Projet LSINF1121 - Algorithmique et structures de données

-

Rapport intermédiaire Mission 3

Groupe 3.2

Boris Dehem (5586-12-00)

Sundeep Dhillon (6401-11-00)

Alexandre Hauet (5336-08-00)

Jonathan Powell (6133-12-00)

Mathieu Rosar (4718-12-00)

Tanguy Vaessen (0810-14-00)

Année académique 2014-2015

Questions et réponses

Question 1

Quelles sont les trois méthodes principales du type abstrait Map ?

Les méthodes principales du type abstrait Map sont : put(k,v),remove(k) et get(k).

Qu'entend-on par clé, valeur et entry dans ce contexte ?

Une clé est un identificateur unique qui est assigné par une application ou un utilisateur à un objet/ une valeur associé(e). Dans le contexte d'un Map, chaque clé doit être unique

Dans un Map, les valeurs sont les données qui constituent les éléments du type abstrait. C'est sur base de ces données que les éléments sont organisés dans la structure.

A chaque valeur, il correspond une clé unique (dans le cas d'un type abstrait Map) qui à elles deux forment une entrée.

Est-il possible d'insérer une valeur null dans un Map ? Expliquez.

Il est possible d'insérer une valeur null dans un Map, bien qu'en pratique ce ne soit pas d'application. Null n'est pas utilisé comme valeur, car celle-ci est spécifique, elle est utilisée comme valeur de retour lorsque la méthode get(k) n'a pas trouvé d'entrée correspondante à la clé passée en paramètre dans le Map.

Citez plusieurs exemples d'application d'un Map et précisez dans chaque cas à quel type d'information correspond chaque entry.

Un cas possible d'utilisation d'un Map est le cas d'un annuaire de code postaux. A chaque ville, les valeurs, correspond un code postal unique, la clé qui forment les entrées du Map.

Une autre utilisation pourrait être un Map utilisé comme dictionnaire, avec chaque entrée constituée d'un index et d'un mot.

Quelles sont les trois méthodes principales du type abstrait `Map` ? Qu'entend-on par clé, valeur et entry dans ce contexte ? Est-il possible d'insérer une valeur null dans un `Map` ? Expliquez.
Citez plusieurs exemples d'application d'un `Map` et précisez dans chaque cas à quel type d'information correspond chaque *entry*.

TODO after ma conférence Ebauche de réponse:

Les trois méthodes principales d'un type abstrait `Map` **M** sont les suivantes:

1. *get(k)*: Si M contient une entrée e = (k,v), avec une clé égale à k, alors la valeur de retour sera v, pour e ou alors renverra **null**.

2. *put(k,v)*: Si M n'a pas d'entrée avec une clé égale à k, alors il faut ajouter une entrée (k,v) à M pour que l'on retourne **null**; sinon, il faut remplacer v par la valeur existante de l'éntrée avec la clé égale à k et retourner la valeur antérieure.

3. *remove(k)*:

Question 2

Citez au moins quatre implémentations différentes d'un dictionnaire. Précisez, dans chaque cas, quelles sont les propriétés principales de ces implémentations. Dans quel(s) cas s'avèrent-elles intéressantes ? Quelles sont les complexités calculatoires de leur principales méthodes ? Une *Skip List* constitue-t-elle une implémentation possible d'un dictionnaire non ordonné ? Pourquoi ? *(Tanguy)*

Un dictionnaire permet de stocker des paires <clé, valeur> appelées des entrées. Contrairement aux *Map's*, qui n'accepte que des clés uniques, un dictionnaire autorise plusieurs entrées avec la même clé. Dans les quatre implémentations ci-dessous, nous augmentons le type des entrées qui seront stockées afin qu'elles possèdent une variable *position*. Cette variable devra à tout moment indiquer quelle est la position de l'entrée dans la structure de données. Ce mécanisme est détaillé dans DSAJ-5 section 4.8.2.
Citez au moins quatre implémentations différentes d'un dictionnaire.

1. Liste non-ordonnée.
   Dans le cas d'une liste non-ordonnée, nous pouvons utiliser la méthode *remove(entry)* comme suit : *list.remove(entry.location())* qui s'exécutera en $\Theta(1)$. Cette implémentation est intéressante lorsqu'on devra à la fois ajouter et retirer de nombreuses entrées.

