EISTI Ecole Internationale des Sciences du Traitement de l'Information	**ING1 : EXAMEN D'ALGORITHMIQUE II : 2009-2010**
Rédigé par : l'équipe pédagogique du module Algorithmique II	Ref : *ING1-ALG-CON-EXAMEN*
A l'intention de : Etudiants des ING1	Créé le : 01/05/2012

1. Exercice : Pile (4pts)

Dans cet exercice, on ne considère que des piles qui contiennent des entiers.

1.1 Questions

1. Écrire un algorithme pour déplacer les entiers d'une pile dans une autre de telle sorte que dans la pile résultat les nombres pairs soient au dessous des nombres impairs. L'ordre initial doit être respecté
2. Écrire un algorithme pour copier dans la pile résultat les nombres pairs contenus dans la pile initiale. Le contenu de la pile initiale après exécution de l'algorithme ne doit pas avoir été modifié. Les nombres pairs dans la pile résultat doivent être dans l'ordre où ils apparaissent dans la pile initiale.

1.2 Rappels

Spécifications du type Pile

- `procédure creerPile(E/S Pile : Pile de Element)`
- `procédure empiler(E/S Pile: Pile de Element, elt : Element)`
- `procédure depiler(E/S Pile: Pile de Element)`
- `fonction estVide(E pile: Pile de Element): Booleen`
- `fonction sommet(E pile: Pile de Element): Element`

2. Graphes : algorithme de Marimont (7pts)

2.1 Présentation de l'algorithme

Dans de nombreuses applications, on souhaite savoir si un graphe possède ou non des circuits. L'algorithme de Marimont permet de résoudre ce problème avec une complexité "raisonnable".

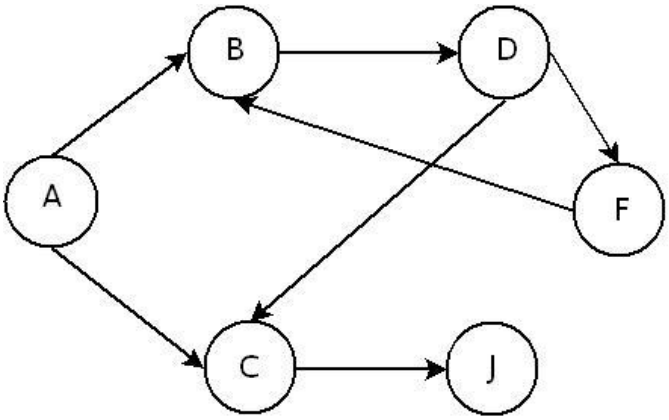
Il est basé sur la propriété qui suit : Soit G un graphe orienté alors les deux propositions suivantes sont équivalentes :

1. G est sans circuit
2. G et tous ses sous-graphes possèdent au moins un **point d'entrée (sommet sans prédécesseur)** et au moins un **point de sortie (sommet sans successeur)**

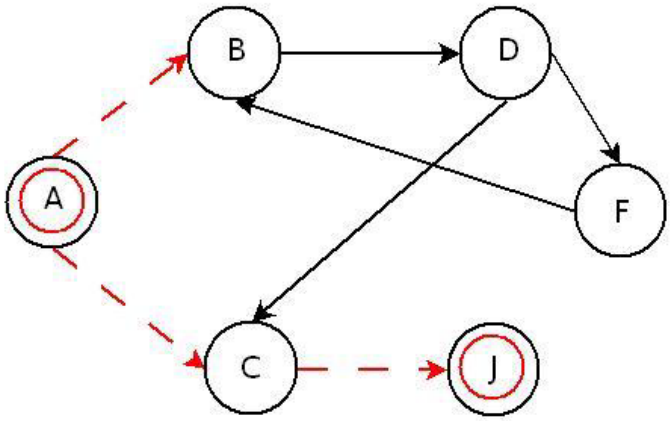
Un exemple d'écriture de l'algorithme de Marimont dans notre contexte est donc le suivant. Soit H le **sous-graphe de G qui contient tous ses sommets non encore marqués**, E l'ensemble des points d'entrée de H, et S l'ensemble des points de sortie de H.

```
H ← G // à l'initialisation, aucun sommet de G n'est marqué
E ← Ensemble des points d'entrée de H
S ← Ensemble des points de sortie de H
TANTQUE (H a des sommets ET E et S ne sont pas vides) FAIRE
   On marque tous les sommets appartenant à E
   On marque tous les sommets appartenant à S
   H ← Sous-graphe de H qui contient tous les sommets non marqués
   E ← Ensemble des points d'entrée de ce nouveau H
   S ← Ensemble des points de sortie de ce nouveau H
FINTANTQUE
L'algorithme renvoie vrai si et seulement si le dernier sous graphe H n'a
plus de sommets
```

