TEMIRACLE

Ekaterina Galkina et Zoe Russell (TD 5)

Sommaire

Explication sur la structure et les algorithmes utilisés

Afin d'organiser notre jeu, nous avons créé deux types de listes : une liste simplement chaînée (*ListeS*), qui contient la partie actuelle (c'est-à-dire les éléments courants dans l'ordre choisi par l'utilisateur), et une liste doublement chaînée (*ListeD*) pour stocker toutes les pièces de même type (4 listes selon la forme et 4 - selon la couleur). Ces deux types de listes peuvent être initialisées avec la fonction *ListeS *creeListeS(void)* pour une liste simplement chaînée et *ListeD *creeListeD(void)* pour une liste doublement chaînée. Ces fonctions renvoient des pointeurs vers leur racine qui sera toujours de valeur NULL, et donc pour accéder au premier élément, il faut écrire : *NomDeLaRacine->suiv*. Les attributs des deux listes sont presque identiques, chaque élément a des caractéristiques comme *suiv* (pointeur sur le prochain élément de la liste), *val* (la valeur de l'élément de la liste), et les éléments des listes doublement chaînées ont également *prec* (pointeur sur l'élément précédent de la liste). Vous pouvez retrouver le schéma des listes simplement et doublement chaînées en annexe. Il faut faire attention aux fonctions effectuant des modifications sur les listes : les lettres à la fin des fonctions S ou D indiquent si ces fonctions s'appliquent aux listes simplement ou doublement enchaînées.

Par ailleurs, pour pouvoir simplement manipuler les deux types de listes, comme vous pouvez remarquer dans le dossier du code source, il existe 4 fichiers : *ListesS.h*, *ListesD.h* qui contiennent les définitions et les prototypes des fonctions de liste correspondante et *ListesS.c*, *ListesD.c* contenant le code des fonctions.

Les fonctions pour les listes simplement chaînées (*ListesS.c* et *ListesS.h*)

void supprimerEnTeteS(ListeS *racine)

Cette fonction supprime le premier élément de la liste chaînée. Elle s'exécute en $\Theta(1)$, car cet élément est directement accessible en temps constant indépendamment de la longueur de la liste.

int supprimerOccrurencesMemeCouleurFormeEnTeteS(ListeS *l, int *val) et int supprimerOccrurencesMemeCouleurFormeEnQueueS(ListeS *l, int *val)

Ces fonctions sont utilisées pour détecter et supprimer (s'il y en a) des occurrences de 3 pièces (de même couleur ou de même forme) respectivement au début et à la fin de la liste. Elles renvoient 1 s'il y a une suppression et 0 sinon. L'une des deux est lancée à chaque fois qu'on ajoute un élément dans la liste principale (si une pièce est mise à gauche, alors c'est la première qui est lancée, si à droite -la deuxième). La première s'exécute en $\Theta(1)$, car il suffit juste de vérifier les trois premiers éléments de la liste, par contre la deuxième - en $\Theta(n)$ (ici n est la longueur de la liste), puisque nous sommes obligés de parcourir toute la liste simplement chaînée pour accéder aux 3 derniers éléments.

void ajouterEnTeteS(ListeS *racine, char val[7]) et void ajouterEnQueueS(ListeS *racine, char val[7])

Ces fonctions ajoutent un élément (d'une valeur val) à la première ou à la dernière position de la liste simplement chaînée. La première s'exécute en $\Theta(1)$, car ici aussi, l'élément est directement accessible en temps constant, et la deuxième fonction - en $\Theta(n)$, puisque nous sommes obligés de parcourir tous les éléments pour accéder au dernier élément courant pour en insérer un nouveau juste après.

void afficheS(ListeS *l)

Enfin, la dernière fonction qui affiche tout simplement le contenu de la liste s'exécute en $\Theta(n)$.

Les fonctions pour gérer les listes doublement chaînées (*ListesD.c* et *ListesD.h*)

void decalerListeD(ListeD *liste, int taille)

Cette fonction décale une fois vers la gauche tous les éléments d'une liste doublement chaînée passe en paramètre. Elle s'exécute en $\Theta(1)$, car grâce à l'attribut prec, il est possible d'échanger la racine de la liste avec le premier élément de la liste (ce qui est un moyen simple de faire un décalage) en effectuant un nombre constant d'opérations.

void ajouterEnTeteD(ListeD *racine, char val[7]) et void ajouterEnQueueD(ListeD *racine, char val[7])

Ces fonctions créent et ajoutent un élément avec une valeur val respectivement en tête et en queue d'une liste doublement chaînée dont la racine est passée en paramètre. Les deux s'exécutent en $\Theta(1)$.

void supprimerEnTeteD(ListeD *racine) et supprimerEnQueueD(ListeD *racine)

De même, ces fonctions suppriment soit le premier soit le dernier élément d'une liste doublement chaînée et s'exécutent en temps constant.

void afficheD(ListeD *l)

Affiche le contenu de la liste, dont la racine est passée en paramètres et s'exécute en $\Theta(n)$, car ici, nous sommes obligés de parcourir tous les éléments de la liste.

