Question 1

Considérons un arbre de hauteur h comportant n nœuds.
Par définition, le nombre de ses niveaux est $h+1$; or il y a au moins un nœud par niveau, d'où $n \geq h+1$.
Par ailleurs, il y a au plus 2^k nœuds au niveau k (par récurrence immédiate, puisque il y a un unique nœud au niveau 0 et chaque nœud du niveau k a au plus 2 fils au niveau $k+1$). Par conséquent

$$\sum_{k=0}^{h} 2^k \geq n \quad \text{c'est-à-dire} \quad n \leq 2^{h+1}-1 \quad \text{soit} \quad n < 2^{h+1} \quad \text{d'où} \quad \log_2 n < h+1.$$

Finalement

$$\boxed{\lfloor \log_2 n \rfloor \leq h \leq n-1.}$$

Notons que ces bornes sont optimales : on a $h = n-1$ pour un arbre *filiforme* (où chaque nœud interne a un seul fils non vide) et $h = \lfloor \log_2 n \rfloor$ pour un arbre parfait, puisque pour un tel arbre, d'après le calcul précédent :

$$2^h - 1 < n \leq 2^{h+1} - 1 \quad \text{soit} \quad 2^h \leq n < 2^{h+1}.$$

Question 2

Il vaut mieux saisir les crochets 2 par 2… La liste associée à l'exemple 1 est

$$\boxed{[1,[2,[\,],[\,]],[3,[4,[5,[\,],[\,]],[6,[\,],[\,]]],[\,]]]}$$

Question 3

Pas de difficulté avec les listes Python !

```
def est_vide(a):
    return a==[]

def arbre_vide():
    return []

def cons(e,g,d):
    return [e,g,d]
```

```
def contenu(a):
    return a[0]

def f_g(a):
    return a[1]

def f_d(a):
    return a[2]
```

Question 4

Comme il est plus facile de tester si un arbre est vide que de tester s'il contient un unique nœud, on convient que la hauteur de l'arbre vide est -1, ce qui permet d'étendre au cas de l'arbre vide la relation

$$h(a) = 1 + \max\left(h(fils\_g(a)), h(fils\_d(a))\right).$$

Cette relation se prête évidemment à une programmation récursive :

```
def hauteur(a):
    if est_vide(a):
        return -1
    else:
        return 1+max(hauteur(f_d(a)),hauteur(f_g(a)))
```

J'utilise la fonction `max` de Python. Terminaison justifiée par la décroissance stricte de la hauteur de l'arbre en paramètre à chaque appel récursif et preuve par récurrence sur ladite hauteur !

Question 5

Je construis récursivement la chaîne de caractères souhaitée :

- si l'arbre est vide, je renvoie la chaîne vide
- si l'arbre est une feuille (deux fils vides), je renvoie la chaîne représentant le nombre en étiquette (conversion avec `str`)
- sinon, l'étiquette est un opérateur, je renvoie donc une parenthèse, suivie de la chaîne associée au sous-arbre gauche (premier opérande), de l'étiquette puis de la chaîne associée au sous-arbre droit (second opérande).

D'où le programme :

```
def parcours_infixe(a):
    if est_vide(a):
        return ''
    elif est_vide(f_g(a)) and est_vide(f_d(a)):
        return str(contenu(a))
    else:
        return '('+parcours_infixe(f_g(a))+contenu(a)+parcours_infixe(f_d(a))+')'
```

N.B. : la dénomination "*parcours infixe*" vient du fait que l'étiquette est écrite **entre** les chaînes correspondant aux sous-arbres. Le résultat pour l'arbre de l'énoncé est

```
((3+(5x(7+2)))-(4+(8x9)))
```

On peut de même programmer récursivement un parcours *préfixe* (avec l'opérateur avant les opérandes, comme dans le langage Lisp, ou comme $f(x, y)$ en maths !) ou un parcours *postfixe* (avec l'opérateur après les opérandes, comme dans la *notation polonaise inverse*, cf. le chapitre 3). Ainsi, le programme suivant :

```
def parcours_postfixe(a):
    if est_vide(a):
        return ''
    elif est_vide(f_g(a)) and est_vide(f_d(a)):
        return str(contenu(a))
    else:
        return parcours_postfixe(f_g(a))+' '+parcours_postfixe(f_d(a))+' '+contenu(a)
```

donne pour l'arbre de l'énoncé la chaîne suivante : `3 5 7 2 + x + 4 8 9 x + -`

Avec cette notation les parenthèses ne sont pas nécessaires, un espace suffit pour séparer deux symboles.

