Chapitre 1 : Les Fichiers

CARACTERISTIQUES

- fichier : collection de données du même type
- données persistantes -> stockées sur mémoire de masse
- capacité de stockage beaucoup plus grande que celle des structures de données residant en mémoire centrale

Fichiers : 2 niveaux

- Niveau système d'exploitation:
  - allocation d'espace disque, organisation des fichiers sur disques (blocs)
  - transfert des blocs d'octets du disque vers la mémoire centrale (et inversément)
  - gestion des répertoires de fichiers (directory)
- Niveau utilisateur (par ex. l'utilisation d'un fichier depuis un programme Modula):
  - stocker des enregistrements sur disque.
    Ex. : ("L'année du déluge", "Eduardo Mendoza",
    "Seuil", 1993)
- rechercher des enregistrements sur le disque
  -> méthodes d'accès aux données

NIVEAU SYSTEME D'EXPLOITATION

Accès individuel et par bloc

RAM:

chaque octet possède une adresse;

accès octet par octet

temps d'accès : 60 à 70 nanosecondes

Disque:

les octets sont regroupés par blocs (p.ex. 512 ou 1024 octets / bloc)

temps d'accès (positionnement de la tête de lecture): 8 à 9 millisecondes

modifier un octet d'un bloc =>

1. lire un bloc du disque en mémoire RAM;
2. modifier l'octet en RAM;
3. récrire le bloc sur disque

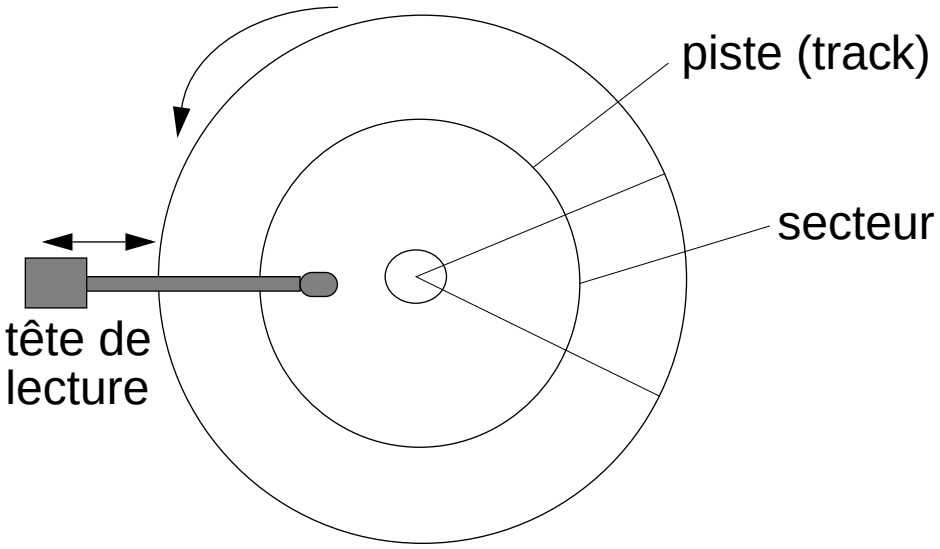
Efficacité => regrouper toutes les opérations portant sur les octets d'un bloc pendant qu'il est en mémoire RAM.

- Grouper par dans le même bloc les données qui sont en général manipulées ensemble.

“Hardware” du disque magnétique

Les bits sont regroupés en octets (bytes, caractères)

Pistes concentriques



Chaque secteur est divisé en blocs

taille d’un bloc : 512, 1024, 2048 ou 4096 octets

Lecture d’un bloc du disque

a) il faut connaître l’adresse du bloc sur le disque

b) positionner la tête de lecture sur la bonne piste

c) attendre que le bon bloc se présente sous la tête de lecture

d) transférer le bloc en mémoire centrale

- temps nécessaire pour b) + c) + d) : 8 à 9 millisec
- temps pour d) : environ 0.1 millisec

  -> il est avantageux de lire plusieurs blocs contigüs à la suite

- “Buffering” de blocs : réservation de plusieurs tampons (un tampon peut recevoir un bloc) dans la mémoire centrale. Lorsqu’un tampon est utilisé pour le transfert depuis le disque, le CPU peut en même temps traiter les données d’un autre tampon;
  -> Le contrôleur de disque et le CPU travaillent en parallèle.
-

Création et écriture de fichiers sur disque

Méthode pour nommer des objets stockés sur disque (équiv. variables dans les programmes)

Fichier

collection d'octets identifié par un nom (le nom du fichier)

Opérations de création et d'écriture

écrire: fichier $\times$ liste-octets $\times$ position -> fichier

créer: …

supprimer: …

vider: …

longueur: fichier -> entier

Réalisation. Un répertoire (directory, folder, dossier, catalogue, etc.) + un ensemble de blocs attribués à chaque fichier.

Organisation.

- blocs contigus
- blocs chaînés
- index des blocs

Organisation - blocs contigus

rép.		A		C	B	

Problème de création d'un fichier

Comment gérer plusieurs fichiers qui grandissent simultanément?

Nécessité de réserver un nombre fixe de blocs à la création.

Où allouer le premier bloc? Combien de blocs réserver?

Problème de fragmentation - trous - => impossible de trouver un espace libre suffisamment grand.

Exemples

Apple II

DEC PDP11 - RT11

IBM 3090 - VM (minidisques)

Organisation - chaîne de blocs

Le répertoire contient le no. du premier bloc du fichier

Chaque bloc contient le no. du suivant

Les blocs inutilisés sont liés entre eux de la même manière

Avantages

Pas de réservation de blocs à la création

Utilisation optimale du disque

Extensibilité des fichiers

Problèmes

Trouver le n-ième bloc du fichier => lire n blocs

Mélange de données (utilisateur) et adresses disque (système) dans le même bloc

Mauvaise résistance aux pannes

Organisation - index de blocs - FAT

Table d'allocation des blocs

Le répertoire contient le no. du premier bloc du fichier

On a une table d'allocation (FAT) qui contient un élément par bloc du disque.

