Techniques d'indexation

Implémentation du modèle relationnel

~

LIFBDW2: Bases de données avancées

Une vision conceptuelle des données :

- Algèbre relationnel, SQL, etc.

Les tables et leurs enregistrements sont stockés physiquement !!

- Les enregistrements ont une adresse (physique, logique).
- Mais les requêtes SQL référencent des valeurs d'attributs, pas des adresses.

SELECT * FROM R WHERE A=12;
SELECT * FROM E WHERE N>10;

COMMENT RETROUVER (EFFICACEMENT) LES ENREGISTREMENTS QUI ONT UNE CERTAINE VALEUR SUR UN ATTRIBUT DONNÉ ?

Métaphore de l'aiguille

- Les données : le *foin* !
- Le résultat de la requête : *l'aiguille* !
- ***Comment trouver très rapidement l'aiguille ?***
- ***Comment éviter de parcourir l'intégralité des données quand ce n'est pas nécessaire ?***

Comment éviter de tout parcourir ?

- Sélection

SELECT * FROM Etudiant
WHERE note<10;

- Jointure naturelle

SELECT * FROM Etudiant
NATURAL JOINT Cours;

- Agrégation

SELECT Max(note) FROM
Etudiant ORDER BY note;

SELECT * FROM Etudiant WHERE note<10;

Sans index, le seul moyen est de parcourir l'intégralité des données : coût de |D|

e1,maths, 12
e1,BD,16
e2,Sport, 8
...
e123,Rx,7
e231,Algo,19

SELECT * FROM Etudiant NATURAL JOINT Cours;

ou plus généralement

SELECT * FROM TABLE1, TABLE2 WHERE TABLE1.X = TABLE2.Y;

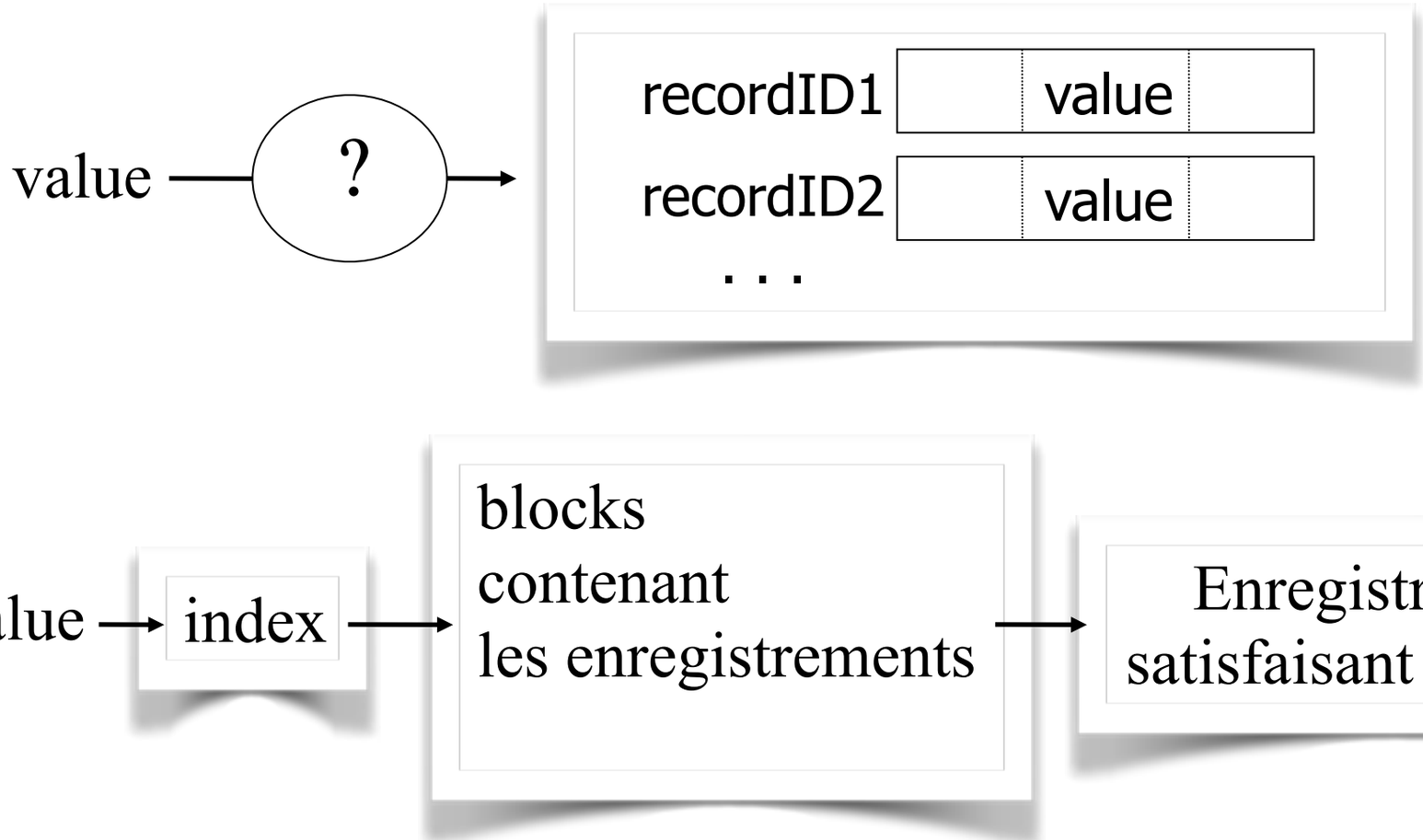
Approche naïve :

Parcourir les deux tables avec une double boucle

Coût : |T1|x|T2|

Les index pour optimiser les réponses aux requêtes!

- Un index sur un fichier :
  - Une structure de données «disk-based»
  - Améliore la sélection des **clés de recherche** (search key) de l'index
  - Un index peut porter sur la valeur d'un ou plusieurs attributs de la relation.
  - ☂: **Search key ≠ key** : un ou plusieurs attributs pour lesquels on veut rechercher efficacement (pas forcément unique)


value
?
recordID1
value
recordID2
value
. . .
value
index
blocks
contenant
les enregistrements
Enregistrements
satisfaisant la requête

Exemple [wikipedia]

```
N° sécu             // N° de sécurité sociale.

Annee_naissance  // Année de naissance du contribuable
Revenu              // Revenu
Frais Reel          // montant des frais réels
```

un index sur (N° sécu)

Exemple de requête utilisant l'index : Contribuable ayant N° sécu = 123456

un index sur (Annee_naissance, Revenu)

Exemple de requête utilisant l'index : Contribuables nés avant 1980, avec un Revenu entre 20 000 et 30 000

Exemple de requête utilisant l'index : Revenu maximal selon l'année de naissance

Concepts

- Les structures de stockage consistent en plusieurs fichiers :
  - «Data files» : stockent les enregistrements d'une relation
  - Search key
  - «Index file» : Association valeur_search_key avec les enregistrements qui supportent la valeur de la clé de recherche.
  - «Sequential files» (fichiers séquentiels): enregistrements triés par rapport à leur **clé primaire**

Questions sur les index

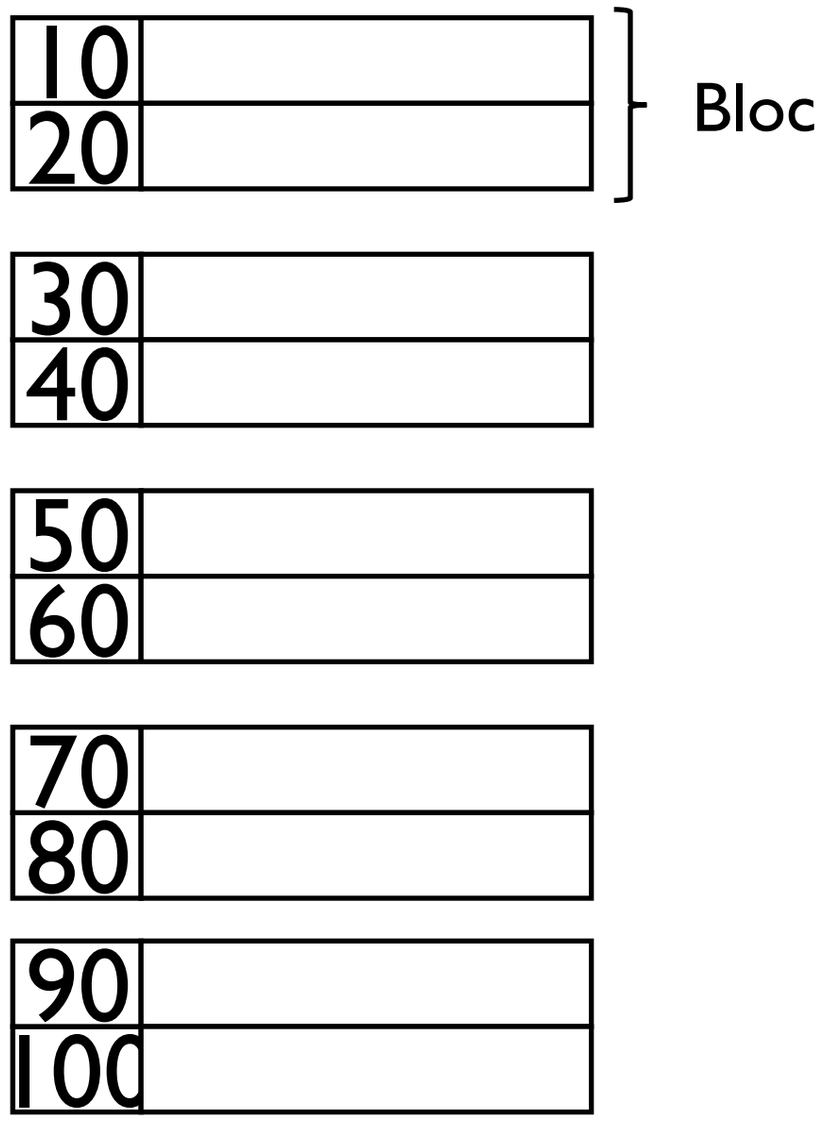
- Quel type de requêtes supportent-ils?
  - sélection de la forme x <op> val
    - Egalité : (op est =)
    - Mais aussi (=<, <, >=, >, etc.)
  - Des requêtes plus complexes :
    - «ranking queries» : 10 restaurants les plus près de Nautibus;
    - Expressions régulières (génome, etc.)

Index sur des fichiers séquentiels

- Rappel : Search key ≠ primary key
- Index primaire : (sur le champ séquencé)
    - l'index sur l'attribut qui détermine le séquençage de la table
- Index secondaire :
    - Index sur n'importe quel autre attibut
- Index **dense** : toutes les clés de recherche présentes
- Index «**creux**» (sparse index)
- Index «multi-niveaux»

Les tuples sont ***ordonnés*** *par rapport à leur clé primaire*

Fichier séquentiel

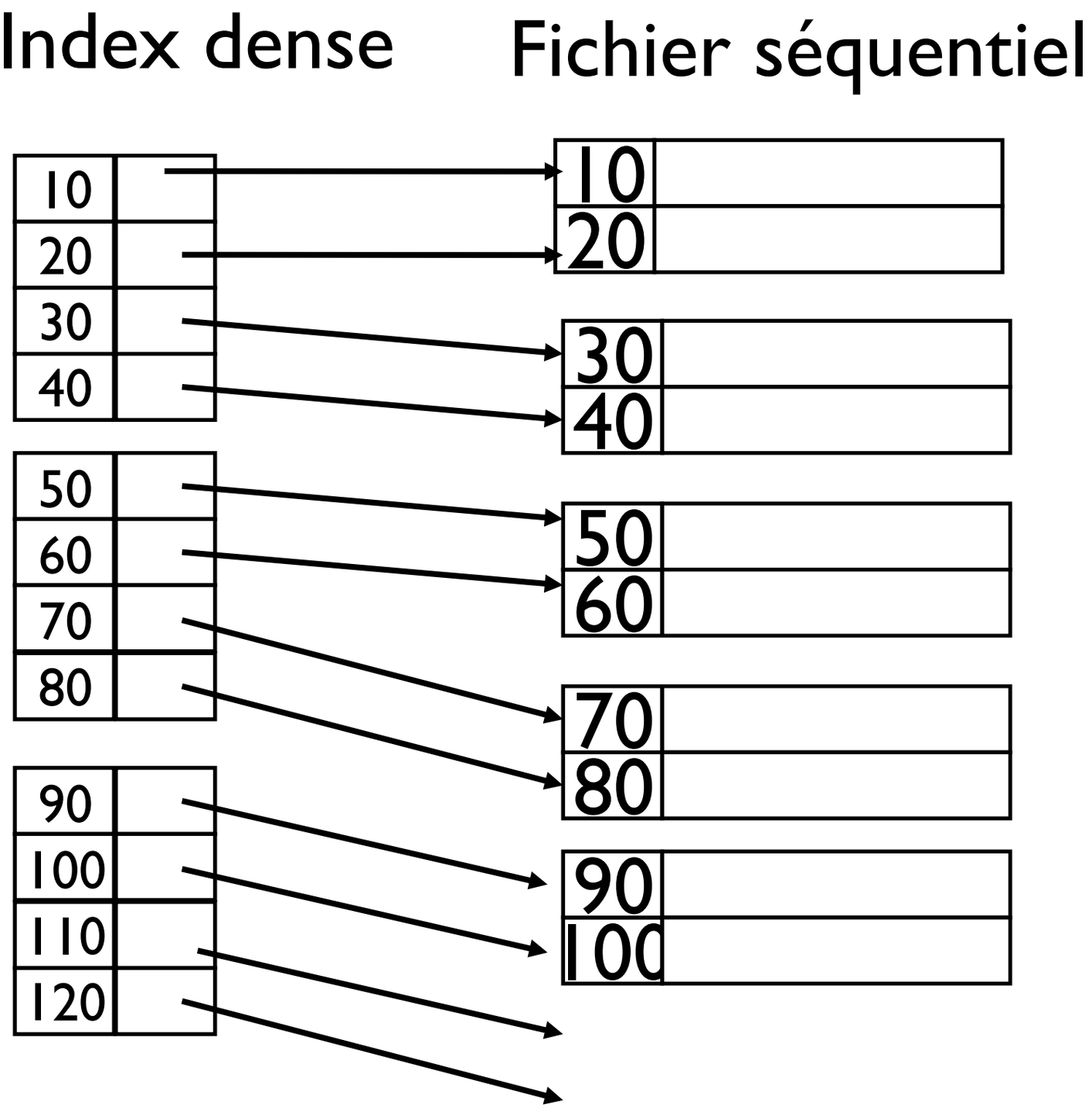


Le fichier index requiert beaucoup **moins** de blocs que le fichier de données.

☞ plus facile à

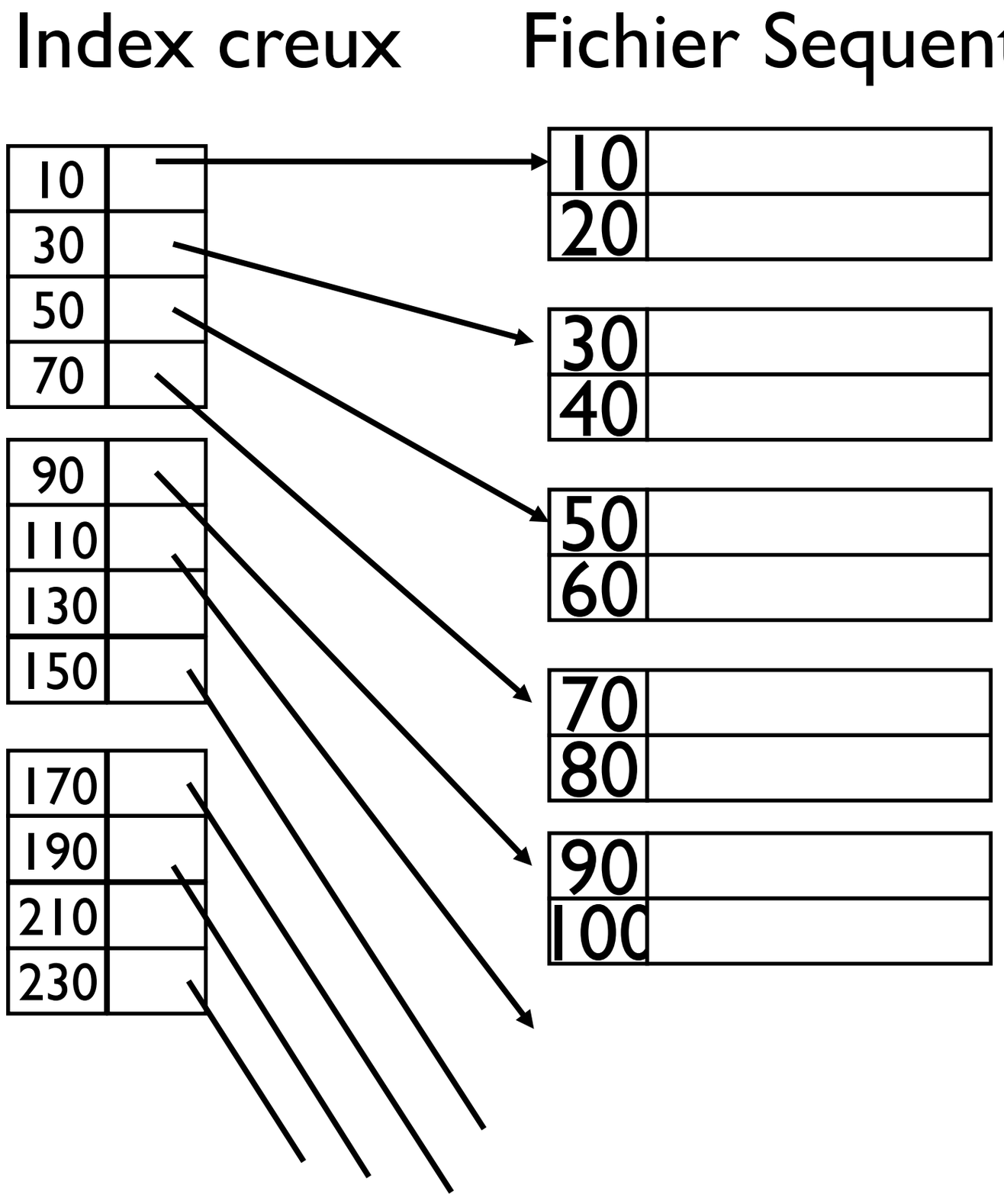
charger en mémoire

Pour une clé K, seulement $\mathbf{log_2 n}$, blocs d'index doivent être parcourus.



Généralement, une seule clé par bloc de données

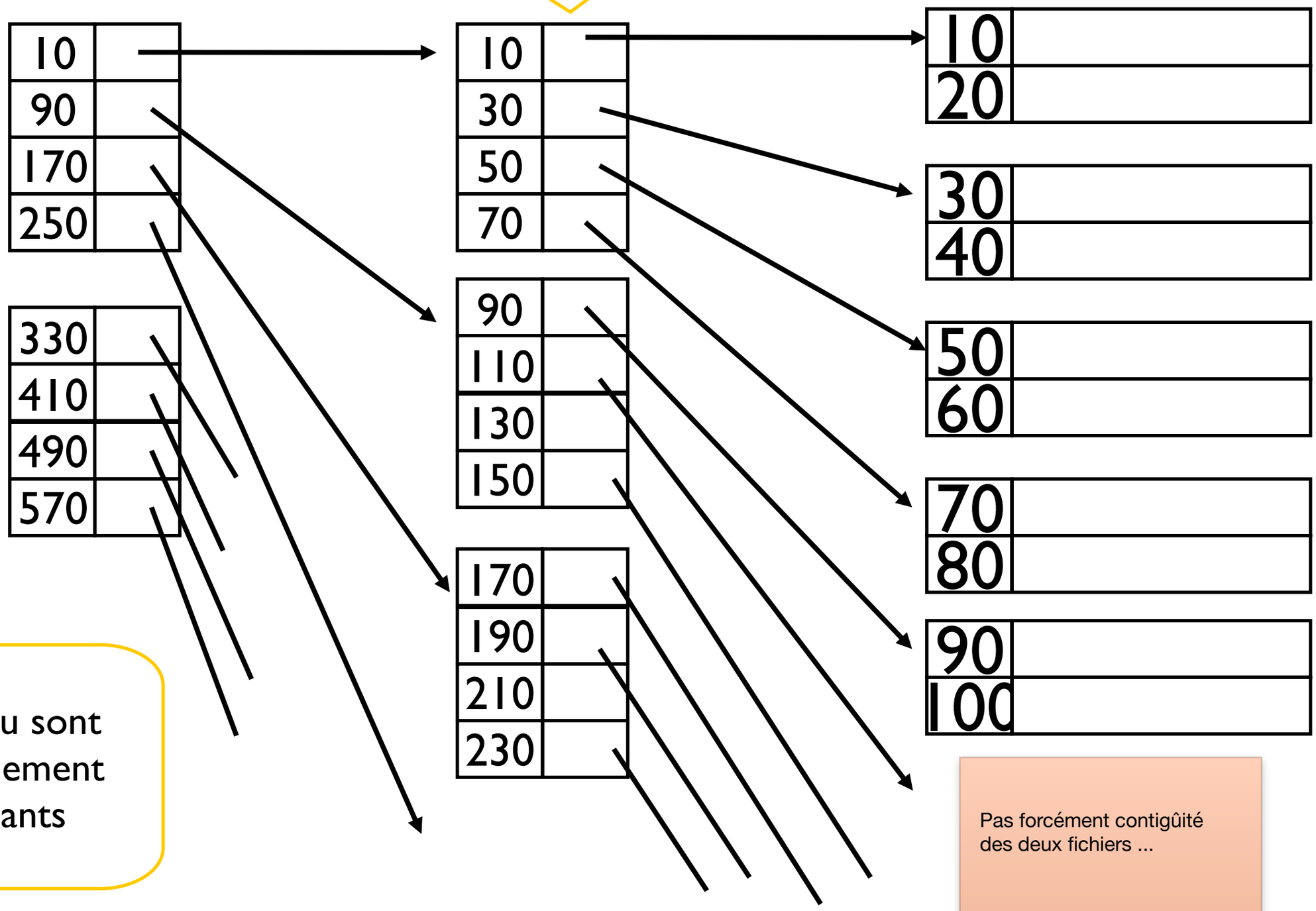
Trouver l'entrée avec la plus grande valeur inférieure ou égale à la valeur recherchée



Index creux double niveau

Traite l'index comme un fichier et construit un index dessus.

Fichier séquent.



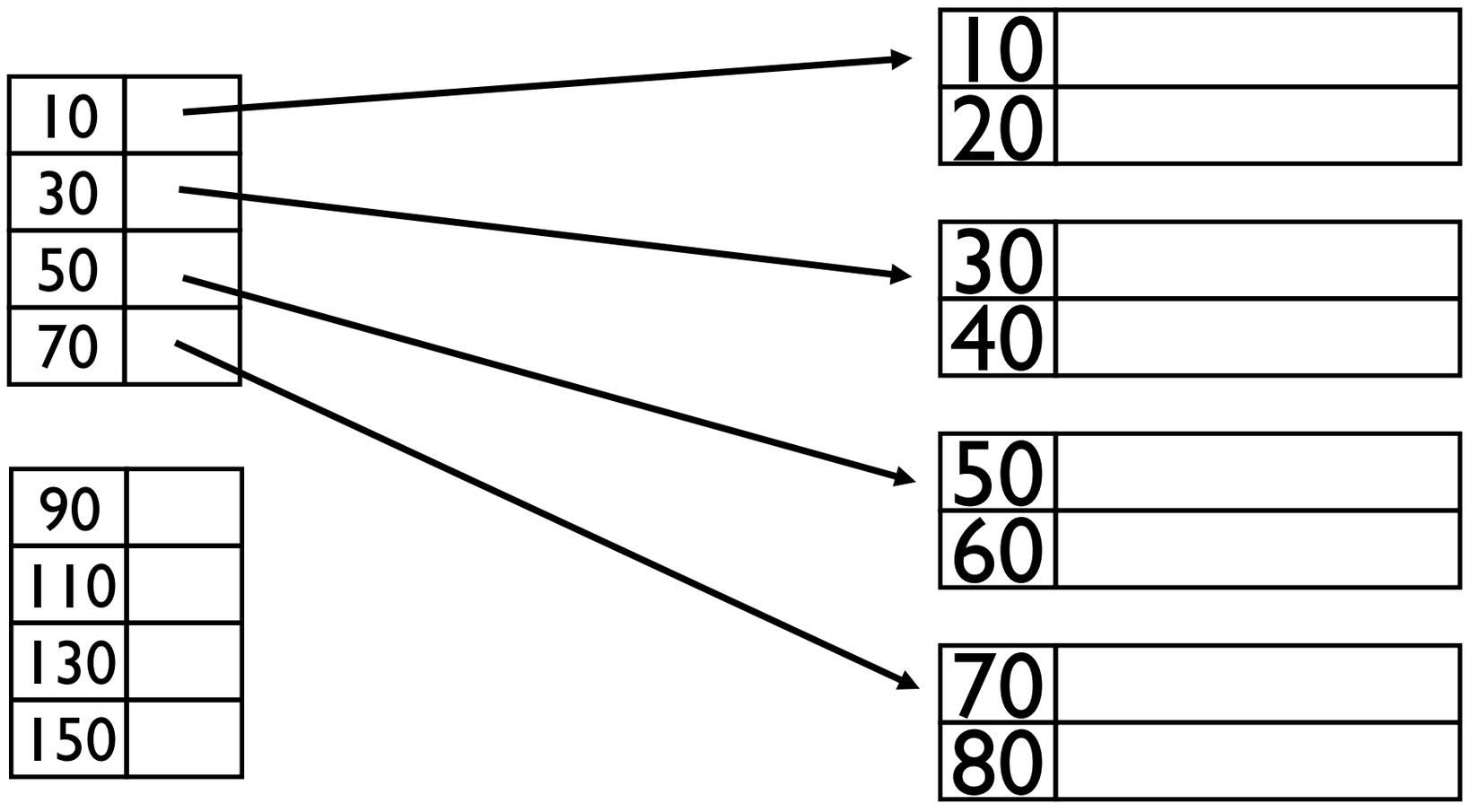
- 2 niveau sont généralement suffisants

Pas forcément contigûité des deux fichiers ...

