Chapitre 6

Transformer pour régner

L’approche « transformer pour régner »

- Cette méthode de conception regroupe plusieurs techniques qui s’appuient toutes sur l’idée générale de transformer les instances pour qu’elles soient plus faciles à solutionner.
- Il existe trois grandes classes de transformations:
  - **Simplification de l’instance**: l’instance est transformée en une instance du même problème mais qui est plus simple (et moins coûteuse) à traiter.
    - Ex: « pré triage »: on trie le tableau avant de le traiter
  - **Changement de représentation**: l’instance est représentée sous une autre forme plus adaptée au problème initial
    - Ex: on transforme un tableau en un tas (monceau)
  - **Réduction** (d’un problème à un autre): le problème initial est transformé en un autre problème pour lequel on possède un algorithme efficace
    - Il faut alors toujours pouvoir reconstruire la solution du problème initial à partir de la solution du nouveau problème (exemples au chapitre 10)

Le « pré triage »

- Plusieurs opérations sur un tableau s'effectuent plus rapidement lorsqu'il est trié.
- Considérons le problème de déterminer si tous les n éléments d'un tableau sont distincts
  - Si l'on tente de résoudre ce problème sur un tableau non trié, il faut alors comparer, en pire cas, toutes les n(n-1)/2 paires d'éléments
    - Cela nécessite $\Theta(n^2)$ comparaisons en pire cas
  - Si l'on trie d'abord le tableau, il suffit simplement d'examiner chaque paire A[i], A[i+1] d'éléments consécutifs
    - Cela nécessite alors $\Theta(n)$ comparaisons en pire cas
    - Mais le tri fusion nécessite $\Theta(n \log n)$ opérations
    - Le temps d'exécution total (trier + balayer) est donc $\Theta(n \log n)$ en pire cas (selon la règle du maximum)
  - Ce problème se résout donc plus rapidement si l'on tri d'abord le tableau.

Le « pré triage » (suite)

- Considérons le problème de recherche d'un élément K dans un tableau de n éléments
  - Nous pouvons effectuer une recherche séquentielle sur un tableau non trié. Cela nécessite $\Theta(n)$ comparaisons en pire cas
  - Nous pouvons trier d'abord le tableau et ensuite faire une recherche binaire sur le tableau trié
    - Le tri-fusion nécessite $\Theta(n \log n)$ opérations en pire cas
    - Le recherche binaire nécessite $\Theta(\log n)$ comparaisons en pire cas
    - Le temps total d'exécution (tri + recherche) en pire cas sera donc de $\Theta(n \log n)$ en vertu de la règle du maximum

- Ce problème se résout donc moins rapidement en triant d'abord

Le « pré triage » (suite)

- (… suite au problème de recherche d'un élément K dans un tableau de n éléments)

- Cependant, si le tableau n'est pas modifié pour les r prochaines recherches
  - La recherche séquentielle nécessite $\Theta(r\ n)$ comparaisons en pire cas
  - Mais le tri, nécessitant $\Theta(n \log n)$ opérations, est exécuté une seule fois
  - Et les r recherches binaires nécessitent $\Theta(r \log n)$ opérations.
  - Cela donne donc un temps d'exécution total de:

$$\Theta(\max\{n \log n, r \log n\}) \text{ en pire cas}$$

  - Cela se compare avantageusement à la recherche séquentielle lorsque $r \in \Omega(\log(n))$.

Les tas (monceaux)

- Un **tas** est une structure de données très utilisée pour la mise en œuvre des **files de priorité**
  - Ex: file d'attente de processus (avec priorités)
- Une file de priorité doit supporter efficacement les opérations suivantes:
  - Trouver un item avec la priorité la plus élevée
  - Enlever un item ayant la priorité la plus élevée
  - Ajouter un item à la file de priorité

- Le tas est une structure intermédiaire utilisée par l'algorithme du **tri par tas** (« heapsort »)
  - Un tableau est d'abord transformé en un tas
  - Ensuite les éléments sont repositionnés en ordre croissant
    - Le tableau est alors trié

