Codage de Huffman

Une compression sans perte de données

Rapport de projet groupe EF08

Programmation Impérative

Étudiants :
Valentin Fontaine
Romain Peyremorte

Encadrante : Mme Nassima Djema

Année : 2021 - 2022

Nos remerciements à madame Nassima Djema, notre encadrante pour ce projet dont l'accompagnement et les conseils nous ont aiguillés tout au long du semestre et nous ont permis de porter le projet à son état actuel

Table des matières

Résumé

Dans le cadre du cours de Programmation Impérative du semestre 5 dans le département Sciences du Numérique de l'ENSEEIHT, nous avons réalisé un projet où nous devions réaliser un compresseur et un décompresseur de fichier texte. Le principe de compression suit le codage de Huffman qui consiste à attribuer un code à la donnée dont la longueur dépend de la fréquence d'apparition de la donnée. Cette méthode de compression est sans perte.

Nous avons réalisé ces programmes en langage de programmation Ada. Nous avons créé différents types de données ainsi que divers modules pour nous aider à la réalisation de la compression et de la décompression, après avoir écrit les raffinages des algorithmes que nous allions utiliser.

Ce projet nous a permis d'appliquer ce que l'on a vu en cours à travers un exemple concret et complet.

Introduction

Le codage de Huffman est un algorithme qui permet la compression de données sans perte. Il a été inventé en 1952 par David Albert Huffman. Son principe repose sur l'utilisation d'un code à longueur variable. En effet, les données à compresser sont d'abord divisées en symboles afin de dresser une table de fréquences. Par la suite, le codage va associer un code plus court aux symboles fréquents.

Nous nous intéresserons ici au codage de Huffman utilisant des octets comme symbole, soit les 256 symboles possibles de la table ASCII. Nous allons réaliser un programme exécutable qui compresse grâce au codage de Huffman un fichier ne texte contenant que des caractères codés en ASCII en un autre fichier, ainsi qu'un autre programme exécutable qui quant à lui décompresse le fichier réalisé par le premier programme pour recréer le fichier texte originel. Les programmes seront écrits en langage Ada, puis compilés pour donner un exécutable.

Cahier des charges

Le cahier des charges de notre projet est le suivant :

- Concevoir un compresseur qui prend en entrée un fichier et qui renvoie en sortie une version compressée de ce fichier selon la méthode de Huffman.
- Concevoir un décompresseur capable de reconstruire le fichier originel à partir du fichier créé par le compresseur
- Mettre en place une interface utilisateur en ligne de commande pour rendre conviviale l'utilisation de nos deux programmes.

Conception de l'application

Les différents types de données

Afin de modéliser les données du problème, nous avons été amené à créer un certain nombre de nouveaux types en plus de ceux déjà proposés par défaut par Ada. Nous n'avons cependant pas négligé l'utilisation de ces types déjà présents dans le langage, qui s'avèrent très pratiques car il y a un grand nombre de fonctions et procédures pour les manipuler. Voici les types les plus importants définis par Ada lui-même :

- les entiers (`Integer`)
- les chaînes de caractères (`String`) et les chaînes de caractères à taille variable (`Unbounded_String`)
- les types pour manipuler (créer, lire, écrire) un fichier (`File_Type`, `Stream_Access`)

Par la suite, nous avons modélisé nos propres types pour faciliter l'implémentation des algorithmes :

- les arbres (`T_Arbre`) pour nous permettre d'implémenter l'arbre de Huffman dans nos programmes. Un arbre tel que nous l'avons défini est un pointeur vers un nœud, disposant d'une clé, d'une valeur, d'un fils droit et d'un fils gauche. Les feuilles de l'arbre d'Huffman se différencient uniquement en ayant une Valeur non nulle et en ayant aucun fils. Ce type est la solution que nous avons choisi pour représenter l'arbre de Huffman
- les listes chaînées (`T_Liste_Chainee`) pour avoir une structure de données dynamique et de longueur variable.
- les bits (`T_Bit`) qui n'ont que 2 valeurs possibles (0 ou 1) et qui sont de taille 1.
- les octets (`T_Octet`) pour représenter les 256 valeurs possibles d'un octet. Un octet est de taille 8.
- les flux binaires (`T_Flux_Binaire`). Ce type est probablement le plus important et le plus utile pour notre application. Il permet une manipulation aisée des bits et octets. C'est avec ce type que l'on construit les codes des symboles.
- les flux de fichiers (`T_Flux_Fichier`) qui sont une surcouche du type `Stream_Access` de Ada. Ces flux particuliers permettent de réduire la lecture d'un fichier bit après bit et non octet après octet, ce qui s'avère particulièrement utile lors de la décompression.

Nous avons également défini nos propres exceptions afin d'avoir une uniformité dans les noms des erreurs. Par exemple pour l'interface utilisateur, nous avons trois nouvelles définitions : `Trop_Arguments_Exception`, `Peu_Arguments_Exception`, `Option_Inconnue_Exception`.

Une définition plus formelle de nos types est disponible en *Annexe A*.

Principaux choix réalisés

Pour modéliser les codes des octets, nous avons choisi d'utiliser notre type `T_Flux_Fichier`, qui est tout à fait adapté car il permet de manipuler facilement des suites binaires. De plus, les flux étant dynamiques, leur taille n'est pas bornée donc nous pouvons effectivement les utiliser pour représenter des codes huffman plus grands que 8 bits (ce qui arrive lorsqu'il y a beaucoup de caractères différents, par exemple un fichier contenant un de chaque, de l'octet 0 à 255).

Pour stocker les codes des différents octets présents dans un fichier, nous utilisons un simple tableau de 256 `T_Flux_Fichier`. Cependant ce n'est pas un tableau classique car il commence à zéro pour avoir une correspondance parfaite entre l'indice dans le tableau et la valeur en ascii. En effet nous avons préféré définir un alias `Intervalle_Ascii` qui va de 0 à 255 et cacher le 0 derrière, plutôt de faire des +1 à chaque fois, ce qui est source d'erreurs.

Nous avons défini le symbole de fin fichier comme un octet possédant la valeur 255, soit le dernier octet de l'Ascii. Cela permet de le manipuler simplement, comme s'il n'était pas un "imaginaire" en dehors de table ASCII. Cela nécessite toutefois de faire attention lorsque l'octet 255 est également présent dans le fichier. C'est en utilisant judicieusement l'attribut clé d'un nœud de l'arbre que nous arrivons toujours à différencier le symbole de fin du vrai symbole 255. En effet lors de la compression le symbole de fin est le seul avec un

clé (fréquence) à zéro. Pour la décompression c'est le même principe, il sera le seul à avoir une clé valant zéro, tous les autres octets ayant un clé valant un pour signifier qu'il ne pas spéciaux.

L'architecture modulaire

Pour pouvoir manipuler les principaux types de données introduits précédemment, nous avons réalisé des modules, parfois génériques, contenant les procédures et fonctions permettant leurs utilisations.

