

Algorithmique et structures de données : Mission 3

Groupe 1.2: Ivan Ahad - Jérôme Bertaux - Rodolphe Cambier
Baptiste Degryse - Wojciech Grynczel - Charles Jaquet

24 octobre 2014

Question 1 (Bertaux Jérôme) Les trois méthodes principales du type abstrait Map sont :

- `get(k)` : Permet de récupérer la valeur associée à la clef `k` et si aucune valeur n'est associée à cette clef on récupère un `null`.
- `put(k,v)` : Si aucune entrée n'est associée à la clef `k` alors cette entrée est ajoutée et on retourne `null`. Sinon on remplace l'ancienne valeur de `k` par la nouvelle et on retourne l'ancienne valeur.
- `remove(k)` : Retire l'entrée associée à la clef `k` et retourne sa valeur et si aucune entrée n'est associée à `k` alors on retourne `null`.

Dans ce contexte :

- `clef` : Une clef est un identifiant unique qui permet d'identifier une valeur dans la Map.
- `valeur` : Une valeur est l'objet de donnée qu'on souhaite stocker dans la Map.
- `entry` : Une *entry* est une paire clef-valeur.

Il est possible d'insérer une entrée `k-null` dans une Map. Donc quand on demande la valeur de la clef `k` on récupère un `null` non pas car il n'existe pas dans la Map mais parce que cette valeur est autorisée. Certaines implémentations de Map interdisent l'utilisation de valeur `null`. Mais pour lever l'ambiguïté dans les implémentations autorisant la valeur `null`, il existe une méthode `containsKey(k)` pour vérifier si `k` est une clef valide. Voici quelques exemples d'application d'un Map :

- Dans un annuaire téléphonique : la clef est le numéro de téléphone et la valeur est le nom de la personne.
- Dans le registre national : la clef est le numéro d'identification du registre national et la valeur est le dossier personnel du citoyen.
- Dans une banque : la clef est le numéro de compte de la personne et la valeur est le dossier bancaire de la personne.

Question 2 (Bertaux Jérôme)

1. Une hashtable : Les opérations basiques de la Map `get`, `put` et `remove` peuvent être implémenter en $O(1)$.
2. Une table de recherche ordonnée : son espace nécessaire est de $O(n)$. Le `get` s'exécute en $O(\log n)$, le `put` en $O(n)$ et le `remove` en $O(n)$.
3. Une liste non ordonnée : Les méthodes fondamentales s'exécute en $O(n)$ dans les pires cas car il faut parcourir toute la liste pour vérifier l'existence d'une entrée déjà existante.
4. Une Skip List est une implémentation possible pour un dictionnaire non ordonné mais il n'est pas judicieux de l'utiliser. Car le système de rangement des entrées est basé sur de l'aléatoire. Donc la recherche se fait en $O(\log n)$, l'insertion en $O(\log n)$ et la suppression en $O(\log n)$.

Q3 (Baptiste Degryse) Il faut utiliser l'algorithme de binary search qui est de complexité $O(\log(n))$. Il faut l'appliquer sur chaque ligne du tableau, multipliant cette complexité par n . L'algorithme est le suivant:

```

1 int [] lastOne=new int [n];
2 for (z=0;z<n;z++){
3     int a=0,b=n, lastOne;
4     while (b-a>1){
5         if (tab [z] [(a+b)/2]==0)
6             b=(a+b)/2;
7         else if (tab [z] [(a+b)/2]==1)
8             a=(a+b)/2;
9     }
10    lastOne [z]=a;
11 }

```

Q4 (Baptiste Degryse) Une spécification de la méthode union:

```

1 /**
2  * @input : d1 et d2, deux dictionnaires ordonnés
3  * @output : un dictionnaire ordonné contenant l'union des éléments des deux
4              dictionnaires passés en entrée.
5  */

```

Pour l'algorithme, dans le cas d'une implémentation des dictionnaires en tables ordonnées:

```

1 Entree tab[] = new Entree [d1.size()+d2.size()]
2 int i,j;
3 while (i+j<tab.size()){
4     if (i>d1.size() || (j<d2.size() & d1[i].key()>d2[j].key())){
5         tab[i+j]=d2[j];
6         j++;
7     }
8     else{
9         tab[i+j]=d1[i];
10        i++;
11    }
12 }
13 return tab;

```

Dans le cas de l'utilisation d'une table de hachage, chaque appel $dx[y]$ deviendrait $dx[h(y)]$.

Q5 (Grynczel Wojciech) Comment obtenir la liste des entrées mémorisées dans un Map implémenté par une table de hachage ?

Pour obtenir la liste des entrées mémorisées, il suffit d'appeler la méthode : `keySet()`

Cette implémentation conviendrait-elle pour un Map ordonné ?

Non, cette implémentation ne conviendrait pas. Quand un élément est ajouté dans une table de hachage, sa position dans la table va dépendre du résultat de la méthode de hachage. Pour implémenter un Map ordonné, il vaut mieux utiliser une autre structure de données. Une table de hachage ne garde pas l'ordre.

Quelles seraient les complexités temporelles des méthodes spécifiques au Map ordonné dans ce cas?

- `firstEntry()` : $O(N)$
- `lastEntry()`: $O(N)$
- `ceilingEntry(k)`: $O(N)$
- `floorEntry(k)`: $O(N)$
- `lowerEntry(k)`: $O(N)$
- `higherEntry(k)`: $O(N)$
- `subMap(k1, k2)`: $O(N)$

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?

