Langage C

Piles & Files en tableaux

15 janvier 2023

Version 1

Fabrice BOISSIER <fabrice.boissier@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 (donc `prenom.nom`).

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.

Consigne Exceptionnelle : Tout code dans les headers (`.h`) sera sanctionné par un 0 (excepté les prototypes, constantes, et globales, si ceux-ci sont nécessaires).

2 Format de Rendu

Responsable(s) du projet :	**Fabrice BOISSIER** `<fabrice.boissier@epita.fr>`
Balise(s) du projet :	**[C][PFT]**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Devoir/Assignment sur Teams
Nom du répertoire :	`login.x-MiniProjet2`
Nom de l'archive :	`login.x-MiniProjet2.tar.bz2`
Date maximale de rendu :	15/01/2023 23h42
Durée du projet :	1 mois
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).
`README`	contient la description du projet et des exercices, ainsi que la façon d'utiliser le projet.

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 l'intégrant à une série de tests automatisés qui seront fournis un peu plus tard. N'attendez SURTOUT PAS que ces tests soient envoyés pour commencer à produire vos fonctions et vos propres tests.

L'arborescence attendue pour le projet est la suivante :

```
login.x-MiniProjet2/
login.x-MiniProjet2/AUTHORS
login.x-MiniProjet2/README
login.x-MiniProjet2/src/
login.x-MiniProjet2/src/stack_array.c
login.x-MiniProjet2/src/stack_array.h
login.x-MiniProjet2/src/queue_array.c
login.x-MiniProjet2/src/queue_array.h
```

