Méthodes de stockage et d'accès

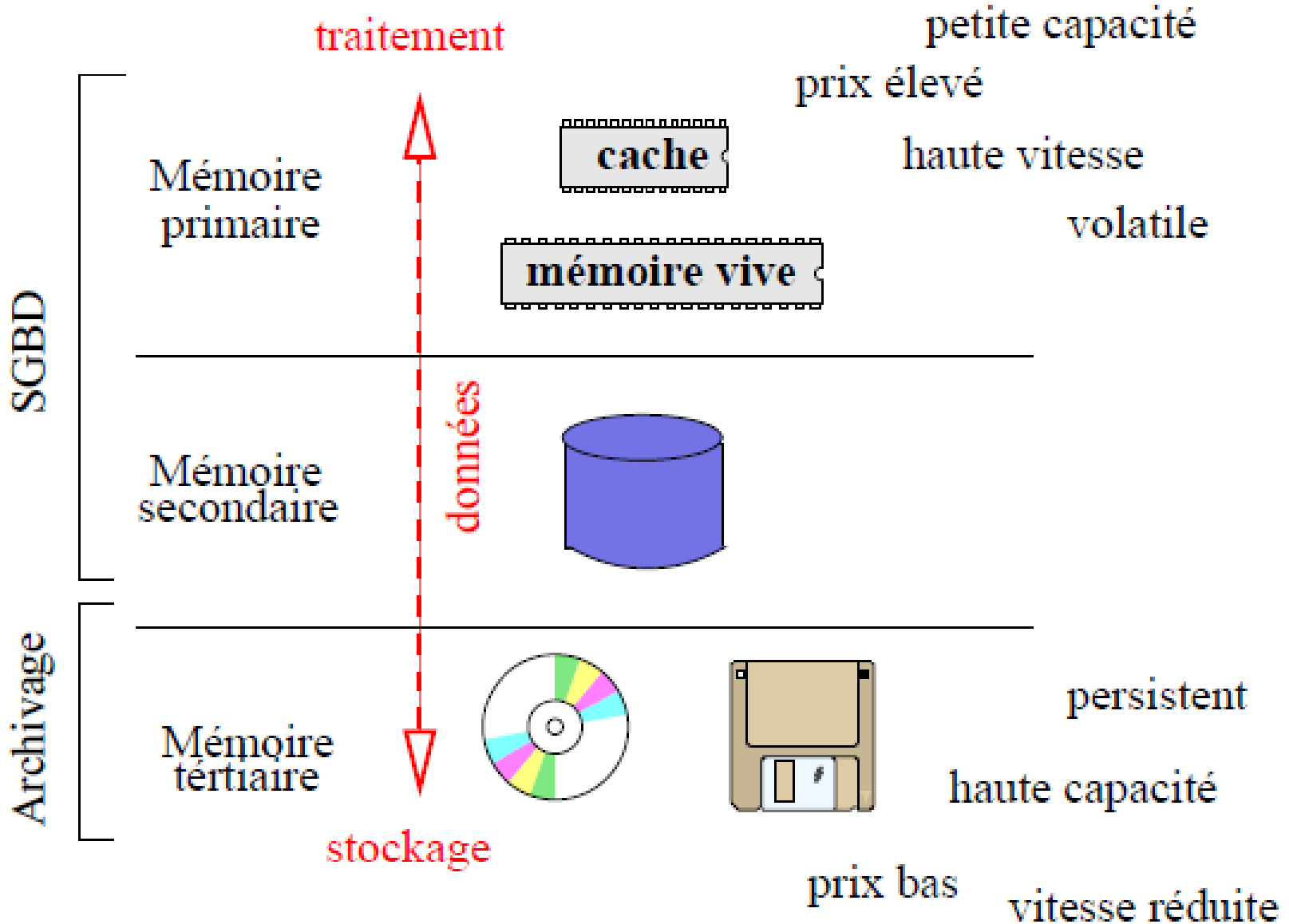
Dr N. BAME

Plan

- **Organisation physique des données**
  - Hiérarchie des mémoires
  - Articles, Pages, Fichiers
- **Méthodes de stockage et d'accès**
  - Séquentielle
  - Séquentielle indexée
  - Arbres B et B+
  - Hachage
- **Gestion des index avec SQL**

Hiérarchie de mémoires

- **Différences** de **temps d'accès**, **prix**, **capacités**



Comparaison mémoire principale / secondaire

Mémoire	Capacité	Prix	Temps d'accès
Principale	Quelques **Go**	Par Go : des **centaines de fois plus petit** pour la **mémoire secondaire**	**nanosecondes**
Secondaire	**centaines** de **Go**, **To**		**millisecondes**

Le **coût** d'une **opération** de la BD est **fonction** du ***nombre d'accès disque*** nécessaires **pour accéder aux données**.

- Il dépend donc fortement de la façon dont les ***données sont organisées*** sur le ***disque***.

Stockage des données

- Les **données** sont **stockées** en mémoire **secondaire** (disques).
- **Disque divisé en *pages(blocs)*** de **taille égale** (même nombre de secteurs contigus)
  - *La **page** est **l'unité*** de **stockage**, et **d'échange** de données entre les **mémoires secondaire et principale** (disque-MC)
    - **Coût** des opérations: **nombre de lectures/écritures de pages**

- Le gérant du disque gère l'espace disque.
  - *L'**unité de gestion est la page***.
  - La ***taille*** de la page (généralement 4K ou 8K) est un ***paramètre du SGBD***.

Fichiers et articles

- Les **données sur disque** sont **stockées** dans ***des fichiers***
  - Un **fichier** occupe plusieurs **pages sur disque**
  - L'accès aux fichiers est géré par le système de gestion de fichiers du système d'exploitation

- Un **fichier** stocke un **ensemble *d'articles*** (enregistrements, n-uplets, lignes)
  - Un article est une séquence de *champs* (attributs)
- Les **articles** sont **stockés dans** des **pages**
  - En général, **taille article < taille page**

Organisation des pages

- Nombreuses variantes, en fonction de **l'organisation du fichier** (séquentielle, triée, …)
- **Page** = **entête de page + zone articles**
- La zone d'articles est divisée en *cellules*
- **Adresse d'un article** dans la page: numéro d'ordre de la cellule occupée
- Suppression d'un article ; sa cellule devient disponible pour accueillir ultérieurement un autre article

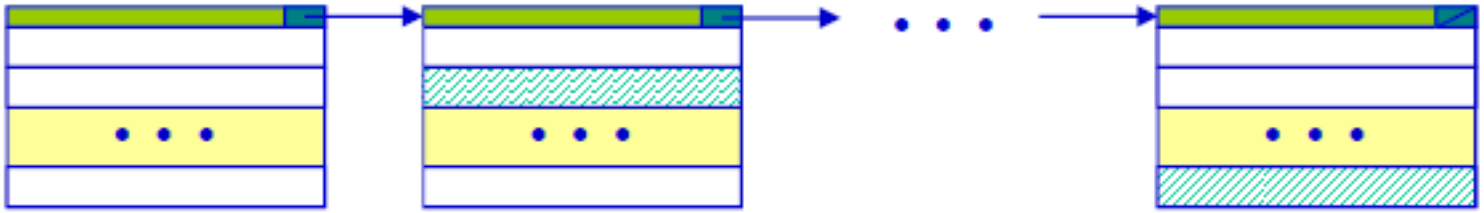
Types d'organisation physique

- Organisation **séquentielle**
  - Stockage des articles suivant **l'ordre d'insertion/suppression**
- Organisation **séquentielle triée**
  - Articles **triés selon une clé**
- Organisation **séquentielle indexée**
  - Articles **triés** selon une **clé + index d'accès rapide par clé**
- **Indexation** avec **arbres B/B+**
  - **Amélioration** du **séquentiel indexé** pour supporter des mises à jour
- Organisation par **hachage**
  - **Regroupement** des articles suivant une **fonction de hachage sur la clé**

Organisation séquentielle

Stockage

- Enchaînement de pages contenant des cellules occupées et des cellules libres (après suppression)



Coût des opérations

- **Rappel**: ce n'est pas le nombre d'opérations sur les articles qui compte, mais ***le nombre d'opérations sur les pages***
- Soit un fichier avec *N* pages
  - Coût exprimé en nombre de **lectures/écritures de pages**

    O(N)

Organisations indexées

- **Objectifs**
  - Accès rapide à partir d'une clé
  - Accès **séquentiel** **trié** ou **non**

- **Moyens**
  - Utilisation de tables permettant la recherche de l'adresse de l'enregistrement à partir de la clé

- **Index**
  - Table (ou plusieurs tables) permettant d'associer à une clé d'enregistrement l'adresse relative de cet enregistrement.

