Tri par tas

Julie Parreaux

2018-2019

Référence du développement : Cormen [1, p.139]

Leçon où on présente le développement : 901 (Structure de données) ; 903 (Algorithmes de tri).

1 Introduction

Le tri par tas est un des tris optimaux par comparaison. Le tri par tas utilise une structure de données pour trié en place et de manière optimale. C'est un exemple de la conception d'un algorithme à l'aide (et autours) d'une structure de données.

Remarques sur le développement

Ce développement présente un tri qui utilise une structure de donnée pour être optimal.

1. Définition de cette structure de données.
2. Étude de la fonction Entasser (principe, algorithme, correction, complexité).
3. Étude de la fonction Construire (principe, algorithme, correction, complexité).
4. Étude de la fonction Tri (principe, algorithme, correction, complexité).

2 Structure de tas binaire

Nous allons étudier la structure de données tas binaire dont les opérations permettent de trier en place un tableau de manière optimale.

Définition. Un tas (binaire, max) est un arbre binaire quasi-complet entassé à gauche dont l'étiquette de chaque nœud est supérieure à celle de ces fils.

Représentation : Un tas peut être vu comme un arbre binaire ou comme un tableau muni de sa taille.
Comment passer d'un arbre à un tableau ? $\mathsf{Parent}(i) = \lfloor \frac{i}{2} \rfloor$ $\mathsf{Gauche}(i) = 2i$ $\mathsf{Droit}(i) = 2i + 1$

La structure de donnée comporte trois opérations :

Entasser	permet de conserver la structure de tas
Construire	permet de construire un tas
Tri	permet de faire un tri d'un tableau en place

La fonction Entasser Son objectif est de conserver la propriété de tas (Algorithme 1). Si cette propriété est violée, on fait alors descendre la clé violant la propriété dans le tas. Pour cela, on fait l'hypothèse que le tas est presque correct (seul notre nœud courant peut violer la propriété) et on rétabli la structure de données. Dans ce cas, la valeur maximale du tas enracinée en notre nœud courant est soit celui-ci soit dans un de ces deux fils.
Hypothèse : $\mathsf{Gauche}(i)$ et $\mathsf{Droit}(i)$ sont des tas max, seul le nœud i peut violer la propriété de tas.

Théorème (Correction)**.** *L'algorithme d'entassement du tas (Algorithme 1) est correcte.*

Algorithm 1 Fonction permettant d'entasser un tas : établir les propriétés du dit tas.

```
1:  function ENTASSER(A, i)                ▷ A est un tas ; i est l'indice du nœud à entasser
2:      l ← Gauche(i)
3:      r ← Droit(i)
4:      max ← i
5:      if l ≤ A.taille et A[l] > A[i] then        ▷ On détermine si max < l
6:          max ← l
7:      end if
8:      if r ≤ A.taille et A[r] > A[i] then        ▷ On détermine si max < r
9:          max ← r
10:     end if
11:     if max ≠ i then                    ▷ Si i n'est pas le max, on échange et on recommence
12:         Échanger A[i] et A[max]
13:         Entasser(A, max)
14:     end if
15: end function
```

Démonstration. A chaque itération, on a trois cas possibles :

Cas 1 : $max = i$ Renvoie un tas enraciné en i

Cas 2 : $max = l$ L'arbre binaire enraciné en r est un tas (par hypothèse comme on ne l'a pas touché). De plus, le nouveau fils gauche l' est un tas (par induction). Comme $max \geq r$ (calcul du maximum) et $max \geq l \geq l'$ (calcul du maximum et propriété du tas enraciné en l avant l'exécution de la fonction).

Cas 3 : $max = r$ Analogue

Donc l'arbre binaire enraciné en max est un tas. □

Théorème (Complexité)**.** *L'algorithme d'entassement du tas (Algorithme 1) s'exécute en* $O(\log n)$.

Démonstration. On effectue une unique descente dans l'arbre binaire du tas (cet arbre est équilibré). Donc, on a une complexité en $O(h)$ où h est la hauteur, soit une complexité en $O(\log n)$.

Une autre manière de le voir : on applique le master theorem à la relation de récurrence suivante :

$$T(n) \leq \underbrace{\frac{2n}{3}}_{\text{taille de l'arbre dans le pire cas car remplit à moitié}} + \underbrace{\Theta(1)}_{\text{complexité des modifs effectuées}}$$

□

La fonction Construire Son objectif est de convertir un tableau en tas : de passer de la représentation d'un tableau simple en structure de données plus complexe : un tas. On va alors appliquer la fonction Entasser à tous les éléments du tableau dans l'ordre décroissant. Cette décroissance nous permet de garantir la correction de notre algorithme.

