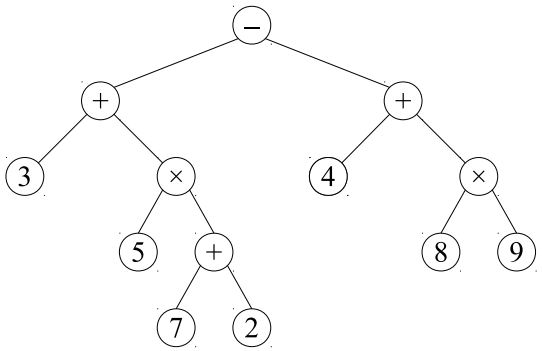
1 Structure d'arbre binaire - représentations par une liste

Question 1 : écrire en Python les primitives ci-dessous dans cette représentation.

- `est_vide(a)` : renvoie `Vrai` si l'arbre `a` est vide, `Faux` sinon ;
- `arbre_vide()` : renvoie l'arbre vide ;
- `cons(e,g,d)` : renvoie l'arbre d'étiquette `e`, de fils gauche (*resp.* droit) `g` (*resp.* `d`) ;
- `contenu(a)` : renvoie l'étiquette de l'arbre **non vide** `a` ;
- `f_g(a)` (*resp.* `f_d(a)`) : renvoie le fils gauche (*resp.* droit) de l'arbre **non vide** `a`.

Question 2 : programmer (récursivement !) dans ce contexte le calcul de la hauteur d'un arbre.

Question 3 : l'arbre suivant représente graphiquement une expression algébrique.



Programmer le *parcours* (récursif !) d'un tel arbre pour écrire dans une chaîne de caractères l'expression algébrique associée. Il sera sage d'ajouter des parenthèses. Tester le programme avec l'exemple ci-dessus.
Remarque

- l'opération inverse consistant à reconstruire l'arbre à partir de la chaîne de caractères est plus délicate.

2 Tri par tas (*heapsort* en anglais)

Nous avons vu en cours la notion de tas. Nous allons voir que l'implémentation se fait très bien avec les tableaux (et de façon **itérative**...), mais la vision arborescente est fondamentale pour imaginer et visualiser les algorithmes. Penser donc à utiliser la fonction `afficher` pour tester les fonctions.
Pour la suite, on suppose donné un tableau initial *ti* contenant les n valeurs à trier. Pour stocker n valeurs dans un tas, on utilisera un tableau de taille $n+1$, les cases "utiles" étant indexées de 1 à n comme expliqué en cours.

2.1 Tri par tas, première version

La version la plus naturelle consiste à utiliser un tableau auxilaire *tas* que l'on remplira avec les données initiales, avant de le vider.

Question 4 : écrire une procédure *entasser*, recevant pour paramètres un tas t et une valeur x et ajoutant x dans t en préservant la structure de tas. On pourra ajouter x à la fin du tas et le faire "remonter vers la racine" par échanges successifs $fils \leftrightarrow père$ autant que nécessaire (cf. l'idée du tri par insertion...).

Question 5 : écrire une procédure *suppr_min*, recevant pour paramètres un tas t et supprimant de t sa racine en préservant la structure de tas. On pourra remplacer la racine par la dernière valeur de t,

supprimer cette dernière valeur et faire “redescendre” cette nouvelle racine dans le tas à l'aide d'échanges $père \leftrightarrow fils$ (attention à choisir le bon fils lorsqu'il y en a deux...).

Question 6 : déduire des deux questions précédentes une fonction *tri_tas*1, recevant pour paramètre le tableau initial *ti* et le renvoyant trié. Seules les valeurs indexées à partir de 1 seront triées.

2.2 Tri par tas en place

Question 7 : modifier les programmes des questions **4**, **5**, **6** afin d'obtenir une fonction *tri_tas*2 triant “en place” le tableau reçu en paramètre (*i.e.* sans utiliser de tableau auxiliaire). Là encore, seules les valeurs indexées à partir de 1 seront triées, mais on pourra utiliser `t[0]` pour stocker le nombre d'éléments considérés dans le tas (différent de la taille du tableau qui sera constante cette fois).

2.3 Optimisation de la création du tas

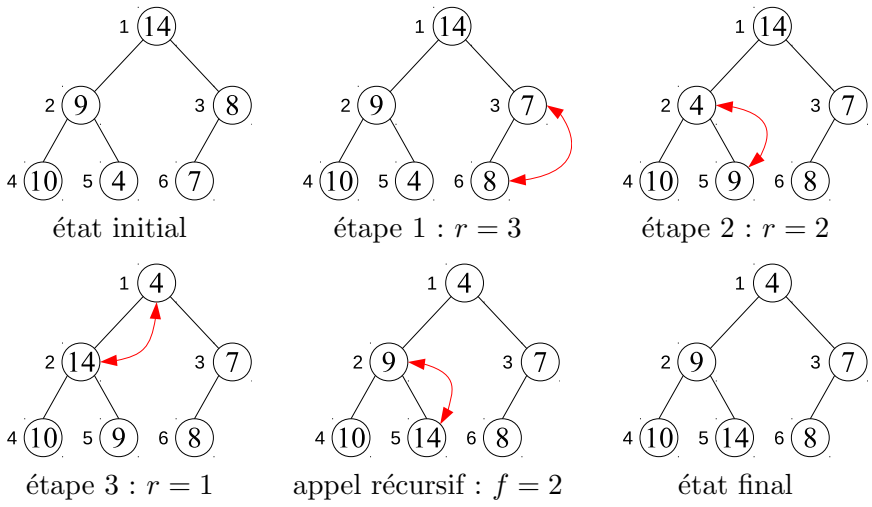
Si l'on s'y prend bien, la complexité de la fonction *entasser* de la question **4** est majorée par $\log_2 i$, où i est le nombre de valeurs dans le tas initial. Il en résulte une création progressive du tas contenant les n valeurs avec une complexité majorée par $\log_2(n!)$, soit un $O(n \ln n)$ grâce à la formule de Stirling. En fait cette création du tas peut se faire avec une complexité linéaire.

