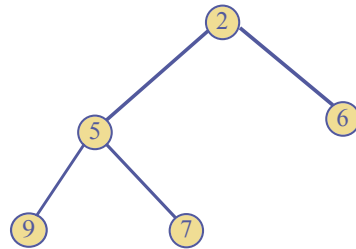


Monceaux (§9.2)



Rappel: TAD: Files avec priorité (§9.1)

- Une file avec priorités garde en mémoire une collection d'entrées (items)
- Chaque **item** est une paire (clé, élément)
- Opérations principales:
 - **insérer(k,x)**: insère un entrée ayant la clé k et l'élément x
 - **enleverMin()**: enlève et retourne une entrée de clé minimale
- Opérations additionnels:
 - **minElement()**: retourne, sans enlever, un élément de clé minimale
 - **minCle()**: retourne la clé minimale
 - **taille()**, **estVide()**

Rappel: Trier en utilisant une file avec priorités

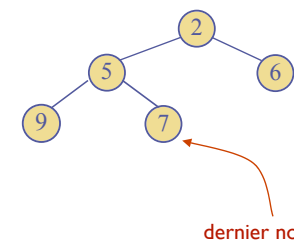
- On peut utiliser une file avec priorités pour trier un ensemble d'éléments comparables
 - 1) Insérer les éléments un par un dans la file avec une série d'opérations **insérer(e,e)**
 - 2) Enlever les éléments dans l'ordre croissant avec une série d'opérations **enleverMin()**
- Le temps d'exécution dépend de l'implémentation de la file avec priorités:
 - séquences non ordonnées:
Tri-Sélection: complexité $O(n^2)$
 - séquences ordonnées:
Tri-Insertion: complexité $O(n^2)$

Algorithme TriFP(S, C)
Entrée séquence S , comparateur C pour les éléments de S
Sortie séquence S triée en ordre croissant selon C
 $P \leftarrow$ file avec priorité utilisant le comparateur C
while $\neg S.estVide()$
 $e \leftarrow S.enlevePremier()$
 $P.insérer(e, e)$
while $\neg P.estVide()$
 $e \leftarrow P.enleverMin()$
 $S.insérerFin(e)$

➔ Peut-on faire mieux ?

Monceaux (§9.2)

- Un monceau est un arbre binaire gardant des clés en mémoire dans ses noeuds et satisfaisant les propriétés suivantes:
 - **Ordre de monceau**: pour chacun des noeuds internes v différents de la racine, on a $clé(v) \leq clé(fil(v))$
 - **Arbre binaire complet**: soit h la hauteur du monceau, alors
 - Pour $i = 0, \dots, h-1$, il y a 2^i noeuds de profondeur i
 - Au niveau (profondeur) $h-1$, les noeuds internes sont situés à la gauche des noeuds externes
- Le dernier noeud d'un monceau est le noeud le plus à droite du niveau h



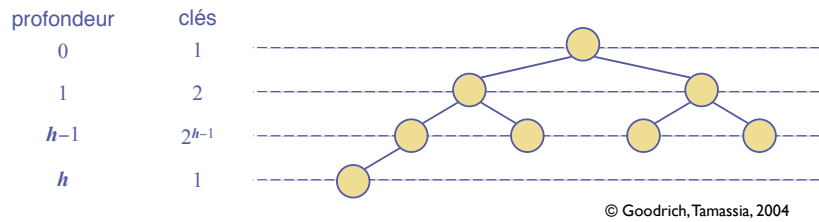
Hauteur d'un monceau

- **Théorème:** Un monceau gardant en mémoire n clés à une hauteur $O(\log n)$

Preuve: On utilise le fait qu'un monceau est un arbre binaire complet:

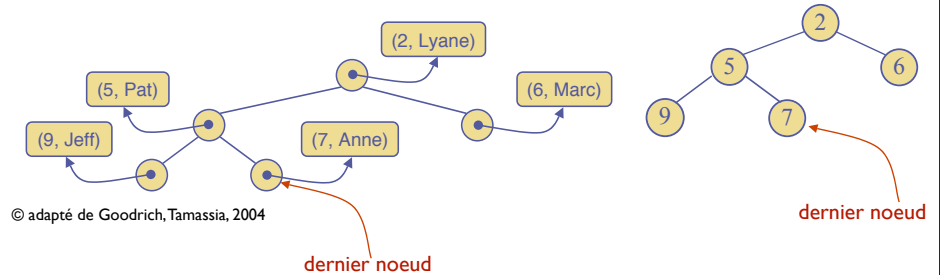
- Soit h la hauteur du monceau gardant en mémoire n clés
- Comme on a 2^i clés de profondeur i , pour $i = 0, \dots, h-1$ et au moins 1 clé de profondeur h , on a

$$n \geq (1 + 2 + 2^2 + \dots + 2^{h-1}) + 1 = \left(\sum_{i=0}^{h-1} 2^i\right) + 1$$
- On a donc $n \geq 2^h$, i.e $h \leq \log n$



