

# Indexation

Patricia Serrano Alvarado

# L'indexation, what for?

---

- Si les tables sont très grandes la recherche séquentielle est très chère

- Comment obtenir « vite » les enregistrements satisfaisant un prédictat ? Supposant qu'uniquement 20% de enregistrements satisfont la requête :

```
SELECT *  
FROM MOVIES  
WHERE studioName='Disney' AND year = 1990;
```

- Comment faire « vite » les jointures ?

```
SELECT name  
FROM Movies, MovieExec  
WHERE title='Stars Wars' AND producerC#=cert#;
```

# Indexes

---

- Un index est une structure de données qui associe à une valeur d'un ensemble d'attributs, l'adresse (ou les adresses/pointeurs) des enregistrements contenant cette valeur.
- L'ensemble d'attributs est appelé clé de l'index

# Sélection d'indexes

- L'existence d'un index sur un attribut peut accélérer l'exécution de requêtes portant sur cet attribut (valeurs/rang de valeurs et jointures)
- MAIS les indexes rendent les insertions, suppressions et modifications plus complexes et chères en temps
- Indexes sur les clés primaires (**index primaire**)
  - Attributs utilisés très souvent dans les requêtes
  - L'unicité fait qu'un index retourne soit une adresse (correspondant à l'enregistrement) soit rien.

# Suite...

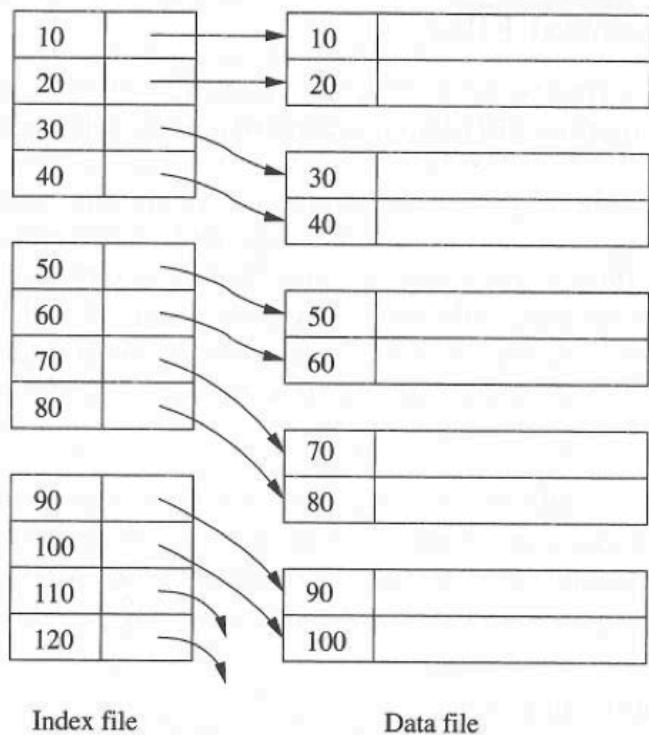
---

- Indexes sur les attributs qui ne sont pas de clés (**indexes secondaires**), intéressants si
  - L'attribut est presqu'une clé (peu d'enregistrements ont la même valeur pour cet attribut)
  - Si les enregistrements sont clustérisés : groupage des tuples avec une même valeur pour un attribut dans le même bloc si possible

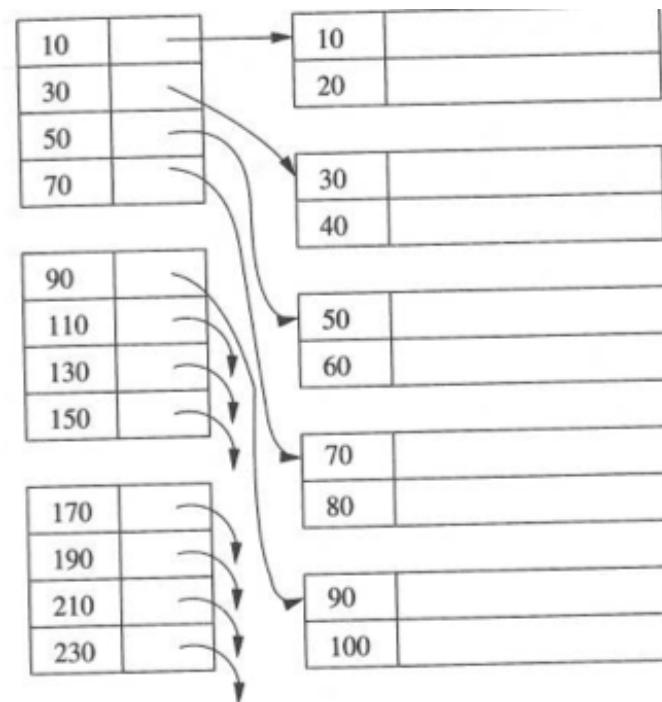
# Concepts de base

- Fichiers séquentiels
  - Les enregistrements d'une relation sont ordonnées par **leur clé primaire**
  - Les enregistrements sont distribués entre blocs avec cet ordre
  - La **recherche binaire** peut être utilisée avec un coût de  $\log_2 n$
- Index dense
  - Séquence de blocs avec toutes les clés des enregistrements et leurs pointeurs
- Index clairsemé
  - Séquence de blocs avec une clé et pointeur par bloc indexé
  - **Atout** : si relation très grande l'index peut comme même tenir dans la mémoire principale

