UNIVERSITÉ DE SHERBROOKE
DÉPARTEMENT D'INFORMATIQUE

IFT 339

Travail pratique#5 : Tables de hachage

Ce travail a pour objectif de vous familiariser avec les tables de hachage et leur utilisation. Quelques modalités :

— Ce travail compte pour **7 %** de la session.

— La date de remise de ce travail est le **lundi 9 décembre** jusqu'à 23h59.
Vous devez remettre sur `https://turnin.dinf.usherbrooke.ca/`.

Partie 1 : dictionnaire avec table de hachage

Vous devez implémenter un dictionnaire avec une table de hachage, avec adressage ouvert. Vous devez utiliser la représentation qui vous est fournie. Nous y utilisons une structure alveole qui contient une clé et sa valeur associée. La table est constituée d'un tableau d'alvéoles allouées dynamiquement (donc de type alveole**). Notez que ce n'est pas un tableau 2D, mais un tableau de pointeurs. Une alvéole bidon fait partie de la classe et sert à identifier les alvéoles qui ont subit une suppression. Donc, étant donné un indice h, on a l'interprétation suivante :

— si (table[h] && table[h] != bidon), alors table[h] contient une vraie paire clé-valeur ;

— sinon, si (table[h] == bidon), alors il n'y a pas d'élément à la position h, mais il y en a eu un dans le passé et il a été supprimé. On peut insérer dans cette alvéole ;

— sinon, si (!table[h]), alors il n'y a pas d'élément à la position h et il n'y en n'a jamais eu.

Fonctions à implémenter
Pour une clé cle, on dénote par hash(cle) l'adresse déterminée par la fonction de hachage. L'ensemble des méthodes publiques à implémenter est le suivant :

- inserer(cle, val) : si cle n'existe pas déjà, créé une alvéole contenant la clé associée à val et retourne true. Vous insérez la paire à l'adresse adr = hash(cle) % cap si elle est disponible (nullptr ou bidon). Sinon, vous insérez à l'adresse adr + 1 si disponible, sinon à adr + 2, et ainsi de suite jusqu'à ce qu'une case soit disponible (en revenant au début de la table à l'indice 0 si nécessaire). Si la clé est déjà présente, vous ne devez pas insérer et retourner false.

  Note. Vous devez aussi gérer le facteur de charge. Si nbelem/cap devient trop gros, vous devez **doubler la capacité** de votre table. Ceci demande de créer une nouvelle table puis d'y réinsérer toutes les paires clé-valeur. À vous de choisir ce que "trop gros" veut dire.

- contient_cle(cle) : retourne true si la clé en argument est présente dans votre structure, et false sinon. Vous devez chercher à l'adresse de la clé, et si elle n'y est pas, vous continuez de chercher les adresses suivantes jusqu'à ce que vous atteignez une case vide à nullptr.
- supprimer(cle) : si la clé existe, supprime l'alvéole contenant la clé et retourne true. Le contenu de cette alvéole doit être remplacé par bidon. Si la clé n'ést pas dans la table, vous devez retourner false.
- operator[cle] : l'opérateur reçoit une clé et retourne une référence vers la valeur associée à la clé. Si la clé n'est pas présente, vous devez insérer la clé et l'associer à une valeur créée avec le constructeur par défaut (c'est le comportement adopté dans la SL). Pour créer un objet de type TYPEVAL par défaut, il vous suffit d'écrire TYPEVAL v = TYPEVAL();. L'opérateur doit retourner la valeur par référence pour pouvoir réaffecter une valeur associée à une clé. Voici des exemples d'utilisation :

```
hashdict<string, int> dict;
dict["xyz"] = 10;        //insere "xyz" avec 10
dict["xyz"] = 30;  // associe maintenant "xyz" avec 30
dict["abc"]; //insere "abc" et l'associe a un int par defaut
bool b = dict.contient_cle("abc");    //retournera true
```

- clear() : supprime tous les éléments, nettoie la mémoire et met la capacité à une valeur par défaut.