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
```

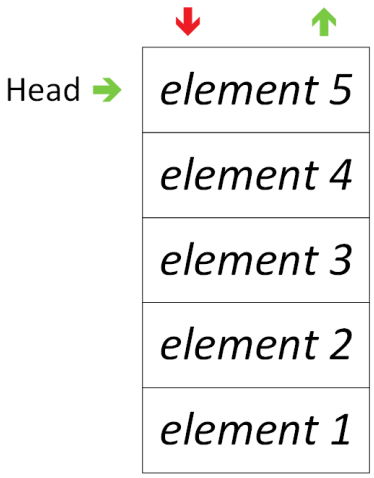
Dans ce sujet précis, vous ferez du code en C et des appels à des scripts shell qui afficheront les résultats dans le terminal (donc des flux de sortie qui pourront être redirigés vers un fichier texte).

4 Cours

Notions étudiées : Tableaux, Pointeurs, Piles

4.1 Rappel sur les piles

Les **piles**, ou **stacks** en anglais, sont des structures visant à stocker les données dans l'ordre d'arrivée, mais ne permettant leur récupération uniquement dans l'ordre inverse. Dans une pile, on ne peut accéder qu'à la dernière donnée stockée, celle se situant au *sommet* de la pile. Ces structures sont aussi appelées **LIFO** (*Last In First Out*).

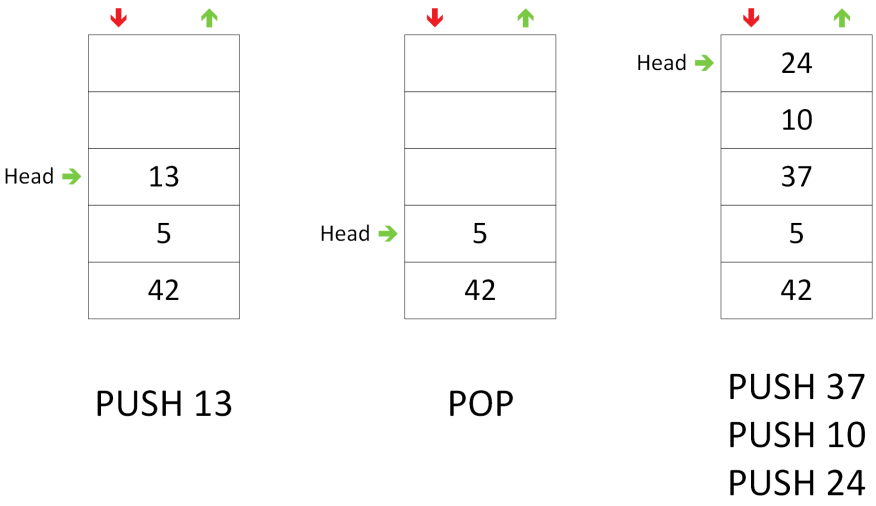


Deux opérations permettent d'utiliser une pile :

- **`PUSH`** : permettant d'*empiler* une donnée supplémentaire dans la pile
- **`POP`** : permettant de *dépiler* une donnée depuis la pile

On ajoute donc une donnée en l'empilant avec un **`PUSH`**, et il est possible de directement y accéder, car elle est au sommet de la pile. À l'inverse, pour accéder à une donnée tout au fond de la pile, il est nécessaire de dépiler autant d'éléments que nécessaire avec un **`POP`**.

Voici un exemple où l'on crée une pile, puis on empile successivement 42, 5, et 13, puis, on dépile une fois (pour récupérer 13), et enfin, on empile successivement 37, 10, 24.

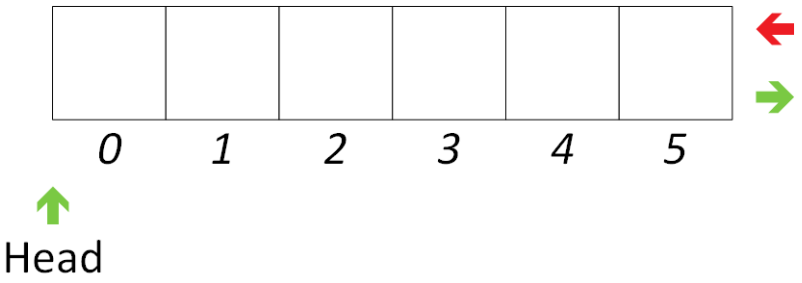


