ALGORITHMES DE RECHERCHE

I.S.I.A.

COURS 3

Franck.Nielsen@sophia.inria.fr

Année 1995-1996

INRIA SOPHIA ANTIPOLIS

Introduction

→Grande capacité de stockage des ordinateurs actuels (Giga octets)

→Besoin de chercher *rapidement* des données satisfiant certaines propriétés.

→Classer, archiver, insérer, supprimer, ...

Minimiser le temps de recherche

Temps de recherche dépendant de l'organisation des données. Si beaucoup de requêtes alors on peut se permettre de construire des SDD coûteuses.
→Données gérées de façon dynamique (fichiers de banque, ...)

Recherche Associative

Soit $\mathcal{E}$ un ensemble d'éléments. En général, à chaque $x \in \mathcal{E}$ on lui associe une **clef** unique le caractérisant complétement. On désire chercher un ensemble d'éléments satisfiant certaines propriétés. Soit $\mathcal{C}$ l'ensemble des clefs.

Si le critère de recherche porte sur la valeur de la clef, on parle de **recherche associative**.

Exercice 1 *Donner quelques exemples d'applications de recherche associative*

Dans les BDD on utilise souvent les recherches **multi-critéres**.

On définit une table $\mathcal{T}$ comme un sous-ensemble de $\mathcal{C} \times \mathcal{E}$. Si toute la table est contenue en mémoire centrale alors on parle de **recherche interne**, sinon **recherche externe** (à l'aide d'un disque, ...)
$\rightarrow$notions de mémoire paginée, cache, ...

Fonctions abstraites

Si chaque clef caractérise de manière unique un élément alors on peut définir sans ambiguïté les opérations abstraites suivantes:

- Insérer() : $\mathcal{C} \times \mathcal{E} \times \mathcal{T} \rightarrow \mathcal{T}$

$$\text{Supprimer}(c, \text{Insérer}((c', e), T), T) : \begin{cases} T & \text{si } c = c' \\ \text{Insérer}((c', e), \text{Supprimer}((c, T))) & \text{sinon} \end{cases}$$

- Rechercher() : $\mathcal{C} \times \mathcal{T} \rightarrow \mathcal{E}$

$$\text{Rechercher}(c, \text{Insérer}((c', e), T)) : \begin{cases} e & \text{si } c = c' \\ \text{Rechercher}(c, T) & \text{sinon} \end{cases}$$

avec

$$\text{Rechercher}((c, T)) = \begin{cases} e & \text{si } (c, e) \in T \\ \text{non trouvé} & \text{sinon} \end{cases}$$

Recherche dans une liste

On implante une table par une liste.
→Première méthode: recherche séquentielle avec sentinelle.

typeTable=**array**[1..n] **of** Cellule;
Cellule=**record**
c:Clef;
e:Element;
end;

function Recherche(T:Table;c:Clef):1..$n+1$;
var
i:1..$n+1$;
begin
$i := 1$;
$T[n+1].c := c$;
while $c <> T[i].c$ **do** $i := i + 1$;
Recherche := i;
end;

Complexité $c(n) = (n+1)(1-p) + \Sigma_{i=1}^{n} ip_i$. Choisir $p_i > p_{i+1}$ pour minimiser $c(n)$.

→Recherche **auto-adaptative**, stratégies globales, locales de placement des éléments.

Recherche Dichotomique

Relation d'ordre total $>$ sur les clefs et liste triée.

function RechercheOrdonnée(T:Table;c:Clef):$1..n+1$;
var
i:$1..n+1$;
begin
$i := 1$;
$T[n+1].c := c$;
while $c > T[i].c$ **do** $i := i+1$;
Recherche := i;

→s'arrête avant la sentinelle. Complexité?
→**recherche dichotomique**

Principe: on compare la clef c à chercher avec la clef médiane du tableau c_m (à la manière du dictionnaire):

1. Si $c = c_m$ alors on a trouvé l'élément.
2. Si $c > c_m$ alors on cherche c dans $T[c_{m+1}..c_n]$
3. Si $c < c_m$ alors on cherche c dans $T[1..c_{m-1}]$

