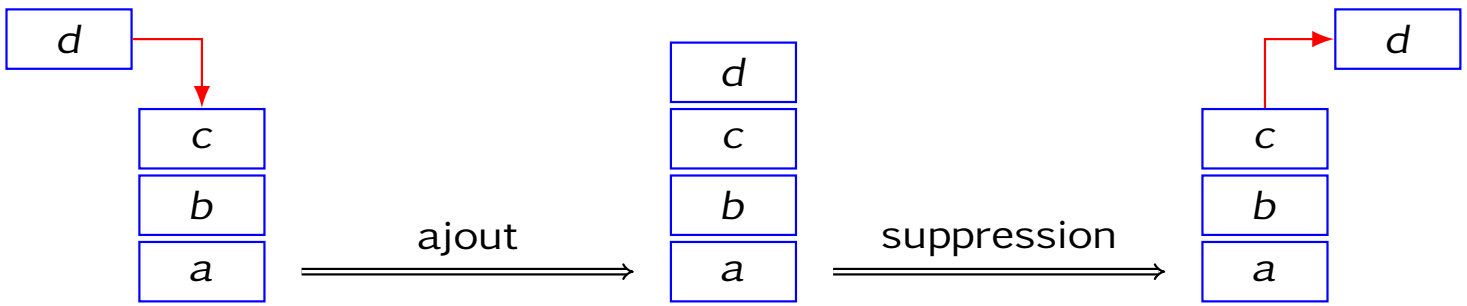
Piles et files

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Piles et Files

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures linéaires dynamiques, qui se distinguent par les conditions d'ajout et d'accès aux éléments.

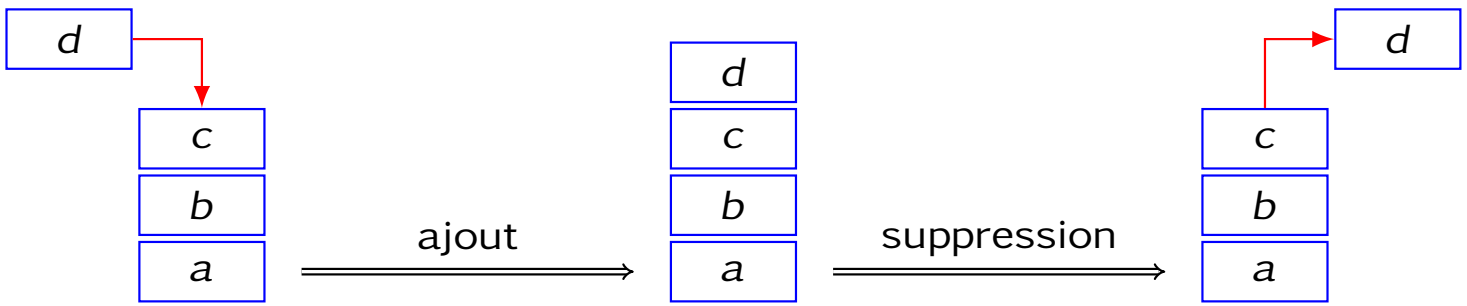
- Les piles sont fondées sur le principe du « dernier arrivé, premier sorti » (LIFO) :



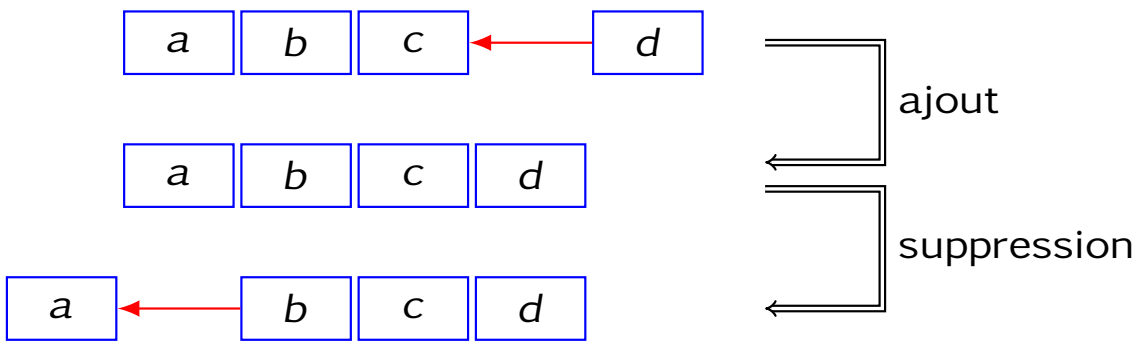
Piles et Files

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures linéaires dynamiques, qui se distinguent par les conditions d'ajout et d'accès aux éléments.

