TP6 – Listes chaînées

Projet de programmation M1

25 Novembre 2014

Dans ce TP, on se propose d'implémenter une structure de données en C pour représenter les listes. Contrairement aux tableaux, les listes n'ont pas une taille fixée. Il va donc falloir utiliser intelligemment l'allocation dynamique avec `malloc` et bien libérer la mémoire qui n'est plus utilisée avec `free`.

Un élément d'une liste chaînée est composé d'une valeur et d'un pointeur vers l'élément suivant dans la liste. Si on est à la fin de la liste, on pointera vers `NULL`, le pointeur vide. On représente les éléments d'une liste chaînée par la structure `node` suivante :

```
struct node {
  int value;
  struct node *next;
};
```

On mettra la valeur de `next` à `NULL` si le nœud n'a pas de successeur.

Enfin, une liste sera simplement un pointeur vers son premier élément, aussi appelé la *tête* de la liste. Une liste vide sera tout simplement représentée par le pointeur `NULL`. En vue de futures améliorations (on pourrait aussi se rappeler de la fin de la liste et de sa taille par exemple), nous allons mettre tout ça dans une structure :

```
struct list {
  struct node *head;
};
```

Nous rappelons que si `n` est de type `node`, on peut accéder au champ "value" de `n` en écrivant `n.value`. Si `n` est de type `node *` (un pointeur vers un `node`) et qu'on veut accéder au champ `value` du `node` pointé par `n`, on doit écrire `(*n).value`. On peut aussi dans ce cas écrire de manière équivalent `n->value`.

Nous rappelons que `malloc(sizeof(node))` réserve suffisamment de mémoire pour stocker un objet de type node et renvoie un pointeur contenant l'adresse mémoire qui vient d'être réservée. Par exemple, si `n` est le dernier `node` d'une liste chaînée (donc que `node.next == NULL`, on peut ajouter un élément en faisant :

```
node *p = malloc(sizeof(node));
p->value = 42;
n.next = p;
```

Enfin, si `p` est un pointeur vers un espace de la mémoire qui a été réservé par `malloc` mais qu'on n'en a plus besoin, on peut "libérer" la mémoire pour une future utilisation avec la commande `free(p)`. Il est important de libérer la mémoire quand elle n'est plus utilisée.

À titre d'exemple, nous fournissons la fonction `print_list(struct list l)` qui affiche une liste à l'écran :

```
void print_list(struct list l) {
  node *i = l.head;
  while(i != NULL) {
    printf("%d,", i->value);
    i = i->next;
  }
}
```

Exercice 1. Écrire une fonction `liste empty()` qui renvoie la liste vide. Vous l'utiliserez pour initialiser vos listes.

Exercice 2. Assurez-vous d'abord de bien comprendre la fonction `print_list`. Implémentez ensuite deux fonctions `void insert_head(struct list *l, int v)` et `void insert_tail(struct list *l, int v)` qui insère la valeur `v` à la tête de la liste pointée par `l` et à la fin de la liste `l` respectivement.

Exercice 3. Implémentez une fonction `struct node* search(struct list l, int v)` qui cherche la valeur `v` dans la liste `l`. Si la valeur est trouvée, votre fonction renvoie un pointeur vers le nœud où elle se trouve. Sinon, elle renvoie le pointeur `NULL`.

Exercice 4. Implémentez une fonction `remove_value(struct list *l, int v)` qui enlève toutes les occurences de la valeur `v` dans la liste `l`. N'oubliez pas de libérer la mémoire désormais inutilisée.

Exercice 5. Écrivez une fonction `void reverse(struct list *l)` qui inverse l'ordre d'une liste.

Exercice 6. Écrivez une fonction `struct list concat(struct list l1, struct list l2)` qui renvoie une liste contenant la concaténation de `l1` et de `l2`. Attention, ni `l1` ni `l2` ne doivent être modifiés par cette opération ; on veut copier toutes les données dans une nouvelle liste `l`.

Exercice 7. Implémentez votre fonction de tri préférée pour les listes.