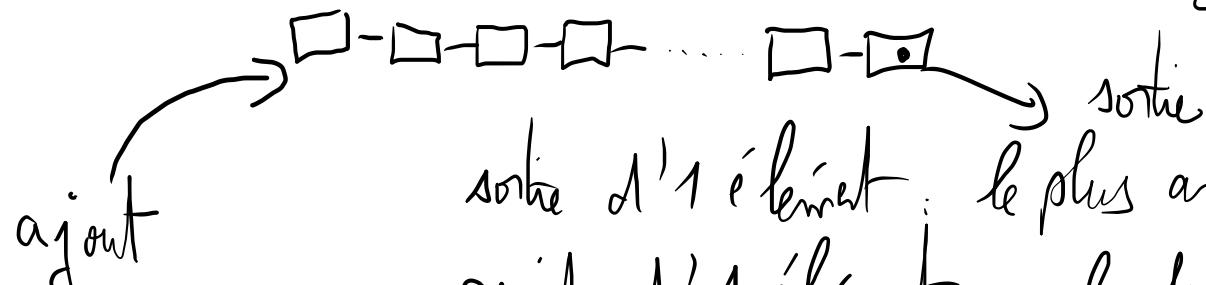


# Structures de données : files

## Paragraphe 3 : FILES

Structure dualle de celle de PILE. structure linéaire (succession de cellules reliées) FIFO : first in first out.

→ sens de la file



sorsie d'un élément : le plus ancien

ajout d'un élément à la file : de l'autre côté

image concrète : file d'attente dans un magasin

imprimante en réseau : gestion des impressions par 1 file

## Primitives de la SD. FILE.

- création d' 1 file vide FileVide()
- test de vacuité EstVide(F)
- Enfiler(e, F)
- Défiler(F) : extrait l'elt le plus ancien.

## Implémentation persistante

FileVide : unit  $\rightarrow$  file

EstVide : file  $\rightarrow$  bool

Enfiler : elt  $\times$  file  $\rightarrow$  file

Défiler : file  $\rightarrow$  elt  $\times$  file

## Implémat. impérative

"

"

Enfiler : elt  $\times$  file  $\rightarrow$  unit

(la file est modifiée)

Défiler : file  $\rightarrow$  elt

(la file est modifiée)

Prochain chapitre : comment implémenter  
- avec 2 listes en octant  
- dans un tableau affe structure de file

Module Queue est dédié aux files. : les files sont impératives.

Queue • create () : crée une file vide

add :  $'a \rightarrow 'a t \rightarrow unit$  : enfile

take :  $'a t \rightarrow 'a$  : defile

autres fcts : peek :  $'a t \rightarrow 'a$  : renvoie le 1<sup>er</sup> él<sup>t</sup> de la file,  
mais ne l'enlève pas de la file

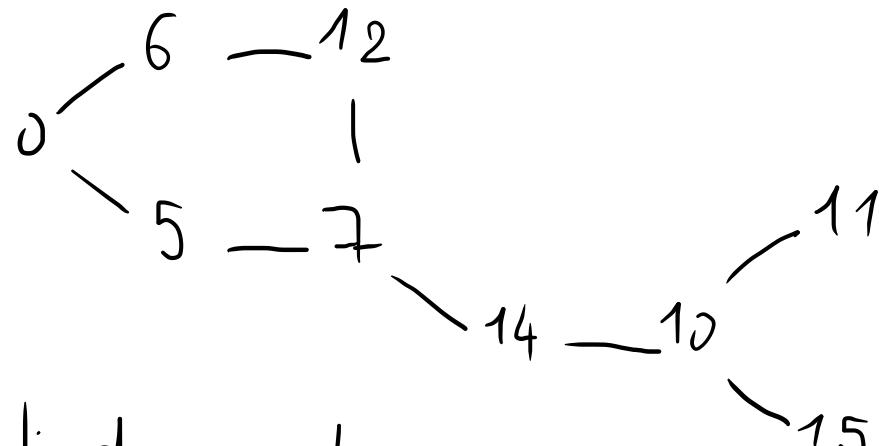
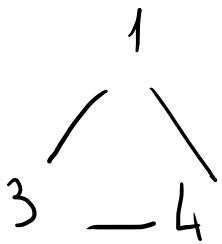
is\_empty :  $'a t \rightarrow bool$

Empty : exception renvoyée pour 1 file vide

ex :



on défile 2 fois  
on ajoute 4, 6, 10 .



