TP 6 : Structures de données

On utilisera la ligne de commande suivante pour compiler en C :

```shell
$ gcc -O0 -Wall -Wextra -Wvla -Werror -fsanitize=address,undefined votre_fichier.c -o votre_executable
```

Exercice 1 – Listes chaînées en C

On peut implémenter le type abstrait liste en C à l'aide du type "concret" suivant (on se limitera à des listes de `double` dans cet exercice) :

```c
struct Chainon {
    double tete;
    struct Chainon* queue;
};

typedef struct Chainon liste;
```

On représente une liste par un pointeur vers cette structure.
On remarque que l'on ne peut pas vraiment représenter de liste vide avec ce type, puisqu'il doit avoir au moins une tête et une queue. Ainsi, on choisit la convention suivante : la liste vide est représentée par le pointeur `NULL`.

▶ **Question 1** Écrire une fonction `liste* liste_a_un_element(double x)` qui crée une liste ne contenant qu'un seul élément `x`.

▶ **Question 2** Implémenter la fonction `liste* cons(double t, liste* q)` qui crée une nouvelle liste enchaînant en tête `t` et `q`.

▶ **Question 3** À l'aide des fonctions précédentes uniquement, écrire une fonction `liste* depuis_tableau(double* tab, int n)` qui crée une liste contenant les éléments du tableau $[tab[0], \dots, tab[n-1]]$ dans le même ordre.

▶ **Question 4** Implémenter les fonctions `hd` et `tl` du type abstrait liste chaînée.

▶ **Question 5** Écrire une fonction `void liberer_liste(liste* l)` qui libère toute la mémoire utilisée par `l`.

▶ **Question 6** Supposons que l'on dispose d'une fonction de prototype `void f(double x)` qui effectue une opération sur un `double` et ne renvoie rien. Écrire une fonction `void iter_liste(liste* l)` qui exécute dans l'ordre de la liste* la fonction `f` sur les éléments de `l`. *On pourra la tester avec une fonction `f` qui affiche `x`.*

▶ **Question 7** Écrire une fonction `double somme(liste* l)` calculant la somme des éléments de `l`.

▶ **Question 8** Écrire une fonction `int longueur(liste* l)` qui calcule la longueur d'une liste.

▶ **Question 9** En déduire une fonction `double* vers_tableau(liste* l)` qui stocke dans un tableau les éléments de `l`.

▶ **Question 10** Écrire une fonction `bool sont_egales(liste* l1, liste* l2)` testant si deux listes contiennent les mêmes éléments.

▶ **Question 11** Quelles sont les différences entre notre implémentation des listes et celle d'OCaml ?

On peut écrire des fonctions sur les listes chaînées de manière similaire à OCaml :

▶ **Question 12** Écrire une fonction `bool est_triee(liste* l)` vérifiant qu'une liste est triée.

▶ **Question 13** Écrire une fonction `liste* insere(double x, liste* l)` qui, pour une liste triée `l`, renvoie une liste triée contenant les mêmes éléments que `l` plus une occurence de `x`. On n'utilisera qu'une seule allocation mémoire supplémentaire.

▶ **Question 14** Faire un schéma de l'état de la mémoire avant et après une insertion `l1 = insere(x, l)`, dans les cas suivants :

- Si la liste `l` est vide,
- Si la liste `l` est non vide et qu'on insère l'élément en première position,
- Si la liste `l` est non vide et qu'on insère l'élément à l'intérieur de `l`.

Comparer avec OCaml.

▶ **Question 15** En déduire pourquoi la fonction `insere` risque d'occasionner des fuites de mémoire non désirées.

▶ **Question 16** Implémenter le tri par insertion sur le type `liste`. Elle ne doit pas modifier la liste initiale, mais renvoyer une nouvelle liste.

