Structure de données, Type Abstrait : Piles, Files

13 novembre 2023

Plan

Administratif

Informations et rappels

- Vous avez un DM à rendre la semaine prochaine ;
- Vous avez un deuxième DS d'informatique la semaine d'après ;
- La semaine prochaine, je vous donnerez des informations supplémentaires sur les choix pour le second semestre.

Erreurs dans le DM

Il y a quelques erreurs dans le DM :

- Quelques mentions d'entiers naturels dans la deuxième section qui sont en réalité des entiers relatifs (la dernière section est bien seulement sur les entiers naturels) ;
- Quelques erreurs de k au lieu de $k+1$.

Retour sur la semaine dernière

Test d'égalité avec un filtrage

```
1 let egal x y = match x with
2 | y -> true
3 | _ -> false
```

Test d'égalité avec un filtrage

```
1 let egal x y = match x with
2 | y -> true
3 | _ -> false
```

```
1 let egal x y = match x with
2 | _ when x = y -> true
3 | _ -> false
```

Test d'égalité avec un filtrage

```
1 let egal x y = match x with
2 | y -> true
3 | _ -> false
```

```
1 let egal x y = match x with
2 | _ when x = y -> true
3 | _ -> false
```

```
1 let egal x y = if x = y
2     then true
3     else false
```

Test d'égalité avec un filtrage

```
let egal x y = match x with
| y -> true
| _ -> false
```

```
let egal x y = match x with
| _ when x = y -> true
| _ -> false
```

```
let egal x y = if x = y
    then true
    else false
```

```
let egal x y = x = y
```

Égalité physique, égalité structurelle

On peut distinguer l'égalité physique == et l'égalité structurelle =.

Égalité physique, égalité structurelle

On peut distinguer l'égalité physique == et l'égalité structurelle =.

Il y a des comportements assez imprévisibles dans l'utilisation de l'égalité physique de manière générale, et c'est lié à la construction des objets en OCaml. L'une des seules choses que l'on peut savoir, c'est que l'égalité physique implique l'égalité structurelle.

Égalité physique, égalité structurelle

On peut distinguer l'égalité physique == et l'égalité structurelle =.

Il y a des comportements assez imprévisibles dans l'utilisation de l'égalité physique de manière générale, et c'est lié à la construction des objets en OCaml. L'une des seules choses que l'on peut savoir, c'est que l'égalité physique implique l'égalité structurelle.

De la même manière, on a la différence physique != et la différence structurelle <>.

Utilisation du type option

```
let ajouter x y = match x, y with
| _, None | None, _ -> None
| Some vx, Some vy -> Some (vx +. vy)
```

Utilisation du type option

```
let ajouter x y = match x, y with
| _, None | None, _ -> None
| Some vx, Some vy -> Some (vx +. vy)
```

```
let racine x = match x with
| None | Some vx when vx < 0. -> None
| Some vx -> Some (sqrt vx)
```

Attention

J'ai eu beaucoup de questions sur de vieilles syntaxes : vous devez pouvoir retrouver les syntaxes qui vous manquent sur des objets que vous avez déjà vu (chaînes de caractères, n-uplets, ...).

Structure de données

Type Abstrait

Définition 1 : Type Abstrait

Un **type abstrait** est un objet mathématique et les fonctions que l'on peut appliquer sur ces objets.

Exemples : ensembles, listes, dictionnaire...

Structure de donnée concrète

Définition 2 : Structure de donnée concrète

Une **structure de donnée concrète** est la réalisation en pratique d'un point de vue informatique et machine d'un type abstrait.

Exemples : variable, constante, tableau, tableau dynamique...

Ainsi, une structure de donnée concrète est l'implémentation d'un type abstrait.
Généralement, c'est la structure concrète qui détermine la complexité et les éventuelles limites, même si on peut définir un type abstrait comme ayant des spécifications de complexités sans en préciser l'implémentation.

Remarques sur la terminologie

On peut parler de *structure abstraite* au lieu de type abstrait, et de même on peut parler de *type concret* au lieu de structure de donnée concrète. Parfois *structure de donnée* peut désigner un type abstrait, mais de manière un peu plus concrète (par exemple en en précisant les complexité).
La distinction principale est dans la vision *mathématique* ou *informatique* de l'objet.

