Partie 5

Dictionnaires

Plan

1. Introduction
2. Arbres binaires de recherche
3. Tables de hachage

Dictionnaires

- Définition : un dictionnaire est un ensemble dynamique d'objets avec des clés comparables qui supportent les opérations suivantes :
  - $\textsc{Search}(S, k)$ retourne un pointeur x vers un élément dans S tel que $x.key = k$, ou NIL si un tel élément n'appartient pas à S.
  - $\textsc{Insert}(S, x)$ insère l'élément x dans le dictionnaire S. Si un élément de même clé se trouve déjà dans le dictionnaire, on met à jour sa valeur
  - $\textsc{Delete}(S, x)$ retire l'élément x de S. Ne fait rien si l'élément n'est pas dans le dictionnaire.
- Pour faciliter la recherche, on peut supposer qu'il existe un ordre total sur les clés.

Dictionnaires

- Deux objectifs en général :
  - minimiser le coût pour l'insertion et l'accès au données
  - minimiser l'espace mémoire pour le stockage des données
- Exemples d'applications :
  - Table de symboles dans un compilateur
  - Table de routage d'un DNS
  - ...
- Beaucoup d'implémentations possibles

Liste liée

Première solution :

- On stocke les paires clé-valeur dans une liste liée
- Recherche :

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL ∧ x.key ≠ k
3        x = x.next
4  return x
```

- Insertion (resp. Suppression)
  - On recherche la clé dans la liste
  - Si elle existe, on remplace la valeur (resp. on la supprime)
  - Si elle n'existe pas, on la place en début de liste (resp. on ne fait rien)
- Complexité au pire cas *(meilleur cas ?)*
  - Insertion : $\Theta(N)$
  - Recherche : $\Theta(N)$
  - Suppression : $\Theta(N)$

Vecteur trié

Deuxième solution :

- On suppose qu'il existe un ordre total sur les clés
- On stocke les éléments dans un vecteur qu'on maintient trié
- Recherche dichotomique (approche "diviser-pour-régner")

```
BINARY-SEARCH(V, k, low, high)
1  if low > high
2        return NIL
3  mid = ⌊(low + high)/2⌋
4  x = ELEM-AT-RANK(V, mid)
5  if k == x.key
6        return x
7  elseif k > x.key
8        return BINARY-SEARCH(V, k, mid + 1, high)
9  else return BINARY-SEARCH(V, k, low, mid − 1)
```

Complexité au pire cas : $\Theta(\log n)$

Vecteur trié

- Insertion : recherche de la position par BINARY-SEARCH puis insertion dans le vecteur par INSERT-AT-RANK (=décalage des éléments vers la droite).
- Suppression : recherche puis suppression par REMOVE-AT-RANK (=décalage des éléments vers la gauche).
- Complexité au pire cas *(meilleur cas ?)*
  - Insertion : $\Theta(N)$ (on doit décaler les éléments à droite de la clé)
  - Recherche : $\Theta(\log N)$ (recherche dichotomique)
  - Suppression : $\Theta(N)$ (on doit décaler les éléments à gauche de la clé)

  (Si le vecteur est implémenté par un tableau extensible !)

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Peut-on obtenir à la fois une insertion et une recherche "efficaces" ?

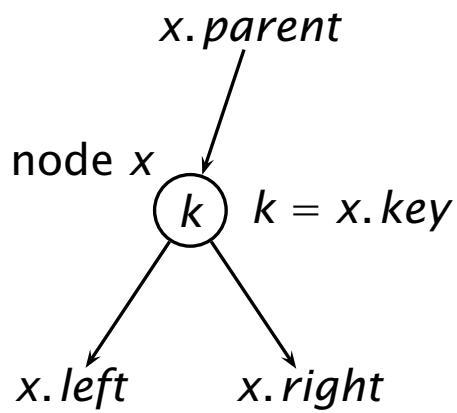
Plan

Plan

Implémentation des arbres binaires

Implémentation des arbres binaires dans ce chapitre (pour simplifier les algorithmes à venir) :

- T représente l'arbre, qui consiste en un ensemble de nœuds
- $T.root$ est le nœud racine de l'arbre T
- Nœud x
  - $x.parent$ est le parent du nœud x
  - $x.key$ est la clé stockée au nœud x
  - $x.left$ est le fils de gauche du nœud x
  - $x.right$ est le fils de droite du nœud x

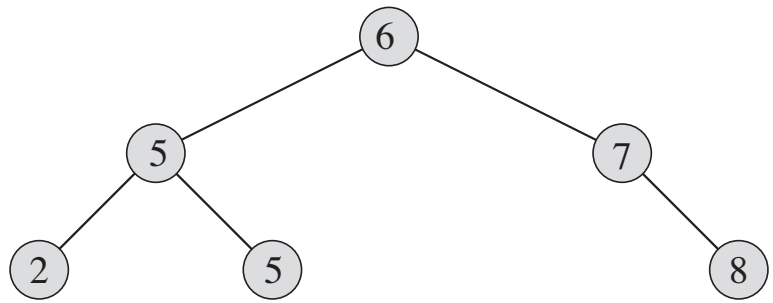


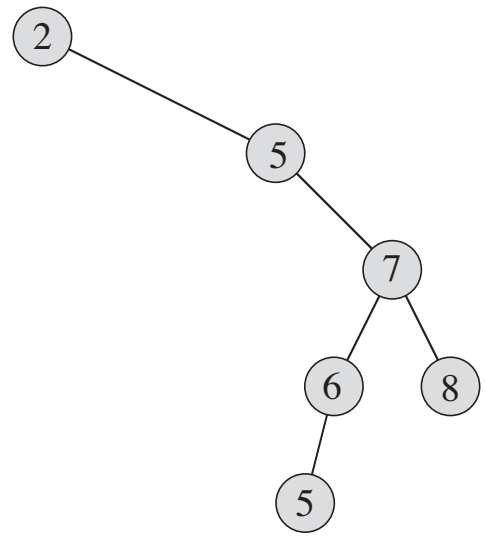
Pour simplifier les notations, nos fonctions seront implémentées directement sur base de cette implémentation (et pas de l'interface générale vue précédemment)

Arbres binaires de recherche

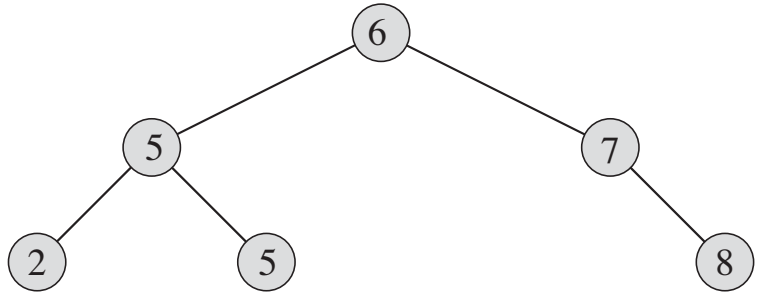
- Une structure d'arbre binaire implémentant un dictionnaire, avec des opérations en $O(h)$ où h est la hauteur de l'arbre

- Chaque nœud de l'arbre binaire est associé à une clé
- L'arbre satisfait à la propriété d'arbre binaire de recherche
  - Soient deux nœuds x et y.
  - Si y est dans le sous-arbre de gauche de x, alors $y.key < x.key$
  - Si y est dans le sous-arbre de droite de x, alors $y.key \geq x.key$





Parcours d'un arbre binaire de recherche



$$\Rightarrow \langle 2, 5, 5, 6, 7, 8 \rangle$$

- Le parcours infixe d'un arbre binaire de recherche permet d'afficher les clés par ordre croissant

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2        INORDER-TREE-WALK(x.left)
3        print x.key
4        INORDER-TREE-WALK(x.right)
```

Recherche dans un arbre binaire

- Recherche binaire

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Appel initial (à partir d'un arbre T)

```
TREE-SEARCH(T.root, k)
```

- Complexité ? $T(n) \in O(h)$, où h est la hauteur de l'arbre
- Pire cas : $h = n$

Recherche dans un arbre binaire

- TREE-SEARCH est récursive terminale.
- Version itérative

```
ITERATIVE-TREE-SEARCH(T, k)
1  x = T.root
2  while x ≠ NIL and k ≠ x.key
3        if k < x.key
4              x = x.left
5        else x = x.right
6  return x
```

Clés maximale et minimale

- Etant donné la propriété d'arbre binaire
  - La clé minimale se trouve dans le nœud le plus à gauche
  - La clé maximale se trouve dans le noœud le plus à droite