Module `Arbre`

Ce module générique a été créé principalement pour pouvoir gérer l'Arbre d'Huffman que nous devions générer pour coder le texte. Le type d'élément (Clé et Valeur) qui se trouve dans chaque type constitue le principal élément générique. Le choix de laisser le type T_Arbre du module en non-privée résulte de difficultés rencontrées.
Le module contient les procédures et fonctions usuelles pour initialiser un arbre (un premier nœud), savoir si un arbre est vide et vider un arbre. Il contient aussi une procédure pour enregistrer un arbre (ou nœud) avec ces différentes caractéristiques (Clé, Valeur, Arbre pointé à Gauche, Arbre pointé à Droite) ainsi qu'une procédure générique pour pouvoir appliquer une procédure sur chaque élément (Clé et Valeur) enregistré dans chaque nœud.

Module `Liste_Chainee`

Ce module générique permet la gestion d'une liste dynamique de n'importe quel type. Il contient des procédures classiques permettant d'initialiser une liste, de savoir sa taille, si elle est vide. Il dispose également de fonctions pour obtenir le n-ième élément, l'indice d'un certain élément ou savoir si un élément est présent ou non. Enfin ce module propose des procédures pour ajouter et supprimer un élément, ainsi que l'insertion avant un certain indice et vidage de la liste, c'est-à-dire la suppression de tous les éléments.

Module `Flux_Binaire`

Ce module est un des piliers de notre application. Il abstrait les opérations binaires afin de les rendre simples d'utilisation grâce aux différents services proposés. Nous pouvons ainsi aisément accéder au n-ième octet du flux ou même au k-ième bit de ce n-ième octet. Il permet bien entendu l'ajout d'un octet ou d'un bit au flux mais également d'ajouter un flux à un autre flux ou de vider totalement un flux.

Cette dernière opération s'avère très pratique pour construire le futur fichier compressé. En effet nous représentons d'abord ce fichier avec un flux, et ajoutons successivement le code de chaque symbole du fichier original, chaque code étant lui-même un flux binaire.

Une procédure particulièrement utile est `Pour_Chaque`, qui permet de lancer un traitement sur chaque octet du flux. C'est avec cette procédure que nous enregistrons le fichier compressé, en enregistrant dans un fichier les octets les un après les autres.
L'implémentation d'un flux binaire se fait à l'aide d'une liste chaînée d'octets.

Enfin, ce module met à disposition une fonction pour convertir un flux en chaîne de caractères, pour pouvoir faire des affichages de flux en toute simplicité.

Module `Types_Huffman`

Il regroupe les types *ad hoc* de notre application, dans le sens où ils sont si précis qu'il n'y a pas d'intérêt de les mettre chacun dans un module car leur réutilisabilité pour un autre problème que la compression de huffman est assez limité. Ce module est utilisé par le module `Interface_Utilisateur` et les programmes `compression` et `decompression`. Il permet ainsi de centraliser les types et évite donc toute redondance dans leur définition.

Pour donner quelques exemples des types présents, nous avons le sous-type `Intervalle_Ascii` qui est un intervalle allant de 0 à 255, soit l'ensemble des valeurs du type `T_Octet`. Il est utilisé pour avoir plus de sémantique là où c'est nécessaire. En effet l'instruction `for I in Intervalle_Ascii loop` est comprise plus facilement que l'instruction `for I in 0 .. 255 loop`, dans laquelle 0 et 255 semblent être des nombres magiques sans un contexte préalable.

C'est également dans ce module que nous définissons les instances de nos modules générique. Nous y retrouvons notamment les modules `Arbre_Integer_Octet` et `Liste_Chainee_Octet` utilisés respectivement dans la création de l'arbre de Huffman et la liste des positions des symboles dans l'arbre.

Module `Interface_Utilisateur`

Ce module fait le lien entre notre implémentation et l'utilisateur. En effet, c'est dans ce module que nous définissons les différentes procédures pour récupérer les informations en ligne de commandes grâce à `Recuperer_Infos_Utilisateur`. C'est par cette procédure que nous passons pour savoir si l'utilisateur a rentré assez d'arguments (au moins le nom du fichier), trop (plus que 2), pas assez (aucun) ou simplement des arguments incorrects (l'option "-c" par exemple, qui est inconnue).

Cette interface met également à disposition divers sous-programmes pour afficher simplement ce qui est nécessaire. Nous retrouvons par exemple la procédure `Afficher_Arbre` qui affiche joliment un arbre représenté par un `T_Arbre`.

Programmes `Compression` et `Decompression`

Nous avons choisi de ne pas faire de module représentant la compression et la décompression, qui peuvent être par définition des modules. En effet, le principe d'un module est de regrouper des éléments qui ont des liens entre eux et de les encapsuler pour

former un ensemble de types et de services (sous-programmes manipulant les types définis) réutilisable ultérieurement et par plusieurs autres programmes. Or ce qui est attendu est un

Ainsi `Compression` et `Decompression` sont des programmes principaux qui utilisent des modules mais ne sont pas des modules eux-mêmes, car ils ont été conçus comme tels. En effet, ils prennent en entrée des arguments passés en ligne de commande et donnent en sortie un fichier : ils sont faits pour être lancés par un utilisateur et non par une procédure.

Pour devenir des modules, il faudrait quelque peu modifier ces entrées et ces sorties pour qu'elles soient plus facilement exploitables par d'autres fonctions et procédures. Une nouvelle approche permettant la modularité de nos programmes compression et décompression est discutée plus tard dans les perspectives et améliorations futures de ce rapport.

Réalisation et Implémentation

La démarche adoptée

La première étape du projet a été de s'imprégner du sujet, de bien comprendre le fonctionnement de la compression et de la décompression. Une fois le cadre bien défini, nous avons débuté la conception de l'application en utilisant la méthode des raffinages. En effet, il est important de bien spécifier les types et sous-programmes principaux avant de démarrer l'implémentation. Cela permet d'avoir une vision d'ensemble sur l'application pour trouver et résoudre plus facilement les potentielles erreurs, car plus un problème est identifié en amont plus la solution sera simple à mettre en œuvre.

Une fois la conception achevée, nous avons commencé la réalisation des spécifications définies par les raffinages en utilisant le langage Ada. Deux démarches ont été adoptées :

- Pour les modules concernant nos types de données, nous avons écrit les interfaces, puis rédigé les tests associés, pour finir avec l'implémentation des fonctions et procédures jusqu'au passage intégral des tests.
- Pour les sous-programmes qui implémentent les algorithmes de compression et décompression, notre approche a été légèrement différente. Nous n'avons pas créé de procédures de tests mais plutôt un fichier exemple à compresser. Ensuite, à chaque nouveau sous-programme terminé, nous avons exécuté notre code pour vérifier qu'il implémentait bien la partie le concernant. Par exemple pour la compression, notre démarche itérative a été la suivante : implémenter la procédure `Determiner_Frequence`, vérifier que les fréquences obtenues étaient les bonnes (avec l'aide d'affichages), implémenter `Creer_Arbre`, vérifier que l'arbre est correct, et ainsi de suite jusqu'à la fin de l'implémentation complète de la compression. La vérification de chaque étape s'est faite en affichant le résultat de l'étape avec les procédures d'affichages qui ont été regroupées dans le module `Interface_Utilisateur`.

