INF421-a
Bases de la programmation et de l'algorithmique

Tables d'associations, hachage

(Bloc 3 / 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

23 septembre 2005

Aujourd’hui

Recherche d’information dans un ensemble dynamique

- Associer des données (*e.g.*, clients / commandes)
- Les retrouver
- Les manipuler (insertion, suppression, etc)

Exemple : Les pages blanches

- Associer un nom (*i.e.*, une chaîne de caractères)
- à un numéro de téléphone

```
"Camille Abate"  "04.91.63.50.30"
"Philippe Zito"  "06.94.16.13.38"
"Laurie Zimmerman"  "04.32.32.39.63"
"Ugo Zelaya"  "04.70.61.50.56"
"Laure Zamor"  "02.67.37.88.41"
"Oceane Archambault"  "06.76.06.22.09"
...
```

Une idée simple : Les listes d'associations

- Créer une structure qui enregistre les références vers les objets à associer
  - ici deux `String`, le nom et le téléphone

```
class ListeAssoc {
    String nom;
    String tel;
    ListeAssoc suivant;
    ListeAssoc(String nom, String tel, ListeAssoc suivant) {
        this.nom = nom;
        this.tel = tel;
        this.suivant = suivant;
    }
}
```

Une idée simple : Les listes d'associations

```
// Trouver un numéro de téléphone
static String telDe(String nm, ListeAssoc l) {
  while (l ≠ null) {
    if (l.nom.equals(nm))
      return l.tel;
    l = l.suivant; }
  return null;}

// Modifier un numéro de téléphone
static void changerTel(String nm, String nouvTel, ListeAssoc l) {
  while (l ≠ null) {
    if (l.nom.equals(nm)) {
      l.tel = nouvTel;
      return; }
    l = l.suivant; }
  throw new Error(nm + " pas enregistré");}
```

Une idée simple : Les listes d'associations

```
static ListeAssoc supprimer(String s, ListeAssoc a) {
    if (a == null)
        return null;
    if (a.nom.equals(s))
        return a.suivant;
    ListeAssoc prec = a, cour = prec.suivant;
    for (; cour ≠ null; prec = cour, cour = prec.suivant)
        if (a.nom.equals(s)) {
            prec.suivant = cour.suivant;
            return a;
        }
    return a;
}
```

Une idée simple : Les listes d’associations

```
long tme = System.currentTimeMillis(); // pour mesurer le temps
ListeAssoc la = null;
for (;;) { // lecture d'un fichier (format "Alex Yu"  "04.91.63.50.30")
    String nom = ST.ReadString(); // lire une chaîne
    if (nom.equals("ENDFILE"))  break; // fin de la lecture du fichier
    la = new ListeAssoc(nom, ST.ReadString(), la); }
System.out.println("CHARGÉ  EN " +
                   (System.currentTimeMillis() - tme) + "ms");
tme = System.currentTimeMillis();
System.out.println("Tel de Laure Archambault : " +
                  ListeAssoc.telDe("Laure Archambault", la) + " " +
                   (System.currentTimeMillis() - tme) + "ms");
tme = System.currentTimeMillis();
System.out.println("Tel de Frederic Zito : " +
                  ListeAssoc.telDe("Frederic Zito", la)+ " " +
                   (System.currentTimeMillis() - tme) + "ms");
```

Les listes d'associations sur 248000 Enregistrements

```
java hash < PrenomNomTel.txt
CHARGÉ  1863ms
Tel de Laure Archambault : 02.32.61.45.56 40ms
Tel de Frederic Zito : 06.46.87.43.15 0ms
```

- Avantage de la simplicité
- Inconvénient majeur : Complexité algorithmique catastrophique
  - Parfois utile quand “l'univers” est petit
- Amélioration ? Trier la liste des abonnés (lookup table).
  - Une fausse bonne idée. Pourquoi ?

