Bases de données avancées

Gestion physique et **Optimisation de requêtes.**

Plan : Gestion Physique.

1. Introduction
2. Hachage
    1. Principe
    2. Hachage statique
    3. Hachage dynamique
3. Méthodes d'accès indexées proprement dites
    1. Principe
    2. Arbres B
    3. Arbres B+
    4. Avantages/Inconvénients
4. Méthodes d'accès multi-attributs.
    1. Index bitmap
    2. Index secondaires.

1. Introduction : remarques préliminaires.

- Minimum de pré-requis :
  - Gestion de la mémoire et des fichiers : cf. cours S.E. et Programmation Système.
  - Algorithmique : complexité, arbres et arbres binaires de recherche. Cf. cours Algo-prog 2 et 3.

- Hypothèse 1 : Tous les enregistrements ont la même longueur.
  - Faux en pratique : les SGBD acceptent des types de longueur variable : binaires, images, texte, etc. (exemple : Varchar, varbinary en mysql).
  - Quelques modifications à faire aux algorithmes présentés pour pouvoir gérer les champs de longueur variable.

1. Introduction : remarques préliminaires.

- E/S sur le disques par blocs complets.
  - Bloc = paquet = page mémoire (ex: 16 K0), lu en une fois, même si seule une partie du bloc est utile.
  - Un bloc = *k* d'enregistrements au plus (taille bloc / taille enregistrement).
  - Vitesse E/S disque <<<< vitesse E/S mémoire.

- Hypothèse 2 : Les enregistrements ont une clé, *attribut particulier permettant d'identifier l'enregistrement*. On recherche les enregistrement ayant une clé particulière.
  - À 99% vrai. Adaptations possibles si pas d'identifiant unique.
  - Méthodes valables aussi pour une clé constituée de plusieurs attributs.

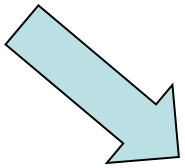
1. Introduction : Problématique.

Exemple : Fichier de *N* =1.000.000 personnes (clé numéro sécu par exemple), un bloc contient 10 personnes au plus.

Chercher une personne ayant une clé particulière ?

Méthode brute = parcourir tout le fichier.

- **100.000 E/S au moins**
- Accès disque couteux…



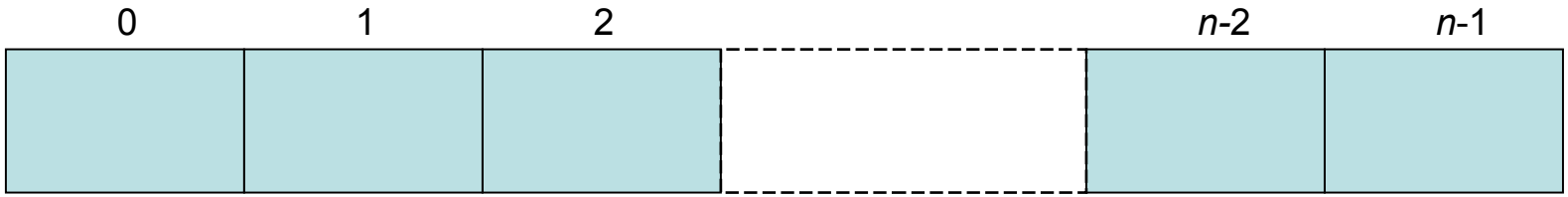
Méthodes permettant de trouver rapidement (en fonction de la clé) le bloc contenant l'enregistrement recherché :

- +- en temps constant dans le cas les plus favorables.
- En *log(N)* le plus souvent.

- => ***une dizaine d'E/S !***

2. Hachage. 1.Principe.

Fichier de *n* blocs



Fonction de hachage : *h(type de la clé)* → *[0, n [*
l'enregistrement de clé *k* doit être rangé dans le bloc de numéro *h(k)*

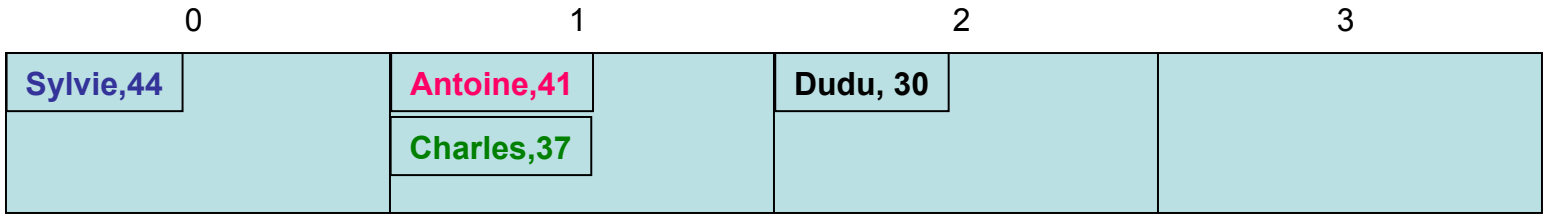
Manipulation de *E*, un enregistrement de clé *k*.

- Insertion
  1. On calcule *x = h(k)*
  2. On rajoute *E* dans le bloc *x*
- Recherche
  1. On calcule *x = h(k)*
  2. On lit le bloc *x* pour y chercher un enregistrement ayant *k* comme clé
- Suppression comme insertion.

=> un seul bloc lu dans l'idéal.

2. Hachage. 1.Principe.

Exemple :



Données :

Nom,	**Age (clé)**
Antoine	41
Charles	37
Sylvie	44
Dudu	30

Fonction de hachage :
h(a:entier) → a modulo 4.

Insertion de Antoine,41 : a) 41 mod 4 = 1 b) Antoine,41 va dans le bloc 1

Insertion de Charles,37 : a) 37 mod 4 = 1 b) Charles,37 va dans le bloc 1

Insertion de Sylvie,44 : a) 44 mod 4 = 0 b) Sylvie,44 va dans le bloc 0

Insertion de Dudu,30 : a) 30 mod 4 = 2 b) Dudu,30 va dans le bloc 2

2. Hachage. 1.Principe.

Bonne fonction de hachage ?

- Rapide à calculer.
- Uniforme.

Problème de l'uniformité :

- Fichier de 26*26 = 676 blocs
- Hachage de chaines de minuscules.
- *h(C:chaine) = 26 *(pos(C(1))-1) + (pos(C(2))-1).* Prend les 2 premières lettres pour calculer la position.

  *exemple : h(« benoit ») = 26 *(2-1) + (5-1) =30*

Sur l'ensemble des chaines de minuscules cette fonction est uniforme.

Mais sur les chaines stockant des noms communs, des prénoms ?

Qui a un prénom commençant par « zz » ?

2. Hachage. 1.Principe.

Remarques générales :

- Si un bloc est très grand et contient beaucoup d'enregistrement ?
  - Mettre un index (voir plus tard) permet de gagner du temps lors d'une recherche d'un enregistrement.
- Simple et rapide si pas de débordement. S'applique à tout type de clé.
- Place perdue, fichier pouvant être très vide.
- Que fait-on en cas de débordement (bloc plein) ?

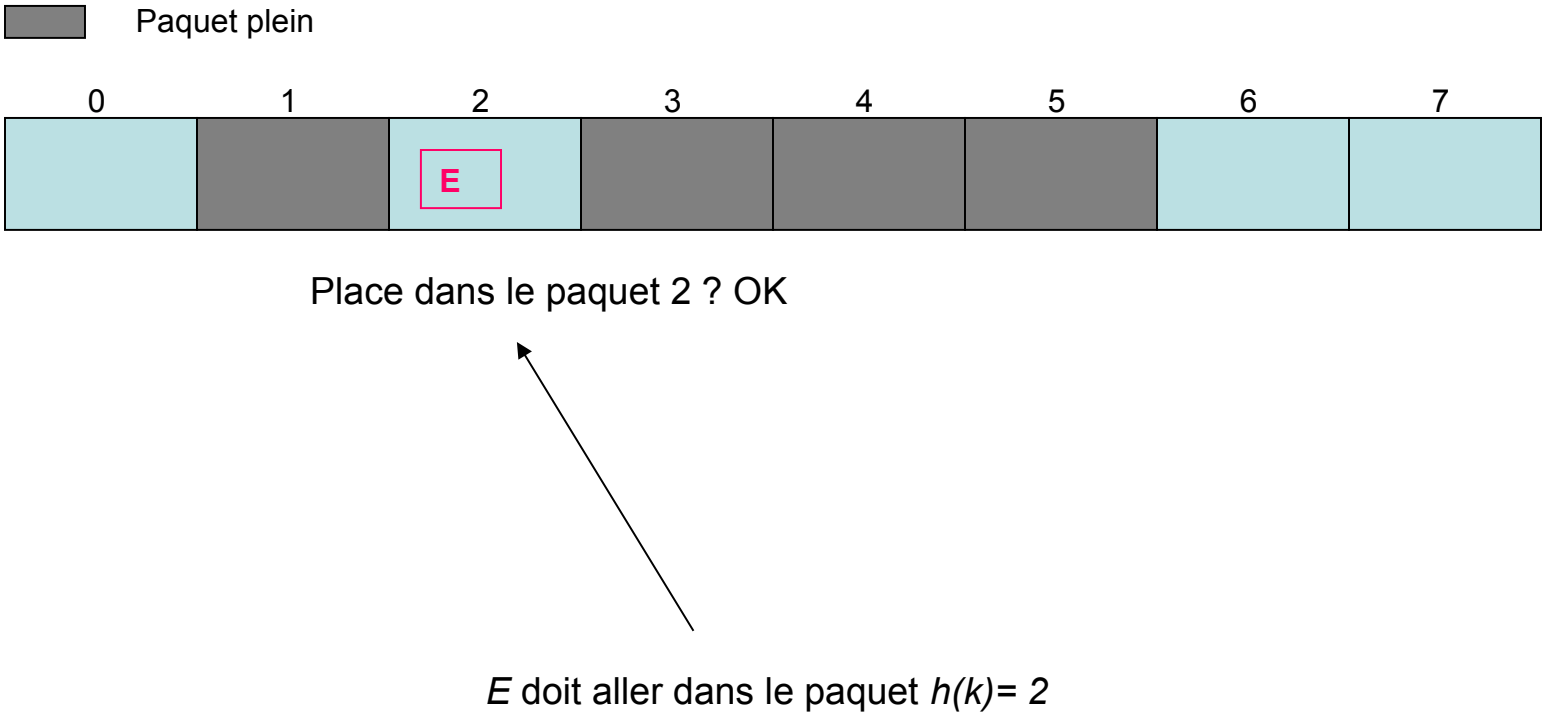
2. Hachage. 2. Hachage Statique

- Taille du fichier reste fixe.
- Organisation du débordement ? Plusieurs solutions