- Les piles sont fondées sur le principe du « dernier arrivé, premier sorti » (LIFO) :



- Les files sont fondées sur le principe du « premier arrivé, premier sorti » (FIFO) :



Piles et Files

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures linéaires dynamiques, qui se distinguent par les conditions d'ajout et d'accès aux éléments.

Primitives

Ces deux structures de données ont des spécifications très semblables. Toutes deux nécessitent :

- un constructeur permettant de créer une pile ou une file vide ;
- une fonction d'ajout d'un élément ;
- une fonction de suppression d'un élément et son renvoi ;
- une fonction permettant de tester si la pile ou la file est vide.

Piles et Files

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures linéaires dynamiques, qui se distinguent par les conditions d'ajout et d'accès aux éléments.

Primitives

Ces deux structures de données ont des spécifications très semblables. Toutes deux nécessitent :

- un constructeur permettant de créer une pile ou une file vide ;
- une fonction d'ajout d'un élément ;
- une fonction de suppression d'un élément et son renvoi ;
- une fonction permettant de tester si la pile ou la file est vide.

En CAML, le module `"stack"` de la bibliothèque standard est dédié aux piles, et le module `"queue"` aux files.

Primitives du module stack

Les piles d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle pile vide.
La fonction **push**, de type *'a -> 'a t -> unit*, empile un élément.
La fonction **pop**, de type *'a t -> 'a*, dépile le sommet de la pile.

Primitives du module stack

Les piles d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle pile vide.
La fonction **push**, de type *'a -> 'a t -> unit*, empile un élément.
La fonction **pop**, de type *'a t -> 'a*, dépile le sommet de la pile.

```
# let pile = stack__new () ;;
pile : '_a t = <abstr>
# for i = 1 to 5 do stack__push i pile done ;;
- : unit = ()
# for i = 1 to 5 do print_int (stack__pop pile) done ;;
54321- : unit = ()
```

Primitives du module stack

Les piles d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle pile vide.
La fonction **push**, de type *'a -> 'a t -> unit*, empile un élément.
La fonction **pop**, de type *'a t -> 'a*, dépile le sommet de la pile.

```
# let pile = stack__new () ;;
pile : '_a t = <abstr>
# for i = 1 to 5 do stack__push i pile done ;;
- : unit = ()
# for i = 1 to 5 do print_int (stack__pop pile) done ;;
54321- : unit = ()
```

L'exception **Empty** est déclenchée lorsqu'on applique **pop** à une pile vide.
Pour tester si une pile est vide, on rattrape l'exception :

```
# let est_vide p =
    try let x = stack__pop p in stack__push x p ; false
    with stack__Empty -> true ;;
est_vide : 'a t -> bool = <fun>
```

Primitives du module queue

Les files d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle file vide.
La fonction **add**, de type *'a -> 'a t -> unit*, enfile un élément.
La fonction **take**, de type *'a t -> 'a*, défile la tête de la file.

Primitives du module queue

Les files d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle file vide.
La fonction **add**, de type *'a -> 'a t -> unit*, enfile un élément.
La fonction **take**, de type *'a t -> 'a*, défile la tête de la file.

```
# let file = queue__new () ;;
file : '_a t = <abstr>
# for i = 1 to 5 do queue__add i file done ;;
- : unit = ()
# for i = 1 to 5 do print_int (queue__take file) done ;;
12345- : unit = ()
```

Primitives du module queue

Les files d'éléments de type *'a* ont pour type *'a t*.