Liste d'association ordonnée (lookup table)

- Insérer un nouvel élément : $O(n)$
  - Il faut déplacer "à droite" les $O(n)$ éléments déjà présents
- Enlever un élément prend : $O(n)$
  - Il faut déplacer "à gauche" les $O(n)$ éléments déjà présents
- Chercher un élément prend dans le pire des cas un temps $O(\log n)$, lorsqu'on utilise la recherche binaire

Recherche binaire dans une liste d'association ordonnée (lookup table)

On cherche un élément de clef k :

- Comparer k avec la clef de l'élément en milieu de séquence
  - Si k est $<$ a cette clef, chercher l'élément de clef k dans la partie gauche
  - sinon, dans la partie droite
- Division par 2 à chaque itération $\longrightarrow O(\log n)$

Rechercher la clef 9

- 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- 1 4 5 6 **9** 12 13 15 19 21 22 24 29 30

Aujourd’hui

Rechercher de l'information dans un ensemble dynamique

- Les éléments de l'ensemble sont constitués de plusieurs champs
- L'un des champs, la clef $\in$ ensemble totalement ordonné U, appelé l'univers des clefs
  - une clef = un entier ; l'univers = un intervalle d'entiers (muni de l'ordre naturel)
  - une clef = une chaîne ; l'univers = un ensemble de chaînes de longueur bornée (muni de l'ordre lexicographique)

  Pour avoir accès à toutes les informations, on recherche l'élément à l'aide de sa clef. Attention plusieurs éléments peuvent avoir la même clef.
- On veut aussi pouvoir insérer ou supprimer un élément.

Une telle structure = dictionnaire (nom générique ; plusieurs codages possibles, table de hachage, arbres binaires de recherche, "skiplists", *etc.*)

Un cas trivial : un petit univers

Si (1) l'univers est petit et (2) les clefs sont uniques (*i.e.*, des éléments distincts ont des clefs distinctes)

- Associer un entier à chaque clef (dans la suite on suppose que la clef = un entier)
- Construire un tableau `T[]` tel que `T[c]` est l'élément dont la clef est c

