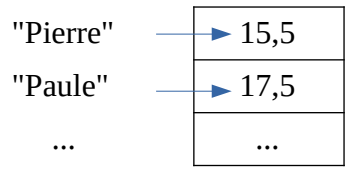
Fiche résumé nº 3 – Les tables

1. Définition et utilisation

a) Définition

Une table est une application finie d'un ensemble C vers V
- C représente l'ensemble des clés
- V représente les valeurs

Ainsi, une table peut être considérée comme un ensemble de doublets (clé,valeur).
Par conséquent, on peut voir dans une table une généralisation des tableaux où l'indice n'est plus forcément un entier :



Fonctions du TAD Table :

```
sorte Table[Clé,Valeur]
utilise Booléen, Entier
fonctions
    new :        ⟶ Table[Clé,Valeur]
    taille :     Table[Clé,Valeur] ⟶ Entier
    val :        Table[Clé,Valeur]×Clé ⇸ Valeur
    ajouter :    Table[Clé,Valeur]×Clé×Valeur ⟶ Table[Clé,Valeur]
    supprimer :  Table[Clé,Valeur]×Clé ⟶ Table[Clé,Valeur]
    contient :   Table[Clé,Valeur]×Clé ⟶ Booléen

préconditions
    ∀t de type Table[Clé,Valeur], ∀c de type Clé
    pré(val(t,c)) = contient(t,c)
```

Pour mettre en œuvre ce type, le *framework* des collections Java contient ainsi l'interface Map et 4 implémentations concrètes :

1. **`HashMap`**
   le cas de table sur lequel nous allons nous concentrer.
2. **`WeakHashMap`**
   table exploitant le concept de références faibles, lui-même lié au aux références douces.
3. **`TreeMap`**
   table exploitant une relation d'ordre permettant de trier les clés (cf. la méthode compareTo(.) des éléments de type Comparable, ou encore les objets de type Comparator).
4. **`Hashtable`**
   classe d'origine antérieure au *framework*, désormais obsolète.

b) Méthodes principales de l'interface générique Map<K,V> de Java

- `V put(K key, V val)` : ajoute la nouvelle association (key,val) à la table (en remplaçant l'ancienne association s'il y en avait une). Si la clé existait déjà, la valeur antérieure est renvoyée (sinon null).
- `V get(K key)` : fournit la valeur associée à la clé key si elle existe (sinon null).
- `boolean containsKey(K key)` : indique si une association de clé key est présente.
- `V remove(K key)` : supprime l'association de clé key si elle existe et retourne sa valeur (sinon null).
- `void clear()` : supprime toutes les associations de la table.
- `int size()` : indique le nombre d'associations de la table.
- `boolean isEmpty()` : indique si la table est vide.
- ...

N.B. isEmpty() est un cas particulier de size(), mais containsKey() est absolument nécessaire car recevoir le résultat null par get() ne prouve pas que la clé est absente : elle pourrait être présente et associée à la valeur null.

c) Itérations sur les tables Java

En Java (contrairement au choix d'autres langages) une Map n'est pas une Collection (cf. architecture du *framework* des collections Java vu précédemment) : elle ne fournit pas d'itérateur et n'est pas itérable !

On ne peut donc pas itérer directement sur une table, mais on peut en obtenir une vue particulière sous forme de collection Java itérable, parmi les trois *Collection Views* :

- `Set<K> keySet()` : les clés
- `Collection<V> values()` : les valeurs
- `Set<Map.Entry<K,V>> entrySet()` : les doublets
  - ces doublets sont modélisés par l'interface `Map.Entry` qui propose quelques méthodes utiles (`getKey`, `getValue`, `setValue`, `equals`, …)

N.B. On remarque que, vu son nom, `Map.Entry` est donc une interface `Entry` définie en interne à l'interface `Map` (et publique).

d) Exemples d'utilisation

* ***Exemple 1*** : en supposant qu'on ne recherche pas la clé null dans la table, réécrire une mise en œuvre externe à la classe pour fournir le même résultat que containsKey.

```
boolean mapContainsKey(Map<K,V> m, K clé) {

    Set<K> s = m.keySet();
    Iterator<K> i = s.iterator();
    while (i.hasNext()) {
        K key = i.next();
        if (clé.equals(key)) return true;
    }
    return false;
}
```

En réalité la mise en œuvre Java utilise la méthode hashCode des clés (cf. plus loin) pour accélérer les comparaisons. Il faut donc faire attention à cette méthode, pour que les clés soient retrouvées.

N.B. Cette écriture in extenso a pour but pédagogique d'expliciter l'itération qui est mise en jeu. Car grâce aux raccourcis d'écritures apparus en Java, on peut exprimer cet algorithme de manière plus concise : en 3 lignes seulement (sans compter les accolades).