Les fonctions pour gérer les différentes modifications dans la liste principale et secondaires lors du jeu (*commandesJeu.c* et *commandesJeu.h)*

void nouvelleFigure(char couleurs[5][3], char formes[5][2], char ch[7])

Génère une pièce aléatoire en choisissant au hasard l'indice d'une couleur et d'une forme parmi les 4 possibles pour chacune des caractéristiques (0-3 pour forme et 0-3 pour couleur) et mets sa valeur dans la chaîne de caractères ch sous la forme "\033[3*m*" où le premier * va contenir un nombre de 2 à 5 et le deuxième une lettre R, T, C ou L, représentant le rectangle, triangle, cercle et losange respectivement. Elle s'exécute en $\Theta(1)$.

void figureType(char *val, int *couleurInd, int *formesInd)

Elle affecte à couleurInd et à formesInd les indices de 0 jusqu'à 3 (dans les tableaux de couleurs et de figures) selon le type de la figure dont la valeur val est passée dans les paramètres de la fonction. Elle s'exécute en $\Theta(1)$.

(Attention : Les deux prochaines fonctions décrites ne sont pas utilisées pendant l'exécution du jeu pour les raisons décrites dans la partie des difficultés rencontrées)

void suppressionOccurenceDebutListesSec(int formeCouleurSupprime, ListeD *tab[8], int taillesTabSec[8])

Elle est censée mettre à jour les listes secondaires doublement chaînées après une suppression d'une occurrence de trois éléments au début du plateau. Cette fonction prend en paramètres un entier formeCouleurSupprime qui contient l'indice de 0 à 7 du type selon lequel la suppression a eu lieu (de 0 à 3 c'est selon la forme, 4 à 7 - selon la couleur). Ensuite, ListeD *tab[8] est un pointeur vers un tableau de listes doublement chainées composée de 8 listes : de 0 à 3 se trouvent les listes remplies uniquement d'une certaine forme dans le meme ordre que dans la liste principale (0 - rectangles, 1 - triangles, 2 - cercles, 3 - losanges) et de 4 à 7 d'une certaine couleur (4 - vert, 5 - jaune, 6 - bleu, 7 - rose).

Enfin, le dernier paramètre est un tableau d'entiers int taillesTabSec[8] qui contient les tailles respectives des listes secondaires. Elle doit s'exécuter uniquement si

supprimerOccurencesMemeCouleurFormeEnTeteS(ListeS *l, int *val)

a renvoyé 1, c'est-à-dire que si la suppression a eu lieu dans la liste principale après un ajout d'un élément au début par le joueur. La fonction a une complexité constante, car il suffit de supprimer les trois premiers éléments de la liste secondaire (repertoriée dans tab à l'indice formeCouleurSupprime) élément par élément en regardant la deuxième caractéristique de chaque élément que nous supprimons et de le supprimer dans sa deuxième liste secondaire associée. Donc si nous avons supprimé selon la couleur, il faut regarder les formes des 3 pièces à supprimer et les supprimer dans les listes secondaires des figures au début.

void suppressionOccurenceFinListesSec(int formeCouleurSupprime, ListeD *tab[8], int taillesTabSec[8])

Cette fonction effectue les mêmes opérations que la précédente, mais lorsqu'il y a une suppression à la fin de la liste. Elle est de même de complexité ($\Theta(1)$).

void decalageMisAJourTOUT(ListeS *listePrincipale, int indiceDuTableauDecalage, ListeD *tab[8])

