

# Dictionnaires ordonnés et “Skip List”

## Dictionnaires ordonnés

- Les clés sont comparés selon un ordre total
- On veut pouvoir faire toutes les opérations habituelles sur le dictionnaire mais en conservant l'ordre des clés
- Nouvelles opérations:
  - **premier()**: Retourne le premier élément du dictionnaire  
**first()**:
  - **dernier()**: Retourne le dernier élément du dictionnaire  
**last()**:

## Dictionnaires ordonnés (suite)

- Nouvelles opérations (suite):
  - **successeurs(k)**: Retourne un itérateur des entrées dont la clé est plus grande ou égale à k; en ordre croissant  
**successors(k)**:
  - **prédécesseurs(k)**: Retourne un itérateur des entrées dont la clé est plus petite ou égale à k; en ordre décroissant  
**predecessors(k)**:
  - **closestKeyBefore(k)**: Retourne la clé (ou la valeur) de l'entrée ayant la plus grande clé plus petite ou égale à k  
**closestValBefore(k)**:
  - **closestKeyAfter(k)**: Retourne la clé (ou la valeur) de l'entrée ayant la plus petite clé plus grande ou égale à k  
**closestValAfter(k)**:

## “Look-up Table”

- On appelle “Look-up table” l'implémentation d'un dictionnaire à l'aide d'une séquence ordonnée
  - On insère les éléments du dictionnaire dans un vecteur, de façon à ce que les clés forment une séquence ordonnée
  - Pour ordonner les clés, on utilise un comparateur de clé externe au dictionnaire

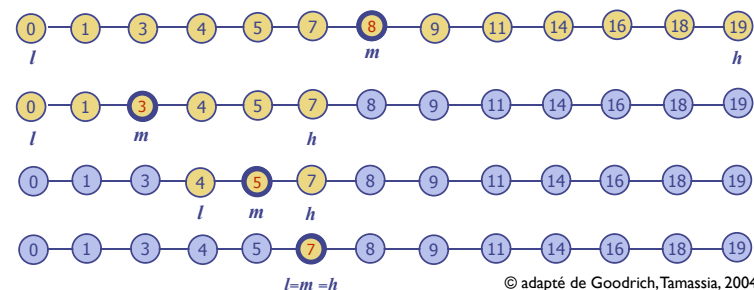
4	4	9	18	28	28	32	59	59	59	75	78	82	88
---	---	---	----	----	----	----	----	----	----	----	----	----	----

## Performances

- Insérer un nouvel élément prend, dans le pire des cas, un temps  $O(n)$ , étant donné que dans ce cas, on doit déplacer les  $n$  éléments déjà présent pour faire de la place à ce nouvel élément
- Enlever un élément prend, dans le pire des cas, un temps  $O(n)$ , étant donné que dans ce cas, on doit déplacer les  $n$  éléments déjà présent vers la gauche (après avoir enlevé l'élément de clé minimal)
- Chercher un élément prend dans le pire des cas un temps  $O(\log n)$ , lorsqu'on utilise la recherche binaire

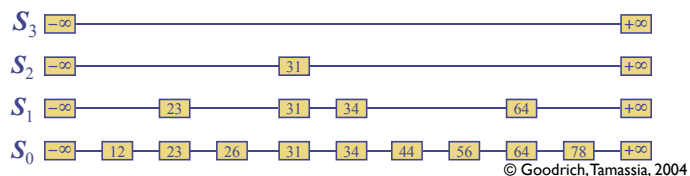
## Recherche binaire

- Comme nos clés sont ordonnées, on peut commencer à chercher pour un élément de clé  $k$ , en comparant  $k$  avec la clé de l'élément en milieu de séquence, si  $k$  est  $<$  que cette clé, on cherche l'élément de clé  $k$  dans la partie gauche de la séquence, si  $>$ , dans la partie droite
- Exemple, recherche d'un élément de clé = 7



