

TD/TME Semaine 4 Othello le retour

Version du 17 novembre 2019

Objectif(s)

- ★ Première manipulation d'un arbre ;
- ★ Consolidation des pointeurs sur fonction ;
- ★ Consolider l'utilisation des fichiers ;
- ★ Réaliser un mini projet complet et ludique.

Exercice(s)

Othello, le retour !

Ce sujet fait suite à un mini-projet proposé dans l'UE de L1, il introduit deux fonctionnalités majeures : un joueur automatique beaucoup plus puissant et la possibilité de réaliser des parties entre joueurs distants (via le réseau).

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 1. 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).

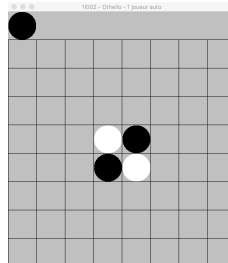


FIGURE 1 – Plateau de jeu de Othello au départ

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.

Par exemple (cf. figure 2), 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 cinq fichiers .c et quatre fichiers .h :

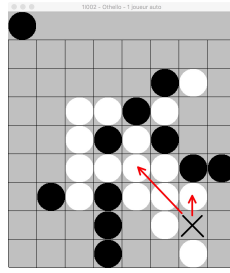


FIGURE 2 – Plateau de jeu de Othello en cours de jeu

- `Affichage.c` et `Affichage.h` qui contiennent toutes les fonctions permettant l’affichage graphique du plateau de jeu. Toutes ces fonctions vous sont fournies.
- `ListePos.c` et `ListePos.h` qui contiendront toutes les fonctions de manipulation de listes de positions jouables.
- `Othello.c` et `Othello.h` qui contiendront toutes les fonctions concernant le jeu lui même.
- `ArbreMiMa.c` et `ArbreMiMa.h` qui contiendront toutes les fonctions concernant la construction de l’arbre de jeu ainsi que son exploitation par l’algorithme de jeu automatique *Min-Max*.
- `Main.c` qui contiendra la fonction `main` constituée de la boucle principale de jeu.

Ces fichiers sont à récupérer dans le répertoire `Semaine4`.

Le plateau de jeu est un tableau à deux dimensions de taille $H \times H$, 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 1 (base) – Structures de données

Notre programme va utiliser un arbre de jeu dont l’objectif va être de représenter l’ensemble des configurations de jeu possibles à partir d’un plateau donné. Le plateau comporte 8×8 cases, initialement 4 pions l’occupent, il reste donc 60 cases vides. l’arbre de jeu complet pourrait donc avoir une profondeur de cet ordre de grandeur, ce qui est tout à fait impossible à stocker en mémoire. Nous allons donc toujours restreindre la profondeur de cet arbre et donc en permanence travailler avec un arbre partiel.

Les nœuds de l’arbre MinMax seront représentés ainsi :

```
typedef struct _posJouables_t
{
    int i;
    int j;
    struct _posJouables_t *suiv;
    struct _NdMiMa_t *Nd;
} PosJouable_t;

typedef struct _NdMiMa_t
{
    int Couleur; // Blanc ou noir joue
    struct _posJouables_t *liste_pos;
    struct _NdMiMa_t *JoueurBloque;
} NdMiMa_t;
```

Chaque nœud de l'arbre correspond à un coup pour une couleur donnée, il lui est associée une couleur ainsi qu'une liste de positions jouables. Si un joueur ne peut placer aucun pion, il passe son tour et donne la main à l'autre joueur. Le pointeur `JoueurBloque` est utilisé dans cet unique cas. À chaque élément de la liste `liste_pos` correspond un case libre où le joueur peut poser un pion, on y trouve ses coordonnées (i, j) et un pointeur correspondant à l'arbre de jeu possible pour l'adversaire consécutivement à ce coup.

1. Dessinez l'arbre de jeu initial pour une profondeur 2 sachant que c'est le NOIR qui commence.
2. Estimez la quantité totale de mémoire qui serait nécessaire pour représenter l'intégralité de l'arbre de jeu en partant des quatre pions initiaux et en faisant l'hypothèse qu'à chaque coup un joueur dispose de 2 positions jouables.

Exercice 2 (*obligatoire*) – Cases jouables et main

1. La fonction `int Est_jouable_gain(int plateau[H][H], int iLigne, int iCol, int couleurQuiJoue)` vous est fournie, elle vous retourne un entier que vous pouvez interpréter comme un booléen indiquant si vous avez le droit de jouer un pion de la couleur `couleurQuiJoue` dans la case de coordonnées $[iLigne, iCol]$. Écrivez la fonction `PosJouable_t *Trouver_liste_pos_jouables(int plateau[H][H], int couleurQuiJoue)` qui parcourt le plateau et retourne la liste des cases jouables pour le joueur `couleurQuiJoue`. Vous pourrez utiliser la fonction `PosJouable_t *Insérer(PosJouable_t *liste_pos, PosJouable_t *pos)` qui est également fournie et qui permet d'ajouter un élément à la liste des positions jouables.
2. Nous aurons également besoin d'une fonction permettant de déterminer si une case de coordonnées $[i, j]$ est présente dans cette liste. Écrivez la fonction `int Est_dans_liste(PosJouable_t *liste_pos, int i, int j)` qui retourne 1 si la case est présente et 0 sinon.