Cette fonction effectue des modifications dans la liste principale et les 5 listes secondaires où le changement pouvait avoir lieu. Elle prend en paramètres la liste principale, un entier indiceDuTableauDecalage qui est l'indice du tableau secondaire (que nous venons de décaler dans le tableau tab des listes doublement chaînées de figures et couleurs). Dans un premier temps, nous mettons à jour la liste principale : nous parcourons simultanément la liste principale élément par élément et la liste secondaire où le changement a eu lieu, puis dès que nous trouvons un élément de la liste secondaire, on recopie la valeur de l'élément de liste secondaire à celle de la liste principale et on avance le pointeur de la liste secondaire vers le prochain élément. Ensuite, nous utilisons une méthode similaire pour mettre à jour les 4 listes secondaires selon le deuxième paramètre. Par exemple, si nous avons décalé une des listes de couleurs, alors il faut s'assurer que tous les éléments des 4 listes des figures sont bien dans le bon ordre. On parcourt la liste principale (qui est déjà modifiée) simultanément avec les 4 listes secondaires pour recopier les valeurs des

éléments de la liste principale dans les éléments des listes secondaires correspondants, dans le bon ordre. Cette fonction s'exécute en Θ(n) puisqu'il faut parcourir tous les éléments de la liste principale.

int StockeIndicesASupprimerApresDecalages(ListeS *listePrincipale, int indiceASupprimer[15])

Cette fonction stocke dans le tableau d'entiers passés en paramètres (int indiceASupprimer[15]) les numéros des éléments dans la liste principale qu'il faut supprimer (s'il y en a) après avoir effectué un décalage. De plus la valeur renvoyée par le programme est la taille de ce tableau (qui est aussi le nombre d'éléments à supprimer). La fonction parcourt chaque élément de la liste principale et calcule les occurrences des couleurs et des figures, puis dès qu'il en trouve une d'au moins 3 pièces, il stocke les indices des pièces correspondantes. Ici de même, nous parcourons tout le plateau du jeu, donc la complexité est linéaire.

void SuppressionMiseAjourTousLesListesApresDecalages(ListeS *listePrincipale, ListeD *tab[8], int indiceASupprimer[15], int tailleIndiceASupprimer, int *taille, int taillesTabSec[8])

Permet d'effectuer des suppressions dans la liste principale et dans les listes secondaires. En effet, cette fonction parcourt simultanément toutes les listes et dès qu'elle tombe sur un élément à supprimer (les numéros des éléments à supprimer sont obtenus grâce à la fonction précédente et sont passés dans les paramètres de cette fonction dans un tableau int indicesASupprimer[15]), nous supprimons cet element des 3 listes suivantes : la liste principale, sa liste secondaire de couleur et sa liste secondaire de forme associée. La complexité de cette fonction est O(n) (n - longueur de la liste principale), puisque dans le pire cas, nous parcourons la liste principale du premier jusqu'au dernier élément à supprimer, si ce dernier élément à supprimer est à la dernière position, nous sommes obligés de parcourir tous les éléments. Donc, en remarquant que toutes les opérations à l'intérieur des itérations s'exécutent en temps constant, la complexité totale du programme est bien O(n).

void reinitialiserListesSecondaires(ListeS *listePrincipale, int taillesTabSec[], ListeD *tab[])

Cette fonction aide à remplir les listes secondaires doublement chaînées à nouveau à partir de la liste principale. Ainsi, d'abord, nous vidons toutes les listes secondaires et ensuite en parcourant la liste principale, nous ajoutons tous les éléments un par un dans l'ordre aux listes secondaires selon leur couleur et forme. Elle s'exécute en Θ(n), car nous avons deux boucles imbriquées (for et while a l'interieur) et une boucle for, mais comme la première boucle for s'exécute un nombre constant de fois (8 - la longueur du tableauListeD *tab[]) et le nombre d'itérations de la boucle while dépend de la taille de la liste principale (si c'est n alors les longueurs des listes secondaires sont majorées par n) donc la premiere partie de la fonction est O(n), car pour chaque iteration de while les deux instructions effectuees ont un cout constant, mais dans la deuxième partie nous sommes obligés de parcourir tous les éléments de la liste principale pour ajouter chaque element dans sa liste de couleur et de forme correspondantes. Donc la deuxième partie s'exécute en Θ(n) (toutes les opérations effectuées lors des itérations sont de complexité constante), donc finalement O(n) + Θ(n) = Θ(n).

Remarque: dans le cadre de ce jeu, il n'y aura jamais plus de 15 pièces sur le plateau de jeu, par conséquent, n ne sera jamais supérieur à 15. Cependant on a quand même décidé de garder n dans l'analyse de complexité des fonctions pour généraliser et avoir une idée de l'efficacité de nos algorithmes pour des n plus grands.

