Chapitre 3: Tables

Christophe Morvan

Université Paris-est, Marne-la-Vallée

20 octobre 2015

Plan

Progression

Premiers exemples

Observation

On souhaite parfois stocker des couples (des tuples) de valeurs :

- Nom/Numéro de téléphone
- Mot/Liste d'occurrences
- Ressource/Quantité

Premiers exemples

Observation

On souhaite parfois stocker des couples (des tuples) de valeurs :

- Nom/Numéro de téléphone
- Mot/Liste d'occurrences
- Ressource/Quantité

Un tableau/une liste de couples n'est pas satisfaisant... Pourquoi ?

Premiers exemples

Observation

On souhaite parfois stocker des couples (des tuples) de valeurs :

- Nom/Numéro de téléphone
- Mot/Liste d'occurrences
- Ressource/Quantité

Un tableau/une liste de couples n'est pas satisfaisant... Pourquoi ?

Table d'association

Construire des associations entre des clés et des valeurs

Copies	$\rightarrow$	220
Brouillons jaunes	$\rightarrow$	150
Brouillons bleus	$\rightarrow$	135

Hachage - 1

Objectif

Le but est de transformer toute entrée en un entier
Idéalement de façon injective (deux clés distinctes doivent être envoyées sur deux entiers distincts

Hachage - 1

Objectif

Le but est de transformer toute entrée en un entier
Idéalement de façon injective (deux clés distinctes doivent être envoyées sur deux entiers distincts

Exemple

Considérons un ensemble arbitraire et fixé de 31 items (mots), et les entiers de 0 à 40
Il est possible de définir *manuellement* une correspondance entre les items et 31 entiers
Cependant on peut observer que il y a $41^{31} \approx 10^{50}$ fonctions, parmi lesquelles seulement $41!/10! \approx 10^{43}$ injections ($10 = 41 - 31$)
Soit 1 sur 10 000 000

Hachage - 2

Observation

La probabilité d'avoir deux personnes ayant le même jour d'anniversaire dans un groupe est supérieur à 50 % pour 23 personnes.
Pour 35 personnes il est au alentours de 80 %.

Hachage - 2

Observation

La probabilité d'avoir deux personnes ayant le même jour d'anniversaire dans un groupe est supérieur à 50 % pour 23 personnes.
Pour 35 personnes il est au alentours de 80 %.

Cette observation avec l'exemple précédent induisent que :

- Les bonnes fonctions de hachage sont rares
- Et difficile à construire

Hachage - 2

Observation

La probabilité d'avoir deux personnes ayant le même jour d'anniversaire dans un groupe est supérieur à 50 % pour 23 personnes.
Pour 35 personnes il est au alentours de 80 %.

Cette observation avec l'exemple précédent induisent que :

- Les bonnes fonctions de hachage sont rares
- Et difficile à construire

Approche générale

On procède souvent en deux étapes :

- Transformation de la donnée en un entier ;
- Hachage de l'entier pour produire une clé de taille satisfaisante.

Hachage - 3

Numérisation (Souvent ad-hoc)

Soit un ensemble fini $\{u_0, u_1, \cdots, u_{s-1}\}$, de taille s.
Pour numériser des données séquentielles : $(u_{i_1}, u_{i_2}, \cdots u_{i_n})$, on a le schéma général suivant :

1. S_0 : Valeur initiale (ex : $S_0 = 0$)
2. Pour $k = 1$ à n : $S_k = f(u_{i_k}, S_{k-1})$ (ex : $S_k = (i_k.s^k) + S_{k-1}$)
3. Valeur de Hachage : $h(S_n, m)$ (ex : $h(S_n, m) = S_n \mod m$)

Hachage - 3

Numérisation (Souvent ad-hoc)

Soit un ensemble fini $\{u_0, u_1, \cdots, u_{s-1}\}$, de taille s.
Pour numériser des données séquentielles : $(u_{i_1}, u_{i_2}, \cdots u_{i_n})$, on a le schéma général suivant :

1. S_0 : Valeur initiale (ex : $S_0 = 0$)
2. Pour $k = 1$ à n : $S_k = f(u_{i_k}, S_{k-1})$ (ex : $S_k = (i_k.s^k) + S_{k-1}$)
3. Valeur de Hachage : $h(S_n, m)$ (ex : $h(S_n, m) = S_n \mod m$)

Hachage (valeur $< m$)

Option simple : $n \mod m$
Néanmoins, le choix de m influence la qualité du hachage
(exemple un m pair ne change pas la parité de la clé)
Bon choix de m : nombre premier qui possède quelques propriétés supplémentaires

Progression

Définition

Soit `K` et `V` deux types quelconques.

Définition

Un table est un ensemble de paires de la forme `(cle,valeur)`, où `cle` et `valeur` ont respecivement les types `K` et `V`.

Définition

Soit `K` et `V` deux types quelconques.