Organisations indexées

Principes de l'indexation

- **Associer** à la clé de l'enregistrement, son adresse dans le fichier à **l'aide d'un index** (table des matières) du fichier
  - Accès rapide à partir de la clé
    - Recherche de l'adresse relative dans *l'index qui est monté en mémoire* dans un tampon
    - Recherche de l'adresse relative de l'enregistrement en mémoire
    - Puis un accès relatif à l'enregistrement dans le fichier
- **Plusieurs méthodes** d'accès indexées
  - qui se distinguent par le mode de placement des enregistrements dans le fichier et par l'organisation de l'index

Organisation séquentielle indexée

- **Amélioration** de l'organisation séquentielle triée
  - ***Fichier de données*** avec les **articles triés sur la clé**
  - On **rajoute *un index* sur les clés**

- ***Index*** : **second fichier**, contenant des **clés** et les **adresses** des **pages** du fichier de données qui correspondent à ces clés
  - Article index: couple **(*c, adr*)**, *c*=clé, *a*=adresse page
    - Signification: dans la page ***adr*** le premier article a la clé ***c***

- **l'index** est lui aussi **trié sur la clé**

Recherche séquentielle indexée

Recherche d'articles de clé *c*

- On **cherche dans l'index la** plus grande clé ***c'*** tel que ***c* <= *c'***
  - **recherche dichotomique**
- On **lit la page qui correspond** à la clé ***c'*** et on cherche les articles de clé ***c*** dans cette page
  - et éventuellement dans les pages suivantes

Avantage: recherche plus rapide **car l'index occupe moins de pages** que le fichier de données

- Si l'index a ***I*** pages, la recherche prend **$C=\log_2 I + 1$**

Catégories d'index

Selon le **nombre de clés dans l'index** :

- ***Index dense***: toutes les clés du fichier de données se retrouvent dans l'index
- ***Index non-dense***: seulement une partie des clés du fichier de données se retrouvent dans l'index
  - Taille réduite, mais le **fichier de données doit être trié**
  - **Un seul index non-dense par fichier (fichier trié sur la clé)**
  - Il contient alors la ***plus grande (ou petite) clé de chaque bloc*** *avec* l'***adresse relative du bloc***.
  - Recherche dichotomique dans l'index

Selon **l'organisation du fichier de données**

- ***Index clustérisé***: **l'index et le fichier** de données **ont la même organisation**
  - Proximité de clés dans l'index → proximité des articles dans le fichier
  - **Un seul** index **clustérisé** par fichier (fichier organisé suivant l'index)
- ***Index non-clustérisé***: les articles du fichier ont une organisation aléatoire par rapport à celle de l'index

Quelques concepts sur les index

- ***Index discriminant*** ou **non** = sur une **donnée discriminante** ou **non** (**identifiant** un enregistrement de manière unique ou **non**)

- ***Index plaçant*** = qui range les articles dans **l'ordre des clés** et les restitue dans l'ordre en lecture séquentielle de la mémoire

- ***Index primaire*** = qui est **basé sur la clé** des articles, permet de les ranger en mémoire

- ***Index secondaire*** = un accélérateur d'accès, cet index est non-discriminant

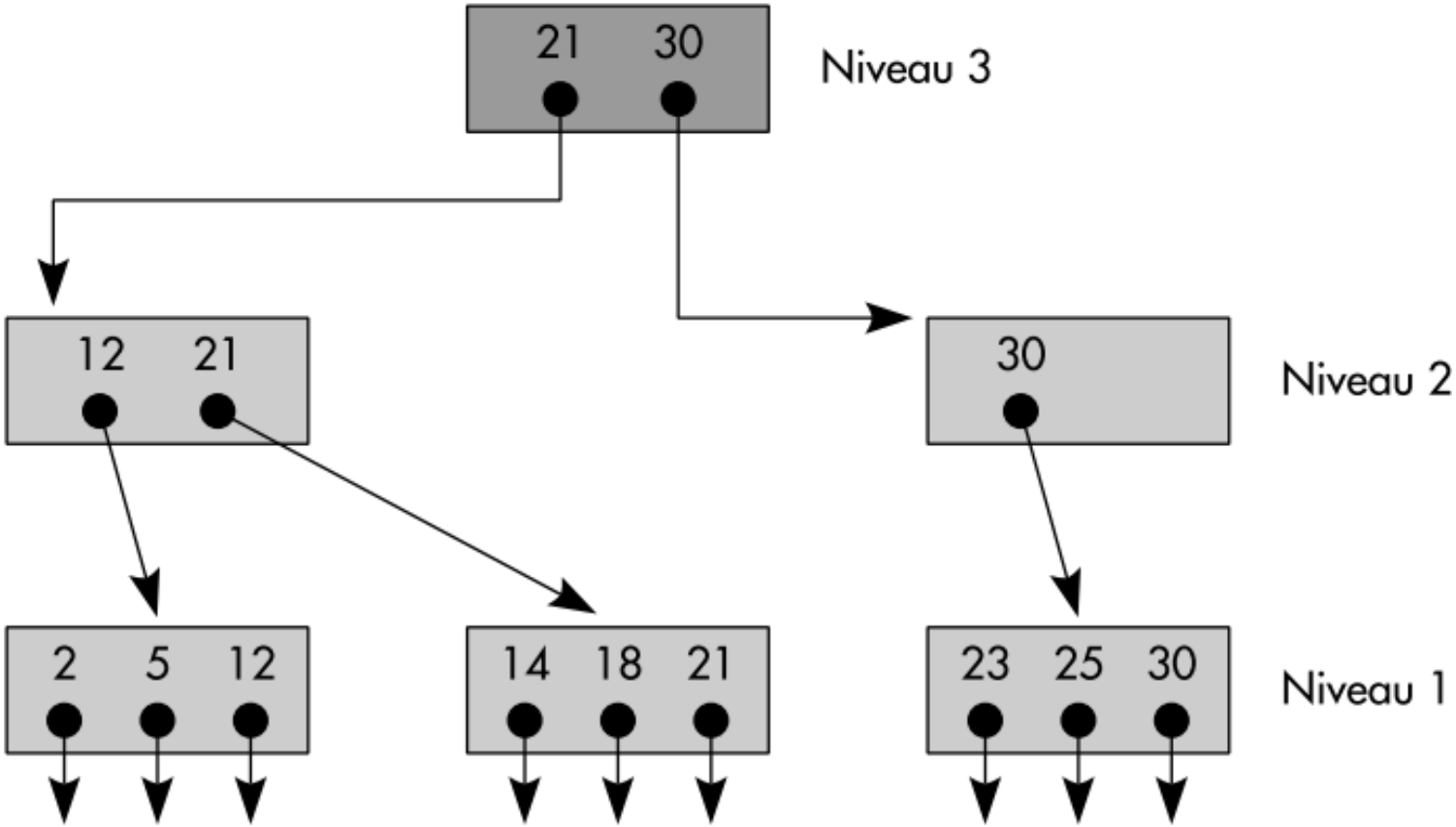
Index hiérarchisé

- Si un index est grand, la **recherche** d'une clé dans l'index peut devenir **très longue**

- **Solution**
  - Créer un ***index de l'index*** vu comme un fichier de clés
    - Index à plusieurs niveaux → ***index hiérarchisé***

- Index à ***n niveaux***, le *niveau k* étant un *index trié* divisé en paquets (pages), possédant lui-même un *index de niveau k+1*, la *clé* de chaque entrée de ce dernier étant la *plus grande* du paquet.

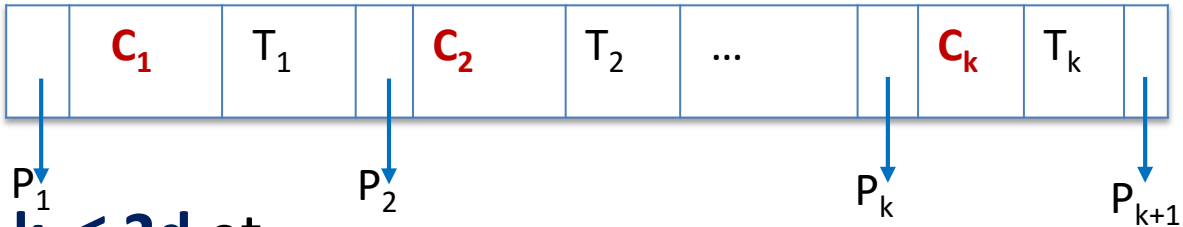
Exemple d'index hiérarchisé



Arbre-B

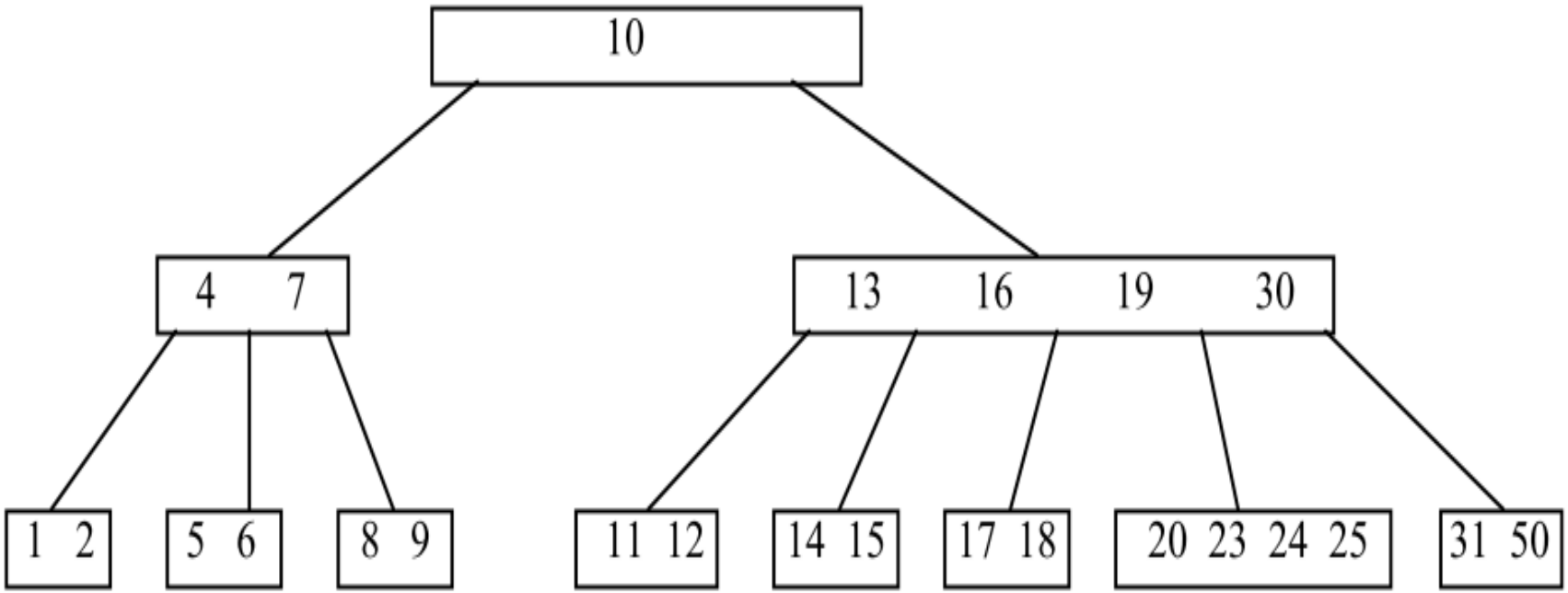
- Un B-arbre d'ordre **d** (d est la mesure de la capacité d'un nœud de l'arbre) est un arbre satisfaisant les conditions ci-après :