* ***Exemple 2*** : supprimer les exemplaires supplémentaires en doublons dans la totalité d'une table.

Cet exemple montre comment contourner l'impossibilité (logique et raisonnable) de supprimer un élément de collection s'il y a une itération en cours sur cette collection, ou de supprimer un élément de table s'il y a une itération en cours sur l'une de ses vue collection : si on faisait ici 2 boucles imbriquées contenant un remove(.), on provoquerait une **`ConcurrentModificationException`**.

```
void supprimerDoublons(Map<K,V> m) {
    // Astuce nº 1, noter pour supprimer plus tard :
    Collection<K> clesASupprimer = new HashSet<K>();
    // Astuce nº 2, plus de doublons dans un ensemble :
    Collection<V> valeurs = new HashSet<V>(m.values());

    for (V val : valeurs) {
        recenserDoublons(m, val, clesASupprimer);
    }
    for (K clé : clesASupprimer) {
        m.remove(clé);
    }
}

void recenserDoublons(Map<K,V> m, V val, Collection<K> clés) {
    // Trouve toutes les occurrences de val et note celles en trop via leur clé
    int nb = 0;
    for (Map.Entry<K,V> doublet : m.entrySet()) {
        if (doublet.getValue().equals(val)) {
            nb++;
            if (nb>1) clés.add(doublet.getKey());
        }
    }
}
```

2. Mise en œuvre par hachage

a) Principe

Un « simple » calcul sur la clé (par l'évaluation d'une fonction, dite de hachage et opportunément souvent notée *h*(clé)) va permettre d'indiquer le rang (la position) de la valeur associée en mémoire.

Selon que cette fonction *h* est injective ou non, 2 cas très différents se présentent.

b) Cas d'une fonction injective[1]

(donc 2 clés distinctes vont donner 2 positions distinctes)
L'adressage est dit calculé.

Un simple schéma permet de constater aisément les avantages et inconvénients.

- AVANTAGES : l'adressage est direct (localisation facile et efficace des valeurs).

- INCONVÉNIENTS : si de nombreuses de clés sont possibles, alors cela nécessite un vaste espace pour chaque position de valeurs possibles. Et si peu de ces clés possibles sont effectivement exploitées, alors il y aura beaucoup de mémoire gâchée (des « trous » inutilisés partout). De plus, bien sûr, cette méthode ne peut être exploitée si le nombre de clés possibles différentes est infini.

c) Cas d'une fonction non injective

(surjective, et même quelconque : 2 clés différentes peuvent donner une même position)
L'adressage est dit dispersé.

- AVANTAGES : cette fois on va économiser la mémoire, rendant la table d'usage raisonnable et même quand le nombre de clés possibles est infini.

1 Une fonction est injective quand $\forall x\ \forall y, (f(x)=f(y)) \Rightarrow (x=y)$.

- INCONVÉNIENTS : lorsque 2 clés engendrent la même position, il va falloir gérer ces conflits (collisions), et cette gestion des collisions peut dégrader la rapidité d'accès. Idéalement il faut avoir une fonction qui optimise un compromis : minimiser les collisions (classes d'équivalences équilibrées en fonction des clés) et maximiser l'occupation de la mémoire allouée (bien disperser les clés).

En cas de survenue de conflit (2 clés veulent stocker leur valeur associée au même endroit), les HashMap de Java recherchent tout simplement la prochaine position non occupée => essais linéaires induisant un coût en $O(n)$. Pour grossir, mais aussi pour éviter de tomber dans un état où trop de conflits vont survenir car l'espace mémoire devient trop rempli, elles respectent un facteur de charge à partir duquel elles augmentent leur espace mémoire occupé pour diminuer les collisions. D'où les constructeurs suivants dans la Javadoc :

- `HashMap()` : *constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75)*.
- `HashMap(int initialCapacity)` : *constructs an empty HashMap with the specified initial capacity and the default load factor (0.75)*.
- `HashMap(int initialCapacity, float loadFactor)` : *constructs an empty HashMap with the specified initial capacity and load factor*.

N.B. Il existe un 4e constructeur pour recopier une Map quelconque dans une HashMap :
- `HashMap(Map<? extends K,? extends V> m)` : *constructs a new HashMap with the same mappings as the specified Map.*

Notez l'usage des *bounded wildcards* nécessaires pour généraliser les utilisations possibles.

D'autres stratégies de résolution des collisions sont possibles :
- chaînage du débordement (listes pour stocker plus d'une valeur à une position)
- hachage en cascade (sous-tables de hachage, selon une autre fonction de hachage, aux conflits).
- etc.

