

Structures de Données & Algorithmes II

-

Tables : des tableaux aux hashmap

Pascal Mérindol (CM, TD, TP)

merindol@unistra.fr

<http://www-r2.u-strasbg.fr/~merindol>

Contenu

- *Les tris (8h)*
- *Arbres & Forêts (12h)*
- *Les graphes (16h)*

- **Les tables (6h)**
 - Quelques définitions
 - Spécification de base
 - Représentations, fonctions et accès
 - Adressage calculé ou associatif
 - Partage de tables, rangements et dispersion
 - Fonction de hachage et gestion des collisions

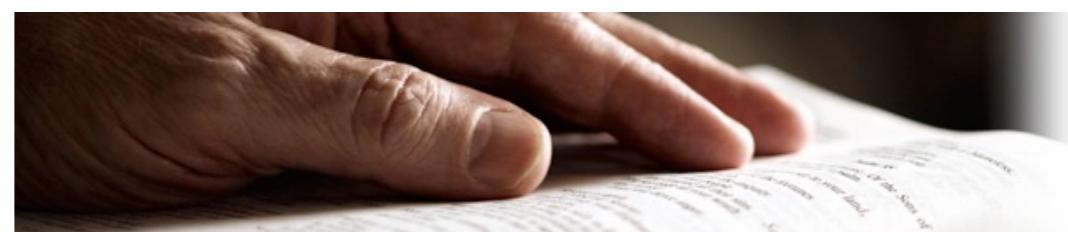
Références

http://en.wikipedia.org/wiki/Hash_table

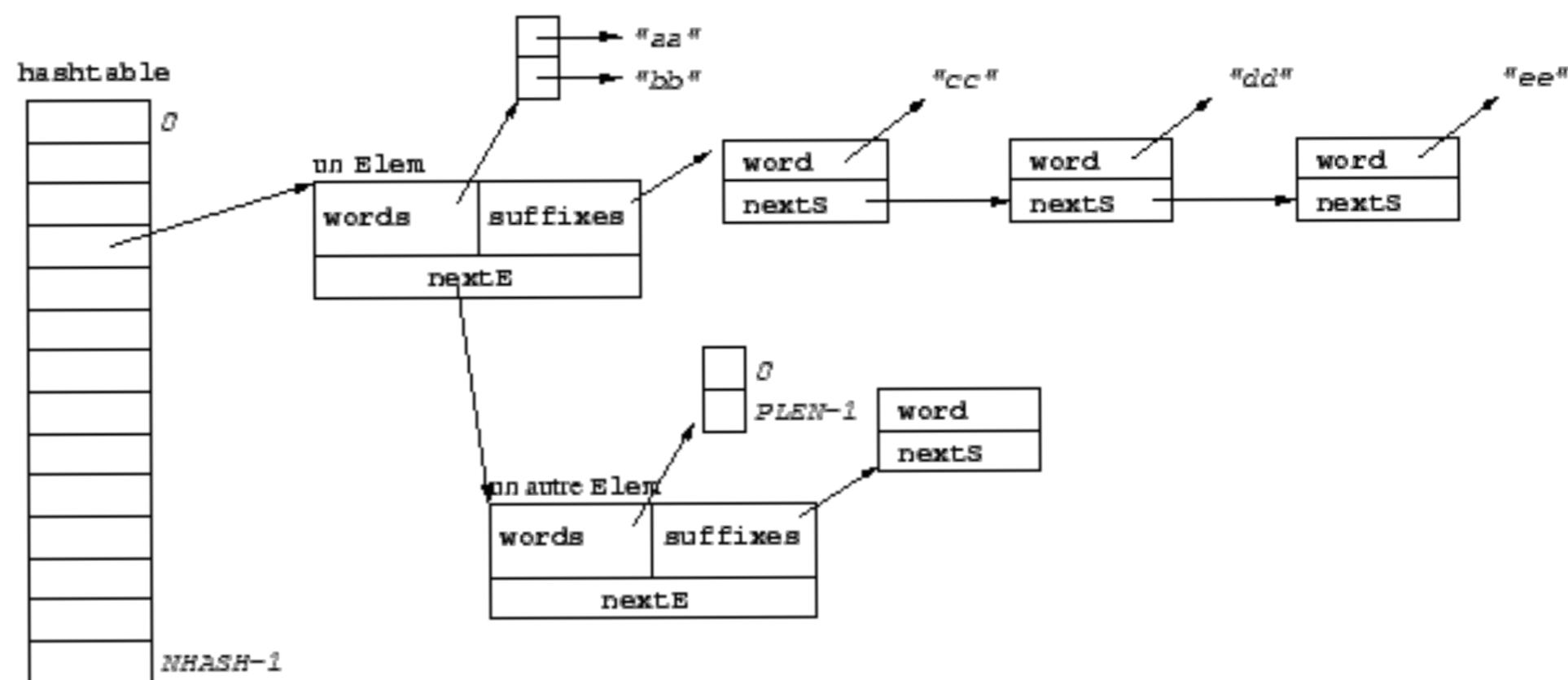
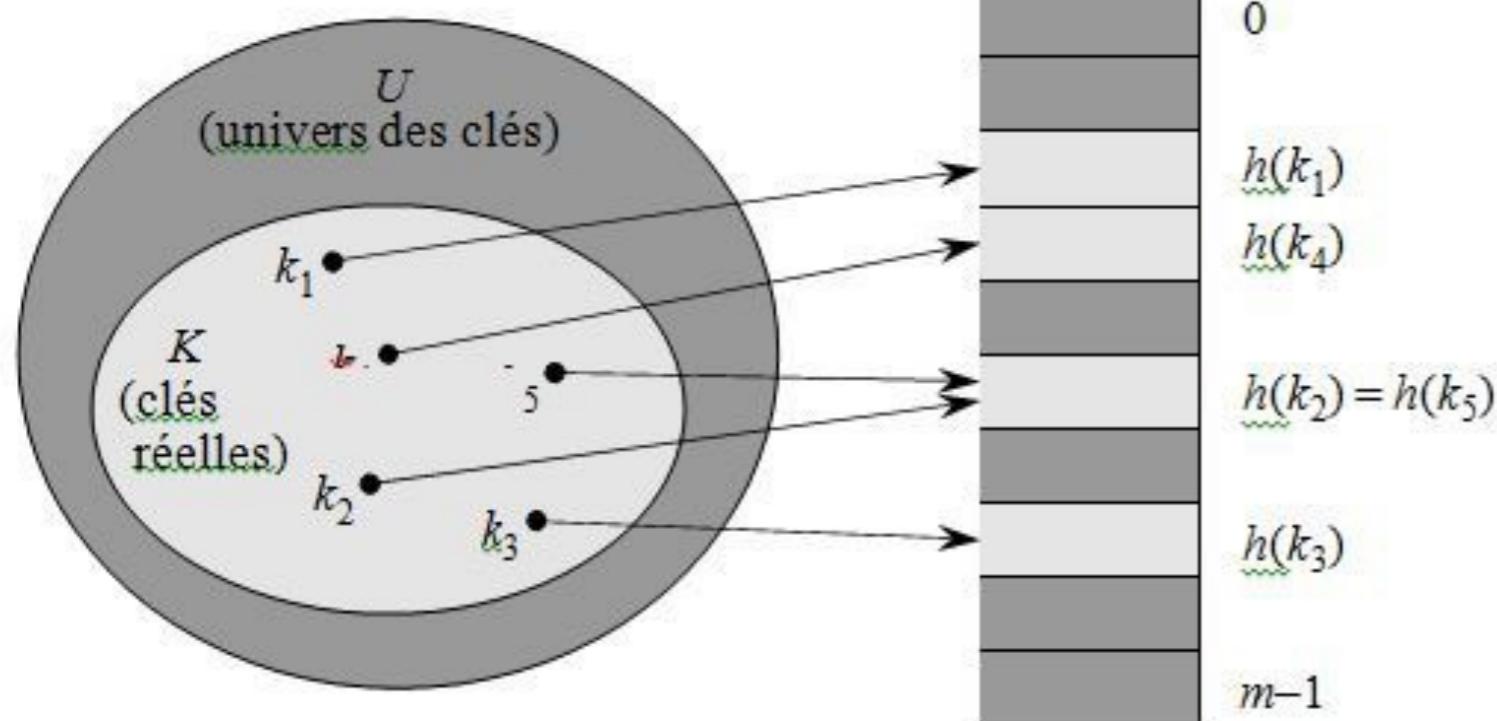
...