- Chaque nœud contient :
  - **k clés** (K_i) triées
  - **k** pointeurs de **données** (T_i) et
  - **k+1** pointeurs **d'index** (P_j)
  - Un pointeur d'index P_j pointe vers un sous-arbre dont chaque clé X est telle que $C_{j-1} < X < C_j$.



- Pour la **racine**, on a **1 ≤ k ≤ 2d** et
- pour les **autres nœuds**, on a **d ≤ k ≤ 2d**
- Pour les **nœuds terminaux** (feuilles) les pointeurs d'index sont **inutilisés**
- Tous les nœuds terminaux ***sont au même niveau*** (on dit que l'arbre est équilibré ou balancé).

Exemple



Recherche dans un arbre B

Recherche des articles de clé *c*

- On part de la racine de l'arbre
  - Soit $K_1, K_2, \ldots, K_n$ les clés du nœud courant
  - Si ***c*** est parmi ces clés ($c=K_i$) →***résultat*** = ***l'article récupéré à l'aide de*** T_i
  - Sinon, si le nœud courant est une feuille → ***résultat*** = ***aucun article***
  - Sinon on choisit l'intervalle $K_{i-1} < c < K_i$ (ou $c < K_1$ ou $c > K_n$) dans lequel se trouve *c* et on continue récursivement avec le nœud fils indiqué par *Pi*

Insertion dans un arbre B

Insertion d'articles de clés c

- On cherche la feuille de l'arbre où doit se faire l'insertion (recherche de *c*)
  - On insère ***c*** à sa place dans la feuille (en gardant l'ordre croissant des clés)
- Si le nœud (**A**) déborde suite à l'insertion (*c* est la $2k+1^{\text{ième}}$ clé)
  - On crée un **nouveau** nœud **B**
  - On garde les *k* premières clés dans A et on met les *k* dernières dans B
  - La **clé du milieu** (la $k+1^{\text{ième}}$) est insérée dans le **nœud parent C**
    - À gauche de cette clé reste le nœud A, à droite se trouvera le nœud B
  - Si le nœud **C** déborde suite à cette insertion, **on continue selon la même méthode**
  - Si la **racine déborde**, la clé du milieu formera une **nouvelle racine** à une seule clé

Suppression dans un arbre B

- On recherche la clé *c* dans l'arbre
  - Si on ne la trouve pas rien à faire
  - Si elle est dans une feuille, on supprime la clé dans la feuille
    - Si la **feuille** reste **avec moins de *k*** clés (manque de clés) on fait une ***rotation inverse*** pour **récupérer une clé à partir de l'un des frères** voisins
    - Si les **frères voisins sont à la capacité minimale** (*k* clés) on ne peut pas emprunter de clé on ***fusionne*** avec l'un des frères voisins en **récupérant une clé du nœud parent**
    - Si le nœud parent reste avec moins de *k* clés, **on applique la même méthode**
    - Si le nœud parent est la racine et qu'elle n'a qu'une seule clé, la racine disparaît et le nœud de fusion devient la nouvelle racine
  - Si elle n'est pas dans une feuille, on la remplace avec la clé ***c'*** immédiatement suivante (ou précédente), qui se trouve toujours dans une feuille
    - Ensuite on supprime ***c'*** de sa feuille suivant la méthode précédente

Suppression dans un arbre B

- Trouver la clé immédiatement suivante à une clé ***c*** d'un nœud interne
  - Dans le sous-arbre à droite de ***c*** on prend la clé la plus petite (qui se trouve dans la feuille la plus à gauche)
  - La clé immédiatement précédente: dans le sous-arbre à gauche de ***c*** on prend la clé la plus grande dans la feuille la plus à droite

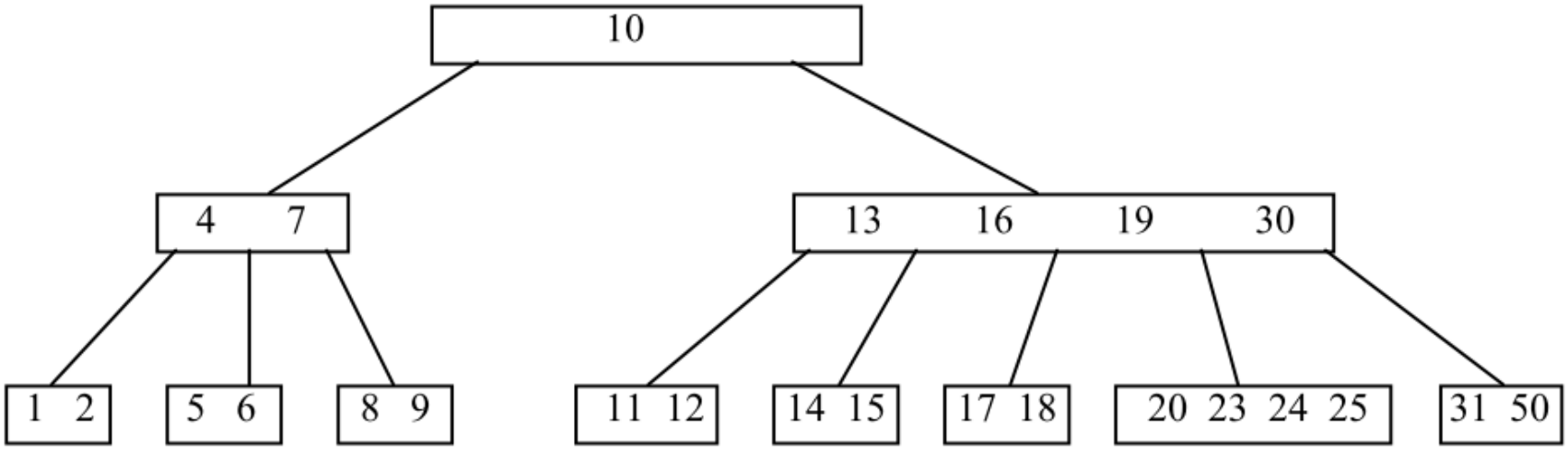
Performances d'un arbre B

- Recherche: parcours d'un chemin dans l'arbre à partir de la racine
  - Nombre de lectures de pages **≤** le nombre de niveaux de l'arbre
    - Hauteur de l'arbre *h* = **nombre d'arcs chemin racine-feuille** = **nombre de niveaux - 1**
  - En moyenne entre ***h* et *h*+1 lectures**
  - Si clés non uniques (ou recherche par intervalle) parcours de plusieurs chemins
- Insertion/suppression : parcours de recherche + insertion/suppression
  - Les éventuels éclatements/fusions suivent un chemin ascendant le nombre d'écritures de pages reste au pire proportionnel à *h*
- **Conclusion : les performances d'un arbre B dépendent de sa hauteur**

Exercices

- On considère le B-arbre d'ordre 2 ci-contre.

1. Décrire les étapes des recherches des enregistrements de clé 19, 9 et 29
2. Déterminer l'arbre obtenu en ajoutant les enregistrements de clés 40 et 21.
3. On souhaite maintenant supprimer l'enregistrement ayant pour clé 5. Donner l'arbre obtenu après suppression.



Arbre B+

- L'utilisation des arbres B pour réaliser des fichiers indexés conduit à des ***traitements séquentiels coûteux*.**
  - En effet, l'accès selon l'ordre croissant des clés à l'index nécessite de **nombreux passages des pages externes aux pages internes**.

