Devoir libre : Élimination de doublons dans une liste

Samedi 12 mars 2011

L'objectif de ce devoir est de mettre en place une fonction permettant de construire une liste de chaînes de caractères telle qu'une chaîne donnée ne peut y apparaître plus d'une fois.

1 Opérations sur les chaînes de caractères

En C les chaînes de caractères sont codées comme des tableaux dont les éléments sont de type `char`. La fin de la chaîne ne correspond pas à la fin du tableau et est explicitement indiquée par un caractère spécial, le caractère nul `\0`. Ainsi une chaîne de longueur n est codée dans un tableau de longueur au moins $n+1$. Par exemple la déclaration

```
char s[5] = "abc";
```

produit en mémoire un pointeur s désignant la première case d'un tableau de la forme

'a'	'b'	'c'	\0	?

. Dans cette partie nous proposons d'implémenter plusieurs opérations courantes sur les chaînes.

1.1. La longueur d'une chaîne est le nombre de caractères qu'elle contient. Dans l'exemple précédent, la longueur de `s` est 3. Écrivez la fonction `int strlen(char* s)` qui calcule la longueur d'une chaîne. Quelle est sa complexité en opérations de comparaisons ?

1.2. Écrivez la fonction `char* strcpy(char* s)` qui produit une copie en mémoire de la chaîne passée en argument. Représentez l'état de la mémoire après les instructions suivantes :

```
char u[5] = "aaa";
char* v = strcpy(u);
```

1.3. Enfin, nous aurons besoin d'une fonction de comparaison entre chaînes, devant indiquer pour deux chaînes u et v si $u < v$, $u = v$ ou $u > v$ selon l'ordre lexicographique. La fonction devra retourner respectivement -1, 0 ou 1. Le prototype attendu est `int strcmp(char* u, char* v)`. On donnera une version en style récursif, et une version en style itératif.

2 Listes de chaînes de caractères

2.1. On souhaite maintenant construire des listes de chaînes de caractères. Ces listes correspondent typiquement aux lignes d'un fichier. Proposer un type `list` permettant de représenter cette structure en mémoire. Il est recommandé de s'inspirer de la structure de liste chaînée vue en cours.

2.2. Illustrer votre choix en donnant la représentation en mémoire d'une liste à 2 éléments.

2.3. Écrire un constructeur pour vos listes, de prototype `list cons(char* s, list l)`. Attention, on attend du constructeur qu'il copie la chaîne donnée en argument.

2.4. Proposer une fonction qui calcule le nombre total de caractères dans une liste.

2.5. Écrire une fonction `void free_list(list l)` libérant **toute** la mémoire occupée par une liste.

3 Tri de la liste

La plupart des tris efficaces requièrent de pouvoir accéder rapidement à n'importe quel élément de la séquence à trier. Les listes ne remplissent pas cette condition puisqu'on ne peut accéder aisément qu'à leur premier élément. Une technique courante consiste à transformer, le temps du tri, la liste en tableau.

3.1. Écrire une fonction qui transforme une liste de chaînes en un tableau de chaînes, de prototype

```
char** array_from_list(list l).
```

Attention, cette fois les chaînes ne doivent pas être recopiées.

3.2. Représenter l'état de la mémoire après les instructions suivantes

```
list l = cons("Mars", cons("Jupiter", cons("Saturne",NULL)));
char** t = array_from_list(l);
```

3.3. Écrivez un tri par insertion pour le tableau produit à la question 3.1 ; le prototype attendu est `sort(char** t, int n)` où `n` correspond à la longueur du tableau `t`. Quelle est la complexité de ce tri en nombre de comparaisons de chaînes (appels à `strcmp`) ?

3.4. Enfin, écrivez une fonction recréant une liste à patir d'une tableau de chaînes :

```
list list_from_array(char** t, int n)
```

où `n` est la taille du tableau `t`. Cette fois-ci les chaînes devront être copiées lors de la création de la liste (utilisez l'opération `cons` à cette effet).

4 Élimination des doublons

On suppose que l'on dispose de listes triées. Par conséquent, si elles contiennent plusieurs fois le même élément, ses occurrences se suivent dans la liste.

4.1. Implémenter l'opération `myst` dont la définition abstraite est

$$\begin{cases} \text{myst} : \mathbb{L} \to \{V, F\} & \\ \text{myst}(h_1 :: h_2 :: t) = V & \text{si } h_1 = h_2 \\ \text{myst}(l) = F & \text{dans tous les autres cas.} \end{cases}$$

où $\mathbb{L}$ désigne l'ensemble des listes. Donnez-lui un nom plus approprié.

4.2. Implémenter l'opération `list chop(list l)` qui supprime le premier élément d'une liste, libère la mémoire correspondante et retourne la queue de la liste. La fonction retourne la liste vide si l'argument est la liste vide.

4.3. À l'aide des deux précédentes questions, proposez une fonction `list supprime_doublons(list l)` qui attend en argument une liste triée et supprime les éléments de la liste en plusieurs exemplaires, de sorte qu'il n'en reste plus qu'une seule occurrence. Par exemple, la liste

$$"aa" :: "ab" :: "ab" :: "abc" :: "abd" :: "abd" :: "abd" :: "zz" :: \emptyset$$

devient

$$"aa" :: "ab" :: "abc" :: "abd" :: "zz" :: \emptyset$$

.

Conclusion du problème

Écrire une fonction `list uniq(list l)` qui à une liste quelconque associe une nouvelle liste en mémoire contenant les mêmes chaînes que `l`, mais débarassée de ses doublons, c'est-à-dire contenant chaque chaîne de caractères au plus une fois. Donner sa complexité au pire en nombre d'opérations sur les chaînes (celles vues à la section 1). Commentaire ?