9.2.1 Algorithmes de recherche et structures de données associées.

On cherche à implémenter la structure de données abstraite Dictionnaire, qui représente un ensemble de valeurs, chacune étant associée à une clé dans un ensemble totalement ordonné. On s'intéresse en particulier aux opérations :

Rechercher : clé × dictionnaire → valeur
Insérer : valeur × dictionnaire → dictionnaire
Supprimer : clé × dictionnaire → dictionnaire

ainsi qu'à des opérations de recherche d'une valeur en fonction du rang de sa clé.

→ Cette structure de données est d'une grande importance pour le stockage d'informations, et l'optimisation des opérations ci-dessus a donc beaucoup de conséquences pratiques : gestion d'une base de données, stockage d'informations sur un disque dur...

→ On suppose dans la suite que les clés sont distinctes.

I] Recherche dans une liste ou un tableau.

1] Recherche par clés :

→ Dans une liste non-triée :

On recherche une clé en parcourant la liste : complexité $O(n)$ au pire et en moyenne ; de même pour la suppression ; insertion en $O(1)$.

↳ Efficace pour peu de données.

→ Dans un tableau trié :

Recherche dichotomique : Si T est trié, on peut faire une méthode diviser-pour-régner :
- Comparer la clé cherchée avec celle de $T.[\lfloor \frac{|T|}{2} \rfloor]$
- Si elle est égale, renvoyer $T.[\lfloor \frac{|T|}{2} \rfloor]$
- Si elle est inférieure (resp. supérieure), rechercher l'élément dans $T.[1..\lfloor \frac{|T|}{2} \rfloor]$ (resp. $T.[\lfloor \frac{|T|}{2} \rfloor + 1 .. |T|]$)

Recherche en temps $O(\log_2 n)$, Insérer et Supprimer nécessitent $O(\log_2 n)$ opérations pour déterminer l'emplacement, puis le coût d'ajout ou de suppression d'un élément, soit $O(n)$.

↳ Utile pour les tableaux statiques (Rechercher uniquement)

Prop 1 : (Optimalité) Les algorithmes de recherche par comparaison pour les tableaux triés ont une complexité en temps $\Omega(\log_2 n)$ (en moyenne et au pire). 9.3.4

→ Optimisation de Insérer / Supprimer ?

Si on a beaucoup de telles opérations à faire, la complexité $O(n)$ peut être un problème. Nous verrons en II] et III] comment améliorer ces performances.

2] Recherche par rangs : [Cor] chap 9

→ Recherche du min, du max, de la médiane, du k^e plus grand

Faire ce type de recherche dans un tableau trié est constant. Pour une liste non triée, on a donc un algorithme en $O(n \log n)$ pour faire ces recherches. Ce n'est pas optimal.

- Rechercher le min (ou le max) : 9.1
  Comparer un minimum courant à toutes les clés.
  ↳ temps $O(n)$ optimal
- Rechercher le k^e plus grand à l'aide d'un diviser pour régner 9.2
  Prendre une clé a quelconque, séparer le tableau T en
  $T_+ = \{b \in T,\ b > a\}$ $T_- = \{b \in T,\ b < a\}$.
  Si $|T_-| < k$, $|T_+| > |T| - k$, appeler récursivement sur T_-
  Si $|T_-| > k$, $|T_+| < |T| - k$, ——— T_+
  Sinon, renvoyer a
  ↳ complexité $O(n^2)$ au pire cas, $O(n)$ en moyenne.

→ Recherche du plus petit élément plus grand qu'une clé x. [?]

On peut construire un algorithme similaire à la recherche du min en $O(n)$.

Les algorithmes de recherche par rangs sont facilement implémentables avec des arbres (II), et efficace.

