TME5 – Arbres de Huffman

Exercice 5.5 (Codage de Huffman).
On cherche à coder, par une liste de 0 et de 1 aussi courte que possible, un message comportant n symboles différents. Un moyen d'y parvenir est d'utiliser la méthode du codage de Huffman qui prend en compte la fréquence de chaque symbole apparaissant dans le message afin d'associer les codes les plus courts aux symboles les plus fréquents. Par exemple, lors du codage de la suite `AABACBAGHAAFEADBA`, on souhaite que le code du caractère `A` soit le plus court puisqu'il apparaît 8 fois, et que le code du caractère `B` qui apparaît 3 fois soit plus court que les codes des caractères `C`, `D`, `E`, `F`, `G` et `H` qui n'apparaissent qu'une fois.

Le code de chaque symbole est obtenu à partir d'un arbre binaire, appelé arbre de Huffman, construit à partir d'une liste de fréquences des symboles. Les feuilles de cet arbre sont étiquetées avec les symboles. Un *chemin* de la racine de l'arbre à une feuille étiquetée par un symbole est représenté par une suite de 0 et de 1 : 0 lorsque l'on accède au sous-arbre gauche ; 1 lorsque l'on accède au sous-arbre droit. Le code associé à un symbole est le chemin de la racine de l'arbre vers ce symbole.

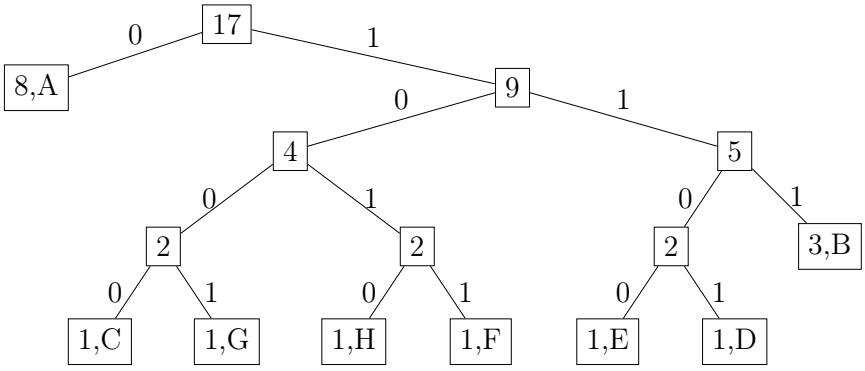
On définit pour cet exercice un type particulier pour les arbres binaires :

```
type 'a htree =
  | Leaf of int * 'a
  | Branch of int * 'a htree * 'a htree
```

La variable de type `'a` représente le type des symboles. Le paramètre de type `int` de chaque constructeur est calculé à partir de la table de fréquence des symboles. Aux feuilles (constructeur `Leaf`) cet entier désigne la fréquence du symbole ; aux nœuds (constructeur `Branch`) cet entier désigne la somme des entiers qui étiquettent la racine des deux sous-arbres. Par exemple, la table des fréquences :

```
[('A', 8); ('B', 3); ('D', 1); ('E', 1); ('F', 1); ('H', 1); ('G', 1);('C', 1)]
```

associée au message `AABACBAGHAAFEADBA` permet d'obtenir l'arbre de Huffman ci-dessous (noté `t_msg` dans la suite de cet exercice) :



La table des codes associés aux caractères `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`, `'G'` et `'H'` est alors la suivante :

'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'
0	111	1000	1101	1100	1011	1001	1010

Dans cet exercice, un code est représenté par une liste d'entiers : sur l'exemple ci-dessus, le code du caractère `'B'` est la liste `[1;1;1]`.

