

Révision Final

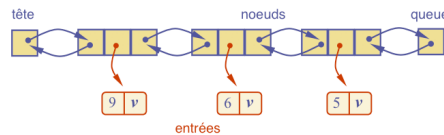
1) Dictionnaires

- Un dictionnaire est une structure de données gardant en mémoire des entrées de la forme (clé, valeur)
- Permet l'insertion, la suppression et la recherche d'un élément de clé k
- Deux types
 - Dictionnaires non-ordonnés → Testeur d'égalités pour les clés
 - Dictionnaires ordonnés → Comparateur pour les clés
 - Nouvelles opérations: **FindAll(k)**
RemoveAll(k)
- On appelle "maps" un dictionnaire dans lequel l'insertion de deux éléments de même clé est interdite
- Implémentations:
 - Dictionnaires non-ordonnés → Listes chaînées, tableaux, tables de hachage
 - Dictionnaires ordonnés → Arbres de recherche

2) Implémentations Dictionnaires non-ordonnés

● Listes doublement chaînées:

- Complexité en temps:
 - ▲ insérer $O(1)$ (pour les "maps" $O(n)$)
 - ▲ enlever $O(n)$
 - ▲ trouver $O(n)$
- Complexité en espace: $O(n)$



● Tableau:

- Complexité en temps:
 - ▲ insérer $O(1)$ (pour les "maps" $O(n)$)
 - ▲ enlever $O(1)$
 - ▲ trouver $O(1)$
- Complexité en espace: $O(N)$

0	1	2	3	4	5	6	7	8	9	10	11
(0,v)	(1,v)	No Key	No Key	(4,v)	No Key	No Key	No Key	No Key	No Key	No Key	(11,v)

● Problème:

- Permet de travailler avec les clés entières seulement
- Collisions
- $N \gg \gg n$

2) Implémentations Dictionnaires non-ordonnés (suite)

● Tables de hachage:

- Un tableau de taille N
- Une **fonction de hachage h**: $h(x) = h_2(h_1(x))$
 - ▲ Code de hachage h_1 : clé → entiers
 - ▲ Fonction de compression h_2 : entiers → $[0, N - 1]$
- On insère un élément de clé k dans la cellule $h(k)$

● Codes de hachage:

- On re-interprète la représentation en bits de la clé par un entier
- Addition de composantes: Si les clés ont une représentation de plus de 32 bits, on partitionne cette représentation en composantes de 32 bits et on fait la somme des entiers correspondants
- Code de hachage polynomial:

▲ On partitionne les bits représentant la clé en une séquence de composantes de même longueur (8,16 ou 32 bits): $a_0 a_1 \dots a_{n-1}$

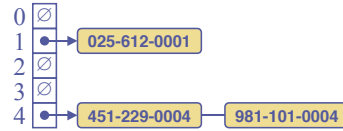
▲ On évalue le polynôme $p(z) = a_0 + a_1 z + \dots + a_{n-1} z^{n-1}$ pour un $z \neq 0$

● Fonction de compression: $h_2(y) = y \bmod N$

3) Résolution de collisions (table de hachage)

● Méthode par chaînage:

- Chaque cellule i de la table pointe vers une liste chaînée contenant les éléments de clé k , telle que $h(k)=i$
- Simple mais requière de la mémoire externe
- On appelle facteur de chargement d'un tableau $\lceil n/N \rceil$ i.e le nombre d'éléments divisé par le nombre de cellules
- Les opérations d'insertions, de recherches et de suppressions se font donc en temps $\mathcal{O}(\lceil n/N \rceil)$
- Donc, si n est $\mathcal{O}(N)$, les opérations se font en temps constant



3) Résolution de collisions (suite)

● hachage ouvert:

- ▲ **sondage linéaire**: place l'entrée en collision dans la prochaine cellule disponible (table circulaire), i.e qu'on a les fonctions de hachage $h_i(k) = h(k) + i \pmod N$ qu'on essaie pour $i = 0, \dots, N-1$, jusqu'à ce qu'on trouve une cellule libre



- Un des problèmes du sondage linéaire est qu'il crée souvent des amoncellements de clés

- ▲ **hachage quadratique**: place l'entrée en collision dans la prochaine cellule disponible selon les fonctions de hachage $h_i(k) = k + i^2 \pmod N$

Avantage: Évite les amoncellements de clés qu'on avait avec le sondage linéaire

Inconvénient: Quelque fois les fonctions sont incapables de trouver une cellule vide même quand le tableau n'est pas plein

- ▲ **hachage aléatoire**: place l'entrée en collision dans la prochaine cellule disponible selon les fonctions de hachage $h_i(k) = k + d_i \pmod N$, où d est une permutation de $[1, N-1]$

