Algorithmique des structures de données arborescentes
Feuille 3 - Arbres binaires et Arbres binaires de recherche.

1 Création/Modification dans les Arbres binaires

1.1 Modification d'étiquettes, modification de la forme de l'arbre

En programmation fonctionnelle pure, les variables sont "immutables". Cela signifie qu'une fois qu'une valeur leur est attribuée, elle ne peut plus être modifiée. Donc toute fonction qui prend en argument un arbre t ne peut pas modifier la variable t (pas d'effet de bord possible). Si l'on doit modifier des étiquettes ou la forme de l'arbre, il faut construire un nouvel arbre à l'aide des constructeurs habituels.

Exercice 1 : Nouvelles valeurs aux feuilles

Écrire une fonction `btree_mult2_leaves` de type `int btree-> int btree`, qui prend en argument un arbre binaire d'entiers t et qui multiplie par deux les étiquettes des feuilles.

Exercice 2 : Extension de l'arbre

Écrire une fonction `btree_expand_leaves` de type `int btree-> int btree`, qui prend en argument un arbre binaire t et qui ajoute à toutes les feuille un fils gauche et un fils droit, qui deviennent les nouvelles feuilles. Si l'étiquette de la feuille de t est x, le fils gauche dans le nouvel arbre sera étiqueté par $2x$ et le fils droit sera étiqueté par $2x + 1$.

On rappelle qu'en programmation fonctionnelle, les fonctions sont des objets de première classe, c'est-à-dire des variables comme les autres, qui peuvent être passées en argument d'une autre fonction.

Exercice 3 : Appliquer une fonction aux étiquettes

Écrire une fonction `btree_map` de type `('a -> 'b) -> 'a btree -> 'b btree` qui prend en argument une fonction `f` et un arbre binaire et qui renvoie l'arbre binaire obtenu en appliquant la fonction `f` à toutes les étiquettes de l'arbre.
Exemple :

```
# let t1 =  (Node(2, Node(4, Node(6, Node(8, Empty, Empty),
                                        Node(10, Node(12, Empty, Empty), Empty)),
                                Node(14, Empty, Empty)),
                        Node(16, Empty, Node(18, Empty, Empty))));;

# btree_map (fun x -> 2.5 *. float_of_int x) t1;;

- : float btree =Node (5., Node (10., Node (15., Node (20., Empty, Empty),
                                        Node (25., Node (30., Empty, Empty), Empty)),
                                                Node (35., Empty, Empty)),
                                Node (40., Empty, Node (45., Empty, Empty)))
```

1.2 Création d'Arbres binaires parfaits

Exercice 4 : Entiers vers arbres binaires parfaits

On considère les arbres binaires qui portent des valeurs de type `int`.
Écrire une fonction `btree_perfect` de type `int -> int btree` qui, étant donné un entier h, $h \geqslant 0$, renvoie un arbre parfait de hauteur h contenant tous les entiers de 1 à $2^{h+1} - 1$. La racine sera étiqueté par 1 et on étiquettera niveau par niveau avec les entiers croissants.
Indication : on pourra utiliser une fonction récursive intermédiaire prenant un paramètre supplémentaire : l'entier qui étiquettera la racine de l'arbre.
Exemple :

```
# btree_perfect 2;;
- : int btree =
Node (1, Node (2, Node (4, Empty, Empty), Node (5, Empty, Empty)),
Node (3, Node (6, Empty, Empty), Node (7, Empty, Empty)))
```

2 Arbres binaires de recherche

Une *structure de données* est une façon d'organiser les données (ex : tableaux, listes, ...). L'objectif d'une telle structure est de permettre un accès et une modification rapide des données. Les opérations habituelles sur une structure de données sont

1. Chercher une donnée,
2. Ajouter une donnée,
3. Supprimer une donnée,
4. Trouver la plus petite (ou la plus grande) donnée,

Ces opérations sont plus ou moins complexes selon la structure de données choisie. Les arbres sont des structures qui permettent de ranger efficacement des données. Nous allons nous intéresser aux *arbres binaires de recherche*. Nous considèrerons des arbres où deux nœuds distincts ont toujours des étiquettes différentes.

