ESISAR
CS353 - Algorithmique
TP numéro 3 et 4
Table de hachage

Table des matières

1 Objectifs des TDM 3 et 4

Les TDM 3 et 4 vont permettre l'implémentation d'un gestionnaire de stock d'un magasin, à l'aide d'une table de hachage.

2 Présentation du gestionnaire du stock

Nous considérons un magasin qui contient une liste de produits. Chaque produit est identifié par 3 champs :

- son code sur 5 chiffres
- son libellé (sur 32 caractères maxi)
- son prix en euros (de 0.00 à 999.99)

Exemple d'un magasin contenant 4 produits :

Code	Libellé	Prix en euros
10001	Sucre	1,20 €
10002	Farine	0,80 €
10003	Confiture fraise	2,50 €
10004	Sel	0,50 €

Deux produits sont identiques si ils ont le même code. Deux produits sont différents si le code est différent.

Dans ce TP, nous allons développer un programme qui va permettre :

- d'afficher tous les produits gérés par le magasin
- d'afficher le prix pour un produit donné
- d'ajouter des produits
- de supprimer des produits
- de modifier le prix ou le libellé d'un produit existant
- de réorganiser la table de hachage
- de rechercher la liste des produits contenant une certaine chaîne de caractère

Ce programme devra être extrêmement rapide, car il sera utilisé par les caisses du magasin.

Pour garantir un temps d'accès très rapide (typiquement en O(1)), ce programme utilisera une table de hashage contenu dans un tableau en mémoire.

Ce tableau en mémoire sera initialisé au démarrage du programme et accessible par une variable globale. La taille du tableau est fixe et ne peut pas être modifiée. Toutes les entrées du tableaux seront initialisés au démarrage du programme.

Dans notre implémentation, nous considérons également que le magasin contient habituellement environ 500 produits. Nous fixons la taille du tableau à **m=1009** éléments.

Pour chaque entrée du tableau, trois cas seront possibles :

- l'entrée du tableau est vide (NULL_ITEM)
- l'entrée du tableau contient un élément supprimé (DELETED_ITEM)
- l'entrée du tableau contient un élément standard

3 Implémentation du gestionnaire du stock

Question 1

Récupérer le code initial fourni dans Chamilo (Aller dans Documents/Travaux pratiques puis fichier TP34-hashage.tar). A lire puis à compiler.
Pour décompresser, utiliser la commande : `tar xvf TP34-hashage.tar`

A noter :

- la structure Item est fournie
- le tableau en mémoire est déclaré en tant que variable globale et est initialisé dans la fonction init() (qui est fournie)
- tous les éléments du tableau dont le code du produit est égal à -1 correspondent à des entrées libres de la table de hashage, tous les éléments du tableau dont le code du produit est égal à -2 correspondent à des entrées supprimées de la table de hashage

- vous placerez vos tests dans le fichier main.c, un jeu d'essai minimaliste est aussi fourni (celui de l'exemple du paragraphe 3)

Question 1 - Fonction de hachage

Nous utiliserons la fonction de hachage suivante (double hachage à adressage ouvert)

h(k, i) = (h1(k) + i * h2(k)) mod m

avec :
- k : le code du produit
- i : le nombre d'essai de recherche
- m : la taille du tableau (la table de hachage)

et
- h1(k) = (k mod m)
- h2(k) = 1 + (k mod (m-1))

Implémenter la fonction :

```
int hashkey(int itemCode,int nbTry) ;
```

Question 2 - Fonction d'insertion

Implémenter la fonction :
```
int insertItem(int itemCode, char *itemName, float itemPrice) ;
```

Cette fonction insère le produit indiqué dans la table de hachage.
Si le produit est inséré avec succès, alors la fonction retourne SUCCESS (0)
Si le produit existe déjà dans la table, alors la fonction retourne INSERT_ALREADY_EXIST (-1), et la table de hachage n'est pas modifiée
Si la table est pleine, alors la fonction retourne TABLE_FULL (-2).

Vous veillerez à prendre en compte la gestion des éléments supprimés.

Question 3 – Suppression d'un produit

Ecrire la fonction de suppression d'un produit du magasin
```
int suppressItem(int itemCode);
```

Si le produit est supprimé avec succès, alors la fonction retourne SUCCESS (0)
Si le produit n'existe pas, alors la fonction retourne DELETE_NO_ROW (-4)

Question 4- Affichage

Ecrire la fonction d'affichage de tous les produits du magasin, dans l'ordre de la table de hashage :
```
void dumpItems();
```

Pour chaque produit, vous devez afficher sur une ligne
- le code du produit
- son libellé
- son prix
- son index dans la table de hashage

Réaliser un formatage soigné, pour obtenir un résultat de cette forme :

```
CODE  LIBELLE                          PRIX  INDEX
10001 Sucre                            1.20  920
10002 Farine                           0.80  921
10003 Confiture fraise                 2.50  922
10004 Sel                              0.50  923
```

Réaliser des tests complets des questions 1,2 et 3. En particulier, vous devez essayer de remplir complètement la table, de la vider complètement, et de vérifier que celle ci fonctionne encore.

