



Langage C

Piles & Files en pointeurs

15 janvier 2023

Version 1



Fabrice BOISSIER <fabrice.boissier@epita.fr>

Copyright

Ce document est destiné à une utilisation interne à EPITA.

Copyright © 2022/2023 Fabrice BOISSIER

La copie de ce document est soumise à conditions :

- ▷ Il est interdit de partager ce document avec d'autres personnes.
- ▷ Vérifiez que vous disposez de la dernière révision de ce document.

Table des matières

1	Consignes Générales	IV
2	Format de Rendu	V
3	Aide Mémoire	VI
4	Cours	1
4.1	Rappel sur les piles	1
4.2	Piles : implémentation avec des listes chaînées	2
4.3	Rappel sur les files	4
4.4	Files : implémentation avec des listes chaînées	5
5	Exercice 1 - Pile avec liste chaînée	7
6	Exercice 2 - File avec liste chaînée	11

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] [PFP]
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Devoir/Assignment sur Teams
Nom du répertoire :	login.x-MiniProjet3
Nom de l'archive :	login.x-MiniProjet3.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-MiniProjet3/  
login.x-MiniProjet3/AUTHORS  
login.x-MiniProjet3/README  
login.x-MiniProjet3/src/  
login.x-MiniProjet3/src/stack_linked_list.c  
login.x-MiniProjet3/src/stack_linked_list.h  
login.x-MiniProjet3/src/queue_linked_list.c  
login.x-MiniProjet3/src/queue_linked_list.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
```

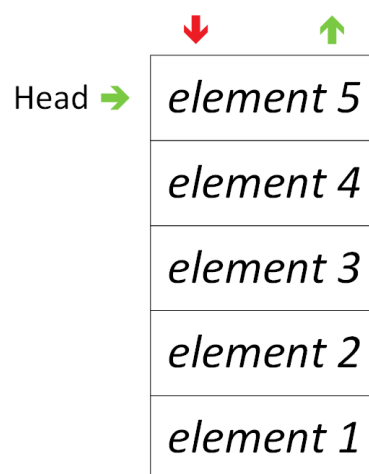
Dans ce sujet précis, vous ferez du code en C et des appels à des scripts shell qui afficheront les résultats dans le terminal (donc des flux de sortie qui pourront être redirigés vers un fichier texte).

4 Cours

Notions étudiées : Tableaux, Pointeurs, Files

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*).

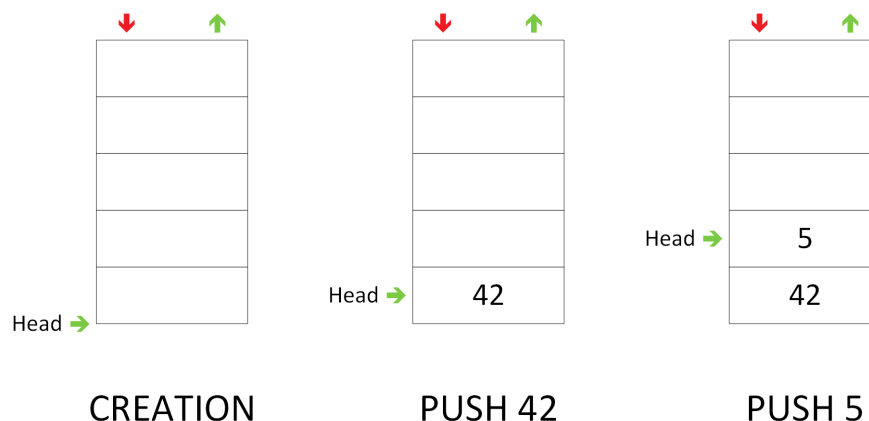


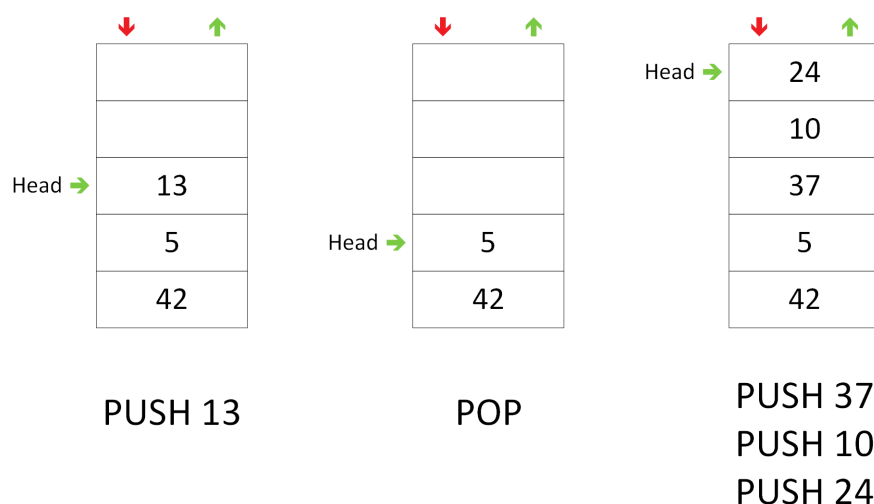
Deux opérations permettent d'utiliser une pile :

- **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.





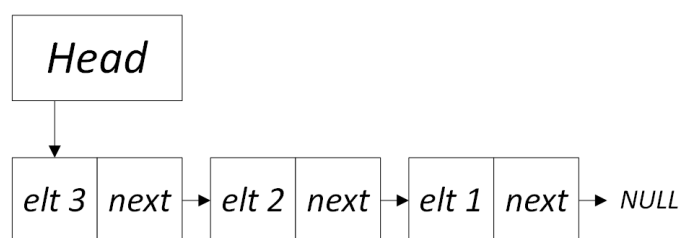
Les piles, et surtout la contrainte d'accès aux objets, sont couramment utilisées : un camion de livraison sera d'abord rempli avec les paquets à livrer en dernier/le camion sera rempli dans l'ordre inverse de livraison (on accède d'abord aux derniers éléments chargés).

