Algorithmique des structures de données arborescentes
Feuille d'exercices 7 (semaine 10)

1 Codage de Huffman

Exercice 7.1 On considère le texte suivant :

SWIMMING IN THE MISSISSIPPI

Appliquer l'algorithme de Huffman vu en cours 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.

Dans la suite de cette feuille, on utilise le type suivant pour représenter les arbres de code de Huffman.

```
type huffman_tree = Single of char | Double of huffman_tree * huffman_tree
```

Seules les feuilles d'un tel arbre sont étiquetées.

Exercice 7.2 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`.

1. Écrire une fonction `insert_sorted` de type `char -> (char * int) list -> (char * int) list` qui prend en argument un caractère `c`, et une liste `l` de couples (élément, multiplicité), triée par ordre croissant sur la 1re composante, et qui ajoute l'élément `c` à la liste `l`. Si `c` est déjà présent dans la liste, sa multiplicité sera augmentée de 1, et sinon, il sera inséré avec multiplicité 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'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 `List.sort`.
3. Écrire une fonction `charlist_to_treelist` de type `char list -> (huffman_tree * 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.
   ```
   charlist_to_treelist ['c'; 'a'; 'a'; 'b'; 'b'; 'a'; 'a'; 'b'; 'c'];;
   - : (huffman_tree * int) list = [(Single 'c', 2); (Single 'b', 3); (Single 'a', 4)]
   ```
4. Écrire une fonction `treelist_to_hufftree` prenant en argument une liste de couples (arbre, poids) qui construit l'arbre de Huffman avec l'algorithme vu en cours. Si la liste est vide, une erreur sera déclenchée avec `failwith`.
5. On accède au i^e caractère d'une chaîne `s` par `s.[i]` (le premier caractère est à la position 0). Écrire une fonction `string_to_hufftree` de type `string -> huffman_tree` qui produit l'arbre de Huffman associé à la chaîne de caractères passée en argument.
6. Écrire une fonction `codelist` de type `huffman_tree -> (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.
7. Écrire une fonction `encode : string -> int list` qui produit la liste de 0 et 1 codant la chaîne argument.

Exercice 7.3 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 `huffman_tree -> int list -> char * int list` qui prend en argument un arbre de Huffman `t` 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 `t` (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.
2. Écrire une fonction `decode : huffman_tree -> int list -> string` telle que `decode t l` produit la chaîne obtenue par décodage de la suite de bits de `l` en utilisant l'arbre de Huffman `t`.