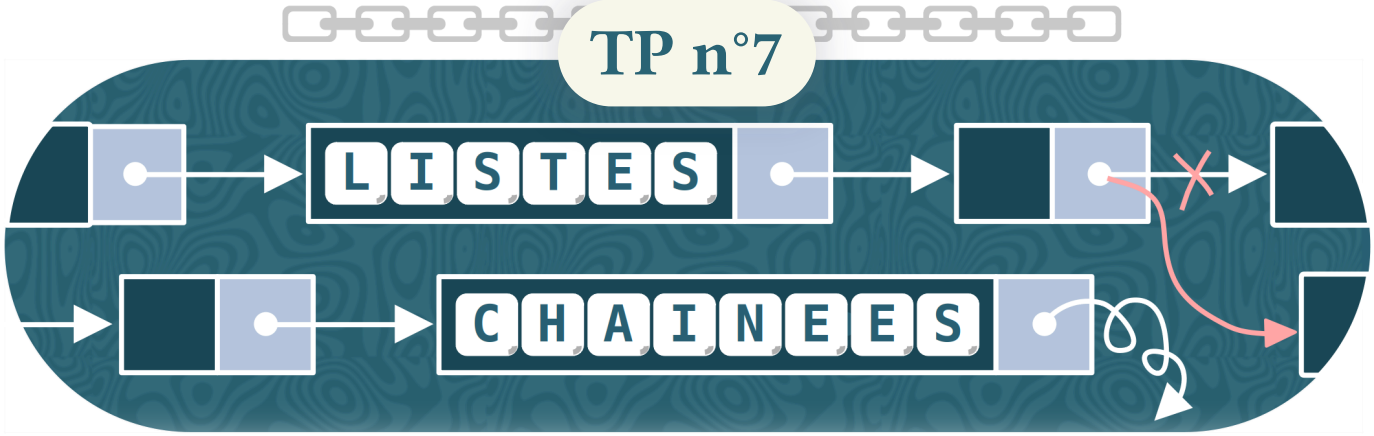
TP n°7



I. Le maillon : élément de base d'une liste chaînée

Type structuré pour un maillon de liste simplement chaînée

Le bloc de base d'une liste chaînée est un **maillon**, qui contient une valeur (ici on prendra un entier, mais on peut mettre ce que l'on souhaite) et un pointeur vers le maillon suivant.




```c
struct maillon_ {
  int valeur;
  struct maillon_* suivant;
};
```

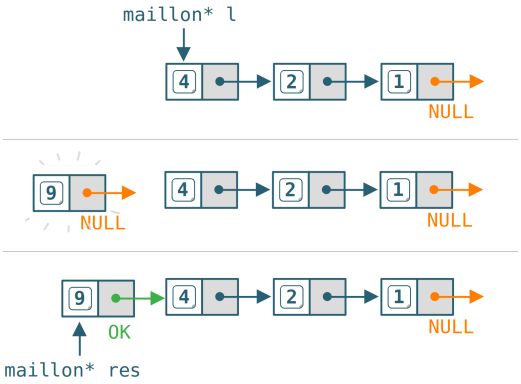
On peut directement définir et renommer un tel type structuré de la forme suivante :

```c
typedef struct maillon_ {
  int valeur;
  struct maillon_* suivant;
} maillon;
```

1 Après avoir défini ce nouveau type structuré, implémenter une fonction de prototype `maillon* nouveau_maillon(int elem)` qui alloue un nouveau maillon sur le tas et renvoie le pointeur associé, après avoir initialisé le champ `valeur` à `elem`, et le pointeur `suivant` au pointeur `NULL`.

II. Du maillon à la liste simplement chaînée

Pour construire une **liste simplement chaînée**, on crée des maillons uns par uns. Chaque nouveau maillon pointe vers le début de l'ancienne liste. On détaille ci-dessous l'ajout d'un quatrième maillon à une liste de trois maillons :

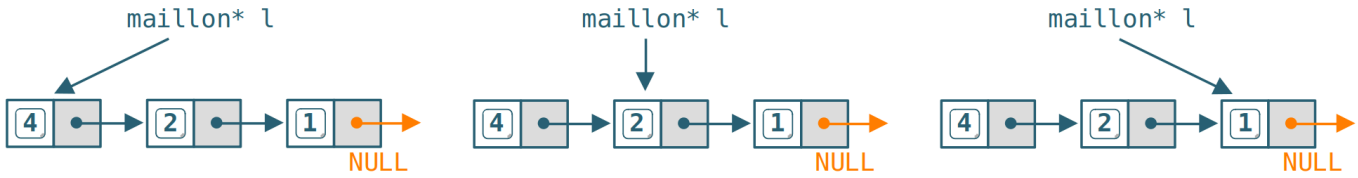


- On part d'une liste de trois maillons et on souhaite en rajouter un quatrième avec une nouvelle valeur.
- On commence par allouer un nouveau maillon sur le tas, et on l'intialise correctement avec sa valeur (ici 9).
- On modifie alors le pointeur `suivant` de ce nouveau maillon pour le faire pointer vers le début de l'ancienne liste. On renvoie un pointeur `res`, pointant vers ce nouveau début de liste.

2 Implémenter une fonction de prototype `maillon* append(maillon* l, int elem)` qui alloue un nouveau maillon sur le tas et renvoie le pointeur associé, après voir intialisé son champ `valeur` avec la valeur donnée et son champ `pointeur` avec le pointeur donné pointant au début de l'ancienne liste.

III. Parcours d'une liste

Parcourir une liste signifie visiter chaque maillon une fois dans l'ordre. Pour parcourir une liste simplement chaînée, on part du pointeur vers le premier maillon, qui sera le maillon actuel. Puis, **tant que** ce pointeur est non nul, on le met à jour en le remplaçant par le pointeur `suivant` du maillon actuel pour se déplacer vers le maillon suivant.



3 Implémenter une fonction de prototype `void affiche_liste(maillon* l)` qui parcourt une liste simplement chaînée et affiche les entiers présents dans chaque maillon.

4 Implémenter une fonction de prototype `int longueur(maillon* l)` qui parcourt une liste simplement chaînée et renvoie le nombre de maillons qu'elle contient.

5 Implémenter une fonction de prototype `int element_n(maillon* l, int n)` qui parcourt une liste et renvoie la valeur contenue dans le $n$ème maillon. On supposera que le premier maillon de la liste est celui d'indice 0, et qu'il y a bien au moins $n+1$ maillons dans la liste en question.

6 Implémenter une fonction de prototype `bool mem(maillon* l, int elem)` qui parcourt une liste et renvoie `true` si un des maillons contient la valeur `elem`, et `false` sinon.

7 Implémenter une fonction de prototype `bool est_triee(maillon* l)` qui parcourt une liste simplement chaînée et vérifie si les éléments contenus dans les maillons sont de plus en plus grands.

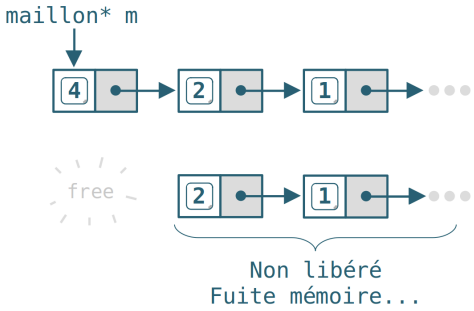
IV. Tests

8 Dans une fonction de tests, créer trois listes simplement chaînées suffisament grandes. Il faudra pour cela ajouter les maillons un par un à partir de la liste vide : le pointeur `NULL`.

9 Poursuivre l'écriture de cette fonction de tests, en y ajoutant divers tests vérifiant le bon comportement des fonctions précedemment implémentées.

10 Pour terminer cette fonction de tests, il est impératif de libérer les listes créées. Sinon, cette fonction provoquera une fuite mémoire. Réfléchir à une telle fonction de libération avant de lire la suite.

V. Libération d'une liste simplement chaînée



La libération d'une liste chaînée n'est pas immédiate. Comme on ne possède qu'un pointeur `m` vers le premier maillon de la liste, `free(m)` ne va pas libérer la liste mais uniquement ce premier maillon.

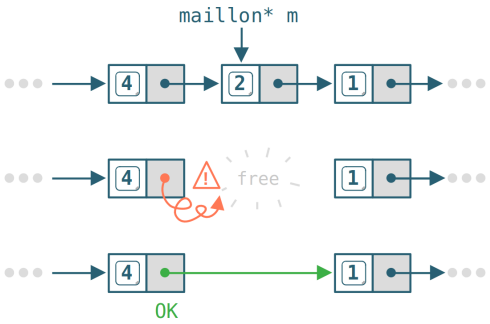
Cela ne libère donc pas tous les maillons suivants et provoque généralement une **fuite mémoire**.

Une bonne manière de faire est d'écrire une fonction dédiée pour libérer une telle structure. Dans cette fonction, on regarde le pointeur vers le maillon suivant du maillon actuel : tant que ce pointeur n'est pas `NULL`, on en garde une copie, on libère le maillon actuel puis le nouveau maillon actuel devient ce maillon suivant, au bout du pointeur gardé.

