Algorithmique 2

Listes - Piles - Files

28 janvier 2024

Version 4

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 :	**`[ALG][LPF]`**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Devoir sur Moodle
Nom du répertoire :	`login.x-MiniProjet1`
Nom de l'archive :	`login.x-MiniProjet1.tar.bz2`
Date maximale de rendu :	28/01/2024 23h42
Durée du projet :	1,5 semaine
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.
`configure`	le script shell de configuration pour l'environnement de compilation.

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

`all`	*[Première règle]* lance la règle `libmylpf`.
`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.
`libmylpf`	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-MiniProjet1/libmylpf.a
login.x-MiniProjet1/libmylpf.so

L'arborescence attendue pour le projet est la suivante :

login.x-MiniProjet1/
login.x-MiniProjet1/AUTHORS
login.x-MiniProjet1/README
login.x-MiniProjet1/Makefile
login.x-MiniProjet1/configure
login.x-MiniProjet1/src/
login.x-MiniProjet1/src/list_linked.c
login.x-MiniProjet1/src/list_linked.h
login.x-MiniProjet1/src/queue_linked.c
login.x-MiniProjet1/src/queue_linked.h
login.x-MiniProjet1/src/stack_linked.c
login.x-MiniProjet1/src/stack_linked.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 néanmoins 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 Exercice 1 - Listes chaînées

Nom du(es) fichier(s) :	**list_linked.c**
Répertoire :	**login.x-MiniProjet1/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une liste chaînée.

Vous devez écrire plusieurs fonctions permettant de gérer des listes chaînées, c'est-à-dire des listes à base de pointeurs. Un fichier **list_linked.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. La structure **list_linked** est déjà déclarée dedans, vous ne devez pas la modifier. La liste chaînée ainsi représentée contient deux champs exactement : *elt* (l'élément inséré) et *next* (pointeur vers le maillon suivant de la chaîne).

Vous devez implémenter les fonctions suivantes :

```
list_linked *add_elt_list_linked(list_linked *list,
                                 int elt,
                                 int pos);
list_linked *del_elt_list_linked(list_linked *list,
                                 int pos);

int length_list_linked(list_linked *list);
int is_empty_list_linked(list_linked *list);

void print_list_linked(list_linked *list);

int search_elt_list_linked(list_linked *list,
                           int elt);
int get_elt_list_linked(list_linked *list,
                        int pos);

int clear_list_linked(list_linked *list);
```

`list_linked *add_elt_list_linked(list_linked *list, int elt, int pos)`

Cette fonction ajoute un élément à la position indiquée de la liste chaînée donnée en paramètre.

Si la liste est vide, il faut y créer un premier élément qui sera considéré comme en position 1.
Si l'élément *elt* donné en paramètre est inférieur à 1, la fonction doit renvoyer un pointeur **`NULL`** sans rien modifier dans la liste.
Si la position n'existe pas (c'est-à-dire si elle est inférieure strictement à 1 ou supérieure strictement à la taille de la liste + 1), la fonction doit renvoyer un pointeur **`NULL`** sans rien modifier dans la liste. Si un élément se trouve déjà en position *pos*, il faut décaler cet élément un cran vers la fin (de na position n à la position $n+1$), puis, on ajoute le nouvel élément à la position donnée en paramètre.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée. En cas d'erreur (pas assez de mémoire), cette fonction doit renvoyer un pointeur **`NULL`**.

`list_linked *del_elt_list_linked(list_linked *list, int pos)`

Cette fonction supprime l'élément à la position indiquée de la liste chaînée donnée en paramètre.

Si la liste donnée en paramètre est vide, cette fonction doit renvoyer un pointeur **`NULL`**.
Si la position n'existe pas (c'est-à-dire si elle est inférieure strictement à 1 ou supérieure strictement à la taille de la liste), la fonction doit renvoyer un pointeur **`NULL`** sans rien modifier dans la liste.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée.

`int length_list_linked(list_linked *list)`

Cette fonction renvoie la taille de la liste donnée en paramètre, c'est-à-dire le nombre d'éléments présents dans la liste.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.

`int is_empty_list_linked(list_linked *list)`

Cette fonction teste si la liste est vide ou non.

Si la liste est vide, c'est-à-dire si le pointeur est **`NULL`**, il faut renvoyer 1 (l'équivalent de *true* en C), sinon, si la liste n'est pas vide, il faut renvoyer 0 (l'équivalent de *faux* en C).

`void print_list_linked(list_linked *list)`

Cette procédure affiche les éléments de la liste. Chaque élément doit être suivi d'un retour à la ligne.

