COMPRESSION DE FICHIERS TEXTE

PAR CODAGE DE HUFFMAN

Introduction

Le but de ce TP est de réaliser un programme en Ada de compression-décompression à l'aide de l'algorithme du codage de Huffman.

Dans cette optique, il nous a été suggéré d'utiliser 4 modules : code.ads, dico.ads, file_priorite.ads et huffman.ads ; ceux-ci contenant des suggestions de fonctions et procédures à employer pour la réalisation de ce projet. Toutefois, d'autres ont été rajoutées par souci de clarté ou de factorisation.

I/ Réalisation des structures de données du projet

code.ads :

Le package code permet de représenter un code Binaire. En interne, le codage est représenté par une liste.

Les types Code_Binaire et Iterateur_Code sont des pointeurs sur leurs homologues homonymes internes :

→ Code_Binaire_Interne est une liste chaînée de Bits : cette implémentation a été retenue car la longueur d'un code est inconnu avant sa détermination. Il aurait pu être possible d'implémenter cette liste dans un tableau de taille variable dans une amélioration future pour améliorer le coup mémoire et le temps de calcul.

→ Iterateur_Code_Interne est un pointeur sur un élément de cette liste (initialement le premier). Il permet de parcourir de manière itérative la liste interne de bit tout en masquant la structure interne.

Dans une amélioration future il aurait pu être possible de séparer l'implémentation d'une liste générique et de l'utiliser dans le package Code.

dico.ads :

Le module Dico permet d'enregistrer la représentation de chaque caractères, et de leur associé un code. La structure principale est un tableau de Code indicé selon des Character (y compris des caractères non imprimables).

La structure Dico_Caracteres est un pointeur sur Dico_Caracteres_Interne. Un Dico_Caracteres_Interne est un enregistrement contenant :

→ le nombre de caractères différents dans le dictionnaire

→ le nombre de caractères total dans le dictionnaire (total des occurrences)

→ un tableau ayant des Character pour indices et des Info_Caractere pour valeurs.

La structure Info_Caractere est un enregistrement contenant :

→ le code associé au caractère (indice du tableau)

→ le nombre d'occurrences du caractère.

file_priorite.ads :

La file de priorité est implémentée comme un tas implémenté par d'un tableau et d'un entier représentant le nombre d'éléments actuellement dans la file de priorité. À chaque ajout ou suppression d'éléments dans le tas, la structure de celui-ci est maintenu à jour.

La structure File_Prio est un pointeur sur File_Interne. Une File_Interne est un enregistrement contenant :

→ Le nombre d'éléments dans la file.

→ Un tableau d'Element_Tas.

La structure Element_Tas est un enregistrement contenant :

→ Une Donnee (type générique).

→ Une Priorite (type générique).

Il aurait été possible dans une amélioration future de séparer l'implémentation du tas.

huffman.ads :

L'arbre de Huffman est implémentée comme un arbre classique.

La structure Arbre_Huffman est un pointeur sur Internal_Huffman. Internal_Huffman est un enregistrement contenant :

→ L'Arbre de Huffman à proprement parler.

→ Le dictionnaire lié à l'arbre.

→ Le nombre de caractères dans l'arbre et le dictionnaire.

Un Arbre est un pointeur sur un Noeud. Un Noeud est un enregistrement contenant :

→ Un caractère.

→ Deux fils, qui sont des Arbres.

Ce module utilise le package file_priorite, en employant les types Arbres et Integer en lieu et place de Donnee et Priorite, ainsi que la fonction "<" pour déteriner la priorité d'un élémet par rapport à un autre.

La majorité des fonctions et procédures ajoutées en plus de celles suggérées sont dans ce package : ainsi, il y a une fonction affichant graphiquement l'arbre, par exemple.

