C avancé

Files

07 avril 2022

Version 7

Fabrice BOISSIER <fabrice.boissier@epita.fr>
Mark ANGOUSTURES <mark.angoustures@epita.fr>

Copyright

Table des matières

1 Consignes Générales

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

Consigne Générale 0 : Vous devez lire le sujet.
Consigne Générale 1 : Vous devez respecter les consignes.
Consigne Générale 2 : Vous devez rendre le travail dans les délais prévus.

Consigne Générale 3 : Le travail doit être rendu dans le format décrit à la section Format de Rendu.

Consigne Générale 4 : Le travail rendu ne doit pas contenir de fichiers binaires, temporaires, ou d'erreurs (`*~`, `*.o`, `*.a`, `*.so`, `*#*`, `*core`, `*.log`, `*.exe`, binaires, ...).

Consigne Générale 5 : Dans l'ensemble de ce document, la casse (caractères majuscules et minuscules) est très importante. Vous devez strictement respecter les majuscules et minuscules imposées dans les messages et noms de fichiers du sujet.

Consigne Générale 6 : Dans l'ensemble de ce document, `login.x` correspond à votre login.

Consigne Générale 7 : Dans l'ensemble de ce document, `nom1-nom2` correspond à la combinaison des deux noms de votre binôme (par exemple pour Fabrice BOISSIER et Mark ANGOUSTURES, cela donnera `boissier-angoustures`).

Consigne Générale 8 : Dans l'ensemble de ce document, le caractère ␣ correspond à une espace (s'il vous est demandé d'afficher ␣␣␣, vous devez afficher trois espaces consécutives).

Consigne Générale 9 : Tout retard, même d'une seconde, entraîne la note non négociable de 0.

Consigne Générale 10 : La triche (échange de code, copie de code ou de texte, ...) entraîne **au mieux** la note non négociable de 0.

Consigne Générale 11 : En cas de problème avec le projet, vous devez contacter le plus tôt possible les responsables du sujet aux adresses mail indiquées.

Conseil : N'attendez pas la dernière minute pour commencer à travailler sur le sujet.

2 Format de Rendu

Responsable(s) du projet :	**Fabrice BOISSIER** `<fabrice.boissier@epita.fr>`
Balise(s) du projet :	**`[CAV][TP2]`**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Envoi par mail
Nom du répertoire :	`login.x-TP2`
Nom de l'archive :	`login.x-TP2.tar.bz2`
Date maximale de rendu :	21/04/2022 23h42
Durée du projet :	2 semaines
Architecture/OS :	Linux - Ubuntu (x86_64)
Langage(s) :	C
Compilateur/Interpréteur :	**`/usr/bin/gcc`**
Options du compilateur/interpréteur :	**`-W -Wall -Werror -std=c99 -pedantic`**

Les fichiers suivants sont requis :

`AUTHORS`	contient le(s) nom(s) et prénom(s) de(s) auteur(s).
`Makefile`	le Makefile principal.
`README`	contient la description du projet et des exercices, ainsi que la façon d'utiliser le projet.

Un fichier **`Makefile`** doit être présent à la racine du dossier, et doit obligatoirement proposer ces règles :

`all`	*[Première règle]* lance la règle `libmyqueue`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance le(s) script(s) de test.
`libmyqueue`	lance les règles `shared` et `static`
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

Votre code sera testé automatiquement, vous devez donc scrupuleusement respecter les spécifications pour pouvoir obtenir des points en validant les exercices. Votre code sera testé en générant un exécutable ou des bibliothèques avec les commandes suivantes :

```
./configure
make
```

Suite à cette étape de génération, les exécutables ou bibliothèques doivent être placés à ces endroits :

login.x-TP2/libmyqueue.a
login.x-TP2/libmyqueue.so

L'arborescence attendue pour le projet est la suivante :

login.x-TP2/
login.x-TP2/AUTHORS
login.x-TP2/README
login.x-TP2/Makefile
login.x-TP2/configure
login.x-TP2/check/
login.x-TP2/check/check.sh
login.x-TP2/src/
login.x-TP2/src/queue_array.c
login.x-TP2/src/queue_array.h
login.x-TP2/src/queue_linked_list.c
login.x-TP2/src/queue_linked_list.h
login.x-TP2/src/queue_static.c
login.x-TP2/src/queue_static.h

Vous ne serez jamais pénalisés pour la présence de makefiles ou de fichiers sources (code et/ou headers) dans les différents dossiers du projet tant que leur existence peut être justifiée (des makefiles vides ou jamais utilisés sont pénalisés).
Vous ne serez jamais pénalisés pour la présence de fichiers de différentes natures dans le dossier `check` tant que leur existence peut être justifiée (des fichiers de test jamais utilisés sont pénalisés).

3 Aide Mémoire

Le travail doit être rendu au format ***.tar.bz2***, c'est-à-dire une archive **bz2** compressée avec un outil adapté (voir **`man 1 tar`** et **`man 1 bz2`**).
Tout autre format d'archive (zip, rar, 7zip, gz, gzip, ...) ne sera pas pris en compte, et votre travail ne sera pas corrigé (entraînant la note de 0).

Pour générer une archive *tar* en y mettant les dossiers *folder1* et *folder2*, vous devez taper :

```
tar cvf MyTarball.tar folder1 folder2
```

Pour générer une archive *tar* et la compresser avec GZip, vous devez taper :

```
tar cvzf MyTarball.tar.gz folder1 folder2
```

Pour générer une archive *tar* et la compresser avec BZip2, vous devez taper :

```
tar cvjf MyTarball.tar.bz2 folder1 folder2
```

Pour lister le contenu d'une archive *tar*, vous devez taper :

```
tar tf MyTarball.tar.bz2
```

Pour extraire le contenu d'une archive *tar*, vous devez taper :

```
tar xvf MyTarball.tar.bz2
```

Pour générer des exécutables avec les symboles de debug, vous devez utiliser les flags **`-g -ggdb`** avec le compilateur. N'oubliez pas d'appliquer ces flags sur *l'ensemble* des fichiers sources transformés en fichiers objets, et d'éventuellement utiliser les bibliothèques compilées en mode debug.

```
gcc -g -ggdb -c file1.c file2.c
```

Pour produire des exécutables avec les symboles de debug, il est conseillé de fournir un script **`configure`** prenant en paramètre une option permettant d'ajouter ces flags aux **`CFLAGS`** habituels.

```
./configure
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic
./configure debug
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic -g -ggdb
```

Pour produire une bibliothèque statique *libtest.a* à partir des fichiers *test1.c* et *file.c*, vous devez taper :

```
cc -c test1.c file.c
ar cr libtest.a test1.o file.o
```

Pour produire une bibliothèque dynamique *libtest.so* à partir des fichiers *test1.c* et *file.c*, vous devez taper (pensez aussi à **-fpic** ou **-fPIC**) :

```
cc -c test1.c file.c
cc test1.o file.o -shared -o libtest.so
```

Pour compiler un fichier en utilisant une bibliothèque dont le **.h** se trouve dans un dossier spécifique, par exemple la *libxml2*, vous devez taper (pensez à vous assurer que les *includes* de la bibliothèque ont été entourés de chevrons **< >**) :

```
cc -c -I/usr/include test1.c
```

Pour lier plusieurs fichiers objets ensemble et avec une bibliothèque, par exemple la *libxml2*, vous devez d'abord indiquer dans quel dossier trouver la bibliothèque avec l'option **-L**, puis indiquer quelle bibliothèque utiliser avec l'option **-l** (n'oubliez pas de retirer le préfixe *lib* au nom de la bibliothèque, et surtout, que l'ordre des fichiers objets et bibliothèques est important) :

```
cc -L/usr/lib test1.o -lxml2 file.o -o executable.exe
```

Si la bibliothèque existe en version *dynamique* (**.so**) et en version *statique* (**.a**) dans le système, l'éditeur de lien choisira en priorité la version *dynamique*. Pour forcer la version *statique*, vous devez l'indiquer dans la ligne de commande avec l'option **-static** :

```
cc -static -L/usr/lib test1.o -lxml2 file.o -o executable.exe
```

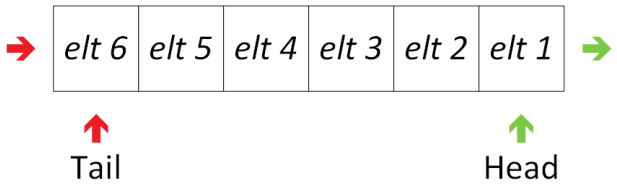
Dans ce sujet précis, vous ferez du code en C et des appels à des scripts shell qui afficheront les résultats dans le terminal (donc des flux de sortie qui pourront être redirigés vers un fichier texte).

