1

Set, Dictionary et Bag

Main Author(s): Bernard Pottier

1.1 Collections non-ordonnées

Les éléments de ces collections ne sont pas rangés selon un ordre prédictible : on ne peut pas accéder aux éléments via une clé externe, tel qu'un index, ou un ordre connu. Cette propriété est liée au mécanisme d'adressage dit *associatif*, où la position d'un élément dans la collection dépend de la valeur de cet élément et de l'historique des accès (voir les explications en section **??**).

La hiérarchie de classes se présente de la manière suivante:

```
Object ()

  Collection ()
    Bag ('contents')
    Set ('tally')
      Dictionary ()
      IdentitySet ()
```

- Set : collection garantissant l'unicité des éléments.
- Dictionary : accès par une clé, qui est en général un objet d'une classe déterminée. La clé garantit l'unicité.
- Bag : chaque élément a un compteur associé,

Une première utilisation de ces collections est la mise en œuvre d'algorithmes très simples reposant uniquement sur leurs propriétés :

Trouver tous les mots apparus dans un texte

```
| bobyADit lesMotsDeBoby |
bobyADit := 'ta pa ta pa tapa tout dit tapa tout dit a ta dou dou'.
lesMotsDeBoby := bobyADit tokensBasedOn: Character space.
lesMotsDeBoby asSet asSortedCollection asArray

"#('a' 'dit' 'dou' 'pa' 'ta' 'tapa' 'tout')"
```

1.2 Set

1.2.1 Création

Les ensembles sont des collections dynamiques, qui grandissent en fonction des besoins. On crée de nouveaux ensembles par Set new, ou, si on est capable d'apprécier correctement une taille idoine, par Set new: nElements. Si on connait déjà les éléments, parce qu'ils sont rangés dans une autre collection, alors on peut aussi instancier par Set withAll: une Collection.

L'algorithme (hachage) qui sert à la mise en œuvre des ensembles a de bonnes caractéristiques en temps de recherche d'un élément.

1.2.2 Accès

L'usage simple de Set peut être résumé de la manière suivante :

- unSet add: unObjet,
  unSet addAll: uneCollection, ajout dans l'ensemble.
- unSet includes: unObjet vrai ou faux selon que unObjet soit présent ou absent.
- unSet remove: unObjet ou
  unSet remove: un Objet ifAbsent: unBlocException.

  Dans le premier cas, on enlève unObjet de unSet, si cet objet existe. Sinon, on déclenche une erreur.

  Si le programmeur n'est pas certain de la présence de l'objet cherché, il utilise la seconde forme en passant un bloc d'exception, qui peut être vide…
- unSet size : nombre d'éléments présents dans l'ensemble,
- unSet capacity : capacité de stockage de l'ensemble. Plus le rapport $capacity/size$ est grand, plus le temps d'accès risque d'être rapide.

1.3 Dictionary

Un Dictionnaire est un ensemble dont les éléments sont des instances de la classe Association, couplant une clé (key), et une valeur (value). L'unicité d'une association dans un ensemble donné est garantie par l'unicité de la clé, propriété héritée de la classe Set.

Les instances de Association sont fréquemment créées à l'aide du message binaire ->. Par exemple, #boulesRouges->40 associe l'entier 40 au symbole #boulesRouges.

La structure de données de dictionnaire implantée par la classe Dictionary est intéressante à plusieurs titres:

- Possibilité de désigner un objet par une clé souvent symbolique (Symbol, String, nombres…)
- Rapidité d'accès, due au hachage (mêmes performances que les ensembles, voir en section **??**).
- Possibilité d'utiliser les dictionnaires comme des tables de hachage, et pour des clés tout à fait quelconques.
- Dynamicité de la collection qui grandit avec les besoins.

1.3.1 Création et propriétés héritées de Set

On peut se reporter à la description donnée dans Set. Si on souhaite utiliser Dictionary withAll: uneCollection, il faut cependant retenir que tous les éléments de uneCollection doivent être des instances de la classe Association. L'exemple qui suit montre comment ajouter un dictionnaire à un autre, comment créer un dictionnaire initialisé.

