Algorithmique et Base de la programmation

Durée : *3h00*
Documents autorisés : **AUCUN**

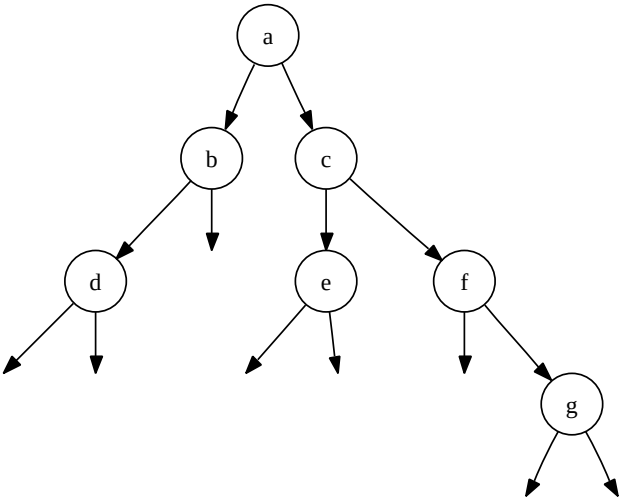
Remarques :
- Veuillez lire attentivement les questions avant de répondre.
- Le barème donné est un barème indicatif qui pourra évoluer lors de la correction.
- Rendez une copie propre.

1 Le nombre de Strahler (3,5 points)

Le nombre de Strahler d'un arbre binaire A est défini par :

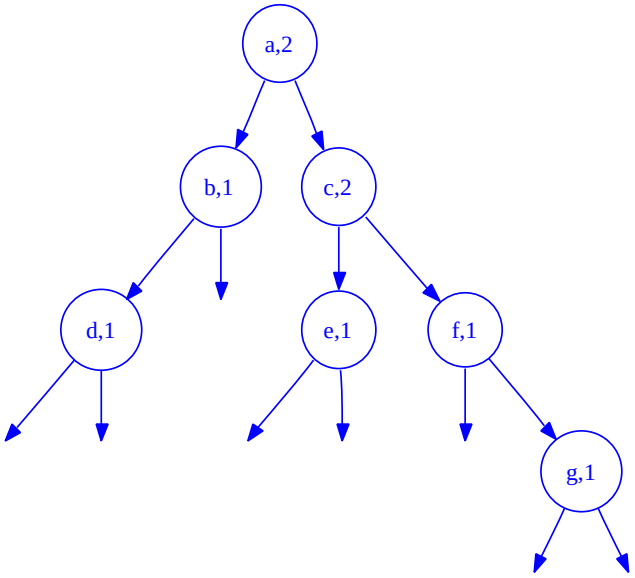
$$\Phi(A) = \begin{cases} 0 & \text{si l'arbre } A \text{ est vide} \\ max(\Phi(A_g), \Phi(A_d)) & \text{si } A \text{ est non vide et } \Phi(A_g) \neq \Phi(A_d) \\ 1 + \Phi(A_d) & \text{si } A \text{ est non vide et } \Phi(A_g) = \Phi(A_d) \end{cases}$$

1. Donnez les valeurs de Φ pour chaque nœud de l'arbre suivant :



2. Donnez un algorithme d'une fonction permettant de calculer le nombre de Strahler d'un arbre binaire.

Solution proposée:



fonction nbDeStralher (a : ArbreBinaire) **: Naturel**
  Déclaration nbAg, nbAd : **Naturel**
debut
  si estVide(a) **alors**
    retourner 0
  sinon
    nbAg ← nbDeStralher(obtenirFilsGauche(a))
    nbAd ← nbDeStralher(obtenirFilsDroit(a))
    si nbAg≠nbAd **alors**
      retourner max(nbAg,nbAd)
    sinon
      retourner 1+nbAd
    finsi
  finsi
fin

2 La SDD ListeDoublementChainee (4 points)

Nous avons vu en cours que la Structure Dynamique de Données (SDD) ListeDoublementChainee (LDC) est conçu de la manière suivante :