2. Hachage. 2. Hachage Statique. 1. Adressage Ouvert

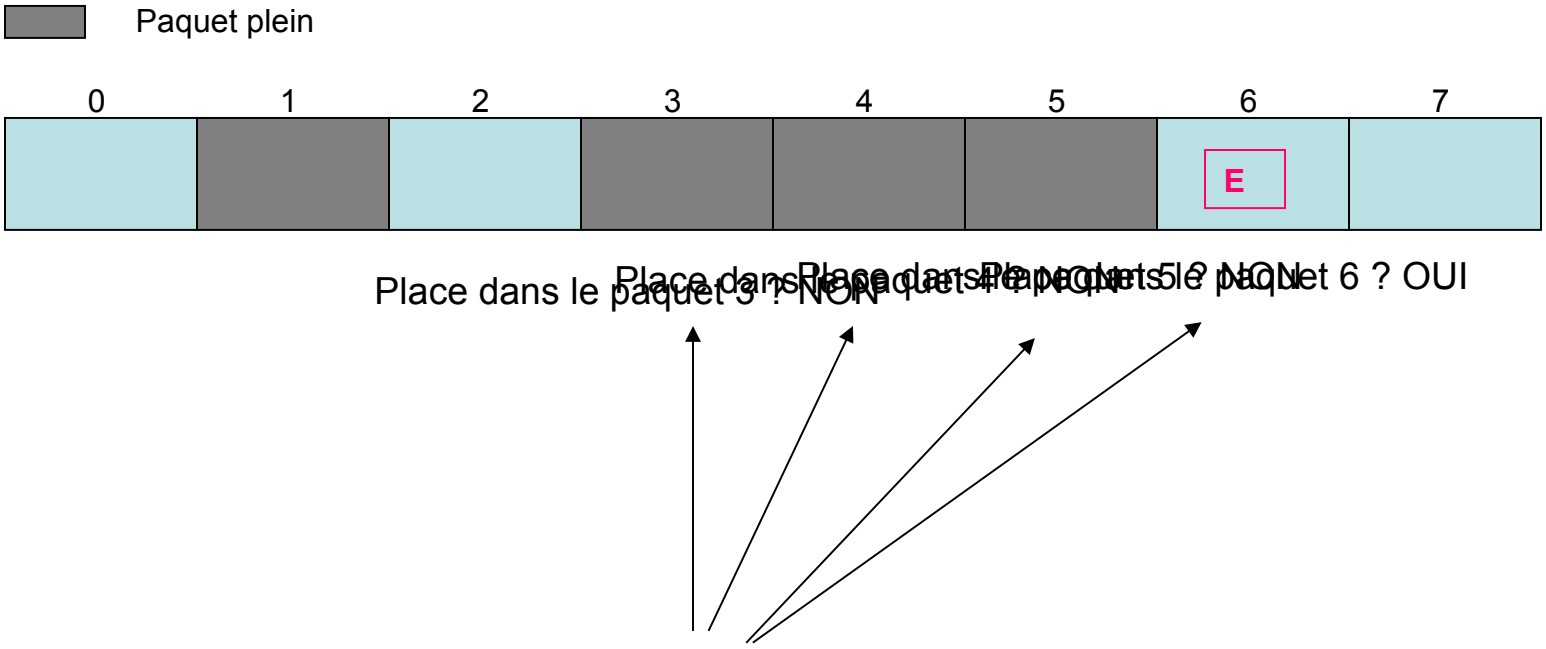
Adressage ouvert : si le paquet est plein, on va voir le suivant.

Insertion :



2. Hachage. 2. Hachage Statique. 1. Adressage Ouvert

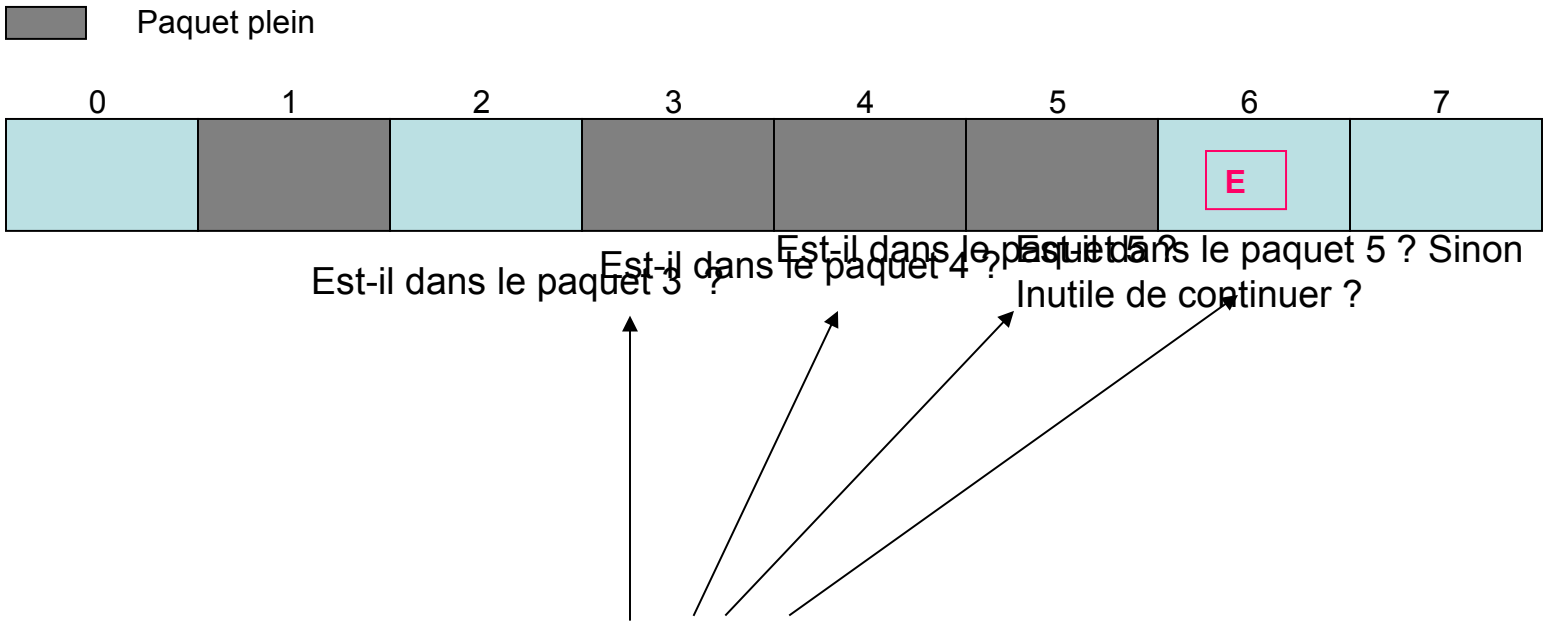
Insertion :



E doit aller dans le paquet *h(k) = 3*

2. Hachage. 2. Hachage Statique. 1. Adressage Ouvert

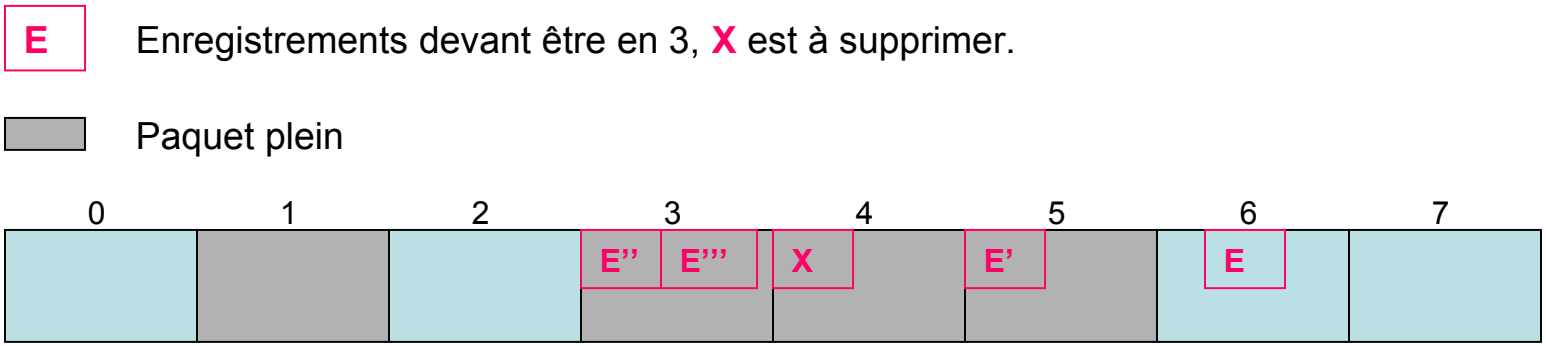
Recherche d'un élément :



E de clé *k* devrait se trouver dans le paquet *h(k) = 3*

2. Hachage. 2. Hachage Statique. 1. Adressage Ouvert

Problème de la suppression/recherche d'un enregistrement :



Suppression de X ? **Que devient la recherche de E ou E'?**

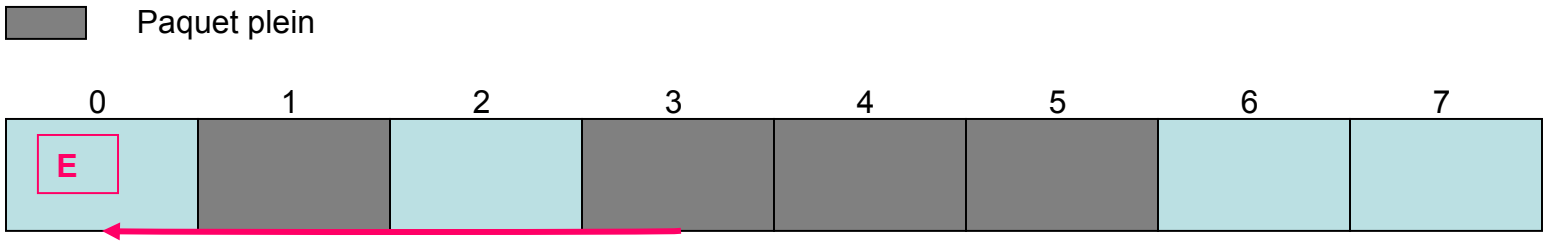
2 possibilités

- Parcourir tout le fichier tant qu'on ne rencontre pas l'élément cherché (recherche inefficace).
- réorganiser le fichier lors de la suppression. (Suppression inefficace)

2. Hachage. 2. Hachage Statique. 2. Adressage Chainé

Adressage Chainé : si le paquet est plein, on cherche de la place ailleurs et on chaine les paquets.

Insertion :



Place dans le paquet 3 ? NON

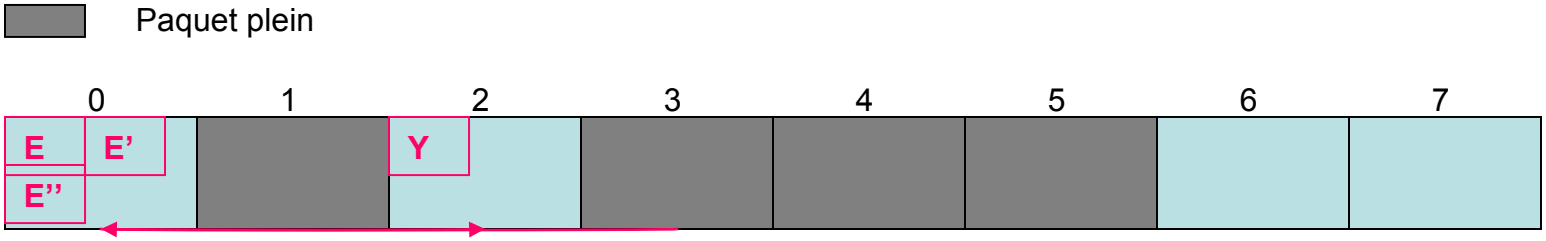
1. Recherche d'un paquet vide => paquet 0.
2. On met E dans le paquet 0 et on chaine

E doit aller dans le paquet *h(k) = 3*

2. Hachage. 2. Hachage Statique. 2. Adressage Chainé

Insertion :

Pour les autres insertions, on suit le chaînage :



Le paquet 0 est plein :

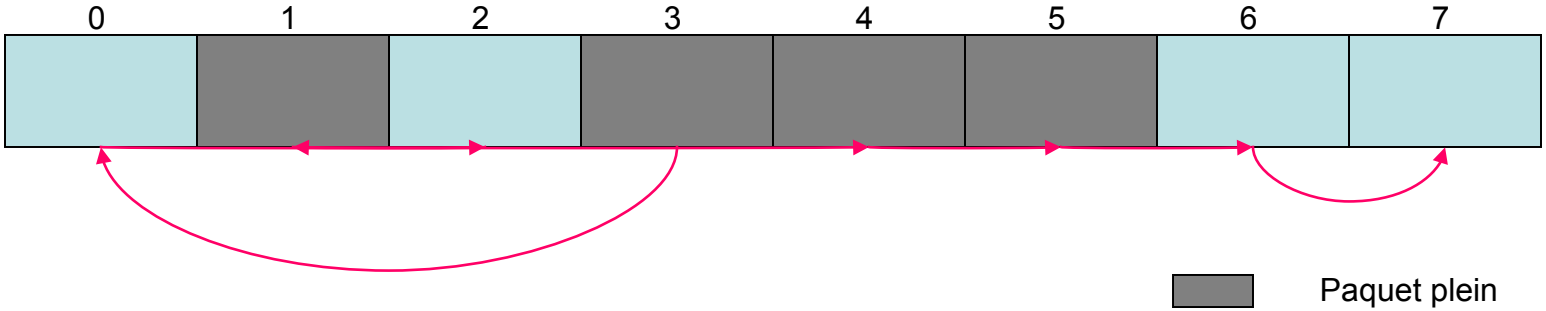
Insertion enregistrement Y de clé *k* dans le paquet 0 car *h(k)=0* ou *h(k)=3*

1. Recherche d'un paquet vide => paquet 2.
2. On met Y dans le paquet 2 et on chaine

2. Hachage. 2. Hachage Statique. 2. Adressage Chainé

Problème de la suppression/recherche d'un enregistrement :

Après des suppressions.



