Structures de données

1 Structures de données abstraites

1.1 Ensemble dynamique

Contrairement à ce qui se passe en Mathématiques, les variables utilisées dans un programme informatique peuvent appartenir à un ensemble dynamique, c'est-à-dire dont le contenu évolue au cours du programme, sous l'effet d'insertions, de suppressions ou de modifications de l'ordre des éléments ou de certains champs.

1.2 Structures de données abstraites

Un type de données abstrait est une entité permettant de regrouper des objets présentant des caractéristiques communes. Dans un langage informatique, les données appartenant à un même type sont représentées en mémoire sous un même format et les fonctions élémentaires agissant sur ces données disposent d'une syntaxe particulière.
On peut citer par exemple les types simples entier, flottant, booléen, caractère, puis les types composés liste, tableau, ensemble, chaîne de caractères, arbre. Selon les besoins, on peut définir des types plus précis, parfois à l'aide de types généraux, comme par exemple un type `complexe` constitué de deux champs de flottants, un type `élève` contenant les champs `nom`, `classe`, `statut` (interne ou externe), ou un type `polynome` représenté par une liste de flottants.

Définition : Une structure de données abstraite (SDA) est un type de données sur lesquelles on définit des opérations de base (appelées primitives de la SDA). La description de ces opérations est appelée leur sémantique.
La liste des objets et des opérations de base est la signature de la SDA. Leur description est la spécification de la SDA.

Outre la création et le test de vacuité, les opérations classiques sur les SDA sont la recherche, l'insertion, la suppression, la détermination du minimum, du maximum, du successeur, du prédécesseur.
L'intérêt d'une SDA est que le programmeur peut l'utiliser comme une boite noire, c'est-à-dire sans avoir besoin de connaître la façon dont sont représentées les données ni comment sont implémentées les opérations de base.
En Caml, les chaînes de caractères et les listes sont des exemples de SDA. On ignore comment elles sont implantées en machine. L'objectif recherché est de pouvoir les utiliser efficacement.

Exemples de structures de données prédéfinies en OCaml :

- Liste (type `list`). Primitives : tête, queue, ajout en tête, concaténation, longueur.
- Tableau (type `array`). Primitives : accès à un élément d'indice donné, modification de cet élément.
- type Chaîne de caractères (type `string`). Primitives : accès à un caractère, concaténation, extraction d'une sous-chaîne, longueur.

Les structures de données classiques apparaissant en informatique sont :
tableau, liste (chaînée), liste doublement chaînée, pile, file, dictionnaire, file de priorité, tas, arbre, graphe...
La présence de certaines structures de données dépend du langage. En Python, la structure de liste n'est pas du tout la même qu'en Caml (il s'agit en fait de tableau redimensionnable). Les langages orientés objet disposent d'une gamme plus large de structures de données déjà implantées.

Elaborer un programme informatique consiste à définir le problème, spécifier le cahier des charges, proposer un algorithme et choisir la bonne structure de données pour implémenter efficacement l'algorithme.

1.3 Réalisation d'une structure de données

On peut chercher à réaliser explicitement une structure de données abstraite, c'est-à-dire à l'implémenter dans un langage informatique.
Une telle structure de données sera souvent réalisée à l'aide d'autres structures plus basiques disponibles dans le langage, par exemple des listes, des enregistrements ou des tableaux.
L'implémentation des structures de données est un objectif important de l'informatique.
Toutefois, on a surtout besoin en pratique d'utiliser une structure de données adaptée au problème. On peut faire un parallèle avec les Mathématiques où les ensembles de nombres se construisent axiomatiquement les uns à partir des autres ($\mathbb{N} \to \mathbb{Z} \to \mathbb{Q} \to \mathbb{R} \to \mathbb{C}$). Pour résoudre un problème mathématique, on ne revient pas nécessairement à la construction de ces ensembles, mais on les utilise en tant que tel.

1.4 Structures linéaires

Une structure de données linéaire est une structure que l'on peut représenter sous la forme d'une suite de cellules, où chaque cellule est reliée à la cellule suivante (successeur) ou éventuellement à la précédente (prédécesseur). On peut représenter l'objet informatique obtenu comme une chaîne de maillons élémentaires reliés entre eux par des indices ou des pointeurs.
Par exemple, une liste, une pile ou une file sont des structures linéaires.
Dans une liste chaînée, chaque cellule contient deux informations : une valeur et un pointeur (ou adresse mémoire) vers la cellule suivante. La dernière cellule pointe vers un symbole appelé fin de liste.
Représentation graphique.
Pour parcourir la liste, on a juste besoin de connaître pour chaque cellule l'adresse de la cellule suivante.
Pour accéder à une cellule, on doit passer par les cellules précédentes (accès séquentiel).

Dans une liste doublement chaînée, chaque cellule comporte trois informations, une valeur et deux pointeurs, un vers la cellule précédente et un vers la cellule suivante. La première cellule pointe en avant vers un symbole de fin de liste et la dernière cellule pointe en arrière vers ce même symbole.

Une liste circulaire se comporte comme une liste chaînée où chaque cellule pointe vers la suivante, la dernière pointant vers la première. Il n'y a pas de symbole de fin de liste.

