SCIENCES SORBONNE UNIVERSITÉ

UFR d'Ingénierie

18 février 2020

L1 (2019-2020)

Éléments de programmation en C (LU1IN002)

TME

Semaines 7 à 11

Semaine 7 - TME

Objectifs

— Structures
— Tableaux de structures

Exercices

Si vous devez recopier des fichiers pour réaliser ce TP, ces derniers se trouvent dans le répertoire `/Infos/lmd/2019/licence/ue/LU1IN002-2020fev/Semaine7`.

Exercice 35 – Système solaire

Nous souhaitons représenter les planètes du système solaire. Chaque planète est caractérisée par :
- son nom (10 caractères au maximum),
- sa densité (un réel),
- sa distance au soleil en millions de kilomètres (un réel),
- le nombre de ses satellites (un entier).

Vous complèterez le fichier `planete.c` qui vous est fourni et testerez vos fonctions au fur et à mesure.

Question 1
Définissez le type `planete` qui permet de déclarer une planète. Pour pouvoir utiliser le fichier qui vous est fourni faites attention à l'ordre dans lequel vous déclarez les différents éléments, il doit correspondre à l'ordre de l'énoncé.

Question 2
Écrivez la fonction `affichePlanete` qui prend en paramètre une planète et qui affiche ses caractéristiques.

Question 3
Écrivez la fonction `afficheToutesPlanetes` qui prend en paramètre un tableau de planètes, sa taille et qui affiche les caractéristiques de toutes les planètes du tableau. Cette fonction doit faire appel à la fonction `affichePlanete`.

Question 4
Nous faisons l'hypothèse que des nouvelles découvertes peuvent remettre en cause les densités actuellement connues. Pour anticiper une telle possibilité, écrivez la fonction `modifieDensite` qui permet de modifier la densité d'une planète, son type de retour doit être **`void`**.

Exercice 36 – Poursuite du TP

Une fois fait le premier exercice de ce sujet, vous avez deux possibilités :
- **Vous n'avez pas fini les exercices des semaines précédentes**, vous devez les terminer, ces exercices abordent des notions de base que vous devez maîtriser.
- **Vous avez fini les exercices des semaines précédentes**, vous pouvez donc continuer avec les exercices de cette semaine (programmation d'un Tetris) qui reviennent sur les différentes notions vues dans les semaines précédentes et les structures.

Exercice 37 – Tetris : structures de données

Cette présentation est en grande partie extraite de Wikipedia (`http://fr.wikipedia.org/wiki/Tetris`).

Le principe du jeu de Tetris est simple : des pièces de couleur et de formes différentes descendent du haut de la fenêtre de jeu. Le joueur ne peut pas agir sur cette chute mais peut décider à quel angle de rotation ($0°$, $90°$, $180°$, $270°$) et à quel emplacement latéral l'objet peut atterrir. Lorsqu'une ligne horizontale est complétée sans vide, elle disparaît et les blocs supérieurs tombent. Si le joueur ne parvient pas à faire disparaître les lignes assez vite et que la fenêtre se remplit jusqu'en haut, la partie est finie.

Les pièces de Tetris sont appelées "tétrominos" (du grec *tetra* : quatre) car elles sont toutes basées sur un assemblage de quatre carrés. Il en existe sept formes différentes, elles sont représentées dans la figure 1. A chacune d'elles est associée une lettre de l'alphabet.

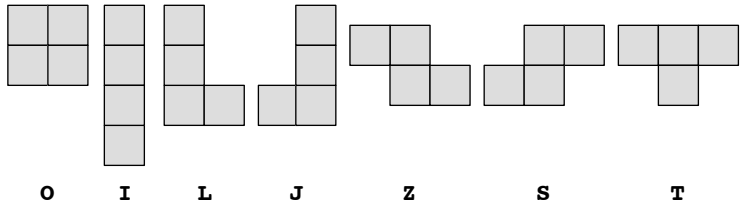

FIGURE 1 – Représentation des sept tétrominos et de la lettre associée

Le plateau de jeu est une grille formée de cases de même taille que les carrés qui composent les pièces. La largeur du plateau est de 10 cases, la hauteur de 22 cases.

Récupérez le fichier `Tetris1.c` qui contient une boucle de jeu qui va vous permettre de tester vos fonctions. Dans cette version du programme, contrairement au vrai jeu de Tetris, aucune action n'est temporisée. Il s'agit en fait de disposer d'un programme dans lequel l'utilisateur maîtrise tous les événements, ce qui rend la mise au point plus facile.

Les pièces de Tetris sont colorées. Le tableau `color` contient 7 couleurs (une couleur est une chaîne de caractères, en anglais et en minuscules), correspondant aux 7 tétrominos.

```
char* color[] = {"light salmon","fuchsia","lime green",
                 "white","yellow","cyan","grey"};
```

Un tétromino est un assemblage de quatre cases. Chacune de ces cases est repérée par deux coordonnées `colonne` et `ligne` qui permettent de la positionner sur le plateau de jeu. La structure `une_case` suivante permet de représenter une case.

```
struct une_case  {
  int colonne;
  int ligne;
};
```

Lors de la création d'une nouvelle pièce sur le plateau de jeu, il faut initialiser (entre autres) les coordonnées de ses quatre cases. Pour faire cette initialisation facilement, nous choisissons tout d'abord de construire un tableau, nommé `tab_pieces`, contenant la description des 7 tétrominos. Nous allons donc déclarer un tableau à deux dimensions : une ligne par tétromino et pour chaque ligne, quatre colonnes correspondant aux 4 cases de chaque tétromino. Pour chaque tétromino, une des cases est en position (0,0), la position des autres cases est donnée relativement à la case en position (0,0). Nous expliquerons un peu plus loin, comment ce tableau est construit.

Pour déterminer la position à un instant donné d'un tétromino, nous choisissons de mémoriser la position sur le plateau de jeu de la case $(0, 0)$, c'est-à-dire, la $colonne$ et la $ligne$ auxquelles elle se trouve. Les emplacements occupés par les autres cases seront calculés en ajoutant les coordonnées relatives de celles-ci à $(colonne, ligne)$.

Enfin, un tétromino a une couleur. Pour limiter les manipulations de chaînes de caractères, nous allons simplement indiquer dans la définition du tétromino la position de sa couleur dans le tableau de couleurs.

La structure `piece` suivante permet de représenter un tétromino : les coordonnées de la case $(0, 0)$ sur le plateau de jeu, les coordonnées relatives de ses 4 cases et sa couleur.

```
struct piece {
  int pos_ligne,pos_colonne;
  struct une_case la_piece[4];
  int type;
};
```

Question 1

Nous revenons sur la construction du tableau `tab_pieces` contenant la description des 7 tétrominos.

Nous choisissons de représenter l'assemblage des quatre cases d'un tétromino par les coordonnées relatives des cases les unes par rapport aux autres : l'une des cases a toujours pour coordonnées $(0, 0)$ et les coordonnées des autres cases sont définies par rapport à celle-ci. Par exemple, les coordonnées des quatre cases du tétromino 'O' (carré) sont données dans la Figure 2.

Supposons que le tableau `tab_pieces` regroupe la description des sept tétrominos et la pièce `O` (carré) se trouve à l'indice `0`. La représentation de cette pièce est donnée dans la Figure 2. L'initialisation d'une pièce se fera ensuite en recopiant dans le champ adéquat de la pièce créée la ligne correspondante du tableau `tab_pieces`. L'ordre des 4 cases dans chaque ligne du tableau `tab_pieces` n'a pas d'importance.

