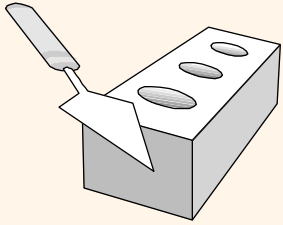


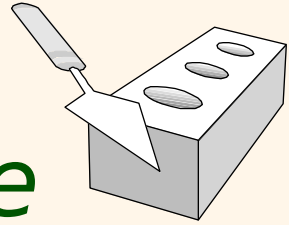
Bases de Données Avancées

Ioana Ileana

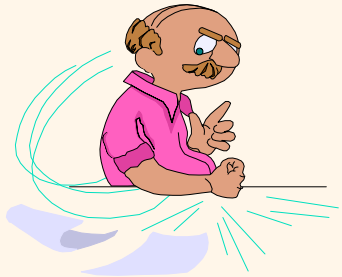
Université Paris Descartes

Cours basé sur le livre (et les diapositives de):
Database Management Systems 3ed, R. Ramakrishnan et J. Gehrke

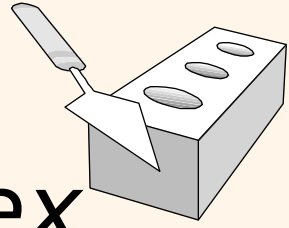
LIPADE
Laboratoire d'Informatique PAris DEscartes



Cours 4: Index de type arbre (tree indexes) 1ère partie



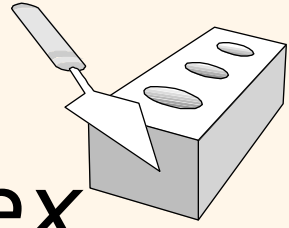
- Diapos traduites et adaptées du matériel fourni en complément du livre Database Management Systems 3ed, par Ramakrishnan et Gehrke ; un grand merci aux auteurs pour la réalisation et la disponibilité de ce matériel !
- Les diapos originales (en anglais) sont disponibles ici :
  http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- Plus particulièrement, ce cours touche aux éléments dans les Chapitres 8 et 10 du livre ci-dessus; lecture conseillée! ;)



Limite des Heap Files; les index

Les Heap Files (fichiers non triés): adaptés à la demande de lister tous les records dans un ordre aléatoire mais peu performants pour trouver des records respectant des conditions :

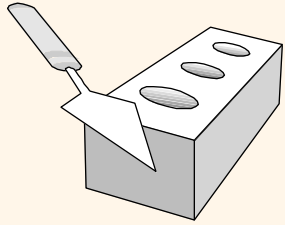
- Recherches / sélections de type « égalité » (champ = valeur)
- Recherches / sélections de type « intervalle » (champ > valeur1 et/ou champ < valeur2)



Limite des Heap Files; les index

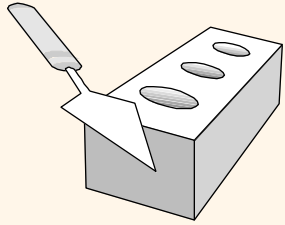
Plus efficaces pour les sélections à conditions :

- Fichiers triés (Sorted Files) – mais les mises à jour très coûteuses !
- Index: Structures de données qui organisent les records pour accélérer les recherches dont les conditions portent sur certains champs qu'on appelle « clés de recherche » (search keys)
  - Grand avantage par rapport aux fichiers triés: mise à jour beaucoup plus rapide! (mais aussi meilleures performances de recherche!)



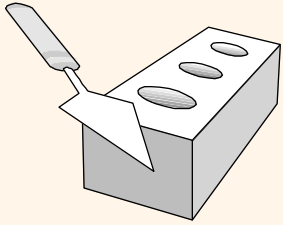
Les index

- Un *index* associé à un fichier de records (=une relation) accélère les sélections / recherches qui concernent les *champs qui forment la clé de recherche (search key fields)* de l'index.
  - Tout sous-ensemble des champs d'une relation peut former la clé de recherche d'un index associé à la relation.
  - *La clé de recherche n'est pas la même chose que la clé* d'une relation (ensemble minimal de champs qui identifient de manière unique un tuple / record de la relation).



Les index

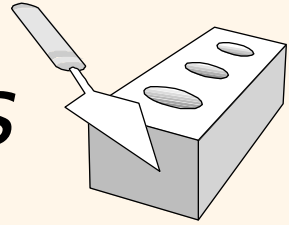
- Un index contient un ensemble d'*entrées de données (data entries)*, et permet de retrouver de manière efficace / rapide toutes les entrées **k*** qui correspondent à des données (records) qui ont une valeur **k** de la clé
- **Les grandes catégories d'index : de type arbre (tree index) ou bien basés sur une fonction de hachage (hash index)**
- Tout comme pour les fichiers de records, en vue de leur stockage sur le disque, on va regrouper le contenu des index *par pages* !



*Contenu d'une entrée de données **k*** dans un index*

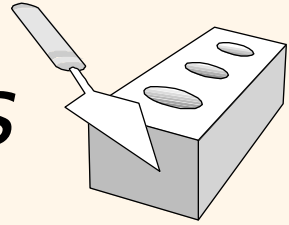
- Dans une entrée de données k* on peut stocker:
  - Le record correspondant (qui a la valeur **k** pour la clé), ou
  - <**k**, rid du record avec la valeur clé **k**>, ou
  - <**k**, liste des rids des records avec la valeur clé **k**>
- Le choix parmi ces alternatives est orthogonal à la technique d'indexation, qui elle vise simplement une manière efficace de trouver les entrées qui correspondent à une clé **k**.