La structure d'arbre est un exemple de structure non linéaire. Chaque cellule peut pointer sur plusieurs cellules, par exemple ses 2 fils, ou la liste de ses fils, ou encore un fils et la liste des frères, l'objet prenant ainsi une forme arborescente.

1.5 Structures persistantes et impératives

Une structure de données est dite **impérative** (ou modifiable ou mutable) lorsque les opérations élémentaires agissant sur ses objets les modifient en place (par modification du contenu de certaines cases mémoires). Les exemples de base rencontrés en Caml sont les références et les tableaux.
Quand un programme modifie la mémoire, on dit qu'il y a un effet de bord.
La programmation impérative, basée sur des modifications continuelles de l'état de la mémoire, utilise naturellement et intuitivement des structures impératives.
Par exemple, `let modifie t i x = t.(i) <- x` modifie le tableau passé en argument.

Une structure de données est dite **persistante** (ou immuable) lorsqu'on ne peut pas la modifier en place sans la recréer complètement. Une telle structure est immuable car les fonctions s'appliquant à un objet de la structure ne le modifient pas mais renvoient un nouvel objet du même type.
L'exemple classique en Caml est la liste : on ne peut pas modifier en place une liste. Pour ajouter un élément à une liste, la fonction ajout (`let ajout a l = a::l`) renvoie une nouvelle liste.
Les chaînes de caractères (type `string`) sont maintenant implantées de façon persistante.
Les arbres en constituent un autre exemple, qui sera vu en fin d'année.

Les langages fonctionnels utilisent de préférence des structures persistantes, bien adaptées aux évaluations de fonctions et à la programmation récursive.
L'utilisation d'une structure persistante améliore souvent la clarté du code. La preuve de la correction d'un algorithme s'appuyant sur une structure persistante est facilitée car on se rapproche d'une preuve mathématique, les objets utilisés n'étant pas modifiés.

Par exemple, la fonction calculant le miroir d'une liste se prouve facilement en prouvant par récurrence sur la longueur de `l` que dans le code ci-dessous, `miroir_aux l m` calcule le miroir de `l` concaténé avec `m`.

```
let miroir =
 let rec miroir_aux l m = match l with
  | [] -> m
  | t::q -> miroir_aux q (t::m)
 in miroir_aux l [] ;;
```

Pour les structures de données immuables, le partage des données permet d'éviter les recopies inutiles. Créons une liste `l1` : `let l1 = [2;3;4;5;6;7]` . La création de `l2` par `let l2 = 1 :: l1` consiste seulement à créer une cellule de valeur 1 et à la raccrocher à la cellule de tête de `l1`, ce qui est très économique et évite de recopier le contenu de `l1` dans `l2`.
On peut facilement basculer de l'une à l'autre : une liste est une structure persistante, mais une référence de liste est impérative, ainsi qu'un enregistrement admettant un champ mutable de type liste.

```
let l = ref [1;2;3;4] ;;
l := List.tl !l ;;
!l ;;
type listmut = {mutable contenu : int list} ;;
let m = {contenu = [1;3;5;7]} ;;
m.contenu <- List.tl(m.contenu) ;;
m ;;
```

2 Piles

2.1 Définition

Définition. Une pile est une structure de données linéaire, où l'insertion et la suppression se font toujours du même côté. Une telle structure est appelée **LIFO** (last in first out) car le dernier élément inséré est le premier qui sera extrait de la pile.

L'image d'une pile est une pile d'assiettes dans un restaurant.

Les opérations principales de la structure de pile sont "Empiler" (push) et "Depiler" (pop).

- Empiler (e, P) : insère l'élément e au sommet de la pile P.
- Depiler (P) : retourne le sommet de la pile P et le supprime de la pile.
- PileVide () : renvoie une pile vide.
- EstVide (P) : teste si la pile est vide.

Les spécifications algébriques (dans le style fonctionnel) de ces opérations sont :

Implémentation persistante

- PileVide : unit $\to$ pile.
- EstVide : pile $\to$ bool.
- Empiler : elt $\times$ pile $\to$ pile.
- Depiler : pile $\to$ elt $\times$ pile.

Alternative possible : Depiler : pile $\to$ pile : renvoie la pile privée du sommet.
Sommet : pile $\to$ elt : renvoie le sommet de la pile (sans dépiler).

Condition

Depiler(P) est valide si et seulement si EstVide (P) = false.

Axiomes

- Depiler (Empiler (e, P)) = (e, P).
- EstVide (PileVide ()) = true.
- EstVide (Empiler (e, P)) = false.

Ces propriétés caractérisent la structure de pile.

Proposition : Soit P une pile. Il existe un unique $n \in \mathbb{N}$ et un unique n-uplet $(a_1, \ldots, a_n)$ tels que $P = \text{Empiler}\,(a_n, \text{Empiler}\,(a_{n-1}, \ldots, \text{Empiler}\,(a_1, \text{PileVide}())\ldots))$.

preuve. Par condition nécessaire, n est la taille de la pile. On vérifie ensuite par récurrence que le contenu de la pile est $[a_n; \ldots; a_1]$. •

Implémentation impérative

- PileVide : unit → pile.
- EstVide : pile → bool.
- Empiler : elt × pile → unit : la pile est modifiée en place.
- Depiler : pile → elt : la pile est modifiée et on renvoie son sommet.