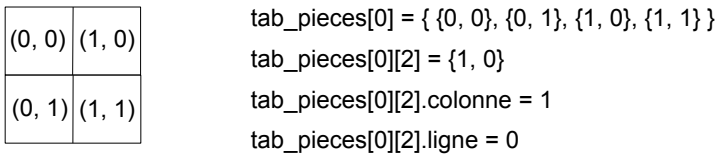

FIGURE 2 – Représentation du tétromino 'O'

Insérez, dans la fonction `main` la déclaration et l'initialisation du tableau `tab_pieces` permettant de stocker les sept tétrominos. Pour simplifier l'insertion d'une nouvelle pièce dans le tableau de jeu, il est judicieux de choisir les coordonnées de sorte qu'aucune coordonnée $ligne$ ne soit négative.

Dans la suite, lorsque nous parlons du *type* d'une pièce il s'agit de son indice dans le tableau que vous devez déclarer. Dans l'exemple de la Figure 2, la pièce de type `0` est la pièce associée à la lettre `O`.

Dans la fonction `main`, l'appel à la fonction `afficher_toutes_pieces`, affiche les septs tétrominos. L'exécution du programme vous permet alors de vérifier que vous avez bien déclaré le tableau `tab_pieces`. Une fois cette vérification faite, vous pouvez mettre l'appel à `afficher_toutes_pieces` en commentaire.

Pour compiler le programme vous devez utiliser l'option `-lcini` pour pouvoir accéder aux fonctions de la bibliothèque graphique. La touche `escape` vous permer de fermer la fenêtre graphique et donc de terminer l'exécution du programme.

Exercice 38 – Tetris : le jeu

Le principe général du jeu consiste à gérer les déplacements d'une pièce sur le plateau de jeu. Celui-ci est représenté par un tableau de `HAUTEUR` lignes et `LARGEUR` colonnes. La valeur stockée dans une case du tableau est la couleur de l'élément qui l'occupe si elle n'est pas vide (sous forme d'un indice référençant la case correspondante dans le tableau de couleurs), une valeur particulière représentée par la constante `VIDE` si la case n'est pas occupée.

Le plateau de jeu

Le plateau de jeu est affiché dans une fenêtre graphique de même taille. Chaque case du plateau (qui a la même taille qu'une case de tétromino) sera représentée par un carré de `TAILLE_CASE` pixels de côté.

Question 1

Complétez la fonction `main` en déclarant et initialisant le plateau de jeu.

Question 2

Écrivez la fonction `afficher_plateau` qui prend en paramètres un plateau de jeu et un tableau de couleurs et affiche chaque case du plateau en fonction de la couleur correspondant à son contenu. Toute case vide sera affichée de la couleur de la fenêtre. Dans la fonction `main`, "décommentez" l'appel à la fonction `afficher_plateau` et complétez le.

Attention :

- les colonnes correspondent à l'axe des abscisses de la fenêtre graphique et les lignes à celui des ordonnées,
- lorsqu'une fonction a en paramètre un tableau à deux dimensions, il faut que sa signature contienne la valeur de chacune des deux dimensions.

NB : Pour que les différentes étapes du jeu ne se superposent pas il faut "effacer" le tableau avant de l'afficher à nouveau. Ceci peut se faire facilement en remplissant la fenêtre de sa couleur de fond (utilisation de la fonction `CINI_fill_window`).

La création d'une pièce

Question 3

Pour choisir le type de la prochaine pièce créée, la boucle de jeu tire aléatoirement une valeur entre 0 et 6 (inclus). L'initialisation consiste ensuite à aller lire le tableau des pièces pour recopier dans la pièce créée les informations correspondant au type tiré. La pièce créée est positionnée en haut et au milieu de la fenêtre.

NB : nous avons fait le choix, pour les fonctions qui modifient une pièce, de passer l'adresse d'une pièce en paramètre, plutôt que d'avoir un résultat de type pièce. En effet, dans ce dernier cas, il faudrait systématiquement recopier *tous* les champs de la pièce modifiée dans la variable résultat, alors qu'en travaillant directement sur la pièce, on peut n'affecter que les champs modifiés. Nous avons fait ce choix aussi pour la fonction d'initialisation, par souci d'homogénéité.

Vous disposez de la fonction `initialiser` qui prend en paramètre l'adresse d'une pièce, les caractéristiques de la pièce créée (la bonne ligne du tableau `tab_pieces`), le type de la pièce créée. Dans la fonction `main`, "décommentez" l'appel à la fonction `initialiser` et complétez le.

Question 4

Pour afficher la pièce créée, nous allons modifier le contenu de la fenêtre graphique, sans modifier le contenu du plateau de jeu. Celui-ci ne sera mis à jour qu'une fois que la pièce aura atteint sa position définitive.

Vous disposez de la fonction `afficher_piece` qui prend en paramètre une pièce et une couleur et qui affiche la pièce dans la fenêtre graphique. Dans la fonction `main`, "décommentez" les trois appels à la fonction `afficher_piece` et complétez les.

Les déplacements

La seule action que nous prenons en compte est la frappe d'une touche. Toute frappe entraîne la descente de la pièce. En fonction de la touche choisie, un mouvement peut être ajouté :

- flèche gauche : décalage de la pièce sur la gauche ;
- flèche droite : décalage de la pièce sur la droite ;
- touche `d` : rotation à gauche ;
- touche `g` : rotation à droite ;

Vous pouvez inverser l'effet des touches `d` et `g` en remplaçant, dans le `switch` de la fonction `main`, `SDLK_d` par `SDLK_g` et inversement.

La flèche vers le bas provoque la création d'une nouvelle pièce et la touche `escape` la fin de la partie.

Après chaque mouvement, il faut réafficher le plateau de jeu.

Question 5

Le mouvement de base d'une pièce est la descente. Avant de faire descendre la pièce, il faut s'assurer que ce déplacement est possible, donc qu'aucune partie de la pièce ne sort de la fenêtre et que toutes les cases du plateau que la pièce occupera dans sa nouvelle position sont vides.

Si la descente est possible, on modifie les coordonnées de la pièce, sinon celle-ci a atteint sa position définitive et ne pourra donc plus descendre, il faut alors mettre à jour le plateau de jeu en marquant les emplacements occupés par cette pièce.

Écrivez et testez la fonction `descendre` qui prend en paramètre le plateau de jeu et l'adresse d'une pièce, qui modifie la pièce si celle-ci s'est déplacée, et qui renvoie un 0 si la pièce est bloquée et 1 sinon (ce résultat sera utilisé dans la suite, en particulier dans la version temporisée du jeu). Dans la fonction `main`, "décommentez" l'appel à la fonction `descendre` et complétez le.

N.B. : cette fonction réalise un mouvement élémentaire : la pièce ne descend que d'un niveau. Elle sera normalement appelée en boucle jusqu'au blocage de la pièce. Le plateau n'est mis à jour que lorsque la pièce est bloquée. Si vous faites apparaître une nouvelle pièce alors que la précédente n'était pas arrivée à sa position définitive, la pièce précédente disparaîtra (elle n'aura pas été enregistrée sur le plateau).

Question 6

Une pièce peut aussi se déplacer latéralement. Écrivez les fonctions `decaler_gauche` et `decaler_droite` qui modifient les coordonnées de la pièce en fonction du mouvement demandé lorsque celui-ci est possible. Si le mouvement est impossible, les fonctions sont sans effet. Ces fonctions prennent les mêmes paramètres que la fonction `descendre`. Dans la fonction `main`, "décommentez" les appels à ces fonctions et complétez les.

