Listes chainées en C

On rappelle les options de compilations, particulièrement importantes aujourd'hui :

```
$ gcc -fsanitize=address,undefined -Wall -Wextra -Wvla prog.c -o prog
```

On ignorera les *warnings* relatifs aux fuites de mémoire jusqu'à avoir écrit la fonction `free_list`, mais on y sera très attentif ensuite.

1 Listes simplement chaînées

Plusieurs variations sont possibles pour définir des listes simplement chaînées en C. Ici, on va utiliser l'une des plus simples.

```c
typedef int T;

struct node {
    T data;
    struct node *next;
};

typedef struct node node;
```

En général, on n'aura pas besoin de supposer que `T` est égal à `int`, mais on supposera en revanche que c'est un type numérique, pour pouvoir faire des additions ou des comparaisons par exemple. Une liste sera simplement un `node *`.

— La liste vide est représentée par le pointeur `NULL`.
— Une liste non vide est représentée par un pointeur vers son premier nœud.

1.1 Création de listes

1. Écrire une fonction créant un nouveau `node` et renvoyant un pointeur vers ce `node`, avec le champ `data` initialisé avec l'argument fourni et le champ `next` initialisé à `NULL`.

```c
node *new_node(T x);
```

2. Écrire une fonction `cons` prenant en entrée une liste `u` et un argument `x`, et renvoyant une liste constitué d'un nœud contenant `x`, suivi de `u`.

```c
node *cons(node *u, T x);
```

3. Écrire une fonction `from_array` ayant le prototype suivant :

```c
node *from_array(T a[], int n);
```

— `a` est un tableau d'objets de type `T`, de longueur `n`.
— La fonction renvoie une liste contenant les mêmes éléments que `a`, dans l'ordre. Autrement dit, l'élément de tête de la liste est `a[0]`.

À partir de ce point, vous trouverez dans le squelette un certain nombre de fonctions vous permettant de tester votre code. Il est strictement interdit de passer à la question $n + 1$ si le test associé à la question n échoue !

1.2 Parcours de listes

1. Écrire une fonction `free_list` qui libère toute la mémoire utilisée par une liste.

```c
void free_list(node *u);
```

2. Écrire une fonction `length` calculant la longueur d'une liste. On écrira une fonction purement itérative.

```
int length(node *u);
```

3. Écrire une fonction `print_list` affichant les éléments d'une liste. On produira l'affichage suivant, terminé ou non par un retour à la ligne suivant la valeur du paramètre `newline`.

```
[7 1 3 4 12]
```

   On pourra supposer ici que `T` est égal à `int`.

```
void print_list(node *u, bool newline);
```

4. Écrire une fonction `to_array` convertissant une liste de longueur n en un tableau de taille n. On stockera cette taille dans `*n`. L'élément en tête de la liste se retrouvera à l'indice 0.

```
T *to_array(node *u, int *n);
```

5. Écrire une fonction `is_equal` qui teste l'égalité de deux listes. On parle ici d'égalité *structurelle*, c'est-à-dire vérifiée lorsque les deux listes ont les mêmes éléments, dans le même ordre, et non d'égalité *physique* qui est vérifiée lorsque les deux pointeurs désignent la même liste.

```
bool is_equal(node *u, node *v);
```

1.3 Ordre et tri

1. Écrire une fonction non récursive `is_sorted` qui prend en entrée une liste et renvoie un booléen indiquant si elle est triée par ordre croissant.

```
bool is_sorted(node *u);
```

2. Écrire une fonction récursive `insert_rec` ayant le prototype suivant :

```
node *insert_rec(node *u, T x);
```

   — `u` est une liste, éventuellement vide, supposée triée par ordre croissant.
   — La fonction renvoie une liste triée par ordre croissant contenant les mêmes éléments que `u`, plus une occurrence de `x`.

   Cette fonction créera un seul nouveau nœud !

