Algorithmique P2

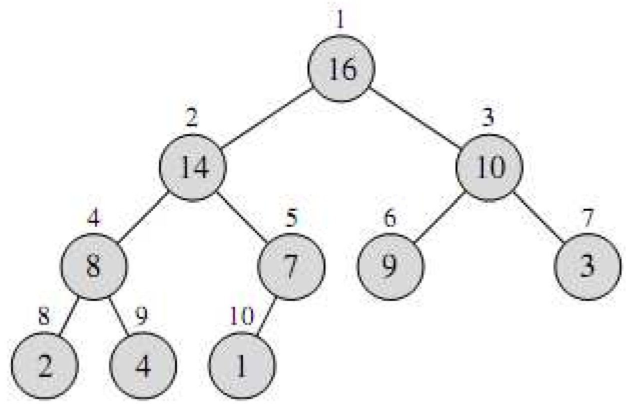
HeapSort et files de priorité

Ulg, 2009-2010

Renaud Dumont

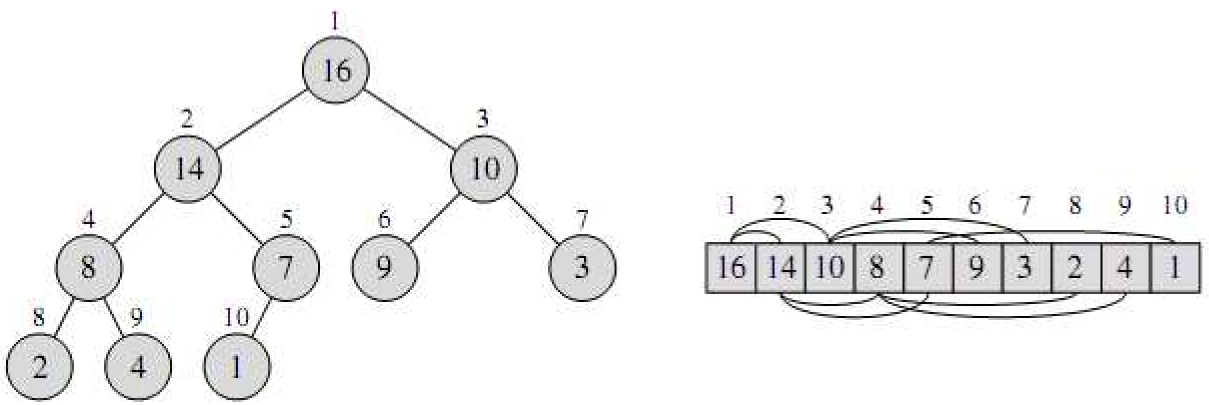
Structure de tas – arbre

- Un tas est une structure de données qui
  - Permet un nouveau type de tri (Tri par tas)
  - Permet l'implémentation de files de priorité
  - Permet des méthodes de compression (Huffman)
  - …
- Un tas peut être vu comme un tableau ou comme un arbre binaire partiellement complet



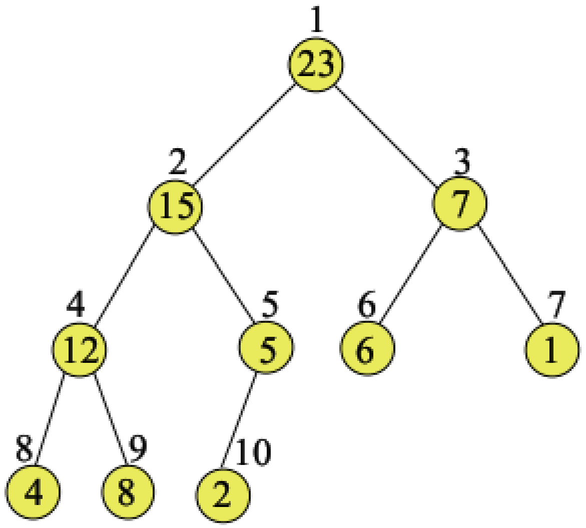
Structure de tas – tableau

- On représente un tas par un tableau A possédant 2 attributs
  - Longueur[A] : nombre d'éléments du tableau
  - Taille[A] : nombre d'éléments du tas rangés dans le tableau
    - Conséquence :
      - Aucun élément après A[Taille[A]] n'est un élément du tas (avec Taille[A] <= Longueur[A])



Indice et parentée

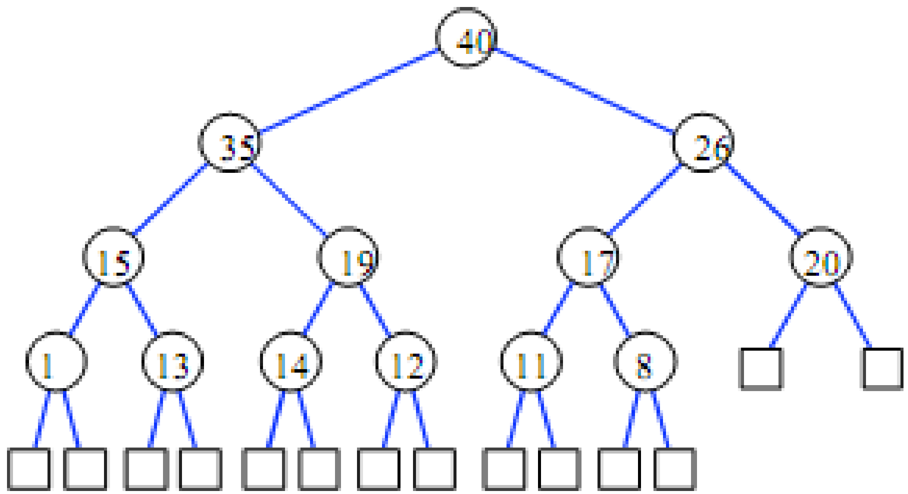
- La racine de l'arbre est A[1], et étant donné l'indice i d'un nœud, on calcule aisément les indices suivants
  - Parent(i) : retourner $\lfloor i/2 \rfloor$
  - Gauche(i) : retourner 2i
  - Droite(i) : retourner 2i+1



