TP5: Structures

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

Consignes

Vous trouverez sur Cahier de Prépa une archive contenant des fichiers pour ce TP. Avant de rendre votre TP, n'oubliez pas de supprimer les exécutables, et de ranger vos exercices dans des dossiers séparés. Et comme toujours, **CAT** : **Commentaires**, **Assertions**, **Tests**.

Structures

Exercice 1

Cet exercice sert à vous familiariser avec les types struct. Si vous pensez être à l'aise, vous pouvez **le sauter** et passer directement à l'exercice 2.

Une université maintient une base de données de ses cours. Pour chaque cours, les administrateurs veulent en stocker, l'intitulé, le/la professeur/e en charge, la salle et le jour. On propose, en C, d'utiliser la structure suivante :

```c
struct cours {
  char intitule[50];
  char prof[50];
  int salle; // identifiant unique de la salle
  int jour;  // 0 (lundi) - 6 (dimanche)
};
```

Q1. Quelle est la taille nécessaire pour stocker un élément du type `struct cours` ?

L'archive du TP contient un fichier `universite.c`, dans lequelle se trouve la définition de ce type struct. La fonction main montre comment y créer un élément de deux manières :

- en initialisant immédiatement les paramètres ;
- en déclarant la variable, puis en remplissant les paramètres, en deux temps.

Q2. Ajouter un cours d'informatique ayant lieu le vendredi, dans la salle d'identifiant 49669.

Q3. Créer une fonction `void afficher_infos(cours_t* c)` prenant en entrée un **pointeur** vers un `cours_t`, et affichant son contenu. La tester sur les structures.

Q4. Ajouter à la structure de cours un attribut booléen indiquant si c'est un TP ou non. Modifiez le main et la fonction `afficher_infos` en conséquence.

Q5. Écrire une fonction `cours_t* creer_cours(char* intitule, char* prof, int salle, int jour)` qui alloue dans le tas un cours, et initialise ses attributs avec les paramètres de la fonction. La tester dans le main (en n'oubliant pas de libérer la mémoire !).

Exercice 2

Dans cet exercice, on allouera toute la mémoire nécessaire dans la pile. Vous n'avez donc jamais besoin d'utiliser `malloc`.

Nous allons implémenter un programme permettant de passer une commande dans un (faux) restaurant. Le menu du restaurant contient plusieurs éléments, qui seront représentés par le type suivant :

```
struct menu_elem {
  char[50] nom;
  float prix_unite;
  bool vegan;
};
typedef struct menu_elem menu_elem_t;
```

Par exemple :

```
menu_elem_t e_1 = {
  .nom = "Brocoli au gingembre",
  .prix_unite = 7.99,
  .vegan = true,
};
```

```
menu_elem_t e_2 = {
  .nom = "Hamburger du chef",
  .prix_unite = 9.99,
  .vegan = false,
};
```

Lorsque l'on écrira des fonctions qui manipulent des éléments de ce type, on mettra des paramètres de type `menu_elem_t*` (donc des pointeurs). En effet, si l'on utilisait directement le type `menu_elem_t`, on devrait recopier les 3 attributs des arguments vers les paramètres, ce qui ferait recopier $50 + 4 + 1 = 55$ octets (même 58 à cause des contraintes d'alignement mémoire), alors qu'en passant par les pointeurs, on ne recopie que 8 octets, ceux du pointeur passé en argument.

L'archive du TP contient un fichier `menu.c` contenant la définition de cette structure.

Q1. Créer une fonction `void test()`, vide pour le moment, ainsi qu'une fonction `main`, qui ne fait qu'appeler la fonction `test`.

Q2. Dans la fonction de test, créez un élément de menu et affichez son prix. Vérifiez que le programme compile et s'exécute correctement.

Dans la suite, tout le code ira dans la fonction test, sauf dans les dernières questions où l'on implémentera le main. Au fur et à mesure que vos tests fonctionnent, vous pouvez les commenter afin de ne pas avoir à tester tout à chaque exécution, mais **ne supprimez rien** !