Définition : Arbres binaires de recherche (ABR)

Un *arbre binaire de recherche*, est un arbre binaire t dont les étiquettes, appelées *clés*, sont prises dans un ensemble ordonné et tel que 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.

L'objectif est de pouvoir réaliser les opérations habituelles citées ci-dessus pour un arbre de taille n en un temps $O(log(n))$ (on rappelle qu'un algorithme est en $O(f(n))$ ssi il existe une constante n_0 telle que pour toutes les entrées de taille supérieure à n_0 on est assuré que l'algorithme ne prend pas plus de $c \cdot f(n)$ étapes, où c est une constante strictement positive).

Attention

Ce n'est pas le cas pour les arbres binaires de recherche en général. Il faudra imposer des conditions supplémentaires sur les arbres binaires de recherche pour obtenir cette complexité.

Pour l'implémentation en OCaml, on utilisera le type déjà vu précédemment pour représenter les ABRs :

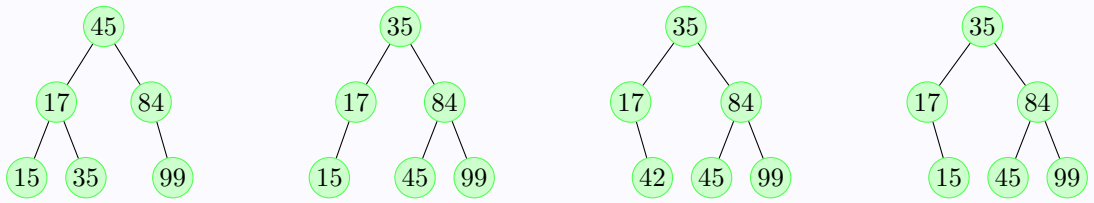
```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Attention

Le type `'a btree` ci-dessus permet de représenter des arbres binaires *quelconques*. Tout objet de type `'a btree` n'est donc pas *nécessairement* un ABR (la condition sur les clés imposée par la définition n'est pas forcément satisfaite).

Exercice 5 : Définition des arbres binaires de recherche

Parmi les arbres suivants, lesquels sont des ABR ? Justifier.



Nous allons commencer par l'opération de recherche d'un élément dans un arbre binaire de recherche.

Exercice 6 : Recherche dans un ABR

1. Proposer un algorithme qui prend en entrée un ABR t et une clé c et teste si c est présente dans t. Dans un arbre binaire de recherche parfait de hauteur h combien de comparaisons sont nécessaire pour trouver la clé qui étiquette la feuille la plus à droite ? Combien de comparaisons seraient nécessaires pour trouver cette clé dans la liste en ordre infixe des étiquettes cet arbre ?
2. Écrire une fonction `bst_search` de type `'a btree -> 'a -> bool` qui implémente l'algorithme de recherche (le premier argument est supposé être un ABR, il est inutile de le vérifier).

Nous allons maintenant voir un algorithme naturel d'insertion dans les arbres bianires de recherche sans aucune restriction sur la forme de l'arbre construit. Par conséquent l'arbre construit ne permettra pas toujours d'obtenir un algorithme de recherche efficace.

Exercice 7 : Insertion dans un ABR

1. Proposer un algorithme d'insertion de clé dans un ABR. Quelle est la complexité au pire de cet algorithme appliqué à un arbre de taille n ?
2. Écrire une fonction `bst_insert` de type `'a btree -> 'a -> 'a btree` qui renvoie l'arbre obtenu en insérant la clé donnée en $2^{\text{ème}}$ argument dans l'ABR donné en 1^{er} argument.

Exercice 8 : Liste vers ABR

1. Écrire une fonction `bst_from_list` de type `'a list -> 'a tree` qui produit un ABR en y insérant successivement tous les éléments d'une liste prise en entrée.
2. Pour une liste de n éléments, quelle est la hauteur de l'ABR construit dans le cas le pire ? Quelle est la complexité de l'algorithme de recherche d'une clé dans l'ABR 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]`.