Informatique *Cours*
Structures de données - Pt 1 Ipesup

Objectifs

À l'issue de cette leçon, l'étudiant doit être capable de :
- comprendre le principe des structures de données séquentielles et leurs invariants ;
- implémenter des piles et des files, mutables et immuables, en OCaml et en C ;
- analyser le coût des opérations classiques (`push`, `pop`, `enqueue`, `dequeue`) et raisonner en complexité amortie ;
- expliquer le fonctionnement des tables de hachage et le rôle des fonctions de hachage ;
- comparer différentes implémentations (listes chaînées, tableaux, tableaux redimensionnables) en termes de performances et d'occupation mémoire.

Les structures de données jouent un rôle essentiel en informatique. Elles permettent d'organiser l'information pour la traiter ensuite efficacement. Pour une même information, de nombreuses structures de données peuvent être utilisées. Le choix dépend des opérations que l'on souhaitera effectuer et de leur efficacité.

Types et abstraction

Prenons l'exemple d'une structure de données dans laquelle on veut stocker tous les mots apparaissant dans l'œuvre de Dante Alighieri *La Divine Comédie* avec, pour chacun, son nombre d'occurrences dans le texte. Il faut se donner les moyens de construire une telle structure de données, ainsi que les moyens de la consulter par la suite. Une solution à ce problème consiste à utiliser une structure de **tableau associatif**.

Il s'agit d'une structure qui associe à des clés d'un certain type, ici les chaînes de caractères qui sont les mots de notre texte, des valeurs d'un autre type, ici des entiers qui sont les nombres d'occurrences. Une structure de tableau associatif permet notamment

- de construire un nouveau tableau associatif, vide de toute entrée ;
- d'ajouter une nouvelle entrée, pour une certaine clé et une certaine valeur ;
- d'obtenir la valeur associée à une clé donnée, le cas échéant.

Bien entendu, on pourrait imaginer d'autres opérations, comme par exemple obtenir le nombre d'entrées ou encore effacer toutes les entrées. Mais cela nous suffit pour l'instant. En particulier, avec ces trois opérations seulement, on peut lire le texte de l'œuvre de Dante Alighieri et remplir un tableau associatif avec les nombres d'occurrences. Pour chaque mot w du texte, on récupère le nombre d'occurrences actuel (zéro s'il n'y en a pas encore), puis on enregistre dans le tableau associatif que le mot w est maintenant associé à la valeur $n+1$.

Programme - interface d'un tableau associatif

Interface d'une structure de tableau associatif où les clés sont des chaînes de caractères et les valeurs associées des entiers.

— En OCaml, dans un fichier `.mli` :

```
type table
val create : unit -> table
val put : table -> string -> int -> unit
val get : table -> string -> int
```

— En C, dans un fichier `.h` :

```
typedef struct Table table;
table *table_create(void);
void table_put(table *t, char *k, int v);
int table_get(table *t, char *k);
void table_delete(table *t);
```

On parle également de **dictionnaire** pour une telle structure de données.

Qu'il s'agisse d'OCaml ou de C, on a délibérément abstrait la structure de données, en ne révélant pas sa représentation. En OCaml, on ne donne pas la définition du type `table` ; en C, on ne donne pas la déclaration de la structure `Table`. On parle de *type abstrait de données*. Pour autant, le compilateur OCaml ou C est tout à fait en mesure de compiler un code client qui utilise notre tableau associatif, par exemple le programme qui ouvre le fichier contenant le texte et remplit le tableau associatif avec les nombres d'occurrences. L'interface contient toute l'information, et seulement l'information, pour que le compilateur soit capable de compiler le code.

Bien evidemment, pour obtenir un programme exécutable, il faudra aprés également fournir une implémentation de notre tableau associatif.

Plus loin, nous verrons au moins trois structures de données différentes permettant de réaliser un tel tableau associatif efficacement : une table de hachage, un arbre binaire de recherche et un arbre préfixe.

Ne pas révéler la représentation de notre tableau associatif est un concept fondamental en informatique, désigné par le terme de *barrière d'abstraction*. La figure suivante illustre ce concept, avec à gauche l'implémentation de la structure de données et à droite le code client qui l'utilise, l'interface se situant entre les deux.

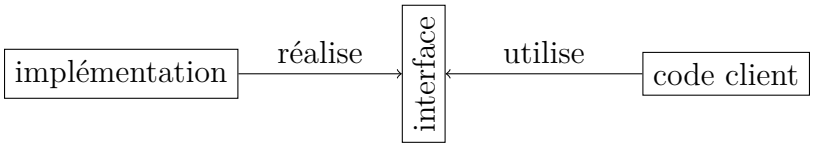


FIGURE 1 – La barrière d'abstraction.

Le compilateur vérifie d'une part le bon usage de l'interface par le code client et d'autre part la réalisation de l'interface par l'implémentation de la structure de données.

