Structures séquentielles : piles, files

Quentin Fortier

July 2, 2022

Structure de donnée

Une structure de donnée est un moyen de stocker un ensemble d'éléments.

Structure de donnée

Une structure de donnée est un moyen de stocker un ensemble d'éléments.

Structures de données que nous allons voir:

1. Liste (chaînée, doublement chaînée, circulaire...)
2. Tableau (dynamique)
3. Pile, file
4. Dictionnaire (avec table de hachage, ABR)
5. Arbre binaire de recherche (TD: AVL, rouge/noir...)
6. File de priorité (avec tas ou ABR)
7. Graphe (plus tard)

Structure de donnée : persistant vs mutable

1. Structure **persistante / fonctionnelle** : ne peut pas être modifiée, seules de nouvelles valeurs sont renvoyées.
   Exemples : `list`, `string`, `arbre`...
2. Structure **mutable** : peut être modifiée.
   Exemples : `array`, `ref`, `mutable`...

Structure de donnée : persistant vs mutable

1. Structure **persistante / fonctionnelle** : ne peut pas être modifiée, seules de nouvelles valeurs sont renvoyées.
   Exemples : `list`, `string`, `arbre`...
2. Structure **mutable** : peut être modifiée.
   Exemples : `array`, `ref`, `mutable`...

Avantage des structures persistantes : moins de risques de bugs, programme plus facile à prouver par **induction structurelle**, backtracking (retour en arrière plus aisé).

Structure de donnée : persistant vs mutable

On peut souvent voir, avec le type d'une fonction, si la structure utilisée est persistante.

Exemple : un algorithme de tri sera de type `'a list -> 'a list` pour une liste et `'a array -> unit` pour un tableau.

Structure de donnée : persistant vs mutable

On peut souvent voir, avec le type d'une fonction, si la structure utilisée est persistante.

Exemple : un algorithme de tri sera de type `'a list -> 'a list` pour une liste et `'a array -> unit` pour un tableau.

Les structures persistantes sont plus adaptées à la programmation fonctionnelle (récursive) et les structures mutables à la programmation impérative (boucles, références).

Structure de donnée : type abstrait, réalisation concrète

1. **Type abstrait** : description des opérations permises sur une structure de donnée.
   Exemple: pile, file...

Structure de donnée : type abstrait, réalisation concrète

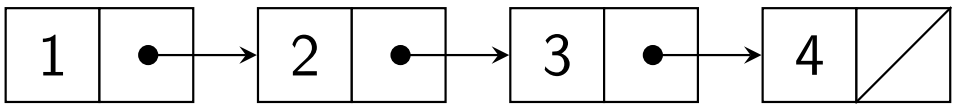
1. **Type abstrait** : description des opérations permises sur une structure de donnée.
   Exemple: pile, file...
2. **Réalisation concrète**: implémentation de ces opérations dans un langage de programmation, qui en détermine la complexité :
   $O(n)$: mauvais
   $O(\log(n))$: bien
   $O(1)$: parfait.

Il peut exister plusieurs réalisations concrètes de la même structure abstraite.

Liste chaînée

Une liste chaînée est composée d'une suite de noeuds.

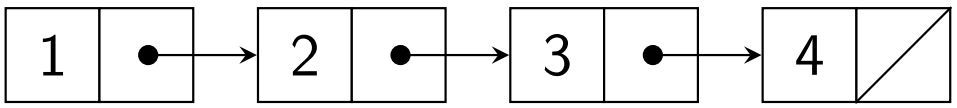
Chaque noeud possède une valeur et un pointeur sur le noeud suivant.



Liste chaînée

Une liste chaînée est composée d'une suite de noeuds.

Chaque noeud possède une valeur et un pointeur sur le noeud suivant.



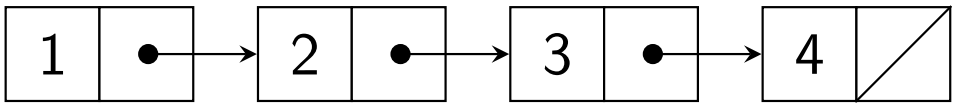
En OCaml :

```
1::2::3::4::[] (* même chose que [1; 2; 3; 4] *)
```

Liste chaînée

Une liste chaînée est composée d'une suite de noeuds.