FAT[i]

= <libre> si le bloc n'est utilisé par aucun fichier

= j si le bloc no. i est utilisé par un fichier f et j suit le bloc no. i dans f

= <fin> si le bloc no. i est utilisé par un fichier f et c'est le dernier bloc de f.

Pour que ce soit efficace il faut que la FAT soit copiée en mémoire centrale.

Problème

Pour le gros disques (1 gigaoctet) la FAT devient très grande

Exemple

DOS

Organisation - index de blocs hiérarchisés

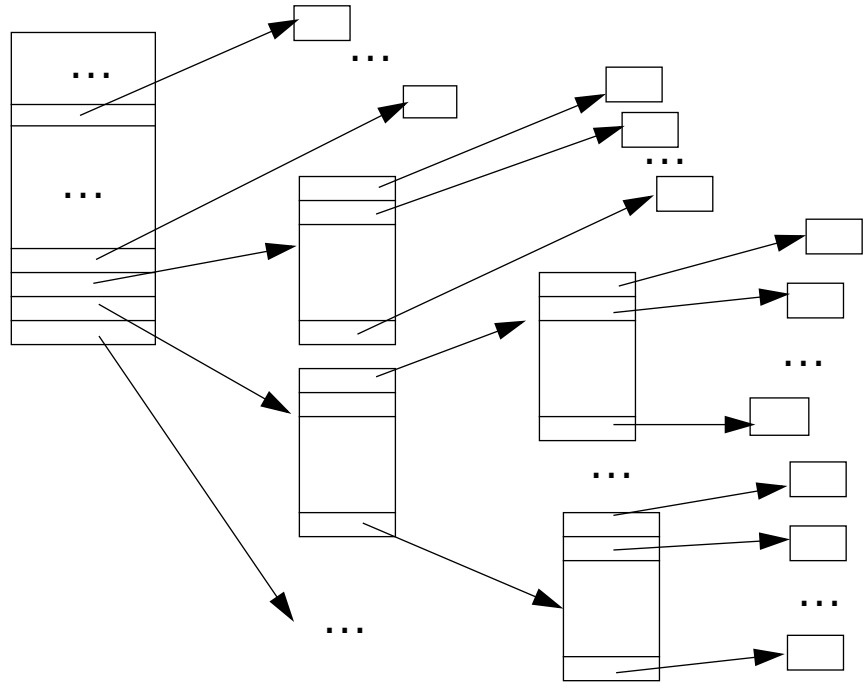
Un *descripteur de fichier* (i-node) contient

- les no.s des 10 premiers blocs
- le no. d'un bloc d'index primaire,secondaire et tertiaire

Si le fichier fait plus de 10 blocs on alloue un bloc d'index primaire qui contient les nos des N blocs suivants.

Si le fichier fait plus de N + 10 blocs on alloue un bloc secondaire qui contient les nos des N blocs d'index primaire.

Si le fichier fait plus de N^2 + N + 10 blocs on alloue un bloc tertiaire …



Exemple : UNIX

NIVEAU UTILISATEUR

Un fichier est une collection d'enregistrements

Un enreg. est composé d'un ou plusieurs champs

Un champ est une chaîne d'octets de long. fixe ou variable

Les champs sont généralement typés (entier, ch. de car...)

Un ou plusieurs parmi les champs jouent le rôle de *clé*

Opérations de base

- ouvrir, fermer, se déplacer dans un fichier
- lire, écrire, supprimer un enregistrement

Organisations

1)Séquentielle, 2)relative, 3)indexée (index B - arbre)

Types d'accès

- séquentiel

  lire: fichier -> enregistrement

  (notion d'enregistrement courant)
- direct (relatif)

  lire: fichier $\times$ no_enregistrement -> enregistrement
- indexé (par la clé)

  lire: fichier $\times$ valeur_clé -> enregistrement

Fichiers de textes / fichiers binaires

On distingue deux types de fichiers:

fichiers de textes

- les données sont stockées sous forme de code ascii. Ex : 35 -> | 63C | 65C | (63c est le code ascii de '3', 65C est le code ascii de '5')
- on peut lire et écrire les fichiers de textes avec un éditeur de textes
- il y a la notion de fin de ligne: elle est indiquée par la présence du caractère de contrôle EOL (36C)

fichiers binaires

- les données sont stockées dans un format interne. Ex: 35 -> | 43C | ($35_{10} = 43_8$)
- lecture et écriture des fichiers binaires par programme uniquement

1) Fichiers en organisation séquentielle

Les enregistrements forment une séquence, c-à-d qu'ils sont arrangés de manière adjacente les uns à la suite des autres.

Sur la plupart des ordinateurs l'entrée sur clavier et la sortie sur l'écran sont assimilés à des fichiers séquentiels.

Support physique: unité de bande magnétique, disques magnétiques, disques optiques.

Remarque: l'organisation séquentielle convient pour les deux types de fichiers (textes et binaires)

Opérations sur les fichiers séquentiels

Opérations d'écriture

- (1) générer un fichier vide
- (2) ajouter un enregistrement à la fin du fichier

Opérations de lecture

- (3) ouvrir et positionner la "tête de lecture" au début du fichier
- (4) lire un enregistrement et passer au suivant

-> l'accès à un enregistrement particulier demande l'ouverture du fichier puis la lecture successive de tous les enregistrements qui précédent l'enregistrement recherché.

-> pour accéder à l'enregistrement précédent, il faut réouvrir le fichier (positionnement au début du fichier)

Etat de lecture / état d'écriture

le fichier ne peut se trouver que dans un seul état à la fois

pour passer à l'état d'écriture: opération (1)

pour passer à l'état de lecture: opération (3)

Problème : comment modifier un enregistrement d'un fichier séquentiel ?