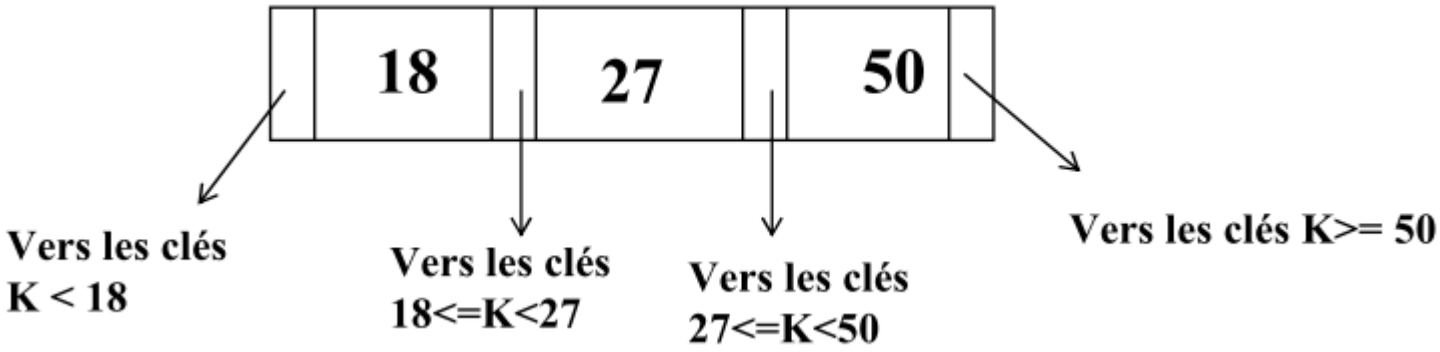
- **Proposition** de ***répéter*** les ***clés*** figurant dans les nœuds internes au ***niveau des nœuds externes***.
- De plus, les pages correspondant aux ***feuilles sont chaînées*** entre elles.
- →**Arbre B+**
  - Les pointeurs externes se trouvent seulement au niveau des feuilles.

Arbre B+

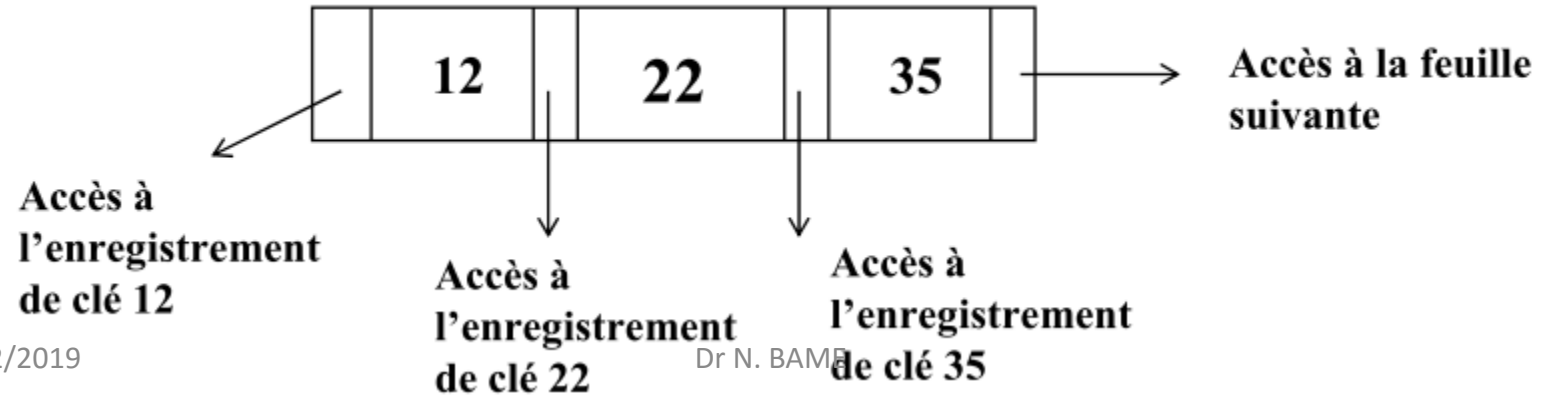
- Les arbres-B+ fournissent des outils de base pour construire des index équilibrés.
- Ils permettent d'améliorer **l'efficacité des recherches**
  - En maintenant automatiquement le bon niveau d'index tout en gardant un arbre équilibré (tous les chemins de la racine aux feuilles ont même longueur)
  - En gérant la place dans les feuilles de façon à ce que chaque feuille soit au moins à moitié pleine.
- **En général**, un arbre B+ **a rarement plus de 4 niveaux**.

Arbre B+

- Les **nœuds internes** servent à **orienter** la recherche



- Les **feuilles** donnent **accès aux enregistrements**



Arbre B+

- ***Recherche*** : Lecture d'un chemin de la racine à un nœud feuille
- ***Insertion :*** Possibilité d'éclatement de nœuds jusqu'à la racine
- **Suppression :** Possibilité de fusion de nœuds jusqu'à la racine

Arbre B+ : Insertion

- Rechercher la feuille où insérer la nouvelle valeur.
- Insérer la valeur dans la feuille en ordre de tri s'il y a de la place.
- Si la feuille est pleine, il y a débordement. Il faut créer un nouveau nœud :
  - Insérer les n/2 premières valeurs dans le nœud original, et les autres dans le nouveau nœud (à droite du premier).
  - La plus petite valeur du nouveau nœud doit être insérée dans le nœud parent, ainsi qu'un pointeur vers ce nouveau nœud.
  - Remarque : les deux feuilles ont bien un nombre correct de valeurs (elles sont au moins à moitié pleines)

Arbre B+ : Insertion

- S'il y a débordement dans le parent, il faut créer un nouveau nœud frère, à droite du premier:
  - Les n/2 premières paires(clé-pointeur) restent dans premier nœud(N), les autres se trouvent dans le nouveau nœud (M).
  - Chaque nœud devant contenir un pointeur de plus que de clé, il y a toujours une clé ne trouvant pas de place. Il s'agit de la clé indiquant la plus petite valeur accessible via le premier fils de M. Cette clé est utilisée par le parent de N et M pour la recherche entre ces deux nœuds.
- Remarque : les divisions peuvent se propager jusqu'à la racine et créer un nouveau niveau pour l'arbre.

Arbre B+ : Suppression

- Supprimer la valeur (et le pointeur vers l'enregistrement) de la feuille où elle se trouve.

- Si la feuille est encore suffisamment pleine, il n'y a rien d'autre à faire. Sinon, il faut
  - redistribuer les valeurs avec une feuille ayant le même parent, afin que toutes les feuilles aient le nombre minimum de valeur requis.
  - répercuter ces modifications au niveau des parents.
  - Si la redistribution est impossible, il faut fusionner des feuilles, et ajuster les valeurs au niveau des parents.
  - Si le parent n'est pas suffisamment plein, appliquer récursivement l'algorithme de suppression.

- **Remarque :** la propagation peut entraîner la suppression d'un niveau.

Exercices

- On dispose d'un arbre B+ d'ordre 2 contenant au départ une seule valeur : 1.
- 1) Donner les arbres obtenus après adjonction des clés successives suivantes : d'abord 15, 3, 12 ; ensuite 17, 4, 11, 7, 2 ; puis 5, 14,6 ; enfin 8, 9, 10, 13, 16.
- 2) Donner les arbres obtenus après suppression des clés suivantes : d'abord 1, puis 12.

Hachage

- Basé sur une **fonction de calcul appliquée à la clé**
  - Renvoie une zone disque ou paquet (page) dans laquelle l'enregistrement est placé
- Un **concurrent** de l'arbre-B+
  - **Meilleur** pour les recherches par clé
  - **N'occupe aucune place**
- Mais
  - se **réorganise difficilement**
  - ne supporte **pas les recherches par intervalle**

- Deux types de méthodes d'accès par hachage
  - Hachage **statique**
    - La taille du fichier est fixe
  - Hachage **dynamique**
    - La taille du fichier peut évoluer

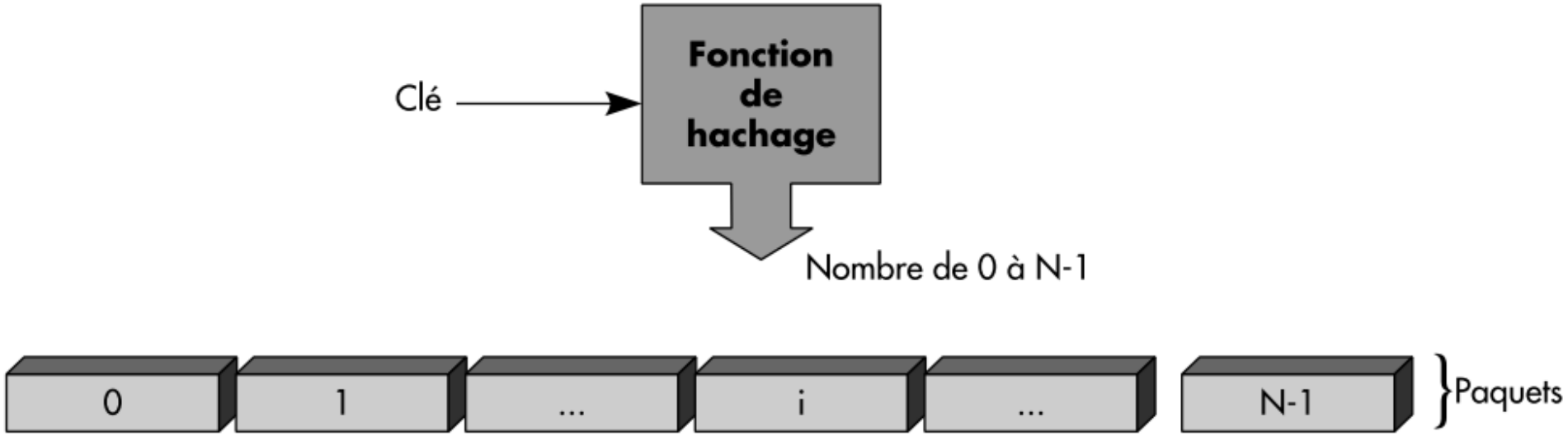
Organisation hachée statique

- **Fichier haché statique** (Static hashed file)
  - Fichier de taille constante fixée à la création dans lequel les enregistrements sont placés dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage fixe appliquée à la clé.
  - Dans un paquet les enregistrements sont rangés par ordre d'arrivée.

