Université Assane Seck de Ziguinchor

COMPLEXITE ET PERFORMANCES DES ALGORITHMES

Dr. Youssou DIENG

Grade Maitre de conférences

Département: …Informatique……

TRI PAR TAS

I-INTRODUCTION

Dans ce chapitre, nous présentons un nouvel algorithme de tri. Comme le tri par fusion, mais contrairement au tri par insertion, le temps d'exécution du tri par tas est $O(n \lg n)$.

Comme le tri par insertion, mais contrairement au tri par fusion, le tri par tas trie sur place : à tout moment, jamais plus d'un nombre constant d'éléments de tableau ne se trouvera stocké hors du tableau d'entrée. Le tri par tas rassemble donc le meilleur des deux algorithmes de tri que nous avons déjà étudiés.

Le tri par tas introduit aussi une autre technique de conception des algorithmes : on utilise une structure de données, appelée ici « tas » pour gérer les informations pendant l'exécution de l'algorithme. Non seulement le tas est une structure de données utile pour le tri par tas, mais il permet aussi de construire une file de priorités efficace. Le terme « tas » fut, au départ, inventé dans le contexte du tri par tas, mais il a depuis pris le sens de « mémoire récupérable » (garbage-collected storage) comme celle fournie par les langages Lisp et Java.

TAS

La structure de tas (binaire) est un tableau qui peut être vu comme un arbre binaire presque complet. Chaque nœud de l'arbre correspond à un élément du tableau qui contient la valeur du nœud. L'arbre est complètement rempli à tous les niveaux, sauf éventuellement au niveau le plus bas, qui est rempli en partant de la gauche et jusqu'à un certain point.

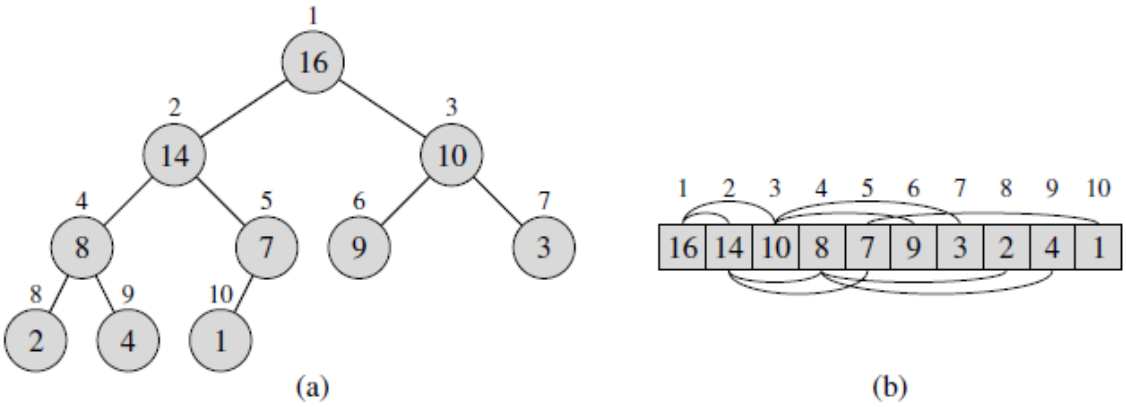
Un tableau A représentant un tas est un objet ayant deux attributs : *longueur*[A], nombre d'éléments du tableau, et *taille*[A], nombre d'éléments du tas rangés dans le tableau A. Autrement dit, bien que A[1 . . *longueur*[A]] puisse contenir des nombres valides, aucun élément après A[*taille*[A]], où *taille*[A] _ *longueur*[A], n'est un élément du tas. La racine de l'arbre est A[1], et étant donné l'indice i d'un nœud, les indices de son parent PARENT(i), de son enfant de gauche GAUCHE(i) et de son enfant de droite DROITE(i) peuvent être facilement calculés :

```
PARENT(i)
    retourner ⌊i/2⌋

GAUCHE(i)
    retourner 2i

DROITE(i)
    retourner 2i + 1
```

Sur la plupart des ordinateurs, la procédure GAUCHE peut calculer $2i$ en une seule instruction, en décalant simplement d'une position vers la gauche la représentation binaire de i. De même, la procédure DROITE peut calculer rapidement $2i + 1$ en décalant d'une position vers la gauche la représentation binaire de i et en ajoutant un 1 comme bit de poids faible. La procédure PARENT peut calculer $i/2$ en décalant i d'une position binaire vers la droite. Dans une bonne implémentation du tri par tas, ces trois procédures sont souvent implémentées en tant que « macros », ou procédures « en ligne ».



