

BDay-MI

Bases de données avancées

Cours de Cristina Sirangelo

IRIF, Université Paris Diderot

Assuré en 2021-2022 par Amélie Gheerbrant

amelie@irif.fr

Organisation physique des données et indexation

Sources (quelques slides empruntés et réadaptés) :

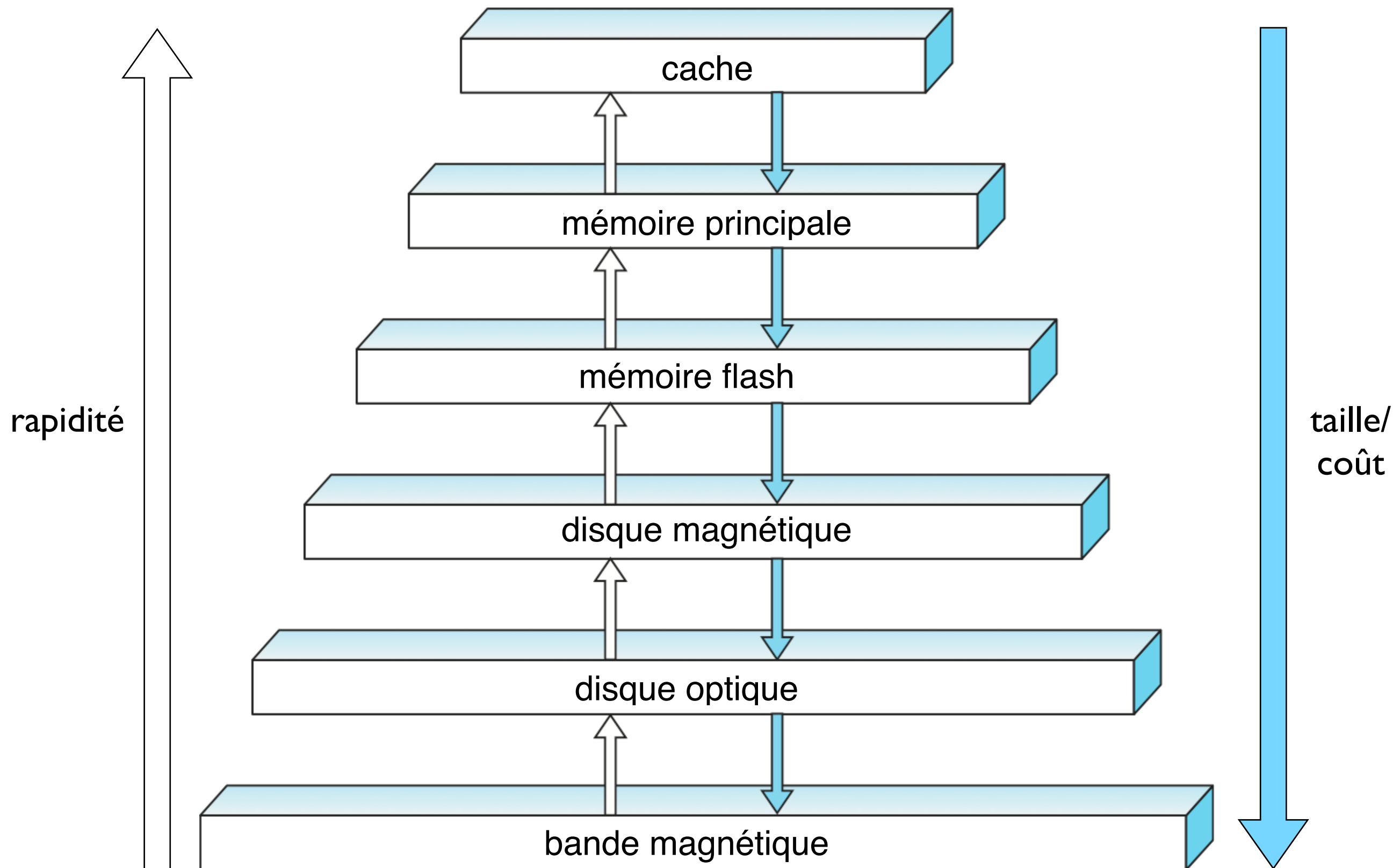
- cours *Database systems principles* - H.G. Molina, Stanford Univ.
- cours *Bases de données* - S.Abiteboul, INRIA, ENS Cachan
- slides du livre *Database systems concepts* -
A. Silberschatz, Yale U. & H. Korth, Lehigh U. & S.Sudarshan, IIT Bombay

Stockage des données

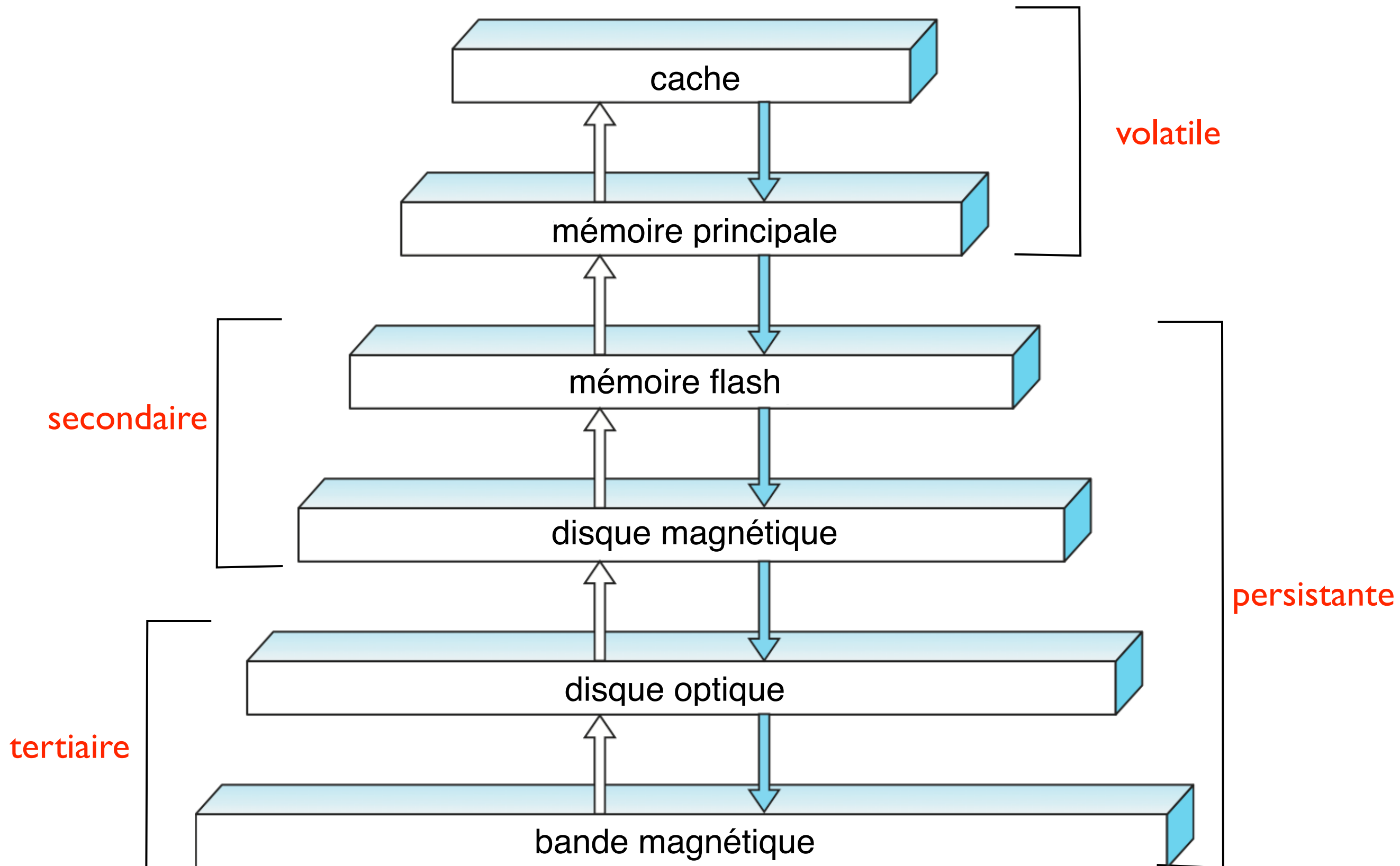
Stockage des données

- Le SGBD stocke les données dans une mémoire **persistante**
 - ▶ **mémoire persistante**: les contenus sont préservés même en l'absence d'alimentation.
 - ▶ **mémoire volatile** : perd son contenu quand l'alimentation s'arrête
- Le SGBD interagit avec plusieurs types de mémoires pour gérer/traiter les données

Hiérarchie de mémoire



Hiérarchie de mémoire



Hiérarchie de mémoire

- **Cache** – la plus rapide et la plus couteuse; volatile. Maintient les infos plus récemment utilisées
- **Mémoire principale (RAM):**
 - ▶ volatile
 - ▶ accès rapide (10s à 100s de nanosecondes)
 - ▶ en générale trop petite (ou trop couteuse) pour stocker la BD entière
 - cela est en train de changer - on commence à parler de mémoires principales suffisamment grandes...
 - la BD doit de toute façon être également stockée en mémoire persistante
 - ▶ taille jusqu'à quelques Gigabytes aujourd'hui
 - la taille augmente (ainsi que le coût se réduit) d'un facteur 2 tous les deux/trois ans à peu près

Hiérarchie de mémoire

- **Mémoire flash** (*Electrically erasable programmable read-only memory*)
 - ▶ persistante
 - ▶ lectures presque aussi rapides qu'en mémoire principale
 - ▶ écritures plus lentes (quelques microsecondes)
 - ▶ largement utilisée dans les systèmes embarqués tels que caméras numériques, téléphones, clefs USB
 - ▶ remplace parfois aussi le disque

Hiérarchie de mémoire

- **Disque magnétique**

- ▶ persistant, disque à écriture/ lecture magnétique
 - des pannes de disque peuvent détruire les données mais c'est rare
- ▶ support principal de stockage long-terme : stocke typiquement la BD entière
- ▶ accès beaucoup plus lent que la mémoire principale (millisecondes)
- ▶ accès direct – il est possible de lire/écrire du disque dans n'importe quel ordre (à la différence de la bande magnétique)
- ▶ taille jusqu'à quelques terabytes typiquement
 - augmente d'un facteur de 2 à 3 tous les deux ans environ

Hiérarchie de mémoire

- **Disque optique**

- ▶ persistant, lectures/écritures optiques au laser
- ▶ CD-ROM (640 MB) et DVD (4.7 to 17 GB) les plus populaires
- ▶ disques au rayons bleu: 27 GB to 54 GB
- ▶ les types *Write-one, read-many* (WORM) sont utilisés pour archivage (CD-R, DVD-R, DVD+R)
- ▶ Des versions à plusieurs écritures sont aussi disponibles (CD-RW, DVD-RW, DVD+RW, et DVD-RAM)
- ▶ lectures et écritures plus lente qu'avec le disque magnétique
- ▶ systèmes "Juke-box" pour le stockage de grande quantités de données

Hiérarchie de mémoire

- **Bande magnétique**

- ▶ persistant, utilisé principalement pour *backup* et pour archivage
- ▶ accès séquentiel – beaucoup plus lent que le disque
- ▶ très grande taille par unité (bande de 40 à 300 GB disponibles)
- ▶ beaucoup moins chère que le disque
- ▶ “Juke-boxes” de bandes pour stocker des quantités massives de données
 - de centaines de terabytes (1 terabyte = 10^{12} bytes)
à plusieurs petabytes (1 petabyte = 10^{15} bytes)

Stockages des données et types de mémoires

- Une BD est stockée en général en **mémoire secondaire** (disques magnétiques, flash)
 - ▶ raisons :
 - persistance, stockage de grande quantité à coût raisonnable
 - accès direct et en temps réel (par rapport à la mémoire tertiaire)
 - ▶ aujourd'hui BD très volumineuse \Rightarrow un seul disque ne suffit pas de plus : besoin de fiabilité des données
 - RAID: *Redundant Arrays of Independent Disks*
 - à plus grande échelle : BD distribuée sur le réseau (*Cloud*)
- Les données sont déplacées du disque en **mémoire principale** pour l'accès, puis réécrites sur disque pour le stockage
- Les données qui ne sont plus modifiées et rarement lues sont transférées en **mémoire tertiaire** (disques optiques, bandes magnétiques) pour l'archivage

Organisation des données sur disque

Modèle physique des données

- **Champ** : une séquence de *bytes* sur disque représentant une valeur d'un type élémentaire (entier, réel, chaîne de caractères, etc.)

Un entier de 2 *bytes*

55

Une chaîne de caractères de longueur fixe 10

s	m	i	t	h					
---	---	---	---	---	--	--	--	--	--

- **Enregistrement** (représentation physique d'un n-uplet) : une séquence de champs

55	s	m	i	t	h					02
----	---	---	---	---	---	--	--	--	--	----

- ▶ le nombre, ordre et types des champs définissent le **format** d'un enregistrement (analogue physique du schéma)
- ▶ un enregistrement contient en général des champs additionnels nécessaires à sa gestion :
 - pointeurs à d'autres zones du disque,
 - méta-données : la longueur de l'enregistrement, un bit "effacé", etc.

Modèle physique des données

- **Fichier de données** (représentation physique d'une table) : une collection d'enregistrements du même format

55	s m i t h	02
32	d u p o n t	01
11	b l a c k	03
24	a r m a n d	02
81	c h a r e n t i n	02

⋮

- ▶ les enregistrements ne sont pas nécessairement tous contigus en mémoire
 - différentes organisations possibles (cf. plus loin)

Blocs et *buffer*

- L'accès au disque se fait en général par blocs
 - ▶ **bloc** :
 - une zone contiguë d'un certain nombre de bytes (taille typique : plusieurs Kilobytes)
 - l'unité de transfert du disque à la mémoire centrale
- **buffer** : plusieurs blocs peuvent être maintenus en mémoire centrale dans un *buffer* pour optimiser l'accès au disque



Blocs et *buffer*

- Quand une lecture/écriture à une adresse du disque est demandée par une application, le système :
 - ▶ vérifie si le bloc contenant cette adresse est dans le *buffer*
 - ▶ s'il n'y est pas, lit du disque et charge dans le *buffer* le bloc entier qui contient la donnée demandée
 - ▶ lit/écrit la donnée demandée dans le *buffer*
- Un bloc est re-transféré du *buffer* au disque uniquement quand il est nécessaire de libérer de la place dans le *buffer*



- Problème : stratégie de remplacement nécessaire pour le *buffer*
 - ▶ typiquement LRU (*least recently used*)
 - ▶ d'autres possibles

Organisation physique d'un fichier de données

- Pour faciliter le transfert, un fichier de données est organisé physiquement en respectant la structure des blocs :
 - ▶ un fichier de données est en général beaucoup plus grand qu'un bloc
 - ▶ un enregistrement est en général beaucoup plus petit qu'un bloc

⇒ fichier : ensemble de blocs,
dans un bloc : un nombre (en général entier) d'enregistrements

- En général : un bloc contient les données d'un seul fichier de données

Organisation physique d'un fichier de données

fichier

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000

bloc 0

32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000

bloc 1

76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

bloc 2

Organisation physique d'un fichier de données

- Des chiffres plus réalistes :
 - ▶ en supposant le format suivant d'un enregistrement :
 - un en-tête : 16 bytes
 - un entier : 4 bytes
 - une chaîne de caractères de longueur 35 : 35 bytes
 - une chaîne de caractères de longueur 10 : 10 bytes
 - ▶ \Rightarrow un enregistrement a une taille de 69 bytes
 - ▶ un bloc de 4K = 4096 bytes contient jusqu'à 59 enregistrements

Organisation physique d'un fichier de données

Organisations des enregistrements dans un bloc

- **Enregistrements de taille fixe** : positionnés en zones contiguës du bloc, jusqu'à ce qu'il n'y ait plus de place dans le bloc

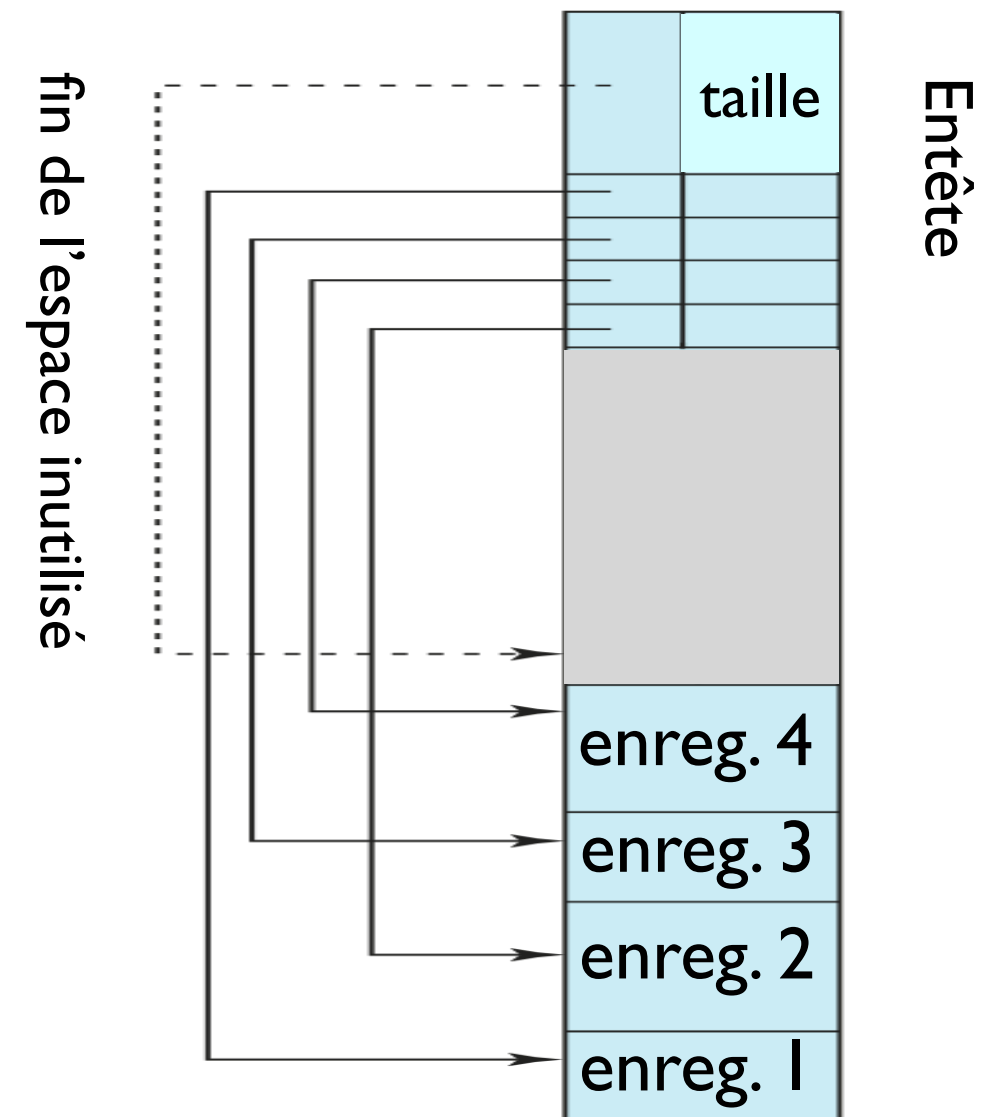
Organisation physique d'un fichier de données

Organisations des enregistrements dans un bloc

- **Enregistrements de taille fixe** : positionnés en zones contiguës du bloc, jusqu'à ce qu'il n'y ait plus de place dans le bloc
- **Enregistrements de taille variable** : **table de "offset"**
le bloc contient un entête qui stocke :
 - ▶ le nombre d'enregistrements dans le bloc
 - ▶ la fin de l'espace inutilisé
 - ▶ un pointeur à chaque enregistrement (avec sa taille)

Remarque : les enregistrements sont compactés à la fin du bloc pour permettre à l'entête de grandir avec l'ajout de nouveaux enregistrements

- **Organisation de champs de taille variable dans un enregistrement** : techniques similaires

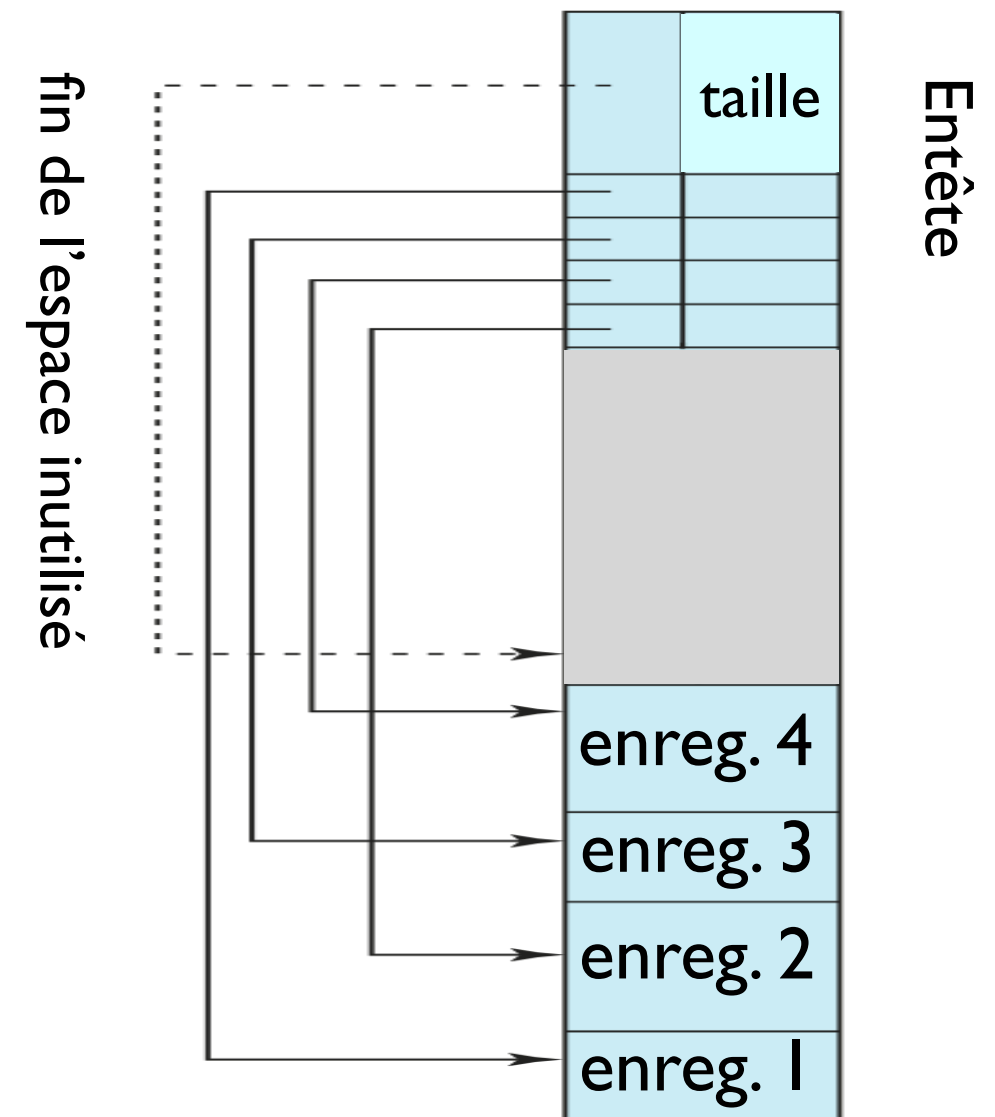


Organisation physique d'un fichier de données

Organisations des enregistrements dans un bloc

Table de *offset*, également avec enregistrements de taille fixée

- Permet une gestion d'**adresses logiques** des enregistrements :
 - ▶ un pointeur externe ne doit pas pointer directement à l'enregistrement, mais à l'entrée pour cet enregistrement dans l'entête de son bloc
 - ▶ les enregistrements peuvent alors être déplacés à l'intérieur du bloc
 - sans modifier les pointeurs externes
 - en modifiant uniquement l'entête du bloc

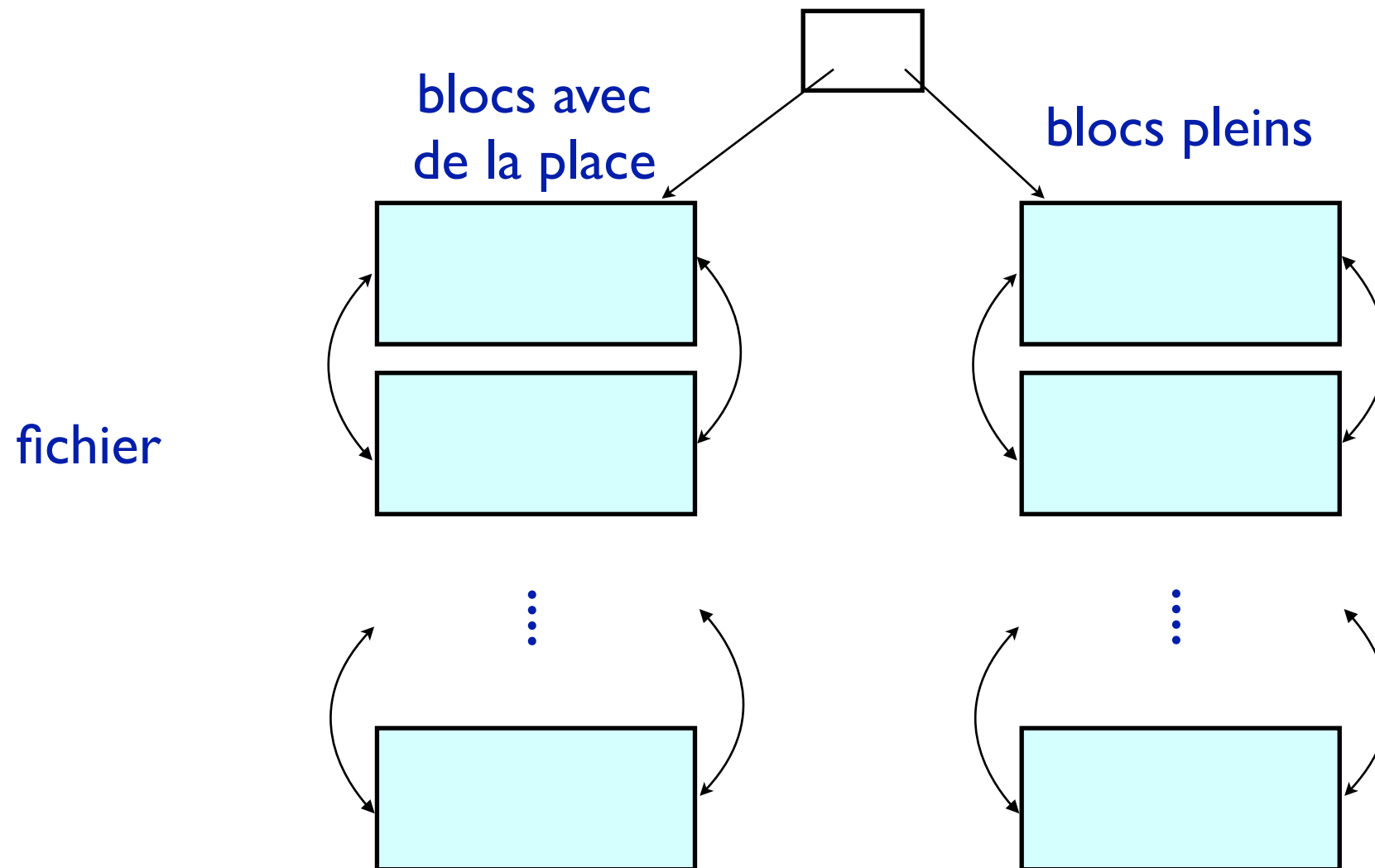


Organisation physique d'un fichier de données

- Plusieurs organisations pour maintenir l'ensemble des blocs d'un fichier
 - ▶ organisation en tas
 - ▶ organisation séquentielle
 - ▶ organisation en “grappe” multi-tables (*multi-table clustering*)
 - ▶ et d'autres...

