Récursivité

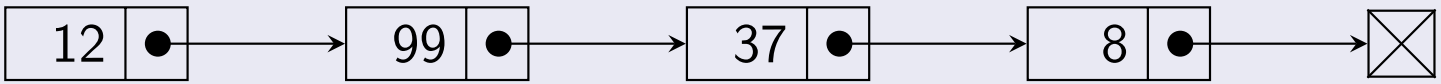
Rappels :

- `let rec f ...` pour déclarer une fonction récursive en Caml ;
- géré par la pile d'appels de fonctions ;
- permet de faire des choses pas faciles lorsqu'il y a plusieurs appels récursifs (Hanoï...)
- attention aux appels qui se chevauchent (Fibonacci...)

Listes chaînées

Rappels

- Structure :



- Dissymétrique ! L'accès est à gauche (tête de liste) !
- Immuable ! On ne peut modifier une liste, seulement créer une nouvelle liste.
- Fonctions : `List.hd q`, `List.tl q`, `x::q`, en $O(1)$.
- On fonctionnera plus souvent par filtrage.

```
let rec somme q = match q with
 | [] -> 0
 | x::p -> x + somme p
;;
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Exemple :

```
# Cons(4,Cons(3,Cons(7,Vide))) ;;
- : int liste = Cons (4, Cons (3, Cons (7, Vide)))
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Exemple :

```
# Cons(4,Cons(3,Cons(7,Vide))) ;;
- : int liste = Cons (4, Cons (3, Cons (7, Vide)))
```

```
let tete q=match q with
  | Vide -> failwith "Vide"
  | Cons (x, _) -> x
;;
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Exemple :

```
# Cons(4,Cons(3,Cons(7,Vide))) ;;
- : int liste = Cons (4, Cons (3, Cons (7, Vide)))
```

```
let tete q=match q with
  | Vide -> failwith "Vide"
  | Cons (x, _) -> x
;;
```

```
let queue q=match q with
  | Vide -> failwith "Vide"
  | Cons (_, p) -> p
;;
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Exemple :

```
# Cons(4,Cons(3,Cons(7,Vide))) ;;
- : int liste = Cons (4, Cons (3, Cons (7, Vide)))
```

```
let tete q=match q with
  | Vide -> failwith "Vide"
  | Cons (x, _) -> x
;;
```

```
let queue q=match q with
  | Vide -> failwith "Vide"
  | Cons (_, p) -> p
;;
```

```
let cons (x,q)=Cons (x, q) ;;
```

Intermède : recréer des listes nous-mêmes

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Exemple :

```
# Cons(4,Cons(3,Cons(7,Vide))) ;;
- : int liste = Cons (4, Cons (3, Cons (7, Vide)))
```

```
let tete q=match q with
  | Vide -> failwith "Vide"
  | Cons (x, _) -> x
;;
```

```
let queue q=match q with
  | Vide -> failwith "Vide"
  | Cons (_, p) -> p
;;
```

```
let cons (x,q)=Cons (x, q) ;;
```

Types :

```
tete : 'a liste -> 'a
queue : 'a liste -> 'a liste
cons : 'a * 'a liste -> 'a liste
```

Piles avec des listes

Non bornées ! Type enregistrement pour rendre mutable.

Piles avec des listes

Non bornées ! Type enregistrement pour rendre mutable.

```
type 'a pile = {mutable contenu: 'a list} ;;

let creer_pile () = {contenu=[]} ;;
let pile_vide p = p.contenu = [] ;;
let empiler p x=p.contenu <- x::p.contenu ;;

let sommet p = match p.contenu with
  | [] -> failwith "pile vide"
  | x::_ -> x
;;

let depiler p = match p.contenu with
  | [] -> failwith "pile vide"
  | x::q -> p.contenu <- q ; x
;;
```

Toutes les opérations sont en $O(1)$.

Files avec des listes

Plus complexe ! Le « bout » d'une liste n'est pas accessible.

Files avec des listes

Plus complexe ! Le « bout » d'une liste n'est pas accessible.
Idée : 2 listes au lieu d'une :

Files avec des listes

Plus complexe ! Le « bout » d'une liste n'est pas accessible.
Idée : 2 listes au lieu d'une :

Opérations faciles à écrire sauf... lorsqu'il faut défiler et que la 2e liste est vide. Dans ce cas, on retourne la 1ere liste dans la 2e.

