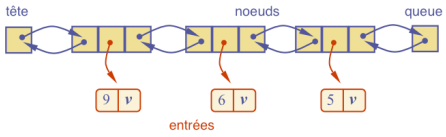
Révision Final

1) Dictionnaires

- Structure de données permettant l'insertion, la supression et la recherche d'un élément de clé k
- Deux types
  - Dictionnaires non-ordonnées → Testeur d'égalités pour les clés
  - Dictionnaires ordonnées → 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-ordonnées → Listes chaînés, tableaux, tables de hachage
  - Dictionnaires ordonnées → "look up table","skip list"

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: $O(n)$

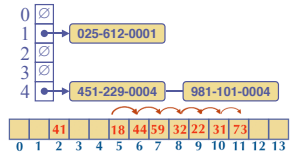


- Tableau:
  - Complexité en temps:
    - insérer $O(1)$ (pour les "maps" $O(n)$)
    - enlever $O(1)$
    - trouver $O(1)$
  - Complexité en espace: $O(N)$

- Problème:
  - Permet de travailler avec les clés entières seulement
  - Collisions
  - N > > > 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 $\rightarrow$ entiers
    - Fonction de compression h_2 : entiers $\rightarrow [0, N-1]$
  - On insère un élément de clé k dans la cellule h(k)
- Méthodes de résolution de collisions + performances:
  - Hachage ouvert: méthode par chaînage
  - Hachage fermé: sondage linéaire, quadratique, aléatoire, hachage double



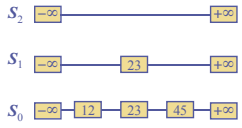
3) Implémentations Dictionnaires ordonnés

- "Look-up table"

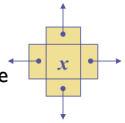
4	4	9	18	28	28	32	59	59	59	75	78	82	88

  - Complexité en temps:
    - insérer $O(n)$
    - enlever $O(n)$
    - trouver $O(\log n)$: recherche binaire
  - Complexité en espace: $O(n)$

- "Skip List"



  - Une "skip list" est une structure de données pour les dictionnaires qui utilise un algorithme randomisé pour l'insertion d'éléments
  - L'implémentation d'une skip list se fait à l'aide d'une structure chaînée, composée de noeuds quadruples:

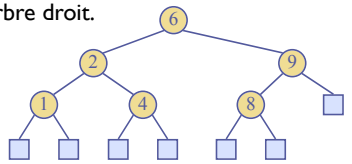


  - On a avec une très haute probabilité les complexités suivantes:
    - insérer $O(\log n)$
    - enlever $O(\log n)$
    - trouver $O(\log n)$
  - Complexité en espace: $O(n)$

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

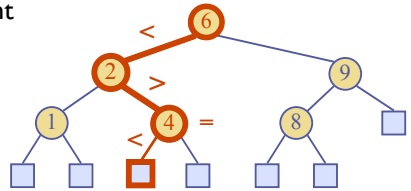
$$clé(u) < clé(v) \leq clé(w)$$



- 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:
  - Cherche(k) se fait en O(h) et donc en O(n) dans le pire des cas
  - Insérer(k) se fait en O(h) et donc en O(n) dans le pire des cas
  - Supprimer(k) se fait en O(h) et donc en O(n) dans le pire des cas

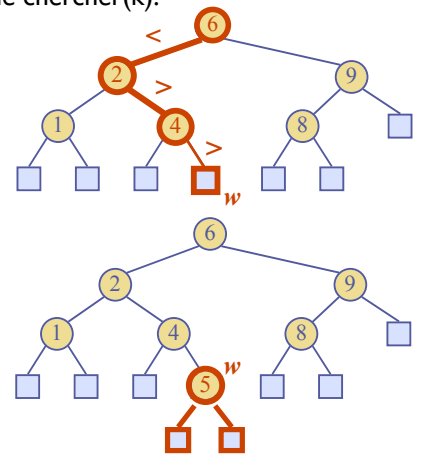
Chercher dans un arbre binaire de recherche

- 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)