Question 7

Enfin, une pièce peut tourner sur elle-même d'un angle de $90°$, vers la droite ou vers la gauche. Compte tenu de l'orientation du repère dans la fenêtre graphique, une rotation vers la droite modifie les coordonnées relatives du point (x, y) en (x', y') avec $x' = -y$ et $y' = x$ alors qu'une rotation vers la gauche aboutit à $x' = y$ et $y' = -x$ (voir Figure 3).

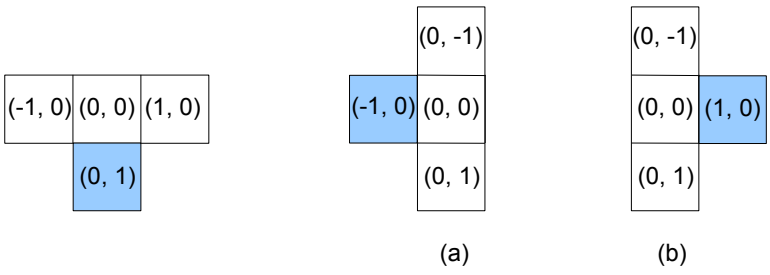

FIGURE 3 – Exemple de rotation droite (a) et gauche (b) : le tétromino 'T'

Écrivez les fonctions `rotation_gauche` et `rotation_droite` qui modifient les coordonnées de la pièce en fonction de la rotation demandée lorsque celle-ci est possible. Ces fonctions prennent les mêmes paramètres que la fonction `descendre`. Dans la fonction `main`, "décommentez" les appels à ces fonctions et complétez les.

Une fois que la pièce est bloquée

Lorsqu'une pièce ne peut plus être déplacée, il faut vérifier si son insertion permet de compléter des lignes. Si c'est le cas, toutes les lignes complètes doivent être supprimées et les lignes situées au-dessus doivent être décalées vers le bas.

Question 8
Écrivez une fonction `supprimer_lignes` qui prend en paramètre le plateau de jeu et qui efface dans le plateau de jeu toutes les lignes complètes. Dans la fonction `main`, "décommentez" l'appel à cette fonction et complétez le.

Question 9
La partie se termine lorsque l'ensemble des pièces empilées atteint le haut de la fenêtre. Écrivez une fonction `partie_perdue` qui prend en paramètre le plateau de jeu et qui détermine si la partie est terminée ou non. Modifiez la boucle de jeu pour prendre en compte cette nouvelle condition de terminaison.

Déplacements rapides

Question 10
L'action `hard_drop` consiste à faire tomber directement une pièce lorsque sa position latérale a été choisie.

Ecrivez la fonction `hard_drop` qui prend les mêmes paramètres que la fonction `descendre`. Dans le `switch` de la fonction `main`, "décommentez" le cas dans lequel l'appel à la fonction `hard_drop` a lieu et complétez l'appel (l'appel à la fonction est associé à une action sur la touche `espace`, cas `SDLK_SPACE`).

Exercice 39 – Tetris : scores et niveaux

Nous allons, dans cet exercice, ajouter des fonctionnalités supplémentaires au jeu liées à la gestion d'une horloge pour faire descendre automatiquement les pièces à une vitesse dépendant du niveau du joueur.

Tetris ne se termine jamais par la victoire du joueur. Avant de perdre, le joueur doit tenter de compléter un maximum de lignes pour obtenir le score le plus élevé possible. Au niveau initial (le niveau `0`), les points sont comptés de la manière suivante :

- une seule ligne complétée rapporte 40 points ;
- deux lignes complétées rapportent 100 points ;
- trois lignes complétées rapportent 300 points ;
- quatre lignes complétées rapportent 1200 points ;

Le niveau détermine la vitesse de la chute des pièces. Plus le niveau est élevé, plus les pièces tombent vite. Le nombre de points est augmenté à chaque niveau selon l'équation $nb\_pts(n) = (n+1).p$, où p est le nombre de points au niveau 0 et n le niveau.

Le niveau augmente de 1 toutes les `CHANGEMENT_NIVEAU` lignes supprimées.

Question 1
Récupérez le fichier `Tetris2.c` qui inclut une gestion d'horloge pour la descente des pièces, insérez-y vos déclarations de types, de variables, vos fonctions, complétez les appels et la condition de la boucle externe du jeu (tout ce que vous avez fait jusqu'à présent). Testez alors le fonctionnement de votre programme dans ce nouvel environnement. Si vous rencontrez des problèmes lors de la compilation ajoutez l'option `-lSDL` à votre commande.

Question 2
Modifiez la fonction `supprimer_lignes` pour prendre en compte le comptage des points. La fonction devra renvoyer le nombre de lignes supprimées et mettre à jour le score. La valeur de retour sera utilisée pour mettre à jour la variable `cpt` qui compte le nombre de lignes supprimées, vous devez donc aussi modifier l'appel à `supprimer_lignes` et déclarer la variable stockant le score.

Pour gérer les changements de niveau, vous pouvez "décommenter" les lignes de code agissant sur la variable `cpt`. Ces instructions diminuent la période de l'horloge de `DIMINUTION_PERIODE` ms et augmentent le niveau lorsque c'est nécessaire (la variable `niveau` est déjà déclarée et initialisée). Vous pouvez changer les valeurs associées à `CHANGEMENT_NIVEAU` et `DIMINUTION_PERIODE`.

Question 3
Pour pouvoir afficher l'évolution du score, nous allons augmenter la largeur de la fenêtre graphique, afin de réserver une partie dans laquelle se fera cet affichage.

Modifiez votre programme pour que :

- la fenêtre affichée soit deux fois plus large,
- une ligne verticale sépare la fenêtre en deux (une moitié sera le plateau de jeu, l'autre sera consacrée à l'affichage),
- modifiez la fonction `afficher_plateau` pour que seule la partie de la fenêtre représentant le plateau de jeu soit "effacée".

Question 4

Nous devons maintenant afficher le score dans la partie de la fenêtre prévue à cet effet.

Comme nous ne disposons pas d'une fonction permettant directement d'afficher graphiquement un entier, nous allons utiliser la fonction `CINI_draw_int_table` et le fait qu'un tableau d'entiers est un pointeur sur un entier.

Ecrivez une fonction `afficher_score` qui affiche le score au bon endroit dans la fenêtre graphique. Dans la fonction `main`, faites en sorte que le score soit à nouveau affiché chaque fois qu'il est modifié.

Semaine 8 - TME

Objectifs

— Bibliothèque : programme découpé en plusieurs fichiers
— Représentation des listes
— Les différents parcours de liste

Exercices

Si vous devez recopier des fichiers pour réaliser ce TP, ces derniers se trouvent dans le répertoire `/Infos/lmd/2019/licence/ue/LU1IN002-2020fev/Semaine8`.

Comme pour les exercices de TD, les exercices de TME s'appuieront sur la structure de données suivante :

```
typedef struct _cellule_t cellule_t;
struct _cellule_t{
  int donnee;
  cellule_t *suivant;
};
```

Exercice 40 – Plusieurs fichiers .c pour un programme

Les types et fonctions définis dans un programme peuvent être utiles à différents programmes, il est alors intéressant de les regrouper dans un même fichier qui sera ensuite inclus par les programmes en ayant besoin. Un tel fichier est appelé "bibliothèque", cela permet d'éviter de dupliquer du code, ce qui est un très bonne pratique.

Recopiez les fichiers `liste_entiers.h`, `liste_entiers.c` et `test_liste.c`. La bibliothèque `liste_entiers` est utilisée par le programme `test_liste`.

