

# Table de hachage et dictionnaire

Quentin Fortier

November 1, 2021

On peut voir un tableau  $t$  comme une fonction qui à chaque indice  $i$  associe  $t.(i)$

Les indices sont forcément des entiers consécutifs à partir de 0.

Un **dictionnaire** est une structure de donnée plus générale qui à chaque **clé** associe une **valeur**. Il possède les opérations suivantes :

- Ajouter une association (clé, valeur)
- Supprimer une association (clé, valeur)
- Obtenir les valeurs associée à une clé donnée

# Dictionnaire : en Python

```
D = {  
    "blue": (0, 255, 0),  
    "yellow": (255, 255, 0)  
}  
  
D["blue"]  
# donne la valeur (0, 255, 0) associée à la clé "blue"  
  
D["white"] = (255, 255, 255)  
# ajoute une clé "white" de valeur (255, 255, 255)  
  
"yellow" in D # donne True  
  
for k in D: # affiche tous les couples (clé, valeur)  
    print(k, D[k])
```

Les dictionnaires Python sont implémentés par table de hachage.

# Dictionnaire : implémentations

Il y a plusieurs implémentations possibles de dictionnaire :

Opération	Liste de couples	ABR équilibré	Table de hachage
ajouter	$O(1)$	$O(\log(n))$	$O(1)$ en moyenne
supprimer	$O(n)$	$O(\log(n))$	$O(1)$ en moyenne
rechercher	$O(n)$	$O(\log(n))$	$O(1)$ en moyenne

On verra les ABR (arbres binaires de recherche) plus tard.

# Dictionnaire : implémentation par liste de couples

On pourrait implémenter un dictionnaire avec une liste de couples (clé, valeur).

```
(* exemple de dictionnaire avec une liste de couples *)  
let d = [("red", (255, 0, 0)); ("black", (0, 0, 0))]
```

```
let add c l = c::l;;  
  
let del k l =  
  List.filter (fun c -> fst c <> k) l;;  
  
let get k l =  
  List.filter (fun c -> fst c = k) l  
  |> List.map snd;;
```

|> est le symbole de *pipe* :  $e \mid\!> f$  est équivalent à  $f\ e$ .

# Dictionnaire : type abstrait

Type abstrait impératif de dictionnaire en OCaml :

```
type ('k, 'v) dict = {  
  add : 'k * 'v -> unit;  
  del : 'k -> unit;  
  get : 'k -> 'v option  
}
```

```
type 'a option = None | Some of 'a
```

get k renvoie **None** si k n'est pas dans le dictionnaire.

## Exercice

Écrire une fonction `frequent` pour déterminer en  $O(n)$  l'élément apparaissant le plus souvent dans une liste de taille  $n$ .

Idee : utiliser un dictionnaire où les clés sont les éléments du tableau et les valeurs sont les fréquences.

Complexité :  $O(n)$  appels à `get` et `add`, ce qui donne  $O(n)$  avec table de hachage

## Exercice

Écrire une fonction `frequent` : `int array -> int` pour déterminer en  $O(n)$  l'élément apparaissant le plus souvent dans une liste de taille  $n$ .

```
let frequent t d = (* d : dict utilisé dans l'algo *)
  let maxi = ref t.(0) in
  let freq_maxi = ref 0 in
  for i = 0 to Array.length t - 1 do
    let freq = match d.get t.(i) with
      | None -> d.add (t.(i), 1); 1
      | Some f -> d.add (t.(i), f + 1); f + 1 in
    if !freq_maxi < freq
    then (maxi := t.(i); freq_maxi := freq)
  done;
  !maxi
```



# Table de hachage

Une table de hachage est constituée:

- 1 d'un **tableau** (dynamique)  $t$  contenant les valeurs
- 2 d'une **fonction de hachage**  $h$ , de l'ensemble des clés vers les indices de  $t$

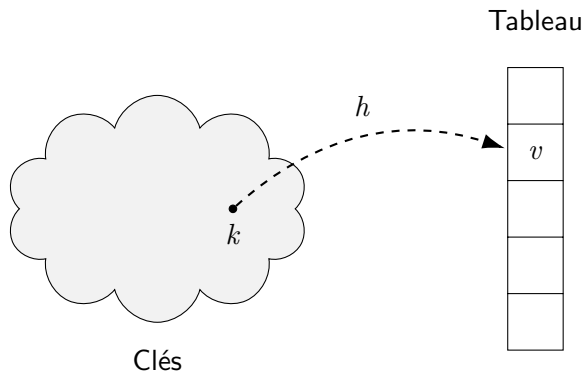
La valeur associée à une clé  $k$  est stockée à l'indice  $h(k)$  du tableau  $t$ .

