Informatique en MP(2)I

Marc de Falco

Séquences et ses implémentations, tableaux, listes chaîn

I	**Structure abstraite séquence ou liste**	**1**
II	**Implémentations**	**1**
III	**Implémentations concrètes des Listes chaînées**	**1**
III.1	En C	1
III.2	En OCaml	5
III.3	Structure de la mémoire	8
IV	**Travaux pratiques**	**8**
IV.1	Tableaux non statiques et tableaux dynamiques	8
IV.2	Listes chaînées	13

I Structure abstraite séquence ou liste

La structure abstraite *liste*, ou également *séquence* dans des contextes, comme en `OCaml`, où le terme `liste` fait références aux listes chaînées, est la structure la plus simple pour stocker des données.

Une *séquence* d'éléments de type `t` est un type `S(t)` dont les éléments représentent des valeurs du type `t` rangées séquentiellement dans un ordre, de la première à la dernière valeur. La vision logique la plus proche de cela est d'imaginer des cases ayant un indice, en commençant en général à l'indice `0`, et contenant des valeurs.

II Implémentations

III Implémentations concrètes des Listes chaînées

III.1 En C

III.1.i Représentation et type

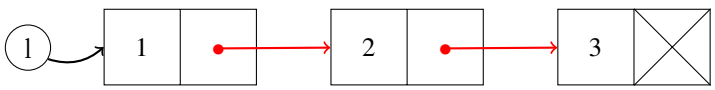
Il existe de nombreuses possibilités d'implémentation des listes chaînées en `C`. On présente ici les opérations autour d'une implémentation et on discutera ensuite des alternatives.

Dans les limites du programme, on ne présente que des listes permettant de contenir un même type, ici des entiers.

Une liste est ainsi un pointeur sur un maillon et un maillon est un couple `(valeur, suivant)` représenté dans une `struct` où `suivant` pointe vers le prochain maillon de la chaîne. Le pointeur nul, de valeur `NULL`, permet ainsi de représenter la liste vide.

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

On représentera graphiquement le pointeur nul par une croix et les maillons par des blocs contenant une valeur et un pointeur. Ainsi, le dernier maillon de la liste contient une croix. La liste `l` correspondant à la valeur qu'on pourrait noter `[1,2,3]` sera représentée ainsi :



III.1.ii Constructeur

On parle de constructeur pour des fonctions qui permettent d'allouer et d'initialiser une valeur d'une structure de donnée. Ici, comme les listes sont des pointeurs sur des maillons, il s'agit uniquement de créer un maillon. Pour cela, on va utiliser `malloc` pour allouer dynamiquement un nouveau maillon.

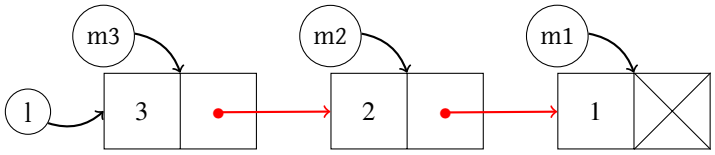
```c
maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}
```

En fait, ce constructeur pourrait être découpé en deux parties : l'allocation qui va se charger de récupérer un emplacement mémoire pour le maillon et l'initialisation qui va attribuer des valeurs.

```c
maillon *maillon_allouer()
{
    return malloc(sizeof(maillon));
}

void maillon_initialiser(maillon *m, int valeur, maillon *suivant)
{
    m->valeur = valeur;
    m->suivant = suivant;
}

maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = maillon_allouer();
    maillon_initialiser(m, valeur, suivant);
    return m;
}
```

On peut alors commencer à créer des listes en enchainant les maillons :

```c
maillon *m1 = creer_maillon(1, NULL); // pas de suivant, c'est le dernier maillon
maillon *m2 = creer_maillon(2, m1);
maillon *m3 = creer_maillon(3, m2);
liste l = m3; // la liste pointe sur le premier maillon
```



Comme le pointeur sur le premier maillon suffit ici, on aurait pu directement écrire :

```c
liste l = creer_maillon(3,
            creer_maillon(2,
                creer_maillon(1, NULL)));
```

III.1.iii Destructeur

Pour détruire un maillon, il suffit de libérer l'espace qu'on lui a attribué.

```c
void maillon_detruire(maillon *m)
{
    free(m);
}
```

Pour détruire une liste, on va par contre avoir besoin de parcourir l'ensemble des maillons qui la constitue. Comme pour les autres parcours, on a alors deux choix :

- **parcours récursif** on a un cas de base quand la liste est vide et dans le cas général, un éventuel appel récursif sur le pointeur `suivant`.

```c
void liste_detruire(liste l)
{
    if (l != NULL)
    {
        liste_detruire(l->suivant);
        maillon_detruire(l); // Attention à l'ordre pour l->suivant
    }
}
```

- **parcours impératif** on boucle tant que la liste est non nulle. On fait ici attention à ne pas accéder à `->suivant` après avoir libéré le maillon.

```c
void liste_detruire(liste l)
{
    while (l != NULL)
    {
        liste suivante = l->suivant;
        maillon_detruire(l);
        l = suivante;
    }
}
```