3. Faire un schéma mémoire illustrant ce qui se passe, en OCaml d'une part et en C d'autre part, lorsque :
   — On part de u vide et l'on fait `v = insert(u, 10)` (ou `let v = insert u 10`).
   — On part de $u = (1, 2, 5, 6)$ et l'on fait `v = insert(u, 0)` (ou l'équivalent OCaml).
   — On part de $u = (1, 2, 5, 6)$ et l'on fait `v = insert(u, 4)` (ou l'équivalent OCaml).
4. Pourquoi vaut-il mieux considérer que la variable `u` est « invalide » après l'instruction `v = insert(u, x)` ? Autrement dit, pourquoi vaut-il mieux se limiter à des instructions du type `u = insert(u, x)` ?
5. Pourquoi serait-il problématique d'avoir en C une fonction d'insertion se comportant comme en OCaml ?
6. Écrire une fonction récursive `insertion_sort_rec` prenant en entrée une liste, éventuellement vide, et renvoyant une liste triée contenant les mêmes éléments. La liste renvoyée ne partagera pas sa représentation mémoire avec la liste initiale qui n'aura pas été modifiée.

```
node *insertion_sort_rec(node *u);
```

1.4 Reverse

1. Écrire une fonction `reverse_copy` qui prend en argument une liste et renvoie une copie de cette liste, à l'envers. La liste et sa copie seront totalement indépendantes en mémoire.

```
node *reverse_copy(node *u);
```

2. Écrire une fonction `copy_rec`, récursive, qui effectue une copie d'une liste, à l'endroit.

```
node *copy_rec(node *u);
```

3. Écrire une fonction `copy` ayant la même spécification mais n'étant pas récursive.
4. Écrire une fonction `reverse` renversant une liste « en place », c'est-à-dire sans créer aucun nouveau nœud. On renverra un pointeur vers le nœud de tête de la nouvelle liste, qui était donc le nœud de queue de l'ancienne liste.

```
node *reverse(node *u);
```

2 Piles et files

En général, il est nettement plus pertinent d'utiliser des tableaux plutôt que des listes chaînées pour implémenter les piles et les files en C. Les tableaux peuvent être dynamiques si l'on ne peut pas se contenter de structures ayant une capacité fixée à la création. Il faut donc voir ce qui suit comme des exercices, intéressants dans le cadre scolaire mais ne correspondant pas vraiment à ce qu'on ferait en réalité.

2.1 Pile à l'aide d'une liste chaînée

On pourrait fournir l'interface minimale d'une pile en utilisant simplement un `node*`. Autrement dit, une liste. Ici, on choisit une variante légèrement différente :

```
struct stack {
  int len;
  node *top;
};

typedef struct stack stack;
```

Écrire les fonctions suivantes :

1. `empty_stack` qui renvoie un pointeur vers une nouvelle pile vide.

```
stack *empty_stack(void);
```

2. `peek` qui renvoie l'élément situé au sommet d'une pile. On mettra un `assert` pour vérifier que la pile n'est pas vide.

```
T peek(stack *s);
```

3. `push` qui rajoute un élément au sommet d'une pile.

```
void push(stack *s, T x);
```

4. `pop` qui supprime l'élément situé au sommet d'une pile, et le renvoie. À nouveau, on mettra une assertion pour générer une erreur prévisible si la pile est vide.

```
T pop(stack *s);
```

5. `free_stack` qui libère toute la mémoire associée à une pile.

```
void free_stack(stack *s);
```

On fera bien attention à maintenir l'invariant sur la longueur : `s.len` doit toujours être égal à la longueur de la liste `s.top`.