1	2	3	4	5	6	7	8	9	10	11	12	…	16
23	15	7	12	5	6	1	4	8	2				

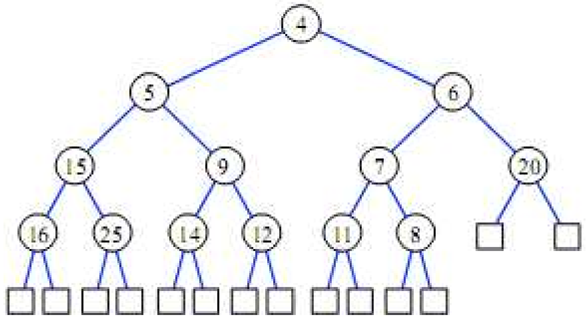
Type de tas

- On distingue deux types de tas
  - Tas max (binary maxheap) → "tas"
    - Propriété : A[Parent(i)]>=A[i]
    - La valeur d'un nœud est au plus égale à celle de son parent
      - La racine contient la valeur la plus grande
      - Les valeurs sont conservées dans les nœuds internes



Type de tas

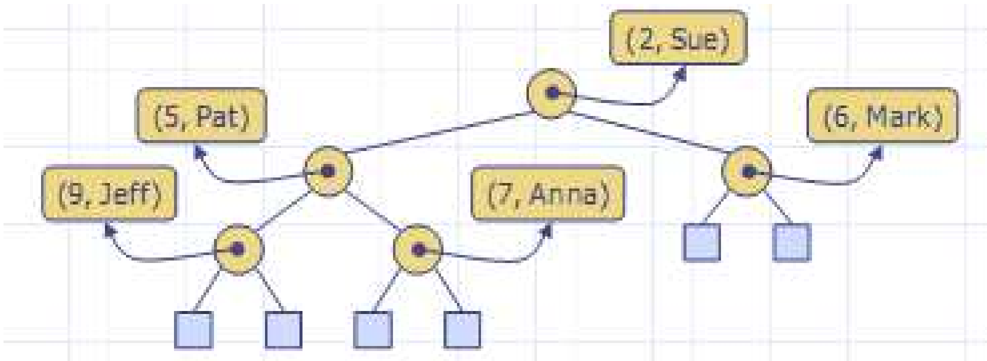
- Tas min (binary minheap)
  - Propriété : A[Parent(i)]<=A[i]
  - La valeur d'un nœud est au minimum égale à celle de son parent
    - La racine contient la valeur la plus petite
    - Les valeurs sont conservées dans les nœuds internes



- Pour l'algorithme de tri par tas, le tas max est utilisé

Tas

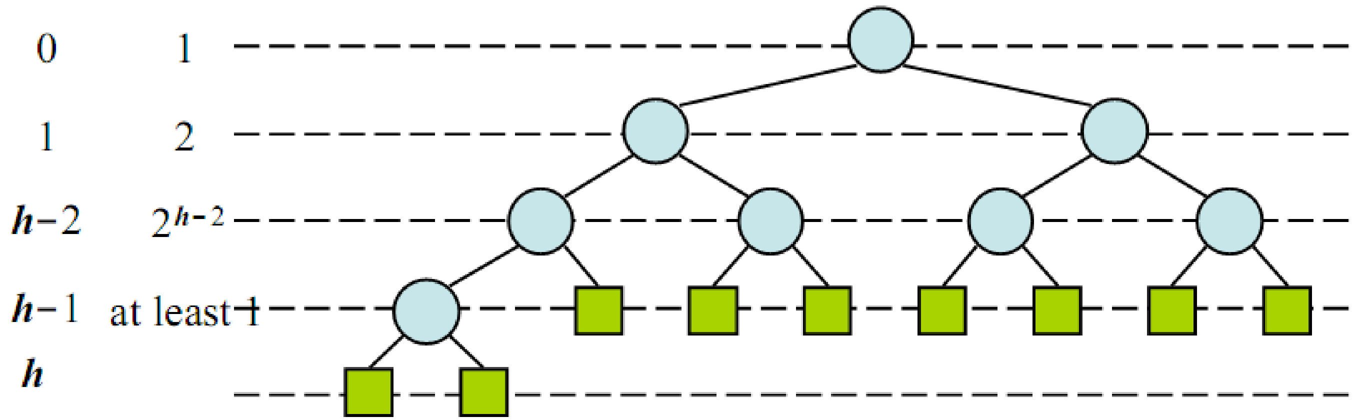
- Pour simplifier les représentations suivantes, les feuilles ne seront pas explicitement représentées
- Les nœuds (internes) conservent en réalité une paire clef-valeur



