Université de Bordeaux **Collège Sciences et Technologies**
2016/2017 **L2 Informatique, UE 4TIN402U**

Algorithmique des structures de données arborescentes
TP noté du jeudi 13 avril 2017
Durée : 1h20

1. Téléchargez le sequelette du tp situé à l'url `http://dept-info.labri.fr/~idurand/ASDA/tp.ml`
2. Renommez le fichier `tp.ml` en `<NOM>-<PRENOM>.ml` en remplaçant `<NOM>` et `<PRENOM>` par vos nom et prénom. Exemple : `TURING-ALAN`.
3. Dans le fichier ainsi renommé, indiquez à nouveau vos nom et prénom et votre numéro de groupe.
4. Pensez à sauver régulièrement votre travail.
5. À la fin du tp,
   (a) copiez votre fichier dans le dossier suivant `/net/ens/ASDA/IN401Ax` en remplaçant `x` par votre numéro de groupe (`1` ou `3`).
   (b) envoyez votre fichier par mail avec comme **sujet** `TP-ASDA` à votre chargée de TD
      — INF401A1 : `irene.durand@u-bordeaux.fr`
      — INF401A3 : `stefka.gueorguieva@u-bordeaux.fr`

1 Code binaire

On dispose d'un ensemble fini d'objets E. On souhaite représenter des sous-ensembles de E de manière à pouvoir effectuer la recherche d'un objet de manière efficace.
Dans ce but, les objets de E sont numérotés de 1 à N où N est le cardinal [1] de E.
L'objet numéro i est codé par la représentation binaire de i sur un nombre nb fixé de bits. La valeur de nb dépend de N. Elle est égale à $1 + \lfloor(log_2(N))\rfloor$.
Pour le tp, l'ensemble E sera l'ensemble des lettres minuscules de l'alphabet.

$$E = \{a, b, \ldots, z\}$$

codées par les caractères `OCaml` correspondants :

```
# alphabet;;
- : char list =
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o';
 'p'; 'q'; 'r'; 's'; 't'; 'u'; 'v'; 'w'; 'x'; 'y'; 'z']
```

Les lettres sont donc numérotées de 1 à 26 en suivant l'ordre alphabétique (qui est aussi l'ordre des caractères). Ainsi, a est numérotée 1, b est numérotée 2 et z est numérotée 26. De plus, d'après la formule vue plus haut, nous aurons besoin de 5 bits pour coder nos 26 lettres. Commençons par le numérotage des lettres.

1. Écrire la fonction `letter_to_int letter` qui, étant donnée une lettre minuscule retourne son numéro. On pourra utiliser la fonction `OCaml int_of_char`.
2. Écrire la fonction inverse `int_to_letter n` qui donne la lettre à partir de son numéro.

Exemples :

1. le nombre d'objets contenus dans E

```
# letter_to_int 'a';;
- : int = 1
# letter_to_int 'b';;
- : int = 2
# letter_to_int 'z';;
- : int = 26
# int_to_letter 3;;
- : char = 'c'
```

3. Écrire une fonction `iota n` qui retourne la liste des n entiers consécutifs allant de `1` à `n`.

Exemples :

```
# iota 0;;
- : int list = []
# iota 1;;
- : int list = [1]
# iota 5;;
- : int list = [1; 2; 3; 4; 5]
```

4. Compléter l'expression de l'exemple ci-dessous de manière à ce qu'après évaluation la variable `alphabet` contienne la liste ordonnée des lettres minuscules. On pourra se servir de la fonction `OCaml List.map`[2]. et de la fonction `iota` définie précédemment.

```
# let alphabet = (* à compléter *) ;;
val alphabet : char list =
  ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o';
   'p'; 'q'; 'r'; 's'; 't'; 'u'; 'v'; 'w'; 'x'; 'y'; 'z']
```

5. Écrire une fonction `complete_with_zeros bits k` qui prend en paramètre une liste de bits (0 ou 1) et un entier `k` et qui complète la liste de bits par des zéros à gauche de façon à ce que la liste soit de longueur au moins égale à `k`.

```
# complete_with_zeros [0; 1; 0 ; 1] 6;;
- : int list = [0; 0; 0; 1; 0; 1]
# complete_with_zeros [0; 1; 0 ; 1] 3;;
- : int list = [0; 1; 0; 1]
```

6. Écrire une fonction `base2_digits n` qui s'applique à un entier `n` **strictement positif** et qui retourne la liste des bits de sa valeur en binaire.

