ASI3 25 novembre 2016

Algorithmique avancée et programmation C

Durée : *1h30*
Documents autorisés : **AUCUN**

Remarques :
- Veuillez lire attentivement les questions avant de répondre.
- Le barème donné est un barème indicatif qui pourra évoluer lors de la correction.
- Rendez une copie propre.
- N'utilisez pas de crayon à papier sur votre copie.
- Une annexe est disponible en fin de sujet.

1 Liste à partir d'un ensemble (3 points)

On suppose que l'on possède la fonction suivante qui permet d'obtenir un naturel compris entre $vMin$ et $vMax$:
- **fonction** naturelAuHasard (vMin,vMax : **Naturel**) **: Naturel**

Proposez la fonction suivante qui calcule une liste qui possède les mêmes élements que ceux de l'ensemble donné mais dont l'ordre est non déterministe :
- **fonction** listeMelangee (e : Ensemble) **:** Liste

Solution proposée :

```
fonction listeMelangee (e : Ensemble) : Liste
    Déclaration  res : Liste
                 el : Element
                 pos : NaturelNonNul
debut
    res ← liste()
    pour chaque el de e
        pos ← naturelAuHasard(1,longueur(res)+1)
        inserer(res,pos,el)
    finpour
    retourner res
fin
```

2 Modélisation d'un labyrinthe (17 points)

L'objectif de cet exercice est d'étudier le modélisation d'un labyrinthe comme celui proposée par la figure 1.

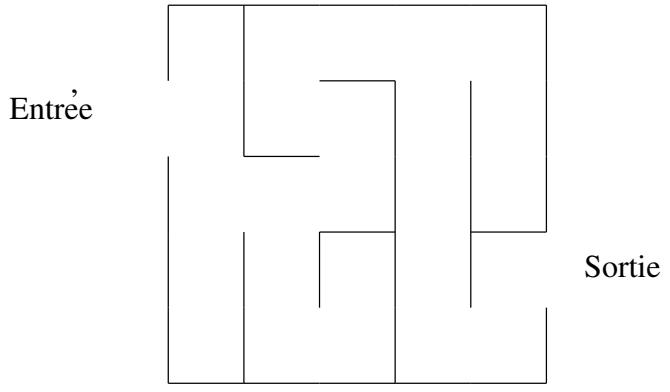

Figure 1 – Un labytinthe

En fait, un labyrinthe est composé de cases. Chaque case est identifiée par un naturel non nul (inférieur ou égale à la largeur multiplié par la hauteur du labyrinthe). On accède à une case à partir d'une case et d'une direction. Les directions possibles sont `Nord`, `Sud`, `Est` et `Ouest`.

Par exemple, comme le montre la figure 2, le labyrinthe précédent peut être considéré comme étant composé de 25 cases. La case numéro 6 est la case d'entrée. La case 20 est la case de sortie. La case 8 est accessible depuis la case 13 avec la direction `Nord`.

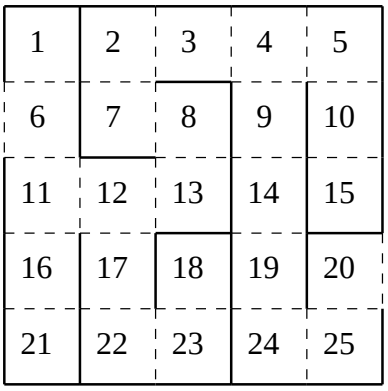

FIGURE 2 – Un labytinthe composé de cases

On considère posséder le type Direction suivant :
— **Type** Direction = {Nord,Sud,Est,Ouest}

2.1 Le TAD labyrinthe (3,5 points)

Les opérations disponibles sur un labyrinthe sont les suivantes :
— obtenir un labyrinthe convenablement initialisé (il y a un chemin qui va de l'entrée à la sortie) à partir d'une largeur, d'une hauteur ;
— obtenir la largeur, la hauteur, la case d'entrée, la case de sortie ;
— obtenir un ensemble de directions possibles depuis une case donnée ;
— obtenir la case accessible depuis une case et une direction ;
— obtenir l'ensemble des cases adjacentes d'une case (accessibles ou pas)

Proposez le TAD correspondant (sans la partie axiome et sémantique).

Solution proposée :

Nom:	Labyrinthe	
Utilise:	Direction, **NaturelNonNul**, Ensemble	
Opérations:	labyrinthe:	**NaturelNonNul** × **NaturelNonNul** → Labyrinthe
	largeur:	Labyrinthe → **NaturelNonNul**
	hauteur:	Labyrinthe → **NaturelNonNul**
	caseDEntree:	Labyrinthe → **NaturelNonNul**
	caseDeSortie:	Labyrinthe → **NaturelNonNul**
	directionsPossibles:	Labyrinthe × **NaturelNonNul** ↛ Ensemble<Direction>
	caseDestination:	Labyrinthe × **NaturelNonNul** × Direction ↛ **NaturelNonNul**
	caseAdjacentes:	Labyrinthe × **NaturelNonNul** ↛ Ensemble<**NaturelNonNul**>
Préconditions:	directionsPossibles(l,c):	c≤largeur(l)*hauteur(l)
	caseDestination(l,c,d):	c≤largeur(l)*hauteur(l) et estPresent(directionsPossibles(l,c),d)
	casesAdjacentes(l,c):	c≤largeur(l)*hauteur(l)

2.2 Conception préliminaire (2 point)

Donnez les signatures des fonctions et procédures correspondant aux opérations décrites ci-dessus.

