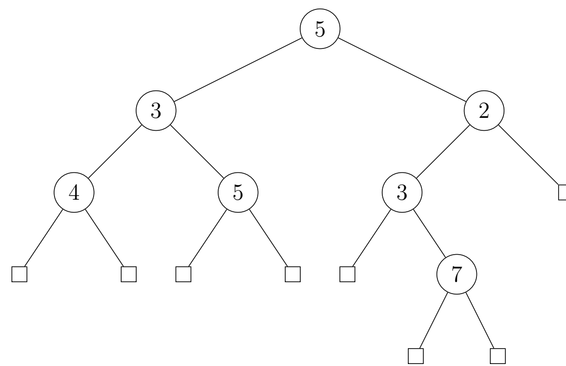
Feuille d'exercices n°5

Cette feuille d'exercice utilise les arbres binaires définis par:

```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree
```

pour les exercices I, II et III. L'exercice IV utilise une variante de la repésentation des arbres binaires non vides.

Exercice I : Exploration



On appelle *profondeur* d'une étiquette la distance du nœud qui la porte par rapport à la racine; la *distance* étant le nombre de noeud à traverser pour atteindre l'étiquette visée. L'étiquette de la racine est de profondeur 0. Sur la figure ci-dessus, la valeur 5 est à la fois à la profondeur 0 et à la profondeur 2.

Q1 – Définir la fonction

```
at_prof (n:int) (x:'a) (bt:'a btree) : bool
```

qui donne `true` si et seulement si l'étiquette `x` est présente à la profondeur `n` dans `bt`.

Q2 – Définir la fonction

```
at_prof_list (n:int) (bt:'a btree) : 'a list
```

qui donne la liste des étiquettes présentes à la profondeur `n` dans `bt`.
Sur l'exemple de notre figure, la fonction `at_prof_list` donne la liste `[4;5;3]` pour la profondeur 2.

Q3 – Définir la fonction

```
etiq_prof_list (x:'a) (bt:'a btree) : int list
```

qui donne la liste des profondeurs auxquelles est présente l'étiquette `x` dans `bt`.
Sur l'exemple de notre figure, `etiq_prof_list` donne la liste `[1;2]` pour l'étiquette 3.

Remarque: vous aurez besoin d'une fonction qui prend en paramètre une liste d'entiers et ajoute 1 à chacun d'entre eux. Vous pouvez: soit directement définir cette fonction; soit utiliser l'itérateur `List.map`.

Q4 – Définir la fonction

```
prof_max (bt:'a btree) : int
```

qui donne la profondeur maximale des étiquettes de l'arbre `bt`. Par convenance, la fonction donne -1 si l'arbre est vide.
Sur l'exemple de la figure ci-dessus, la valeur obtenue sera 3, qui est la profondeur du nœud étiqueté avec 7.

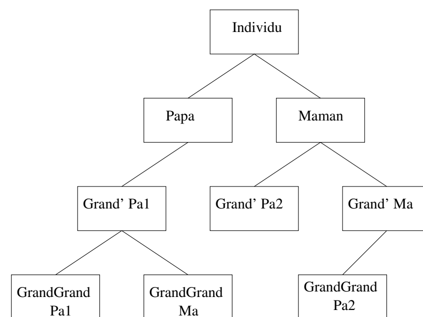
Q5 – Définir la fonction

```
max_prof_etiq (x:'a) (bt:'a btree) : int
```

qui donne la profondeur maximale de l'étiquette `x` dans `bt`. Par convenance, la fonction donne -1 si `x` n'apparaît pas dans `bt`.
Sur l'exemple de la figure ci-dessus, `max_prof_etiq` donnera 2 pour l'étiquette 5 et 3 pour l'étiquette 7.

Exercice II : Application: arbres de parenté

On utilise les arbres binaire pour représenter la structure de parenté d'un individu:



Vous noterez que les générations allant, l'information peut être incomplète.

Avec cette représentation, les parents sont situés à profondeur 1, les grands parents à profodeur 2, etc.