Une version assez complète de la fonction `main` vous est fournie tout comme une petite bibliothèque d'affichage du jeu. Le programme utilise les arguments de la ligne de commande pour définir son comportement. Le premier argument correspond au mode, cinq valeurs sont possibles :

0. Jeu à deux joueurs interactifs :

```
Othello 0
```

1. Jeu à un joueur interactif NOIR contre un joueur automatique BLANC, le niveau du joueur automatique (valeur numérique > 0) doit être indiqué en argument et correspond au nombre de coups anticipés :

```
Othello 1 Niveau_blanc
```

2. Jeu à deux joueurs automatiques NOIR contre BLANC. le niveau respectif des deux joueurs est indiqué en argument :

```
Othello 2 Niveau_noir Niveau_blanc
```

3. Jeu à un joueur interactif de couleur `couleur` contre un joueur distant. La synchronisation entre les instances du programme se fait par l'intermédiaire de fichiers dont les chemins sont fournis en argument :

```
Othello 3 Couleur_Locale Fichier_Local Fichier_Distant
```

En supposant que les deux *Othello* soient lancés depuis le même dossier et que la première instance joue pour le NOIR, les deux programmes seront lancés comme suit :

```
Othello 3 noir noir.txt blanc.txt&
Othello 3 blanc blanc.txt noir.txt
```

L'intérêt de cette option n'est pas de lancer les deux instances d'*Othello* sur la même machine mais de permettre la confrontations de deux joueurs utilisant des ordinateurs différents. Les deux machines devront donc nécessairement partager un même système de fichier (NFS). Cette solution utilisant des fichiers pour permettre la communication entre deux programmes a pour principal intérêt d'être très simple à mettre en œuvre

4. Jeu à un joueur automatique local de couleur `couleur` contre un joueur distant :

```
Othello 4 Couleur_Locale Fichier_Local Fichier_Distant Niveau_Local
```

Cette option va vous permettre de confronter vos joueurs automatiques !!

Dans la fonction `main` se trouve une première partie qui gère les arguments de la ligne de commande et utilise ces arguments pour initialiser la série des variables suivantes :

```
int joueurLocal = NOIR;
char *fichierDistant = NULL;
char *fichierLocal = NULL;
int Niveau[2]; /*Niveau[0] correspond au niveau du joueur NOIR*/
```

Passé ces initialisations la fonction contient essentiellement la boucle de jeu qui va contenir trois alternatives :

```
while (Partie_terminee(plateau) == 0)
{
    if ((listeJouable = Trouver_liste_pos_jouables(plateau, joueurCourant)))
    {
        /*Joueur interactif*/
        if (Mode==0 || ( Mode == 1 && joueurCourant==NOIR) || ( Mode == 3 && joueurCourant==joueurLocal))
        {
            do
            {
                if (Loop_until_play(plateau, &i, &j, joueurCourant) == -1)
                {
                    /*Le joueur a ferme la fenetre*/
                    if (Mode == 3 || Mode == 4) remove(fichierLocal);
                    listeJouable = Detruire_liste(listeJouable);
                    Detruire_fenetre();
                    return 0;
                }
                while (!Est_dans_liste(listeJouable, i , j));
            }
            /*Joueur distant*/
            else if ((Mode == 3 || Mode == 4) && joueurCourant != joueurLocal)
            {
                FILE *fdistant = NULL;
                /*Test si fichier distant existe*/

                /*Notification de lecture*/
                remove(fichierLocal);
            }
            /*Joueur automatique*/
            else
            {
                sleep(1);
                NdMiMa_t *arbreMiMa = Construire_arbre(plateau, Niveau[joueurCourant -1], joueurCourant);
                MeilleurPos(arbreMiMa, plateau, EvaluerPlateau[joueurCourant -1], &i, &j);
                arbreMiMa = Detruire_arbre(arbreMiMa);
            }
        }
    }
}
```

A l'issue de ces trois alternatives les coordonnées $[i, j]$ de la case jouée doivent avoir été mises à jour.

La synchronisation avec un joueur distant est réalisée par l'intermédiaire de 2 fichiers texte contenant sur une ligne et séparés par une virgule les coordonnées de la case jouée. Si le joueur NOIR correspondant au joueur local décide de jouer la case $[5, 4]$ il va créer le fichier `fichierLocal` et y écrire une ligne contenant 5,4. Ainsi le joueur BLANC (distant) qui a accès en lecture au fichier (qui pour lui correspond à `fichierDistant`) va attendre la création de ce fichier par le joueur NOIR. Une fois le fichier créé il va y lire les coordonnées de la case jouée par le joueur NOIR.

- Écrivez la séquence de code correspondant à l'attente de création du fichier distant suivi de la lecture des coordonnées jouées. Attention une fois le fichier créé, l'écriture des coordonnées dans ce fichier peut prendre un peu de temps, il est donc indispensable de tester la validité des coordonnées lues.

Pour notifier au joueur distant que son jeu a bien été pris en compte le joueur local supprime son fichier `remove(fichierLocal)`; il va donc pouvoir se mettre en attente du joueur NOIR (local).

Une fois les coordonnées de la case jouée connues il ne reste plus qu'à mettre à jour le plateau et passer au joueur suivant. Pour le joueur local cela correspond à la séquence de code suivante :

```
Jouer_pion(plateau, i, j, joueurCourant);
Dessine_plateau_graph(plateau, joueurCourant);
listeJouable = Detruire_liste(listeJouable);
joueurCourant = Autre_joueur(joueurCourant);
/*Fin de la boucle de jeu*/
```

4. Dans le cas de l'existence d'un joueur distant il sera également nécessaire de lui transmettre les coordonnées de la case jouée. Écrivez la séquence de code correspondante, c'est à dire écriture des coordonnées dans le fichier local puis attente de prise en compte par le joueur distant. Il s'agit de la séquence d'opérations symétrique de celle présentée dans la question précédente.

Exercice 3 (*obligatoire*) – Construction de l'arbre de jeu

Dans cet exercice nous allons nous intéresser à la construction de l'arbre de jeu nécessaire à la réalisation d'un joueur automatique. Évidemment la construction de l'arbre va devoir se limiter à une profondeur maximum donnée.

1. Écrivez la fonction :

```
NdMiMa_t *Construire_arbre(int plateau[H][H], int prof, int couleurQuiJoue);
```

Cette fonction très intrinsèquement récursive pourra utiliser les fonctions `Partie_terminee` pour identifier un cas de base et la fonction `Trouver_liste_pos_jouables` pour obtenir la liste des cases jouables.

Pensez pour chaque case jouable à mettre à jours le plateau ou sa copie avant d'appeler la création du sous arbre correspondant. Vous pourrez utiliser pour cela la fonction `Jouer_pion`. Voici pour rappel les prototypes de ces fonctions :

```
int Partie_terminee(int plateau[H][H])
PosJouable_t *Trouver_liste_pos_jouables(int plateau[H][H], int couleurQuiJoue)
void Copier_plateau(int dst[H][H], int src[H][H])
void Jouer_pion(int plateau[H][H], int iLigne, int iCol, int couleurQuiJoue)
```

Exercice 4 (*obligatoire*) – Joueur automatique

Pour réaliser notre joueur automatique nous allons utiliser l'algorithme **MinMax** très classique dans la théorie des jeux et qui s'applique parfaitement à l'Othello. Cet algorithme consiste à minimiser les pertes dans le pire des cas et il présuppose que l'adversaire cherche à jouer de manière à maximiser ses gains.

