



INFO0501

ALGORITHMIQUE AVANCÉE

COURS 2

STRUCTURES DE DONNÉES ÉLÉMENTAIRES
TAS, PILES, FILES, LISTES CHAÎNÉES, ARBRES



UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

Pierre Delisle
Département de Mathématiques, Mécanique et Informatique
Septembre 2018

Plan de la séance

- Tas
- Structures de données dynamiques élémentaires
 - Piles
 - Files
 - Listes
 - Arbres
- Bibliographie
 - T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Algorithmique", 3^e édition, Dunod, 2010



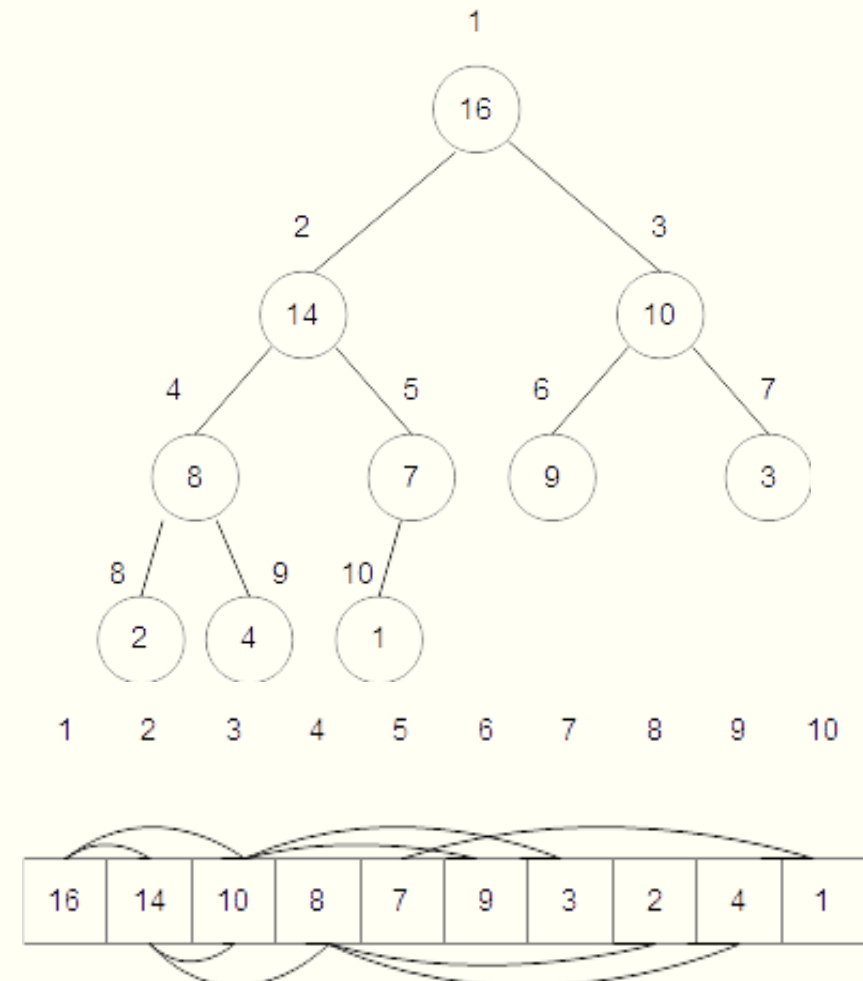
TAS

Tas (binaire)

- Tableau ayant des caractéristiques spécifiques sur l'emplacement des éléments
 - Peut être vu (mais non stocké) comme un arbre binaire presque complet
- Plusieurs utilisations
 - Tri
 - Files de priorités
 - ...