La fonction **new**, de type *unit -> '_a t*, crée une nouvelle file vide.
La fonction **add**, de type *'a -> 'a t -> unit*, enfile un élément.
La fonction **take**, de type *'a t -> 'a*, défile la tête de la file.

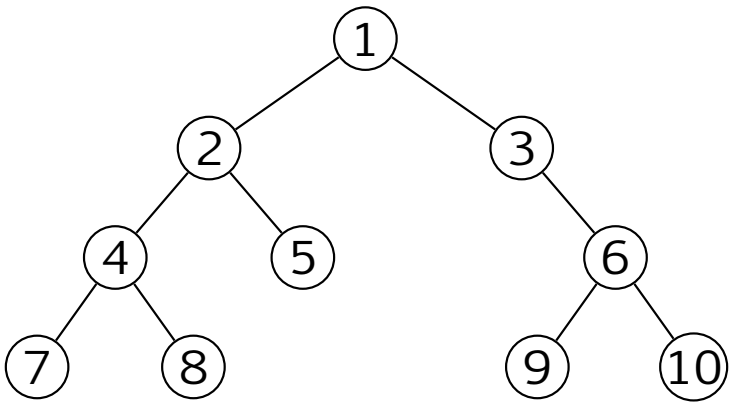
```
# let file = queue__new () ;;
file : '_a t = <abstr>
# for i = 1 to 5 do queue__add i file done ;;
- : unit = ()
# for i = 1 to 5 do print_int (queue__take file) done ;;
12345- : unit = ()
```

La fonction **peek** renvoie la tête de file sans la supprimer. On l'utilise pour déterminer si une file est vide :

```
# let est_vide f =
    try queue__peek f ; false with queue__Empty -> true ;;
est_vide : 'a t -> bool = <fun>
```

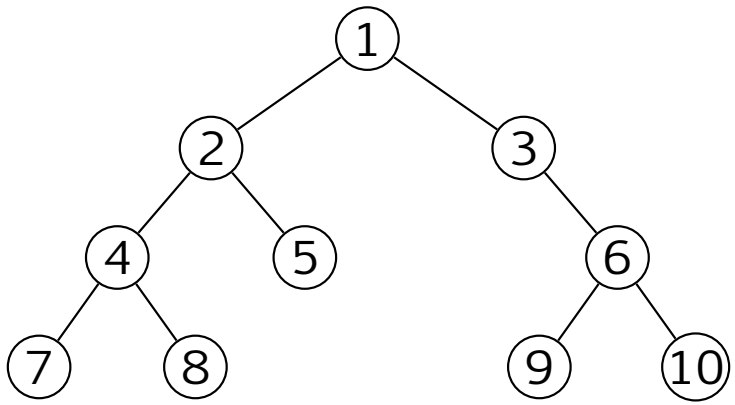
Parcours hiérarchique d'un arbre

Pour l'arbre suivant, il s'agit de l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :



Parcours hiérarchique d'un arbre

Pour l'arbre suivant, il s'agit de l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :

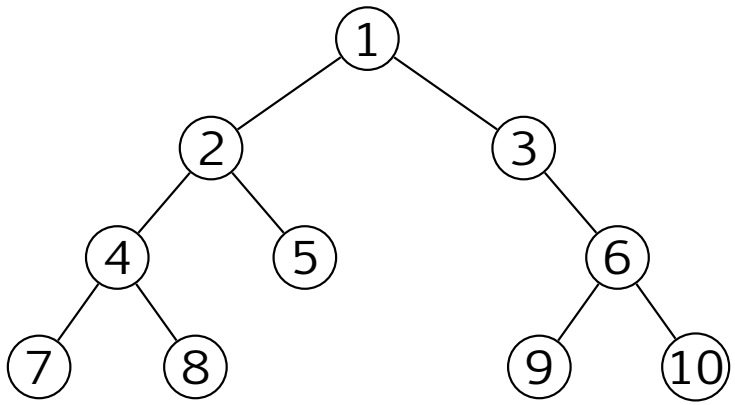


On utilise une file dans laquelle figure au départ le racine de l'arbre, et on adopte la règle :