Heaps Presentation, Goodrich & Tamassia, 2002

Hauteur d'un tas

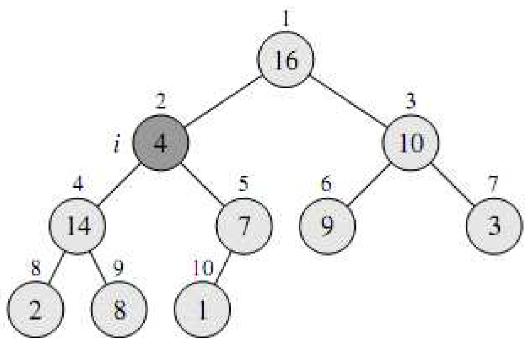
- Théorème : un tas de n nœuds a une hauteur O(log n)
- Démo :
  - Soit h, la hauteur d'un tas de n nœuds
  - Puisqu'il y a 2^i nœuds au niveau i = 0, 1, 2,…, h−2 et au moins un nœud interne au niveau h−1
    - on a $n >= 1+2+4+\ldots+2^{h-2}+1=2^{h-1}-1+1$
  - Donc $n >= 2^{h-1}$, càd $h >= \log n + 1$



- Conséquence
  - les opérations proportionnelle à h sont O(log n)

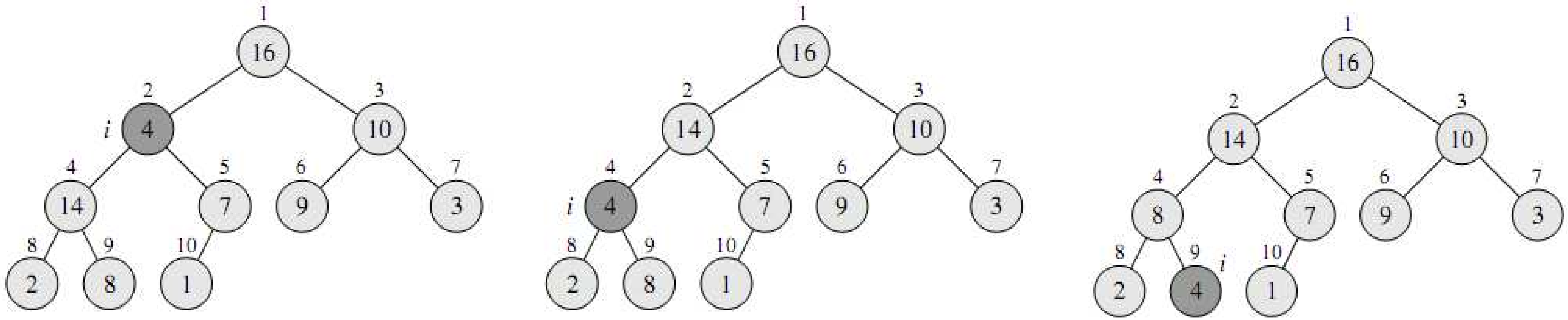
Conservation de la structure de tas

- Entrée : tableau A et un indice i
- Condition :
  - les arbres binaires Gauche(i) et Droite(i) sont des tas max
- A[i] peut être plus petit que ses enfants (et donc un tri est nécessaire)
- La procédure Entasser-Max va faire évoluer l'arbre afin d'obtenir un tas max en i



