Algorithmique des structures de données arborescentes
Feuille d'exercices 10

1 Codage de Huffman

Dans cette feuille on s'interesse à la compression sans perte de fichiers texte. L'objectif est de réduire la taille des fichiers. La difficulté est de pouvoir reconstruire le fichier original sans perte.

Un fichier est une suite de caractères. Un codage associe à chaque caractère une représentation par un ou plusieurs octets (exemple : codage ASCII, codage UTF-8). Le codage de Huffman est une technique classique de compression de fichiers texte, qui associe aux caractères d'un texte un nombre variable de bits, éventuellement moins que 8, en fonction de leur fréquence dans le texte. Plus la fréquence est haute, plus le codage doit être petit pour compresser au mieux le texte original.
Appelons mot binaire une suite de 0 et de 1. Par exemple 01001. Le principe du codage de Huffman est de construire un arbre binaire permettant d'associer à chaque caractère du texte un mot binaire de sorte que plus le caractère est fréquent dans le texte plus le mot binaire associé sera court.

arbres de Huffman

Les arbres de Huffman sont des arbres binaires complets tels que :

- les feuilles sont étiquetées par des caractères (les caractères du texte à compresser)
- l'arête entre un nœud et son fils gauche est étiquetée par 0 et
- l'arête entre un nœud et son fils droit est étiquetée par 1
- le mot binaire associé à un caractère c est celui lu sur la branche qui mène de la racine à la feuille étiquetée par c

Par exemple la figure 10.1 représente un arbre de Huffman.

On obtient ensuite le codage du texte à compresser par la concaténation des mots binaires correspondant à chacun de ses caractères. Par exemple, dans le code précédent, la chaîne "tentant" est représentée par 0111100110100 (concaténation des mots : 0 . 111 . 10. 0 . 110 . 10 . 0).

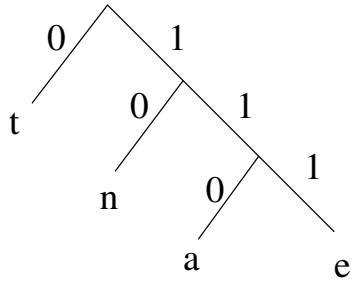

FIG. 10.1 : Un arbre de Huffman

1.1 Rappel sur le tri par insertion :

Exercice 10.1

- Écrire une fonction `insert_sorted` de type `'a -> 'a list -> ('a -> 'b) -> 'a list` qui prend en argument un élément `x`, une liste `l` et une fonction `key_fun` à un argument qui donne une clef de comparaison utilisée pour le tri. La liste `l` est supposée triée en ordre croissant selon la fonction `key_fun` : `x` est avant `y` dans `l` si et seulement si `key_fun x <= key_fun y`. La fonction `insert_sorted` insère l'élément `x` dans la liste `l` à bonne place.

- Écrire une fonction `insertion_sort` de type `'a list -> ('a -> 'b) -> 'a list` qui prend en argument une liste `l` et une fonction `key_fun` qui donne la clef de comparaison, et qui applique à `l` le tri par insertion.

1.2 Construction d'un arbre binaire correspondant au codage de Huffman :

Nous nous intéressons maintenant à la construction d'un arbre de Huffman. Les codes des différents caractères doivent tenir compte de leurs fréquences respectives. Pour cela, nous allons construire dans un premier temps la liste `l` de paires `(c, freq)` où `freq` correspond à la fréquence d'apparition du caractère `c` dans le texte (pour rappel, la fréquence est le nombre d'occurrences du caractère divisé par le nombre total de caractères du texte). La liste `l` sera triée dans l'ordre croissant des fréquences.
Ainsi par exemple, pour le texte "tentant", la liste `l` est la suivante :
`l = [('e',0.14); ('a',0.14); ('n',0.29); ('t',0.43)]`.
Dans cet exemple, le calcul des fréquences est arrondi au deuxième chiffre après la virgule.

On commence par remplacer dans `l` chaque caractère `c` par un arbre réduit à une feuille `Leaf(c)`. L'algorithme de Huffman consiste ensuite à itérer le processus suivant sur la liste `l` jusqu'à ce que celle-ci soit réduite à un unique élément :

1. **Répéter** tant que a plus d'un élément :
   (a) Retirer de la liste `l` les deux premières paires `(n1,freq1)` et `(n2,freq2)`. Ce sont les deux paires de fréquences minimales.
   (b) Construire un nouveau nœud `n'` avec fils gauche `n1` et fils droit `n2`
   (c) Insérer dans la liste `l`, en respectant l'ordre croissant, la nouvelle paire `(n', freq1 + freq2)`.
2. L'algorithme **se termine lorsque** la liste `l` est réduite à un unique couple `(n,f)` : `n` est alors l'arbre de Huffman recherché.

Exercice 10.2 On considère le texte suivant :

SWIMMING IN THE MISSISSIPPI

Appliquer l'algorithme de Huffman sur ce texte pour
- créer l'arbre de Huffman permettant d'associer un code à chaque caractère,
- produire la suite de bits correspondant au texte.

Pour représenter un arbre de Huffman, en pratique, il suffit de considérer un arbre binaire dans lequel nous convenons que d'aller regarder le sous-arbre gauche équivaudra à lire un 0, et le sous-arbre droit à lire un 1. On utilisera donc dans la suite de cette feuille, le type suivant pour représenter les arbres de Huffman.

```
type htree = Leaf of char | Node of htree * htree
```

Seules les feuilles d'un tel arbre sont étiquetées. Les noeuds internes ne portent pas de valeur.

