Implémentation d'un gestionnaire de ludothèque

Alexis MATHEY
Célestin VALLOIS

Introduction

Objectifs

Il nous a été demandé d'écrire un programme permettant la gestion d'une ludothèque numérique. Dans cette ludothèque, chaque jeu est identifié par son nom, son genre (ambiance , coopératif, plateau, RPG ou hasard), un nombre de joueurs (mini et maxi) ainsi qu'une durée de jeu moyenne. Le gestionnaire va nous permettre de gérer plusieurs ludothèques automatiquement triées par ordre alphabétique des noms des jeux. Nous pourrons ajouter et supprimer des jeux dans ces ludothèques, afficher leur contenu et effectuer une recherche par genre, nombre de joueurs ou durée.

Réalisation

L'organisation de notre projet codé en C est la suivante :

- Un fichier d'entête tp3.h contenant les déclarations des structures et fonctions de bases ;

- Un fichier source tp3.c contenant la définition de chaque fonction ;

- Un fichier source main.c contenant le programme principal.

Problèmes rencontrés

Au cours de notre TP, nous n'avons pas rencontré de problème majeur/spécifique. Il a fallu essayer de prendre en compte tous les cas possibles pour chaque fonction. Comme par exemple les espaces et la casse des données rentrées par l'utilisateur. Nous avons allouée dynamiquement toute la mémoire utilisées qui est libérée lorsqu'on quitte le programme.

Nos choix de programmation

Les structures de données

Dans notre fichier header tp3.h, nous avons déclaré 3 types à partir de 3 structures de données que nous avons créés :

- **genre_jeu** qui est une énumération des différents genre de jeu possible pour les jeux de la ludothèque ;

- **t_jeu** qui reprend la structure jeu : ce type contient donc toutes les informations concernant un jeu, ainsi que le jeu qui le suit dans la ludothèque pour permettre la manipulation des listes chaînées ;

- **t_ludotheque** qui est issu de la struct ludotheque : ce type reprend le nombre de jeu d'une ludothèque et un pointeur vers le premier jeu de cette ludothèque.

Ce type de structure implique évidemment une gestion dynamique de la mémoire à l'aide des fonctions **malloc()** lors de la création d'un élément et **free()** lors de la suppression.

Les fonctions

Le menu

Pour accéder aux fonctions, nous avons créé un menu proposant les choix suivants :

- Créer une ludothèque (**creer_ludotheque()**)
- Ajouter un jeu dans une ludothèque (**creer_jeu()** et **ajouter_jeu()**)
- Supprimer un jeu (**retirer_jeu()**)
- Afficher le contenu d'une ludothèque (**afficher_ludotheque()**)
- Effectuer une recherche de jeu (par genre, durée ou nombre de joueurs) (**requete_jeu()**)
- Fusionner deux ludothèques existantes en une (**fusion()**)
- Quitter

Ce menu s'affiche en boucle tant que l'utilisateur ne sélectionne pas l'option « Quitter ».

Création de ludothèque - creer_ludotheque()

Cette fonction crée une ludothèque vide (donc nb_jeu =0 et debut = NULL) et renvoie l'adresse de cette ludothèque (un pointeur sur le type **t_ludotheque**) ou bien NULL si la fonction a échoué.

La ludothèque crée est allouée dynamiquement à l'aide de la fonction **malloc()**. Nous testons ensuite si l'allocation a bien marché à l'aide d'un IF, si ce n'est pas bon on renvoie NULL sinon on renvoie l'adresse de la ludothèque crée.Tout ceci se fait en temps constant.

Complexité : *0(1)*

Ajout d'un jeu - creer_jeu() et ajouter_jeu()

Cette opération se fait en deux étapes : la création du jeu puis l'insertion dans la ludothèque.

La création du jeu avec **creer_jeu()** est une opération simple dans laquelle on alloue la mémoire pour la structure de donnée puis on initialise tous les champs du jeu avec les valeurs souhaitées. Le fonction retourne un pointeur vers le jeu créé. Ceci se fait en temps constant.

Ensuite, on insère le jeu dans la ludothèque en respectant l'ordre alphabétique avec la fonction **ajouter_jeu()**. Nous avons donc plusieurs cas :

1. Ludothèque vide ou nom du jeu est inférieur alphabétiquement à celui du premier jeu de la ludothèque : On insère le jeu au début de la ludothèque. (On a créé une fonction **inserer_debut()** pour plus de lisibilité et compréhension)
2. Ludothèque non vide : on parcourt la ludothèque jusqu'à ce qu'on rencontre un jeu d'ordre alphabétique supérieur au jeu à insérer et on insère à cet endroit (fonction **inserer**. Si le jeu est déjà dans la ludothèque, on n'insère rien.

L'insertion dans une liste chaînée se fait en temps constant donc :

- Dans le meilleur des cas (lorsque la ludothèque est vide) : on insère directement donc *Ω(1)*
- Dans le pire des cas (le jeu est inséré en fin de liste) : *O(n)*

Complexité : Ω(1) et 0(n)

Suppression d'un jeu - retirer_jeu()

A partir d'un nom de jeu et d'une ludothèque, cette fonction recherche ce jeu dans la ludothèque et le supprime si elle le retrouve, elle renvoie ensuite 1 si la suppression a réussi, 0 sinon. Il y a donc deux étapes :

- Le recherche : comme vu précédemment, elle est de complexité *Ω(1) et O(n).*
- La suppression : on libère la mémoire avec **free()** et on modifie le prédécesseur et le successeur pour ne pas briser la liste chaînée. Cette opération est en temps constant.