# Exemples



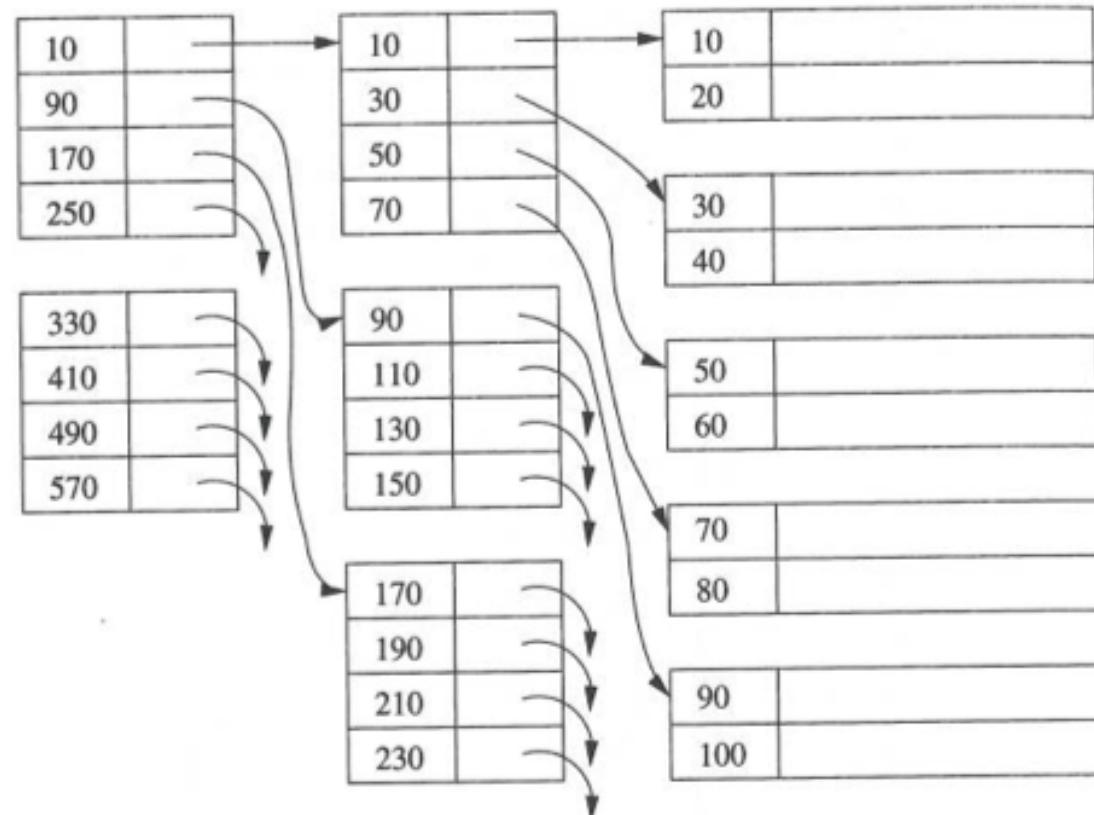
Index dense



Index clairsemé

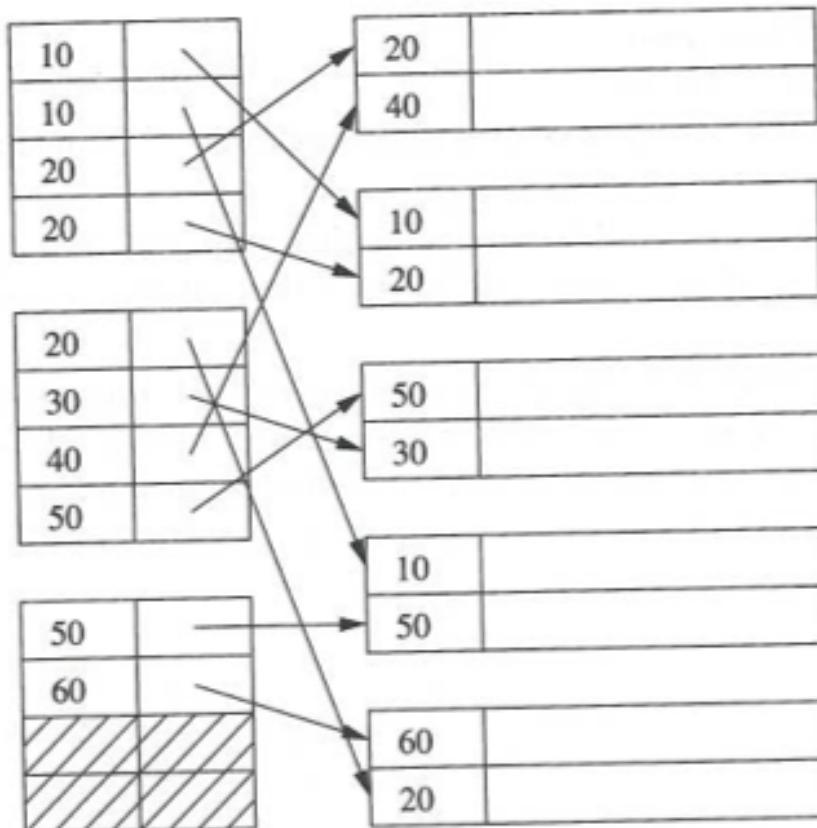
# Index multiniveaux

- Index sur l'index pour accélérer la recherche
- Le première niveau peut être dense ou clairsemé mais les suivants doivent être clairsemés



# Indexes secondaires

- Donnent la localité exacte des enregistrements, celle ci est décidée par les clés primaires
- Ils sont toujours denses
- Moins efficaces que les indexes primaires