La barrière d'abstraction a de multiples avantages. En premier lieu, le programme est clairement séparé en deux composantes (voire plus), qui peuvent être développées indépendam-

ment, par exemple par deux personnes différentes, à partir du moment où elles s'entendent sur une interface. C'est notamment ainsi que l'on peut proposer des bibliothèques de programmes facilement réutilisables.

La barrière d'abstraction permet également de remplacer facilement une implémentation de la structure de données par une autre, idéalement plus efficace. En effet, si l'interface ne dévoile pas la représentation, comme dans notre exemple, alors le code client ne sera pas impacté par un changement de l'implémentation. Il suffit que le type s'appelle toujours `table` et que les trois opérations aient toujours les mêmes noms, les mêmes types et les mêmes comportements.

Enfin, la barrière d'abstraction permet de *encapsuler* les détails de représentation, qui sont propres à la structure de données, et notamment ses invariants internes. Par exemple, la structure de données pourrait reposer sur un tableau qui est maintenu trié en permanence. Si le tableau sous-jacent était révélé, alors il pourrait être modifié par le code client et l'invariant, sur lequel repose la correction des opérations, pourrait être cassé. La barrière d'abstraction permet des programmes plus sûrs.

L'interface donnée dans le programme offre une barrière d'abstraction qui suffit au compilateur, mais elle n'en est pas parfaite pour autant.
Ainsi, l'interface ne dit pas explicitement si la structure de données derrière le type `table` est *mutable*, c'est-à-dire qu'elle est modifiée en place par les opérations, ou au contraire *immutable*, c'est-à-dire qu'une opération ne modifie pas la structure.
Bien entendu, le fait que la fonction `put` ne renvoie pas de valeur est un indice flagrant qu'il s'agit là d'une structure mutable, mais ce n'est pas toujours aussi simple. On pourrait ainsi imaginer une opération qui renvoie une table mais qui pour autant modifie la table. De même, on ne peut pas savoir à la lecture de cette interface si la fonction `get` modifie ou non la table.
Les types OCaml et C donnés aux opérations suffisent au compilateur pour vérifier le bon usage au regard du système de types, mais ils ne vont pas plus loin et ne capturent notamment pas des propriétés comme avoir un effet de bord. Dans la même idée, l'interface ne dit pas quel est le comportement de la fonction `get` lorsque la clé ne se trouve pas dans la table. En OCaml, par exemple, il est idiomatique que la fonction `get` lève l'exception `Not_found` pour signaler une clé absente. Pour autant, le type de `get` ne le montre pas. En C, on peut choisir que la fonction `table_get` renvoie une valeur particulière lorsque la clé est absente, comme par exemple -1. Mais il serait tout également possible de supposer un comportement non défini en cas d'accès à une clé qui n'est pas dans le tableau associatif[1]. Là encore, le type de `table_get` ne l'indiquerait pas. C'est pourquoi une interface contient typiquement une *documentation*, sous forme de commentaires, qui accompagne les déclarations du langage pour en préciser les comportements.

L'interface peut également échouer à offrir une vraie barrière d'abstraction.
On peut imaginer qu'une chaîne de caractères C, passée en argument à `table_put`, est ensuite modifiée par le code client. Dès lors, allons-nous retrouver la valeur associée à la chaîne originale, comme on pourrait l'espérer, ou bien plutôt une valeur associée à la nouvelle chaîne, par un effet de bord ? Ou allons-nous obtenir une erreur car un invariant interne de la structure a été brisé ? Le comportement va dépendre de l'implémentation. On peut en particulier imaginer une implémentation défensive qui va prudemment copier les chaînes passées en argument à `table_put`, même si cela coûte un peu cher, afin de se prémunir contre toute modification ultérieure. Mais d'autres implémentations pourraient ne pas prendre cette peine, exposant le code client à des désagréments. De manière générale, le partage d'une donnée

1. Il serait alors pertinent que l'interface fournisse aussi une fonction `table_contains` pour tester la présence d'une clé dans le tableau associatif.

mutable (ici une chaîne de caractères C) entre deux morceaux de code (ici la structure de données et le code client) s'appelle un *alias*. Maîtriser les alias dans un programme impératif est extrêmement difficile.

Ajoutons enfin que l'abstraction en C reste un vœu pieux. Même si l'interface n'expose pas la définition de la structure `Table`, le code client peut tout à fait lui en donner une. Si elle est différente de la définition utilisée dans l'implémentation, le pire peut arriver. Et si elle est identique, alors le code client a maintenant accès à la représentation interne de la structure et peut la modifier à loisir, en mettant à mal ses invariants. Le code client peut également contourner la barrière d'abstraction en déclarant l'existence d'une fonction qui n'est pas révélée par l'interface. Pour autant, l'utilisation d'une interface en C reste une bonne pratique. À la différence de C, la barrière d'abstraction d'OCaml interdit bien de donner une définition à un type abstrait ou de déclarer l'existence d'une fonction cachée.

