TP8: Tables de hachage

MP2I Lycée Pierre de Fermat

Une archive à télécharger se trouve dans le dossier du TP sur Cahier de Prépa.

Pour ce TP, vous aurez besoin d'utiliser la librairie `string.h`. En particulier, on rappelle que `int strcmp(char* str1, char* str2)` renvoie :

- 0 si les chaînes sont égales
- un entier strictement 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 pré-existante en C (dans `<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 s'utilise, 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, etc...). Appuyez sur q pour en sortir.

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.

La pile, le tas et le segment de données

Q1. Le code suivant crée deux chaînes de caractères ayant le même contenu, et affiche leur adresse de stockage :

```c
#include <stdio.h>

int main(){
    char* s = "bonjour";
    char* t = "bonjour";

    printf("%p\n", s);
    printf("%p\n", t);

    return 0;
}
```

Recopiez et exécutez ce code : que remarquez-vous ?

Q2. Rajoutez au `main` l'instruction `s[0] = 'B';` pour remplacer la première lettre de `s` avant l'affichage. Que se passe-t-il ?

En plus de la pile et du tas, une troisième zone, appelée **segment de données**, ou **data segment** en anglais, sert à stocker diverses données. En particulier, les variables globales et les chaînes de caractères directement écrites dans le code y sont stockées. Par exemple, dans le code suivant :

```c
int x = 3;

int main(){
    char* s = "bonjour";
    char* t = "bonjour";
}
```

Le segment de données contient La variable `x`, ainsi qu'un tableau des caractères `b`, `o`, ..., `r`. De plus, une partie du segment de données est en lecture seule, ce qui signifie qu'on ne peut pas modifier son contenu. Les chaînes littérales, c'est à dire les strings écrits directement dans le code, sont stockées dans cette partie de la mémoire. Ainsi, dans l'exemple précédent, `s` et `t` sont des variables locales de la fonction `main`, et sont stockées dans la pile, mais elles **pointent vers** la partie en lecture seule du segment de données. GCC applique alors une légère optimisation : puisque ces deux chaînes sont identiques et non-modifiables, il n'y a pas besoin d'utiliser deux chaînes distinctes, les deux pointeurs peuvent pointer vers la même adresse.

Lorsque l'on manipule des chaînes de caractères dans un programme, il faut faire attention à ne pas mélanger les strings alloués sur le tas avec ceux alloués sur la pile ou dans le segment de données.

Q3. On considère le code suivant :

```c
/* Libère l, liste de n chaînes de caractères. */
int free_string_list(char** l, int n){
    for (int i=0; i<n; i++){
        free(l[i]);
    }
    free(l);
}

int main(){
    char** l = malloc(3 * sizeof(char*));

    l[0] = "bonjour";

    l[1] = malloc(7*sizeof(char));
    strcpy(l[1], "coucou");

    l[2] = l[1];
}
```

Pour libérer toute la mémoire correctement, faut-il appeler `free(l)` ? `free_string_list(l)` ? Aucun des deux ? Et si la liste contenait des milliers de chaînes, allouées parfois dans le tas et parfois pas ?

Dans ce TP, on prendra toujours garde à ce que toute structure de données qui stocke des strings les stocke dans le tas, et à ce qu'elle ne contienne pas de doublons en mémoire : deux strings stockés dans la structure devront pointer vers des **adresses** distinctes. Ainsi, les fonctions de libération de mémoire seront plus simples.

On présente la fonction `strdup` de la librairie `string.h`, qui permet de créer une copie dans le tas d'une chaîne de caractère. Vous pouvez consulter sa documentation dans le manuel pour voir sa spécification précise. Par exemple, le code suivant est une version du code précédent qui utilise strdup pour tout réserver dans le tas, il s'exécute sans fuite mémoire :

```c
/* Libère l, liste de n chaînes de caractères.
   Chaque case de l doit pointer vers une zone
   allouée dans le tas, sans doublon */
int free_string_list(char** l, int n){
    for (int i=0; i<n; i++){
        free(l[i]);
    }
    free(l);
}

int main(){
    char** l = malloc(3 * sizeof(char*));

    l[0] = strdup("bonjour");

    l[1] = malloc(7*sizeof(char));
    strcpy(l[1], "coucou");

    l[2] = strdup(l[1]);
    // chaque case de l pointe vers une zone distincte du tas
    free_string_list(l, 3);
}
```

Dictionnaires et tables de hachage

On rappelle la spécification du type abstrait **dictionnaire** : ce type permet de stocker des associations entre des clés d'un ensemble K et des valeurs d'un ensemble V, de telle sorte qu'une clé de K apparaît au plus une fois dans une dictionnaire. On peut donc voir un dictionnaire comme une fonction partielle de K dans V.

— Créer un dictionnaire vide

— Déterminer si une clé k est dans un dictionnaire D

— Récupérer la valeur associée à une clé k dans un dictionnaire D

— Supprimer une clé k d'un dictionnaire D

On rappelle également le principe des tables de hachage : une table de hachage est un tableau de m cases, munie d'une fonction $h : K \to [\![0, m-1]\!]$ appelée fonction de hachage. Les opérations concernant une clé k se feront dans la case i du tableau. La fonction de hachage n'étant pas nécessairement injective, deux clés distinctes peuvent être hachées vers le même indice i, on parle alors de collision.