Un tas-max vu comme **(a)** un arbre binaire et **(b)** un tableau. Le nombre à l'intérieur du cercle, à chaque nœud de l'arbre, est la valeur contenue dans ce nœud. Le nombre au-dessus d'un nœud est l'indice correspondant dans le tableau. Au-dessus et au-dessous du tableau sont des lignes matérialisant les relations parent-enfant ; les parents sont toujours à gauche de leurs enfants. L'arbre a une hauteur de trois ; le nœud d'indice 4 (avec la valeur 8) a une hauteur de un.

Il existe deux sortes de tas binaires : les tax-max et les tas-min. Pour ces deux types de tas, les valeurs des nœuds satisfont à une ***propriété de tas***, dont la nature spécifique dépend du type de tas. Dans un ***tas max***, la ***propriété de tas max*** est que, pour chaque nœud *i* autre que la racine,

$$\mathbf{A[PARENT(i)] \geq A[i]\,,}$$

En d'autres termes, la valeur d'un nœud est au plus égale à celle du parent. Ainsi, le plus grand élément d'un tas max est stocké dans la racine, et le sous-arbre issu d'un certain nœud contient des valeurs qui ne sont pas plus grandes que celle du nœud lui-même. Un ***tas min*** est organisé en sens inverse ; la ***propriété de tas min*** est que, pour chaque nœud *i* autre que la racine,

$$\mathbf{A[PARENT(i)] \leq A[i]\,.}$$

Le plus petit élément d'un tas min est à la racine.

Pour l'algorithme du tri par tas, on utilise des tas max. Les tas min servent généralement dans les files de priorités. Nous préciserons si nous prenons un tas max ou un tas-min pour une application particulière ; pour les propriétés qui s'appliquent indifféremment aux tax-max et aux tas-min, nous parlerons de « tas » tout court.

En considérant un tas comme un arbre, on définit la ***hauteur*** d'un noeud dans un tas comme le nombre d'arcs sur le chemin simple le plus long reliant le noeud à une feuille, et on définit la hauteur du tas comme étant la hauteur de sa racine. Comme un tas de n éléments est basé sur un arbre binaire complet, sa hauteur est $\boldsymbol{\Theta}(\lg n)$. On verra que les opérations élémentaires sur les tas s'exécutent dans un temps au plus proportionnel à la hauteur de l'arbre et prennent donc un temps $O(\lg n)$. Le reste de ce chapitre présentera quelques procédures élémentaires et montrera comment elles peuvent être utilisées dans un algorithme de tri et dans une structure de données représentant une file de priorité.

- La procédure ENTASSER-MAX, qui s'exécute en $O(\lg n)$, est la clé de voûte de la conservation de la propriété de tas max.
- La procédure CONSTRUIRE-TAS-MAX, qui s'exécute en un temps linéaire, produit un tas max à partir d'un tableau d'entrée non-ordonné.
- La procédure TRI-PAR-TAS, qui s'exécute en $O(n \lg n)$, trie un tableau sur place.

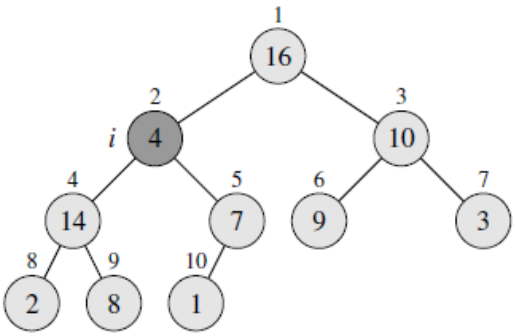
- Les procédures INSÉRER-TAS-MAX, EXTRAIRE-MAX-TAS, AUGMENTERCLÉ-TAS et MAXIMUM-TAS, qui s'exécutent en $O(\lg n)$, permettent d'utiliser la structure de données de tas pour gérer une file de priorité.