Invariant de structure

De la même façon qu'un invariant de boucle décrit une propriété maintenue par chaque itération d'une boucle, un *invariant de structure* décrit une propriété d'une structure de données qui est établie à la création de la structure (comme la fonction `create`) et qui est maintenue par chaque opération de cette structure (comme les fonctions `put` et `get`). La barrière d'abstraction permet alors de garantir que, quel que soit l'enchaînement des opérations, toute instance de la structure de données a son invariant de structure établi. Du côté de l'implémentation, chaque opération peut faire l'hypothèse que l'invariant est établi en entrée et s'engage en retour à le garantir en sortie. Entre les deux, l'invariant peut être temporairement rompu. Les opérations qui ne sont pas exportées dans l'interface peuvent en revanche manipuler des états de la structure qui ne respectent pas l'invariant.

Structures de données séquentielles

Plusieurs structures peuvent être construites construites à partir de *tableaux* et de *listes*, deux structures fondamentales où les éléments sont naturellement *ordonnés*.

1 Tableaux

Le *tableau* est la structure de données la plus simple et la plus efficace.

Definition

Un tableau est une séquence de n valeurs, consécutives en mémoire, auxquelles on accède avec un indice entier entre 0 et $n-1$.

La propriété fondamentale d'un tableau est l'accès en temps constant à un élément, que ce soit en lecture ou en écriture.

En anglais, la mémoire vive est d'ailleurs désignée par l'acronyme RAM, pour *Random Access Memory*. Cela signifie très exactement que l'on peut accéder à n'importe quel élément (sous-entendu, directement), par opposition à un accès qui serait uniquement séquentiel.

En C. En C, un tableau peut être alloué sur la pile, dans une variable locale à la fonction, ou sur le tas avec `calloc`.
Un tableau alloué sur la pile n'est pas initialisé par défaut, mais un tableau alloué avec `calloc` est initialisé avec 0.
Si `a` est un tableau, on accède à la case i avec `a[i]` et on la modifie avec `a[i] = v`.
La taille d'un tableau n'est pas stockée en mémoire. Elle doit donc être passée en argument avec le tableau lorsqu'elle est nécessaire.

En OCaml. Un tableau OCaml est alloué sur le tas, avec `Array.make`.
Il est nécessairement initialisé, avec une valeur passée en argument à `Array.make`.
Si `a` est un tableau, on accède à la case i avec `a.(i)` et on la modifie avec `a.(i) <- v`.
La taille du tableau `a` s'obtient avec `Array.length a`, en temps constant.

1.1 Tableaux redimensionnables

Un tableau est une structure simple, compacte et efficace. Mais il est nécessaire d'en déterminer la taille au moment de sa création, ce qui peut être contraignant. Si par exemple on lit des données dans un fichier, une par ligne, pour les ranger dans un tableau, il n'est pas forcément facile d'en déterminer le nombre à l'avance. La lecture pourrait se faire, par exemple, sur l'entrée standard.

Le *tableau redimensionnable*(parfois également appelé *tableau dynamique*, ou `Vector` ou encore `ArrayList`) apporte une solution élégante à ce problème.

Comme avec un tableau traditionnel, on peut accéder en lecture et en écriture aux cases du tableau avec un indice entier, en temps constant. Mais à la différence d'un tableau traditionnel, on dispose d'une opération supplémentaire pour modifier la taille du tableau.

Tableaux redimensionnables en C

Programme - interface C d'un tableau redimensionnable

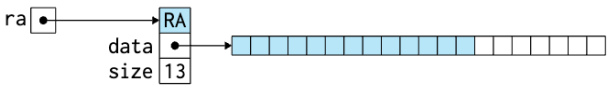
```
typedef struct Vector vector;
vector *vector_create(int capacity);
int vector_size(vector *v);
int vector_get(vector *v, int i);
void vector_set(vector *v, int i, int x);
void vector_resize(vector *v, int s);
void vector_delete(vector *v);
```

Le type `vector` est un raccourci pour la structure `Vector`, dont le contenu n'est pas révélé. La fonction `vector_create` permet de construire un nouveau tableau redimensionnable, d'une capacité donnée. La taille s'obtient avec la fonction `vector_size` et elle peut être modifiée avec la fonction `vector_resize`. Les fonctions `vector_get` et `vector_set` permettent l'accès et la modification d'une case du tableau. Enfin, la fonction `vector_delete` permet de désallouer la structure.

Principe de l'implémentation. L'implémentation d'un tableau redimensionnable repose sur l'utilisation, en interne, d'un tableau usuel. Les éléments du tableau redimensionnable sont stockés au début de ce tableau.

Lorsque la taille du tableau redimensionnable est modifiée, on remplace si besoin le tableau interne par un autre tableau, vers lequel on recopie tous les éléments. Il est fondamental de comprendre que ce changement est *transparent* pour l'utilisateur du tableau redimensionnable.

On illustre ici une variable `ra` qui contient un tableau redimensionnable dont le tableau interne a une capacité de 20 éléments, à l'intérieur duquel 13 éléments sont effectivement utilisés.



