Arbres binaires de recherche

I. Dictionnaires et ensembles

On s'intéresse à une structure de données permettant de stocker des informations à l'aide d'une *clé*, de les rechercher et les modifier aussi efficacement que possible.

On veut avoir les fonctions suivantes pour un type `('a, 'b) dict` où `'a` est le type des clés, et `'b` est le type des données associées aux clés :

```
val empty : unit -> ('a, 'b) dict
val find : ('a, 'b) dict -> 'a -> 'b
val add : ('a, 'b) dict -> 'a -> 'b -> ('a, 'b) dict
val remove : ('a, 'b) dict -> 'a -> 'b -> ('a, 'b) dict
```

On remarque que l'on a ici une structure de données *persistente*, puisque toutes les fonctions de modification (`add` et `remove`) retournent un dictionnaire, ce qui suggère que le dictionnaire passé en argument n'est pas changé, et que le dictionnaire résultat seul comporte les modifications.

Si l'on omet les données associées à une clé, et que l'on ne garde que ces dernières, on obtient une structure de donnée qui permet de manipuler des ensembles.

Si l'on a une structure de dictionnaire contenant n entrées, on a, suivant les implémentations, les complexités suivantes :

	Pire des cas		**En moyenne**	
	Recherche	Ajout	Recherche	Ajout
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Liste ordonnée	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Arbres de recherche équilibrés	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Tables de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Comme nous nous intéresserons plus particulièrement à la gestion des clés (et non des valeurs associées), nous considérons en fait une implémentation d'un type `'a set` ayant les les fonctions suivantes :

```
val empty : unit -> 'a set
val is_present : 'a set -> 'a -> bool
val add : 'a set -> 'a -> 'a set
val remove : 'a set -> 'a -> 'a set
```

Dans la suite, nous allons étudier les arbres binaires de recherche, et nous verrons

en TD comment les rendre efficaces en s'assurant qu'ils sont équilibrés (i.e. de hauteur en $\Theta(\log n)$ pour un coût raisonnable).

II. Rappels sur les arbres binaires

Nous allons représenter des arbres binaires dont les nœuds internes contiennent des informations de type `'a` à l'aide du type suivant :

```
type 'a arbre =
| F (* Feuille *)
| N of 'a arbre * 'a * 'a arbre  (* Noeud interne *)
```

Définition

☞ La **taille** d'un arbre est le nombre de nœuds internes qu'il contient. Elle est définie par

$$t(\mathtt{F}) = 0 \qquad t(\mathtt{N}(g, v, d)) = 1 + t(g) + t(d)$$

☞ La **hauteur** d'un arbre est la taille du plus long chemin simple partant de son sommet :

$$h(\mathtt{F}) = 0 \qquad h(\mathtt{N}(g, v, d)) = 1 + \max\big(h(g), h(d)\big)$$

Proposition

Étant donné un arbre binaire de hauteur h ayant n nœuds, on a :

$$h + 1 \leq n \leq 2^{h+1} - 1 \qquad \text{et} \qquad \lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Preuve

Pour le premier encadrement, le nombre de nœuds à une profondeur $k \in [\![0, h]\!]$ est compris entre 1 et 2^k (pourquoi?), d'où

$$h + 1 = \sum_{k=0}^{h} 1 \leq n \leq \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

Exercice 1 Prouver le second encadrement.

Exercice 2 Écrire les fonctions qui calculent la taille et la hauteur d'un arbre.

Exercice 3

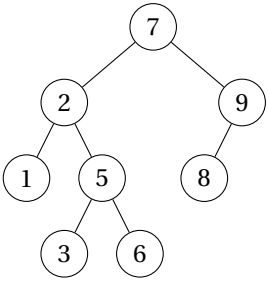
1. Écrire une fonction `to_list` retourne la liste des éléments d'un arbre binaire parcouru de gauche à droite, dans l'ordre *infixe*.
2. (plus difficile) Faire la même chose sans utiliser `@`.

III. Arbres binaires de recherche

Définition (Arbre binaire de recherche) Un **arbre binaire de recherche** (ou *a.b.r.*) est un arbre binaire étiqueté tel que pour chaque nœud `N`(g, v, d) de l'arbre,

1. Toute clé présente dans g (resp. dans d) est strictement inférieure (resp. supérieure) à v ;
2. Les sous-arbres g et d sont des a.b.r.