3. Exemples et hashCode de Java

*** Exemple simple d'adressage calculé**
Considérons des étudiants INSA servant de clés pour accéder à leur fiche d'informations dans une table à la scolarité de l'établissement.
Chacune des spécialités comporte 3 années qui comptent 72 étudiants au maximum (soit 216).
Un étudiant est donc caractérisé par son département $d \in [\![1;7]\!]$, sa promotion $p \in [\![1;3]\!]$ et son rang alphabétique dans la promotion $n \in [\![1;72]\!]$:

$$h(\textit{étudiant}) = (d-1)\times 216 + (p-1)\times 72 + (n-1)$$

N.B. L'idée est tout simplement de préréserver un numéro d'emplacement pour tous les cas possibles, même si l'étudiant n'existe pas (par exemple le 70e étudiant d'une promo de 48).
- Donc on confirme la présence de beaucoup de « trous » pour toutes les promo < 72.
- Pire, si on rajoute le STPI alors on démultiplie les trous : seulement 2 années occupées, mais environ 300 étudiants à prévoir par année au lieu de 72.

*** Exemple simple d'adressage dispersé**
Considérons des mots constitués de lettres majuscules servant de clés.

Solution 1 : se baser sur l'initiale du mot pour « prédire » la position en mémoire.

$$h_1(mot) = (\text{int})\, mot.charAt(0) - (\text{int})\text{'A'}$$

N.B. On retranche le code de la lettre A au code de la l'initiale pour obtenir $h_1 \in [\![0 ; 25]\!]$

Remarques :

- On comprend que cette première approche provoque des collisions et donc des conflits à régler pour chaque mot commençant par la même lettre.
- On comprend aussi que cette approche n'optimise pas du tout l'équité des conflits en fonction des clés : idéalement chaque classe d'équivalence devrait concerner 1/26 de toutes les clés possibles, mais on sait bien que statistiquement il y a plus de mots commençant par certaines lettres que d'autres.

Solution 2 : on peut conserver un espace similaire pour la table, mais mieux disperser les clés de manière « aléatoire » et ainsi mieux les répartir un peu partout en employant toutes les lettres.

$$h_2(mot) = \left(\sum_{k}^{mot.length()-1} (\text{int}) mot.charAt(k) \right) \bmod p$$

N.B. La somme fournit un nombre quelconque et le modulo ramène dans l'espace dimensionné par p, qui peut être 26 comme dans le cas précédent ou une autre valeur.

Solution généralisée : à partir de l'exemple précédent découle la méthode dite de la division pour passer d'une clé à une position dans la table.

$$h(clé) \ : \ clé \underbrace{\longrightarrow}_{\text{(1) lié à la clé}} nombre \underbrace{\longrightarrow}_{\text{(2) lié à la table}} adresse\,dans\,la\,table$$

C'est l'approche en 2 étapes retenue par Java dans son *framework* :

1. la classe Object fournit la méthode hashCode (qui renvoie simplement la référence, à mieux redéfinir dans tout objet servant de clé).
   - C'est par exemple le calcul de la somme dans h_2 plus haut.
2. Ce hashCode est exploité dans les HashMap par simple opération modulo p (reste de la *division* par la taille p de la table) pour indiquer une case de la table parmi celles possibles.
   - C'est le modulo dans h_2 pour adapter l'amplitude de la somme à la taille de la table.

N.B. Les méthodes hashCode et equals doivent être redéfinies en cohérence dans une classe : 2 instances considérées identiques par equals doivent fournir la même valeur par hashCode.

Pour éviter des phénomènes d'accumulation concentrant les clés vers les mêmes positions, il ne faut pas prendre p quelconque :

- par exemple un modulo 1000 ne prendrait en compte que les 3 derniers chiffres de le représentation en base 10 fournie par la clé
- de même un modulo 2^n isolerait les n bits de poids faible de la représentation binaire fournie par la clé

=> les études mathématiques du hachage indiquent qu'il faut privilégier un nombre premier pour p.

N.B. Même si cette méthode de la *division* est très répandue, car simple et performante, il n'y a pas de stratégie universelle et d'autres approches peuvent souvent se rencontrer :

- *extraction de bits* : partant de la représentation binaire de la clé, choisir p bits pour fournir 2^p combinaisons, soit une valeur entière de 0 à 2^p-1.
- *compression de bits* : la représentation binaire de la clé est découpée en n tanches de p bits, que l'on peut combiner par différentes opérations, souvent le *ou-exclusif.*
- *multiplication* : multiplier la représentation binaire de la clé par un un réel r, conserver la partie décimale (donc < 1) puis multiplier par la taille p et prendre la partie entière.