Le champ `size` maintient le nombre d'éléments du tableau redimensionnable, ici 13.
Le champ `data` contient un pointeur vers le tableau où les éléments sont stockés.
C'est cette *indirection* matérialisée par le champ `data` qui permet d'agrandir (resp. rétrécir) le tableau interne, en le remplaçant par un tableau plus grand (resp. plus petit), sans pour autant modifier la valeur de la variable `ra`.

Le programme suivante contient une implémentation C de tableau redimensionnable.

Programme - tableau redimensionnable en C

```
typedef struct Vector {
  int capacity;
  int *data;   // tableau de taille capacity
  int size;    // invariant 0 <= size <= capacity
} vector;
```

```
vector *vector_create(int capacity) {
  vector *v = malloc(sizeof(struct Vector));
  v->capacity = capacity;
  v->data = calloc(capacity, sizeof(int));
  v->size = 0;
  return v;
}

int vector_size(vector *v) {
  return v->size;
}

int vector_get(vector *v, int i) {
  assert(0 <= i && i < v->size);
  return v->data[i];
}

void vector_set(vector *v, int i, int x) {
  assert(0 <= i && i < v->size);
  v->data[i] = x;
}

void vector_resize(vector *v, int s) {
  assert(0 <= s);
  if (s > v->capacity) {
    v->capacity = 2 * v->capacity;
    if (v->capacity < s) v->capacity = s;
    int *old = v->data;
    v->data = calloc(v->capacity, sizeof(int));
    for (int i = 0; i < v->size; i++) {
      v->data[i] = old[i];
    }
    free(old);
  }
  v->size = s;
}
```

La structure `Vector` contient le tableau stockant les éléments dans le champ `data`, la taille totale de ce tableau dans le champ `capacity` et le nombre d'éléments effectivement stockés dans le tableau dans le champ `size`.

Il est important de noter que la fonction `vector_create` renvoie un tableau redimensionnable dont la capacité est spécifiée par l'utilisateur mais dont la taille est pour l'instant 0. Il faut commencer par se servir de la fonction `vector_resize` pour définir la taille du tableau redimensionnable.
Bien entendu, on pourrait imaginer passer également une taille initiale à la fonction `vector_create`.

Toute la subtilité se trouve dans le code de la fonction `vector_resize` :

— Si la taille demandée s ne tient pas dans la capacité actuelle, on agrandit le tableau interne.
  1. On choisit de *doubler* la capacité (on s'assure également que la nouvelle capacité est suffisante, au cas où s dépasse $2 \times$ `capacity`).
  2. On alloue alors un nouveau tableau dans `v->data`, vers lequel on copie tous les éléments, sans oublier de désallouer ensuite l'ancien tableau.

— Lorsque la taille demandée s est en revanche plus petite que la capacité, il n'y a rien à faire, si ce n'est mettre à jour le champ `v->size`.

Accumulation

Un tableau redimensionnable peut notamment être utilisé pour accumuler des éléments dont on ne connaît pas le nombre à l'avance. Pour cela, on peut avantageusement écrire une fonction `vector_push` qui agrandit le tableau d'une unité et stocke un nouvel élément dans la dernière case.

```
void vector_push(vector *v, int x) {
  int n = vector_size(v);
  vector_resize(v, n+1);
  vector_set(v, n, x);
}
```

Complexité Considérons une séquence de n opérations `vector_push`, à partir d'un tableau initialement vide, et montrons que le coût total est proportionnel à n. Ainsi, on pourra considérer qu'une opération `vector_push` a une complexité amortie $O(1)$. On peut faire la preuve élémentairement : on va faire successivement $k = \log(n)$ redimensionnements du tableau, avec des coûts respectifs $1, 2, 4, \ldots, 2^k$, auxquels s'ajoutent un coût constant pour chaque opération `push`, ce qui fait un total de

$$n \times 1 + \sum_{i=0}^{k} 2^i = n + 2^{k+1} - 1 = n + 2n - 1.$$

On peut également faire cette preuve avec la méthode du potentiel. On pose

$$\Phi(v) \stackrel{\text{def}}{=} \max(0,\, 4 \times v.\texttt{size} - 2 \times v.\texttt{capacity}).$$

Dans la suite, on note s la valeur de `v.size` et c la valeur de `v.capacity`. Pour une opération `vector_push(x, v)`, il y a deux cas de figure.

— Si v n'est pas redimensionné, alors le coût réel est 1 et donc le coût amorti est

$$a = 1 + \Phi(\text{après}) - \Phi(\text{avant}).$$

  — Si le potentiel valait 0 et reste à 0, car $s + 1 \leq c/2$, alors $a = 1$.
  — Sinon, on a

$$\begin{aligned} a &= 1 + \Phi(\text{après}) - \Phi(\text{avant}) \\ &= 1 + (4(s+1) - 2c) - (4s - 2c) \\ &= 5. \end{aligned}$$

— Si v est au contraire redimensionné, parce que $s = c$, alors le coût réel est $1 + 2s$ (déplacement vers un tableau de taille $2s$ et quelques opérations constantes) et donc le coût amorti est

$$\begin{aligned} a &= 1 + 2s + \Phi(\text{après}) - \Phi(\text{avant}) \\ &= 1 + 2s + (4(s+1) - 4s) - (4s - 2s) \\ &= 5. \end{aligned}$$

Le coût amorti est donc toujours inférieur ou égal à 5. Dès lors, le Theoreme de amortissement nous dit que la suite de n opérations a un coût réel total

$$\sum c_i \leq \sum a_i \leq 5n,$$

c'est-à-dire proportionnel au nombre n d'opérations.

