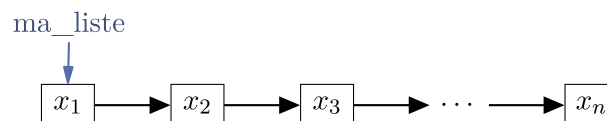
Algorithmique

TD n°5 : Tableaux et listes

Exercice 1 – Fonctions simples sur les listes simplement chaînées

On considère la structure ci-dessous pour représenter des listes simplement chaînées :




```
// la structure des boites de la liste chaînée
typedef struct list_box_t {
  int value;
  struct list_box_t* next;
} list_box_t;

// la structure définissant une liste
// ⟹ permet d'avoir un pointeur sur un list_t et de rajouter
// un élément en tête de liste sans modifier le pointeur sur le list_t
typedef struct {
  list_box_t* tete;
} list_t;
```

Q 1.1 Écrivez une fonction `length` qui, étant donné une liste `ma_liste` de type `list_t`, renvoie la longueur de celle-ci.

Q 1.2 Écrivez une fonction `get` qui, étant donné une liste `ma_liste` de type `list_t` et un entier `i` renvoie le `i`ème élément de la liste (le premier élément correspondant à `i=0`). Si la liste comporte moins de `i+1` éléments, la fonction lèvera une exception.

Q 1.3 Écrivez une fonction `rev` qui, étant donné une liste de type `list_t`, renvoie une nouvelle liste de même type, dont les éléments sont dans l'ordre inverse.

Q 1.4 Écrivez une fonction `append` qui, étant donné deux listes `L1` et `L2` de type `list_t`, concatène à `L1` la liste `L2`. Si l'on supprime la liste `L2`, cela ne doit avoir aucun impact sur `L1`.

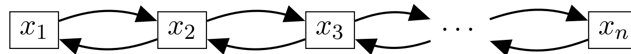
Q 1.5 Écrivez une fonction `iter` qui, étant donné une liste `ma_liste` de type `list_t` et un pointeur de fonction `void (*fonction) (int)`, applique à chaque élément de la liste la fonction `fonction`. Le prototype de la fonction est donc :

```
void iter(list_t ma_liste, void (*fonction) (int));
```

Q 1.6 Écrivez une fonction `map` qui, étant donné une liste `ma_liste` de type `list_t` et un pointeur de fonction `int (*f) (int)`, renvoie une nouvelle liste constituée des valeurs obtenues par l'application de `f` aux éléments de `ma_liste`. Par exemple, si `ma_liste =` ⟨`1,2,3,4,5`⟩, la liste renvoyée par la fonction `map` est ⟨`f(1),f(2),f(3),f(4),f(5)`⟩.

Exercice 2 – Listes doublement chaînées

Une liste doublement chaînée peut être représentée par un schéma comme celui ci-dessous :



Q 2.1 Proposez une structure C pour représenter de telles listes.

Q 2.2 Écrivez une fonction `create` qui renvoie une nouvelle liste vide.

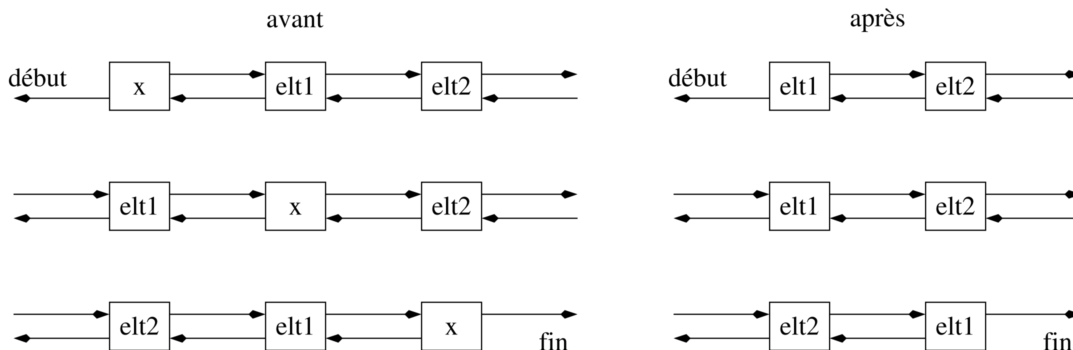
Q 2.3 Écrivez une fonction `front` qui, étant donné une liste doublement chaînée, renvoie le premier élément de la liste s'il existe ou bien lève une exception dans le cas contraire.

Q 2.4 Écrivez deux fonctions `prev` et `next` qui, étant donné une boite de la liste, renvoie la boite précédente et la boite suivante respectivement.