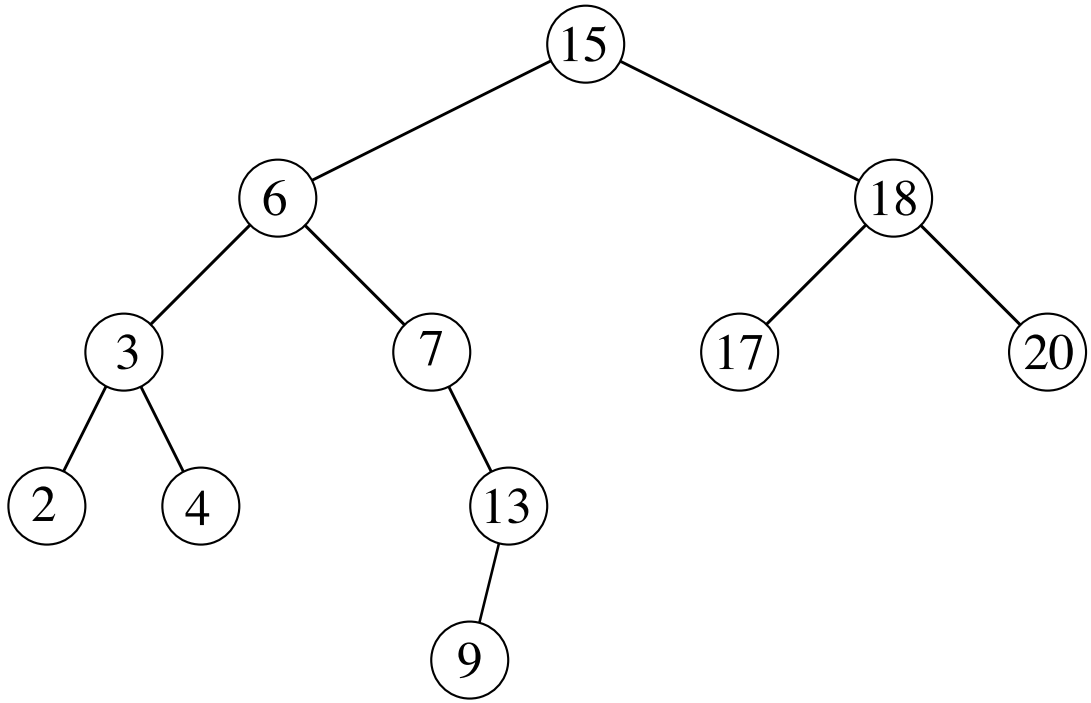
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2        x = x.right
3  return x
```

- Complexité : $O(h)$, où h est la hauteur de l'arbre.

Successeur et prédécesseur

- Etant donné un nœud x, trouver le nœud contenant la valeur de clé suivante (dans l'ordre)

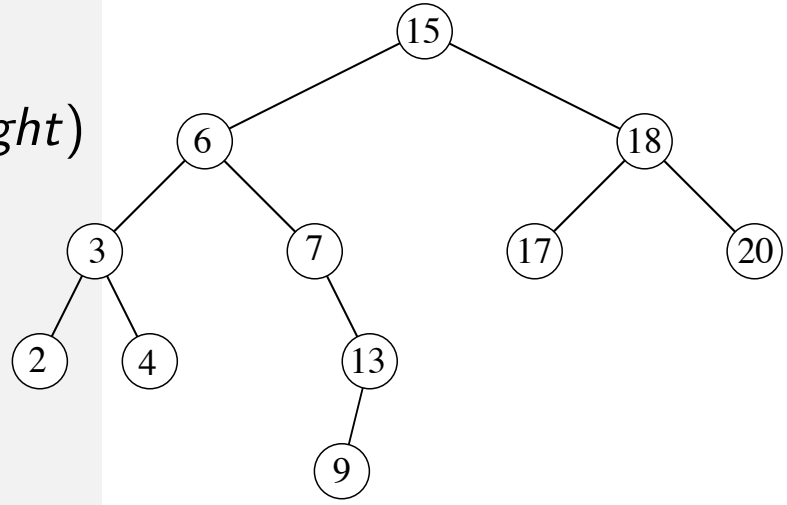


Ex : successeur de 15 $\rightarrow$ 17, successeur de 4 $\rightarrow$ 6.

- Le successeur de x est le minimum du sous-arbre de droite s'il existe
- Sinon, c'est le premier ancêtre a de x tel que x tombe dans le sous-arbre de gauche de a.

Successeur et prédécesseur

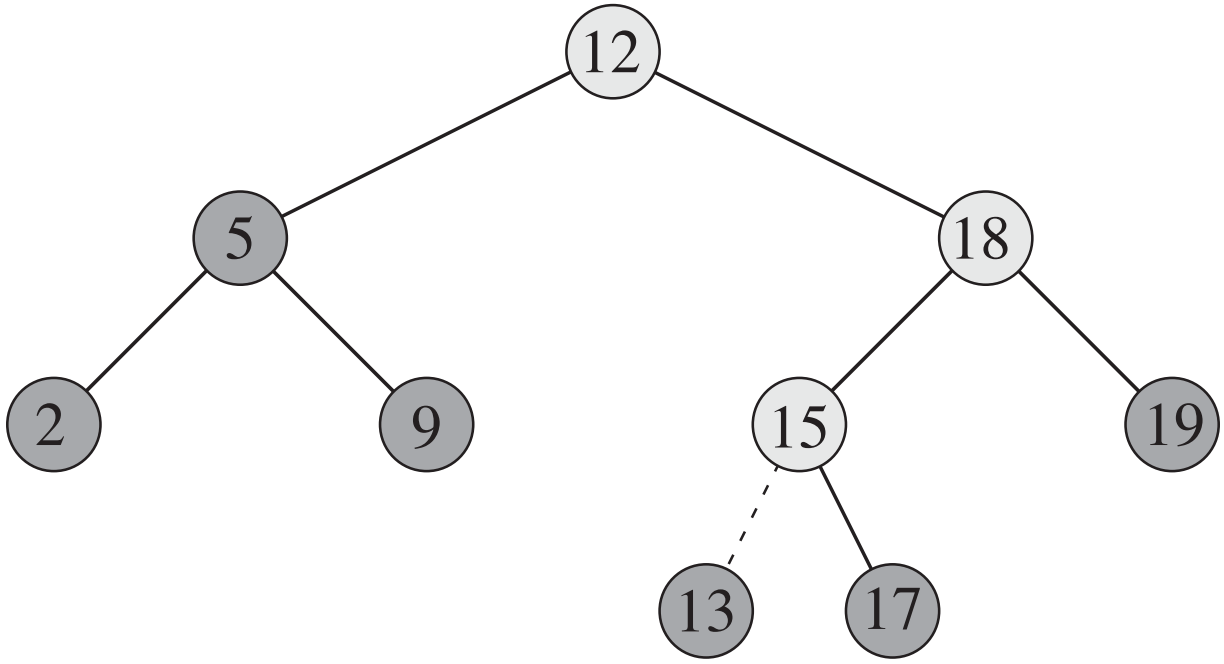
```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.parent
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.parent
7  return y
```



Complexité : $O(h)$, où h est la hauteur de l'arbre

(Exercice : TREE-PREDECESSOR*)*

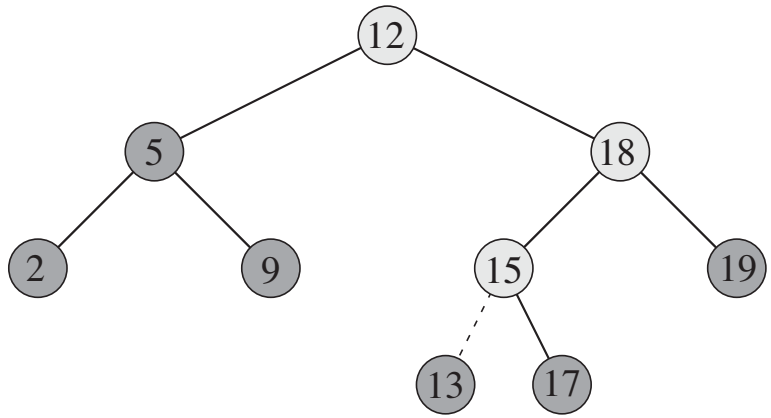
Insertion



- Pour insérer x, on recherche la clé $x.key$ dans l'arbre
- Si on ne la trouve pas, on l'ajoute à l'endroit où la recherche s'est arrêtée.

Insertion

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y == NIL
10        // Tree T was empty
11        T.root = z
12  elseif z.key < y.key
13        y.left = z
14  else y.right = z
```

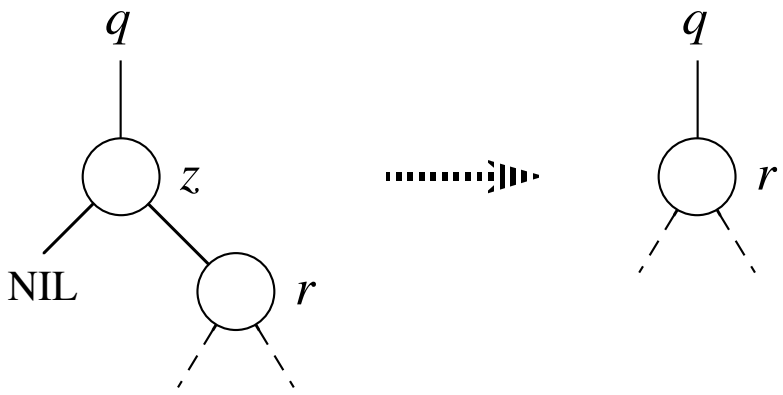


Complexité : $O(h)$ où h est la hauteur de l'arbre

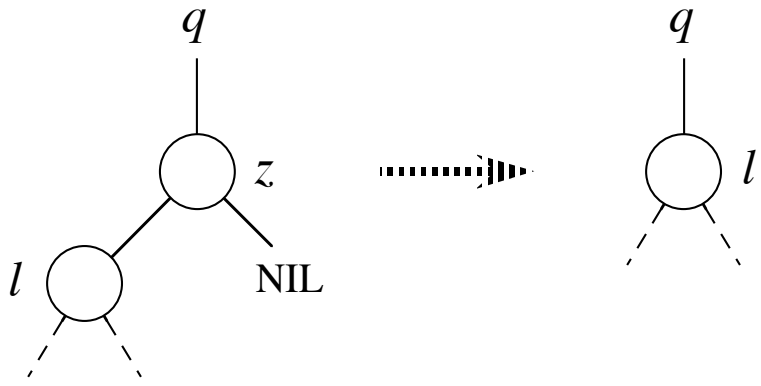
Suppression

3 cas à considérer en fonction du nœud z à supprimer :

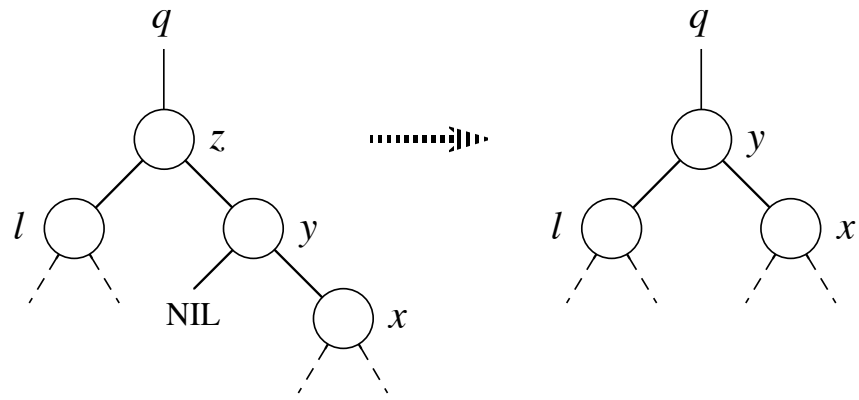
- z n'a pas de fils gauche : remplacer z par son fils droit



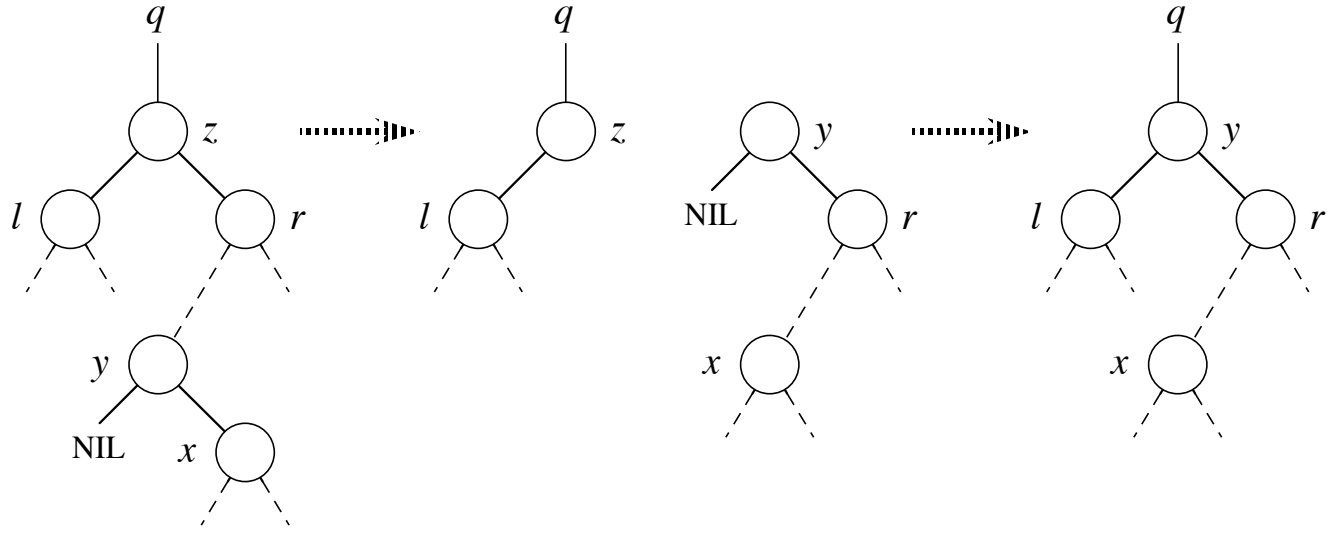
- z n'a pas de fils droit : remplacer z par son fils gauche



- z a deux fils : rechercher le successeur y de z.
  NB : y est dans le sous-arbre de droite et n'a pas de fils gauche.
  - Si y est le fils droit de z, remplacer z par y et conserver le fils droit de y



  - Sinon, y est dans le sous-arbre droit de z mais n'en est pas la racine. On remplace y par son propre fils droit et on remplace z par y.