Les piles, et surtout la contrainte d'accès aux objets, sont couramment utilisées : un camion de livraison sera d'abord rempli avec les paquets à livrer en dernier/le camion sera rempli dans l'ordre inverse de livraison (on accède d'abord aux derniers éléments chargés).
En informatique, on utilisera les piles dans certains *parsers* (analyse grammaticale) pour connaître en premier l'opérateur à exécuter (opérateur binaire ? unaire ?) et dépiler par la suite le nombre exact de paramètres. La *pile d'appels* est également une convention fondamentale partagée par les processeurs et les systèmes d'exploitation permettant de passer des paramètres (et d'autres informations de contexte) aux fonctions appelées par les programmes, ou en cas d'interruption pour sauvegarder l'adresse de l'instruction qui était en cours d'exécution.

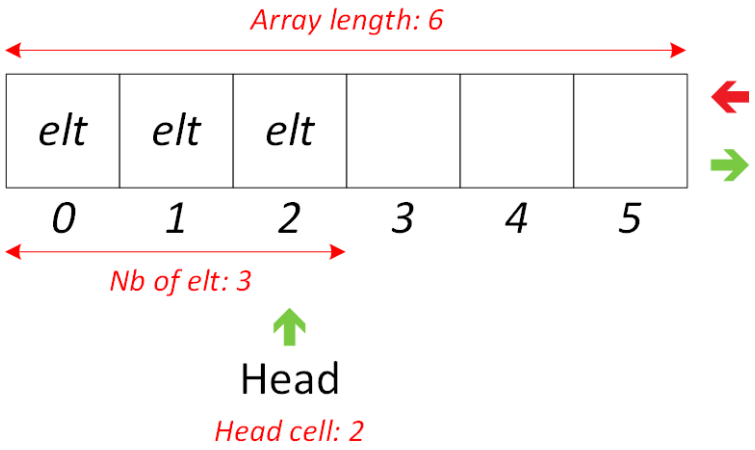
Afin d'implémenter une pile, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et un indicateur de l'élément en haut de la pile. Nous allons maintenant voir comment implémenter une pile avec des listes chaînées et un tableau de taille fixe.

4.2 Piles : implémentation avec un tableau de taille fixe

Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la pile aura une taille maximale, et on peut refuser l'ajout d'un élément si la pile 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 :



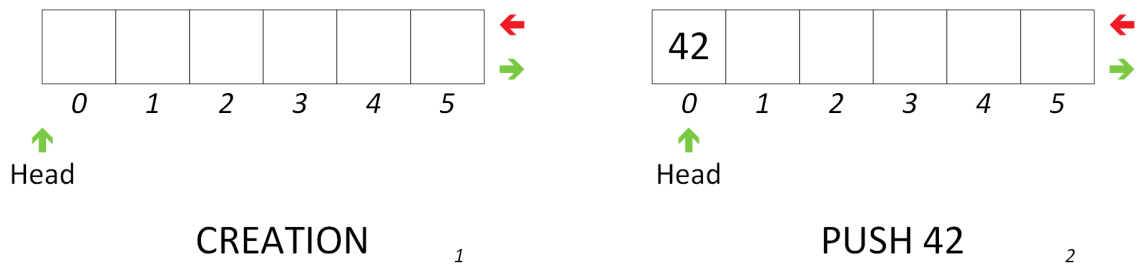
