INF 421 — 04 Luc Maranget

Associations

Luc.Maranget@inria.fr
http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Plan

A Table d'association/(Java : interface)

B Table de hachage/(Java : bibliothèque)

C Bonus Java

Les tables d'associations

- Qu'est-ce que c'est ? Des associations de n'importe quoi à n'importe quoi.
- Comment ça marche ? Trois implémentations.
  - Avec des listes.
  - Avec notre table de hachage.
  - Avec les *tables de hachage* de la bibliothèque.

La table d'association

Soit des « *informations* », munies d'une « *clé* ».

- La clé identifie d'information.
  - Nom (et prénom) → numéro de téléphone dans l'annuaire.
  - Matricule → soldat.
  - Numéro d'immatriculation → véhicule, dans le fichier des cartes grises.
- On se place dans le cas où la clé identifie une information unique (par ex. fichier des cartes grises).

La table d'association

Ensemble *dynamique* d'informations.

Type abstrait de données, défini par les opérations.

- Trouver l'information associée à une clé donnée.
- Ajouter une nouvelle association entre une clé et une information.
- Retirer une clé de la table (avec l'information associée).

Et plus précisément (unicité des clés)

- S'il existe déjà une information associée à la clé dans la table, alors la nouvelle information remplace l'ancienne.
- Sinon, une nouvelle association est ajoutée à la table.

Application

On veut produire une statistique des mots d'un texte.

- Voici par exemple un texte bien connu.

```
% cat marseillaise.txt
Allons ! Enfants de la Patrie !
Le jour de gloire est arrivé !
etc.
```

- Statistique des fréquences des mots de la Marseillaise (mots de taille ≥ 4, majuscules enlevées).

```
% java Freq marseillaise.txt
nous: 10
vous: 8
français: 7
  etc.
```

Conception de `Freq`

En trois étapes.

1. Lire le texte mot-à-mot.
2. Compter les occurrences des mots.
3. Produire un bel affichage (par fréquences décroissantes).

Plus précisément.

1. Procéder selon le principe du `Reader`.
2. Par une table d'association des mots aux comptes. (clé = un mot $\rightarrow$ information = un compte).
3. Par un tri à la fin.

Lecture des mots, tri

On suppose donnés :

- Une classe des objets « lecteur de mots » (`WordReader`).
  - ▷ Constructeur `WordReader (String name)` pour créer le lecteur des mots d'un fichier de nom `name`.
  - ▷ Méthode `String read()`, renvoie le mot suivant (ou **null** quand c'est fini).
- Un tri des paires, mot $\times$ compte (notées $w \to c$). Selon l'ordre total :

$$(w_1 \to c_1) < (w_2 \to c_2)$$

$$\Updownarrow$$

$$(c_1 > c_2) \vee (c_1 = c_2 \wedge w_1 < w_2)$$

Définition précise de notre table

Une table est un objet `Assoc`. C'est une table d'association des mots (`String`) aux **int**.

- Méthode **int** `get(String w)`, renvoie le compte associé à `w`.
  - Ou bien 0 si `w` n'est pas dans la table.
- Méthode **void** `put(String w, int c)`, associe le compte `c` au mot `w`.

Spécification en Java, par une *interface* (commentée).

```
interface Assoc {
  /* Créer/remplacer l'association key → val */
  void put(String key, int val) ;
  /* Trouver l'entier associé à key, ou zéro. */
  int get(String key) ;
}
```

Code de `main`

```
class Freq {
  ...
  static void countFile(String name, Assoc t) {
    WordReader wr = new WordReader(name) ;
    count(wr, t) ;
    wr.close() ;       // Fermer l'entrée (plus propre)
    Sort.println(t) ; // Affichage trié
  }

  public static void main(String [] arg) {
    countFile(arg[0], ...)
  }

}
```

Code de `count`

```
static void count(WordReader in, Assoc t) {
  for (String word = in.read() ;
       word != null ;
       word = in.read()) {
    if (word.length() >= 4) {
      word = word.toLowerCase() ;
      t.put(word, t.get(word)+1) ;
    }
  }
}
```

Remarquer : `Assoc` n'est pas une classe, mais on peut l'utiliser comme un type, celui du deuxième argument de `count`.

Les listes d'associations

La façon la plus simple d'associer un compte à un mot ?

