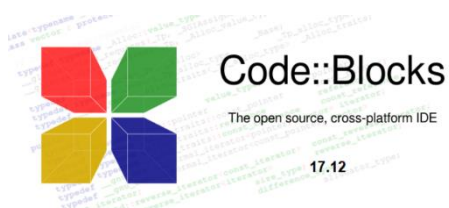
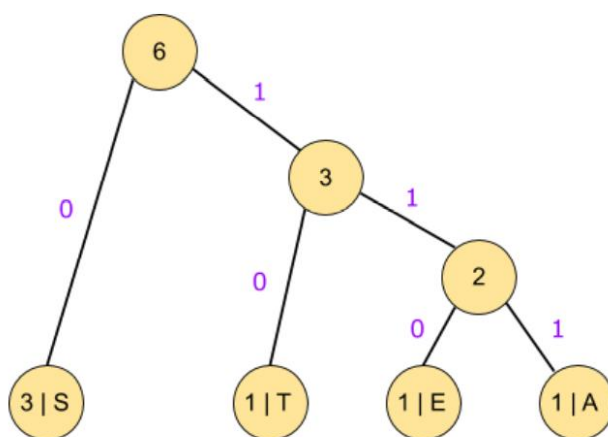


RAPPORT DE PROJET HUFFMAN CODING



Lindsay Rébeau

L2 BN

SOMMAIRE

Introduction générale	3
I - Analyse.....	4
A - Présentation du projet Huffman coding	4
B - Principe du codage de Huffman	4
II - Conception	4
A - Architecture logicielle du projet	4
B - Modélisation des structures de données utilisées	5
C - Implémentation des fonctions	7
1) Présentation des fonctions du projet	7
2) Explication des fonctions essentielles du projet.....	9
III - Organisation	25
Conclusion générale.....	26
Références bibliographiques	26
Annexe 1 : Fichiers textes utilisés	27
Annexe 2 : Répartition des tâches en lots	28
Annexe 3 : Documentation de code	30

Ce rapport de projet « Huffman coding » est réalisé dans le cadre de l'UE d'Informatique générale. Le projet est commun aux deux modules TI301 – Fondamentaux de l'algorithmique 3 et TI304 – Introduction au génie logiciel. Toutefois, ce rapport concerne la partie fondamentaux de l'algorithmique qui est axée sur l'implémentation en langage C et les structures de données.

L'ensemble des fonctionnalités demandées a été réalisé en utilisant l'IDE Code::Blocks sous Windows ainsi que l'appliquatif Git CMD, la plateforme en ligne GitHub et le générateur de documentation logicielle Doxygen. Plus précisément, l'objectif du projet est d'implémenter un algorithme de compression en utilisant des fichiers textes. Cela signifie que l'on cherche à réduire au maximum la place que prend une information, sans perte de données. Nous nous intéresserons ici surtout au fonctionnement du code de Huffman et à la représentation des bits en caractères 0 et 1.

Exceptionnellement, au vu de ma situation personnelle, il a été convenu que je mène les deux premières parties de ce projet seule. En effet, l'idée a été de prendre deux identités afin de tout de même pouvoir s'exercer au travail collaboratif. J'ai donc fonctionné avec l'identité Lindsay-04 en tant qu'administrateur et j'ai créé une identité fictive que j'ai nommée Lindsay-01 avec laquelle j'ai collaboré. Cela a permis de laisser des traces d'activités pour chacune des deux identités. Concernant la répartition des tâches, nous avons par conséquent fonctionné en lots (**Annexe 2: Répartition des tâches en lots**).

Ce rapport comporte trois parties. La première partie est une analyse du projet. La seconde partie décrit sa conception. La troisième partie aborde l'organisation dans le cadre du projet. Pour compléter ce rapport, nous fournissons trois annexes : « **Annexe 1 : Fichiers textes utilisés** », « **Annexe 2 : Répartition des tâches en lots** » et « **Annexe 3 : Documentation de code** ».

A - Présentation du projet Huffman coding

Le projet Huffman coding est composé de trois parties :

- **Partie 1 : De la lettre au bit** où il s'agit de visualiser ce que donnerait un texte en bits. Chaque caractère va être traduit en une chaîne binaire de 0 et de 1 correspondant à son octet ASCII.
- **Partie 2 : Le code de Huffman version naïve** où il s'agit de réduire au maximum le nombre de binaires nécessaires pour coder un texte en entrée. Cette partie se divise en trois étapes principales :
 - ❶ Les occurrences : établir une liste contenant tous les caractères présents dans un texte donné ainsi que leurs occurrences.
 - ❷ L'arbre : assigner à chaque lettre une suite binaire, en fonction de sa fréquence et sans préfixe. En effet, une lettre apparaissant de nombreuses fois doit être codée par une suite binaire la plus courte possible. De plus, un caractère ne peut pas être codé en tant que préfixe d'un autre. Le code binaire associé à chaque lettre va être défini par un arbre binaire que l'on va construire.
 - ❸ Le dictionnaire : coder/décoder un texte donné grâce au dictionnaire en convertissant un caractère en sa représentation binaire ASCII et grâce au code de Huffman.
- **Partie 3 : Optimisation** où il s'agit d'optimiser les différentes fonctionnalités du projet.

B - Principe du codage de Huffman

Le codage de Huffman repose sur la traduction en un code court d'un caractère, en fonction de sa fréquence d'apparition. Plus un caractère apparaît souvent dans le texte à coder, plus sa traduction sera courte. Nous rappelons qu'en codage ASCII, un caractère est codé sur un octet. Par conséquent, pour chaque caractère d'un texte, il faut huit bits en mémoire. Nous cherchons ici à réduire ce nombre de bits grâce au code de Huffman.

II – Conception

A - Architecture logicielle du projet

Afin de répondre aux problèmes posés par le projet, nous avons tout d'abord conçu l'architecture logicielle du projet qui comprend les cinq thèmes suivants, organisés en modules (fichier .h et fichier .c) :

➤ FromLetterToBit

- Un module Part 1 constitué de deux fichiers Part 1.c et Part 1.h pour les fonctions de la partie 1 : lecture d'un texte dans un fichier et traduction en son équivalent 0 et 1 dans un autre fichier, compter le nombre de caractères dans un fichier txt, affichage du nombre de caractères dans un fichier txt.

➤ HuffmanCodeNaiveVersion

- Un module Part 2 constitué de deux fichiers Part 2.c et Part 2.h pour les fonctions de la partie 2 : recherche d'occurrences, création d'un arbre Huffman, création d'un dictionnaire, encodage, décodage.

➤ Optimization

- Un module Part 3 constitué de deux fichiers Part 3.c et Part 3.h pour les fonctions de la partie 3 : optimisation des occurrences, optimisation de l'arbre, optimisation du dictionnaire, optimisation de l'encodage, optimisation du décodage.

➤ Menu

- Un module menu constitué de deux fichiers menu.c et menu.h pour la fonction menu () : liste d'options présentées à l'utilisateur.

➤ FunctionTest

- Un module test_parts constitué de deux fichiers test_parts.c et test_parts.h pour tester les fonctions.

Chaque thème est représenté par un dossier (répertoire) dans le système de fichiers.

Cette architecture logicielle nous a permis de réaliser un squelette de code implémenté sur Code::Blocks et qui compile.

Dans l'optique de réaliser un travail collaboratif, nous avons utilisé la plateforme en ligne GitHub sur laquelle nous avons créé un dépôt comportant quatre branches protégées :

- La branche main qui contient le code source final.
- La branche dev_start qui contient l'ensemble du programme mais sans code dans les fonctions c'est-à-dire le squelette de code. Le squelette de code a été élaboré ainsi : les déclarations prototypes des fonctions d'interface utilisent le mot-clé *extern*. On utilise le mot-clé *const* en paramètre des fonctions si cela est nécessaire. Pour la déclaration des fonctions on utilise des tabulations afin d'assurer une meilleure lisibilité.
- La branche dev_1 qui contient le squelette de code complété par Lindsay-01.
- La branche dev_2 qui contient le squelette de code complété par Lindsay-04.

B - Modélisation des structures de données utilisées

Pour stocker les données en mémoire, le projet nécessite la création des structures suivantes :

<pre>typedef struct LinkedList { char character; int number_of_occurrences; struct LinkedList* next; } LinkedList;</pre>	<pre>typedef struct Node { char character; int frequency; struct Node* left; struct Node* right; } Node;</pre>	<pre>typedef struct Element { Node* data; int number_of_occurrences; struct Element* next; } Element;</pre>	<pre>typedef struct Queue { Element* data_queue; } Queue;</pre>
---	---	--	--

Part 2.h

➤ Structure LinkedList

Nous avons eu besoin d'implémenter une liste chaînée composée d'une succession de maillons. Chacun de ces maillons contenant chaque caractère présent dans un texte donné et le nombre d'occurrences de ce caractère. C'est pour cela que la structure *LinkedList* est composée des trois champs suivants :

- une variable de type char représentant le caractère : *char character*.
- une variable entière (de type int) représentant son nombre d'occurrences : *int number_of_occurrences*
- un pointeur vers l'élément suivant : *struct LinkedList* next*

➤ Structure Node

Il a également fallu construire un arbre binaire. L'arbre binaire est une structure constituée d'éléments appelés nœuds reliés par des branches. Un nœud peut être :

- la racine de l'arbre : le premier élément de l'arbre (nœud sans prédécesseur).
- une feuille : nœud situé aux extrémités des branches défini par un caractère et sa fréquence (nœud externe).
- interne s'il est défini par sa fréquence ou son poids. Un nœud interne n'est ni une feuille ni la racine.

Ces nœuds sont liés par une relation de parenté. Excepté la racine et les feuilles, chaque nœud possède un parent et deux enfants : un « enfant gauche » et un « enfant droit ». La structure *Node* permet de définir un nœud ou l'arbre. Chaque nœud comporte les quatre champs suivants :

- une variable de type char représentant le caractère si le nœud est une feuille : *char character*
- une variable entière (de type int) représentant le nombre d'apparitions du caractère : *int frequency*
- l'adresse de l'enfant gauche : *struct Node* left*
- l'adresse de l'enfant droit : *struct Node* right*

➤ Structure Element et structure Queue

Nous avons utilisé une file à priorités afin de construire l'arbre de Huffman. Il s'agit d'une structure de données efficace permettant de stocker les données dans l'ordre FIFO (First In First Out c'est-à-dire « Premier Entré Premier Sorti ») et sur laquelle nous avons pu effectuer diverses opérations : insertion d'un élément dans la file (« enqueue »), retrait d'un élément dans la file (« dequeue »).