1. la tête de liste est traitée ;
2. les fils de cet élément sont ajoutés en queue de liste ;

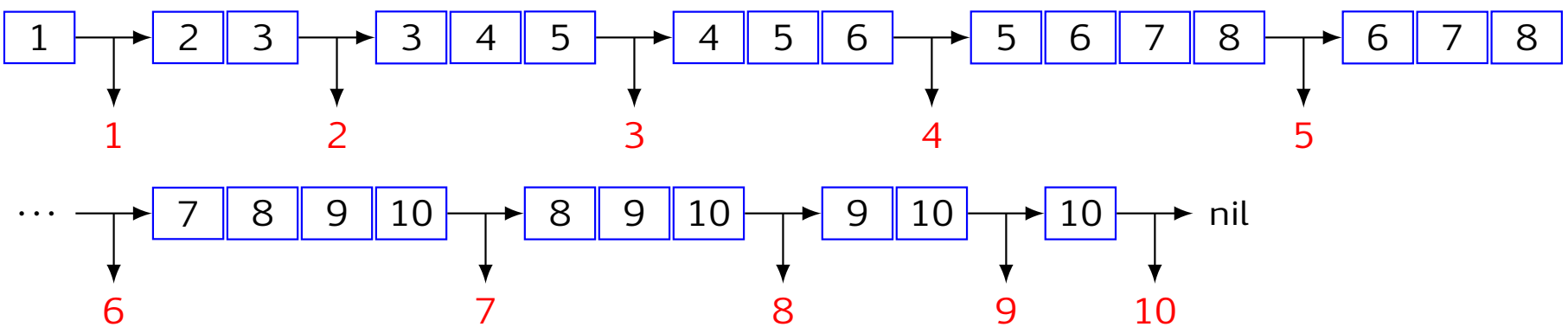
Parcours hiérarchique d'un arbre

Pour l'arbre suivant, il s'agit de l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :



On utilise une file dans laquelle figure au départ le racine de l'arbre, et on adopte la règle :

1. la tête de liste est traitée ;
2. les fils de cet élément sont ajoutés en queue de liste ;



Parcours hiérarchique d'un arbre

Mise en œuvre

On suit la règle :

1. la tête de liste est traitée ;
2. les fils de cet élément sont ajoutés en queue de liste ;

jusqu'à exhaustion de la file.

```
# type 'a btree = Nil | Node of 'a * 'a btree * 'a btree ;;
Type btree defined.
# let parcours arbre =
    let file = new () in
    add arbre file ;
    let rec aux () = match (take file) with
      | Nil                          -> aux ()
      | Node (r, fils_g, fils_d) -> print_int r ;
                                    add fils_g file ;
                                    add fils_d file ;
                                    aux ()
    in try aux () with Empty -> () ;;
parcours : int arbre -> unit = <fun>
```

Représentation d'une expression mathématique

La formule $\frac{1+2\sqrt{3}}{4}$ est représentée :

- sous forme infixe par l'expression $(1 + (2 \times (\sqrt{} 3))) \div 4$

Représentation d'une expression mathématique

La formule $\frac{1+2\sqrt{3}}{4}$ est représentée :

- sous forme infixe par l'expression $(1 + (2 \times (\sqrt{} 3))) \div 4$
- sous forme postfixe par l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$

Représentation d'une expression mathématique

La formule $\frac{1+2\sqrt{3}}{4}$ est représentée :

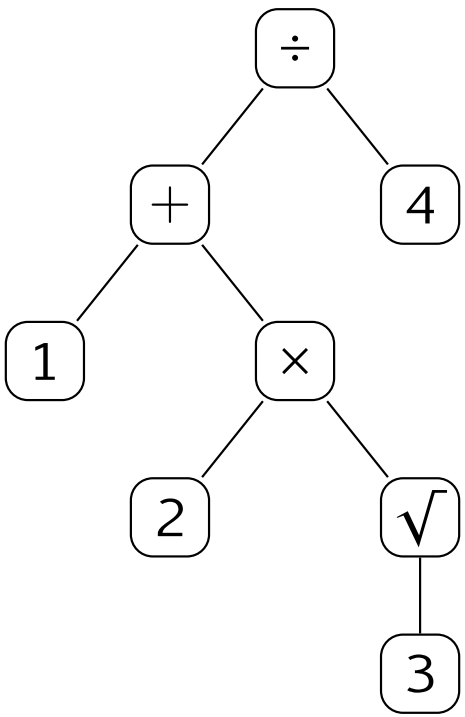
- sous forme infixe par l'expression $(1 + (2 \times (\sqrt{} 3))) \div 4$
- sous forme postfixe par l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$
- sous forme préfixe par l'expression $\div\ +\ 1\ \times\ 2\ \sqrt{}\ 3\ 4$