Définition du tas

- Un arbre binaire dont chaque nœud contient une valeur (ou un élément) est appelé un tas lorsque ces deux propriétés sont satisfaites:
  - L'arbre binaire est essentiellement complet: tous les niveaux sont remplis excepté (possiblement) le dernier et, dans ce cas, les nœuds manquant sont tous à droite
  - La valeur d'un nœud est toujours supérieure où égale à celle de ses enfants (propriété du tas)
    - la valeur de la racine est donc la valeur maximale du tas

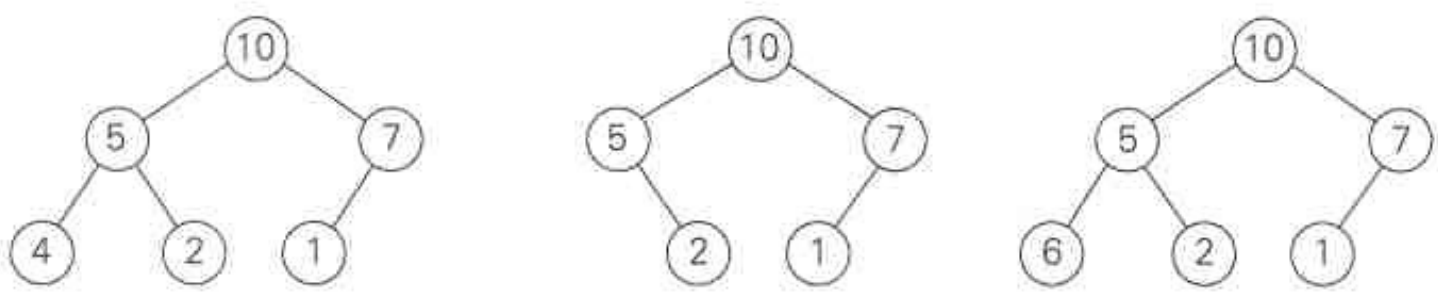
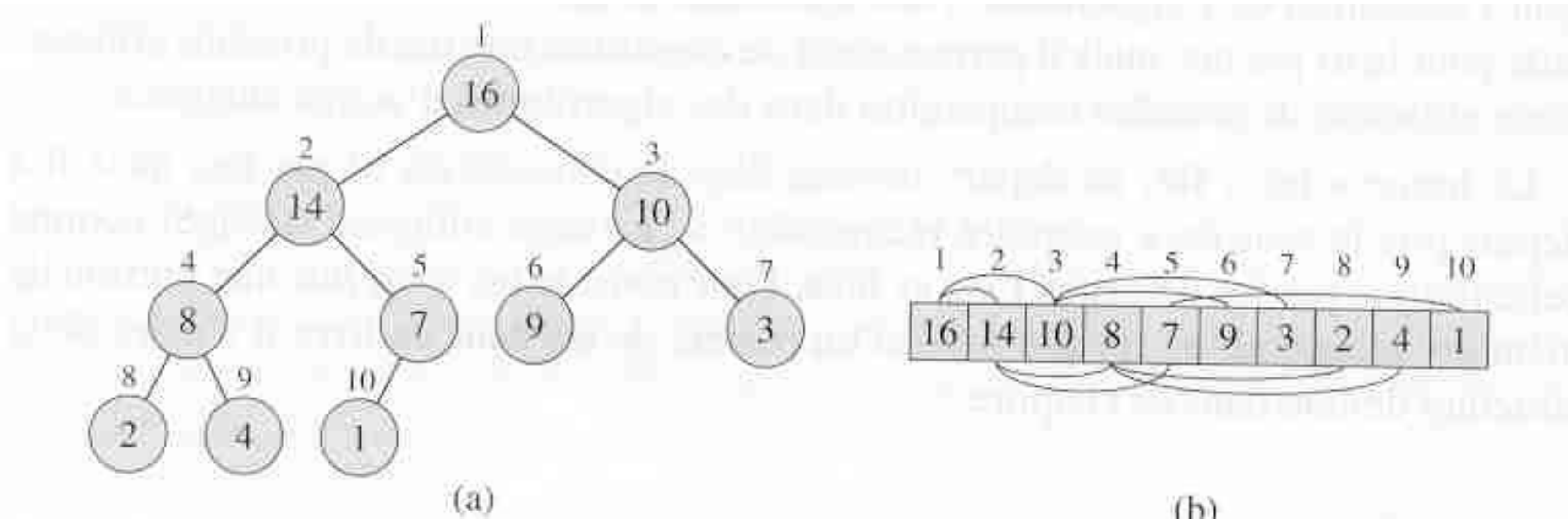


FIGURE 6.9 Illustration of the definition of "heap": only the leftmost tree is a heap.