Définition

Un table est un ensemble de paires de la forme `(cle,valeur)`, où `cle` et `valeur` ont respecivement les types `K` et `V`.

En java il y a deux interfaces :

- `Map <K,V>` : pour les tables ordinaires
- `SortedMap<K,V>` : pour les tables dont les clés triables

Définition

Soit `K` et `V` deux types quelconques.

Définition

Un table est un ensemble de paires de la forme `(cle,valeur)`, où `cle` et `valeur` ont respecivement les types `K` et `V`.

En java il y a deux interfaces :

- `Map <K,V>` : pour les tables ordinaires
- `SortedMap<K,V>` : pour les tables dont les clés triables

Méthodes essentielles

- `V put(K key, V value)` : place un couple dans la table (fournit l'ancienne valeur associée à `key`)
- `boolean containsKey(Object key)` (idem pour `Value`)
- `V get(Object key)` : `null` si la clé n'est pas dans la table
- `V remove(Object key)` : supprime l'assiociation

Exemple

Exemple

```
Map<Character, Integer> compte =
    new HashMap<Character, Integer> ();
String maCh = "Bonjour cher Monsieur";
for (Character c : maCh.toCharArray()){
    Integer v = compte.get(c);
    if (v==null)v=0; // Le caractère absent
    compte.put(c, v+1);
}
for (Character c : compte.keySet())
    System.out.println("Caractère : "+c+" "
        +compte.get(c)+" fois");
```

Progression

Implémentation

L'implémentation de base des tables est `HashMap<K,V>`
Il s'agit d'une table de hachage.

Implémentation

L'implémentation de base des tables est `HashMap<K,V>`
Il s'agit d'une table de hachage.

Principe

- Une table est stockée sous forme d'un tableau de paires de `Map.Entry<K,V>`
- Pour chaque clé une valeur de hachage (inférieure à la taille du tableau) est calculée la paire est placée dans la case correspondant
- Lorsque une nouvelle clé a une valeur de hachage égale à une clé déjà présente dans le tableau une stratégie est déployée pour résoudre le problème

Implémentation (Principes)

Plusieurs choix sont fondamentaux pour implémenter une table d'association :

- Fonction de hachage
- Stratégie en cas de collision

Implémentation (Principes)

Plusieurs choix sont fondamentaux pour implémenter une table d'association :

- Fonction de hachage
- Stratégie en cas de collision

Hachage

La classe `Object` dispose de la méthode `hashCode` dont le résultat est un `int` (voir plus loin)

Implémentation (Principes)

Plusieurs choix sont fondamentaux pour implémenter une table d'association :

- Fonction de hachage
- Stratégie en cas de collision

Hachage

La classe `Object` dispose de la méthode `hashCode` dont le résultat est un `int` (voir plus loin)

Stratégie de collision

Une stratégie simple est de stocker des listes de paires dans chacune des cellules du tableau
(Autre stratégie : placer les paires dans des cases disponibles de la table – Attention lors de la suppression d'une paire)

Implémentation effective – 1

Programme exemple

```
/**
 * Classe interne
 */
private class Entry<S,T>{
        S key;
        T value;
        public Entry(S key, T value){
            this.key=key;
            this.value=value;   }
        public S getKey() {
            return key;      }
        public void setKey(S key) {
            this.key = key;   }
        public T getValue() {
            return value;      }
        public void setValue(T value) {
            this.value = value;       }
}
```

Implémentation effective – 2

Programme exemple

```
public class AssoMap <K,V> implements Map<K,V>{

     /* Nombre de cases dans le tableau
      * Nombre minimal de cases, taille totale de la table */
    private int size,minSize,nbElem;
    /*
     * Reservoir où sont stockées les "Entry"
     */
    private List<Entry<K,V>> [] bucket;
        public AssoMap(int size){
        this.size=this.minSize=size;
        this.nbElem=0;
        bucket = new LinkedList[size];
        /* On ne peut utiliser *parfaitement * les types generiques
            dans les tableaux*/  }
    public AssoMap(){
        this(16);
    }
```

Implémentation effective – 3

Programme exemple

```
public class AssoMap <K,V> implements Map<K,V>{
    public int size() {
        return this.nbElem;  }
    public boolean isEmpty() {
        return this.nbElem==0;  }
    /**
     * Calcule l'index d'une cle, selon la taille de la table
     * *Attention* : prendre la valeur absolue de la valeur de hachage. */
    private int computeIndex(Object key){
        return (Math.abs(key.hashCode()))%this.size; }

    private Entry<K,V> getEntry(Object key){
        List<Entry<K,V>> l;
        l=this.bucket[this.computeIndex(key)];
        if (l!=null){
            for(Entry<K,V> e : l){
                if (key.equals(e.getKey()))return e; } }
        return null; }
```