3) Résolution de collisions (suite)

- ▲ **hachage double**: utilise une fonction de hachage secondaire $d(k)$ pour résoudre les collisions, en plaçant l'entrée dans la première cellule disponible dans la série $(h(k) + jd(k)) \pmod N$ pour $j = 0, 1, \dots, N-1$
 - La fonction de hachage secondaire $d(k)$ ne peut jamais être nulle.
 - Un choix commun pour $d(k)$ est $d(k) = q - k \pmod q$ où q est un nombre premier $< N$
- ▲ **Performance du hachage ouvert**:
 - Dans le pire cas, les opérations d'insertion, de recherche et de suppression se font en $\mathcal{O}(n)$
 - Le pire cas arrive lorsqu'il y a collision à chaque insertion d'une entrée
 - Le facteur de chargement de la table affecte la performance

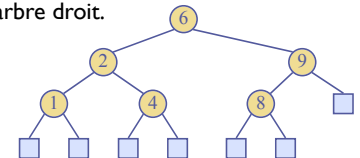
4) Arbre de recherche binaire

- Un arbre binaire de recherche est un arbre binaire qui garde en mémoire des entrées (clé-valeur) dans ses noeuds internes et qui satisfait la propriété suivante:

- Soient u, v et w trois noeuds tels que u est dans le sous-arbre gauche de v et w dans son sous-arbre droit.

Alors, on a

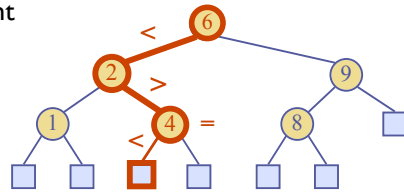
$$\text{clé}(u) < \text{clé}(v) \leq \text{clé}(w)$$



- Les noeuds externes ne gardent en mémoire aucune entrée
- Un parcours symétrique de l'arbre visite les clés en ordre croissant
- **Complexité en temps des opérations principales**:
 - **Chercher(k)** se fait en $\mathcal{O}(h)$ et donc en $\mathcal{O}(n)$ dans le pire des cas
 - **Insérer(k)** se fait en $\mathcal{O}(h)$ et donc en $\mathcal{O}(n)$ dans le pire des cas
 - **Supprimer(k)** se fait en $\mathcal{O}(h)$ et donc en $\mathcal{O}(n)$ dans le pire des cas

Chercher dans un arbre binaire de recherche

- Pour chercher un élément de clé k dans un arbre binaire de recherche, on va suivre un chemin descendant en commençant la recherche à la racine.
- **Exemple 1:** Chercher(4)
- Le prochain noeud visité dépend du résultat de la comparaison de k avec la clé du noeud dans lequel on se trouve.
- Si on trouve un noeud interne de clé k , on retourne la valeur correspondant à cette entrée de clé k



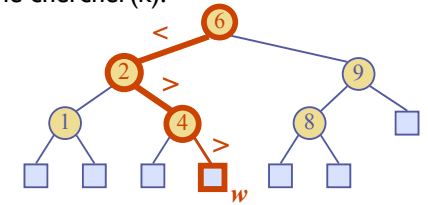
- Sinon, on retourne NULL
 - **Exemple 1:** Chercher(3)

Insérer dans un arbre binaire de recherche

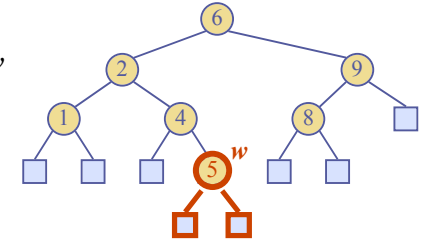
- Pour insérer un élément (k,v) dans un arbre binaire de recherche, on commence par exécuter l'algorithme chercher(k).

- **Exemple 1:** Insérer(5,v)

- Si k n'est pas dans l'arbre l'algorithme chercher(k) se terminera dans une feuille w



- On insère k dans w et on change w en un noeud interne

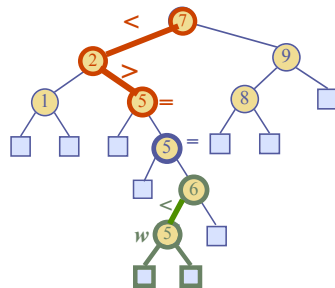


Insérer dans un arbre binaire de recherche

- Si k est dans l'arbre l'algorithme chercher(k) se terminera sur un noeud interne v . On appelle récursivement l'algorithme sur le filsDroit(v), jusqu'à ce qu'on arrive à un noeud externe w

- **Exemple 2:** Insérer(5,v)

- On insère k dans w et on change w en un noeud interne

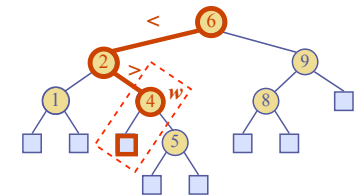


Supprimer dans un arbre binaire de recherche

- Pour enlever un élément de clé k dans un arbre binaire de recherche, on commence par exécuter l'algorithme chercher(k).