Monceaux et files avec priorités

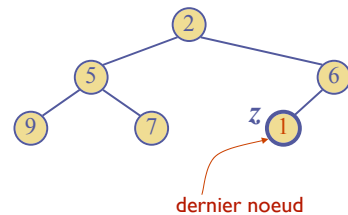
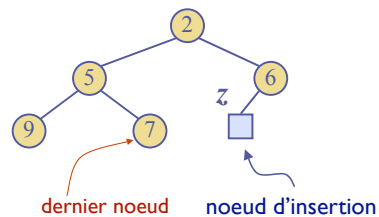
- On peut utiliser un monceau pour implémenter une file avec priorités
- On garde en mémoire dans les noeuds de l'arbre les items (clé, élément)
- On garde en mémoire la position du dernier noeud
- Pour simplifier, on ne montre que la partie clé des entrées dans les dessins



Insertion d'éléments dans un monceau

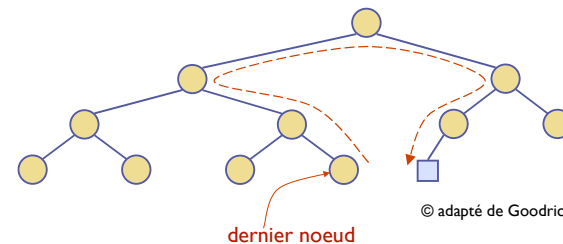
- L'opération **insérer(k,x)** du TAD files avec priorités correspond à l'insertion d'un item **(k,x)** dans le monceau
- L'algorithme d'insertion consiste en trois étapes:

- 1) Trouver le noeud d'insertion z (qui deviendra le nouveau dernier noeud)
- 2) Insérer **(k,x)** dans le noeud z
- 3) Restorer l'ordre dans le monceau



Trouver le noeud d'insertion

- On peut trouver le noeud d'insertion en suivant un chemin de longueur $O(\log n)$, où ici, longueur = nombre de noeuds visités
 - À partir du dernier noeud inséré, monter jusqu'à ce qu'un fils gauche soit atteint (ou jusqu'à la racine) **Attention! Si on est déjà dans un fils gauche alors, il faut y rester.**
 - Si un fils gauche est atteint, alors parcourir l'arbre jusqu'à son frère droit
 - Si ce noeud est libre, c'est le noeud d'insertion
 - Sinon, si le fils gauche ou droit de ce noeud est libre, c'est le noeud d'insertion
 - Sinon, descendre vers la gauche, jusqu'à ce que vous trouviez une feuille. Le noeud d'insertion est le fils gauche de cette feuille.



Restorer l'ordre vers le haut

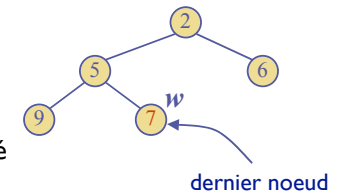
- Après l'insertion d'une nouvelle clé k , il est possible que l'ordre du monceau soit perturbé
- L'algorithme **upheap** rétablit l'ordre du monceau en faisant monter la clé k , le long du chemin ascendant partant du noeud d'insertion et allant jusqu'à la racine, avec des opérations d'échange de clés (swap)
- L'algorithme **upheap** se termine quand la clé k atteint la racine ou un noeud dont le parent a une clé plus petite ou égale à k
- Comme la hauteur du monceau est $O(\log n)$, la complexité en temps de **upheap** est aussi $O(\log n)$