- Très utiles pour les fichiers clustérisés

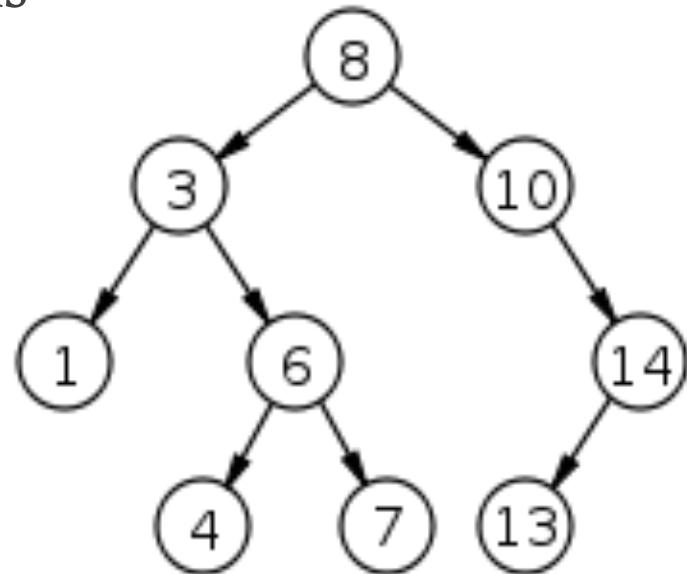


# Indexes Arbres B

- Comme un index mult;niveaux
- Pas besoin d'avoir un fichier trié
- La variante Arbre B+ est la plus implémentée dans les SGBD
- Les arbres B+
  - Sont équilibrés
  - Maintiennent autant de niveaux que nécessaire pour le fichier à indexer
  - Chaque nœud est un (sous)index
  - Chaque niveau est soit **rempli à moitié** soit **rempli complètement**
  - Se réorganisent dynamiquement