Si la même clé est associée à plusieurs valeurs alors  $t$  doit être un tableau de listes.

Si les clés sont des entiers, on peut choisir  $h : x \mapsto x \bmod n$ .

Sous quelques hypothèses, on peut montrer que les opérations de table de hachage sont en complexité moyenne  $O(1)$ .

# Table de hachage



$h$  associe à chaque clé  $k$  un indice du tableau, dans lequel est stockée la valeur associée à  $k$

## Table de hachage

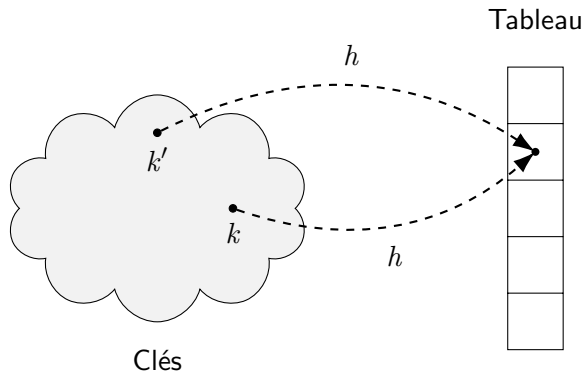
```
type ('k, 'v) hashtable = {  
    t : ('k * 'v) option array;  
    h : 'k -> int  
};;  
  
let hashtable_add ht (k, v) =  
    ht.t.(ht.h k) <- Some v;;  
  
let hashtable_get ht k =  
    ht.t.(ht.h k);;  
  
let hashtable_del ht k =  
    ht.t.(ht.h k) <- None;;
```

## Table de hachage : implémentation de dict

```
type ('k, 'v) dict = {  
  add : 'k * 'v -> unit;  
  del : 'k -> unit;  
  get : 'k -> 'v option
```

```
(* n est la taille du tableau à utiliser *)  
let dict_of_hashtable n =  
  let ht = {  
    t = Array.make n None;  
    h = fun k -> k mod n  
  } in {  
    add = hashtable_add ht;  
    get = hashtable_get ht;  
    del = hashtable_del ht  
  }  
}
```

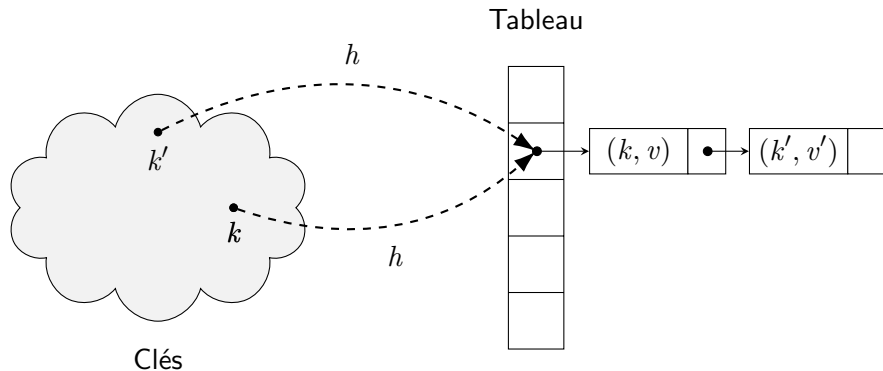
# Table de hachage : collision



Si deux clés ont le même hash, il y a un **collision**.

Si le nombre de clés est supérieur à la taille du tableau, il y a forcément collision.

# Table de hachage : collision



On peut résoudre ce problème en stockant une liste des couples (clé, valeur) dans chaque case du tableau

# Ensemble

Un **ensemble** (set) est une structure de donnée avec 3 opérations :

- add : ajoute un élément à l'ensemble
- del : supprime un élément de l'ensemble
- has : teste si un élément est dans l'ensemble

Type abstrait :

```
type 'a set = {  
  add : 'a -> unit;  
  del : 'a -> unit;  
  has : 'a -> bool  
}
```

# Ensemble

```
type ('k, 'v) dict = {  
  add : 'k * 'v -> unit;  
  del : 'k -> unit;  
  get : 'k -> 'v option  
}
```

On peut implémenter un set avec un dict, en utilisant que les clés :

```
let set_of_dict d = {  
  add = (fun e -> d.add (e, 0));  
  del = (fun e -> d.del e);  
  get = (fun e -> match d.get e with  
    | None -> false  
    | Some _ -> true)  
}
```

Avec table de hachage les opérations d'ensemble sont en  $O(1)$  en moyenne.



## Exercice

Expliquer comment déterminer si une liste contient un doublon, en utilisant un set.