Gestion des collisions par chaînage

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

Téléchargez l'archive du TP sur cahier de prépa. Elle contient plusieurs fichiers :

```
keyval.h / keyval.c
dico_chaine.h / dico_chaine.c
dico_hash.h / dico_hash.c
```

Les fichiers `keyval.h/.c` concernent les clés et les valeurs de nos dictionnaires. Y sont implémentées des fonctions pour comparer, afficher et libérer les clés et les valeurs. Ils contiennent également **une fonction de hachage** simple.

Les fichiers `dico_chaine.h/.c` serviront à implémenter les dictionnaires par des listes chaînées stockant des couples (clé, valeur).

Vous n'aurez presque pas à modifier ces fichiers pendant le TP. Lisez les `.h` pour voir l'interface précise proposée, et lisez le `.c` afin de vérifier que le code est cohérent.

Les fichiers `dico_hash.h/.c` sont presque vides, ça sera à vous de les remplir au fil du TP. Ils contiennent pour le moment la définition d'une structure pour les tables de hachage, ainsi qu'un mécanisme permettant de redimensionner la table : lorsque le taux de remplissage passe en dessous de 0.1 ou au dessus de 0.5, on redimensionne la table de façon à ramener le taux de remplissage entre les deux [1].

1. C'est la stratégie utilisée dans l'implémentation C standard de python !

Q4. 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 une fonction, vous devez la tester dans “test.c” avec un bloc de code de la forme :

```
int main(){
   ...

   /* DEBUT TEST nom_fonction_ou_nom_test*/
   ...
   ...
   ...
   /* FIN TEST nom_fonction_ou_nom_test */

   ...
   return 0;
}
```

Les jeu de tests pourront être des vérifications à l'aide d'assert, ou, lorsque ce n'est pas possible, un simple affichage.
Lancez votre programme de test régulièrement, et utilisez valgrind si possible pour éviter les fuites mémoires.

Q5. Pour le moment, en utilisant `assert`, ajoutez des tests permettant de vérifier que `hash(x) % m` vaut bien y pour :

- $x =$ ”bonjour”, $m = 101, y = 60$
- $x =$ ”voici un texte a hacher”, $m = 503, y = 151$
- $x =$ ”voici un texte a macher”, $m = 503, y = 156$

Q6. Ajoutez un jeu de tests pour la fonction `egal` de “keyval.h”.

Q7. Écrivez un jeu de test pour l'implémentation des dictionnaires par listes chaînées, proposée dans `dico_chaine.h/.c`. Votre jeu de test doit couvrir autant d'éventualités que possible (chaîne vide ou presque vide, recherche d'un élément non présent, ajout d'une association pour une clé déjà présente, nombre de clés bien comptabilisé, etc...). Par exemple, créez une chaîne, ajoutez quelques associations dedans, cherchez quelques clés, supprimez quelques associations, recherchez d'autres clés.

Q8. Lancez votre jeu de test : vous devez constater qu'il y a des erreurs. Identifiez les erreurs dans l'implémentation, et corrigez-les à l'aide de vos tests, puis passez à la suite.

Passons maintenant à l'implémentation des tables de hachage.

Q9. Lisez le contenu de `dico_hash.h` pour voir les différentes fonctions proposées dans l'interface. Documentez celles qui ne le sont pas déjà. Regardez également dans `dico_hash.c` pour voir les fonctions d'affichage et de libération qui sont déjà implémentées. Le mécanisme de table redimensionnable est déjà implémenté par la fonction `resize`. Lisez le commentaire de documentation de cette fonction : vous devrez l'appeler dans vos fonctions.

Q10. Implémentez les fonctions relatives à la structure `hashtable_t` dans `dico_hash.c`, en les testant au fur et à mesure dans `test.c`.

Q11. Ajoutez aux dictionnaires l'opération suivante : Renvoyer la liste (sous la forme d'un tableau alloué dans le tas) des clés d'un dictionnaire.

A 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'occurrences d'un mot.

On considère le texte intégral de "Vingt Mille Lieues sous les mers" de Jules Verne, stocké dans `vingt_mille_lieues.txt`. On veut savoir quel est le mot le plus utilisé parmi ceux faisant plus de 8 lettres. On utilise pour cela l'algorithme vu en cours utilisant le dictionnaire des occurrences :