4 Cours

Notions étudiées : Tableaux, Pointeurs, Files

4.1 Rappel sur les files

Les **files**, ou **queues** en anglais, sont des structures visant à stocker et rendre les données dans l'ordre d'arrivée. Une file dispose donc d'une *tête* contenant l'élément le plus ancien (inséré avant tous les autres), et une *queue* contenant l'élément inséré le plus récemment. Ces structures sont aussi appelées **FIFO** (*First In First Out*).

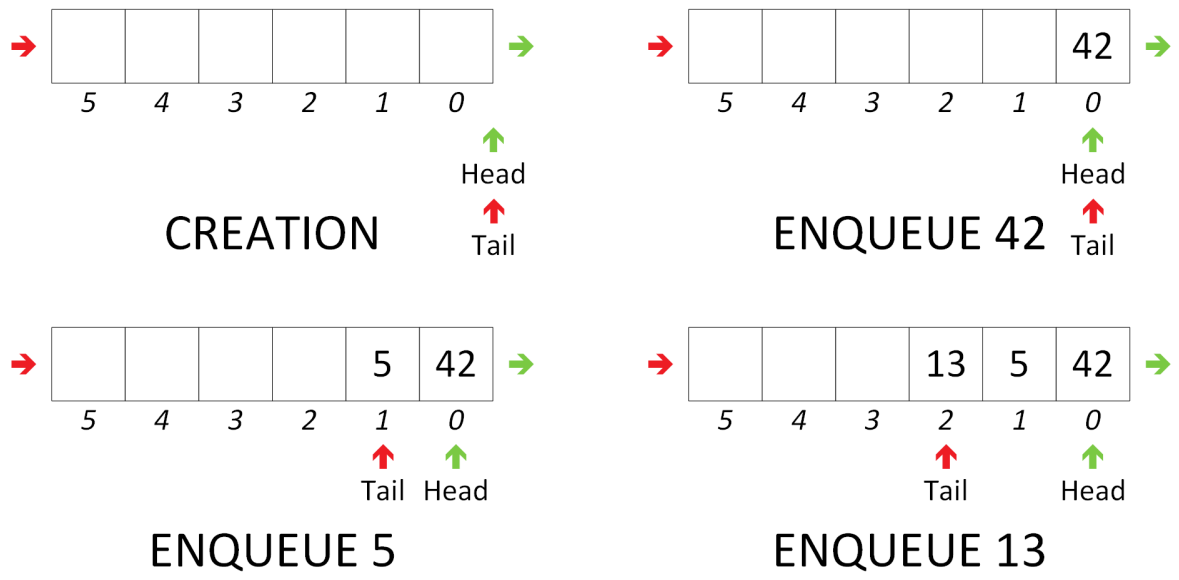


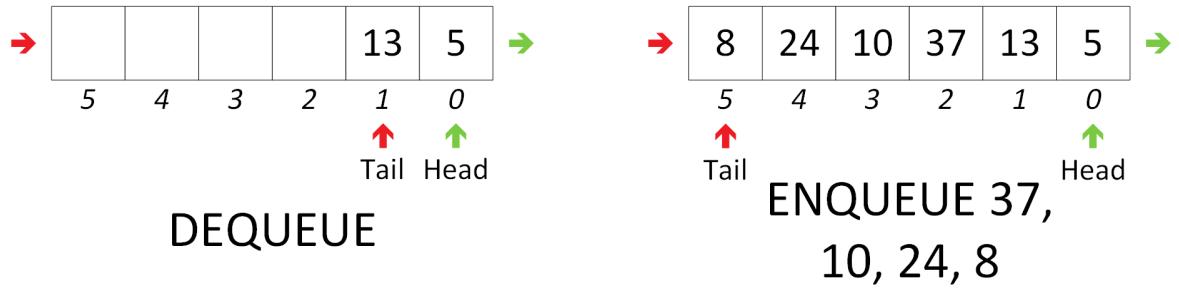
Deux opérations permettent d'utiliser une file :

- **`ENQUEUE`** : permettant d'*enfiler* une donnée supplémentaire dans la file
- **`DEQUEUE`** : permettant de *défiler* une donnée depuis la file

On ajoute donc une donnée en l'enfilant avec un **`ENQUEUE`**, celle-ci se retrouve en *queue* de file, c'est-à-dire au fond de la file. On récupère une donnée en défilant avec un **`DEQUEUE`**, celle-ci se trouvait en *tête* de file, c'est-à-dire qu'elle attendait son tour depuis son insertion. On accède donc aux éléments dans l'ordre d'arrivée.

Voici un exemple où l'on crée une file, puis on enfile successivement 42, 5, et 13, puis, on défile une fois (pour récupérer 42), et enfin, on enfile successivement 37, 10, 24, et 8.



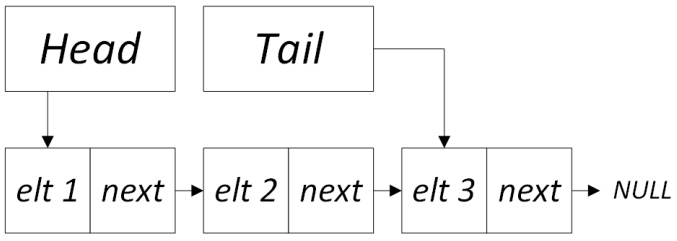


Les files, et surtout le respect de l'ordre d'arrivée des objets, sont couramment utilisés : mise en attente de personnes face à des guichets (voitures à un péage, clients face à une caisse, etc).
En informatique, on utilisera les files pour stocker temporairement et traiter les requêtes dans leur ordre d'arrivée. Dans le cas des *schedulers* (ordonnanceurs) visant à déterminer quel processus exécuter sur le cœur d'un processus, on ajoute parfois une priorité à chaque objet de la file. Ceci implique de mettre à jour l'ordre des objets dans la file lors de certains évènements (par exemple lorsque l'on enfile ou défile un élément, ou après un certain temps).

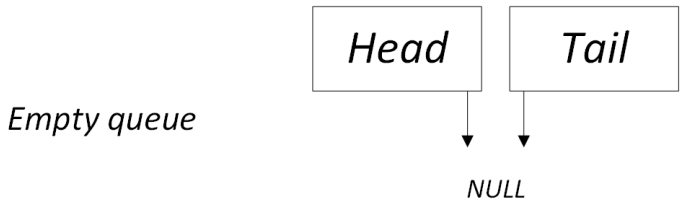
Afin d'implémenter une file, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et deux indicateurs pour l'élément tête de file et l'élément en queue de file. Nous allons maintenant voir comment implémenter une file avec des listes chaînées et un tableau de taille fixe.