11 Faire un schéma évolutif illustrant cette fonction de libération. Implémenter une fonction de prototype `void libere_liste(maillon* l)` qui libère convenablement une liste simplement chaînée.

12 Terminer votre fonction de tests et l'appeler dans `main` pour vérifier que tout est bon.

VI. Suppression d'un élément



Si l'on souhaite supprimer un maillon au milieu d'une liste, il ne suffit pas de libérer le maillon. Cela crée potentiellement un **pointeur pendouillant**, et la liste est désormais coupée en deux morceaux.

Une bonne manière de faire est de ne pas chercher à libérer le maillon actuel, mais le maillon suivant.

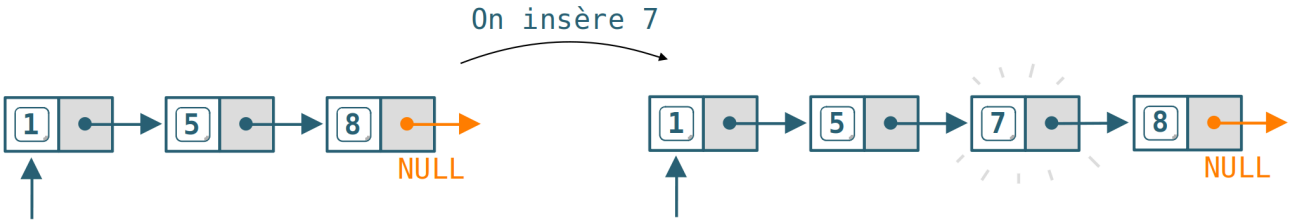
13 Faire un schéma évolutif illustrant étape par étape comment supprimer le maillon suivant depuis un maillon actuel, en précisant quels pointeurs doivent être modifiés et comment.

14 Implémenter une fonction de prototype `maillon* supprime_n(maillon* l, int n)` qui supprime le $n$ème maillon d'une liste simplement chaînée et renvoie un pointeur vers la nouvelle liste ainsi modifiée. On considérera que le premier maillon est d'indice $i = 0$, et qu'il n'y a rien à supprimer si la liste possède i maillons ou moins.

VII. Deux applications

Tri par insertion

On s'intéresse ici aux listes dont les éléments sont **triés par ordre croissant**. Si l'on souhaite ajouter un élément à une telle liste tout en maintenant la liste triée, on parcourt la liste depuis le premier maillon **tant que** le maillon suivant existe et est de valeur plus petite que l'élément. Une fois que ce n'est plus le cas, c'est que l'on doit **insérer** ce nouveau maillon entre le maillon actuel et l'éventuel maillon suivant.



15 Faire un schéma illustrant l'insertion d'un maillon au milieu d'une liste simplement chaînée, en prenant soin d'indiquer quels pointeurs doivent être modifiés et comment.

16 Implémenter une fonction de prototype `maillon* insere(maillon* l, int elem)` qui insère un nouveau maillon alloué dans le tas, contenant la valeur `elem` dans une liste supposée triée au bon endroit afin de maintenir la nouvelle liste triée.

17 Implémenter une fonction de prototype `maillon* tri_insertion(int t[], int taille)` qui insère itérativement chaque élément d'un tableau dans une liste à l'aide de la fonction précédente, afin de construire une liste triée de ces éléments.

Détection de cycle

On se donne une fonction $f : \mathbb{N} \mapsto \mathbb{N}$, et une suite définie par un U_0 quelconque et $U_{n+1} = f(U_n)$. On s'intéresse au premier cycle de cette suite, c'est-à-dire aux premiers valeurs $U_i, U_{i+1}, \ldots, U_j$ rencontrées telles que $U_i = U_j$. Une liste chaînée est ici particulièrement intéressant. En effet, il faut pouvoir facilement ajouter de nouvelles valeurs, et les stocker toutes. On ne connaît pas à l'avance la taille que prendra ce cycle.

18 Implémenter une fonction de prototype `int f(int n)` qui renvoie $\frac{n}{2}$ si n est pair, et $\frac{n+31}{2}$ s'il est impair.

19 Implémenter une fonction de prototype `int taille_cycle(int n)` qui renvoie la taille du premier cycle rencontré en partant de $U_0 = n$. Tester sur quelques exemples.

20 Il existe une manière plus efficace de résoudre ce problème. Regarder, comprendre et implémenter l'algorithme du lièvre et de la tortue de Floyd (ou Knuth ?) : https://fr.wikipedia.org/wiki/Algorithme_du_li%C3%A8vre_et_de_la_tortue