Voici un exemple d'application :

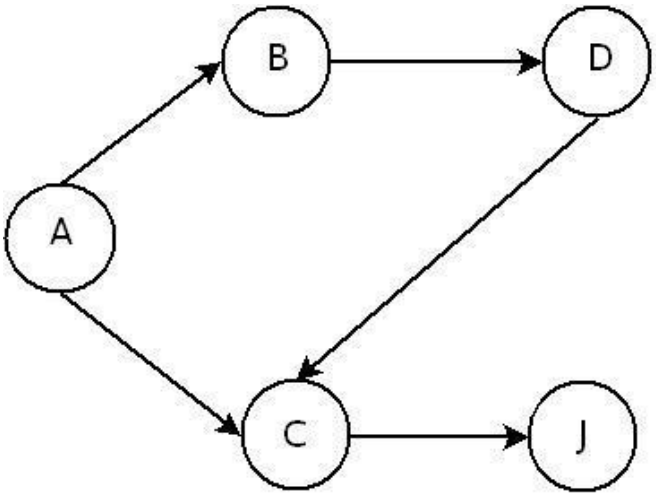


H <-- G
E <-- {A}
S <-- {J}

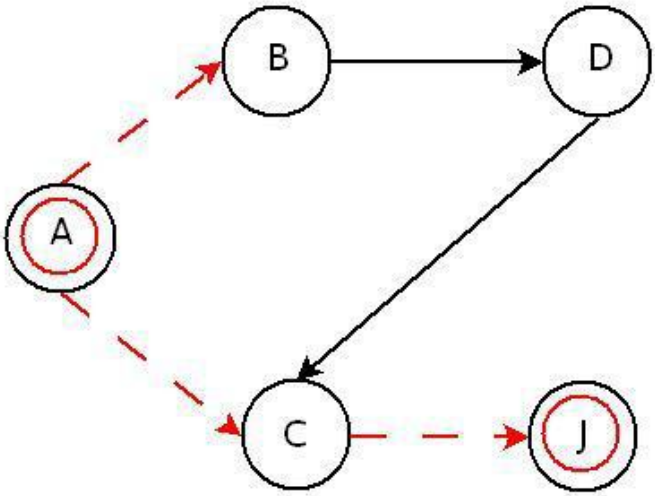


H <-- H \ {A,J}
E <-- {}
S <-- {C}
Fin de l'itération (E est vide)
H a encore des sommets
Il y a donc au moins un circuit

Autre exemple d'application :



H <-- G
E <-- {A}
S <-- {J}



H <-- H \ {A,J}
E <-- {B}
S <-- {C}

2.2 Questions

1. Écrire la méthode qui construit le sous-graphe constitué de tous les sommets non marqués du graphe sur lequel on a appelé l'opération.
2. Écrire la méthode qui construit l'ensemble des points d'entrée du graphe sur lequel on a appelé l'opération.
3. Écrire la méthode qui construit l'ensemble des points de sortie du graphe sur lequel on a appelé l'opération.
4. Écrire la méthode qui exécute l'algorithme de Marimont. Le résultat est un booléen qui vaut vrai si le graphe sur lequel on a appelé la méthode contient au moins un circuit et faux sinon.

2.3 Spécifications du type Graphe

Ce type utilise les types Element et Arete (donc AreteValuee)

Méthodes de construction et d'initialisation

- fonction grapheVide() : Graphe

Méthodes de mise à jour

- fonction ajouterSommet(gr : Graphe, e : Element) : Graphe
- fonction ajouterArete(gr : Graphe, a : Arete) : Graphe
- fonction supprimerSommet(gr : Graphe, e : Element) : Graphe
- fonction supprimerArete(gr : Graphe, a : Arete) : Graphe

