Promo 2024

Projet Introduction au Génie Logiciel

PROJET HUFFMAN

MICHAEL SOK

NOE LEFEVRE

ARTHUR GOBILLIARD

ERIC WANG

Intitulé	**Introduction au génie logiciel**
Activité	Projet *Huffman – Grp. D*
Intervenant(s)	MAURICE, Benjamin

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Projet Huffman

Table des matières

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Projet Huffman

Introduction générale

Pour ce premier projet d'Introduction au Génie Logiciel, nous devons nous baser notre projet en Structure de Données et appliquer ce que nous avons appris lors de nos TP. Nous allons donc appliquer cela pour créer une architecture logicielle tout en respectant de bonnes pratiques de développement. Cela a pour but de rendre notre code modulaire et facile de maintenance. Pour cela nous nous sommes aidés des TP sur le HugeNumberCalculator et nous nous sommes basés dessus.

Nous avons d'abord partagé les tâches afin de pouvoir avancer rapidement et efficacement. Pour la programmation en C, nous avons adopté la méthode de programmation en binôme, ainsi, une personne s'occupe de l'écriture tandis que l'autre sera là pour détecter les problèmes et conseiller celui qui écrit. Les binômes vont tourner en permanence afin de créer une vraie cohésion d'équipe mais aussi pour aider à la compréhension du code.

Nous avons décidé d'écrire un code qui est fonctionnel, puis après que tout fonctionne, de commencer la partie optimisation afin d'améliorer la maintenance du code.

Outils utilisés

Pour communiquer durant cette période de confinement, nous avons utilisé Discord pour pouvoir nous appeler en groupe, mais aussi pour pouvoir partager notre écran à tous.

Pour tout ce qui est schéma, nous avons utilisé des modules sur Excel pour pouvoir modéliser notre plan de projet.

Pour la programmation en C, nous avons utilisé de nombreux IDE, (xCode, CodeBlocks, Visual Studio). Le compilateur MacOS nous a été très utile pour repérer les erreurs et pour les corriger.

Pour finir, nous avons utilisé Git et GitHub pour le partage du projet, ainsi chacun pouvait avancer sans à avoir attendre que quelqu'un envoie le fichier à chaque modification et Doxygen pour la documentation de notre projet.

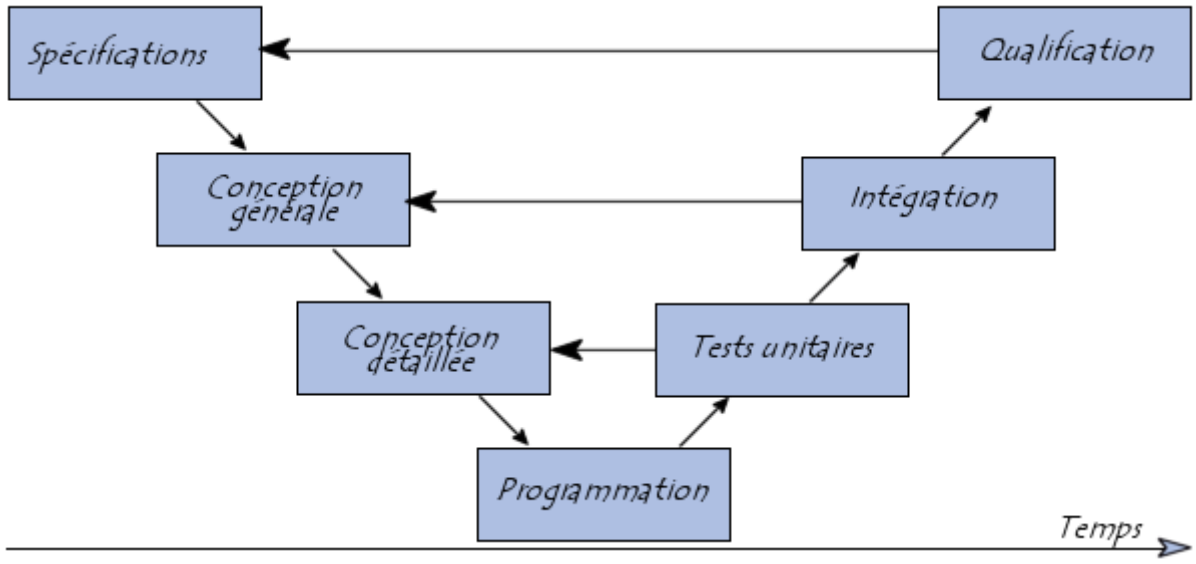
LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Projet Huffman