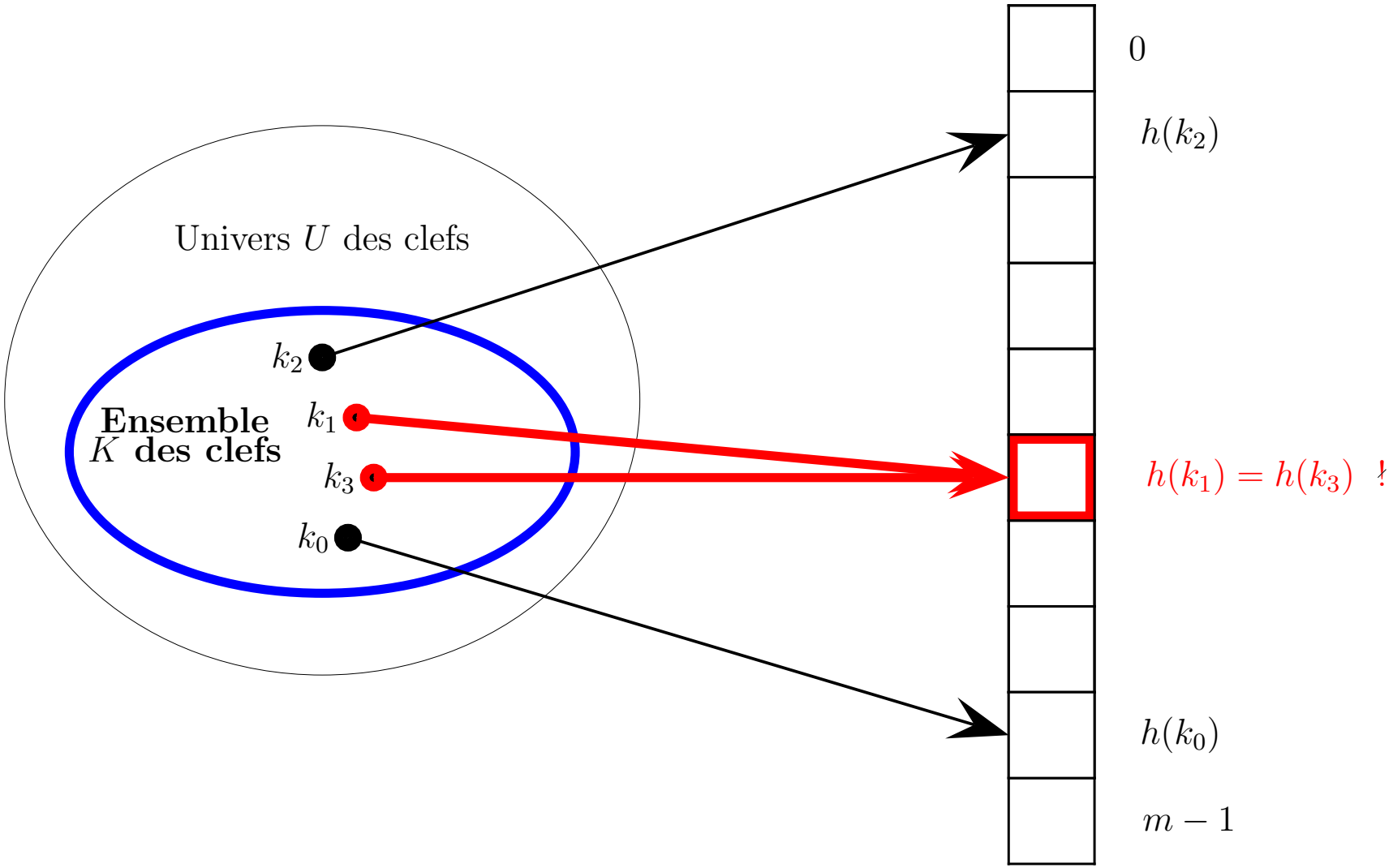
Hachage : la fonction de hachage

- Ranger dans un tableau `T` de taille m ($\ll$ l'univers) toutes les clefs
- Fonction de hachage $h : U \to \{0, \ldots, m-1\}$
- Un élément de clef c est rangé dans $T[h(c)]$

Problèmes :

- Comment choisir m ? Que faire quand le tableau déborde ?
- Comment construire h ?
- Plusieurs clefs sont hachées identiquement. Que faire ?
  - T ne peut donc pas être un simple tableau