Q3. Écrire une fonction `affiche_elem(menu_elem_t* e)` qui affiche un élément de menu, sous le format suivant :

```
Brocoli au gingembre (V): 7.99€
Hamburger du chef: 7.99€
```

Le "`(V)`" indique un plat végan. Ajoutez de quoi tester cette fonction dans la fonction `test`. **Indication :** le format `"%.2f"` permet d'afficher un flottant avec exactement deux décimales.

Q4. (Optionnelle) quel sorte de problèmes peuvent survenir en utilisant un flottant pour stocker le prix des plats ? Comment pourrait-on y remédier ?

Le menu du restaurant est représenté par le type suivant :

```c
#define MAX_ELEMS 128

// Invariant: nb_elems <= MAX_ELEMS
struct menu {
  menu_elem_t elements[MAX_ELEMS];
  int nb_elems; // nombre d'éléments du menu.
};
typedef struct menu menu_t;
```

Un menu peut donc avoir jusqu'à 128 éléments distincts. Notons que la structure `menu_t` n'est rien de plus qu'un tableau, mais l'utilisation d'une structure permet d'encapsuler le tableau et sa taille.
La condition `nb_elems <= MAX_ELEMS` est un **invariant de structure** : elle doit être valide tout au long de l'existence d'un élément de type `menu_t`.

Q5. Ajoutez la macro `MAX_ELEMS` ainsi que la structure `menu_t` dans `menu.h`.

Q6. Écrire une fonction `void affiche_menu(menu_t* m)` permettant d'afficher un menu, en numérotant les éléments. Ajoutez du code à votre fonction de test pour que votre programme crée et affiche le menu suivant :

```
Menu:
1. Pâtes au saumon : 11.99€
2. Curry de butternut (V) : 8.99€
3. Soupe de poulet : 15.99€
```

Une commande sera modélisée par un tableau `int commande[MAX_ELEM]`, où la case `commande[i]` contient le nombre d'exemplaires de l'élément d'indice i. Attention, il y a un décalage de 1 entre le numéro affiché pour un élément et son véritable indice. Par exemple, sdans le menu précédent, si un utilisateur commande 2 pâtes au saumon, c'est bien la case `commande[0]` qui vaudra 2.

Q7. Écrire une fonction `void init_commande(int* commande, int nb_elems` qui initialise une commande en mettant toutes les cases à 0. Le paramètre `nb_elems` sera le nombre d'éléments du menu.

Pour prendre une commande, l'utilisateur/rice va écrire le numéro des éléments à commander dans le terminal, et le programme va les lire. La commande s'arrêtera lorsqu'autre chose qu'un entier, par exemple '#', est lu dans le terminal. On rappelle que la fonction `scanf` renvoie le nombre d'objets qui ont été lus, et donc que si l'on écrit

```c
scanf("%d", &c);
```

et que le terminal contient un '#', la fonction ne modifie pas la valeur de `c` et renvoie 0, alors que si le terminal contient un entier n, la fonction change la valeur de `c` en n et renvoie 1.

Q8. Écrire une fonction `void prendre_commande(int* commande, int nb_elems)` qui prend la commande de l'utilisateur, selon le format expliqué au dessus :

```
Choisissez les plats à commander en notant leurs numéros,
puis '#' pour marquer la fin:
1 3 1 1 2 1 2 #
```

Q9. Écrire une fonction `void resume_commande(int* commande, menu_t* m)` affichant un résumé de la commande :

```
Vous avez commandé:
4x Pâtes au saumon
2x Curry de butternut
1x Soupe de poulet
```

ainsi qu'une fonction `float total_commande(int* commande, menu_t* m)` calculant le prix total à payer pour la commande. Ajouter ces fonctions au main pour afficher le total à l'utilisateur. *Attention, ces fonctions vous demanderont d'utiliser à la fois des notations avec des* `.` *et avec des* `->`.