CONSERVATION DE LA STRUCTURE DE TAS

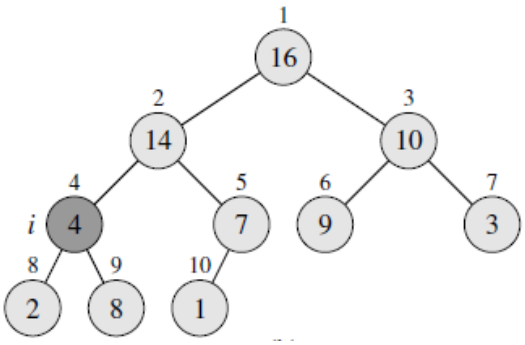
ENTASSER-MAX est un sous-programme important pour la manipulation des tas max, qui prend en entrée un tableau A et un indice i. Quand ENTASSER-MAX est appelée, on suppose que les arbres binaires enracinés en GAUCHE(i) et DROITE(i) sont des tas max, mais que $A[i]$ peut être plus petit que ses enfants, violant ainsi la propriété de tas max. Le rôle de ENTASSER-MAX est de faire « descendre » la valeur de $A[i]$ dans le tas max de manière que le sous-arbre enraciné en i devienne un tas-max.

```
ENTASSER-MAX(A, i)
 1  l ← GAUCHE(i)
 2  r ← DROITE(i)
 3  si l ⩽ taille[A] et A[l] > A[i]
 4     alors max ← l
 5     sinon max ← i
 6  si r ⩽ taille[A] et A[r] > A[max]
 7     alors max ← r
 8  si max ≠ i
 9     alors échanger A[i] ↔ A[max]
10           ENTASSER-MAX(A, max)
```

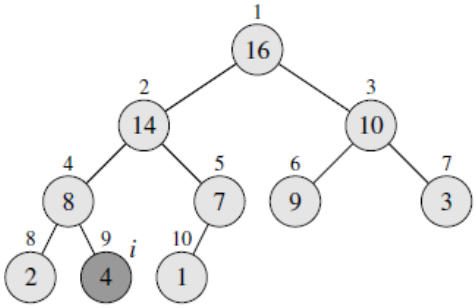
La figure suivante illustre l'action de ENTASSER-MAX. A chaque étape, on détermine le plus grand des éléments $A[i]$, A[GAUCHE(i)], et A[DROITE(i)], et son indice est rangé dans *max*. Si $A[i]$ est le plus grand, alors le sous-arbre enraciné au noeud i est un tas max et la procédure se termine. Sinon, c'est l'un des deux enfants qui contient l'élément le plus grand, auquel cas $A[i]$ est échangé avec $A[max]$, ce qui permet au noeud i et à ses enfants de satisfaire la propriété de tas max. Toutefois, le noeud indexé par *max* contient maintenant la valeur initiale de $A[i]$, et donc le sous-arbre enraciné en *max* viole peut-être la propriété de tas max. ENTASSER-MAX doit donc être appelée récursivement sur ce sous-arbre.



(a)



(b)



(c)

L'action de ENTASSER-MAX(*A*, 2), où *taille*[*A*] = 10. **(a)** La configuration initiale du tas, avec la valeur *A*[2] au nœud *i* = 2 viole la propriété de tas max puisqu'elle n'est pas supérieure à celle des deux enfants. La propriété de tas max est restaurée pour le nœud 2 en **(b)** via échange de *A*[2] avec *A*[4], ce qui détruit la propriété de tas max pour le nœud 4. L'appel récursif ENTASSER-MAX(*A*, 4) prend maintenant *i* = 4. Après avoir échangé *A*[4] avec *A*[9], comme illustré en **(c)**, le nœud 4 est corrigé, et l'appel récursif ENTASSER-MAX(*A*, 9) n'engendre plus de modifications de la structure de données.

Le temps d'exécution de ENTASSER-MAX sur un sous-arbre de taille *n* enraciné en un noeud *i* donné est le temps $\Theta(1)$ nécessaire pour corriger les relations entre les éléments *A*[*i*], *A*[GAUCHE(*i*)], et *A*[DROITE(*i*)], plus le temps d'exécuter ENTASSERMAX sur un sous-arbre enraciné sur l'un des enfants du noeud *i*. Les sous-arbres des enfants ont chacun une taille au plus égale à 2*n*/3 (le pire des cas survient quand la dernière rangée de l'arbre est remplie exactement à moitié), et le temps d'exécution de la procédure ENTASSER-MAX peut donc être décrit par la récurrence

$$T(n) \leq T(2n/3) + \Theta(1) .$$

La solution de cette récurrence, d'après le cas 2 du théorème général au chapitre 4, est $T(n) = O(\lg n)$. On peut également caractériser le temps d'exécution de ENTASSER-MAX sur un noeud de hauteur h par $O(h)$.

