Le but de ce TP est d'étudier deux implémentations possibles d'une structure de **file**, qui repose sur le principe "**FIFO**" (First In First Out) : premier arrivé, premier sorti. Ainsi, lorsqu'on insère un élément dans la file, celui-ci ne pourra être retiré qu'après que tous les éléments insérés avant l'aient été.

Les opérations à écrire pour implémenter une file sont les suivantes :

- **création** d'une file **vide** ;
- **test d'égalité** au vide ;
- **retrait** de l'élément en tête de file non vide ;
- **ajout** d'un élément en queue d'une file.

Puisque le C ne possède pas de polymorphisme, dans la suite, on travaillera avec des files stockant des valeurs de type `valeur_t`, où `valeur_t` est défini au préalable (par exemple comme étant le type `int`).

```c
typedef int valeur_t; // à modifier selon le type qu'on veut stocker dans la file
```

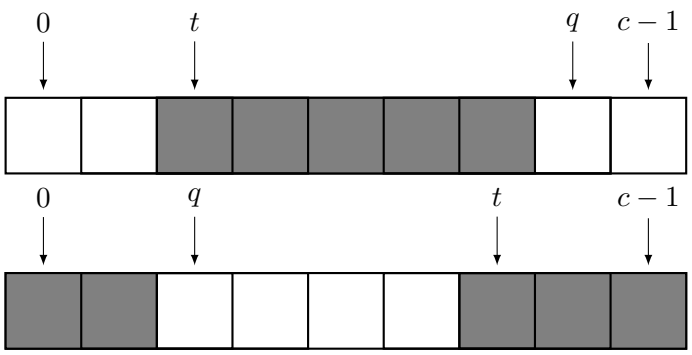
1. Télécharger le fichier `fenelon_tp09.zip` sur la page web du cours, et extraire tous les fichiers dans un dossier adéquat de votre ordinateur (cf. TP précédent).

1 Implémentation par un tableau circulaire

```c
struct file {
  valeur_t* contenu;
  int capacite;
  int nb;
  int tete;
  int queue;
};
typedef struct file file;
```

Dans cette partie, on se propose d'implémenter une file avec ce qu'on appelle un **tableau circulaire**, via le type ci-dessus, où, si `f` est une variable de type `file*`, on a :

- `f->contenu` est un tableau de longueur `f->capacite`, dont les cases contiennent des éléments de type `valeur_t` ;
- `f->nb` indique combien d'éléments sont stockés dans la file, (c'est à dire combien de cases de `f->contenu` sont réellement utilisées) ;
- `f->tete` contient l'indice de la case de `f->contenu` qui est en tête de la file ;
- `f->queue` contient l'indice de la case de `f->contenu` qui devra être utilisée pour rajouter un élément en queue de la file ;
- en abrégeant (t, q, c) les valeurs de (`f->tete`, `f->queue`, `f->capacite`), les éléments stockés dans la file sont, dans l'ordre, les éléments stockés dans les cases de `f->contenu` aux indices :
  - $\{t, t+1, t+2, \dots, q-1\}$ si $t < q$ (cf. le premier schéma ci-dessous) ;
  - $\{t, t+1, t+2, \dots c-1, 0, 1, \dots, q-1\}$ sinon (cf. le second schéma ci-dessous).



Dans cette partie et la suivante, on étudie des implémentations **mutables** d'un tel type **file**. Ainsi, une fois le type `file` créé, en C, on manipulera des pointeurs de type `file*`.

2. Ouvrir dans VSCode le fichier `file_tableau_circulaire.c`.
3. Écrire une fonction `file* creer_file(void);` créant une file initialement vide ayant les valeurs initiales suivantes : `f->capacite = 3` et `f->nb = f->tete = f->queue = 0`.
   On renverra un pointeur vers cette file.
4. Écrire une fonction `bool file_est_vide(file* f);` prenant en argument un pointeur vers une file et renvoyant `true` si elle est vide (et `false` sinon).
5. Écrire une fonction `valeur_t defiler(file* f);` prenant en argument un pointeur vers une file supposée non vide, et qui retire l'élément en tête de la file (et le renvoie).
   Si la file est vide, on pourra renvoyer une erreur à l'aide de la fonction `assert`.
6. Écrire une fonction `bool file_est_pleine(file* f);` prenant en argument un pointeur vers une file et renvoyant `true` si toutes les cases de `f->contenu` sont utilisées (et `false` sinon).