Alternatives pour le contenu des entrées de données

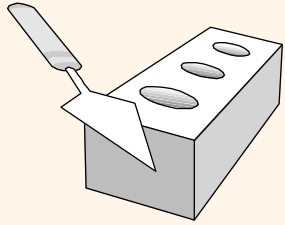


- Alternative 1 (stocker le record directement dans l'entrée):
  - Si cette alternative est choisie, l'index lui-même devient une implémentation de fichier de records (à la place d'un Heap File ou d'un fichier trié…).
  - Au plus un index parmi ceux associés à une relation peut utiliser cette alternative (sinon, les records sont dupliqués → stockage redondant et danger d'inconsistance!)
  - Si les records sont très volumineux, le nombre de pages qui contiennent des entrées de données sera élevé, ce qui typiquement implique aussi que le volume de l'information auxiliaire dans l'index sera élevé (donc gros volume global!)

Alternatives pour le contenu des entrées de données

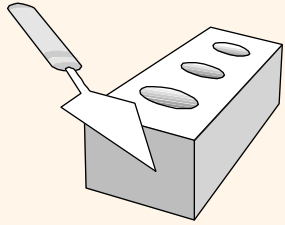


- Alternatives 2 and 3 (“pointage” des records par des rids stockés dans les entrées de données):
  - Cela mène a des entrées de données de petite taille, donc mieux que l’alternative 1, notamment si les records sont gros et la clé est en revanche petite. La partie de l’index utilisée pour guider la recherche, qui dépend de la taille des entrées de données, sera aussi beaucoup plus petite qu’en utilisant l’alternative 1.
  - L’alternative 3 (grouper tous les pointeurs de records avec la même clé dans une même entrée) est plus compacte que l’alternative 2, mais nécessite des entrées de données de taille variable même si les clés sont de longueur fixe !



Classification des index

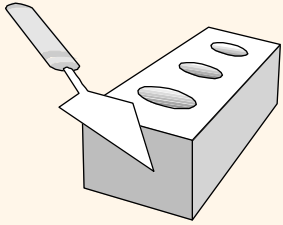
- *Primaire* vs. *secondaire*: Si la clé de recherche contient la clé primaire, l'index s'appelle "primaire"
  - Également, notion d'index *"unique"* : la clé de recherche contient une clé candidate
- *Clustered* vs. *unclustered*: Si l'ordre des records est le même que (ou "proche de") celui des entrées de données correspondant à ces records, alors l'index s'appelle *clustered*.
  - En réalité, on obtient en pratique des index clustered quasi uniquement si on utilise l'alternative 1
  - Pour une recherche de type « intervalle », le coût de l'accès aux records via une l'index varie beaucoup en fonction du type clustered / unclustered de l'index!



Les index de type arbre (tree indexes)

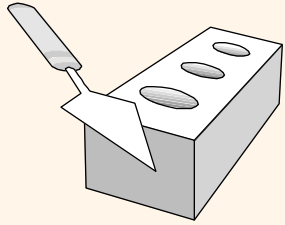
- Les index de type arbre permettent de faire efficacement des *recherches par intervalle (range searches)* et des *recherches par égalité (equality searches)*.
  - Les index hash ne gèrent pas les intervalles, mais sont en général plus efficaces pour les recherches par égalité!
- Les feuilles des tree indexes contiennent des entrées de données (data entries) k* (choix parmi les trois alternatives!)
- Les nœuds intermédiaires des tree indexes contiennent des entrées d'index (index entries) : typiquement sous la forme de couples (clé, pointeur vers un nœud fils). Ces entrées d'index servent à « diriger la recherche vers les feuilles ».

Les index de type arbre (tree indexes)

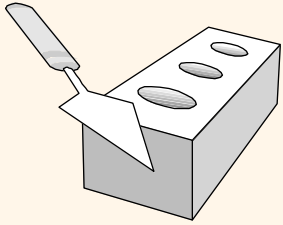


- Dans ce cours, deux structures de tree index :
  - *ISAM*: structure statique (perd en général en perfos si beaucoup d'insertions).
  - *B+ tree*: dynamique, s'adapte bien aux insertions et suppressions → le type d'index le plus utilisé !

Intuition pour les tree indexes

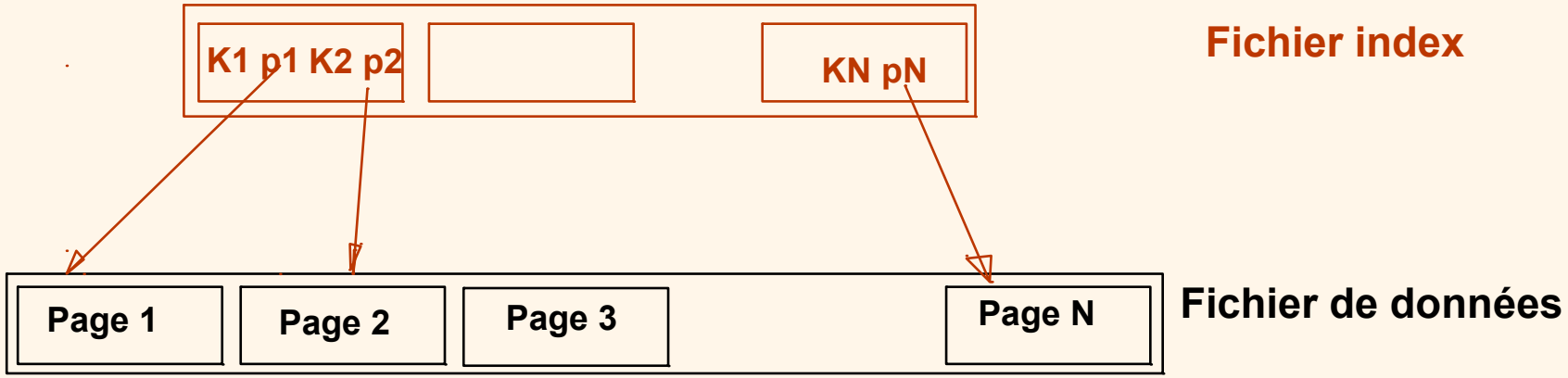


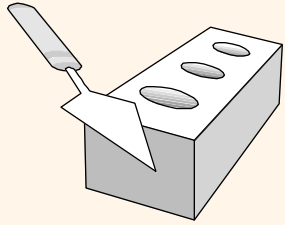
- Supposons qu'on dispose d'un fichier des étudiants trié par la moyenne et que toutes les moyennes sont distinctes.
- Supposons également la recherche suivante : trouver l'étudiant qui a la moyenne de 10,61.
  - Comment faire notre recherche de manière efficace?
- Si le fichier est de taille importante, même la recherche dichotomique peut devenir coûteuse ! Pensez au nombre de pages à lire depuis le disque !
- Idée très simple pour améliorer les perfos (→ diminuer le nombre de pages à lire): créer un nouveau fichier qui contient une entrée pour chaque page de notre fichier de données original
- Les entrées de notre nouveau fichier seront de la forme (première valeur de clé sur la page, pointeur vers la page) ; elles seront aussi triés sur la clé (moyenne) !