CONSTRUCTION D'UN TAS

On peut utiliser la procédure ENTASSER-MAX à l'envers pour convertir un tableau $A[1 . . n]$, avec $n = length[A]$, en tas max. Il n'est pas difficile de remarquer que, les éléments du sous-tableau $A[(n/2 + 1) . . n]$ sont tous des feuilles de l'arbre, et donc chacun est initialement un tas à 1 élément. La procédure CONSTRUIRE-TAS-MAX parcourt les autres nœuds de l'arbre et appelle ENTASSER-MAX pour chacun.

```
CONSTRUIRE-TAS-MAX(A)
1  taille[A] ← longueur[A]
2  pour i ← ⌊longueur[A]/2⌋ jusqu'à 1
3      faire ENTASSER-MAX(A, i)
```

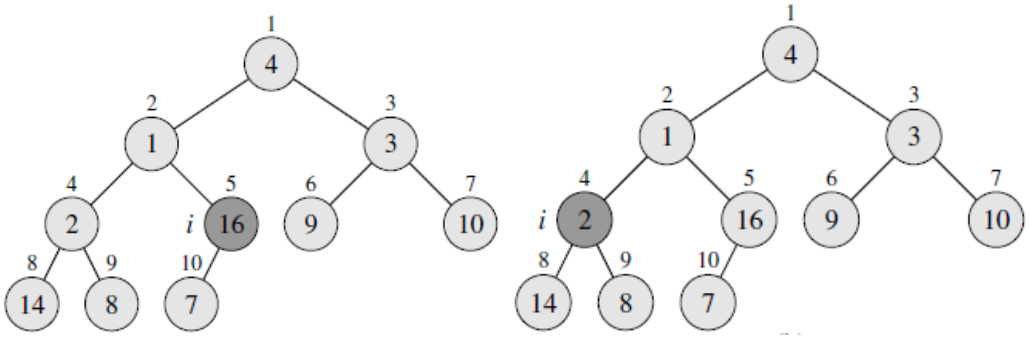
La figure suivante montre un exemple de l'action de CONSTRUIRE-TAS-MAX. Pour prouver la conformité de CONSTRUIRE-TAS-MAX, on utilise l'invariant de boucle suivant :

- Au début de chaque itération de la boucle **pour** des lignes 2–3, chaque nœud $i + 1, i + 2, . . . , n$ est la racine d'un tas max.

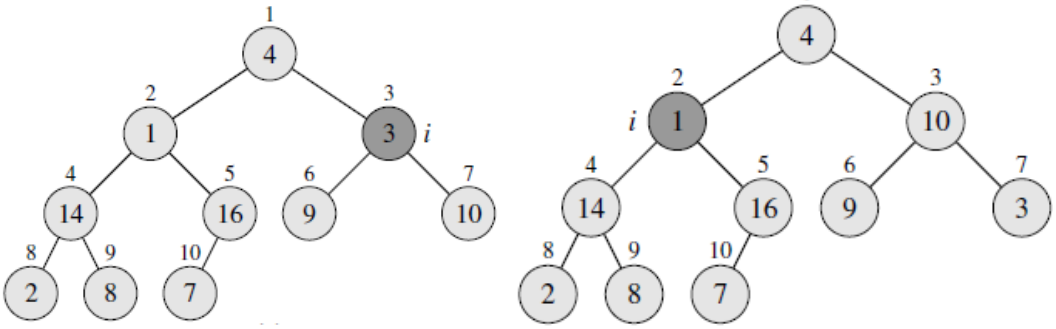
Il faut montrer que cet invariant est vrai avant la première itération, que chaque itération le conserve et qu'il fournit une propriété permettant de prouver la conformité après la fin de l'exécution de la boucle.

