Organisation des Fichiers

- Introduction
- Structures simples
- Les méthodes d'index
- Organisation M-Arbres
- Les méthodes de hachage

Une introduction aux structures de fichiers:

- **Vue logique du fichier**

Le fichier est un ensemble d'articles. A chaque article est associée une clé qui l'identifie de façon unique. C'est ce qu'on appelle la clé primaire. Les autres champs peuvent constituer des clés secondaires (identification non unique).

- **Vue physique du fichier**

Physiquement le fichier est un ensemble de blocs. Un bloc contient n articles. Un article contient n champs. la longueur de l'article peut être fixe ou variable. Le bloc constitue généralement l'unité de transfert entre la RAM et la mémoire secondaire.

- **Méthode d'accès**

C'est l'approche utilisée pour localiser un article dans un fichier. En général, on peut classer les méthodes en deux catégories : l'accès séquentiel et l'accès direct.

- **Organisation du fichier**

C'est la combinaison de structures physique et conceptuelle utilisées pour distinguer un article d'un autre, un champ d'un autre, etc.

- **Adressage**

Un bloc est repéré par son numéro. Un article est repéré par son adresse octet par rapport au début du fichier ou par le couple (numéro du bloc, déplacement dans le bloc).

Objectifs : Répondre aux deux questions suivantes :

Comment définir :

1. - l'organisation du fichier : séparation des champs, des articles, ..
2. - la méthode d'accès : comment localiser un article sur le fichier

Ça revient donc à étudier les opérations de base :

- recherche
- insertion
- suppression
- requête à intervalle

sous différentes formes d'accès qui sont :

- fichier vu comme un tableau
- fichier vu comme une liste linéaire chaînée
- les méthodes d'index
- l'accès multi-clés
- les méthodes d'arbres : arbres de recherche m-aires et arbres B
- les méthodes de hachage.

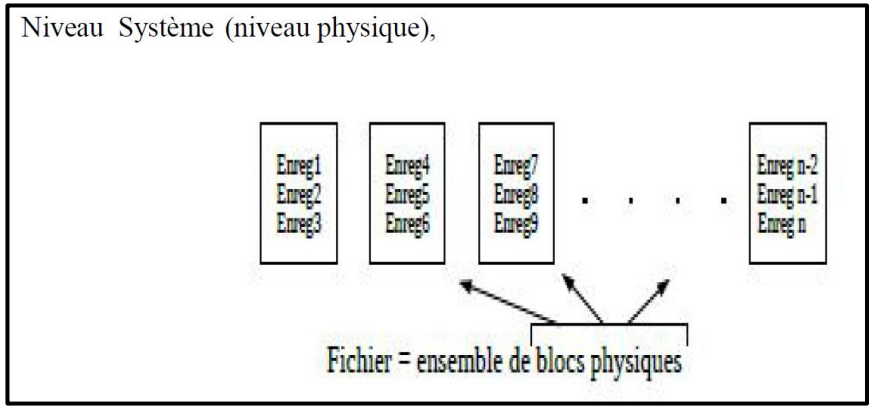
Introduction

On désigne par organisation, la manière dont sont organisés physiquement les enregistrement du fichier sur le support. Elle permet d'attribuer un emplacement sur le support physique pour tout enregistrement logique du fichier.

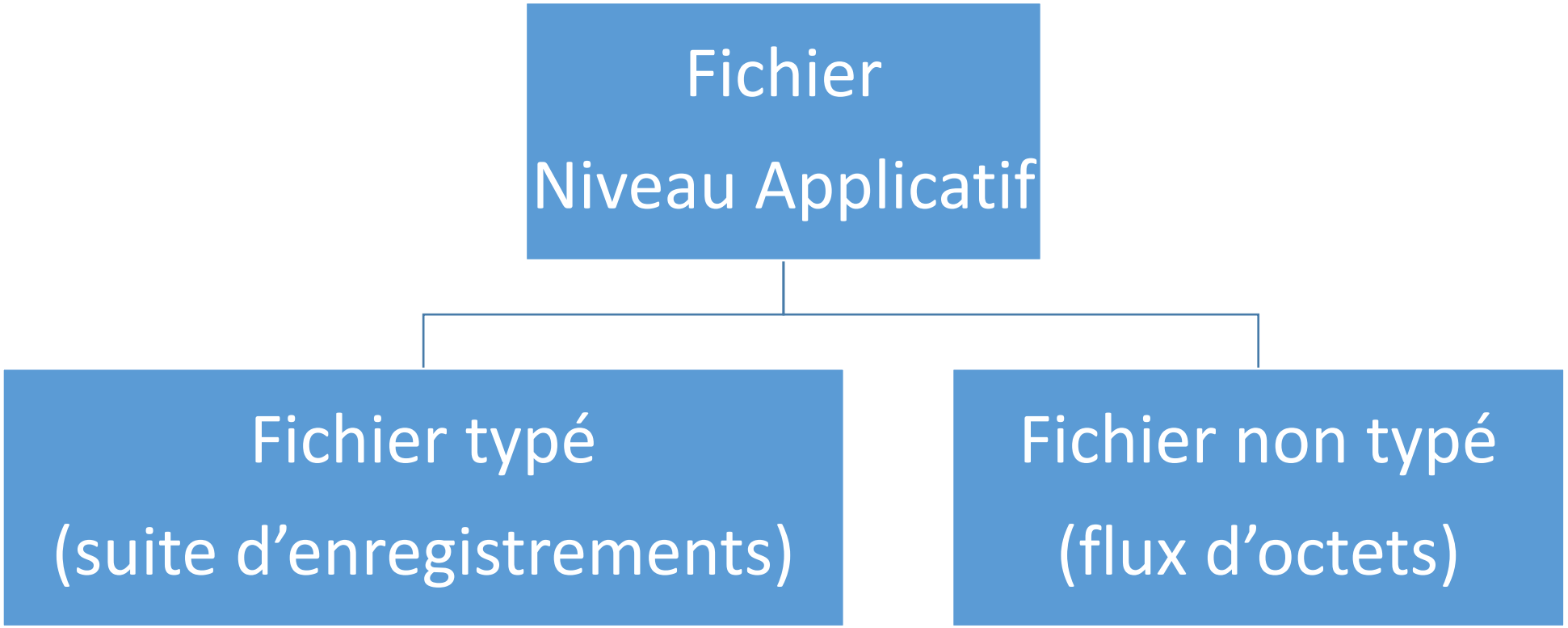
Un fichier est le concept à travers lequel, un programme ou une application stocke des données en MS. Les fichiers sont utilisés à différents niveaux d'abstraction avec des sémantiques différentes:

Niveau Application

Niveau Système



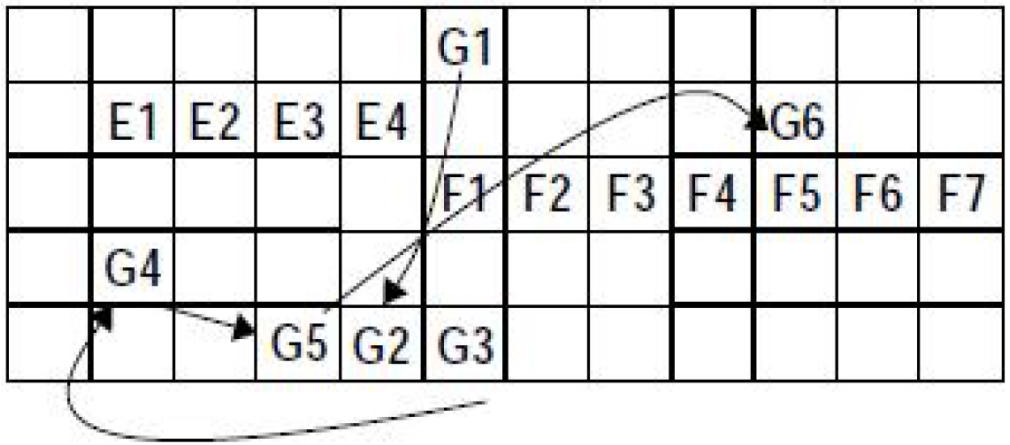
- Les fichiers typés est un ensemble d'enregistrements (ou articles). L'accès aux enregistrements se fait via des opérations spéciales pour les fichiers (ouverture, lecture …).
- On peut aussi définir un fichier comme étant une suite d'octets, on parle alors de flux (ex en C : FILE *). C'est des fichiers « non typés ». Ces flux (streams) généralisent la notion d'E/S pour être indépendante du type de périphérique utilisé (disques, imprimantes, écrans, clavier, réseaux, …).



Remarque : Toutes les opérations du niveau 'application' (niveau logique) passent par le système qui les traduit en opérations de bas niveau pour accéder physiquement aux blocs d'E/S. Comme les opérations d'E/S sont coûteuses (en temps), le système maintient en MC une zone spéciale de taille limitée (la zone tampon ou "buffer cache") lui permettant de garder en MC les copies de quelques blocs physiques choisis selon certaines stratégies (par exemple les plus utilisés). Cette zone tampon est totalement transparentes aux programmes d'application qui utilisent les fichiers.

Exemple : une application demande la lecture d'un enregistrement donné, le système vérifie d'abord si le bloc concerné ne se trouve pas déjà en MC (dans la zone tampon). S'il y est, l'enregistrement cherché sera directement transmis à l'application, sans qu'il y ait de lecture physique.

Modélisation des fichiers : on modélise la MS par une zone contiguë de blocs numérotés séquentiellement (ces numéros représentent les adresses de blocs).Les blocs sont des zones contiguës d'octets de même taille, renfermant, entre autre, les données (enregistrements) des fichiers.



Modélisation des fichiers

Pour écrire des algorithmes sur les structures de fichiers on utilisera la machine abstraite définie par le modèle suivant:

{ouvrir, fermer, lireDir, ecrireDir, lireSeq, ecrireSeq, aff_entete, entete, allocbloc }

Dans ce modèle, on manipule des numéros de blocs relatifs au début de chaque ficher (adresses logiques). L'utilisation des adresses physiques n'est pas d'une utilité particulière à ce niveau. Un fichier est donc un ensemble de blocs **numérotés logiquement (1, 2, 3, ... n)**

Caractéristiques et bloc d'en-tête

Pour que le système puisse gérer un fichier, il a besoin de connaître les informations sur ses caractéristiques : les blocs utilisés par le fichier, l'organisation du fichier, les droit d'accès associés, ...

pour un certain type de système de fichier , le bloc 0 pourrait être réservé pour contenir une table où chaque ligne renseigne sur les caractéristiques d'un fichier (nom, taille, blocs utilisés, ...). Quand une application désire ouvrir un fichier de nom donné, le système récupère ses informations à partir de cette table.

Méthode d'accès

Une structure de fichier consiste à définir une organisation des blocs d'un fichier sur MS afin d'implémenter certaines opérations d'accès pour la manipulation des données du fichier.

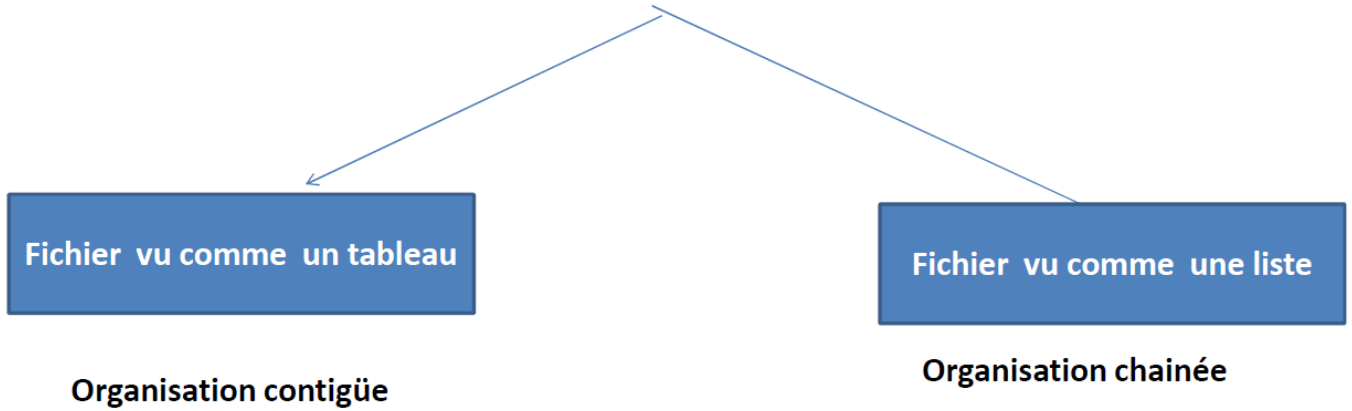
Une structure de fichier (méthode d'accès) consiste à définir :

- une manière d'organiser les blocs d'un fichier sur MS
- le placement des enregistrements à l'intérieur des blocs
- l'implémentation des opérations d'accès (recherche, insertion, suppression, ...).

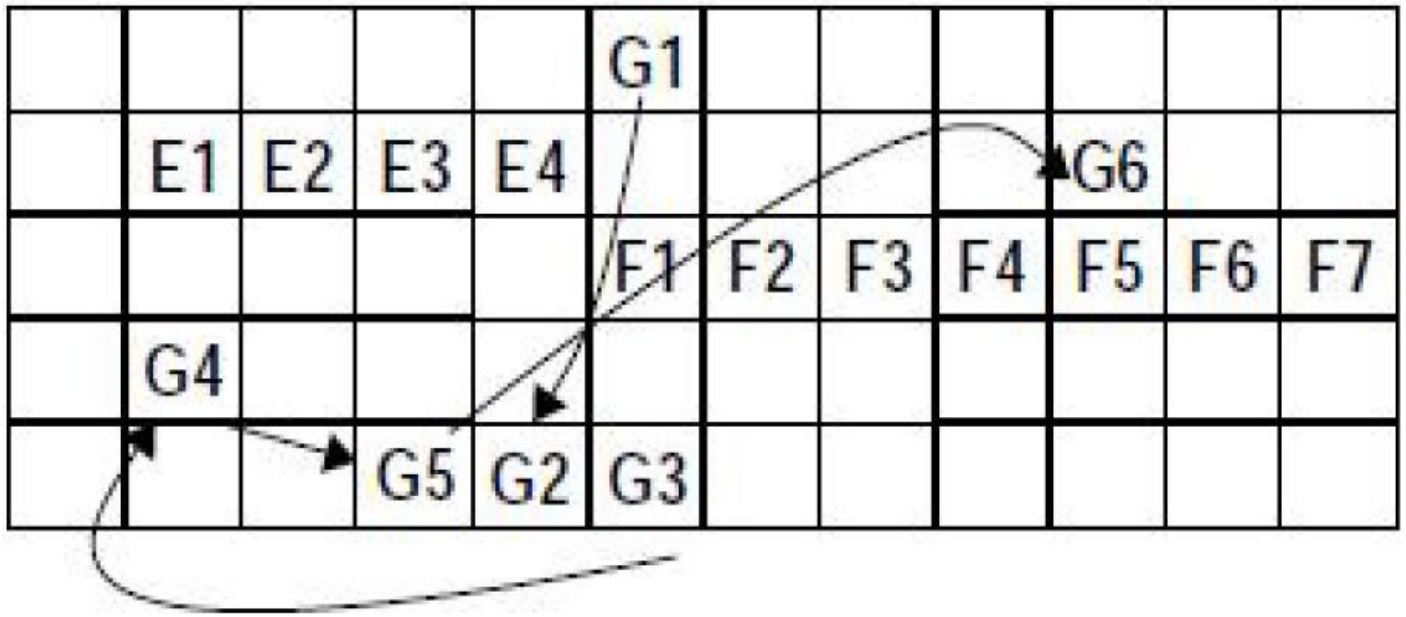
Le but des structures de fichiers est d'optimiser les performances d'accès (temps d'exécution et occupation mémoire).