Écrire les éléments du menu directement dans le code n'est pas pratique, et demande de recompiler le programme à chaque fois que l'on change le menu. Nous allons implémenter de quoi permettre au programme de lire le menu depuis un fichier.

Q10. Écrire une fonction `ajouter_element(menu_t* m, char* nom, float prix_unite, bool vegan)` permettant d'ajouter un élément à un menu.

Les fichiers de menu seront des fichiers textes, contenant une ligne par élément, chaque ligne contenant le prix, un booléen (0 ou 1) indiquant si le plat est végan ou non, puis le nom du plat. Par exemple, pour le menu à trois éléments précédent, on aura le fichier ci-contre.

```
11.99 0 Pâtes au saumon
8.99 1 Curry de butternut
15.99 0 Soupe de poulet
```

Q11. Écrire une fonction `bool lire_element(FILE* f, menu_elem_t* e)` qui lit dans le fichier `f` les informations d'un seul élément de menu, et stocke les informations dans la structure pointée par `e`. La fonction renverra un booléen indiquant si la lecture a atteint la fin du fichier (EOF) ou non.

Q12. Écrire une fonction `void lire_menu(char* nom_fichier, menu_t* m)` qui lit les informations d'un menu dans un fichier et stocke les informations dans la structure pointée par `m`.

Q13. Remplir la fonction `main` pour que l'on pusse exécuter le programme en mettant en argument le fichier du menu. Exemple d'exécution :

```
fredfrigo@ubuntu:~$ gcc menu.c -o menu -Wall -Wextra
fredfrigo@ubuntu:~$ ./menu menu_test.txt
Menu:
1. Pâtes au saumon : 11.99€
2. Curry de butternut (V) : 8.99€
3. Soupe de poulet : 15.99€
Choisissez les plats à commander en notant leurs numéros,
puis '#' pour marquer la fin:
1 3 1 1 2 1 2 #
Vous avez commandé:
4x Pâtes au saumon
2x Curry de butternut
1x Soupe de poulet
Total: 81.93€
```

Exercice 3: Optionnel

Le fichier `equipe.c` de l'archive du TP contient deux définitions de structures modélisant des étudiants et des équipes d'étudiants.

Dans cet exercice, on allouera systématiquement les structures créées dans le tas, en utilisant malloc. Il faudra donc bien libérer toute la mémoire. Vérifiez vos programmes avec valgrind pour détecter les fuites mémoires (et les autres erreurs).
Pour les deux premières questions uniquement, ne libérez pas la mémoire.

Q1. Rajoutez une fonction main, dans laquelle vous allez créer une équipe de 3 personnes :

- Camille, 23 ans
- Leila, 20 ans (la capitaine)
- Thibault, 22 ans

Q2. Écrire deux fonctions d'affichage `void print_etu(etu_t* e); void print_equipe(equipe_t* e)`.

Q3. Implémentez deux fonctions `void free_etu(etu_t* e); void free_equipe(equipe_t* e)` permettant de libérer la mémoire allouée, et rajoutez au main de quoi libérer la place allouée. N'oubliez pas de vérifier les fuites mémoires avec valgrind.

Implémentons maintenant une fonction `void agrandir_equipe(equipe_t* dst, equipe_t* src)` qui rajoute les membres de l'équipe source à l'équipe destination. La/le capitaine de la nouvelle équipe sera la/le capitaine de la plus grande des deux équipes initiales.

Cette description de fonction est vague, et en particulier n'explique pas ce qui se passe au niveau de la mémoire : la fonction duplique-t-elle les structures des membres de l'équipe source, ou bien les deux équipes vont-elle partager des pointeurs communs vers les membres rajoutés ? La structure source est-elle encore valide après l'opération ? Ainsi de suite...