Les fichiers séquentiels en Modula

- Fichiers de textes:

Le module “InOut”

Lecture et écriture de chaînes de caractères et de nombres. Ex: ReadString, ReadCard, WriteString,...

La lecture se fait depuis le clavier ou depuis un fichier de texte.

L’écriture se fait sur l’écran ou dans un fichier de texte.

Fin du flot d’entrée ou lecture de la fin du fichier:
Done = FALSE ou in.eof = TRUE.

Le module “Terminal”

Lecture et écriture de charactères et de chaînes de caractères au terminal. Ex: KeyPressed, Read,Write,...

Remarque: la procédure Read ne fait pas l’écho du caractère lu (il faut utiliser Write pour faire l’écho)

- Fichiers binaires:

Le module *LogiFile* (voir fichiers en organisation relative)

2) Les fichiers en organisation relative (ou fichiers à accès direct)

Les enregistrements sont identifiés par un numéro d'ordre (relatif par rapport au début du fichier).

L'enregistrement de numéro d'ordre **n** occupe la **n**ème position dans le fichier.

Deux type d'accès possible:

- séquentiel (idem fichiers séquentiels)
- direct, à condition de connaître le numéro d'ordre de l'enregistrement.

Avantages accès direct:

- rapidité de recherche
- lecture et écriture n'importe où dans le fichier

Support physique: disques magnétiques, disques optiques.

Remarque: les fichiers relatifs sont de type texte ou binaire (mais utilisation limitée de l'accès direct pour les fichiers de textes)

Opérations sur les fichiers relatifs

Opérations de positionnement

- ouvrir un fichier
- positionner “la tête de lecture / écriture” sur l’enregistrement de numéro d’ordre **n**
- positionner “la tête de lecture / écriture” au début du fichier

Opérations d’écriture

- créer un fichier
- écrire un enregistrement

Opération de lecture

- lire un enregistrement

Les fichiers relatifs en Modula

Le module “LogiFile” (exemples d’opérations):

Les enregistrements sont adressés par un déplacement relatif en nombre d’octets à partir du début du fichier
(déplacement = ‘highpos’∗10000H + ‘lowpos’)

```
PROCEDURE GetPos( f : File;
                  VAR  highpos  : CARDINAL;
                  VAR  lowpos   : CARDINAL; )
(* donne la position courante du fichier *)

PROCEDURE SetPos( f          : File;
                  highpos    : CARDINAL;
                  lowpos     : CARDINAL; )
(*positionne la tête de lecture / d’écriture dans le fichier*)

PROCEDURE ReadNBytes( f              : File
                      buffPtr        : ADDRESS;
                      requestedBytes: CARDINAL;
                  VAR read           : CARDINAL; )
(*lit dans le fichier ‘f’ n octets (n=’requestedBytes’) et les
copie dans le buffer à l’adresse ‘buffptr’*)

PROCEDURE WriteNBytes( f             : File
                       buffPtr       : ADDRESS;
                       requestedBytes: CARDINAL;
                   VAR written       : CARDINAL; )
(* idem mais pour écrire dans le fichier ‘f’ *)
```

3) Les fichiers en organisation séquentielle indexée

Origine: langage COBOL (ISAM : Index Sequential Access Method)

Implique deux structures de données: données + index.

Idée générale: index de fichier ≡ index à la fin d'un livre.

Clé

Le champ (ou les champs) par lequel les enregistrements sont référencés s'appelle *clé d'accès.*
-> on peut accéder à un enregistrement en précisant la valeur de la clé.

Un fichier indexé doit avoir au moins une clé (la clé primaire).

Il peut avoir d'autre clés (secondaire, tertiaire,...).

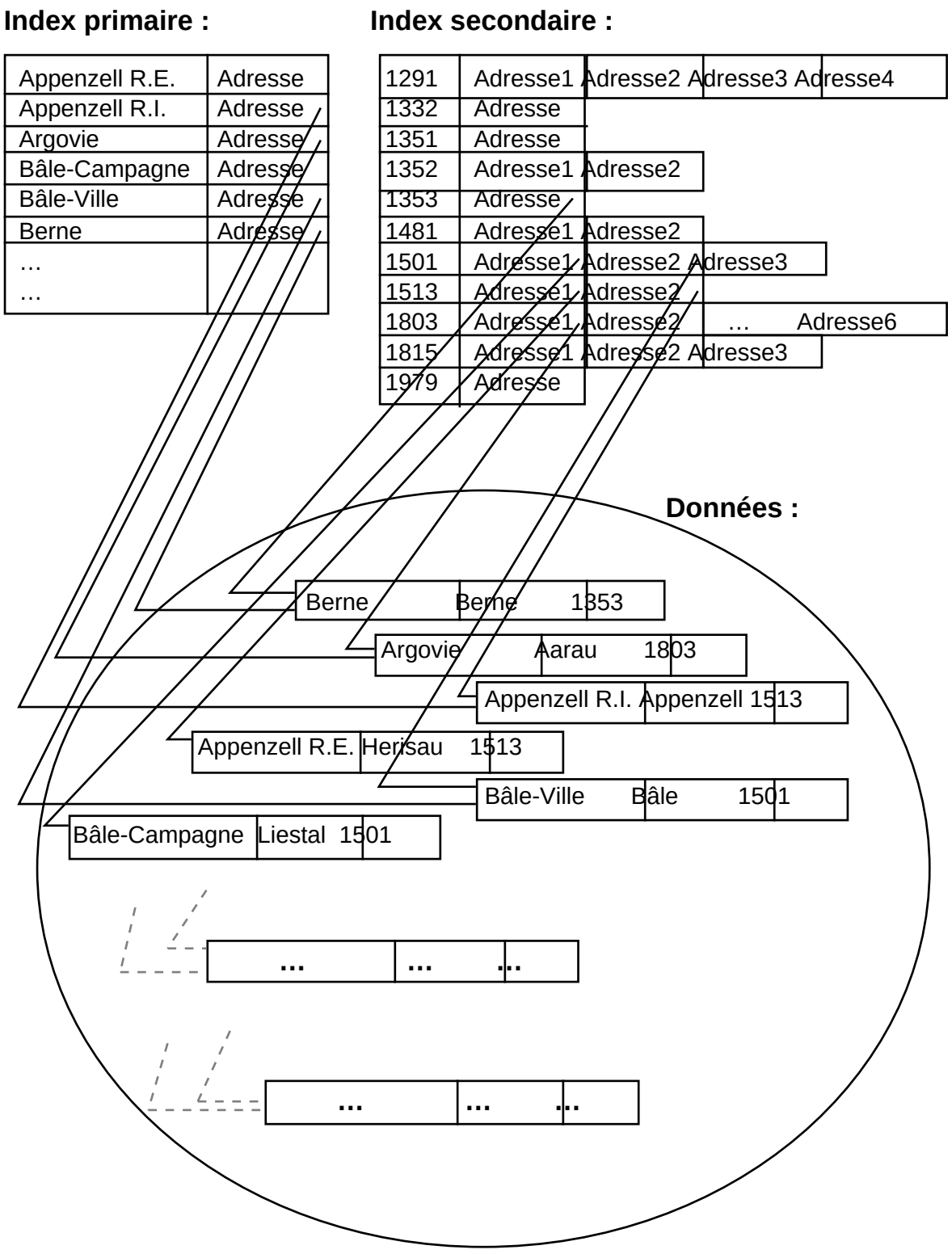
La clé d'accès peut être unique ou non.
Ex. clé unique:
Ex. clé non unique:

La clé primaire est unique (mais il y a des exceptions...).

Index

L'*index* est une suite de paires <K,A> où A est l'adresse de l'enregistrement qui a K pour valeur de clé.
Remarque: L'index est généralement trié selon la clé.

Vue schématique d'un fichier séquentiel indexé



Les fichiers séquentiels indexés (suite)

Types d'accès possibles:

- séquentiel (dans l'ordre de la clé primaire)
- indexé
- (direct)

Support physique: disques magnétiques, disques optiques.

Remarque: les fichiers relatifs sont de type binaire

Implémentation

Deux parties distinctes:

- les enregistrements complets stockés dans l'ordre croissant de leur clé primaire
- un index* pour chaque clé.

Certaines implémentations stockent les enregistrements et les index dans deux fichiers différents (c'est le cas du module "Isam" que nous verrons par la suite)

* Pour permettre une consultation rapide des index, ceux-ci sont généralement triés selon la valeur de la clé et structuré en arbre (binaire, B-arbre, B^+-arbre). Cette arborescence est construite automatiquement par le système de fichiers au fur et à mesure que l'utilisateur introduit les enregistrements.

Opérations sur les fichiers séquentiels indexés

Création d'un fichier

Définition des enregistrements.

Définition des clés.

Lecture

Accès à un enregistrement en fonction d'une clé et d'une valeur de clé.

Parcours séquentiel du fichier selon la clé primaire, éventuellement selon les autres clés.

Ecriture

Ajout d'un enregistrement.

Modification d'un enregistrement désigné par une clé et une valeur de clé.

Les fichiers séquentiels indexés en Modula

Le Module “Isam”:

Accès indexé.

Accès séquentiel selon n’importe quelle clé.

IsamResultType = (isamDone, truncRead, nextRead,
notFound, keyFileFull, notUnique, endOfFile,...)

Index:

- L’utilisateur peut définir plusieurs clés (max. 30).
- Clé n° 0: clé unique, associée automatiquement à chaque enregistrement. Peut être utilisée pour des références.
- Structuration des index: un B^+-arbre par clé.

Attention: les valeurs des clés sont stockées sans notion de typage -> surprises désagréables avec chaînes de caractères et nombres.

Exemple d’utilisation du module “Isam”

Enoncé du problème:

Créer un ficher contenant une description élémentaire des cantons suisses. Chaque canton ou demi-canton est décrit par:

- Son nom de canton
- Sa date d'entrée dans la Confédération suisse
- Le nom de son chef-lieu

La structure de ce fichier devra être adéquate à une consultation rapide d’un canton selon le nom du canton ou la date d’entrée dans la confédération.

-> Voir listing du module “creerSuisse”

Fichiers: avantages / inconvénients

Séquentiels

+ Peu couteux si implanté sur bande magnétique.
  Bien adapté au traitement par lot (flot de données).
  Favorable pour traiter de grandes quantités d'enregistrements à la fois.

- Consultation lente.
  Modification des enregistrements fastidieuse.

Relatifs (accès direct)

+ Accès immédiat à un enregistrement (lecture et écriture).

- Table numéro->enregistrement: gérée par l'utilisateur

Séquentiels indexés

+ Accès par clé directement sur le bon enregistrement.
  Mise à jour des enregistrements ponctuelle.
  Index géré par le système.

- Les index peuvent prendre beaucoup de place.
  Accès séquentiel lent pour grand fichier (car parcours des index)

Les index pour les fichiers

But: accélérer l'accès aux enregistrements de données (idée similaire aux index de livre)

Contenu: L'*index* est une suite de paires <K,A> où A est l'adresse de l'enregistrement dans le fichier et K la valeur de clé de l'enregistrement.

Stockage: dans un fichier séparé (généralement)

Type d'index

- dense: contient une entrée pour chaque enregistrement de données
- non dense: contient une entrée pour chaque bloc d'enregistrements de données

Structuration des index

- tables ordonnées
- hachage (hash coding)
- bitmap, encodage par supperposition
- arbres de recherche
- B-arbres

Remarque: nous allons nous intéresser uniquement aux arbres de recherche et aux B-arbres.

