Outils logiques et algorithmiques – TP/DM – Compression de Huffman

Ce TP est commencé en classe lors de la séance du 13, du 14 ou du 17 avril selon les groupes. Il doit ensuite être complété à la maison pour le 3 mai. Le travail est individuel, et est à rendre sur ecampus, dans la rubrique correspondant à votre groupe de TD. Le rendu doit prendre la forme d'un unique fichier `.ml` *portant votre nom, dans lequel les réponses aux questions doivent apparaître dans l'ordre. Lorsqu'une question demande d'écrire une fonction caml, l'introduire par un commentaire donnant le numéro de la question, puis donner le code. Par exemple :*

```
(* Question 2 *)
let rec taille t = match t with
  ...
```

Lorsqu'une question demande une réponse rédigée en français, donner l'intégralité de la réponse dans un commentaire. Par exemple :

```
(* Question 22
   Montrons par récurrence structurelle sur t que
     pour tout x, taille(ajoute(x, t)) = 1 + taille(t)
   Cas E
     ...
   Cas
     ...
 *)
```

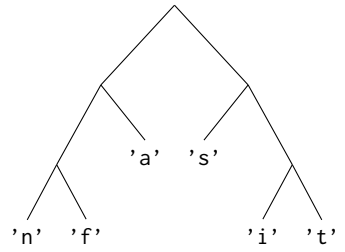
Après chaque question de code, il est judicieux d'inclure quelques tests. Les types imposés pour les structures de données et les fonctions sont à respecter ***impérativement*** *(sinon, c'est zéro).*

Considérons un fichier contenant un texte écrit en français. Dans un format standard (par ex. ASCII ou UTF8), chaque caractère ordinaire y est représenté par un octet, c'est-à-dire une séquence de 8 bits (éventuellement deux octets/16 bits pour des caractères accentués ou spéciaux). Pour rendre le fichier plus compact, on peut adopter une représentation des caractères différente des formats standards, avec des séquences de bits plus courtes pour les caractères les plus utilisés, quitte à avoir en contrepartie des séquences plus longues pour les caractères rares. Le nombre de bits par caractère sera ainsi plus faible en moyenne.

L'algorithme de Huffman définit une telle représentation à partir de statistiques sur la fréquence des caractères.

1. Arbre de Huffman

On va représenter le codage des caractères par un *arbre préfixe* (voir TD9, exercice 3). Rappel : un arbre préfixe est ici un arbre binaire, où chaque nœud est associé au mot sur l'alphabet $\{0, 1\}$ décrivant les directions à prendre pour atteindre ce nœud depuis la racine (0 pour aller au fils gauche, et 1 pour aller au fils droit). On définit un codage en étiquetant chaque feuille de l'arbre par un caractère. Voici ci-dessous un tel arbre, avec la table de correspondance entre les caractères et des séquences de bits. Remarquez que certains caractères sont codés par une suite de 2 bits, et d'autres par une suite de 3 bits, et que cette longueur correspond à la profondeur du caractère dans l'arbre.



'n'	000
'f'	001
'a'	01
's'	10
'i'	110
't'	111

Cet arbre permet de retrouver le caractère associé à une séquence de bits : si la séquence commence par 0 on poursuit le décodage dans le sous-arbre gauche, et si la séquence commence par 1 on poursuit dans le sous-arbre droit. Et à la fin, on consulte la feuille atteinte. Pour l'instant, on considère le code comme donné, et nous verrons plus tard comment construire un code « judicieux ».

On se donne le type suivant pour représenter un arbre de Huffman en caml.

```
type arbre =
  | C of char
  | N of arbre * arbre
```

Ainsi, `C 'a'` est la feuille contenant le caractère `'a'`, et `N(C 'n', C 'f')` est le nœud dont les sous-arbres sont des feuilles contenant respectivement les caractères `'n'` et `'f'`. On représente les mots binaires par de simples listes d'entiers dont on suppose qu'elles ne contiennent que des `0` et des `1`.

```
type mot = int list
```

Ainsi, le mot 110 codant le caractère `'i'` dans l'arbre ci-dessus sera représenté par la liste `[1; 1; 0]`.