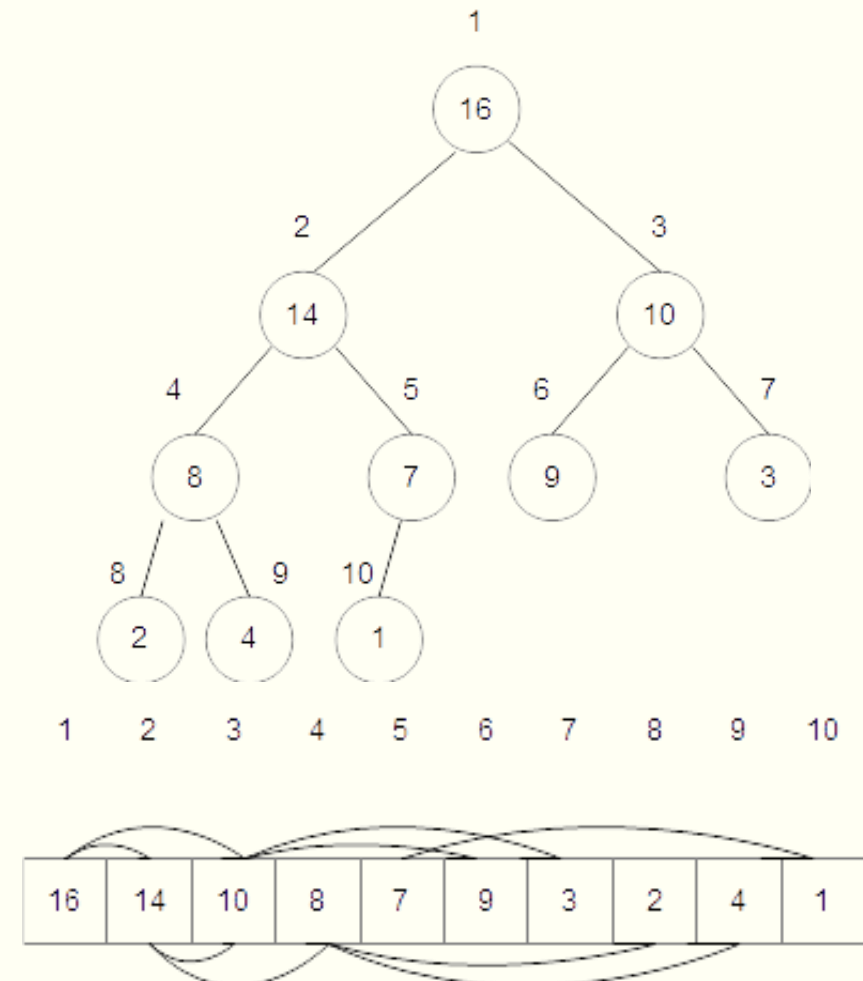
Tas (binaire)

- Chaque nœud de l'arbre correspond à un élément du tableau
- L'arbre est complètement rempli à tous les niveaux
 - Sauf éventuellement le dernier qui est rempli de gauche à droite
- Un tableau t représentant un tas possède 2 attributs
 - **longueur** : nombre maximum d'éléments
 - **taille** : nombre d'éléments du tas effectivement rangés dans le tableau
- Éléments valides du tas $\rightarrow t[1 .. t.taille]$