Piles et files

Principe d'une pile (1)

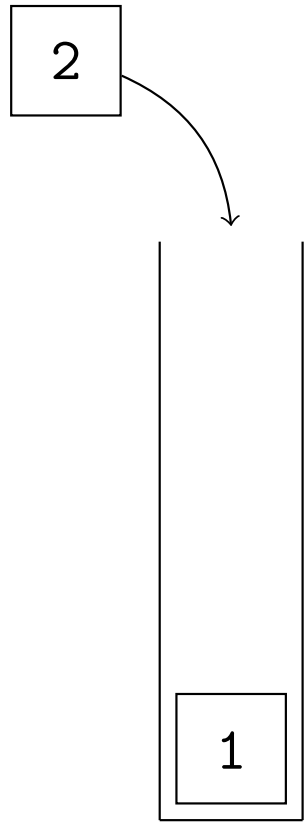
Définition 3 : Pile

Une **pile** est une structure de donnée linéaire dans laquelle on peut ajouter et retirer des éléments avec la contrainte qu'on ne peut retirer que l'élément encore dans la pile qui a été ajouté le plus tard.

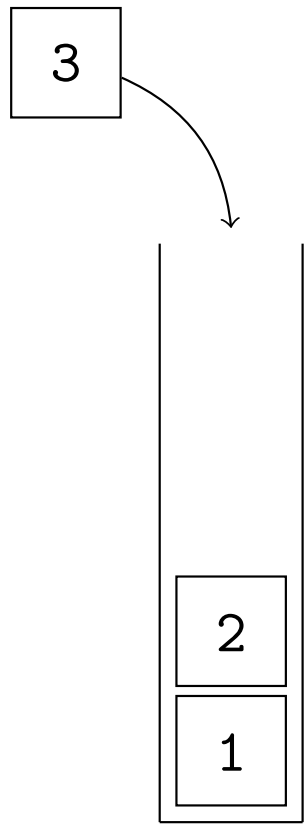
Une pile est ainsi une structure sur laquelle on empile des éléments, mais pour laquelle, on ne peut retirer que l'élément qui est au sommet de la pile.

En anglais, on parle de structure *LIFO* (*Last In First Out*) ou de *Stack*.