- **Différents types de fonctions**
  - Pliage de la clé (combinaison de bits de la clé)
  - Conversion en nb entier
  - ***Modulo P***

- **But**
  - Obtenir une **distribution uniforme** pour éviter de saturer un paquet

Hachage statique



Hachage statique

- ***Très efficace*** pour la recherche par égalité
  - condition ***d'égalité***

- Bonne méthode quand il y a ***peu d'évolution***

- Choix de la fonction de hachage :
  - ***Mauvaise*** fonction de hachage ==> ***Saturation*** locale et ***perte*** de place
  - **Solution :** autoriser les ***débordements***

Débordement

- **Techniques**
  - ***l'adressage ouvert***
    - place l'enregistrement qui devrait aller dans un paquet plein dans le ***premier paquet suivant*** ayant de la ***place libre;***
    - il faut alors mémoriser tous les paquets dans lequel un paquet plein a débordé.
  - ***le chaînage***
    - constitue ***un paquet logique par chaînage*** d'un paquet de **débordement** à un paquet plein.
  - ***le rehachage***
    - applique une **deuxième fonction de hachage** lorsqu'un paquet est plein pour placer en **débordement**.
- ***Dégrade les performances*** et ***complique*** la gestion des fichiers hachés

- **Hachage dynamique**
  - techniques permettant de faire grandir progressivement un fichier haché saturé en distribuant les enregistrements dans de nouvelles régions allouées au fichier.
- Deux techniques principales
  - Hachage **extensible**
  - Hachage **linéaire**

Hachage extensible

- Le fichier **est étendu** dès qu'un paquet est plein
  - Ajout d'un **nouveau** paquet

- Seul le **paquet saturé est doublé** lors de l'extension
  - Il éclate ***selon le bit suivant*** du résultat de la fonction de hachage appliquée à la clé
    - Les enregistrements ayant ce bit à 0 reste dans le paquet, alors ceux ayant ce bit à 1 partent dans le nouveau paquet

- La fonction de hachage adresse un répertoire des adresses de paquets

- La gestion des débordements n'est pas nécessaire

Hachage extensible

- **Fonctionnement**
- Le hachage extensible associe à chaque fichier un répertoire des adresses de paquets.

- Au départ, M bits de la fonction de hachage sont utilisés pour adresser le répertoire.

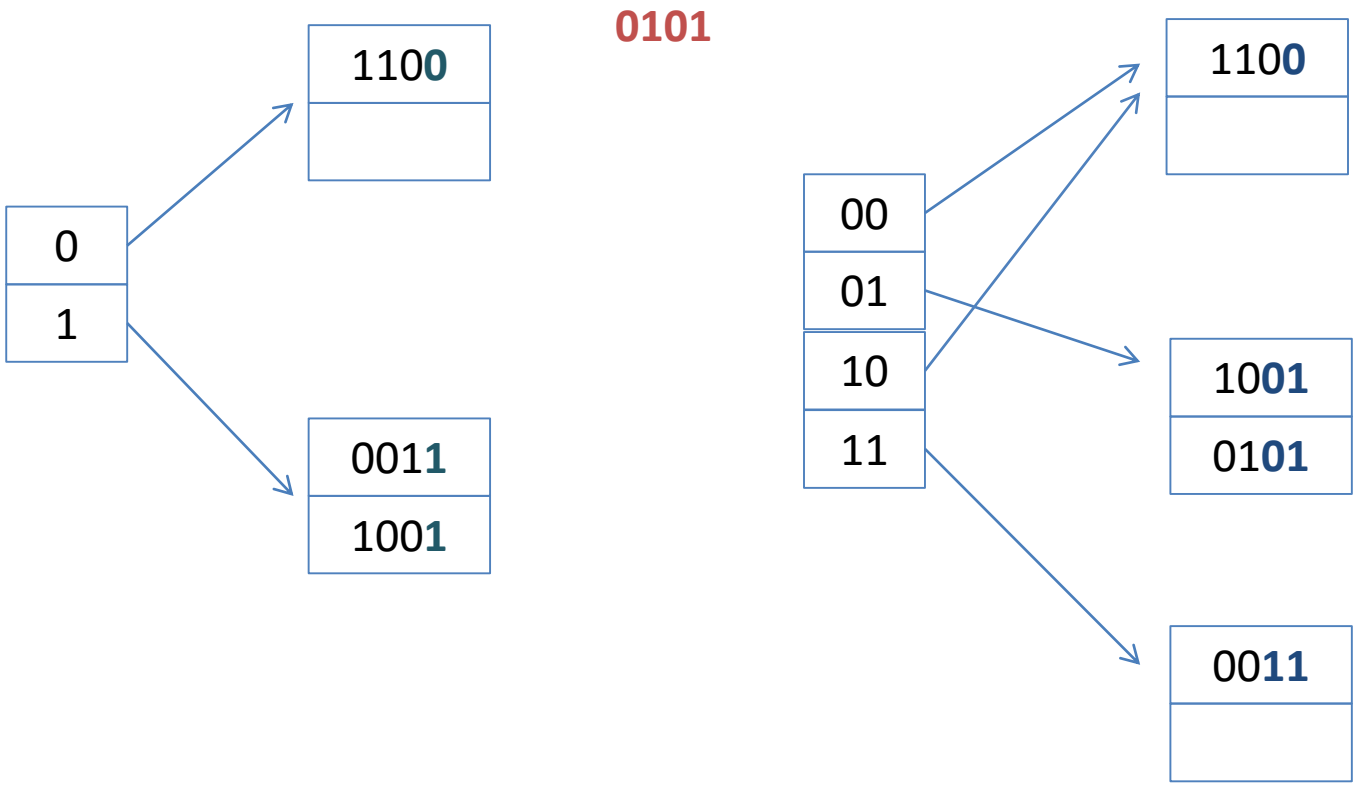
- À la première saturation d'un paquet, le répertoire est doublé, et un nouveau paquet est alloué au fichier.

- Le paquet saturé est distribué entre l'ancien et le nouveau paquets, selon le bit suivant (M+1) de la fonction de hachage.

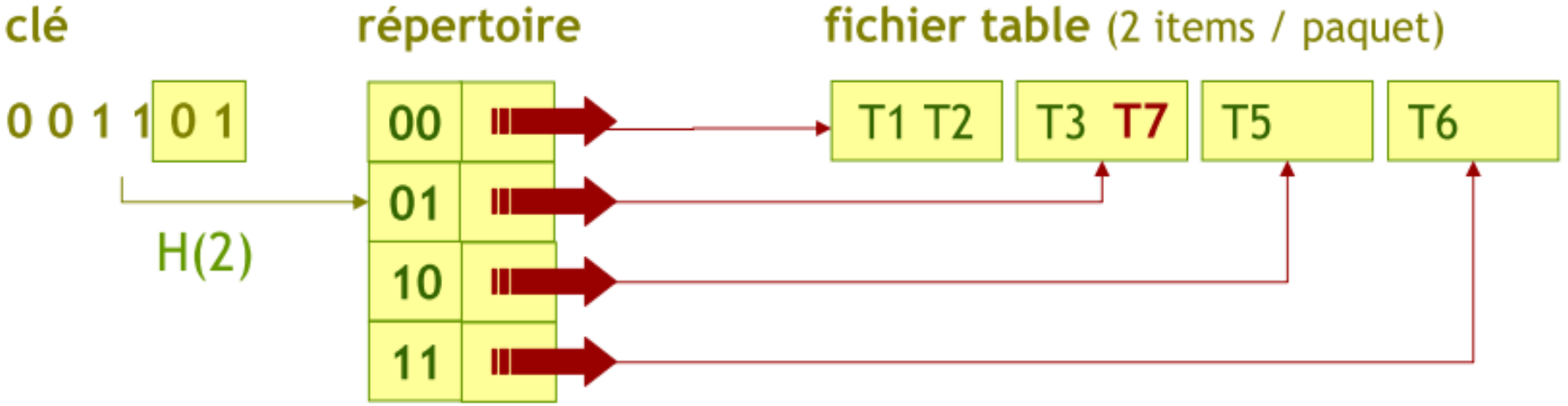
- Ensuite, tout paquet plein est éclaté en deux paquets, lui-même et un nouveau paquet alloué au fichier.

- L'entrée du répertoire correspondant au nouveau paquet est mise à jour avec l'adresse de ce nouveau paquet
  - si elle pointait encore sur le paquet plein.
  - Sinon, le répertoire est à nouveau doublé.