2. Hashtable.
   Nous avons ici une table hachée avec un *"bucket array"*, en français un *tableau de seaux* littéralement. l'implémentation utilise la fonction de hachage h et utilise des chaines séparées pour gérer les collisions. Cette explication est représentée schématiquement sur la figure **??**[1]. Lors de l'appel à la méthode *remove(entry)*, la méthode de hachage est utilisée pour accéder au bon emplacement dans le tableau (au bon *"bucket"*). Avec une méthode de hachage efficace, les collisions sont rares et chaque bucket ne contiendra que 1 ou 0 élément mais rarement plus. Si il y en a plus, la variable *location* permettra de déterminer la position de l'élément recherché. La méthode *hachtable.remove(entry.location())* s'exécutera donc en $\Theta(1)$. Cette implémentation sera intéressante quand on aura très peu d'entrées avec la même clé (car il y aura alors peu de collision).

3. Table ordonnée de recherche. Dans une table ordonnée de recherche, la variable *location* est un entier indiquant l'index de l'élément dans la table. La méthode *table.remove(entry.location())* s'exécutera en $\Theta(n)$ car il faudra déplacer n éléments de la liste pour la maintenir ordonnée. Le meilleur des cas étant le cas où *entry* est la dernière entrée de la liste et le pire des cas où *entry* est la première entrée de la liste. Cette implémentation est intéressante si l'on ajoute les entrées de manière ordonnée puis que l'on doit faire de nombreuses recherches dessus.

[1] Source wikipédia: `https://en.wikipedia.org/wiki/Hash_table`

4. Skip List.
   Une Skip List est une liste de listes (S_0, S_1, ..., S_h) où h est la taille de la Skip List. Chaque sous-liste S_i contient les éléments $+\infty$ et $-\infty$ ainsi qu'un sous-ensemble des éléments de S_{i-1}. Ainsi, la liste S_0 contient tous les éléments plus $+\infty$ et $-\infty$. Ces éléments sont stockés dans l'ordre croissant. $-\infty$ étant plus petit que toutes entrées pouvant être insérée dans la liste, il sera toujours le premier élément et $+\infty$, de manière équivalente, sera toujours le dernier élément. Pour construire la sous-liste S_i, après avoir ajouté les éléments $+\infty$ et $-\infty$, les éléments de S_{i-1} sont parcourus un à un et chacun d'eux à une probabilité de 0,5 d'être ajouté dans S_i. Si S_0 possède n éléments, on s'attend à ce que S_1 en possède $n/2$, S_2 $n/4$ et S_i $n/2^i$. On s'attend donc aussi à ce que la hauteur h de la skip list soit égale à $log(n)$ La figure **??** représente une Skip List de hauteur 5 contenant 10 éléments[2]. Nous pouvons voir une Skip List comme étant une structure à deux dimensions de positions agencées horizontalement en niveaux et verticalement en tours. Chaque niveau est une liste S_i et chaque tour contient les positions référençant la même entrée à travers les différentes listes. Les Skip List permette d'effectuer des recherches grâce à une méthode appelée SkipSearch. Cette méthode est détaillée dans DSAJ-5 page 413. L'idée générale est de rechercher une entrée ayant pour clé k en partant du haut de la liste (l'élément le plus à droite de la liste la plus haute, c'est-à-dire toujours $-\infty$). Ensuite, on parcourt le niveau courant vers la droite tant que la valeur de la clé de l'élément suivant est $< k$. Ce cas existera toujours car chaque niveau contient l'élément $+\infty$. Ensuite on navigue vers le bas dans la tour et on recommence la méthode jusqu'à atteindre S_0. Les méthodes permettant d'insérer, retirer et rechercher une entrée dans une Skip List s'exécuteront en moyenne en $\Theta(\log(n))$. Cette implémentation est intéressante dans le cas où on ferait de nombreuses opérations (insertion, recherche, suppression).
   Une Skip List constitue-t-elle une implémentation possible d'un dictionnaire non ordonné ? Pourquoi ?
   Il est possible d'implémenter un dictionnaire en utilisant une Skip List. La varialbe *location* de chaque entrée sera égale à la position de l'entrée dans le bas de la liste (en S_0).