Organisation globale des blocs

Dans un premier temps, on étudiera deux possibilités distinctes d'organiser les blocs au sein d'un fichier :



Dans la figure ci-dessous, on a deux fichiers vus comme tableau (E et F) et un fichier vu comme une liste (G). Les blocs F1, F2, … F7 sont contigus, de même pour les blocs E1, E2, E3 et E4. Par contre les blocs G1, G2, … G6 ne sont pas contigus, ils sont chaînés entre eux formant une liste de blocs.



Parmi les caractéristiques nécessaires pour manipuler un fichier vu comme **tableau**, on pourra avoir par exemple :

- Le numéro du premier bloc,
- Le numéro du dernier bloc (ou alors le nombre de blocs utilisés).

Pour un fichier vu comme **liste chainée**, il suffirait par contre de connaître **le numéro du premier bloc** (la tête de la liste), car dans chaque bloc, il y a **le numéro du prochain bloc** (comme le champ suivant dans une liste). Dans le dernier bloc, le numéro du prochain bloc pourra être mis à une valeur spéciale (par exemple -1) pour indiquer la fin de la liste.

Organisation interne des blocs

Les blocs sont censés contenir les enregistrements d'un fichier. Ces derniers peuvent être :

Enregistrements de longueur fixe : Chaque bloc pourra alors contenir un tableau d'enregistrements de même type.

Enregistrements de longueur variable : Chaque enregistrement sera vu comme étant une chaîne de cararactère (de longueur variable).

- ✓ Il y a un ou plusieurs champs ayant **des tailles variables**,
- ✓ L**e nombre de champs varie** d'un enregistrement à un autre.

➢ Implantation

Manière d'organiser les informations sur le support.

- **Organisation contiguë(Consécutive)**: les enregistrements sont placés consécutivement sur le support il s'agit d'une obligation pour les supports non adressables.
- **Organisation chainée(Dispersé)** : ne peut s'effectuer que sur des supports **adressables.** A chaque enregistrement, on ajoute un pointeur sur l'adresse de l'enregistrement suivant et précédent. Les deux méthodes d'accès (séquentiel ou direct) sont possibles.

➢ Méthode d'accès

Moyen de retrouver un enregistrement sur le support.

- **Séquentiel :** Pour accéder à l'enregistrement m, il faut tout d'abord accéder aux enregistrements précédents. Cette méthode d'accès est utilisable quelque soit le type de mémoire de stockage (adressable ou non).
- **Direct :** Accès direct à l'enregistrement grâce à son adresse – ce qui implique une obligation d'utiliser des mémoires adressables.

Typologie des organisations

L'organisation est une composition de l'implantation et de la méthode d'accès.

- **Séquentielle**: implémentation consécutive – accès séquentiel.
- **Séquentielle indexée** : combinaison d'implémentation consécutive et accès séquentiel ou sélectif. Chaque fois qu'un article est écrit, une clé lui est associée et une adresse de l'article dans le fichier sont inscrites dans une table appelée table d'indexe. Lors d'un accès au fichier, on sélectionne l'enregistrement dans la table d'index puis on effectue un accès Direct.
- **Directe** : implémentation directe – accès direct : L'indicatif de chaque article est traduit en adresse physique en appliquant une fonction de randomisation – algorithme de calcul appliqué lors de la création du fichier ; il y a un risque de collision, c'est à dire que deux indicatifs différents "donnent" le même enregistrement.

Si on est intéressé par des enregistrements de longueur fixe, chaque bloc pourra alors contenir un tableau d'enregistrements de même type.

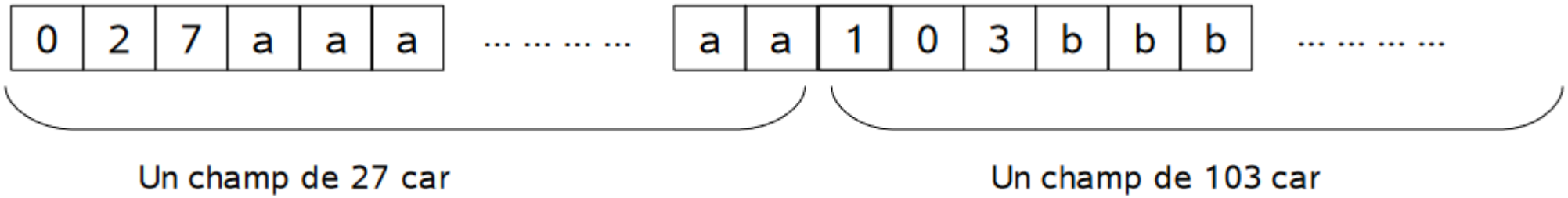
Ex:

```
Type Tenreg = structure
matricule : chaine(10);
nom : chaine(20);
age : entier;
...
fin;
    Type Tbloc = structure
    tab : tableau[1..b] de Tenreg; // tab pouvant contenir (au max) b enreg.
    NB : entier; // nb d'enregistrements insérés dans tab.
    fin;
```

Si on opte pour des enregistrements de tailles variables, chaque enreg sera vu comme étant une chaîne de car (de longueur variable).

Les enreg seront de longueur variable, car par exemple, il y a un ou plusieurs champs ayant des tailles variables, ou alors le nombre de champs varie d'un enreg à un autre.

Pour séparer les champs entre eux (à l'intérieur de l'enregistrement), on pourra soit utiliser un caractère spécial ('#') ne pouvant pas être utilisé comme valeur légale, ou alors préfixer le début des champs par leur taille (sur un nombre fixe de positions). Dans la fig ci-dessous, on utilise 3 positions pour indiquer la taille des champs.



Le bloc ne peut pas être défini comme étant un tableau d'enregistrements, car les éléments d'un tableau doivent toujours être de même taille. La solution c'est de considérer le bloc comme étant (ou contenant) une grande chaîne de caractères renfermant les différents enregistrements (stockés caractère par caractère).

Pour séparer les enregistrements entre eux, on utilise les mêmes techniques que celles utilisées dans la séparation entre les champs d'un même enregistrement (soit avec un caractère spécial '$', soit on préfixe chaque enregistrement par sa taille). Voici un exemple de déclaration d'un type de bloc pouvant être utilisé dans la définition d'un fichier vu comme liste avec format (taille) variable des enregistrements.

```
Type Tbloc = structure
tab : tableau[1..b'] de caractères; // tableau de car pour les enreg.
suiv : entier; // num du bloc suivant dans la liste
fin;
```

remarque: même si les enregistrements sont de longueurs variables, la taille des blocs reste toujours fixe.

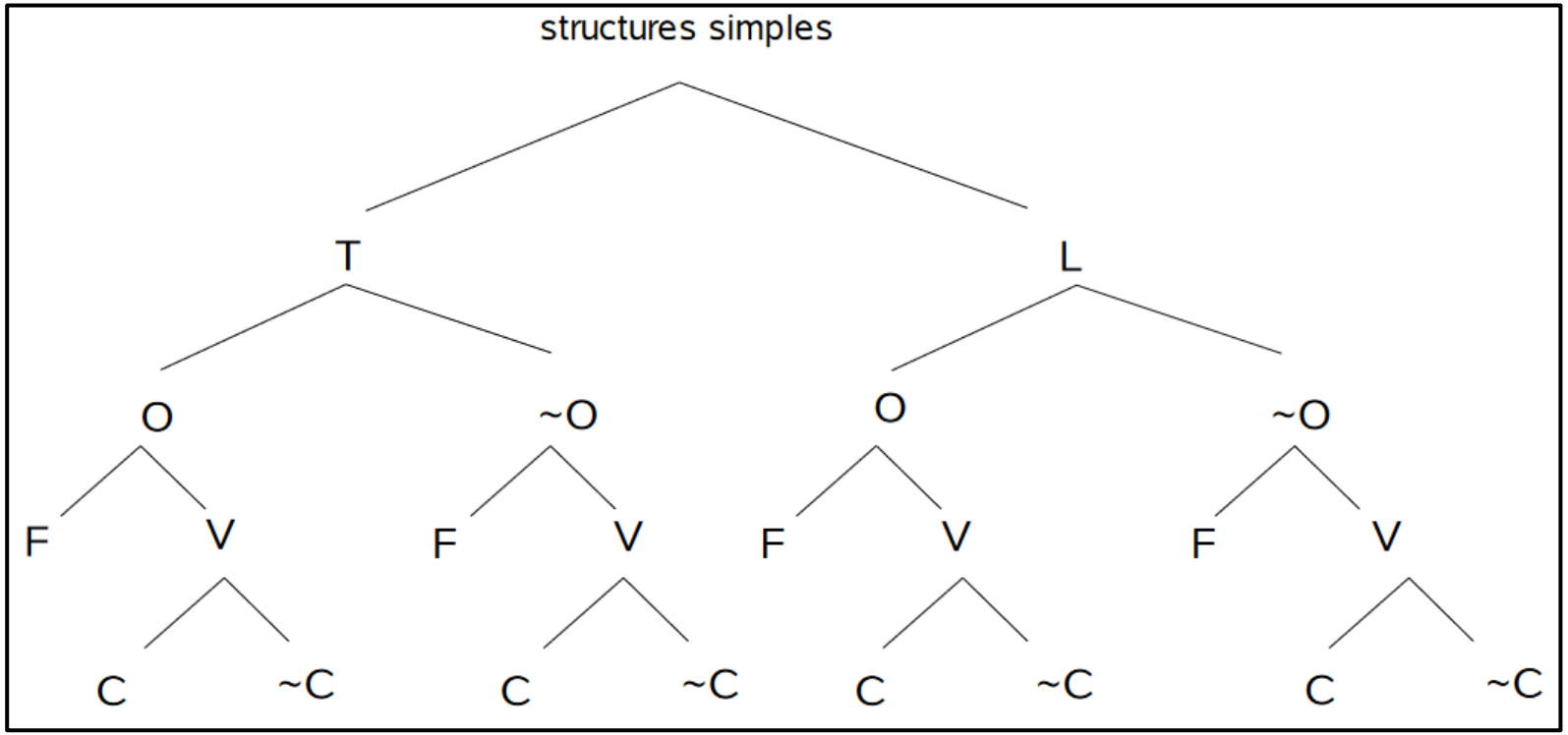
Chevauchement

Pour minimiser l'espace perdu dans les blocs (dans le cas : format variable uniquement), on peut opter pour une organisation avec chevauchement entre deux ou plusieurs blocs. Quand on veut insérer un nouvel enregistrement dans un bloc non encore plein et où l'espace vide restant n'est pas suffisant pour contenir entièrement cet enregistrement, celui-ci sera découpé en 2 parties de telle sorte à occuper tout l'espace vide du bloc en question par la 1ere partie, alors que le reste (la 2e partie) sera insérée dans un nouveau bloc alloué au fichier. On dit alors que l'enregistrement se trouve à cheval entre 2 blocs.

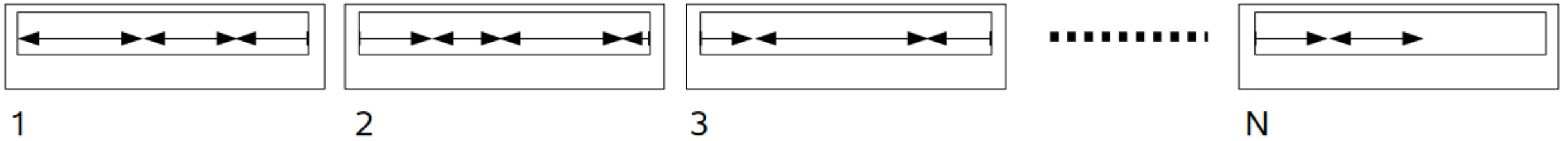
Taxonomie des structures simples de fichiers

En combinant entre l'organisation globale des fichiers (tableau ou liste) et celle interne aux blocs (format fixe ou variable des enregistrements), on peut définir une classe de méthodes d'accès (dites « simples ») pour organiser des données sur disque. Si de plus on prend en compte la possibilité de garder le fichier ordonné ou non, suivant les valeurs d'un champ clé particulier, on doublera le nombre de méthodes dans cette classe de structures simples de fichiers.

utilisant la notation suivante: T (pour fichier vu comme tableau), L (pour liste), O (pour fichier ordonné), ~O (non ordonné), F (pour format fixe des enreg), V (pour format variable), C (pour chevauchement des enreg entre blocs), ~C(pour pas de chevauchement), les feuilles de l'arbre suivant, représentent les 12 méthodes d'accès simples:

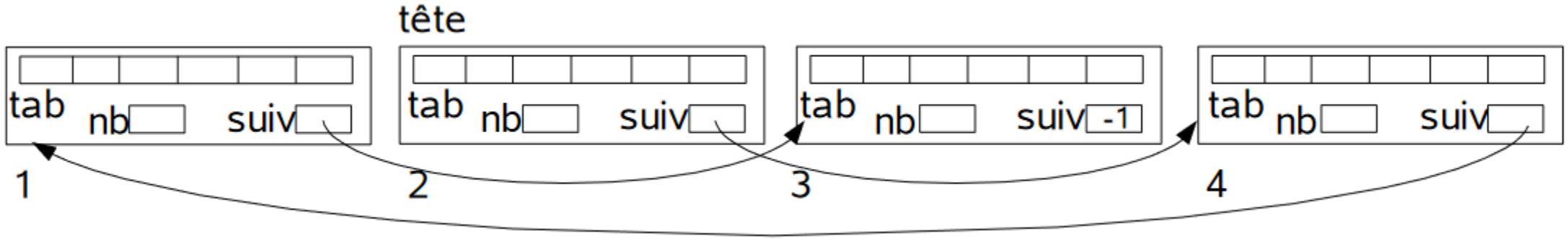


Par exemple la méthode T~OVC représente l'organisation d'un fichier vu comme tableau, non ordonné, avec des enregistrements de taille variables et acceptant les chevauchements entre blocs :



La recherche est séquentielle, l'insertion en fin de fichier et la suppression est logique.

