# Devoir surveillé IM41 : Programmation en C

Durée 2h

Jeudi 17 Mars 2011

Aucun document papier ou électronique n'est autorisé. Le sujet est volontairement long : lisez bien tout le devoir, ne bloquez pas sur une question et n'hésitez pas à vous servir des résultats non trouvés pour avancer. La difficulté des questions au sein d'un problème est croissante, et les trois problèmes sont de même difficulté.

## 1 Files d'attente

1.1. Donner la définition mathématique des listes, et quelques exemples de listes. Dans tout le problème, on travaillera sur des listes de réels, les réels étant représentés en C par des `float`.

1.2. Donner et expliquer la représentation en langage C des listes vue en cours (appelée représentation par listes simplement chaînées).

1.3. Écrire le constructeur correspondant, de prototype `list cons(float h, list t)`.

1.4. On s'intéresse à l'opération `sum`, qui à une liste associe la somme de ses éléments (rappel : les éléments sont supposés réels).

(a) Définissez `sum` comme dans un type abstrait.

(b) Proposez une implémentation en style récursif.

(c) Proposez une implémentation en style itératif.

1.5. Implémentez l'opération `get` qui renvoie le premier élément d'une liste (attention à la précondition).

1.6. Implémentez l'opération `pop`, qui supprime le premier élément d'une liste et renvoie sa queue.

1.7. On souhaite maintenant disposer d'une opération `push`, qui ajoute un élément à la fin d'une liste. Le prototype attendu est `list push(float e, list l)`.

(a) Proposez une implémentation en style récursif.

(b) Proposez une implémentation en style itératif.

1.8. On se donne le programme suivant

```
list l = push(2,NULL);
l = push(3,l);
printf("%d", get(l));
pop(l);
l = push(1,l);
printf("%d", get(l));
pop(l);
printf("%d", get(l));
pop(l);
```

Représentez l'état de la mémoire après l'exécution de chaque ligne, et déterminez ce que le programme affiche.

1.9. Le type abstrait constitué des opérations `get`, `push` et `pop` est appelé *file d'attente* dans la littérature. Expliquez pourquoi. Si chaque élément dans la liste correspond à une durée, que calcule la fonction `sum` ?

## 2 Représentation mémoire des matrices

On souhaite représenter des matrices réelles avec le type `matrix` suivant :

```
struct matrix_t {
  int m; // nombre de lignes
  int n; // nombre de colonnes
  float** coef;
};
typedef matrix_t* matrix;
```

2.1. Proposez un constructeur `matrix create_matrix(int m, int n)` qui initialise les coefficients à 0. La solution devra produire $m + 2$ allocations mémoire pour une matrice à $m$ lignes.

2.2. Montrer sur un exemple qu'avec cette représentation, les données de la matrice sont éparpillées en mémoire (avec une matrice de dimension $3 \times 4$).

2.3. Les processeurs actuels savent optimiser très efficacement l'exécution d'un code lorsque les données traîtées successivement sont proches en mémoire. Pour en tenir compte, proposez une modification de votre constructeur telle que les coefficients de la matrice produite se trouvent dans une même région mémoire. Votre solution devra produire 3 allocations mémoire.

2.4. Illustrer l'impact de votre modification en donnant la nouvelle représentation mémoire de l'exemple donné en question 2.2

2.5. On rappelle qu'une matrice symétrique est une matrice $(a_{ij})$ telle que $a_{ij} = a_{ji}$. Dans ce cas particulier, on peut optimiser la représentation pour qu'elle prenne moins de place en mémoire. Proposez une amélioration du constructeur dans ce sens. En supposant que les pointeurs, ainsi que les variables de type `int` et `float` occupent 4 octets, déterminer en fonction de $m$ et $n$ (resp. nombre de lignes et de colonnes de la matrice) la taille respective en octets des deux représentations.

2.6. Écrire une fonction `float get(int i, int j)` permettant d'accéder à un élément de la matrice, dans le cas du codage des matrices symétriques (attention aux préconditions !).

# 3 Tri à bulles sur les listes

On cherche dans cet exercice à adapter aux listes le tri à bulles vu en cours (et au passage, améliorer sa complexité). Pour rappel, le tri à bulles consiste à parcourir la séquence à trier en regardant deux éléments successifs, et en les permutant s'ils ne sont pas dans le bon ordre. On part de la représentation des listes vue en cours et qui est utilisée dans le problème 1.

3.1. Écrire une fonction `int has_inversion(list l)` qui détecte si une liste a au moins deux éléments, et telles que ses deux premiers éléments $x$ et $y$ vérifient $x > y$.

3.2. Écrire une fonction `list swap_head(list l)` qui permute les deux premiers éléments d'une liste, s'ils existent (et échoue sinon). Exemple : si on donne en argument une liste représentant $1 :: 2 :: 3 :: \emptyset$ à `swap_head`, elle retourne la liste représentant $2 :: 1 :: 3 :: \emptyset$.

3.3. Soit le programme suivant

```
list l = cons(3,cons(2,cons(1,NULL)));
l = swap_head(l);
```

Représentez l'état de la mémoire avant et après appel de `swap_head`.

3.4. On définit formellement l'opération `ascension` par la récurrence suivante :

$$\begin{cases} \text{ascension}(\emptyset) = \emptyset \\ \text{ascension}(h :: \emptyset) = h :: \emptyset \\ \text{ascension}(h :: h' :: t) = h :: h' :: t & \text{si } h \leq h' \\ \text{ascension}(h :: h' :: t) = h' :: \mathit{ascension}(h :: t) & \text{si } h > h' \end{cases}$$

Intuitivement la tête remonte dans la liste comme une bulle dans un liquide, et stoppe quand elle rencontre un élément plus grand qu'elle. Implémenter cette opération en utilisant (si vous le souhaitez) les fonctions des questions précédentes, et en veillant bien à la gestion de la mémoire.

3.5. Quelle est la complexité de l'opération `ascension` ?

3.6. Implémenter le tri à bulles, en vous aidant de la récurrence suivante :

$$\begin{cases} \text{sort}(\emptyset) = \emptyset \\ \text{sort}(h :: t) = \mathit{ascension}(h :: \mathit{sort}(t)) \end{cases}$$

3.7. Déterminer et justifier la complexité de ce tri.