Si la liste donnée en paramètre est vide, la procédure ne fait rien.
Le format attendu est le suivant :

***`elt`*`\n`**

Ce qui donnerait cet affichage pour la liste suivante :

```
$ ./liste_ll_example1
42
21
8
24
64
$
```

pos	1	2	3	4	5
elt	42	21	8	24	64

`int search_elt_list_linked(list_linked *list, int elt)`

Cette fonction cherche un élément et renvoie sa position.

La première position démarre à la valeur 1.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer -1. Si l'élément n'est pas trouvé, la fonction doit renvoyer -2.

Si plusieurs cas d'erreur se produisent simultanément, leur gestion doit se faire dans cet ordre précisément : le test de la liste vide en premier, puis en dernier le test de l'existence de la valeur.

`int get_elt_list_linked(list_linked *list, int pos)`

Cette fonction renvoie l'élément présent à la position indiquée.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer -1.
Si la position n'existe pas (c'est-à-dire si elle est inférieure strictement à 1 ou supérieure strictement à la taille de la liste), la fonction doit renvoyer -2.

Si plusieurs cas d'erreur se produisent simultanément, leur gestion doit se faire dans cet ordre précisément : le test de la liste vide en premier, puis en dernier le test de la position.

`int clear_list_linked(list_linked *list)`

Cette fonction vide la liste de tous ses éléments, puis renvoie le nombre d'éléments qui ont été supprimés.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.

5 Exercice 2 - Bibliothèques statique et dynamique

Nom du(es) fichier(s) :	**`Makefile`**
Répertoire :	**`login.x-MiniProjet1/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Outils recommandés :	**`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 suivants vont vous conduire à produire des fonctions manipulant des structures, afin d'offrir un service à des utilisateurs : des arbres binaires et leur interface pour les exploiter (une `API`). Vous devez donc écrire le (ou les) *makefile(s)* de votre projet et un fichier `configure` (éventuellement vide) afin de produire une bibliothèque statique nommée **`libmylpf.a`** et une bibliothèque dynamique nommée **`libmylpf.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 **`libmylpf.a`**, et la cible *shared* devra produire une bibliothèque dynamique nommée **`libmylpf.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/ld.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 `libmylpf`.
`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.
`libmylpf`	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 3 - Piles

Nom du(es) fichier(s) :	**stack_linked.c**
Répertoire :	**login.x-MiniProjet1/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une des structures de base : les piles à base de listes chaînées.

Vous devez écrire plusieurs fonctions permettant de gérer des piles à base de listes chaînées, c'est-à-dire des piles exploitant des listes à base de pointeurs. Un fichier **stack_linked.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devrez réutiliser la structure **list_linked** telle quelle issue du fichier **list_linked.h** : **vous ne devez pas créer de nouvelle structure**. La liste chaînée ainsi représentée contient deux champs exactement : *elt* (l'élément inséré) et *next* (pointeur vers le maillon suivant de la chaîne).

Vous devez implémenter les fonctions suivantes :

```
list_linked *push_stack_linked(list_linked *list,
                               int elt);
list_linked *pop_stack_linked(list_linked *list);

int length_stack_linked(list_linked *list);
int is_empty_stack_linked(list_linked *list);

void print_stack_linked(list_linked *list);

int get_head_stack_linked(list_linked *list);

int clear_stack_linked(list_linked *list);
```

`list_linked *push_stack_linked(list_linked *list, int elt)`

Cette fonction empile un nouvel élément, c'est-à-dire qu'elle ajoute un élément en tête de la pile, c'est-à-dire qu'elle ajoute un élément en première position de la liste chaînée donnée en paramètre.

Si la pile est vide, il faut y créer un premier élément.
Si l'élément *elt* donné en paramètre est inférieur à 1, la fonction doit renvoyer un pointeur **`NULL`** sans rien modifier dans la liste.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée. En cas d'erreur (pas assez de mémoire), cette fonction doit renvoyer un pointeur **`NULL`**.

`list_linked *pop_stack_linked(list_linked *list)`

Cette fonction dépile l'élément en tête, c'est-à-dire qu'elle supprime l'élément en première position de la liste chaînée donnée en paramètre.

Si la liste donnée en paramètre est vide, cette fonction doit renvoyer un pointeur **`NULL`**.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée.

`int length_stack_linked(list_linked *list)`

Cette fonction renvoie la taille de la pile donnée en paramètre, c'est-à-dire le nombre d'éléments présents dans la liste.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.

`int is_empty_stack_linked(list_linked *list)`

Cette fonction teste si la pile est vide ou non.