Dans le cas d'un fichier LOF, chaque bloc contient un tableau d'enregistrements, un entier indiquant le nombre d'enregistrements dans le tableau et un entier pour garder la trace du bloc suivant dans la liste :



La recherche est séquentielle, l'insertion provoque des décalages intra-blocs et la suppression est physique.

Exemple complet: **fichier de type « TOF » :** (fichier vu comme tableau, ordonné, et enregistrements à taille fixe)

- La recherche d'un enregistrement est dichotomique (rapide).
- L'insertion peut provoquer des décalages intra et inter-blocs (coûteuse).
- La suppression peut être réalisée par des décalages inverses (suppression physique coûteuse) ou alors juste par un indicateur booléen (suppression logique beaucoup plus rapide). Optant pour cette dernière alternative.
- On commence généralement par faire un chargement initial du fichier en laissant un peu de vide dans chaque bloc, afin de minimiser les décalages pouvant être provoqués par les futures insertions => C'est l'opération de chargement initial des fichiers ordonnés.
- Avec le temps, le facteur de chargement du fichier (nombre d'insertions / nombre de places disponibles dans le fichier) augmente à cause des insertions futures, de plus les suppressions logiques ne libèrent pas de places. Donc les performances se dégradent avec le temps. Il est alors conseillé de réorganiser le fichier en procédant à un nouveau chargement initial => C'est l'opération de réorganisation

Déclaration du fichier:

```
const
b = 30; // capacité maximale des blocs (en nombre d'enregistrements)
type
Tenreg = structure
          effacé : booleen; // booléen pour la suppression logique
          cle : typeqlq; // le champs utilisé comme clé de recherche
          champ2 : typeqlq; // les autres champs de l'enregistrement,
          champ3 : typeqlq; // sans importance ici.
          ...
Fin;
Tbloc = structure // le bloc renferme :
       tab : tableau[1..b] de Tenreg; // un tableau d'enreg d'une capacité b
       NB : entier; // nombre d'enreg dans tab ( <= b)
Fin;
var // globales
       F : Fichier de Tbloc Buffer buf Entete (entier, entier);
```

```
/*
        L'entête contient deux caractéristiques:
        - la première sert à garder la trace du nombre de bloc utilisés (ou alors le numéro logique du dernier bloc du fichier)
        - la deuxième servira comme un compteur d'insertions pour pouvoir calculer rapidement le facteur de chargement, et
        donc voir s'il y a nécessité de réorganiser le fichier.
*/
```

Module de recherche: (dichotomique)

en entrée la clé à chercher et le nom externe du fichier.
en sortie le booleen Trouv, le num de bloc (i) contenant (c) et le déplacement (j)

Rech(c:typeqlq; nomfich:chaine; var Trouv:bool; var i,j:entier)

```
var
        bi, bs, inf, sup : entier;
        trouv, stop : booleen;
DEBUT
        Ouvrir( F, nomfich, 'A' );
        bs := entete( F,1 ); // la borne sup (le num du dernier bloc de F)
        bi : = 1; // la borne inf (le num du premier bloc de F)
        Trouv := faux; stop := faux; j := 1;
        TQ ( bi <= bs et Non Trouv et Non stop )
        i := (bi + bs) div 2; // le bloc du milieu entre bi et bs
        LireDir( F, i, buf );
```

```
SI ( c >= buf.tab[1].cle et c <= buf.tab[buf.NB].cle )
        // recherche dichotomique à l'intérieur du bloc (dans la variable buf)...
        inf := 1; sup := buf.NB;
        TQ inf <= sup et Non Trouv
                        j : = (inf + sup) div 2;
                        SI c = buf.tab[j].cle: Trouv := vrai
                        SINON
                                SI c < buf.tab[j].cle: sup := j-1
                                SINON inf := j+1
                                FSI
                        FSI
        FTQ
        SI ( Non Trouv )
                j := inf
        FSI
        // fin de la recherche interne. j indique l'endroit où devrait se trouver c
        stop := vrai
```

```
SINON // non ( c >= buf.tab[1].cle et c <= buf.tab[buf.NB].cle )
        SI ( c < buf.tab[1].cle )
                bs := i-1
        SINON // c > buf.tab[buf.NB].cle
                bi := i+1
        FSI
FSI
FTQ
SI ( Non Trouv )
        i := bi
FSI
fermer( F )
FIN
```

Module d'insertion: (avec éventuellement des décalages intra et inter blocs)

Inserer(e:Tenreg; nomfich:chaine)

```
var
      trouv : booleen;
      i,j,k : entier;
      e,x : Tenreg;
DEBUT
            // on commence par rechercher la clé e.cle avec le module précédent pour localiser l'emplacement (i,j)
            // où doit être insérer e dans le fichier.
Rech( e.cle, nomfich, trouv, i, j );
            SI ( Non trouv )                  // e doit être inséré dans le bloc i à la position j
            Ouvrir( F,nomfich, 'A');       // en décalant les enreg j, j+1, j+2, ... vers le bas
            continu := vrai;
                                                      // si i est plein, le dernier enreg de i doit être inséré dans i+1
            TQ ( continu et i<= entete(F,1) )        // si le bloc i+1 est aussi plein son dernier enreg sera
                  LireDir( F, i, buf );                  // inséré dans le bloc i+2, etc ... donc une boucle TQ.
                  // avant de faire les décalages, sauvegarder le dernier enreg dans une var x ...
                  x := buf.tab[buf.NB];
                  // décalage à l'intérieur de buf ...
                  k := buf.NB;
                  TQ k > j
                  buf.tab[k] := buf.tab[k-1];
                  k := k-1
                  FTQ
```

```
        buf.tab[j] := e;          // insérer e à la pos j dans buf ...
                                  // si buf n'est pas plein, on remet x à la pos NB+1 et on s'arrête ...
        SI ( buf.NB < b )               // b est la capacité max des blocs (une constante)
                    buf.NB := buf.NB+1;
                    buf.tab[buf.NB] := x;
                    EcrireDir( F, i, buf );
                    continu := faux;
        SINON                     // si buf est plein, x doit être inséré dans le bloc i+1 à la pos 1 ...
        EcrireDir( F, i, buf );
                    i := i+1;
                    j := 1;
                    e := x; // cela se fera (l'insertion) à la prochaine itération du TQ
        FSI // non ( buf.NB < b )
FTQ
SI i > entete( F, 1 )   // si on dépasse la fin de fichier, on rajoute un nouveau bloc contenant un seul enregistrement e
        buf.tab[1] := e;
        buf.NB := 1;
        EcrireDir( F, i, buf ); // il suffit d'écrire un nouveau bloc à cet emplacement
        Aff-entete( F, 1, i ); // on sauvegarde le num du dernier bloc dans l'entete 1
FSI
Aff-entete( F, 2 , entete(F,2)+1 ); // on incrémente le compteur d'insertions
Fermer( F );
FSI
FIN
```

La suppression logique consiste à rechercher l'enregistrement et positionner le champs 'effacé' à vrai

Suppression(c:typeqlq; nomfich:chaine)

```
Var     trouv : booleen;  i,j : entier;
DEBUT
        // on commence par rechercher la clé c pour localiser l'emplacement (i,j) de l'enreg à supprimer
        Rech( c, nomfich, trouv, i, j );
        // ensuite on supprime logiquement l'enregistrement
        SI ( trouv )
        Ouvrir( F,nomfich, 'A');
        LireDir( F, i, buf ); // lecture pas vraiment nécessaire à cause de l'effet de bord de Rech sur buf
        buf.tab[j].effacé := VRAI;
        EcrireDir( F, i, buf );
        Fermer( F )
        FSI
FIN // suppression
```

Le chargement initial d'un fichier ordonné consiste à construire un nouveau fichier contenant dès le départ n enregistrements. Ceci afin de laisser un peu de vide dans chaque bloc, qui pourrait être utilisé plus tard par les nouvelles insertions tout en évitant les décalages inter-blocs (très coûteux en accès disque) :

```
Chargement_Initial( nomfich : chaine; n : entier; u : reel )
// u est un réel compris entre 0 et 1 et désigne le taux de chargement voulu au départ
Var   e : Tenreg;  i,j,k : entier;
DEBUT
Ouvrir( F, nomfich, 'N' ); // un nouveau fichier
i := 1; // num de bloc à remplir
j := 1; // num d'enreg dans le bloc
ecrire( 'Donner les enregistrements en ordre croissant suivant la clé : ' );
POUR k:=1 , n
        lire( e );
        SI j <= u*b // ex: si u=0.5, on remplira les bloc jusqu'à b/2 enreg
        buf.tab[j] := e
        j := j+1;
        SINON // j > u*b : buf doit être écrit sur disque
```

```
        SINON // j > u*b : buf doit être écrit sur disque
            buf.NB = j-1;
            EcrireDir( F, i, buf );
            buf.tab[1] := e; // le kème enreg sera placé dans le prochain bloc, à la position 1
            i := i+1;
            j := 2;
        FSI
FP
// à la fin de la boucle, il reste des enreg dans buf qui n'ont pas été sauvegardés sur disque
buf.NB := j-1;
EcrireDir( F, i, buf );
// mettre à jour l'entête (le num du dernier bloc et le compteur d'insertions)
Aff-entete( F, 1, i );  Aff-entete( F, 2, n );
Fermer( F )
FIN // chargement-initial
```

La réorganisation du fichier consiste à recopier les enreg vers un nouveau fichier de telle sorte à ce que les nouveaux blocs contiennent un peu de vide (1-u). Cette opération ressemble au chargement initial sauf que les enregistrements sont lus à partir de l'ancien fichier.

- Structures simples
- **Les méthodes d'index**
- **L'accès multi-clés**

Si l'opération de recherche est très fréquente pour une application donnée, la meilleure alternative parmi les structures simples de fichiers est le choix d'un tableau ordonné avec un format fixe des articles. Par conséquent, la méthode se trouve limitée à des fichiers dont la taille devienne volumineuse.

→ Les opérations d'accès (recherche, insertion, ...) deviennent très inefficaces.

→ Les méthodes d'index permettent d'améliorer, dans une certaine mesure, les performances en gérant une structure auxiliaire (table d'index) accélérant la recherche.

En l'absence d'un index, seules solutions :

- **Parcours séquentiel (complexité linéaire)**
- **Recherche par dichotomie si fichier trié (complexité logarithmique)**

Avec un index :

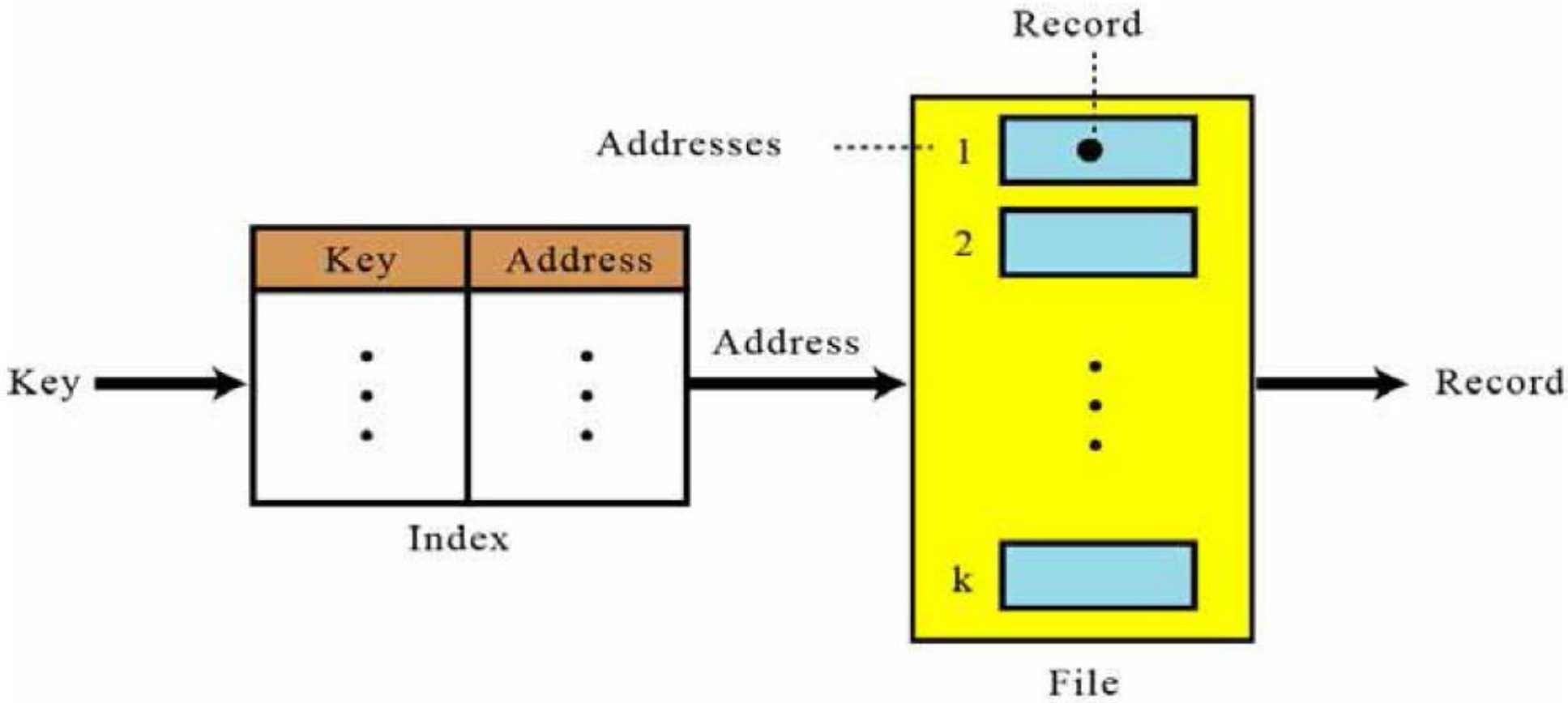
Parcours de l'index, puis accès direct à l'enregistrement Mais attention : mises à jour plus coûteuses !!

Un **index** est (généralement) une table ordonnée contenant les couples <clé, adr > utilisée pour accélérer la recherche des enregistrements d'un fichier.
Pour ces méthodes, deux fichiers sont utilisés : *le* ***fichier d'index*** et *le* ***fichier au données***. En règle générale, le fichier de données n'est pas trié. Il peut être un tableau ou une liste linéaire chaînée de blocs sur le disque. Le **fichier d'index est supposé en mémoire et est trié selon les clés des articles**. L'index peu être à un ou plusieurs niveaux.