Organisation **en tas** d'un fichier de données

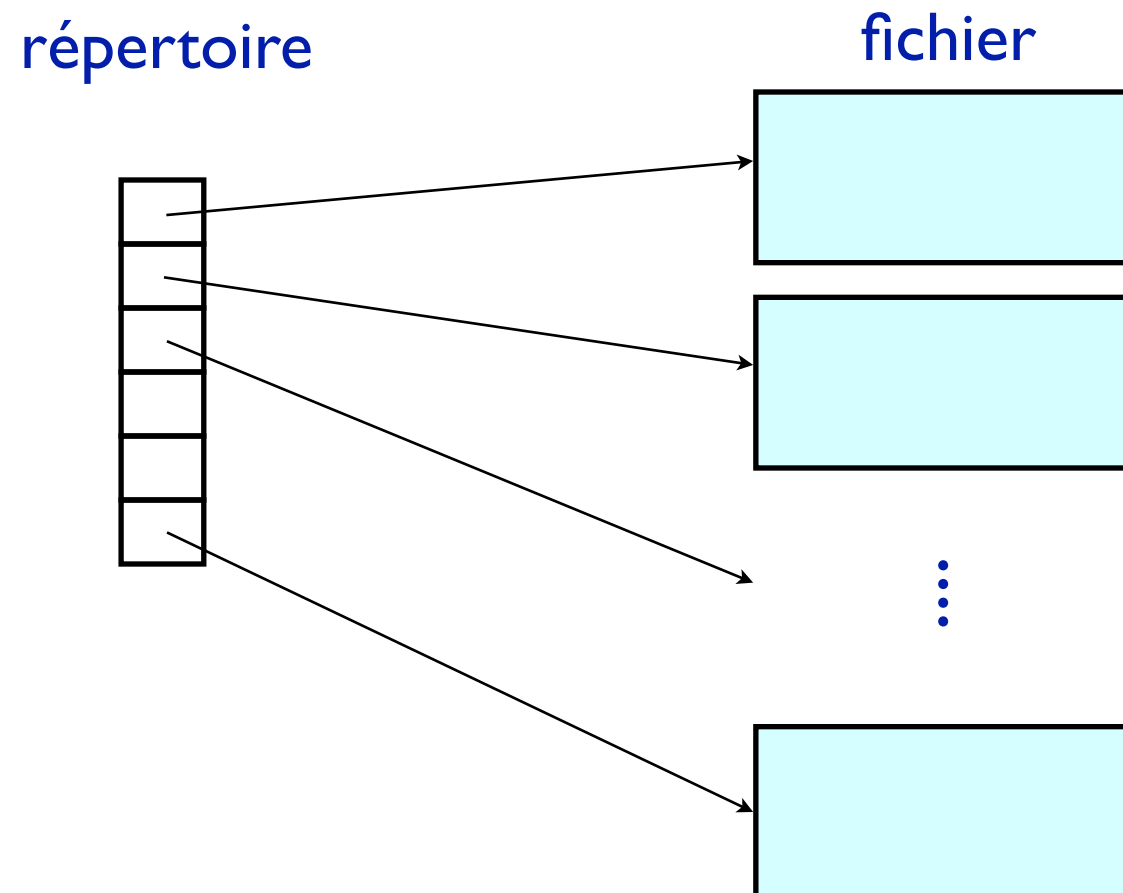
- Les enregistrements sont distribués dans les blocs sans aucun ordre particulier
- Une organisation possible : les blocs sont enchaînés (liste doublement chaînée)



- Inconvénient : à l'insertion d'un nouvel enregistrement il faut parcourir la liste des blocs avec espace disponible

Organisation **en tas** d'un fichier de données

- **Organisation alternative** : garder une liste non-ordonnée de pointeurs aux blocs du fichier (structure appelée *répertoire* du fichier)



- En plus des pointeurs, le répertoire peut stocker des informations sur chaque bloc (un bit pour indiquer s'il y a de la place, la quantité de place disponible etc...)
- À l'insertion d'un nouvel enregistrement seulement le répertoire (beaucoup plus petit) est parcouru

Organisation **séquentielle** d'un fichier de données

- Les enregistrements sont stockés dans un ordre particulier : ils sont triés par valeur croissante d'une clef de recherche

Clef de recherche : un ensemble de champs (par exemple la clef primaire de la table, mais pas nécessairement)

- Aussi appelée organisation ISAM (*indexed-sequential access method*)
- Motivation : pour rendre possible un parcours séquentiel efficace de la table
- Implémentation :
 - ▶ Les enregistrements dans un bloc sont triés et les blocs sont triés entre eux
 - ▶ Une liste triée des pointeurs aux blocs est maintenue grâce à une structure auxiliaire appelée *index* (non-dense)
 - ▶ On supposera initialement que les blocs ont une table de *offset* qui permet de bouger facilement leurs enregistrements

Organisation séquentielle d'un fichier de données

- Exemple. Clef de recherche : premier champ

The diagram illustrates a sequential file organization with a non-dense index. A label 'index non-dense' is shown in a box on the left, with three arrows pointing to the first column of each of the three tables below. Each table has four columns: a key (first column), a name (second column), a field (third column), and a value (fourth column). The tables are separated by horizontal lines.

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000

32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000

76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- Le rôle des index va au delà de cela, et on les étudiera en détail plus loin

Maintenir l'organisation d'un fichier de données

- Les deux structures (tas, séquentielle) doivent être maintenues avec les mises à jour du fichier de données
 - ▶ insertion et suppression d'enregistrements
- Pour l'organisation en tas
 - ▶ mise à jour des listes chaînées ou répertoire, ...
- Pour l'organisation séquentielle
 - ▶ maintien de l'ordre, mise à jour de l'index, ...
- Dans les deux cas : plus ou moins facile selon qu'on puisse aisément déplacer les enregistrements dans les blocs

Maintenir l'organisation **en tas** d'un fichier de données

Insertion d'un nouvel enregistrement

- Chercher un bloc avec suffisamment de place disponible et y insérer l'enregistrement
- Si tous les blocs sont pleins, créer un nouveau bloc avec l'enregistrement et le connecter à la structure du fichier (liste chaînée ou répertoire)

Suppression d'un enregistrement

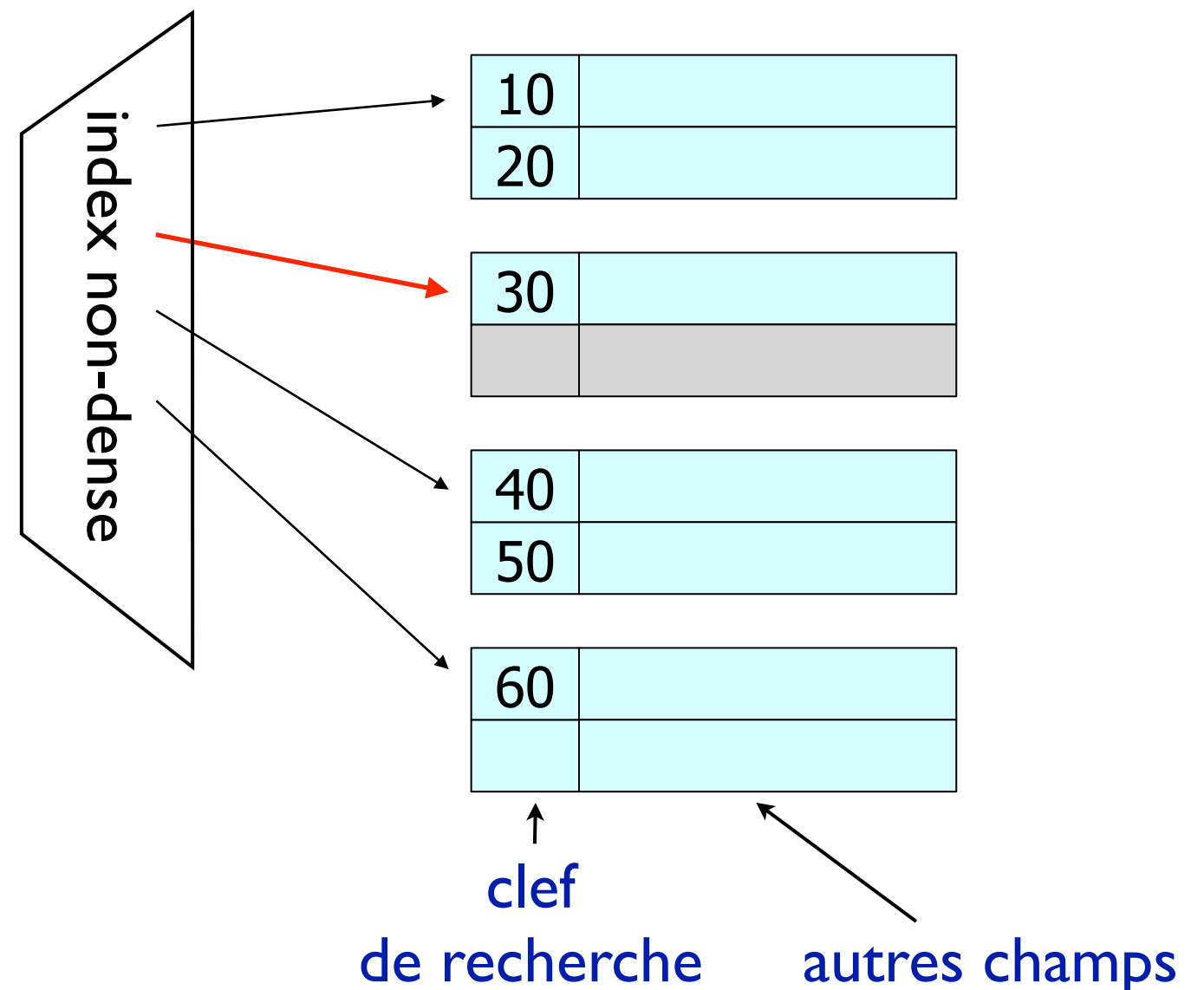
- Si les enregistrements peuvent être bougés dans un bloc (table de *offset*), re-compacter les enregistrements pour libérer la place
- Sinon marquer l'enregistrement comme “effacé” (bit dans l'entête de l'enregistrement)
- En cas d'enchaînement de blocs : mettre à jour la liste des blocs avec de la place
- Si le bloc devient vide l'éliminer et re-compacter la structure (liste ou répertoire)

Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- Trouver le bloc où l'enregistrement doit être inséré (déterminé par sa clef de recherche)
 - ▶ l'index est utilisé à cet effet, voir plus loin
- S'il y a de la place dans le bloc, y insérer le nouvel enregistrement

- Ex. Insérer un enregistrement avec clef de recherche = 34

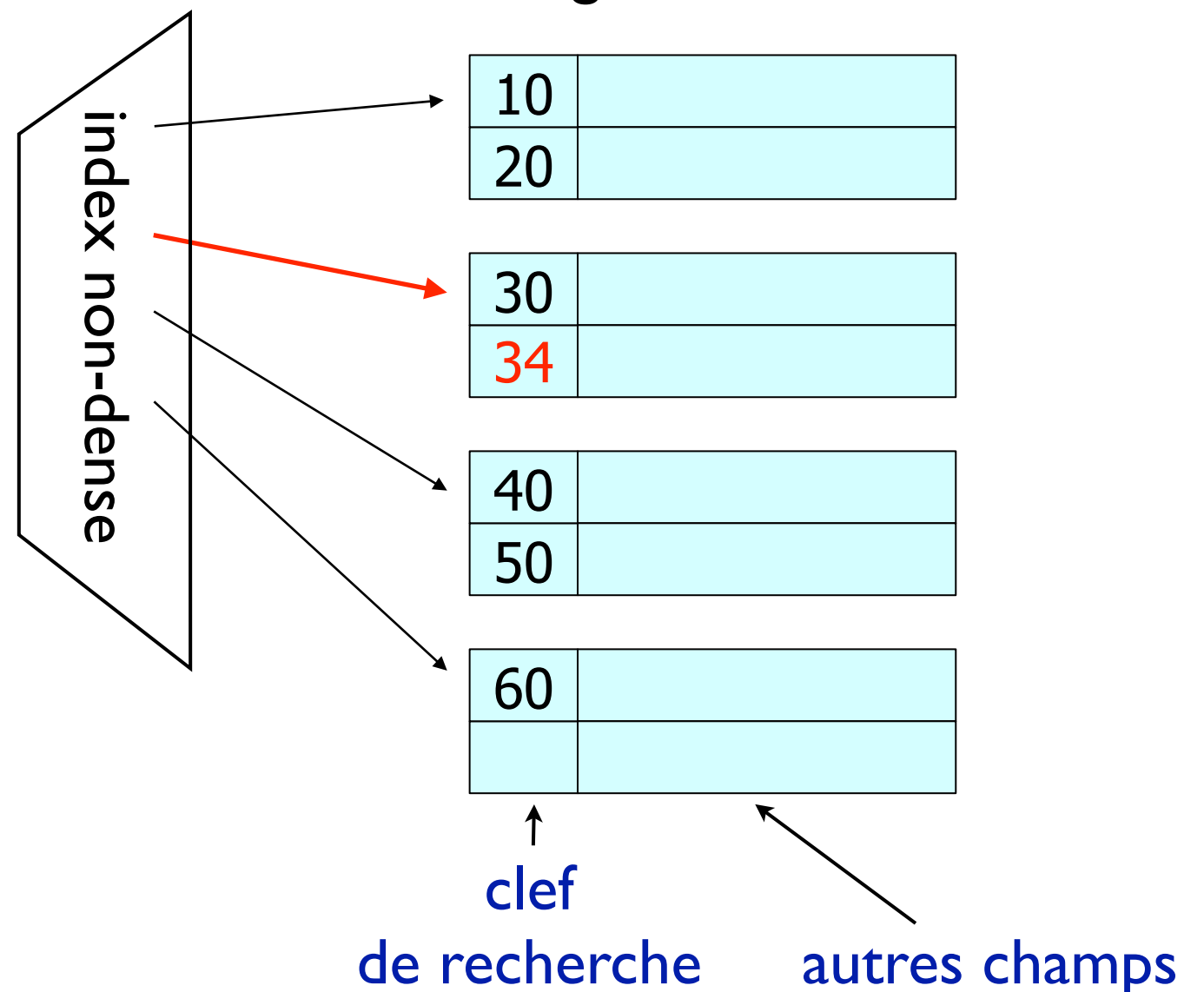


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- Trouver le bloc où l'enregistrement doit être inséré (déterminé par sa clef de recherche)
 - ▶ l'index est utilisé à cet effet, voir plus loin
- S'il y a de la place dans le bloc, y insérer le nouvel enregistrement

- Ex. Insérer un enregistrement avec clef de recherche = 34

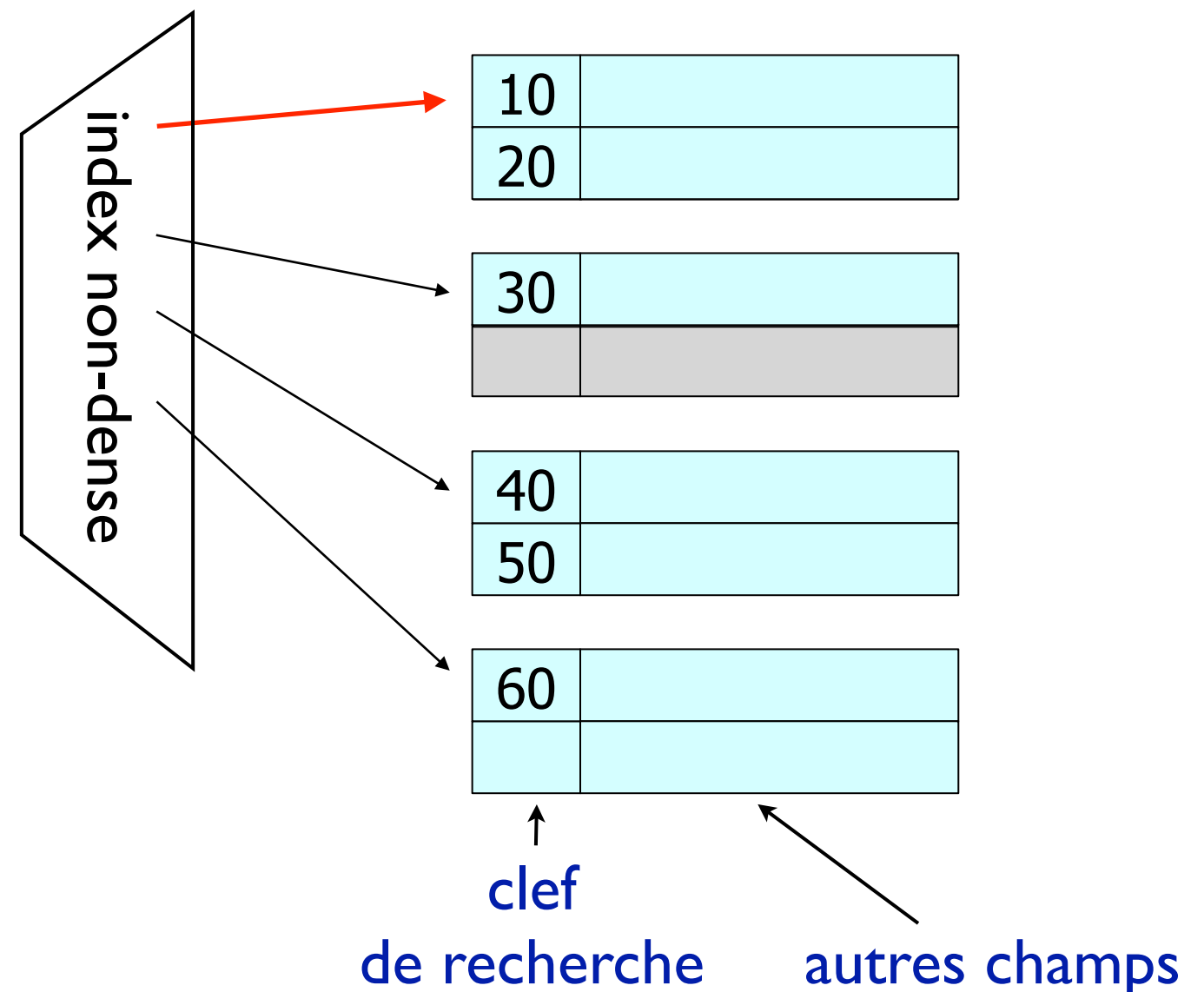


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place dans le bloc, mais il y a de la place dans un bloc voisin
 - ▶ déplacer le (les) dernier(s) enregistrement(s) du bloc au début du suivant.
 - ▶ stocker le nouvel enregistrement dans le bloc

- Ex. Insérer un enregistrement avec clef de recherche = 15

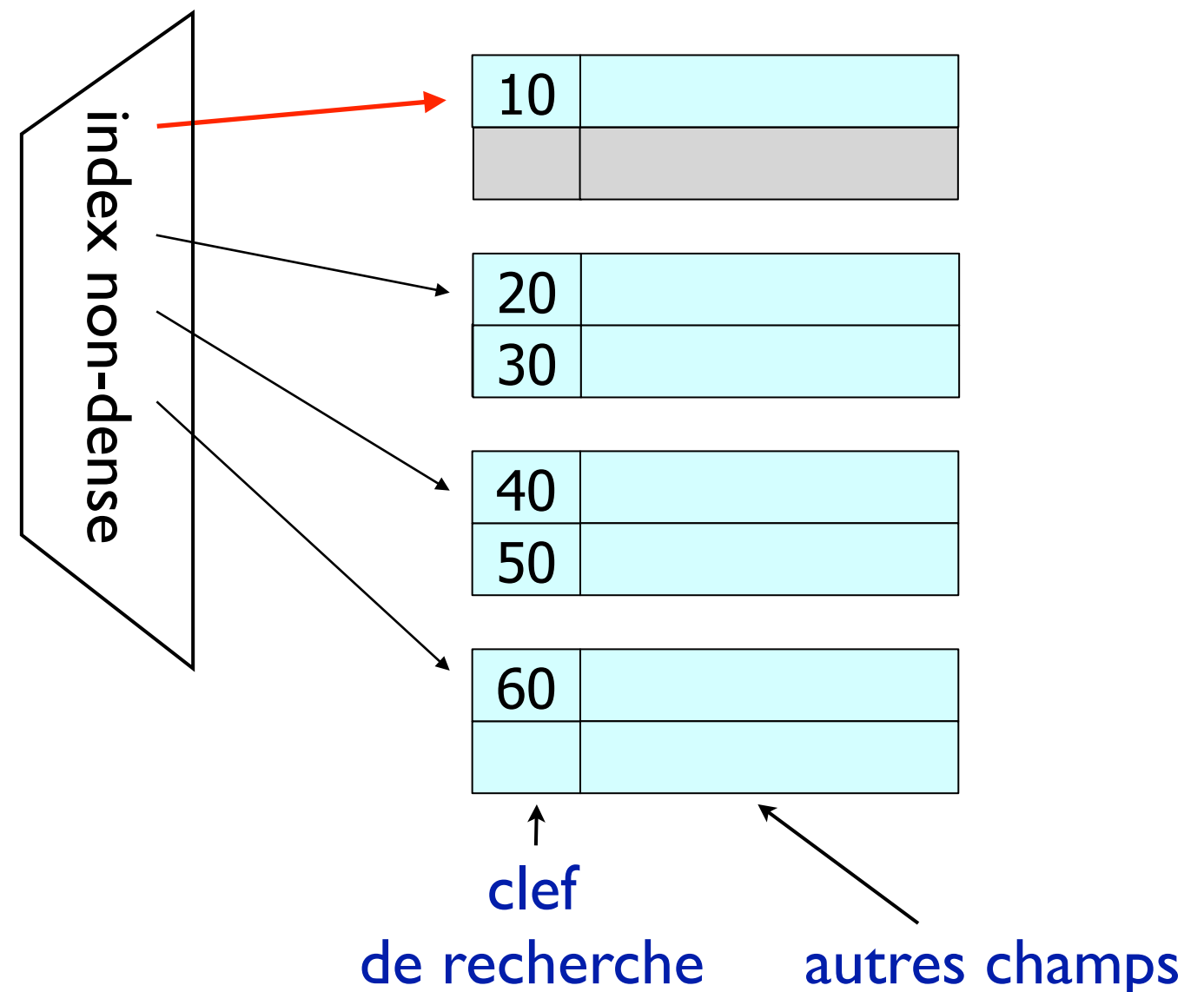


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place dans le bloc, mais il y a de la place dans un bloc voisin
 - ▶ déplacer le (les) dernier(s) enregistrement(s) du bloc au début du suivant.
 - ▶ stocker le nouvel enregistrement dans le bloc

- Ex. Insérer un enregistrement avec clef de recherche = 15

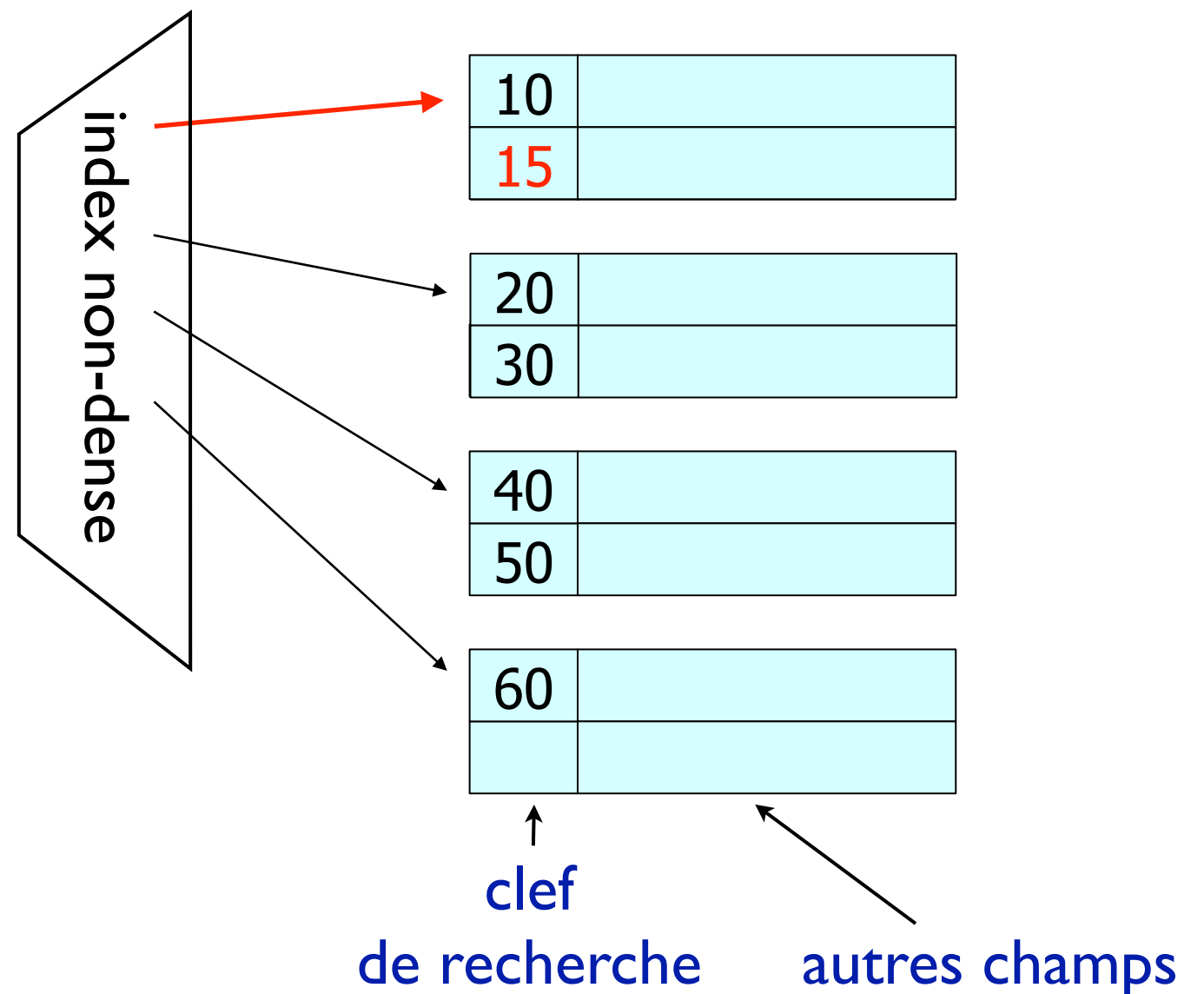


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place dans le bloc, mais il y a de la place dans un bloc voisin
 - ▶ déplacer le (les) dernier(s) enregistrement(s) du bloc au début du suivant.
 - ▶ stocker le nouvel enregistrement dans le bloc

- **Ex.** Insérer un enregistrement avec clef de recherche = **15**

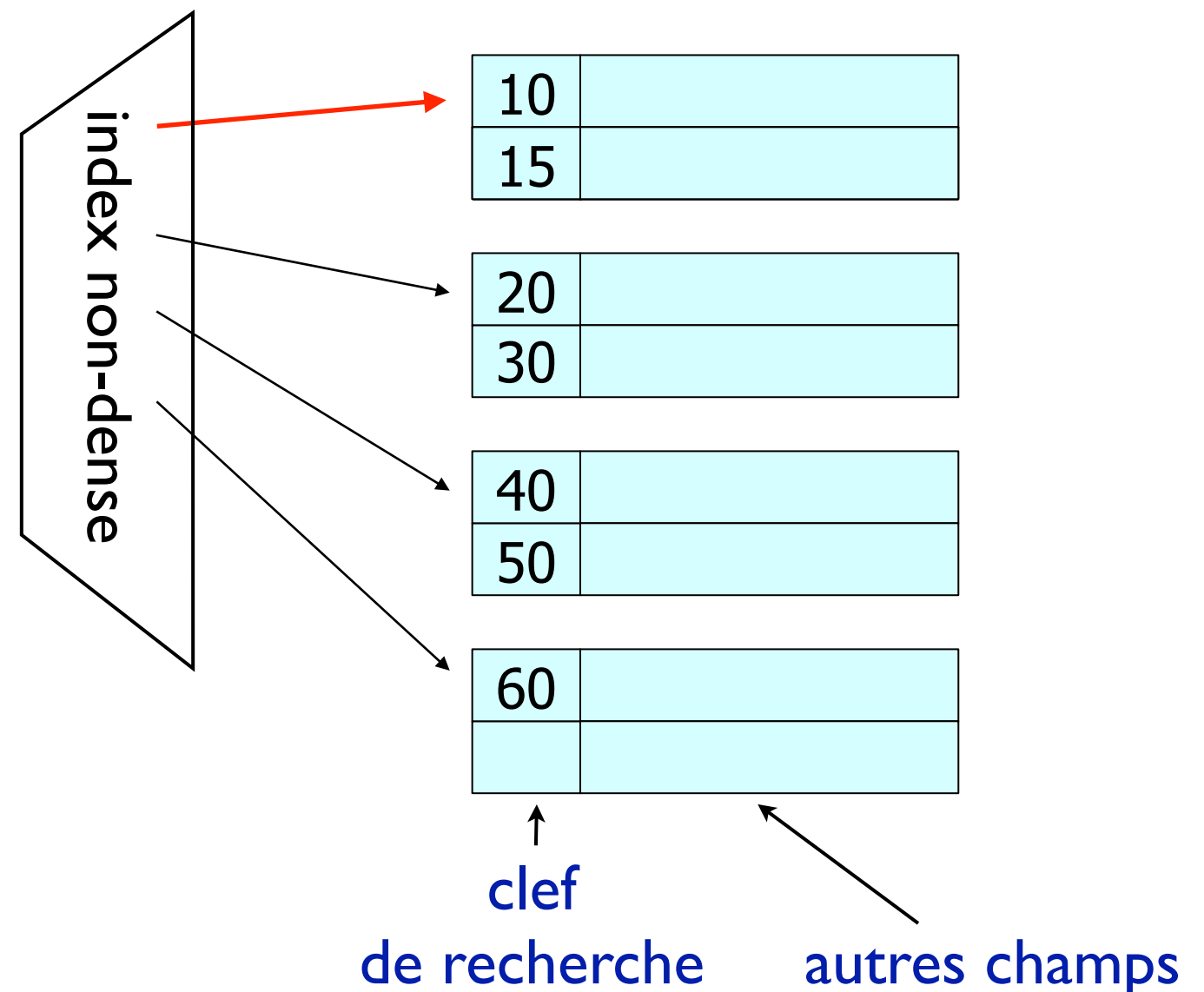


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place ni dans le bloc, ni dans les blocs voisin
 - ▶ créer un nouveau bloc qui suit le bloc d'origine dans l'ordre, et y stocker le (les) dernier(s) enregistrements du bloc d'origine
 - ▶ stocker le nouvel enregistrement dans le bloc d'origine