Hachage : les collisions

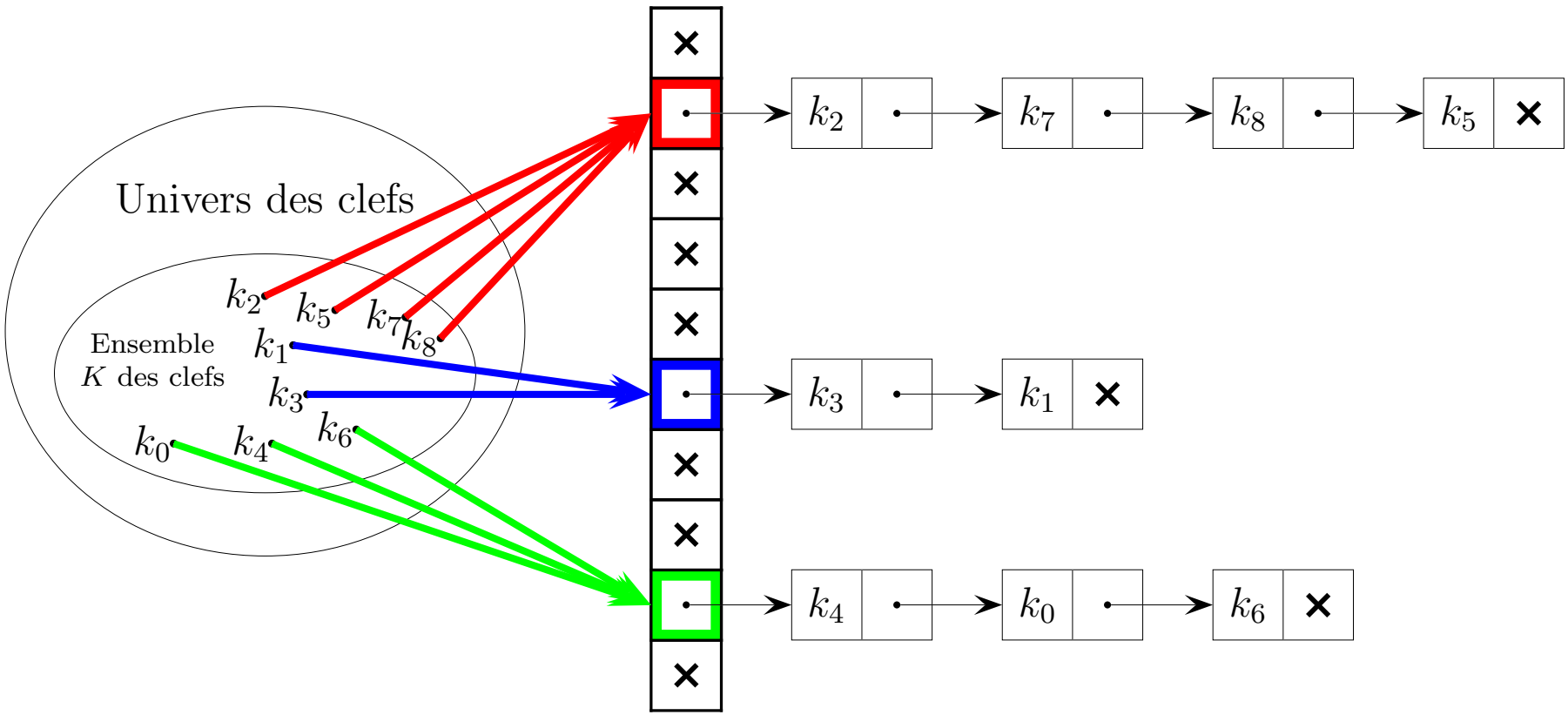


Hachage : les collisions

Deux solutions pour résoudre les collisions :

- Le chaînage : Les clefs qui ont la même valeur de hachage sont mises dans la même liste
- L'adressage ouvert : On cherche la première place libre dans le tableau après la valeur de hachage

Hachage : le chaînage



Hachage : le chaînage par l'exemple

- Une table de taille 12 et une (très mauvaise) fonction de hachage qui renvoie la somme des codes ASCII modulo 12.
- Insertion de Lea Abate 02.77.32.38.62, Sarah Zhao 05.46.69.77.66, Lea Eabat 05.77.89.98.00, Ugo McAdam 06.72.75.09.34, Quentin Dupaul 03.60.22.93.66, Bela Atea 05.77.39.38.21.
- Les codes :
  - Lea Abate $\longrightarrow$ 3
  - Sarah Zhao $\longrightarrow$ 5
  - Lea Eabat $\longrightarrow$ 3
  - Ugo McAdam $\longrightarrow$ 2
  - Quentin Dupaul $\longrightarrow$ 11
  - Bela Atea $\longrightarrow$ 3

Beaucoup de collisions (*i.e.*, de clefs rangées dans le même "bucket"). Pourquoi ?

Digression : ASCII

- American Standard Code for Information Interchange
- Les caractères sont codés sur 7 bits
  - 128 caractères possibles, de 0 à 127
- Les codes 0, 1,... , 31 = caractères de contrôle (CR, BEL)
- Les codes 65 à 90 : LES MAJUSCULES
- Les codes 97 à 122 : les minuscules

$A \longrightarrow 41$, $B \longrightarrow 42$, $C \longrightarrow 43$, $D \longrightarrow 44$, $E \longrightarrow 45$, $F \longrightarrow 46$, ..., $a \longrightarrow 61$, $b \longrightarrow 62$, $c \longrightarrow 63$, $d \longrightarrow 64$, $e \longrightarrow 65$, $f \longrightarrow 66$,...,

Hachage : le chaînage

```
class TableHachageChainee {
    static final int m = 350000; // taille de la table
    ListeAssoc[] table;
    TableHachageChainee() {
        table = new ListeAssoc[m]; }
    static int codeHachage(String s) { // la mauvaise fonction de h
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
            sum += s.charAt(i); }
        return sum % m; }
    static void ajout(String nom, String tel, TableHachageChainee t) {
        int h = codeHachage(nom);
        t.table[h] = new ListeAssoc(nom, tel, t.table[h]); }
    static String telDe(String nom, TableHachageChainee t) {
        int h = codeHachage(nom);
        return ListeAssoc.telDe(nom, t.table[h]); }
}
```

Hachage : le chaînage

- Insertion en $O(1)$
- Recherche d'un élément se fait
  - dans le pire des cas en $O(n)$
  - en moyenne en temps $O(1+\alpha)$, où $\alpha = n/m$ (la charge) si les h(k) sont uniformément distribués

