

Objectif

L'objectif de ce TP est de faire un programme `mp2zip.exe` qui nous permettra d'exécuter :

- `mp2zip comp foobar` pour compresser un fichier `foobar` en un fichier `foobar.mp2zip` via l'algorithme de Huffman ;
- `mp2zip decomp foobar.mp2zip` pour décompresser un fichier précédemment compressé par notre programme.

Modules

Pour cela, on va découper le TP en plusieurs sous-parties qui seront codées pour certaines dans des **modules**. On a déjà vu les modules en OCaml quand on a utilisé par exemple `List`, `Printf` ou encore `Graphics`. Pour simplifier, on va utiliser `dune` ici et en compilant le projet, si on a un fichier `monfichier.ml` on pourra accéder aux fonctions et types définis avec `Monfichier.mafonction`. Tout comme en C, on peut donc travailler avec plusieurs fichiers, mais il n'y a pas besoin de faire de fichiers `.h` ou de `#include`.

Remarque. On pourrait vouloir cacher une partie de l'interface aux appelants, comme on peut le faire avec un `.h` qui ne rend pas tout visible. C'est possible en OCaml, on crée pour ça un fichier `.mli` contenant l'interface. Ici, on bénéficie juste d'un mécanisme de OCaml pour générer des interfaces maximales par défaut.

Vous trouverez sur la page web du cours un fichier `mp2zip.zip` contenant :

- un fichier `mp2zip.ml` qui sera le programme principal et qui appellera les fonctions des autres modules ;
- un fichier `dune` et un fichier `dune-project` permettant de compiler votre projet avec la commande `dune build`.

Remarque. Si vous avez une erreur `PROJECT ROOT`, ouvrez le fichier `dune-project` et changez la valeur `2.9` en `3.0`.

Organisation du projet

On va avoir les modules suivants :

- `Heap` (donc un fichier `heap.ml`) permettant de gérer un tas max dans un tableau comme on l'a vu en cours ;
- `PriorityQueue` (donc `priorityQueue.ml`, attention à la majuscule à Q) permettant de gérer un file de priorité avec un tas.

Modules définissant une structure de données

Quand on écrit un module définissant une implémentation de structure de données, il est d'usage de nommer le type `t` ainsi si le module est `MaStructure` le type sera en fait `MaStructure.t` qui a le mérite d'être léger.

Exercice 1 (fichier `heap.ml`) : On va définir un tas max, et pour cela on va utiliser le type suivant :

```
type 'a t = {
  elts : 'a array;
  mutable length : int
}
```

Écrire les fonctions suivantes :

1. `make` : `int` -> `'a` -> `'a t` où `make n v` crée un tas vide de taille maximale `n` en utilisant la valeur `v` comme valeur par défaut dans le tableau.
2. `father` : `int` -> `int` qui étant donné un indice $i > 0$ renvoie l'indice de son père.
3. `left` : `int` -> `int` qui étant donné un indice renvoie celui de son fils gauche.
4. `right` : `int` -> `int` qui étant donné un indice renvoie celui de son fils droit.
5. `nth` : `'a t` -> `int` -> `'a` qui renvoie l'élément d'indice n dans le tas.
6. `length` : `'a t` -> `int` qui renvoie la taille d'un tas.
7. `add` : `'a` -> `'a t` -> `unit` qui ajoute un élément au tas. Si le tas est plein, on lèvera une exception `Overflow` définie dans le module avec `exception Overflow`.
8. optionnelle mais peut être utile pour la suivante, `max_with_sons` : `'a t` -> `int` -> `'a` qui renvoie la valeur maximale entre l'élément d'indice i et ses deux fils (dans l'optique où on a cassé le tas dans une suppression).
9. `take` : `'a t` -> `'a` qui retire et renvoie l'élément maximal du tas. Si le tas est vide, on lèvera une exception `Empty` définie dans le module avec `exception Empty`.
10. `peek` : `'a t` -> `'a` qui renvoie l'élément maximal du tas **sans l'enlever**.

