Rapport Projet IPF

Le projet traite de la compression/décompression de fichier texte via l'algorithme de Huffman.
Nous avons décidé d'utiliser le code UTF-8 des caractères au-lieu du ASCII pour pouvoir prendre en compte plus de caractère.

Compression

Premièrement, on récupère la fréquence des lettres du texte et on l'inscrit dans une table de hachage car l'écriture a un coût constant. Nous avons entièrement construit un type `hash`, nous avons choisit la technique de double hachage. Les entrées de la table sont un type somme : `Occ of (int * int)` pour représenter `(code UTF-8 d'un caractère, nombre d'occurence)` ou `Code of (int * string)` pour représenter `(code UTF-8 d'un caractère, code huffman)`.

Secondement, nous avons transformé la table de hachage en tas. Nous avons donc défini un type `heap` représenté sous forme de tableau comme un produit nommé `length : int ; tab : 'a array ; comp : ('a -> 'a -> int)` où l'on a spécifié la fonction de comparaison entre éléments.

A partir de cela, c'était plus facile de construit l'arbre Huffman. De plus, nous avons du définir un nouveau type `tree` comme somme d'une feuille `L of int` et d'un noeud `N of (tree * tree)`. Ici, les feuilles contiennent les codes UTF-8 des caractères du texte.

Après l'arbre construit, nous avons crée une liste de `Code` obtenu en lisant l'arbre. Et, nous avons remplacé les champs de la table, toujours de type `Occ` par le type `Code`. Cest pour cela que l'on a opté pour un type somme dans les champs de la table : cela permet de ne pas recréer une table.

Enfin, nous avons écrit dans le nouveau fichier de compression : l'arbre Huffman en préfixe sous forme de bloc de 24 bits (le code du caractère en binaire et un bourrage de zéros avant) et pour les noeuds internes il s'agit du code 31. Puis, on écrit le texte compressé où chaque caractère est remplacé par son code Huffman.

Décompression

Tout d'abord, on lit le début du fichier `.hf` et on construit l'arbre Huffman. Lorsque l'arbre est correctement construit, la lecture s'arrête juste avant le corps du texte car cela découle de l'écriture préfixe.

Après, on utilise l'arbre pour décripter chaque caractère et l'on écrit en simultanée dans le fichier décompressé.

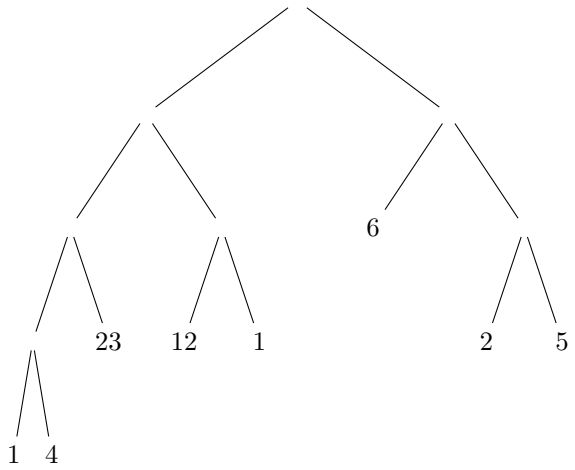
Jeux de tests

Heap

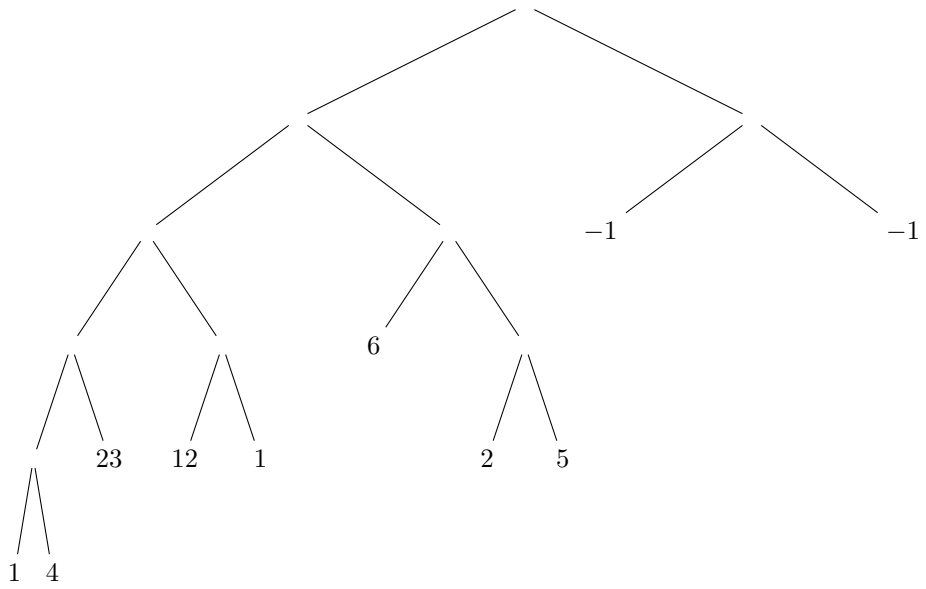
Pour exécuter le fichier de test, il faut lancer la commande `./heap_test.exe`
Ce fichier teste la structure `heap` : les fonctions d'ajout d'un élément, de récupération du minimum et de sa sortie.
En effet, le fichier test transforme la liste `[1 ; 5 ; 3 ; 11 ; 4]` en un tas, soit `| 1 | 4 | 3 | 11 | 5 |` en ajoutant petit à petit chaque élément (il n'existe pas un tas unique, tout dépend de l'ordre qu'on insère les éléments : tant que le père d'une valeur est plus petite qu'elle-même.), et c'est effectivement ce qu'on obtient.
Ensuite, on demande le minimum du tas (soit le premier élément), et on le retire et on vérifie que l'on obtient encore un tas : on obtient `| 3 | 4 | 5 | 11 |` qui est un tas (on peut redessiner facilement l'arbre associé pour vérifier).
Pour finir, nous ajoutons 6 dans le tas et on revérifie : on obtient `| 3 | 4 | 5 | 11 | 6 |` qui est un tas.