Q1 – En utilisant la fonction `at_prof_list` de l'exercie précédent, donnez la définition de la fonction

```
grands_parents (x:'a) (ag:'a list) : ('a list)
```

qui donne la liste des grands parents de l'élément `x` dans l'arbre `ag`. La fonction donne la liste vide si `x` n'est pas une étiquette dans `ag`.

Sur l'arbre de la figure ci-dessus, la fonction donne la liste `[Grand'Pa1; Grand'Pa2; Grand'Ma]` pour `Individu`; elle donne `[GrandGrandPa; GrandGrandMa]` si le paramètre `x` vaut `Papa`.

Exercice III : Les sources du Nil

Parmi les critères envisagés pour déterminer la source d'un fleuve, on puet en siter deux

1. La source se situe au point le plus éloigné de l'embouchure.
2. La source se trouve en remontant de puis l'embouchure en suivant le plus fort débit.

Ces critères correspondent à deux manières de calculer une branche dans un arbre binaire:

1. Calculer la plus longue branche.

2. calculer la branche déterminée par la plus grande valeur d'une étiquette.

Q1 – Définir la fonction

```
max_length_branch (bt:'a btree) : 'a list
```

qui donne une branche (liste d'étiquettes) de longue de l'arbre `bt`. Si l'arbre contient plusieurs branche de même longueur, on choisi la branche la plus à gauche.

Remarque: on peut utiliser la fonction `List.length` pour connaître la longueur d'une liste.

Q2 – Utiliser la fonction `List.length` pour déterminer la longueur d'une branche dans le calcul de la barnche la plus longue amène à refaire plusieurs fois le même parcours de liste. On peut améliorer les choses en définissant une fonction auxiliaire locale qui calcule à la fois la longueur et le contenu d'une branche.

Donner une nouvelle définition de

```
max_length_branch (bt:'a btree) : 'a list
```

qui utilise une fonction auxiliaire locale de signature

```
loop (bt:'a btree) : (int * 'a list)
```

.

Q3 – Définir la fonction

```
max_flow_branch (bt:'a btree) : 'a list
```

qui donne une branche dite *de flot maximal* de l'arbre `bt`.

Une branche *de flot maximal* s'obtient en sélectionnant, à chaque nœud le sous arbre dont l'étiquette est la plus grande. Pour simplifier, on utilise l'ordre générique `<`.

Par exemple, sur l'arbre figuré à l'exercice I, la liste obtenue sera `[5;3;5]`.

EXERCICE IV : Exercice avancé sur les arbres: codage de Huffman – à finir à la maison

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 intervenant 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.

On calcule le code de chaque symbole en construisant une structure d'arbre binaire à partir d'une liste de fréquences des symboles. Les feuilles de cet arbre sont étiquetées avec les symboles. On peut représenter le *chemin* qui va de la racine de l'arbre à un symbole par une suite de 0 et de 1: 0 pour descendre dans le sous-arbre gauche; 1 pour descencdre dans le 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`, c'est la fréquence du symbole); aux nœud (constructeur `Branch`), c'est la somme des entiers que l'on trouve à la racine des deux sous-arbres. Par exemple, si l'on a la table de fréquences `[('A',8); ('B',3); ('C',1); ('D',1)]`, on aura l'arbre

```
Branch(9, Leaf(4,'A'),
          Branch(5, Leaf(3,'B'),
                    Branch(2, Leaf(1,'C'), Leaf(1,'D'))))
```

Il est de type `char htree`. Nous appellerons cet arbre `ht_ABCD` dans la suite de cet exercice.

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

'A'	0
'B'	10
'C'	110
'D'	111

Pour simplifier, on représente les codes par des listes d'entiers. Cette table sera représentée par la liste de couples (liste d'association)

```
[('A', [0]); ('B', [1;0]); ('C', [1;1;0]); ('D', [1;1;1])]
```

Elle est de type `(char * int list) list`.