Opération de m.a.j

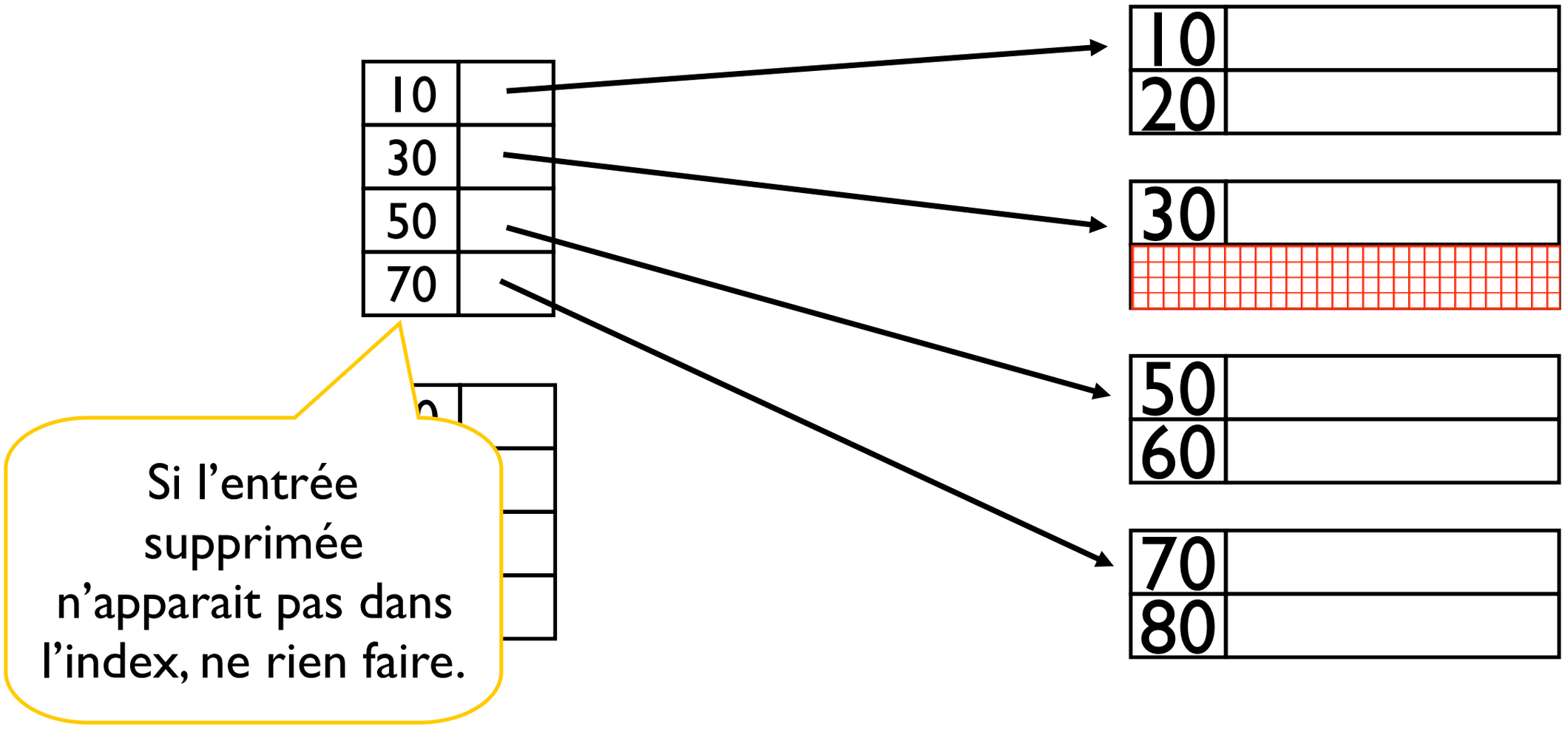
- Insertion
- Suppression
- Quid de ces opérations par rapport aux types d'index ?

Suppression dans index creux



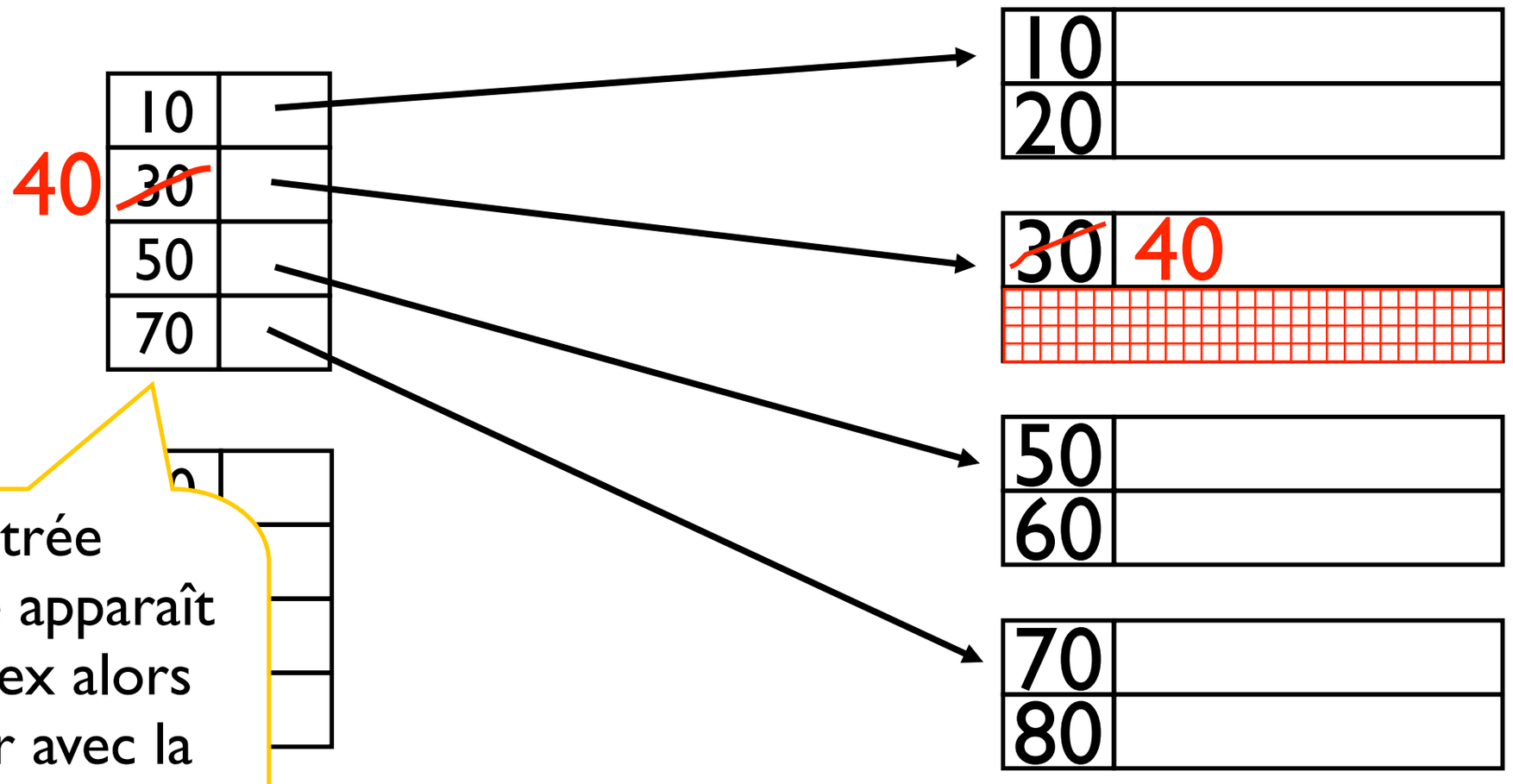
Suppression dans index creux

– supprimer l'enregistrement 40



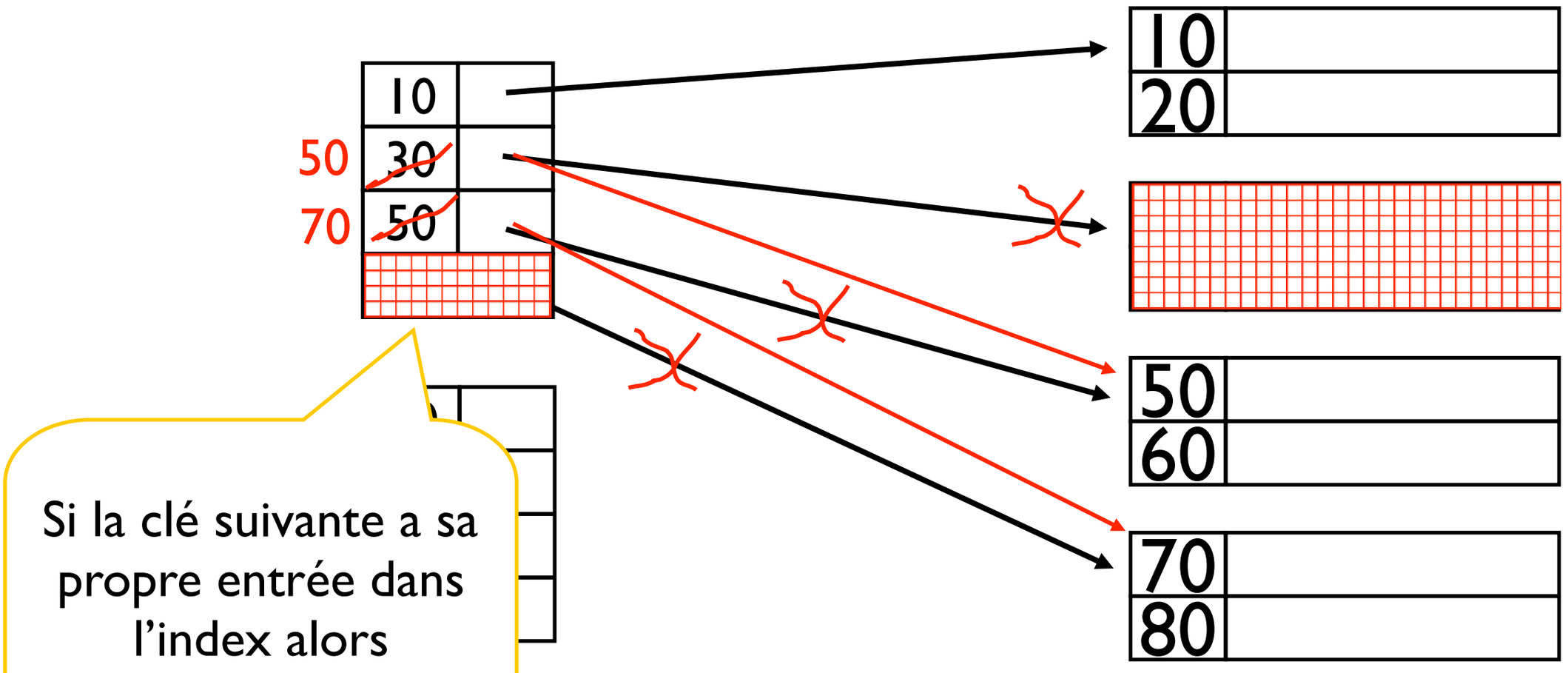
Suppression dans index creux

- supprimer l'enregistrement 30

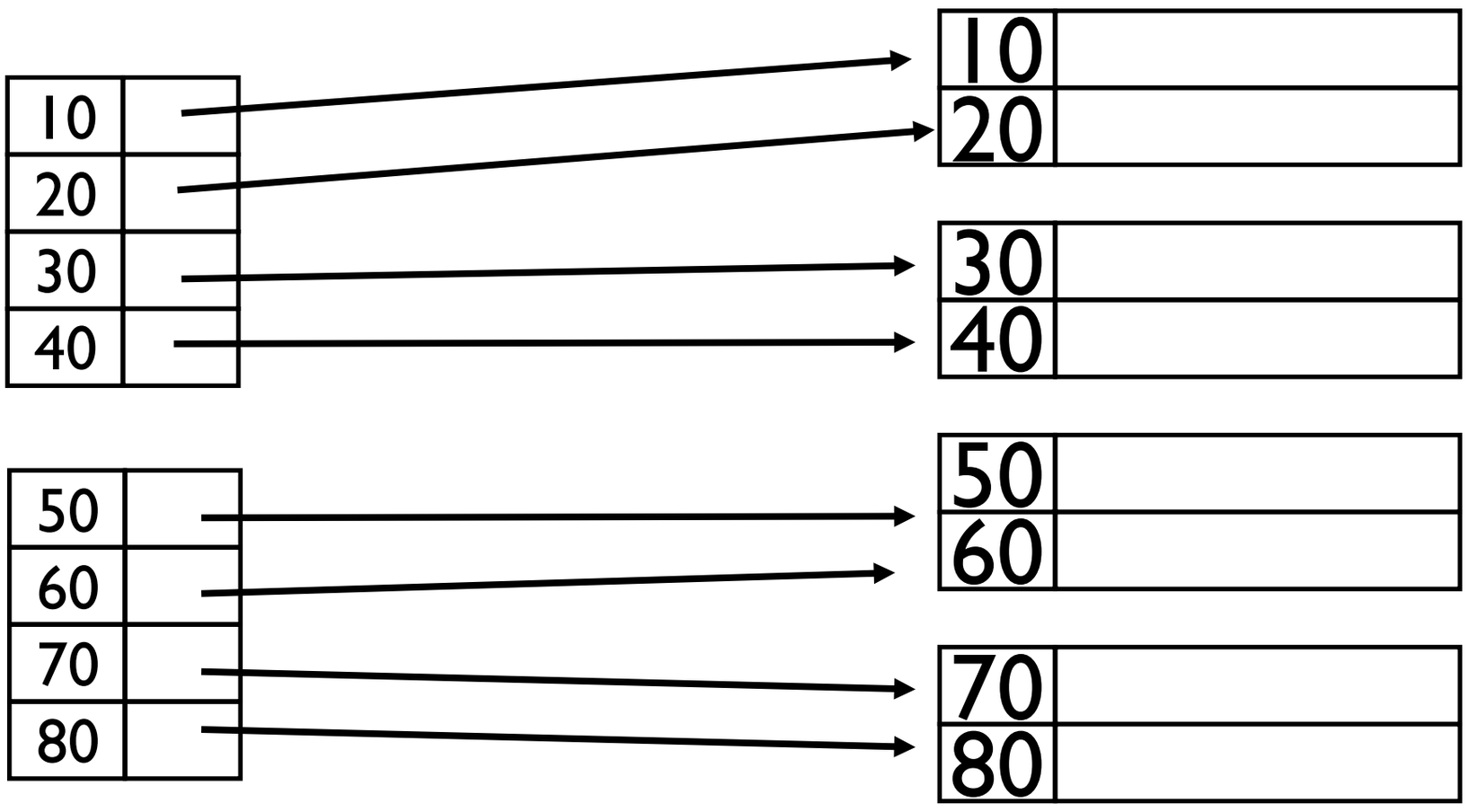


Suppression dans index creux (tjs +)

– supprimer 30 et 40

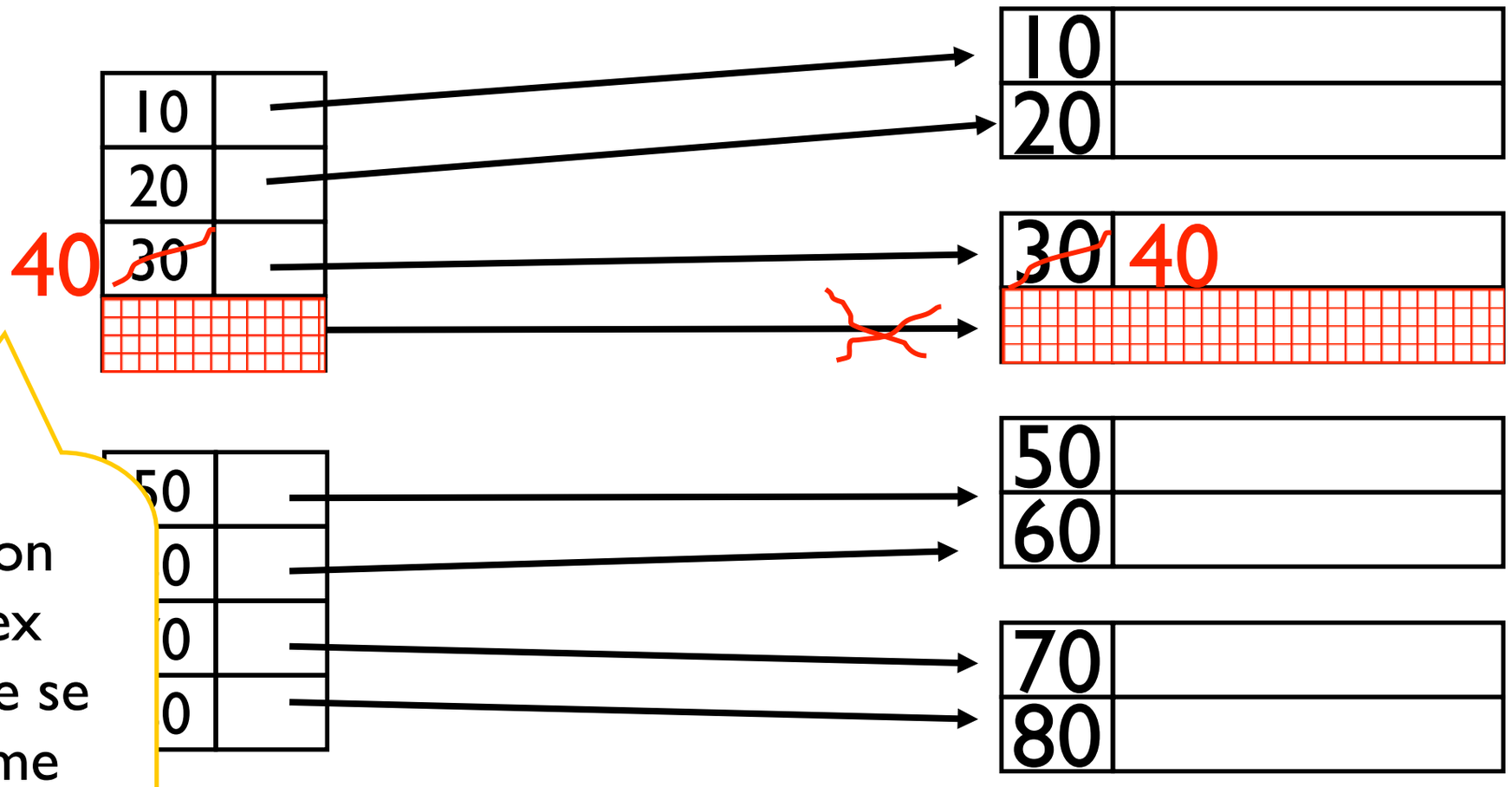


Suppression dans index dense



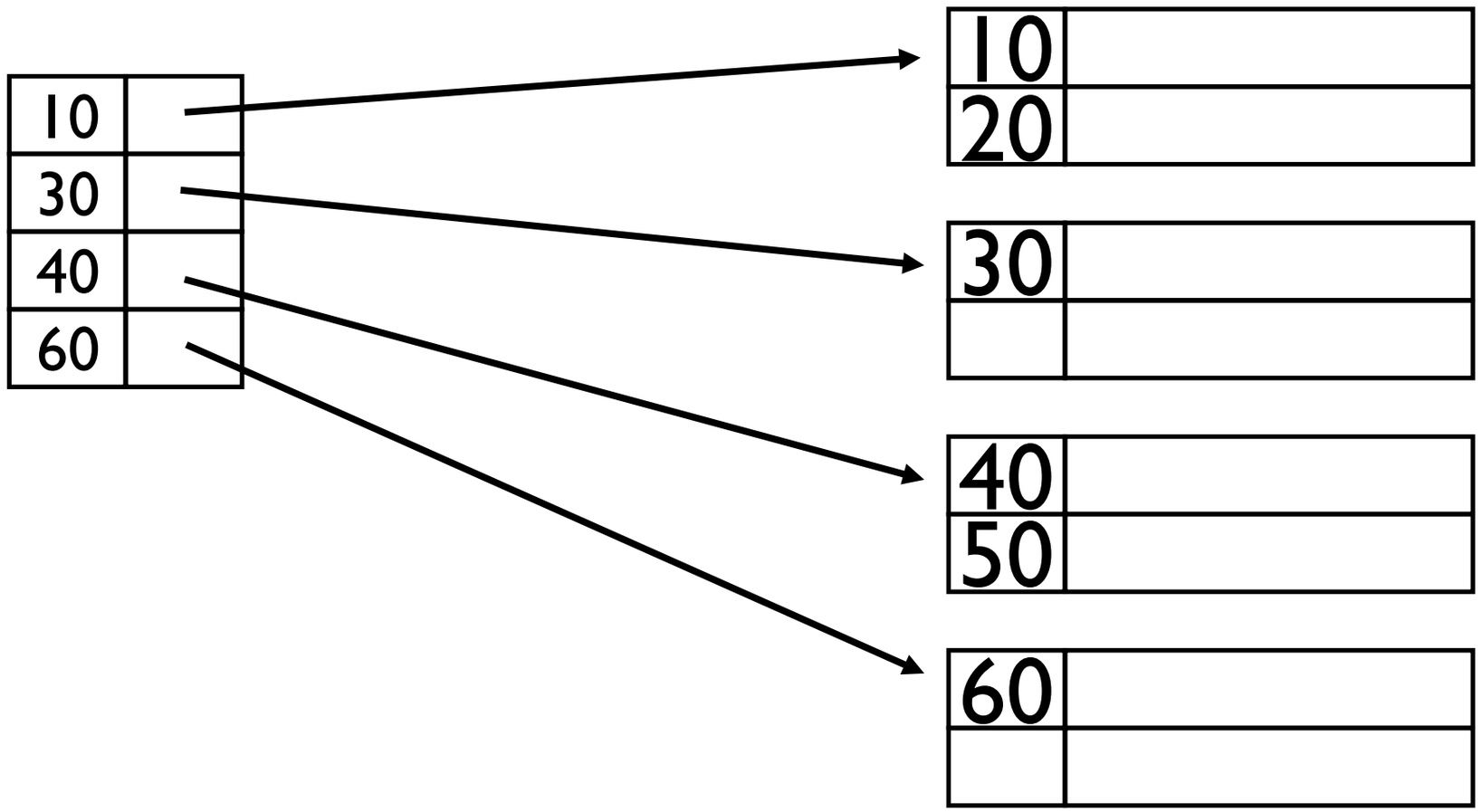
Suppression dans index dense

– supprimer l'enregistrement 30



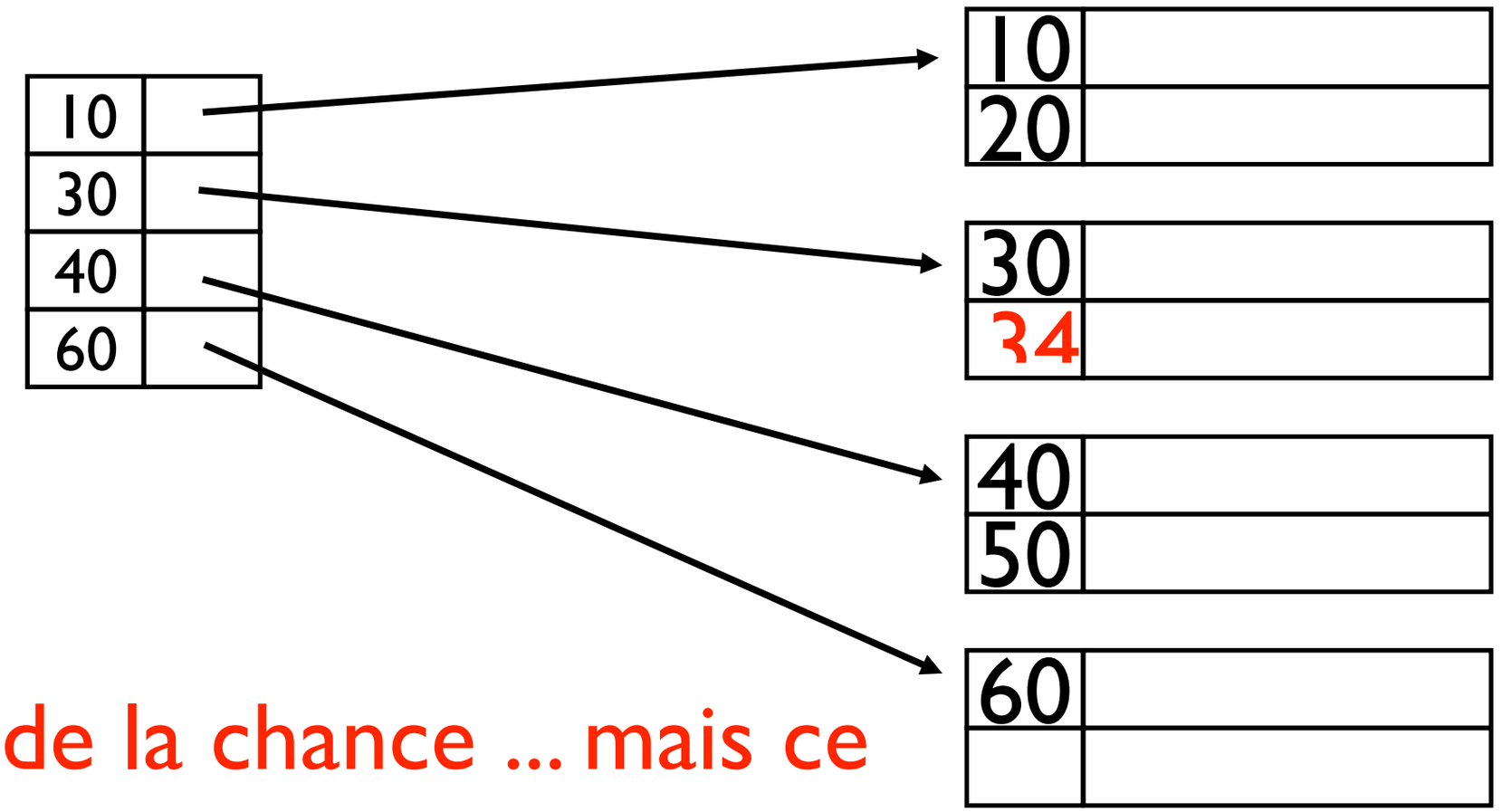
La suppression dans un index primaire dense se fait de la même façon que dans un