Le tas et son tableau associé

- Les éléments (valeurs) du tas peuvent se positionner dans un tableau H[1..n] comme suit:
  - La valeur de la racine est placée en H[1]
  - Le premier élément du niveau suivant est placé en H[2]
  - Le second élément de ce niveau est placé en H[3] …
  - L'assignation se fait donc du niveau supérieur au niveau inférieur en balayant chaque niveau de gauche à droite



Propriétés des tas

- Un nœud n'ayant pas d'enfants est appelé une **feuille**
- Un nœud ayant au moins un enfant est appelé un **parent** (ou nœud interne)
- La **hauteur d'un arbre** est la longueur du chemin menant de la racine à la feuille la plus profonde
  - Ex: l'arbre de la page précédente possède une hauteur de 3
- **Considérons un tas de n nœuds et de hauteur h**
- Numérotons chaque niveau en débutant par le niveau supérieur
  - Le niveau 0 contient 2^0 nœud (la racine)
  - Le niveau 1 contient 2^1 nœuds
  - Le niveau i contient 2^i noeuds
- Seul le niveau h (niveau inférieur) est possiblement non plein
- Le nombre **l** de nœuds dans ce niveau inférieur est au minimum 1 et au maximum 2^h.

Propriétés des tas (suite)

- Nous avons alors:
  - $n = 2^0 + 2^1 + \ldots 2^{h-1} + l$ avec $1 \leq l \leq 2^h$
  - Alors: $\mathbf{n = 2^h - 1 + l}$
- Alors:
  - $n \geq 2^h \Rightarrow h \leq \log_2(n)$
- De plus:
  - $n \leq 2^h - 1 + 2^h = 2^{h+1} - 1 \Rightarrow n < 2^{h+1} \Rightarrow \log_2(n) < h + 1$
- Alors: $h \leq \log_2(n) < h + 1 \Rightarrow \lfloor \log_2(n) \rfloor = h$

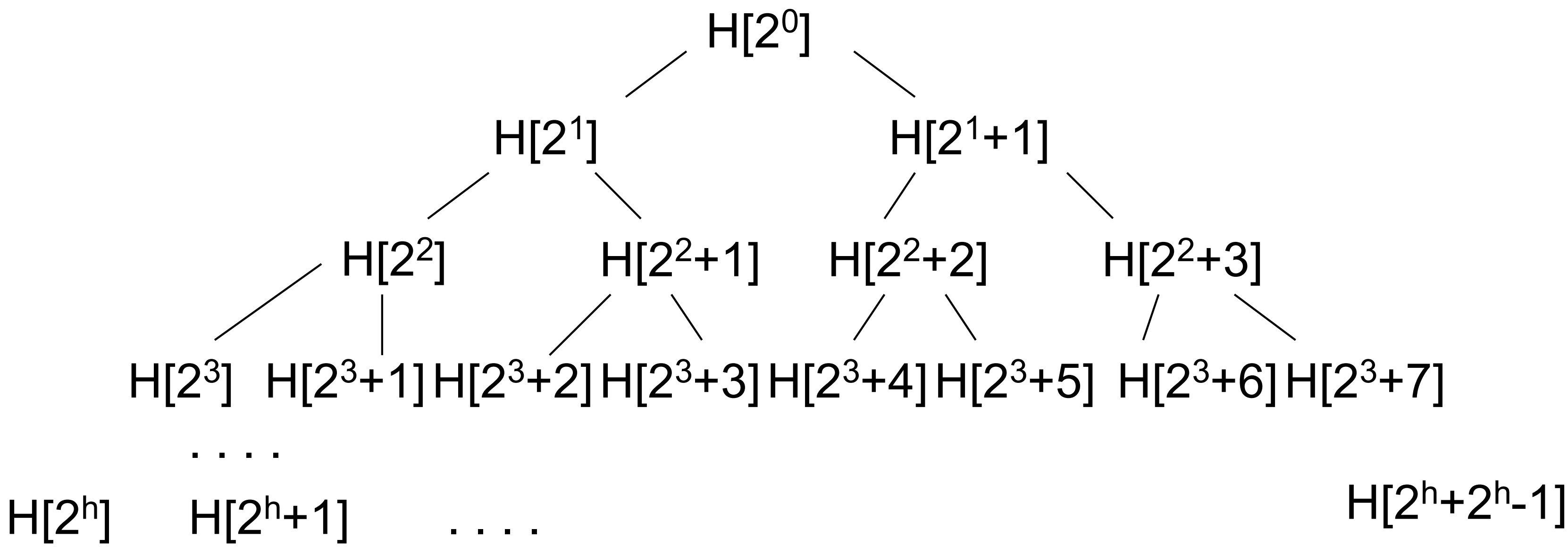
- **La hauteur h d'un tas de n nœuds est alors donnée par:**