$$\textit{Complexité : } \Omega(1) \text{ et } 0(n)$$

Afficher une ludothèque - afficher_ludotheque()

Pour afficher le contenu d'une ludothèque, on la parcourt intégralement et on affiche les caractéristiques de chaque élément. De cette façon, nous sommes en complexité *O(n).*

Pour des raisons de lisibilité et d'esthétique, nous avons fait des ajouts à cette fonction pour que le résultat soit un tableau régulier avec des titres de colonne et des séparateurs entre les colonnes. Nous utilisons donc des fonctions d'affichage d'espaces, de tirets ou de mots. Ces fonctions s'effectuent toutes en temps constant donc elles ne changent pas la complexité de l'algorithme. D'où :

$$\textit{Complexité : } 0(n)$$

On pourra noter la fonction **afficher_genre()** qui permet de prend le genre du jeu (de type **genre_jeu** (énumération) en paramètre et qui retourne ce genre dans une chaîne de caractères.

Supprimer une ludothèque – supprimer_ludotheque()

Cette fonction supprime une ludothèque et tous les jeux la composant. Pour ce faire, on parcourt toute la ludothèque en supprimant tous ces jeux à l'aide de **free()**. Une fois fait, la ludothèque n'a plus de jeu et nous pouvons la supprimer à son tour à l'aide de **free()** encore une fois. Ces opérations se font en temps constant donc :

$$\text{Complexité : } 0(1)$$

Recherche d'un jeu - requete_jeu()

Cette fonction recherche dans une ludothèque des jeux correspondant aux critères donnés en paramètres (genre, durée ou nombre de joueurs) et retourne une nouvelle ludothèque contenant ces jeux. L'utilisateur peut entrer -1 pour ignorer un critère.

On parcourt donc toute la ludothèque d'origine et on insère les jeux correspondant aux critères dans une nouvelle ludothèque avec **ajouter_jeu()**. On retourne un pointeur vers la nouvelle ludothèque.

Puisqu'on parcourt toute la ludothèque (donc n boucles) et que l'ajout d'un jeu dans la nouvelle ludothèque est en *0(n')* avec n' ≤ n et et n' qui augmente de 0 à n au fur et à mesure qu'on lui ajoute des jeux. Dans le pire cas, on ajoute tous les jeux à la fin de la nouvelle ludothèque ce qui donne :

$$\textit{Complexité : } O\left(\sum_{i=1}^{n} i\right)$$

Fusion de ludothèques – fusion()

Cette fonction permet la fusion de 2 ludothèques entrées en argument en une nouvelle ludothèque (**créer_ludotheque()**). Il faut donc prendre en compte plusieurs conditions : la ludothèque de fusion doit aussi être en ordre alphabétique et ne doit pas posséder de doublons.

Nous avons donc 3 cas lorque nous ajoutons un jeu :

- Le jeu de la ludothèque 1 est inférieur alphabétiquement à celui la ludothèque 2 ou bien la ludothèque 2 est vide : on ajoute le jeu de la ludothèque 1 à la ludothèque de fusion puis on pointe vers le jeu suivant de la ludothèque 1.
- Le jeu de la ludothèque 2 est inférieur alphabétiquement à celui la ludothèque 1 ou bien la ludothèque 1 est vide : on ajoute le jeu de la ludothèque 2 à la ludothèque de fusion puis on pointe vers le jeu suivant de la ludothèque 2.
- Les deux jeux sont les mêmes : on ajoute un des deux jeux puis on pointe vers le jeu suivant des deux ludothèques.

Si on note n la taille de la ludothèque 1 et m la taille de la ludothèque 2, nous réalisons la boucle max(n,m) fois. À chaque boucle, on ajoute un jeu dans la ludothèque résultat et comme **ajouter_jeu()** est de complexité O(n') avec n' la taille de la ludothèque de fusion. Nous avons donc :

$$\textit{Complexité :} \quad O\left(max\left(n,m\right)\cdot\sum_{i=1}^{n} i\right)$$

Conclusion : suggestions, améliorations

Nous avons ajouté une option « supprimer jeu » dans notre programme principal qui n'apparaît pas dans le cahier des charges afin d'utiliser la fonction **retirer_jeu()** et d'avoir une meilleure gestion des ludothèques.

Une possible amélioration de ce programme serait d'implémenter en plus du programme actuel, une gestion de fichier permettant d'enregistrer les différentes ludothèques créées et de les consulter de nouveau plus tard. Ici comme nous ne nous intéressions qu'au principe de gestion des listes chaînées, à chaque fois qu'on lance le programme, nous n'avons aucune ludothèque.

Afin d'éviter des problème de buffer, nous avons utilisée **fflush(stdin)** pour ne pas avoir de problème avec une entrée d'utilisateur prise en compte dans 2 variables demandées à la suite. Mais cette option utilisée à grand échelle ralentirait énormément le programme, il faudrait donc créer une alternative. Enfin nous pourrions essayer de réduire le temps d'exécution de chaque fonction en optimisant le nombre d'instructions et de boucles à effectuer.

En conclusion, ce projet nous a permis de manipuler des listes chaînées ainsi qu'un tableau de liste chaînée. Nous avons pu voir leurs utilisations au sein de plusieurs fonctions. Nous avons aussi pu concevoir un menu que nous voulions le plus lisible et compréhensible possible car c'est le lien direct entre l'utilisateur et le programme. L'utilisation de bibliothèques supplémentaires nous aurait permis de quitter la console et de créer un environnement fenêtré bien plus agréable pour l'utilisateur.