Dans le cas le pire…après plusieurs insertions et suppressions

- Tous les paquets chainés entre eux.
- Même problèmes que pour le chainage ouvert… **Obligé de consulter tous les paquets**

2. Hachage. 2. Hachage Statique. 3. Rehachage

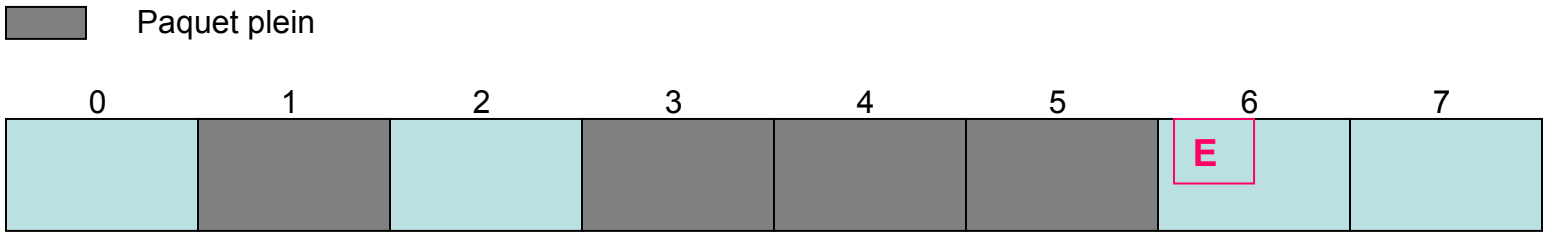
Rehachage :

- Autant de fonctions de hachage différentes que de paquets.

- Ces fonctions $h_0, h_1, h_2 \ldots, h_{n-2}, h_{n-1}$ doivent être telles que
  - Quelques soient une clé ***k*** et un entier ***p <n*** , une des fonctions de hachage ***hi*** est telle que ***hi(k)=p***
  - Autrement dit : quelque soit la clé ***k***, en essayant les ***n*** fonctions de hachage, on passe en revue **tous** les entiers de 0 à n-1
- Exemple :
  - si n paquets $h_i(k) = ((k+i)*P)$ ***modulo n*** où **P** est premier avec **n**
  - Pour 4 paquets
    - ***h0(k) = ((k)*3) modulo 4***
    - ***h1(k) = ((k+1)*3) modulo 4***
    - ***h2(k) = ((k+2)*3) modulo 4***
    - ***h3(k) = ((k+3)*3) modulo 4***

2. Hachage. 2. Hachage Statique. 3. Rehachage

Insertion de *E* de clé *k* :



Calcul de h0(k)= 3 *Paquet 3 plein ?*

Calcul de h1(k)= 1 *Paquet 1 plein ?*

Calcul de h2(k)= 6 *Paquet 6 plein ?*

On met E dans 6.

Pour cette insertion les fonctions h3, h4, h5, h6 et h7 ne sont pas utilisées

2. Hachage. 2. Hachage Statique. 3. Rehachage

Problème de la suppression/recherche d'un élément

Après plusieurs suppressions, recherche de **E** de clé ***k***?

Calcul de h0(k)= 3 *dans le paquet 3 ?*

Calcul de h1(k)= 1 *dans le paquet 1 ?*

Calcul de h2(k)= 6 *dans le paquet 6?*

Calcul de h3(k)= 2 *dans le paquet 2 ?*

Calcul de h4(k)= 7 *dans le paquet 7?*

Calcul de h5(k)= 4 *dans le paquet 4 ?*

Calcul de h6(k)= 0 *dans le paquet 0 ?*

Calcul de h7(k)= 5 *dans le paquet 5 ?*

Elément absent = test de tous les paquets.

Même punition que pour les autres hachages statiques.

Avantage/Inconvénient des méthodes statiques

- Avantages :
  - simple
- Inconvénients :
  - très lent si trop de débordement.
  - obligation de fixer la taille à la création.
  - si le fichier se remplit trop ?
    - obligation de réorganiser complètement le fichier pour pouvoir l'agrandir.

2. Hachage 3. Hachage dynamique. 1. Principe Général

Principe général :

- *h(k)* génère *N* bits (32 par exemple).
- au début seuls les *M* premiers bits sont utilisés pour un fichier de 2^M paquets.
- Lorsqu'un paquet est plein, on :
  1. augmente la taille du fichier à $2^{(M+1)}$ paquets
  2. on utilise les ***M+1*** premiers bits de la clés.
  3. on « éclate » les anciens paquets en répartissant leurs enregistrements entre l'ancienne et la nouvelle partie.

2. Hachage 3. Hachage dynamique 2. Hachage Extensible

Hachage extensible :

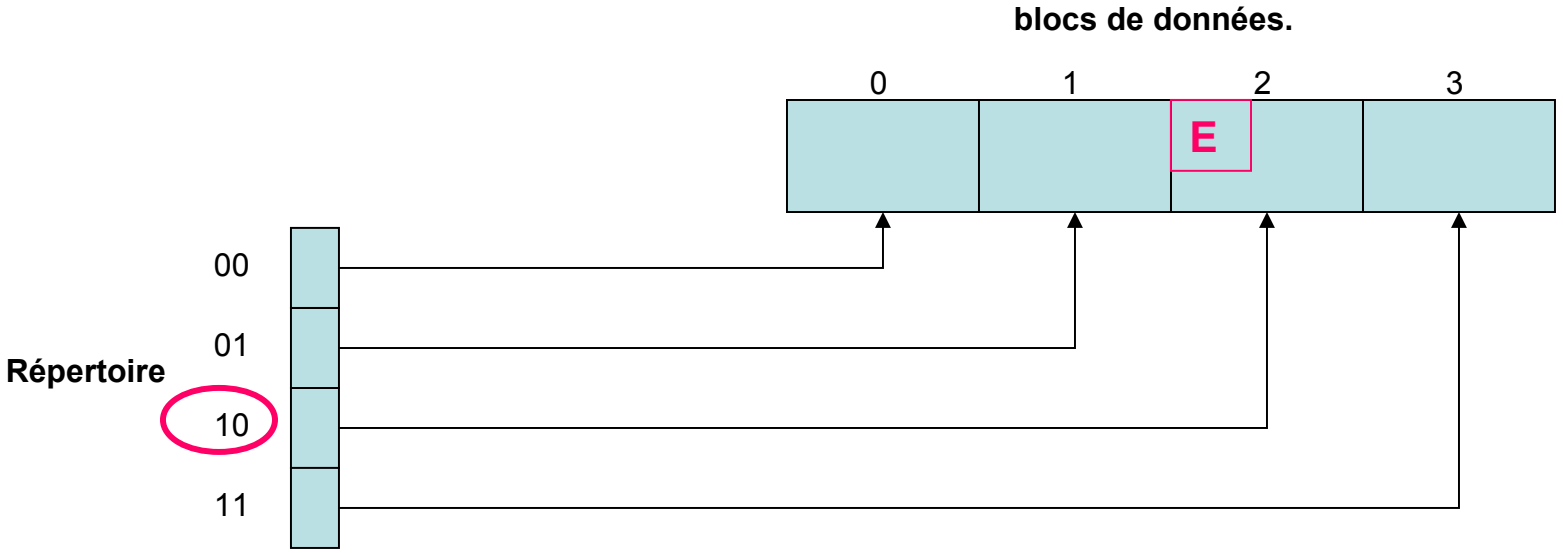
- On éclate le fichier dès qu'un paquet est plein.
- Seul le paquet plein est éclaté. Articles de ce paquet redistribués entre l'ancien et le nouveau paquet en fonction d'un bit supplémentaire :
  - 0 : l'enregistrement reste en place.
  - 1 : l'enregistrement déménage dans le nouveau paquet

Comment faire ?

- On gère un répertoire de taille $2^{(M+N)}$ (N dépend des éclatements déjà effectués).
- Ce répertoire indique le numéro du paquet en fonction des M+N premiers bits de la clé

2. Hachage 3. Hachage dynamique 2. Hachage Extensible

Fichier : répertoire + paquets de données.
Exemple : *M=2* => 4 paquets initiaux.

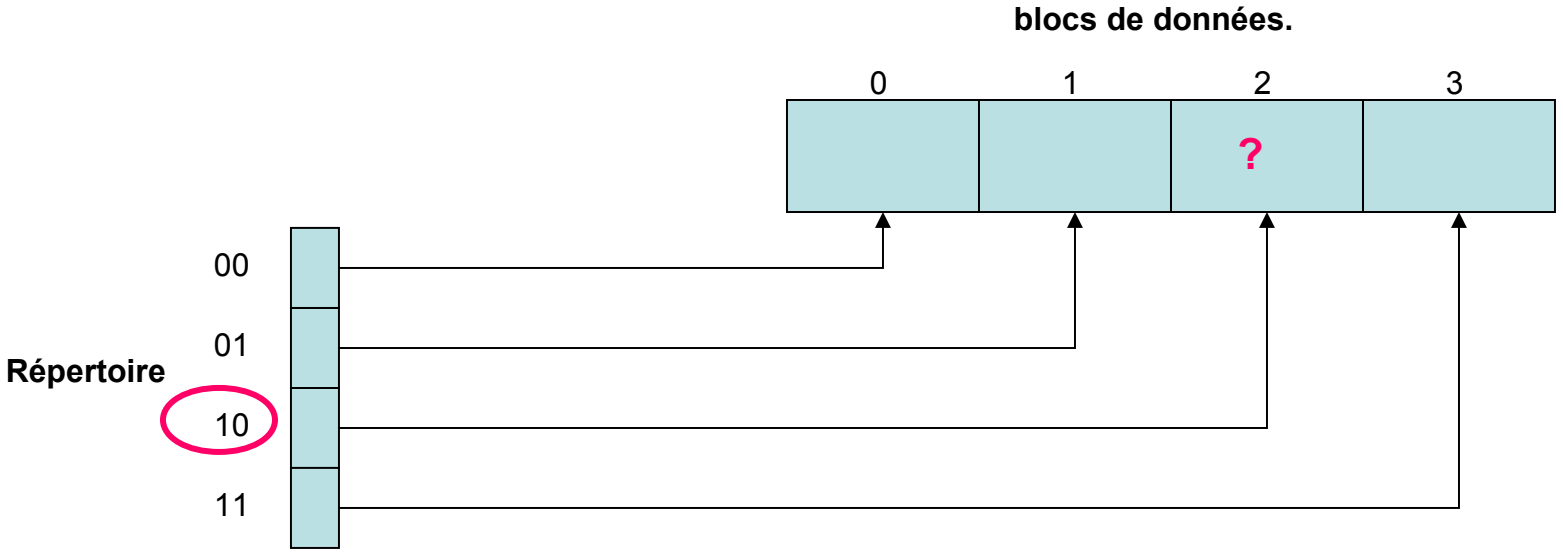


Insertion de *E* de clé *k* ?

