1. Tas (ou monceau ou “heap”)
2. Hauteur d’un tas
3. Tas et file d’attente avec priorités
4. `Insérer` dans un tas (vu en IFT1025)
5. `Retirer` dans un tas (vu en IFT1025)
6. Trier par tas (“heapsort”)
7. Implémenter un tas dans un tableau
8. Construire efficacement un tas initial

Tas / Monceau
(Heap)

Les deux stratégies pour implémenter une file d'attente avec priorités soulignent un compromis intéressant. Lorsqu'on utilise une **liste non triée**, nous pouvons effectuer des insertions en temps dans $O(1)$, mais trouver ou supprimer l'entrée avec la clé minimum nécessite une boucle qui prend un temps dans $O(n)$.

En revanche, si on utilise une **liste triée**, nous pouvons trivialement trouver ou supprimer l'entrée avec la clé minimum en temps dans $O(1)$, mais l'ajout d'un nouvel élément à la file d'attente nous coûte un temps dans $O(n)$.

Dans cette section, nous voyons une structure **plus efficace** pour implémenter une file d'attente avec priorités, soit un **tas binaire**, appelé aussi **monceau** ("**heap**").

Cette structure de données va nous permettre d'effectuer à la fois des insertions et des suppressions en **temps logarithmique**, représentant une amélioration significative comparativement aux implémentations basées sur des listes.

Le fondement pour réaliser cette amélioration provient de l'utilisation d'une structure d'**arbre binaire** qui permet de garder les éléments de manière **ni parfaitement ni complètement triés**.

Tas-min (Tas par défaut pour ce cours)

Un **tas** est un arbre binaire stockant des clés sur ses nœuds et satisfaisant les propriétés suivantes :

Contrainte ou propriété relationnelle (ou d'ordre) :

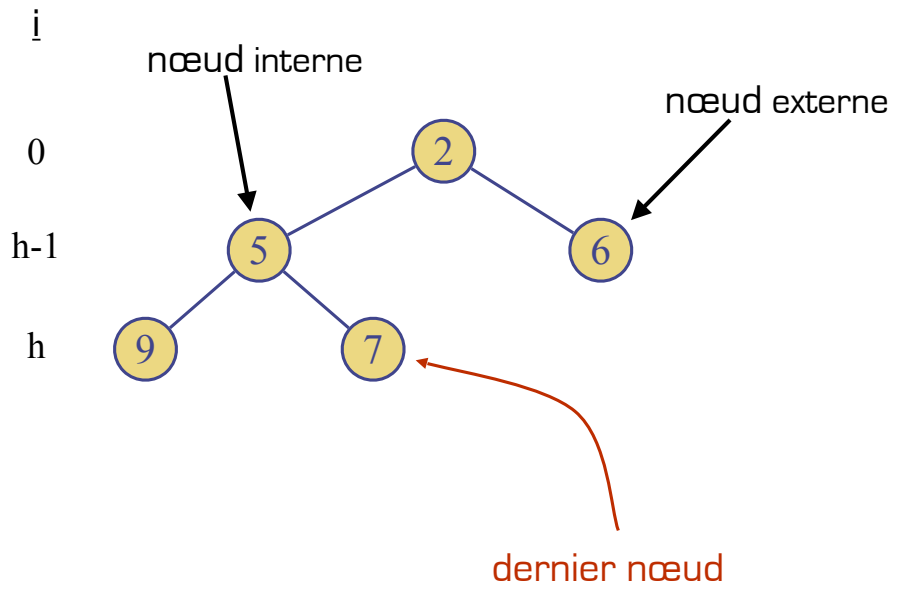
Ordre : pour chaque nœud v autre que la racine, clé(v) >= clé(parent(v))

Contrainte ou propriété structurale :

Arbre binaire complet : soit h la hauteur du tas

- pour $i = 0, \ldots, h - 1$, il y a $2i$ nœuds de profondeur i
- à la profondeur h, les nœuds sont à gauche des nœuds externes à la profondeur h-1.

Le dernier nœud d'un tas est le noeud le plus à droite à la profondeur h.