Insertion dans index creux



Insertion dans index creux

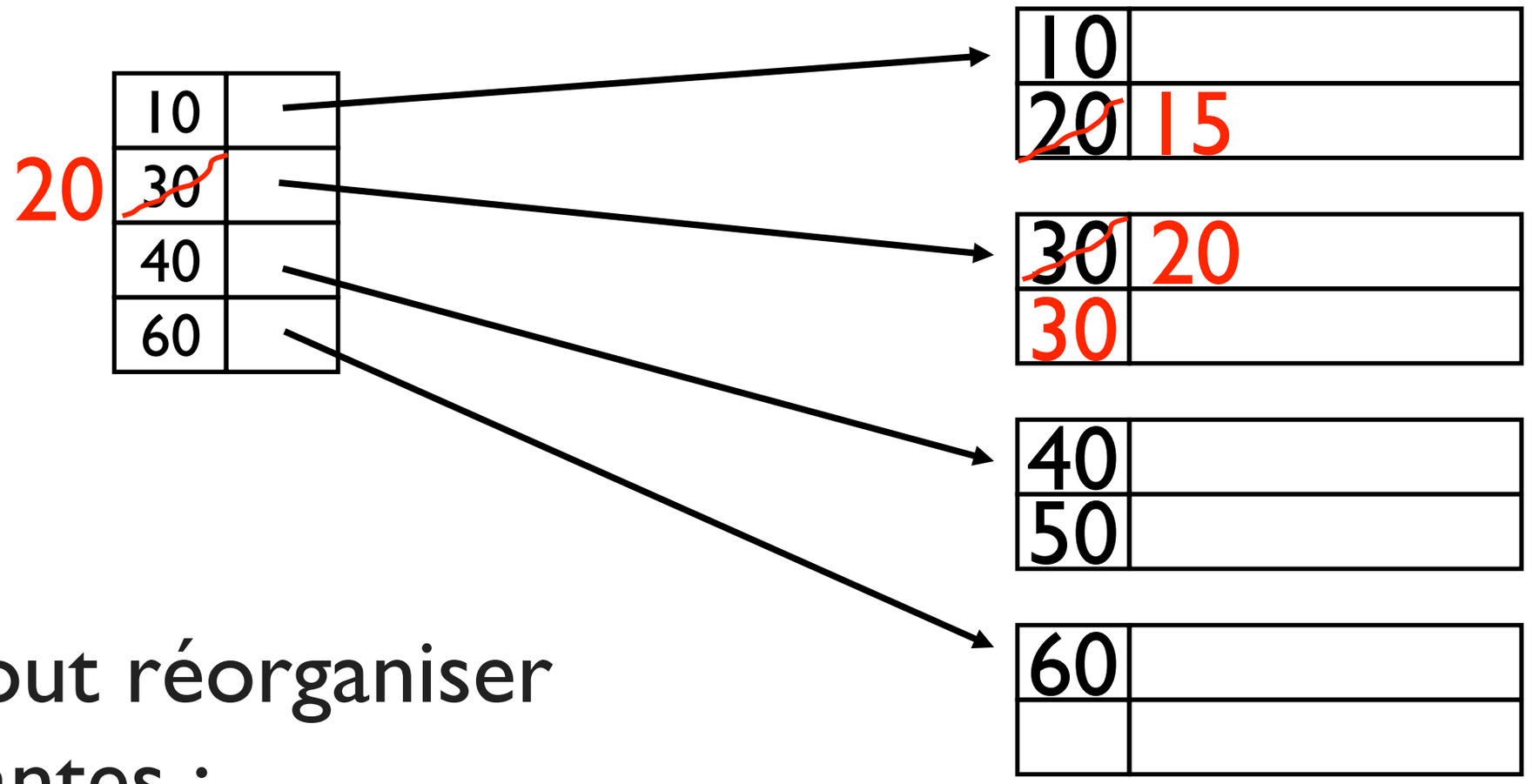
- insérer l'enregistrement 34



- On a de la chance ... mais ce n'est pas toujours le cas

Insertion dans index creux

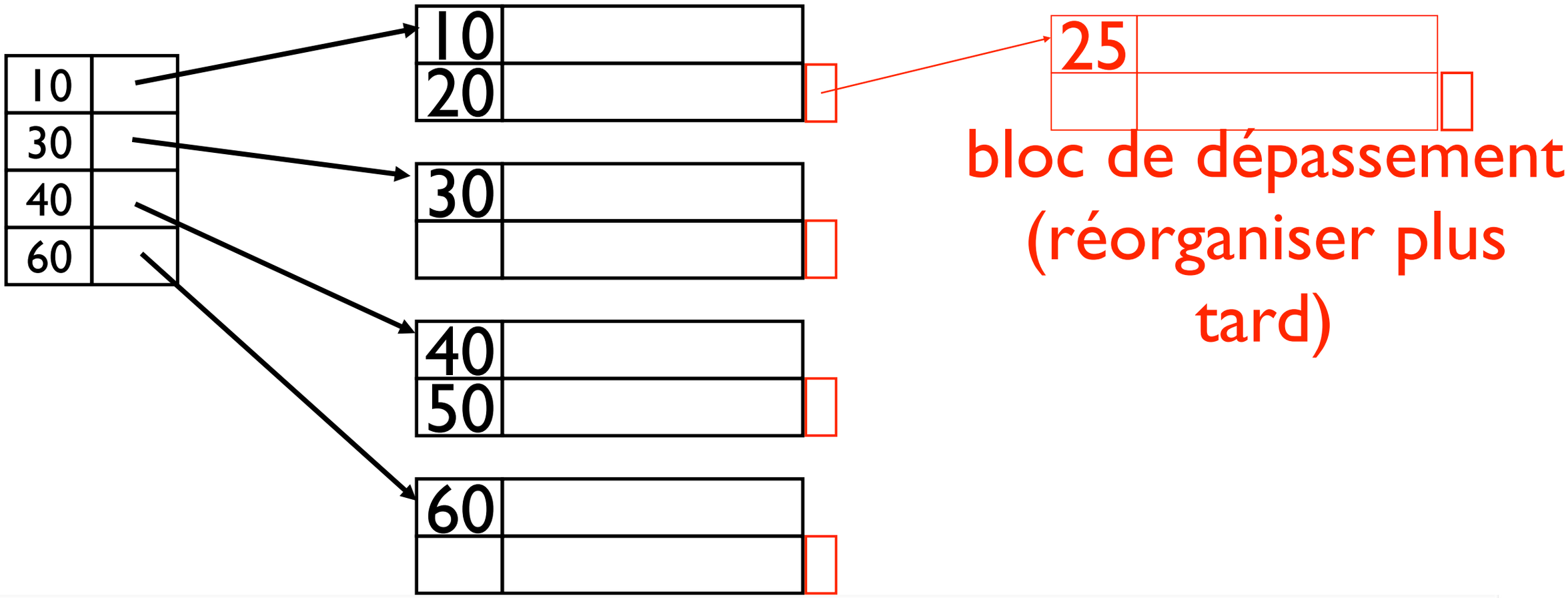
– insérer 15



- Il faut tout réorganiser
- des variantes :
  - insérer un nouveau bloc
  - m.a.j index

Insertion dans index creux

–insérer 25

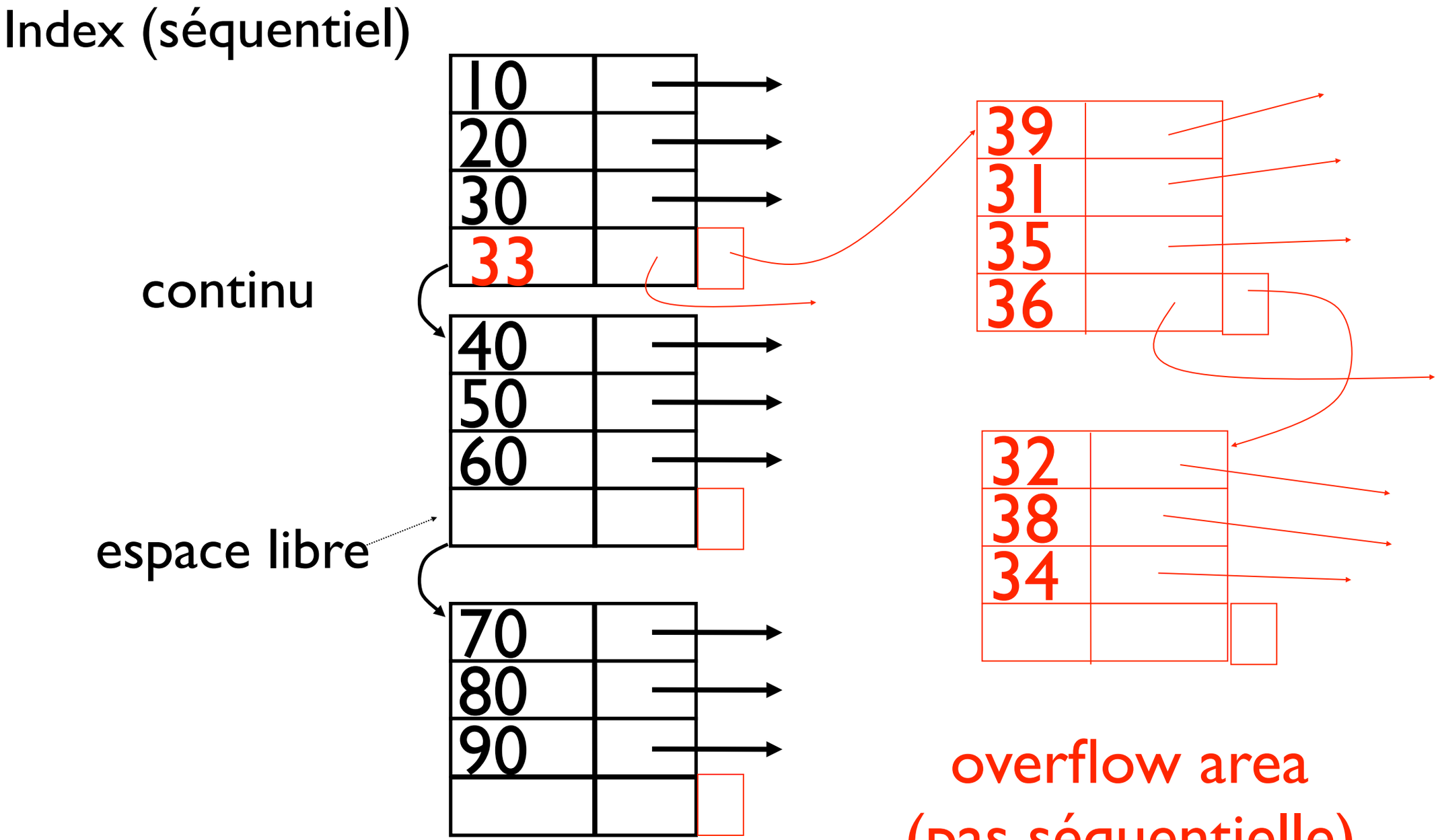


De nombreuses questions :

Quand réorganiser ? Quel coût?

Les **B-arbres** offrent des réponses plus convaincantes

Insertion ++


Index (séquentiel)
10
20
30
33
40
50
60
70
80
90
continu
espace libre
39
31
35
36
32
38
34
overflow area
(pas séquentielle)

Index basiques

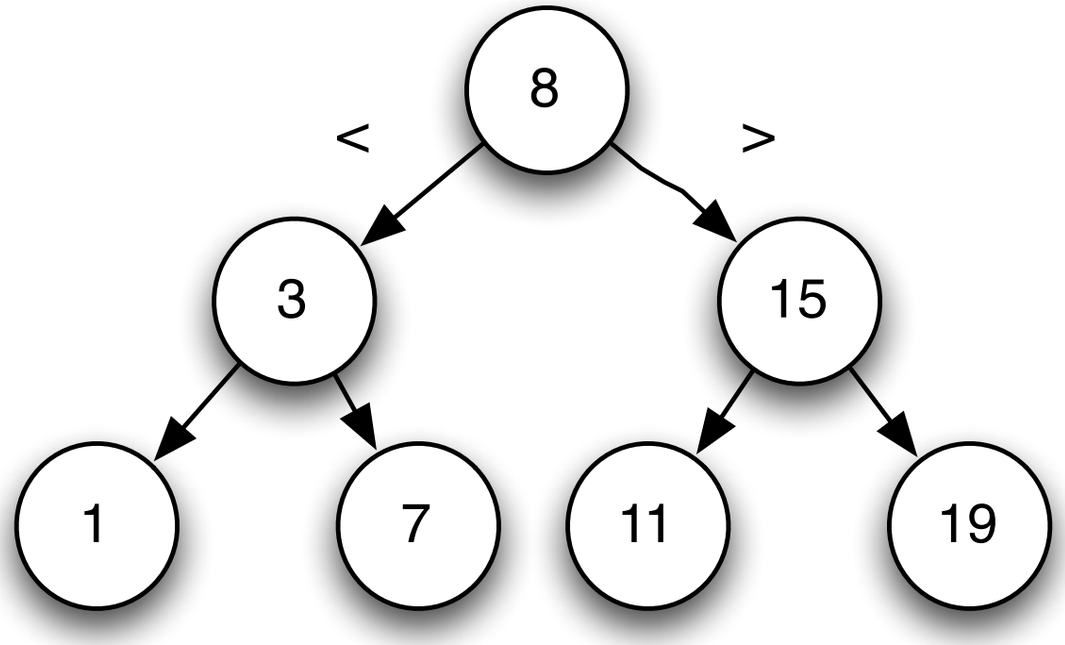
- Avantages :
  - Algorithmes très simples
  - L'index est un fichier séquentiel
    - facile pour effectuer des passes dessus
- Inconvénients :
  - Insertions coûteuses, et/ou
  - la notion de séquence est perdue à cause des overflows (blocs de dépassement)
  - des réorganisations sont nécessaires

Index basé sur les B(+)-Arbres

Rappel ABR :

- Niveau recherche :

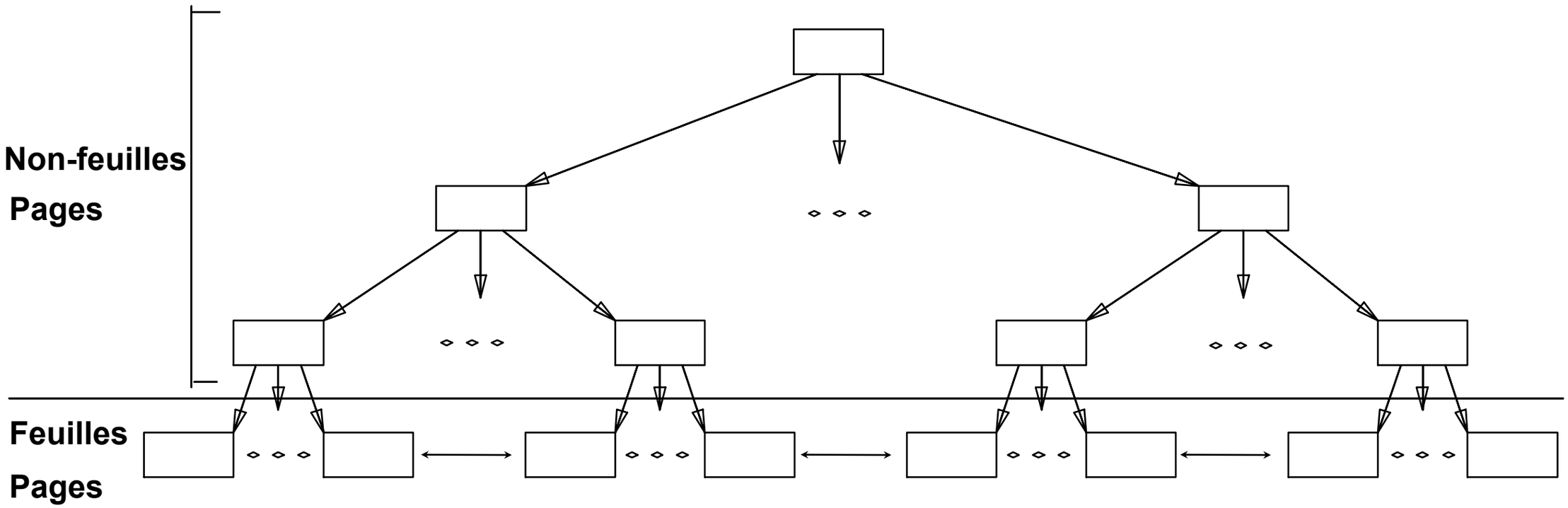
- Généralisation des arbres binaires de recherche.



B-Arbres

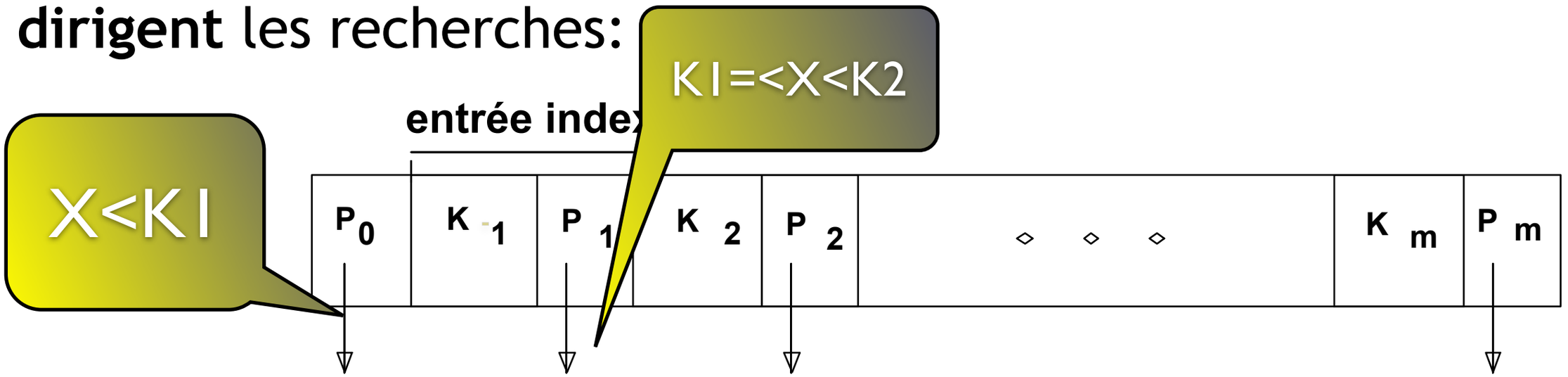
- Chaque noeud correspond à un bloc
- Les B-arbres sont équilibrés (toutes les feuilles à la même profondeur).
    - Garantie d'accès efficaces
- B-arbres garantissent une occupation minimale
- n : nombre maximum de clés dans le noeud, le nombre minimum étant n/2.
- Exception : la racine peut contenir une unique clé
- m+1 pointeurs associés (m est le nombre de clés contenues)

B-arbre +



•Les noeuds feuilles contiennent les **enregistrements** et sont **chainés**.

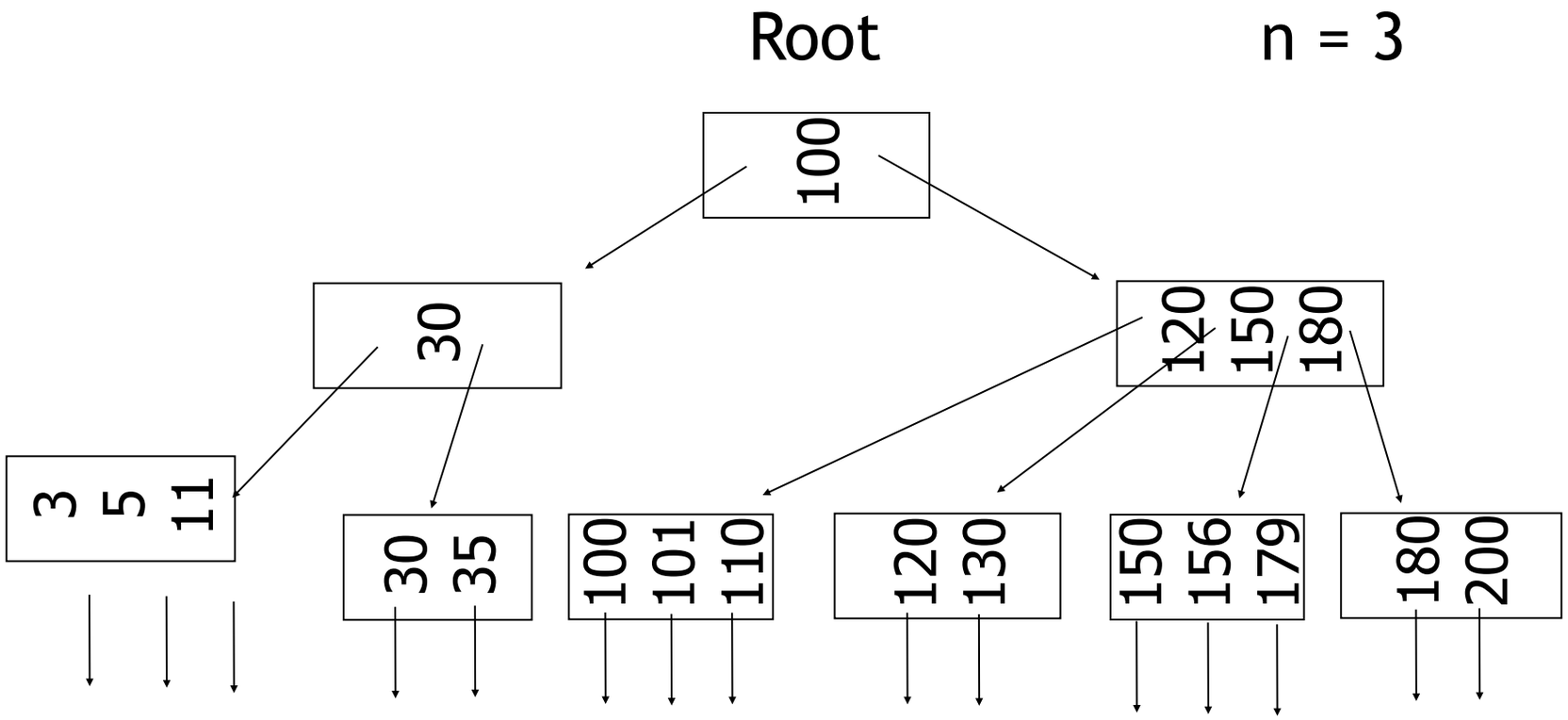
•Les noeuds non-feuilles contiennent les **entrées** de l'index et **dirigent** les recherches:



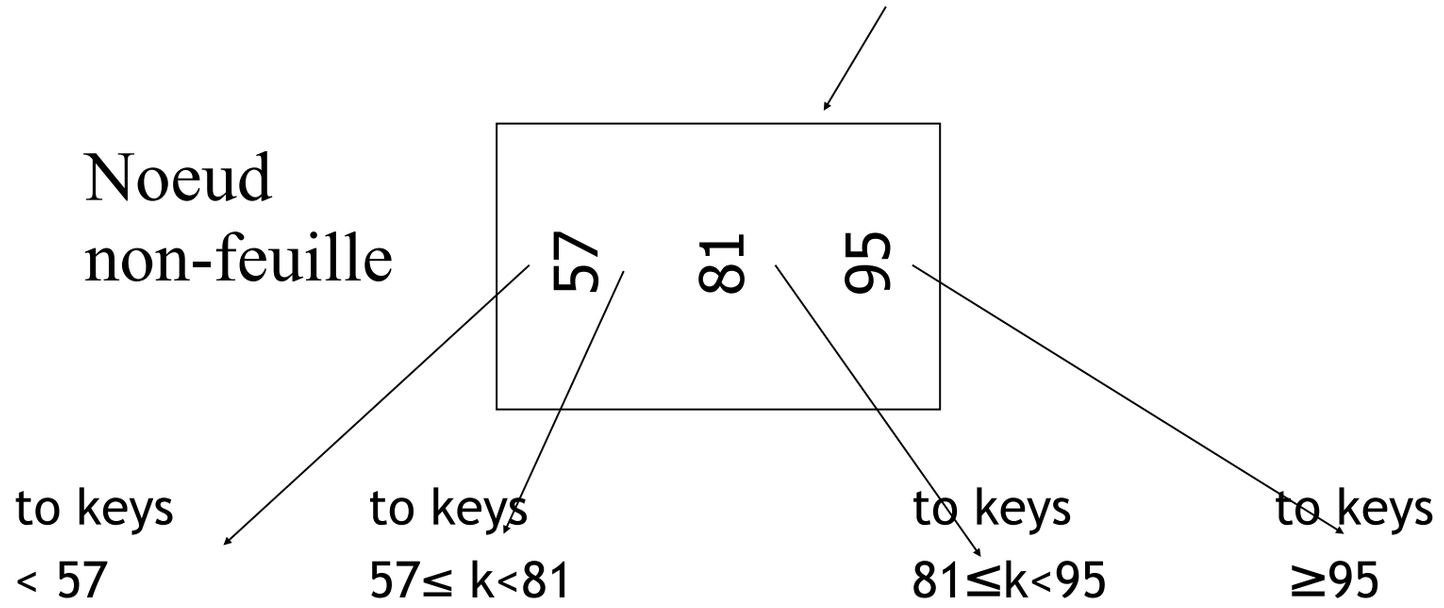
B-Arbres

- Format des noeuds : (p_1,k_1, . . ., p_n,k_n,p_{n+1})
  p_i: pointeur, k_i: clé de recherche
- Un noeud avec *m* pointeurs a *m* enfants sous arbres correspondants.
- *n+1* ème entrée a seulement 1 pointeur. Au niveau des feuilles ces pointeurs identifient le noeud suivant (liste chaînée).
- *propriété* : *i*-ème sous-arbre contient des enregistrement avec une clé de recherche $k < k_i$, i+1-ème sous-arbre contient des entrées avec une clé $k >= k_i$.

