TP n°26 - Algorithme de compression de Huffman

Nous l'avons vu, l'algorithme de **compression** de Huffman nécessite plusieurs étapes

1. Analyse de la donnée à compresser et comptage du nombre d'occurrences de chaque octet
2. Construction inductive de l'arbre de Huffman
3. Génération de la table d'encodage à partir de l'arbre de Huffman
4. Encodage de la donnée à l'aide de la table d'encodage

La **décompression** se fait à partir de l'arbre de Huffman utilisé pour encoder la donné. L'arbre de Huffman est par exemple transmis en même temps que la donnée compressée. En tout cas, il doit être disponible pour effectué la décompression.
On lit la donnée compressée donnée sous forme de bits, chaque bit indiquant le chemin à prendre dans l'arbre de Huffman (0 pour aller à gauche, 1 pour aller à droite). Lorsque l'on atteint un feuille, on a trouvé un code complet et décodé un caractère de la donnée originale. On recommence à « filtrer » les bits suivant en repartant tout en haut de l'arbre.

Toutes les fonctions de ce TP seront écrites en OCaml dans un fichier nommé `NOM_huffman.ml`.
Le module `Char` d'OCaml permet de récupérer facilement le code ASCII (entier positif entre 0 et 255) d'un caractère ou, à l'inverse, d'obtenir le caractère correspond à un certain code ASCII. La documentation de ce module est disponible en ligne :

https://v2.ocaml.org/api/Char.html

Exercice 1 (Étape 1 : Comptage du nombre d'occurrences de chaque caractère à l'aide d'un tableau associatif).

Nous l'avons vu, l'algorithme de Huffman débute par le décompte du nombre d'occurrences de chaque caractère dans la donnée à compresser.

Implémenter une fonction `huffman_count_occurrences: string -> int * int array` qui utilise un tableau associatif pour effectuer le comptage du nombre d'occurrences de chaque caractère (motif de 8 bits) dans une donnée interprétée comme une chaîne de caractères. La fonction renvoie un couple indiquant le nombre total de caractères de la chaîne et le tableau associatif donnant le nombre d'occurrences de chaque caractères ASCII.

On pourra, dans une deuxième version, utiliser la fonction `String.iter: (char -> unit) -> string -> unit` qui permet d'appliquer une fonction à valeur de sortie `unit` sur tous les caractères d'une chaîne de caractères (c'est une fonction d'ordre supérieur...)

Pour construire inductivement l'arbre de Huffman, nous avons vu que l'on commence par créer une forêt d'arbres réduits à une seule feuille étiquetée avec l'un des caractères présent dans la donnée. Puis on place tous les arbres de cette forêt dans une file de priorité, où l'on utilise le poids de chaque arbre (au début égal au nombre d'occurrences de chaque caractère) comme indicateur de priorité mais inversé : plus l'arbre a un poids faible, plus il sera prioritaire pour être utilisé dans la construction inductive.

Il nous faut donc une structure de file de priorité stockant des entrées de type clé-valeur, la clé indiquant le niveau de priorité de l'entrée. Normalement, vous avez implémenté les primitives d'une telle structure de données pour aujourd'hui en OCaml en utilisant une structure de données de tas implémentée par un tableau.
Téléchargez, sur la page Moodle de ce TP, le module `Pqueue` que j'ai implémenté pour vous dans un fichier `pqueue.ml`.
Pour utiliser ce module, il suffit d'utiliser la directive `#use` au début de votre script, suivi du nom du fichier dans lequel mon module est défini.
Vous disposez maintenant d'une **structure de données de file de priorité polymorphe mutable** `('k, 'v) pqueue`, avec les primitives de manipulation classiques :

```
type ('k, 'v) pqueue =
      {
        mutable size : int; (* taille courante de la file *)
        fun_is_priority_greater: 'k -> 'k -> bool; (* fonction de comparaison *)
        capacity : int; (* capacité max de la file, représentée par un tas stocké dans un tableau *)
        data : ('k*'v) array (* tableau dont les éléments sont des couples clé-valeur. La clé correspond au degré de priorit
      }
;;

val create : int -> 'k * 'v -> ('k -> 'k -> bool) -> ('k, 'v) pqueue

val enqueue : ('k, 'v) pqueue -> 'k * 'v -> unit

val dequeue : ('k, 'v) pqueue -> 'k * 'v

val is_empty : ('k, 'v) pqueue -> bool

val is_full : ('k, 'v) pqueue -> bool

val length : ('k, 'v) pqueue -> int
```