Ce n'est pas la meilleure solution. Même si la lecture de n'importe quel élément se fera en complexité de $O(1)$, les données stockées dans la table de hachage ne seront plus triées. Une table de hachage est utile seulement dans le cas où la notion d'ordre n'est pas importante.

Quelle est la complexité spatiale d'une table de hachage?

Une table contient m adresses et n maillons portant les éléments contiennent chacun un objet et une adresse. $n \text{ objets} + (n + m) \text{ adresses}$

Q6 (Rodolphe Cambier) La collision c'est quand plusieurs objets sont assignés à la même clé dans la table de hachage. La complexité de la fonction de recherche d'un élément d'une table de hachage est généralement en $O(1)$, mais si des collisions surviennent, la complexité peut monter jusqu'à $O(n)$. En effet, si, dans le cas extrême, tous les éléments se retrouvent sur la même clé, il faudra, au pire, tous les parcourir afin de trouver celui qu'on recherche.

Pour gérer les collisions, on peut chaîner les différents éléments d'une même clé et tous les mettre dans cette même clé. On peut également rechercher une clé vide proche et utiliser cette clé à la place de celle de départ. Ces techniques permettent d'éviter les collisions, mais elles ralentissent la recherche.

Q7 (Ivan)

Pouvez-vous déterminer précisément de quelle variante de table de hachage la classe `Java.util.Hashtable` s'agit? La classe `java.util.Hashtable` est une `AbstractHashMap`.

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 les caractérise. -Interface `Map`

-Classe `AbstractMap` : L'`AbstractMap` est un type abstrait de données qui représente une collection de paires clef-valeur et qui implémente l'interface `Map`.

-Classe `UnsortedTableMap` : La classe `UnsortedTableMap` étend la classe `AbstractMap` et correspond à une table dont les éléments de la collection ne sont pas ordonnés, ce qui implique un manque d'efficacité dû au fait que chaque méthode a une complexité linéaire au maximum au lieu d'être constante car il faut scanner tous les éléments pour trouver une entrée.

-Classe `AbstractHashMap` : La classe `AbstractHashMap` étend la classe `AbstractMap`. La principale différence est que les tables de hachages peuvent avoir des clés qui ne sont pas des entiers mais qui peuvent être des symboles.

-Classe `ChainHashMap` : Cette classe étend la classe `AbstractHashMap` et utilise le chaînage de sorte que toutes les valeurs ayant la même clef soient regroupées dans une structure chaînée. Cela permet d'éviter les collisions.

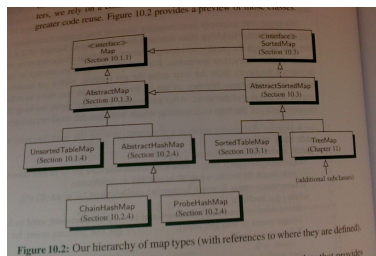
-Classe `ProbeHashMap` : Cette classe étend la classe `AbstractHashMap` et utilise aussi un système permettant d'éviter les collisions. Le système est tel que si l'on veut insérer un élément dans la table mais qu'une clef possède déjà une valeur, on essaye d'insérer la valeur à la clef suivante, et ainsi de suite jusqu'à trouver une clef non-occupée.

-Interface `SortedMap` : L'interface `SortedMap` étend l'interface `Map`. La particularité de `SortedMap` est que les valeurs sont mappées de sorte qu'elles soient ordonnées.

-Classe `AbstractSortedMap` : Il s'agit d'un type abstrait de données qui représente une collection ordonnée de paires clef-valeur. Cette classe implémente l'interface `SortedMap`.

-Classe `SortedTableMap` : Cette classe étend la classe `AbstractSortedMap` et correspond à une table dynamique dans laquelle sont stockées les entrées dans un ordre croissant selon la valeur des clefs. Grâce à cette structure la recherche binaire est possible.

-Classe `TreeMap` : Cette classe permet de représenter les valeurs sous forme hiérarchique.



==> Diagramme représentant les classes et interfaces se rapportant à Map.
(Source : DSAJ6, page 374)

Qu'est-ce qui peut servir de clef pour une hashtable en Java?

Les chaînes de caractères peuvent servir de clef en Java car il s'agit d'un type plus général que les caractères ou les entiers. Ainsi on peut utiliser plus de symboles pour définir les clefs.

Q8 (Charles Jacquet) La question est de savoir comment faire pour implémenter la fonction `remove(k)` lorsqu'on utilise la technique du linear probing. Premièrement, le linéar probing signifie que lorsqu'on veut placer un élément, on en prend le hashcode, on le compresse. Ensuite, s'il y a une collision lors du placement dans la map, cette technique veut qu'on mette à l'élément à la position libre suivante. Voici les étapes à exécuter pour supprimer un élément:

- Chercher la position de l'élément grâce à la hashtable.
- On le supprime
- On crée une liste dans laquelle on met tous les éléments suivant jusqu'à ce qu'il y ait un élément vide.
- On refait `put(k)` pour chaque élément pour être certain qu'ils soient au bon endroit.

Par contre, ici, la complexité varie, c'est-à-dire que dans le meilleur des cas, il y a un trou juste après l'élément, alors la complexité est en $O(1)$. Par contre, dans le pire des cas, c'est-à-dire si l'élément à supprimer est le premier et que toute la map est remplie alors c'est en $O(n)$.

Q9