Si l'on souhaite rajouter un élément dans la file alors que le tableau `f->contenu` est plein, on va adopter la même stratégie que dans un TP précédent : on va créer un nouveau tableau de taille $2 \times n + 1$ (où n est la taille du tableau de départ), et on va recopier toutes les cases de l'ancien tableau dans le nouveau. On est alors ramener au cas où le tableau n'est pas plein.

7. Écrire une fonction `void agrandir(file* f);` qui effectue cette opération.
   On prendra garde à ne pas avoir de fuite mémoire, et à bien avoir les éléments stockés dans la file dans le même ordre qu'au départ.
8. Écrire une fonction `void enfiler(file* f, valeur_t x);` prenant en arguments un pointeur vers une file et une valeur x, et qui rajoute l'élément x en queue de la file.
9. Écrire une fonction `void detruire_file(file* f);` prenant en argument un pointeur vers une file et libérant toute la mémoire utilisée par cette file.
10. Tester votre programme via la commande suivante :

```
$ gcc -Wall -pedantic -Werror file_tableau_circulaire.c test.c -o test_partie1
$ ./test_partie1
Test 1 : OK
Test 2 : OK
Test 3 (doit renvoyer 12 42) : 12 42
Test 4 : OK
Test 5 (doit renvoyer 0 1 2 3 4) : 0 1 2 3 4
Test 7 (doit renvoyer 5 6 7 8 9) : 5 6 7 8 9
Test 8 (doit renvoyer 10 --> 20) : 10 11 12 13 14 15 16 17 18 19 20
Test 9 : OK
Test 10 : OK
```

11. Analyser la complexité de chacune de vos fonctions.

2 Implémentation par une liste doublement chainée

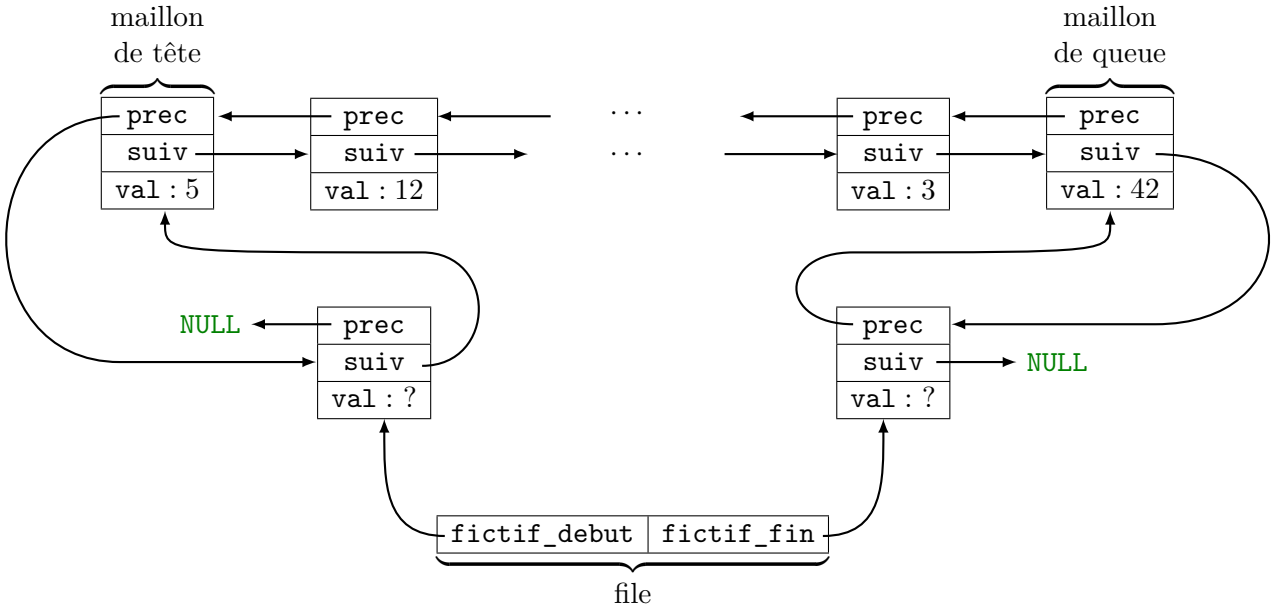
On peut également implémenter une structure de **file** à l'aide d'une généralisation des **listes chaînées** : les **listes doublement chaînées**. Dans une telle structure, chaque maillon possède :

