TP 6 : Implémentation des arbres binaires de recherche

Le but de ce TP est de manipuler les arbres binaires de recherche que l'on a vu en cours, et de les utiliser pour représenter des dictionnaires et des ensembles. Leur implémentation par les ABR est explicitement au programme de l'option : ce que le programme attend que vous connaissiez sans avoir beaucoup de contexte correspond grosso modo à l'exercice 1. Sans doute que pour vos révisions l'année prochaine, on va réfléchir à une version améliorée de ces structures de données.

Exercice 1 – Manipulation des arbres binaires de recherche

Dans ce TP, on utilise le type suivant :

```OCaml
type 'a abr =
  | V
  | N of 'a abr * 'a * 'a abr
```

▶ **Question 1** Écrire une fonction `recherche : 'a -> 'a abr -> bool` déterminant si une étiquette est présente dans l'ABR.

★ **Question 2** Montrer que votre fonction est correcte par induction sur les arbres binaires de recherche.

▶ **Question 3** Écrire deux fonctions `max_abr : 'a abr -> 'a` et `min_abr : 'a abr -> 'a` calculant la plus grande et la plus petite étiquette d'un arbre binaire de recherche. On lèvera une exception si l'arbre en entrée est vide.

▶ **Question 4** En déduire une fonction vérifiant si un arbre non vide est un arbre binaire de recherche. Estimer sa complexité (ou au moins trouver une relation de récurrence vérifiée par la complexité de votre fonction).

On va maintenant coder une fonction d'*insertion* dans un arbre binaire de recherche : on a un ABR A et une nouvelle étiquette x, on veut construire un nouvel ABR B qui contient toutes les étiquettes de A et x, avec la stratégie suivante :

- si l'étiquette x est strictement inférieure à l'étiquette de la racine de A, on insère x récursivement dans le fils gauche,
- si l'étiquette x est strictement supérieure à l'étiquette de la racine de A, on insère x récursivement dans le fils droit,
- si on tombe sur un arbre vide, on la remplace par une feuille `N (V, x, V)`.

Ainsi, une insertion avec cette stratégie créera toujours une nouvelle feuille, à la place d'un arbre vide. On considérera que si l'on insère une étiquette déjà présente dans un ABR, on renvoie l'ABR non modifié.

▶ **Question 5** Implémenter cette stratégie par une fonction `insere : 'a -> 'a abr -> 'a abr`.

★ **Question 6** Déterminer la complexité de la fonction `insere`.

▶ **Question 7** Écrire une fonction `abr_of_list : 'a list -> 'a abr` qui, depuis une liste d'éléments `l`, renvoie l'arbre dans lequel on a successivement inséré les éléments de `l`.

▶ **Question 8** En déduire une fonction `tri : 'a list -> 'a list` qui renvoie la liste en entrée triée en calculant l'arbre binaire de recherche associé à l'entrée par `abr_of_list`, puis en parcourant cet arbre binaire de recherche par un parcours infixe. *Il y a une condition à imposer sur l'entrée pour que cette fonction soit correcte : laquelle ?*

Revenons à notre objectif initial, qui est d'utiliser ces arbres binaires de recherche pour implémenter des ensembles. On rappelle notre façon de voir : on représente un ensemble Y par un ABR dont l'ensemble des étiquettes est exactement X. On a implémenté la recherche dans un ensemble, l'insertion d'un élément, la production d'un ensemble vide. On pourrait y ajouter facilement une fonction qui à un ensemble associe un élément quelconque à l'intérieur (en renvoyant la racine de l'ABR, par exemple). N'empêche qu'il manque un gros bout : la suppression d'un élément d'un ensemble.

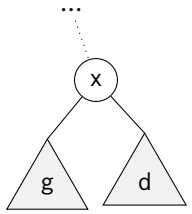
Exercice 2 – Suppression d'une étiquette dans un arbre binaire de recherche

On implémente l'opération qu'il nous manquait pour implémenter les ensembles dans une structure d'arbre binaire de recherche : étant donné une étiquette x et un ABR A, renvoyer un nouvel ABR contenant les mêmes étiquettes que A moins x.
Pour cet exercice, je vous conseille **vivement**, si ce n'est pas déjà fait, de sortir un papier et un crayon pour réfléchir à une façon d'implémenter un arbre binaire de recherche. On va se baser sur le squelette de code suivant :

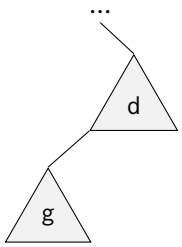
```OCaml
let rec supprime a x = match a with
  | V -> ...
  | N (V, e, d) when e = x -> ...
  | N (g, e, V) when e = x -> ...
  | N (g, e, d) when e < x -> ...
  | N (g, e, d) when e > x -> ...
  | N (g, e, d)            -> ... (* e = x *)
```

▶ **Question 1** Compléter les lignes 2 à 6.