III.1.iv Ajout et suppression en tête

Pour ajouter ou supprimer un maillon en tête de la liste, on va avoir besoin de modifier le pointeur vers le premier maillon. Pour cela, on a deux approches possibles :

- passer un pointeur vers la liste elle-même, c'est-à-dire un pointeur sur un pointeur sur un maillon, ce qui en C aura le type `maillon **`, qui s'écrit aussi `liste *`. On aura alors le prototype :

```c
void liste_ajout_en_tete(liste *l, int valeur);
```

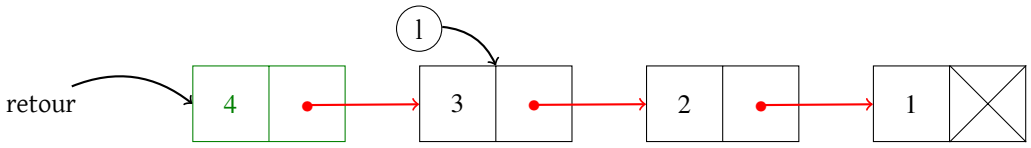
- renvoyer un pointeur vers le nouveau premier maillon. On aura alors le prototype :

```c
liste liste_ajout_en_tete(liste l, int valeur);
```

On présente ici les versions renvoyant une nouvelle liste :

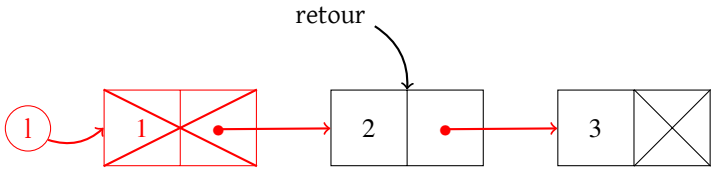
```c
liste liste_ajout_en_tete(liste l, int valeur)
{
    maillon *m = maillon_creer(valeur, l);
    return m;
}
```

Si `l` est la liste précédente contenant `3,2,1` et qu'on ajoute 4 en tête, on va donc directement créer un nouveau maillon et renvoyer un pointeur vers celui-ci.




```c
liste liste_suppr_en_tete(liste l)
{
    assert(l != NULL);
    liste queue = l->suivant;
    maillon_detruire(l); // Attention, on détruit juste le maillon, pas la liste
    return queue;
}
```

Si `l` est la liste précédente contenant `3,2,1` et qu'on supprime le maillon de tête, on va renvoyer un pointeur sur le second maillon. **Attention**, ici le pointeur initial `l` est devenu invalide.



III.1.v Longueur de la liste

On présente ici le calcul de la longueur comme exemple de parcours de la liste. C'est encore très proche du parcours effectué dans le destructeur.

```c
int liste_longueur(liste l)
{
    int longueur = 0;
    while(l != NULL)
    {
        longueur = longueur + 1;
        l = l->suivant;
    }
    return longueur;
}
```

III.1.vi Accès au nième maillon de la liste

On effectue un parcours similaire pour accéder au nième maillon.

```c
maillon *liste_nieme(liste l, int n)
{
    while(n > 0)
    {
        assert(l != NULL);
        l = l->suivant;
        n = n-1;
    }
    assert(l != NULL);
    return l;
}
```

Toujours avec un programme similaire, on peut chercher un maillon avec sa valeur :

```c
maillon *liste_recherche(liste l, int x)
{
    while(l != NULL && l->valeur != x)
    {
        l = l->suivant;
    }
    return l;
}
```

Ici, pas besoin d'asserts, en cas d'échec de la recherche, on renvoie un pointeur nul.

III.1.vii Ajout/Suppression ailleurs qu'en tête

Pour ajouter ou supprimer ailleurs qu'en tête, il est nécessaire de pouvoir repérer précisément un maillon. Pour cela, on peut le faire :

- par son indice, celui utilisé dans `liste_nieme`;
- par sa valeur, avec une recherche;
- ou encore directement par un pointeur sur le maillon.

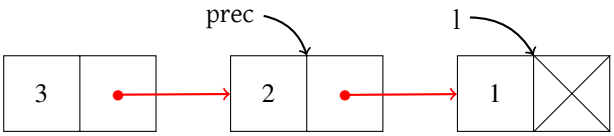
Une fois le maillon ajouté/supprimé, on peut procéder comme pour un ajout/suppression en tête, cependant il va falloir reconnecter le pointeur `suivant` du maillon précédent. Pour cela deux choix :

- soit on considère qu'on ajoute après, auquel cas on dispose du precédent
- soit on considère qu'on ajoute avant/suprprime et pour cela on effectue une boucle pour déterminer le maillon précédent.