Les listes de Python

Le langage Python fournit nativement une structure de tableau redimensionnable, appelée *liste* (type `list` de Python), avec une syntaxe agréable. On peut ainsi écrire

```
t = [1, 2, 3]
t[2] = 42
t.append(4)
```

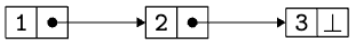
Ici, la méthode `append` agrandit le tableau redimensionnable `t` pour lui ajouter à droite un quatrième élément, exactement comme notre fonction `vector_push`. On dispose inversement d'une méthode `pop` pour retirer et renvoyer le dernier élément. Comme expliqué dans cette section, on peut considérer que les méthodes `append` et `pop` ont une complexité amortie $O(1)$.
Le langage Python fournit également des opérations pour extraire un fragment de liste, sous la forme d'une nouvelle liste. Ainsi, on peut écrire `t[i:j]` pour extraire la liste des éléments de `t` située entre les indices i inclus et j exclus. À la différence de `append` et `pop`, c'est là une opération très coûteuse, en $O(j-i)$.

Le module Buffer d'OCaml

La bibliothèque standard d'OCaml ne fournit pas de structure de tableau redimensionnable en toute généralité, c'est-à-dire pour des éléments d'un type quelconque, mais fournit en revanche un module `Buffer` de tableaux redimensionnables contenant des caractères de type `char`. Sa réalisation est tout à fait analogue à ce que nous venons de faire en C.
Avec le module `Buffer`, on peut donc construire de grandes chaînes par concaténations successives, de caractères ou de chaînes, avec une complexité totale linéaire. Une fois la construction terminée, on peut récupérer la chaîne complète, de type `string`.

2 Listes chaînées

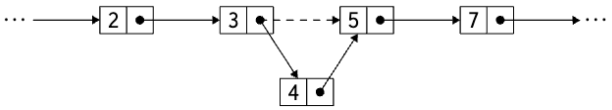
Une *liste chaînée* permet de représenter une séquence finie de valeurs, par exemple des entiers. Comme le nom le suggère, sa structure est caractérisée par le fait que les éléments sont chaînés entre eux, permettant le passage d'un élément à l'élément suivant. Ainsi, chaque élément est stocké dans un petit bloc alloué quelque part dans la mémoire, que l'on pourra appeler *maillon* ou *cellule*, et y est accompagné d'une deuxième information : l'adresse mémoire où se trouve la cellule contenant l'élément suivant de la liste. Ici, on a illustré une liste contenant



trois éléments, respectivement 1, 2 et 3. Chaque élément de la liste est matérialisé par un

emplacement en mémoire contenant d'une part sa valeur (dans la case de gauche) et d'autre part l'adresse mémoire de la valeur suivante (dans la case de droite). Dans le cas du dernier élément, qui ne possède pas de valeur suivante, on utilise une valeur spéciale désignée ici par le symbole $\perp$ et marquant la fin de la liste.

On comprend que la liste chaînée va utiliser plus de mémoire que le tableau pour stocker un même nombre d'éléments. En revanche, elle permet de réaliser certaines opérations plus efficacement qu'avec un tableau. Ainsi, on peut facilement insérer un nouvel élément dans une liste chaînée, entre deux éléments consécutifs, avec seulement deux affectations.



Ici, pour insérer 4 entre 3 et 5, on a simplement fait pointer 3 vers 4 et 4 vers 5. De la même façon, on peut supprimer un élément avec une seule affectation. Il suffit de faire pointer l'élément précédent vers l'élément suivant, pour « sauter » par-dessus l'élément supprimé.

Implémentation

En C

```c
typedef struct Cell {
  int value;
  struct Cell *next;
} list;

list *list_cons(int x, list *n) {
  list *l = malloc(sizeof(struct Cell));
  l->value = x;
  l->next = n;
  return l;
}
```

Le programme contient une structure `Cell` pour représenter des cellules de listes chaînées contenant des entiers. Le champ `value` contient la valeur entière et le champ `next` contient un pointeur vers la suite de la liste, ou `NULL` s'il s'agit de la dernière cellule de la liste. Le type `list` est un raccourci pour cette structure. Le programme contient également une fonction `list_cons` pour allouer sur le tas une nouvelle cellule. Ainsi, on peut écrire

```c
list *lst = list_cons(1, list_cons(2, list_cons(3, NULL)));
```

ce qui a pour effet d'allouer trois cellules en mémoire, chaînées pour former une liste de longueur 3. La variable `lst` contient un pointeur vers la première cellule, ce que l'on peut illustrer ainsi :
Il est important de comprendre que les noms `value` et `next` ne sont pas matérialisés en mémoire. Le compilateur C associe aux noms `value` et `next` des positions à l'intérieur du bloc mémoire qui stocke la structure et produit du code qui ne fait que de l'arithmétique de pointeurs.

