Examen. Session 1. Documents autorisés. Durée 3h.

1 Arbre binaire de recherche d'entiers

Un arbre binaire de recherche est type particulier d'arbre binaire adapté à la recherche de clé. Dans un tel arbre, chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci. Il suffit alors de comparé la clé recherchée au nœud courant pour savoir s'il faut la rechercher dans le sous-arbre droit ou le sous-arbre gauche.

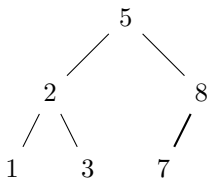

FIGURE 1 – Exemple d'arbre binaire de recherche

Exercice 1 (Type) Définir le type `intbintree` des arbres binaires d'entiers.

Exercice 2 (Opérations) Donner les contrats et définir les fonctions suivantes : `inserer`, `retirer`, `rechercher`, qui respectivement renvoient le sous-arbre gauche, le sous-arbre droit et la racine d'un arbre binaire hétérogène non vide (garanti par typage).

Exercice 3 (Parcours) On souhaite définir une fonction `parcourir` qui transforme un arbre binaire de recherche en une liste ordonnée de ses éléments. Quel est le type de parcours à effectuer ? Définir la fonction `parcourir`.

Pour que la recherche dans un arbre binaire de recherche se fasse en temps logarithmique il est nécessaire que l'arbre soit équilibré. Un arbre binaire est dit équilibré si pour chaque nœud de l'arbre la différence entre la hauteur du fils gauche et celle du fils droit est comprise entre 1 et -1. Par exemple, l'arbre donné par la figure 1 est équilibré.

Pour pouvoir équilibrer un arbre, on utiliser des opérations de rotations gauche et droite. La rotation droite d'un arbre consiste en les étapes suivantes :

— Le fils gauche devient la racine de l'arbre
— Son fils gauche reste son fils gauche
— La racine initiale devient son fils droit et son fils droit initial devient de fils gauche de la racine initiale
— Le fils droit de la racine initiale n'est pas modifié

La rotation gauche correspond à l'opération inverse. Ces deux opérations correspondent à ce qu'il se passe dans la figure 2.

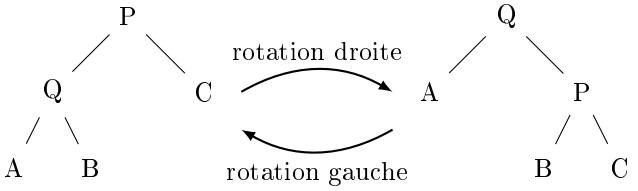

FIGURE 2 – Rotation

Pour équilibrer un arbre nous allons explorer deux méthodes, une utilisant la hauteur et l'autre le cardinal.

Exercice 4 (Hauteur et Cardinal) Écrire de manière récursive terminale les fonctions suivantes :

1. hauteur, qui donne la hauteur d'un arbre binaire
2. cardinal, qui donne le nombre de nœuds dans un arbre binaire

Exercice 5 (Équilibrage Cardinal) Écrire une fonction equilibrageCardinal, utilise les rotations pour équilibrer un arbre binaire de recherche de manière à ce que pour chaque nœud de l'arbre, la différence de cardinal entre son fils gauche et son fils droit est au plus d'un. Justifiez qu'un tel arbre est équilibré dans le sens définit précédemment.

En pratique, il n'est pas nécessaire de rééquilibrer la totalité de l'arbre à chaque étape mais seulement de faire les opérations d'insertion de manière à préserver le caractère équilibré de l'arbre. On remarque que ces opérations peuvent déséquilibrer l'arbre mais seulement au point où la différence de hauteur entre le fils droit et le fils gauche d'un nœuds serait égale à 2. Pour l'insertion le schéma de récursion est donc le suivant :

— Si l'arbre est vide on insère directement la clé
— Sinon, on fait une insertion **équilibrée** dans le bon fils du nœud en fonction de sa valeur
— Si on constate que cette insertion crée un déséquilibre entre le fils droit et le fils gauche alors on effectue des rotations pour corriger. Supposons le fils gauche plus haut que le fils droit alors :
  — Si le fils gauche du fils gauche est plus haut que le fils droit du fils gauche, une rotation droite suffit.
  — Sinon, il faut faire une rotation gauche-droite, c'est à dire une opération de rotation gauche sur le fils gauche suivi d'une opération de rotation droite sur le fils droit.

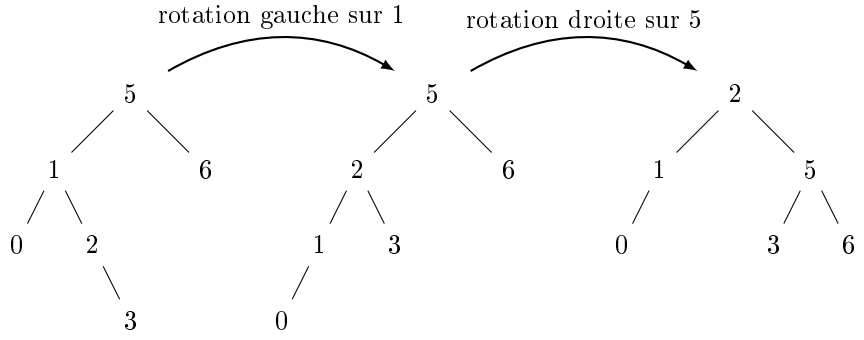

FIGURE 3 – Équilibrage avec rotation gauche-droite

Exercice 6 (Insertion et retrait équilibrées) Écrire les fonctions :

1. retirereq , qui effectue le retrait équilibrée d'un entier dans l'arbre.
2. insererreq, qui effectue l'insertion équilibrée d'un entier dans l'arbre.

