TD : Evaluation de performances - Table de hachage

Nous avons étudié précédemment deux types de conteneur : (i) les tableaux et (ii) les éléments chainés. Les premiers sont dits à *accès direct* : on peut aisément et rapidement retrouver un élément à partir de sa position. Les seconds sont dits à *accès séquentiel* : accéder à un élément donné nécessite d'accéder d'abord à tous les éléments de position inférieure. Bien que leur caractéristiques d'accès soient différentes, la recherche d'un élément dans les deux conteneurs est de complexité O(n) : il faut parcourir les éléments les uns après les autres, en accès direct ou séquentiel, jusqu'à trouver celui recherché.

Nous allons développer et étudier dans ce TD un nouveau type de conteneur : les tables de hachage (hash tables). Elles ont la propriété importante d'avoir une complexité de recherche constante, i.e. O(1), tout en maintenant une complexité d'ajout et de retrait constante également.

Table des matières

1 Principe

Le principe d'une table de hachage est simple : (i) sa structure de base est un tableau, contenant les éléments stockés, (ii) l'indice d'un élément dans ce tableau est calculé par une fonction, dite de hachage, à partir de l'élément lui-même. C'est à cet indice qu'on ajoute l'élément, qu'on va le chercher pour déterminer s'il est présent et qu'on va le chercher pour le retirer.

La principale qualité d'une fonction de hachage est de retourner des indices très dispersés, de façon à ce que les éléments occupent – idéalement – des cellules distinctes du tableau. Il est cependant possible que la fonction retourne le même indice pour deux éléments distincts : c'est ce qu'on appelle une *collision*. Pour prendre en compte cette situation, une cellule du tableau contient non pas *un seul* élément, mais une *liste* d'éléments ayant la même valeur de hachage. Cette liste, dite de collision, est implémentée sous forme d'éléments chainés, avec un simple pointeur de tête : l'ajout d'un élément se fait en tête.

Les tables de hachage sont utilisées pour implémenter efficacement des structures de données de haut niveau telles que les ensembles (éléments en exemplaire unique : set Python) ou les dictionnaires (associations clé – valeur : dict Python). Nous nous focalisons dans ce TD sur les dictionnaires : la fonction de hachage sera appliquée à la partie clé d'un couple (clé, valeur).

2 Version simple

Les fonctions essentielles d'un dictionnaire sont :

- ajouter(cle, valeur), qui ajoute le nouveau couple (cle, valeur) si la clé n'existe pas encore, ou remplace la valeur courante de la clé par la nouvelle valeur si la clé existe déjà,
- valeur(cle), qui retourne la valeur courante associée à la clé si elle existe, ou NULL si elle n'existe pas,
- retirer(cle), qui retire le couple (cle, valeur) correspondant à la clé,
- taille(), qui donne le nombre de couples (clé, valeur) existants.

Pour simplifier, nous développons dans un premier temps un dictionnaire dont les clés et les valeurs sont des chaines de caractères.

2.1 Interface du service

Sur le modèle de l'interface liste, développez le fichier le fichier `dictionnaire.h` matérialisant l'interface d'un dictionnaire telle que décrite ci-dessus.

2.2 Programme de test

Développez dans le fichier `testDictionnaire.c` un programme de test fonctionnel du service, qui exerce toutes ses fonctions essentielles.

2.3 Implémentation du service

Développez dans le fichier `dictionnaire.c` une implémentation simple du service :

- Le tableau est de taille fixe, définie par une constante, par exemple `TAILLE_TABLEAU`.
- La fonction de hachage est également simple : elle retourne par exemple le premier caractère de la clé, considéré comme un entier, ou la somme des caractères de la clé pour limiter les risques de collision. Noter que dans tous les cas, ce n'est pas la valeur h retournée par la fonction que l'on utilise directement pour indicer le tableau, mais `h % TAILLE_TABLEAU`.

2.4 Test du service

Testez que votre implémentation fonctionne correctement.

Testez ensuite qu'elle continue à fonctionner correctement dans le cas particulier où tous les éléments se trouvent sur une seule et même liste de collision. C'est ce qui arrive quand (i) la fonction de hachage retourne une constante, et/ou (ii) le tableau est de taille 1. Vous aurez ainsi testé que vous avez correctement développé les opérations d'ajout, de retrait et de recherche dans les listes de collision.