En OCaml

Pour définir des listes simplement chaînées en OCaml, on pourrait définir un type comme

```
type 'a list = Nil | Cons of 'a * 'a list
```

où le constructeur constant `Nil` représente la liste vide et le constructeur `Cons` représente une cellule de liste. Il se trouve qu'un tel type `list` est prédéfini en OCaml. Seuls les noms des constructeurs sont différents : la liste vide se note `[]` et une cellule est construite avec `::`.
Le type `'a list` est un *type polymorphe*, où `'a` représente le type des éléments de la liste.

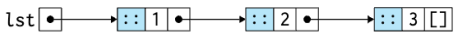
Exemple

la liste `1 :: 2 :: 3 :: []` a le type `int list`
la liste `true :: false :: []` a le type `bool list`.
Tous les éléments d'une même liste doivent avoir le même type.

Comme en C, une valeur de type `list`, autre que la liste vide `[]`, est un pointeur vers un bloc mémoire contenant deux valeurs, l'élément et la suite de la liste. Ainsi, lorsqu'on écrit

```
let lst = 1 :: 2 :: 3 :: []
```

on alloue trois blocs mémoire, pour les trois cellules de la liste, et la variable `lst` contient un pointeur vers la première cellule, ce que l'on peut illustrer ainsi :



Comparaison
La représentation en mémoire est légèrement différente de celle du C. Une partie du bloc mémoire est en effet utilisée par OCaml pour stocker de la méta-information, comme la nature du bloc et sa taille. Cette information est notamment utilisée par le GC d'OCaml. C'est ce que j'ai représenté ici en bleu.

Au-delà de cette petite différence de représentation en mémoire, il y a deux aspects plus importants qui distinguent nos listes chaînées en C et en OCaml.

— Les listes C sont *mutables*, c'est-à-dire qu'on peut modifier la valeur d'un champ `value` ou d'un champ `next` d'une cellule de liste, alors que les listes OCaml sont *immuables*, c'est-à-dire qu'on ne peut modifier ni le contenu ni la structure d'une liste une fois qu'elle est construite.
— Les listes C sont *monomorphes*, c'est-à-dire qu'elles ne contiennent que des valeurs de type `int`, là où les listes OCaml sont *polymorphes*, c'est-à-dire qu'elles peuvent contenir des valeurs d'un type quelconque.

2.1 Opérations sur les listes chaînées

Dans cette section, nous programmons quelques opérations élémentaires sur les listes chaînées, à la fois en C et en OCaml. En ce qui concerne les listes d'OCaml, le module `List` de la bibliothèque standard fournit déjà la plupart de ces opérations.

Longueur d'une liste. Pour calculer la longueur d'une liste, il suffit de parcourir toutes ses cellules jusqu'à atteindre la liste vide, en maintenant le décompte des cellules parcourues.

OCaml

Sur les listes d'OCaml, on peut le faire avec une fonction récursive qui examine la structure de la liste.

```
let rec length l = match l with
| [] -> 0
| _ :: t -> 1 + length t
```

La complexité est clairement proportionnelle à la longueur de la liste. Sur une liste très grande, une telle fonction pourrait faire déborder la pile d'appels.
Exercice : écrire une variante de la fonction `length` qui ne risque plus de faire déborder.

C

Sur les listes C, on pourrait également calculer la longueur avec une fonction récursive, comme ci-dessus, mais il est plus idiomatique de le faire avec une boucle. On commence par introduire une variable locale `len` pour décompter les cellules de la liste.

```
int list_length(list *l) {
  int len = 0;
```

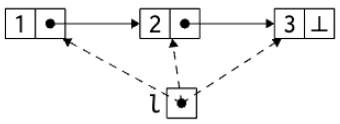
On procède ensuite avec une boucle `while`. Tant que la variable `l` ne contient pas `NULL`, on incrémente `len` puis on passe à la cellule suivante.

```
  while (l != NULL) {
    len++;
    l = l->next;
  }
```

Une fois sorti de la boucle, on renvoie la longueur.

```
  return len;
}
```

La complexité est clairement proportionnelle à la longueur de la liste. Il est important de comprendre que seule la variable `l`, locale à la fonction `list_length`, est modifiée.



Elle pointe successivement sur chaque cellule de la liste, et contient `NULL` au final. Elle disparaît avec le retour de fonction.
Si d'aventure on s'amusait à modifier le champ `next` de l'une des cellules pour le faire pointer sur une cellule précédente, alors la fonction `list_length` ne terminerait plus.