- * ***Spécification algébrique, algorithmique et programmation*** : Jean-François Dufourd, Dominique Bechmann, Yves Bertrand.
- * ***The Art of Computer Programming, Volume 3, Sorting and Searching*** : Donald Knuth.
- * ***Introduction to Algorithms*** : Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.



Illustrations



Notions de base

- **Une table (map) t est une fonction partielle d'un ensemble E d'entrées dans un ensemble V de valeurs**
 - On peut noter
 - $\text{dom}(t)$:= domaine de définition de t ou ensemble des entrées effectives de E (sous-ensemble des entrées potentielles)
 - $\text{im}(t)$:= ensemble des images de t
 - les entrées ~ indexes, indicatifs, indices, clés (dépend du contexte)
- **Exemple : table finie ($\text{dom}(t)$ fini)**
 - copo : Chaine -> Chaine // num de département -> num de code postal (de base)
 - on complète t en t' grâce à un w \in V, w \notin im(t)
 - $t'(i) = \text{si } i \in \text{dom}(t) \text{ alors } t(i) \text{ sinon } w$
 - copo ("BAS-RHIN") = 67 mais copo ("ALSACE") = ""

SPÉCIFICATION ALGÉBRIQUE (de BASE0)

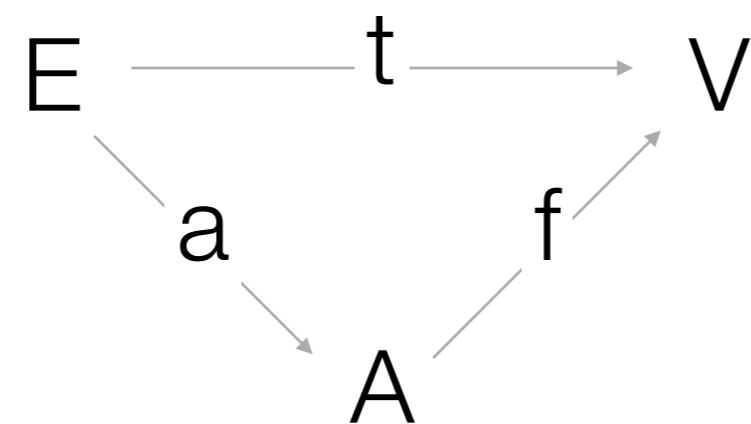
- **spec TABLE0 (EG:= renommé E , EG:= renommé V)**
- **etend BASE**
- **sorte Table**
- **opérations**
 - tabnouv : \rightarrow Table
 - adjt : Table E V \rightarrow Table
 - supt : Table E \rightarrow Table
 - modt : Table E V \rightarrow Table
 - élém : Table E \rightarrow V
 - vide : Table \rightarrow Bool

Préconditions et axiomes

- **Indice : définir un w synonyme d'une recherche infructueuse**
- + “permutativité”...