- Ex. Insérer un enregistrement avec clef de recherche = 12

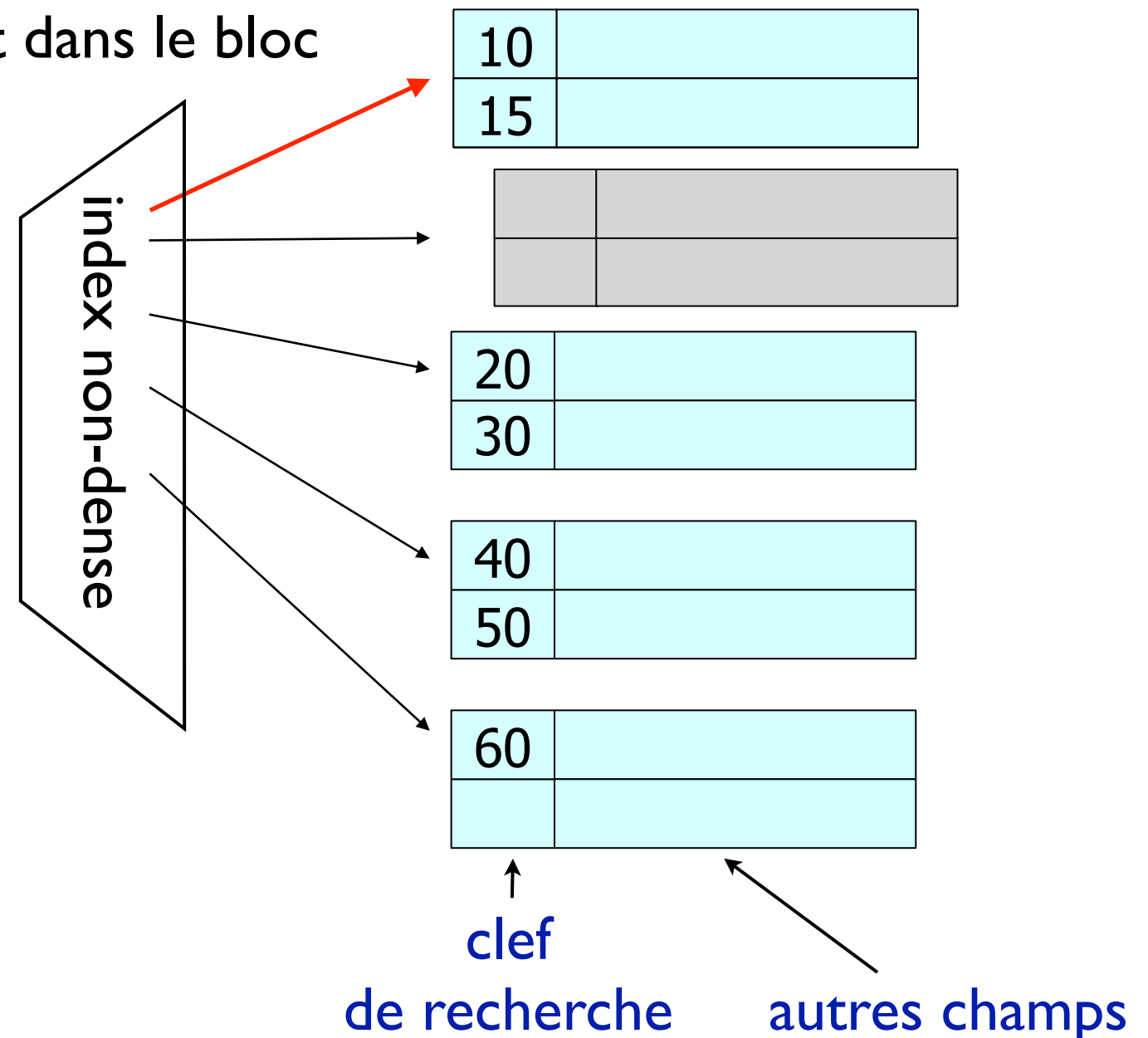


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place ni dans le bloc, ni dans les blocs voisin
 - ▶ créer un nouveau bloc qui suit le bloc d'origine dans l'ordre, et y stocker le (les) dernier(s) enregistrements du bloc d'origine
 - ▶ stocker le nouvel enregistrement dans le bloc d'origine

- Ex. Insérer un enregistrement avec clef de recherche = 12

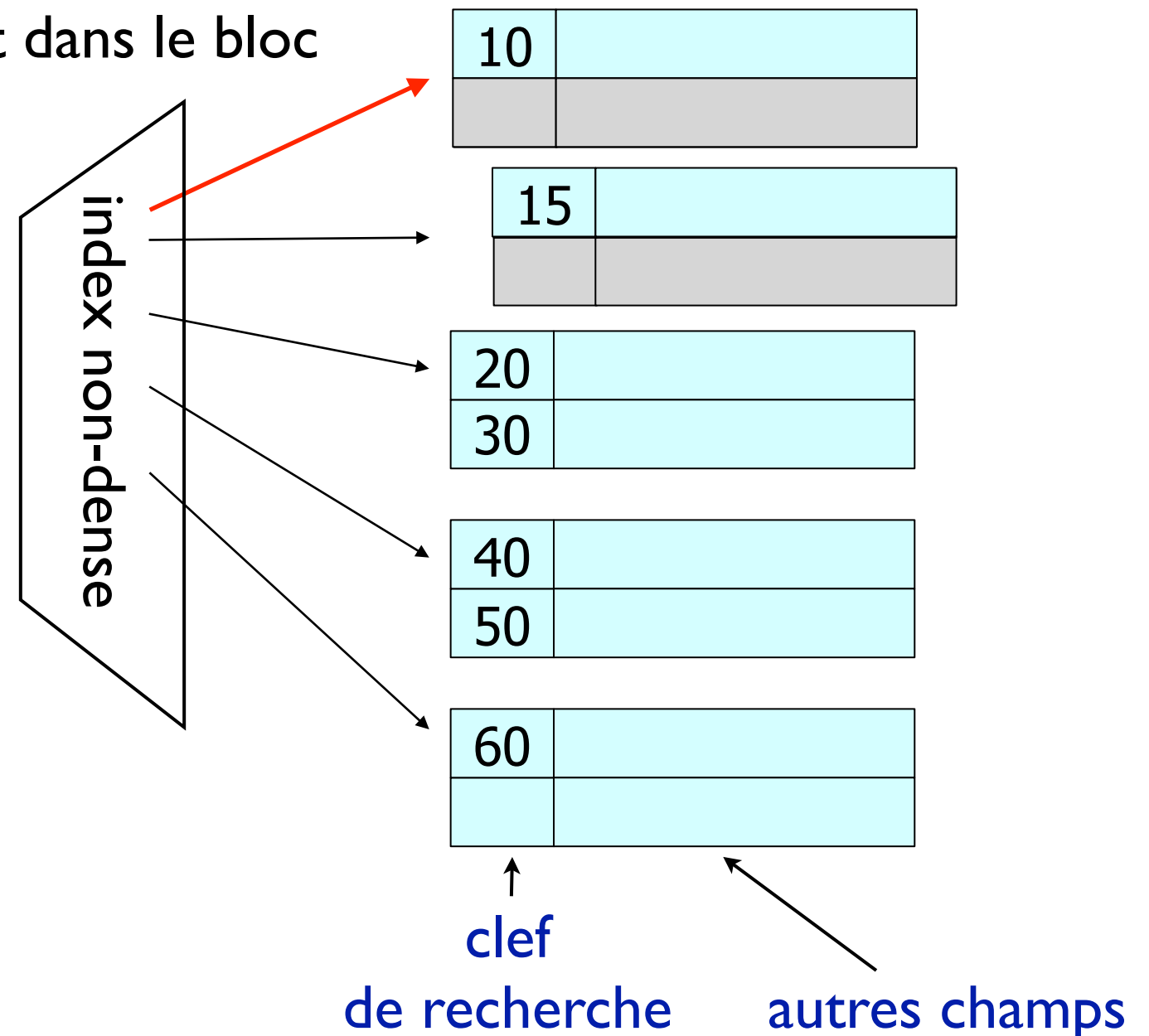


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place ni dans le bloc, ni dans les blocs voisins
 - ▶ créer un nouveau bloc qui suit le bloc d'origine dans l'ordre, et y stocker le (les) dernier(s) enregistrements du bloc d'origine
 - ▶ stocker le nouvel enregistrement dans le bloc d'origine

- Ex. Insérer un enregistrement avec clef de recherche = 12

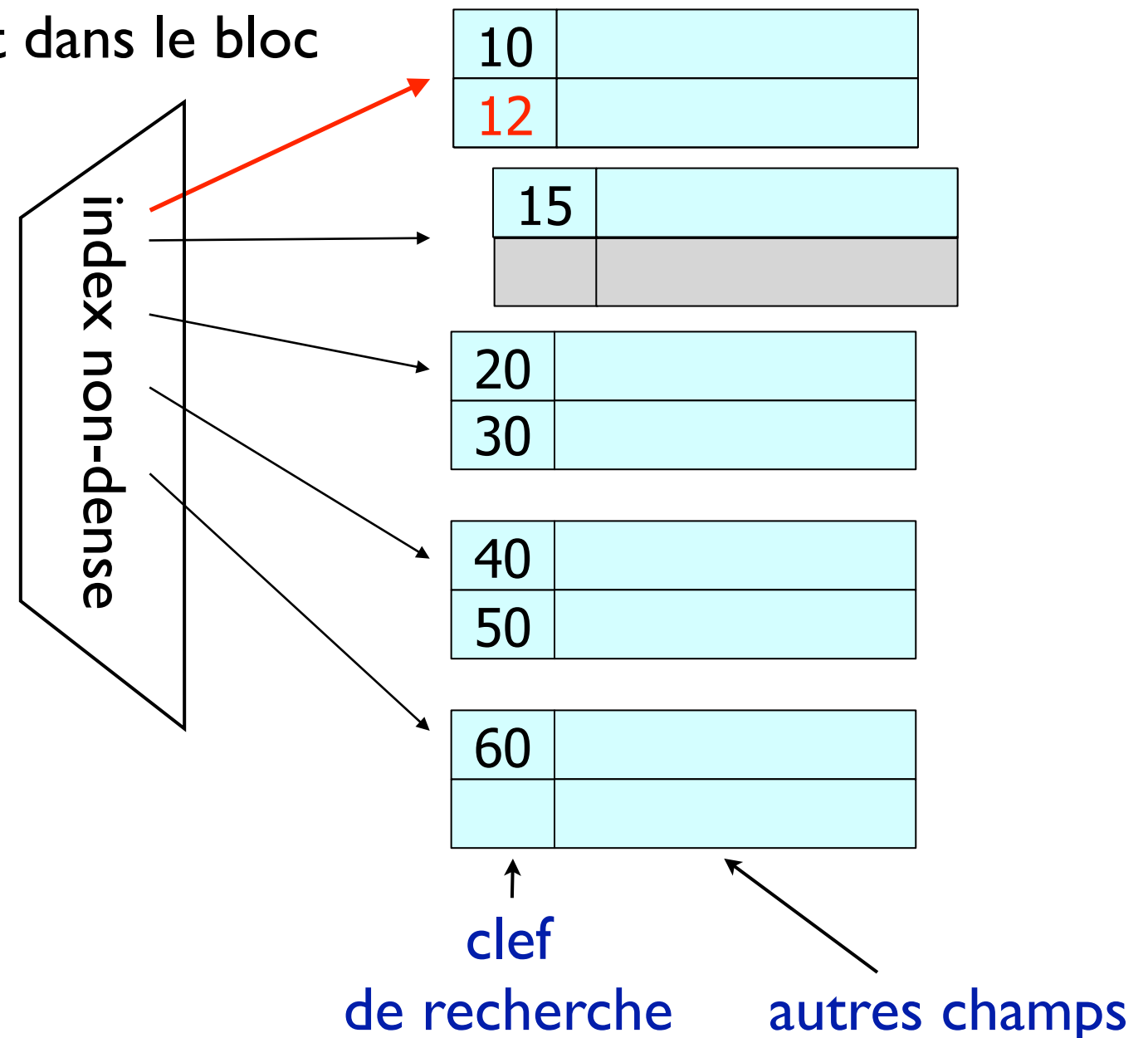


Maintenir l'organisation **séquentielle** d'un fichier de données

Insertion d'un nouvel enregistrement

- S'il n'y a pas de place ni dans le bloc, ni dans les blocs voisin
 - ▶ créer un nouveau bloc qui suit le bloc d'origine dans l'ordre, et y stocker le (les) dernier(s) enregistrements du bloc d'origine
 - ▶ stocker le nouvel enregistrement dans le bloc d'origine

- Ex. Insérer un enregistrement avec clef de recherche = 12

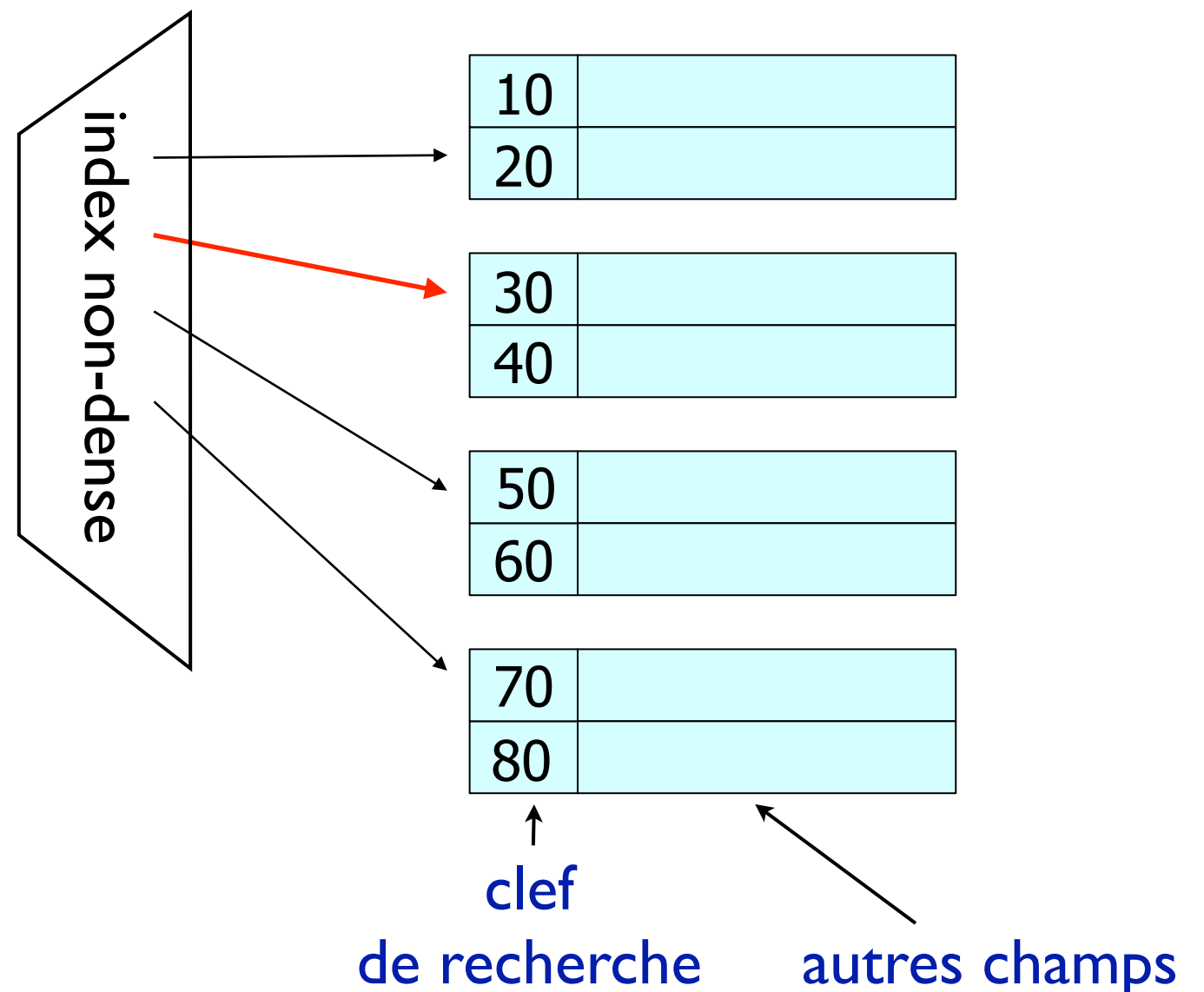


Maintenir l'organisation **séquentielle** d'un fichier de données

Suppression d'un enregistrement

- Trouver le bloc contenant l'enregistrement (à travers l'index)
 - ▶ libérer l'espace qu'il occupe (re-compacter les autres enregistrements)

- Ex. Eliminer l'enregistrement avec clef de recherche = 30

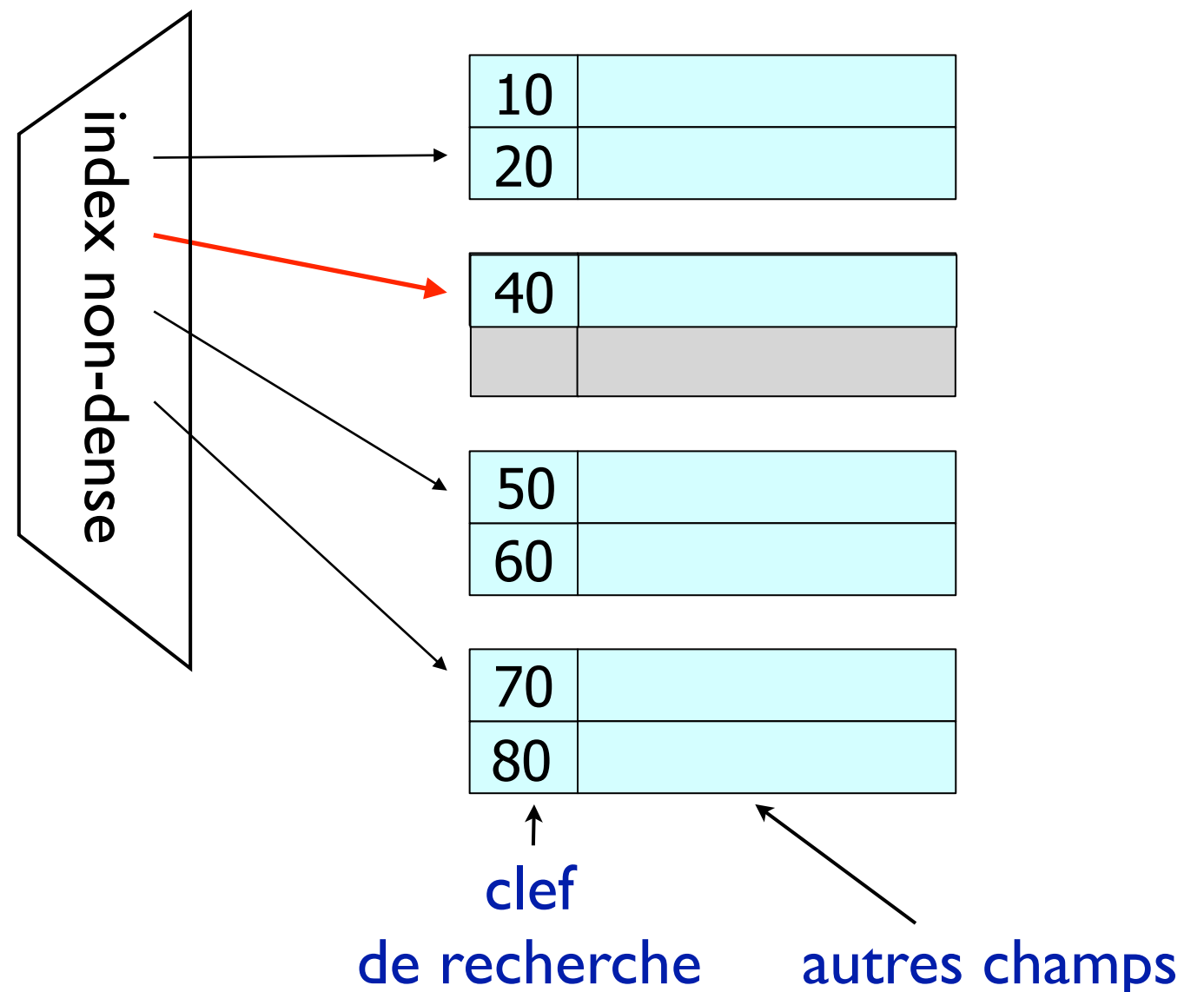


Maintenir l'organisation **séquentielle** d'un fichier de données

Suppression d'un enregistrement

- Trouver le bloc contenant l'enregistrement (à travers l'index)
 - ▶ libérer l'espace qu'il occupe (re-compacter les autres enregistrements)

- Ex. Eliminer l'enregistrement avec clef de recherche = 30

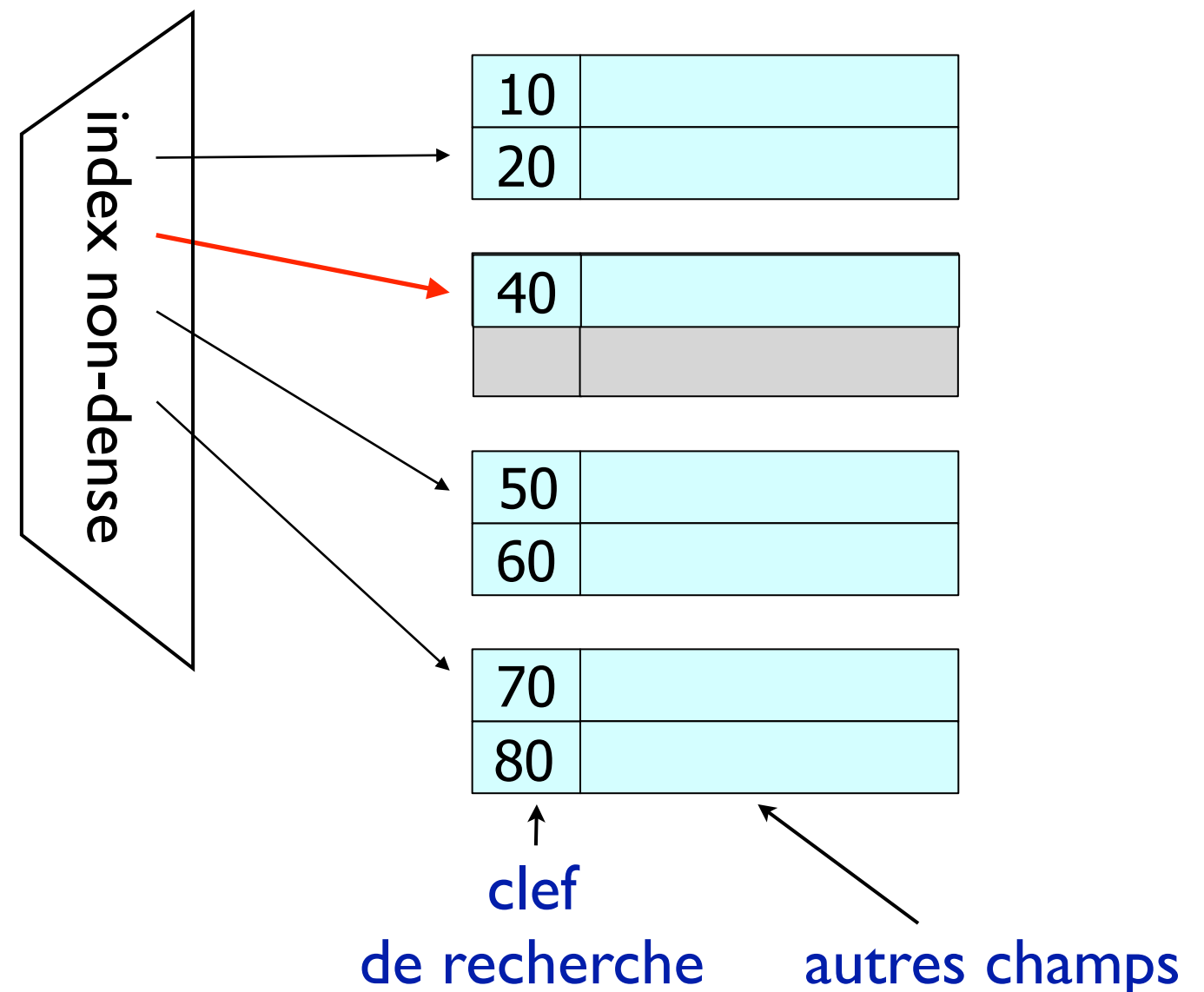


Maintenir l'organisation **séquentielle** d'un fichier de données

Suppression d'un enregistrement

- Trouver le bloc contenant l'enregistrement (à travers l'index)
 - ▶ libérer l'espace qu'il occupe (re-compacter les autres enregistrements)
 - ▶ si le bloc reste vide l'éliminer

- Ex. Eliminer l'enregistrement avec clef de recherche = 30



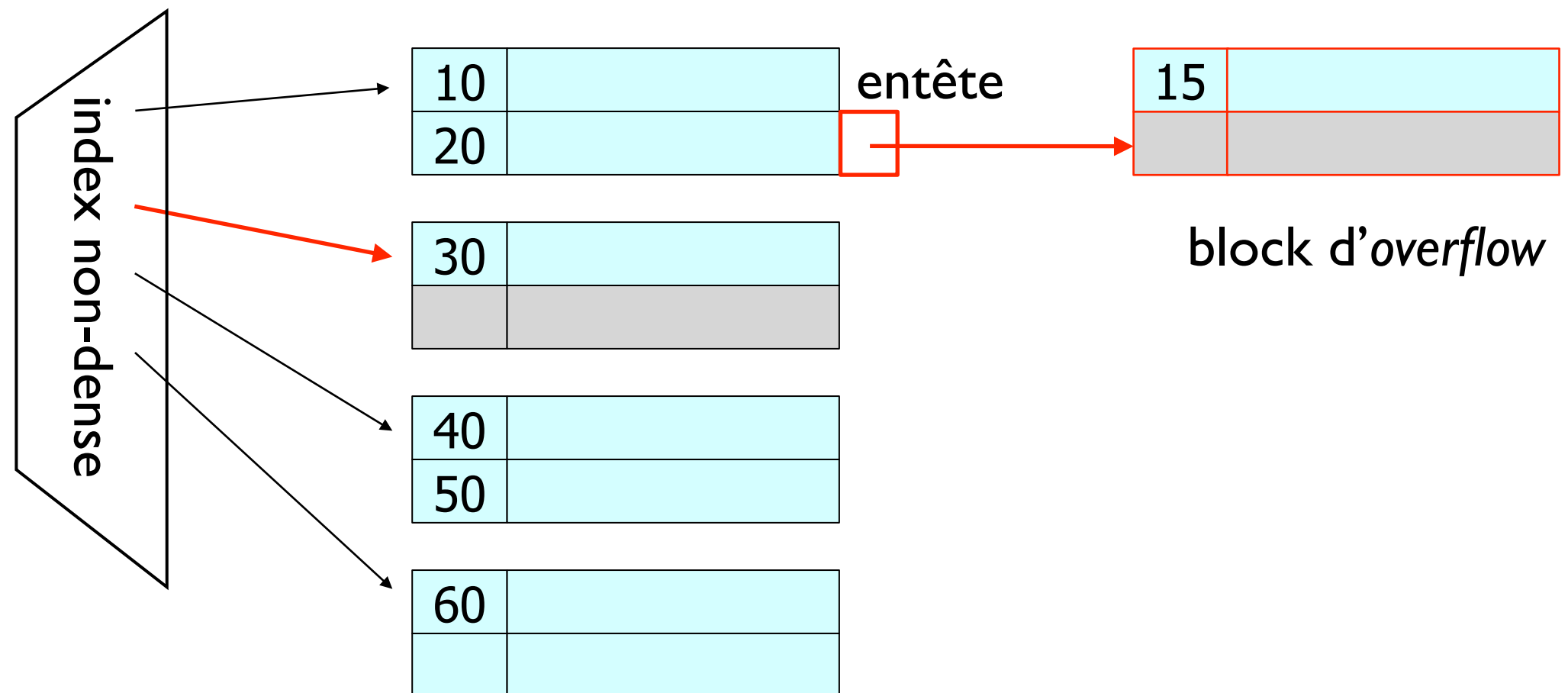
Maintenir l'organisation **séquentielle** d'un fichier de données

- **Insertion et suppression** : dans la plupart des cas l'index doit être également mis à jour
 - ▶ l'insertion / élimination de pointeurs à blocs (inévitables)
 - ▶ déplacement d'enregistrements (surcoût)
- Pour éviter le surcoût immédiat on peut préférer une **stratégie de réorganisation différée**, qui évite les déplacements d'enregistrements
 - ▶ utilisée également quand les adresses sont absolues (pas de table de offset dans les blocs)

Maintenir l'organisation **séquentielle** d'un fichier de données

Réorganisation différée : chaque pointeur dans l'index pointe à un bloc principal connecté à une liste de blocs d'*overflow*