1. Définir un arbre `t` correspondant à l'exemple ci-dessus.
2. Écrire une focntion `taille: arbre -> int` qui prend en paramètre un arbre de Huffman `t`, et qui renvoie le nombre de caractères contenus dans `t` (on suppose qu'il n'y a pas de doublons, c'est-à-dire que chaque occurrence de `C` porte bien un caractère distinct des autres).
3. Écrire une fonction `contient: char -> arbre -> bool` qui prend en paramètres un caractère `c` et un arbre de Huffman `t`, et qui renvoie `true` si et seulement si `t` contient `c`.
4. Écrire une fonction `code_char: char -> arbre -> mot` qui prend en paramètres un caractère `c` et un arbre de Huffman `t` contenant `c`, et qui renvoie le mot associé à `c`. La fonction peut déclencher une erreur si `t` ne contient pas `c`.
5. Quelle est la complexité de cette fonction `code_char` ? Peut-on lier cette complexité au nombre de caractères dans l'arbre ? Si oui, préciser comment et expliquer pourquoi ce lien est valide.
6. Écrire une fonction `reconnait: mot -> arbre -> bool` qui prend en paramètres un mot `m` et un arbre de Huffman `t`, et qui renvoie `true` si et seulement si `m` désigne bien un caractère dans `t`.
7. Écrire une fonction `decode_mot_simple: mot -> arbre -> char` qui prend en paramètres un mot `m` et un arbre de Huffman `t` reconnaissant `m`, et qui renvoie le caractère associé à `m`. La fonction peut déclencher une erreur si `t` ne contient pas `c`.
8. Quelle est la complexité de la fonction `decode_mot_simple` ?

2. Codage d'un texte entier

En pratique, la séquence de bits à décoder n'est pas un mot représentant un unique caractère, mais une séquence de bits obtenue en concaténant les codes de tous les caractères du message d'origine. Ainsi, avec l'exemple ci-dessus le message « `satisfaisant` » va être codé par la séquence « `10 01 111 110 10 001 01 110 10 01 000 111` ». Note : les espaces dans cet exemple vous permettent de facilement détecter les codes individuels des différents caractères, mais ils n'existent pas dans la séquence codée, qui est simplement `100111111010001011101001000111`. Le décodage d'un texte complet prend donc une tournure légèrement différente de ce qu'on a vu jusque là : on lit des bits du texte codé un à un jusqu'à obtenir le code d'un caractère, puis on recommence avec le reste de la séquence, jusqu'à ce que la séquence soit vide. Ainsi, la première étape du décodage de `100111111010001011101001000111` consiste à reconnaître que les deux premiers bits codent le caractère `'s'`, et on poursuit l'analyse avec la séquence restante `011111101000101111010001000111`.

9. Écrire une fonction `code_texte: char list -> arbre -> mot` qui prend en paramètres une liste `l` de caractères représentant le message à coder, et un arbre de Huffman `t`, et qui renvoie le mot binaire codant `l`.
10. Justifier que, quelque soit le mot `m` et l'arbre de Huffman `t` considérés, il existe au plus un entier k tel que les k premiers bits de `m` sont le code valide d'un caractère.
11. Écrire une fonction `decode_mot: mot -> arbre -> char * mot` qui prend en paramètres un mot `m` et un arbre de Huffman `t`, et qui renvoie une paire `(c, m')` composée du premier caractère `c` codé par `m`, et du mot `m'` restant à décoder après `c`. La fonction doit échouer si aucun préfixe de `m` n'est un code valide pour `t`.
12. Écrire une fonction `decode_texte: mot -> arbre -> char list` qui prend en paramètres un mot `m` et un arbre de Huffman `t`, et qui renvoie la liste des caractères représentés par le mot codé `m`. La fonction doit échouer si le mot codé `m` n'est pas une séquence valide.

3. Construction de l'arbre de Huffman

Pour construire un code de Huffman, on part de statistiques sur la fréquence de chaque caractère dans les textes que l'on souhaitera compresser. On peut par exemple simplement considérer le nombre d'occurrences de chaque caractère dans un grand texte représentatif de la langue utilisée dans les messages à compresser. Ici, on va donc utiliser un tableau `stats` d'entiers tel que `stats.(k)` contient le nombre d'occurrences du caractère dont le code ASCII est l'entier `k`. Note caml : étant donné un caractère `c` (type `char`), on obtient son code ASCII avec `Char.code(c)`. À l'inverse, étant donné un entier `k` de l'intervalle $[0, 256[$, on obtient le caractère correspondant avec `Char.chr(k)`.

On considère ensuite chacun de ces caractères `a` comme une feuille `C a`, et on va petit à petit regrouper les feuilles en petits arbres, puis les petits arbres en arbres plus grands, jusqu'à n'avoir plus qu'un arbre unique. À chaque étape on regroupe les deux arbres de plus faible poids, où le *poids* d'un arbre est la somme des occurrences de tous les caractères qu'il contient.

13. Si on fixe les nombres d'occurrences suivants

`c`	`'a'`	`'f'`	`'i'`	`'n'`	`'s'`	`'t'`
`stats.(Char.code(c))`	3	1	2	1	3	2

montrer, dans l'ordre, tous les regroupements qui sont faits pour former l'arbre exemple de la première page.