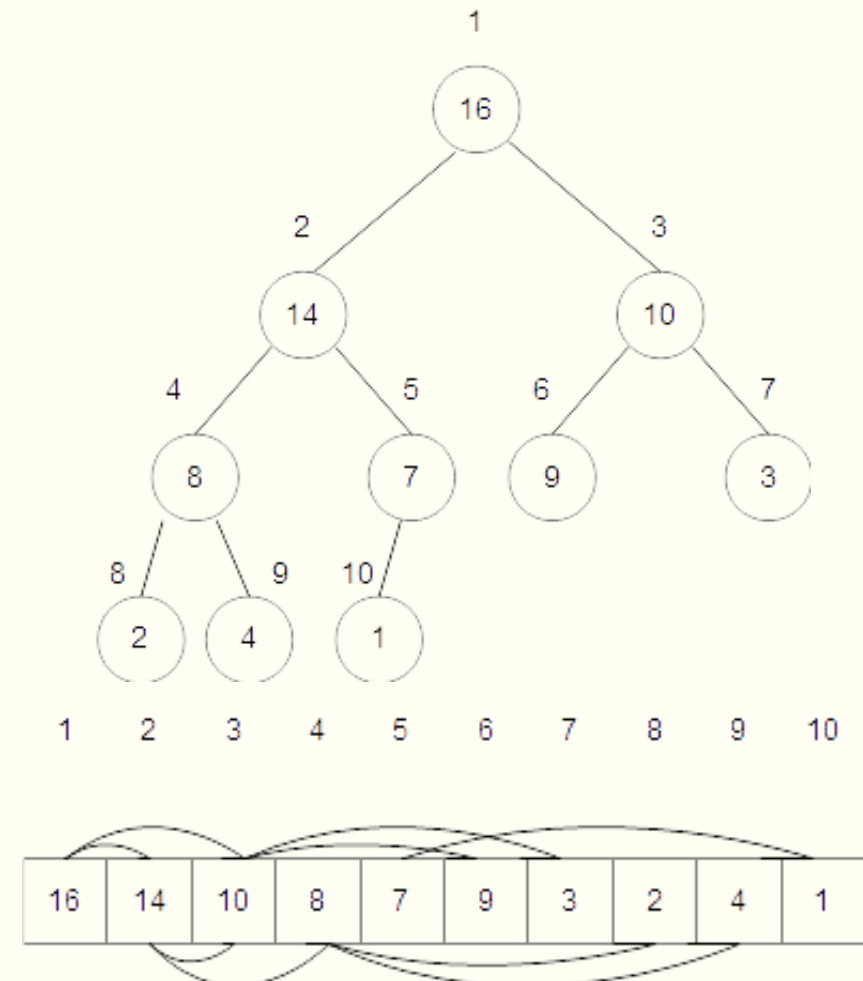
Tas (binaire)

- Racine de l'arbre
 - $t[1]$
- Étant donné l'indice i d'un nœud, on peut calculer
 - L'indice de son parent $\text{PARENT}(i)$
 - $\lfloor i / 2 \rfloor$
 - L'indice de son enfant de gauche $\text{GAUCHE}(i)$
 - $2i$
 - L'indice de son enfant de droite $\text{DROITE}(i)$
 - $2i + 1$



Tas (binaire)

- Propriété de tas (ici pour un tas max)
 - Pour chaque nœud i autre que la racine
 - La valeur d'un nœud est au plus égale à celle du parent
 - $t[\text{PARENT}(i)] \geq t[i]$
- Plus grand élément stocké à la racine
- Hauteur d'un nœud
 - Nombre d'arcs sur le chemin le plus long reliant le nœud à une feuille
- Hauteur d'un tas de n éléments
 - Hauteur de la racine
 - $\Theta(\lg n)$



Conservation de la propriété de tas

- Une modification du tableau doit assurer la conservation de la propriété de tas
- Procédure ENTASSER-MAX(t, i)
 - Suppose que les arbres binaires enracinés en GAUCHE(i) et DROITE(i) sont des tas max
 - ... mais que $t[i]$ puisse être plus petit que ses enfants
 - ... violant ainsi la propriété de tas max
 - On fait alors descendre la valeur de $t[i]$ dans le tas max de sorte à rétablir la propriété de tas
- Fonctionnement de ENTASSER-MAX
 - À chaque étape, on détermine le plus grand des éléments $t[i]$, GAUCHE(i) et DROITE(i)
 - Si $t[i]$ est le max, on a déjà un tas max
 - Sinon, on échange $t[i]$ avec l'enfant qui est le max et on rappelle ENTASSER-MAX récursivement
- Exemple 1
 - ENTASSER-MAX avec $t = \langle 16, 4, 10, 14, 7, 9, 3, 2, 8, 1 \rangle$
- Temps d'exécution proportionnel à la hauteur de l'arbre
 - $O(\lg n)$

Construction d'un tas

- Pour convertir un tableau $t[1..n]$ (avec $n = t.\text{longueur}$) en tas max
 - On utilise la propriété suivante
 - Les éléments du sous-tableau $t[(n / 2 + 1) .. n]$ sont tous des feuilles de l'arbre, donc des tas max à 1 élément
 - On utilise la procédure ENTASSER-MAX sur les autres nœuds de l'arbre, à l'envers
- Exemple 2
 - CONSTRUIRE-TAS-MAX avec $t = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
- Temps d'exécution : $O(n)$ appels à ENTASSER-MAX
 - $O(n \lg n)$ (une analyse plus fine peut démontrer que c'est en fait $\mathcal{O}(n)$)

