GIS3 — Structures de Données — Épreuve de Mai 2017
Durée 2 heures — Aucun document autorisé

---

1 Petits Exercices (8 points)

Question 1 [2 pts]. Insérer les clés 10, 22, 31, 4, 15, 28, 17 dans une table de hachage de $N = 11$ alvéoles, gérée avec la technique du double hachage. La fonction de hachage est $h(s) = (h_1(s), h_2(s)) = (s \bmod N, 1 + s \bmod (N - 1))$. Indiquer les collisions.

Question 2 [1 pt]. L'algorithme A a une complexité en temps en $O(n)$. L'algorithme B a une complexité en temps en $O(n^2)$. Peut-on affirmer que A est plus rapide que B, quand n tend vers l'infini (justifier) ?

Question 3 [1 pt]. L'algorithme A a une complexité en temps en $O(n)$. L'algorithme B a une complexité en temps en $\Omega(n^2)$. Peut-on affirmer que A est plus rapide que B, quand n tend vers l'infini (justifier) ?

Question 4 [1 pt]. On souhaite affecter à une variable `p` une zone mémoire allouée dynamiquement, aussi petite que possible, capable de recevoir la chaîne de caractères `"elephant"`. Donner la déclaration de la variable et l'instruction qui lui alloue de la mémoire.

Question 5 [3 pts]. Écrire une action en C qui ajoute un double en queue d'une liste. Utiliser la déclaration suivante.

```
struct maillon_double {
    double valeur;
    struct maillon_double* suivant;
};

#define NIL (struct maillon_double*)0
```

```
struct liste_double {
    struct maillon_double* tete;
    int nbelem;
};
```

2 Problème (12 points)

Toutes les explications et les questions se rapportent au document préparatoire ci-joint. L'idée consiste à interpréter chaque égalité entre caractères en une règle de substitution de la forme $src \rightarrow dst$. Par exemple, les deux égalités

é = è
è = e

qui définissent la classe d'équivalence { é, è, e }, peuvent s'interpréter en les deux substitutions suivantes

$$\begin{array}{ccc} \text{é} & \rightarrow & \text{è} \\ \text{è} & \rightarrow & \text{e} \end{array}$$

Si on applique ces règles à é, on obtient e en deux substitutions. Si on les applique à è, on obtient e en une substitution. Le caractère e est appelé le *représentant canonique* de la classe { é, è, e }. Pour tester l'équivalence de deux caractères, on teste l'égalité de leur représentant canonique. Comme les deux caractères é et è ont le même représentant canonique, on conclut qu'ils sont équivalents.

On cherche donc une structure de données, une sorte de dictionnaire D dont les valeurs seraient des règles de substitution et qui permettrait de calculer rapidement le représentant canonique de la classe d'équivalence d'un caractère [1] c donné. En pseudo-code, l'algorithme du calcul de représentant canonique de la classe d'équivalence de c, vis-à-vis de la relation d'équivalence contenue dans D s'écrit :

Soit c un caractère
while D contient une règle de la forme $c \rightarrow c'$ do
  Affecter c' à c
end do
Le résultat est dans c

Question 6 [2 pts]. Citer deux exemples de structures de données étudiées en cours permettant d'implanter un tel dictionnaire. Quelles seraient les clés des valeurs ?

Question 7 [2 pts]. Choisir une des deux structures ci-dessus. Donner la déclaration C de la structure avec sa spécification. Ne pas donner les prototypes des fonctions exportées.

Pour que l'algorithme du représentant canonique soit correct, il faut que les règles de substitution présentes dans le dictionnaire vérifient certaines contraintes. Par exemple, il faut que deux règles de substitution différentes aient des champs `src` différents. Il faut aussi que les règles de substitution ne crèent pas de « cycles ».

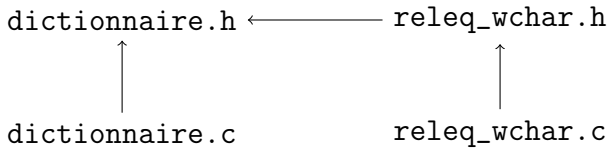
Question 8 [2 pts]. On suppose ces contraintes satisfaites. Écrire une fonction C paramétrée par un caractère c et un dictionnaire D, qui retourne le représentant canonique de la classe d'équivalence de c, vis-à-vis de D.

Votre solution peut être récursive ou itérative. Programmez la fonction sans supposer [2] l'existence des fonctions de dictionnaires étudiées en cours.

Pour respecter les contraintes mentionnées plus haut lors de la création du dictionnaire à partir du fichier contenant les égalités entre caractères, on décide 1) d'encapsuler le dictionnaire dans une structure de données nommée `struct releq_wchar`, comportant un unique

1. Tous les caractères considérés dans cet énoncé sont des *caractères étendus* de type `wchar_t`.
2. Si votre solution utilise des listes chaînées, vous pouvez supposer l'existence des fonctions de listes étudiées en cours.

champ (le dictionnaire) et 2) d'écrire avec soin l'action qui ajoute une nouvelle règle de substitution aux règles existantes. En supposant le dictionnaire de règles implanté dans un module `dictionnaire`, le découpage logiciel devient :



Question 9 [2 pts]. Écrire le fichier `releq_wchar.h`. Ne donner que les prototypes des fonctions nécessaires au programme principal de la Figure 1.