Initialisation : Avant la première itération de la boucle, $i = n/2$. Chaque nœud $n/2+1, n/2+2, . . . , n$ est une feuille et est donc la racine d'un tas max trivial.

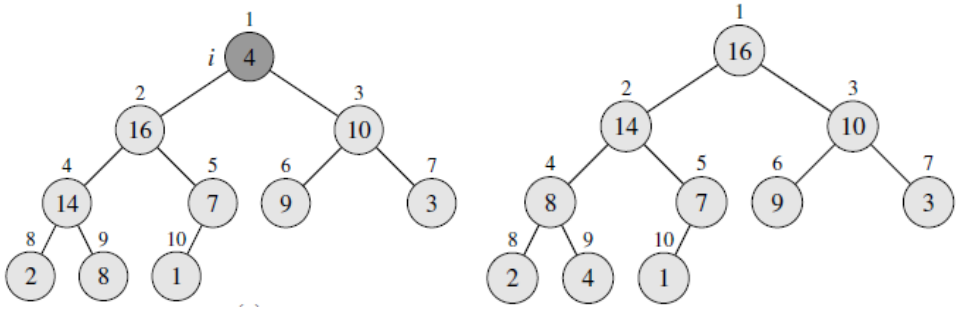
A	4	1	3	2	16	9	10	14	8	7



(a) (b)



(c) (d)



(e) (f)

Fonctionnement de CONSTRUIRE-TAS-MAX, montrant la structure de données avant l'appel à ENTASSER-MAX en ligne 3 de CONSTRUIRE-TAS-MAX. **(a)** Un tableau de 10 éléments en entrée, avec l'arbre binaire qu'il représente. La figure montre que l'indice de boucle i pointe vers le nœud 5 avant l'appel ENTASSER-MAX(A, i). **(b)** La structure de données résultante. L'indice de boucle i de l'itération suivante pointe vers le nœud 4. **(c)–(e)** Les itérations suivantes de la boucle **pour** de CONSTRUIRE-TAS-MAX. Observez que, chaque fois qu'il y a appel de ENTASSERMAX sur un nœud, les deux sous arbres de ce nœud sont des tas max. **(f)** Le tas max après que CONSTRUIRE-TAS-MAX a fini.

Conservation : Pour voir que chaque itération conserve l'invariant, observez que les enfants du nœud i ont des numéros supérieurs à i. D'après l'invariant, ce sont donc tous les deux des racines de tas max. Telle est précisément la condition exigée pour que l'appel ENTASSER-MAX(A, i) fasse du nœud i une racine de tas max. En outre, l'appel ENTASSER-MAX préserve la propriété que les nœuds $i + 1$, $i + 2$, . . . , n sont tous des racines de tas max. La décrémentation de i dans la partie actualisation de la boucle **pour** a pour effet de rétablir l'invariant pour l'itération suivante.

Terminaison : À la fin, $i = 0$. D'après l'invariant, chaque nœud 1, 2, . . . , n est la racine d'un tas max. En particulier, le nœud 1.

On peut calculer un majorant simple pour le temps d'exécution de CONSTRUIRETAS-MAX de la manière suivante. Chaque appel à ENTASSER-MAX coûte $O(\lg n)$, et il existe $O(n)$ appels de ce type. Le temps d'exécution est donc $O(n \lg n)$. Ce majorant, quoique correct, n'est pas asymptotiquement serré.

On peut trouver un majorant plus fin en observant que le temps d'exécution de ENTASSER-MAX sur un nœud varie avec la hauteur du nœud dans l'arbre, et que les hauteurs de la plupart des nœuds sont réduites. Notre analyse plus fine s'appuie sur les propriétés d'un tas à n éléments : sa hauteur est $\lfloor \lg n \rfloor$ et le nombre de nœuds ayant la hauteur h est au plus $\lceil n/2^{h+1} \rceil$

Le temps requis par ENTASSER-MAX quand elle est appelée sur un nœud de hauteur h étant $O(h)$, on peut donc exprimer le coût total de CONSTRUIRE-TAS-MAX ainsi

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

La dernière sommation peut être évaluée en substituant $x = 1/2$ dans la formule, ce qui donne

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

Donc, le temps d'exécution de CONSTRUIRE-TAS-MAX peut être borné par

$$O\left(n\sum_{h=0}^{\lfloor \lg n \rfloor}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$
$$= O(n)\,.$$