Question 3

Nous considérons un tableau A à 2 dimensions ($n * n$ éléments) qui contient des 1 et des 0, de telle sorte que, sur chaque ligne de A, tous les 1 présents viennent avant tout 0 sur cette ligne. En supposant que A est déjà en mémoire, décrivez un algorithme qui compte le nombre de 1 dans A et s'exécute en $O(n\ log\ n)$ (et non pas $\Theta(n^2)$). Justifiez pourquoi la complexité temporelle de votre algorithme est bien en $\Theta(n\ log\ n)$ *(Tanguy)*

```
int compterNombreDe1(Tableau A){
        int nombreDe1 = 0;
```

[2]Source: DSAJ-5, Figure 9.9, page 412

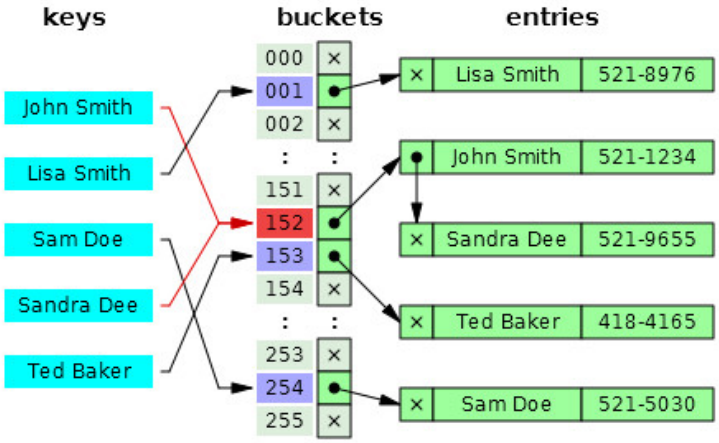


Figure 1: Hachtable with collision resolve by separate chaining.

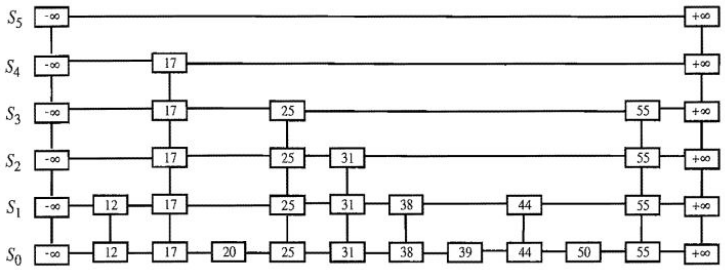


Figure 2: Example of skip list storing 10 entries.

```
        foreach(ligne in A){
                int indexCourant = n/2; //n est le nombre d'elements d'une ligne

                //Boucle tant qu'on n'a pas atteint une des extremites ou
                    jusqu'a ce que l'element courant vaille 1 et le suivant 0
                while( indexCourant != 0 && indexCourant != n-1 &&
                    !(ligne.element(indexCourant) == 1 &&
                    ligne.element(indexCourant + 1) == 0) )
                {
                        if(ligne.element(indexCourant) == 1){
                                //voir dans le sous-tableau de droite
                                indexCourant += (n - indexCourant) / 2;
                        }
                        else {
                                //voir dans le sous-tableau de gauche
                                indexCourant /= 2;
                        }
                }

                if(ligne.element(indexCourant) == 1 && (indexCourant== (n-1) ||
                    ligne.element(indexCourant + 1) == 0)){
                        nombreDe1 += indexCourant + 1; //car l'index part de 0
                            donc s'il vaut 6 il faut ajouter 7 au nombre de 1.
                }
        }
        return nombreDe1;
}
```

Le pseudo-code ci-dessus s'exécute en $\Theta(n \log(n))$ car à chaque itération de la boucle *while*, la moitié des sous-possibilités est éliminée et qu'il faut parcourir les n lignes du tableaux.

Question 4

Imaginez une nouvelle méthode à ajouter au type abstrait Dictionary. Cette méthode produit l'union de deux dictionnaires. Écrivez une spécification d'une telle méthode. Écrivez ensuite un algorithme qui construit l'union de deux dictionnaires ordonnés implémentés par des tables ordonnées (ordered search tables). Comment appliquer votre algorithme si les dictionnaires étaient implémentés par une table de hachage ? (Alexandre)

Precondition: Les dictionaires a et b sont de type ordoné
Postcondition: Retourne l'union ordonnée des deux dictionnaires a et b
Dictionary union(Dictionary a, Dictionary a)

Input: a et b sont deux dictionnaires ordonnés
Output: *result* est l'union de a et b