Une liste de paires `String` → **int**.

```
class AList {
  private String word ;
  private int count ;
  private AList next ;

  AList (String word, int count, AList next) {
    this.word = word ; this.count = count ; this.next = next ;
  }
}
```

Attention

- On suppose qu'à un mot est associé au plus un compte (unicité des clés).
- La table vide est représentée par... **null**.

La table avec une liste

Étant donnés un mot (`String`) w et une liste d'association (`AList`) p.

Conception « ***top-down*** » **:** On suppose écrite une méthode `lookCell(`p`, `w`)` qui renvoie

- La cellule de liste q de la liste p, telle que q`.word` égal à w.
- Ou **null**, sinon.

On peut maintenant écrire recherche et mise à jour (`get`/`put`),

Recherche

Écrivons une méthode `get` *statique* (Pourquoi ? **null** est une liste).

- `get` prend une `AList` et un mot w et renvoie le compte.
- Trouver la cellule q, avec q`.word` égal à w.
- Si q est **null**, renvoyer `0`.
- Sinon, renvoyer q`.count`

```
class AList {
    ...
  static int get(AList p, String w) {
    AList q = lookCell(p, w) ;
    if (q == null) return 0
    else return q.count ;
  }
}
```

Création/Mise à jour d'une association de la liste

Écrivons une autre méthode *statique* dans la classe `AList`.

- `put` prend une `AList` p, un mot w, un compte c, et renvoie la table augmentée de la paire $(w \to c)$.
- Trouver la cellule q.
- Si q est **null**, ajouter une association.
- Sinon, modifier l'association et renvoyer la table modifiée.

```
static AList put(AList p, String w, int c) {
  AList q = lookCell(p, w) ;
  if (q == null) {
    return new AList (w, c, p) ;
  } else {
  q.count = c ;
  return p ;
}}
```

Code de `lookCell`

```
static AList lookCell(AList p, String w) {
  for ( ; p != null ; p = p.next) {
     if (w.equals(p.word)) {
       return p ;
     }
  }
  return null ; // Pas trouvé
}
```

Remarquer le **return** dans la boucle, et le **return** après la boucle.

Résumé de la classe `AList`

```
class AList {
  ⋮ // Un constructeur, la méthode lookCell
  static int get(AList p, String w) { ... }
  static AList put(AList p, String w, int count) { ... }
}
```

Un objet de la classe `AList` peut-il prendre `Assoc` comme type ?

```
interface Assoc {
  int get(String key) ;
  void put(String key, int val) ;
}
```

Non, essentiellement parce que `get` et `put` sont des méthodes statiques de la classe `AList`.

Il faut encapsuler

```
class L implements Assoc {
  private AList p ;

  L() { p = null ; }

  public int get(String w) { return AList.get(p, w) ; }

  public void put(String w, int c) { p = AList.put(p, w, c) ; }
}
```

- On demande au compilateur `javac` de vérifier qu'un objet `L` peut prendre le type `Assoc`. On dit que la classe `L` *implémente* l'interface `Assoc`.
- Les méthodes déclarées dans l'interface sont obligatoirement **public** (c'est comme ça).

Bénéfice

On peut maintenant écrire :

```
class Freq {
  ...
  public static void main(String [] arg) {
    countFile(arg[0], new L()) ;
  }
}
```

Deux programmeurs peuvent travailler indépendamment (l'un écrit `Freq`, l'autre écrit `L`).

1. Leurs classes sont est compilables.
2. L'interface les oblige à préciser leurs idées.

Mais La classe `L` n'est pas très efficace.

- Pour N mots distincts $\rightarrow O(N^2)$.

Autre bénéfice (de l'interface)

Nous pouvons fabriquer une autre implémentation (`H`) de l'interface `Assoc` (plus efficace que `L`), et l'utiliser à la place, *sans changer `Freq`*.

Un programme de test des diverses implémentations des tables d'association :

```
// java Freq -L file -> liste d'association
// java Freq -H file -> autre implémentation
class Freq {
  ...
  public static void main(String arg) {
    if (arg[0].equals("-L")) countFile(arg[1], new L()) ;
    else if (arg[0].equals("-H")) countFile(arg[1], new H()) ;
    ...
  }
}
```

Détour : une vue des tableaux

Nous connaissons les tableaux :

- Accès : `t[`k`]`
- Mise à jour : `t[`k`] = ...`

Nous pouvons donc voir un tableau comme une table d'association dont les clés sont des **int** dans `[0...t.length[`.