Je prends les ***M=2*** derniers bits de h(k) (par exemple 10)

Je range E dans le bloc pointé par la case 10 du répertoire

Fichier : répertoire + paquets de données.
Exemple : *M=2* => 4 paquets initiaux.

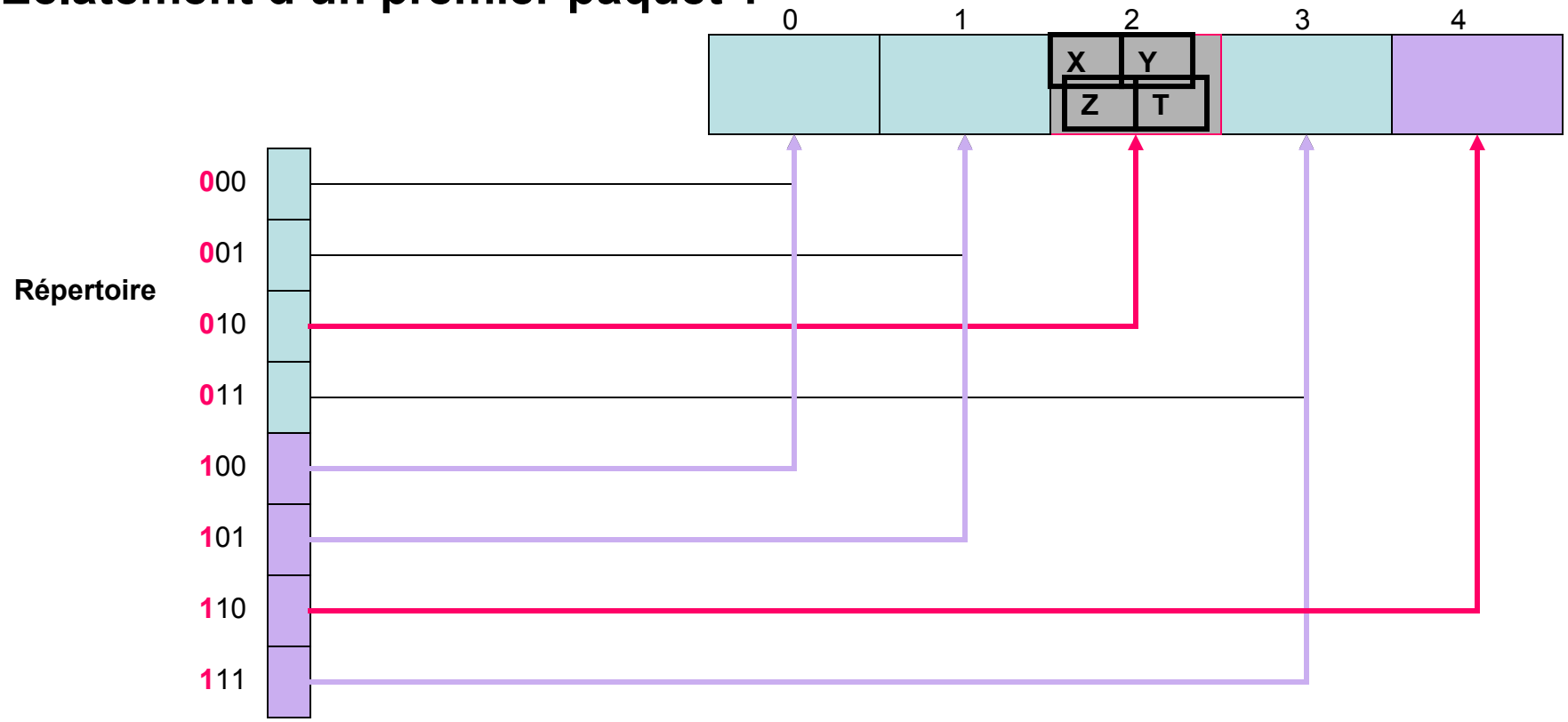


Recherche de *E* de clé *k* ?

Je prends les ***M=2*** derniers bits de h(k) (par exemple 10)

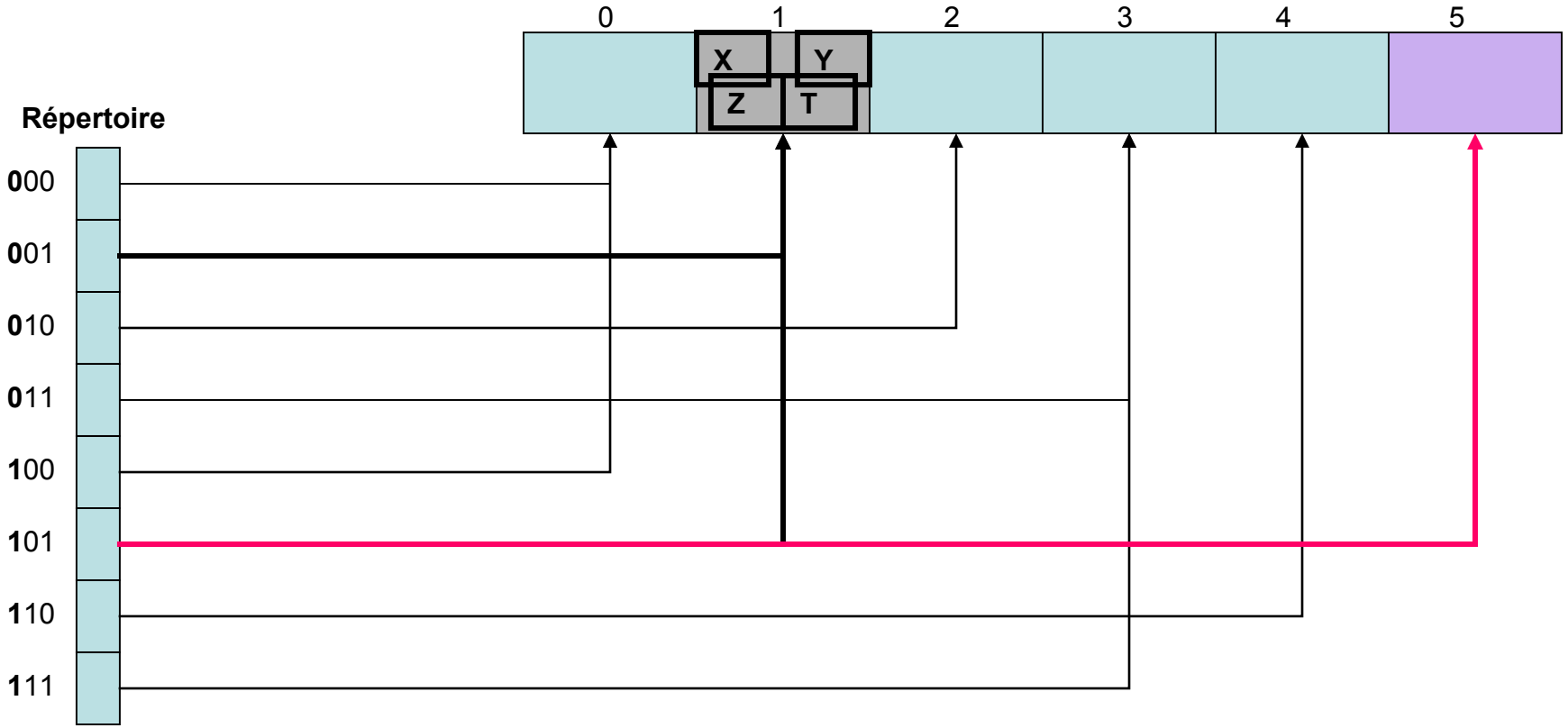
Je regarde dans le bloc pointé par la case 10 du répertoire

Eclatement d'un premier paquet ?



1. On rajoute un nouveau paquet
2. Je multiplie par 2 la taille du répertoire en dupliquant les entrées non éclatées.
3. On pointe le nouveau paquet avec la nouvelle entrée éclatée
4. Je répartis les enregistrement entre les 2 paquets

Eclatement d'un paquet n'ayant jamais éclaté ?



1. On rajoute un nouveau paquet
2. Je modifie le chainage de la moitié des entrées pointant sur ce paquet
3. Je répartis les enregistrements entre les 2 paquets

2. Hachage 3. Hachage dynamique. 2 Hachage Extensible

A chaque instant on doit connaitre :

- ***M*** : nombre initial de bits à prendre en compte. (Au départ le fichier comprenait 2^M blocs)
- ***Qi***: nombre d'éclatements du paquet ***i***.
- **P** : Max des **Qi.** (le répertoire comprend $2^{(M+P)}$ entrées).

2. Hachage 3. Hachage dynamique. 2 Hachage Extensible

Recherche d'un enregistrement de clé k :

1. Je calcule h(k)
2. Je prends les M+P derniers bits pour consulter dans le répertoire le numéro du paquet.
3. Je regarde dans le paquet si l'enregistrement s'y trouve.

2. Hachage 3. Hachage dynamique. 2 Hachage Extensible

Ajout d'un enregistrement saturant le paquet *i* :

1. Je rajoute le nouveau paquet ***x***
2. Si ***Qi < P*** (le paquet ***i*** a déjà éclaté moins de ***P*** fois) alors
    1. partager les entrées du paquet ***i*** entre les paquets ***i*** et ***x*** .
    2. répartir les enregistrements du paquet ***i*** entre les paquets ***i*** et ***x.***
    3. mettre à jour *:* ***Qi ← Qi +1 ; Qx ← Qi +1*** ;
3. sinon (***Qi = P***, le paquet ***i*** a déjà éclaté ***P*** fois). soit ***iii*** l'entrée du paquet ***i***
    1. dupliquer le répertoire et les entrées non concernées.
    2. le nouveau paquet est pointé par l'entrée ***1iii***. l'ancien reste pointé par l'entrée **0iii**.
    3. répartir les enregistrements du paquet ***i*** entre les paquets ***i*** et ***x.***
    4. mettre à jour *:* ***Qi ← Qi +1 ; Qx ← Qi +1*** ; ***P ← P+1***

2. Hachage 3. Hachage dynamique. 2 Hachage Extensible

Suppression d'un enregistrement :