Q 2.5 Écrivez une fonction `insert` qui, étant donné une liste doublement chaînée et un élément `elt`, rajoute celui-ci en premier élément de la liste.

Q 2.6 Écrivez une fonction `remove` qui, étant donné une liste doublement chaînée, supprime le premier élément de la liste s'il existe ou bien lève une exception dans le cas contraire.

Q 2.7 Écrivez une fonction `supress` qui, étant donné une liste doublement chaînée et un élément `x`, supprime la première occurrence de celui-ci dans la liste. Si l'élément n'existe pas, la fonction lève une exception. Idée : il suffit de rechaîner la liste comme indiqué dans la figure ci-dessous :



Exercice 3 – Les listes circulaires

Nous allons illustrer dans cet exercice l'utilisation de listes doublement chaînées circulaires en simulant l'envoi de messages par des ordinateurs reliés dans un réseau de type *token ring*. Dans un tel réseau les ordinateurs (tout au moins les MSAU (multistation access units)) sont reliés les uns aux autres de manière à former un anneau (cf. figure 1). Un jeton (le token) circule dans le réseau. Seul un ordinateur possédant le jeton peut envoyer des messages aux autres. Si un ordinateur reçoit le jeton mais n'a rien à transmettre, il passe le jeton à l'ordinateur suivant. Quand un ordinateur veut envoyer un message

à un autre ordinateur, il attend le jeton, le saisit, transmet son information et, lorsque cette dernière est arrivée à destination, il passe le jeton à la station suivante, etc.

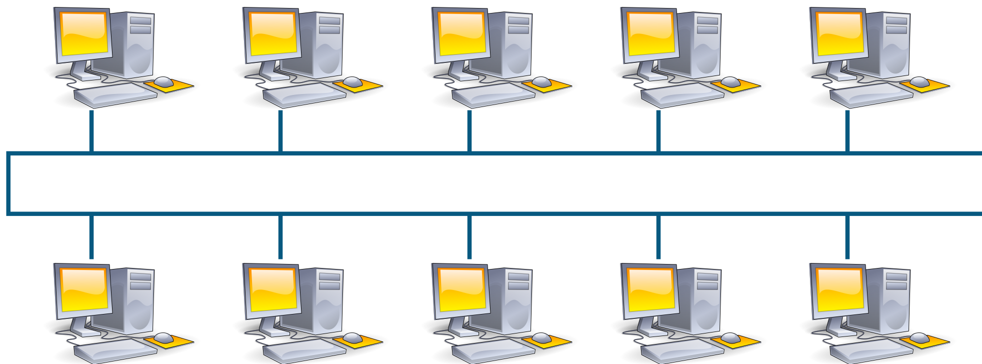


FIGURE 1 – Un réseau en token ring.

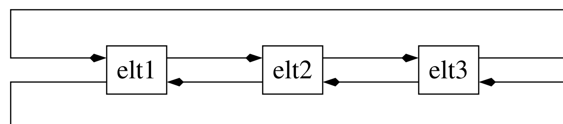
On définit le type `machine_t` pour symboliser l'envoi de message par un ordinateur : un ordinateur `mac` va envoyer un message via une fonction `message`. La probabilité qu'à un instant t `mac` veuille transmettre un message est fournit par le champ `proba`.

```
typedef struct {
  char mac[20]; // nom de la machine
  int  proba;   // 100 * probabilité d'avoir un message à transmettre
  void (*message) (char mach[20], int prob); // fonction qui transmet le message
} machine_t;
```

Q 3.1 Définissez une fonction `message` qui prend en argument le nom d'une machine X et une probabilité p ($\times$ 100), et qui affiche à l'écran le message « émission de X » avec une probabilité de p% et ne fait rien avec une probabilité de $(100 - p)$%. Utilisez la fonction `rand()` pour simuler la probabilité.

Q 3.2 Créez trois ordinateurs de type `machine_t` ayant pour fonction d'envoi de messages la fonction ci-dessus, et ayant des probabilités différentes.

Q 3.3 Une liste doublement chaînée circulaire est une liste dont tous les éléments ont un prédécesseur et un successeur, cf. la figure ci-dessous :



Q 3.4 Proposez une structure C pour représenter des listes circulaires doublement chaînées dont les éléments sont de type `machine_t`.

Q 3.5 Écrivez une fonction `circular_empty` qui renvoie une liste circulaire vide.

Q 3.6 Écrivez une fonction `circular_insert` qui, étant donné une liste circulaire `liste` et une machine `e` rajoute l'élément `e` à `liste`. Deux cas sont à envisager :

1. `liste` est vide. Dans ce cas il faut créer une liste ne comportant qu'une seule case contenant `e` et qui est elle-même son propre prédécesseur et successeur.

