TP #20 — Arbres binaires de recherche (auto-équilibrés)

Organisation des fichiers source. On conseille fortement de rédiger chaque exercice de ce sujet dans un fichier séparé, afin d'éviter les conflits entre fonctions & types de mêmes noms.

Consignes relatives aux tests. Pour chacun des exercices, les tests devront être écrits dans des fonctions dédiées qui pourront ensuite facilement être exécutées toutes ensemble, par exemple comme :

```
let _test_all () =
    assert (_test_add ()) ;
    assert (_test_del ()) ;
    (* snip *) ;
    true
```

Si vous faites les choses sérieusement, la majorité de votre code devrait être dédiée à des fonctions de test.

Exercice 1. *ABRs fonctionnels sans équilibrage — en OCaml*

Dans tout cet exercice, on utilisera le type :

```
type 'a tree = E | N of 'a tree * 'a * 'a tree
```

pour représenter des arbres binaires en OCaml

On rappelle qu'un *arbre binaire de recherche* (ou ABR) est un arbre binaire (construit sur un ensemble d'étiquettes totalement ordonnable) qui est ou bien un arbre vide `E`, ou bien de la forme `N (_A, b, _C)` et tel que les valeurs des étiquettes de l'arbre `_A` sont toutes strictement inférieures à l'étiquette `b`, et celles des étiquettes de l'arbre `_C` sont toutes strictement supérieures à `b`.
On cherche dans un premier temps à implémenter une fonction permettant de tester si un arbre binaire est un ABR.

1. Soit t un arbre binaire, montrez que la liste des étiquettes de t construite par un parcours en profondeur infixe est triée *ssi.* t est un ABR.
2. Déduisez-en une fonction :

   ```
   is_bst : 'a tree -> bool
   ```

   telle que `is_bst t` s'évalue à `true` si t est un arbre binaire de recherche, et `false` sinon.
3. Testez.
4. Quel est le coût de votre fonction, en temps et en espace ?

On souhaite écrire une nouvelle fonction de mêmes spécifications de coût en espace inférieur. (La condition d'être un ABR ne pouvant pas être vérifiée localement, on ne peut pas significativement améliorer le coût en temps).
On propose deux approches pour cela ; soit `N (_A, b, _C)` un arbre dont on veut déterminer s'il est ABR :

— la première (purement fonctionnelle) consiste à vérifier (récursivement) que les étiquettes du sous-arbre gauche (resp. droit) se trouvent dans l'intervalle admissible $[\![-\infty, b[\![$ (resp. $]\!]b, +\infty]\!]$), où $-\infty$ et $+\infty$ sont des valeurs spéciales respectivement strictement plus petite et plus grande que toute valeur du domaine des étiquettes ;
— la seconde (qui peut bénéficier de l'utilisation de variables mutables, bien que ce ne soit pas strictement nécessaire) consiste à vérifier récursivement que `a` est un ABR et calculer au passage la valeur de son élément maximum, vérifier que celui-ci est inférieur à `b`, mettre à jour ce maximum, puis récursivement vérifier que `c` est un ABR dont les étiquettes sont plus grandes que le maximum en suivant le même procédé.

5. Implémentez l'une des deux approches ci-dessus dans une fonction OCaml `is_bst'` (et éventuellement l'autre dans une fonction `is_bst''`).
Comme d'habitude, n'hésitez pas si besoin à introduire des fonctions intermédiaires (par exemple une fonction `int_okay` qui vérifie si son troisième argument appartient à l'intervalle défini par ses deux premiers). Dans les deux approches, l'utilisation d'un type `option` peut être une solution élégante pour représenter $\pm\infty$.
6. Testez.

On s'intéresse maintenant à l'utilisation & manipulation d'ABR.