Codage en Caml : une implémentation simple peut se faire au moyen d'une référence de liste.

2.2 Utilisation

• Un langage récursif utilise une pile pour stocker ses appels récursifs. Il les empile, puis les dépile au moment de l'évaluation.
• Dans un éditeur de texte évolué, les caractères saisis au clavier et certaines actions de l'utilisateur sont stockés dans plusieurs piles (do/undo, mémorisation des copier/coller ...)
• L'historique des pages visitées par un navigateur internet a une structure de pile.
• Certains parcours de graphes ou de listes utilisent des piles.
• Les algorithmes de backtracking utilisent des piles (trajet dans un labyrinthe, résolution d'un sudoku, problèmes des huit dames de Gauss ou du cavalier d'Euler sur un échiquier).

2.3 Module OCaml Stack

Le module `Stack` s'ouvre par la commande `open Stack`. Il est implanté de façon impérative (les piles sont modifiées en place). Le type `'a t` représente une pile d'éléments du type `'a`.

Les fonctions principales du module `Stack` sont :
`create : unit -> 'a t` : crée une pile vide.
`pop : 'a t -> 'a` : dépile en renvoyant le sommet de la pile.
`push : 'a -> 'a t -> unit` : empile.
`is_empty : 'a t -> bool` : teste si une pile est vide.
L'exception `Empty` est renvoyée pour une pile vide.

On dispose en plus des fonctions suivantes :
`top : 'a t -> 'a` : renvoie le sommet de la pile sans dépiler.
`length : 'a t -> int` : renvoie la longueur de la pile.
`clear : 'a t -> unit` : vide la pile.
`iter : ('a -> unit) -> 'a t -> unit` : applique une fonction aux éléments d'une pile.

Exercice.
- Echanger les 2 éléments situés au sommet d'une pile.
- Lister tous les éléments d'une pile (la pile sera vide à la fin).

2.4 Tri insertion en place

• La fonction récursive `insertion : 'a -> 'a t -> unit` insère un élément x dans une pile dont les éléments sont triés : Si x est inférieur au sommet y, on l'empile, sinon on dépile y, on insère x récursivement dans la pile et on empile y.
• La fonction `tri : 'a array -> 'a array` insère les éléments du tableau dans une pile initialement vide puis remplit le tableau initial en dépilant entièrement la pile. Le tableau initial est alors trié en place.

```
open Stack ;;
let rec insertion x p =
  if Stack.is_empty p then Stack.push x p
  else let y = pop p in
   if x < y then (Stack.push y p; Stack.push x p)
               else (insertion x p; Stack.push y p)
;;
let tri t =
 let n = Array.length t and p = create () in
 for k = 0 to n-1 do insertion t.(k) p done;
 for k = 0 to n-1 do t.(k) <- pop p done;
 t ;;
```

La complexité est quadratique.

2.5 Evaluation d'une expression arithmétique postfixée

2.5.1 Expressions arithmétiques. Représentation infixe, préfixe, postfixe

Une expression arithmétique est composée de constantes et de variables, d'opérateurs unaires (fonctions à un argument) et d'opérateurs binaires (fonctions à deux arguments).
Une expression arithmétique peut être représentée de plusieurs manières : infixe, postfixe, préfixe.

En notation infixe, les opérateurs binaires sont placés entre leurs arguments, par exemple $a + b$.
En présence de plusieurs opérateurs binaires, on doit utiliser des parenthèses pour définir l'ordre dans lequel les opérations doivent être effectuées (par exemple $(a + b) \times (c + d)$).
- Si *op* est binaire, on écrira récursivement `infixe(exp1) op infixe (exp2)`
- Si *op* est unaire, on écrira récursivement `(op infixe(exp1))`

Exemples : $((\sin x) + (\cos y))$, $(((x + y) \times (y - z)) + (u \times (-v)))$
La suppression de certaines parenthèses est possible grâce à la priorité de certaines opérations. Par exemple, en arithmétique, la multiplication est prioritaire sur l'addition.
Exemples : $\sin(x) + \cos(y)$, $(x + y) \times (y - z) + u \times (-v)$.

En notation postfixe, tous les opérateurs sont placés après leurs arguments. Récursivement, cela s'écrit :
- Si *op* est binaire, on représente *exp1 op exp2* par `postfixe(exp1) postfixe(exp2) op`
- Si *op* est unaire, on représente *op (exp1)* par `postfixe(exp1) op`

Les deux formules précédentes s'écrivent en postfixée : x sin y cos $+$, $x\ y\ +\ y\ z\ -\ \times\ u\ v\ \times +$

En notation préfixe, tous les opérateurs sont placés après leurs arguments :
exp1 op exp2 s'écrit `op prefixe(exp1) prefixe(exp2)`
Ainsi la 2ème formule s'écrit : $+\ \times\ +\ x\ y\ -\ y\ z\ \times\ u -\ v$
Il n'y a pas besoin de parenthèses en notation postfixe ou préfixe.