1. Définir une fonction de signature `huff_tab (t:'a htree) : (('a * (int list)) list` qui étant donné un arbre de Huffman `t` retourne la liste d'association contenant les paires (c, h) où c est un symbole apparaissant sur une feuille de `t` et h est le code associé à c par `t`.

```
# huff_tab t_msg;;
- : (char * int list) list =
[('A', [0]); ('C', [1; 0; 0; 0]); ('G', [1; 0; 0; 1]); ('H', [1; 0; 1; 0]);
 ('F', [1; 0; 1; 1]); ('E', [1; 1; 0; 0]); ('D', [1; 1; 0; 1]);
 ('B', [1; 1; 1])]
```

2. Définir une fonction de signature

   `code (m : 'a list) (c : ('a * (int list)) list) : int list`

   qui construit le code du message `m` à partir de la liste d'association `c` du codage de Huffman.

```
# let lc_msg = huff_tab t_msg in
  let msg =
    ['A';'A';'B';'A';'C';'B';'A';'G';'H';'A';'A';'F';'E';'A';
     'D';'B';'A']
  in code msg lc_msg;;
- : int list = [0; 0; 1; 1; 1; 0; 1; 0; 0; 0; 1; 1; 1; 0; 1; 0; 0; 1; 1; 0; 1;
                0; 0; 0; 1; 0; 1; 1; 1; 1; 0; 0; 0; 1; 1; 0; 1; 1; 1; 1; 0]
```

3. Définir une fonction de signature

   `decode1 (l : int list) (t : 'a htree) : ('a * (int list))`

   telle que si `l` est le code d'un message et `t` est l'arbre de Huffman qui a servi pour son codage, alors `(decode1 l t)` construit le couple formé du premier symbole codé de `l` et du reste du message codé privé du code de ce premier symbole.

```
# decode1  [0; 0; 1; 1; 1] t_msg;;
- : char * int list = ('A', [0; 1; 1; 1])
```

```
# decode1 [1; 1; 1] t_msg;;
- : char * int list = ('B', [])
```

4. Définir une fonction de signature `decode (l : int list) (t : 'a htree) : 'a list` qui décode le message `l` en utilisant l'arbre de Huffman `t`.

```
# decode [0;0;1;1;1;0;1;0;0;0;1;1;1;0;1;0;0;1;1;0;1;0;0;0;1;0;
          1;1;1;1;0;0;0;1;1;0;1;1;1;1;0] t_msg;;
- : char list = ['A'; 'A'; 'B'; 'A'; 'C'; 'B'; 'A'; 'G'; 'H'; 'A'; 'A'; 'F';
                 'E'; 'A'; 'D'; 'B'; 'A']
```

L'arbre de Huffman est construit à partir de la liste d'association des fréquences de chaque symbole selon le principe de suivant :

— construction d'une liste d'arbres à partir de la liste des fréquences des symboles : chaque couple `(s,n)` de la liste est transformé en une feuille `Leaf(n,s)` (fonction `leaf_list`)

— puis, tant que la liste d'arbres contient au moins deux éléments, on applique le principe de «réduction» suivant :

1. sélectionner deux arbres d'étiquette minimale à la racine et les retirer de la liste (fonctions `ht_less` et `min_sauf_min`)
2. fusionner ces deux arbres (fonction `ht_branch`)
3. remplacer dans la liste les deux arbres minimaux par l'arbre obtenu par les fusionnant

lorsque la liste ne contient plus qu'un seul arbre, cet arbre est l'arbre de Huffman cherché (fonction `make_huff`)

5. Définir une fonction de signature `freq_ht (t : 'a htree) : int` qui retourne la fréquence associée à la racine de l'arbre de Huffman `t`.

```
# freq_ht (Leaf(3,'B'));;          # freq_ht (Branch(5,Leaf(2,'Z'),Leaf(3,'B')));;
- : int = 3                        - : int = 5
```

6. Définir une fonction de signature `ht_less (t1 : 'a htree) (t2 : 'a htree) : bool` qui détermine si la fréquence associée à la racine de l'arbre `t1` est strictement inférieure à la fréquence associée à la racine de l'arbre `t2`.

```
# ht_less (Leaf(3,'B')) (Branch(5,Leaf(2,'Z'),Leaf(3,'B')));;
- : bool = true
# ht_less (Branch(5,Leaf(2,'Z'),Leaf(3,'B'))) (Leaf(3,'B'));;
- : bool = false
```