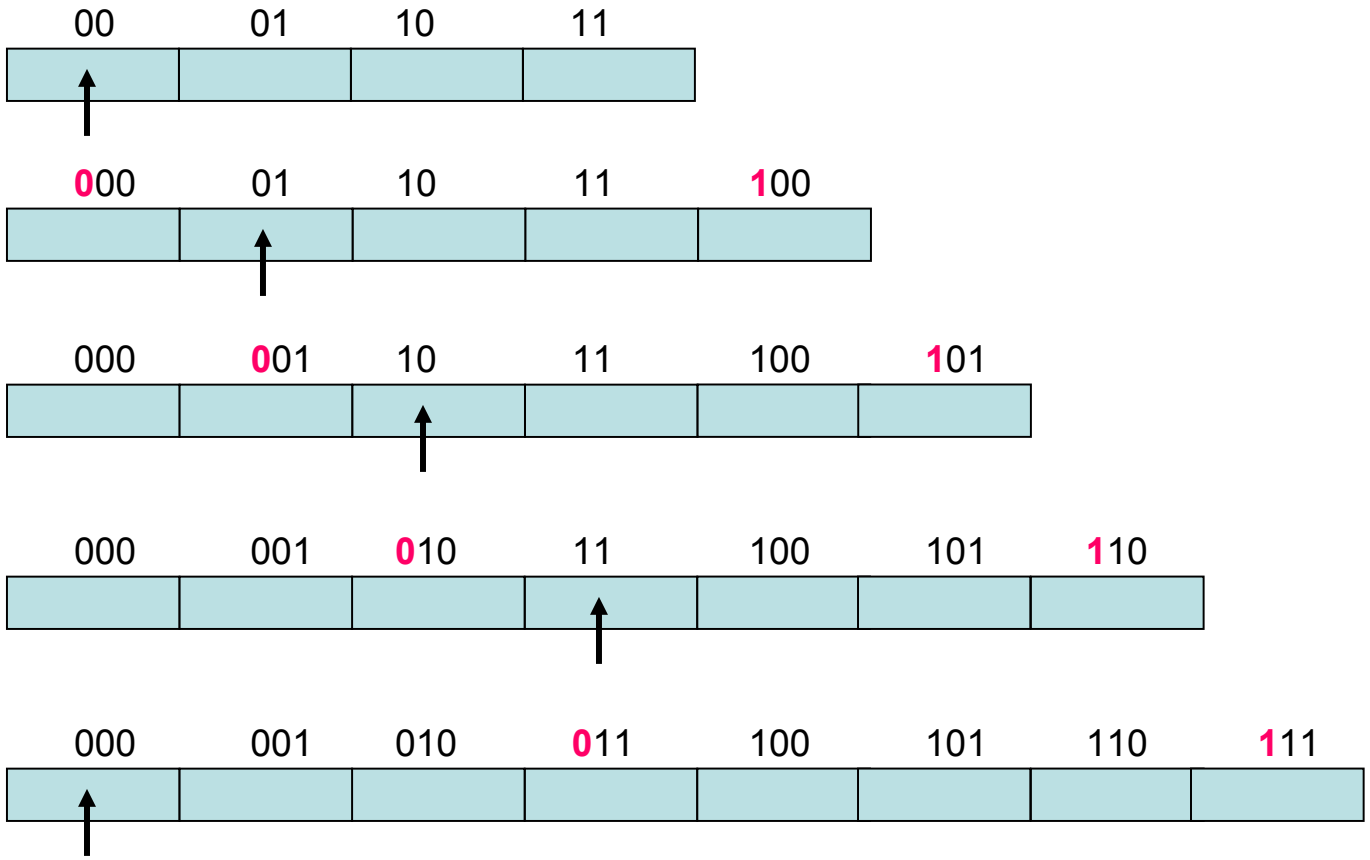
- place récupérée => fusion de paquets ?
- mécanisme possible mais compliqué.
- pas toujours très utile de récupérer la place. Mieux vaut attendre,
  (Compactage à la demande par exemple)

Hachage Linéaire :

- On garde un pointeur indiquant le prochain paquet à éclater. Ce curseur avance d'un cran à chaque éclatement.
- Si saturation on éclate le paquet du curseur, **même** si ce n'est pas lui qui posait problème.
  - débordements à gérer,
  - ou bien plusieurs éclatements successifs avant de résoudre le problème.
- outre le pointeur, besoin de ***P*** : (nb de passages du curseur = nb de fois que chaque paquet a éclaté) initialisé à 0.
- Arrivé après le paquet $2^{(M+P)}$: curseur ramené en tête et on incrémente ***P***.

2. Hachage 3. Hachage dynamique 3. Hachage Linéaire

Exemple : *4 paquets => M=2 , P = 0 au départ.*



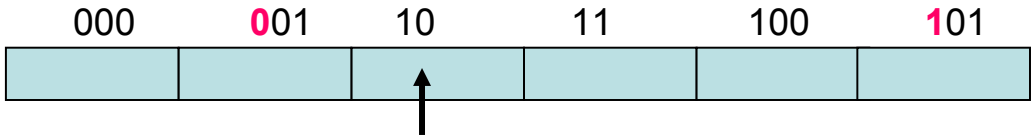
2. Hachage 3. Hachage dynamique. 3. Hachage Linéaire

Accès :

Calculer M+P bits pour trouver le paquet : si on est avant le curseur, utiliser un bit de plus pour prendre en compte les paquets éclatés une fois de plus.

Remarques :

- Les paquets : au plus 1 éclatement d'écart et répartition particulière.
- On peut utiliser un répertoire comme précédemment.
- Avantage : le répertoire ne s'accroit que d'une entrée à chaque fois.



2(M+P) premiers paquets **+ nouveaux paquets éclatés P+1 fois.**

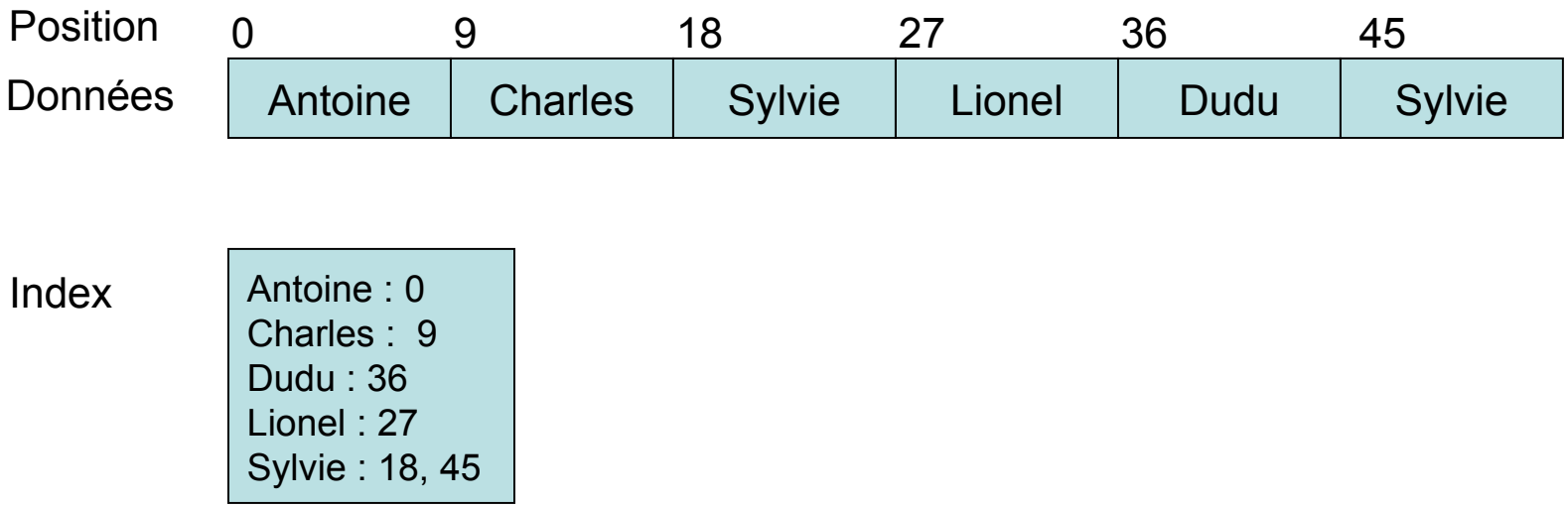
Parmi les 2(M+P) premiers paquets : **ceux avant le curseur ont éclatés P+1 fois**, **les autres P fois.**

Rappels des avantages/Inconvénients des méthodes de hachage :

- Si utilisation d'un répertoire, les paquets de données peuvent être disjoints.
- **Efficace :** si pas trop de débordement, accès en temps constant (si le répertoire tient en RAM)
- Le taux de remplissage doit rester mesuré => perte de place => **inadapté aux très grands volumes de données**.
- **Inutile pour toutes les recherches autre qu'une valeur exacte** sur la clé.
  - SELECT … WHERE prix > 2200.
  - SELECT … WHERE nom LIKE ''&JEAN&''

3. Méthodes d'accès indexées. 1.Principe.

- On met les articles dans le fichier.
- On rajoute un index comme dans les livres.



3. Méthodes d'accès indexées. 1.Principe.

Accès :

1. On charge l'index en mémoire.
2. On cherche dans l'index l'adresse (les adresses) des enregistrements concernés.
3. On charge le (ou les blocs) concernés.

=> Peu d'accès disques : chargement de l'index et uniquement les blocs utiles

3. Méthodes d'accès indexées. 1.Principe.

Variantes :

- Index Trié ?

n clés (ou paquets si l'index est grand)

Index non trié

Lionel :	27
Dudu :	36
Antoine :	0
Sylvie :	18, 45
Charles :	9

Index trié

Antoine :	0
Charles :	9
Dudu :	36
Lionel :	27
Sylvie :	18,45

Recherche : En ***O(n)*** En ***O(log(n))***

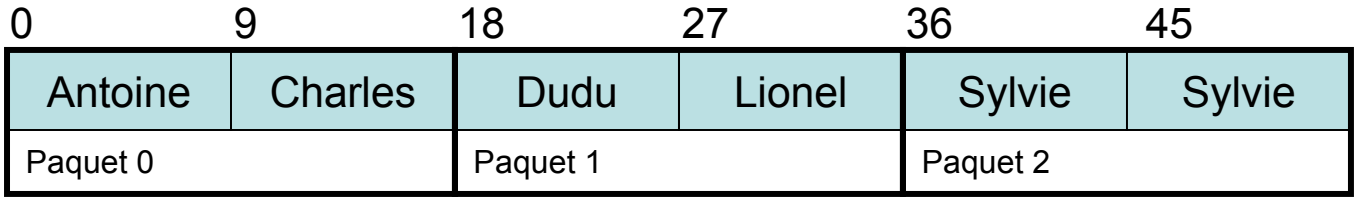
Tests en mémoire ou E/S de paquets si l'index est grand

3. Méthodes d'accès indexées. 1.Principe.

Variantes :

- Fichier Trié *vs.* Index *Dense* ?

Fichier Trié



Index *non dense*

Antoine :	0
Dudu :	18
Sylvie :	36

Recherche de Charles ?

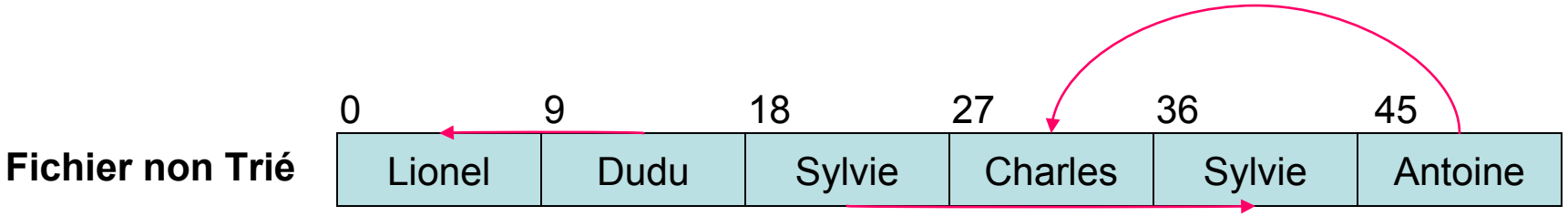
- Entre les positions de Antoine et Dudu.

En général, on indexe les enregistrements en début de paquet.

3. Méthodes d'accès indexées. 1.Principe.

Variantes :

- Fichier Trié *vs.* Index *Dense* ?



Recherche de Charles ?

Soit : Index dense

Antoine :	45
Charles :	27
Dudu :	9
Lionel :	0
Sylvie :	18, 36

Soit : Index non dense et chainage

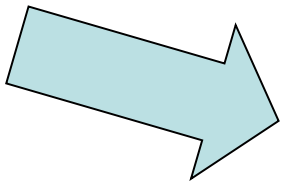
Antoine :	45
Dudu :	9
Sylvie :	18

3. Méthodes d'accès indexées. 1.Principe.

Remarques :

1. Ralentit les insertions/suppression et mises-à-jour de clés. Il faut gérer l'index également.
2. Fichier Indexé => L'index prend de la place.
   - Index dense ?
   - Taille de Clé / taille enregistrement ?
3. Fichier très grand ? Charger l'index en mémoire peut-être un problème !

(Problème 3 et 2)



Index hiérarchiques.

