

Nom :

Année scolaire : 2007–2008


Prénom :

Date : 23 juin 2008

Module INF₄₄₆
Session de juin

Programmation avancée en C

Contrôle de connaissance¹ de 45 minutes

 ERCI de répondre (au moins) dans les blancs.

Lire tout le sujet en entier du début à la fin, en commençant à la première page et jusqu'à la dernière page, avant de commencer à répondre : cela peut vous donner de l'inspiration et vous permettre de mieux allouer votre temps en fonction de vos compétences.

Plus la solution est simple, mieux c'est. Pensez aussi à la complexité algorithmique des solutions que vous proposez (cela ne sert à rien de refaire calculer par l'ordinateur plusieurs fois la même chose sans raison valable...).

Chaque question sera notée entre 0 et 10 et la note globale sera calculée par une fonction des notes élémentaires. La fonction définitive sera choisie après correction des copies.

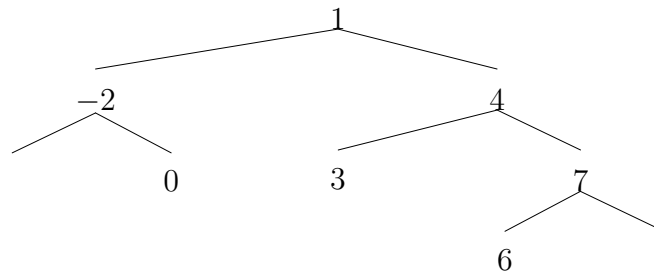
Attention : tout ce que vous écrirez sur cette copie pourra être retenu contre vous, voire avoir une influence sur la note d'INF₄₄₆ !

1 Généralités

Question 1 : Afin de ressembler plus à du Java, un programmeur a envie de déclarer un type `string` pour représenter les chaînes de caractère. Écrire la ligne de `typedef` qui va bien pour remplacer

```
1 char * chaine;
```

¹Avec document, sans triche, sans copie sur les voisins, sans micro-ordinateur portable ou non, sans macro-ordinateur, sans téléphone portable ou non, sans oreillette de téléphone ni de dictaphone, sans talkie-walkie, sans télépathie, sans métempsycose, sans pompe. Sont tolérés : anti-sèche, tatouage ou vêtement imprimé en rapport avec le sujet, mouchoir de poche pré-imprimé, piercing ou scarification en rapport avec l'INF₄₄₆, bronzage à code barre ou 2D...

FIG. 1 – Exemple d’arbre représentant l’ensemble $(1, -2, 4, 7, 6, 0, 3)$.

par

```
1 string chaine;
```

(2 minutes) □

→

2 Manipulation d’arbres

On se propose dans la suite de construire un système de manipulation d’ensembles d’entiers. Afin de permettre un accès rapide à un élément de l’ensemble, on réalise ce système sous forme d’arbre binaire ordonné. L’idée est d’avoir pour un nœud donné avec une valeur v que des valeurs $< v$ dans le sous-arbre accessible par le fils gauche et que des valeurs $> v$ dans le sous-arbre accessible par le fils droit.

Par exemple, l’ensemble $(1, -2, 4, 7, 6, 0, 3)$ peut être réalisé sous la forme de la figure ??.

On utilisera la structure suivante pour définir un nœud de l’arbre :

```

1 struct node {
    int value;
3     struct node *left;
    struct node *right;
5 };
typedef struct node node;
```

et les pointeurs `left` et `right` pointant respectivement vers les sous-arbres gauche et droit. Si un nœud n’a pas de sous-arbre, le pointeur correspondant vaut `NULL` (comme par exemple sur la figure ?? où la branche gauche du nœud avec la valeur -2 n’a pas d’élément). L’ensemble vide (donc l’arbre vide dans cette réalisation) est par généralisation² aussi représenté par `NULL`. Un ensemble (vide au départ) est donc déclaré par :

```
1 node * ens = NULL;
```

Pour faire plus joli on pourrait aussi écrire :

²Et simplification de la programmation, vous verrez...

```
1 typedef node * ensemble;  
   ensemble ens = NULL;
```

Question 2 : Écrire une fonction

```

1  void affiche(node *n) {
2      ...
3  }

```

qui affiche la valeur de l'entier du nœud n .

Toutes les questions sont relativement indépendantes afin de permettre de répondre aux questions mêmes si les précédentes sont restées sans réponse ou fausses. Les réponses demandées sont courtes mais demandent un peu de réflexion... (6 minutes) ☐

→

Question 3 : On veut maintenant écrire une fonction

```
1 void apply(node *n, void (*f)(node *))
```

qui parcourt par valeur croissante tous les éléments de l'arbre passé en paramètre et qui applique une fonction passée en paramètre (par pointeur) à chaque nœud rencontré. Aide : `apply` (un grand classique de la programmation fonctionnelle) sera évidemment une fonction récursive...

Afficher tous les éléments d'un ensemble e par ordre croissant s'exprimera donc simplement par

```
1  apply(e, affiche)
```

Puissant, non ?

(10 minutes) \square

→

→
→
→
→

Question 4 : Sur le même principe³, on veut pouvoir compter le nombre d'éléments de l'ensemble avec une méthode du style

```
1 // Valeur temporaire utilisée par count_elem() :
  int nombre;
3 int count(ensemble ens) {
    nombre = 0;
5   apply(ens, count_elem);
    return nombre;
7 }
```

Écrire le contenu de la fonction `count_elem()` pour que `count()` renvoie le nombre d'éléments de l'ensemble. (2 minutes) □

→
→
→
→

Question 5 : Toujours sur le même principe⁴, on veut faire la somme des éléments de l'ensemble avec une méthode du style

```
1 // Accumulateur temporaire utilisé par add_elem() :
  int somme;
3 int add(ensemble ens) {
    somme = 0;
5   apply(ens, add_elem);
    return somme;
7 }
```

Écrire le contenu de la fonction `add_elem()` pour que `add()` renvoie la somme des éléments de l'ensemble. (2 minutes) □

→
→
→
→

Question 6 : Au cas où vous n'auriez pas compris le concept⁵, écrire une fonction pour savoir si un élément est dans un ensemble ou pas qu'on utilisera ainsi :

³C'est trop beau !