Exemple d'arbre binaire de recherche



Exercice 4 (Caractérisation et validation des a.b.r.)

1. Prouver qu'un arbre binaire étiqueté est un arbre binaire de recherche si et seulement si le parcours **infixe** des clés de l'arbre donne une suite strictement croissante.
2. Écrire une fonction qui teste si l'arbre binaire étiqueté en argument est bien un a.b.r.

Exercice 5 Écrire une fonction `maximum` qui renvoie le plus grand élément d'un arbre binaire de recherche *non vide*.

1. Recherche

Cet algorithme découle directement de la structure d'arbre binaire de recherche.

```
let rec cherche arbre valeur =
  match arbre with
  | F -> false
  | N (g, v, d) ->
      if v = valeur then
        true
      else if valeur < v then
        cherche g valeur
      else
        cherche d valeur
```

Analyse

☞ Concernant la terminaison et la complexité, les appels récursifs se font avec l'un des sous-arbres comme arguments, on a donc une complexité dans le pire des cas en $\Theta(h)$ où h est la hauteur de l'arbre.

☞ La correction de l'algorithme découle de la structure d'arbre binaire de recherche : les valeurs étant ordonnées selon le parcours infixe, la comparaison avec la valeur d'un arbre indique s'il faut continuer à chercher dans le sous-arbre gauche ou le droit.

2. Insertion

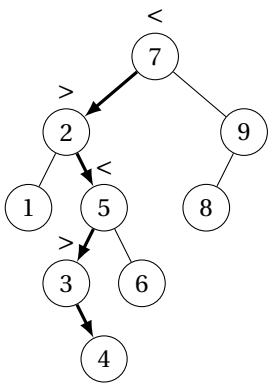
Nous allons voir deux méthodes pour ajouter une valeur dans un a.b.r. : soit aux feuilles, soit à la racine.

2.1. Insertion aux feuilles

Il suffit de se déplacer convenablement suivant les valeurs des nœuds rencontrés.

```
let rec insere_feuille arbre valeur =
  match arbre with
  | F -> N (F, valeur, F)
  | N (g, v, d) ->
      if v = valeur then arbre
      else if valeur < v then N (insere_feuille g valeur, v, d)
      else N (g, v, insere_feuille d valeur)
```

Exemple : Ajout aux feuilles de 4 dans l'a.b.r. précédent



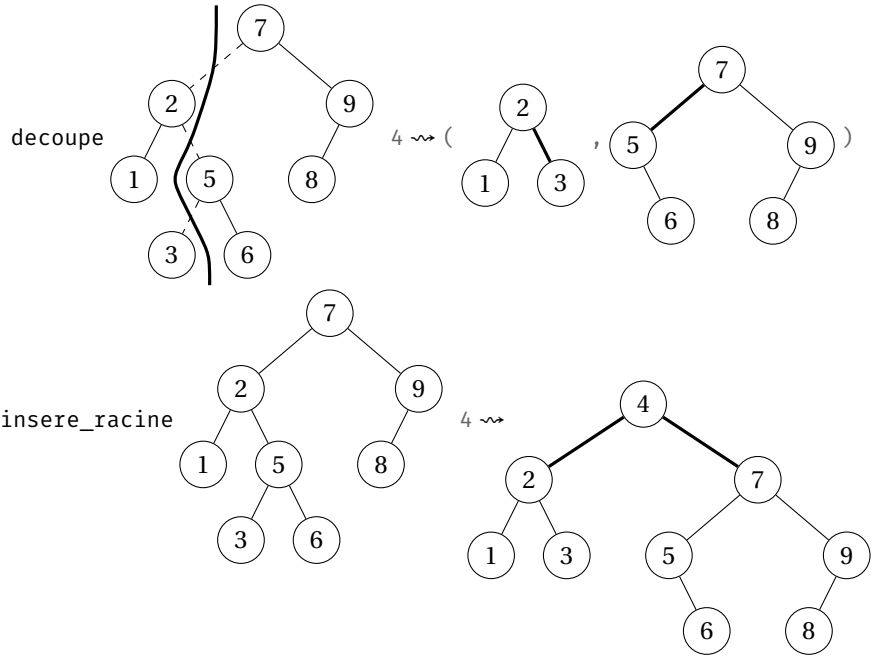
2.2. Insertion à la racine

On procède cette fois en deux temps. Tout d'abord, on scinde l'arbre en deux

nouveaux arbres binaires de recherche, contenant les valeurs strictement plus petites (resp. plus grandes) que la valeur à insérer. Il est alors facile de construire un nouvel a.b.r. avec la valeur à insérer à la racine.