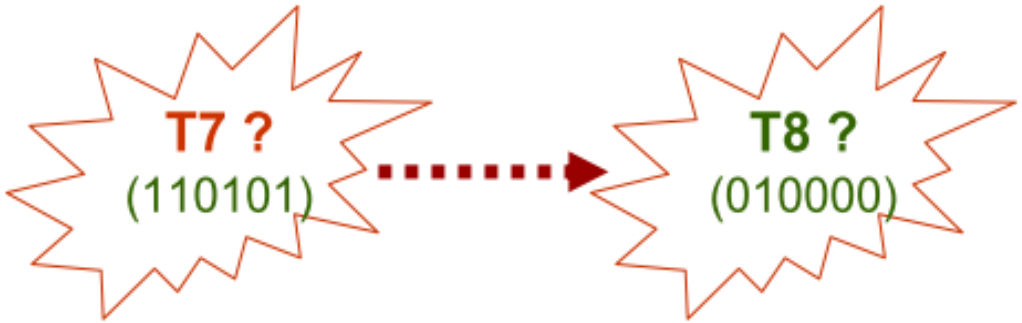
Hachage extensible



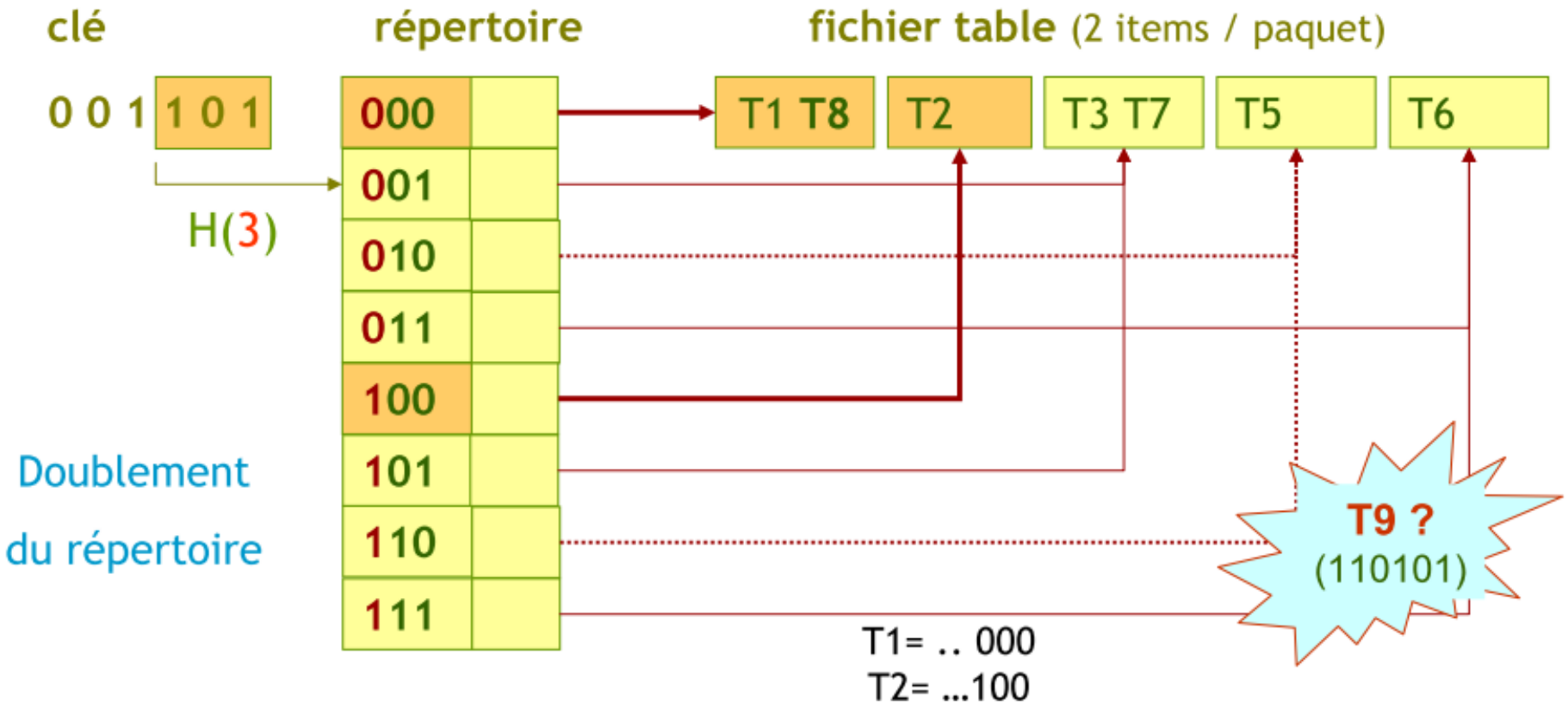
Hachage extensible: insertion



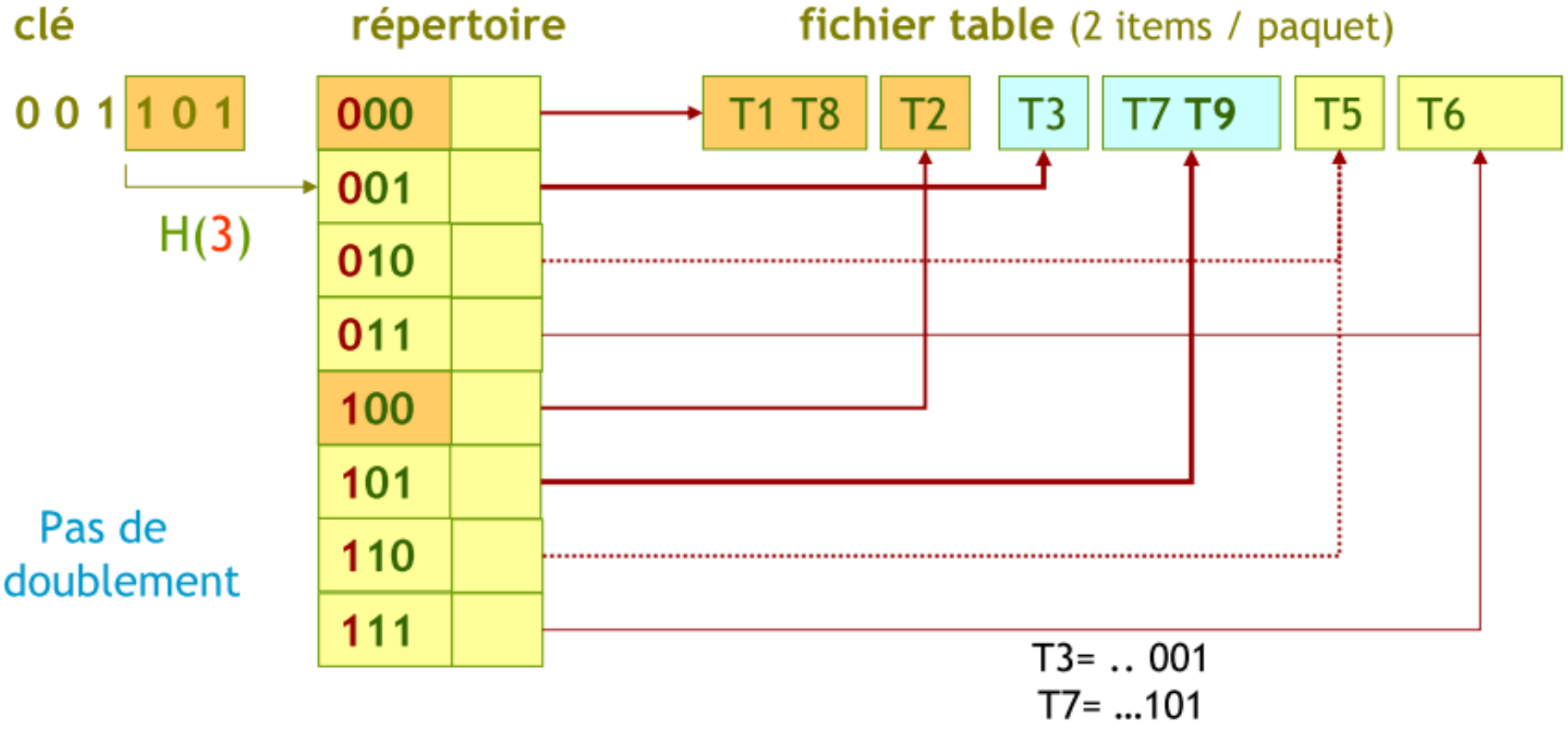
Insertion



Hachage extensible: insertion



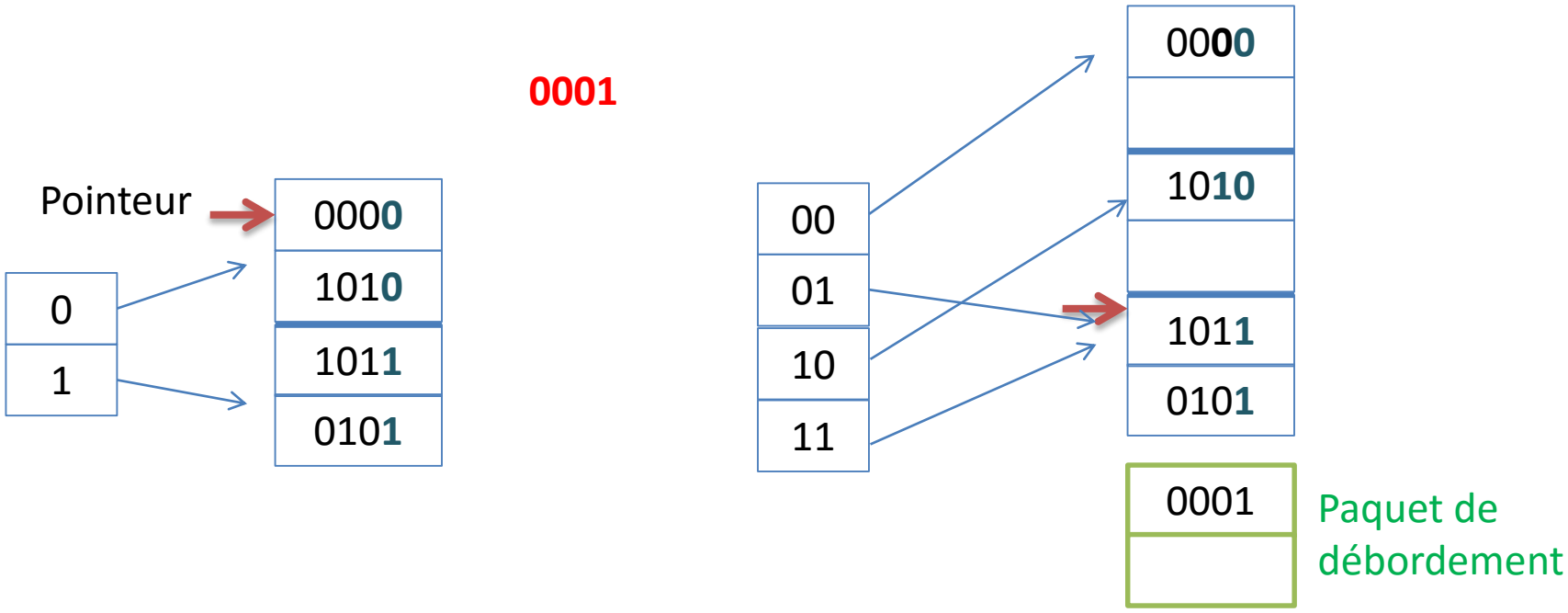
Hachage extensible: insertion



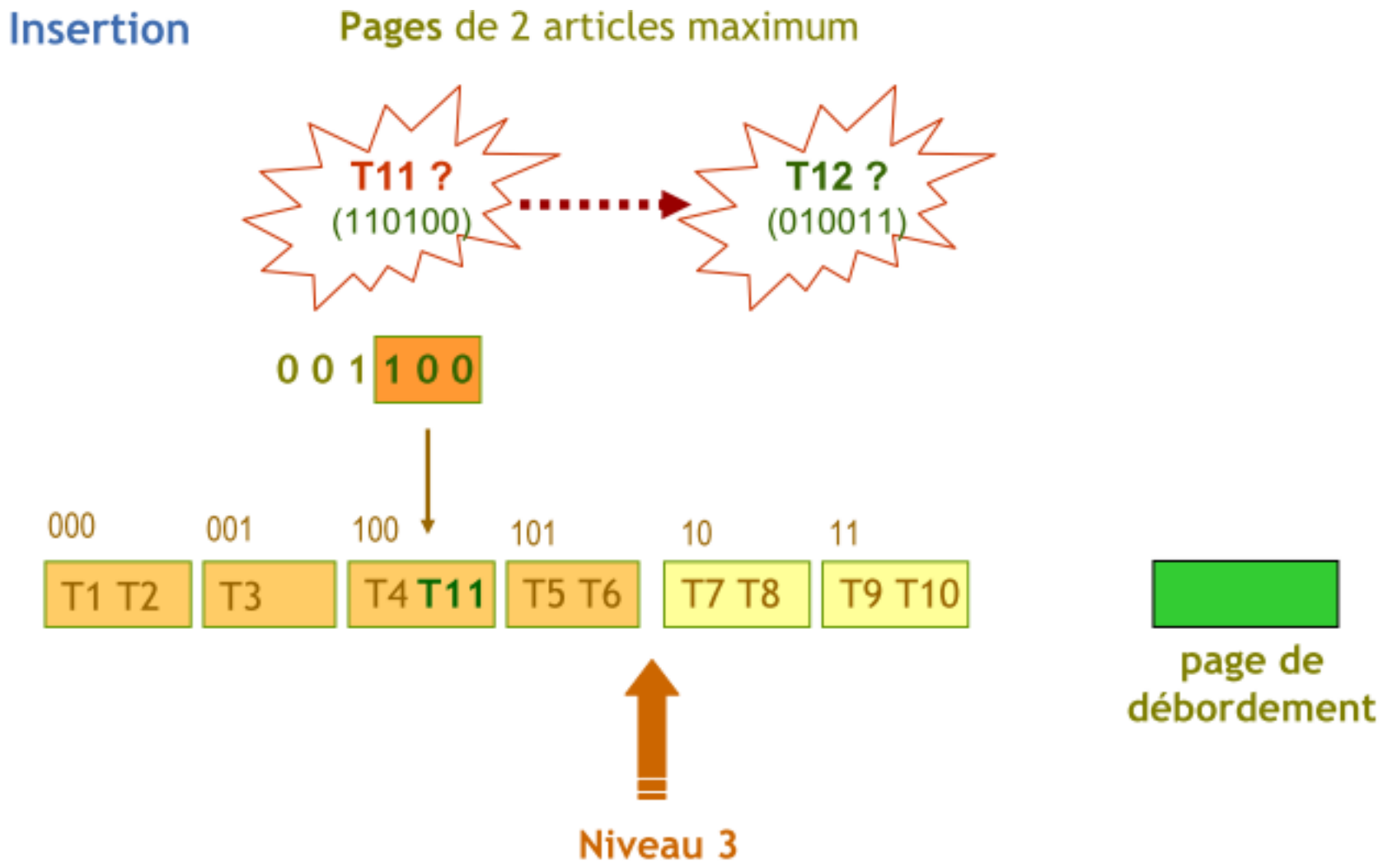
Hachage linéaire

- Le fichier est étendu dès qu'un paquet est plein
- Le paquet **doublé** ***n'est pas*** *celui qui est* ***saturé*** mais ***un paquet pointé par un pointeur courant*** initialisé au premier paquet du fichier est incrémenté de 1 à chaque éclatement d'un paquet
  - Lorsque le pointeur atteint la fin du fichier, il est repositionné au début du fichier
- Un niveau d'éclatement P du fichier est conservé dans le descripteur du fichier afin de préciser la fonction de hachage.
- Une gestion de débordement est nécessaire puisque le paquet plein n'est pas en général éclaté