Composition de l'équipe/répartition des tâches

Nom	Rôle	Tâche
LEFEVRE Noé	Chef de projet	Tests unitaires des fonctions
GOBILLIARD Arthur	Architecte	Création de l'architecture logicielle
SOK Michaël	Architecte	Création de l'architecture logicielle
WANG Eric	Documentaliste	Documentation du projet

Pour le projet nous avons adopté un cycle en V :

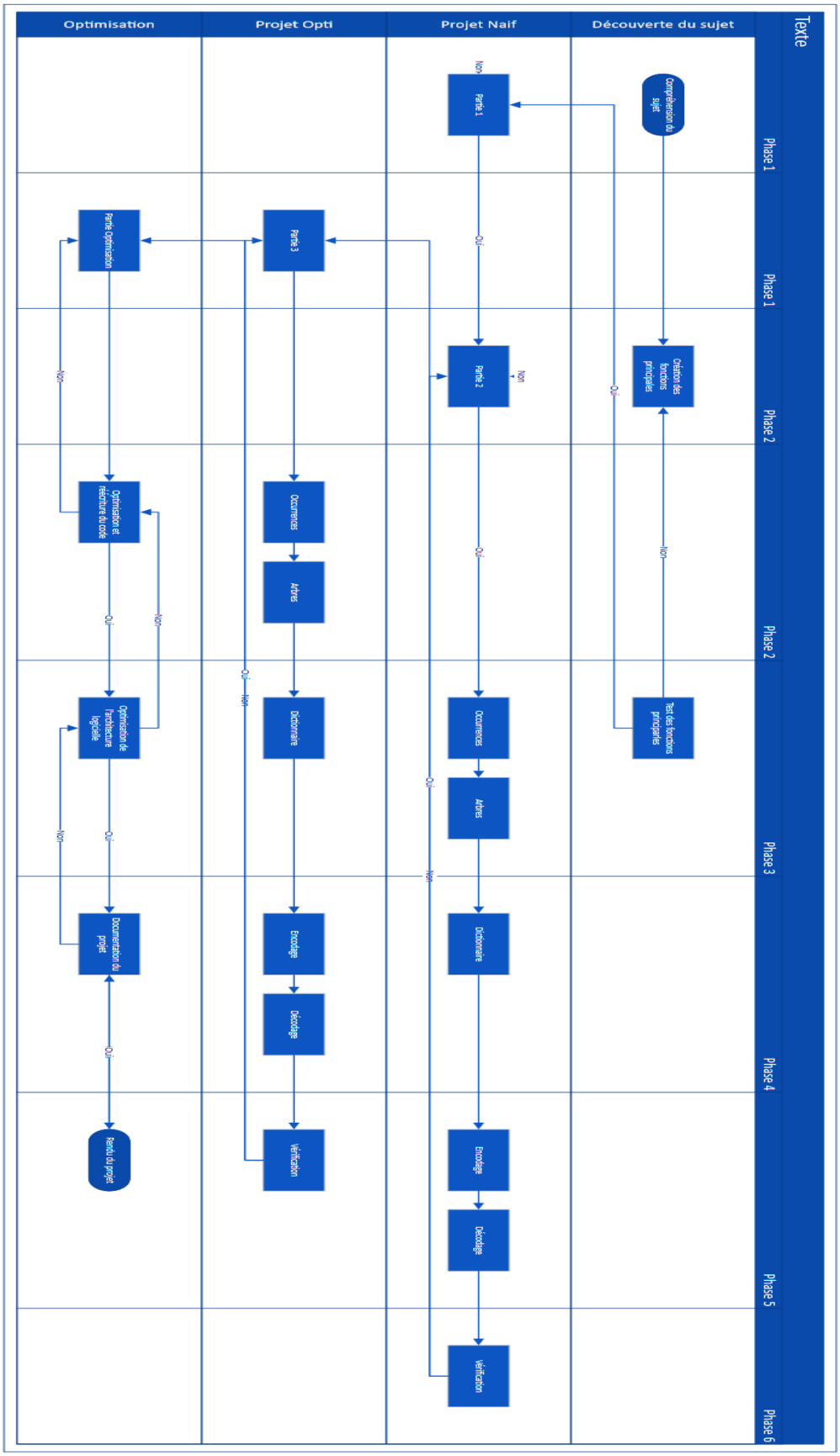


LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Nous nous sommes imposé ce diagramme fonctionnel pour suivre l'avancement de notre projet :

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Projet Huffman



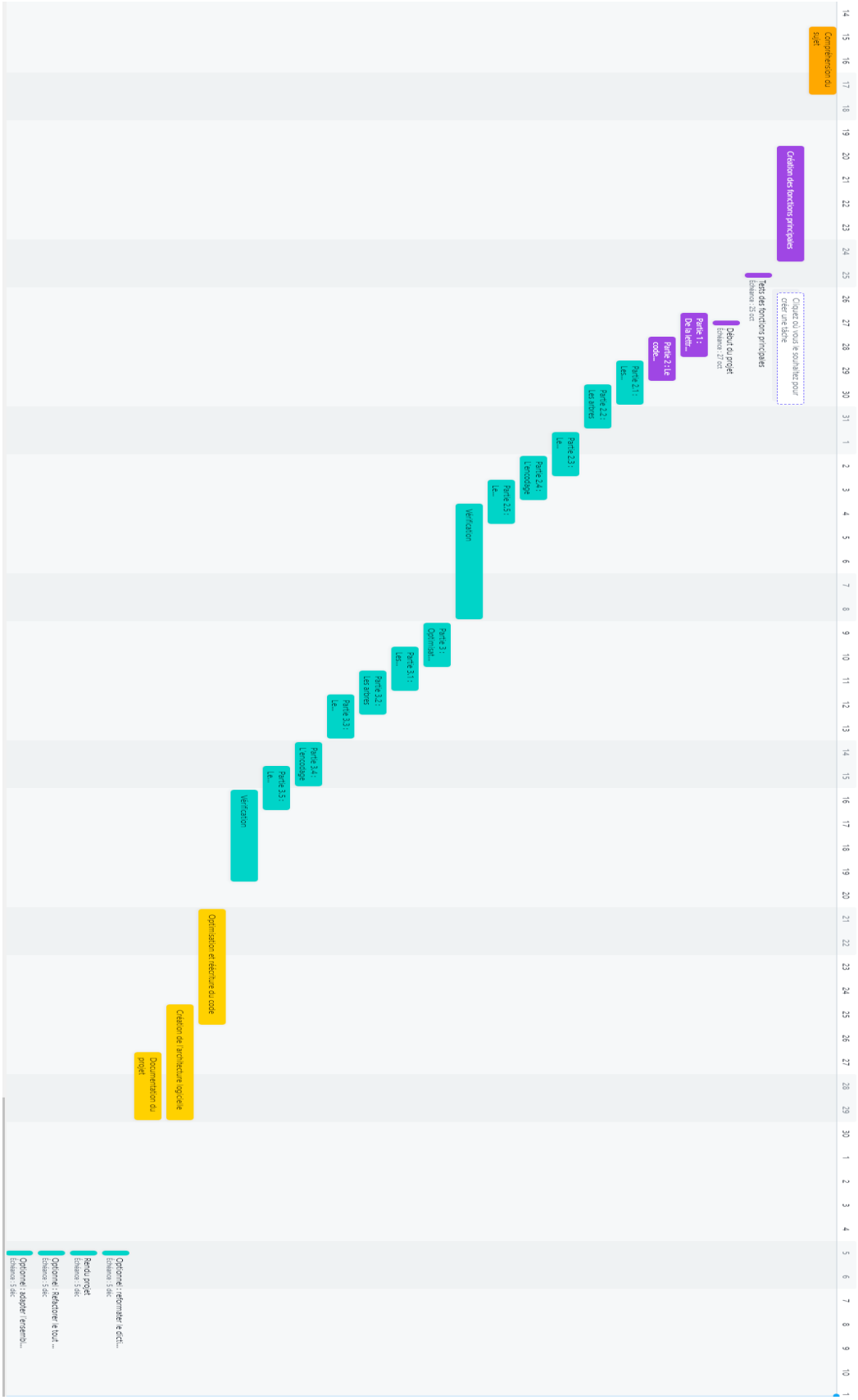
LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Planification des étapes du projet