Intuition pour les tree indexes

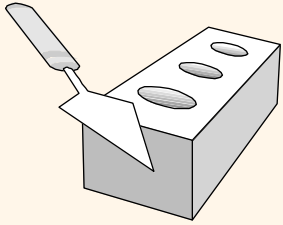
- Le nouveau fichier fonctionnera comme un *index* de notre fichier original !
  - Dirigera / filtrera les recherches
- Entrées d'index : (clé, pointeur vers page) ; elles seront regroupées sur des pages aussi (car on veut stocker notre fichier d'index sur le disque) !
- Par construction, les entrées du fichier index seront triées aussi !





Intuition pour les tree indexes

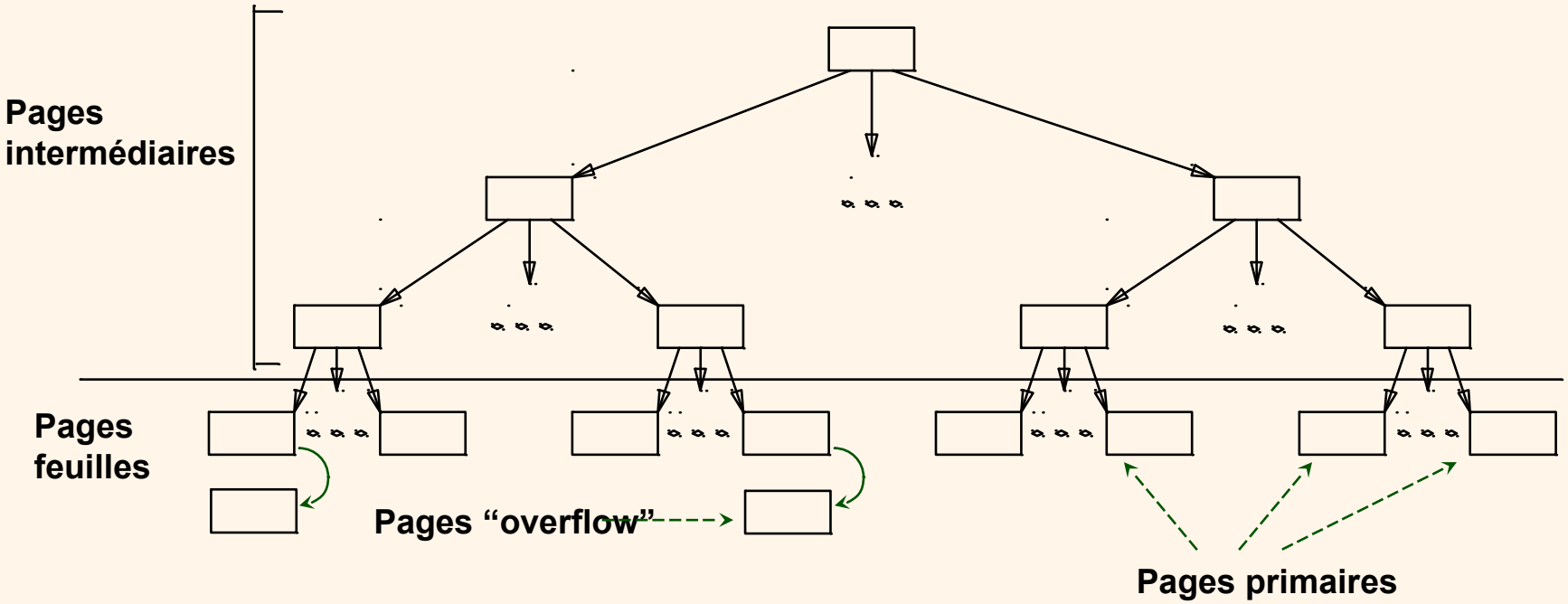
- Recherche avec notre nouveau fichier « index », pour trouver tous les records correspondant à une moyenne de 10,61 :
  - Recherche dichotomique sur les entrées du fichier index, pour trouver *la clé qui correspond à la page ou devrait se trouver le record avec la moyenne 10,61.*
    - Comment faire ? Quelle condition sur la clé en question ?
  - Prendre la page de records correspondante (via le pointeur) et refaire une recherche dichotomique sur cette page pour identifier l'entrée avec 10,61.
    - Existe-t-il toujours une telle entrée ?



Intuition pour les tree indexes

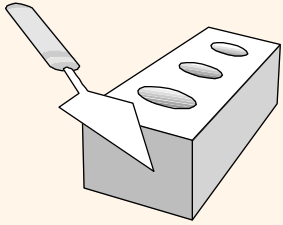
- Entrées dans le fichier index : plus petites, et moins nombreuses →moins de pages à charger en mémoire, et recherche dichotomique plus rapide !
- MAIS peut devenir coûteux aussi
- *IDÉE : pourquoi ne pas répéter la procédure en construisant des « index des pages d'index » et ainsi de suite jusqu'à ce qu'on obtienne juste une page ?*
- *=> STRUCTURE ARBORESCENTE !*

ISAM (Index Sequential Access Method)