Pour réaliser la structure *Queue*, une autre structure nommée *Element* sera utilisée pour stocker les données dans la file. La structure *Queue* est donc composée d'un unique champ : *Element* data_queue*.

La structure *Element* est composée des trois champs suivants :

- une variable de type Node représentant les données : *Node* data*
- une variable entière (de type int) représentant le nombre d'occurrences : *int number_of_occurrences*
- un pointeur vers l'élément suivant : *struct Element* next*

➤ Structure Dico_Node

La structure *Dico_Node* serait utile pour la troisième partie du projet (non traitée).

```
typedef struct Dico_Node {  
  
    char character;  
    char *code;  
    struct Dico_Node* left;  
    struct Dico_Node* right;  
  
} Dico_Node;
```

Part 3.h

Enfin, nous avons choisi de définir le status de retour des fonctions de test dans l'énumération suivante :

<pre>typedef enum { OK = 0, ERREUR = 1 } Status;</pre>	}	status.h
--	---	----------

C - Implémentation des fonctions

1) Présentation des fonctions du projet

Cette partie du rapport concerne l'implémentation du projet en langage C afin d'apporter des solutions aux problèmes posés dans les parties 1 (De la lettre au bit) et 2 (Le code de Huffman version naïve). Les fonctions implémentées sont réparties selon différents modules organisés suivant les cinq thèmes de l'architecture logicielle.

➤ Fonctions du thème FromLetterToBit

Les fichiers Part 1.c et Part 1.h comportent trois fonctions :

- | |
|--|
| · la fonction <i>read_translate_file</i> |
| · la fonction <i>number_of_characters_file</i> |
| · la fonction <i>display_number_of_characters_file</i> |

➤ Fonctions du thème HuffmanCodeNaiveVersion

Les fichiers Part 2.c et Part 2.h comportent vingt-huit fonctions :

· la fonction <i>create_element</i>	· la fonction <i>display_tree</i>
· la fonction <i>add_element_start</i>	· la fonction <i>free_tree</i>
· la fonction <i>position</i>	· la fonction <i>create_queue</i>
· la fonction <i>number_of_occurrences_file</i>	· la fonction <i>is_empty_queue</i>
· la fonction <i>characters_occurrences_text</i>	· la fonction <i>enqueue</i>
· la fonction <i>display_list</i>	· la fonction <i>dequeue</i>
· la fonction <i>display_occurrences</i>	· la fonction <i>display_queue</i>
· la fonction <i>free_list</i>	· la fonction <i>delete_element_linkedlist</i>
· la fonction <i>create_node</i>	· la fonction <i>min_linkedlist</i>
· la fonction <i>create_empty_node</i>	· la fonction <i>create_Huffman_tree</i>
· la fonction <i>add_node</i>	· la fonction <i>Huffman_dictionary</i>
· la fonction <i>add_leaf_node</i>	· la fonction <i>translate_file_dictionary</i>
· la fonction <i>add_left</i>	· la fonction <i>compress_file</i>
· la fonction <i>add_right</i>	· la fonction <i>decompress_file</i>

➤ Fonctions du thème Optimization

Les fichiers Part 3.c et Part 3.h comportent sept fonctions :

- la fonction *optimization_occurrences*
- la fonction *sort_array_occurrences*
- la fonction *optimization_create_Huffman_tree*
- la fonction *organize_nodes_AVL*
- la fonction *optimization_compress_file*
- la fonction *optimization_decompress_file*
- la fonction *free_dico_tree*

➤ Fonctions du thème Menu

Les fichiers menu.c et menu.h comportent la fonction *menu*. Cette fonction ne possède pas de type de retour (void) et est appelée dans le main.

➤ Fonctions du thème FunctionTest

Les fichiers test_parts.c et test_parts.h comportent quatorze fonctions de test (allant de A à N). Ces fonctions, comportant un status de retour défini dans une énumération (**voir II - B**), permettent de tester les fonctionnalités demandées dans le projet. Le fichier test_parts.h ci-après contient l'ensemble des prototypes des fonctions de test.

```
test_parts.h x
1  /******
2  * \file   test_parts.h
3  * \brief  Header of the library which groups test functions.
4  *
5  * \author Lindsay REBEAU lindsay.rebeau@efrei.net
6  * \date   November 2020
7  * *****/
8
9  #ifndef TEST_PARTS_H_INCLUDED
10 #define TEST_PARTS_H_INCLUDED
11
12 #include "Part 1.h"
13 #include "Part 2.h"
14 #include "Part 3.h"
15 #include "status.h"
16
17 // Declaration of the test function prototypes
18
19 Status test_function_A();
20 Status test_function_B();
21 Status test_function_C();
22 Status test_function_D();
23 Status test_function_E();
24 Status test_function_F();
25 Status test_function_G();
26 Status test_function_H();
27 Status test_function_I();
28 Status test_function_J();
29 Status test_function_K();
30 Status test_function_L();
31 Status test_function_M();
32 Status test_function_N();
33
34 #endif // TEST_PARTS_H_INCLUDED
```


2) Explication des fonctions essentielles du projet

Dans ce rapport, nous avons choisi de développer les fonctions que nous avons jugées essentielles pour le projet.

➤ Fonctions de la partie 1 du projet (thème FromLetterToBit)

a) la fonction `read_translate_file` testée par la fonction de test A

Implémentation de la fonction `read_translate_file` sur Code::Blocks

Afin de lire un texte dans un fichier et le traduire en son équivalent 0 et 1 dans un autre fichier, nous sommes appuyés sur la table de correspondance entre un caractère et sa représentation binaire proposée dans le sujet. Chaque lettre possédant une représentation décimale ainsi qu'une représentation binaire. Le texte que nous avons utilisé est tiré du fichier Alice.txt.

La fonction `read_translate_file` prend comme paramètres le fichier d'entrée Alice.txt et le fichier de sortie Output.txt. Le but est de passer du décimal au binaire. Pour convertir un nombre décimal en nombre binaire (passage de la base 10 à la base 2), nous choisissons d'utiliser la méthode des divisions successives. Celle-ci consiste à effectuer des divisions entières successives par deux jusqu'à ce que le quotient devienne nul. Le reste de la division est un digit (chiffre) du résultat. Une boucle while permet de lire les caractères un par un dans le fichier. Elle s'arrête quand la fonction `fgetc` - définie dans la bibliothèque `#include <stdio.h>` et qui permet de lire un caractère - renvoie EOF (« End Of File » c'est-à-dire « fin du fichier »). C'est le cas si la fonction n'a pas pu lire de caractère. A la fin, nous obtenons la représentation binaire du nombre cherché en commençant par le dernier reste obtenu. Nous veillons à ne pas oublier les 0 du début s'il y en a.

Explication du code proposé

Au début de la fonction, nous créons deux pointeurs de FILE que nous initialisons à NULL afin de diminuer le risque d'erreur par la suite. Puis, nous déclarons les variables nécessaires à la fonction :

- une variable entière (de type int) nommée `current_character` permettant de stocker la valeur du caractère actuel au fur et à mesure de l'avancement dans la boucle
- une variable entière (de type int) nommée `i` permettant de parcourir le tableau
- une variable entière (de type int) nommée `j` permettant le parcours de la boucle for pour l'affichage
- une variable entière (de type int) nommée `quotient` permettant de stocker le quotient de la division euclidienne réalisée
- un tableau d'entiers nommé `current_character_binary` et de taille BUFSIZ - macro définie dans la bibliothèque `#include <stdio.h>` - pour stocker le reste des divisions successives et former la représentation binaire.

A l'exception de la variable `i` qui est initialisée à 1, toutes les variables sont par défaut initialisées à 0.

Nous procédons ensuite à l'ouverture du fichier d'entrée en lecture seule afin de pouvoir uniquement lire le contenu du fichier. Puis, nous procédons à l'ouverture du fichier de sortie en écriture seule afin de pouvoir uniquement écrire dans le fichier. Si le pointeur vaut NULL, nous indiquons que l'ouverture des fichiers a échoué. Si le pointeur est différent de NULL, cela signifie que l'ouverture des fichiers a bien fonctionné et que nous pouvons débiter la lecture et l'écriture dans les fichiers.

Nous commençons par lire le premier caractère du texte puis nous entrons dans la boucle while qui teste si ce caractère est différent de EOF. Les variables `i`, `j` et `quotient` doivent être réinitialisées à chaque entrée dans la boucle while car elles changent de valeur au fur et à mesure de l'avancement dans la boucle. La variable `quotient` reçoit la variable `current_character`. En effet, la première division se fait avec le dividende c'est-à-dire la représentation décimale du premier

caractère lu. A partir de la deuxième division, on considère que le dividende est le quotient de la division qui précède. Le diviseur est toujours deux. Tant que le quotient n'est pas nul, nous calculons le quotient et le reste. Le tableau d'entiers *current_character_binary* stocke le reste de la division euclidienne (opérateur modulo %) pour former le nombre binaire. Puis, la variable *quotient* stocke la division du quotient par deux. Nous incrémentons la variable *i* afin de passer à la case suivante du tableau.

Nous réalisons ensuite l'affichage de la représentation binaire sur huit bits de chaque caractère sur le fichier de sortie, grâce à la fonction *fprintf* en parcourant une boucle for. Cette fonction est définie dans la bibliothèque `#include <stdio.h>` et écrit une chaîne « formatée » dans un fichier. Pour cela, on initialise la variable *j* à huit et on réalise une décrémentation pour respecter le sens de lecture de la représentation binaire.

A la fin de l'exécution de la boucle, nous veillons à la fermeture des fichiers en utilisant la fonction *fclose*. Cette fonction est définie dans la bibliothèque `#include <stdio.h>` et permet la fermeture du fichier, la libération de la mémoire.

