Le but du TP est de manipuler des piles et des files avec les modules `Stack` et `Queue` :

```
Stack.create: unit -> 'a Stack.t
Stack.is_empty: 'a Stack.t -> bool
Stack.push: 'a -> 'a Stack.t -> unit
Stack.pop: 'a Stack.t -> 'a
```

```
Queue.create: unit -> 'a Queue.t
Queue.is_empty: 'a Queue.t -> bool
Queue.push: 'a -> 'a Queue.t -> unit
Queue.pop: 'a Queue.t -> 'a
```

Exercice 1.

Récupérer le fichier annexe disponible sur la page du cours et prévoir la réponse d'OCaml lors de l'évaluation de chaque ligne du fichier.

Exercice 2.

1. Écrire une fonction de type « `'a array -> 'a Stack.t` » qui crée une pile contenant tous les éléments du tableau donné en entrée. On fera en sorte que l'élément se trouvant au sommet de la pile (c'est à dire l'élément renvoyé par `Stack.pop`) soit le dernier élément du tableau.
2. Écrire une fonction itérative « `rev_stack: 'a Stack.t -> unit` » qui inverse l'ordre des éléments d'une pile. Votre fonction doit modifier la pile et ne rien renvoyer.

Exercice 3. Nombres de Hamming

Un nombre de Hamming est un entier naturel de la forme $2^p 3^q 5^r$ où $p, q, r \in \mathbb{N}$. Soit $n \in \mathbb{N}$, on souhaite créer une tabeau T_n contenant les n premiers nombres de Hamming triés par ordre croissant. Par exemple T_{15} est le tableau :

```
[|1; 2; 3; 4; 5; 6; 8; 9; 10; 12; 15; 16; 18; 20; 24|]
```

À l'aide de files, écrire une fonction de complexité $\mathcal{O}(n)$ qui prend en entrée n et renvoie T_n. Vérifier que le $15^{\text{ème}}$ nombre de Hamming est 24 et que le $1111^{\text{ème}}$ est 102 515 625. On pourra utiliser la fonction `Queue.peek` qui renvoie le premier élément d'une file (sans le supprimer de la file, contrairement à `Queue.pop`).

Exercice 4. Expressions bien parenthésées

Soit `n` $\in \mathbb{N}$ et `ouvr`, `ferm` deux tableaux de taille `n` contenant des caractères. On suppose que les éléments de `ouvr` (resp. `ferm`) sont distincts deux à deux et que ces deux tableaux n'ont pas d'élément en commun. Par exemple :

```
let ouvr0 = [| '('; '['; '<'; '{' |];;
let ferm0 = [| ')'; ']'; '>'; '}' |];;
```

L'ensemble des expressions bien parenthésées noté $\mathcal{E}$ est le plus petit ensemble tel que :

(P_1) Si `s` est une chaîne de caractères ne contenant aucun élément de `ouvr` ou de `ferm`, alors `s` $\in \mathcal{E}$.

(P_2) Si `s` $\in \mathcal{E}$ et `t` $\in \mathcal{E}$, alors `s ^ t` $\in \mathcal{E}$ où `^` désigne l'opérateur de concaténation.

(P_3) Pour tout `s` $\in \mathcal{E}$ et tout `i` $\in [\![$`0, n - 1`$]\!]$, si on note :

- → $\texttt{s}_1$ la chaîne de caractères dont l'unique caractère est `ouvr.(i)`.
- → $\texttt{s}_2$ la chaîne de caractères dont l'unique caractère est `ferm.(i)`.

alors $\texttt{s}_1$ `^ s ^` $\texttt{s}_2 \in \mathcal{E}$.

Par exemple, d'après (P_1), les expressions suivantes sont bien parenthésées :

```
"let a = "    "1+2"    "*"    "3+4"    "-8"    " in "    "a; a+1; a+2"    ";;"
```

En utilisant (P_2) et (P_3) pour combiner les expressions ci-dessus, on obtient d'autres expressions bien parenthésées :

```
"(1+2)"      "(3+4)"      "[a; a+1; a+2]"      "(3+4)-8"      "((3+4)-8)"

"let a = (1+2)*((3+4)-8) in [a; a+1; a+2];;"
```

Cette expression est également bien parenthésée :

```
"let li = (if 0 = 1 then [[((1)+(1))]; [((1-1))]] else [[1]; [2]; [3]]) in [([li])];;"
```

En revanche ces expressions ne sont pas bien parenthésées :

```
"(1+2)*)4"      "5+(4*3-(2)"      "10*(7+5]"      "]1; 2; 3["
```

Étant donné un caractère `c` du tableau `ouvr` (resp. `ferm`), on veut pouvoir déterminer en temps constant l'élément associé à `c` dans `ferm` (resp. `ouvr`). Pour cela, on utilisera deux tables de hachage « `d_ouvr: corresp` » et « `d_ferm: corresp` » où :

```
type corresp = (char, char) Hashtbl.t;;
```

Par exemple, avec `ouvr` et `ferm` les deux tableaux donnés au début de l'énoncé :

```
# Syntaxe Python:
d_ouvr0 = { '(' : ')' , '[' : ']' , '<' : '>' , '{' : '}' }
d_ferm0 = { ')' : '(' , ']' : '[' , '>' : '<' , '}' : '{' }
```

1. Écrire une fonction

   ```
   make_corresp: char array -> char array -> (corresp * corresp)
   ```

   qui prend en entrée `ouvr` et `ferm`, et renvoie `d_ouvr` et `d_ferm`. On utilisera les fonctions du module `Hashtbl` décrites en cours.

