IMA 3ème année
Programmation Avancé

TP9 Table de hachage, utilisation de bibliothèque

1 Objectifs

— Savoir utiliser une bibliothèque
— Savoir construire une table de hachage et l'utiliser

Contexte et préparation : Dans ce TP, nous allons construire et utiliser une table de hachage. Les tables de hachage étant classiquement utilisées pour coder des dictionnaires, nous allons réaliser un dictionnaire anglais sommaire.

2 Questions du TP (à faire impérativement)

2.1 Récupération de fichiers

1. Récupérez votre librairie et votre fichier entête du TP7 (`listechaines.h` et `listechaines.a`).
2. Copier les fichiers `hash.c`, `dico.english` et `Makefile` fournis dans `~wrudamet/public/IMA3/TP9/`
3. Vérifier que le Makefile fonctionne : commande `make`.
4. Lancer le programme (que fait-il ?) et noter son temps d'exécution (commande `time`).

2.2 Explications tables de hachage

Lire attentivement l'annexe.

2.3 Travail demandé

Dans le fichier fourni `hash.c` :

1. Déclarer le type `Hashtable` (vecteur de taille `TABLE_SIZE` de `Liste`).
2. Écrire une fonction `void init_ht(Hashtable ht)` qui initialise `ht` (listes à `NULL`).
3. Écrire une fonction `void update_ht(char *word, Hashtable ht)` qui ajoute `word` dans `ht` (fonction `ajout_alphab` fournie) à la liste d'indice `hash(word)`. La fonction de hachage est fournie dans le `.c`.
4. Écrire une fonction load_ht(FILE *f, Hashtable ht) analogue à `charge_fichier` qui charge les mots du fichier dans ht.
5. Modifier la fonction `main` en remplaçant la liste par une Hashtable.
6. Comparer les temps d'exécution avec la liste (version précédente) [1]. Expérimenter avec des valeurs progressives de `TABLE_SIZE` :
   `TABLE_SIZE = 1` (vous devriez retrouver des performances très comparables)
   `TABLE_SIZE = 10, 20, ..., 100, 200, 300, 400`
7. Ecrire une fonction `void collisions(Hashtable ht)` qui affiche, pour chaque indice `i`, la taille de la liste `ht[i]`. Sur l'exemple, (avec `TABLE_SIZE=50`) :

```
0 335
1 309
2 308
3 318
...
```

1. N'oubliez pas de comparer ce qui est comparable en mettant en commentaire les traces ou affichages éventuels.

3 Questions s'il vous reste du temps

1. Rediriger la sortie standard lors de l'exécution pour écrire le résultat de la fonction collision dans un fichier `trace.txt` (mettez en commentaire tout autre affichage). Visualiser la répartition des collisions avec `gnuplot` et faire de même avec différentes valeurs de `TABLE_SIZE`.

   ```
   ./hash dico.english > trace.txt
   gnuplot
   gnuplot> plot "./trace.txt" with lines
   ```

   Remarquer que la courbe ne change plus après `TABLE_SIZE = 250` (environ).

2. Déterminer la plus grande valeur de hachage possible du dictionnaire : Écrire une fonction : `void max_hash(FILE *fp, char *max_word, int *hmax)` qui détermine `hmax`, la plus grande somme de codes ascii des mots du dictionnaire, et fournit le mot correspondant dans `max_word`.

3. Afficher `max_word` et `hmax`. Fixer `TABLE_SIZE` à `hmax+1`, vous devez retrouver `max_word` en affichant la liste de collisions d'indice `hmax`.

4 Annexes : Tables de hachage

Une table de hachage est un compromis entre les listes chaînées efficaces sur les ajouts et les vecteurs (ou listes contiguës) efficaces sur les accès. Soit un ensemble fini D de données (ici les mots du dictionnaire) que l'on va stocker dans un vecteur ht. Pour ranger un élément x de D, on calcule son image par une fonction de hachage *hash*, qui donne son indice (appelé *indice de hachage*) dans ht, x sera donc rangé dans $ht[hash(x)]$. Il reste à trouver une fonction *hash* adéquate :

— L'idéal est de trouver une fonction **bijective** entre D et l'intervalle d'indices du vecteur, ainsi chaque case du vecteur contient un élément de D. Mais il est difficile de trouver une telle fonction bijective, ne serait-ce que parce que D est rarement connu a priori.

— L'utilisation d'une fonction **injective** (ie : qui associe à chaque x de D une case différente de D) mène à un vecteur de taille $sup\{hash(x), x \in D\}$, c'est à dire un vecteur à trous, ce qui peut générer une perte importante de place.

— Le principe consiste alors à prendre une fonction **surjective** obtenue généralement par modulo sur une taille limitée de vecteur, soit $TABLE\_SIZE << card(D)$. On tombe alors sur un problème de *« collisions »* parce que plusieurs données peuvent avoir le même indice de hachage. On choisit donc une structure mixte constituée d'un vecteur statique de listes chaînées contenant les éléments du même indice de hachage (appelées *« listes de collisions »*). Si possible, ces listes sont ordonnées pour optimiser les accès. L'ajout d'un élément x revient alors à calculer $hash(x)$ de coût quasi constant et insérer x dans la liste $ht[hash(x)]$, en $O(n)$, n étant la taille de la liste correspondante. Toute la difficulté est de trouver un bon compromis entre la taille du vecteur et taille des listes. Remarquez que prendre $TABLE\_SIZE = 1$ revient à faire une simple liste.