La clé servant à adresser le dictionnaire est simplement un entier, la valeur est un caractère extrait d'une chaine. On remplit le premier dictionnaire en notant les index des blancs et les index des ponctuations. Le dictionnaire apparait alors comme un tableau "creux".

```
testDico1
   "self testDico1"

   | dico1 dico2 vers |
   dico1 := Dictionary new.
   dico2 := Dictionary new.
   vers := 'le petit homme de la jeunesse, a casse son lacet de soulier,'.
   vers keysAndValuesDo:
          [:index :car | car = $
                                ifTrue: [ dico1 at: index put: car ]
                                ifFalse: [ car isLetter ifFalse: [ dico1 at: index put: car ]]].
  dico1 addAll: dico2 associations. "ajout d'un dictionnaire a un autre dictionnaire"
  ^Dictionary withAll: dico1 associations "creation d'un dictionnaire pre-rempli"
```

Parmi les messages hérités de la classe Set, se trouvent les opérations ensemblistes qui portent sur les clés: intersection, union, soustraction... Dans le code ci-dessus, la soustraction des deux dictionnaires serait simplement spécifiée par dico1 - dico2.

1.3.2 Accès, ajouts et suppressions

Les méthodes les plus simples pour insérer, supprimer, tester sont les suivantes :

- unDictionnaire at: uneCle put: unObjet,
  par exemple dico at: #titi put: 12.
- unDictionnaire add: uneAssociation,
  par exemple dico add: #titi->12.
- unDictionnaire removeKey: uneCle,
  par exemple dico removeKey: #titi. Le résultat est la valeur associée à la clé (12).

  On dispose aussi de la variante unDictionnaire removeKey: uneCle ifAbsent: unBloc.
- unDictionnaire includesKey: uneCle,
  par exemple dico includesKey: #titi rendrait true dans le contexte ci-dessus.
- unDictionnaire includes: unObjet,
  par exemple dico includes: 12 rendrait true, alors que dico includes: #titi rendrait false,

  De même unDictionnaire occurencesOf: unObjet, rend le nombre de fois où unObjet est apparu en valeur de l'association.
- unDictionnaire keyAtValue: unObjet renvoie une clé associée à une valeur, si toutefois cette valeur existe. par exemple dico keyAtValue: 12 rendrait #titi...

1.3.3 Itérations

On peut distinguer les itérations générales, portant sur les *valeurs* plutôt que sur les clés, les itérations portant sur les clés, et enfin celles qui portent sur le couple clé-valeur.

- unDico do: unBloc
  itération sur toutes les valeurs, par exemple:

```
| dicoDesMots versDePrevert stream |
dicoDesMots := Dictionary new.
stream := WriteStream on: ''.
versDePrevert := 'le petit homme qui dansait dans ma tete'.
(versDePrevert tokensBasedOn: $ )
    do: [ :mot | dicoDesMots at: mot asSymbol put: mot size ].
dicoDesMots do: [:longueurs | stream nextPutAll: longueurs printString; space].
^stream contents
 " '2 4 4 5 7 3 2 5 '"
```

La transformation de la chaine de caractère en symbole (asSymbol) n'est pas indispensable.

- keysDo: itère sur les clés du dictionnaires:

```
...
dicoDesMots keysDo: [ :mot | stream nextPutAll: mot; space ].
^stream contents
" 'dans tete homme dansait qui ma petit le '"
```

- keysAndValuesDo: permet d'accéder simultanément à la clé et à la valeur:

```
...
dicoDesMots keysAndValuesDo: [:mot :taille |
          stream nextPutAll: mot, '(', taille printString,')'; space].
..
" 'tete(4) homme(5) ma(2) petit(5) qui(3) le(2) dans(4) dansait(7) '"
```

- collect: et fonctionne sur les valeurs associées aux clés.

```
| dicoDesMots versDePrevert  |
dicoDesMots := Dictionary new.
versDePrevert := 'les sept eclats de glace de ton rire etoile'.
(versDePrevert tokensBasedOn: $ )
   do: [ :mot | dicoDesMots at: mot size put: mot asSymbol ].
^ dicoDesMots collect:  [ :mot   | mot reverse ]"
" OrderedCollection ('ed' 'not' 'erir' 'ecalg' 'eliote')"
```

- select: unBloc, reject: unBloc opèrent sur les valeurs associées aux clés, mais renvoient un dictionnaire.

```
...
  ^ dicoDesMots select:  [:mot   | mot size ¡= 4 ]
    " Dictionary (2->#de 3->#ton 4->#rire )"
```

1.4 Bag

Cette classe s'appuie sur Dictionary ou IdentityDictionary pour compter le nombre d'occurences d'un élément. L'intérêt est triple: vitesse d'accès (due aux dictionnaires), comptage, et compacité de la structure de données si le nombre des occurrences est élevé.