# Rappel sur Arbre binaire

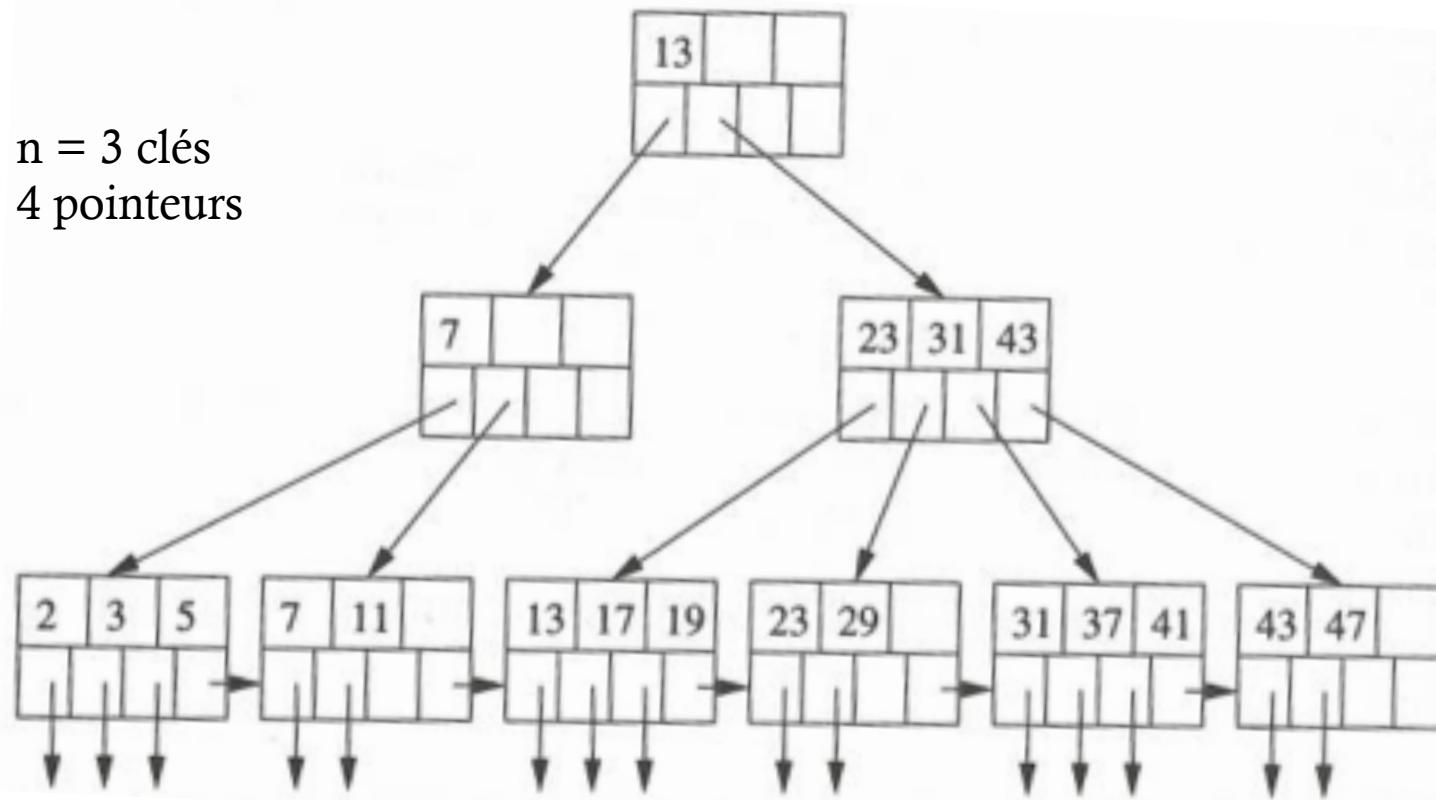
- Chaque nœud possède une clé et 2 fils
  - Le fils de gauche a une clé inférieure
  - Le fils de droite a une clé supérieure
- Coût d'insertion et de recherche
  - $O(\log n)$  en moyenne,
  - $O(n)$  au pire des cas si l'arbre s'est converti en une liste chaînée,
- Coût de suppression
  - Parcours jusqu'à une feuille, au pire des cas  $O(n)$



# Arbre B+

- Chaque **nœud** est stocké dans un **bloc**, la capacité des blocs détermine la forme de l'arbre
- La **racine** a au moins 2 pointeurs (et une clé)
- Les **nœuds** intérieurs pointent vers les blocs du niveau suivante et au moins  $\lceil (n+1)/2 \rceil$  pointeurs doivent être utilisés
- Toutes les clés sont dans les feuilles de l'arbre et chaque clé a un **pointeur** vers l'enregistrement correspondant
- Les **clés** dans les feuilles sont réparties et ordonnées de gauche à droite et au moins  $\lceil (n+1)/2 \rceil$  pointeurs doivent être utilisés
- Dans chaque feuille, un **pointeur supplémentaire** (le dernier) pointe vers la feuille suivante à droite (vers le bloc avec la suite de clés)

# Un Arbre B+



# Taille d'un arbre B+

- Un paramètre n est associé à chaque arbre, il détermine la taille et forme des blocs
  - Chaque bloc a espace pour n clés et n+1 pointeurs
  - n doit être aussi grande que possible selon la taille d'un bloc
  - Exemple, soit :
    - la taille d'un bloc de 4096 octets
    - les clés sont des entiers de 4 octets
    - les pointeurs des entiers de 8 octets
- Alors  $4n + 8(n+1) \leq 4096$  ca fait  $n=340$

# La recherche dans un Arbre B+

- L'enregistrement recherché est  $K$
- La recherche récursive commence à la racine et quand on arrive aux feuilles on trouve le pointeur de l'enregistrement
- Sur tous les nœuds on fait la même comparaison pour savoir quel nœud fils doit être examiné ensuite
- Supposons un nœud avec clés  $K_1, K_2, \dots, K_n$ 
  - si  $K < K_1$ , le fils à examiner est le premier
  - Autrement si  $K_1 \leq K < K_2$ , le fils à examiner est le second fils, et ainsi de suite.
  - Lorsque le nœud est une feuille, si on trouve  $K$  alors on obtient le pointeur correspondant, si on ne trouve pas  $K$  alors la clé n'existe pas dans l'arbre.

