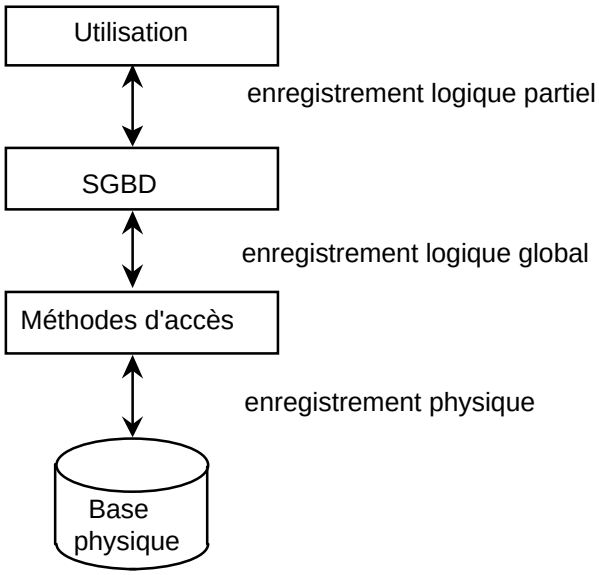
VIII. ORGANISATIONS PHYSIQUES

1. Introduction

a) Notion de chemins d'accès

Un SGBD est greffé du SGBD sur un système d'exploitation. Il utilise donc les organisations physiques disponibles à ce niveau (fichiers séquentiels, fichiers relatifs, etc.). Les possibilités d'organisations plus sophistiquées sont spécifiques à chaque SGBD.



Accès aux données dans une base de données

Plusieurs possibilités de chemins d'accès à un enregistrement permettent de minimiser les temps de réponse.

Il existe toujours au moins un chemin d'accès pour un enregistrement : par la clé primaire, avec la méthode d'accès définie sur le fichier correspondant.

Il est possible de définir d'autres chemins d'accès à cet enregistrement, à partir des valeurs des autres champs, en vue de faciliter la résolution de certaines requêtes. Il devient alors nécessaire d'utiliser des emplacements de mémorisation supplémentaires sur le support : le volume d'un fichier peut croître dans des proportions allant de 1,5 à 2.

Les chemins d'accès sont définis *a priori* lors de la conception de la base. Il est possible de les remanier ultérieurement dans certains cas seulement.

Critères à considérer pour le choix d'un chemin d'accès :

- le temps de réponse à une recherche élémentaire (en nombre d'Entrées/Sorties) ;
- les facilités de modification, de suppression et d'insertion ;
- le stockage supplémentaire ;
- les facilités de réorganisation ;
- les facilités de reprise après pannes.

b) Méthodes d'accès standards du système d'exploitation

Accès séquentiel.

Accès direct :

- accès direct à un enregistrement à partir de son adresse absolue (ex. accès à l'enregistrement N) ;
- accès direct relatif à un enregistrement à partir de son adresse relative (ex. accès à l'enregistrement suivant).

c) Méthodes d'accès non standards

Accès aléatoire (hachage).

Accès chaîné.

Accès indexé.

2. Organisations aléatoires

a) Hachage statique

- L’adresse absolue ou l’adresse relative de l’enregistrement est obtenue par une *fonction de hachage* h.
- La fonction est généralement appliquée sur la clé primaire.
- La fonction peut être fournie par le système ou par l'utilisateur.
- Permet un accès presque aussi performant que l'accès direct à partir de la valeur externe d'un champ et non plus à partir d'une adresse physique ou relative.
- C’est un *accès associatif* (c’est-à-dire un accès par valeur).

Inconvénients :

- mauvaise utilisation de l'espace de stockage car des zones vides peuvent subsister ;
- risque de collision car la procédure de hachage peut fournir une même adresse pour deux valeurs de clé différentes (⇒ nécessité d’une zone de débordement pour stocker les enregistrements en collision).

Exemples de fonctions de hachage :

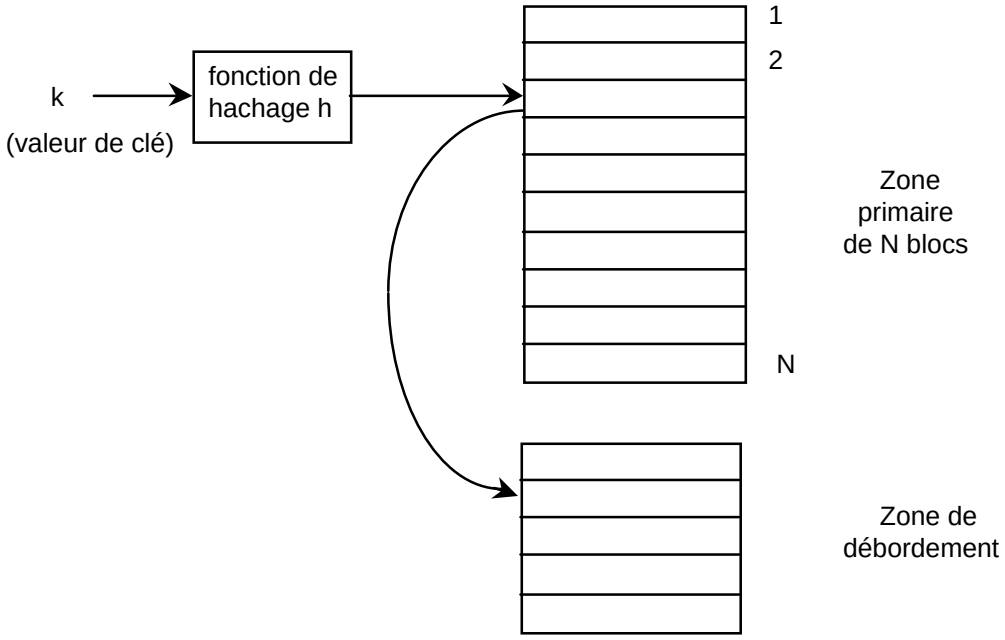
- conversion en nombre binaire suivie d'une troncature ou d'une extraction ;
- élévation au carré suivie d'une extraction ;
- reste de la division par le nombre premier P le plus proche du nombre N de blocs.

