TP 12 : Histogramme d'un texte par table de hachage

Notions : Tableaux de Listes, fonctions de hachage

1 Objectif

La notion de table de hachage va être utilisée pour réaliser l'histogramme d'un texte. Vous disposerez de plusieurs textes de M. Proust, G. Sand et A. Musset, H Beyle dit Stendhal, J Renard, A Dumas. L'histogramme est réalisé grâce à une table d'association ou table de hashage, dont la clé est un mot et la valeur le nombre d'occurences de cette clé dans le texte étudié. Vous pourrez comparer l'efficacité d'une table de hashage à celle d'une liste classique, pour laquelle nous fournissons le code. Vous afficherez aussi le mot le plus courant de ces textes.

2 Introduction

Le problème pratique sous jacent à ce TD est celui de la recherche **rapide** d'un élément quelconque dans une grande quantité d'informations. Des exemples sont :

- dictionnaire : retrouver la définition d'un mot dans un dictionnaire ;
- annuaire inversé : retrouver le nom d'une personne à partir de son numéro de téléphone ;
- histogramme : compter le nombre d'occurrences de chaque mot de la langue française dans un texte.
- etc.

Ce problème se formalise donc de la manière suivante : une clé (le mot, le numéro de téléphone dans les exemples précédents) donne accès à une donnée ou information utile (l'existence, la définition ou la traduction du mot pour un dictionnaire, le nom de l'appelant correspondant au numéro de téléphone, le nombre de fois où le mot est utilisé). On parle alors d'adressage associatif. Il s'agit alors de rechercher la clé dans une structure de données, en minimisant le nombre de comparaisons à faire pour savoir si la clé est présente ou non, puis avoir accès à sa définition. Notons que l'espace des clés (ensemble des mots, ensemble des numéros de téléphones possibles) est largement supérieur à celui des clés utiles (mots possibles, numéro possible des abonnés).

Le type abstrait le plus adapté et efficace pour ce type de situations est la table de hachage.

3 Fonction de hachage

Le rôle de la fonction de hachage est de calculer un indice entier dans l'intervalle $[0..M-1]$, avec M la dimension de la table, à partir d'une clé (c'est à dire à partir d'un mot, dans le cas de notre histogramme). Cet indice est appelé indice de hachage.

La première propriété voulue pour la fonction de hachage est qu'elle répartisse les indices le plus uniformément possible sur l'intervalle $[0..M-1]$, où M est la dimension de la table. Cela permet de réduire les collisions : il y a une collision quand 2 mots différents, qui donc n'ont pas la même information associée, ont le même indice de hachage. La deuxième propriété voulue pour la fonction de hachage est sa rapidité de calcul.

Si le mot est la chaîne de caractères $c_0c_1 \dots c_k$, on utilisera dans ce TP la fonction suivante :
$h(c_0c_1...c_k) = (c_0 + a * c_1 + a^2 * c_2... + a^k * c_k)\%M$

Remarques

— a ne doit pas être une puissance de 2. En java, a=31 ;

— un char est un octet en C, i.e. un nombre sur 8 bits : les 2 syntaxes `c ='a'` et `c=64` sont équivalentes en C. Il n'y a donc aucune conversion à réaliser entre un caractère et un entier pour le calcul de la fonction de hashage.

— Attention à la déclaration des paramètres et des variables servant effectivement à calculer le hash-code, car l'arithmétique est signée par défaut. Nous avons supposé que les données sont positives, ce qui implique que les caractères `c` doivent être *castés* en type `unsigned char` par une instruction de type `(unsigned char)c[i]`, et `a` est de type `unsigned long`. Nous manipulons alors des données positives.

— Le prototype de la fonction qui calcule l'indice de hachage sera :
`unsigned int hachage(char* mot, int dim_tab_hach) ;`

— Attention à l'opérateur modulo : `-2 % M` donne `-2` en C.

— Pour programmer cette fonction, ne pas utiliser la fonction `pow` pour calculer ak car elle est beaucoup trop coûteuse en temps ! Il suffit de remarquer que $c_0 + a * c_1 + a^2 * c_2 \dots + a^k * c_k = (c_0 + a * (c_1 + a * (c_2 + a * (c_3 \dots + a * c_k))))$. Il n'y a qu'une multiplication par a à réaliser pour chaque puissance de a. C'est le principe du schéma de Horner.