2. À l'aide d'une pile, écrire une fonction

   ```
   est_bien_par: corresp -> corresp -> string -> bool
   ```

   qui prend en entrée `d_ouvr`, `d_ferm`, ainsi qu'une chaîne de caractères `s` et indique si `s` est bien parenthésée. Votre fonction devra être de complexité linéaire en la taille de `s`. Testez avec les exemples du début de l'énoncé.

Exercice 5. Files de priorité

Une file de priorité de type « `'a file_prio` » permet de stocker des éléments de type « `'a elem_fp` » composés d'une valeur de type `'a` et d'une priorité de type `int` :

```
type 'a elem_fp = {valeur: 'a; prio: int};;
```

C'est une structure des données dynamique et mutable dans laquelle les éléments sont extraits en commençant par celui ayant la priorité la plus élevée :

(a) Créer une file de priorité vide.

Priorité	
Valeur	

(b) Insérer `"DS Physique"` avec priorité `12`

Priorité	12
Valeur	`"DS Physique"`

(c) Insérer `"DM Maths"` avec priorité `10`

Priorité	12	10
Valeur	`"DS Physique"`	`"DM Maths"`

(d) Insérer `"Rendu ITC"` avec priorité `20`

Priorité	12	10	20
Valeur	`"DS Physique"`	`"DM Maths"`	`"Rendu ITC"`

(e) Extraction ⇝ renvoie `"Rendu ITC"` (élément de priorité maximale)

Priorité	12	10
Valeur	`"DS Physique"`	`"DM Maths"`

(f) Insérer `"Rendu OPT"` avec priorité `23`

Priorité	12	10	23
Valeur	`"DS Physique"`	`"DM Maths"`	`"Rendu OPT"`

(g) Modifier la priorité de `"DM Maths"` en `50`

Priorité	12	50	23
Valeur	`"DS Physique"`	`"DM Maths"`	`"Rendu OPT"`

(h) Extraction ⇝ renvoie `"DM Maths"`

Priorité	12	23
Valeur	`"DS Physique"`	`"Rendu OPT"`

On implémentera quatre opérations sur les files de priorité :

→ `create` qui crée une file de priorité vide.

→ « `is_empty: 'a file_prio -> bool` » qui teste si une file de priorité est vide.

→ « `insert: 'a file_prio -> 'a elem_fp -> unit` » qui ajoute un élément à une file de priorité.

→ « `pull: 'a file_prio -> 'a elem_fp` » qui supprime un élément de priorité maximale de la file et renvoie cet élément. Dans le cas où plusieurs éléments sont possibles, la fonction `pull` choisira arbitrairement l'élément à renvoyer parmi ceux de priorité maximale.

Implémentation 1. On souhaite implémenter les files de priorité à l'aide d'une liste qui sera maintenue triée par ordre décroissant des priorités : l'élément avec la plus grande priorité est au début de la liste et celui dont la priorité est minimale est à la fin de la liste.

1. Définir un type « `'a file_prio1` » pour représenter une file de priorité à l'aide d'une liste triée.
2. Écrire les fonctions `create_fp1`, `is_empty_fp1`, `insert_fp1` et `pull_fp1`. La fonction pour créer la file vide doit être de type « `unit -> 'a file_prio1` ».
3. Évaluer les complexités de ces fonctions.

Implémentation 2. On choisit d'implémenter les files de priorités en utilisant un tableau. Ce tableau sera maintenu trié par ordre croissant des priorités.

4. Définir un type « `'a file_prio2` » pour représenter une file de priorité à l'aide d'un tableau.

Étant donné qu'un tableau a une taille fixe, la fonction `create` doit prendre en argument la taille maximale de la file de priorité (qui est également la taille du tableau).

5. Écrire les fonctions « `create_fp2: int -> 'a file_prio2` » et `is_empty_fp2`. Si besoin, on pourra rajouter des arguments à la fonction `create_fp2`.

Pour ajouter un élément à la file de priorité, on l'insérera dans le tableau en décalant tous les éléments dont la priorité est supérieure. De plus, on calculera la position de l'élément dans le tableau à l'aide d'une recherche dichotomique.

6. Écrire les fonctions `insert_fp2` et `pull_fp2`.
7. Évaluer les complexités de ces fonctions.