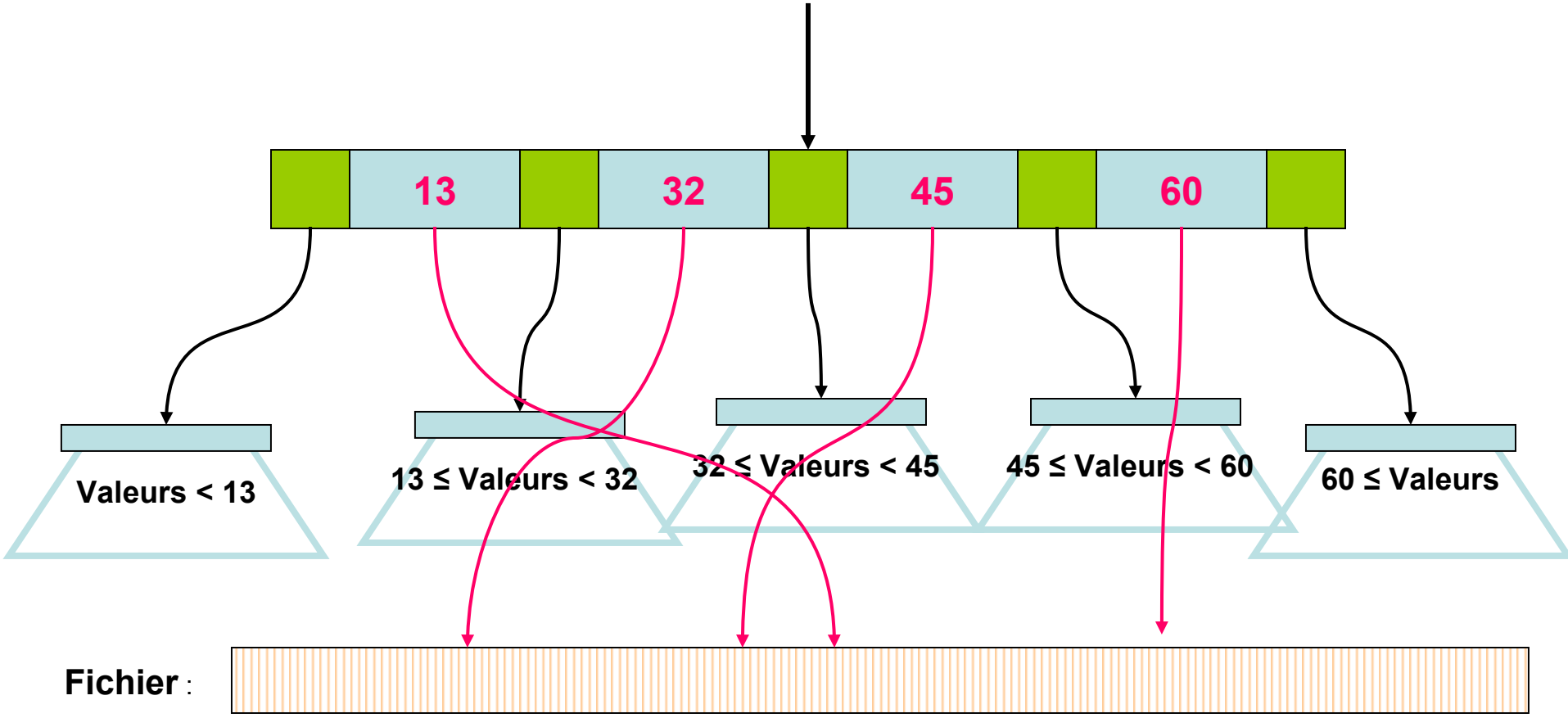
3. Méthodes d'accès indexées. 2. Arbres B.

Arbre B d'ordre m :

- ≈ Arbre (2m)-aire de recherche.
  - Entre **0** et ***2m+1*** fils dans la racine (entre ***0*** et ***2m*** clés)
  - Tous les autres nœuds ont entre ***m+1*** et ***2m+1*** fils. (entre ***m*** et ***2m*** clés)
- Fortement équilibré = Toutes les feuilles sont au même niveau.
- Organisation particulière des nœuds internes.

3. Méthodes d'accès indexées. 2. Arbres B.

Exemple : clés entières, nœud à 4 clés (donc 5 fils).



3. Méthodes d'accès indexées. 2. Arbres B.

Remarques :

- Feuille = pas de pointeurs vers les fils.
- Nœud à ***k*** clés => ***k*** clés et ***2k+1*** pointeurs à stocker.
- Recherche : + - identique à l'arbre binaire de recherche.
- Choix de l'ordre de l'arbre ?
  - Souvent un nœud rempli au max = un bloc.
- Nb d'E/S = Hauteur de l'arbre ?
  - Fonction de l'ordre ***m*** et du nombre de clés ***N***
  - Hauteur la pire = arbre rempli au min
    - 1 clé à la racine, autres nœuds avec ***m*** clés et ***(m+1)*** fils
    - hauteur ***h*** => contient $N = 1+2\ ((m+1)^{(h-1)}-1)$ clés au minimum.
    - stocker ***N*** clés => au pire $h = 1 + \log_{(m+1)}((N+1)/2)$
  - Exemple : ***m = 99,*** fichier de ***1.999.999*** articles
    - $h = 1+ \log_{100}10^6 = 4$
    - ***4 E/S*** pour trouver la position d'un article parmi ***1999999***

3. Méthodes d'accès indexées. 2. Arbres B.

Remarques :

- Comment conserver un arbre fortement équilibré respectant les contraintes sur la taille des nœuds ?
  - => Mécanismes particuliers d'insertion (et de suppression).

Insertion :

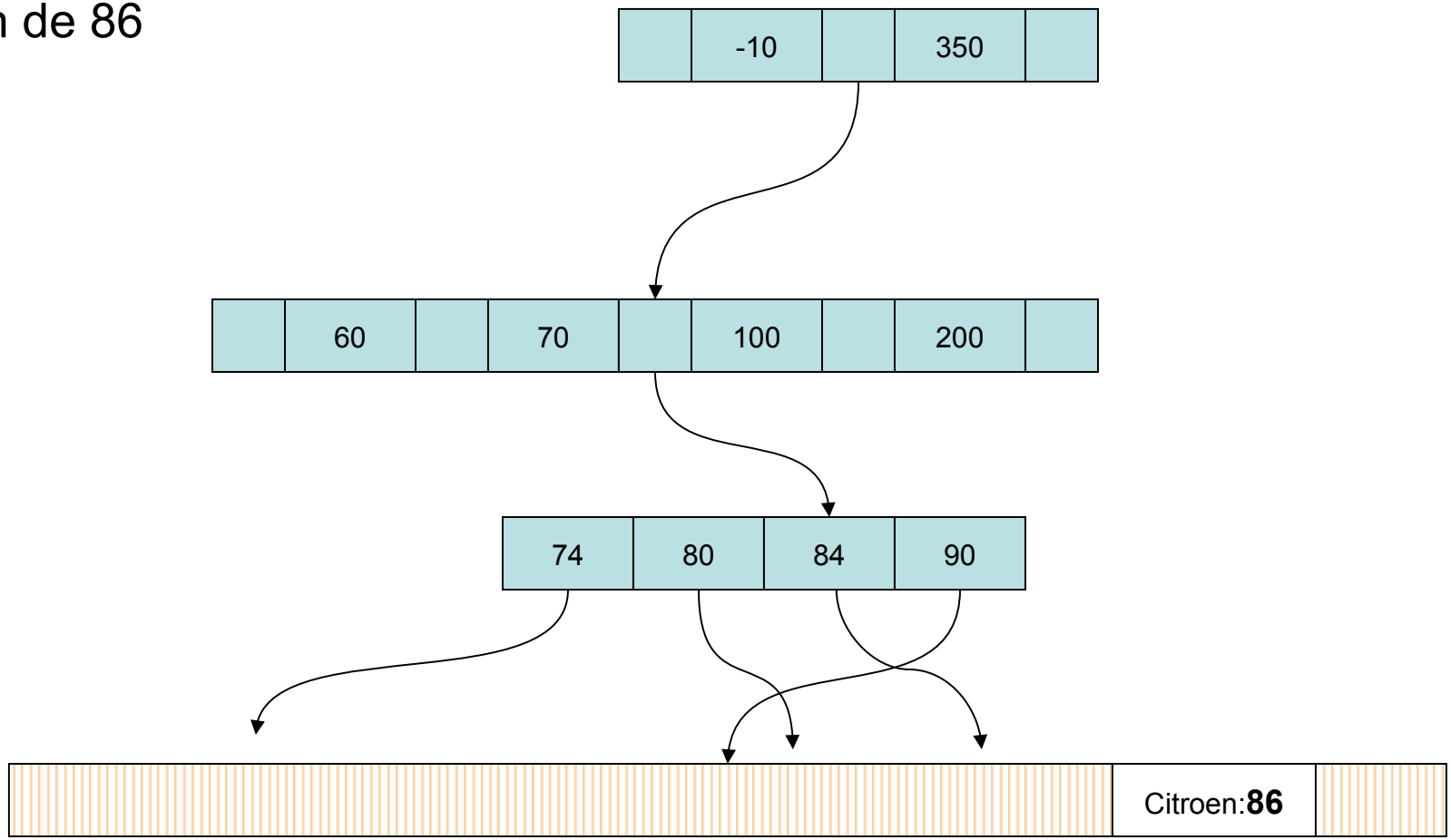
1. On insère la nouvelle clé, sans tenir compte du nombre de clés dans le nœud.
2. ***Si*** sur ce nœud ***nb_clés ≤ 2m,***
   - OK, fini.

   Sinon ***(nb_clés = 2m +1)***
   i. On coupe ce nœud au milieu
   ii. On remonte la clé médiane au nœud père.
   iii. On revient en 2 avec le nœud père

3. Méthodes d'accès indexées. 2. Arbres B.

Exemple : Arbre d'ordre 2

Insertion de 86

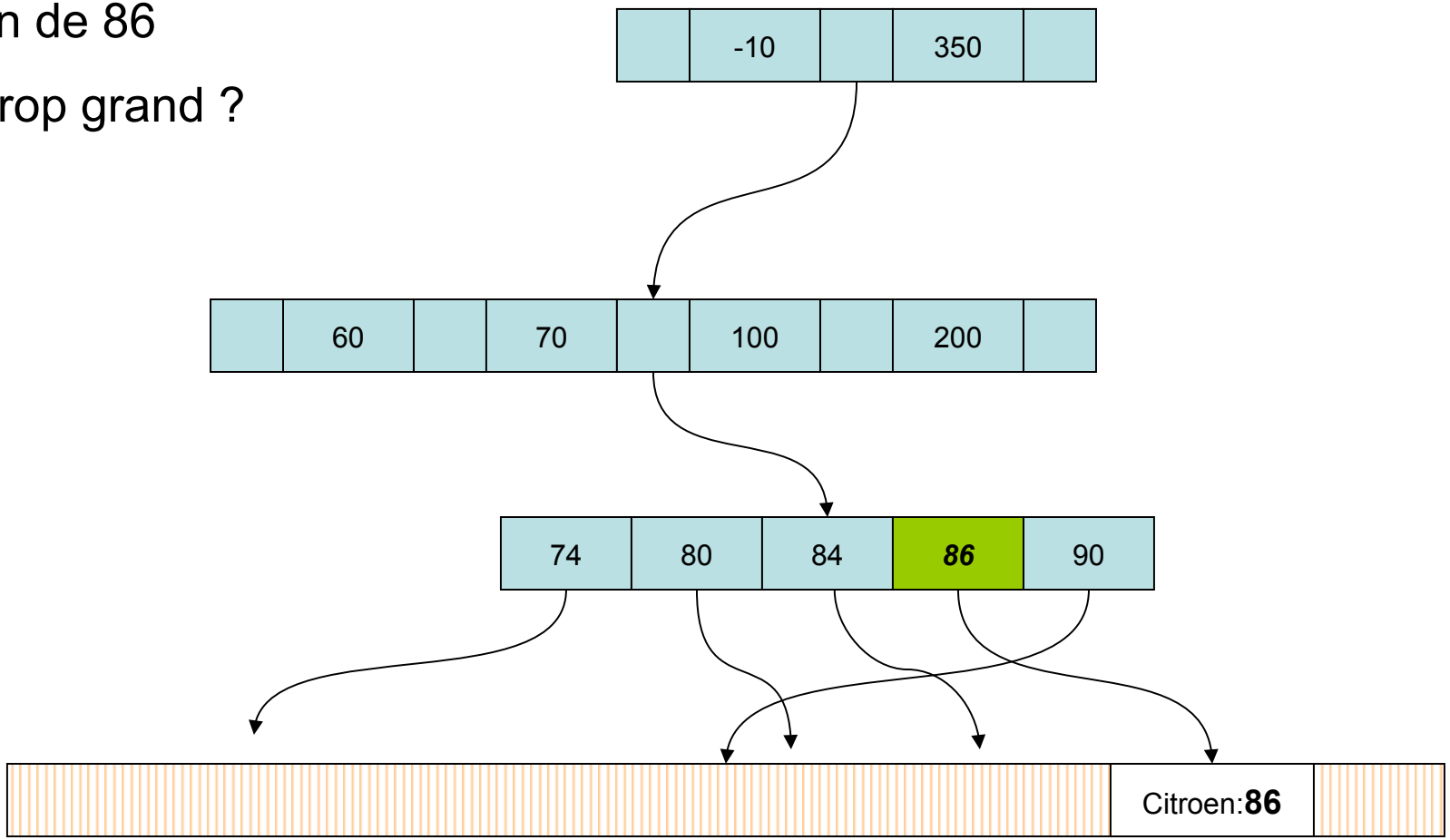


3. Méthodes d'accès indexées. 2. Arbres B.

Exemple : Arbre d'ordre 2

Insertion de 86

Nœud trop grand ?