→diminue la taille du tableau: $n, \frac{n}{2}, \frac{n}{4}, ...$

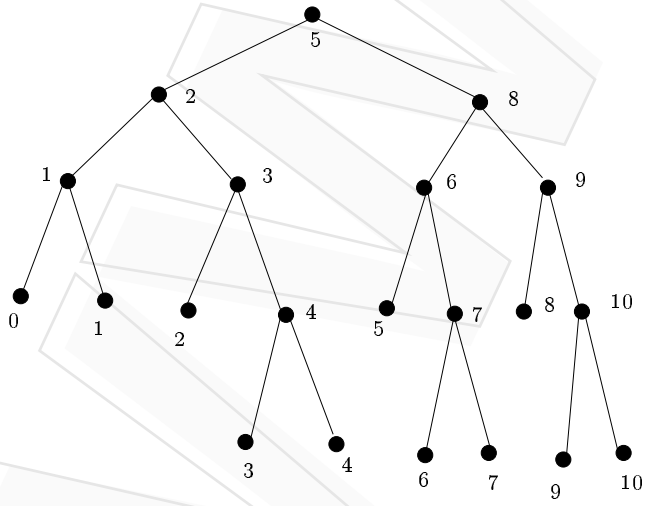
Recherche Dichotomique (le programme)

function RechercheDicho(T:Table;c:Clef;g, d:0..n):0..n;
var
i:1..n; m:0..n;
begin
if $(g \leq d)$**then begin**
$m := (g + d)$**div**2;
if $c = T[m].c$ **then** $RechercheDicho := m$;
else
if $c < T[m].c$ **then**
$RechercheDicho := RechercheDicho(T, c, g, m-1)$
else
$RechercheDicho := RechercheDicho(T, c, m+1, d)$
end
else $RechercheDicho := 0$
end;

Exercice 2 *Executer cet algorithme avec* $c = 32$ *pour* $T = [1..50]$. *Que se passe-t-il si on utilise une liste chaînée? Démontrer la correction totale de l'algorithme.*

Complexité de la recherche dichotomique

→dans le cas le pire.
→Arbre de décision (la clef cherchée est considérée comme générique).



→chemin dans cet arbre: déroulement.

Complexité & Optimalité

$\rightarrow$Recherche dichotomique en $c(n) = \lfloor \log_2 n \rfloor + 1$. Puis on montre la borne inférieure $\Omega(\log n)$.
L'arbre de décision (binaire) à $f = n + 1$ feuilles d'où la hauteur minimale d'un arbre de décision $\log f = \log(n + 1) \geq \frac{1}{2} \log n$. La recherche dichotomique requiert $\Omega(\log n)$ opérations de comparaison (modéle de machine).
Soit h_n la hauteur de l'arbre de décision. Montrons par récurrence que $h_n = \lfloor \log n \rfloor + 1$. On a $h_2 = 2$.

- Si $n = 2p + 1$ alors $h_n = 1 + h_p$
- Si $n = 2p$ alors $h_n = 1 + \max\{h_p, h_{p-1}\} \leq 1 + h_p$

Si $n = 2p + 1$ alors $h_n = 1 + 1 + \lfloor \log p \rfloor = 1 + \lfloor \log 2p \rfloor$ mais $\lfloor \log 2p \rfloor = \lfloor \log 2p + 1 \rfloor$ et $h_n = \lfloor \log n \rfloor + 1$.
Si $n = 2p$ alors $h_n = 1 + 1 + \lfloor \log p \rfloor = 1 + \lfloor \log n \rfloor$.


Recherche par interpolation

→éviter de comparer à l'élément du milieu si les clefs ont une progression linéaire (par exemple pour chercher un mot dans le dictionnaire).

→besoin d'un modéle de distribution des données.

On peut choisir:

$$m = g + \frac{c - T[g].c}{T[d].c - T[g].c}(d - g)$$

→**Interpolation**
→En pratique marche bien pour beaucoup de données. En particulier si on utilise de la mémoire externe (disque).

Exercice 3 *Donner un exemple de recherche par interpolation qui prend beaucoup plus de temps que la recherche dichotomique.*

Méthodes arborescentes

→Les arbres de décision montrent que la recherche séquentielle s'apparente à un chemin de la racine à une feuille. Pourquoi ne pas organiser les données dans un *arbre de recherche* afin d'accélerer les recherches?

