Il existe cependant certains cas où l'utilisation de listes pour représenter un arbre est commode : il s'agit des arbres binaires, et plus particulièrement des arbres binaires complets.

III.1 . Arbres binaires complets

Un arbre binaire de hauteur n est dit *complet* si tous les noeuds de chaque niveau sont présents, sauf peut-être dans le cas des feuilles. Dans ce cas, le dernier niveau sera rempli de gauche à droite (cette dernière condition n'est pas toujours utile).

Exemple d'un arbre binaire complet :



Un arbre binaire complet peut alors être représenté par une liste de longueur 2^n-1, de sorte que l'élément stocké à l'indice i ait ses deux fils situés aux indices $2i+1$ et $2i+2$. Les emplacements correspondant aux noeuds manquants peuvent être remplis par une valeur fictive (`None` par exemple).

Dans l'exemple ci-dessus, on a ainsi étiqueté chaque noeud par la position qu'il occupe dans le tas.

Exercice :

Écrire une fonction `afficher_arbre_binaire(liste)` qui affiche la liste sous une forme d'arbre plus facile à lire.

Solution :

```
from math import log

def affiche_arbre_binaire(liste):
      # nombre de niveaux de l'arbre
    hauteur = int(log(len(liste),2)) + 1

    for i in range(hauteur):
        ligne=""
        # ecartement entre les noeuds sur cette ligne
        ecart_ligne = (2**(hauteur-i+1) - 3)*" "
        # écart au début
        ecart_debut= (2**(hauteur-i)-2)*" "
        for j in range(2**i-1,min(2**(i+1)-1,len(liste))) :
            ligne += "{0:^3}".format(str(liste[j]))
            if j < min(2**(i+1)-1,len(liste)) - 1 :
                ligne += ecart_ligne
        print(ecart_debut+ligne)

liste = [1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9,1,2,3,4,5]
affiche_arbre_binaire(liste)
```

```
                                1
                    2                               3
            4               5               6               7
        8       9       1       2       3       4       5       6
    7   8   9   1   2   3   4   5
```

III.2 . Les tas

Un *tas* (*heap* en Anglais) est un arbre binaire complet qui vérifie en plus la propriété suivante : la clé de chaque noeud doit être supérieure ou égale (resp. inférieure ou égale) aux clés de chacun de ses fils.

Dans le premier cas ($\geqslant$), on parle de *tas-max* (ou *max-heap*) et dans le second cas ($\leqslant$) on parle de *tas-min* (ou *min-heap*).

Compte tenu de l'importance de cette structure de données, nous allons détailler les opérations que l'on peut réaliser sur les tas, en nous limitant au cas de tas-min. Bien sûr, ces opérations doivent conserver la structure d'ordre du tas.

■ **Ajout d'un élément :**

Le principe pour ajouter un élément x à un tas-min T est le suivant :

- Mettre x dans $T[n]$, où n est la première place disponible dans le tas.
- On compare alors cette valeur à celle contenue dans le noeud père (c'est-à-dire $T[n/2]$) ; si elle est supérieure, le processus s'arrête puisque l'on a encore un tas. Sinon, on échange les éléments $T[n]$ et $T[n/2]$.
- On remonte alors au père, et on réitère l'opération précédente jusqu'à ce que le tas soit convenablement ordonné.

En fait, dans le programme qui suit, on n'a même pas échangé les éléments deux par deux : on s'est contenté de descendre l'élément père s'il n'était pas bien plaçé, puis on remonte dans l'arbre jusqu'à trouver le bon emplacement pour y mettre x.

Si le tas a n éléments, le nombre d'opérations n'excédera pas la hauteur de l'arbre correspondant. Or la hauteur d'un arbre binaire complet de n noeuds est $E(\log_2(n))+1$. Donc l'algorithme d'insertion d'un élément dans un tas est de complexité $O(\log(n))$.

Programme :

```
def insere_tas(tas,x):
    tas.append(x)
    i = len(tas) - 1
    pere = (i-1)//2
    # on remonte dans l'arbre jusqu'à trouver la bonne place
    while pere >=0 and tas[pere] >= x :
        tas[i] = tas[pere]
        i = pere
        pere = (i-1)//2
    tas[i] = x
```

```
from Affiche_Arbre_Binaire import *

from Insere_Tas import *

tas = [5,9,11,14,18,19,21,30,17,25]
affiche_arbre_binaire(tas)
print('-'*60)

insere_tas(tas,7)
affiche_arbre_binaire(tas)
print('-'*60)

insere_tas(tas,4)
affiche_arbre_binaire(tas)
print('-'*60)

insere_tas(tas,35)
affiche_arbre_binaire(tas)
```

```
                5
        9               11
   14       18      19       21
30   17   25
------------------------------------------------------------
                5
        7               11
   14       9       19       21
30   17   25   18
------------------------------------------------------------
                4
        7                5
   14       9       11       21
30   17   25   18   19
------------------------------------------------------------
                4
        7                5
   14       9       11       21
30   17   25   18   19   35
```