Limitations :

- Les *clefs* k sont nécessairement des entiers,
- pris dans un intervalle donné.

Idée du hachage

Soit une clé quelconque k (pour les mots k est un `String`).

- Se donner une fonction h de *hachage* des clés vers les entiers.
- Si...
  - Le co-domaine de h est de la forme $[0 \dots m[$,
  - Et h injective ($h(k_1) = h(k_2) \Rightarrow k_1 = k_2$)
- Alors on peut employer un tableau de taille m comme table d'association.

Hachage en théorie

- Transformer les clés en entiers, facile. Par exemple :

  Une chaîne w est vue comme l'écriture en base 2^{16} d'un entier.

- Dans l'intervalle $[0 \dots m[$, facile.

  Prendre le reste de la division euclidienne par m.

- Fonction de hachage injective, difficile, et surtout contradictoire avec le point précédent.

Lorsque $h(k_1) = h(k_2)$ pour $k_1 \neq k_2$, on dit qu'il y a une *collision*.

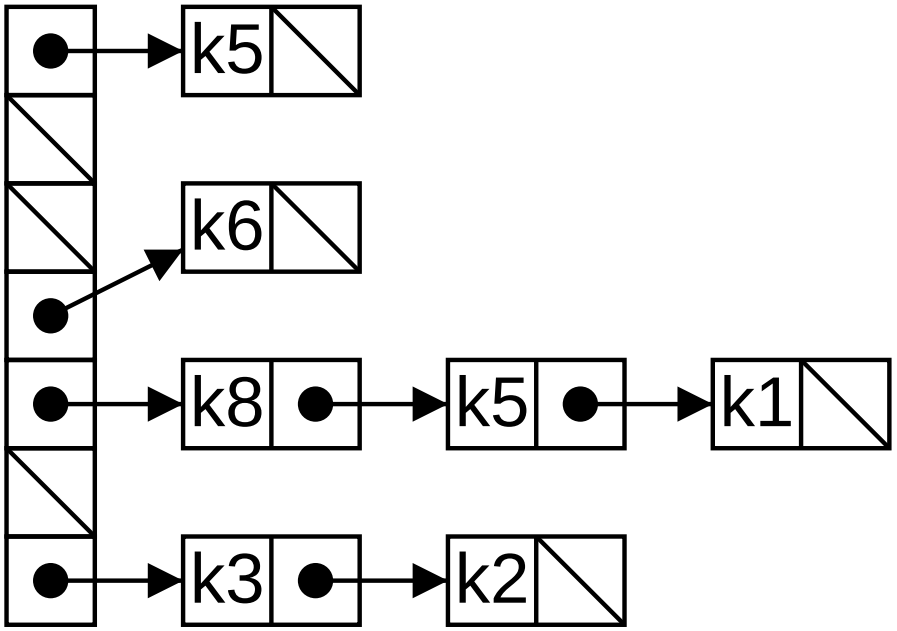
Si il n'y a pas de collisions, le hachage est *parfait*.

Idée du hachage II, résolution des collisions

On suppose donc des collisions :

$$\exists k\, k', k \neq k' \text{ et } h(k) = h(k')$$

Le chaînage : la case i du tableau t contient la liste des informations $(k \to v)$ avec $h(k) = i$.



Implémentation

- Bon à remarquer : les « listes d'informations $(k \to v)$ » sont des listes `AList`.
- Bon à savoir : java fournit une méthode de hachage, ici la méthode **int** `hashCode()` des `String`.

$$h(w) = w\texttt{.hashCode()} \bmod m$$

```
class H implements Assoc {
  private final static int SZ = 1024 ;
  private AList [] t ;
  H() { t = new AList [SZ] ; }

  // Fonction de hachage h.
  private int hash(String w) {
    return Math.abs(w.hashCode()) % t.length ;
  }
```

Chercher dans bonne liste

```
public int get(String w) { // Chercher dans t[h(w)]
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (q == null)
    return 0 ;
  else
    return q.count ;
}
```

Ajouter/Remplacer dans la bonne liste

```
public void put(String w, int c) {
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (r == null)
    t[i] = new AList(w, c, t[i]) ;
  else
    q.count = c ;
}
```

Coût du hachage

On se donne quelques hypothèses.