II] Arbres binaires de recherche [Cor] chap 12 [Froi] chap 10 [Sed] chap 14

1] Définition et implémentation

Déf 2 : Un arbre binaire de recherche (ABR) est un arbre binaire étiqueté tel que pour tout nœud v, les éléments du sous arbre gauche (resp. droit) sont inférieurs (resp. supérieurs) à v. (cf Annexe 1)

On représente un dictionnaire par un ABR en étiquetant les sommets par les clés :

Recherche x A : (Complexité $O(\text{haut}(A))$)
- v = racine(A)
- Si $x = v$, terminé.
- Si $x < v$, Recherche x gauche(A)
- Si $x > v$, Recherche x droite(A)

Insérer x A : (Complexité $O(\text{haut}(A))$)
- Si A = ∅, renvoyer x
- Sinon, v = racine(A)
  - Si $x < v$, Insérer x gauche(A)
  - Si $x > v$, Insérer x droite(A)

Supprimer x A : (Compl. $O(\text{haut}(A))$)
- Rechercher x A
- Si x est une feuille, supprimer x.
- Si x n'a qu'un fils y, remplacer x par y
- Si x a deux fils g et d, remplacer x par le plus grand élément du sous-arbre g

Notation : $h(A)$ = hauteur, $N(A)$ = nbre de nœuds, $f(A)$ = nbre de feuilles

Il est également aisé de trouver le Successeur ou le Prédécesseur d'un élément, de trier, ou de trouver le $k^{\text{ème}}$ élément dans un ABR.

→ Pire cas : Arbre dégénéré (c'est que n = nbre d'éléments) Opérations en $O(n)$.

Se produit si on construit l'arbre à partir d'un tableau trié.

→ Meilleur cas : Arbre équilibré : Opérations en $O(\log_2 n)$.

[Froi] 10.4.2

Prop 3 : Les opérations Recherche, Insérer, Supprimer se font en moyenne en temps $O(\log_2 n)$ dans un ABR.

Si le pire cas est inévitable, on peut chercher à maintenir l'arbre équilibré.

2] Équilibrage et AVL [Froi] Partie 3, chap 11

→ 11.1

→ Rotations

Pour équilibrer un ABR, on va utiliser des opérations de rotation (représentées en Annexe 2) sur les arbres : une réassignation d'un nœud et d'un de ses fils (et éventuellement petit-fils) et de leurs sous-arbres afin de modifier les hauteurs relatives des sous-arbres → se fait en temps constant.

→ AVL

11.2

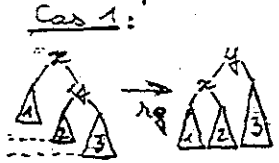
Déf 4 : Un AVL est un ABR tel que pour tout nœud, la différence de hauteur entre ses sous-arbres gauche et droit est d'au plus 1.

Pour implémenter un AVL, on stocke dans un nœud les hauteurs de ses sous-arbres gauche et droit.

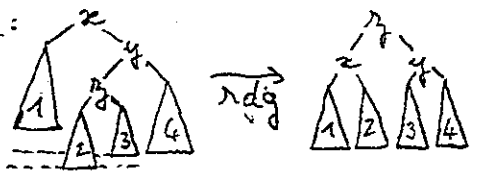
Prop 5 : Si A est un AVL à n nœuds, alors $\text{haut}(A) = O(\log_2(n))$ 11.2.1

La recherche, insertion et suppression fonctionnent comme pour un ABR, mais pour insérer et supprimer, il faut remonter dans l'arbre à partir de l'élément modifié et rééquilibrer le sous-arbre dans lequel on se trouve :

Cas 1 :



Cas 2 :



Prop 6 : Les opérations Recherche, Insérer, Supprimer se font au pire en temps $O(\log_2 n)$ dans un AVL.

Remarque 7 : Il existe d'autres types d'arbres permettant d'équilibrer les ABR : arbres rouge-noir, B-arbres (non binaire).

3] Applications

→ ABR optimaux et compression [Cor] 15.5, [Knu]

Si p_i est la probabilité que l'élément i soit recherché, et q_j $(0 \le j \le n)$ celle qu'un élément $j < x < j+1$ soit recherché,

on cherche à construire un arbre minimisant le temps moyen d'une recherche ⇝ Ceci peut se faire en $O(n^3)$.
↳ Dans le cas où les p_i sont nuls, ceci permet de réaliser le codage de Huffman pour compresser un texte.

[PS] 7.2 (p281)

→ Intersections de segments dans le plan : DEV

Etant donnés N segments dans le plan, on cherche à lister leurs points d'intersection : Utilisation d'ABR pour représenter la structure du problème ⇝ si I = nb d'intersections : $O((I+N)\log_2 N)$.