Représentations : généralités

- **Attention : table n'est pas toujours un tableau !**
 - entrées peuvent être différentes d'entiers naturels...
- **Plein de manières de procéder mais en général on utilise un ensemble d'adresse A**
 - a : fonction d'adressage
 - f : fonction d'accès
 - selon A et les représentations de a et f plusieurs combinaisons sont envisageables !



Représentation de la fonction d'accès

- **Accès direct**
 - A : adresses de cellules mémoire
 - f : donne le contenu d'une cellule
- Soit, en C, un tableau V v[N] (ou V * v avec alloc dynamique), alors pour une entré i
 - $a(i)$ indice de la case de tableau contenant $t(i) = f(a(i)) = v[a(i)]$
- Exemple : $a(i) = i-1$ pour la gestion des nombres de jours dans chaque mois (1...12)
- Cas particuliers : w= 0 pour les mois numéros de mois n'existant pas ou non encodés...
- **Efficace si les valeurs sont de taille fixe**

Accès indirect (données de taille variable)

- **A : Adresses de cellules permettant d'obtenir une adresse de second niveau**
- **f : renvoie une valeur associé à cette seconde adresse**
- **Exemple : en C, on utilisera un tableau ta pour ranger les adresses de second niveau**
 - t : Nat -> Chaine (num dpt -> nom département)
 - $a(i) = i-1$
 - f : extrait de v la sous-chaine dont les indices de début et de fin sont $ta[a(i)]$ et $ta[a(i)+1]$

Adressage calculé (efficace mais gourmand en espace)

- **a : E -> A est une fonction de calcul injective ($i \neq j \Rightarrow a(i) \neq a(j)$)**
- **Exemples**
 - $a(i) = i - 1$ ou plus compliqué...
 - tableaux ayant un nb quelconque p de dimensions p.e vu comme une table t : E -> V ie
 - $E = [a_1, b_1] * \dots * [a_p, b_p]$ // début,fin de chaque dimension
 - on y accède en fournissant p indices i_1, \dots, i_p avec $a_k \leq i_k \leq b_k$
 - on peut représenter un tel tableau avec un tableau à une seule dimension
 - On a bien-sûr $n = \prod_{i=1}^p (b_i - a_i + 1)$
 - problème : réaliser l'adressage injectif :
 - $i = (i_1, \dots, i_p) \rightarrow a^p(i)$ (on aura $t(i) = v[a^p(i)]$)

Exercices

- **Limitations ?**
- **Représentations 2D, 3D ?**
- **Etude analytique pour $p=1, 2$ et 3 puis généralisation**
 - coefficients constants sur la différence ($i_k - a_k$) après développement
 - coefficients enregistrés dans le descripteur de tableau associé

Représentations

Analyse

Adressage associatif

- **Objectif : se concentrer sur les entrées effectives**
 - adressage par le contenu
- **Mémoriser les entrées effectives dans des cellules dont les adresses \in A**
 - utilisation d'une liste l où la position détermine l'adresse $a(i)$ dans un espace mémoire v
 - pour obtenir $a(i)$ il faut faire une recherche dans la liste l de i
 - dichotomique de préférence (liste contiguë et triée)
- **Exemple (en exercice) : fonction num dpt -> chef lieu**
- **Possibilité de mixer contiguë et chainée : sous ensemble fréquemment consulté**