- **Ex d'insertion.** Insérer un enregistrement avec clef de recherche = 15



pour maintenir l'ordre : chaque enregistrement pointe à son successeur

- **Suppression** : mettre le bit "effacé" de l'enregistrement à 1
- **Réorganisation périodique nécessaire** : reconstruire l'ordre, éliminer les *overflow*

Organisation en “grappe” multi-table

Plusieurs tables sont stockées dans le même fichier de données

table département

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

table enseignant

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

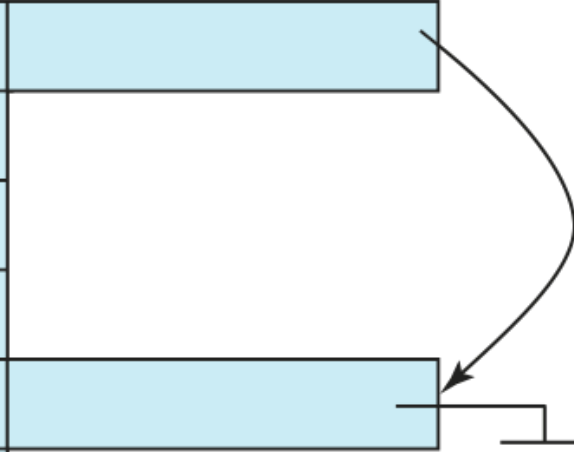
fichier de données
en “grappe” multi-table de
département et
enseignant

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Organisation en “grappe” multi-table

- Efficace pour des requêtes de jointure *departement* ⋈ *enseignant*
- Mauvaise pour des requêtes uniquement sur *departement*
 - ▶ on peut ajouter des chaînes de pointeurs pour relier les enregistrements d’une relation particulière

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	



- Enregistrements de taille variables plus difficiles à gérer

Indexation

Indexation

- **Objectif** : être capable de trouver efficacement les n-uplets d'une table avec une valeur particulière d'un (ou plusieurs) attribut(s)

$\sigma_{A1 = a1, \dots, Ak = ak} (R)$ c'est-à-dire

```
SELECT ...  
FROM R  
WHERE A1=a1 AND ...  
AND Ak= ak
```

clef de recherche : $A1, \dots, Ak$, valeur de la clef de recherche : $a1, \dots, ak$

- **Modèle de coût** pour évaluer l'efficacité :
 - ▶ **Mesure du coût** : nombre de lectures/écritures de blocs du disque
 - ▶ **Justification** :
 - coût d'un accès au disque (lecture/ écriture d'un bloc) : millisecondes
 - coût d'un accès en mémoire centrale : nanosecondes
- ⇒ toutes les opérations de traitement d'un bloc en mémoire centrale ont un coût négligeable

Indexation

```
SELECT ...  
FROM R  
WHERE A1=a1 AND ... AND Ak= ak
```

- Approche naïve : parcourir tous les blocs de R
 - ▶ Coût : environ $\lceil n r / B \rceil$
 - n : nombre d'enregistrements dans R,
 - r : taille d'un enregistrement
 - B : taille d'un bloc
- Si n est très grand cela peut être prohibitif

Index : une structure physique auxiliaire construite sur une clef de recherche ($A1...Ak$) qui permet d'accéder “presque directement” aux n -uplets ayant une valeur particulière de la clef de recherche

Index

- D'un point de vue abstrait, un index sur une clef de recherche est une collection de couples $\langle \text{clef}, \text{pointeur} \rangle$

$\langle C, \bullet \rangle$

pointeur à un enregistrement
ayant valeur C de la clef de recherche

$\langle 10101, \rightarrow \rangle$	→	10101	Srinivasan	Comp. Sci.	65000
$\langle 12121, \rightarrow \rangle$	→	12121	Wu	Finance	90000
$\langle 15151, \rightarrow \rangle$	→	15151	Mozart	Music	40000
$\langle 22222, \rightarrow \rangle$	→	22222	Einstein	Physics	95000
$\langle 32343, \rightarrow \rangle$	→	32343	El Said	History	60000
$\langle 33456, \rightarrow \rangle$	→	33456	Gold	Physics	87000
$\langle 45565, \rightarrow \rangle$	→	45565	Katz	Comp. Sci.	75000
$\langle 58583, \rightarrow \rangle$	→	58583	Califieri	History	62000
$\langle 76543, \rightarrow \rangle$	→	76543	Singh	Finance	80000
$\langle 76766, \rightarrow \rangle$	→	76766	Crick	Biology	72000
$\langle 83821, \rightarrow \rangle$	→	83821	Brandt	Comp. Sci.	92000
$\langle 98345, \rightarrow \rangle$	→	98345	Kim	Elec. Eng.	80000

index

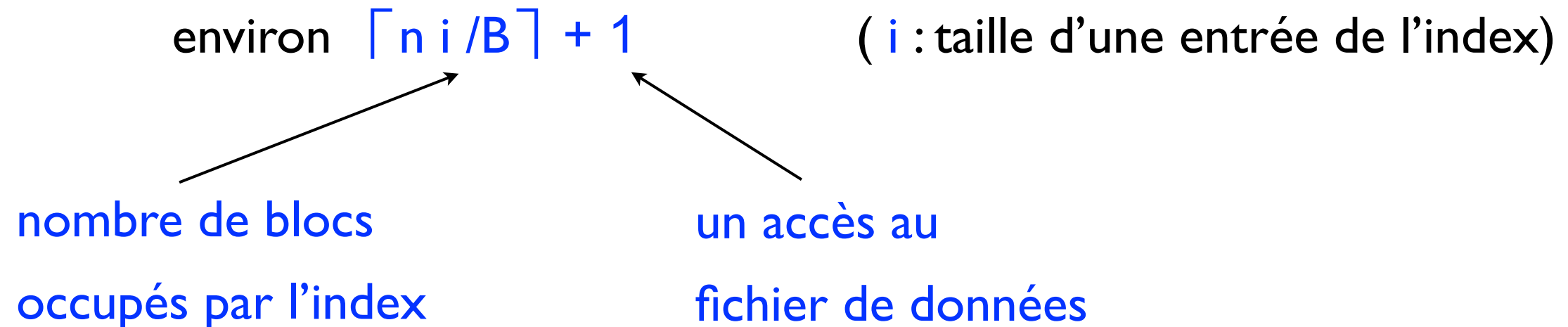
fichier de données

Indexation : principe

- Rechercher des enregistrements ayant une valeur c de la clef de recherche (pour simplifier supposer clef de recherche = clef primaire) :
 - ▶ parcourir l'index
 - ▶ si un couple $\langle c, \cdot \rangle$ est trouvé suivre le pointeur
 - charger en mémoire centrale le bloc contenant l'enregistrement
- Nombres d'entrées dans l'index : au plus le nombre n d'enregistrements dans le fichier de données
- Mais un couple $\langle c, \cdot \rangle$ occupe beaucoup moins de place qu'un enregistrement !

Indexation : principe

- Coût de la recherche : dépend de l'implémentation de l'index mais au pire

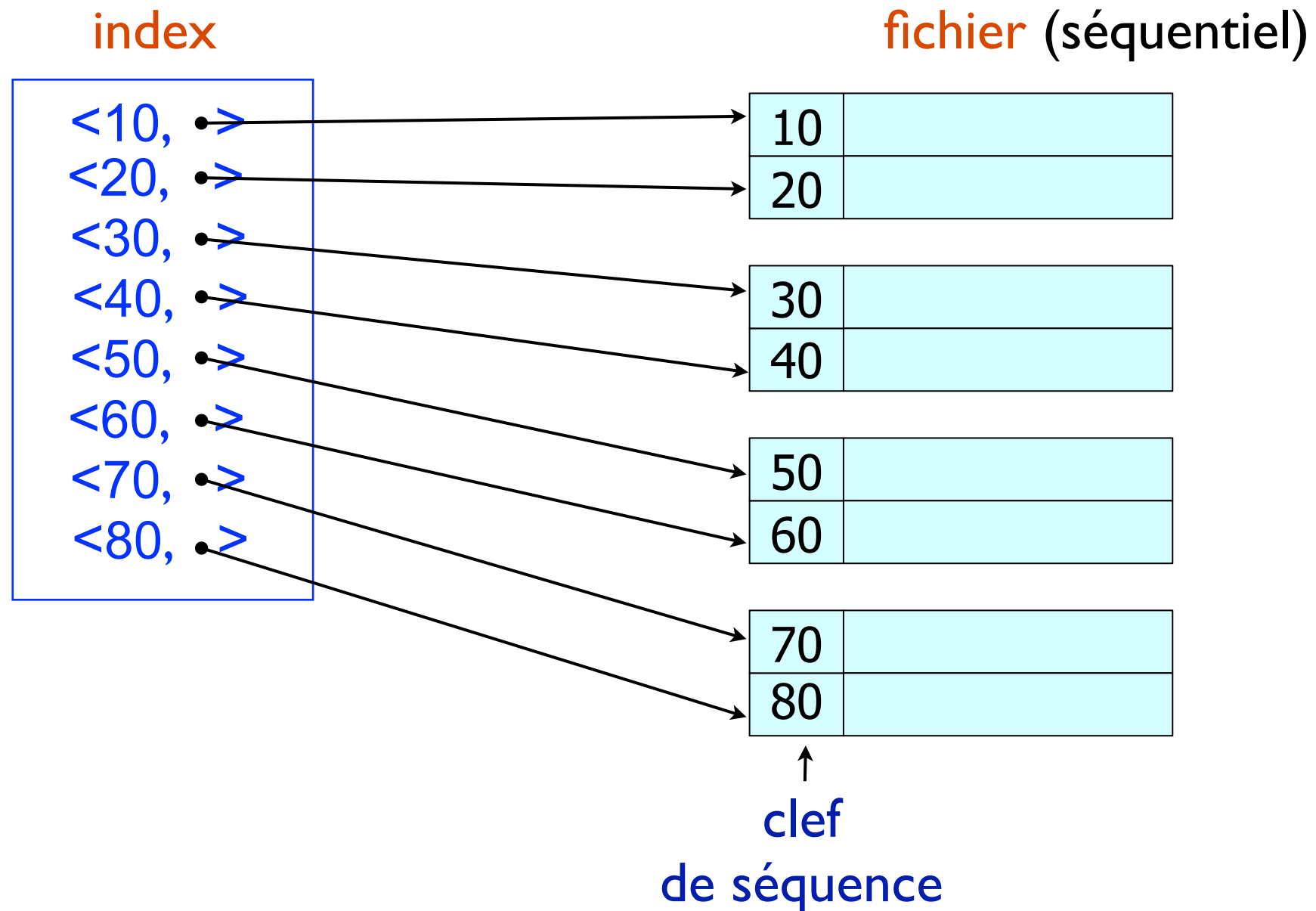


$$\lceil n i / B \rceil \ll \lceil n r / B \rceil \quad \text{puisque } i \ll r$$

- De plus il existe des implémentations efficaces des index :
le coût de la recherche d'un couple $\langle c, \cdot \rangle$ dans l'index peut passer de $\lceil n i / B \rceil$ à “presque” constant !

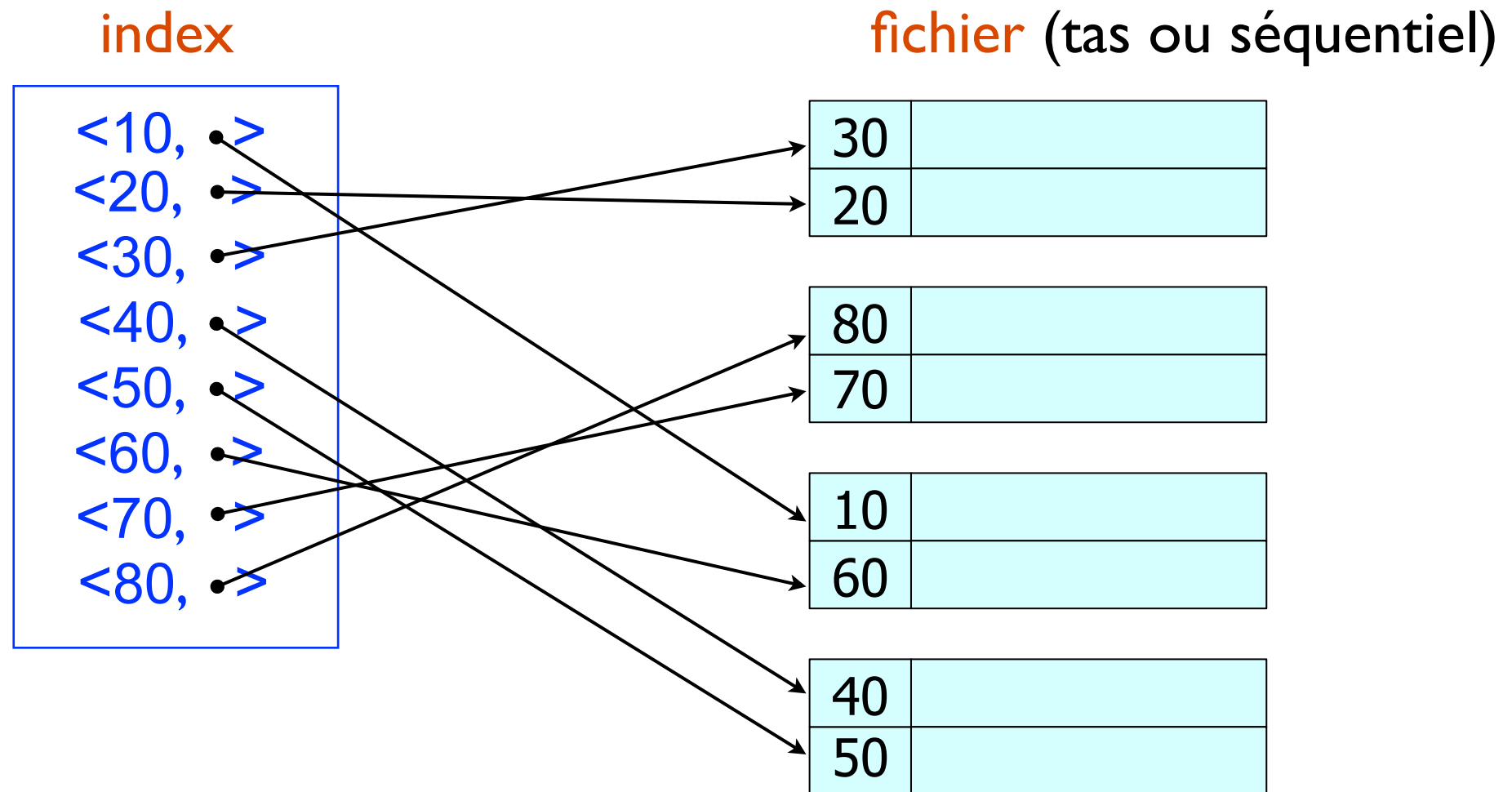
Types d'index

- **Index primaire:** le fichier (séquentiel) est trié par la clef de recherche de l'index (i.e. clef de recherche de l'index = clef de séquence du fichier)



Types d'index

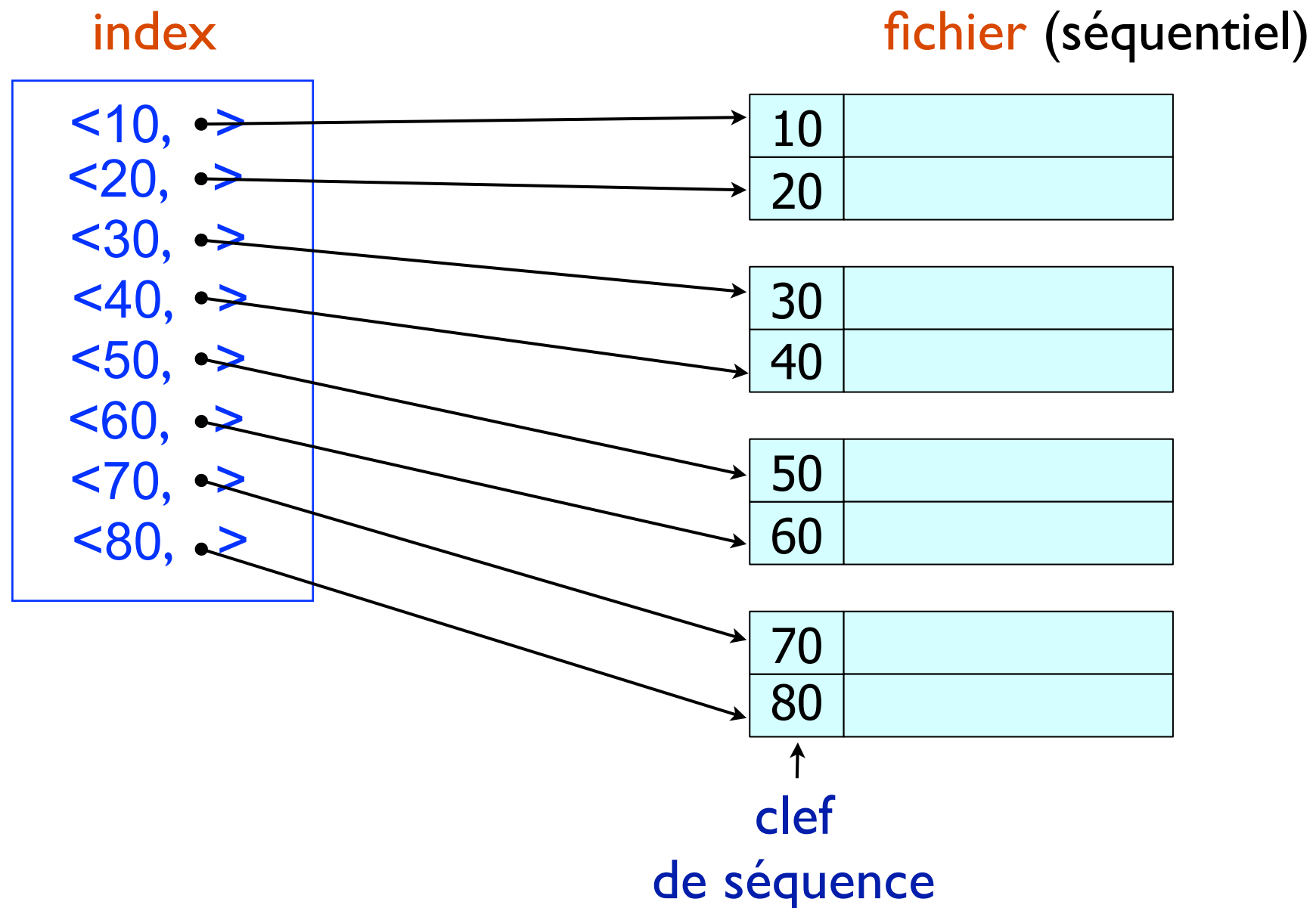
- **Index secondaire:** le fichier n'est pas trié par la clef de recherche de l'index



Types d'index

- **Index dense** : une entrée dans l'index pour chaque valeur de la clef dans le fichier

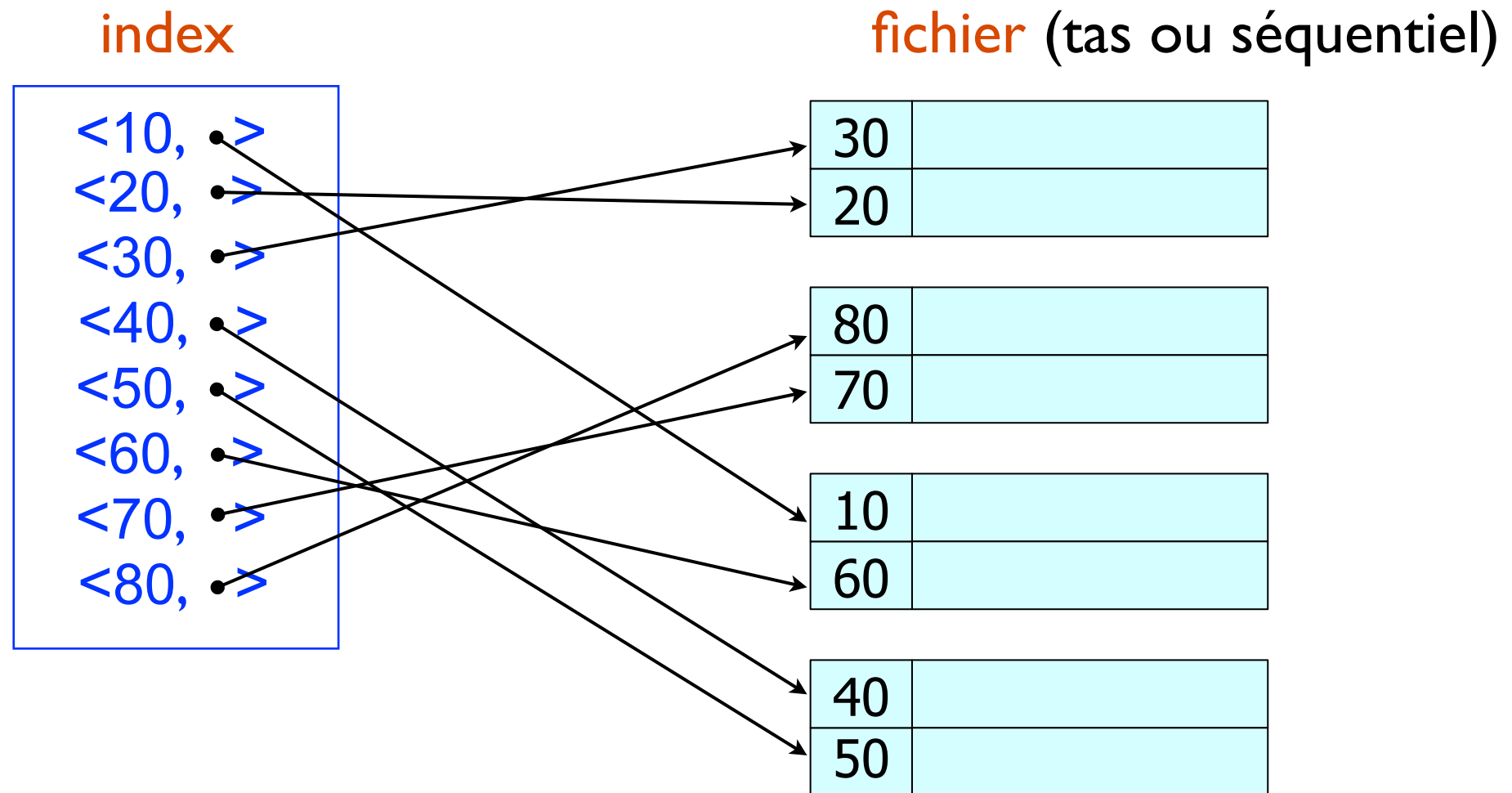
Ex. Index dense primaire



Types d'index

- **Index dense** : une entrée dans l'index pour chaque valeur de la clef dans le fichier

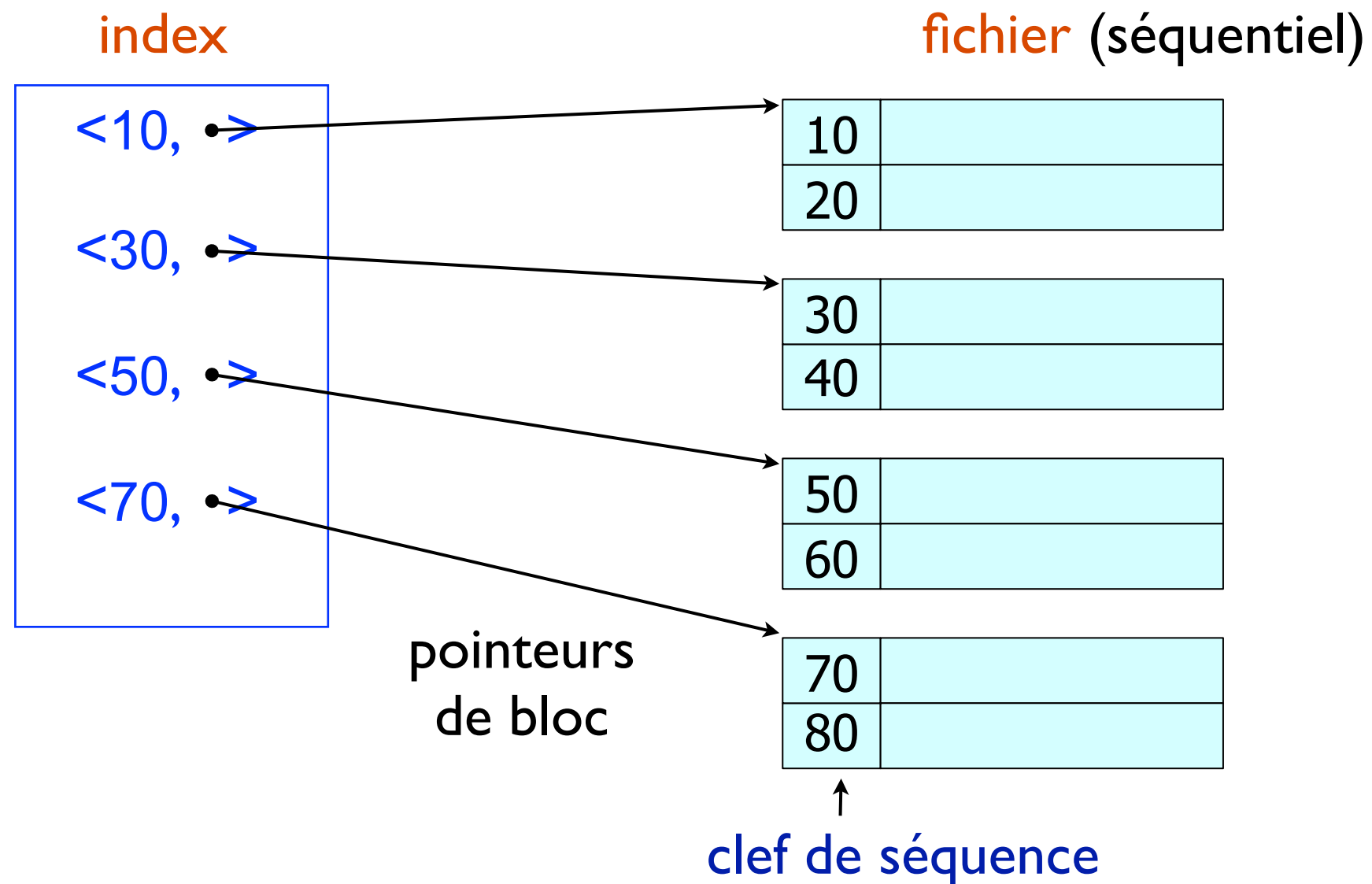
Ex. Index dense secondaire



Types d'index

- **Index non-dense** : une entrée dans l'index pour chaque bloc du fichier de données (en général la première clef du bloc)

Seulement un index primaire peut être non-dense



- **Remarque** : un index non-dense est en général présent sur un fichier séquentiel rien que pour sa gestion

Gestion des doublons

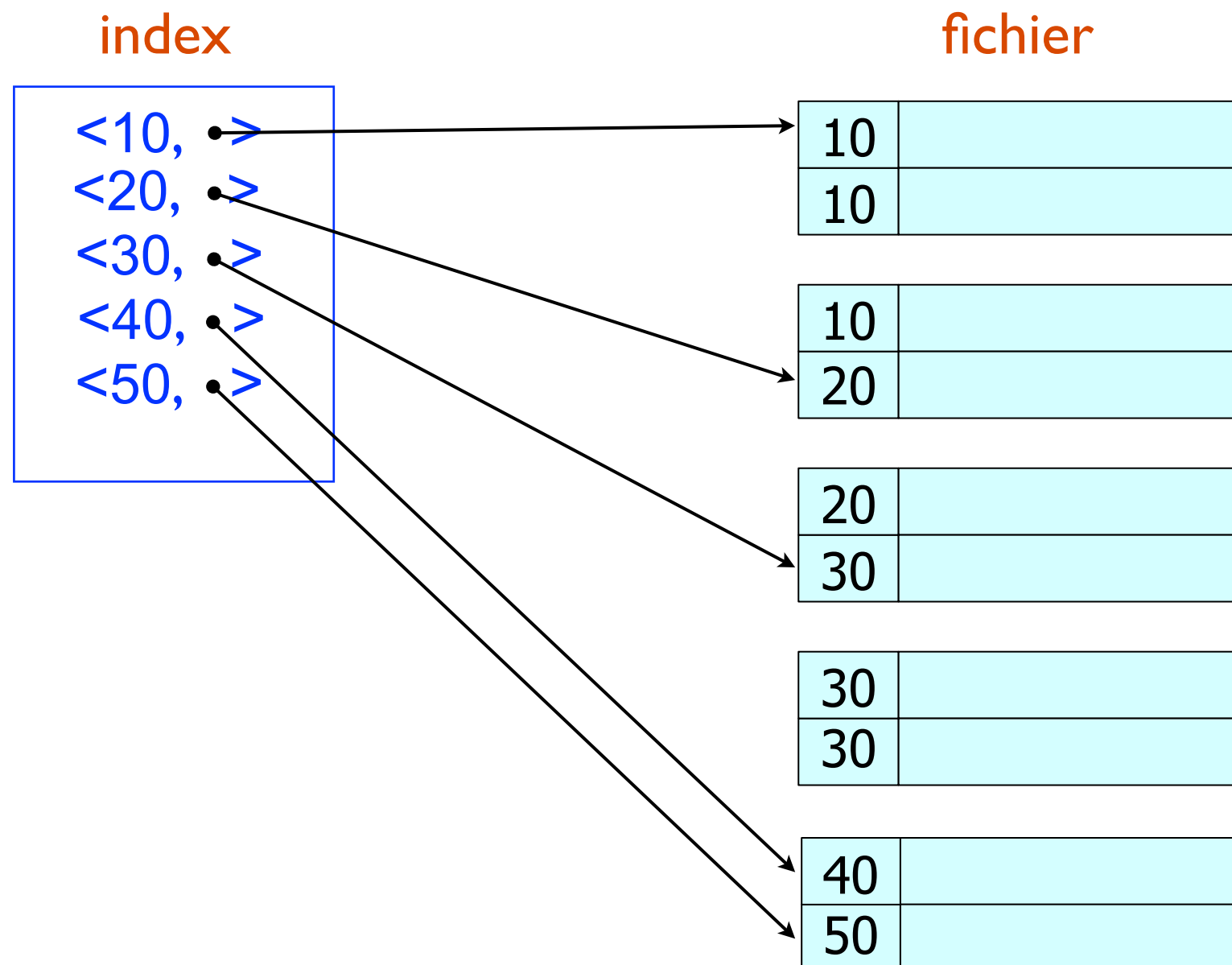
- Si la clef de recherche de l'index (dense/ non-dense, primaire/ secondaire) n'est pas une clef de la table
 - ▶ \Rightarrow doublons possibles dans l'index (plusieurs entrées avec la même clef)
 - ▶ des solutions pour les éliminer / représenter efficacement