Entasser–Max

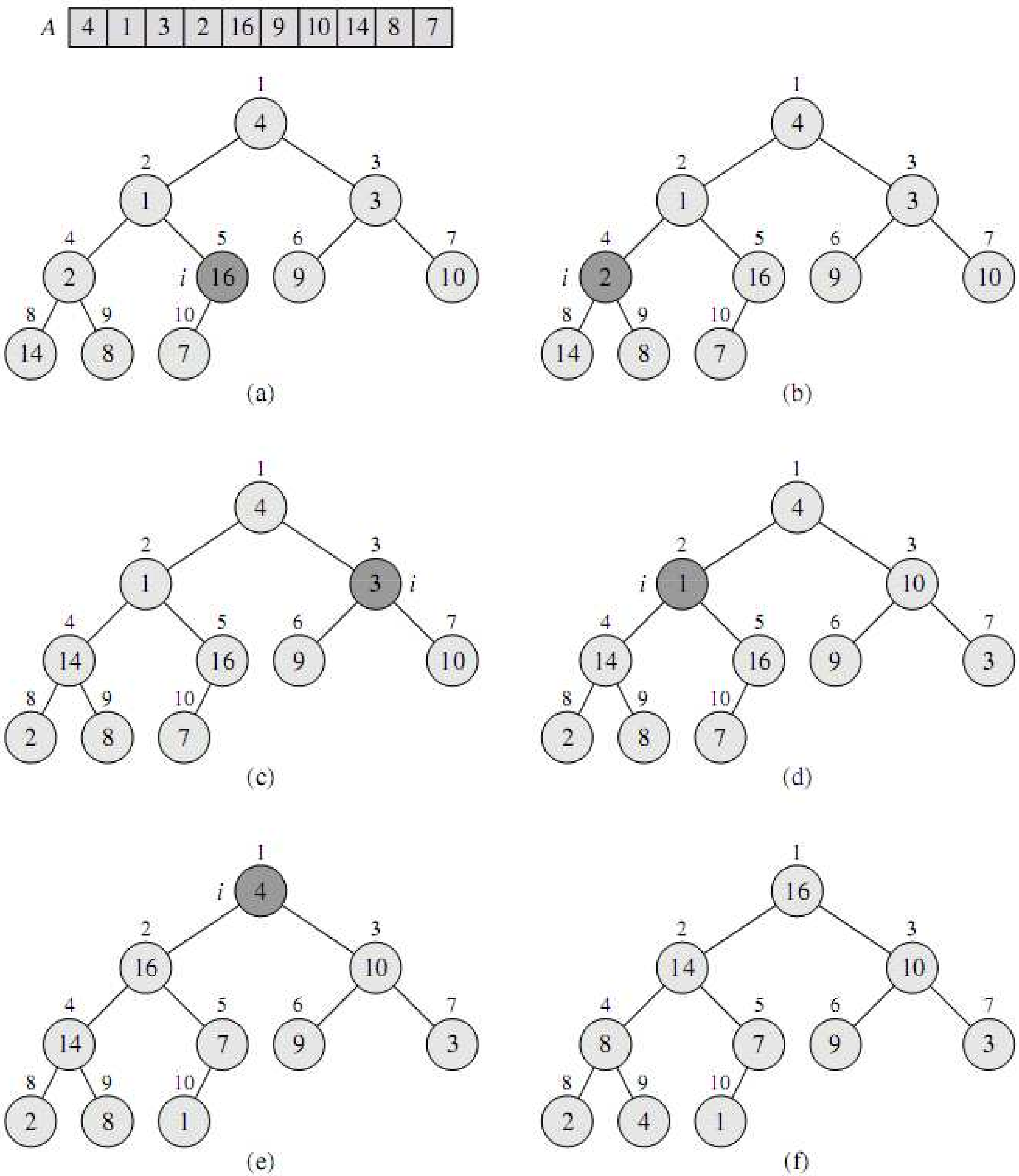
```
ENTASSER-MAX(A, i)
l ← GAUCHE(i)
r ← DROITE(i)
si l ⩽ taille[A] et A[l] > A[i]
    alors max ← l
    sinon max ← i
si r ⩽ taille[A] et A[r] > A[max]
    alors max ← r
si max ≠ i
    alors échanger A[i] ↔ A[max]
          ENTASSER-MAX(A, max)
```



Construction du tas (*Heapify*)

- A partir d'un tableau A[1..n] avec n=longueur[A], on peut construire un tas.
- Les éléments du sous-tableau A[(⌊n/2⌋+1)..n] sont les feuilles de l'arbre
  - Chacun d'eux est donc un tas à 1 élément
- De ce fait, il vient

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


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

Construction

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

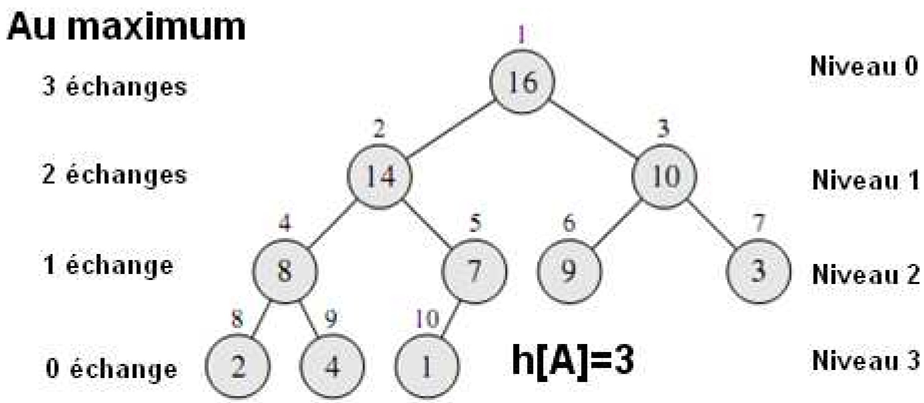
- Invariant de la boucle
  - Au début de chaque itération de la boucle "pour", chaque nœud i+1, i+2, …, n est la racine d'un tas max.
- Initialisation
  - Avant la première itération de la boucle, i=n/2
  - Chaque nœud n/2 + 1, n/2 + 2, … n est une feuille et donc un tas max.
- Conservation de l'invariant
  - Les fils de i sont des éléments d'indice supérieur à i → ce sont des tas max → Entasser-Max(A,i) fait de i une racine d'un tas max
- Terminaison
  - Par construction, chaque nœud est la racine d'un tas max, et en particulier le nœud 1.