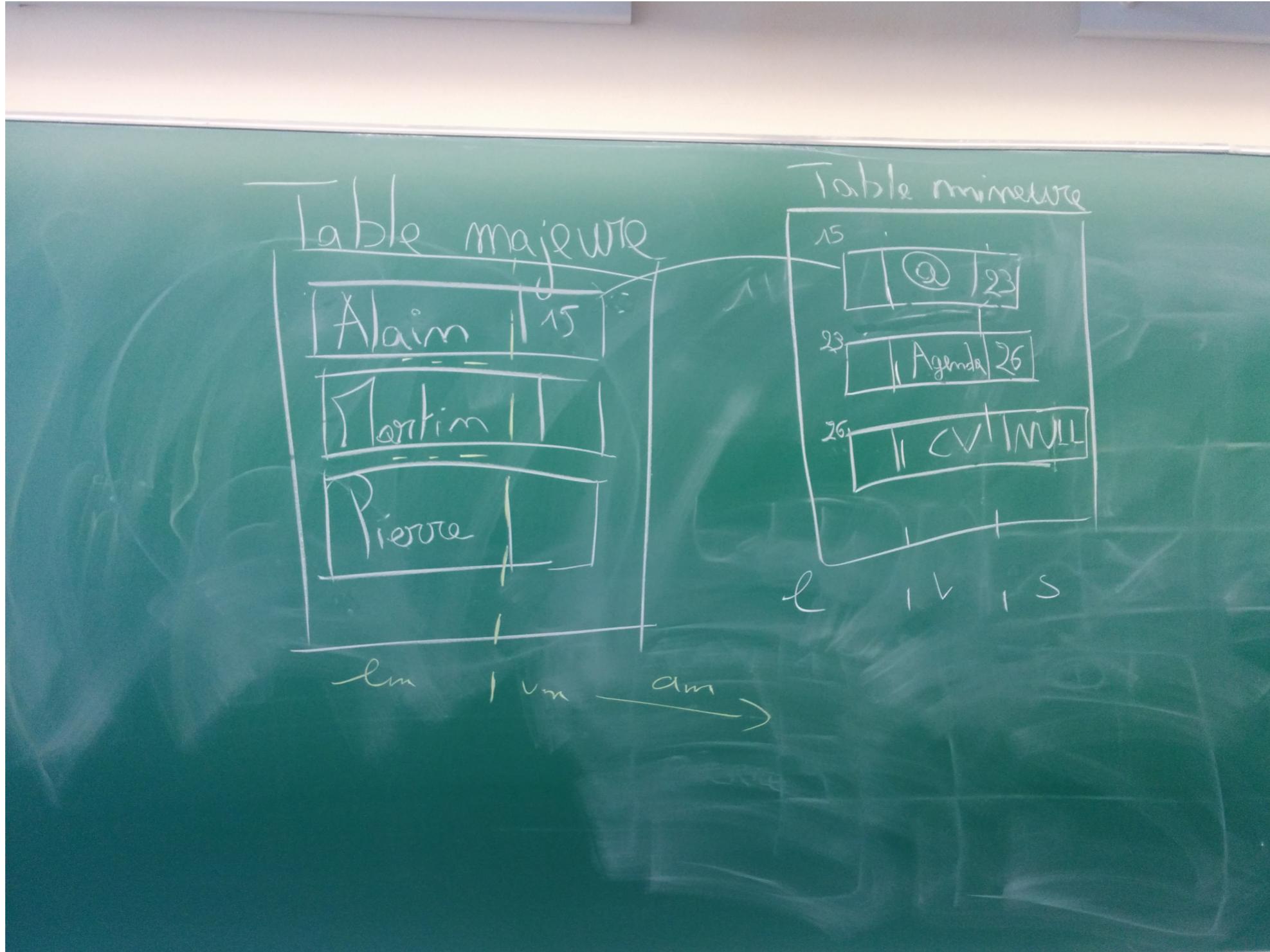
Partage de tables

- **Notion de sous-tables pour les tables volumineuses**
 - Une table majeure
 - Une ou plusieurs sous tables mineures (possibilité de hiérarchie à $p > 2$ niveaux)
- $i \rightarrow am(i) \rightarrow fm(am(i)) = tm(i)$
 - am étant non injective car plusieurs index vers même adresse
- **Deux types de rangement**
 - partitionné
 - indexé

Rangement partitionné

- **Ensemble E partitionné en sous ensemble E_1, \dots, E_m**
 - index i obtenu par tm avec am associatif
 - tm rangés dans une liste lm : position dans la liste
- **Exemple répertoire simple**
 - i = NOM/OBJ
 - NOM par liste contiguë (tableau de refs vers les autres ensembles)
 - OBJ par chainage avec fm indirect
- **Exercice : dessin DUPONT/AGENDA**