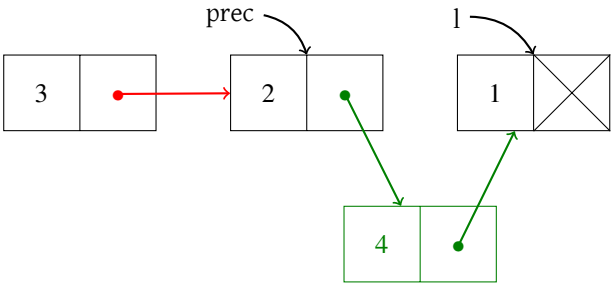
```c
void liste_ajout_apres(liste l, maillon *m, int x)
{
```

```c
    m->suivant = liste_ajout_en_tete(m->suivant, x);
}

void liste_ajout_avant(liste l, maillon *m, int x)
{
    liste prec = NULL;

    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
    prec->suivant = liste_ajout_en_tete(m, x);
}
```

Si on reprend la liste `l` contenant `3,2,1` et qu'on souhaite insérer la valeur `4` avant la valeur `1`, on va passer un pointeur `m` vers le maillon contenant `1` et effectuer un parcours jusqu'à avoir `prec` et `l` dans la configuration :



On dispose alors des pointeurs permettant de réaliser l'ajout :




```c
void liste_suppr_non_en_tete(liste l, maillon *m)
{
    liste prec = NULL;

    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
    prec->suivant = liste_suppr_en_tete(m);
}
```

En fait, lorsqu'on regarde le parcours précédent, on remarque deux points :

- comme on a donné le maillon concerné, le parcours a uniquement pour but de repérer le maillon qui le précède ;
- on pourrait se contenter d'utiliser uniquement un pointeur sur le maillon précédent lors du parcours car le maillon qui le suit est accessible avec `->suivant`.

III.1.viii Autres implémentations

Une autre implémentation standard consiste à *cacher* le pointeur sur le premier maillon, ce qui permet de donner également un pointeur sur le dernier maillon. En effet, le pointeur sur le dernier maillon permet de réaliser un ajout en fin de liste en $O(1)$ car il n'y a pas besoin de parcourir la liste pour faire cet ajout.

On obtient alors un type comme :

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

```c
struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Cette implémentation est l'occasion de se poser la question sur la répartition entre pile et tas pour les données. On peut légitimement penser que les listes ici ne sont que des couples de pointeurs, et qu'ainsi les passer par copie est léger comparativement à la complexité induite par une allocation sur le tas.

III.2 En OCaml

III.2.i Cas des 'a list

Tout d'abord, il faut se rendre compte que le type par défaut

```
type 'a list = (::) of 'a * 'list | []
```

fait intervenir des maillons et des pointeurs. La différence principale avec C est qu'on ne peut pas changer la valeur des pointeurs.

III.2.ii Type des listes chaînées

Pour retrouver la richesse du type précédent, on peut définir un type comme :

OCaml
```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a liste
} and 'a liste = Vide | Lien of 'a maillon
```

Ici, le type est une traduction directe du type précédent. On remarque que les types sont mutuellement récursifs car un maillon contient une liste. Le type somme `'a liste` ressemble fortement à un pointeur qui peut être nul ou pointer sur un maillon. On remarque qu'on aurait pu aussi se contenter d'écrire le type suivant :

OCaml
```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a maillon option
}

type 'a liste = 'a maillon option
```

Mais on va préférer le premier type qui a l'avantage de permettre de bien faire apparaitre la structure.

III.2.iii Ajout et suppression en tête

Pour rajouter un maillon en tête, on peut écrire :

OCaml
```
let cons x l = Lien { valeur = x; suivant = l}
```

On remarque que la fonction est beaucoup plus simple que celle en C car l'allocation est automatique et l'initialisation se fait naturellement dans la syntaxe. Cette fonction renvoie une nouvelle liste, on aurait pu aussi rajouter une référence pour les listes afin de permettre de les rendre modifiables, c'est la même discussion que dans la partie précédente.

La suppression en tête, cela revient à renvoyer la queue de la liste comme le fait `List.tl` :

OCaml
```
let tl l  = match l with
    | Vide -> failwith "Liste vide"
    | Lien { valeur = _; suivant = q} -> q
```

On remarque le filtrage imbriqué `| Lien { valeur = t; suivant = q }` qui correspond au filtrage `| t :: q` du type `a list`.

III.2.iv Exemple de parcours sans modification : calcul de la longueur

Vu les remarques précédentes, il n'est pas étonnant que le parcours d'une liste de manière récursive soit très proche de ce qu'on a déjà pu voir avec les listes de base.

```OCaml
let rec longueur l = match l with
    | Vide -> 0
    | Lien { valeur = _ ; suivant = q } -> 1 + longueur q
```

On en déduit de même une fonction renvoyant un maillon par son indice :

```OCaml
let rec nieme l n = match l with
    | Vide -> failwith "Liste vide"
    | Lien m -> if n = 0 then m else nieme m.suivant (n-1)
```

Remarque Il est possible de réecrire la fonction précédente en permettant à la fois de donner un nom, ici `m`, au maillon et de faire un filtrage sur ce qu'il contient. Pour cela, on utilise le mot clé `as` en `OCaml` :

```OCaml
let rec nieme l n = match l with
    | Vide -> failwith "Liste vide"
    | Lien ({ valeur=_ ; suivant = q} as m) ->
          if n = 0 then m else nieme q (n-1)
```

III.2.v Exemple de parcours avec modification : ajout d'une maillon en fin de liste

On va montrer un exemple de modification de liste en rajoutant un maillon en fin d'une liste non vide. Ici, on effectue un parcours jusqu'à tomber sur le dernier maillon auquel on rajoute le nouveau à la suite.