$sizeResult \longleftarrow sizeA$+sizeB;
result dictionnaire de taille sizeResult;
$xa \leftarrow 0$;
$xb \leftarrow 0$;
for $xr \leftarrow 0$ **to** *sizeResult* **do**
  $valeurA \leftarrow find(xa)$;
  $valeurB \leftarrow find(xb)$;
  if $valeurA == NULL$ *or* $valeurB < valeurA$ **then**
    $result[xr] \leftarrow b[xb]$;
    $xb \leftarrow xb + 1$;
  end
  else if $valeurB == NULL$ *or* $valeurA < valeurB$ **then**
    $result[xr] \leftarrow a[xa]$;
    $xa \leftarrow xa + 1$;
  end
  else
    $result[xr] \leftarrow a[xa]$;
    $ra \leftarrow xa + 1$;
    $rx \leftarrow xr + 1$;
    $result[xr] \leftarrow b[xb]$;
    $xb \leftarrow xb + 1$;
  end
end
return *result*

Cet algorithme ne serait pas utilisé pour des dictionnaires implémentés avec une tables de hachage. Car il suffit de parcourir un des deux dictionnaires et de lire les valeurs à fin de les ajouter à l'autre.

Question 5

Comment obtenir la liste des entrées mémorisées dans un Map implémenté par une table de hachage ? Cette implémentation conviendrait-elle pour un Map ordonné ? Quelles seraient les complexités temporelles des méthodes spécifiques au Map ordonné dans ce cas ? Si l'on dispose d'une liste triée de toutes les entrées d'un dictionnaire et que l'on suppose cette liste fixe (aucun ajout ou retrait n'est permis), est-il intéressant de mémoriser ces entrées dans une table de hachage ? Justifiez votre réponse. Quelle est la complexité spatiale d'une table de hachage ? (Boris)

La méthode entrySet() permet d'obtenir une liste des entrées clé-valeur. Elle parcourt la table de hachage, et pour chaque élément non vide, elle crée un élément dans un tableau. Cette implémentation convient pour un Map ordonné. La complexité temporelle pour cette méthode spécifique au Map ordonné est 0(n). Si l'on dispose d'une liste triée fixe de toutes les entrées d'un dictionnaire, il n'est pas intéressant de la stocker dans une table de hachage, car on peut la stocker dans un tableau ordonné, ce qui prendrait moins d'espace car on ne gaspillerait pas d'espace pour éviter les collisions. De plus, une table de hachage mélangerait toutes les données du dictionnaire. La complexité spatiale d'une table de hachage est 0(1) car ajouter ou retirer un élément de la table ne change pas sa taille (sauf au cas exceptionnel où on décide d'augmenter la taille de la table car le « load factor » est trop élevé.

Question 6

Qu'entend-on par la notion de collision dans une table de hachage ? Les collisions ont elles une influence sur la complexité des opérations ? Si oui, quelle(s) opération(s) avec quelle(s) complexité(s), sinon précisez pourquoi. Quelles sont les techniques utilisées pour gérer les collisions ? Peut-on, grâce à ces techniques, éviter les collisions ? (Boris)

Une collision a lieu lorsque la fonction de hachage envoie une clé à une adresse déjà occupée. A cause des collisions, les foncions get(), put() et remove() pourraient théoriquement s'exécuter en 0(n) dans le pire des cas, car si tous emplacements qu'on essaie sont occupés, il faudra tous les parcourir avant d'en trouver un libre (dans le cas du « open adressing » ou bien il faudra parcourir tous les éléments qui ont déjà été stockés à cette adresse (dans le cas du « separate chaining ») En réalité, on minimise les collisions, de sorte à ce que ces fonctions s'exécutent en 0(1) la majorité du temps. La technique du « separate chaining » consiste à avoir la possibilité de stocker plusieurs éléments dans le même emplacement d'une table de hachage, par exemple en les liant à l'aide d'une liste chainée. La technique du « Open adressing » consiste à trouver un autre emplacement dans la table de hachage que celui renvoyé par la fonction de hachage. Ces techniques parmettent de gérer les collisions, mais pas de les éviter. On peut par d'autres moyens les minimiser, mais jamais complètement les éviter.

Question 7

Java fournit la classe java.util.Hashtable comme implémentation de l'interface java.util.Map. Pouvez-vous déterminer précisément de quelle variante de table de hachage il s'agit ? (Alexandre)

La classe Hashtable du package java.util permet de créer des collections d'objets associés à des noms, en quelque sorte des dictionnaire. Une même Hashtable peut contenir des objets de classe quelconque.

Java fournit-il d'autres implémentations de l'interface Map ? Faites un diagramme qui représente les interfaces et les classes qui se rapportent à Map et précisez ce qui, dans chaque cas, les caractérise.