Si la liste est vide, c'est-à-dire si le pointeur est **`NULL`**, il faut renvoyer 1 (l'équivalent de *true* en C), sinon, si la liste n'est pas vide, il faut renvoyer 0 (l'équivalent de *faux* en C).

`void print_stack_linked(list_linked *list)`

Cette procédure affiche les éléments de la pile depuis la tête. Chaque élément doit être suivi d'un retour à la ligne.

Si la liste donnée en paramètre est vide, la procédure ne fait rien.
Le format attendu est le suivant :

***`elt`*`\n`**

Ce qui donnerait cet affichage pour la liste suivante :

```
$ ./liste_ll_example1
42
21
8
24
64
$
```

pos	1	2	3	4	5
elt	42	21	8	24	64

`int get_head_stack_linked(list_linked *list)`

Cette fonction renvoie l'élément en tête de la pile, c'est-à-dire celui à la première position.
Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer -1.

`int clear_stack_linked(list_linked *list)`

Cette fonction vide la pile de tous ses éléments, puis renvoie le nombre d'éléments qui ont été supprimés.
Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.

7 Exercice 4 - Files

Nom du(es) fichier(s) :	**queue_linked.c**
Répertoire :	**login.x-MiniProjet1/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une des structures de base : les files à base de listes chaînées.

Vous devez écrire plusieurs fonctions permettant de gérer des files à base de listes chaînées, c'est-à-dire des files exploitant des listes à base de pointeurs. Un fichier **queue_linked.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devrez réutiliser la structure **list_linked** telle quelle issue du fichier **list_linked.h** : **vous ne devez pas créer de nouvelle structure**. La liste chaînée ainsi représentée contient deux champs exactement : *elt* (l'élément inséré) et *next* (pointeur vers le maillon suivant de la chaîne).

Vous devez implémenter les fonctions suivantes :

```
list_linked *enqueue_queue_linked(list_linked *list,
                                  int elt);
list_linked *dequeue_queue_linked(list_linked *list);

int length_queue_linked(list_linked *list);
int is_empty_queue_linked(list_linked *list);

void print_queue_linked(list_linked *list);

int get_head_queue_linked(list_linked *list);

int clear_queue_linked(list_linked *list);
```

`list_linked *enqueue_queue_linked(list_linked *list, int elt)`

Cette fonction enfile un nouvel élément, c'est-à-dire qu'elle ajoute un élément en fin de file, c'est-à-dire qu'elle ajoute un élément en dernière position de la liste chaînée donnée en paramètre.

Si la file est vide, il faut y créer un premier élément.
Si l'élément *elt* donné en paramètre est inférieur à 1, la fonction doit renvoyer un pointeur **`NULL`** sans rien modifier dans la liste.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée. En cas d'erreur (pas assez de mémoire), cette fonction doit renvoyer un pointeur **`NULL`**.

`list_linked *dequeue_queue_linked(list_linked *list)`

Cette fonction défile l'élément en tête, c'est-à-dire qu'elle supprime l'élément en première position de la liste chaînée donnée en paramètre.

Si la liste donnée en paramètre est vide, cette fonction doit renvoyer un pointeur **`NULL`**.

La fonction doit retourner un pointeur vers la tête de liste, ou vers l'éventuelle nouvelle tête de liste si celle-ci a été modifiée.

`int length_queue_linked(list_linked *list)`

Cette fonction renvoie la taille de la file donnée en paramètre, c'est-à-dire le nombre d'éléments présents dans la liste.

Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.

`int is_empty_queue_linked(list_linked *list)`

Cette fonction teste si la file est vide ou non.

Si la liste est vide, c'est-à-dire si le pointeur est **`NULL`**, il faut renvoyer 1 (l'équivalent de *true* en C), sinon, si la liste n'est pas vide, il faut renvoyer 0 (l'équivalent de *faux* en C).

`void print_queue_linked(list_linked *list)`

Cette procédure affiche les éléments de la file depuis la tête. Chaque élément doit être suivi d'un retour à la ligne.

Si la liste donnée en paramètre est vide, la procédure ne fait rien.
Le format attendu est le suivant :

***`elt`*`\n`**

Ce qui donnerait cet affichage pour la liste suivante :

```
$ ./liste_ll_example1
42
21
8
24
64
$
```

pos	1	2	3	4	5
elt	42	21	8	24	64

`int get_head_queue_linked(list_linked *list)`

Cette fonction renvoie l'élément en tête de la file, c'est-à-dire celui à la première position.
Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer -1.

`int clear_queue_linked(list_linked *list)`

Cette fonction vide la file de tous ses éléments, puis renvoie le nombre d'éléments qui ont été supprimés.
Si la liste donnée en paramètre est **`NULL`**, la fonction doit renvoyer 0.