Pour planifier les tâches du projet, nous avons utilisé un diagramme de Gantt pour pouvoir avoir une prévision précise de notre planning afin de rendre le projet dans les temps.

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

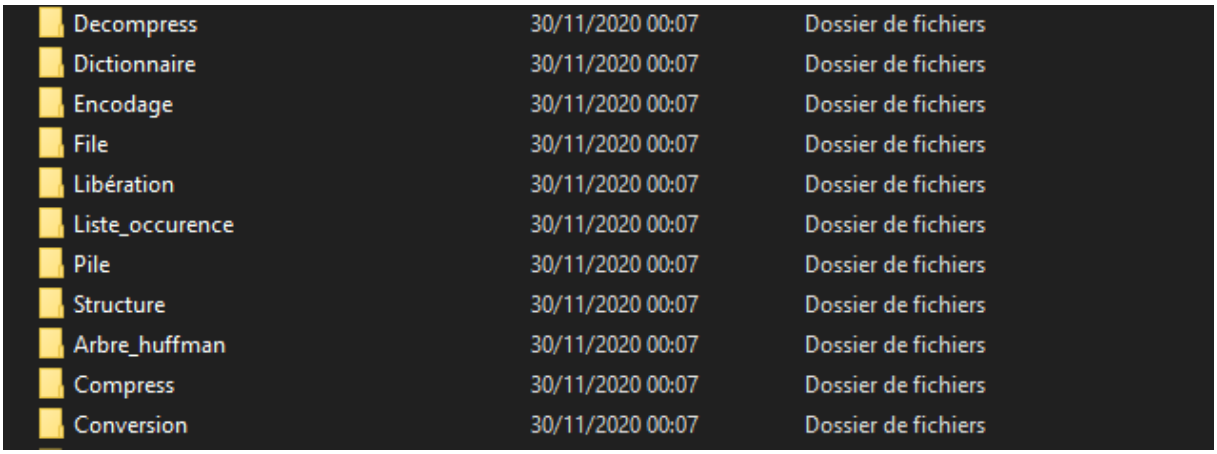
Projet Huffman



LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Description de l'architecture logicielle

Pour ce projet, nous avons décidé de définir un module pour chaque fonctionnalité importante. Il y en a 11 au total :



Les modules sont définis de cette manière :

Decompress :