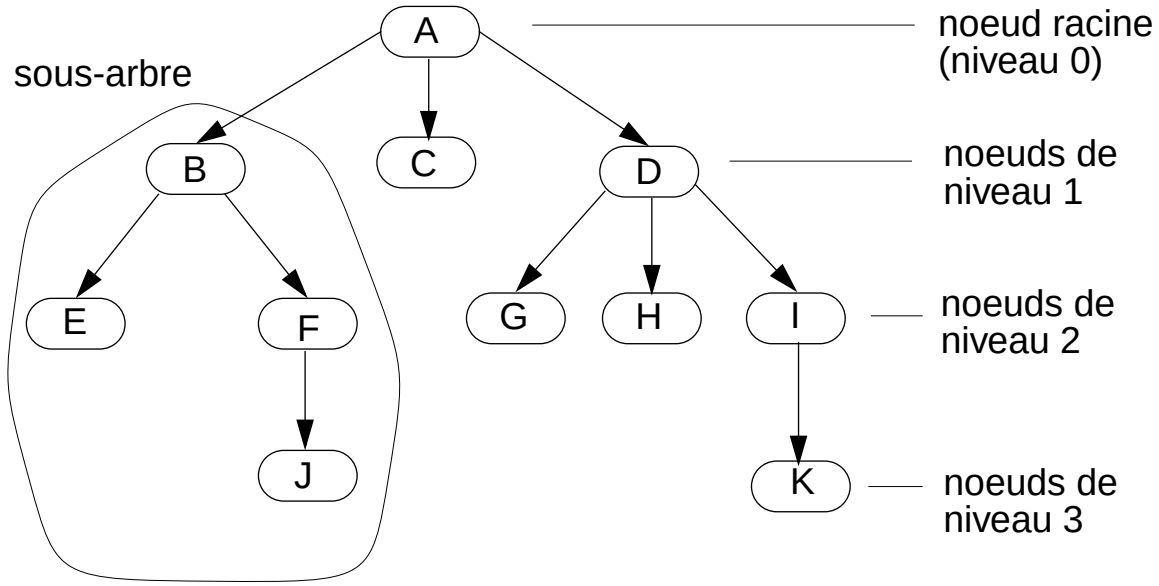
La structure d'arbre

Structure utilisée pour stocker des données en reflètant les liens hiérarchiques qui existent entre-elles.

Définition. Un arbre est:

- soit la structure vide
- soit un noeud auquel est associé un nombre fini de sous-arbres disjoints (descendants)

Terminologie:



(le noeud A est le noeud racine, les noeuds B,F,D et I sont les noeuds internes et les noeuds E,J,C,G,H et K sont les noeuds feuille de l'arbre)

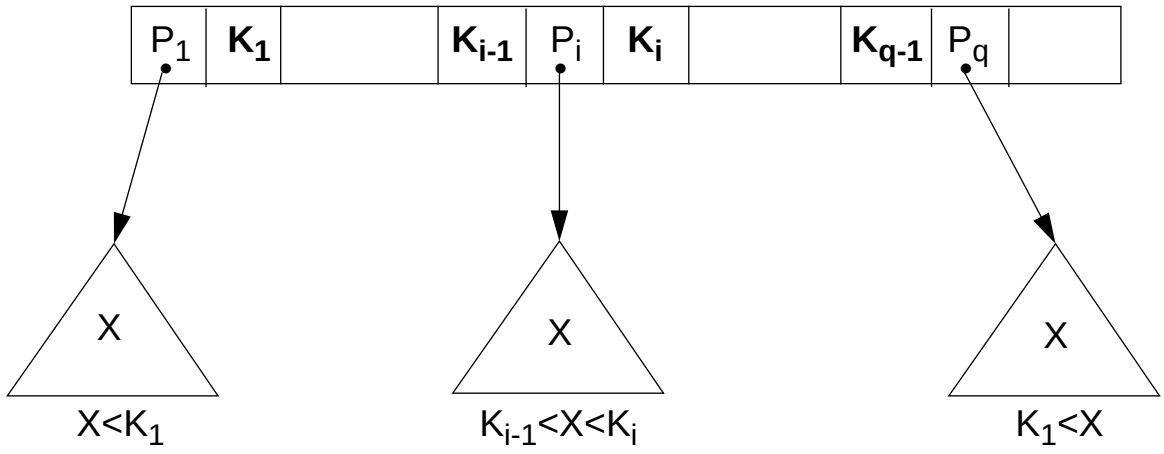
- Chaque noeud a exactement un noeud *parent* (excepté la racine) et 0 à n noeuds *enfant*.
- *Sous-arbre* d'un noeud: noeud lui-même + tous ses noeuds *descendants*.

Arbres de recherche

Arbre spécial pour guider la recherche d'un enregistrement en fonction d'une clé et d'une valeur de clé.

Arbre de recherche d'ordre p:

- Chaque noeud contient q-1 valeurs de clé et q pointeurs dans l'ordre $<P_1,K_1,P_2,K_2,...,P_{q-1},K_{q-1},P_q>$ avec $q \leq p$ et $K_1 < K_2 < ... < K_{q-1}$
  ($P_1,P_2,...,P_{q-1},P_q$ sont les pointeurs vers les noeuds descendant)



Un noeud peut donc avoir au maximum p descendants

Hauteur pour n valeurs: $\log_p(n)$ –> complexité de recherche $O(\log_p(n))$

Remarque:

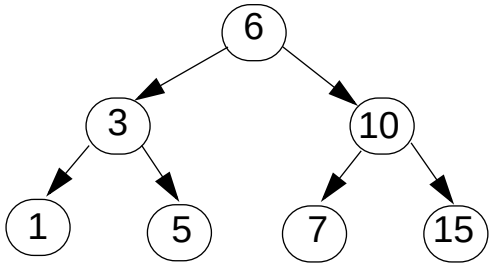
Nous supposons que la clé est unique. Cette restriction peut être levée, mais il faut changer légèrement les formules.