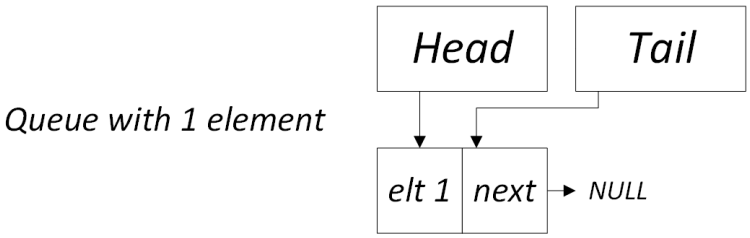
4.2 Files : implémentation avec des listes chaînées

Une implémentation à l'aide d'une liste chaînée permet d'exploiter la mémoire et d'être donc beaucoup plus flexible en terme de nombre maximum d'éléments. Le schéma suivant illustre une file sous forme de liste chaînée en mémoire :

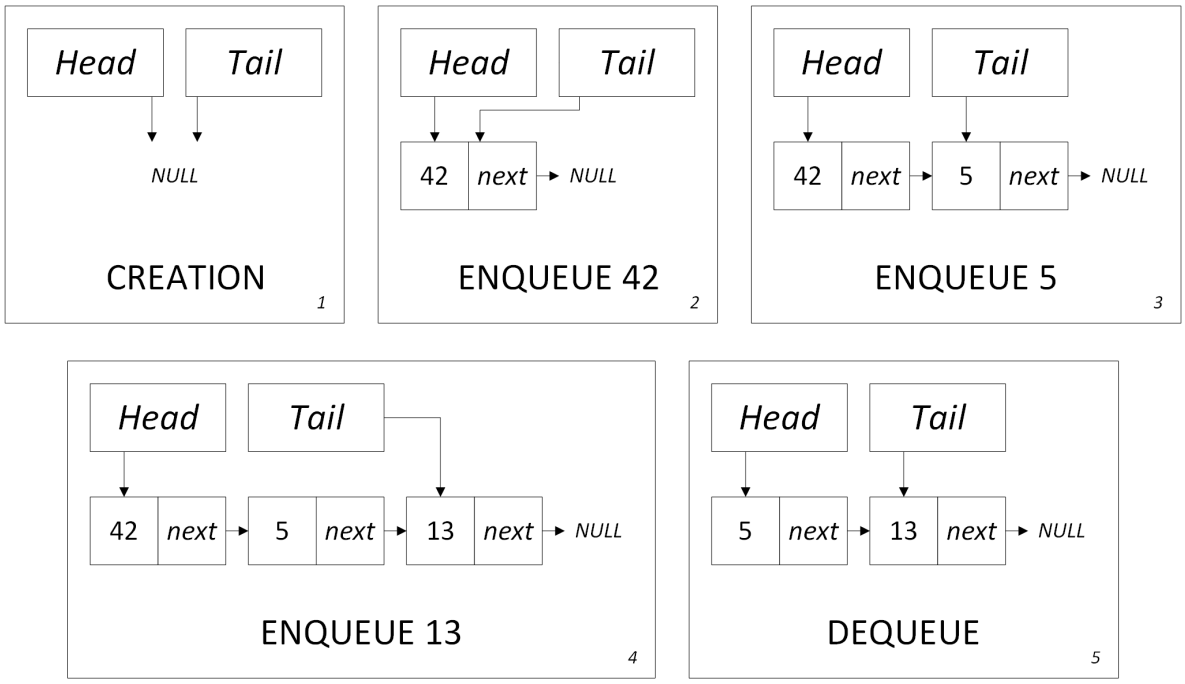


On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant), ainsi que deux pointeurs indiquant respectivement la tête de la file (*head*) et la queue de la file (*tail*).
Deux cas particuliers concernent les files où le pointeur de tête et le pointeur de queue valent la même chose : la file vide où les pointeurs contiennent **`NULL`**, et la file contenant un seul élément vers lequel les deux pointeurs renvoient. Une file nouvellement créée se trouve dans l'état vide.





L'exemple suivant montre l'évolution d'une file au fur et à mesure des insertions (enfiler / **ENQUEUE**) et suppressions (défiler / **DEQUEUE**).



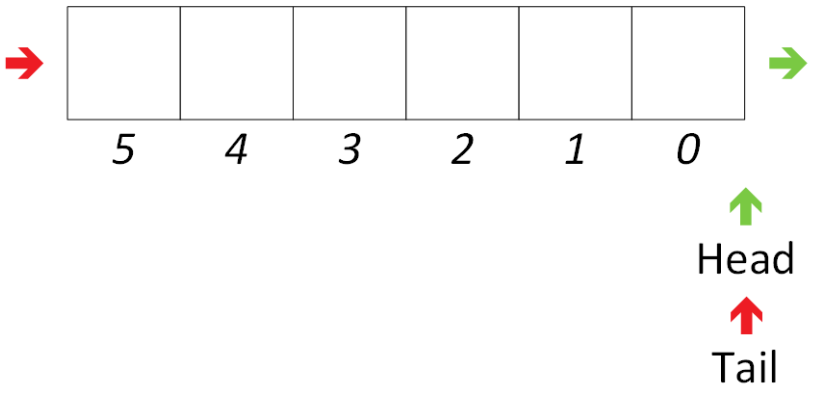
Les principales opérations se résument ainsi :

- Création : on alloue en mémoire la structure générale de la file, et on fixe les tête et queue de la file à **NULL**.
- Enfiler : on alloue en mémoire un nouvel élément, on met son pointeur *next* à **NULL**, puis, si la file est vide, on met les pointeurs de tête et de queue sur le nouvel élément, sinon, on met l'adresse du nouvel élément sur le pointeur *next* de l'élément pointé par la queue, et on met à jour le pointeur de queue sur le nouvel élément.
- Défiler : si la file est vide, on retourne une erreur, sinon, on récupère tout d'abord l'adresse de l'élément suivant celui en tête, puis, on libère l'élément en tête, puis, on met à jour le pointeur de tête de la file vers l'adresse de l'élément suivant. Si l'élément suivant est **NULL**, on met à jour le pointeur de queue également.

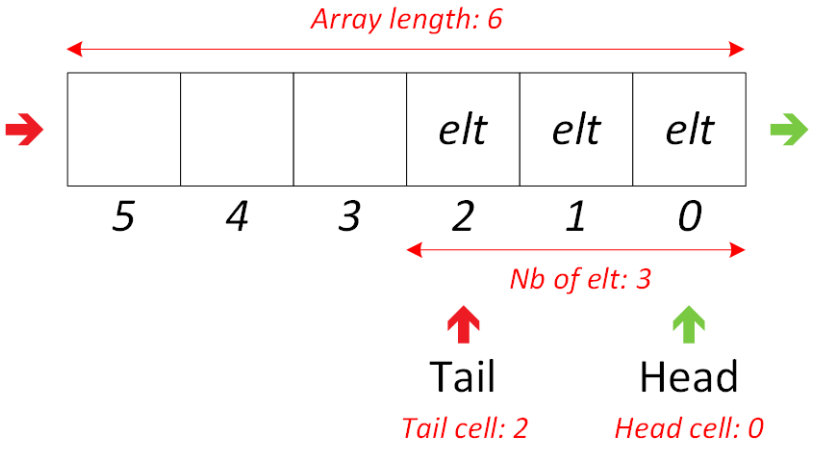
— Vider : on défile successivement tous les éléments jusqu'à obtenir la tête à **NULL** (ne pas oublier que l'opération qui défile met à jour la queue dans le cas **NULL**).
— Tête : on renvoie le contenu de l'élément en tête de file (le prochain élément qui sera défilé).
— Queue : on renvoie le contenu de l'élément en queue de file (le dernier élément qui sera défilé).