Parcours d'une liste
D'une manière générale, un parcours de liste va ressembler à ceci :

— En Ocaml

```
let rec f l = match l with
| [] -> ...
| x :: r -> ... f r ...
```

— en C

```
for (list *l = ...; l != NULL; l = l->next)
  ... l->value ...
```

Ce n'est pas une règle absolue pour autant. Ainsi, on peut parcourir deux listes à la fois, faire un double parcours d'une même liste, etc.

Accès au n-ième élément d'une liste
On pourrait tout à fait écrire une fonction pour renvoyer le n-ième élément d'une liste :

— En C

```
int list_nth(list *l, int n)
```

— en OCaml

```
nth : 'a list -> int -> 'a
```

Ce n'est pas pour autant une bonne idée, car c'est une opération coûteuse : accéder au n-ième élément d'une liste a un coût directement proportionnel à n. En particulier, il serait catastrophique de parcourir une liste de la manière suivante

```
int n = list_length(l);
for (int i = 0; i < n; i++)
  ... list_nth(l, i) ...
```

car le coût total serait alors quadratique ($1 + 2 + \cdots + n \sim n^2/2$).

S'il est nécessaire d'accéder souvent au i-ième élément, il convient de se demander si la structure de liste chaînée est la plus adaptée.
Si un tableau, voire un tableau redimensionnable, peut être utilisé à la place, il ne faut pas hésiter. Et si le tableau n'est pas une option, il existe des structures à base d'arbres, que nous décrirons plus loin, qui permettent un accès au i-ième élément en temps logarithmique.

Concaténation de deux liste
Considérons maintenant la concaténation de deux listes, c'est-à-dire l'opération ajoutant les éléments d'une seconde liste à la fin d'une première liste. Ainsi, si on concatène la liste $1, 2, 3$ avec la liste $4, 5$, on obtient la liste $1, 2, 3, 4, 5$.

Il y a fondamentalement deux façons de procéder.

1. Soit on modifie la première liste, en place, pour que sa dernière cellule pointe désormais sur la première cellule de la seconde liste.
2. Soit on construit une troisième liste contenant le résultat de la concaténation, sans modifier les deux listes initiales.

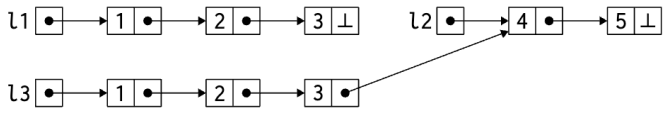
Dans le **langage OCaml**, on n'a pas le choix. Les listes sont en effet immuables, et la première solution n'est donc pas envisageable. On écrit la concaténation comme une fonction récursive `append` qui parcourt la première liste `l1`.

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x1 :: t1 -> x1 :: append t1 l2
```

On peut apprécier cette définition, quasi littérale, de la concaténation de deux listes. Il est intéressant de bien comprendre ce que fait cette fonction. Si on exécute par exemple le programme suivant,

```
let l1 = [1; 2; 3]
let l2 = [4; 5]
let l3 = append l1 l2
```

alors on se retrouve à avoir construit 8 cellules de liste au total, organisées comme ceci :



On note que les cellules de `l2` sont *partagées* entre les listes `l2` et `l3`. En effet, lorsque la fonction `append` est arrivée à la fin de la liste `l1`, elle s'est contentée de renvoyer `l2`, c'est-à-dire l'adresse de la première cellule de `l2`. Un tel partage ne pose aucun problème, car la liste $[4; 5]$ est immuable. Au contraire, on a ainsi gagné de la mémoire.

Les cellules de la liste `l1` ont en revanche été dupliquées. On le voit dans le code de la fonction `append`, où chaque constructeur `::` de `l1` donne lieu à une nouvelle application de `::`.
Cette duplication des cellules de `l1` est inévitable, car la dernière cellule doit maintenant pointer vers `l2`, et donc être différente, ce qui implique que les précédentes soient différentes également.
Notons au passage que la complexité de la fonction `append` est directement proportionnelle à la longueur de `l1` ; la longueur de `l2` n'importe pas.

Il est important de comprendre, par ailleurs, que le langage OCaml ne duplique jamais des structures allouées en mémoire par lui-même. Son passage par valeur implique la copie de valeurs, mais ces valeurs ne sont que des adresses ou des entiers. Seul un code écrit par le programmeur, intentionnellement ou accidentellement, peut se retrouver à dupliquer des blocs mémoire, comme ici notre fonction `append`.

Dans le **langage C**, on pourrait écrire exactement la même fonction de concaténation. C'est certes un peu plus verbeux, mais le fonctionnement serait exactement le même.

Cela étant, il y a tout de même une différence : la liste `l2` étant mutable, son partage dans la liste résultat n'est pas anodin. Toute modification de `l2` entraînerait une modification de la concaténation.
Inversement, une modification de la seconde partie de la concaténation modifierait `l2`. Une alternative consisterait alors à dupliquer également les cellules de `l2`.

Mais les listes sont ici mutables : une troisième solution consiste à modifier la dernière cellule de `l1` pour qu'elle pointe désormais sur `l2`, c'est-à-dire en faisant une concaténation *en place*.

Il y a cependant une petite difficulté. Si la liste `l1` est vide, alors il n'y a aucune cellule que l'on puisse ainsi modifier. Pour y remédier, on va écrire une fonction