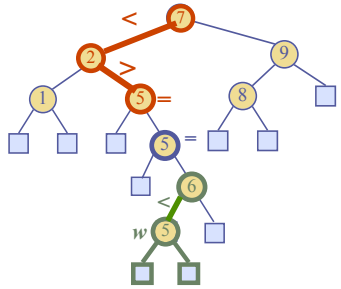
Insérer dans un arbre binaire de recherche

- 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



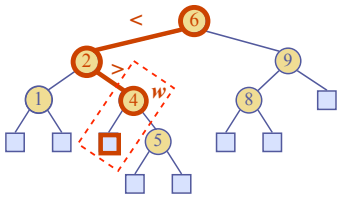
Insérer dans un arbre binaire de recherche

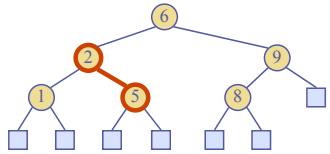
- Si k est dans l'arbre l'algorithme chercher(k) se terminera sur un noeud interne v. On appelle récursivement l'algorithme sur le filsDroit(v), jusqu'à ce qu'on arrive à un noeud externe w
- Exemple 2: Insérer(5,v)
- On insère k dans w et on change w en un noeud interne



Supprimer dans un arbre binaire de recherche

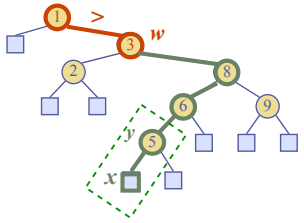
- Pour enlever un élément de clé k dans un arbre binaire de recherche, on commence par exécuter l'algorithme chercher(k).
- Exemple 1: Enlever(4)
- Si k est dans l'arbre l'algorithme chercher(k) se terminera dans un noeud interne w
- Si l'un des enfant de w est une feuille, on enlève cette feuille et w
- Sinon...

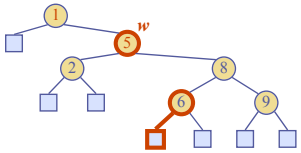




Supprimer dans un arbre binaire de recherche (suite)

- Si k est dans l'arbre, l'algorithme 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

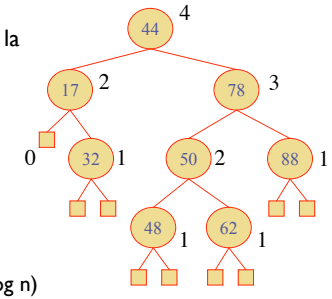




5) Arbre AVL

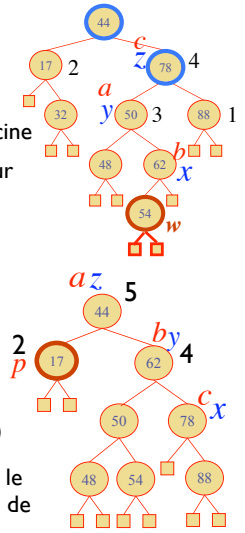
- Un arbre AVL est un arbre binaire de recherche balancé selon la propriété suivante:
  - Propriété de balance: Pour chaque noeud interne v, la hauteur des enfants de v diffère d'au plus 1
- La hauteur d'un arbre AVL est en $O(\log n)$
- Complexité en temps des opérations principales:
  - L'algorithme 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 débalancement
    - Restructurer l'arbre prend un temps