Explication de la méthode des divisions successives basée sur l'exemple de la lettre « l »

Dans le mot « Alice », prenons l'exemple de la lettre « l ». On lit le caractère « l » dont la représentation décimale est 108. La variable *quotient* reçoit la valeur du caractère actuel c'est-à-dire 108.

Dec	Hex	Binary	HTML	Char	Description
108	6C	01101100	l	l	Small l

1) Première division

Dividende = 108
Diviseur = 2

108 / 2 = 54
Quotient = 54

108 % 2 = 0
Reste = 0

2) Deuxième division

Quotient = 54
Diviseur = 2

54 / 2 = 27
Quotient = 27

54 % 2 = 0
Reste = 0

3) Troisième division

Quotient = 27
Diviseur = 2

27 / 2 = 13
Quotient = 13

27 % 2 = 1
Reste = 1

4) Quatrième division

Quotient = 13
Diviseur = 2

13 / 2 = 6
Quotient = 6

13 % 2 = 1
Reste = 1

5) Cinquième division

Quotient = 6
Diviseur = 2

6 / 2 = 3
Quotient = 3

6 % 2 = 0
Reste = 0

6) Sixième division

Quotient = 3
Diviseur = 2

3 / 2 = 1
Quotient = 1

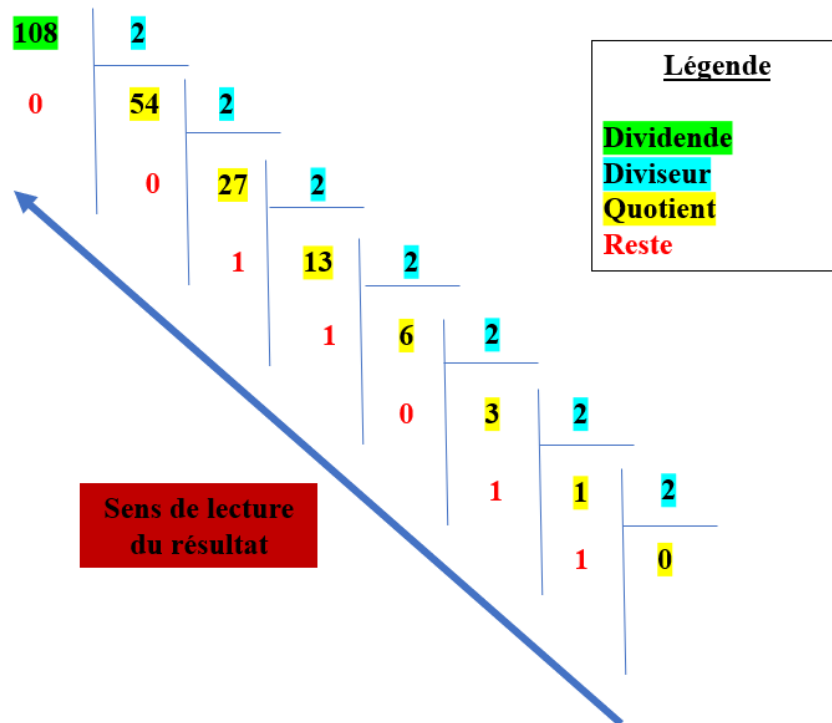
3 % 2 = 1
Reste = 1

7) Septième division

Quotient = 1
Diviseur = 2

1 / 2 = 0
Quotient = 0

1 % 2 = 0
Reste = 1



Nous obtenons bien la représentation binaire sur huit bits de 108 en veillant à ne pas oublier les 0 du début : 01101100.

Cette fonction `read_translate_file` est appelée dans la fonction de test A. Les sorties attendues de ce test doivent correspondre au tableau suivant issu du sujet du projet :

Alice.txt	Output.txt
Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do	01000001011011000110100101100011011001010 01000000111011101100001011100110010000001 10001001100101011001110110100101101110011 01110011010010110111001100111001000000111 01000110111100100000011001110110010101110 10000100000011101100110010101110010011110 01001000000111010001101001011100100110010 10110010000100000011011110110011000100000 01110011011010010111010001110100011010010 11011100110011100100000011000100111100100 10000001101000011001010111001000100000011 10011011010010111001101110100011001010111 00100010000001101111011011100010000001110 10001101000011001010010000001100010011000 01011011100110101100101100001000000110000 10110111001100100001000000110111101100110 00100000011010000110000101110110011010010 11011100110011100100000011011100110111101 11010001101000011010010110111001100111001 00000011101000110111100100000011001000110 1111

- b) les fonctions *number_of_characters_file* et *display_number_of_characters_file* testées par la fonction de test B

Nous cherchons ensuite à afficher le nombre de caractères dans un fichier texte. Pour cela, nous avons choisi de créer deux fonctions distinctes, prenant comme paramètres le nom du fichier texte à étudier. Cela assure une simplicité au niveau de l'écriture et au niveau du test :

- une fonction *number_of_characters_file* de type int qui retourne le nombre de caractères présents dans un fichier texte ou -1 s'il est impossible d'ouvrir le fichier texte.
- une fonction d'affichage nommée *display_number_of_characters_file* ne possédant pas de type de retour (void).

Implémentation de la fonction *number_of_characters_file* sur Code::Blocks

Dans la fonction *number_of_characters_file*, nous avons choisi de réaliser une boucle while permettant de lire les caractères un par un dans le fichier comme nous l'avons fait précédemment pour la fonction *read_translate_file*.

Explication du code proposé

Au début de la fonction, nous créons un pointeur de FILE que nous initialisons à NULL afin de diminuer le risque d'erreur par la suite. Puis, nous déclarons les variables nécessaires à la fonction que nous initialisons par défaut à 0 :

- une variable entière (de type int) nommée *current_character* permettant de stocker la valeur du caractère actuel au fur et à mesure de l'avancement dans la boucle
- une variable entière (de type int) nommée *counter* représentant le nombre de caractères présent dans le texte. Ce compteur s'incrémente à chaque itération de la boucle.

Nous procédons ensuite à l'ouverture du fichier texte en lecture seule afin de pouvoir uniquement lire le contenu du fichier. Si le pointeur vaut NULL, nous indiquons que l'ouverture du fichier a échoué. Si le pointeur est différent de NULL, cela signifie que l'ouverture du fichier a bien fonctionné et que nous pouvons débiter la lecture dans le fichier.

Nous commençons par lire le premier caractère du texte puis nous entrons dans la boucle while qui teste si ce caractère est différent de EOF. Tant que *fgetc* n'a pas retourné EOF c'est-à-dire à chaque caractère du texte rencontré, nous incrémentons la variable *counter*. A la fin de l'exécution de la boucle, nous veillons à la fermeture du fichier en utilisant la fonction *fclose*.

Implémentation de la fonction *display_number_of_characters_file* sur Code::Blocks

Explication du code proposé

Au début de la fonction *display_number_of_characters_file*, nous créons un pointeur de FILE que nous initialisons à NULL afin de diminuer le risque d'erreur par la suite. Puis, nous déclarons et initialisons à 0 une variable entière (de type int) nommée *result* permettant de stocker le retour de la fonction *number_of_characters_file*.

Nous procédons ensuite à l'ouverture du fichier texte en lecture seule afin de pouvoir uniquement lire le contenu du fichier. Si le pointeur vaut NULL, nous indiquons que l'ouverture du fichier a échoué. Si le

pointeur est différent de NULL, cela signifie que l'ouverture du fichier a bien fonctionné et que nous pouvons débiter la lecture dans le fichier.

Nous appelons la fonction *number_of_characters_file* et le retour de la fonction est stockée dans la variable *result*. Grâce à la fonction *printf* - définie dans la bibliothèque `#include <stdio.h>` et qui permet d'afficher du texte à l'écran - nous affichons sur la console la valeur de la variable *result* c'est-à-dire le nombre de caractères présents dans le fichier texte. Nous veillons à la fermeture du fichier en utilisant la fonction *fclose*.

Ces deux fonctions *number_of_characters_file* et *display_number_of_characters_file* sont appelées dans la fonction de test B. Les sorties attendues de ce test doivent correspondre au tableau suivant issu du sujet du projet (module Fondamentaux de l'algorithmique 3) :

Nom du fichier	Alice.txt	Output.txt
Nombre de caractères du fichier	103	824

➤ Fonctions de la partie 2 du projet (thème HuffmanCodeNaiveVersion)

1) Les occurrences

- a) les fonctions *characters_occurrences_text*, *display_list*, *display_occurrences*, et *free_list* testées par la fonction de test C

Implémentation de la fonction *characters_occurrences_text* sur Code::Blocks

Afin d'obtenir la correspondance entre caractères et occurrences, nous cherchons le maillon contenant la lettre correspondante pour y ajouter une occurrence. S'il n'existe pas, nous ajoutons un maillon contenant cette lettre. Ici, l'ordre n'a pas d'importance.

La fonction *characters_occurrences_text* prend comme paramètres la liste à retourner, le nouvel élément à ajouter à la liste s'il n'est pas encore présent ainsi que le fichier texte à étudier. Cette fonction retourne la liste contenant chaque caractère présent dans le texte ainsi que le nombre d'occurrences de ce caractère. De plus, elle fait appel à quatre autres fonctions également implémentées dans le fichier Part 2.c : les fonctions *position*, *create_element*, *add_element_start* et *number_of_occurrences_file*.

Dans cette fonction, une boucle *while* est nécessaire pour lire les caractères un par un dans le fichier. Celle-ci s'arrête quand la fonction *fgetc* renvoie EOF c'est-à-dire si la fonction n'a pas pu lire de caractère.

Explication du code proposé

Au début de la fonction, nous créons un pointeur de FILE que nous initialisons à NULL afin de diminuer le risque d'erreur par la suite. Puis, nous déclarons et initialisons une variable entière (de type *int*) nommée *current_character* permettant de stocker la valeur du caractère actuel au fur et à mesure de l'avancement dans la boucle.