2.2 File à l'aide d'une liste simplement chaînée

On peut implémenter une file de manière efficace (c'est-à-dire avec insertion et extraction en O(1), le facteur constant étant en revanche assez mauvais par rapport à ce qu'on obtiendrait avec un tableau) en utilisant une liste simplement chaînée, à condition que cette liste soit mutable. En C cela ne posera aucun problème, mais en OCaml cela demanderait de définir un nouveau type liste. L'idée a été expliquée dans le chapitre *Piles et files*, on remet les illustrations :

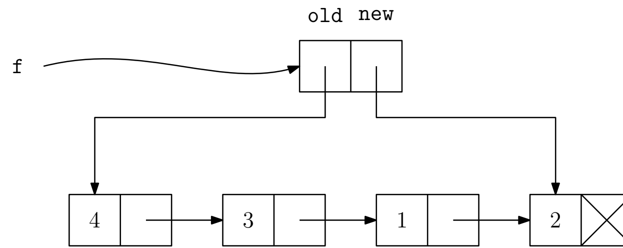

FIGURE 1 – La file ← $(4, 3, 1, 2)$ ← (4 est l'élément le plus ancien).

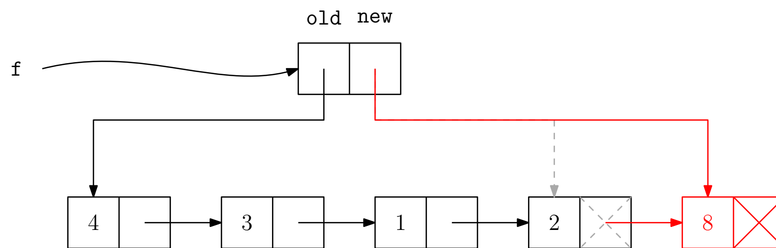

FIGURE 2 – Ajout de l'élément 8.

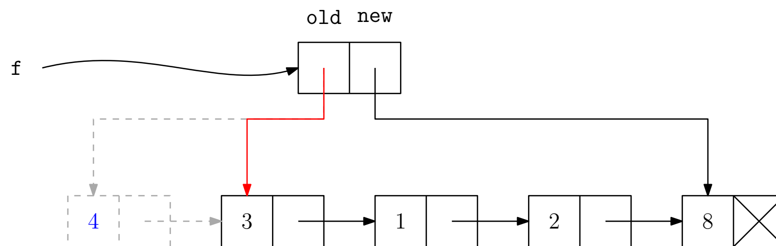

FIGURE 3 – Extraction d'un élément.

À nouveau, on choisit de stocker également la longueur dans la structure de file. On obtient donc :

```
struct queue {
  int len;
  node *left;
  node *right;
};

typedef struct queue queue;
```

La convention est la même que sur les illustrations :
— Les pointeurs de la liste vont de la gauche vers la droite.
— Les insertions se font à droite.
— Les extractions se font à gauche.

1. Pourrait-on, efficacement, faire des insertions à gauche ? des extractions à droite ?
2. Écrire les fonctions suivantes, dont la spécification devrait être claire.

```
queue *empty_queue(void);
```

```
void free_queue(queue *q);
```

```
T peek_left(queue *q);
```

```
void push_right(queue *q, T data);
```

```
T pop_left(queue *q);
```

3 Tris sur les listes chainées

3.1 Tri insertion

Écrire une version purement itérative du tri insertion.

3.2 Tri fusion

1. Écrire une fonction **split** qui prend en argument une liste u et un entier n et a le comportement suivant :
   — Elle renvoie une liste composée des $|u| - n$ derniers éléments de u.
   — Elle modifie u de manière à ce qu'elle ne contienne plus que ses n premiers éléments.
   Cette fonction sera purement itérative, ne créera aucun nouveau nœud, et traitera les cas où $n > |u|$ avec des assertions.

```
node *split(node *u, int n);
```

2. Écrire une fonction **merge** qui prend en entrée deux listes supposées croissantes et renvoie leur fusion. Cette fonction sera purement itérative et ne créera aucun nouveau nœud.

```
node *merge(node *u, node *v);
```

3. Écrire une fonction **merge_sort** qui trie une liste en utilisant l'algorithme du tri fusion. Ce tri se fera « en place », dans le sens où l'on ne créera aucun nouveau nœud :on ne fera que réarranger des pointeurs.

```
node *merge_sort(node *u);
```