Analyse d'un type Python (`list`) et ré-implémentation en `C`

L'objectif de cette séance est d'implémenter en langage `C` le type `list` du langage Python, avec la même complexité pour les opérations de base (notamment l'ajout d'un élément).

1 Une première implémentation

On considère le fichier d'en-tête `list.h` suivant :

```
#ifndef __LIST_H__
#define __LIST_H__

typedef long long element;
typedef struct list_t *list_p;

extern list_p list_new();
extern void list_insert(list_p list, element value);
extern void list_append(list_p list, element value);

#endif /* !__LIST_H__ */
```

où
- `list_new()` créée une nouvelle liste ;
- `list_insert()` ajoute un élément en tête de liste ;
- `list_append()` ajoute un élément en queue de liste.

Nous allons par la suite considérer deux versions.

Question 1 : (TD et TP) – Complétez le fichier `list_v1.c` suivant, manipulant les listes simplement chaînées, chaque élément étant de type `long long` (entier sur 64 bits) :

```
#include <stdlib.h>
#include <stdio.h>
#include "list_v1.h"

typedef struct cell_t *cell_p;

typedef struct cell_t
{
  element value;
  cell_p next;
} cell_t;

cell_p cell_new(int value)
{
  cell_p cell = (cell_p) malloc(sizeof(cell_t));
  cell->value = value;
  cell->next = NULL;
  return cell;
}
```

```
typedef struct list_v1_t
{
  cell_p first;
} list_v1_t;

// à compléter
```

(la fonction `malloc()` effectue l'allocation mémoire,
et `sizeof(type)` retourne la taille occupée par le `type`).

2 Première analyse de la complexité

Rappel : Soit N le nombre d'éléments dans la liste. La complexité (en temps) mesure le nombre d'opérations nécessaires pour effectuer une fonction. Les complexités les plus courantes sont :

- $O(1)$: temps constant, indépendant de N ;
- $O(\log(N))$: temps logarithmique ;
- $O(N)$: temps linéaire, proportionnel à N ;
- $O(N\log(N))$;
- $O(N^2)$: temps quadratique.

Question 2 : (TD) – Quelle est la complexité de la fonction `list_v1_insert()` ? Justifiez.

Question 3 : (TD) – Quelle est la complexité de la fonction `list_v1_append()` ? Justifiez.

3 Analyse de la consommation mémoire

La complexité en espace mesurerait, quant à elle, le nombre d'octets nécessaires pour effectuer une fonction, du moins un ordre de grandeur de ce nombre.

Mais nous allons plutôt nous intéresser à la mémoire occupée par la structure données.

Question 4 : (TD, à vérifier en TP) – Combien d'octets sont nécessaires pour une liste de N éléments avec la première implémentation ?

4 Seconde implémentation

Pour économiser de la mémoire, on peut aussi implémenter les listes sous la forme de tableaux extensibles. On utilisera alors la fonction `realloc()`, qui peut changer la taille mémoire allouée, par exemple en doublant celle d'un tableau ainsi :

```
array = (element *) realloc(array, 2*array_size * sizeof(element));
```

Question 5 : (TD et TP) – Complétez le fichier `list_v2.c` suivant :

```
#include <stdlib.h>
#include <stdio.h>
#include "list_v2.h"

typedef struct list_v2_t
{
  element *array;
  size_t array_size;
  size_t number_of_elements;
```

```
} list_v2_t;

// à compléter
```

Question 6 : (TD, à vérifier en TP) – Combien d'octets sont nécessaires pour une liste de N éléments avec cette seconde implémentation ?

Question 7 : (TD et TP) – Écrivez un programme `list_example.c` qui, en séquence :

1. créée une liste V1 `l11` et y ajoute les 65536 premiers entiers avec la fonction `list_v1_insert()` ;
2. créée une liste V1 `l12` et y ajoute les 65536 premiers entiers avec la fonction `list_v1_append()` ;
3. créée une liste V2 `l21` et y ajoute les 65536 premiers entiers avec la fonction `list_v2_insert()` ;
4. créée une liste V2 `l22` et y ajoute les 65536 premiers entiers avec la fonction `list_v2_append()`.

Question 8 : (TP) – Calculez le profil mémoire du programme précédent dans les deux cas suivants :
— `typedef long long element;`
— `typedef int element;`
Que constatez-vous ?

5 Seconde analyse de la complexité

Question 9 : (TD) – Quelle est la complexité de la fonction `list_v2_insert()` ? Justifiez.

Question 10 : (TD) – Quelle est la complexité de la fonction `list_v2_append()` ? Justifiez.

Question 11 : (TP) – Calculez le profil temporel du programme `list_example.c` et vérifiez que ce sont bien les fonctions `list_v1_append()` et `list_v2_insert()` qui sont les plus gourmandes en temps de calcul.

6 Profilage mémoire du type `list` Python

On considère le fichier `time_example.py` suivant :

```
l = list()

def solution1(l):
    for i in range(65536):
        l.append(i)
    return l.reverse()

def solution2(l):
    for i in range(65536):
        l.insert(0, i)
    return l
```

qui donne deux façons de construire une liste contenant, de manière ordonnée, les 65536 premiers nombres entiers.

Question 12 : (TD et TP) – Modifiez ce fichier pour chronométrer ou profiler les deux fonctions `solution1()` et `solution2()`.

Question 13 : (TP) – Comparez les temps d'exécution de ces fonctions. Laquelle des deux solutions est la plus rapide (avec `append()` ou avec `insert`) ? En déduire laquelle des deux implémentations `C` (cellules chaînées ou tableau extensible) il faut retenir.

Le profilage mémoire de `memory_example.py` (cf. TD précédent) montre que c'est l'implémentation par tableau extensible (V2) qui doit être retenue pour des raisons d'espace mémoire.
Mais les profilages temporels de `time_example.py` et `list_example.c` montrent que c'est l'implémentation V1 qui a le même comportement que le type `list` Python du point de vue du temps de calcul / de la complexité. Par conséquent, il faudrait faire une V3 avec un tableau extensible (pour des raisons espace mémoire) mais en y stockant les éléments dans l'ordre inverse (pour des raisons de complexité en temps, pour que `append` soit plus efficace que `insert`).

Pour les plus rapides : à vous de jouer !