Implémentation

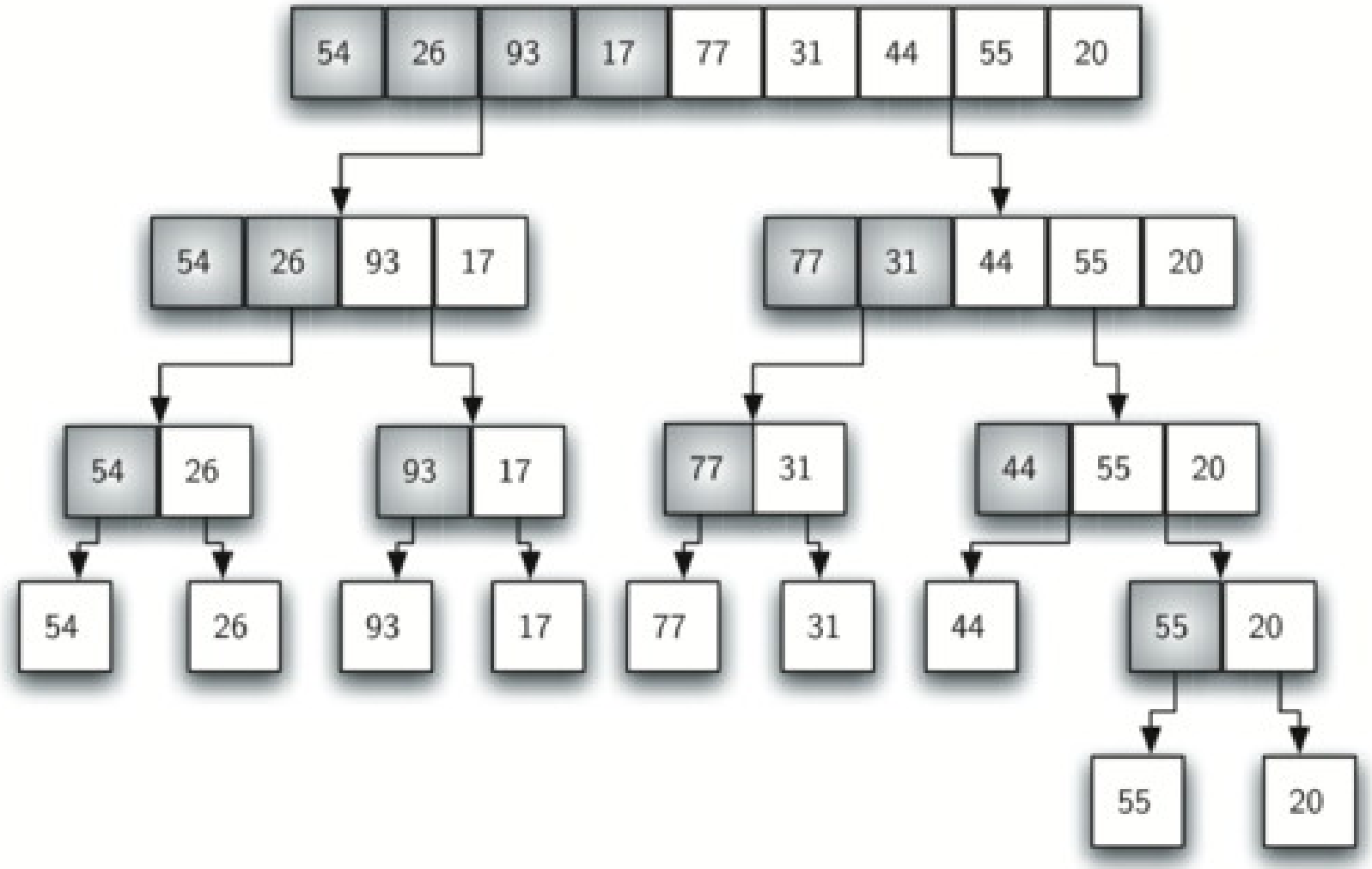
```
type 'a file =
 {mutable entree: 'a list ; mutable sortie: 'a list}
;;

let creer_file () =
  {entree = []; sortie = []}
;;

let file_vide f=
  f.entree = [] && f.sortie = []
;;

let enfiler f x=
  f.entree <- x::f.entree
;;

let rec defiler f=match f.sortie with
  | [] when f.entree = [] -> failwith "file vide"
  | [] -> f.sortie <- List.rev f.entree ; f.entree <- [] ;
          defiler f
  | x::q -> f.sortie <- q ; x
;;
```