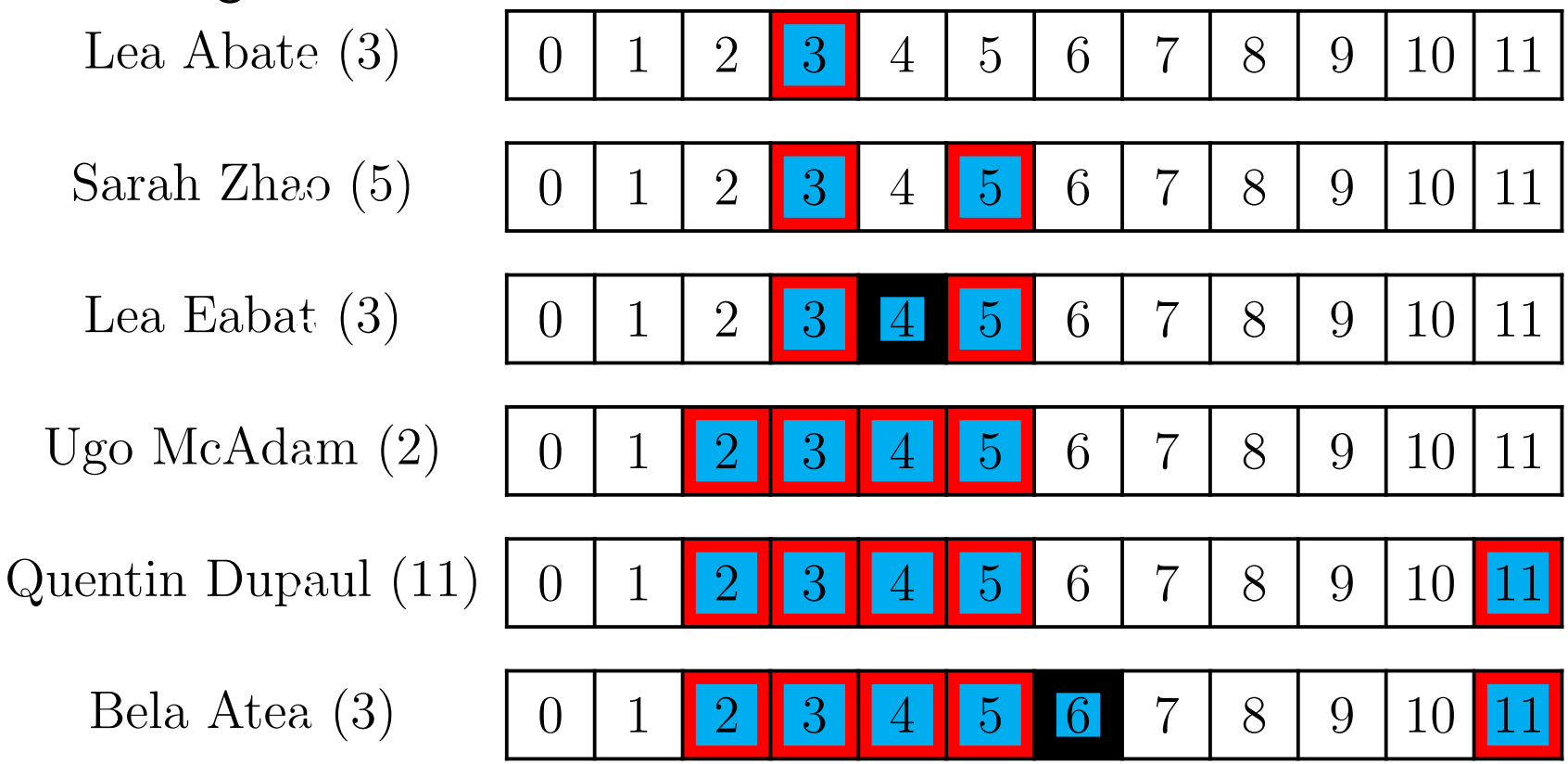
Hachage : le chaînage

Que faire quand la table est pleine (ou presque) ?

- Doubler la capacité de la table
- Trouver une nouvelle fonnction de hachage
- Recopier la vieille table dans la nouvelle
- Espérer que ca n'arrive pas trop souvent

Hachage : l'adressage ouvert

On cherche la première place libre dans le tableau après la valeur de hachage.



Hachage : l'adressage ouvert

Une vesrion simplifiée de `ListeAssoc`

```
class Couple {
    String nom;
    String tel;
    Couple(String nom, String tel) {
        this.nom = nom;
        this.tel = tel;
    }
}
```

Hachage : l'adressage ouvert

```
class TableHachageAdrOuvert {
    static final int m = 550000; // taille de la table
    static long nbCollisions = 0;
    static long nbIn = 0;
    Couple[] table;
    TableHachageAdrOuvert() {
        table = new Couple[m];
    }
    static int codeHachage(String s) { // une mauvaise fonction de H
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            sum += c;
        }
        return sum % m;
    }}
```