Une "bonne" fonction de hachage doit assurer une répartition aussi uniforme que possible dans l'espace d'adressage de façon à réduire les collisions et à utiliser au mieux l’espace disponible.

Remarques :

- Il est possible de grouper les enregistrements dans des blocs (pages).
- Le nombre moyen d'Entrées/Sorties nécessaires à un accès est légèrement supérieur à 1 si le nombre de collisions reste limité.

- Il est nécessaire de fixer *a priori* la taille de la zone primaire et de procéder périodiquement à des réorganisations.



Hachage statique

b) Hachage dynamique avec répertoire

La valeur k' = h(k) obtenue par la fonction de hachage sert à accéder aux entrées d'un *répertoire* qui contiennent chacune un *pointeur* vers un bloc de données.

Dans k’ on ne considère que les d digits de poids supérieurs.

Pour éviter d'utiliser plus de blocs de données que nécessaire, plusieurs entrées successives du répertoire peuvent pointer sur le même bloc de données.

Un bloc de données peut donc rassembler des données dont seulement les p (avec p < d) premiers digits sont égaux.

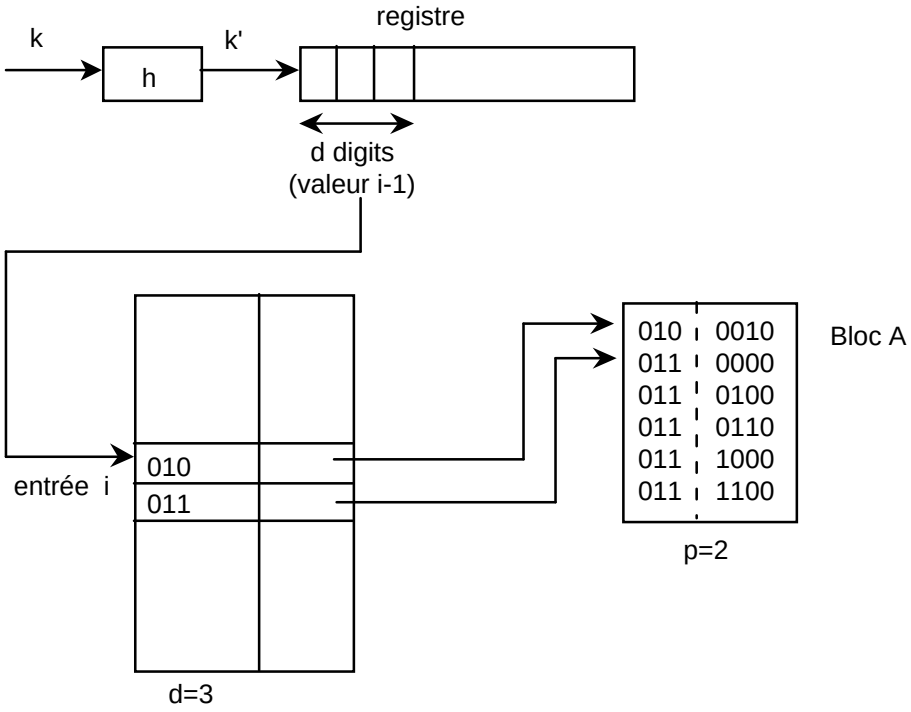
p = profondeur de discrimination du bloc

d = profondeur de discrimination du répertoire

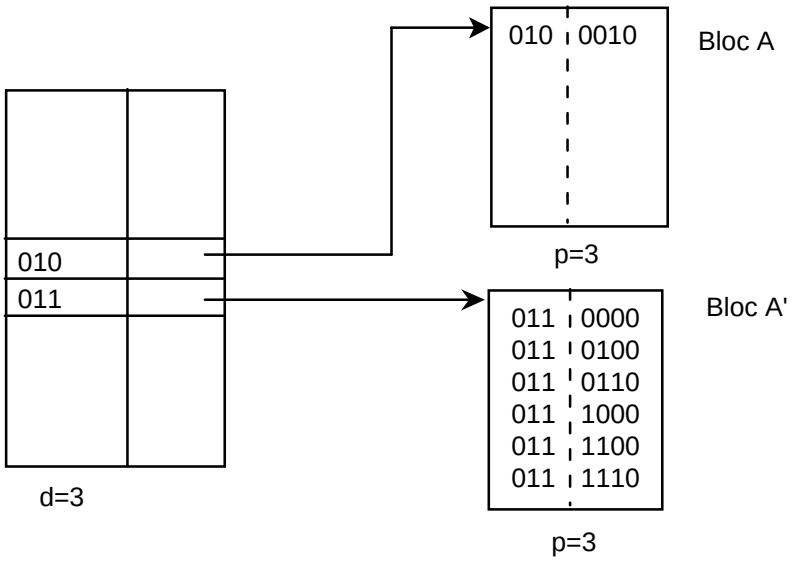
À saturation d'un bloc A, une partie de son contenu est placée dans un nouveau bloc A'. La profondeur de discrimination des deux blocs A et A' devient p+1. Dans A, on laisse tous les enregistrements pour lesquels le digit de rang p+l dans la clé est égal à 0. Dans A', on place tous les enregistrements pour lesquels le digit de rang p+l dans la clé est égal à 1.