7. Écrivez une fonction OCaml :
`mem : 'a -> 'a tree -> bool`
telle que pour un ABR `t`, `mem x t` s'évalue à `true` s'il existe un nœud dans `t` d'étiquette `x`, et `false` sinon.
Cette fonction devra avoir un coût pire-cas linéaire en la *hauteur* de `t`.
8. Écrivez une fonction :
`add : 'a -> 'a tree -> 'a tree`
telle que pour un ABR `t`, `add x t` construit un nouvel ABR `t'` dont l'ensemble des étiquettes est égal à l'ensemble des étiquettes de `t`, auquel on a ajouté `x` (s'il ne s'y trouvait pas déjà).
Cette fonction devra avoir un coût pire-cas linéaire en la *hauteur* de `t`.
9. Testez votre fonction. Notamment, vérifiez que celle-ci construit bien des *ABRs* et ne « supprime » pas d'éléments.
10. Écrivez une fonction OCaml :
`of_list : 'a list -> 'a tree`

telle que `of_list x` construit naïvement un ABR dont les éléments sont contenus dans la liste `x`.

Conseil. Cela peut-être une bonne occasion de manipuler `List.fold_left` et `Fun.flip` (pas vraiment au programme, mais bon…)

11. Testez. Vérifiez notamment que le résultat est un ABR, et qu'il contient les éléments voulus.

La suppression d'un élément dans un ABR est un peu plus subtile que l'ajout, et l'on propose de procéder en deux temps, comme vu en cours.

12. Écrivez une fonction OCaml :

`pop_min : 'a tree -> 'a * 'a tree`

telle que pour un ABR `t` non vide d'ensemble d'étiquettes L, `pop_min t` s'évalue en le couple `m, t'`, où $m = \min L$, et `t'` est un ABR d'ensemble d'étiquettes $L \setminus \{m\}$.

Cette fonction devra avoir un coût pire-cas linéaire en la *hauteur* de `t`.

13. Testez votre fonction. Notamment, vérifiez que celle-ci construit bien des *ABRs* et « supprime » bien exactement son minimum.

14. Écrivez une fonction OCaml :

`del : 'a -> 'a tree -> 'a tree`

telle que pour un ABR `t`, `del x t` construit un nouvel ABR `t'` dont l'ensemble des étiquettes est égal à celui de `t` dont on a supprimé `x` (s'il s'y trouvait).

On propose pour cela de procéder de la façon suivante : hors cas de base, pour supprimer `x` de `N (_A, x, _C)`, on supprime le minimum de `_C` (obtenant un élément `b` et un nouvel arbre `_C'`) et l'on construit le résultat comme `N (_A, b, _C')`.

Cette fonction devra avoir un coût pire-cas linéaire en la *hauteur* de `t`.

15. Testez votre fonction. Notamment, vérifiez que celle-ci construit bien des *ABRs* et « supprime » bien au plus un élément.

Exercice 2. *ABRs impératifs sans équilibrage — en C*

Dans tout cet exercice, on utilisera le type :

```
struct btree
{
    struct btree *left;
    struct btree *right;
    int val;
    size_t size;
};
```

pour représenter des arbres binaires d'étiquettes `int` en C. Le champ `size` permettra de stocker la taille (le nombre de nœuds non vides) de l'arbre dont la racine est représentée par l'objet de type `struct btree` correspondant. Ceci évitera de

coûteux recalculs dans certains cas et aidera dans l'écriture de tests, mais demande une attention particulière lors de l'écriture de toute fonction modifiant cette taille ; vous êtes prévenus !

Remarque. Comme toujours en C, l'utilisation des *sanitizers* est essentielle afin de faciliter la détection d'erreurs mémoire (y compris les fuites).