ALGORITHME DU TRI PAR TAS

L'algorithme du tri par tas commence par utiliser CONSTRUIRE-TAS-MAX pour construire un tas max à partir du tableau $A[1\,..\,n]$, où $n = longueur[A]$. Comme l'élément maximal du tableau est stocké à la racine $A[1]$, on peut le placer dans sa position finale correcte en l'échangeant avec $A[n]$. Si l'on « ôte » à présent le nœud n du tas (en décrémentant $taille[A]$), on observe que $A[1\,..\,(n-1)]$ peut facilement être transformé en tas max. Les enfants de la racine restent des tas max, mais la nouvelle racine risque d'enfreindre la propriété de tas max. Pour restaurer la propriété de tas max, il suffit toutefois d'appeler une seule fois ENTASSER-MAX(A, 1) qui laisse un tas max dans $A[1\,..\,(n-1)]$. L'algorithme du tri par tas répète alors ce processus pour le tas max de taille $n-1$ jusqu'à arriver à un tas de taille 2.

```
TRI-PAR-TAS(A)
1  CONSTRUIRE-TAS-MAX(A)
2  pour i ← longueur[A] jusqu'à 2
3      faire échanger A[1] ↔ A[i]
4           taille[A] ← taille[A] − 1
5           ENTASSER-MAX(A, 1)
```

La figure 6.4 montre l'action du tri par tas juste après la construction du tas max. Chaque tas max est montré au début d'une itération de la boucle **pour** des lignes 2–5. La procédure TRI-PAR-TAS prend un temps $O(n \lg n)$, puisque l'appel à CONSTRUIRE-TAS-MAX prend un temps $O(n)$ et que chacun des $n-1$ appels à ENTASSER-MAX prend un temps $O(\lg n)$.

FILES DE PRIORITÉ

Le tri par tas est un excellent algorithme, mais une bonne implémentation du tri rapide

(quicksort), est généralement plus performante en pratique. Néanmoins, la structure de données tas offre intrinsèquement de gros avantages. Dans cette section, nous allons présenter l'une des applications les plus répandues du tas, à savoir la gestion d'une file de priorité efficace. Comme c'est le cas avec les tas, il existe deux sortes de files de priorité : les files max et les files min. Nous nous concentrerons ici sur la mise en oeuvre des files de priorité max, lesquelles s'appuient sur des tas max.

Une ***file de priorité*** est une structure de données qui permet de gérer un ensemble S d'éléments, dont chacun a une valeur associée baptisée ***clé***. Une ***file de priorité max*** reconnaît les opérations suivantes.

- INSÉRER(S, x) insère l'élément x dans l'ensemble S. Cette opération pourrait s'écrire sous la forme $S \leftarrow S \cup \{x\}$.
- MAXIMUM(S) retourne l'élément de S ayant la clé maximale.
- EXTRAIRE-MAX(S) supprime et retourne l'élément de S qui a la clé maximale.
- AUGMENTER-CLÉ(S, x, k) accroît la valeur de la clé de l'élément x pour lui donner la nouvelle valeur k, qui est censée être supérieure ou égale à la valeur courante de la clé de x.

L'une des applications des files de priorité max est de planifier les tâches sur un ordinateur. La file max gère les travaux en attente, avec leurs priorité relatives. Quand une tâche est terminée ou interrompue, l'ordinateur exécute la tâche de plus forte priorité, sélectionnée via EXTRAIRE-MAX parmi les travaux en attente. La procédure INSÉRER permet d'ajouter à tout moment une nouvelle tâche à la file.

Une ***file de priorité min*** reconnaît les opérations INSÉRER, MINIMUM, EXTRAIREMIN et DIMINUER-CLÉ. Une file de priorité min peut servir à gérer un simulateur piloté par événement. Les éléments de la file sont des événements à simuler, chacun étant affecté d'un temps d'occurrence qui lui sert de clé. Les événements doivent être simulés dans l'ordre des temps d'occurrence, vu que la simulation d'un événement risque d'entraîner la simulation ultérieure d'autres événements. Le programme de simulation emploie EXTRAIRE-MIN à

chaque étape pour choisir le prochain événement à simuler. À mesure que sont produits de nouveaux événements, ils sont insérés dans la file de priorité min *via* INSÉRER.