1.4.1 Ajouts et suppressions

Les sacs (classe Bag) ont un interface permettant d'ajouter ou d'enlever des éléments:

- unSac add: unObjet
  Le compteur des occurences de unObjet est incrémenté.
- unSac remove: unObjet
  unSac remove: unObjet ifAbsent: aBlock
  Le compteur des occurences de unObjet est décrémenté. Si ce compteur tombe à 0, l'objet disparait du sac.

  Dans la seconde version le bloc d'exception est évalué au cas où l'objet n'existe pas: enlever un objet absent est une erreur.
- unSac addAll: uneCollection unSac removeAll: uneCollection
  ajoute ou enlève une collection d'objets.
- unSac add: newObject withOccurrences: n
  unSac removeAllOccurrencesOf: anObject ifAbsent: aBlock
  ajoute n occurences du même objet, ou enlève toutes les occurences d'un objet.

1.4.2 Énumérations

- do: collect: inject: ... : Le principe utilisé est d'évaluer les blocs paramêtres pour chaque élément de l'ensemble. Par exemple, si unObjet apparait 10 fois dans le sac, il y aura 10 évaluations du bloc paramêtre.

  Lorsque l'itérateur renvoie une collection, celle-ci est également un Bag. Dans l'exemple qui suit voyelles est un Bag qui ne comporte que false et true.

  Compter les voyelles dans une chaine

```
| b voyelles |
   b := Bag new.
   b addAll: 'il pleut sans cesse sur Brest'.
   voyelles := (b collect: [:lettre | lettre isVowel ]).
   voyelles occurrencesOf: true
   "8"
```

- valuesAndCountsDo: unBloc : S'il est utile d'associer les éléments et leurs compteurs associés, on utilise ce message et on construit un bloc à deux paramêtres (élément et son compteur).

Imprimer par ordre croissant le nombre de caractères apparus dans une chaine

```
| stream sort |
stream := WriteStream on: (String new:200).
"preparation d'une collection comportant des Associations triees selon le champs value"
sort := SortedCollection new sortBlock: [:a1 :a2 | a1 value < a2 value].
'le petit homme qui chantait sans cesse' asBag valuesAndCountsDo: [:v :c | sort add: v -> c].
sort do: [:association | stream nextPut: association key ; space.
    stream nextPutAll: association value printString, ' - '].
    stream contents
"'q 1 - p 1 - o 1 - l 1 - u 1 - n 2 - m 2 - h 2 - c 2 -
 i 3 - a 3 - t 4 - s 4 - e 5 -  6 - '"
```

1.5 Performances

On veut comparer les performances en insertion et suppression sur les collections Set, SortedCollection et OrderedCollection. Dans un premier temps, on laisse l'ensemble grossir au fur et à mesure des besoins, en testant les 3 collections. Dans un second temps, on fixe d'emblée la taille de l'ensemble à 4 fois la taille des éléments. Dans les deux cas les objets insérés sont des nombres réels aléatoires.

1.5.1 Boucle externe du test et formatage

On y reconnait un stream en écriture sur une chaine de caractères, qui sera utilisé pour produire une table au format LaTeX (voir la table **??**).

Le bloc eval possède deux paramêtres, l'un contient la classe sur laquelle on effectue le test de performance, l'autre est un bloc servant à instancier cette classe. On repère les 6 évaluations de ce bloc en fin de méthode.

Dans le bloc, le temps de calcul insertion/suppression est produit en expédiant le message testCollection: aClass creationBlock: aBlock times: nTimes. Ce message est décrit plus bas.

Il faut noter que l'on répète nTimes fois les opérations et que les temps sont rendus dans une OrderedCollection de taille nTimes.

```
testNew
  "Test testNew"

  | stream eval resultats nTimes |
  nTimes := 3.
  stream := WriteStream on: ''.
  stream nextPutAll: 'Classe & '.
  nTimes timesRepeat: [stream nextPutAll: 'add', ' & ', 'remove', '& '].
  stream nextPutAll: 'add moy.', ' & ', 'remove moy. ', '\\hline'; cr.
  eval :=
      [ :aClass :aBlock |
      | sumX sumY |
      stream nextPutAll: aClass name, ' & '.
      resultats := self
            testCollection: aClass
            creationBlock: aBlock
            times: nTimes.
      sumX := sumY := 0.
      resultats
        do:
          [ :point |
          stream nextPutAll: point x printString, ' & ', point y printString, ' & '.
          sumX := sumX + point x.
          sumY := sumY + point y ].
      stream nextPutAll: (sumX // resultats size) printString, ' & ',
                  (sumY // resultats size) printString, '\\hline'; cr ].
  eval value: Set value: [ Set new ].
  eval value: SortedCollection value: [ SortedCollection new ].
  eval value: OrderedCollection value: [ OrderedCollection new ].
  eval value: Set value: [ Set new: 8000 ].
  eval value: SortedCollection value: [ SortedCollection new: 8000 ].
  eval value: OrderedCollection value: [ OrderedCollection new: 8000 ].
  ^stream contents
```

1.5.2 Boucle interne du test

Ici, on effectue une série de mesures identiques sur une classe passée en paramêtre, que l'on instancie via un bloc également passé en paramêtre.

tRand est le temps de génération de 2000 nombres. tAdd est le temps d'insertion, tRemove, le temps de suppression.

