Rapport du Mini-Projet

Gestion d'une Bibliothèque en C

***

Projet réalisé par :

- SAYOUD Maissa
  Matricule : **191931040670**
  Groupe : **03**

Enseignante

- Mme A. AYAD

***

Février 2022

Introduction

Dans le cadre de notre deuxième année en informatique générale, nous avons eu pour tâche la réalisation d'un mini-projet qui s'intéresse à la gestion d'une bibliothèque. Nos objectifs étaient de savoir analyser un problème, concevoir un algorithme, maitriser le concept des structures dynamiques ainsi que la notion de modularité et enfin implémenter la solution en C.

Dans ce rapport nous allons tout d'abord exposer l'analyse des structures de données, nous détaillons par la suite les fonctions utilisées dans le programme et enfin nous terminons par une conclusion résumant les étapes principales de ce travail.

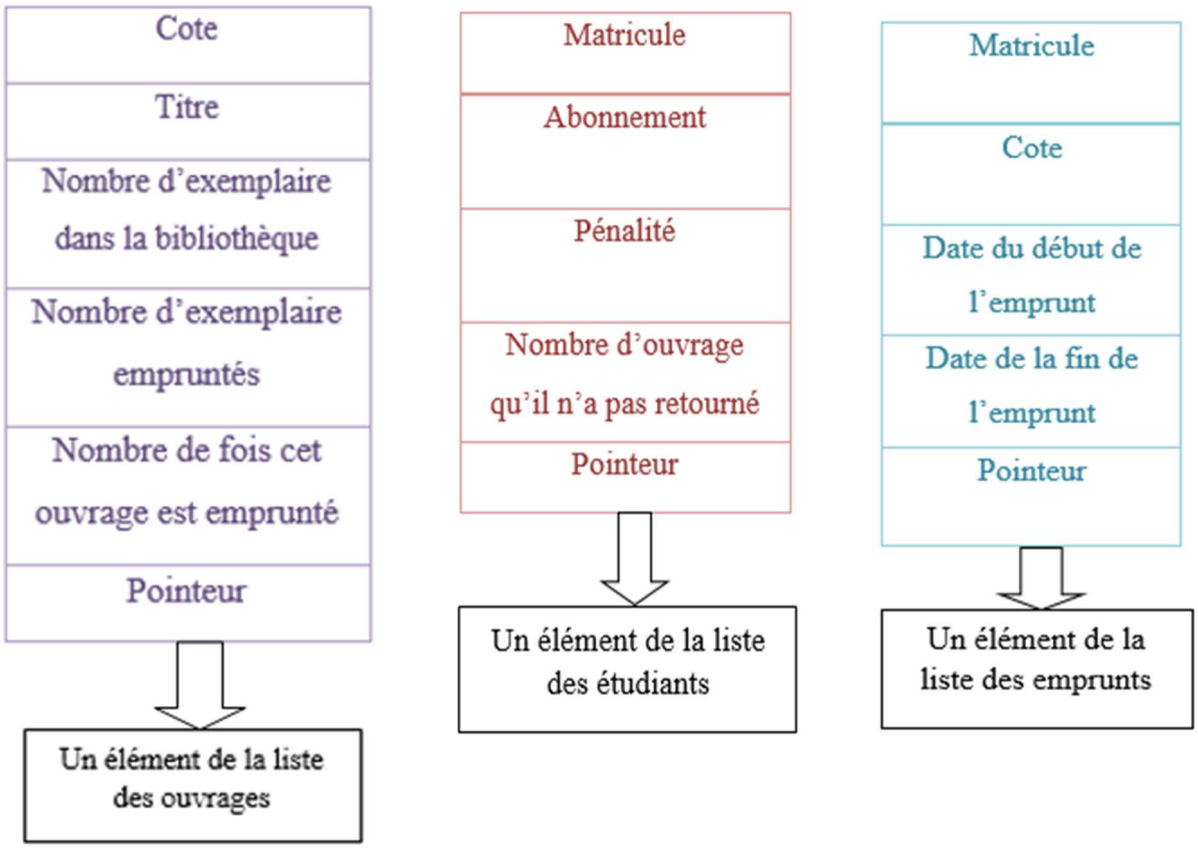
Partie I : Analyse du problème

Nous voulons modéliser et implémenter une application console pour la gestion d'une bibliothèque. L'application doit gérer trois listes : la liste des ouvrages, la liste des étudiants et la liste des emprunts.