- Le coût du calcul de h est en $O(1)$ (ou négligé).
- Le hachage est uniforme :

  Pour une table de taille m, $h(w)$ vaut i dans $[0 \dots m[$ avec une probabilité $1/m$.

Alors, le coût *moyen* d'une recherche/ajout dans une table contenant n associations est en $O(1 + n/m)$ (admis).

Donc si m est de l'ordre de n : $O(1)$.

On note $\alpha = n/m$ le facteur de charge, c'est aussi la longueur moyenne des listes de collision (intuitivement clair ?).

Remarques sur le coût en $O(1)$

- m de l'ordre de n veut aussi dire : facteur de charge α borné par une constante.
- Le coût dans le cas le pire est $O(n)$ (toutes les clés sont en collision).
- Mais adopter le hachage c'est :
  - Faire confiance au hasard (hachage uniforme).
  - Procéder à beaucoup de de `put`/`get` (coût en moyenne significatif du coût en pratique).

    Produire les statistiques d'un texte de N mots :

$$N \times O(1) \sim O(N)$$

Redimensionnement

Ce qui était vrai des piles/files l'est aussi des tables de hachages.

Il est pratique et normal que ce soit le code de la table de hachage qui se charge de maintenir α borné.

```
private final static double alpha = 4.0 ; // borne sur la charge
private int nbKeys = 0 ; // n (m est t.length)

public void put(String w, int c) {
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (r == null) {
    t[i] = new AList(w, c, t[i]) ;
    nbKeys++ ; // Car vient d'ajouter une association
    if (t.length * alpha <= nbKeys) // La charge est trop forte
      resize() ;
  } else
    q.count = c ;
}
```

La méthode `resize`

Pour assurer un coût amorti en $O(1)$

- Progression géométrique de la taille du tableau.
- Coût de `resize` proportionnel à la taille du tableau.

Et voilà.

```
private void resize() {
  AList [] old = t ;

  t = new AList [2*old.length] ; // Remplacer d'abord
  for (int i = 0 ; i < old.length ; i++) { // Copier old -> t
    for (AList p = old[i] ; p != null ; p = p.next) {
      int h = hash(p,word) ;
      t[h] = new AList (p.word, p.count, t[h]) ;
    }
  }
}
```

Bibliothèque

Les tables de hachage sont d'un emploi très courant.

La classe de la bibliothèque est générique à deux paramètres :
`HashMap <K,V>` (package `java.util`).

- `K` classe des clés (pour nous `String`).
- `V` classe des valeurs (pour nous `Integer`).

Le codage est des plus facile.

```
import java.util.* ;

class Lib implements Assoc {
  private HashMap <String,Integer> t ;
  Lib() { t = new HashMap <String,Integer> () ; }
  ...
}
```