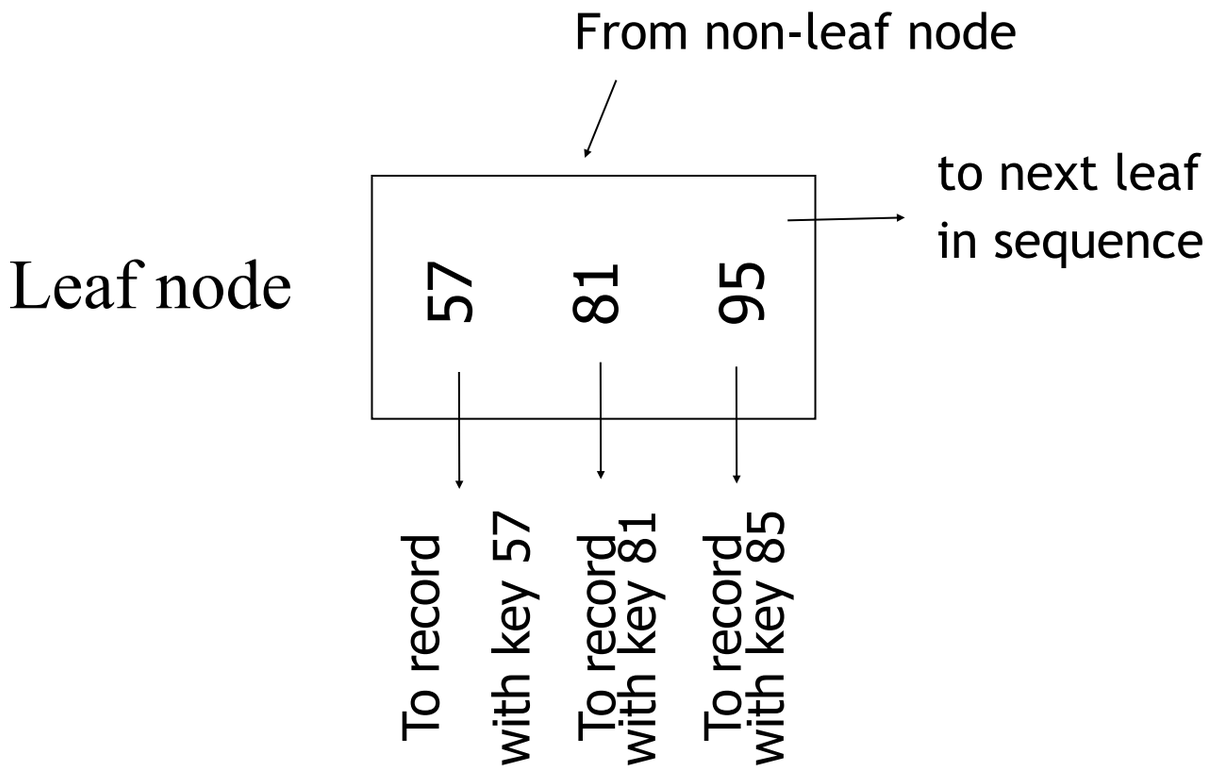
Exemple (1)


Root
n = 3
100
30
120 150 180
3 5 11
30 35
100 101 110
120 130
150 156 179
180 200

Exemple (2)


Noeud
non-feuille
57
81
95
to keys
< 57
to keys
57≤ k<81
to keys
81≤k<95
to keys
≥95

Exemple(3)


From non-leaf node
Leaf node
57
81
95
to next leaf
in sequence
To record
with key 57
To record
with key 81
To record
with key 85

B-Arbre : espace mémoire

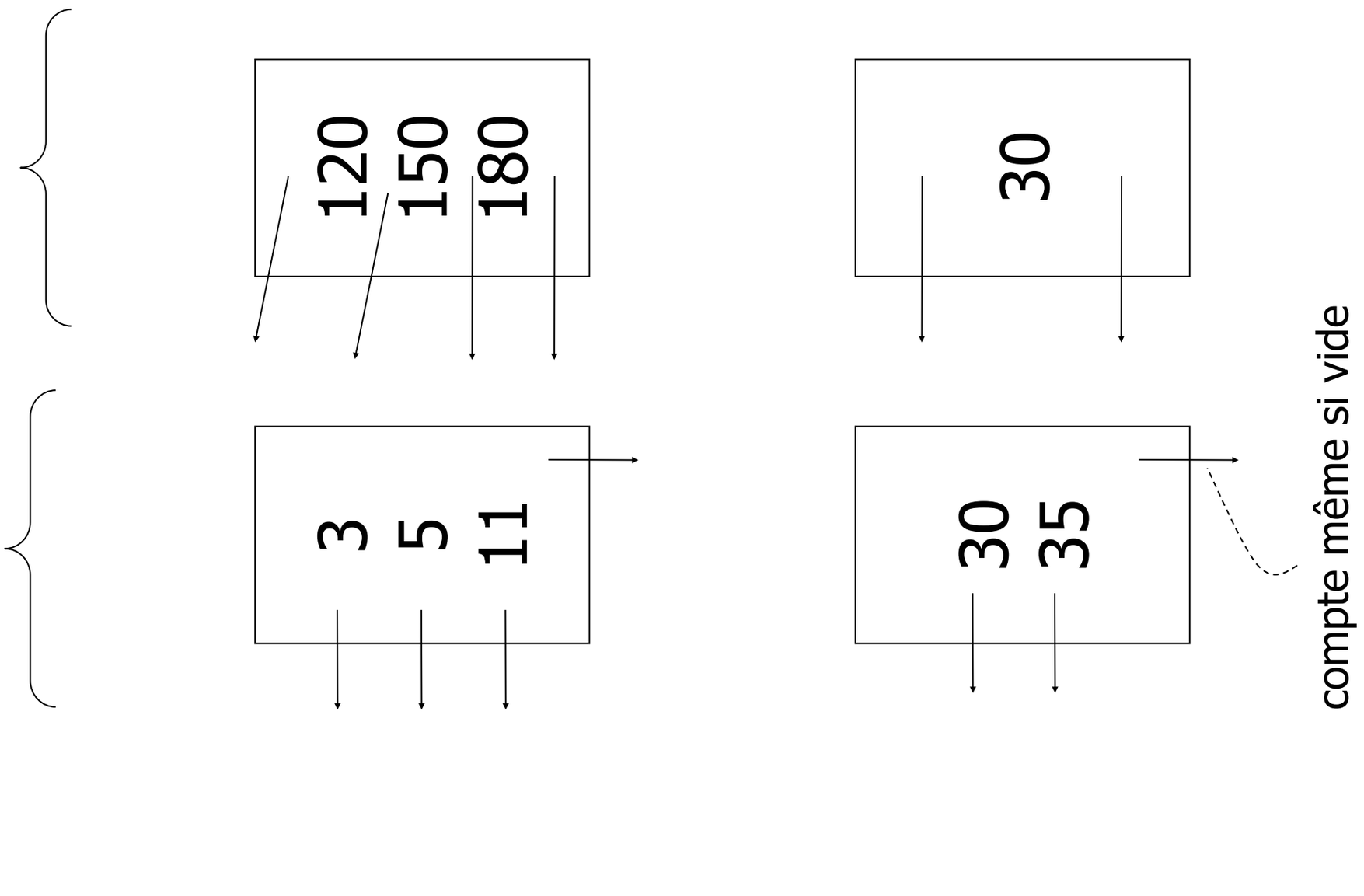
n = 3

Noeud saturé

Noeud min.

Non-feuilles

Feuilles



Utilisation mémoire

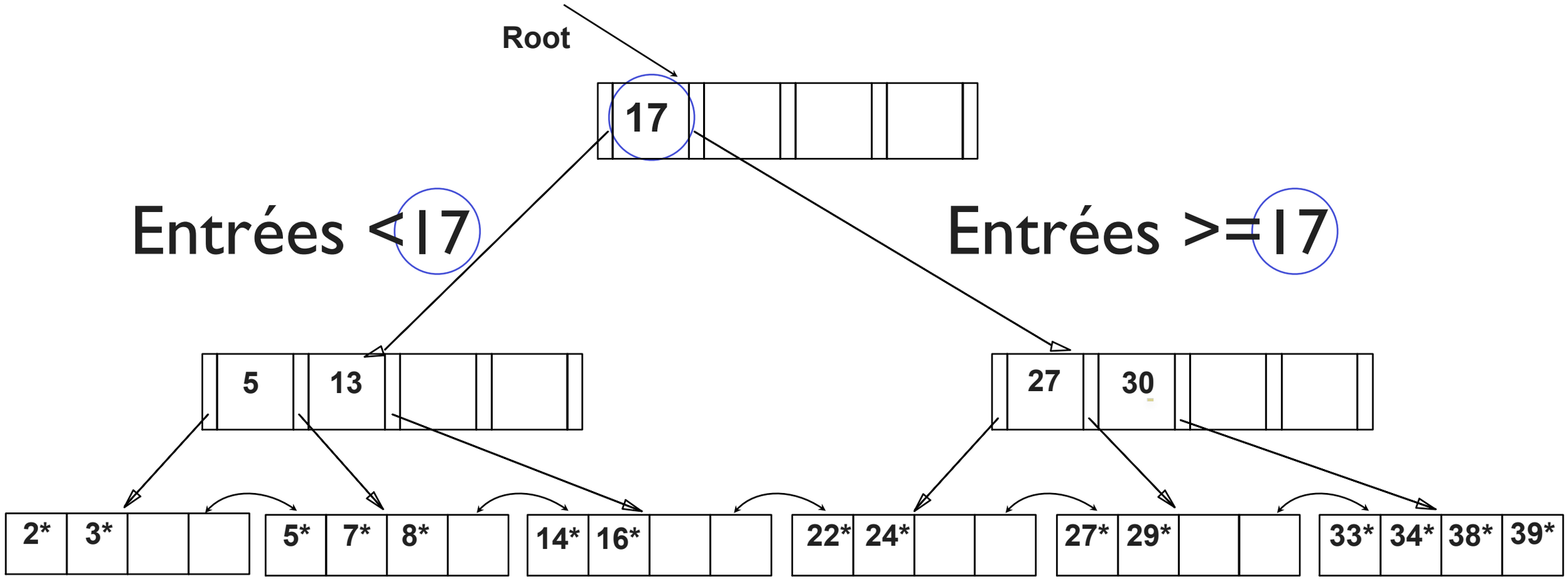
Nombre de pointeurs/clés pour un B-arbre

	Max ptrs	Max clés	Min ptrs→data	Min clés
Non-feuilles (non-racine)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
feuilles (non-racine)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Racine	n+1	n	1	1

Requêtes

- Recherche avec la clé *k*; On commence par la racine
- A un noeud donné, trouver la "plus proche clé" k_i et suivre à gauche (p_i) ou droite (p_{i+1}) le pointeur en fonction de la comparaison entre k et k_i.
- Continuer jusqu'à atteindre une feuille.
- Exploration d'un seul chemin de la racine vers la feuille.
- La hauteur du B-arbre est $O(\log_{n/2} N)$ où N est le nombre d'enregistrements indexés.
- → compléxité de la recherche $O(\log N)$

Exemple (4) : recherche



- Trouver 28*? 29*? All > 15* and < 30*
- Insertion / suppression : Trouver l'entrée dans une feuille, puis la changer. Besoin d'ajuster les parents dans certains cas.
  - Et répercussion dans les noeuds supérieurs de l'arbre

Insertion

- Toujours insérer dans la feuille adéquate
- Arbre grandit de bas en haut
- Plusieurs cas possibles :
  - Espace libre dans la feuille : facile
  - plus de place dans la feuille : plus dur
    - répercution sur les ancêtres
    - cas extrême : jusqu’à la création d’une nouvelle racine

Insertion d'un enregistrement

- Si le B-arbre est vide :
  - Créer une racine de type feuille;
  - y insérer l'enregistrement;
- Sinon rechercher la feuille *L* où insérer l'enregistrement :
  - Si *L* n'est pas pleine
    - insérer l'enregistrement en préservant l'ordre

- Si *L* est pleine :
    - Créer une nouvelle feuille *L'*;
    - Répartir les enregistrements entre *L* et *L'*;
    - M.à.j le B-arbre en remontant dans celui-ci;

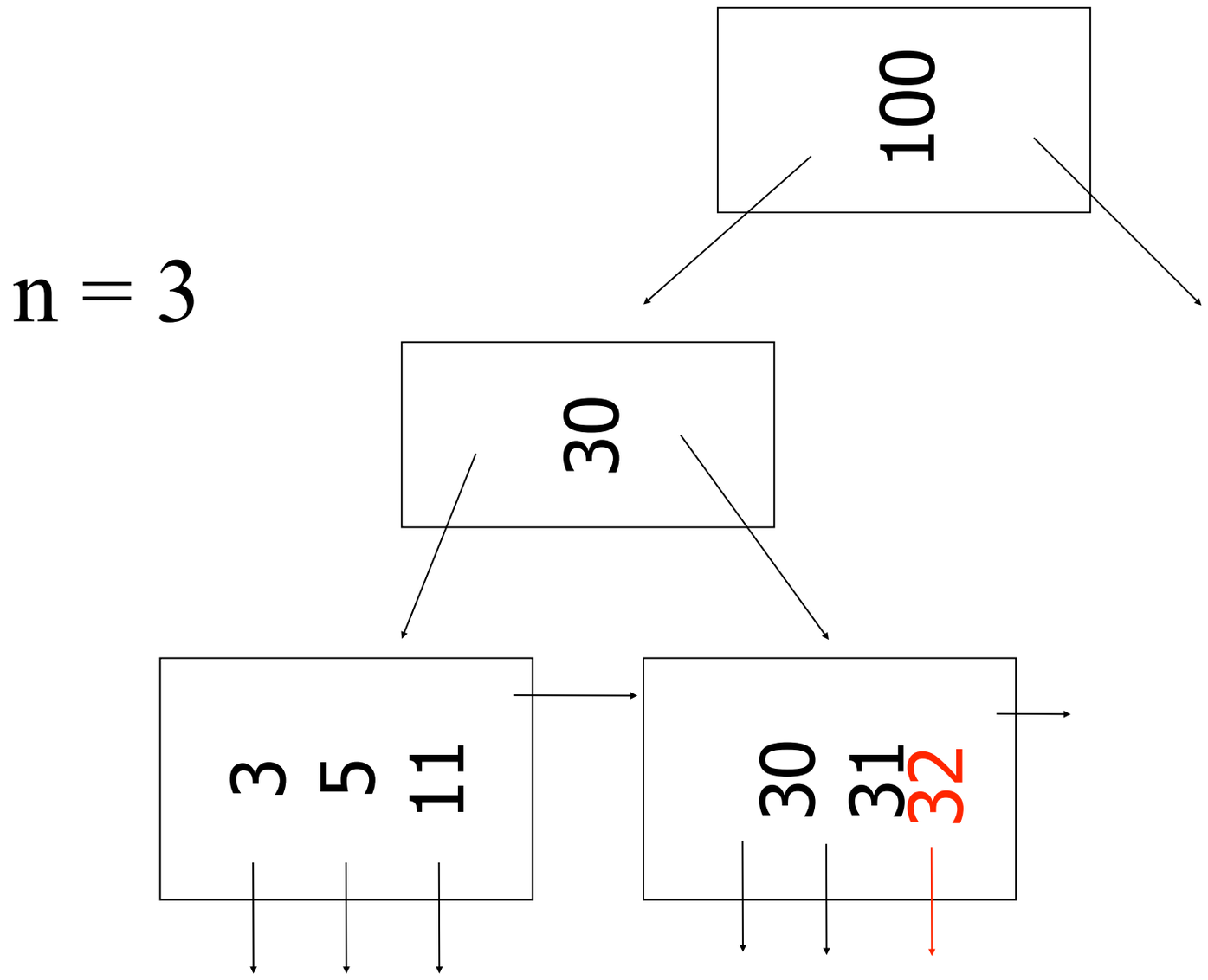
La division peut se propager tant que les ancêtres de *L* sont pleins.

Si on remonte jusqu'à la racine, on la divise en deux et on crée une nouvelle racine;

Insertion (espace libre)

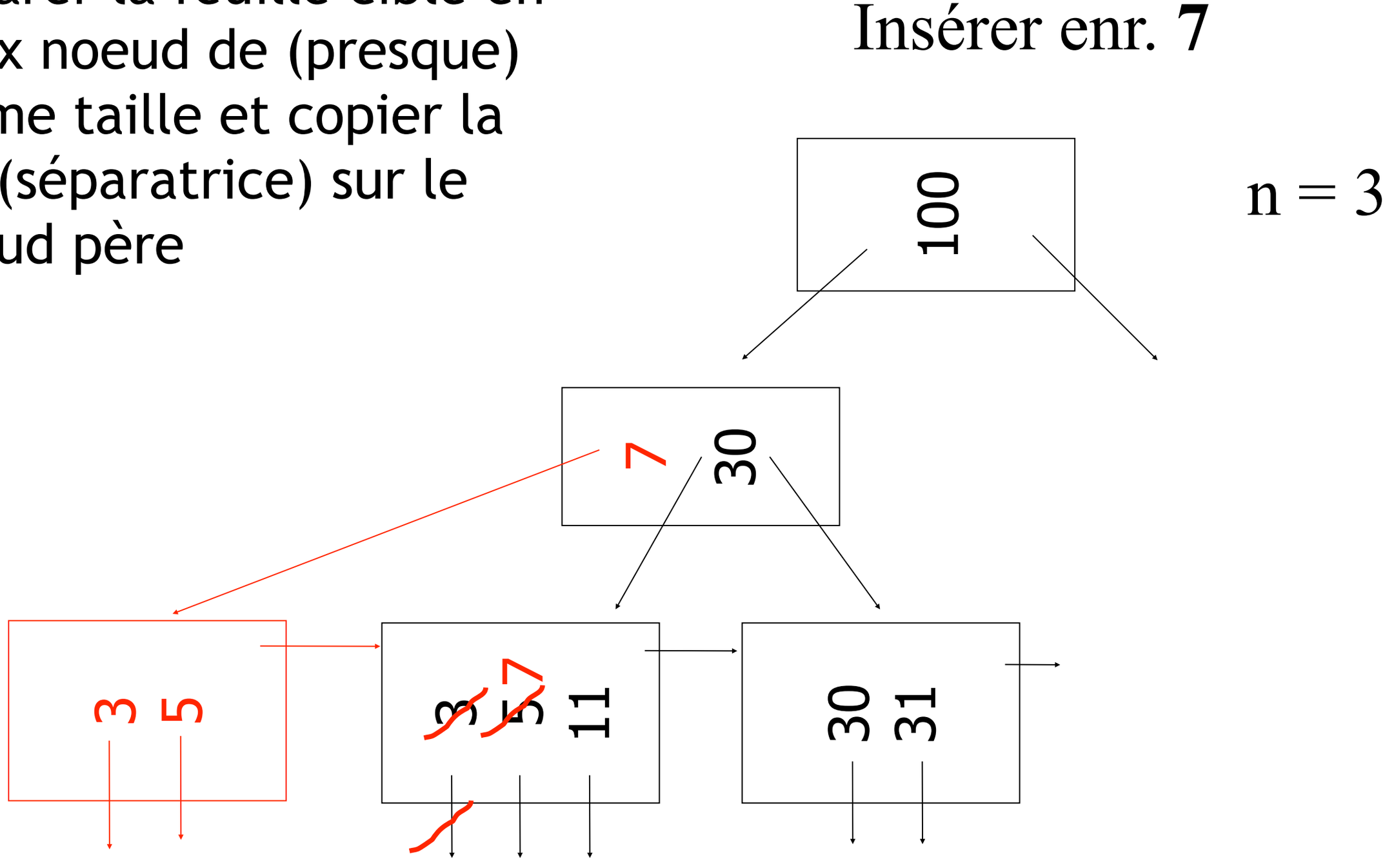
Insérer enr. **32**

n = 3



Plus de place (I)

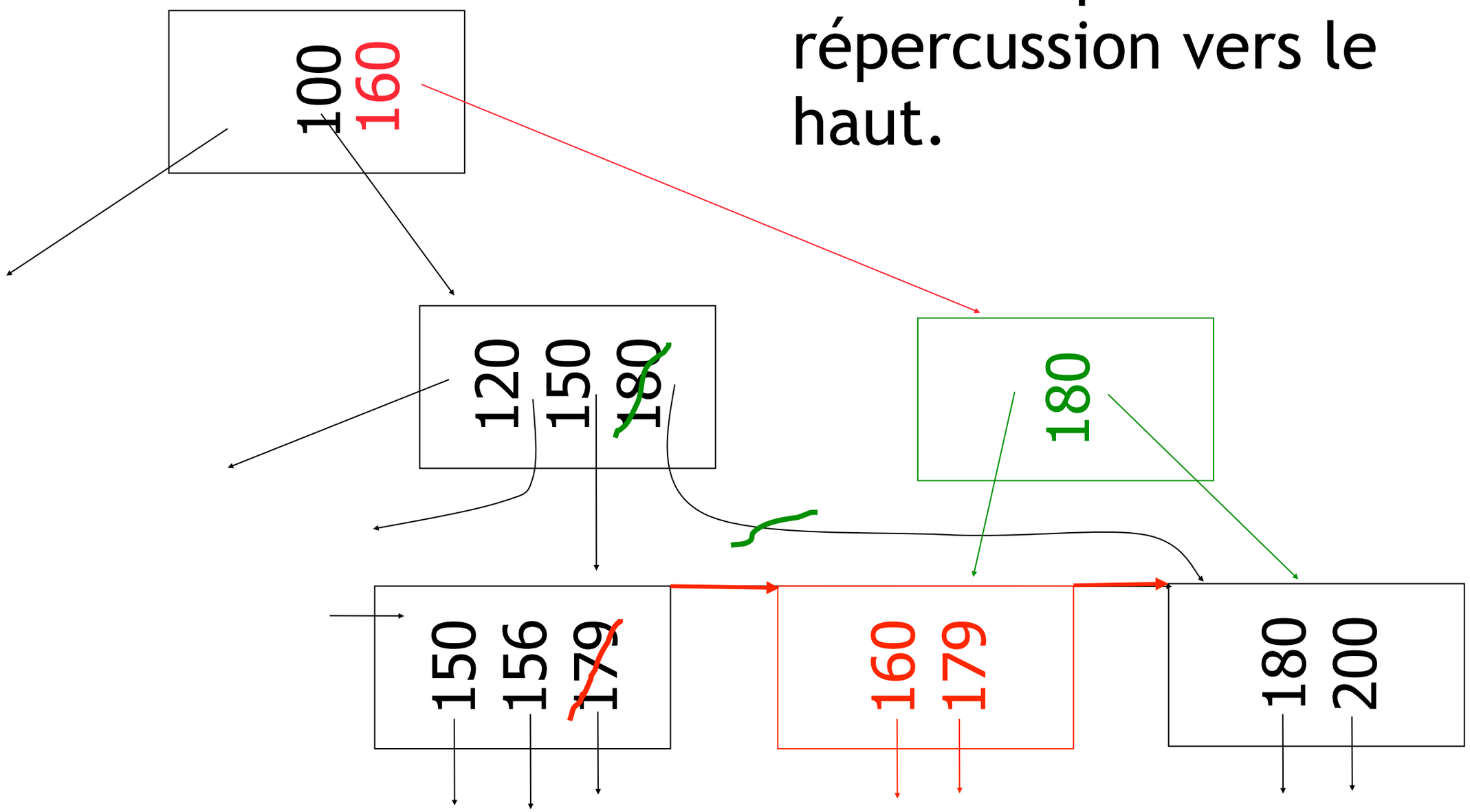
- Séparer la feuille cible en deux noeud de (presque) même taille et copier la clé (séparatrice) sur le noeud père



Plus de place (II)

Insérer enr. **160**

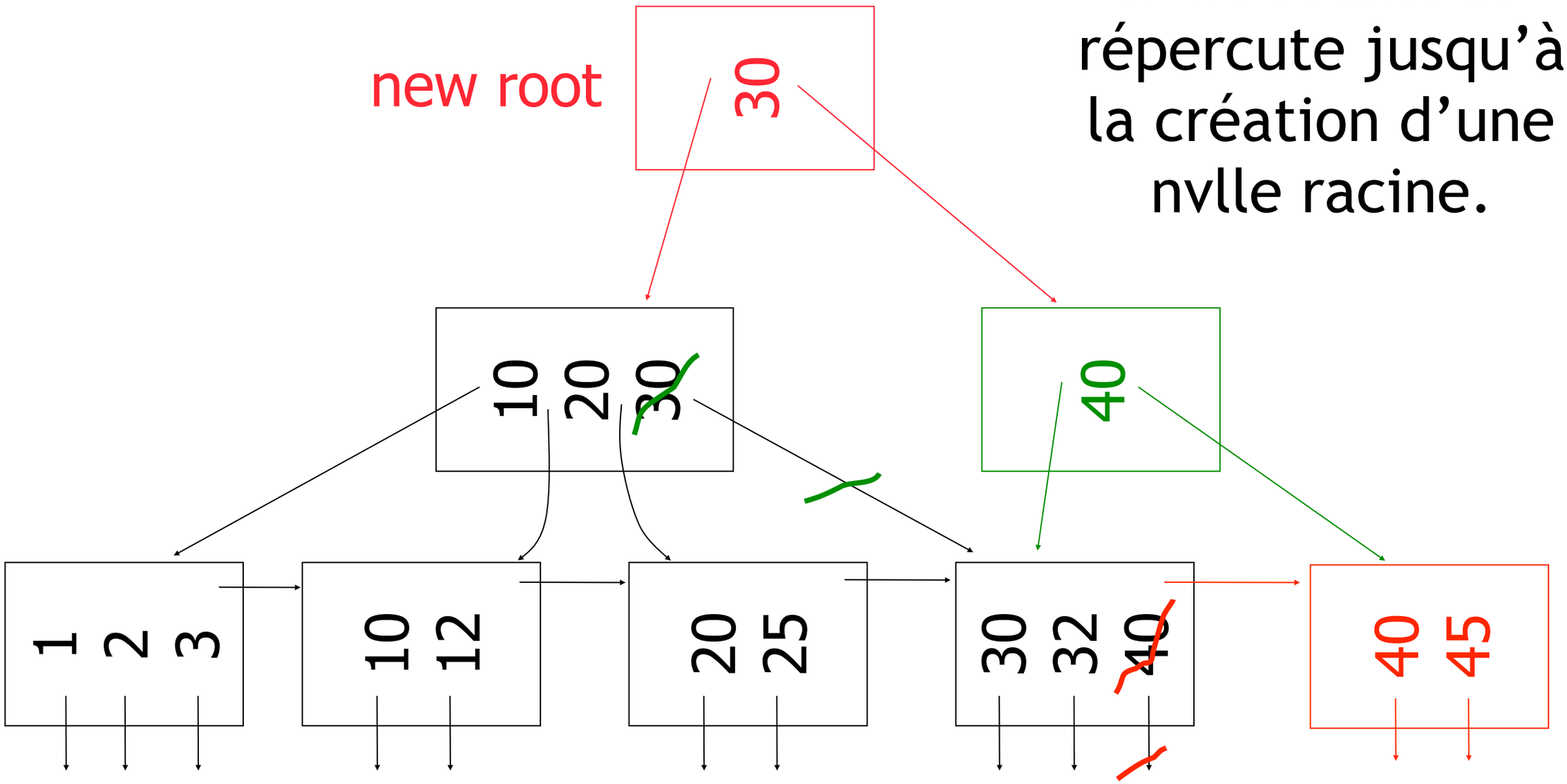
Séparation en deux de la feuille pleine et répercussion vers le haut.