Pour cela, nous avons choisi la structure : **liste simplement chaînée** pour les raisons suivantes :

- c'est une structure à allocation **dynamique** dans la mémoire;
- la taille des données est **non limitée**;
- **simplicité** de la mise à jour.

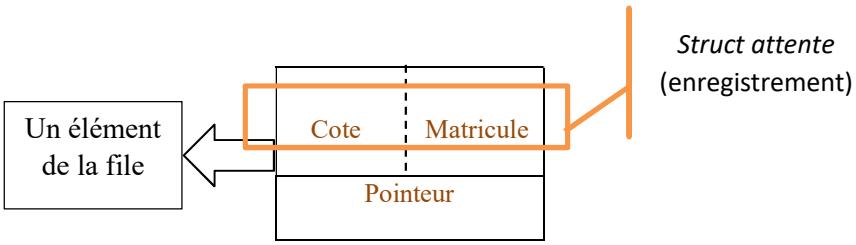
Les trois listes citées précédemment ont des caractérisations différentes qui sont schématisées comme suit :



Remarque

Un champ a été rajouté dans la structure d'un ouvrage et un autre dans la structure d'un étudiant afin de faciliter certaines actions.

En plus des listes, nous avons utilisé aussi les files pour gérer les files d'attente. En ce qui concerne la structure de ces dernières, nous avons préféré qu'elles soient de type **enregistrement** à 2 champs (cote et matricule) pour pouvoir stocker les demandes mises en attente.



Partie II : Les fonctions utilisées

Dans cette partie nous expliquons le déroulement des fonctions et procédures utilisées dans ce programme.

1. L'action : Ajouter des ouvrages

- **But :** Insérer de nouveaux ouvrages dans la liste des ouvrages.
- **Idée :** Pour réaliser cette action, nous l'avons divisé en deux fonctions (procédures) :
  a. Fonction **void CreerListeOuvrage (OListe *TO, int N_Ouvrage)** : Elle se charge de la création de la tête et l'ajout des ouvrages à la fin avec un mode de création FIFO.
  b. Fonction **void AjouterOuvrage (OListe *TO, int i)** : Elle se charge de l'ajout des ouvrages à la fin de la liste déjà créé.

  L'utilisateur introduit au départ le nombre d'ouvrages qu'il veut ajouter dans cette liste. Si la liste est vide, nous appelons la procédure CreerListeOuvrage avec le pointeur **TO** en entrée/sortie (car il va pointer un nouvel élément). Si la liste contient déjà des éléments, nous appelons la procédure AjouterOuvrage.

- **Tests :** *N_Ouvrage* (*nombre d'ouvrages à introduire*) soit strictement positif.

2. L'action : Consulter la liste de tous les ouvrages

- **But :** Afficher tous les ouvrages existants dans la liste des ouvrages.
- **Idée :** La réalisation de cette procédure : **void AfficherListeOuvrages (OListe TO)** est simple, il suffit de parcourir tous les éléments de cette liste en commençant par la tête (pointeur TO en entrée seulement) et afficher leur contenu (Cote, Titre, n_exmplr_biblio, n_exmplr_emprnt) un par un. Quand nous arrivons au suivant du dernier élément qui est NULL nous arrêtons.
- **Tests :** La liste ne doit pas être vide, i.e. la tête (pointeur *TO*) est différente de NULL.

3. L'action : Supprimer un ouvrage

- **But :** Supprimer un ouvrage précisé par l'utilisateur.
- **Idée :** Pour appeler cette procédure : **void SupprimerOuvrage (OListe *TO, char cote_Ouvrage[15], int *creer_O)** l'utilisateur doit d'abord introduire une cote d'ouvrage. La procédure va ensuite vérifier la validité de cette dernière et chercher l'ouvrage correspondant en parcourant toute la liste du début vers la fin et enfin le détacher de la liste. La position de l'ouvrage à supprimer était prise en considération (suppression au début / au milieu/ à la fin).
- **Tests :** *Cote_Ouvrage* valide. i.e. la cote introduite existe dans la liste et la liste ne soit pas vide, i.e. la tête *TO* est différente de NULL.