7. Définir une fonction de signature

```
min_sauf_min (lt : ('a htree) list) : ('a htree) * (('a htree) list)
```

qui étant donnée une liste `lt` d'arbres de Huffman retourne une paire (a, r) où a est un arbre de `lt` dont la fréquence associée à sa racine est minimale (dans `lt`) et r est la liste `lt` privée de a. On supposera que la liste `lt` n'est pas vide (on pourra lever l'exception `Invalid_argument` lorsque `lt` est vide).

```
# min_sauf_min [Branch(5,Leaf(2,'Z'),Leaf(3,'W')); Leaf(3,'B'); Leaf(3,'K')];;
- : char htree * char htree list =
   (Leaf (3, 'K'), [Branch (5, Leaf (2, 'Z'), Leaf (3, 'W')); Leaf (3, 'B')])
```

8. Définir une fonction de signature `ht_branch (t1:'a htree) (t2:'a htree):'a htree` qui étant donnés deux arbres de Huffman `t1` et `t2` retourne un nouvel arbre de Huffman dont la racine est étiquetée par la somme des fréquences associées aux racines de arbres `t1` et `t2` et admettant `t1` pour fils gauche et `t2` pour fils droit.

```
# ht_branch (Branch(5, Leaf(2, 'Z'), Leaf(3, 'W')))  (Leaf(3, 'B'));;
- : char htree =
   Branch (8, Branch (5, Leaf (2, 'Z'), Leaf (3, 'W')), Leaf (3, 'B'))
```

9. Définir une fonction de signature `make_huff (lt:('a htree) list):'a htree` qui construit l'arbre de Huffman obtenu en appliquant le principe de «réduction» donné en introduction de ce paragraphe sur la liste d'arbres de Huffman `lt`. Par exemple, l'arbre de Huffman `t_msg` peut être obtenu comme suit.

```
# make_huff
  [Leaf (8,'A'); Leaf (3,'B'); Leaf (1,'D'); Leaf (1,'E'); Leaf (1,'F');
   Leaf (1,'H'); Leaf (1,'G'); Leaf (1,'C')];;
  - : char htree =
Branch (17, Leaf (8, 'A'),
            Branch (9,Branch (4, Branch (2, Leaf (1, 'C'), Leaf (1, 'G')),
                                 Branch (2, Leaf (1, 'H'), Leaf (1, 'F'))),
                      Branch (5, Branch (2, Leaf (1, 'E'), Leaf(1, 'D')),
                                 Leaf (3, 'B'))))
```

10. Définir une fonction de signature `leaf_list (f : ('a * int) list) : ('a htree) list` qui construit la liste des feuilles `Leaf(n,x)` obtenues à partir des couples `(x,n)` de la liste d'association `f` des fréquences de chaque symbole.

```
# leaf_list [('A', 8); ('B', 3); ('D', 1); ('E', 1); ('F', 1); ('H', 1);
             ('G', 1);('C', 1)];;
- : char htree list =
[Leaf (8, 'A'); Leaf (3, 'B'); Leaf (1, 'D'); Leaf (1, 'E'); Leaf (1, 'F');
 Leaf (1, 'H'); Leaf (1, 'G'); Leaf (1, 'C')]
```

11. Définir une fonction de signature `huff_of_freq (f : ('a * int) list) : 'a htree` qui construit un arbre de Huffman obtenue à partir de la liste d'association `f` des fréquences de chaque symbole. Par exemple, l'arbre de Huffman `t_msg` peut être obtenu en évaluant l'expression :

    ```
    huff_of_freq
      [('A',8); ('B',3); ('D',1); ('E',1); ('F',1); ('H',1); ('G',1); ('C',1)]
    ```

12. Définir une fonction de signature `make_code (m : 'a list) : (int list) * 'a htree` qui étant donné un message `m` construit le couple (c, t) où t est un arbre de Huffman construit à partir des fréquences des symboles de `m` et c est le codage de `m` obtenu en utilisant l'arbre t. On pourra utiliser la fonction `freq` de l'exercice **??**, de signature `freq (l : 'a list) : ('a * int) list`.

Remarque. C'est la manière de programmer `min_sauf_min` et `make_huff` qui détermine la forme de l'arbre de Huffman construit qui dépend du choix des deux arbres de poids minimaux (lesquels choisir en cas d'égalité ? ou placer dans la liste l'arbre résultant de la fusion ?). Ces choix sont indifférents (i.e., il existe plusieurs solutions correctes possibles) et il existe donc plusieurs arbres de Huffman possibles.