921 : Algorithmique de recherche et structures de données associées.

26/05 2015

À partir d'un ensemble de données de même type, on va chercher des structures de données adaptées pour effectuer les opérations :

- insérer. - supprimer. - rechercher.

I Méthodes simples.

A) Recherche dans une liste non triée.

Pour rechercher un élément dans la liste : on le compare avec le premier élément ; si c'est négatif, on passe au suivant... Par exemple, la suppression infructueuse est en $O(n)$.

B) Recherche dans une liste triée.

On a une relation d'ordre sur les éléments. L'insertion d'un élément en fin de liste se fait en $O(n)$ comparaisons. Performance moyenne : $\Theta(n)$.

C) Recherche dichotomique.

Recherche d'une occurence d'un élément quelconque se fait $\Theta(\log n)$ comparaisons. On utilise du récursif.

En bilan : un algorithme fonctionnant par comparaisons ne peut faire mieux que $O(\log_2 n)$ (arbre de décision).

[FRO]

II Structure d'arbre.

A) Arbres de recherche.

[BBC p 147]

Déf 1 : Un univers U est un ensemble totalement ordonné. Pour éviter de déplacer les données $x = \{x, y, z\}$ à trier, on associera à chaque donnée une clé, qui est un élément de l'univers. Réciproquement, une clé nous permet d'accéder à une unique donnée : de $c(x)$ on trouve x.

Déf 2 : Un arbre binaire de recherche est un arbre muni d'une fonction clé : $\forall x$ sommet, $\forall y \in A_g(x)$ (sous-arbre gauche du noeud x) $\forall z \in A_d(x)$ (sous-arbre droit du noeud x), $c(y) < c(x) < c(z)$.

Déf 3 : Une arborescence ordonnée de recherche est un arbre tel qu'en chaque noeud x, à $d(x)$ fils et muni de $d(x)-1$ clés $c_1(x), \dots, c_{d(x)-1}(x)$, en posant $A_1(x), \dots, A_{d(x)}(x)$ les sous-arbres de x, pour tout y_i sommet de $A_i(x)$, on a : $c(y_1) < c_1(x) < c(y_2) < \dots < c_{d(x)-1}(x) < c(y_{d(x)})$.

Déf 4 : Un arbre de recherche est un ABR ou un AOR.

Rem 5 : Les structures sont dynamiques, elles sont amenées à changer au cours du temps (exemple : dictionnaire).

[BBC p 151]

Déf 6 : On effectue les opérations suivantes sur les arbres binaires : rotations gauche, droite, (g ou d) g - d ou d - g. (voir annexes 1 - 2).

Prop 7 : Si A est un ABR, alors les rotations sur A préservent cette structure. De plus, elles se réalisent en temps constant.

B) Arbres AVL

← préciser l'utilité

Déf 8 : On note δ la fonction "différence de hauteurs" définie sur les ABR : $\delta(A) = \text{hauteur}(A_g) - \text{hauteur}(A_d)$.

Les AVL sont les ABR respectant la condition : $\delta(A) \in \{-1; 0; 1\}$.

Prop 9 : Soit A un AVL ayant n sommets et de hauteur h. Alors :

$$\log_2(1+n) \leq 1+h \leq 1{,}44 \log_2(2+n).$$

La borne minimale est essentiellement atteinte par les arbres de Fibonacci : $\Phi_0 = \emptyset$, $\Phi_1 = \cdot$, $\Phi_{n+2} = \Phi_{n+1} \overset{\circ}{\wedge} \Phi_n$

Prop 10 : Les opérations insertion, suppression et recherche se font en temps $O(\ln n)$. [DEV]

C) Arbres a-b.

[BBC p 160]

Déf 11 : Soient a et b deux entiers avec $a \geq 2$ et $b \geq 2a - 1$. Un arbre a-b est un arbre A vérifiant :

i) toutes les feuilles ont la même profondeur.

ii) la racine a au moins 2 et au plus b fils.

iii) les autres noeuds ont au moins a et au plus b fils.

