UFR de **mathématique** et d'**informatique**
Université de Strasbourg

Programmation de type fonctionnelle et de type mutation

Le but de cette séance de Travaux Pratiques est d'implémenter une structure de file FIFO (*First In First Out*) sous forme fonctionnelle et par mutation. Vous trouverez sur Moodle une archive contenant un certain nombre de fichiers pour vous aider à démarrer. Après l'avoir téléchargée :

```
$ tar -xf sda1-tp1.tar.gz
$ cd sda1-tp1
```

Le fichier `Makefile` contient les règles nécessaires pour compiler deux programmes interactifs intitulés `fonctionnel` et `mutation`. Par exemple :

```
$ make
$ ./fonctionnel
```

Les programmes `fonctionnel` et `mutation` se comportent de la même façon. Le programme crée une file vide puis attend des instructions de la forme :

- `aide` : Affiche une aide.
- `ajoute <x>` : Ajoute l'entier x à la file.
- `suppr` : Affiche l'entier en tête de file et le supprime de la file.
- `etat` : Affiche "vide" ou "non-vide" selon l'état de la file, et sa taille si elle n'est pas vide.
- `affiche` : Affiche le contenu de la file.
- `quit` : Détruit la file et termine le programme.

Pour que ces 2 programmes fonctionnent correctement, vous devrez modifier les fichiers `files_fct.h`, `files_fct.c`, `files_mut.h` et `files_mut.c`. Certaines fonctions sont déjà documentées ou implémentées, à vous de modifier ou compléter les documentations et implémentations des autres fonctions.

Une fois que ces programmes fonctionnent, vous pouvez les tester à la main ou bien utiliser les jeux de tests fournis dans le dossier `tests/`.

```
$ ./fonctionnel <tests/jeutest1.txt
$ ./mutation tests/jeutest1.txt sortie
```

Notez que les programmes `fonctionnel` et `mutation` acceptent un nombre variable d'arguments sur la ligne de commande :

- 0 : les commandes sont lues depuis l'entrée standard
- 1 : les commandes sont lues depuis le fichier fourni
- 2 : les commandes sont lues depuis le premier fichier fourni et les résultats sont écrits dans le deuxième fichier fourni.

Astuces :

- Pensez à utiliser la commande `diff(1)` pour comparer vos résultats aux résultats attendus (voir le répertoire `resultats/`).
- Le programme `rlwrap(1)` peut faciliter l'utilisation des programmes `fonctionnel` et `mutation` en mode interactif (quand aucun argument n'est fourni) :

  ```
  $ rlwrap ./fonctionnel
  $ rlwrap ./mutation
  ```

1 Implémentation « fonctionnelle »

Voici la structure proposée pour implémenter les files fonctionnelles (voir le fichier `files_fct.h` :

```
struct file {
  /** \brief Tableau d'éléments de type S. */
  S elements[TAILLE_MAX];
  /** \brief Position du premier élément dans le tableau file::x. */
  size_t debut;
  /** \brief Nombre d'éléments dans la file. */
  size_t taille;
};
typedef struct file;
```

Le tableau `elements` est utilisé de manière circulaire : le premier élément de la file se trouve à la position indiquée dans le champ `debut` tandis que le champ `taille` contient le nombre d'éléments dans la file. La figure 1 en présente un exemple.

