Chapitre 5

Piles et files.

1 Pile LIFO

La notion de pile est une notion fondamentale en informatique. Tout processeur utilise une pile.
Une pile informatique est tout à fait comparable à une pile d'assiettes. Quand on range une assiette dans une pile, on la pose sur le haut de la pile et quand on veut une assiette dans une pile, on prend celle qui est sur le dessus. Dans une pile informatique, c'est le même principe, on a accès qu'au sommet de la pile, les autres éléments étant invisibles.

Pour les piles, il existe deux opérations : la fonction d'empilement et la fonction de dépilement.

- Dans la fonction d'empilement (on dit aussi empilage et push en anglais) on ajoute un élément au sommet de la pile.
- Dans la fonction de dépilage (pop en anglais), on retourne l'élément qui est au sommet de la pile et on le supprime de la pile.

À ces deux opérateurs, il faut ajouter une fonction de création de pile. Dans la littérature anglo-saxonne, les piles se nomment stack ainsi que les files que nous verrons un peu plus tard. Dans une pile, le dernier élément rentré est le premier qui sort ce qui donne en anglais : (Last In, First Out), on parle de façon abrégée de pile LIFO.

1.1 En utilisant des listes référencées.

On utilise le type suivant :

```
type 'a pile = 'a list ref;;
```

La fonction de création de pile devient donc :

```
let cree_pile ()=  ( ref [] : 'a pile);;
```

Exemple :

```
let pile1 = cree_pile ();;
```

On crée ainsi une pile, initialement vide.

Exercice 1 :

1. Écrire la fonction `empile` qui prend en entrée une pile p, un élément a et ajoute l'élément à la pile.

   ```
   empile :  'a pile -> 'a -> unit
   ```

2. Écrire la fonction `pile_vide` qui renvoie le booléen correspondant au caractère vide de la pile.

   ```
   pile_vide :  'a pile -> bool
   ```

3. Écrire la fonction `depile` qui prend en entrée une pile p et retourne l'élément qui est au sommet de la pile tout en le supprimant de la pile.

```
depile :  'a pile -> 'a
```

Exercice 2 :
En utilisant les quatre fonctions précédentes, écrire une fonction `somme` qui somme les éléments d'une pile d'entiers.

```
somme : int pile -> int
```

1.2 Implémentation par des listes dans des enregistrements mutables

On peut aussi définir les piles à l'aide d'une liste dans un enregistrement ce qui évite d'avoir à forcer le typage des fonctions :

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

Exercice 3 :
Réécrire les fonctions `cree_pile`, `empile`, `pile_vide` et `depile` avec ce type.

1.3 Implémentation en utilisant des tableaux :

On utilise ici un tableau de taille fixe et un indicateur de la position courante(indice du premier emplacement libre du tableau) :

```
type 'a pile = {p : 'a array; long : int; mutable pos : int};;
```

Exercice 4 :
Réécrire les fonctions `cree_pile` (on donnera ici la taille de la pile en argument, ainsi qu'un élément pour définir le type de la pile),`pile_vide` , `pile_pleine` , `empile`, et `depile` avec ce type.

2 Pile FIFO

Dans une file de caisse d'un supermarché, la première personne qui est rentrée dans la file est la première qui doit en sortir. En informatique, nous avons aussi besoin de gérer des files.

Pour les listes, il existe deux opérations de base : la fonction de mise en file d'attente et la sortie de la file d'attente.

- Dans la fonction d'ajout (add en anglais) d'un élément en fin d'une liste d'attente.
- Dans la fonction de prise (take en anglais) de l'élément qui est en tête de la file d'attente et que l'on le supprime de la file.

À ces deux opérateurs, il faut ajouter une fonction de création de file. Dans la littérature anglo-saxonne, les files se nomment queue. Dans une file, le premier élément rentré est le premier qui sort ce qui donne en anglais : (Firt In, First Out), on parle de façon abrégée de pile FIFO.

Nous allons donc simuler les files de deux façons, l'une utilisant des tableaux et l'autre des listes.

2.1 Utilisation de tableaux

Dans ce cas, on représente une liste à l'aide d'un enregistrement contenant :

- un tableau dans lequel sont stockés les éléments de la file,
- un entier (mutable) indiquant le premier élément de la file,
- un entier (mutable) indiquant la première case vide de la file,
- un entier donnant la longueur du tableau.

Lorsque les pointeurs sur la position de fin et de début de file sont égaux, c'est que la file est vide.
Au fur et à mesure de l'ajout d'éléments dans la file, le pointeur de fin de liste est incrémenté modulo la longueur du tableau. De même, au fur et à mesure du retrait d'éléments de la liste, le pointeur est incrémenté modulo la longueur du tableau. On a ainsi une occupation (tournante) du tableau, le premier élément de la file n'étant pas à une place déterminé dans le tableau.

Pour différencier une file vide d'une file pleine, on dira que la file est pleine si la case du premier élément de la file est juste derrière la première case vide. En fait, si notre tableau est de longueur n, on utilisera au plus $n-1$ cases, la case précédant la case du premier élément de la file devant toujours rester vide.

On définie une file par :

```
type 'a file = {tab : 'a array; mutable debut : int; mutable fin : int; long :
 int};;
```

Exercice 5 :
Écrire la fonction `cree_file` qui crée une file(en entrée on donnera la longueur de la file et un élément `a`),puis les fonctions `file_pleine`, `file_vide`, `ajoute` et `retire`.

2.2 Utilisation de listes

Dans le cas de l'utilisation des listes, c'est plus simple. La file est représentée par une liste (mutable). L'élément en tête de liste représente le début de la file et l'élément en queue de liste représente la fin de la file. Donc logiquement, pour ajouter un élément à une file ainsi représentée, on l'ajoute en queue de liste et pour retirer un élément, on prend celui qui est en tête de liste.

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

Exercice 6 :
Écrire la fonction `cree_file` et les fonctions `file_vide`, `ajoute` et `retire`.