Hauteur d'un tas

Théorème : Un tas stockant n noeuds possède une hauteur dans $O(log\ n)$

Preuve : (nous appliquons la propriété d'un arbre binaire complet)

Soit h la hauteur d'un tas stockant n noeuds

Comme il y a $2i$ noeuds en profondeur $i = 0, ..., h - 1$ et entre 1 et $2h$ noeuds en profondeur h, on a entre

$1 + 2 + 4 + ... + 2h\text{-}1 + 1$ (minimum) et

$1 + 2 + 4 + ... + 2h\text{-}1 + 2h$ (maximum)

noeuds pour un arbre de hauteur h.

Comme $1 + 2 + 4 + ... + 2h\text{-}1 = 2h - 1$, on a entre

$2h - 1 + 1$ (minimum) et

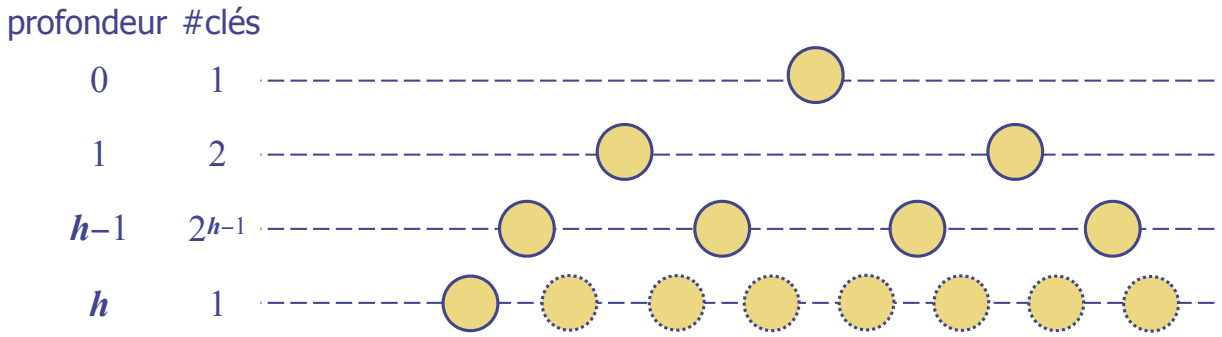
$2h - 1 + 2h$ (maximum) noeuds, soit

$n >= 2h$ et $n <= 2h+1 - 1$.

On prend le log de chaque côté :

$log\ n >= h$ et $log(\ n+1\ ) - 1 <= h$ (exemple : pour $h = 3$, on aurait entre 8 et 15 noeuds)

Puisque h est un entier, les 2 inégalités implique que $h = \lceil\ log\ n\ \rceil$

