Programmation en C

Examen du 18/06/2012 - 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`, `u`, on dit que `t` peut être obtenu par *collage dans* `u` si le tableau `u` est de la forme

$$\{x_0;\ \ldots;\ x_{m\,-\,1}\}$$

avec `t` de la forme

$$\{x_0;\ \ldots;\ x_{i\,-\,1};\ y_0;\ \ldots;\ y_{p-1};\ x_i;\ \ldots;\ x_{m\,-\,1}\}$$

Ecrire une fonction `int est_collage(int *t, int n, int *u, int m)`. Cette fonction suppose que `t` est l'adresse d'un tableau à `n` éléments, et que `u` est l'adresse d'un tableau à `m` éléments. Elle doit renvoyer `1` si `t` peut être obtenu par collage dans `u`, et `0` sinon. Le contenu des tableaux `t`, `u` ne devra être lu qu'au plus une fois – plus précisément, cette fonction devra être linéaire en `n`.

II - Chaînes de caractères

Les fonctions qui suivent supposent que `s` est l'adresse d'une texte dont chaque ligne se termine par `'\n'`, y compris la toute dernière ligne. Le texte est supposé écrit sans accents ni majuscules, et tous les caractères autres que `'a'`...`'z'` (ponctuations, espaces, etc.) seront considérés comme des séparations.

Exercice 2

Rappelons qu'un texte est une anagramme d'un autre texte si, sans tenir compte des séparations, il est possible de recomposer le premier à partir du second par simple déplacements de lettres.

Appelons *héréditairement anagrammatique* un texte dont chaque ligne est une anagramme de chaque autre ligne. Par exemple (aux accents près, et extrait de la revue d'écriture Soleils & Cendre de mars 2010) :

> *la terre est bleue comme une orange,*
> *l'orage seme brune nue, ecarte le mot,*
> *en somme la rage, brule, ecorne et tue,*
> *une : l'ecart en mot remue le sage orbe,*
> *la rage butee cerne, rue et nomme l'os,*
> *l'ombre use et glace une arene morte.*

Ecrire une fonction `int est_ha(char *s)` renvoyant `1` si le texte d'adresse `s` est héréditairement anagrammique, et `0` sinon.

Exercice 3

Appelons *acrostiche dualement anagrammatique* tout texte vérifiant la conditions suivante :

- Le texte formé de la suite des premières lettres de chaque ligne est une anagramme du texte formé de la suite des dernières lettres de chaque ligne.

Ecrire une fonction `int est_ada(char *s)` renvoyant `1` si le texte stocké à l'adresse `s` est un acrostiche dualement anagrammatique, et `0` sinon. Le texte ne devra être lu qu'une seule fois.

Exercice 4

Appelons *acrostiche dualement palindromique* tout texte vérifiant la condition suivante :

- La suite des premières lettres de chaque ligne est égale à la suite inversée des dernières lettres de chaque ligne. Autrement dit, si la suite des premières lettres de chaque ligne est $c_0 \dots c_{n-1}$ et si la suite des dernières lettres de chaque ligne est $d_0 \dots d_{n-1}$, on a $c_0 \dots c_{n-1} = d_{n-1} \dots d_0$.

Ecrire une fonction `int est_adp(char *s)` renvoyant `1` si le texte stocké à l'adresse `s` est un acrostiche dualement palindromique, et `0` sinon. Cette fonction pourra utiliser `str_len`, mais pas plus d'une fois, en tout début de fonction. Elle devra ensuite renvoyer sa réponse le plus rapidement possible. Elle ne doit pas utiliser de tableau auxiliaire – ni déclaré, ni alloué dynamiquement.

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, `malloc` et/ou `free`. Attention aux cas limites.

Exercice 5

On numérote les cellules et les clefs d'une liste à partir de `0`. Appelons *fusion à la position* `n` l'opération consistant, étant donnée une liste `pc` contenant au moins `n + 2` cellules, à remplaçer sa clef de rang `n` par la somme des clefs de rangs `n` et `n + 1`, puis à supprimer sa cellule de rang `n + 1`.

Par exemple, si la suite des clefs de `pc` est $(1, 7, 2, 3, 0)$, une fusion à la position 2 transformera cette suite de clefs en $(1, 7, 5, 0)$.

Ecrire une fonction `void fusionner(struct cell *pc, int n)`. Lorsque cette opération est possible, cette fonction doit effectuer sur `pc` une fusion à la position `n`, en libérant l'espace mémoire alloué pour la cellule de rang `n + 1`. La liste ne devra être parcourue qu'au plus une fois, et laissée telle quelle si l'opération est impossible.

Exercice 6

Inversement, et toujours en numérotant les cellules et les clefs d'une liste à partir de `0`, appelons *déploiement à la position* `n` l'opération consistant, étant donnée une liste `pc` contenant au moins `n + 1` cellules, à faire décroître de `1` sa clef de rang `n`, puis à insérer immédiatement après cette clef une nouvelle clef de valeur `1`.

Par exemple, si la suite des clefs de `pc` est $(1, 7, 2, 3, 0)$, une expansion à la position 1 transformera cette suite de clefs en $(1, 6, 1, 2, 3, 0)$.

Ecrire une fonction `void deployer(struct cell *pc, int n)`. Lorsque cette opération est possible, cette fonction doit effectuer sur `pc` un déploiement à la position `n`, en allouant l'espace-mémoire pour la clef insérée. La liste ne devra être parcourue qu'au plus une fois, et laissée telle quelle si l'opération est impossible.