Question 10 [2 pts]. Écrire une action C, paramétrée par une relation d'équivalence (type `struct releq_wchar`) en mode donnée-résultat, deux caractères a et b, et qui ajoute une règle de substitution à la relation pour prendre en compte la nouvelle égalité $a = b$.

L'idée consiste à enregistrer dans le dictionnaire la règle de substitution $a' \to b'$ où a' et b' désignent les représentants canoniques de a et de b dans le dictionnaire existant. Que faut-il faire si a et b ont déjà le même représentant canonique ?

Vous pouvez supposer que votre implantation de dictionnaire comporte les actions et les fonctions étudiées en cours, à condition de rappeler leur spécification.

Question 11 [2 pts]. Écrire une version de la fonction `wcscmp_releq_wchar` utilisée dans le programme de la Figure 1. En cas d'égalité, cette fonction doit retourner 0. En cas de non égalité, elle doit retourner la première différence non nulle entre les représentants canoniques des caractères des chaînes (les représentants canoniques sont des `wchar_t`, qui peuvent être vus comme des entiers). Les chaînes peuvent être de longueurs différentes.

On voudrait pouvoir tester l'égalité entre deux mots « aux signes diacritiques (accents, cédilles, ...) près ». Par exemple, on voudrait une fonction qui nous indique que, si on ne tient pas compte des accents, les deux mots `"événement"` et `"évènement"` sont égaux. L'idée consiste à définir, dans un fichier, une relation d'équivalence entre caractères, présentée sous la forme d'une suite d'égalités entre caractères. Voici un exemple jouet :

```
$ cat equivalences.txt
é = e
è = e
ë = ê
é = ë
î = i
y = ÿ
ç = c
```

La relation d'équivalence définie par le fichier contient les égalités du fichier mais aussi toutes celles qui peuvent s'en déduire automatiquement, par transitivité. Par exemple, la relation contient `ê = è` bien que cette égalité ne figure pas explicitement dans le fichier. Le programme de la Figure 1 montre un exemple de ce qu'on voudrait pouvoir faire.

Les caractères manipulés sont codés par des *wide char*. Le langage C comporte un type pour les caractères étendus (*wide char*). Il s'agit du type `wchar_t`, défini dans `wchar.h`. La taille (`sizeof`) et le codage des *wide char* n'est pas spécifié : il dépend du compilateur et de la plateforme. Le type `wint_t` est prévu pour convertir les *wide char* en entiers. Ce type peut lui-même être converti en `int` sans perte d'information. Les constantes de type `wchar_t` se notent comme les constantes de type `char`, précédées d'un `L` majuscule. Par exemple `L'a'` et `L'\n'` désignent la lettre « a » et le retour-chariot de type *wide char*. Les chaînes de *wide char*, appelées *wide character strings*, sont des tableaux de `wchar_t` terminés par le *wide char* `L'\0'`. Elles sont de type `wchar_t*`. Les constantes *wide character strings* se notent comme les constantes chaînes de caractères, précédées d'un `L` majuscule. Par exemple `L"bonjour"` désigne la chaîne « bonjour » de type `wchar_t*`. De nombreuses fonctions bien connues de manipulation de chaînes de caractères ont un équivalent *wide character string*. Par exemple `wcscmp` et `wcscpy` sont les équivalents de `strcmp` et `strcpy`, pour les *wide character strings*. La fonction `wprintf` est la généralisation *wide char* de `printf`. Son premier paramètre (le « format ») est une *wide character string*. Toutes les *spécifications de conversion* (précédées d'un `%`) de `printf` sont disponibles. Deux nouvelles *spécifications de conversion* sont introduites : `%lc` pour un *wide char* et `%ls` pour une *wide character string*.

```
#include <stdio.h>
#include <locale.h>
#include <assert.h>
#include "releq_wchar.h"

int main ()
{   FILE* f;
    struct releq_wchar A;
    wchar_t a, b;
    wchar_t mot1 [80], mot2 [80];

    assert (setlocale (LC_ALL, "fr_FR.UTF-8"));

    init_releq_wchar (&A);
/*
 * Chargement de la relation d'équivalence dans A
 */
    f = fopen ("equivalences.txt", "r");
    assert (f != NULL);
    while (fwscanf (f, L"%lc = %lc\n", &a, &b) == 2)
        ajout_releq_wchar (&A, a, b);
    fclose (f);
/*
 * Test de l'équivalence de deux chaînes de wide char.
 * La fonction wcscmp_releq_wchar est une généralisation de wcscmp et strcmp.
 */
    wprintf (L"Entrez deux mots: ");
    wscanf (L"%ls", mot1);
    wscanf (L"%ls", mot2);
    if (wcscmp_releq_wchar (mot1, mot2, &A) == 0)
        wprintf (L"Les deux mots sont equivalents\n");
    else
        wprintf (L"Les deux mots ne sont pas equivalents\n");

    clear_releq_wchar (&A);
    return 0;
}
```

FIGURE 1 – Dans un premier temps, la relation d'équivalence est chargée dans la variable `A`. Dans un deuxième temps, elle est utilisée pour tester l'équivalence entre deux mots. L'instruction initiale indique au programme qu'il est exécuté dans un environnement qui utilise le format UTF-8.