## "Skip List"

- Une "skip list"  $S$  pour un dictionnaire  $D$  consiste en une série de séquences  $\{S_0, S_1, \dots, S_h\}$ . Chaque séquence contient un sous-ensemble des entrées du dictionnaire et, de plus,  $S$  satisfait:
  - Chaque séquence  $S_i$  contient des entrées de clés spéciales:  $-\infty$  et  $+\infty$
  - La séquence  $S_0$  contient tous les éléments du dictionnaire, placés en ordre croissant de clé
  - Chaque séquence est une sous-séquence de la précédente:
 
$$S_h \subseteq S_{h-1} \subseteq \dots \subseteq S_0$$
  - La séquence  $S_h$  contient seulement les 2 éléments de clés spéciales

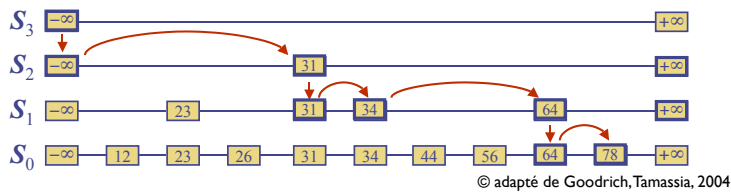


## Algorithmes randomisés

- On utilise un algorithme randomisé pour insérer les entrées dans une "skip list".
- Un algorithme randomisé utilise un algorithme de génération aléatoire de bits pour contrôler son exécution. (Ex. Pile ou Face)
- Son temps d'exécution dépend de la séquence de bits générée (ou de la séquence de piles-faces).
- Le temps d'exécution, dans le pire des cas, est souvent grand mais arrive selon une très faible probabilité. (ex. on a eu seulement des piles)

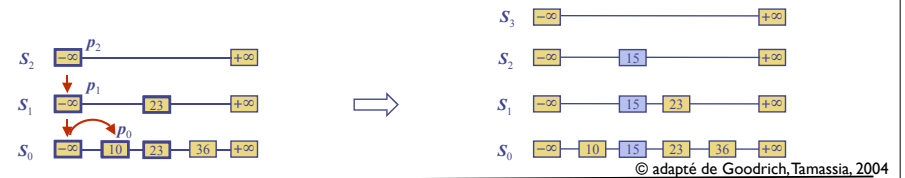
## Chercher

- Exemple: chercher un élément de clé 78
- Pour chercher un élément de clé  $k$  dans une “skip list”, on procède de la façon suivante:
  - On se place dans la première clé de la liste du haut
  - Tant qu'on peut descendre et qu'on n'a pas trouvé l'élément de clé  $k$ , on exécute les opérations suivantes:
    - On descend
    - Tant que la clé à droite est plus petite que  $k$ , on se déplace vers la droite



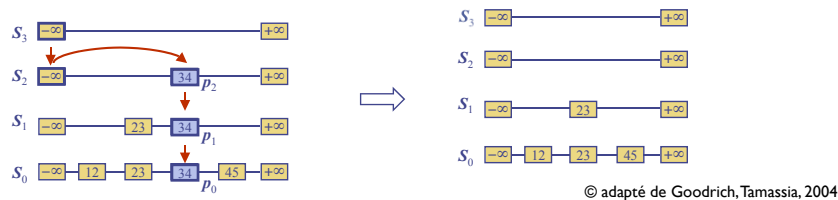
## Insérer

- Pour insérer un élément  $(k,v)$  dans une “skip list”, on utilise un algorithme randomisé comme suit:
  - On joue à “pile ou face” jusqu'à ce que l'on obtienne “pile”. On dénote  $i$  le nombre de fois où on a eu “face” avant ce “pile”.
  - Si  $i \geq h$ , on ajoute à la “skip list” de nouvelles séquences  $S_{h+1}, \dots, S_{i+1}$ , chacune contenant seulement les éléments de clés spéciales  $+\infty$  et  $-\infty$
- Exemple: insérer un élément de clé 15,  $i=2$ 
  - On utilise l'algorithme chercher( $k$ ) pour trouver les positions  $p_0, p_1, \dots, p_i$  des entrées de plus grande clé plus petite ou égale à  $k$  dans chaque séquence  $S_0, S_1, \dots, S_i$
  - Pour  $j \leftarrow 0, \dots, i$ , on insère  $(k,v)$  dans  $S_j$  après la position  $p_j$