Question 6

J'applique l'indication de l'énoncé :

- j'ajoute l'élément x à la fin de t (méthode `append`) ; x se trouve alors dans $t[i]$ où i vaut $len(t) - 1$ (rappelons que $t[0]$ n'est pas utilisé)
- tant que $t[i]$ a un père (condition $i > 1$) et est plus petit que son père $t[p]$ (où $p = i//2$), j'échange $t[i]$ et $t[p]$ et j'actualise les valeurs de i et de p

D'où le programme :

```
def entasser(x,t):
    t.append(x)
    i=len(t)-1 #l'indice de cette dernière valeur
    p=i//2 #le père
    while i>1 and t[i]<t[p]:
        t[i],t[p]=t[p],t[i]
        i,p=p,p//2
```

Question 7

Selon l'indication de l'énoncé, je commence par remplacer la racine $t[1]$ par la dernière valeur $t[-1]$, puis je supprime cette dernière valeur (méthode `pop`). Ainsi la valeur minimum (racine initiale) a disparu, mais t n'est *a priori* plus un tas, il faut faire "redescendre" la nouvelle racine en préservant la structure de tas. Pour cela, il est clair qu'il suffit d'échanger cette valeur $t[i]$ avec le plus petit de ses deux fils (ou avec son fils si elle n'en a qu'un, cas $2i = n$), noté $t[f]$, cela tant que $t[i]$ a au moins un fils (condition $i \leq n//2$) et que $t[i] > t[f]$.

Si $t[i] \leq t[f]$ on peut sortir, d'où le `return None`, puisque nous n'avons rien à renvoyer, le résultat de l'exécution de cette *procédure* est la modification de `t`.

D'où le programme :

```
def suppr_min(t):
    t[1]=t[-1]
    t.pop()
    n=len(t)-1
    i=1
    while i<=n//2:
        if 2*i==n or t[2*i]<t[2*i+1]:
            f=2*i
        else:
            f=2*i+1
        if t[i]>t[f]:
            t[i],t[f]=t[f],t[i]
            i=f
        else:
            return None
```

Noter que les deux premières lignes peuvent être remplacées par `t[1]=t.pop()`, **sauf** pour un tableau d'une seule valeur, qui doit être vidé par notre fonction.

Question 8

Selon le préambule, je remplis un tableau auxiliaire *tas* (initialisé avec un 0 en première place, qui restera inutilisé...) où j'entasse les valeurs du tableau initial *ti*, puis je reremplis *ti* avec les valeurs (dans l'ordre croissant !) extraites de *tas* l'une après l'autre (l'utilisation de *suppr_min* assure que la racine de *tas* est la plus petite de ses valeurs à chaque étape !).

N.B. Les n valeurs de *ti* sont bien stockées dans $tas[1], \ldots, tas[n]$ pour bénéficier des liens père-fils de l'énoncé...

La complexité en $O(n \log_2 n)$ est justifiée dans l'énoncé (au **II-4**). On remarquera que cet algorithme (diabolique) donne un programme de tri efficace et **itératif** !
Voici le programme :

```
def tri_tas1(ti):
    tas=[0]
    for x in ti:
        entasser(x,tas)
    ti=[]
    while len(tas)>1:
        ti.append(tas[1])
        suppr_min(tas)
    return ti
```

Question 9

Pour effectuer les opérations précédentes sans tableau auxiliaire, je donne comme paramètre à la fonction *entasser* l'**indice** i au lieu de la **valeur** x (ajoutée à la fin du tas en construction, à la question **6**).