On notera cette fois que plusieurs informations distinctes doivent être conservées : l'adresse du tableau, le numéro de case correspondant au sommet de la pile (*head* dans notre cas), la taille du tableau (le nombre maximum d'objets pouvant être stockés), le nombre d'éléments dans le tableau.
Le schéma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

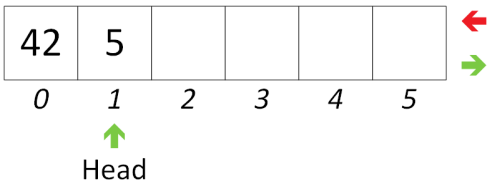


Dans le cas d'un tableau de taille fixe, le pointeur de sommet ne peut pas utiliser la valeur **`NULL`** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que le sommet est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer le sommet de la pile et le cas vide :

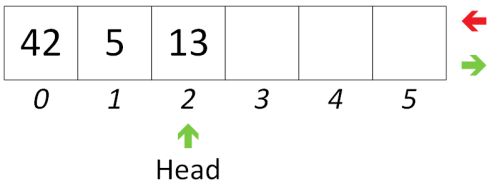
- On enregistre dans la structure de la pile une variable servant à compter le nombre d'éléments présents (le sommet peut donc prendre n'importe quelle valeur tant que la pile est vide).
- On utilise un entier relatif pour indiquer le sommet, et -1 indique que la pile est vide (l'ajout d'un élément décalera le sommet à 0, c'est-à-dire la case où sera l'élément).
- On place le sommet de la pile sur la première case non utilisée, et l'accès au premier élément se fait donc en retirant 1 au pointeur de sommet (ainsi, un sommet à la case 0 indique que la pile 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).

L'exemple suivant montre l'évolution d'une pile implémentée avec un tableau fixe au fur et à mesure des ajouts (empiler / **`PUSH`**) et suppressions (dépiler / **`POP`**).

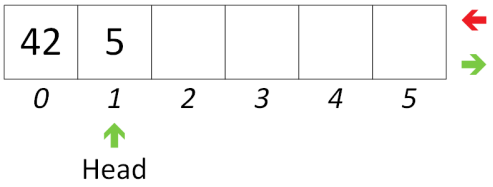