1. Écrivez une fonction C de signature :
   ```
   struct btree *
   make(int v, struct btree *k1, struct btree *k2)
   ```
   qui construit et renvoie l'adresse d'un nouvel objet de type `struct btree` de durée de vie allouée, qui représente un arbre dont la racine a une valeur (d'étiquette) `v` et les sous-arbres gauche et droit sont respectivement d'adresses `k1` et `k2`.
2. Écrivez une fonction C de signature :
   ```
   void destruct(struct btree *t)
   ```
   qui détruit (met fin à la durée de vie) de l'entièreté de l'arbre représenté par l'objet dont son argument contient l'adresse.
3. Écrivez une fonction C de signature :
   ```
   size_t fill_infix(struct btree *src, int *dst)
   ```
   qui « remplit » le « tableau » (supposé suffisamment grand) dont l'adresse est donnée par `dst` avec les éléments de l'arbre représenté par l'objet d'adresse `src` dans l'ordre d'un parcours en profondeur infixe, et renvoie le nombre d'éléments ainsi écrits.
4. Déduisez-en une fonction C de signature :
   ```
   int *to_arr(struct btree *t)
   ```
   qui construit un « tableau » dont les éléments sont ceux de l'arbre supposé non vide représenté par l'objet d'adresse `t` dans l'ordre d'un parcours en profondeur infixe, et renvoie son adresse.
5. Testez (si cela n'est pas déjà fait) l'ensemble de vos fonctions écrites jusque là. Vérifiez notamment l'absence de fuites mémoire (du mieux que possible).
6. Écrivez une fonction C de signature :
   ```
   bool is_bst(struct btree *t)
   ```
   qui décide si son argument est un arbre binaire de recherche.
7. Testez.
8. Écrivez une fonction C de signature :
   ```
   bool mem(struct btree *t, int v)
   ```
   qui décide si l'arbre binaire de recherche représenté par l'objet d'adresse `t` contient un nœud de valeur (d'étiquette) `v`.
9. Testez.
10. Si votre fonction `mem` était récursive, écrivez une variante `mem2` de mêmes signature & spécifications qui soit purement itérative.

11. Écrivez une fonction C non récursive de signature :

```
bool add(struct btree **t, int v)
```

qui modifie l'arbre binaire de recherche représenté par l'objet dont l'adresse est donnée par `*t` pour y ajouter un nœud de valeur `v` s'il ne s'en trouvait pas déjà un. Cette fonction renvoie `true` si un ajout a réellement été effectué, et `false` sinon.

Conseil. En l'absence initiale de garantie sur la présence de `v` dans l'arbre, on ne peut guère s'économiser une remontée pour la mise à jour des tailles, ou un test préalable avec `mem`. Cette dernière solution est sans-doute la plus simple.

12. Testez largement.

Les questions suivantes demandent d'implémenter la suppression dans un ABR. Elles sont nettement plus délicates que les précédentes, et peuvent être considérées comme optionnelles (dans le sens, *pas prioritaires*). Dans tous les cas, on conseille fortement de s'aider de schémas pour visualiser les différentes chaînes de références et les modifications nécessaires.

13. Écrivez une fonction C non récursive de signature :

```
int pop_min(struct btree **t)
```

qui supprime et renvoie l'élément de valeur minimale dans l'arbre représenté par `*t`, supposé non vide.

(Attention à la légalité des accès, aux fuites mémoires (libérez ce qu'il faut, mais pas plus !) et à la mise à jour des tailles.)

14. Écrivez une fonction C non récursive de signature :

```
bool del(struct btree **t, int v)
```

qui modifie l'arbre binaire de recherche représenté par l'objet dont l'adresse est donnée par `*t` pour y supprimer la valeur `v` si elle s'y trouvait. Cette fonction renvoie `true` si une suppression a réellement été effectuée, et `false` sinon.

15. Testez largement.

Exercice 3. *ABRs rouge-noirs*

Cet exercice a pour but d'implémenter en OCaml l'insertion dans des ABRs rouge-noirs. On représente de tels arbres par les types :

```
type colour = R | B
type 'a tree = E | N of colour * 'a tree * 'a * 'a tree
```

(On aurait également pu faire le choix de se passer d'un champ couleur et d'à la place dupliquer les constructeurs de nœud, comme :

```
type 'a tree = E
    | R of 'a tree * 'a * 'a tree
    | B of 'a tree * 'a * 'a tree
```

Ceci aurait allégé l'écriture des motifs, mais aussi augmenté les modifications des fonctions d'ABR déjà écrites.)