Méthodes d'accès

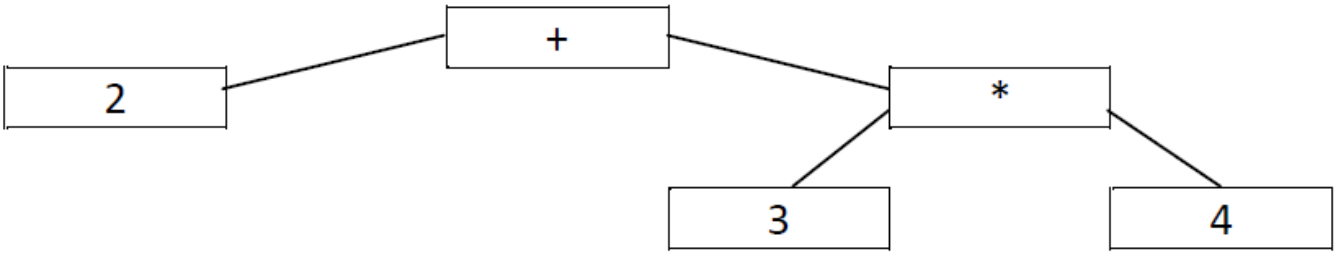
- fonction recNbSommets(gr : Graphe) : Entier
- fonction recNbAretes(gr : Graphe) : Entier
- fonction recSommets(gr : Graphe) : Vecteur
- fonction recAretes(gr : Graphe) : Vecteur
- fonction existeArete(gr : Graphe, o : Element, d : Element, S ar : Arete) : Booleen
- fonction recSuccesseurs(gr : Graphe, e : Element) : Vecteur
- fonction recPredecesseurs(gr : Graphe, e : Element) : Vecteur

3. Evaluation d'une expression numérique arborescente

On considère une expression numérique simple. On suppose que cette expression n'est formée que de constantes numériques et des quatre opérateurs +, -, *, /. Il s'agit dans ce problème de transformer cette expression sous forme d'un arbre (voir schéma ci-dessous) puis de l'évaluer.

Exemple : expression 2+3 * 4 se transforme comme suit :

Schéma arborescent de l'expression 2 + 3 * 4



On modélise ce problème à l'aide d'un nouveau type nommé **ExpSim_ExpArb**.

On définit les méthodes de base comme suit :

1. Une méthode nommée **creerExp** qui crée un nouvel objet en recevant en paramètre l'expression simple sous forme d'un vecteur de Termes (indicé de 1 à ???). L'expression doit être syntaxiquement correcte.
2. Une méthode nommée **recNbTermes** qui permet de récupérer le nombre de termes de l'expression simple
3. Une méthode nommée **recTerme** qui permet de récupérer le n^{ième} de l'expression simple.
4. Une méthode nommée **insererTermes** qui reçoit en paramètres un entier r, un terme de type opérateur et un terme de type opérande. Cette méthode insère dans l'expression simple l'opérateur et l'opérande juste après le n^{ième} terme de l'expression simple.

3.1 Questions

1. Donner la signature de ces opérations avec les pré-conditions et les axiomes.
2. Ecrire l'opération d'extension nommée **evaluerExp** qui permet d'évaluer l'expression simple en utilisant sa représentation sous forme d'arbre. Dans cette question, on suppose que l'opération nommée **recExpArb** qui permet de récupérer l'expression sous forme d'arbre est déjà écrite.
3. Calculer la complexité de la méthode précédente.
4. On suppose maintenant qu'en plus des constantes, on peut avoir des variables dans l'expression. Proposer une autre définition des opérations de base qui intègre cette nouvelle possibilité.

3.2 Spécifications du type Terme

- `fonction creerTermeOperande(operande : Reel) : Terme`
- `fonction creerTermeOperateur(operateur : Operateur) : Terme`
- `fonction estOperateur() : Booleen`
- `fonction  recOperateur() : Chaine`
- `fonction  recOperande() : Reel`

3.3 Spécifications du type ArbreKAire

- `fonction arbreVide() : ArbreKAire`
- `fonction creerArbre(e : Element,ar : Entier) : ArbreKAire`
- `fonction modifierRacine(ES ar : ArbreKAire, e: Element) : ArbreKAire`
- `fonction affEnfant(ES arp :ArbreKAire,i:Entier,arf: ArbreKAire): ArbreKAire`
- `fonction supprimerFeuille(ES ar :ArbreKAire, i : Entier) :ArbreKAire`
- `fonction estVide (ar : ArbreKaire) : Booleen`
- `fonction recRacine(ar : ArbreKaire) : Element`
- `fonction existeParent(ar : ArbreKaire) : Booleen`
- `fonction recEnfant(ar : ArbreKaire,Entier) : ArbreKAire`
- `fonction recParent(ar : ArbreKaire) : ArbreKAire`
- `fonction recArite(ar : ArbreKaire) : Entier`

4. Annexe

4.1 Spécification du type Vecteur

- `fonction creerVecteur(bi : Entier, bs : Entier) : Vecteur`
- `fonction ieme(v : Vecteur, i : Entier) : Element`
- `fonction modifieme(E/S v : Vecteur, i : Entier, e : Element) : Vecteur`
- `fonction estInitialise(v : Vecteur, i : Entier) : Booleen`
- `fonction borneInf(v : Vecteur) : Entier`
- `fonction borneSup(v : Vecteur) : Entier`