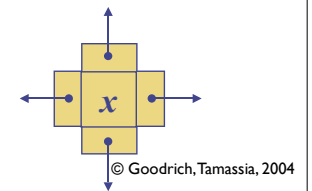
## Supprimer

- Pour supprimer un élément de clé  $k$  dans une “skip list”, on procède comme suit:
  - On cherche dans la “skip list” et on trouve les positions  $p_0, p_1, \dots, p_i$  d'une entrée de clé  $k$ , où la position  $p_j$  est dans la séquence  $S_j$
  - On enlève les éléments de positions  $p_0, p_1, \dots, p_i$  des séquences  $S_0, S_1, \dots, S_i$
  - On ne garde qu'une séquence ne contenant que les deux clés spéciales
- Exemple: Enlever(34)



## Implémentation

- On peut implémenter une “skip list” avec des noeuds quatruples de cette forme:
- Le noeud garde en mémoire:
  - l'entrée  $x = (k,v)$
  - un lien vers le noeud précédent
  - un lien vers le noeud suivant
  - un lien vers le noeud au-dessus
  - un lien vers le noeud en-dessous
- Le comparateur des clés pour ces noeuds doit être capable de comparer les clés spéciales



## Complexité en espace

- L'espace utilisé par une "skip list" dépend des bits randomisés utilisés lors de chaque appel de la fonction d'insertion.
- Dans le cas de l'utilisation de l'algorithme randomisé "pile ou face", on utilise les deux faits suivants en probabilité:
  - **Fait 1:** La probabilité d'obtenir  $i$  faces consécutives lors de lancers d'une pièce de monnaie est de  $1/2^i$
  - **Fait 2:** Si étant donné  $n$  éléments, on choisit leur appartenance à un ensemble avec une probabilité  $p$ , alors la taille espérée de l'ensemble est  $np$
- Donc, si l'on a une "skip list" avec  $n$  entrées
  - Par le **fait 1**, on insère un élément dans la séquence  $S_i$  avec probabilité  $1/2^i$
  - Par le **fait 2**, la taille espérée de la séquence  $S_i$  est  $n/2^i$

## Complexité en espace (suite)

- Le nombre espéré de noeuds dans une "skip list" est:

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

- La complexité en espace d'une "skip list" est donc de  $O(n)$

## Hauteur de la "skip list"

- Le temps d'exécution des algorithmes de recherche et d'insertion dépend de la hauteur  $h$  de la "skip list"
- On va montrer que, avec une très grande probabilité, la hauteur d'une "skip list" ayant  $n$  éléments est de  $O(\log n)$
- Pour ce faire, on va utiliser les faits suivants:
  - **Fait 1:** La probabilité d'obtenir  $i$  faces consécutives lors de lancers d'une pièce de monnaie est de  $1/2^i$
  - **Fait 2:** Si étant donné  $n$  éléments, on choisit leur appartenance à un ensemble avec une probabilité  $p$ , alors la taille espérée de l'ensemble est  $np$
  - **Fait 3:** Si  $n$  événements ont chacun une probabilité  $p$  d'arriver, alors la probabilité qu'au moins un événement arrive est au plus  $np$

## Complexité en temps

- Le temps d'exécution de l'algorithme de recherche dans une "skip list" est proportionnel au
  - Nombre de descentes +
  - Nombre de comparaisons vers la droite
- Le nombre de descentes est borné par la hauteur de la "skip list" et est donc en  $O(\log n)$  avec une très haute probabilité
- Le nombre total de comparaisons vers la droite est aussi en  $O(\log n)$
- La complexité en temps de l'algorithme de recherche est  $O(\log n)$
- L'analyse des algorithmes d'insertions et de suppressions donne des résultats similaires