Dans le cas de la ligne 7, où l'on a trouvé le nœud contenant `e = x`, c'est plus compliqué. On cherche à créer un nouvel arbre qui ne contient que les étiquettes de `g` et `d`.



▶ **Question 2** Implémenter la stratégie qui consiste à mettre `g` tout à gauche de `d`, de manière à obtenir un arbre de la forme ci-contre. Justifier que l'on obtient bien un ABR. *On complètera la ligne 7 et on prendra soin de tester cette fonction pour vérifier qu'elle fait bien ce que l'on attend.*



★ **Question 3** Est-ce que vous voyez un désavantage à cette stratégie ?

Une stratégie similaire ajouterait `g` tout à droite de `d`. On propose une autre stratégie :

- d'abord, on remarque que l'étiquette minimale `m` de l'arbre `d` qui ne peut pas avoir de fils gauche,
- supprimer `m` de l'arbre `d` tombe donc dans le cas ligne 3, on calcule alors `d'` l'ABR obtenu après suppression de `m` dans `d`,
- on se rend compte que l'arbre `N (g, m, d')` est bien un arbre binaire de recherche qui convient !

▶ **Question 4** Implémenter cette stratégie dans une nouvelle fonction `supprimer2 : 'a abr -> 'a -> 'a abr`. *On pourra écrire une fonction auxiliaire* `supprime_min : 'a abr -> 'a * 'a abr` *tel que* `supprime_min a` *est le couple contenant l'étiquette minimale de* `a` *et un ABR contenant le reste des étiquettes.*

▶ **Question 5** En général, a-t-on `insere (supprime2 a x) x = a` ?

Exercice 3 – Complexité des opérations

On suppose que l'on utilise la fonction `supprimer2` pour réaliser des suppressions d'étiquette.

▶ **Question 1** Montrer que le nombre de comparaisons utilisées lors d'un appel à `recherche`, `insere`, `supprime2` sur un arbre A est en $\mathcal{O}(h(A))$.

▶ **Question 2** Dans quel cas peut-on en conclure que la complexité temporelle de l'algorithme est en le même grand-$\mathcal{O}$ que le nombre de comparaisons ?

▶ **Question 3** Exprimer ces complexités dans le meilleur cas et dans le pire cas en fonction de $n(A)$.

Exercice 4 – Implémentation des dictionnaires

On va maintenant s'intéresser au type abstrait dictionnaire. Pour les implémenter, on va légèrement adapter le type `abr` :

```OCaml
type ('a, 'b) abr_dict =
  | V2
  | N2 of 'a * 'b * ('a, 'b) abr_dict * ('a, 'b) abr_dict
```

▶ **Question 1** Implémenter les opérations suivantes :

- `vide : ('a, 'b) abr_dict` égal au dictionnaire vide,
- `ajoute : 'a -> 'b -> ('a, 'b) abr_dict -> ('a, 'b) abr_dict` ajoutant une association clé-valeur à un dictionnaire,
- `supprime : 'a -> ('a, 'b) abr_dict -> ('a, 'b) abr_dict` supprimant une association clé-valeur d'un dictionnaire,
- `find : 'a -> ('a, 'b) abr_dict -> 'b` renvoyant la valeur associée à une clé dans un dictionnaire.

▶ **Question 2** Implémenter une variante de `find` de type `find_opt : 'a -> ('a, 'b) dict -> 'b option` qui renvoie `Some v` avec `v` la valeur associée à la

clé en entrée si elle existe, `None` sinon.

On remarque ici que la complexité des opérations découle naturellement de celles calculées dans l'Exercice 3. Cependant, il faut faire attention au type utilisé pour les clés : par exemple, si on utilise `string`, alors une comparaison sera en grand-$\mathcal{O}$ de la taille des chaînes de caractères. On utilisera plutôt des *tries* ou des tables de hachage dans ce cas.