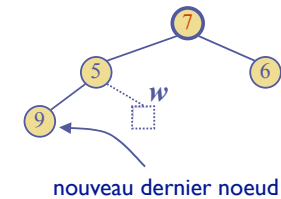
© Goodrich, Tamassia, 2004

Suppression d'éléments dans un monceau

- L'opération **enleverMin()** du TAD files avec priorités correspond à la suppression de la racine d'un monceau
- L'algorithme de suppression est constitué de trois étapes:
 - Remplacer la clé de la racine par la clé du dernier noeud
 - Enlever le dernier noeud
 - Restorer l'ordre dans le monceau



© adapté de Goodrich, Tamassia, 2004



Restorer l'ordre vers le bas

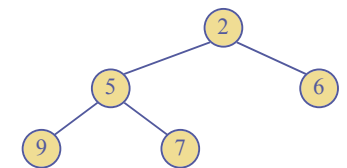
- Après le remplacement de la clé de la racine par la clé du dernier noeud, il est possible que l'ordre du monceau soit perturbé
- L'algorithme **downheap** rétablit l'ordre du monceau en faisant descendre la clé k , le long du chemin descendant partant de la racine, avec des opérations d'échange de clés (swap)
- L'algorithme **downheap** se termine quand la clé k atteint une feuille ou un noeud dont les fils ont des clés plus grande ou égale à k
- Comme la hauteur du monceau est $O(\log n)$, la complexité en temps de **downheap** est aussi $O(\log n)$



© Goodrich, Tamassia, 2004

Implémentation d'un monceau

- On peut représenter un monceau contenant n clés par une liste de longueur $n+1$
- Pour le noeud au rang i de la liste:
 - son fils gauche est au rang $2i$
 - son fils droit est au rang $2i + 1$
- La cellule de rang 0 n'est pas utilisée
- L'opération **insérer(k,e)** correspond à insérer au rang $n+1$
- L'opération **enleverMin** retourne l'élément au rang 1



© Goodrich, Tamassia, 2004

Tri-Monceau

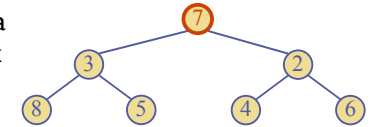
- Supposons qu'on a une liste avec priorités, contenant n éléments, implémentée à l'aide d'un monceau. Alors,
 - la complexité en espace est de $O(n)$
 - les opérations `insérer(k,e)` et `enleverMin()` ont une complexité en temps de $O(\log n)$
 - les opérations `taille()` et `estVide()` ont une complexité en temps de $O(1)$
- La complexité en temps du tri de n éléments, avec cette implémentation est de $O(n \log n)$
- On appelle l'algorithme de tri avec monceau, le Tri-Monceau

Fusionner deux monceaux

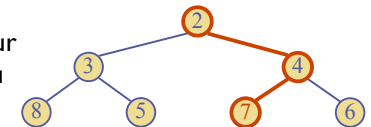
- On nous donne 2 monceaux et une clé k



- On crée un nouveau monceau avec la clé k en racine et les deux monceaux comme sous-arbres



- On utilise l'algorithme `downheap` pour mettre en ordre le nouveau monceau



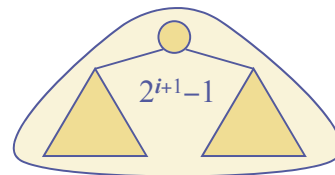
© adapté de Goodrich, Tamassia, 2004

Construire un monceau de bas en haut

- On peut construire un monceau contenant n clés en utilisant une méthode de construction allant du bas vers le haut et comportant $\log(n)$ étapes

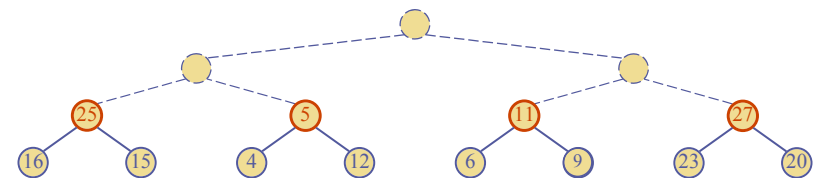
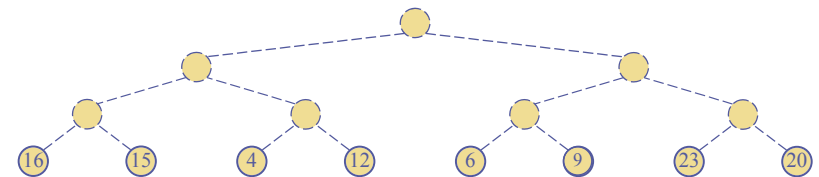