Il n'y a rien d'étonnant à ce qu'un tas permette de gérer une file de priorité. Dans une application donnée, par exemple dans un planificateur de tâches ou dans un simulateur piloté par événement, les éléments de la file de priorité correspondent à des objets de l'application. Il est souvent nécessaire de déterminer quel est l'objet de l'application qui correspond à un certain élément de la file de priorité, et *vice-versa*.

Quand on utilise un tas pour gérer une file de priorité, on a donc souvent besoin de stocker dans chaque élément du tas un ***repère*** référençant l'objet applicatif correspondant. La nature exacte du repère (pointeur ? entier ? etc.) dépend de l'application. De même, on a besoin de stocker dans chaque objet applicatif un repère référençant l'élément de tas correspondant. Ici, le repère est généralement un indice de tableau. Comme les éléments du tas changent d'emplacement dans le tableau lors des opérations de manipulation du tas, une implémentation concrète doit, en cas de relogement d'un élément du tas, actualiser en parallèle l'indice de tableau dans l'objet applicatif correspondant. Comme les détails de l'accès aux objets de l'application dépendent grandement de l'application et de son implémentation, nous n'en parlerons plus, si ce n'est pour noter que, dans la pratique, il importe de gérer correctement ces repères.

Nous allons maintenant voir comment implémenter les opérations associées à une file de priorité max. La procédure MAXIMUM-TAS implémente l'opération MAXIMUM en un temps $\Theta(1)$.

```
MAXIMUM-TAS(A)
1  retourner A[1]
```

La procédure EXTRAIRE-MAX-TAS implémente l'opération EXTRAIRE-MAX. Elle ressemble au corps de la boucle **pour** (lignes 3-5) de la procédure TRI-PARTAS.

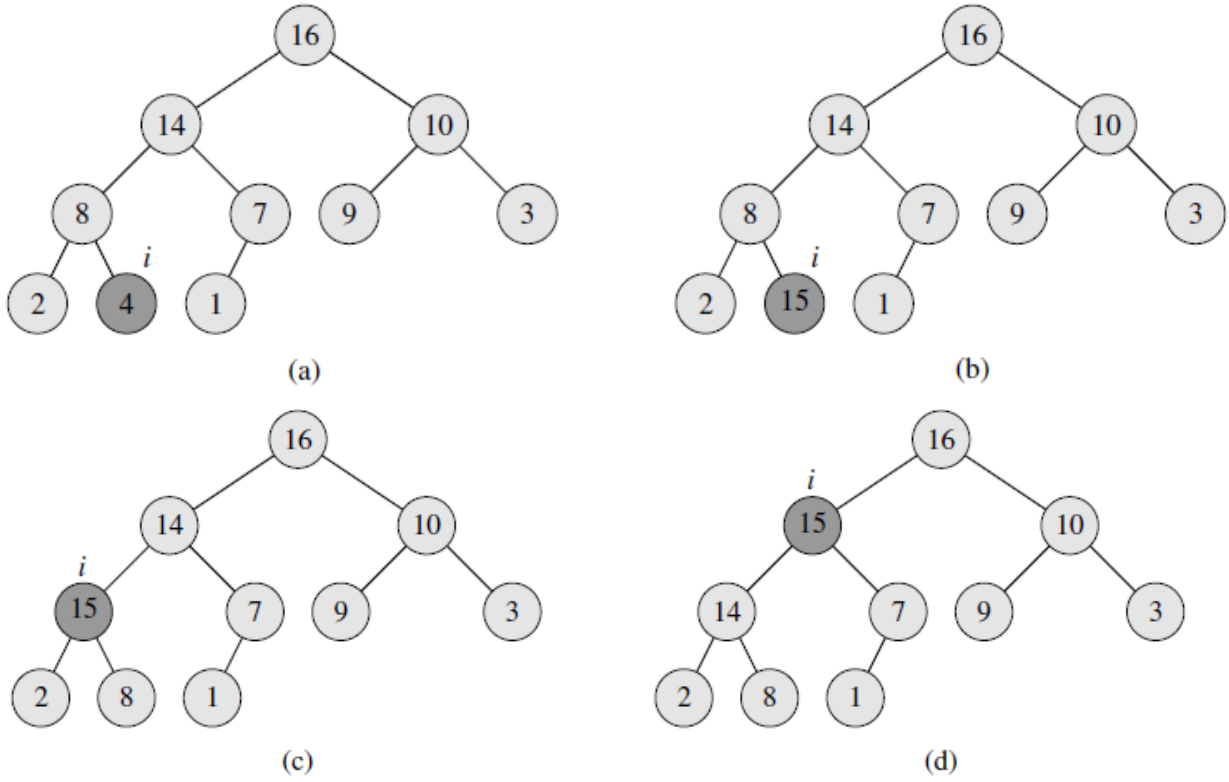
```
EXTRAIRE-MAX-TAS(A)
1  si taille[A] < 1
2     alors erreur « limite inférieure dépassée »
3  max ← A[1]
4  A[1] ← A[taille[A]]
5  taille[A] ← taille[A] − 1
6  ENTASSER-MAX(A, 1)
7  retourner max
```

