I Structure de données

I.1 Définition

Exercice 1

- `array` : mutable et statique
- `string` : mutable et statique
- `char * char` : persistant
- `int * int ref` : mutable et statique
- `int array ref` : mutable et dynamique

I.2 Création de type en Caml

Exercice 2

1.
```
type nat = Zero | S of nat;;
```

2.
```
let nul = function
    | Zero -> true
    | _ -> false;;
```

3.
```
let rec entier = function
    | Zero -> 0
    | S x -> 1 + entier x;;

let rec peano = function
    | 0 -> Zero
    | n -> S (peano (n - 1));;
```

4.
```
let rec addition a b = match a, b with
    | Zero, _ -> b
    | S x, y -> addition x (S y);;

let rec multiplication a b = match a, b with
    | Zero, _ -> Zero
    | S x, y -> addition y (multiplication x y);;
```

5.
```
let rec plus_grand a b = match a, b with
    | _, Zero -> true
    | Zero, _ -> false
    | S x, S y -> plus_grand x y;;
```

II Listes chaînées

II.1 Type de données abstrait

II.2 Implémentation Caml

Exercice 3

```
let deuxieme = function
   | [] -> failwith "liste vide"
   | [x] -> failwith "Un seul élément"
   | x :: y :: q -> y;;
```

II.3 Utilisation de la récursivité

II.4 Fonctionnelles sur les listes

Exercice 4

```
let length = List.fold_left (fun x y -> x + 1) 0;;

let mem a = List.fold_left (fun x y -> x || y = a) false;;

let map f l = List.fold_right (fun x y -> (f x) :: y) l [];;

let iter f = List.fold_left (fun x y -> f y) ();;
```

III D'autres structures linéaires

III.1 Piles

Exercice 5

```
let creer_pile () = {liste = []};;

let pile_vide p = p.liste = [];;

let empiler p x = p.liste <- x :: p.liste;;

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

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

Toutes ces fonctions sont de complexité constante.

III.2 Files

Exercice 6

```
type 'a file = {mutable enfilage : 'a list ; mutable defilage : 'a list};;

let creer_file () = {enfilage = [] ; defilage = []};;

let file_vide p = p.enfilage = [] && p.defilage = [];;

let enfiler p x = p.enfilage <- x :: p.enfilage;;

let vider_enfilage p = let rec aux l1 l2 = match l1 with
   | [] -> l2
   | x :: q -> aux q (x :: l2) in
   p.defilage <- aux p.enfilage p.defilage;
   p.enfilage <- [];;

let premier p =
   if p.defilage = [] then vider_enfilage p;
   match p.defilage with
     | [] -> failwith "file vide"
     | x :: q -> x;;

let dernier p =
   let rec dernier_liste = function
      | [] -> failwith "file vide"
      | [x] -> x
      | x :: q -> dernier_liste q in
   if p.enfilage <> [] then List.hd p.enfilage
   else dernier_liste p.defilage;;

let defiler p =
   if p.defilage = [] then vider_enfilage p;
   match p.defilage with
     | [] -> failwith "file vide"
     | x :: q -> p.defilage <- q; x;;
```

Les trois premières fonctions se font en temps constant. La fonction `vider_enfilage` permet de vider la pile d'enfilage dans celle de défilage.
La fonction `premier` et `defiler` peuvent ponctuellement être exécutée en complexité linéaire (lorsqu'un vidage est nécessaire). Cependant, en moyenne, elles s'exécutent en temps constant. En effet, chaque élément qui passe par la file n'est manipulé que trois fois (une fois à l'enfilage, une fois au vidage et une fois au défilage).
La fonction `dernier` peut s'exécuter en temps constant tant que la pile d'enfilage n'est pas vide. Par contre, elle s'exécutera en temps linéaire si la pile d'enfilage est vide.