Réorganisation des pointeurs dans le répertoire :

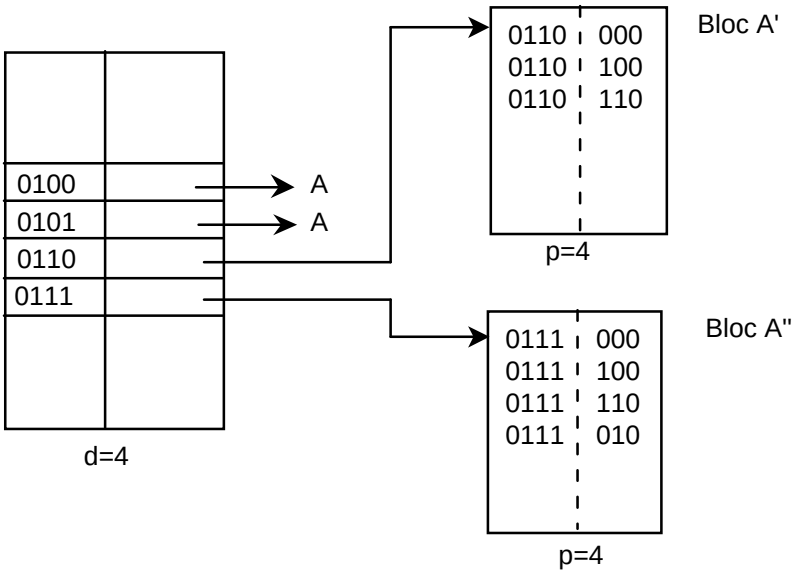
Si p < d, il existe une ou plusieurs entrées pouvant accueillir un pointeur vers A et un pointeur vers A'. Si p = d, il faut introduire de nouvelles entrées dans le répertoire. On les obtient en faisant passer sa profondeur de discrimination à d+1, c'est-à-dire en doublant sa taille.



Hachage dynamique avec répertoire



Partage du bloc A suite à l'insertion d'un enregistrement de clé 011 1110



Partage du bloc A' et évolution du répertoire suite à l'insertion d'un enregistrement de clé 011 1010

c) Hachage dynamique linéaire

Le problème des collisions est résolu en divisant le contenu d'un bloc. Mais ce bloc n'est pas obligatoirement celui où s'est produit la collision. Avantage : croissance linéaire de l'espace d'adressage.

Soient N le nombre de blocs et h_0 une fonction de hachage sur ces N blocs, numérotés de 0 à N-1.

Première collision sur l'un quelconque des N blocs : on crée un nouveau bloc de numéro N et on répartit le contenu du bloc 0 dans les blocs 0 et N en appliquant une nouvelle fonction de hachage h_1.

Deuxième collision : on crée un nouveau bloc de numéro N+1 et on répartit le contenu du bloc 1 dans les blocs 1 et N+1 toujours avec la fonction h_1.

Et ainsi de suite.

Un enregistrement en collision reste chaîné à son bloc initial, tant que ce dernier n'a pas fait l'objet d'une division.

Après N collisions, la taille de l'espace d'adressage a été multipliée par 2.

Le processus de division est réitéré sur l'ensemble des 2N blocs existants en commençant par le premier et ainsi de suite. Sur les blocs divisés et les nouveaux blocs créés on applique maintenant une nouvelle fonction de hachage h_2.

Lorsque la taille de l'espace d'adressage atteint 2^jN blocs, la procédure de division s'applique à nouveau au bloc 0 avec une nouvelle fonction de hachage h_j.

Il faut trouver une famille de fonctions de hachage h_j, permettant d'assurer une répartition aussi uniforme que possible après division d'un bloc. Par exemple on pourra prendre pour h_j le reste de la division par 2^jN.

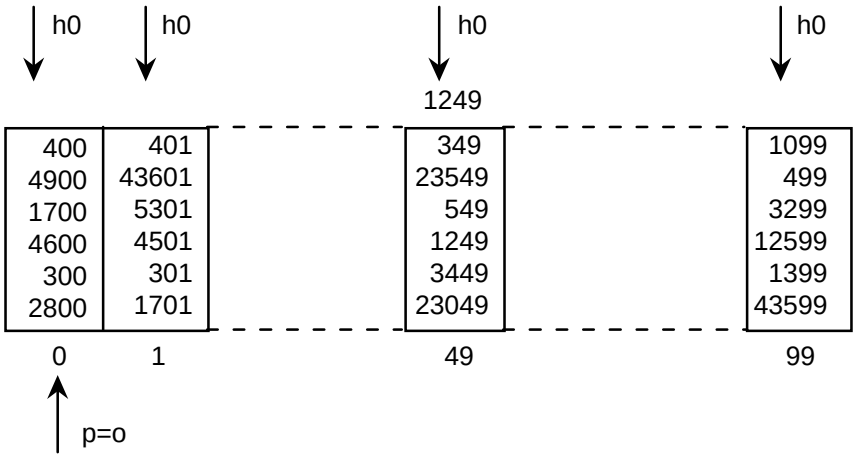
L'indice j de la dernière fonction de hachage utilisée est lié au numéro n du dernier bloc créé par la formule $2^{j-1}N \leq n < 2^jN$.

Algorithme de recherche d'un enregistrement de clé k :

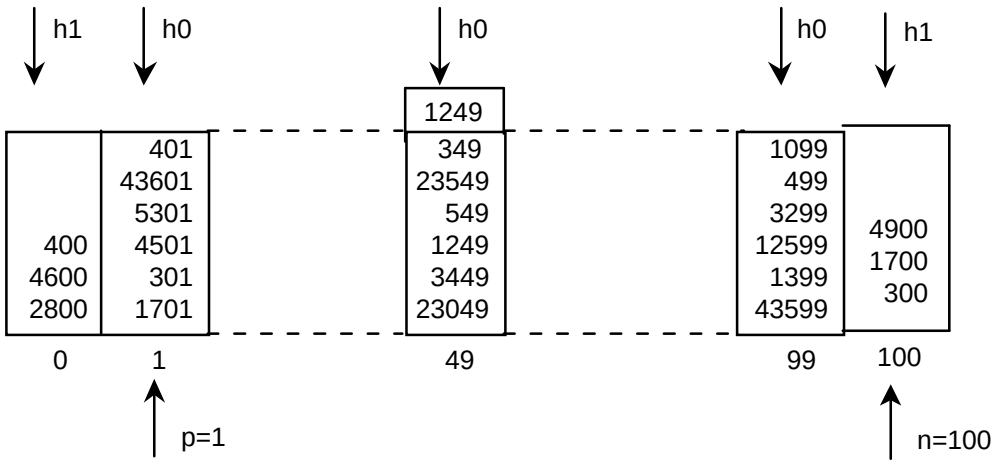
(j = indice de la dernière fonction de hachage)
(p = numéro du prochain bloc à éclater)