Suppression

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSPLANT(T, z, z.left)
 5  else // z has two children
 6        y = TREE-SUCCESSOR(z)
 7        if y.parent ≠ z
 8              TRANSPLANT(T, y, y.right)
 9              y.right = z.right
10              y.right.parent = y
11        // Replace z by y
12        TRANSPLANT(T, z, y)
13        y.left = z.left
14        y.left.parent = y
```

```
TRANSPLANT(T, u, v)
1  if u.parent == NIL
2        T.root = v
3  elseif u == u.parent.left
4        u.parent.left = v
5  else u.parent.right = v
6  if v ≠ NIL
7        v.parent = u.parent
```

Complexité : $O(h)$ pour un arbre de hauteur h
(Tout est $O(1)$ sauf l'appel à TREE-SUCCESSOR).

Arbres binaires de recherche

- Toutes les opérations sont $O(h)$ où h est la hauteur de l'arbre
- Si n éléments ont été insérés dans l'arbre :
  - Au pire, $h = n - 1 \in \Theta(n)$
    - Elements insérés en ordre
  - Au mieux, $h = \lceil \log_2 n \rceil \in \Theta(\log n)$
    - Pour un arbre binaire complet
  - En moyenne, on peut montrer que $h \in \Theta(\log n)$
    - En supposant que les éléments ont été insérés en ordre aléatoire

Profondeur moyenne d'un nœud

- Montrons que la profondeur moyenne d'un nœud dans un ABR construit aléatoirement est $\Theta(\log n)$
- Soit $P(n)$ la somme totale moyenne des profondeurs des nœuds d'un ABR construit à partir de n clés introduites dans un ordre aléatoire. La profondeur moyenne recherchée est $P(n)/n$.
- On a :

$$P(n) = \sum_{i=0}^{n-1} \frac{1}{n}(P(i) + P(n-i-1) + n - 1), P(1) = 0$$

  En effet :

  - La première clé introduite dans l'ABR a autant de chance d'être à chaque position de la liste triée des clés.
  - Si elle est à la position $i+1$ le sous-arbre de gauche contient i clés et celui de droite $n-i-1$ clés.
  - La profondeur de chaque nœud à droite et à gauche de la racine est augmentée de 1 par rapport à sa profondeur dans les sous-arbres à gauche et à droite.

Profondeur moyenne d'un nœud

- Cette récurrence peut se réécrire en :

$$P(n) = \sum_{i=0}^{n-1} \frac{1}{n}(P(i) + P(n-i-1)) + n - 1.$$

  qui est identique à celle du QUICKSORT (voir le transp. 179)
- On en déduit :

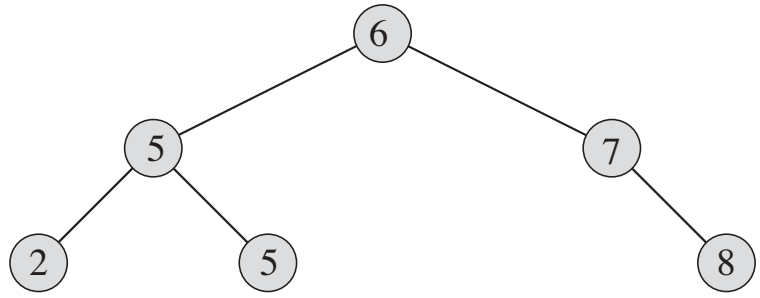
$$\Rightarrow \quad P(n) \in \Theta(n \log n)$$
$$\Rightarrow \quad \frac{P(n)}{n} \in \Theta(\log n)$$

- En moyenne, la recherche d'une clé parmi celles stockées dans un ABR obtenu après insertion des clés dans un ordre aléatoire est donc $\Theta(\log n)$

Tri avec un arbre binaire de recherche

```
BINARY-SEARCH-TREE-SORT(A)
1  T = "Empty binary search tree"
2  for i = 1 to n
3        TREE-INSERT(T, A[i])
4  INORDER-TREE-WALK(T.root)
```

- Exemple : $A = [6, 5, 7, 2, 5, 8]$



- Complexité en temps identique au quicksort
  - Insertion : en moyenne, $n \cdot O(\log n) = O(n \log n)$, pire cas : $\Theta(n^2)$
  - Parcours de l'arbre en ordre : $\Theta(n)$
  - Total : $\Theta(n \log n)$ en moyenne, $\Theta(n^2)$ pour le pire cas
- Complexité en espace cependant plus importante, pour le stockage de la structure d'arbres.

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Peut-on obtenir $\Theta(\log n)$ dans le pire cas ? Oui !
- Deux solutions :
  - Utiliser de la randomisation pour que le probabilité de rencontrer le pire cas soit négligeable
  - Maintenir les arbres équilibrés

Plan

Arbres équilibrés

- Solution générale pour obtenir une complexité au pire cas en $\Theta(\log n)$:
  - Définir un invariant sur la structure d'arbre
  - Prouver que cet invariant garantit une hauteur $\Theta(\log n)$
  - Implémenter les opérations d'insertion et suppression de manière à maintenir l'invariant
  - Si ces opérations ne sont pas trop coûteuses (p.ex., $O(\log n)$), on aura gagné

- Plusieurs types d'arbres équilibrés :
  - Arbres AVL
    - Invariant : Arbres H-équilibrés
  - Arbres rouges et noirs (voir INFO0027)
  - Arbres 2-3-4
  - Splay trees, Scapegoat trees, treaps...

Arbres H-équilibrés

- Définition :

$$T \text{ est } H-\text{équilibré} \Leftrightarrow |h(g(T')) - h(d(T'))| \leq 1,$$

pour tout sous-arbre T' de T, et où $g(X)$, $d(X)$ et $h(X)$ sont resp. le sous-arbre gauche, le sous-arbre droit et la hauteur [3] de l'arbre X.

(Les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un)

- Propriété :
Pour tout arbre H-équilibré de taille n et de hauteur h, on a

$$h \in \Theta(\log n)$$

Plus précisément, on peut prouver :

$$\log(n+1) \leq h+1 < 1,44 \log(n+2)$$

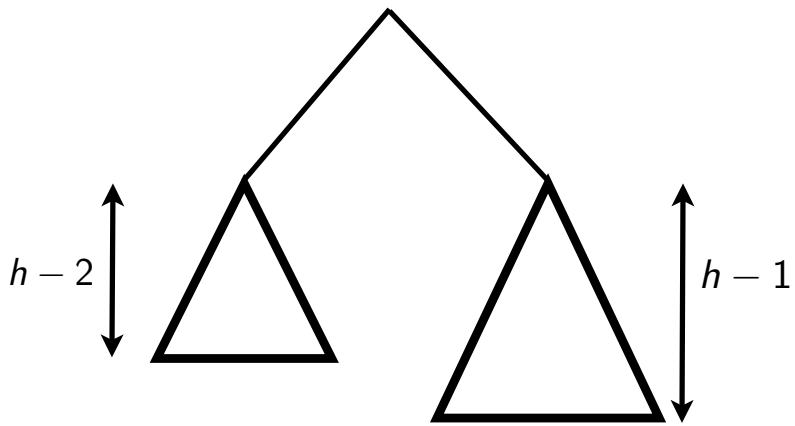
3. Par définition, la hauteur d'un arbre vide est -1.

Arbres H-équilibrés

Démonstration

Etant donné un arbre H-équilibré de taille n et de hauteur $h \geq 1$, pour h fixé, n est

- Maximum : quand l'arbre est complet, soit quand $n = 2^{h+1} - 1 \Rightarrow n + 1 \leq 2^{h+1} \Rightarrow \log(n+1) \leq h + 1 \Rightarrow h \in \Omega(\log n)$
- Minimum : quand $n = N(h)$ où $N(h)$ est la taille d'un arbre H-équilibré de hauteur h qui a le moins d'éléments.
  - $N(h)$ peut être défini par récurrence par $N(h) = 1 + N(h-1) + N(h-2)$ avec $N(0) = 1$ et $N(1) = 2$.



- On a donc
$$\begin{aligned} & N(h) = 1 + N(h-1) + N(h-2) \\ \Rightarrow \quad & N(h) > 2N(h-2) \text{ (car } N(h-1) > N(h-2)) \\ \Rightarrow \quad & N(h) > 2^{h/2} \\ \Rightarrow \quad & h < 2 \log N(h) \end{aligned}$$
- dont on peut tirer que $h \in O(\log n)$

- On en déduit que
$$h = \Theta(\log n)$$

□

Borne supérieure plus précise

- En notant $F(h) = N(h) + 1$, on a $F(h) = F(h-1) + F(h-2)$ avec $F(0) = 2$, $F(1) = 3$
- F est un récurrence de Fibonacci qui a pour solution

$$F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3}) \text{ avec } \phi = \frac{1+\sqrt{5}}{2} \text{ et } \phi' = \frac{1-\sqrt{5}}{2}$$

- On a

$$N(h) + 1 = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3})$$

- ce qui donne

$$n + 1 \geq \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3}) > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1)$$

(car $|\phi'| < 1$)

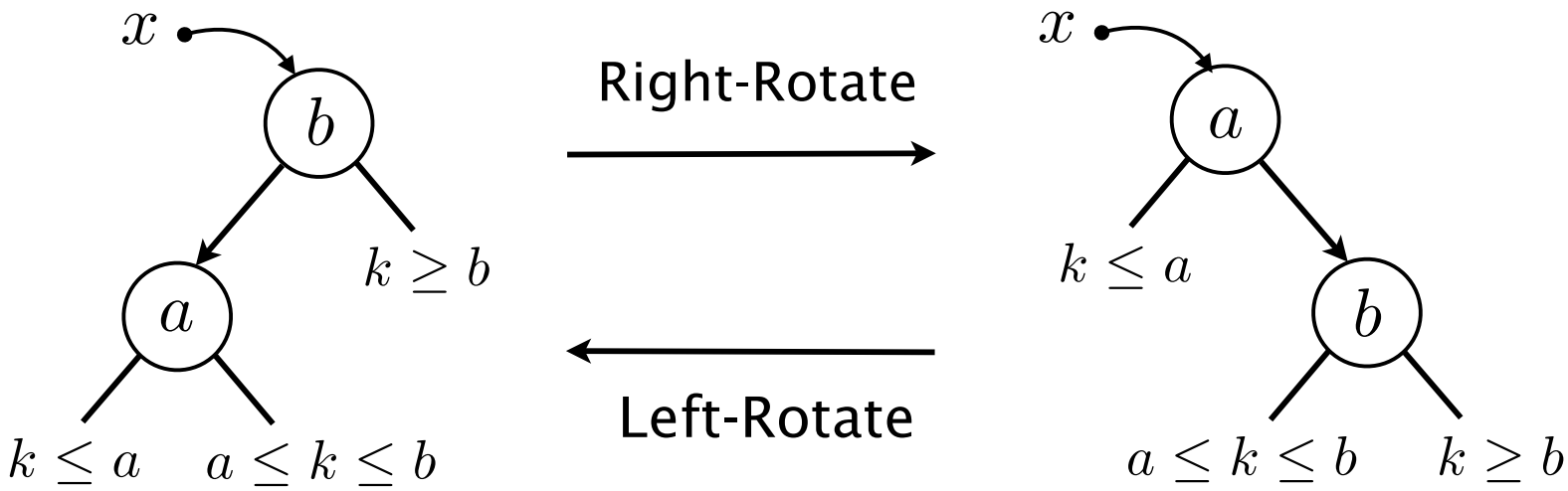
- En prenant le $\log_\phi$ des deux membres :

$$h + 1 < 1,44 \log(n+2)$$

Arbres AVL

- Définition : Un arbre AVL est un arbre binaire de recherche H-équilibré
- Inventé par Adelson-Velskii et Landis en 1960
- Recherche :
  - Par la fonction TREE-SEARCH puisque c'est un arbre binaire
  - Complexité $\Theta(\log n)$ étant donné la propriété
- Insertion :
  - On insère l'élément comme dans un arbre binaire classique
  - On vérifie que l'invariant est respecté
  - Si ce n'est pas le cas, on modifie l'arbre

Rotations

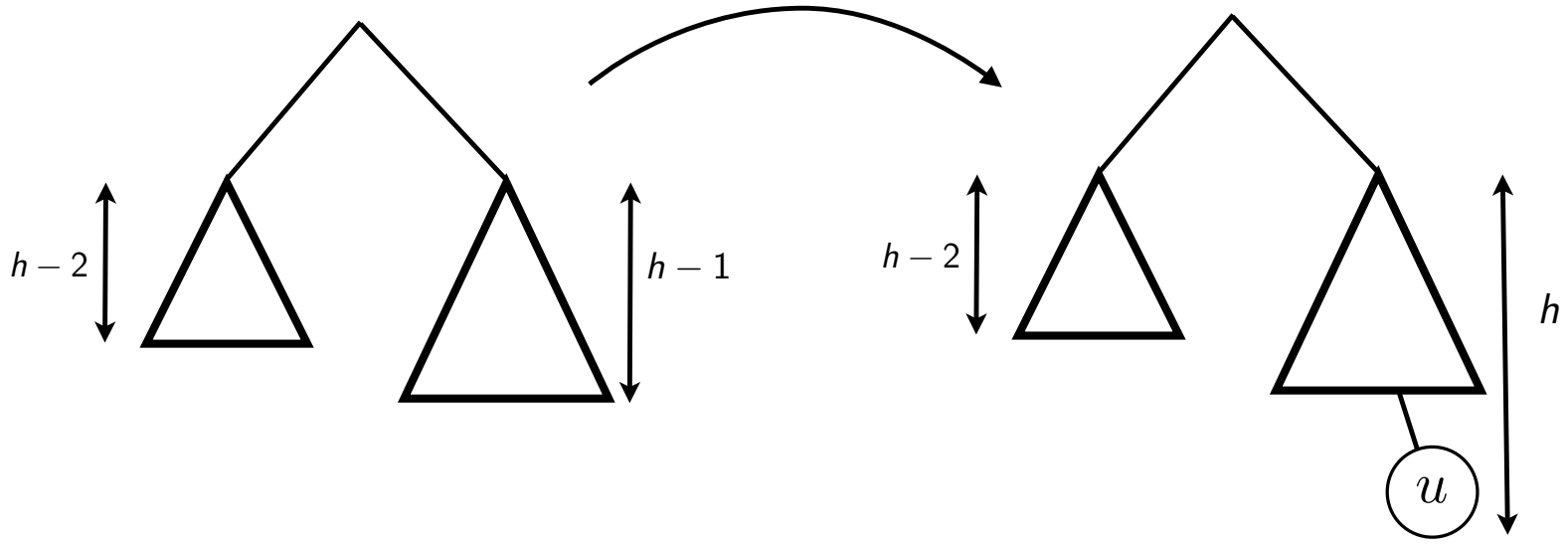



