GIS2A — Structures de Données — Épreuve d'Avril 2013
Durée 2 heures — Aucun document autorisé

---

1 Petits exercices (10 points)

1.1 Tables de hachage

On considère l'insertion des clés $\square$, $\bigcirc$, $\triangle$, $*$ 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
$\square$	2	3
$\bigcirc$	5	7
$\triangle$	5	7
$*$	4	1

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

Question 2 [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 ?

Question 3 [1 pt]. Expliquer en quelques mots comment gérer les collisions lorsqu'une table est gérée avec une technique de simple hachage.

1.2 Files de priorité

On considère le tableau $T = \{14, 109, 50, 52, 120, 51\}$.

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 Complexité

Question 7 [2 pts]. L'algorithme A a une complexité en temps en $\Theta(n)$. L'algorithme B a une complexité en temps en $\Theta(n^2)$. L'un de ces deux algorithmes est plus rapide que l'autre. Lequel ? Est-il plus rapide pour tout n ?

Question 8 [2 pts]. On s'intéresse à l'algorithme de la recherche d'un élément x dans un tableau T de n éléments. Que serait un pire des cas ? un meilleur des cas ?

2 Modification d'une implantation (5 points)

La Figure 1 contient le code C de deux fichiers `abr.h` et `abr.c` réalisant un module d'arbres binaires de recherche. On souhaite modifier cette implantation afin de gérer les suppressions d'éléments : le type `struct abr` doit contenir un booléen `detruit` (valant `false` par défaut). Lorsque ce booléen vaut `true`, la valeur présente dans la structure est considérée comme détruite.

Question 9 [3 pts]. Réaliser la modification demandée. Indications :
- pour la fonction `afficher`, n'imprimer que les éléments non détruits ;
- pour la fonction `ajouter`, si la valeur de A est égale à x mais est considérée détruite, il suffit de changer la valeur du champ `detruit`.

Éviter de recopier tout le code de la Figure 1. Il est conseillé de reporter les modifications mineures directement sur la Figure et d'incorporer cette feuille à la copie, en y indiquant vos nom et prénom.

Question 10 [2 pts]. Écrire une fonction `supprimer`, paramétrée par un entier x, un `struct abr*` A et qui supprime x de A, si x est présent dans A.

3 Réapprovisionnement des stations de V'Lille (5 points)

Un camion transportant des vélos est chargé de parcourir un ensemble de stations de V'Lille afin de déposer des vélos dans les stations trop vides et de prélever des vélos dans les stations trop pleines. On cherche à écrire une partie d'un logiciel qui détermine l'ordre de visite des stations (une station peut être visitée plusieurs fois). Pour le camion, les déclarations sont les suivantes :

```
struct camion {
    int nombre_velos;
    int nombre_visites;
};
```

Le champ `nombre_velos` contient le nombre de vélos présents dans la remorque du camion. Le champ `nombre_visites` contient le nombre de stations visitées depuis le début de la journée. Pour les stations, les déclarations sont les suivantes :

```
struct station {
    char* nom;          /* alloué dynamiquement */
    int numero_derniere_visite;
    int nombre_velos;
};
```

Le champ `nom` comporte le nom de la station. Le champ `nombre_velos` contient le nombre de vélos présents dans la station. Le champ `numero_derniere_visite` contient le « numéro » de la dernière visite faite par le camion : initialement, ce champ vaut zéro. À chaque fois que le camion s'arrête dans une station, le champ `nombre_visites` du camion est incrémenté de 1 et mémorisé dans le champ `numero_derniere_visite` de la station. La première station visitée reçoit donc le numéro 1, la deuxième le numéro 2 etc. Si une station est visitée plusieurs fois, seul le numéro de la dernière visite est mémorisé.

On souhaite gérer l'ordre de visite des stations par une file de priorité F : à chaque fois que le camion doit repartir d'une station, la prochaine station à visiter, S, est obtenue en défilant une station de F. La station S est ensuite ré-enfilée dans F, en vue d'une éventuelle future visite.

On veut écrire une fonction de priorité pour F qui assure que toutes les stations sont visitées équitablement : on veut éviter qu'une station S soit visitée deux fois alors qu'une station S' ne l'a pas encore été.

Question 11 [1 pt]. Préciser la règle simple qu'il suffit d'appliquer pour déterminer si une station est plus prioritaire qu'une autre.

Question 12 [1 pt]. Écrire la fonction de priorité correspondante en C.

On s'intéresse maintenant à la fonction de priorité plus sophistiquée suivante : on souhaite toujours que les stations soient visitées équitablement mais, en priorité, on veut éviter de visiter une station vide lorsque la remorque du camion est vide.

Question 13 [1 pt]. Préciser la règle qu'il suffit d'appliquer pour déterminer si une station est plus prioritaire qu'une autre.

Question 14 [1 pt]. Écrire la fonction de priorité correspondante en C (vous pouvez supposer que la structure décrivant le camion est passée à la fonction, comme paramètre supplémentaire).

Question 15 [1 pt]. Écrire un destructeur pour le type `struct station` en évitant les instructions inutiles.

NOM : PRÉNOM :

```
/* abr.h */

#include <stdbool.h>

struct abr {
    struct abr* gauche;
    struct abr* droit;
    int valeur;
};
#define NIL (struct abr*)0

extern bool rechercher (int, struct abr*);
extern struct abr* ajouter (struct abr*, int);
extern void afficher (struct abr*);

/* abr.c */

bool rechercher (int x, struct abr* A)
{   if (A == NIL)
        return false;
    else if (x == A->valeur)
        return true;
    else if (x < A->valeur)
        return rechercher (x, A->gauche);
    else
        return rechercher (x, A->droit);
}

struct abr* ajouter (struct abr* A, int x)
{   if (A == NIL)
    {   A = (struct abr*)malloc (sizeof (struct abr));
        A->gauche = A->droit = NIL;
        A->valeur = x;
    } else if (x < A->valeur)
        A->gauche = ajouter (A->gauche, x);
    else if (x > A->valeur)
        A->droit = ajouter (A->droit, x);
    return A;
}

void afficher (struct abr* A)
{   if (A != NIL)
    {   afficher (A->gauche);
        printf ("%d ", A->valeur);
        afficher (A->droit);
    }
}
```

FIGURE 1 – Un module minimal d'arbres binaires de recherche