GIS3 — Structures de Données — Épreuve d'Avril 2013
Durée 2 heures — Aucun document autorisé

---

1 Petits exercices (11 points)

Question 1 [2 pts]. Expliquer comment il est possible d'implanter une file de N éléments dans un tableau, de telle sorte que chaque opération de file ait une complexité en $\Theta(1)$. Aucune ligne de code n'est demandée.

1.1 Tables de hachage

On considère l'insertion des clés □, ◯, △, ∗ dans une table de hachage de $N = 8$ 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
□	2	3
◯	5	5
△	5	5
∗	4	1

Question 2 [1 pt]. Insérer les alvéoles dans la table. Donner un schéma de la table résultat. Indiquer les éventuelles collisions.

Question 3 [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 ?

1.2 Files de priorité

On considère le tableau $T = \{35, 48, 50, 52, 120, 51\}$.

Question 4 [1 pt]. Donner une représentation graphique de ce minimier.

Question 5 [1 pt]. Enfiler 17. Dessiner le minimier obtenu.

Question 6 [1 pt]. Défiler un élément à partir du minimier de la question précédente. Quel élément sort ? Dessiner le minimier obtenu.

1.3 Arbres binaires de recherche

On considère la déclaration suivante :

```
struct abr {
    long valeur;
    struct abr* gauche;
    struct abr* droit;
};
```

Question 7 [2 pts]. Écrire une fonction récursive `nb_noeuds`, paramétrée par un `struct abr*`, qui retourne son nombre de nœuds.

Question 8 [2 pts]. Écrire une fonction récursive `affiche`, paramétrée par un `struct abr*`, qui imprime les valeurs de cet arbre, par valeurs croissantes.

2 Modification d'une implantation (3 points)

La Figure 3 contient le code C de deux fichiers `liste.h` et `liste.c`, réalisant un module de listes simplement chaînées, comme on les a étudiées en cours : chaque maillon contient un pointeur `next` vers le maillon suivant.

On souhaite modifier cette implantation pour obtenir un module de listes doublement chaînées : chaque maillon doit contenir, en plus du pointeur `next`, un pointeur `prev` vers le maillon précédent. Un exemple est donné Figure 1.

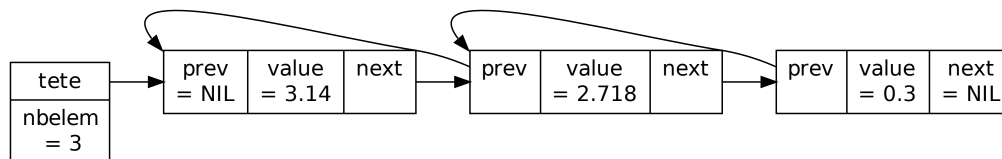

FIGURE 1 – Une liste doublement chaînée de trois éléments : 3.14, 2.718 et 0.3

Question 9 [3 pts]. Réaliser la modification demandée. Éviter de recopier tout le code de la Figure 3. Il est conseillé de reporter les modifications mineures directement sur la Figure 3 et d'incoporer cette feuille à la copie, en y indiquant vos nom et prénom.

3 Distance entre fichiers (6 points)

On souhaite écrire un logiciel qui prenne en entrée deux fichiers A et B (par exemple deux fichiers C écrits par deux étudiants, lors de la réalisation d'un projet) et qui calcule une "distance" entre ces fichiers.

Pour cela, on fait glisser une fenêtre de 10 caractères sur chacun des deux fichiers, ce qui revient à énumérer des chaînes de caractères de longueur 10 et on compare. La distance entre les deux fichiers est alors définie par la formule :

$$\frac{\text{nombre de chaînes présentes dans } A \text{ et dans } B}{\text{nombre de chaînes présentes dans } A \text{ ou dans } B}.$$

Par exemple, supposons que A comporte les 13 caractères "`le petit chat`" et que B comporte les 14 caractères "`le petit chien`". Les chaînes de 10 caractères présentes dans les deux fichiers sont listées ci-dessous. La distance entre A et B vaut donc 2/7.

```
le petit c      dans A et dans B
e petit ch      dans A et dans B
 petit cha      dans A seulement
petit chat      dans A seulement
 petit chi      dans B seulement
petit chie      dans B seulement
etit chien      dans B seulement
```

La distance entre les deux fichiers A et B peut être estimée grâce à l'algorithme décrit Figure 2.

```
function chaines_communes (A, B)
begin
  AetB = 0
  for chaque chaîne a présente dans A do
    for chaque chaîne b présente dans B do
      if a = b then
        AetB += 1
      end if
    end do
  end do
  return AetB
end
```

Figure 2 – Un algorithme élémentaire pour estimer le nombre de chaînes communes à deux fichiers A et B. Le nombre de chaînes présentes dans A ou B peut s'estimer en utilisant les tailles des fichiers.

Question 10 [1 pt]. On suppose que les deux fichiers sont de même longueur n et on s'intéresse au nombre $f(n)$ de comparaisons de chaînes. Donner un équivalent asymptotique de $f(n)$.

Question 11 [1 pt]. Parmi les structures de données étudiées en cours, en choisir une qui permette de réduire la complexité de l'algorithme précédent (conseil : lire les questions qui suivent avant de choisir). Réécrire l'algorithme de la Figure 2 (dans le même style de pseudo-langage) en utilisant cette structure de données.

Question 12 [1 pt]. Expliquer, du point de vue de la complexité, pourquoi le nouvel algorithme est meilleur. Remarque : suivant la structure de données choisie, vous pouvez déterminer un équivalent asymptotique de $f(n)$ dans le pire des cas ou considérer que le pire des cas n'est pas représentatif et vous intéresser à un cas moyen et, éventuellement, mener une analyse de complexité en moyenne.

Question 13 [3 pts]. Écrire un fichier d'entête ".h" pour votre structure de données. Spécifier la structure et les fonctions exportées.

NOM : PRÉNOM :

```
/* liste.h */

struct maillon
{   double value;
    struct maillon* next;
};

struct liste
{   struct maillon* tete;
    int nbelem;
};

extern void init_liste (struct liste*);
extern void clear_liste (struct liste*);
extern void ajouter_en_tete_liste (struct liste*, double);

/* liste.c */

void init_liste (struct liste* L)
{
    L->tete = (struct maillon*)0;
    L->nbelem = 0;
}

void ajouter_en_tete_liste (struct liste* L, double d)
{   struct maillon* nouveau;

    nouveau = (struct maillon*)malloc (sizeof (struct maillon));
    assert (nouveau != (struct maillon*)0);
    nouveau->value = d;
    nouveau->next = L->tete;
    L->tete = nouveau;
    L->nbelem += 1;
}

void clear_liste (struct liste* L)
{   struct maillon* courant;
    struct maillon* suivant;
    int i;

    courant = L->tete;
    for (i = 0; i < L->nbelem; i++)
    {   suivant = courant->next;
        free (courant);
        courant = suivant;
    }
}
```

FIGURE 3 – Un module minimal de listes simplement chaînées