Performances : 1.7 accès disque en moyenne pour la recherche d'un enregistrement, avec un taux de remplissage de 80%.

```
Débutproc
    rechercher(k)
    si p = 0 alors m ← hj(k) sinon
        début
           m ← hj(k)
           si m<p alors m ← hj+1(k)
        fin
    finsi
    inspecter le bloc d'adresse m et éventuellement son bloc de
    débordement
finproc
```



a- Une collision intervient pour le bloc 49



b- A chaque collision le bloc pointé par p est divisé

Hachage dynamique linéaire

3. Organisations chaînées

a) Principes

- Chaque enregistrement contient au moins un pointeur vers l'enregistrement "suivant".
- Possibilité de plusieurs pointeurs : un même enregistrement apparaît alors dans plusieurs chaînes (ou listes).
- Le chaînage permet de dissocier complètement l'ordre logique et l'ordre physique et d'éviter la redondance.
- On accède à un enregistrement en parcourant la liste et en testant à chaque pas la valeur de la clé ou de la donnée concernée.
- *Tête de liste* : contient l'adresse du premier enregistrement.
- *Fin de liste* : marquée par une valeur spécifique (pointeur NIL par exemple).

b) Les pointeurs

Les pointeurs sont des adresses et peuvent être de trois types :

- *adresse machine* : l'enregistrement est caractérisé par son adresse physique sur le support (par exemple, numéro de cylindre, numéro de piste, numéro de secteur) ;
- *adresse relative* : l'enregistrement est caractérisé par sa position relative dans un fichier ;
- *adresse logique ou symbolique* : l'enregistrement est caractérisé par la valeur de sa clé primaire (l'enregistrement est stocké dans un fichier avec accès associatif).

	avantages	inconvénients
adresse physique	.temps d'accès court	.longeur de l'adresse .dépendance physique .faible évolutivité
adresse relative	.temps d'accès court .faible encombrement .indépendance physique	.dépendance par rapport à l'ordre de création
adresse symbolique	.indépendance physique .indépendance par rapport à l'ordre de création	.temps d'accès long .existence d'une clé

Avantages et inconvénients des différents types de pointeurs

c) Implémentation

Deux types d'implémentation :

- soit directement sur un support adressable (ce qui suppose une implémentation en langage d'assemblage) ;
- soit par l'intermédiaire d'un fichier à accès direct (adresse absolue ou adresse relative) ou d'un fichier à accès aléatoire (adresse symbolique).

Dans les deux cas on dispose d'un ensemble fini de cellules dans lequel sont implantées les différentes chaînes. La gestion de cet ensemble de cellules est assurée comme pour des listes implantées en mémoire centrale.

4. Organisations indexées

a) Principe

Index primaire = table ou ensemble de tables permettant d'associer à la clé d'un enregistrement d'un fichier à accès sélectif, l'adresse de cet enregistrement.

Index secondaires = permet d'associer une valeur d'un champ aux adresses des enregistrements présentant cette valeur pour ce champ.

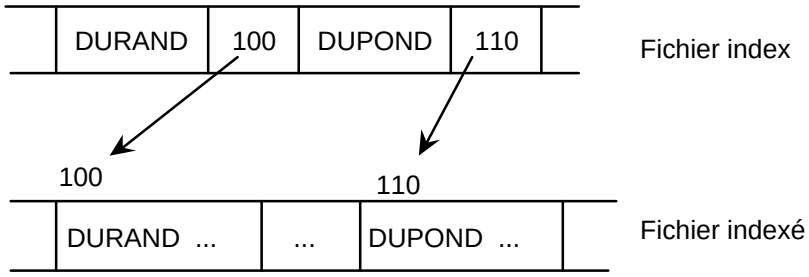
Un index est une solution pour mettre en place un *accès associatif*.

Le type de l'adresse dépend du type d'organisation du fichier :

- organisation directe – adresse absolue,
- organisation relative – adresse relative,
- organisation aléatoire – adresse symbolique.

Un index primaire construit sur un fichier à organisation aléatoire ne présente aucun intérêt. Par contre, un index secondaire avec un tel type de fichier reste *insensible* à toute réorganisation de ce fichier. Un index défini sur un fichier à organisation directe ou relative doit être remanié si le fichier est réorganisé.

Un index est lui-même un fichier qui doit faire l'objet d'une organisation. On parle donc de *fichier index* et de *fichier indexé*.



Fichier index et fichier indexé

b) Caractéristiques des index

Densité :

Toutes les clés ne figurent pas obligatoirement dans l'index.

Densité = quotient du nombre de clés dans l'index par le nombre d'enregistrements du fichier (comprise entre 0 et 1).

Un index *dense* a une densité égale à l.

Un index *non dense* (ou creux) a une densité inférieure à 1. Dans ce cas, l'index et le fichier doivent être triés. Le fichier est alors divisé en paquets organisés comme suit : dans chaque paquet les enregistrements sont stockés par ordre croissant (ou décroissant) et sont tels que tous ceux du paquet P(i) précédent (succèdent) ceux du paquet P(i+1). La méthode d'accès dite séquentielle indexée utilise de tels index.

Si un est index trié, il y a possibilité de recherche dichotomique.

Index combiné :

L'indexation est effectuée sur plusieurs champs combinés.

