TD12

Langage C (LC4)

semaine du 3 mai 2010

1 Listes

On commence par travailler avec le type de listes suivant :

```
struct Liste {
  int valeur ;
  struct Liste * suivant ;
};
```

Question 1. Écrire une fonction
`void decoupe (struct Liste * l, struct Liste ** l1, struct Liste ** l2)`
qui prend une l iste l en argument et la découpe en deux listes :
– la liste formée du premier maillon de l, puis du troisième, puis du cinquième, . . .
– la liste formée du deuxième maillon de l, puis du quatrième, puis du sixième, . . . Le pointeur vers le premier maillon de la première (respectivement seconde) liste devra être écrit dans *l1 (respectivement *l2).

Question 2. Écrire une fonction `void filtre(struct Liste * l)` qui efface le deuxième maillon de la liste, puis le quatrième, puis le sixième, . . .

2 Arbres

On travaille avec le type :

```
struct arbre {
  int valeur ;
  struct arbre * gauche ;
  struct arbre * droite ;
};
```

Question 3. Écrire une fonction qui teste si deux arbres ont le même squelette.

Question 4. Écrire une fonction qui teste si un arbre f est un préfixe d'un arbre g, c'est-à-dire si l'on peut obtenir f à partir de g en remplaçant certains sous-arbres par des arbres vides.

Question 5. Comment définir un type d'arbre où chaque nœud peut avoir un nombre variable de fils ? Écrire, pour ce type, une fonction calculant l'arité maximale d'un nœud.

3 Pointeurs

Question 6. Écrire une fonction `char* chaine_binaire (char zero, char un, int n)` qui construit une chaîne de caractère correspondant à la représentation binaire de n. On utilisera $zero$ pour représenter 0 et un pour représenter 1. On suppose que n est stocké sur 32 bits.

Question 7. Écrire une fonction `int** tri (int t, int* tab)`
qui calcule un tableau de pointeur de taille t à partir de tab, lui aussi de taille t. Dans la case i du tableau, on trouve un pointeur vers le plus petit élément de tab qui est strictement plus grand que $tab[i]$. Si cet élément n'existe pas alors cette case vaut NULL.

Question 8. Qu'affichera le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  double tab[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 };
  double *d = tab;
  double *f = d + sizeof tab / sizeof(double);
  int  i = 4;
  double *p = &tab[i];
  double x;
  printf("%3.1f\n",*(d+2));
  printf("%3.1f\n",*d + 2);
  printf("%d\n",p-d);
  printf("%3.1f\n",*(p-2)+3);
  x = *p++;
  printf("f-p = %d\n", f-p);
  printf("x = %3.1f, *p= %3.1f\n", x, *p);
  return EXIT_SUCCESS;
}
```

Question 9. Écrire la fonction
`char * coder(const char *text, const char *from, const char *to)` qui sera utilisée pour coder la chaîne `text`. La fonction retournera une nouvelle chaîne de caractères de même longueur que la chaîne `text` (il faut allouer la mémoire pour la nouvelle chaîne, la chaîne `text` ne doit pas être modifiée). Les chaînes `from` et `to` définissent les correspondances entre les lettres dans le codage. Par exemple si `from="abacd"` et `to="xyzx"` on aura la correspondance suivante :
a → x,
b → y,
c → x,
on a donc la première lettre de `from` qui correspond à la première lettre de `to`, la deuxième lettre de `from` qui correspond à la deuxième lettre de `to`, etc.
Vous noterez que seulement la première occurrence d'une lettre dans `from` compte, en effet, la deuxième occurrence de a dans `from` ne donne pas lieu à une correspondance.
De plus, la lettre d dans `from` n'a pas de correspondance parce que la chaîne `to` est plus courte que la chaine `from`.

La chaîne codée à retourner sera construite de la façon suivante : chaque caractère qui n'a pas de correspondance sera recopié sans changement, si un caractère dans `from` a une correspondance dans `to` alors c'est le caractère codé qui sera recopié dans la chaîne codée. Dans notre exemple les caractères `a,b,c` seront remplacés respectivement par `x,y,x` tandis que tous les autres caractères seront recopiés sans modification.