Exercice 10.3 Pour la compression, on veut d'abord créer une liste de caractères avec leur fréquence. Pour écrire un vrai compresseur, il faut lire les caractères à partir d'un fichier. Pour simplifier dans cette feuille d'exercices, on supposera que les caractères proviennent d'une liste de type `char list` et on utilisera le nombre d'occurrences (int) plutôt que la fréquence (float).
Exemple avec "tentant" : `let text = ['t'; 'e'; 'n'; 't'; 'a'; 'n'; 't']`

1. Écrire une fonction `add_char` de type `char -> (char * int) list -> (char * int) list` qui prend en argument un caractère `c` , et une liste `l` de couples (caractère, nb_occurrences), supposée triée par ordre croissant des caractères, et qui ajoute le caractère `c` à la liste `l` : si `c` est déjà présent dans la liste, son nombre d'occurrences sera augmentée de `1`, et sinon, il sera inséré avec nombre d'occurrence `1` *à la bonne place* : si `c` < `d`, alors `c` doit être inséré avant `d`.
2. Écrire une fonction `charlist_to_freqlist` de type `char list -> (char * int) list` qui, à partir d'un texte donné sous forme d'une liste de caractères, retourne la liste des couples (caractère, nombre d'occurrences) triée par ordre croissant du nombre d'occurrences. On peut utiliser la fonction `insertion_sort` écrite dans l'exercice précédent.

Exemple avec ”tentant” :

```
# charlist_to_freqlist text;;
- : (char * int) list = [('a', 1); ('e', 1); ('n', 2); ('t', 3)]
```

3. Écrire une fonction `charlist_to_treelist` de type `char list -> (htree * int) list` qui produit la liste initiale de couples (arbres à un unique nœud, nombre d'occurrences) utilisée dans l'algorithme de Huffman. Les éléments de cette liste devront être triés par seconde composante croissante.
Exemple avec ”tentant” :

```
#  charlist_to_treelist text;;
- : (htree * int) list =
[(Leaf 'a', 1); (Leaf 'e', 1); (Leaf 'n', 2); (Leaf 't', 3)]
```

4. Écrire une fonction `treelist_to_hufftree` prenant en argument une liste de couples (arbre, poids) qui construit l'arbre de Huffman. Si la liste est vide, une erreur sera déclenchée avec `failwith`. On peut utiliser la fonction `insert_sorted` écrite dans l'exercice précédent.
Exemple avec ”tentant” :

```
# treelist_to_hufftree l;;
  - : htree = Node (Leaf 't', Node (Node (Leaf 'a', Leaf 'e'), Leaf 'n'))
```

1.3 Encodage de texte :

Exercice 10.4

1. Écrire une fonction `codelist` de type `htree -> (char * int list) list` qui, à partir d'un arbre de Huffman, produit une liste de couples (caractère, code), où le code d'un caractère est une liste de 0 et de 1.
Exemple avec ”tentant” :

```
#  codelist (Node(Leaf 't', Node(Node(Leaf 'a', Leaf 'e'), Leaf 'n')));;
- : (char * int list) list =
[('t', [0]); ('a', [1; 0; 0]); ('e', [1; 0; 1]); ('n', [1; 1])]
```

2. Écrire une fonction `encode : char list -> int list` qui produit la liste de 0 et 1 codant le texte donné sous forme d'une liste de caractères.
Exemple avec ”tentant” :

```
# encode ['t'; 'e'; 'n'; 't'; 'a'; 'n'; 't'];;
- : int list =
[0; 1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0]
```

1.4 Decodage de texte :

Exercice 10.5 Pour la décompression, on utilise un arbre de Huffman `t` et une liste `l` de 0 et de 1, et on veut produire la liste des caractères correspondants à `l`.

1. Écrire une fonction `nextchar_and_tail` de type `htree -> int list -> char * int list` qui prend en argument un arbre de Huffman `hufftree` et une liste `l` de 0 et de 1, et qui renvoie le couple (caractère, reste) où le caractère est l'étiquette de la feuille atteinte en lisant le début de `l` dans `hufftree` (en interprétant 0 comme “gauche” et 1 comme “droite”), et où `reste` est ce qui n'a pas été consommé dans `l`. La fonction renverra une erreur si on n'atteint pas une feuille de `hufftree`.
Exemple avec la liste `[0; 1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0]` et l'arbre du paragraphe précédent :

```
# nextchar_and_tail hufftree [0; 1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0];;
      - : char * int list = ('t', [1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0])
# nextchar_and_tail hufftree [1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0];;
- : char * int list = ('e', [1; 1; 0; 1; 0; 0; 1; 1; 0])
# nextchar_and_tail hufftree [1; 1; 0; 1; 0; 0; 1; 1; 0];;
- : char * int list = ('n', [0; 1; 0; 0; 1; 1; 0])
```

2. Écrire une fonction `decode : htree -> int list -> string` telle que `decode hufftree l` produit la chaîne obtenue par décodage de la suite de bits de `l` en utilisant l'arbre de Huffman `hufftree`.
   Exemple de décodage de `[0; 1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0]` :

```
# decode t [0; 1; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0];;
- : string = "tentant"
```