```
list *list_append(list *l1, list *l2)
```

qui renvoie la liste résultat de cette concaténation en place. Si `l1` n'est pas vide, alors la valeur renvoyée sera la valeur de `l1`, c'est-à-dire l'adresse de la première cellule de `l1`. Si en revanche `l1` est vide, alors ce sera l'adresse de la première cellule de `l2` qui sera renvoyée, ou `NULL` si les deux listes sont vides.

On commence par se débarrasser du cas où `l1` est vide.

```
list *list_append(list *l1, list *l2) {
  if (l1 == NULL) return l2;
```

Sinon, il faut parcourir la liste `l1` jusqu'à atteindre son dernier bloc. On le fait avec une boucle `while`, en maintenant dans la variable `p` le dernier bloc rencontré.

```
  list *p = l1, *c = l1->next;
  while (c != NULL) {
    p = c;
    c = c->next;
  }
```

Une fois sorti de la boucle, on peut modifier le champ `next` de la dernière cellule pour le faire pointer sur `l2`, et enfin renvoyer `l1`.

```
  p->next = l2;
  return l1;
}
```

La complexité est ici directement proportionnelle à la longueur de la liste `l1` ; la longueur de `l2` n'importe pas.

Gestion de la mémoire
L'utilisation de listes chaînées conduit fatalement à des cellules de listes qui ne sont plus utilisées. Il convient que cet espace mémoire soit libéré, afin de pouvoir être réutilisé dans la suite de l'exécution du programme.

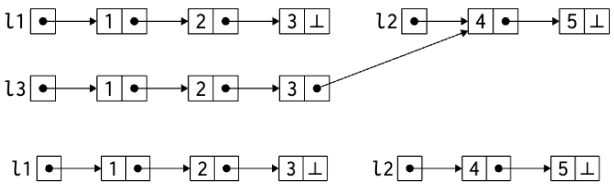
Dans le cas du langage OCaml, cette libération est assurée, automatiquement, par le GC. Illustrons-le sur un petit exemple qui utilise la fonction `append` définie plus haut.

```
let l1 = [1; 2; 3]
let l2 = [4; 5]
let l1 = append l1 l2
```

Après l'exécution des deux premières lignes, on se retrouve avec la situation suivante :

Après l'exécution de la troisième ligne, on a maintenant `l1` qui pointe vers la liste $[1; 2; 3; 4; 5]$, les premiers trois blocs ayant été copiés, comme nous l'avons expliqué plus haut.

Sur cet exemple très simple, les trois cellules de la liste initiale $[1; 2; 3]$ ne sont plus accessibles à partir des variables du programme et peuvent donc être récupérées par le GC.



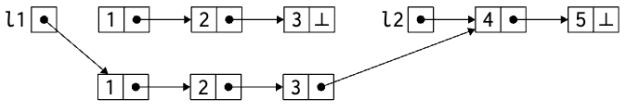
De manière générale, le critère utilisé par le GC pour déterminer les éléments utiles ou non à un instant donné est leur accessibilité à partir des variables du programme (variables globales du programme ou variables locales des appels de fonction en cours d'exécution) : un élément en mémoire que l'on ne peut plus atteindre en partant de ces variables peut être considéré comme définitivement perdu, et l'espace mémoire qu'il occupe est alors recyclé. On ne sait pas *quand* cette libération de la mémoire aura lieu, mais on sait qu'elle arrivera tôt ou tard.

En C, la libération des blocs mémoire alloués avec `malloc` et désormais inutilisés est laissée à la charge du programmeur. On utilise pour cela la fonction `free`. Le programme suivant contient le code d'une fonction `list_delete` qui libère toutes les cellules d'une liste chaînée. On notera comment une cellule est libérée *après* avoir accédé à son champ `next`. En effet, il ne serait pas correct d'accéder au bloc mémoire après l'avoir libéré. Il faut avoir bien conscience que la libération explicite avec `free` comporte des risques. D'une part, il ne faut pas libérer un même bloc deux fois. D'autre part, il ne faut plus chercher à accéder à un bloc qui a été libéré. Le langage C ne nous aide absolument pas à cet égard.

Programme - destruction d'une liste chaînée

```
void list_delete(list *l) {
  while (l != NULL) {
    list *p = l;
    l = l->next;
    free(p);
  }
}
```

Variantes des listes chaînées. Il existe de nombreuses variantes de la structure de liste chaînée, dont la *liste cyclique*, où le dernier élément est lié au premier, ou la *liste doublement chaînée*, où chaque élément est lié à l'élément suivant et à l'élément précédent dans la liste, ou encore la liste cyclique doublement chaînée qui combine ces deux variantes.


l1
1
2
3
⊥
l2
4
5
⊥
1
2
3