4.3 Files : implémentation avec un tableau de taille fixe

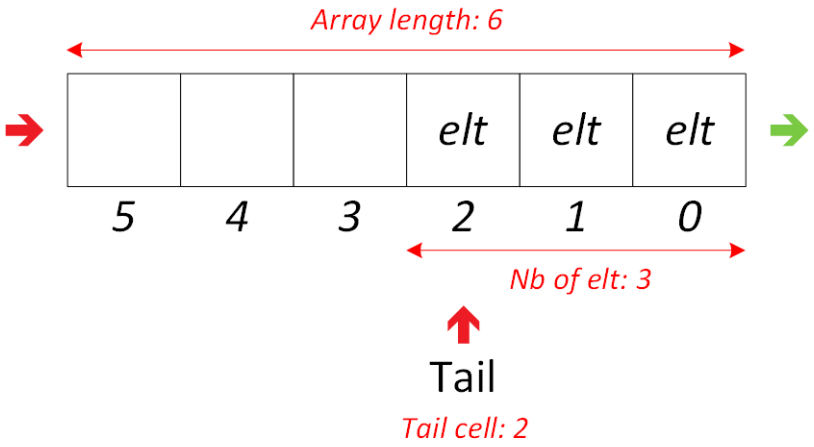
Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la file aura une taille maximale, et on peut refuser l'ajout d'un élément si la file est déjà pleine. La structure diffère également du fait que le tableau est alloué une seule fois lors de sa création (voire même lors de la compilation dans le cas statique).
Le schéma suivant présente la structure générale :



On notera cette fois que plusieurs informations distinctes doivent être conservées : l'adresse du tableau, le numéro de case correspondant à la tête de la file (*head*), le numéro de case correspondant à la queue de la file (*tail*), la taille du tableau (le nombre maximum d'objets pouvant être stockés), le nombre d'éléments dans le tableau.
Le schéma suivant détaille certaines informations de façon plus explicite :



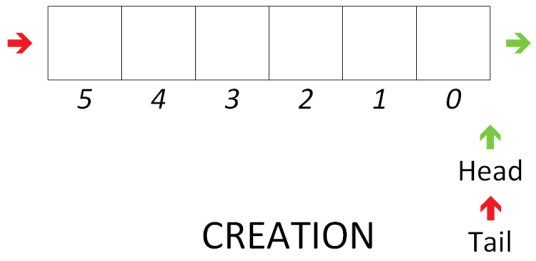
Une façon d'implémenter le tableau permet de supposer que la case 0 contiendra toujours l'élément de tête. Dans ce cas très précis, on peut donc se passer de la variable de tête, et au contraire, s'appuyer sur la variable donnant le nombre d'éléments dans le tableau pour savoir si la file est vide ou non. Cette implémentation ressemble donc à cela :



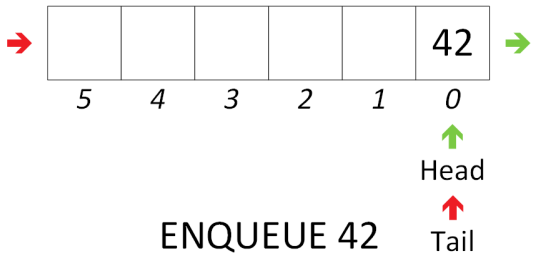
Dans le cas d'un tableau de taille fixe, les pointeurs de tête et de queue ne peuvent pas utiliser la valeur **`NULL`** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que la queue est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer les cases où se trouvent la tête et la queue de la file, ainsi que le cas vide :

- On enregistre dans la structure de la file une variable servant à compter le nombre d'éléments présents (la queue peut donc prendre n'importe quelle valeur tant que la file est vide).
- On utilise un entier relatif pour indiquer la position, et -1 indique que la file est vide (l'ajout d'un élément décalera la tête et la queue à 0, c'est-à-dire la case où sera l'élément).
- On place la queue de la file sur la première case non utilisée, et l'accès au dernier élément se fait donc en retirant 1 au pointeur de sommet (ainsi, une queue à la case 0 indique que la file est vide). Attention : dans ce cas précis, un tableau plein aura un sommet hors des cases du tableau (il ne faudra donc *jamais* le déréférencer s'il atteinte une telle valeur).
- On représente complètement différemment la file : on décale les pointeurs de tête et de queue au fur et à mesure des insertions et suppressions ($+1$ / -1). Le pointeur de queue peut passer de la dernière case à la première, car les éléments actuellement dans la file se trouvent uniquement entre la tête et la queue. Vider ce tableau revient uniquement à passer les pointeurs à la valeur -1. Attention : dans ce cas précis, il est nécessaire de faire très attention à l'ordre de lecture des éléments entre la tête et la queue.

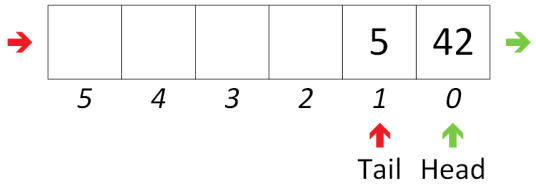
L'exemple suivant montre l'évolution d'une file implémentée avec un tableau fixe au fur et à mesure des ajouts (enfiler / **`ENQUEUE`**) et suppressions (défiler / **`DEQUEUE`**).



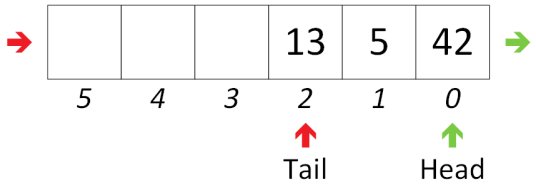
CREATION



ENQUEUE 42



ENQUEUE 5

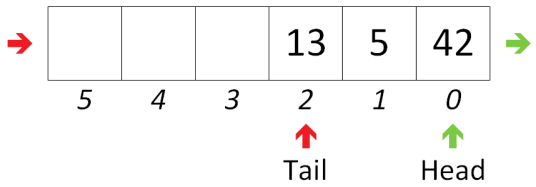


ENQUEUE 13

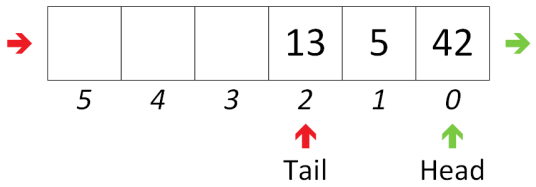
Dans les schémas suivants, deux versions sont présentées :

— celui de gauche présente la version standard où la tête ne bouge pas (il est nécessaire de décaler l'ensemble des éléments du tableau dès que l'on défile),

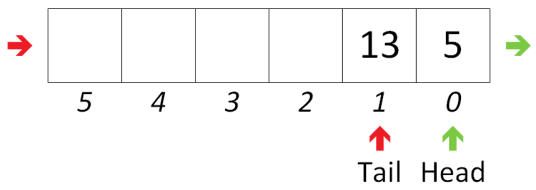
— celui de droite présente la version où seuls les pointeurs de tête et de queue sont décalés (il faut faire attention à l'ordre de lecture entre la tête et la file, et s'appuyer sur les modulos).