- Le fichier `liste_entiers.h` contient la définition du type `cellule_t` et la signature de la fonction `creerListe` (dont vous n'avez pas besoin de comprendre le fonctionnement pour cette séance). Ce fichier `.h` est appelé *header*.
- Le fichier `liste_entiers.c` contient la définition de la fonction `creerListe` et inclut le fichier `liste_entiers.h` pour avoir accès au type `cellule_t`.
- Le fichier `test_liste.c` contient la fonction `main` permettant de tester la fonction définie dans le fichier `liste_entiers.c`. Le fichier `liste_entiers.h` est donc inclus dans le fichier `test_liste.c`.

Pour compiler un programme découpé en plusieurs fichiers, la manière la plus simple est de procéder comme habituellement mais en incluant dans la commande de compilation *tous les fichiers source* utilisés par le programme.

Question 1

Compilez le programme précédent en exécutant la commande suivante et utilisez ddd pour visualiser la liste construite. La compilation signale un avertissement que vous ignorerez pour cette question.

```
gcc -Wall -g -o test_liste liste_entiers.c test_liste.c
```

Question 2

Vous allez maintenant enrichir la bibliothèque `liste_entiers` en y ajoutant une fonction **`void`** `AfficherListeInt(cellule_t *liste)` qui affiche le champs `donnee` de tous les éléments de `liste`. Vous devez alors :

- ajouter la signature de la fonction au fichier `liste_entiers.h` ;
- ajouter la définition de la fonction au fichier `liste_entiers.c` ;
- modifier le fichier `test_liste.c` pour tester la fonction `AfficherListeInt`.

Exercice 41 – Parcours de listes

Les fonctions suivantes sont à ajouter à la bibliothèque `liste_entiers`. Vous devez donc pour chacune d'elles,
- ajouter la signature de la fonction au fichier `liste_entiers.h` ;
- ajouter la définition de la fonction au fichier `liste_entiers.c` ;
- modifier le fichier `test_liste.c` pour tester la nouvelle fonction.

Question 1
Écrivez une fonction `int nb_occurences(int val, cellule_t *liste)` qui renvoie le nombre de fois où la valeur `val` apparaît dans la liste `liste`.

Question 2
Écrivez une fonction `int tous_plus_grand(int val, cellule_t *liste)` qui renvoie `1` (vrai) si tous les éléments de la liste sont supérieurs ou égaux à la valeur `val`, `0` sinon.

Question 3
Écrivez une fonction `cellule_t* Maximum(cellule_t *liste)` qui renvoie un pointeur vers la cellule dont le champ `donnee` a la valeur maximum dans la liste. Si plusieurs cellules vérifient cette condition, la fonction renvoie un pointeur vers la première rencontrée.

Question 4
Écrivez une fonction `int Renvoyer_val_element_pos(int pos, cellule_t* liste)` qui renvoie la valeur du champ `donnee` de l'élément en position `pos` de la liste (le premier élément est en position `0`).

Nous ferons l'hypothèse que la fonction est toujours appelée avec une valeur de `pos` inférieure au nombre d'éléments de la liste et supérieure ou égale à 0.

Question 5
Écrivez une fonction itérative `cellule_t *Concatener_it(cellule_t *liste1, cellule_t *liste2)` qui renvoie la liste obtenue en ajoutant les éléments de `liste2` à la fin de `liste1`. Si l'une des deux listes est vide, la fonction renvoie l'autre liste. Si les deux listes sont vides, la fonction renvoie `NULL`. Si aucune des deux listes n'est vide, la fonction modifie `liste1` et renvoie la tête de la liste après ajout des éléments de `liste2`.

Question 6
Écrivez une fonction `int nb_maximum(cellule_t *liste)` qui calcule et renvoie le nombre de fois où la valeur maximale est présente dans la liste. La liste ne devra être parcourue qu'*une seule fois*.

Exercice 42 – Finir le TD

Faites les exercices non terminés lors de la séance de TD. Ajoutez les fonctions demandées à la bibliothèque `liste_entiers`

Semaine 9 - TME

Objectifs

— Ajout d'un élément dans une liste
— Suppression d'un élément d'une liste
— Parcours simultané de plusieurs listes

Exercices

Si vous devez recopier des fichiers pour réaliser ce TP, ces derniers se trouvent dans le répertoire `/Infos/lmd/2019/licence/ue/LU1IN002-2020fev/Semaine9`.

Nous allons considérer des listes représentant des multi-ensembles. Un multi-ensemble est un ensemble dans lequel chaque élément peut apparaître plusieurs fois. Nous utilisons les types suivants pour représenter un élément de la liste. Le champ `frequence` correspond au nombre de fois où `valeur` apparaît dans l'ensemble. Deux éléments de la liste ne doivent donc jamais avoir la même `valeur`.

```
typedef struct _element_t element_t;
struct _element_t{
  int valeur;
  int frequence;
  element_t *suivant;
};
```

Recopiez les fichiers `multi_ensembles.h`, `multi_ensembles.c` et `test_multi_ensembles.c`. La bibliothèque `multi_ensembles` est utilisée par le programme `test_multi_ensembles`.

Comme en TD, les fonctions que vous allez écrire cette semaine modifient la liste sur laquelle elles s'appliquent soit en y ajoutant un élément soit en en supprimant un. Si l'ajout se fait en tête de liste ou si le premier élément de la liste est supprimé, la tête de liste est modifiée, il faut donc récupérer sa nouvelle valeur. Ces fonctions vont donc prendre en paramètre un pointeur sur un élément de liste (le premier élément de la liste avant modification) et renvoyer un pointeur sur un élément de liste (le premier élément de la liste modifiée).
Attention, ces fonctions modifient la liste initiale et renvoient le pointeur sur le premier élément de la liste après modification. La liste initiale n'existe plus après appel à l'une de ces fonctions, elle a été modifiée.

Exercice 43 – Création d'un multi-ensemble

Question 1

Complétez la fonction `Recherche_val` pour qu'elle renvoie un pointeur sur le premier élément de valeur `val` du multi-ensemble. La fonction renvoie `NULL` si la valeur ne se trouve pas dans le multi-ensemble.

Question 2

Complétez la fonction `Ajout_tete_ensemble` pour qu'elle ajoute au multi-ensemble passé en paramètre un élément de valeur `val` et de fréquence `freq`. Si la valeur apparaît déjà dans le multi-ensemble, sa fréquence sera augmentée de `freq`, sinon un nouvel élément sera créé avec la fréquence `freq` et ajouté en tête de liste. La fonction renvoie la tête de la nouvelle liste. Vous pouvez bien-sûr faire appel à la fonction `Recherche_val` pour déterminer si la valeur est présente dans le multi-ensemble ou non.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Exercice 44 – Suppression d'un élément d'un multi-ensemble

Question 1

Ajoutez à la bibliothèque une fonction `Supprime_total_element_ensemble` qui prend en paramètre un multi-ensemble et une valeur et qui supprime du multi-ensemble l'élément de la valeur passée en paramètre. L'element ne doit plus appaître dans l'ensemble même si initialement sa fréquence était supérieure à 1. Si la valeur ne se trouve pas dans le multi-ensemble, ce dernier n'est pas modifié. La fonction doit renvoyer le pointeur sur le premier élément du multi-ensemble après suppression.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Question 2

Ajoutez à la bibliothèque une fonction `Supprime_element_ensemble` qui cette fois va mettre a jour la fréquence associée à la valeur à supprimer. Si la fréquence est supérieure à 1, elle est simplement diminuée de 1. Si la fréquence est égale à 1, la valeur est supprimée du multi-ensemble. Comme pour la fonction précédente, si la valeur ne se trouve pas dans le multi-ensemble, ce dernier n'est pas modifié. La fonction doit renvoyer le pointeur sur le premier élément du multi-ensemble après suppression.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Exercice 45 – Ajout et suppression dans un multi-ensemble trié

Nous allons maintenant créer un multi-ensemble en triant ses éléments par valeur croissante et supprimer des éléments de cet ensemble en prenant en compte son caractère trié.