Les fonctions pour gérer les fichiers de données (*gestionFichiers.c* et *gestionFichiers.h)*

int ajoutScoreClassement(FILE *fichierMeilleursScores, int score, char nomClassement[100])

Cette fonction ajoute le score du joueur au classement si celui-ci se trouve dans le top 10. Un nouveau fichier (la copie) est créé dans lequel on recopie les noms des joueurs du

classement (avec le score et le nom associé passés en paramètre) au bon endroit dans la liste. Le fichier du classement original est supprimé et la copie est renommée. Cette fonction s'exécute en temps constant, car on parcourt toujours la même liste des (au plus) 10 noms déjà présents sur le fichier nomsSauvegardes.txt.

int enregistrerPartie(ListeS *listePrincipale, ListeS *figuresAPlacer, int *score, int *taille, char nomClassement[100])

Cette fonction sauvegarde toutes les données nécessaires de la partie en cours dans un fichier texte portant le nom du pseudo de la partie en cours. On recopie simplement ligne à ligne dans le fichier de la sauvegarde les données dans l'ordre suivant : le pseudo, la taille du plateau de jeu actuel, chaque pièce du plateau, puis les 5 pièces des prochaines figures à placer et enfin le score actuel. On doit également ajouter le pseudo au fichier qui recense les noms des parties disponibles. D'abord, on vérifie que le pseudo n'y est pas déjà (c'est le cas lorsqu'on sauvegarde une partie qui a déjà été sauvegardée), ensuite, on recopie le pseudo à la fin du fichier, en prenant soin de toujours laisser un retour à la ligne finale. Cette fonction s'exécute en temps linéaire, car on effectue toujours le même nombre d'opérations pour recopier les données nécessaires (la seule donnée qui varie en fonction de la partie est le plateau de jeu, dont on recopie toujours au plus 14 éléments).

void chargerDonnees(ListeS *listePrincipale, ListeS *figuresAPlacer, int *score, int *taille, char nomClassement[100])

Cette fonction s'occupe de lire les données dans le fichier texte dont le nom est celui passé en paramètre. On lit simplement le fichier ligne par ligne en affectant les valeurs lues aux variables passées en paramètre. Cette fonction s'exécute en temps constant, comme les fonctions précédentes.

int recupererNomsSauvegardes(char tousNomsSauvegardes[100][100], int *tailleNomsSauvegardes)

Cette fonction récupère les noms des parties sauvegardées disponibles (en les plaçant dans tousNomsSauvegardes), en lisant simplement les noms sur le fichier nomsSauvegardes.txt et en renvoyant le nombre de parties disponibles. Cette fonction s'exécute en $\Theta(n)$ avec n le nombre de parties disponibles.

void mettreAJourSauvegardes(char *nomClassement)

Cette fonction s'occupe de mettre à jour les sauvegardes lorsque le joueur a terminé une partie qui a été sauvegardée. Il faut supprimer le pseudo de la partie dans le fichier nomsSauvegardes.txt, mais aussi supprimer le fichier de la sauvegarde. Pour supprimer le pseudo du fichier des noms, nous procédons de la même manière que dans ajoutScoreClassement : il faut créer une copie du fichier dans laquelle on recopie tous les noms sauf celui à supprimer, puis on le renomme et on supprime le fichier original. Cette fonction s'exécute en temps constant.

Les fonctions pour les affichages et les animations (*affichage.c* et *affichage.h*)

void affichePiece(WINDOW *tableauDuJeu, int x, int y, char figures[5][2], int indFigure, int indCouleur)

Cette fonction permet d'afficher une pièce en fonction de sa forme, stockée dans le tableau passe entre les paramètres char figures[5][2] à l'indice indfigure, et sa couleur d'indice indCouleur. Elle s'exécute en temps constant.

void afficheListePieces(WINDOW *win, int x, int y, char figures[5][2], ListeS * listeAAfficher)

Cette fonction permet d'afficher tout le plateau actuel aux coordonnées passées en paramètre (en l'occurrence, ce sera toujours dans la fenêtre du plateau de jeu). Comme cette fonction appelle la fonction figureType et affichePiece (exécutées en temps constant) pour chaque figure, nous en déduisons que afficheListePieces s'exécute en temps linéaire.

void animationPlacemenFigure(WINDOW *tableauDuJeu, int milieu_x, int milieu_y, int taille, int pieceFigure, int pieceCouleur, int sens, char figuresAffichage[5][2])