Le temps d'exécution de EXTRAIRE-MAX-TAS est $O(\lg n)$, car elle n'effectue qu'un volume constant de travail en plus du temps $O(\lg n)$ de ENTASSER-MAX.

La procédure AUGMENTER-CLÉ-TAS implémente l'opération AUGMENTERCLÉ. L'élément de la file de priorité dont il faut accroître la clé est identifié par un indice i pointant vers le tableau. La procédure commence par modifier la clé de l'élément $A[i]$ pour lui donner sa nouvelle valeur. Accroître la clé de $A[i]$ risque d'enfreindre la propriété de tas max ; la procédure, d'une manière qui rappelle la boucle d'insertion (lignes 5–7) de TRI-INSERTION vue à la section 2.1, parcourt donc un chemin reliant ce noeud à la racine, afin de trouver une place idoine pour la clé qui vient d'être augmentée. Tout au long du parcours, elle compare un élément à son parent ; elle permute les clés puis continue si la clé de l'élément est plus grande, et elle s'arrête si la clé de l'élément est plus petite, vu que la propriété de tas max est alors satisfaite.

```
AUGMENTER-CLÉ-TAS(A, i, clé)
1  si clé < A[i]
2     alors erreur « nouvelle clé plus petite que clé actuelle »
3  A[i] ← clé
4  tant que i > 1 et A[PARENT(i)] < A[i]
5     faire permuter A[i] ↔ A[PARENT(i)]
6          i ← PARENT(i)
```

La figure suivante montre un exemple d'opération AUGMENTER-CLÉ-TAS. Le temps d'exécution de AUGMENTER-CLÉ-TAS sur un tas à n éléments est $O(\lg n)$, car le chemin reliant le noeud, modifié en ligne 3, à la racine a la longueur $O(\lg n)$.



Cette figure on illustre le fonctionnement de AUGMENTER-CLÉ-TAS. **(a)** Le tas max de la figure 6.4(a) avec un noeud dont l'indice est *i* en gris foncé. **(b)** Ce noeud voit sa clé passer à 15. **(c)** Après une itération de la boucle **tant que** des lignes 4–6, le noeud et son parent s'échangent leurs clés et l'indice *i* remonte pour devenir celui du parent. **(d)** Le tas max après une itération supplémentaire de la boucle **tant que**. À ce stade, $A[\text{PARENT}(i)] \_ A[i]$. La propriété de tas max étant maintenant vérifiée, la procédure se termine.

La procédure INSÉRER-TAS-MAX implémente l'opération INSÉRER. Elle prend en entrée la clé du nouvel élément à insérer dans le tas max *A*. La procédure commence par étendre le tas max en ajoutant à l'arbre une nouvelle feuille dont la clé est $-\infty$. Elle appelle ensuite AUGMENTER-CLÉ-TAS pour affecter à la clé du nouveau noeud la bonne valeur et conserver la propriété de tas max.

```
INSÉRER-TAS-MAX(A, clé)
1  taille[A] ← taille[A] + 1
2  A[taille[A]] ← −∞
3  AUGMENTER-CLÉ-TAS(A, taille[A], clé)
```

Le temps d'exécution de INSÉRER-TAS-MAX sur un tas à *n* éléments est $O(\lg n)$. En résumé, un tas permet de faire toutes les opérations de file de priorité sur un ensemble de taille *n* en un temps $O(\lg n)$.