III Tables de Hachage [Cor; Ch. 11]

Dans cette partie, on suppose les clés dans $[\![1, m]\!]$.

1 Un tableau pour les rechercher toutes

→ Adressage direct

On déclare un tableau T de taille m, chaque valeur sera envoyée dans la case de sa clé.
La recherche, l'insertion, la suppression se font alors en $O(1)$
↳ si le nombre de valeurs est petit devant m, on utilise inutilement beaucoup trop de mémoire...

→ Hachage

On choisit $a \in [\![1, m]\!]$ et $h : [\![1, m]\!] \to [\![1, a]\!]$ fct de hachage.
Une valeur de clé k sera envoyée sur l'alvéole $h(k)$.
Si les valeurs sont envoyées sur des cases différentes, les coûts de recherche, insertion et suppression sont toujours $O(1)$
↳ si $a < m$, il y a forcément des clés k, l telles que $h(k) = h(l)$.

Def 8 : On dit qu'il y a collision si deux valeurs doivent être envoyées sur la même case.

→ Gérer les collisions

- adressage : si une alvéole est déjà remplie à l'insertion, on trouve une alvéole vide algorithmiquement.
  recherche/suppression : si $T(h(\text{clé})) \neq$ valeur, on suit l'algorithme d'insertion.
- chaînage : (utilisé pour la suite) : utiliser des listes (cf Annexe 3)

2 Faire un bon hachage

But : minimiser/empêcher les collisions

Def 9 : Un hachage $h : [\![1, m]\!] \to [\![1, a]\!]$ est dit uniforme si pour chaque valeur de clé k tirée, et pour tout $i \in [\![1, a]\!]$, $\mathbb{P}(h(k) = i) = \frac{1}{a}$

↳ impossible en pratique sans hypothèse sur les distributions des clés ; on peut cependant bien s'en approcher.

Avec un tel hachage, le coût moyen d'une opération insertion/suppression/recherche pour n valeurs présentes est $O(\frac{n}{a})$
→ Si la fonction est connue d'avance, un utilisateur malveillant peut tenter de saturer une alvéole (ex : attaque d'un site web).

Pour régler ce problème, on prendra à la création du tableau une fonction au hasard parmi une classe universelle :

Def 10 : Une classe universelle $\mathcal{H}$ est un ensemble de fonctions de hachage $[\![1, m]\!] \to [\![1, a]\!]$ tel que pour chaque paire de clés (k, l), $|\{h \in \mathcal{H},\ h(k) = h(l)\}| \leqslant \frac{|\mathcal{H}|}{a}$

ex : soit p premier, $p \geqslant a$; on pose

$$\mathcal{H}_{p,a} = \{k \mapsto ((xk + y) \bmod p) \bmod a \ /\ x, y \in \mathbb{Z}/p\mathbb{Z},\ x \neq 0\}$$

→ Hachage parfait. [Cor ; 11.5]

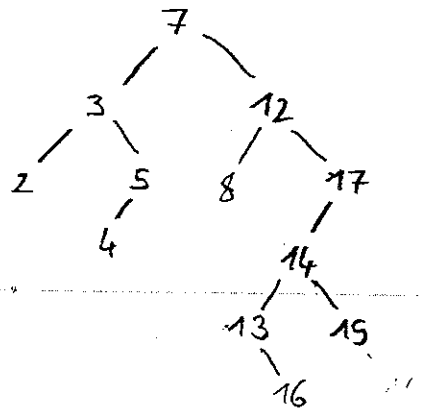
Il est possible, en utilisant un double hachage, de ranger n valeurs dans une structure de taille mémoire $O(n)$ telle que la recherche dans cette structure soit en $O(1)$
La création de cette structure se fait en $O(n)$ DEV