Exemple

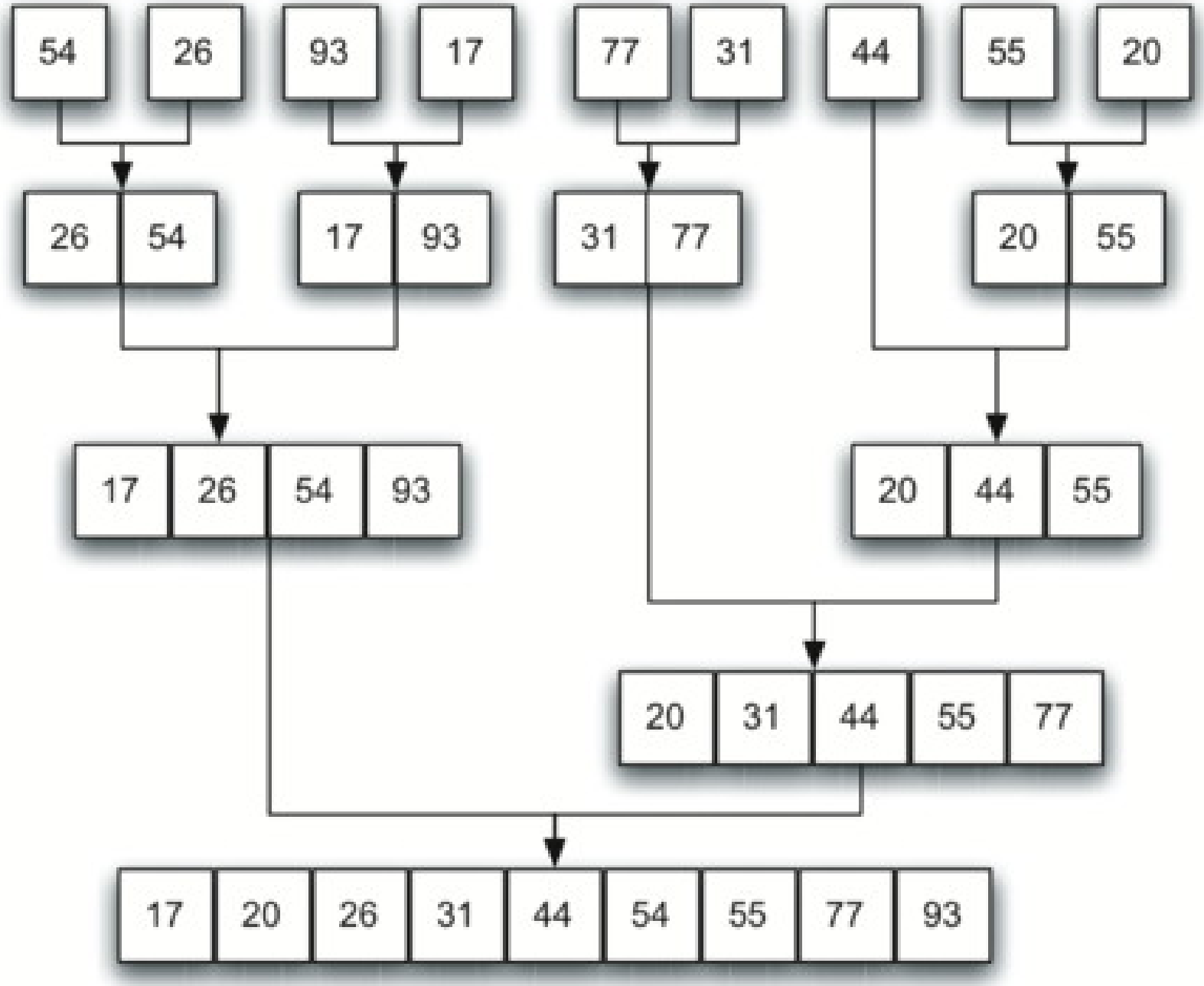
```
# let f=creer_file () ;;
val f : '_a file = {entree = []; sortie = []}
# for i=0 to 10 do enfiler f i done ;;
- : unit = ()
# f ;;
- : int file =
   {entree = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0]; sortie = []}
# defiler f ;;
- : int = 0
# f ;;
- : int file =
   {entree = []; sortie = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]}
```

Complexités des opérations : $O(1)$, *amortie* pour `defiler`.