— Attention aux dépassements de capacités des entiers : 31^n devient rapidement plus grand que la limite des entiers. Pour calculer efficacement cette fonction, on peut effectuer le calcul suivant, inspiré du schéma de Horner :
$h = c_k$
$h = (h * a + c_{k-1})$
$h = (h * a + c_{k-2})$
$\dots$
$h = (h * a + c_0)$
$h = h\%M$

— Dans la suite, on veut faire abstraction des majuscules. Par exemple, la clé `"DiplOdoCUs"` doit être considérée comme la clé `"diplodocus"`. La fonction `char tolower(char c)` (resp `char toupper(char c)`) permet de convertir le caractère `c` en minuscule (resp majuscule). Il faut inclure le fichier d'entête `ctype.h`. La fonction `int strcasecmp(const char *s1, const char *s2)` compare les deux chaînes en ignorant la casse : elle retourne donc 0 si les deux chaînes sont égales, sans se préoccuper des majuscules/minuscules.

— Dans un premier temps, il est conseillé d'utiliser une fonction de hachage plus simple, comme la somme des caractères de la chaîne, avant de programmer la fonction proposée.

4 Le TAD `hashtable_t`

Dans l'exemple proposé, la table de hashage va être utilisée pour réaliser l'histogramme des mots d'un texte. Les clés sont donc les mots du texte, les valeurs le nombre d'occurrences de ce mot dans le texte.

Pour implanter le TAD `hasttable_t`, vous utilisez le hashage par listes. Nous proposons la définition des types à utiliser pour construire ce TAD `hastable_t`.

4.1 Elément de la liste

Pour la liste d'association, la clé est un mot et la valeur un entier. Un élément des listes de collisions contient donc ces 2 types de données. Les fichiers `element.h` et `element.c` contiennent les définitions de types et les fonctions sur les éléments et sont fournis.

Nous définissons les types :

1. `keys_t` comme une chaîne de caractères `char*`,
2. `value_t` comme un entier `int`,
3. `element_t` comme une structure.

Notez que la chaîne de caractère n'est manipulée que par un simple pointeur. Dans le fichier `element.c` fourni, la fonction `element_new` alloue donc un espace mémoire à l'aide de la fonction `calloc` ; et la fonction `element_delete` libère l'espace mémoire précédemment alloué à l'aide de la fonction `free`.

Voici le fichier `element.h`

```
#ifndef _ELEMENT
#define _ELEMENT

typedef char* keys_t;

typedef int value_t;

typedef struct {
  keys_t key;
  value_t value;
} element_t;


// Retourne un nouvel element initialise avec les valeurs cle et valeur.
// L'espace mémoire pour stocker la cle est alloue dynamiquement.
element_t element_new(keys_t key, value_t value);
// Libère l'espace mémoire alloué pour l'element et retourne une élément vide (empty)
element_t element_delete(element_t e);
// Retourne un clone (meme cle, meme valeur)
element_t element_clone(element_t e);

// Affiche la cle (mot) et la valeur (nombre d'occurrences)
void element_print (element_t e);

// Initialise un couple cle, valeur avec le couple NULL,0
void element_init (element_t* e);
// Retourne un couple cle, valeur avec les valeurs NULL,0
element_t element_empty();
// Teste si le couple cle, valeur est NULL,0
int element_is_empty(element_t e);

// Teste si 2 clés (chaînes de char) sont egales, sans se préoccuper des majuscules et minuscules
int key_equal(keys_t, keys_t);
// Teste si 2 couples sont égaux (meme cle, meme valeur)
int element_equal(element_t*, element_t*);
#endif
```

4.2 Les listes de collisions

Les listes utilisées pour gérer les collisions sont donc des listes simples contenant les couples clés-valeurs définis dans le fichier `element.h`. Les fonctions sont les fonctions classiques du TAD `list_t` et sont déjà écrites dans les fichiers `list.h` et `list.c`.

2 fonctions supplémentaires ont été ajoutées, pour retrouver une clé donnée dans une liste ou supprimer une clé donnée dans une liste. Elles seront utiles pour écrire les fonctions du TAD `hasttable_t`

```
// Trouve la cle k dans la list l et retourne le maillon trouve ou NULL si la cle n'est pas dans la
   liste
list_t list_find_key(keys_t k, list_t l);

//Supprime le maillon de cle k et retourne la liste modifiee
list_t list_delete_key(keys_t k, list_t l);
```

4.3 le TAD `hasttable_t`

Pour implanter le TAD `hasttable_t`, vous utilisez le hashage par listes.