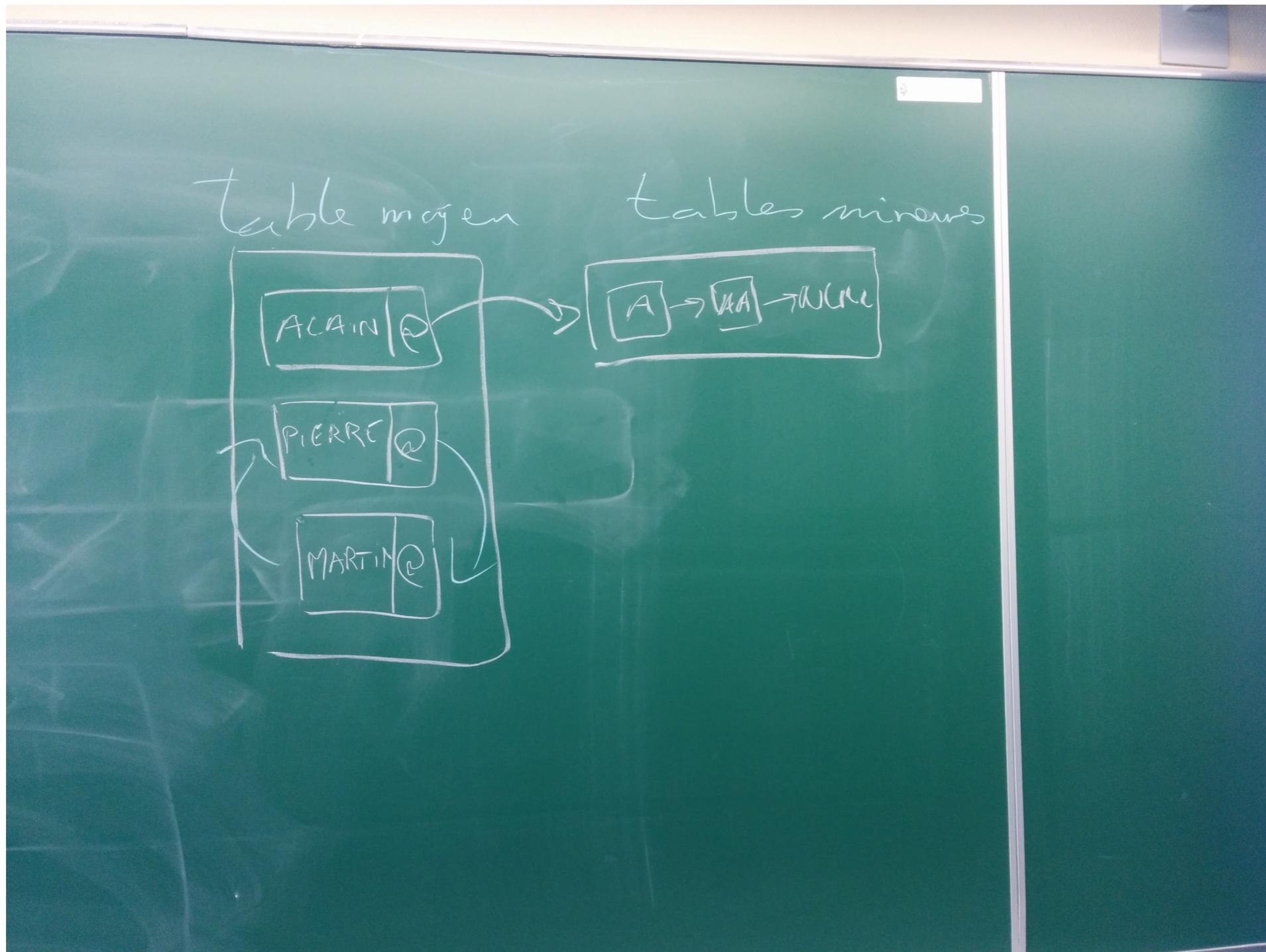
Illustration



Rangement indexé

- **Ensemble E totalement ordonné :**
 - découpage en intervalles de E_i vers des tables mineures
 - indexé lexicographiquement : liste contiguë et trié
 - recherche dichotomique
- **Exercice : dessiner pour des prénoms**
 - voir illustration à la page suivante
- **Arbres de recherche, B-arbres et B*-arbres**

Illustration



Hachage / Hashmap

- **Principes**
 - am adressage calculé := fonction de hachage $h(i) = am(i)$
 - $h(i) == h(j)$:= collision primaire
 - comment gérer ces conflits ?
 - tableau de hachage de m cellules , n nombres d'entrées effectives ($<< E$)
 - exercice : comment faire avec des prénoms ? quelles opérations ?
- **Fonction de hachage uniforme $P[h(i)=k] = 1/m$ pour tout $k \in 0...m-1$**
 - -> pas une fonction injective : quelle est la probabilité que h soit injective ?

Fonctions de hachage :

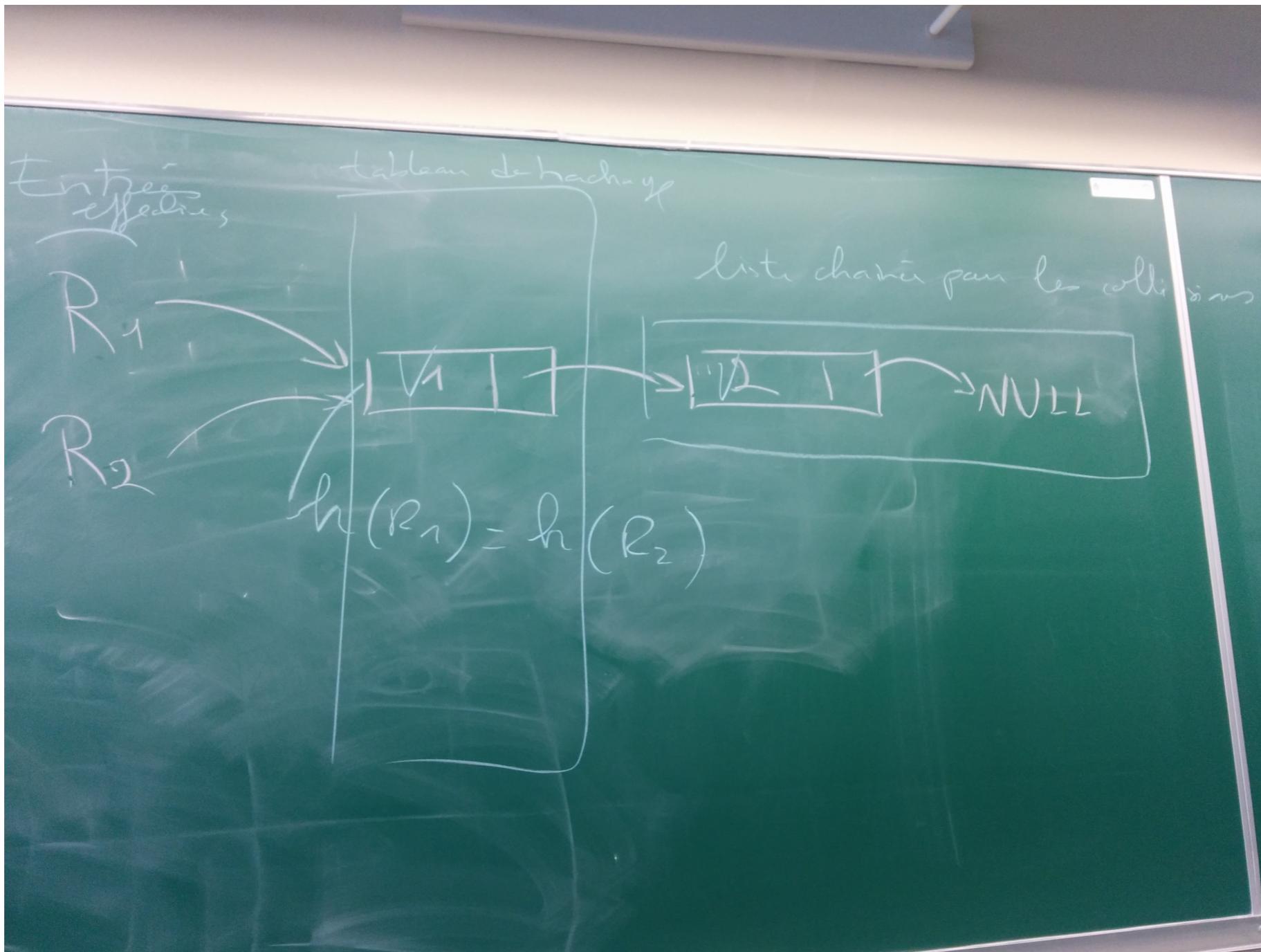
Extraction, Compression, Division, Multiplication