Hachage linéaire

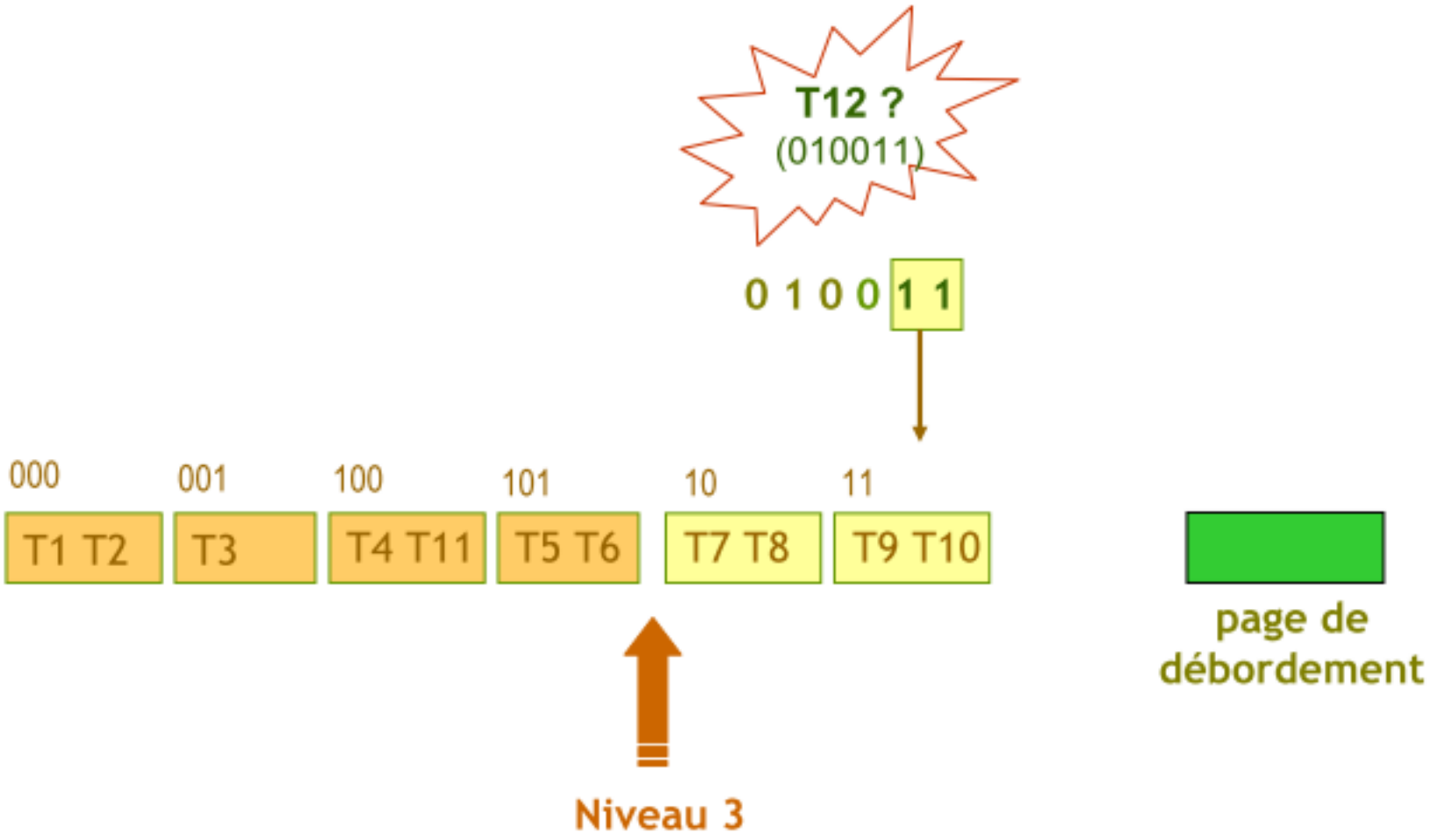


Hachage linéaire : insertion



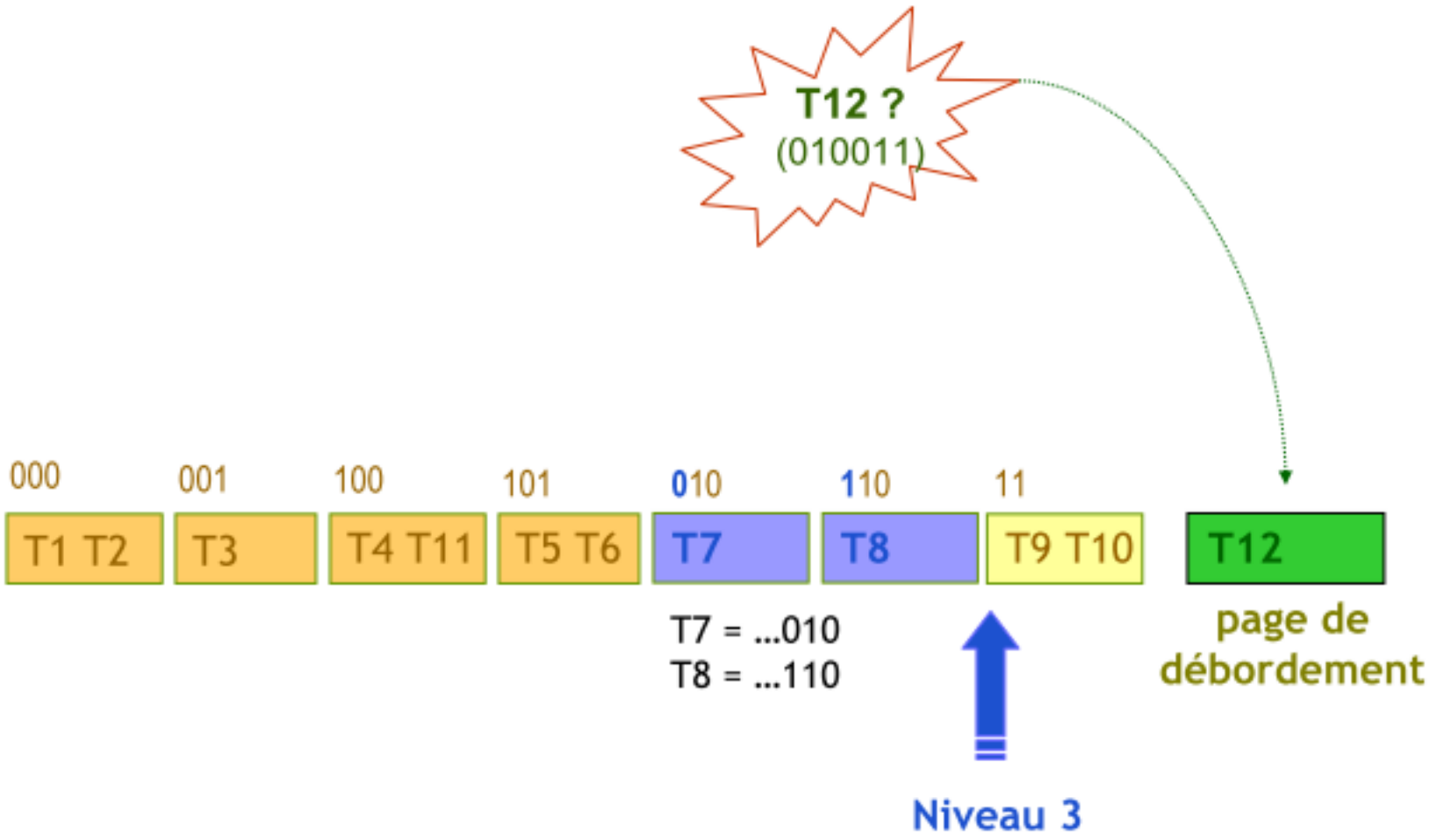
Hachage linéaire : insertion

Insertion avec débordement

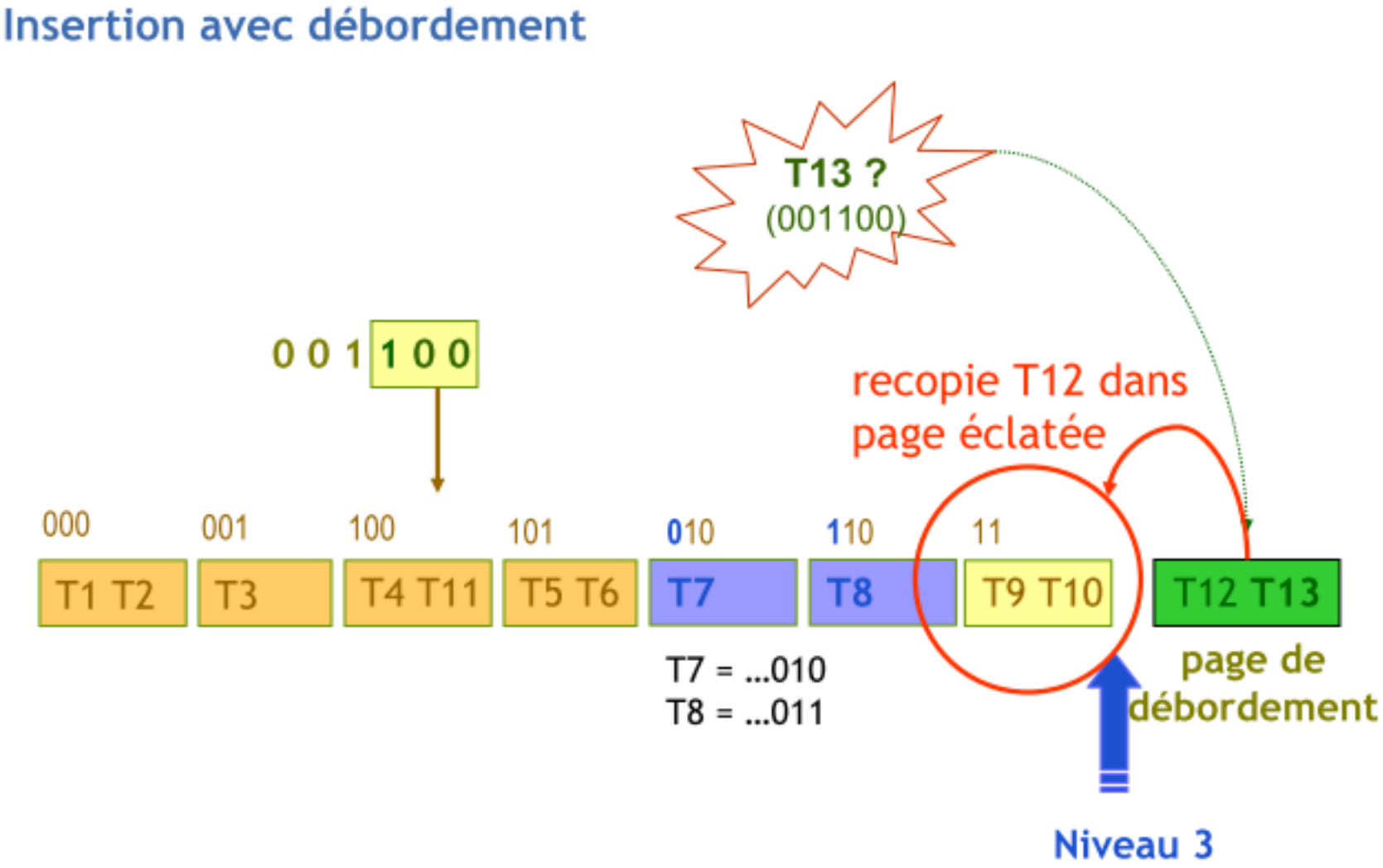


Hachage linéaire : insertion

Insertion avec débordement : éclatement du paquet courant



Hachage linéaire : insertion



Comparaisons B+Tree vs. Hachage

Arbre B+

- Supporte les ***point*** (égalité) et ***range*** (inégalité) ***queries***
- Le nombre **d'E/S** dépend de la **hauteur** de l'arbre (**usuellement** h reste **petit**)
- La **taille de l'index** reste potentiellement **importante**
- Le coût de mise à jour est potentiellement important

Hachage

- Ne supporte que les ***point queries***
- Performance **optimale** (resp. **mauvaise**) quand les données **sont bien** (resp. **mal**) **distribuées**

Gestion des index sous SQL

- **Création**
  - **CREATE** [**UNIQUE**] **INDEX** **index_name** **ON** **table_name** [**USING** {**btree**|**hash**}] **(indexed_column_names)**
- **Exemples**
  - **CREATE INDEX Ind_Nom_Etudiant ON Etudiant USING btree** (**nomE**);
  - **CREATE UNIQUE INDEX Uind_Eval ON** Evaluation **USING hash** (Id_Etudiant, CodeMatiere);
- **Suppression**
  - **DROP INDEX index_name FROM** *table_name*

Gestion des index sous SQL

- Les SGBDR **créent systématiquement** un index chaque fois que l'on pose une clé primaire (**PRIMARY KEY**) ou une contrainte d'unicité (**UNIQUE**) sur une table.
- En revanche, il **n'y a pas d'index** créé automatiquement par le SGBDR derrière une **FOREIGN KEY** (clef étrangère).