Lorsque l'implémentation selon les raffinages fut terminée, nous avons analysé notre code pour voir quelles parties étaient redondantes. Ces redondances ont ensuite été regroupées en modules pour éliminer les doublons. C'est ainsi que les modules `Interface_Utilisateur` et `Types_Huffman` ont été créés. La dernière étape du projet a été de lancer des tests valgrind pour vérifier qu'il n'y avait pas de fuite de mémoire. Les rapports valgrind des différents exécutables sont disponibles en *Annexe D*.

Les principaux algorithmes

Les raffinages de chacun des deux principaux programmes se trouvent en *Annexe B* pour la compression et en *Annexe C* pour la décompression.

Compresser

Les principales parties de la compression consiste à récupérer la fréquence de chaque caractère du texte, puis de créer chaque feuille de l'arbre de Huffman. Ensuite il faut construire l'arbre et en déduire le codage de chaque caractère en créant la table de Huffman. Enfin, il faut coder l'arbre en récupérant la position du code de fin ainsi que l'ordre de chaque caractère dans l'arbre, mais aussi la structure des nœuds des feuilles.

Récupérer les fréquences des caractères

Pour pouvoir récupérer et gérer les fréquences de caractères, nous avons choisi d'utiliser un tableau d'entier dont la taille est le nombre de caractères possibles (256). Ainsi, l'indice dans le tableau permet de récupérer directement l'information sur le caractère, et l'entier la fréquence. Donc pour récupérer les fréquences, on parcourt l'ensemble des octets du texte, et à l'indice de l'octet où l'on est, on augmente de 1 la fréquence. Ainsi les caractères du texte qui devront apparaître dans l'arbre de Huffman sont ceux qui ont une fréquence non nulle.

Créer les différentes feuilles de l'arbre

Pour permettre représenter l'arbre, nous avons créé le type `T_Arbre`, mais il est principalement utile pour l'arbre de Huffman fini, pas lorsque nous avons les feuilles non rassemblées. Pour cela nous avons créé un tableau qui peut contenir 257 feuilles possibles (les 256 caractères ASCII plus l'élément de fin de fichier) dont on connaît la taille dynamique (via un enregistrement). On crée donc ce tableau de feuille (qui ne sont que des arbres pointants vers rien) en parcourant le tableau de fréquence et en enregistrant les caractères de fréquences non nulles (la clé étant leur fréquence et la valeur de leur code ASCII en Octet). Ensuite, on enregistre le symbole de fin de texte dans une feuille. Ainsi on a notre tableau de feuilles dont on sait combien il y a d'arbres (feuilles) en cours dedans.

Créer l'arbre de Huffman à partir des feuilles

Pour réaliser l'arbre de Huffman, nous appliquons l'algorithme qui consiste à rassembler les deux nœuds de plus faibles fréquences sous un autre nœud qui aura comme clé la somme des clés des deux nœuds. Pour obtenir ces deux plus petits nœuds en, nous

trions le tableau de nœuds par ordre décroissant par rapport à leur clé. Ainsi le dernier à la plus petite clé, et l'avant dernier, la deuxième plus petite. On crée donc un nouveau nœud pointant sur les deux derniers. On place ce nouveau nœud à l'avant dernière place du tableau et nous réduisons la taille du tableau de 1, ainsi les deux anciens plus petits nœuds ne sont plus dans le tableau de nœud. Pour trier le tableau, nous avons choisi d'utiliser la méthode du tri par sélection que nous trouvions la plus facile à implémenter en Ada et bien adaptée à notre cas. Nous répétons le processus jusqu'à avoir un seul élément dans le tableau : ce dernier élément est notre arbre de Huffman.

Récupérer le code de chaque caractère

Après avoir créé l'arbre, il nous faut maintenant en extraire les différents codes des caractères. Pour cela nous avons un tableau de `T_Flux_Binaire` pour contenir les codes de chaque caractère en ayant l'indice indiquant le caractère en octet, et le `T_Flux_Binaire` le code correspondant, ainsi qu'une variable spécialement déclarée pour le symbole de fin de fichier.
Pour pouvoir remplir ces deux variables, on parcourt l'arbre :

- lorsque l'on est à un noeud et pas une feuille (c'est-à-dire que ce noeud a de fils), on ajoute au code un 0 pour la suite du parcours vers la Gauche du noeud, et un 1 pour la suite vers la Droite
- lorsque l'on arrive à une feuille (pas de fils), on récupère le code que l'on a construit au fur et à mesure et on l'ajoute à la variable correspondante : soit le code pour le symbole de fin, soit le tableau de codage pour un caractère (la différenciation entre les deux se fait au niveau de la clé car le symbole de fin est le seul à avoir une clé nulle)

Le code généré est en `T_Flux_Binaire` qui est le type le plus adapté selon nous à la représentation d'un code d'un caractère. Ainsi pour représenter la table de Huffman, on a un tableau avec les codes pour les caractères présents dans le textes ainsi que le code pour le symbole de fin.

Coder la structure de l'arbre

Il est important de préciser que le symbole de fin est le seul à avoir une fréquence nulle parmi toutes les feuilles de l'arbre de Huffman. Or la création de l'arbre est de telle sorte que lors de la création d'un nouveau nœud dans l'arbre, le sous-arbre de gauche est celui dont la fréquence est la plus faible. On en déduit que la feuille du symbole de fin est obligatoirement à gauche d'un nœud. Lorsque l'on a dû créer la liste des positions des symboles de l'arbre, on a vu le risque d'avoir une liste de 257 éléments (tous les 256 caractères plus le symbole de fin), et donc lorsqu'il aurait fallu mémoriser la position du symbole de fin de fichier, on s'est demandé si cette position pouvait être la 257ème et donc ne pas pouvoir être codé sur 1 octet (8 bits). Cependant, le 257ème élément du tableau de Huffman dans notre cas de lecture de gauche à droite, est celui le plus à droite, c'est-à-dire qu'il est à la droite d'un nœud et que tous ces nœuds parents sont eux-même à la droite d'un autre. Or on a indiqué plutôt que le symbole de fin est obligatoirement, dès le début de la création de l'arbre par le bas, à gauche d'un nœud. Nous en avons donc déduit que la position du symbole de fin ne peut être 257 et ainsi elle peut être codé sur 1 octet.