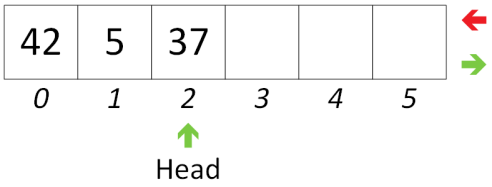
PUSH 5 3



PUSH 13 4



POP 5



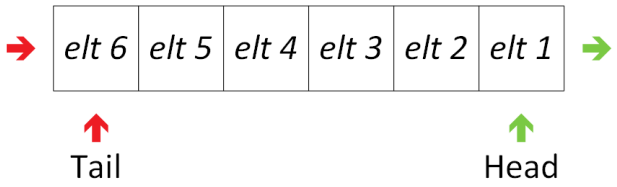
PUSH 37 6

Les principales opérations se résument ainsi :

- Création : on alloue en mémoire le tableau (sauf s'il est statique), et on fixe le sommet de la pile 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].
- Empiler : si le tableau est plein, on retourne une erreur, sinon, on ajoute un élément, et on décale le sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].
- Dépiler : si la pile est vide, on retourne une erreur, sinon, on réduit la valeur du sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].
- Vider : on fixe le sommet de la pile 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].
- Sommet : on renvoie le dernier élément ajouté (cela dépend de comment le sommet a été implémenté !).

4.3 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*).

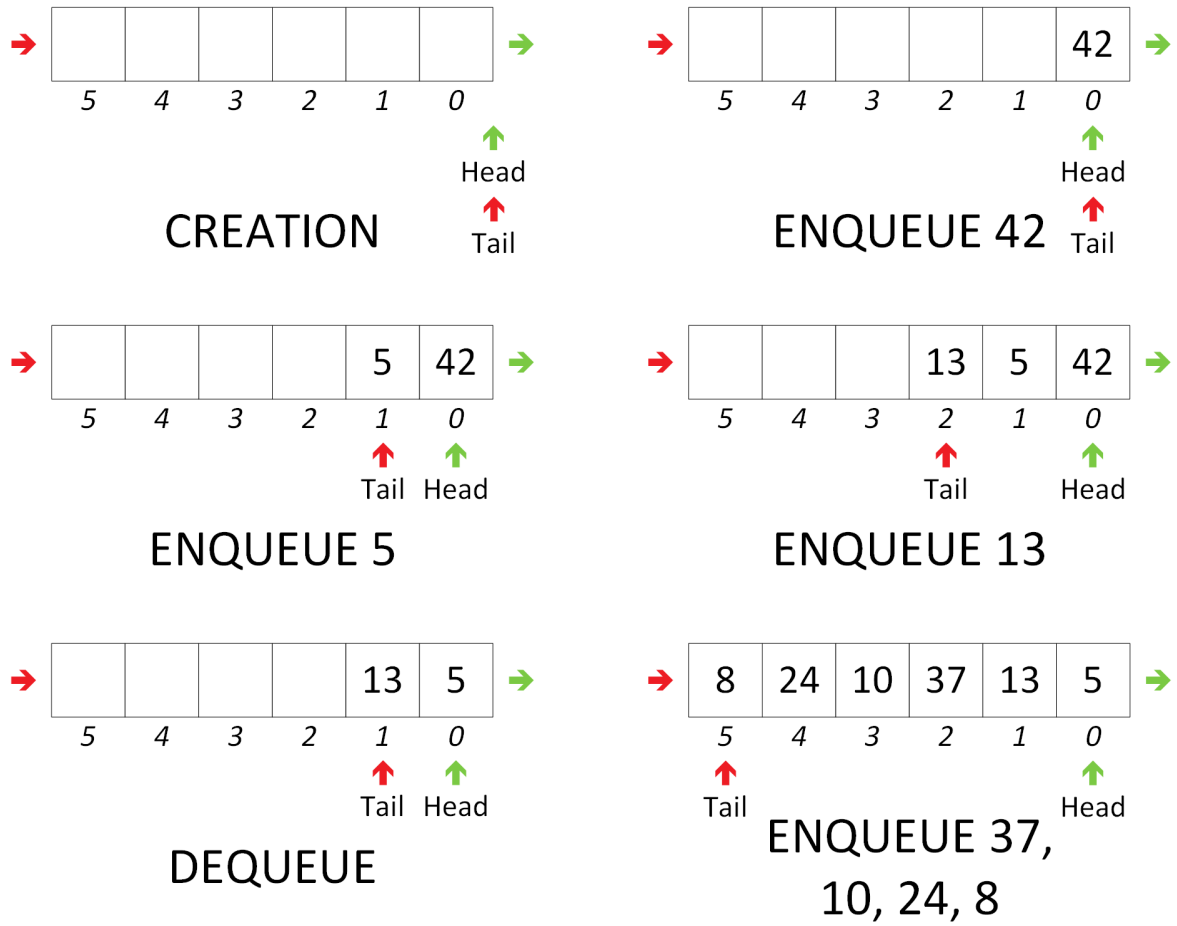


Deux opérations permettent d'utiliser une file :

- **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.

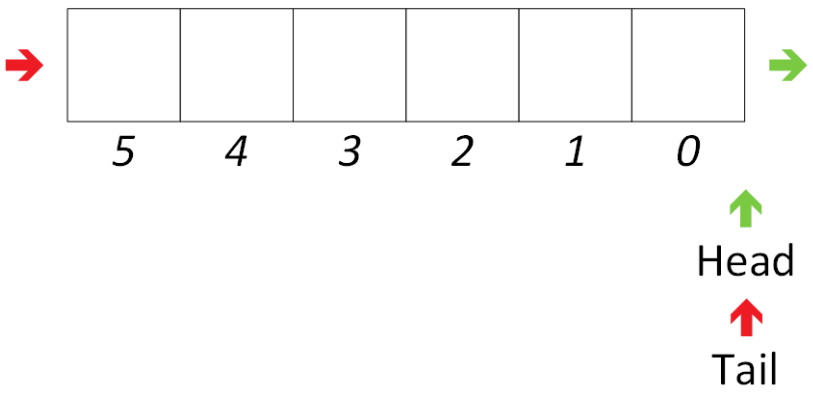


Les files, et surtout le respect de l'ordre d'arrivée des objets, sont couramment utilisés : mise en attente de personnes face à des guichets (voitures à un péage, clients face à une caisse, etc).
En informatique, on utilisera les files pour stocker temporairement et traiter les requêtes dans leur ordre 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).

