Conteneurs

G. Dewaele

Conteneurs mutables

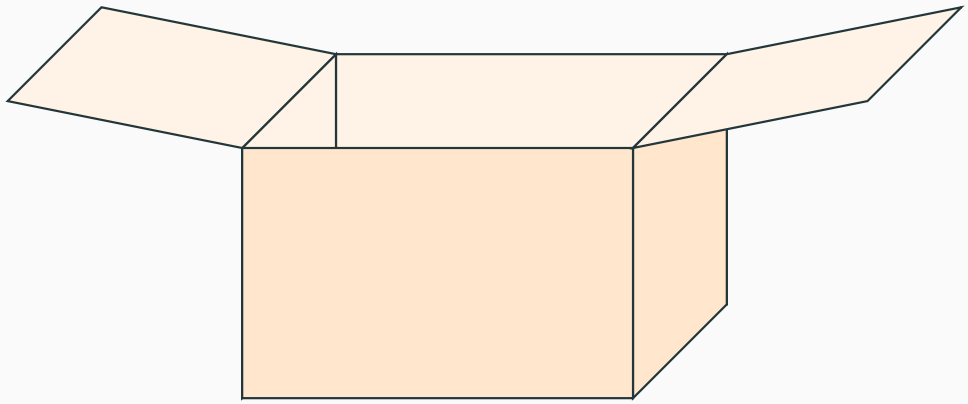
Nous avons vu trois objets mutables en OCaml :

- les références (`'a ref`)
- les tableaux (`'a array`)
- les champs mutables des types enregistrements

Ces structures contiennent un nombre *prédéfini* d'éléments

Conteneurs mutables

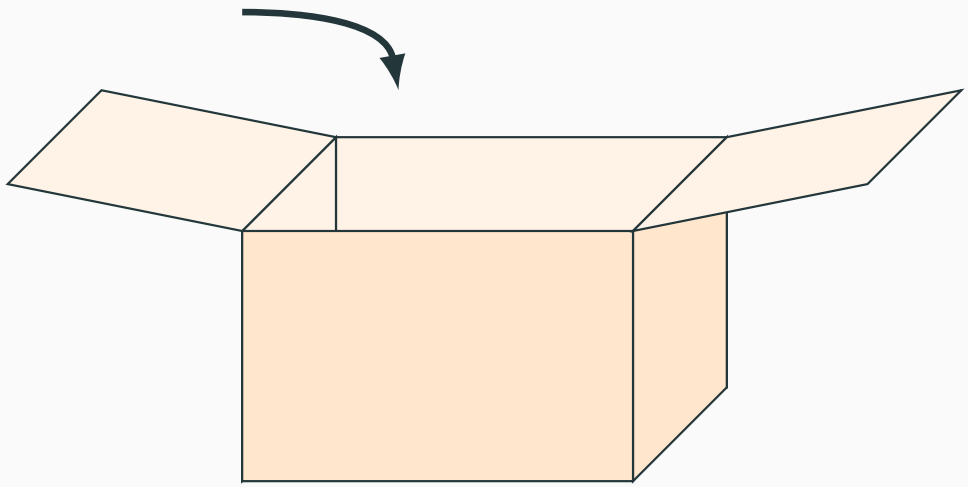
On ne dispose pas, pour l'instant, de stockage qui permet



Conteneurs mutables

On ne dispose pas, pour l'instant, de stockage qui permet

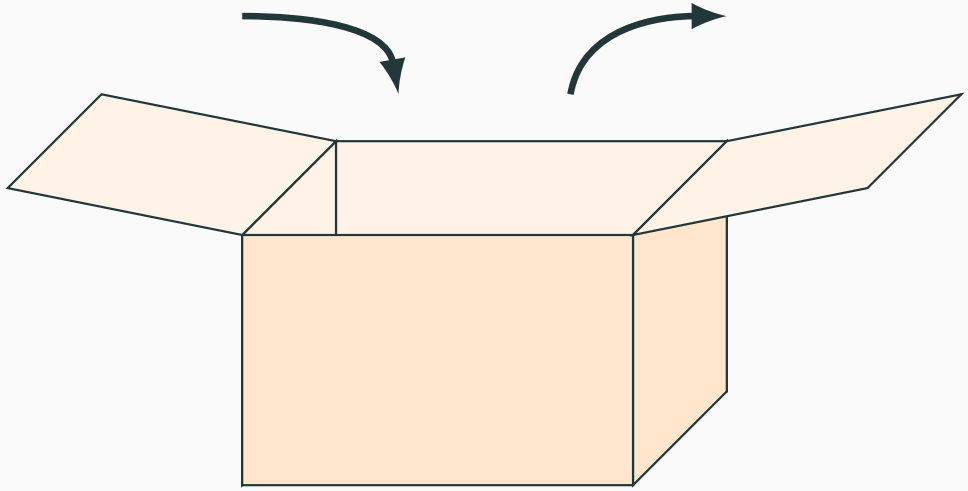
- l'ajout d'éléments



Conteneurs mutables

On ne dispose pas, pour l'instant, de stockage qui permet

- l'ajout d'éléments
- le retrait d'éléments



Conteneurs mutables

C'est pourtant très utile !

On peut stocker :

- des objets à traiter
- des tâches à faire...

Conteneurs mutables

La question est de savoir dans quel ordre les éléments ressortent...

- Dans l'ordre d'insertion ?
- Le plus récent ajout d'abord ?
- Selon un critère d'« importance » ?
- Grâce à un mécanisme permettant de les retrouver ?

Les piles

Les conteneurs « LIFO » sont appelés *piles*

(Last In is First Out, le dernier entré est le premier à sortir)

Les piles

Les conteneurs « LIFO » sont appelés *piles*

(Last In is First Out, le dernier entré est le premier à sortir)

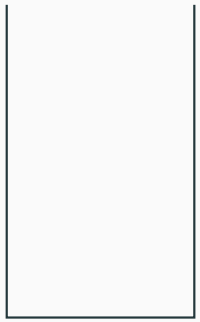
Les piles

On doit disposer de fonctions élémentaires :

Les piles

On doit disposer de fonctions élémentaires :

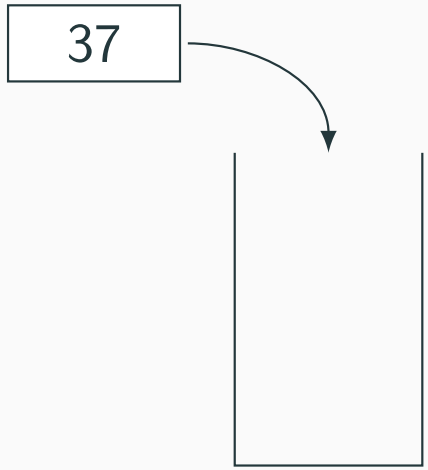
- créer une pile vide



Les piles

On doit disposer de fonctions élémentaires :

- créer une pile vide
- ajouter des éléments (*push*)



Les piles

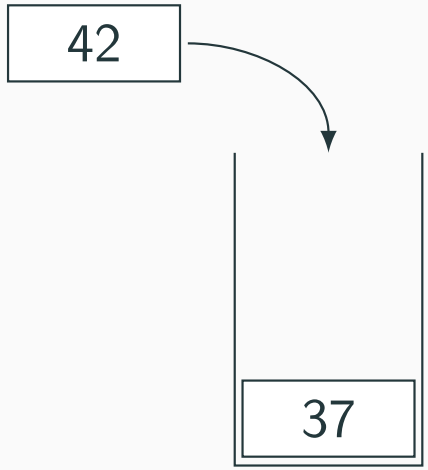
On doit disposer de fonctions élémentaires :

- créer une pile vide
- ajouter des éléments (*push*)

Les piles

On doit disposer de fonctions élémentaires :

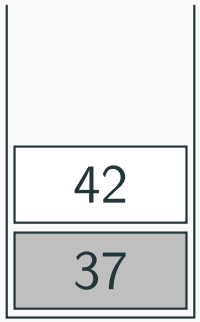
- créer une pile vide
- ajouter des éléments (*push*)



Les piles

On doit disposer de fonctions élémentaires :

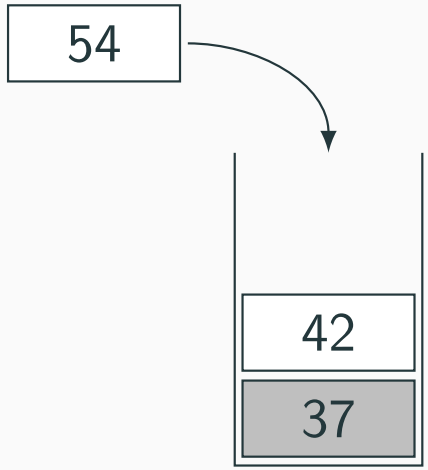
- créer une pile vide
- ajouter des éléments (*push*)



Les piles

On doit disposer de fonctions élémentaires :

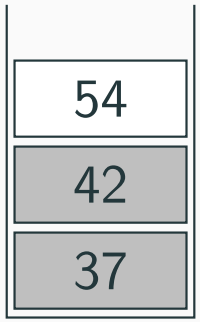
- créer une pile vide
- ajouter des éléments (*push*)



Les piles

On doit disposer de fonctions élémentaires :

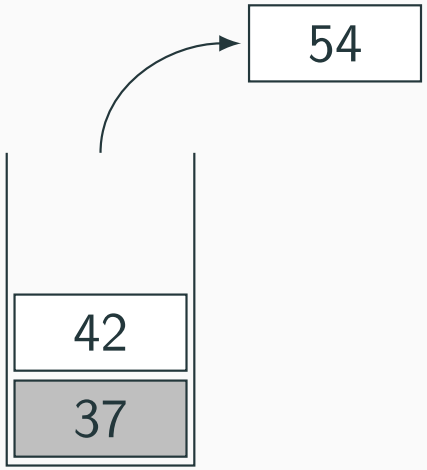
- créer une pile vide
- ajouter des éléments (*push*)



Les piles

On doit disposer de fonctions élémentaires :

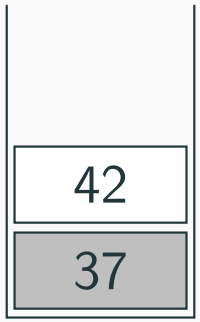
- créer une pile vide
- ajouter des éléments (*push*)
- retirer des éléments (*pop*)



Les piles

On doit disposer de fonctions élémentaires :

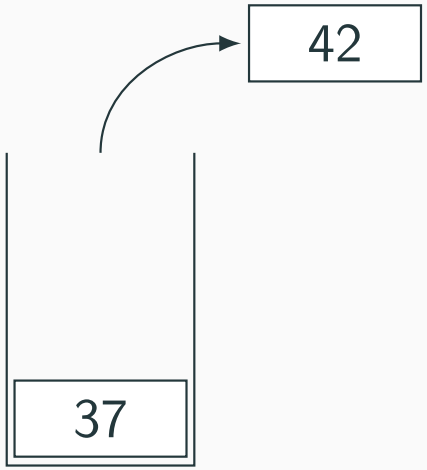
- créer une pile vide
- ajouter des éléments (*push*)
- retirer des éléments (*pop*)



Les piles

On doit disposer de fonctions élémentaires :

- créer une pile vide
- ajouter des éléments (*push*)
- retirer des éléments (*pop*)



Les piles

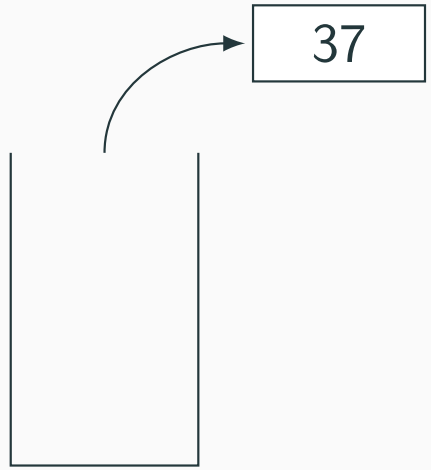
On doit disposer de fonctions élémentaires :

- créer une pile vide
- ajouter des éléments (*push*)
- retirer des éléments (*pop*)

Les piles

On doit disposer de fonctions élémentaires :

- créer une pile vide
- ajouter des éléments (*push*)
- retirer des éléments (*pop*)



Le module stack

Les fonctions utiles sont dans un module appelé `Stack`

- `Stack.create` `unit -> 'a Stack.t`
- `Stack.push` `'a -> 'a Stack.t -> unit`
- `Stack.pop` `'a Stack.t -> 'a`
- `Stack.top` `'a Stack.t -> 'a`
- `Stack.is_empy` `'a Stack.t -> bool`

Les exceptions

En Caml, on ne peut normalement pas sortir d'une boucle **`for`**

Une solution est de lever une exception...

... et de la rattraper hors de la boucle !

À utiliser avec parcimonie et pertinence

Les exceptions

Par exemple, pour un équivalent de `Array.mem` en impératif :

```
# exception Found;;

# let array_mem elem arr =
    try
      for i = 0 to Array.length arr - 1 do
        if arr.(i) = elem then raise Found
      done;
      false (* Retourne false à la fin de la boucle *)
    with
      | Found -> true;;  (* retourne true si trouvé *)

val array_mem : 'a -> 'a array -> bool = <fun>
```

Les exceptions

Les exceptions peuvent « transporter » des données

Une exception transportant un entier est définie par :

```
# exception MyException of int;;
```

Les exceptions

Pour localiser la première occurence d'un élément dans un tableau (la fonction retourne -1 s'il est absent)


```
# exception Position of int;;

# let position elem arr =
    try
      for i = 0 to Array.length arr - 1 do
        if arr.(i) = elem then raise (Position i);
      done;
      -1    (* L'élément n'est pas présent dans tab *)
    with
      | Position k -> k;;   (* Retourne la position *)

val position : 'a -> 'a array -> int = <fun>
```

Les exceptions

On peut évidemment faire sans exceptions avec un while :


```
# let array_mem elem arr =
    let found = ref false
    and i = ref 0 in
      while not !found && !i < Array.length arr do
        if arr.(!i) = elem then found := true;
        incr i
      done;
      !found;;

val array_mem : 'a -> 'a array -> bool = <fun>
```

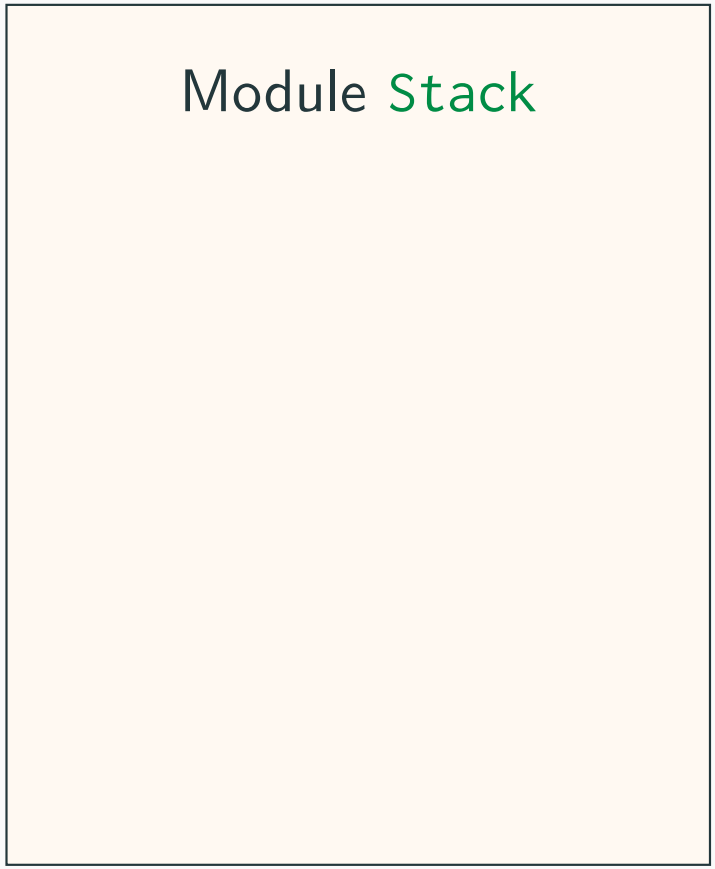
Les exceptions

On peut évidemment faire sans exceptions avec un while :

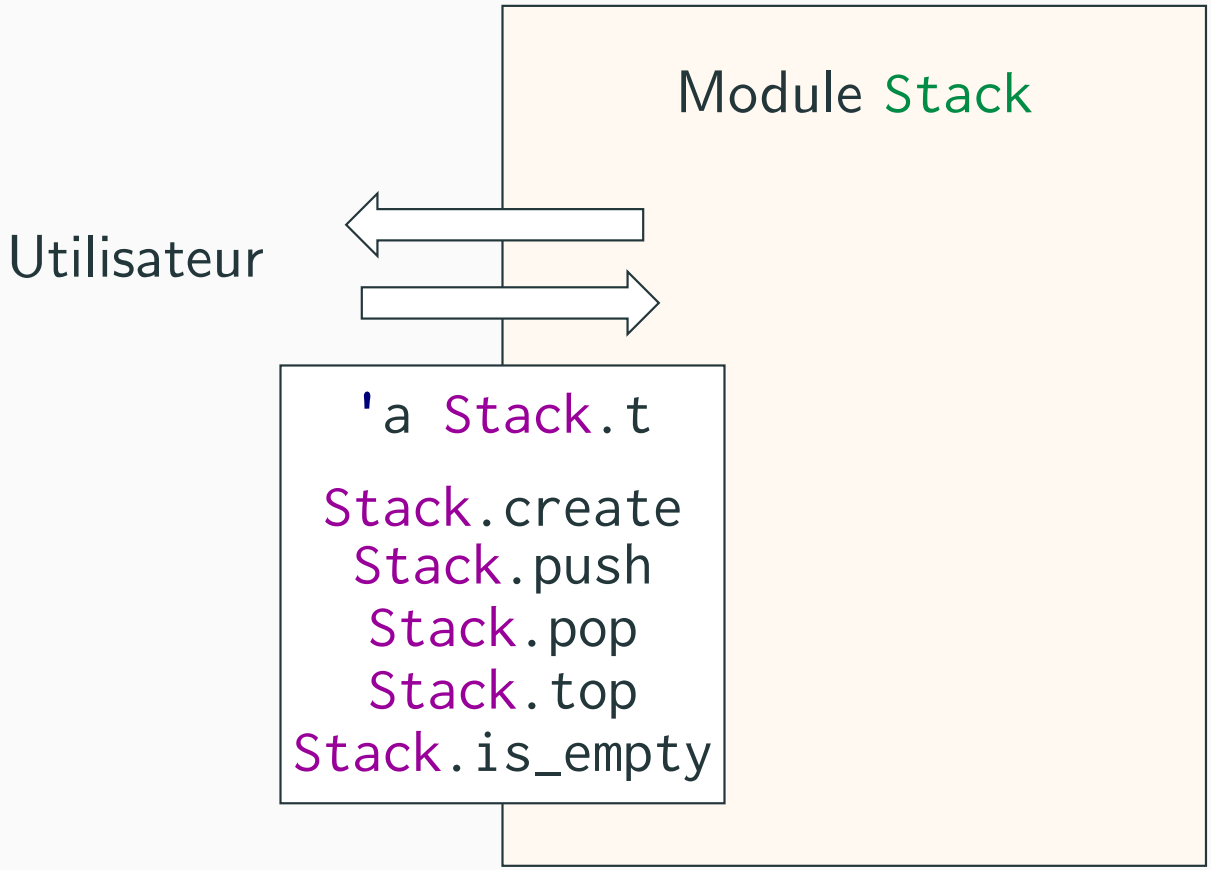

```
# let position elem arr =
    let pos = ref (-1)
    and i = ref 0 in
      while !pos = -1 && !i < Array.length arr do
        if arr.(!i) = elem then pos := !i;
        incr i
      done;
      !pos;;

val position : 'a -> 'a array -> int = <fun>
```

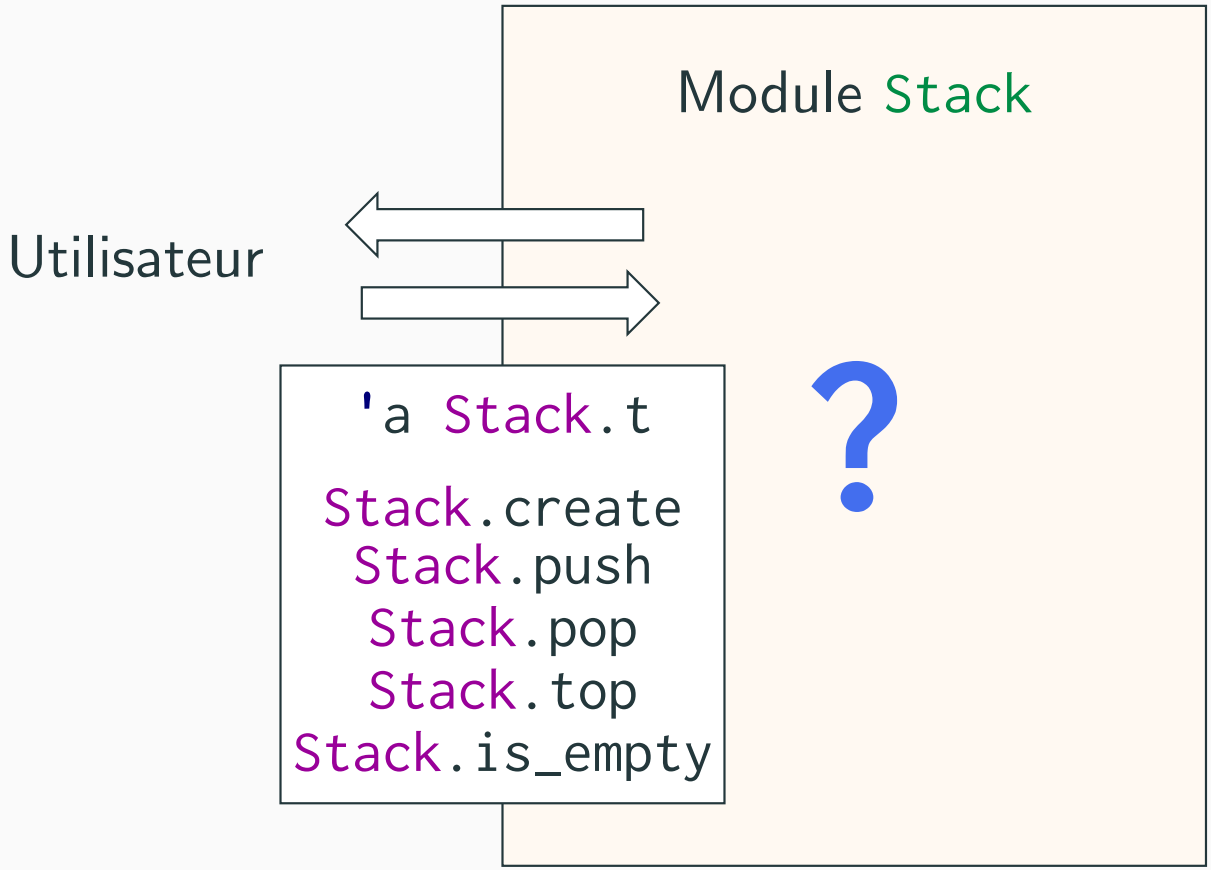
Les interfaces de programmation



Les interfaces de programmation



Les interfaces de programmation



Implémentation

Une liste OCaml **n'est (généralement) pas** une pile !

Car c'est un objet *immutable* :

```
# let x = [37; 42; 54];;
val x : int list = [37; 42; 54]
```


On ne peut changer ce que désigne x sans le redéfinir

Implémentation

```
# type 'a stack_t = { mutable content : 'a list };;

# let create () = { content = [] };;

# let push elem s = s.content <- elem :: s.content;;

# exception Empty;;

# let pop s =
    match s.content with
      | h::t -> s.content <- t; h
      | []   -> raise Empty;;
```

Implémentation

```
# type 'a stack_t = { content : 'a list ref };;

# let create () = { content = ref [] };;

# let push elem s = s.content := elem :: !(s.contenu);;

# exception Empty;;

# let pop s =
    match !(s.content) with
      | h::t -> s.content := t; h
      | []   -> raise Empty;;
```

Implémentation

Utiliser une liste n'est pas la seule solution...

Implémentation

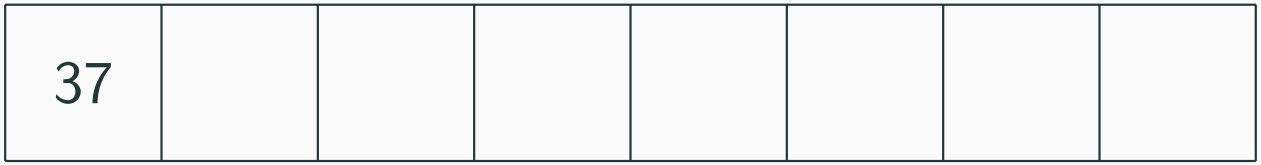
Utiliser une liste n'est pas la seule solution...

On peut utiliser un tableau !

Implémentation

Utiliser une liste n'est pas la seule solution...

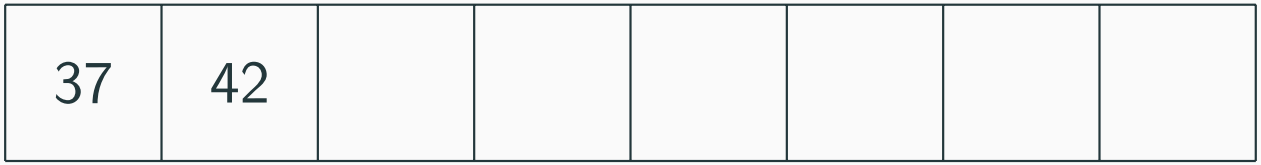
On peut utiliser un tableau !



Implémentation

Utiliser une liste n'est pas la seule solution...

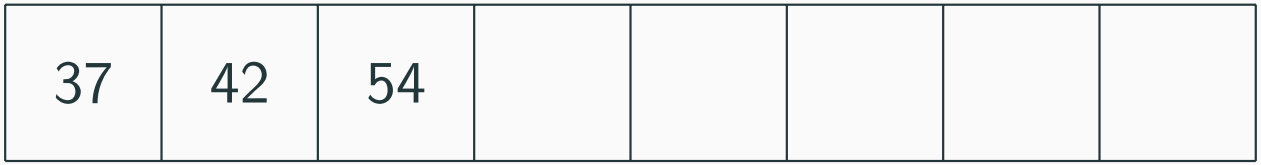
On peut utiliser un tableau !



Implémentation

Utiliser une liste n'est pas la seule solution...

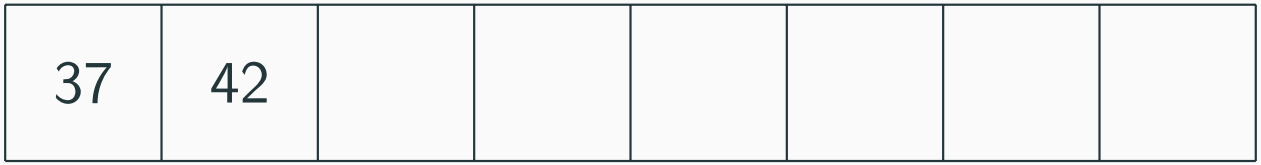
On peut utiliser un tableau !



Implémentation

Utiliser une liste n'est pas la seule solution...

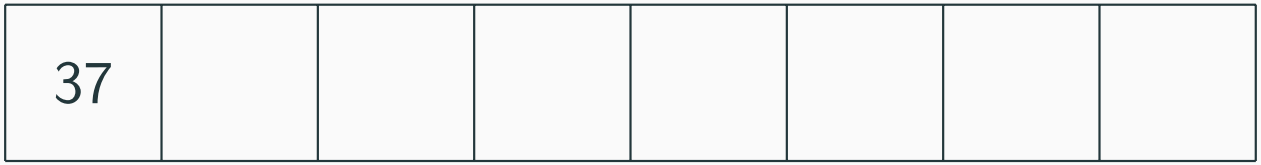
On peut utiliser un tableau !



Implémentation

Utiliser une liste n'est pas la seule solution...

On peut utiliser un tableau !



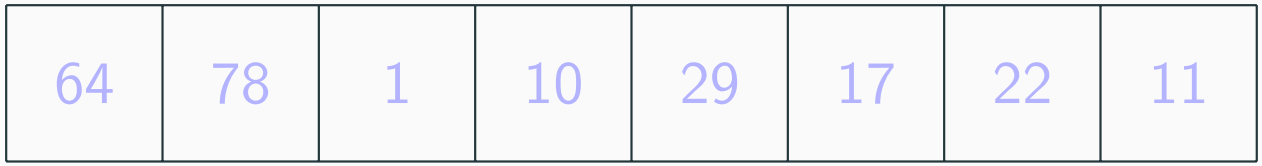
Implémentation

Utiliser une liste n'est pas la seule solution...

On peut utiliser un tableau !

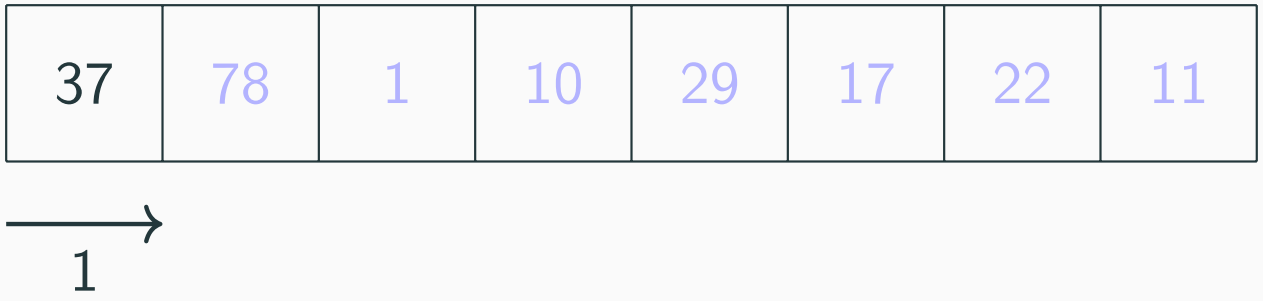
Implémentation

Problème : les cases ne peuvent être vides !



Implémentation

Problème : les cases ne peuvent être vides !



Il nous faut aussi mémoriser le nombre d'éléments

Implémentation

Problème : les cases ne peuvent être vides !

Il nous faut aussi mémoriser le nombre d'éléments

Implémentation

Problème : les cases ne peuvent être vides !

Il nous faut aussi mémoriser le nombre d'éléments

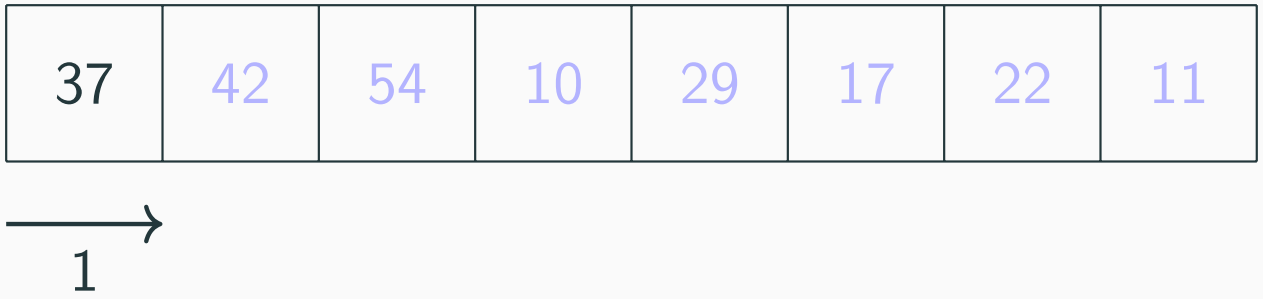
Implémentation

Problème : les cases ne peuvent être vides !

Il nous faut aussi mémoriser le nombre d'éléments

Implémentation

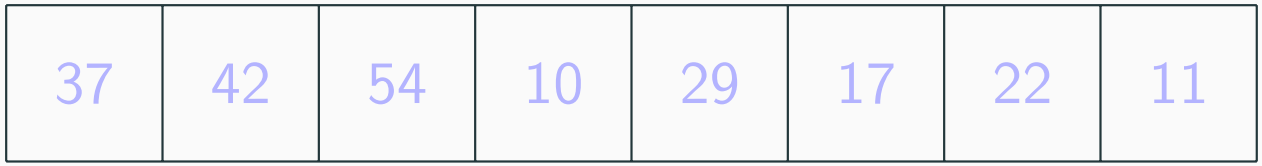
Problème : les cases ne peuvent être vides !



Il nous faut aussi mémoriser le nombre d'éléments

Implémentation

Problème : les cases ne peuvent être vides !



0

Il nous faut aussi mémoriser le nombre d'éléments

Implémentation

On définit donc un type :

```
# type 'a stack_t = { mutable size : int;
                      content : 'a array };;
```

Implémentation

On définit donc un type :

```
# type 'a stack_t = { mutable size : int;
                      content : 'a array };;
```


Pour créer une pile vide :

Implémentation

On définit donc un type :

```
# type 'a stack_t = { mutable size : int;
                      content : 'a array };;
```

Pour créer une pile vide :

```
# let create () = { size = 0; content = [| ?? |] };;
```

Implémentation

Il y a un autre problème : que faire si le tableau déborde ?

Nous allons définir contenu comme *mutable* :

```
# type 'a stack_t = { mutable size : int;
                      mutable content : 'a array };;
```

Cela donne une solution pour créer une pile vide :

```
# let create () = { size = 0; content = [| |] };;

val create : unit -> 'a stack_t = <fun>
```

Implémentation

La fonction pop s'écrit aisément :

```
# exception Empty;;

# let pop s =
    if s.size = 0 then raise Empty;
      s.size <- pile.size - 1;
      s.content.(pile.size);;

val pop : 'a stack_t -> 'a = <fun>
```

Implémentation


```
# let push elem s =
    if s.content = [| |] then
      begin (* La première insertion crée le tableau *)
        s.content <- Array.make 4 elem;
        s.size <- 1
      end
    else let n = s.size in
      if n < Array.length s.content then
        (* S'il reste de la place, on enregistre l'élément *)
        s.content.(n) <- elem
      else begin
          (* S'il n'y a plus de place, on crée un nouveau tableau
             et on y recopie le contenu de l'ancien *)
          let new_content = Array.make (2*n) elem in
            for i = 0 to n-1 do
              new_content.(i) <- s.content.(i)
            done;                          (* stack.content.(n)    *)
            s.content <- new_content;      (* contient déjà elem ! *)
      end;
      s.size <- n+1;;
```

Inconvénients de l'implémentation avec un tableau

Cette approche semble moins bonne :

Inconvénients de l'implémentation avec un tableau

Cette approche semble moins bonne :

- on gaspille de la place en mémoire

Inconvénients de l'implémentation avec un tableau

Cette approche semble moins bonne :

- on gaspille de la place en mémoire
- les copies de tableau prennent du temps

Modération des inconvénients

Toutefois, dans une liste chaînée :

Modération des inconvénients

Toutefois, dans une liste chaînée :

- on gaspille de la place pour mémoriser le « suivant »

Modération des inconvénients

Toutefois, dans une liste chaînée :

- on gaspille de la place pour mémoriser le « suivant »
- on crée un nouvel objet à chaque ajout

Modération des inconvénients

Toutefois, dans une liste chaînée :

- on gaspille de la place pour mémoriser le « suivant »
- on crée un nouvel objet à chaque ajout
- les éléments sont éparpillés en mémoire

Modération des inconvénients

Pour ce qui est des copies :

Modération des inconvénients

Pour ce qui est des copies :

- après 4 ajouts, on copie 4 éléments

Modération des inconvénients

Pour ce qui est des copies :

- après 4 ajouts, on copie 4 éléments
- après 4 ajouts supplémentaires, on copie 8 éléments

Modération des inconvénients

Pour ce qui est des copies :

- après 4 ajouts, on copie 4 éléments
- après 4 ajouts supplémentaires, on copie 8 éléments
- après 8 ajouts supplémentaires, on copie 16 éléments
- etc.

Modération des inconvénients

Pour ce qui est des copies :

- après 4 ajouts, on copie 4 éléments
- après 4 ajouts supplémentaires, on copie 8 éléments
- après 8 ajouts supplémentaires, on copie 16 éléments
- etc.

Le nombre *total* de copies est de l'ordre du nombre *total* d'ajouts

Si certains des ajouts dans la piles sont très coûteux...

... en *moyenne*, les ajouts sont en $O(1)$!

Modules OCaml

Un fichier `Stack.mli` qui contient :

```
type 'a t
exception Empty
val create : unit -> 'a t
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
```


Un fichier `Stack.ml` qui contient les définitions

Et en C ?


```c
struct cell { int value; struct cell* next; };
struct stack { struct cell* top; };

typedef struct pile pile;

stack create(void);
bool is_empty(stack s);
void push(stack* ptr_s, int n);
int pop(stack* ptr_s);
int top(stack s);
```

Et en C ?

```c
stack create(void) {
    stack s = { .top = NULL; };
    return s;
}
```

```c
stack s = create();
```

Et en C ?

```c
bool is_empty(stack s) {
    return s.top == NULL;
}
```

Et en C ?


```c
void push(stack* ptr_s, int n) {
    struct cell* ptr_cell =
        (struct cell*)malloc(sizeof(struct cell));
    if (ptr_cell == NULL) {
        return;
    }
    (*ptr_cell).value = n;
    (*ptr_cell).next = (*ptr_s).top;
    (*ptr_s).top = ptr_cell;
}
```

Et en C ?

On peut utiliser `ptr->field` plutôt que `(*ptr).field` :


```c
void push(stack* ptr_s, int n) {
    struct cell* ptr_cell =
        (struct cell*)malloc(sizeof(struct cell));
    ptr_cell->value = n;
    ptr_cell->next = ptr_s->top;
    ptr_s->top = ptr_cell;
}
```

Et en C ?

```c
int top(stack s) {
    return (s.top)->value;
}
```

Attention, `top(NULL)` est UB !

Et en C ?


```c
int pop(stack* ptr_s) {
    struct cell* ptr_cell = ptr_s->top;
    ptr_s->top = ptr_cell->next;  // On retire la cellule
    int n = ptr_cell->value;      // On mémorise la valeur
    free(ptr_cell);               // On libère la cellule
    return n;
}
```

Et en C ?

```c
void clear(stack* ptr_s) {
    while (!is_empty(*ptr_s)) {
        pop(ptr_s);   // On ignore ici
    }                 // la valeur retournée
}
```

Et en C ?

Avec un tableau :


```c
struct stack {
    int* arr;
    int arr_size;
    int stack_size;
};

typedef struct stack stack;

stack create(void);
bool is_empty(stack s);
void push(stack* ptr_s, int n);
int pop(stack* ptr_s);
int top(stack s);
```

Et en C ?


```c
stack create(void) {
    stack s = { .arr = NULL, .arr_size = 0,
                .stack_size = 0 };
    return s;
}
```



```c
bool is_empty(stack s) {
    return s.stack_size == 0;
}
```

Et en C ?

```c
int pop(stack* ptr_s) {
    ptr_s->stack_size = ptr_s->stack_size - 1;
    return ptr_s->arr[ptr_s->stack_size];
}
```

```c
int top(stack s) {
    return s.arr[s.stack_size-1];
}
```

Et en C ?


```c
void push(stack* ptr_s, int n) {
    if (ptr_s->stack_size == ptr_s->arr_size) {
        // On crée un tableau plus grand (de taille au moins 4)
        int new_size = 2 * ptr_s->arr_size;
        if (new_size < 4) { new_size = 4; }
        // On recopie le contenu du tableau actuel
        int* new_arr = (int*)malloc(new_size * sizeof(int));
        for(int i=0; i<ptr_s->arr_size; ++i) {
            new_arr[i] = ptr_s->arr[i]
        }
        // On remplit le reste avec n
        for(int i=ptr_s->arr_size; i<ptr_s->new_size; ++i) {
            new_arr[i] = n;
        }
        // On met à jour les champs et on libère l'ancien tableau
        free(ptr_s->arr);
        ptr_s->arr = new_arr;
        ptr_s->arr_size = new_size;
        ptr_s->stack_size = ptr_s->stack_size + 1;
    }
}
```

Les files

Les conteneurs « FIFO » sont appelés *files*

(First In is First Out, le premier entré est le premier à sortir)

Les files

Les conteneurs « FIFO » sont appelés *files*

(First In is First Out, le premier entré est le premier à sortir)

Les files

Là encore, des fonctions élémentaires :

Les files

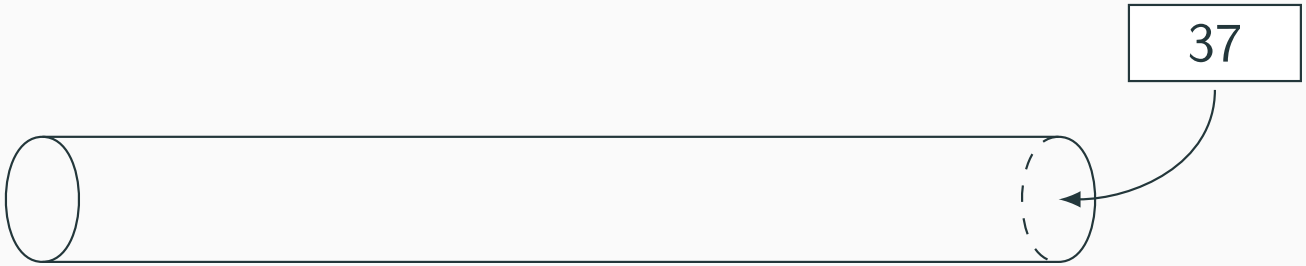
Là encore, des fonctions élémentaires :

- créer une file vide

Les files

Là encore, des fonctions élémentaires :

- créer une file vide
- ajouter des éléments (*add*/*push*)



Les files

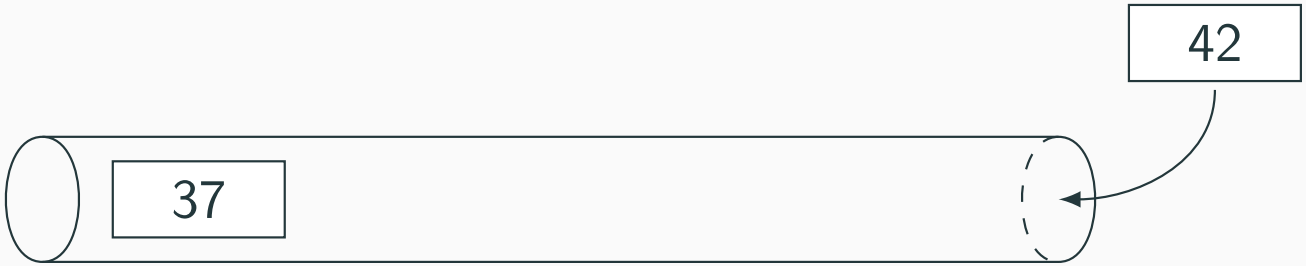
Là encore, des fonctions élémentaires :

- créer une file vide
- ajouter des éléments (*add*/*push*)

Les files

Là encore, des fonctions élémentaires :

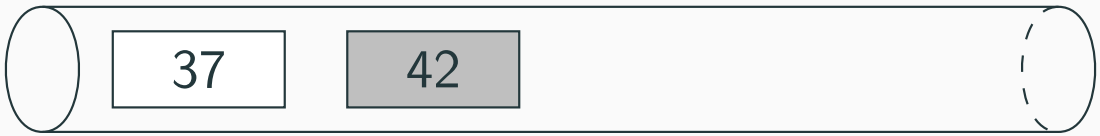
- créer une file vide
- ajouter des éléments (*add*/*push*)



Les files

Là encore, des fonctions élémentaires :

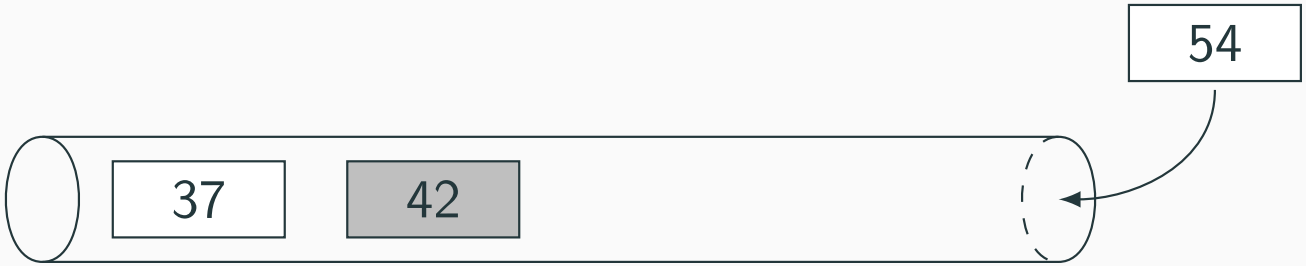
- créer une file vide
- ajouter des éléments (*add*/*push*)



Les files

Là encore, des fonctions élémentaires :

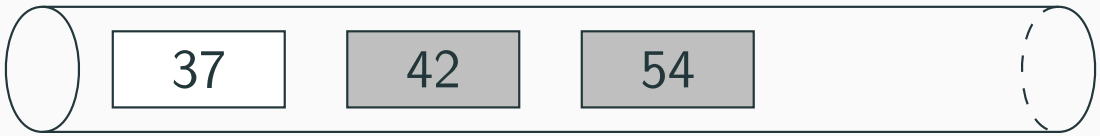
- créer une file vide
- ajouter des éléments (*add*/*push*)



Les files

Là encore, des fonctions élémentaires :

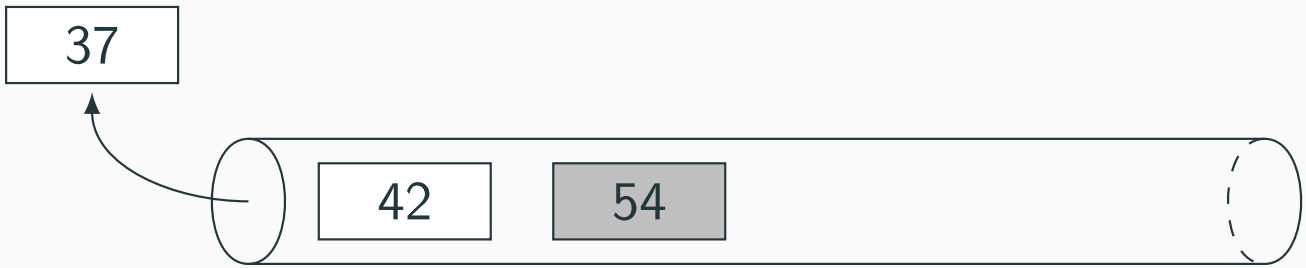
- créer une file vide
- ajouter des éléments (*add*/*push*)



Les files

Là encore, des fonctions élémentaires :

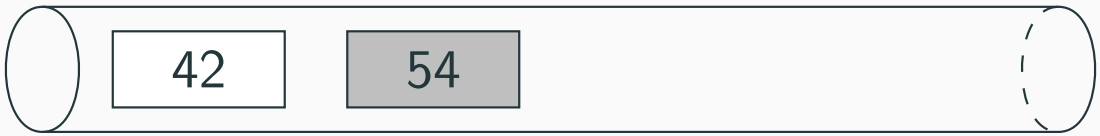
- créer une file vide
- ajouter des éléments (*add*/*push*)
- retirer des éléments (*take*/*pop*)



Les files

Là encore, des fonctions élémentaires :

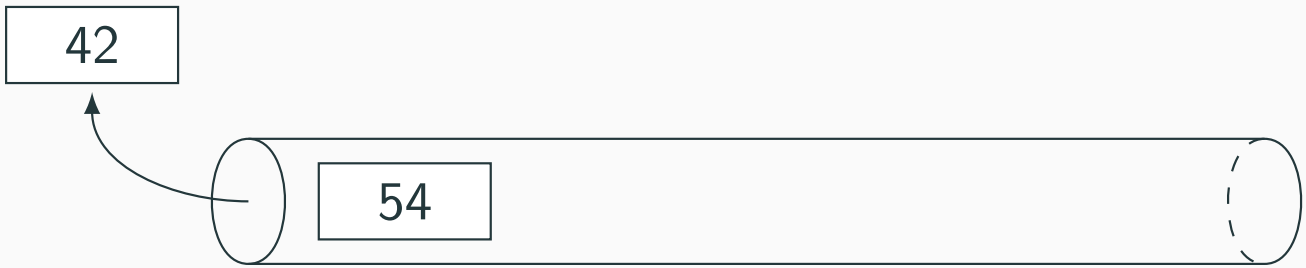
- créer une file vide
- ajouter des éléments (*add*/*push*)
- retirer des éléments (*take*/*pop*)



Les files

Là encore, des fonctions élémentaires :

- créer une file vide
- ajouter des éléments (*add*/*push*)
- retirer des éléments (*take*/*pop*)



Les files

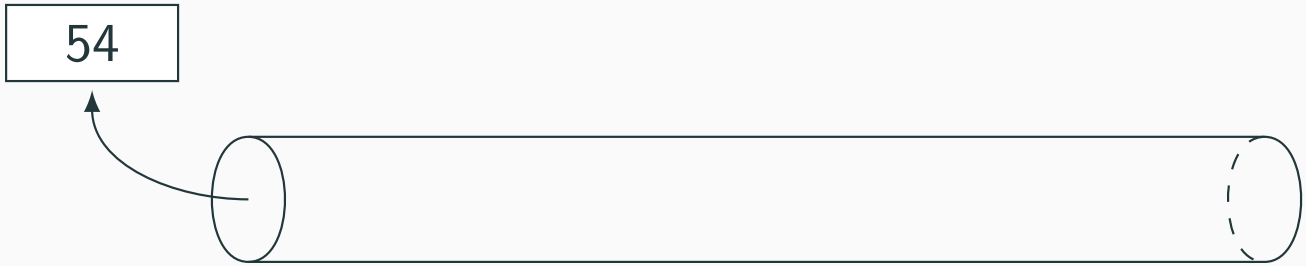
Là encore, des fonctions élémentaires :

- créer une file vide
- ajouter des éléments (*add*/*push*)
- retirer des éléments (*take*/*pop*)

Les files

Là encore, des fonctions élémentaires :

- créer une file vide
- ajouter des éléments (*add*/*push*)
- retirer des éléments (*take*/*pop*)



Le module Queue

Les fonctions utiles sont dans un module appelé `Queue`

- `Queue.create` `unit -> 'a Queue.t`
- `Queue.push` `'a -> 'a Queue.t -> unit`
- `Queue.pop` `'a Queue.t -> 'a`
- `Queue.top` `'a Queue.t -> 'a`
- `Queue.is_empy` `'a Queue.t -> bool`

Le module Queue

La fonction top doit être implémentée explicitement !

Pour `is_empty`, on pourrait utiliser top :

```
# let is_empty q =
    try
      let elem = Queue.top q in false
    with
      | Empty -> true;;

val is_empty : 'a Queue.t -> bool = <fun>
```

Implémentation des files

L'implémentation est plus délicate que pour une pile

On ne peut pas modifier une liste à droite !

Pour le permettre, il faut revenir aux listes chaînées...

Implémentation par une liste chaînée

On part d'une liste chaînée :

Implémentation par une liste chaînée

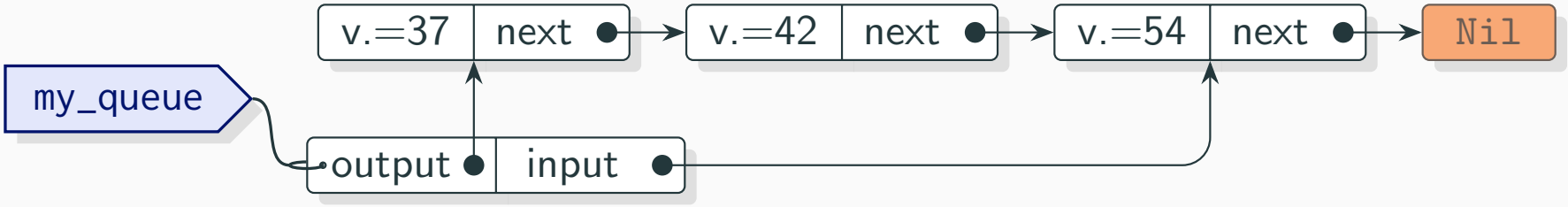
On part d'une liste chaînée :

next doit pouvoir être modifié !

```
# type 'a clst =
  | Nil
  | Cell { value : 'a ; (* les éléments *)
           mutable next : 'a clst }
```

Implémentation par une liste chaînée

Par ailleurs, il faut garder un lien direct vers le dernier élément

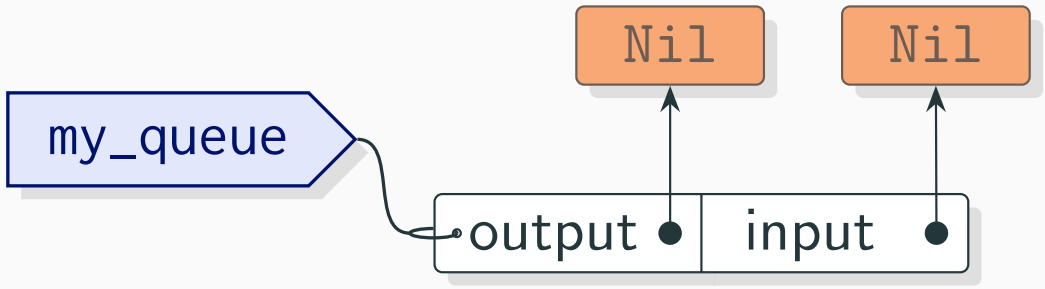


Une file possède un lien vers les deux extrémités :

```
type 'a queue_t = { mutable input  : 'a clst ;
                    mutable output : 'a clst }
```

Implémentation par une liste chaînée

Pour créer une file vide :

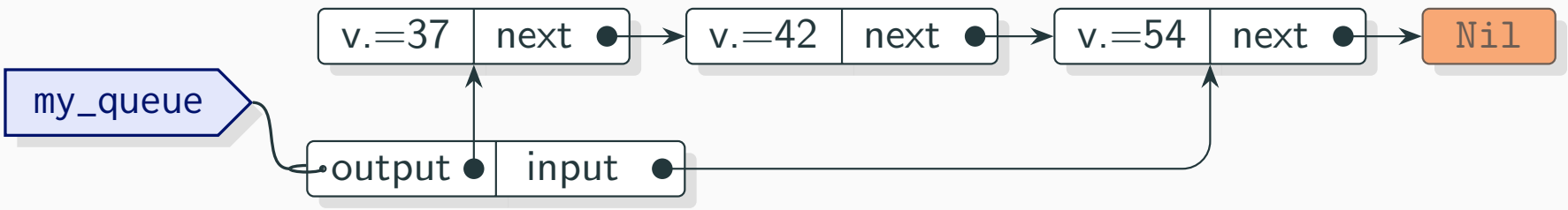



```
# let create () = { input = Nil ; output = Nil };;

val create : unit -> 'a queue = <fun>
```

Implémentation par une liste chaînée

La fonction top ne pose pas de difficulté :




```
# exception Empty;;

# let top q = match q.output with
    | Nil    -> raise Empty
    | Cell c -> c.value;;

val top : 'a queue_t -> 'a = <fun>
```

Implémentation par une liste chaînée

Pour ajouter un élément, il faut distinguer le cas d'une file vide :


```
# let push elem q =
    let c = Cell { v = elem; next = Nil }
      in match q.input with
        | Nil    -> q.input <- c;
                    q.output <- c
        | Cell ci -> q.input <- c;
                     ci.next <- c;;

val push : 'a -> 'a queue_t -> unit = <fun>
```

Implémentation par une liste chaînée

Pour retirer un élément, là aussi, plusieurs cas :


```
# let pop q = match q.output with
    | Nil    -> raise Empty
    | Cell c -> q.output <- c.suiv;
                if c.next = Nil
                  then q.input <- Nil;
                c.v;;

val take : 'a queue_t -> 'a = <fun>
```

Implémentation par un tableau

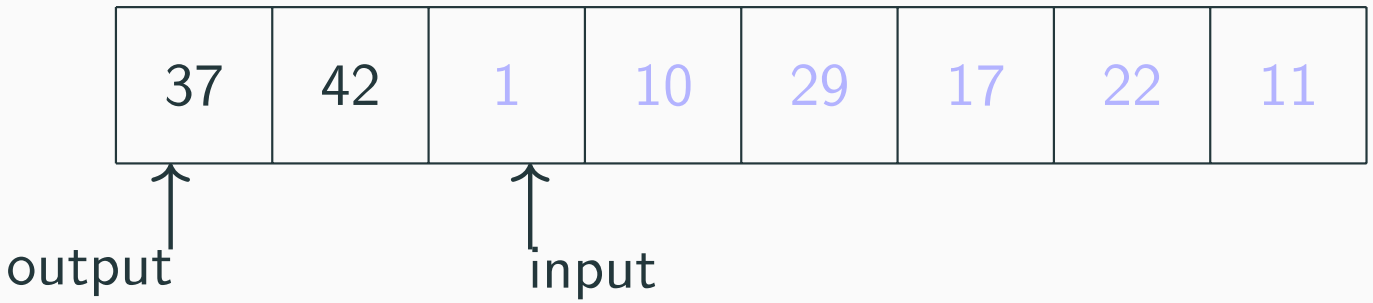
On mémorise des index `entree` et `sortie`

Implémentation par un tableau

On mémorise des index `entree` et `sortie`

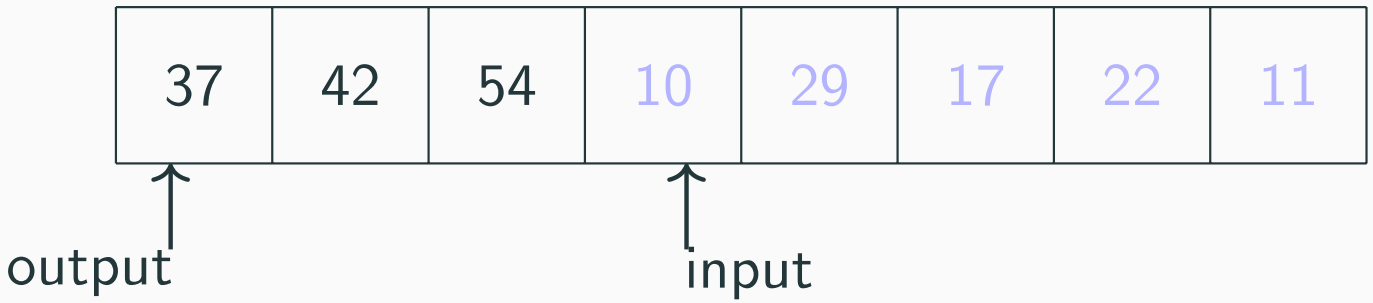
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



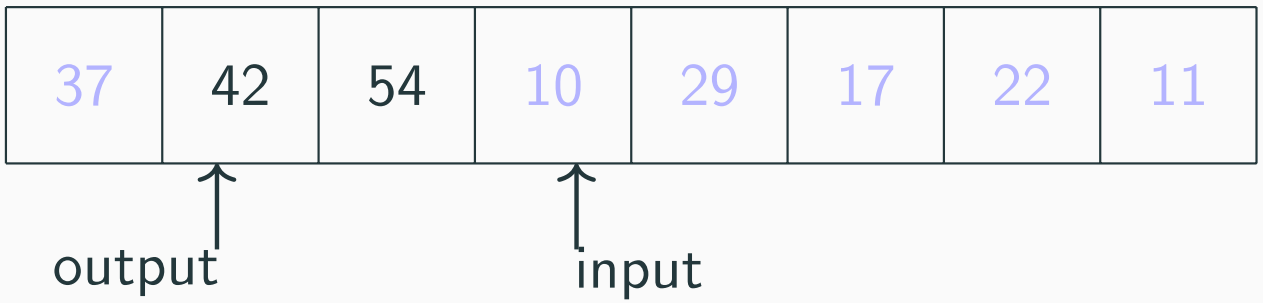
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



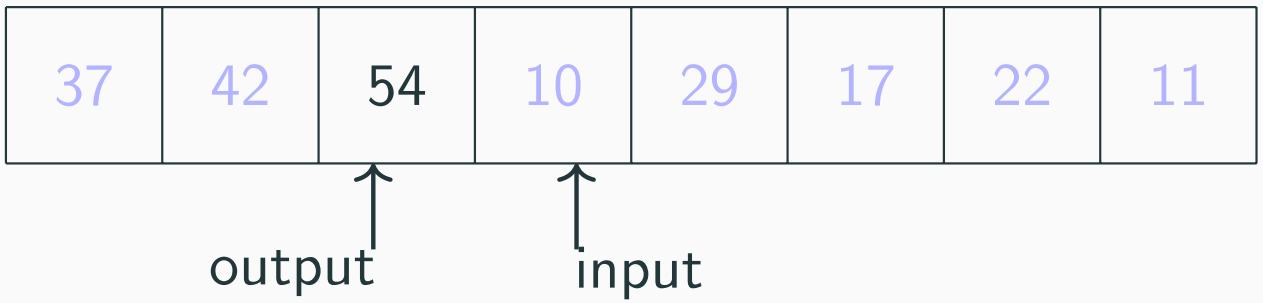
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



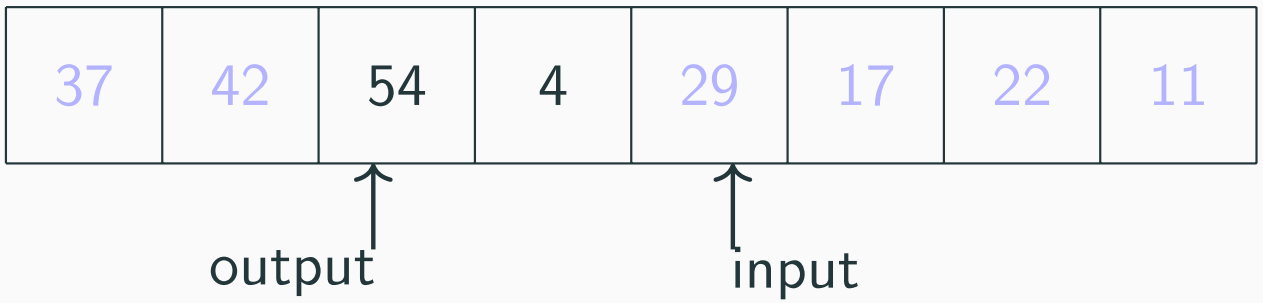
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



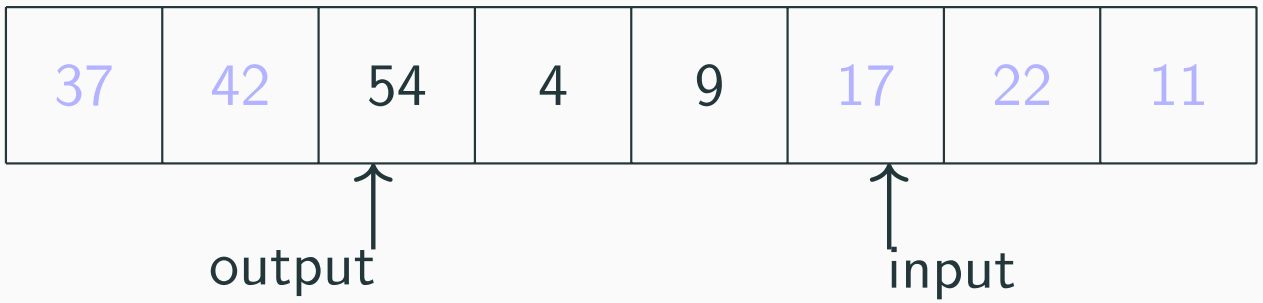
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



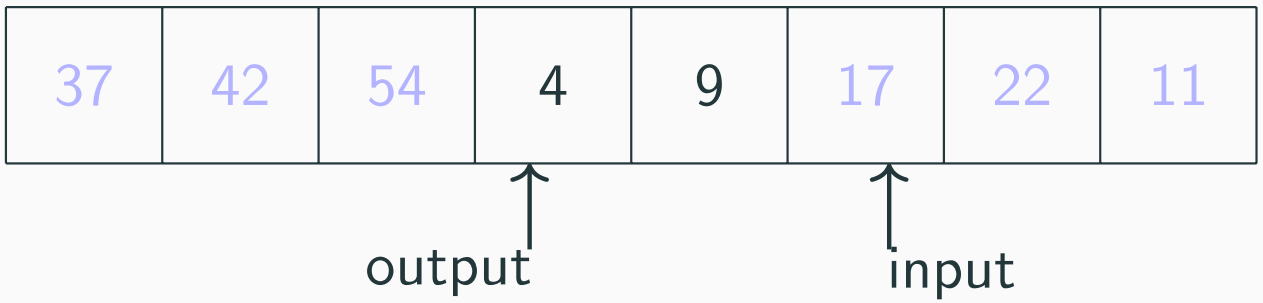
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



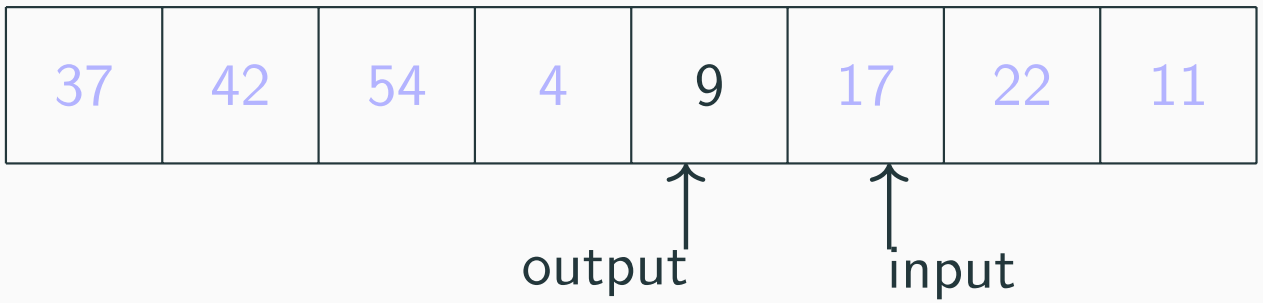
Implémentation par un tableau

On mémorise des index `entree` et `sortie`



Implémentation par un tableau

On mémorise des index `entree` et `sortie`



Implémentation par un tableau

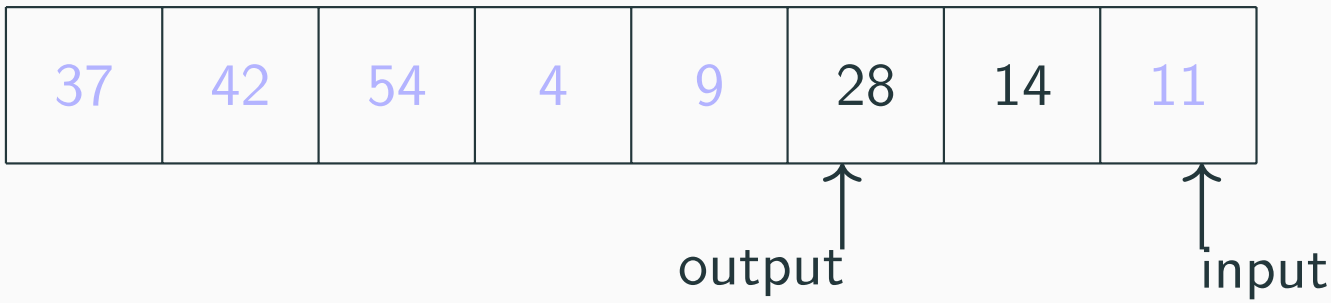
On mémorise des index `entree` et `sortie`

Implémentation par un tableau

On mémorise des index `entree` et `sortie`

Implémentation par un tableau

On mémorise des index `entree` et `sortie`



Implémentation par un tableau

On mémorise des index `entree` et `sortie`

Implémentation par un tableau

On mémorise des index `entree` et `sortie`

Implémentation par un tableau

Pour ce qui est du type :

```
# type 'a queue_t = { mutable content : 'a array ;
                      mutable input : int ;
                      mutable output : int };;
```


Pour créer une file vide :

```
# let create () = { content = [| |];
                    input = 0;
                    output = 0 };;

val create : unit -> 'a queue_t = <fun>
```

Implémentation par un tableau

On peut aisément tester si la file est vide :

```
# let is_empty q =
    (q.input = q.output);;

val is_empty : 'a queue_t -> bool = <fun>
```

Pour que cela fonctionne, on va garantir que

- soit le tableau est de longueur nulle
- soit il a au moins une case non utilisée

Implémentation par un tableau

Pour obtenir (sans l'extraire) le prochain élément à sortir :

```
# let top q =
    if is_empty q then raise Empty
    q.content.(q.output);;

val top : 'a queue_t -> 'a = <fun>
```

Implémentation par un tableau

Et extraire le prochain élément à sortir :


```
# let pop q =
    if is_empty q then raise Empty;
    let elem = q.content.(q.output) in
      q.output <- (q.output + 1)
                          mod Array.length q.content;
      elem;;

val pop : 'a queue_t -> 'a = <fun>
```

Implémentation par un tableau

La difficulté, encore, est l'ajout

Il faut maintenir la condition précédemment énoncée

- si le tableau est de longueur nulle, il en faut un autre
- si c'est la dernière case libre, il en faut un plus grand

Implémentation par un tableau

Tout d'abord, si le tableau est de taille nulle :


```
# let push elem q =
    let size = Array.length q.content in
      if size = 0 then      (* Insertion du premier élément,       *)
        begin               (* on crée un premier tableau non vide *)
          q.content <- Array.make 4 elem;
          q.input <- 1
        end
      else
        ...
```

Implémentation par un tableau

Dans le cas contraire :


```
...
else
  begin
    q.content.(q.input) <- elem;
    q.input <- (q.input + 1) mod size;
    if q.outout = q.input then          (* Si le tableau est    *)
      begin                             (* plein, on crée un    *)
        let new_content = Array.make    (* nouveau tableau deux *)
                    (size*2) elem in    (* fois plus grand dans *)
          for i = 0 to size-1 do        (* lequel on va         *)
            new_content.(i) <-          (* recopier les données *)
              q.content.((q.output + i) mod size)
          done;
        q.content <- new_content;
        q.input <- size;                (* les "size" premieres *)
        q.output <- 0                   (* cases sont occupées  *)
      end
  end;;
```

Implémentation par un tableau

Les avantages et inconvénients sont les mêmes que pour une pile implémentée par un tableau

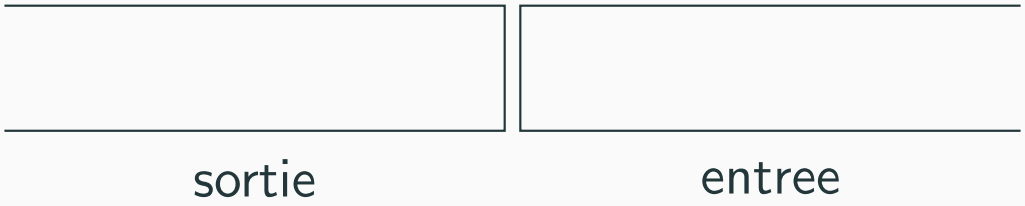
En particulier, l'ajout est $O(1)$ *en moyenne*

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

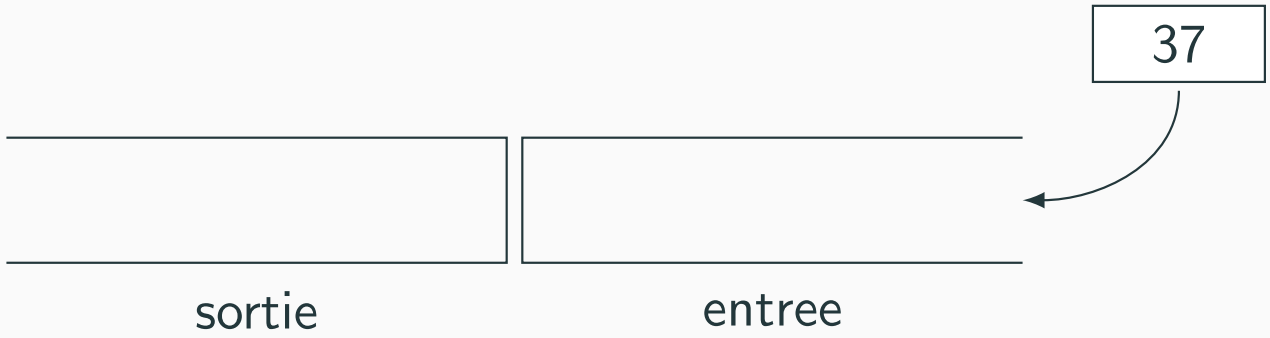
Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !



Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

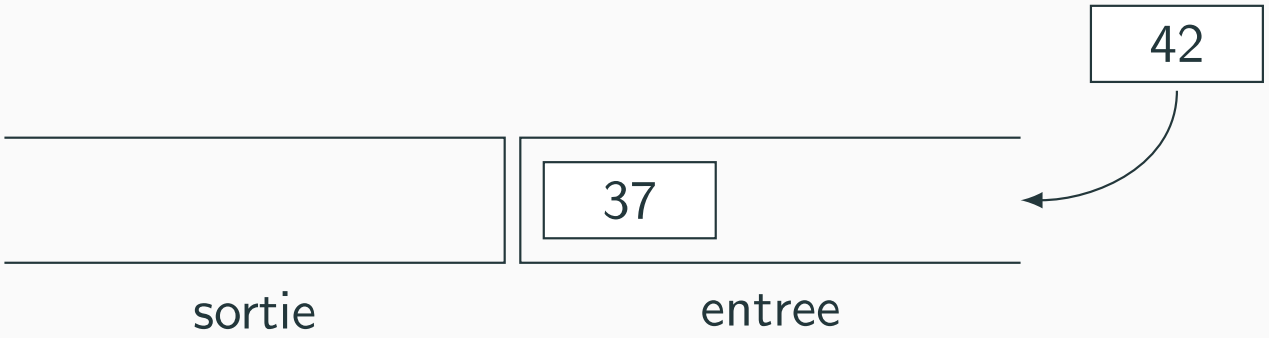


Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !



Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

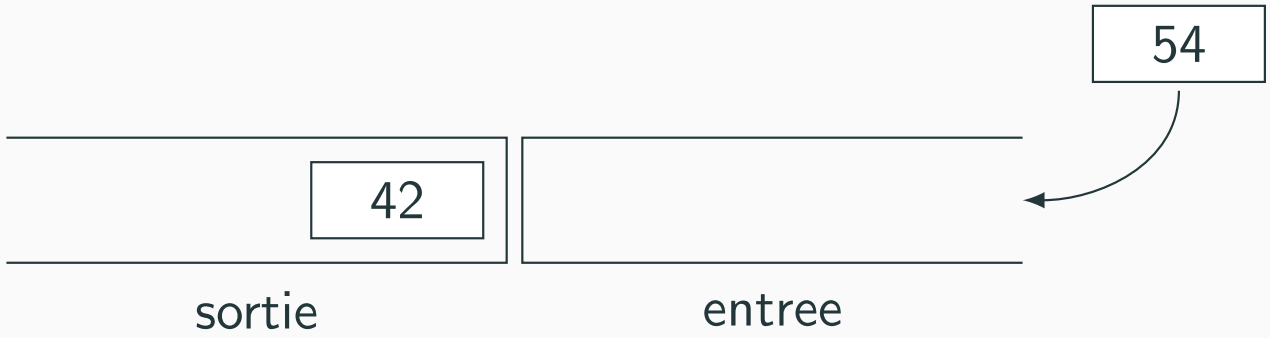
Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !



Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

On peut aussi simuler une file avec deux piles !

Lorsque l'on a besoin de la tête de la file et que la pile de gauche est vide, on transfère tous les éléments !

Implémentation avec deux piles

La définition serait par exemple :

```
type 'a queue_t = { mutable input  : 'a list;
                    mutable output : 'a list; };;
```


Pour tester si la file est vide, on vérifie les deux piles :

```
# let is_empty q =
    (q.input = [] && q.output = []);;

val is_empty : 'a queue_t -> bool = <fun>
```

Implémentation avec deux piles

Cette fois, c'est l'ajout qui est très facile :

```
# let push elem q =
    q.input <- elem::q.input;;

val add : 'a -> 'a queue_t -> unit = <fun>
```

Implémentation avec deux piles

Intéressons-nous ensuite à top :


```
# exception Empty;;

# let top top =
    if q.output = [] then
      while q.input <> [] do
        q.output <- List.hd q.input :: q.output;
        q.input <- List.tl q.input
      done;
      if q.output = [] then raise Empty;
      List.hd q.output;;

val top : 'a queue_t -> 'a = <fun>
```

Implémentation avec deux piles

Enfin, pour extraire un élément de la file :

```
# let pop q =
    let elem = top q in
      q.output <- List.tl q.output;
      elem;;

val pop : 'a queue_t -> 'a = <fun>
```

Et en C ?


```c
struct queue {
    int* content;
    int arr_size;
    int i_input;
    int i_output;
};

typedef struct queue Queue;

Queue create(void);
bool is_empty(Queue q);
void push(Queue* ptr_q, int elem);
int top(Queue q);
int pop(Queue* ptr_q);
```

Et en C ?


```c
Queue create(void) {
    Queue q = { .content = NULL,
                .arr_size = 0,
                .i_input = 0,
                .i_output = 0 };
    return q;
}
```

Et en C ?

```c
bool is_empty(Queue q) {
    return q.i_input == q.i_output;
}
```

Et en C ?

```c
int pop(Queue* ptr_q) {
    int elem = top(*ptr_q);

    ptr_q->i_output = (ptr_q->i_output + 1)
                                % ptr_q->arr_size;
}
```

Et en C ?


```c
void push(Queue* ptr_q, int elem) {
    if (ptr_q->content == NULL) {
        // Premier ajout, on crée le tableau
        ptr_q->content = (int*)malloc(4*sizeof(int));
        if ((ptr_q->content) != NULL) {
            ptr_q->size = 4;
            for (int i=0; i<4; ++i) {
                ptr_q->content[i] = elem;
            }
        }
    } else {
        // Ajout de l'élément
        ptr_q->content[ptr_q->i_input] = elem;
        ptr_q->i_input = (ptr_q->i_input+1) % ptr_q->arr_size;
        ...
```

Et en C ?


```
        if (ptr_q->i_input == ptr_q->i_output) {
            // Le tableau est plein, on en crée un neuf
            int new_arr_size = 2 * ptr_q->arr_size;
            int* new_content = (int*)malloc(new_arr_size*sizeof(int));
            if (new_content != NULL) {
                for(int i=0; i<ptr_q->arr_size; ++i) {
                    int j = (ptr_q->i_input + i) % ptr_q->arr_size;
                    new_content[i] = ptr_q->content[j];
                }
                for(int i=ptr_q->arr_size; i<new_arr_size; ++i) {
                    new_content[i] = elem;
                }
                free(ptr_q->content);
                ptr_q->i_output = 0;
                ptr_q->i_input = ptr_q->arr_size;
                ptr_q->arr_size = new_arr_size;
                ptr_q->content = new_content;
            }
        }
    }
}
```

Et en C ?

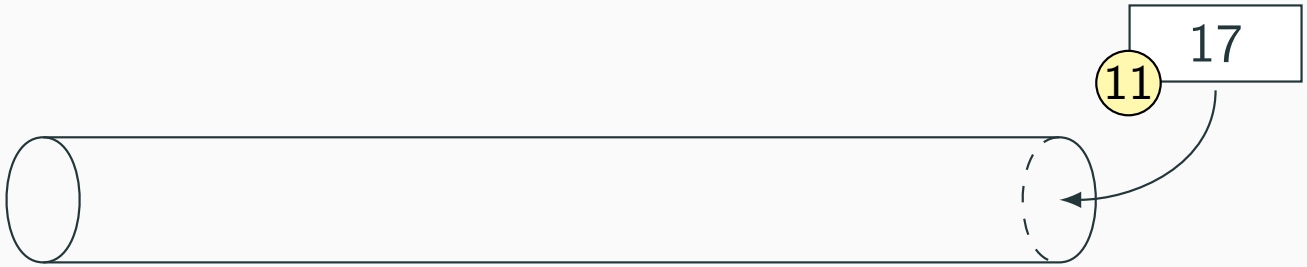
```c
void clear(Queue* ptr_q) {
    free(ptr_q->content);
    ptr_q->content = NULL;
    ptr_q->arr_size = 0;
    ptr_q->i_input = 0;
    ptr_q->i_output = 0;
}
```

Files de priorité

Il s'agit d'une file où les éléments ont des priorités :

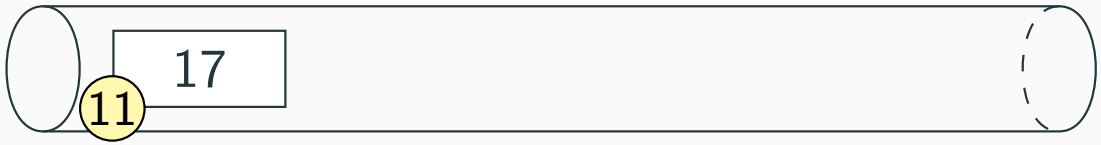
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



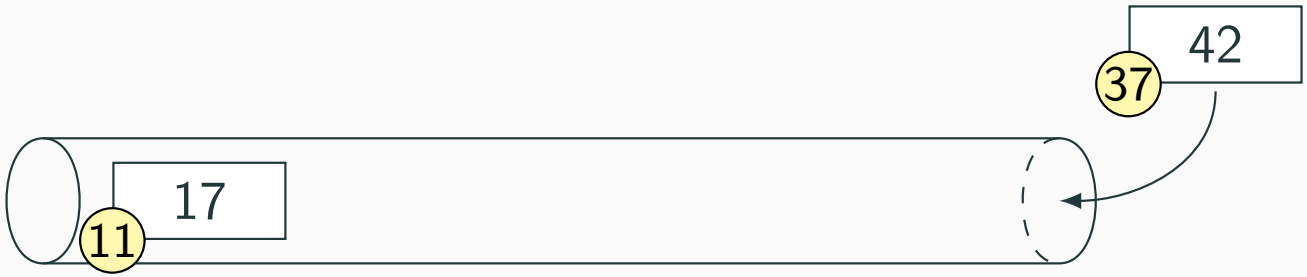
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



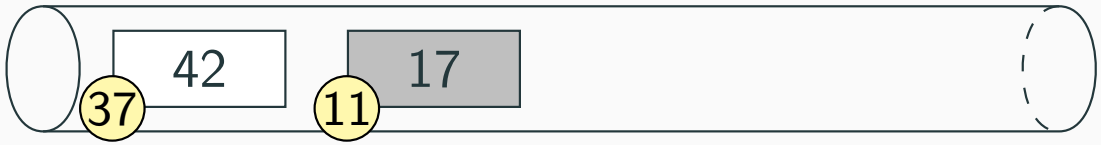
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



Files de priorité

Il s'agit d'une file où les éléments ont des priorités :

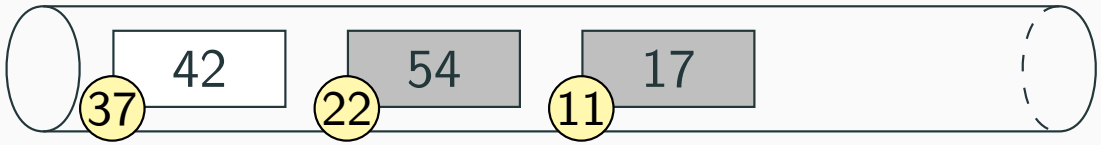


Files de priorité

Il s'agit d'une file où les éléments ont des priorités :

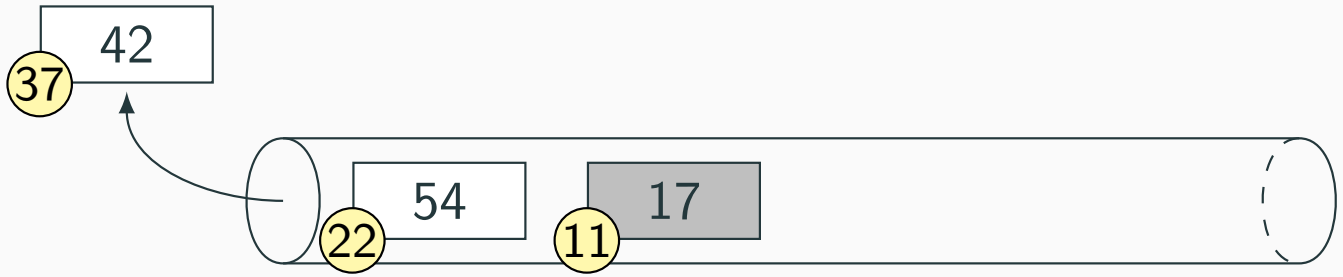
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



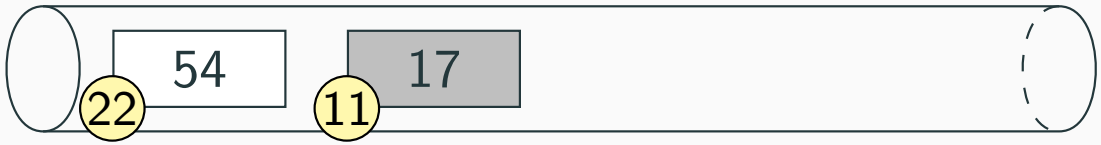
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



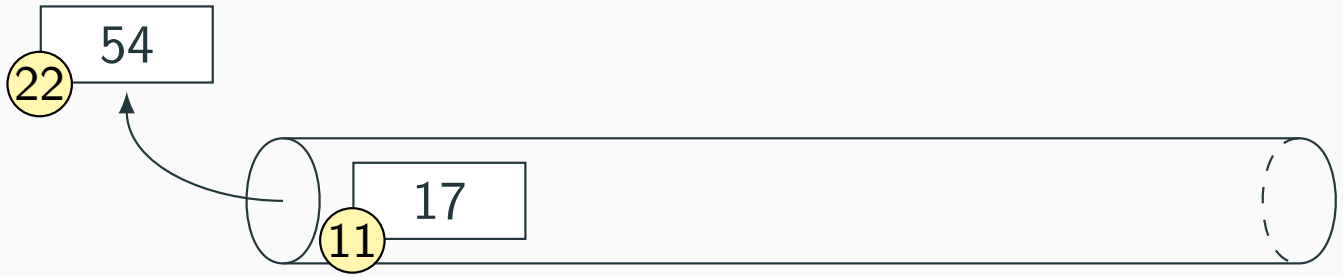
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



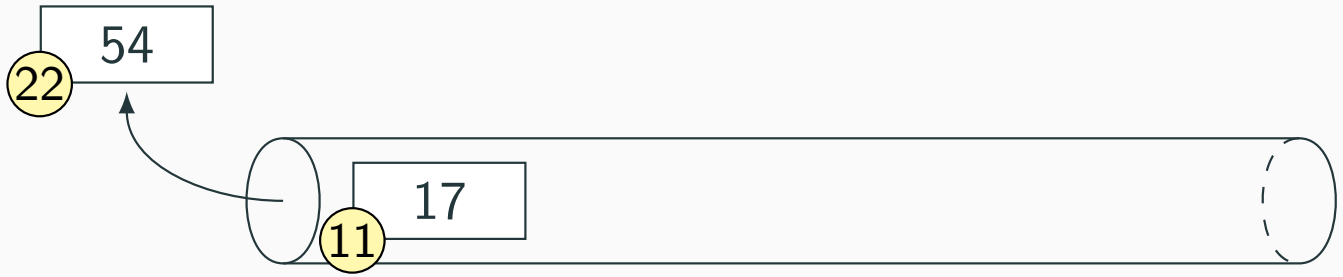
Files de priorité

Il s'agit d'une file où les éléments ont des priorités :



Files de priorité

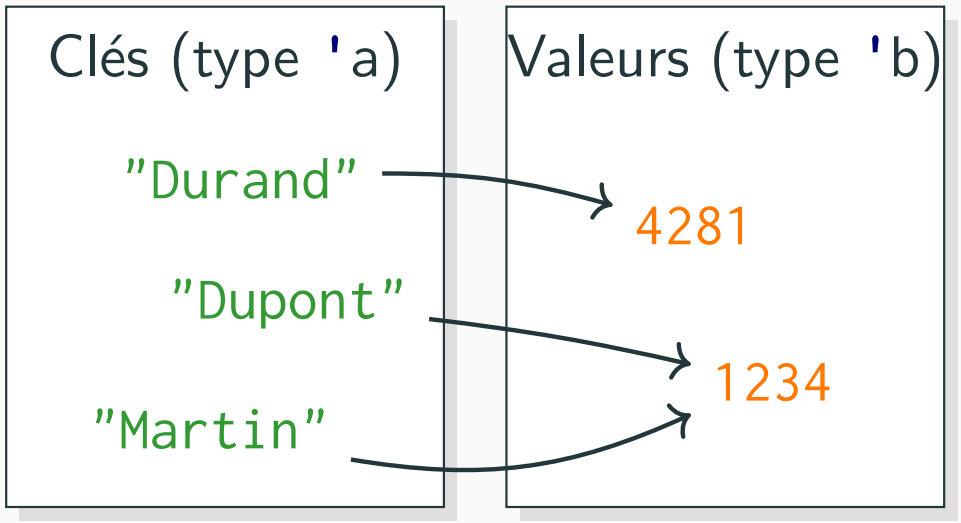
Il s'agit d'une file où les éléments ont des priorités :



Ces files seront implémentées dans le prochain chapitre

Les dictionnaires

Association clé → valeur



Opérations

Au minimum :

- Créer un dictionnaire vide
- Ajouter une association clé → valeur
- Trouver la valeur associée à une clé
- Vérifier la présence d'une clé

Possiblement :

- Modifier la valeur associée à une clé
- Supprimer une association clé-valeur
- Itérer sur les couples (clé, valeur)

Module Hashtbl

- `Hashtbl.create (int -> ('a, 'b) Hashtbl.t)`
- `Hashtbl.add (('a, 'b) Hashtbl.t -> 'a -> 'b -> unit)`
- `Hashtbl.find (('a, 'b) Hashtbl.t -> 'a -> 'b)`
- `Hashtbl.mem (('a, 'b) Hashtbl.t -> 'a -> bool)`
- `Hashtbl.remove (('a, 'b) Hashtbl.t -> 'a -> unit)`
- `Hashtbl.iter`
  `(('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit)`

Module Hashtbl

- `Hashtbl.replace`
  `(('a, 'b) Hashtbl.t -> 'a -> 'b -> unit)`

Agit comme `Hashtbl.add` si la clé n'est pas présente

OCaml gère les associations comme des piles de valeurs associées aux clés

Comment l'implémenter ?

Une liste de couples est inefficace

Un tableau trié de couples aussi

Mais on veut la rapidité d'accès d'un tableau

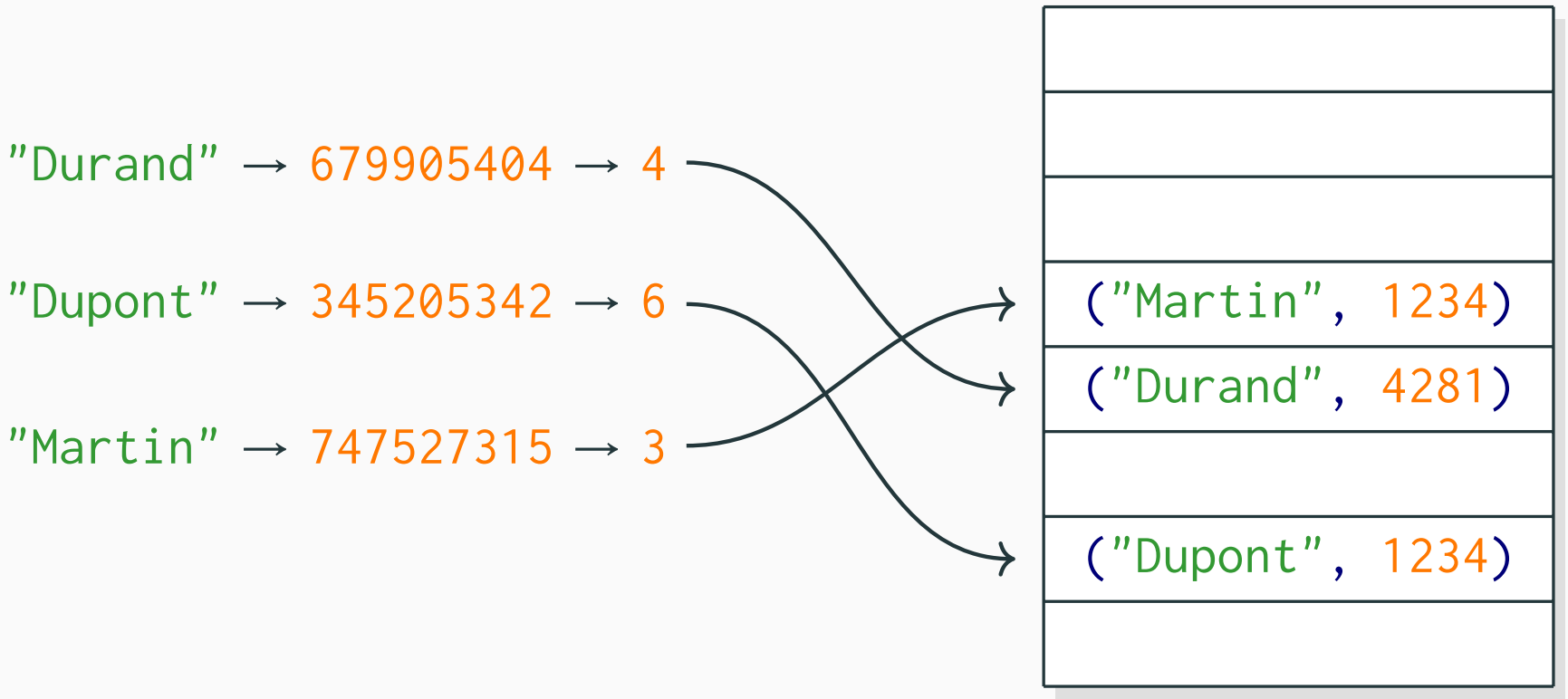
Accès avec une clé et non un indice !

Hashtbl.hash

Hachage : `'a -> int`

`Hashtbl.hash`

Table de hachage



Propriétés de la fonction de hachage

Qu'est-ce qui est souhaitable ?

Propriétés de la fonction de hachage

Qu'est-ce qui est souhaitable ?

- rapide
- distribution aussi uniforme que possible
- imprévisible

Collisions

m cases, n clés, au moins

$$1 - \frac{m!}{m^n (m-n)!}$$

collisions

$m = 365,\ n = 45 \qquad \rightarrow \qquad 95\%$

$m = 1000,\ n = 50 \qquad \rightarrow \qquad 71\%$

Implémentation

```
type ('a, 'b) t = { table : ('a * 'b) list array }
```



```
# let create n =
    { table = Array.make n [] };;

val create : int -> ('a, 'b) t = <fun>
```

Implémentation


```
# let find dict key =
    let hsh = Hashtbl.hash key
              mod (Array.length dict.table) in

    let rec auxFind = function
      | []                      -> raise Not_found
      | (k, v)::_ when k=key -> v
      | _::t                    -> auxFind t
    in auxFind dict.table.(hsh);;

val find : ('a, 'b) t -> 'a -> 'b = <fun>
```

Implémentation


```
# let mem dict key =
    let hsh = Hashtbl.hash key
              mod (Array.length dict.table) in

    let rec auxMem = function
      | []         -> false
      | (k, v)::t -> k=key || auxMem t
    in auxMem dict.table.(hsh);;

val mem : ('a, 'b) t -> 'a -> bool = <fun>
```

Implémentation


```
# let add dict key value =
    let hsh = Hashtbl.hash key
              mod (Array.length dict.table)

    in dict.table.(hsh)
         <- (key, value)::dict.table.(hsh));;

val add : ('a, 'b) t -> 'a -> 'b -> unit = <fun>
```

Implémentation


```
# let replace dict key value =
    let hsh = Hashtbl.hash key
              mod (Array.length dict.table) in

    let rec auxReplace = function
      | []                        -> [(key, value)]
      | (k, v)::t when k=key -> (key, value)::t
      | h::t                      -> h::(auxReplace t)
    in dict.table.(hsh)
         <- (auxReplace dict.table.(hsh));;

val replace : ('a, 'b) t -> 'a -> 'b -> unit = <fun>
```

Implémentation


```
# let remove dict key =
    let hsh = Hashtbl.hash key
              mod (Array.length dict.table) in

    let rec auxRemove = function
      | []                      -> []
      | (k, _)::t when k=key -> t
      | h::t                    -> h::(auxRemove t)
    in dict.table.(hsh)
         <- (auxRemove dict.table.(hsh));;

val remove : ('a, 'b) t -> 'a -> unit = <fun>
```

Implémentation

Pour pouvoir agrandir la table :

```
type ('a, 'b) t =
  { mutable table : ('a * 'b) list array }
```

Implémentation


```
let extend dict =
  let m = Array.length dict.table in     (* Ancienne taille m          *)
    let p = 2*m+1 in                     (* Nouvelle taille p = 2m+1   *)
      let n_table = Array.make p [] in   (* Nouvelle table de hachage *)
        let addkv = function (k, v)
          -> let hsh = Hashtbl.hash k mod p
               in n_table.(hsh) <- (k, v)::n_table.(hsh)
        in for i=0 to n-1 do             (* On la remplit avec tous   *)
          List.iter addkv                (* les couples (clé,valeur)  *)
            (List.rev dict.table.(i))    (* dans le bon ordre         *)
        done;                            (* dans l'ancienne table     *)
      dict.table <- n_table;;    (* Elle remplace la table précédente *)

val extend : ('a, 'b) t -> unit = <fun>
```

Clés et mutabilité

Que se passe-t-il si les clés sont mutables ?

Et en C ?

Quelques difficultés :

- pas de polymorphisme
- pas de gestion des erreurs
- pas de fonction de hachage
- pas de listes chaînées pour les cases
- quid de la propriété des objets pointés ?