TP7: Tables de hachage

MP2I Lycée Pierre de Fermat

Le TP est à rendre pour le mardi 17 janvier avant 22h00. Une archive à télécharger se trouve sur Cahier de Prépa.

Pour ce TP, vous aurez besoin d'utiliser la librairie string.h. En particulier, on rappelle les fonctions suivantes :

- void strcpy(char* dst, char* src) copie le contenu de src dans dst.
- int strcmp(char* str1, char* str2) renvoie :
  - 0 si les chaînes sont égales
  - un entier stricement négatif si str1 vient avant str2 dans l'ordre alphabétique
  - un entier strictement positif si str2 vient après str2 dans l'ordre alphabétique

Remarque 1. Le ***manuel Unix***, accessible avec la commande **man**, est une source d'informations pratiques pour toutes les fonctions existante en C (<stdio.h>, <stdlib.h>, <string.h>, etc...). En particulier, lorsque vous voulez utiliser une fonction f mais que vous ne savez plus exactement comment elle fonctionne, vous pouvez taper dans le terminal : **man f**. Ceci ouvrira une page du manuel concernant la fonction. Par exemple, si vous tapez **man strcmp**, vous obtenez une page commençant par :

```
STRCMP(3)                Linux Programmer's Manual                STRCMP(3)

NAME
       strcmp, strncmp - compare two strings

SYNOPSIS
       #include <string.h>

       int strcmp(const char *s1, const char *s2);

       int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
       The  strcmp()  function  compares the two strings s1 and s2.  The
       locale is not taken into account (for a locale-aware  comparison,
       see  strcoll(3)).   It returns an integer less than, equal to, or
       greater than zero if s1 is found, respectively, to be less  than,
       to match, or be greater than s2.

       The  strncmp()  function  is similar, except it compares only the
       first (at most) n bytes of s1 and s2.
```

Cette page contient la signature de la fonction, ainsi que des fonctions de la même famille, puis une description détaillée, suivi de nombreuses informations diverses (valeur de retour, potentiels codes d'erreur, ...).

Allez lire la page de manuel de strcpy, en particulier la section "BUGS" en bas de la page qui est assez intéressante et montre qu'en informatique, les bugs peuvent causer tout et n'importe quoi : "anything can happen". Autrement dit, d'une exécution à l'autre, un programme buggé ne fera pas la même chose, car le comportement dépendra de l'état précis de la mémoire de la machine, ou du contexte de l'exécution, etc... Il est donc important de programmer de manière à éviter tous les bugs, y compris ceux qui semblent peu probables, et ceux qui semblent inoffensifs.

Les ***preuves*** de programmes et les ***tests*** font partie des méthodes qui permettent de se protéger des bugs. La ***pratique de programmation*** joue aussi un rôle important : bien commenter son code, utiliser des fonctions, segmenter le code, etc... Une pratique de programmation très bonne à avoir est la programmation défensive

Définition 1. La ***programmation défensive*** consiste à toujours considérer que les programmes et les fonctions que l'on écrit seront exécutées par un "adversaire", qui tentera de les faire bugger.

Par exemple, si l'on écrit une fonction calculant la racine carrée, et que l'on écrit dans la documentation que l'entrée doit être positive, on ne pourra pas supposer dans le code que c'est le cas : on doit forcément utiliser assert pour lever une erreur lorsque l'entrée est négative. Gardez en tête la notion de programmation défensive pour ce TP et pour le reste de l'année.

Implémentation de la structure

Le but de cette section est d'implémenter les dictionnaires par table de hachage, et de tester l'implémentation sur des exemples simples.

Les fichiers "hashtable.h" et "element.h" dans l'archive du TP contiennent les types nécessaires, ainsi que les déclarations des différentes fonctions à implémenter. Les fichier "hashtable.c" et "element.c" serviront à implémenter les fonctions. Une fonction de hachage y est déjà implémentée. Notez que les définitions des structures sont dans les ".h", donc les structures sont accessibles en boîte blanche : vous pouvez accéder à leurs attributs depuis l'extérieur.