Implémentation effective – 4

Programme exemple

```
public boolean containsKey(Object key) {
    return this.getEntry(key)!=null; }
public V get(Object key) {
    if (this.getEntry(key)!=null)
        return this.getEntry(key).getValue();
    else return null;  }
public V put(K key, V value) {
    V res=null; Entry<K,V> tmp = getEntry(key);
    if (tmp!=null){ res=tmp.getValue();
        tmp.setValue(value);   }
    else{ this.nbElem++;
        if (nbElem >=this.size) this.enlageBucket();
        tmp=new Entry<K,V>(key,value);
        List<Entry<K,V>> ll=this.bucket[this.computeIndex(key)];
        if (ll==null){ ll= new LinkedList<Entry<K,V>>();
            this.bucket[this.computeIndex(key)]=ll; }
        ll.add(tmp); }
    return res; }
```

Implémentation effective – 5

Programme exemple

```
public class AssoMap <K,V> implements Map<K,V>{
    public V remove(Object key) {
        V res=null;
        if (this.containsKey(key)){
            this.nbElem--;
            List<Entry<K,V>> l;
            Entry<K,V> entry;
            l=this.bucket[this.computeIndex(key)];
            Iterator<Entry<K,V>> iter=l.iterator();
            while (iter.hasNext()&&res==null){
                entry=iter.next();
                if (entry.getKey().equals(key)){
                    res=entry.getValue();
                    iter.remove(); }}
            if (l.size()==0)
                this.bucket[this.computeIndex(key)]=null;
            if ((this.minSize<this.size/2)&&(this.nbElem<this.size/4))
                this.reduceBucket();
```

Progression

Construction d'un index

Programme exemple

```
public class Indexeur {
  public static Map<String, List<Integer>> makeIndex (String fileName) {
    Map<String,List<Integer>> res = new HashMap<String,List<Integer>>();
    BufferedReader read;    String line="",word="";
    List<Integer> tmpL;    int lineNbr=0;
    try {
     read = new BufferedReader (new FileReader(fileName));
     while ((line=read.readLine())!=null){ lineNbr++;
          Scanner sc = new Scanner(line);
          while(sc.hasNext()){  word=sc.next();
             if (!(res.containsKey(word))){
               tmpL=new LinkedList<Integer>();
               res.put(word, tmpL);}
             else tmpL = res.get(word);
             tmpL.add(lineNbr); }}// Fin des deux "while"
     read.close(); }
    catch (IOException e){} // Traitement minimal de l'exception
    return res; }} // Fin de la methode et de la classe
```

Progression

La classe `Object` (egalité)

== : identité c-à-d : mêmes instances

La classe `Object` (egalité)

== : identité c-à-d : mêmes instances

Méthode `equals` (relation sur les références)

- Réfléxive : `x.equals(x)` $\Rightarrow$ `true`
- Symétrique : `y.equals(x) == x.equals(y)`
- Transitive : `x.equals(y) and y.equals(z)` $\Rightarrow$ `x.equals(z)`
- Consistante : le résultat ne doit pas changer si les valeurs ne sont pas changées
- Pour toute référence non-nulle, `x.equals(null)` $\Rightarrow$ `false`

La classe `Object` (egalité)

== : identité c-à-d : mêmes instances

Méthode `equals` (relation sur les références)

- Réfléxive : `x.equals(x)` ⇒ `true`
- Symétrique : `y.equals(x) == x.equals(y)`
- Transitive : `x.equals(y) and y.equals(z)` ⇒ `x.equals(z)`
- Consistante : le résultat ne doit pas changer si les valeurs ne sont pas changées
- Pour toute référence non-nulle, `x.equals(null)` ⇒ `false`

Programme exemple

```
public boolean equals(Object obj) {
    if (obj instanceof MaClasse)
        if (this.attrib!=null)
            return this.attrib.equals(((MaClasse)obj).attrib);
        else return ((MaClass)obj).attrib==null;
   return false; }
```

La classe `Object` (valeur de hachage))

Méthode `hashCode`

Cette méthode fournit comme résultat une valeur de hachage pour l'objet courant.

La classe `Object` (valeur de hachage))

Méthode `hashCode`

Cette méthode fournit comme résultat une valeur de hachage pour l'objet courant.

Contrat

Cette méthode garantit que lorsque deux objets sont égaux (au sens de la méthode `equals`), ils ont la même valeur de hachage.
Donc, lorsque `equals` est redéfinie, la méthode `hashCode` doit l'être également.

La classe `Object` (valeur de hachage))

Méthode `hashCode`

Cette méthode fournit comme résultat une valeur de hachage pour l'objet courant.

Contrat

Cette méthode garantit que lorsque deux objets sont égaux (au sens de la méthode `equals`), ils ont la même valeur de hachage.
Donc, lorsque `equals` est redéfinie, la méthode `hashCode` doit l'être également.

Programme exemple

```
public int hashCode() {
     return attrib.hashCode();
 }
```