$$\mathbf{h = \lfloor \log_2(n) \rfloor}$$

Propriétés des tas (suite)

- Déterminons le nombre de feuilles **f** contenues dans un tas de **n** noeuds.
  - Soit l = nombre de nœuds présents dans le niveau (inférieur) **h**
    - Rappel: $1 \leq l \leq 2^h$
  - Le nombre de nœuds présent dans le niveau h-1 est de 2^{h-1}
  - Si $l = 1 \Rightarrow f = 2^{h-1} + 0$
  - Si $l = 2 \Rightarrow f = 2^{h-1} + 1$
  - Si $l = 3 \Rightarrow f = 2^{h-1} + 1$
  - Si $l = 4 \Rightarrow f = 2^{h-1} + 2$
  - Si $l = i \Rightarrow f = 2^{h-1} + \lfloor i/2 \rfloor$
  - Alors: $f = 2^{h-1} + \lfloor \mathbf{l/2} \rfloor$
  - Or: $l = n + 1 - 2^h \Rightarrow \lfloor \mathbf{l/2} \rfloor = \lfloor (n+1)/2 - 2^{h-1} \rfloor = \lfloor (n+1)/2 \rfloor - 2^{h-1}$
  - Alors: $f = \lfloor (n+1)/2 \rfloor = \lceil \mathbf{n/2} \rceil \mathbf{= f}$
- Puisqu'un nœud est soit une feuille ou un parent, le nombre **p** de parents contenus dans un tas de **n** nœuds est $\mathbf{p = \lfloor n/2 \rfloor}$.

Propriétés des tas (suite)



- Les éléments du tas sont disposés dans H[1..n] comme ci-dessus
- Nous remarquons que les enfants de $H[2^k]$ sont $H[2^{k+1}]$ et $H[2^{k+1}+1]$ (lorsqu'ils existent)
- Et les enfants de $H[2^k+j]$ sont $H[2^{k+1}+2j]$ et $H[2^{k+1}+2j+1]$ (s'ils existent)
- **Les enfants de H[i] sont alors H[2i] et H[2i+1] (lorsqu'ils existent)**
- **Le parent de H[i] est alors $H[\lfloor i/2 \rfloor]$**

Construction d'un tas du bas vers le haut

- Pour que le tableau H[1..n] initial soit un tas il doit satisfaire H[i] ≥ H[2i] et H[i] ≥ H[2i+1] pour i = 1...**$\lfloor n/2 \rfloor$** (les nœuds parents).
- Nous devons permuter certains éléments pour obtenir cette propriété
- L'algorithme de construction débute avec le parent i = $\lfloor n/2 \rfloor$.
- Si H[i] < max{H[2i], H[2i+1]}, on «swap» H[i] avec max{H[2i], H[2i+1]}
- On recommence avec le parent i-1 jusqu'à la racine (i=1)
- Lorsque l'on interchange H[i] avec l'un de ses enfants H[j] il faut recommencer cette procédure avec cet enfant H[j] (et non avec l'autre)