Nous procédons ensuite à l'ouverture du fichier d'entrée en lecture seule afin de pouvoir uniquement lire le contenu du fichier. Puis, nous procédons à l'ouverture du fichier de sortie en écriture seule afin de pouvoir uniquement écrire dans le fichier. Si le pointeur vaut NULL, nous indiquons que l'ouverture du fichier a échoué. Si le pointeur est différent de NULL, cela signifie que l'ouverture du fichiers a bien fonctionné et que nous pouvons débiter la lecture dans le fichier.

Nous commençons par lire le premier caractère du texte puis nous entrons dans la boucle `while` qui teste si ce caractère est différent de EOF.

Nous appelons la fonction itérative *position* qui retourne la position d'un élément donné en argument dans une liste ou -1 dans le cas où l'élément ne se trouve pas dans la liste. Nous considérons que la position du premier élément est 1. Sachant que le dernier élément de la liste doit pointer vers NULL, nous parcourons la liste tant que NULL n'est pas atteint. L'utilisation du pointeur *next* permet de passer à l'élément qui suit à chaque fois. Si la fonction *position* retourne -1, la fonction *create_element* nous permet de créer avec la fonction *malloc* - définie dans la bibliothèque `#include <stdlib.h>` et qui permet d'allouer dynamiquement de la mémoire - un élément de la liste comportant le caractère lu.

Puis, nous ajoutons cet élément au début de la liste à l'aide de la fonction *add_element_start* dans laquelle l'adresse de l'élément à ajouter est initialisée au premier élément de la liste qui par la suite devient le nouvel élément.

Nous définissons le nombre d'occurrences de chaque caractère lu grâce à la fonction *number_of_occurrences_file*. Cette fonction de type `int` retourne le nombre d'occurrences du caractère recherché dans un fichier texte. Au début de cette fonction, nous créons un pointeur de `FILE` que nous initialisons à NULL afin de diminuer le risque d'erreur par la suite. Puis, nous déclarons les variables nécessaires à la fonction que nous initialisons par défaut à 0 :

- une variable entière (de type `int`) nommée *current_character* permettant de stocker la valeur du caractère actuel au fur et à mesure de l'avancement dans la boucle
- une variable entière (de type `int`) nommée *counter* représentant le nombre d'occurrences d'un caractère.. Ce compteur s'incrémente à chaque itération de la boucle.

Nous procédons ensuite à l'ouverture du fichier texte en lecture seule afin de pouvoir uniquement lire le contenu du fichier. Si le pointeur vaut NULL, nous indiquons que l'ouverture du fichier a échoué en retournant -1. Si le pointeur est différent de NULL, cela signifie que l'ouverture du fichier a bien fonctionné et que nous pouvons débiter la lecture dans le fichier. Nous commençons par lire le premier caractère du texte puis nous entrons dans la boucle `while` qui teste si ce caractère est différent de EOF. Si le caractère cherché en paramètre se trouve dans le fichier texte, nous incrémentons la variable *counter*. A la fin de l'exécution de la boucle, nous veillons à la fermeture du fichier en utilisant la fonction *fclose*.

De même, à la fin de l'exécution de la boucle de la fonction *characters_occurrences_text*, nous veillons à la fermeture du fichier en utilisant la fonction *fclose*.

Implémentation de la fonction *display_list* sur Code::Blocks

Nous cherchons à afficher une liste. La fonction *display_list* est une fonction d'affichage qui ne possède pas de type de retour (`void`). Elle prend comme paramètre une liste précédée du mot-clé *const* car celle-ci ne sera pas modifiée. Dans cette fonction, si la liste n'est pas vide, nous partons du premier élément et nous affichons le contenu de chaque élément de la liste (un caractère). Nous choisissons d'utiliser la récursivité car cela permet de réutiliser du code, simplifier l'écriture et résoudre des problèmes complexes. Ici, nous utilisons le pointeur *next* en tant que paramètre de l'appel récursif de la fonction *display_list* afin de passer à l'élément qui suit à chaque fois.

Implémentation de la fonction *display_occurrences* sur Code::Blocks

Nous cherchons à afficher le nombre d'occurrences de chaque caractère présent dans le texte. La fonction *display_occurrences* est une fonction d'affichage qui ne possède pas de type de retour (`void`). Elle prend comme paramètre une liste. Sachant que le dernier élément de la liste doit pointer vers NULL, une boucle `while` nous permet de parcourir la liste tant que NULL n'est pas atteint afin d'afficher les occurrences de chaque élément de la liste. Ceci si la liste n'est pas vide. Pour cela, nous utilisons le pointeur temporaire de type *LinkedList* * nommé *tmp*.

Initialement, nous plaçons *tmp* au début de la liste grâce à l'instruction *tmp* reçoit *linkedList*. L'utilisation du pointeur *next* permet de passer à l'élément qui suit à chaque fois.

Implémentation de la fonction *free_list* sur Code::Blocks

Nous cherchons à libérer la mémoire d'une liste. La fonction *free_list* est une fonction de libération de la mémoire qui ne possède pas de type de retour (void). Elle prend comme paramètre une liste. Dans cette fonction, si la liste n'est pas vide, on initialise les champs de chaque maillon de la liste, à 0 pour le champ *character* et à NULL pour le champ *next*. De même que dans la fonction *display_list*, nous partons du premier élément et nous utilisons la récursivité. De plus, le pointeur *next* est utilisé afin de passer à l'élément qui suit à chaque fois en tant que paramètre de l'appel récursif de la fonction *free* définie dans la bibliothèque `#include <stdlib.h>` et qui permet la libération dynamique de mémoire.

2) L'arbre

a) la fonction *create_Huffman_tree* testée par la fonction de test D

Implémentation de la fonction *create Huffman tree* sur Code::Blocks

Nous cherchons à renvoyer un arbre de Huffman à partir d'une liste d'occurrences. Pour cela, nous allons comparer des maillons de la liste avec les nœuds. Nous allons donc construire un arbre binaire en mettant au plus profond les caractères apparaissant le moins souvent c'est-à-dire « remonter l'arbre » de son extrémité avec les occurrences les plus faibles, vers la racine avec ses occurrences les plus fortes.

Nous créons une file à priorités contenant les caractères (lettres) présents dans la liste d'occurrences. Nous trions les lettres selon leur nombre d'occurrences par ordre croissant puis stocker des maillons de liste dans cette file. Après avoir créé un nœud vide, nous recherchons les deux nœuds comportant les valeurs minimum de la file. Enfin, nous calculons la somme de ces deux valeurs minimum que nous attribuons au nœud interne créé (nœud vide), inséré à l'arbre. Le texte que nous avons utilisé est tiré du fichier Tasses.txt.

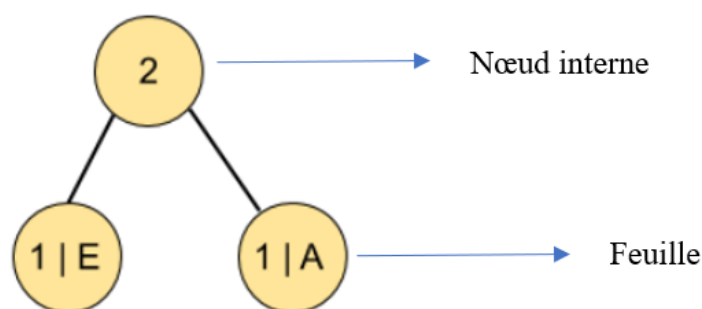
Explication du code proposé

Au début de la fonction, nous créons, déclarons et initialisons les variables nécessaires à la fonction :

- une file à priorités nommée *q* en utilisant les fonctions *create_queue* et *malloc*
- un pointeur temporaire de type *LinkedList ** nommé *tmp* initialement placé au début de la liste grâce à l'instruction *tmp* reçoit *linkedList*
- une variable de type *LinkedList ** nommée *minimum_value* permettant de stocker le plus petit élément de la liste lors de l'appel à la fonction *min_linkedlist*
- une variable de type *Node ** nommée *tree* permettant de stocker le nœud créé par la fonction *create_node*
- une variable de type *Node ** nommée *left* permettant de stocker les éléments de la file ou NULL si la file est vide
- une variable de type *Node ** nommée *right* permettant de stocker les éléments de la file ou NULL si la file est vide
- une variable de type *Node ** nommée *empty_node* permettant de stocker le premier nœud interne de l'arbre de Huffman (« 2 »)
- une variable de type *Node ** nommée *Huffman_tree* permettant de stocker le premier sous-arbre de l'arbre de Huffman
- une variable temporaire de type *Node ** nommée *temp* permettant de stocker un nœud
- une variable temporaire de type *Node ** nommée *temp2* permettant de stocker un nœud
- une variable de type *Node ** nommée *left2* permettant de stocker les éléments de la file ou NULL si la file est vide.
- une variable de type *Node ** nommée *right2* permettant de stocker les éléments de la file ou NULL si la file est vide

- une variable de type *Node ** nommée *empty_node 2* permettant de stocker le deuxième nœud interne de l'arbre de Huffman (« 3 »)
- une variable de type *Node ** nommée *Huffman_tree2* permettant de stocker le deuxième sous-arbre de l'arbre de Huffman
- une variable de type *Node ** nommée *left3* permettant de stocker les éléments de la file ou NULL si la file est vide
- une variable de type *Node ** nommée *right3* permettant de stocker les éléments de la file ou NULL si la file est vide
- une variable de type *Node ** nommée *empty_node 3* permettant de stocker le troisième nœud interne de l'arbre de Huffman (« 6 »)
- une variable de type *Node ** nommée *Huffman_tree3* permettant de stocker le troisième sous-arbre de l'arbre de Huffman
- une variable de type *Node ** nommée *Huffman_tree_final* permettant de stocker l'arbre final de Huffman
- une variable entière (de type int) nommée *sum_frequency* initialisée à 0 et qui permet de stocker la somme des fréquences minimales de l'enfant gauche et de l'enfant droit (premier sous-arbre)
- une variable entière (de type int) nommée *sum_frequency2* initialisée à 0 et qui permet de stocker la somme des fréquences minimales de l'enfant gauche et de l'enfant droit (deuxième sous-arbre)
- une variable entière (de type int) nommée *sum_frequency3* initialisée à 0 et qui permet de stocker la somme des fréquences minimales de l'enfant gauche et de l'enfant droit (troisième sous-arbre)

Premier sous-arbre de l'arbre de Huffman



Sachant que le dernier élément de la liste doit pointer vers NULL, une boucle while nous permet de parcourir la liste, à l'aide de la variable *tmp*, tant que NULL n'est pas atteint. Nous recherchons le plus petit élément de la liste d'occurrences reçue en paramètres de la fonction *create_Huffman_tree* en utilisant la fonction *min_linkedlist* de type *LinkedList ** qui isole et renvoie le plus petit élément d'une liste. Nous conservons l'élément dans la variable *minimum_value* puis nous créons un nœud avec la fonction *create_node* de type *Node **. Ce nœud comporte le caractère et la fréquence du plus petit élément de la liste. Une fois le nœud créé, nous l'insérons à la fin de la file avec la fonction *enqueue* qui ne possède pas de type de retour (void). De plus, nous définissons les données de la file (nombre d'occurrences). Enfin, nous supprimons l'élément *minimum_value* de la liste chaînée en utilisant la fonction *delete_element_linkedlist*. Nous utilisons le pointeur *next* de la variable *tmp* pour passer à l'élément suivant de la liste. Nous envisageons également le cas où la liste ne contient qu'un élément. La finalité de cette boucle while est de stocker les caractères dans la file de priorités *q* et par ordre croissant.