Pour cela, on part du tableau t contenant les valeurs à trier (initialement dans un ordre quelconque) dans les cases indexées de 1 à n, on pose $d = \lfloor n/2 \rfloor$, indice du nœud le plus à droite de l'avant-dernier niveau du tas. L'idée est d'ordonner successivement les sous-arbres dont les racines sont les nœuds d'indices $d, d-1, \ldots, 1$.

Pour ce faire on écrit une fonction $ordonner(t, r)$ qui traite le sous-arbre dont la racine a pour indice r, en supposant que les sous-arbres de ce dernier ont déjà été traités (ce qui justifiera une preuve par récurrence descendante !). Le traitement se déroule ainsi :

- on recherche l'indice f du plus petit fils de $t[r]$
- si besoin on échange $t[r]$ et $t[f]$ et dans ce cas, si $t[f]$ n'est pas une feuille, on appelle (récursivement) $ordonner(t, f)$ puisque le sous-arbre de racine $t[f]$ n'est peut-être plus un tas...

Voici un exemple à partir du tableau $t = [0, 14, 9, 8, 4, 10, 7]$ où l'on appelle successivement *ordonner* avec $r = 3, 2, 1$ (on a fait figurer son numéro à gauche de chaque nœud) :



Question 8 : programmer la fonction *ordonner* et l'utiliser pour écrire une fonction *tri_tas*3.

Question 9 (facultative) : montrer que la complexité de la création du tas à l'aide de la fonction *ordonner* est un $O(n)$.

3 Tri par arbres binaires de recherche - programmation objet

Un *arbre binaire de recherche*, est un arbre binaire tel que l'étiquette de tout nœud est :
- supérieure à toutes les étiquettes du sous-arbre gauche ;
- inférieure à toutes les étiquettes du sous-arbre droit.

Dans cette partie, on propose d'implémenter une classe `Abr`.
Les attributs de cette classe sont :
- `vide` : un booléen indiquant si l'arbre est vide ;
- `gauche` : `None` ou un `Abr` qui code le sous-arbre gauche ;
- `droite` : `None` ou un `Abr` qui code le sous-arbre droite ;
- `e` : un `int` qui code l'étiquette de la racine de l'arbre.

À sa création, un `Abr` est vide, ses sous-arbres ne sont pas créés et l'étiquette de la racine (inexistante) a une valeur nulle par défaut.

```
class Abr:
    def __init__(self):
        self.vide = True
        self.gauche = None
        self.droite = None
        self.e = 0
```

La commande `a = Abr()` crée un arbre vide. Pour accéder aux attributs d'un arbre `a` donné, on écrit `a.nom_attribut`. Pour définir les *méthodes* de la classe `Abr`, on définit des fonctions comme d'habitude, indentées d'un cran pour signifier l'appartenance à la classe. Le premier paramètre dans la définition est toujours `self`, mais il n'est pas utilisé dans les appels dont la syntaxe est `a.methode(arg2,arg3,...)`.

Question 10 : écrire une méthode dont l'entête est `def ajoute(self,x):` qui ajoute l'élément `x` dans l'arbre (s'il n'y est pas déjà présent) en maintenant la structure d'arbre binaire de recherche.

Question 11 : écrire une méthode dont l'entête est `def recherche(self,x):` qui renvoie un booléen indiquant si `x` est une étiquette de l'arbre.

Question 12 : écrire une méthode dont l'en-tête est `def hauteur(self):` qui renvoie la hauteur de l'arbre (un arbre vide étant de hauteur.

Question 13 : écrire une méthode dont l'entête est `def affiche(self):` qui va construire un tableau représentant l'arbre puis utiliser la fonction d'affichage définie en cours pour afficher l'arbre. (Ne pas hésiter à définir des méthodes auxiliaires)

Question 14 : écrire une méthode dont l'en-tête est `def taille(self):` qui renvoie le nombre de nœuds dans l'arbre.

Question 15 : écrire une méthode dont l'en-tête est `def tri(self):` qui renvoie un tableau contenant l'ensemble des étiquettes de l'arbre dans l'ordre croissant (il suffit de faire un parcours infixe).

Question 16 : Estimer la complexité du tri par arbre binaire de recherche en fonction de la hauteur de l'arbre. Que peut-on en conclure ?

Question 17 (subsidiaire) : écrire une méthode dont l'en-tête est `def supprime(self,x):` qui supprime l'élément `x` dans l'arbre (s'il y est présent) en maintenant la structure d'arbre binaire de recherche. Cette question est difficile et nécessite la définition de méthodes auxiliaires.