Chercher, remplacer/ajouter dans un HashMap

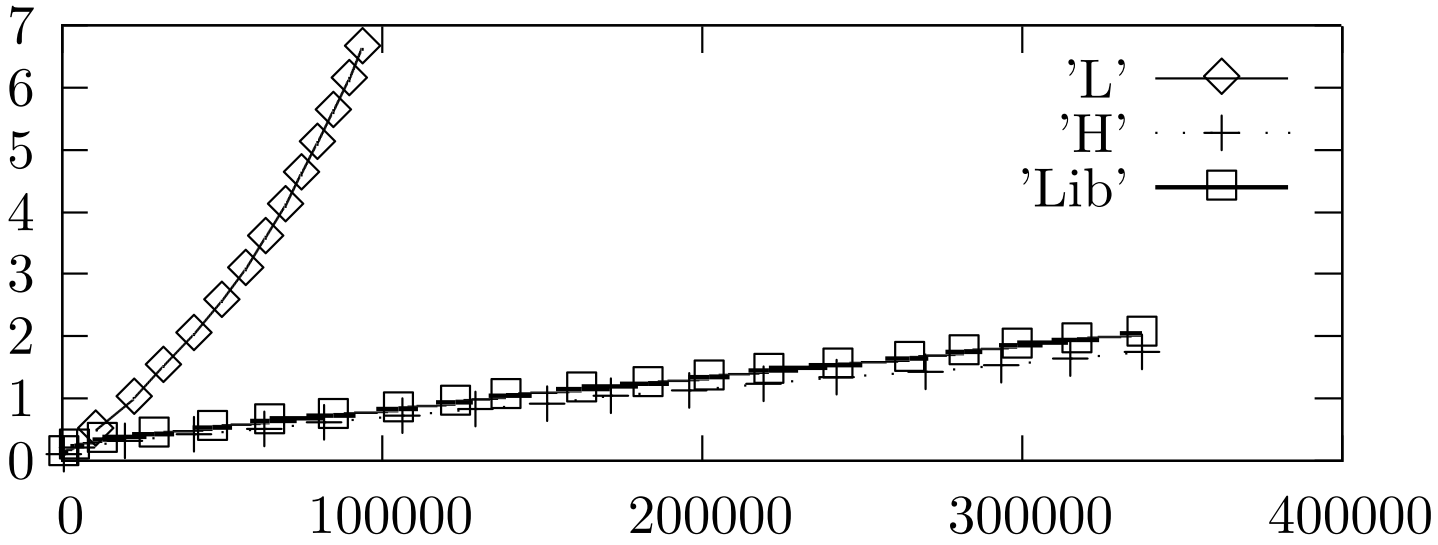
```
public int get(String w) {
 Integer c = t.get(w) ;
 if (c == null) // Si pas d'assoc (w → c) t.get(w) -> null
   return 0 ;
 else
   return c ; // Compris comme 'return c.intValue()'
}

public void put(String w, int c) {
 t.put(w, c) ; // ...c....  -> ... Integer.valueOf(c) ...
}
```

Et c'est tout !

Performance

Faire la statistique des mots de sources Java. Temps d'exécution de `count` (sec.), fonction du nombre de mots lus.



On note :

- Comportement supra-linéaire des listes.
- Comportement linéaire des tables de hachage.

Et au cas où vous vous poseriez la question

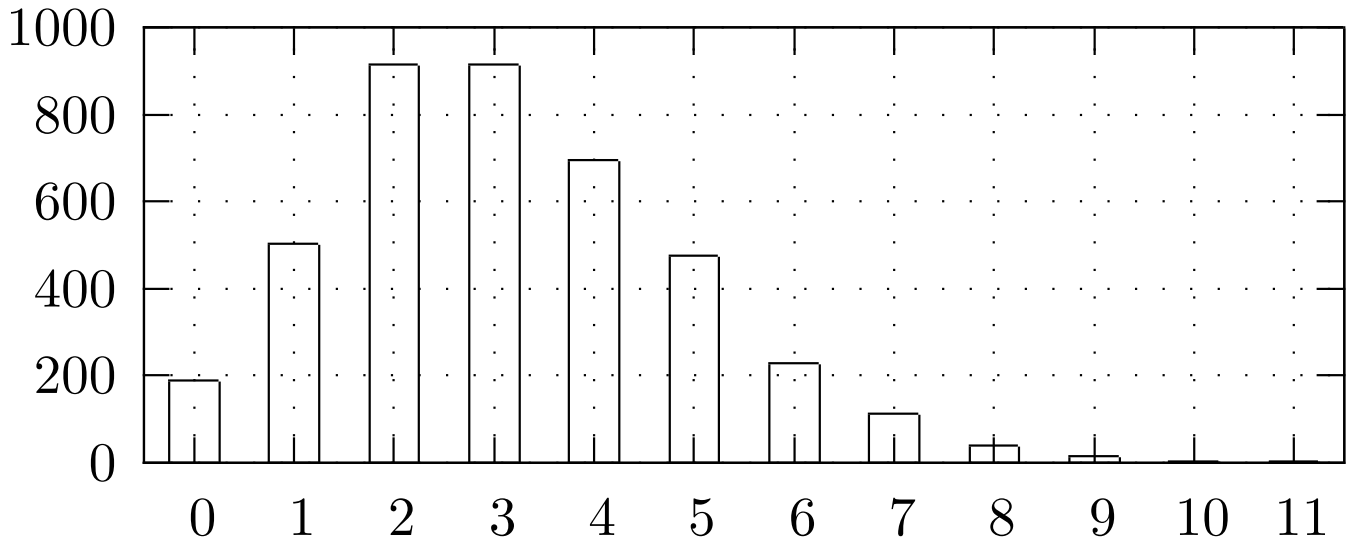
Statistique des sources Java

```
return: 13221       private: 4216       abnorme: 1
static: 9666                            aborted: 1
null: 8191            etc.
string: 6672                              etc.
system: 6526        abandoned: 1
void: 6007          abba: 1             zhoulai: 1
public: 5464        abbcdefgh: 1        ziegler: 1
else: 5376          ability: 1          zones: 1
println: 4650       abnormally: 1       zyvabis: 1
```

Bilan des collisions

L'efficacité provient principalement du « bon » hachage des clés.