Arbre

Pour exécuter le fichier de test, il faut lancer la commande `./tree_test.exe`
Ce fichier teste la structure `tree` : la construction de l'arbre et les fonctions d'affichage.
En effet, on construit un arbre donc voici sa représention :



Si nous lisons cet arbre en préfixe, avec des underscores _ pour représenter les noeuds internes, nous obtenons `_ _ _ _ 1 4 23 _ 12 1 _ 6 _ 2 5` : c'est bien ce qu'affiche la fonction print.
De plus, si nous fusionons cet arbre avec une queue de cerise de -1, nous obtenons :



soit `_ _ _ _ _ 1 4 23 _ 12 1 _ 6 _ 2 5 _ -1 -1` en préfixe : c'est bien ce que nous obtenons.

Table de Hachage

Pour exécuter le fichier de test, il faut lancer la commande `./hash_table_test.exe`
Ce fichier teste la structure `tree` : la construction du tas, la recherche d'un élément, ainsi que l'ajout et la modification d'un élément.
Dans ce cas, nous allons utiliser le type `Occ` comme élément, mais cela fonctionnerait tout autant avec `Code`.
Tout d'abord, on transforme la liste `[ Occ(25, 2) ; Occ(1263, 5) ; Occ(24, 263) ; Occ(152, 1) ; Occ(254, 24) ]` en une table de hachage. Puisque c'est une table de hachage, nous ne pouvons pas vraiment prédire l'ordre des éléments, à part calculer un par un toutes les valeurs de hachages, mais nous nous attendons à avoir toutes les valeurs de la liste dedans : ce qui est le cas, en sachant que le double hachage gère les conflits.
De plus, nous testons si la valeur 152 et 12 sont dedans : en effet, le fonction qui cherche un élément renvoie l'indice de l'élément dont la première composante est la valeur, plus un booléen indiquant s'il est dedans, et donc pour 152 et 12, on est censé obtenir vrai et faux, et c'est bien le cas. Pour aussi vérifier que l'on peut changer la valeur, nous ajoutons l'élément `Occ(12, 1)` et nous modifions l'élément ayant comme valeur 152 : là aussi, tout fonctionne.

Fichier contenant « satisfaisant »

Pour exécuter le fichier de test, il faut lancer la commande `./test.exe`
Ce fichier tester tout l'algorithme du programme sur un petit texte contenant le mot « satisfaisant ». Il n'utilise pas les fonctions `compress` et `decompress`, mais décortique petit à petit le process :

- On compte la fréquence de chaque lettre dans une table de hachage qu'on convertit en tas et on affiche les valeurs ;
- On traduit ce tas en un arbre de Huffman ;
- On compresse le fichier en y stockant l'arbre puis le texte compressé ;
- On décompress seulement l'arbre pour vérifier qu'il est identique à l'arbre construit plus tôt ;

Ainsi, tout fonctionne et on obtient bien tout ce que l'on souhaitait, y compris obtenir les deux arbres à l'identique.

Répartition du travail

Au début, pendant les séances de TP, nous travaillions ensemble sur un même ordinateur ce qui nous permettait de mettre en commun nos idées. De plus, nous avions choisi de commencer par la table de hachage et l'arbre avant de récupérer les lettres du texte. Après cela, nous avons réalisé les fonctions de compression.

Lorsque les séances de TP se sont arrêtées, Romain a crée la structure de tas, écrit les documentations et les commandes du fichier `huff.exe`. Matthieu a écrit la fonction de décompression, les tests et l'implémentation du tas dans nos fonctions.

Exemple d'une fonction

Nous avons choisi la fonction `Read_write.header_to_tree`.
Il s'agit de la fonction qui reconstruit l'arbre stocké au début du fichier compressé, en écriture préfixe.
La fonction `aux` lit sur le stream et construit l'arbre. Elle est ainsi appelée pour les deux sous-arbres.
La fonction `aux1` permet de lire chacun des caractères du préfixe, en lisant 24 bits à la fois et stocke les chiffres significatifs du code binaire dans une liste par ordre croissant de poids. Par exemple, le bloc `000000000000000000011010` donnera `[ 0 ; 1 ; 0 ; 1 ; 1 ]`.
La fonction `aux2` traduit cette écriture binaire en décimal. Ici, `[0;1;0;1;1]` donne `26`.
Si cet entier vaut `31`, cela signifie que c'est un noeud interne, on appelle alors deux fois `aux` et l'on construit l'arbre à partir de ces deux renvois.
Sinon, c'est une feuille, on renvoie la feuille avec comme valeur l'entier.