2.5.2 Evaluation d'une expression postfixée

Quand on attribue des valeurs aux variables, on peut évaluer une formule postfixée à l'aide de l'algorithme suivant :

- On initialise une pile vide.
- On lit la formule de gauche à droite. Tant qu'il reste des caractères à lire, on note c le caractère courant.
  - Si c est une valeur, on l'empile.
  - Si c est un opérateur unaire : soit x le sommet de la pile, on dépile x et on empile $c(x)$.
  - Si c est un opérateur binaire, on dépile deux fois la pile, on obtient successivement z et y et on empile $c(y, z)$.
- A la fin de la lecture, si la pile contient un seul élément, c'est l'évaluation cherchée, sinon, la formule est incorrecte.

Cet algorithme se prouve par récurrence sur la longueur de la formule postfixée, en distinguant les cas $f\ E_1$ et $op\ E_1\ E_2$. Sa complexité est linéaire par rapport à la taille de la formule.
Exemple de fonctionnement : on part de l'expression infixe $(3 + 5) \times (5 - 1) + 2 \times (-4)$. Elle s'écrit en notation postfixe : $3\ 5\ +\ 5\ 1\ -\ \times\ 2\ 4\ \ -\ \times +$
L'évolution de la pile au cours de l'évaluation est : $[3]\ [3;5]\ [8]\ [8;5]\ [8;5;1]\ [8;4]\ [32]\ [32;2]\ [32;2;4]\ [32;2;-4]\ [32;-8]\ [24]$

```
type 'a terme = V of 'a | Un of ('a -> 'a) | Bin of ('a -> 'a -> 'a) ;;

let evaluation l =
 let form = ref l and pile = Stack.create () in
 while !form <> [] do
  begin  match List.hd (!form) with
    | V x -> Stack.push x pile
    | Un f -> let y = Stack.pop pile in Stack.push (f y) pile
    | Bin g -> let z = Stack.pop pile in let y = Stack.pop pile in Stack.push (g y z) pile
  end;
  form:= List.tl(!form)
 done;
 let r = Stack.pop pile in
 if Stack.is_empty pile then r else failwith "formule incorrecte"
;;
(* Exemple *)
let plus x y = x+y;; let moins x y = x-y;; let mult x y = x*y;; let opp x = -x;;
```

```
let e = [V 3;V 5;Bin plus;V 5;V 1;Bin moins;Bin mult;V 2;V 4;U opp;Bin mult;Bin plus];;
evaluation e ;;  (* réponse : 24 *)
let f = [V 3; V 5] ;;
evaluation f ;;  (* réponse : Exception non rattrapée: Failure "formule incorrecte" *)
let g = [V 3; V 5; Bin plus; Bin plus] ;;
evaluation g ;;  (* réponse : Exception: Stack.Empty *)
```

2.6 Expressions bien parenthésées

On étudie des mots formés uniquement de parenthèses ouvrantes et fermantes. Un mot est dit bien parenthésé lorsqu'à chaque parenthèse ouvrante correspond exactement une parenthèse fermante située plus loin. Par exemple, `(()())` et `()(())` sont bien parenthésés alors que `())` et `())(` ne le sont pas.
Plus précisément, l'ensemble des mots bien parenthésés est caractérisé par les règles suivantes :
- le mot vide est bien parenthésé.
- si m_1 et m_2 sont bien parenthésés, alors (m_1) et m_1m_2 sont bien parenthésés.

On veut écrire une fonction testant si un mot est bien parenthésé et calculant pour chaque parenthèse ouvrante la parenthèse fermante qui lui est associée. Par exemple, pour `(())()`, on veut renvoyer `[(0,1);(2,5);(3,4)]`.
A l'aide d'une pile, on parcourt le mot en empilant les parenthèses ouvrantes et en dépilant à chaque fois qu'on lit une parenthèse fermante. Le mot est bien parenthésé si et seulement si la pile est vide lorsqu'on atteint la fin du mot et qu'on n'a pas cherché à dépiler une pile vide avant.
La fonction `est_bien_parenthesee` retourne `true` si le mot est bien parenthésé et `false` sinon.

```
let rec list_of_string s = match String.length s with
    0 -> []
  | n -> s.[0] :: (list_of_string (String.sub s 1 (n-1)))
;; (* la fonction renvoie la liste des caractères d'une chaine *)
let est_bien_parenthesee s =
  let rec aux p = function
    [] -> Stack.is_empty p
    | '(' :: q ->  Stack.push '(' p; aux p q
    | ')' :: q -> let a = Stack.pop p in aux p q
 in let p = Stack.create () in
 try aux p (list_of_string s)
 with _ -> false ;;
```

Pour connaitre les associations de parenthèses, on empile la position de chaque parenthèse ouvrante et on dépile à chaque fois qu'on lit une parenthèse fermante en stockant le couple d'indices (ouvrante, fermante) obtenu.
La fonction `parenthesage` retourne la liste des associations de parenthèses (ouvrante,fermante) et renvoie une exception si l'expression n'est pas bien parenthésée.

```
let parenthesage s =
  let rec aux p res i = function
    [] -> if Stack.is_empty p then res else failwith "erreur"
    | '(' :: q ->  Stack.push i p; aux p res (i+1) q
    | ')' :: q -> let j = Stack.pop p in aux p ((j,i)::res) (i+1) q
  in let p = Stack.create () in
  try aux p [] 0 (list_of_string s)
  with _ -> failwith "erreur" ;;
```