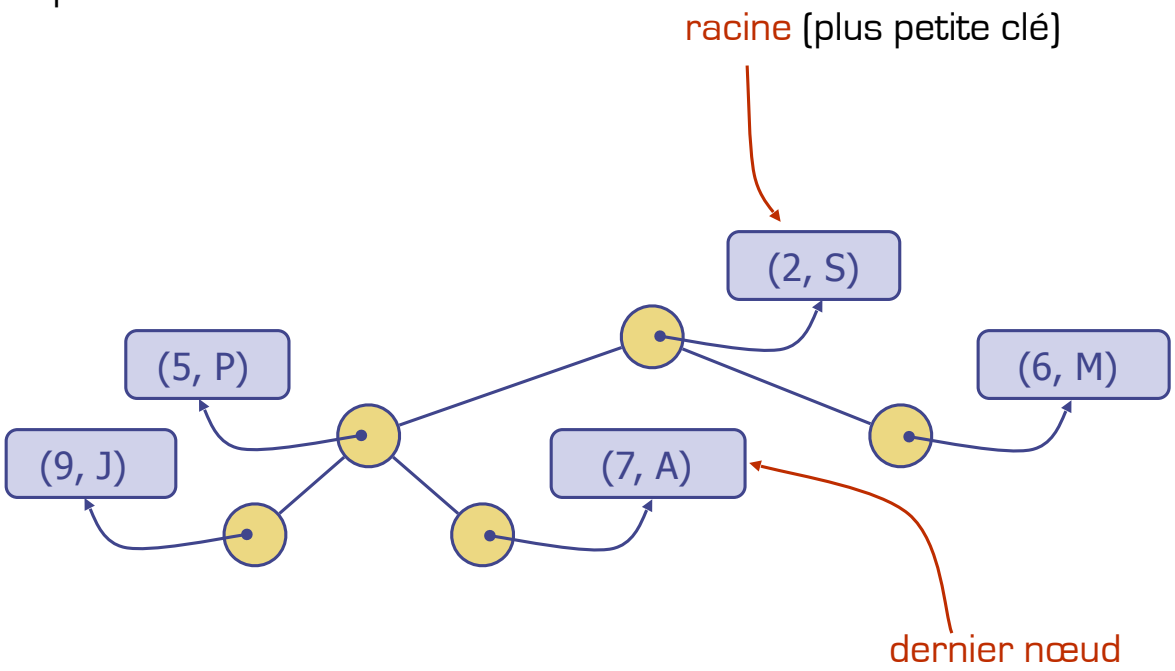


Tas et file d'attente avec priorités

Nous pouvons utiliser un tas pour implémenter une file d'attente avec priorités

Nous stockons une entrée `(clé, valeur)` dans chaque nœud interne

Nous gardons une trace sur la position du dernier noeud

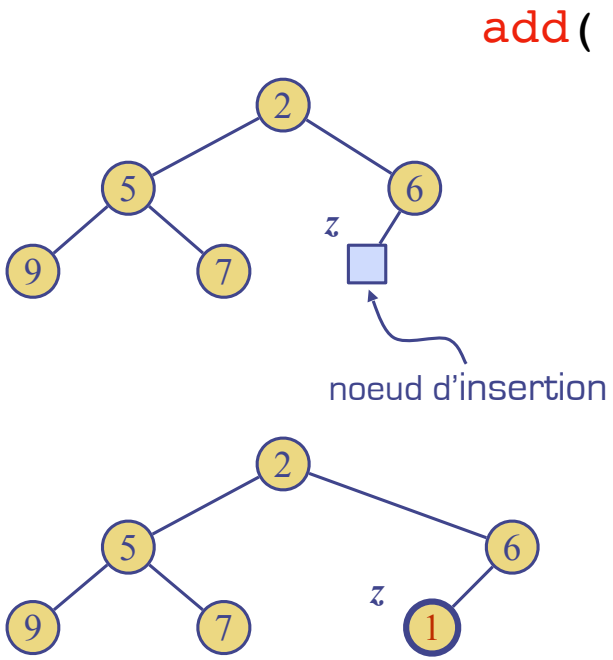


Insertion dans un tas

La méthode **`insert`** de l'ADT d'une file d'attente prioritaire correspond à l'insertion d'une clé k dans le tas

L'algorithme d'insertion consiste en trois étapes :

- Trouver le noeud d'insertion z (le nouveau dernier noeud)
- Stocker k à z
- Restaurer la propriété de l'ordre du tas



👉 Pour simplifier, nous montrons que les clés.

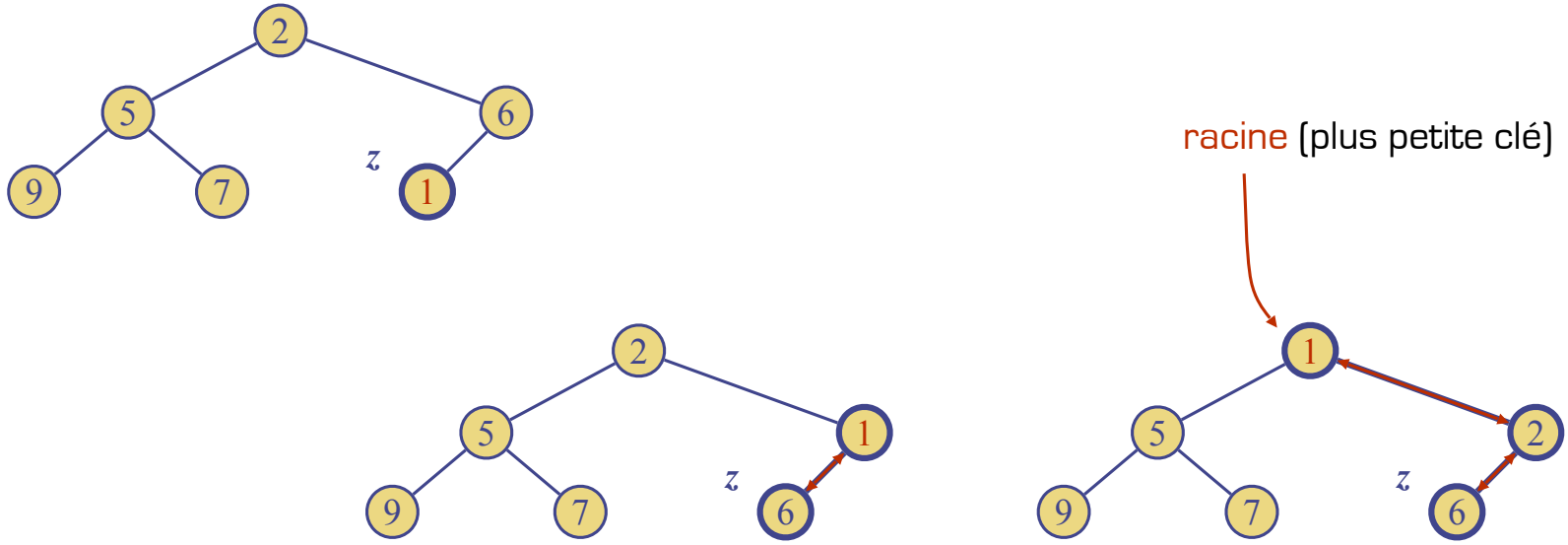
Faire-monter (nager)