Q4. Lisez le code suivant : quelles erreurs peuvent survenir selon les différents choix d'implémentation de la fonction `agrandir_equipe` ?

```
1  equipe_t* e1;
2  equipe_t* e2;
3  /*
4    ...
5    Code pour allouer et remplir e1 et e2
6    ...
7  */
8  agrandir_equipe(e1, e2);
9
10 print_equipe(e1);
11 free_equipe(e1);
12
13 print_equipe(e2);
14 free_equipe(e2);
```

Q5. Choisissez une spécification plus précise et implémentez la fonction en la respectant. Précisez dans le commentaire de documentation le comportement au niveau de la mémoire : une fois que l'on a appelé la fonction `agrandir_equipe`, y a t-il des structures qui deviennent invalides ? Et comment doit-on libérer la mémoire ?

Exercice 4: Listes chaînées

Les structures en **listes chaînées** sont très fréquente en informatique, et sont à la base de nombreuses structures plus complexes que l'on verra plus tard en cours.
Une liste chaînée est constituée de **maillons** reliés entre eux. Chaque maillon connaît son voisin de droite :

Le premier maillon d'une chaîne est appelé sa **tête**.
Il y a plusieurs manières d'implémenter des listes chaînées, ici on en étude une avec deux structures : une pour les maillons, et une pour la liste elle-même :

```c
struct maillon {
  struct maillon* suivant;
  int contenu; // donnée stockée sur le maillon
};
typedef struct maillon maillon_t;

struct liste {
  maillon_t* tete;
};
typedef struct liste liste_t;
```

La fin d'une liste chaînée est marquée par le pointeur nul : le dernier maillon a pour suivant `NULL`. Ceci permet de **parcourir une liste chaînée** avec une boucle :

```c
// l est une liste chaînée définie précédemment
maillon_t* m = l->tete;
while (m != NULL){
  // traiter le maillon m
  m = m->suivant;
}
```

L'archive du TP contient un fichier `liste_chainee.c` contenant la définition de ces structures, d'une fonction créant une liste d'une taille donnée contenant des entiers aléatoires, et d'une fonction affichant le contenu d'une liste.

Q1. Lisez la documentation des deux fonctions, puis créez un main, ou vous créerez une liste aléatoire de taille 10 et afficherez son contenu. Ne libérez pas la mémoire pour le moment.

Q2. Au crayon à papier, simulez l'exécution de `liste_aleatoire(3)`, et dessinez la mémoire au fil des instructions. (Il n'y a pas besoin de rendre cette question)

Q3. Implémentez une fonction `liste_range` qui prend en entrée un entier n et renvoie une liste chaînée dont les maillons contiennent $0, 1, \ldots, n-1$ dans cet ordre.

Q4. Implémentez une fonction permettant de libérer la mémoire d'une liste chaînée. *Faites des schémas si vous avez du mal !*

Q5. Écrivez une fonction `bool liste_recherche(int x, liste_t* L)` qui détermine si la liste L contient au moins une fois l'élément x.

Q6. Écrivez une fonction qui ajoute un élément à la fin d'une liste chaînée, et une autre qui ajoute un élément au début. Faites des dessins pour voir comment les différents pointeurs sont modifiés/ajoutés. En particulier, faites attention à ce qui peut arriver à l'attribut `tete` de la liste dans les deux cas.

Q7. (Optionnel) Écrivez une fonction qui prend en entrée une liste chaînée et un entier, et retire de la liste le premier maillon contenant cet entier. A nouveau, faites des dessins pour voir comment recoller les morceaux.

Exercice 5: Optionnel

L'archive du TP contient un fichier `promo_2014.txt` qui contient les noms des élèves qui étaient à votre place il y a 10 ans, avec le métier que chacun et chacune exerce à présent. Le fichier est formaté comme suit :

- Sur la première ligne, un entier n indiquant le nombre d'élèves
- Puis, n blocs de 3 lignes, chaque bloc correspondant à un élève, avec :
  - Sur une ligne, le prénom suivi du nom de famille (maximum 50 caractères)
  - Sur la ligne suivante, la date de naissance sous le format `JJ MM AAAA`
  - Sur la dernière ligne, le métier de cet élève aujourd'hui (maximum 50 caractères).