Question 1

Ajoutez à la bibliothèque la fonction `Ajout_ensemble_trie` pour qu'elle ajoute au multi-ensemble passé en paramètre un élément de valeur `val` et de fréquence `freq` en respectant l'ordre, après ajout l'ensemble est toujours trié en ordre coissant des valeurs. Si la valeur apparaît déjà dans la liste, sa fréquence sera augmentée de `freq`, sinon un nouvel élément sera créé et ajouté à la bonne place dans la liste, avec la fréquence `freq`. La fonction renvoie la tête de la nouvelle liste.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Question 2

Ajoutez à la bibliothèque les fonctions `Supprime_element_ensemble_trie` et `Supprime_total_element_ensemble_trie` qui tiennent compte du caractère trié du multi-ensemble pour supprimer un élément du multi-ensemble (diminuer sa fréquence de 1 ou l'enlever complètement de l'ensemble).

Modifiez le fichier `test_multi_ensembles.c` pour tester vos fonctions.

Exercice 46 – Parcours simultané de plusieurs multi-ensembles

Question 1

Ajoutez à la bibliothèque la fonction `Inclus` qui prend en paramètre deux multi-ensembles et renvoie 1 si le premier est inclus dans le deuxième, 0 sinon. Un multi-ensemble `e1` est inclus dans un multi-ensemble `e2` si tout élément de `e1` apparaît dans `e2` avec une fréquence supérieure ou égale. Nous ferons l'hypothèse que les multi-ensembles sont triés en ordre croissant des valeurs et qu'une même valeur n'apparaît qu'une fois dans un multi-ensemble.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Question 2

Ajoutez à la bibliothèque la fonction `Intersection_vide` qui prend en paramètre deux multi-ensembles et renvoie 1 si leur intersection est vide, 0 sinon. Nous ferons l'hypothèse que les multi-ensembles sont triés en ordre croissant des valeurs et qu'une même valeur n'apparaît qu'une fois dans un multi-ensemble.

Modifiez le fichier `test_multi_ensembles.c` pour tester votre fonction.

Semaine 10 - TME

Objectifs

— Récursivité
— Extraction d'une sous-liste
— Parcours simultané de plusieurs listes

Exercices

Dans ce TP vous allez continuer à enrichir la bibliothèque `multi_ensembles`. Vous devez donc compléter les fichiers `multi_ensembles.h`, `multi_ensembles.c` et `test_multi_ensembles.c` que vous avez écrits la semaine dernière. Même si ce n'est pas explicitement demandé, vous devez bien sûr modifier le fichier `test_multi_ensembles.c` pour tester chaque nouvelle fonction.

Exercice 47 – Parcours récursif "simple" et extraction d'une sous-liste

Question 1
Ajoutez à la bibliothèque la fonction **récursive** `taille` qui prend en paramètre un multi-ensemble et qui renvoie le nombre d'élements qu'il contient (un élément sera compté autant de fois que sa fréquence).

Question 2
Ajoutez à la bibliothèque la fonction `Supprime_frequence_inf_seuil` qui prend en paramètres un multi-ensemble et un entier et qui supprime du multi-ensemble tous les éléments dont la fréquence est inférieure à la valeur passée en paramètre. La liste initiale est donc modifiée. La fonction renvoie la nouvelle tête de liste

Exercice 48 – Opérations ensemblistes - Parcours simultané de plusieurs listes

Dans cet exercice vous allez programmer les principales opérations ensemblistes (inclusion, intersection, différence). Une fois l'une de ces opérations réalisée, ne recommencez pas tout à zéro pour les opérations suivantes, demandez-vous quelles sont le différences entre les opérations déjà réalisées et celle que vous voulez ajouter. Vous vous rendrez alors compte que vous n'avez plus grand chose à faire !

Question 1
Ajoutez à la bibliothèque la fonction `Inclus_rec` qui est la version récursive de la fonction `Inclus` demandée la semaine dernière. La fonction prend en paramètre deux multi-ensembles et renvoie 1 si le premier multi-ensemble est inclus dans le second, 0 sinon. Nous ferons l'hypothèse que les multi-ensembles sont triés par ordre croissant des valeurs et qu'une même valeur n'apparaît qu'une fois dans un multi-ensemble.

Question 2
Ajoutez à la bibliothèque la fonction itérative `Union` qui prend en paramètres deux multi-ensembles triés, crée le multi-ensemble égal à l'union des deux multi-ensembles passés en paramètre et renvoie un pointeur sur le premier élément du nouveau multi-ensemble (les deux multi-ensembles passés en paramètres ne sont pas modifiés). Dans le nouveau multi-ensemble, chaque valeur ne doit apparaître qu'une seule fois avec la bonne fréquence. Pour ajouter un nouvel élément au multi-ensemble résultat, vous ferez appel à la fonction `Ajout_tete_ensemble` (la liste sera alors triée en ordre décroissant des valeurs).

Question 3
Nous souhaitons maintenant que le multi-ensemble résultat de l'union de deux multi-ensemble soit trié en ordre croissant sur les valeurs des éléments. Nous ne souhaitons pas remplacer les appels à la fonction `Ajout_tete_ensemble` par des appels à la fonction `Ajout_ensemble_trie` car dans ce cas, la liste serait parcourue entièrement à chaque ajout. Nous allons écrire une fonction qui évite ce parcours.

Ajoutez à la bibliothèque la fonction `Ajout_suivant` qui prend en paramètres un pointeur sur un élément d'un multi-ensemble, une fréquence et une valeur et qui ajoute à la suite de l'élément passé en paramètre un nouvel élément dont la valeur et la fréquence correspondent aux paramètres. Le suivant du nouvel élément sera le suivant de l'élément passé en paramètre. La fonction renvoie l'adresse du nouvel élément créé.

Question 4
Ajoutez à la bibliothèque la fonction `Union_triee` qui agit comme la fonction `Union` mais qui fait appel à la fonction `Ajout_suivant` pour que le multi-ensemble résultat soit trié en ordre croissant des valeurs. N'oubliez pas de modifier la tête de liste lorsque c'est nécessaire.

Question 5
Ajoutez à la bibliothèque la fonction `Union_triee_rec` qui agit comme la fonction `Union_triee` mais qui est récursive. Pour obtenir un ensemble trié dans ce cas, l'ajout d'un élément se fera par appel à la fonction `Ajout_tete_ensemble`.

Question 6
Ajoutez à la bibliothèque la fonction `Intersection_triee` qui prend en paramètres deux multi-ensembles triés, crée le multi-ensemble égal à l'intersection des deux multi-ensembles passés en paramètre et renvoie un pointeur sur le premier élément du nouveau multi-ensemble (les deux multi-ensembles passés en paramètres ne sont pas modifiés). Dans le nouveau multi-ensemble, chaque valeur ne doit apparaître qu'une seule fois avec la bonne fréquence (minimum entre la fréquence de chacune des listes). Si l'intersection est vide, la fonction renvoie `NULL`. Le multi-ensemble résultat est trié par ordre croissant des valeurs. À vous de choisir si vous souhaitez écrire une fonction récursive ou non.

Question 7
Ajoutez à la bibliothèque la fonction `Difference_triee` qui prend en paramètres deux multi-ensembles triés, crée le multi-ensemble contenant les éléments présents dans le premier paramètre mais pas dans le second et renvoie un pointeur sur le premier élément du nouveau multi-ensemble (les deux multi-ensembles passés en paramètres ne sont pas modifiés). Dans le nouveau multi-ensemble, chaque valeur ne doit apparaître qu'une seule fois avec la bonne fréquence . Si le résultat ne contient aucune valeur, la focntion renvoie `NULL`. Le multi-ensemble résultat est trié par ordre croissant des valeurs. À vous de choisir si vous souhaitez écrire une fonction récursive ou non.

