TP: Listes simplement chaînées

Listes simplement chaînées

On donne les types suivants :

```c
// Maillons d'une liste chaînée
typedef struct Element Maillon;
struct Element
{
  int val;
  Maillon *next;
};

typedef struct Liste Liste;
struct Liste
{
    Maillon * first ;
};
```

Création, ajout d'éléments, remplissage

Q.1 Ecrire la fonction `Liste *initialisation(int x)` qui crée un pointeur vers une `Liste` qui pointe vers un élément dont la valeur est `x`.

```c
Liste premier = *( initialisation )(3);
printf("premier. first ->val=%d\n",premier.first->val);// affiche 3

```

Q.2 Ecrire la procédure `void ajouterDebut(Liste *liste, int x)` qui ajoute un élement de valeur `x` en tête de la liste pointée par `liste`.

```c
ajouterDebut(&premier, 5);// où premier est défini en Q1
printf("premier. first ->val=%d\n",premier.first->val);// affiche 5

```

Q.3 Ecrire la procédure `void afficheListe(Liste *liste)` qui affiche dans l'ordre les valeurs des différents éléments de la liste chaînée sur laquelle pointe `liste` et **None** si la liste pointée est vide.

```c
Liste premier = *( initialisation )(3);
for(int i=4; i<8; i++)
    ajouterDebut(??);// ajouter le maillon de valeur i au début

afficheListe (&premier);

Liste *vide = ??; // création d'une liste
vide ?? ?? = NULL;// initialiser la liste comme vide
afficheListe (vide);
// remplacer les ?? par un code correct.
```

On doit obtenir

```
7->6->5->4->3->NULL
None
```

Q.4 Écrire la fonction `Liste* vide()` de création d'une liste vide.

Q.5 Ecrire une procédure `void ajouterFin(Liste *liste, int x)` qui ajoute un élément de valeur `x` à la fin de la liste pointée par `liste`.
Par exemple

```
ajouterFin(&premier, 10);
afficheListe (&premier);

```

génère l'affichage

```
7->6->5->4->3->10->NULL
```

Q.6 On crée une liste `liste` de longueur n. Quelle est en fonction de n la complexité de cette création sachant que

(a) on n'utilise que la procédure `ajouterDebut` ?

(b) on n'utilise que la procédure `ajouterFin` ?

Q.7 Écrire la fonction `void liberer(Liste*)` qui libère tous les pointeurs d'une liste.

Q.8 Ecrire la fonction `Liste * challenge(int n)` qui prend en paramètre un entier `n` et retourne un pointeur sur une liste de longueur $2n+1$ dont les $n+1$ premiers éléments sont $n, n-1, \ldots, 0$ et les n derniers sont $1, 2, \ldots, n$. Le challenge consiste à utiliser les seules fonctions déjà créées (sans aucune fonction auxiliaire) et une seule boucle sans aucun appel récursif à `challenge`. La complexité doit être linéaire.
Par exemple

```
Liste *chg5 = challenge(5);
afficheListe (chg5);

```

affiche

```
5->4->3->2->1->0->1->2->3->4->5->NULL
```

Q.9 Ecrire une fonction `bool estVide(Liste* liste)` qui renvoie `true` ou `false` selon que la liste pointée par `liste` est vide ou non.

```
printf("%d\n", estVide(vide));// affiche 1
printf("%d\n", estVide(chg5));// affiche 0

```

Longueur, position, suppression

Q.10 Ecrire la fonction `int length(Liste *liste)` qui retourne le nombre d'éléments de la liste pointée par `liste`. La liste vide est de longueur 0.

Q.11 Ecrire la fonction `Maillon* position(Liste *liste, int p)` qui renvoie l'adresse du maillon en position p de la liste dont l'adresse est passée en argument.
Une erreur d'assertion est soulevée si p est trop grand.

Q.12 Ecrire la fonction `int supprimer(Liste *liste, int pos)` qui supprime l'élément en position `pos` de la liste pointée par `liste` et retourne la valeur de ce maillon. Une erreur d'assertion est soulevée si la position est trop grande.
Ne pas oublier de libérer la mémoiree allouée au pointeur vers le maillon supprmé.

Q.13 Ecrire la fonction dont voici le prototype :

```
int trouve(Liste *liste, int v);
// retourne la position du 1er maillon dont la valeur est v
// -1 si on ne trouve pas

```

Q.14 Tout ce qu'il manque à nos listes pour être qualifiées de piles, c'est une fonction `int pop(Liste*)`. L'implanter.