Type LDC = ^Noeud
Type Noeud = **Structure**
  element : **Entier**
  lSuivante : LDC
  lPrecedente : LDC
finstructure

On peut l'utiliser à l'aide des fonctions et procédures suivantes :
- **fonction** lDC () **:** LDC
- **fonction** estVide (l : LDC) **: Booleen**
- **procédure** inserer (**E/S** l : LDC, **E** element : Entier)
- **fonction** obtenirElement (l : LDC) **:** Entier

⌊**précondition(s)** $non(estVide(l))$

— **fonction** obtenirListeSuivante (l : LDC) **:** LDC

⌊**précondition(s)** $non(estVide(l))$

— **fonction** obtenirListePrecedente (l : LDC) **:** LDC

⌊**précondition(s)** $non(estVide(l))$

— **procédure** fixerElement (**E** l : LDC, e : **Entier**)

⌊**précondition(s)** $non(estVide(l))$

— **procédure** fixerListeSuivante (**E/S** l : LDC, l' : LDC)

⌊**précondition(s)** $non(estVide(l))$

— **procédure** fixerListePrecedente (**E/S** l : LDC, l' : LDC)

⌊**précondition(s)** $non(estVide(l))$

— **procédure** supprimerNoeud (**E/S** l : LDC)

⌊**précondition(s)** $non\ estVide(l)$

— **procédure** supprimer (**E/S** l : LDC)

Questions

Donnez l'algorithme de :

1. la procédure `inserer` qui insère l'entier à la position donnée (2 points)
2. la procédure `supprimerNoeud` (2 points)

Solution proposée:

```
procédure inserer (E/S l : LDC, E e : Element)
   Déclaration  temp : LDC
debut
   allouer(temp)
   fixerElement(temp,e)
   si estVide(l) alors
      l ← temp
      fixerListeSuivante(l,lDC(l))
      fixerListPrecedente(l,lDC(l))
   sinon
      fixerListeSuivante(temp,l)
      fixerListePrecedente(temp,obtenirListePrecedente(l))
      fixerListePrecedente(l,temp)
      l ← temp
      si non estVide(obtenirListePrecedente(l)) alors
         temp ← obtenirListePrecedente(l)
         fixerListeSuivante(temp,l)
      finsi
   finsi
fin
procédure supprimerNoeud (E/S l : LDC)
   ⌊précondition(s)  non estVide(l)
```

```
   Déclaration  g,d,temp : LDC
debut
   g ← obtenirListePrecedente(l)
   d ← obtenirListePrecedente(l)
   desallouer(l)
   si non estVide(d) alors
      l ← d
      fixerListePrecedente(l,g)
      si non estVide(g) alors
         fixerListeSuivante(g,l)
      finsi
   sinon
      si non estVide(g) alors
         l ← g
         fixerListeSuivante(l,d)
      finsi
   finsi
fin
```

3 Une liste ordonnée d'entiers (4,5 points)

On se propose de concevoir la collection ListeOrdonneeDEntiers (LODE) à l'aide de la SDD ListeChainee :

```
Type ListeOrdonneeDEntiers = Structure
   entiers : ListeChainee<Entier>
   nbEntiers : Naturel
finstructure
```

Questions

1. Donnez les signatures des fonctions et procédures permettant d'utiliser la SDD `ListeChainee` (0,5 points) ;
2. Proposez l'algorithme de `inserer` qui insère un entier dans une LODE. L'insertion de l'entier dans la liste chaînée devra être itérative (2 points) ;
3. Proposez l'algorithme de `supprimer` qui supprime la première occurrence d'un entier dans une LODE. La suppression dans la liste chaînée devra être récursive (2 points).