Chaque noeud possède une valeur et un pointeur sur le noeud suivant.



En OCaml :

```
1::2::3::4::[] (* même chose que [1; 2; 3; 4] *)
```

En C :

```
struct list {
  int elem;
  struct list *next;
};
```

Liste chaînée

```c
struct list {
    int elem;
    struct list *next;
};

list *add(list *l, int v) {
    list *l_new = (list*)malloc(sizeof(list));
    l_new->elem = v;
    l_new->next = l;
    return l_new;
}

list *l = NULL;
for(int i=4; i > 0; i--)
    l = add(l, i);
```

Liste chaînée

Application : la blockchain du Bitcoin stocke toutes les transactions sous forme de liste chaînée.

Liste chaînée

Application : la blockchain du Bitcoin stocke toutes les transactions sous forme de liste chaînée.

Sur https://github.com/bitcoin/bitcoin/blob/master/src/chain.h :

```
class CBlockIndex
{
    // ...
    //! pointer to the index of
    // the predecessor of this block
    CBlockIndex* pprev{nullptr};
    // ...
}
```

Liste chaînée

Pour une liste de taille n:

Opération	Complexité
taille	
test liste vide	
accéder/supprimer/ajouter au début	
accéder/supprimer/ajouter en position qcq	
recherche élément	
`l1 @ l2`	

Liste chaînée

Pour une liste de taille n:

Opération	Complexité
taille	O(n)
test liste vide	O(1)
accéder/supprimer/ajouter au début	O(1)
accéder/supprimer/ajouter en position qcq	O(n)
recherche élément	O(n)
`l1 @ l2`	O(taille de l1)

Tableau

Pour un tableau de taille n:

Opération	Tableau	Tableau trié
taille	$O(1)$	$O(1)$
`Array.make n x`	$O(n)$	$O(n)$
`t.(i)`	$O(1)$	$O(1)$
`t.(i) <- ...`	$O(1)$	$O(n)$
recherche élément	$O(n)$	$O(\log(n))$

Il est impossible d'ajouter un élément à un tableau (la taille est fixée à la création du tableau).

Tableau

```
let t1 = [|0; 1; 2|];;
let t2 = t1;;
t2.(0) <- 3;;
```

Que vaut `t1`?

Tableau

```
let t1 = [|0; 1; 2|];;
let t2 = t1;;
t2.(0) <- 3;;
```

Que vaut `t1` ?

`t1` vaut alors `[|3; 1; 2|]` : la modification de `t2` se répercute sur `t1`.

Tableau : matrices

Ne surtout pas créer une matrice $n \times p$ en écrivant:

```
Array.make n (Array.make p 0)
```

Tableau : matrices

Ne surtout pas créer une matrice $n \times p$ en écrivant:

```
Array.make n (Array.make p 0)
```

Utiliser `Array.make_matrix n p e` à la place, ou le recoder :

```
let make_matrix n p e =
    let m = Array.make n [||] in
    for i = 0 to n - 1 do
        m.(i) <- Array.make p e
    done;
    m
```

Tableau : matrices

Tableau en C :

```c
int array[10]; // tableau contenant 10 entiers
array[0] = 3; // modification du premier élément
printf("%d", array[0]); // affiche 3

for(int i = 0; i < 10; i++) // parcours de tableau
    array[i] = 1;
```

Tableau : tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

Tableau : tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

Idée 1 : créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité : $\mathrm{O}(n)$

Tableau : tableau dynamique

Idée 2 (**tableau dynamique**) : créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments.

Tableau : tableau dynamique

Idée 2 (**tableau dynamique**) : créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments.
Si l'on réalise n ajouts à partir d'un tableau de taille 1, il y a des recopies pour les tailles 1, 2, 4, ..., $2^{\lfloor \log_2(n) \rfloor}$ et le nombre total de valeurs recopiées est

$$\sum_{k=0}^{\lfloor \log_2(n) \rfloor} 2^k = \mathsf{O}(2^{\lfloor \log_2(n) \rfloor}) = \mathsf{O}(n)$$

Complexité **amortie** (moyenné sur n opérations) :

Tableau : tableau dynamique

Idée 2 (**tableau dynamique**) : créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments.
Si l'on réalise n ajouts à partir d'un tableau de taille 1, il y a des recopies pour les tailles 1, 2, 4, ..., $2^{\lfloor \log_2(n) \rfloor}$ et le nombre total de valeurs recopiées est

$$\sum_{k=0}^{\lfloor \log_2(n) \rfloor} 2^k = \mathrm{O}(2^{\lfloor \log_2(n) \rfloor}) = \mathrm{O}(n)$$

Complexité **amortie** (moyenné sur n opérations) : $\mathrm{O}(1)$

Remarque : une `list` de Python est en fait un tableau dynamique!