Exercices à rendre au plus tard le 03/04/2025 à 20h

Exercice 6.

1. Sans utiliser de fonction auxiliaire, écrire une fonction récursive « `queue_of_list: 'a list -> 'a Queue.t` » qui crée une file contenant tous les éléments de la liste donnée en entrée. On fera en sorte que l'élément se trouvant au début de la file (c'est à dire l'élément renvoyé par `Queue.pop`) soit le dernier élément de la liste.
2. Sans utiliser de fonction auxiliaire, écrire une fonction récursive « `rev_queue: 'a Queue.t -> unit` » qui inverse l'ordre des éléments d'une file. Par exemple, la file `( 1 7 5 3 )` devient `( 3 5 7 1 )`.

Exercice 7. Nombres montants

Soit $k \in \mathbb{N}^*$. On dit que k est un ***nombre montant*** si la suite de ses chiffres est croissante. Par exemple, 115889 est un nombre montant car $1 \leqslant 1 \leqslant 5 \leqslant 8 \leqslant 8 \leqslant 9$.

Soit $n \in \mathbb{N}^*$. Notre objectif est d'afficher tous les nombres montants avec n chiffres. Par exemple pour $n = 3$, on souhaite afficher :

```
999 899 889 888 799 789 788 779 778 777 699 689 688 679 678 677 669 668 667 666 599 589 588 579
578 577 569 568 567 566 559 558 557 556 555 499 489 488 479 478 477 469 468 467 466 459 458 457
456 455 449 448 447 446 445 444 399 389 388 379 378 377 369 368 367 366 359 358 357 356 355 349
348 347 346 345 344 339 338 337 336 335 334 333 299 289 288 279 278 277 269 268 267 266 259 258
257 256 255 249 248 247 246 245 244 239 238 237 236 235 234 233 229 228 227 226 225 224 223 222
199 189 188 179 178 177 169 168 167 166 159 158 157 156 155 149 148 147 146 145 144 139 138 137
136 135 134 133 129 128 127 126 125 124 123 122 119 118 117 116 115 114 113 112 111
```

Pour cela, on propose d'utiliser une pile P de type « `(int * int) Stack.t` ». Cette pile contiendra des couples (k, m) où k est un nombre montant et m est le nombre de chiffres dans k :

- → Initialement, P contient tous les nombres montants avec un seul chiffre.
- → Tant que P n'est pas vide, on extrait l'un des éléments k de P. Si k contient n chiffres, on affiche k ; sinon on ajoute dans P tous les nombres montants obtenus en concaténant un nouveau chiffre à droite de k.

1. À l'aide de la procédure décrite ci-dessus, écrire une fonction « `nb_montants: int -> unit` » qui prend en entrée $n \in \mathbb{N}^*$ et affiche tous les nombres montants de n chiffres. Chaque nombre montant doit apparaître une et une seule fois, mais l'ordre d'apparition n'est pas important.

Exercice 8. Implémentation de dictionnaires (facultatif)

Dans cet exercice, on va implémenter la structure de dictionnaire à l'aide d'une table de hachage. Les dictionnaires contiendront des objets de type « `'a elem_dict` » composés d'une clé de type `int` et d'une valeur de type `'a` :

```
# type 'a elem_dict = {cle: int; valeur: 'a};;
```

On souhaite utiliser une ***table de hachage*** et gérer les collisions par ***chaînage*** (dont le principe a été décrit dans le cours). Les éléments du dictionnaire seront donc stockés dans un tableau de taille m. La case d'indice i du tableau contient la liste chaînée de tous les éléments dont la clé c vérifie $h(c) = i$. On utilisera la fonction de hachage :

$$h : c \mapsto \left\lfloor m \times c\frac{\sqrt{5}-1}{2} \right\rfloor \mod m$$

Remarque. Lorsqu'on ajoute un élément `e` dont la clé appartient déjà au dictionnaire, on se contente d'ajouter `e` dans la liste appropriée sans supprimer l'élément de même clé qui appartenait déjà au dictionnaire. Cela signifie que si on ajoute successivement deux éléments `e1` et `e2` de clé c (avec la fonction `add`), puis qu'on supprime l'élément de clé c (avec la fonction `remove`), alors le dictionnaire contient encore l'élément `e1`. Notez que les objets de type `Hashtbl` en OCaml fonctionnent de cette manière.

1. Définir un type « `'a dict` » et écrire les fonctions :
   - → « `create_dict: int -> 'a dict` » qui crée un dictionnaire vide. L'entier en argument est la taille du tableau à créer.
   - → « `add_dict: 'a dict -> 'a elem_dict -> unit` » qui ajoute un élément au dictionnaire.
   - → « `find_dict: 'a dict -> int -> 'a` » qui renvoie la valeur de l'élément dont la clé est donnée en argument.
   - → « `remove_dict: 'a dict -> int -> unit` » qui supprime l'élément dont la clé est donnée en argument.
2. Évaluer les complexités de vos fonctions.