Exercice 2 – Piles par les listes

Vous pourrez refaire cet exercice en OCaml en utilisant le type `type 'a pile = 'a liste ref`. Comme dans le cours, on utilise des notations en OCaml pour décrire le type abstrait pile.
On va utiliser le type `liste` défini dans l'exercice précédent pour implémenter les *piles impératives*.
Les piles sont un type de données permettant d'*empiler* des éléments les uns sur les

autres. Cette structure ressemble peu ou prou à une liste chaînée. Ici, on présentera le principe d'une pile *mutable*, c'est-à-dire où l'on commence par construire une pile vide puis l'on la modifie pour la remplir.

Opération	Spécification	Type de l'opération
Constructeur	Crée une pile vide	`new_stack : unit -> 'a pile`
Accesseur	Teste si une pile est vide	`is_empty : 'a pile -> bool`
Transformateur	Ajoute un élément à une pile	`push : 'a -> 'a pile -> unit`
Transformateur	Retire un élément de la pile et le renvoie	`pop : 'a pile -> 'a`

On y ajoute les axiomes suivants : un appel à `push` puis à `pop` laisse la pile inchangée, comme un appel à `pop` puis à `push` avec la valeur dépilée. De même, `est_vide empty_pile` est vrai et après un appel à `push x p`, `is_empty p` est faux. Ici, on ajoute une précondition : pour pouvoir appeler `pop p`, il faut que `not (is_empty p)`. Notamment, une pile respecte le principe LIFO (*Last In First Out*) : le dernier élément à être empilé sera le premier dépilé.

C
```c
typedef struct {
  liste* donnees;
  int taille;
} pile;
```

▶ **Question 1** Écrire les fonctions de la signature pour implémenter une pile et indiquer leurs complexités. Elles ne devront pas permettre de fuite de mémoire.

▶ **Question 2** Supposons maintenant que l'on ne connaît pas les détails de l'implémentation du type abstrait pile, et donc que l'on ne peut les manipuler qu'avec les fonctions `pop`, `push`, `is_empty`. Écrire les fonctions suivantes :

- `double peek(pile* p)` renvoyant la dernière valeur ajoutée à la pile. Après l'appel, la pile ne doit pas être modifiée.
- `bool egal(pile* p1, pile* p2)` qui détermine si deux piles sont égales. Après l'appel, les deux piles ne doivent pas être modifiées.
- `void iter_pile(pile* p)` qui exécute la fonction `f` de l'exerrcice 1 sur tous les éléments de la pile. Après l'appel, la pile ne doit pas être modifiée.

On remarque que ça commence à faire du code pénible, alors qu'en ayant accès à l'implémentation, ça serait beaucoup plus simple (et d'une bien meilleure complexité). Par exemple, c'est pour cela que les modules OCaml proposent beaucoup de fonctions au-delà de la signature minimale du type abstrait correspondant.

Exercice 3 – Piles fonctionnelles par les listes chaînées

On peut aussi choisir de définir un type abstrait de piles fonctionnelles, où les deux transformateurs sont transformées en des constructeurs, c'est-à-dire qui construisent de nouvelles piles en utilisant les anciennes.

Opération	Spécification	Type de l'opération
Constructeur	Crée une pile vide	`'a pile_fonc`
Accesseur	Teste si une pile est vide	`is_empty : 'a pile_fonc -> bool`
Accesseur	Ajoute un élément à une pile	`push : 'a -> 'a pile_fonc -> 'a pile_fonc`
Accesseur	Retire un élément de la pile et le renvoie	`pop : 'a pile_fonc -> 'a * 'a pile_fonc`

▶ **Question 1** Est-ce que ce type de pile fonctionnelle vous fait penser à un type que l'on a beaucoup utilisé en OCaml ?

▶ **Question 2** Implémenter les opérations précédentes avec ce type en OCaml.

Il est aussi possible d'implémenter les files dans des tableaux : vous pouvez lire une proposition d'implémentation dans la Section IV.B du Chapitre 9. Après avoir lu cette section, vous pouvez faire l'exercice suivant :

Exercice 4 – Implémentation des files dans un tableau circulaire

Le cours propose une idée d'implémentation du type abstrait file, que vous pouvez trouver dans la Section IV.C.