Gestion des doublons

- Index primaire dense

une seule entrée $\langle c, p \rangle$ dans l'index pour chaque valeur c de la clef de recherche dans le fichier

- ▶ p pointe au premier enregistrement de clef c

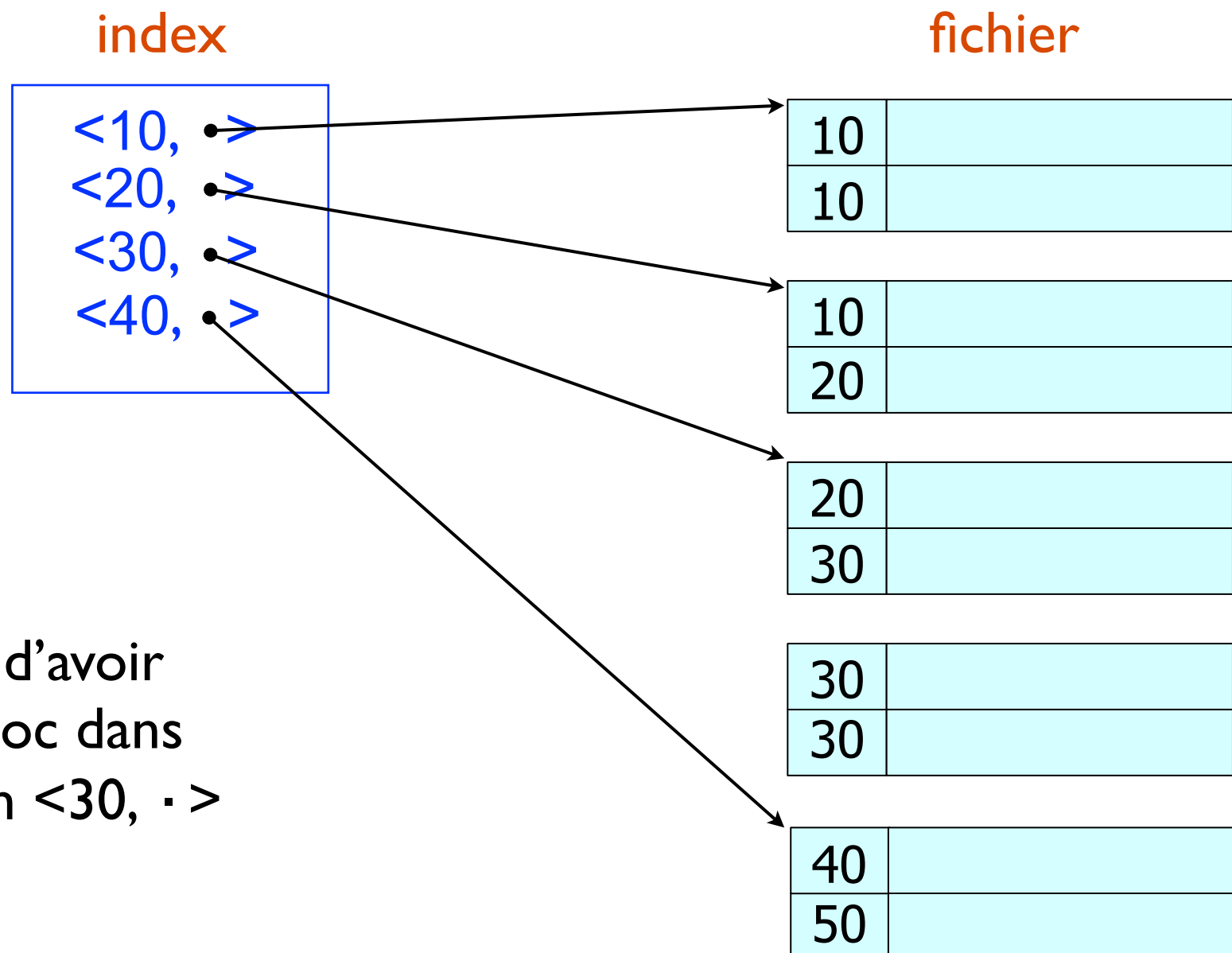


Gestion des doublons

- Index primaire non-dense

une entrée $\langle c, p \rangle$ dans l'index pour chaque bloc p

- ▶ c est la première clef du bloc p pas déjà indexée



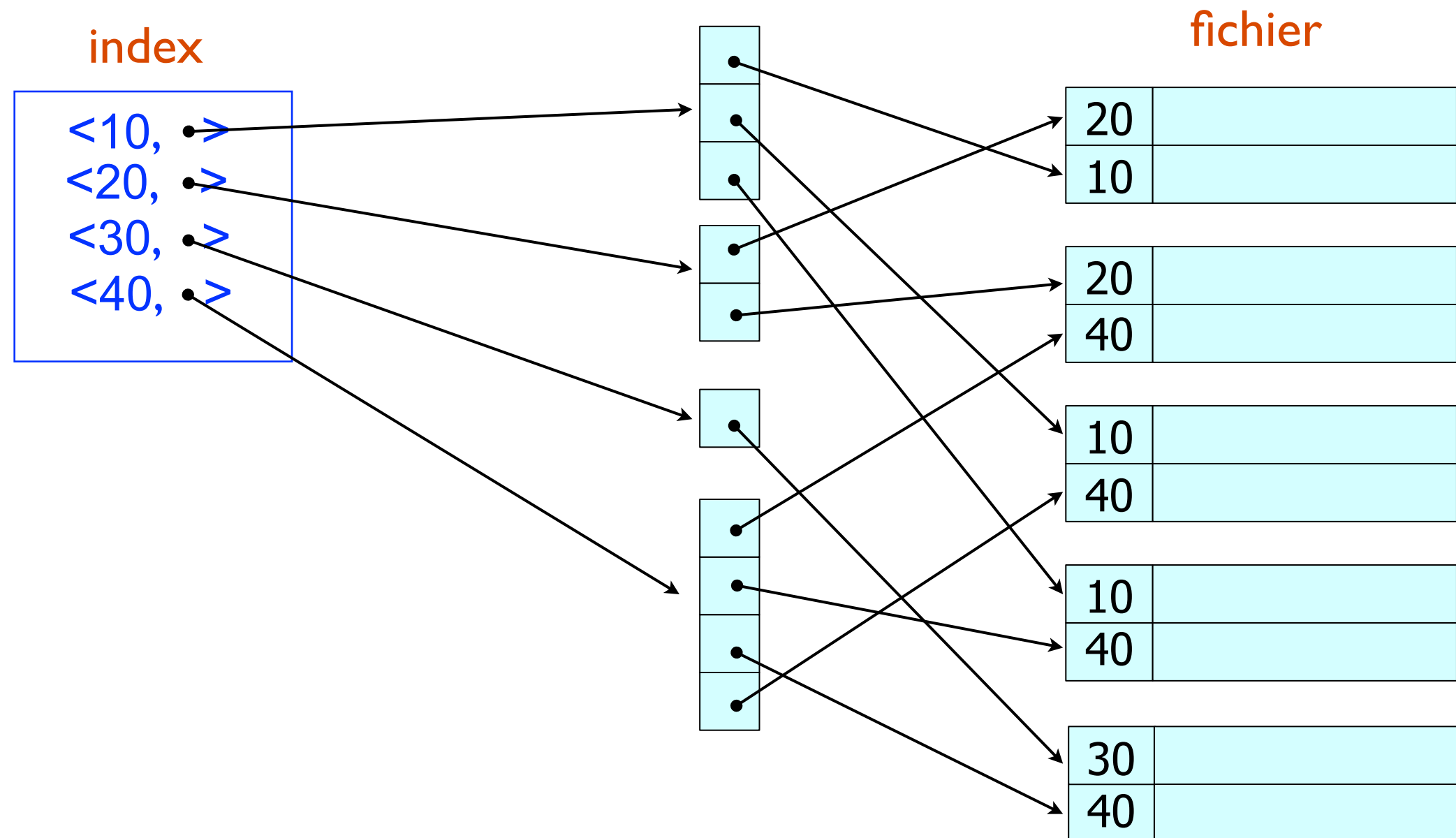
Remarque : On suppose d'avoir accès à l'avant dernier bloc dans l'ordre, sinon un doublon $\langle 30, \bullet \rangle$ est nécessaire

Gestion des doublons

- Index secondaire

une entrée $\langle c, p \rangle$ dans l'index pour chaque valeur c de la clef de recherche

► p pointe à une liste (appelé *bucket*) de pointeurs à enregistrements



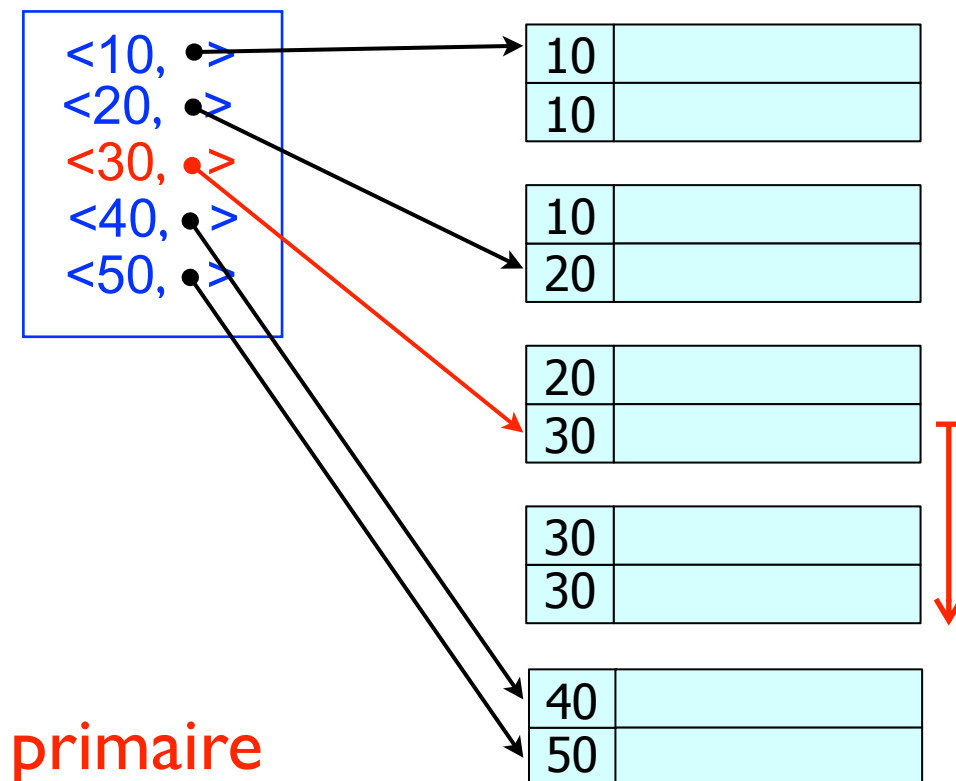
Recherche par index

Opération : recherche des enregistrements avec clef de recherche = c

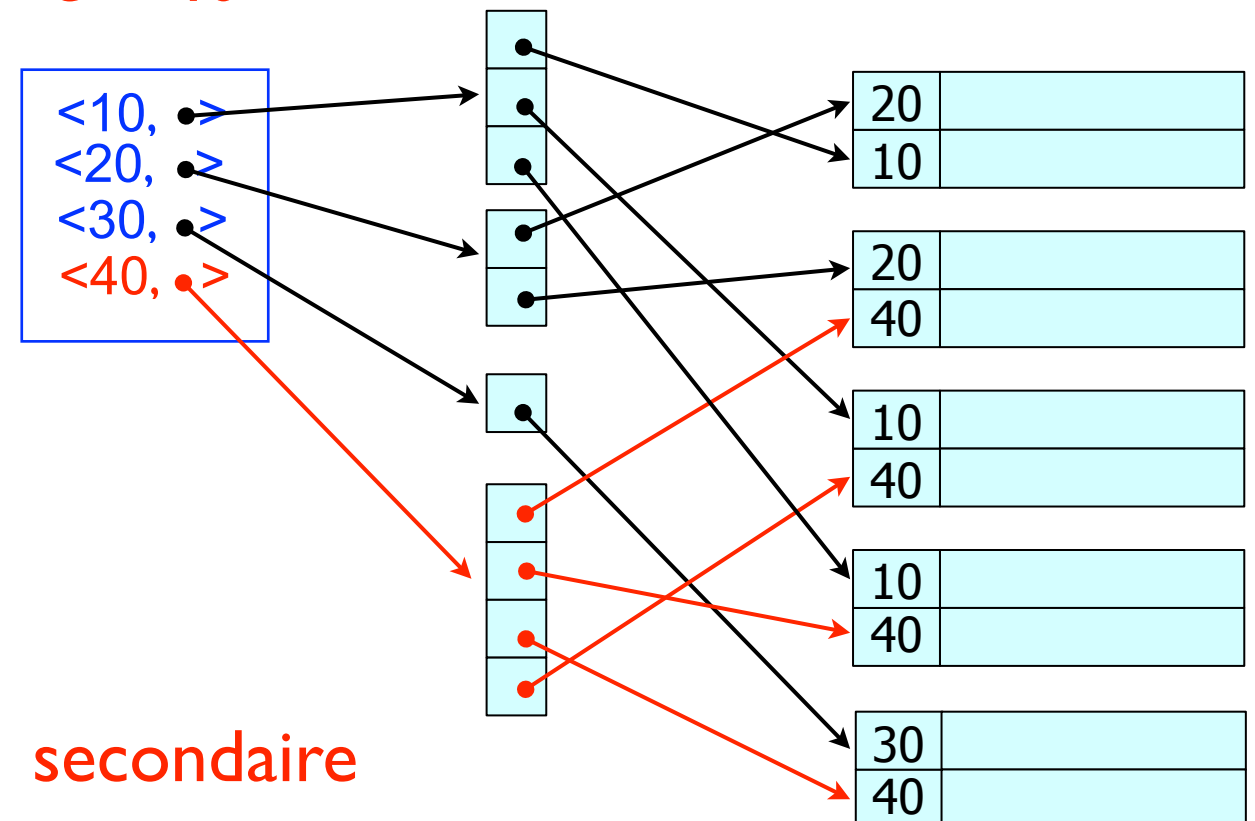
- **Index dense - cas général : doublons possibles**

- ▶ chercher dans l'index l'entrée de clef c , soit $\langle c, p \rangle$, si elle existe
- ▶ **index primaire** : à partir de l'adresse p parcourir dans le fichier tous les enregistrements de clef c
- ▶ **index secondaire** : accéder à tous les enregistrements pointés par le *bucket* p

$c = 30$



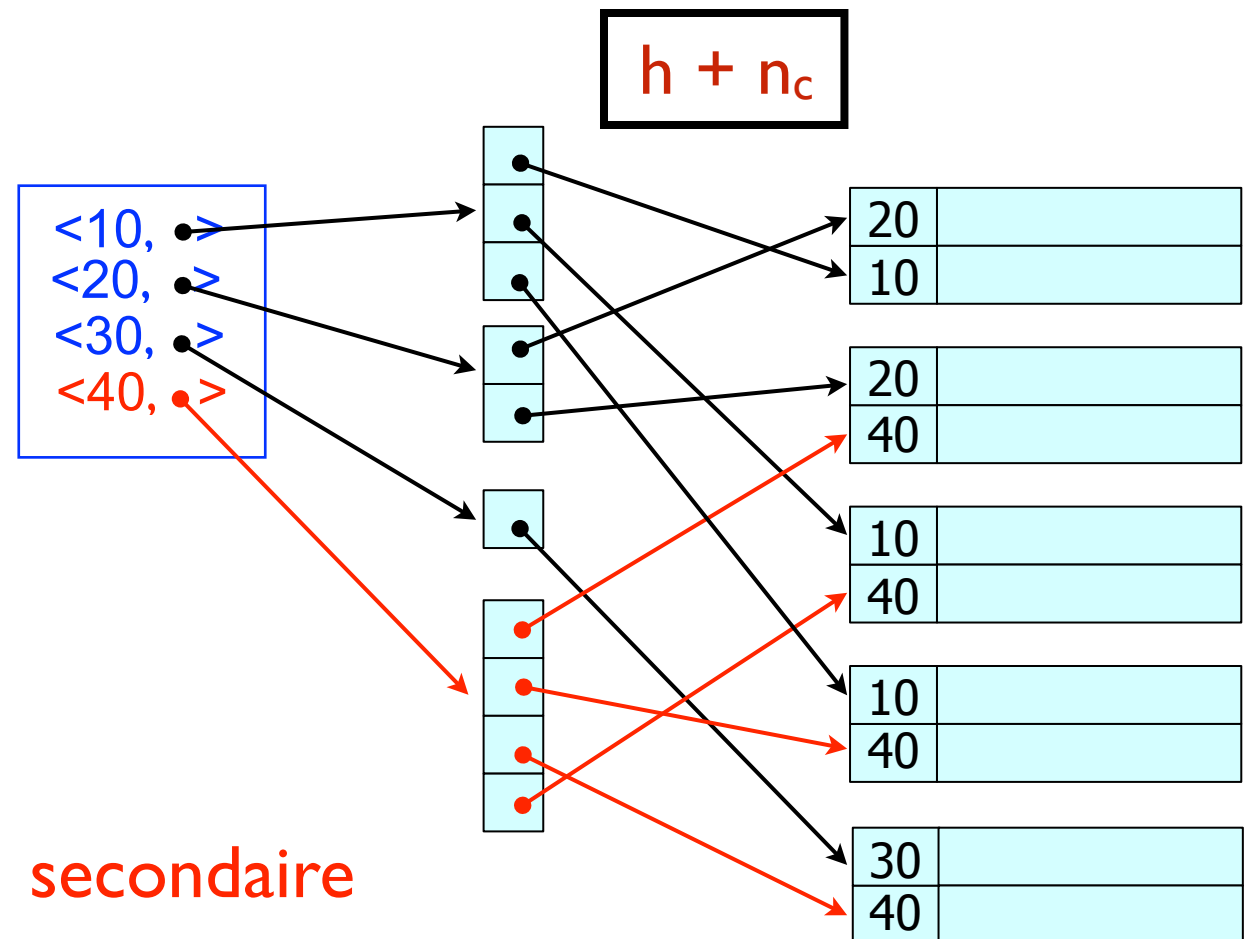
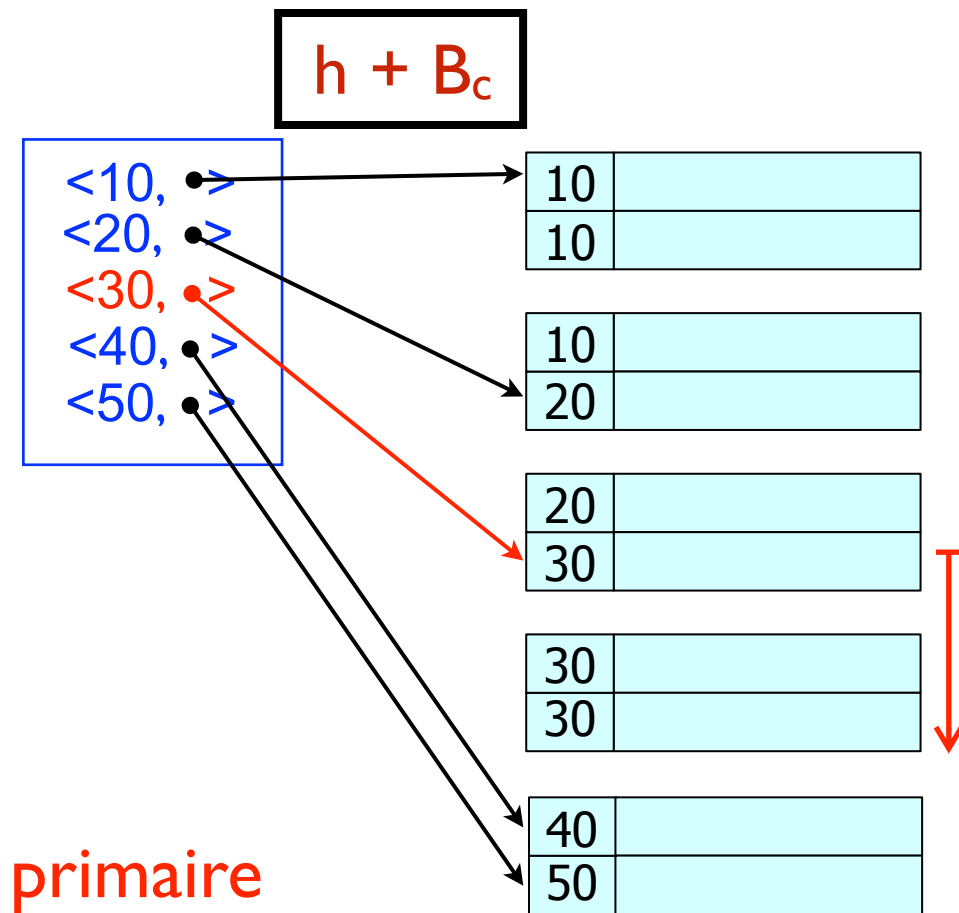
$c = 40$



Recherche par index

Opération : recherche des enregistrements avec clef de recherche = c

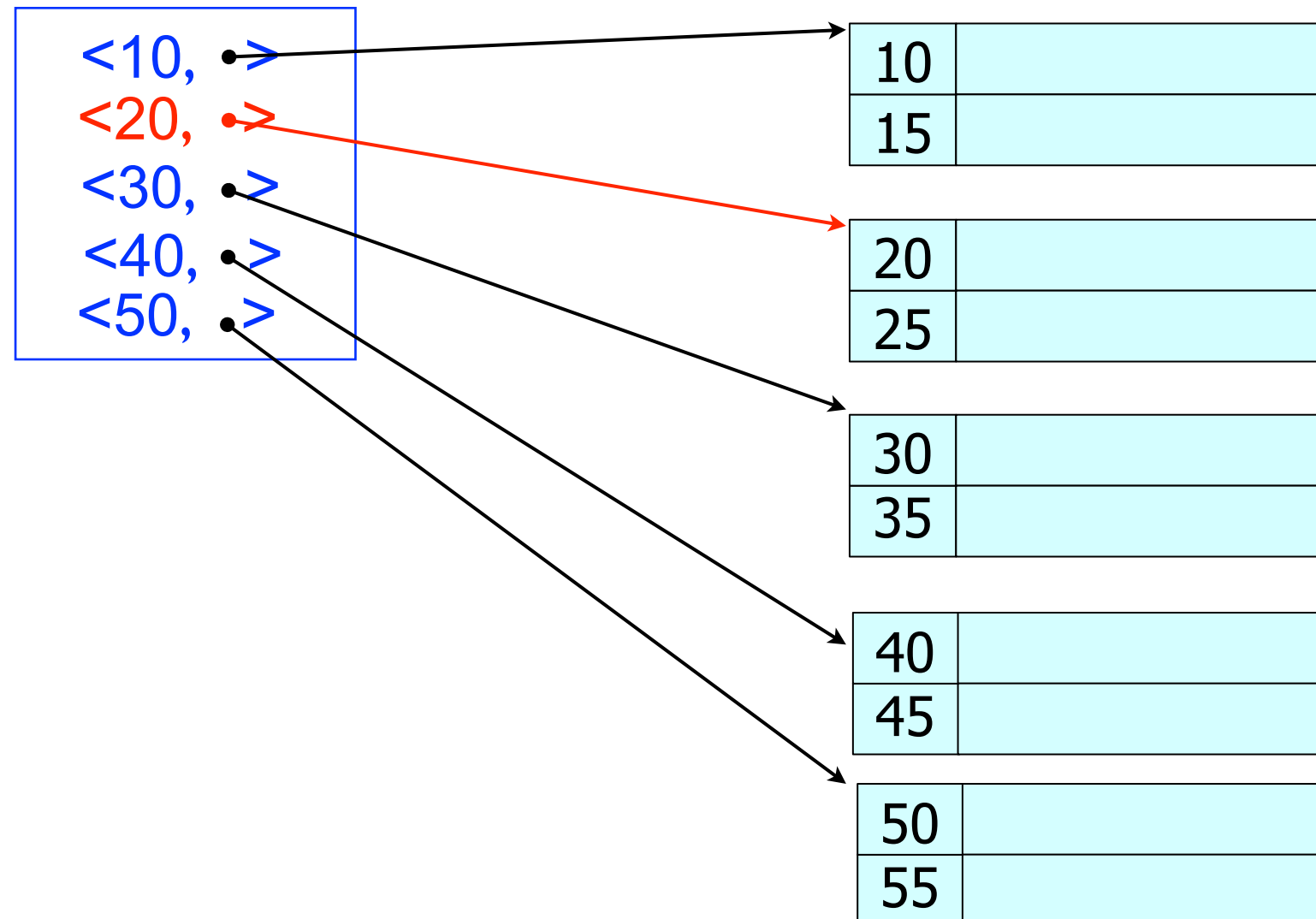
- Index dense - cas général : doublons possibles
- **Coût** : soit
 - ▶ **h** le cout de recherche de c dans l'index
 - ▶ Depend de l'implementation de l'index (cf plus loin)
 - ▶ Idéalement “proche” de constant en pratique
 - ▶ **n_c** le nombre d'enregistrement avec clef c
 - ▶ **B_c** le nombre de blocs occupés par n_c enregistrements



Recherche par index

- **Opération** : recherche des enregistrements avec clef de recherche = c
- **Index non-dense** : le cas sans doublons d'abord

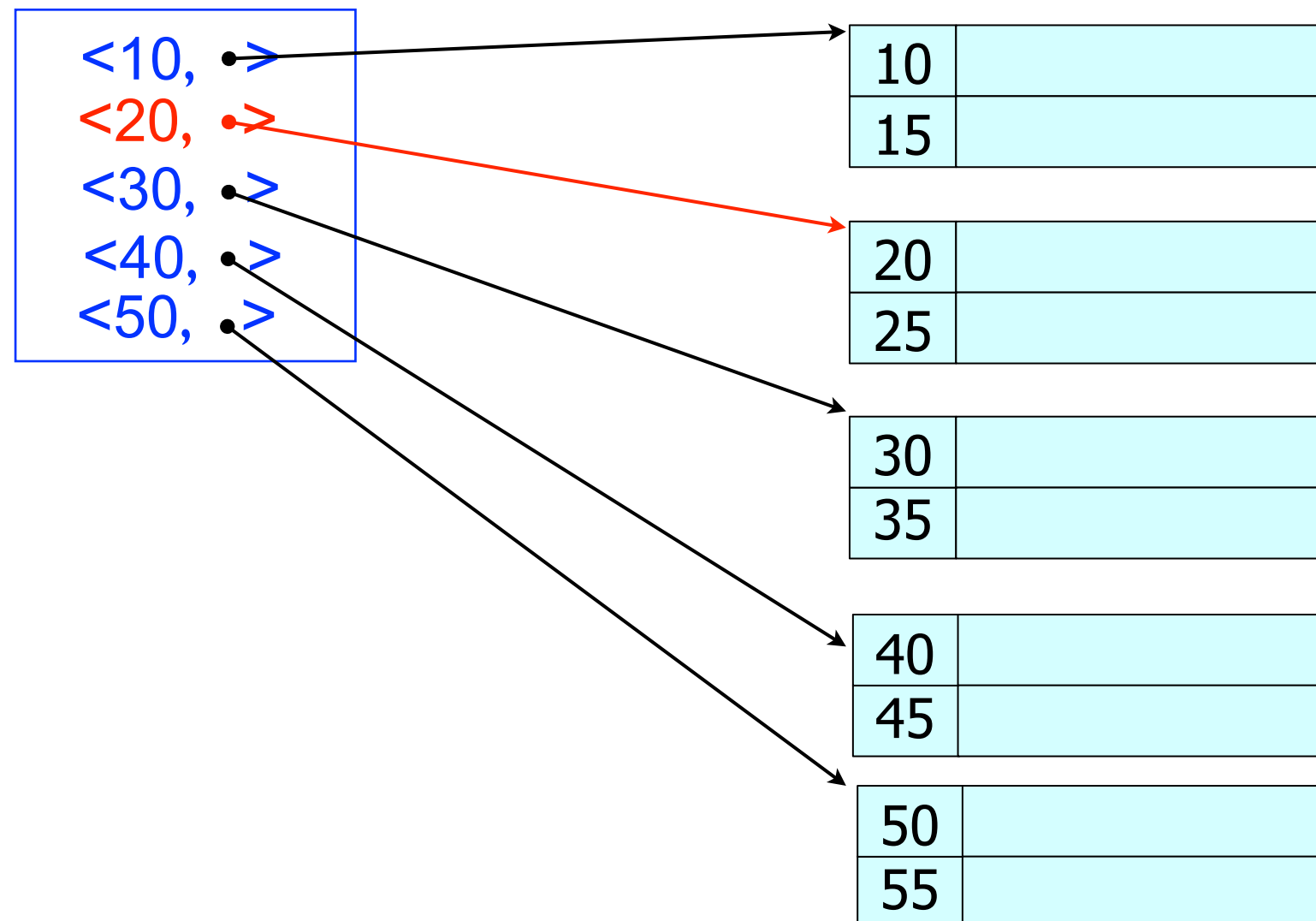
c = 25



Recherche par index

- **Opération** : recherche des enregistrements avec clef de recherche = c
- **Index non-dense** : le cas sans doublons d'abord
 - ▶ chercher dans l'index la plus grande clef $\leq c$
 - ▶ si la recherche est positive, soit $\langle c', p' \rangle$ l'entrée trouvée
 - ▶ parcourir le bloc p'