```
LEFT-ROTATE(x)
1  r = x.right
2  x.right = r.left
3  r.left = x
4  return r
```

```
RIGHT-ROTATE(x)
1  l = x.left
2  x.left = l.right
3  l.right = x
4  return l
```

Les rotations maintiennent la propriété d'arbre binaire de recherche

Insertion dans un AVL

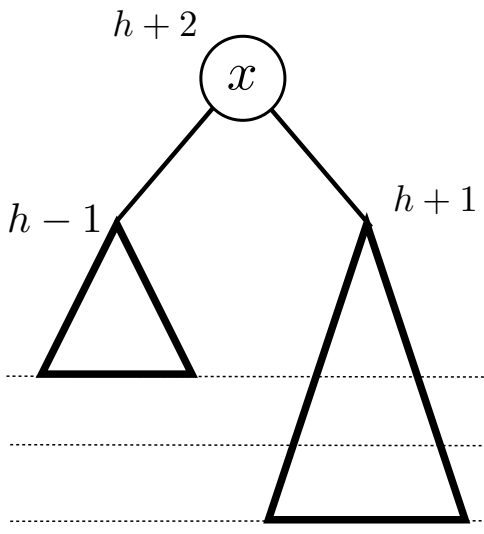


- Insérer le nouvel élément comme dans un arbre binaire de recherche ordinaire
- L'insertion peut créer un déséquilibre (l'arbre n'est plus H-équilibré)
- Remonter depuis le nouveau nœud jusqu'à la racine en restaurant l'équilibre des sous-arbres rencontrés si nécessaire

Equilibrage

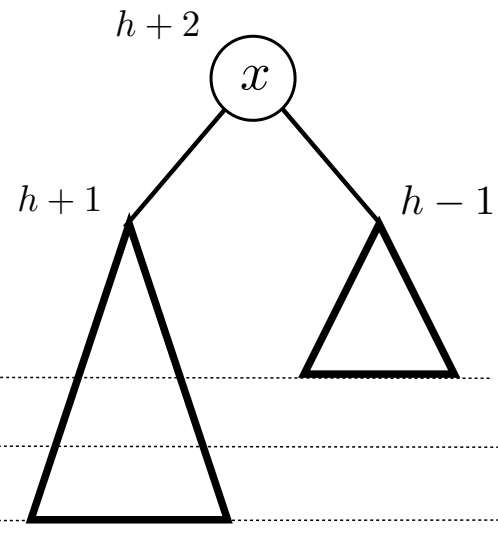
- Soit x le nœud le plus bas violant l'invariant après l'insertion
  - Tous ses sous-arbres sont H-équilibrés
  - Il y a une différence d'au plus 2 niveaux entre ses sous-arbres gauche et droit
- Comment rétablir l'équilibre ?
- Deux cas possibles (selon insertion à droite ou à gauche) :

Cas 1



(Déséquilibre à droite)

Cas 2

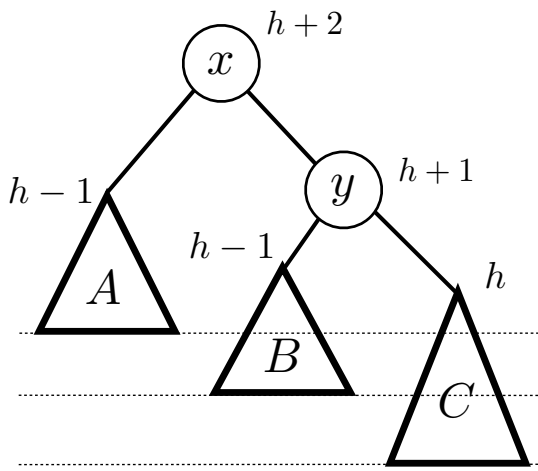


(Déséquilibre à gauche)

Cas 1 : déséquilibre à droite

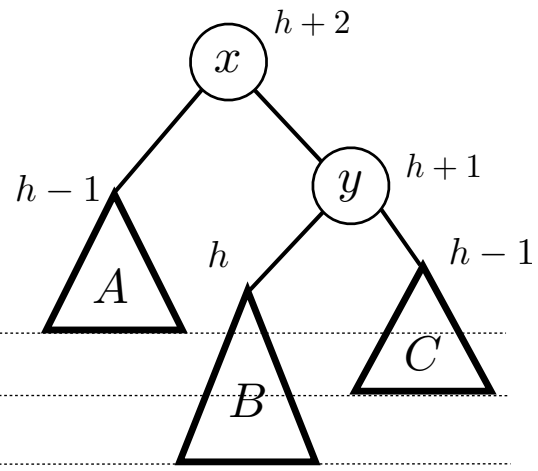
- Deux sous-cas possibles

Cas 1.1



Déséquilibre à l'extérieur
(Cas droite-droite)

Cas 1.2

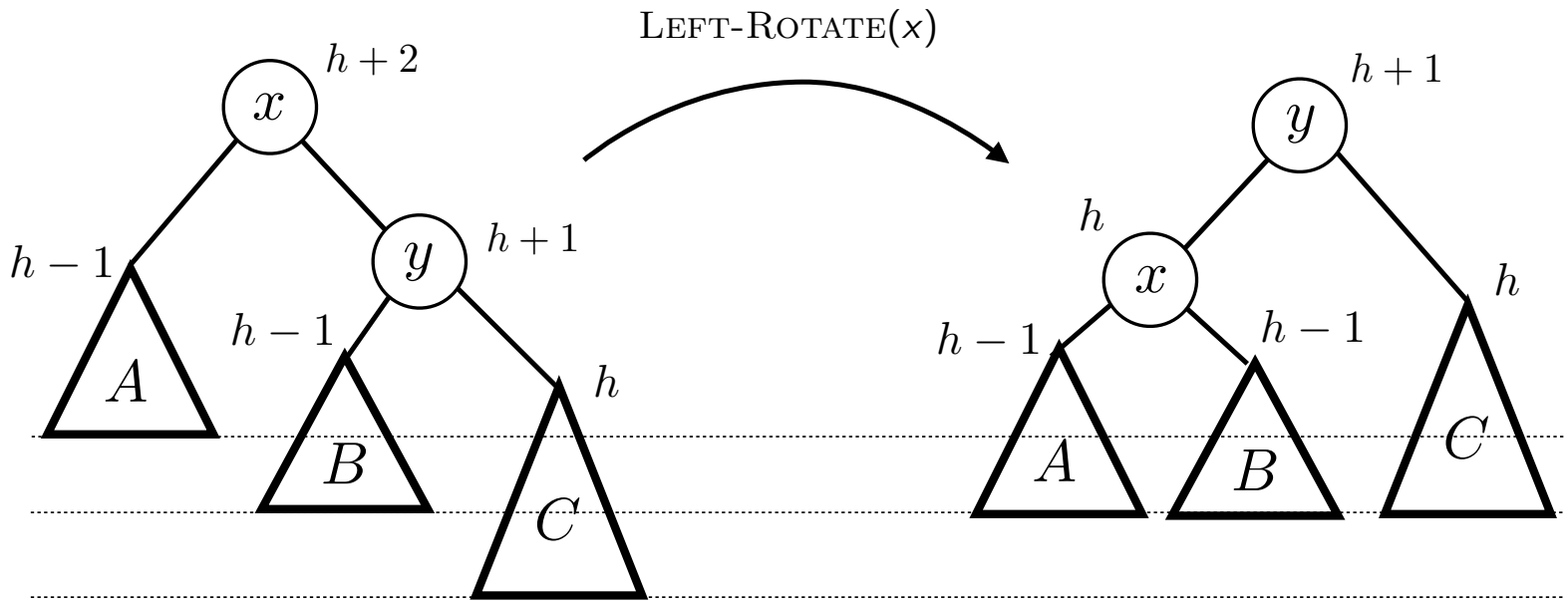


Déséquilibre à l'intérieur
(Cas droite-gauche)

(Pourquoi le cas B et C de hauteur h n'est pas possible ?)

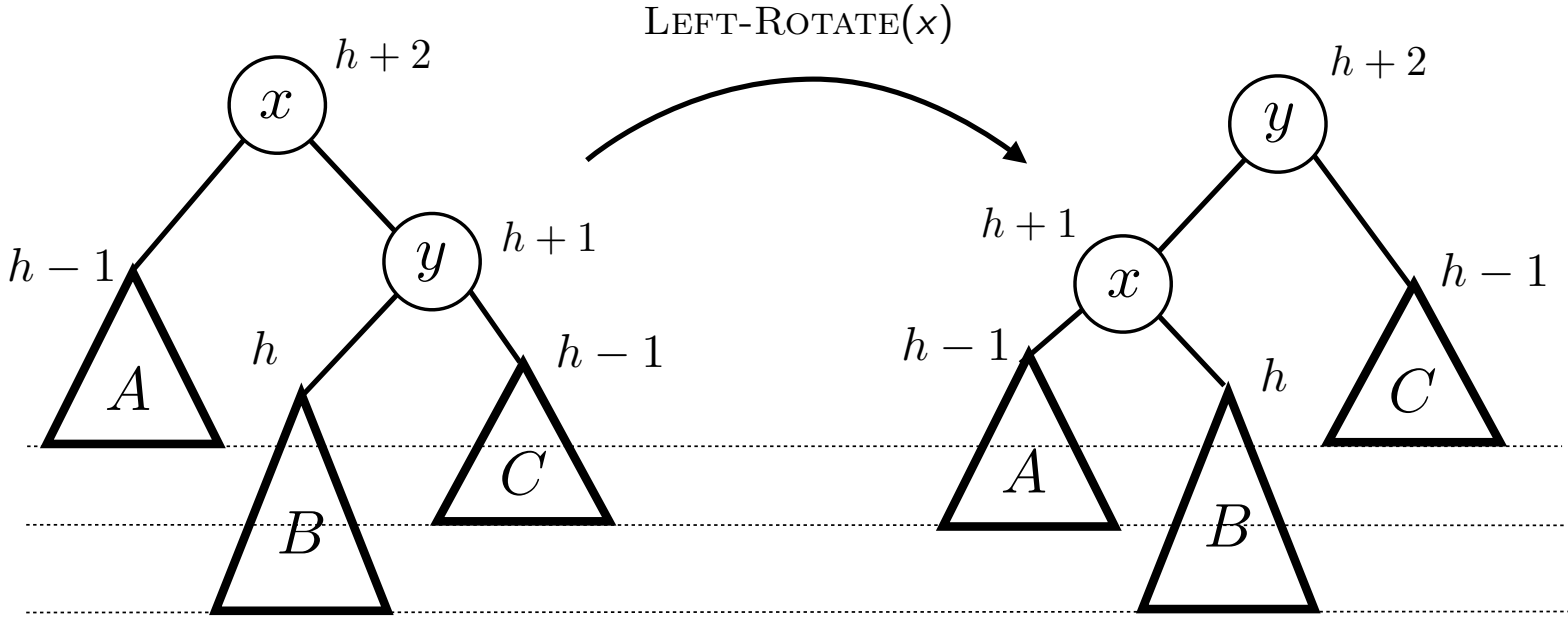
Cas 1.1 : déséquilibre à droite, extérieur (droite-droite)

- Equilibre rétabli par une rotation à gauche de x

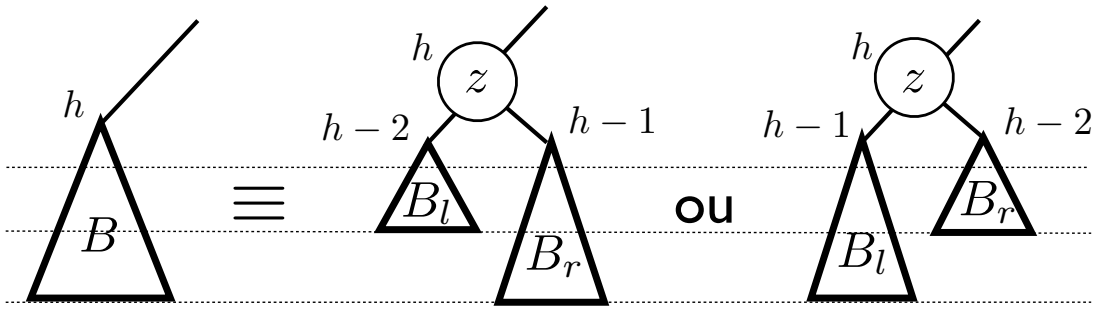


Cas 1.2 : déséquilibre à droite, intérieur (droite-gauche)

- Une rotation à gauche ne permet pas de rétablir l'équilibre

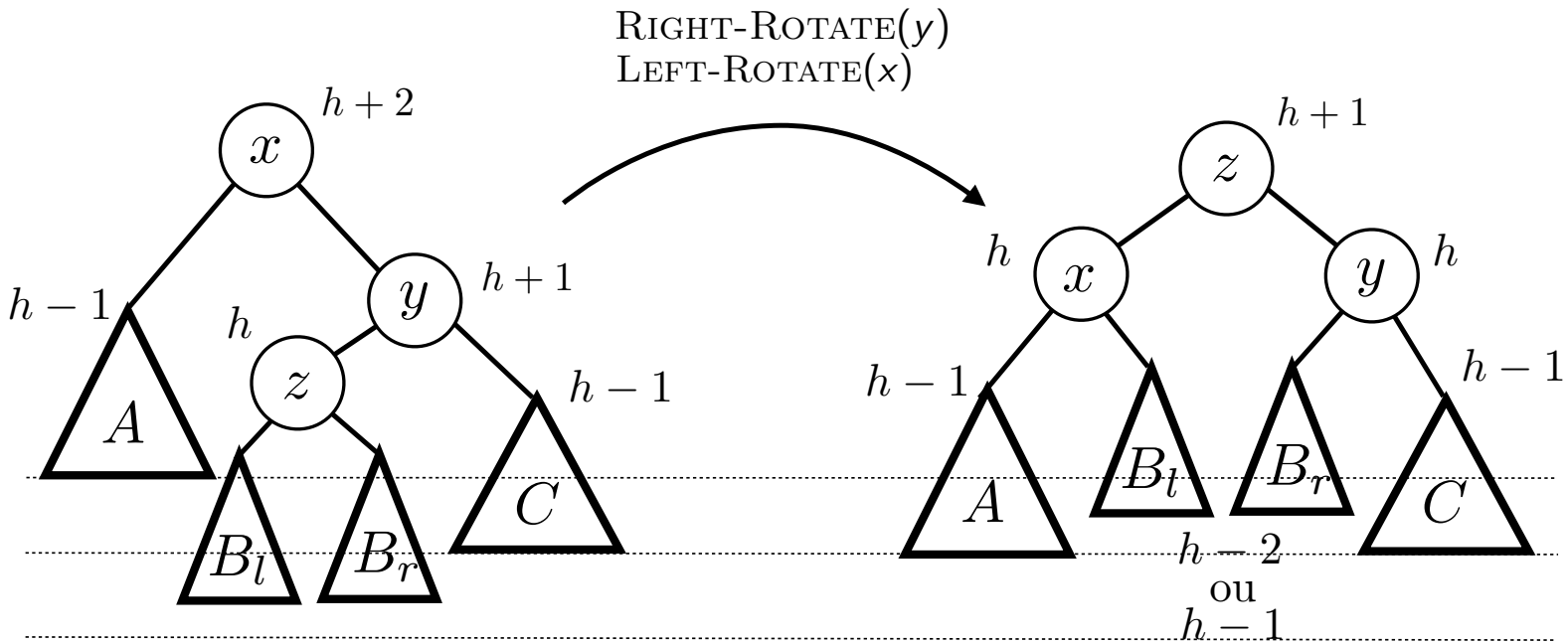


- Le sous-arbre B contient au moins un élément (l'élément inséré)



Cas 1.2 : déséquilibre à droite, intérieur (droite-gauche)

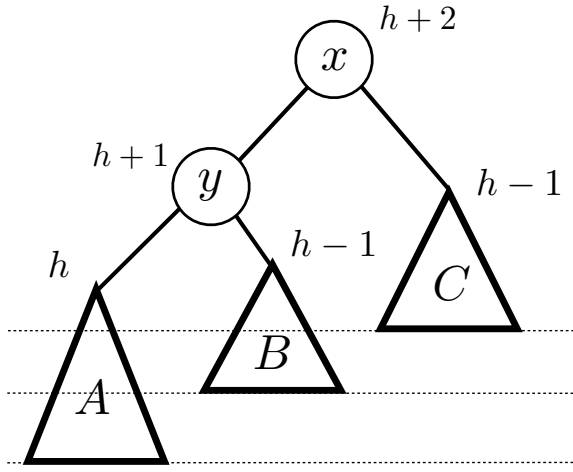
- Equilibre rétabli par deux rotations



Cas 2 : déséquilibre à gauche

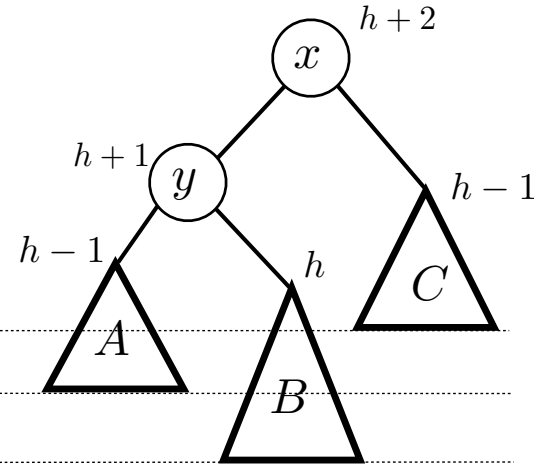
- Symétrique du cas 1
- Deux sous-cas possibles

Cas 2.1



Déséquilibre à l'extérieur
(Cas gauche-gauche)

Cas 2.2



Déséquilibre à l'intérieur
(Cas gauche-droite)

- Résolus respectivement par une rotation (à droite) et une double rotation.

Implémentation

- Algorithme récursif : Pour insérer une clé dans un arbre T :