Les éléments du multi-ensemble résultat et leur fréquence seront calculés de la façon suivante pour chaque valeur `val`. Nous notons `f1` la fréquence de la valeur dans le premier multi-ensemble et `f2` la fréquence dans le second multi-ensemble (si la valeur n'est pas présente dans un multi-ensemble cette fréquence sera égale à 0) ;

- si `f1` $\leq$ `f2`, le multi-ensemble résultat ne contiendra pas la valeur `val` ;
- Si `f1` $>$ `f2`, le multi-ensemble résultat contiendra `f1` - `f2` fois la valeur `val`.

Question 8
Ajoutez à la blibliothèque la fonction `Xor_triee` qui prend en paramètres deux multi-ensembles triés, crée le multi-ensemble trié contenant les éléments présents dans un des deux multi-ensembles mais pas dans les deux et renvoie un pointeur sur le premier élément du nouveau multi-ensemble (les deux multi-ensembles passés en paramètres ne sont pas modifiés). Vous ferez appel aux fonctions `Union_triee` et `Difference_triee`.

Question 9
L'utilisation des fonctions `Union_triee` et `Difference_triee` a deux inconvénients :

- plusieurs parcours des mêmes listes,
- création de multi-ensembles dont l'usage n'est que temporaire, sans libération de la mémoire utilisée.

Dans cette question nous nous intéressons au problème de la gestion de la mémoire. Pour ne pas utiliser inutilement de la mémoire, il est nécessaire de supprimer les multi-ensembles dont l'usage n'est que temporaire une fois qu'ils ne sont plus utilisés.

Ajoutez à la blibliothèque la fonction `Detruire` qui prend en paramètre un multi-ensemble et qui libère toute la mémoire qu'il occupe, la fonction ne renvoie rien.

En utilisant la fonction `Detruire`, modifiez la fonction `Xor_triee` pour que les multi-ensembles dont l'usage est temporaire soient détruits. Après l'appel à la fonction seuls les multi-ensembles passés en paramètres et le résultat doivent occuper de la mémoire.

Semaine 11 - TME

Objectifs

— Évaluation

Exercices

Cette semaine est consacrée à une évaluation de TP qui durera 1h45 par étudiant. Chaque étudiant ne sera donc présent qu'à une des deux séances de la semaine. La répartition des étudiants sur les deux séances vous sera donnée par votre enseignant.

En plus de répondre aux questions de programmation, vous devrez savoir :

— créer un répertoire ;
— recopier des fichiers d'un répertoire à un autre ;
— compiler un programme ;
— produire un exécutable à partir de plusieurs fichiers .c ;
— soumettre le contenu d'un répertoire.

Semaine 12 - TME

Objectifs

— Pour aller plus loin

Exercices

Ce dernier TME n'utilise aucune nouvelle connaissance. Il permet juste de revoir et d'appliquer les notions vues jusqu'à présent à un petit problème, et de construire un programme un peu plus complexe que d'habitude. Ce dernier TME n'est à faire que si vous avez déjà fini tous les exercices des séances précédentes.

Si vous devez recopier des fichiers pour réaliser ce TP, ces derniers se trouvent dans le répertoire `/Infos/lmd/2019/licence/ue/LU1IN002-2020fev/Semaine12`.

L'objectif du TME est de programmer une interface et (au moins) un joueur automatique pour le jeu Othello (ou Reversi).

Othello se joue à 2, sur un plateau unicolore de 64 cases (8 sur 8), avec des pions bicolores, noirs d'un côté et blancs de l'autre. Le but du jeu est d'avoir plus de pions de sa couleur que l'adversaire à la fin de la partie, celle-ci s'achevant lorsque aucun des deux joueurs ne peut plus jouer de coup légal, généralement lorsque les 64 cases sont occupées. Au début de la partie, la position de départ est indiquée figure 4 . Les noirs commencent. Le pion en haut à gauche de la figure correspond au prochain pion à poser (donc à la couleur du joueur dont c'est le tour).

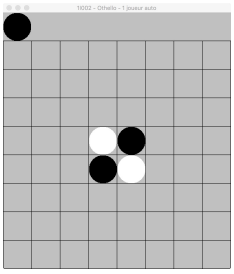

FIGURE 4 – Plateau de jeu de Othello au départ

Chacun à son tour, les joueurs vont poser un pion de leur couleur sur une case vide, adjacente à un pion adverse. Chaque pion posé doit obligatoirement encadrer un ou plusieurs pions adverses avec un autre pion de sa couleur, déjà placé. Le joueur retourne alors le ou les pions adverse(s) qu'il vient d'encadrer. Les pions ne sont ni retirés du plateau de jeu, ni déplacés d'une case à l'autre. On peut encadrer des pions adverses dans les huit directions et plusieurs pions peuvent être encadrés dans chaque direction.

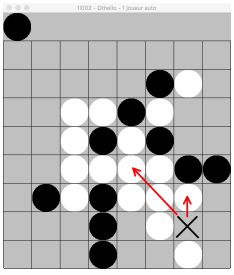

FIGURE 5 – Plateau de jeu de Othello en cours de jeu

Par exemple (cf. figure 5), si le joueur noir joue à l'endroit marqué par une croix, il retourne deux pions blancs en diagonale et un autre pion au dessus. Il n'y a pas de réaction en chaîne : les pions retournés ne peuvent pas servir à en retourner d'autres lors du même tour de jeu.

Si aucune case vide ne permet le retournement de pions adverses, le joueur est bloqué et passe son tour et c'est à l'adversaire de jouer.

La partie se termine si les deux joueurs sont bloqués ou si toutes les cases sont remplies.

Le programme sera organisé en quatre fichiers `.c` et trois fichiers `.h` que vous devez recopier :

— `Affichage.c` et `Affichage.h` qui contiennent toutes les fonctions permettant l'affichage du plateau de jeu. Toutes ces fonctions vous sont fournies.
— `ListePos.c` et `ListePos.h` qui contiendront toutes les fonctions de manipulation de listes que vous écrirez.
— `Othello.c` et `Othello.h` qui contiendront toutes les fonctions concernant le jeu lui même.
— `Main.c` qui contiendra la fonction `main`.

Le plateau de jeu est un tableau à deux dimensions de taille `H`×`H`, `H` étant définie (avec un **`#define`**) comme valant 8 dans `Othello.h`. La case en haut à gauche sera la case `(0,0)` Chaque case de ce tableau peut soit être vide, soit être occupée par un pion noir, soit être occupée par un pion blanc. Dans le fichier `Othello.h` sont aussi définies ces trois valeurs possibles :

```
#define VIDE 0
#define NOIR 1
#define BLANC 2
```

Exercice 49 – Règles du jeu Othello

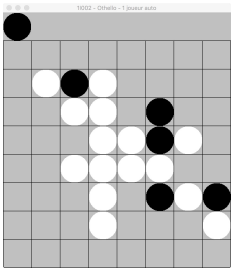

FIGURE 6 – Plateau de jeu de Othello en cours de jeu

Question 1

Le plateau de jeu d'une partie en cours est présenté figure 6. Pour chacune de ces cases, dites si la position est jouable ou non pour les noirs et si oui, combien de pions blancs vont être retournés. Dans les couples `(i,j)`, `i` correspond à la ligne et `j` à la colonne.

- `(2,1)`
- `(3,2)`
- `(5,2)`
- `(7,2)`

Question 2

Quel est le nombre de positions jouables pour les noirs pour le plateau présenté figure 7 ?