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.

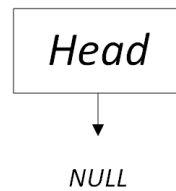
4.2 Piles : implémentation avec des listes chaînées

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 pile sous forme de liste chaînée en mémoire :

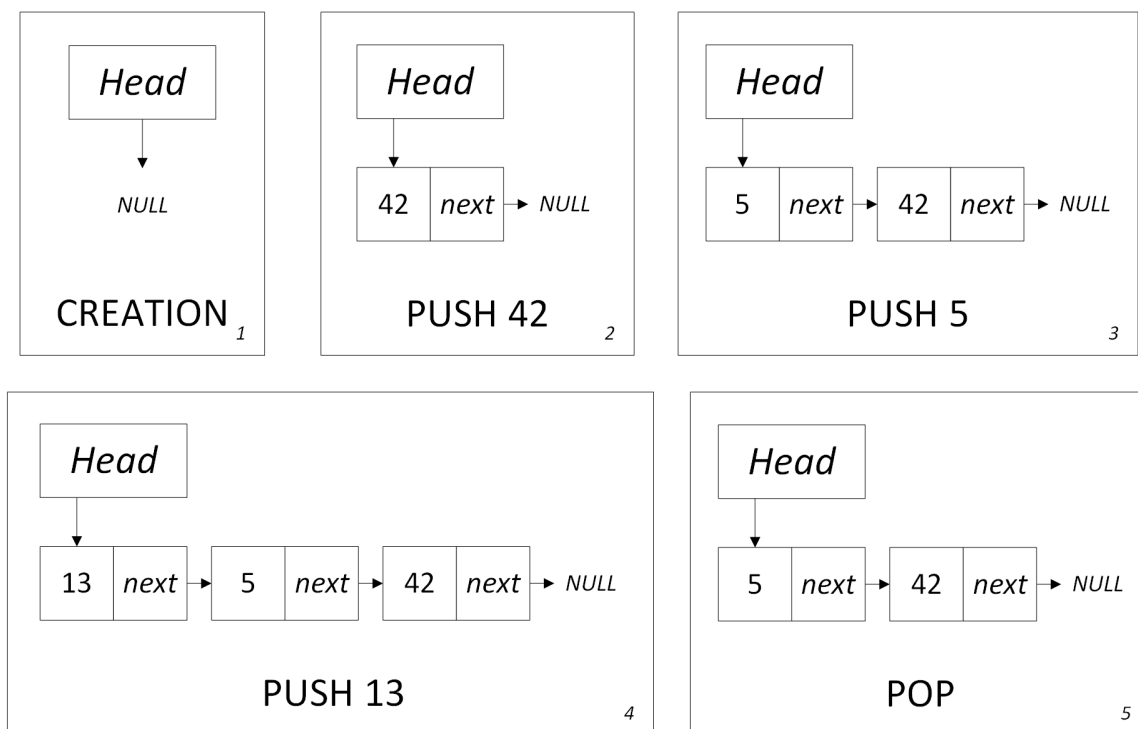


On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant), ainsi qu'un pointeur indiquant le sommet de la pile (*head* dans notre cas).

L'unique cas particulier concerne une pile vide : le pointeur de sommet vaut dans ce cas **NULL**. Il s'agit également de l'état dans lequel se trouve une pile vidée ou nouvellement créée.



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

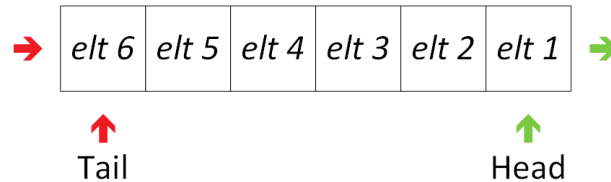


Les principales opérations se résument ainsi :

- Création : on alloue en mémoire la structure générale de la pile, et on fixe le sommet de la pile à **NULL**.
- Empiler : on alloue en mémoire un nouvel élément dont le pointeur *next* pointe vers l'actuel élément au sommet de la pile, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de ce nouvel élément.