Appliqué à un arbre de jeu de profondeur 1 l'algorithme se bornera à sélectionner le coup qui maximisera nos gains (exclusivement pour ce coup). Pour une profondeur 2 l'algorithme va sélectionner la case qui minimisera les pertes quelle que soit la case jouée par l'adversaire au coup suivant. Ainsi l'algorithme va alterner la recherche du gain maximum pour les niveaux de l'arbre correspondant à un coup du joueur et à minimiser cette valeur pour les autres car ce gain minimum correspond au gain maximum pour son adversaire.

Par convention les gains seront comptabilisés positivement pour le joueur NOIR et négativement pour le BLANC.

1. Un ensemble de fonctions pourront être utilisées pour évaluer un plateau, une version intuitive consiste à faire la différence du nombre de pions noirs et le nombre de pions blancs. Cette fonction d'évaluation est connue pour être particulièrement inefficace mais nous vous proposons de l'utiliser pour une première implémentation.

Écrivez cette fonction :

```
int EvaluerPlateau_0(int plateau[H][H])
```

L'implémentation de l'algorithme **MinMax** va reposer sur deux fonctions. Une première fonction (`MinMax`) va permettre d'évaluer l'arbre. La seconde (`MeilleurPos`) va déterminer la case à jouer, celle pour laquelle la fonction `MinMax` aura retourné sa valeur maximum dans le cas du joueur NOIR et minimum dans le cas du BLANC. La fonction `EvaluerPlateau` ne sera appelée qu'au niveau des feuilles de l'arbre et le parcours en profondeur de l'arbre consistera en une alternance de recherche du maximum et du minimum parmi un ensemble de valeurs entières.

2. Écrivez la fonction :

```
int MinMax(NdMiMa_t *arbre, int plateau[H][H], int (*EvaluerPlateau)(int plateau[H][H]))
```

Transmettre la fonction d'évaluation du plateau en argument permet d'en tester différentes versions sans modifier l'implémentation de l'algorithme. Ce sera notamment très utile lors de la confrontation de deux joueurs automatiques (mode = 2) utilisant des fonctions d'évaluation différentes.

3. Pour compléter le projet il ne reste plus qu'à écrire la fonction qui détermine le meilleur coup, elle est très semblable à `MinMax` et fournit comme principal résultat les coordonnées de la case à jouer. Cette fonction va bien évidemment faire appel à `MinMax` pour explorer l'arbre de jeu.

```
int MeilleurPos(NdMiMa_t *arbre, int plateau[H][H], int (*EvaluerPlateau)(int plateau[H][H]), int *pi, int *pj)
```

4. Vous disposez maintenant d'un programme complet. N'hésitez pas à le tester, à évaluer jusqu'à quelle profondeur d'arbre vous pouvez aller, proposer des solutions classiques d'élagage et surtout expérimenter d'autres fonctions d'évaluations car toutes les cases ne sont pas du tout équivalentes.



TD/TME Semaine 5 Bibliothèque générique d'arbres

Version du 17 novembre 2019

Objectif(s)

- ★ Approfondir les arbres (binaires, génériques)
- ★ Approfondir les bibliothèques génériques (écriture, utilisation)

Exercice(s)

TD : Conception d'une bibliothèque générique d'arbres

Dans ce TD, vous allez concevoir une bibliothèque d'arbres qui permet de gérer n'importe quel type de données. Comme pour les listes, vous aurez donc à définir les structures de données, les fonctions nécessaires pour manipuler les données (qui devront être définies pour chaque nouveau type à gérer), et des fonctions génériques qui réaliseront leur traitement en s'appuyant, si besoin, sur ces fonctions spécifiques aux données.

Exercice 1 (*base*) – Arbres binaires

1. Quelles fonctions de manipulation des données stockées dans un noeud seront nécessaires ?
2. En vous inspirant de ce qui a été fait pour les listes génériques, définissez la ou les structures de données des arbres binaires génériques. Vous ajouterez un champ `copie` de type `char` à la structure d'arbre pour indiquer si les données doivent être dupliquées lors de la création d'un noeud.
3. Écrivez la fonction de création d'un arbre binaire vide.
4. Écrivez la fonction de création d'un noeud. La donnée `data` ne sera dupliquée que si le champ `copie` de l'arbre est différent de 0. Prototype :

```
PNoeudBinaire creer_noeud_binaire(PArbreBinaire pab, void *data);
```

Le noeud ne sera pas ajouté à l'arbre. Le pointeur sur l'arbre permet juste de récupérer la fonction de duplication de la donnée, si besoin.

5. Écrivez la fonction de destruction d'un arbre. Vous prendrez en compte le champ `copie` pour libérer ou non la mémoire sur laquelle pointe `data`. Vous ferez une fonction de destruction de l'arbre qui fera appel à une fonction récursive que vous écrirez également.

Prototype :

```
void detruire_ab(PArbreBinaire pab);
```

6. Écrivez la fonction d’affichage d’un arbre avec un parcours préfixe. Comme pour la question précédente, vous ferez une fonction d’affichage qui fera appel à une fonction récursive que vous écrirez également. Prototype :

```
void afficher_ab_prefixe(PArbreBinaire pab);
```

Vous pourrez éventuellement ajouter des "(" et des ")" pour indiquer la structure de l’arbre. Exemple :

```
( ( 1 ( ( 17) 22 ( 44) ) 52 ( 64) ) ) 72 ( 85 ( ( 86) 98) ) )
```

7. Que faudrait-il changer à cette fonction pour obtenir un affichage avec un parcours en infixe ? En postfixe ?

Exercice 2 (*obligatoire*) – Arbres binaires de recherche

Les fonctions écrites jusqu’à présent ne faisaient aucune hypothèse particulière sur la façon d’organiser l’arbre (mis à part le fait que c’était un arbre binaire). Vous allez maintenant écrire les fonctions permettant de gérer un arbre binaire de recherche. Pour cela il suffira d’écrire la fonction d’ajout dans l’arbre et la fonction de recherche.