Parcours en largeur à partir du sommet 0.

on traite 0 . . on traite les voisins de 0 . on traite les voisins des voisins de 0 qui n'ont pas été traités , et ainsi de suite .

on traite les sommets par ordre croissant de leur distance au sommet initial 0  
(n° d'arêtes de s à x)

exemple :  $s=0$

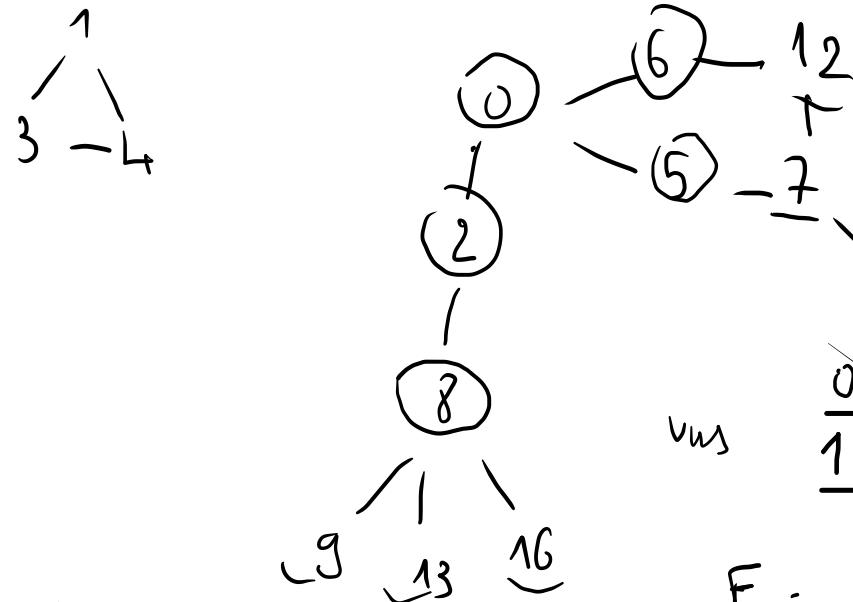
$0, 6, 5, 12, 7, 14, 10, 11, 15$

pas terminé

distance à 0	0	1	2	3	4	5
	0	6, 5	12, 7	14	10	11, 15

Il permet de traiter la composante connexe de  $s$ .  
 $s=1$  .  $1, 3, 4$  ou  $1, 4, 3$

$s=14$  :  $14, 7, 10, 5, 12, 11, 15, 6, 0$  .



en revanche  $\text{riv}(L)$ .

$\rightarrow$  parcours en largeur.

vns

0	2	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	1	1	1	1	0	0	1	0	0	0	1

Parcours à partir de 0.

C'est parti:

$$\cdot c = 0.$$

$$F: \leftarrow$$

$$L = [ ]$$

$$L = [0]$$

$$G[0] = \{2, 5, 6\}$$

$$F = \overrightarrow{6, 5, 2}$$

$$\cdot c = 2$$

$$F = \overrightarrow{6, 5}$$

$$L = [2, 0]$$

$$G[2] = \{0, 8\}$$

$$F = \overrightarrow{8, 6, 5}$$

$$\cdot c = 5$$

$$F = \overrightarrow{8, 6}$$

$$L = [5, 2, 0]$$

$$F = \overrightarrow{7, 8, 6}$$

$$F = \overrightarrow{12, 7, 8}$$

$$\cdot c = 6$$

$$F = \overrightarrow{7, 8}$$

$$L = [6, 5, 2, 0]$$

$$F = \overrightarrow{12, 7, 8}$$

$$F = \overrightarrow{16, 13, 9, 12, 7}$$

$$\cdot c = 8$$

$$F = \overrightarrow{12, 7}$$

$$L = [8, 6, 5, 2, 0]$$

$$F = \overrightarrow{15, 14, 11, 10, 9, 8, 7, 6, 5, 2, 0}$$

$$F = \overrightarrow{\phi}$$

etc

10, 14, 16, 13, 9, 12, 7, 8, 6, 5, 2, 0

on appelle 11 au degrés 11       $c = 10, F = \emptyset$        $c = 11, F = \emptyset$   
 on appelle 15 au degrés 15       $F = 15$        $c = 15, F = \emptyset$        $c = 16, F = \emptyset$

Algo impératif : let parcours\_bangimp g s =
   
 let f = Queue.create() and l = ref [] and m = Array.length g in
   
 let vus = Array.make m 0 and c = ref s in
   
 Queue.add s f; vus.(s) := 1;
   
 let rec aux l = match l with []
   
 | x :: g when vus.(x) = 0 → Queue.add x f; vus.(x) ← 1; aux g
   
 | x :: g → aux g
   
 in
   
 while not (Queue.is\_empty f) do
   
 c := Queue.take f;
   
 l := (!c) :: (!l);
   
 aux g.(!c)
   
 done;
   
 List.rev (!l);

{  
 let aux\_main  
 mise à jour d'un sommet

## Version récursive

parc\_lang g vus reste : renvoie la liste vus renversée concaténée avec les parcours en largeur sans rejeter les sommets dont les origines décrivent la liste reste

```
let rec parc_lang g vus reste =  
    match reste with  
        [] → List.rev vus
```

| t :: q when List.mem t vus → parc\_lang g vus q

| t :: q → parc\_lang g (t :: vus) (q @ g.(t))

```
let parcours_lang g s = parc_lang g [] [s];;
```

le fait de concaténer  $g.(t)$  à droite de  $q$  permet de traiter les sommets de  $q$  avant ceux de  $g.(t)$

Comparaison avec

parc\_prof g vus reste

"

"

"

" → parc\_prof g (t :: vus)  
(g.(t) @ q)

concaténe les sommets de  $g.(t)$  avant ceux de  $q \rightarrow$  on suit l'branché

## Structure de dictionnaire

(ou tableau d'associations)

des valeurs

A chaque clé, correspond une unique valeur

ex de base : tableau  $T$  de  $m$  cases. clés : entiers de  $0 à m-1$

valeurs associées à la clé  $i = T[i]$

ex. dictionnaire (au sens usuel)

des mots

valeurs : sens des mots

girafe  $\rightarrow$  mammifère vivant en Afrique ...

ex  
département

clé = numéro

valeur = [nom, préfecture, population, ...]

autre ex. graph.

les sommets étant indexés par entier de  $0 \text{ à } n-1$

~~Si~~ on enlève des sommets, on ne respecte plus les indices de  $0 \text{ à } n-1$

dictionnaire.