# L'insertion dans un arbre B+

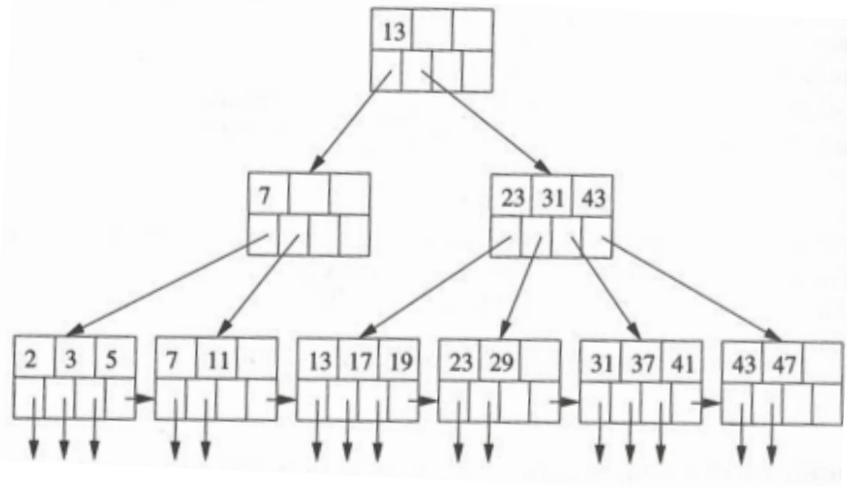
---

1. D'abord on fait une recherche pour trouver la place de la nouvelle clé
2. Itérativement, on essaie d'insérer la nouvelle clé dans le nœud (feuille si la première fois) correspondante s'il y a de la place
3. S'il n'y a pas de place, le nœud est découpé en 2 et les clés sont réparties entre les 2 nœuds (chaque nœud est maintenant à moitié plein)

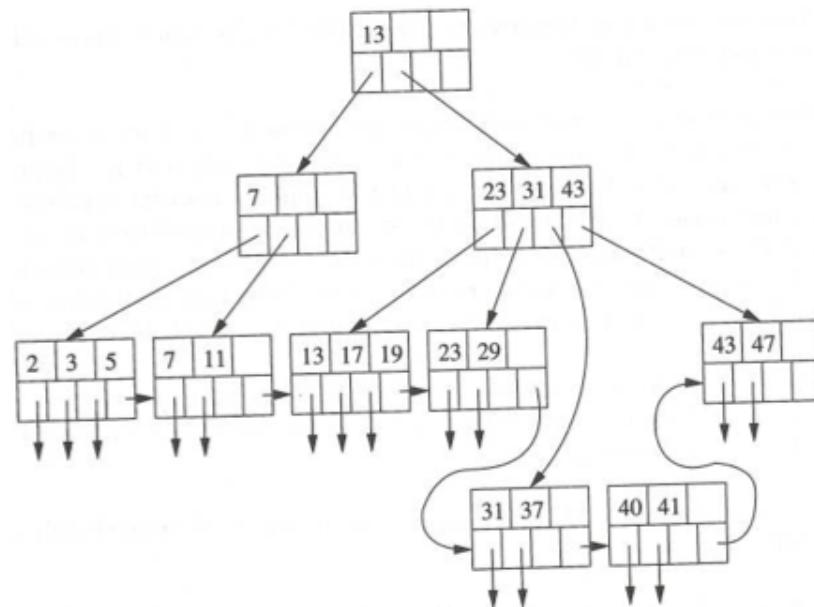
Ce découpage fait que le nœud parent du nœud découpé ait besoin d'avoir un nouveau pointeur

4. Et on recommence à 2, et ainsi de suite jusqu'à la racine.

# Insertion de la clé 40

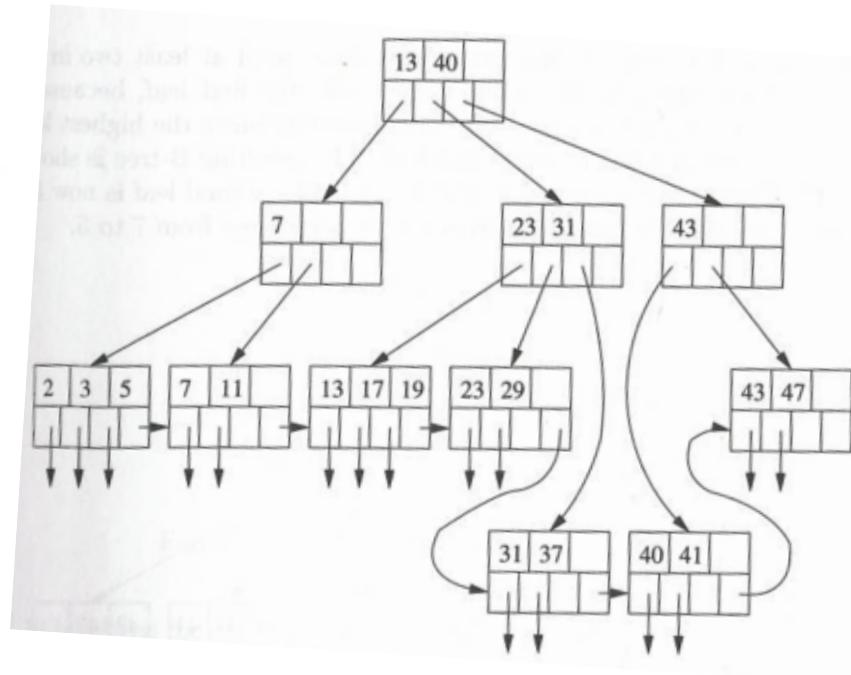
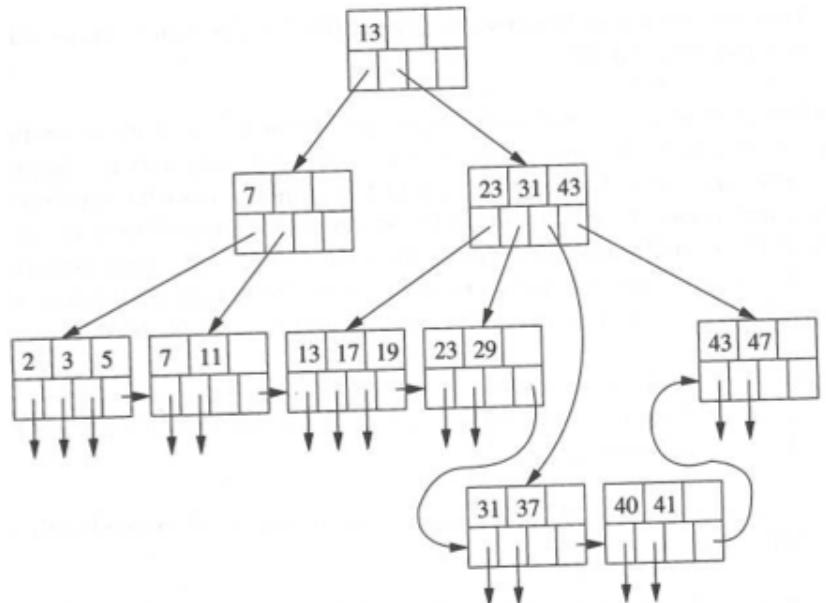