Cette fonction permet d'animer le placement d'une nouvelle pièce à gauche (si sens = 0) ou à droite (si sens = 1). La nouvelle figure apparaît à peu près au milieu du plateau quelques lignes plus haut et puis nous affichons cette figure de plus en plus s'avançant vers l'endroit où l'utilisateur a choisi de la placer. Ainsi, nous appelons un peu plus que la moitié de la taille du plateau actuel la fonction d'affichage d'une pièce décrite précédemment et en écrasant l'affichage précédent par l'espace avec une pause de 0,1 seconde. Ainsi, la complexité de cette fonction est linéaire.

void animationSuppression(WINDOW *tableauDuJeu, int x, int y, int indiceASupprimer[15], int tailleIndexToDelete, int listePlateauCopie, int taille, char figuresAffichage[5][2])

Grâce à cette fonction, nous pouvons faire clignoter 3 fois les éléments qui vont se supprimer en passant entre paramètre le tableau des entiers indiceASupprimer, qui stocke les indices des éléments qui seront supprimés. Ainsi, cette fonction affiche d'abord le plateau principal passé en paramètre sous la forme d'un tableau d'entiers à deux dimensions (Par exemple, si notre plateau est sous cette forme : R -> C -> L, alors listePlateauCopie associé sera : {{0,0},{2,1},{3,3}} ici donc le premier chiffre stocke l'indice de la figure et le deuxième - celui de couleur) avec les éléments qui seront supprimés

remplacés par les espaces (ce qui permet d'écraser l'affichage précédent du tableau), puis nous attendons 0,2 seconde et affichons tout le tableau en entier et à nouveau attendons 0,2 seconde. Nous répétons ces deux affichages 3 fois (ce qui donnera l'illusion du clignotage). Par conséquent, la complexité de cette fonction est $\Theta(n)$.

Les fonctionnalités apportées

Actuellement, dans notre jeu, comme vous avez pu remarquer, il y a une très belle interface graphique qui se redimensionne en fonction de la taille du terminal du joueur (Vous pouvez également changer les dimensions du terminal pendant le jeu sans relancer le programme). Nous avons aussi fait des petites animations de suppression des pièces et des ajouts à droite et à gauche. De plus, vous pouvez sauvegarder plusieurs parties (votre plateau, votre score et les 5 prochaines figures sont sauvegardés). Si le joueur se trompe dans le choix du nom ou s'il n'y a pas de parties disponibles, une nouvelle partie va se lancer. Vous pouvez aussi visionner le tableau des 10 meilleurs joueurs triés en ordre croissant, ainsi dès qu'une partie est finie si vous avez eu un score assez élevé, alors votre nom que vous avez entré au début du jeu, ainsi que votre score seront dans ce tableau.

Concernant le jeu, les 5 prochaines figures sont affichées et vous pouvez placer des figures à droite, à gauche et faire les décalages à gauche (via les listes des couleurs et des formes) le nombre de fois que vous souhaitez. Si le joueur se trompe dans le numéro des décalages possibles ou demande de faire un décalage alors que ce n'est pas possible (toutes les listes de couleurs et de formes contiennent moins de deux éléments), le programme va l'indiquer. Le jeu s'arrête lorsque le plateau contient 15 pièces. Concernant le score : si vous avez réussi à faire une occurrence de 3 éléments au début ou à la fin, vous avez 30 points en plus de votre score actuel, si vous avez fait une suppression à travers les décalages vous gagnez 15 fois le nombre de figures que vous avez supprimé et, enfin, si vous avez fait une suppression en cascade, vous gagnez 15 * (nombre de figures supprimés) * (tour de cascade).

Exemple de suppression en cascade : Imaginons que votre score est 600 et vous avez le plateau suivant :

Ici en faisant un décalage à travers les triangles nous obtenons ceci :

Vous avez une occurrence au milieu donc votre score sera égal à 600 + 3 * 15 * 1 = 645, et plateau sera :

Ici, comme vous pouvez, voir il y a encore une occurrence donc elle sera supprimée et votre score sera égal à 645 + 4 * 15 * 2 = 765, et le plateau final sera :

Enfin, notre projet se compile sans erreur ni warning avec l'option -Wall et l'allocation et la libération de mémoire sont parfaites (sans l'interface graphique).