Tri par tas

- Démarre en construisant un tas max avec le tableau d'entrée
- Principe
 - Élément maximal du tableau \rightarrow racine de l'arbre $t[1]$
 - On peut donc le placer à sa position finale correcte, à la fin du tableau, en l'échangeant avec $t[n]$
 - ... et en enlevant le nœud n du tas en décrémentant $t.taille$
 - On rétablit ensuite la propriété de tas max en appelant ENTASSER-MAX sur la racine
 - Et on répète le processus jusqu'à arriver à un tas de taille 2
- Exemple 3
 - TRI-PAR-TAS avec $t = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$
- CONSTRUIRE-TAS-MAX
 - $O(n \lg n)$ (ou $O(n)$, mais ça ne change pas le résultat)
- n appels à ENTASSER-MAX
 - $O(n \lg n)$
- Temps d'exécution
 - $\Theta(n \lg n)$
 - (Comme le tri par fusion)
- Trie sur place
 - (Comme le tri par insertion)



STRUCTURES DE DONNÉES DYNAMIQUES ÉLÉMENTAIRES

Ensemble

- Notion fondamentale en informatique
- Ensembles dynamiques
 - Peuvent croître, diminuer et subir des modifications au cours du temps
- Principales opérations sur les ensembles
 - Insérer
 - Supprimer
 - Tester l'appartenance
- Un ensemble qui supporte ces trois opérations est appelé dictionnaire
- La meilleure implémentation d'un ensemble dynamique dépend des opérations qu'il doit reconnaître
- Implémentation classique
 - Chaque élément est représenté par un objet
 - Attributs manipulés par un pointeur vers l'objet
- Clé
 - Attribut qui sert à identifier l'objet et à implémenter l'ensemble
- Données satellites
 - Attributs qui servent à stocker les autres données de chaque objet de l'ensemble

Opérations sur les ensembles dynamiques

Requêtes

- RECHERCHER (S, k)
 - Étant donné un ensemble S et une valeur de clé k
 - ... retourne un pointeur x sur un élément de S tel que $x.cle = k$ ou NIL si l'élément n'appartient pas à S
- MINIMUM (S)
 - Retourne l'élément de S (totalement ordonné) ayant la plus petite clé
- MAXIMUM (S)
 - Retourne l'élément de S (totalement ordonné) ayant la plus grande clé
- SUCCESSEUR (S, x)
 - Étant donné un élément x dont la clé appartient à S (totalement ordonné), retourne le prochain élément de S qui est plus grand que x , ou NIL si x est l'élément maximal
- PRÉDÉCESSEUR (S, x)
 - Étant donné un élément x dont la clé appartient à S (totalement ordonné), retourne le prochain élément de S qui est plus petit que x , ou NIL si x est l'élément minimal

Opérations de modification

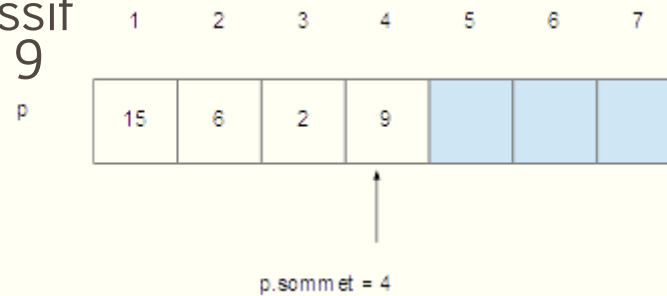
- INSERTION (S, x)
 - Ajoute à l'ensemble S l'élément pointé par x
 - Suppose que tous les attributs de l'élément x requis pour la définition de l'ensemble ont déjà été initialisés
- SUPPRESSION (S, x)
 - Étant donné un pointeur x vers un élément de l'ensemble S
 - ...élimine x de S
 - Utilise un pointeur vers l'élément x et non une valeur de clé