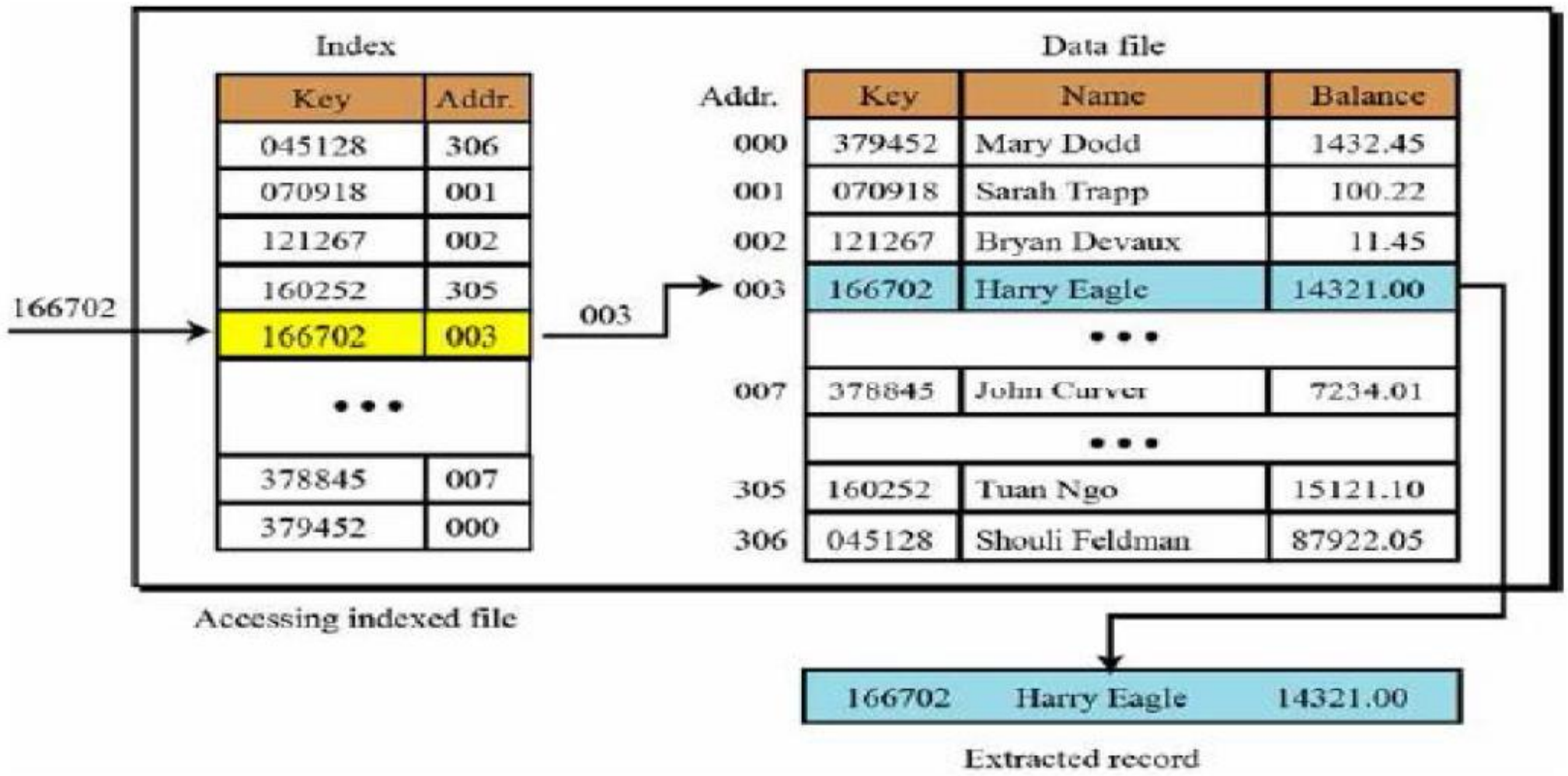
Exemple 1:

L'utilisation des index est basée sur l'observation suivante: pour trouver un livre dans une bibliothèque, au lieu d'examiner un par un chaque livre (ce qui correspond à une recherche séquentielle), il est plus rapide de consulter le catalogue où ils sont classés par thème, auteur et titre.
Chaque entrée d'un index comporte une valeur extraite des données et un pointeur sur son emplacement d'origine. Un enregistrement peut être ainsi facilement retrouvé en recherchant sa localisation dans l'index

Exemple 2:


Record
Addresses
1
2
k
Key
Address
Key
Address
Record
Index
File

Exemple 3:


Index
Key
Addr.
045128
306
070918
001
121267
002
160252
305
166702
003
378845
007
379452
000
166702
003
Data file
Addr.
Key
Name
Balance
000
379452
Mary Dodd
1432.45
001
070918
Sarah Trapp
100.22
002
121267
Bryan Devaux
11.45
003
166702
Harry Eagle
14321.00
007
378845
John Curver
7234.01
305
160252
Tuan Ngo
15121.10
306
045128
Shouli Feldman
87922.05
Accessing indexed file
166702 Harry Eagle 14321.00
Extracted record

Avantages des méthodes d'index

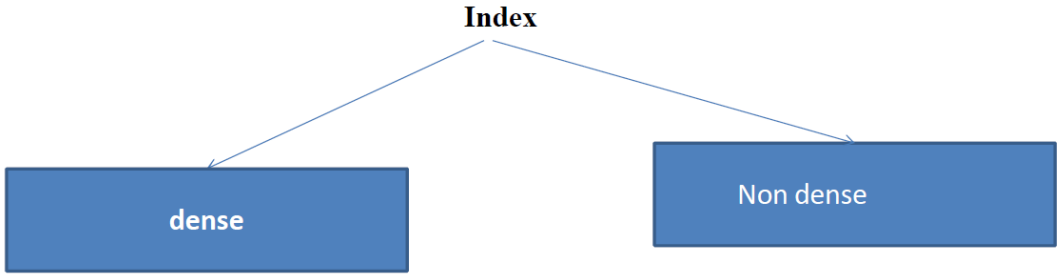
Même si l'index n'est pas en mémoire, les méthodes d'index ont les avantages suivants :

- Permettent la recherche dichotomique même pour les articles de longueur variable.
- Faire la recherche dichotomique sur le fichier d'index est beaucoup plus rapide que sur le fichier de données lui-même.
- L'effacement des articles par des "flag" se fait sans accès disque.

Inconvénients des méthodes d'index

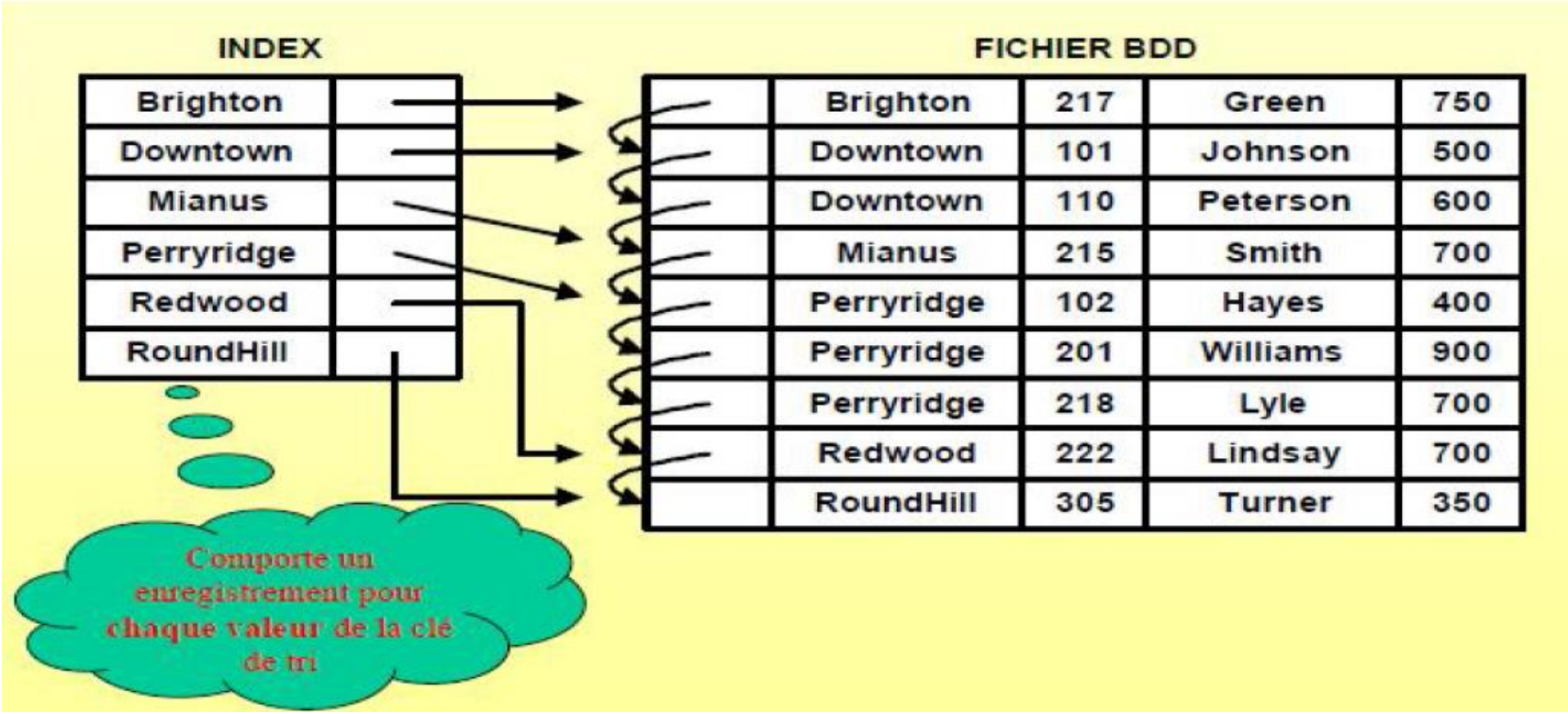
- Si l'index est large pour être contenu en mémoire.
- Recherche binaire chère.
- Réarrangement coûteux à cause des décalages; on a recours à d'autres techniques : Arbre, hachage.

Typologies selon la dense du clé



Index dense : Comporte **un enregistrement pour *chaque valeur de la clé* de tri du fichier indexé**. Avec le temps, ce type d'index peut occuper beaucoup d'espace mémoire. Mais, ils ont un temps de recherche très court.

Exemple d'index dense :

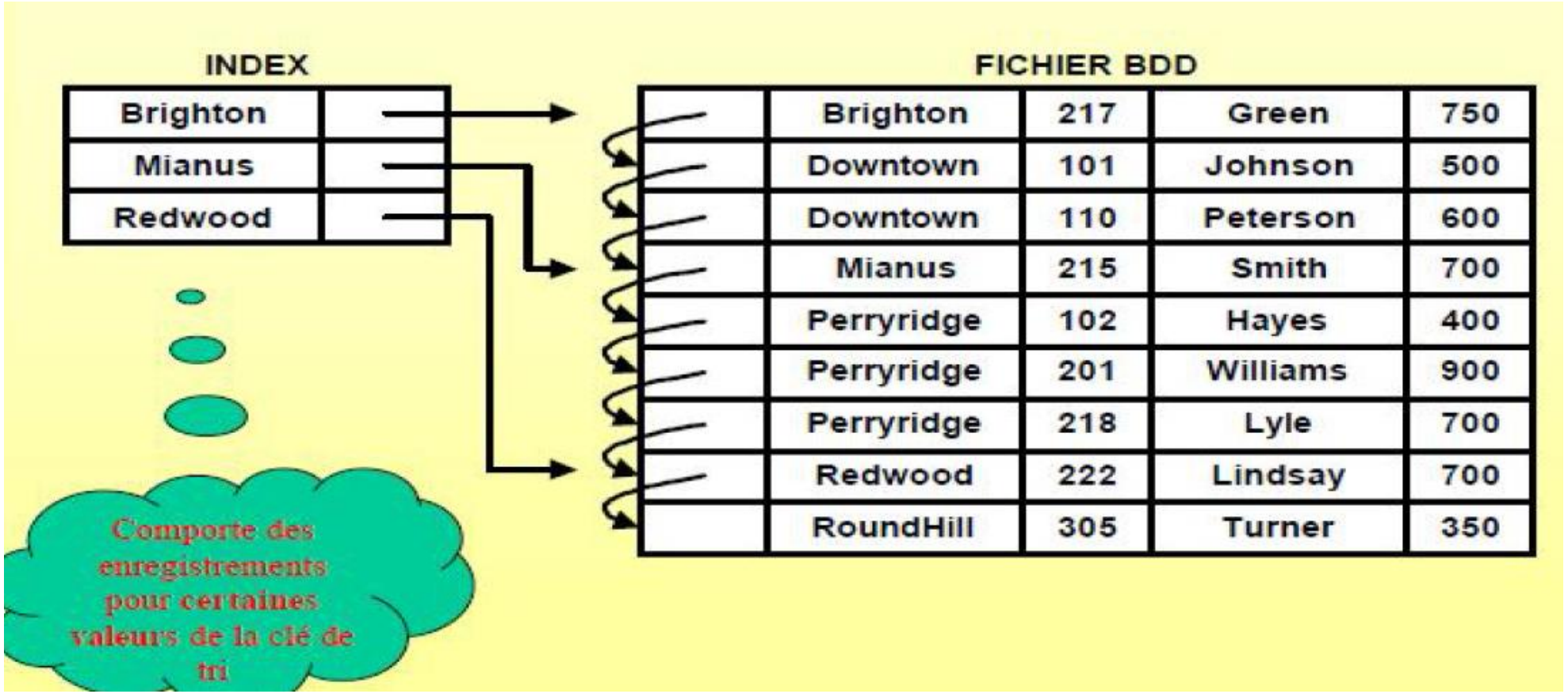


Index non dense :

Comporte des enregistrements pour certaines valeurs de la clé de tri du fichier indexé.

Les index non dense ont des avantages :

- Ils occupent **moins d'espace;**
- Ils imposent moins de servitudes lors d'insertions et suppressions.