2.7 Parcours en profondeur d'un graphe

Un **graphe orienté** est la donnée d'un couple (S, A) où S est un ensemble fini (les sommets) et A une partie de $S \times S$ (les arcs).
Un graphe sera représenté ici par le tableau de ses listes d'adjacence, c'est-à-dire que les sommets sont numérotés de 0 à $n-1$ et $G.(i)$ est la liste des successeurs du sommet i. Le typage sera `int list array`.
Un **parcours** de graphe est une énumération injective (suivie éventuellement d'un traitement) des sommets à partir d'un sommet donné (l'origine).

Le parcours en profondeur (ou DFS : depth-first search) d'un graphe à partir d'un sommet s consiste à visiter tous les sommets accessibles à partir de s en suivant un chemin partant de s aussi loin que possible en visitant les sommets au fur et à mesure qu'on les découvre. Arrivé à une impasse, on remonte jusqu'à la première bifurcation conduisant à un sommet non traité et on poursuit avec cette nouvelle branche. On récupère ainsi tous les sommets accessibles à partir de s.

L'algorithme est le suivant :
- on initialise une liste `vus` de sommets visités à `[]` et une pile `reste` de sommets à traiter à `[s]`.
- tant que la pile des sommets à traiter n'est pas vide :
  - on dépile un sommet.
  - on l'ajoute à la liste des sommets visités.
  - on ajoute ses voisins non encore visités à la pile des sommets à traiter.

Pour l'élégance du code, on adopte une programmation récursive dans laquelle la pile est représentée par une liste.

```
let rec parc_prof g vus reste = match reste with
  | [] -> List.rev vus
  | t::q when List.mem t vus -> parc_prof g vus q
  | t::q -> parc_prof g (t::vus) (g.(t) @ q)
(* les voisins du sommet courant t sont empilés pour être traités avant les sommets de q *)
;;
let parcours_prof g s = parc_prof g [] [s]
;;
```

`parc_prof g vus reste` renvoie la concaténation de la liste `vus` renversée et des parcours en profondeur successifs sans répétition dont les origines sont les éléments de la liste `reste`.

Exemple :
Pour simplifier, le graphe est supposé symétrique (si (a, b) est une arête, (b, a) également).

```
let graphe = [|[2;5;6];[3;4];[0;8];[1;4];[1;3];[0;7];[0;12];[5;12;14];[2;9;13;16]; [8];[11;14;15];
[10];[6;7];[8];[7;10];[10];[8]|];;
parcours_prof graphe 0 ;;  on obtient [0; 2; 8; 9; 13; 16; 5; 7; 12; 6; 14; 10; 11; 15]
parcours_prof graphe 1 ;;  on obtient [1; 3; 4]
parcours_prof graphe 7 ;;  on obtient [7; 5; 0; 2; 8; 9; 13; 16; 6; 12; 14; 10; 11; 15]
```

La programmation impérative consiste à faire évoluer une pile tant qu'elle n'est pas vide en mettant à jour un tableau `vus` au fur et à mesure qu'un sommet est visité. La fonction (récursive) `aux` sert à empiler les sommets non vus d'une liste.

```
let parcours_prof_imp g a =
  let p = Stack.create () and l = ref [] and n = Array.length g in
  let vus = Array.make n 0 in
  Stack.push a p;
  let rec aux l = match l with
    [] -> ()
    | x::q when vus.(x)=0 -> Stack.push x p; aux q
    | x::q -> aux q
```

```
  in
  while not (Stack.is_empty p) do
    let c = Stack.pop p in
    if vus.(c) = 0 then
    begin
    l := c :: !l; aux(List.rev g.(c)); vus.(c) <- 1
    end
  done;
  List.rev !l
;;
```

3 Files

3.1 Définition

Une file est une structure de données linéaire où l'insertion se fait d'un côté de la file et la suppression de l'autre côté.
Une telle structure est appelée **FIFO** (first in first out).
L'image concrète d'une file est une file d'attente (à un guichet, dans un magasin).
Dans un réseau informatique, c'est une file d'attente qui gère les documents envoyés à une imprimante.

Les opérations fondamentales de la structure de file sont "Enfiler" et "Defiler".

- Enfiler (e, F) : ajoute l'élément e dans la file F.
- Defiler (F) : extrait le premier élément de la file et le renvoie.
- FileVide () : crée une file vide.
- EstVide (F) : teste si la file est vide.

Les spécifications algébriques (dans le style fonctionnel) de ces opérations sont :

Implémentation persistante

- FileVide : unit $\rightarrow$ file.
- EstVide : file $\rightarrow$ bool.
- Enfiler : elt $\times$ file $\rightarrow$ file.
- Defiler : file $\rightarrow$ elt $\times$ file.

Alternative possible : Defiler : file $\rightarrow$ file.
Premier : file $\rightarrow$ elt.

Condition

Defiler (F) valide si et seulement si EstVide (F) = false.