Affichage de la file de priorités obtenue avec la fonction *display_queue*

```

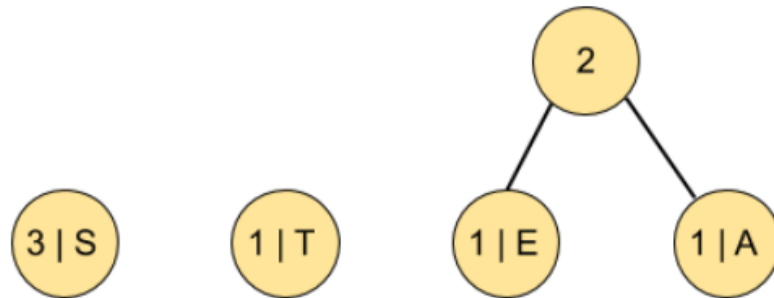
[E] [A] [T] [S]

Process returned 0 (0x0)   execution time : 2.570 s
Press any key to continue.
  
```

Nous retirons les deux minimums de la liste. Dans le cas du mot « TASSES », nous retirons « E » et « A » de la file avec la fonction *dequeue* de type *Node ** pour les regrouper un nœud interne « 2 ». Ce nœud va

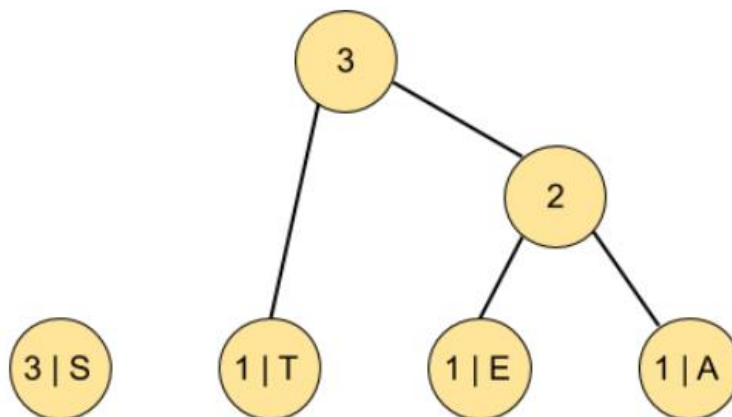
contenir deux enfants (E et A), et son poids va maintenant être 2 (1 pour le E + 1 pour le A). Pour cela, nous créons un nœud vide *empty_node* en utilisant la fonction *create_empty_node* de type *Node**. Ce nœud interne a pour fréquence la somme des deux fréquences minimales de « E » et « A ». Puis, nous attribuons les fréquences minimales aux enfants droit et gauche du nœud vide. Nous définissons la variable *sum_frequency*. Nous insérons ensuite le nœud interne « 2 » de fréquence *sum_frequency* dans l'arbre *Huffman_tree*. Nous ajoutons respectivement à gauche puis à droite les feuilles « 1| E » et « 1| A ». Enfin, nous affichons le premier sous-arbre obtenu en utilisant la fonction *display_tree*.

Construction de l'arbre (source : sujet du projet) – figure 1



Nous regroupons ensuite les deux plus petites occurrences ensemble (ici 2 et T). Pour cela, nous réalisons un nouveau tri dans la liste. Nous stockons « T » dans la variable *temp* et nous stockons « S » dans *temp2*. Nous utilisons la fonction *dequeue* pour retirer l'élément restant puis la fonction *enqueue* pour ajouter successivement à la file les nœuds « 2 », « T », et « S ». Le nœud vide *empty_node2* est créé en utilisant la fonction *create_empty_node*. Ce nœud interne a pour fréquence la somme des deux fréquences minimales de « 2 » et « T ». Puis, nous attribuons les fréquences minimales aux enfants droit et gauche du nœud vide. Nous définissons la variable *sum_frequency2*. Nous insérons ensuite le nœud interne « 3 » de fréquence *sum_frequency2* dans l'arbre *Huffman_tree2*. Nous ajoutons respectivement à gauche puis à droite la feuille « 1| T » et le nœud interne « 2 ». Enfin, nous affichons le deuxième sous-arbre obtenu en utilisant la fonction *display_tree*.

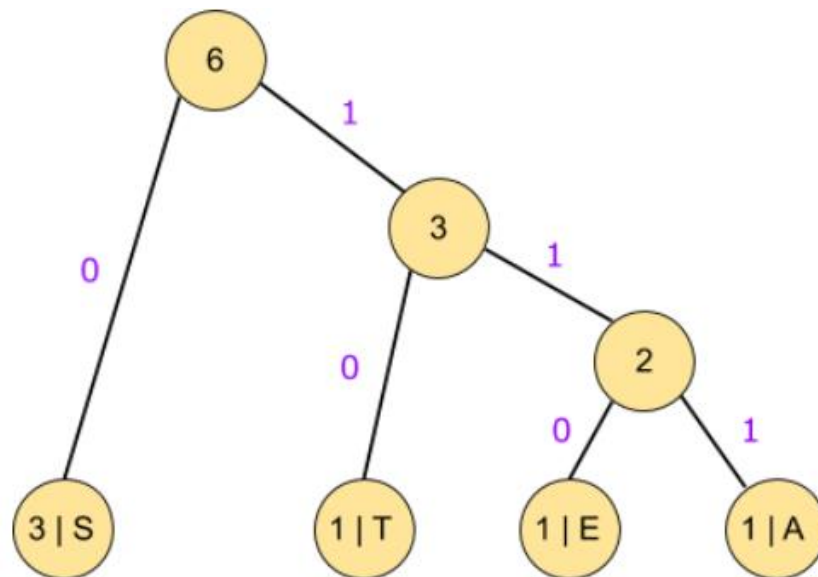
Construction de l'arbre (source : sujet du projet) – figure 2



Nous terminons la construction de l'arbre de Huffman avec les deux nœuds de poids 3. Nous ajoutons donc le nœud interne « 3 » à la file avec la fonction *enqueue*. Nous utilisons la fonction *dequeue* pour retirer « S » et « 3 ». Le nœud vide *empty_node3* est créé en utilisant la fonction *create_empty_node*. Ce nœud interne a pour fréquence la somme des deux fréquences minimales de « S » et « 3 ». Puis, nous attribuons les fréquences minimales aux enfants droit et gauche du nœud vide. Nous définissons la variable *sum_frequency3*. Nous insérons ensuite le nœud interne « 6 » de fréquence *sum_frequency3* dans l'arbre *Huffman_tree3*. Nous ajoutons respectivement à gauche puis à droite la feuille « 3| S » et le nœud interne « 3 ». Enfin, nous affichons le troisième sous-arbre obtenu en utilisant la fonction *display_tree*.

Pour construire l'arbre final, nous utilisons les fonctions *add_node* et *add_leaf_node*.

Arbre final (source : sujet du projet) – figure 3



Implémentation de la fonction *display_tree* sur Code::Blocks

Nous cherchons à afficher un arbre. Pour cela, il est possible de réaliser un parcours en largeur (itératif) ou un parcours en profondeur (itératif ou récursif) selon l'ordre préfixe, infixe, suffixe. La fonction *display_tree* est une fonction d'affichage qui ne possède pas de type de retour (void). Elle prend comme paramètre un arbre. Dans cette fonction, si l'arbre n'est pas vide, nous partons de la racine de l'arbre puis nous affichons les nœuds en suivant un parcours en profondeur récursif selon l'ordre préfixe.

Un nœud pouvant être interne, une feuille, ou la racine de l'arbre, nous envisageons deux affichages possibles. Une feuille sera affichée avec sa lettre et sa fréquence suivant le format (%d | %c). La racine de l'arbre ou un nœud interne seront affichés avec sa fréquence uniquement suivant le format (%d). Pour les mêmes raisons que nous avons explicitées auparavant, nous choisissons d'utiliser la récursivité dans les deux cas. De plus, nous utilisons les pointeurs *left* et *right* en tant que paramètre de l'appel récursif de la fonction *display_tree* afin de parcourir l'arbre et ses enfants.