2. `liste` n'est pas vide. Dans ce cas, vous insérerez `e` juste après le premier élément de `liste`. Même si la liste est circulaire, d'après la définition de votre type de liste circulaire, une telle liste non vide possède un « premier » élément.

Q 3.7 Créez une liste circulaire qui contient les 3 machines que vous aviez créées précédemment.

Q 3.8 Écrivez une fonction `envois`, prenant en argument un entier `n` ainsi qu'une liste circulaire `liste` de machines, et telle que `envois(n,liste)` simule `n` transmissions de jetons dans le token ring constitué des machines de `liste`. Chaque fois qu'une machine possède le jeton, elle lance sa fonction `message` avec son nom de machine et sa probabilité.

Exercice 4 – Polymorphisme

Comme vous avez pu le constater, si vous créez une liste contenant des entiers, celle-ci ne peut être utilisée pour stocker des nombres réels ou des chaînes de caractères. Pour avoir à disposition une liste de réels, il faut dupliquer tout le code des listes d'entiers et remplacer `int` par `float`. Une alternative consiste à ne créer qu'un seul code et à faire en sorte qu'il s'applique pour tous les types de données, c'est ce que l'on appelle le `polymorphisme`. Le C++ supporte le polymorphisme avec la notion de `template`. Malheureusement, vous ne connaissez que le C qui, lui, ne connaît pas cette notion. Pour la simuler, je vous propose d'utiliser une astuce exploitant le préprocesseur du C. Dans ce dernier, on peut définir des constantes :

```
#define MA_CONSTANTE 42
```

Ainsi, quand le préprocesseur rencontre la chaîne de carcactères « `MA_CONSTANTE` » dans votre code C, il remplace cette chaîne par la chaîne « 42 ». Ensuite, c'est le nouveau code ainsi créé qui est transmis au compilateur. Avec le préprocesseur, on peut également créer des macros :

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Ainsi, chaque fois que le préprocesseur rencontre dans votre code `MAX` suivi de 2 arguments, par exemple `MAX(4,ma_var)`, il remplace cette chaîne de caractères par la chaîne « `((x) > (y) ? (x) : (y))` » en substituant les `x` et `y` par les paramètres passés au `MAX`. Donc « `MAX(4,ma_var)` » est remplacé par la chaîne de caractères « `((4) > (ma_var) ? (4) : (ma_var))` ».

Enfin, le préprocesseur possède un opérateur de concaténation `##`. Ainsi, si l'on définit une macro :

```
#define F(x,y) x ## y
```

celle-ci remplace toutes les occurrences de « `F(x,y)` » par la concaténation de `x` et de `y`. Par exemple, dans votre code, « `F(toto,titi)` » sera substitué par « `tototiti` ».

En exploitant ces informations, on peut écrire les deux fichiers `ma_struct_float.h` et `ma_struct.h` suivants :

ma_struct_float.h
```
#ifndef MA_STRUCT_FLOAT_H
#define MA_STRUCT_FLOAT_H

#define MYTYPE float
#define MYFUNC(func) func ## _ ## float
#define MYSTRUCT(struct) struct ## _ ## float ## _t

#include "ma_struct.h"

#undef MYTYPE
#undef MYFUNC
#undef MYSTRUCT

#endif /* MA_STRUCT_FLOAT_H */
```

ma_struct.h
```
// definition de la structure
typedef struct MYSTRUCT(ma_struct) {
  MYTYPE value;
  struct MYSTRUCT(ma_struct)* next;
} MYSTRUCT(ma_struct);


// les fonctions opérant sur la structure
MYTYPE MYFUNC(create) () {
  MYTYPE x;
  return x;
}

....... autres fonctions
```

Tout le code correspondant à la structure et à ses fonctions est écrit dans le fichier `ma_struct.h`. Dans le fichier `ma_struct_float.h`, qui est entièrement défini ci-dessus, on réalise juste l'inclusion du fichier `ma_struct.h` et, à l'aide des macros du préprocesseur, on change les noms de la structure et de ses fonctions pour les suffixer avec `float`. Dans le code générique, cela revient à remplacer le nom de la structure (`ma_struct`) par `MYSTRUCT(ma_struct)`, le nom du type des éléments contenus dans `ma_struct` par `MYTYPE(ma_struct)` et les noms des fonctions (`create`) par `MYFUNC(create)`.

Q 4.1 Copiez-collez les codes ci-dessus et exécutez `gcc -E ma_struct_float.h`. Cela vous affichera le code produit après les substitutions par le préprocesseur.

Q 4.2 Reprenez les codes de vos listes et rendez-les génériques. Cela vous servira, notamment, dans les TD suivants.