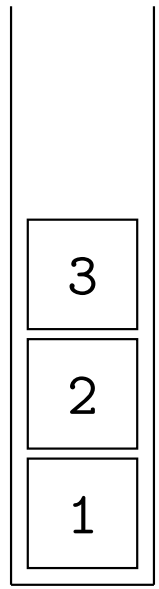
Principe d'une pile



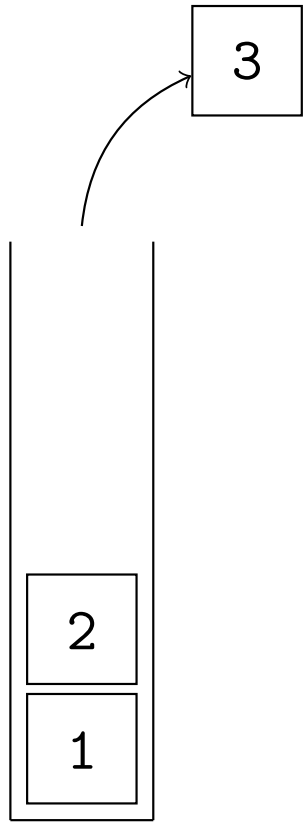
Principe d'une pile



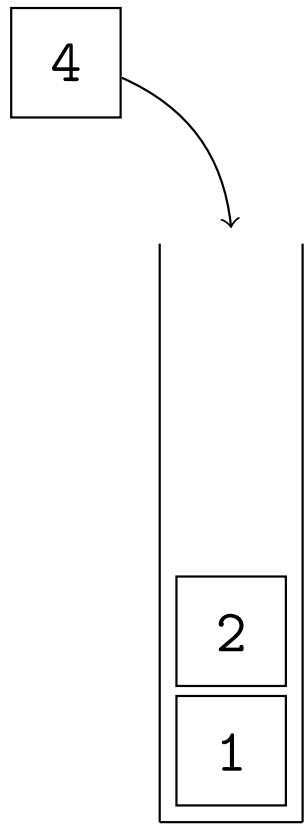
Principe d'une pile



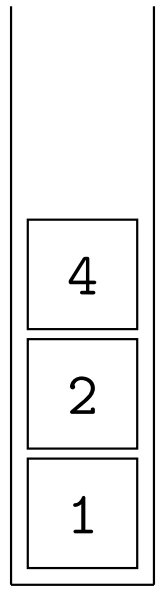
Principe d'une pile



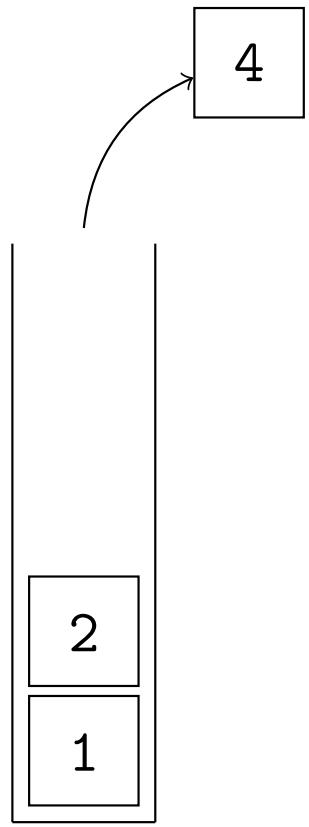
Principe d'une pile



Principe d'une pile



Principe d'une pile



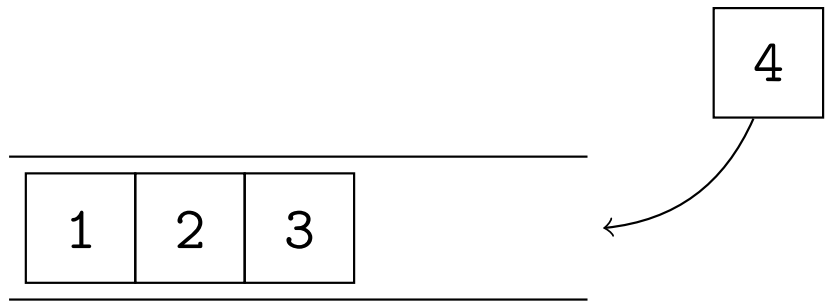
Principe d'une file

Définition 4 : File

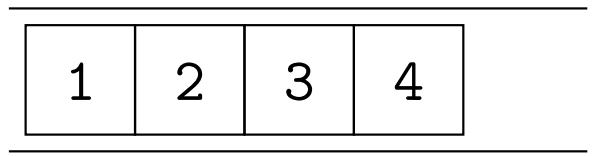
Une **file** est une structure de donnée linéaire dans laquelle on peut ajouter et retirer des éléments avec la contrainte qu'on ne peut retirer que l'élément encore dans la pile qui a été ajouté le plus tôt.

Ainsi, une file est une structure linéaire dans laquelle
En anglais, on parle de structure *FIFO* (*first in first out*) ou de *Queue*.

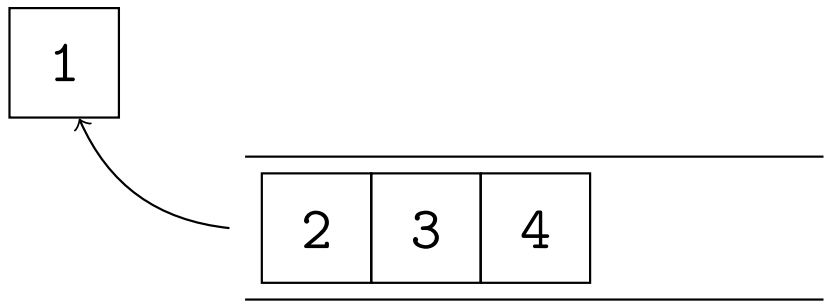
Principe d'une file



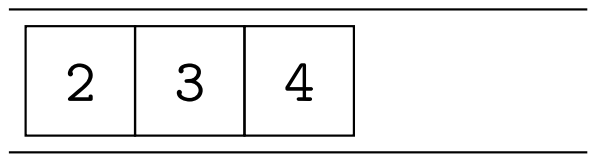
Principe d'une file



Principe d'une file



Principe d'une file



Rappel : donée mutable, donnée persistante

Définition 5 : Donnée mutable

Une **donnée mutable** est une donnée qui peut être modifiée.

Ex : référence, tableau, ...

Définition 6 : Donnée persistante

Une **donnée persistante** est une donnée qui ne peut pas être modifiée.

Ex : liste OCaml, constante, variable OCaml, ...

Quand on veut modifier une donnée persistante, il est nécessaire de construire la structure modifiée et de la renvoyer. Dans le cas d'une donnée mutable, on peut se contenter de faire la modification directement.

Interface de programmation

Définition 7 : Interface

L' **interface de programmation** d'une composante logicielle est l'ensemble des fonctions et constantes qu'elle met à disposition pour l'utiliser.

Une interface est généralement concrète : elle définit exactement le nom des fonction, leur type, l'ordre des arguments...

Interface d'une pile

On peut proposer l'interface suivante pour une pile en OCaml implémentée de manière persistante :

- Le type `'a pile` ;
- `creer` de signature `unit -> 'a pile` qui crée une pile vide ;
- `empiler` de signature `'a pile -> 'a -> unit` qui ajoute un élément dans la pile ;
- `depiler` de signature `'a pile -> 'a * 'a pile` qui retire un élément au sommet de la pile et le renvoie, ainsi que la pile modifiée ;
- `est_vide` de signature `'a pile -> bool` qui renvoie si une pile est vide.