Pour la partie Decompress on a séparé cela en 2 fonctions, une fonction qui permet de recréer l'arbre d'Huffman à partir du fichier dictionnaire (puisqu'on est censé pouvoir décompresser en dehors de la fonction de compression, on ne peut pas garder l'arbre d'Huffman en mémoire). Et une autre qui décompresse à l'aide de l'arbre créé précédemment. Pour utiliser les bonnes fonctions nous avons inclut les fonctions provenant de la partie Arbre Huffman pour pouvoir utiliser notamment la fonction qui crée un nœud, pour éviter de redéfinir cette fonction dans le .h de la partie Decompress. Bien évidemment, dans chaque .c nous incluons le .h correspondant à chaque partie (pour la partie décompression, nous incluons decompression.h dans le .c). Pour éviter les cycles de dépendances, nous incluons les autres .h dont nous avons besoin dans le .c.

Dictonnaire :

Nous avons 2 fonctions principales, celle qui crée le fichier dictionnaire et celle qui crée l'arbre (AVL) associé au dictionnaire. Pour l'arbre nous avons besoin de sous-fonctions permettant d'ajouter un nœud dans l'AVL, notamment celle qui ajoute un nœud dans un ABR et celle qui équilibre l'arbre. Pour le dictionnaire, nous avons besoin de des fonctions issues de la partie liste occurrence notamment pour avoir accès à la fonction liste_size. Nous incluons également la partie libération pour libérer les pointeurs utilisés et pour finir nous incluons les différentes structures utilisées comme les piles et les structures Node.

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Encodage :

Pour l'encodage, nous utilisons 2 fonctions, une qui va permettre de rechercher le caractère encodé dans l'AVL et une autre permettant d'écrire le code correspondant au caractère dans un fichier. Il n'a pas besoin de beaucoup d'inclusion puisque l'on n'utilise pas d'autres fonctions externes, seul l'inclusion des structures nous importent.

File :

Cette partie nous permet de créer l'arbre d'Huffman de façon optimisée, nous utilisons une nouvelle structure appelée ''Queue'' qui contient comme information un Nœud. Nous avons les fonctions de base d'une file, c'est-à-dire l'enfilement et le défilement.

Libération :

Pour ne pas avoir de fuite de mémoire, nous avons créé des fonctions qui libèrent des listes chaînées, des files, des piles et des arbres binaires. Pour chaque fonction on parcourt la structure de données et à chaque maillon on libère le pointeur donné.

Liste_occurence :

Cette partie, crée une liste qui va être triée au fur et à mesure du parcours du fichier texte que l'on souhaite compresser. Pour ce faire nous avons créé un algorithme de recherche dichotomique, nous avons donc besoin d'une fonction qui permet de détecter si le caractère que l'on a extrait du fichier existe ou non (en effet, on ne peut pas accéder facilement à un caractère d'une liste chaînée à partir de son indice dans la liste contrairement à un tableau), et un autre permettant d'insérer un élément dans la liste dans le cas où l'élément tiré du fichier texte n'apparaît pas dans la liste.

Pile :

Sa description ressemble à la partie File, à l'exception qu'elle est composée des fonctions empilement et dépilement.

Structure :

On a les principales structures utilisées dans le programme. La structure de la liste chaînée contenant chaque caractère ainsi que son nombre d'occurrence dans le fichier texte (cf. Liste_occurrence). On a de plus la structure de la liste chaînée qui contient les Nœuds et non plus le caractère associé à son occurrence, cette structure a été créée par soucis de simplicité pour créer l'arbre d'Huffman.
On a enfin les 2 structures de Nœuds : une permettant de créer l'arbre d'Huffman avec pour information le caractère associé à son occurrence et l'autre de créer le dictionnaire avec chaque caractère associé à son code.

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Arbre_huffman :

Pour cette partie-là nous avons besoin de différentes fonctions qui permettent de traduire des listes chainées d'éléments en une liste chainée de Nœuds pour pouvoir créer l'arbre plus facilement et une autre permettant de créer un Nœud évitant d'écrire à chaque fois le même code lorsqu'on va créer un nœud.
Pour optimiser la recherche du minimum on a utilisé deux files d'où l'inclusion de la partie file dans le .c de l'arbre d'Huffman.