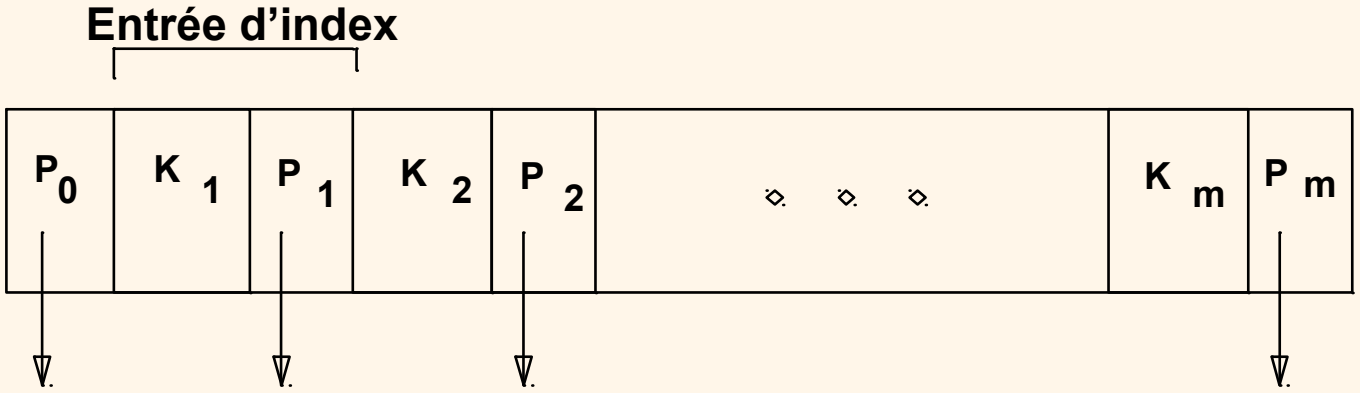


- Chaque nœud = une page
- Entrées de données: dans les pages feuilles
- Des pages « overflow » sont chaînées à certaines pages feuilles
- A l'origine, utilise l'alternative (1), mais (2) et (3) possibles

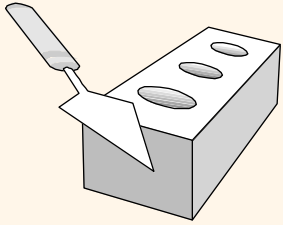
ISAM : nœuds intermédiaires



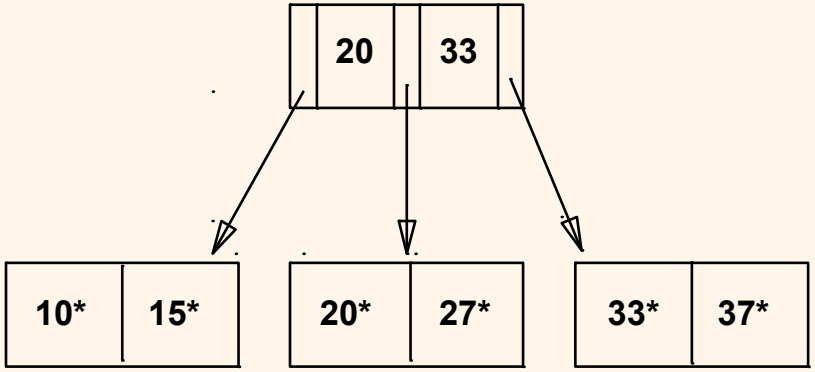
- En plus des entrées d'index sous la forme (clé, pointeur), les nœuds intermédiaires comporteront un pointeur supplémentaire ; cela nous permet de « pointer N+1 pages » avec N clés !
- Ainsi les clés qui figurent dans les entrées fonctionneront comme des *séparateurs des pages pointées par les pointeurs gauche et droite*
  - *Pour un pointeur P_i, tous les clés dans le sous-arbre pointé auront valeur strictement inférieure à la clé K_{i+1} (si elle existe) et supérieure ou égale à la clé K_i (si elle existe)*