Représentation d'une expression mathématique

La formule $\frac{1+2\sqrt{3}}{4}$ est représentée :

- sous forme infixe par l'expression $(1 + (2 \times (\sqrt{} 3))) \div 4$
- sous forme postfixe par l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$
- sous forme préfixe par l'expression $\div\ +\ 1\ \times\ 2\ \sqrt{}\ 3\ 4$

Les formes préfixe et postfixe correspondent au parcours préfixe et postfixe de l'arbre associé à l'expression mathématique :



Évaluation d'une expression postfixée

On utilise une pile pour évaluer une expression postfixée, en suivant les règles :

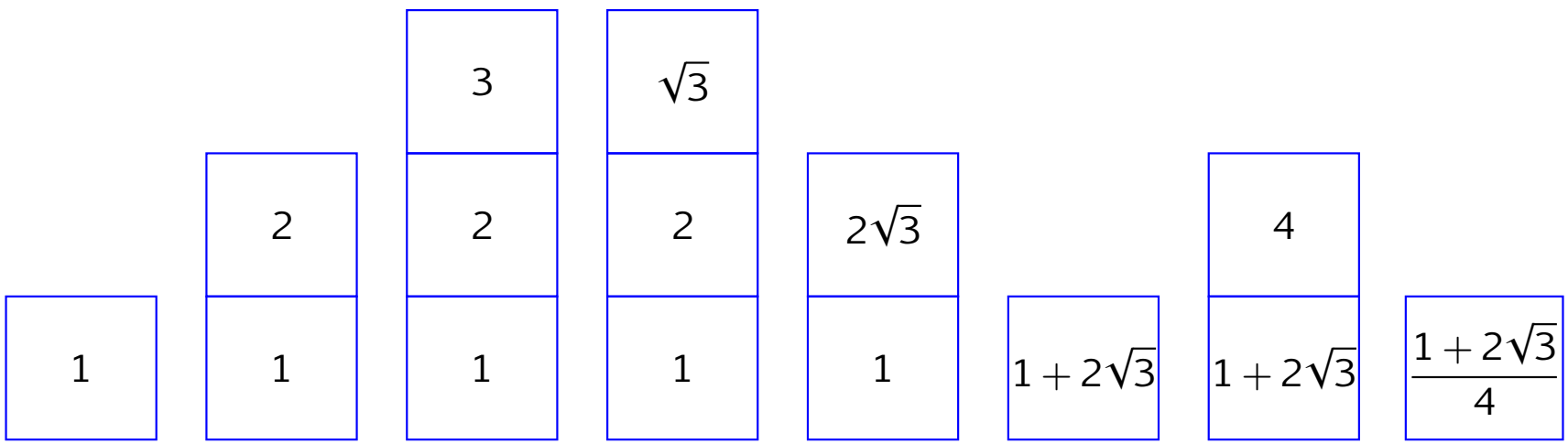
- si la tête de liste est un nombre a, on l'empile ;
- si la tête est un opérateur unaire f, on dépile le sommet a et on empile $f(a)$;
- si la tête est un opérateur binaire f, on dépile les deux éléments a et b les plus hauts, et on empile $f(a,b)$.

Évaluation d'une expression postfixée

On utilise une pile pour évaluer une expression postfixée, en suivant les règles :

- si la tête de liste est un nombre a, on l'empile ;
- si la tête est un opérateur unaire f, on dépile le sommet a et on empile $f(a)$;
- si la tête est un opérateur binaire f, on dépile les deux éléments a et b les plus hauts, et on empile $f(a,b)$.

