TP : ABR en OCAML

Exercice 1. Ecrire une fonction `check` qui prend en paramètre un arbre de type

```
type 'a arbre = Nil | Noeud of ('a arbre * 'a * 'a arbre);;
```

et retourne un bouléen indiquant si l'arbre est ou non un ABR.

Exercice 2.

1. Ecrire une fonction `mini` (resp. `maxi`) qui retourne le minimum d'un ABR. Donner la complexité dans le pire des cas.
2. Écrire ensuite une fonction qui retourne le prédecesseur (s'il existe) de l'étiquette x dans le parcours en profondeur infixe SANS calculer la totalité de la liste.
   On demande deux versions : une avec propagation d'une exception, l'autre avec maintien en mémoire de la dernière fois qu'on a tourné à droite.

Exercice 3.

1. Soit u_n le nombre de squelettes d'arbres binaires à n nœuds.
   Exprimer u_n en fonction de nombres u_k d'indices inférieurs.
2. Soit un squelette S d'arbre binaire à n nœuds et n nombres distincts.
   Combien y a-t-il de façons différentes de placer les n nombres dans S pour en faire un ABR ?
3. Soit v_n le nombre de squelettes d'arbres binaires de recherches à n nombres distincts (il y a n étiquettes et elles sont toutes différentes).
   Evaluer v_n.

Exercice 4. Déterminer le nombre d'ABR de hauteur 2 associés aux éléments $\{1, 2, 3, 4, 5, 6\}$.
Même question pour la hauteur 3.

Exercice 5. Dans cet exercice on considère que les étiquettes de nos arbres sont de type :

```
type ('a,'b) bucket = {key : 'a; value : 'b};;
```

Le type **'a** est supposé totalement ordonné.
Ainsi, les ABR vont implanter la notion de dictionnaire immutable.

1. Écrire la fonction **trouve (y : 'a) (a : ('a,'b) bucket arbre) : 'b** qui retourne la valeur associée à la clé y dans le dictionnaire immutable a.
2. Nous avons vu en cours comment insérer une nouvelle valeur sous une feuille. Nous voulons maintenant insérer la nouvelle valeur à la racine. Il faut alors
   — mettre dans le fils gauche du nouvel arbre toutes les étiquettes dont la clé est strictement plus petite que celle du nouvel élément inséré ;
   — mettre dans le fils droit du nouvel arbre toutes les étiquettes dont la clé est strictement plus grande que celle du nouvel élément inséré ;

Si un élément de même clé est déjà présent que celle du nouvel élément à insérer est déjà présent, on renvoie l'arbre sans modification.
Écrire une fonction

insere_racine (b : ('a, 'b) bucket) (a : ('a, 'b) bucket arbre) : ('a, 'b) bucket arbre

qui insère le nouveau bucket b à la racine de a.

3. On s'intéresse à la suppression de nœud.
   (a) Ecrire la fonction
   sup_d_abr : ('a, 'b) bucket arbre -> ('a, 'b) bucket * ('a, 'b) bucket arbre qui prend en paramètre un arbre binaire de recherche et retourne le couple formé de son plus grand élément (c.a.d. l'élément de plus grande clé) et du reste de l'arbre (c.a.d l'arbre initial dont on a juste enlevé le plus grand élément).
   En clair, c'est la fonction de suppression du plus grand élément de l'arbre.
   (b) Écrire la fonction

   suppression_abr (a : ('a, 'b) bucket arbre) (x :'a) : ('a, 'b) bucket arbre

   qui supprime le nœud d'étiquette x dans a. Si aucun nœud d'étiquette x n'est présent, on convient de renvoyer l'arbre à l'identique.
   On fait une recherche du nœud de clé x. Si c'est une feuille, on la remplace par l'arbre vide. Si c'est un nœud à un fils, on met ce fils à la place, sinon, on utilise la fonction précédente.

1 Tas

Exercice 6.
1. Montrer que dans un tas représenté par un tableau Caml à $n+1$ cases (avec première case non significative), les feuilles correspondent aux indices strictement supérieurs à $\lfloor \frac{n}{2} \rfloor$.
2. Montrer que dans un tas de taille n et de hauteur h, le nombre de noeuds de profondeur k est majoré par $\lfloor \frac{n}{2^{h-k}} \rfloor$.
3. Au lieu de majorer le nombre de nœuds d'une *profondeur* donnée (distance à la racine), on majore maintenant le nombre de nœuds d'une *hauteur* donnée (distance aux feuilles).
   Montrer que dans un tas de taille n, le nombre de noeuds de hauteur h est majoré par $\lceil \frac{n}{2^{h+1}} \rceil$.

Exercice 7. Donner une fonction **find_last** qui prend en paramètre un tas-max représenté par un tableau d'entiers et l'indice d'un nœud **N** pour retourner l'indice du dernier descendant de **N** (dans l'ordre du parcours en largeur) s'il y en a un. Si il n'y a pas de descendant, la fonction retourne l'indice donné en entrée.

Exercice 8. Ecrire une fonction `min_tas_max` qui retourne l'élément minimal d'un tas-max d'entier.

Exercice 9. Un arbre binomial d'ordre p est :
— un arbre réduit à une feuille si $p = 0$

— un arbre qui admet p fils qui sont respectivement des arbres binomiaux d'ordre $p-1, p-2, \ldots, 1, 0$.

1. Dessiner le squelette d'un arbre binomial d'ordre 4, puis exprimer en fonction de p :
   — la taille d'un arbre binomial d'ordre p
   — la hauteur d'un arbre binomial d'ordre p ;
   — le nombre de nœuds de profondeur k d'un arbre binomial d'ordre p.
2. Un tas binomial est une arbre binomial qui a une propriété de tas : l'information portée par un nœud est (dans le cas d'un tas-min) inférieure ou égale à l'information portée par chacun de ses fils. Nous allons étudier un algorithme de conversion d'un tas binaire de taille 2^p en un tas binomial d'ordre p sans faire aucune comparaison.

   Un tas binaire A de taille 2^p (p > 1) est constitué d'une racine r, d'un sous-arbre gauche F_g qui est un tas de taille 2^{p-1} et d'un sous-arbre droit F_d qui est un tas de taille $2^{p-1}-1$. Décrire un algorithme qui transforme en un tas binaire de taille 2^{p-1} le sous-arbre droit F_d et la racine r. Précisez le nombre d'opérations effectuées.
3. Considérons que les deux tas binaires obtenus, F_g et le nouveau tas créé à la question précédente, ont été transformés (récursivement) en tas binomiaux d'ordre $p-1$.

   Expliquer comment les réunir pour former un tas binomial d'ordre p.
4. Illustrer cet algorithme en transformant en tas binomial le tas binaire plus bas.
5. Calculer le nombre total d'échanges dans un tas binaire ainsi que le nombre de concaténations de tas binomiaux effectués lors de la conversion d'un tas binaire de taille 2^p . Sachant que ces opérations peuvent être réalisées à coût constant, quel est le coût total de l'algorithme ?

   On rappelle le *théorème de d'Alembert* :

   Théorème 1.1. *Soit (u_n) une suite de réels strictement positifs telle que le quotient $\frac{u_{n+1}}{u_n}$ possède une limite $\ell \in [0; +\infty]$.*

   (a) Si $\ell < 1$ alors la série $\sum u_n$ converge ;

   (b) Si $\ell > 1$ alors la série $\sum u_n$ diverge ;

   (c) Si $\ell = 1$, on ne peut rien conclure sur la convergence de la série.

FIGURE 1 – Un tas-min binaire à transformer en tas binomial.