```OCaml
let rec ajout_fin l x =
    match l with
    | Vide -> failwith "Liste vide"
    | Lien m ->
       if m.suivant = Vide
       then m.suivant <- Lien { valeur = x ; suivant = Vide }
       else ajout_fin m.suivant x
```

III.2.vi Raffinement pour permettre l'ajout à la fin en temps constant

Le programme précédent est à comparer au programme suivant :

```OCaml
let rec ajout_fin l x =
    match l with
    | [] -> [x]
    | t::q -> t : : ajout_fin q x
```

On n'a pas l'impression d'avoir vraiment gagné en expressivité ou en efficacité.

Cependant, on peut se dire qu'en changeant le type `'a liste` on peut tirer partie des maillons pour obtenir un ajout en fin de liste en temps constant. Le programme suivant présente une interface permettant de le faire.

```OCaml
type 'a maillon = {
    mutable valeur : 'a ;
    mutable suivant : 'a maillon_ptr
  }
and 'a maillon_ptr = 'a maillon option
and 'a liste = {
      mutable premier : 'a maillon_ptr ;
      mutable dernier : 'a maillon_ptr
  }

let liste_vide () = { premier = None ; dernier = None }

let ajout_debut l x =
    l.premier <- Some { valeur = x ; suivant = l.premier } ;
    if l.dernier = None
    then l.dernier <- l.premier

let suppr_debut l =
    match l.premier with
    | None -> failwith "Liste vide"
    | Some { valeur = _ ; suivant = l' } ->
          l.premier <- l'
```

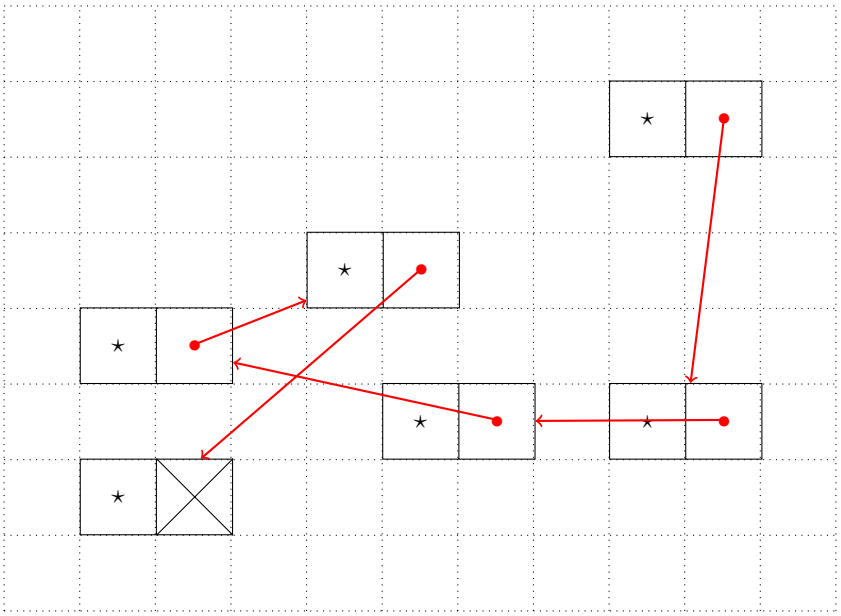
```
let ajout_fin l x =
    let m = { valeur = x; suivant = None } in
    (match l.dernier with
    | Some m' -> m'.suivant <- Some m
    | None -> () );
    l.dernier <- Some m;
    if l.premier = None
    then l.premier <- l.dernier
```

Quelques remarques sur ce programme :

- comme on change directement `premier` et `dernier`, il est nécessaire de générer une nouvelle liste vide, ce qui est assuré ici par le paramètre `()` ;
- les pointeurs sont représentés par des options dans des champs mutables ;
- afin de préserver l'intégrité des deux pointeurs, on est obligé de gérer les cas où il n'y a qu'un seul maillon ;
- attention à la priorité du cas de filtrage sur `;` qui oblige à mettre des parenthèses dans `ajout_fin`.

III.3 Structure de la mémoire

En mémoire, les maillons d'une liste chaînée, comme celle vue en C, sont sur le tas et de manière désorganisée. Cela signifie qu'il n'y a aucune raison que deux maillons proches dans une liste soient proches en mémoire.



Or, les processeurs optimisent la gestion de la mémoire à l'aide d'un cache qui, au lieu de n'accéder qu'à une seule valeur située à une adresse, va charger une zone mémoire autour de cette adresse. Ceci encourage une cohérence spatiale dans l'organisation de la mémoire afin de profiter au maximum de cette mise en cache.

Une stratégie pour réaliser des listes chaînées efficacement peut consiste à allouer un tableau de maillons et à gérer ensuite les allocations parmi cette réserve.

IV Travaux pratiques

On présente ici deux énoncés de travaux pratiques en C en lien avec ce chapitre.

IV.1 Tableaux non statiques et tableaux dynamiques

On va définir ici une structure de donnée pour gérer des tableaux dont la taille ne sera connu qu'à l'exécution. Pour simplifier, ce seront des tableaux d'entiers `int` mais la méthode s'adapte naturellement pour des tableaux de n'importe quoi.

IV.1.i Type `array`

On définit le type suivant :