Après l'insertion d'une nouvelle clé k, la propriété d'ordre du tas peut être violée (ici, 1 est < 6)

La méthode **swim** restaure la propriété d'ordre du tas en remplaçant k le long d'un chemin ascendant depuis le noeud d'insertion

swim se termine lorsque la clé k atteint la racine ou un noeud dont le parent a une clé plus petite ou égale à k

Étant donné qu'un tas possède une hauteur dans $O(\log n)$, la remontée s'effectue en temps dans $O(\log n)$



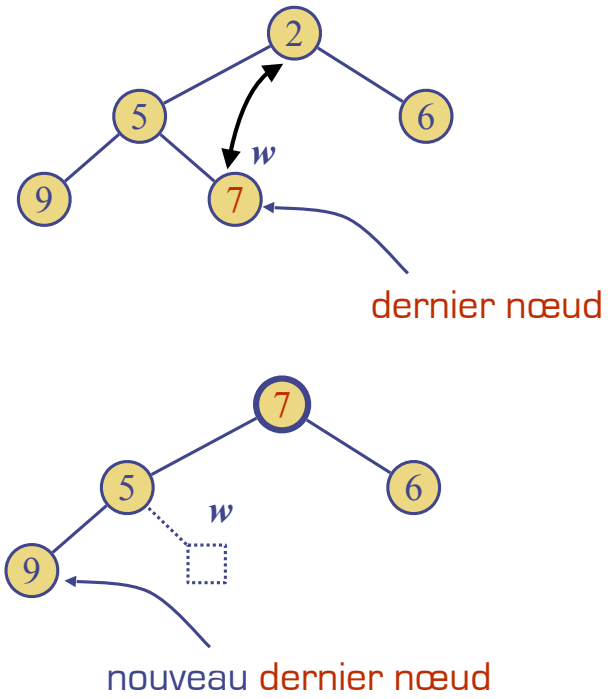
Suppression dans un tas

La méthode **`removeMin`** de la file d'attente avec priorités correspond à la suppression de la clé à la racine du tas (là où se trouve l'élément de plus grande priorité ; la plus petite clé)

L'algorithme de suppression consiste en trois étapes :

- Remplacer la clé à la racine par la clé du dernier nœud w
- Supprimer w
- Restaurer la propriété d'ordre

`removeMin( )`



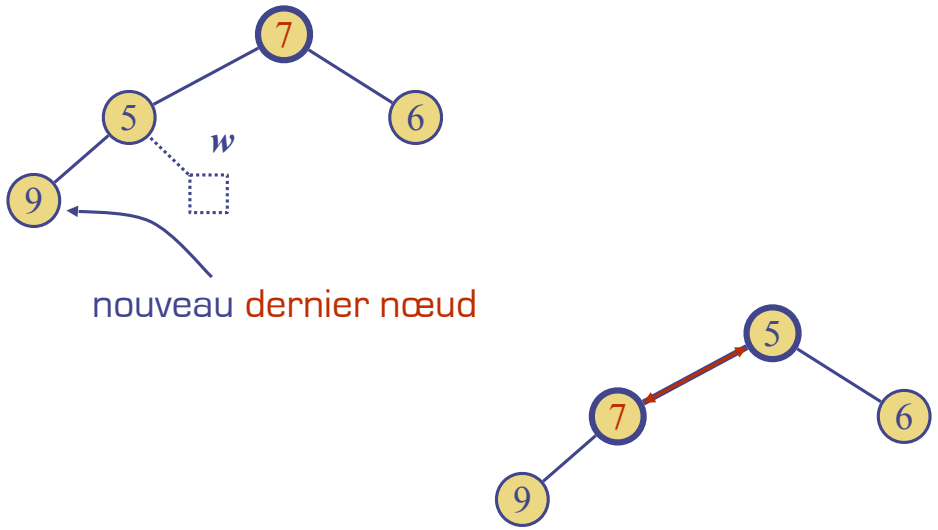
Faire-descendre (couler)