Exercice 2 (fichier `priorityQueue.ml`) : On va définir un type file de priorité où les priorités seront des entiers et où la stratégie sera de retirer l'élément de plus grande priorité. Bien entendu, on utilisera un tas. Le type est

```
type 'a t = (int * 'a) Heap.t
```

En comptant sur l'ordre lexicographique naturel qui va ordonner sur la première composante les couples, on peut directement réutiliser un tas.

On vous demande d'écrire, entre autres, les fonctions suivantes :

1. `make` : `int` -> `'a` -> `'a t` où `make n v` crée une file de priorité vide de taille maximale `n` en utilisant la valeur `v` comme valeur par défaut pour le tas.
2. `length` : `'a t` -> `int` qui renvoie la taille de la file de priorité.
3. `add` : `'a` -> `int` -> `'a t` -> `unit` qui ajoute un élément au tas avec une priorité donnée. On lèvera une exception `Overflow` si la file est pleine. On pourra se contenter du `Heap.Overflow` précédent ou rajouter une exception propre à ce module en rattrapant l'exception précédente si ça ne vous fait pas peur.
4. `take_full` : `'a t` -> `int * 'a` qui retire l'élément de plus grande priorité et renvoie le couple (priorité, élément). On lèvera une exception `Empty` avec la même remarque que précédemment.
5. `take` : `'a t` -> `'a` qui retire l'élément de plus grande priorité et le renvoie. C'est juste un cas particulier de la fonction précédente quand on ne veut pas avoir la priorité.
6. `peek` : `'a t` -> `'a` qui renvoie l'élément de plus grande priorité **sans le retirer**.

Exercice 3 (fichier `fileIO.ml`) : Ce fichier est déjà fourni dans l'archive, mais si vous finissez le TP en avance, vous pouvez essayer de le recoder vous-même.

Ce module permet de lire et d'écrire dans des fichiers bit par bit, plutôt qu'uniquement octet par octet. On dispose ainsi de deux nouveaux types de fichiers `in_channel_bits` et `out_channel_bits`.

On présente ici les fonctions de lecture :

1. `open_in_bits` : `string` -> `in_channel_bits` qui ouvre un fichier en lecture bit à bit.
2. `input_byte` : `in_channel_bits` -> `int` qui lit un octet dans le fichier comme si c'était un fichier normal.
ATTENTION : à n'utiliser qu'avant de faire des lecture bit à bit.
3. `input_bit` : `in_channel_bits` -> `bool` qui lit un bit sous la forme d'un booléen.
4. `close_in_bits` : `in_channel_bits` -> `unit` qui ferme le fichier.

Et d'écriture :

- `open_out_bits` : `string` -> `out_channel_bits` qui ouvre en écriture bit à bit un fichier.
- `output_byte` : `out_channel_bits` -> `int` -> `unit` qui écrit un octet dans le fichier comme si c'était un fichier normal.
ATTENTION à n'utiliser qu'avant de faire des écritures bits à bits.
- `output_bit` : `out_channel_bits` -> `bool` -> `unit` qui écrit un bit dans le fichier.
- `close_out_bits` : `out_channel_bits` -> `unit` qui ferme le fichier.

On donne également un fonction permettant d'appliquer une fonction sur chaque octet d'un fichier :

- `input_byte_iter` : `string` -> (`int` -> `unit`) -> `unit` qui va ouvrir le fichier du nom donné et appliqué sur chaque octet la fonction passée en paramètre.

Exercice 4 (fichier `huffman.ml`) : Ce fichier va contenir les fonctions liées à la création et l'utilisation du code de Huffman. On va utiliser le type suivant pour l'arbre :

```
type tree = Char of int | Node of tree * tree
```

On identifie ainsi complètement un caractère à l'octet, et donc l'entier entre 0 et 255, qui le représente.

Les occurrences de caractères seront donnés dans un tableau `int array` de 256 valeurs, où la i -ème valeur donne la fréquence du caractère de numéro i . On aura alors des valeurs nulles pour les caractères non présents, elles seront ignorées lorsqu'on va construire l'arbre.