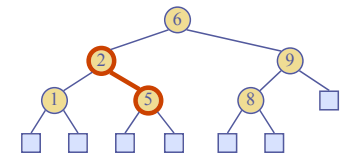
- **Exemple 1:** Enlever(4)

- Si k est dans l'arbre l'algorithme chercher(k) se terminera dans un noeud interne w



- Si l'un des enfant de w est une feuille, on enlève cette feuille et w

- Sinon...

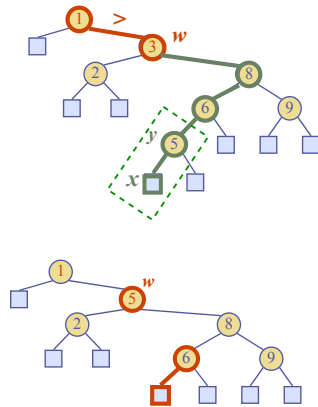


Supprimer dans un arbre binaire de recherche (suite)

● Si k est dans l'arbre, l'algorithme $\text{chercher}(k)$ se terminera dans un noeud interne w . Si les fils de w sont tous les deux des noeuds internes alors

● Exemple 2: Enlever(3)

- On trouve le noeud interne y qui suit w lors d'un parcours symétrique de l'arbre et son fils gauche x
- On enlève l'entrée dans w et on la remplace par l'entrée dans y
- On enlève les noeuds y et x



5) Arbre AVL

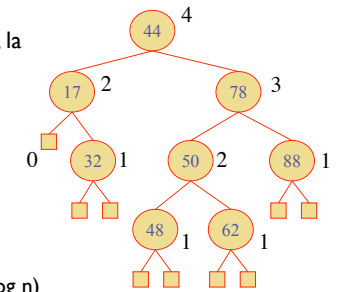
● Un arbre AVL est un arbre binaire de recherche **balancé** selon la propriété suivante:

- **Propriété de balance:** Pour chaque noeud interne v , la hauteur des enfants de v diffère d'au plus 1

● La hauteur d'un arbre AVL est en $O(\log n)$

● Complexité en temps des opérations principales:

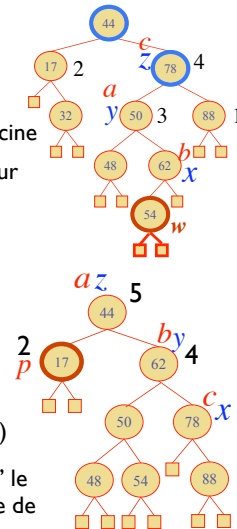
- L'algorithme de recherche prend un temps $O(\log n)$
- L'algorithme d'insertion prend un temps $O(\log n)$
 - Chercher l'endroit où insérer prend un temps $O(\log n)$
 - Trouver un noeud non balancé (si il y en a un) prend un temps $O(\log n)$
 - Restructurer l'arbre prend un temps $O(1)$
- L'algorithme de suppression prend un temps $O(\log n)$
 - Chercher le noeud à supprimer prend un temps $O(\log n)$
 - Trouver un débalecement
 - Restructurer l'arbre prend un temps] → jusqu'à la racine: $O(\log n)$



