Cartouche du document

Année : ING 1
Matière : Algorithmique II
Activité : Travail dirigé

Objectifs

Cet ensemble d'exercices a comme objectif :

- 1) de modèliser des problèmes à l'aide des types abstraits
- 2) d'utiliser le conteneur Vecteur

Type abstrait Vecteur
Concept
Ce type permet de modèliser une liste indicée u_{bi} , ..., u_{bs}.
Les entiers bi et bs sont fixés une fois pour toute à la création du vecteur.

Opérations de base
Constructeur Vecteur : creerVecteur(Entier bi, Entier bs) : Vecteur
Transformateur Vecteur : affVal(Entier i,**Element** e) : Vecteur
Observateur Vecteur : recVal(Entier i) : Entier
Observateur Vecteur : estInitialise(Entier i) : Booleen
Observateur Vecteur : borneInf() : Entier
Observateur Vecteur : borneSup() : Entier

Axiomes
creerVecteur(bi, bs).borneInf() = bi
creerVecteur(bi, bs).borneSup() = bs
i <= bs ET i >= bi => NON CreerRacine(bi, bs).estInitialise(i)
i <= v.borneSup() ET i >= v.borneInf() => v.affVal(i,e).recVal(i) = e
i <= v.borneSup() ET i >= v.borneInf() => v.affVal(i,e).estInit(i)

Sommaire des exercices

Corps des exercices

1 - Manipulation de suites

Énoncé :

On désire étudier des suites arithmétiques et géométriques

Question 1)

Énoncé de la question

Définir un type abstrait qui permet de manipuler indistinctement une suite arithmétique ou une suite géométrique

Solution de la question

Type abstrait TypeSuite
Ce type permet d'énumérer différents types de suites
Opération de base
Constructeur TypeSuite : suiteGeo() : TypeSuite
Opération de base
Constructeur TypeSuite : suiteAri() : TypeSuite

Type abstrait SuiteAriGeo
Ce type permet de modéliser une suite arithmétique ou géométrique
Opération de base
Constructeur SuiteAriGeo : creerSuiteAriGeo(Reel uO, Reel coeff, TypeSuite ts) : SuiteAriGeo
Opération de base
Observateur SuiteAriGeo : recUO() : Reel
Opération de base
Observateur SuiteAriGeo : recCoeff() : Reel
Opération de base
Observateur SuiteAriGeo : recTypeSuite() : TypeSuite

Axiomes
recU0(creerSuiteAriGeo(u0,coeff,ts)) = u0
recCoeff(creerSuiteAriGeo(u0,coeff,ts)) = coeff
recTypeSuite(creerSuiteAriGeo(u0,coeff,ts)) = ts

Question 2)

Énoncé de la question

Ajouter dans ce type une opération d'extension qui permet de calculer et stocker dans un conteneur adéquat les n premiers termes d'une suite

Solution de la question

Le source

```
Opération d'extension
Observateur SuiteAriGeo : stockerTermesSuiteAriGeo(Entier  n) : Vecteur
// Le paramètre implicite
Observé sag

Références locales
Reel  u, coeff
Entier  i
Vecteur termes
TypeSuite ts

// Début de l'algorithme
DEBUT
      termes  <--  creerVecteur(0,n)
      u  <--  sag.recU0()
      coeff  <--  sag.recCoeff()
      ts  <--  sag.recTypeSuite()
      termes.affVal(0,u)
      Pour i  <--  1 à n pas 1
            SI ts.estEgal(suiteAri()) ALORS  // Structure  conditionnelle
                  u  <--  u + coeff
            SINON // Alternative  conditionnelle
                  u  <--  u * coeff
            FINSI
            termes.affVal(i, u)
      FinPour
      retourner termes
FIN
```

Question 3)

Énoncé de la question

En utilisant l'opération précédente, écrire une opération d'extension qui permet de calculer la monotonie de la suite sur les n premiers termes.

