Implémentation d'une file circulaire

Un tableau circulaire de taille n est un tableau qui contient n cellules dont la cellule d'index $n-1$ est aussi adjacente à la cellule d'index 0.

La figure 1 montre un exemple de tableau circulaire.

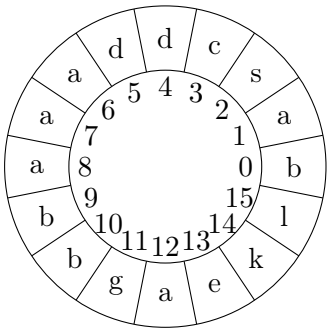

FIGURE 1 – un tableau circulaire

Dans un tableau circulaire, il n'y a pas de dernier élément, car toute cellule a toujours un successeur. En effet, le successeur de la cellule d'index $n-1$ est la cellule d'index 0.

Comme en informatique, il n'existe de pas de mémoire circulaire, les tableaux circulaires sont implémentés à l'aide de tableaux classiques. Ainsi, il n'y a pas de différence, en mémoire, entre un tableau classique et un tableau circulaire.

Par exemple, dans la figure 2, vous trouverez l'implémentation en mémoire du tableau circulaire de la figure 1.

	b	a	s	c	d	d	a	a	a	b	b	g	a	e	k	l	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

FIGURE 2 – l'implémentation en mémoire du tableau de la figure 1

L'objectif de cet exercice est d'implémenter un file circulaire. Une file circulaire est une file implémentée à l'intérieur d'un tableau circulaire.

Un file est un tableau contenant au moins une cellule. La première cellule est appelée la queue et la dernière cellule est appelé la tête.

La figure 3 montre 3 exemples de files.

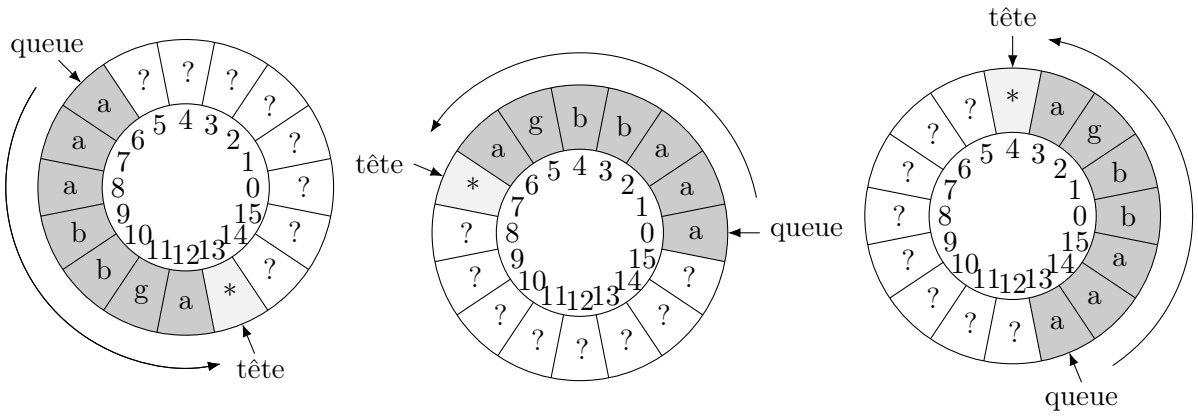

FIGURE 3 – La même file située à différents endroits dans le tableau circulaire

Les éléments d'une file sont les éléments contenus dans les cellules qui vont de la queue (inclue) à la tête (exclue).

En fait, la figure 3 montre différents états de la mémoire pour une même file contenant 8 cellules et les 7 même éléments. La différence de position de la file dépendent de l'historique d'utilisation de la file.

On dit que la file est pleine si la file utiliser tout les ces cellules du tableaux. Une file est vide si elle est réduite à sa tête (c'est à dire qu'elle ne contient qu'une cellule et aucun élément).

Par exemple, la figure 4 montre un exemple de file pleine (la file de gauche) et de file vide (la file de droite).

