

Algorithmique et structures de données : Mission 2

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

18 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 ordonnée:

```

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

Q6

Q7

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