1. Écrivez la fonction d’ajout dans l’arbre. Pour rappel, dans un arbre binaire de recherche, le sous-arbre gauche contient les données inférieures au noeud courant et le sous-arbre droit contient les données supérieures. Lors de l’ajout d’une donnée, il faut donc parcourir l’arbre en continuant dans le sous-arbre gauche ou droit jusqu’à atteindre la fin de l’arbre. La donnée est alors ajoutée dans une feuille créée à cette occasion. La fonction ne fait rien si la donnée est déjà dans l’arbre. Prototype :

```
void ajouter_abr(PArbreBinaire pab, void *data);
```

Comme précédemment, cette fonction fera appel à une fonction récursive que vous écrirez également.

2. Écrivez la fonction de recherche dans l’arbre. Prototype :

```
PNoeudBinaire chercher_abr(PArbreBinaire pab, void *data);
```

3. Écrivez une fonction de lecture de l’arbre depuis un fichier. Chaque élément lu sera inséré dans l’arbre avec la fonction d’ajout écrite ci-dessus. Prototype :

```
void lire_abr(PArbreBinaire pab, const char *nom_fichier);
```

L’arbre fourni en argument est un arbre vide (dont la racine vaut NULL). Il est transmis pour que la fonction de lecture de l’arbre puisse utiliser la fonction de lecture de donnée appropriée.

La fonction de lecture alloue la mémoire des données lors de la lecture. Afin d’éviter de dupliquer ces données, pendant l’ajout des éléments lus, vous prendrez soin de mettre le champ `copie` à 0. Vous mettrez ensuite ce champ à 1 pour que ces données soient bien libérées lorsque la mémoire associée à cet arbre sera libérée.

4. Écrivez une fonction d’écriture de l’arbre dans un fichier. Choisissez la méthode de parcours de l’arbre qui permettra à la fonction de lecture ci-dessus de reconstruire l’arbre à l’identique après lecture. Ce parcours sera dans une fonction récursive que vous écrirez également. Prototype :

```
void ecrire_ab(PArbreBinaire pab, const char *nom_fichier);
```

Exercice 3 (*approfondissement*) – Utilisation de la fonction map

Un certain nombre des fonctions écrites jusqu’à présent reprennent un même schéma. Pour éviter d’avoir à répéter ce code, vous allez écrire une fonction `map` pour appliquer une fonction donnée en argument à tous les éléments de l’arbre. Vous l’appliquerez ensuite pour redéfinir quelques-unes des fonctions vues jusqu’à présent.

1. Écrivez la fonction `map` permettant d’appliquer à chaque élément de l’arbre, avec un parcours préfixe, une fonction transmise en argument. Comme précédemment, il faut faire une première fonction qui appelle une fonction récursive qu’il faut aussi écrire. Prototype :

```
void map_ab_prefixe(PArbreBinaire pab, void (*fonction)(void *data, void *oa), void *optarg);
```

`optarg` est un argument permettant de transmettre des arguments supplémentaires à la fonction qui sera appelée sur chaque élément. Cela peut aussi permettre de récupérer une valeur calculée.

2. La fonction `map` que l'on a définie ci-dessus nécessite, pour plus de souplesse une fonction prenant en argument un `void *` et un argument supplémentaire `optarg`. La fonction d'affichage ne prend qu'un `void *`. Définissez une fonction ayant le prototype approprié pour redéfinir une fonction d'affichage en s'appuyant sur `map`, fonction que vous écrirez également. Astuce : transmettez un pointeur sur l'arbre via `optarg`.
3. Utilisez la fonction `map` pour réécrire la fonction d'écriture dans un fichier. Vous aurez pour cela besoin de définir une structure dédiée à la transmission des informations appropriées via l'argument `optarg` et vous aurez également à écrire la fonction qui sera transmise en argument à `map`.
4. Pouvez-vous écrire une version `map` de la fonction de destruction de l'arbre ? Si non, que faudrait-il faire ?

TME : Utilisation de la bibliothèque

Récupérez les fichiers fournis pour cette séance. Ils contiennent les implémentations des fonctions vues en TD.

Exercice 4 (*base*) – Tests de la bibliothèque d'arbres binaires

1. Écrivez une fonction `main` pour tester la bibliothèque d'arbres (fichier `ex_ab_entiers.c`). Dans un premier temps, vous la testerez avec des entiers. Vous créerez un arbre binaire de recherche dans lequel vous insèrerez une dizaine d'entiers aléatoires compris entre 0 et 99. Vous afficherez l'arbre puis vous ferez une boucle pour chercher tous les entiers compris entre 0 et 99 afin de vérifier le bon fonctionnement de votre fonction de recherche. Vous écrirez l'arbre dans un fichier, puis vous le lirez dans un nouvel arbre que vous afficherez. Vous finirez votre `main` par la libération de toute la mémoire allouée. Vous prendrez soin de tester votre exécutable avec `valgrind` pour vérifier qu'il n'y a pas de fuite mémoire.
2. Écrivez une fonction `main` pour tester la bibliothèque d'arbres avec des mots (fichier `ex_ab_mots.c`). Vous créerez un arbre binaire de recherche dans lequel vous insèrerez les mots lus depuis le dictionnaire contenu dans le fichier `"french_za_reordered"`. Vous rechercherez quelques mots qui sont ou ne sont pas dans le dictionnaire pour vérifier le bon fonctionnement de la bibliothèque. Vous pourrez également l'écrire dans un autre fichier et vérifier que le contenu est le même. Choisissez la fonction d'écriture permettant de récupérer le fichier trié par ordre alphabétique. Pour la comparaison, vous pourrez utiliser la commande shell `diff` qui permet de comparer deux fichiers et la commande shell `sort` qui permet de trier un fichier (pour générer une version triée de `"french_za_reordered"`). Vous prendrez soin de tester votre exécutable avec `valgrind` pour vérifier qu'il n'y a pas de fuite mémoire.