- une **valeur** ;
- un pointeur vers le maillon **suivant** ;
- un pointeur vers le maillon **précédent**.

Il faudra également qu'on garde un **pointeur** vers le **début de la chaîne**, et un autre **pointeur** vers la **fin de la chaîne**. Pour cela, il sera plus pratique d'avoir **deux maillons fictifs** : un au **début** et un à la **fin**. Ces maillons ne contiendrons pas de "vraie" valeur de la file, mais on aura que :

- le champ "suivant" du maillon fictif du début pointe vers la tête ;
- le champ "précédent" du maillon fictif de fin pointe vers la queue.

On résume tout cela par le schéma ci-dessous :



Pour implémenter cette structure, on se donne les types C suivants :

```c
struct maillon {
  valeur_t val;
  maillon* suiv;
  maillon* prec;
};
typedef struct maillon maillon;

struct file {
  maillon* fictif_debut;
  maillon* fictif_fin;
};
typedef struct file file;
```

12. Ouvrir dans VSCode le fichier `file_liste_doublement_chainee.c`.
13. Écrire une fonction `file* creer_file(void);` créant une file initialement vide (les deux maillons fictifs pointent l'un sur l'autre).
    On renverra un pointeur vers cette file.
14. Écrire une fonction `bool file_est_vide(file* f);` prenant en argument un pointeur vers une file et renvoyant `true` si elle est vide (et `false` sinon).
15. Écrire une fonction `valeur_t defiler(file* f);` prenant en argument un pointeur vers une file supposée non vide, et qui retire l'élément en tête de la file (et le renvoie).
    On prendra garde à ne pas avoir de fuite mémoire !
    Si la file est vide, on pourra renvoyer une erreur à l'aide de la fonction `assert`.
16. Écrire une fonction `void enfiler(file* f, valeur_t x);` prenant en arguments un pointeur vers une file et une valeur x, et qui rajoute l'élément x en queue de la file (en créant un nouveau maillon).
17. Écrire une fonction `void detruire_file(file* f);` prenant en argument un pointeur vers une file et libérant toute la mémoire utilisée par cette file.
18. Ouvrir le fichier `test.c` et changer son entête pour qu'il utilise vos fonctions de la partie 2 :

```c
#include <stdio.h>
// #include "file_tableau_circulaire.h"
#include "file_liste_doublement_chainee.h"
```

19. Tester votre programme via la commande suivante :

```
$ gcc -Wall -pedantic -Werror file_liste_doublement_chainee.c test.c -o test_partie2
$ ./test_partie2
Test 1 : OK
Test 2 : OK
Test 3 (doit renvoyer 12 42) : 12 42
Test 4 : OK
Test 5 (doit renvoyer 0 1 2 3 4) : 0 1 2 3 4
Test 7 (doit renvoyer 5 6 7 8 9) : 5 6 7 8 9
Test 8 (doit renvoyer 10 --> 20) : 10 11 12 13 14 15 16 17 18 19 20
Test 9 : OK
Test 10 : OK
```

20. Analyser la complexité de chacune de vos fonctions.