Conclusion

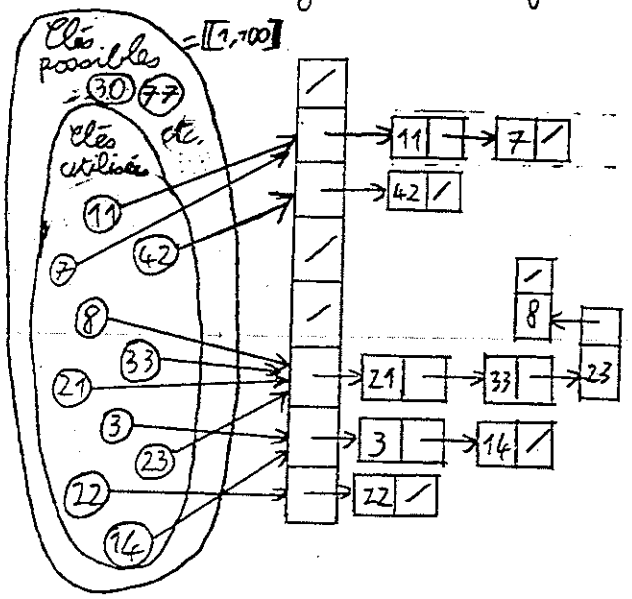
	Recherche	Insertion	Suppression	Recherche par rang
Liste (non triée)	$O(n)$	$O(1)$	$O(n)$	$O(n)$ ~~moyen~~ ; $O(n^2)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
ABR (pire cas)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
ABR (moyenne) = AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Hachage (pire)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Hachage (moy)	$O(1)$	$O(1)$	$O(1)$	$O(n)$

Fraîcheur

Annexe 1 : Arbre binaire



Annexe 3 : Hachage avec chaînage



Références :

[Cor] Cormen et al.

[PS] Preparata-Shamos, Computational Geometry – an Introduction. (Beaulieu)

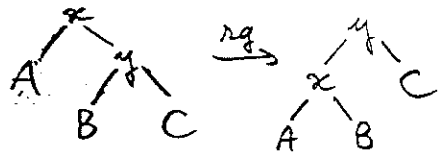
[Sed] Sedgewick, Algorithms (Beaulieu)

[Froi] Froidevaux – Gaudel – Soria
Types de données et algorithmes

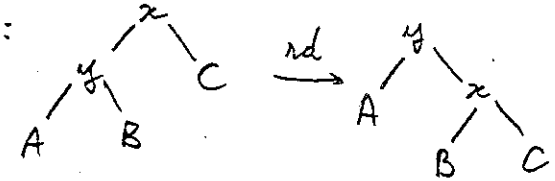
[Knu] Knuth, The art of computer programming, Part 2 : Sorting and Searching

Annexe 2 : Rotation d'arbres

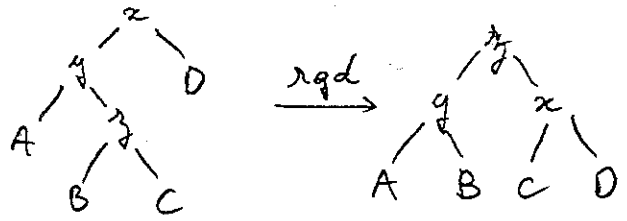
Gauche :



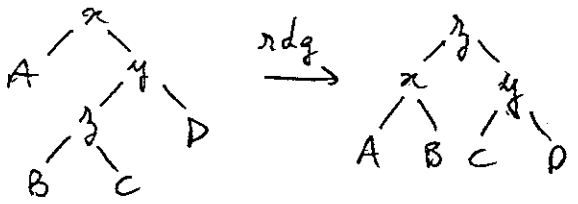
Droite :



Gauche-droite :



Droite-gauche :



Il y a pas cas d'appl. ds le titre. P. ĉ en dépl, bof

Intersection de segments dans le plan.

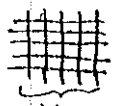
(Algo de Bentley-Ottmann)

[Preparata - Shamos] Computational geometry, an introduction 7.2 p.281 et suivantes

Le problème: + TROP long — Faire avec des segments horizontaux et verticaux ?