Le temps est évalué en millisecondes en utilisant le message Time millisecondsToRun: unBloc. On renvoie une collection de points comportant en abcisse le temps d'insertion, en ordonnée le temps de suppression (par pure commodité).

```
testCollection: collClass creationBlock: aBlock times: n
  | rand set2 tAdd tRemove tRand set1 resultats |
  resultats := OrderedCollection new: n.
  rand := Random new. "Generateur de nombre aleatoire"
  n
    timesRepeat:
      [ set2 := aBlock value.
      tRand := Time millisecondsToRun: [2000 timesRepeat: [rand next * 100]].
      tAdd := Time millisecondsToRun: [2000 timesRepeat: [set2 add: rand next * 100]].
      set1 := set2 copy asArray.
      tRemove := Time millisecondsToRun: [set1 do: [:elt | set2 remove: elt]].
      resultats add: tAdd - tRand @ tRemove].
  ^resultats
```

1.5.3 Bilan

Classe	add	remove	add	remove	add	remove	add moy.	remove moy.
Set	514	946	503	908	507	920	508	924
SortedCollection	769	2983	805	1564	759	2787	777	2444
OrderedCollection	50	1574	41	2876	17	1568	36	2006
Set	267	361	244	353	264	349	258	354
SortedCollection	1380	2869	852	1551	1326	2846	1186	2422
OrderedCollection	49	1544	5	2887	53	1613	35	2014

Table 1.1: Performances comparées. Le premier groupe de tests porte sur des collections créées avec new. Le second groupe porte sur des ensembles portés, d'emblée à une capacité de 8000

- On peut constater que le coût du sous dimensionnement de la structure du départ peut être important : dans le cas de Set, cela compte pour un facteur 2.
- Si la structure est grande, le coût en suppression est réduit (le second Set est 4 fois trop grand, et on remarque que les suppressions sont à peine plus lentes que les insertions). Dans le cas contraire, le coût des suppressions peut être jusque deux fois plus élevé que celui des insertions.

  L'explication se trouve dans la gestion des collisions et le nombre de celles-ci…
- l'accès en insertion sur les OrderedCollection est très rapide, on ne peut pas en dire autant des suppressions.
- les SortedCollection marquent un avantage en recherche qui n'apparait pas ici.

1.6 Décomposition d'un nombre entier en facteurs premiers

Décrire un algorithme de décomposition s'appuyant sur un ensemble de diviseurs possibles. L'ensemble de ces diviseurs est réduit chaque fois qu'un de ses éléments est traité. S'il s'agit d'un diviseur, on réduit les nombres de l'ensemble à l'aide de ce diviseur. Dans le cas contraire on élimine tous les multiples du diviseur. Les diviseurs sont accumulés dans un Bag.

```
!Integer methodsFor: 'set-exemple'!
decompose
  | diviseurs diviseursPossibles nb minBlock min maxBlock max sizeColl |
  diviseurs := Bag new.
  diviseurs add: 1. nb := self.
  sizeColl := OrderedCollection new.
  diviseursPossibles := (2 to: self // 2) asSet.
  min := 2. max := 1.
  minBlock := [:number | min := min min: number].
  maxBlock := [:number | max := max max: number].
  [diviseursPossibles isEmpty or: [nb = 1]]
    whileFalse:
      [| diviseur division |
      sizeColl add: diviseursPossibles size.
      diviseur := min.
      division := nb / diviseur.
      min := nb // max.
      (division isKindOf: Integer)
        ifTrue:
          [diviseurs add: diviseur.
          maxBlock value: diviseur.
          minBlock value: diviseur.
          nb := division.
          diviseursPossibles := diviseursPossibles
                collect:
                  [:n |
                  | x | x := n / diviseur.
                  x := (x isKindOf: Integer)    ifTrue: [x]    ifFalse: [n].
                  x > (nb / max)
                    ifTrue: [x := min]
                    ifFalse: [(#(0 1) includes: x) ifTrue: [x := max]].
                  minBlock value: x.
                  x]]
        ifFalse: [diviseursPossibles := diviseursPossibles
                reject:
                  [:n |  minBlock value: n.
                  ((n / diviseur isKindOf: Integer) or: [n > (nb / max)])
                    or: [n = diviseur]]]].
  Transcript cr; show: sizeColl printString ; cr.
  ^diviseurs
"(3*25*8*10*4) decompose
 OrderedCollection (11999 5999 2999 1499 749 374 186 93 93 20 13 13 1 1)
 Bag (1 2 2 2 2 2 2 3 5 5 5)"
```