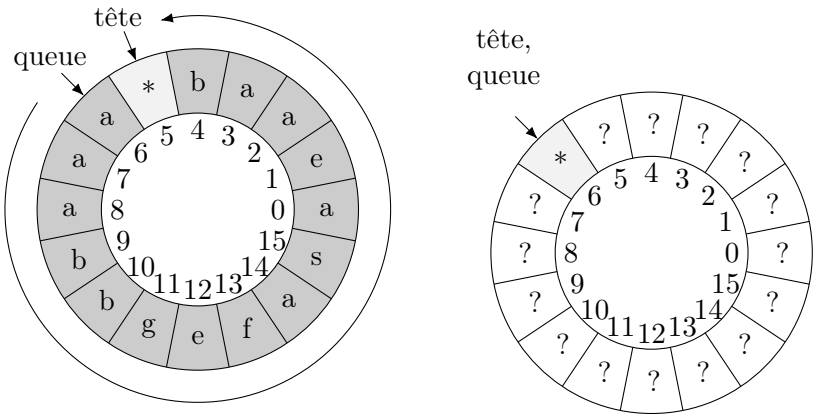

FIGURE 4 – Exemple de file pleine et vide

Dans un file deux opérations sont possibles : enfiler un élément et défiler un élément.

— Enfiler une élément :
Si la file n'est pas pleine, c'est à dire si la queue ne suit pas la tête, alors il est possible d'enfiler un élément en décalant la tête sur la cellule suivante.

La figure 5 montre l'ajout de l'élément r dans la file.

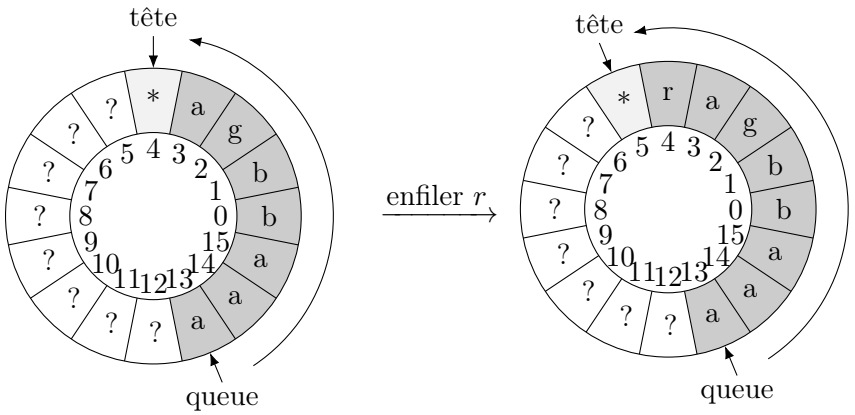

FIGURE 5 – Enfiler un élément r dans une filer

En fait, la tête de la file est utilisée comme une mémoire tampon servant à préparer une future donnée à insérer dans la file. C'est à dire que lorsque l'on souhaite ajouter un élément, on commence par copier cet élément dans la tête de la file avant de décaler le tête.

— Défiler un élément :
Si la file n'est pas vide, c'est à dire si la queue et la tête ne représente pas la même cellule, alors il est possible de défiler un élément en décalant la queue sur le cellule suivante.
La figure 6 montre la suppression d'un élément de la file.