Vous devez aussi implémenter un constructeur et un destructeur. Vous pouvez ajouter de nouvelles méthodes privées au besoin, mais pas de méthode publique ou nouvelles variables membres.

Vous n'avez pas besoin de vous préoccuper des fonctions de hachage. Vous pouvez prendre pour acquis que l'utilisateur les a déjà définies. Pour transformer une clé de type TYPECLE en adresse, il vous suffit de faire size_t h = hash<TYPECLE>()(cle);, suivi d'un modulo pour rester dans la taille de votre table. En utilisant le *namespace* std, hash sera déjà défini pour les types standards tels que int, size_t, long, string, pointeur, etc. Notez toutefois que la fonction hash ne connaît pas la taille de votre tableau, et retourne un entier entre 0 et l'entier maximum.

Partie 2 : anticipation de dénis de services

Vous voulez provoquer un déni de service sur un serveur qui gère des requêtes [1]. Dans vos requêtes, vous spécifiez un identifiant, qui est un entier entre 0 et $n-1$. Quand le serveur reçoit la requête, l'identifiant sert de clé dans une table de hachage en adressage chainé avec une capacité m. Vous savez que vous serez bloqué après k requêtes, et vous voulez choisir les k requêtes qui ralentissent le serveur au maximum en provoquant des collisions. Pour ce faire, vos k identifiants doivent se retrouver dans un nombre *minimum* d'alvéoles. Avant d'envoyer vos requêtes, vous allez faire des tests pour trouver les k pires insertions possibles.

1. Ou dans la version gentille, vous gérez le serveur et vous voulez tester sa résistance aux dénis de service.

Vous devez donc implémenter une fonction qui reçoit n, m, k où :
– les clés possibles sont les nombres entre 0 et $n-1$;
– m est la capacité de la table de hachage. Ceci veut dire que si on insère la requête avec la clé i, elle va se retrouver dans l'alvéole à l'index `hash<int>()(i) % m` ;
– k est le nombre d'insertions que vous pouvez faire.

Vous devez retourner le nombre minimum d'alvéoles qu'il est possible de remplir avec k clés parmi les n possibles.

Prenons un exemple. Le tableau suivant suppose que $n = 10$ et $m = 4$. Il montre une valeur de hachage hypothétique pour chaque id possible, ainsi que l'adresse dans laquelle chaque id se trouvera (colonne 3).

id	hash(id)	hash(id)%m
0	9	1
1	7	3
2	2	2
3	3	3
4	6	2
5	13	1
6	3	3
7	15	3
8	8	0
9	5	1

Si $k = 3$, alors la fonction retournerait 1, car on peut choisir les clés $1, 3, 6$ qui se retrouveront dans une seule alvéole. Si $k = 5$, la fonction retournerait 2 car le plus petit nombre d'alvéoles qu'on peut remplir avec 5 insertions est 2 (par exemple avec les clés $1, 3, 6, 7, 2$). Si $k = 6$, vous devriez vous convaincre qu'on retournerait aussi 2.

Suggestion. Faites une boucle de 0 à $n-1$ pour obtenir une structure qui, pour chaque adresse de 0 à $m-1$, connaît le nombre de clés qui vont à cette adresse. Mettez ensuite les comptes dans une structure que vous pouvez trier en ordre décroissant avec

```
#include <algorithm>
...
std::sort(mastructure.begin(), mastructure.end(), std::greater<>());
```

Évaluation

Comme pour les autres devoirs, vous serez pénalisés pour les erreurs et imprécisions selon les critères suivants : Respect des spécifications (45 points) ; Qualité du code (45 points) ; Respect des normes de programmation (10 points).

Remise du travail

Vous devez remettre les fichiers noeud.h et main.cpp. Ne remettez pas d'autre fichier ni surtout d'exécutable. Il ne doit y avoir qu'une seule remise par équipe, à partir d'un seul CIP. Les noms des co-équipiers doivent être clairement indiqués en entête de chaque fichier soumis. Notez qu'il n'y a pas de script de correction automatique - votre code sera inspecté à la correction.