Axiomes

- EstVide (FileVide ()) = true.
- EstVide (Enfiler (e, F)) = false.
- Defiler (Enfiler (e, FileVide ()) = (e, FileVide ()).
- Si EstVide (F) = false, alors Defiler (Enfiler (e, F)) = (u, Enfiler (e, G)) où (u, G) = Defiler (F).

Ces propriétés caractérisent la structure de file.

Proposition : Soit F une file. Il existe un unique $n \in \mathbb{N}$ et un unique n-uplet $(a_1, \ldots, a_n)$ tels que $F = \text{Enfiler}\,(a_n, \text{Enfiler}\,(a_{n-1}, \ldots, \text{Enfiler}\,(a_1, \text{FileVide}\,())\ldots))$.

preuve. n est la taille de F, déterminée de façon unique. La preuve se fait par récurrence sur n. •

Implémentation impérative

- FileVide : unit $\rightarrow$ file.
- EstVide : file $\rightarrow$ bool.
- Enfiler : elt $\times$ file $\rightarrow$ unit.
- Defiler : file $\rightarrow$ elt.

En Caml, une file peut s'implémenter de façon persistante au moyen de deux listes et de façon impérative au moyen d'un tableau (voir chapitre suivant).

3.2 Module OCaml Queue

Le module `Queue` s'ouvre par la commande `open Queue`. Il est implanté de façon impérative. Le type `'a t` représente une file d'éléments du type `'a`.

Les fonctions principales du module `Queue` sont :
`create : unit -> 'a t` : crée une file vide.
`add : 'a -> 'a t -> unit` : ajoute un élément au bout de la file.
`take : 'a t -> 'a` : renvoie en la supprimant la tête de la file.
`peek : 'a t -> 'a` : renvoie la tête de la file.
`is_empty : 'a t -> bool` : teste si une file est vide.
L'exception `Empty` est renvoyée pour une file vide.

On dispose en plus des fonctions suivantes :
`length : 'a t -> int` : renvoie la longueur de la file.
`clear : 'a t -> unit` : vide la file.
`iter : ('a -> unit) -> 'a t -> unit` applique une fonction aux éléments d'une file.

Remarque. Si les deux modules `Stack` et `Queue` sont ouverts, on précisera devant le nom des fonctions à quel module il se rattache. Par exemple `Stack.create()` crée une pile, alors que `Queue.create()` crée une file.

3.3 Parcours en largeur d'un graphe

Le parcours en largeur d'un graphe à partir d'un sommet s consiste à renvoyer la liste des sommets accessibles à partir de s, en commençant par ceux situés à distance 1 de s puis ceux à distance 2 de s, et ainsi de suite.

Principe de l'algorithme.

On stocke dans une liste L les éléments qui figureront dans le parcours en largeur et on repère dans un tableau `vus` les éléments déjà découverts. Le parcours est réalisé en gérant une file F constituée au départ du sommet s. Tant que F est non vide, sa tête est le nouvel élément courant. On l'extrait, puis on ajoute à F ses voisins qui n'ont pas été déjà traités et on met à jour L et `vus`.

Ecriture de l'algorithme en pseudo-langage.

$F \leftarrow$ FileVide (), $L \leftarrow$ [ ], initialisation de vus, enfiler (s, F)
tant que F non vide, $\quad c \leftarrow$ defile F, $L \leftarrow c :: L$
$\quad$ pour $x \in G[c]$, $\quad$ si $vus[x] = 0$, enfiler (x, F), $vus[x] \leftarrow 1$ fin si
fin tant que
retourner (rev L)

Comme pour le parcours en profondeur, on propose un programme récursif, plus élégant, et un programme impératif.

Ecriture de l'algorithme en Caml.

`parc_larg g vus reste` renvoie la concaténation de la liste `vus` renversée et des parcours en largeur successifs sans répétition dont les origines sont les éléments de la liste `reste`.

```
(* Programmation récursive *)
let rec parc_larg g vus reste = match reste with
  | [] -> List.rev vus
  | t::q when List.mem t vus -> parc_larg g vus q
  | t::q -> parc_larg g (t::vus) (q @ g.(t))
;;
let parcours_larg g s = parc_larg g [] [s] ;;
```

A noter que la seule différence par rapport au programme récursif du parcours en profondeur réside dans la concaténation de `g.(t)` à la liste `q` effectuée en tête pour le parcours en profondeur et en queue pour le parcours en largeur.

```
(* Programmation impérative *)
open Queue ;;
let parcours_larg_imp g s =    (* initialisation*)
 let f = Queue.create () and l = ref [] and n = Array.length g in
```

```
 let vus = Array.make n 0 and c = ref s in
 Queue.add s f; vus.(s) <- 1;
 let rec aux l = match l with    (* fonction de mise à jour d'un sommet *)
       [] -> ()
   | x::q when vus.(x)=0 -> Queue.add x f; vus.(x) <- 1; aux q
   | x::q -> aux q
 in while not (Queue.is_empty f) do   (* boucle produisant le parcours en largeur *)
   c := Queue.take f;
   l := !c :: !l;
   aux g.(!c)
 done;
 List.rev !l
;;
```

Exemple :

```
let g = [|[10];[2;3];[1;4;5;6];[1;7;8];[2;5];[2;4];[2];[3;8;9];[3;7;9];[7;8];[0]|] ;;
parcours_larg_imp g 1 ;; parcours_larg g 1 ;;
```
On obtient dans les deux cas `[1; 2; 3; 4; 5; 6; 7; 8; 9]`.
`parcours_larg g 0 ;;` renvoie `[0; 10]`.