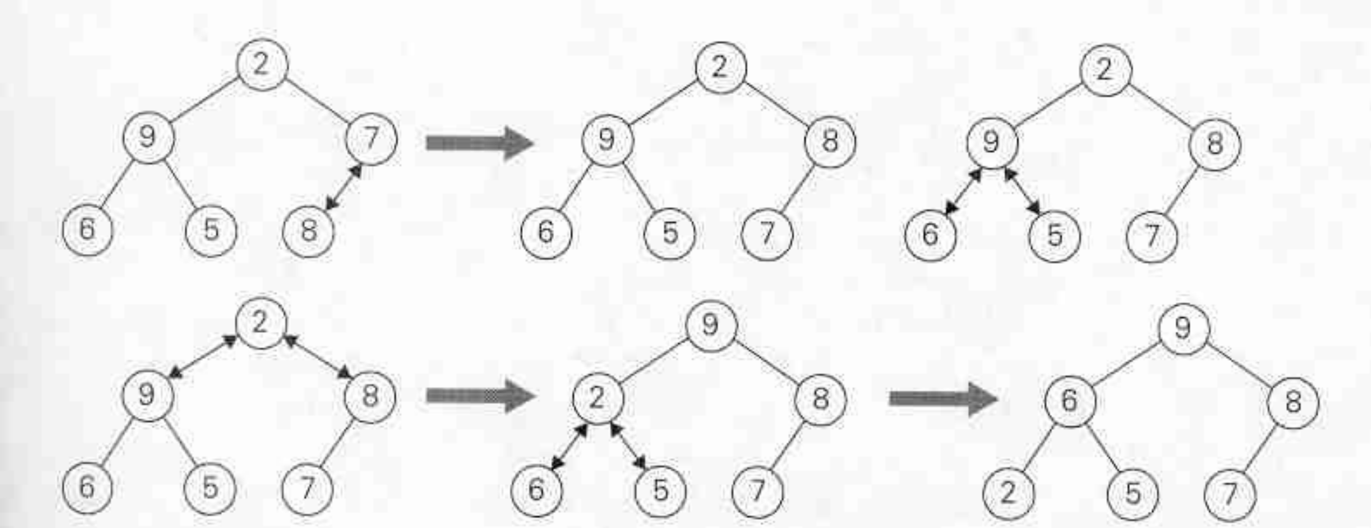


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

L'algorithme HeapBottomUp pour la construction d'un tas

Procédure `HeapBottomUp` $(H[1..n])$

// Construit un monceau à partir des éléments du tableau
// Input : Un tableau $H[1..n]$ d'éléments pouvant être comparés
// Output : Un monceau $H[1..n]$
pour $i = \lfloor n/2 \rfloor$ *en descendant jusqu'à 1* **faire**
    `Tamiser`$(H[1..n], i)$

L'algorithme HeapBottomUp pour la construction d'un tas

Procédure `Tamiser` ($H[1..n]$, i)

// Tamise l'élément $H[i]$ dans le monceau jusqu'à ce qu'il soit plus grand
// que ses deux enfants ou qu'il devienne un enfant.
// Input : Un tableau $H[1..n]$ d'éléments pouvant être comparés et
// un index $1 \leq i \leq n$
// Output : Le tableau modifié
$k \leftarrow i$
$v \leftarrow H[k]$
monceau $\leftarrow$ faux
tant que $\neg$monceau $\wedge\ 2 \times k \leq n$ **faire**
    $j \leftarrow 2 \times k$
    // S'il y a deux enfants
    si $j < n$ **alors**
        si $H[j] < H[j+1]$ **alors** $j \leftarrow j + 1$
    si $v \geq H[j]$ **alors**
        monceau $\leftarrow$ vrai
    sinon
        $H[k] \leftarrow H[j]$
        $k \leftarrow j$
$H[k] \leftarrow v$