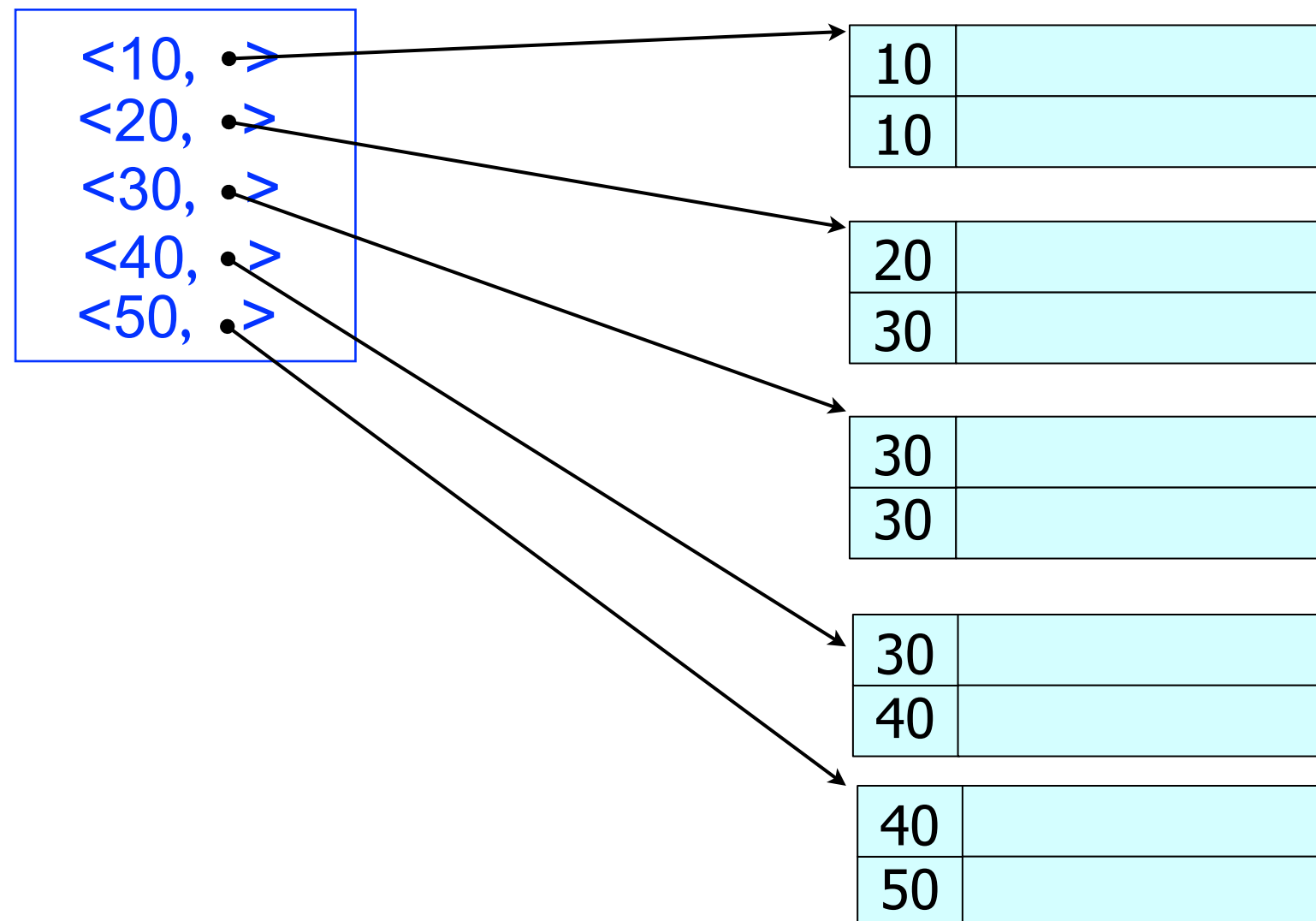
$c = 25$



Recherche par index

- **Opération** : recherche des enregistrements avec clef de recherche = c
- **Index non-dense** - avec doublons

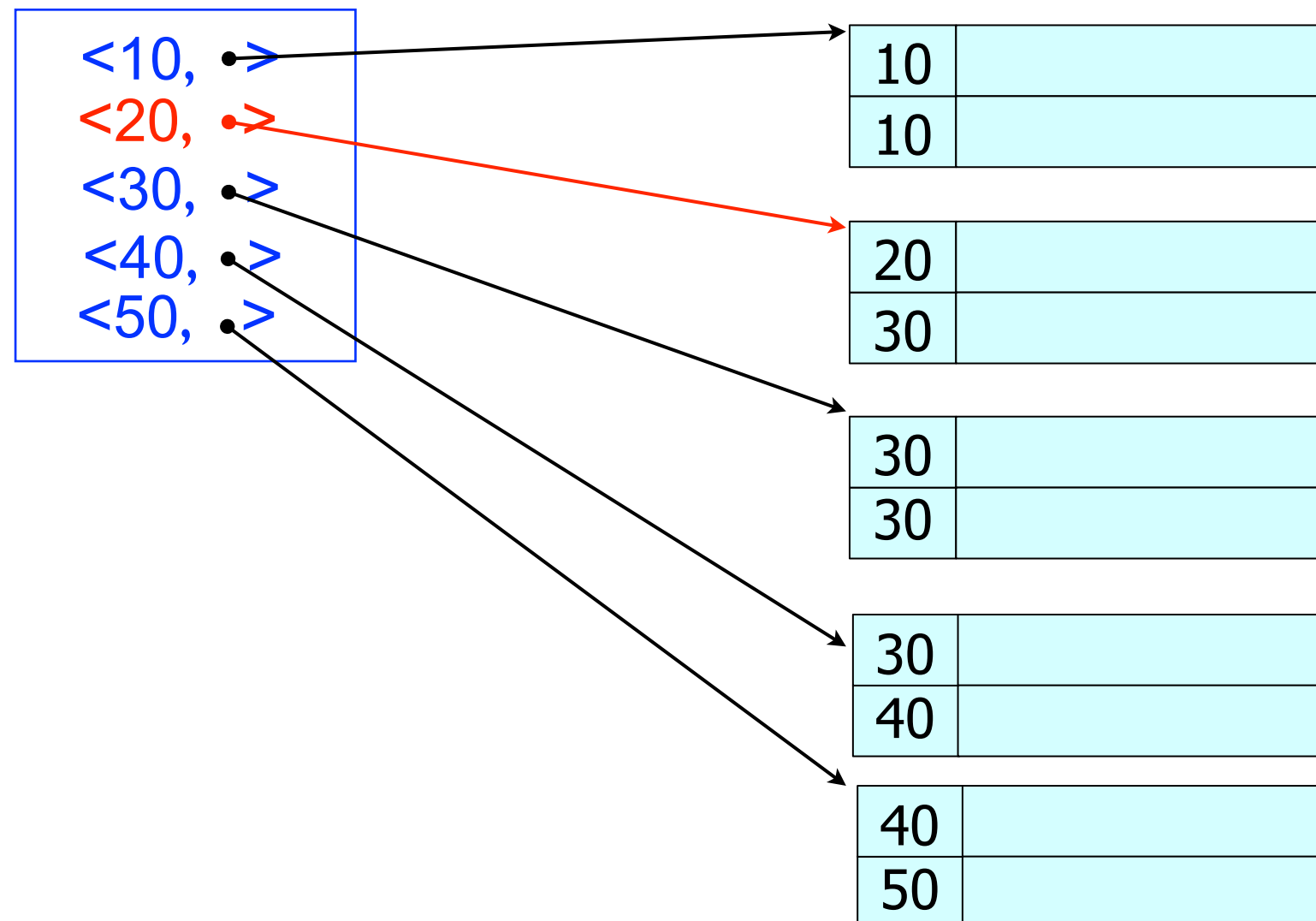
c = 30



Recherche par index

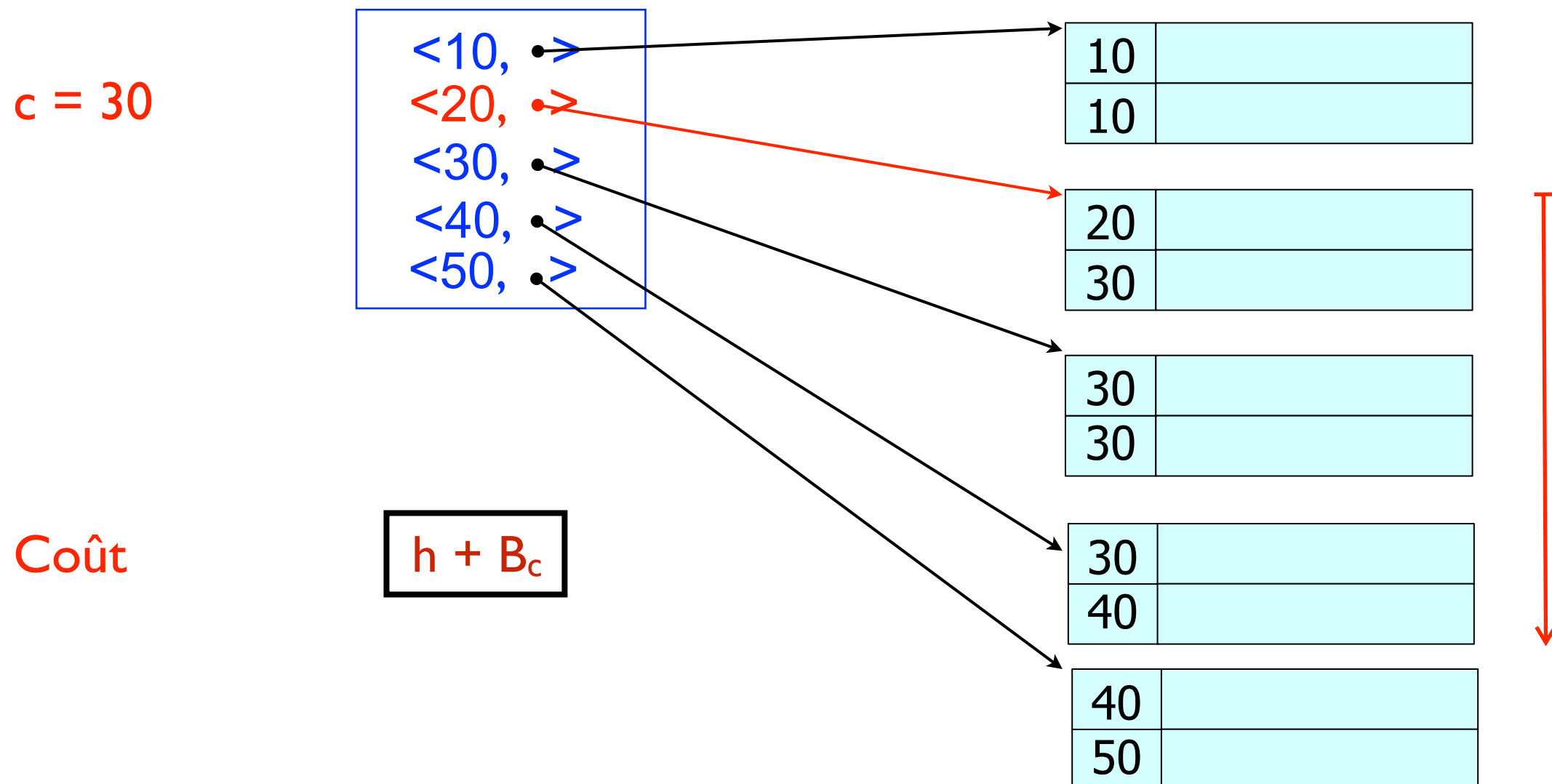
- **Opération** : recherche des enregistrements avec clef de recherche = c
- **Index non-dense - avec doublons**
 - ▶ chercher dans l'index la plus grande clef $< c$ (ou si elle n'existe pas, chercher c)
 - ▶ si la recherche est positive, soit $<c', p'\rangle$ l'entrée trouvée
 - ▶ à partir du bloc p' parcourir tous les enregistrements de clef c

$c = 30$



Recherche par index

- **Opération** : recherche des enregistrements avec clef de recherche = c
- **Index non-dense - avec doublons**
 - ▶ chercher dans l'index la plus grande clef $< c$ (ou si elle n'existe pas, chercher c)
 - ▶ si la recherche est positive, soit $< c', p' >$ l'entrée trouvée
 - ▶ à partir du bloc p' parcourir tous les enregistrements de clef c



Mise à jour de l'index

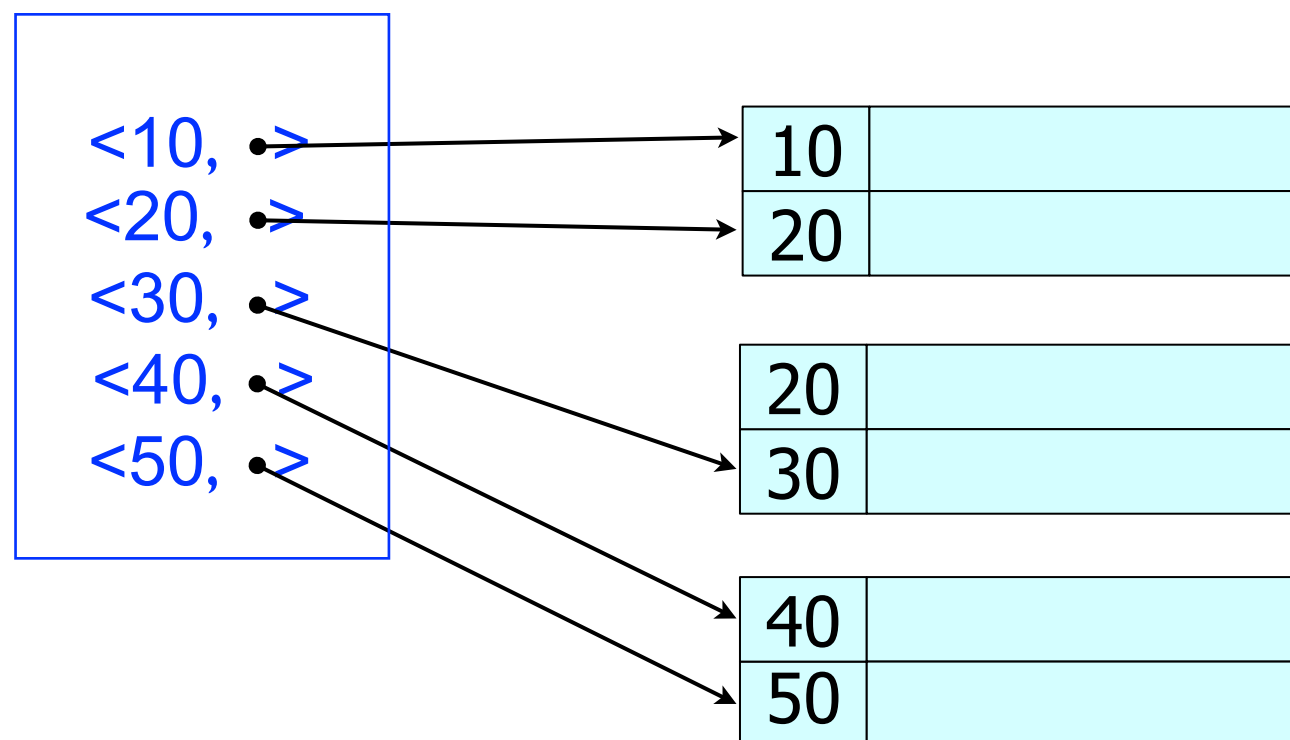
- Insertion d'un nouvel enregistrement avec clef de recherche c :
 - ▶ insertion dans le fichier de données à l'adresse p avec les techniques vues
 - ▶ mise à jour de l'index
 - insertion de $\langle c, p \rangle$ si besoin
 - mise à jour des pointeurs de l'index si les enregistrements pointés changent de bloc
 - techniques légèrement différentes selon que l'index est dense/non-dense, primaire/secondaire doublons/clefs distinctes, ...
- Suppression d'un enregistrement :
 - ▶ suppression de l'enregistrement du fichier de données avec les techniques vues
 - ▶ mise à jour de l'index : duale à l'insertion, techniques similaires

Pas au programme >>

Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

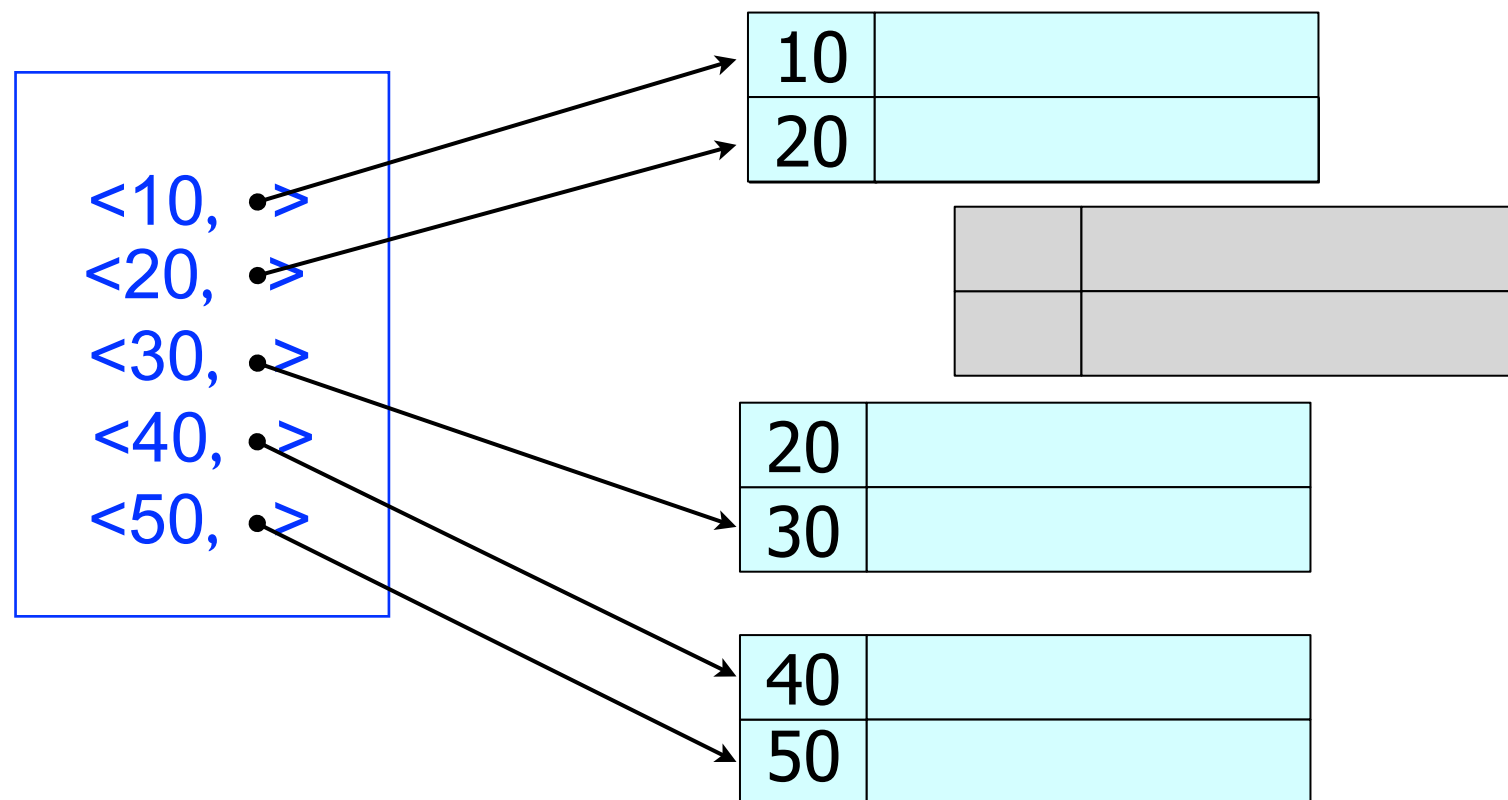
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

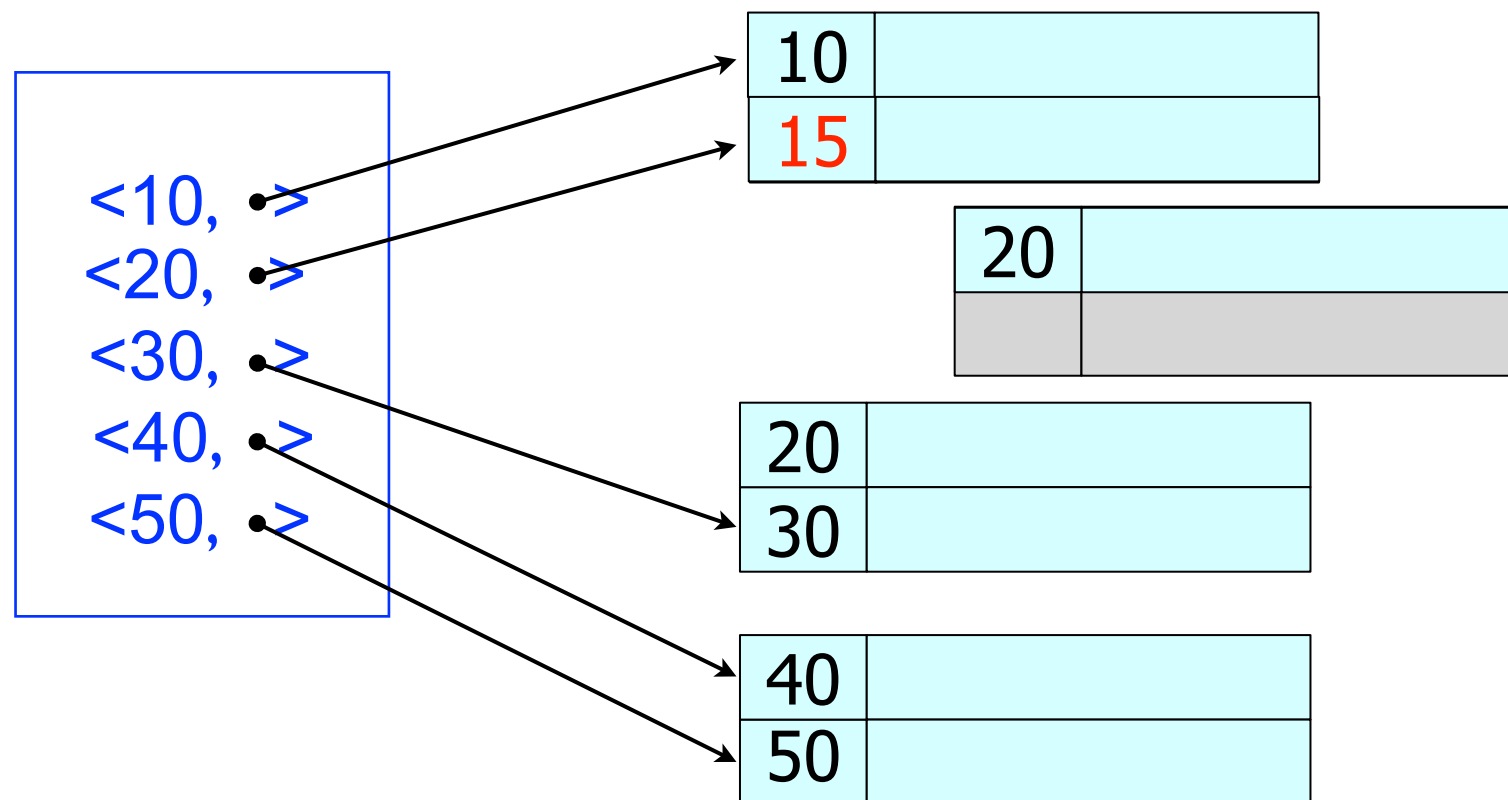
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

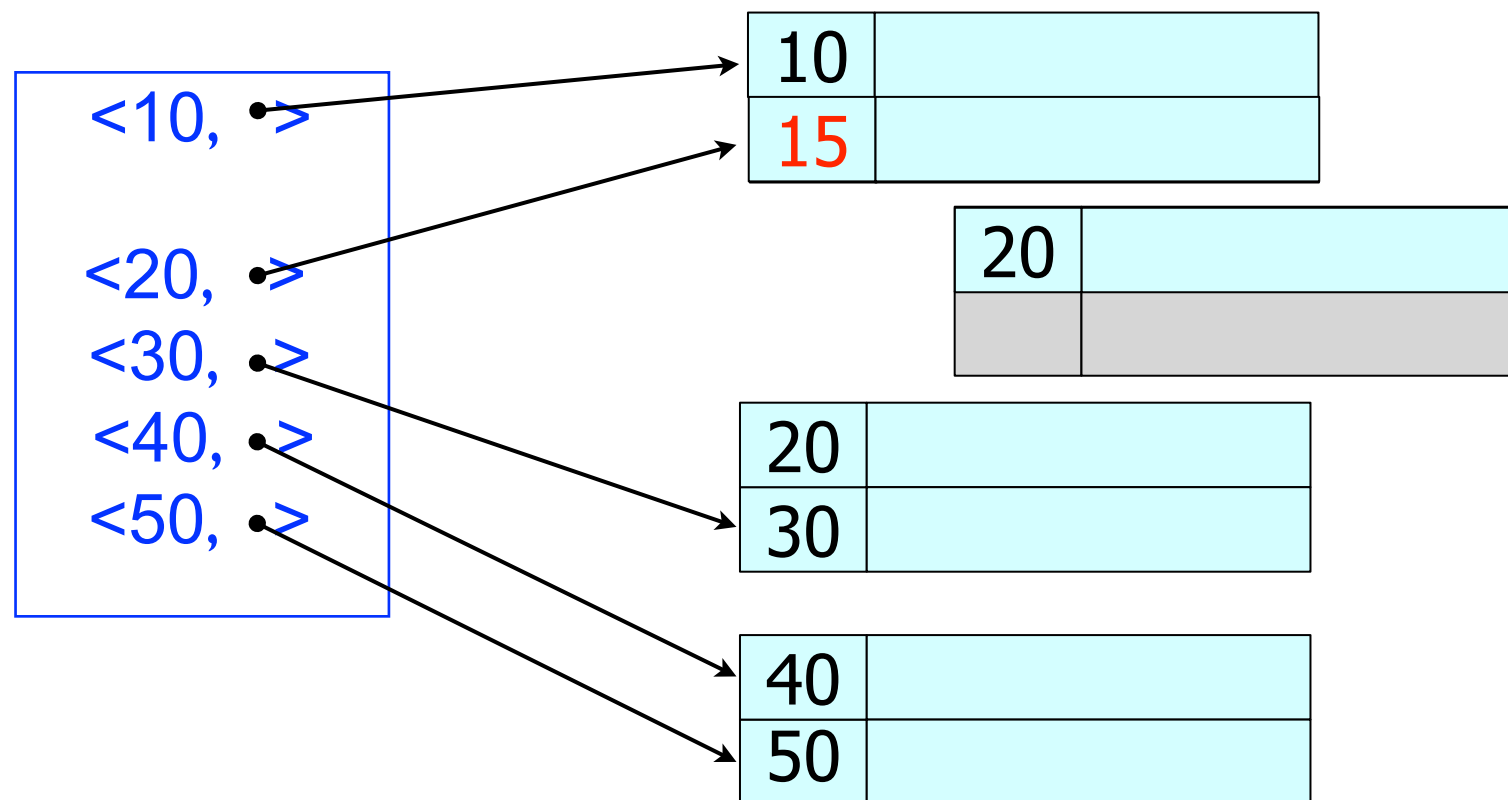
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