Nous l'avons vu, les structures de données de type tas, ABR, dictionnaire, file de priorité peuvent s'appliquer sur tout type de clés, à condition de définir une relation d'ordre total sur cet ensemble de clé. Pour créer une structure de données générique, polymorphe, on donne donc à la création de notre structure de données une fonction de comparaison prototype `'k -> 'k -> bool`. Cette fonction indique si la première clé est supérieure ou égale à la seconde. Elle servira dans toutes les fonctions qui nécessitent de faire des comparaisons sur les clés, notamment pour ajouter ou supprimer une entrée de la structure de données.

Exercice 2 (Étape 2 : Création de l'arbre de Huffman associée à la donnée).

1. Téléchargez le fichier `pqueue.ml` disponible sur Moodle, allez voir son contenu et appropriez-vous le module `Pqueue` pour pouvoir l'utiliser ensuite.
2. Déroulez à la main l'algorithme de création de l'arbre de Huffman optimal sur la chaîne de caractères `abracadabra`.
3. Réécrire en pseudo code sur papier l'algorithme de création de l'arbre de Huffman optimal associé à une chaîne de caractères. Obligatoire!! Ne foncez pas sur le code!
4. Proposez un type OCaml `hufftree` permettant d'implémenter une structure d'arbre de Huffman.
5. Écrire une fonction `huffman_build_tree: string -> hufftree` générant l'arbre de Huffman associé à la chaîne de caractères que l'on souhaite compresser.
6. Validez votre fonction sur l'exemple déroulé plus haut et sur l'exemple du cours `scienceinformatique`

OCaml propose un module `Hashtbl` implémentant une structure de données de table de hachage à travers le type polymorphe `('a, 'b) Hashtbl.t`. Là encore, le polymorphisme est double, sur les clés et sur les valeurs : `'a` est le type des clés, `'b` celui des valeurs associées. L'implémentation des primitives de manipulation de tables de hachage est détaillée dans la documentation

https://v2.ocaml.org/api/Hashtbl.html

Exercice 3 (Étape 3 : Création de la table d'encodage).

1. Écrire une fonction `huffman_build_encoding_table: string -> hufftree * (char, string) Hashtbl.t` , qui, à partir d'une chaîne de caractères, renvoie un couple formé de l'arbre de Huffman et de la la table d'encodage sous la forme d'un dictionnaire associant chaque caractère à une `string` formée de '0'et de '1'correspondant à son code.

   On rappelle que l'opérateur de concaténation de chaînes de caractères s'écrit `^` en OCaml.
2. Écrire une fonction d'affichage `print_encoding_table` permettant d'afficher proprement la table d'encodage. On pourra utiliser une fonction bien pratique du module `Hashtbl`.

Le module `Hashtbl` est l'occasion d'introduire une nouvelle syntaxe OCaml, qui est au programme de MPI : le type `'a option`. Ce module permet de gérer de manière explicite et propre la présence ou l'absence d'une valeur, sans manipuler des exceptions.

Le type `'a option` est un type somme à deux constructeurs :

```
type 'a option = None | Some of 'a;;
```

Il permet de gérer le cas de valeurs indéfinies ou d'absence de valeurs sans avoir recours aux exceptions, un peu à la manière de `None` en Python.