II/ Couts des algorithmes principaux

code.ads :

Étant implémentée sous forme simplement d'une liste simplement chaînée, la procédure Ajoute_Avant est en O(1), tandis que Ajoute_Apres et Longueur sont en O(n). Tous les autres algorithmes du package sont en O(1).

dico.ads :

La structure principale du dictionnaire étant un tableau dont on accède directement depuis un caractère, la majorité des algorithmes sont en O(1). Seul la procédure Affiche est en O(n), étant donné que l'on parcourt le tableau intégralement.

file_priorite.ads :

La file de priorité est implémentée comme tas stocké dans un tableau : les opérations principales (Insere et Supprime) sont en $\log_2(n)$. La fonction vérifiant si un élément appartient à la file est en O(n), et tous les autres algorithmes sont en O(1).

huffman.ads :

La procédure Affiche est en O(n), vu qu'il faut lire tout les éléments de l'arbre. Cree_Huffman est en O(nombre de caractères du fichier à compresser), de même que Ecrit_Huffman. En effet, il faut lire une première fois tout le texte pour connaître le nombre d'occurrences de chaque lettre, pu le lire une deuxième fois pour compresser chaque caractères. Les autres fonction sont en O(1).

Le package contient également une fonction de tests qui permet sur un exemple simple connu de tester son fonctionnement.

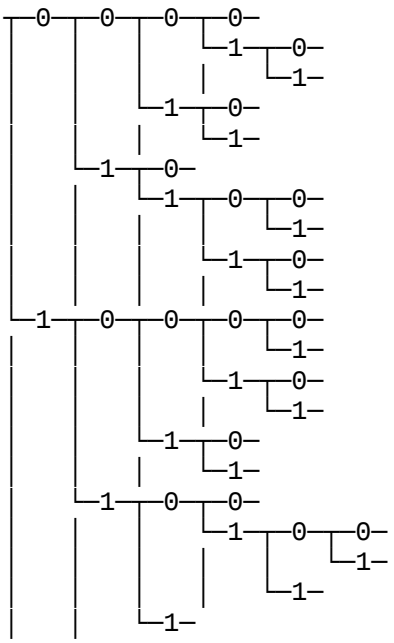
III/ Résultats obtenus

Le développement de ce projet s'est effectué de manière parallèle : nous avons ainsi codé et testé les packages séparément afin de s'assurer de leur rigidité. Nous avons de plus créer des tests unitaire pour valider le fonctionnement de tous modules.

Ainsi, les fichiers test_code.adb, test_file_priorite.adb et test_huffman vérifient les packages qui leur sont associés.

Pour les tests, nous avons générés un petit arbre de Huffman grâce au fichier Tests/3a_4b_5c_6d_7e.txt. Ce fichier contient 3 fois la lettre 'a', 4 fois la lettre 'b'… et nous a permis de tester facilement les étapes de compression et de décompression.

L'affichage graphique des arbres de Huffman s'est avérée extrêmement pratique pour débugger. En voici un exemple avec le fichier mini.txt fourni dans le répertoire Tests :



```
t:  ( 2 occurrences)
':  ( 1 occurrences)
:  ( 1 occurrences)
a:  ( 2 occurrences)
c:  ( 2 occurrences)
e:  ( 4 occurrences)
d:  ( 1 occurrences)
:  ( 1 occurrences)
n:  ( 1 occurrences)
m:  ( 1 occurrences)
h:  ( 1 occurrences)
I:  ( 1 occurrences)
P:  ( 1 occurrences)
l:  ( 1 occurrences)
p:  ( 2 occurrences)
.:  ( 3 occurrences)
r:  ( 3 occurrences)
,:  ( 1 occurrences)
T:  ( 1 occurrences)
s:  ( 2 occurrences)
 :  ( 7 occurrences)
```

Malheureusement nous n'avons pas réussis à terminer le TP dans le temps imparti, mais nous étions proche du but. En effet, seul les tout premiers et les tout derniers caractères ne sont pas correctement décompressés :

- Le fichier Tests/3a_4b_5c_6d_7e.txt contient : aaabbbbccccccddddddeeeeeee
- On obtient en sortie : ccedeaabbbbccccccddddddeeeeeeecccFin

À cela près, tout les packages autres que Huffman sont fonctionnels et testés en détails.

Conclusion

Au cours de ce projet, nous avons manipulé simultanément les principales structures de données rencontrées en programmation ; chaque structure est plus ou moins indiquée suivant la situation, il n'y a pas forcément de structure meilleure dans tous les cas.

A partir d'un moment, il a été prévu, si on avait eu le temps, de créer des modules génériques de liste, de tas et d'arbre génériques à utiliser respectivement pour les modules code.ads, file_priorite.ads et huffman.ads, mais cela ne s'est pas révélé être le cas.