Le type `hasttable_t` est une structure contenant le tableau de listes chaînées `data` ainsi que la taille `size` de ce tableau. Ce tableau de listes doit être alloué et initialisé à la création de la table de hashage.

Les fonctions à écrire sont déclarées dans le fichier `tabhash.h`.

```
#ifndef _TADHASH
#define _TADHASH

#include "element.h"
#include "list.h"

typedef struct {
  int size;     // Taille du tableau de listes de collisions
  list_t* data; // Tableau des listes de collisions
} hashtable_t;

// Fonction de hashage
unsigned int hash(char *s,int n) ;

// Cree et retourne une nouvelle table de m listes vides ;
hashtable_t hashtable_new(int m)  ;

/* Associe la valeur   v   a la cle   k   dans la table t.
 Si la cle   k   n'est pas encore dans la table, ajoute le couple [cle k , valeur v] a la table.
 Si la cle   k   est deja presente dans la table, alors change la valeur associee a cette cle
 Retourne 1 si reussi, 0 sinon
*/
int hashtable_put(keys_t k, value_t v, hashtable_t t);

// retourne 1 (true) si la cle k existe dans la table, 0 sinon.
int hashtable_contains_key(keys_t k, hashtable_t t);

/*  Trouve et retourne la valeur associee a la cle k dans la table.
Si la cle est presente, remplit la variable pointee par pv avec la valeur de la cle et retourne 1.
Si la cle n'est pas presente, retourne 0.
*/
int hashtable_get_value(keys_t k, value_t* pv, hashtable_t t);

// supprime la cle k et la valeur associee de la table t retourne 1 si la suppression est realisee, 0
   sinon
int hashtable_delete_key(keys_t k, hashtable_t t);

// supprime tous les elements, detruit la table et retourne une table vide
hashtable_t hashtable_delete(hashtable_t t);

// Affiche toutes les paires [cle, valeur] contenues dans la table
void hashtable_print(hashtable_t t);

// Analyse des performances de la table et affiche les resultats de l'analyse
void hashtable_analyse(hashtable_t t);

#endif
```

5 Travail à réaliser

L'objectif du TD est de programmer une table de hachage dont les clés sont des chaînes de caractères, les valeurs sont des entiers représentant le nombre d'occurrences des mots d'un texte contenu dans un fichier. Vous pourrez aussi comparer l'efficacité de cette solution par rapport à une solution séquentielle utilisant une simple liste. La table de hachage aura une dimension donnée par une variable lue au clavier et sera donc allouée dynamiquement. Vous pourrez aussi tester l'efficacité de la table en fonction de sa taille.

Le travail consiste donc à écrire les fonctions de base du TAD `hashtable_t`, puis à écrire un programme qui lit les mots dans un fichier texte, recherche le mot dans la table.

Lorsqu'on traite un mot, si il n'est pas encore présent dans la tacle de hachage, il faut insérer le couple [mot, 1 occurence] dans la table. Si le mot est déjà présent dans la table, il faut incrémenter le nombre d'occurrences du mot au sein de la table.

Les questions à préparer a minima avant la séance sont les questions 1, 2, 4, 6, 8 et 9

1. Telechargez le fichier tp12.zip
2. Complétez le fichier `tabhash.c` en écrivant les 6 premières fonctions :

   `hash, hashtable_new, hashtable_contains_key, hashtable_put, hashtable_get, hashtable_print`

   Avant d'écrire ces fonctions, lire leur contrat dans les commentaires du fichier `tabhash.h` !
3. Tester vos fonctions avec le programme fourni dans le fichier `testhashtable.c`, en compilant à l'aide de la commande `make testhashtable`, et en utilisant des tailles de tables différentes. Commentez les fonctions de suppression dans un premier temps.
4. Lisez puis complétez le programme principal donné dans le fichier `histo_texte.c`. Ce fichier contient une fonction `main` qui crée une table de hashage qui sera l'histogramme, lit les mots d'un texte contenu dans un fichier, et doit les intégrer à la table de hashage. Il faut soit ajouter un nouveau couple [mot,1] si c'est la première fois que le mot est rencontré, soit incrémenter la valeur associée au mot sinon. L'algorithme est donné en 1. **Il manque à la fonction `main` les instructions C correspondant aux lignes 6 à 12 de l'algorithme : à vous de les écrire**.

