GIS2A3 — Structures de Données — Épreuve de Mai 2016
Durée 2 heures — Aucune calculatrice, aucun document autorisé

---

1 Petits exercices (6 points)

1.1 Listes chaînées

On rappelle les déclarations :

```
struct maillon {
    int valeur;
    struct maillon* suivant;
};
#define NIL (struct maillon*)0
struct liste {
    struct maillon* tete;
    int nbelem;
};
```

Question 1 [2 pts]. Écrire une fonction C paramétrée par un entier `d`, une liste `L`, et qui ajoute `d` en queue de `L`.

1.2 Tableaux redimensionnables

On considère la déclaration C suivante. Le champ `tab` pointe vers un tableau alloué dynamiquement. Le champ `alloc` contient le nombre de doubles alloués à `tab`. Le champ `size` contient le nombre de doubles utilisés dans `tab`. On a bien sûr `size` $\leq$ `alloc`. En cas d'égalité, le tableau est plein.

```
struct tableau {
    double* tab;
    int alloc;
    int size;
};
```

Question 2 [2 pts]. Écrire une fonction C paramétrée par un double `d`, un tableau `T` (du type ci-dessus) et qui enregistre le double à la fin du tableau. Redimensionner `T` s'il est plein.

1.3 Complexité

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

Question 4 [1 pt]. Les algorithmes A et B ont tous deux une complexité en temps en $\Theta(n^3)$. Peut-on affirmer que, quand n tend vers l'infini, les temps de calcul des deux algorithmes sont de plus en plus proches ?

2 Problème (14 points)

Toutes les questions s'appliquent au document préparatoire rappelé en fin d'énoncé.

Question 5 [1 pt]. La méthode utilisée pour associer un indice dans `T` à un identificateur de fonction, via la fonction `h`, rappelle une structure de données étudiée en cours. Laquelle ?

2.1 L'instruction `assert` de la ligne 34

Question 6 [1 pt]. Quel est le rôle de cette instruction ?

Question 7 [1 pt]. Est-il raisonnable de penser qu'aucune erreur ne se déclenchera ligne 34, tant que le nombre d'identificateurs présents dans `T` sera inférieur à `TMAX`/2 ?

Question 8 [1 pt]. Quels mécanismes étudiés en cours pourraient être utilisés pour éviter cette utilisation de `assert` ?

2.2 Réécriture de code

On souhaite revoir la conception du programme donné en annexe.

En quelque sorte, la structure de données mise en œuvre dans le programme implante le calcul rapide du complémentaire d'un ensemble B dans un ensemble A d'identificateurs de fonctions, c'est-à-dire l'ensemble :

$$A \setminus B \quad = \quad \{a \in A \mid a \notin B\}\,.$$

On cherche donc à concevoir une structure de données suffisamment bien faite pour que l'implantation de cette structure de données soit invisible depuis le programme principal. On aimerait aussi que la structure de données ait un certain degré de généralité, pour permettre sa réutilisation dans d'autres applications.

Question 9 [2 pts]. Donner la déclaration d'une structure C `struct compens` correspondant à cette structure de données. Donner une spécification en une phrase pour chaque champ de la structure.

Question 10 [2 pts]. Donner les prototypes des fonctions exportées. Spécifier ces fonctions en expliquant bien le rôle de chaque paramètre.

Question 11 [2 pts]. Modifier le code du programme principal pour qu'il utilise votre structure de données. Ne pas tout recopier. Indiquer les numéros des lignes à changer. Utiliser éventuellement la feuille de l'énoncé.

2.3 Format d'entrée

Dans le programme donné en annexe, les identificateurs de fonctions testées sont stockés dans le tableau `G`. Ce n'est pas très réaliste.

En pratique, on dispose d'un ensemble de programmes de test. Chaque programme de test est représenté par son nom, qu'on peut assimiler à un identificateur de fonction. À chaque programme de test, on associe la liste des identificateurs de fonctions, testées par le programme.

Pour fixer les idées, voici un exemple possible. Le programme `test1` teste la liste `[init_liste, impression_liste, clear_liste]`. Le programme `test2` teste la liste `[init_liste, ajout_en_tete_liste, clear_liste]`.

Question 12 [1 pt]. Donner la déclaration d'un type permettant de représenter les listes d'identificateurs, dans le cadre d'un module `liste_ident`.

Question 13 [1 pt]. Donner la déclaration d'un type `struct pgmtest` permettant de représenter les informations relatives à un programme de test (nom et fonctions testées), dans le cadre d'un module `pgmtest`.

Question 14 [1 pt]. Donner la déclaration d'un type permettant de représenter les listes de programmes de tests, dans le cadre d'un module `liste_pgmtest`.

Question 15 [1 pt]. Décrire sous la forme d'un diagramme le découpage en fichiers qu'on obtiendrait si on implantait tous ces modules. Bien faire apparaître les directives d'inclusion.

A Document préparatoire à l'épreuve écrite

Le problème de l'épreuve écrite relève de la thématique de la « couverture de tests ». On suppose qu'on a écrit un module (mettons) de listes et qu'on a écrit quelques programmes principaux (appelés programmes de test) qui testent les fonctions exportées du module. On aimerait savoir si toutes les fonctions du module sont bien testées.