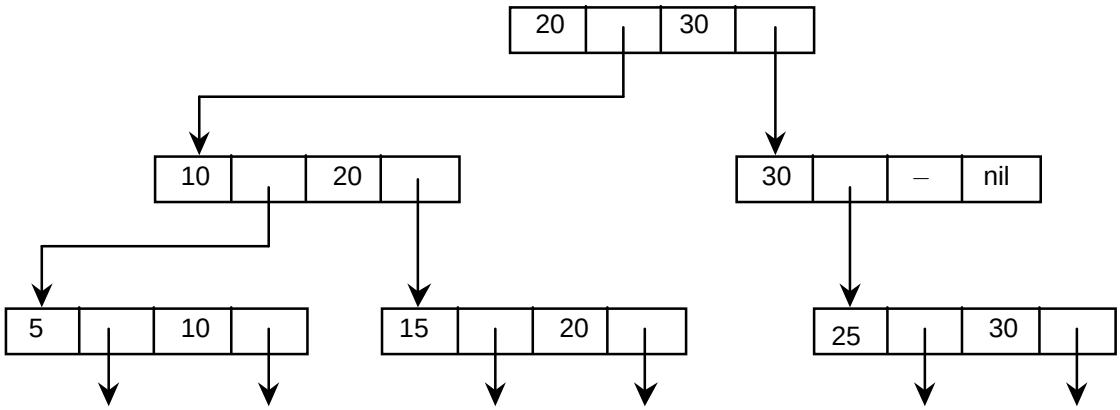
Soit un index combiné sur les champs R_1 / R_2 / ... / R_n. Un tel index permet de sélectionner rapidement les enregistrements qui satisfont à une condition du type :

$R_1 = a_1$ et $R_2 = a_2$ ... et $R_n = a_n$

On remarquera qu'un index combiné sur R_1 / R_2 / ... / R_n est aussi un index pour un sous-ensemble de $\{R_1, R_2, ..., R_n\}$.

Index hiérarchisé :

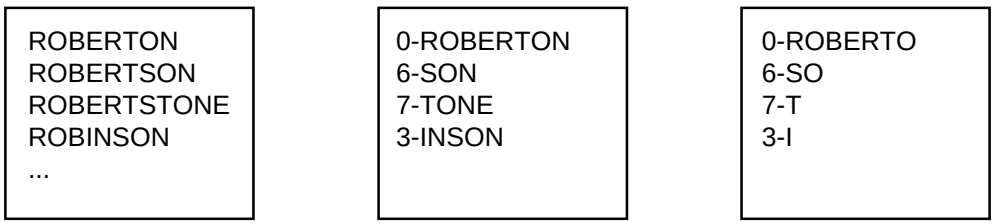
Ce type d'index est utilisé pour faciliter l'exploration d'un index important. Un index hiérarchisé à *n* niveaux est un index hiérarchisé à *n-1* niveaux possédant lui-même un index à un niveau. L'implémentation s'effectue sous forme d'arbres.



Index hiérarchisé sous forme d'arbre

Index compacté :

Un index compacté permet l'amélioration des performances (diminution du volume de stockage, augmentation de la rapidité des échanges) par compactage des entrées. Cela est possible seulement pour des index triés comportant des entrées alphanumériques.



a- index initial

b- index après compression en tête

c- index après compression en tête et en queue

Index compacté

c) Quelques organisations d'index à un niveau

Fichier inverse (cas b, c, d) :

Utilisation d'un index secondaire dense. L'intérêt de cette organisation est de faciliter considérablement la résolution de questions faisant intervenir les valeurs du champ inversé. Deux inconvénients : d'une part, les enregistrements de l'index sont de longueur variable ; d'autre part, cette structure doit être remaniée à chaque mise à jour du fichier de données.

Tableau de digits (cas d) :

C'est une solution qui permet de s'affranchir de la longueur variable. Elle n'est viable que si un élément du tableau est effectivement représenté par un bit machine.

Inverse avec adresses symboliques (cas c) :

Index insensible aux réorganisations physiques du fichier de données.

d) Les B-arbres

C'est structure la plus utilisée pour implémenter des index hiérarchisés. Le nom provient de l'anglais *B-tree* (B pour *Balanced* : balancé, équilibré).

Définition :

Un B-arbre d'ordre d est un arbre satisfaisant les conditions ci-après.

1) Chaque nœud contient k clés (K_i) triées selon l'ordre lexicographique gauche droite, k pointeurs de données (T_i) et k+1 pointeurs d'index (P_j) et est structuré comme suit : $< P_1 <K_1, T_1> P_2 ... P_i <K_i, T_i> P_{i+1} ... P_k <K_k, T_k> P_{k+1} >$. Un pointeur de données T_i pointe vers l'enregistrement de clé K_i du fichier de données. Un pointeur d'index P_j pointe vers un sous-arbre dont chaque clé X est telle que $K_{j-1} < X < K_j$.
2) Pour la racine, on a $1 \leq k \leq 2d$ et pour les autres nœuds, on a $d \leq k \leq 2d$.
3) Pour les nœuds terminaux (feuilles) les pointeurs d'index sont inutilisés.
4) Tous les nœuds terminaux sont au même niveau (on dit que l'arbre est équilibré ou balancé).

Fichier de données à indexer

Adresses relatives	Clés A	B	C	D
0	P	1	8	M
1	Q	2	11	F
2	R	3	16	M
3	S	4	15	M
4	T	1	30	M
5	U	2	17	F
6	V	4	9	F
7	W	4	5	F
8	X	2	21	M
9	Y	4	14	F
10	Z	1	12	M