Algorithm 1 Programme principal

```
1:  Allouer dynamiquement la table de hachage (tableau de listes) avec une dimension donnee sur la
    ligne de commande
2:  Ouvrir le fichier dont il faut faire l'histogramme
3:  tant que un mot est lu dans le fichier faire
4:    si le mot n'est pas vide ou un espace alors
5:      Afficher et incrementer le nombre total de mots
6:      si le mot n'est pas deja present dans la table alors
7:        Ajouter le couple mot,1 a la table de hashage
8:      sinon
9:        retrouver la valeur associee au mot dans la table de hashage
10:       Incrementer la valeur
11:       Ajouter le couple mot,valeur a la table de hashage
12:     fin si
13:   fin si
14: fin tant que
15: Afficher la table de hashage
16: Afficher le temps d'execution
17: Liberer la memoire
```

5. Compiler avec la commande `make histo_texte` et utiliser le fichier `test.txt` contenant quelques mots pour vérifier le bon fonctionnement de votre table de hashage.
6. Dans le fichier `tabhash.c`, écrire les fonctions `hashtable_delete_key` et `hashtable_delete`. Tester à nouveau votre code avec le fichier `testhashtable.c`, puis avec le fichier `histo_texte.c` en décommentant les lignes correspondantes.
7. Tester votre code sur des textes de taille plus importantes (quelques Moctets) : *les 3 mousquetaires*, *A la recherche du temps perdu*, *le rouge et le noir*, etc.. Ces fichiers sont disponibles dans le répertoire `/users/prog1a/C/librairie/textes`. Utiliser la fonction fournie `void hashtable_sort_print(hashtable_t t);` qui affiche la table de hashage par ordre croissant des valeurs d'occurences.
8. Afficher le mot le plus fréquent. Pour cela, dans le fichier `tabhash.c`, écrire une fonction `value_t hashtable_get_max(hashtable_t t);`. Cette fonction doit parcourir l'ensemble des listes de collisions et trouver le maxima de toutes les listes.

9. Dans le fichier `tabhash.c`, écrire une fonction `void hashtable_analyse(hashtable_t t)` d'analyse de la table : cette fonction doit calculer et afficher les valeurs suivantes :
   - Moyenne des longueurs des listes non vides
   - Moyenne des longueurs avec les listes non vides
   - Ecart type des longueurs de listes
   - Maxima des longueurs des listes
   - Nombre de listes vides

   L'écart type et le nombre de listes vides sont de bons indicateurs de la bonne/mauvaise répartition de la fonction de hashage. Quelle est la taille optimale de la table ?
10. Faites varier la taille de la table de hachage (utilisez une taille de 100, 1000, 10000, 100000) pour voir son influence sur le temps d'exécution. Vous comparerez aussi ce temps avec celui d'un histogramme réalisé par une simple liste chaînée, disponible dans le fichier `sequentiel.c`. Pour construire le programme `sequentiel`, utiliser la commande `make sequentiel`.
11. Tester d'autres fonctions de hachage, en terme de temps d'exécution et d'efficacité de la table :

$$
\begin{array}{lcl}
hash1(s) = & \sum_{i=0}^{n} s[i].a^{n-i} & avec \;\; a = 33 \;\; ou \;\; a = 131 \\
hash2(s) = & \sum_{i=0}^{n} s[i].a^{n-i-1}.b^{n-i} & avec \;\; a = 378551 \;\; et \;\; b = 63689 \\
hash3(s) = & (s[0].2^{n})\hat{}(S[1].2^{n-1})\hat{}...(S[n].2^{0}) & avec \;\; \hat{} \; operateur \; OU \; exclusif
\end{array}
$$

12. Tracer les courbes du temps de recherche sur un fichier en fonction de la taille de la table de hachage.

6 Annexes : Fonctions diverses fournies

2 fonctions qui peuvent être utiles pour la lecture du fichier. Ces fonctions sont dans les fichiers `divers.h` et `divers.c`.

```
#ifndef _DIVERS
#define _DIVERS
#include <stdio.h>

/* Compte le nombre de lignes d'un fichier : utile pour connaitre la taille du fichier si besoin */
int file_length(char *);

/* Lecture d'un mot dans un fichier texte en supprimant les ponctuations, apostrophe, etc... */
int file_read_word(FILE* , char*) ;
#endif
```

Un fichier `Makefile` est donné pour construire des executables `histo_text` et `sequentiel` à partir des fichiers `histo_text.c` et `sequentiel.c` qui contiennent les programmes principaux pour réaliser un histogramme par table de hashage pour `histo_text.c` et par liste classique pour `sequentiel.c`. Ces programmes nécessitent les fichiers `tadhash.c, list.c, divers.c, element.c`.