Pour pouvoir récupérer la position du symbole de fin, l'ordre des caractères et la structure des nœuds et des feuilles, nous parcourons une nouvelle fois l'arbre. Cette fois-ci :

- Lorsque l'on est à un nœud, on enregistre juste un 0 dans le `T_Flux_Binaire` qui code la structure entre nœuds et feuilles
- Lorsque l'on est à une feuille, on enregistre un 1 dans le `T_Flux_Binaire` qui code l'ordre entre nœuds et feuilles. Cette feuille est soit celle du symbole de fin, dans ce cas on récupère juste sa position, soit c'est une feuille d'un caractère donc on ajoute l'octet représentant le caractère à la liste des positions.

Ainsi, on a les différents éléments permettant de coder le texte et créer un fichier en remplaçant chacun des caractères par son code de Huffman associé.

Décompresser

Pour synthétiser, le fichier compressé est construit comme suit :

- Le premier octet est la position du symbole de fin dans l'arbre (par parcours infixe)
- Une suite d'octets représentant les symboles présents dans le fichier originel, également dans leur ordre infixe. Le dernier octet est doublé pour signifier la fin de la liste.
- Une suite de 0 et de 1 modélisant la structure de l'arbre de Huffman. Un 0 signifie que c'est un noeud, et un 1 indique une feuille
- Le reste des octets du fichier sont les données compressées du fichier. C'est la concaténation de tous les codes des symboles du fichier originel. Autrement dit, chaque octet du fichier de base a été remplacé par son code dans le fichier compressé.

Décoder la liste des positions

La première étape de la compression consiste à retrouver la liste des symboles. Cette partie est assez simple. En effet, il suffit de lire un octet pour avoir la position du symbole de fin, puis lire successivement tous les octets suivants jusqu'à tomber sur un doublon. Enfin, nous insérons dans la liste le symbole le fin, qui comme indiqué précédemment a pour valeur arbitraire 255. Ainsi l'indice d'un symbole dans la liste représente sa position dans l'arbre. Pour l'implémentation nous avons choisi de stocker cette suite de symboles dans une liste chaînée d'octets.

Reconstruire l'arbre de Huffman

La seconde étape est le décodage de la structure de l'arbre. Nous commençons par lire le fichier bit après bit et non plus octet après octet. Si le bit est 0 alors cela signifie qu'il y a un nœud, donc nous créons un nœud intermédiaire et appelons récursivement la procédure sur le sous-arbre gauche et droit ce de nouveau nœud. Si le bit est 1 alors nous sommes sur une feuille. Un compteur nous permet de savoir que c'est la n-ième feuille. Nous enregistrons ensuite la n-ième valeur de la liste des positions dans cette feuille. Nous ajoutons également à la clé de la feuille 0 ou 1 pour dire s'il s'agit d'un symbole classique ou

non. Cela nous permettra plus tard de différencier le symbole de fin du symbole 255 s'il est lui aussi présent dans l'arbre.

Reconstruire le fichier originel

Une fois l'arbre de Huffman reconstruit, le fichier originel s'obtient sans trop de difficulté. Nous continuons notre lecture bit à bit du fichier en la synchronisant avec un parcours de notre arbre. Ainsi, nous descendons à gauche de l'arbre si nous lisons un 0 et descendons à droite si c'est un 1. Une fois arrivé sur une feuille, nous regardons la clé associée à cette feuille. Si c'est un 1 alors le symbole est un symbole classique nous écrivons l'octet de ce symbole (qui est la valeur de la feuille) dans le fichier décompressé. La décompression se termine lorsqu'une feuille à pour clé 0, car cela signifie que nous sommes au symbole de fin et tous les octets originaux ont été décodés.

Gestion du projet

Organisation et planification

Comme énoncé précédemment, nous avons commencé par réfléchir sur le problème chacun de notre côté pour développer chacun notre propre point de vue. Nous nous sommes ensuite concertés pour mettre au clair les idées et répondre aux questions que chacun se posait de son côté. Une fois ce travail préliminaire terminé, nous avons entamé la rédaction des raffinages ensemble.

Concernant la répartition de la charge de travail, c'est Valentin qui s'est occupé de terminer les premiers raffinages de la compression et décompression, car certaines parties du projet n'étaient pas encore totalement comprises par Romain, notamment pour la compression. À propos des différents modules, Romain a réalisé le module `Arbre`, tandis que Valentin s'est occupé des modules `Liste_Chainee` et `Flux_Binaire`. Par la suite Valentin a débuté le programme de compression avant de laisser Romain le terminer en grande partie. La décompression a été implémentée par Valentin. Il y a également les divers affichages, l'interface utilisateur en ligne de commande et la gestion des exceptions que nous avons réalisés ensemble. Enfin, Romain s'est occupé de reprendre les raffinages.

Difficultés et changements majeurs

Lors de la création du module de l'arbre et de ses tests, nous avons rencontré des difficultés sur le moyen de parcourir l'arbre et de modifier et accéder à ces composants, c'est pourquoi nous sommes passés d'un type privée à un type non privé après le premier rendu de ce module.

Après réalisation des programmes, nous avons modifié les raffinages pour qu'ils collent aux changements que nous avons dû faire lors de la programmation après rédaction des modules génériques. En effet, des types se sont affinés, des solutions différentes ont

été appliquées, ce qui fait que les raffinages actuels sont différents des raffinages déposés lors du premier rendu.

Bilan et Perspectives

En conclusion, nous sommes plutôt satisfait du programme final car nous jugeons que le cahier des charges a été respecté. Le bilan technique est positif, dans le sens où la compression et la décompression fonctionne sur tous nos fichiers de tests. Ainsi nos programmes sont pour le moment capables de compresser et décompresser le fichier d'exemple du sujet, un fichier contenant les 256 octets possible. Le fichier vide est également pris en charge. Nous n'avons pas identifié de fuites de mémoire, tous nos tests Valgrind sont passés avec succès.
Nous pensons également avoir bien réparti le travail entre nous, de manière équitable. Ce projet a été très enrichissant car il nous a permis d'appliquer les enseignements que nous avons reçus dans le module de Programmation Impérative de ce semestre.

Effectivement, nous avons été amenés à concevoir une application de zéro à partir d'un problème posé. Nous avons trouvé la démarche très intéressante, à savoir commencer par la compréhension du problème et de ce qui est demandé. Puis réfléchir ensemble à une solution en élaborant nos propres sous-programmes grâce à la méthode des raffinages. Vient ensuite la réalisation avec l'implémentation des spécifications.

Nous avons bien apprécié cette dernière partie car nous avons pu mettre en œuvre l'ensemble de nos compétences apprises durant les travaux pratiques et les mini-projets dans le cadre d'un vrai projet, ce qui est une expérience plutôt exaltante. Pour finir, nous sommes convaincus que le savoir-faire acquis durant ce projet, notamment cette démarche en trois étapes (compréhension, conception et réalisation) nous aidera dans nos futurs projets.

