GIS2A3 — Structures de Données — Épreuve d'Avril 2014
Durée 2 heures — Aucun document autorisé

---

1 Exercices et questions de cours (9 points)

1.1 Complexité

Question 1 [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 2 [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 ?

1.2 Listes chaînées

On considère les déclarations suivantes :

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

Question 3 [2 pts]. Écrire une fonction `somme`, paramétrée par une liste L et qui retourne la somme des éléments de L.

Question 4 [1 pt]. Que fait la fonction suivante (la liste `dst` est supposée initialement vide) ?

```
void kanaputz (struct liste* dst, struct liste src)
{   struct maillon* M;
    M = src.tete;
    while (M != NIL)
    {   ajouter_en_tete (dst, M->valeur);
        M = M->next;
    }
}
```

1.3 Récursivité

Question 5 [2 pts]. Écrire une fonction récursive `puissance`, paramétrée par un double x, un entier positif ou nul d, qui calcule x^d à partir des relations de récurrence avec condition initiale suivantes :

$$\begin{aligned} x^0 &= 1 \\ x^d &= \text{le carré de } x^{d/2} \text{ (si } d \text{ est pair)} \\ x^d &= x \times x^{d-1} \text{ (si } d \text{ est impair)} \end{aligned}$$

1.4 Tables de hachage

On considère l'insertion des clés $\Box$, $\bigcirc$, $\triangle, *$ 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
$\Box$	3	5
$\bigcirc$	7	7
$\triangle$	7	9
$*$	4	3

Question 6 [1 pt]. Insérer les clés dans la table. Donner un schéma de la table résultat. Indiquer les éventuelles collisions.

Question 7 [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 ?

2 Problème (11 points)

Toutes les questions se rapportent au document préparatoire ci-joint.

On implante un graphe orienté, binaire, acyclique (GOBA) par un pointeur du type `struct goba*`, avec la définition suivante :

```
enum couleur { vert, orange, rouge };

struct goba {
    enum couleur color;
    char sommet;
    struct goba* succ1; /* premier successeur */
    struct goba* succ2; /* second successeur */
};
#define NIL (struct goba*)0
```

Les éléments de type `struct goba` sont appelés des *nœuds*. Les champs `succ1` et `succ2` peuvent chacun valoir `NIL` indépendamment l'un de l'autre. Le champ `color` permet de colorier les nœuds et éviter ainsi aux algorithmes de parcourir plusieurs fois les sous-graphes accessibles par plusieurs chemins, depuis la racine.

Question 8 [2 pts]. Écrire une fonction `colorier_goba`, paramétrée par un GOBA G, une couleur c (un élément du type `enum couleur`), et qui colorie tous les nœuds accessibles à partir de G avec c. Cette fonction-ci peut être amenée à parcourir plusieurs fois des sous-graphes d'un graphe donné.

On souhaite maintenant écrire une fonction qui imprime (exactement une fois) tous les sommets d'un GOBA G donné. L'idée est la suivante : commencer par colorier tous les nœuds accessibles à partir de G en vert puis, imprimer les sommets verts, grâce à un parcours inspiré des parcours d'ABR. À chaque fois qu'un sommet est imprimé, il suffit de le colorier en rouge pour éviter de l'imprimer une deuxième fois.

Question 9 [2 pts]. Écrire une fonction `imprimer_goba`, paramétrée par un GOBA G et qui imprime tous ses sommets. On conseille d'écrire deux fonctions : une fonction principale qui colorie d'abord tous les nœuds en vert, ainsi qu'une fonction récursive auxiliaire qui effectue l'affichage.

On s'intéresse maintenant au destructeur de GOBA. Si on l'écrivait à la façon du destructeur d'ABR, il ressemblerait à ceci :

```
void clear_goba (struct goba* G)
{   if (G != (struct goba*)0)
    {   clear_goba (G->succ1);
        clear_goba (G->succ2);
        free (G);
    }
}
```

Question 10 [1 pt]. Expliquer pourquoi cette fonction est incorrecte.

Question 11 [1 pt]. On pourrait imaginer de la corriger en utilisant une technique de coloriage, un peu comme pour l'affichage. Obtiendrait-on ainsi une fonction correcte ? Si oui, donner cette fonction. Si non, expliquer pourquoi.