Pistes examinées

Au début, nous étions confrontés à un choix : est-ce que nous avons besoin d'implémenter les deux fonctions de suppression :

*(int supprimerOccrurencesMemeCouleurFormeEnTeteS(ListeS *l, int *val)* et

*int supprimerOccrurencesMemeCouleurFormeEnQueueS(ListeS *l, int *val))*,

alors que nous en avons une qui vérifie toutes les occurrences de la liste. Mais en effet, ces deux fonctions sont nécessaires pour rendre le programme plus rapide, puisqu'elles aident à vérifier immédiatement ce que nous venons de modifier. Effectivement, si l'utilisateur ajoute un élément à gauche (resp. à droite), dans ce cas, il est inutile de vérifier toute la liste, puisqu'il n'est pas possible que quelque part sauf les 3 premiers (derniers) éléments de la liste il ait une occurrence. De même, il est inutile ici de vérifier l'existence des 4-uplets/

5-uplets…, car la fonction supprime les occurrences de trois premiers éléments dès que le 3e est ajouté. Par conséquent, la fonction

*int supprimerOccrurencesMemeCouleurFormeEnTeteS(ListeS *l, int *val)*

économise le temps, puisqu'elle a une complexité inférieure à la fonction qui vérifie toutes les occurrences, et la fonction

*int supprimerOccrurencesMemeCouleurFormeEnQueueS(ListeS *l, int *val)*

aussi, car malgré la même complexité, celle-ci ne vérifie par tous les éléments, donc nous avons moins de comparaisons au total.

D'autre part, pour pouvoir stocker la partie actuelle et gérer les décalages et les suppressions des occurrences, nous avons réfléchi à quels types de structures nous avons besoin et à quel moment. Sans doute, une liste simplement chaînée pour la liste principale est nécessaire tout le long du jeu, mais quant aux listes secondaires doublement chaînées, ceci sont utiles uniquement pour effectuer des décalages. Ainsi, nous nous sommes demandées ce qui sera moins coûteux en termes de temps d'exécution : conserver les listes chaînées tout le long de programme (en effectuant des petites modifications à chaque fois que la liste principale est mise à jour) ou bien remplir ces listes à partir de la liste principale uniquement si l'utilisateur a envie de faire un décalage et de les vider dès que le décalage est effectué. On a finalement décidé que la première option était meilleure pour plusieurs raisons :

- Premièrement, concernant le temps d'exécution : comme les fonctions améliorées décrites précédemment pour la mise à jour des listes secondaires s'exécutent en $\Theta(1)$, nous pouvons considérer ces opérations comme peu coûteuses. Cependant, la fonction

  *void reinitialiserListesSecondaires(ListeS *listePrincipale, int taillesTabSec[], ListeD *tab[])*

  mentionnée avant, s'exécute plus lentement en partie suite aux allocations et libération de mémoire, et par conséquent, si le joueur fait souvent des décalages, une telle complexité sera non négligeable.

- Deuxièmement, le maintien des listes de couleurs et de formes tout le long de l'exécution permet de stocker plus facilement les tailles de ces listes, ce qui permet

de détecter si l'utilisateur a la possibilité de faire un décalage. En effet, en un simple parcours de la liste des tailles (qui a un nombre d'éléments fixe (puisque le nombre de listes secondaires est fixe)) nous pouvons savoir s'il existe au moins une liste secondaire qui contient au moins deux éléments (sinon cela ne sert à rien de décaler). Il est, en effet, possible de faire ce test en cherchant le maximum de la liste et de le comparer à 2 (comme nous l'avons fait initialement), mais ce sera mieux de simplement proposer un décalage dès que nous tombons sur une taille supérieure ou égale à 2, ce qui réduit encore plus le temps d'exécution.

Ce choix a néanmoins quelques inconvénients :

- Le programme devient plus complexe ce qui peut entraîner des bugs et des erreurs (dans la partie sur les difficultés rencontrées, nous allons plus détailler cet aspect).

- Le code devient moins facile à comprendre pour un programmeur externe : il y a un nombre plus grand de fonctions, ce qui augmente le temps passé pour se familiariser avec le script.

En outre, au début du projet, nous avons eu des fonctions spéciales pour obtenir les tailles de listes doublement et simplement chaînées, mais finalement, il est beaucoup plus avantageux de stocker et modifier ces valeurs le long d'exécution. Comme dans les fonctions pour la mise à jour des listes il est très simple de savoir le type de changement (suppression ou ajout) et le nombre d'éléments concernées, ainsi une simple incrémentation de décrémentation qui coûtent 2 opérations environ (addition/soustraction et affectation) au bon moment permet d'éviter de lancer des fonctions de tailles qui ont un coût minimum linéaire, ce qui est bien sur non négligeable en prenant en compte combien de fois nous nous servons de la taille des différentes listes (est-ce que le plateau contient 15 pièces, savoir s'il est possible de faire un décalage, quels sont les figures ou couleurs à travers desquels le joueur peut faire un décalage etc...).