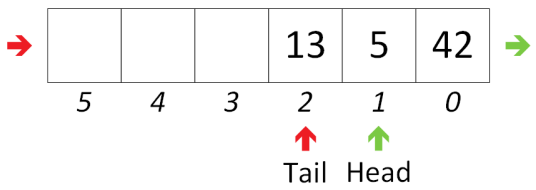
ENQUEUE 13



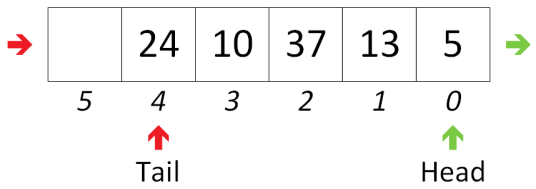
ENQUEUE 13



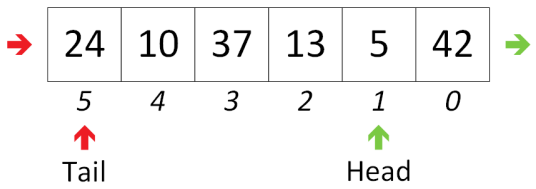
DEQUEUE



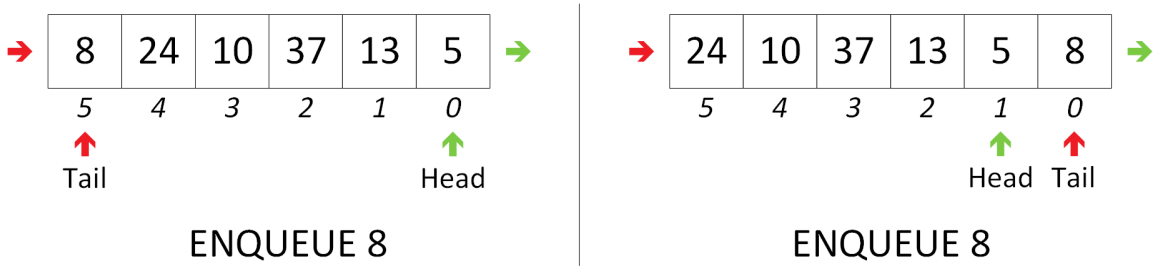
DEQUEUE



ENQUEUE 37, 10, 24



ENQUEUE 37, 10, 24



Dans ce dernier cas, lorsque le tableau de gauche était complètement rempli, on a décalé le pointeur de queue au début du tableau, simplement en utilisant un modulo de la taille du tableau. Il est important de respecter l'ordre de lecture : on lit bel et bien depuis le pointeur de tête, en effectuant un $+1$ (lecture vers la gauche) tout en appliquant un modulo de la taille du tableau au résultat, jusqu'à atteindre le pointeur de queue.
On peut également comprendre qu'il y a eu un dépassement en comparant la position du pointeur de tête et de queue : la différence entre leurs positions est négative ! ($pos.tail - pos.head = 0 - 1 = -1$)

Cette autre façon de gérer la file est un tout petit peu plus complexe dans l'écriture des algorithmes de gestion, mais elle évite énormément de réécritures dans le tableau/en mémoire (inutile de lire une case, l'écrire ailleurs, et recommencer ainsi de suite lors de chaque *dequeue*). Pour ce TP, vous êtes libres de choisir l'implémentation que vous souhaitez réaliser.

Les principales opérations se résument ainsi :

- Création : on alloue en mémoire le tableau (sauf s'il est statique), et on fixe la tête et la queue de la file à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].
- Enfiler : si le tableau est plein, on retourne une erreur, sinon, on ajoute le nouvel élément à gauche de la position du pointeur de queue, et on décale la queue de la file d'un cran à gauche [éventuellement, on met à jour le nombre d'objets dans le tableau]. Autre version : on ajoute le nouvel élément dans la case à gauche de la position du pointeur de queue modulo la taille du tableau, et on décale la queue de la file.
- Défiler : si la file est vide, on retourne une erreur, sinon, on décale l'ensemble des éléments vers la droite [éventuellement, on met à jour le nombre d'objets dans le tableau]. Autre version : on décale la tête de la file d'un cran à gauche.
- Vider : on fixe les pointeurs de tête et de queue à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].
- Tête : on renvoie le contenu de la case du pointeur de tête de file (le prochain élément qui sera défilé).
- Queue : on renvoie le contenu de la case du pointeur de queue de file (le dernier élément qui sera défilé).

5 Exercice 1 - Bibliothèques statique et dynamique

Nom du(es) fichier(s) :	**Makefile**
Répertoire :	**login.x-TP2/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions recommandées :	**gcc(1)**, **ar(1)**

Objectif : Le but de l'exercice est de faire fonctionner l'ensemble de votre projet avec un makefile simple, et de produire une bibliothèque statique ainsi qu'une dynamique.

Une bibliothèque est un ensemble de fonctions et procédures prêtes à être utilisées (ayant éventuellement des variables statiques, ou encore faisant appel à `malloc(3)`) par un utilisateur ne connaissant pas leur implémentation. L'utilisateur développe et écrit son propre programme (utilisant un `main` pour s'exécuter) et il peut éventuellement s'appuyer sur des bibliothèques (n'ayant pas de `main`) pour réutiliser des implémentations existantes.

Les exercices précédents vous ont conduit à produire des fonctions manipulant des structures, afin d'offrir un service à des utilisateurs : une file et son interface pour l'exploiter (une `API`). Vous devez maintenant écrire le (ou les) *makefile(s)* de votre projet (et éventuellement un fichier `configure`) afin de produire une bibliothèque statique nommée **libmyqueue.a** et une bibliothèque dynamique nommée **libmyqueue.so**.

Le *makefile* que vous allez écrire rendra votre projet complet et autonome. Pour cela, vous devez faire en sorte que plusieurs *cibles* soient présentent dans le makefile (celles-ci sont rappelées dans le paragraphe suivant).

Plusieurs stratégies existent pour compiler avec des makefiles, l'une d'entre elle consiste à placer un makefile par dossier afin que le makefile principal appelle les suivants avec la bonne règle (par exemple : le makefile principal va appeler le makefile du dossier *src* pour compiler le projet, mais le makefile principal appellera celui du dossier *check* lorsque l'on demandera d'exécuter la suite de tests).
Pour ce premier contact avec les makefiles, vous pouvez vous contenter d'un seul makefile à la racine exécutant des lignes de compilation toutes prêtes sans aucune réécriture ou extension.

Afin de générer des bibliothèques statiques et dynamiques, vous devez compiler vos fichiers pour produire des fichiers *objets* (des fichiers **.o**), mais vous ne devez pas laisser l'étape d'*édition de liens* classique se faire. À la place, la cible *static* de votre makefile devra produire une bibliothèque statique nommée **libmyqueue.a**, et la cible *shared* devra produire une bibliothèque dynamique nommée **libmyqueue.so**.

Pour générer une bibliothèque statique (*static library* en anglais), on utilise la commande **ar** qui créée des *archives*. Pour produire la bibliothèque statique *libtest.a* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
ar cr libtest.a test1.o file.o
```

Pour générer une bibliothèque dynamique (*shared library* en anglais), on utilise l'option **-shared** du compilateur. Pour produire la bibliothèque dynamique *libtest.so* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
cc test1.o file.o -shared -o libtest.so
```

La première commande génère des fichiers objets, et la deuxième les réunit dans un seul fichier. Il arrivera sur certains systèmes qu'il soit nécessaire d'ajouter l'option **-fpic** ou **-fPIC** (*Position Independent Code*) pour générer des bibliothèques dynamiques. Gardez en tête que *toutes* les bibliothèques **doivent** être préfixées par un *lib* dans le nom des fichiers **.a** et **.so** générés. Mais, lorsque l'on utiliser une bibliothèque dans un projet, on doit ignorer ce préfixe lors de la compilation et l'édition de liens.

Pour utiliser une bibliothèque dynamique sur votre système, plusieurs méthodes existent selon le système où vous vous trouvez.

— L'une d'entre elle consiste à ajouter le dossier contenant la bibliothèque à la variable d'environnement **LD_LIBRARY_PATH** (comme pour la variable **PATH**). Selon votre système, il peut s'agir de la variable d'environnement **DYLD_LIBRARY_PATH**, **LIBPATH**, ou encore **SHLIB_PATH**. Pour ajouter le dossier courant comme dossier contenant des bibliothèques dynamiques en plus d'autres dossiers, vous pouvez taper :
export LD_LIBRARY_PATH=.:/usr/lib:/usr/local/lib

— Une autre méthode consiste à modifier le fichier **/etc/ls.so.conf** et/ou ajouter un fichier contenant le chemin vers votre bibliothèque dans **/etc/ld.so.conf.d/** puis à exécuter **/sbin/ldconfig** pour recharger les dossiers à utiliser.

— Enfin, vous pouvez demander les droits administrateur et installer votre bibliothèque dans **/usr/lib** ou **/usr/local/lib**.