- **Extraction : extrait de valeurs de la représentation décimale**
 - pas terrible
- **Compression : XOR, etc**
 - permutations d'un même mot identique : décalages circulaires
- **Division : $h(i) = i \% m$**
 - comment choisir m ? premier ?
- **Multiplication : $\lfloor i \text{ceil dec}(i) * \theta * m \rfloor \text{ceil}$**
 - nombre d'or bon diffuseur uniforme
 - une des meilleures :)

Résolution des collisions par chainage

- **Chainage séparé**
 - une sous table de collision par liste chaînée sur le conflit
 - exercice : représentation graphique
- **Exercice :**
 - représentez graphiquement
 - calcul de probabilité pour la recherche négative et positive

Représentation graphique



Recherche négative et positive

- **Négative facile, positive un peu plus dur...**
 - Quelles hypothèses ?

Code

```

/* Hashtable */
struct list_head hashtable[HT_LENGTH_A]; //Router level (alias) info

/* Init */
for (i=0;i<HT_LENGTH_A;i++)
INIT_LIST_HEAD(&hashtable[i]);

/* ajout : store it in the global hashtable */
    struct conf * router = (struct conf *)malloc(sizeof(struct conf));
    int hash = jhash_1word(src,0)%HT_LENGTH_A;
    router->IP = src;
    router->nlist = NULL;
    router->flag = 1; /* router has been accessed in the last time slice */
    router->alrdflush = 0;
    list_add(&router->list,&hashtable[hash]);

/* suppression : obtain the index of the alias */
    struct conf * routo;
    int hasho = jhash_1word(src,0)%HT_LENGTH_A;
    int foundo = 0;
    list_for_each_entry(routo,&hashtable[hasho],list)
{if(routo->IP == src){foundo = 1;break;}};
```

Chainage avec collision secondaire

- **Comment faire avec une seule zone mémoire ?**
 - maximiser l'espace utilisé : utiliser les premières cases libres...
 - ...mais collisions secondaires
 - coalescence de listes, hachage coalescent
- **Exercice : représentation**
- **Complexité compliqué :(**

Méthodes sans chainage

- **Pas de gâchis mémoire du aux liens de chainage**
 - visite des permutations du mot par essais successifs : $E \rightarrow [0,m-1]^m$
 - v tableau de taille $m \geq n$
 - permutation de l'élément i noté $s(i) = (s_1(i)=h(i), \dots, s_m(i))$ de telle manière que
 - lors de la recherche
 - exploration successive jusqu'à une case vide
 - lors de la suppression en position k
 - la place est libérée pour réoccupation ultérieure
 - lors de l'adjonction
 - cherche une place libre i.e vide ou libérée
 - **Quelles stratégies ?**

Hachage linéaire

- ou à pas constant
- $s_1(i) = h(i)$
- $s_k(i) = (s_{k-1}(i) + 1) \% m = (h(i) + k - 1) \% m$
- **Attention aux effets d'accumulation à pas constant de 1**
- **Analyse de la différence avec pas p : $s_k(i) - s_l(i) = p * (k - l) \% m$**
 - Théorème de Gauss
 - si p premier avec m, alors m pas avant retour à la même adresse
 - effet d'accumulation de p en p
- **Complexité recherche positive et négative : analyse plus difficile**

Hachage quadratique (ou à pas linéaire)

- **Pas bien (perte d'espace mémoire)**
- **Essayez avec $S_K(i) = H(i) + AK + BK^2$**
 - plus de confusion en cas de collisions
 - $s(i)$ pas une vraie permutation
- **Complexité de la recherche**
 - compliqué