Tableau : tableau dynamique

```
type 'a dyn = {mutable t : 'a array; mutable n : int};;

let copy t1 t2 = (* copie t1 dans t2 *)
  for i = 0 to Array.length t1 - 1 do
    t2.(i) <- t1.(i)
  done;;

let add e d =
    if d.n < Array.length d.t then d.t.(d.n) <- e
    else if d.n = 0 then d.t <- [|e|]
    else let t' = Array.make (2*d.n) d.t.(0) in
        (copy d.t t'; t'.(d.n) <- e; d.t <- t');
    d.n <- d.n + 1 (* exécuté dans tous les cas *)
```

Tableau : tableau dynamique

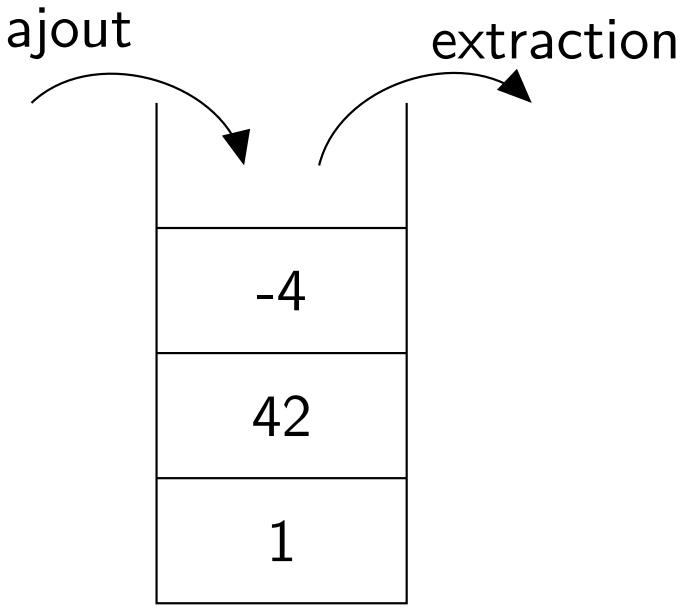
Ne pas confondre :

1. Complexité **amortie** : complexité moyennée sur un certain nombre d'appels de la fonction.
2. Complexité **en moyenne** : complexité moyennée sur toutes les entrées possibles.

Pile (stack)

Une pile est une structure de donnée possédant trois opérations:

- `push` : Ajout d'un élément au dessus de la pile.
- `pop` : Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.
- `empty` : Test pour savoir si la pile est vide.



Pile contenant 1, 42 et -4

Pile (stack) : applications

- **Call stack** (pile d'appel) qui stocke les appels de fonctions en cours d'exécution.
- Passer de récursif à itératif en simulant des appels récursifs (cf TD).
- Dans un éditeur de texte, chaque action est ajoutée à une pile. Lorsque vous revenez en arrière (Ctrl + Z), l'éditeur extrait le dessus de la pile pour revenir à l'état précédent.
- Codage de Huffman

Pile (stack) : implémentation fonctionnelle

La façon la plus naturelle d'implémenter une pile en OCaml est d'utiliser une liste, où la tête de liste correspond au dessus de la pile.

Pile (stack) : implémentation fonctionnelle

La façon la plus naturelle d'implémenter une pile en OCaml est d'utiliser une liste, où la tête de liste correspond au dessus de la pile.

La pile suivante est représentée par `[-4; 42; 1]` :

Pile (stack) : implémentation fonctionnelle

La façon la plus naturelle d'implémenter une pile en OCaml est d'utiliser une liste, où la tête de liste correspond au dessus de la pile.

```
let stack_empty p = p = [];;

let stack_push p e = e::p;;

let stack_pop p = match p with
    | [] -> failwith "Pile vide"
    | e::q -> (e, q);;
```

Chaque opération a complexité O(1).