Q1 – Définir la fonction

```
huff_tab (ht:'a htree) : ('a * int list) list
```

qui extrait de l'arbre `ht` la table des codes de Huffman (liste d'association) pour chaque symbole présent aux feuilles de l'arbre.

Q2 – Définir la fonction

```
code (msg:'a list) (htab:('a * int list) list) : int list
```

qui code le message `msg` à partir de la table de codage de Huffman `htab`.

Avec notre exemple, le code du message (liste de caratères) `['A';'B';'C';'A';'B';'D';'A';'B';'A']` sera `[0; 1;0; 1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0]`.

Q3 – Définir la fonction

```
decode1 (bs:int list) (ht:'a htree) : ('a * int list)
```

telle que si `bs` est le code d'un message et `ht` l'arbre qui a servi pour son codage, alors `(decode1 bs ht)` donne le couple formé du premier symbole codé de `bs` et du reste du message codé privé du code de ce premier symbole.

Par exemple:
`(decode1 [0; 1;0; 1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0] ht_ABCD)` donne le couple
`('A', [1;0; 1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0])`
`(decode1 [1;0; 1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0] ht_ABCD)` donne le couple
`('B', [1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0])`
`(decode1 [1;1;0; 0; 1;0; 1;1;1; 0; 1;0; 0] ht_ABCD)` donne le couple
`('C', [0; 1;0; 1;1;1; 0; 1;0; 0])`
etc.

Q4 – Utilisez `decode1` pour définir la fonction

```
decode (bs:int list) (ht:'a htree) : 'a list
```

qui décode le message `bs` en utilisant l'arbre de Huffman `ht`.

Construction de l'arbre – à finir à la maison L'arbre de Huffman est construit à partir d'une table de fréquences des symboles selon le principe de suivant:

- on commence par transformer la liste des fréquences en une liste d'arbres: chaque couple `(s,n)` de la liste fréquences est transformé en une liste de feuilles `Leaf(n,s)` (voir 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 (voir fonctions `ht_less` et `get_min`)
  2. fusionner ces deux arbres (voir fonction `ht_branch`)
  3. recommencer en remplaçant dans la liste les deux arbres minimaux par l'arbre obtenu par leur fusion.
- si la liste contient un seul arbre, on termine avec cet arbre

(voir fonction `build_huff`)

Q5 – Définir la fonction

```
ht_less (ht1:'a htree) (ht2:'a htree) : bool
```

qui donne `true` si et seulement si l'étiquette entière de la racine de `ht1` est inférieure à celle de `ht2`.

Exemples:

`(ht_less (Branch(5,Leaf(3,'A'),Leaf(2,'Z'))) (Leaf(6,'X')))` vaut `true`

`(ht_less (Leaf(6,'X')) (Branch(5,Leaf(3,'A'),Leaf(2,'Z'))) )` vaut `false`

Q6 – Définir la fonction

```
get_min (hts : ('a htree) list) : ('a htree * ('a htree) list)
```

qui donne le couple constitué d'un arbre minimal (au sens de `ht_less`) de `hts` et de la liste obtenue en retirant cet arbre de `hts`.

Q7 – Définir la fonction

```
ht_branch (ht1 : 'a htree) (ht2 : 'a htree) : 'a htree
```

qui donne l'arbre `Branch(n, ht1, ht2)` où `n` est la somme des étiquettes entières aux racine de `ht1` et `ht2`.

Q8 – Déduire de ce qui précède la définition de

```
build_huff (hts : ('a htree) list) : 'a htree
```

qui construit l'arbre de Huffman obtenue en suivant le principe de «réduction» donné en introduction de ce paragraphe.

Q9 – En utilisant `List.amp`, définir la fonction

```
leaf_list (freq:('a * int) list) : ('a htree) list
```

qui donne la liste des feuilles `Leaf(n,x)` obtenus à partir des couples `(x,n)` de la liste `freq`.

Q10 – Déduire de tout cela la définition de la fonction

```
huff_of_freq (freq:('a * int) list) : 'a htree
```

qui construit un arbre de Hufman à partir de la liste de fréquence `freq`.