Exemple d'index non dense

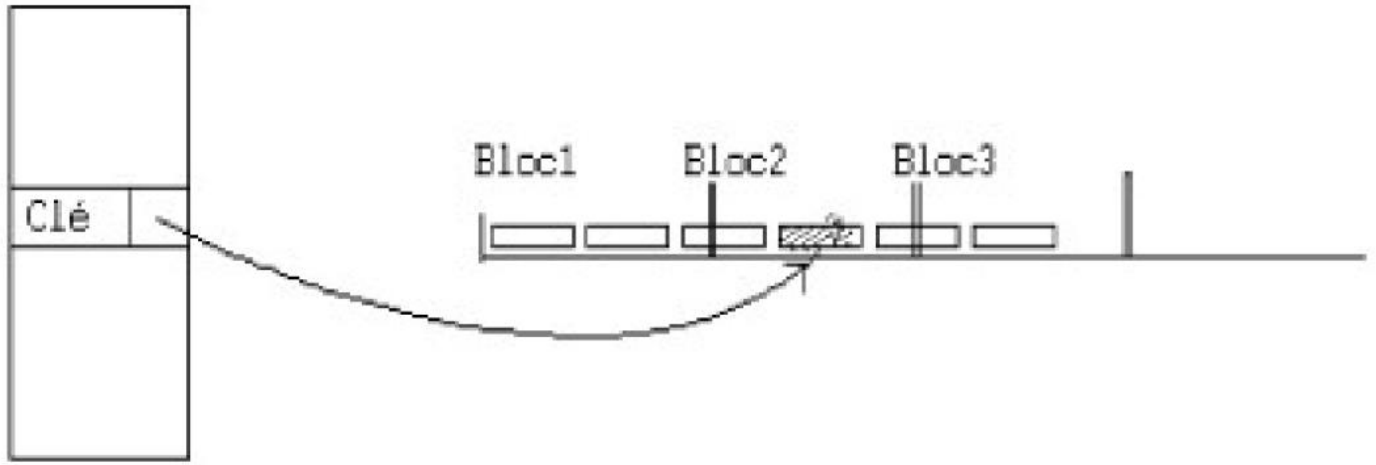
Typologies selon la nature du clé

Index primaire

Si le champ clé ne contient de valeurs en double, l'index est alors « primaire »

index primaire à un niveau

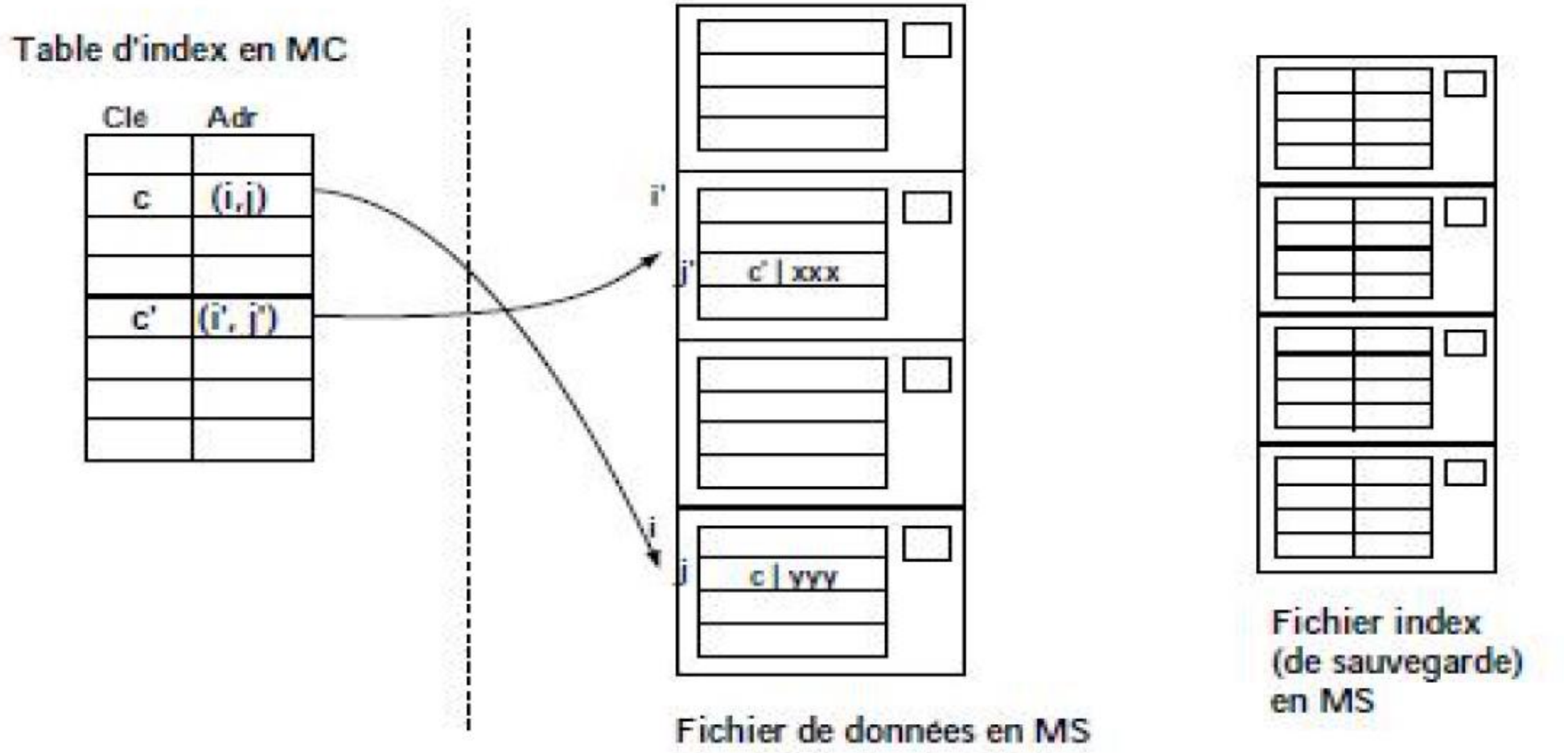
L'index est un ensemble de couples(clé, adresse) rangés entièrement en mémoire centrale. Le fichier est un ensemble de blocs (dans l'exemple les blocs sont contigus) sur le disque. Le bloc contient un ensemble d'articles. les articles peuvent être à cheval sur deux blocs logiquement consécutifs. C'est le cas de la figure suivante :



Index ordonné **Fichier de données non ordonné**

index primaire à un niveau

Les fichiers de données et d'index peuvent être de n'importe quelle structure (blocs contigus, blocs chaînés, ...). De même que les enregistrements peuvent être à format fixe ou variable (avec ou sans chevauchement).



Operations de base

Pour mettre au point une méthode d'index, les opérations suivantes sont nécessaires :

- **créer un fichier d'index et de données**
- **charger le fichier d'index en mémoire avant utilisation**
- **rechercher un article de clé donnée,**
- **insérer un article,**
- **supprimer un article,**
- **modifier un article.**

Rechercher un article de clé donnée

Pour rechercher un article de clé donnée dans le fichier, on commence par faire une recherche dichotomique sur l'index.

- Si la clé est trouvée, le bloc contenant l'article est ramené en mémoire centrale afin de récupérer l'article.
- Si ce dernier est à cheval sur deux bloc, le deuxième bloc est ramené pour récupérer le reste de l'article.

Insérer un article

Une insertion d'article est réalisée comme suit :

- si la clé n'est pas trouvée dans l'index, elle est y insérée et peut donc entraîner des décalages.
- l'article est inséré en fin de fichier.

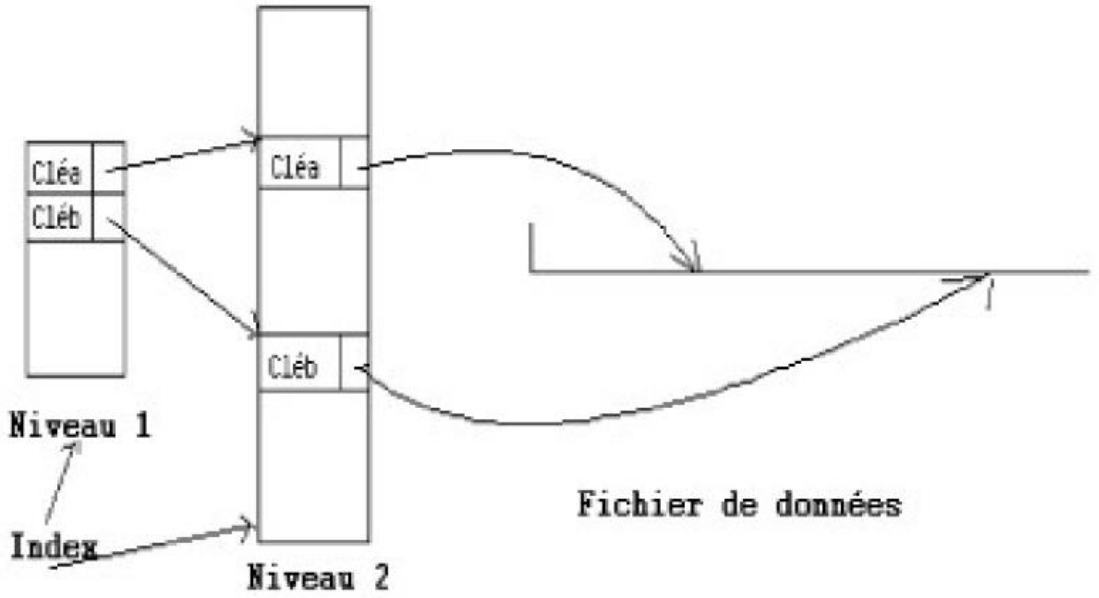
Supprimer un article

- Une suppression est généralement logique. Elle est donc extrêmement rapide.
- Une réorganisation est donc à prévoir afin d'éliminer physiquement les articles supprimés.

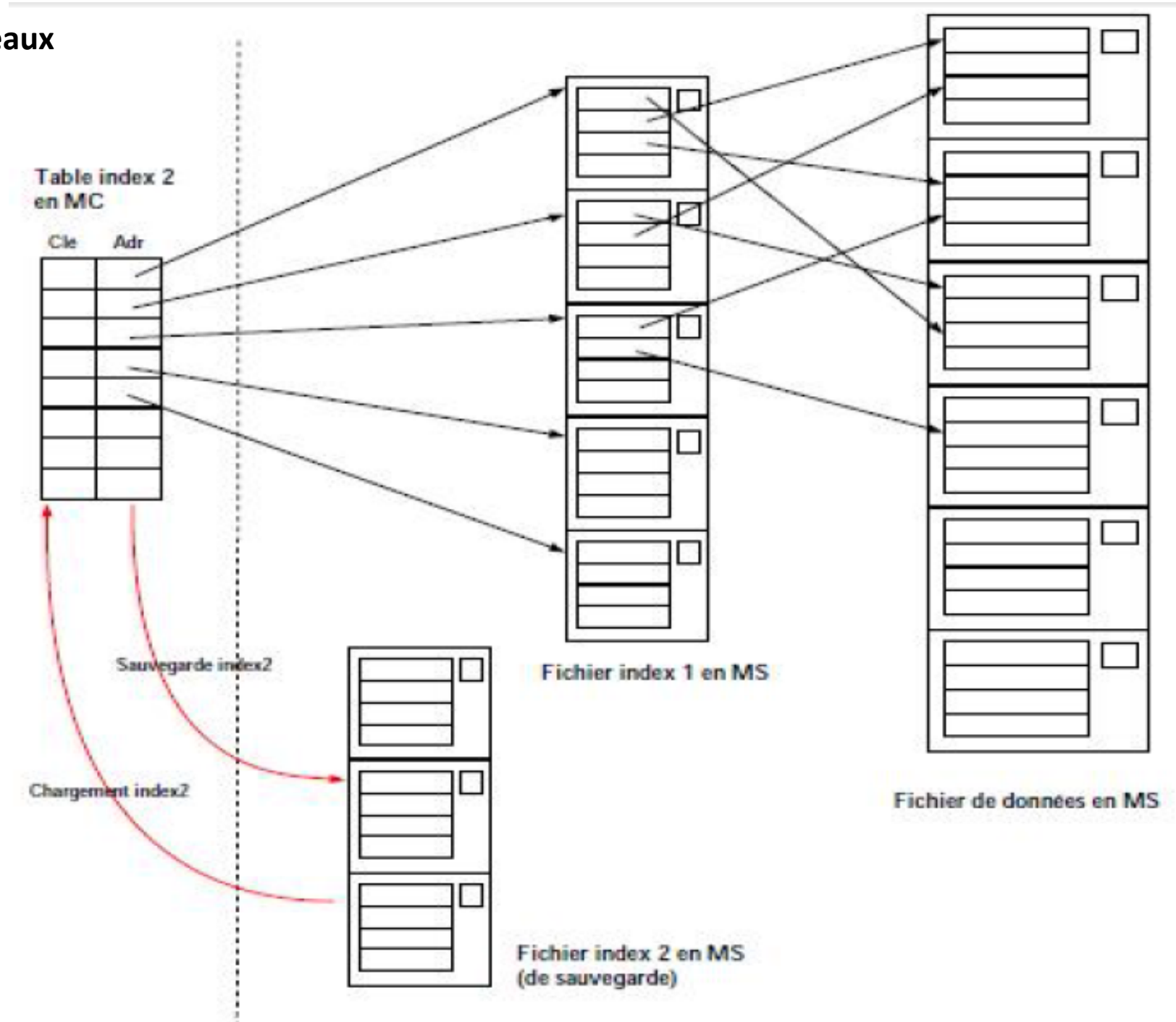
Généralement, on construit un autre fichier.

Index primaire à plusieurs niveaux

- Si l'index est trop grand pour résider en MC, on construit un deuxième index sur le fichier index (ordonné). Dans ce cas, on choisira une seule clé pour chaque bloc du fichier index (index non dense) pour construire le deuxième index.
- Si le deuxième index est encore trop grand pour résider en MC, on le stocke sur disque (deuxième fichier index) et on construit un troisième index en choisissant une clé par bloc du deuxième fichier index. On peut répéter ce procédé au tant que nécessaire.



Index primaire à plusieurs niveaux



Rechercher un article de clé donnée

Pour rechercher un article de clé donnée, on commence par rechercher sa clé dans l'index du niveau 1. un intervalle est alors sélectionné. La recherche continue dans l'index de niveau 2 uniquement dans cet intervalle. Si la clé est trouvée, on procède de la même façon que dans le cas avec un seul index. Il est clair que les deux recherches dichotomiques sur les deux petits vecteurs (index de niveau 1 et une partie de l'index de niveau 2) sont beaucoup plus rapides qu'une seule recherche dichotomique sur un seul grand vecteur (index de niveau2).

Propriétés des méthodes d'index

- Les méthodes d'index permettent d'améliorer, dans une certaine mesure, les performances en gérant une structure auxiliaire (table d'index) accélérant la recherche.
- Les méthodes d'index comme celles présentées dans ce chapitre, sont dédiées aux fichiers statiques (c-a-d dans les cas où le nombre d'insertions et de suppressions est relativement faible).