Bilan Valentin

Ce projet a été une belle expérience. J'ai bien apprécié le travail avec mon binôme, nous ne connaissions pas avant mais nous avons réussi à bien nous entendre. Concernant les compétences acquises, j'ai pu mettre en application mes connaissances sur les opérations binaires (acquises grâce aux cours d'architecture des ordinateurs) avec le développement du module des flux binaires. J'ai trouvé l'exercice bien encadré, avec un sujet très bien présenté en donnant un certain nombre de détails, sans toutefois tout indiquer, pour laisser libre cours à notre réflexion et nos capacités techniques. J'ai hâte de débuter de nouveaux projets aussi intéressant que celui-ci !

Bilan Romain

Le projet était intéressant car il nous a permis d'appliquer presque l'ensemble du cours. Ce fut intéressant de travailler avec quelqu'un d'autre pour découvrir de nouvelles méthodes ainsi qu'un nouveau point de vue. Ce fut enrichissant autant sur le plan technique, mais aussi sur le plan social.

Je regrette seulement les premiers rendus dont je trouve qu'ils ont un délais de rendus trop courts, surtout pour un élément aussi important que les raffinages.

Perspectives

Nous avons pensé à certaines fonctionnalités additionnelles qui n'ont pas pu être réalisées avant la fin du projet. La première serait de transformer les programmes de compression et décompression en modules. Il faudrait pour cela modifier les programmes pour qu'ils prennent en entrée un flux binaire et rendent également en sortie un flux binaire. Ainsi il serait possible d'appeler la compression et la décompression dans un projet où il y aurait besoin de compresser des données.

Un ajout intéressant pourrait être de faire des fonctions pour supprimer un bit ou octet d'un flux binaire. Elles n'ont pas été implémentées car non nécessaires pour notre utilisation, cependant les avoir rendrait le module encore plus utile et réutilisable par d'autres programmes.

Nous aurions pu aussi plus développer l'interface utilisateur en lui proposant d'autres options comme par exemple permettre la décompression dans un fichier qui aurait un nom différent que celui d'origine.

Enfin, concernant la compression de Huffman en elle-même, nous pourrions modifier légèrement l'algorithme et supprimer toute la logique derrière le symbole de fin '\$', et simplement rajouter sur 3 bits au début du fichier le nombre de bits inutilisés dans le dernier octet (0 à 7 bits). Cela simplifierait quelques programmes qui n'auraient plus à gérer un caractère qui n'est pas dans la table ASCII.

Annexes

Annexe A : Types de données

Module Arbre

```
type T_Arbre est pointeur sur Noeud;

type Noeud est enregistrement
        Cle : Entier
        Valeur : T_Octet
        Gauche : T_Arbre
        Droite : T_Arbre
fin enregistrement
```

Module Liste Chaînée

```
type T_Liste_Chainee est pointeur T_Cellue

type T_Cellule est enregistrement
        Valeur : T_Octet
fin enregistrement
```

Module Flux Binaire

```
type T_Bit est modulo 2

type T_Octet est modulo 256

module Liste_Octet est nouveau Liste_Chainee (T_Valeur => T_Octet)

type T_Flux_Binaire est enregistrement
        Nombre_Bits_Inutilises : Entier
        Liste : Liste_Octet.T_Liste_Chainee
fin enregistrement
```

Module Types Huffman

```
Symbole_Fin : constant T_Octet := T_Octet'Last;

Fichier_Inexistant_Exception : Exception;
```

```
Arbre_Huffman_Vide : Exception;
Pas_Compression_Huffman : Exception;

module Arbre_Integer_Octet est nouveau Arbre (T_Cle => Entier, T_Valeur => T_Octet)

module Liste_Chainee_Octet est nouveau Liste_Chainee (T_Valeur => T_Octet)

sous-type Intervalle_Ascii est T_Octet entre T_Octet'First .. T_Octet'Last

type T_Frequences est tableau de (Intervalle_Ascii) de Entier

type T_Liste est tableau de (1..257) de  Arbre_Integer_Octet.T_Arbre

type T_Tableau est enregistrement
        Tableau : T_Liste
        Taille : Entier
fin enregistrement

type T_Codage est tableau (Intervalle_Ascii) de T_FluxBinaire;

type T_Flux_Fichier est enregistrement
        Flux_Interne : Stream_Access
        Octet : T_Octet
        Indice_Bit : Entier
fin enregistrement
```

Annexe B : Raffinage de Compresser

R0 : Compresser un fichier txt avec l'arbre d'Huffman

Exemples :

./compresser exemple.txt => exemple.txt.hff

./compresser -b exemple.txt => afficher les fréquences des caractères
afficher l'arbre d'Huffman
afficher le code des caractères
exemple.txt.hff

./compresser –bavard exemple.txt => afficher les fréquences des caractères
afficher l'arbre d'Huffman
afficher le code des caractères
exemple.txt.hff

./compresser exmple.txt => "Fichier à compresser inexistant."

./compresser -a exemple.txt => "Option entrée non reconnue."

./compresser => "Vous n'avez pas à entré d'arguments."

./compresser a b c => "Vous avez entré trop d'arguments"

R1 : Comment "Compresser un fichier txt avec l'arbre d'Huffman" ?

Déterminer Nom_Fichier avec les arguments de la ligne de commande
Affichage : out Booléen,
Nom_Fichier : out Chaîne_caractère

Déterminer les fréquences des caractères du texte
Nom_Fichier : in,
Tableau_frequence : out T_Frequences

Créer arbre
Tableau_Frequence : in ;
List_Char : out T_Arbre

Déterminer les codes des caractères
List_Char : in ;
Codage : out T_Codage

Encoder l'arbre de Huffman
List_Char : in,
Arbre_Encode : out T_FluxBinaire

Encoder le fichier grâce à l'arbre
Nom_Fichier : in Chaine, Codage : in
Arbre_Encode : in

Afficher les différentes étapes
Affichage : in, Tableau_frequence : in,
List_Char : in, Codage : in

R2 : Comment "Déterminer Nom_Fichier avec les arguments de la ligne de commande" ?

```
Affichage <- False
SI Argument_Count > 2 ALORS
    LEVER Trop_Arguments_Exception
SINON SI Argument_Count = 1 ALORS
    Nom_Fichier <- Argument(1)
SINON SI Argument_Count = 0 ALORS
```

```
        LEVER Peu_Arguements_Exception
    SINON
        Lire le double argument      Affichage  : out Booléen,
                                     Nom_Fichier : out Chaîne_caractère
    FIN SI
```

R2 : Comment “Déterminer les fréquences des caractères du texte” ?

```
    Initialiser les cases de Tableau_Frequence à 0      Tableau_Frequence : out
    Ouvrir le fichier                                   Nom_Ficher : in ;
                                                        Fichier : out,
                                                        Flux_Fichier : out
    TANT QUE non Fin_Du_Fichier (Nom_Fichier) FAIRE
        Lire octet                          Nom_Texte : in ; Octet : out T_Octet
        Mettre à jour la fréquence de Octet     Octet : in ;
                                                Tableau_Frequence : in out
    FIN TANT QUE
```