- clé : nom des sommets

- valeur : liste d'adjacence

répertoire

. clé : nom + prénom

. valeur : n° de tel, adresse, adresse mail...

opérations élémentaires

---

- insertion : - recherche (à partir de la clé)  
(on ajoute (clé, valeur))

- suppression

DicoVide( )

EstVide(D).

Comment implementer la SD dictionnaire ?

---

- { - liste mutable
- tableau

pas performant (recherche =  $O(n)$ )

Améliorations :

- Table de hachage (à suivre)
- arbre binaire de recherche  
(bonne complexité si équilibrés)

arbre AVL

autres binaires

# Module OCaml dédié

Map

structure persistante de dictionnaire

D'abord module Clés

module Dictionnaire = Map.Make(Clés)

Mar example

clés = ( nom, prénom )

valeurs = ( classe, origine, intégration )

# File de priorité

$U = \{ \text{dés} \}$  Nat. ordonné

$V = \{ \text{valeurs} \}$

une file de priorité = ensemble dynamique composé de dés et de valeurs  
les dés définissent la priorité d'un élément

ex : gestion des décollages d'un aéroport

(3, vol 737)

(2, vol 124)

(5, vol ABA 5) (...)

Opérations élémentaires de cette SD:

S = file de priorité

- Insérer ( $x, S$ )
- Maximum ( $S$ )
- Extraire-max ( $S$ )

: déterminer l'elt de priorité maximale  
(avec une complexité optimale)

: extraire l'elt de priorité maximale de S

Implém. impérative

Créer : unit  $\rightarrow$  fp

EstVide : fp  $\rightarrow$  bool

Insérer : de  $\times$  elt  $\times$  fp  $\rightarrow$  unit

ExtraireMax : fp  $\rightarrow$  elt (fp est modifié)

Maximum : fp  $\rightarrow$  elt (fp non modifié)

implémenter par liste chaînée : insertion :  $O(1)$

extraire\_max :  $O(n)$

La bonne implémentation d'une file de priorité consiste à utiliser

un Tas (heap)

Intérêt (page 13) En dispose d'une structure de file de priorité dans laquelle les opérations "insertion" et "extraire\_max" sont en  $O(\log N)$

Alors on peut implémenter avec cette structure de f.p un algorithme de tri de complexité en  $O(N \log N)$ .

Preuve: E l'ensemble à trier (selon des clés). En toute  $\sum_{k=1}^{N-1}$  succ. les élts de E ds une f.p, vide au départ. Coût dell'insertion du k+1<sup>e</sup> élft :  $O(\log k)$ .  $\sum_{k=1}^{N-1} O(\log k) = O(N \log N)$

A la fin, la file contient tous les élts de E

2<sup>e</sup> phase : on extrait le max de la f.p. N fois et on le stocke  
dans une liste (ou un tableau)

à la fin, on aura trié E

compt :  $O(\log N) + O(\log(N-1)) + \dots$

$$\sum = O(\log h) = O(N \log N).$$

Au total, la complexité est  $O(N \log N)$ .

creat-fp : ' $a \rightarrow$ ' a fp      renvoie un fp initialisé à 1 valeur

insérer : ' $a \rightarrow$ ' a fp  $\rightarrow$  unit

extraction : ' $a \rightarrow$ ' a fp  $\rightarrow$  ' $a$ '

fbt cachees

$v$  est un tableau d'entiers

let tri  $v =$

let  $n = \text{Array.length } v$  in let  $w = \text{Array.make } n \ 0$  in

let  $f = \text{creat-fp } 0$  in

for  $i = 0$  to  $n-1$  do insérer  $v.(i)$   $f$  done ;

for  $i = 1$  to  $n$  do  $w.(n-i) \leftarrow \text{extraction } f$  done ;

$w;$

Algorithme

. Principe:

G graphe

tableau  $vus$  initialisé à 0, tout sommet traité passe à 1

liste  $L$ : construction du parcours en largeur

file  $F$ : on enfile  $s$  (sommet de départ)

Tant que  $F$  est non vide,

$c \leftarrow \text{defile } F$ , on ajoute  $c$  à  $L$  ( $L \leftarrow c :: L$ )

on regarde les voisins de  $c$ :  $G[c]$

$\forall \alpha \in G[c]$ , si  $vus[\alpha] = 0$ , {enfiler  $\alpha, F$  }  
 $vus[\alpha] = 1$

on retourne  $rev(L)$ .