Question 1. Créez un fichier de test, "test.c", que vous utiliserez pour tester vos fonctions au fur et à mesure que vous avancez dans le TP. A chaque fois que vous écrivez des fonctions, vous devez la tester dans "test.c" avec un bloc de code de la forme :

```
1  int main(){
2    ...
3
4    /* TEST nom_fonction_ou_nom_test*/
5    ...
6    ...
7    ...
8    /* FIN TEST nom_fonction_ou_nom_test */
9
10   ...
11   return 0;
12 }
```

Les jeu de tests pourront être des vérifications à l'aide d'assert, ou, lorsque ce n'est pas possible, un simple affichage.

Vous devez régulièrement lancer votre programme de test, et l'analyser avec valgrind pour éviter les fuites mémoires. A la fin du TP, votre programme de test devra avoir des blocs de test pour toutes les fonctions, ne supprimez rien. Pour le moment, en utilisant assert, ajoutez des tests permettant de vérifier que hash(x, m) renvoie bien y pour :

- $x =$ "bonjour", $m = 101, y = 53$
- $x =$ "voici un texte a hacher", $m = 503, y = 488$
- $x =$ "voici un texte a macher", $m = 503, y = 236$

Question 2. Ajoutez un jeu de tests pour la fonction egal de "element.h", puis implémentez cette fonction dans "element.c".

Question 3. Implémentez les deux fonctions d'affichage restantes dans "element.c".

Question 4. Écrivez un jeu de test pour les chaines. Votre jeu de test doit couvrir tout le code que vous aurez écrit. Pensez à tester des cas limites (chaîne vide ou presque vide, recherche d'un élément non présent, etc...). Implémentez les fonctions pour les chaînes et testez les au fur et à mesure.

Une fois que vous avez bien vérifié que les chaînes sont correctement implémentées et sans fuites mémoires, vous pouvez passer aux tables de hachage

Question 5. Écrivez un jeu de test pour la structure DICT, puis implémentez les fonctions relatives à cette structure, en les testant au fur et à mesure.

Manipulation de texte

Les dictionnaires ont de nombreuses applications dans l'algorithmique du texte (recherche de motif, compression...). Voyons une application simple plus simple sur le nombre d'occurences d'un mot.

On considère le texte intégral de "Vingt Mille Lieues sous les mers" de Jules Verne. On veut savoir quel est le mot le plus utilisé parmis ceux faisant plus de 8 lettres. On utilise pour cela l'algorithme suivant :

Algorithme 1 : Mot le plus fréquent

Entrée(s) : t un texte, K un taille de mot minimale
Sortie(s) : Le mot de longueur $\geq K$ ayant le plus d'occurences dans t

```
1 D ← dictionnaire vide;
2 pour M mot de t faire
3     si |M| ≥ K et M ∈ D alors
4         D[M] ← D[M] + 1;
5     Else D[M] ← 0;
6     Chercher dans D la clé de valeur associée maximale;
```

Question 6. Quelle est la complexité de cet algorithme dans le pire cas ? Et en moyenne en utilisant une table de hachage avec une bonne fonction de hachage ?

On commence par implémenter une fonction utilitaire. Vous ajouterez sa déclaration dans “hashtable.h”, et son implémentation dans ”hashtable.c”. Vous devez la tester dans “test.c”.

Question 7. Le programme à écrire manipulera des chaînes de caractères qui auront été allouées sur le tas. Or, votre fonction de désallocation des dictionnaires ne libère pas a priori les clés et les valeurs. Écrivez une fonction `void free_keys(DICT* d)` qui parcourt `d` et libère toutes ses clés, et uniquement les clés. Pour tester cette fonction, vous devrez utiliser des strings réservés dans le tas. Le code suivant utilise `strcpy`, de la librairie `<string.h>`, pour copier le contenu d'une chaîne constante vers une zone réservée par malloc :

```
1  char* x;
2  x = malloc(8*sizeof(char));
3  strcpy(x, "bonjour");
4  // x contient maintenant "bonjour"
```

Ce schéma de code vous permettra de ne pas avoir à écrire les mots case par case.

Créez maintenant un fichier “manip_texte.c” qui servira à implémenter l'algorithme précédent. Créez-y une fonction main, et une fonction `void test()` où vous mettrez les tests. Pour le moment, votre main devra ressembler à :

```
1 int main(){
2   test();
3   return 0;
4 }
```

Question 8. Écrivez une fonction `char* trouver_max(DICT* d)` renvoyant la clé dont la valeur associée est maximale. Vous pouvez supposer que les valeurs sont toutes positives ou nulles. Le dictionnaire ne doit pas être vide. Écrivez également un jeu de tests afin de tester cette fonction.