Algorithme de l'affichage de l'arbre selon l'ordre préfixe
(source : support de cours du module Fondamentaux de l'algorithmique 3)

Prefix Order

```

preorder(node : Node)
If (node ≠ null)
    visit(node)
    preorder(node->left)
    preorder(node->right)
Endif
  
```

Implémentation de la fonction *free_tree* sur Code::Blocks

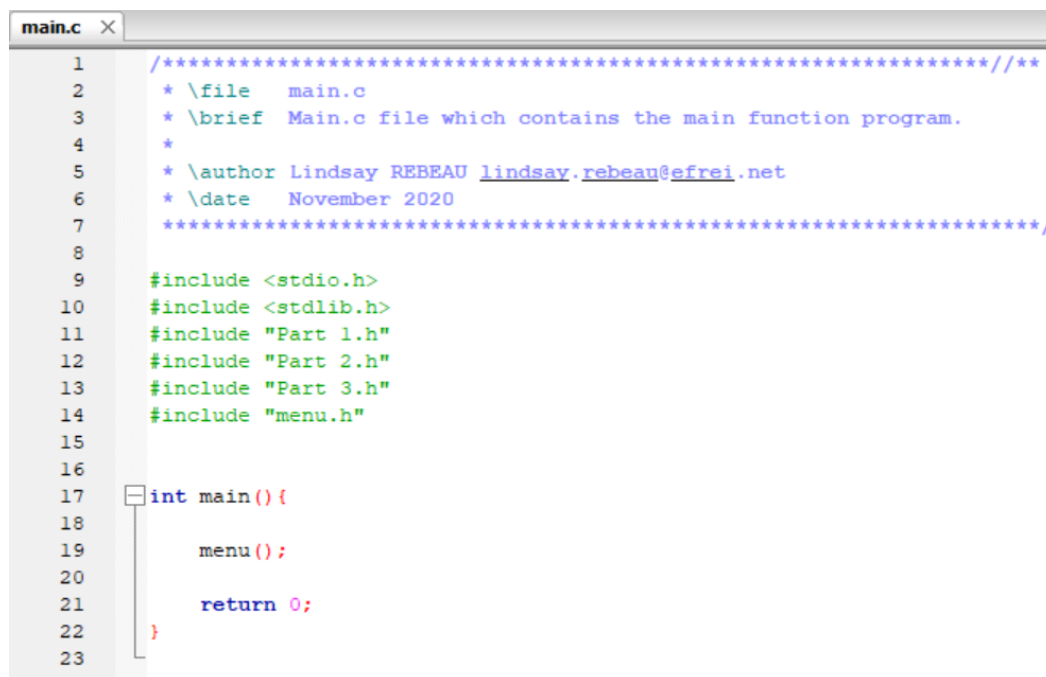
Nous cherchons à libérer la mémoire d'un arbre. La fonction *free_tree* est une fonction de libération de la mémoire qui ne possède pas de type de retour (void). Elle prend comme paramètre un arbre. Dans cette fonction, si l'arbre n'est pas vide, nous libérons récursivement les enfants situés à gauche et à droite ainsi que l'arbre grâce à la fonction *free*.

➤ **Fonctions de la partie 3 du projet (thème Optimization - non codées)**

L'implémentation des fonctions de test I, J, K, L, M et N font appel à plusieurs fonctions (**voir II - C**).

➤ **Fonctions du thème Menu**

Appel de la fonction *menu* dans le main sur Code::Blocks

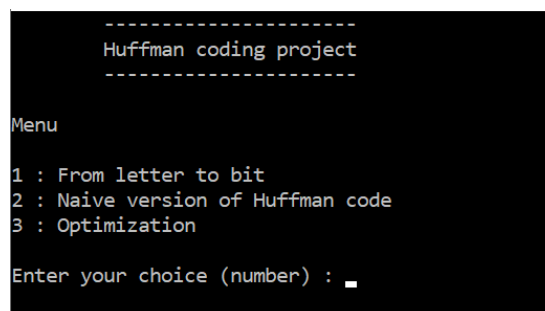


```
1  /*****
2  * \file   main.c
3  * \brief  Main.c file which contains the main function program.
4  *
5  * \author Lindsay REBEAU lindsay.rebeau@efrei.net
6  * \date   November 2020
7  *****/
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include "Part 1.h"
12 #include "Part 2.h"
13 #include "Part 3.h"
14 #include "menu.h"
15
16
17 int main(){
18     menu();
19
20     return 0;
21 }
22
23
```

La fonction *menu* permet de choisir la fonction à exécuter. Dans ce menu géré par plusieurs switch, l'utilisateur doit choisir parmi différentes options possibles. A la fin de chaque possibilités (*case*), une instruction *break* permet de sortir de la boucle. De plus, nous prévoyons un cas *default* indiquant que le choix de l'utilisateur est incorrect si les données sont mal renseignées.

Un premier switch teste la valeur de la variable *part*. Cette variable entière (de type *int*) stocke le choix de l'utilisateur c'est-à-dire la partie du projet qu'il souhaite tester.

Sortie console de la fonction menu (switch qui teste la valeur de la variable *part*)



```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : _
```

Un autre switch teste la valeur de la variable *function*. Cette variable entière (de type *int*) stocke le choix de l'utilisateur c'est-à-dire la fonction qu'il souhaite tester. En effet, dans un deuxième temps, en fonction de la partie choisie, l'utilisateur a le choix parmi les fonctions de test allant de A à N.

Sortie console de la fonction menu

```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 1

You have chosen part 1 : From letter to bit.

A : Function that reads a text in a file, and translates it to its 0 and 1 equivalent in another file.
B : Function that displays the number of characters in a txt file.

Enter A or B : _
```

Ici, l'utilisateur choisit la partie 1 du projet et peut tester les fonctions A ou B.

Sortie console de la fonction menu

```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 2

You have chosen part 2 : The naive version of the Huffman code.

C : Function that returns a list containing each character present in the text, and the number of occurrences of this character.
D : Function that returns a Huffman tree, from a list of occurrences.
E : Function that stores the dictionary from the Huffman tree in a txt file.
F : Function that translates a text into a binary sequence based on a Huffman dictionary.
G : Function that compresses a text file.
H : Function that decompresses a text file from a Huffman tree.

Enter C, D, E, F, G or H :
```

Ici, l'utilisateur choisit la partie 2 du projet et peut tester les fonctions allant de C à H.

Sortie console de la fonction menu

```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 3

You have chosen part 3 : Optimization.

I : Function which, by dichotomous search, adds to an occurrence to an array of nodes when the character has already been found, or which adds the node of the character otherwise.
J : Function that sorts an array of nodes based on occurrences.
K : Function which, using two queues, creates the Huffman tree from an array of nodes sorted by occurrences.
L : Function that organizes the nodes in an AVL according to the order of the characters present.
M : Function that optimally compresses a text file.
N : Function that decompresses a text file from a Huffman dictionary file.

Enter I, J, K, L, M or N : _
```

Ici, l'utilisateur choisit la partie 3 du projet et peut tester les fonctions allant de I à N.

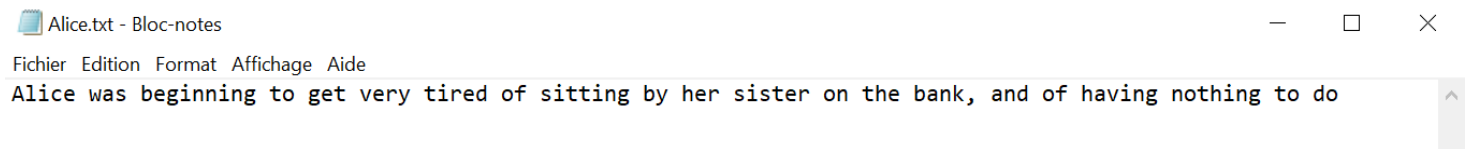
➤ Fonctions du thème FunctionTest

Fonction de test A

La fonction de test A doit permettre la lecture d'un texte dans un fichier, et sa traduction en son équivalent 0 et 1 dans un autre fichier. Pour cela, la fonction *read_translate_file* implémentée dans Part 1.c est appelée en prenant comme paramètres le fichier d'entrée Alice.txt et le fichier de sortie Output.txt.

Les sorties attendues de ce test concernent donc le fichier de sortie Output.txt. C'est pour cela que nous indiquons que l'exécution de la fonction de test A donne lieu à la modification du fichier Output.txt. Ce dernier initialement vide contiendra par la suite la représentation binaire du texte trouvé dans le fichier d'entrée Alice.txt.

Contenu du fichier d'entrée Alice.txt



Sortie console du test de la fonction de test A

```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 1

You have chosen part 1 : From letter to bit.

A : Function that reads a text in a file, and translates it to its 0 and 1 equivalent in another file.
B : Function that displays the number of characters in a txt file.

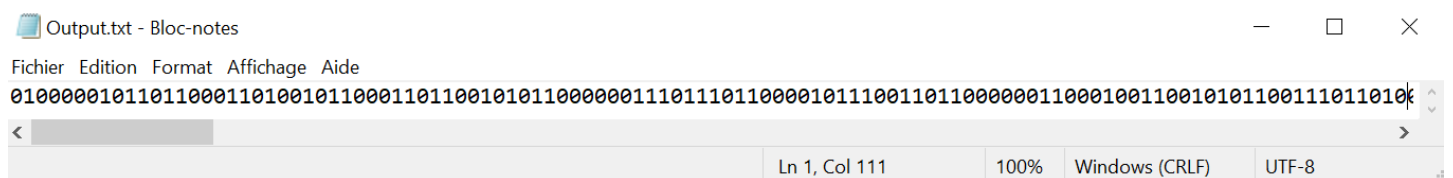
Enter A or B : A

You have chosen function A.

Now, the Output.txt file contains the binary representation of the text found in the Alice.txt file.

Process returned 0 (0x0)   execution time : 6.919 s
Press any key to continue.
```

Aperçu du contenu du fichier de sortie Output.txt après exécution de la fonction de test A



Fonction de test B

La fonction de test B doit permettre l’affichage du nombre de caractères dans un fichier texte. Pour cela, la fonction *display_number_of_characters_file* implémentée dans Part 1.c est appelée en prenant comme paramètre le fichier texte à étudier. La fonction *display_number_of_characters_file* fait elle-même appel à la fonction *number_of_characters_file*, également implémentée dans Part 1.c. Nous testons donc la fonctionnalité avec le fichier d’entrée Alice.txt puis avec le fichier de sortie Output.txt.

Les sorties attendues de ce test concernent donc les deux fichiers. Le fichier Alice.txt doit comporter 103 caractères. Le fichier Output.txt doit en comporter 824. Nous pouvons observer cela sur la capture d’écran de la sortie console du test.