Oui, Java fournit d'autres implémentations de l'inferface Map

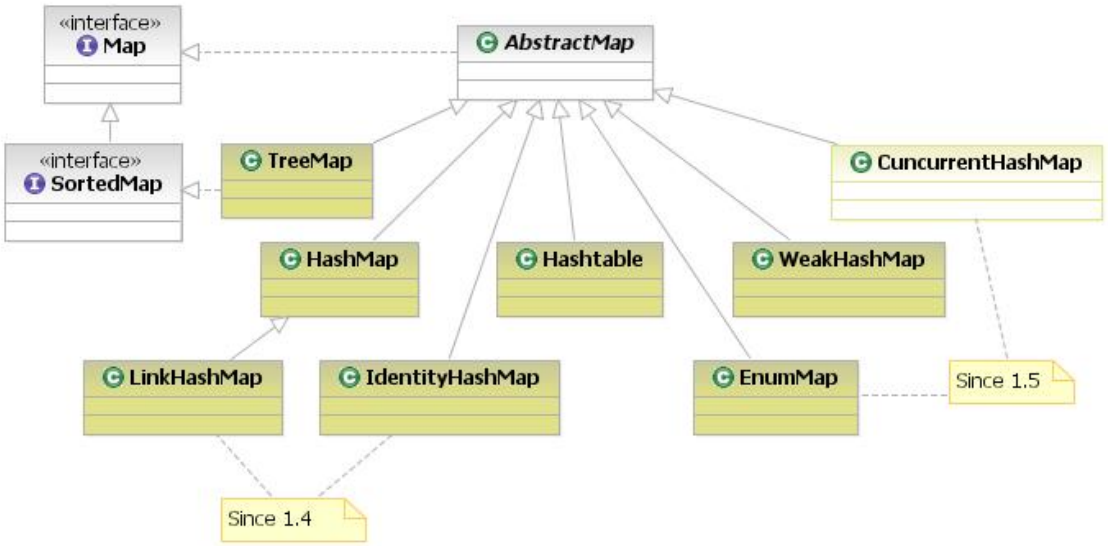
Les interfaces :

- Bindings : Mapping des paires clé/valeur, toutes les clés sont des strings
- ConcurrentMap<K,V> : Ajoute les méthodes putIfAbsent, remove et replace par rapport à Map
- ConcurrentNavigableMap<K,V> : ConcurrentMap qui support les opérations sur les NavigableMap
- NavigableMap<K,V> : Ajoute des méthodes de navigation à Map
- SortedMap<K,V> : Les éléments sont comparables entre eux

Les classes :

- AbstractMap : Implémente quelques méthodes de java.util.Map. Tel que size() et isEmpty()
- Attributes : Map qui attribue un nom à un string
- AuthProvider : Table de hashage avec login et mots de passe
- ConcurrentHashMap : Identique à Hashtable mais implémente en plus l'interface ConcurrentMap
- ConcurrentSkipListMap : Identique à Hashtable mais implémente en plus l'interface ConcurrentMap
- EnumMap : Ne prend comme key que des objets de la class Enum
- HashMap : Équivalent à Hashtable mis à part qu'elle permet les éléments null
- IdentityHashMap : Classe spéciale, elle n'implémente pas rigoureusement Map. C'est une table de hachage qui utilise == et non equals pour comparer deux éléments

- LinkedHashMap : Table de hachage avec une liste complémentaire reprenant les éléments dans l'ordre dans lequel ils ont été introduit
- Properties : Permet la lecture de fichier xml. Fait correspondre une valeur à une propriété
- Provider : Fait partie de java.security. Peut utiliser les algo DSA, SHA-1, MD5, RSA...
- RenderingHints : Contient des éléments qui peuvent être utilisé par Graphics2D
- TabularDataSupport : Implémente en plus de Map la classe TabularData
- TreeMap : Implémente SortedMap, c'est un arbre ordonné
- UIDefaults : Fait pour les composant swing, contient les valeurs par défaut
- WeakHashMap : Même caractéristiques que HashMap. En plus retire les entrées dont la clé n'est plus utilisé de manière normal.



Qu'est-ce qui peut servir de clé pour une Hashtable en Java ? Tout objet non null peut être utilisé comme clé. Par exemple le nom d'un nombre peut-être utilisé comme clé.

```
Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

Question 8

Énoncé

Réponse

Question 9

Question liée spécifiquement au problème posé]

Le temps calcul pour les opérations principales associées à un dictionnaire correspond-il à ce à quoi vous vous attendiez sur base de la complexité temporelle de ces opérations ? Comment faire une analyse expérimentale pertinente de ce temps calcul ?

Décrivez le protocole que vous allez mettre en place et, une fois le problème résolu, commentez les résultats.

Réponse