Piles

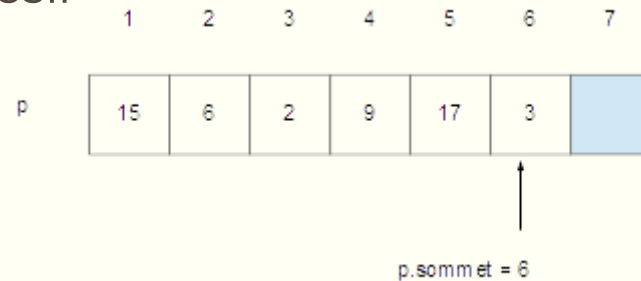
- Dernier entré, premier sorti → LIFO
- Insérer → Empiler
- Supprimer → Dépiler
- Implémentation par tableau
 - Au plus n éléments → tableau p $[1...n]$
 - Possède un attribut $p.sommet$ qui indexe l'élément le plus récemment inséré
 - $p[1]$ → élément situé à la base de la pile
 - $p[p.sommet]$ → élément situé au sommet de la pile

Piles

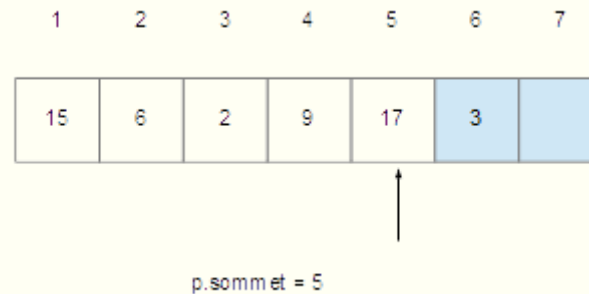
- Après empilage successif des valeurs 15, 6, 2 et 9



- Après empilage successif des valeurs 17 et 3



- Après un dépilage
 - On récupère la valeur 3
 - La case 6 est ensuite non définie, on ne peut pas la réutiliser



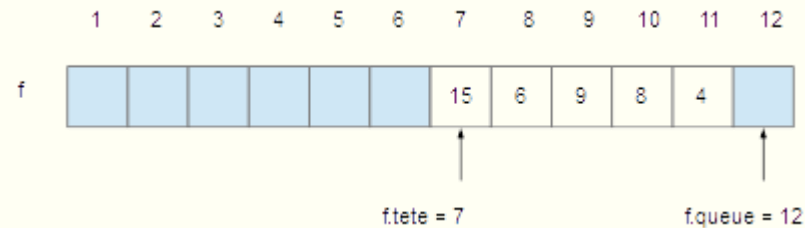
- EMPLER (p, x)
 - $\alpha(1)$
- DÉPILER (p)
 - $\alpha(1)$
- PILE-VIDE (p)
 - $\alpha(1)$
- RECHERCHER ?

Files

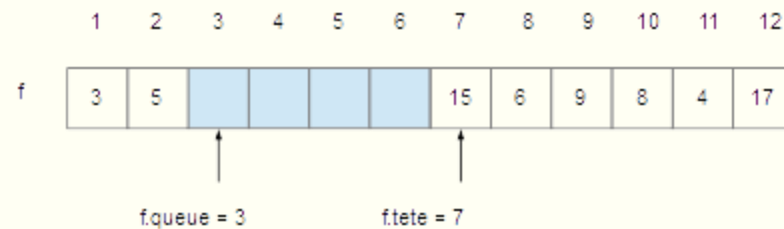
- Premier entré, premier sorti → FIFO
- Insérer → Enfiler
- Supprimer → Défiler
- Implémentation par tableau (circulaire)
 - Au plus $n - 1$ éléments → tableau $f[1...n]$
 - Attribut $f.queue$ qui indexe la queue
 - Enfiler : insertion en $f[f.queue]$
 - Attribut $f.tête$ qui indexe la tête
 - Défiler : suppression en $f[f.tete]$