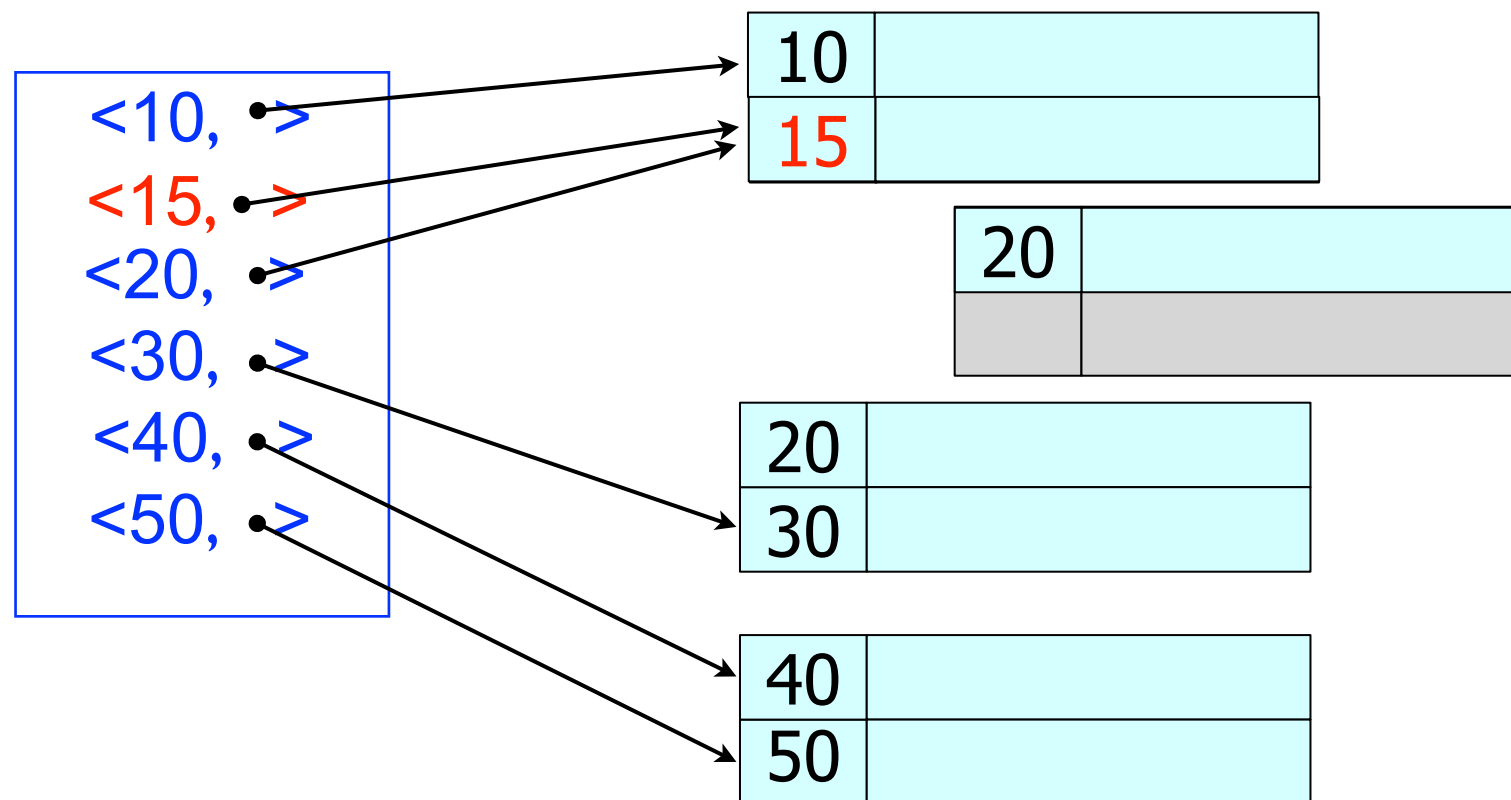
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

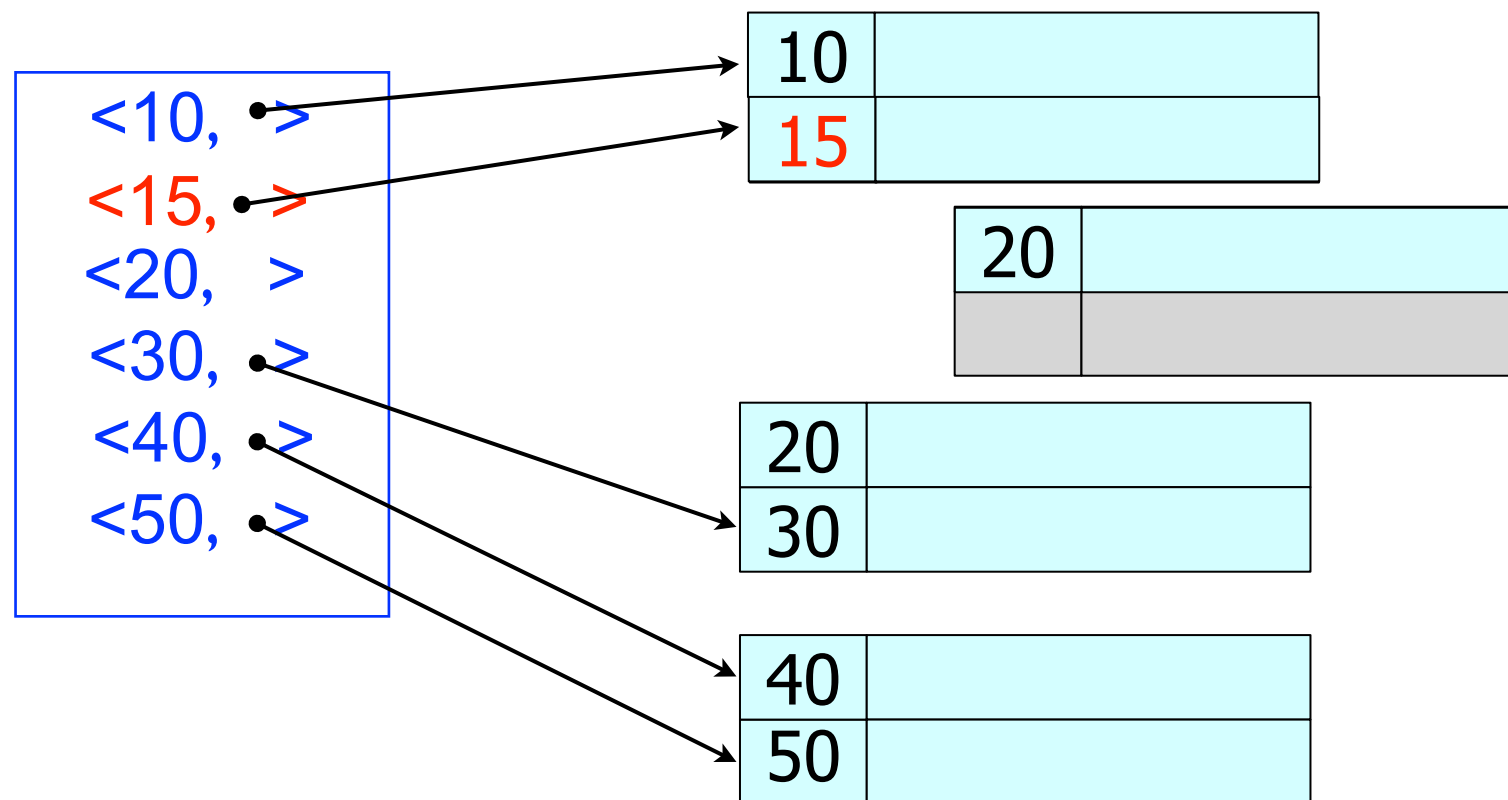
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

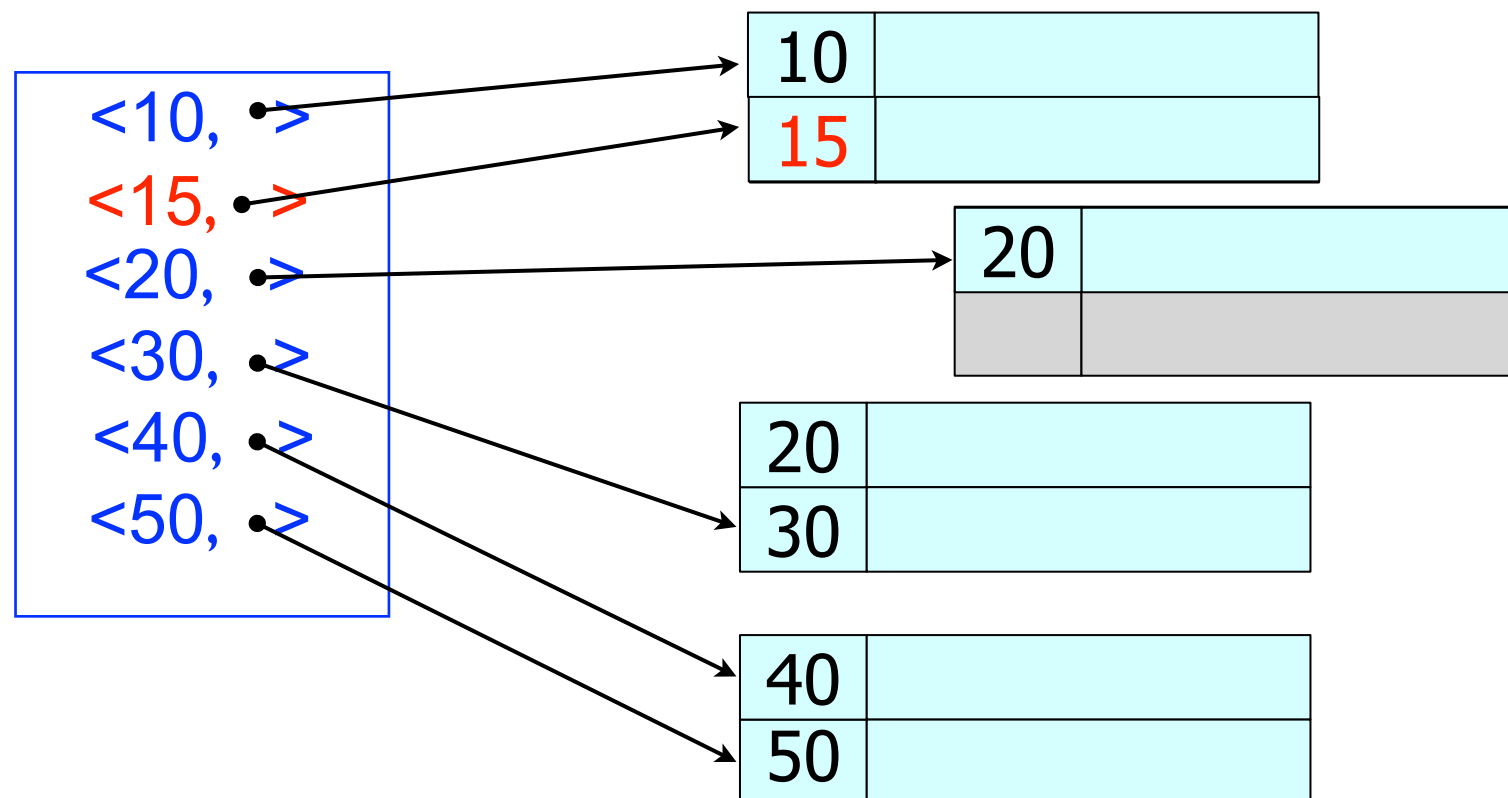
Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

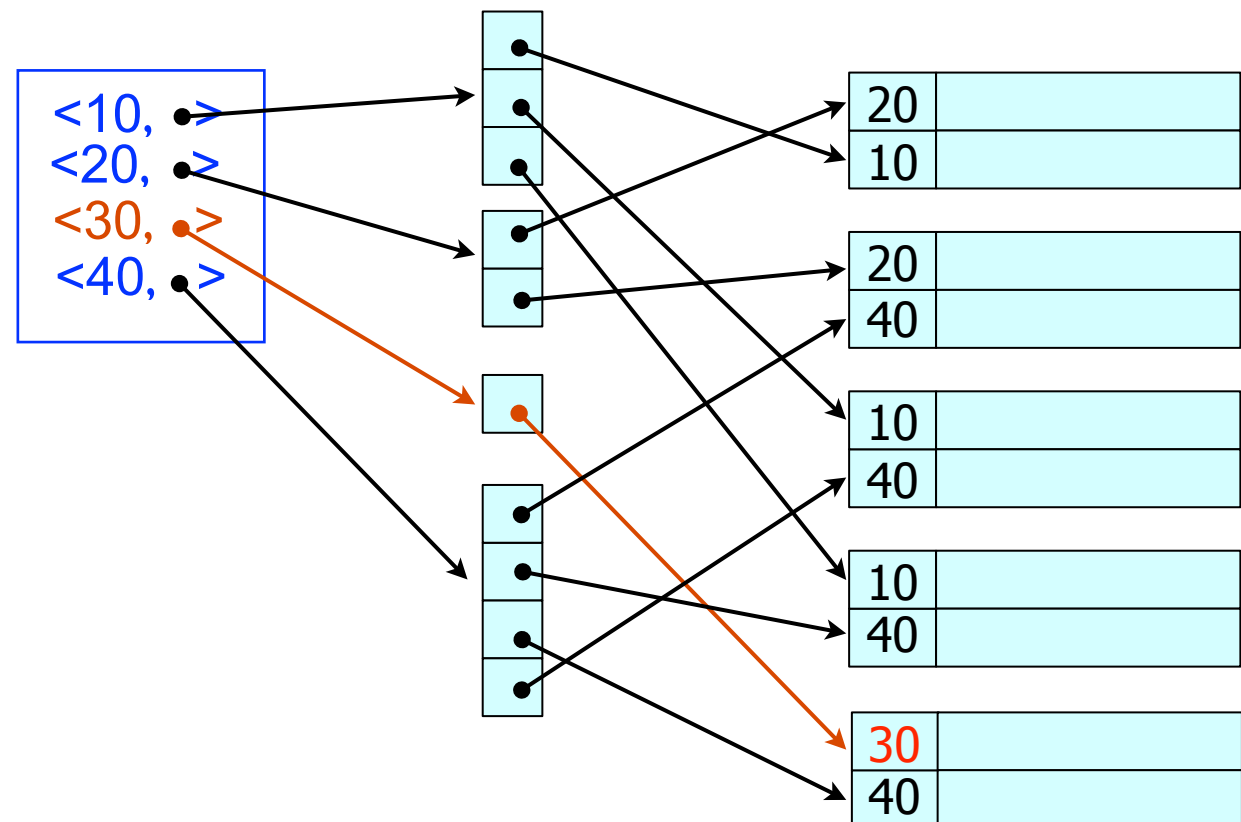
- Index dense primaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé insérer $\langle c, p \rangle$ dans l'index
 - ▶ en cas de réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index

Ex. insertion d'un enregistrement avec clef de recherche 15



Mise à jour de l'index : insertion

- Index (dense) secondaire :
 - ▶ chercher c dans l'index
 - ▶ si c n'est pas trouvé,
 - créer un nouveau *bucket* contenant p
 - insérer c avec un pointeur au *bucket* dans l'index
 - ▶ sinon, c est déjà dans l'index, alors ajouter p à son *bucket*
 - ▶ en cas de fichier séquentiel et réorganisation immédiate : si un enregistrement pointé par l'index change de bloc, mettre à jour son pointeur dans l'index



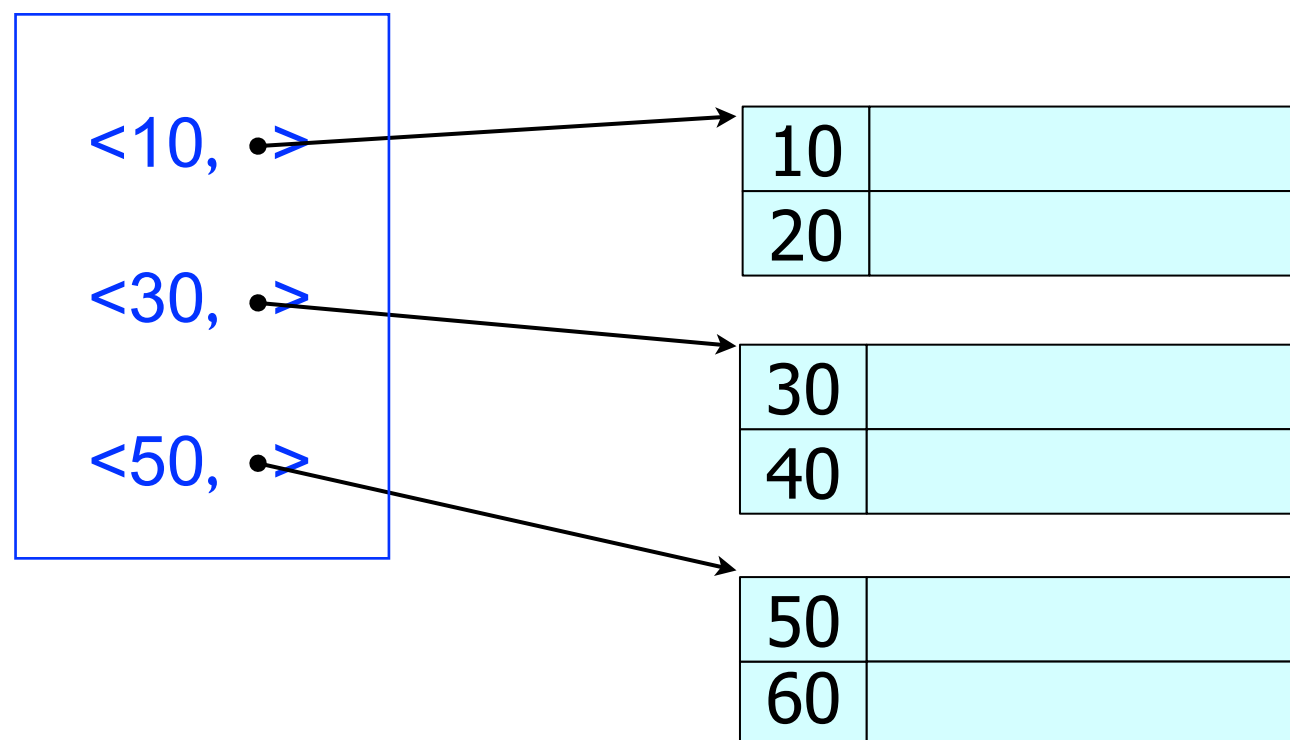
Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)

1. si l'insertion de l'enregistrement détermine la création d'un nouveau bloc ordinaire

- insérer $\langle c', p' \rangle$ dans l'index,
 p' : pointeur au nouveau bloc, c' : première clef du nouveau bloc

Ex. insertion d'un enregistrement avec clef de recherche 15



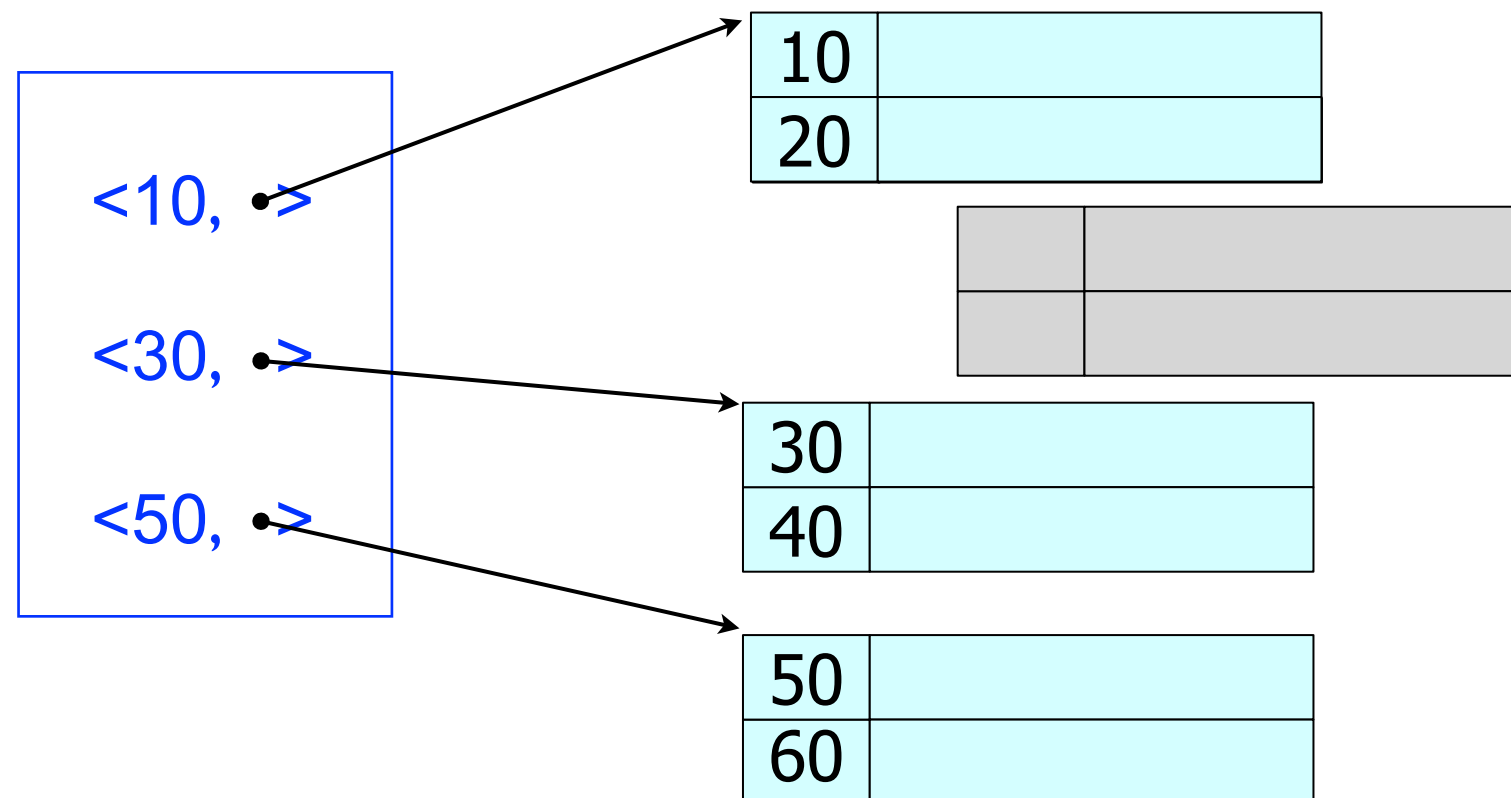
Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)

1. si l'insertion de l'enregistrement détermine la création d'un nouveau bloc ordinaire

- insérer $\langle c', p' \rangle$ dans l'index,
 p' : pointeur au nouveau bloc, c' : première clef du nouveau bloc

Ex. insertion d'un enregistrement avec clef de recherche 15



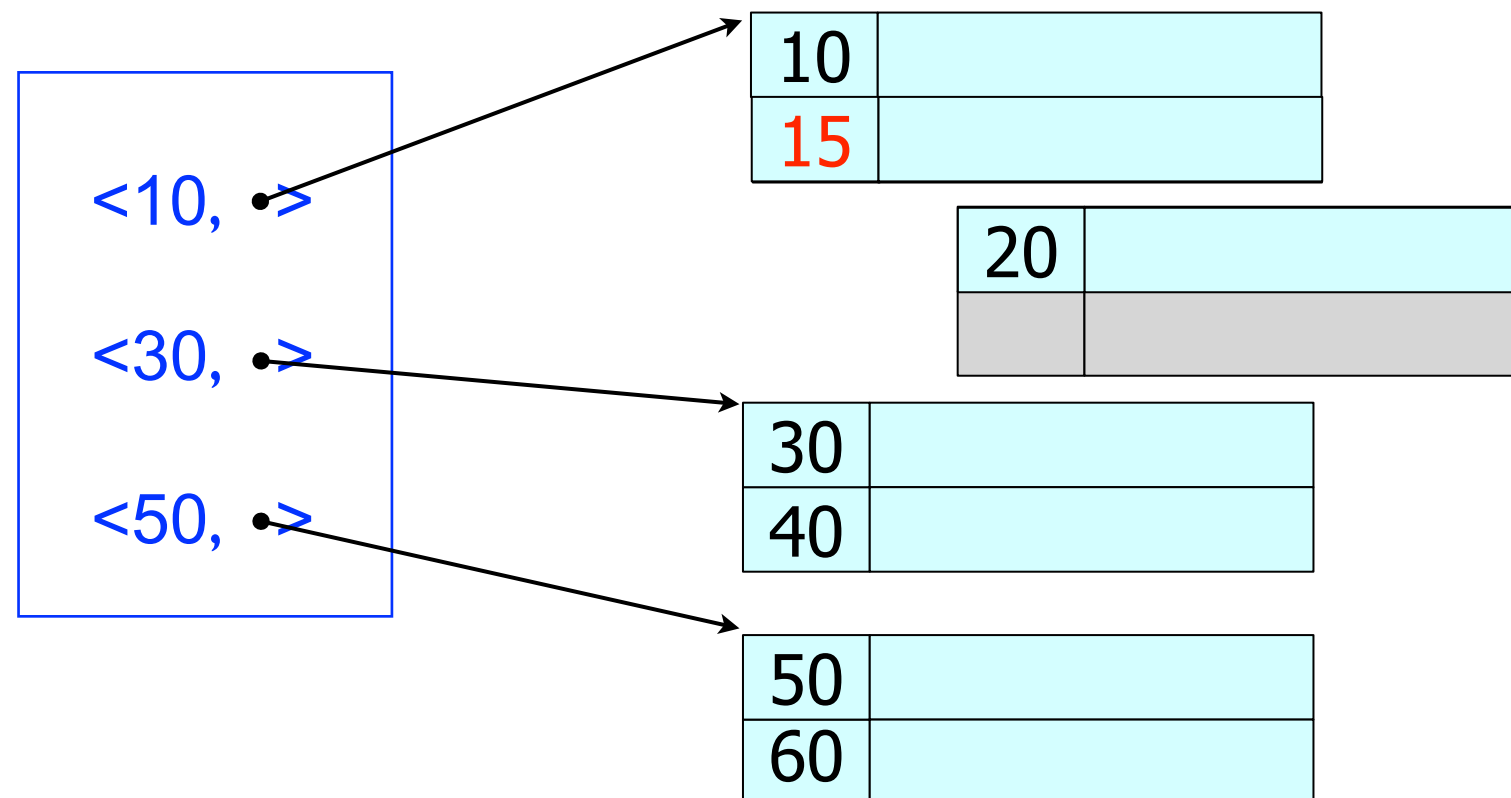
Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)

1. si l'insertion de l'enregistrement détermine la création d'un nouveau bloc ordinaire

- insérer $\langle c', p' \rangle$ dans l'index,
 p' : pointeur au nouveau bloc, c' : première clef du nouveau bloc

Ex. insertion d'un enregistrement avec clef de recherche 15



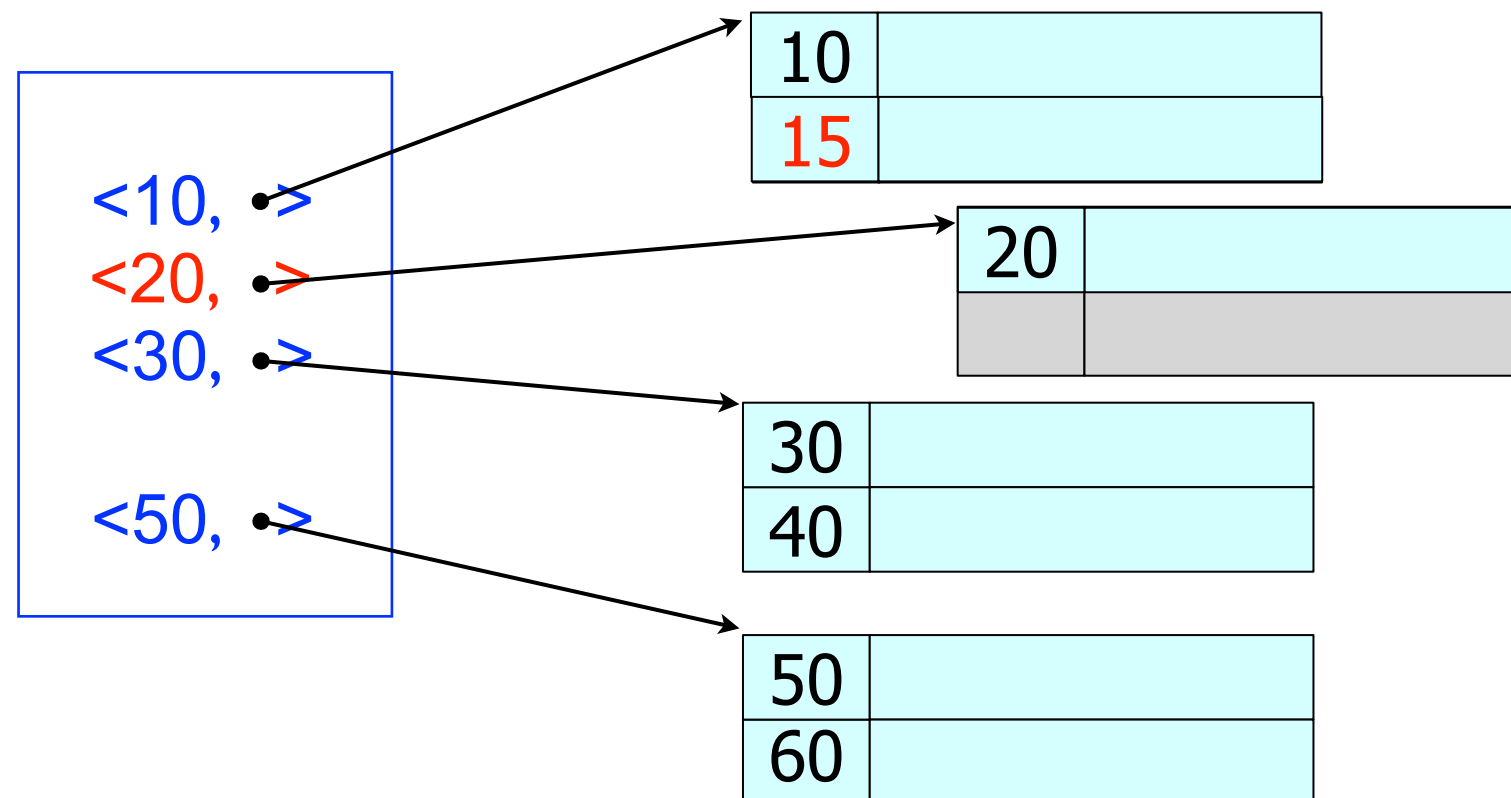
Mise à jour de l'index : insertion

- Index non-dense (sans doublons) - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)