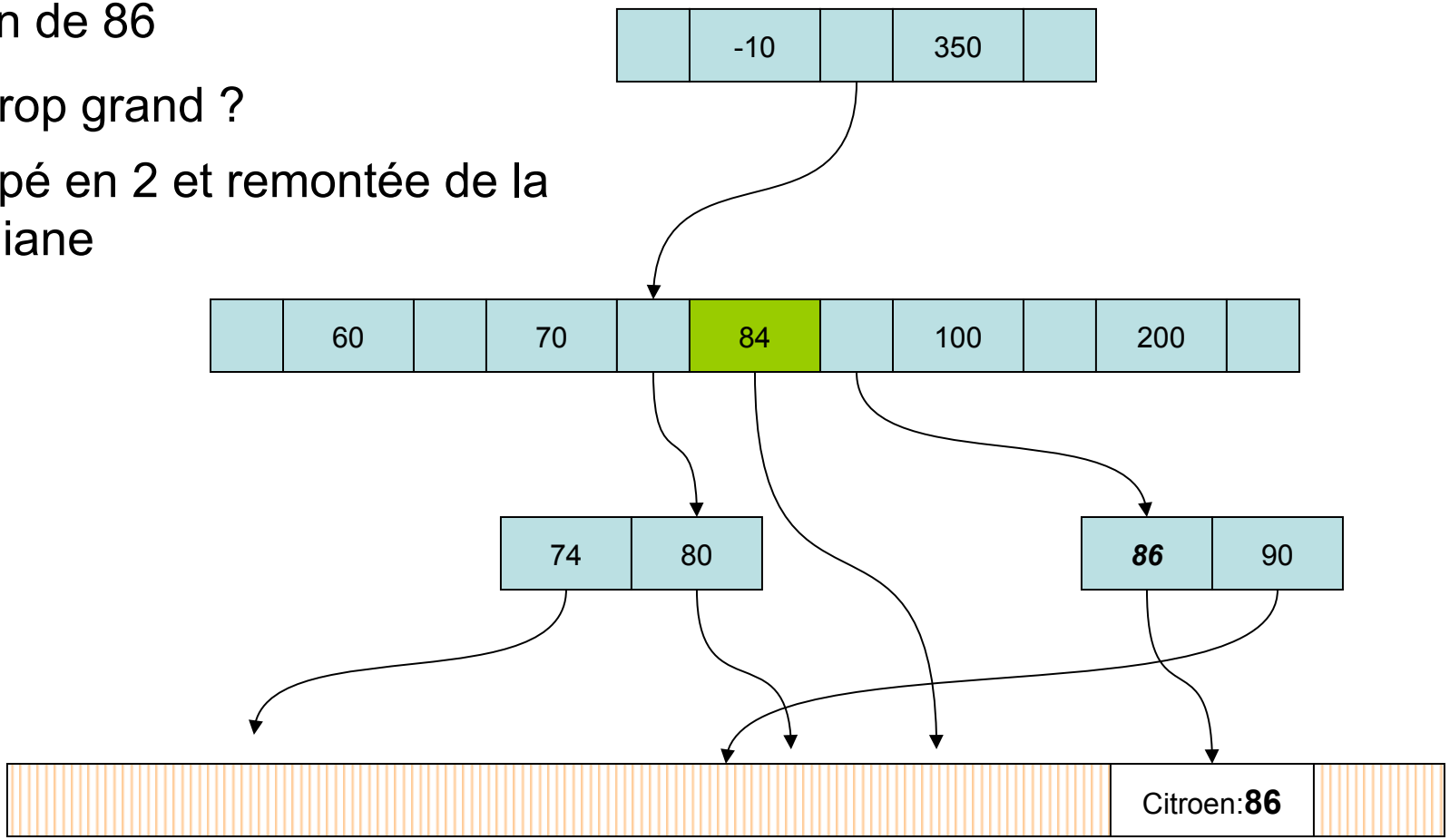
3. Méthodes d'accès indexées. 2. Arbres B.

Exemple : Arbre d'ordre 2

Insertion de 86

Nœud trop grand ?

Coupé en 2 et remontée de la médiane



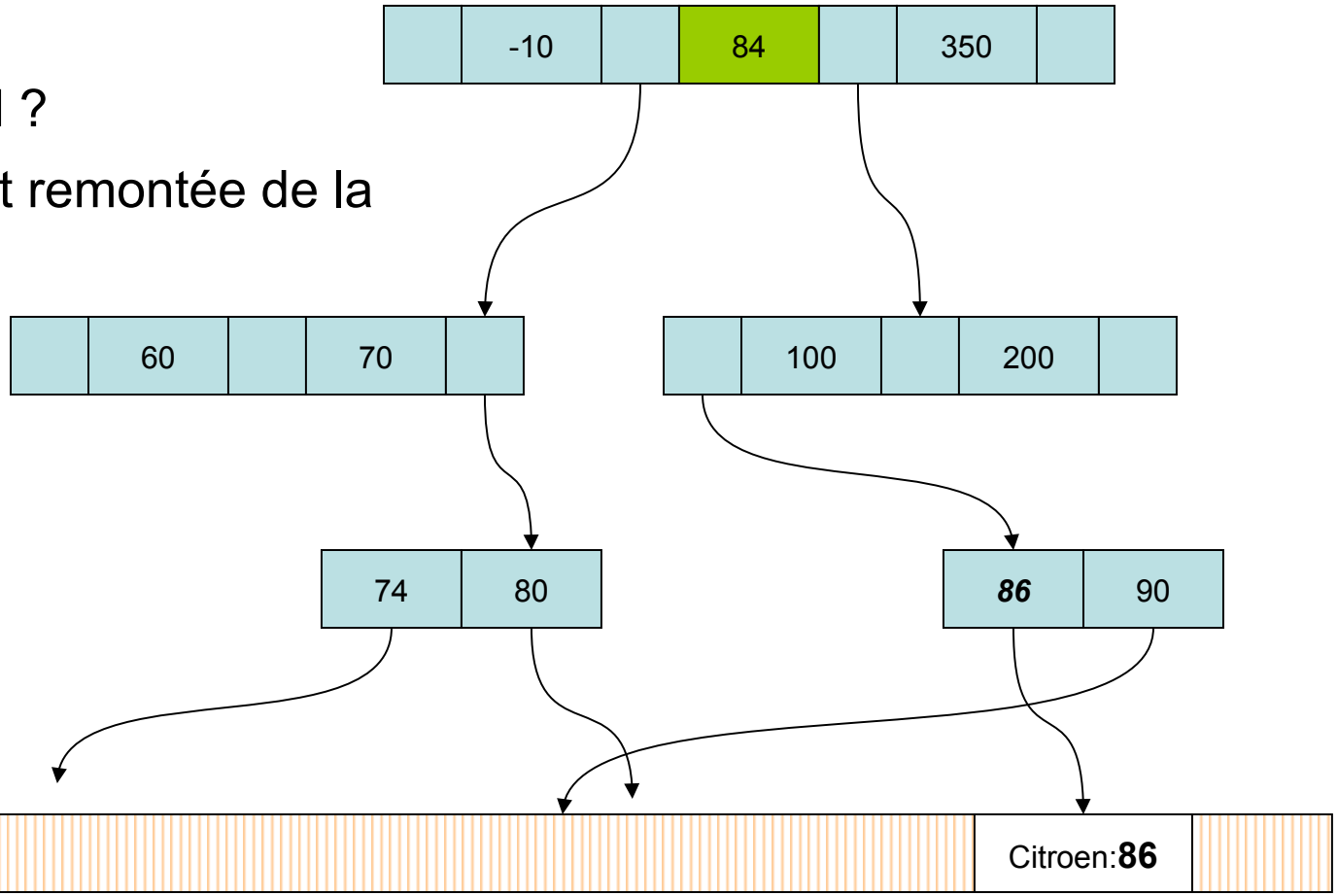
3. Méthodes d'accès indexées. 2. Arbres B.

Exemple : Arbre d'ordre 2

Insertion de 86

Nœud trop grand ?

Coupé en 2 et remontée de la médiane



3. Méthodes d'accès indexées. 3. Arbres B+.

Problème des arbres B :

- ORDER BY souvent utilisé
- = Afficher les clés dans l'ordre croissant ?
  - => Parcours préfixe de l'arbre => Charger/Décharge/Recharger les nœuds internes.
  - => ***Perte de temps.***

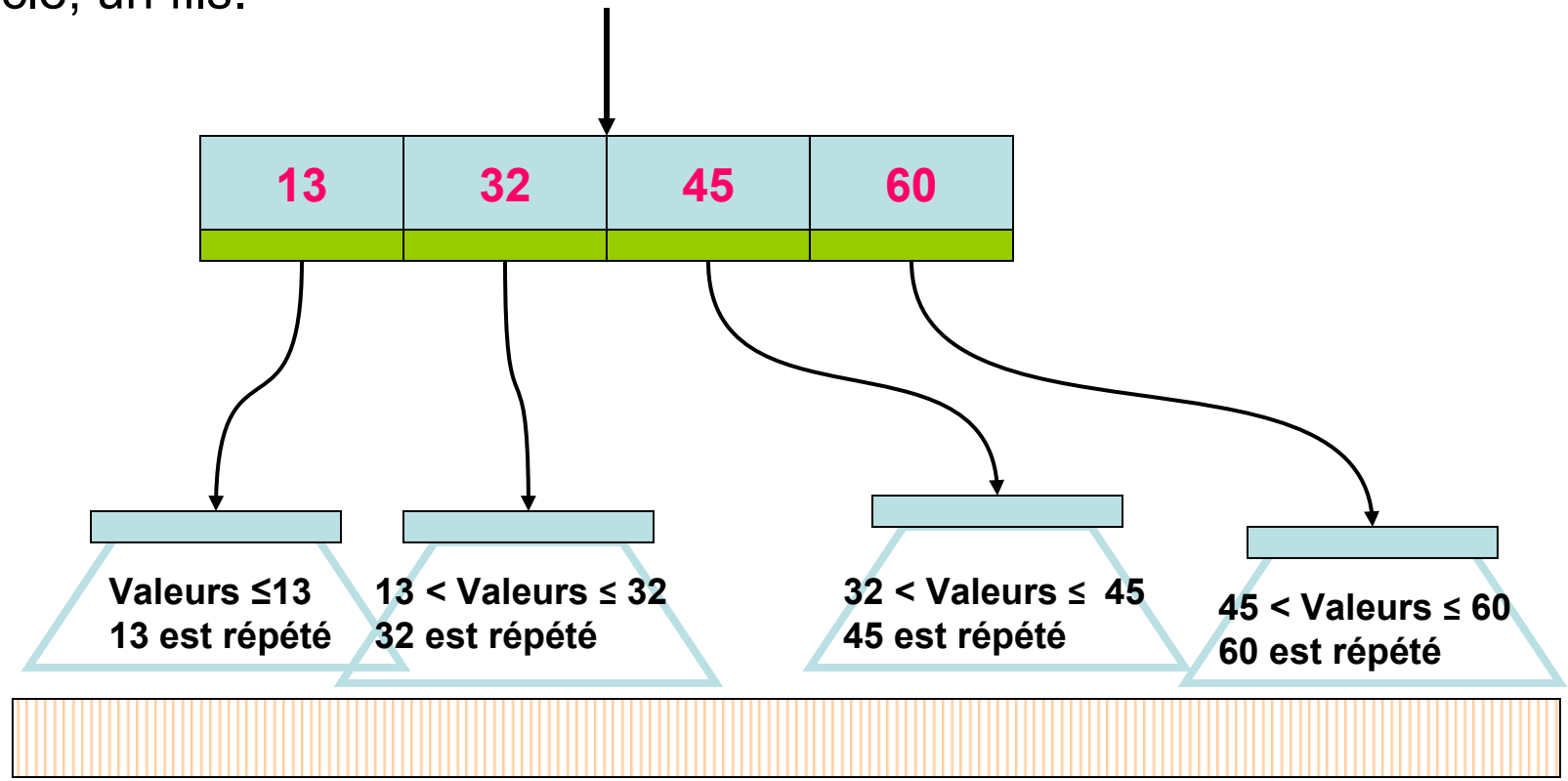
Solution :