Compress :

Cette fonction permet la compression du fichier et regroupe toutes les fonctions qui permettent la compression : la fonction qui crée la liste d'occurrence, celle qui trie cette liste en fonction des occurrences, celle qui crée l'arbre d'Huffman à partir de cette liste, celle qui crée le dictionnaire (sous forme d'AVL) et enfin celle qui permet l'encodage à partir de l'arbre dictionnaire. On a ensuite libéré en mémoire l'arbre dictionnaire puisque l'on n'en n'avait plus besoin. En effet, on va créer par la suite un fichier dictionnaire qui va pouvoir être transporté en mémoire pour la décompression.

Conversion :

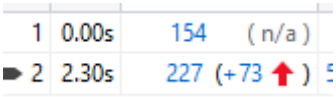
Cette partie avait principalement pour but de convertir le fichier texte en fichier binaire, pour cela on utilise une fonction qui va traduire un nombre en base 10 en une suite binaire.
On a également utilisé une fonction qui compte le nombre de caractères pour faire nos tests et pour voir si le fichier a bien été compressé.

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric

Principaux tests unitaires sur les fonctions essentielles

Fuites de mémoire :

Les fuites de mémoire, problème récurrent dans les programmes et souvent oubliées, peuvent créer des difficultés d'exécution et ainsi ralentir le programme, voire le faire planter.
Nous avons utilisé Visual Studio pour analyser la gestion de la mémoire au cours du programme. Pour cela, nous avons positionné deux breakpoints dans le main, un au début et un à la fin, et nous avons comparé la différence de mémoire allouée entre ces deux étapes. Après l'ajout de fonctions de libération et quelques optimisations, nous passons de 1097 éléments alloués en fin de programme à 73 après désallocation de mémoire. Le temps indiqué à gauche n'est pas représentatif car des changements ont été effectués entre-temps, augmentant le temps d'exécution.



Etat de la mémoire à l'origine, puis après une première puis une deuxième modification.

Warnings :

Toujours grâce à Visual Studio, nous avons encore plus optimisé le programme en résolvant le maximum de warnings possibles.

Pour ce faire, nous avons utilisé l'outil Build Solution qui nous a permis d'identifier un certain nombre de warnings, comme représentés ci-dessous :

```
'list_size' undefined; assuming extern returning int
warning C4013: 'binaire' undefined; assuming extern returning int
warning C4047: '=': 'int *' differs in levels of indirection from 'int'

: warning C4101: 'temp2': unreferenced local variable
```

Après résolution de ces divers warnings, nous arrivons finalement à cette solution :