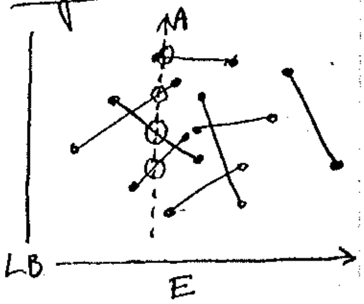
Entrée: N segments du plan, donnés par les coordonnées de leurs extrémités, que l'on suppose toutes distinctes, ainsi que celles de leurs intersections.

Sortie: La liste de leurs points d'intersection.

Algorithme naïf: On cherche les points d'intersection pour chaque paire de segments : $O(N^2)$; c'est même la complexité optimale, puisque sur une grille  $\}N/2$, il y a $O(N^2)$ points d'intersection.

On présente un algorithme en $O((N+I)\log_2 N)$ où I est le nombre de points d'intersection, ce qui en fait un algorithme plus efficace si $I = o(N^2)$.

Algorithme de balayage:



On balaye la figure avec une ligne de balayage verticale LB de gauche à droite.
On maintient un AVL E contenant les événements qui indiquent les points d'importance pour LB : extrémités (données a priori) et intersections (déterminées pendant le balayage).

Pour détecter les intersections, on constate que deux segments qui se croisent doivent être "adjacents" immédiatement avant, c'est-à-dire que leurs intersections avec LB doivent être côte-à-côte si on les trie par ordonnées croissantes.
Or, l'adjacence de deux segments (relativement à LB) n'est modifiée que lorsqu'un segment commence ou termine, ou lorsqu'un point d'intersection est atteint.
On maintient donc un AVL A pour avoir accès aux segments qui intersectent LB à un moment donné, dans l'ordre d'ordonnées croissantes.

* **Initialisation :** On stocke les coordonnées des N segments dans un AVL E en utilisant l'abscisse comme clé. On initialise l'AVL A à $\emptyset$, et on pose L = liste vide (liste des intersections détectées dans un tour de boucle).

* **Boucle :** Tant que E n'est pas vide, soit $e = \min E$, $L = \emptyset$.
  (min renvoie e, et l'efface de E)

① **Si** e est un début de segment s.
- Insérer s A
- s_1 = succ s A, s_2 = prédécesseur s A) nouvelles adjacences
- Si $s_1 \cap s \neq \emptyset$, alors $L \leftarrow (s_1, s)$ Notation bof
  s_2

② **Si** e fin de segment s
- s_1 = succ s A, s_2 = prédécesseur s A) nouvelles adj
- Si $s_1 \cap s_2 \neq \emptyset$ à droite de LB, alors $L \leftarrow (s_1, s_2)$
- Supprimer s A

③ **Si** $e = s_1 \cap s_2$ (avec $s_1 > s_2$ dans A avant e)
- s_3 = succ s_1 A, s_4 = prédécesseur s_2 A) nouv. adj
- Si $s_3 \cap s_2 \neq \emptyset$ alors $L \leftarrow (s_3, s_2)$; $s_4 \cap s_1$ (s_4, s_1)
- Échanger s_1, s_2 dans A.
- Afficher e

④ **Pour** $(s_1, s_2) \in L$
- Si $s_1 \cap s_2 \notin E$, alors Insérer $s_1 \cap s_2$ E
  (Rechercher $s_1 \cap s_2$ E)

Il manque l'affichage.

Correction : Chaque intersection est détectée une fois (parce que toutes les adjacences sont mises à jour) et une seule (grâce à ④).

Complexité : Initialisation : $O(N \log N)$

①, ② et ③ : $O(\log N)$ (opérations dans A)

④ : On a détecté au plus 2 intersections, donc $O(\log(N+K)) = O(\log N)$

Boucle : $O((N+K)\log(N))$ Simplifier pour détailler les structures de données.