Files

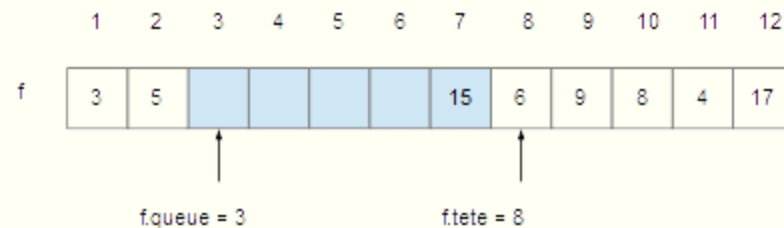
- File contenant 5 éléments



- Après enfilage des valeurs 17, 3 et 5



- Après défilage
 - Retourne la valeur 15
 - L'emplacement 7 n'est plus accessible



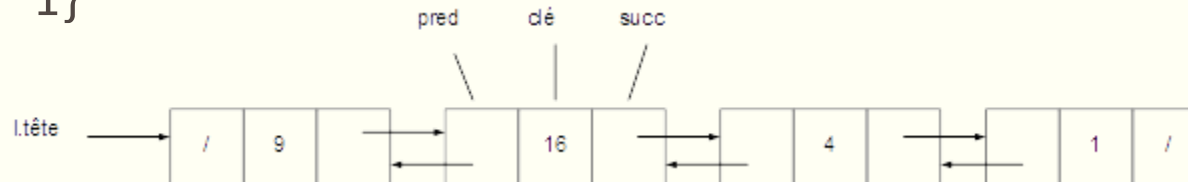
- ENFILER (f, x)
 - $\mathcal{O}(1)$
- DÉFILER (f)
 - $\mathcal{O}(1)$
- RECHERCHER ?

Listes chaînées

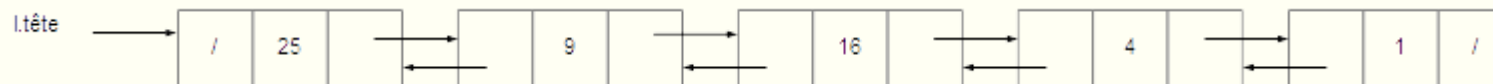
- Objets arrangés linéairement
 - Ordre déterminé par un pointeur dans chaque objet
- Liste doublement chaînée
 - Chaque élément/objet x comporte 3 attributs
 - *clé*
 - *succ* → pointeur sur le successeur de x dans la liste (NIL pour le dernier élément de la liste)
 - *préd* → pointeur sur le prédécesseur de x dans la liste (ou NIL pour le premier élément de la liste)
 - tête → pointeur sur le premier élément (NIL si vide)

Listes (doublement) chaînées

- Liste représentant l'ensemble {9, 16, 4, 1}



- Liste après insertion d'un élément dont la clé vaut 25



- Liste après suppression de l'élément dont la clé vaut 4



- RECHERCHER-LISTE (l, k)

- $\mathcal{O}(n)$

- INSÉRER-LISTE (l, x)







- $\mathcal{O}(1)$

- SUPPRIMER-LISTE (l, x)

- $\mathcal{O}(1)$ si on a déjà un pointeur sur l'élément x à supprimer

- $\mathcal{O}(n)$ si on supprime à partir de la clé k

Liste ou tableau ?