14. Écrire une fonction `poids: arbre -> int array -> int` qui prend en paramètre un arbre de Huffman `t` et un tableau `stats`, et qui renvoie le poids total de `t`.

Supposons que l'on dispose d'un type `'a prio` pour des files de priorité contenant des éléments d'un type générique `'a`, et des fonctions suivantes.

- `create: unit -> 'a prio`, qui crée une nouvelle file de priorité,
- `is_empty: 'a prio -> bool`, qui prend en paramètre une file de priorité, et qui renvoie `true` si et seulement si cette file est vide,
- `is_singleton: 'a prio -> bool`, qui prend en paramètre une file de priorité, et qui renvoie `true` si et seulement si cette file contient exactement un élément,
- `insert: 'a -> 'a prio -> unit`, qui prend en paramètre un élément `e` et une file de priorité `f`, et qui insère `e` dans `f` en tenant compte de la priorité de `e`,
- `extract_min: 'a prio -> 'a`, qui prend en paramètre une file de priorité `f` et qui retire de la file et renvoie l'élément le plus prioritaire.

On peut alors créer un arbre de Huffman à partir d'un tableau `stats`, en utilisant une file de priorité dans laquelle on stocke les arbres de Huffman que l'on doit regrouper.

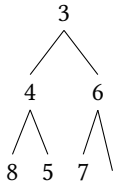
```
let huffman stats =
  let file = create() in
  Array.iteri
    (fun i occ -> if occ > 0 then insert (occ, C(Char.chr i)) file)
    stats;
  while not (is_singleton file) do
    assert (not (is_empty file));
    let p1, t1 = extract_min file in
    let p2, t2 = extract_min file in
    insert (p1+p2, N(t1, t2)) file
  done;
  snd (extract_min file)
```

15. Quel est le type de `file` dans ce code ?
16. Dans quelles situations ce code pourrait-il échouer ?

4. Tas de Braun

Comme vu en cours, on peut réaliser une file de priorité à l'aide d'une structure de tas binaire, et cette structure est efficace si les tas sont équilibrés. La version vue en cours (les *skew heaps*) maintient un équilibre sur le long terme, mais peut être ponctuellement déséquilibrée. Nous allons voir une structure alternative, les *tas de Braun*, qui garantit un équilibre permanent.

Un *arbre de Braun* est un arbre binaire dans lequel, pour tout nœud $\mathsf{N}(a_1, n, a_2)$, la taille de a_1 est égale à la taille de a_2 ou lui est de 1 supérieure. Un *tas de Braun* est un arbre de Braun qui a en plus la priopriété de tas (tout élément porté par un nœud est inférieur ou égal aux éléments portés par ses fils). Voici un tas de Braun avec six éléments.



On se donne le type caml

```
type 'a tas =
  | E
  | N of 'a * 'a tas * 'a tas
```

pour représenter un tas. Le tas de Braun précédent peut donc être représenté par la valeur caml

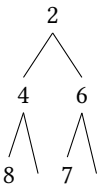
```
N(3, N(4, N(8, E, E), N(5, E, E)), N(6, N(7, E, E), E))
```

17. Où se situe l'élément minimal d'un tas ? Et l'élément maximal ?
18. Dans cette question on ne s'intéresse qu'à la forme de l'arbre, pas aux éventuelles valeurs portées par les nœuds. Combien y a-t-il d'arbres de Braun de taille 2 ? de taille 3 ? de taille 4 ? de taille 5 ? de taille 6 ? de taille 7 ? Donner pour chacun une représentation en caml.
19. Comment peut-on relier la taille et la hauteur d'un arbre de Braun ?

On propose la fonction suivante pour ajouter un élément x dans un tas de Braun t.

$$\begin{array}{rcll} \text{ajoute}(x, \mathsf{E}) & = & \mathsf{N}(x, \mathsf{E}, \mathsf{E}) & \\ \text{ajoute}(x, \mathsf{N}(n, t_1, t_2)) & = & \mathsf{N}(x, \text{ajoute}(n, t_2), t_1) & \text{si } x \leq n \\ \text{ajoute}(x, \mathsf{N}(n, t_1, t_2)) & = & \mathsf{N}(n, \text{ajoute}(x, t_2), t_1) & \text{sinon} \end{array}$$

20. Écrire une fonction ajoute : 'a -> 'a tas -> 'a tas transcrivant les équations précédentes en caml.
21. Appliquer cette fonction à l'élément 3 et au tas de Braun



22. Démontrer que la taille de ajoute(x, t) est exactement 1 de plus que la taille de t, par récurrence structurelle sur t. En déduire que le résultat de l'application de ajoute à un arbre de Braun est toujours un arbre de Braun.

Il est également possible de retirer certains éléments d'un tas de Braun tout en en préservant la structure.