4. L'action: Vérifier la disponibilité des exemplaires

- **But :** Vérifier si un exemplaire est disponible.
- **Idée :** La fonction **int ExemplaireDispo (OListe TO, char cote_Ouvrage[15])** est une fonction booléenne qui retourne un 0 dans le cas où l'ouvrage précisé par l'utilisateur est disponible et un 1 dans le cas contraire. Pour cela, la fonction vérifie premièrement l'existence de la cote (introduite par l'utilisateur) dans la liste des ouvrages, elle la repère par un parcours du début jusqu'à l'ouvrage demandé

et enfin elle vérifie si son nombre d'exemplaires disponibles dans la bibliothèque est strictement positif ou nul.

- **Conditions :** Liste des ouvrages est non Vide (tête TO différente de NULL) et Cote_Ouvrage existe dans la liste des ouvrages.

5. L'action : Ajouter des étudiants

- **But :** Insérer de nouveaux étudiants dans la liste des étudiants.
- **Idée :** L'idée est la même que celle de l'action Ajouter des ouvrages (action 1). A part le champs *pénalité* de cette structure qui est **automatiquement** initialisé à 0.
  Ainsi, nous définissons les procédures liées à cette action :
  a. **void CreerListeEtud (EListe *TE, int N_Etud).**
  b. **void AjouterEtud (EListe *TE, int i).**
- **Tests :** *N_Etud* (*nombre d'étudiants à introduire*) doit être un nombre positif strictement supérieur à 0.

6. L'action: Consulter un membre

- **But :** Afficher le contenu des champs d'un étudiant.
- **Idée :** Pour appeler cette procédure **void ConsulterMembre(EListe TE, char Matricule [13])** l'utilisateur doit introduire un matricule. La procédure va ensuite vérifier la validité de ce dernier et chercher l'élément de la liste des étudiants correspondant à ce matricule en parcourant toute la liste du début vers la fin et enfin afficher ses champs.
- **Conditions**
  - la liste des étudiants ne soit pas vide ;
  - le matricule existe dans la liste des étudiants. (Une fonction booléenne **int ExisteMatric(EListe TE, char Matricule [13])** a été ajoutée dans le but de faciliter la vérification de l'existence d'un matricule donné).

7. L'action : Supprimer un membre

- **But :** Supprimer un étudiant de la liste des étudiants.
- **Idée :** La même idée que la suppression d'un ouvrage, mais avec la procédure : **void SupprimerMembre (EListe *TE, char Matricule [13], int *Creer_E)**.
- **Conditions :** *Matricule* valide. i.e. le matricule introduit doit exister dans la liste des étudiants et la liste ne doit pas être vide, i.e. la tête *TE* soit différente de NULL.

8. L'action : Modifier un membre

- **But :** Modifier un champ précis d'un étudiant précis.
- **Idée :** à travers le matricule saisi par l'utilisateur, nous appelons la procédure : **void ModifierMembre (EListe *TE, char Matricule [13])** qui se charge de la recherche de l'élément du matricule introduit par l'utilisateur. Une fois l'élément trouvé nous demandons à l'utilisateur de choisir le champ qu'il souhaitera modifier. Les champs sont : le matricule, le type d'abonnement et le total des pénalités.
- **Conditions :** *Matricule* valide. i.e. le matricule introduit doit exister dans la liste, la liste ne doit pas être vide, i.e. la tête *TE* soit différente de NULL, le numéro du champ sélectionné soit valide, le type d'abonnement doit être soit premium soit classique, le type d'étudiant à modifier son total des pénalités soit de type classique.

9. L'action: Ajouter des Emprunts

- **But :** Effectuer un emprunt d'un ouvrage à un étudiant et l'ajouter dans la liste des emprunts.
- **Idée :** Le principe de cette action est pratiquement le même que les actions 1 et 5.
  Nous avons les deux procédures suivantes :

  a. **void CreerTeteEmprunt (PListe *TP, OListe TO, EListe TE, int Duree_Emprunt, int Max_Ouvrage_Emprunte, int *creer_P)** : avant l'exécution de cette fonction, nous vérifions si le bibliothécaire a défini la durée des emprunts et le maximum d'ouvrages à emprunter, sinon nous lui demandons de les introduire.

  b. **void AjouterEmprunt (PListe *TP, OListe TO, EListe TE, File *FP, File *FC) :** L'idée est de créer un nouvel emprunt à partir d'un matricule et une cote d'ouvrage (introduits par l'utilisateur) **valides**. Si les deux champs, Matricule et Cote existent et l'étudiant n'est pas interdit de l'emprunt nous vérifions si

l'ouvrage est disponible, si oui, l'ajout sera effectué exactement comme les ajouts précédents sauf que le champ date de l'emprunt est ajouté automatiquement selon la date d'aujourd'hui récupérée par le système, le champ date du retour de l'emprunt est aussi incrémenté automatiquement à l'aide de la procédure **void DateFin(int Duree_Emprunt,int *f_day,int *f_mois,int *f_an)** sans oublier de décrémenter le *nombre d'exemplaires disponible dans la bibliothèque* et incrémenter le *nombre d'exemplaires empruntés* et le *nombre d'ouvrages que cet étudiant a emprunté*. Sinon la combinaison (matricule, cote) sera stockée dans la file d'attente selon le type de l'étudiant (file classique ou premium) grâce aux fonctions et procédures prédéfinies des files.

- **Conditions :** En plus des conditions, des fonctions ont été utilisées pour vérifier la validité des deux champs (cote et matricule) :
  - *Duree_Emprunt* et *Max_Ouvrage_Emprunte* sont déjà définis ;
  - **int ExisteMatric(EListe TE, char Matricule [13])** ;
  - *Cote_Ouvrage* existe dans la liste des ouvrages ;
  - **int MatriculeNonInterdit (EListe TE, char Matriculee [13]) :** fonction booléenne qui retourne 0 dans le cas où l'étudiant est interdit de l'emprunt et 1 dans le cas contraire ;
  - **int ExemplaireDispo (OListe TO, char cote_Ouvrage[15]).**

10. L'action: Consulter un emprunt

- **But :** Afficher le contenu des champs d'un emprunt précis.
- **Idée :** Pour réaliser cette action, nous appelons la procédure **void ConsulterEmprunt (PListe TP, char cote_Ouvrage[15], char Matricule [13])** qui est de même principe que l'action 6.
- **Conditions :** La liste des emprunts ne soit pas vide et le *matricule*, la *cote* et l'*emprunt* (combinaison (cote,matricule)) doivent exister.

11. L'action: Supprimer un emprunt

- **But :** Supprimer un emprunt précis de la liste des emprunts.
- **Idée :** La même idée de l'action suppression d'un ouvrage mais avec la procédure: **void SupprimerEmprunt (PListe *TP, char cote_Ouvrage[15], char Matricule [13]),** les changements sont au niveau des conditions et des tests d'arrêts de la boucle.
- **Conditions :** Le *matricule* existe dans la liste des étudiants, la *cote* existe dans la liste des ouvrages et l'*emprunt* (combinaison (cote, matricule)) existe dans la liste des emprunts.

12. L'action: Modifier un emprunt

- **But :** Modifier le contenu d'un champ précis d'un emprunt précis.
- **Idée :** La même idée de l'action Modifier un membre mais avec la procédure: **void SupprimerEmprunt (PListe *TP, char cote_Ouvrage[15], char Matricule [13]),** les changements sont au niveau des conditions et tests d'arrêt de la boucle de la recherche.
- **Conditions :** Le *matricule* existe dans la liste des étudiants, la *cote* existe dans la liste des ouvrages, l'*emprunt* (combinaison (cote, matricule)) existe dans la liste des emprunts et le numéro du champ sélectionné soit valide aussi.

13. L'action: Retourner un emprunt

- **But :** Retourner un ouvrage déjà emprunté par un étudiant et effectuer un nouvel emprunt si un autre étudiant attend sa disponibilité dans l'une des files d'attente.
- **Idée :**

  Pour réaliser cette tâche, il faut d'abord vérifier si le matricule de l'étudiant, la cote de l'ouvrage et la combinaison de ces deux derniers (l'emprunt) sont **valides**. Si oui, nous incrémentons le nombre d'exemplaires disponibles dans la bibliothèque, nous décrémentons le nombre d'exemplaires empruntés en parcourant la liste des ouvrages et enfin nous décrémentons le nombre d'ouvrages qu'il a cet étudiant en faisant un autre parcours dans la liste des étudiants et ainsi l'opération de retourner un ouvrage est terminée.

  Avant de passer à une autre action, nous devons vérifier si l'ouvrage est demandé dans l'une des files, pour cela, nous parcourons la file d'attente premium d'abord (si elle n'est pas vide) et nous cherchons élément par élément, du début vers la fin, un qui a la même cote que celle de l'ouvrage retourné. Le premier élément trouvé sera emprunté à l'étudiant qui l'a demandé (l'ajout d'un nouvel emprunt : l'état de la liste a été pris en considération (ajout dans une liste vide ou ajout à la fin)). Si la file premium est

vide ou aucun étudiant n'attend la disponibilité de cet ouvrage, nous faisons de même avec la file d'attente classique.

14. L'action: Ajouter une pénalité

- **But :** Incrémenter le total des pénalités d'un étudiant (de type d'abonnement classique).
- **Idée :** Comme son nom l'indique, la procédure **void AjouterPenalite (EListe *TE)** incrémente le nombre des pénalités. Cette opération se fait en parcourant la liste des étudiants, chercher l'élément qui correspond au matricule **valide** (introduit par l'utilisateur), vérifier le type d'abonnement : s'il est premium nous ne faisons rien, sinon (classique) nous incrémentons la pénalité.

15. L'action: Supprimer une pénalité

- **But :** Initialiser le total des pénalités à 0.
- **Idée :** De la même façon que l'action précédente (action 14) la procédure **void SupprimerPenalite (EListe *TE)** au lieu d'incrémenter la pénalité, elle lui affecte directement un 0.

16. L'action : Afficher la liste des pénalités

- **But :** Consulter tous les étudiants ayant des pénalités.
- **Idée :** Parcourir tous les éléments de la liste des étudiants un par un, si leur pénalité est différente de 0 nous affichons le matricule sinon nous continuons jusqu'à la fin de la liste.
  la procédure : **void AfficherListePenalit (EListe TE).**

17. L'action: Définir la durée des emprunts

- **But :** définir la durée des emprunts par jours.
- **Idée :** affecter une valeur à la durée des emprunts ou changer l'ancienne valeur.
  la procédure : **void DureeEmprunt (int *Duree_Emprunt).**

18. L'action: Définir le maximum d'ouvrages à emprunter

- **But :** définir le maximum d'ouvrages à emprunter.
- **Idée :** affecter une valeur à la durée des emprunts ou changer l'ancienne valeur.
  la procédure : **void MaxEmprunt (int *Max_Ouvrage_Emprunte).**

19. L'action: Consulter les ouvrages les plus sollicités

- **But :** Afficher les ouvrages les plus empruntés.
- **Idée :** à l'aide du champ *n_emprunt* qu'on a rajouté à la structure d'un élément de la liste des ouvrages, il suffit de chercher les éléments qui ont le maximum des emprunts et afficher leurs titres.
  la procédure : **void OuvrgePlusEmprunte (OListe TO).**

20. L'action: Consulter l'ouvrage le moins emprunté

- **But :** afficher les ouvrages les moins empruntés.
- **Idée :** contrairement à la fonction précédente, nous cherchons l'élément qui a le minimum des emprunts et nous affichons son titre (il peut y avoir plusieurs éléments).
  la procédure : **void OuvrgeMoinsEmprunte (OListe TO).**

Conclusion

Le travail effectué dans ce mini projet s'intéresse à l'implémentation d'une application console de la gestion d'une bibliothèque. Pour ce faire, nous avons implémenté vingt actions différentes avec succès, qui peuvent être regroupée en quatre groupes principaux notamment : des actions dédiées à la gestion des ouvrages, des actions dédiées à la gestion des étudiants, des actions dédiées à la gestion des emprunts et enfin des actions dédiées à la gestion des pénalités. En plus de quatre autres actions, deux parmi eux sont des bonus et les restes pour définir certaines caractéristiques de l'emprunt.

De plus, ce programme arrive à signaler les erreurs relatives à une mauvaise manipulation de l'utilisateur comme erreurs d'introduction de cote ou matricule, etc. par des beeps sonores. Il est capable aussi de gérer automatiquement certains champs comme la date du début et de la fin de l'emprunt, etc. sans aucune intervention de l'utilisateur. L'application ainsi implémenté a été testé avec succès.