  - On l'insère (récursivement) dans le sous-arbre approprié (gauche ou droit)
  - Si l'arbre résultant T devient déséquilibré, on effectue une rotation simple ou double selon le cas dans lequel on se trouve
- L'arbre après rééquilibrage étant de la même hauteur qu'avant l'insertion, on n'aura à faire qu'au plus une rotation (simple ou double).
- L'implémentation est facilitée si on maintient en chaque nœud x un attribut $x.h$ avec la hauteur du sous-arbre en x.
- Complexité :
  - $O(h)$ où h est la hauteur de l'arbre,
  - c'est-à-dire $O(\log n)$ vu que l'arbre est H-équilibré.

Suppression

- Comme pour l'insertion, on doit rétablir l'équilibre suite à la suppression
- La suppression d'un nœud peut déséquilibrer le parent de ce nœud
- Contrairement à l'insertion, on peut devoir rééquilibrer plusieurs ancêtres du nœud supprimé.
- Chaque rotation étant d'ordre $O(1)$, la complexité d'une suppression reste cependant $O(h)$ pour un arbre de hauteur h et donc $O(\log n)$ pour un AVL.

Tri avec un AVL

- Comme avec un arbre de binaire de recherche ordinaire, on peut trier avec un AVL
  - On insère les éléments successivement dans l'arbre
  - On effectue un parcours en ordre de l'arbre
- Complexité en temps : $\Theta(n \log n)$ (comme pour le tri par tas)
- Complexité en espace : $\Theta(n)$ (pour la structure d'arbre temporaire) (versus $O(1)$ pour le heap-sort)

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Peut-on faire mieux ?
- Oui, en changeant radicalement de philosophie

Demo

Illustrations :

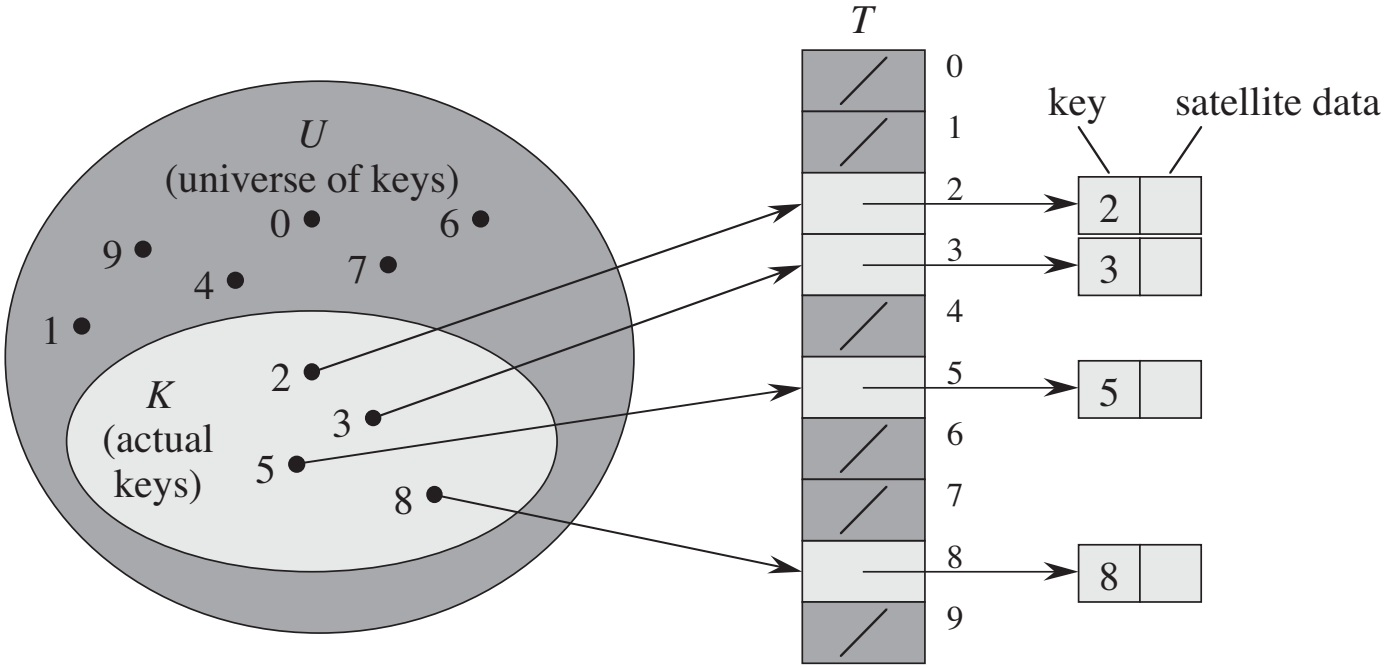
- `http://people.ksp.sk/~kuko/bak/`
- `http://www.csi.uottawa.ca/~stan/csi2514/applets/avl/BT.html`
- `http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html`
- `http://www.cs.usfca.edu/~galles/visualization/flash.html`

Plan

Tableau à accès direct

- On suppose :
  - que chaque élément a une clé tirée d'un univers $U = \{0, 1, \ldots, m-1\}$ où m n'est pas trop grand
  - qu'il ne peut pas y avoir deux éléments avec la même clé.
- Le dictionnaire est implémenté par un tableau $T[0 \ldots m-1]$:
  - Chaque position dans la table correspond à une clé de U.
  - S'il y a un élément x avec la clé k, alors $T[k]$ contient un pointeur vers x.
  - Sinon, $T[k]$ est vide ($T[k] = \mathrm{NIL}$).

Tableau à accès direct