1. `build_tree` : `int array` -> `tree` qui renvoie l'arbre de Huffman pour un tableau d'occurrences en utilisant l'algorithme glouton.
2. `codes` : `tree` -> `bool list array` qui renvoie un tableau de 256 listes de booléens donnant le code de chacun des caractères avec la convention `true` = 1 = droite et `false` = 0 = gauche. On les donnera dans l'ordre de la racine vers la feuille, donc il faudra surement faire un `List.rev` sur l'accumulateur une fois la feuille atteinte.

On a besoin de stocker ou lire l'arbre de Huffman dans le fichier de manière simple. Comme on va le faire dans le même fichier que les données compressées, on utilise les types de fichiers bit à bit.

On vous propose le codage suivant :

- une feuille `Char` c est codée par un octet 1 suivi de l'octet c ;
- un noeud `Node`(x, y) est codé par un octet 0 suivi du codage de x puis du codage de y .

Il s'agit en fait de coder l'ordre préfixe à plat. Comme l'arbre est binaire strict, cela suffit à le reconstruire uniquement.

Ce codage se traduit par deux fonctions :

3. `output_tree` : `out_channel_bits` -> `tree` -> `unit` qui écrit l'arbre.
4. `input_tree` : `in_channel_bits` -> `tree` qui lit l'arbre.

Exercice 5 (fichier `compress.ml`) : Pour compresser, on va adopter la stratégie suivante :

- on calcule le tableau de fréquences d'un fichier ;
- on en déduit son arbre de Huffman ;
- on écrit dans la sortie l'arbre de Huffman ;
- on calcule les codes de chaque octet ;
- pour chaque octet du fichier d'entrée on écrit le code correspondant dans la sortie.

Ainsi, on va écrire les fonctions suivantes :

1. `stats` : `string -> int array` qui renvoie le tableau de fréquences associé au contenu du fichier dont le nom est passé en argument.
On utilisera la fonction auxiliaire d'itération sur les octets donnée dans `FileIO`.
2. `output_code` : `out_channel_bits -> bool list -> unit` qui écrit un code dans le fichier bit à bit.
3. `compress` : `string -> unit` qui va adopter la stratégie précédente pour écrire le fichier compressé.
Si le nom du fichier est `fn` on écrira dans `fn ^ ".mp2zip"`.

Exercice 6 (fichier `decompress.ml`) : Pour décompresser, on va adopter la stratégie suivante :

- on lit l'arbre de Huffman ;
- tant que le fichier en entrée n'a pas été intégralement parcouru, on décode un caractère à l'aide de l'arbre et on l'écrit dans la sortie.

Pour la boucle de lecture dans le fichier, on pourra utiliser la construction suivante :

```
try
  while true do
    (* lecture du fichier *)
  done
with End_of_file -> ()
```

1. Écrire une fonction `input_code` : `in_bits_channel -> Huffman.tree -> int` qui lit un octet dans le fichier depuis son code compressé en utilisant l'arbre de Huffman en argument.
2. Écrire une fonction `decompress` : `string -> unit` qui va adopter la stratégie précédente pour écrire le fichier décompressé. Si le nom du fichier est `monfichier.mp2zip` on écrira dans `String.sub fn 0 (String.length fn - 7)` pour obtenir `monfichier`.

Exercice 7 (programme principal) : Ici, on va modifier `mp2zip` pour réaliser la compression ou décompression. Pour cela, on va utiliser le tableau `Sys.argv` qui contient les paramètres passés sur la ligne de commande. Ce tableau contient toujours le nom de l'exécutable à l'indice 0 puis les arguments passés.

Si on passe moins de 2 arguments ou que le premier argument n'est pas `"comp"` ou `"decomp"`, on affichera une ligne de description de l'usage du programme.

Sinon, on effectuera la compression ou la décompression selon les fonctions précédemment définies.

Exercice 8 (tests) : Essayez de compresser un peu tout. On pourra notamment essayer de compresser le fichier `proust.txt` contenant l'intégrale de Proust, qu'on pourra lire s'il vous reste du temps.