On propose de stocker les informations des élèves dans la structure suivante :

```c
struct eleve{
  char nom_complet[51];
  int jour, mois, annee; // date de naissance
  char metier[51];
};
typedef struct eleve eleve_t;
```

Nous allons commencer par écrire une fonction `eleve_t* lire_eleve(FILE* f)` qui lit les informations d'un/e élève dans le fichier f, et renvoie une structure contenant ces informations. Cette fonction renverra NULL si la fin du fichier est atteint.

Attention : lorsque vous scannez un entier avec le format `"%d"`, s'il y a un retour à la ligne après l'entier lu, il n'est pas consommé. Par exemple :

```c
fscanf(f, "%d", &n);
len = getline(ligne, &taille, f);
```

Si l'on exécute ce code et que le fichier `f` contient :

```
5
bonjour
```

alors la ligne scannée par `getline` sera vide, car après le `fscanf`, la tête de lecture du fichier se situe juste après le 5, avant le retour à la ligne.
Pour corriger cela, il faut consommer à la main le retour à la ligne. On peut donc utiliser un `char` comme poubelle :

```c
fscanf(f, "%d", &n);
char poubelle;
fscanf(f, "%c", &poubelle); // consomme le '\n'
len = getline(ligne, &taille, f);
```

Pour cet exercice, cette méthode fonctionnera bien car dans le fichier fourni, les retours à la ligne sont bien encodés par le caractère `'\n'`. Cependant sur certains systèmes, notamment Windows, les retours à la ligne ne se font pas avec le caractère `'\n'` mais avec DEUX caractères : `'\r'` et `'\n'`. Le premier s'appelle le retour chariot, un terme qui vient des machines à écrire et qui désigne le fait de remettre le curseur tout à gauche du papier (ou du terminal pour un ordinateur). Pour un fichier créé sous Windows, il faudrait donc faire :

```
fscanf(f, "%d", &n);
char poubelle;
fscanf(f, "%c", &poubelle); // consomme le '\r'
fscanf(f, "%c", &poubelle); // consomme le '\n'
len = getline(ligne, &taille, f);
```

Q1. En prenant en compte les informations précédentes, implémenter la fonction `eleve_t* lire_eleve(FILE* f)` permettant de lire dans un fichier les informations d'un/e élève.

Q2. Écrire une fonction `eleve_t** lire_promo(char* filename, int* n)` qui renvoie un **tableau de pointeurs de structures** correspondant à la liste des élèves stockés dans le fichier de nom `filename`. Cette fonction stockera également dans `*n` le nombre d'élèves.

Q3. Écrivez un programme qui permet d'afficher d'une manière un peu jolie les informations des élèves du fichier `promo_2014.txt`. N'oubliez pas de bien libérer **TOUTE** la mémoire allouée !

Livre dont vous êtes le héros ou la héroïne

Exercice 6: Optionnel

Un ***Livre dont vous êtes le héro / la héroïne***, où LDVELH, est un type de livre interactif A chaque page, on est amené à faire un choix, et selon l'option choisie, à se rendre à une page particulière. Ainsi, un LDVELH ne se lit pas dans l'ordre mais en sautant de page en page en suivant les choix faits, en commençant à la page 1.