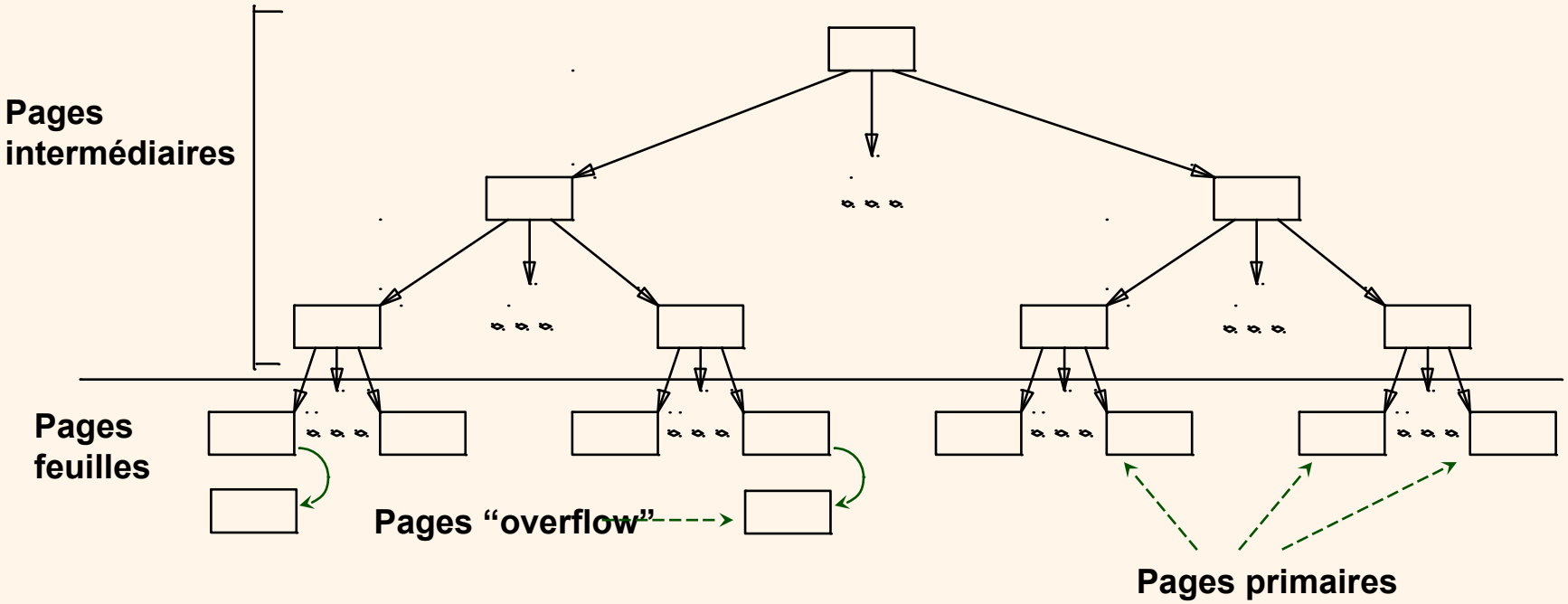
ISAM : nœuds intermédiaires



- En plus des entrées d'index sous la forme (clé, pointeur), les nœuds intermédiaires comporteront un pointeur supplémentaire ; cela nous permet de « pointer N+1 pages » avec N clés !
- Ainsi les clés qui figurent dans les entrées fonctionneront comme des *séparateurs des pages pointées par les pointeurs gauche et droite*
  - *Pour un pointeur P_i, tous les clés dans le sous-arbre pointé auront valeur strictement inférieure à la clé K_{i+1} (si elle existe) et supérieure ou égale à la clé K_i (si elle existe)*

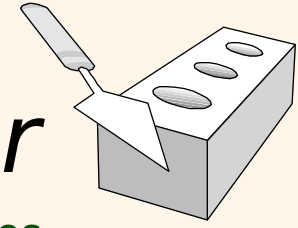


ISAM : création du fichier

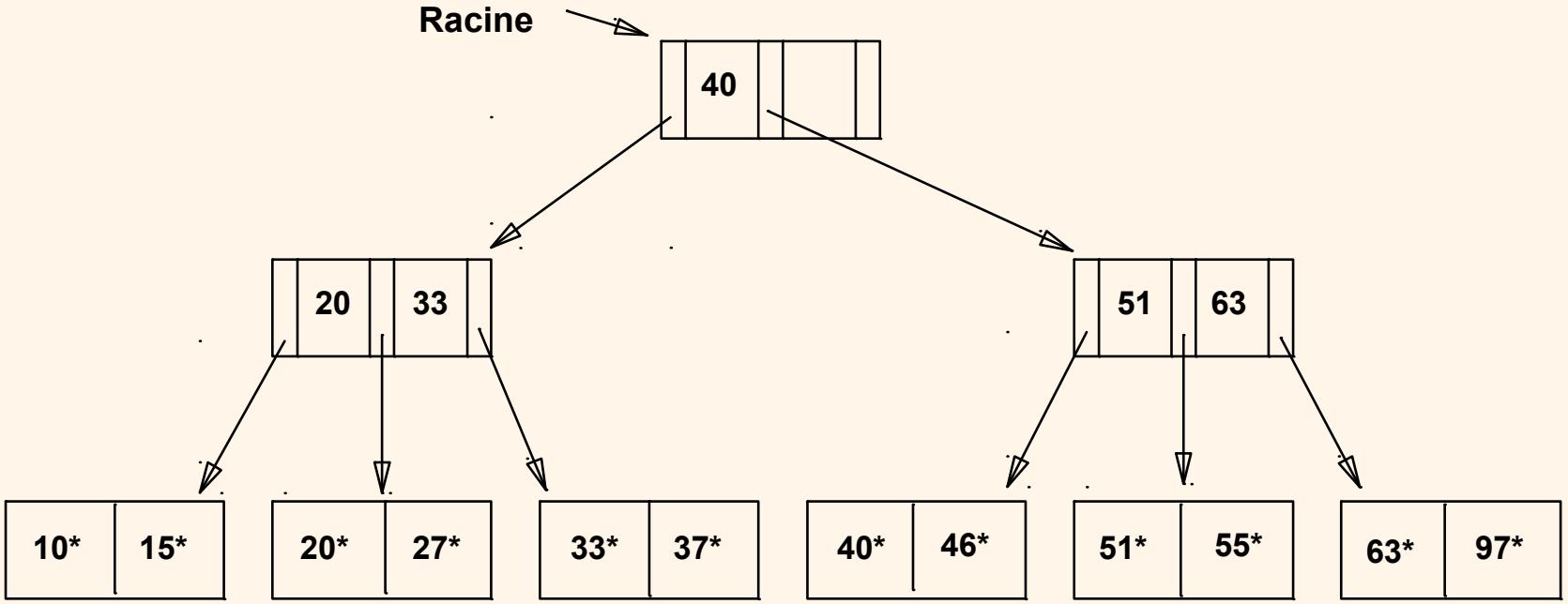


- D'abord les pages feuilles allouées de manière séquentielle et remplies des records triés.
- Puis les pages intermédiaires créées
- Enfin réserver de l'espace pour l'allocation futures des pages overflow