Soit $d(x)$ le nombre de fils de x, et $A_i(x)$ le $i$ème sous-arbre de x.

Déf 12 : Soit S un ensemble de clés. A est un arbre a-b pour S si les éléments de S sont rangés aux feuilles de A en ordre croissant de la gauche vers la droite, et si pour chaque noeud x à $d(x)-1$ clés $k_1 < \dots < k_{d(x)-1}$ (les balises de x) vérifient :

pour toute clé c_i de $A_i(x)$, on a :

$$c_1 \leq k_1 < c_2 \leq k_2 < \dots \leq k_{d(x)-1} < c_{d(x)}.$$

Ex 13 : voir annexe 3.

Propo 14 : La recherche dans un arbre a-b se fait en temps $O(\log n)$.

Principe 15 de l'insertion : ① On descend dans l'arbre à partir de la racine, et on ajoute au père une balise (cf annexe 4).
② On vérifie que le père a entre a et b fils. Si ce n'est pas le cas, on réalise un éclatement (cf annexe 5), et si nécessaire, on continue jusqu'à la racine.

Propo 16 : L'insertion dans un arbre a-b se fait en temps $O(\log n)$.

Principe 17 de la suppression : on recherche la clé, on la supprime, on supprime une balise (si nécessaire, réaliser une fusion).

Propo 18 : La suppression dans un arbre a-b se fait en temps $O(\log n)$.

D) Arbres bicolores (rouge-noir)

Déf 19 : Un arbre bicolore (A,c) où A est un arbre complet binaire et c une coloration des sommets dans {rouge, noir} est un arbre vérifiant :
a) toutes les feuilles sont noires ; b) la racine est noire ;
c) le père d'un sommet rouge est noir ;
d) les chemins d'un même sommet à une feuille ont le même nombre de sommets noirs.

Un arbre bicolore pour un ensemble ordonné S est un arbre bicolore qui est un ABR pour S, où les éléments de S ne sont que sur les noeuds.

Ex 20 : voir annexe 6 :

[BRC]

Déf 21 : Une fonction rang rg définie sur un ABR complet A est une application de S (sommets de A) dans $\mathbb{N}$ vérifiant :
i) $\forall$ feuille x, $rg(x)=0$ et si x a un père, $rg(p(x))=1$.
ii) $\forall$ sommet x, $rg(x) \leq rg(p(x)) \leq rg(x)+1$.
iii) $\forall$ sommet x, $rg(x) < rg(p(p(x)))$.

Propo 22 : si (A,c) est un arbre bicolore, alors il existe une fonction rang sur A. Réciproquement, l'existence d'une fonction rang indique qu'il existe une coloration pour A tel qu'il soit bicolore.

Propo 23 : les opérations d'insertion et de suppression se font en temps $O(\log n)$.

E) Arbres binaires de recherche optimaux.

[COR p 363]

Déf 24 : Soient $k_1, \dots, k_n$ n clés triées par ordre croissant. La probabilité qu'une clé k_i soit demandée est p_i. On ajoute $n+1$ clés factices, $d_0, \dots, d_n$, où d_i représente les mots recherchés entre k_i et k_{i+1}. Soit q_i la probabilité qu'on recherche d_i. Un ABRO est un arbre binaire minimisant :
$$\mathbb{E}[\text{coût de recherche dans } A] = 1 + \sum_{i=1}^{n} prof_A(k_i)\, p_i + \sum_{i=0}^{n} prof_A(d_i) \cdot q_i$$

Rem 25 : Cette structure est statique, l'insertion et la suppression d'un élément entraînent une reconstruction totale de l'arbre.

Ex 26 : cf annexe 7.

III Application à l'algorithmique du texte.

A) Plus longue sous-séquence commune.

[COR p 362]

Exemple : Comparaison de deux brins d'ADN.