Sortie console du test de la fonction de test B

```
-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 1

You have chosen part 1 : From letter to bit.

A : Function that reads a text in a file, and translates it to its 0 and 1 equivalent in another file.
B : Function that displays the number of characters in a txt file.

Enter A or B : B

You have chosen function B.

File Alice.txt
Number of characters in the file : 103

File Output.txt
Number of characters in the file : 824

Process returned 0 (0x0)   execution time : 3.776 s
Press any key to continue.
```

Fonction de test C

La fonction de test C doit permettre de renvoyer une liste contenant chaque caractère présent dans le texte, ainsi que le nombre d’occurrences de ce caractère. Nous souhaitons obtenir la correspondance entre caractère et occurrences. Pour cela, la fonction *characters_occurrences_text* implémentée dans Part 2.c est appelée.

Dans ce test, nous proposons ensuite d’afficher la liste grâce à la fonction *display_list* puis ses occurrences grâce à la fonction *display_occurrences*. Puis, nous veillons à bien libérer la mémoire allouée.

Nous testons donc la fonctionnalité avec le fichier d’entrée Alice.txt. Les sorties attendues de ce test doivent correspondre au tableau suivant issu du sujet du projet (module Fondamentaux de l’algorithmique 3) :

Alice.txt	Caractère	Occurrences
Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do		20
	n	10
	i	9
	t	9
	e	8
	o	7
	g	6
	r	4
	s	4
	a	4
	h	4
	b	3
	d	3
	f	2
	v	2
	y	2
	k	1
	A	1
	l	1
	w	1
	c	1

Sachant que l'ordre n'a ici pas d'importance, nous pouvons observer cela sur la capture d'écran de la sortie console du test.

Sortie console du test de la fonction de test C

```

-----
Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 2

You have chosen part 2 : The naive version of the Huffman code.

C : Function that returns a list containing each character present in the text, and the number of occurrences of this character.
D : Function that returns a Huffman tree, from a list of occurrences.
E : Function that stores the dictionary from the Huffman tree in a txt file.
F : Function that translates a text into a binary sequence based on a Huffman dictionary.
G : Function that compresses a text file.
H : Function that decompresses a text file from a Huffman tree.

Enter C, D, E, F, G or H : C

You have chosen function C.

List display : , k h f d y r v o t n g b s a w e c i l A

```

```

Display of occurrences

Occurrences of , : 1
Occurrences of k : 1
Occurrences of h : 4
Occurrences of f : 2
Occurrences of d : 3
Occurrences of y : 2
Occurrences of r : 4
Occurrences of v : 2
Occurrences of o : 7
Occurrences of t : 9
Occurrences of n : 10
Occurrences of g : 6
Occurrences of b : 3
Occurrences of s : 4
Occurrences of a : 4
Occurrences of w : 1
Occurrences of : 20
Occurrences of e : 8
Occurrences of c : 1
Occurrences of i : 9
Occurrences of l : 1
Occurrences of A : 1

Process returned 0 (0x0)   execution time : 3.996 s
Press any key to continue.

```

Fonction de test D

La fonction de test D doit permettre de renvoyer un arbre de Huffman, à partir d'une liste d'occurrences. Pour cela, nous affichons d'abord le nombre de caractères présents dans le fichier texte étudié en faisant appel à la fonction *display_number_of_characters_file*. Puis, nous renvoyons une liste contenant chaque caractère présent dans le texte, ainsi que le nombre d'occurrences de ce caractère grâce à la fonction *characters_occurrences_text*. Dans ce test, nous proposons ensuite d'afficher la liste grâce à la fonction *display_list* puis ses occurrences grâce à la fonction *display_occurrences*.

Le test se poursuit par la création de l'arbre de Huffman grâce à l'appel de la fonction *create_Huffman_tree* prenant en paramètres la liste d'occurrences précédemment établie. Après avoir affiché l'arbre de Huffman, selon l'ordre préfixe, grâce à la fonction d'affichage *display_tree*, nous veillons à bien libérer la mémoire allouée.

Contenu du fichier Tasses.txt



Nous testons donc la fonctionnalité avec le fichier Tasses.txt contenant le mot « TASSES ». Les sorties attendues de ce test sont donc les suivantes :

- Le fichier Tasses.txt doit comporter 6 caractères.
- Sachant que l'ordre n'a également ici pas d'importance, la liste doit correspondre au schéma suivant issu du sujet du projet (module Fondamentaux de l'algorithmique 3) :

Exemple : pour le texte "TASSES", on veut la liste :



- L'arbre de Huffman doit s'afficher dans l'ordre choisi c'est-à-dire l'ordre préfixe. Il est précédé de l'affichage de chacun de sous-arbres qui le constitue.

Nous pouvons observer l'ensemble des sorties attendues sur la capture d'écran de la sortie console du test.

Sortie console du test de la fonction de test D

```
-----
 Huffman coding project
-----

Menu

1 : From letter to bit
2 : Naive version of Huffman code
3 : Optimization

Enter your choice (number) : 2

You have chosen part 2 : The naive version of the Huffman code.

C : Function that returns a list containing each character present in the text, and the number of occurrences of this character.
D : Function that returns a Huffman tree, from a list of occurrences.
E : Function that stores the dictionary from the Huffman tree in a txt file.
F : Function that translates a text into a binary sequence based on a Huffman dictionary.
G : Function that compresses a text file.
H : Function that decompresses a text file from a Huffman tree.

Enter C, D, E, F, G or H : D

You have chosen function D.

File Tasses.txt
Number of characters in the file : 6

List display : E S A T
```



```

Display of occurrences

Occurrences of E : 1
Occurrences of S : 3
Occurrences of A : 1
Occurrences of T : 1

Huffman tree

-----
Display subtree (prefix order) : (2) (1 | E) (1 | A)
-----
Display subtree (prefix order) : (3) (1 | T) (2)
-----
Display subtree (prefix order) : (6) (3 | S) (3)
-----

Tree display (prefix order) : (6) (3 | S) (3) (1 | T) (2) (1 | E) (1 | A)

Process returned 0 (0x0)   execution time : 2.807 s
Press any key to continue.

```

Fonction de test E : La fonction de test E doit permettre de stocker dans un fichier texte le dictionnaire issu de l'arbre de Huffman.

Fonction de test F : La fonction de test F doit permettre de traduire un texte en une suite binaire basée sur un dictionnaire de Huffman.

Fonction de test G : La fonction de test G doit permettre de compresser un fichier texte. Le fichier d'entrée ne doit pas être modifié, un autre fichier, contenant le texte compressé doit être créé.

Fonction de test H : La fonction de test H doit permettre de décompresser un fichier texte à partir d'un arbre de Huffman. Le fichier d'entrée ne doit pas être modifié, un autre fichier, contenant le texte décompressé doit être créé.

Fonction de test I : La fonction de test I doit permettre, par recherche dichotomique, d'ajouter à un tableau de nœuds une occurrence quand le caractère a déjà été trouvé, ou d'ajouter le nœud du caractère sinon.

Fonction de test J : La fonction de test J doit permettre de trier un tableau de nœuds en fonction des occurrences.

Fonction de test K : La fonction de test K doit permettre, en utilisant deux files, de créer l'arbre de Huffman à partir d'un tableau de nœuds trié par occurrences.

Fonction de test L : La fonction de test L doit permettre d'organiser les nœuds dans un AVL en fonction de l'ordre des caractères présents.

Fonction de test M : La fonction de test M doit permettre de compresser un fichier texte de façon optimisée. Le fichier d'entrée ne doit pas être modifié, un autre fichier, contenant le texte compressé doit être créé.

Fonction de test N : La fonction de test N doit permettre de décompresser un fichier texte à partir d'un fichier dictionnaire d'Huffman. Le fichier texte d'entrée ne doit pas être modifié, un autre fichier, contenant le texte décompressé doit être créé.

III – Organisation

Les conditions particulières dans lesquelles j'ai dû réaliser ce projet ne m'ont pas empêché d'expérimenter un travail collaboratif. Effectivement, l'utilisation d'identités fictives qui m'a été conseillée

fut une bonne solution à l'impossibilité d'intégrer une équipe de projet. De plus, cela m'a confrontée à la gestion de tous les paramètres d'un projet. Toutefois, travailler ainsi peut aussi avoir des inconvénients car il peut être plus bénéfique de pouvoir partager et échanger sur le travail demandé avec une « vraie » équipe. Cela peut permettre une réflexion plus approfondie sur le sujet, une répartition des tâches qui facilite l'avancement du projet et le respect des délais.

Par ailleurs, j'ai rencontré quelques difficultés pour intégrer la documentation Doxygen et plus particulièrement le fichier Doxyfile au dépôt GitHub. De plus, il ne m'a pas été possible de respecter l'ensemble des attendus du projet (délais, codage des fonctions E, F, et G de la partie 2 du projet) comme je l'aurais souhaité et comme cela avait été convenu avec les enseignants.

Conclusion générale

En somme, ce projet fut très enrichissant. En effet, il m'a permis d'approfondir le langage C, d'acquérir des connaissances en génie logiciel et de maîtriser de nouveaux outils tels que Git, GitHub, Doxygen. J'ai pu aborder le codage de Huffman, définir une architecture logicielle, mettre en œuvre la programmation modulaire, réaliser des tests unitaires, et acquérir de bonnes pratiques de programmation et de documentation de code.

Malgré les difficultés rencontrées, je suis satisfaite du travail auquel j'ai pu aboutir bien qu'il pourrait être amélioré et complété. J'ai apprécié et je pense réutiliser à l'avenir les principes et les méthodes de gestion de projet qui ont été nécessaires pour mener à bien ce projet.