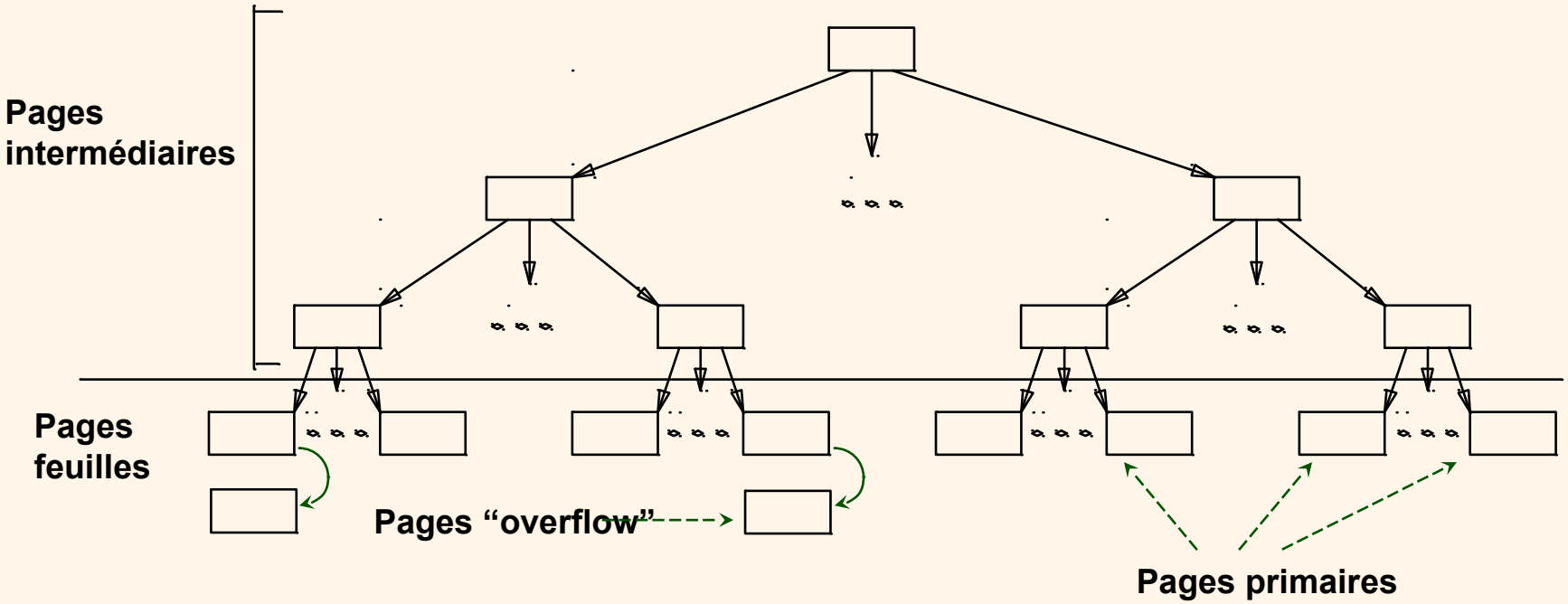
Exemple ISAM: création du fichier



- Rappel: pages feuilles primaires allouées de manière séquentielle, elles sont « fixes », leur nombre est fixe et connu à la création!
- On « fait remonter » la première valeur dans la première page feuille du sous-arbre ; on commence toujours par le deuxième sous-arbre
- Dans notre exemple : deux entrées / page (feuilles et index)