Exercice 5 (*obligatoire*) – Arbres d'expressions

Vous allez maintenant utiliser la bibliothèque générique d'arbres pour implémenter des arbres d'expression. Un arbre d'expression permet de représenter une expression arithmétique. Les noeuds internes de l'arbre correspondent à des opérateurs, qui ne seront ici que des opérateurs binaires (ce qui permettra de représenter ces expressions avec des arbres binaires). Les feuilles de l'arbres seront soit des constantes, soit des variables. L'évaluation d'un noeud de l'arbre est sa valeur, s'il s'agit d'une constante ou d'une variable, ou le résultat de l'opération portée par le noeud et appliquée aux opérandes correspondant à leurs sous-arbres gauche et droit. L'évaluation de l'arbre est le résultat de l'évaluation de sa racine. Remarque : l'évaluation de l'arbre nécessite de demander la valeur des variables. La figure 1 présente un exemple d'arbre d'expression. Il représente l'expression arithmétique : $(3 * X) + (Y/6)$.

Un fichier contenant une fonction `main` vous est fourni (fichier `main_expr.c`). Vous pourrez l'utiliser en le modifiant à votre guise pour tester vos fonctions au fur et à mesure.

Les noeuds pourront être de 3 types : constante, variable ou opérateur. Un type énuméré permettra d'identifier le type d'un noeud :

```
typedef enum _type_noeud {VAR, CST, OP} TypeNoeud;
```

La donnée portée par un noeud associera à ce type, les champs nécessaires pour gérer les 3 cas de figure :

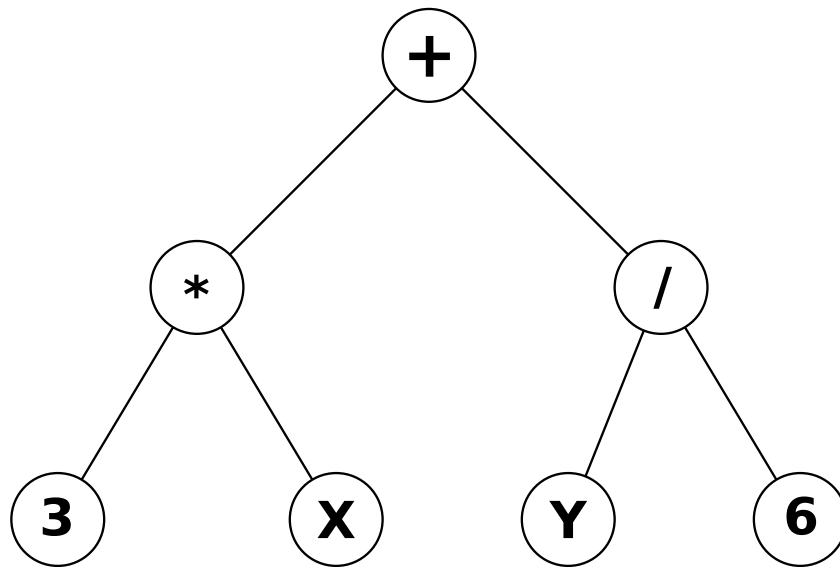


FIGURE 1 – Exemple d'arbre d'expression.

```

typedef struct _dataExpr
{
    TypeNoeud type_noeud;
    unsigned int indice_var;
    char operateur;
    float constante;
} DataExpr;

```

Nous ne considérerons que les variables représentées par une lettre majuscule. Le champ associé à une variable est l'indice de cette variable : 0 correspond à la variable A, 1 à B, ... Un opérateur est représenté par un caractère. Nous ne considérerons que les 4 opérateurs suivants : addition (représentée par un '+'), soustraction ('-'), multiplication ('*') et division ('/').

1. En vue de leur utilisation dans la bibliothèque d'arbres génériques, écrivez les fonctions permettant de dupliquer, copier et détruire de telles données (fichier `arbre_expr.c`). Prototypes :

```

void *dupliquer_expr(const void *src);
void copier_expr(const void *src, void *dst);
void detruire_expr(void *data);

```

Remarque : nous n'utiliserons pas la fonction de comparaison, il suffit donc de définir une fonction renvoyant, par exemple, toujours 0.

2. Écrivez la fonction d'écriture d'une donnée. Cette fonction écrira, selon le type de la donnée, soit la variable ('A' si l'indice est 0, 'B' si c'est 1, ...), soit la constante, soit le caractère représentant l'opération. Un espace sera ajouté avant et après cet affichage. La fonction d'affichage utilisera le même format. Vous pourrez donc vous contenter de la définir à partir de la fonction d'écriture à qui vous transmettez `stdout`, qui correspond à la sortie standard. Prototypes :

```

int ecrire_expr(const void *data, FILE *f);

void afficher_expr(const void *data);

```

3. La fonction de lecture d'une donnée nécessite de gérer les 3 différents cas de figure. Plusieurs implémentations sont possibles. L'une d'entre elle consiste à mettre dans un buffer la prochaine donnée à analyser. Lorsque les premiers espaces éventuels ('espace' au sens large de `isspace`, tapez `man 3 isspace` dans un terminal pour en savoir plus) ont été enlevés, il s'agit des caractères avant le prochain 'espace'. Vous pouvez ensuite analyser les premiers caractères pour distinguer les 3 cas de figures et initialiser la donnée lue en fonction de cela. Vous pourrez utiliser les fonctions de test de caractères `isdigit` et `isupper`.

Prototype :

```

void *lire_expr(FILE *);

```

4. La lecture de l'arbre ne peut pas s'appuyer sur la bibliothèque générique, car la fonction vue précédemment est dédiée aux arbres binaires de recherche. La lecture de l'arbre d'expression nécessite la création d'une fonction récursive qui va lire noeud après noeud. La fonction principale de lecture se contentera de créer l'arbre et de faire appel à la fonction récursive. La fonction récursive lira la donnée d'un noeud à l'aide de la fonction vue précédemment et, si le noeud est un opérateur, fera deux appels récurrents à elle-même pour lire les sous-arbres gauche et droit. Comme précédemment, vous penserez à mettre le champ `copie` à 0 pendant la lecture et à 1 ensuite pour que les données soient bien effacées lorsque l'arbre sera détruit.

