TP n°13 – Listes doublement chaînées

Une liste doublement chaînée est une structure chaînée dans laquelle chaque maillon contient un lien vers le maillon suivant et un lien vers le maillon précédent. L'avantage est de pouvoir supprimer un maillon quelconque en temps constant, contrairement à la structure chaînée simple.

Attention : on ne confondra pas la **liste** (la structure abstraite de données) et les listes doublement chainées. Dans la suite on écrira **liste** pour la structure abstraite et liste pour la structure doublement chainée.

1 Première version

Pour imiter ce qu'on a fait avec le chainage simple, on peut utiliser la structure suivante :

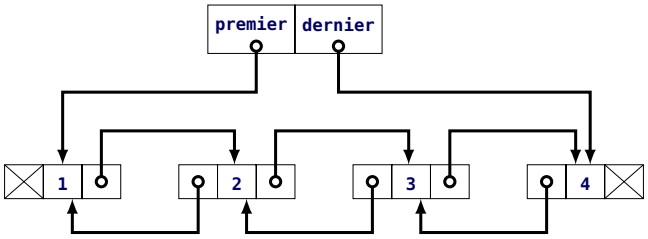

Figure 1 - Liste doublement chaînée `(1, 2, 3, 4)`, première version.

On représente cette structure avec les définitions suivantes en C.

```c
struct MaillonDouble {
    int valeur;
    struct MaillonDouble* prec;
    struct MaillonDouble* suiv;
};

typedef struct MaillonDouble maillondouble;
```

```c
struct LDC1 {
    maillondouble* premier;
    maillondouble* dernier;
};

typedef struct LDC1 ldc1;
```

Si `u` est de type `ldc1*`, alors on utilise les conventions suivantes :

- la liste est vide si et seulement si `u->premier == NULL` et `u->dernier == NULL` ;
- si la liste n'est pas vide, alors `u->premier->prec == NULL`, `u->dernier->suiv == NULL`, et ce sont les deux seuls pointeurs nuls présents dans la structure chaînée.

On donne les fonctions permettant de créer un nouveau maillon (avec une valeur donnée et des pointeurs nuls des deux côtés) et une nouvelle liste (vide) :

```c
maillondouble* nouveau_maillon(int x){
    maillondouble* n = malloc(sizeof(maillondouble));
    n->valeur = x;
    n->suiv = NULL;
    n->prec = NULL;
    return n;
}
```

```c
ldc1* new_ldc1(){
    ldc1 *d = malloc(sizeof(ldc1));
    d->premier = NULL;
    d->dernier = NULL;
    return d;
}
```

- **Q1.** Écrire une fonction `void retire_maillon(ldc1* d, maillondouble* n)` qui supprime un maillon d'une liste. Cette fonction prend un pointeur vers le maillon et un pointeur vers la liste en arguments, et elle gère à la fois la suppression du maillon et la libération de la mémoire correspondante.
- **Q2.** Écrire une fonction `void insere_avant(ldc1* d, maillondouble* n, int x)` qui insère un maillon (avec la valeur fournie) juste avant le maillon passé en argument.
- **Q3.** Écrire la fonction symétrique `void insere_apres(ldc1* d, maillondouble* n, int x)`.
- **Q4.** Ces fonctions sont-elles suffisantes pour écrire par exemple une fonction `ldc1* init_tableau(int* tab, int long)` (initialise une liste doublement chainée avec les valeurs d'un tableau) ? Si non, pourquoi ?

2 Version avec sentinelle

On pourrait écrire les fonctions manquantes pour la structure précédente, mais les fonctions d'ajout et de retrait ont pas mal de cas particuliers à considérer (liste vide, de 1 ou de 2 éléments). On peut s'éviter tous les cas particuliers en changeant légèrement la structure.

On ne change rien au type `maillondouble`, mais on convient de rajouter un maillon « fictif », appelé sentinelle, à l'extrémité de la liste. La valeur présente dans le champ valeur de ce maillon ne sera pas significative (représenté par ?), et la liste aura la structure suivante :

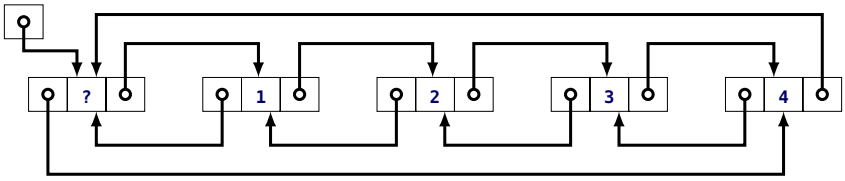

Figure 2 - Liste doublement chaînée `(1, 2, 3, 4)`, version avec sentinelle.