→Arbre binaire de recherche. A une même table correspond une multitude d'arbres de recherche. Si on connaît une loi de probabilité d'accés des clefs alors on peut essayer de construire **un arbre binaire de recherche optimal** afin d'optimiser le coût d'accés à une clef (→code de Huffman).

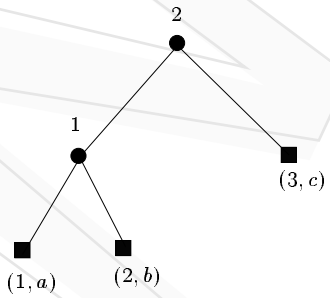
→Opérations Insérer(), Supprimer() et Rechercher() sont logarithmiques en fonction du nombre de nœuds de l'arbre de recherche. Tout le problème consiste donc à maintenir la **compacité** de l'arbre quand on insére, supprime un élément.

Arbres Ordonnés

Table dynamique implantée par un arbre strictement binaire où les éléments se trouvent dans les feuilles et les nœuds internes contiennent les clefs.
La définition récursive est alors (cf. CAML)

$$\mathcal{A}(\mathcal{C},\mathcal{E}) = (\mathcal{C}\times\mathcal{E})\cup\{c < A_g, A_d >\} \text{ et } A_g, A_d \in \mathcal{A}(\mathcal{C},\mathcal{E})$$

$$\boxed{2}<\boxed{1}<(1,a),(2,b)>,(3,c)>$$



Un arbre est **ordonné** ssi pour tout nœud la clef de la racine est supérieure à son fils gauche et strictement inférieure à son fils droit.

Fonctions abstraites

Insérer()

$$\text{Insérer}((c,e),[]) = (c,e)$$

$$\text{Insérer}((c,e),(c,e')) = \begin{cases} (c,e) & \text{si } c = c' \\ c[(c,e),(c',e')] & \text{si } c < c' \\ c'[(c',e'),(c,e)] & \text{si } c > c' \end{cases}$$

$$\text{Insérer}((c,e),c'[A_g,A_d]) = \begin{cases} c'[\text{Insérer}((c,e),A_g),A_d] & \text{si } c \leq c' \\ c'[A_g,\text{Insérer}((c,e),A_d)] & \text{sinon} \end{cases}$$

Supprimer()

$$\text{Supprimer}(c,[]) = []$$

$$\text{Supprimer}(c,(c',e)) = \begin{cases} [] & \text{si } c = c' \\ (c',e) & \text{sinon} \end{cases}$$

$$\text{Supprimer}(c,c'[A_g,(c'',e)]) = A_g \text{ si } c = c''$$

$$\text{Supprimer}(c,c'[(c'',e),A_d]) = A_d \text{ si } c = c''$$

$$\text{Supprimer}(c,c'[A_g,A_d]) = \begin{cases} c'[\text{Supprimer}(c,A_g),A_d] & \text{si } c \leq c' \\ c'[A_g,\text{Supprimer}(c,A_d)] & \text{sinon} \end{cases}$$

Rechercher()

$$\text{Rechercher}(c,[]) = ?$$

$$\text{Rechercher}(c,(c'e)) = e \text{ si } c = c'$$

$$\text{Rechercher}(c,c'[A_g,A_d]) = \begin{cases} \text{Rechercher}(c,A_g) & \text{si } c \leq c' \\ \text{Rechercher}(c,A_d) & \text{sinon} \end{cases}$$

Arbres Équilibrés – Arbres $BB[\alpha]$

$\rightarrow$Garantir une hauteur logarithmique de la hauteur de l'arbre afin de faire les opérations d'actualisation en temps logarithmique.
$\rightarrow$Un arbre de n nœuds dont la hauteur est $\Theta(\log n)$ est dit équilibré (exemple: arbre binaire complet, arbres 2-3, arbre AVL (Adelson-Velskii-Landis, arbres de Fibonnaci, arbres à équilibre borné (Nievergelt et Reingold.$\rightarrow$Arbres $BB[\alpha]$)...)

Les insertions/suppressions déséquilibrent l'arbre. Il faut donc reéquilibrer celui-ci afin de conserver une hauteur logarithmique.