Exemples :

```
# base2_digits 1;;
- : int list = [1]
# base2_digits 2;;
- : int list = [1; 0]
# base2_digits 9;;
- : int list = [1; 0; 0; 1]
```

D'après la formule vue plus haut, nous avons besoin de $nb = 5$ bits pour coder 26 lettres.

7. Utiliser les deux lignes de code ci-dessous pour initialiser la variable `code_size`.

2. Voir en Annexe Page 6 des exemples d'utilisation de la fonction `List.map`

```
let nb_bits n = 1 + int_of_float (log (float_of_int n) /. log 2.)
let code_size = nb_bits 26
```

Nous allons donc coder chaque lettre par son code binaire écrit sur 5 bits. Ainsi la lettre a est codée par 00001, la lettre b par 00010 et la lettre z par 11010.
Pour la programmation en `OCaml` et de manière à pouvoir exploiter l'information, le code de chacun des objets (donc des lettres en ce qui nous concerne) sera représenté par liste de bits. Ainsi la lettre e de code 00101 sera représentée par la liste `[0; 0; 1; 0; 1]`.

8. Écrire une fonction `letter_to_bits letter nb` qui prend en paramètre une lettre `letter` et un entier `nb` et qui retourne la liste des bits de longueur `nb` codant la lettre.
9. En utilisant la variable `code_size` et la fonction `letter_to_bits letter nb`,
   écrire une fonction `encode_letter letter` qui retourne le code de la lettre `letter` sur `code_size` bits.

Exemples :

```
# letter_to_bits 'f' 8;;
- : int list = [0; 0; 0; 0; 0; 1; 1; 0]
# code_size;;
- : int = 5
# encode_letter 'f';;
- : int list = [0; 0; 1; 1; 0]
```

10. Écrire une fonction `letters_code letters` qui prend en paramètre une liste de lettres et retourne la liste des couples $(lettre, code)$ associée. On pourra utiliser la fonction `List.map`.

Exemple :

```
# letter_codes ['p'; 'r'; 'i'; 'n'; 't'; 'e'; 'm'; 'p'; 's'];;
- : (char * int list) list =
[('p', [1; 0; 0; 0; 0]); ('r', [1; 0; 0; 1; 0]); ('i', [0; 1; 0; 0; 1]);
 ('n', [0; 1; 1; 1; 0]); ('t', [1; 0; 1; 0; 0]); ('e', [0; 0; 1; 0; 1]);
 ('m', [0; 1; 1; 0; 1]); ('p', [1; 0; 0; 0; 0]); ('s', [1; 0; 0; 1; 1])]
```

11. Utiliser la fonction précédente pour définir une variable `alphabet_codes` à partir de la variable déjà définie `alphabet` et en utilisant la fonction `letters_code`.

Exemple :

```
# let alphabet_codes = (* à compléter *) ;;
val alphabet_codes : (char * int list) list =
  [('a', [0; 0; 0; 0; 1]); ('b', [0; 0; 0; 1; 0]); ('c', [0; 0; 0; 1; 1]);
   ('d', [0; 0; 1; 0; 0]); ('e', [0; 0; 1; 0; 1]); ('f', [0; 0; 1; 1; 0]);
   ('g', [0; 0; 1; 1; 1]); ('h', [0; 1; 0; 0; 0]); ('i', [0; 1; 0; 0; 1]);
   ('j', [0; 1; 0; 1; 0]); ('k', [0; 1; 0; 1; 1]); ('l', [0; 1; 1; 0; 0]);
   ('m', [0; 1; 1; 0; 1]); ('n', [0; 1; 1; 1; 0]); ('o', [0; 1; 1; 1; 1]);
   ('p', [1; 0; 0; 0; 0]); ('q', [1; 0; 0; 0; 1]); ('r', [1; 0; 0; 1; 0]);
   ('s', [1; 0; 0; 1; 1]); ('t', [1; 0; 1; 0; 0]); ('u', [1; 0; 1; 0; 1]);
   ('v', [1; 0; 1; 1; 0]); ('w', [1; 0; 1; 1; 1]); ('x', [1; 1; 0; 0; 0]);
   ('y', [1; 1; 0; 0; 1]); ('z', [1; 1; 0; 1; 0])]
```

2 Recherche digitale

2.1 Arbre de recherche digitale (définitions)

Un *arbre de recherche digitale* est un arbre binaire permettant de stocker des éléments en vue de leur recherche. Les éléments sont stockés aussi bien dans **les noeuds** que dans **les feuilles**. On utilisera le type `OCaml` suivant pour représenter ce type d'arbre.