Construction

CONSTRUIRE-TAS-MAX(A)

$taille[A] \leftarrow longueur[A]$
pour $i \leftarrow \lfloor longueur[A]/2 \rfloor$ **jusqu'à** 1
    faire ENTASSER-MAX(A, i)

- Théorème : Heapify est O(n)
- Démo :
  - Complexité dépendante de Entasser-Max avec h[A]
    - Opérations d'échange :
      - Pour le niveau 0, pour un nœud, h échanges (au max)
      - Pour le niveau 1, pour un nœud, h-1 échanges (au max)
      - …
      - Pour le niveau i, pour un nœud, h-i échanges (au max)



Construction

- Au niveau i, il y a 2^i nœuds
- Le nombre total d'échanges pour ces 2^i nœuds est donc majoré par

$$2^i(h-i)$$

- Le nombre total d'échanges pour tous les nœuds est donc majoré par

$$\sum_{i=0}^{h} 2^i (h-i) = \sum_{j=0}^{h} 2^{h-j}\, j = 2^h \sum_{j=0}^{h} 2^{-j}\, j$$

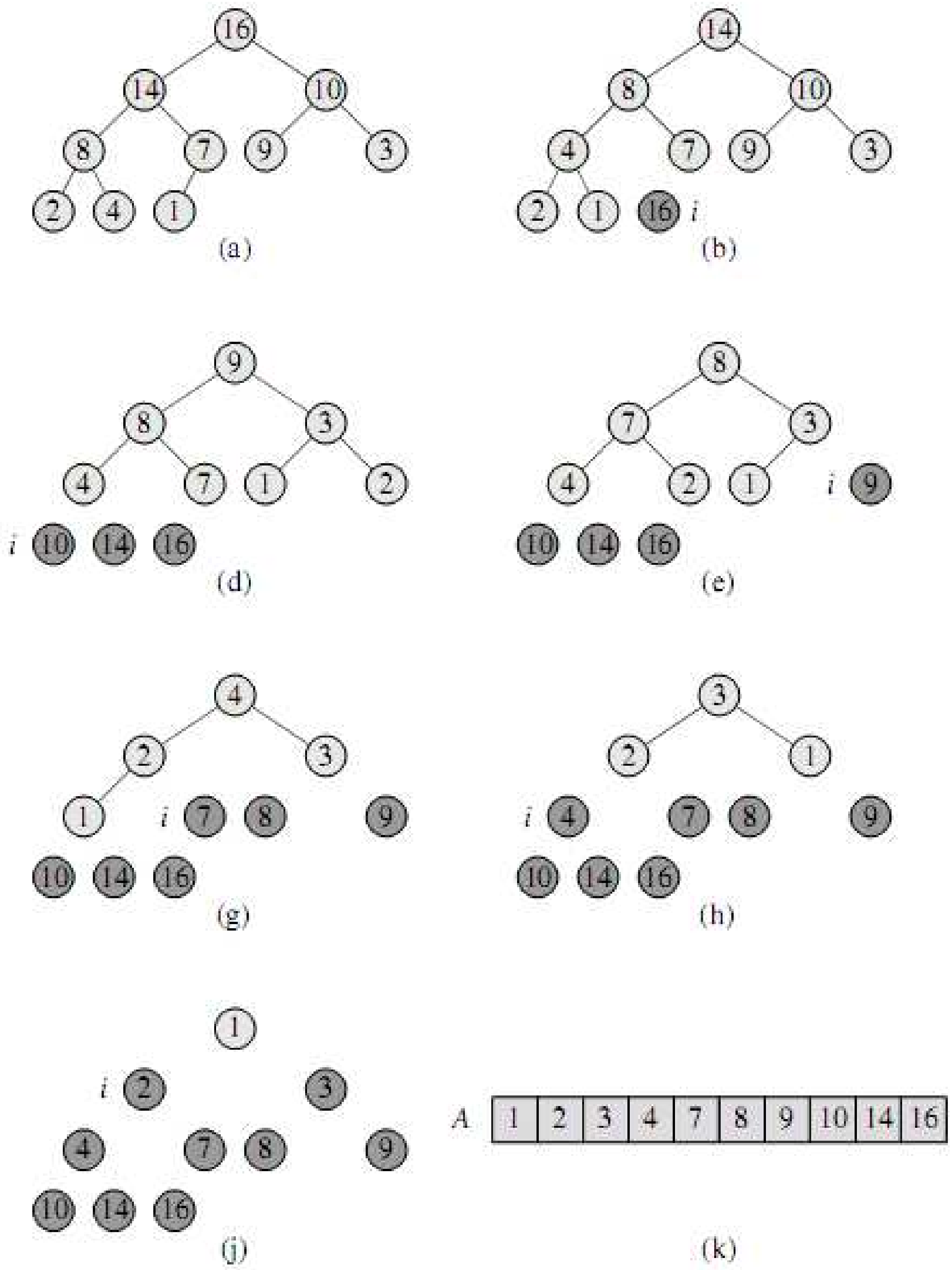
  en posant $j=h-i$.

- Or, $\sum_{j=0}^{h} 2^{-j}\, j \leq 2$
- Donc, $2^h \sum_{j=0}^{h} 2^{-j}\, j \leq 2^h.2 = 2n$ → O(n)

Tri par tas (HeapSort)

- Construction d'un tas max à partir d'un tableau A[1..n] où n = longueur[A]
- Ensuite, A[1] contenant l'élément de valeur la plus élevée, on peut l'échanger avec A[n].
- On retire le nœud "n" du tas (Taille[A]−1) et on réorganise A[1..(n−1)] pour qu'il corresponde à un tas max.
- On répète l'opération jusqu'à ce que le tas ait une taille de 1.