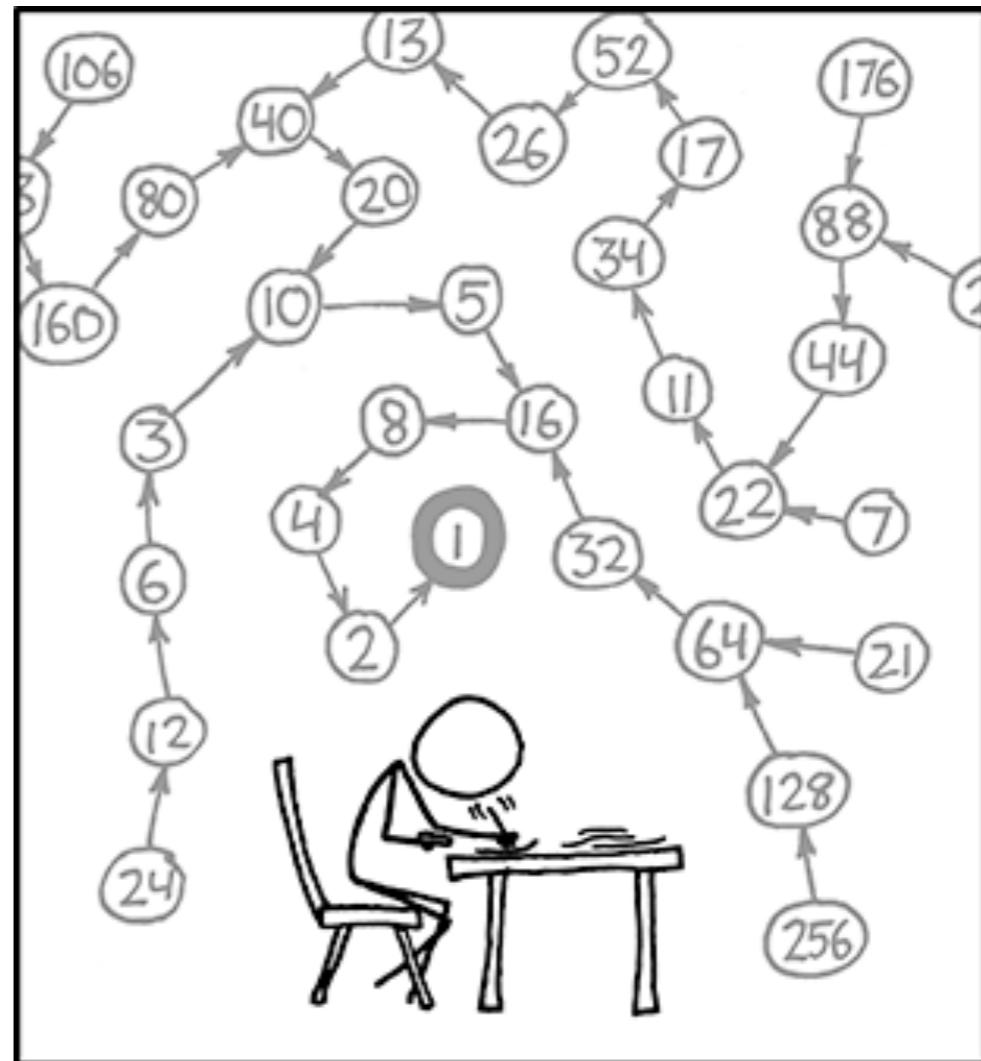
Double hachage

- $s_1(i) = h(i)$
- $s_k(i) = (s_{k-1}(i) + d(i)) \% m = (h(i) + d(i) * (k-1)) \% m$
- **balaie tout le tableau si m est premier avec tout d(i)**
 - vrai si m premier
 - $m = 2^q$ facile si $d(i) = 2d'(i) + 1$
 - permutations équiprobables : analyse de complexité facilitée
- **Complexité de la recherche**
 - négative : $1/(1-\alpha)$
 - positive : $1/\alpha \ln (1/(1-\alpha))$

Résumé : recherche négative vs recherche positive

- $\alpha = n/m := \text{taux de remplissage}$
- **fondation de hachage uniforme**
- **Double hachage : $1/(1-\alpha)$ vs $1/\alpha \ln(1/(1-\alpha))$**
- **Avec un as linéaire : $(1+1/(1-\alpha)^2)/2$ vs $(1+1/(1-\alpha))/2$**
- **Avec chainage**
 - séparé (collision primaire) : $1 + \alpha$ vs $1 + \alpha/2 - 1/(2m)$
 - pour collisions secondaires : 2.1 vs 1.8

Révisions !



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Exercices & révisions

- **Arbres & Forêts :**
 - 1- parcours post-fixe
 - Hanoi en (impératif) itératif
- **Graphes :**
 - 2- parcours suffixe récursif mais en fonctionnel
 - 3- parcours en largeur avec liste pointée, 5- puis parcours pour un tri topologique
 - Tarjan (composantes fortement connexes) + Prim, Kruskal, Dijkstra, etc en fonctionnel
- **Tables :**
 - 4- complexités (recherches positive ET négative)
 - 6- questions de compréhension et illustrations : comment gérer les collisions ? etc.

Corrections A & F

- 1- Parcours en profondeur post-fixe général (pas nécessairement binaire) d'un arbre

parcprofpostfixe : Arbre -> Rien

parcprofpostfixe(a, pr) = rien

avec (p, todo) = init (desg(a, 0, pilenouv), rien) tant que $\neg \text{vide}(p)$

 répéter si $\text{vide}(p')$ alors (p' , todo')

 sinon si $n == \text{nf}(a'')$

 alors (p' , todo')

 sinon (desg(fils(a'' , $n+1$), $n+1$, p'), todo')

 avec (a'' , n') = sommet(p')

 avec (a' , n) = sommet(p)

 avec $p' = \text{dépiler}(p)$, $\text{todo}' = \text{pr}(\text{r}(a'))$

desg : Arbre Int Pile -> Pile

desg(a, n, p) = p'

avec (a' , p') = init(empiler(p, (a, n))) tant que $\text{nf}(a') > 0$

 répéter (fils($a', 1$), empiler (p' , (fils($a', 1$), 1)))

Corrections A & F

- **Les tours de Hanoi en itératif impératif**
- **...à vous de jouer !**
 - + version fonctionnelle itérative ?