    ⟶ jusqu'à la racine: $O(\log n)$

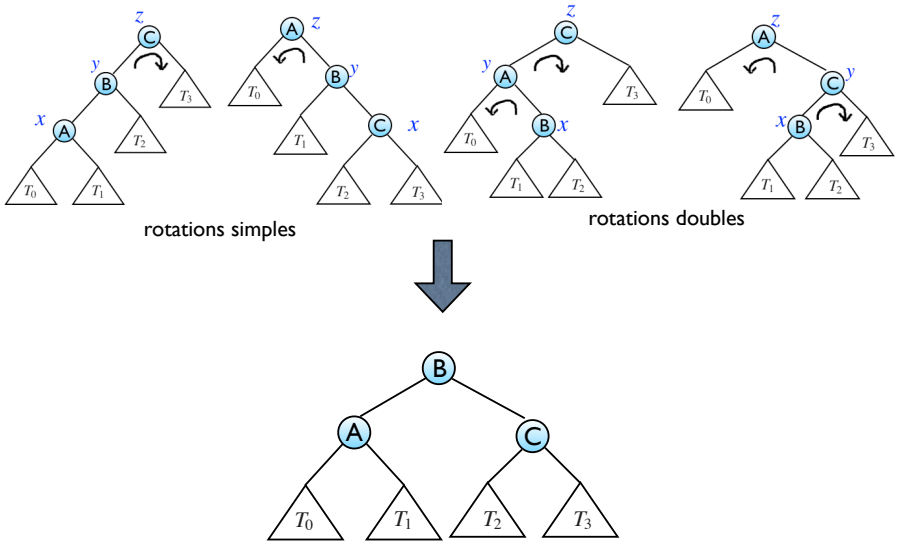


Arbre AVL: restructuration

- 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 débalancés sont situés sur le chemin allant du noeud w à la racine
  - Dans le cas d'une suppression, le noeud débalancé est situé sur le chemin allant du parent du noeud supprimé à la racine
  - On va nommer z le premier noeud débalancé 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.



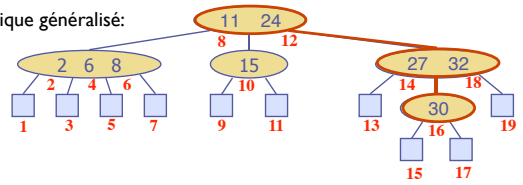
Arbre AVL: restructuration: 4 cas possibles



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 $k_1, k_2, \ldots, k_{d-1}$ et ayant pour enfants les noeuds $n_1, n_2, \ldots, n_d$ 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} ($i = 2, \ldots, d-1$)
    - les clés dans le sous-arbre de racine n_d sont plus grandes ou égales à k_{d-1}

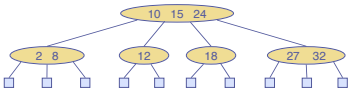
parcours symétrique généralisé:



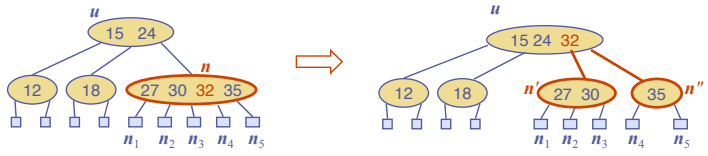
Chercher 30

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

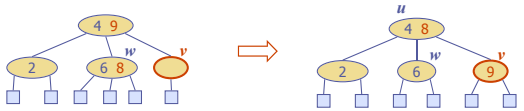


- Complexité en temps des opérations principales:
  - L'algorithme de recherche prend un temps $O(\log n)$
  - L'algorithme d'insertion prend un temps $O(\log n)$
    - Chercher l'endroit où insérer en temps $O(\log n)$
    - L'insertion peut causer un débordement
    - On exécute un fractionnement qui 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 frère est un 3 ou 4 -noeud, on exécute une opération de tranfert
    - 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

8) Arbre (a,b)

- Un arbre (a,b) est une généralisation d'un arbre (2,4) dans le sens suivant:
  - Nombre d'enfants: tout noeud interne a au moins a fils (sauf possiblement la racine) et au plus b fils.
  - Propriété de profondeur: tous les noeuds externes ont la même profondeur
  - Spécifications pour a et b: a et b sont des entiers, tels que

$$2 \leq a \leq \frac{(b+1)}{2}$$

- Proposition: La hauteur d'un arbre (a,b) est $\Omega(\log n / \log b)$ et $O(\log n / \log a)$

- L'insertion et la suppression dans un arbre (a,b) sont une généralisation de l'insertion et la suppression dans un arbre (2,4).