- Dépiler : si la pile est vide, on retourne une erreur, sinon, on récupère tout d'abord l'adresse de l'élément suivant celui au sommet, puis, on libère l'élément au sommet, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de l'élément suivant.
- Vider : on dépile successivement tous les éléments jusqu'à obtenir un sommet à **NULL**.
- Sommet : on renvoie le contenu de l'élément au sommet de la pile.

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*).

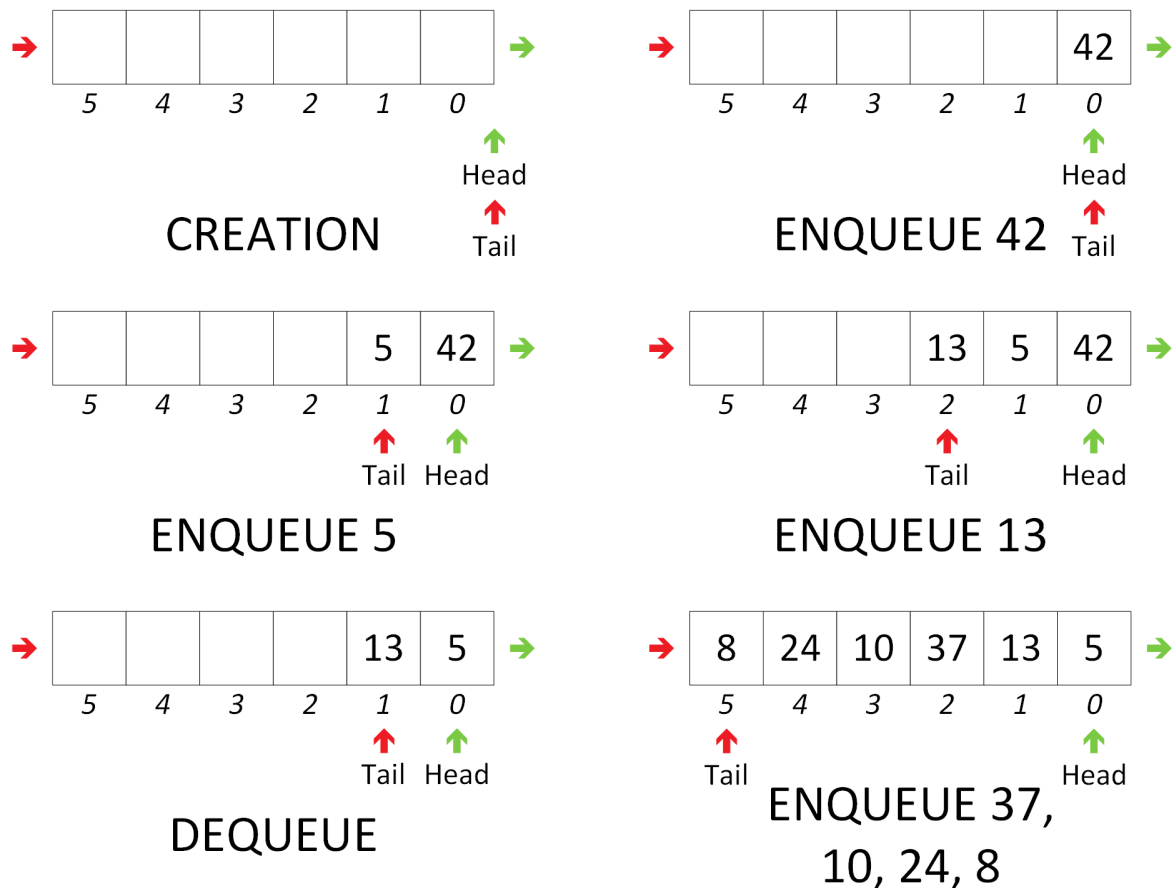


Deux opérations permettent d'utiliser une file :

- **ENQUEUEE** : permettant d'*enfiler* une donnée supplémentaire dans la file
- **DEQUEUEE** : permettant de *défiler* une donnée depuis la file

On ajoute donc une donnée en l'enfilant avec un **ENQUEUEE**, 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 **DEQUEUEE**, 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.



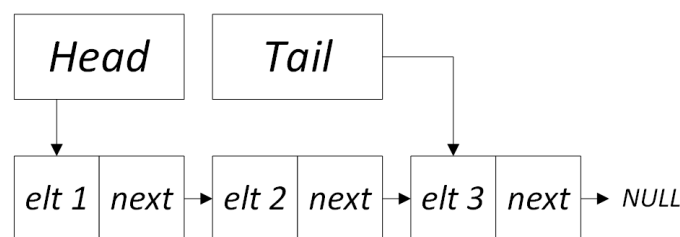
Les files, et surtout le respect de l'ordre d'arrivée des objets, sont couramment utilisés : mise en attente de personnes face à des guichets (voitures à un péage, clients face à une caisse, etc).