Analyse de HeapBottomUp

- Comptons le nombre de comparaisons effectuées par HeapBottomUp pour estimer son temps d'exécution
- Pour chaque parent i de $\lfloor n/2 \rfloor$ … 1, il faut comparer H[i] avec ses deux enfants H[2i] et H[2i+2]
  - (le parent i = $\lfloor n/2 \rfloor$ possède parfois un seul enfant)
- Alors disons que nous effectuons 2 comparaisons pour chaque parent
- En meilleur cas, H[1..n] est déjà (initialement) un tas et satisfait H[i] ≥ H[2i] et H[i] ≥ H[2i+1] pour i = 1… $\lfloor n/2 \rfloor$
  - Pour ces cas, chaque parent occasionnera uniquement 2 comparaisons et il ne sera jamais nécessaire de vérifier la propriété du tas pour les enfants
  - On a alors $C_{best}(n) = 2 \lfloor n/2 \rfloor$.
- En pire cas, il y a toujours un des enfants H[j] de H[i] qui a H[j] > H[i]
  - Cela se produit lorsque H[i] < H[2i] pour i = 1… $\lfloor n/2 \rfloor$.
  - Ex: lorsqu'un tableau d'éléments tous distincts est déjà trié

Analyse de HeapBottomUp (suite)

- Dans ces pire cas, pour chaque parent i, il faut faire 2 comparaisons par niveau qu'il y a sous le parent i.
  - Si le nœud i se trouve au niveau k, le nombre de niveaux qu'il y a sous le noeud i est donné par h – k (où h est la hauteur du tas)
  - Le nombre de comparaisons qu'il faut faire en pire cas pour le parent i est alors de 2(h – k)
  - Soit p(k) le nombre de parents présents au niveau k
  - Nous avons $p(k) = 2^k$ pour k = 0, 1, ... h – 2
  - Pour k = h – 1, nous avons $p(k) \leq 2^k$ (voir figures pages 8 et 12)
  - Le nombre de comparaisons $C_{worst}(n)$ effectuées au total en pire cas est donc donné par:

$$
\begin{aligned}
C_{worst}(n) &= 2\sum_{k=0}^{h-1} p(k)\cdot(h-k) \;\leq\; 2\sum_{k=0}^{h-1} 2^k(h-k) \\
&= 2h\cdot(2^h-1) - 2\sum_{k=0}^{h-1} k2^k
\end{aligned}
$$

Analyse de HeapBottomUp (suite)

- Or en annexe A on trouve:

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

- Alors:

$$\sum_{k=0}^{h-1} k2^k = \sum_{k=1}^{h-1} k2^k = (h-2)2^h + 2$$

- Ainsi:

$$\begin{aligned} C_{worst}(n) &\leq 2h \cdot (2^h - 1) - 2\sum_{k=0}^{h-1} k2^k \\ &= 2h \cdot (2^h - 1) - 2\left[(h-2)2^h + 2\right] \\ &= 4(2^h - \frac{h}{2} - 1) \end{aligned}$$

Analyse de HeapBottomUp (suite)

- Or, en page 10, nous avions $2^h - 1 = n - l < n$ et $h = \lfloor \log_2(n) \rfloor$
- Alors:
  - $C_{worst}(n) \leq 4(2^h - h/2 - 1) < 4n - 2h = 4n - 2\lfloor \log_2(n) \rfloor \in \Theta(n)$
- Alors $C_{worst}(n) \in O(n)$
- Or nous avions $C_{best}(n) \in \Theta(n)$
- **Le temps requis pour construire un tas de n noeuds avec l'algorithme HeapBottomUp est donc $\in \Theta(n)$**

Insertion d'un élément dans un tas

- Pour la mise en œuvre d'une file de priorité, nous devons pouvoir insérer rapidement un nouvel item dans un tas
- Pour cela, nous insérons d'abord le nouvel élément K dans une nouvelle feuille que nous positionnons juste après la dernière feuille du tas (ou, plus simplement, nous faisons $H[n+1] \leftarrow K$)
- Nous comparons K avec son parent $H[\lfloor (n+1)/2 \rfloor]$:
  - Si $K \leq H[\lfloor (n+1)/2 \rfloor]$ ne rien faire car $H[1..n+1]$ est un tas
  - Sinon on interchange K avec $H[\lfloor (n+1)/2 \rfloor]$
  - Nous recommençons jusqu'à ce que K est ≤ à la valeur de son parent (ou jusqu'à ce que K devienne la racine)
- Le nombre maximal de comparaisons requises est donc de $h' = \lfloor \log_2(n+1) \rfloor$