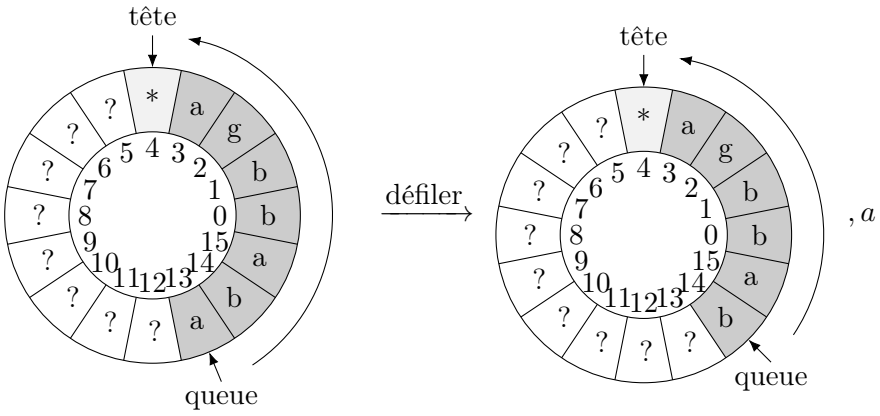

FIGURE 6 – Défiler un élément d'une file

Proposer une structure `file_t` et une implémentation de l'API suivante qui permet de manipuler des files circulaires.

```c
#include <stdlib.h>

struct file_t;
```

```c

struct file_t* allouer_file(size_t capacite, size_t taille_cellule);
  // allouer en mémoire une file pouvant contenir au plus `capacite`
  // éléments de `taille_cellule` octets chacun.
void desallouer_file(struct file_t* file);
  // libère tout la mémoire utilisée par la file `file`.

void* obtenir_tete(struct file_t* file);
  // Renvoie l'adresse du premier octet associé à la tête de la
  // file.
void enfiler_tete(struct file_t* file);
  // Si la file n'est pas pleine, ajoute la tête de la file parmi
  // les éléments de la file.
  // Si la file est pleine, ne fait rien.

void* obtenir_queue(struct file_t* file);
  // Renvoie l'adresse du premier octet associé à la queue de la
  // file.
void defiler(struct file_t* file);
  // Si la file n'est pas vide, retire la queue de la file.
  // Sinon, affiche un message d'erreur et arrête le programme.

int file_est_vide(struct file_t* file);
  // Renvoie vrai si la file est vide et faux sinon.
int file_est_pleine(struct file_t* file);
  // Renvoie vrai si la file est pleine et faux sinon.
```

On rappelle que `size_t` est le type d'un entier non signé défini par `stdlib.h`.

Votre implémentation devra permettre au programme suivant de fonctionner correctement :

```c
#include "file.h"
#include <stdlib.h>
#include <stdio.h>

typedef struct {
  float temps;
  float position;
} echantillon_t;

int main(){
  // On alloue la file
  unsigned int capacite_logs = 32;
  struct file_t* logs = allouer_file(
    capacite_logs, sizeof(echantillon_t)
  );
  // On ajoute un échantillon dans la file
  if( !file_est_pleine(logs) ){
    echantillon_t * tete = (echantillon_t*) obtenir_tete(logs);
```

```
19      tete->temps = 12.03; tete->position = 1.4;
20      enfiler_tete(logs);
21    }
22
23    // On affiche tous les échantillons de la file
24    echantillon_t * queue;
25    while( !file_est_vide(logs) ){
26      queue = (echantillon_t*) obtenir_queue(logs);
27      fprintf(stdout, "t=%f, p=%f \n", queue->temps, queue->position);
28      defiler(logs);
29    }
30
31    // On désalloue la file
32    desallouer_file(logs);
33    exit(EXIT_SUCCESS);
34  }
```

Correction

```c
#include "file.h"
#include <stdlib.h>
#include <stdio.h>

struct file_t {
  char* tableau;
  unsigned int queue;
  unsigned int tete;
  size_t taille_cellule;
  size_t capacite;
};

struct file_t* allouer_file(size_t capacite, size_t taille_cellule){
  struct file_t * res = (struct file_t *) malloc(
    sizeof(struct file_t)
  );
  if( res == NULL ){
    fprintf( stderr, "Not enough memory !\n" );
    exit(EXIT_FAILURE);
  }
  res->tableau = (char*) malloc(capacite * taille_cellule);
  if( res->tableau == NULL ){
    fprintf( stderr, "Not enough memory !\n" );
    exit(EXIT_FAILURE);
  }
  res->capacite = capacite;
  res->taille_cellule = taille_cellule;
  res->queue = 0;
  res->tete = 0;
  return res;
}

void desallouer_file(struct file_t* file){
  if( file == NULL ) return;
  if( file->tableau != NULL ){
    free(file->tableau);
    file->tableau = NULL;
  }
  free(file);
}

unsigned int next_cell_id(
  struct file_t* file, unsigned int cell_id
){
  return (cell_id+1)%file->capacite;
```

```
46 }
47
48 void* obtenir_tete(struct file_t* file){
49   return file->tableau + file->tete * file->taille_cellule;
50 };
51 void enfiler_tete(struct file_t* file){
52   if( file_est_pleine(file) ) return;
53   file->tete = next_cell_id( file, file->tete );
54 }
55
56 void* obtenir_queue(struct file_t* file){
57   return file->tableau + file->queue * file->taille_cellule;
58 };
59
60 void defiler(struct file_t* file){
61   if( file_est_vide(file) ){
62     fprintf(
63       stderr,
64       "Impossible de retirer un élément d'une file vide \n"
65     );
66     exit(EXIT_FAILURE);
67   }
68   file->queue = next_cell_id( file, file->queue );
69 }
70
71 int file_est_vide(struct file_t* file){
72   return file->tete == file->queue;
73 }
74
75 int file_est_pleine(struct file_t* file){
76   return next_cell_id( file, file->tete) == file->queue;
77 }
```