Déf 27 : $Z = \langle z_1, \dots, z_k \rangle$ est une sous séquence de $X = \langle x_1, \dots, x_m \rangle$, s'il existe $\varphi : [\![1;k]\!] \to [\![1;m]\!]$ strictement croissante telle que $\forall i \in [\![1;k]\!]$, $z_i = x_{\varphi(i)}$

On recherche la plus longue sous-séquence commune (PLSC) à $X = \langle x_1, \dots, x_m \rangle$ et à $Y = \langle y_1, \dots, y_n \rangle$.

Propo 28 : Soit Z une plus longue sous-séquence commune à X et à Y.
Si $z_k = x_m = y_n$, alors Z_{k-1} est une PLSC à X_{m-1} et Y_{n-1}.
Si $z_k \neq x_m$ alors Z_k est une PLSC à X_{m-1} et Y_n.
Si $z_k \neq y_n$ alors Z_k est une PLSC à X_m et Y_{n-1}.

Résolution par programmation dynamique ; récurrence sur la taille de X et sur la taille de Y.

B) Distance d'édition.

Ex : Comparer les mots REMUER et RUMEUR.

On veut minimiser la distance d'édition avec trois possibilités d'action :

- aligner une lettre de X et une de Y.
- sauter une lettre de X.
- sauter une lettre de Y.

O(nm)

ex :

R E M U - E - R
R - - U M E U R

distance est de 4.

c) Tri des suffixes.

Pour rechercher un mot dans un texte : création de la table des suffixes. Un mot qui sera défini par une suite de lettres $y[0], \ldots, y[n-1]$, et on veut trier les suffixes $y[0,..,n-1]$, $y[1,..,n-1]$, .., $y[n-1]$.

On va chercher une permutation p telle que $y[p(0),..,n-1] < y[p(1),..,n-1] < \ldots$

Déf 29 : Soit $k>0$. Pour $u \in A^*$, le début d'ordre k du mot u :

$$\mathrm{prem}_k(u) = \begin{cases} u & \text{si } |u| \leq k \\ u[0,..,k-1] & \text{sinon.} \end{cases}$$

On note $R_k[i]$ le rang de de $\mathrm{prem}_k(y[i,...,n-1])$ dans l'ordre croissant des $(R_k[i])_{i \leq n-1}$. Pour tout k, on a la relation d'équivalence :

$i \equiv_k j$ ssi $R_k[i] = R_k[j]$.

Pour $i \geq n$, on pose $R_k[i] = -1$.

Lemme 30 (de dédoublement) : $R_{2k}[i]$ est le rang du couple $(R_k[i], R_k[i+k])$ dans la liste croissante de ces couples.

Th 31 : L'algorithme TRI-SUFFIXES appliqué à y de longueur n crée la table des suffixes en temps $O(n \log n)$ |DVP|

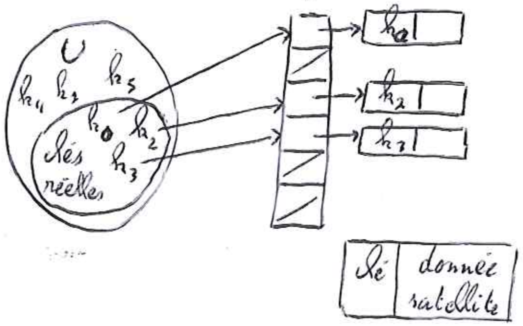
[CRO]

IV Tables de hachage.

A) Tables à adressage direct.

[COR]

Déf 32 : Une table à adressage direct est un tableau $T[0,..,m-1]$ dans lequel chaque position (ou alvéole) correspond à une clé de l'univers U.



Prop 33 : Chacune des opérations : recherche, insertion et suppression se fait en temps $O(1)$.

B) Tables de hachage.

Problème : Si $|U|$ grand, on ne peut pas toujours gérer T aussi grand.

Déf 34 : Une fonction de hachage est une fonction $h : U \to [0, m-1]$, où $|m| \ll |U|$; le tableau $T[0,..,m-1]$ est la table de hachage. On dit qu'un élément de clé k est haché dans l'alvéole $h(k)$.