```c
struct array {
    int *elements;
    unsigned int size;
};
typedef struct array array;
```

Un `array` est donc une structure qui contient :

- un pointeur vers un tableau `elements` d'entiers
- un entier indiquant la taille de ce tableau

Remarque importante quand on copie un pointeur, on copie uniquement un entier qui est l'adresse pointée. Ainsi, quand on copie un `array`, ce n'est finalement qu'un couple d'entiers qu'on copie. C'est pourquoi, dans la suite, on passe tous les `array` par valeur. C'est-à-dire que pour définir une fonction de recherche d'un élément dans un `array` on va écrire :

```c
int search(array t, int x)
{
    for(int i = 0; i < t.size; i++)
    {
        if(t.elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Quand on va appeler la fonction, on va avoir une copie de l'argument, c'est-à-dire du `array`, mais le tableau `elements` lui sera toujours à la même place. L'alternative serait de passer les valeurs `array` par pointeurs comme dans la variante suivante :

```c
int search(array *t, int x)
{
    for(int i = 0; i < t->size; i++)
    {
        if(t->elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

On n'a rien à gagner au surplus de complexité induit par cela. L'unique avantage serait de permettre de modifier les paramètres de la structure passée en argument, mais on réservera ça aux fonctions qui le nécessitent.

IV.1.ii Allocation, Initialisation, Libération

Pour les fonctions suivantes, on fera usage de `malloc` et `free`.

Question IV.1 Écrire une fonction de prototype `array array_alloc(unsigned int size)` qui alloue un tableau de `size` entiers avec `malloc` et renvoie le `array` pointant dessus.

Conformément à ce qui est écrit au-dessus, on peut créer un `array` en variable locale et le renvoyer avec `return`, ce qui compte c'est la mémoire pointée.

■ **Preuve**

```c
array array_alloc(unsigned int size)
{
    array a;
    a.elements = (int *)malloc(sizeof(int) * size);
    a.size = size;
    return a;
}
```

■

Question IV.2 Écrire une fonction de prototype `void array_free(array t)` qui libère le tableau pointé par `t` avec `free`. A partir de ce moment, on ne peut plus utiliser cette adresse sans faire d'erreurs.

■ **Preuve**

```c
void array_free(array a)
{
    free(a.elements);
}
```

■

Question IV.3 Le tableau alloué n'est pas initialisé, écrire une fonction de prototype `array array_make(unsigned int size, int def)` qui alloue un tableau et initialise toutes les valeurs de celui-ci à `def`.

■ **Preuve**

```c
array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    return a;
}
```

■

Question IV.4 Il est possible de mesurer le temps pris par un programme à l'aide de la fonction `time` :

```
$> time ./main
./main  0.38s user 0.00s system 99% cpu 0.379 total
```

Comparer le temps d'exécution d'un programme qui
- alloue puis libère 100 fois un tableau d'un million d'entiers
- alloue *en initialisant à 0* avec `make` puis libère 100 fois un tableau d'un million d'entiers.

Que peut-on en conclure ?

■ **Preuve**

On se rend compte que l'allocation est presque instantanée alors que l'initilisation prend un temps linéaire en la taille des données. Ceci est cohérent avec le fait que la mémoire est allouée de manière paresseuse. On s'en rend d'autant plus compte en examinant l'empreinte mémoire des programmes.

■

Remarque Il existe d'autres fonctions que `malloc` comme

```c
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

qui permettent :
- pour `calloc` d'allouer un tableau de `nmemb` membres qui sont chacun de taille `size` et **d'initialiser les octets à 0**. Cette fonction prend donc un temps linéaire en le nombre d'octets.
- pour `realloc` de déplacer en mémoire un tableau en changeant sa taille (en plus ou en moins). Si jamais il y a de la place on ne bouge pas le tableau. C'est donc forcément plus efficace que de le faire à la main.

Ces deux fonctions sont **hors programme !**

IV.1.iii Affichage, copie

Question IV.5 Écrire une fonction `void array_print(array a)` qui affiche `[a1,a2,...,an]` si `a` contient les éléments de `a1` à `an`.

Remarque cette fonction vous sera utile pour tester ce que vous avez fait dans la suite.

