Arbres de recherche

Tas (Heaps)

Définition : Un *arbre partiellement ordonné* est un arbre binaire étiqueté tel que la valeur contenue dans tout nœud est inférieure ou égale aux valeurs contenues dans les sous-arbres de ce nœud.

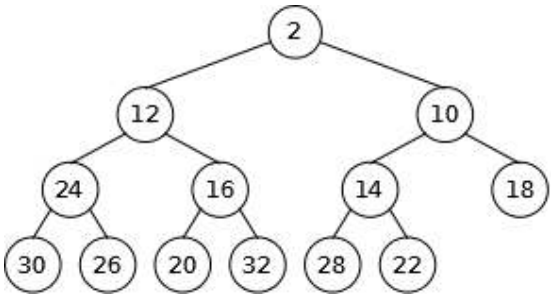
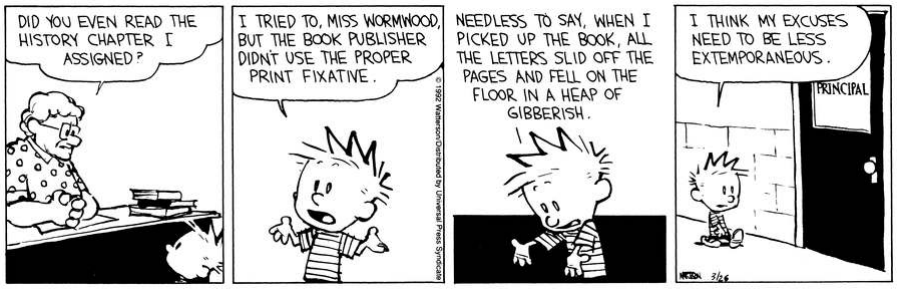


Figure 1 – Arbre parfait partiellement ordonné

Le *tas* permet de représenter un *arbre binaire* ***parfait*** *partiellement ordonné*. L'arbre étant parfait, on utilise un vecteur utilisant la numérotation hiérarchique pour le stocker. Ici, le vecteur contient des couples contenant chacun l'élément et sa valeur pour le tri. La case i du vecteur contient le couple $(value, element)$ de numéro hiérarchique i.

L'arbre de la figure 1 contient les valeurs associées aux éléments suivants :

élément	A	B	C	D	E	F	G	H	I	J	K	L	M
valeur	2	10	12	14	16	18	20	22	24	26	28	30	32



Exercice 1.1 (Représentation par un tas)

1. Soit H le vecteur représentant un arbre binaire parfait, n la taille de cet arbre :
   (a) Où est la racine ?
   (b) Comment retrouver les fils d'un nœud ?
   (c) Comment retrouver le père d'un nœud ?
   (d) Comment savoir si un nœud est une feuille ?
   (e) Comment savoir si un nœud est un point simple ?

2. Donner la représentation par tas (le vecteur contenant des couples (valeur, élément) de l'arbre de la figure 1).

Exercice 1.2 (Utilisation)

1. **Ajout :**
   (a) Comment ajouter un élément à un tas afin qu'il conserve toutes ses propriétés ?
   (b) Ajouter les éléments N de valeur 5 puis O de valeur 15 enfin P de valeur 1 à l'arbre de la figure 1 *(donner l'arbre et le vecteur).*
   (c) Écrire la fonction `heap_push(`H`, `val`, `elt`)` qui ajoute la nouvelle paire (val, elt) au tas H (le tas est modifié en place, pas besoin de le retourner).

```
>>> H = [None]
>>> heap_push(H, 8, 'A')
>>> H
[None, (8, 'A')]
>>> heap_push(H, 9, 'D')
>>> H
[None, (8, 'A'), (9, 'D')]
>>> heap_push(H, 1, 'C')
>>> H
[None, (1, 'C'), (9, 'D'), (8, 'A')]
>>> heap_push(H, 2, 'B')
>>> H
[None, (1, 'C'), (2, 'B'), (8, 'A'), (9, 'D')]
```

2. **Suppression :**
   (a) Comment supprimer d'un tas l'élément de valeur minimum afin qu'il conserve toutes ses propriétés ?
   (b) Supprimer le plus petit élément de l'arbre obtenu à la question précédente (donner l'arbre et le vecteur).
   (c) Écrire la fonction `heap_pop(`H`)` qui supprime et retourne la paire (val, elt) telle que val est la plus petite valeur du tas H (celle en racine). Le tas est modifié en place. La fonction lève une Exception si le tas est vide.

```
>>> H = [None, (1, 'C'), (2, 'B'), (8, 'A'), (9, 'D')]
>>> heap_pop(H)
(1, 'C')
>>> H
[None, (2, 'B'), (9, 'D'), (8, 'A')]
>>> heap_pop(H)
(2, 'B')
>>> H
[None, (8, 'A'), (9, 'D')]
```

Exercice 1.3 (Modification ?)

1. **Minimisation :**
   (a) Dans le tas obtenu à l'exercice 2, la valeur de M change, elle passe à 4. Donner le nouveau tas obtenu après modification de cette valeur.

   (b) Écrire la procédure `heap_update(`H`,` *pos*`,` *newval*`)` qui remplace la valeur de l'élément à la position *pos* dans le tas H par *newval* (le tas est modifié en place, pas besoin de le retourner). La fonction lève une exception si *pos* n'est pas une position valide pour H.

```
>>> H = [None, (1, 'C'), (2, 'B'), (8, 'A'), (9, 'D')]
>>> heap_update(H, 3, 0)
>>> H
[None, (0, 'A'), (2, 'B'), (1, 'C'), (9, 'D')]
```

2. **Optimisation :**
   (a) Quelle serait la complexité d'une fonction permettant de trouver la position d'un élément quelconque ?

   (b) Que faudrait-il ajouter à la représentation pour pouvoir modifier le tas en cas de minimisation d'un élément quelconque du tas en gardant une complexité optimale ?

Exercice 1.4 (Bonus : Tri par tas)

Ecrire une fonction `heap_sort(L)` qui trie (pas en place) la liste associative de paires (val, elt) L en ordre croissant à l'aide d'un tas.

```
>>> L = [(1, 'C'), (8, 'A'), (9, 'D'), (2, 'B')]
>>> heap_sort(L)
[(1, 'C'), (2, 'B'), (8, 'A'), (9, 'D')]
```

Exercice 1.5 (Bonus : Heapify)

Soit T un arbre binaire parfait déjà représenté dans une liste (avec la numérotation hiérarchique). Écrire la fonction `heapify(T)` qui transforme cet arbre en tas en place (pas besoin de retourner le résultat.

```
>>> T0 = [None]
>>> heapify(T0)
>>> T0
[None]

>>> T1 = [None, (3, 'A'), (2, 'B'), (1, 'C')]
>>> heapify(T1)
>>> T1
[None, (1, 'C'), (2, 'B'), (3, 'A')]

>>> T2 = [None, (20, 'G'), (18, 'F'), (28, 'K'), (16, 'E'), (24, 'I'), (2, 'A'),
   (14, 'D'), (32, 'M'), (30, 'L'), (22, 'H'), (10, 'B'), (26, 'J'), (12, 'C')]

>>> heapify(T2)
>>> T2
[None, (2, 'A'), (10, 'B'), (12, 'C'), (16, 'E'), (18, 'F'), (20, 'G'), (14,
   'D'), (32, 'M'), (30, 'L'), (22, 'H'), (24, 'I'), (26, 'J'), (28, 'K')]
```

À noter que la solution n'est pas unique.