Algorithm 2 Fonction permettant de construire un tas : on transforme un tableau en un tas.

```
1: function CONSTRUIRE(A)                  ▷ A est un tableau que l'on veut transformer en tas
2:     for i = ⌊A.longueur / 2⌋ à 1 do
3:         Entasser(A, i)                  ▷ on commence par le bas les hypothèses sont ok)
4:     end for
5: end function
```

Théorème (Correction)**.** *L'algorithme de construction d'un tas (Algorithme 2) est correcte.*

Démonstration. Invariant de la boucle FOR : "A chaque itération i, chaque nœuds $i+1, \ldots, n$ est la racine d'un tas max." On le prouve par récurrence sur le nombre d'itération $i \in \{1, \ldots, n\}$.

Initialisation ok car $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$ sont des feuilles d'un tas.

Hérédité Les enfants du nœud i ont des numéros supérieurs à i. Donc, tous les deux sont racines d'un tas max (par l'invariant). L'hypothèse d'application de la fonction Entasser (nécessaire à sa correction) est vérifier, donc l'arbre binaire enraciné en i se transforme en tas max (sans altérer les propriétés sur les nœuds suivants).

Donc l'invariant de la boucle est vérifiée. L'argument clé de cette démonstration est la décroissance de i dans la boucle car elle permet d'établir l'invariant de boucle sur les nœuds suivants. □

Lorsque l'on souhaite calculer la complexité de la fonction Construire, une majoration grossière nous donne $O(n \log n)$. En effet, par la boucle FOR, nous appliquons n fois la fonction Entasser qui s'exécute en $O(\log n)$. Cette complexité est suffisante pour établir la complexité du tri. Cependant, en remarquant que la complexité de la fonction Entasser dépend de la hauteur du tas, on obtient une complexité en $O(n)$.

Théorème (Complexité)**.** *L'algorithme de construction d'un tas (Algorithme 2) s'exécute en $O(n)$.*

Démonstration. On remarque que la complexité de la fonction Entasser dépend de la hauteur du tas. La complexité de la fonction Construire s'exprime donc comme suit :

$$\sum_{h=0}^{\lfloor \log nrfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log nrfloor} \frac{h}{2^h})$$

Comme $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$,

$$O(n \sum_{h=0}^{\lfloor \log nrfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

□

La fonction Tri Son objectif est de trier un tableau. Pour cela, on construit un tas max à partir du tableau, puis pour chaque racine on la sort du tas et on recommence.

Algorithm 3 Fonction permettant de trier un tableau à l'aide d'un tas.

```
1: function TRI-PAR-TAS(A)                    ▷ A est un tableau que l'on souhaite trier
2:     Construire(A)
3:     for i = A.longueur à 2 do
4:         Échanger A[1] et A[i]               ▷ On sort la racine du tas
5:         A.taille ← A.taille − 1
6:         Entasser(A, i)                      ▷ On recommence
7:     end for
8: end function
```

Théorème (Correction)**.** *L'algorithme de tri par tas (Algorithme 3) est correcte.*

Démonstration. Invariant de la boucle FOR : "A chaque itération i, chaque sous tableau $A[A.taille, :]$ est trié et $A[:, A.taille]$ est un tas." On le prouve en appliquant la correction des deux autres. □

Théorème (Complexité)**.** *L'algorithme de tri par tas (Algorithme 3) s'exécute en $O(n \log n)$.*

Démonstration. On exécute l'opération Construire un fois en $O(n)$. Puis on exécute n fois (pour la boucle), l'opération Entasser. On a donc une complexité en $O(n \log n)$ □

3 Application à la file de priorité

Définition. Une file de priorité [1, p.149] est une structure de données permettant la gestion d'un ensemble S via leur clé. On a les opérations suivante : Insérer, Max, Extraire-Max et Augmenter-Clé.