■ **Suppression de l'élément racine :**

Pour supprimer le premier élément du tas (donc le minimum dans le cas d'un tas-min), il faut alors retirer la racine de l'arbre représentant la file, ce qui donne deux arbres ! Le plus simple pour reformer un seul arbre est d'appliquer l'algorithme suivant :

- on met l'élément le plus à droite de la dernière ligne (donc le dernier élément de la liste) à la place de la racine.
- on compare alors sa valeur avec celle de ses fils et on échange cette valeur avec la plus petite de ces deux valeurs (bien sûr, il faut faire attention, quand un noeud n'a qu'un fils).
- on réitère cette opération en descendant dans l'arbre jusqu'à ce que le tas soit convenablement ordonné.

A nouveau, la suppression du premier élément du tas ne prend pas un temps supérieur à la hauteur de l'arbre représentant la file. Donc, pour un tas de n éléments, l'algorithme de suppression de la racine est de complexité $O(\log(n))$.

Programme :

```python
from Affiche_Arbre_Binaire import *

def supprime_racine_tas(tas):
    x = tas.pop(-1)
    tas[0] = x
    n = len(tas) - 1
    i = 0
    while (2*i + 1) <= n :
        j = 2*i + 1
        if j < n :
            if tas[j + 1] < tas[j] :
                j += 1
        if x <= tas[j] :
            tas[i] = x
            return()
        tas[i] = tas[j]
        i = j
    tas[i] = x

tas = [5,9,11,14,23,19,21,30,17,25,31,22]
affiche_arbre_binaire(tas)
print('-'*60)
supprime_racine_tas(tas)
affiche_arbre_binaire(tas)
print('-'*60)
supprime_racine_tas(tas)
```

```
affiche_arbre_binaire(tas)
```

```
                5
        9               11
    14      23      19      21
30  17  25  31  22
------------------------------------------------------------
                9
        14              11
    17      23      19      21
30  22  25  31
------------------------------------------------------------
               11
        14              19
    17      23      31      21
30  22  25
```

■ **Construction d'un tas :**

Il s'agit maintenant de fabriquer un tas-min à partir d'une liste d'éléments quelconque.

Une solution élémentaire consiste à partir d'un tas formé du premier élément de la liste donnée, et de lui ajouter les éléments suivants un par un à l'aide de la procédure `insere_tas`. Puisque le nombre d'opérations pour insérer un élément dans un tas de taille k est inférieur à $C\log(k)$ où C est une constante, le nombre d'opérations pour construire un tas de taille n sera inférieur à $C\sum_{k=2}^{n}\log(k)$, et puisque par comparaison série-intégrale on a

$$\sum_{k=2}^{n}\log(k) \underset{n\to+\infty}{\sim} \int_2^n \log(t)\,\mathrm{d}t \underset{n\to+\infty}{\sim} n\log(n)\,,$$

cet algorithme est de complexité $O(n\log(n))$.

Programme :

```
from Affiche_Arbre_Binaire import *

from Insere_Tas import *

liste = [5,1,90,34,2,66,3,18,12,4,34,25,7]

tas=[liste[0]]
for i in range(1,len(liste)):
    insere_tas(tas,liste[i])

affiche_arbre_binaire(tas)
```

```
                1
        2               3
    12      4       7       66
34  18  5   34  90  25
```

■ **Le tri par tas (*heapsort*) :**

Pour trier une liste d'éléments par ordre croissant, il ne reste plus qu'à construire un tas à partir de cette liste, puis d'en extraire les éléments triés un par un à l'aide de la fonction `supprime_racine_tas`. Puisque cette fonction est de complexité $O(\log(n))$ et qu'elle est exécutée n fois, cet affichage sera de complexité $O(n\log(n))$ et puisque la construction du tas a une complexité similaire, l'algorithme *heapsort* sera encore de complexité $O(\log(n))$.

Programme :

```
from Insere_Tas import *

from Supprime_Racine_Tas import *

from random import random
```

```
6
7  n=20
8  liste = [0]*n
9  for i in range(n):
10     liste[i] = int(100*random())
11 print(liste)
12
13 #construction du tas
14 tas=[liste[0]]
15 for i in range(1,len(liste)):
16     insere_tas(tas,liste[i])
17
18 #construction de la liste triée
19 liste_tri = [0]*n
20 for i in range(n) :
21     liste_tri[i] = tas[0]
22     if len(tas) >1 : supprime_racine_tas(tas)
23     # la procédure supprime_racine_tas ne fonctionne pas si le tas a un seul élément
24
25 print(liste_tri)
```

```
[52, 37, 49, 82, 32, 13, 28, 64, 36, 80, 67, 89, 93, 41, 56, 71, 90, 23, 23, 13]
[13, 13, 23, 23, 28, 32, 36, 37, 41, 49, 52, 56, 64, 67, 71, 80, 82, 89, 90, 93]
```