Solution proposée:

```
procédure insererDansListeChainee (E/S l : ListeChainee <Entier> ; E element : Entier)
   Déclaration  parcours, nouveau, temporaire : ListeChainee<Entier>
debut
   si estVide(l) alors
      l ← ajouter(l,element)
   sinon
      si obtenirElement(l) > element alors
         ajouter(l,element)
```

```
        sinon
            g ← l
            d ← obtenirListeSuivante(g)
            tant que non estVide(d) et obtenirElement(d) < element faire
                g ← d
                d ← obtenirListeSuivante(g)
            fintantque
            ajouter(d,element)
            fixerListeSuivante(g,d)
        finsi
    finsi
fin
procédure inserer (E/S l : ListeOrdonneeDEntiers ;E element : Entier)
    Déclaration  temp : ListeChainee<Entier>
debut
    temp ← obtenirEntiers(l)
    insererDansListeChainee(temp, element)
    fixerEntiers(l, temp)
    fixerNbEntiers (l, longueur(l) + 1)
fin
procédure supprimerDansListeChainee (E/S l : ListeChainee, E e : Entier, S suppressionEffective :
Booleen)
    Déclaration  temp : ListeChainee  <Entier>
debut
    si non estVide(l) alors
        si obtenirElement(l) = e alors
            supprimerTete(l)
            suppressionEffective ← VRAI
        sinon
            temp ← obtenirListeSuivante(l,e)
            supprimerDansListeChainee(temp,e)
            fixerListeSuivante(l, temp)
        finsi
    sinon
        suppressionEffective ← FAUX
    finsi
fin
procédure supprimer (E/S l : ListeOrdonneeDEntiers ;E element : Entier)
    ⌊précondition(s)  estPresent(l, e) = VRAI
    Déclaration  temp : ListeChainee  <Entier>
debut
    temp ← obtenirEntiers(l)
    supprimerDansListeChainee(temp,element)
    fixerEntiers(l, temp)
    fixerNbEntiers(l, longueur(l) - 1)
fin
```

4 Crazy Circus (8 points)

Le but ici est de concevoir une version informatique du jeu de société « Crazy circus ».

Règles du jeu [1]

Dans Crazy Circus chaque joueur incarne un dompteur s'efforçant de donner les (bons) ordres aux animaux, plus vite que les autres. Au début de chaque tour, une carte objectif est tirée au sort (cf. la carte présentée par la figure 1(b)). Elle montre la position que les animaux doivent prendre sur les deux podiums (le podium rouge à gauche, et bleu à droite). Les animaux en bois, sur le plateau de jeu, indiquent la position de départ (cf. la figure 1(a)).

(a) Position de départ (b) Objectif

FIGURE 1 – Le jeu Crary Circus

Pour passer de la position de départ à celle d'arrivée, les dompteurs doivent trouver la bonne séquence d'ordres, parmi cinq possibles : SO, MA, NI, KI et LO. Chaque ordre correspond à un déplacement, comme indiqué dans le tableau 1.

4.1 Le TAD CrazyCircus (1,5 points)

Soit les types `CouleurPodium`, `Animal` et `Ordre` suivants :
— **Type** CouleurPodium = {Rouge, Bleu}
— **Type** Animal = {Lion, Elephant, Ours}
— **Type** Ordre = {SO, MA, NI, KI, LO}
Proposez le TAD `CrazyCircus` sachant que l'on doit pouvoir :
— obtenir un `CrazyCircus` avec des positions pour les trois animaux aléatoires ;
— donner un ordre (lorsqu'un ordre n'est pas applicable, l'état d'un `CrazyCircus` est inchangé) ;
— obtenir les animaux se trouvant sur un podium.

