Algorithmique et Programmation 2
Travaux Pratiques – Séance 5

À déposer à la fin de la séance sur Arche

1 La librairie `liste.h`

Dans cette séance de TP, nous utiliserons l'enregistrement `struct Liste` pour représenter de très grands entiers naturels.

```
typedef struct Liste* liste;
struct Liste{
  unsigned int  premier ;
  liste suivant;
};
```

L'enregistrement est muni des opérations primitives suivantes :

```
/* SIGNATURES DES OPERATIONS PRIMITIVES */
// constructeurs
liste l_vide () ;
liste cons (int x, liste L) ;
// acces
bool est_vide (liste L) ;
int prem (liste L) ;
liste reste (liste L) ;
void liberer_liste (liste L) ;
```

Vous complèterez le fichier ci-dessous avec les programmes demandés.

```
/* fichier tp5.c */
/* etudiant : nom prenom */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "liste.h"

int main(int argc, char** argv){
  return EXIT_SUCCESS;
}
```

La commande de compilation/exécution est la suivante :

```
super_etudiant@ap2-2021:$ gcc -Wall tp5.c liste.c -o tp5
super_etudiant@ap2-2021:$ ./tp5
```

Question 1 — voir exercice 19 page 28
Écrivez et testez chacune des opérations suivantes :

```
/* SIGNATURES DES OPERATIONS A CODER */
unsigned int longueur(liste L);
bool appartient(int x, liste L);
unsigned int neme_element(unsigned int n, liste L);
liste neme_reste(unsigned int n, liste L);
unsigned int nb_occurrence(int a, liste L);
bool egales(liste L1, liste L2);
unsigned int dernier(liste L1);
bool sous_sequence_de(liste L1, liste L2);
```

Question 2 — voir exercice 19 page 28
Écrivez et testez chacune des opérations suivantes :

```
/* SIGNATURES DES OPERATIONS A CODER */
liste supprimer_element(int x, liste L);
liste snoc(liste L, int e);
liste concatener(liste L1, liste L2);
```

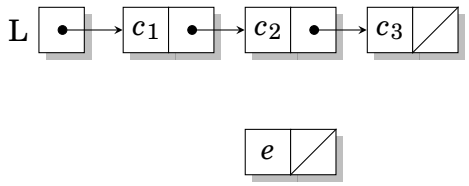
2 Opérations destructrices *vs.* opérations non destructrices

Considérons la fonction `liste inserer_element(int e, unsigned int n, liste L)` codée comme suit :

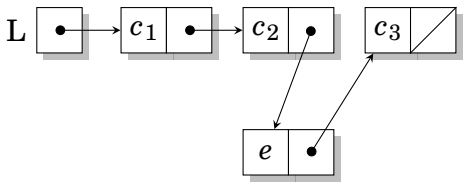
```
liste inserer_element(int e, unsigned int n, liste L){
  if (n == 0){
    printf ("Erreur_liste");
    return (l_vide ()) ;
  }
  if ((n == 1) || (est_vide (L)))
    return (cons (e, L)) ;
  else
    return cons(prem(L), inserer_element (e, n-1, reste(L))) ;
}
```

A chaque appel de la fonction `cons`, la fonction `inserer_element` crée un nouvel élément. A la fin de son exécution, une deuxième liste identique à la liste d'entrée, excepté l'élément `e` sera créée. Cela peut être problématique si la liste `L` est très grande et que la création d'un clone de celle-ci nécessite énormément de mémoire dont nous ne disposons peut-être pas, sans compter le temps nécessaire à l'allocation mémoire pour chaque élément créer.

Nous souhaitons maintenant insérer « physiquement » l'élément `e` dans la liste `L` sans effectuer une copie de chaque élément de celle-ci. Par exemple, on veut modifier la liste L (voir Figure 1a) en la liste de la Figure 1b.



(a) Avant modification



(b) Après modification

Question 3 — Opération non destructrice
Écrivez et tester une procédure `void inserer_element(int e, unsigned int n, liste L)` qui insère l'élement `e` dans la liste `L` sans créer une copie de celle-ci.

3 D'autres opérations sur les listes

Question 4 — voir exercice 19 page 28
Écrivez et testez chacune des opérations suivantes :

```
/* SIGNATURES DES OPERATIONS A CODER */
liste renverser(liste L);
liste supprimer_repetition(liste L);
bool sous_liste_de(liste L1, liste L2);
```