Exercice 50 – Affichage du plateau, prise en main de l'affichage

Les fonctions d'affichage que vous aurez à utiliser sont :

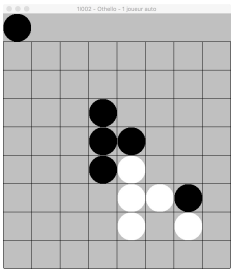

FIGURE 7 – Plateau de jeu de Othello en cours de jeu

- **void** `Creer_fenetre(`**char** `*ModeStr);` qui prend en argument le titre de la fenêtre (correspondant au mode de jeu : 2 joueurs...) et crée une fenêtre avec le plateau de jeu ;
- **void** `Detruire_fenetre();` qui détruit la fenêtre avec le plateau de jeu ;
- **int** `Loop_until_play(`**int** `plateau[H][H],` **int** `*pi,` **int** `*pj,` **int** `joueurCourant);` qui prend en argument le plateau de jeu, deux pointeurs vers des entiers et le joueur courant (`NOIR` pour le joueur des pions noirs et `BLANC` pour le joueur des pions blancs). Cette fonction attend que le joueur clique dans la fenêtre et met à jour (les variables dont l'adresse est passée en paramètre) avec les indices de la case du plateau où le joueur a cliqué. Cette fonction renvoie -1 si la fenêtre a été fermée (0 sinon).

Question 1

Dans le fichier `Othello.c`, vous complèterez la fonction

void `Initialiser_plateau(`**int** `plateau[H][H]);`

qui met toutes les cases du plateau donné en argument à vide sauf les 4 cases du milieu qui doivent contenir des pions blancs ou noirs comme présenté précédemment.

Question 2

Dans le fichier `Othello.c`, vous complèterez la fonction **int** `Autre_joueur(`**int** `joueur);` qui prend en argument la couleur du joueur et qui renvoie `BLANC` (2) si le joueur est `NOIR` (1) et `NOIR` (1) sinon.

Question 3

Dans le fichier `Main.c`, vous complèterez la fonction `main` qui appellera les fonctions précédentes pour :

- initialiser le plateau ;
- créer une fenêtre ;
- Tant que la fenêtre n'est pas fermée, mettre alternativement un pion noir ou un pion blanc là où l'utilisateur a cliqué ; pour ajouter un pion dans une case `(i,j)` donnée il suffit de mettre la couleur jouée dans cette case ;
- détruire la fenêtre.

Pour compiler votre programme, il vous faudra compiler en même temps tous les fichiers `.c` dont vous utilisez des fonctions. Ainsi, dans ce premier exercice, vous devrez compiler à la fois `Affichage.c`, `Othello.c` et `Main.c`. Comme nous utilisons la bibliothèque SDL2, il faut inclure cette bibliothèque ; vous devrez ajouter `` `sdl2-config --libs` `` à votre ligne de commande de compilation qui deviendra [1] :

```
gcc -Wall -o Othello Affichage.c Othello.c Main.c `sdl2-config --libs`
```

Si vous préférez, nous vous avons aussi fourni un *Makefile* qui permet d'effectuer facilement la compilation. Vous apprendrez éventuellement l'année prochaine comment cela fonctionne. Pour l'utiliser, et compiler votre programme, il suffit de taper `make` dans le terminal.

Si tout se passe bien, vous pourrez mettre des pions blancs ou noirs sur le plateau de jeu, mais sans respecter les règles de placement. C'est ce que nous allons programmer dans la suite du sujet.

1. Si vous souhaitez compiler votre programme sous mac, il vous faut avoir la bibliothèque SDL2 et remplacer le `` `sdl2-config --libs` `` avec `-I/Library/Frameworks/SDL2.framework/Headers -framework SDL2`

Exercice 51 – Liste de positions jouables

Nous allons maintenant restreindre les positions où un joueur peut placer son pion aux positions permettant d'encadrer un ou plusieurs pions adverses avec un autre pion de sa couleur, déjà placé. Pour cela, pour une couleur donnée et un joueur donné, nous allons construire la liste des positions possibles. Les fonctions de manipulation de listes sont à écrire dans le fichier `ListePos.c`. La structure d'un élément de liste vous est donnée dans le fichier `ListePos.h`. La voici :

```
typedef struct _cellule_t cellule_t;
struct _cellule_t{
  int i,j;
  int val;
  cellule_t *suivant;
};
```

Elle comporte 4 champs : les coordonnées `i` et `j` de la position sur le plateau, `val` le nombre de points que ce coup rapporterait (*ie.* le nombre de pions adverses qui seraient retournés) et `suivant`, le pointeur vers l'élément suivant de la liste.

Question 1

Dans le fichier `ListePos.c` vous complèterez les fonctions suivantes en vous inspirant des fonctions vues lors des TD et TP précédents :

- `cellule_t *Creer_cellule(int i, int j, int val)` qui prend en argument les coordonnées `i` et `j` ainsi que `val`, le nombre de points associés au coup, qui alloue un élément de liste et qui renvoie un pointeur vers cet élément ;
- `void Afficher_liste(cellule_t *liste)` qui prend en argument une liste et qui affiche celle ci dans le terminale (cela vous permettra de vérifier que vos fonctions sont correctes) ;
- `void Detruire_liste(cellule_t *liste)` qui prend en argument une liste `liste` et qui libère toute la mémoire réservée pour la liste.
- une des deux fonctions suivantes :
  - `cellule_t *Inserer_tete(int i, int j, cellule_t *liste, int val)` qui prend en argument les coordonnées `i` et `j`, le nombre de points associés au coup `val` (qui peut être à 0 si nécessaire) et une liste `liste` (qui peut être `NULL`), qui alloue un nouvel élément de liste, l'ajoute en tête de list et qui renvoie un pointeur vers la liste ;
  - `cellule_t *Inserer_decrois(int i, int j, cellule_t *liste, int val)` qui prend en argument les coordonnées `i` et `j`, le nombre de points associés au coup `val` (qui peut être à 0 si nécessaire) et une liste `liste` (qui peut être `NULL`), qui alloue un nouvel élément de liste, l'ajoute dans la liste qui renvoie un pointeur vers la liste ; attention, la liste est ordonnée par ordre décroissant des valeur du champs `val` ; cet ordre devra être maintenu ;

Vous prendrez soin de bien tester vos fonctions en les appelant (transitoirement) dans la fonction `main`.

Exercice 52 – Une position est elle jouable ?

Pour savoir si une position est jouable nous allons compter le nombre de pions adverses qui seraient retournés si ce coup était joué. Il y a 8 directions possibles dans lesquelles les pions sont retournés : vers le haut, le bas, la gauche, la droite, mais aussi les 4 diagonales. La première fonction `Gain_dir` renverra le nombre de pions retournés pour une direction donnée tandis que `Est_jouable_gain` renverra le nombre de pions retournés dans toutes les directions. Enfin, la fonction `Trouver_liste_pos_jouables` renverra la liste de toutes les positions jouables. Ces trois fonctions sont à compléter dans le fichier `Othello.c`

Question 1

Vous allez écrire la fonction

```
int Gain_dir(int plateau[H][H], int iLigne, int iCol, int dirLigne, int dirCol, int couleurQuiJoue)
```

qui prend en argument le plateau de jeu `plateau`, les coordonnées de la case `iLigne` et `iCol`, la direction (`dirLigne` et `dirCol`) et la couleur du joueur qui joue `couleurQuiJoue`. La direction est représentée par deux entiers valant -1, 0 ou 1. Ainsi, si par exemple `dirLigne` vaut -1 et `dirCol` vaut 0, la direction sera celle de même colonne et d'indices de ligne décroissant, c'est-à-dire vers le haut (pour rappel, la case en haut à gauche est la case d'indice `(0,0)`). La fonction renverra le nombre de pions de couleur adverse potentiellement retournés. Pour rappel, s'il n'y a pas de pion de la couleur du joueur courant, aucun pion ne sera retourné.