$\rightsquigarrow$ $\boxed{O((N+K)\log N)}$

Cormen (2e édition) 11.5

Hachage Parfait

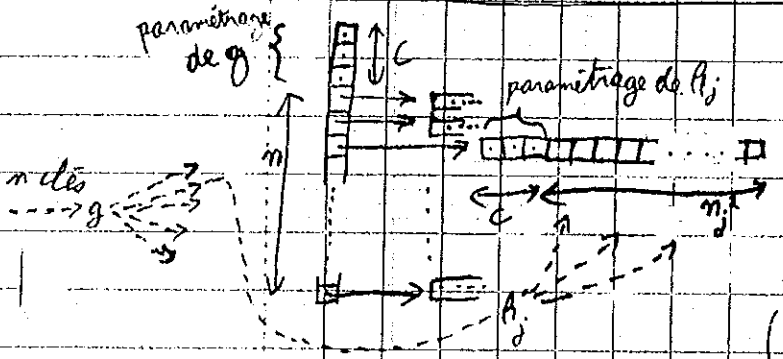
Thm: On peut stocker n clés (statiques) dans une structure de données donnant une recherche en $O(1)$ et de taille mémoire $O(n)$

Prérequis: hachage uniforme classe universelle.

Idée:
- Construire un hachage sans collisions sur une mémoire $O(n^2)$ ($\Rightarrow$ recherche en $O(1)$)
- Faire un "grand hachage" à n alvéoles, puis utiliser le hachage précédent sur chacune des alvéoles.

Lemme 1: Si on stocke n clés dans une table de taille n^2 avec une fonction h de hachage uniforme choisie aléatoirement, alors la probabilité d'avoir une collision est inférieure à $\frac{1}{2}$

Lemme 2: Si on stocke n clés dans une table de taille n^2 avec une fonction g de hachage uniforme (aléatoire), alors, en notant n_j le nombre de clés envoyées sur l'alvéole j ($1 \le j \le n$), on a
$$\mathbb{E}\left[\sum_{j=1}^{n} n_j^2\right] < 2n \quad \text{et donc} \quad \mathbb{P}\left(\sum_{j=1}^{n} n_j^2 \ge 4n\right) < \frac{1}{2}$$



Espace total nécessaire:
$$c + n + nc + \sum_{i=1}^{n} n_j = O(n)$$
↳ quitte à refaire des choix pour g et les h_i

(Le c vient du fait que l'on peut paramétrer un choix dans une classe universelle par trois ou quatre constantes.)

Démo Lemme 1 :

Une ~~Un couple~~ paire de clés étant donné, il y a une probabilité de $\frac{1}{n^2}$ qu'elles soient envoyés sur la même valeur.

↳ avec un hachage uniforme

Sachant qu'il y a $\binom{n}{2}$ clés, on a,

avec X la variable aléatoire comptant le nombre de collisions,

$$\mathbb{E}(X) \underset{(*)}{=} \binom{n}{2} \cdot \frac{1}{n^2} = \frac{1}{2} - \frac{1}{2n} < \frac{1}{2}$$

D'où $\mathbb{P}(X \geq 1) \leq \mathbb{E}(X) < \frac{1}{2}$: on a moins d'une chance sur deux d'avoir une collision. □

Démo Lemme 2 :

On remarque : $\forall a \in \mathbb{N}, \quad a^2 = a + 2\binom{a}{2}$

Alors $\mathbb{E}\left[\sum_{j=1}^{n} n_j^2\right] = \mathbb{E}\left[\sum_{j=1}^{n} n_j\right] + 2\mathbb{E}\left[\sum_{j=1}^{n} \binom{n_j}{2}\right]$ → linéarité de l'espérance

$= n + 2\mathbb{E}\left[\sum_{j=1}^{n} \binom{n_j}{2}\right]$ → $\sum n_j = n$ et n n'est pas aléatoire

de plus, $\sum_{j=1}^{n} \binom{n_j}{2}$ est le nombre total de collisions

Comme g est un hachage uniforme, cette somme est bornée par $\binom{n}{2}\frac{1}{n} = \frac{n-1}{2}$

donc $\mathbb{E}\left[\sum_{j=1}^{n} n_j^2\right] \leq n + 2\frac{n-1}{2} < 2n$

On a alors $\mathbb{P}\left(\sum_{j=1}^{n} n_j^2 \geq 4n\right) \underset{(*)}{\leq} \frac{\mathbb{E}\left[\sum_{j=1}^{n} n_j^2\right]}{4n} < 1/2$ □

(*) On utilise l'inégalité de Markov : $\mathbb{P}(X \geq \varepsilon) \leq \frac{\mathbb{E}(X)}{\varepsilon}$