■ **Preuve**

```c
void array_print(array a)
{
    printf("[");
    for(int i = 0; i < a.size; i++)
    {
        printf("%d", a.elements[i]);
        if(i < a.size - 1)
            printf(",");
    }
    printf("]\n");
}
```

■

Question IV.6 Écrire une fonction de prototype

```c
void array_blit(array src, int src_start,
    array dst, int dst_start, int len)
```

qui copie `len` éléments depuis le tableau `src` à partir de l'indice `src_start` dans le tableau `dst` à partir de l'indice `dst_start`.

Remarque on a déjà écrit cette fonction en `OCaml`, vous pouvez vous en inspirer.

■ **Preuve**

```c
void array_blit(array src, int src_start,
            array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
}
```

■

IV.1.iv Implémentation d'une pile non bornée

Question IV.7 Écrire une fonction de prototype `void array_push(array *a, int x)` qui rajoute `x` à la fin du array `a`, c'est-à-dire :

- on va allouer un nouveau `array b` dont le tableau contient `a->size+1` éléments
- on va copier les anciennes valeurs de `*a` dans `b`
- écrire `x`
- libérer `*a`
- remplacer avec `*a = b`.

Remarque On peut tout à faire écrire `*a = b` pour que `a` ait les mêmes valeurs que `b` en dehors de la fonction;

Rappel pour une `struct` par pointeurs, on utilise `->` pour accéder aux champs. Donc `a->elements` et `a->size` ici.

Remarque Faites des tests comme en créant un tableau de taille 0 et en empilant les entiers de 1 à 10 et en affichant le résultat. Faites les tests avec `-fsanitize=address`.

■ **Preuve**

```c
void array_push(array *a, int x)
{
    array b = array_alloc(a->size+1);
    array_blit(*a, 0, b, 0, a->size);
```

```
    b.elements[a->size] = x;
    array_free(*a);
    *a = b;
}

// Note qu'on aurait pu proposer une interface persistante pour push/pop ainsi
// pas de pointeurs mais on renvoie le nouveau array
array array_push_2(array a, int x)
{
    array b = array_alloc(a.size+1);
    array_blit(a, 0, b, 0, a.size);
    b.elements[a.size] = x;
    // pas de raison de faire de free ici de a, c'est
    // a l'appelant de gérer.
    return b;
}
```

■

Question IV.8 Écrire une fonction `int array_pop(array *a)` qui retire le dernier élément du tableau `a` et renvoie sa valeur. Il faudra donc :

- allouer un nouveau `array b` dont le tableau contient `a->size-1` éléments
- copier les anciennes valeurs sauf une
- récupèrer dans une variable `x` la valeur de la dernière case de `a` (*parce qu'il n'est pas possible d'y accéder après l'étape suivante*)
- libérer l'ancien tableau
- remplacer avec `*a = b`.

■ **Preuve**

```c
int array_pop(array *a)
{
    array b = array_alloc(a->size-1);
    array_blit(*a, 0, b, 0, a->size-1);
    int x = a->elements[a->size-1];
    array_free(*a);
    *a = b;
    return x;
}
```

■

IV.1.v Tableaux dynamiques

Reprendre toutes les questions précédentes en changeant la structure de donnée pour permettre de faire des tableaux dynamiques. Il faudra ainsi avoir deux entiers : `int p_size` qui contiendra la taille *physique* en mémoire et `int l_size` qui contiendra la taille *logique* c'est-à-dire les éléments signifiants stockés.

Quand on a besoin de reallouer la mémoire, on choisira de doubler la taille physique. Le pop ne libérera jamais de taille physique.

■ **Preuve**

```c
#include <stdio.h>
#include <stdlib.h>

struct array {
    int *elements;
    unsigned int p_size;
    unsigned int l_size;
};

typedef struct array array;

array array_alloc(unsigned int size)
{
    array a;
    a.elements = (int *)malloc(sizeof(int) * size);
```

```
    a.p_size = size;
    a.l_size = 0;
    return a;
}

void array_free(array a)
{
    free(a.elements);
}

array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    a.l_size = size;
    return a;
}

void array_print(array a)
{
    printf("[");
    for(int i = 0; i < a.l_size; i++)
    {
        printf("%d", a.elements[i]);
        if(i < a.l_size - 1)
            printf(",");
    }
    printf("]\n");
}

void array_blit(array src, int src_start,
            array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
}

void array_push(array *a, int x)
{
    if (a->l_size < a->p_size)
    {
        a->elements[a->l_size] = x;
        a->l_size += 1;
    }
    else
    {
        array b = array_alloc(2*a->l_size);
        array_blit(*a, 0, b, 0, a->l_size);
        b.elements[a->l_size] = x;
        b.l_size = a->l_size + 1;
        array_free(*a);
        *a = b;
    }
}

int array_pop(array *a)
{
    a->l_size -= 1;
    return a->elements[a->l_size];
}


int main(void)
{
    array a = array_alloc(0);
    for(int i = 0; i < 10; i++)
```

```c
        array_push(&a, i);
    for(int i = 0; i < 3; i++)
        printf("Pop %d\n", array_pop(&a));
    array_print(a);
    array_free(a);
}
```

■

IV.2 Listes chaînées

IV.2.i Listes simplement chaînées

On va définir des fonctions sur les listes chaînées en utilisant le type suivant :

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

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Remarque **Invariants à maintenir**

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Si la liste est vide alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Question IV.9 Écrire une fonction de prototype

```c
liste liste_vide();
```

qui renvoie la liste vide. Ici, on pourrait dire **une** liste vide mais cela correspond à une unique valeur de la structure `liste`.