Les structures d'arbres

Les méthodes par tables d'index sont limites a certains type de fichiers (petits fichiers ou fichiers statiques). Les méthodes basées sur les structures d'arbres sont mieux adaptées aux fichier volumineux et/ou dynamiques. Afin de mieux occuper l'espace des blocs, on utilise des arbres m-aires.

Les arbres de recherche m-aires

Un arbre de recherche m-aires d'ordre n est un arbre ou chaque nœud peut avoir au maximum n fils : (Fils[1..n]) et n-1 valeurs (Val[1..n-1]).

Les valeurs à l‘ intérieur d'un nœud sont ordonnées en ordre croissant.

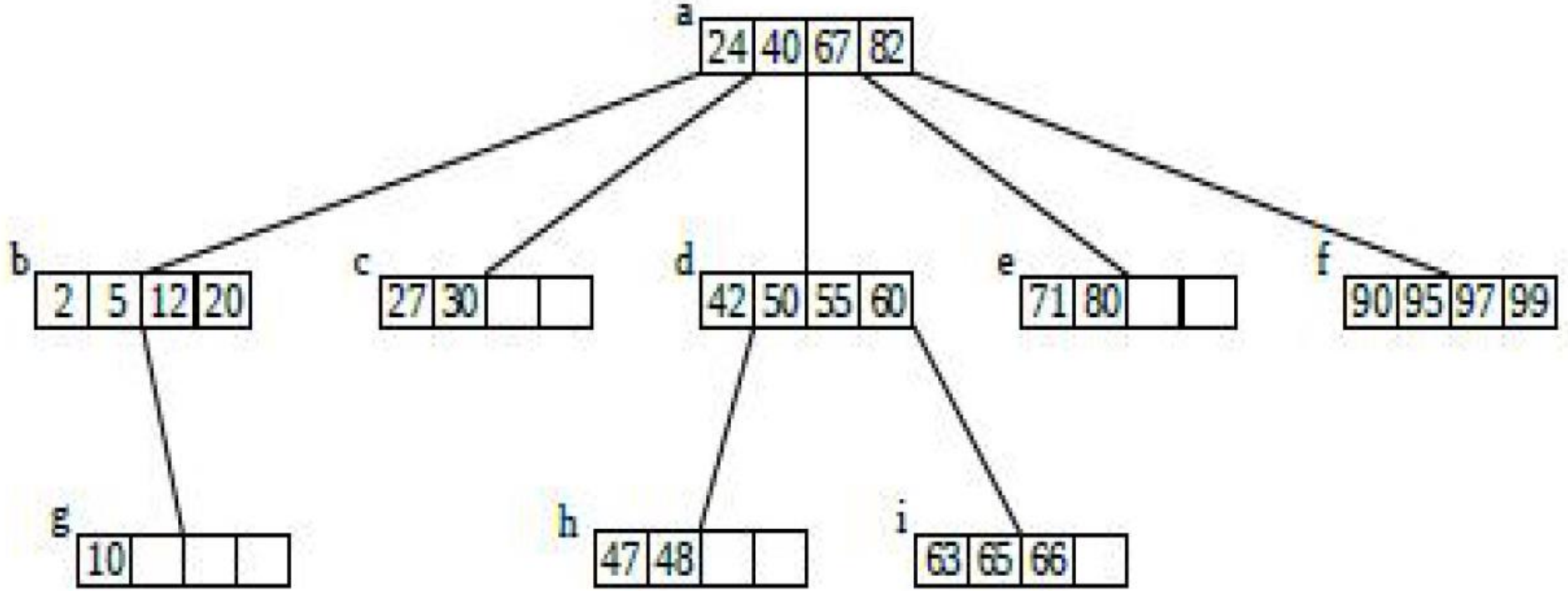
Le degré d'un nœud (le nombre de fils) est le nombre de valeurs stockées + 1.

Les fils sont organisés en fonction des valeurs du noeud, selon les règles suivantes :

i) Le Fils[1] pointe un sous-arbre contenant des valeurs < Val[1]

ii) Le Fils[i] pointe un sous-arbre contenant des valeurs > Val[i-1] et < Val[i] , pour i=2..n-1

iii) Le Fils[n] pointe un sous-arbre contenant des valeurs > Val[n-1]



La figure montre un arbre de recherche m-aires d'ordre 5 de racine a. Le noeud b est le fils 1 de a et f est le fils 5 de a.

Un noeud interne peut avoir certains fils à null et d'autres non. Par exemple tous les fils de b à part le 3^{e} sont à null. De même, les fils 1, 3 et 4 du nœud d sont à null, alors que son fils 2 (pointe h) et son fils 5 (pointe i) sont différents de null.

La structure générale d'un bloc est alors comme suit :

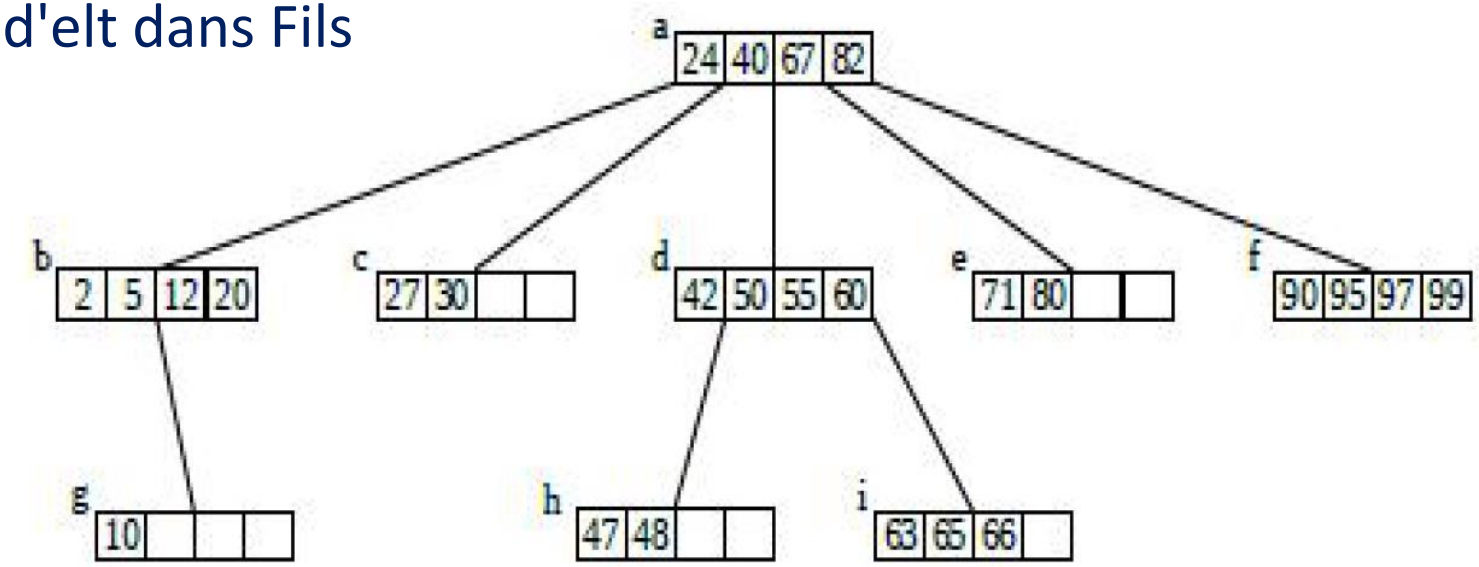
Type Tbloc = structure

Val : tableau[N-1] de typeqlq; // enregistrements ou (cles-adr)

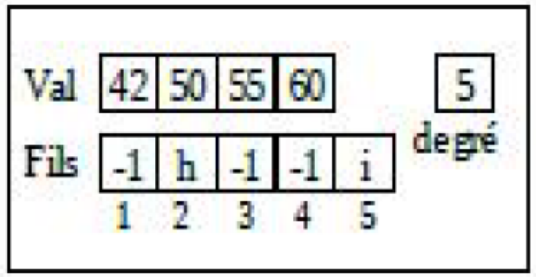
Fils : tableau[N] d'entier; // numéros de blocs

degre : entier; // nb d'elt dans Fils

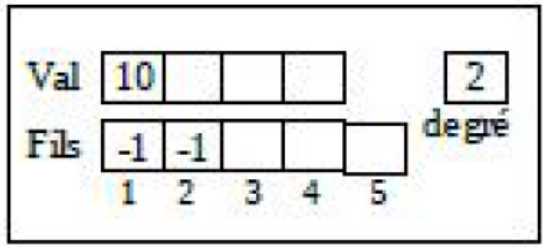
Fin



Bloc d



alors que le contenu du bloc g serait :



1. Recherche:

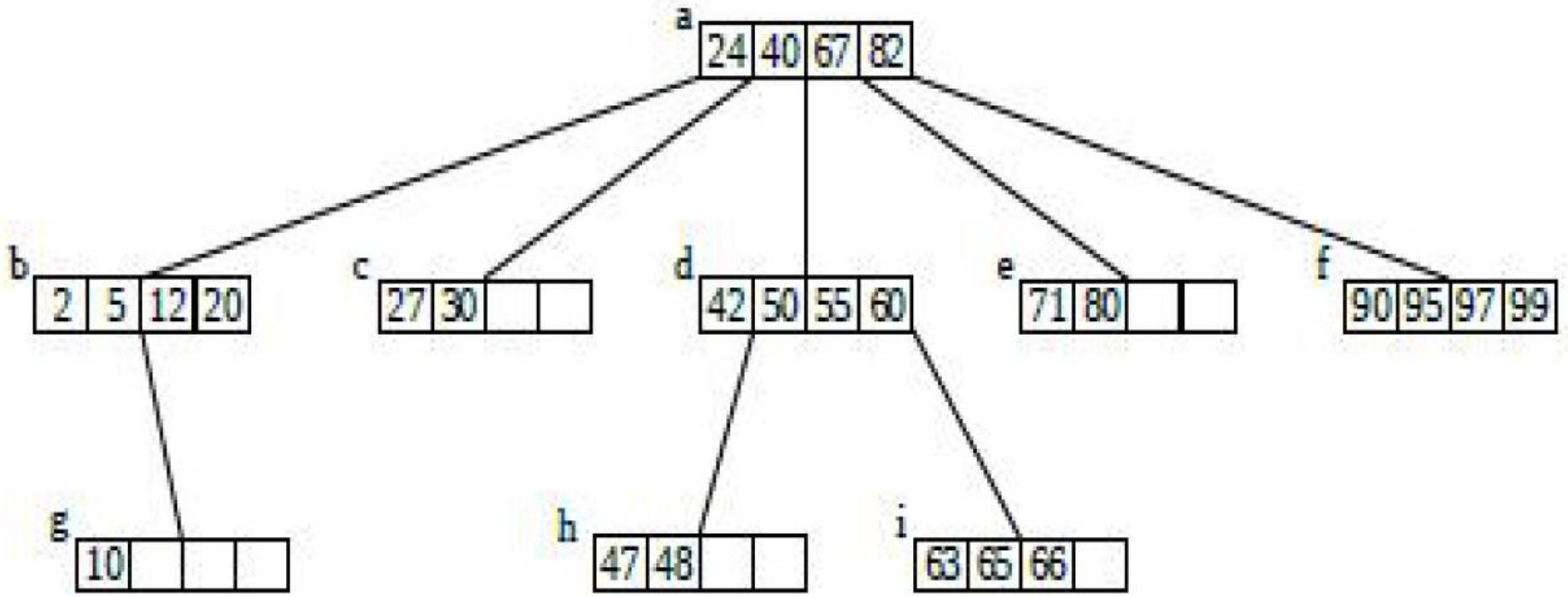
La recherche d'une valeur C commence dans le noeud racine P et se poursuit le long d'une branche :

1- Si C existe dans P alors stop avec succès

2- Si C n'existe pas dans P alors soit k la position dans P ou devrait être insérer C (pour que les valeurs restent ordonnées) : P = Fils[k]

Si P différent de Null alors aller a 1 Sinon stop avec échec.

```
Rech( c:Typeqlq; nomf:chaine; var trouv:booleen; var i,j,prec:entier)
Debut
Si ( nomf <> '' ) Ouvrir( F, nomf, 'A') Fsi;
i := Entete(F,1); // le num du bloc racine
prec := -1;
j := 1;
trouv := FAUX;
TQ ( Non trouv et i <> -1 )
LireDir( F, i, buf );
// recherche interne dans le bloc …
j := 1;
TQ ( Non trouv et j < buf.degre )
Si ( c = buf.Val[j].cle ) trouv := VRAI Sinon j := j+1 Fsi
FTQ; // fin de la recherche interne.
Si ( Non trouv )
prec := i; i := buf.Fils[j]
Fsi
FTQ;
Si ( nomf <> '' ) Fermer(F) Fsi;
Fin
```



Par exemple, la recherche de 48 dans l'arbre de la figure précédente se déroule comme suit :

On commence la recherche dans le nœud a. La valeur 48 n'existe pas et est comprise entre 40 et 67. Donc le prochain nœud à visiter est le Fils[3] (le nœud d).

La recherche se poursuit dans le nœud d, et le prochain nœud a explorer est le Fils[2], car 48 est comprise entre 42 et 50.

On continue la recherche dans le nœud h. La valeur 48 existe (Val[2]), on s'arrête donc avec succès.

Si on avait recherche la valeur 15, on aurait visité d'abord le nœud a, puis le nœud b (Fils[1] de a, car 15 < 24). Là on serait arrêté avec un échec, car 15 est comprise entre 12 et 20 et le Fils[4] de b est a null.