Pour l'extraction des valeurs dans l'ordre, au lieu de supprimer la valeur à la racine comme à la question **7**, je l'échange avec $t[n]$ (n étant fourni en paramètre) et je recrée un tas avec les valeurs $t[1], \ldots, t[n-1]$.

Bien entendu, la méthode ci-dessus va rendre les valeurs initiales en ordre décroissant si je crée des tas avec la plus petite valeur à la racine... Il suffit d'inverser le sens des inégalités dans les tests pour retrouver l'ordre croissant !

Ci-dessous les programmes (à peine) modifiés :

```
def entasser(i,t): #entasse t[i] parmi t[1],...,t[i-1]
    p=i//2 #le père
    while i>1 and t[i]>t[p]:
        t[i],t[p]=t[p],t[i]
        i,p=p,p//2

def suppr_min(t,n): #échange t[1] et t[n] et
                    #refait un tas avec les n-1 premières valeurs
    t[1],t[n]=t[n],t[1]
    n-=1
    i=1
    while i<=n//2:
        if 2*i==n or t[2*i]>t[2*i+1]:
            f=2*i
        else:
            f=2*i+1
        if t[i]<t[f]:
            t[i],t[f]=t[f],t[i]
            i=f
        else:
            return None

def tri_tas2(ti):
    n=len(ti)-1
    for i in range(2,n+1):
        entasser(i,ti)
    for i in range(n,1,-1):
        suppr_min(ti,i)
    return ti
```

La complexité en temps est similaire, la complexité en mémoire est améliorée !

Question 10

L'algorithme est décrit dans l'énoncé... Il faut se rappeler que le père du nœud d'indice n a pour indice $r = n//2$ et que deux cas se présentent :

- soit $n = 2r$, auquel cas le nœud d'indice r a un seul fils
- soit $n = 2r + 1$, auquel cas le nœud d'indice r a deux fils, d'indices $2r$ et $2r + 1$

Dans tous les cas, le nœud d'indice $n//2$ est le dernier à avoir au moins un fils, d'où le test conditionnant l'appel récursif.

```
def ordonner(t,r):
    n=len(t)-1
    if 2*r==n or t[2*r]>t[2*r+1]:
        f=2*r
    else:
        f=2*r+1
    if t[r]<t[f]:
        t[r],t[f]=t[f],t[r]
        if f<=n//2:
            ordonner(t,f)
```

La terminaison du programme est justifiée par le fait que le second paramètre est au moins doublé à chaque appel récursif, tandis que l'appel récursif $ordonner(t, f)$ est soumis à la condition $f <= n//2$ (avec n qui reste inchangé, le tableau est réorganisé mais sa taille est constante !).

La correction se prouve par récurrence sur la hauteur du sous-arbre de racine $t\,[r]$.

Noter que, comme au **9**, je construis le tas avec le maximum à la racine (je choisis ci-dessus le plus **grand** des deux fils), pour obtenir un tri croissant après la seconde phase, identique à celle du **9**.

Voici le programme final, qui opère aussi en place :

```
def tri_tas3(ti):
    n=len(ti)-1
    d=n//2
    for r in range(d,0,-1):
        ordonner(ti,r)
    for i in range(n,1,-1):
        suppr_min(ti,i)
    return ti
```

Question 11

Du fait de la boucle `for r...` ci-dessus, la complexité de la création du tas est la somme des $C(r)$ pour $r = 1, \dots, d$, où $C(r)$ est le nombre de comparaisons (entre éléments du tableau comme d'habitude) effectuée lors de l'appel de *ordonner* (t, r).