Effectifs des listes de collisions, selon leur taille. Pour $n = 12904$, $m = 4096$, $\alpha = 3.15$.



Il a par ex. environ 900 (sur 4096) listes de collisions de taille 2.

Autre exemple : linguistique récréative

Un texte sur le hachage :

> Pour des clés plus générales cette façon de mélanger est critiquée, mais nous allons écrire une méthode count, qui compte le nombre d'occurrences des mots d'un texte, dans le cas du hachage modulo un nombre premier, il reste possible de calculer modulo m. Mais en fait ce n'est pas divisible par m, pour des entiers successifs, un cas aussi simple est exceptionnel en pratique.

Ce texte est « inspiré » du chapitre du poly sur les tables de hachage.

Il a été produit automatiquement, avec... une table de hachage.

Un texte qui ressemble à un autre

On suppose que l'enchaînement des mots d'un texte suit un modèle de Markov.

Soit en simplifiant : deux mots w_1 puis w_2 déterminent le mot suivant w_3 (parmi un ensemble de mots possibles).

Idée :

- Produire automatiquement un texte T' « vraisemblable ».
- Il nous faut un *corpus* (un texte T écrit par un humain).
- S'assurer que toutes les sous-suites de trois mots de T' sont des sous-suites de T.

Un algorithme

1. w_1 et w_2 sont les deux premiers mots de T.
2. Afficher w_1 puis w_2.
3. Boucle :
   (a) Choisir un w_3 qui suit w_1–w_2 dans T.
   (b) Si w_3 est « *(fin)* », terminer.
   (c) Afficher w_3.
   (d) Remplacer w_1 par w_2, puis w_2 par w_3.

Les suites de deux mots

Objets de la classe `Prefix`.

```
class Prefix {
  private String w1, w2 ;
  Prefix (String w1, String w2) {
    this.w1 = w1 ; this.w2 = w2 ;
  }

  Prefix shift(String w3) {  // (w1, w2) → (w2, w3)
    return new Prefix (w2, w3) ;
  }
}
```

Programmation

```
static void chain() {
  Prefix pref = ...
  for ( ; ; ) { // Boucle
    String w3 = step(pref)     // Choisir le troisième mot.
    if (w3 == null) return ;   // pref = deux derniers mots.
    System.out.print(w3 + " ") ;
    pref = pref.shift(w3) ;
  }
}
```

Une seule difficulté : comment choisir le troisième mot efficacement.

Choisir le troisième mot

Il nous fait une association des préfixes de deux mots de T, vers les mots suivants.

À un préfixe donné peut correspondre plus d'un mot suivant.

`[...] des clés distinctes [...] des clés successives [...]`

Donc association des objets `Prefix` vers par ex. les tableaux de `String`.

En Java.

```
static HashMap<Prefix,String []> h =
                new HashMap<Prefix,String []> () ;
```

Par la suite on suppose que la table est correctement remplie (lecture de T mot-à-mot et `h.put`).

Programmation de `step`

Très facile, avec la table de hachage.

```
static HashMap<Prefix,String []> h =
  new HashMap<Prefix,String []> () ;
static Random rand = new Random () ; // Source pseudo-aléatoire

static String step(Prefix pref) {
  String[] t = h.get(pref) ;
  return t[rand.nextInt(t.length)] ;
}
```

Pour `Random` voir la doc[a].

[a] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Random.html`

Code `get` de la bibliothèque

Schématiquement :

```
private V[] t = ... ; // tables de listes de collision.

public V get(K key) {
  int h = Math.abs(key.hashCode()) % t.length ;
  for (AList q = t[h] ; q != null ; q = q.next) {
    if (key.equals(q.key)) return q.val ;
  }
  return null ;
}
```

Méthodes `hashCode` et `equals` de nos clés (classe `Prefix`) ?