Question 2

Vous allez maintenant écrire la fonction

```
int Est_jouable_gain(int plateau[H][H], int iLigne, int iCol, int couleurQuiJoue)
```

qui prend en argument le plateau de jeu `plateau`, les coordonnées de la case `iLigne` et `iCol`, et la couleur du joueur qui joue `couleurQuiJoue`. Cette fonction renverra le nombre de pions de couleur adverse potentiellement retournés dans toutes les directions.

Question 3

Vous allez maintenant écrire la fonction

```
cellule_t *Trouver_liste_pos_jouables(int plateau[H][H], int couleurQuiJoue)
```

qui prend en argument le plateau de jeu `plateau` et la couleur du joueur qui joue `couleurQuiJoue`. Cette fonction renvoie la liste des positions jouables, chaque position contenant aussi le nombre de pions potentiellement retournés dans le champs `val` de la structure `cellule_t`.

Exercice 53 – Limitation du jeu aux positions jouables

Nous voulons maintenant modifier la fonction `main` pour que seules les cases jouables puissent être jouées (*ie.* qu'un pion ne soit ajouté que si la case est jouable). Pour cela, nous auront besoin de savoir si la case sur laquelle l'utilisateur a cliqué est jouable.

Question 1

Ecrivez, dans le fichier `ListePos.c` une fonction

```
int Est_dans_liste(cellule_t *liste, int i, int j)
```

qui prend en argument une liste `liste` et les coordonnées `i` et `j` d'une case et qui renvoie 1 si une case ayant ces coordonnées existe dans la liste et 0 sinon.

Question 2

Vous modifierez ensuite votre fonction `main` pour que seules les cases jouables puissent être jouées. Si l'utilisateur clique sur une case qui n'est pas jouable, le plateau n'est pas modifié et un autre clic dans une case est attendu. Attention à ne pas générer de fuite mémoire avec les listes.

Question 3

Nous allons maintenant réfléchir aux conditions de terminaison du jeu. Le jeu se termine si les deux joueurs sont bloqués, ce qui peut arriver si le plateau est plein, ou non. On sait qu'un joueur est bloqué si sa liste de positions jouables est vide. Vous modifierez votre fonction `main` pour que le jeu s'arrête si la fenêtre est fermée ou que les deux joueurs sont bloqués.

Exercice 54 – Othello à deux joueurs

Pour terminer notre implémentation du jeu Othello il nous faut encore retourner les pions, et compter le score de chaque joueur.

Question 1

Dans le fichier `Othello.c` , écrire la fonction

```
void Retourner_pions(int plateau[H][H], int iLigne, int iCol, int dirLigne, int dirCol, int couleurQuiJoue)
```

qui prend en argument un plateau (`plateau`), une case jouée de coordonnées `iLigne` et `iCol`, une direction (`dirLigne`, `dirCol`) et une couleur de joueur (`couleurQuiJoue`). Elle retournera tous les pions adverses qui seront encadrés par le pions joué et l'autre pion de même couleur le plus proche dans la direction donnée. Attention, il est possible qu'aucun pion ne doive être retourné dans cette direction.

Question 2

Vous complèterez ensuite la fonction

```
void Jouer_pion(int plateau[H][H], int iLigne, int iCol, int couleurQuiJoue)
```

qui pour un plateau (`plateau`), une case jouée de coordonnées `iLigne` et `iCol`, une couleur de joueur (`couleurQuiJoue`) retourne tous les pions adverses qui sont encadrés par le pion joué et un autre pion de même couleur dans toutes les directions.

Question 3

Vous modifierez votre fonction `main` pour que les pions soient retournés lorsqu'un pion est joué.

Question 4

Il serait plus satisfaisant maintenant de savoir qui a gagné. Vous écrirez pour cela une fonction `Nb_pions` (dont vous trouverez le prototype) qui renvoie le nombres de pions noirs et le nombre de pions blancs présents sur le plateau. Vous utiliserez cette fonction dans le `main` pour afficher dans le terminal qui a gagné et de combien.

Exercice 55 – Othello avec un joueur automatique simple

Nous allons maintenant implémenter un joueur automatique simple : il jouera à chaque tour le coup rapportant le plus de points pour ce tour. Le joueur automatique sera toujours le joueur blanc. Pour cela, il suffit de trouver dans la liste des coup possibles le meilleur coup.

Question 1

Ecrivez dans le fichier `ListePos.c` une fonction `Max_liste` - dont vous déterminerez vous même le prototype - qui Renvoie le meilleur coup pour ce tour de jeu. Il y a plusieurs possibilités :

- si votre liste est déjà triée (*ie.* votre fonction d'insertion ajoute les éléments en respectant l'ordre du champs `val`), le premier élément de la liste est le maximum,
- si votre liste n'est pas triée (*ie.* vous pouvez soit trier la liste puis renvoyer le premier élément, soit parcourir la liste pour trouver le maximum.

Question subsidiaire, quelle est, selon vous, la méthode la plus efficace ?

Question 2

Nous avons maintenant deux modes possibles pour notre jeu. Le mode sera choisi par l'utilisateur en ajout un argument (0 ou 1) sur la ligne de commande permettant de lancer le jeu. Ces arguments sont accessibles avec les arguments `argc` et `argv` de la fonction `main`. Suivant l'argument, le second joueur sera donc soit un joueur humain soit le joueur automatique. Vous modifierez votre fonction `main` pour que ce soit le cas.

Exercice 56 – Othello avec un joueur automatique à deux coups

Nous allons maintenant implémenter un joueur automatique normalement un peu plus malin. Nous voulons qu'il cherche le meilleur coup possible en regardant un coup plus loin. Ainsi, pour chaque coup possible au tour i, il supposera que l'adversaire au tour i+1 jouera le meilleur coup et calculera quel est alors le meilleur coup possible au tour i+2.

Question 1

Vous écrirez, dans le fichier `Othello.c` la fonction

`void Joueur_Auto_2(int plateau[H][H], int *pi, int*pj)` qui met à jour les variables pointées par `pi` et `pj` avec les coordonnées du meilleur coup prédit en regardant deux tours plus loin.

Pour cela, les étapes sont :

1. Calculer la liste des positions jouables (coups) du joueur courant au tour *i* ;
2. Pour chaque coup possible `c` :
   (a) Recopier le plateau dans un plateau temporaire ;
   (b) Jouer le coup `c` sur ce plateau temporaire ;
   (c) Calculer la liste des positions jouables du joueur adverse au tour *i+1* ;
   (d) Trouver le coup rapportant le plus de point au joueur adverse au tour *i+1* ; conserver le nombre de points de ce coup ;
   (e) Jouer ce coup sur le plateau temporaire ;
   (f) Calculer la liste des positions jouables du joueur courant au tour *i+2* avec ce plateau temporaire ;
   (g) Trouver le coup rapportant le plus de point au joueur courant ; conserver le nombre de points de ce coup ;
   (h) Si le nombre de points gagnés pour le joueur courant pour ces trois coups successifs est meilleur que ceux qui ont été vus auparavant, conserver le coup `c` comme étant celui à jouer.

N'oubliez pas de détruire les listes au fur et à mesure.

Question 2

Vous modifierez votre fonction `main` pour proposer un troisième mode de jeu avec ce joueur automatique.