Lorsqu'on lit dans un fichier avec fscanf, la valeur renvoyée par fscanf (à ne pas confondre avec les valeurs lues et stockées dans les arguments) correspond au nombre de mots lus lors de l'appel. Si la fonction arrive à la fin du fichier, elle renvoie une valeur spécifique appelée `EOF`. On peut donc lire les mots d'un fichier `f` avec :

```
1 // str doit être un string assez grand pour stocker les mots lus
2 while (fscanf(f, "%s", str) != EOF){
3   /* traitement du mot */
4 }
```

Attention, vous ne pouvez pas vous contenter d'écrire :

```
1 while (fscanf(f, "%s", str) != EOF){
2   inserer(d, str, 1);
3 }
```

car str n'est qu'un pointeur, et donc on n'insère qu'une référence vers la zone mémoire où la chaîne `str` est stockée. Il faut réserver de la mémoire à chaque fois qu'on lit un mot pour le stocker. Par exemple :

```
1 while (fscanf(f, "%s", str) != EOF){
2   char* mot = malloc((strlen(str) + 1)*sizeof(char)); // allouer de 'espace
3   strcpy(mot, str); // copier le mot lu dans `mot`
4   inserer(d, str, 1);
5 }
```

Dans ce code, on écrase à chaque fois le pointeur `mot`, mais ça ne crée pas directement de fuite mémoire, car on a gardé une copie du pointeur à l'intérieur du dictionnaire `d`. En quelque sorte, on a donné à `d` la responsabilité de la mémoire réservée, et c'est donc avec `free_keys` que l'on pourra libérer la mémoire réservée par une telle boucle.

Question 9. Écrivez une fonction void mot_plus_frequent(char* filename, int K) qui trouve le mot le plus fréquent dans le fichier filename, parmis ceux ayant une longueur au moins K, et l'affiche ainsi que son nombre d'occurences. La fonction affichera aussi le nombre total de mots dans le texte. Créez plusieurs petits fichiers permettant de tester cette fonction (pensez aux cas limites). Pour la taille de la table de hachage, vous définirez une variable globale, que vous pouvez fixer à 5000 pour le moment.

Question 10. Une fois que tous vos tests fonctionnent et qu'il n'y a pas de fuite mémoire, vous pouvez commenter la ligne du main qui lance les tests. Complétez maintenant le main pour obtenir un programme prenant en ***argument*** :

- Un nom de fichier fn
- Une borne K
- Une taille de table m

Ce programme affichera le mot d'au moins K lettres le plus fréquent dans le fichier fn spécifié, en utilisant une table de hachage avec m alvéoles. Si m n'est pas spécifié, on prendra $m = 15319$. Si le programme n'est pas exécuté avec le bon nombre d'argument, il affichera un message explicatif avant de s'arrêter.

Question 11. Quel est le mot de plus de 8 lettres le plus utilisé dans "Vingt Mille Lieues sous les mers" ? Combien d'occurences a t'il ? Et pour les mots de plus de 2 lettres ?

Étudions maintenant les performances de notre programme. On pose $K = 0$, autrement dit on cherche simplement le mot le plus fréquent. Étudions le comportement du programme selon la taille m choisie pour les tables de hachage.

Question 12. La méthode naïve pour résoudre ce problème consiste à stocker les mots et leur nombre d'occurence dans une liste chaînée, autrement dit consiste à utiliser un dictionnaire par liste chaînée. Si l'on réduit la taille de notre dictionnaire à une seule alvéole, on obtient donc la version naïve. Comparez le temps d'exécution pour $m = 1$, $m = 100$, $m = 10000$, $m = 1000000$, où m est le nombre d'alvéoles.

Question 13. Donnez la taille moyenne des alvéoles non-vides pour $m = 500$, $m = 1024$, $m = 1025$, $m = 8191$, et $m = 8192$. Que constatez-vous ? Commenter.

La dernière question est en bonus, pour aller un peu plus loin :

Question 14. Écrivez une fonction void mots_plus_frequents(char* filename, int K, int n_mots) qui affiche les n_mots mots les plus communs dans le fichier filename, ainsi que leurs nombres d'occurences, parmis ceux de longueur au moins K. La fonction devra être correcte même s'il y a moins de n_mots mots de longueur K.