⁴Ah, c'est trop fort !

⁵Mais que c'est bon !

```

1 #include <stdbool.h>
  // Booléen temporaire utilisée par inefficent_find() :
3 bool found_elem;
  // Met ici la valeur à chercher pour la rendre accessible à find_elem() :
5 int value_to_find;
  int inefficent_find(ensemble ens) {
7     found_elem = false;
      apply(ens, find_elem);
9     return found_elem;
  }

```

Écrire le contenu de la fonction `find_elem()` pour que `inefficent_find()` renvoie `true` si la valeur a été trouvée et `false` autrement.

Notons que c'est une recherche totalement inefficace car avec une complexité en $\mathcal{O}(n)$ qui n'exploite pas du tout le fait que les valeurs sont triées dans l'arbre représentant l'ensemble. Mais patience... (2 minutes) \square

→
→
→
→

Question 7 : On veut maintenant faire des recherches dichotomiques pour avoir un temps de recherche en $\mathcal{O}(\log n)$ dans l'ensemble pour savoir si un élément s'y trouve ou pas. Écrire une fonction `find()` (récursive, évidemment) qui trouve le nœud contenant la valeur si elle se trouve dans l'ensemble et renvoie l'adresse de ce nœud, ou bien renvoie l'adresse du pointeur (NULL) dans le nœud le plus profond existant par où on serait allé chercher le nœud avec la valeur si elle s'était trouvée dans l'ensemble. Afin de simplifier les choses, la fonction doit avoir le prototype

```

1 void find(node **n, int value, struct find_t * result)

```

et modifier la structure passée en paramètre par adresse :

```

1 struct find_t {
    bool found;
3     union {
        node *found;
5         node **empty_leaf;
    } address;
7 };

```

Si le nœud est trouvé, il faut mettre `found` à `true` et l'adresse du nœud dans `address . found` sinon mettre `found` à `false` et dans `address . empty_leaf` l'adresse du pointeur NULL vers la feuille où aurait dû se trouver le nœud manquant⁶. Vous devez écrire cette fonction `find()` qui va s'utiliser de la manière suivante :

⁶Lire la question suivante pour mieux comprendre ce point et trouver une justification.

→
→
→
→

Question 8 : Jusqu'à présent on n'a fait qu'utiliser des ensembles qu'on supposait préexistants. On va utiliser la fonction précédente pour écrire une fonction d'insertion d'un élément de valeur v dans un ensemble⁷ :

```

1 void insert(node **ens, int v) {
2     struct find_t find_info;
3     find(ens, v, &find_info);
4     if (find_info.found)
5         // Pas la peine de continuer, la valeur s'y trouve déjà !
6         return;
7     else
8         add_node_at(find_info.address.empty_leaf, v)
9 }

```

Cette fonction s'utilise de la manière suivante pour rajouter 4 dans un ensemble `ens` :

```

1 insert(&ens, 4)

```

On a un double pointeur afin que cette fonction puisse marcher aussi sur l'arbre vide au départ : dans ce cas il faut bien que la fonction `insert` modifie la valeur du pointeur ensemble passé en paramètre pour le faire passer de son ancienne valeur `NULL` à l'adresse de la structure du premier élément.

Vous devez donc écrire la fonction

```

1 add_node_at(node **n, int value)

```

qui alloue une structure nœud et l'initialise pour stocker la valeur *value* et la raccroche à l'arbre en écrivant son adresse dans le pointeur dont on a passé l'adresse. (5 minutes) □

→
→
→
→
→
→
→
→
→
→
→
→
→
→
→

⁷Si la valeur s'y trouve déjà, on ne fait rien : on fait des ensembles, pas des listes...

→
→
→
→
→
→

Question 9 : Maintenant que vous savez insérer, écrire une fonction qui crée une union de 2 ensembles⁸ simplement en utilisant des `apply` d'une fonction qui appelle `insert` sur la destination.

Écrire la fonction `union_ensemble()` afin qu'elle soit utilisable ainsi :

```
1 ensemble destination = NULL;
  ...
3 // On crée des ensembles ens1 et ens2
  ...
5 union_ensemble(&destination, ens1, ens2);
  // Affiche le résultat de l'union pour vérifier :
7 apply(destination, affiche);
```

(5 minutes) □

→
→
→
→
→
→
→
→
→
→
→

On a maintenant les briques de bases pour pouvoir faire plein de choses amusantes.

Ce genre de programmation fonctionnelle sert par exemple dans le moteur de recherche Google à implémenter leur méthodes reposant sur l'usage de *map-reduce*, où on aurait remplacé les entiers par des mots clés ou des identifiants de pages. Évidemment il faut ensuite distribuer efficacement les données et paralléliser les calculs sur des dizaines de milliers d'ordinateurs...

C'est aussi la base de langage fonctionnels comme Lisp (utilisé dans Emacs...) ou des langages parallèles de type flot de données qui reviennent à la mode pour programmer les cartes graphiques modernes.

Durée totale estimée : 44 minutes.

⁸Si vous êtes très forts en C, écrire une fonction générique qui fait une union d'un nombre quelconque d'ensembles passés e paramètre avec des varags.