Après avoir remplacé la clé racine par la clé k du dernier noeud, la propriété d'ordre du tas peut être violée

La méthode **sink** restaure la propriété d'ordre du tas en faisant couler la clé k le long d'un chemin qui descend depuis la racine

sink se termine lorsque la clé k atteint une feuille ou un noeud dont les enfants ont des clés supérieures ou égales à k

Comme un tas possède une hauteur dans $O(\log n)$, faire-descendre s'exécute en temps dans $O(\log n)$



Analyse de la file d'attente avec priorités implémentée avec un tas

En supposant que deux clés peuvent être comparées en temps dans $O(1)$ et que le tas T est implémenté avec une représentation arborescente, les méthodes de l'ADT file d'attente avec priorités peuvent être exécutées en $O(1)$ ou en $O(\log n)$, où n est le nombre d'entrées.

L'analyse du temps d'exécution des méthodes repose sur les éléments suivants :

- Le tas T possède n nœuds, chacun stockant une référence à une entrée `(clé, valeur)`
- La hauteur du tas T est $O(\log n)$, puisque T est un arbre binaire complet
- L'opération **`min`** s'exécute en $O(1)$ car la racine de l'arbre contient une entrée de clé minimum
- La localisation de la dernière position d'un tas, comme requis pour **`insert`** et **`removeMin`**, peut être effectuée en $O(1)$ pour une représentation basée sur un tableau, ou en $O(\log n)$ pour une représentation arborescente chaînée
- Dans le pire des cas, **`swim`** et **`sink`** effectuent un nombre d'échanges égal à la hauteur de T

Méthode	Temps d'exécution
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

* temps amorti (implémentation `ArrayList`)
* pire cas (implémentation `LinkedTree`)

Tri-par-tas (heapsort)

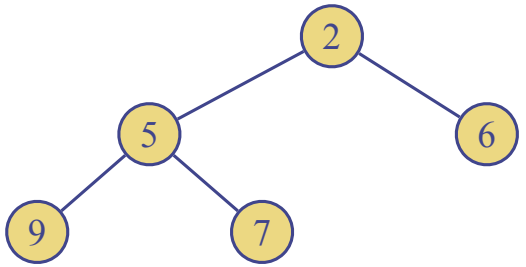
Considérez une file d'attente prioritaire avec n éléments implémentée au moyen d'un tas

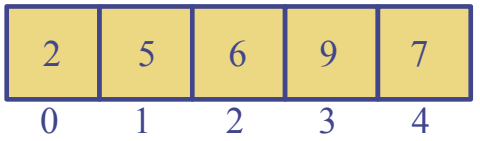
- l'espace utilisé est dans $O(n)$
- les méthodes **`insert`** et **`removeMin`** prennent un temps dans $O(\log n)$
- les méthodes **`size`**, **`isEmpty`** et **`min`** prennent un temps dans $O(1)$

- En utilisant une file d'attente prioritaire basée sur le tas, nous pouvons trier une séquence de n éléments en temps dans $O(n \log n)$
- L'algorithme résultant est appelé tri-par-tas ("heapsort")
- Le tri-par-tas est beaucoup plus rapide que les algorithmes de tri quadratiques, tels que le tri par insertion et le tri par sélection.
- **On a seulement changé la structure de données** !!!