Inconvénient : la non injectivité de h peut envoyer deux clés dans la même alvéole : on a un phénomène de collision ! Pour le résoudre, on utilise le chaînage (cf annexe 8).

Déf 35 : On définit $\alpha = \frac{n}{m}$ le facteur de remplissage, c'est-à-dire pour n éléments stockés dans un tableau de taille m.

Th 36 : Dans une table de hachage dont la résolution se fait par chaînage, le coût d'une recherche réussie ou d'une recherche infructueuse est $\Theta(1+\alpha)$, dans le cas d'un hachage uniforme simple (chaque élément a la même chance d'être haché vers l'une quelconque des alvéoles).

C) Des exemples de fonctions de hachage.

Déf 37 : Méthode la division (les clés sont des entiers).

$h : k \mapsto k \ [\mathrm{mod}\ m]$.

Si les données sont stockées en bit, on évite que m soit $m = 2^p$.

Déf 38 : Méthode de la multiplication. Soit $0 < A < 1$.

$h : k \mapsto m(kA - \lfloor kA \rfloor) \ [\mathrm{mod}\ 1]$.

Déf 39 : Si la fonction de hachage est donnée, un ennemi peut choisir les clés telles qu'elles soient stockées dans la même cellule. On sélectionne donc aléatoirement une fonction de hachage : c'est le hachage universel.

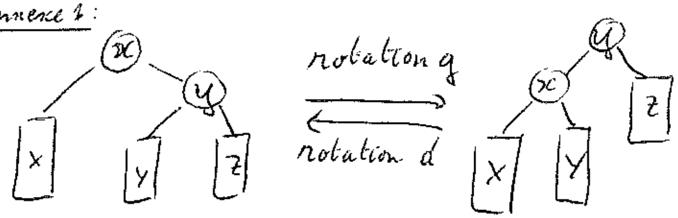
Déf 40 : Une collection de fonctions de hachage est universelle si :

$$\#\{h \in \mathcal{H} ; h(k) = h(l)\} \leq \frac{|\mathcal{H}|}{m}, \quad \forall k, l \leq m.$$

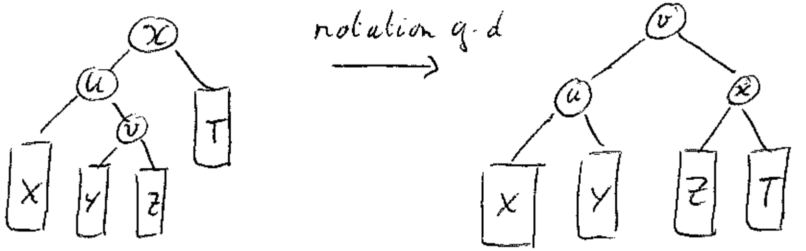
Th : Soit n_i la longueur de la liste $T[i]$. Pour une fonction de hachage choisie aléatoirement dans une collection universelle, et k la clé recherchée : si k est dans la table : $\mathbb{E}[n_{h(k)}] \leq \alpha = \frac{n}{m}$; sinon : $\mathbb{E}[n_{h(k)}] \leq \alpha + 1$.

+ recherche de clé minimal (TAS-MIN)

Annexe 1 :

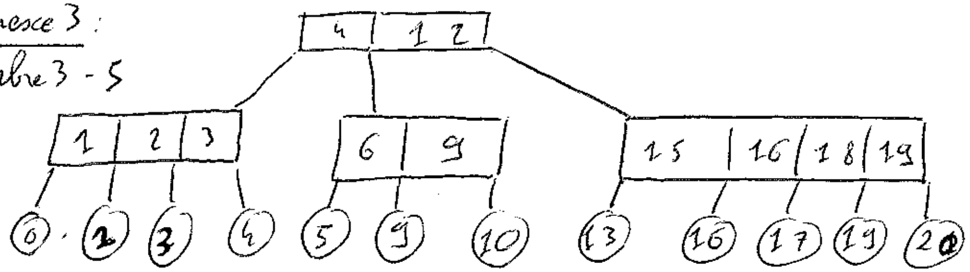


Annexe 2 :