■ **Preuve**

```c
liste liste_vide()
{
    liste l;
    l.premier = NULL;
    l.dernier = NULL;
    return l;
}
```

■

Question IV.10 Écrire une fonction de prototype

```c
void liste_affiche(liste l);
```

qui affiche le contenu de la liste sous la forme `[1,2,3]`.

■ **Preuve**

```c
void liste_affiche(liste l)
{
    putchar('[');
    maillon *m = l.premier;
    while(m != NULL)
    {
```

```c
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar('\n');
}
```

■

Question IV.11 Écrire une fonction de prototype

```c
maillon *maillon_creer(int valeur, maillon *suivant);
```

qui alloue et initialise un nouveau maillon.

■ **Preuve**

```c
maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}
```

■

Question IV.12 Écrire une fonction de prototype

```c
void maillon_detruire(maillon *m);
```

qui détruit, c'est-à-dire libère, la mémoire associée à un maillon.
Écrire une fonction de prototype

```c
void chaine_detruire(maillon *m);
```

qui détruit la chaîne de maillon accessible depuis le maillon `m`.
En déduire une fonction de prototype

```c
void liste_detruire(liste l);
```

qui détruit la chaîne pointée par une liste.

■ **Preuve**

```c
void maillon_detruire(maillon *m)
{
    free(m);
}

void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant;
        maillon_detruire(m);
        m = suivant;
    }
}

void liste_detruire(liste l)
{
    chaine_detruire(l.premier);
}
```

■

Question IV.13 Écrire une fonction de prototype

```c
int liste_longueur(liste l);
```

qui renvoie la longueur de la liste `l`.

■ **Preuve**

```c
int liste_longueur(liste l)
{
    int longueur = 0;
    maillon *m = l.premier;
    while(m != NULL)
    {
        longueur += 1;
        m = m->suivant;
    }
    return longueur;
}
```

■

Question IV.14 Écrire des fonctions de prototype

```c
int liste_tete(liste l);
liste liste_queue(liste l);
```

qui renvoient respectivement la tête et la queue d'une liste. **Attention**, ici, contrairement au type vu plus haut, le maillon suivant n'est pas une liste.

■ **Preuve**

```c
int liste_tete(liste l)
{
    assert(l.premier != NULL);
    return l.premier->valeur;
}

liste liste_queue(liste l)
{
    assert(l.premier != NULL);
    liste q;
    q.premier = l.premier->suivant;
    q.dernier = l.dernier;
    return q;
}
```

■

Question IV.15 Écrire une fonction de prototype

```c
void liste_ajout_en_tete(liste *l, int valeur);
```

qui ajoute un maillon en tête de la liste pointée par `l` en $O(1)$. On fera attention au cas où `l` pointe la liste vide.

■ **Note 1** Dans cette fonction et les suivantes, on a fait le choix de passer des pointeurs sur des listes pour permettre de modifier les valerus des pointeurs `premier` et `dernier`. Alternativement, on aurait pu faire en sorte que ces fonctions renvoient des `liste` par copie.

■

■ **Preuve**

```c
void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, l->premier);
```

```c
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}
```

■

Question IV.16 Écrire une fonction de prototype

```c
void liste_suppr_en_tete(liste *l);
```

qui supprime un maillon en tête de la liste `l` supposée non vide en $O(1)$.
On fera attention au cas où on supprime l'unique maillon de la liste.

■ **Preuve**

```c
void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    maillon_detruire(m); // on libère la mémoire
    if(l->premier == NULL) // on a vidé la liste
        l->dernier = NULL;
}
```

■

Question IV.17 Écrire une fonction de prototype

```c
void liste_ajout_en_fin(liste *l, int valeur);
```

qui ajoute un maillon en fin de la liste `l` en $O(1)$.

■ **Preuve**

```c
void liste_ajout_en_fin(liste *l, int valeur)
{
    maillon *m = l->dernier;
    l->dernier = maillon_creer(valeur, NULL);
    m->suivant = l->dernier;
    if(l->premier == NULL) // la liste était vide
        l->premier = l->dernier;
}
```

■

Question IV.18 Écrire une fonction de prototype

```c
maillon *liste_cherche_maillon(liste l, int valeur);
```

qui renvoie un pointeur sur le premier maillon de valeur `valeur` ou renvoie `NULL` s'il n'y en a pas.