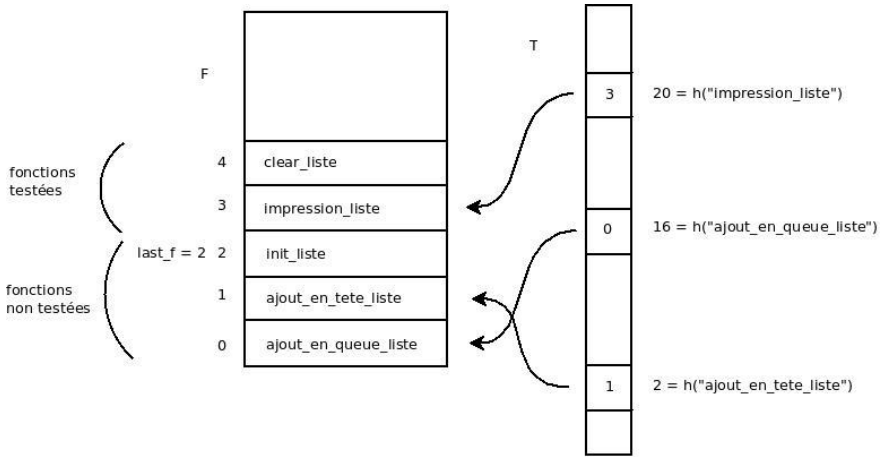

FIGURE 1 – Le tableau `F` contient des identificateurs de fonctions. Le tableau `T` contient des indices dans `F`. Pour toute fonction `f` appartenant au tableau `F`, l'entier `h(f)` est un indice dans `T` et `f = F[T[h(f)]]`. L'indice `last_f` sépare l'ensemble des fonctions présentes dans `F` en deux sous-ensembles : celles qui sont testées (partie supérieure) et celles qui ne le sont pas (partie inférieure).

L'ensemble des fonctions du module est enregistré dans une structure de données constituée de deux tableaux `F` et `T`. L'ensemble des fonctions de `F` est divisé en deux parties : les fonctions non testées (en bas) et les testées (en haut). La frontière est repérée grâce à un indice : `last_f`. Voir la Figure 1. Le tableau `G` contient les fonctions testées par les programmes de test. Initialement, toutes les fonctions sont considérées comme non testées. À chaque itération de la boucle principale, une fonction de `G` est considérée et la structure de données est mise-à-jour de façon à ce qu'elle soit considérée comme testée, à la fin de l'itération. la boucle détermine rapidement l'indice dans `F` de la fonction courante du tableau `G`. Si cet indice pointe dans la partie supérieure de `F`, il n'y a rien à faire. S'il pointe dans la partie inférieure de `F`, une permutation est effectuée à la fois dans `F` et dans `T` et l'indice `last_f` est diminué de 1. À l'exécution, le programme `main.c` affiche :

```
Fonctions non testees:
init_liste
clear_liste

Fonctions testees:
ajout_en_queue_liste
impression_liste
ajout_en_tete_liste
```

```
    // NOM et PRÉNOM:

1   #include <stdio.h>
2   #include <string.h>
3   #include <assert.h>
4
5   #define LMAX 64
6   typedef char identificateur [LMAX];
7
8   #define FMAX 100
9   #define TMAX 101
10
11  int h (identificateur s)
12  {   int i, r;
13      r = 0;
14      for (i = 0; s[i] != '\0'; i++)
15          r = (r + s[i]) % TMAX;
16      return r;
17  }
18
19  int main ()
20  {   identificateur F [] = { "ajout_en_queue_liste", "ajout_en_tete_liste",
21                      "init_liste", "impression_liste", "clear_liste" };
22      identificateur G [] = { "ajout_en_tete_liste", "impression_liste",
23                      "ajout_en_queue_liste", "impression_liste" };
24      int T [TMAX];
25      int nb_f, last_f, f1, f2, t1, t2, g1, nb_g;
26
27      nb_f = sizeof (F) / sizeof (identificateur);
28      nb_g = sizeof (G) / sizeof (identificateur);
29
30      for (t1 = 0; t1 < TMAX; t1++)
31          T [t1] = -1;
32      for (f1 = 0; f1 < nb_f; f1++)
33      {   t1 = h (F [f1]);
34          assert (T [t1] == -1);
35          T [t1] = f1;
36      }
37      last_f = nb_f-1;
38      for (g1 = 0; g1 < nb_g; g1++)
39      {   t1 = h (G [g1]);
40          f1 = T [t1];
41          f2 = last_f;
42          if (f1 <= f2)
43          {   if (f1 < f2)
44              {   identificateur tmp;
45                  t2 = h (F [f2]);
46                  strcpy (tmp, F [f1]);
47                  strcpy (F [f1], F [f2]);
48                  strcpy (F [f2], tmp);
49                  T [t1] = f2;
50                  T [t2] = f1;
51              }
52              last_f -= 1;
53          }
54      }
55      printf ("Fonctions non testees:\n\n");
56      for (f1 = 0; f1 <= last_f; f1++)
57          printf ("%s\n", F [f1]);
58      printf ("\nFonctions testees:\n\n");
59      for (f1 = last_f+1; f1 < nb_f; f1++)
60          printf ("%s\n", F [f1]);
61      return 0;
62  }
```