Implémentation d'un tas avec un tableau

- Nous pouvons représenter un tas avec n clés au moyen d'un tableau de longueur n
- Pour le nœud au rang i
  - l'enfant à gauche est à l'index $2i + 1$
  - l'enfant à droite est à l'index $2i + 2$
- Les liens entre les nœuds ne sont pas explicitement stockés
- L'opération **`insert`** correspond à l'insertion à l'index n (à la fin)
- L'opération **`removeMin`** correspond à supprimer à l'index $n-1$
- Le tri-par-tas peut donc s'effectuer "en-place" : les éléments triés à droite du tableau et le tas à gauche (demandez vous pourquoi à gauche ?)

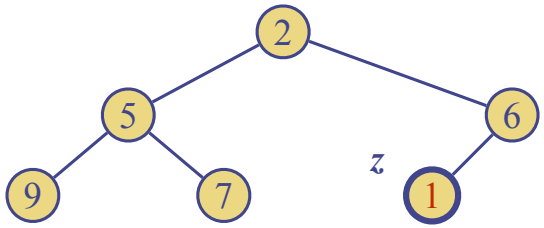


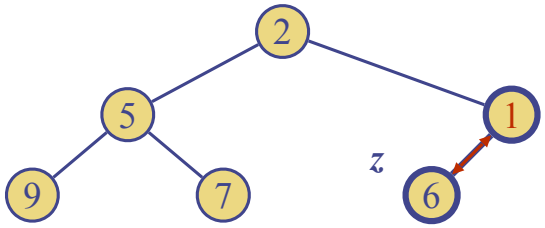


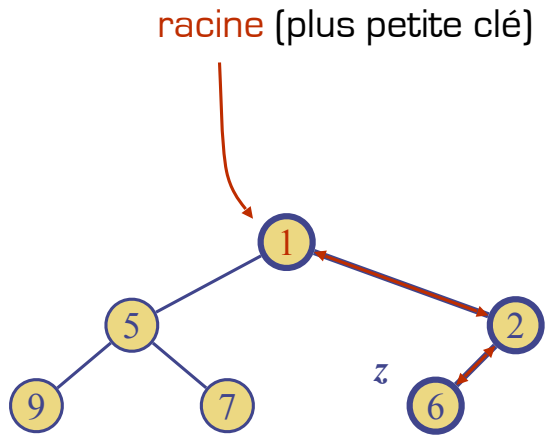
Allons voir le code de

`HeapPriorityQueue.java`

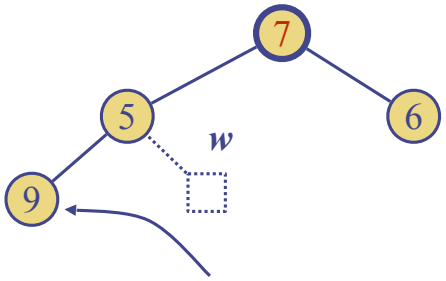
swim

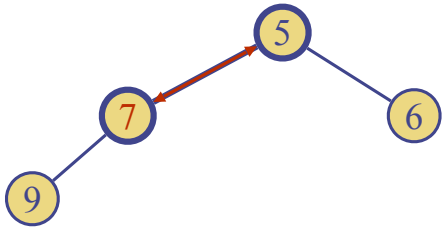






`sink`





Construire un tas en temps $O(n)$

On peut ajouter un constructeur qui prend en argument une liste de n entrées. Quel est le coût de construire un tas initial à partir des ces n éléments ?

Si on insère les n éléments, un à un, on le construit en $O(n \log n)$, soit on lance **`swim`**, qui s'effectue en $O(\log n)$ sur les n éléments.