```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
1  return T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
1  return T[x.key] = NIL
```

Tableau à accès direct

- Complexité de toutes les opérations : $O(1)$ (dans tous les cas)
- Problème :
  - Complexité en espace : $\Theta(|U|)$
  - si l'univers de clés U est grand, stocker une table de taille $|U|$ peut être peu pratique, voire impossible
- Souvent l'ensemble des clés réellement stockées, noté K, est petit comparé à U et donc l'espace alloué est gaspillé.

- Comment bénéficier de l'accès rapide d'une table à accès direct avec une table de taille raisonnable ?

  $\Rightarrow$ Table de hachage :
  - Réduit le stockage à $\Theta(|K|)$
  - Recherche en $O(1)$ (en moyenne !)

Table de hachage

- Inventée en 1953 par Luhn
- Idée :
  - Utiliser une table T de taille $m \ll |U|$
  - stocker x à la position $h(x.key)$, où h est une fonction de hachage :

$$h : U \to \{0, \ldots, m-1\}$$

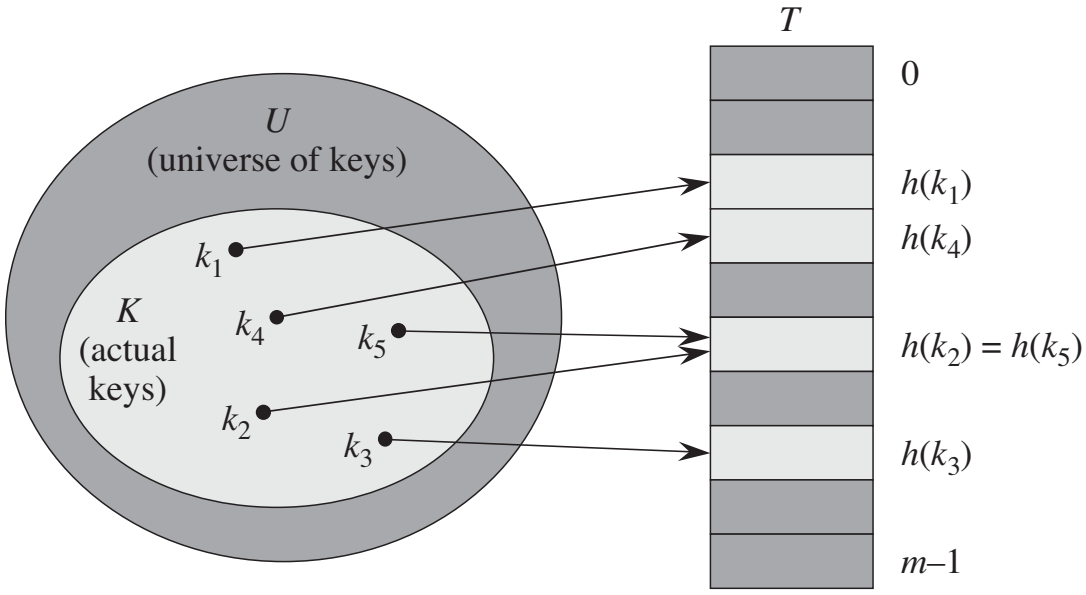
```
HASH-INSERT(T, x)
1  T[h(x.key)] = x
```

```
HASH-DELETE(T, x)
1  T[h(x.key)] = NIL
```

```
HASH-SEARCH(T, x)
1  return T[h(x.key)]
```

Est-ce que ces algorithmes sont corrects ?

Table de hachage : collisions



- Collision : lorsque deux clés distinctes k_1 et k_2 sont telles que $h(k_1) = h(k_2)$
- Cela se produit toujours lorsque le nombre de clés observées est plus grand que la taille du tableau T ($|K| > m$)
- Très probable, même lorsque la fonction de hachage répartit les clés uniformément $\Rightarrow$ Paradoxe des anniversaire

Paradoxe des anniversaires

- Hypothèse :
  - On néglige les années bissextiles
  - Les 365 jours présentent la même probabilité d'être un jour d'anniversaire
- Si p est la probabilité d'une collision d'anniversaires :

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} \cdots \frac{365 - (n-1)}{365} = \frac{365!}{(365-n)!365^n}$$

  Exemples :
  - $n = 23 \Rightarrow p > 0,5$
  - $n = 57 \Rightarrow p > 0,99$
  - $n = 70 \Rightarrow p > 0,999$

- Pour une table de hachage :
  - $m = 365$ et 57 clés $\Rightarrow$ plus de 99% de chance de collision
  - $m = 1000000$ et 2500 clés $\Rightarrow$ plus de 95% de chance de collision

Collision

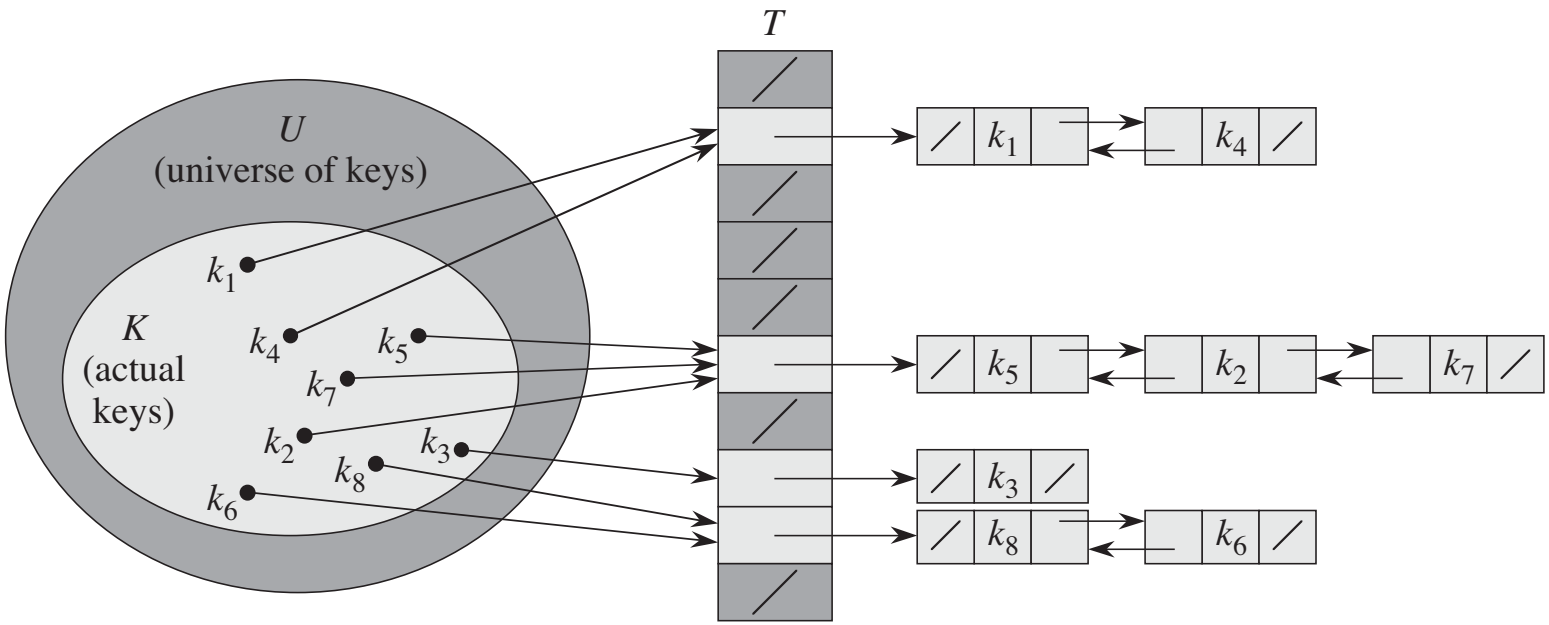
- Pour éviter les collisions :
  - on veille à utiliser une fonction de hachage qui disperse le plus possible les clés vers les différents compartiments.
  - on utilise un nombre de compartiments suffisamment grand

  Cependant, même dans ce cas, la probabilité de collision peut être non négligeable.

- Deux approches pour prendre en compte les collisions :
  - Le chaînage (adressage fermé)
  - Le sondage (adressage ouvert)

Résolution des collisions par chaînage

Solution : mettre les éléments qui sont “hachés” vers la même position dans une liste liée (simple ou double)



Implémentation des opérations

