GIS3 — Structures de Données — Épreuve de Mars 2016
Durée 2 heures — Aucun document autorisé

---

1 Petits exercices (9 points)

1.1 Listes Chaînées

Certaines implantations de listes chaînées autorisent plusieurs listes à partager des maillons. Voir figure 1.

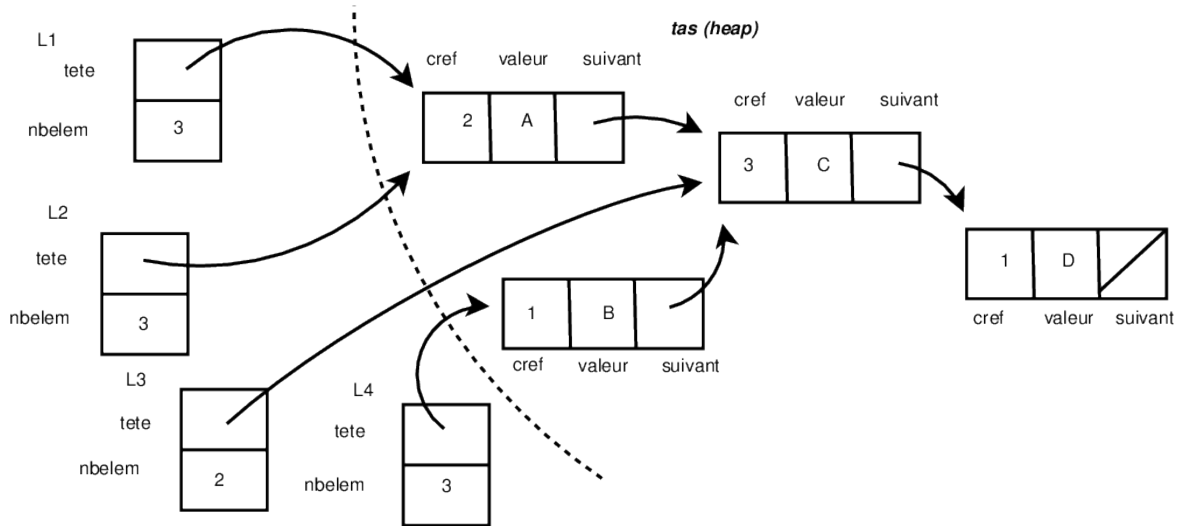

FIGURE 1 – Les variables $L_1, L_2, \ldots, L_4$ contiennent quatre listes chaînées. Ces listes partagent des maillons. Chaque maillon est muni d'un *compteur de référence* (`cref`) qui compte le nombre de pointeurs pointant vers lui.

Question 1 [2 pts]. Rappeler le code du destructeur de listes étudié en cours et expliquer pourquoi il ne peut plus être utilisé dans ce cas.

Pour pouvoir écrire un destructeur, l'idée consiste à munir chaque maillon d'un *compteur de références* (`cref`), qui compte le nombre de pointeurs qui pointent vers lui. L'algorithme devient alors celui de la figure 2.

Question 2 [2 pts]. Coder en C l'algorithme de la figure 2. Les déclarations sont les suivantes :

```
algo : clear_liste (L)
begin
  M = la tête de L
  fini = false
  while not fini do
    if M = NIL then
      fini = true
    else
      décrémenter de 1 le champ cref de M
      if ce champ reste strictement positif then
        fini = true
      else
        libérer le maillon et faire avancer M sur le maillon suivant
      end if
    end if
  end do
end
```

FIGURE 2 – Destructeur de listes avec maillons munis de compteurs de références. Si on l'applique sur la liste L_4 de la figure 1, le maillon B est libéré, le compteur de références du maillon C passe de 3 à 2 et l'algorithme s'arrête.

```
struct maillon {
    int cref;
    char valeur;
    struct maillon* suivant;
};
```

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

1.2 Tables de hachage

On considère l'insertion des clés $\square$, $\bigcirc$, $\triangle, *$ dans une table de hachage de $N = 12$ alvéoles. La table est gérée avec la technique du double hachage. Les valeurs de hachage des différents éléments sont les suivantes :

	h_1	h_2
$\square$	3	5
$\bigcirc$	7	7
$\triangle$	7	9
$*$	4	3

Question 3 [2 pts]. Insérer les clés dans la table. Donner un schéma de la table résultat. Indiquer les éventuelles collisions.

Question 4 [1 pt]. La fonction h_2 retourne un nombre impair dans tous les cas. Est-ce suffisant pour trouver un alvéole libre s'il en existe au moins un dans la table ?