Solution proposée :
— **fonction** labyrinthe (largeur,hauteur : **NaturelNonNul**) **:** Labyrinthe
— **fonction** largeur (laby : Labyrinthe) **: NaturelNonNul**
— **fonction** hauteur (laby : Labyrinthe) **: NaturelNonNul**
— **fonction** caseDEntree (laby : Labyrinthe) **: NaturelNonNul**
— **fonction** caseDeSortie (laby : Labyrinthe) **: NaturelNonNul**
— **fonction** directionsPossibles (laby : Labyrinthe, case : **NaturelNonNul**) **:** Ensemble<Direction>

⌊**précondition(s)** case≤largeur(laby)*hauteur(laby)

— **fonction** caseDestination (laby : Labyrinthe, case : **NaturelNonNul**, dir : Direction) **: NaturelNonNul**

⌊**précondition(s)** case≤largeur(laby)*hauteur(laby) et estPresent(directionsPossibles(laby,case),dir)

— **fonction** casesAdjacentes (laby : Labyrinthe, case : **NaturelNonNul) :** Ensemble<**NaturelNonNul**>

⌊**précondition(s)** case≤largeur(laby)*hauteur(laby)

2.3 Utilisation (3,5 points)

1. Proposez une fonction qui retourne l'ensemble des cases accessibles depuis une case donnée.
2. Proposez une fonction qui retourne l'ensemble des cases adjacentes non accessibles d'une case donnée.

Solution proposée :

fonction casesAccessibles (laby : Labyrinthe, case : **NaturelNonNul) :** Ensemble<**NaturelNonNul**>
⌊**précondition(s)** case≤largeur(laby)*hauteur(laby)
Déclaration res : Ensemble<**NaturelNonNul**>
debut
res ← ensemble()
pour chaque d **de** directionsPossibles(laby,case)
ajouter(res,caseDestination(laby,case,d))
finpour
retourner res
fin
fonction casesNonAccessibles (laby : Labyrinthe, case : **NaturelNonNul) :** Ensemble<**NaturelNonNul**>
⌊**précondition(s)** case≤largeur(laby)*hauteur(laby)
debut
retourner soustraction(casesAdjacentes(laby,case),casesAccessibles(laby,case))
fin

2.4 Conception détaillée (8 points)

On se propose de concevoir le TAD Labyrinthe de la façon suivante :

Type Labyrinthe **= Structure**
largeur : **NaturelNonNul**
hauteur : **NaturelNonNul**
entree : **NaturelNonNul**
sortie : **NaturelNonNul**
casesAccessibles : Dictionnaire<**NaturelNonNul**,Ensemble<**NaturelNonNul**>>
finstructure

2.4.1 Casser un mur (2 points)

Proposez une procédure `casserMur` qui permet de casser un mur entre deux cases adjacentes non accessibles.

Solution proposée :

procédure casserMur **(E/S** laby : Labyrinthe, **E** c1,c2 : **NaturelNonNul)**
⌊**précondition(s)** c1≤largeur(laby)*hauteur(laby) et c3≤largeur(laby)*hauteur(laby) et estPresent(casesNonAccessibles(laby,c1),c2)
Déclaration temp : Ensemble<**NaturelNonNul**>
debut
temp ← obtenirValeur(laby.casesAccessibles,c1)
ajouter(temp,c2)
ajouter(laby.casesAccessibles,c1,temp)
temp ← obtenirValeur(laby.casesAccessibles,c2)
ajouter(temp,c1)
ajouter(laby.casesAccessibles,c2,temp)
fin

2.4.2 Labyrinthe totalement fermé (2 points)

Proposez l'algorithme de la fonction suivante qui retourne un labyrinthe totalement fermé (toutes les cases sont entourés de murs) tel que la case d'entrée a pour valeur 1 et la case de sortie $largeur * hauteur$:

— **fonction** labyrintheFerme **(**largeur, hauteur : **NaturelNonNul) :** Labyrinthe

Solution proposée :

fonction labyrintheFerme **(**largeur, hauteur : **NaturelNonNul) :** Labyrinthe

Déclaration res : Labyrinthe

debut

```
res.largeur ← largeur
res.hauteur ← largeur
res.entree ← 1
res.sortie ← largeur*hauteur
res.casesAccessibles ← dictionnaire()
pour i ←1 à largeur*hauteur faire
    ajouter(res.casesAccessibles,i,ensemble())
finpour
```

fin

2.4.3 Création des chemins (4 points)

Nous allons ici utiliser une exploration exhaustive pour créer les chemins d'un labyrinthe que l'ont sait au départ totalement fermé.