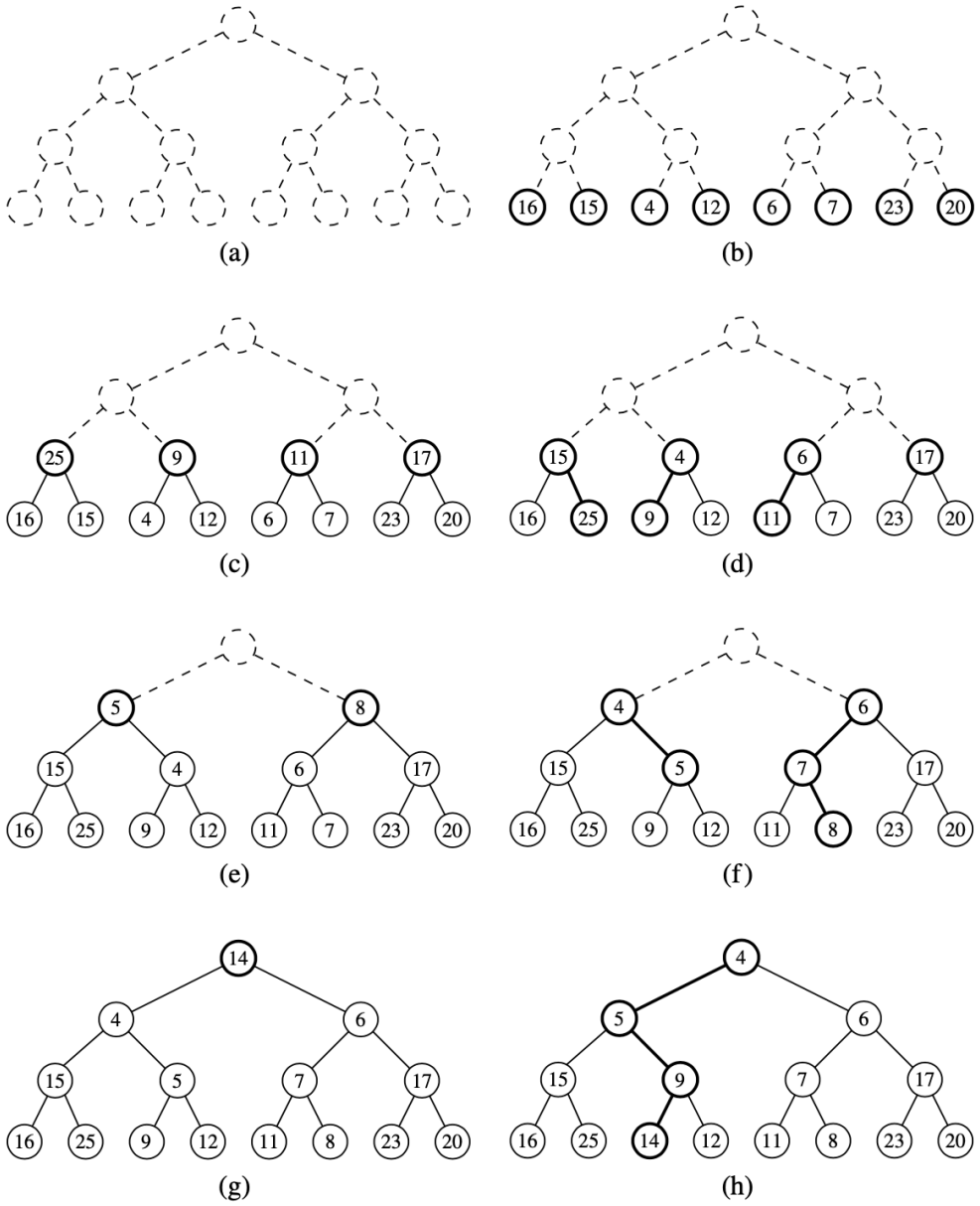
On peut faire mieux ! On insère les n éléments dans le tas dans un ordre quelconque. Ensuite, on peut lancer **`sink`** sur les noeuds internes, du bas vers la racine.

Allons voir le la méthode `heapify` dans
`HeapPriorityQueue.java`

Construction ascendante d'un tas avec 15 entrées: (a et b) nous commençons par construire des tas à 1 entrée au niveau inférieur; (c et d) nous combinons en tas de 3 entrées, puis (e et f) de 7 entrées, jusqu'à ce que (g et h) nous créons le tas final.

Les chemins des **`sink`** sont mis en évidence dans (d, f et h). Pour plus de simplicité, uniquement la clé dans chaque nœud est montrée.

En faisant cela, nous avons appliqué la méthode **`sink`** sur tous les noeuds internes du tas.



Analyse de la construction du tas en temps dans $O(n)$

Nous visualisons le pire cas d'un **`sink`** pour chaque nœud interne avec un chemin qui va d'abord à droite et ensuite à plusieurs reprises à gauche jusqu'à la fin du tas (n.b. ce chemin peut différer du chemin réel lors de la construction et passe par un maximum de nœuds)

Comme **chaque nœud est traversé par au plus deux chemins**, le nombre total de visites est dans O(n)

Cette construction est plus rapide que de faire n insertions successives (qui est dans $O(n \log n)$, et accélère la première phase du tri-par-tas ("heapsort").