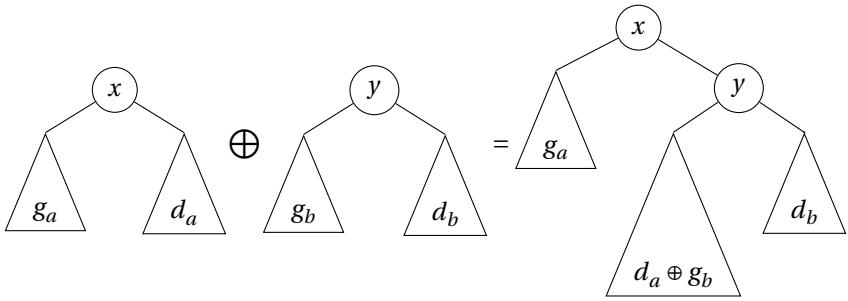
```
let rec decoupe arbre valeur =
  match arbre with
  | F -> (F, F)
  | N (g, v, d) ->
      if v = valeur then (g, d)
      else if valeur < v then
        let gg, gd = decoupe g valeur in
        (gg, N (gd, v, d))
      else
        let dg, dd = decoupe d valeur in
        (N (g, v, dg), dd)

let insere_racine arbre valeur =
  let g, d = decoupe arbre valeur in
  N (g, valeur, d)
```

Exemple : Ajout à la racine de 4 dans l'a.b.r. précédent



3. Suppression

On propose une méthode assez simple basée sur la concaténation de deux arbres binaires a et b dont le comportement est illustré par la figure suivante :



Sa traduction en OCaml est directe :

```
let rec concat a b =
  match (a, b) with
  | V, _ -> b
  | _, V -> a
  | N (ga, x, da), N (gb, y, db) ->
      N (ga, x, N (concat da gb, y, db))
```

Proposition - Propriétés de la concaténation

☞ Si a et b sont des a.b.r., et si toute clé présente dans a est strictement inférieure à toute clé présente dans b, alors `concat` a b est un a.b.r.

☞ L'ensemble des clés de arbre ainsi obtenu est la réunion des ensembles de clés des deux arbres de départ.

La fonction de suppression se déduit directement de la fusion, car si l'on rencontre le nœud à supprimer dans l'arbre, il suffit de remplacer le sous-arbre correspondant par la fusion de ses sous-arbres gauche et droit.

```
let rec supprime arbre valeur =
  match arbre with
  | V -> V
  | N (g, v, d) ->
      if v = valeur then concat g d
      else if valeur < v then N (supprime g valeur, v, d)
      else N (g, v, supprime d valeur)
```

4. Quelques mots sur la complexité

Toutes les opérations que l'on vient de voir se programme de façon récursive en parcourant une branche depuis la racine. Il en découle qu'elles sont toutes de complexité en $\Theta(h)$ où h est la hauteur de l'arbre.

Il est donc primordial de s'assurer que h demeure en $\Theta(\log n)$ pour avoir une complexité optimale. De nombreuses méthodes existent pour cela, comme les arbres *AVL* ou les arbres *rouge-noir* que nous étudierons en TD.

5. Exercices

Exercice 6 Dans la fonction d'insertion, si la valeur insérée est déjà présente, la fonction va néanmoins créer de nouveaux nœuds de la racine jusqu'au nœud où se trouve la valeur. Idem lors de la suppression d'une valeur qui n'est pas présente dans l'arbre. Peut-on éviter cela ?

Exercice 7 (Fusion) Écrire une fonction qui, étant donné deux a.b.r., renvoie l'a.b.r. dont les clés est la réunion des clés deux arbres passés en argument.

Exercice 8 (Tri) Expliquer comment on peut utiliser les arbres binaires de recherche pour effectuer un tri.

Exercice 9 (Ancêtre commun) Un ancêtre commun de deux nœuds n_1 et n_2 d'un arbre est un nœud dont descendent n_1 et n_2. Ainsi, la racine d'un arbre et toujours ancêtre commun de toute paire de nœuds de l'arbre.

