Exercice 1 : Files en C par maillons chaînés

Dans cet exercice, on représente une file en C par des maillons chainées. En plus de l'accès au début de la séquence de maillons chaînés, on garde un pointeur vers le dernier élément de la séquence de maillons. La raison est la suivante : Le *point d'entrée* de la file sera représenté par la *fin* de la séquence de maillons chaînés. Le *point de sortie* de la file sera représenté par le *début* de la séquence de maillons chaînés.
On peut alors facilement ajouter un élément à la file en ajoutant un élément à la fin de la séquence de maillon grâce au pointeur vers le dernier maillon, et défiler un élément en supprimant le premier maillon. On ne peut supprimer facilement le dernier maillon puisque nous n'avons pas facilement accès à un pointeur vers l'avant dernier maillon de la séquence.

Q. 1 Définir un type structuré `int_file` contenant :
- un pointeur vers le premier élément d'une séquence de maillons ;
- un pointeur vers le dernier élément de cette même séquence de maillons.

Q. 2 Définir une fonction `int_file file_vide()` retournant une file vide.

Q. 3 Définir une fonction `bool file_est_vide(int_file file)` permettant de tester si une file est vide.

Q. 4 Définir une fonction `void enfiler(int elem, int_file file)` permettant d'enfiler un entier dans une file.

Q. 5 Définir une fonction `int defiler(int_file file)` permettant de sortir un entier se trouvant dans une file.

Exercice 2 : Implantation d'une file avec deux piles en C

Cet exercice nécessite l'utilisation d'une structure de pile en C. Si vous n'avez pas encore d'implantation d'une telle structure, l'implanter puis revenir à l'exercice courant.

Q. 1 Définir une fonction `void transvaser(pile p1, pile p2)` vidant la pile `p1` dans la pile `p2`.

Q. 2 Définir un type `file` comme étant deux piles.

Q. 3 Compléter l'implantation de file en y ajoutant les opérations `file file_vide();`, `bool est_vide(file f);`, `void enfiler(int elem, file* f)`, et `int defiler(file* f)`. Noter que la signature des fonctions induit une manipulation *par effet de bord* de vos files.

Exercice 3 : Retour à la NPI

Dans cet exercice on s'efforce de faire le travail manquant au TP de la semaine dernière sur la NPI : la transformation d'une chaîne de caractères en NPI à savoir une liste de composants, qui sont :
- ou bien un opérateur ;
- ou bien un entier.

L'algorithme de transformation peut être lui-même implanté au moyen d'une pile de *travail à faire* et d'une liste résultat qui est l'expression sous forme NPI en construction. À chaque itération de l'algorithme on retire la chaîne de caractères en sommet de pile et :

— Si cette chaîne de caractères admet un opérateur (+, -, *, /) ne se trouvant pas dans une paire de parenthèses, alors
  — on empile la sous chaîne précédant ce caractère,
  — on empile la sous chaîne suivant ce caractère,
  — on ajoute l'opérateur en question au résultat.
  Dans le schéma ci-dessous, ce sont les transformations des étapes 2 vers 3, 4 vers 5, 7 vers 8.
— Sinon si cette chaîne de caractères commence par une parenthèse ouvrante, c'est que la parenthèse fermante associée est en fin de chaîne, on ajoute à la pile la sous chaîne privée de ces deux parenthèses. Dans le schéma ci-dessous, ce sont les transformations des étapes 1 vers 2, 3 vers 4, 6 vers 7.
— Sinon cette chaîne de caractères est un entier, cette entier est ajouté dans le résultat. Dans le schéma ci-dessous, ce sont les transformations des étapes 5 vers 6, 8 vers 9, 9 vers 10, 10 vers 11

Étape	1	2	3	4	5
Pile	(1+((2+5)*3))	1+((2+5)*3)	((2+5)*3) 1	(2+5)*3 1	3 (2+5) 1
Résultat			+	+	* +

Étape	6	7	8	9	10	11
Pile	(2+5) 1	2+5 1	5 2 1	2 1	1	
Résultat	3 * +	3 * +	+ 3 * +	5 + 3 * +	2 5 + 3 * +	1 2 5 + 3 * +

Implantation modulaire Afin de faire un premier pas dans la direction de la modularité en OCaml, il vous est demandé d'implanter une structure de Pile dans un `module`. Un module est un ensemble de type, valeurs et fonctions qui sont regroupées sous un même ensemble de nom, c'est le cas par exemple du `module List`.
On peut définir simplement un module en OCaml en écrivant :

```
module Pile = struct

  (* le type des piles *)
  type 'a t = ...

  (* Les différentes fonctions de manipulation de pile *)
  let empty = ...
  let is_empty (l: 'a t) : bool = ...
  let empiler (x: 'a) (l: 'a t) = ...
  let depiler (l: 'a t) : 'a * 'a t = ...

end
```

Vous pourrez alors accéder aux types et fonctions du module en écrivant `Pile.…`

Chaînes de caractères L'accès à une chaîne de caractères en OCaml se fait au moyen de la notation (e_1`.[`e_2`]`) où e_1 a un type `string` et e_2 a un type `int`. Par exemple `"0123".[1+1] = '2'`. On

pourra (devra ?) utiliser les fonctions : `String.sub s i len` fabriquant la sous-chaîne de `s` commençant en `i` et de longueur `len`, ainsi que `int_of_string: string -> int` permettant la transformation d'une chaîne de caractères en la valeur entière représentée.

Exemples :

```
String.sub "012345" 2 3 = "234"
int_of_string "234" = 234
```

Plutôt que de manipuler des sous-chaînes d'une chaîne `s` dans tout l'énoncé, on manipulera des paires d'entiers (i, j) représentant alors la sous-chaîne de `s` allant des indices i à j inclus, que l'on dénotera `s`$[$`i`$\ldots$`j`$]$.

Q. 1 Implanter votre module de pile favori (ou mieux copier-coller une implantation précédente).

Q. 2 Définir une fonction `find_op_out_of_parent (s: string) (i: int) (j: int): unit` prenant en arguments une chaîne de caractères `s`, deux entiers `i` et `j`, cette fonction s'évaluera en `()` si elle ne trouve pas d'opérateur dans la sous-chaîne `s`$[$`i`$\ldots$`j`$]$ se trouvant en dehors de parenthèses et devra lever l'exception `Split(x)` où `x` est l'indice dans la sous-chaîne de caractères `s`$[$`i`$\ldots$`j`$]$ de l'opérateur se trouvant en dehors de toute parenthèse.

Exemples :

`find_op_out_of_parent "(1+2)" 1 3` devra lever l'exception `Split(2)`

`find_op_out_of_parent "((1+2))" 1 5` devra s'évaluer en `()`

Q. 3 Définir une fonction `to_npi : string -> expr` (où `expr` est le type des expressions sous forme NPI, défini la semaine dernière) appliquant l'algorithme décrit ci-avant. On rappelle que votre pile devra manipuler des paires d'entiers et non des chaînes de caractères.

Exercice 4 : Tri en place d'une file

On dit d'un algorithme de tri qu'il est *en place* si sa consommation mémoire ne dépend pas de la taille de son entrée. À savoir l'algorithme n'alloue pas de mémoire en plus de celle qui lui est passée en argument, mis à part une mémoire de taille constante, ne dépendant pas de l'entrée.

Q. 1 Proposer un algorithme de tri en place d'une file. Votre algorithme devra prendre en entrée une file, la manipuler par des opérations de file, ne pas agrandir cette file plus qu'un ajout constant (ne dépendant pas du contenu de la file). À la fin de l'exécution de l'algorithme la file doit contenir les éléments initiaux dans un ordre de sortie décroissant.

Q. 2 Implanter l'algorithme précédent dans le langage de programmation de votre choix.