■ **Preuve**

```c
maillon *liste_cherche_maillon(liste l, int valeur)
{
    maillon *m = l.premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

■

Question IV.19 Écrire une fonction de prototype

```c
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);
```

qui insère en $O(1)$ un maillon après le maillon `m`. On fera en sorte que l'insertion soit valide même si `m` est le dernier maillon de la liste.

■ **Preuve**

```c
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = maillon_creer(valeur, m->suivant);
}
```

■

Question IV.20 Écrire une fonction de prototype

```c

```

qui ajoute un maillon avant `m`. Dans le cas où `m` est le premier ou le dernier maillon, l'insertion sera en $O(1)$.

■ **Preuve**

```c
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)
{
    if (m == l->premier)
        liste_ajout_en_tete(l, valeur);
    else if (m == l->dernier)
        liste_ajout_en_fin(l, valeur);
    else
    {
        maillon *prec = l->premier;
        while(prec->suivant != m)
        {
            prec = prec->suivant;
        }
        prec->suivant = maillon_creer(valeur, m);
    }
}
```

■

Question IV.21 Écrire une fonction de prototype

```c
void liste_suppr_maillon(liste *l, maillon *m);
```

qui supprime un maillon. Dans le cas où `m` est le premier maillon, la suppression sera en $O(1)$.

■ **Preuve**

```c
void liste_suppr_maillon(liste *l, maillon *m)
{
    if (m == l->premier)
        liste_suppr_en_tete(l);
    else
    {
        maillon *prec = l->premier;
        while(prec->suivant != m)
        {
            prec = prec->suivant;
        }
        prec->suivant = m->suivant;
        maillon_detruire(m);
    }
}
```

■

IV.2.ii Listes doublement chaînées

Pour les listes doublement chaînées, on va adapter le type précédent en rajoutant juste un pointeur `precedent` dans les maillons :

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

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Remarque **Invariants à maintenir**

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Dans un maillon, `precedent` est soit `NULL` si c'est le premier maillon de la chaîne, soit un pointeur vers le maillon qui le précède.
- Si la liste est vide alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Question IV.22 Reprendre les questions précédentes avec ce nouveau type.

■ **Preuve**

Peu de modifications à faire.

```c
liste liste_vide()
{
    liste l;
    l.premier = NULL;
    l.dernier = NULL;
    return l;
}

void liste_affiche(liste l)
{
    putchar('[');
    maillon *m = l.premier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar('\n');
}

maillon *maillon_creer(int valeur,
        maillon *precedent, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    m->precedent = precedent;
    return m;
}

void maillon_detruire(maillon *m)
{
```

```
    free(m);
}

void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant;
        maillon_detruire(m);
        m = suivant;
    }
}

void liste_detruire(liste l)
{
    chaine_detruire(l.premier);
}

int liste_longueur(liste l)
{
    int longueur = 0;
    maillon *m = l.premier;
    while(m != NULL)
    {
        longueur += 1;
        m = m->suivant;
    }
    return longueur;
}

int liste_tete(liste l)
{
    assert(l.premier != NULL);
    return l.premier->valeur;
}

liste liste_queue(liste l)
{
    assert(l.premier != NULL);
    liste q;
    q.premier = l.premier->suivant;
    q.dernier = l.dernier;
    return q;
}

void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, NULL, l->premier);
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}

void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    l->premier->precedent = NULL;
    maillon_detruire(m); // on libère la mémoire
    if(l->premier == NULL) // on a vidé la liste
        l->dernier = NULL;
}

void liste_ajout_en_fin(liste *l, int valeur)
{
    maillon *m = l->dernier;
    l->dernier = maillon_creer(valeur, m, NULL);
    m->suivant = l->dernier;
    if(l->premier == NULL) // la liste était vide
        l->premier = l->dernier;
```

```c
}

maillon *liste_cherche_maillon(liste l, int valeur)
{
    maillon *m = l.premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}

void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = maillon_creer(valeur, m, m->suivant);
}
```

■

Question IV.23 Écrire une fonction de prototype

```c
void liste_suppr_en_fin(liste *l);
```

qui supprime en $O(1)$ le dernier maillon.

■ **Preuve**

```c
void liste_suppr_en_fin(liste *l);
{
    maillon *fin = l->dernier;
    l->dernier = fin->precedent;
    l->dernier->suivant = NULL;
    if (l->dernier == NULL)
        l->premier = NULL;
    maillon_detruire(fin);
}
```

■

Question IV.24 Écrire une fonction de prototype

```c
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur);
```

qui ajoute un maillon avant `m` en $O(1)$.

■ **Preuve**

```c
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)
{
    if (m == l->premier)
        liste_ajout_en_tete(l, valeur);
    else if (m == l->dernier)
        liste_ajout_en_fin(l, valeur);
    else
    {
        // En O(1)
        m->precedent->suivant = maillon_creer(valeur,
                m->precedent, m);
        m->precedent = m->precedent->suivant;
    }
}
```

■

Question IV.25 Écrire une fonction de prototype

```c
void liste_suppr_maillon(liste *l, maillon *m);
```

qui supprime en $O(1)$ le maillon `m`.

■ **Preuve**

```c
void list_suppr_maillon(liste *l, maillon *m)
{
    if (m == l->premier)
        liste_suppr_en_tete(l);
    else
    {
        m->precedent->suivant = m->suivant;
        m->suivant->precedent = m->precedent;
        maillon_detruire(m);
    }
}
```

■