Hachage : l'adressage ouvert

```
static void ajout(String nom, String tel, TableHachageAdrOuvert t) {
    int h = codeHachage(nom);
    for (int i = h; i ≠ ((h-1) % m); i = (i+1) % m)
        if (t.table[i] == null) {
            t.table[i] = new Couple(nom, tel);
            return; }
    throw new Error("Table pleine");
}
static String telDe(String nom, TableHachageAdrOuvert t) {
    int h = codeHachage(nom);
    for (int i = h; i < ((h-1) % m); i = (i+1) % m) {
        if (t.table[i] == null)
            return null;
        if (t.table[i].nom.equals(nom))
            return t.table[i].tel; }
    return null;
}}
```

Hachage : l'adressage ouvert

- Attention aux suppressions !
- Si on supprime un element, le test de recherche peut ne pas fonctionner !
- En cas de suppression, il faut tout re-hacher

Hachage : l'adressage ouvert

Attention aux phénomènes de regroupements.

- Même avec une distribution de $h(k)$ uniforme, la probabilité d'occupation d'une case n'est pas constante
  - Bien plus forte quand la case est immeditament après un bloc de cases déjà utilisées
- Solution : double hachage .
  - Deux fonctions de hachage $h : U \to \{0, \ldots, m-1\}$ et $h' : U \to \{0, \ldots, r-1\}$.
  - Collision $\to$ on choisit successivement les cases $h(k) + h'(k)$, $h(k) + 2h'(k)$, $h(k) + 3h'(k)$, etc.

Hachage : complexité de l'adressage ouvert

Quelle est la complexité en moyenne d'une recherche ?

- Piste : au moins 1 test et avec probabilité α ($\alpha = \frac{1}{m}$, la charge) 2 tests ; avec probabilité α^2 tests, etc.
- Si la table est à moitié pleine $\longrightarrow$ 2 itérations (en moyenne)
- Si la table est à pleine à 90% $\longrightarrow$ 10 opérations (en moyenne)
- Qu'en est-il des autres opérations ?
- Règle de cuisine : table saturée si $\alpha > 40\%$

Les fonctions de hachage, les recettes

Une fonction de hachage associe un entier à un objet (une chaîne, un entier, ou un objet quelconque)

4 règles d'or (irréalisables)

1. La valeur de hachage ne dépend que de l'objet d'entrée
2. La fonction de hachage utilise tous les champs de l'objet d'entrée
3. La fonction de hachage distribue uniformément les données
4. La fonction de hachage génère des valeurs très différentes pour des objets quasi identiques

Les fonctions de hachage, les recettes

Les 4 règles sont-elles vérifiées ?

```
static int codeHachage(String s) { // une mauvaise fonction de H
    int sum = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        sum += c;
    }
    return sum % m;
}
```

Mauvaise fonction de hachage : “Bela Atea” et “Lea Abate” donnent la même `sum` et donc ont le même H code.

Les fonctions de hachage pour les entiers

- $h(k)$ le reste de la division de k par m : $h(k) = k \mod m$
  - Attention si $m = 2^r$, $h(k)$ ne dépend que des r derniers bits de k.
  - Une bonne valeur pour m est un nombre premier (pas trop proche d'une puissance de 2). Peut se comporter très mal quand même.
- Variante de la division $h(k) = k(k+3) \mod m$ (Knuth)
- Multiplication . $m = 2^r$ et A un réel $h(k) = \lfloor m(Ak - \lfloor Ak \rfloor) \rfloor$.
  - Knuth propose le nombre d'or $A = \frac{1}{2}(\sqrt{(5)} - 1)$

Les fonctions de hachage pour les `String`

- Partir du code ASCII du caractère (entier sur 7 bits, compris entre 0 et 127)
- H code d'une chaîne de caractère = entier en base 128.
- Le H code de “java” est alors
  $106 \times 128^3 + 97 \times 128^2 + 118 \times 128 + 97 = 223902561$

Aujourd’hui

En pratique, la classe `Hashtable`

- Dans le package “`java.util`”
- un package = ensemble de classes java
  - Utilisation d'une classe `Hashtable` d'un package `java.util` : `java.util.Hashtable`
  - ou plus simplement `import java.util.*` au début du fichier puis `Hashtable`
- `Hashtable` implémente une structure associe à une donnée (un objet) associe une clef (un autre objet).
- “Comme” un tableau d'objets avec des indices non numériques.