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 événements (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.

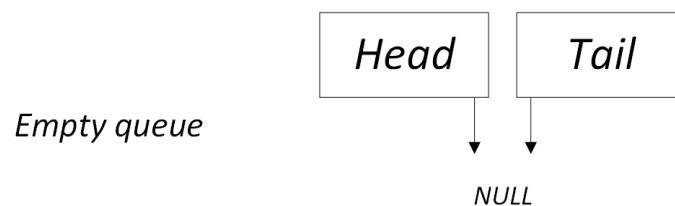
4.4 Files : implémentation avec des listes chaînées

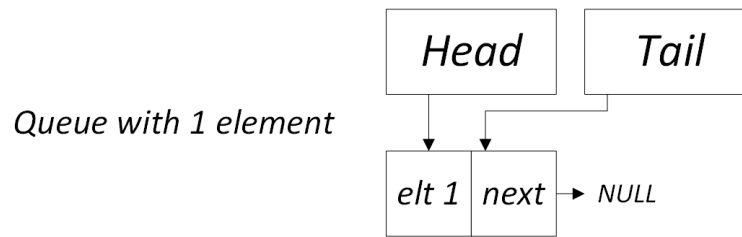
Une implémentation à l'aide d'une liste chaînée permet d'exploiter la mémoire et d'être donc beaucoup plus flexible en terme de nombre maximum d'éléments. Le schéma suivant illustre une file sous forme de liste chaînée en mémoire :



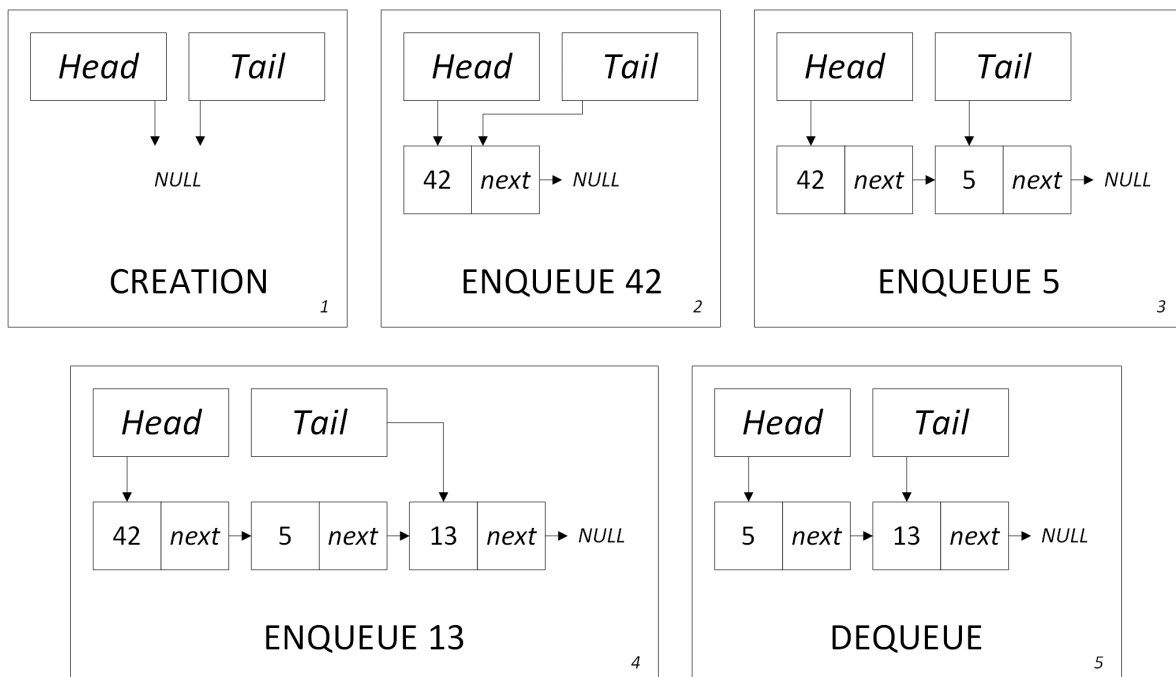
On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant), ainsi que deux pointeurs indiquant respectivement la tête de la file (*head*) et la queue de la file (*tail*).

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).

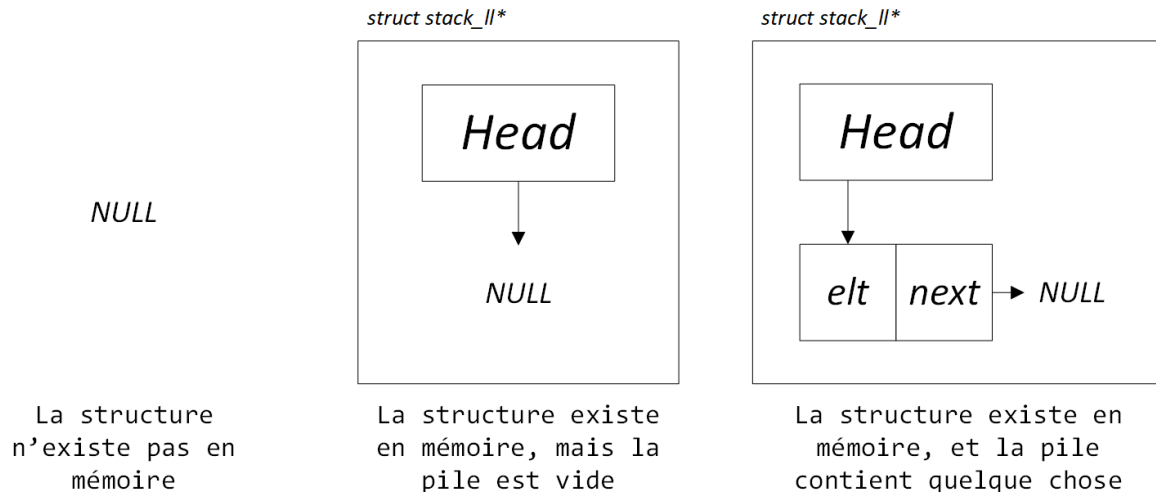
5 Exercice 1 - Pile avec liste chaînée

Nom du(es) fichier(s) : **stack_linked_list.c**
Répertoire : **login.x-MiniProjet3/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 à base de liste chaînée.

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

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



Vous devez implémenter les fonctions suivantes :

