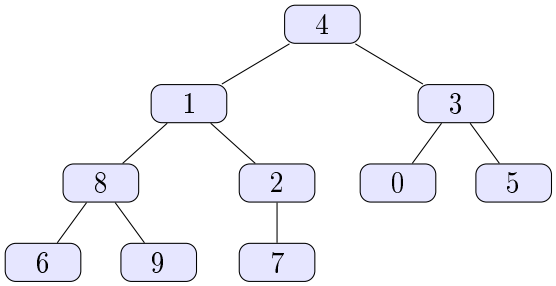
Travaux Pratiques 6

Rappels de cours *On considère un tableau numérique A où les indices sont numérotés de 0 à taille$(A)-1$, comme en Python. Le tableau A définit une structure d'arbre de racine $A[0]$ par la fonction* parent $p(i) = \lfloor (i-1)/2 \rfloor$. *Cet arbre est* binaire *car chaque élément a au plus un enfant* gauche $g(i) = 2i+1$ *et un enfant* droit $d(i) = 2i+2$.

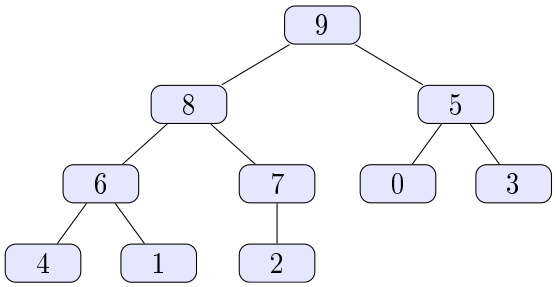
Voici un exemple de tableau et l'arbre binaire correspondant. Le mécanisme décrit revient à inscrire successivement les éléments du tableau dans la racine, puis dans les nœuds de profondeur 1, puis dans ceux de profondeur 2, etc. (processus similaire à un parcours en largeur).

$A = [4, 1, 3, 8, 2, 0, 5, 6, 9, 7]$



Si pour tout indice i, on a $A[p(i)] \geq A[i]$, alors on dit que le tableau A (ou l'arbre qui y est associé) est un tas. *En voici un exemple.*

$A = [9, 8, 5, 6, 7, 0, 3, 4, 1, 2]$

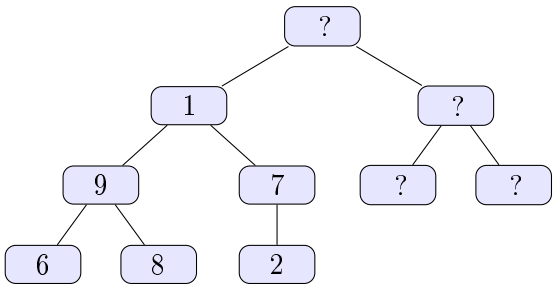


Dans le fichier `TP6_Display.py`, vous trouverez une fonction `display_as_tree` qui permet d'afficher un tableau sous forme d'arbre. Lors de ce TP, à chaque fois que vous modifiez le tableau A, nous vous conseillons d'exécuter `print(A)` et `display_as_tree(A)` afin de vérifier que tout se passe comme prévu.

Exercice 1 Entasser

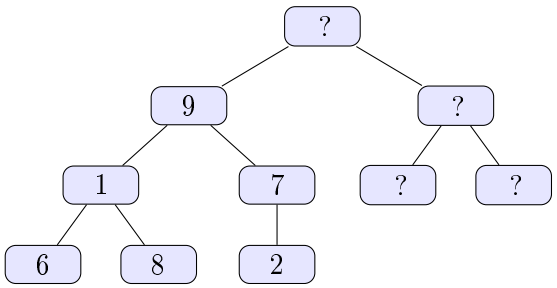
Considérons le tableau suivant (la valeur des points d'interrogation n'a aucune importance pour cet exemple).

$A = [?, 1, ?, 9, 7, ?, ?, 6, 8, 2]$

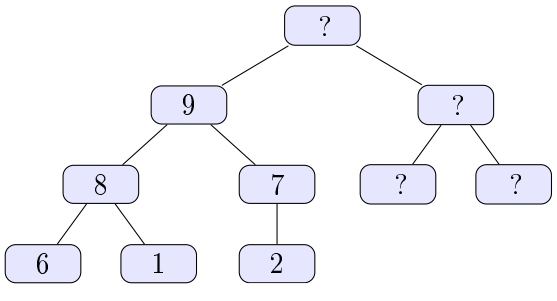


Les arbres enracinés en 9 et 7 sont bien des tas, mais l'arbre enraciné en 1 n'est pas un tas car 1 est plus petit qu'au moins un de ses enfants (en l'occurrence, les deux). Nous allons donc faire descendre 1 « par gravité », jusqu'à obtenir un tas correct. À chaque étape, nous échangeons 1 avec son enfant le plus grand.