A	Adresse paquet
R	0
U	3
X	6
Z	9

a- index par paquets sur la clé primaire

B	Adresse relative
1	0, 4, 10
2	1, 5, 8
3	2
4	3, 6, 7, 9

b- index dense sur la clé secondaire B

B	A
1	P, T, Z
2	Q, U, X
3	R
4	S, V, W, Y

c- même index que b mais avec adresses symboliques

B	adresse relative 0	1	2	3	4	5	6	7	8	9	10
1	1	0	0	0	1	0	0	0	0	0	1
2	0	1	0	0	0	1	0	0	1	0	0
3	0	0	1	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	1	1	0	1	0

d- index avec représentation des adresses par tableau de digits

C	adressse paquet 0	3	6	9
5	0	0	1	0
10	1	0	1	0
15	1	1	0	1
20	1	1	0	0
25	0	0	1	0
30	0	1	0	0

e- index par paquets tableau de digits

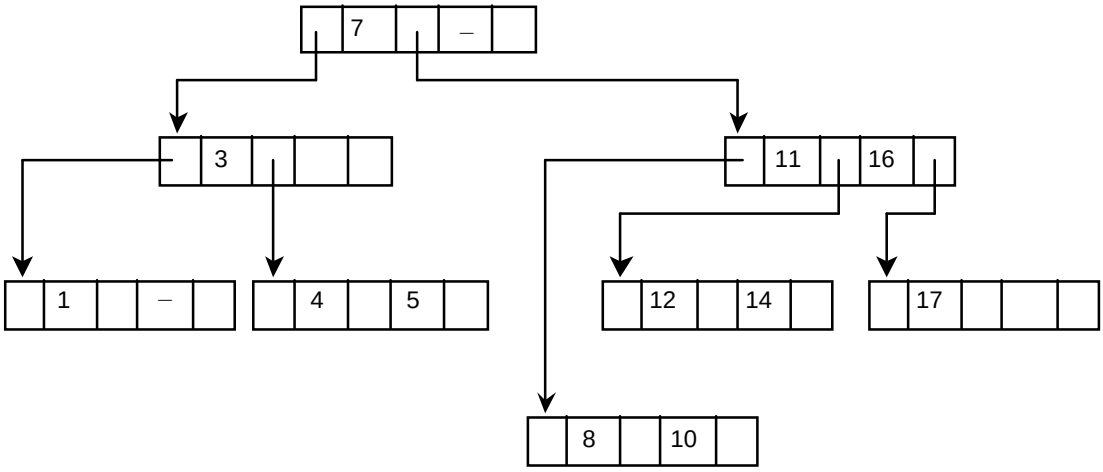
x	adresse relative 0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	1	1	0	0	0
15	1	1	0	1	0	0	1	1	0	1	1
20	1	1	1	1	0	1	1	1	0	1	1
25	1	1	1	1	0	1	1	1	1	1	1
30	1	1	1	1	1	1	1	1	1	1	1

f- index cumulatif sur C: C Š x ?

B	D	adresse relative
1	F	-
1	M	0, 4, 10
2	F	1, 5
2	M	8
3	F	-
3	M	2
4	F	6, 7, 9
4	M	3

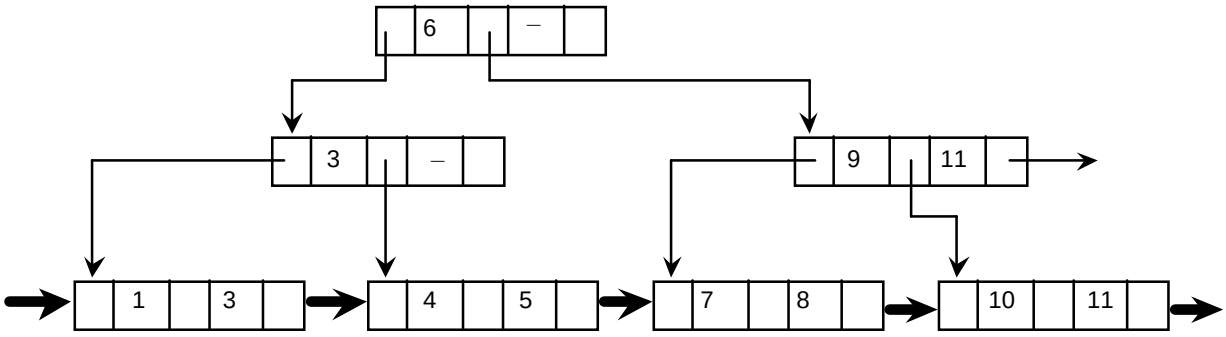
g- index combiné sur B et D

Quelques organisations d'index à un niveau



B-arbre d'ordre 1 (seuls les pointeurs d'index sont représentés)

B^+-arbre = les clés effectives (et les pointeurs de données éventuels) n'apparaissent que dans les feuilles. Les nœuds internes contiennent des valeurs possibles de clés qui servent uniquement à localiser les feuilles. De plus les feuilles sont liées séquentiellement.



B^+-arbre

Recherche dans un B-arbre :

Pour rechercher une clé c, il faut la comparer successivement à chacune des clés de la racine. S'il y a égalité, on peut accéder directement à l'enregistrement (par la clé ou par le pointeur de donnée associé). Si c est plus petite, il faut accéder au nœud fils de gauche par le pointeur index correspondant. Si c est plus grande, il faut poursuivre avec la clé suivante du même nœud, sauf s'il s'agit de la dernière clé du nœud, auquel cas il faut accéder au nœud fils de droite. Le processus se poursuit sur le nœud fils ainsi accédé.

Performances : un nœud par bloc ⇒ l'accès à un nœud entraîne un accès disque. Le nombre d'accès disque pour explorer l'index est donc égal au plus au nombre de niveaux de l'arbre. Soit h la hauteur du B-arbre. La capacité minimale N_{min} et la capacité maximale N_{max} de l'arbre, en nombre de clés, sont respectivement égales à :