1.7 Hachage

Le hachage d'un objet permet d'adresser une table par le contenu. Cette technique permet d'éviter des opérations de recherche par énumération ou dichotomie. Cette technique est à la base des Set, donc de Dictionary, donc de Bag qui encapsule un dictionnaire.

La présente section décrit sommairement le mécanisme. La réalisation d'algorithmes performants requière une plus grande attention que cette première approche.

Les trois figures qui suivent montrent une opération de recherche :

- On utilise une fonction qui rend un index à partir d'un objet (message hash expédié à l'objet). Des exemples très simples de hachage sont un modulo appliqué à un objet entier, une fonction $c1 \times 256 + c2$ appliquée aux deux premiers caractères d'une chaine…

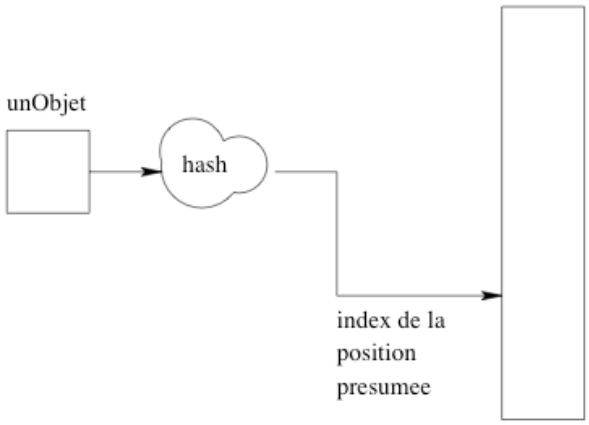

Figure 1.1: L'objet est haché et on obtient un index dans la table

- L'index présumé ayant été produit, on accède à la table. Si la table ne contient rien (nil), alors l'objet n'est pas dans la table. Sinon on compare l'objet cherché et l'objet trouvé (a = b). Voir figure **??**.

  S'il y a égalité, on a trouvé l'objet…

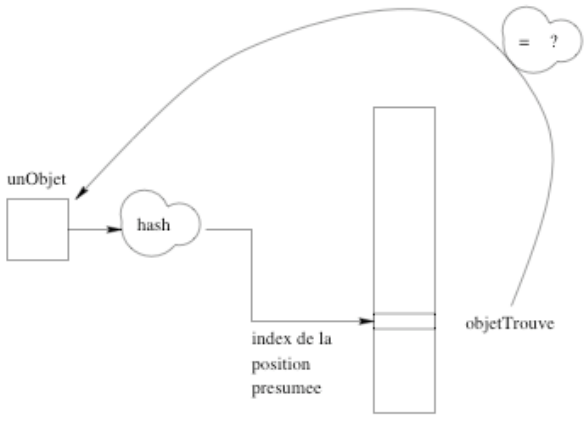

Figure 1.2: On compare l'objet implanté dans la table et l'objet cherché

- Sinon, on descend en séquence en cherchant soit nil, soit l'égalité (voir figure **??**).

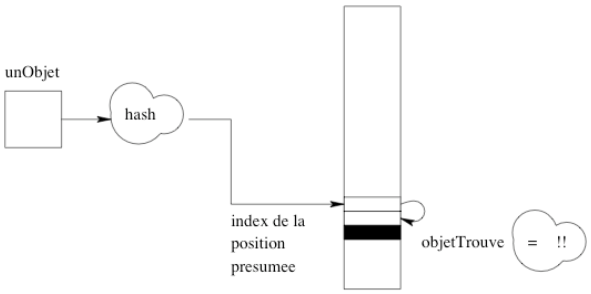

Figure 1.3: Recherche en séquence

Les opérations d'insertion sont menées de la même manière, il s'agit ici de trouver le premier index libre lors de la recherche en séquence. Lorsque la quantité de collisions grandit (place libre de plus en plus petite), il est préférable de reconstruire la table en doublant sa taille.

Les suppressions requièrent une reconstruction au moins partielle de la table.