1. Écrivez une fonction OCaml :
   `height : 'a tree -> int`
   qui s'évalue en la hauteur (usuelle) de son argument.
2. Écrivez une fonction OCaml :
   `bheight : 'a tree -> int`
   qui s'évalue en la hauteur *noire* de son argument. On considérera pour cela que les nœuds vides `E` sont noirs et ont une hauteur noire de 1.
3. Écrivez une fonction OCaml :
   `is_rbt : 'a tree -> bool`
   qui décide si son argument est équilibré au sens rouge-noir. Cette fonction devra être suffisamment efficace pour permettre de réaliser des tests sur des arbres de taille au moins 1 000.
4. Testez.
5. Adaptez votre fonction `mem` de l'exercice 1 aux ABRs rouge-noirs.

Pour maintenir l'équilibrage (au sens rouge-noir) des arbres construits par la fonction d'insertion `add`, on va procéder de la façon suivante :

— on utilise une fonction intermédiaire `_add` identique à la fonction `add` de l'exercice 1, si ce n'est que son cas de base colorie son résultat en rouge (`E -> N (R, E, x, E)`) et ses cas récursifs appellent une fonction d'équilibrage `_bal` sur leur résultat ;
— la fonction d'insertion `add` se contente d'effectuer un appel à `_add` et de recolorier la racine du résultat en noir.

6. Adaptez votre fonction `add` de l'exercice 1 comme ci-dessus, en utilisant pour l'instant l'identité pour `_bal`.
7. Testez, et constatez notamment la nécessité d'un équilibrage non trivial.
8. Écrivez une fonction OCaml :
   `_bal : 'a tree -> 'a tree`
   qui rééquilibre localement son argument s'il se trouve dans l'un des quatre cas vus en cours, et ne fait rien (se comporte comme l'identité) sinon.
9. Testez. Vérifiez notamment que des arbres de taille environ 1 000 obtenus par ajouts successifs sont bien des ABRs équilibrés au sens rouge-noir. Faites notamment cela pour des arbres contenant $[\![0, n]\!]$ pour un certain n, et pour d'autres dont les éléments ont été générés aléatoirement (par ex. en utilisant `Random.int`).
10. Testez les performances de votre implémentation en mesurant le temps nécessaire à la construction d'un ABR contenant $[\![0, n]\!]$ pour $n = 2^i - 1$, $i \in [\![22, 24]\!]$ quand votre programme est compilé en code natif :

```
(* rbt.ml *)
(* snip *)
;;
_bench (1 lsl 22) (* 1 lsl 22 = 2^22 *)
```

puis

```
pierre@gma>ocamlopt -O3 rbt.ml
pierre@gma>time ./a.out
./a.out  2.41s user 0.12s system 99% cpu 2.544 total
```

N'hésitez pas à me demander de l'aide si besoin.

11. Écrivez une fonction OCaml :

```
rbt_sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

de mêmes spécifications que `List.sort`, qui utilise un arbre rouge-noir pour garantir un coût pire-cas quasi-linéaire en la taille de son second argument. Prenez garde à correctement gérer la possible présence de plusieurs éléments identiques dans la `list` à trier.

Indice (?). Il est possible d'écrire cette fonction de façon extrêmement compacte, une fois implémentées les bonnes abstractions de parcours. Par exemple, comment pourriez vous utiliser des fonctions :?

```
fold_left_mapi :
(int -> 'a -> 'b) -> ('c -> 'b -> 'c) -> 'c -> 'a list
-> 'c
_to_list_filter : ('a * 'b) tree -> 'a list
```

12. Testez sa correction en comparant le résultat de votre fonction avec celui de `List.sort` dans des cas variés.

13. Comparez ses performances avec `List.sort` dans le cas d'`int list` de quelques millions d'éléments déjà triés ou triés à l'envers.

FIGURE 1 – "Ribbit" says the red-black frog — Photo : Mauricio Rivera Correa

Exercice 4. *ABRs AVL par constructeurs intelligents*

Cet exercice a pour but d'implémenter en OCaml l'insertion et la suppression dans des ABRs AVL, en suivant l'approche par « constructeurs intelligents » vue en cours. Les arbres seront représentés par le :