$$N_{min} = 2(d+1)^h - 1$$
$$N_{max} = (2d + 1)^{h+1} - 1$$

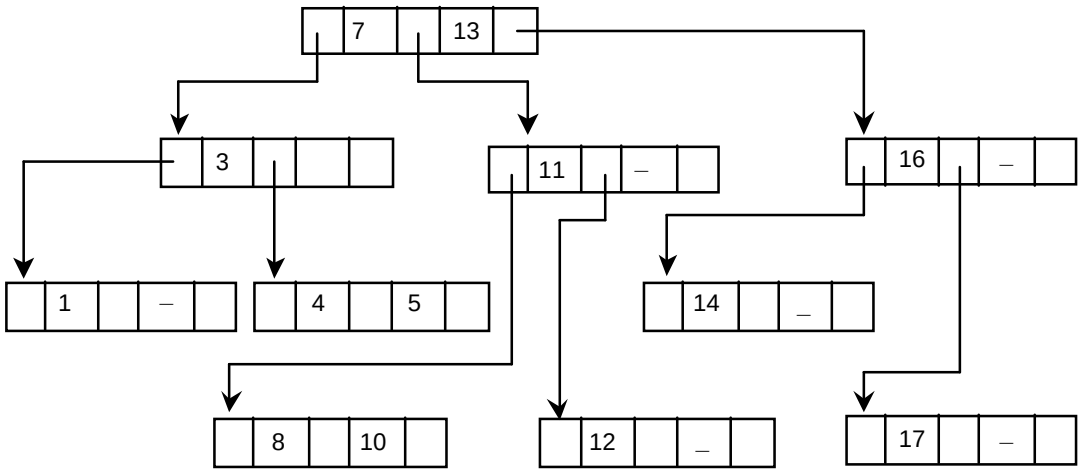
Exemple : pour d = 50 (chaque nœud peut accueillir 100 clés) et h = 3, on a N_{min} = 265 301 et $N_{max} \approx$ 104 millions. L'accès à un enregistrement nécessite alors au plus 4 accès disque (3 pour l'exploration de l'index et 1 pour l'accès au fichier).

Insertion dans un B-arbre :

Pour insérer une clé c, il faut rechercher la feuille contenant les clés encadrant c (en général l'insertion se fait dans une feuille), puis insérer cette clé dans la feuille s'il reste de la place disponible. En cas de débordement, il faut éclater la feuille en deux (en prélevant un nœud dans l'espace libre) et répartir les clés comme suit :

- les d clés les plus petites sont placées dans la feuille de gauche ;
- les d clés les plus grandes sont placées dans la feuille de droite ;
- la clé médiane est insérée dans le nœud de niveau supérieur.

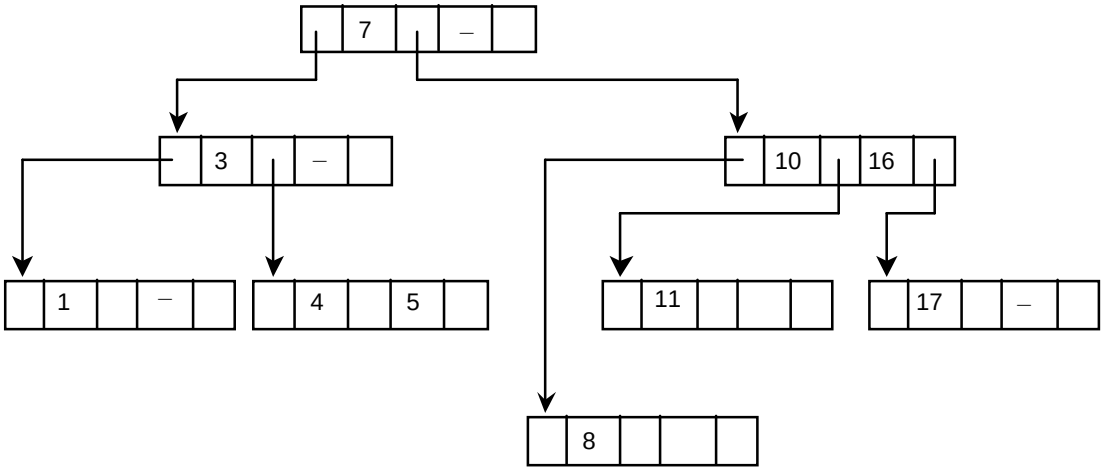
Si ce dernier nœud est saturé, l'opération d'éclatement peut se poursuivre dans les niveaux supérieurs, éventuellement jusqu'à la racine. La hauteur de l'arbre augmente alors de 1. Il est possible de retarder l'éclatement en redistribuant les clés dans les nœuds adjacents.



Évolution suite à l'insertion de la clé 13

Suppression dans un B-arbre :

Pour supprimer une clé c, il faut rechercher le nœud A contenant c, puis supprimer c dans ce nœud. Si ce nœud n'est pas une feuille, on remplace c par la clé la plus à gauche dans le sous-arbre droit de c (c'est la première clé de la feuille la plus à gauche de ce sous-arbre). Si le nombre de clés devient inférieur à d (sous remplissage), il faut tenter soit une redistribution des clés sur un des nœuds adjacents (si son nombre de clés est supérieur à d), soit une concaténation avec un nœud adjacent (sinon). Des concaténations successives peuvent se propager jusqu'à la racine. La hauteur de l'arbre diminue alors de 1.



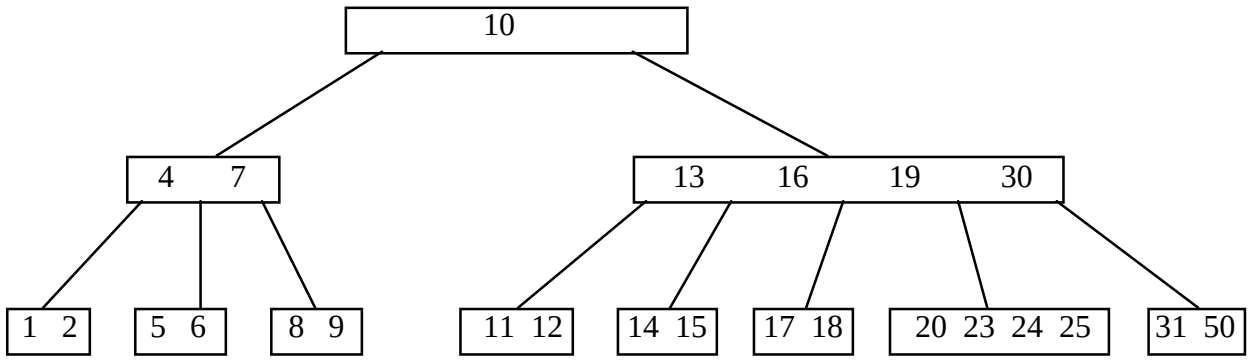
Évolution suite aux suppressions des clés 12 et 14

Coûts d'insertion et de suppression : Ils sont proportionnels à log(N) (base d).