4 Dictionnaires

Un dictionnaire, ou table d'associations, est une structure de données composée d'objets associant à un ensemble de clés un ensemble de valeurs. Pour rendre les opérations de base efficaces, les clés pourront appartenir à un ensemble totalement ordonné. A chaque clé est associée une unique valeur. Ce type de données généralise celui de tableau où les clefs sont les indices du tableau.

Exemples.

- Le dictionnaire `departement`. On le représente par une liste de triplets.
  Par exemple : `[(75,"Paris","Paris");(78,"Yvelines","Versailles");(92,"Hauts de Seine","Nanterre");(91,"Essonne","Evry");(35,"Ille et Vilaine","Rennes")]`.
  La clé est le numéro du département, la valeur est le couple (nom, préfecture).
- Un dictionnaire dont les clés sont des marques automobiles et les valeurs des listes de modèles. `Renault : [Clio;Kadjar]  Peugeot : [308;3008;5008]  BMW : [120i;218d;325i;X3]`.
- Un graphe. Les clés sont les noms (ou les numéros) des sommets, les valeurs sont les voisins des sommets et éventuellement des données sur les sommets.

L'image concrète d'un dictionnaire peut être un agenda ou un annuaire téléphonique. Les clés sont les noms, les valeurs sont les numéros de téléphone ou les adresses.
Un autre exemple est tout simplement un dictionnaire dans l'acception courante où les clés sont les mots et les valeurs les descriptions de ces mots. Les bases de données fournissent un autre exemple, beaucoup plus élaboré, de dictionnaire.

La structure de dictionnaire doit supporter les opérations élémentaires de recherche d'un élément à partir de sa clé, d'insertion et de suppression.

Les objectifs de cette structure de données sont de minimiser le coût pour l'insertion et l'accès aux données, ainsi que de minimiser l'espace mémoire servant au stockage des données.

Dans le cas d'un tableau, la rigidité de la structure est un handicap pour les opérations d'insertion et de suppression.

Les opérations élémentaires de cette structure de données sont les suivantes :

- DicoVide () : crée un dictionnaire vide.
- EstVide (D) : teste si le dictionnaire D est vide.
- Chercher (k, D) : recherche la valeur présente dans le dictionnaire D associée à la clé k.
- Inserer ($(k, x), D$) : ajoute une nouvelle entrée x de clé k au dictionnaire D.
- Supprimer (k, D) : supprime l'entrée de clé k du dictionnaire D.

On peut facilement implémenter la structure de dictionnaire par une liste mutable ou un tableau. Le coût des opérations élémentaires sera linéaire par rapport au nombre d'entrées, ce qui limite son intérêt. Pour améliorer ce coût, il faut utiliser une implémentation plus complexe à l'aide d'une table de hachage (structure modifiable), ce qui sera fait dans le chapitre suivant, ou d'un arbre binaire de recherche (structure persistante, voir programme de 2ème année), ou plus efficacement d'un arbre AVL.

Module OCaml Map

Le module `Map` permet de gérer une structure persistante de dictionnaire dans laquelle les clés appartiennent à un ensemble totalement ordonné. L'implémentation à partir d'arbres équilibrés garantit que la recherche et l'insertion ont un coût logarithmique.

Pour définir un module gérant des dictionnaires à partir d'un certain type de clés, on doit d'abord créer un module définissant les clés. La syntaxe est la suivante :

```
module Clés =
   struct
     type t = ...
     let compare ...
    end
```

Si `t` est le type des clés, la fonction `compare : t -> t -> int` permet de disposer d'un ordre total sur les clés :

`compare x y` > 0 si et seulement si `x > y`
`compare x y` < 0 si et seulement si `x < y`
`compare x y` = 0 si et seulement si `x = y`

Puis on crée le module principal dictionnaire : `module Dictionnaire = Map.make(Clés)`

`empty : 'a t` : crée un dictionnaire vide.

Fonctions principales du module `Map`

`is_empty : 'a t -> bool` : teste si un dictionnaire est vide.
`add : key -> 'a -> 'a t -> 'a t` : crée un dictionnaire en ajoutant la nouvelle entrée au précédent.
`find : key -> 'a t -> 'a` : renvoie la valeur associée à la clé ou déclenche l'exception `Not_found` s'il n'y en a pas.
`remove : key -> 'a t -> 'a t` : renvoie un dictionnaire en enlevant du précédent l'élément éventuel de clé donnée.
`iter : (key -> 'a -> unit) -> 'a t -> unit` : itère une fonction sur les éléments d'un dictionnaire.