```
stack_ll *stack_ll_create(void);
void stack_ll_delete(stack_ll *stack);

int stack_ll_length(stack_ll *stack);

int stack_ll_push(int elt, stack_ll *stack);
int stack_ll_pop(stack_ll *stack);
int stack_ll_head(stack_ll *stack);

int stack_ll_clear(stack_ll *stack);
int stack_ll_is_empty(stack_ll *stack);

int stack_ll_search(int elt, stack_ll *stack);
stack_ll *stack_ll_reverse(stack_ll *stack);
void stack_ll_print(stack_ll *stack);
```

Liste des fonctions pour une pile avec liste chaînée

stack_ll *stack_ll_create(void)

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

void stack_ll_delete(stack_ll *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_ll_length(stack_ll *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_ll_push(int elt, stack_ll *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 . S'il y a un problème de mémoire, la fonction renvoie -3 .

int stack_ll_pop(stack_ll *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_ll_head(stack_ll *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_ll_clear(stack_ll *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_ll_is_empty(stack_ll *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_ll_search(int elt, stack_ll *stack)

Cette fonction recherche un élément dans la pile 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 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_ll *stack_ll_reverse(stack_ll *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_ll_print(stack_ll *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_linked_list
42
5
13

$
```

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


```
$ ./my_stack_linked_list  
  
$
```

Exemple d’affichage d’une pile vide

```
$ ./my_stack_linked_list  
$
```

Exemple d’affichage d’un pointeur NULL

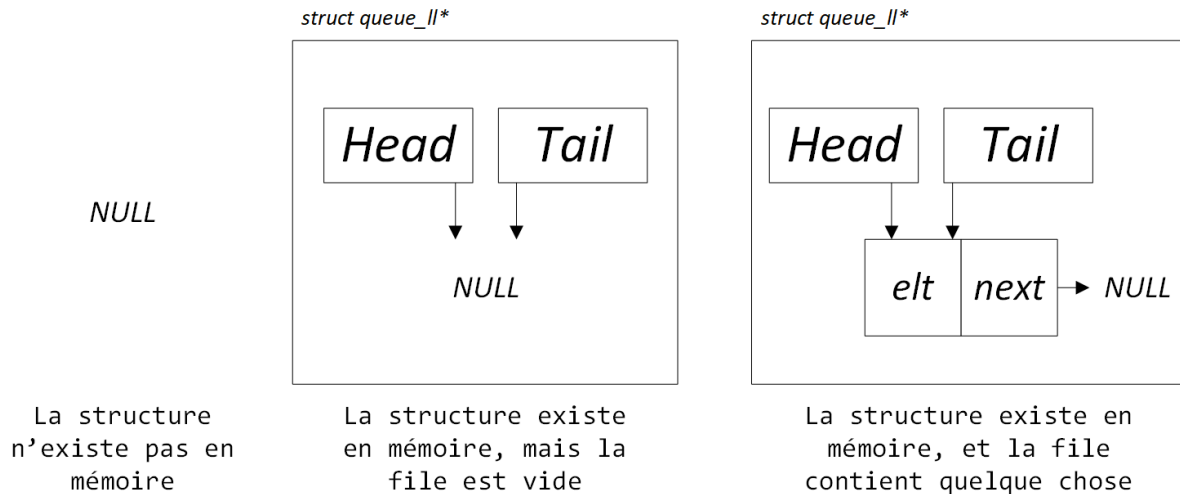
6 Exercice 2 - File avec liste chaînée

Nom du(es) fichier(s) :	<code>queue_linked_list.c</code>
Répertoire :	<code>login.x-MiniProjet3/src/</code>
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	<code>malloc(3)</code> , <code>free(3)</code> , <code>memcpy(3)</code> , <code>printf(3)</code>

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

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