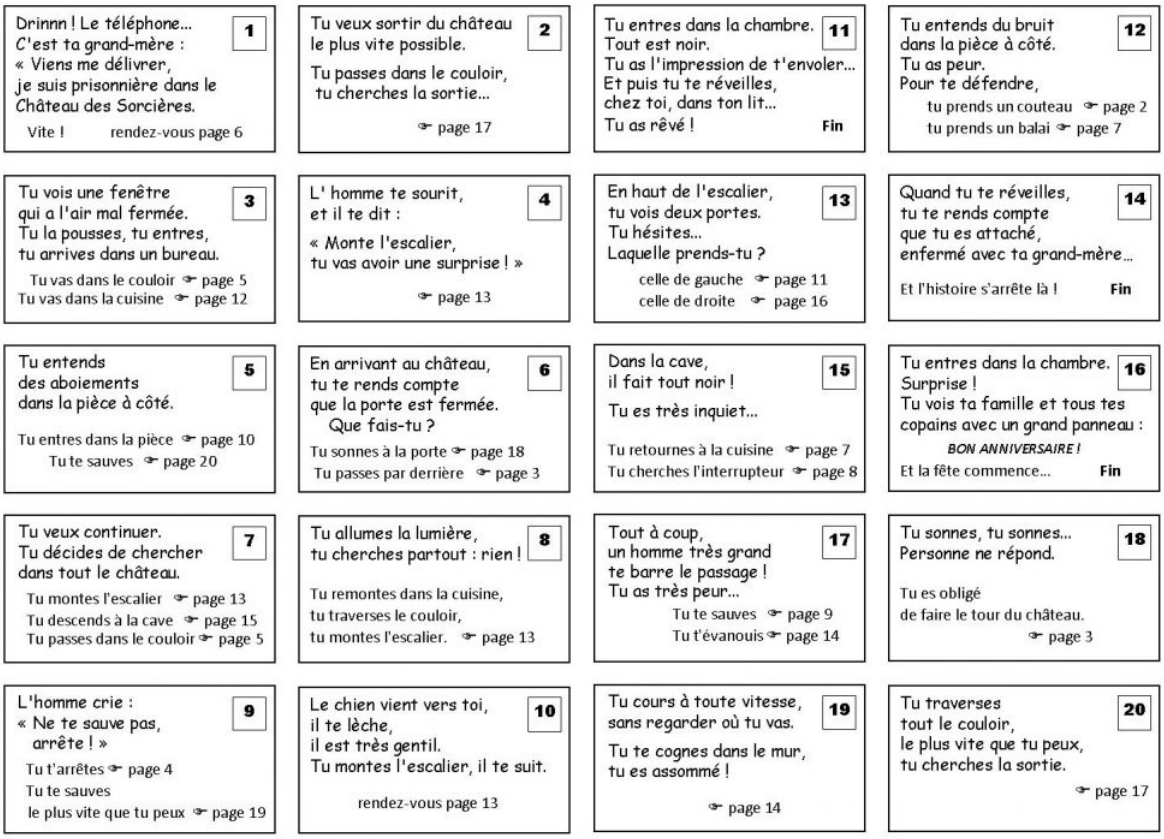

Figure 1 – Un LDVELH assez court

Le but de l'exercice est d'écrire un programme qui peut lire un LDVELH écrit dans un fichier texte sous un format bien spécifique, et en faire une version interactive, où l'utilisateur peut rentrer ses choix au clavier.

Pour représenter un LDVELH, on propose la modélisation suivante :

- Un **livre** est constitué de plusieurs pages, indexées par $0, 1, \dots N-1$, où N est le nombre de pages
- Une **page** est constituée d'un texte, ainsi que de plusieurs choix, indexés par $0, 1, \dots K-1$, où K est le nombre de choix
- Un **choix** est constitué d'un texte ainsi que du numéro de la page suivante lorsque l'on suit ce choix.

Pour représenter sous format texte un LDVELH, on écrit dans un fichier texte :

- Sur la première ligne, un entier n indiquant le nombre de pages total du livre
- Pour chaque page, on écrit :
  - Sur une ligne, le nombre de choix
  - Le texte de cette page sur une ligne
  - Pour chaque choix de la page, on écrit :
    - Le texte du choix sur une ligne
    - Le numéro de la page suivante

Une page marque la fin de l'aventure si, et seulement si, elle a 0 choix ; il peut y avoir plusieurs fins. La page d'indice 0 est la première page du livre.