```
CHAINED-HASH-INSERT(T, x)
1  LIST-INSERT(T[h(x.key)], x)
```

```
CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```

```
CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)
```

- Complexité :
  - Insertion : $O(1)$
  - Suppression : $O(1)$ si liste doublement liée, $O(n)$ pour une liste de taille n si liste simplement liée.
  - Recherche : $O(n)$ si liste de taille n.

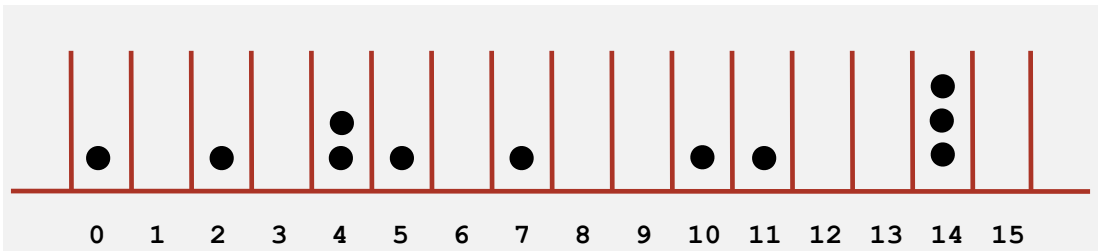
Analyse du cas moyen

- Recherche d'une clé k dans la table :
  - recherche positive : la clé k se trouve dans la table
  - recherche négative : la clé k n'est pas dans la table

- Le facteur de charge d'une table de hachage est donné par $\alpha = \frac{n}{m}$ où :
  - n est le nombre d'éléments dans la table
  - m est la taille de la table (c'est-à-dire, le nombre de listes liées)

- hachage uniforme simple : Pour toute clé $k \in U$,

$$Proba\{h(k) = i\} = \frac{1}{m}, \forall i \in \{0, \ldots, m-1\}$$



Analyse du cas moyen

- Hypothèses :
  - h produit un hachage uniforme simple
  - le calcul de $h(k)$ est $\Theta(1)$
  - Insertion en début de liste
- $\Rightarrow$ complexités moyennes :
  - recherche négative : $\Theta(1+\alpha)$
  - recherche positive : $\Theta(1+\alpha)$
- Si $n = O(m)$, *(m croît au moins linéairement avec n)*,

$$\alpha = \frac{O(m)}{m} = O(1)$$

- Toutes les opérations sont donc $O(1)$ en moyenne

Analyse du cas moyen : recherche négative

- La clé k ne se trouve pas dans la table
- Par la propriété de hachage uniforme simple, elle a la même probabilité d'être envoyée vers chaque position dans la table.
- Recherche négative requière le parcours de la liste $T[h(k)]$ complète
- Cette liste a une longueur moyenne $E[n_{h(k)}] = \alpha$.
- Le nombre d'éléments à examiner lors d'une recherche négative est donc α.
- En ajoutant le temps de calcul de la fonction de hachage, on arrive à une complexité moyenne $\Theta(1 + \alpha)$.

Analyse du cas moyen : recherche positive

- La clé k se trouve dans la table
- Supposons qu'elle ait été insérée à la i-ième étape (parmi n).
  - Le nombre d'éléments à examiner pour trouver la clé est le nombre d'éléments insérés à la position $h(k)$ après k plus 1 (la clé k elle-même).
  - En moyenne, sur les $n - i$ insertions après k, il y en aura $(n - i)/m$ qui correspondront à la position $h(k)$.
- k ayant pu être insérée à n'importe quelle étape parmi n avec une probabilité $1/n$:

$$\sum_{i=1}^{n} \frac{1}{n}(1+\frac{n-i}{m}) = 1+\frac{1}{nm}(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i) = 1+\frac{1}{nm}(n^2-\frac{n(n+1)}{2}) = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

- En tenant compte du coût du hachage, la complexité en moyenne est donc $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

Plan

Fonctions de hachage

- Idéalement, la fonction de hachage
  - devrait être facile à calculer ($O(1)$)
  - devrait satisfaire l'hypothèse de hachage uniforme simple
- La deuxième propriété est très difficile à assurer en pratique :
  - La distribution des clés est généralement inconnue
  - Les clés peuvent ne pas être indépendantes
- En pratique, on utilise des heuristiques basées sur la nature attendue des clés
- Si toutes les clés sont connues, il existe des algorithmes pour construire une fonction de hachage parfaite, sans collision (Exemple : le logiciel gperf)

Fonctions de hachage : codage préalable

- Les fonctions de hachage supposent que les clés sont des nombres naturels
- Si ce n'est pas le cas, il faut préalablement utiliser une fonction de codage
- Exemple : codage des chaînes de caractères :
  - On interprète la chaîne comme un entier dans une certaine base
  - Exemple pour "SDA" : valeurs ASCII (128 possibles) :

$$S = 83, D = 68, A = 65$$

  - Interprété comme l'entier : *(Pourquoi pas 83+68+65 ?)*

$$(83 \cdot 128^2) + (68 \cdot 128^1) + (65 \cdot 128^0) = 1368641$$

  - Calculé efficacement par la méthode de Horner :

$$((83 \cdot 128 + 68) \cdot 128 + 65)$$

Méthode de division

La fonction de hachage calcule le reste de la division entière de la clé par la taille de la table

$$h(k) = k \bmod m.$$

Exemple : $m = 20$ et $k = 91 \Rightarrow h(k) = 11$.

Avantages : simple et rapide (juste une opération de division)

Inconvénients : Le choix de m est très sensible et certaines valeurs doivent être évitées

Exemples :

- Si $m = 2^p$ pour un entier p, $h(k)$ ne dépend que des p bits les moins significatifs de k
  - Exemple : "SDA" mod 128= "GAGA" mod 128=65
- Si k est une chaîne de caractères codée en base 2^p et $m = 2^p - 1$, permuter la chaîne ne modifie pas le valeur de hachage
  - Exemple : "SDA"=1368641, "DSA"=1124801
    $\Rightarrow$ 1368641 mod 127 = 1124801 mod 127 = 89

Méthode de division

- Si la fonction de hachage produit des séquences périodiques, il vaut mieux choisir m premier
- En effet, si m est premier avec b, on a :

$$\{(a + b \cdot i) \bmod m | i = 0, 1, 2, \ldots\} = \{0, 1, 2, \ldots, m - 1\}$$

- Exemple : hachage de $\{206, 211, 216, 221, \ldots\}$
  - $m = 100$: valeurs hachées possibles : 6, 11,. . . , 96
  - $m = 101$: toutes les entrées sont exploitées

$\Rightarrow$ Bonne valeur de m : un nombre premier pas trop près d'une puissance exacte de 2

Méthode de multiplication

- Fonction de hachage :

$$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$

où

  - A est une constante telle que $0 < A < 1$.
  - $kA \bmod 1 = kA - \lfloor kA \rfloor$ est la partie fractionnaire de kA.

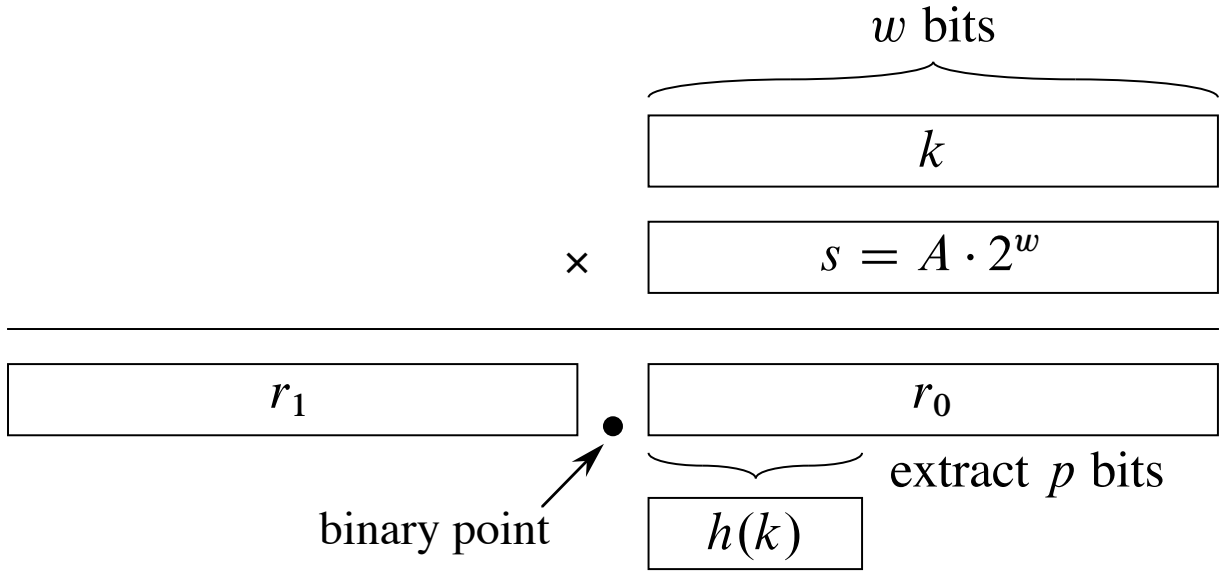
- Inconvénient : plus lente que la méthode de division
- Avantage : la valeur de m n'est plus critique
- La méthode marche mieux pour certaines valeurs de A. Par exemple :

$$A = \frac{\sqrt{5} - 1}{2}$$

Méthode de multiplication : implémentation

Calcul aisé si :