$A = [?, 9, ?, 1, 7, ?, ?, 6, 8, 2]$



$A = [?, 9, ?, 8, 7, ?, ?, 6, 1, 2]$



Formellement, ce mécanisme est décrit par l'algorithme 1.

Algorithme 1 : ENTASSER(A, i)

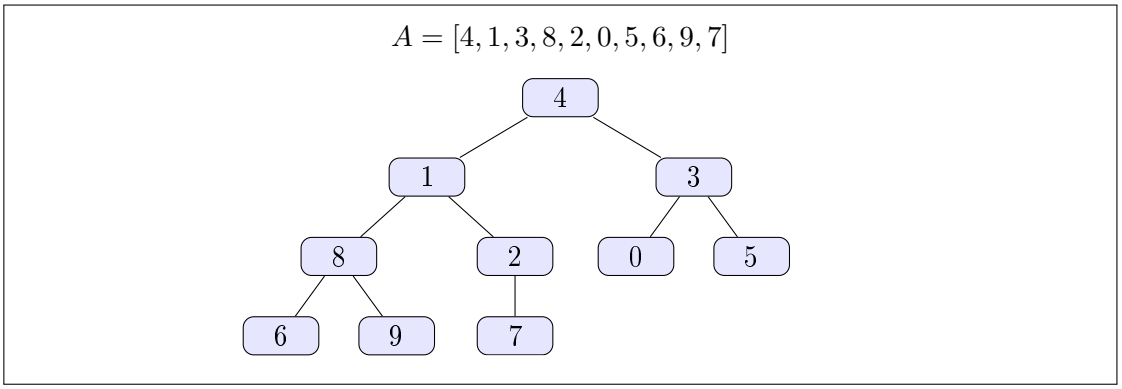
Requiert: Les arbres enfants de la position i dans le tableau A sont des tas.
Assure: L'arbre enraciné en position i dans A est un tas, comprenant les mêmes éléments qu'à l'appel de l'algorithme.

$l \leftarrow 2i + 1$
$r \leftarrow 2i + 2$
$maxi \leftarrow i$
si $l < taille[A]$ et $A[l] > A[maxi]$ **alors**
  $maxi \leftarrow l$
fin si
si $r < taille[A]$ et $A[r] > A[maxi]$ **alors**
  $maxi \leftarrow r$
fin si
si $maxi \neq i$ **alors**
  Échanger $A[i]$ et $A[maxi]$
  ENTASSER$(A, maxi)$
fin si

Écrivez une fonction `siftdown` qui implante cet algorithme. Testez cette fonction sur le tableau `['x', 1, 'x', 9, 7, 'x', 'x', 6, 8, 2]`.

Exercice 2 Convertir en tas

Reprenons notre premier exemple.



Pour convertir ce tableau en tas, nous allons commencer par considérer le dernier nœud possédant au moins un fils (le nœud 2), appliquer l'algorithme 1 pour assurer que l'arbre correspondant soit un tas, puis continuer avec le nœud précédent (8), puis 3, 1 et enfin 4. Formellement, ceci se traduit par l'algorithme 2.

Algorithme 2 : CONVERTIR-EN-TAS(A)

Requiert: A est un tableau.
Assure: A est un tas, comprenant les mêmes éléments qu'à l'appel de l'algorithme.
  pour $i \leftarrow \lfloor taille[A]/2 \rfloor - 1$ à (décroître) 1 **faire**
    ENTASSER(A, i)
  fin pour

Écrivez une fonction `heapify` qui implante cet algorithme.

Exercice 3 Tri par tas

L'algorithme 3 est l'algorithme de **tri par tas** (*heapsort*) qui permet de trier un tableau numérique dans l'ordre croissant.

Algorithme 3 : TRIER-TAS(A)

Requiert: A est un tableau.
Assure: A est trié et comprend les mêmes éléments qu'à l'appel de l'algorithme.
  CONVERTIR-EN-TAS(A)
  $result \leftarrow [\,]$
  tant que A est non vide **faire**
    Échanger $A[0]$ et $A[-1]$
    Supprimer $A[-1]$ et l'insérer au début de $result$.
    ENTASSER($A, 0$)
  fin tant que
  $A \leftarrow result$

Écrivez une fonction `heap_sort` qui implante l'algorithme 3.

Note : Une variante répandue de l'algorithme consiste à conserver un tableau A de taille constante. Dans ce cas, le début de A contient ce qui reste du tas et la fin de A contient les éléments déjà triés (tandis que nous les stockons dans $result$ ci-dessus).