- Reporter les clés des nœuds internes dans les nœuds fils
  - Structure du nœud un peu modifiée, ***n*** clés **=> *n*** fils.
- Chainer les blocs des feuilles.
- Les pointeurs vers les données ne sont que dans les feuilles.

3. Méthodes d'accès indexées. 3. Arbres B+

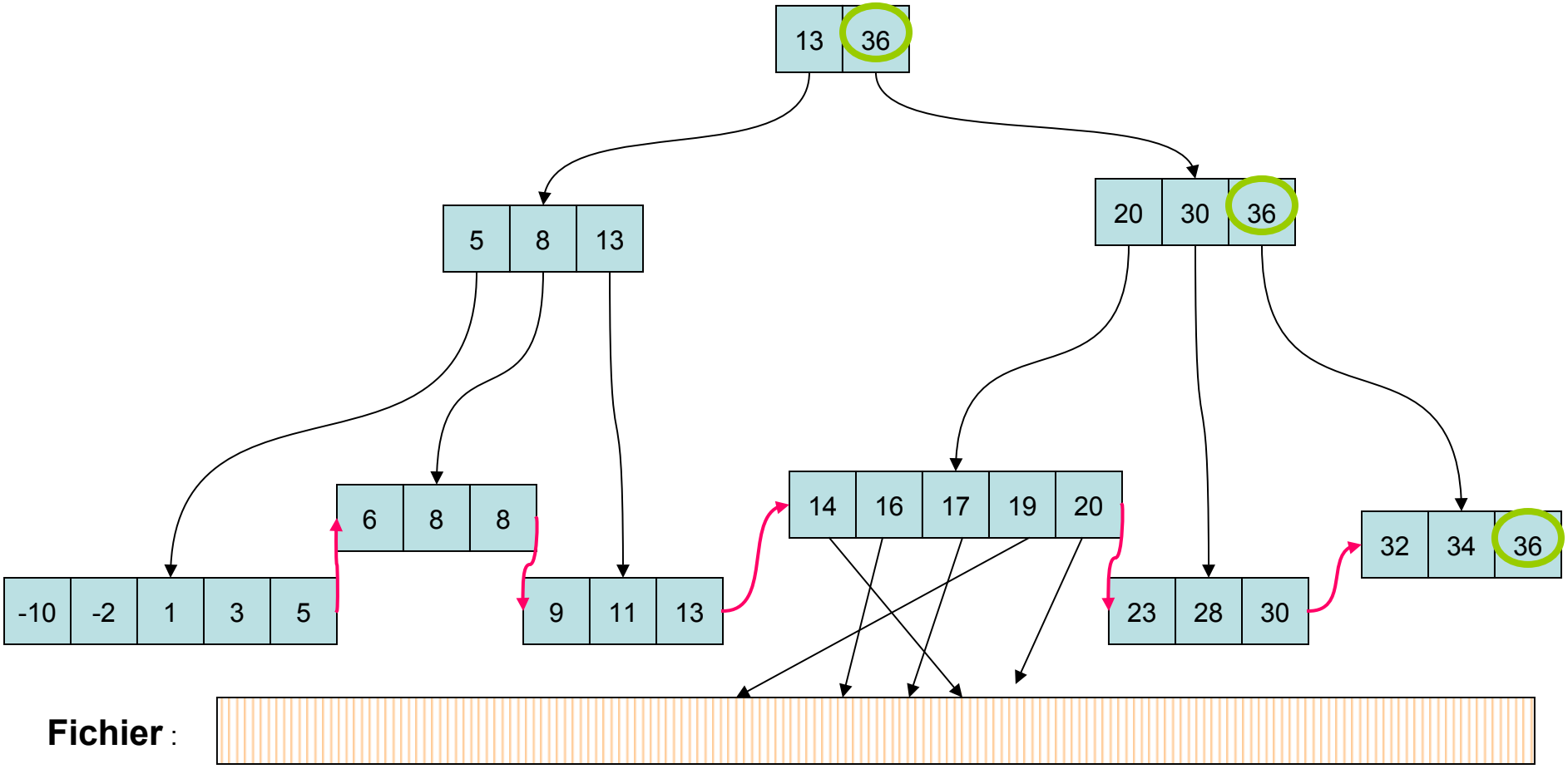
Structure du noeud : Arbre B+ d'ordre m

- Racine : entre 0 et m+1 clés.
- Autres nœuds : entre m+1 et 2m+1 clés.
- Une clé, un fils.



3. Méthodes d'accès indexées. 3. Arbres B+

Exemple : Arbre B+ d'ordre 2



3. Méthodes d'accès indexées. 3. Arbres B+

Remarques :

- Avec la convention choisie ici sur les inégalités, le max de l'arbre est la dernière clé de la racine
  - On peut prendre une autre convention.
- L'arbre B+ peut être utilisé pour :
  - Implémenter seulement l'index : les feuilles ne contiennent que les clés et des pointeurs sur le fichier.
  - Implémenter index + fichier. Au niveau des feuilles, on met directement l'enregistrement.
- Si un attribut est clé primaire ?
  - À chaque Update/Insert => vérification d'unicité de la clé.
  - => Index très souvent créé automatiquement sur les attributs clés.

3. Méthodes d'accès indexées. 4. Avantages/Inconvénients

Avantages :

- Accès rapide : en ***log(N)*** (**N** nombre d'articles)
- Accélération de requêtes avec une condition
  - WHERE nom > ''gardarin'' AND nom < ''lester''.
  - WHERE age > 2000

Inconvénients :

- Inutile pour accélérer
  - WHERE nom LIKE ''%gey%''
  - WHERE (2 * age * age) = 50
- Mauvaise solution pour certains attributs,
  - WHERE sexe = 'M'
    - 2 valeurs possibles => on ramène toujours la moitié des articles…
- Prennent de la place.
- Mise à jour de la table => mise à jour de l'index.

4. Méthodes d'accès Multi attributs.

- Remédier à quelques inconvénients des méthodes précédentes.
  - « WHERE nom LIKE ‘’%jean%’’ »
  - Attributs au domaine restreint, ex : {essence,diesel,GPL}, {célibataire, veuf, marié, divorcé}....
- Accélération de recherche concernant plusieurs attributs.

Cf. TD

4. Méthodes d'accès Multi attributs. 1. Index Bitmap

Dans le cas d'un domaine restreint :

- Carb. {essence, gazole, GPL}, Sexe{masculin, féminin}, etc.

=> problèmes particuliers.

On récupère à chaque interrogation une grosse partie des articles.

=> La partie de l'index chargée en mémoire devient très grande.

Exemple :

- sexe : {masculin, féminin}
- 1.000.000 d'articles.
- Arbre B+ d'ordre 100

- En moyenne on doit récupérer 500.000 articles.
- Pour un article : ''*clé*'' (7 octets) + 2 pointeurs (4 octets/pointeurs) nœuds et enregistrements.
- **=> Total** = *500.000 *15* Octets. = ***7,5 MO***.

4. Méthodes d'accès Multi attributs. 1. Index Bitmap

Autre solution : index bitmaps.

Tables de ***n*** enregistrements, ***k*** valeurs possibles pour l'attribut ***A***.

- Index constitué par une matrice ***M n***x***k*** de bits
- ***M(i,j)*** vaut ***1*** si l'article ***i*** contient la ***j***ème valeur possible de ***A***

Exemple précédent :

	Féminin	**Masculin**
Charles	0	1
Isabelle	1	0
Eon	0	0

Taille de l'index complet ? :

1.000.000 * 2 bits = 2Mbits = ***250 kO***

4. Méthodes d'accès Multi attributs. 1. Index Bitmap

Recherche des personnes de sexe féminin ?

1. Parcourir l'index pour récupérer les indices ***i*** tels que ***M(i,féminin) = 1***.
2. Pour chaque indice ***i*** trouvé, récupérer le **i**ème article du fichier.
   - (Calcul possible du bloc correspondant avec la taille d'un article et la taille du bloc, si les blocs sont pleins)

Autres utilisations :

- attributs multivalués
- codes de plage de valeurs
- un index bitmap pour plusieurs attributs différents.

Composition	**Nylon**	**Coton**	**Laine**
Chemise	0	1	0
Pull	0	1	1
Pantalon	1	0	1

Prix	[0-10[	[10-20[	[20-∞[
Chemise	0	1	0
Pull	0	0	1
Pantalon	0	1	0

4. Méthodes d'accès Multi attributs. 1. Index Bitmap

Pourquoi pas plus simple ?

Un tableau ***T*** de n entiers, ***T(i) = j*** si le ***i***eme article a la ***j***ème valeur possible?

- gestion des attributs multivalués
- AND OR, NOT faciles à gérer.
  - contient du coton ou de la laine

4. Méthodes d'accès Multi attributs. 2. Index secondaires

Index secondaire :

Index sur une donnée non discriminante (i.e. pas la clé de la relation)

Exemple : PERSONNE (NOM, VILLE, NUM_SS).

- Index « automatique » sur la clé NUM_SS. (doublons interdits)
- Index sur VILLE pour accélérer les recherches des personnes habitant XXX (doublons autorisés)
- Index sur NOM pour accélérer les recherches des personnes de nom YYY (doublons autorisés)

4. Méthodes d'accès Multi attributs. 2. Index secondaires

Exemple : PERSONNE (NOM, VILLE, NUM_SS).

Utilisation pour rechercher les « TITI habitant BORDEAUX » :

- Si on sait qu'il est très rare que 2 personnes ont le même nom, mais villes peuplées.
  1. Chercher avec l'index sur le NOM, les TITI
  2. Parmi ces personnes, ne garder que celles qui habitent BORDEAUX.
- Si la base stocke un arbre généalogique (beaucoup de personnes ont le même nom).
  1. Chercher avec l'index sur la VILLE, les habitants de BORDEAUX
  2. Parmi ces personnes, ne garder que les TITI.

Plan

- Gestion Physique et Optimisation de requêtes
  - Différentes indexations possibles.
  - Intérêt et fonctionnement d'un optimiseur.
- Gestion des transactions
  - Transaction ?
  - Récupération des erreurs.
  - Gestion de la concurrence.
- Gestion des droits et vues
- Intégrité et BD Actives.
  - Les contraintes d'intégrité.
  - Les outils en SQL.