On s'intéresse maintenant à l'algorithme du tri topologique. La donnée est un GOBA G. La sortie est un tableau T, muni d'un indice n. L'algorithme est implanté ici de façon itérative, en utilisant une pile. Les trois couleurs sont utilisées. Les nœuds qui n'ont pas encore été visités sont coloriés en vert. Ceux qui sont dans la pile sont coloriés en orange. Les sommets complètement traités sont en rouge. On range dans T un sommet dès qu'il est colorié en rouge.

```
function tri_topologique (G, T, n)
Donnée : G. Résultat : T et n
begin
  initialiser n à zéro et colorier tous les nœuds en vert
  vider la pile
  colorier G en orange et l'empiler
  while la pile n'est pas vide do
    soit B le nœud en sommet de pile
    B ne peut pas avoir de successeur orange sinon, le graphe ne serait pas acyclique
    if le premier successeur de B est vert then
      colorier ce successeur en orange et l'empiler
    elif le second successeur de B est vert then
      colorier ce successeur en orange et l'empiler
    else
      dépiler l'élément en sommet de pile (c'est-à-dire B)
      le colorier en rouge
      le ranger dans T à l'indice n et incrémenter n
    end if
  end do
end
```

Avant de coder l'algorithme ci-dessus en C, on souhaite concevoir un module de pile qui permette de l'implanter facilement.

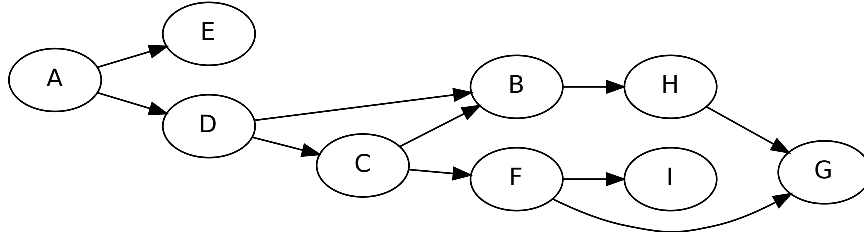
Question 12 [1 pt]. Donner la déclaration C d'une structure qui puisse servir à implanter une pile de `struct goba*`. Spécifier cette structure de données.

Question 13 [2 pts]. Donner les prototypes des fonctions exportées du module de pile. Ces fonctions doivent permettre de réaliser toutes les opérations de pile exécutées par l'algorithme, et de masquer ainsi totalement l'implantation de cette pile.

Question 14 [2 pts]. Coder la fonction tri_topologique en C en utilisant le module de pile.

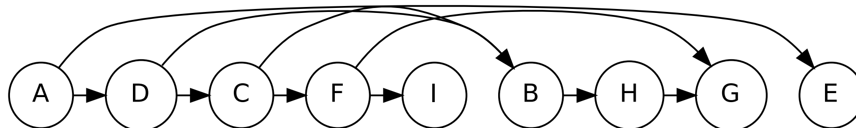
Graphes Binaires Orientés Acycliques

On considère des graphes orientés acycliques, binaires (c'est-à-dire que chaque nœud a au plus deux fils) et munis d'une racine (c'est-à-dire d'un noeud à partir duquel tous les autres nœuds sont accessibles). Un exemple (la racine est A) est donné ci-dessous :



L'algorithmique sur ces graphes est proche de l'algorithmique rencontrée sur les arbres binaires de recherche mais il y a quand même quelques différences. Par exemple, si on cherche un nœud à partir de son nom, il est nécessaire, dans le pire des cas, de parcourir tout le graphe. Le destructeur aussi présente des difficultés.

Beaucoup de difficultés algorithmiques se résolvent très facilement grâce à un sous-algorithme, spécifique aux graphes orientés acycliques : le *tri topologique*. Il consiste à ranger les n sommets du graphe dans un tableau T, entre les indices 0 et n, de telle sorte que tous les arcs soient orientés dans la même direction (de la droite vers la gauche, ou la gauche vers la droite). Un exemple est donné ci-dessous. Le résultat est normalement un simple tableau de sommets. On a dessiné les arcs pour plus de lisibilité :



L'algorithme du tri topologique est une application directe du parcours en profondeur d'abord du graphe. Il s'implante avec une pile. On peut colorier les sommets du graphe pour éviter de parcourir deux fois le même sous-graphe (problème qu'on ne se poserait pas avec un arbre). Il suffit en fait de ranger un sommet dans T dès qu'il est dépilé. En C, le coloriage des sommets peut s'implanter en ajoutant aux nœuds un champ du type suivant :

```
enum couleur { rouge, vert, bleu };
```