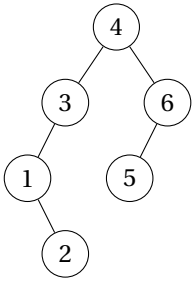
1. Montrer que pour tout entier h fixé, des nœuds n_1 et n_2 admettent **au plus** un ancêtre commun à la profondeur h.

On appelle **plus proche ancêtre commun** (ou p.p.a.c.) de n_1 et n_2 leur ancêtre commun de profondeur maximale. D'après la question précédente, celui-ci est unique.

2. Prouver que tout ancêtre commun de n_1 et n_2 est ancêtre de leur p.p.a.c..
3. En fixant deux nœuds n_1 et n_2, montrer que leur p.p.a.c. est…
   (a) n_1 si n_2 est un descendant de n_1 ;
   (b) n_2 si n_1 est un descendant de n_2 ;
   (c) l'unique nœud n de l'arbre tel que n_1 appartient au fils gauche (resp. droit) de n, et n_2 appartient au fils droite (resp. gauche) de n sinon.
4. En déduire une fonction `ppac` qui, étant donné un a.b.r. `arbre` et deux valeurs `p` et `q`, retourne la valeur de leur p.p.a.c..

Exercice 10 (Nombre de constructions)

1. Déterminer toutes les permutations possible de $[\![1,6]\!]$ qui donne l'arbre suivant en effectuant dans insertions aux feuilles.



2. Montrer, plus généralement, que le nombre de permutations conduisant à un a.b.r. donné de taille n est égal à $n!$ divisé par le produit des tailles de tous ses sous-arbres non vides (y compris lui-même).

Exercice 11 (Complexité en moyenne d'une recherche) On considère les a.b.r. obtenus en inserant tous les entiers de 1 à n à partir de l'arbre vide, les $n!$ ordres d'insertion étant supposés équiprobables.
On veut calculer la complexité moyenne a_n, en nombre de comparaisons, de la recherche d'une clé dans l'arbre.

1. Montrer que la probabilité que la clé i soit la racine est égale à $\frac{1}{n}$.
2. En notant $a_{i,n}$ la complexité moyenne de recherche d'une clé dans un arbre de taille n et dont la racine a pour clé i, montrer que :

$$a_{i,n} = \frac{i-1}{n}\big(a_{i-1}+2\big) + \frac{1}{n} + \frac{n-i}{n}\big(a_{n-i}+2\big)$$

3. En déduire que :

$$a_n = 2 - \frac{1}{n} + \frac{2}{n^2}\sum_{i=1}^{n-1} i a_i$$

4. En calculant $n^2 a_n - (n-1)^2 a_{n-1}$, montrer que

$$a_n = \frac{1}{n^2}\big((n^2-1)a_{n-1} + 4n - 3\big)$$

5. Prouver que $a_n = O\big(\log_2(n)\big)$ (on pourra considérer $u_n = \frac{n}{n+1}a_n$).

Exercice 12 (d'après Arne Andersson, *A Note on Searching in Binary Trees*) Lors de la recherche dans un a.b.r., on fait environ trois tests par étape de récursion :

☞ le filtrage du nœud courant,

☞ si l'on a un nœud interne, le test d'égalité entre la valeur du nœud courant et celle recherchée ;

☞ sinon, la comparaison permettant de savoir dans quel sous-arbre poursuivre

la recherche.

1. Comme améliorer facilement cela pour ne faire, en moyenne, qu'environ 2,5 tests par étape de récursion ?

On peut encore améliorer cela. Pour cela, considérons la fonction ci-dessous.

2. Justifier que la fonction `cherche2` effectue correctement la recherche d'une valeur dans un arbre binaire de recherche (on pourra s'intéresser au rôle de la variable `candidat` et essayer la fonction sur des exemples).
3. Avec cette fonction, à chaque recherche, on est obligé de parcourir une branche jusqu'à arriver à une feuille. À quelles conditions peut-on considérer que la fonction `cherche2` est plus rapide que `cherche` ?

```
let cherche2 arbre valeur =
  let rec aux arbre candidat =
    match arbre with
    | F -> candidat
    | N (g, v, d) ->
        if valeur < v then
          aux g candidat
        else
          aux d (Some v)
  in
  match aux arbre None with
  | None -> false
  | Some v -> v = valeur
```