1.3 Arbres Binaires de Recherche

Question 5 [2 pts]. Insérer les clés suivantes dans un ABR initialement vide. Donner les états successifs de l'ABR. Bien dessiner les arcs de façon oblique.

$$67,\ 99,\ 31,\ 22,\ 88,\ 70\,.$$

2 Problème (11 points)

Toutes les questions s'appliquent au document préparatoire rappelé en fin d'énoncé. La section 2.1 (complexité) est indépendante des sections suivantes.

2.1 Complexité

Question 6 [2 pts]. Décrire un pire des cas. Donner un majorant asymptotique $O(g(n))$ de la fonction $f(n)$, dans ce pire des cas. La fonction $g(n)$ doit être un monôme simple.

Question 7 [1 pt]. Si on implante l'optimisation mentionnée dans l'énoncé, ce majorant asymptotique peut-il être amélioré ?

2.2 Réécriture de code

On souhaite réaliser un module `ecc` (les initiales de *ensemble de composantes connexes*) qui permette de simplifier le programme de la figure 3. Le type `struct ecc` à réaliser doit contenir le tableau T et la variable n. Voici quelques directives :

1. le tableau peut être codé dans un champ de taille fixe mais l'interface du module doit prévoir une évolution vers une implantation plus compliquée ;
2. dans l'interface du module, les sommets doivent être désignés par leur numéro (et pas par un caractère) ;
3. le constructeur peut être paramétré par le nombre de sommets.

Question 8 [2 pts]. Écrire un fichier `ecc.h`.

Question 9 [2 pts]. Écrire un fichier `ecc.c`. Implanter récursivement la fonction qui calcule le représentant de la composante connexe. Spécifier chacune des fonctions.

Question 10 [2 pts]. Modifier le programme de la figure 3 pour qu'il utilise votre module. Inutile de recopier tout le code. Vous pouvez indiquer les modifications à faire directement sur l'énoncé (n'oubliez pas vos nom et prénom sur la feuille).

2.3 Amélioration

On aimerait maintenant pouvoir utiliser le module `ecc` sans devoir passer au constructeur le nombre total de sommets et sans être limité par un nombre de sommets N fixé arbitrairement.

Question 11 [2 pts]. Décrire en quelques phrases votre solution. Indiquer les transformations à apporter aux fichiers `ecc.h` et `ecc.c`.

Document préparatoire à l'épreuve écrite

Le « problème » de l'épreuve de Structures de Données portera sur le programme C de la figure 3, qui implante un algorithme de la théorie des graphes. Il n'est nécessaire, ni de bien comprendre cet algorithme, ni de connaître le point de théorie des graphes, pour pouvoir répondre aux questions mais il est important de comprendre la « mécanique » de l'algorithme.

Arbres couvrants. Le programme C de la figure 3 est inspiré de [1, chapitre 21 (a)]. Il calcule un arbre couvrant d'un graphe non orienté connexe G. L'arbre couvrant est un sous-graphe de G : c'est un arbre qui contient tous les sommets et un sous-ensemble des arêtes de G. À l'exécution, le programme considère chaque arête l'une après l'autre et décide, soit de la garder, soit de la rejeter. À la fin de l'exécution, l'arbre couvrant est défini par l'ensemble des arêtes gardées. Voici une trace d'exécution :

```
Etape  0: (E, B) rep(E) = E, rep(B) = B,  garder (E, B)
Etape  1: (B, F) rep(B) = E, rep(F) = F,  garder (B, F)
Etape  2: (B, C) rep(B) = E, rep(C) = C,  garder (B, C)
Etape  3: (E, C) rep(E) = E, rep(C) = E, rejeter (E, C)
Etape  4: (F, E) rep(F) = E, rep(E) = E, rejeter (F, E)
Etape  5: (G, D) rep(G) = G, rep(D) = D,  garder (G, D)
Etape  6: (A, F) rep(A) = A, rep(F) = E,  garder (A, F)
Etape  7: (A, B) rep(A) = A, rep(B) = A, rejeter (A, B)
Etape  8: (G, F) rep(G) = G, rep(F) = A,  garder (G, F)
Etape  9: (D, E) rep(D) = G, rep(E) = G, rejeter (D, E)
Etape 10: (G, A) rep(G) = G, rep(A) = G, rejeter (G, A)
Etape 11: (F, D) rep(F) = G, rep(D) = G, rejeter (F, D)
Etape 12: (D, C) rep(D) = G, rep(C) = G, rejeter (D, C)
```

Principe de l'algorithme. Plaçons-nous à une étape et appelons H le sous-graphe de G contenant tous les sommets de G et l'ensemble des arêtes gardées aux étapes précédentes. Le sous-graphe H n'est pas nécessairement connexe. Il le sera obligatoirement en fin d'algorithme mais initialement, par exemple, il ne contient aucune arête et chaque sommet est une composante connexe de H à lui tout seul. Appelons (a, b) l'arête courante. L'algorithme teste si a et b appartiennent à la même composante connexe de H. Si c'est le cas, l'arête est rejetée. Si ce n'est pas le cas, l'arête est gardée.