Plus de place (III) : nvlle racine

Insérer enr. **45**

La création d'une nvlle feuille se répercute jusqu'à la création d'une nvlle racine.



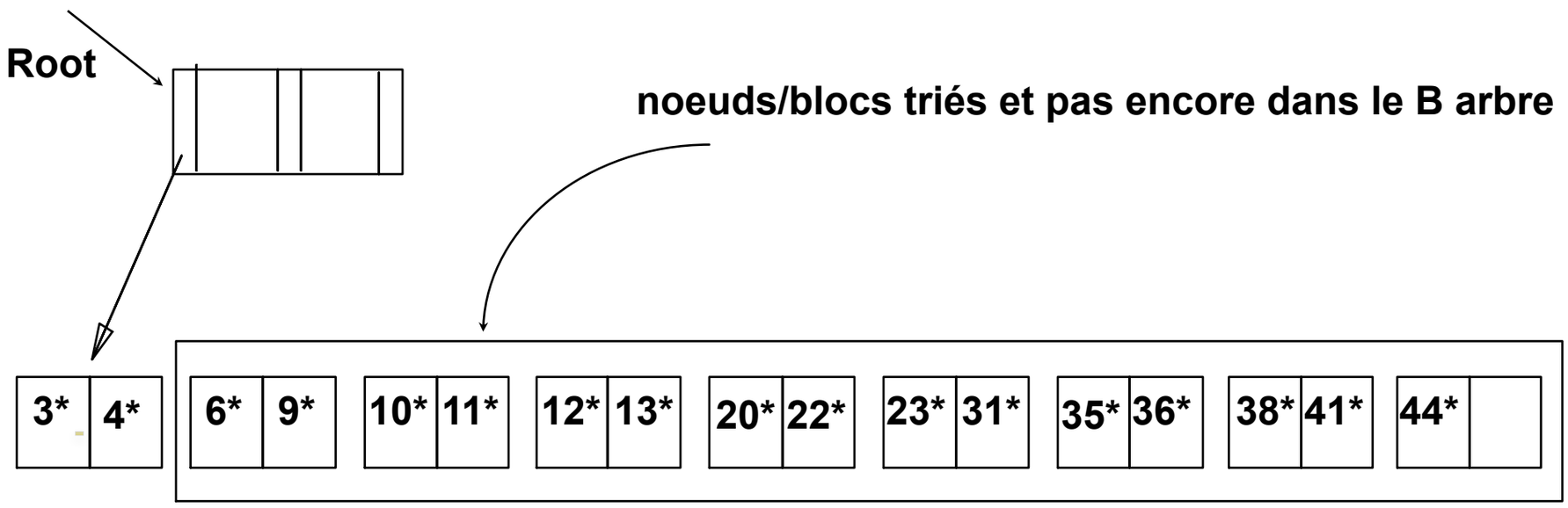
Ex: Construction de l'arbre

- N=3 (chaque bloc contient 3 valeurs et 4 pointeurs)

- Construire le B-arbre à partir de la liste de valeurs suivante : (24, 123, 3, 12, 32, 1, 2, 5) en insérant les valeurs une à une.

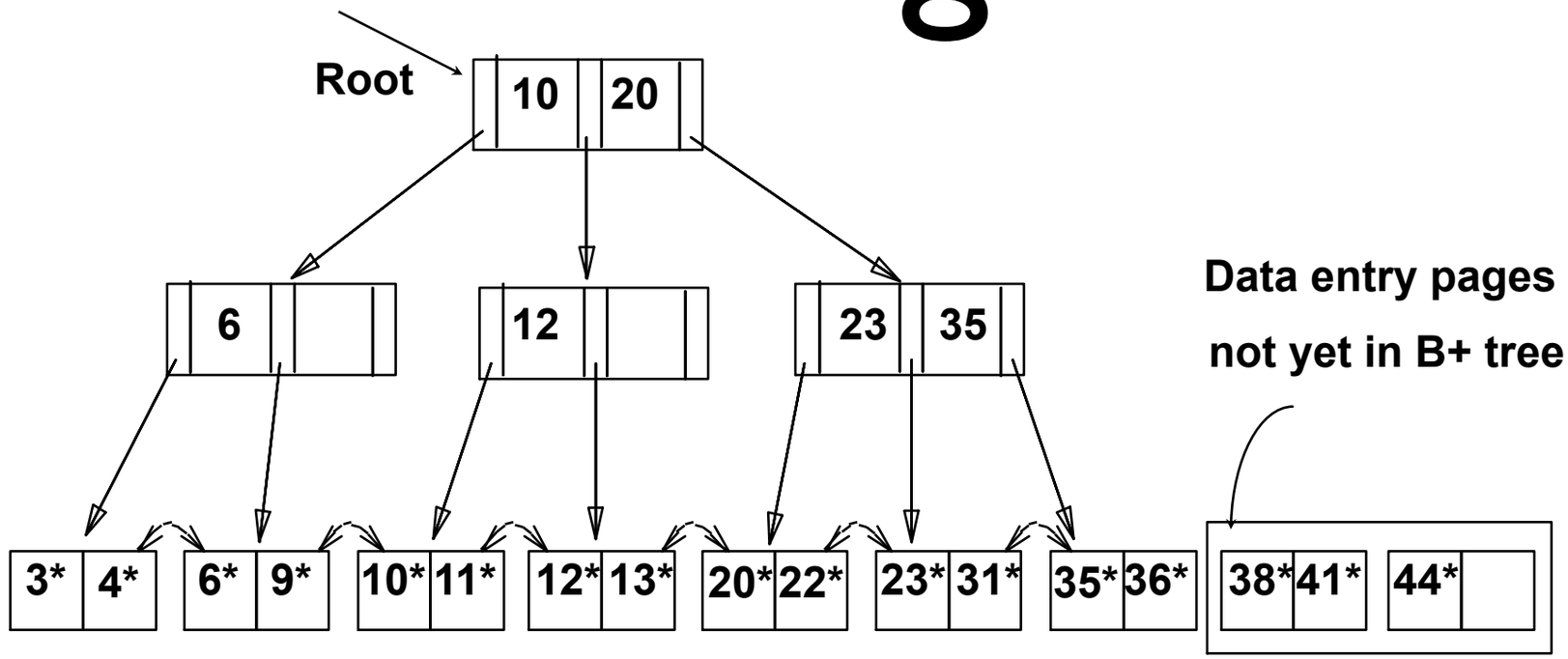
Bulk Loading

- Si on veut créer un B-Arbre un attribut d'une grande collections d'enregistrements, faire des insertions une à une est très lent
- Bulk Loading est plus efficace.
- Initialisation: trier les entrées, insérer un pointeur sur le premier bloc feuille dans une nouvelle racine.

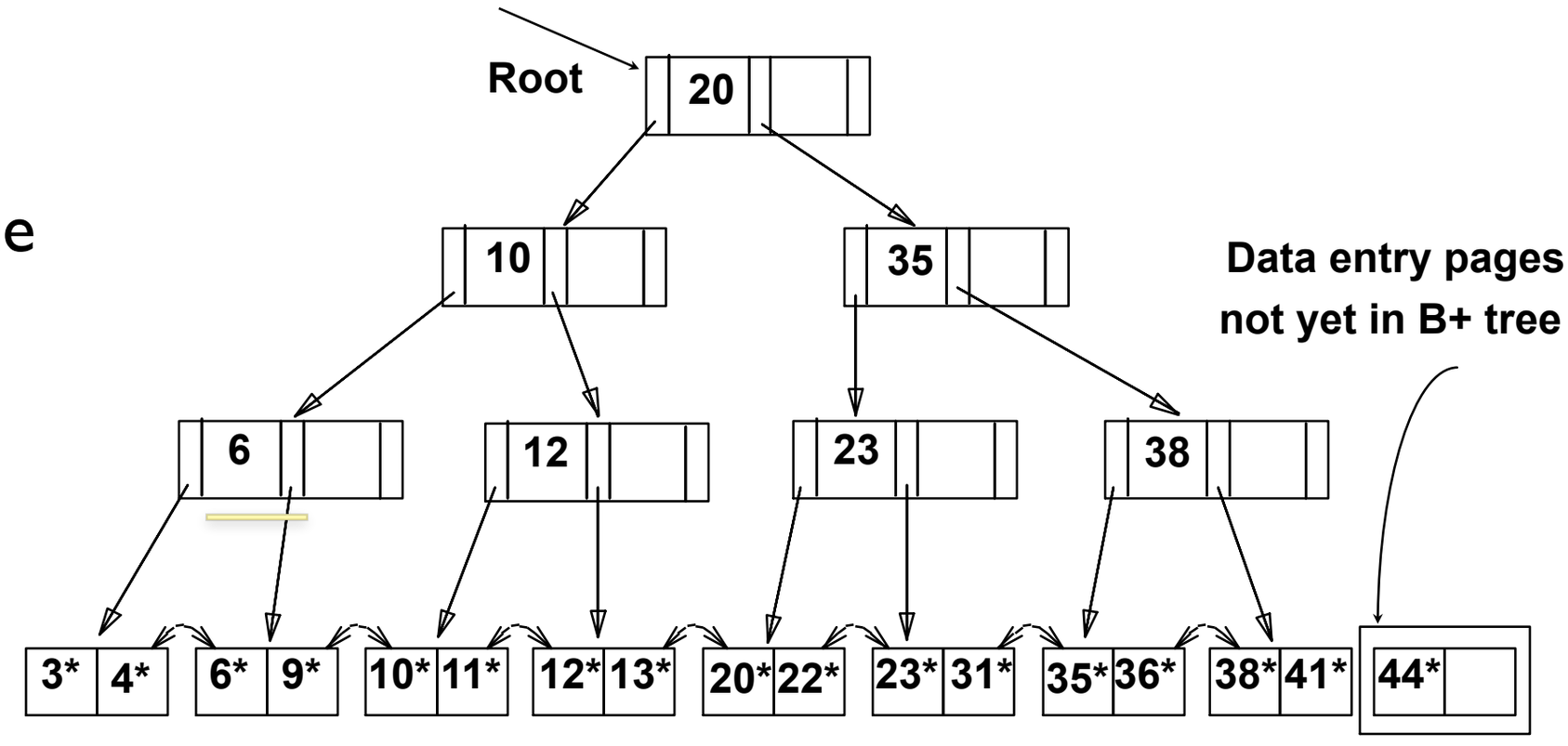


Bulk Loading

- Les blocs feuilles entrent toujours par la droite sur le noeud (non feuille) le plus à droite.Quand il est plein, le couper en deux. (Le Split peut se propager sur le chemin le plus à droite jusqu'à la racine.)



- Beaucoup plus rapide que des insertions successives !!!



Suppression

- Localiser le noeud correspondant
- Supprimer l'entrée spécifiée
- Plusieurs cas possibles
  - Le noeud feuille possède encore suffisamment d'entrées (cas facile)
  - Fusion avec des éléments de la fratrie
  - Redistribution des clés
  - Fusion et redistribution aux niveaux supérieurs

Plus précisément

- Effacement d'un enregistrement de clé k d'un B-arbre.
  - Si le B-arbre n'est pas vide.
    - Soit l la feuille contenant l'enregistrement à supprimer (trouvé par une recherche). Après avoir supprimé l'enregistrement de cette feuille, trois cas peuvent se présenter.

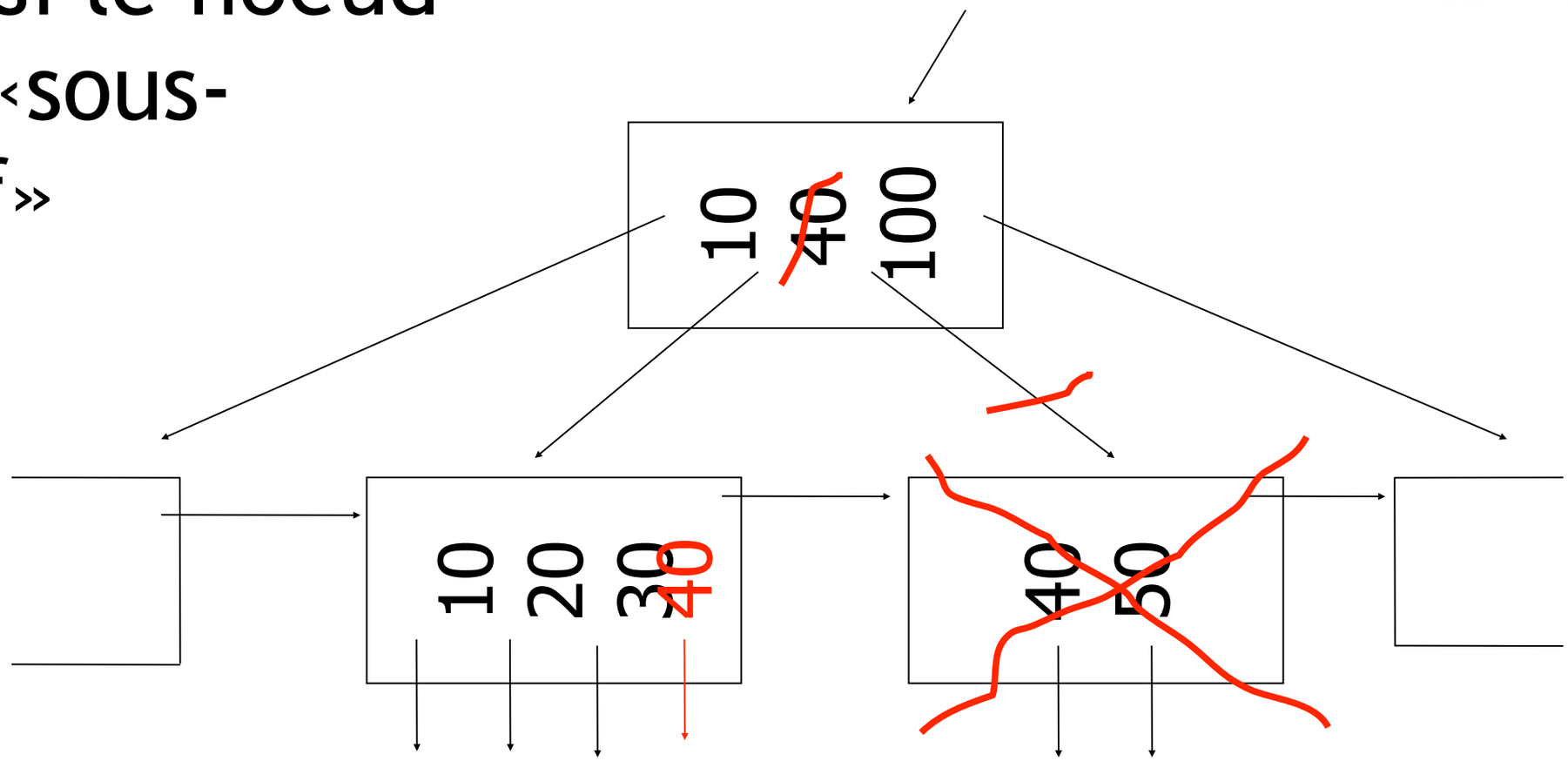
1. La feuille l est la racine du B-arbre et elle est devenue vide. On supprime cette feuille et le B-arbre devient vide.

2. La feuille l compte au moins e enregistrements. Si la clé k apparaît dans un noeud d'index, mettre l'index à jour.

- 3. La feuille l ne contient que e − 1 enregistrements. Réorganiser les enregistrements de l et ceux d'une feuille voisine l' (possédant au moins e éléments) d'une des deux manières suivantes.
    - Si l' a plus de e enregistrements, distribuer équitablement les enregistrements entre l et l'.
    - Si l' a exactement e éléments, fusionner les enregistrements de l et de l' et supprimer la feuille devenue vide.
- Propager cette mise à jours en remontant vers la racine. Si l'on arrive à la racine du B-arbre et que celle-ci ne comporte plus qu'un sous-arbre supprimer la racine, le B arbre se réduisant à ce successeur unique.

Suppression (I)

Fusion avec un élément de la fratrie si le noeud est en «sous-effectif»

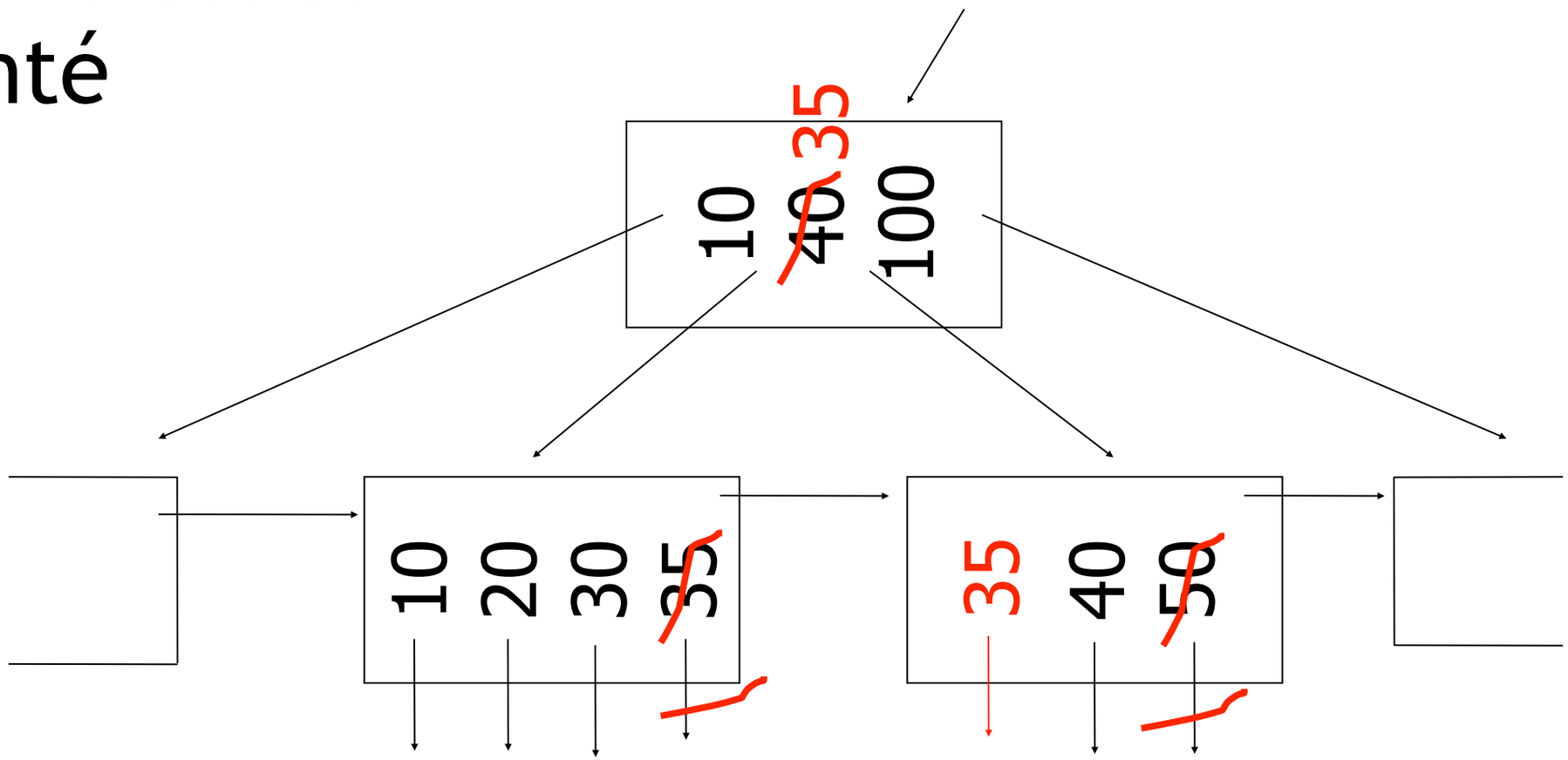


Suppression (II)

Redistribution des clés si le noeud est sous alimenté et que la fratrie est sur-alimenté

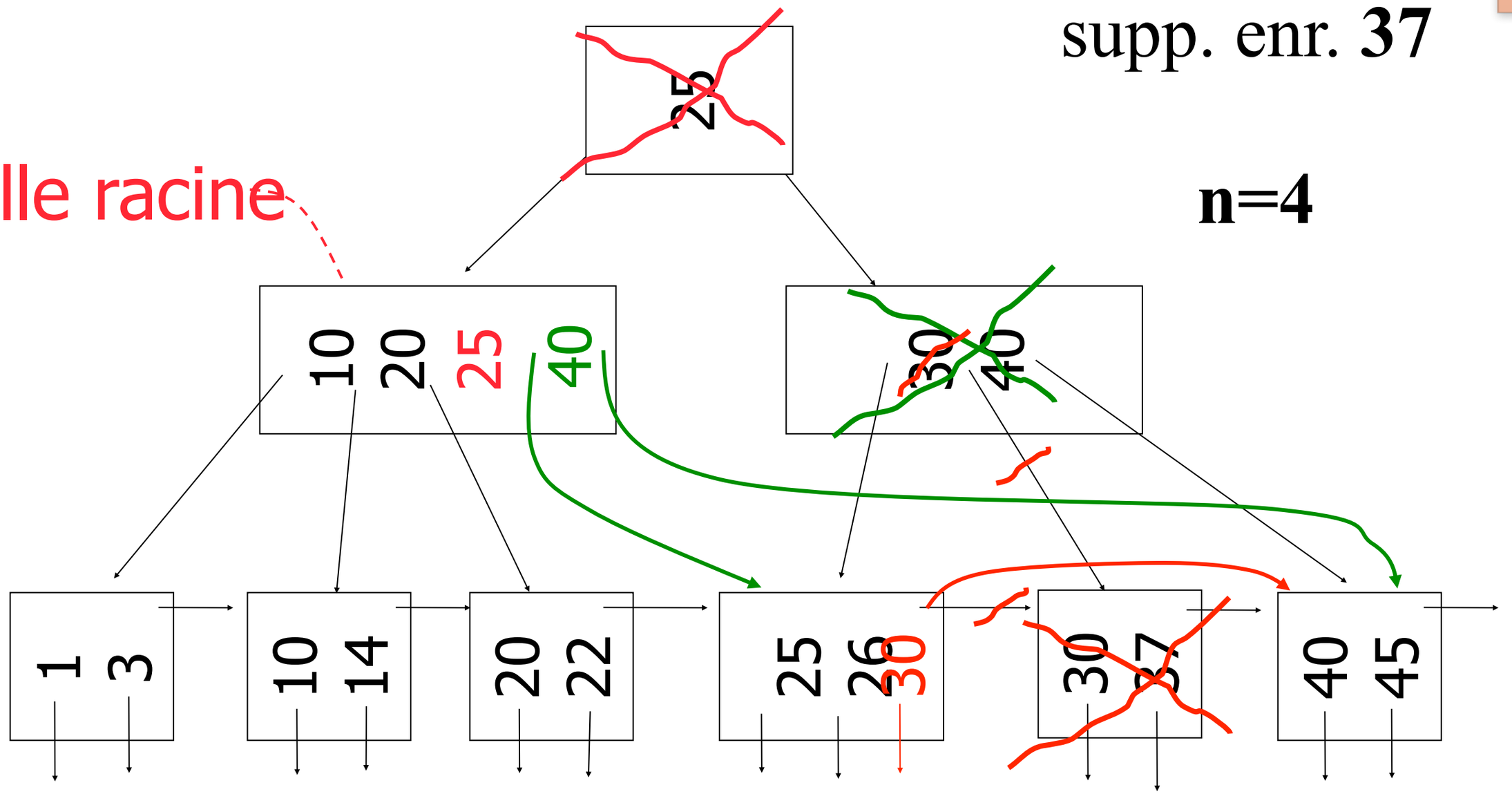
supp. enr. **50**

n=4



Suppression (III)

Suppression racine



En pratique

- Ordre: 200, espace utilisé : 67%. I.e., *average fanout* = 133.
- Capacités :
  - Profondeur 4: 133^4 = 312,900,700 enregistrements,
  - Profondeur 3: 133^3 = 2,352,637 enregistrements.
- On peut souvent charger les niveaux supérieurs en mémoire
  - Level 1 = 1 blocs = 8 Ko,
  - Level 2 = 133 blocs = 1 Mo,
  - Level 3 = 17,689 blocs = 133 Mo.