Obscur `Object`

- Tout objet (*i.e.* valeur crée par **new** `Class (...)`) peut être vu comme de type `Object`. C'est le *sous-typage*.
- À la naissance, tout objet possède déjà les méthodes définies par la classe `Object`. C'est *l'héritage*.

Exemple de méthode ainsi (pré)-définie pour tous les objets ?

- La méthode `toString`. Mais aussi...
- Les méthodes (publiques) **boolean** `equals(Object o)` et **int** `hashCode()`, bien utiles pour... les tables de hachage,
  - `hashCode` pour trouver la bonne liste de collisions.
  - `equals` pour chercher dedans.

Il faut redéfinir les méthodes

Les méthodes héritées ne nous conviennent pas.

C'est la la même histoire que `toString` (amphi 01)

- Si *o* est un objet, le code `out.print(`*o*`)`, revient à afficher la chaîne renvoyée par l'appel de méthode *o*`.toString()`.
- Par défaut, *o*`.toString()` affiche la valeur de *o* (une flèche).

  `out.print(`**new** `Prefix(`*"des"*`,`*"clés"*`))` ⇒ `Prefix@6f0472`
- On *redéfinit* la méthode `toString` (dans la classe `Prefix`).

  public `String toString() {` **return** `w1 + " " + w2 ; }`
- Et alors

  `out.print(`**new** `Prefix(`*"des"*`,`*"clés"*`)` ⇒ `des clés`

Méthode `equals` livrée d'origine

Sa *signature* (type) :

public boolean `equals(Object obj)`

Sa *sémantique* (ce qu'elle fait) :

> The equals method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns **true** if and only if `x` and `y` refer to the same object (`x == y` has the value **true**).

En résumé :

new `Prefix("a","b").equals(`**new** `Prefix("a","b"))` $\Rightarrow$ **false**

Nous voulons l'égalité structurelle.

Redéfinition de `equals`

Premier essai (classe `Prefix`).

```
boolean equals(Prefix p) {
  return
    this.w1.equals(p.w1) && this.w2.equals(p.w2) ;
}
```

S'agit-il d'une *redéfinition* de **`equals`** des `Object` ?

Non ! Pourquoi ?

- La signature de **`equals`** des objets est

  public boolean `equals (Object o)`

  Et surcharge (*overloading*).

- Notre `Prefix` possède maintenant deux méthodes **`equals`** : `equals(Prefix p)` (ci-dessus) et `equals(Object o)` (héritée).

Redéfinition de `equals (Object o)`

Deuxième essai (classe `Prefix`).

```
public boolean equals(Object o) {
  return
    this.w1.equals(o.w1) && this.w2.equals(o.w2) ;
  // NB: w1.equals(..) et w2.equals(..), equals des String
}
```

Est-ce que ça fonctionne ? Non ! Que se passe-t-il ?

Refus de compiler la classe `Prefix`.

```
% javac Prefix.java
Prefix.java:15: cannot resolve symbol
symbol  : variable w1
location: class java.lang.Object
    this.w1.equals(o.w1) && ... ;                    ^
```

Redéfinition de `equals (Object o)`

Dans l'essai précédent, le compilateur n'a aucun moyen de savoir que l'`Object` que l'on lui donne est toujours un `Prefix`.

Il faut le lui dire !

```
public boolean equals(Object o) {
 // downcast (conversion de type vers le bas)
  Prefix p = (Prefix)o ;
  return
    this.w1.equals(p.w1) && this.w2.equals(p.w2) ;
}
```

Que se passe-t-il avec p.`equals("coucou")` ?

Un échec (à l'exécution) ! Mais plus précisément ?

L'exception `java.lang.ClassCastException` est levée.

Redéfinition de `hashCode ()`

Il s'agit produire un **int** à partir de `w1` et `w2` (deux `String`).

```
public int hashCode() {
// Un calcul à partir de this.w1 et this.w2
}
```

Pour garantir le bon fonctionnement de la table de hachage, il faut :

p_1`.equals(`p_2`)` entraîne p_1`.hashCode() ==` p_2`.hashCode()`

Le mélangeur multiplicatif (par un nombre premier).

```
public int hashCode() {
  return
    37 * this.w1.hashCode() + this.w2.hashCode() ;
}
```

Inspiré du hachage des chaînes de Java.