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


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
A
1 2 3 4 7 8 9 10 14 16

Tri par tas

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

- Tri O(n log n)
  - Appel à Construire-Tas-Max : O(n)
  - Dans la boucle "pour" : n appels
    - Appel à Entasser-Max : O(log n)
    - Boucle : O(n log n)
  - Complexité totale : O(n log n)

- Et si le tableau est déjà trié ?
  - HeapSort mélange les premiers éléments du tableau pour la construction du tas

Files de priorité (*Priority Queue*)

- Structure de données permettant de gérer un ensemble S d'éléments auxquels on associe une "clé"
  - Cette clé permet la définition des priorités
- Application directe de la structure de tas
  - 4 opérations (File-Max)
    - Maximum(S)
    - Extraire-Max(S)
    - Augmenter-Clé(S,x,k)
    - Insérer(S,x)
  - Si on inverse l'ordre des priorités, on obtient les opérations symétriques (Minimum, Extraire-Min, Diminuer-Clé) → File-Min
- Utilisation :
  - Cas d'une liste de tâches sur un ordinateur
    - La file de priorité gère les tâches en attente selon leur priorité
    - Quand une tâche est terminée, on exécute la tâche de priorité la plus élevée suivante
    - Il est possible d'insérer dans la file une tâche, éventuellement prioritaire.

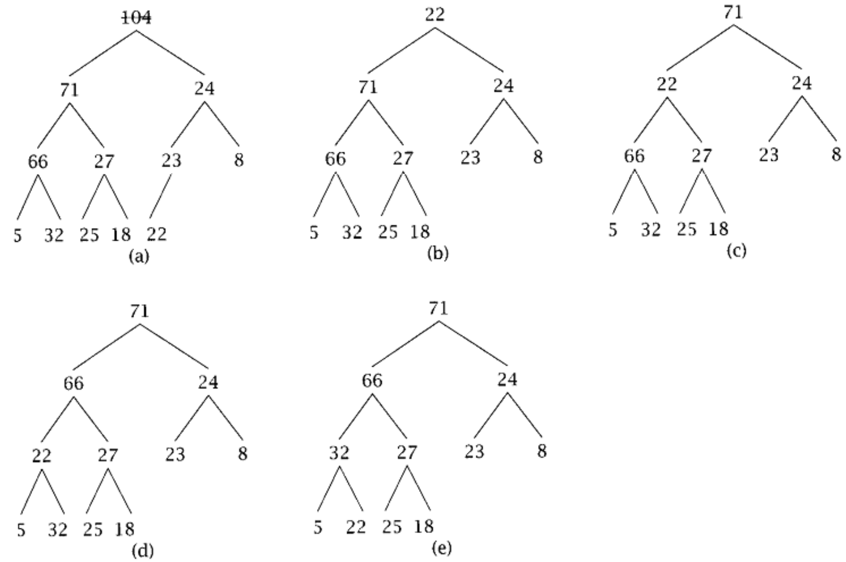
Files de priorité (max)

- Retourner l'élément ayant la clé maximale

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

- Retourner l'élément ayant la clé maximale en le supprimant de la file

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

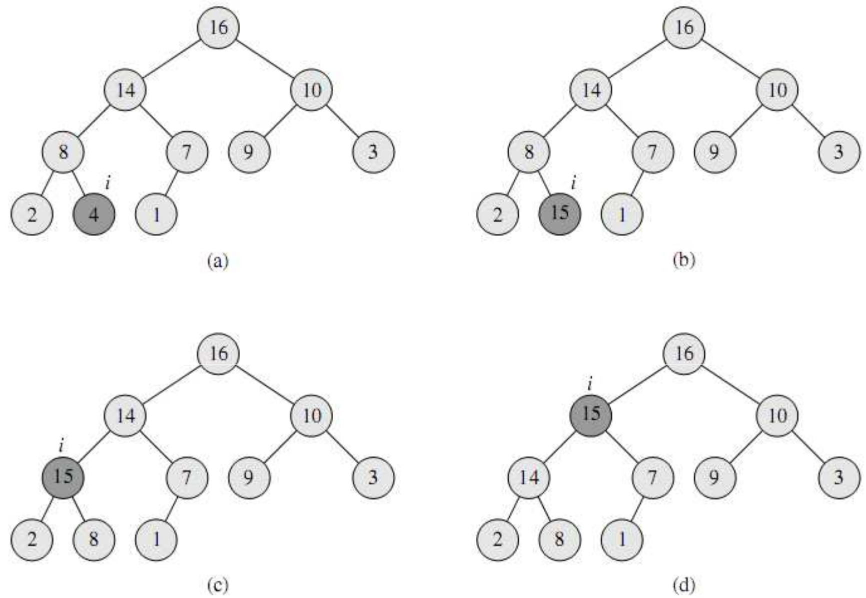


  - Parcours *DownHeap*
    - Reconstruction du tas en déplaçant le nœud courant de haut en bas
      - Echange avec le fils de clé maximale (→ Entasser-Max)
      - Arrêt quand feuille ou clé supérieure à celles des 2 fils.