Exercice 7 (Interface et module) Définissez une interface ensembleEntiers contenant les fonctions vide, inserer , retirer , rechercher et un module avlEntiers qui implémente cette interface à l'aide d'arbres binaires de recherche équilibrés.

2 Formules propositionnelles

Une formule propositionnelle est une formule logique qui peut être composée de variables (représentés par une chaine de caractère), des valeurs booléennes Vrai et Faux et des connecteurs logiques Et, Ou et Non. Par exemple : Et (x, Non y) représente la formule x et non y.

Exercice 8 (Type) Définir le type formule des formules propositionnelles et le type valuation qui est une table d'association entre les variables et un booléen.

Exercice 9 (Évaluation) Écrire le contrat et implémenter la fonction evaluer qui prend en paramètre une formule et une valuation et renvoi l'évaluation de la formule pour cette valuation.

Exercice 10 (Satisfaction) Écrire le contrat et implémenter la fonction satisfait qui prend en paramètre une formule et renvoi la liste des valuations (ne contenant que les variables appartenant à la formule) pour lesquelles cette formule est vrai.

Exercice 11 (Implication) Écrire le contrat et implémenter la fonction implique qui prend en paramètre deux formules et dit la première formule implique la deuxième, c'est à dire qu'à chaque fois que la première formule est vraie, la deuxième l'est aussi.

Dans la suite on cherche à enrichir petit à petit nos formules. On commence en retirant les variables pour simplifier au maximum nos formules.

Exercice 12 (Interface formules propositionnelles) Écrire une interface PropSimple représentant le type des formules propositionnelles sans variables et EvalSimple définissant l'évaluation de ces formules.

Exercice 13 (Foncteur exemple) Écrire un foncteur correspondant à l'exemple Et (Ou (Faux, Vrai), Non Vrai).

L'ajout des variables impliquent de gérer correctement les valuations, ainsi notre traitement doit EvalVar doit renvoyer une fonction

Exercice 14 (Ajout des variables) Écrire une interface PropVar et un module EvalVar permettant d'ajouter les cas des variables. Définir également EvalSimpleVar qui propage la valuation dans le cas de l'évaluation des formules simples sans variables.

Exercice 15 (Expressions)
- Écrire une interface ExpSimple représentant les expressions d'entiers combinés par les opérations de multiplication et addition.
- Écrire un module EvalExp permettant d'ajouter les cas des variables.
- Écrire une interface PropEgalite qui ajoute au formules propositionnelles les tests d'égalités entre deux expressions d'entiers.
- Écrire un module EvalEgalite qui traite le cas des tests d'égalité dans les formules.
- Définir par inclusion d'interface l'interface Prop définissant les formules propositionnelles avec variables et tests d'égalité.
- Définir par inclusion de modules le module EvalProp définissant le traitement de l'évaluation pour ces formules.

3 Flux et séries de Taylor

On va représenter et manipuler des séries de Taylor en 0 de fonctions à un paramètre. On n'étudiera pas les problèmes éventuels de convergence. À titre de rappel, voici deux séries classiques :

$$\frac{1}{1-x} = \sum_{i \in \mathbb{N}} x^i \qquad e^x = \sum_{i \in \mathbb{N}} \frac{x^i}{i!}$$

On représentera une série comme le flux infini des coefficients associés aux monômes x^i. On donne ci-dessous une interface du type abstrait Flux.

```
module type Flux =
sig
 type 'a t
 val vide : 'a t
 val cons : 'a -> 'a t -> 'a t
 val uncons : 'a t -> ('a * 'a t) option
 val apply : ('a -> 'b) t -> ('a t -> 'b t)
 val recursion : ('a -> 'a) -> ('a -> 'a t)
 val filter : ('a -> bool) -> 'a t -> 'a t
```

```
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
end
```

Un flux représente une liste potentiellement infinie d'éléments :
- vide représente un flux vide ne contenant aucun élément
- cons permet d'ajouter un élément au début du flux
- uncons permet de récupérer l'élément au début du flux ainsi que la suite du flux (c'est l'inverse de la fonction cons)
- apply permet d'appliquer un flux de fonction aux éléments correspondant dans un autre flux
- recursion permet de définir un flux par récurrence en prenant en paramètre une fonction représentant la relation de récurrence et l'élément initial du flux
- filter permet de filtrer les éléments du flux selon une fonction booléenne
- map permet d'appliquer une fonction aux éléments d'un flux
- map2 permet d'appliquer une fonction à deux paramètre aux éléments de deux flux

Les deux séries données en exemple seront respectivement représentées par les flux suivants :

```
cons 1. (cons 1. (cons 1. (...
cons 0. (cons 1. (cons 0.5 (cons 0.3333... (cons 0.25 (cons 0.2 (...
```

Exercice 16 (Flux élémentaires)

1. Définir le flux nats : float Flux.t des entiers successifs en partant de 1.
2. Définir les flux correspondants aux fonctions exemples $x \mapsto \frac{1}{1-x}$ et $x \mapsto e^x$.

Exercice 17 (Opérations linéaires) Définir les quatre fonctions suivantes :

1. Addition de deux séries et multiplication d'une série par une constante.
2. Dérivée et primitive d'une série donnée, en utilisant le flux nats.

La multiplication de $f = \sum_{i \in \mathbb{N}} f_i * x^i$ par g, notée $f \times g$, peut être décomposée ainsi :

$$(\sum_{i \in \mathbb{N}} f_i * x^i) \times g = (f_0 * g) + x * ((\sum_{i \in \mathbb{N}} f_{i+1} * x^i) \times g)$$

Exercice 18 (Multiplication)

1. Définir l'opération de produit d'une série par le monôme x.
2. À l'aide des fonctions précédentes, définir l'opération de multiplication entre deux flux, en suivant le schéma proposé.