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'occurrences 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] ← 1;
6 Chercher dans D la clé k_0 ayant la valeur associée maximale;
7 retourner k_0
```

Q12. Quelle est la complexité de cet algorithme en utilisant une table de hachage avec une fonction de hachage idéale ?

Q13. Créez un dossier `texte`, et copiez-collez y les 6 fichiers `.h/.c` de l'implémentation des tables de hachage. Modifiez tout le code nécessaire pour que les clés soient des chaînes de caractères et les valeurs des entiers positifs.

Q14. Créez maintenant un fichier "occurrences.c" qui servira à implémenter l'algorithme précédent. Créez-y une fonction main, et une fonction `void test()` où vous mettrez les tests.

Q15. Écrivez une fonction `char* argmax(hashtable_t* d)` renvoyant la clé dont la valeur associée est maximale. On supposera en précondition que le dictionnaire n'est pas vide, et que les valeurs sont des entiers positifs. Écrivez également un jeu de tests afin de tester cette fonction.

Q16. Écrivez une fonction `hashtable_t* occurences(char* filename, int K)` qui renvoie le dictionnaire des occurrences des mots de `filename`, parmi les mots d'au moins K lettres. N'oubliez pas les questions de gestion mémoire expliquées au début du TP !

Q17. 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'occurrences. 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).

Q18. 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*** (via `argc` et `argv`) un nom de fichier `fn` et une borne K et affichant le mot le plus fréquent dans `fn` ayant plus de K lettres. Testez les performances de votre programme sur `vingt_mille_lieues.txt`.

Choix de la fonction de hachage

Une bonne fonction de hachage doit induire le moins de collisions possible. On peut même souhaiter qu'il soit difficile de **calculer** des collisions.

La fonction de hachage implémentée dans l'archive additionne toutes les lettres du mot, on obtient alors l'indice de la table de hachage en prenant le résultat modulo m :

$$h(k_0k_1\dots k_{l-1}) = \sum_{i=0}^{l-i} k_i \bmod m$$

Q19. Donner deux mots ayant le même hash.

Pour évaluer les performances d'une fonction de hachage, on pourra chronométrer le temps d'exécution du programme écrit à la partie précédente. **Rappel :** nous avons vu comment chronométrer du code C avec la fonction `clock` au TP sur le tri rapide, allez voir si vous ne vous souvenez plus comment cette fonction s'utilise.

Hachage par division On propose la fonction de hachage suivante :

$$h(k_0k_1\dots k_{l-1}) = \sum_{i=0}^{l-i} k_i 256^i \bmod m$$

Q20. Mesurer le temps d'exécution avec cette fonction de hachage, et comparer avec la première fonction.

Q21. Donner une collision sur cette fonction lorsque $m = 256^k$ avec $k \in \mathbb{N}$

La fonction précédente prend le résultat modulo m, la taille de la table de hachage. On peut d'abord appliquer un autre modulo en utilisant un grand nombre premier, ce qui a pour effet de répartir les bits de manière assez aléatoire :

$$h(k_0k_1\dots k_{l-1}) = (\sum_{i=0}^{l-i} k_i 256^i \bmod p) \bmod m$$

On peut prendre par exemple $p = 1610612741$, un grand nombre premier entre 2^{30} et 2^{31}.

Q22. Implémenter cette fonction de hachage et évaluer ses performances.

Q23. Trouver une collision pour cette fonction. On pourra écrire p en base 256.

La fonction précédente peut être vue comme une évaluation d'un polynôme dont les coefficients sont les k_i. h évalue ce polynôme sur 256. On peut changer cette valeur pour obtenir une différente fonction de hachage :

$$h(k_0k_1\dots k_{l-1}) = \sum_{i=0}^{l-i} k_i B^i \bmod m$$

où B est un entier quelconque.

Q24. Évaluer les performances de cette fonction de hachage lorsque $B = 33$ (c'est l'algorithme appelé DJBX33A).

Algorithme FNV La fonction de hachage précédente est une évaluation de polynôme, et peut s'implémenter à l'aide de l'algorithme de Horner en $\mathcal{O}(l)$ comme suit :

Algorithme 2 : Horner

Entrée(s) : $k_0 \dots k_{l-1}$ des entiers, $B \in \mathbb{N}$, $m \in \mathbb{N}$
Sortie(s) : $\sum_{i=0}^{l-1} k_i B^i \bmod m$

```
1 res ← INIT;
  // Invariant: res = k_{i+1} + k_{i+2}B + ··· + k_{l-1}B^{l-i-2} mod m
2 pour i = l − 1 à 0 faire
3     res ← res × B mod m;
4     res ← res + k_i mod m;
  // En sortie: i = −1, donc res = k_0 + k_1B + ··· + k_{l-1}B^{l-1} mod m
5 retourner res
```

L'algorithme de hachage Fowler-Noll-Vo est une modification de cette méthode où, au lieu d'une addition, on effectue un XOR bit à bit :

Algorithme 3 : FNV

Entrée(s) : $k_0 \dots k_{l-1}$ clé à hacher
Sortie(s) : Valeur de hachage

```
1 res ← INIT;
2 pour i = 0 à l − 1 faire
3     res ← res × P;
4     res ← res XOR k_i;
5 retourner res mod m
```

Dans cet algorithme, **INIT** et **P** sont deux paramètres à choisir, **P** devant être un grand nombre premier. En pratique, lorsque l'on travaille sur 32 bits (i.e. avec des tables de hachage ayant au plus 2^{32} cases), on peut choisir les paramètres suivants :

$$\mathbf{P} = 16777619, \mathbf{INIT} = 2166136261$$

Q25. Implémenter cet algorithme de hachage, et évaluer ses performances.

Q26. Chercher une collision (on pourra supposer m suffisamment large).