Etat initial de l'arbre



Début de l'insertion de 40

# Suite

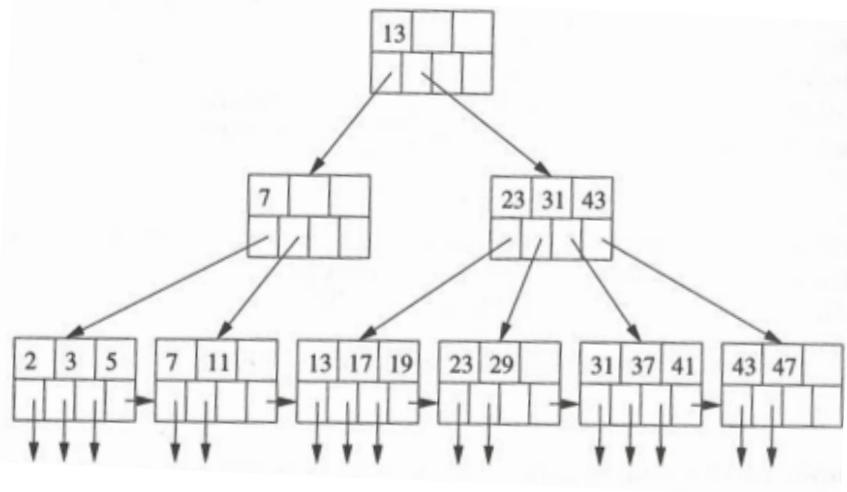


Etat final de l'arbre

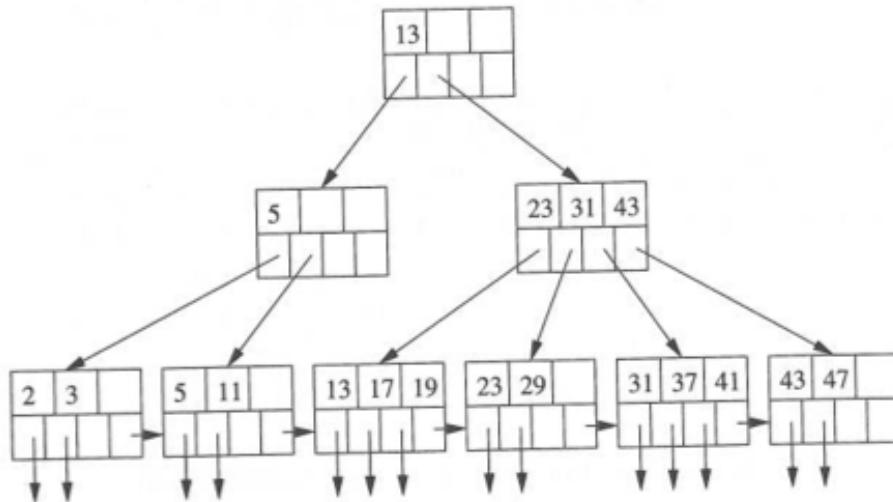
# Supprimer dans un arbre B+

1. D'abord on fait la recherche du nœud qui a la clé à supprimer, supposons que c'est le nœud N
2. Ensuite, on supprime la clé, son pointeur et l'enregistrement correspondant
3. Si le nœud N a encore le nombre minimal de clés obligatoires  $\lfloor (n+1)/2 \rfloor$  la suppression est terminé
4. Autrement, si un nœud frère adjacent de N, supposons M, a plus de clés que le minimal, alors on déplace une clé de M vers N et le nœud parent de M doit possiblement réajuster ses clés et la suppression est terminée
5. Si M est au nombre minimal de clés alors on merge N et M, les clés du parent doivent être réajustées et une clé doit être supprimée. Ensuite on recommence à 3 avec le nœud parent comme N.

