Programmation en C

Examen du 17/06/2013 - 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 - Chaînes de caractères

Les fonctions ci-dessous supposent que `s` est l'adresse d'une chaîne de caractères. Ce qu'on appelle *occurrence* dans une chaîne s est simplement la donnée d'un caractère c et d'une position dans s à laquelle apparaît c. La *distance* entre deux occurrences est le nombre de positions intercalées entre celles-ci. Par exemple, dans `aaba`, les deux premières occurrences de `a` sont à distance 0, la seconde et la troisième sont à distance 1.

Sans relecture ni allocation

Exercice 1

Etant donnés une chaîne s et un caractère c, *l'espacement en c de s* est défini comme la distance minimale entre deux occurrences de c dans s. Par convention, cet espacement est égal à `-1` si c n'apparaît pas plus d'une fois dans s.

Par exemple, si s est `abbac`, l'espacement en `a` de s vaut 2, l'espacement en `b` vaut `0`, et l'espacement en toute autre caractère (même en `c`) vaut `-1`.

Ecrire une fonction `int espacement(char *s, char c)`, renvoyant l'espacement en `c` de `s`. Cette fonction ne doit ni déclarer, ni allouer de nouveaux tableaux, et la chaîne ne pourra être lu qu'une fois au plus - pas de retours en arrière, pas de `strlen`.

Sans relecture, avec allocation(s)

Exercice 2

Ecrire une fonction `int *espacements(char *s)`. Elle doit allouer, remplir et renvoyer l'adresse d'un tableau `t` à 256 éléments construit de la manière suivante : `t[0]` vaut `-1` ; pour chaque `i` entre `1` et `255`, si `i` est le code ASCII du caractère `c`, alors `t[i]` vaut l'espacement en `c` de la chaîne d'adresse `s`.

Cette fonction doit évidemment allouer au moins un tableau, et peut en allouer plusieurs, mais la chaîne ne devra être lu qu'une seule fois - vraiment une seule, à vous de trouver comment.

Exercice 3

Ecrire une fonction `int *distances(char *s, int len)`. Cette fonction suppose que la chaîne d'adresse `s` est de longueur `len`. Elle doit allouer, remplir et renvoyer l'adresse d'un tableau `t` à `n` éléments construit de la manière suivante : si `s[i]` vaut `c`, et si la position `i` n'est pas celle de la dernière occurrence de `c`, alors `t[i]` vaut la distance entre cette occurrence de `c` et la suivante ; sinon `t[i]` vaut `-1`. Par exemple, si la chaîne est `abbac`, le tableau résultant sera `[2,0,-1,-1,-1]`.

Cette fonction doit allouer un tableau et peut en allouer plusieurs, mais la chaîne ne devra être lu qu'une seule fois - c'est possible, en mémorisant les bonnes informations. Noter que la longueur de la chaîne étant connue, il est inutile de la recalculer.

Avec relecture partielle, sans allocation

On suppose à présent que la chaîne d'adresse `s` est un texte découpé en lignes. Chaque ligne, y compris la dernière, se termine par un unique `\n`, et il peut y avoir des lignes vides - qui, mêmes vides, comptent comme des lignes.

Exercice 4

Etant donné un texte s découpé en lignes, on dira que ce texte forme un *super-palindrome* si la première ligne est égale à la dernière, la seconde à l'avant-dernière, la troisième à l'avant-avant-dernière, etc.

```
ce texte est bien un super-palindrome.
il verifie la condition precedente, et donc,
il verifie la condition precedente, et donc,
ce texte est bien un super-palindrome
```

Ecrire `int super_palindrome (char *s)` renvoyant `1` si la chaîne d'adresse `s` est un super-palindrome, et `0` sinon. Cette fonction ne devra faire aucune allocation, et minimiser le nombre de relectures de la chaîne.

Exercice 5 (bonus)

Cet exercice ne doit être traité que s'il vous reste du temps après avoir intégralement traité le reste, notamment la partie II. Appelons *mega-palindrome* tout super-palindrome dont chaque ligne forme aussi un palindrome. Prenez le temps de réfléchir à la structure du texte dans ce cas précis, décrivez-la, et écrivez une fonction de test associée la plus efficace possible.

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`. Attention aux cas limites et à l'ordre des opérations.

Exercice 6

Une liste `pc` sera dite *sans paliers* si elle ne contient pas deux clefs successives égales. C'est par exemple de la liste dont la suite de clefs est $\langle 1, 0, 2, 1\rangle$, mais pas si cette suite est $\langle 1, 0, 0, 2\rangle$. Ecrire

```
int sans_paliers(struct cell *pc)
```

renvoyant `1` si `pc` est sans paliers, et `0` sinon.

Exercice 7

Une liste `pc` sera dite *en dents de scie* si elle ne contient pas deux clefs successives et égales, et si pour chaque triplet c_1, c_2, c_3 de clefs successives, on a : ou bien $c_1 < c_2$ et $c_2 > c_3$; ou bien $c_1 > c_2$ et $c_2 < c_3$. C'est le cas par exemple de la liste dont la suite de clefs est $\langle 1, 0, 2, 1\rangle$, ou encore $\langle 1, 2, 0, 2\rangle$. Ecrire

```
int en_dents_de_scie(struct cell *pc)
```

renvoyant `1` si `pc` est en dents de scie, et `0` sinon.

Exercice 8

Ecrire

```
struct cell *jonction_voisins(struct cell *pc)
```

Si la liste `pc` ne contient pas plus d'une cellule, cette fonction se contente de renvoyer `pc`. Sinon, elle doit allouer et insérer entre chaque couple de cellules successives une nouvelle cellule, dont la clef sera la somme des clefs des deux cellules voisines. Si par exemple la suite de clefs de `pc` est $\langle 1, 0, 2, 1\rangle$ la liste résultante aura pour suite de clefs t $\langle 1, \underline{1}, 0, \underline{2}, 2, \underline{3}, 1\rangle$ (les clefs ajoutées sont soulignées).