2. Insertion:

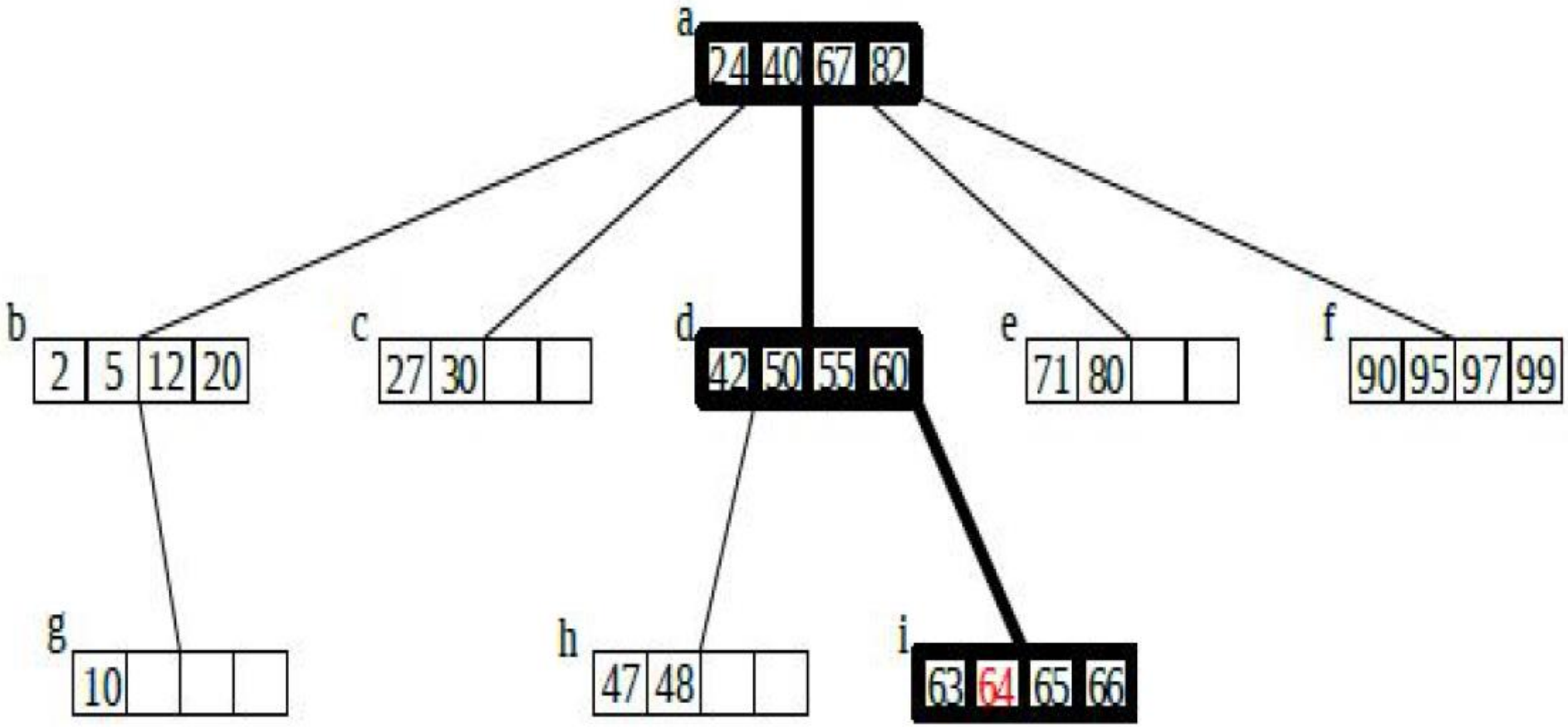
Pour insérer une nouvelle valeur V dans un arbre de recherche m-aires, on recherche d'abord la valeur. La recherche retourne aussi l'indice k où devrait être insérée V s'il y a avait de l'espace dans P.

Si P n'est pas plein, on insère V dans P, par décalages afin de garder le tableau de valeurs ordonne.

Sinon (P est plein) : on alloue un nouveau nœud Q contenant une seule valeur (Val[1] =V) et deux fils à null (Fils[1] = null et Fils[2] = null).

On fait pointer le nouveau nœud Q par le Fils[k] de P (qui était, avant l'insertion, forcément à null).

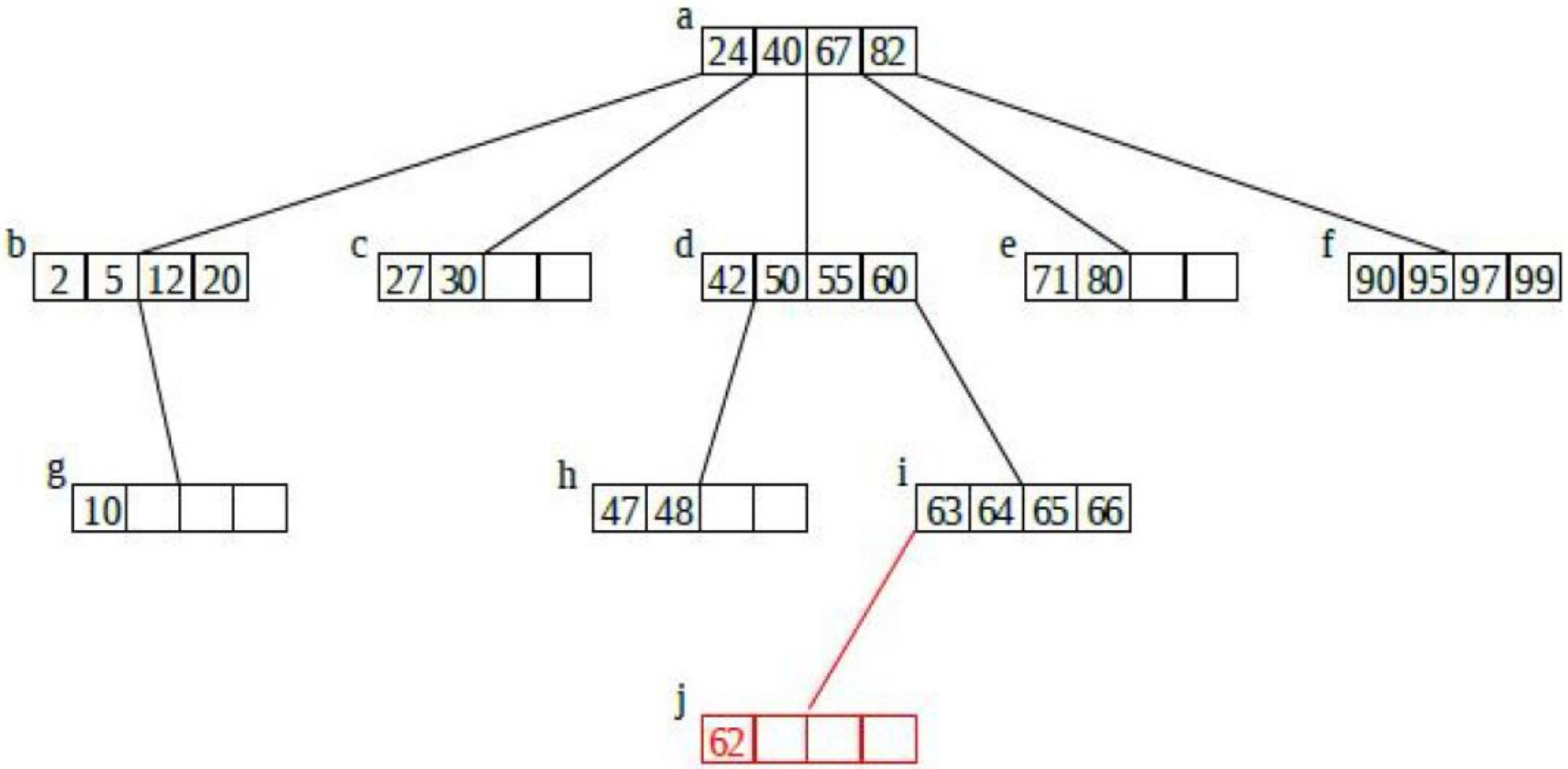
L'indice k est celui retourne par la recherche.



Par exemple, si on insère la valeur 64 dans l'arbre de la figure précédente, on procédera comme suit :

1- recherche de 64 → échec (le dernier nœud visite est i et la position ou devrait être insérée 64 est k=2)

2- comme le noeud i n'est pas plein, on peut alors insérer 64 a la position 2 en décalant à droite les valeurs > 64

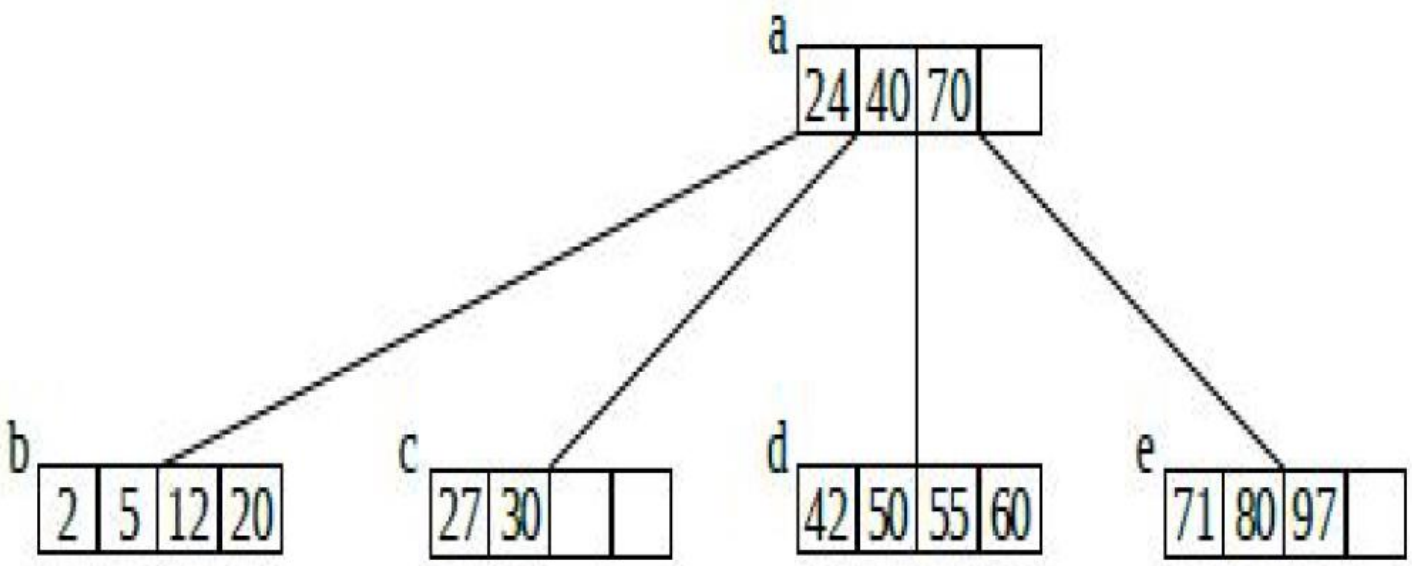


Si on insère encore la valeur 62, on allouera alors un nouveau nœud (j) qui va contenir 62 et il sera pointé par le Fils[1] du nœud i (i étant le dernier nœud visité dans la recherche de 62 et la position où devrait être insérer 62 dans i, s'il y avait de l'espace, est k = 1)

Les B-Arbres :

Ceux sont des arbres de recherche m-aires qui restent toujours équilibres et sont donc très utilisés pour gérer des fichiers volumineux et dynamiques.

Pour simplifier la présentation, on choisira un ordre impair (N = 2d+1) ; Dans un B-Arbre d'ordre N : tous les nœuds à part la racine, sont remplis au minimum à 50% (soit d valeurs) la racine peut contenir au minimum 1 valeur toutes les branches ont la même longueur (arbre complètement équilibré).



La recherche dans un B-Arbre est similaire a la recherche dans un arbre de recherche m-aires. La différence se situe dans l'algorithme d'insertion et l'algorithme de suppression.

Algorithme d'insertion

Pour insérer une valeur V dans un B-Arbre de racine R, on procède comme suit :

On insère V et son fils droit fd (initialement fd = -1), c-a-d si V sera insérée à une position j dans un nœud, alors il faudra insérer aussi ≪ son fils droit ≫ fd a la position j+1 (dans le tableau des fils)

1- rechercher V, pour vérifier qu'elle n'existe pas et trouver le dernier nœud feuille visite (P)

2- si P n'est pas plein, on insère (V,fd) dans P par décalages internes et on s'arrête.

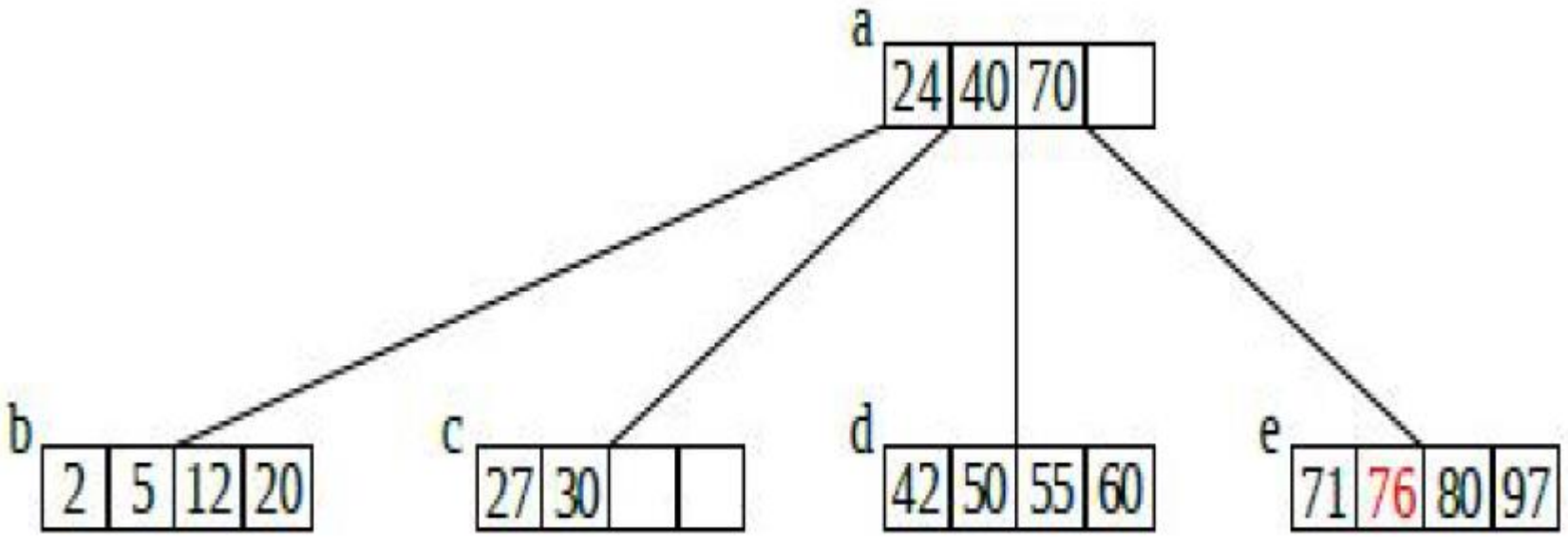
3- si P est déjà plein, alors il va ≪ éclater ≫ en deux (P et Q, un nouveau nœud alloue) :

