Programmation en C

Examen du 10/01/2013 - 3h

Bien qu'inutiles, les notes de cours sont autorisées, pas les livres. Dans une certaine mesure, la correction tiendra compte de la lisibilité du code produit et de son efficacité en termes de temps d'exécution et de quantité de mémoire utilisée. Chaque question peut être traitée indépendamment. Ne perdez pas de temps à écrire des `main` pour les fonctions demandées, ils ne seront pas pris en compte.

I - Tableaux

Exercice 1

Etant donnés deux tableaux `t` et `u` de même taille, on dira que `u` est la *propagation droite* de `t` si pour chaque indice `i`, `u[i]` vaut la somme des éléments de `t` dont l'indice est inférieur ou égal à `i`. Ecrire de manière optimale les deux fonctions suivantes :

1. `int est_propagation(int *t, int *u, int n)`.

   Cette fonction suppose que `t, u` sont les adresses de deux tableaux à `n` éléments. Elle doit renvoyer `1` si le tableau d'adresse `u` est la propagation droite de celui d'adresse `t`, et `0` sinon.

2. `int *propagation_inverse(int *u, int n)`.

   Cette fonction suppose que `u` est l'adresse d'un tableau à `n` éléments. Elle doit allouer, remplir puis renvoyer l'adresse d'un tableau dont `u` est la propagation droite.

Exercice 2

Ecrire une fonction `int *filtrage(int *t, int n, int *pos, int m)` effectuant le traitement suivant.

Cette fonction suppose que `t` est l'adresse d'un tableau à `n` éléments, et `pos` celle d'un tableau à `m` éléments. Elle doit allouer, remplir et renvoyer l'adresse d'un tableau contenant, dans l'ordre, la suite de toutes les valeurs du tableau d'adresse `t` dont la position apparaît dans le tableau d'adresse `pos`. Le contenu du tableau d'adresse `pos` sera supposé quelconque : avec des répétitions, des valeurs en dehors des limites du premier tableau, etc. Par exemple, si le premier tableau contient `{42, 27, 12, 3, 12}` et le second `{7, 2, 0, -1, 4, 2}`, le tableau résultant contiendra `{42, 12, 12}`, soit `{t[0], t[2], t[4]}`.

Le choix de la méthode est libre, mais votre solution devra rester simple et efficace. Une indication : il existe une solution sans boucles imbriquées, avec une allocation mémoire.

II - Chaînes de caractères

Sans relecture ou allocation

Les fonctions ci-dessous supposent que `s`, `t` sont des adresses de chaînes de caractères. Elles ne doivent ni déclarer, ni allouer de nouveaux tableaux, et chaque chaîne ne pourra être lu qu'une fois au plus - pas de retours en arrière, pas de `strlen`.

Exercice 3

Etant donnés deux textes s et t, on dira que s est un *pré-acrostiche* de t si les deux textes contiennent le même nombre de lignes, et si pour chaque numéro de ligne i, la i-ième ligne de t est préfixe de la i-ième ligne de s.

Ecrire une fonction `int est_pre_acro(char *s, char *t)`. Cette fonction suppose que les chaînes d'adresses `s`, `t` représentent deux textes dont toutes les lignes, y compris la dernière, se terminent par un retour à la ligne `'\n'`. Elle doit renvoyer `1` si le premier texte est pré-acrostiche du second, et `0` sinon.

Exercice 4

Etant donnée une chaîne de caractères s, appelons *lipossible* de s toute chaîne obtenue en supprimant de s toutes les occurrences d'un caractère de s. Par exemple, *"le chat aime les ombres"* est un lipossible de *"le chant anime les nombres"*, obtenu par suppression des occurrences de *'n'*.

Ecrire une fonction `int lipossible(char *s, char *t)` renvoyant `1` si la chaîne d'adresse `t` est un lipossible de celle d'adresse `s`, et `0` sinon. N'oubliez pas de respecter les consignes ci-dessus.

Avec relecture partielle et allocation

Un peu plus difficile, la fonction ci-dessous suppose que `s` est l'adresse d'une chaîne de caractères, mais contrairement aux précédentes, elle peut demander une allocation mémoire et relire une partie des données, pourvu que ces relectures soient justifiées et réduites au minimum. La chaîne devra être préservée.

Exercice 5

Etant donné un texte s, on dira que ce texte contient une *colonne constante* s'il existe un i tel que les caractères de position i dans chaque ligne de s soient deux-à-deux égaux.

```
par exemple, ce texte contient a la
position dix-neuf une colonne ne
mentionnant que la lettre 'e', i.e.
une colonne constante, au sens precedent.
```

Ecrire `int col_cst(char *s)`. Cette fonction suppose que la chaîne d'adresse `s` représente un texte dont toutes les lignes, y compris la dernière, se terminent par un retour à la ligne `'\n'`. Elle doit renvoyer `1` si le texte contient au moins une colonne constante, et `0` sinon.

III - Listes chaînées

On considère le type usuel permettant de représenter en C des listes chaînées d'entiers, avec les conventions habituelles :

```
struct cell {
  int clef;
  struct cell *suiv;
};
```

Les fonctions ci-dessous doivent être écrites par récurrence pure : pas de boucles, pas de variables statiques. Chaque fonction ne peut appeler qu'elle-même et, si nécessaire `free`. Aucune ne nécessite d'allocation mémoire. Attention aux cas limites et à l'ordre des opérations.

Exercice 6

Ecrire

```
struct cell *elagage_gauche(struct cell *pc)
```

renvoyant la liste obtenue en supprimant, lorsque cette dernière cellule existe, toutes les cellules de `pc` sauf la dernière – *i.e.* si la liste contient au plus une cellule, cette fonction se contente de renvoyer `pc`. En fin d'appel, l'espace mémoire pour le stockage des cellules supprimées devra avoir été intégralement libéré.

Exercice 7

Ecrire

```
struct cell *elagage_alterne(struct cell *pc)
```

renvoyant la liste obtenue en supprimant une cellule sur deux de `pc`, à partir de la première. Si par exemple la suite de clefs est $\langle 2, 1, 4, 3, 42 \rangle$, la liste résultante aura pour suite de clefs $\langle 1, 3 \rangle$. Là encore, l'espace mémoire pour le stockage des cellules supprimées devra avoir été intégralement libéré en fin d'appel.

Exercice 8

On souhaite généraliser la fonction précédente. On numérote les cellules de listes à partir de `0`. Ecrire

```
struct cell *elagage(struct cell *pc, int nc, int k)
```

renvoyant la liste obtenue à partir de `pc` en supprimant une cellule sur `k`, à partir de la cellule de numéro `nc`. Cette fonction suppose que `nc` et `k` valent au moins `0` – noter que si `k` est nul, *toutes* les cellules à partir de celle de numéro `nc` devront avoir été libérées en fin d'appel.