	Insertion/ Suppression	Mémoire	Accès aléatoire
Tableau			
Liste			

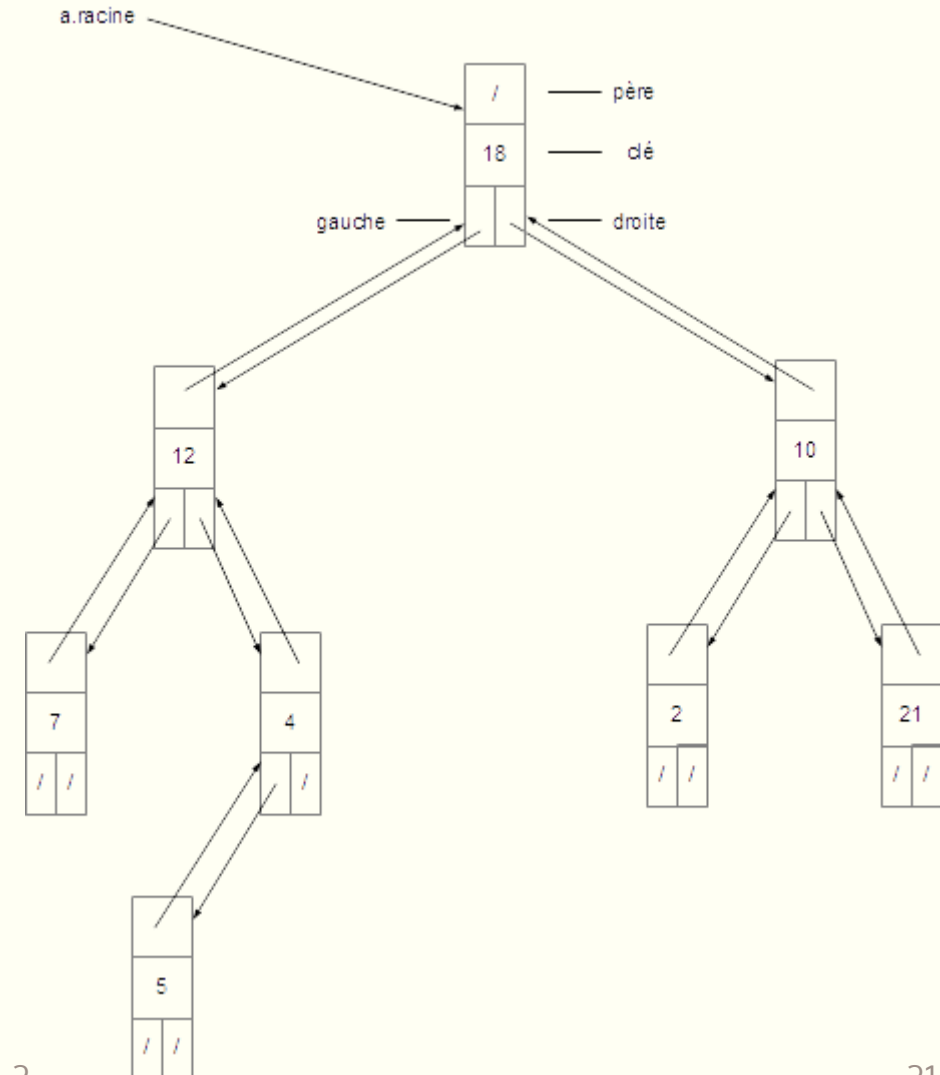
Arbres (binaires)

■ Représentation chaînée

- Chaque nœud/objet contient les attributs
 - *clé*
 - *père* → pointeur vers le père (NIL pour la racine)
 - *gauche* → pointeur vers le fils gauche
 - *droite* → pointeur vers le fils droit
- *racine* → pointeur sur l'élément racine (NIL si arbre vide)

■ Temps d'exécution ?

- Dépend de l'ordre et de l'organisation des noeuds
- Certaines propriétés doivent être respectées pour que les arbres puissent constituer un dictionnaire efficace
- On verra ça un peu plus tard...



Programmation ?

Java

- Piles
 - Classe Stack (legacy)
- Piles et Files
 - Interface Deque (ArrayDeque, LinkedList)
 - addFirst(..), addLast(..), removeFirst(), removeLast()
- Listes
 - Interface List (ArrayList, LinkedList)
- Tas/Files de priorité
 - Classe PriorityQueue

C++

- Piles
 - Template stack (containers : vector, deque, list)
 - push(..) (push_back), pop() (pop_back), ...
- Files
 - Template queue (containers : deque, list)
 - push(..) (push_back), pop() (pop_front), ...
- Listes
 - Template list (container : doubly linked list)
- Tas/Files de priorité
 - Template priority_queue (cont. : vector, deque)



PROCHAIN COURS

STRUCTURES DE DONNÉES
DYNAMIQUES « UN PEU PLUS
AVANCÉES »

TABLES DE HACHAGE, ARBRES BINAIRES DE
RECHERCHE