Files de priorité

- Augmenter la valeur d'une clé

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



  - Parcours *UpHeap*
    - Reconstruction du tas en déplaçant le nœud courant du bas vers le haut
      - Echange avec le père (1 seul choix possible)
      - Arrêt quand racine ou clé inférieure à celle du père

Files de priorité

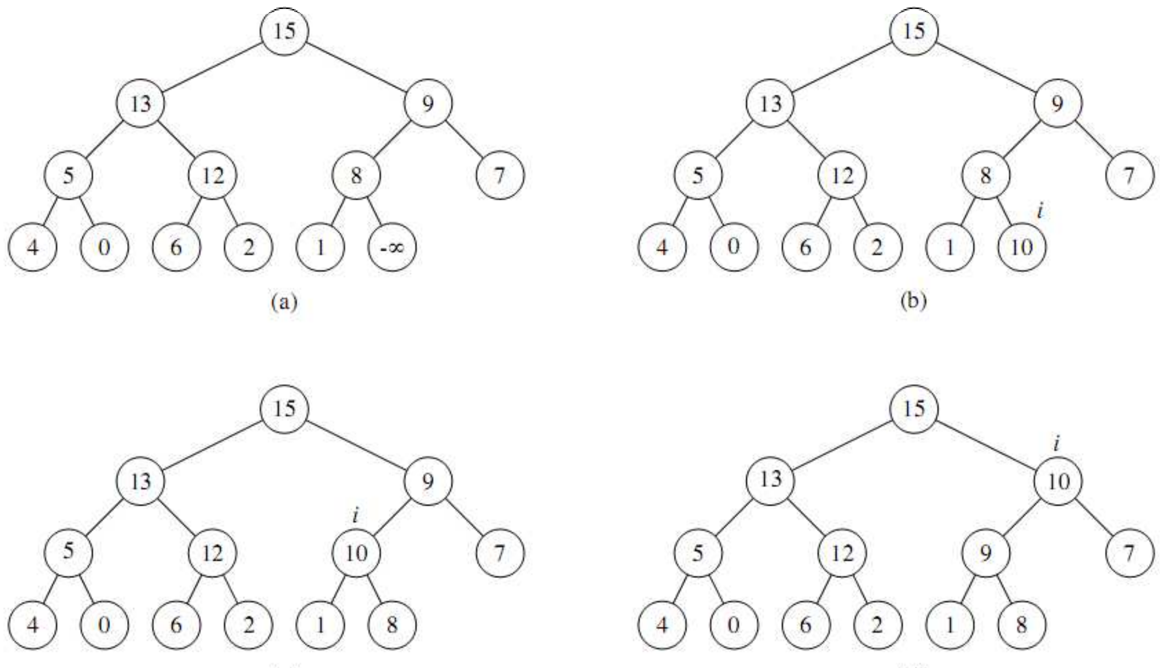
- Insérer un élément

INSÉRER-TAS-MAX(A, *clé*)

$taille[A] \leftarrow taille[A] + 1$

$A[taille[A]] \leftarrow -\infty$

AUGMENTER-CLÉ-TAS(A, $taille[A]$, *clé*)



(a) (b) (c) (d)

Files de priorité

- Comment implémenter une pile à l'aide d'une file de priorité ?
  - File-max
  - Priorité : date d'insertion
- Comment implémenter une file à l'aide d'une file de priorité ?
  - File-Min
  - Priorité : date d'insertion
- Comment implémenter une file aléatoire (enqueue, dequeue aléatoire) à l'aide d'une file de priorité ?
  - Priorités aléatoires entre 0 et 1

Algorithmique P2

Une application des arbres : le codage de Huffman

Ulg, 2009-2010
Renaud Dumont

Codage de Huffman

- Soit une suite de caractères et une table des fréquences d'occurrences de ces caractères
- Exemple
  - Fichiers de 100.000 caractères
  - 6 caractères et nombre d'occurrences associé

	a	b	c	d	e	f
Fréquence (en milliers)	45	13	12	16	9	5

  - Longueur fixe : 300.000 bits

Mot de code de longueur fixe	000	001	010	011	100	101

  - Longueur variable : 224.000 bits

Mot de code de longueur variable	0	101	100	111	1101	1100

    - Gain d'environ 25%

Codage de Huffman

- Codage préfixe
  - Aucun mot de code n'est préfixe d'un autre

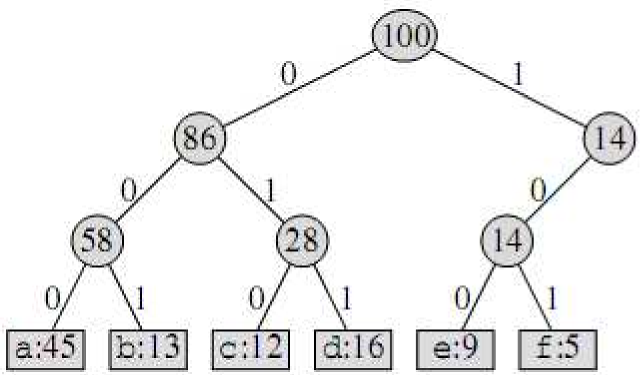
Mot de code de longueur variable	0	101	100	111	1101	1100