Les difficultés rencontrées

Dans la première partie du travail (expérience de l'utilisateur), nous avons rencontré une énorme quantité de bugs avec la mise à jour des listes secondaires. Dans un premier temps, pour la suppression et la mise à jour des éléments dans les listes secondaires, nous avons utilisé à chaque changement la fonction de réinitialisation, qui est simple à implémenter (et ayant une mauvaise complexité) mais au fur et à mesure nous avons réfléchi à d'autres moyens de faire les mises à jour. Cependant, modifier les listes partiellement est effectivement plus complexe et de nombreux cas particuliers et petites subtilités se présentent.

D'autre part, la recherche des occurrences à supprimer utilise également un algorithme pas très évident, qui doit tenir compte des occurrences selon 2 critères (figure et couleur). Des fois, plus d'éléments qu'il fallait se supprimaient à cause d'un mauvais comptage des occurrences ou d'indices etc… Mais en faisant un très grand nombre de tests et en enregistrant à chaque fois les plateaux du jeu qui ont causé des problèmes, nous avons réussi à réduire la quantité de problèmes.

Cependant, notre programme contient toujours une fonction qui fonctionne avec des bugs:

> void suppressionOccurenceFinListesSec(int formColorDeleted, ListeD *tab[8], int taillesTabSec[8]).

Malgré plusieurs relectures du code et affichage de chaque mouvement des variables de cette fonction, nous n'avons pas réussi à résoudre le problème. C'est pour cette raison que notre programme principal utilise actuellement la fonction de réinitialisation au lieu de celle-ci. Pour rappel, cette fonction, qui ne fonctionne pas, est destinée à supprimer des occurrences a la fin du plateau après un ajout d'un élément. Les problèmes ont rarement lieu, la plupart du temps lorsqu'il y a très peu de figures présentes sur le plateau comme par exemple s'il y a une occurrence de 3 éléments et un élément différent. C'est aussi très difficile à corriger puisque lorsque nous avons essayé de reproduire cette situation (exécuter le code avec ce tableau à nouveau avec plus d'affichages pour trouver l'erreur), tout s'exécute parfaitement sans erreurs dans les listes secondaires.

Exemple d'une telle erreur :

le tour actuel : L -> C -> C

Les listes secondaires à ce tour :

-> C -> C (les cercles)

-> L (les losanges)

-> L -> C (Jaune)

-> C (Rouge)

(Les autres listes secondaires sont vides)

La prochaine figure à placer est un C, que l'on met à droite pour obtenir une occurrence et voici ce que nous obtenons de temps en temps (la plupart du temps, exactement la même situation s'exécute sans fautes) :

le tour actuel : L

Les listes secondaires à ce tour :

-> (les cercles)

-> L (les losanges)

-> L -> C (Jaune)

-> (Rouge)

Donc il y a un élément qui n'a pas été supprimé (le C).

Quant à la fonction analogique

(void suppressionOccurenceFinListesSec(int formColorDeleted, ListeD *tab[8], int taillesTabSec[8]))

qui fait aussi des suppressions, mais cette fois au début du plateau, aucun bug n'a été détecté jusqu'à présent, mais comme elle utilise exactement le même algorithme que la première fonction, nous avons pris la décision de ne pas risquer et mettre également à sa place la fonction de réinitialisation.

En outre, nous avons eu des problèmes avec l'exécution du code avec la commande valgrind. Cette vérification supplémentaire a ouvert nos yeux à encore une tonne d'erreurs. Bien qu'en ce moment-là, le programme marchait très bien sans aucune erreur comme dans l'exemple plus haut et les décalages ainsi que les suppressions après fonctionnaient, et de plus la compilation avec le drapeau -Wall ne sortait aucune erreur ni avertissement, quand à Valgrind ce n'était pas pareil. Il n'y avait pas de fuite mémoire, mais l'erreur la plus importante et la plus fréquente à laquelle on a été confrontées est liée au fait que quelque part dans le programme nous avons une condition qui fait un test avec une variable non initialisée ("conditional jump or move depends on uninitialized value").

L'action qui nous a considérablement aidé était de diviser le code en plusieurs petites fonctions et de relancer valgrind, mais avec des commandes supplémentaires comme —track-origin=yes qui permet de détecter l'origine de variables non initialisées, grâce à quoi nous avons pu savoir l'endroit exact où des problèmes ont eu lieu et corriger toutes les erreurs.

Fait amusant : la compilation et l'exécution parfaites ne garantissent pas que valgrind ne va pas vous afficher "More than 10 000 000 total errors detected. I'm not reporting any more. Final error counts will be inaccurate. Go fix your program!"