Pile (stack) : implémentation impérative

On peut aussi stocker les éléments de la pile dans un tableau, où le haut de pile est l'élément de plus grand indice.

Pile (stack) : implémentation impérative

On peut aussi stocker les éléments de la pile dans un tableau, où le haut de pile est l'élément de plus grand indice.

Comme on ne peut pas augmenter la taille d'un tableau, il faut construire un grand tableau pour pouvoir rajouter des éléments. Le nombre d'éléments de la pile est donc potentiellement inférieur à la taille du tableau.

Pile (stack) : implémentation impérative

On peut aussi stocker les éléments de la pile dans un tableau, où le haut de pile est l'élément de plus grand indice.

Comme on ne peut pas augmenter la taille d'un tableau, il faut construire un grand tableau pour pouvoir rajouter des éléments. Le nombre d'éléments de la pile est donc potentiellement inférieur à la taille du tableau.
(Remarque : on pourrait utiliser un tableau dynamique à la place)

Pile (stack) : implémentation impérative

On peut aussi stocker les éléments de la pile dans un tableau, où le haut de pile est l'élément de plus grand indice.

Comme on ne peut pas augmenter la taille d'un tableau, il faut construire un grand tableau pour pouvoir rajouter des éléments. Le nombre d'éléments de la pile est donc potentiellement inférieur à la taille du tableau.
(Remarque : on pourrait utiliser un tableau dynamique à la place)

On conserve l'indice correspondant au haut de la pile, qu'on augmente à chaque ajout d'élément.
Il faut faire attention à ne pas dépasser du tableau (la taille maximum de la pile doit être choisie au moment de la création du tableau).

Pile (stack) : implémentation impérative

```
type 'a stack = {t : 'a array; mutable n : int}
```

On utilise `n` pour connaître le nombre d'éléments dans la pile.

```
let stack_empty p = p.n = 0;;  (* O(1) *)

let stack_push p e = (* O(1) *)
    if p.n >= Array.length p.t
    then failwith "Pile pleine"
    else (p.t.(p.n) <- e; p.n <- p.n + 1);;

let stack_pop p = (* O(1) *)
    if p.n = 0 then failwith "Pile vide"
    else (p.n <- p.n - 1; p.t.(p.n))
```

Pile (stack) : type abstrait

On peut définir un type abstrait possédant les opérations de pile :

```
type 'a stack_imperative = {
    empty : unit -> bool;
    push : 'a -> unit;
    pop : unit -> 'a
};;
```

Vu les types des opérations, il s'agit d'une pile impérative (mutable). Une valeur de type `stack` correspond à une implémentation concrète de pile impérative.

Pile (stack) : type abstrait

Implémentation par tableau :

```
let stack_of_array t =
    let n = ref 0 in {
    empty = (fun () -> !n = 0);
    push = (fun e -> if !n >= Array.length t
             then failwith "Pile pleine"
             else (t.(!n) <- e; incr n);
    pop = (fun () -> if !n = 0
            then failwith "Pile vide"
            else (decr n; t.(!n)))
    }
```

`stack_of_array t` donne une valeur de type `stack` en utilisant un tableau.

Pile (stack) : type abstrait

Implémentation par liste :

```
let stack_of_list l =
    let l = ref l in {
    empty = (fun () -> !l = []);
    push = (fun e -> l := e::!l);
    pop = (fun () -> let e::q = !l in
                     l := q; e)
    }
```

Pile (stack) : type abstrait

Intérêts de ces types abstraits :

- Utiliser la structure abstraite (les opérations) sans se préoccuper de l'implémentation concrète
- Avoir un algorithme qui fonctionne sur n'importe quelle implémentation concrète de pile

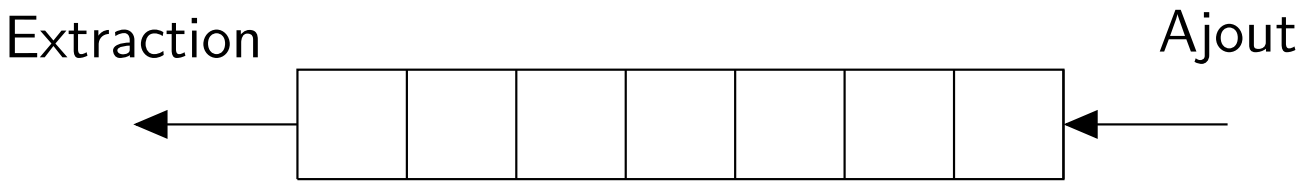
Exercice

1. Écrire une fonction `stack_print : int stack -> unit` affichant les éléments d'une pile.
2. Modifier votre fonction pour éviter de modifier la pile en argument.

File (queue)

Une file est une structure de donnée possédant trois opérations:

- Ajout d'un élément à la fin de la file.
- Extraction (suppression et renvoi) de l'élément au début de file. Ainsi, l'élément extrait est l'élément le plus ancien.
- Test pour savoir si la file est vide.



File (queue) : applications

Application des files :

- Parcours en largeur dans un graphe
- CPU scheduling (pour que les programmes soient exécutés à tour de rôle)

File (queue) : implémentation persistante avec 2 listes

On pourrait utiliser une liste pour implémenter une file, en ajoutant d'un côté et supprimant de l'autre.
Problème :

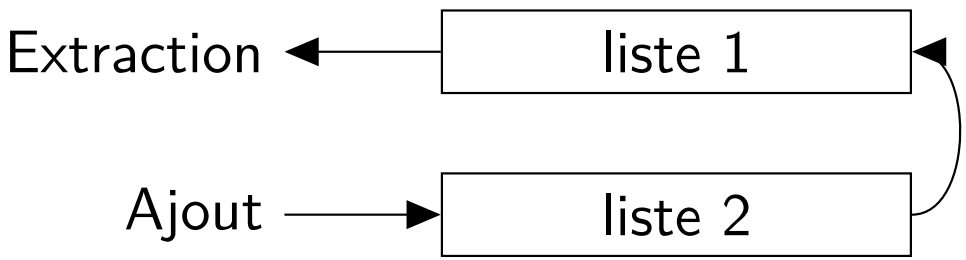
File (queue) : implémentation persistante avec 2 listes

On pourrait utiliser une liste pour implémenter une file, en ajoutant d'un côté et supprimant de l'autre.
Problème : il faut une complexité linéaire pour accéder au dernier élément d'une liste.

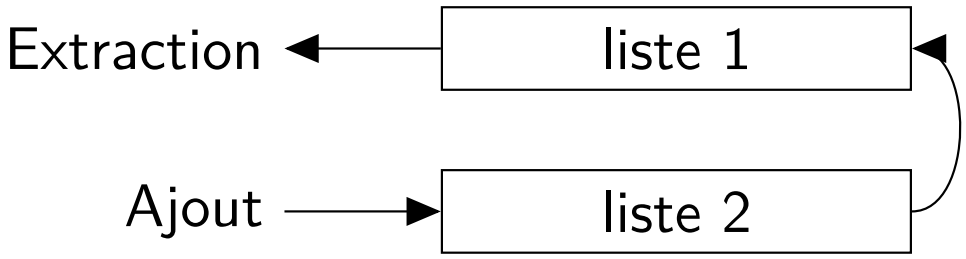
File (queue) : implémentation persistante avec 2 listes

On pourrait utiliser une liste pour implémenter une file, en ajoutant d'un côté et supprimant de l'autre.
Problème : il faut une complexité linéaire pour accéder au dernier élément d'une liste.

À la place on utilise deux listes, où le dernier élément de la 1ère liste est juste avant le dernier élément de la 2ème liste :

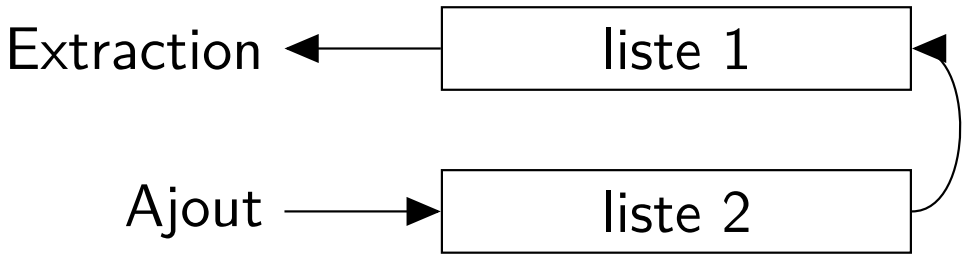


File (queue) : implémentation persistante avec 2 listes



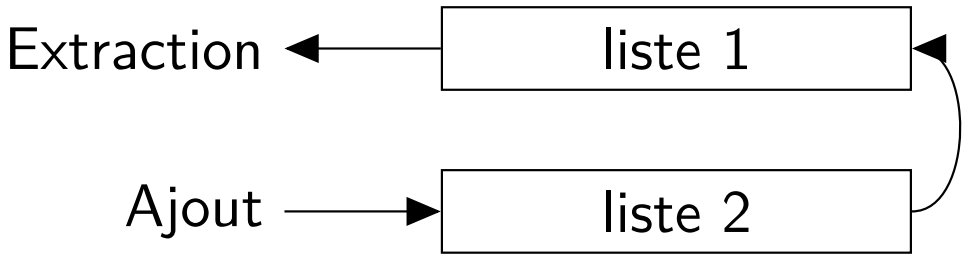
- Pour ajouter un élément :

File (queue) : implémentation persistante avec 2 listes



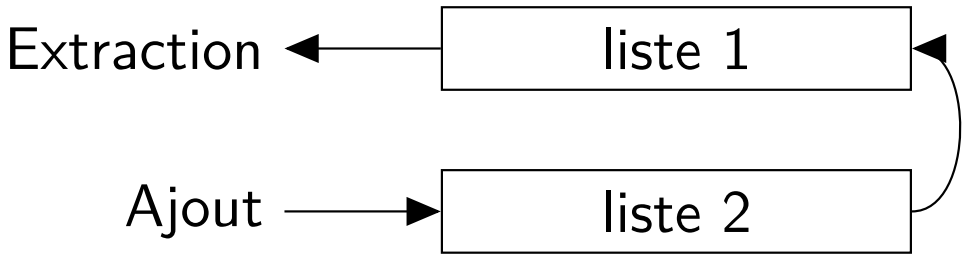
- Pour ajouter un élément : on le met au début de la liste 2 en O(1)
- Pour extraire un élément :

File (queue) : implémentation persistante avec 2 listes



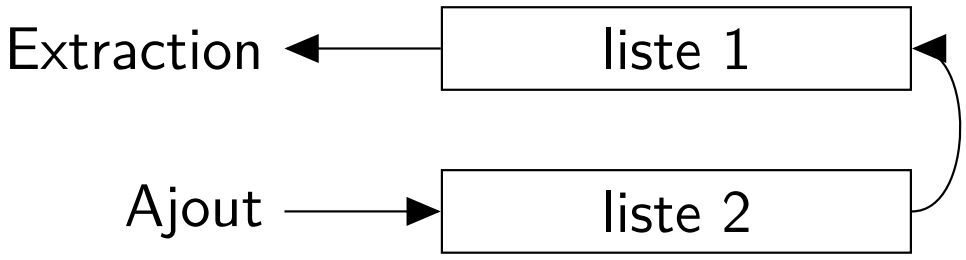
- Pour ajouter un élément : on le met au début de la liste 2 en O(1)
- Pour extraire un élément :
  si la liste 1 n'est pas vide, on extrait au début de la liste 1 en O(1)
  si la liste 1 est vide,

File (queue) : implémentation persistante avec 2 listes



- Pour ajouter un élément : on le met au début de la liste 2 en O(1)
- Pour extraire un élément :
  si la liste 1 n'est pas vide, on extrait au début de la liste 1 en O(1)
  si la liste 1 est vide, on renverse la liste 2 et on la met à la place de la liste 1

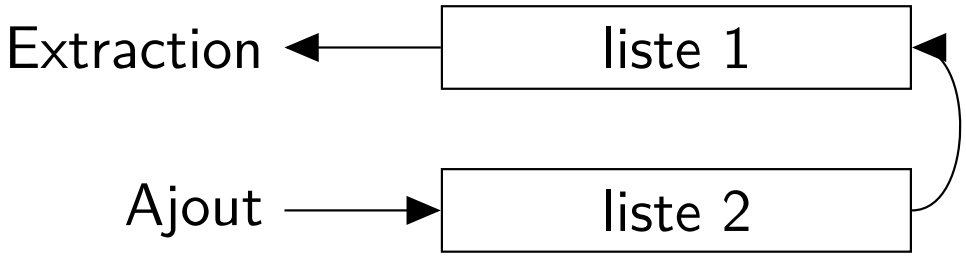
File (queue) : implémentation persistante avec 2 listes



- Pour ajouter un élément : on le met au début de la liste 2 en O(1)
- Pour extraire un élément :
  si la liste 1 n'est pas vide, on extrait au début de la liste 1 en O(1)
  si la liste 1 est vide, on renverse la liste 2 et on la met à la place de la liste 1

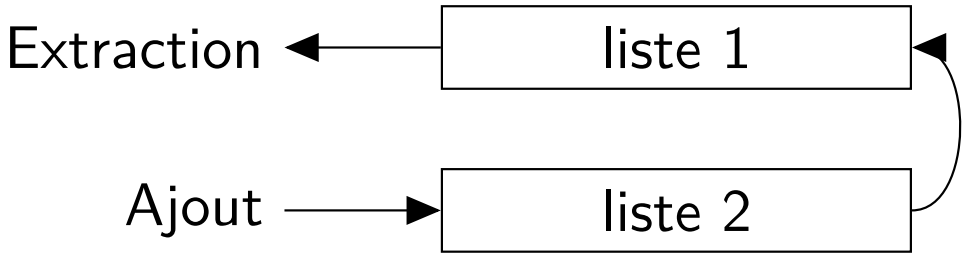
L'extraction demande un renversement (avec `List.rev`) donc est linéaire dans le pire cas... mais O(1) en complexité amortie.

File (queue) : implémentation persistante avec 2 listes




```
type 'a queue_2lists = {l1 : 'a list; l2 : 'a list}
```

File (queue) : implémentation persistante avec 2 listes




```
type 'a queue_2lists = {l1 : 'a list; l2 : 'a list}
```

```
let queue_empty f = f.l1 = [] && f.l2 = [];;

let queue_add f e = {l1 = e::f.l1; l2 = f.l2};;

(* suppose que f est non vide *)
(* renvoie (élément, liste obtenue) *)
let rec queue_pop f = match f.l1 with
    | e::q -> e, {l1 = q; l2 = f.l2}
    | [] -> queue_pop {l1 = List.rev f.l2; l2 = []};;
```

File (queue) : implémentation persistante avec 2 listes

Question

Si on effectue n ajouts et n extractions dans un ordre quelconque (en partant d'une file vide), quelle sera la complexité totale des n extractions?

File (queue) : implémentation persistante avec 2 listes

Question

Si on effectue n ajouts et n extractions dans un ordre quelconque (en partant d'une file vide), quelle sera la complexité totale des n extractions?

Chaque élément est « renversé » exactement une fois, donc la complexité totale des `List.rev` est le nombre total de renversements, c'est à dire $O(n)$.
Donc la complexité totale des n extractions est $O(n)$.

File (queue) : implémentation persistante avec 2 listes

Question

Si on effectue n ajouts et n extractions dans un ordre quelconque (en partant d'une file vide), quelle sera la complexité totale des n extractions?

Chaque élément est « renversé » exactement une fois, donc la complexité totale des `List.rev` est le nombre total de renversements, c'est à dire $\mathrm{O}(n)$.
Donc la complexité totale des n extractions est $\mathrm{O}(n)$.
La complexité moyenne d'une extraction (**complexité amortie**) est donc $\frac{\mathrm{O}(n)}{n} = \boxed{\mathrm{O}(1)}$.

File (queue) : implémentation avec tableau

Fait en TD (sujet CentraleSupélec) :

On veut implémenter une file d'attente à l'aide d'un vecteur circulaire. On définit pour cela un type particulier nommé `file` par

```
type 'a file={tab:'a array ; mutable deb: int ; mutable fin: int ; mutable vide: bool}
```

`deb` indique l'indice du premier élément dans la file et `fin` l'indice qui suit celui du dernier élément de la file, `vide` indiquant si la file est vide. Les éléments sont rangés depuis la case `deb` jusqu'à la case précédent `fin` en repartant à la case 0 quand on arrive au bout du vecteur (cf exemple). Ainsi, on peut très bien avoir l'indice `fin` plus petit que l'indice `deb`. Par exemple, la file figure 5 contient les éléments 4, 0, 1, 12 et 8, dans cet ordre, avec `fin=2` et `deb=9`.

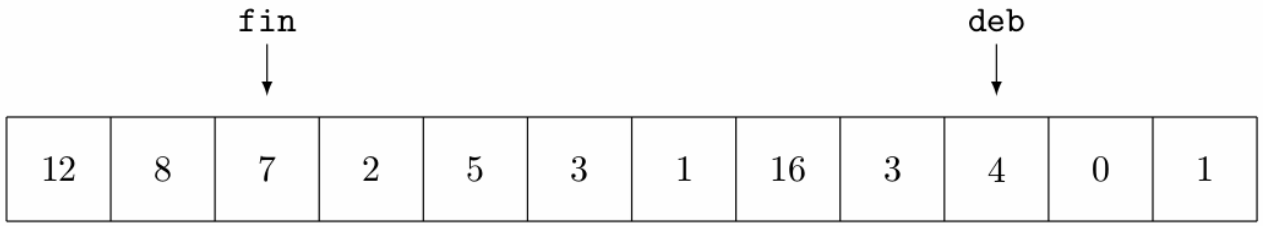
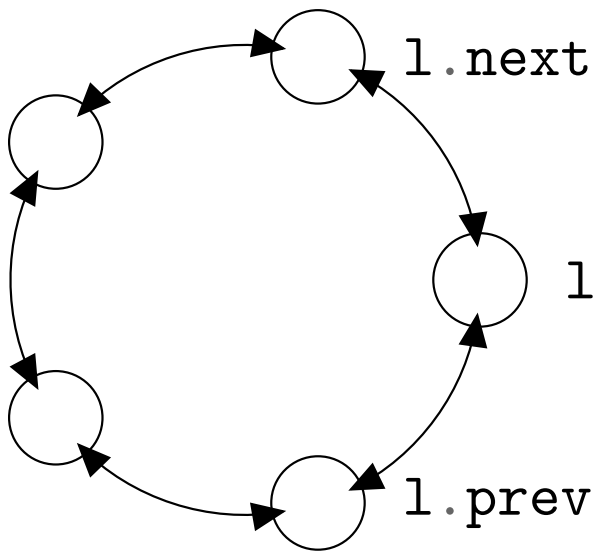


Figure 5 Un exemple de file où `fin` < `deb`

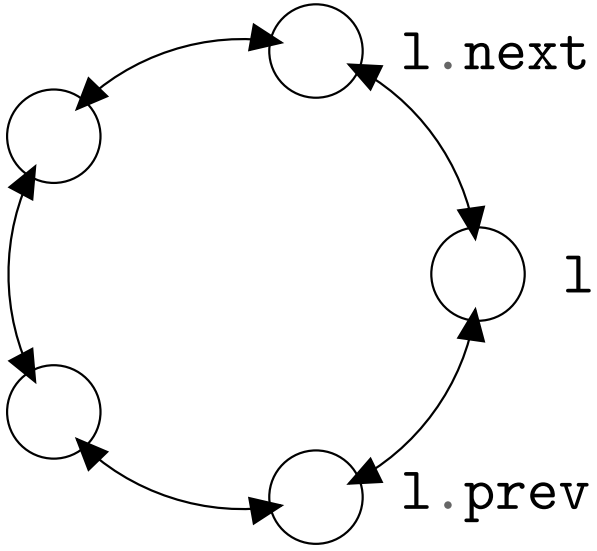
File (queue) : avec liste doublement chaînée cyclique

```
type 'a l2c = {
    elem : 'a;
    mutable prev : 'a l2c;
    mutable next : 'a l2c
}
```



File (queue) : avec liste doublement chaînée cyclique

On peut même implémenter une *doubly-ended queue (deque)* qui permet de rétirer/ajouter un élément aussi bien au début qu'à la fin.



File (queue) : librairie standard OCaml

La stdlib OCaml implémente les files avec une liste chaînée :

```
type 'a cell =
    | Nil
    | Cons of { content: 'a; mutable next: 'a cell }

type 'a t = {
    mutable length: int;
    mutable first: 'a cell;
    mutable last: 'a cell
}
```