- $m = 2^p$ pour un entier p
- Les mots sont codés en w bits et les clés k peuvent être codées par un seul mot
- A de la forme $s/2^w$ pour $0 < s < 2^w$



Exemple : $m = 2^3$, $w = 5$ $(\Rightarrow 0 < s < 2^5)$, $s = 13$, $A = 13/32 \Rightarrow h(21) = 4$

Plan

Adressage ouvert : principe

- Alternative au chaînage pour gérer les collisions
- Tous les éléments sont stockés dans le tableau (pas de listes chaînées)
- Ne fonctionne que si $\alpha \leq 1$
- Pour insérer une clé k, on sonde les cases systématiquement à partir de $h(k)$ jusqu'à en trouver une vide.
- Différentes méthodes en fonction de la stratégie de sondage

Adressage ouvert : stratégie de sondage

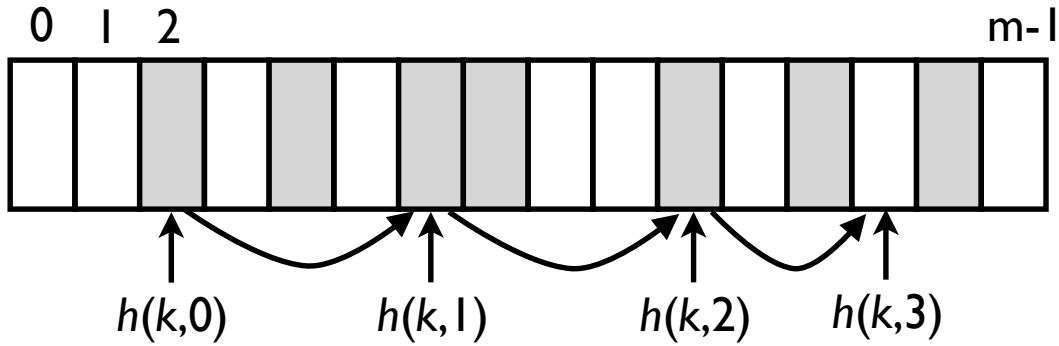
On définit une nouvelle fonction de hachage qui dépend de la clé et du numéro du sondage :

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

et qui est telle que

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

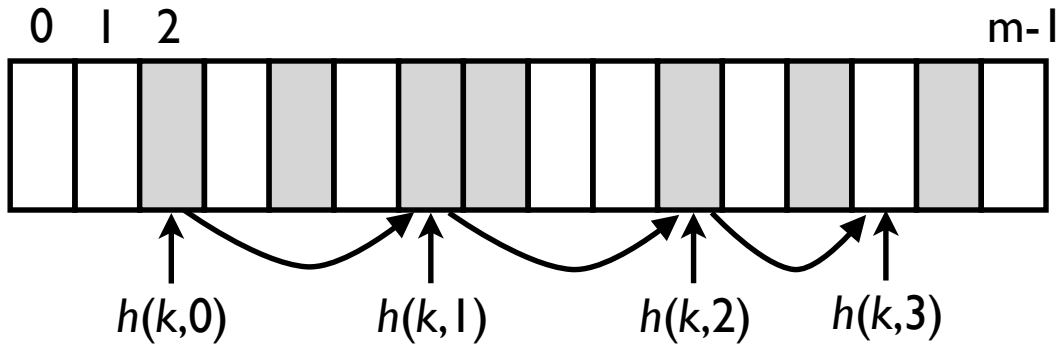
est une permutation de $\langle 0, 1, \ldots, m-1 \rangle$.



Adressage ouvert : recherche et insertion

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```



Adressage ouvert : suppression

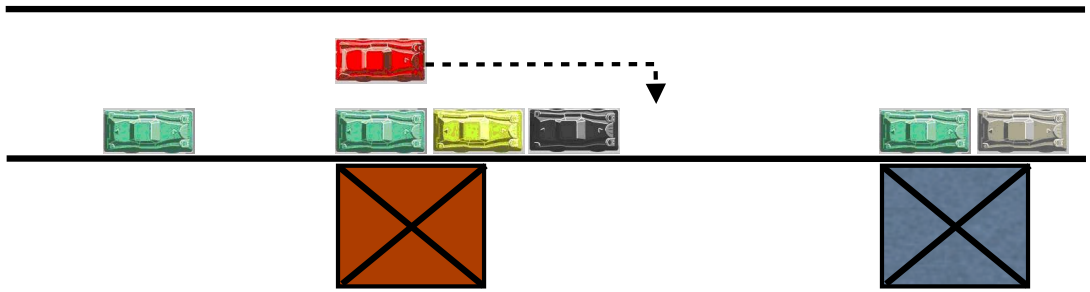
- La suppression est possible mais pas aisée
  - On évitera l'utilisation de l'adressage ouvert si on prévoit de nombreuses suppressions de clés dans le dictionnaire
- On ne peut pas naïvement mettre NIL dans la case contenant la clé k qu'on désire effacer
- Solution :
  - Utiliser une valeur spéciale $\mathrm{DELETED}$ au lieu de NIL pour signifier qu'on a effacé une valeur dans cette case
  - Lors d'une recherche : considérer un case contenant $\mathrm{DELETED}$ comme une case contenant une clé
  - Lors d'une insertion : considérer une case contenant $\mathrm{DELETED}$ comme une case vide.
- Inconvénient : le temps de recherche ne dépend maintenant plus du facteur de charge α de la table

Stratégies de sondage

- Soit $h_k = \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ la séquence de sondage correspondant à la clé k.
- Hachage uniforme :
  - chacun des $m!$ permutations de $\langle 0, 1, \ldots, m-1 \rangle$ a la même probabilité d'être la séquence de sondage d'une clé k.
  - Difficile à implémenter.
- En pratique, on se contente d'une garantie que la séquence de sondage soit une permutation de $\langle 0, 1, \ldots, m-1 \rangle$.

- Trois techniques pseudo-uniformes :
  - sondage linéaire
  - sondage quadratique
  - double hachage

Sondage linéaire



$$h(k,i) = (h'(k) + i) \bmod m,$$

où $h'(k)$ est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$.
Propriétés :

- très facile à implémenter
- effet de grappe fort : création de longues suites de cellules occupées
  - La probabilité de remplir une cellule vide est $\frac{i+1}{m}$ où i est le nombre de cellules pleines précédant la cellule vide
- pas très uniforme

Sondage quadratique

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

où h' est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$, c_1 et c_2 sont deux constantes non nulles.

Propriétés :

- nécessité de bien choisir les constantes c_1 et c_2 (pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$)
- effet de grappe plus faible mais tout de même existant :
  - Deux clés de même valeur de hachage suivront le même chemin

$$h(k,0) = h(k',0) \Rightarrow h(k,i) = h(k',i)$$

- meilleur que le sondage linéaire

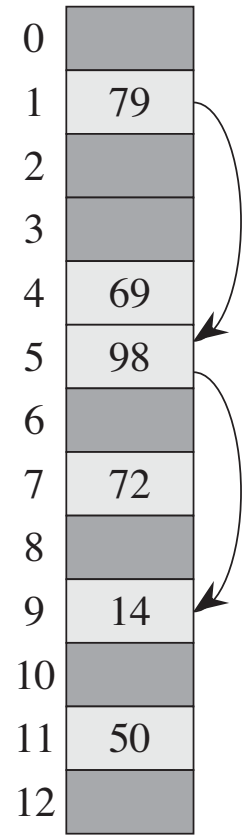
Double hachage

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$

où h_1 et h_2 sont des fonctions de hachage ordinaires à valeurs dans $\{0, 1, \ldots, m-1\}$.

Propriétés :

- difficile à implémenter à cause du choix de h_1 et h_2 ($h_2(k)$ doit être premier avec m pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$).
- très proche du hachage uniforme
- bien meilleur que les sondages linéaire et quadratique



Exemple : $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$, *insertion de la clé 14*

Adressage ouvert : élément d'analyse

Pour une table de hachage à adressage ouvert de taille m contenant n éléments ($\alpha = n/m < 1$) et en supposant le hachage uniforme

- Le nombre moyen de sondages pour une recherche négative ou un ajout est borné par $\frac{1}{1-\alpha}$
- Le nombre moyen de sondages pour une recherche positive est borné par $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$

$\Rightarrow$ Si α est constant ($n = O(m)$), la recherche est $O(1)$.

- Si $\alpha = 0.5$, une recherche nécessite en moyenne 2 sondages ($1/(1-0.5)$).
- Si $\alpha = 0.9$, une recherche nécessite en moyenne 10 sondages ($1/(1-0.9)$).

Adressage ouvert versus chaînage

- Chaînage :
  - Peut gérer un nombre illimité d'éléments et de collisions
  - Performances plus stables
  - Surcoût lié à la gestion et le stockage en mémoire des listes liées
- Adressage ouvert :
  - Rapide et peu gourmand en mémoire
  - Choix de la fonction de hachage plus difficile (pour éviter les grappes)
  - On ne peut pas avoir $n > m$
  - Suppression problématique
- D'autres alternatives existent :
  - Two-probe hashing
  - Cuckoo hashing
  - ...

Le rehachage

- Lorsque α se rapproche de 1, les performances s'effondrent
- Solution : rehachage : création d'une table plus grande
  - allocation d'une nouvelle table
  - détermination d'une nouvelle fonction de hachage, tenant compte du nouveau m
  - parcours des entrées de la table originale et insertion dans la nouvelle table
- Si la taille est doublée, le coût asymptotique constant des opérations est conservé (voir slide 237).

Universal hashing

- Les performances d'un table de hachage se dégrade fortement en cas de collisions multiples
- Connaissant la fonction de hachage, un adversaire malintentionné pourrait s'amuser à entrer des clés créant des collisions. Exemples :
  - Création de fichiers avec des noms bien choisis dans le kernel Linux 2.4.20
  - 28/12/2011 : `http://www.securityweek.com/hash-table-collision-attacks-could-trigger-ddos-massive-scale`
- C'est un exemple d'attaque par déni de service

- Parade : hachage universel : choisir la fonction de hachage aléatoirement à chaque création d'une nouvelle instance de la table
- Exemple :

$$h(k) = ((ak + b) \bmod p) \bmod m,$$

où p est un premier très grand et a et b deux entiers choisis aléatoirement

Demo

- `http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm`

Plan

Dictionnaires : résumé

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Cas moyen valable uniquement sous l'hypothèse de hachage uniforme
- Comment obtenir $\Theta(\log n)$ dans le pire cas avec une table de hachage ?

ABR/AVL versus table de hachage

Tables de hachage :

- Faciles à implémenter
- Seule solution pour des clés non ordonnées
- Accès et insertion très rapides en moyenne (pour des clés simples)
- Espace gaspillé lorsque α est petit
- Pas de garantie au pire cas (performances “instables”)

Arbres binaire de recherche (équilibrés) :

- Performance garantie dans tous les cas (stabilité)
- Taille de structure s'adapte à la taille des données
- Supportent des opérations supplémentaires lorsque les clés sont ordonnées (parcours en ordre, successeur, prédécesseur, etc.)
- Accès et insertion plus lente en moyenne