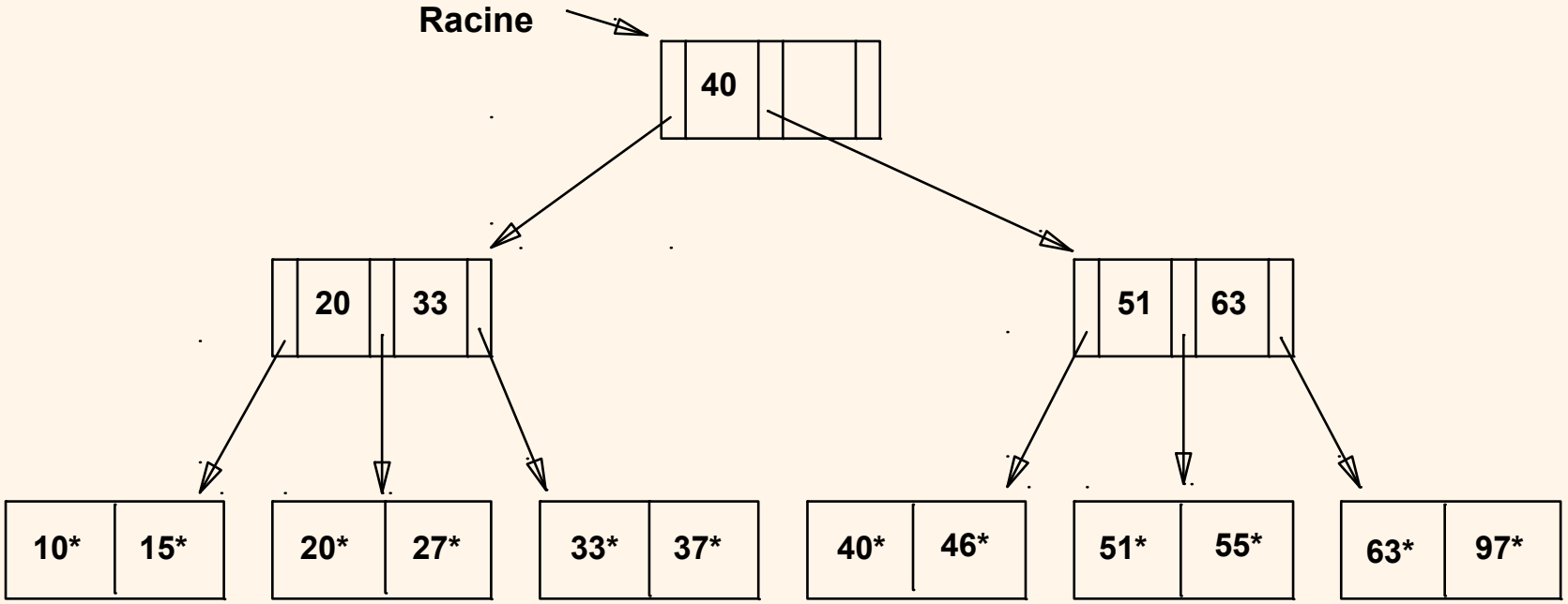
ISAM : recherche



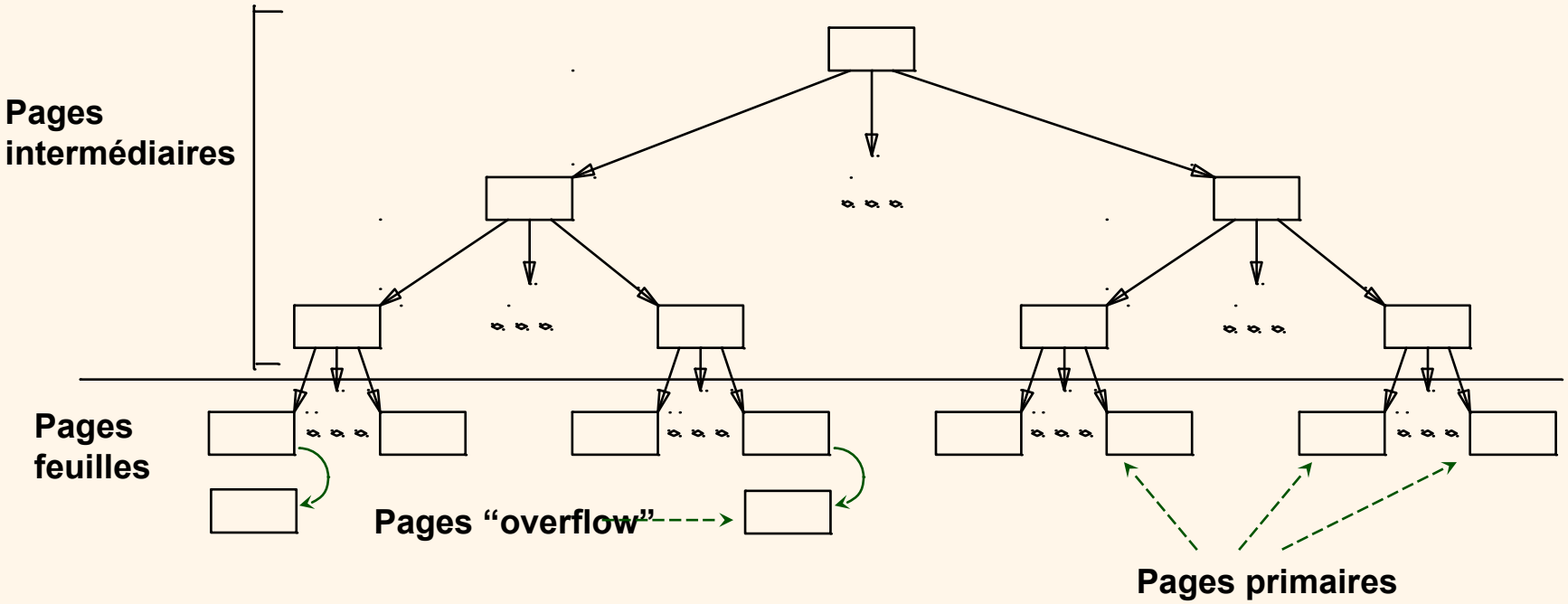
- *Recherche:* démarrer à la racine, utiliser des comparaisons sur les clés pour "naviguer" jusqu'aux feuilles; Coût (en ignorant les pages overflow): $\log_F N$; F = # entrées/page index, N = # pages feuilles (cela correspond à la hauteur de l'arbre)

Exemple ISAM: recherche

- Recherche de 27* ?
- Recherche de 38* ?
- Recherche de toutes les entrées correspondant à des valeurs supérieures ou égales à 46 ? (rappel : *les pages feuilles sont allouées de manière séquentielle*!)