Solution de la question

Le source

```
Type abstrait Montonie
 Ce type permet d'énumérer différents types de monotonie d'une suite
Opération de base
Constructeur Monotonie : nonMonotone() : Monotonie
Opération de base
```

Constructeur Monotonie : croissante() : Monotonie
Opération de base
Constructeur Monotonie : decroissante() : Monotonie

Opération d'extension
Observateur SuiteAriGeo : recMonotonieSuiteAriGeo(Entier n) : Monotonie
// **Le paramètre implicite**
Observé sag

Références locales
Reel diff1, diff2
Entier i
Vecteur termes

// **Début de l'algorithme**
```
DEBUT
    termes <-- sag.stockerTermesSuiteAriGeo(n)
    diff1 <-- termes.recVal(1) - termes.recVal(0)
    diff2 <-- copie(diff1)
    i <-- 2
    TantQue i <= n ET (diff1 * diff2) >= 0 Faire
        diff1.egalA(diff2)
        diff2 <-- termes.recVal(i) - termes.recVal(i-1)
        i <-- i + 1
    FinTantQue
    SI (diff1 * diff2) < 0 ALORS // Structure conditionnelle
        retourner nonMonotone()
    Sinon Si diff1 >= 0
        retourner croissante()
    SINON // Alternative conditionnelle
        retourner decroissante()
    FinSi
FIN Observateur
```

2 - Equation du second degré

Énoncé :

Cet exercice a pour but de résoudre une équation du seconde degré à l'aide d'un type abstrait.

Question 1)

Énoncé de la question

On vous demande de modéliser une équation du second degré dans R. Le type abstrait nommé EQ2D_V1 sera défini comme suit :

- - les opérations de base suivantes :
  - le constructeur creerEQ2D pour définir l'équation
  - un observateur recCoeff pour récuperer n'importe quel coefficient de l'équation
- - l'opération d'extension resoudreEQ2D de type observateur qui :
  - 1) calcule les racines réelles de l'équation,
  - 2) les rangent dans un objet d'un type X à définir,
  - 3) renvoie une référence de cet objet.

Solution de la question

Type abstrait RacinesEq2D
ce type modélise le stockage de la solution d'une équation du second degré

Opérations de base
Constructeur RacinesEq2D : creerRacinesEq2D(Vecteur racines) : RacinesEq2D
Constructeur RacinesEq2D : creerPasDeRacinesEq2D() : RacinesEq2D
Observateur RacinesEq2D : recNbRacines() : Entier
Observateur RacinesEq2D : recRacines() : Vecteur

Axiomes
creerRacinesEq2D(racines).recNbRacines().estEgal(borneSup(racines))
creerPasDeRacinesEq2D(racines).recNbRacines().estEgal(0)
creerRacinesEq2D(racines).recRacines().estEgal(racines)
creerPasDeRacinesEq2D().recRacines() = refNulle

Type abstrait Eq2D_V1
Ce type permet de modéliser la définition et la résolution d'une équation du second degré

Opérations de base
Constructeur Eq2D : creerEq2D(Reel a, Reel b, Reel c) : Eq2D
Observateur Eq2D : recCoeff(Entier no) : Reel

Axiomes
a = 0 => creerEq2D(a,b,c) = refNulle
creerEq2D(a,b,c).recCoeff(2) = a
creerEq2D(a,b,c).recCoeff(1) = b
creerEq2D(a,b,c).recCoeff(0) = c

Opération d'extension
Observateur Eq2D : resoudreEq2D() : RacinesEq2D
// Le paramètre implicite
Observé eq2d

Objets locaux
Reel a, b,c, delta, r, racineDelta
Vecteur res
RacinesEq2D req2d

// Début de l'algorithme
DEBUT
    // on récupère les coefficients de l'équation
    a <-- eq2d.recCoeff(2)
    b <-- eq2d.recCoeff(1)
    c <-- eq2d.recCoeff(0)