Prototype :

```
PNoeudBinaire construire_arbre_expr_rec(FILE *f, PArbreBinaire pab);  
PArbreBinaire construire_arbre_expr(FILE *f);
```

5. Écrivez la fonction d'évaluation de l'arbre. Cette fonction reçoit en argument l'arbre d'expression ainsi que le tableau des valeurs des variables contenues dans l'arbre.

Prototype :

```
float evaluer(PNoeudBinaire pabe, float var_lue[]);
```



TD/TME Semaine 6

Compression par l'algorithme de Huffman

Version du 17 novembre 2019

Objectif(s)

- ★ Utiliser les bibliothèques génériques de listes et d'arbres
- ★ Combiner les arbres et les listes

Exercice(s)

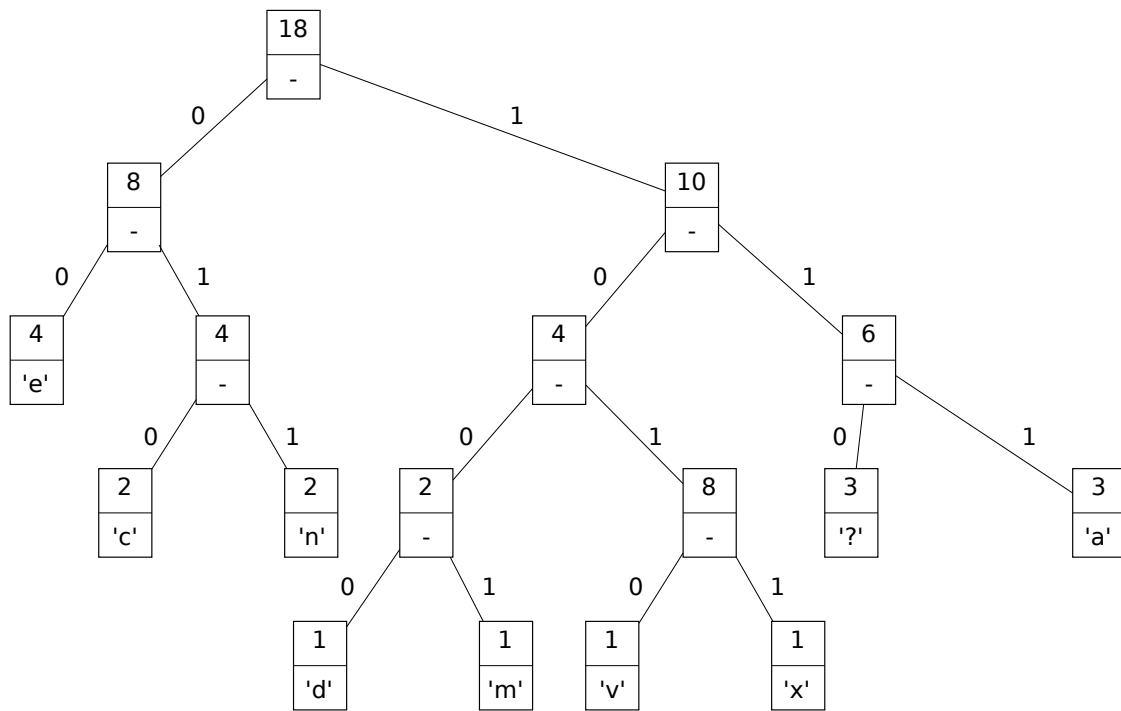
Le codage de Huffman est un algorithme de compression qui fut mis au point en 1952 par David Huffman. C'est une compression de type statistique qui utilise un arbre pour coder des octets. Les codes étant de taille variable, les octets revenant le plus fréquemment sont codés avec un code beaucoup plus court (nombre de bits inférieur à 8) que ceux qui sont moins fréquents. C'est ce qui permet de compresser les données sans perte. Cet algorithme offre des taux de compression assez bons et est utilisé dans de nombreux formats de compression (MP3, bzip2 etc.)

Le principe du codage de Huffman repose sur la création d'un arbre binaire de codage qui va permettre d'attribuer des codes les plus courts possibles aux caractères présents dans le fichier à coder et à ceux-là uniquement. Supposons que la phrase à coder soit "examen ? de ? c ? avance". On recherche tout d'abord le nombre d'occurrences de chaque caractère, soit pour notre exemple la distribution suivante :

?	a	c	d	e	m	n	v	x
3	3	2	1	4	1	2	1	1

A partir de ces informations, on va construire un arbre binaire dont les feuilles représenteront un caractère particulier. Les arêtes seront étiquetées 0 ou 1 et le chemin à parcourir de la racine pour atteindre une feuille (un caractère) indique le code à utiliser pour représenter celui-ci.

A chaque nœud est associé un poids. Cette valeur correspond au nombre d'occurrence du caractère dans le cas d'une feuille et à la somme des poids des nœuds fils du sous-arbre pour les nœuds "non feuille". Plus le poids d'un nœud est élevé, plus il doit être près de la racine, ainsi les codes correspondants aux caractères les plus fréquents seront les plus courts. Ces poids interviennent uniquement dans le processus de construction de l'arbre. Pour notre exemple, voici un bon arbre de codage :



On peut déduire de cet arbre les codes correspondant aux caractères présents :

?	a	c	d	e	m	n	v	x
110	111	010	1000	00	1001	011	1010	1011

Notre phrase contenant 18 caractères ($18 \times 8 = 144$ bits) peut être codée sur 54 bits (sans prendre en compte ceux nécessaires au stockage de l'arbre) comme suit :

e x a m e n ? d e ? c ? a v a n c e
 00 1011 111 1001 00 011 110 1000 00 110 010 110 111 1010 111 011 010 00

TD : construction de l'arbre et compression

Exercice 1 (base)

- La construction de l'arbre se fait en observant la fréquence de chaque symbole afin de construire le code le plus compact possible pour les symboles les plus présents. Écrivez la fonction :

```
int *distri_code(FILE *f);
```

Cette fonction doit parcourir le fichier `f` fourni en argument et retourner un pointeur sur un tableau (qu'elle aura alloué) de 256 entiers. La case `i` du tableau devra contenir le nombre d'occurrences de l'octet de valeur `i` dans le fichier. Par exemple, dans le cas d'un fichier texte, si le fichier contient 5 'a', la case 97 (code ASCII de 'a') devra contenir 5.