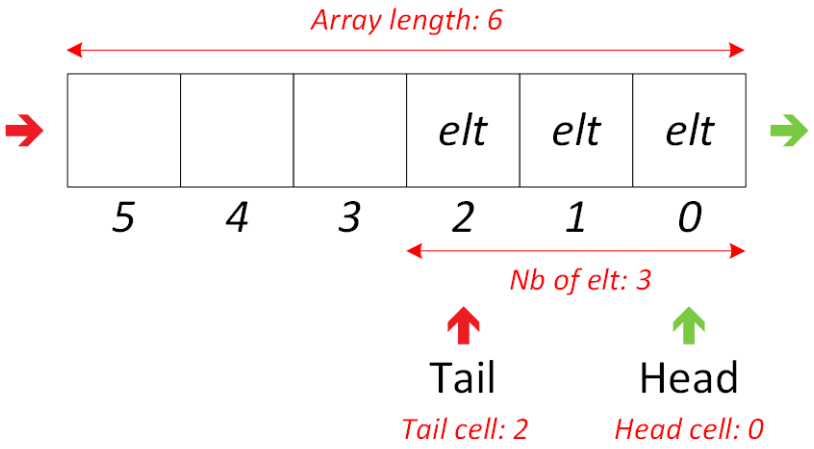
Afin d'implémenter une file, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et deux indicateurs pour l'élément tête de file et l'élément en queue de file. Nous allons maintenant voir comment implémenter une file avec des listes chaînées et un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

4.4 Files : implémentation avec un tableau de taille fixe

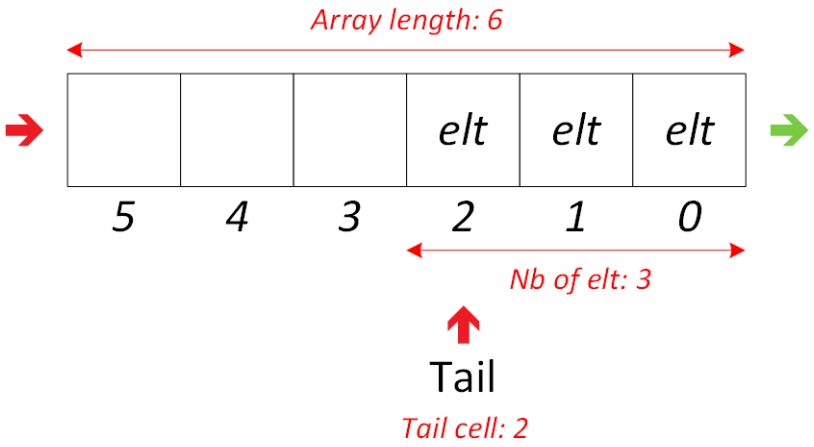
Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la file aura une taille maximale, et on peut refuser l'ajout d'un élément si la file est déjà pleine. La structure diffère également du fait que le tableau est alloué une seule fois lors de sa création (voire même lors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



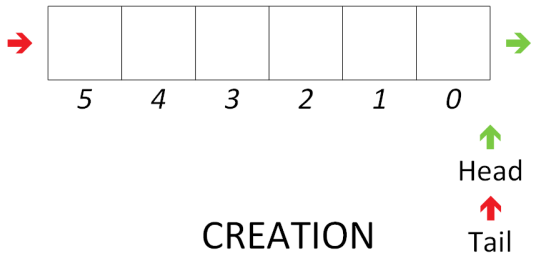
Une façon d'implémenter le tableau permet de supposer que la case 0 contiendra toujours l'élément de tête. Dans ce cas très précis, on peut donc se passer de la variable de tête, et au contraire, s'appuyer sur la variable donnant le nombre d'éléments dans le tableau pour savoir si la file est vide ou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



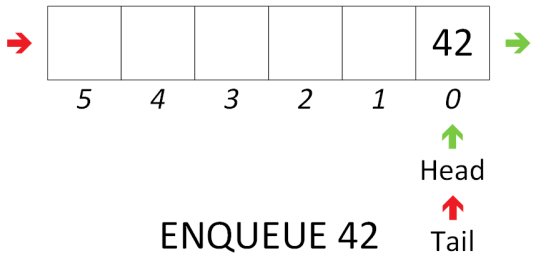
Dans le cas d'un tableau de taille fixe, les pointeurs de tête et de queue ne peuvent pas utiliser la valeur **`NULL`** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que la queue est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer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