- À l'étape i , des paires de monceaux ayant $2^i - 1$ clés sont fusionnés en monceaux ayant $2^{i+1} - 1$ clés



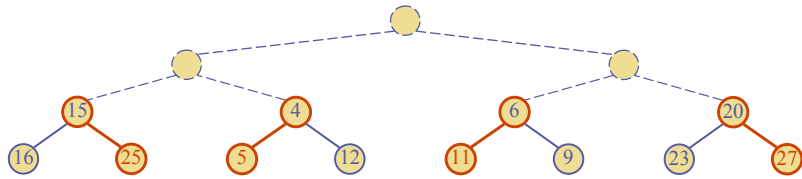
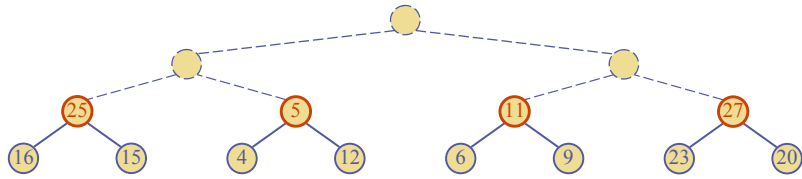
© Goodrich, Tamassia, 2004

Exemple:



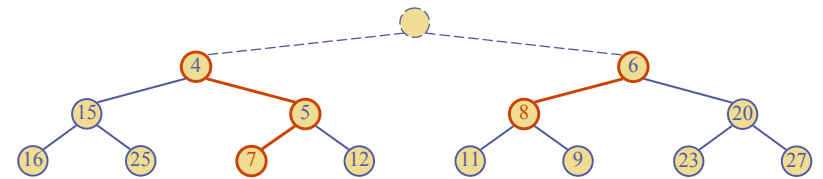
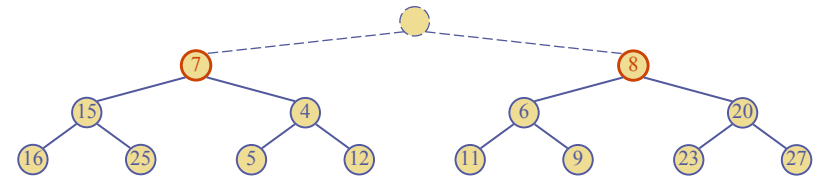
© adapté de Goodrich, Tamassia, 2004

Exemple: (suite)



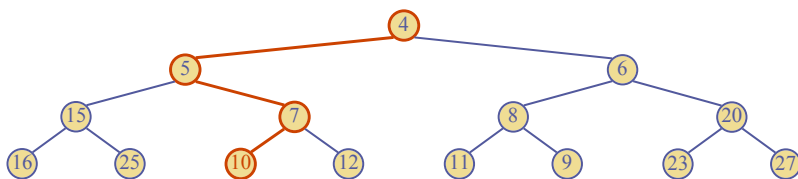
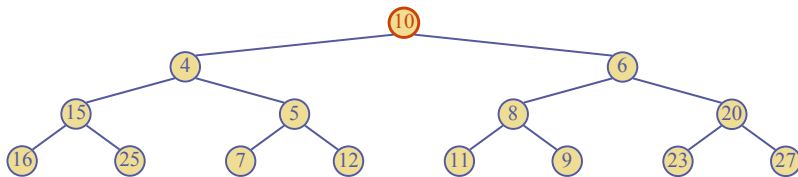
© adapté de Goodrich, Tamassia, 2004

Exemple: (suite)



© adapté de Goodrich, Tamassia, 2004

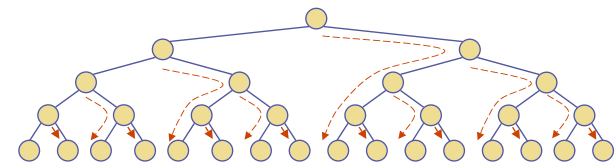
Exemple: (suite)



© adapté de Goodrich, Tamassia, 2004

Analyse de la complexité en temps

- Dans le pire des cas, à chaque insertion d'un nouveau noeud, on doit pour remettre de l'ordre, le faire descendre jusqu'aux feuilles (voir dessin).
- L'algorithme de construction de bas en haut à une complexité en temps de $O(n)$



© Goodrich, Tamassia, 2004