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

Comme dans le code de Huffman, le bit 0 indique le sous-arbre gauche et le bit 1 indique le sous-arbre droit. Un élément e est placé dans l'arbre dès qu'il trouve une place et en suivant son code.
Soit l'élément e ayant pour code $[b_0, \ldots, b_{k-1}]$ à insérer dans un arbre t.

— si l'arbre est vide, l'élément e se place dans un unique noeud qui est à la fois la racine et l'unique feuille de l'arbre.

— si l'arbre n'est pas vide, l'élément e est inséré avec le code $[b_1, \ldots, b_{k-1}]$
  — dans le sous-arbre gauche si $b_0 = 0$,
  — dans le sous-arbre droit si $b_0 = 1$.

À noter que l'ordre d'insertion des éléments dans l'arbre conditionne leur position dans l'arbre : les éléments insérés en premier prennent les *meilleures* places, c'est-à-dire les plus proches de la racine.

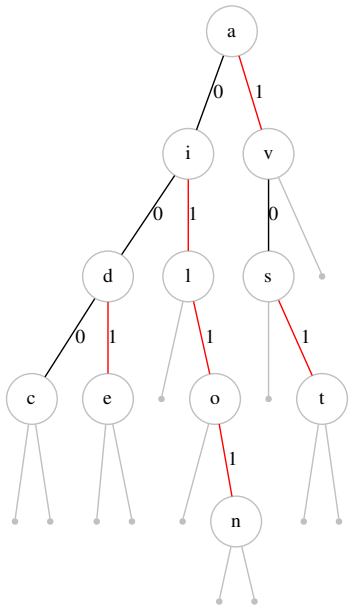

FIGURE 1 – Arbre de recherche digitale obtenu par insertion des lettres de la liste `letters`

Ainsi si nous construisons l'arbre à partir de la liste de lettres

```
let letters = ['a'; 'v'; 'i'; 's'; 'd'; 'e'; 'l'; 'o'; 'c'; 't'; 'n']
```

nous obtenons l'arbre donné par la figure 1 tandis que si nous insérons les lettres dans l'ordre inverse

```
['n'; 't'; 'c'; 'o'; 'l'; 'e'; 'd'; 's'; 'i'; 'v'; 'a']
```

nous obtenons celui de la figure 2.

```
# letters_to_dtree letters;;
- : char dtree =
Node ('a',
 Node ('i', Node ('d', Node ('c', Empty, Empty), Node ('e', Empty, Empty)),
```

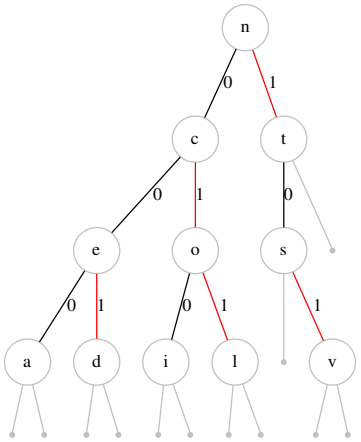

FIGURE 2 – Arbre de recherche digitale obtenu par insertion des lettres de la liste `letters` **inversée**

```
  Node ('l', Empty, Node ('o', Empty, Node ('n', Empty, Empty)))),
 Node ('v', Node ('s', Empty, Node ('t', Empty, Empty)), Empty))
# letters_to_dtree (List.rev letters);;
- : char dtree =
Node ('n',
 Node ('c', Node ('e', Node ('a', Empty, Empty), Node ('d', Empty, Empty)),
  Node ('o', Node ('i', Empty, Empty), Node ('l', Empty, Empty))),
 Node ('t', Node ('s', Empty, Node ('v', Empty, Empty)), Empty))
```

Nous écrirons la fonction d'insertion dans un arbre de recherche digitale plus tard.