a) former la ≪ séquence ordonnée ≫ des valeurs de P + la valeur V

b) affecter les d premières valeurs avec les d+1 premiers fils au noeud P

c) affecter les d dernières valeurs avec les d+1 derniers fils au nouveau nœud Q

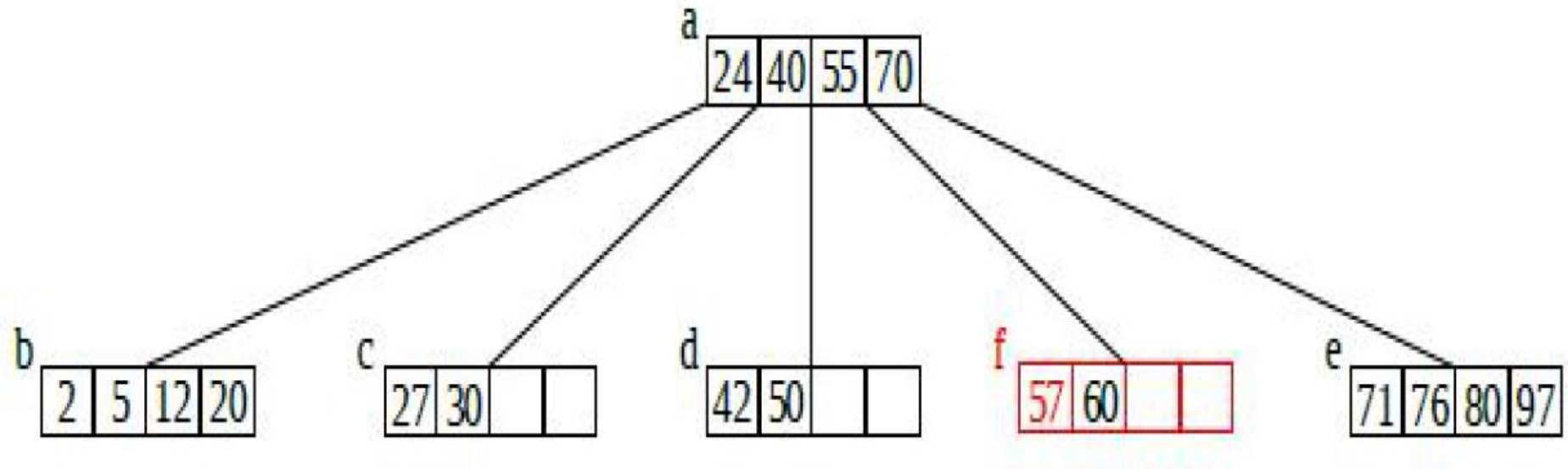
d) insérer la valeur du milieu (celle qui se trouve a la position d+1 dans la séquence ordonnée) dans le nœud parent de P. Le fils droit de cette valeur sera le nœud Q – Aller a 2



Par exemple, l'insertion de 76 dans le B-Arbre de la figure précédente, se déroule comme suit:

1- recherche de 76 → trouv =FAUX, le dernier visite = e (c'est une feuille) et la position ou doit être insérée 76 est 2.

2- comme le nœud e n'est pas plein, on insère alors 76 avec son ≪ fils droit ≫ -1 respectivement aux positions 2 dans le tableau Val et 3 dans le tableau Fils du nœud e.



Si on insère maintenant la valeur 57, c'est le nœud d qui sera visite en dernier par la recherche.

La position de 57 dans d devrait alors être 4 (et son fd a la position 5); Comme le nœud d est déjà plein, il y aura alors un ≪ éclatement ≫ (cela veut dire, allocation d'un nouveau nœud, f par exemple)

a) formation de la ≪ séquence ordonnée ≫:

Val : 42, 50, 55, 57, 60

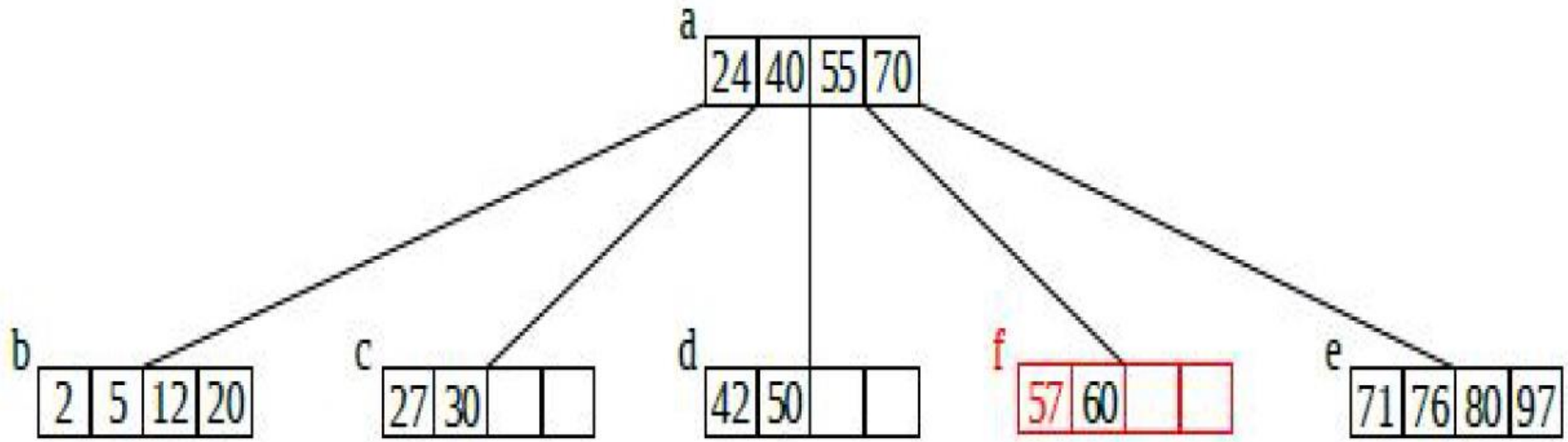
Fils: -1, -1, -1, -1, -1, -1

b) les d premières valeurs (42, 50) et les d+1 premiers fils (-1, -1, -1) seront affectés au nœud d

les d dernières valeurs (57, 60) avec les d+1 derniers fils (-1, -1, -1) seront affectés au nœud f

La valeur du milieu (55) avec comme fils-droit (f) seront insérés dans le nœud parent (a)

comme le nœud a n'est pas plein, l'insertion de (55, f) peut se faire par décalages internes



Si on continue l'insertion de 7, le nœud b va ≪ éclater ≫ (donc allocation d'un nouveau nœud g), la séquence ordonnée des valeurs de b + la valeur 7 est formée:

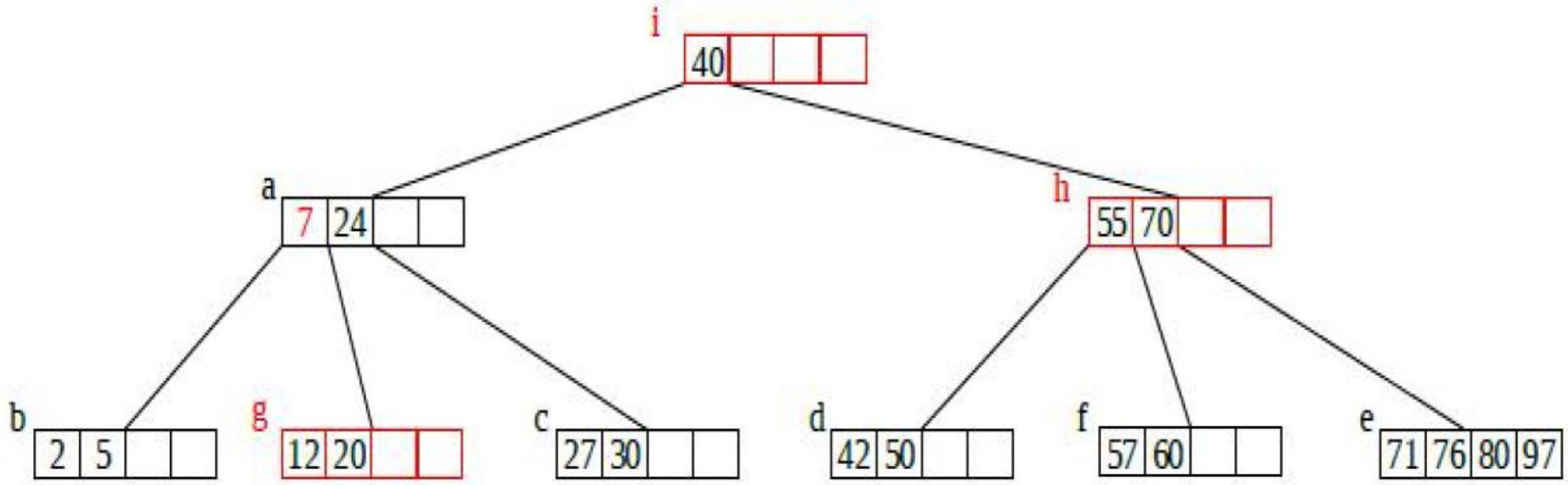
- Val : 2 , 5 , 7 , 12 , 20
- Fils: -1, -1, -1, -1 , -1 , -1

Les valeurs (2 et 7) avec les 3 premiers fils restent dans b. Les valeurs (12 et 20) avec les 3 derniers fils seront affectes au nouveau nœud g. La valeur du milieu (7) avec comme fils-droit le nœud g seront inseres dans le pere (a).

Comme le nœud a est lui aussi plein, il va alors ≪ éclater ≫ a son tour. Un nouveau nœud h est alors alloue et la séquence ordonnée est formée :

- Val : 7 , 24 , 40, 55, 70
- Fils: b , g , c , d , f , e

— Les d premières valeurs (7 et 24) avec les 3 premiers fils (b, g et c) restent dans a. Les d dernières valeurs (55 et 70) avec les 3 derniers fils (d, f et e) seront affectes au nouveau nœud h. La valeur du milieu (40) avec comme fils-droit le nœud h seront insérés dans le père du nœud a.
— Comme le nœud a était la racine de l'arbre (il n'a donc pas de père), une nouvelle racine (i) est alors allouée contenant uniquement la valeur 40 avec comme fils-gauche (Fils[1]), l'ancienne racine (a) et comme fils-droit (Fils[2]) le nœud h.
— Quand la racine d'un B-Arbre éclate (a cause d'une insertion), la profondeur de l'arbre augmente d'un niveau.



Conclusion : Méthodes d'arbres

Avantages :

- **Évolutivité** : La méthode permet de gérer des fichiers de grande taille sans contraintes majeures.
- **Organisation** : Les fichiers sont ordonnés, facilitant ainsi les recherches et les accès.

Inconvénients :

- **Adaptabilité limitée** : Ces méthodes sont mieux adaptées aux fichiers statiques, car les modifications fréquentes peuvent compliquer leur gestion.
- **Déséquilibre de l'arbre** : Les insertions ou suppressions successives peuvent déséquilibrer la structure de l'arbre, entraînant une dégradation des performances. Cependant, une réorganisation périodique de l'arbre peut être envisagée pour rétablir un équilibre optimal et garantir des performances efficaces.

HACHAGE

- La recherche séquentielle peut être faite en temps d'accès d'ordre ***O(N),*** ce qui veut dire que le nombre de "seeks" augmente au même taux que la taille du fichier.
- Les arbres B diminuent ce taux puisqu'ils permettent un temps d'accès d'ordre $O(Log_k N)$ ou k est une mesure de la taille d'une feuille (c'e.a.d., le nombre d'enregistrements qui peuvent etre sauvegarder dans une feuille).
- Ce que l'on voudrait, cependant, est un temps d'acces d'ordre O(1) ce qui veut dire que quelque soit la taille du fichier le temps d'accès a un enregistrement est toujours un petit nombre de "seeks".
- ***Hashcoding Statique***: Ces techniques peuvent atteindre une telle performance si la taille du fichier n'augmente pas.

Fonction de hachage

- **Définition** : Une fonction mathématique qui transforme une clé (ex. numéro de compte) en une adresse (ex. emplacement mémoire ou position dans le fichier).
- **Exemple** :

  Si la clé est un entier K, une fonction de hachage simple peut être :

  H(K)=K mod M où M est la taille du fichier ou du tableau.

Qu’Est-ce que le Hashcoding?

- Une ***fonction Hash*** est une fonction h(K) qui peut transformer une cle K en une adresse.
- Hashcoding est relie à la construction d’indexes qui correspond à associer une clé à une adresse relative d’enregistrement.
- Hashcoding, cependant, est ***diffèrent*** de la construction d’indexes en deux respects:
  - Avec le hashcoding, il n’y a pas de connexion évidente entre la clé et l’adresse.
  - Avec le hashcoding, deux clés différentes peuvent être transformées en la même adresse.

Collisions

- Lorsque deux clés différentes produisent la même adresse, il y a une ***collision***. Les clés impliquées dans cette collision s'appellent des ***synonymes***.
- Néanmoins, il est très difficile de trouver une fonction hash qui évite les collisions. Il est plus simple de trouver des ***moyens de résoudre*** ces collisions.
- ***Solutions Possibles***:
  - Espacer les enregistrements
  - Utilisation de mémoire supplémentaire
  - Placement de plus d'un enregistrement à une adresse.

Un Algorithme de Hashcoding Simple

- ***Etape 1:*** Représenter la clé en forme numérique.
- ***Etape 2:*** Plier et Ajouter (Fold and Add)
- ***Etape 3:*** Division par un nombre premier et utilisation du reste comme d'une adresse.

Fonctions Hashcoding et Distribution d'Enregistrements

- Les enregistrements peuvent être distribués sur toutes les adresses de manières différentes: Il peut y avoir (a) aucun synonymes (distribution uniforme); (b) que des synonymes (pire cas); (c) quelques synonymes (cela arrive avec des distributions au hasard).
- Les distributions uniformes pures sont difficiles à obtenir et ne valent pas la peine d'être cherchées.
- Les distributions au hasard peuvent être facilement dérivées mais elles ne sont pas parfaites car elles peuvent générer un nombre important de synonymes.
- On veut des méthodes de hashcoding meilleures que cela.

D’autres Methodes de Hashcoding

- Bien qu’il n’existe pas de fonction hashcoding qui garantisse des distributions meilleures que les distributions au hasard dans tous les cas, si on considère le clés qui vont être “hachées”, certaines améliorations sont possibles.
- Voici quelques méthodes qui sont potentiellement meilleures que les méthodes au hasard:
  - Examiner les clés pour trouver un patron
  - “Plier” certaines parties de la clé
  - Diviser la clé par un nombre
  - Elever la clé au carre et prendre le milieu
  - Faire une transformation radix.

Autres Techniques de Resolution de Collision

Il y a quelques variations sur le hashing au hasard qui peut améliorer la performance:

- **Double Hashing**: Utiliser une seconde fonction de hachage pour calculer les déplacements
- **Chaînage** : associer à chaque position une liste chaînée pour stocker tous les enregistrements ayant la même adresse.
- L'**essai linéaire** : en cas de collision (si la position calculée par la fonction de hachage est déjà occupée), on cherche la prochaine position libre dans la table en avançant de manière séquentielle (linéaire).