On appelle **facteur d'équilibre** $\rho(A)$
d'un arbre $A =< r, A_g, A_d >$
strictement binaire le rapport
$\frac{\#\text{feuilles}(A_g)}{\#\text{feuilles}(A)}$

Arbres $BB[\alpha]$

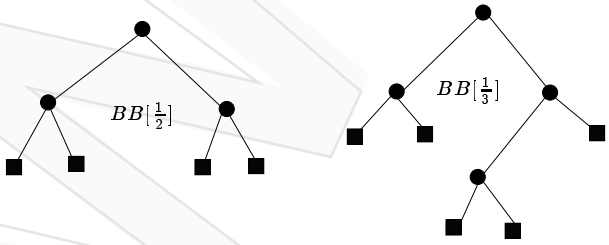
Soit $0 < \alpha \leq \frac{1}{2}$. Un arbre binaire est $BB[\alpha]$ si $\alpha \leq \rho(A) \leq 1 - \alpha$ et les sous-arbres gauches et droits sont $BB[\alpha]$.

$\rightarrow$Arbre d'équilibre α-borné.

Soit A un $BB[\alpha]$ à f feuilles et h la hauteur d'une feuille de A. On a:

$$\frac{\log_2 f}{\log_2 \frac{1}{\alpha}} \leq h \leq \frac{\log_2 f}{\log_2 \frac{1}{1-\alpha}}$$

Exercice 4 *Démontrer par récurrence le lemme ci-dessus*



Équilibrage des arbres $BB[\alpha]$

→Maintenir la propriété des arbres $BB[\alpha]$ sous l'insertion/suppression d'un élément.

→Opérations en temps constant $O(1)$ pour la réorganisation de A. 4 opérations de base: des **rotations**.

Rotation. $\text{Rotation}(c_1[A_1, c_2[A_2, A_3]) = c_2[c_1[A_1, A_2], A_3]$

Rotation Miroir.

$$\text{RotationMiroir}(c_1[c_2[A_1, A_2], A_3]) = c_2[A_1, c_1[A_2, A_3]]$$

Double Rotation.

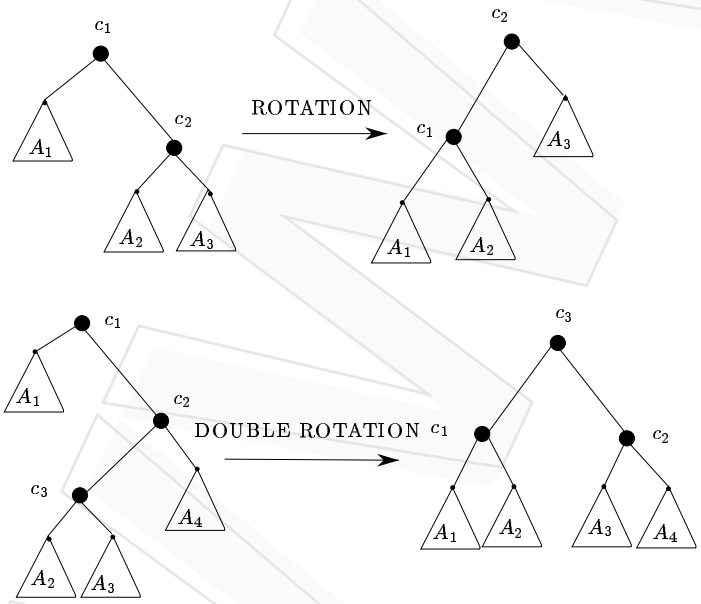
$$\text{DoubleRotation}(c_1[A_1, c_2[c_3[A_2, A_3], A_4]]) = c_3[c_1[A_1, A_2], c_2[A_3, A_4]]$$

Double Rotation Miroir.

$$\text{DoubleRotationMiroir}(c_1[c_2[A_1, c_3[A_2, A_3]], A_4]) = c_3[c_2[A_1, A_2], c_1[A_3, A_4]]$$

→Conserve l'état ordonné de l'arbre (étiquettes c_i).