Évidemment, à long terme, l'objectif est d'avoir une bibliothèque installée dans le répertoire **/usr/lib**. Mais, lors du développement, il est préférable d'utiliser la variable **LD_LIBRARY_PATH**. Pour vous assurer du fonctionnement de votre exécutable, vous pouvez utiliser **ldd(1)** avec le chemin complet vers un exécutable. **ldd** vous indiquera quelles bibliothèques sont requises, et où celui-ci les trouve. Si l'une d'entre elle n'est pas trouvée, **ldd** vous en informera :

```
ldd /usr/bin/ls
ldd my_main
```

Pour rappel voici les cibles demandées pour le **Makefile** principal à la racine de votre projet :

`all`	*[Première règle]* lance la règle `libmyqueue`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance le(s) script(s) de test.
`libmyqueue`	lance les règles `shared` et `static`
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

6 Exercice 2 - File avec liste chaînée

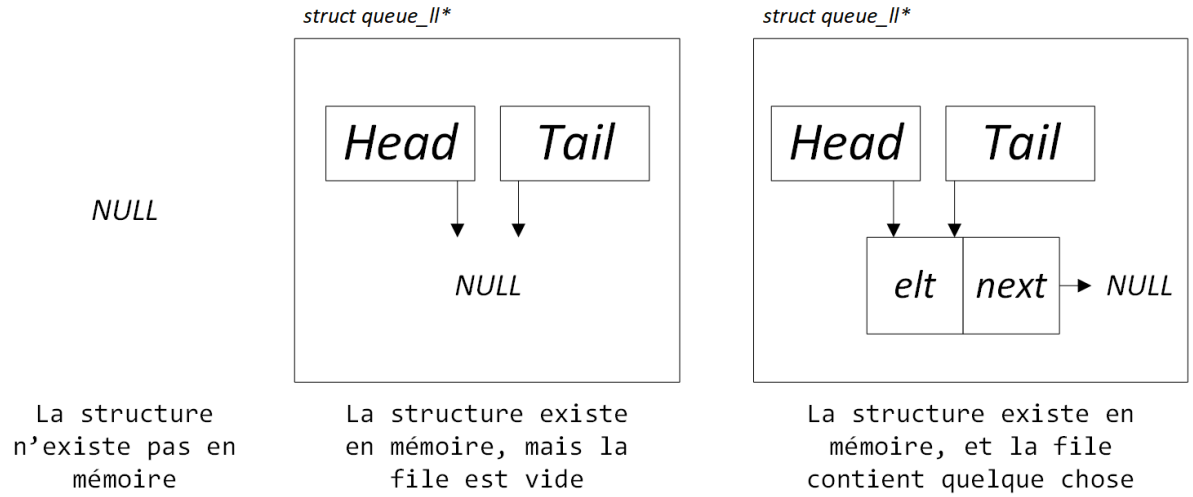
Nom du(es) fichier(s) :	**queue_linked_list.c**
Répertoire :	**login.x-TP2/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **memcpy(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une file en C à base de liste chaînée.

Les fonctions demandées dans cet exercice devront se trouver dans une bibliothèque nommée **libmyqueue**. Après un appel à la commande `make` à la racine du projet, il faut que votre chaîne de compilation produise à la racine de votre projet une version statique de la bibliothèque (qui se nommera **libmyqueue.a**) ainsi qu'une version dynamique de la bibliothèque (qui se nommera **libmyqueue.so**).

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une file. Un fichier **queue_linked_list.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **queue_ll** et l'ajouter dans **queue_linked_list.h**. N'oubliez pas de déclarer également une structure qui contiendra les éléments de la liste chaînée. Pour les premières étapes, vous devrez implémenter une version simplifiée de la file qui ne prend en charge que des entiers positifs.

Conceptuellement, les fonctions manipulant des files de type **queue_ll*** devront pouvoir gérer ces 3 cas :



Vous devez implémenter les fonctions suivantes :

```
queue_ll *queue_ll_create(void);
void queue_ll_delete(queue_ll *queue);

int queue_ll_length(queue_ll *queue);

int queue_ll_enqueue(int elt, queue_ll *queue);
int queue_ll_dequeue(queue_ll *queue);
int queue_ll_head(queue_ll *queue);
int queue_ll_tail(queue_ll *queue);

int queue_ll_clear(queue_ll *queue);
int queue_ll_is_empty(queue_ll *queue);

int queue_ll_insert(int elt, int pos, queue_ll *queue);
int queue_ll_replace(int elt, int pos, queue_ll *queue);

int queue_ll_search(int elt, queue_ll *queue);
queue_ll *queue_ll_reverse(queue_ll *queue);
void queue_ll_print(queue_ll *queue);
```

Liste des fonctions pour une file avec liste chaînée

`queue_ll *queue_ll_create(void)`

Cette fonction crée une file vide. En cas d'erreur (pas assez de mémoire), elle renvoie un pointeur **`NULL`**.

`void queue_ll_delete(queue_ll *queue)`

Cette fonction vide une file de l'ensemble de ses éléments, et détruit la structure restante. Si le paramètre donné est **`NULL`**, la fonction ne fait rien.

`int queue_ll_length(queue_ll *queue)`

Cette fonction renvoie la longueur de la file (c'est-à-dire le nombre d'éléments actuellement dans la file). Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int queue_ll_enqueue(int elt, queue_ll *queue)`

Cette fonction enfile un élément dans une file, c'est-à-dire qu'elle ajoute un élément en queue. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. S'il y a un problème de mémoire, la fonction renvoie -3.

`int queue_ll_dequeue(queue_ll *queue)`

Cette fonction défile un élément d'une file, c'est-à-dire qu'elle supprime l'élément en tête. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_ll_head(queue_ll *queue)`

Cette fonction renvoie l'élément en tête de file. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_ll_tail(queue_ll *queue)`

Cette fonction renvoie l'élément en queue de file. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_ll_clear(queue_ll *queue)`

Cette fonction vide une file de l'ensemble de ses éléments, sans détruire la structure de la file. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si le paramètre donné est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie 0.

`int queue_ll_is_empty(queue_ll *queue)`

Cette fonction teste si une file est vide ou non. Si la file est vide, la fonction renvoie 1. Si la file n'est pas vide, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`int queue_ll_insert(int elt, int pos, queue_ll *queue)`

Cette fonction ajoute un élément dans la file à l'emplacement *pos*. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. L'ancien élément qui était présent à cette position est décalé d'un cran en arrière (vers la queue). Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on ajoute l'élément en queue. Si l'emplacement n'existe pas et est négatif, on ajoute l'élément en tête. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. S'il y a un problème de mémoire, la fonction renvoie -3.

`int queue_ll_replace(int elt, int pos, queue_ll *queue)`

Cette fonction remplace un élément dans la file à l'emplacement *pos*, et renvoie l'élément qui était présent à cet endroit. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on remplace l'élément en queue. Si l'emplacement n'existe pas et est négatif, on remplace l'élément en tête. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_ll_search(int elt, queue_ll *queue)`

Cette fonction recherche un élément dans la file et renvoie sa position dans la liste chaînée. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`queue_ll *queue_ll_reverse(queue_ll *queue)`

Cette fonction inverse la position de tous les éléments de la file. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. En cas de succès, la fonction renvoie le pointeur vers l'éventuelle nouvelle adresse en mémoire de la structure de la file inversée. En cas de problème mémoire, on renvoie **`NULL`**, et l'ancienne file doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie **`NULL`**.

`void queue_ll_print(queue_ll *queue)`

Cette fonction affiche le contenu de la file. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de file sera affiché en premier. Si la file donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la file donnée en paramètre est **`NULL`**, rien n'est affiché.

```
$ ./my_queue_linked_list
42
5
13

$
```

Exemple d'affichage du cas normal : file contenant 42, 5, 13

```
$ ./my_queue_linked_list

$
```