R2 : Comment “Créer arbre” ?

```
    Initialiser un tableau                  List_Char : out T_Tableau
    Créer la liste de noeud                 Tableau_frequence : in ;
                                            List_Char : in out
    Trier la liste de noeud de manière décroissante     List_Char : in out
    TANT QUE List_Char.Taille > 1 FAIRE
        Créer noeud intermédiaire                       List_Char : in out
        Trier la liste de noeud de manière décroissante List_Char : in out
    FIN TANT QUE
```

R2 : Comment “Déterminer les codes des caractères” ?

```
    Initialiser                             Code_vide : out T_Flux_Binaire
    Avoir le code de l’arbre                List_Char .Tablea(1) : in T_Arbre,
                                            Code_vide : in ;
                                            Codage : out T_Codage
```

R2 : Comment “Encoder l’arbre de Huffman” ?

```
    Initialiser                             Liste_Position : out Liste_Chainee
    Initialiser                             Code_Structure_Arbre : out T_Flux_Binaire
    Initialiser                             Fichier_Encode : out T_Flux_Binaire
    Déterminer la liste des positions des caractères dans l’arbre
                                            List_Char.Tableau(1) : in ;
                                            Liste_Position : in out T_Liste ;
                                            Code_Structure_Arbre : in out
    Encoder la liste des positions          Liste_Position : in, Arbre_Encode : out
```

R2 : Comment “Encoder le fichier grâce à l’arbre” ?

Ouvrir le fichier — Nom_Ficher : in ;
Fichier : out,
Flux_Fichier : out

Nom_Compresse <- Nom_Fichier + “.hff”
Créer le fichier Nom_Compresse — Nom_Fichier : in Chaine
Ouvrir le fichier Nom_Fichier
Ecrire (Nom_Compresse, Arbre_Encode)
Encoder les octets de Nom_Fichier dans Nom_Compresse
Nom_Fichier, Nom_Compresse : in

R2 : Comment “Afficher les différentes étapes” ?
Afficher les fréquences de chaque caractères — Tableau_frequences : in
Afficher l’arbre d’Huffman — List_Char : in
Afficher les codes de chaques caractères — Codage : in

R3 : Comment “Lire le double argument” ?

```
SI Argument(1) = “-b” OU Argument(1) = “--bavard” ALORS
    Affichage <- True
    Nom_Ficher <- Argument(2)
SINON
    LEVER Option_Inconnue_Exception
FIN SI
```

R3 : Comment “Initialiser les cases de Tableau_Frequence à 0” ?

```
POUR I DE 1 À 256 FAIRE
    Tableau_Frequence(I) <- 0
FIN POUR
```

R3 : Comment “Ouvrir le fichier” ?

```
Ouvrir (Fichier, In_File, Nom_Fichier)
Flux_Fichier <- Stream(Fichier)
Exception
    Quand Nom_Fichier non trouver => LEVER Fichier_Inexistant_Exception
```

R3 : Comment “Mettre à jour la fréquence de Octet” ?

```
Tableau_Frequence(Entier(Octet) +1) <- Tableau_Frequence(Entier(Octet) +1) + 1
```

R3 : Comment “Initialiser un tableau” ?

```
Entrée :    Tableau : out T_Tableau

    Tableau.Taille <- 0
```

R3 : Comment “Créer la liste de nœuds” ?

```
POUR i = 1..256 FAIRE
    Ajouter les caractères de fréquences non nulles    Tableau_Frequence : in,
                                                       i : in T_Octet ;
```

```
                                                        List_Char : in out
    FIN POUR
    List_Char.Taille <- List_char.Taille + 1
    Créer un nouveau noeud      0 : in Entier,
                                T_Octet'Last : in T_Octet,
                                null : in T_Arbre,
                                null : in T_Arbre ;
                                List_char.Tableau(List_char.Taille) : out T_Arbre

R3 : Comment "Créer un noeud intermédiaire" ?
    Plus_Faible_1 <- List_char.Tableau[List_char.Taille]
    Plus_Faible_2 <- List_char.Tableau[List_char.Taille -1]
    Créer un nouveau noeud      Plus_Faible_1.Taille+Plus_Faible_2.Taille : in Entier,
                                0 : in Entier,
                                Plus_Faible_1 : in T_Arbre,
                                Plus_Faible_2 : in T_Arbre ;
                                Nouv_Noeud : out T_Arbre
    List_Char.Tableau(Liste.Taille) <- null
    List_char.Taille <- List_char.Taille - 1
    List_char.Tableau[List_char.Taille] <- Nouv_Noeud

R3 :  Comment "Trier la liste de noeud de manière décroissante" ?
    POUR Indice = 1..(Tab.Taille-1) FAIRE
        Déterminer le maximum   Tab : in,
                                Indice : in ;
                                Indice_Max : out Entier
        Déplacer le maximum     Indice : in,
                                Indice_Max : in ;
                                Tab : in out
    FIN POUR

R3 : Comment "Avoir le code de l'arbre" ?
Entrées :   Arbre : in T_Arbre,
            Code : in T_FluxBinaire,
            Codage : out T_Codage

    SI Arbre.Gauche = Rien ALORS
        Codage(Arbre.Valeur) <- Code
    SINON
        Créer code          0 : in Entier,
                            Code : in ;
                            Code_Gauche : out T_Flux_Binaire
        Créer code          1 : in Entier,
                            Code : in ;
                            Code_Droite : out T_Flux_Binaire

        Avoir Code Arbre    Arbre.Gauche : in, Code_Gauche : in ; Codage : out
        Avoir Code Arbre    Arbre.Droite : in, Code_Droite : in ; Codage : out
```

```
            Vider                   Code_Gauche : in out
            Vider                   Code_Droite : in out
        FIN SI
```

R3 : Comment “Déterminer la liste des positions des caractères dans l'arbre” ?

```
        SI Arbre = Rien ALORS
            Rien
        SINON SI Arbre.Gauche = Rien ALORS
            Ajouter_Bit             1 : in Entier,
                                    Code_Structure_Arbre : in out ;
            Ajouter Arbre.Valeur à Liste_Position       Arbre.Valeur : in T_Octet,
                                                        Liste_Position : in out
        SINON
            Ajouter_Bit             0 : in Entier ;
                                    Code_Structure : in out
            Déterminer la liste des positions des caractères dans l'arbre
                                        Arbre.Gauche : in ;
                                        Liste_Position : in out
                                        Code_Structure_Arbre : in out
            Déterminer la liste des positions des caractères dans l'arbre
                                        Arbre.Droit : in ;
                                        Liste_Position : in out,
                                        Code_Structure_Arbre : in out
        FIN SI
```