Les variables de l'algorithme. Le graphe G est donné par un tableau d'arêtes, appelé G lui-aussi. L'entier m reçoit le nombre d'arêtes du graphe. L'entier n reçoit le nombre de sommets. Chaque sommet est représenté soit par un caractère (de `'A'` à `'G'`) soit par un numéro (de 0 à $n-1$). On calcule le numéro d'un caractère c en soustrayant la constante `'A'` à c.

L'ensemble des composantes connexes. Le tableau T sert à tester si deux sommets appartiennent à la même composante connexe de H. Les indices de T sont des numéros de sommets (de 0 à $n-1$). Les cases de T contiennent soit -1 soit un numéro de sommet.

Chaque composante connexe est représentée par un de ses sommets, appelé le *représentant* de la composante. Soit $0 \leq i < n$ un numéro de sommet. Le représentant $\text{rep}(i)$ de la composante du sommet i est défini ainsi :

$$\text{si } T[i] = -1 \text{ alors } \text{rep}(i) = i \text{ sinon } \text{rep}(i) = \text{rep}(T[i]). \tag{1}$$

Tester si deux sommets appartiennent à la même composante connexe revient à tester s'ils ont même représentant. À chaque fois qu'une arête (a, b) est gardée, il faut fusionner les composantes connexes des sommets a et b. Cela se fait en affectant $\text{rep}(a)$ à $T[\text{rep}(b)]$ (ou le contraire).

Analyse de complexité. Dans le pire des cas, le graphe G est complet et le nombre d'arêtes $m = \frac{1}{2}\, n\,(n-1)$. La taille de la donnée est donc représentée par le nombre de sommets n. Pour $f(n)$, il est naturel de compter le nombre de fois où une case de T est comparée avec -1 (le nombre de fois où le test $T[i] = -1$ est exécuté).

Le tableau T peut être vu comme une sorte de graphe orienté où chaque sommet pointe vers au plus un autre sommet. La complexité de l'algorithme va donc naturellement dépendre de la longueur maximale des chemins dans T.

Pour finir, on signale qu'il existe une optimisation peu coûteuse, qu'on n'implantera pas ici, mais qui assure que cette longueur maximale des chemins ne dépasse pas $\log_2(n)$, même dans le pire des cas.

Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique.* Dunod, Paris, 2ème edition, 2002.

NOM et PRENOM :

```
 1 #include <stdio.h>
 2
 3 struct arete {
 4     char a;  /* une extrémité de l’arête */
 5     char b;  /* l’autre extrémité */
 6 };
 7
 8 #define N 10
 9
10 int main ()
11 {   struct arete G [] = {
12         { ’E’, ’B’ }, { ’B’, ’F’ }, { ’B’, ’C’ }, { ’E’, ’C’ },
13         { ’F’, ’E’ }, { ’G’, ’D’ }, { ’A’, ’F’ }, { ’A’, ’B’ },
14         { ’G’, ’F’ }, { ’D’, ’E’ }, { ’G’, ’A’ }, { ’F’, ’D’ },
15         { ’D’, ’C’ } };
16     int m = sizeof (G) / sizeof (struct arete); /* nombre d’arêtes */
17     int n;                                       /* nombre de sommets */
18     int T[N];
19     int i;
20
21     n = ’G’ - ’A’ + 1;
22     for (i = 0; i < n; i++)
23         T[i] = -1;        /* initialement, une comp. connexe par sommet */
24     for (i = 0; i < m; i++)
25     {   int r, s;
26         printf ("Etape %2d: (%c, %c) ", i, G[i].a, G[i].b);
27         r = G[i].a - ’A’;
28         while (T[r] != -1)
29             r = T[r];         /* r = rep (G[i].a) */
30         printf ("rep(%c) = %c, ", G[i].a, r + ’A’);
31         s = G[i].b - ’A’;
32         while (T[s] != -1)
33             s = T[s];         /* s = rep (G[i].b) */
34         printf ("rep(%c) = %c, ", G[i].b, s + ’A’);
35         if (r != s)           /* si les comp. connexes sont différentes */
36         {   printf (" garder (%c, %c)\n", G[i].a, G[i].b);
37             T[s] = r;         /* fusion des comp. connexes */
38         } else
39             printf ("rejeter (%c, %c)\n", G[i].a, G[i].b);
40     }
41     return 0;
42 }
```

FIGURE 3 – Calcul d’un arbre couvrant