5. Exercices

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

1) Déterminer l'arbre obtenu en ajoutant l'enregistrement de clé 21.
2) On souhaite maintenant supprimer l'enregistrement ayant pour clé 5. Donner l'arbre obtenu après suppression.

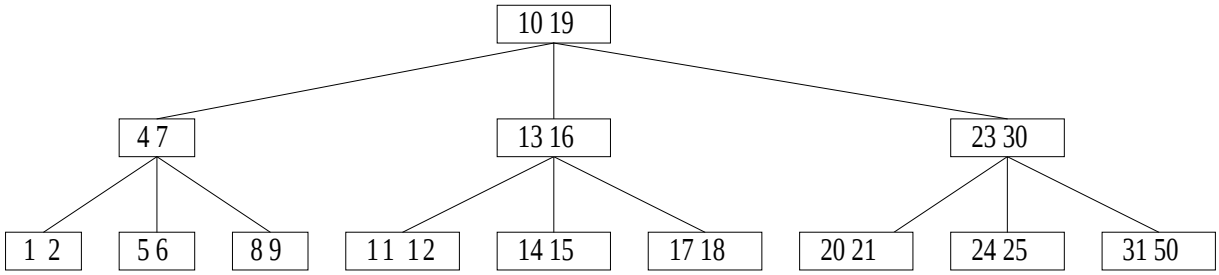


II. On dispose d'un B-arbre d'ordre 2 contenant au départ une seule valeur : 1.

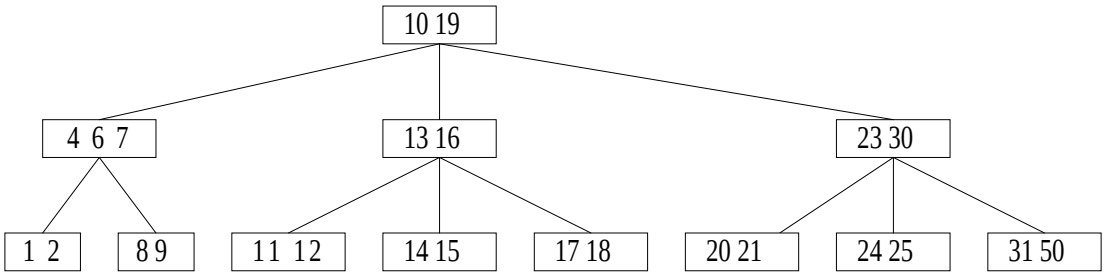
1) Donner les arbres obtenus après adjonction des clés successives suivantes : d'abord 15, 3, 12, 6 ; ensuite 4, 11, 7, 2 ; puis 5, 14 ; enfin 8, 9, 17, 10, 13, 16.
2) Donner les arbres obtenus après suppression des clés suivantes : d'abord 1, puis 12.

Correction des exercices

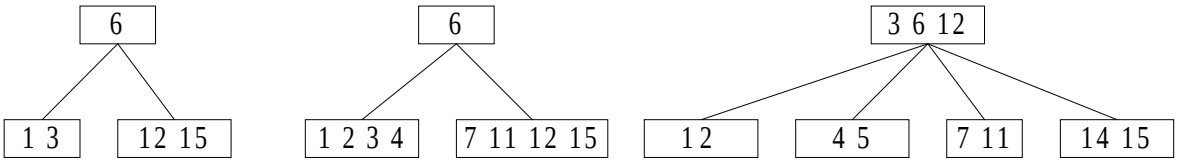
I. 1)

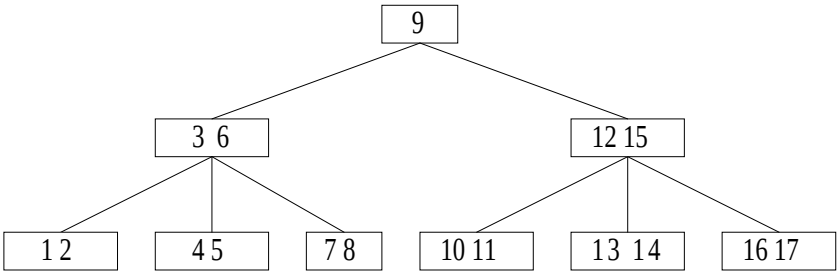


2)

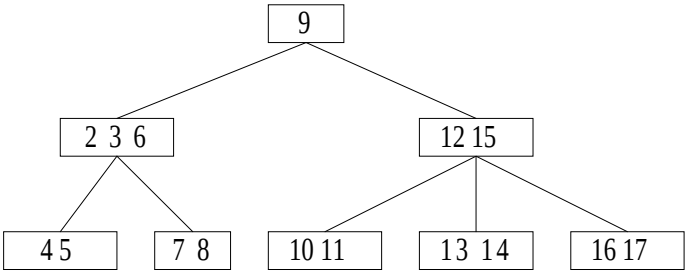


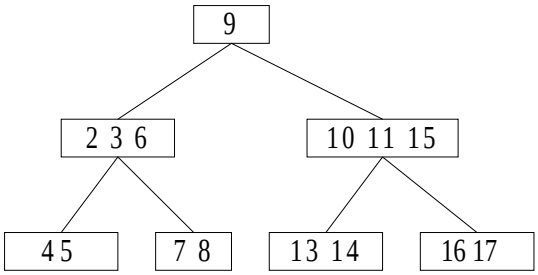
II. 1)





2)




9
2 3 6
10 11 15
4 5
7 8
13 14
16 17