Corrections Graphes

- **2- Parcours suffixe récursif**

parcprofsuffixe : Graphe → Liste

parcprofsuffixe(g) = l avec (l, em) = parcprof1(g, es(g), listenouv, Λ)

parcprof1 : Graphe Ens Liste Ens → Liste Ens

pré **parcprof1**(g, e, l, em) = (z ∈ e ∨ app(l, z) ∨ z ∈ em) ⊃ exs(g, z)

parcprof1(g, e, l, em) =

si v(e) alors (l, em)

sinon si z ∈ em alors parcprof1(g, s(e, z), l, em)

sinon parcprof1(g, s(e, z), adjq(l1, z), em1)

avec (l1, em1) = parcprof1(g, esuc(g, z), l, i(em, z))

avec z = ch(e)

Corrections Graphes

- **3- Parcours en largeur avec une liste pointée**

parclarg : Graphe Ens \rightarrow Liste

pré **parclarg**(g, e) = $z \in e \supset \text{exs}(g, z)$

parclarg(g, e) = début(lp) avec lp = **parclarg1**(g, aqe(fin(lp), e))

Pointeurs Liste l / file f: Début(lp) / Fin(lp)

Obtenir le point de coupure liste / file : fenêtre(lp)

Déplacer coupure vers la droite : dcvd(lp)

Composition par concaténation de deux listes : comp(l1, l2)

Suppression de l'élément dans la fenêtre : sp(lp)

Test d'appartenance : app(lp, x) = app(début(lp), x) \wedge app(fin(lp), x)

Adjonction en queue d'une liste l des sommets d'un ensemble e : aqe(l, e)

parclarg1 : Graphe Listepointée \rightarrow Listepointée

pré **parclarg1**(g, lp) = app(lp, z) $\supset \text{exs}(g, z)$

parclarg1(g, lp) =

si vide(fin(lp)) alors lp

sinon si app(lp, z) alors **parclarg1**(g, sp(lp))

 sinon **parclarg1**(g, dcvd(comp(début(lp), aqe(fin(lp), esuc(g, z)))))

 avec z = fenêtre(lp)

Corrections Graphes

- 5- Vers un parcours en largeur pour un tri topologique (version avec ch)

```

racines : Graphe -> Ens
  racines(g) = recherche(g, es(g), Λ)
recherche : Graphe Ens Ens -> Ens
  recherche(g, es, er) =
    si v(es) alors er
    sinon si di(g, x) == 0 alors recherche(g, s(es, x), i(er, x))
      sinon recherche(g, s(es, x), er)
    avec x = ch(es)

```

```

tritopo : Graphe -> Liste
  tritopo(g) = si v(g) alors listenouv sinon adjt(tritopo(supsa(g, x)), x)
    avec x = ch(racines(g))

```

ou avec une opération intermédiaire algo, mémorisant les sommets candidats dans er :

```

tritopo(g) = algo(g, racines(g), listenouv)
algo : Graphe Ens Liste -> Liste
  algo(g, er, l) =
    si v(er) alors si v(g) alors l sinon algo(g, racines(g), l)
    sinon algo(supsa(g, x), s(er, x), adjq(l, x))
  avec x = ch(er)

```

Corrections Graphes

- **Tri topologique avec un parcours en largeur (version avec file)**

`tritopo : Graphe -> Liste`

`tritopo(g) = parclarg3(g, adjqe(filenouv, racines(g)), listenouv)`

avec les opérations `adjqe(f, e)` ajoutant les éléments de l'ensemble `e` en queue de la file `f`, et `appe(l, e)` testant si tous les éléments de l'ensemble `e` sont dans la liste `l` :

`parclarg3 : Graphe File Liste -> Liste`

`parclarg3(g, f, l) =`

`si vide(f) alors l`

`sinon si app(l, z)`

`alors parclarg3(g, supt(f), l)`

`sinon si appe(l, epred(g, z))`

`alors parclarg3(g, adjqe(supt(f), esuc(g, z)), adjq(l, z))`

`sinon parclarg3(g, adjq(supt(f), z), l)`

`avec z = tête(f)`

Corrections Graphes

- **Algos plus évolués : Tarjan, Kruskal, Prim, etc**
- **...à vous de jouer...**

Corrections Tables

- **4- Complexité moyenne d'une recherche positive dans une hashmap (avec résolution des collisions par chainage séparé & ajout en queue sans ré-organisation de la sous liste)**
 - le coût moyen de la recherche d'un élément est égal au cout de l'ajout de cet élément

$$C_+(t) = \frac{1}{n} \sum_{i=1}^n C_{aj}(t,i)$$

- le coût moyen de l'ajout d'un élément est égal au coût moyen de sa recherche négative au pire

$$C_{aj}(t,i) = C_-(t,i-1)$$

- D'où (indépendance à t mais seulement à n et m et son rapport...) :

$$C_+(t) = \frac{1}{n} \sum_{i=1}^{n-1} C_-(t,i) = \frac{1}{n} \sum_{i=1}^{n-1} \left(1 + \frac{i}{m}\right) = 1 + \frac{(n-1)}{2m} = 1 + \alpha / 2 - 1 / 2m$$

Questions de compréhension

- **Exemples**

- A quoi peut bien servir un parcours en profondeur...
 - ...préfixe ? ...infixe ? ...suffixe ?
- Quel est le meilleur algorithme pour calculer une fermeture transitive ? justifiez.
- Dans quel cas utiliser Floyd plutôt que n*Dijkstra pour le calcul de tous les chemins ?
- Qu'est-ce qu'une collision dans le contexte des tables ?
 - donnez plusieurs façons de gérer le problème.
- Quelle est la complexité d'une recherche dichotomique ? Justifiez.
- Pourquoi une recherche négative est plus couteuse qu'une recherche positive ?
- Comment fonctionne l'adressage calculé ? quels sont ses limitations ?
- Quel est le défaut des méthodes avec chainage par rapport à celles avec chainage ?
- Quelle est la probabilité qu'une fonction de hachage uniforme produise k collisions sur une case et aucune sur les autres cases ?