En pratique, la classe Hashtable

- Les constructeurs :
  - `Hashtable()` : crée une table vide de dimension 0,
  - `Hashtable(int n)` : crée une table vide de dimension n
  - `Hashtable(int n, float alpha)` : crée une table vide de dimension n et dont la taille augmente quand la charge dépasse alpha
- `put(Object clef, Object monObjet)` : ajoute un élément monObjet et sa clef à la table
- `get(Object clef)` retourne l'élément de la table dont la clef est clef.
- `remove(Object clef)` supprime l'élément de la table dont la clef est clef.

Attention ! put et get ne sont pas static

En pratique, la classe Hashtable

```
String[] tab = new String[500000]; long tm = System.currentTimeMillis()
Hashtable ht = new Hashtable(); int taille = 0;
for (;;) {                                  // REMPLIRE
    String nom = ST.ReadString();
    tab[taille] = nom;
    if (nom.equals("ENDFILE")) break;
    ht.put(nom, new Couple(nom, ST.ReadString()));
    taille++;}
System.out.println("CHARGE  EN " +
                   (System.currentTimeMillis() - tm) + "ms");
tm = System.currentTimeMillis();  // TESTER
for (int i = 0; i < tab.length && i < taille; i++) {
    Couple c = ((Couple) (ht.get(tab[i])));
    if (c == null) throw new Error("VIDE  pour " + tab[i]) ;
}
System.out.println("cpu = " + (System.currentTimeMillis() - tm));
```

Object ?

- Tout ce qui n'est pas primitif est un `Object`
- A la création d'un objet (comme `Hashtable` ou `Couple`), on hérite des méthodes définies par la classe `Object`.
- Certaines de ces méthodes peuvent être redéfinies
  - Attention, suivre exactement la signature
- Un exemple déjà vu : `toString`

```
class Personne {
    String nom; int age;
    public String toString() {
        return nom + " j'ai " + age + " ans";  }}
```

- D'autres méthodes utiles :
  - `public boolean equals(Object o)`
  - `public int hashCode()`

Le cast

Que veut dire
`Couple c = ((Couple) (ht.get(tab[i])));` ?

- Rappel : `Object get(Object clef)` retourne l'élément de la table dont la clef est clef
- C'est un `object`
- mais dans notre cas, on sait que c'est un couple mais le compilateur n'a aucun moyen de le savoir
- Si on écrit `Couple c = ht.get(tab[i])`, on obtient une erreur de compilation

```
hash.java:348: incompatible types
found    : java.lang.Object
required: Couple
```

- Il faut explicitement convertir le type ; c'est un "downcast"

Aujourd’hui

L'art du raccourci

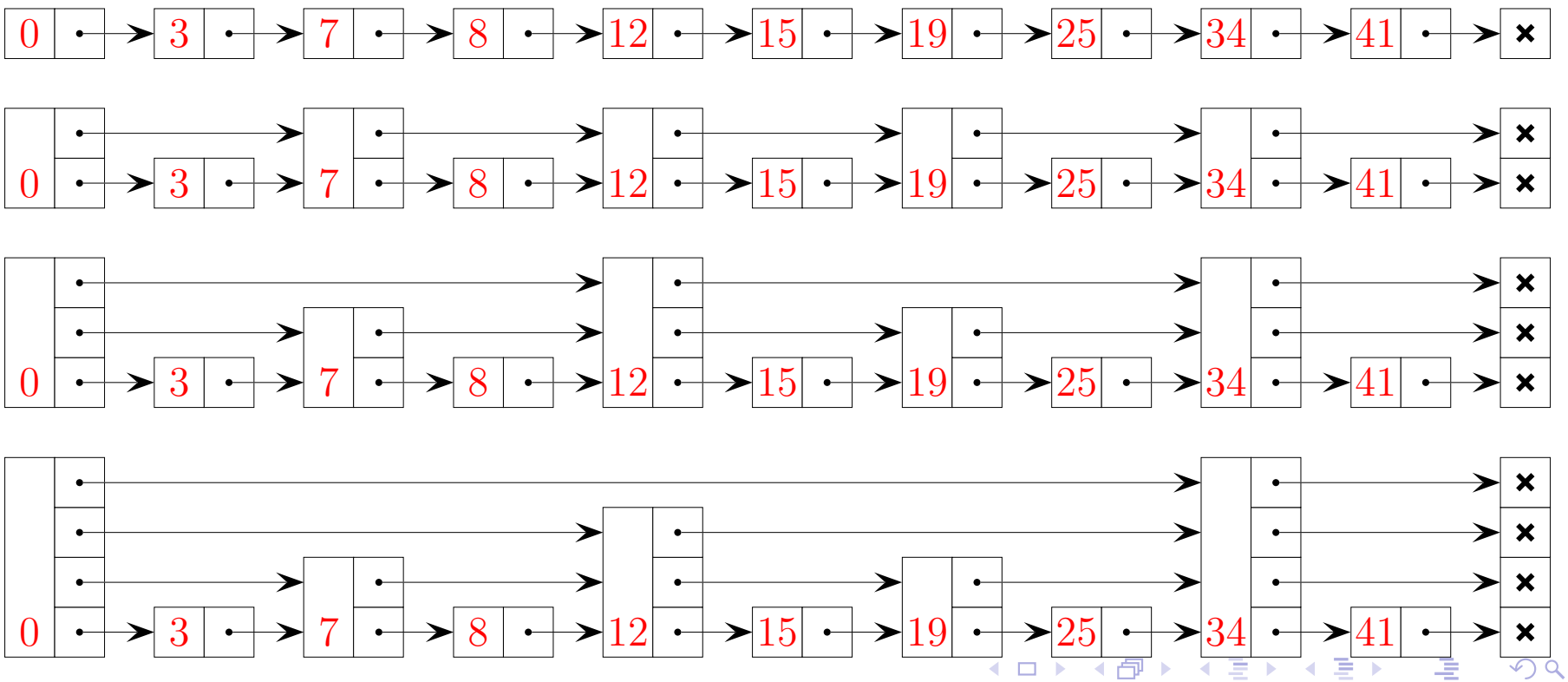
- Rappel : recherche d'un élément d'une liste chaînée en $O(n)$
- Rappel : suppression d'un élément d'une liste chaînée en $O(n)$