Une bonne fonction de hachage

- En théorie,
  - Tout est entier.
  - Uniforme sur l'espace des clés.
  - Rapide à calculer.
- En pratique.
  - Oui, mais de grand entiers.
  - Répartition uniforme des clés effectivement rencontrées.
    - ★ Utilise bien tout l'intervalle $[0 \ldots m[$ (surjection).
    - ★ Opère un mélange de *toutes* les informations contenues dans la clé.
    - ★ Une petite modification de la clé entraîne une grosse modification du hachage (analyse + tests).
  - Rapide à calculer.

Conclusion, table de hachage

Les tables de hachage sont des tableaux généralisés.

- Avec des indices arbitraires.

$$\begin{array}{rcl} i \in [0 \ldots m[& \Rightarrow & k \in K \\ \texttt{t[}i\texttt{]} & \Rightarrow & \texttt{t.get(}k\texttt{)} \\ \texttt{t[}i\texttt{] = }v & \Rightarrow & \texttt{t.put(}k\texttt{,}v\texttt{)} \end{array}$$

- Coût asymptotique identique en $O(1)$, *en pratique*, sous-reserve de bon choix de la fonction de hachage.
- Une différence : l'occupation de la mémoire.
  - Table de hachage (avec α borné et `resize`) : proportionnel au nombre de clés.
  - Tableau : plus grande clé.

  Structure creuse *vs* structure dense.

Bonus interface

`put` et `get` c'est quand même un peu juste.

Comment effectuer la dernière étape du programme `Freq` avec les tables de hachage de la bibliothèque ?

3. Produire un bel affichage.

Une solution un peu plus détaillée.

- Récupérer toutes les associations $(w \to c)$ dans une liste.
- Trier la liste (amphi 02)
- Afficher la liste triée (jugé évident).

Récupérer l'ensemble des clés d'une table

C'est ce que fait la méthode `keySet` des `HashMap<`K`,`V`>`).

public `Set<`K`> keySet()`

Itérer sur cet ensemble, pour construire la liste, magique :

```
// t est la table de bibliothèque HashMap<String,Integer>
  Set<String> keys = t.keySet() ;
  AList r = null ;
  for (String k : keys) {
    r = new AList (k, t.get(k), r) ;
  }
// r contient la liste des (w → c)
```

Au delà de la magie

Une interface souvent utilisée : `Iterator<E>` (package `java.util`, doc[a]), un itérateur sur une « *collection* » de `E`.

Deux méthodes intéressantes :

- Reste-t-il un élément ? **boolean** `hasNext()`
- Extraire l'élément suivant : `E next()`

Usage :

```
Iterator<E> it = ...
while (it.hasNext()) {
  E e = it.next() ;
  // Traiter e
}
```

[a] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html`

Au delà de la magie II

De nombreuses classes de la bibliothèque (par exemple `Set<E>`) implémentent l'interface `Iterable<E>` (package `java.lang`, doc[a])

Une seule méthode :

- Récupérer un itérateur `Iterable<E> iterator()`

Usage :

```
Iterable<E> es = ...
Iterator<E> it = es.iterator() ;
while (it.hasNext()) { E e = it.next() ; ... }
```

Notation abrégée :

```
Iterable<E> es = ...
for (E e : es) {
  ...
}
```

[a] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Iterable.html`

Au delà de la magie III

Et donc :

```
Set<String> keys = ... ;
for (String k : keys)
  r = new AList (k, t.get(k), r) ;
```

Est traduit par le compilateur en :

```
   Set<String> keys = ... ;
// keys implémente Iterable<String>
  {
     Iterator<String> it = keys.iterator() ;
// it est un itérateur sur les elts de keys
     while (it.hasNext()) {
        String k = it.next() ;
        r = new AList (k, t.get(k), r) ;
     }
  }
```

Bonus magique

Tout ce passe comme si un tableau de type T `[]` implémentait l'interface `Iterable`. Plus directement,

```
T [] t = ... ;
for (T v : t) {
  ...
}
```

Est l'expression succinte de :

```
T [] t = ... ;
for (int k = 0 ; k < t.length ; k++) {
  ...
}
```

- TP exemple de table de hachage.
- Prochaine fois les arbres (et contrôle machine).