Algorithmique des structures de données arborescentes
Feuille d'exercices 4

1 Arbres binaires de recherche

Dans les exercices suivants, les nœuds des arbres binaires sont étiquetés par des couples (c, m), où c est appelée la *clé* portée par le nœud, et où m est un entier strictement positif appelé la *multiplicité* de la clé. Chaque clé apparaît au plus dans un nœud de l'arbre. Un arbre représente ainsi un multi-ensemble de clés. Par exemple, un nœud étiqueté par $(10, 3)$ représente l'élément 10 en 3 exemplaires dans le multi-ensemble.

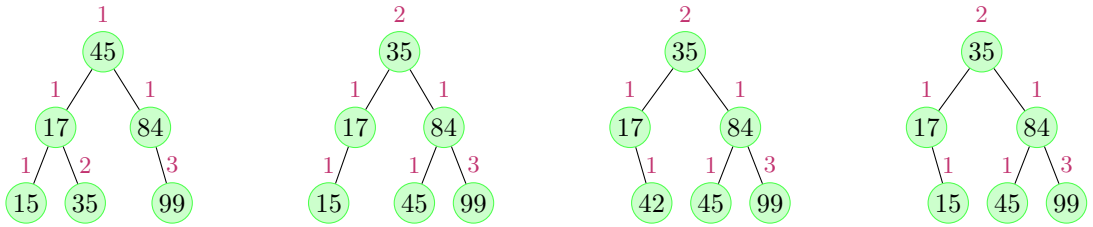
Dans les figures, les clés seront des entiers. Les clés sont représentées à l'intérieur des nœuds, et la multiplicité correspondante au dessus. On utilisera le type déjà vu précédemment dans les fonctions à écrire en OCaml :

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

Un arbre binaire t est appelé un *arbre binaire de recherche* (ABR, ou BST en anglais, pour binary search tree) si, pour ***tout*** nœud x de l'arbre t, on a les propriétés suivantes :

1. Tous les nœuds du sous-arbre gauche de x ont une clé strictement inférieure à celle de x.
2. Tous les nœuds du sous-arbre droit de x ont une clé strictement supérieure à celle de x.

Exercice 3.1 — Définition des arbres binaires de recherche. Parmi les arbres suivants, lesquels sont des ABR ? Justifier.



Exercice 3.2 — Définition des arbres binaires de recherche.

1. Proposer un algorithme pour tester qu'un arbre binaire est un ABR.
2. Comment modifier l'algorithme pour obtenir une complexité linéaire en fonction de la taille de l'arbre ?
3. Écrire une fonction `is_bst` de type `('a * int) tree -> bool` qui teste si un arbre est un ABR.

Exercice 3.3 — Recherche d'un élément dans un ABR.

1. Proposer un algorithme pour rechercher la multiplicité d'une clé dans un ABR (0 si la clé est absente).
2. Quelle est la complexité de votre algorithme en fonction de la hauteur de l'ABR ? de sa taille ?
3. Écrire une fonction `bst_search` de type `('a * int) tree -> 'a -> int` qui renvoie la multiplicité d'une clé dans un ABR (le premier argument est supposé être un ABR, il est inutile de le vérifier).

Exercice 3.4 — Insertion d'un élément dans un ABR. On veut maintenant insérer une clé dans un ABR, c'est-à-dire ajouter un exemplaire de la clé.

1. Que faut-il faire lorsque la clé est déjà présente dans l'ABR ?
2. Proposer un algorithme d'insertion de clé dans un ABR. Quelle est sa complexité dans le cas le pire ?
3. Écrire une fonction `bst_insert` de type `('a * int) tree -> 'a -> ('a * int) tree` qui renvoie l'arbre obtenu en insérant la clé donnée en $2^{\text{ème}}$ argument dans l'ABR donné en 1^{er} argument.

Exercice 3.5 — Liste vers ABR.

1. Écrire une fonction `bst_from_list` de type `'a list -> ('a * int) tree` qui produit l'arbre obtenu en partant de l'arbre vide et en y insérant successivement tous les éléments de la liste, pris de gauche à droite.
2. Calculer la complexité de cette fonction, dans le cas le pire.
3. L'ordre des éléments dans la liste a-t-il une importance ? Dessiner l'ABR obtenu à partir des listes suivantes :
   - `[1; 2; 3; 4; 5; 6; 7]`,
   - `[4; 2; 1; 3; 6; 5; 7]`.

Exercice 3.6 — ABR vers liste.

1. Écrire une fonction `list_from_bst` de type `('a * int) tree -> 'a list` qui produit la liste obtenue en parcourant l'ABR en parcours ***infixe*** (les clés seront répétées dans la liste autant de fois que leur multiplicité).
2. Quelle est la propriété de la liste obtenue ? Proposer une preuve de cette propriété.

Exercice 3.7 — Max, Prédécesseur.

1. Proposer un algorithme pour calculer l'élément maximal dans un ABR.
2. Écrire une fonction `bst_max` de type `('a * int) tree -> ('a * int) option` qui renvoie `None` si l'ABR est vide, et sinon, `Some (c,m)` où `c` est la clé maximale de l'arbre et `m` sa multiplicité.
3. Proposer un algorithme pour calculer le prédécesseur d'un élément dans un ABR.
4. Écrire une fonction `bst_pred` de type `('a * int) tree -> 'a -> ('a * int) option` telle que `bst_pred t x` renvoie `None` si `x` est l'élément minimal de l'ABR `t`, sinon, `Some (y,m)` où `y` est la clé qui précède `x` dans `t` et où `m` est la multiplicité de `y`.

Exercice 3.8

1. Proposer un algorithme pour supprimer une clé dans un ABR, en distinguant plusieurs cas selon la multiplicité de la clé. Lorsqu'il faut supprimer un nœud, distinguer également plusieurs cas selon que le nœud a zéro, un ou deux fils vides.
2. Écrire une fonction `bst_remove` de type `('a * int) tree -> 'a -> ('a * int) tree` qui renvoie l'ABR original si la clé donnée en second argument n'y est pas présente, et l'ABR obtenu en supprimant une occurrence de cette clé sinon.