Par exemple, les premières lignes du fichier "mamie.txt" représentant le LDVELH Fig. 1 seraient les suivantes (les commentaires ne seraient pas présents dans le fichier) :

```
20    // nombre de pages
1  // début de la page 0: un seul choix
Drinnn ! Le téléphone... C'est ta grand mère: "[...]" // texte de la page
Vite !  // texte de l'unique choix
5      // le choix mène à la page 5
1  // début de la page 1: un seul choix
Tu veux sortir du chateau [...] tu cherches la sortie... // texte de la page
    // texte du choix: cette ligne est vide
16  // le choix mène à la page 16
2  // début de la page 2: deux choix
Tu vois une fenêtre ... tu arrives dans un bureau. // texte de la page
Tu vas dans le couloir // texte du choix 1
4   // choix 1 -> aller page 4
Tu vas dans la cuisine // texte du choix 2
11  // choix 2 -> aller page 11
...
```

En C, on propose les structures suivantes pour représenter ces informations :

```c
struct CHOIX{
  char* texte;
  int page_suivante;
}

struct PAGE{
  char* texte;
  int n_choix;
  struct CHOIX** choix; //tableau de pointeurs vers des choix
}

struct LIVRE{
  int n_pages;
  struct PAGE** pages //tableau de pointeurs vers des pages
}
```

Vous pouvez utiliser typedef pour renommer ces structs si vous le souhaitez.

Q1. Écrivez un fichier text.txt avec un exemple minimal de LDVELH (au moins 3 pages, au moins une page avec plusieurs choix)

Q2. Écrivez une fonction `struct CHOIX* generer_choix(FILE* f)` qui, étant donné un fichier f, dont on suppose que la tête de lecture est positionnée à un endroit contenant un choix au format décrit plus haut, renvoie un pointeur de structure de type `struct CHOIX` en utilisant les données lues dans le fichier.

Q3. Écrivez selon le même principe deux fonctions permettant de lire une page et un livre depuis un fichier :

```
1 struct PAGE* generer_page(FILE* f);
2 struct LIVRE* generer_livre(FILE* f);
```

Q4. Écrivez une fonction `void free_livre(struct LIVRE* livre)` qui, étant donné un pointeur vers une structure struct LIVRE, libère toute la mémoire allouée à ce livre, y compris pour le texte. Vous pouvez écrire des fonctions auxiliaires pour vous aider (par exemple pour libérer les structures intermédiaires).

Q5. En utilisant les fonctions précédentes, écrivez un programme prenant comme premier argument un nom de fichier texte, dont on suppose qu'il contient un LDVELH sous format texte, et qui permet à l'utilisateur de jouer/lire le livre.

L'exécution du programme doit ressembler à :

```
guillaume@MSI:~/prof/code/cyoa$
guillaume@MSI:~/prof/code/cyoa$ ./cyoa livre.txt
Nous sommes vendredi matin. Demain, il y a DS d'informatique. La journée doit se dérouler sans accroc afin que vous
ayez le temps de réviser ce soir, de dormir, et donc que vous puissiez obtenir une bonne note au DS. Il est 10h, le
cours de physique de Mme Goutelard vient de se terminer. Que faites-vous ?

  1. Vous allez en cours d'informatique.

  2. Vous n'allez pas en cours d'informatique, il fait beau et vous préférez aller vous balader sur les quais de la
Garonne.

Votre choix: 2
Vous n'allez pas en cours d'informatique, et vous ratez des informations capitales pour réviser. Vous n'êtes pas bie
n préparé pour le DS. GAME OVER
```

Vous trouverez dans l'archive du TP un fichier "livre.txt" écrit sous le format décrit plus haut, avec lequel vous pouvez tester votre programme une fois qu'il fonctionne sur votre exemple minimal. Vous y trouverez également un livre `multiverse.txt` écrit par Corentin, un élève de la promo 2022, qui a accepté de le laisser pour les promos suivantes.

Enfin, vous trouverez sur perso.ens-lyon.fr/guillaume.rousseau/ldvelh/ une version graphique du logiciel que vous avez écrit, permettant de visualiser les LDVELH sous forme d'un **graphe** (voir chapitre 13) : chaque page est représentée par un sommet, et un choix d'une page A permettant d'accéder à une page B est représenté par une arête de A à B.

