Ce chapitre contient de nombreux exemples d'interfaces (comme par exemple l'interface d'une pile dans le programme 7.8) et d'implémentations de ces interfaces (comme par exemple les programmes 7.9, 7.10, 7.11 et 7.12 pour réaliser une pile). Les exemples de codes clients se trouvent plutôt dans les autres chapitres, notamment là où les structures de données sont utilisées pour réaliser des algorithmes, mais aussi parfois dans les exercices (comme par exemple l'exercice 83 qui utilise une pile).

Exercice 83 p.426

Invariant de structure

De la même façon qu'un invariant de boucle (voir page 193) 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.
Dans ce chapitre, nous verrons plusieurs exemples d'invariants de structure.

7.2 Structures de données séquentielles

Dans cette section, on présente plusieurs structures de données élémentaires construites à partir de *tableaux* et de *listes*, deux structures fondamentales où les éléments sont naturellement *ordonnés*.

7.2.1 Tableaux

Le *tableau* est la structure de données la plus simple et la plus efficace. Rien de surprenant à cela : l'ordinateur *est* un tableau. 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 (comme la lecture d'une bande magnétique sur les premiers ordinateurs).

En C. Les tableaux C sont présentés en détail dans la section 4.2.2. On rappelle qu'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.

7.2.2 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*[2] 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. Nous verrons de belles applications du tableau redimensionnable dans les sections 7.2.4 et 7.2.6 notamment.

Le programme 7.2 contient l'interface C d'une structure de tableau redimensionnable contenant des entiers. 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, et dont la taille vaut 0. 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

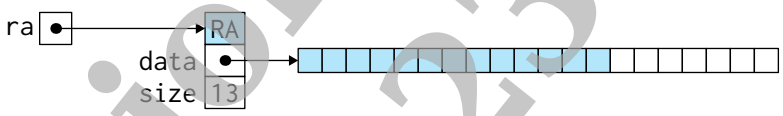
2. Le tableau redimensionnable est parfois également appelé *tableau dynamique*. Dans certaines bibliothèques, on le trouve sous le nom de `Vector` ou encore `ArrayList`.

Programme 7.2 – 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);
```

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`.

Implémentation en C. Le programme 7.3 contient une implémentation C de tableau redimensionnable. 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`.

Exercice 75 p.425

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, ou d'une fonction comme `vector_push` que nous verrons plus loin dans ce chapitre. Bien entendu, on pourrait imaginer passer également une taille initiale à la fonction `vector_create`.

Programme 7.3 – 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;
}
```

Toute la subtilité se trouve dans le code de la fonction `vector_resize`. Lorsque la taille demandée `s` ne tient pas dans la capacité actuelle, on agrandit le tableau interne. On choisit ici de *doubler* la capacité. Nous verrons ci-dessous en quoi cette stratégie est pertinente. On s'assure également que la nouvelle capacité est suffisante, au cas où `s` dépasse $2 \times$ `capacity`. 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`. On pourrait cependant choisir de rétrécir le tableau interne, ce que l'exercice 76 propose de faire.

Exercice 76 p.425

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);
}
```

Comme nous allons le voir maintenant, c'est bien plus efficace qu'il n'y paraît.

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

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

On peut également faire cette preuve avec la méthode du potentiel décrite dans la section 6.3.7. On pose

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

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 figures.

- 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 $s + 1 \leqslant c/2$, 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 théorème 6.9 nous dit que la suite de n opérations a un coût réel total

$$\sum c_i \leqslant \sum a_i \leqslant 5n$$

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)
```

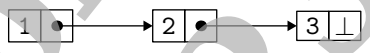
Ici, la méthode `append` agrandit le tableau redimensionnable `t` pour lui ajouter à droite un quatrième élément, exactement comme notre fonction `vector_push`. On dispose inversement d'une méthode `pop` pour retirer et renvoyer le dernier élément. Comme expliqué dans cette section, on peut considérer 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 $O(\texttt{j} - \texttt{i})$. Cela n'est pas surprenant quand on a compris qu'il s'agit d'un tableau redimensionnable.

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`. Nous nous servirons du module `Buffer` dans la section 9.5.2 pour écrire des programmes de compression de texte.

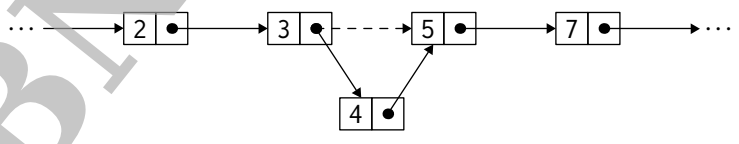
7.2.3 Listes chaînées

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 ⊥ et marquant la fin de la liste [3].

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 :



3. Le symbole ⊥, dont le nom officiel est « taquet vers le haut », est utilisé pour désigner plusieurs choses en mathématiques et informatique. Les logiciens, par exemple, l'utilisent pour désigner la contradiction. En anglais, on le désigne parfois sous le nom de *bottom*.

Programme 7.4 – listes chaînées, en 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;
}
```

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.

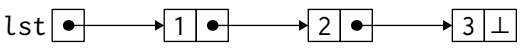
Dans cette section, nous montrons comment réaliser la structure de liste chaînée en C et en OCaml, puis comment écrire des opérations sur cette structure. Dans les trois sections suivantes, nous verrons des applications de la structure de liste chaînée, pour réaliser respectivement des piles, des files et des tables de hachage.

7.2.3.1 Implémentation

En C. Le programme 7.4 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 7.4 contient également une fonction `list_cons` pour allouer sur le tas une nouvelle cellule. Ainsi, on peut écrire

```
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 (typiquement 0 et 4) 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 `::`. Nous avons déjà croisé les listes d'OCaml dans le chapitre 3. Le type `'a list` est *un type polymorphe*, où `'a` représente le type des éléments de la liste. Ainsi, la liste `1::2::3::[]` a le type `int list` et 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 :

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 qu'on a représenté ici en bleu.

Comparaison. 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.

7.2.3.2 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. 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
```

Pour la liste vide, on renvoie 0.

```
  | []     -> 0
```

Pour une liste non vide, on calcule récursivement la longueur de la queue de la liste, à laquelle on ajoute un.

```
  | _ :: t -> 1 + length t
```

Exercice 77 p.425

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. L'exercice 77 propose d'écrire une variante de la fonction `length` qui ne risque plus de faire déborder la pile.

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;
```

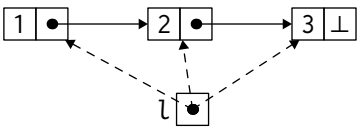
On procède ensuite avec une boucle **while**. Tant que la variable `l` ne contient pas `NULL`, on incrémente `len` puis on passe à la cellule suivante.

```
  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.



Programme 7.5 – calcul de la longueur d'une liste

- En OCaml, avec une fonction récursive :

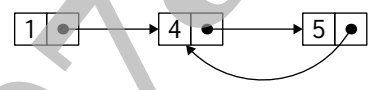
```
let rec length (l: 'a list) : int =
  match l with
  | []     -> 0
  | _ :: t -> 1 + length t
```

- En C, avec une boucle :