La pile associée à l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$ va évoluer comme suit :



Mise en œuvre pratique

On définit le type suivant :

```
# type lexeme = Nombre of float
              | Op_binaire of float -> float -> float
              | Op_unaire of float -> float ;;
Type lexeme defined.
```

et on procède par filtrage sur un élément de type *lexeme list* :

```
# let evalue lst =
    let pile = new () in
    let rec aux = function
      | []                  -> pop pile
      | (Nombre a)::q       -> push a pile ; aux q
      | (Op_unaire f)::q    -> let a = pop pile in
                               push (f a) pile ; aux q
      | (Op_binaire f)::q -> let b = pop pile in let a = pop pile in
                               push (f a b) pile ; aux q
    in aux lst ;;
evalue : lexeme list -> float = <fun>
```

Détection des erreurs de syntaxe

Deux types d'erreurs peuvent se produire : dépiler la pile vide, ou la pile contient plus d'un élément à la fin du traitement.

On crée une nouvelle exception :

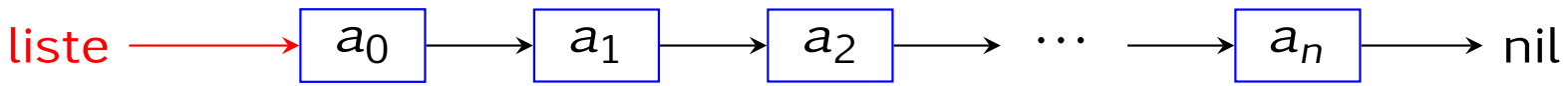
```
# exception Syntax_Error ;;
Exception Syntax_Error defined.
```

L'instruction **raise** permet de lever cette exception.

```
let evalue lst =
  let pile = new () in
  let rec aux = function
    | []                  -> let rep = pop pile in
                             try pop pile ; raise Syntax_Error
                             with Empty -> rep
    | (Nombre a)::q       -> push a pile ; aux q
    | (Op_unaire f)::q    -> let a = pop pile in
                             push (f a) pile ; aux q
    | (Op_binaire f)::q -> let b = pop pile in let a = pop pile in
                             push (f a b) pile ; aux q
  in try aux lst with Empty -> raise Syntax_Error ;;
```

Implémentation d'une pile

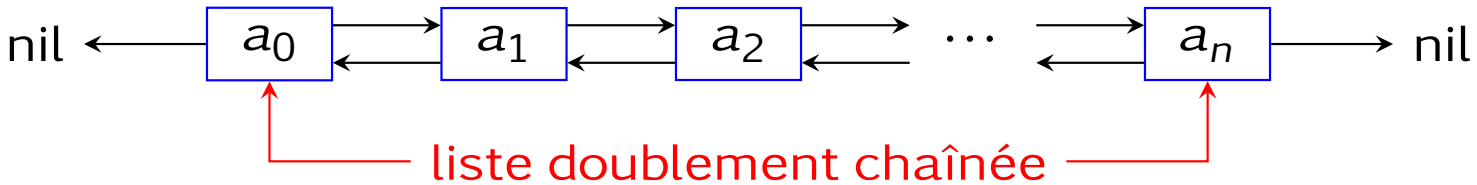
Une solution consiste à utiliser une liste mutable, la tête de la liste correspondant au sommet de la pile :




```
# type 'a pile = {mutable Liste : 'a list} ;;
Type pile defined.
# exception Empty ;;
Exception Empty defined.
# let new () = {Liste = []} ;;
new : unit -> 'a pile = <fun>
# let push x p = p.Liste <- x::p.Liste ;;
push : 'a -> 'a pile -> unit = <fun>
# let pop p = match p.Liste with
    | []   -> raise Empty
    | t::q -> (p.Liste <- q ; t) ;;
pop : 'a pile -> 'a = <fun>
```

Implémentation d'une file

Une solution consiste à utiliser une liste doublement chaînée :



qui se définit de la façon suivante :