3 Redimensionnement dynamique du tableau

Les performances de recherche dans la table vont se dégrader si le taux de collisions devient trop élevé. C'est notamment le cas si le nombre d'éléments stockés est plus grand que la taille du tableau.

Développez une seconde version de votre implémentation qui agrandisse automatiquement le tableau quand le taux de collision dépasse un certain seuil et repositionne les éléments dans le nouveau tableau.

Développez le mécanisme inverse, qui rapetisse automatiquement le tableau, à partir d'un critère et d'un seuil que vous définirez.

4 Analyse des performances cpu et mémoire

Sur le modèle de ce que vous avez fait pour les listes, concevez un programme de test de performances destiné à montrer que la recherche d'un élément est en O(1). Quels sont ses paramètres de lancement ?

Développez le programme que vous avez conçu, puis exécutez-le pour vérifier la propriété, au besoin en utilisant `gprof`.

A l'aide de `valgrind massif`, mesurez l'occupation mémoire de votre dictionnaire dans différentes configurations. Comparez avec votre propre évaluation papier, à partir de la structure du dictionnaire.

A l'aide de `valgrind memcheck` ou de l'option `-fsanitize=address` de `gcc`, vérifiez les programmes et services que vous avez écrits allouent et désallouent correctement la mémoire.

5 Parcours des éléments

Les éléments d'une liste peuvent être parcourus du premier au dernier à l'aide du code simple suivant :

```
for (int i = 0; i < taille(liste); i++) {
   printf("element = %s\n", lire(liste, i));
}
```

Il n'en est pas de même avec notre dictionnaire. Pourquoi ?

5.1 Version simple

Pour pallier ce manque, nous complétons l'interface `dictionnaire` pour permettre le parcours de l'ensemble des clés, en ajoutant les deux fonctions suivantes :

- `debuterParcours()` initialise le parcours, ou réinitialise un nouveau parcours,
- `cleSuivante()` retourne la clé suivante dans le parcours, ou NULL si toutes les clés ont été parcourues.

Ainsi, le code simple suivant permettra de lister toutes les clés d'un dictionnaire, et si nécessaire, les valeurs associées :

```
debuterParcours(dictionnaire);

char* cle = cleSuivante(dictionnaire);

while (cle != NULL) {

   char* val = valeur(dictionnaire, cle);

   printf("(cle, valeur) = (%s, %s)\n", cle, val);

   cle = cleSuivante(dictionnaire);

}
```

Quelles sont les informations dont `cleSuivante()` a besoin pour fonctionner ? Ajoutez-les à la structure `dictionnaire`, puis implémentez `debuterParcours()` et `cleSuivante()`. Testez votre implémentation en complémentant votre programme de test fonctionnel.

5.2 Iterateur

Le service implémenté ci-dessus n'autorise qu'un seul parcours des clés à la fois, ce qui est beaucoup trop limitatif pour la plupart des applications. On veut donc développer une version améliorée du service, qui permette un nombre quelconque de parcours simultanés. Cela permettrait par exemple à plusieurs threads de parcourir un même dictionnaire indépendamment les uns des autres, à leur vitesse propre.

On remarque que la limitation vient de ce que les informations de contrôle du parcours n'existent qu'en un seul exemplaire et sont stockées dans la structure `dictionnaire` même. On en déduit que pour permettre plusieurs parcours, il suffit de (i) regrouper ces informations dans une structure propre, distincte de `dictionnaire`, (ii) instancier cette structure pour chacun des parcours à réaliser. Chaque instance, ainsi que les fonctions qui la manipule, constitue ce qu'on appelle un *itérateur*.

Développez et testez cette nouvelle implémentation. Elle met en œuvre la structure `diterateur`, ainsi que les fonctions suivantes :

- iterateur(dictionnaire), qui retourne un nouvel iterateur pour le dictionnaire spécifié,
- cleSuivante(iterateur), qui retourne la clé suivante dans le parcours spécifié.

Le code utilisateur mettra en œuvre ces fonctions de la façon suivante :

```
diterateur diterateur = iterateur(dictionnaire);
char* cle = cleSuivante(&diterateur);
while (cle != NULL) {
   char* val = valeur(dictionnaire, cle);
   printf("(cle, valeur) = (%s, %s)\n", cle, val);
   cle = cleSuivante(&diterateur);
}
```

Noter que pour changer et simplifier un peu, l'itérateur n'est pas alloué dynamiquement et n'a donc pas besoin d'être libéré après usage.