R3 : Comment “Encoder la liste des positions” ?

```
        Ajouter à Arbre_Encode la position de ‘\$’   Liste_Position : in ;
                                                     Fichier_Encode : in out
        Ajouter tous les octets de la position       Liste_Position : in ;
                                                     Fichier_Encode : in out
        Ajouter_Flux                                 Code_Structure_Arbre : in ;
                                                     Fichier_Encode : in out
```

R3 : Comment “Encoder les octets de Nom_Fichier dans Nom_Compresse”

```
        TANT QUE  non Fin_Du_Fichier (Nom_Fichier) FAIRE
            Lire un octet                   Nom_Fichier : in ; Octet : out T_Octet
            Ecrire (Nom_Compresse, Codage(Entier(Octet) +  1))
        FIN TANT QUE
```

R4 : Comment “Ajouter les caractères de fréquences non nulles” ?

```
        SI Tableau_frequence(i) /= 0 ALORS
            List_char.Taille <- List_char.Taille + 1
            Créer un nouveau noeud      Tableau_frequence(i) : in Entier,
                                        I : in T_Octet,
                                        null : in T_Arbre,
```

```
                                        null : in T_Arbre ;
                                        List_char.Tableau(List_char.Taille) : out T_Arbre
        FIN SI
```

R4/5 : Comment “Créer un nouveau noeud” ?

```
Entrées :   Frequence : in Entier,
            Caractere : in T_Octet,
            Noeud_Gauche: in T_Arbre,
            Noeud_Droite : in T_Arbre,
            Arbre : out T_Arbre

        Initialiser         Arbre : out
        Enregistrer         Arbre : in out ;
                            Frequence : in,
                            Caractere : in,
                            Noeud_Gauche: in,
                            Noeud_Droite : in
```

R4 : Comment “Déterminer le maximum” ?

```
        Indice_Max <- Indice
        POUR Indice2 = Indice+1..Tab.Taille FAIRE
            Remplacer l’indice du maximum       Indice2 : in
                                                Tab : in ;
                                                Indice_Max : in out
        FIN POUR
```

R4 : Comment “Déplacer le maximum” ?

```
        Memoire <- Tab.Tableau(Indice)
        Tab.Tableau(Indice) <- Tab.Tableau(Indice_Max)
        Tab.Tableau(Indice_Max) <- Memoire
```

R4 : Comment "Créer code” ?

```
Entrées :
        Entier_code : in Entier
        Code : in T_Flux_Binaire
        Codage : out T_Flux_Binaire

        Initialiser         Codage : out
        Ajouter_Flux        Code : in ;
                            Codage : in out
        Ajouter_Bit         Entier : in ;
                            Codage : in out
```

R5 : Comment “Remplacer l’indice du maximum” ?

```
        SI Tab.Tableau(Indice2).Cle >= Tab.Tableau(Indice_Max).Cle ALORS
```

```
            Indice_Max <- Indice2
FIN SI
```

Annexe C : Raffinage de Décompresser

R0 : Décompresser un fichier .hff compressé avec un arbre d'Huffman

Exemples :

./decompresser exemple.txt.hff => exemple.txt.

./decompresser -b exemple.txt.hff => afficher l'arbre d'Huffman exemple.txt

./decompresser –bavard exemple.txt.hff => afficher l'arbre d'Huffman exemple.txt

./decompresser exemple.txt => "Le fichier à décompresser n'a pas été compressé par Huffman"

./decompresser exmple.txt.hff => "Fichier à décompresser inexistant."

./decompresser -a exemple.txt => "Option entrée non reconnue."

./decompresser => "Vous n'avez pas entré d'arguments."

./decompresser a b c => "Vous avez entré trop d'arguments."

R1 : Comment "Décompresser" ?

Déterminer Nom_Fichier avec les arguments de la ligne de commande
Affichage : out Booléen,
Nom_Fichier : out Chaîne_caractère

SI Fin_Nom_Fichier = ".hff" ALORS

Ouvrir le fichier Nom_Fichier — Nom_Fichier : in ;
Flux_Fichier : out T_Flux_Fichier

Recréer l'arbre de Huffman — Flux_Fichier : in ;
Arbre_Huffman : out T_Arbre

Reconstruire le fichier original — Flux_Fichier : in, Nom_Fichier : in,
Arbre_Huffman : in

SINON

LEVER Pas_Compression_Huffman

FIN SI

R2 : Comment "Déterminer Nom_Fichier avec les arguments de la ligne de commande" ?

Affichage <- False

SI Argument_Count > 2 ALORS

LEVER Trop_Arguments_Exception

SINON SI Argument_Count = 1 ALORS

Nom_Fichier <- Argument(1)

SINON SI Argument_Count = 0 ALORS

LEVER Peu_Arguements_Exception

SINON

Lire le double argument — Affichage : out Booléen,
Nom_Fichier : out Chaîne_caractère

FIN SI

R2 : Comment “Ouvrir le fichier Nom_Fichier” ?
	Ouvrir (Fichier, In_File, Nom_Fichier)
	Flux_Fichier_Brut <- Stream(Fichier)
	Initialiser le Flux du Fichier		Flux_Fichier_Brut : in Stream_Access ;
						Flux_Fichier : out
Exception
	Quand Nom_Fichier non trouver => LEVER Fichier_Inexistant_Exception

R2 : Comment “Recréer l’arbre Huffman” ?
	Décoder la liste des positions		Flux_Fichier : in ;
						Liste_Position : out T_Liste_Chainee
	Décoder la structure de l’arbre		Flux_Fichier : in, Liste_Position : in,
						Arbre_Huffman : out T_Arbre

R2 : Comment “Reconstruire le fichier original” ?
	Nom_Decompresse <- Nom_Fichier - “.hff”
	Créer et ouvrir le fichier Nom_Decompresse		Nom_Decompresse : in Chaine
	Décoder les octets de Nom_Fichier dans Nom_Decompresse		Arbre_Huffman :
in
							Flux_Fichier : in

R3 : Comment “Lire le double argument” ?
	SI Argument(1) = “-b” OU Argument(1) = “--bavard” ALORS
		Affichage <- True
		Nom_Ficher <- Argument(2)
	SINON
		LEVER Option_Inconnue_Exception
	FIN SI

R3 : Comment “Initialiser le Flux du Fichier” ?
	Flux.Fichier.Flux_Interne <- Flux_Fichier_Brut
	Flux_Fichier.Octet <- 0
	Flux_Fichier.Indice_Bit <- 8