Exemple d'affichage d'une file vide

```
$ ./my_queue_linked_list
$
```

Exemple d'affichage d'un pointeur NULL

7 Exercice 3 - File avec tableau

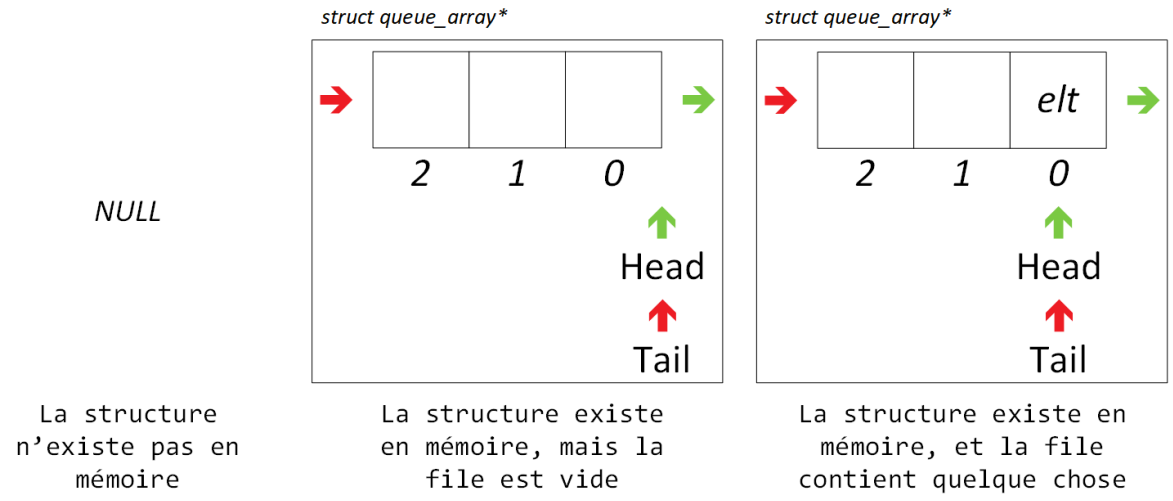
Nom du(es) fichier(s) :	**queue_array.c**
Répertoire :	**login.x-TP2/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **memcpy(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une file en C utilisant un tableau de taille fixe.

Les fonctions demandées dans cet exercice devront également se trouver dans la bibliothèque nommée **libmyqueue**.

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une file. Un fichier **queue_array.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **queue_array** et l'ajouter dans **queue_array.h**. Pour les premières étapes, vous devrez implémenter une version simplifiée de la file qui ne prend en charge que des entiers positifs.

Conceptuellement, les fonctions manipulant des files de type **queue_array*** devront pouvoir gérer ces 3 cas :



Vous devez implémenter les fonctions suivantes :

```
queue_array *queue_array_create(int max_length);
void queue_array_delete(queue_array *queue);

int queue_array_length(queue_array *queue);
int queue_array_max_length(queue_array *queue);
```

```
int queue_array_enqueue(int elt, queue_array *queue);
int queue_array_dequeue(queue_array *queue);
int queue_array_head(queue_array *queue);
int queue_array_tail(queue_array *queue);

int queue_array_clear(queue_array *queue);
int queue_array_is_empty(queue_array *queue);

int queue_array_insert(int elt, int pos, queue_array *queue);
int queue_array_replace(int elt, int pos, queue_array *queue);

int queue_array_search(int elt, queue_array *queue);
queue_array *queue_array_reverse(queue_array *queue);
void queue_array_print(queue_array *queue);
```

Liste des fonctions pour une file avec liste chaînée

`queue_array *queue_array_create(int max_length)`

Cette fonction crée une file vide. Étant donné qu'il s'agit d'une implémentation à base de tableau de taille fixe, la taille maximale du tableau est donnée en paramètre. En cas d'erreur (pas assez de mémoire), elle renvoie un pointeur **`NULL`**.

`void queue_array_delete(queue_array *queue)`

Cette fonction vide une file de l'ensemble de ses éléments, et détruit la structure restante. Si le paramètre donné est **`NULL`**, la fonction ne fait rien.

`int queue_array_max_length(queue_array *queue)`

Cette fonction renvoie la longueur du tableau contenant la file. Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int queue_array_length(queue_array *queue)`

Cette fonction renvoie la longueur de la file (c'est-à-dire le nombre d'éléments actuellement dans la file). Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int queue_array_enqueue(int elt, queue_array *queue)`

Cette fonction enfile un élément dans une file, c'est-à-dire qu'elle ajoute un élément en queue. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si le tableau est déjà plein, la fonction renvoie -3.

`int queue_array_dequeue(queue_array *queue)`

Cette fonction défile un élément d'une file, c'est-à-dire qu'elle supprime l'élément en tête. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_array_head(queue_array *queue)`

Cette fonction renvoie l'élément en tête de file. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_array_tail(queue_array *queue)`

Cette fonction renvoie l'élément en queue de file. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_array_clear(queue_array *queue)`

Cette fonction vide une file de l'ensemble de ses éléments, sans détruire la structure de la file. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si le paramètre donné est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie 0.

`int queue_array_is_empty(queue_array *queue)`

Cette fonction teste si une file est vide ou non. Si la file est vide, la fonction renvoie 1. Si la file n'est pas vide, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`int queue_array_insert(int elt, int pos, queue_array *queue)`

Cette fonction ajoute un élément dans la file à l'emplacement *pos*. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. L'ancien élément qui était présent à cette position est décalé d'un cran en arrière (vers la queue). Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on ajoute l'élément en queue. Si l'emplacement n'existe pas et est négatif, on ajoute l'élément en tête. En cas de succès, la fonction renvoie 0. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si le tableau est déjà plein, la fonction renvoie -3.

`int queue_array_replace(int elt, int pos, queue_array *queue)`

Cette fonction remplace un élément dans la file à l'emplacement *pos*, et renvoie l'élément qui était présent à cet endroit. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on remplace l'élément en queue. Si l'emplacement n'existe pas et est négatif, on remplace l'élément en tête. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la file donnée en paramètre est vide, la fonction renvoie -2.

`int queue_array_search(int elt, queue_array *queue)`

Cette fonction recherche un élément dans la file et renvoie sa position dans le tableau. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`queue_array *queue_array_reverse(queue_array *queue)`

Cette fonction inverse la position de tous les éléments de la file. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. En cas de succès, la fonction renvoie le pointeur vers l'éventuelle nouvelle adresse en mémoire de la structure de la file inversée. En cas de problème mémoire, on renvoie **`NULL`**, et l'ancienne file doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la file donnée en paramètre est **`NULL`**, la fonction renvoie **`NULL`**.

`void queue_array_print(queue_array *queue)`

Cette fonction affiche le contenu de la file. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de file sera affiché en premier. Si la file donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la file donnée en paramètre est **`NULL`**, rien n'est affiché.

```
$ ./my_queue_array
42
5
13

$
```

Exemple d'affichage du cas normal : file contenant 42, 5, 13

```
$ ./my_queue_array

$
```

Exemple d'affichage d'une file vide

```
$ ./my_queue_array
$
```

Exemple d'affichage d'un pointeur NULL

8 Exercice 4 - Suite de tests

Nom du(es) fichier(s) :	**`check.sh`**
Répertoire :	**`login.x-TP2/check/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	750
Fonctions recommandées :	**`diff(1)`**, **`find(1)`**, **`printf(3)`**

Objectif : Le but de l'exercice est de construire une suite de tests pour valider le fonctionnement de la file.

Vous devez écrire plusieurs tests démontrant que vos implémentations des files fonctionnent. Pour cela, vous devrez faire plusieurs programmes en C qui utilisent vos files, et un script shell qui compare les résultats.
Dans les exercices précédents, vous n'avez pas écrit de fonction *main* pour la bonne raison qu'il s'agissait d'implémenter une structure (une file) de différentes façons. Il est maintenant temps de réutiliser tous les tests que vous avez écrits lors du développement de vos files ! Vous devez néanmoins penser aux cas un peu plus complexes que le cas général.

Un exemple avec 3 programmes C de test serait :
- Un programme qui utilise la file implémentée avec des listes chaînées, imprime l'état de la file à différents moments, et imprime certains retours de fonctions.
- Un programme qui utilise la file avec tableau, imprime l'état de la file à différents moments, et imprime certains retours de fonctions.
- Un programme témoin qui servira de vérité absolue sur le comportement attendu.
- Les trois programmes enfilent et défilent les mêmes valeurs dans le même ordre, afin que le même comportement soit visible d'un point de vue utilisateur. Seule l'implémentation sous-jacente varie.

Votre fichier **`main.c`** pourra contenir un scénario précis à suivre qui imprimera différentes valeurs dans le terminal, et vous pouvez éventuellement comparer cela à une sortie texte que vous avez manuellement préparée dans un fichier. Ainsi, le fichier préparé manuellement sera stocké en dur dans le dossier *check* et sera comparé avec un **`diff`** pour confirmer que tout se passe bien, ou au contraire que des problèmes existent lors de certains tests.