Autres fonctions :

`mem : key -> 'a t -> bool` : teste si une clé est présente dans le dictionnaire.
`cardinal : 'a t -> int` : renvoie le nombre d'entrées du dictionnaire.
`equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool` : teste l'égalité de deux dictionnaires (égalité des clés, puis égalité des valeurs avec la fonction passée en premier argument).
`bindings : 'a t -> (key * 'a) list` : renvoie la liste triée (selon les clés) des couples (clés, valeurs) du dictionnaire.
`min_binding : 'a t -> key * 'a` : renvoie le couple de clé minimale. Idem avec `max_binding`.
Les fonctions `exists`, `for_all`, `filter`, `fold`, `map` ont la même signification que pour les listes.

Exemple :

On saisit une liste d'élèves, dans laquelle chaque élément de la liste est un tuple comportant le nom, le prénom, l'année, la classe, le lycée d'origine en Terminale, l'intégration.
On crée un dictionnaire dont les clés sont les couples (nom, prénom) et les valeurs le reste des informations.
On écrit une fonction remplissant automatiquement le dictionnaire à partir d'une liste d'élèves.
On écrit une fonction recherchant les élèves venant d'une classe une année donnée.
On écrit une fonction recherchant les élèves venant d'un lycée donné.

```
let eleve18 = 2018,
[("Adler","Malo","A","Hoche",""); ...]
;;
let remplissage dico couple =
  let (an,l) = couple in List.fold_left
  (fun map (nom,prenom,classe,origine,integration) -> PairsMap.add (nom,prenom) (an,classe,origine,inte
  dico l
;;
let recherche an classe dico =
  let l = PairsMap.bindings dico in
  List.filter (fun (x,(a,b,c,d)) -> a = an && b = classe) l
;;
let dico2016 = remplissage PairsMap.empty eleve16 ;;
let dico2017 = remplissage dico2016 eleve17;;
let dico = remplissage dico2017 eleve18 ;;

PairsMap.find("Fauvel","Axel") dico ;;
PairsMap.cardinal dico ;;
PairsMap.bindings dico ;;
recherche 2016 "B" dico ;;
```

5 Files de priorité

5.1 Définition

On dispose d'un ensemble U totalement ordonné de clés et d'un ensemble V de valeurs.
Une file de priorité est une structure de données permettant de gérer un ensemble dynamique S inclus dans V, chaque élément de S ayant une clé appartenant à U indiquant sa priorité. Une file de priorité doit supporter les opérations suivantes :

- Inserer (x, S) : insère l'élément x dans S (avec sa priorité).
- Maximum (S) : retourne l'élément de S ayant la plus grande priorité.
- Extraire-max (S) : retourne et supprime l'élément de S ayant la plus grande priorité.

On peut remplacer la recherche du maximum par celle du minimum.
La version basique d'une file de priorité est celle où clé et valeur sont confondues (par exemple une liste d'entiers).

Les files de priorité peuvent être utilisées par exemple pour la planification des tâches sur un ordinateur, en attribuant à chaque tâche une priorité, ou pour la gestion du contrôle aérien dans un aéroport.

Implémentation impérative

- Creer : unit $\to$ fp : crée une file de priorité vide.
- EstVide : fp $\to$ bool : teste si une file est vide.
- Inserer : cle $\times$ elt $\times$ fp $\to$ unit : insère un élément avec sa priorité dans une file.
- ExtraireMax : fp $\to$ elt : extrait de la file l'élément de plus haute priorité.
- Maximum : fp $\to$ elt : renvoie l'élément de plus haute priorité (sans le supprimer).

Une implémentation naturelle peut se faire au moyen d'une liste chaînée, où chaque maillon est le couple (clé, valeur). L'insertion est en $\mathcal{O}(1)$ mais la recherche du maximum est en $\mathcal{O}(n)$, ce qui présente peu d'intérêt.
On peut améliorer l'implémentation d'une file de priorité en utilisant une structure de tas.

5.2 Tri à l'aide d'une file de priorité

Proposition. *Si une file de priorité offre des opérations d'ajout et de retrait de coûts logarithmiques par rapport à sa taille, alors elle permet d'implanter un algorithme de tri de N éléments en $\mathcal{O}(N \log N)$.*

preuve. Soit E l'ensemble des clés à trier. On insère successivement les éléments de E dans une file de priorité initialement vide. Le coût de l'insertion est $\mathcal{O}(\sum_{k=1}^{N} \log k)$. On extrait le maximum de la file de priorité successivement jusqu'à vider la file. Le coût global est donc $\mathcal{O}(\sum_{k=1}^{N} \log k) = \mathcal{O}(N \log N)$. •

Dans le code suivant, on suppose pour simplifier que clé et valeur coïncident.

Opérations élémentaires :
`creer_fp : 'a -> 'a fp ;;` `creer_fp v` crée une file de priorité initialisée à la valeur `v`.
`inserer : 'a -> 'a fp -> unit ;;` `inserer e f` insère l'élément `e` dans la file de priorité `f`.
`extraire_max : 'a fp -> 'a ;;` `extraire_max f` extrait l'élément de priorité maximale de la file de priorité `f`.

```
let tri v =
 let n = Array.length v in let w = Array.make n 0 in
 let f = creer_fp 0 in
 for i = 0 to n-1 do inserer v.(i) f done;
 for i = 1 to n do w.(n-i) <- extraire_max f done;
 w ;;
```