Remarque. On a les même en MIN

Représentation : le tas binaire

Algorithm 4 Fonction permettant de donner le maximum d'une file de priorité.

```
1: function MAX(A)                          ▷ A : file de priorité
2:     A[1]
3: end function
```

Algorithm 5 Fonction permettant d'insérer un nouvel élément dans une file de priorité.

```
1: function MAX(A, cle)                     ▷ A : file de priorité ; cle : clé à insérer
2:     A.taille ← A.taille + 1
3:     A[A.taille] ← −∞
4:     Augmenter-Clé(A, A.taille, cle)
5: end function
```

Algorithm 6 Fonction permettant de retirer le maximum d'une file de priorité.

```
1: function EXTRAIRE-MAX(A)                 ▷ A : file de priorité
2:     if A.taille < 1 then
3:         Erreur
4:     end if
5:     max ← A[1]
6:     A[1] ← A[A.taille]
7:     A.taille ← A.taille − 1
8:     Entasser(A, 1)
9:     Renvoie max
10: end function
```

Algorithm 7 Fonction permettant d'augmenter la clé d'un élément dans une file de priorité.

```
1: function AUGMENTER-CLÉ(A, i, cle)        ▷ A : file de priorité ; i : position de l'élément ; cle : clé à augmenter
2:     if cle < A[i] then
3:         Erreur
4:     end if
5:     A[i] ← cle
6:     while i > 1 et A[Parent(i)] < A[i] do
7:         Échanger A[i] et A[Parent(i)]
8:         i ← Parent(i)
9:     end while
10: end function
```

Les fonctions ainsi définies pour implémenter les opérations de la file de priorité sont : Max en $O(1)$, Extraire-Max en $O(\log n)$ (appel Entasser), Augmenter-Clé en $O(\log n)$ (descend dans l'arbre) et insérer en $O(\log n)$ (appel Augmenter-Clé).

Amélioration : tas de Fibonacci [1, p.467] Une amélioration possible du tas binaire est le tas de Fibonacci qui est une amélioration d'un tas fusionnable.

Définition. Un tas fusionnable est une structure de données implémentant les cinq opérations suivantes : Créer, Insérer, Min, Extraire-Min et Union.

Un tas de Fibonacci est un tas fusionnable muni des deux opérations suivantes : Diminuer-Clé et Supprimer. C'est un ensemble d'arbre ordonnés enraciné en tas min (chacun des tas est un tas min). Chaque nœud est relié à ses enfants, ses parents et ses frères avec une liste doublement chaînée circulaire. On ajoute également un pointeur qui indique quel est l'élément minimal du tas (nécessairement une racine). Les opérations de ce tas sont alors implémentées comme suit :

Insérer On insère le nouveau nœud comme une racine d'un nouvel arbre. On vérifie ensuite qu'il n'est pas le plus petit.

Min On donne l'élément pointé par le pointeur sur l'élément minimal.

Union On concatène les deux listes de racines. Ensuite on regarde lequel des deux minimum est réellement le minimum du tas.

Extraire-Max On transforme les fils de l'élément minimal en racine que l'on supprime. On appelle alors une fonction de consolidation qui relie les racines de degré égal jusqu'à obtention d'une unique racine par degré. On trouve deux racines et on en fait un tas : la plus grande racine vient se greffer

Opération	Tas binaire (pire cas)	Tas de Fibonacci (amortie)
Créer	$\Theta(1)$	$\Theta(1)$
Insérer	$\Theta(\log n)$	$\Theta(1)$
Min	$\Theta(1)$	$\Theta(1)$
Extraire-Min	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
Diminuer-Clé	$\Theta(\log n)$	$\Theta(1)$
Supprimer	$\Theta(\log n)$	$\Theta(\log n)$

TABLE 1 – Comparaison des complexités en fonction de la structure de tas (certaines sont admises).

sur l'autre. C'est lors de l'extraction du minimum que l'on diminue la taille de la liste des racines de notre tas.

Applications des files de priorité Dans la pratique, les files de priorité servent à beaucoup de chose. Elles améliorent des algorithmes qui utilise en grand nombre l'extraction d'un minimum ou de la suppression dans un ensemble de données. Dans le cas de graphes denses qui appelle à la diminution de clé (exemple de problème de flots), elles sont très utilisées. Elles sont, notamment, le support d'algorithmes de calcul d'arbre couvrant minimal (algorithme de Prim [1, p.586]), de recherche de chemin le plus cours (algorithme de Dijstra [1, p.609]). Cependant, elles sont absolument inefficace pour la recherche.

Références

[1] Rivest R. Stein C. Cormen T., Leiserson C. *Algorithmique, 3ème édition*. Dunod, 2010.