Problème de l’équilibre

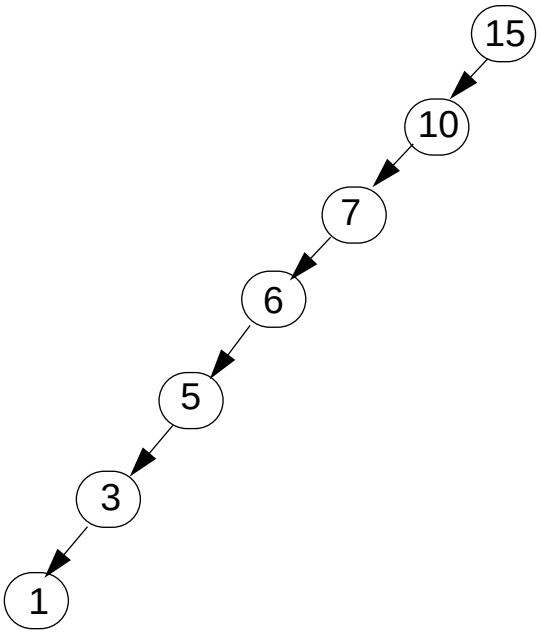
La forme de l’arbre dépend de l’ordre dans lequel on introduit les données.

Ex. avec arbre de recherche d’ordre 2 (arbre binaire):

- suite de nombres 6,3,1,10,7,5,15 –> arbre équilibré



suite de nombres 15,10,7,6,5,3,1–> arbre déséquilibré



Problème de l’équilibre (suite)

Conséquences

Complexité de recherche:

- $O(\log_2(n))$ si arbre équilibré (ex: max. 3 comparaisons)
- $O(n)$ si arbre déséquilibré (ex: max. 7 comparaisons)

Solutions:

- algorithmes de rééquilibrage
- B-arbres

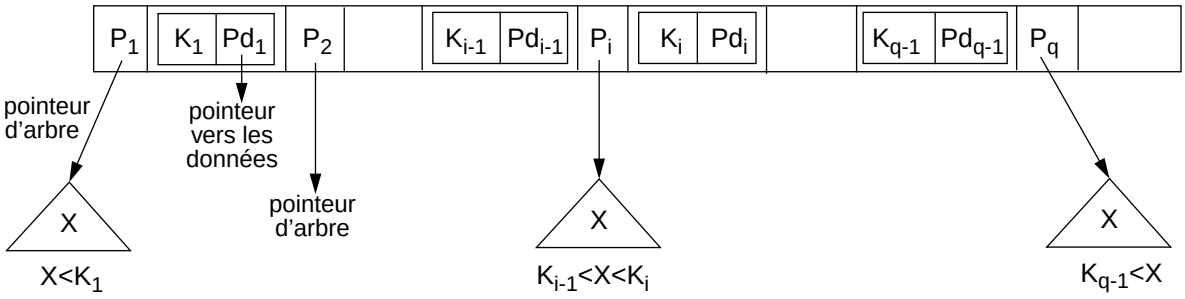
Les B-arbres

Arbre de recherche avec propriétés supplémentaires:

- équilibré, bien rempli(noeuds au moins à moitié pleins)

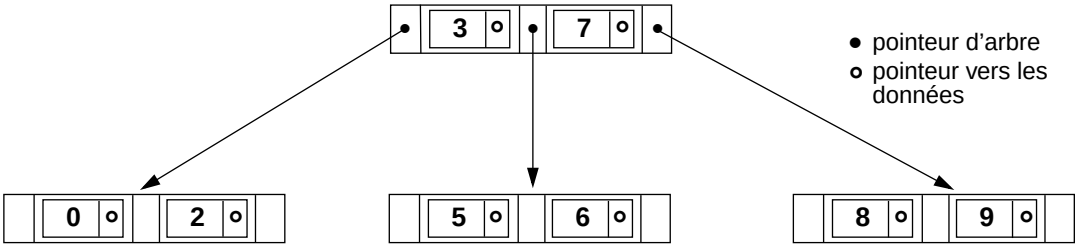
B-arbre d'ordre p

- chaque noeud interne a la forme: $<P_1,<K_1,Pd_1>P_2,<K_2,Pd_2>,...,P_{q-1},<K_{q-1}Pd_{q-1}>,P_q>$ avec $q \leq p$ ($Pd_1,Pd_2,...,Pd_{q-1}$ sont des pointeurs vers des enregistrement de données)



- La racine a au moins 2 descendants - sauf si c'est une feuille
- Chaque noeud intérieur a au max. p descendants et au min. [p/2] descendants
- Toutes les feuilles apparaissent au même niveau

Exemple: un B-arbre d'ordre 3



Les B-arbres: performances

Complexité de la recherche: O($\log_p(n)$)

Remarque:

On réserve un bloc disque pour le stockage de chaque noeud.

Calcul du nombre de points d'entrée. Exemple:

Hypothèse:

- longueur de la clé: 9 octets
- taille bloc disque: 512 octets
- taille pointeur vers un bloc: 6 octets

–> ordre du B-arbre: $(p * 6) + ((p-1) * (6+9)) \leq 512$
–> p = 25

Hypothèse:

- les noeuds sont remplis à 69% (chiffre obtenu par simulation) –> remplissage moyen 17 ptr / noeuds

–> nombre d'entrées en fonction du niveau de l'arbre:

racine:	1 noeud	16 entrées	17 pointeurs
niveau 1:	17 noeuds	272 entrées	289 pointeurs
niveau 2:	289 noeuds	4624 entrées	4913 pointeurs
niveau 3:	4913 noeuds	78608 entrées	83521 pointeurs

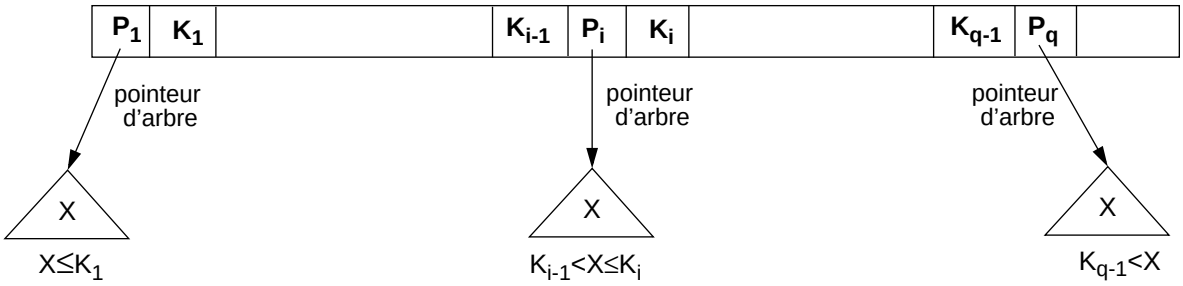
Les B^+-arbres

La plupart des implantations des index de fichiers utilisent une variation du B-arbre appelée B^+-arbre.

