Compte Rendu du Projet d'algorithmie et de structure de données.

Répartition des tâches:

- Point Allan G5
  - Fonctions de traitement des dates
  - Fonctions de traitement des adhérents
- Ribémont Mathis G5
  - Fonctions de traitement des emprunts et des réservations
  - Fonctions de traitement des listes d'emprunts et de réservations
- Rouault Martin G5
  - Fonctions de traitement des jeux.
  - Fonctions de traitement des tableaux de jeux

Sujet: programme aidant la gestion des adhérents, des emprunts et des réservations d'une ludothèque.

Généralités

Le type Booléen

Un type booléen a été créé pour rendre le code plus explicite.

```
typedef enum {FALSE, TRUE}Bool;
```

Les codes erreurs

Pour simplifier et uniformiser la gestion des codes erreurs, une énumération de codes erreurs a été créée.
A posteriori nous pensons qu'il aurait été intéressant de créer une fonction affichant les messages appropriés aux différents codes.

```
typedef enum {ERR_OUVERTURE_FICHIER=-10, ERR_ALLOCATION /*-9*/,...} CodeErreur;
```

Choix

Cette fonction permet de poser une question "Oui/Non" de manière simple, la fonction gérant les tests sur l'entrée.

```
Bool IO_Choix_O_N(char message[]);
```

Cette fonction test si les caractères `O` (pour oui) ou `N` (pour non) sont entrés par l'utilisateur. Le message en paramètre est la question posée sans `?`.

Date du jour

La date du jour est demandée à l'utilisateur au début du programme pour le bon fonctionnement de ce dernier. En effet, grâce à cela, il est possible de transmettre la date du jour à n'importe quelle partie du code en faisant passer cette date par argument lors d'appel de fonctions qui on besoin de cette date pour fonctionner, sans avoir à la redemander à l'utilisateur.

Fonctionnalités pour les utilisateurs finaux

Généralités

Les différentes fonctionnalités sont accessibles à partir d'un menu. Les fonctions suivantes se chargent respectivement de l'affichage et du traitement des entrées de celui ci.

```
void afficheMenu(void);
void Ludotheque(void);
```

La liste des fonctionnalités est représentée par l'énumération suivante.

```
typedef enum {CHOIX_ANNULER_RESERVATION =1 , CHOIX_EMPRUNTER, CHOIX_RETOUR_JEU, CHOIX_AFFICHE_JEU,
    CHOIX_TRIER_JEU, CHOIX_AJOUTER_JEU, CHOIX_MODIFIER_SUPPRIMER_JEU, CHOIX_NOUV_Adherent,
    CHOIX_RENOUV_Adherent, CHOIX_AFFICHE_Adherent, CHOIX_AFFICHE_EMPRUNT, CHOIX_AFFICHE_RESERVATION,
    CHOIX_AFFICHE_RESERVATION_JEU, CHOIX_AFFICHE_EMPRUNT_JEU, CHOIX_SAUVEGARDER, CHOIX_QUITTER} ChoixAction;
```

La fonction `void Ludotheque(void)` se charge aussi d'initialiser, de charger et de libérer toutes les listes et tableaux stockant adhérents, emprunts, réservations et jeux.

Sauvegarder et quitter

`CHOIX_SAUVEGARDER` et `CHOIX_QUITTER` dans l'énumération ChoixAction.

```
void GLOBAL_Sauvegarder(TableauJeu* tabJeu, Adherent tAdherent[], unsigned int nbElemAdhearant,
    ListeReservation liste_Reservation, int nb_Reservation,
    ListeEmprunt liste_Emprunt, int nb_Emprunt);
```

La fonction `GLOBAL_Sauvegarder` est responsable de la sauvegarde des différentes données.

```
case CHOIX_QUITTER:
    if (IO_Choix_O_N("Souhaitez vous sauvegarder avant de quitter"))
        GLOBAL_Sauvegarder(&tabJeu, tAdherent, nbElemAdhearant, liste_Reservation, nb_Reservation,
            liste_Emprunt, nb_Emprunt);
    lance = FALSE;
    break;
```

Il n'y a pas de fonction pour quitter. Il est proposé de sauvegarder avant de quitter.

Affichage

`CHOIX_AFFICHE_JEU` `CHOIX_AFFICHE_Adherent` `CHOIX_AFFICHE_EMPRUNT` `CHOIX_AFFICHE_RESERVATION`
Ces quatre affichages utilisent les fonctions d'affichages fournies avec les différentes structures.

`CHOIX_AFFICHE_RESERVATION_JEU` `CHOIX_AFFICHE_EMPRUNT_JEU`
Ces deux fonctionnalités d'affichage utilisent une fonction globale.

```
void GLOBAL_afficherListeERJeu_Interactif(ListeER liste, TableauJeu* tabJeu, Bool isReservation)
```

Cette fonction permet la recherche d'un jeu pour lequel on souhaite afficher les réservations ou les emprunts associés.
On fait passer un booléen à la fonction pour savoir s'il s'agit d'afficher des réservations ou des emprunts, de manière à modifier l'affichage en conséquence.

Gestion des jeux

`CHOIX_TRIER_JEU` `CHOIX_AJOUTER_JEU`
Ces deux fonctionnalités s'appuient uniquement sur des fonctions de `TableauJeu`.

`CHOIX_MODIFIER_SUPPRIMER_JEU`
Celle ci requiert différents tests sur les emprunts et les réservations pour éviter de corrompre leurs listes respectives.

```
Bool GLOBAL_ModifierSupprimerJeu(TableauJeu* tabJeu, ListeReservation* liste_Reservation,
    unsigned int *nb_Reservation, ListeEmprunt liste_Emprunt);
```

`GLOBAL_ModifierSupprimerJeu` se charge donc d'assurer la cohérence d'une suppression ou d'une modification de jeu.

Gestion des adhérents

Ajouter un `Adherent`

C'est la fonction `GLOBAL_NouvelAdherent` qui s'occupe d'ajouter un `Adherent` dans le tableau d'adhérents s'il n'est pas déjà enregistré. Elle retourne un booléen pour indiquer si l'`Adherent` a bien était ajouté.

```
Bool GLOBAL_NouvelAdherent(Adherent* tAdherent[], int* nbElemAdhearant,
     unsigned int* tMaxAdherent, unsigned int* rangNouvAdherent, Date dateDuJour)
```

Renouveler un abonnement

C'est la fonction `GLOBAL_RenouvellerAdherent` qui s'occupe de renouveler un abonnement. Elle demande un montant et renouvelle l'abonnement en fonction de ce dernier. Elle renvoi un booléen pour indiquer si tout c'est bien passé.

```
Bool GLOBAL_RenouvellerAdherent(Adherent tAdherent[], unsigned int nbElemAdhearant)
```

Emprunt et réservation de jeux

Voici les différentes fonctionnalités que l'utilisateur peut utiliser.

Emprunter et réserver

Pour emprunter un jeu, on utilise la fonction

```
Bool GLOBAL_Emprunter(ListeReservation* liste_Reservation,
  unsigned int* nb_Reservation, ListeEmprunt* liste_Emprunt,
  unsigned int* nb_Emprunt, TableauJeu* tabJeu, Adherent* tAdherent[],
  int* nbElemAdhearant, unsigned int* tMaxAdherent, Date dateDuJour)
```

S'il n'y a pas d'exemplaire du jeu disponible, on demande à l'utilisateur s'il doit créer une réservation. Si oui, une réservation est créée.

Annuler une réservation

```
Bool GLOBAL_Anuller_Reservation(ListeReservation* lr,
  unsigned int* nb_Reservation, Adherent tAdherent[],
  unsigned int nbElemAdhearant, TableauJeu* tabJeu)
```

Cette fonction permet d'annuler une réservation en demandant les informations à l'utilisateur.

Retourner un jeux

```
Bool GLOBAL_RetourJeu(Adherent tAdherent[], unsigned int nbElemAdhearant,
  TableauJeu* tabJeu, ListeEmprunt* liste_Emprunt, unsigned int* nb_Emprunt,
  ListeReservation* liste_Reservation, Date dateDuJour)
```

Cette fonction permet de retourner un jeu. Si des réservations du jeu rendu existent, la plus ancienne réservation est donc transformée en emprunt.

Fonctions concernant les dates et les adhérents

Les dates

Tout d'abord

L'implémentation des dates de trouve dans les fichiers `source/Date.h` et `source/Date.c`.

Structure des fichiers

Le fichier d'entête (*source/Date.h*) débute par une *header guard* affin de protéger le fichier des inclusions multiples. S'en suit les inclusions nécessaires au bon fonctionnement du code (*stdio.h*, *stdlib.h* et *CodeErreur.h*) puis l'implémentation de la structure `Date`. La suite du fichier consiste à commenter les prototypes des fonctions qui sont définis en dessous de ces commentaires.

Le fichier source (*source/Date.c*) inclue simplement son fichier d'entête et contient ensuite l'implémentation commentée des fonctions prototypées dans le fichier d'entête.

La structure `Date`

```
typedef struct {
  int jour;
  int mois;
  int annee;
} Date;
```

Cette structure contient simplement 3 entiers représentant les éléments que l'on trouve dans une date, à savoir `jour`, `mois` et `année`.

Fonctionnalités avec `Date`

Les `Date` n'ont pas de fonctionnalités propres proposées dans le menu mais elles servent au bon déroulement du programme.

Les adhérents

Tout d'abord

L'implémentation des dates de trouve dans les fichiers `source/Adherent.h` et `source/Adherent.c`.

Structure des fichiers

Le fichier d'entête (*source/Adherent.h*) débute par un *header guard* affin de protéger le fichier des inclusions multiples. S'en suit les inclusions nécessaires au bon fonctionnement du code (*stdio.h*, *stdlib.h*, *Bool.h*, *Date.h* et *CodeErreur.h*) puis l'implémentation de l'énumération `Civilite` et de la structure `Adherent`. La suite du fichier consiste à commenter les prototypes des fonctions qui sont définis en dessous de ces derniers.

Le fichier source (*source/Adherent.c*) inclue simplement son fichier d'entête et contient ensuite l'implémentation commentée des fonctions prototypées dans le fichier d'entête.

La Structure `Adherent`

```
typedef struct
{
    unsigned int id;
    Civilite civilite;
    char nom[22];
    char prenom[22];
    Date dateInscri;
} Adherent;
```

La structure `Adherent` est composée des champs requis permettant la sauvegarde de cette structure.

Les `Adherent` serons par la suite stockés dans un tableau dynamiquement alloué. Ce choix a été fait car les `Adherent` seront plus lus que modifiés et car de nouveaux `Adherent` peuvent venir s'ajouter au fur et à mesure du temps.

Fonctionnalités avec `Adherent`

```
void afficheTabAdherent(Adherent tAdherent[], unsigned int nbElem, FILE* flux, Bool entete);
Adherent nouvAdherent(unsigned int id, Date dateDuJour);
CodeErreur renouvelerInscription(Adherent* ad, Date* nouvelleDate);
CodeErreur sauvegarderAdherent(Adherent tAdherent[], unsigned int nbElem, char nomDuFichier[]);
```

Toutes ces fonctions sont proposées dans le menu et sont toutes opérationnelles. Elles permettent ainsi le bon déroulement du programme. Aucune de ces fonctions n'ont posées problème lors de leur implémentation.

Voici le reste des fonctions qui servent au fonctions implémentées si dessus :

```
Adherent lireAdherent(FILE* flux);
void afficheAdherent(Adherent ad, FILE* flux, Bool entete);
int insererAdherent(Adherent* tAdherent[], unsigned int nbElem, unsigned int *tMax, Adherent* ad);
int supprimerAdherent(Adherent tAdherent[], unsigned int nbElem, unsigned int id);
void decalageAGaucheAdherent(Adherent tAdherent[], unsigned int debut, unsigned int nbElem);
void decalageADroiteAdherent(Adherent tAdherent[], unsigned int debut, unsigned int nbElem);
unsigned int rechercherUnAdherent(Adherent tAdherent[], unsigned int nbElem, unsigned int id, Bool* trouve);
int chargerLesAdherents(Adherent* tAdherent[], unsigned int* tMax, char nomDuFichier[]);
Bool checkInscriptionValide(Adherent* ad, Date* dateDuJour);
CodeErreur copieTabAdherent(Adherent tAdherent1[], unsigned int nbElem1,
    Adherent tAdherent2[], unsigned int tMax2);
unsigned int rechercherIDAdherentLibre(Adherent tAdherent[], unsigned int nbElem);
```

Ici, plusieurs fonctions ont été problématiques lors de leur implémentation. En effet, les fonctions `copieTabAdherent` et `insererAdherent` m'ont posé problème. Je (*Allan POINT*) n'arrivais pas à réallouer le tableaux dans la fonction

`copieTabAdherent` et donc j'ai dû le faire dans `insererAdherent`. C'est pour cette raison que `copieTabAdherent` renvoi **ERR_OUT_OF_RANG** en cas d'erreur.

Fonctions concernant les emprunts et les réservations

Structure emprunt et réservation

Les emprunts et les réservations sont issus de la structure dans `source/EmpruntReservation.h` et les fonctions de traitement de cette structure sont décrites dans `source/EmpruntReservation.c`.

```
typedef struct
{
    unsigned int id;
    unsigned int idAdherent;//reference un adherent
    unsigned int idJeu;//reference à un jeu
    Date date;
} Emprunt, Reservation, EmpruntReservation;
```

Les emprunts et les réservations contiennent les mêmes variables, on peut donc utiliser la même structure pour les deux. De plus, le type de la structure porte plusieurs nom selon le contexte (si on définie un emprunt ou une réservation).

Pour traiter cette structure, plusieurs fonctions sont disponibles dont voici les prototypes.

```
EmpruntReservation lireEmpruntReservation(FILE* flux);
void afficherEmpruntReservation(EmpruntReservation *Empruntreservation, FILE* flux);
```

Passer le flux en paramètre permet de pouvoir réunir la fonction d'affichage à l'écran et la fonction d'écriture en une seul. Le paramètre flux est donc prévue pour prendre deux arguments possibles:

- flux → qui sera un flux vers le fichier dans lequel on souhaite écrire
- stdout → flux vers la sortie standard (écran)

Listes contenant les emprunts et les réservations

Les emprunts et les réservations sont contenus dans des listes, qui permettent une manipulation plus fluide d'un point de vue mémoires que les tableaux, car une liste n'a pas de taille pré-définie. La structure des liste se trouve dans `source/ListeEmpruntReservation.h` et les fonctions des traitements listes sont décrites dans `source/ListeEmpruntReservation.c`.

```
typedef struct element
{
    EmpruntReservation empRes;
    struct element* suiv;
} Element, *ListeReservation, *ListeEmprunt, *ListeER;
```

Là aussi, le type porte plusieurs nom selon le contexte à la définition.

Chaque liste est initialisé grâce à la fonction `ListeER listeER_Vide(void);`.
Pour savoir si une liste est vide, il y a la fonction `Bool listeER_estVide(ListeER liste);` qui renvoi `TRUE` si la liste est vide.
Enfin, les listes sont triées par identifiant des emprunts et des réservations.

Fonction d'affichage

Il existe deux fonctions pour afficher une liste.

```
void afficherListeEmpruntReservation(ListeER liste, FILE* flux,int nb);
void afficherListeERJeu(ListeER liste, unsigned int idJeu);
```

La première fonction affiche une liste entière. Elle est aussi utilisée pour écrire dans les fichiers de données lors de la sauvegarde (en précisant un flux vers un fichier), justifiant le paramètre `nb` (nombre d'élément dans la liste).

La seconde fonction affiche uniquement sur la sortie standard les éléments d'une liste concernant un jeu donné.

Fonction de chargement

Le chargement des fichiers d'emprunts et de réservations est assuré par la fonction suivante.

```
chargerListeEmpruntReservation(char nomDeFichier[], unsigned int *nb);
```

Cette fonction prend en paramètre une chaîne de caractères `nomDeFichier` qui sera utilisée pour créer le flux vers le fichier dans lequel sont sauvegardées les données à charger. La fonction prend aussi en paramètre un pointeur vers la variable dans laquelle sera enregistré le nombre d'éléments dans la liste.
Il faut appeler cette fonction pour chaque liste a créer.

Cette fonction fait appel à la fonction `lireEmpruntReservation(FILE* flux)` pour lire dans le fichier.

Fonction d'insertion

Pour insérer un élément dans une liste, il faut appeler la fonction suivante.

```
ListeER insererEmpruntReservation(ListeER liste, unsigned int *nb, EmpruntReservation er);
```

Elle prend en paramètre la liste dans laquelle on insère l'élément, un pointeur sur le nombre d'élément dans la liste, et l'emprunt ou la réservation à insérer. Cette fonction utilise la fonction

```
ListeER insererDevantEmpruntReservation(ListeER liste, EmpruntReservation er);
```

Suppression d'un élément

La suppression d'un élément se passe de la même manière avec deux fonctions.

```
ListeER supprimerEmpruntReservation(ListeER liste, unsigned int id, unsigned int *nb, CodeErreur* cErr);
ListeER supprimerDevantEmpruntReservation(ListeER liste);
```

La différence ici c'est qu'on fait passer l'identifiant de l'élément à supprimer au lieu d'un emprunt ou d'une réservation. On lui fait aussi passer un pointeur vers une variable `cErr` pour retourner un code erreur si besoin.

Les fonctions de recherche

Plusieurs fonction de recherche existent selon ce qu'on recherche et les informations qu'on possède.

```
unsigned int rechercherIdLibre(ListeER liste);
unsigned int rechercherListeER_AdJeu(ListeER liste, unsigned int idAdherent, unsigned int idJeu, Bool* trouve
```

```
unsigned int rechercherListeER_Jeu(ListeER liste, unsigned int idJeu, Bool* trouve);
```

La fonction `rechercherIdLibre()` permet de rechercher le premier identifiant libre (inutilisé) de la liste qu'on lui passe, ce qui permet de garder les liste triée par identifiant. Cette identifiant servira à définir un nouvel élément à insérer dans la liste.

La fonction `rechercherListeER_AdJeu()` permet de rechercher un élément, selon l'identifiant du jeu emprunté et l'identifiant de l'adhérent qui a emprunté le jeu.
Cette fonction retourne l'identifiant de l'élément. Elle permet aussi de savoir avec le pointeur `trouve` si l'élément a été trouvé ou non.

La fonction `rechercherListeER_Jeu` permet de rechercher un élément selon un identifiant de jeu. Cette fonction est surtout utilisée pour savoir si un emprunt ou une réservations d'un jeu existe.

Réservation la plus ancienne

Quand un jeu est retourné par un adhérent, on recherche les réservations du jeu rendu pour changer la plus ancienne réservation en emprunt. Pour récupérer la réservation la plus ancienne, il faut utiliser la fonction suivante

```
Reservation plusVieilleReservationJeu(ListeReservation liste_Reservation, unsigned int idJeu);
```

Sauvegarde

```
CodeErreur sauvegarderListeER(ListeER liste, char nomDeFichier[], int nb);
```

Cette fonction écrit une liste dans un fichier dont le nom est donné par la chaîne de caractère `nomDeFichier`. Elle prend en paramètre `nb`, le nombre d'éléments dans la liste. Pour écrire dans le fichier, la fonction utilise `afficherListeEmpruntReservation()`. Comme décrite plus tôt, cette fonction prend en paramètre un flux (fichier ou stdout) et le nombre d'élément dans la liste.

Fin du programme

Tous les éléments des listes sont créés par allocation dynamique. Pour supprimer tous ces éléments, il faut utilisé la fonction suivante

```
ListeER supprimerListe(ListeER liste);
```

Cette fonction va libérer la mémoire pour chaque élément de la liste qui lui est passée. Elle retourne `NULL` quand tous les éléments de la liste sont supprimés.

Traitement des jeux

Préambule

Mon objectif principal était de permettre une certaine versatilité en terme d'accès aux jeux. Pour cela je me suis concentré sur les systèmes de tri et de recherche.

Petit point sur les fichiers

Les headers `source/TableauJeu.h` et `source/Jeu.h` disposent de header guard. Ils incluent tout deux `source/CodeErreur.h` et `source/Bool.h` en plus de quelques headers standards.

`source/TableauJeu.c` et `source/Jeu.c` n'incluent que leur `.h` respectifs. Ils contiennent les commentaires des fonctions.

Note au lecteur

Les termes `Jeu` et `structure Jeu` seront tout deux utilisés pour designer la structure `Jeu`. De même pour la structure `TableauJeu`.

Les fichiers Jeu.h et Jeu.c

Jeu : une structure simple

La structure

La structure jeu représente un jeu (et ses exemplaires).

```
typedef struct
{
    unsigned int id;
    char nom[41];
    char type[15];
    unsigned int nbExemplaireTotal;
    unsigned int nbExemplaireDispo;
} Jeu;
```

La structure de jeu est composée des éléments requis auxquels s'ajoute le nombre d'exemplaires disponibles. Cette variable permet de savoir efficacement si un jeu est disponible ou non.
Le type de jeu (ou catégorie) n'est pas contraint pour offrir une place aux genres de niche pouvant exister.

Les fonctions associées

```
Jeu* lireJeu(FILE* flux);
void afficheJeu(Jeu* jeu, FILE* flux);
Jeu* nouvJeu(unsigned int id);

void copyJeu(Jeu* jd, Jeu* js);
Jeu* allocJeu(void);
```

Les trois premières fonctions offrent les fonctionnalités basiques d'entrée et de sortie pour la structure Jeu.
Les deux dernières concernent plus la gestion des ressources.
Même si j'essaie de copier le plus rarement possible les jeux, lorsque c'est nécessaire il faut que ce soit bien fait. Jeu contenant des chaînes de caractères, j'ai crée `copyJeu`.
L'allocation mémoire pouvant être sensible je l'ai encapsulé dans `allocJeu` de manière à ne pas multiplier les possibilités d'erreurs.

ElementJeu : une énumération facilitant les opérations

L'énumération

`ElementJeu` permet d'indiquer un des éléments de la structure Jeu.

```
typedef enum { ELEM_JEU_NONE , ELEM_JEU_ID, ELEM_JEU_NOM, ELEM_JEU_TYPE,
    ELEM_JEU_NB_EXEMPLAIRE_TOTAL, ELEM_JEU_NB_EXEMPLAIRE_DISPO} ElementJeu;
```

Cette énumération offre au fonctions sur les jeux la capacité d'agir sur les différentes variables des jeux avec plus de simplicité (et de manière uniforme). Toute les variables de Jeu y sont référencées, et "aucune" est représenté par ELEM_JEU_NONE.

Les fonctions liées

Ces trois fonctions permettent de factoriser le code concernant `ElementJeu`.

```
ElementJeu choisirElementJeu(char utilite[]);
Bool elementJeuExiste(ElementJeu elementJeu, Bool noneAutorisee);
void afficheAllElementJeu();
```

Des fonctions associant Jeu et ElementJeu

```
CodeErreur entrerValeurElementJeu(Jeu* jeu, ElementJeu elementJeu);
int jeuCmp(Jeu* j1, Jeu* j2, ElementJeu elementJeu);
```

`jeuCmp` défini l'ordre entre les jeux selon les éléments.
`entrerValeurElementJeu` facilite l'entrée d'une valeur pour tous les éléments.

Les fichiers TableauJeu.h et TableauJeu.c

TableauJeu : Un tableau entouré

La structure

`TableauJeu` contient un tableau de jeux et les données nécessaires à son fonctionnement.

```
#define TAILLE_MAX_TAB_JEU 100

typedef struct
{
    unsigned int nbElement;
    ElementJeu triSur;
    Jeu* jeux[TAILLE_MAX_TAB_JEU];
} TableauJeu;
```

Le tableau à proprement parler est un tableau statique de pointeurs vers des Jeu. Sa taille est défini par une macro du préprocesseur (Ce qui permet de l'utiliser dans les fonctions en ayant besoin).
Nous avons choisi un tableau de pointeurs pour éviter un maximum la copie des Jeu durant les nombreux tris effectués sur le tableau de jeux.
Le nombre de jeux dans le tableau est stocké dans la variable `nbElement`.
`TableauJeu` contient aussi une variable `ElementJeu` indiquant si et comment le tableau est ordonné.

Les fonctions sur TableauJeu

Les fonctions ayant besoin d'un `TableauJeu` prennent un pointeur sur un `TableauJeu` de manière à éviter de copier la structure.

Initialisation et libération

```
void initTabJeu(TableauJeu* tabJeu);
void libererTabJeu(TableauJeu* tabJeu);
```

Ces deux fonctions sont à appelées respectivement à la création et à la fin de vie du `TableauJeu`.

Chargement et sauvegarde

```
CodeErreur chargerTabJeu(TableauJeu* tabJeu, char nomFichier[]);
CodeErreur sauvegarderTabJeu(TableauJeu* tabJeu, char nomFichier[]);
```

Ces fonctions manipulent les fichiers pour sauvegarder et charger un `TableauJeu`.

Initialisation

```
void UTILE_InitNbJeuDispo(ListeEmprunt liste_Emprunt, TableauJeu* tabJeu);
```

Cette fonction n'est pas une fonction de `TableauJeu.h` mais une fonction de `Ludotheque.h`.
Elle initialise les nombres d'exemplaires disponibles des différents jeux.

Affichage

```
void afficheTabJeu(TableauJeu* tabJeu, FILE* flux);
void affichePartieTabJeu(TableauJeu* tabJeu, unsigned int begin, unsigned int end, FILE* flux);
```

Ces fonctions écrient tout ou partie d'un tableau de jeu dans un flux.

Recherche

```
CodeErreur rechercherJeuInteractif(TableauJeu* tabJeu, Bool* trouve, unsigned int* rang);
unsigned int rechercherJeu(TableauJeu* tabJeu, Jeu* jeu, ElementJeu elementJeu, Bool* trouve, Bool cherchePre
unsigned int _rechercherPremierJeu_TabNonTrie(TableauJeu* tabJeu, Jeu* jeu, ElementJeu elementJeu, Bool* trou
unsigned int _rechercherDernierJeu_TabNonTrie(TableauJeu* tabJeu, Jeu* jeu, ElementJeu elementJeu, Bool* trou
unsigned int _rechercherPremierJeu_TabTrie(TableauJeu* tabJeu, Jeu* jeu, ElementJeu elementJeu, Bool* trouve)
unsigned int _rechercherDernierJeu_TabTrie(TableauJeu* tabJeu, Jeu* jeu, ElementJeu elementJeu, Bool* trouve)
```

Ces fonctions permettent la recherche de jeu dans un tableau.
Les fonctions débutants par _ sont uniquement présentes pour servir `rechercherJeu` et `rechercherJeuInteractif`.

Tri

```
void triTabJeuInteractif(TableauJeu* tabJeu);
void triTabJeu(TableauJeu* tabJeu, ElementJeu elementJeu);
//les fonctions suivantes sont utilisé en interne par triTabJeu
void _triJeu(Jeu* tSource[], unsigned int nbElem, ElementJeu elementJeu);
void copyTabJeu(Jeu* tSource[], unsigned int debut, unsigned int fin, Jeu* tDest[]);
void fusionTabJeu(Jeu* tSource1[], unsigned int nbElem1, Jeu* tSource2[], unsigned int nbElem2, ElementJeu el
    Jeu* tDest[]);
```

`triTabJeuInteractif` et `triTabJeu` sont des fonctions faisant l'interface entre les fonctions utilisant le tri et `_triJeu`, une fonction de tri par dichotomie.

Ajout et délétion

```
CodeErreur retirerJeu(TableauJeu* tabJeu, Jeu* jeu);
CodeErreur retirerJeuInteractif(TableauJeu* tabJeu);

CodeErreur ajouterJeu(TableauJeu* tabJeu, Jeu* jeu);
CodeErreur ajouterJeuInteractif(TableauJeu* tabJeu);
//les fonctions suivantes sont utilisé en interne par retirerJeu et ajouterJeu
void _decalageAGaucheJeu(TableauJeu* tabJeu, unsigned int debut);
void _decalageADroiteJeu(TableauJeu* tabJeu, unsigned int debut);
```

Ces fonctions permettent d'ajouter ou de retirer des `Jeu` du tableau avec ou sans interaction avec l'utilisateur. `retirerJeuInteractif` n'est pas utilisé dans ce projet car il est nécessaire de faire des tests sur les listes de réservations et d'emprunts.