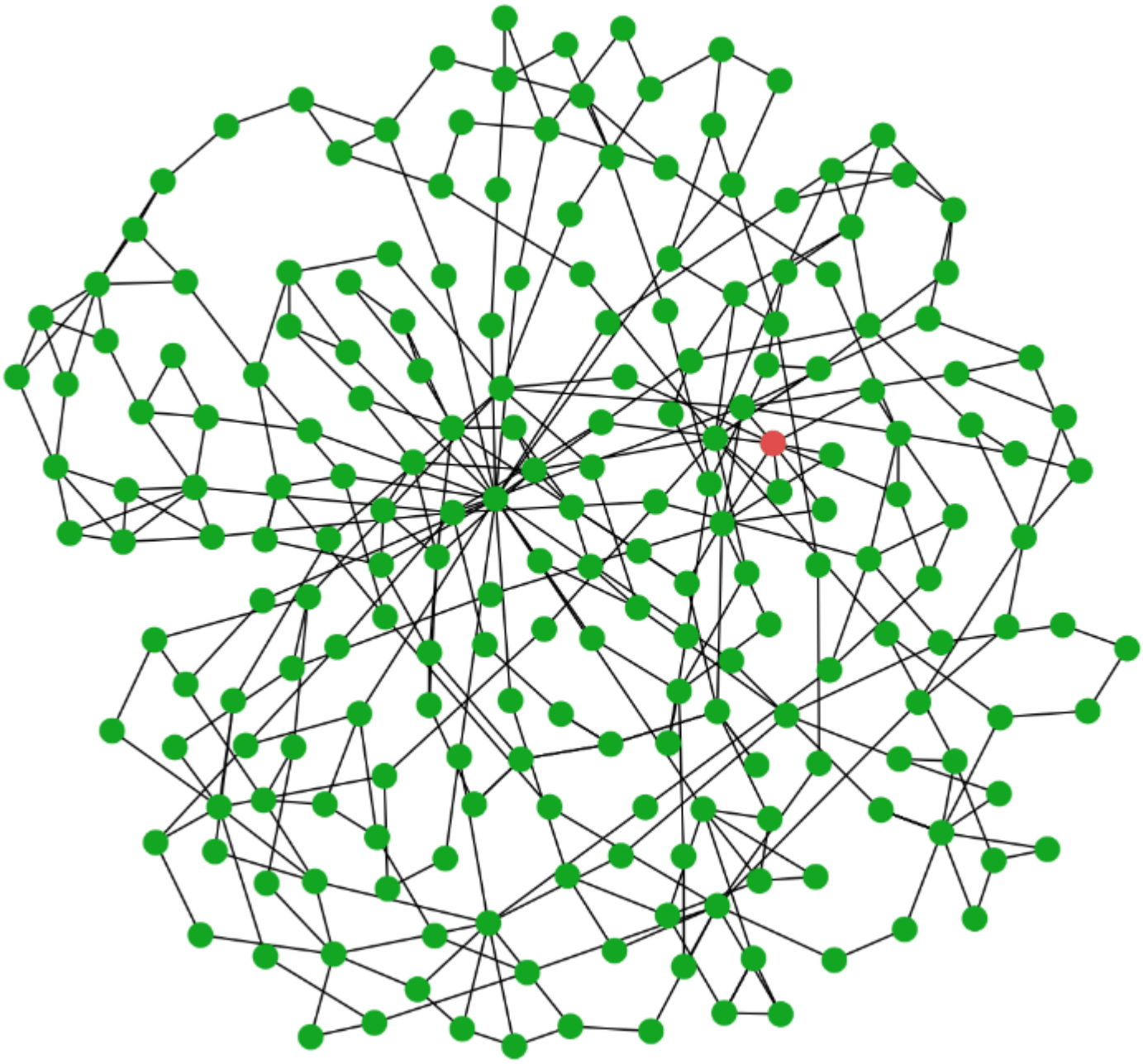

FIGURE 2 – Graphe du livre `multiverse.txt`