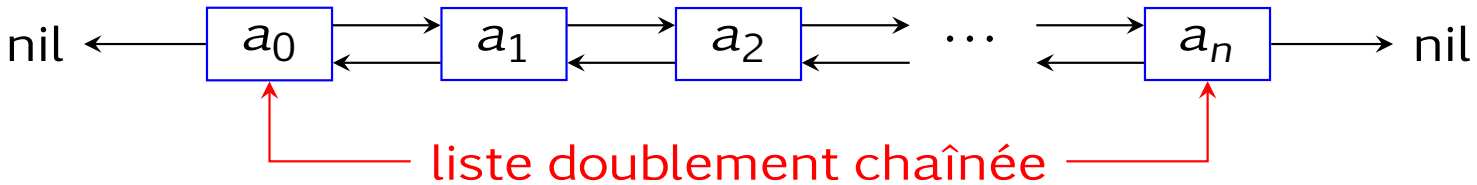
```
# type 'a cell = {Valeur : 'a ;
                  mutable Avant : 'a liste ;
                  mutable Après : 'a liste}
  and 'a liste = Nil | Cellule of 'a cell ;;
Type cell defined.
Type liste defined.
```

Une file n'est alors qu'un enregistrement mutable qui pointe vers la tête et la queue d'une liste doublement chaînée :

```
# type 'a file = {mutable Tête : 'a liste ;
                  mutable Queue : 'a liste} ;;
Type file defined.
```

Implémentation d'une file

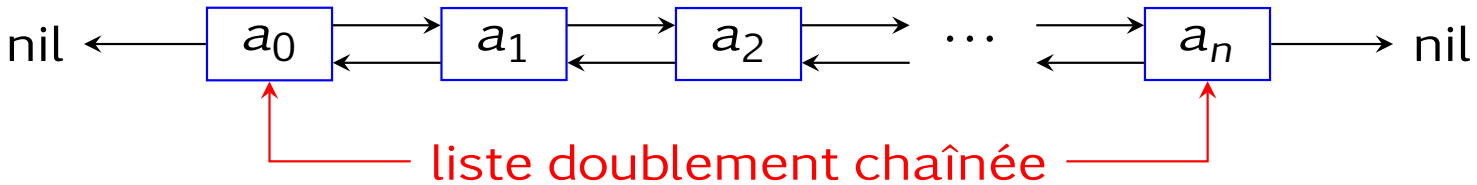
Une solution consiste à utiliser une liste doublement chaînée :




```
# exception Empty ;;
Exception Empty defined.
# let new () = {Tête = Nil; Queue = Nil} ;;
new : unit -> 'a file = <fun>
# let add x f =
    let c = Cellule {Valeur = x; Avant = f.Queue; Après = Nil} in
      match f.Queue with
        | Nil       -> (f.Tête <- c ; f.Queue <- c)
        | Cellule d -> (d.Après <- c ; f.Queue <- c) ;;
add : 'a -> 'a file -> unit = <fun>
# let take f = match f.Tête with
    | Nil -> raise Empty
    | Cellule c -> match c.Après with
        | Nil       -> (f.Tête <- Nil ; f.Queue <- Nil ; c.Valeur)
        | Cellule d -> (d.Avant <- Nil ; f.Tête <- c.Après ;
                        c.Valeur) ;;
take : 'a file -> 'a = <fun>
```

Implémentation d'une file

Une solution consiste à utiliser une liste doublement chaînée :




```
# exception Empty ;;
Exception Empty defined.
# let new () = {Tête = Nil; Queue = Nil} ;;
new : unit -> 'a file = <fun>
# let add x f =
    let c = Cellule {Valeur = x; Avant = f.Queue; Après = Nil} in
      match f.Queue with
        | Nil       -> (f.Tête <- c ; f.Queue <- c)
        | Cellule d -> (d.Après <- c ; f.Queue <- c) ;;
add : 'a -> 'a file -> unit = <fun>

# let peek f = match f.Tête with
    | Nil       -> raise Empty
    | Cellule c -> c.Valeur ;;
peek : 'a file -> 'a = <fun>
```