# Suppression de la clé 7

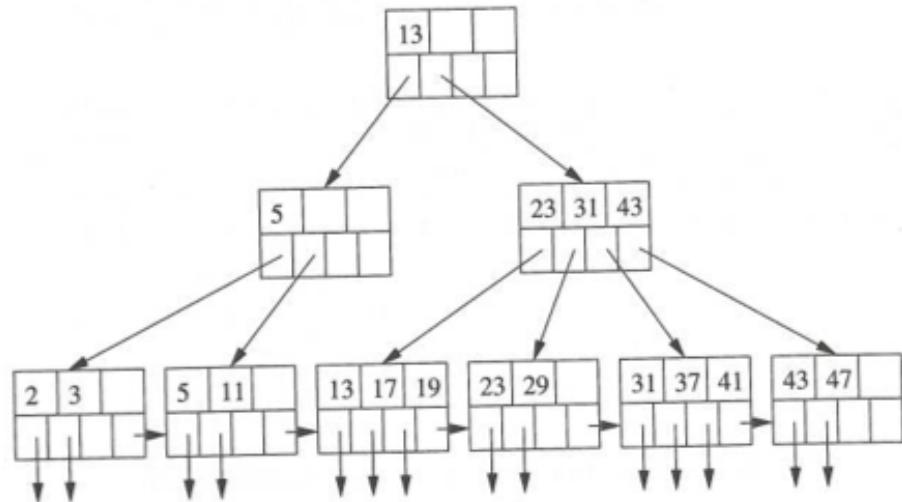


Etat initial de l'arbre

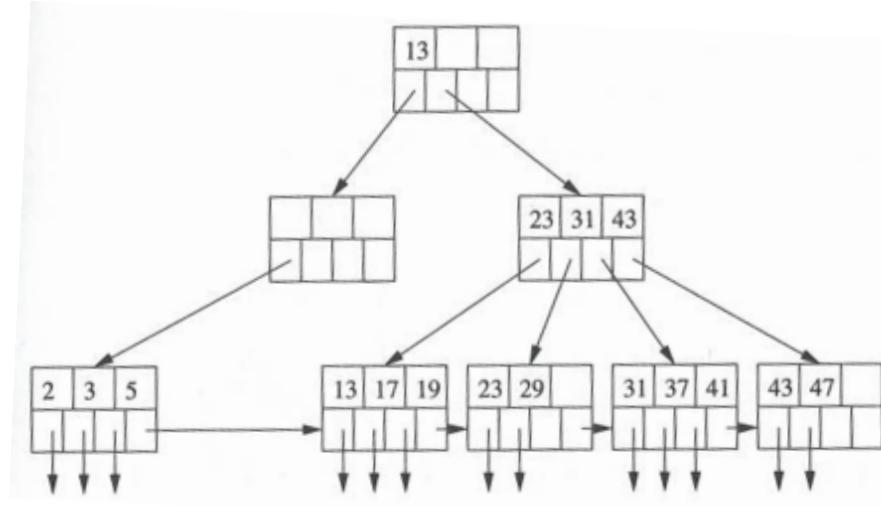


Suppression de la clé 7

# Suppression de la clé 11

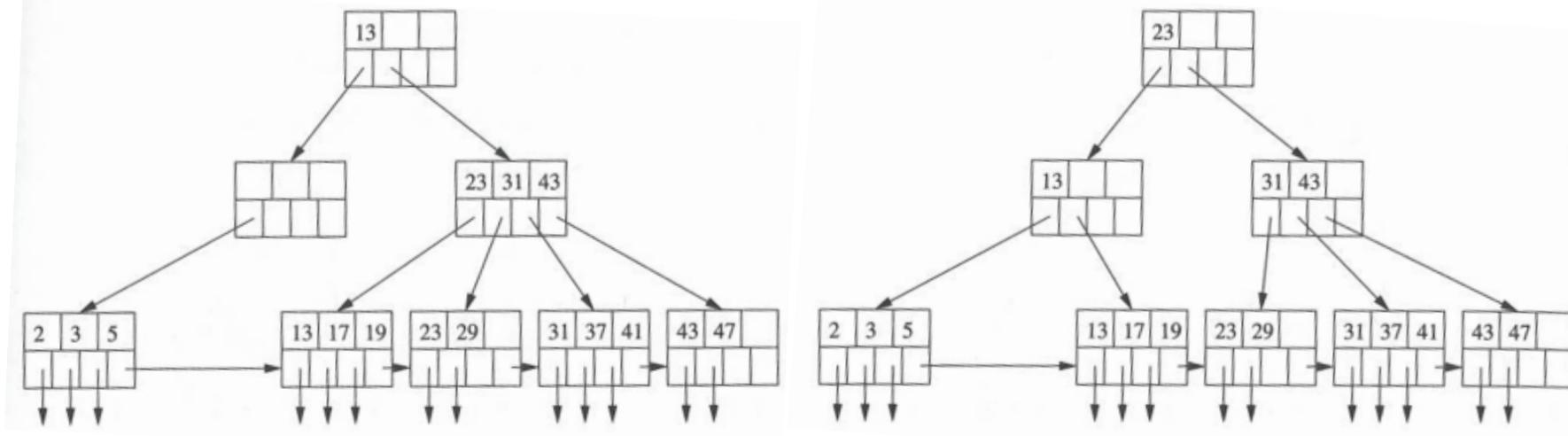


Etat initial de l'arbre



Début de la suppression de 11

# Suite



Etat final de l'arbre

# Avantages des B-arbres

---

- Restent équilibrés.
- Les blocs sont partiellement vides pour accélérer les insertions et les suppressions
- Si  $n$  est suffisamment large, la réorganisation de noeuds (découpage et merge) est rare et souvent cela reste au niveau des feuilles
- Dans chaque bloc une recherche binaire peut être faite
- Excellentes performances pour
  - l'interrogation et l'update : nombre de I/O = nombre de niveaux + 1
  - insert et delete : coût d'une interrogation+ le coût de réorganisation
- Les performances ne se dégradent pas lorsque les tables grossissent.
- Efficient pour recherches sur un rang de valeurs ( $<,>$ ) et l'égalité

# Index bitmap

- Idéal pour
  - Des attributs qui ont peu de valeurs distincts (sexe, situation familiale, etc.)
  - Des colonnes avec un nombre important de lignes (entrepôt de données)
  - Des données qui ne sont pas modifiées fréquemment (pas performant dans les systèmes OLTP)
- Une valeur d'index est associée à une suite de bits
  - Chaque bit correspond à un enregistrement
  - Si le bit est à 1, l'enregistrement contient la valeur d'index
  - L'accès s'effectue par la lecture de l'index
- Avantages
  - Meilleures performances (que les arbres B+) du fait de l'utilisation d'opérations binaires (AND, OR, XOR, NOT)