▶ **Question 1** Implémenter les files par des tableaux circulaires en C, avec des types enregistrements. On choisira avec soin quel est le comportement quand on pousse un *n*ième élément dans la file.

▶ **Question 2** On appelle *Deque* (*double-ended queue*, file à double extrémité) le

type de donnée abstrait où l'on s'autorise à ajouter et retirer un élément à gauche et à droite de la file, avec les opérations `pop_left`, `pop_right`, `push_left`, `push_right`. Modifier votre implémentation pour obtenir une structure de donnée *Deque*.

Exercice 5 – Files par les liste

On va implémenter le type abstrait de file fonctionnelle comme un couple de listes. L'idée va être de représenter la file contenant `x0 x1 ... xn` par un couple de listes `[x0; x1 ... xk] [xn; xn-1; ... xk+1]`. Notamment, on aura :

```OCaml
type 'a queue_fonc = 'a list * 'a list
let empty_queue = ([], [])
```

▶ **Question 1** Écrire les fonctions `is_empty` et `push`. Écrire une fonction `list_to_file : 'a list -> 'a file` qui renvoie la file obtenue après avoir enfilé successivement les éléments de la liste. *Indication : ces trois opérations sont en* $\mathcal{O}(1)$.

▶ **Question 2** Écrire une fonction `egal : 'a queue -> 'a queue -> bool` vérifiant si deux piles sont égales. *Attention,* `let egal q r = (q = r)` *ne fonctionne pas.*

▶ **Question 3** Dans quel cas défiler n'est pas trivial dans cette représentation ? Proposer une implémentation de la fonction pop. *Pour rappel, la fonction* `List.rev` *renvoie la liste en entrée retournée.*

L'intérêt de cette stratégie est que l'opération coûteuse (retourner une liste) est exécutée rarement. C'est le principe de la *complexité amortie* : dans la suite, on s'intéressera à la complexité $C(f)$ comme le nombre d'utilisations du constructeur `::` (que ce soit pour construire une nouvelle liste ou pour filtrer une liste). Notons $|l|$ la taille d'une liste l. Pour une file `f = (lg, ld)`, on appelle $\phi(f) = 2|lg|$ le potentiel de f.

▶ **Question 4** Majorer $C_{push}(f) + \phi(f') - \phi(f)$, avec C_{push} la complexité de push et f' la file obtenue après un appel à push. Faire de même pour pop.

▶ **Question 5** En déduire la complexité de n appels successifs à push ou pop est en $\mathcal{O}(n)$.

Ainsi, en moyenne les appels à push ou pop se font en $\mathcal{O}(1)$.

▶ **Question 6** En n'utilisant pas l'implémentation des files par un couple de liste, programmer les fonctions suivantes :

- `somme : int file -> int` calculant la somme des éléments d'une file,
- `affiche_file : float file -> unit` affichant une file de flottant dans l'ordre d'insertion (on choisira librement la présentation),
- `file_to_list : 'a file -> 'a list` calculant la liste des éléments de la file, le prochain élément à défiler étant en tête de la liste,
- `list_to_file : 'a list -> 'a file` calculant l'inverse. Comment pourrait-on faire plus efficace en ayant accès à l'implémentation ?
- `iter_file : ('a -> unit) -> 'a file -> unit` qui itère la fonction en entrée sur les éléments défilés successivement,
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b file -> 'a` tel que l'appel `fold_left fonc a f` est évalué `fonc (... (fonc (fonc a f0) f1) ...) fn` avec `f0 ... fn` les éléments dans l'ordre de défilement de la file `f`.

L'exercice suivant permet d'implémenter les tableaux redimensionnables en OCaml, ce qui n'est possible que si vous avez déjà vu les tableaux, références et champs mutables en OCaml (dans le chapitre Chapitre 8). Après Noël donc !