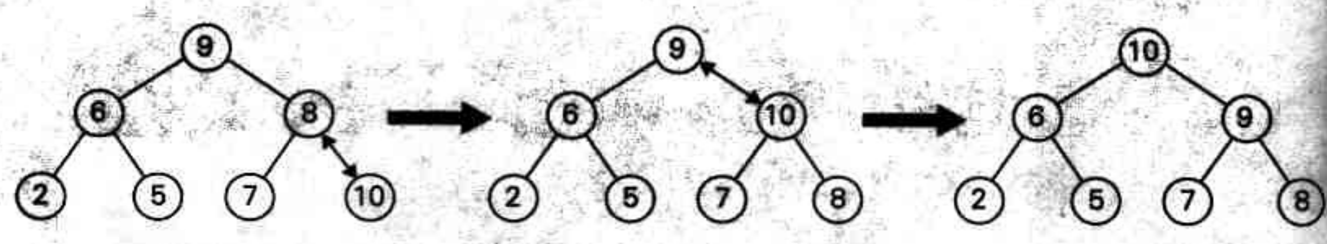


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Insertion d'un élément dans un tas

Procédure `AjouteElement` ($H[1..n], v$)

// Ajoute l'élément v au monceau
// Input : Un monceau $H[1..n]$ et une valeur v
// Output : Un monceau $H[1..n+1]$ contenant l'élément v
$H[n+1] \leftarrow v$ // Ajout d'un élément à la fin du tableau
`Percoler`($H[1..n+1], n+1$)

Procédure `Percoler` ($H[1..n], i$)

// Fait percoler l'élément $H[i]$ dans le monceau H jusqu'à ce qu'il soit plus
// petit que son parent ou qu'il soit la racine
// Input : Un tableau $H[1..n]$ d'éléments pouvant être comparés et un
// index $1 \leq i \leq n$
// Output : Le tableau H modifié
$v \leftarrow H[i]$
tant que $i > 1 \wedge H[\lfloor i/2 \rfloor] < v$ **faire**
  $H[i] \leftarrow H[\lfloor i/2 \rfloor]$
  $i \leftarrow \lfloor i/2 \rfloor$
$H[i] \leftarrow v$

Enlever la racine d'un tas

- Pour la mise en œuvre d'une file de priorité, nous devons fréquemment enlever la racine d'un tas car c'est un élément dont la priorité est la plus élevée
- Pour cela, nous interchangeons l'élément H[n] avec H[1] et nous reconstruisons le tas H[1..n-1] comme suit:
  - nous comparons la valeur de H[1] avec celle de ses enfants et l'interchangeons avec le max de ses enfants si c'est nécessaire
  - Nous continuons jusqu'au niveau inférieur (si c'est nécessaire) tel que prescrit par HeapBottomUp pour l'élément i = 1.
- Cela nécessite au plus $2\lfloor \log_2(n-1) \rfloor$ comparaisons (pour reconstruire H[1..n-1])

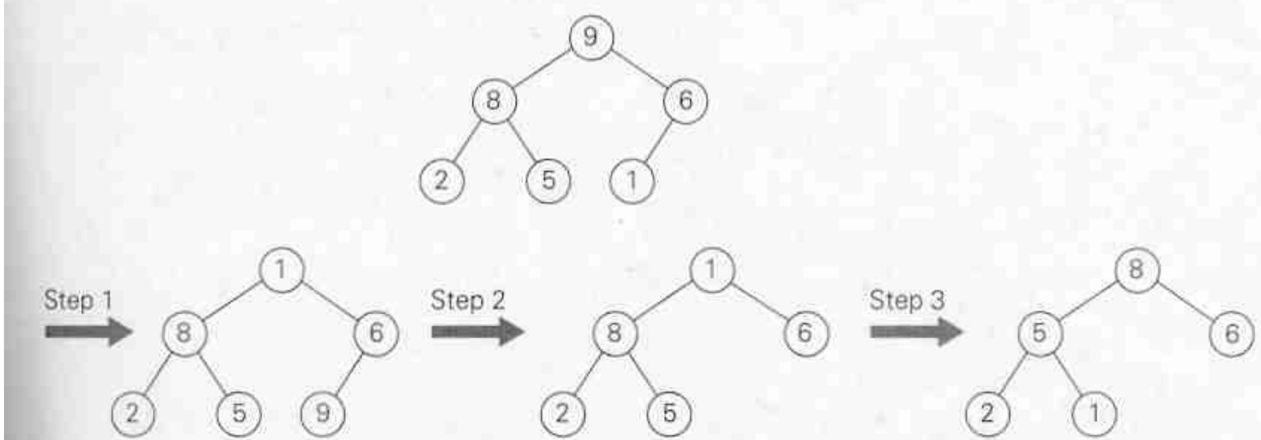


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key, after which the smaller tree is "heapified" by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