Exercice 5 – On continue à manipuler les arbres binaires de recherche

▶ **Question 1** Écrire une fonction `separe : 'a abr -> 'a -> 'a abr * 'a abr` tel que `separe a x` renvoie un couple d'ABR contenant toutes les étiquettes strictement inférieures à `x` d'un côté, et toutes les étiquettes supérieures ou égales à `x` de l'autre. *Sa complexité temporelle sera linéaire en la hauteur.*

▶ **Question 2** Écrire une fonction linéaire en le nombre de nœuds d'un arbre vérifiant qu'il est bien un arbre binaire de recherche : `est_abr : 'a abr -> bool`.

On peut utiliser les arbres binaires de recherche pour implémenter le type abstrait de *multiensembles* : on utilise les mêmes opérations que les ensembles, sauf qu'un élément peut y être présent plusieurs fois : alors, supprimer un élément de l'ensemble n'en supprime qu'une occurrence.

▶ **Question 3** Adapter les fonctions écrites pour le type `('a, int) dict`, en utilisant les valeurs de l'ensemble comme clés et leur multiplicité comme valeur associée. On supprimera les valeurs dont la multiplicité redescend à 0.

▶ **Question 4** Écrire la fonction `nb_occurences : ('a, int) dict -> 'a -> int` renvoyant le nombre d'occurrences d'un élément dans le multiensemble.

▶ **Question 5** Écrire une fonction `('a, int) dict -> 'a -> int` donnant la taille du multiensemble (c'est-à-dire son nombre d'éléments en comptant leur multiplicité).

▶ **Question 6** Écrire une fonction `nth_plus_petit : ('a, int) dict -> int -> 'a` tel que `nth_plus_petit d k` renvoie la k^e plus petite valeur de d en comptant les multiplicités.

▶ **Question 7** Déterminer leur complexités.

On pourrait les améliorer en utilisant un type d'arbre binaire de recherche qui *connaît* la taille du sous-multiensemble associé à chaque nœud (en ajoutant un argument au constructeur `N2`, par exemple).

Exercice 6 – La hauteur moyenne d'un arbre binaire de recherche est en $\mathcal{O}(\log_2 n)$

Pour simplifier (énormément) l'analyse, on va limiter grandement notre façon de produire des arbres binaires de recherche. Pour $n \in \mathbb{N}$, on produit un arbre binaire de recherche à n éléments de $[\![1, n]\!]$ aléatoirement en choisissant une permutation de ces n éléments uniformément parmi les $n!$ possibles et en insérant ces n éléments successivement.

Théorème 1

La hauteur moyenne des ABR produits aléatoirement par cette méthode est en $\mathcal{O}(\log_2 n)$.

Notons A_n la variable aléatoire égale à l'arbre produit par ces n insertions. On va introduire des variables aléatoires : X_n est la hauteur de $h(A_n)$, $Y_n = 2^{X_n}$, et R_n l'étiquette de la racine de A_n (c'est-à-dire le premier élément de la permutation, le premier élément inséré). Le théorème prétend donc que $\mathbb{E}[X_n] = \mathcal{O}(\log_2 n)$.

▶ **Question 1** Justifier sommairement que Y_n et $2\max(Y_{R_n-1}, Y_{n-R_n})$ sont deux variables aléatoires égales en loi (elles suivent la même loi de probabilité).

On peut montrer que $\mathbb{P}(R_n = k) = \frac{1}{n}$.

▶ **Question 2** Montrer que $\mathbb{E}[Y_n] = \frac{2}{n}\sum_{k=0}^{n-1}\mathbb{E}[\max(Y_{k-1}, Y_{n-k})]$. *On pourra introduire si besoin $Z_{n,k}$ la variable aléatoire égale à 1 si $R_n = k$, 0 sinon.*

▶ **Question 3** Montrer que $\mathbb{E}[Y_n] \leq \frac{2}{n}\sum_{k=0}^{n-1}(\mathbb{E}[Y_{k-1}] + \mathbb{E}[Y_{n-k}])$.

▶ **Question 4** En déduire une relation de récurrence forte vérifiée par $\mathbb{E}[Y_n]$ (c'est-à-dire en majorant $\mathbb{E}[Y_n]$ en fonction de $\mathbb{E}[Y_0] \dots \mathbb{E}[Y_{n-1}]$).

▶ **Question 5** En utilisant l'identité $\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}$, montrer que $\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$.

On peut conclure en utilisant l'inégalité de Jensen : pour une fonction convexe f et une variable aléatoire X, $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$.

▶ **Question 6** Montrer le Théorème 1.