Pour utiliser une bibliothèque statique, lors du développement, vous devez disposer de la bibliothèque (le **`.a`** ou **`.so`**) et du fichier **`.h`** associé à l'interface (l'API) de votre bibliothèque, c'est-à-dire aux fonctions exportées.

Lors de la phase de *compilation*, vous devrez parfois ajouter un paramètre *INCLUDE* indiquant le dossier où trouver les *headers* de la bibliothèque (par exemple pour la *libxml2*, on ajoutera ce flag : **`-I/usr/include/libxml2`**). Ce flag **`-I`** permet d'ajouter un dossier dans lequel chercher les fichiers d'en-têtes propres aux bibliothèques (donc les **`.h`** entourés de **`< >`** dans vos *includes*). Vous pouvez cumuler plusieurs flags **`-I`** à la suite.

Lors de la phase d'*édition de liens*, vous devrez parfois ajouter un paramètre *LIBRARY* indiquant le nom de la bibliothèque à utiliser (par exemple pour la *libxml2*, on ajoutera ce flag : **`-lxml2`**). Ce flag **`-l`** (littéralement : *tiret L minuscule*) permet d'indiquer à

l'éditeur de lien qu'il doit utiliser une bibliothèque dynamique dont le nom est donné en paramètre (attention : le préfixe *lib* est supprimé des noms de bibliothèques). Attention, le placement de la bibliothèque par rapport aux fichiers objets (les fichiers **`.o`**) dans la ligne de commande a une importance.

Vous devrez également indiquer dans quel dossier trouver les bibliothèques grâce au flag **`-L`** (par exemple, pour chercher une bibliothèque dans le dossier courant, on ajoutera le flag : **`-L.`**). Vous pouvez cumuler plusieurs flags **`-L`** et **`-l`** à la suite.

Si vous disposez de la même bibliothèque avec une version statique et une dynamique, l'*éditeur de liens* choisira la version dynamique par défaut, mais vous pouvez forcer la statique grâce au flag **`-static`**.

La méthode idéale consiste à produire plusieurs scénarios numérotés et décrits, puis d'afficher quels tests ont réussi, et lesquels ont échoué. Pour cet exercice, vous n'êtes pas obligés de descendre à ce niveau de précision. Une simple petite suite de tests est suffisante, mais n'oubliez pas de vérifier autre chose que le cas général.
N'hésitez pas à vous appuyer sur le fait que votre projet produit des bibliothèques ! Vous pouvez développer un programme entier qui affiche une suite de tests avec un script shell très simple appelant ce programme, ou, vous pouvez utiliser un script shell plus complet qui fait de nombreux tests en s'appuyant sur un ou des programmes C très simples.

9 Exercice 5 - File avec tableau statique (bonus)

Nom du(es) fichier(s) :	**queue_static.c**
Répertoire :	**login.x-TP2/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**write(2)**

Objectif : Le but de l'exercice est d'implémenter une file en C utilisant un tableau statique et sans jamais modifier le tas du processus (c'est-à-dire sans utiliser `malloc(3)`, `mmap(2)`, ou tout autre appel système ou fonction modifiant le tas ou réservant des pages mémoire lors de l'exécution).

Les fonctions demandées dans cet exercice devront également se trouver dans la bibliothèque nommée **libmyqueue**.

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une file. Un fichier **queue_static.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **queue_static** et l'ajouter dans **queue_static.h**. N'oubliez pas qu'un tableau statique est généré par le compilateur : vous devrez indiquer dans la constante de pré-compilation **QUEUE_STATIC_MAX_LEN** la taille maximum. La file étant statique, il vous faudra déclarer une variable globale statique nommée **g_queue_static** qui pointera vers la structure (elle-même déclarée dans la variable globale nommée **g_s_queue_static**). Pour les premières étapes, vous devrez implémenter une version simplifiée de la file qui ne prend en charge que des entiers positifs.
*Attention : étant donné qu'il s'agit d'une version statique, vous ne devez **JAMAIS** utiliser **malloc**, **free**, ou toute autre fonction ou appel système visant à réserver de la mémoire.*
Vous devez implémenter les fonctions suivantes :

```
void queue_static_create(void);
void queue_static_delete(void);

int queue_static_length(void);
int queue_static_max_length(void);

int queue_static_enqueue(int elt);
int queue_static_dequeue(void);
int queue_static_head(void);
int queue_static_tail(void);

int queue_static_clear(void);
int queue_static_is_empty(void);
```

```
int queue_static_insert(int elt, int pos);
int queue_static_replace(int elt, int pos);

int queue_static_search(int elt);
void queue_static_reverse(void);
void queue_static_print(void);
```

Liste des fonctions pour une file avec liste chaînée

`void queue_static_create(void)`

Cette fonction initialise les valeurs de la structure.

`void queue_static_delete(void)`

Cette fonction vide une file de l'ensemble de ses éléments.

`int queue_static_max_length(void)`

Cette fonction renvoie la longueur du tableau contenant la file.

`int queue_static_length(void)`

Cette fonction renvoie la longueur de la file (c'est-à-dire le nombre d'éléments actuellement dans la file).

`int queue_static_enqueue(int elt)`

Cette fonction enfile un élément dans une file, c'est-à-dire qu'elle ajoute un élément en queue. En cas de succès, la fonction renvoie 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si le tableau est déjà plein, la fonction renvoie -3.

`int queue_static_dequeue(void)`

Cette fonction défile un élément d'une file, c'est-à-dire qu'elle supprime l'élément en tête. En cas de succès, la fonction renvoie 0. Si la file est vide, la fonction renvoie -2.

`int queue_static_head(void)`

Cette fonction renvoie l'élément en tête de file. Si la file est vide, la fonction renvoie -2.

`int queue_static_tail(void)`

Cette fonction renvoie l'élément en queue de file. Si la file est vide, la fonction renvoie -2.

`int queue_static_clear(void)`

Cette fonction vide une file de l'ensemble de ses éléments, sans détruire la structure de la file. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si la file est vide, la fonction renvoie 0.

int queue_static_is_empty(void)

Cette fonction teste si une file est vide ou non. Si la file est vide, la fonction renvoie 1. Si la file n'est pas vide, la fonction renvoie 0.

int queue_static_insert(int elt, int pos)

Cette fonction ajoute un élément dans la file à l'emplacement *pos*. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. L'ancien élément qui était présent à cette position est décalé d'un cran en arrière (vers la queue). Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on ajoute l'élément en queue. Si l'emplacement n'existe pas et est négatif, on ajoute l'élément en tête. En cas de succès, la fonction renvoie 0. Si le tableau est déjà plein, la fonction renvoie -3.

int queue_static_replace(int elt, int pos)

Cette fonction remplace un élément dans la file à l'emplacement *pos*, et renvoie l'élément qui était présent à cet endroit. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on remplace l'élément en queue. Si l'emplacement n'existe pas et est négatif, on remplace l'élément en tête. Si la file est vide, la fonction renvoie -2.

int queue_static_search(int elt)

Cette fonction recherche un élément dans la file et renvoie sa position dans le tableau. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la file), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4.

void queue_static_reverse(void)

Cette fonction inverse la position de tous les éléments de la file. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. *Attention : vous ne devez pas utiliser de* ***malloc*** *ou de tableau temporaire pour effectuer cette fonction.*

void queue_static_print(void)

Cette fonction affiche le contenu de la file. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de file sera affiché en premier. Si la file donnée en paramètre est vide, seul un retour à la ligne est affiché. *Attention, en version statique, vous devrez utiliser* `write(2)` *et non pas* `printf(3)`*.*

```
$ ./my_queue_static
42
5
13

$
```

Exemple d'affichage du cas normal : file contenant 42, 5, 13

```
$ ./my_queue_static

$
```

Exemple d'affichage d'une file vide