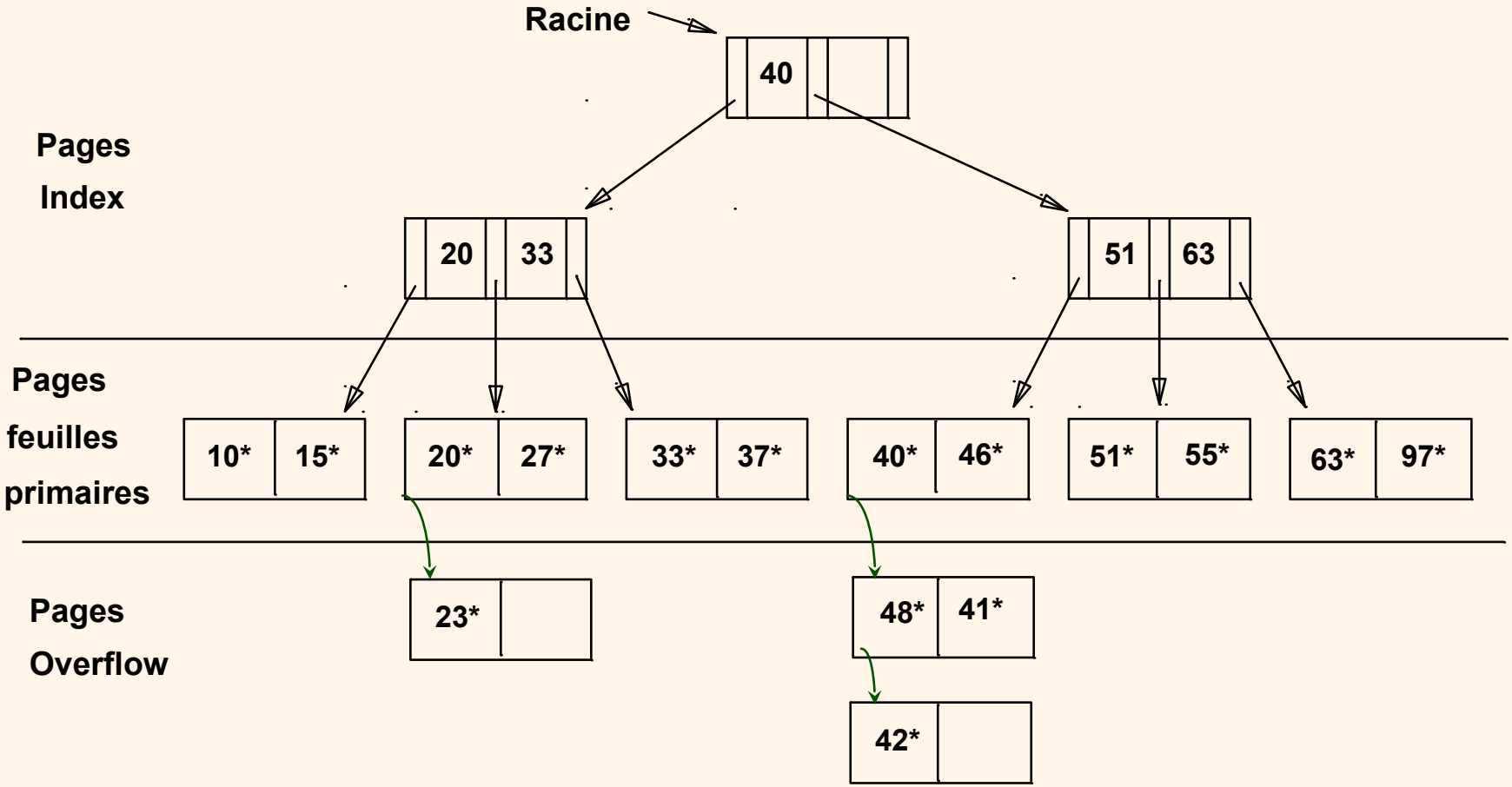


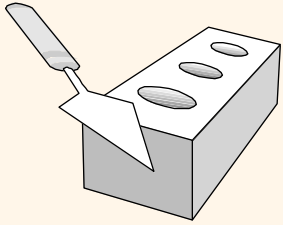
ISAM : insertion et suppression



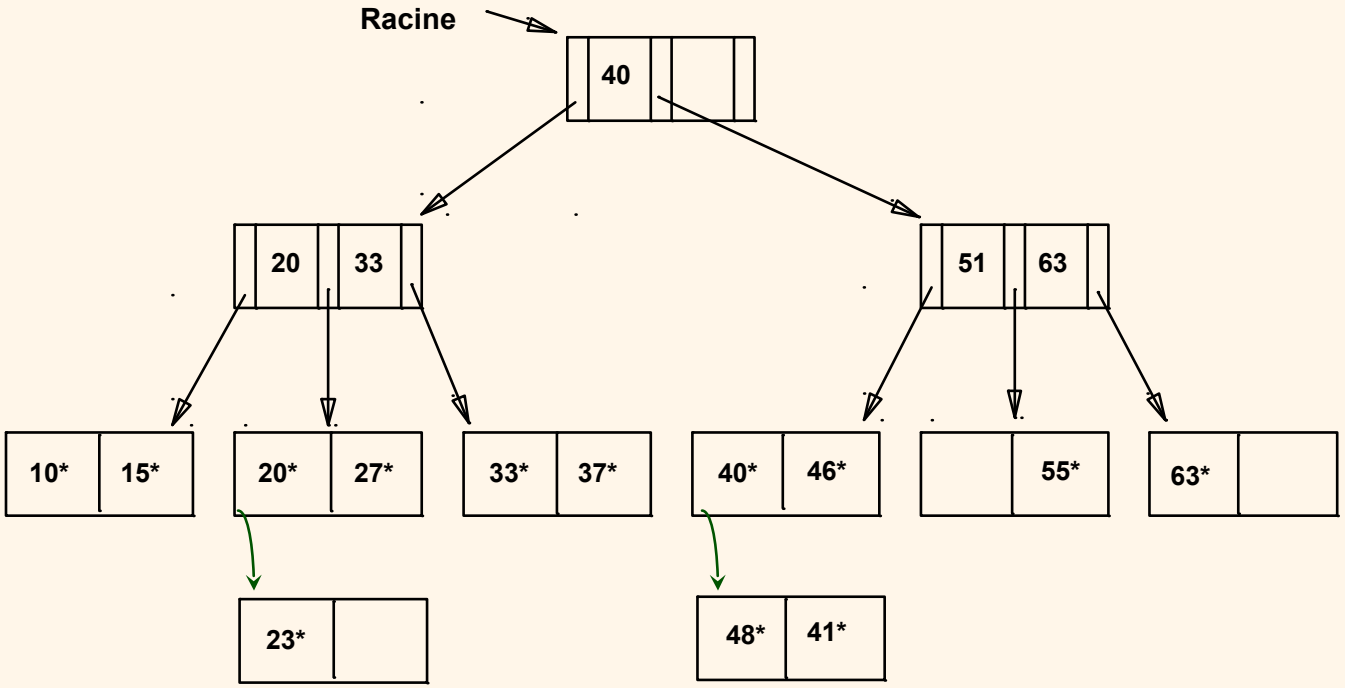
- **Structure statique**: *les insertions/suppressions affectent uniquement les pages feuilles, ce qui explique la présence des pages overflow!*
  - *Insertion:* trouver la page feuille qui va bien et y insérer la nouvelle entrée ; rajout de page overflow si nécessaire !
  - *Supression:* trouver la feuille et supprimer ; si page overflow et qu'elle devient vide, la désallouer (mais on ne supprime jamais une page primaire)

Exemple ISAM : après insertion de 23, 48*, 41*, 42* ...*



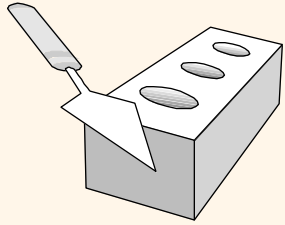


... Puis suppression de 42*, 51*, 97*



☛ *51 reste dans les nœuds index !*

ISAM : le problème des pages overflow



- ISAM : structure statique, les pages primaires feuilles sont en nombre fixe et allouées « au tout début », à la construction de la structure
- Pour accommoder les insertions, des pages overflow sont créées et remplies avec les clés / records insérés ! Pour minimiser le coût, souvent ces pages ne sont pas triées (les entrées y sont donc « dans le désordre »)
- → Si « longue chaîne de pages overflow », cela pénalise le temps de recherche !!!
  - Pour une feuille, on peut avoir à examiner en réalité plein de pages ; comme il n'y a pas eu de tri, dans ces pages on ne peut, de plus, pas utiliser la recherche dichotomique !