23. Écrire une fonction `extrait_gauche: 'a tas -> 'a * 'a tas` qui prend en paramètre un tas de Braun et qui renvoie son élément le plus à gauche, ainsi qu'un tas de Braun contenant tous les autres éléments.
24. On propose la définition suivante pour une fonction fusionnant deux tas de Braun a et b, en supposant que a a une taille égale, ou de 1 supérieure, à la taille de b :

$$\begin{aligned}
\mathsf{fusion}(a, \mathsf{E}) &= a \\
\mathsf{fusion}(\mathsf{N}(n_a, a_1, a_2), \mathsf{N}(n_b, b_1, b_2)) &= \mathsf{N}(n_a, \mathsf{N}(n_b, b_1, b_2), \mathsf{fusion}(a_1, a_2)) \qquad \text{si } n_a \leq n_b \\
\mathsf{fusion}(\mathsf{N}(n_a, a_1, a_2), \mathsf{N}(n_b, b_1, b_2)) &= \mathsf{N}(n_b, \mathsf{fusion}(b_1, b_2), \mathsf{N}(n_a, a_1, a_2)) \qquad \text{sinon}
\end{aligned}$$

    Quel est le problème de cette définition ?
25. On peut régler le problème précédent en déplaçant un élément d'un sous-arbre à un autre dans le cas problématique. Définir une fonction `fusion: 'a tas -> 'a tas -> 'a tas` qui prend en paramètres deux tas de Braun `a` et `b`, avec `a` de taille égale, ou de 1 supérieure, à la taille de `b`, et qui renvoie un tas de Braun contenant tous les éléments de `a` et `b`.
    Indication : vous pourrez utiliser d'autres fonctions définies sur les tas de Braun aux questions précédentes.

Bonus. Codage et décodage améliorés

Voici quelques pistes pour rendre notre codage de Huffman plus efficace et plus réaliste. À réaliser à volonté, mais seulement si les versions de base fonctionnent. En réalisant ces fonctions, vous serez naturellement amenés à vous écarter des types imposés dans le début du sujet.
Attention : si la réalisation d'un de ces bonus vous amène à écrire une nouvelle version d'une des fonctions du début du sujet, conservez bien les deux versions dans votre fichier (celle d'origine à sa place, et la nouvelle à la fin du fichier).

- On peut accélerer le codage d'un texte en évitant les appels à répétition à la fonction `code_char`. Pour cela il suffit, à partir de l'arbre de Huffman, de précalculer un tableau `code`, qui à chaque caractère associe la séquence de bits qui le représente.
- Puisqu'on parle de textes et de caractères, on pourrait préférer que le texte à coder, et le résultat du décodage, soient de type `string` plutôt que `char list`.
  Note caml : pour construire une chaîne en y ajoutant progressivement des caractères, le module `Buffer` *apporte une solution efficace.*
- Pour que les opérations de codage et de décodage s'appliquent à des textes longs, il vaut mieux que les fonctions manipulant des listes soient récursives terminales.
- La compression de Huffman consiste à produire une séquence de bits. Pour pousser jusqu'au bout, il faudrait donc traduire le mot binaire de type `mot` (c'est-à-dire `int list`) que vous donne `code_texte` en une séquence de bits brute. Une solution pour cela consiste à regrouper les bits par 8 (c'est-à-dire par *octets*), à interpréter chaque groupe comme le code ASCII d'un caractère, et enfin construire la chaîne des caractères obtenus.
  Note : le nombre de bits obtenu après codage peut ne pas être un multiple de 8. Dans ce cas, on complète artificiellement le dernier octet. Et dans tous les cas, on accompagne la chaîne codée du nombre de bits significatifs qu'elle contient.
  Autre note : les caractères obtenus n'auront aucun lien apparent avec le texte d'origine, et certains ne seront pas même affichables. Ils ne valent que pour la séquence de bits qui forme leur code.
  À l'inverse, le décodage part d'une telle séquence brute, et doit en extraire la séquence de bits pour ensuite effectuer le décodage.
- Pour obtenir une application de compression de texte en ligne de commande, on peut vouloir lire le texte à coder dans un fichier, et écrire le résultat du codage dans un nouveau fichier. Et inversement pour la décompression.
- Pour obtenir le tableau `stats` initial, il faut analyser un texte de référence, par exemple lu dans un fichier.
- Plutôt que d'utiliser un code précalculé pour une langue donnée, on peut calculer un tableau `stats` directement à partir du texte à coder. Le code sera alors optimisé pour ce texte ! En revanche, pour permettre le décodage il faut, d'une manière ou l'autre, inclure l'arbre de Huffman dans le fichier binaire codé. Lors du décodage on commence alors par reconstruire l'arbre, puis on passe au décodage du texte lui-même.