9) B-arbre

- Un B-arbre d'ordre d est un arbre (a,b) avec

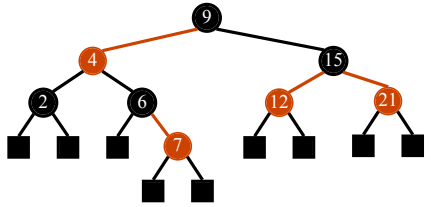
$$a = \lceil \frac{d}{2} \rceil \quad \text{et} \quad b = d$$

- On choisit d de sorte que la taille des blocks B de la mémoire externe soit suffisante pour garder en mémoire d-1 entrées et les références aux d fils d'un noeud, i.e d est $\Theta(B)$

- La hauteur de l'arbre est en $O(\log n / \log a) \overset{\text{ici}}{=} O(\log_2 n / \log_2 B) = \log_B n$

10) Arbre rouge-noir

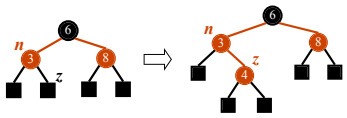
- Un arbre rouge-noir peut aussi être défini comme étant un arbre binaire de recherche qui satisfait les propriétés suivantes:
  - Propriété de racine: La racine est **noir**
  - Propriété externe: Les noeuds externes sont **noirs**
  - Propriété interne: Les enfants d'un noeud **rouge** sont **noirs**
  - Propriété de profondeur: Tous les noeuds externes ont la même profondeur **noir**, qui est définie comme étant le nombre d'ancêtre interne **noir**



- La hauteur d'un arbre rouge-noir gardant en mémoire n éléments est en O(log n)

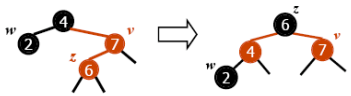
Arbre rouge-noir (suite)

- 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 $O(\log n)$
    - On colore rouge le nouveau noeud insérer, sauf si c'est la racine
    - Il est possible que l'insertion cause un double-rouge:



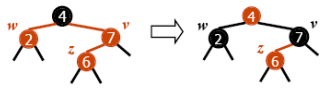
- Considérons un **double rouge**: v, le parent **rouge**, z, le fils **rouge** et considérons w, le frère de v.

Cas 1: w est **noir**
- Restructuration:



- La propriété interne est restorée et les autres propriétés sont préservées

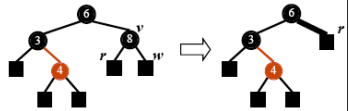
Cas 2: w est **rouge**
- Recoloration:



- Après une recoloration, il est possible que le double-rouge se propage

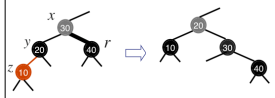
Arbre rouge-noir (suite)

- Complexité en temps des opérations principales (suite):
  - L'algorithme de suppression prend un temps $O(\log n)$
    - Chercher le noeud interne v et le noeud externe w qui seront supprimer par l'opération $O(\log n)$
    - Soit r, le frère de w. Si v ou r était rouge, on colore r noir et on a terminé
    - Sinon, le suppression va créer un double-noir



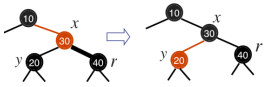
- Soit r le noeud **double noir**. Soit y, le frère de r.

Cas 1: y est **noir** et a un fils z **rouge**
- Restructuration:



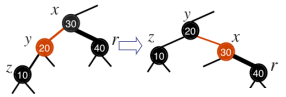
- À la suite de la restructuration, toutes les propriétés des arbres rouges-noirs sont rétablies.

Cas 2: y est **noir** et les deux fils de y sont **noirs**
- Recoloration:



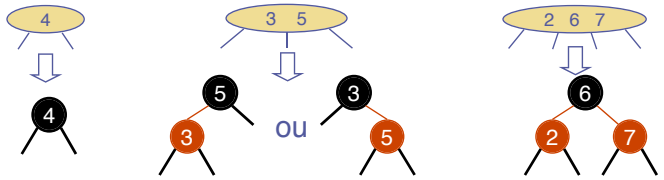
- La recolaration peut causer un problème de **double noir** chez le parent de r.