Annexe 3 :

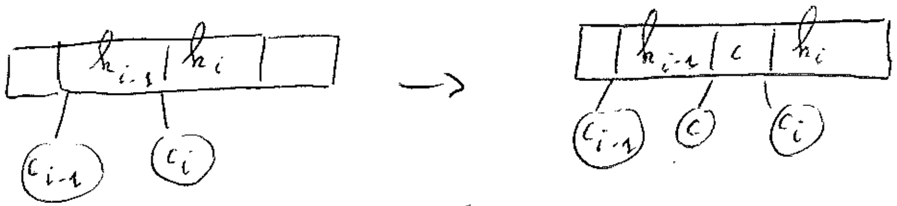
arbre 3 - 5



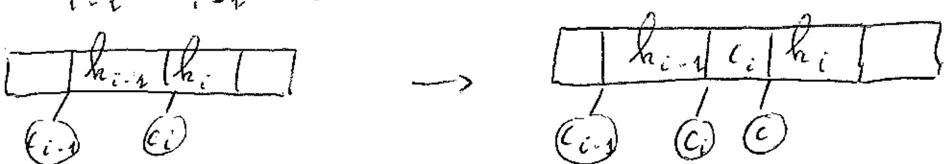
$S = \{10 ; 4 ; 3, 16 ; 19 ; 5 ; 20 ; 13 ; 0 ; 17 ; 2 ; 9\}$

Annexe 4 :

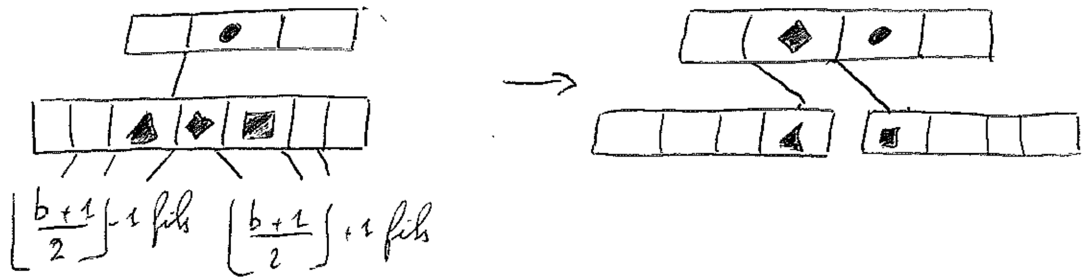
Si $c_{i-1} \leq h_{i-1} < c < c_i \leq h_i$:



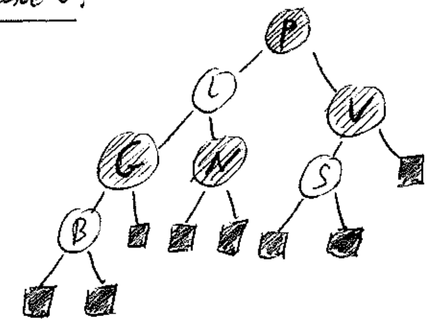
Si $c_{i-1} \leq h_{i-1} < c_i < c \leq h_i$:



Annexe 5 :

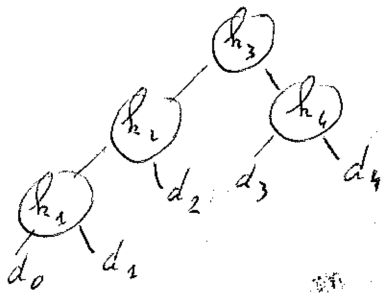


Annexe 6 :



Annexe 7 :

i	0	1	2	3	4
p_i		0,05	0,05	0,05	0,5
q_i	0,05	0,05	0,05	0,05	0,15



Annexe 8 :