Trouver une/des structures de données pour aller plus vite quand les éléments ont une clef

Une idée très simple les listes de niveaux : On mémorise des raccourcis vers des éléments de la liste triée

Les listes de niveaux

Un tableau de pointeurs par élément (1 pointeur = 1 niveau) et Le niveau k contient un élément sur 2^k



Les listes de niveaux

Quelques questions fondamentales : pour une liste à niveaux triée de taille $O(n)$,

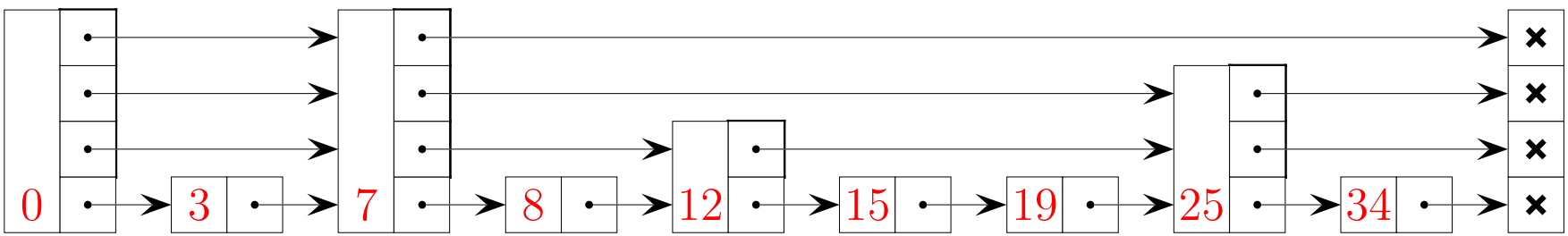
- quel est l'espace mémoire utilisé ?
  - $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... = O(n)$
- quel est le coût de recherche d'un élément ?
  - Dichotomie
- quel est le coût d'insertion d'un élément ?
  - Insérer et tout reconstruire

Une skiplist, un algorithme randomisé

- Un algorithme randomisé est contrôlé par une suite de tirages aléatoires. (*e.g.* pile ou face)
- Dans la plupart de ces algorithmes, le temps d'exécution, au pire cas , n'arrive qu'avec une très faible probabilité. (*e.g.* toujours "pile" jamais "face")

Une skiplist

- Liste t riée à niveaux randomisés
- p un réel fixé $(0 < p < 1)$
- Probabilité [niveau $= q$] $= p^{q+1}$



En pratique, on utilise pour simplifier le codage un niveau maximal possible

Les skiplists : une applet java

La suite au cours 4

Le TD du jour

Des statistiques sur les prénoms des élèves du cours INF421
$\longrightarrow$ Rechercher de l'information dans des ensembles dynamiques