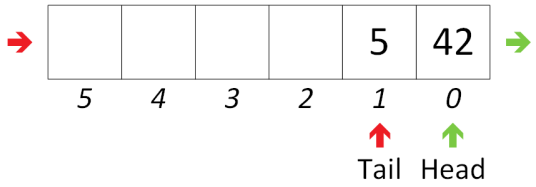
L'exemple suivant montre l'évolution d'une file implémentée avec un tableau fixe au fur et à mesure des ajouts (enfiler / **`ENQUEUE`**) et suppressions (défiler / **`DEQUEUE`**).



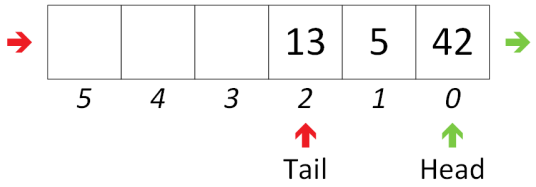
CREATION



ENQUEUE 42



ENQUEUE 5

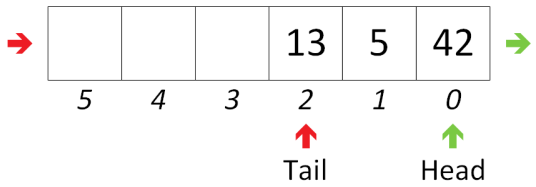


ENQUEUE 13

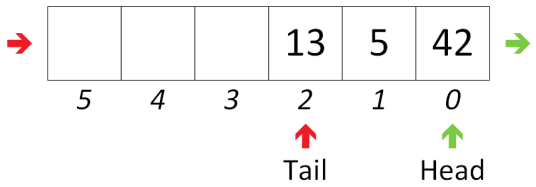
Dans les schémas suivants, deux versions sont présentées :

— 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),

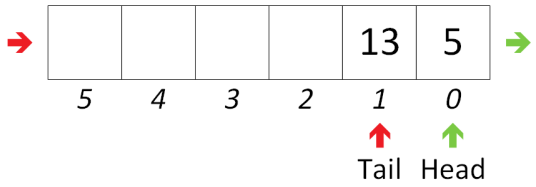
— celui de droite présente la version où seuls les pointeurs de tête et de queue sont décalés (il faut faire attention à l'ordre de lecture entre la tête et la file, et s'appuyer sur les modulos).



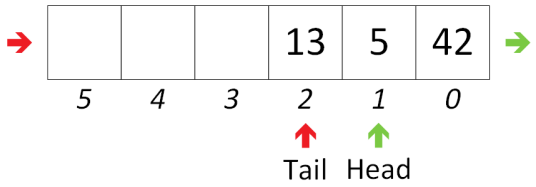
ENQUEUE 13



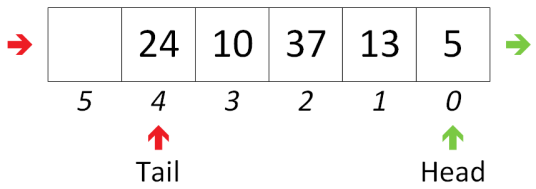
ENQUEUE 13



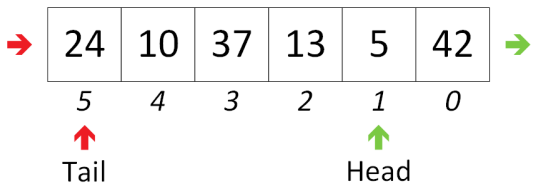
DEQUEUE



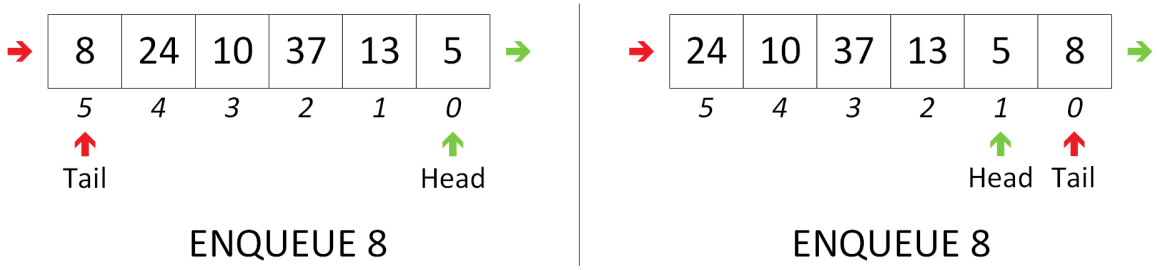
DEQUEUE



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).

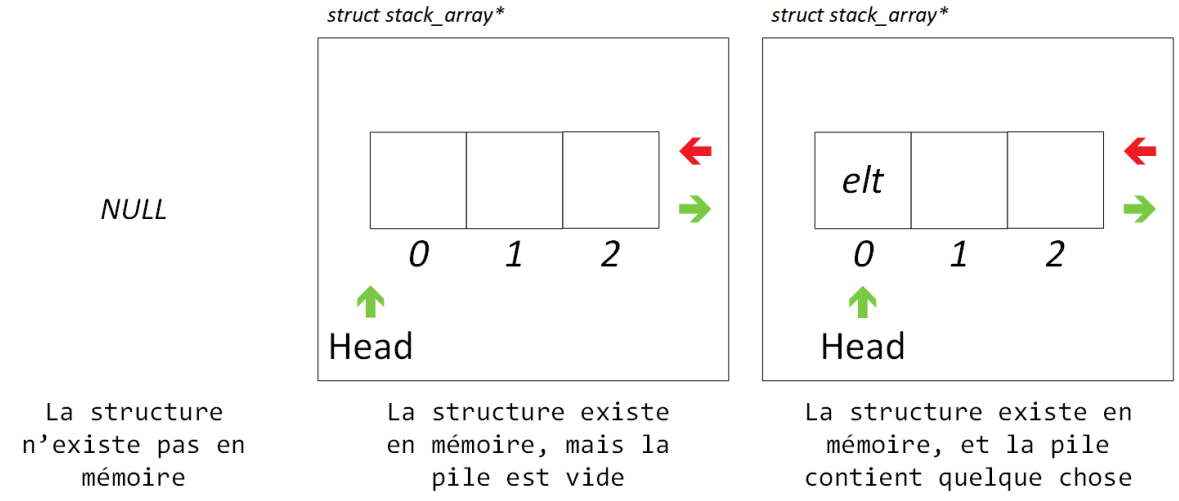
5 Exercice 1 - Pile avec tableau

Nom du(es) fichier(s) :	**stack_array.c**
Répertoire :	**login.x-MiniProjet2/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 pile en C utilisant un tableau de taille fixe.

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

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



Vous devez implémenter les fonctions suivantes :