La construction de l'arbre consiste à créer un arbre de Huffman par symbole, puis à regrouper les arbres correspondant au nombre d'occurrence le plus faible. Ce regroupement crée un nouvel arbre, dont la racine ne porte pas de symbole mais un nombre d'occurrences égal à la somme des nombres d'occurrences des symboles ainsi regroupés. Ce processus de regroupement des deux arbres ayant le plus faible nombre d'occurrence se poursuit ensuite jusqu'à ce qu'il ne reste plus qu'un seul arbre. Pour implémenter ce processus, nous allons nous appuyer sur une liste chaînée d'arbres de Huffman. Cette liste chaînée sera triée par ordre croissant des nombre d'occurrences portés par la racine des arbres. Une étape du processus consistera donc à extraire les deux éléments en tête de liste et à les réinsérer à leur place.

Dans la suite, vous allez donc manipuler des arbres. Vous utiliserez la bibliothèque générique implémentée précédemment. La donnée sera de type lettre-fréquence avec la structure suivante :

```
typedef struct _LettreFrequence {
    char lettre;
    unsigned int nb_occ;
} LettreFrequence;
```

Les fonctions de manipulation de cette structure de données seront à implémenter en TME. Leur prototype est le suivant :

```
void *dupliquer_lf(const void *src);
void copier_lf(const void *src, void *dst);
void detruire_lf(void *data);
void afficher_lf(const void *data);
int comparer_lf(const void *a, const void *b);
int ecrire_lf(const void *data, FILE *f);
void * lire_lf(FILE *);
```

La construction de l'arbre nécessite de manipuler une liste d'arbres. Vous utiliserez donc la bibliothèque générique de listes vue précédemment avec pour donnée un arbre de Huffman. Les fonctions de manipulation de l'arbre nécessaires à la définition de la liste sont les suivantes :

```
void *dupliquer_ah(const void *src);
void copier_ah(const void *src, void *dst);
void detruire_ah(void *data);
void afficher_ah(const void *data);
int comparer_ah(const void *a, const void *b);
int ecrire_ah(const void *data, FILE *f);
void * lire_ah(FILE *);
```

Elles seront implémentées en TME.

- Écrivez la fonction permettant de construire la liste d'arbres de Huffman constitués d'un seul noeud à partir du tableau d'occurrences construit à la question précédente. Vous pourrez créer une fonction de création d'un arbre de Huffman ne contenant que la racine avec la lettre et le nombre d'occurrences passé par argument. Prototypes :

```
PArbreBinaire creer_arbre_huffman(char lettre, unsigned int nb_occ);
```

```
PListe creer_liste_occ(int *);
```

- Écrivez les fonctions permettant de construire l'arbre de Huffman à partir de la liste des arbres ne contenant qu'un seul noeud. Prototypes :

```
PArbreBinaire fusion_arbres(PArbreBinaire pab1, PArbreBinaire pab2);
```

```
PArbreBinaire construire_arbre_codes(PListe plocc);
```

La première fonction fusionne deux arbres en créant un nouveau noeud avec, comme nombre d'occurrences la sommes des nombres d'occurrences portés par les racines des deux arbres et en les insérant comme sous-arbres gauche et droit de cette racine.

Attention à la gestion de la mémoire : assurez-vous de libérer toute la mémoire qui ne sera plus nécessaire.

- Une fois que l'arbre est construit, il faut le parcourir pour construire le tableau associant un symbole à son code. Peut-on utiliser les fonctions de parcours vues précédemment ? Si non, proposez une nouvelle fonction de parcours générique. Nous utiliserons en priorité un parcours préfixe. Vous pourrez écrire une fonction d'affichage du code associé à un symbole dans un arbre de Huffman.

5. La construction du tableau des codes s'appuie sur un parcours de l'arbre. L'argument optionnel sera utilisé pour transmettre le code en cours de construction ainsi que le tableau des codes (qui sera le résultat de ce processus). La structure de donnée utilisée pour transmettre cette information sera la suivante :

```
typedef struct _TabCode {
    char code_courant[257];
    char *code[256];
} TabCode;
```

Écrivez la fonction qui sera transmise à la fonction de parcours. Prototype :

```
void construire_tableau_code_fonction(void *data, int prof, TypeChemin tc,
                                     TypeNoeudArbreBinaire tnab, void *oa);
```

Écrivez la fonction de construction du tableau de codes (qui utilisera le parcours de l'arbre avec la fonction ci-dessus). Prototype :

```
char **construire_tableau_code(PArbreBinaire pah);
```

À ce stade, un code sera représenté sous la forme d'une chaîne de caractère ne contenant que des '0' et des '1' (le code associé à 'v' d'après l'arbre indiqué précédemment sera donc "1010"). Cette chaîne sera transformée en binaire au moment de l'écriture dans le fichier compressé.

6. La compression consiste à parcourir le fichier à compresser une première fois pour construire les codes les plus courts possible compte tenu des données à compresser¹. Vous vous appuyerez pour cela sur les fonctions vues précédemment. Au passage, la fonction comptera le nombre de symboles contenus dans le fichier.

La compression consistera ensuite à écrire l'arbre de Huffman au début du fichier compressé, le nombre de symboles contenus dans le fichier puis les codes associés à chaque symbole du fichier source. La difficulté est que ces codes sont des chaînes de bits qui seront souvent plus courtes qu'un octet. Il est impératif de ne pas se réaligner sur les octets sinon le fichier compressé prendra plus d'espace que la source !

Le principe de la compression consiste donc à remplir un buffer et à écrire ce buffer (ou une partie de ce buffer) dans le fichier de destination dès lors qu'il est suffisamment gros (pour limiter les accès disque). Il ne faudra pas oublier à la fin d'écrire les derniers bits. Comme il faut écrire un nombre entier d'octets, il faudra peut-être écrire quelques bits qui ne font pas partie du fichier source, c'est la raison pour laquelle nous avons écrit le nombre de symboles à lire : lors de la décompression, ces bits supplémentaires pourront donc être ignorés.

