INF2010 - Structures de données et algorithmes

Travail Pratique 2

Tables de hachage

Département de génie informatique et logiciel

École Polytechnique de Montréal

Été 2020

Objectifs

- Apprendre le fonctionnement d'une table de hachage
- Comprendre la complexité asymptotique d'une table de hachage
- Utiliser une table de hachage dans un problème complexe

Pour ce laboratoire, il est recommandé d'utiliser l'IDE IntelliJ offert par JetBrains. Vous avez accès à la version complète (Ultimate) en tant qu'étudiant à Polytechnique Montréal. Il suffit de vous créer un compte étudiant en remplissant le formulaire au lien suivant:

https://www.jetbrains.com/shop/eform/students

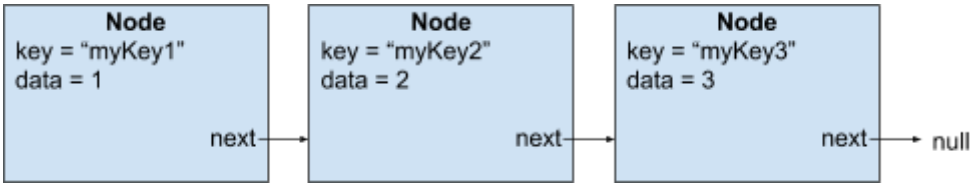
La correction du travail pratique sera partiellement réalisée par les tests unitaires implémentés dans les fichiers sources fournis. La qualité de votre code ainsi que la performance de celui-ci (complexité asymptotique) seront toutes deux évaluées par le correcteur. Un barème de correction est fourni à la fin de ce *.pdf.

ATTENTION! ATTENTION! ATTENTION!
Pour ceux qui voudraient déposer leur laboratoire sur **GitHub**, assurez-vous que vos répertoires soient en mode **privé** afin d'éviter la copie et l'utilisation non autorisée de vos travaux. **Un répertoire public peut mener à une sanction de plagiat**.

Partie 1 : Implémentation d'une table de hachage

Une table de hachage est une structure de données qui utilise une fonction de dispersement pour donner une valeur numérique à une clé qui peut être d'un type quelconque (string, int, MyCustomClass, ..). Cette valeur numérique retournée par la fonction de dispersement est utilisée comme indice dans un tableau, ce qui nous donne un opération d'accès en O(1).

Il arrive que la fonction de dispersement retourne la même valeur numérique pour deux clés différentes. Ce phénomène nommé « collision » est un problème connu des tables de hachage et plusieurs techniques existent pour pallier à ce problème. Dans le cadre de ce laboratoire, l'utilisation de listes chaînées nous permettra de gérer les collisions. La classe «Node» contenue dans HashMap.java vous permettra de créer des listes chaînées de la manière suivante :



Pour bien implémenter ladite table de hachage, suivez les tests contenus dans HashMapTester.java. Aussi, n'oubliez pas d'utiliser *hash(KeyType key)* comme fonction de dispersement.

Une note de 0 sera attribuée à cette partie si l'étudiant utilise une table de hachage déjà implémentée provenant d'une librairie quelconque.

Partie 2 : Problème typique d'entrevue

Le problème que vous aurez à résoudre s'intitule «Matching Pairs».

Entrées

- Collection de nombres entiers (*values*)
- Entier (*targetSum*)

Sortie

Collection de toutes les paires sommant à *targetSum* parmi les valeurs dans *values* en excluant les permutations

Contraintes

- La collection en sortie ne doit contenir aucune permutation
- La collection en sortie doit contenir toutes les combinaisons existantes
- *values* peut contenir des doublons
- *values* peut contenir des entiers négatifs

Pour bien implémenter l'algorithme, suivez les tests contenus dans InterviewTester.java.

Il est permis d'utiliser la librairie java.util pour cette partie. Une note de 0 sera attribuée à cette partie si l'étudiant utilise quelconque autre librairie.

Permutations et combinaisons

Pour définir ces deux concepts, prenons un exemple tiré de l'algorithme que vous devez implémenter.

Supposons *values* = $\{4_1, 4_2, 6_1\}$ et *targetSum* = 10

Réponse sans permutation : $\{(4_1, 6_1), (4_2, 6_1)\}$

Réponse avec permutations : $\{ (4_1, 6_1), (6_1, 4_1), (6_1, 4_2), (4_2, 6_1), \}$

Par définition, **une combinaison est un ensemble de valeurs non ordonnées**, c'est donc dire que les paires $(4_1, 6_1)$ et $(6_1, 4_1)$ sont en fait la même combinaison, soit l'ensemble réunissant 4_1 et 6_1.

Par définition, **une permutation est un ensemble de valeurs ordonnées**, c'est donc dire que les paires $(4_1, 6_1)$ et $(6_1, 4_1)$ forment deux permutations différentes.

Dans la réponse avec permutations, il est possible d'observer qu'il s'agit en fait de la réponse sans permutation additionnée de la permutation (ordre différent) de chacune de ses réponses.

Barème de correction

Partie 1	Réussite des tests	/11.5
Partie 2	Réussite des tests	/4
	Complexité en temps	/3.5
Qualité du code		/1
		/20

Un chargé s'assurera que votre code ne contourne pas les tests avant de vous attribuer vos points dans la catégorie «Réussite des tests». Il est important de respecter les complexités en temps mises dans la description de chaque fonction. **Une fonction n'ayant pas la bonne complexité entraîne la perte des points de tous les tests utilisant cette fonction.**

Pour avoir tous les points dans la catégorie « Complexité en temps » de la partie 2, vous devez réaliser un algorithme respectant les commentaires situés dans InterviewTester.java.

Qu'est-ce que du code de qualité ?

- Absence de code dédoublé
- Absence de *warnings* à la compilation
- Absence de code mort
- Uniformité des conventions de codage pour tout le projet
- Variables, fonctions et classes avec des noms explicitant l'intention et non le comportement

Instructions pour la remise

Veuillez envoyer les fichiers .java contenant le code source utile à la résolution des parties 1 et 2. Minimalement, les fichiers suivants devraient faire partie de votre remise :

- HashMap.java
- Interview.java

Vos fichiers devront être compressés dans une archive *.zip. Le nom de votre fichier compressé devra respecter la formule suivante où MatriculeX < MatriculeY :

inf2010_lab2_MatriculeX_MatriculeY

Chaque jour de retard créera une pénalité additionnelle de 20%. Aucun travail ne sera accepté après 4 jours de retard.