Concernant l'interface graphique, ici aussi, nous avons rencontré des difficultés. Après avoir fini de travailler sur le fonctionnement, nous nous sommes renseignés sur les 3 bibliothèques proposées dans le sujet pour faire une interface graphique. Au début, nous avons choisi d'utiliser SDL, puisque cette bibliothèque permet de dessiner des vraies formes, d'agrandir la taille de police, mettre des images, et même ajouter une musique de fond (nous avons voulu mettre une vraie musique de tetris). Malheureusement, comme personne de nous deux travaille avec le système d'exploitation Linux (Zoe - Windows et Ekaterina - MacOs) nous avons eu du mal à connecter cette librairie à notre projet sur Visual Studio Code. Nous avons regardé une énorme quantité de tutoriels et demandé de l'aide à nos camarades de groupe, mais à chaque fois, il arrive un moment où quelque chose nous bloque et nous n'arrivons pas à passer d'une étape à l'autre.

Quant à Ncurses, nous avons réussi à l'installer et à l'utiliser dans notre projet assez simplement sur MacOs et un petit peu plus difficilement sur Windows, mais au final nous avons pu l'exécuter sur ce système d'exploitation aussi. Pour compenser au maximum au meilleur l'affichage que nous aurions pu faire avec SDL, nous avons essayé d'ajouter beaucoup de petits détails pour améliorer l'expérience de l'utilisateur : nous avons ajouté des animations de suppressions et des ajouts des élément à droite et à gauche, redimensionnement de l'écran, utiliser des vraies formes avec les caractères spéciaux, etc.

Néanmoins, à part le problème avec la fonction décrite au début de cette partie, nous sommes quand même parvenus à corriger toutes les erreurs qui ont eu lieu lors de l'exécution jusqu'à présent.

Concernant les améliorations envisageables, la plus évidente sera la résolution du bug de la fonction suppressionOccurenceFinListesSec, à part ceci nous aurons pu également faire des animations de décalages, mais nous n'avons pas eu assez de temps pour les mettre en place.

La répartition du travail

Afin de mettre en pratique notre jeu, rendre son code clair et préparer un bon rapport sur ce projet, premièrement nous nous sommes mis d'accord sur le moyen dont nous pouvons stocker et gérer les figures, ainsi que sur la façon dont le jeu va apparaître à l'utilisateur.

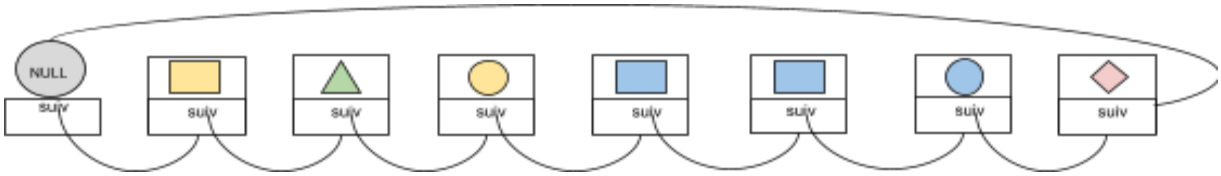
D'abord l'une des deux (Ekaterina) s'est occupée de la partie d'écriture des structures des deux types de listes, ainsi que des fonctions pour assurer l'ajout des éléments du côté désiré, la gestion des décalages et la suppression des éléments dès qu'une occurrence apparaît (selon les différents contextes), ainsi que le système de score et des petits animations, et l'autre (Zoe) de la description du code (commentaires explicatifs dans le fichier du code), écriture des fonctions pour assurer la gestion du fichier de classement, la possibilité de sauvegarder plusieurs parties et reprendre la partie désirée ou bien commencer une nouvelle, ainsi que de l'interface graphique en utilisant la bibliothèque ncurses.

Pour ne pas se perdre dans le code de chacune, après chaque petite fonction ou bout de script fonctionnel, nous nous expliquions ce que nous avions fait et discutions à propos des améliorations possibles.

Concernant les deux fichiers explicatifs, le README et dev.pdf sont préparés par nous deux.

Annexe (Schéma représentant la forme des listes)

Si notre plateau est sous cette forme ■▲●■■●◆, alors notre liste principale est le suivant :
Attention : C'est juste un exemple de stockage des données, au cours du jeu il y aurait eu une suppression d'une occurrence au milieu.



Et les listes secondaires de figures et de couleurs contenues dans le tableau *tab* sont: