université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Projet IPF
LDD2 IM

Opérations sur l'arbre binaire, tests, rapport

Objectif

Le but de cette feuille est de :

1. donner quelques pistes pour les algorithmes sur l'arbre binaire (arbre de Huffman)
2. donner quelques informations sur le stockage de suite de *bits* dans un fichier
3. donner quelques pointeurs vers des fichiers de tests intéressant
4. donner des précisions sur ce qui est attendu

Cette feuille n'est qu'un guide, en particulier si vous avez réussi à bien avancer sans, vous n'êtes aucunement obligé de vous y conformer. Elle contient cependant des informations utiles pour vous assurer que vous n'avez rien oublié et donne aussi des pointeurs sur ce qu'il faut mettre dans le rapport accompagnant le code.

Remarque : il s'agit d'un **projet** et non pas de TP notés. Des indications trop précises (comme la liste exacte des fonctions à écrire avec leur type) irait à l'encontre de l'esprit de l'UE qui doit laisser une bonne part à votre originalité dans la façon d'aborder le problème. Évidemment, le problème étant relativement guidé, il est normal que tous les groupes qui arrivent à la fin obtiennent une solution similaire, mais certains choix d'implémentation rendront ces projets différents.

1 Arbre binaire

Un exemple d'abre de Huffman est représenté à la figure 1. Deux problématiques principales sont

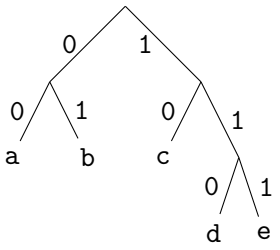

FIGURE 1 – Un arbre de Huffman

liées à de tels arbres :

— On veut le manipuler en mémoire de façon simple : construire un arbre, le parcourir, y insérer des valeurs, trouver le code d'un caractère stocké ...
— Le stocker dans un fichier et le relire ensuite.

1.1 Structure de données

Le langage OCaml brille particulièrement lorsqu'il s'agit de modéliser des structures récurisves. On propose de représenter un arbre binaire par le type récursif suivant :

```
1   type tree =
2       | Leaf of int
3       | Node of tree * tree
```

L'entier stocké dans la feuille correspond au code du caractère. On peut alternativement stocker des valeurs du type `char`. Cependant, les caractères étant parfois utilisés en tant qu'indices de tableau (par exemple le tableau des fréquences de chaque caractère), stocker un entier plutôt qu'un `char` évite d'alourdir le code avec des conversions entre ces deux types.

Évidemment tout type semblable mais qui diffère légèrement est tout à fait acceptable (nom différent pour les constructeurs, types différents pour les feuilles, ...).

Étant donné un tel arbre, on souhaite calculer pour chacun des caractères se trouvant aux feuilles son code en binaire. Par exemple, pour l'arbre :

```
1   let ex_tree = Node (Node (Leaf 97, Leaf 98),
2                       Node (Leaf 99, Node (Leaf 100, Leaf 101)))
```

correspondant à l'arbre de la figure 1, (le code ASCII de `a` est 97), on veut calculer les codes : `a`↦`00`, `b`↦`01`, `c`↦`10`, `d`↦`110` et `e`↦`111`.

Une façon simple de procéder est de faire un parcours récursif de l'arbre avec un argument supplémentaire servant d'accumulateur. Lors que l'on va à gauche, on « ajoute un 0 » en fin d'accumulateur et lorsqu'on va à droite on « ajoute un 1 » en fin d'accumulateur. La nature de l'accumulateur et ce que signifie « ajouter en fin » est laissé à votre appréciation.

1.2 Sérialisation et déserialisation de l'arbre

La partie la plus subtile est la sérialisation de l'arbre de Huffman dans le fichier (écriture) et sa déserialisation (lecture). En réalité, écrire l'arbre est très simple (on peut voir ça comme une variante de la fonction de débuggage permettant d'afficher l'arbre dans la console). Par contre, relire une suite de bits et reconstruire un arbre binaire est plus compliqué.

Un algorithme classique pour enregistrer un arbre binaire est de le parcourir récursivement. Si on est sur un nœud interne, on écrit un `1` puis on écrit récursivement le sous-arbre gauche puis le droit. Si on est sur une feuille on écrit un `0`. Relire l'arbre se fait de façon duale : si on lit un `0` alors on crée un `Leaf (-1)`. Si on lit un `1`, alors on lit récursivement un premier arbre t_1, puis un autre arbre t_2 et on construit `Node(`t_1`,` t_2`)`. Ainsi on peut représenter (dans un fichier) l'arbre précédent par la suite de bits :

`1 1 0 0 1 0 1 0 0`	`01100001`	`01100010`	`01100011`	`01100100`	`01100101`
structure de l'arbre	97	98	99	100	101
ordre de feuille	1	2	3	4	5

Une fois reconstruit l'arbre avec des `-1` aux feuilles, on peut lire les octets suivants que l'on aura stockés à la suite de l'arbre. Le premier octet correspond à la première feuille (la plus à gauche), puis le second à la feuille suivante (rencontrée lors d'un parcours en profondeur d'abord) et ainsi de suite.

1.3 Écriture de bits dans un fichier

Un fichier est une collection **d'octets** se trouvant sur un système de fichiers. Cependant, notre fichier compressé sera composés d'une suite de *bits* dont la taille n'est pas forcément un multiple de 8. Par exemple supposons que l'on veuille compresser le texte `"abcde"`. Si le calcul de l'arbre de Huffman est celui de la figure 1, on obtient la suite de bits représentée à la figure 2. La taille totale du fichier

`110010100`	`01100001`	`01100010`	`01100011`	`01100100`	`01100101`	`00`	`01`	`10`	`110`	`111`
arbre	97	98	99	100	101	`a`	`b`	`c`	`d`	`e`
taille en bits	9	8	8	8	8	8	2	2	2	3	3

FIGURE 2 – La représentation binaire du texte `abcde` compressé

compressé [1] est de 61 bits, ou 7 octets et 5 bits. Les primitives d'OCaml (ou de tout autre langages) ne permettent pas directement de stocker les 5 bits restants dans un fichier (on pourrait stocker un octet entier, mais il faudrait se souvenir que sur cet octet, les 3 derniers bits sont inutiles, donc stocker de l'information supplémentaire).

Pour simplifier cette problématique, une petite bibliothèque `Bs` vous est fournie (pour *bit stream*). Cette dernière permet de lire et écrire des suites de *bits* dans un fichier. Vous êtes invités à lire le fichier `bs.mli`, se dernier contenant la documentation et les fonctions publiques. On donne ci-dessous un simple exemple permettant d'écrire et de lire manuellement la suite de bits de la figure 2.

```
let cout = open_out "test.bin"  (* ouverture d'un fichier *)
let os = Bs.of_out_channel cout (* descripteur de fichier bit à bit *)
let () =
   Bs.write_n_bits os 9 404; (* 404 = 110010100 en base 2 *)
   Bs.write_n_bits os 8 97;
   Bs.write_n_bits os 8 98;
   Bs.write_n_bits os 8 99;
   Bs.write_n_bits os 8 100;
   Bs.write_n_bits os 8 101;
   Bs.write_n_bits os 2 0;  (* 00 *)
   Bs.write_n_bits os 2 1;  (* 01 *)
   Bs.write_n_bits os 2 2;  (* 10 *)
   Bs.write_n_bits os 3 6;  (* 110 *)
   Bs.write_n_bits os 3 7;  (* 110 *)
   Bs.finalize os;
   close_out cout
```

Les deux dernières instructions sont très importantes :

- `Bs.finalize` permet d'écrire la fin de fichier, en particulier les informations de combiens de bits sont inutilisés dans le dernier octet.
- `close_out` est la fonction standard d'OCaml permettant de fermer le fichier, et donc d'envoyer les octets sur le disque.

Des fonctions symmétriques permettent la lecture bit à bit d'un fichier. Enfin, il est particulièrement important de ne **jamais** tenter d'accéder au descripteur de fichier `cout` avec des fonctions standard d'OCaml (écriture d'une chaîne, d'un octet,...) car cela faussera le compteur interne du nombre de bits écrits, maintenu dans la structure `os`.

Attention, un comportement non intuitif de la bibliothèque a été corrigé. Vous devez donc télécharger la nouvelle version du fichier `bs.ml` sur la page du cours et le placer dans votre squelette.

2 Tests

Il convient de faire deux types de tests :

- des tests sur de petits fichiers (comme `satisfaisant`, mais pas seulement) en calculant manuellement l'arbre de Huffman et en essayant de prédire ce que votre programme va stocker.
- des tests sur de gros fichiers textes, afin de vous assurer que les algorithmes et structures de données employés ne sont pas trop naïfs

Pour des gros fichiers, vous pouvez par exemple utiliser le site `https://www.gutenberg.org/`. Ce dernier contient un grand nombre de livres en format numérique (dont fichier texte). Les livres proposés sont dans le domaine public.

1. Ici, évidemment le texte est tellement court que l'impact de la compression est négatif, le fichier compressé est plus gros que l'original.

3 Rapport

Le rapport doit se trouver à la racine de votre dépot `git`, au **format PDF**. Il doit être succinct (environs 5 pages, pas plus que 10). Il doit mentionner :

- les types de données demandés (arbre de Huffman, file de priorité, autres types auxiliaires)
- donner un exemple d'**une** fonction que vous avez trouvé complexe en expliquant son fonctionnement sur une exemple. Vous pouvez vous contenter de simplement expliquer le comportement des fonctions auxiliaires auxquelles vous fait appel sans en détailler le code.
- décrire les jeux de tests que vous avez effectués : les fichiers utilisés, la commande pour tester, le resultat attendu et le resultat obtenu. Si, au moment du rendu, votre code est encore buggué, donner un jeu de test qui illustre le bug permettra de récupérer une partie des points.
- la répartition du travail au sein du binôme

La date finale du rendu est fixée au dimanche 26 janvier, 20h00. À cette date, nous ferrons un « clone » de votre dépôt git qui devra contenir le code, le rapport et les jeux de tests.