```
int list_length(list *l) {
  int len = 0;
  while (l != NULL) {
    len++;
    l = l->next;
  }
  return len;
}
```

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, comme ceci,



alors la fonction `list_length` ne terminerait plus. L'exercice 82 propose un algorithme efficace pour déterminer si une liste est cyclique à partir d'un certain rang.

Exercice 82 p.426

Le programme 7.5 regroupe les deux fonctions de calcul de la longueur d'une liste en OCaml et en C. On peut rendre le code C un peu plus compact, et un peu plus idiomatique, en utilisant plutôt une boucle `for` :

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

C'est essentiellement affaire de style.

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

Exercice 80 p.426

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

et un parcours de liste en C va ressembler à ceci

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

Exercice 82 p.426

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, elle pourrait prendre la forme d'une fonction `int list_nth(list *l, int n)` et en OCaml la forme d'une fonction `nth: 'a list -> int -> 'a`. C'est un exercice relativement simple d'écrire une telle fonction. 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. La liste chaînée n'en reste pas moins utile, comme nous le verrons dans la suite de ce chapitre.

Concaténation de deux listes. 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. Soit on modifie la première liste, en place, pour que sa dernière cellule pointe désormais sur la première cellule de la seconde liste. 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
```

Si la liste `l1` est vide, il suffit de renvoyer `l2`.

```
  | []       -> l2
```

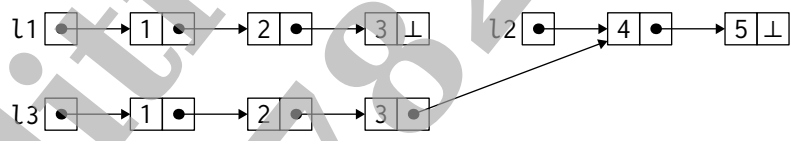
Sinon, c'est que la liste `l1` commence par une certaine valeur, `x1`. On construit alors une nouvelle cellule avec `x1` et avec la concaténation du reste de la liste et de `l2`.

```
  | 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 bien 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. Cette idée de partage de structures immuables est importante. Nous y reviendrons par la suite, notamment avec les arbres. 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 qu'en OCaml. 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 puisque 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;
}
```

Programme 7.6 – concaténation de deux listes

- En OCaml, on réalise une concaténation purement applicative, qui renvoie une nouvelle liste. Les cellules de la liste `l2` sont partagées, celles de la liste `l1` sont dupliquées.

```
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
  match l1 with
  | []       -> l2
  | x1 :: t1 -> x1 :: append t1 l2
```

- En C, on réalise une concaténation en place, c'est-à-dire qu'on modifie la dernière cellule de la liste `l1` pour qu'elle pointe désormais sur la liste `l2`. La fonction renvoie la première cellule de la liste.

```
list *list_append(list *l1, list *l2) {
  if (l1 == NULL) return l2;
  list *p = l1, *c = l1->next;
  while (c != NULL) {
    p = c;
    c = c->next;
  }
  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.

Le programme 7.6 regroupe les deux fonctions de concaténation de listes, en OCaml et en C.

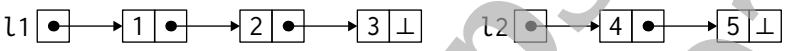
Gestion de la mémoire. L'utilisation de listes chaînées conduit fatalement à des cellules de listes qui ne sont plus utilisées. Il convient que cet espace mémoire soit libéré, afin de pouvoir être réutilisé dans la suite de l'exécution du programme.

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.

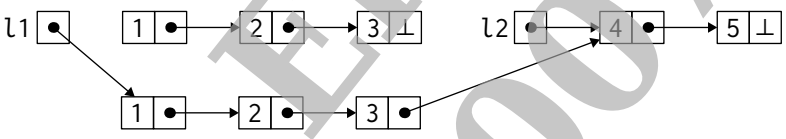
```
let l1 = [1; 2; 3]
let l2 = [4; 5]
```

```
let l1 = append l1 l2
```

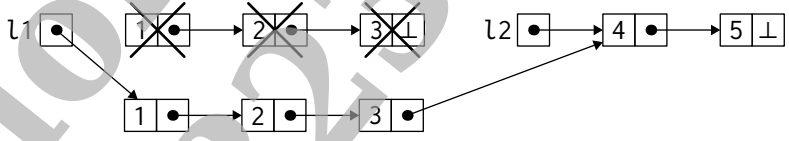
Après l'exécution des deux premières lignes, on se retrouve avec la situation suivante :



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.

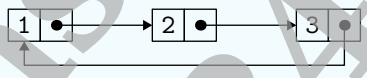
En C, la libération des blocs mémoire alloués avec `malloc` et désormais inutilisés est laissée à la charge du programmeur. On utilise pour cela la fonction `free`. Le programme 7.7 contient le code d'une fonction `list_delete` qui libère toutes les cellules d'une liste chaînée. On notera comment une 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 7.7 – destruction d'une liste chaînée

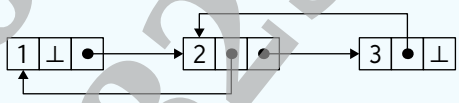
```
void list_delete(list *l) {
  while (l != NULL) {
    list *p = l;
    l = l->next;
    free(p);
  }
}
```

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,

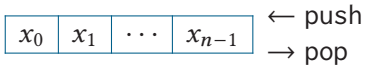


ou encore la liste cyclique doublement chaînée qui combine ces deux variantes.

7.2.4 Piles

Le concept d'une pile est bien connu. Tout le monde en fait l'expérience en empilant des assiettes. C'est le concept « dernier entré, premier sorti ». En anglais, on parle de LIFO pour *Last In First Out*.

En termes de structure de données, une pile est une séquence $x_0, x_1, \ldots, x_{n-1}$ de valeurs où il est possible de retirer un élément, avec une fonction pop, et d'ajouter un élément, avec une fonction push, du *même côté de la séquence*.



Programme 7.8 – interface d'une pile mutable

- En C, pour des éléments de type **int**.

```
typedef struct Stack stack;
stack *stack_create(void);
bool stack_is_empty(stack *s);
void stack_push(stack *s, int x);
int stack_top(stack *s);
int stack_pop(stack *s);
void stack_delete(stack *s);
```

- En OCaml, pour des éléments d'un type 'a.

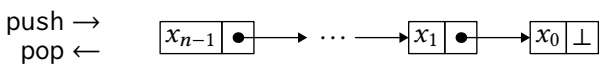
```
type 'a stack
val create: unit -> 'a stack
val is_empty: 'a stack -> bool
val push: 'a stack -> 'a -> unit
val top: 'a stack -> 'a
val pop: 'a stack -> 'a
```

Le sommet de la pile, ici dessiné à droite, contient l'élément x_{n-1}, qui est le dernier élément à avoir été ajouté sur la pile et qui sera le premier à sortir de la pile ; le fond de la pile, ici dessinée à gauche, contient l'élément x_0 qui a été ajouté en premier dans la pile et que sera le dernier à en sortir.

Le programme 7.8 contient l'interface d'une telle structure de pile (en anglais, on parle de *stack*). Pour le langage C, on a fait ici le choix de piles contenant des entiers. Pour le langage OCaml, les piles sont polymorphes. On va maintenant proposer différentes réalisations de cette interface.

7.2.4.1 Implémentation avec une liste chaînée

Il est immédiat de réaliser une pile avec une liste simplement chaînée, le sommet de la pile coïncidant avec la tête de la liste.



Le tête de la liste contient le dernier élément mis sur la pile, ici x_{n-1}. Le dernier élément de la liste contient le fond de la pile, c'est-à-dire le tout premier élément mis sur la pile, ici x_0.

Programme 7.9 – pile réalisée en C avec une liste chaînée

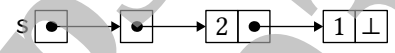
```
typedef struct Stack { list *head; } stack;

stack *stack_create(void) {
  stack *s = malloc(sizeof(struct Stack));
  s->head = NULL;
  return s;
}
bool stack_is_empty(stack *s) {
  return s->head == NULL;
}
void stack_push(stack *s, int x) {
  s->head = list_cons(x, s->head);
}
int stack_top(stack *s) {
  assert(!stack_is_empty(s));
  return s->head->value;
}
int stack_pop(stack *s) {
  assert(!stack_is_empty(s));
  int v = s->head->value;
  list *p = s->head;
  s->head = p->next;
  free(p);
  return v;
}
void stack_delete(stack *s) {
  list_delete(s->head);
  free(s);
}
```

En C. Le programme 7.9 contient une implémentation C de cette idée, où la liste chaînée est encapsulée dans une structure `Stack`. Les fonctions `stack_top` et `stack_pop` sont défensives. Elles vérifient que la pile est non vide avec `assert`. Le code échouera donc si on tente d'accéder à une pile vide. La fonction `stack_is_empty` est là pour permettre à l'utilisateur de tester si une pile est vide avant d'y accéder.

Il est important de bien comprendre l'intérêt de la structure `Stack` dans laquelle la liste chaînée est encapsulée. Si on crée une pile dans une variable `s`, puis qu'on y ajoute deux éléments, on se retrouve dans la situation suivante :

```
stack *s = stack_create();
stack_push(s, 1);
stack_push(s, 2);
```



La variable `s` pointe vers une structure `Stack`, elle-même pointant vers une liste chaînée (par le champ `head`). Cette indirection permet notamment à la fonction `stack_push` d'ajouter un élément la toute première fois, lorsque la pile est vide. Si on avait stocké la liste chaînée directement dans la variable `s`, alors il ne serait pas possible d'écrire `stack_push(s, 1)` avec une variable `s` qui vaut `NULL`.

En OCaml. Le type `list` d'OCaml réalise *de facto* une pile immuable. L'opération `::` permet l'ajout d'un élément et le filtrage permet d'accéder au sommet de la pile. Si on veut une pile mutable conforme au programme 7.8, il suffit de placer la liste dans une référence. Le programme 7.10 en décrit l'implémentation. La bibliothèque standard d'OCaml fournit déjà un module `Stack` de piles mutables réalisées avec le type `list`. En interne, le type `Stack.t` est défini comme ceci :

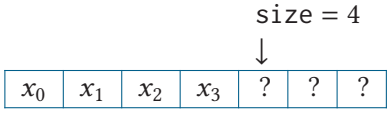
```
type 'a t = { mutable c: 'a list; mutable len: int; }
```

Comme on le voit, la taille de la pile est également maintenue, ce qui permet de l'obtenir en temps constant.

Complexité. Que ce soit en C ou en OCaml, nos piles réalisées avec une liste chaînée fournissent des opérations qui sont toutes de temps constant. On le constate facilement à la lecture du code.

7.2.4.2 Implémentation avec un tableau

Si la capacité de la pile est bornée, on peut avantageusement la réaliser directement dans un tableau. Plus précisément, les éléments de la pile sont placés au début du tableau, le fond de la pile étant l'indice 0. On illustre ici une pile de capacité 7 contenant 4 éléments.



Il suffit de maintenir la taille de la pile, ici notée `size`, pour accéder à la première case libre (à l'indice `size`) ou au sommet de la pile (à l'indice `size` − 1).

Programme 7.10 – pile mutable en OCaml

```
type 'a stack = 'a list ref

let create () : 'a stack =
  ref []

let is_empty (st: 'a stack) : bool =
  !st = []

let push (st: 'a stack) (x: 'a) : unit =
  st := x :: !st

let top (st: 'a stack) : 'a =
  match !st with
  | []     -> invalid_arg "top"
  | x :: _ -> x

let pop (st: 'a stack) : 'a =
  match !st with
  | []     -> invalid_arg "pop"
  | x :: l -> st := l; x
```

Programme 7.11 – pile réalisée en C avec un tableau

```
typedef struct Stack {
  int capacity;
  int size;     // 0 <= size <= capacity
  int *data;    // tableau de taille capacity
} stack;

stack *stack_create(int c) {
  stack *s = malloc(sizeof(struct Stack));
  s->capacity = c;
  s->size = 0;
  s->data = calloc(c, sizeof(int));
  return s;
}
```

Le programme 7.11 donne une réalisation en C de cette idée. La structure `Stack` contient la capacité de la pile, son nombre d'éléments et le tableau qui les contient. À noter que la fonction `stack_create` demande maintenant la capacité de la pile en argument. Les opérations, immédiates à réaliser, sont laissées en exercice.

Pour ne pas avoir à borner la capacité de la pile, on peut remplacer le tableau par un tableau redimensionnable (section 7.2.2). Il n'est même pas nécessaire d'introduire une nouvelle structure, car un tableau redimensionnable *est* une pile. Le programme 7.12 définit des opérations pour manipuler un tableau redimensionnable comme une pile.

Complexité. Pour une pile réalisée par un tableau, les opérations sont toutes de temps constant, à l'exception de la création. Pour un tableau redimensionnable, en revanche, une opération `push` ou `pop` peut prendre un temps arbitraire, si elle déclenche un agrandissement ou un rétrécissement du tableau interne au tableau redimensionnable.

Cela étant, nous avons montré dans la section 7.2.2 qu'une séquence de n opérations `push` a tout de même une complexité totale $O(n)$, nous permettant de considérer que `push` a une complexité amortie constante. Il en va de même pour `pop`, et plus généralement pour une séquence arbitraire d'opérations `push` et `pop`, pourvu que le rétrécissement, le cas échéant, soit correctement réalisé (voir exercice 76).

Exercice 76 p.425

Programme 7.12 – pile réalisée en C avec un tableau redimensionnable

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

int vector_top(vector *v) {
  int n = vector_size(v) - 1;
  assert(0 <= n);
  return vector_get(v, n);
}

int vector_pop(vector *v) {
  int n = vector_size(v) - 1;
  assert(0 <= n);
  int r = vector_get(v, n);
  vector_resize(v, n);
  return r;
}
```

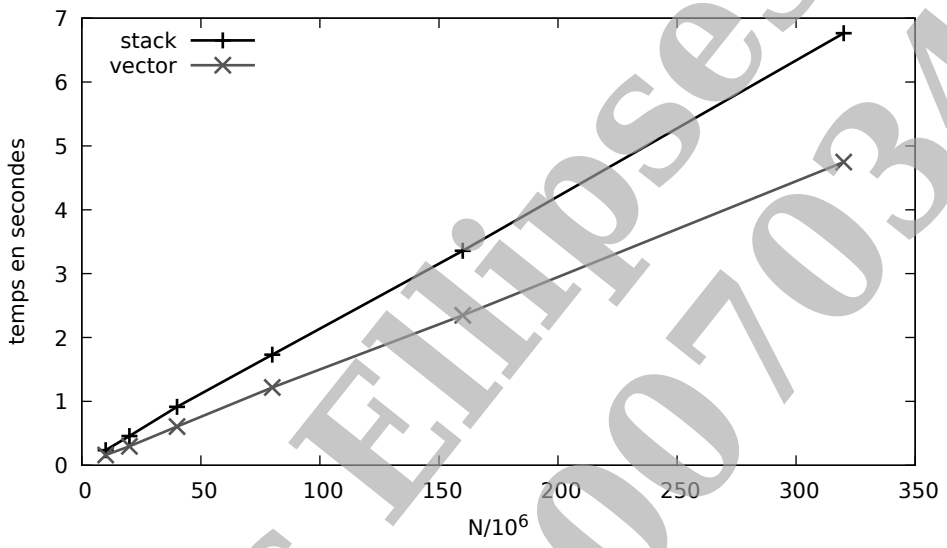


Figure 7.2 – Comparaison de deux structures de pile.

7.2.4.3 Comparaison

On compare ici les performances relatives de nos deux structures de pile, respectivement avec une liste chaînée et avec un tableau redimensionnable. Pour cela, on insère successivement les n premiers entiers dans une pile initialement vide et on mesure le temps total d'exécution. La figure 7.2 illustre les valeurs mesurées, jusqu'à $n = 3,2 \times 10^8$.

La première constatation est que le temps d'exécution est directement proportionnel à n. Cela est conforme à nos calculs de complexité. La seconde constatation est que la structure `vector` est 30% plus efficace que la structure `stack`, et ce malgré les copies qui sont faites d'un tableau vers un autre à chaque agrandissement du tableau redimensionnable. Ceci s'explique en particulier par un nombre significativement moindre d'appels à `malloc`.

Ajoutons que l'espace mémoire occupé est également en faveur de `vector`. En effet, dans le pire des cas, la moitié du tableau est inutilisé, soit $4n$ octets en plus de la place occupée par les éléments de la pile (en supposant des entiers 32 bits). Mais dans le cas de `stack`, les pointeurs `next` de la liste chaînée occupent *toujours* $8n$ octets supplémentaires.

Programme 7.13 – interface C d'une file contenant des entiers

```
typedef struct Queue queue;

queue *queue_create(void);
int queue_size(queue *q);
bool queue_is_empty(queue *q);
void queue_enqueue(queue *q, int x);
int queue_peek(queue *q);
int queue_dequeue(queue *q);
void queue_delete(queue *q);
```

7.2.5 Files

Le concept d'une file est bien connu. Tout le monde en fait l'expérience en allant acheter du pain à la boulangerie. C'est le concept « premier entré, premier sorti ». En anglais, on parle de FIFO pour *First In First Out*.

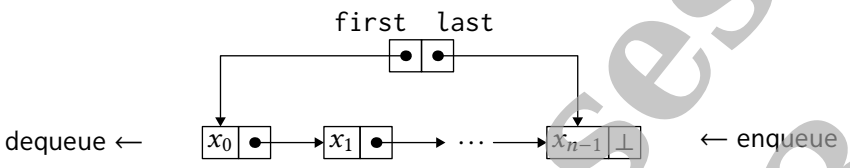
En termes de structure de données, une file est une séquence $x_0, x_1, \ldots, x_{n-1}$ de valeurs où il est possible de retirer un élément d'un côté, avec une fonction dequeue, et d'ajouter un élément de l'autre côté, avec une fonction enqueue.

La tête de la file, ici dessinée à gauche, contient l'élément x_0, qui sera le premier à sortir ; et la queue de la file, ici dessinée à droite, contient l'élément x_{n-1} qui sera (pour l'instant) le dernier à sortir.

Le programme 7.13 contient l'interface C d'une telle structure de file (en anglais, on parle de *queue*). La fonction `queue_create` renvoie une nouvelle file, pour l'instant vide. La fonction `queue_enqueue` ajoute un nouvel élément dans la file, la file étant modifiée en place. Il s'agit donc là d'une structure mutable. La fonction `queue_peek` permet d'examiner le premier élément de la file, s'il existe, sans le retirer de la file pour autant, alors que la fonction `queue_dequeue` retire et renvoie ce premier élément.

7.2.5.1 Implémentation avec une liste chaînée

On peut réaliser une file avec une liste simplement chaînée. On forme la liste des éléments dans l'ordre de leur insertion et on maintient un pointeur `first` vers le premier élément de la liste et un pointeur `last` vers le dernier élément de la liste.



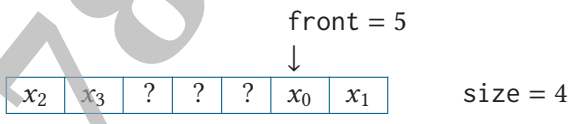
L'insertion d'un nouvel élément se fait à la fin de la liste. Elle peut être réalisée en temps constant car on a accès au dernier élément avec le pointeur `last`. Le retrait d'un élément se fait au début de la liste. Là encore, on peut le réaliser en temps constant car on a accès au premier élément avec le pointeur `first`. Lorsque la file est vide, les deux pointeurs `first` et `last` sont nuls.

Le programme 7.14 contient un code C qui réalise cette idée. Il réutilise le type `list` des listes simplement chaînées du programme 7.4 page 334. Il maintient par ailleurs le nombre d'éléments de la file dans un champ `size`. La principale difficulté de ce code consiste à maintenir l'invariant que `first` et `last` sont nuls si et seulement si la file est vide.

Pour ce qui est d'OCaml, la bibliothèque standard fournit un module `Queue` de files mutables tout à fait identique au programme 7.14.

7.2.5.2 Implémentation avec un tableau

Si la capacité de la file est bornée, on peut avantageusement la réaliser avec un simple tableau. Les éléments de la file y sont rangés à partir d'un certain indice et le tableau est utilisé *circulairement*[4]. On illustre ici une file de capacité 7 contenant 4 éléments rangés à partir de l'indice 5 dans le tableau.



Pour retirer un élément de la file, en supposant qu'elle n'est pas vide, on le récupère dans la case d'indice `front`, puis on incrémente `front`, modulo la taille du tableau, et on décrémente `size`. Pour ajouter un élément dans la file, en supposant qu'elle n'est pas pleine, il suffit de l'ajouter dans la case d'indice `front + size`, modulo la taille du tableau, puis on incrémente `size`.

Exercice 86 p.427

Le programme 7.15 contient une structure C pour mettre en œuvre cette idée. L'exercice 86 propose de réaliser les opérations sur cette structure. Il peut être intéressant d'ajouter à notre interface une fonction pour déterminer si la file est pleine, c'est-à-dire

```
bool ring_buffer_is_full(ring_buffer q);
```

4. En anglais, on parle de *ring buffer*.

Programme 7.14 – files mutables, en C, avec une liste chaînée

```
typedef struct Queue { int size; list *first, *last; } queue;

queue *queue_create(void) {
  queue *q = malloc(sizeof(struct Queue));
  q->size = 0;
  q->first = q->last = NULL;
  return q;
}
int queue_size(queue *q) { return q->size; }
bool queue_is_empty(queue *q) { return queue_size(q) == 0; }
void queue_enqueue(queue *q, int x) {
  if (is_empty(q)) {
    q->first = q->last = list_cons(x, NULL);
    q->size = 1;
    return;
  }
  list *c = list_cons(x, NULL);
  q->last->next = c;
  q->last = c;
  q->size++;
}
int queue_peek(queue *q) {
  assert(!queue_is_empty(q));
  return q->first->value;
}
int queue_dequeue(queue *q) {
  assert(!queue_is_empty(q));
  int v = q->first->value;
  list *p = q->first;
  q->first = p->next;
  free(p);
  if (q->first == NULL) q->last = NULL;
  q->size--;
  return v;
}
```

Programme 7.15 – files mutables, en C, dans un tableau circulaire

```
typedef struct RingBuffer {
  int capacity;
  int size;     // 0 <= size <= capacity
  int front;    // tête de la file, 0 <= front < capacity
  int *data;    // tableau de taille capacity
} ring_buffer;
```

Exercice 87 p.427

Pour ne pas limiter la capacité, on pourrait utiliser un tableau redimensionnable (voir section 7.2.2) plutôt qu'un tableau. C'est tout à fait possible, mais pas complètement évident à mettre en œuvre (voir exercice 87).

7.2.5.3 Files immuables

Les deux structures de files que nous venons de voir sont des structures mutables : la liste chaînée ou le tableau est modifié en place lorsque des éléments sont ajoutés ou retirés de la file. Dans cette section, nous présentons des files immuables, où les opérations ne modifient pas la file mais renvoient de nouvelles files. Le programme 7.16 présente une interface OCaml de files immuables polymorphes. La file vide, `empty`, n'est pas une fonction mais une constante. La fonction `enqueue` renvoie une nouvelle file, sans modifier son argument. La fonction `dequeue` renvoie à la fois l'élément retiré et une nouvelle file, sous la forme d'une paire, là encore sans modifier son argument.

Il existe une façon très simple et très élégante de réaliser de telles files. On se sert de *deux piles* : dans la première, on ajoute les éléments qui entrent dans la file ; dans la seconde, on retire les éléments qui sortent de la file. Lorsque la seconde pile est épuisée, on y déplace tous les éléments de la première pile.

Le programme 7.17 contient un code OCaml qui met en œuvre cette idée. Les deux piles sont ici réalisées directement par le type `list` d'OCaml et stockées dans deux champs `front` et `rear` d'un type enregistrement `queue`. La liste `front` est celle dans laquelle on retire des éléments et la liste `rear` celle dans laquelle on ajoute des éléments. Ces files sont immuables car le type `list` d'OCaml est immuable [5]. En conséquence, l'opération `enqueue` renvoie une nouvelle file, avec l'élément ajouté, et l'opération `dequeue` renvoie une paire contenant l'élément retiré de la file et une nouvelle file où cet élément a été retiré. Une autre conséquence du caractère

Exercice 85 p.427

5. Mais on pourrait tout à fait utiliser la même idée avec deux piles mutables ; on obtiendrait alors une file mutable. L'exercice 85 page 427 propose de le faire en C.

Programme 7.16 – Interface OCaml de files immuables

```
type 'a queue

val empty: 'a queue
val is_empty: 'a queue -> bool
val enqueue: 'a queue -> 'a -> 'a queue
val dequeue: 'a queue -> 'a * 'a queue
```

Programme 7.17 – Files immuables, en OCaml

```
type 'a queue = {
  front: 'a list; (* liste où on retire les éléments *)
   rear: 'a list; (* liste où on ajoute les éléments *)
}

let empty : 'a queue =
  { front = []; rear = [] }

let is_empty (q: 'a queue) : bool =
  q.front = [] && q.rear = []

let enqueue (q: 'a queue) (x: 'a) : 'a queue =
  { front = q.front; rear = x :: q.rear }

let dequeue (q: 'a queue) : 'a * 'a queue =
  match q.front with
  | x :: f ->
      x, { front = f; rear = q.rear }
  | [] ->
      begin match List.rev q.rear with
      | []     -> invalid_arg "dequeue"
      | x :: f -> x, { front = f; rear = [] }
      end
```

immuable est le fait que la file vide `empty` est une *valeur* et non pas une fonction.

Le code de la fonction `enqueue` est immédiat. Il s'exécute en temps constant. Le code de la fonction `dequeue`, en revanche, est plus complexe. S'il existe au moins un élément au début de la liste `front`, c'est immédiat. Si en revanche la liste `front` est vide, il faut alors renverser la liste `rear`, ce que l'on fait avec `List.rev`. Si le résultat est vide, c'est que `rear` était vide et on lève une exception signalant une tentative de retrait depuis une file vide. Sinon, on retire le premier élément `x` du résultat et le reste de la liste devient la nouvelle liste `front`. Le coût de l'opération `dequeue` est donc variable. Si le renversement n'est pas nécessaire, alors le coût est constant. Sinon, il est proportionnel à la longueur de la liste `rear`, qui peut être arbitrairement grande. Nous allons voir maintenant que cela reste néanmoins efficace.

Complexité. Montrons que les opérations `enqueue` et `dequeue` ont une *complexité amortie constante*, c'est-à-dire qu'une séquence de n opérations, où chaque opération est appliquée à la file obtenue avec l'opération précédente, s'exécute en un temps total proportionnel à n.

Pour cela, nous allons utiliser la méthode du potentiel décrite dans la section 6.3.7. Pour une file `q`, on définit son potentiel comme

$$\Phi(\texttt{q}) \stackrel{\text{def}}{=} \text{la longueur de la liste } \texttt{q.rear}.$$

Calculons alors le coût amorti de chaque opération. Pour `enqueue x q`, on ajoute `x` au début d'une liste `q.rear` contenant ℓ éléments, avec un coût réel 1 et donc un coût amorti

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

Pour `dequeue q`, il y a deux cas de figure.

- Si `q.front` n'est pas vide, le coût réel est 1, la liste `q.rear` ne change pas, et donc le coût amorti est

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

- Si `q.front` est vide et que la liste `q.rear` a pour longueur ℓ, alors le coût réel est $\ell + 1$ (le renversement d'une liste de ℓ éléments auquel s'ajoute un coût constant), la liste `q.rear` devient vide, et donc le coût amorti est

$$\begin{aligned} a &= \ell + 1 + \Phi(\text{après}) - \Phi(\text{avant}) \\ &= \ell + 1 + 0 - \ell \\ &= 1. \end{aligned}$$

Le coût amorti de chaque opération est donc toujours inférieur ou égal à 2. Dès lors, le théorème 6.9 nous dit qu'une suite de n opérations a un coût réel total

$$\sum c_i \leqslant \sum a_i \leqslant 2n$$

c'est-à-dire proportionnel au nombre n d'opérations. Dit autrement, tout se passe comme si chaque opération enqueue et dequeue avait un coût constant, et ce quel que soit l'enchaînement des opérations.

Variante

Une variante naturelle des piles et des files est une structure de données où les éléments peuvent être ajoutés et retirés *aux deux extrémités*. En anglais, on parle de *dequeue*, pour *double-ended queue*, ce que l'on peut traduire par *file à deux bouts*. Il est relativement facile d'adapter la structure de file utilisant un tableau (7.2.5.2) ou encore une paire de listes (7.2.5.3). Pour ce qui est de la liste chaînée (7.2.5.1), il suffit de la remplacer par une liste *doublement chaînée*.

7.2.6 Tables de hachage

Une *table de hachage* est une structure de données qui réalise un tableau associatif. Des valeurs y sont associées à des clés et on peut réaliser des opérations comme ajouter de nouvelles entrées dans la table, chercher la valeur associée à une clé donnée, ou encore supprimer la valeur associée à une clé. Le programme 7.18 donne l'interface d'un tel tableau associatif où les clés sont des chaînes de caractères. Comme on le comprend à la lecture de cette interface, il s'agit là d'une structure mutable.

Comme nous le verrons, la table de hachage est une structure extrêmement efficace, qu'il faut utiliser sans hésiter dès lors que c'est possible.

7.2.6.1 Principe

Le principe d'une table de hachage est simple. Si les clés étaient des entiers entre 0 et $m-1$, on utiliserait directement un tableau. D'où l'idée de se ramener à cette situation avec une fonction, dite *fonction de hachage*, qui envoie les clés vers des entiers entre 0 et $m-1$. En pratique, on se donne plutôt une fonction h : clé $\to \mathbb{Z}$ qui envoie les clés vers des entiers puis on réalise la table de hachage avec un tableau de taille m et on range l'entrée correspondant à la clé k dans la case $h(k) \pmod m$ du tableau.

On anticipe qu'il va être très difficile, voire impossible, de choisir l'entier m et la fonction h de façon à ce que chaque clé donne un indice différent modulo m par la fonction de hachage. En conséquence, on ne va pas chercher à garantir cette propriété d'injectivité. Si deux clés se voient attribuer la même case par la fonction de

Programme 7.18 – interface d'une table de hachage

Les clés sont ici des chaînes de caractères.

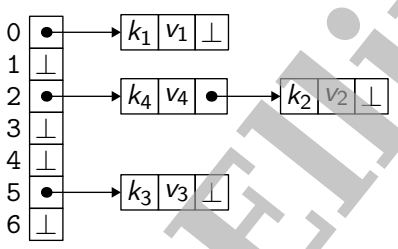
- En OCaml, les valeurs sont d'un type quelconque `'v`.

```
type 'v hashtable
val create: int -> 'v hashtable
val put: 'v hashtable -> string -> 'v -> unit
val contains: 'v hashtable -> string -> bool
val get: 'v hashtable -> string -> 'v
val remove: 'v hashtable -> string -> unit
```

- En C, les valeurs sont ici de type **int**.

```
typedef struct Hashtbl hashtbl;
hashtbl *hashtbl_create(int capacity);
void hashtbl_put(hashtbl *h, char *k, int v);
bool hashtbl_contains(hashtbl *h, char *k);
int hashtbl_get(hashtbl *h, char *k);
void hashtbl_remove(hashtbl *h, char *k);
void hashtbl_delete(hashtbl *h);
```

hachage — on parle de *collision* — alors elles iront toutes les deux dans cette case-là. Autrement dit, notre tableau contient dans chaque case non pas une entrée mais un *ensemble* d'entrées, qu'on appelle un *seau* (en anglais *bucket*). On peut choisir par exemple de réaliser les seaux par des listes simplement chaînées. Voici une illustration d'une table de hachage contenant quatre entrées (des clés k_1, k_2, k_3, k_4 associées à des valeurs v_1, v_2, v_3, v_4), rangées dans un tableau de taille $m = 7$.



On a ici deux clés en collision, à savoir k_2 et k_4, c'est-à-dire que $h(k_2) \equiv h(k_4) \pmod{m}$. Une image classique, mais utile, consiste à voir une table de hachage comme une commode contenant m tiroirs, la fonction de hachage envoyant nos vêtements vers chacun des tiroirs.

Pour chercher l'entrée correspondant à une clé k dans la table, il suffit de parcourir la liste rangée dans la case d'indice $h(k) \pmod{m}$ à la recherche de cette clé. Et pour ajouter une nouvelle entrée (k, v) dans la table, il suffit de l'ajouter à cette liste. Comme on le comprend, l'efficacité de la table de hachage va directement dépendre du choix de l'entier m et de la fonction h. Si l'entier m est petit devant le nombre d'entrées, les seaux seront longs et les opérations coûteuses. Et même si l'entier m est suffisamment grand, une mauvaise fonction de hachage pourrait provoquer de nombreuses collisions, ce qui ne changerait rien. Dans un cas extrême, toutes les entrées pourraient se retrouver dans le même seau et la table de hachage n'est alors pas différente d'une simple liste chaînée, ce qui n'est pas une façon efficace de réaliser un tableau associatif.

Néanmoins, nous allons voir qu'il n'est pas si difficile que cela de concevoir une fonction de hachage qui limite les collisions. Et pour ce qui est de la valeur de m, on la choisit de l'ordre de grandeur du nombre d'entrées, si on le connaît à l'avance. Dans le cas contraire, il suffit d'utiliser un tableau redimensionnable (voir section 7.2.2) plutôt qu'un tableau, en adaptant ainsi m dynamiquement au nombre d'entrées de la table.

7.2.6.2 Implémentation en OCaml

Écrivons une structure de table de hachage en OCaml pour des clés de type `string`. Notre objectif est donc de réaliser l'interface donnée dans le programme 7.18. Comme on l'a expliqué, la table de hachage est un tableau de seaux et chaque seau est une liste de paires (*clé*, *valeur*). On se donne donc le type suivant.

Cohérence entre hachage et comparaison des clés

Lorsque l'on construit une table de hachage pour des clés d'un type quelconque, on se donne une fonction de hachage h mais également une fonction de comparaison des clés ; appelons-la cmp. Il faut alors garantir la propriété suivante pour toute paire de clés x et y :

$$\text{si } cmp(x, y) \text{ est vrai, alors } h(x) = h(y)$$

Sans cette propriété, la recherche de la clé x dans une table contenant une entrée pour la clé y ne se ferait pas en examinant le bon seau.
Lorsque la comparaison des clés est tout simplement l'égalité, comme par exemple des chaînes de caractères comparées avec = en OCaml ou `strcmp` en C, cette propriété énonce tout simplement que la fonction de hachage doit être déterministe. De manière générale, il serait incorrect d'introduire du hasard dans la fonction de hachage.

```
type 'v hashtable = (string * 'v) list array
```

Ici, le type `'v` représente le type des valeurs, qui ne sera connu qu'à l'utilisation. Pour créer une table de hachage, il faut choisir une taille m pour ce tableau. Notre interface propose à l'utilisateur de fournir cette valeur.

```
let create m =
  if m <= 0 then invalid_arg "create";
  Array.make m []
```

On prend soin de garantir une taille strictement positive, car on s'apprête à calculer des indices modulo cette taille.

Pour écrire les opérations sur la table de hachage, il faut se donner maintenant une fonction de hachage sur les chaînes, c'est-à-dire une fonction `hash: string -> int`. Il y a bien une fonction de ce type qui existe déjà, à savoir `String.length`, mais ce serait une bien piètre fonction de hachage. Si on prend par exemple des chaînes qui sont des mots du dictionnaire français, cela veut dire que la valeur de dépassera jamais 25 et donc que l'on n'utilisera jamais plus de 25 seaux, et ce *quelle que soit la valeur de m*. Il nous faut donc une fonction de hachage qui donne de plus grandes valeurs. On pourrait par exemple imaginer faire l'addition des codes de tous les caractères de la chaîne. C'est déjà une meilleure fonction de hachage, mais ses valeurs restent relativement modestes sur les mots du dictionnaire. Avec 25 lettres au plus dans un mot, et en supposant des caractères dans l'encodage Latin-1, une majoration grossière nous donne au plus $25 \times 255 = 6375$ seaux. Cela reste petit en comparaison du nombre de mots dans le dictionnaire. Ainsi, il serait regrettable de se donner un tableau contenant $m = 10^5$ cases et de n'en uti-

liser que 6375. Par ailleurs, se contenter de faire la somme des codes des caractères a pour effet de donner la même valeur de hachage à toutes les permutations d'un même mot, et donc de les envoyer toutes dans le même seau.

Nous cherchons donc une fonction qui donne des valeurs relativement grandes et qui soit sensible à l'ordre des caractères. Par ailleurs, nous souhaitons autant que possible une fonction facile à calculer. Voici une fonction qui répond à ces trois critères :

$$h(s_0 s_1 \dots s_{n-1}) = \sum_{0 \leqslant i < n} 31^i \times \mathsf{code}(s_i)$$

Le choix de la constante 31 est relativement arbitraire ; nous y reviendrons plus loin. En particulier, la formule ci-dessus s'évalue facilement avec la *méthode de Horner*. Ainsi, on peut écrire le code suivant qui ne fait que n multiplications et $2n$ additions pour une chaîne de longueur n.

```
let hash k =
  let n = String.length k in
  let rec hash h i =
    if i = n then h else hash (31*h + Char.code k.[i]) (i+1) in
  hash 0 0
```

Le lecteur attentif aura remarqué que cette fonction ne calcule pas exactement la formule ci-dessus, substituant 31^{n-1-i} à 31^i. Cela n'a pas d'importance, cependant, car cela reste une fonction de hachage avec les propriétés recherchées. La chaîne vide est une clé parfaitement valable, pour laquelle la fonction `hash` renvoie 0.

Écrivons maintenant une fonction `bucket` qui calcule dans quel seau d'une table de hachage `h` se range l'entrée d'une certaine clé `k`. Intuitivement, il s'agit de la case `hash k`, calculée modulo la taille du tableau, c'est-à-dire modulo `Array.length h`. Il y a cependant une petite difficulté, car notre fonction `hash` peut renvoyer une valeur négative suite à un débordement arithmétique. C'est le cas par exemple de `hash "extraordinaire"`, dont la valeur exacte devrait être 2563735193583827804945 mais dont la valeur calculée est −362232661799869679. Or, il faut savoir que la fonction `mod` d'OCaml, qui n'est autre que celle de notre machine, renvoie une valeur dont le signe est celui de son premier argument. Dès lors, `(hash "extraordinaire") mod (Array.length h)` va renvoyer une valeur négative. Pour s'en prémunir, on efface le bit de signe de `hash k` avant de calculer le modulo.

Exercice 88 p.427

```
let bucket h k =
  let i = (hash k) land max_int in
  i mod (Array.length h)
```

Munis de cette fonction `bucket`, nous pouvons enfin écrire les opérations pour modifier et consulter la table de hachage. Commençons par la fonction `get` qui cherche dans une table de hachage `h` la valeur associée à une clé `k` donnée, la renvoie si elle existe et lève l'exception `Not_found` sinon. On l'écrit avec une fonction récursive locale `lookup` qui va parcourir le seau.

```
let get h k =
  let rec lookup b = match b with
```

Si le seau est vide, c'est qu'on est parvenu à son terme et on lève `Not_found`.

```
    | [] -> raise Not_found
```

Sinon, on compare la clé `k` avec la clé `k'` en tête du seau. En cas d'égalité, on renvoie la valeur `v` associé à `k'` ; sinon, on poursuit la recherche avec le reste du seau.

```
    | (k', v) :: b -> if k = k' then v else lookup b
```

Enfin, la fonction `get` appelle `lookup` sur le seau donné par la fonction `bucket` écrite précédemment.

```
  in
  lookup h.(bucket h k)
```

Exercice 91 p.427

Le programme 7.19 contient le code complet de nos tables de hachage, qui contient en outre des fonctions `put` et `contains` écrites sur le même principe. La fonction `remove` est laissée en exercice au lecteur.

Complexité. Si les clés ne sont pas des valeurs trop grosses, on peut considérer que le calcul de la fonction `hash`, et donc de la fonction `bucket`, se fait en temps constant. Dès lors, le coût d'une opération `put`, `contains` ou `get` est dans le pire des cas proportionnel à la longueur du seau. C'est donc la quantité qui nous intéresse.

D'une manière générale, il est très difficile de majorer la longueur des seaux dans une table de hachage. Cela dépend de la fonction de hachage choisie, de l'ensemble des clés utilisées et enfin de la taille m du tableau. En jouant sur un ou plusieurs de ces paramètres, on peut aboutir à des situations extrêmes où un seau se retrouve contenir un grand nombre d'éléments, voire tous les éléments. C'est notamment le cas si la fonction de hachage est constante ou donne toujours le même entier modulo m. Inversement, pour une fonction de hachage donnée, on pourrait chercher à forger un ensemble de clés qui ont toutes la même valeur pour cette fonction. C'est pourquoi nous allons nous contenter d'une évaluation empirique de nos tables de hachage.

Exercice 89 p.427

Programme 7.19 – tables de hachage en OCaml

```
type 'v hashtable = (string * 'v) list array

let create (m: int) : 'v hashtable =
  if m <= 0 then invalid_arg "create";
  Array.make m []

let hash (k: string) : int =
  let n = String.length k in
  let rec hash h i =
    if i = n then h else hash (31*h + Char.code k.[i]) (i+1) in
  hash 0 0

let bucket (h: 'v hashtable) (k: string) : int =
  let i = (hash k) land max_int in (* on garantit i >= 0 *)
  i mod (Array.length h)

let put (h: 'v hashtable) (k: string) (v: 'v) : unit =
  let rec update b = match b with
    | [] -> [k, v]
    | (k', _) :: b when k = k' -> (k, v) :: b
    | e :: b -> e :: update b in
  let i = bucket h k in
  h.(i) <- update h.(i)

let contains (h: 'v hashtable) (k: string) : bool =
  let rec lookup b = match b with
    | [] -> false
    | (k', _) :: b -> k = k' || lookup b in
  lookup h.(bucket h k)

let get (h: 'v hashtable) (k: string) : 'v =
  let rec lookup b = match b with
    | [] -> raise Not_found
    | (k', v) :: b -> if k = k' then v else lookup b in
  lookup h.(bucket h k)
```

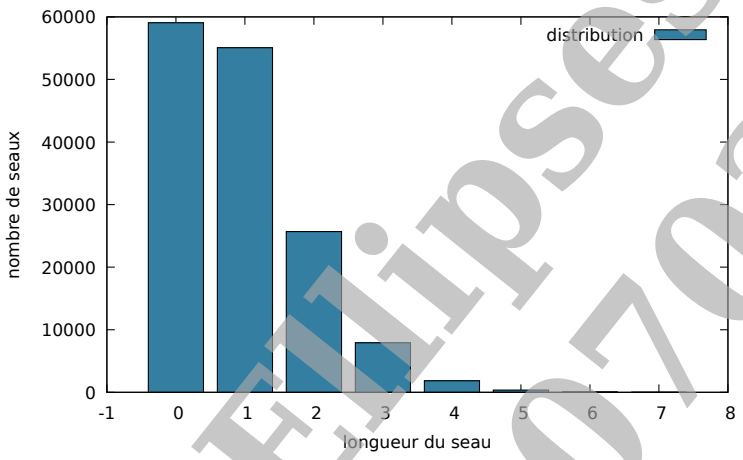


Figure 7.3 – Longueur des seaux d'une table de hachage.

Évaluation expérimentale. Bien que très simple, notre fonction de hachage de chaînes de caractères donne de très bons résultats en pratique. Si on prend par exemple les 139 719 mots contenus dans le dictionnaire français `/usr/share/dict/french` sous Linux, alors notre fonction de hachage ne donne *que trois collisions*, de deux chaînes chacune, à savoir

- `hash "n'" = hash "le" = 3449`;
- `hash "fig." = hash "fiel" = 3142826`;
- `hash "l'" = hash "je" = 3387`.

Si on avait pris en revanche la constante 10 plutôt que la constante 31 dans le code de la fonction `hash`, on aurait eu alors 942 collisions, dont 937 collisions de deux chaînes et 5 collisions de trois chaînes (comme par exemple `hash "retiens" = hash "ressens" = hash "pythons" = 125376315`).

Prêtons-nous maintenant à une autre expérience avec notre dictionnaire des mots français. Ajoutons tous les mots comme autant de clés dans une table de hachage dont la capacité est de 150 000, c'est-à-dire de l'ordre de grandeur du nombre total de mots. La figure 7.3 contient un histogramme montrant la distribution de la longueur des seaux de notre table de hachage après l'insertion de tous les mots. Il y a plusieurs constatations à faire :

- près de 60 000 seaux sont vides ;
- parmi les seaux utilisés, une majorité (plus de 55 000) ne contiennent qu'une seule entrée ;
- aucun seau ne contient plus de 7 entrées.

On peut être déçus que tant de seaux ne soient pas utilisés, mais l'information principale est que la recherche (ou l'ajout ou la suppression) dans une telle table ne demandera jamais plus que la comparaison avec sept autres clés, et le plus souvent même avec aucune ou une seule clé.

Le module `Hashtbl`. La bibliothèque standard d'OCaml fournit des tables de hachage dans un module `Hashtbl`, avec une implémentation tout à fait similaire à celle décrite dans cette section. Ces tables sont polymorphes en les clés et les valeurs. Elles utilisent pour cela une fonction de hachage prédéfinie par OCaml, qui s'applique à une valeur de n'importe quel type. Pour les curieux, c'est la fonction `Hashtbl.hash`. À noter que cette fonction donne 9 collisions de 2 chaînes sur les mots du dictionnaire français, là où notre fonction de hachage ne donne que 2 collisions. En contrepartie, la fonction `Hashtbl.hash` a d'autres propriétés, comme le fait de donner de plus grandes valeurs ou encore le fait de s'appliquer à des clés d'un type quelconque.

Lorsque le taux de remplissage de la table de hachage devient trop important, le module `Hashtbl` double la capacité du tableau, puis y réinsère toutes les entrées. Le coût de ce redimensionnement s'amortit sur l'ensemble des opérations, exactement comme nous l'avons expliqué pour les tableaux redimensionnables dans la section 7.2.2.

Nous utiliserons abondamment le module `Hashtbl` dans le reste de cet ouvrage, notamment pour construire des arbres préfixes dans la section 7.3.5, puis dans le chapitre 9.

Listes d'association

Nos seaux sont des listes de paires (*clé*, *valeur*). On appelle cela une *liste d'association*. Le module `List` de la bibliothèque OCaml fournit quelques opérations sur les listes d'association (`assoc`, `mem_assoc`, `remove_assoc`) et nous aurions pu les utiliser pour simplifier le code de nos tables de hachage. En soi, les listes d'association ne constituent pas une structure très efficace pour réaliser un tableau associatif, car les opérations (chercher ou mettre à jour) ont un coût proportionnel au nombre d'entrées. Mais pour réaliser nos seaux, qui contiennent très peu d'entrées la plupart du temps, cela convient parfaitement.

7.2.6.3 Implémentation en C

Les programmes 7.20 et 7.21 contiennent une implémentation en C des tables de hachage. Dans les grandes lignes, l'implémentation est semblable à celle que nous avons faite en OCaml dans la section précédente. Les clés sont toujours des chaînes de caractères et la fonction de hachage est la même. Il y a cependant quelques différences notables entre les codes OCaml et C.

Programme 7.20 – tables de hachage en C (1/2)

```c
typedef struct Entry {
  char *key;
  int value;
  struct Entry *next;
} entry;

typedef struct Hashtbl {
  int capacity; // 0 < capacity
  int size;     // nombre d'entrées dans la table
  entry **data;  // tableau de taille capacity
} hashtbl;

hashtbl *hashtbl_create(int capacity) {
  assert(0 < capacity);
  hashtbl *h = malloc(sizeof(struct Hashtbl));
  h->capacity = capacity;
  h->data = calloc(capacity, sizeof(entry*));
  for (int i = 0; i < capacity; i++) {
    h->data[i] = NULL;
  }
  h->size = 0;
  return h;
}

int hash(char *k) {
  int h = 0;
  char c;
  while ((c = *k++) != 0) {
    h = 31 * h + c;
  }
  return h;
}

int hashtbl_bucket(hashtbl *h, char *k) {
  int i = hash(k) & INT_MAX; // s'assurer que i >= 0
  return i % h->capacity;    // avant de prendre le modulo
}
```

Programme 7.21 – tables de hachage en C (2/2)

```
entry *hashtbl_find_entry(entry *e, char *k) {
  while (e != NULL) {
    if (strcmp(k, e->key) == 0) {
      return e;
    }
    e = e->next;
  }
  return NULL;
}
entry *create_entry(char *k, int v, entry *n) {
  entry *e = malloc(sizeof(struct Entry));
  e->key = k;
  e->value = v;
  e->next = n;
  return e;
}

void hashtbl_put(hashtbl *h, char *k, int v) {
  int b = hashtbl_bucket(h, k);
  entry *e = hashtbl_find_entry(h->data[b], k);
  if (e == NULL) {
    h->data[b] = create_entry(k, v, h->data[b]);
    h->size++;
  } else {
    e->value = v;
  }
}
bool hashtbl_contains(hashtbl *h, char *k) {
  int b = hashtbl_bucket(h, k);
  return hashtbl_find_entry(h->data[b], k) != NULL;
}
int hashtbl_get(hashtbl *h, char *k) {
  int b = hashtbl_bucket(h, k);
  entry *e = hashtbl_find_entry(h->data[b], k);
  if (e == NULL) return 0;
  return e->value;
}
```

- On introduit une structure `Entry` pour représenter les seaux. Il s'agit d'une liste simplement chaînée (champ `next`), dont les éléments sont des paires (champs `key` et `value`).
- Les seaux sont mutables, là où ils étaient immuables en OCaml. En particulier, la fonction `hashtbl_put` peut modifier le seau en place lorsque la clé est déjà présente, là où le code OCaml reconstruit un nouveau seau.
- Le code est organisé un peu différemment, avec une fonction `hashtbl_find_entry` pour chercher et renvoyer une entrée dans un seau.
- Dans le code C, l'ajout d'une nouvelle entrée se fait en tête de seau, là où il se fait en fin de seau pour le code OCaml. La complexité est la même dans les deux cas, dans la mesure où on commence toujours par parcourir tout le seau avant de faire une nouvelle entrée dans la table. En revanche, cette différence pourrait impacter la performance de recherches futures. Mais comme on l'a expliqué plus haut, on peut espérer en pratique des seaux très petits et donc très peu d'impact.

Exercice
91 p.427
92 p.428

L'exercice 91 propose d'ajouter à ce code C une fonction pour supprimer une entrée de la table de hachage. L'exercice 92 explore une autre solution au problème des collisions.

7.3 Structures de données hiérarchiques

Comme nous l'avons expliqué dans la section précédente, la table de hachage constitue une excellente structure de données, avec laquelle on réalise facilement et efficacement des tableaux associatifs. Il existe cependant des situations où la table de hachage n'est pas une solution. Ainsi, on peut avoir besoin d'une structure immuable, ce que la table de hachage, construite sur un tableau, ne propose pas. De même, on peut stocker des éléments totalement ordonnés dans une structure et vouloir ensuite retrouver le n-ième élément, ou encore tous les éléments compris entre deux valeurs particulières, ce que là encore la table de hachage ne permet pas de faire. Dans cette section, nous introduisons des *structures arborescentes* pour apporter des solutions satisfaisantes à ces différentes questions.

7.3.1 Arbres binaires

Parmi les structures arborescentes, les arbres binaires occupent une place importante et pour cette raison nous les étudions en premier lieu. Nous verrons plus loin d'autres structures arborescentes (sections 7.3.4 et 7.3.5).