Exercice 6 – Tableaux redimensionnables

L'objectif de cet exercice est d'implémenter le type abstrait tableau redimensionnable, qui à la signature des tableaux ajoute l'opération `resize : 'a vector -> int -> unit` tel que `resize t n` redimensionne `t` à la taille `n` : si cela fait grandir le tableau au-delà de ce que l'on avait déjà prévu, on crée un nouveau tableau plus grand. On remarque que le type `'a array` ne dispose pas d'une telle opération : par exemple, la fonction `Array.append : 'a array -> 'a array -> 'a array` crée un nouveau tableau qui contient les éléments du premier tableau puis du deuxième, mais ne modifie pas ses arguments.
On utilisera le type suivant :

```OCaml
type 'a vector = {
  mutable data : 'a array;
  mutable length : int;
  default : 'a
}
```

▶ **Question 1** Écrire les fonctions de la signature du type abstrait tableau. À la création d'un tableau redimensionnable, on y précise sa capacité initiale, sa taille initiale et la valeur par défaut de ses cases. Y ajouter une fonction `length : 'a vector -> int` renvoyant la taille du tableau redimensionnable.

▶ **Question 2** On propose une première stratégie qui tel que si `'a vector` est de

taille `len` et que `n > len`, alors on redimensionne le tableau pour contenir exactement `n` éléments.

▶ **Question 3** Supposons qu'on initialise un tableau dynamique à la taille 0 et que pour $n = 1$ à N, on exécute `resize t n`. Déterminer la complexité de chaque appel, puis la complexité de la séquence d'instructions entière. Déterminer la moyenne de la complexité de chaque appel (c'est-à-dire la complexité *amortie* de `resize`).

▶ **Question 4** Modifier la stratégie pour que l'appel si `n > len`, `resize t n` redimensionne le tableau à la taille $2^{\lceil \log_2 n \rceil}$, c'est-à-dire, la plus petite puissance de deux supérieure ou égale à n. Avec cette stratégie, quel est la complexité amortie de `resize` ?

▶ **Question 5** Pourrait-on utiliser cette structure de donnée pour définir une pile de taille non bornée ? Une file de taille non bornée ? *Les critères à considérer sont la complexité en temps et en espace des opérations sur une telle structure de donnée.*

En OCaml, les piles sont implémentées par des références de liste chaînées (avec un autre champ mutable pour leur taille, similaire à l'exercice 2), les files par des listes chaînées modifiables avec un pointeur sur le premier élément et un pointeur sur le dernier.

Exercice 7 – Une application des piles : la calculatrice polonaise inversée

La calculatrice polonaise inversée est une façon d'écrire des expressions arithmétiques sur les entiers sans avoir besoin de parenthèses. L'idée est d'écrire d'abord les deux *opérandes* (c'est-à-dire les valeurs) puis les *opérateurs* arithmétiques. Par exemple, au lieu d'écrire $3 \times (4 + 5)$, on écrira plutôt $3\ 4\ 5\ +\ \times$. On parle aussi de notation *postfixe*.

▶ **Question 1** Évaluer l'expression postfixe $3\ 14\ +\ 5\ 2\ -\ \times$.

▶ **Question 2** Convertir l'expression infixe (c'est-à-dire une expression arithmétique "normale") en notation polonaise inversée : $((2 \times (7 - 9)) + 3) \times 6$

Pour représenter une expression arithmétique postfixe, on va utiliser les types suivants :

OCaml
```
type operateur = Plus | Moins | Mult | Div

type expr_postfixe =
  | Val of int
  | Op of operateur
```

Une expression sera alors une liste de type `expr_postfixe list`.

▶ **Question 3** Représenter les exemples précédents par ce type.

Maintenant, l'objectif est d'évaluer une expression postfixe en utilisant une pile. On le fera en OCaml, en utilisant le module `Stack` implémentant les piles impératives en OCaml, notamment les fonctions `Stack.create` `Stack.is_empty` `Stack.push` `Stack.pop`. Ensuite, on lit l'expression dans l'ordre de la liste :

- Si on lit un opérande (`Val _`), on l'empile.
- Si on lit un opérateur, on dépile les deux dernières valeurs dans la pile, on leur applique l'opérateur et on empile le résultat.

