TD1.7 – Arbres binaires de recherche d'intervalles

Objectifs

À la fin de cette séance, vous devriez être capable de :
— écrire des fonctions de manipulation pour un type abstrait dynamique ;
— jongler avec des ABRI ...

Exercice 1 :

Considérons des arbres binaires dont les noeuds sont étiquetés par des intervalles (par exemple d'entiers). Nous définissons la relation suivante entre intervalles :

$$[a..b] < [c..d] \Leftrightarrow b \leq c$$

Note : ce n'est pas "exactement" une relation d'ordre. Pourquoi ?

Cependant, cette relation nous permet de définir des arbres binaires de recherche d'intervalles. Pourquoi ?

1. **Mise en route**
   a) Dessinez un ABRI d'une dizaine de noeuds. Est-il possible que des intervalles se chevauchent ? Est-il possible que des intervalles se « touchent » ?
   b) Définissez le type abstrait arbre, et les primitives d'accès et de construction qui vous seront utiles.
   c) Écrivez une fonction qui calcule la somme des longueurs des intervalles d'un ABRI.
   d) À quelle condition est-il possible d'insérer un intervalle $[a..b]$ dans un ABRI ? Écrivez une fonction qui insère un intervalle dans un ABRI. Précisez la convention adoptée pour indiquer si l'insertion a été effectivement possible.

2. **Parcours**
   a) Écrivez une fonction qui vérifie si un arbre binaire d'intervalles est un ABRI.
   b) Écrivez un fonction qui calcule la longueur du plus grand "trou" entre deux intervalles (d'un ABRI).

3. **Transformation**
   a) Écrivez une fonction qui transforme un ABRI en "tassant" tous les intervalles à gauche : les intervalles seront alors tous contigüs.

Correction de l'exercice 1

1. **Mise en route**
   a)
   b) Opérations :
   — Vide : () → ABRI
   — EstVide : ABRI → bool
   — Inf : ABRI → entier
   — Sup : ABRI → entier
   — G : ABRI → ABRI
   — D : ABRI → ABRI
   — Creer : ABRI, entier, entier, ABRI → ABRI
   Axiomes :
   — Inf(Creer(g, inf, sup, d)) = inf
   — etc idem pour Sup, G et D

— EstVide(Vide()) = vrai
— EstVide(Creer(A, a, b, B)) = faux
On utilisera les notations allégées :
— A.Inf pour Inf(A) etc
Et la primitive Creer_feuille(a, b) équivalent à Creer(Vide(), a, b, Vide())

c) Il suffit d'effectuer un parcours quelconque de l'arbre, par exemple en profondeur, et d'additionner la longueur de l'intervalle de la racine avec les valeurs renvoyées par les 2 appels récursifs.

d) L'insertion est possible si l'intervalle ne chevauche aucun des intervalles présents dans l'ABRI.
La fonction *inserer* renvoie un booléeen qui vaut vrai ssi l'insertion a été possible.
Données : un ABRI A, deux entiers a et b
Résultat : un booléen indiquant si l'insertion a été possible. l'ABRI est modifié par effet de bord.
Précondition : A est un ABRI, $a \leq b$
Postcondition : si $[a..b]$ ne chevauche aucun intervalle présent initialement dans A, cet intervalle a été inséré dans A.

```
Inserer(A : ABRI, a, b : entiers) renvoie booléen
si a < A.Inf < b ou a < A.Sup < b alors OK = faux
sinon
    si b ≤ A.inf alors
        si EstVide(A.G) alors
            A.G=Creer_feuille(a, b)
            Ok = vrai
        sinon
            OK = Inserer(A.G, a, b)
    sinon (idem à droite)
renvoyer Ok
```

2. **Parcours**

a) Si A n'est pas vide, A est un ABRI ssi A est une feuille ou si
— ses sous-arbres gauche et droit (non vides) sont des ABRI
— si $A.G$ n'est pas vide, la plus grande borne supérieure d'un intervalle du sous-arbre gauche de A est inférieure à $A.Inf$,
— et idem à droite.
En un seul parcours, on effectue cette vérification en calculant par effet de bord les valeurs extrêmes des intervalles de l'ABRI.

```
EstABRI(A : ABRI : min, max : entiers résultats) renvoie booléen // précondition A n'est pas vide
si EstFeuille(A) alors
    min = A.Inf
    max = A.Sup
    Ok = vrai
sinon
    si EstVide(A.G) alors
        minG = A.Inf
        maxG = A.Inf
        OkG = vrai
    sinon
        OkG = EstABRI(A.G, minG, maxG)
    idem à droite
    Ok = OkG et OkD et maxG ≤ A.Inf et A.Sup ≤ minD
    min = minG
    max = maxD
renvoyer Ok
```

b) En un seul parcours, on calcule le plus grand trou ainsi que les bornes extrémales de l'arbre.
Données : un ABRI A non vide
Résultats : 3 entiers trou, min et max
Postcondition : trou est la taille du plus grand trou, min est la plus petite borne inf d'un intervalle de A, et max la plus grande borne sup

Le plus grand trou entre 2 intervalles de A est le maximum entre
— le plus grand trou dans son sous-arbre gauche
— le plus grand trou dans son sous-arbre droit
— le trou entre l'intervalle de la racine et la plus grande borne sup à gauche,
— le trou entre l'intervalle de la racine et la plus petite borne inf à droite.

```
PlusGrandTrou(A : ABRI, trou, min, max : entiers résultats)
si EstFeuille(A) alors
    min=A.inf ; max=A.sup ; trou=0
sinon
    si EstVide(A.G) alors
        minG=A.inf ; maxG=A.inf ; trouG=0 ;
    sinon
        PlusGrandTrou(A.G, trouG, minG, maxG)
    idem à droite
    trou=max(trouG, trouD, A.inf-maxG, minD-A.sup)
    min=minG
    max=maxD
```

3. **Transformation**

a) Principe : l'intervalle le plus à gauche est conservé, et on tasse les autres intervalles contre lui.
Données : un ABRI A non vide
Résultats : ABRI A (transformé), et un entier max
Postcondition : A est tassé à gauche (son intervalle le plus à gauche est inchangé, et il n'y a plus aucun trou), max est la plus grande borne supérieure d'un intervalle de A.

```
TasserGauche(A : ABRI donnée-résultat ; max : entier résultat)
    TasserGaucheContre(A, BorneInf(A), max)
```

où BorneInf(A) est la borne inférieure de l'intervalle le plus à gauche de A et où TasserGaucheContre(A, min, max) tasse l'arbre A contre la valeur min, et renvoie dans max la borne sup de l'arbre tassé :

```
TasserGaucheContre(A : ABRI donnée-résultat ; min : entier ; max : entier résultat)
si EstVide(A.G) alors
  maxG=min
sinon
  TasserGaucheContre(A.G, min, maxG)
A.sup=A.sup-A.inf+maxG
A.inf=maxG
si EstVide(A.D) alors
  max=A.sup
sinon
  TasserGaucheContre(A.D, A.sup, maxD)
  max=maxD
```