2.2 Recherche dans un arbre de recherche digitale

La recherche d'un élément e dans un arbre (ou sous-arbre) de recherche digitale t s'effectue à partir de e, t et la partie du code non encore utilisée.
Si l'arbre est vide, e n'appartient pas à l'arbre, sinon si e est à la racine, e appartient à l'arbre sinon en supposant que le code est de la forme $(b_0, \ldots, b_{k-1})$ on relance la recherche avec le code $(b_1, \ldots, b_{k-1})$ dans le sous-arbre gauche si $b_0 = 0$ dans le sous-arbre droit si $b_0 = 1$.

12. Écrire une fonction `dtree_read letter dtree bits` qui prend en paramètre, `letter` la lettre recherchée, `dtree` l'arbre (ou sous-arbre) de recherche digitable et `bits` correspondants aux bits du code non encore utilisés et retourne `true` si la lettre est trouvée et `false` sinon.
13. En déduire une fonction `dtree_search letter dtree` qui effectue la recherche de la lettre `letter` dans l'arbre `dtree` c'est-à-dire qui retourne `true` si la lettre appartient à l'arbre et `false` sinon.

Exemples :

```
# dtree;;
val dtree : char dtree =
  Node ('a',
   Node ('i', Node ('d', Node ('c', Empty, Empty), Node ('e', Empty, Empty)),
    Node ('l', Empty, Node ('o', Empty, Node ('n', Empty, Empty)))),
   Node ('v', Node ('s', Empty, Node ('t', Empty, Empty)), Empty))
# encode_letter 'v';;
- : int list = [1; 0; 1; 1; 0]
```

```
# dtree_read 'v' dtree (encode_letter 'v') ;;
- : bool = true
# encode_letter 'm';;
- : int list = [0; 1; 1; 0; 1]
# dtree_read 'm' dtree (encode_letter 'm') ;;
- : bool = false
#  encode_letter 's';;
- : int list = [1; 0; 0; 1; 1]
# dtree_read 's' (Node ('v', Node ('s', Empty, Node ('t', Empty, Empty)), Empty)) [0; 1; 1];;
- : bool = true
# dtree_search 'v' dtree;;
- : bool = true
```

3 Insertion dans un arbre de recherche digitale

14. En utilisant l'algorithme présenté Page 4, écrire la fonction `dtree_insert letter dtree` qui retourne l'arbre résultant de l'insertion de la lettre `letter` dans l'arbre de recherche digitale `dtree`.

Exemples :

```
# let dtree = dtree_insert 'a' Empty;;
val dtree : char dtree = Node ('a', Empty, Empty)
# let dtree = dtree_insert 'v' dtree;;
val dtree : char dtree = Node ('a', Empty, Node ('v', Empty, Empty))
# let dtree = dtree_insert 'i' dtree;;
val dtree : char dtree =
  Node ('a', Node ('i', Empty, Empty), Node ('v', Empty, Empty))
# let dtree = dtree_insert 's' dtree;;
val dtree : char dtree =
  Node ('a', Node ('i', Empty, Empty),
   Node ('v', Node ('s', Empty, Empty), Empty))
# let dtree = dtree_insert 'd' dtree;;
val dtree : char dtree =
  Node ('a', Node ('i', Node ('d', Empty, Empty), Empty),
   Node ('v', Node ('s', Empty, Empty), Empty))
```

15. Écrire une fonction `letters_to_dtree letters` qui construit un arbre de recherche digitale par insertions successives des lettres de la liste `letters` dans un arbre initialement vide.

Exemple :

```
# letters_to_dtree letters;;
val dtree : char dtree =
  Node ('a',
   Node ('i', Node ('d', Node ('c', Empty, Empty), Node ('e', Empty, Empty)),
    Node ('l', Empty, Node ('o', Empty, Node ('n', Empty, Empty)))),
   Node ('v', Node ('s', Empty, Node ('t', Empty, Empty)), Empty))
```

Annexe

Exemples d'utilisation de la fonction `List.map` :

```
# List.map pred [2; 4; 6; 8];;
- : int list = [1; 3; 5; 7]
# List.map (fun x -> x * x)  [1; 2; 3; 4];;
- : int list = [1; 4; 9; 16]
}
```