    // on calcule le discriminant
    delta <-- b * b - 4.0 * a * c

    // on envisage les différents cas de figure suivant la valeur du discriminant
    SI delta = 0 ALORS // **Structure conditionnelle**
        res <-- creerVecteur(1,1)
        // on calcule la racine double
        r <-- - b / (2.0 * a)
        // on stocke cette racine dans le vecteur res
        res.affVal(1,r)
        req2d <-- creerRacinesEq2D(res)
    SINON SI delta > 0 ALORS // **Alternative conditionnelle**
        res <-- creerVecteur(1,2)
        // on calcule les deux racines et on les stocke dans le vecteur res
        racineDelta <-- delta.racineCarree()
        r <-- (- b + racineDelta) / (2.0 * a)
        res.affVal(1,r)
        r <-- (- b - racineDelta) / (2.0 * a)
        res.affVal(2,r)
        req2d <-- creerRacinesEq2D(res)
    SINON // **Alternative conditionnelle**
        req2d <-- creerPasDeRacines()
    FinSi
    retourner req2d
FIN

Question 2)

Énoncé de la question

Si on appelle plusieurs fois l'opération resoudreEQ2D, on refait le calcul des racines. Définir un nouveau type abstrait EQ2D_V2 qui pallie à cette faiblesse.

Le type abstrait X devra nous permettre de connaître :

- **1) le nombre de racines**
- **2) l'ensemble des racines stockées dans un vecteur**

Solution de la question

Type abstrait EQ2D_V2
Ce type permet de modéliser la définition et la résolution d'une équation du second degré

Opérations de base
Constructeur EQ2D : creerEQ2D(Reel a, Reel b, Reel c) : EQ2D
Observateur EQ2D : recCoeff(Entier no) : Reel
Transformateur EQ2D : affRacines(RacinesEQ2D racines) : EQ2D
Observateur EQ2D : recRacines() : RacinesEQ2D

Axiomes
a = 0 => creerEQ2D(a,b,c) = refNulle
creerEQ2D(a,b,c).recCoeff(2).estEgal(a)
creerEQ2D(a,b,c).recCoeff(1).estEgal(b)
creerEQ2D(a,b,c).recCoeff(0).estEgal(c)
eq2d.affRacines(req2d).estEgal(req2d)

Opération d'extension
Transformateur EQ2D : resoudreEQ2D() : RacinesEQ2D
// Le paramètre implicite
Transformé eq2d

Objets locaux
Reel a, b,c, delta, r, racineDelta
Vecteur res
RacinesEQ2D req2d

// Début de l'algorithme
DEBUT
SI eq2d.recRacines() = refNulle ALORS **// Structure conditionnelle**
// l'équation est non résolue. On la résout

```
        // on récupère les coefficients de l'équation
        a <-- eq2d.recCoeff(2)
        b <-- eq2d.recCoeff(1)
        c <-- eq2d.recCoeff(0)

        // on calcule le discriminant
        delta <-- b * b - 4.0 * a * c

        // on envisage les différents cas de figure suivant la valeur du discriminant
        SI delta = 0 ALORS // Structure conditionnelle
            res <-- creerVecteur(1,1)
            // on calcule la racine double
            r <-- - b / (2.0 * a)
            // on stocke cette racine dans le vecteur res
            res.affVal(1,r)
            eq2d.affRacines(creerRacinesEQ2D(res))
        SINON SI delta > 0 ALORS // Alternative conditionnelle
            res <-- creerVecteur(1,2)
            // on calcule les deux racines et on les stocke dans le vecteur res
            racineDelta <-- delta.racineCarree()
            r <-- (- b + racineDelta) / (2.0 * a)
            res.affVal(1,r)
            r <-- (- b - racineDelta) / (2.0 * a)
            res.affVal(2,r)
            eq2d.affRacines(creerRacinesEQ2D(res))
        SINON // Alternative conditionnelle
            eq2d.affRacines(creerPasDeRacines())
        FinSi
    FinSi
    retourner eq2d
FIN Transformateur
```