8 Tri par tas

8.1 Heapisation

Opération heapify (A) met les éléments du tableau $A[1..n]$ dans l'ordre de tas. Triviale ?

$$H \leftarrow \emptyset\,;\ \textbf{for}\ i \leftarrow 1,\dots,n\ \textbf{do}\ \textsc{insert}(A[i], H, i)\,;\ A \leftarrow H$$

$\Rightarrow$ prend $\Theta(n \log n)$ au pire

Meilleure solution :

```
HEAPIFY(A) // tableau arbitraire A[1..n]
for i ← ⌈n/2⌉, ..., 1 do SINK(A[i], i, A, n)
```

Théorème 8.1. HEAPIFY *met les éléments dans l'ordre de tas en temps* $O(n)$.

Démonstration. SINK prend $O(h)$ temps où h est la hauteur du nœud qui correspond à l'indice i dans la représentation arborescente du tas. Il y a $\leq \lceil n/2^h \rceil$ nœuds internes avec hauteur h. Donc le temps de calcul est borné par $T(n) \leq \sum_{h=1}^{1+\lfloor \lg n \rfloor} \lceil \frac{n}{2^h} \rceil \times O(h) = O(n) \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \right) = O(n)$. ■

8.2 Tri par tas

W(fr)

On peut utiliser une file de priorité pour tri : mettre tous éléments dans la file (heapify), et retirer-les un après l'autre dans l'ordre croissant. Avec un tas binaire, on peut faire le tri en place. Après heapify, on maintient l'ordre de tas dans le préfixe $A[1..i]$ en une boucle $i \leftarrow n, n-1, \dots 2$. Le suffixe $A[i..n]$ est toujours trié en ordre décroissant. À chaque itération, après avoir échangé $A[1] \leftrightarrow A[i]$, on rétablit l'ordre de tas en $A[1..i-1]$ pour la prochaine itération. (On finira avec l'ordre décroissant — pour l'ordre croissant, utiliser un max-tas.)

```
    HEAPSORT(A[1..n])                                  // tableau non-trié
H1  heapify(A)
H2  for i ← n, ... 2 do
H3      v ← A[i] ; A[i] ← A[1]                         // échange A[i] ↔ A[1]
H4      SINK(v, 1, A, i − 1)   // rétablissement de l'ordre de tas en A[1..i − 1]
```

⋆ **heapsort** : temps $O(n \log n)$ dans le pire des cas, sans espace additionnelle !
⋆ **quicksort** : $O(n^2)$ dans le pire des cas
⋆ **mergesort** : $O(n \log n)$ dans le pire des cas mais utilise un espace auxiliaire de taille n.

8.3 Autres implantations de files de priorité

Il existe d'autres implantations (nécessaires pour un merge efficace) : binomial heap, skew heap, Fibonacci heap. L'opération decreaseKey est important dans quelques algorithmes fondamentaux sur des graphes (plus court chemin, arbre couvrant minimal).

	liste triée	**liste non-triée**	**binaire**	***d*-aire**	**binomial**	**skew** (amorti)	**Fibonacci** (amorti)
deleteMin	O(1)	O(n)	O(log n)	O(d log n/ log d)	O(log n)	O(log n)	O(log n)
insert	O(n)	O(1)	O(log n)	O(log n/ log d)	O(log n)	O(1)	O(1)
merge	O(n)	O(1)	O(n)	O(n)	O(log n)	O(1)	O(1)
decreaseKey	O(n)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(1)

Tas d-aire. Au lieu d'un arbre binaire, on peut baser le tas sur un arbre complet avec arité arbitraire $d \geq 2$. On encode l'arbre dans le tableau $A[1..n]$. Parent de l'indice i est $\lceil (i-1)/d \rceil$; les enfants sont à $d(i-1)+2..di+1$. Ordre de tas : W(en)

$$A[i] \geq A\left[\left\lceil \frac{i-1}{d} \right\rceil\right] \qquad \text{pour tout } i > 1$$

- ⋆ deleteMin : $O(d \log_d n)$ dans un tas d-aire sur n éléments
- ⋆ insert : $O(\log_d n)$ dans un tas d-aire sur n éléments
- ⋆ findMin : $O(1)$
- ⋆ SWIM et SINK : $O(\log_d n)$ et $O(d \log_d n)$

⇒ Permet de balancer le coût de l'insertion et de la suppression si on a une bonne idée de leur fréquence.