On peut par ailleurs rajouter quelques fonctions qui sont parfois utilisées :

- `regarder` de signature `'a pile -> 'a` qui renvoie l'élément au sommet de la pile mais sans le retirer ;
- `taille` de signature `'a pile -> int` qui renvoie le nombre d'éléments dans la pile actuellement.

Interface d'une file

On peut proposer l'interface suivante pour une file en OCaml implémentée de manière persistante :

- Le type `'a file` ;
- `creer` de signature `unit -> 'a file` qui crée une file vide ;
- `enfiler` de signature `'a file -> 'a -> unit` qui ajoute un élément dans la file ;
- `defiler` de signature `'a file -> 'a * 'a file` qui retire le premier élément à retirer de la file et le renvoie ainsi que la file modifiée ;
- `est_vide` de signature `'a file -> bool` qui renvoie si une file est vide.

On peut par ailleurs rajouter quelques fonctions qui sont parfois utilisées :

- `regarder` de signature `'a file -> 'a` qui renvoie le prochain élément à retirer mais sans le retirer ;
- `taille` de signature `'a file -> int` qui renvoie le nombre d'éléments dans la file actuellement.

Autres notions d'interface

Le concept d'interface est plus générale : il désigne généralement une frontière de communication entre deux entités qui ne sont pas nécessairement logicielles. On peut parler d'interface aussi pour le matériel ou pour l'interraction humain-machine.
En règle générale, quand nous parlerons d'interface, nous parlerons d'interface de programmation.

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
1 type 'a pile = 'a list
```

Comment créer une pile vide ?

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
1 type 'a pile = 'a list
```

Comment créer une pile vide ?

```
1 let creer () = []
```

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
1 type 'a pile = 'a list
```

Comment créer une pile vide ?

```
1 let creer () = []
```

Comment ajouter un élément dans la pile ?

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Comment retire un élément dans la pile ?

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Comment retire un élément dans la pile ?

```
let depiler l = let x::q = l in x, q
```

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Comment retire un élément dans la pile ?

```
let depiler l = let x::q = l in x, q
```

Comment vérifier si une pile est vide ?

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Comment retire un élément dans la pile ?

```
let depiler l = let x::q = l in x, q
```

Comment vérifier si une pile est vide ?

```
let est_vide l = l = []
```

Implémentation persistante d'une pile avec une liste

On propose d'implémenter une pile à l'aide d'une liste. On se donne le type suivant :

```
type 'a pile = 'a list
```

Comment créer une pile vide ?

```
let creer () = []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = x::l
```

Comment retire un élément dans la pile ?

```
let depiler l = let x::q = l in x, q
```

Comment vérifier si une pile est vide ?

```
let est_vide l = l = []
```

Quelle est la complexité de ces fonctions ?

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
1 type 'a pile = 'a list ref
```

Comment créer une pile vide ?

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l := x::!l
```

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l := x::!l
```

Comment retire un élément dans la pile ? On peut changer l'interface pour ne pas renvoyer la pile modifiée.

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l:= x::!l
```

Comment retire un élément dans la pile ? On peut changer l'interface pour ne pas renvoyer la pile modifiée.

```
let depiler l = let x::q = !l in l:=q ; x
```

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l:= x::!l
```

Comment retire un élément dans la pile ? On peut changer l'interface pour ne pas renvoyer la pile modifiée.

```
let depiler l = let x::q = !l in l:=q ; x
```

Comment vérifier si une pile est vide ?

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l:= x::!l
```

Comment retire un élément dans la pile ? On peut changer l'interface pour ne pas renvoyer la pile modifiée.

```
let depiler l = let x::q = !l in l:=q ; x
```

Comment vérifier si une pile est vide ?

```
let est_vide l = !l = []
```

Implémentation mutable d'une pile avec une liste

On propose d'implémenter cette fois-ci la pile de manière mutable : au lieu de renvoyer la pile quand on la modifie, on décide de la modifier pour avoir une structure mutable :

```
type 'a pile = 'a list ref
```

Comment créer une pile vide ?

```
let creer () = ref []
```

Comment ajouter un élément dans la pile ?

```
let empiler l x = l:= x::!l
```

Comment retire un élément dans la pile ? On peut changer l'interface pour ne pas renvoyer la pile modifiée.

```
let depiler l = let x::q = !l in l:=q ; x
```

Comment vérifier si une pile est vide ?