Tables de hachage

- Des structures d'index basées sur des arbres permettent de retrouver des enregistrements en fonction de leur clé de recherche via une structure arborescente (B-Tree).
- Les tables de hachage assurent aussi ce mapping Clé/ Enregistrement via une **fonction de hachage**.
- Les tables "hash" sont une deuxième forme d'index qui est aussi fréquemment utilisée, mais plutôt pour des relations **temporaires conservées en mémoire**.

Tables de hachage

- Soit une clé de recherche **K**
- Une fonction de hachage **h** :
  - **h(k) ∈ {0,1,...,B-1}**
- B : nombre de **buckets** (bac)

Tables de hachage

- Le choix d'une bonne fonction de hachage est cruciale pour les performances.
  - Une mauvaise fonction produit trop de collisions et dégrade les performances.
  - Une bonne fonction : le nombre attendu de clé doit être similaire pour tous les buckets (distribution homogène).
- C'est très difficile à accomplir pour les clés de recherche qui ont une distribution très asymétriques (e.g. noms).
- Une fonction de hachage fréquente :
  - *K* = 'x1x2 … xn' une chaine de caractères de ***n*** octets

$$h(K) = (x_1 + ... + x_n)\ MOD\ B$$

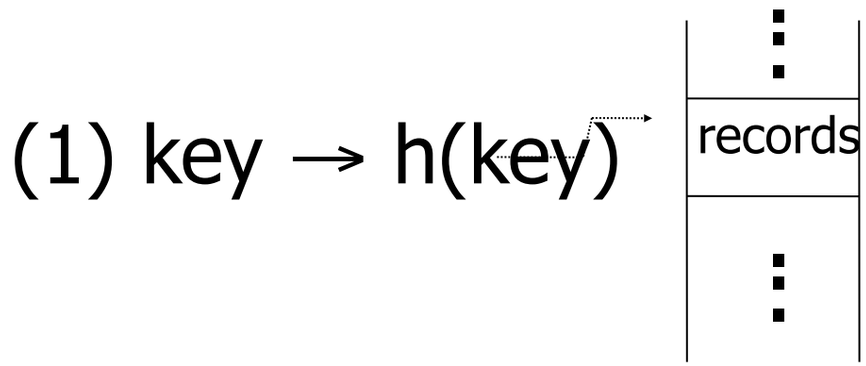
B est souvent un nombre premier.

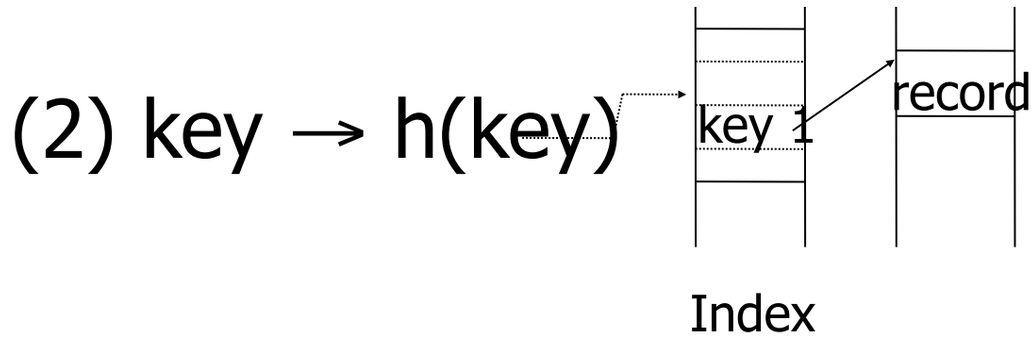
Principe

- Les enregistrements sont répartis entre des bacs regroupés dans un tableau. Un bac peut contenir un ou plusieurs enregistrements.
- On prévoit un nombre de bacs supérieur au nombre d'enregistrements prévus, ce qui donne un nb moyen d'enregistrements par bloc < 1.
- Le numéro de bac dans lequel un enregistrement est placé est calculé à partir de la clé de l'enregistrement par une fonction "hash". Une fonction "hash" idéale répartirait les enregistrements uniformément entre les bacs.
- Pour avoir accès à un enregistrement, il suffit de calculer son numéro de bac à l'aide de la fonction "hash".

- *Bucket*: collection de blocs.
- Initiallement, le bucket se compose d'un bloc.
- Les enregistrements dont la fonction de hachage renvoie b sont stockés dans le bucket *b*.
- Si la capacité du bucket est atteinte, alors une chaine de bacs de dépassement (overflow buckets) est liée.
- On suppose que l'adresse du premier bloc du bucket *i* peut être calculée étant donné *i*.

- Les tables de hachage peuvent accomplir leur mapping directement ou indirectement.





Insertions

- Insertion d'un enregistrement possédant une clé de recherche *K*.
- Calcul de la fonction de hachage $h(K) = i$.
- Insertion de l'enregistrement dans le premier bloc du bac *i* qui possède suffisamment d'espace libre.
- S'il n'y a pas de bloc avec suffisamment d'espace libre alors ajouter un nouveau bloc à la chaine de dépassement (overflow chain) et y stocker l'enregistrement.

Insertions

INSERER:

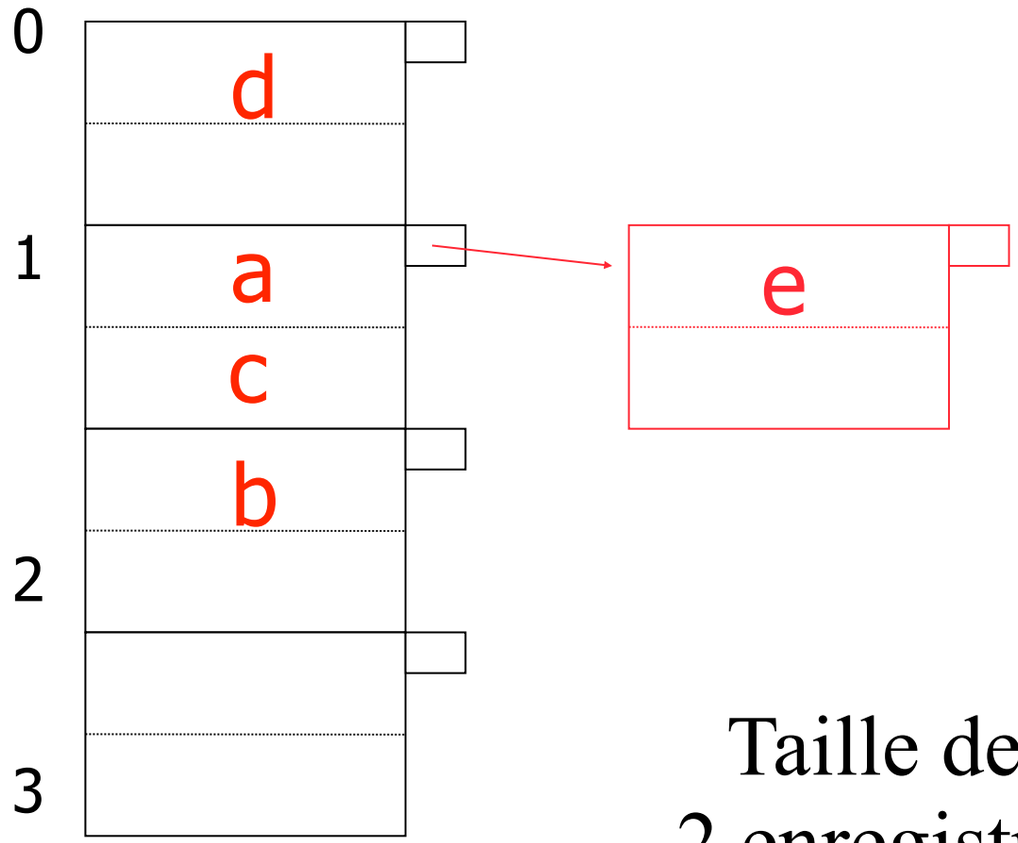
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



Taille des bacs
2 enregistrements

Suppressions

Supprimer l'enregistrement de clé de recherche *K*.

- Calcul de $h(K) = i$.
- Localisation dans le bac *i* de ou des enregistrements indexés par une clé *K*.
- Si possible, réorganiser les enregistrements restant dans le bloc (espace libre à la fin du bloc).
- Si possible, déplacer les enregistrements restants qui sont stockés dans la chaine de dépassement pour libérer des blocs

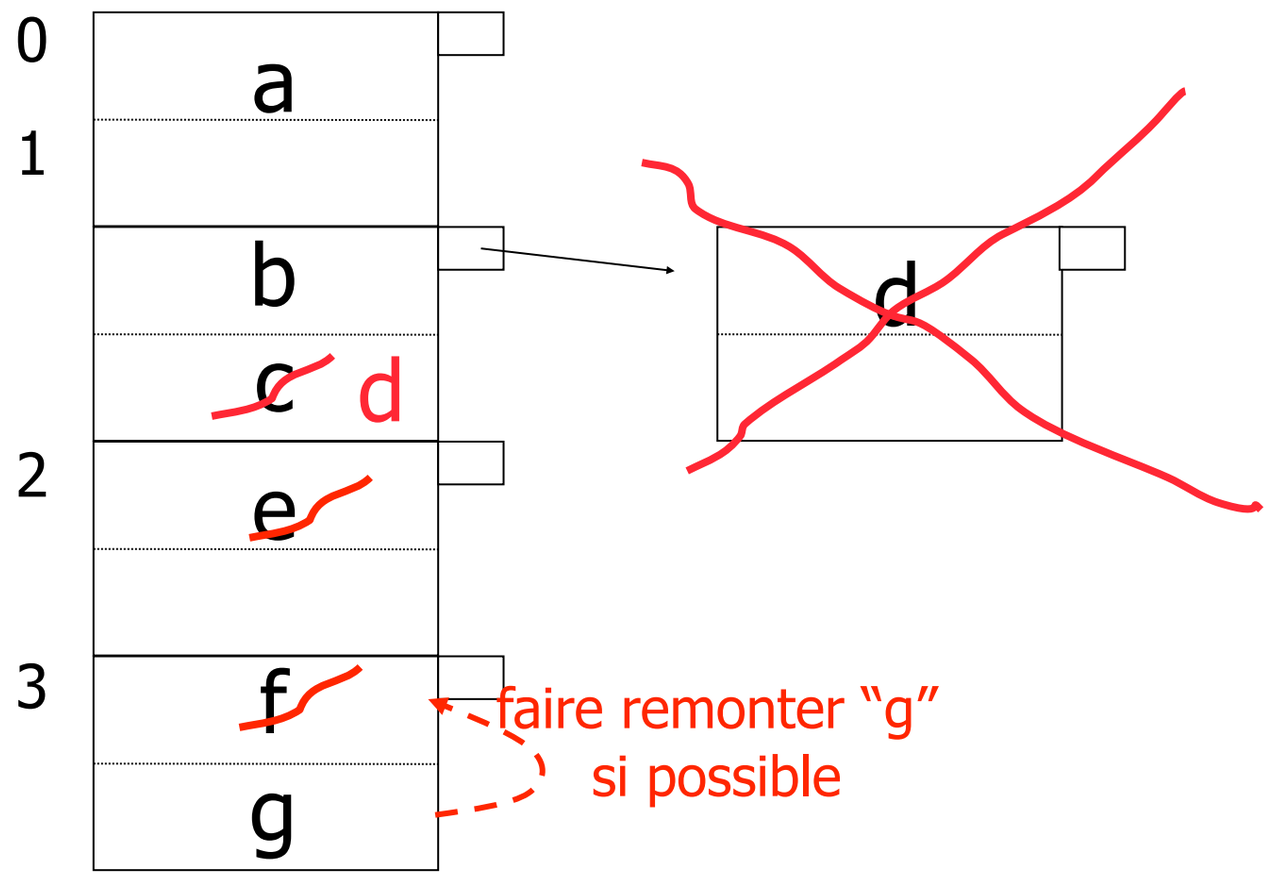
Suppressions

Supprimer :

e

f

c



Complexité des requêtes

Recherche enregistrement(s) de clé *K*.

- Calcul de *h*(*K*) = *i*. (Attention à la complexité du calcul de h).
- Localisation de(s) enregistrement(s) de clé *K* dans le bac *i*, et dans la chaine de dépassement.
- En l'absence de dépassement, seulement un bloc I/O nécessaire , i.e. complexité de O(1) : c'est à dire borné indépendamment du nombre d'enregistrements.

Hash table vs B-Tree(+)

- Accès à un enregistrement particulier généralement plus efficace que les B-Trees.
- Par contre, les tables de hachage ne sont pas efficaces pour les requêtes reposant sur des données triées (<,>, etc.). Au contraire des B-Trees +

Utilisation d'espace

- Pour limiter la taille des chaines de dépassement, il faut un espace d'utilisation compris entre 50% et 80%.
- $$space\ utilization\,(b) = \frac{\#\,\text{keys in bucket } b}{\#\,\text{keys that fit in } b}$$
- Si space utilization < 50%: espace gaspillé.
- SI space utilization > 80%: chaines de dépassement deviennent significatives.
- L'espace utilisé dépend de la fonction de hachage et de la capacité des bacs.

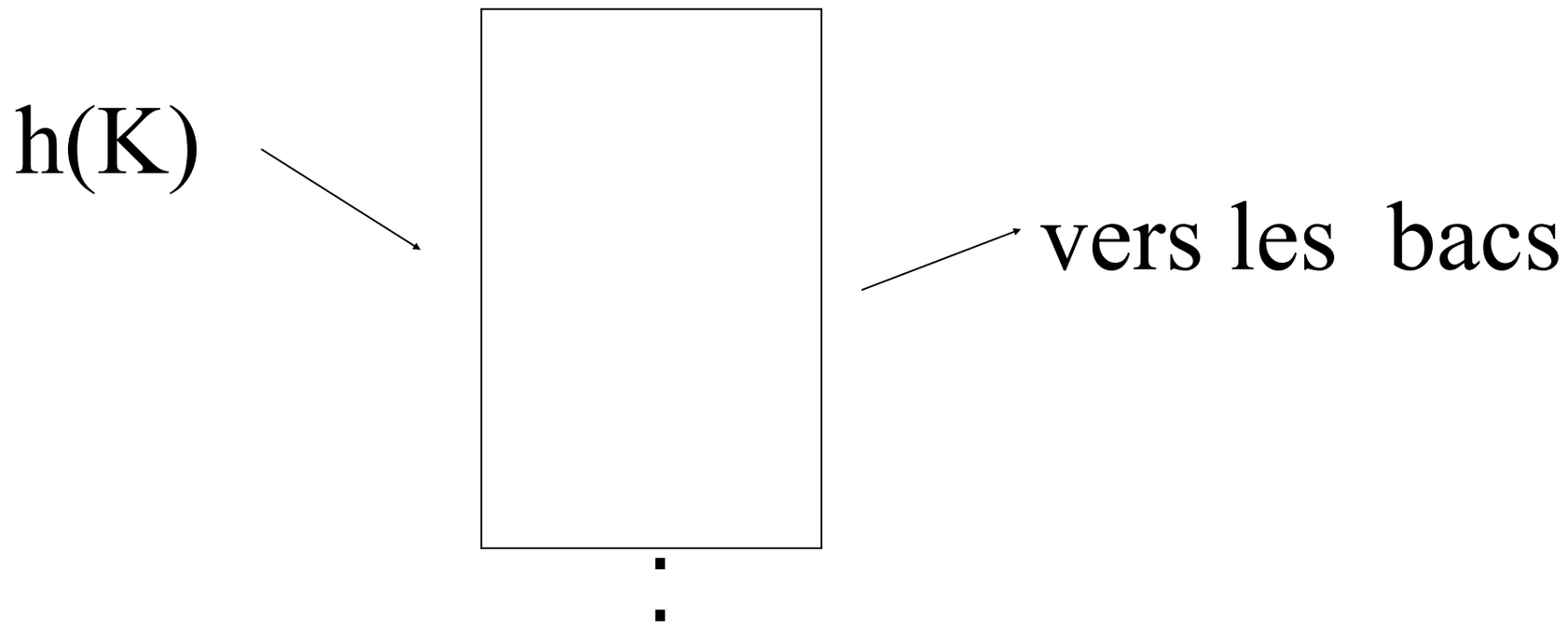
Jusqu'à présent, nous avons seulement abordé les *tables de hachage statiques* où le nombre de bacs ne change jamais.

- Quand le nombre d'enregistrements grandit, on ne peut pas garantir le maintient de l'espace utilisé entre 50% et 80%.

- *Les tables de hachage dynamiques* adaptent *B* en fonction du nombre d'enregistrements stockés.
- But : avoir approximativement 1 bloc par bac.
- Deux méthodes dynamiques:
  - Extensible Hashing,
  - Linear Hashing.

Extensible Hash Tables

•Ajouter un niveau d'indirection pour les bacs, un *repertoire* contenant les pointeurs sur les blocs. 1 répertoire pour chaque valeur de la fonction de hachage.

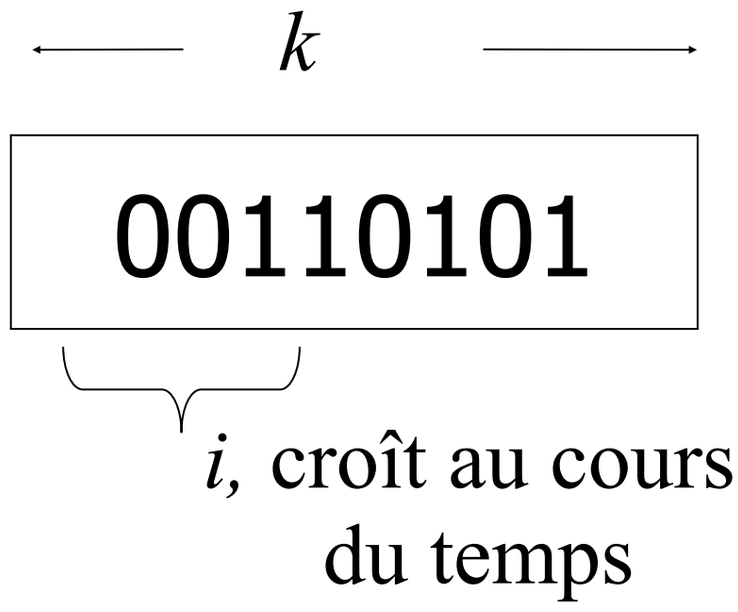


La taille du répertoire double à chaque «croissance» de la table de hachage.

Extensible Hash Tables

• Plusieurs bacs peuvent partager un bloc de données s'ils ne contiennent pas trop d'enregistrements.

- La fonction de hachage calcule des séquences de *k bits*, mais les numéros des bacs utilisent seulement les *i* premiers bits. *i* est le *niveau* de la table de hachage.



$$\text{size of directory} = 2^i,$$

$$\text{initially } i = 0$$

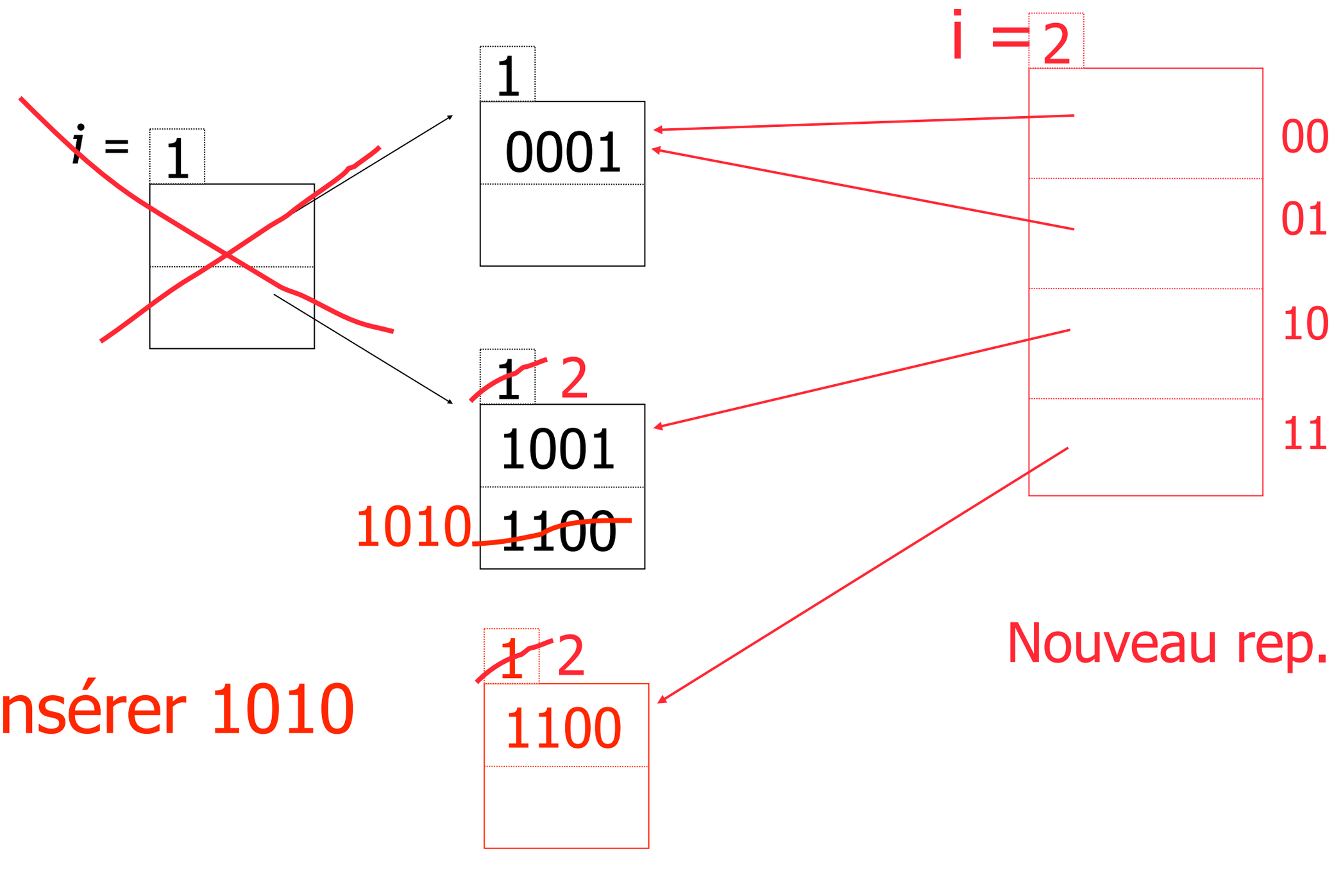
Extensible Hash Tables

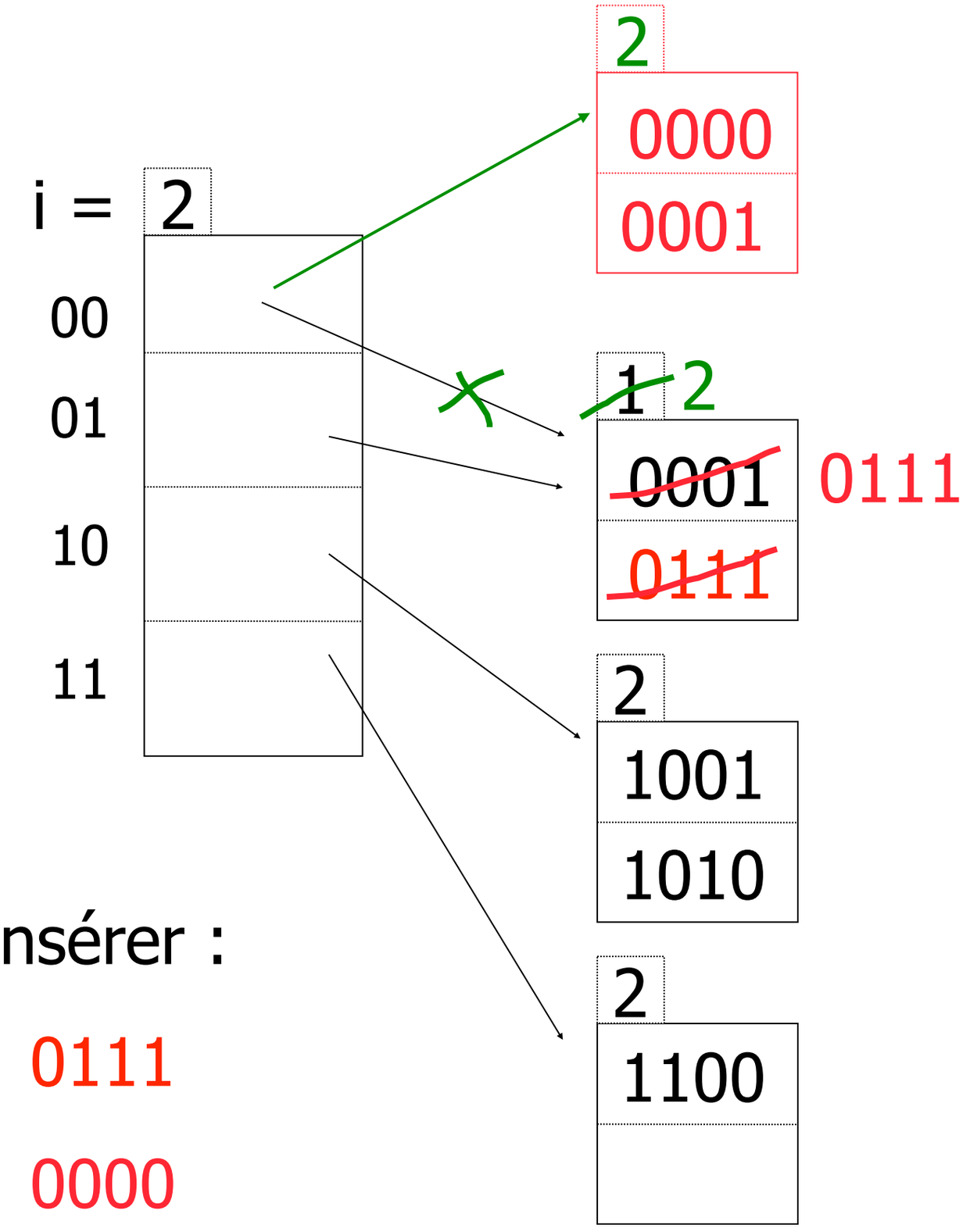
- Insertion de l'enregistrement de clé *K*.
- Calcul de *h*(*K*) et prendre les *i* premiers bits. *Niveau global* *i* fait partie de la structure de données.
- Retrouver l'entrée correspondante dans le répertoire.
- Suivre le pointeur menant au bloc *b*. *b* au *niveau local* *j* <= *i*.
- Si *b* possède suffisamment de place, y insérer l'enregistrement.
- Sinon, diviser *b* en 2 blocs.

Extensible Hash Tables

- Si $j < i$, distribuer les enregistrements dans b en se basant sur le (j+1)ème bit de $h(K)$: si 0, ancien bloc b, si 1 nouveau bloc b'.
- Incrémenter le niveau local de b et b'.
- Ajuster le pointeur dans le répertoire qui pointait sur b, il doit maintenant pointer sur b'.
- Si $j = i$, incrémenter i. Doubler la taille du répertoire et dupliquer toutes les entrées. Puis faire comme dans le cas où $j < i$.

Exemple

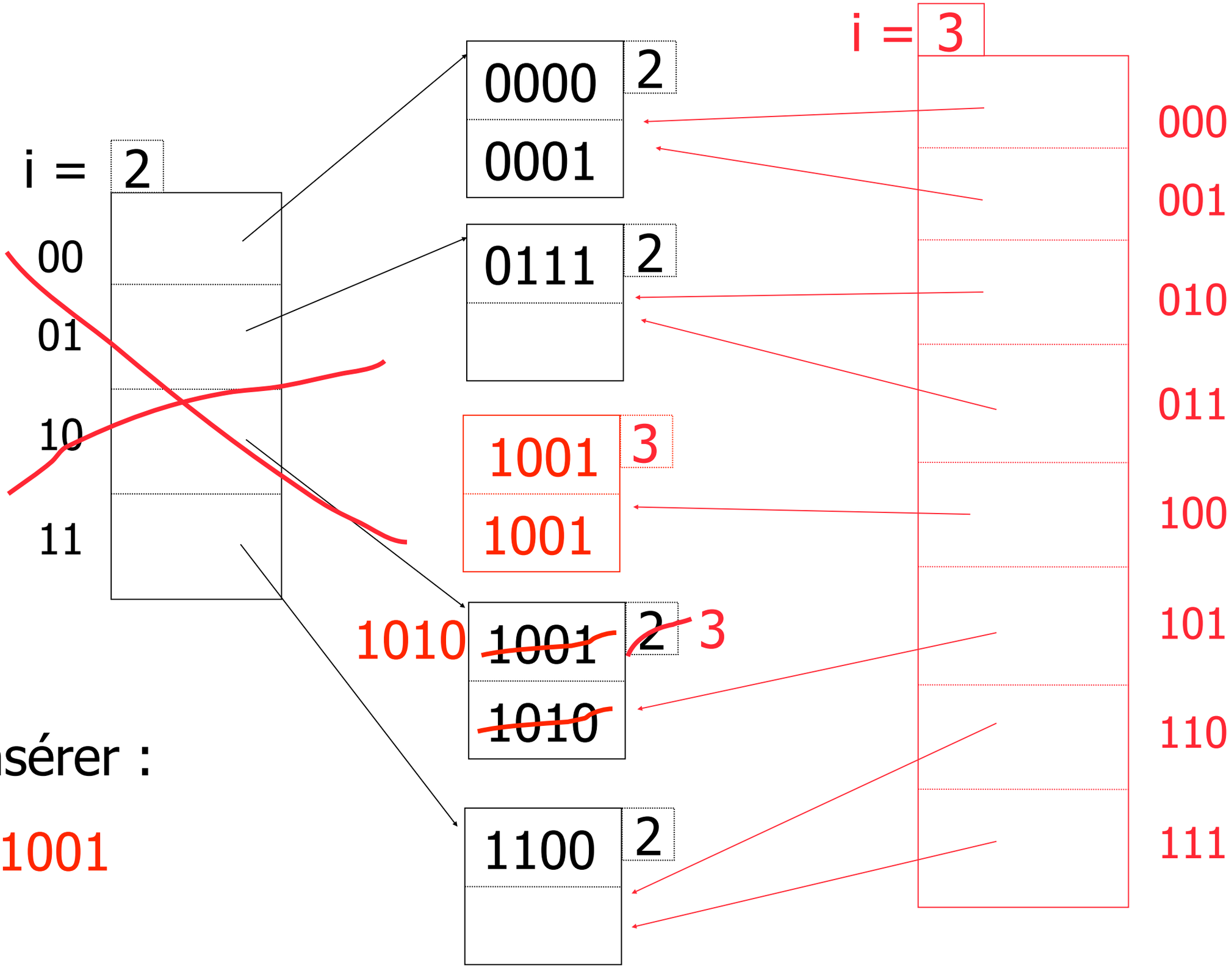

i = 1
1
0001
1
1001
1010
1100
i = 2
00
01
10
11
1
2
1100
Insérer 1010
Nouveau rep.



Insérer :

0111

0000



Insérer :

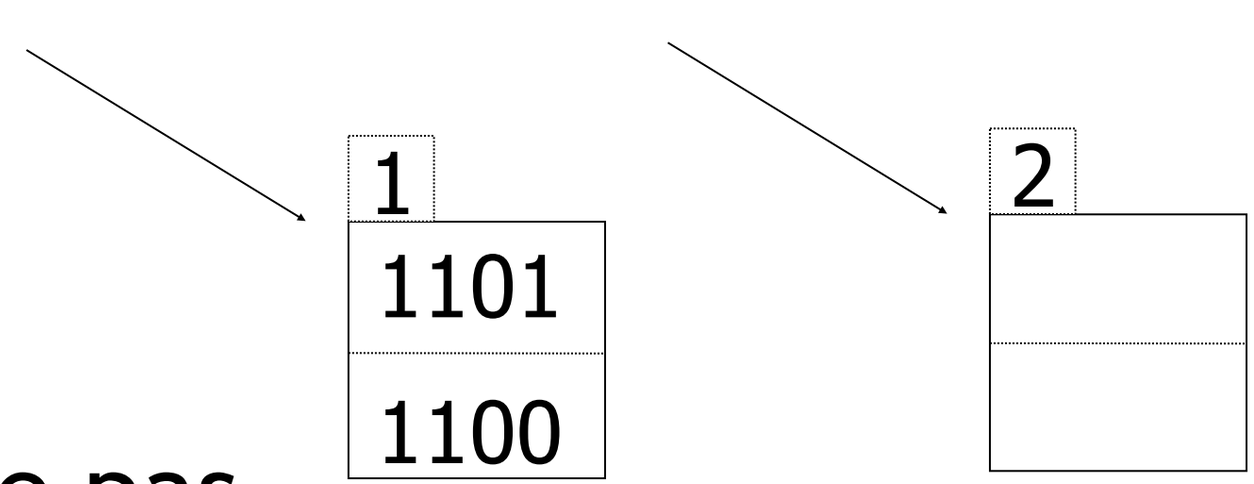
1001

Overflow

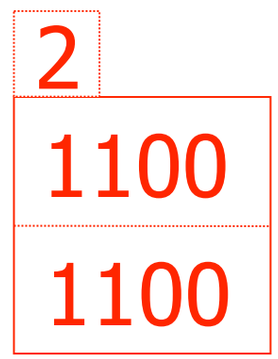
- On peut encore avoir besoin de chaines de dépassement en présence d'un nombre trop important de duplication.

Si nous divisons :

insérer 1100

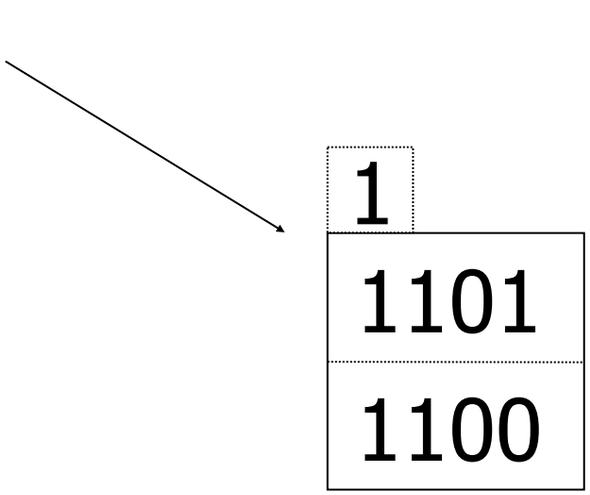


- La division n'aide pas si toutes les entrées se retrouvent dans un seul des deux blocs!

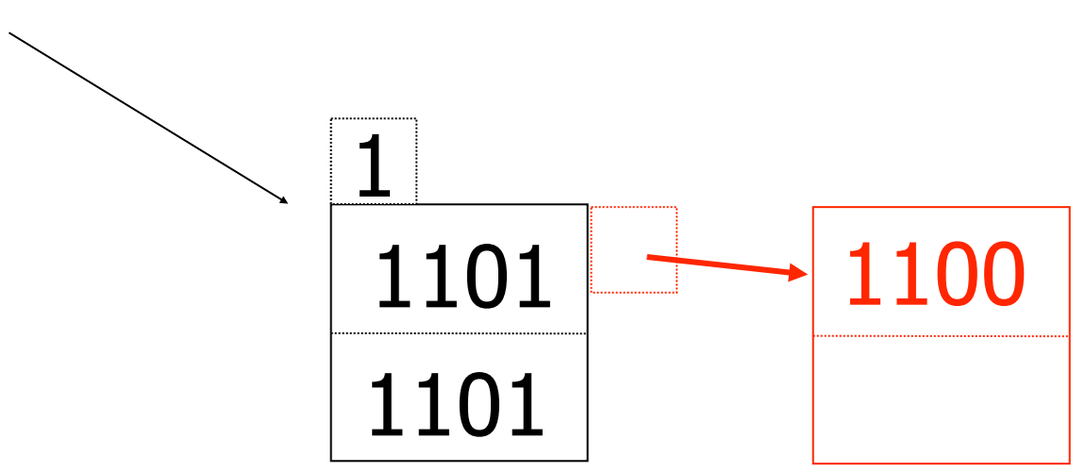


Overflow

insérer 1100



Ajout d'un bloc d'overflow :



Suppression

Suppriment enregistrement de clé *K*.

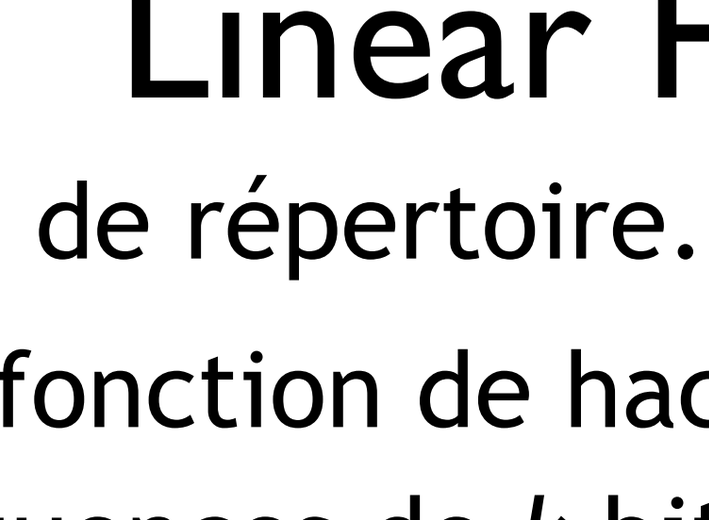
- En utilisant le répertoire, localiser le bloc *b* correspondant et y supprimer l'enregistrement.
- Si possible, fusionner le bloc *b* avec le bloc «ami» *b'* and ajuster les pointeurs de *b* et *b'* dans le répertoire.
- Si possible, réduire de moitié le répertoire.
  → inverse de la procédure d'insertion.

Discussion

- Gestion du nombre croissant de bacs sans gaspiller trop d’espace.
- Cela suppose que le répertoire tient en mémoire.
- Jamais d’accès à plus d’un bloc de données (si pas d’overflow) pour une requête.
- Doubler le répertoire est une opération très coûteuse qui interrompt les autres opérations et peut nécessité un stockage sur disque.

Linear Hash Tables

- Pas de répertoire.
- La fonction de hachage calcule des séquences de *k* bits. Prend seulement les *i* derniers et les interprète comme le bac de numéro *m*.



- *n* : *numéro du dernier bac*, le premier est 0.

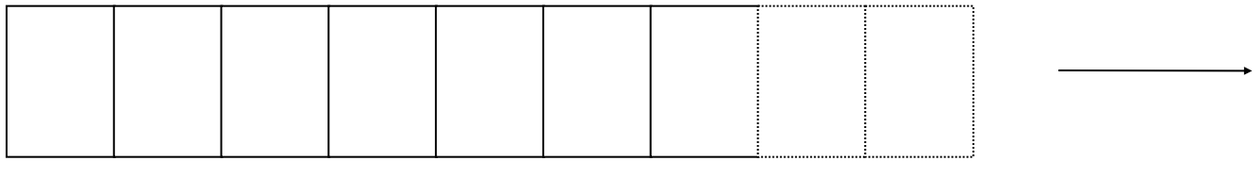
Insertion

- Si *m* <= *n*, stocker dans le bac *m*. Sinon, stockage dans le bac de numéro $m - 2^{i-1}$
- Si overflows, ajout d'un bloc d'overflow.
- Si *space utilization* devient trop important, ajouter un bac à la fin et incrémenter n de 1.

$$space\ utilization = \frac{r}{(n+1) \times c}, \text{where } r = \text{total number of records}$$

$$\text{and } c = \text{bucket capacity (number records)}$$

- → croissance linéaire

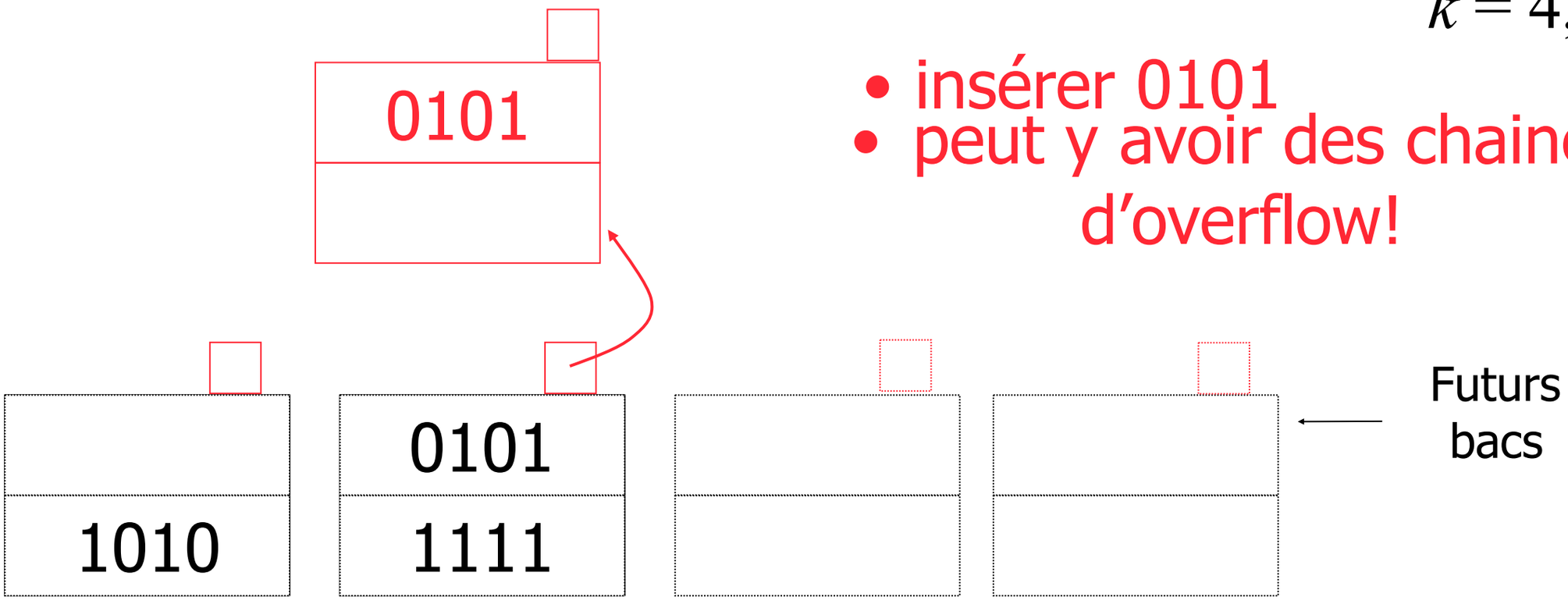


Insertion

- Le bac ajouté n'est pas généralement dans le rayon des clés où un overflow est apparu.
- Quand $n > 2^i$, incrementer i de 1.
- i est le nombre de tours (*rounds)* doublant la taille de la table.
- Pas besoin de déplacer des entrées.

Exemple

$k = 4,\ i = 2$

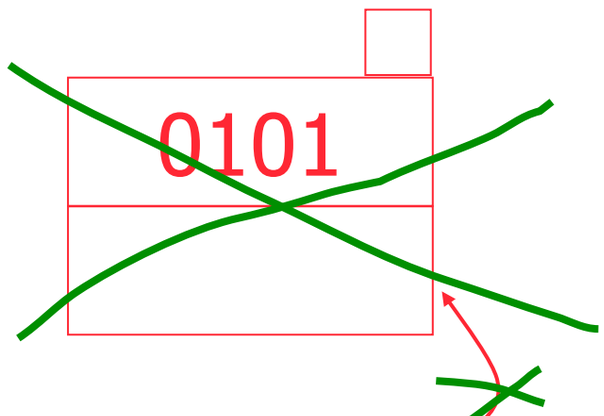


- insérer 0101
- peut y avoir des chaines d'overflow!

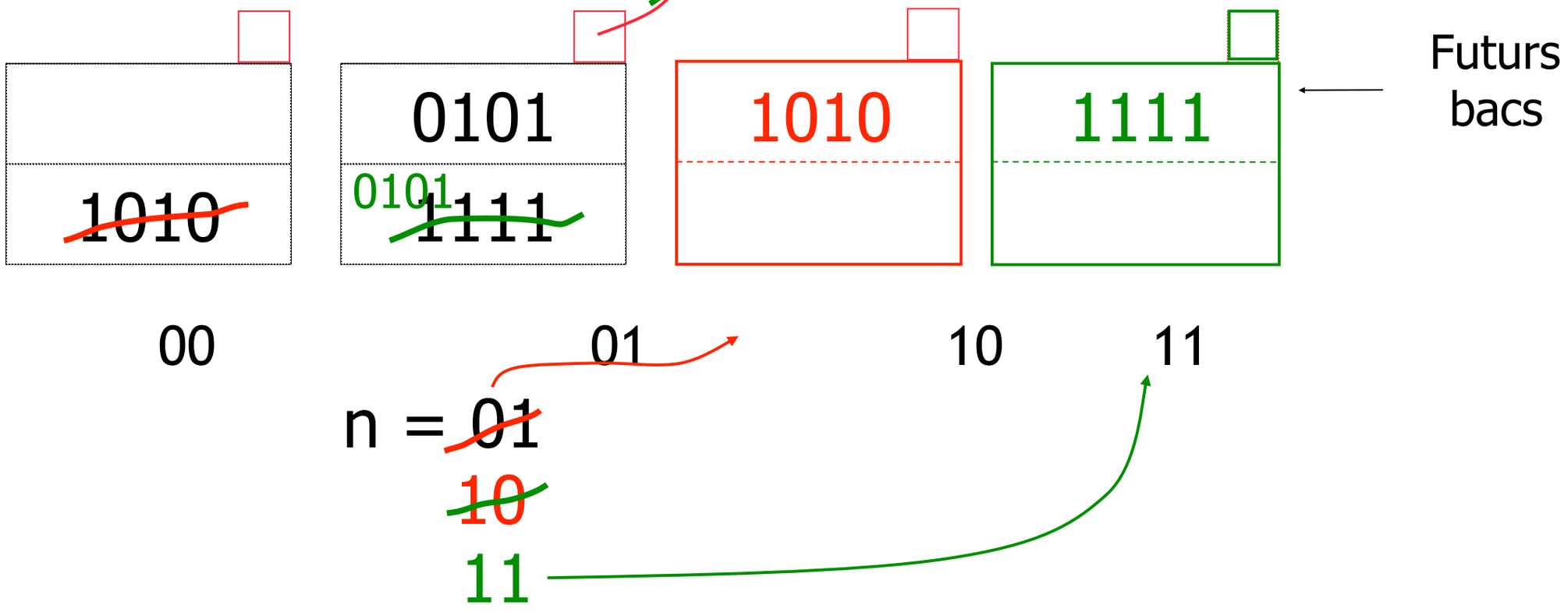
Si h(k)[i] ≤ n, alors
regarder bac h(k)[i]
sinon, regarder bac h(k)[i] – 2^{i-1}

Exemple

$k = 4,\ i = 2$



- insérer 0101



Exemple

$k = 4$

i = ~~2~~ 3

	~~0101~~	1010	1111		0101
	~~0101~~				0101

0 00 0 01 0 10 0 11 100 101 . . .

~~100~~ ~~101~~ 110 111

n = ~~11~~

~~100~~

101

Discussion

- Pas de gaspillage d'espace.
- Pas de répertoire, i.e. pas d'indirection d'accès et pas d'opération coûteuse pour doubler la taille.
- Besoin de chaines de dépassement, même si pas de duplications parmi les *i* derniers bits des valeurs de hachage.

Résumé

- Tables de hachage: meilleures pour les requêtes d'égalité, ne supportent pas les autres recherches.
- Hachage statique peut provoquer de longues chaines d'overflow.
- Extendible Hashing évite les overflows en divisant les bacs pleins quand un nouvel enregistrement est ajouté. (les overflows sont encore possibles.)
  - Répertoire double périodiquement.
  - Peut devenir très grand avec des données distribuées asymétriquement (I/O en plus si ça ne tient pas en mémoire)

- Linear Hashing évite de faire un répertoire en divisant les bacs et en utilisant des overflows.
  - Overflow chaine ne sont pas trop longues
  - Duplications gérées facilement.
  - Space utilization peut être plus faible que les Extendible Hashing, puisque les divisions ne se concentrent pas que sur les zones denses.

Le hachage intéressant quand :

- Le jeu de données est figé,
- Les recherches se font par clé (égalité)

Ce sont des situations relativement courantes : le hachage n'occupe alors pas de place. Sinon, une structure de type B-Arbre(+) est plus pertinente.

Index Bitmap

Comment indexer une table sur un attribut qui ne prend qu'un **petit nombre de valeurs** ?

- Avec un B-Arbre : pas très bon car chaque valeur est peu sélective.
- Avec un hachage : pas très bon car il y a beaucoup de collisions.

Or situation **fréquente**, notamment dans les entrepôts de données.

Exemple : codification de films

rang	titre	genre	...
1	Vertigo	Suspense	...
2	Brazil	Science-Fiction	...
3	Twin Peaks	Fantastique	...
4	Underground	Drame	...
5	Easy Rider	Drame	...
6	Psychose	Drame	...
7	Greystoke	Aventures	...
8	Shining	Fantastique	...
...	...	...	...

Principes de l'index Bitmap

Soit un attribut **A** prenant ***n*** valeurs possibles [v1,v2,...,vn].

- On crée *n* tableaux de bits, un pour chaque valeur ***vi***.
- Ce tableau contient 1 bit pour chaque enregistrement **e**.
- Le bit d'un enregistrement **e** est à **1** si **e.A = vi**, à **0 sinon**.

Exemple

	1	2	3	4	5	6	7	8
Drame	0	0	0	1	1	1	0	0
Science-Fiction	0	1	0	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0

	9	10	11	12	13	14	15	16
Drame	0	0	0	0	0	0	1	0
Science-Fiction	0	1	1	0	0	0	0	0
Comédie	1	0	0	1	0	0	0	1

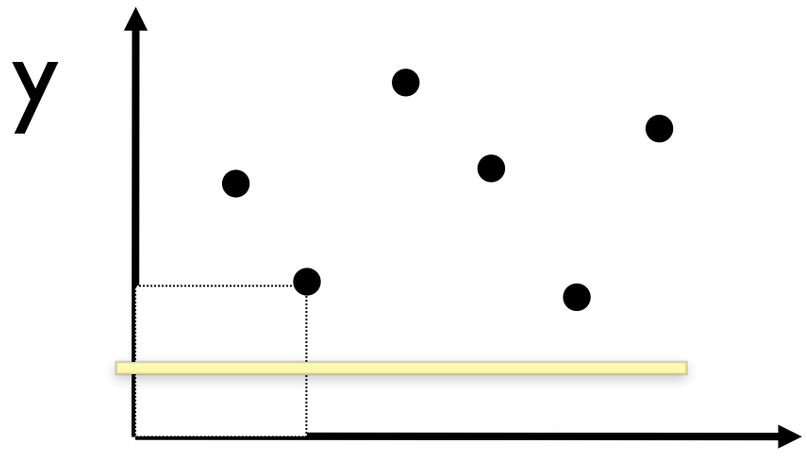
Recherche

- Soit une requête comme : **SELECT** * **FROM** FILM **WHERE** Genre = ‘Drame’
  - On prend le tableau pour la valeur **Drame**.
    - On garde toutes les cellules à 1.
    - On accède aux enregistrements par l’adresse.
- => Très efficace si **n**, le nombre de valeurs est petit.

Autre exemple

- Select count(*) from film where genre in (‘Drame’, ‘Comedie’).

Index multidimensionnel



Données géographiques

Données:

<X1,Y1,Attributs>

<X2,Y2,Attributs>

⋮

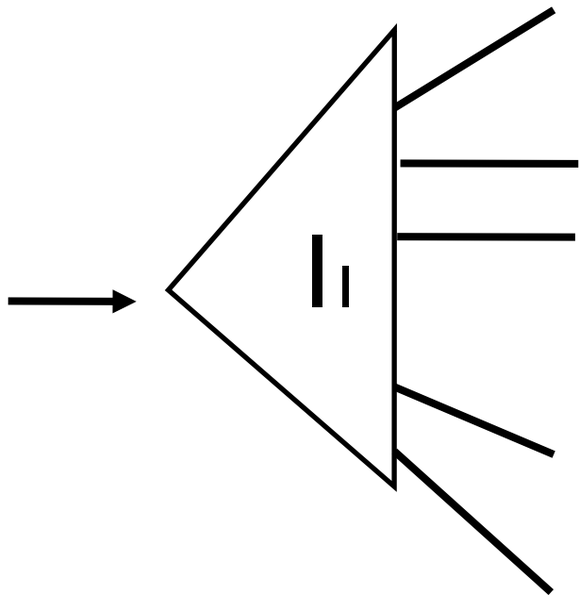
Requête :
Qui est dans un rayon de 5 km autour de <Xi,Yi>?
Quel est le point le proche de <Xi,Yi>?

Index MD

- Multi-key Index
- Grid Files
- Partitioned Hash Indexes
- R Trees
- etc.

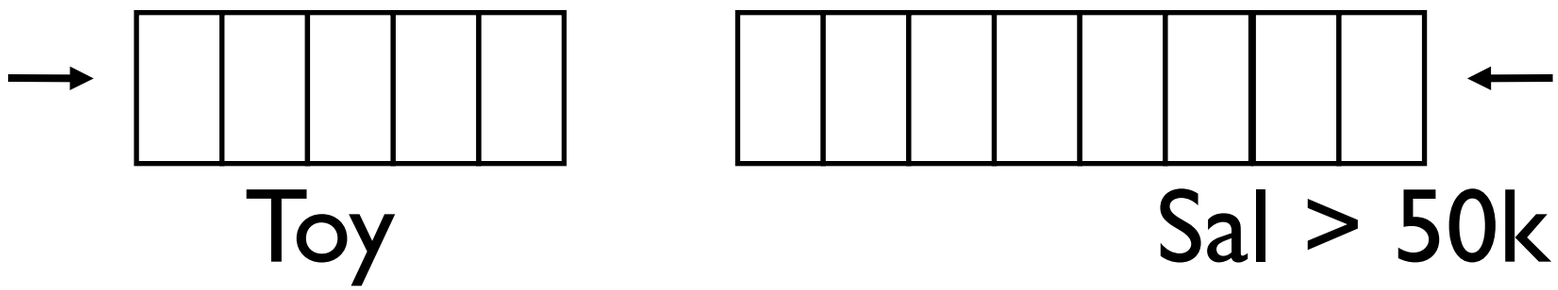
Multi-Key Index : Strategie I

- Utiliser, par exemple Dept.
- Prendre tous les enregistrements Dept = “Toy” et regarder les salaires.

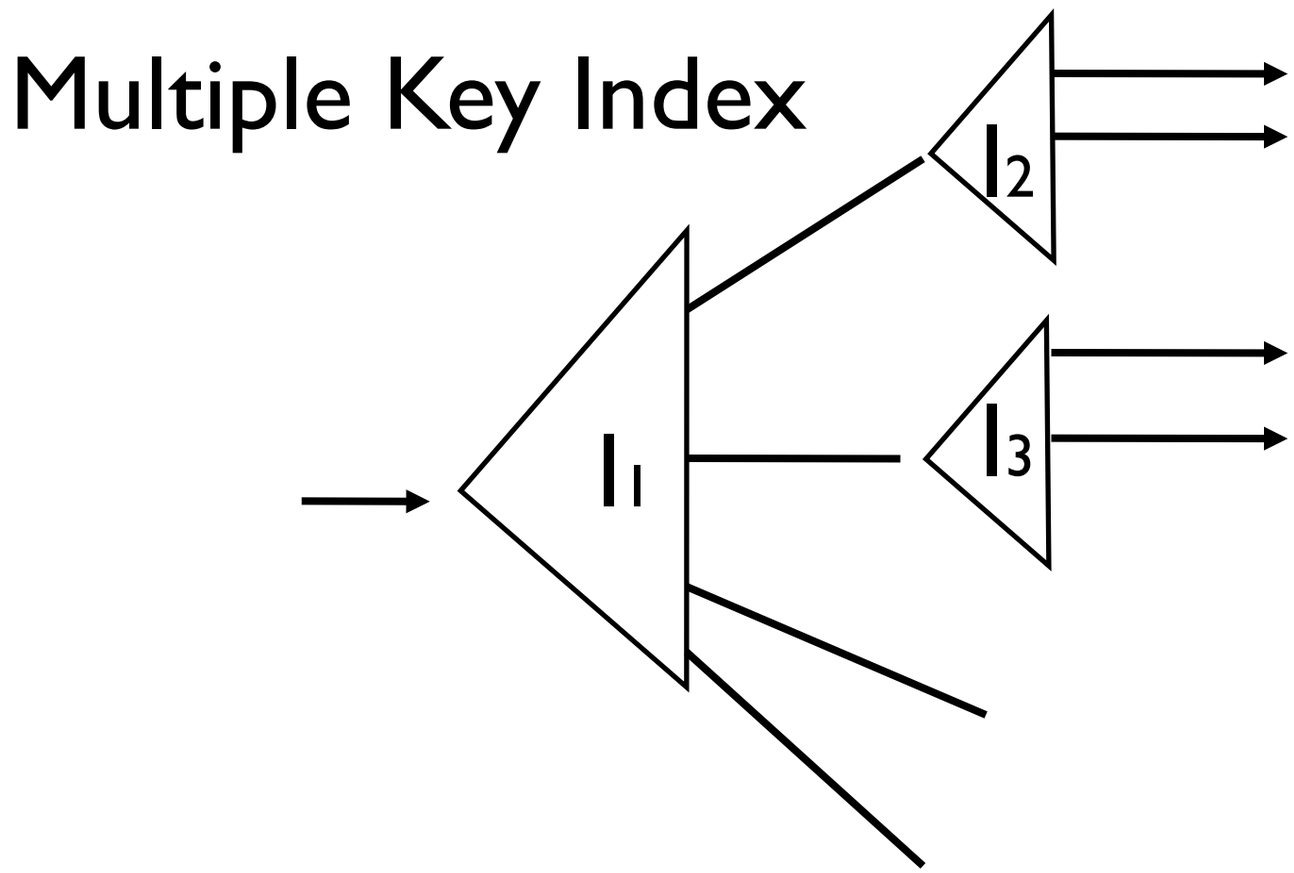


Multi-Key Index : Strategie II

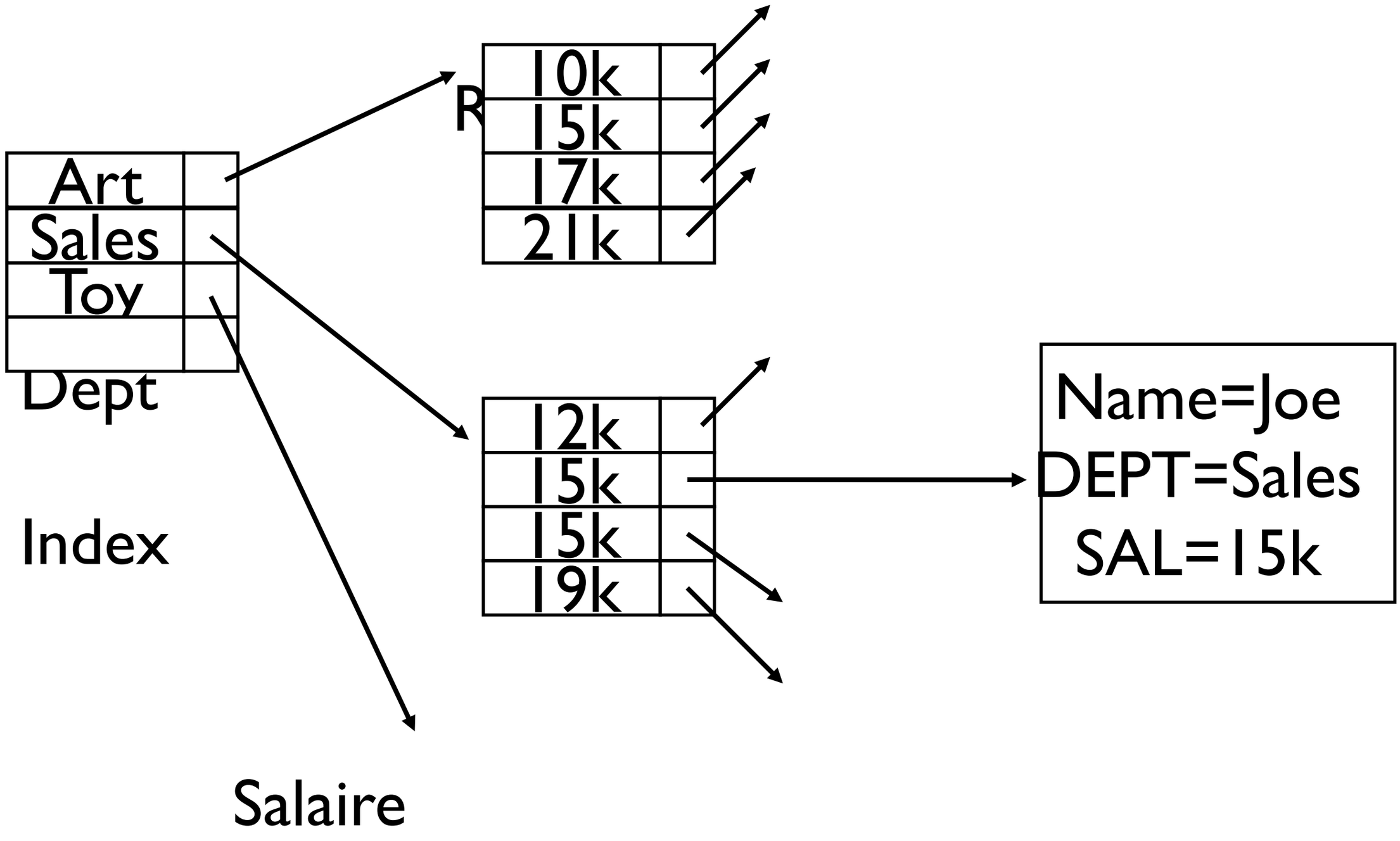
- Utiliser 2 Index et manipuler des pointeurs.



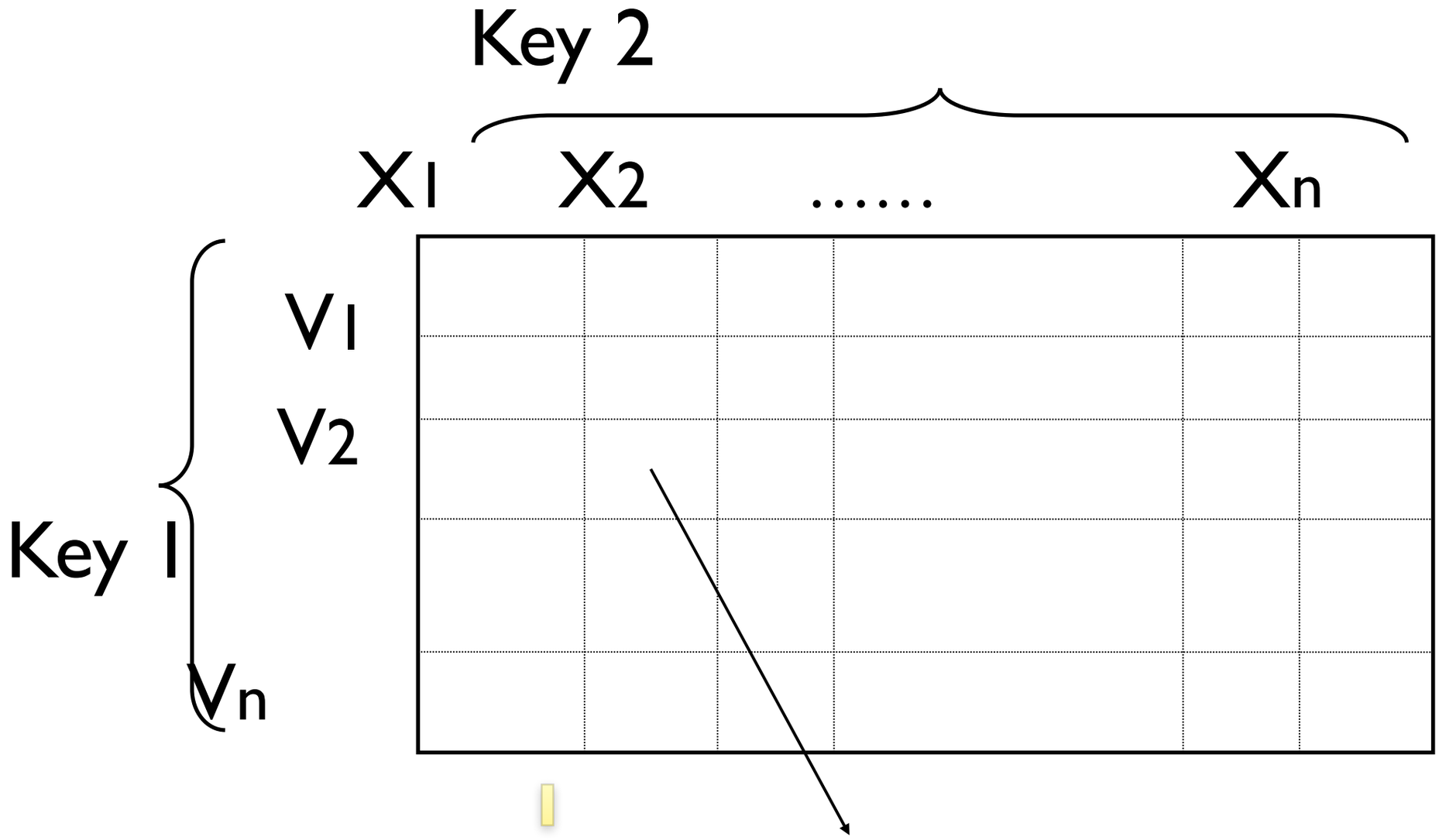
Multi-key Index : Stratégie III



Example


R
10k
15k
17k
21k
Art
Sales
Toy
Dept
Index
12k
15k
15k
19k
Name=Joe
DEPT=Sales
SAL=15k
Salaire

Grid-File Index



Enregistrements clé key1=V_2, key2=X_2

On peut rapidement retrouver les enregistrements :

key 1 = V_i ∧ Key 2 = X_j

key 1 = V_i

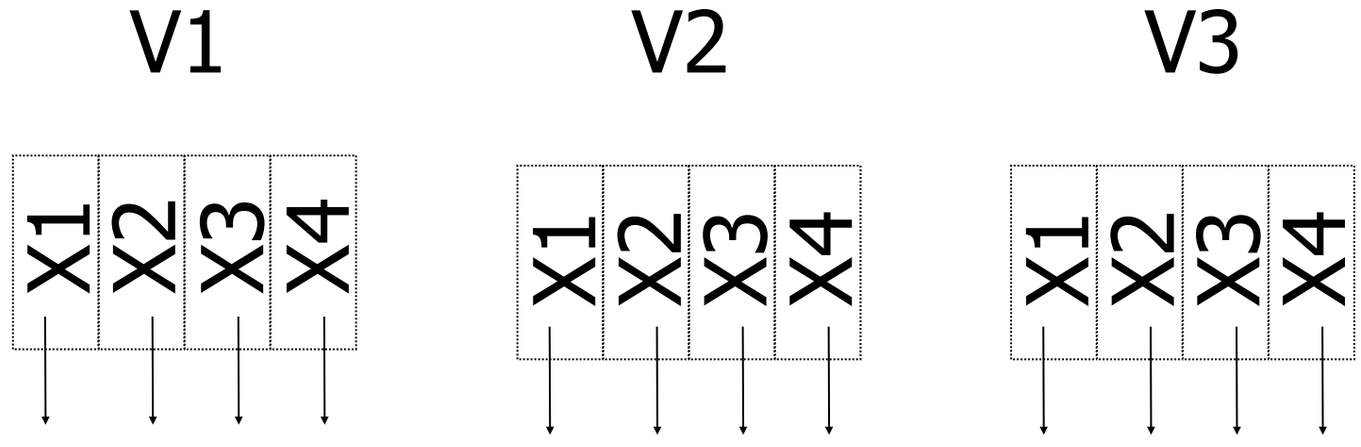
Et aussi avec des requêtes plus complexes

E.g., key 1 ≥ V_i ∧ key 2 < X_j

Problèmes

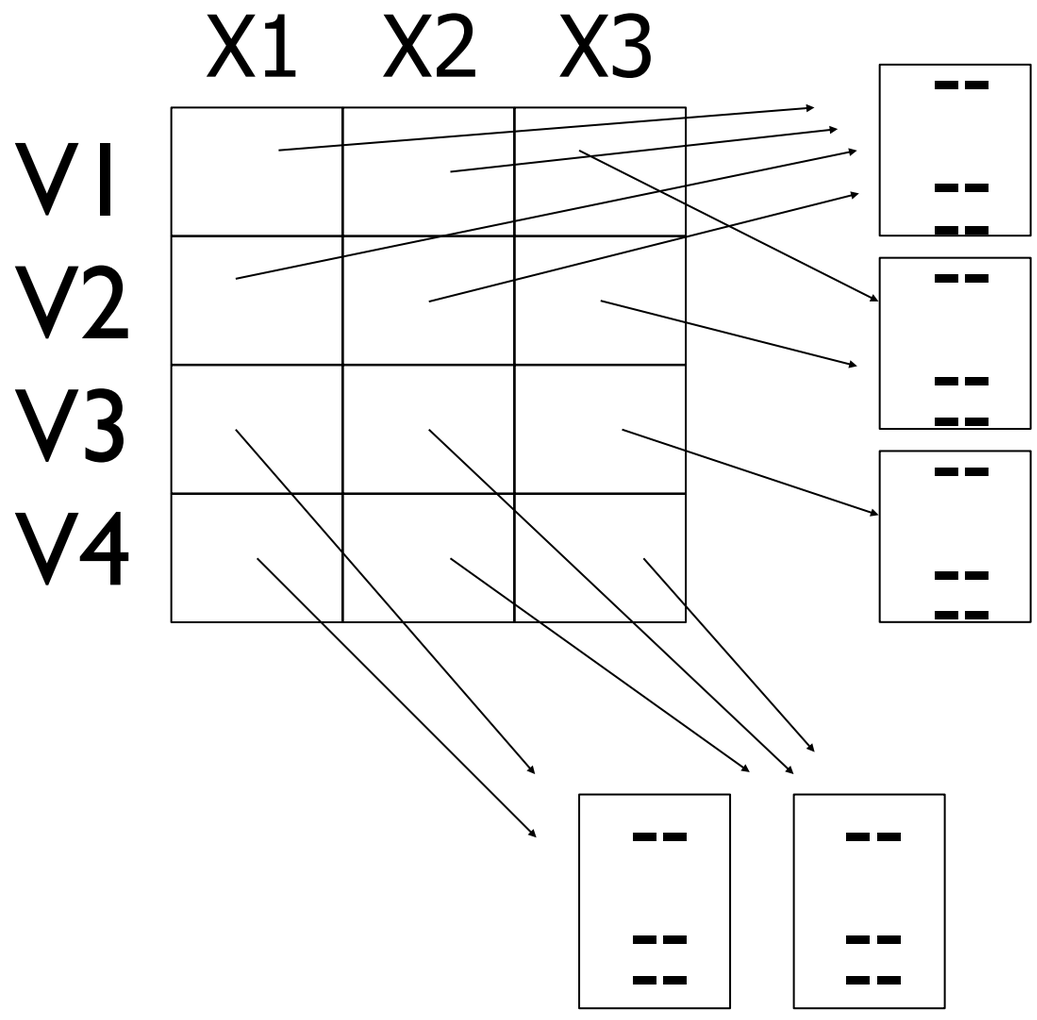
- Comment stocker ces fichiers sur le disque ?
- Besoin de régularité si veut calculer la position de <Vi, Xj>.

Comme les tableau

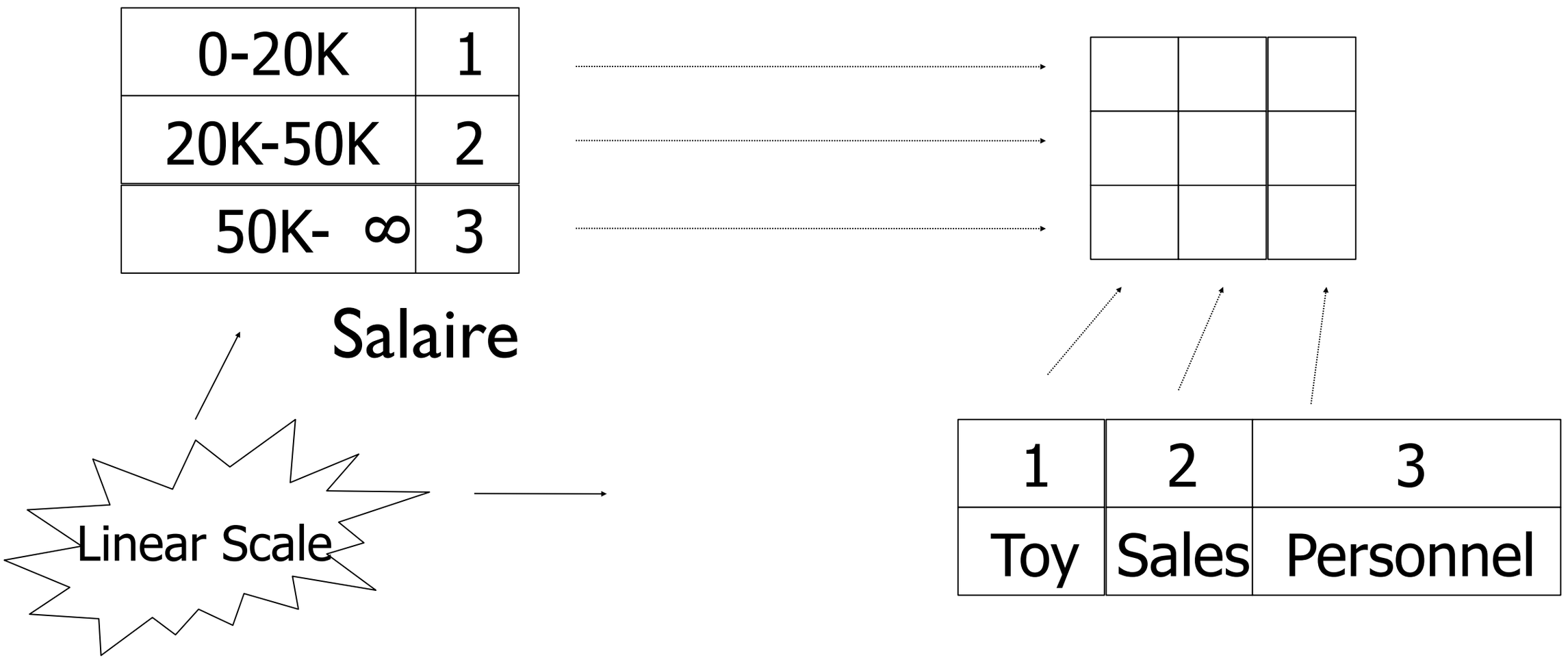


Solutions

- La grille ne contient que des pointeurs sur des buckets.



Partitionner les domaines



Grid files

⊕ Efficace pour des recherche multi-keys

⊖ Gestion de l'espace

⊖ Besoin de partitionnement qui distribue bien les clés.

Les index en SQL

- Les index sont spécifiés au moment où la table est créée.
- En général, un index est créé par défaut sur une clé primaire
- Il est possible d'ajouter ou supprimer un index d'une table : create index, drop index.
- Il est parfois possible de préciser quelle structure utilisée (B-Tree, hash, index bitmap, R-Tree)

L'implémentation des opérations de l'algèbre relationnelle

- Critère de performance : temps de calcul
- Paramètres utilisés pour une relation *r* :
  - nr : nombre de tuples de *r*
  - *V(X,r) : nombres de valeurs distinctes pour l'attribut X dans r*
  - *Sr : taille des tuples*

r1 ∪ r2

- Sans index : O(nr1 + nr2)
- O((nr1 + nr2) log(nr1+nr2)) (tri)
- Avec index sur r1 : n'ajouter à r1 que les tuples de r2 non présents dans r1 :
  - O(nr1 + nr2 log(nr1))

r1-r2

- trier les deux relations et les parcourir ensuite séquentiellement, éliminant de r1 les tuples de r2 : O(nr1 log(nr1) + nr2 log(nr2)) ;
- utiliser un index existant pour r2 : considérer chaque tuple de r1, le chercher dans r2 en utilisant l'index : O(nr1 log(nr2)).

Projection

- $\pi_X r1$:
  - En général : parcours de r1 avec extraction des composantes nécessaires de chaque tuple : O(nr1).
  - Si l'élimination des occurences multiples est nécessaire, alors il faut trier la relation r1 : O(nr1 log(nr1))

Jointure

- Supposons que r2 possède un index pour les attributs communs X.
- On parcourt r1 et pour chaque tuple de r1 (et sa valeur pour X), on va chercher dans r2 les tuples qui ont la même valeur pour X.
  - Complexité : O(nr1(1 + log(nr2))), pour autant que nr2/V (X,r2) soit borné

pour les attributs communs X.