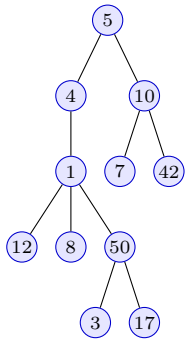
Algorithmique des structures de données arborescentes
Feuille d'exercices 6 (semaine 9)

1 Parcours d'arbres

Exercice 6.1 On considère l'algorithme suivant en pseudo-langage, prenant en paramètre un arbre planaire t non vide :

PARCOURS(t) :
p ← Créer_Pile_Vide()
x_0 ← racine(t)
Empiler(p, x_0)
tant que p n'est pas vide **faire**
  x ← Dépiler(p)
  Traiter(x)
  pour tout fils y de x pris de droite à gauche **faire**
    Empiler (p, y)
  fin pour
fin tant que



1. Simuler l'exécution de l'algorithme lorsque t est l'arbre planaire ci-dessus, en détaillant l'évolution de la pile et l'ordre de traitement des nœuds.
2. Quel est le type de parcours réalisé par cet algorithme ?
3. On change l'algorithme de la façon suivante :
   - On utilise une file au lieu d'une pile,
   - On insère les fils d'un nœud dans la file en les considérant de gauche à droite.

   Simuler l'exécution de l'algorithme obtenu sur l'arbre planaire ci-dessus, en détaillant l'évolution de la file et l'ordre de traitement des nœuds.
4. Quel est le type de parcours réalisé par cet algorithme ?

Exercice 6.2 On considère le type `type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree` permettant de représenter des arbres binaires. Écrire les deux fonctions suivantes. On demande que leur complexité soit $O(n)$, où n est la taille de l'arbre pris en argument.

1. Écrire une fonction `dfs : 'a tree -> 'a list` qui produit la liste des nœuds de l'arbre passé en argument, dans l'ordre donné par le parcours préfixe en profondeur.

   Indication. La difficulté est d'assurer une complexité linéaire. Il faut éviter d'utiliser l'opérateur de concaténation de listes `@`. Vous pouvez écrire une fonction auxiliaire qui prend en 1er argument une liste d'arbres, représentant une pile, et en 2e argument une liste accumulant les valeurs déjà visitées. Cette fonction ajoute la racine de l'arbre du haut de pile à l'accumulateur.
2. Écrire une fonction `bfs : 'a tree -> 'a list` qui produit la liste des nœuds de l'arbre passé en argument, dans l'ordre donné par le parcours en largeur. **Indication.** On peut utiliser le même principe que dans la question précédente en remplaçant *pile* par *file*. Pour implémenter une file de façon efficace, on peut utiliser deux piles (représentées par deux listes).

2 Arbres binaires, planaires et représentation par mots

Exercice 6.3 Écrire

- tous les arbres binaires à 3 nœuds,
- tous les arbres binaires complets à 7 nœuds,
- tous les arbres planaires à 4 nœuds,

- tous les mots de Dyck de longueur 6.

Exercice 6.4 Justifier qu'il y a autant d'arbres binaires de taille n que d'arbres binaires complets de taille $2n+1$ en donnant une bijection entre ces deux ensembles. Expliciter cette bijection pour $n = 3$.

Exercice 6.5 On associe à tout (squelette d')arbre planaire un (squelette d')arbre binaire de la façon suivante :

- Les nœuds de l'arbre binaire sont les mêmes que ceux de l'arbre planaire.
- La racine de l'arbre binaire est la racine de l'arbre planaire.
- Dans l'arbre binaire, un nœud a comme fils gauche son 1^{er} fils (s'il existe) dans l'arbre planaire, et comme fils droit son frère situé immédiatement à sa droite dans l'arbre planaire (s'il existe).

1. Construire l'arbre binaire correspondant à l'arbre planaire de l'exercice 1.
2. Justifier que cette fonction ne définit **pas** une bijection de l'ensemble des arbres planaires à n sommets dans l'ensemble des arbres binaires à n sommets.
3. Montrer que si on supprime la racine du squelette d'arbre binaire produit de cette façon, on obtient une bijection entre arbres planaires à n nœuds et arbres binaires à $n-1$ nœuds. Le vérifier pour $n = 4$.

On considère les types suivants :

```
type 'a planar_tree = Node of 'a * 'a planar_tree list
type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree
```

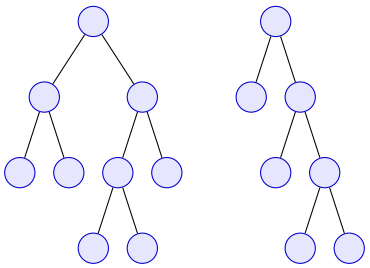
Le premier type représente les arbres planaires non vides. On représente un mot de Dyck par une liste de directions :

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

La bijection entre squelettes d'arbres binaires complets de taille $2n+1$ et mots de Dyck de taille $2n$ est obtenue ainsi :

- on numérote les nœuds de l'arbre dans l'ordre préfixe en profondeur, de 1 à $2n+1$.
- Pour $1 \leqslant k \leqslant 2n$, la k^{e} lettre du mot de Dyck produit est
  - `Up` si le nœud de numéro k est interne,
  - `Down` si le nœud de numéro k est une feuille.

Exercice 6.6 1. Expliciter la bijection pour $n = 3$. Quels sont les mots obtenus à partir des arbres complets suivants ?



2. Quel est l'arbre correspondant au mot de Dyck `[Up;Down;Up;Up;Down;Up;Down;Down]` ?
3. Écrire une fonction `full_to_dyck : 'a tree -> dyckword` qui retourne le mot de Dyck correspondant à l'arbre passé en argument, supposé complet.
4. Écrire une fonction `dyck_to_full : dyckword -> int tree` qui retourne l'arbre complet correspondant au mot de Dyck passé en argument (les étiquettes des nœuds n'ont pas d'importance).

Exercice 6.7 1. Écrire le mot de Dyck correspondant à l'arbre planaire de l'exercice 1 dans la bijection vue en cours.

2. Écrire une fonction `planar_to_dyck : planar_tree -> dyckword` produisant le mot de Dyck correspondant à un arbre planaire dans la bijection vue en cours.

Exercice 6.8

1. Écrire une fonction `planar_size : 'a planar_tree -> int` calculant la taille d'un arbre planaire. Vous pouvez utiliser les fonctions `List.map` et `List.fold_left`.
2. Écrire une fonction `planar_height : 'a planar_tree -> int` calculant la hauteur d'un arbre planaire.