Exemple fondamental : tri fusion



Exemple fondamental : tri fusion



Exemple fondamental : tri fusion

Principe « diviser pour régner » :

- Si la liste est de taille au plus 1, elle est triée !
- Sinon :
  - séparer la liste en deux morceaux de même taille (à 1 près) ;
  - trier récursivement les deux morceaux ;
  - reconstituer une liste triée par fusion.

Exemple fondamental : tri fusion. Fonction de « fission ».

```
let rec fission q=match q with
  | [] | [_] -> q, []
  | x::y::p -> let a,b=fission p in x::a, y::b
;;
```

Exemple fondamental : tri fusion. Fonction de « fission ».

```
let rec fission q=match q with
  | [] | [_] -> q, []
  | x::y::p -> let a,b=fission p in x::a, y::b
;;
```

```
# fission [54; 26; 93; 17; 77; 31; 44; 55; 20] ;;
- : int list * int list =
   ([54; 93; 77; 44; 20], [26; 17; 31; 55])
```

Exemple fondamental : tri fusion. Fonction de « fusion ».

```
let rec fusion p1 p2=match p1, p2 with
  | [],_ -> p2
  | _, [] -> p1
  | x::q1, y::_ when x<=y -> x::(fusion q1 p2)
  | _,x::q2 -> x::(fusion p1 q2)
;;
```

Exemple fondamental : tri fusion. Fonction de « fusion ».

```
let rec fusion p1 p2=match p1, p2 with
  | [],_ -> p2
  | _, [] -> p1
  | x::q1, y::_ when x<=y -> x::(fusion q1 p2)
  | _,x::q2 -> x::(fusion p1 q2)
;;
```

```
# fusion [20; 44; 77; 54; 93] [17; 26; 31; 55] ;;
- : int list = [17; 20; 26; 31; 44; 55; 77; 54; 93]
```

Exemple fondamental : tri fusion. Fonction finale

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in
          fusion (tri_fusion a) (tri_fusion b)
;;
```

Exemple fondamental : tri fusion. Fonction finale

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in
         fusion (tri_fusion a) (tri_fusion b)
;;
```

```
# tri_fusion [54; 26; 93; 17; 77; 31; 44; 55; 20] ;;
- : int list = [17; 20; 26; 31; 44; 54; 55; 77; 93]
```

Exemple fondamental : tri fusion. Fonction finale

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in
         fusion (tri_fusion a) (tri_fusion b)
;;
```

```
# tri_fusion [54; 26; 93; 17; 77; 31; 44; 55; 20] ;;
- : int list = [17; 20; 26; 31; 44; 54; 55; 77; 93]
```

- Très efficace !

Exemple fondamental : tri fusion. Fonction finale

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in
         fusion (tri_fusion a) (tri_fusion b)
;;
```

```
# tri_fusion [54; 26; 93; 17; 77; 31; 44; 55; 20] ;;
- : int list = [17; 20; 26; 31; 44; 54; 55; 77; 93]
```

- Très efficace !
- Complexité $O(n \log n)$.

Exemple fondamental : tri fusion. Fonction finale

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in
         fusion (tri_fusion a) (tri_fusion b)
;;
```

```
# tri_fusion [54; 26; 93; 17; 77; 31; 44; 55; 20] ;;
- : int list = [17; 20; 26; 31; 44; 54; 55; 77; 93]
```

- Très efficace !
- Complexité $O(n \log n)$.
- Existe en Caml (prend une fonction de tri en entrée) :

```
# List.sort ;;
- : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
# let f x y = x - y ;;
val f : int -> int -> int = <fun>
```