Question 5 – Affichage du prix pour un produit donné

Ecrire la fonction de recherche :

```
float getPrice(int itemCode);
```

Cette fonction retourne le prix du produit dont le code est itemCode.
Cette fonction retourne SELECT_NO_ROW (-3) si le produit n'existe pas.

Question 6 : Modification du prix et du libellé d'un produit existant

Ecrire la fonction de mise à jour d'un produit :

```
int updateItem(int itemCode, char* itemName, float itemPrice) ;
```

Si le produit est mis à jour avec succès, alors la fonction retourne SUCCESS (0)
Si le produit n'existe pas, alors la fonction retourne UPDATE_NO_ROW (-5)

Question 7 – Réorganisation de la table in situ

Comme nous l'avons vu, lorsque la table se remplit, le nombre de collisions augmente et la performance se dégrade. En particulier, si la table contient beaucoup d'éléments supprimés, les séquences de sondage deviennent très longues.

Quand la table est saturée d'éléments supprimés, il faut la réorganiser. Nous proposons ici de réaliser un redimensionnement **in situ**.

Le redimensionnement **ex situ** est la solution la plus facile: il s'agit seulement de créer une autre table de plus grande dimension et d'y transférer toutes les clefs contenues dans la table originale. Il suffit en effet de balayer la premier table, et, pour chaque clef trouvée, l'insérer dans la nouvelle table. Une fois l'opération terminée, on substitue la nouvelle table à l'ancienne, et le tour est joué.

C'est simple, efficace, mais nécessite d'allouer de la mémoire pour les deux tables

simultanément. Dans certains cas, ceci représente une trop importante utilisation de mémoire.

Il nous reste donc à redimensionner **in situ**, c'est-à-dire à l'intérieur de la table même. On positionne sur chaque clef dans la table un bit pour indiquer si la clef « sale » ou « propre. » Au début, toutes les clefs sont sales. On balaye le tableau jusqu'à la prochaine clef sale et on la re-hash pour lui trouver une nouvelle location. Si elle atterrit dans une case vide, sa réinsertion est terminée. Si elle atterrit sur une case sale, on échange la clef sale avec la clef maintenant propre, et on réinsère la clef que nous venons de retirer du tableau. Si la clef atterrit sur une clef propre, on utilise la résolution de collision pour lui trouver une nouvelle adresse.

Ecrire la fonction de réorganisation in situ :
```
void rebuildTable() ;
```

Question 8 – Recherche par libellé

On va mettre en place une nouvelle structure pour la récupération des résultats :

```
typedef struct structResult
{
        Item *item;
        struct structResult *next;
} Result;
```

Faire une fonction simple de recherche des produits par libellé :
```
Result *findItem(char* itemName);
```

Cette fonction retourne une liste chaînée avec des pointeurs vers tous les produits dont le libellé est égal à itemName.
Exemple : si il y a trois produits 1 - « Sel » , 2 - « Sel» et 3-« Confiture », alors
```
findItem(« Sel »)
```
retourne les deux produits 1-« Sel » et 2-« Sel ».

Que pouvez vous dire du temps de recherche de votre fonction ?

Question 9 – Mise en place d'un index pour recherche par libellé.

On souhaite obtenir un temps de recherche très rapide à partir du libellé (en O(1)).
Pour obtenir cela, nous proposons de déclarer une deuxième table de hachage, qui sera un index pour la recherche à partir de libellé.

Le principe est le suivant :

- la deuxième table de hachage a pour taille M, elle contient des pointeurs vers les Item déclarés dans la première table de hachage

- pour faire la recherche, le libellé est transformé en un nombre N (on pourra utiliser `hashIndex fourni ci-dessous)`.

- Ce nombre N sera utilisé avec la fonction de hachage de la question 1 afin de fournir un nombre compris entre 0 et M-1, et index[hash(N)] donne directement un pointeur vers le produit. Attention, si deux produits donnent le même nombre N (si hashIndex collisionne

sur deux libellés distincts, ou que deux produits ont le même libellé), Il faut résoudre les collisions comme ce qui a été fait dans le début de l'exercice.

Vous devez :

- utiliser une fonction de hachage pour le libellé (voir ci dessous)
- créer la fonction

```
Result *findItemWithIndex(char* itemName);
```

qui fait la recherche en utilisant l'index, et qui retourne une liste chainée de pointeurs vers les items

- mettre à jour les fonctions insertItem, updateItem, deleteItem pour que celles ci mettent à jour l'index

Pour transformer le libellé en nombre, diverses fonctions existent, nous proposons celle ci :

```
unsigned int hashIndex(const char *buffer, int size)
{
     unsigned int h = 0;
     for (int i=0; i<size; i++)
     {
          h = ( h * 1103515245u ) + 12345u + buffer[i];
     }
     return h;
  }
```