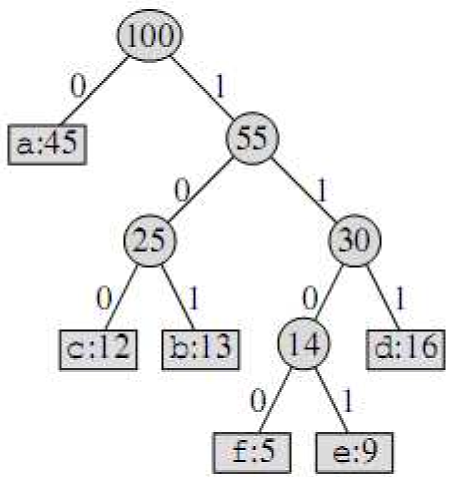
    - La chaine abc sera représentée par la concaténation des codes respectifs de a, b et c soit a = 0, b = 101, c = 100
      - 0101100
  - Avantage : décodage simplifié car absence d'ambiguïté
    - Identification du premier code, traduction, suppression
    - Exemple
      - La chaine 001011101 sera décodée en 0,0,101,1101 et donc aabe

Huffman

	a	b	c	d	e	f
Fréquence (en milliers)	45	13	12	16	9	5
Mot de code de longueur fixe	000	001	010	011	100	101
Mot de code de longueur variable	0	101	100	111	1101	1100

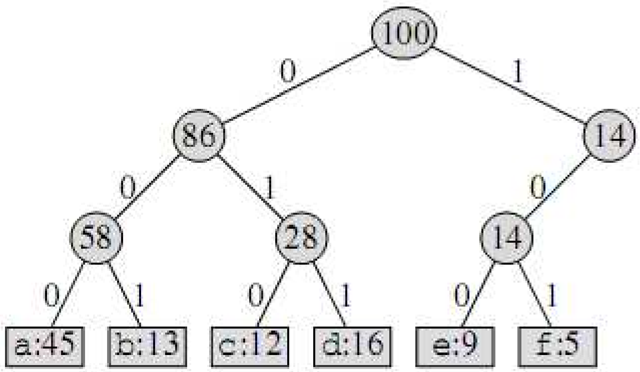
- Représentation sous forme d'un arbre
  - Arbre binaire dont les feuilles sont les caractères donnés
  - Le code associé à un caractère = chemin de la racine à ce caractère avec
    - 0 = vers le fils à gauche
    - 1 = vers le fils à droite



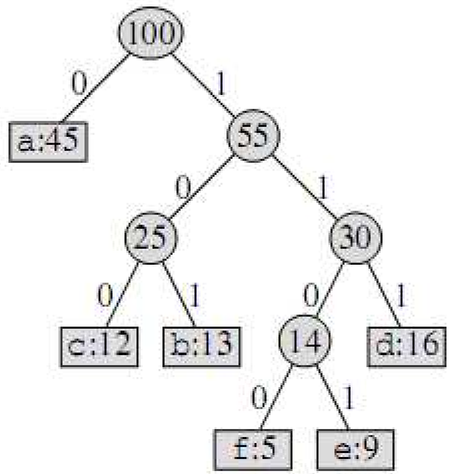


Codage de Huffman

- Codage optimal = arbre localement complet
  - Si C est l'alphabet dont sont issus les caractères
  - Alors l'arbre représentant un codage préfixe optimal possède exactement C feuilles (une par lettre de l'alphabet) et C-1 nœuds internes.



Codage de taille fixe

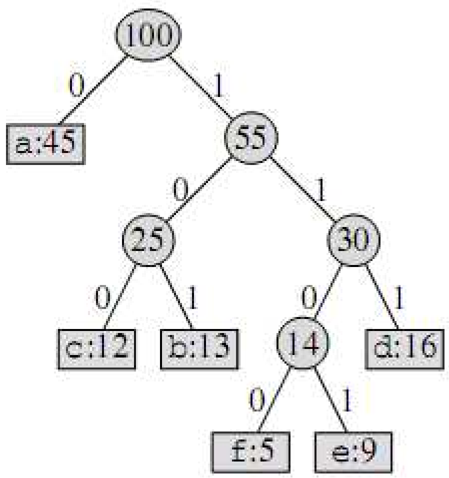


Codage optimal

Codage de Huffman

- Chaque feuille est étiquetée avec un caractère et sa fréquence d'apparition.
- Chaque nœud interne est étiqueté avec la somme des fréquences des feuilles de ses sous-arbres.

- Principe :
  - Fusionner les objets (feuilles ou nœuds) dont les fréquences d'apparition sont les plus faibles.



Codage de Huffman

- Construction de l'arbre
  - Soit un alphabet C de n caractères dont chaque caractère a une fréquence f[c]
  - On construit l'arbre de bas en haut
    - Au départ des |C| feuilles, on effectue |C|−1 fusions
    - Une file de priorité min Q, dont les clés sont prises dans f, permet d'identifier les 2 objets les moins fréquents à fusionner
    - Le résultat d'une fusion est un nouveau nœud dont la fréquence est la somme des deux objets fusionnés.

```
HUFFMAN(C)
n ← |C|
Q ← C
pour i ← 1 à n − 1
    faire allouer un nouveau nœud z
        gauche[z] ← x ← EXTRAIRE-MIN(Q)
        droite[z] ← y ← EXTRAIRE-MIN(Q)
        f[z] ← f[x] + f[y]
        INSÉRER(Q, z)
```

Codage de Huffman