R3 : Comment “Décoder la liste des positions” ?
	Initialiser		Liste_Positions : out
	Lire et ajouter la position du symbole de fin		Flux_Fichier : in ;
							Position_Fin : out T_Octet
	Ajouter (Liste_Position , Position_Fin)
	Lire un octet		Flux_Fichier: in, Octet : out T_Octet
	RÉPÉTER
		Ajouter (Liste_Position, Octet)
		Dernier_Octet <- Octet

```
        Lire un octet                          Flux_Fichier: in, Octet : out T_Octet
    JUSQU'À Octet = Dernier_Octet
```

```
R3 : Comment "Décoder la structure de l'arbre" ?
    Initaliser      Arbre_Huffman : out
    Lire un bit     Flux_Fichier : in, Bit : out Entier
    Position <- 0
    SI Bit = 1 ALORS
        Déterminer la clé de la valeur      Position : in, Position_Symbole_Fin : in ;
                                            Pas_Symbole_Fin : out Entier
        Enregistrer             Arbre_Huffman : in out ;
                                Pas_Symbole_Fin  : in,
                                La_Valeur(Liste_Positions, Positions) : in,
                                null : in,
                                null : in
        Position <- Position + 1
    SINON
        Enregistrer             Arbre_Huffman : in out ;
                                0 : in,
                                0 : in,
                                null : in,
                                null : in

        Décoder la structure de l'arbre     Arbre_Huffman .Gauche : out
        Décoder la structure de l'arbre     Arbre_Huffman .Droite : out
    FIN SI
```

```
R3 : Comment "Décoder les octets de Nom_Fichier dans Nom_Decompresse" ?
    Est_Fin_Du_Fichier <- False
    TANT QUE  non Est_Fin_Du_Fichier FAIRE
        Déterminer le prochain caractère    Nom_Fichier : in, Arbre : in,
                                            Octet : out T_Octet,
                                            Est_Fin_Du_Fichier out Booléen
        SI non Est_Symbole_Fin ALORS
            Ecrire (Nom_Decompresse, Octet)
        FIN SI
    FIN TANT QUE
```

```
R4 : Comment "Déterminer la clé de la valeur" ?
    SI Position = Position_Symbole_Fin ALORS
        Pas_Symbole_Fin <- 0
    SINON
        Pas_Symbole_Fin <- 1
```

```
        FIN SI

R4 : Comment “Déterminer le prochain caractère” ?
        TANT QUE Arbre.Gauche /= Rien FAIRE
                Lire un bit                 Nom_Fichier : in, Bit : out Entier
                Déterminer quelle branche parcourir         Bit : in ; Arbre : out
        FIN TANT QUE
        Octet <- Arbre.Valeur
        Est_Symbole_Fin <- Arbre.Cle = 0

---------------------------------------

R5 : Comment “Déterminer quelle branche parcourir” ?
        SI Bit = 0 ALORS
                Arbre <- Arbre.Gauche
        SINON
                Arbre <- Arbre.Droite
        FIN SI
```

Annexe D : Rapport Valgrind

Rapport pour `test_arbre` (OK)

```
vfontain@n7-ens-lnx008:~/pim/projet/src$ valgrind ./test_arbre
==38362== Memcheck, a memory error detector
==38362== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==38362== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==38362== Command: ./test_arbre
==38362==
Création d'arbres vides
Modification du noeud d'un arbre
Modification de l'arbre droit directement
Afficher Arbre
Le noeud à la valeur          10 pour la lettre e.
Le noeud à la valeur          20 pour la lettre a.
Le noeud à la valeur          50 pour la lettre o.
Test Sommage cle
Suppression Arbre
Fin des tests : OK.
==38362==
==38362== HEAP SUMMARY:
==38362==     in use at exit: 0 bytes in 0 blocks
==38362==   total heap usage: 9 allocs, 9 frees, 360 bytes allocated
==38362==
==38362== All heap blocks were freed -- no leaks are possible
==38362==
==38362== For lists of detected and suppressed errors, rerun with: -s
==38362== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rapport pour `test_liste_chainee` (OK)

```
vfontain@n7-ens-lnx008:~/pim/projet/src$ valgrind ./test_liste_chainee
==44955== Memcheck, a memory error detector
==44955== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==44955== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==44955== Command: ./test_liste_chainee
==44955==
Test Initialiser OK.
Test Est_Vide OK.
Test Taille OK.
Test Ajouter OK.
Test Enregistrer OK.
Test Inserer_Avant OK.
Test Supprimer OK.
Test Supprimer_Indice OK.
Test Valeur_Presente OK.
Test La_Valeur OK.
Test L_Indice OK.
Test Vider OK.
Test Pour_Chaque OK.

Fin des tests de Liste_Chainee : OK.
==44955==
==44955== HEAP SUMMARY:
==44955==     in use at exit: 0 bytes in 0 blocks
==44955==   total heap usage: 38 allocs, 38 frees, 608 bytes allocated
==44955==
==44955== All heap blocks were freed -- no leaks are possible
==44955==
==44955== For lists of detected and suppressed errors, rerun with: -s
==44955== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rapport pour `test_flux_binaire` (OK)

```
vfontain@n7-ens-lnx008:~/pim/projet/src$ valgrind ./test_flux_binaire
==45252== Memcheck, a memory error detector
==45252== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==45252== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==45252== Command: ./test_flux_binaire
==45252==
Test Initialiser OK.
Test Est_Vide OK.
Test Taille OK.
Test Le_Bit OK.
Test L_Octet OK.
Test Ajouter_Bit OK.
Test Ajouter_Octet OK.
Test Ajouter_Flux OK.
Test Vider OK.
Test Une_Chaine OK.

Fin des tests de Flux_Binaire : OK.
==45252==
==45252== HEAP SUMMARY:
==45252==     in use at exit: 0 bytes in 0 blocks
==45252==   total heap usage: 30 allocs, 30 frees, 528 bytes allocated
==45252==
==45252== All heap blocks were freed -- no leaks are possible
==45252==
==45252== For lists of detected and suppressed errors, rerun with: -s
==45252== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rapport pour `compresser` (OK)

```
vfontain@n7-ens-lnx008:~/pim/projet/src$ valgrind ./compresser exemple.txt
==45564== Memcheck, a memory error detector
==45564== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==45564== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==45564== Command: ./compresser exemple.txt
==45564==
==45564==
==45564== HEAP SUMMARY:
==45564==     in use at exit: 0 bytes in 0 blocks
==45564==   total heap usage: 119 allocs, 119 frees, 37,148 bytes allocated
==45564==
==45564== All heap blocks were freed -- no leaks are possible
==45564==
==45564== For lists of detected and suppressed errors, rerun with: -s
==45564== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rapport pour `decompresser` (OK)

```
vfontain@n7-ens-lnx008:~/pim/projet/src$ valgrind ./decompresser exemple.txt.hff
==45787== Memcheck, a memory error detector
==45787== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==45787== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==45787== Command: ./decompresser exemple.txt.hff
==45787==
==45787==
==45787== HEAP SUMMARY:
==45787==     in use at exit: 0 bytes in 0 blocks
==45787==   total heap usage: 52 allocs, 52 frees, 27,908 bytes allocated
==45787==
==45787== All heap blocks were freed -- no leaks are possible
==45787==
==45787== For lists of detected and suppressed errors, rerun with: -s
==45787== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```