1. si l'insertion de l'enregistrement détermine la création d'un nouveau bloc ordinaire

- insérer $\langle c', p' \rangle$ dans l'index,
 p' : pointeur au nouveau bloc, c' : première clef du nouveau bloc

Ex. insertion d'un enregistrement avec clef de recherche 15



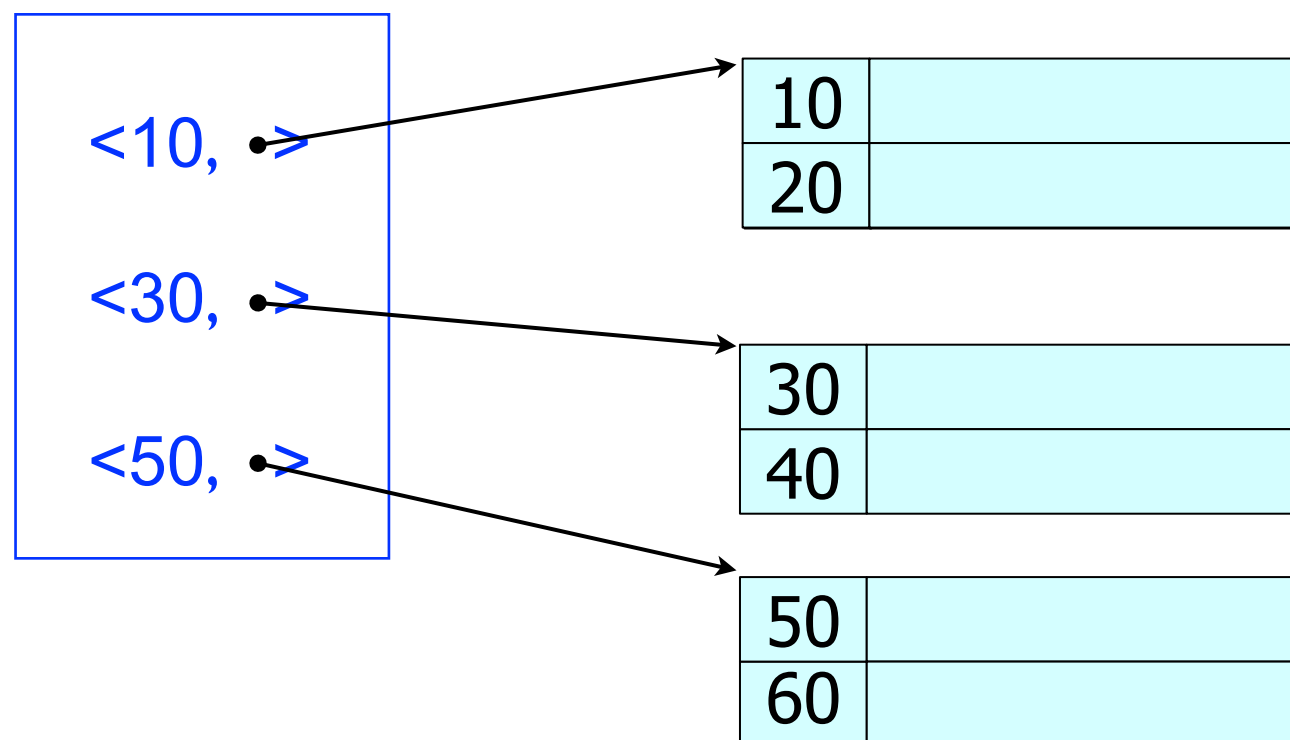
Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
- 2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
 - modifier la clef correspondante dans l'index

Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

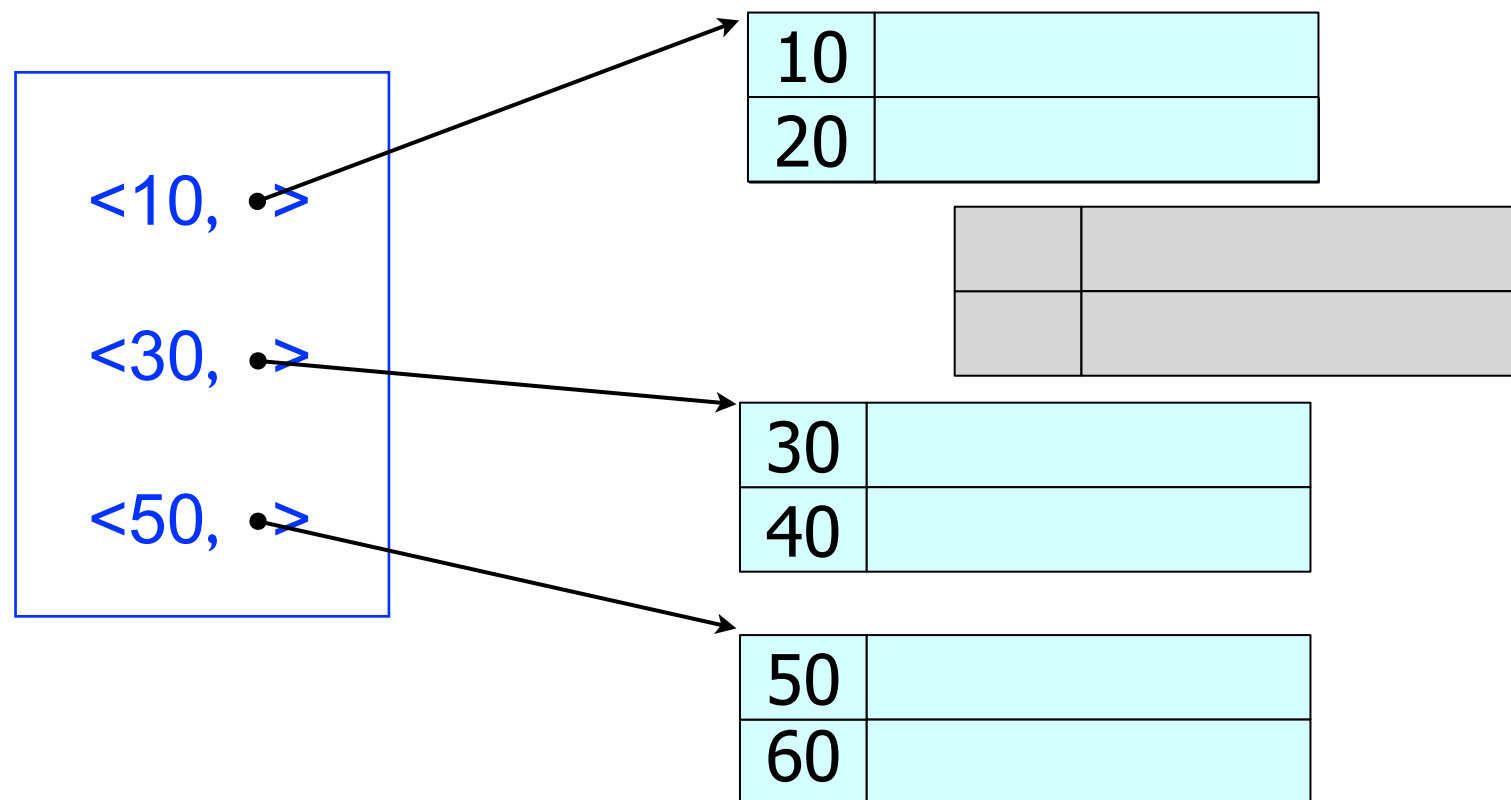
Ex 1. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

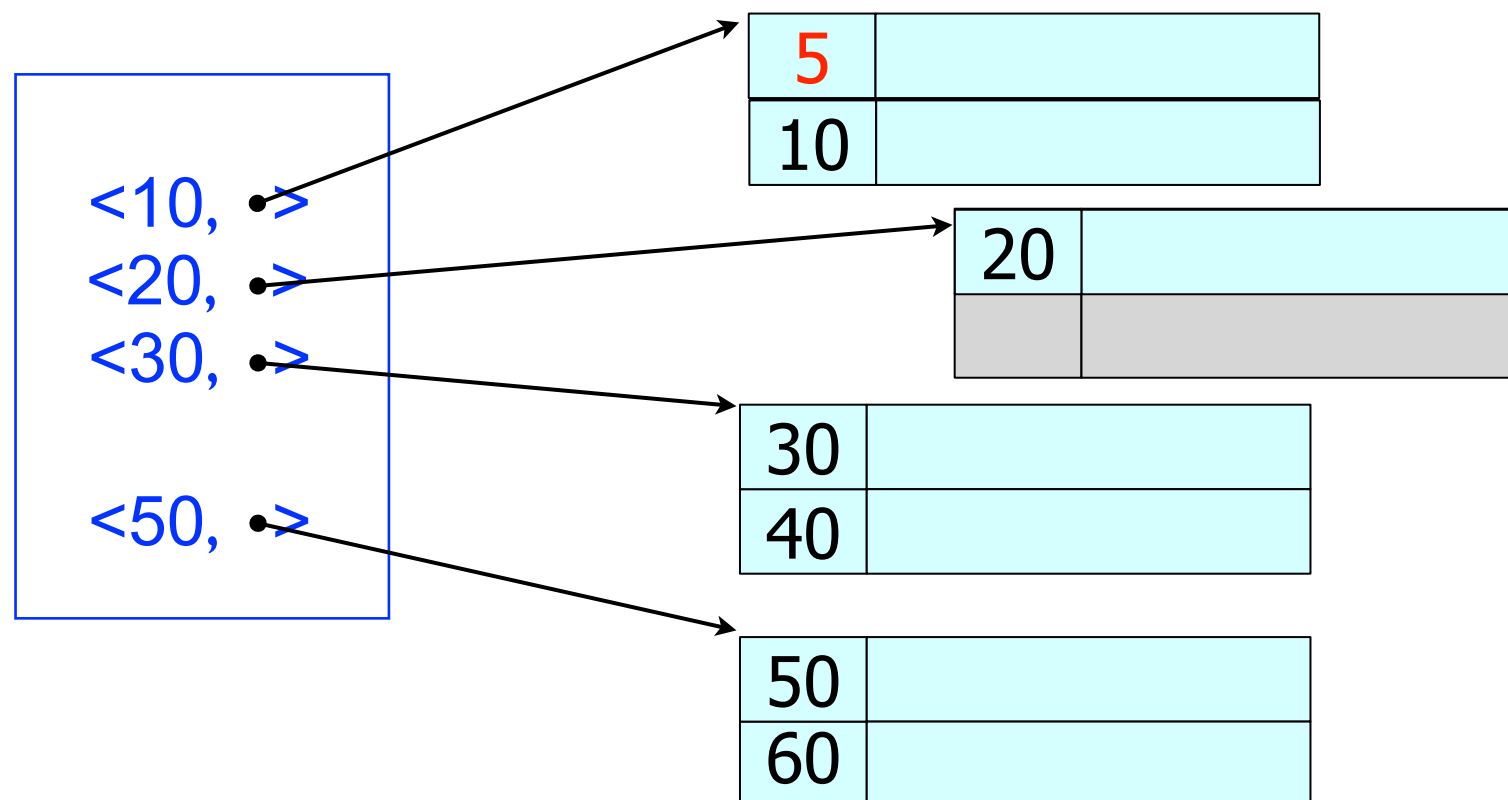
Ex 1. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

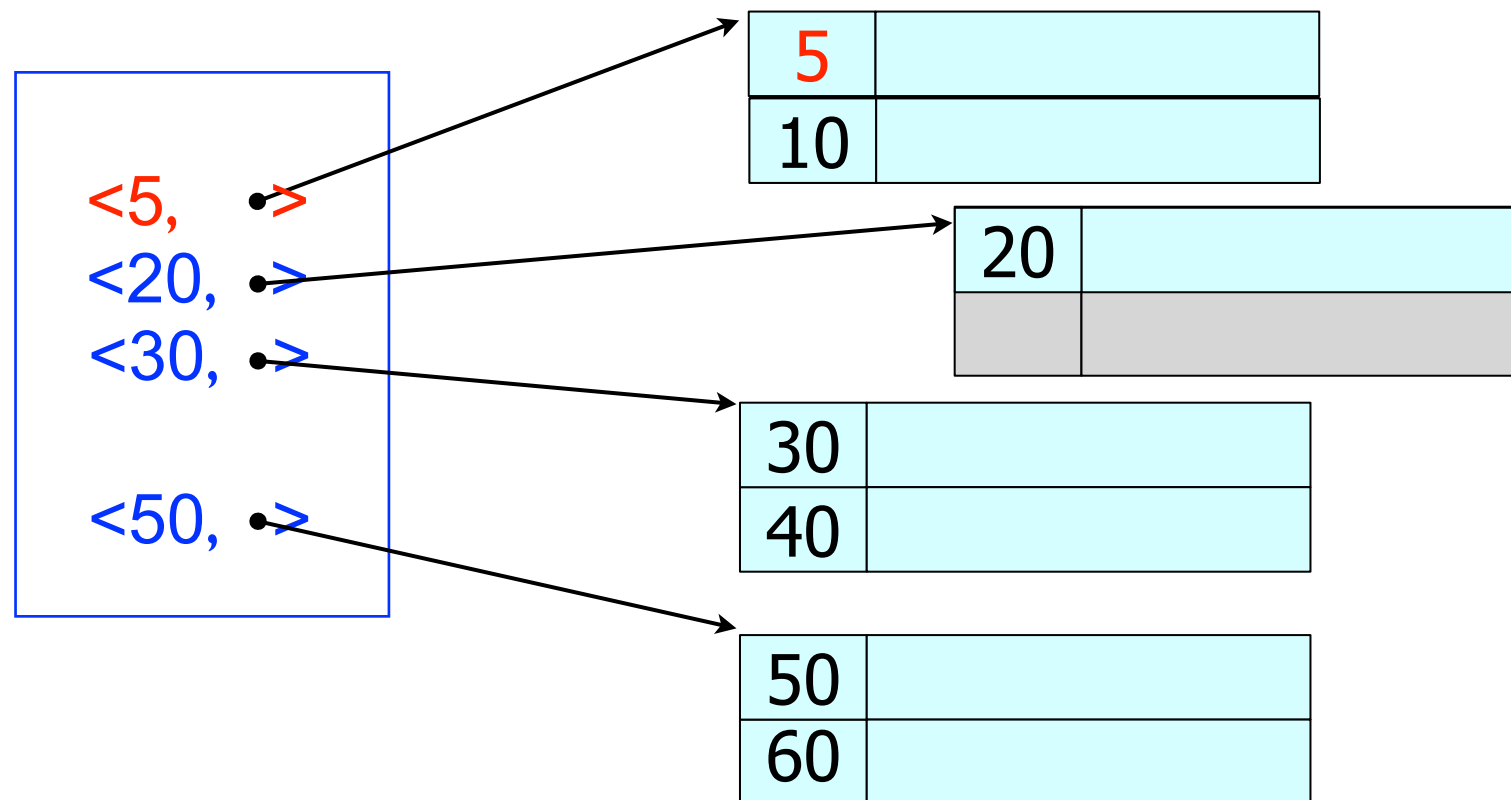
Ex 1. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

Ex 1. insertion d'un enregistrement avec clef de recherche 5



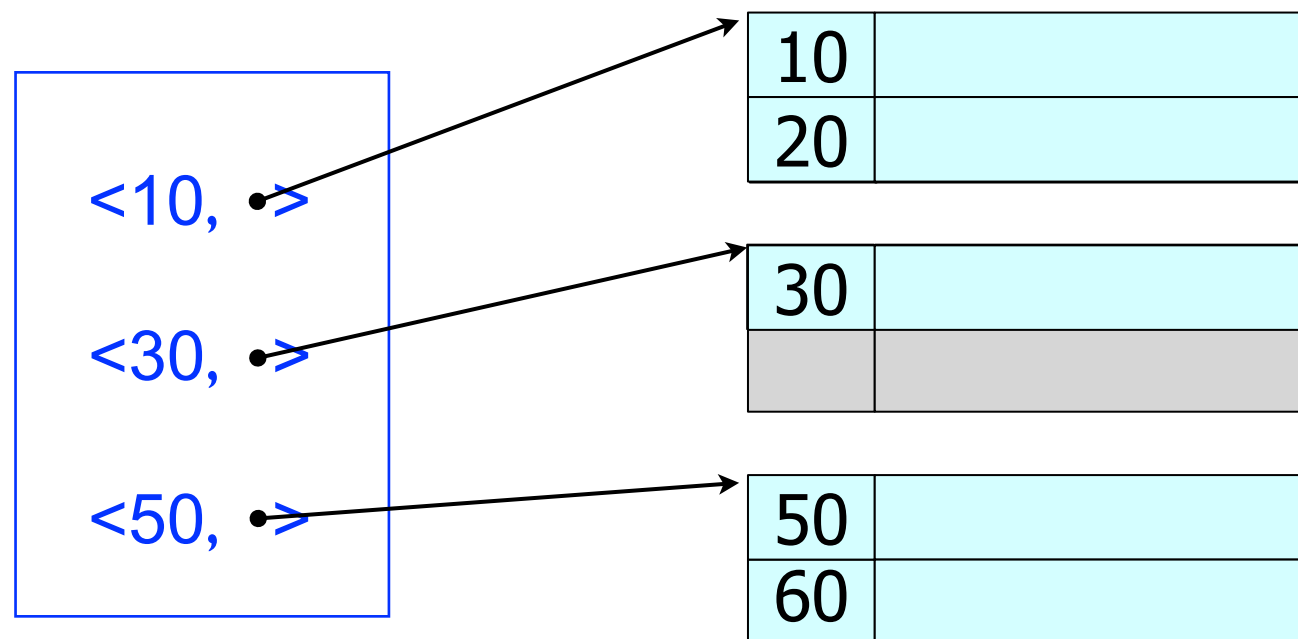
Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
- 2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
 - modifier la clef correspondante dans l'index

Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

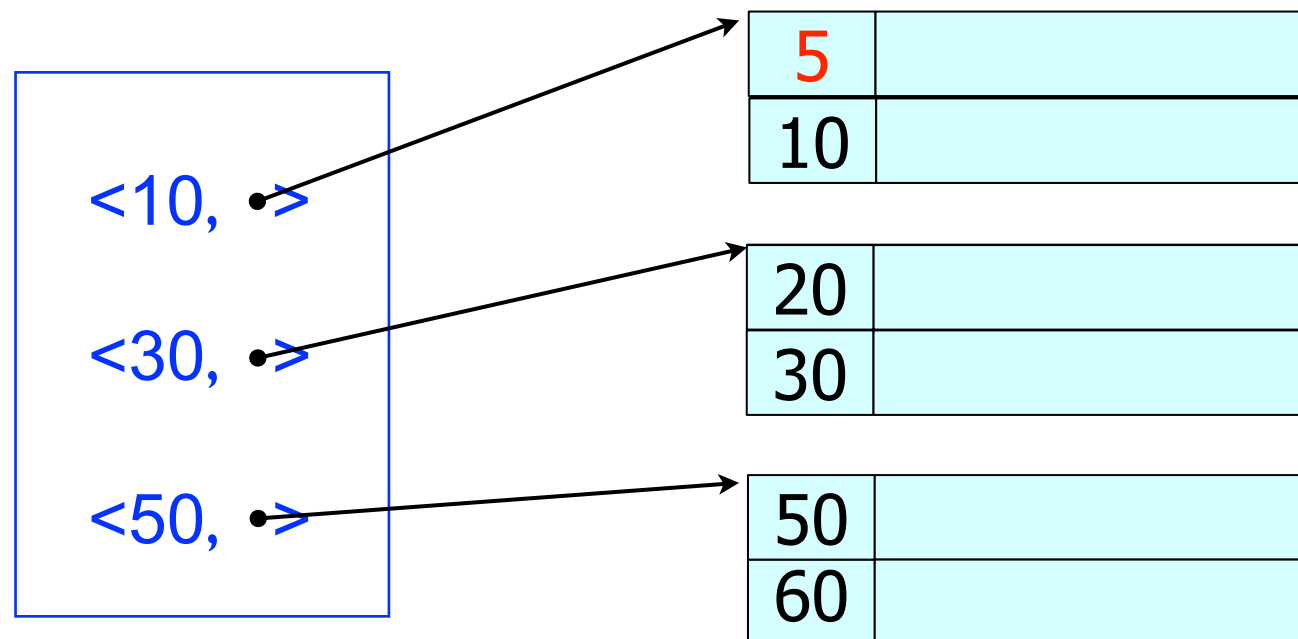
Ex 2. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

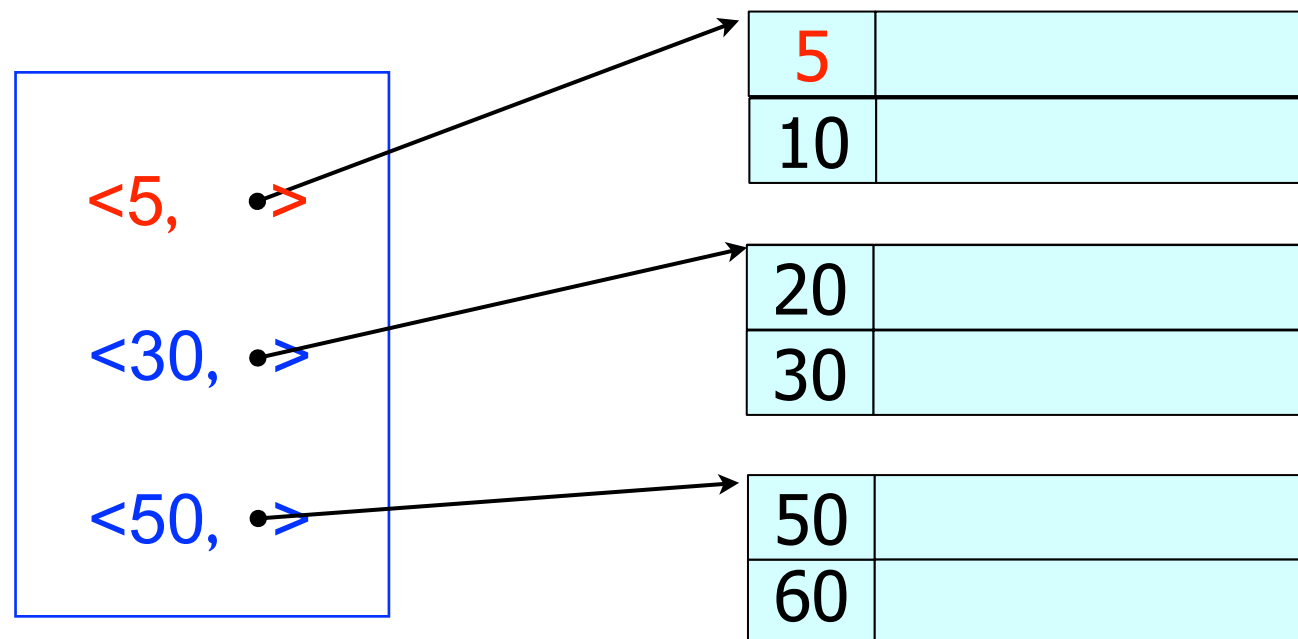
Ex 2. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

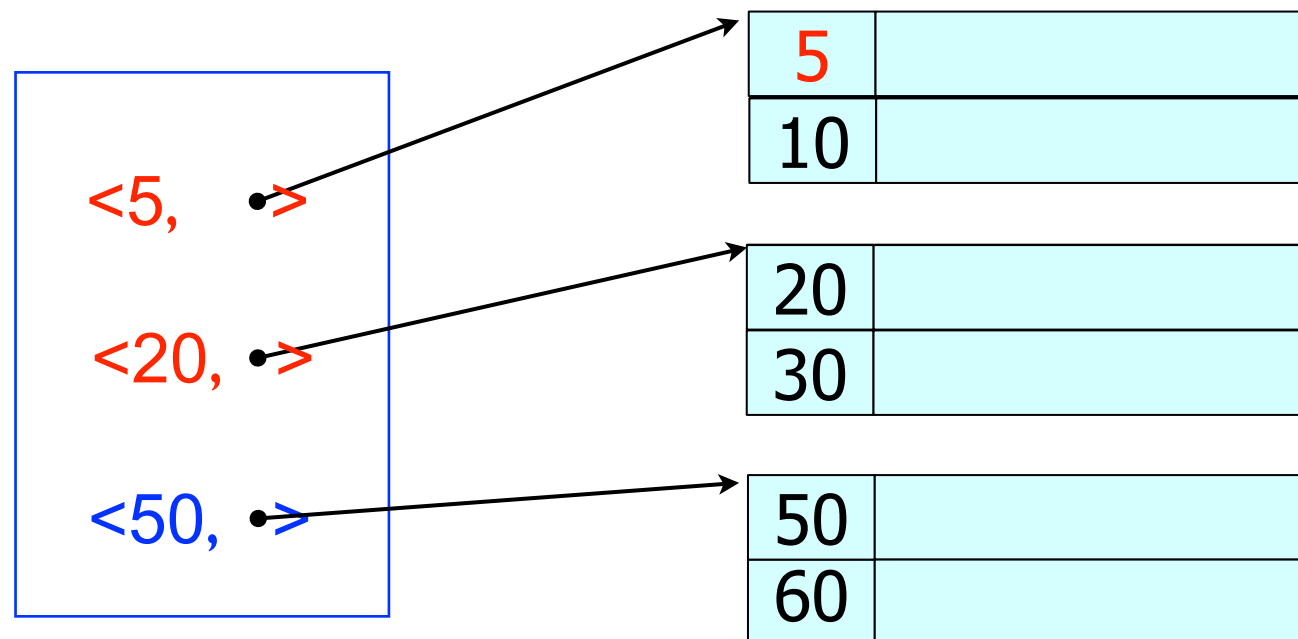
Ex 2. insertion d'un enregistrement avec clef de recherche 5



Mise à jour de l'index : insertion

- **Index non-dense (sans doublons)** - souvent pas de modification de l'index (cas avec doublons légèrement plus complexe mais similaire)
2. si l'insertion de l'enregistrement modifie la première clef d'un bloc existant
- modifier la clef correspondante dans l'index

Ex 2. insertion d'un enregistrement avec clef de recherche 5



Création d'index

- Le SGBD crée automatiquement un index sur la clef primaire de chaque table
 - ▶ raison : accès fréquent par clef primaire
- On peut demander explicitement la création d'index sur d'autres clefs de recherche avec la commande (spécifique-SGBD) :

```
CREATE INDEX ON maTable (att1, ...,attk);
```

Ou

```
CREATE UNIQUE INDEX ON maTable (att1, ...,attk);
```

Pour indiquer que `(att1,...,attk)` est une clef candidate de la table (absence de doublons)

- ▶ pas nécessaire si la contrainte UNIQUE est associée à `(att1,..., attk)`

Création d'index

- Avantage de la création d'index supplémentaires :
 - ▶ efficacité des requêtes qui accèdent à la table par une clef de recherche sur laquelle un index est présent
- Inconvénient
 - ▶ Mises à jour de la table plus lourdes : mises à jour des index
- Les avantages sont en général plus importants
 - ▶ sauf en cas de mises à jour très fréquentes

Implementation des index

- Différentes structures de données sont adaptées à l'implémentation d'index
- Les plus fréquentes :
 - ▶ B+-tree, B-tree
 - ▶ Hash
 - ▶ Bitmap
- D'autres types d'index :
 - ▶ recherches spécifiques (e.g recherche *full-text*),
 - ▶ spécifique au SGBD
- La plupart des SGBD permettent de choisir l'implémentation de l'index au moment de la création :

```
CREATE INDEX ON maTable USING BTREE(att1, ..., attk);
```