Enlever la racine d'un tas

Procédure `ExtraisLaRacine` $(H[1..n])$

// Retire la racine du monceau et retourne sa valeur
// Input : Un monceau $H[1..n]$
// Output : Le monceau modifié $H[1..n-1]$ et la valeur de la racine
$r \leftarrow H[1]$
$H[1] \leftarrow H[n]$
`Tamiser`$(H[1..n-1], 1)$
retourner r

Modifier la valeur d'un élément

- Il est possible de changer la valeur d'un élément dans un monceau.
- Par exemple, dans une file de priorité, on pourrait vouloir changer la priorité d'une tâche.
- Il faut alors tamiser ou percoler l'élément modifié selon s'il a été augmenté ou diminué.

Procédure RemplaceValeur($H[1..n]$, i, v)

// Remplace la valeur $H[i]$ par la valeur v
// Input : Un monceau $H[1..n]$
// Output : Le monceau $H[1..n]$ modifié
$w \leftarrow H[i]$
$H[i] \leftarrow v$
si $w > v$ **alors**
    Tamiser($H[1..n], i$)
sinon
    Percoler($H[1..n], i$)

Le tri par tas

- À partir d'un tableau non trié, nous construisons d'abord un tas H[1..n] à l'aide de l'algorithme HeapBottomUp en un temps $\in \Theta(n)$
  - La racine est donc un élément de valeur la plus élevée
- Nous interchangeons H[1] avec H[n] et reconstruisons le tas H[1..n-1] à l'aide de l'algorithme d'extraction de la racine.
  - Cela nécessite au plus $2\lfloor \log_2(n-1) \rfloor$ comparaisons
- Nous recommençons cette opération avec H[1..n-1], ensuite H[1..n-2], et puis H[1..n-3] … finalement l'on s'arrête en H[1].
  - H[1..n] est alors trié
- Le pseudo code de l'algorithme se trouve à la page suivante
- Le nombre C(n) de comparaisons requises est alors borné par:

$$C(n) \leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2(2) \rfloor$$

Analyse du tri par tas

- Alors le nombre de comparaisons est borné par:

$$\begin{aligned} C(n) &\leq 2\sum_{i=2}^{n-1} \lfloor \log_2(i) \rfloor \leq 2\sum_{i=2}^{n-1} \log_2(n-1) \\ &= 2(n-2)\log_2(n-1) \Rightarrow C(n) \in O(n \log n) \end{aligned}$$

- Nous venons donc de démontrer que C(n) ∈ O(n log n)
- Alors $C_{worst}(n) \in O(n \log n)$

- En fait, il est possible de démontrer (voir chap 10) que tous les algorithmes de tri par comparaison doivent avoir $C_{worst}(n) \in \Omega(n \log n)$.

- Alors, pour le tri par tas, nous avons: $\mathbf{C_{worst}(n) \in \Theta(n \log n)}$.

Analyse du tri par tas (suite)

- Le meilleur cas est celui où tous les éléments du tableau sont identiques. Dans ce cas HeapBottomUp produit un tas qui est à la fois un tableau trié et HeapSort effectue au plus deux comparaisons (une par enfant) pour chaque valeur de m de la boucle **for**. Alors:

$$C_{best}(n) \in \Theta(n)$$

- De plus, certains ont démontré que $C_{avg}(n) \in \Theta(n \log n)$ mais la démonstration est difficile…

- Empiriquement nous observons, qu'en moyenne, le tri par tas est légèrement plus rapide que le tri fusion mais légèrement moins rapide que le tri rapide.

Lecture (Levitin)

- Chapitre 6 Transform-and-Conquer
  - 6.1 Presorting
  - 6.4 Heaps and Heapsort