On change donc de type pour représenter nos listes doublements chainées :

```
struct LDC {
    maillondouble* sentinel; //Pointeur vers la sentinelle
};

typedef struct LDC ldc;
```

- **Q5.** Écrire la fonction `ldc* nouvelle_ldc()` qui crée une nouvelle liste doublement chaînée. Cette liste sera vide, ce qui signifie qu'elle ne contiendra que le maillon sentinelle, correctement initialisé.
- **Q6.** Ré-écrire les fonctions `retire_maillon`, `insere_avant` et `insere_apres` pour la nouvelle structure. La fonction `retire_maillon` pourra supposer sans le vérifier que le maillon passé en argument n'est pas le maillon sentinelle (et aucune de ces fonctions n'aura besoin de prendre la liste elle-même en argument).
- **Q7.** Écrire une fonction `void detruit_ldc(ldc* d)` qui libère la totalité de la mémoire utilisée par une liste doublement chaînée.
- **Q8.** Écrire une fonction `ldc* init_tableau(int* t, int long)` qui convertit un tableau en liste doublement chaînée.
- **Q9.** Écrire les primitives de la **structure de données liste** (cf cours) avec la structure doublement chainée.

3 Nombres chanceux

Les nombres chanceux sont définis par le processus suivant :

- on part de la liste des entiers impairs (jusqu'à une certaine borne n) ; l'entier 1 est *chanceux* ;
- on considère l'entier qui suit 1 dans la liste (c'est 3) ;
- on élimine un nombre sur 3 de la liste, en commençant au début ; l'entier 3 est *chanceux* ;
- on considère l'entier qui suit 3 dans la liste (c'est 7) ;
- on élimine un nombre sur 7 de la liste, en commençant au début ; l'entier 7 est *chanceux* ...

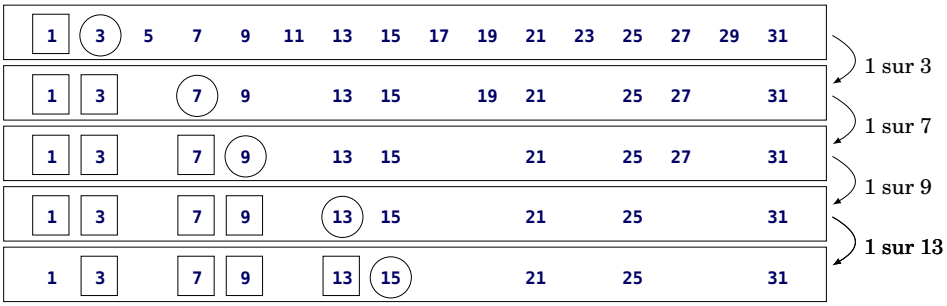

Figure 3 - Génération des nombres chanceux.

- **Q10.** Combien vaut la somme des nombres chanceux inférieurs ou égaux à 105 ? Utiliser une liste doublement chaînée pour répondre à la question.
  Remarque : Ce n'est pas la méthode la plus simple ou la plus efficace, il s'agit d'une application de ce qu'on a fait au-dessus.
- **Q11.** (Bonus) Une liste doublement chaînée permet de réaliser facilement la structure abstraite de `deque` (double ended queue, ou file bilatère).
  Il s'agit d'une structure séquentielle (impérative) à laquelle on peut ajouter des éléments, à droite et à gauche ; on peut également retirer l'élément le plus à droite et l'élément le plus à gauche
  Écrire les quatre fonctions suivantes qui effectuent l'ajout et le retrait d'un élément sur un côté de la structure (le côté est écrit dans le nom des fonctions). Pour `retire_gauche` et `retire_droite`, on vérifiera la licéité de l'appel à l'aide d'un `assert`.

```
void ajoute_gauche(ldc *d, int x);
void ajoute_droite(ldc *d, int x);
int retire_gauche(ldc *d);
int retire_droite(ldc *d);
```

D'après un TP de JB. Bianquis