TD/TP 11 — corrigés partiels

1 Structure de données

Voici plusieurs tentatives de définition du type des listes. Pour chacune, indiquer si elle est correcte ou non. Comparer les définitions correctes. Comment définir une liste vide ?

```
typedef struct {
    int x;
    liste1 q;
} liste1;
```

Rejetée par le compilateur : le nom de type `liste1` n'est pas encore disponible là où il est utilisé.

```
struct li2 {
    int x;
    liste2 q;
};
typedef struct li2 liste2;
```

Rejetée par le compilateur : le nom de type `liste2` n'est pas encore disponible là où il est utilisé.

```
struct li3 {
    int x;
    liste3 q;
};
typedef struct li3 *liste3;
```

Rejetée par le compilateur : le nom de type `liste1` n'est pas encore disponible là où il est utilisé.

```
struct li4 {
    int x;
    struct li4 q;
};
typedef struct li4 *liste4;
```

Rejetée par le compilateur : on ne peut pas mettre une `struct li4` comme champ d'elle-même ! (Ça donnerait un type de taille infinie...)

```
struct li5 {
    int x;
    struct li5 *q;
};
typedef struct li5 liste5;
```

Le type `liste5` est parfaitement correct, mais ce n'est pas le type des listes (c'est le type des listes non vides).

```
struct li6 {
    int x;
    struct li6 *q;
};
typedef struct li6 *liste6;
```

Correct. La liste vide est représentée par `NULL`.

```
struct li7 {
    int x;
    struct li7 *q;
};
typedef struct li7 **liste7;
```

Correct. Ici toute liste est représentée par un pointeur qui ne change pas même si on modifie la liste (ce qui n'est pas possible avec `liste6`, par exemple si on ajoute un élément à une liste vide).

```
struct li8;
typedef struct li liste8;
struct li8 {
    int x;
    liste8 q;
};
```

Syntaxiquement correct, mais rejeté pour la même raison que `liste4`. La première ligne *déclare* une structure appelée `li8`, et la troisième commence une *définition* de cette structure.

```
struct li9;
typedef struct li *liste9;
struct li9 {
    int x;
    liste9 q;
};
```

Correct ; cette définition est équivalente à `liste6`.

Remarque On ne parle ici que de listes d'entiers. Le langage C n'a pas directement de moyen de définir de listes génériques, c'est-à-dire de listes dans lesquelles on pourrait stocker n'importe quel type de données. Cependant on peut se débrouiller, au pire en ajoutant un niveau d'indirection supplémentaire, en faisant des listes de `void*`. Après, si on veut par exemple une liste d'entiers, on stocke l'adresse de ces entiers au lieu des entiers eux-mêmes dans la liste.

On place la définition du type liste et les prototypes des fonctions dans un en-tête `liste.h`.

```
#ifndef LISTE_H_
#define LISTE_H_
struct liste {
    int x;
    struct liste *q;
};
typedef struct liste *liste;
int est_vide(liste l);
int tete(liste l);
liste queue(liste l);
liste cons(int x, liste l);
/* etc */
#endif
```

Voici les fonctions de base, avec lesquelles on peut presque tout faire.

```
#include <stdlib.h>
#include "liste.h"
int est_vide(liste l) {
    return l == NULL;
}
int tete(liste l) { /* tête = premier élément */
    return l->x; /* si la liste est vide, on déréférence un pointeur nul */
}
liste queue(liste l) { /* queue = tout sauf la tête */
    return l->q; /* si la liste est vide, on déréférence un pointeur nul */
}
liste cons(int x, liste q) { /* construire une nouvelle cellule */
    struct liste *l = malloc(sizeof(*l));
    if (l == NULL) abort();
    l->x = x; l->q = q;
    return l;
}
```

Noter que `cons` utilise `malloc`. Si on définissait une variable locale de type `struct liste`, la zone mémoire utilisée pour la stocker serait récupérée lorsque la fonction `cons` se termine.

2 Parcours de base

(a) Écrire une fonction qui détermine la longueur d'une liste.
(b) Écrire une fonction qui renvoie le i-ème élément d'une liste.

La boucle `for` de la fonction `longueur` illustre le parcours d'une liste, qui ressemble au parcours habituel d'un tableau : on commence au début (tête/indice 0), on passe au suivant après traitement (queue/incrémentation de l'indice), et on s'arrete à la fin (vide/dernier indice).

Dans `element`, j'ai utilisé une boucle `while`, pour changer.

```
#include <stdlib.h>
#include "liste.h"
size_t longueur(liste l0) {
    size_t n = 0;
    liste l;
    for (l = l0; !est_vide(l); l = queue(l)) ++n;
    return n;
}
int element(liste l, size_t i) {
```

```
    while (!est_vide(l) && i > 0) {--i; l = queue(l);}
    return tete(l);
}
```

3 Ajout

(a) Écrire une fonction qui insère un entier dans une liste à une position donnée. Par exemple, `insere(&l,0,42)` *insère 42 au début de la liste, tandis que* `insere(&l,1,42)` *insère 42 après le premier élément.*

Pourquoi un argument de type `liste*` *(c'est-à-dire* `struct li **`*) et non* `liste` *?*

(b) Écrire une fonction qui ajoute un entier à la fin d'une liste.

```
#include <stdlib.h>
#include "liste.h"
void insere(liste *p, size_t i, int x) {
    liste c;
    for (; i > 0; i--) p = &((*p)->q);
    c = cons(x, *p);
    *p = c;
}
void ajoute(liste *p, int x) {
    liste c = cons(x, NULL);
    while (*p != NULL) p = &((*p)->q);
    *p = c;
}
```

Les fonctions qui modifient des listes prennent comme arguments des pointeur vers des listes, et non des listes, au cas où il faudrait toucher à la première cellule. Par exemple, si on ajoute un élément à la liste vide, il faut que l'appelant puisse accéder à la cellule nouvellement construite, donc il faut qu'il dispose de plus qu'un pointeur nul.

Dans `insere`, si la liste est trop courte, on déréférence un pointeur nul.

Les fonctions `ajoute` et `insere` font un peu de gymnastique avec les pointeurs. La variable `p` contient toujours l'adresse d'un pointeur sur une cellule, et la boucle sert à placer dans `p` soit l'adresse du champ `q` de la cellule après laquelle on veut faire l'insertion (ou l'adresse du pointeur sur le début de la liste si on insère au tout début). Lorsqu'on modifie ensuite `*p`, c'est ce champ `q` qu'on modifie.

(c) La fonction `ajoute` *prend un temps proportionnel à la longueur de la liste. Comment modifier la structure de données retenue pour les listes, si on veut rendre l'opération d'ajout plus efficace ?*

En plus de l'adresse de la première cellule (ou `NULL` pour une liste vide), on peut garder accessible l'adresse de la dernière cellule (ou `NULL` pour une liste vide). Ceci permet d'ajouter rapidement un élément à n'importe quel bout de la liste, mais de supprimer rapidement seulement le premier élément. Si on veut pouvoir ajouter ou supprimer rapidement un élément au deux bouts, on peut utiliser une *liste doublement chaînée*, dans laquelle chaque cellule contient non seulement un pointeur vers la cellule suivante mais aussi un pointeur vers la cellule précédente.

4 Quelques fonctions classiques

Vous trouverez dans le fichier `liste-classiques.c` quelques fonctions classiques sur les listes.