```
type 'a tree = E | N of int * 'a tree * 'a * 'a tree
```

où le premier argument de `N` sert à stocker la hauteur de l'arbre dont ce nœud est la racine.

1. Écrivez une fonction OCaml :
   `height : 'a tree -> int`
   qui s'évalue en la hauteur de son argument. Cette fonction devra avoir un coût constant.

On va définir un premier constructeur (semi) intelligent pour faciliter la manipulation des hauteurs.

2. Écrivez une fonction OCaml :
   `_node : 'a tree -> 'a -> 'a tree -> 'a tree`
   telle que `_node k1 x k2` construit un arbre de racine d'étiquette `x` et de sous-arbre gauche (resp. droit) `k1` (resp. `k2`), et qui calcule et stocke sa hauteur à sa racine.
3. Copiez et adaptez (si besoin) vos fonctions de l'exercice 1 pour ce nouveau type. Les constructions d'arbres devront exclusivement utiliser le constructeur `E` ou le constructeur (semi) intelligent `_node` ; l'utilisation directe du constructeur `N` (hors fonction `_node`) ne devra servir qu'à l'écriture de motifs de filtrage.
4. Écrivez une fonction OCaml :
   `_is_locally_bal : 'a tree -> 'a tree -> bool`
   telle que `_is_locally_bal k1 k2` s'évalue à `true` si un arbre d'enfants `k1` et `k2` satisfait *localement* la condition d'équilibrage d'un arbre AVL, et `false` sinon.
5. Écrivez une fonction OCaml :
   `is_bst_avl : 'a tree -> bool`
   qui décide si son argument est un arbre binaire de recherche équilibré au sens AVL.
6. Testez et constatez notamment la nécessité d'un équilibrage non trivial. (L'utilisation de la fonction `of_list` peut être pratique pour cela.)
7. Écrivez une fonction OCaml :
   `_bal : 'a tree -> 'a tree`
   qui implémente le rééquilibrage comme vu en cours. Plus précisément, soit `t` un ABR dont les deux enfants sont des ABRs AVL de différence de hauteur au plus deux, `_bal t` doit construire un ABR AVL de mêmes éléments que `t` et de différence de hauteur avec `t` au plus un.

Conseil. L'écriture de cette fonction bénéficie particulièrement de l'utilisation de motifs de filtrage couplés à des conditions sémantiques, ce qui en OCaml peut se réaliser *via* le mot-clef `when` (hors programme). Par exemple, l'expression associée au motif syntaxico-sémantique :

```
N (_, _B, _, _C) when _is_locally_bal _B _C
```

ne sera évaluée que si l'argument de la `function` correspondante satisfait à la fois le motif (syntaxique) `N (_, _B, _, _C)` et est tel que l'expression `_is_locally_bal _B _C` s'évalue à `true`.
(De façon générale mieux vaut ne pas abuser du `when` ; il peut souvent être avantageusement remplacé par un simple test dans l'expression associée au cas de filtrage.)

La composition de `_bal` et `_node` fournit un constructeur vraiment intelligent, suffisant pour maintenir l'équilibrage AVL.

8. Écrivez une fonction OCaml :

```
_bn : 'a tree -> 'a -> 'a tree -> 'a tree
```

qui compose simplement `_bal` et `_node`.

9. Modifiez toutes vos fonctions portant sur les arbres binaires de recherche en remplaçant les appels à `_node` par des appels à `_bn` (quand pertinent).

10. Testez. Vérifiez notamment comme à l'exercice 3 que des arbres de taille environ 1 000 obtenus par ajouts successifs sont bien des ABRs équilibrés au sens AVL. Faites de même pour des arbres obtenus par suppressions successives.

11. Testez les performances de votre implémentation comme à l'exercice 3, et comparez ces deux stratégies d'auto-équilibrage dans des cas simples.