```
stack_array *stack_array_create(int max_length);
void stack_array_delete(stack_array *stack);

int stack_array_length(stack_array *stack);
int stack_array_max_length(stack_array *stack);

int stack_array_push(int elt, stack_array *stack);
int stack_array_pop(stack_array *stack);
```

```c
int stack_array_head(stack_array *stack);

int stack_array_clear(stack_array *stack);
int stack_array_is_empty(stack_array *stack);

int stack_array_search(int elt, stack_array *stack);
stack_array *stack_array_reverse(stack_array *stack);
void stack_array_print(stack_array *stack);
```

Liste des fonctions pour une pile avec liste chaînée

`stack_array *stack_array_create(int max_length)`

Cette fonction crée une pile 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 stack_array_delete(stack_array *stack)`

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

`int stack_array_max_length(stack_array *stack)`

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

`int stack_array_length(stack_array *stack)`

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

`int stack_array_push(int elt, stack_array *stack)`

Cette fonction empile un élément dans une pile, c'est-à-dire qu'elle ajoute un élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile 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 stack_array_pop(stack_array *stack)`

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

`int stack_array_head(stack_array *stack)`

Cette fonction renvoie l'élément au sommet de la pile. Si la pile donnée en paramètre est **NULL**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie -2.

`int stack_array_clear(stack_array *stack)`

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

`int stack_array_is_empty(stack_array *stack)`

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

`int stack_array_search(int elt, stack_array *stack)`

Cette fonction recherche un élément dans la pile 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 le fond de la pile), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`stack_array *stack_array_reverse(stack_array *stack)`

Cette fonction inverse la position de tous les éléments de la pile. 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 pile inversée. En cas de problème mémoire, on renvoie **`NULL`**, et l'ancienne pile doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie **`NULL`**.

`void stack_array_print(stack_array *stack)`

Cette fonction affiche le contenu de la pile. 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 pile sera affiché en premier. Si la pile donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la pile donnée en paramètre est **`NULL`**, rien n'est affiché.

```
$ ./my_stack_array
42
5
13

$
```

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

```
$ ./my_stack_array

$
```

Exemple d'affichage d'une pile vide

```
$ ./my_stack_array
$
```

Exemple d'affichage d'un pointeur NULL

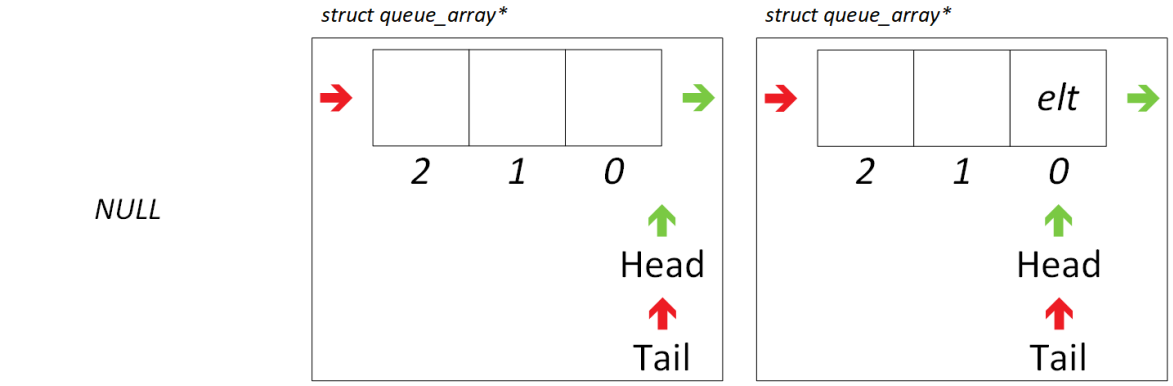
6 Exercice 2 - File avec tableau

Nom du(es) fichier(s) :	**queue_array.c**
Répertoire :	**login.x-MiniProjet2/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.

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);
```

```c
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