Je fixe h entier tel que $2^{h-1} \leq d < 2^h$; h est le niveau des derniers nœuds qui ne sont pas des feuilles et j'ai, à la lecture du programme *ordonner*

$$\forall r \in \left[\!\left[2^{h-1}, d\right]\!\right] \quad C(r) = 2$$

et

$$\forall r < 2^{h-1} \quad C(r) \leq 2 + \max\left(C(2r), C(2r+1)\right) \tag{1}$$

J'en déduis par récurrence sur $j \in [\![0, h]\!]$

$$\mathcal{P}_j : \forall r \in [\![1, d]\!] \quad r \in \left[\!\left[2^{h-j-1}, 2^{h-j}\right[\!\right[\Rightarrow C(r) \leq 2(1+j).$$

En effet, j'ai $\mathcal{P}_0$ d'après ce qui précède et si je suppose $\mathcal{P}_j$ et que je considère r entier de $\left[\!\left[2^{h-j-2}, 2^{h-j-1}\right[\!\right[$ j'ai

$2^{h-j-2} \leq r \leq 2^{h-j-1} - 1$ d'où $2^{h-j-1} \leq 2r \leq 2^{h-j-1} - 2$ et $2^{h-j-1} + 1 \leq 2r + 1 \leq 2^{h-j-1} - 1$

par conséquent $2r$ et $2r+1$ sont dans $\left[\!\left[2^{h-j-1}, 2^{h-j}\right[\!\right[$, d'où grâce à (1) et à $\mathcal{P}_j$

$$C(r) \leq 2 + 2(1+j) = 2(1+j+1)$$

et donc $\mathcal{P}_{j+1}$ est vérifiée !

Or l'intervalle $\left[\!\left[2^{h-j-1}, 2^{h-j}\right[\!\right[$ contient exactement 2^{h-j-1} entiers, d'où

$$\sum_{r \in [2^{h-j-1}, 2^{h-j}[} C(r) \leq 2^{h-j-1} \times 2(1+j+1) = 2^h \times \frac{j+1}{2^j} \tag{2}$$

(pour $j = 0$, remplacer $\left[\!\left[2^{h-1}, 2^h\right[\!\right[$ par $\left[\!\left[2^{h-1}, d\right]\!\right]$, la majoration est "encore plus vraie" !).

Or on connaît pour tout z complexe tel que $|z| < 1$ la relation (produit de Cauchy de la série géométrique de raison z par elle-même !)

$$\sum_{j=0}^{\infty} (j+1) z^j = \frac{1}{(1-z)^2} \quad \text{d'où} \quad \sum_{j=0}^{\infty} \frac{j+1}{2^j} = 4.$$

Il en résulte en sommant les inégalités (2) pour $j = 0, \dots, h$ et en majorant généreusement

$$\sum_{r=1}^{d} C(r) \leq 2^h \times 4 \quad \text{or} \quad 2^h \leq n$$

d'où finalement :

$$\boxed{\text{La complexité de la création du tas est ici un } O(n).}$$

On notera que — pour la complexité globale du tri par tas — le bénéfice reste modeste (quelques pour cent) et diminue lorsque n augmente, puisque c'est la phase d'extraction, toujours en $n \log_2 n$, qui est prépondérante... Mais la structure de tas peut servir à d'autres choses qu'au tri par tas !

Question 12 (un peu technique mais pratique...)

Je choisis par exemple de réserver 3 caractères pour représenter un nombre (la sortie sera correcte pour les arbres parfaits contenant des entiers de 0 à 999). Je calcule alors pour chaque niveau le nombre d'espaces à afficher en début de ligne et entre deux champs de 3 caractères (faire un dessin !). La commande `format` permet de centrer les nombres à un chiffre, ceux à deux chiffres sont calés à gauche... Le programme suivant donne bien la sortie de l'énoncé.

```
def afficher(t):
    n=len(t)-1 #tas indexé de 1 à n
    h=int(log2(n)) #hauteur de l'arbre
    nbsc=[1] #nb d'espaces sur les côtés
    for i in range(h):
        nbsc=[2*nbsc[0]+1]+nbsc
    nbsi=[1] #nb d'espaces intermédiaires
    for i in range(h):
        nbsi=[2*nbsi[0]+3]+nbsi
    for i in range(h+1):
        ligne=' '*nbsc[i]
        for j in range(2**i,min(2**(i+1),n+1)):
            ligne=ligne+'{:^3}'.format(t[j])+' '*nbsi[i]
        print(ligne)
```