À la fin du parcours de la liste, si elle est une expression postfixe correcte, la pile ne contient qu'une seule valeur qui est l'évaluation de cette expression.

▶ **Question 4** Implémenter cette stratégie en une fonction `evalue_postfixe : expr_element list -> int`. On lèvera une exception si la liste en entrée n'est pas une expression postfixe correctement formée.

On peut aussi évaluer une expression infixe (les expressions "normales") en la convertissant d'abord en une expression postfixe, puis en utilisant `evalue_postfixe`. On représentera les expressions infixes par des `expr_infixe list` avec :

OCaml
```
type expr_infixe =
  | Val2 of int
  | Par_ouvr
  | Par_ferm
  | Op2 of operateur
```

▶ **Question 5** Écrire une fonction vérifiant qu'une expression infixe est bien parenthésée.

La première étape (infixe vers postfixe) utilise aussi deux piles `p1` et `p2` et parcourt l'expression de gauche à droite en utilisant la stratégie suivante :

- si on lit un opérande, on l'empile dans `p2`,
- si on lit une parenthèse ouvrante, on l'empile dans `p1`,
- si on lit un opérateur, on l'empile dans `p1`,
- si on lit une parenthèse fermante, on dépile l'opérateur au sommet de `p1` puis la parenthèse ouvrante correspondante, et on empile cet opérateur dans `p2`
- si on est dans un autre cas, c'est une erreur

La pile `p2` contient alors l'expression dans l'ordre postfixe : il suffit alors de renvoyer la liste des éléments de la pile (convertie dans le type réservé à la notation postfixe).

▶ **Question 6** Implémenter cette stratégie, en supposant que l'expression infixe est complètement parenthésée (des parenthèses partout, même englobant l'expression

entière).

▶ **Question 7** Adapter cette stratégie dans le cas où l'expression n'est pas complètement parenthésée (et les opérateurs sont évalués avec l'ordre de priorité usuel en arithmétique).

Exercice 8

Vous êtes engagés par un hôpital pour remplacer le système manuel de tri des patients arrivant aux urgences par un système informatique.
Les patients arrivant aux urgences sont immédiatement évalués en fonction de la gravité de leur pathologie et de l'urgence de leur prise en charge. Ils sont notés : cette évaluation prend la forme d'une note allant entre 0 (les cas les moins urgents et les plus bénins) à $N-1$ (les cas extrêmes à prendre en charge en premier). Ensuite, ils vont dans une salle d'attente en attendant qu'un médecin les prenne en charge.
La méthode pour choisir le prochain patient en salle d'attente est la suivante : on prend d'abord le patient ayant la note la plus élevée. S'il y a plusieurs patients ayant la même note, on prend d'abord en charge le patient étant arrivé le plus tôt.

▶ **Question 1** Proposer un type abstrait permettant de représenter les patients en salle d'attente. On veut notamment avoir une opération permettant d'ajouter un patient dans la salle d'attente, et une opération permettant de choisir et de retirer le prochain patient à traiter. *On écrira une version fonctionnelle et une version impérative de ce type abstrait.*

▶ **Question 2** Implémenter ce type abstrait en utilisant des listes de couple `(score * id_patient) list`. Déterminer les complexités de chacune des opérations.

▶ **Question 3** Adapter ce type abstrait tel que la recherche du prochain patient soit en $\mathcal{O}(1)$. Quelle est la complexité des autres opérations ?

▶ **Question 4** Proposer une implémentation de ce type abstrait tel que l'ajout d'un patient se fasse en $\mathcal{O}(1)$ et le choix d'un nouveau patient en $\mathcal{O}(N)$. On permettra une complexité en $\mathcal{O}(N)$ pour l'initialisation de cette structure de données.