université de BORDEAUX	**Année 2017/2018** — L2 Info/MI, UE 4TIN402U Algorithmique des structures de données arborescentes Vendredi 4 mai 2018, 9h–10h30 — Durée : 1h30 Documents : **non autorisés** Le barème est donné à titre **indicatif**	**Collège Sciences et Technologies**

Exercice 1 *Arbres rouges et noirs (4 points)*
Quel est l'arbre rouge et noir produit par l'algorithme d'insertion vu en cours lorsqu'on insère les entiers 7, 6, 5, 3, 4, 2, 1 dans cet ordre, à partir de l'arbre constitué d'une unique feuille ? Détailler chaque étape d'insertion.
Rappel. Les feuilles sont noires et n'ont pas de valeur. Vous pouvez donc, au choix, les représenter ou non.
Note. Lisez attentivement la suite d'entiers : elle n'est pas décroissante à cause de la position de 3 et 4.

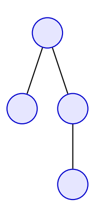
Exercice 2 *Algorithme de Huffman (3 points)*
On considère le texte **OVERNERVOUSNESSES** (de 17 caractères). Appliquer sur ce texte l'algorithme de Huffman vu en cours pour :
1. créer l'arbre de Huffman permettant d'associer un code à chaque caractère,
2. produire la liste de couples (caractère, code),
3. produire la suite de bits correspondant aux 4 premières lettres du texte : **OVER**.

Exercice 3 *Preuve de propriété (3 points)*
Soit t un arbre binaire **plein** à n nœuds. On suppose que $n \geqslant 2$. Montrer par récurrence sur la hauteur de t que t possède deux feuilles qui ont la même profondeur.

Exercice 4 *Arbres planaires et mots de Dyck (4 points)*
On représente un mot de Dyck par une liste de directions :

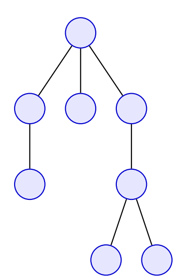
```
type direction = Up | Down
type dyckword = direction list
```

On rappelle que la bijection entre squelettes d'arbres planaires à $n+1$ nœuds et mots de Dyck de taille $2n$ est obtenue en parcourant l'arbre en profondeur, de la racine à la racine, et en produisant `Up` lorsque l'on descend le long d'une arête et `Down` lorsqu'on remonte le long d'une arête. Une illustration est donnée sur la Figure 1 (a).



Arbre planaire correspondant au
mot de Dyck `[Up;Down;Up;Up;Down;Down]`

(a)



(b)

FIGURE 1 – Arbres planaires et mots de Dyck

1. Écrire le mot de Dyck correspondant à l'arbre planaire de la Figure 1 (b).

On considère le type suivant pour représenter les arbres planaires non vides :

```
type planar_tree = Node of planar_tree list
```

Par exemple, l'arbre constitué de 4 nœuds, une racine ayant 3 fils, est `Node[Node[];Node[];Node[]]`.

2. Écrire une fonction `planar_to_dyck : planar_tree -> dyckword` produisant le mot de Dyck correspondant à un arbre planaire passé en argument.
   Note. Vous pouvez utiliser toutes les fonctions de la bibliothèque OCaml, par exemple `List.flatten` qui aplatit une liste de listes : `List.flatten [[1]; [2; 3]]` vaut `[1; 2; 3]`.

Exercice 5 *Largeur d'un arbre (6 points + 2 points bonus)*

Dans cet exercice, on utilise le type suivant pour représenter des arbres binaires :

```
type 'a tree = Empty | Bin of ('a * 'a tree * 'a tree)
```

Par ailleurs, on suppose fourni un type `'a queue` permettant de représenter des **files** dont les éléments sont de type `'a`. On suppose les valeurs et fonctions suivantes déjà écrites :

```
empty_queue : 'a queue
enqueue : 'a -> 'a queue -> 'a queue
dequeue :'a queue -> ('a * 'a queue)
queue_size : 'a queue -> int
```

La valeur `empty_queue` est la file vide. L'appel `enqueue x q` renvoie la file obtenue lorsqu'on enfile l'élément `x` dans la file `q`. L'appel `dequeue q` renvoie un couple $(\mathtt{x}, \mathtt{q}')$ où `x` est le premier élément de la file `q`, et $\mathtt{q}'$ est la file obtenue en défilant `x` de `q` (cette fonction produit une erreur si `q` est la file vide). Enfin, l'appel `queue_size q` renvoie le nombre d'éléments de la file `q`.

1. Que valent les files `q1`, `q2`, `q3` et `q4` ainsi que les entiers `x3` et `x4` après exécution du code suivant ?

```
let q1      = enqueue 1 empty_queue
let q2      = enqueue 2 q1
let (x3,q3) = dequeue q2
let (x4,q4) = dequeue q3
```

2. On considère la fonction `next_level` suivante, qui prend en paramètre une file `q` d'arbres (chacun de type `'a tree`), une liste `acc_nodes` de valeurs (chacune de type `'a`) et une file `acc_trees` d'arbres (chacun de type `'a tree`).

```
let rec next_level q acc_nodes acc_trees =
  if queue_size q = 0  then (acc_nodes, acc_trees)
  else let (t, q') = dequeue q in
    match t with
      Empty -> next_level q' acc_nodes acc_trees
    | Bin(x, left, right) ->
             next_level q' (x::acc_nodes) (enqueue right (enqueue left acc_trees))
```

Simuler l'exécution du code suivant, en décrivant, pour les 2 appels à `next_level`, chaque appel récursif avec ses arguments.

```
let t = Bin(1, Bin(2, leaf 3, Bin(4, Bin(5, leaf 6, leaf 7), Empty)),
               Bin(8, Empty, leaf 9))
let (acc_nodes, acc_trees) = [], (enqueue t empty_queue)
let (acc_nodes, acc_trees) = next_level acc_trees acc_nodes empty_queue
let (acc_nodes, acc_trees) = next_level acc_trees acc_nodes empty_queue
```

3. Utiliser la fonction `next_level` pour écrire une fonction prenant en argument un arbre `t` et produisant la liste des étiquettes de ses nœuds, listées dans un parcours en **largeur**.
4. *(Bonus, 2 points)*. On définit la *largeur* d'un arbre non vide comme le nombre maximal de nœuds se trouvant à une même profondeur. Par exemple, la largeur de l'arbre de la question 2 vaut 3, car il y a 3 nœuds à profondeur 2, et il n'y a jamais plus de 3 nœuds aux autres profondeurs.
   Modifier l'algorithme de la question 3 pour qu'il calcule la largeur de l'arbre.