Particularités:

- les pointeurs vers les enregistrements de données apparaissent uniquement au niveau des feuilles
- les feuilles sont reliées entre-elles par des pointeurs

Noeud interne:



Noeud feuille :

Les B^+-arbres: performances

Toutes les valeurs de clé apparaissent dans les noeuds feuille de l'arbre

–> parcours séquentiel dans l'ordre des clés très rapide;

–> certaines valeurs de clés sont stockées plusieurs fois;

–> à la suppression il faut également ajuster les blocs supérieurs.

Complexité de la recherche: O($\log_p(n)$) (idem B-arbre)

Calcul du nombre de points d'entrée. Exemple:

(on prend les mêmes hypothèses que pour les B-arbres)

–> ordre du B^+-arbre: $(p * 6) + ((p-1) * 9) \leq 512$
–> p = 34

–> remplissage moyen: 23 pointeurs par noeud

–> nombre d'entrées en fonction du niveau de l'arbre:

racine:	1 noeud	22 entrées	23 pointeurs
niveau 1:	23 noeuds	506 entrées	529 pointeurs
niveau 2:	529 noeuds	11638 entrées	12167 pointeurs
niveau 3:	12167 noeuds	267674 entrées	

(remarque: le niveau 3 correspond au niveau feuille)

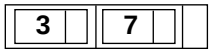
Insertion dans les B^+-arbres

Insérer un enregistrement avec k pour valeur de clé:

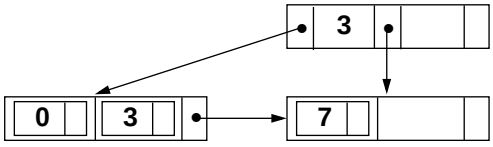
1. En partant de la racine, chercher la feuille f qui devra contenir k
2. insérer k dans f
3. si débordement de f:
   - éclater la feuille f en deux feuilles f' et f" en redistribuant les valeurs de clé de manière $f' \leq k' < f"$ (k' est la valeur médiane de $f' \cup f"$)
   - insérer la valeur médiane de $f' \cup f"$ k' dans le noeud parent
   - si le noeud parent deborde, l'éclater à son tour (les éclatements peuvent éventuellement se propager jusqu'à la racine)

Insertion dans les B$^+$-arbres : exemple

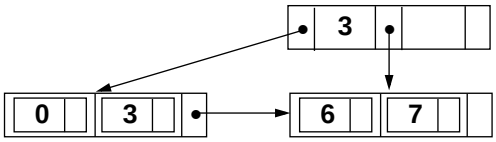
Insertion de 3 et 7



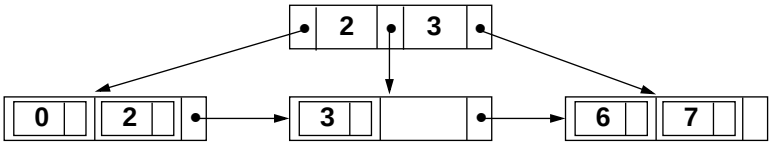
Insertion de 0: débordement, création d'un nouveau niveaux



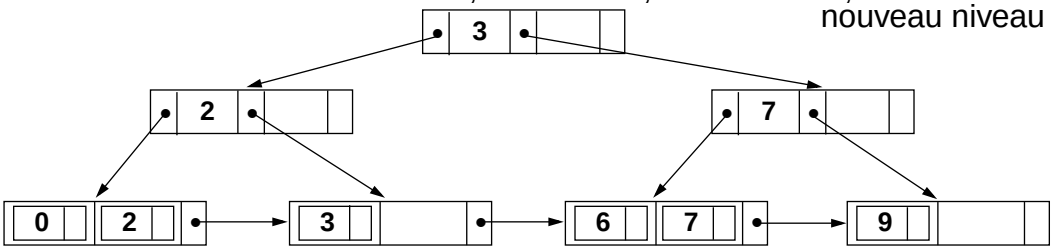
Insertion de 6



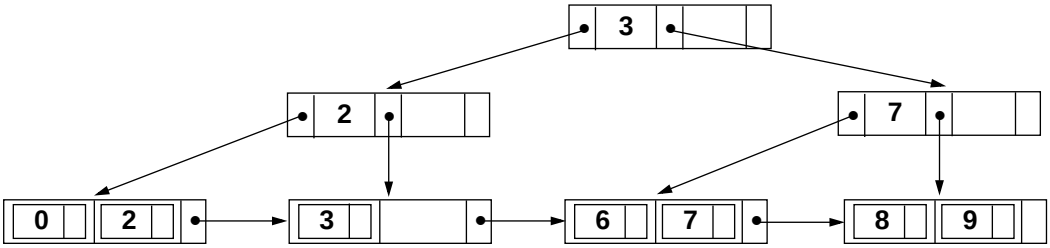
Insertion de 2: débordement, éclatement, redistribution



