TD 8 : Pointeurs et structures de données

Programmation en C (LC4)

Semaine du 26 Mars 2012

1 Arbres d'arité variable

On veut manipuler des arbres contenant des entiers, dont les nœuds possèdent potentiellement plusieurs fils. Mais on ne sais pas précisément combien de fils a un nœud, à l'avance (contrairement au cas d'un arbre binaire, par exemple, où tout nœud aura deux fils ou aucun). On représente une telle structure de donnée en C par le type suivant :

```
typedef struct nnoeud {
    int arite;
    nnoeud **fils;
  } *narbre;
```

où le pointeur `fils` représente un tableau de pointeurs de `struct nnoeud`, qui contient tous les fils du nœud représenté par cette structure. La taille de ce tableau est indiquée par l'entier `arite`, qui représente l'arité du nœud. Un tel arbre est appelé *arbre n-aire*, car chaque nœud a une arité n qui peut varier.

Question 1 Quel est le meilleur moyen de représenter un arbre n-aire vide, si on utilise cette structure de donnée ? Comment va-t-on représenter une feuille dans un tel arbre ?

Question 2 Ecrire une fonction `narbre narbre_vide()`, qui crée un nouvel arbre n-aire vide et le renvoie. Ecrire également une fonction `narbre narbre_cons(int k, narbre *l)` qui prend en argument un entier k et un tableau d'arbres n-aires de taille k, et qui construit un arbre contenant tous ces arbres comme fils, puis le renvoie.

Question 3 Ecrire une fonction `void afficher(narbre a, int indent)` qui peut afficher le contenu d'un arbre en décalant l'affichage du contenu des fils d'un nœud de deux espaces par rapport au contenu du nœud (comme cela était fait dans le TP 7). Tester les fonctions de construction sur quelques exemples et afficher les arbres créés.

Un arbre n-aire est une structure assez générale, mais qui est plus complexe à manipuler qu'un arbre binaire, par exemple (ou une liste, qui est un arbre unaire). On peut vouloir se ramener à ces cas plus simples, en modifiant l'arbre n-aire.

Question 4 Ecrire une fonction `void binairiser(narbre a)` qui prend en argument un arbre n-aire quelconque et le transforme en un arbre binaire. Pour cela, on remplacement un nœud n à k fils par un sous-arbre à k feuilles, où l'on attachera aux feuilles les fils du nœud n.

Question 5 Ecrire une fonction `void unairiser(narbre a)` qui prend en argument un arbre n-aire quelconque et le transforme en une liste (c'est-à-dire en un arbre dont tous les nœuds, sauf la feuille, sont d'arité 1). La liste produite devra être telle qu'un nœud à gauche d'un autre dans l'arbre se retrouve à gauche de ce nœud dans la liste.

Notez que puisque la structure d'un arbre est récursive, toutes ces fonctions de manipulation des arbres seront récursives. Pour les deux dernières, qui sont plus complexe, on pourra faire appel à des fonctions auxiliaires, notamment pour la construction du résultat.

2 Tableaux à plusieurs dimensions

On considère des rectangles de pixels de taille $n \times k$, pour n et k fixés, où un pixel est représenté par un caractère, soit vide `'_'` soit plein `'@'`. On souhaite manipuler ces rectangles, et notament les composer en les accolant les uns aux autres. On représente un rectangle en utilisant le type (on définit arbitrairement des valeurs pour n et k) :

```
#define N 5
#define K 5
typedef char[N][K] rect;
```

Question 6 Ecrire une fonction qui compose 4 rectangles en renvoyant un grand rectangle du type `grandrect`, défini comme `char[2*N][2*4]`, tel que ce rectangle contienne les 4 rectangles accolés les uns aux autres. C'est donc un grand rectangle de deux rectangles de haut, et deux rectangles de large.

Question 7 On souhaite composer un nombre de rectangles indéfini à l'avance. Ecrire pour cela une fonction `composer(int a, int b, rect *r)`, qui compose $a \times b$ rectangles en un grand rectangle de a rectangles de haut et b rectangles de large. Peut-on encore utiliser un type de tableau à double dimension pour la valeur de retour de cette fonction ?

Notez que de manière générale, on aurait pu ne pas utiliser de tableau à deux dimensions, mais un tableau de tableau "dynamique", c'est-à-dire un tableau donc chaque case contient elle-même un tableau, via un pointeur.