Solution proposée:

Nom:	CrazyCircus	
Utilise:	CouleurPodium, Animal, Ordre, Liste	
Opérations:	crazyCircus:	→ CrazyCircus
	donnerOrdre:	CrazyCircus × Ordre → CrazyCircus
	animaux:	CrazyCircus × CouleurPodium → Liste<Animal>

1. Issues de http ://crazycircus.free.fr/

	SO : l'animal au sommet du podium rouge échange sa place avec l'animal au sommet du podium bleu.		MA : l'animal en bas du podium rouge va en haut de ce même podium.
	NI : la même chose que MA sur l'autre podium : l'animal en bas du podium bleu va en haut de ce même podium.		KI : l'animal en haut du podium bleu va au sommet du podium rouge.
	LO : la même chose que KI sur l'autre pordium : l'animal en haut du podium rouge va au sommet du podium bleu.		

TABLE 1 – Signification des ordres

4.2 Conception préliminaires (0,5 points)

Donnez les signatures des fonctions et procédures du TAD `CrazyCircus`.

Solution proposée:

- **fonction** crazyCircus **() :** CrazyCircus
- **procédure** donnerOrdre **(E/S** cc : CrazyCircus, **E** o : Ordre**)**
- **fonction** animaux **(**cc : CrazyCircus, cp : CouleurPodium**) :** Liste<Animal>

4.3 Utilisation (6 points)

1. Proposez l'algorithme de la fonction suivante qui donne l'état d'un `CrazyCircus` après l'application de plusieurs ordres (1 point) :
   - **fonction** appliquerOrdres **(**cc : CrazyCircus, lo : Liste<Ordre>**) :** CrazyCircus
2. Proposez l'algorithme de la fonction suivante qui permet de savoir si une suite d'ordres résout un `CrazyCircus` (0,5 point) :
   - **fonction** resout **(**positionDepart, objectif : CrazyCircus, lo : Liste<Ordre>**) : Booleen**
3. Proposez l'algorithme de la procédure qui essaye de résoudre le problème du `CrazyCircus` de manière optimale (avec un nombre d'ordre minimal). Cette procédure prend en entrée la position de départ, l'objectif et le nombre d'ordres maximal. En sortie nous avons un booléen qui indique si le problème a pu être résolu (avec le nombre d'ordres donné en entrée) et une liste d'ordres qui résout le problème si c'est possible. Pour résoudre ce problème, vous pouvez vous inspirer du parcours en largeur d'un graphe avec l'utilisation d'une file de liste d'ordres (4,5 points).

Solution proposée:

fonction appliquerOrdres **(**cc : CrazyCircus, lo : Liste<Ordre>**) :** CrazyCircus

```
   Déclaration  o : Ordre
debut
   pour chaque o de lo
      donnerOrdre(cc,o)
   finpour
   retourner cc
fin

fonction resout (positionDepart, objectif : CrazyCircus, lo : Liste<Ordre>) : Booleen
debut
   retourner appliquerOrdres(positionDepart,lo)=objectif
fin

procédure enfilerListesAvecUnOrdreDePlus (E/S f : File<Liste<Ordre>>, E lRef : Liste<Ordre>)

   Déclaration  o : Ordre
debut
   pour o ←SO à LO faire
      temp ← lRef
      inserer(temp,longueur(temp)+1,o)
      enfiler(f,temp)
   finpour
fin
procédure resoudre (E positionDepart, objectif : CrazyCircus, nbOrdresMax : NaturelNonNul,
S solutionTrouvee : Booleen, solution : Liste<Ordre>)
   Déclaration  f : File<Liste<Ordre>>
debut
   si positionDepart = objectif alors
      solutionTrouvee ← VRAI
      solutionTrouvee ← liste()
   sinon
      f ← file()
      solutionTrouvee ← FAUX
      enfilerListesAvecUnOrdreDePlus(f,liste())
      tant que non solutionTrouvee et non estVide(f) faire
         temp ← obtenirElement(f)
         defiler(f)
         si resout(positionDepart,objectif,temp) alors
            solution ← temp
            solutionTrouvee ← VRAI
         sinon
            si longueur(temp)<nbOrdresMax alors
               enfilerListesAvecUnOrdreDePlus(f,temp)
            finsi
         finsi
      fintantque
   finsi
fin
```