LEFEVRE Noé
GOBILLIARD Arthur
SOK Michaël
WANG Eric


```
1>------ Build started: Project: Huffmab_opti, Configuration: Debug Win32 ------
1>Arbre_huffman.c
1>C:\Users\lefev\source\repos\Huffmab_opti\Arbre_huffman\Arbre_huffman.c(51,1): warning C4047: 'initializing': 'char' differs in levels of indirection from 'char [2]'
1>Compress.c
1>Conversion.c
1>Decompress.c
1>C:\Users\lefev\source\repos\Huffmab_opti\Decompress\Decompress.c(8,1): warning C4047: 'initializing': 'char' differs in levels of indirection from 'char [2]'
1>Dictionnaire.c
1>Encodage.c
1>File.c
1>Liberation.c
1>Liste_occurence.c
1>main.c
1>Pile.c
1>Generating Code...
1>Huffmab_opti.vcxproj -> C:\Users\lefev\source\repos\Huffmab_opti\Debug\Huffmab_opti.exe
1>Done building project "Huffmab_opti.vcxproj".
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Autres :

Le principal problème rencontré pour la compression a été le parcours de l'arbre d'Huffman pour pouvoir encoder chaque lettre. En effet, nous devions le parcourir intelligemment sans repasser par des feuilles déjà parcourues. Après avoir mûrement réfléchi à ce problème, nous avons décidé qu'à chaque parcours, nous allions supprimer la feuille qui vient d'être encodée pour éviter de la reparcourir. La logique de parcours a été explicitée dans la partie dictionnaire du rapport. De plus, pour la suppression des nœuds et non pas des feuilles, l'utilisation d'une pile a été décidée tardivement : nous utilisions les listes chaînées auparavant. Nous avons ainsi constaté que cette dernière pouvait remplir les conditions d'une pile.

Nous avions aussi rencontré des problèmes concernant la fonction de tri par dichotomie ; en effet nous avions du mal à comprendre la logique, surtout avec des listes chaînées (l'accès à un maillon avec un indice n'est pas aussi simple qu'avec un tableau classique). La fonction d'origine faisait plus de 100 lignes, nous l'avons raccourcie a à peine 50 lignes.

Enfin, pour la partie dictionnaire dans la partie optimisée, nous devions créer le dictionnaire sous forme de fichier et d'arbre. Nous avions créé ainsi deux fonctions différentes, une qui crée l'arbre et une qui crée le fichier et nous avons constaté que ces deux fonctions se ressemblaient fortement : pour factoriser le code et le rendre plus performant, nous avons décidé de rassembles ces deux fonctions en une seule.

Conclusion

Ce projet nous a apporté beaucoup de choses. Il nous a appris a bien nous coordonner dans nos tâches et faire une répartition optimisée du projet. Ce projet est d'autant plus intéressant car il nous permet d'appliquer concrètement ce que l'on a appris en cours d'Introduction au Génie Logiciel. Il nous a appris à concevoir un code modulaire et optimisé tout en améliorant nos méthodes de programmation. On a aussi appris a utiliser de nouveaux outils qui sont fort intéressant, par exemple GitHub. Avant pour partager nos projets, nous nous envoyions le projet en .zip sur Discord, cela posait déjà des problèmes, lorsqu'il y avait une modification, il fallait renvoyer tout le fichier. Cela était pire lorsque qu'il y avait des modifications provenant de plusieurs personnes. Grâce à GitHub, le partage du programme est beaucoup plus facile et cela règle aussi le problème des modifications multiples. Nous avons aussi appris à utiliser Doxygen. Ce logiciel est extrêmement utile pour nous car il nous a permis de commenter le code et de fournir un wiki de façon précise et très rapide.

La fusion des deux matières (Structure de Données et Introduction au Génie Logiciel) est quelque chose qui nous apporté beaucoup de compétence, cela nous a habitué à respecter un cahier des charges tout en prenant compte de l'utilisateur final. Cela nous prépare ainsi pour notre future vie professionnelle.

Références bibliographiques

Qualité logicielle — Wikipédia (wikipedia.org)
Architecture logicielle — Wikipédia (wikipedia.org)
Antipattern — Wikipédia (wikipedia.org)
Réinventer la roue — Wikipédia (wikipedia.org)
Duplication de code — Wikipédia (wikipedia.org)
Réinventer la roue carrée — Wikipédia (wikipedia.org)
Programmation par copier-coller — Wikipédia (wikipedia.org)
Test driven development — Wikipédia (wikipedia.org)
Snippet — Wikipédia (wikipedia.org)
Test de régression — Wikipédia (wikipedia.org)
Extreme programming — Wikipédia (wikipedia.org)
Réusinage de code — Wikipédia (wikipedia.org)
Réingénierie logicielle — Wikipédia (wikipedia.org)
Programmation par contrat — Wikipédia (wikipedia.org)
Fuite de mémoire — Wikipédia (wikipedia.org)
Analyse statique de programmes — Wikipédia (wikipedia.org)
Cppcheck — Wikipédia (wikipedia.org)
Code smell — Wikipédia (wikipedia.org)