Pour gérer l'écriture bit à bit dans un buffer, vous vous appuyerez sur la fonction `ajouter_bits`. Cette fonction ajoutera un code donné sous forme d'une chaîne de caractère à un buffer (supposé de taille suffisante) à partir de la position `ptr` (position en bits, et non en octets). La fonction renvoie la position du prochain bit à écrire. Prototype de la fonction :

```
int ajouter_bits(char *code, char* buffer, int ptr);
```

Vous n'aurez pas à écrire cette fonction. Vous prendrez soin de fermer les fichiers ouverts et de libérer la mémoire qui ne sera plus utilisée à la fin de la fonction.

TME : fonctions de manipulation des données et décompression

Exercice 2 (base)

1. Cette version a l'inconvénient de devoir inclure l'arbre de Huffman dans le fichier compressé, ce qui occupe de l'espace. Il existe des variantes qui, soit s'appuient sur un arbre fixe, soit le construisent à la volée. Nous ne les verrons pas ici.

1. Écrivez les fonctions de manipulation de la structure lettre-fréquence, donnée de l'arbre de Huffman (fichier `arbre_freq.c`).

Prototypes :

```
void *dupliquer_lf(const void *src);
void copier_lf(const void *src, void *dst);
void detruire_lf(void *data);
void afficher_lf(const void *data);
int comparer_lf(const void *a, const void *b);
int ecrire_lf(const void *data, FILE *f);
void *lire_lf(FILE *);
```

La comparaison de deux variables de type `LettreFrequence` ne portera que sur le nombre d'occurrences. La fonction d'écriture écrira les deux champs à la suite (par exemple "a 3") que la fonction de lecture permettra de lire.

2. La construction de l'arbre s'appuie sur une liste chaînée d'arbres de Huffman. Écrivez les fonctions de manipulation de ces arbres, fonctions qui seront utilisées par notre bibliothèque de liste chaînée.

Prototypes :

```
void *dupliquer_ah(const void *src);
void copier_ah(const void *src, void *dst);
void detruire_ah(void *data);
void afficher_ah(const void *data);
int comparer_ah(const void *a, const void *b);
```

La fonction de copie s'appuiera sur une fonction récurrente qui copiera noeud par noeud. Prototype :

```
PNoeudBinaire copier_ab(PArbreBinaire pab, PNoeudBinaire pnb);
```

Les fonctions de destruction et d'affichage pourront s'appuyer sur les fonctions de la bibliothèque générique d'arbres. La comparaison portera sur le nombre d'occurrences de la racine uniquement (en utilisant la fonction de comparaison définie précédemment).

3. L'écriture de l'arbre doit être la plus compacte possible. L'arbre sera en effet écrit avec le fichier compressé pour pouvoir décompresser son contenu. Le nombre d'occurrence ne sert qu'à construire l'arbre. Pour cette raison, il ne sera pas sauvegardé. L'arbre sera écrit avec un parcours préfixe, en écrivant un 0 sur un octet pour les noeuds internes et en écrivant la valeur du symbole pour les feuilles (sur un octet également).

Prototypes :

```
int ecrire_ah(const void *data, FILE *f);
void *lire_ah(FILE *);
```

Vous pourrez vous appuyer sur des fonctions récurrentes. Prototypes :

```
PNoeudBinaire lire_ah_rec(FILE *f);
```

```
void ecrire_noeud_huffman(void *data, int prof, TypeChemin tc, TypeNoeudArbreBinaire ttab, void *oa);
```

4. Quelle est la limite de cette fonction d'écriture ? Que faudrait-il faire pour l'éviter ?
5. Écrivez une fonction `main` dans le fichier `main_huffman.c`. Cette fonction prendra soit un, soit deux arguments. Si elle prend un seul argument, cet argument sera le nom du fichier à compresser. Si elle en prend deux, ce sera le nom du fichier à décompresser ainsi que le fichier de destination. Écrivez cette fonction, compilez votre programme et testez le pour compresser un fichier de taille conséquente (par exemple le dictionnaire utilisé dans une séance précédente). Vérifiez que la taille du fichier compressé est inférieure à la taille initiale.
6. Écrivez la fonction de décompression (fichier `huffman.c`). Prototype :

```
int decompresser(const char *nom_fichier_in, const char *nom_fichier_out);
```

Cette fonction ouvre le fichier à décompresser ainsi que le fichier de destination. Elle lit ensuite l'arbre de Huffman ainsi que le nombre de symboles à décompresser. Elle lit ensuite le fichier source octet par octet et utilise les bits de ces octets lus pour avancer dans l'arbre de Huffman jusqu'à atteindre une feuille. Lorsqu'une feuille est atteinte, le symbole correspondant est écrit dans le fichier de destination et l'interprétation des bits lus reprend en repartant de la racine de l'arbre de Huffman. Chaque fois d'un octet a été entièrement pris en compte (lorsque ses 8 bits ont été utilisés), on lit un nouvel octet. Ce processus s'arrête lorsque l'on a décompressé tous les symboles qui avaient été compressés.

Exemple : l'octet 01100001 a été lu (il correspond à l'entier 97 en base 10). Telle que la fonction `ajouter_bits` est conçue, le bit de poids faible (le 1 des unités) correspond au bit de poids fort du code, donc au premier choix à faire dans le parcours de l'arbre (ici aller dans la branche droite). Si on considère l'arbre de Huffman représenté au début de l'énoncé, le 1 envoie sur la branche droite, le 0 suivant envoie alors sur la branche gauche, puis encore 0 et donc à gauche également, 0 et donc encore à gauche. On arrive sur la feuille 'd'. Les 4 premiers bits 0001 ont donc été décodés en un 'd'. Il reste à lire les bits 0110. En procédant de la même façon, on obtient un 'n' (bits 110 interprétés comme le code "011"). Il reste le bit de poids fort de l'octet lu : le 0, qui, pour le prochain symbole à décoder, envoie sur la branche gauche de l'arbre de Huffman. On a alors épuisé les bits lus. Pour continuer le décodage de ce symbole, il faut donc lire un autre octet.

Testez votre fonction en vérifiant que le fichier décompressé est identique au fichier initial.