Informatique *Cours*
Structures de données - Pt 2 Ipesup

Objectifs

À l'issue de cette leçon, l'étudiant doit être capable de :
- 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.

1 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.

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 dessiné à gauche, contient l'élément x_0 qui a été ajouté en premier dans la pile et qui sera le dernier à en sortir.

Le programme suivante 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.

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);
```

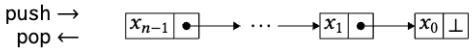
— En OCaml, pour des éléments d'un type `'a`.

```
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
```

On va maintenant proposer différentes réalisations de cette interface.

1.1 Implémentation avec une liste chaînée

Il est immédiat de réaliser une pile avec une liste simplement chaînée : le sommet de la pile correspond à la tête de la liste, et le fond de la pile correspond au dernier élément de la liste.



En C.

Pile réalisée en C avec une liste chaînée

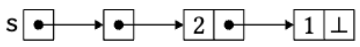
```
typedef struct Stack {
    list *head;
} stack;

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);
}
```

Le programme 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 en C, il suffit de placer la liste dans une référence. Le programme suivant en décrit l'implémentation.

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
```

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 :

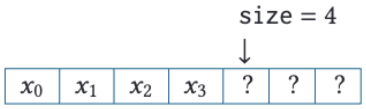
```
type 'a t = { mutable c : 'a list; mutable len : int }
```

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.

1.2 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`).

Le programme suivante donne une réalisation en C de cette idée.

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;
}
```

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. Il n'est même pas nécessaire d'introduire une nouvelle structure, car un tableau redimensionnable est une pile. Le programme suivante définit des opérations pour manipuler un tableau redimensionnable comme une pile.

Pile réalisée en C avec un tableau redimensionnable

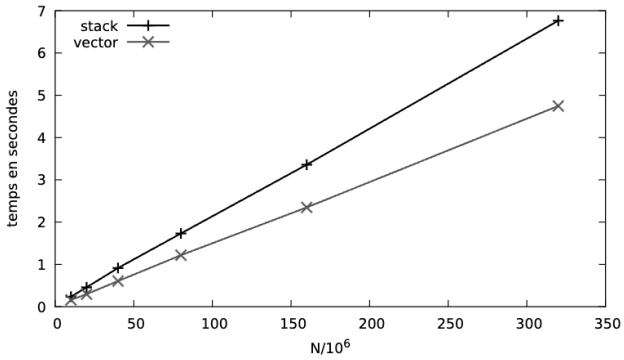
```
void vector_push(vector *v, int x) {
  int n = vector_size(v);
  vector_resize(v, n + 1);
  vector_set(v, n, x);
}

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;
}
```

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 leçon precedente 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.

1.3 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 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 inutilisée, 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.

2 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 suivante contient l'interface C d'une telle structure de file (en anglais, on parle de *queue*).

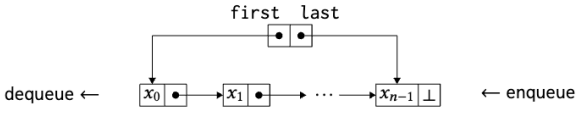
Interface C d'une file contenant des entiers

```
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);
```

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.

2.1 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 direct 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 direct au premier élément avec le pointeur `first`. Lorsque la file est vide, les deux pointeurs `first` et `last` sont nuls.

Le programme suivante contient un code C qui réalise cette idée.

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;
}

```

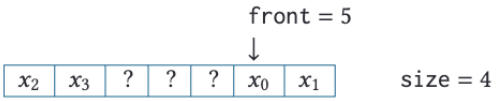
```
int queue_size(queue *q) {
    return q->size;
}

bool queue_is_empty(queue *q) {
    return queue_size(q) == 0;
}

void queue_enqueue(queue *q, int x) {
    if (queue_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;
}
```

Il réutilise le type `list` des listes simplement chaînées du programme. 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 en C.

2.2 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. 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`.

Le programme suivante contient une structure C pour mettre en œuvre cette idée.

Files mutables en C dans un tableau circulaire

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

Il peut être intéressant d'ajouter à notre interface une fonction pour déterminer si la file est pleine, c'est-à-dire

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

2.3 Files immuables

Les deux structures de files que nous venons de voir sont des structures mutables : la liste chaînée ou le tableau sont modifié en place lorsque des éléments sont ajoutés ou retirés de la file. Pour les files immuables, les opérations ne modifient pas la file mais renvoient de nouvelles files.

Le programme suivante présente une interface OCaml de files immuables polymorphes.

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
```

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 de réaliser de telles files. On a 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.

Le programme suivante contient un code OCaml qui met en œuvre cette idée.

Files immuables en OCaml

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

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
```

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[1]. 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.

3 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 suivante donne l'interface d'un tel tableau associatif où les clés sont des chaînes de caractères.

1. Mais on pourrait tout à fait utiliser la même idée avec deux piles mutables ; on obtiendrait alors une file mutable.

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);
```

Comme on le comprend à la lecture de cette interface, il s'agit 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.

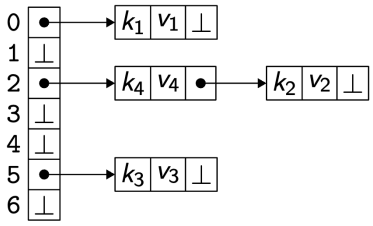
3.1 Principe

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 $h(k) \bmod m$ 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 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 plutôt qu'un tableau, en adaptant ainsi m dynamiquement au nombre d'entrées de la table.

3.2 Implémentation en OCaml

Écrivons une structure de table de hachage en OCaml pour des clés de type `string`.

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.

```
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 ne 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 utiliser 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 :

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

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 `256373519353882780494 5` 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.

```
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.

```
let get h k =
  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)
```

Le programme suivante contient le code complet de nos tables de hachage, qui contient en outre des fonctions `put` et `contains` écrites sur le même principe.

Tables de hachage en OCaml

```
1  type 'v hashtable = (string * 'v) list array
2
3  let create (m : int) : 'v hashtable =
4    if m <= 0 then invalid_arg "create";
5    Array.make m []
6
7  let hash (k : string) : int =
8    let n = String.length k in
9    let rec hash h i =
10     if i = n then h
11     else hash (31 * h + Char.code k.[i]) (i + 1)
12   in
13   hash 0 0
14
15 let bucket (h : 'v hashtable) (k : string) : int =
16   let i = (hash k) land max_int in (* on garantit i >= 0 *)
17   i mod (Array.length h)
18
19 let put (h : 'v hashtable) (k : string) (v : 'v) : unit =
20   let rec update b =
21     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)
```

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.

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.

C'est pourquoi nous allons nous contenter d'une évaluation empirique de nos tables de hachage.

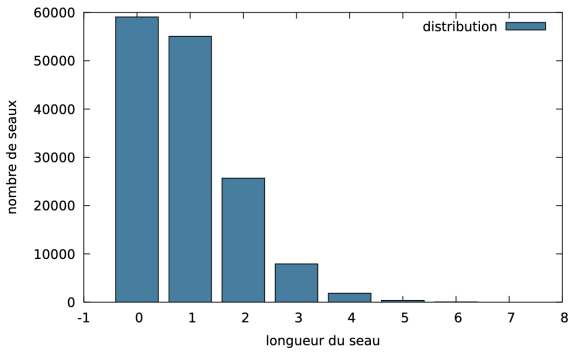
Évaluation expérimentale

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 ici 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 (`Hashtbl.hash`), qui s'applique à une valeur de n'importe quel type. 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.

Nous utiliserons abondamment le module `Hashtbl` pour construire des arbres préfixes.

3.3 Implémentation en C

Le programmes suivante contiennent une implémentation en C des tables de hachage.

Tables de hachage en 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
}

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;
}
```

Dans les grandes lignes, l'implémentation est proche de celle présentée en OCaml dans la section précédente. On observe toutefois plusieurs différences notables entre les versions OCaml et C :

— Une structure `Entry` est introduite pour représenter les seaux. Chaque seau est une liste simplement chaînée (champ `next`) dont les éléments sont des paires clé–valeur (champs `key` et `value`).
— Les seaux sont mutables en C, alors qu'ils étaient immuables en OCaml. Ainsi, la fonction `hashtbl_put` peut modifier un seau en place lorsque la clé est déjà présente, tandis que le code OCaml reconstruit un nouveau seau.
— L'organisation du code diffère légèrement, notamment avec l'introduction de la fonction `hashtbl_find_entry`, chargée de rechercher et de renvoyer une entrée dans un seau.
— En C, l'insertion d'une nouvelle entrée se fait en tête de seau, alors qu'elle se fait en fin de seau en OCaml. La complexité reste identique dans les deux cas, puisqu'on parcourt

toujours l'intégralité du seau avant l'insertion. Cette différence peut toutefois influencer les performances des recherches ultérieures. Néanmoins, comme indiqué précédemment, on peut s'attendre en pratique à des seaux très courts, rendant cet impact négligeable.