Références bibliographiques

- Documents et ressources du module Fondamentaux de l'algorithmique 3 (supports de cours, supports de TD/TP)
- Documents et ressources du module Introduction au génie logiciel (supports de cours, supports de TP, lectures)
- Manuel du programmeur Linux

Alice.txt

- Fichier source contenant le texte à compresser (extrait du chapitre 1 du roman « Alice au pays des merveilles » de Lewis Carroll)

Output.txt

- Traduction du fichier Alice.txt en sa représentation binaire

Tasses.txt

- Fichier source contenant le texte à compresser (le mot « TASSES »)

Tasses - Output.txt

- Traduction du fichier Tasses.txt en sa représentation binaire

Dico.txt

- Dictionnaire de Huffman obtenu du fichier Tasses.txt

Annexe 2 : Répartition des tâches en lots

Lot 1 – Administrateur (Lindsay-04)

- Création du projet Huffman_coding_project
- Configuration de l'identité par défaut du compte (nom, prénom, adresse mail)
- Création du dépôt distant sur GitHub
- Ajout du collaborateur : Lindsay-01
- Création des protections de branches :
 - main
 - dev_*
- Création d'une branche dev_start
- Ajout de main.c dans dev_start
- Ajout de status.h dans dev_start
- Demande de merge de dev_start vers main (Pull request Start dev)
- Création des thèmes de l'architecture logicielle dans dev_start
 - FromLetterToBit
 - HuffmanCodeNaiveVersion
 - Optimization
 - Menu
 - FunctionTest
- Ajout de Part 1.c dans dev_start
- Ajout de Part 1.h dans dev_start
- Ajout de Part 2.c dans dev_start
- Ajout de Part 2.h dans dev_start
- Ajout de Part 3.c dans dev_start
- Ajout de Part 3.h dans dev_start
- Ajout de menu.c dans dev_start
- Ajout de menu.h dans dev_start
- Ajout de test_parts.c dans dev_start
- Ajout de test_parts.h dans dev_start
- Ajout de Alice.txt
- Ajout de Output.txt
- Demande de merge de dev_start vers main (Pull request Code skeleton)
- Validation des pull request de Lindsay-01

Lot 2 – Collaborateur (Lindsay-01)

- Clône du dépôt distant
- Création d'une branche dev_1
- Implémentation du code dans le fichier Part 1.c
- Implémentation du code dans le fichier Part 2.c
- Demande de merge de dev_1 vers main (Pull request Dev 1)

Lot 3 – Administrateur (Lindsay-04)

- Ajout de la documentation Doxygen
- Création d'une branche dev_2

- Mise à jour du code dans le fichier Part 2.h
- Mise à jour du code dans le fichier Part 2.c
- Mise à jour du code dans le fichier test_parts.c
- Ajout de Tasses.txt dans dev_2
- Ajout de Tasses-Output.txt dans dev_2
- Ajout de Dico.txt dans dev_2

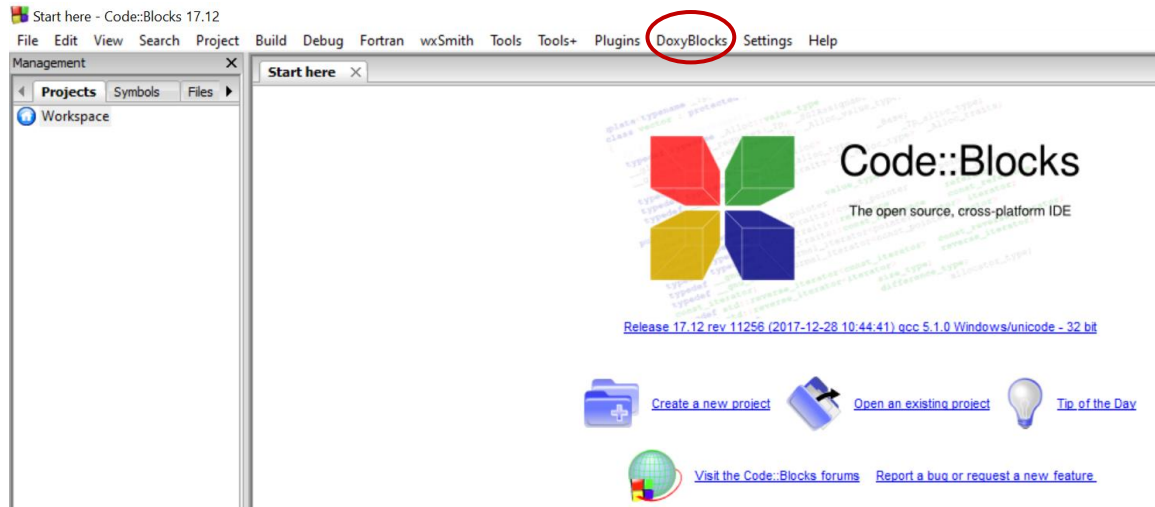
Lot 4 – Collaborateur (Lindsay-01)

- Demande de merge de dev_2 vers main (Pull request Dev 2)

Annexe 3 : Documentation de code

Pour documenter le code, nous avons utilisé DoxyBlocks, un plugin pour Code::Blocks, qui intègre Doxygen dans l'IDE. Cela nous a permis d'insérer des commentaires dans le code, d'ajouter de la documentation pour les fichiers du projet dans le but de faciliter la compréhension du programme.

Localisation de DoxyBlocks sur Code::Blocks



Aperçu de la documentation Doxygen du projet Huffman coding

Huffman coding project: File List

D:\Huffman_coding_project\doxygen/html/files.html

Huffman coding project

Implementation of a compression algorithm

Main Page Related Pages Classes Files

File List

Here is a list of all documented files with brief descriptions:

FromLetterToBit	
Part 1.h	
FunctionTest	
test_parts.h	Header of the library which groups test functions
HuffmanCodeNaiveVersion	
Part 2.h	
Menu	
menu.h	Header of the library which defines the menu module
Optimization	
Part 3.h	
main.c	Main.c file which contains the main function program
status.h	Header of the library which allows the management of the tests

Generated by **doxygen** 1.8.20

Exemple de documentation des structures de données utilisées

➤ Structure LinkedList

The screenshot shows the Doxygen documentation for the `LinkedList` structure. The browser address bar indicates the file path: `D:/Huffman_coding_project/doxygen/html/struct_linked_list.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is present on the right. The main content area is titled "LinkedList Struct Reference". Under "Public Attributes", the following are listed: `char character`, `int number_of_occurrences`, and `struct LinkedList * next`. The "Member Data Documentation" section shows the `next` member, which is a pointer to `LinkedList`, with the description "the next element of the list". At the bottom, it states that the documentation was generated from the file `HuffmanCodeNaiveVersion/Part 2.h`. The footer indicates it was generated by `doxygen` 1.8.20.

➤ Structure Node

The screenshot shows the Doxygen documentation for the `Node` structure. The browser address bar indicates the file path: `D:/Huffman_coding_project/doxygen/html/struct_node.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is present on the right. The main content area is titled "Node Struct Reference". Under "Public Attributes", the following are listed: `char character`, `int weight`, `struct Node * left`, and `struct Node * right`. The "Member Data Documentation" section shows the `left` and `right` members, which are pointers to `Node`. At the bottom, it states that the documentation was generated from the file `HuffmanCodeNaiveVersion/Part 2.h`. The footer indicates it was generated by `doxygen` 1.8.20.

➤ Structure Element

The screenshot shows the Doxygen documentation for the `Element` structure. The browser address bar indicates the file path: `D:/Huffman_coding_project/doxygen/html/struct_element.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is present on the right. The main content area is titled "Element Struct Reference". Under "Public Attributes", the following are listed: `Node * data` and `struct Element * next`. The "Member Data Documentation" section shows the `data` member, which is a pointer to `Node`, and the `next` member, which is a pointer to `Element`. At the bottom, it states that the documentation was generated from the file `HuffmanCodeNaiveVersion/Part 2.h`. The footer indicates it was generated by `doxygen` 1.8.20.

➤ Structure Queue

The screenshot shows a web browser window with the URL `D:/Huffman_coding_project/doxygen/html/struct_queue.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is on the right. The main content area is titled "Queue Struct Reference". Below this, there is a section for "Public Attributes" which lists a single element: `data_queue`. A note states that the documentation was generated from the file `HuffmanCodeNaiveVersion/Part 2.h`. The footer indicates it was generated by doxygen 1.8.20.

➤ Structure Dico_Node

The screenshot shows a web browser window with the URL `D:/Huffman_coding_project/doxygen/html/struct_dico__node.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is on the right. The main content area is titled "Dico_Node Struct Reference". Below this, there is a section for "Public Attributes" which lists: `character` (char), `code` (char *), `left` (struct Dico_Node *), and `right` (struct Dico_Node *). A note states that the documentation was generated from the file `Optimization/Part 3.h`. The footer indicates it was generated by doxygen 1.8.20.

Exemple de documentation du fichier test_parts.h

The screenshot shows a web browser window with the URL `D:/Huffman_coding_project/doxygen/html/test_parts_8h.html`. The page title is "Huffman coding project" with the subtitle "Implementation of a compression algorithm". The navigation bar includes "Main Page", "Related Pages", "Classes", and "Files". A search bar is on the right. The main content area is titled "test_parts.h File Reference". Below this, there is a section for "Functions" which lists 14 functions: `test_function_A()` through `test_function_N()`. A "Detailed Description" section follows, containing the header of the library which groups test functions, the author "Lindsay REBEAU", and the date "November 2020".

Exemple de code source documenté du fichier Part 1.h

```
Part 1.h x
1  /*****
2   * \file   Part 1.h
3   * \brief  Header of the library allowing the management of Part 1.
4   *
5   * \author Lindsay REBEAU lindsay.rsbeau@sfr.fr lindsay.rsbeau@sfr.fr
6   * \date   November 2020
7   *****/
8
9  #ifndef PART_1_H_INCLUDED
10 #define PART_1_H_INCLUDED
11
12 #include <stdio.h>
13 #include <stdlib.h>
14
15 /**
16  * \brief Function to read a text in a file (source_file), and to translate it to its 0 and 1 equivalent in another file (destination_file).
17  * \param name_source_file, a string to define the input file.
18  * \param name_destination_file, a string to define the output file.
19  */
20 extern void read_translate_file (const char* name_source_file, const char* name_destination_file);
21
22 /**
23  * \brief Function to count the number of characters in a text file.
24  * \param name_file, a string to define the text file.
25  * \return \c -1 if it's impossible to open the text file.
26  * \return \c the number of characters present in the text file.
27  */
28 extern int number_of_characters_file (const char* name_file);
29
30 /**
31  * \brief Function to display the number of characters in a text file.
32  * \param name_file, a string to define the text file.
33  */
34 extern void display_number_of_characters_file (const char* name_file);
35
36 #endif // PART_1_H_INCLUDED
```