```
let est_vide l = !l = []
```

Quelle est la complexité de ces fonctions ?

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une référence de liste.

```
1 type 'a file = 'a list
```

Comment créer une file vide ?

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une reférence de liste.

```
type 'a file = 'a list
```

Comment créer une file vide ?

```
let creer () = []
```

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une référence de liste.

```
1 type 'a file = 'a list
```

Comment créer une file vide ?

```
1 let creer () = []
```

Comment ajouter un élément dans la file ?

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une référence de liste.

```
1 type 'a file = 'a list
```

Comment créer une file vide ?

```
1 let creer () = []
```

Comment ajouter un élément dans la file ?

```
1 let enfiler l x = x::l
```

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une référence de liste.

```
1 type 'a file = 'a list
```

Comment créer une file vide ?

```
1 let creer () = []
```

Comment ajouter un élément dans la file ?

```
1 let enfiler l x = x::l
```

Comment vérifier si une file est vide ?

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une référence de liste.

```
type 'a file = 'a list
```

Comment créer une file vide ?

```
let creer () = []
```

Comment ajouter un élément dans la file ?

```
let enfiler l x = x::l
```

Comment vérifier si une file est vide ?

```
let est_vide l = l = []
```

Implémentation persistante d'une file avec une liste

On propose d'implémenter la file à l'aide d'une reférence de liste.

```
type 'a file = 'a list
```

Comment créer une file vide ?

```
let creer () = []
```

Comment ajouter un élément dans la file ?

```
let enfiler l x = x::l
```

Comment vérifier si une file est vide ?

```
let est_vide l = l = []
```

Quelle est la complexité de ces fonctions ?

Implémentation persistante d'une file avec une liste

```
1 type 'a file = 'a list
```

Comment retirer un élément dans la file ?

Implémentation persistante d'une file avec une liste

```
type 'a file = 'a list
```

Comment retirer un élément dans la file ?

```
let defiler l =
    let rec aux l = match l with
        | [] -> failwith "File vide"
        | [x] -> x, []
        | p::q -> let el, reste = aux q in
        el, p::reste
    in aux l
```

Implémentation persistante d'une file avec une liste

```
type 'a file = 'a list
```

Comment retirer un élément dans la file ?

```
let defiler l =
    let rec aux l = match l with
        | [] -> failwith "File vide"
        | [x] -> x, []
        | p::q -> let el, reste = aux q in
        el, p::reste
    in aux l
```

Quelle est la complexité de cette fonction ?

Implémentation persistante d'une file avec une liste

```
type 'a file = 'a list
```

Comment retirer un élément dans la file ?

```
let defiler l =
    let rec aux l = match l with
        | [] -> failwith "File vide"
        | [x] -> x, []
        | p::q -> let el, reste = aux q in
        el, p::reste
    in aux l
```

Quelle est la complexité de cette fonction ?
On va voir des exercices pour d'autres implémentations des files dont certaines plus efficaces ou permettent d'autres choses.

Utilisation des piles

- Pile des appels dans des appels de fonctions ;
- Parenthèsage ;
- Parcours en profondeur d'un graphe.

Utilisation des files

- File d'attente ;
- Mémoire tampon pour gérer des communications ;
- Parcours en largeur d'un graphe.

Module OCaml Stack

OCaml propose un module de pile [1] qui comprend une implémentation d'une pile qui peut être utilisée dans le cas où les exercices.

- `Stack.create` pour créer une nouvelle pile ;
- `Stack.push` pour ajouter un élément à la pile ;
- `Stack.pop` pour récupérer un élément.

Cette structure est mutable.

1. https://v2.ocaml.org/api/Stack.html

Module OCaml Queue

Il y a aussi un module de file[2] .

- Queue.create pour créer une nouvelle file ;
- Queue.push pour ajouter un élément à la file ;
- Queue.take pour récupérer un élément.

Cette structure est mutable.

2. https://v2.ocaml.org/api/Queue.html

Mot-clef open

De sorte à pouvoir simplifier du code, on peut vouloir utiliser le mot-clef `open` qui permet de rajouter les fonctions d'un module à l'environnement.

```
1 open Queue
2 let a = create ()
```

Mot-clef open

De sorte à pouvoir simplifier du code, on peut vouloir utiliser le mot-clef `open` qui permet de rajouter les fonctions d'un module à l'environnement.

```
1 open Queue
2 let a = create ()
```

Cela peut poser des problèmes de conflit dans les noms :

```
1 open Queue
2 open Stack
3 let a = create ()
```

En règle général, comme en Python, on préfère ne pas ouvrir de modules.