Illustration des rotations



Algorithme de mise à jour

Soit $\frac{2}{11} \leq \alpha \leq 1 - \frac{\sqrt{2}}{2}$ et $A = c[A_g, A_d]$. L'arbre A' obtenu par insertion/suppression d'un élément de A_g/A_d est tel que:

- Si $\rho(A'_d) < \alpha$ alors
  - si $\rho(A') \leq \frac{1-2\alpha}{1-\alpha}$ une Rotation(A') équilibre A'
  - sinon une DoubleRotation(A') l'équilibre.
- Si $\rho(A') > 1 - \alpha$ alors
  - si $1-\rho(A'_g) \leq \frac{1-2\alpha}{1-\alpha}$ une RotationMiroir(A') l'équilibre
  - sinon on fait une DoubleRotationMiroir(A')

Exercice 5 *Donner les algorithmes de mise à jour lors d'une insertion et suppression d'une feuille (on pourra disposer à chaque nœud d'un compteur du nombre de feuilles enracinées à ce nœud). Quelles sont les complexités de recherche, insertion et suppression d'un élément?*

Hachage

On implante une table grâce à une table de **hachage** qui se révéle utile en pratique.

→Complexité en temps constant $O(1)$ en moyenne pour les opérations de recherche, insertion et suppression.
Jusqu'à présent, la valeur de $c \in \mathcal{C}$ était indépendante de $e \in \mathcal{E}$. Dans les techniques de hachage, on **calcule** la clef grâce à l'élément. La clef sert d'indice pour stocker l'élément dans un tableau $T \in \mathcal{T}$.

On appelle **fonction de hachage** la méthode de calcul de la clef à partir d'un élément. Fonction de $\mathcal{E}$ dans $\mathcal{I}$, l'ensemble des indices du tableau.

On peut prendre par exemple la longueur d'un mot, la somme de ses caractères ASCII, le XOR des caractères, etc...

Hachage (fonctions)

Soit $\mathcal{I} = \{1, ..., M\} = [M]$ l'ensemble des indices de T. Alors la fonction de hachage doit être dans $[M]$. On aimerait une bijection entre l'ensemble $\mathcal{E}$ et le tableau T. Ce qui n'est pas possible dès que $|\mathcal{E}| > M$. Si deux éléments différents ont la même clef, on dit qu'il y a **collision**. Si $n = |\mathcal{E}|$ alors il y a M^n fonctions possibles parmi lesquelles $\mathbf{A}_M^n$ sont sans collisions.

→Régler les conflits de collisions.
→Fonctions de hachage bien faites.

Essentiellement, deux techniques de base pour construire des fonctions de hachage:

- Méthode utilisant la division
- Méthode utilisant la multiplication

La méthode de division

On utilise la division entière (ou l'arithmétique modulo M) de façon à rester dans $\mathcal{I} = [M]$.

$$\boxed{\textsc{Clef}(e) = v(e) \bmod M + 1}$$

On choisit M premier afin d'éviter quelques collisions? Expliquer pourquoi?

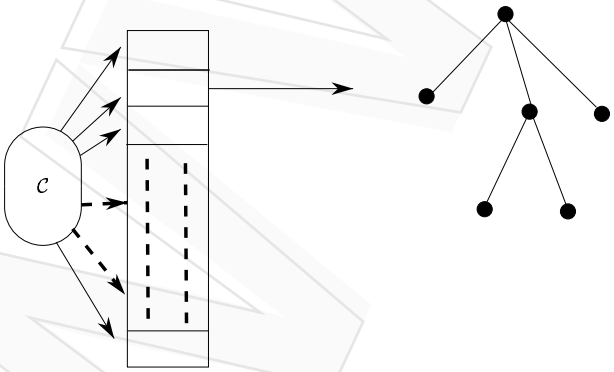
→Calculer les clefs pour les mots suivants: OF, THE, FOR, AND, THAT avec $M = 41$.
→Il reste encore des collisions.

La méthode de multiplication

On multiplie la valeur d'un élément par un nombre réel $\alpha \in (0,1)$. On prend la partie décimale du résultat et on le multiplie par M.

$$\boxed{\textsc{Clef}(e) = \lfloor \text{Décimal}(v(e) \times \alpha) \times M \rfloor + 1}$$

→La **taille** du tableau n'a pas d'importance.
→Choix crucial de la valeur de α.
Des valeurs de α qui répartissent le plus uniformément sont: $\alpha = \phi$ et $\alpha = 1 - \phi$ avec $\phi = \frac{\sqrt{5}-1}{2}$.
→On obtient toujours des collisions: deux méthodes de résolution:



- Résolution directe.
- Résolution indirecte

Résolution indirecte

Si plusieurs éléments ont la même clef alors on regroupe ceux-ci dans une autre structure de données comme par exemple, une liste, un arbre de recherche ordonné, ...
Lors d'une recherche, insertion ou suppression, on se raméne à la manipulation de la structure de données secondaire.
Si la fonction de hachage est uniforme alors la longueur moyene d'une liste est $\frac{n}{M}$ (preuve par les fonctions génératrices)
Soit $p_{n,k}$ la probabilité qu'une liste ait comme longueur k pour n éléments. On a:

$$p_{n,k} = \frac{1}{M} p_{n-1,k-1} + (1 - \frac{1}{M}) p_{n-1,k}$$

La longueur moyenne d'une liste est $\Sigma_{k \geq 0}\, k p_{n,k}$. On utilise la S.G. $G_n(z) = \Sigma_{k \geq 0}\, p_{n,k} z^k$.
Soit ρ le taux de remplissage. Pour une fonction de hachage uniforme en utilisant la méthode de résolution indirecte, le nombre moyen de comparaisons est de l'ordre de $1 + \frac{\rho}{2}$ pour une recherche fructueuse et de l'ordre de $\rho + e^{-\rho}$ pour une recherche infructueuse.

Résolution directe

On range tous les éléments dans le tableau plutôt que de les chaîner. Pour cela, on calculer la clef CLEF(e). A l'aide de cette première entrée on va créer une suite d'entiers balayant obligatoirement tout le tableau sans répétition. On associe un statut à chaque case du tableau: **libre**, **occupée** ou **supprimée**.

Insertion. On balaye à partir du premier indice les cases de T jusqu'à obtenir une case LIBRE ou SUPPRIMÉE. On fixe alors le statut à OCCUPÉE.

Recherche. comme pour l'insertion. On s'arrête si on tombe sur une case LIBRE.

Suppression. On fait d'abord une recherche puis en cas de succés on marque la case à SUPPRIMÉE (et non pas LIBRE).

Exercice 6 *Donner les algorithmes de recherche, insertion et suppression.*

Calcul de la suite des places possibles: méthode linéraire et méthode de Bell&Kaman.

Résolution directe linéaire

Soit p_i la i^e place de la suite, la méthode linéaire est simplement définie par

$$p_i = \begin{cases} \textsc{Clef}(e) & \text{si } i = 1 \\ p_{i-1} \bmod M + 1 & \text{sinon.} \end{cases}$$

$\rightarrow$apparition de **groupement primaire**: tendance à remplir le tableau par paquets.
On peut montrer que le nombre moyen de comparaison lors d'une recherche fructueuse est $\frac{1}{2}(1 + (\frac{1}{1-\rho})^2)$ et de l'ordre de $\frac{1}{2}(1 + (\frac{1}{1-\rho})^2)$ pour une recherche infructueuse avec ρ le quotient du nombre de cases occupées ou supprimées avec M.

Résolution directe de Bell&Kaman

→évite les groupements primaires. On utilise deux fonctions de hachage h_1 et h_2. La fonction h_2 doit fournir un entier dans $[0, M-1]$ premier avec M. On pose:

$$p_1 = \textsc{Clef}(e)$$
$$p_i = ((p_{i-1} - h_2(e)) \bmod M) + 1$$

Si M est choisi tel que M et $M-2$ soient premiers (1021 et 1019) alors on pourra prendre $p_1 = v(e) \bmod M + 1$ et $h_2(e) = (v(e) \bmod (M-2)) + 1$.
On peut montrer que la suite $\{p_i\}_i$ balaie bien $\mathcal{I}$ sans répétition.
Le nombre moyen de comparaisons pour une recherche fructueuse avec une fonction de hachage uniforme est de l'ordre de $-\frac{1}{\rho}\log(1-\rho)$ et de l'ordre $\frac{1}{1-\rho}$ pour une recherche infructueuse.