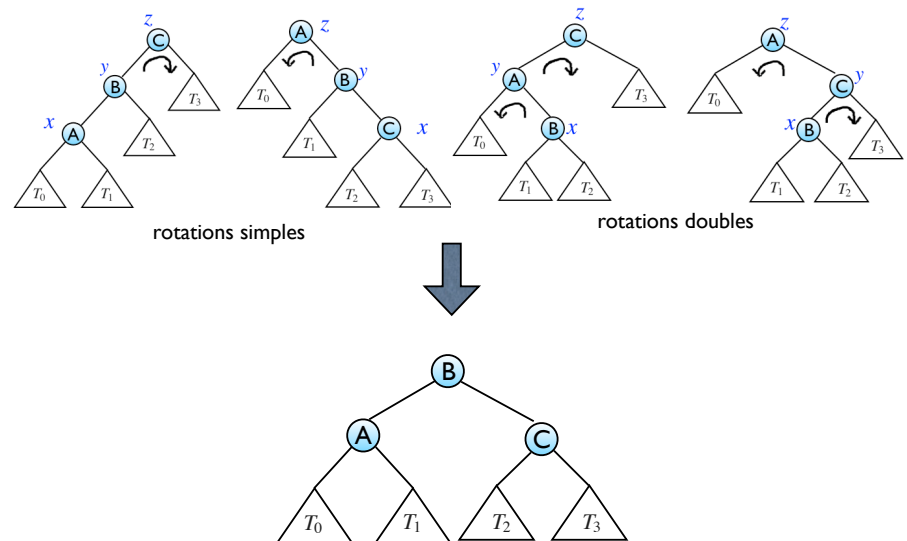
5) Arbre AVL: restructuration

● Pour retrouver la propriété de balance, on va devoir restructurer l'arbre comme suit:

- Dans le cas d'une insertion dans un noeud w , les noeuds débalecés sont situés sur le chemin allant du noeud w à la racine
- Dans le cas d'une suppression, le noeud débalecés est situé sur le chemin allant du parent du noeud supprimé à la racine
- On va nommer z le premier noeud débalecés qu'on trouve sur l'un ou l'autre de ces chemin.
- On va nommer y le fils de z de plus grande hauteur
- On va nommer x le fils de y de plus grande hauteur (si égalité, on choisit le fils qui est un ancêtre de w dans le cas de l'insertion et n'importe quel fils dans le cas de la suppression)
- Étant donné les noeuds internes x, y et z , on renomme par "a" le premier de ces sommets visités lors d'un parcours symétrique de l'arbre, par "b", le deuxième et par "c", le troisième.



5) Arbre AVL: restructuration: 4 cas possibles

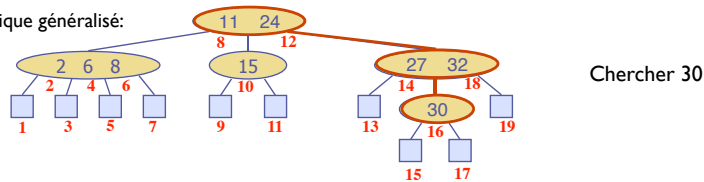


6) Arbre de recherche généralisé

● Un arbre de recherche généralisé est un arbre ordonné ayant les propriétés suivantes:

- Chaque noeud interne a au moins deux enfants et garde en mémoire $d-1$ éléments (k_i, v_i) , où d est le nombre d'enfants
- Pour chaque noeud interne gardant en mémoire les clés k_1, k_2, \dots, k_{d-1} et ayant pour enfants les noeuds n_1, n_2, \dots, n_d on a
 - ▲ les clés dans le sous-arbre de racine n_1 sont plus petites que k_1
 - ▲ les clés dans le sous-arbre de racine n_i sont plus petites que k_i et plus grande ou égale à k_{i-1} ($i = 2, \dots, d-1$)
 - ▲ les clés dans le sous-arbre de racine n_d sont plus grandes ou égales à k_{d-1}

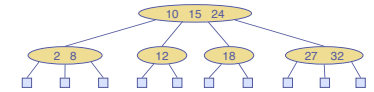
parcours symétrique généralisé:



7) Arbre (2,4)

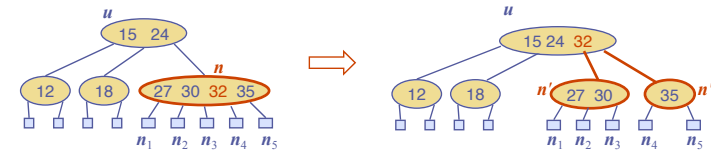
● Un arbre (2,4) est un arbre de recherche généralisé ayant les propriétés suivantes:

- **Nombre d'enfants:** tout noeud interne a au plus 4 enfants
- **Propriété de profondeur:** tous les noeuds externes ont la même profondeur



● Complexité en temps des opérations principales:

- L'algorithme de recherche prend un temps $O(\log n)$
- L'algorithme d'insertion prend un temps $O(\log n)$
 - Chercher l'endroit où insérer en temps $O(\log n)$
 - L'insertion peut causer un débordement
 - On exécute un fractionnement qui peut entraîner la propagation du débordement



Arbre (2,4) (suite)

● Complexité en temps des opérations principales (suite):

- L'algorithme de suppression prend un temps $O(\log n)$
 - Chercher le noeud dans lequel on va supprimer une clé prend un temps $O(\log n)$
 - La suppression de la clé peut vider le noeud
 - Si l'un des frères est un 3 ou 4 -noeud, on exécute une opération de transfert
 - Après l'opération de transfert, l'arbre (2,4) est bien structuré



- Sinon, on exécute une opération de fusion
- Après l'opération de fusion, il est possible que le parent du noeud fusionné soit vide