Voici un exemple d'utilisation en OCaml, qui généralise une fonction d'addition de deux valeurs dans le cas où l'une des valeurs est non définie :

```
# let a = Some(3);;
val a : int option = Some 3
# let b = None;;
val b : 'a option = None
# let c = Some(4);;
val c : int option = Some 4
# let add_opt a b =
  match (a,b) with
  |(None, _) -> None
  |(_, None) -> None
  |(Some(va), Some(vb)) -> Some(va+vb);;
val add_opt : int option -> int option -> int option = <fun>
# add_opt a b;;
- : int option = None
# add_opt a c;;
- : int option = Some 7
```

Dans le module OCaml `Hashtbl`, il existe une fonction `Hashtbl.find_opt` qui utilise ce type optionnel et renvoie un objet de type `'a option` : si la clé cherchée n'est pas présente dans la table, la fonction renvoie un objet de type `'a option` construit avec le constructeur `None`. Sinon, elle renvoie un objet construit avec le constructeur `Some` avec, comme argument, la valeur associée à la clé trouvée dans la table.

```
(* creation table de hachage à 7 seaux *)
let ht = Hashtbl.create 7;;
val ht : ('_a, '_b) Hashtbl.t = <abstr>
#
(* ajout du couple clé-valeur ('c', 3) *)
Hashtbl.add ht 'd' 3;;
- : unit = ()
#
(* ajout du couple clé-valeur ('a', 5) *)
Hashtbl.add ht 'a' 5;;
- : unit = ()
# ht;;
- : (char, int) Hashtbl.t = <abstr>
#
(* recherche de la clé 'a' dans la table *)
Hashtbl.find_opt ht 'a';;
- : int option = Some 5
#
(* recherche de la clé 'f' dans la table, non présente*)
Hashtbl.find_opt ht 'f';;
- : int option = None
```

Exercice 4 (Étape 4 : Fonction de compression).

1. Écrire une fonction `huffman_compress: string -> string*hufftree`, qui compresse une chaîne de caractères en utilisant l'algorithme de Huffman et renvoie la série binaire associée à la chaîne compressée sous la forme d'une chaîne de caractères 0 et 1 ainsi que l'arbre de Huffman associé à la compression, qui sera indispensable pour la décompression.

   On cherchera un caractère dans la table d'encodage en utilisant la fonction `Hashtbl.find_opt` et en gérant proprement les cas d'erreurs par l'analyse du type `'a option` renvoyé.

   On utilisera, dans cette première version, l'opérateur de concaténation de chaînes `^` et on utilisera une référence sur un chaîne de caractère pour rendre notre chaîne de caractères mutable.
2. Adaptez votre fonction pour qu'elle **renvoie et affiche** également le taux de compression. Vérifiez la justesse de votre calcul et l'interprétation que vous en faites en comparant les tailles des chaînes originale et compressée.

La bibliothèque standard d'OCaml ne fournit pas de structure de tableau redimensionnable générique, c'est-à-dire capable de manipuler des éléments d'un type quelconque, mais fournit en revanche un module `Buffer` de tableaux redimensionnables dont les éléments sont des octets (un octet se dit *byte* en anglais), c'est-à-dire des caractères.
Avec ce module, on peut construire de grandes chaînes par concaténations successives, de caractères ou de chaînes. Une fois la construction terminée, on peut récupérer la chaîne complète, de type `string`.
Plus d'informations sont disponibles sur le site :

https://v2.ocaml.org/api/Buffer.html

Exercice 5 (Étape 5 : Fonction de décompression).

1. Sauvegardez la fonction `huffman_compress: string -> string` en une fonction `huffman_compress_old: string -> (string, hufftree)`. Modifiez la fonction `huffman_compress: string -> string` en utilisant cette fois le module `Buffer` plutôt que l'opérateur de concaténation de chaînes `^`. Expliquez pourquoi cette nouvelle implémentation est meilleure que la précédente.
2. Continuez l'amélioration en utilisant la fonction `String.iter` plutôt qu'une boucle pour l'encodage caractère par caractère de la chaîne originale.
3. Écrire une fonction récursive `huffman_uncompress: (string, hufftree) -> string` qui décompresse une donnée compressée à partir de l'arbre de Huffman qui a été utilisé pour cette compression. *Indication : pour concaténer un caractère en fin de chaîne avec `^`, il faudra d'abord le convertir en `string` en utilisant `String.make`.*
4. Essayez de rendre cette fonction récursive terminale si ce n'est pas déjà le cas.
5. Sauvegarder votre fonction de décompression `huffman_uncompress` en la copiant et en la renommant `huffman_uncompress_old` puis adaptez votre fonction `huffman_uncompress` pour qu'elle utilise le module `Buffer`