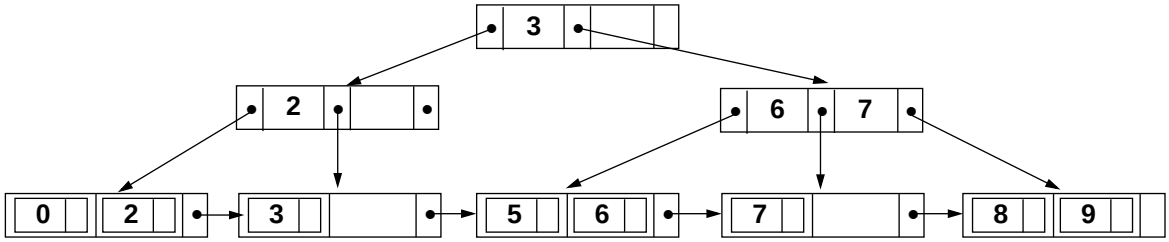
Insertion de 9: débordement, éclatement, redistribution, création d'un nouveau niveau



Insertion de 8



Insertion de 5: débordement, éclatement, redistribution

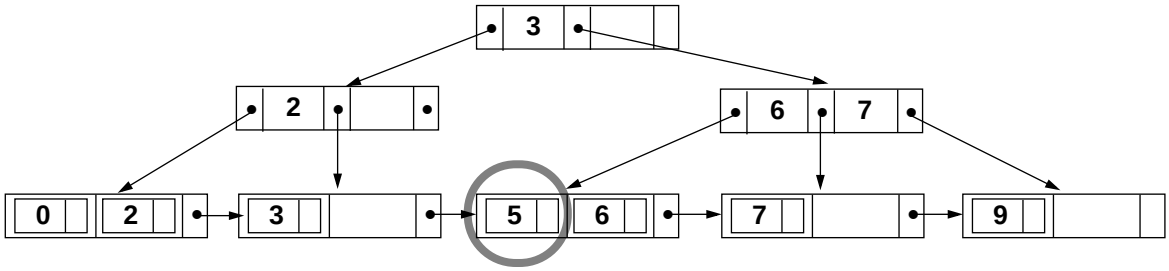


Suppression dans les B^+-arbres

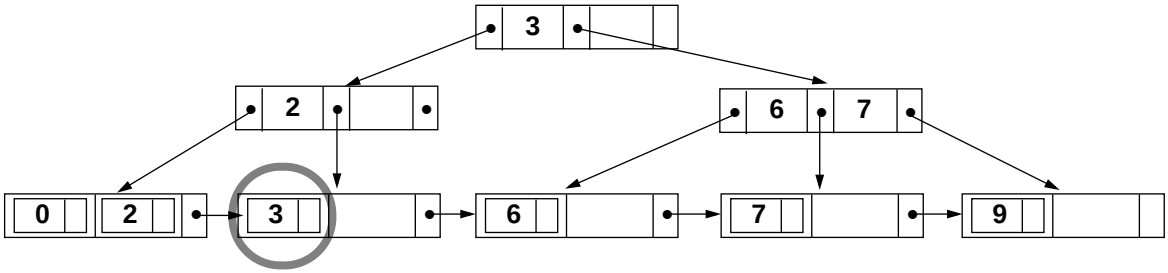
Supprimer un enregistrement avec k pour valeur de clé:

1. En partant de la racine, chercher la feuille f qui contient k
2. supprimer k et le pointeur vers l'enregistrement de données associé
3. supprimer k dans tous les noeuds parents où elle apparaît
4. si sous-occupation de f (i.e. f moins qu'à moitié plein):
   - redistribuer les valeurs de clé sur les feuilles de l'arbre
   - si la redistribution est impossible, fusionner f avec la feuille adjacente
5. si sous-occupation d'un noeud parent :
   - idem feuille
     (les fusions peuvent éventuellement se propager jusqu'à la racine)

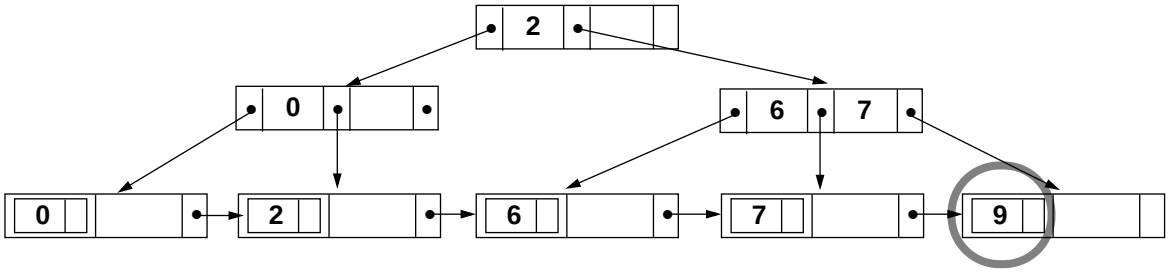
Suppression dans les B$^+$-arbres: exemple



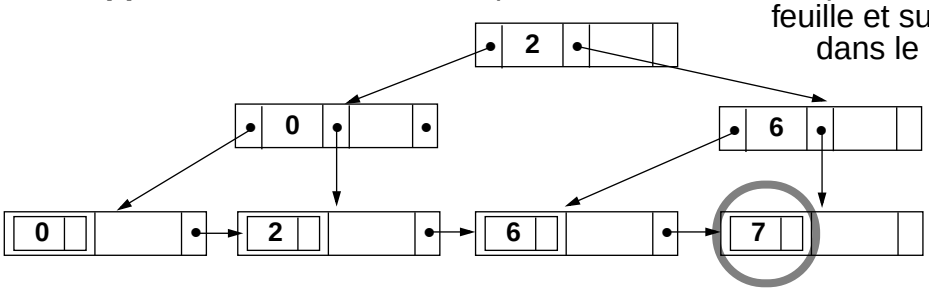
Suppression de 5



Suppression de 3: sous-occupation, redistribution



Suppression de 9: sous-occupation, redistribution impossible, fusion de la feuille et supression de 7 dans le noeud interne



Suppression de 7: sous-occupation, redistribution impossible, fusion de la feuille, fusion de niveaux