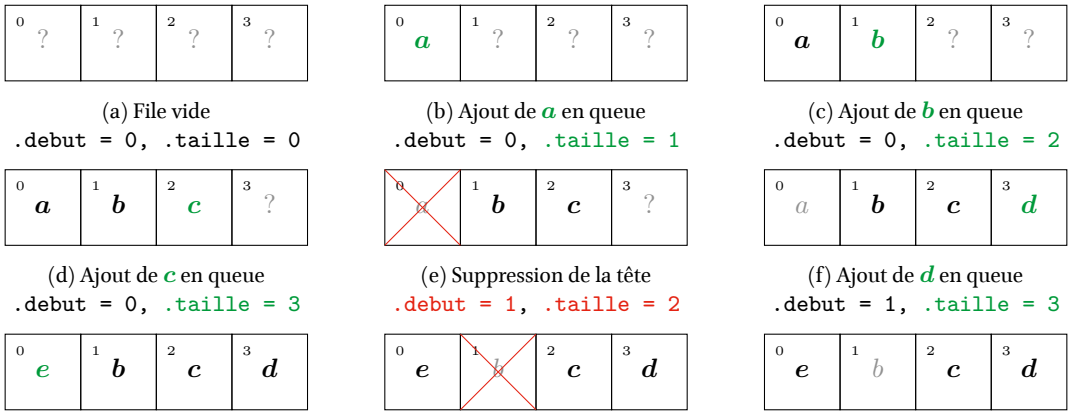

FIGURE 1 – Exemple de file circulaire pour `TAILLE_MAX = 4`

Il vous faudra implémenter les fonctions suivantes dans le fichier `files_fct.c` :

```
file file_nouv();              /* Nouvelle file vide. */
file file_adjq(file f, S x);   /* Ajoute un élément x en queue. */
file file_supt(file f);        /* Supprime l'élément en tête. */
S file_tete(file f);           /* Retourne l'élément en tête. */
bool file_estvide(file f);     /* Teste si une file est vide. */
size_t file_taille(file f);    /* Retourne le nombre d'éléments. */
S file_ieme(file f, size_t i); /* Retourne le i^ième élément. */
void file_detruit(file f);     /* Détruit (inutile pour l'instant). */
```

Lors de l'ajout d'un élément en queue, il faudra calculer la bonne position en fonction de `TAILLE_MAX` et des valeurs des champs `debut` et `taille`. De même, lors de la suppression de l'élément en tête il faudra veiller à prendre en compte le fait qu'on utilise le tableau de manière circulaire lors de la modification de la valeur du champ `debut`.

1. Modifier ou compléter les fonctions dans `files_fct.c`.
2. Modifier ou compléter la documentation dans `files_fct.h`.
3. Vérifier le bon fonctionnement du programme `fonctionnel`.

2 Implémentation par mutation

Le but de cet exercice est de remplacer les files avec programmation de type fonctionnelle par des files avec programmation de type mutation à apparence fonctionnelle dans les fichiers `files_mut.h` et `files_mut.c`.

1. Modifier ou compléter les fonctions dans `files_mut.c`.
2. Modifier ou compléter la documentation dans `files_mut.h`.
3. Vérifier le bon fonctionnement du programme `mutation` (attention aux fuites mémoires).

3 Comparaison

Changez la taille maximale des files et comparez les temps d'éxécution de des programmes `fonctionnel` et `mutation` sur le jeu de test énorme :

```
$ make clean
$ make CPPFLAGS='-DTAILLE_MAX=100000'
$ time ./fonctionnel tests/jeutest6_enorme.txt >/dev/null
$ time ./mutation tests/jeutest6_enorme.txt >/dev/null
```

Constatez-vous une différence ?

A Divers

A.1 Minimum

Les programmes de tests fournis nécessitent *au minimum* que les fonctions suivantes aient été complétées :

— `file_taille()`

— `file_ieme()`

A.2 Lenteurs

La production du fichier `tests/jeutest6_enorme.txt` risque de poser problème sur un système de fichiers distribué. Dans ce cas, remplacez le chemin par un chemin qui ne se trouve pas dans un système distribué comme par exemple le chemin `/tmp/$USER-jeutest6_enorme.txt` (la plupart des systèmes d'exploitation modernes placent le répertoire `/tmp` dans la RAM).

B Fuites mémoires

Pour rappel ou information, vous pouvez vérifier l'absence de fuites mémoires à l'aide d'outils tels que `valgrind(1)` (voir Section B.1) ou bien en activant des options d'instrumentation (voir Section B.2).

B.1 Valgrind

Compilez vos programmes avec les informations de *debugging*, puis éxécutez `valgrind(1)` en passant en paramètre le programme cible et ses options :

```
$ make -B CFLAGS="-Og -ggdb"
$ valgrind ./chaine <tests/jeutest1.txt
$ valgrind --leak-check=full ./chaine <tests/jeutest1.txt
```

B.2 Instrumentation

Compilez vos programmes en ajoutant l'option d'instrumentation `-fsanitize=address` puis éxécutez votre programme de la manière classique :

```
$ make -B CFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address"
$ ./chaine <tests/jeutest1.txt
```