  - Plus compact que les arbres B+
  - Structure qui se prête bien à la compression (ceci facilite une lecture rapide)

# Exemple d'index bitmap

- La table Personnes

Ligne	Nom	Sexe	Age
1	Martin	Homme	30
2	Monroe	Femme	25
3	Sosa	Homme	22
4	Zola	Femme	18

- La table Géographie de France

Row	City	Region
1	Lille	North
2	Nancy	East
3	Nantes	West
4	Rennes	West
5	Montpellier	South
6	Dunkerque	North

- Index à 2 bits pour l'attribut sexe

Ligne	Homme	Femme
1	1	0
2	0	1
3	1	0
4	0	1

Row	North	East	West	South
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	1	0
5	0	0	0	1
6	1	0	0	0

- Index à 4 bits pour l'attribut région

# Les mises à jour et les index

---

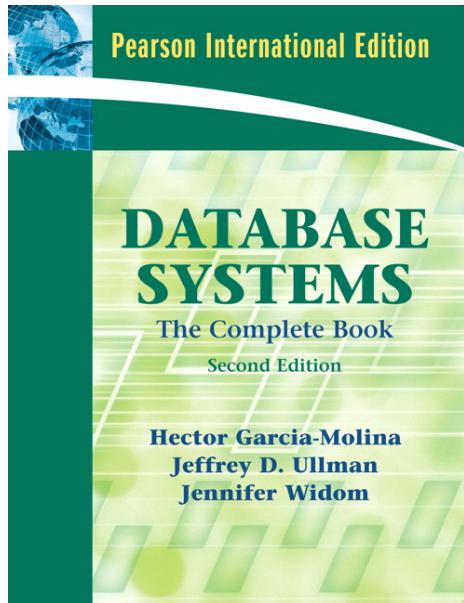
- Chaque modification de l'ensemble d'enregistrements implique une mise à jour des index.
- Les index servent à optimiser, mais il faut vérifier que les performances ne sont pas dégradées par la présence de nombreux index (ou/et mal définis)

# Création d'un index : syntaxe

---

- CREATE [BITMAP] INDEX <nomi> ON  
<nom\_table>(<attr1> [ASC,DESC], ... <attrn>  
[ASC,DESC]) [propriétés de l'index]
- Syntaxe simplifiée, pour la création d'un index sur une table.
- Ex : create index index\_labo on robots (labo desc) ;

# Références



Database management systems.  
Third edition. Raghu Ramakrishnan,  
Johannes Gehrke. Mc Graw Hill,  
2003.

Database systems. The complete book, Second edition. Hector Garcia Molina, Jeffrey D. Ullman, Jennifer Widom. Pearson International Edition, 2009.