L'algorithme est le suivant : depuis une case donnée c, si toutes les cases du labyrinthe n'ont pas été visitée, il faut tester aléatoirement toutes les cases adjacentes non accessibles c' de c, n'ayant pas été visitées en :

— cassant le mur entre c et c',
— indiquant que c' a été visitée,
— explorant exhaustivement le labyrinthe depuis c'.

À la première exécution de l'algorithme, les paramètres effectifs sont :

— un labyrinthe totalement fermée,
— la case d'entrée,
— un ensemble de cases qui représente les cases déjà visitées.

La figure 3 présente deux exemples d'exécution de cet algorithme à partir de labyrinthe de largeur 6 et hauteur 5 affiché en mode texte, tel que la case d'entrée est représentée par un 'E' et la case de sortie par un 'S'. Sur le labyrinthe de gauche l'ensemble de cases donné comme paramètre effectif est vide, alors que pour celui de droite cet ensemble contient la case d'entrée.

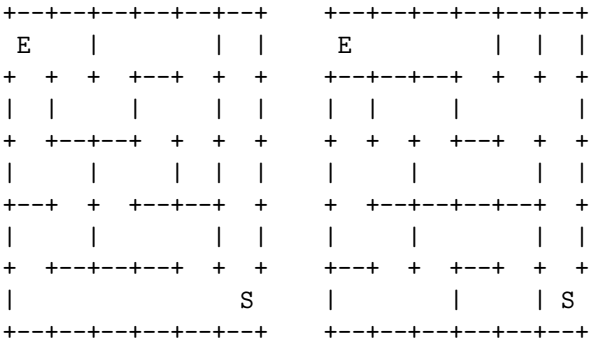

FIGURE 3 – Deux exemples de création de chemins

1. Quelle différence de comportement a cet algorithme lorsque l'ensemble de cases donné comme paramètre effectif est vide ou lorsqu'il contient la case d'entrée ? (0,5 point)
2. Proposez l'algorithme de la procédure suivante (3,5 points) :
   — **procédure** creerChemins **(E/S** l : Labyrinthe, **E/S** casesVisites : Ensemble<**NaturelNonNul**>, **E** c : **NaturelNonNul)**

Solution proposée :

Si l'ensemble ne contient pas la case d'entrée au lancement de l'algorithme, alors il y aura deux directions possibles depuis l'entrée

procédure creerChemins **(E/S** l : Labyrinthe, **E/S** casesDejaVisites : Ensemble<**NaturelNonNul**>, **E** c : **NaturelNonNul)**

Déclaration cases : Liste<**NaturelNonNul**>

```
debut
   si cardinalite(casesVisites<largeur(l)*hauteur(l)) alors
      cases ← listeMelangee(casesNonAccessibles(l,c))
      pour chaque c' de cases
         si non estPresent(casesDejaVisites,c') alors
            casserMur(l,c,c')
            ajouter(casesDejaVisites,c')
            creerChemins(l,casesDejaVisites,c')
         finsi
      finpour
   finsi
fin
```

Annexe

Voici la conception préliminaire de quelques TAD.

Liste

- **fonction** liste **()** : Liste
- **fonction** estVide **(**uneListe : Liste**)** : **Booleen**
- **procédure** insérer **(E/S** uneListe : Liste, **E** position : **Naturel**, element : Element**)**

  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe) + 1$
- **procédure** supprimer **(E/S** uneListe : Liste, **E** position : **Naturel)**

  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** obtenirElement **(**uneListe : Liste, position : **Naturel)** : Element

  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** longueur **(**uneListe : Liste**)** : **Naturel**

Ensemble

- **fonction** ensemble **()** : Ensemble
- **procédure** ajouter **(E/S** unEnsemble : Ensemble, **E** element : Element**)**
- **procédure** retirer **(E/S** unEnsemble : Ensemble, **E** element : Element**)**
- **fonction** estPrésent **(**unEnsemble : Ensemble, element : Element**)** : **Booleen**
- **fonction** cardinalite **(**unEnsemble : Ensemble**)** : **Naturel**
- **fonction** union **(**e1, e2 : Ensemble**)** : Ensemble
- **fonction** intersection **(**e1, e2 : Ensemble**)** : Ensemble
- **fonction** soustraction **(**e1, e2 : Ensemble**)** : Ensemble

Dictionnaire

- **fonction** dictionnaire **()** : Dictionnaire
- **procédure** ajouter **(E/S** unDictionnaire : Dictionnaire, **E** clef : Clef, element : Valeur**)**
- **procédure** retirer **(E/S** unDictionnaire : Dictionnaire, **E** clef : Clef**)**
- **fonction** estPresent **(**unDictionnaire : Dictionnaire, clef : Clef**)** : **Booleen**
- **fonction** obtenirValeur **(**unDictionnaire : Dictionnaire, clef : Clef**)** : Valeur

  ⌊**précondition(s)** estPresent(unDictionnaire, clef)
- **fonction** obtenirClefs **(**unDictionnaire : Dictionnaire**)** : Liste<Clef>
- **fonction** obtenirValeurs **(**unDictionnaire : Dictionnaire**)** : Liste<Valeur>