Cas 3: y est **rouge**
- Ajustement:



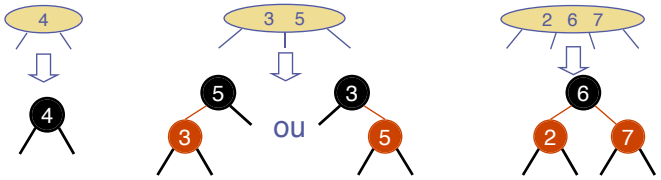
- L'ajustement va nous ramener soit dans le cas 1, soit dans le cas 2

Correspondance rouge-noir - (2,4)



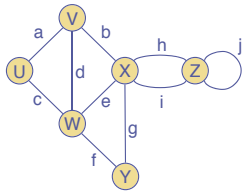
Insertion: Remédier à un double rouge		
Opérations arbres rouge-noir	Opérations arbres (2,4)	Résultats
Restructuration	Changement de représentation d'un 4-noeud	Le double rouge est enlevé
Recoloration	Fractionnement	Le double rouge est enlevé ou il se propage vers le haut

Correspondance rouge-noir - (2,4)



Suppression: Remédier à un double noir		
Opérations arbres rouge-noir	Opérations arbres (2,4)	Résultats
Restructuration	Transfert	Le double noir est enlevé
Recoloration	Fusion	Le double noir est enlevé ou il se propage vers le haut
Ajustement	Changement de représentation d'un 3-noeud	Suivi d'une restructuration ou d'une recoloration

11) Graphe + structure de données

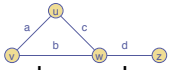


- Extrémités d'une arête
- Arêtes incidentes à un sommet
- Sommets adjacents
- Degré d'un sommet
- Chemin
- Arêtes multiples:
- Boucle
- Cycle

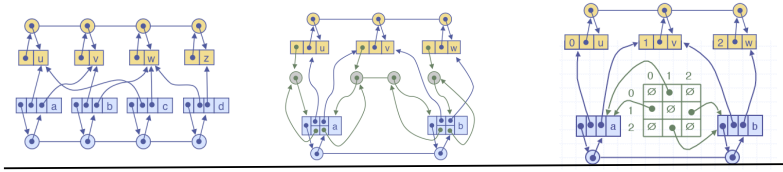
Propriété 1: $\sum_{s \in S} deg(s) = 2m$

Propriété 2: Dans un graphe non-orienté n'ayant aucune boucle ou arête multiple, on a

$$m \leq \frac{n(n-1)}{2}$$



- Structures de données pour les graphes:
  - Liste d'arêtes
  - Liste d'adjacence
  - Matrice d'adjacence



Graphe + structure de données (suite)

Performance

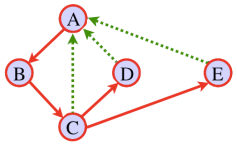
n sommets, m arêtes aucune arête-multiple aucune boucle	Liste Arêtes	Liste d'adjacence	Matrice d'adjacence
Espace	$n + m$	$n + m$	n^2
incidences(v)	m	deg(v)	n
sontAdjacent(v,w)	m	min(deg(v), deg(w))	1
insèreSommet(o)	1	1	n^2
insèreArête(v, w, o)	1	1	1
enlèveSommet(v)	m	deg(v)	n^2
enlèveArête(e)	1	1	1

12) Parcours de graphe

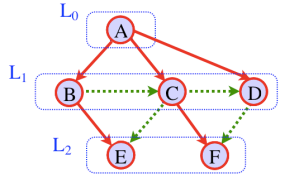
- Parcours en profondeur (DFS)
- Parcours en largeur (BFS)

- Visite tous les sommets et toutes les arêtes de G
- Détermine si G est connexe ou non
- Calcule les composantes connexes de G
- Calcule une forêt couvrante pour G
- Complexité en temps O(n+m) si on utilise la structure de liste d'adjacence
- Trouve un cycle dans un graphe

- Trouve et retourne un chemin entre deux sommets



- Trouve et retourne un chemin de longueur minimale entre deux sommets



13) Chemin de poids minimal

Algorithme de Dijkstra

- L'algorithme de Dijkstra est un algorithme glouton qui calcule les distances entre un sommet v et tous les autres sommets d'un graphe
- On va donner une étiquette d(u) à chaque sommet, représentant la distance entre v et u dans le sous-graphe constitué des sommets dans le nuage et des arêtes adjacentes
- À chaque étape:
  - On ajoute au nuage le sommet u extérieur au nuage qui a la plus petite étiquette d(u)
  - On met à jour les étiquettes des sommets adjacents à u (relaxation des arêtes)
- Fonctionne seulement avec des arêtes de poids non-négatifs
- Complexité O((n+m) log n)