# Algorithmique avancée et Programmation C

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.

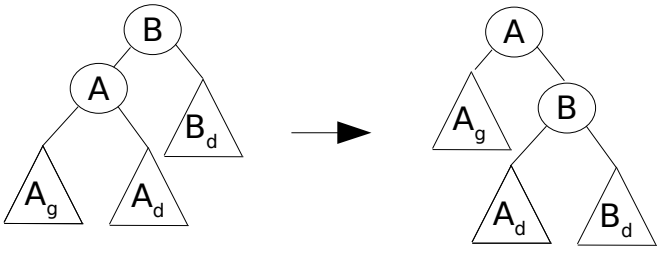
## 1 Arbre binaire : simple et double rotations (7 points)

1. Rappelez à l'aide de deux schémas par algorithme (avant et après l'exécution de l'algorithme) ce que réalisent les algorithmes de simple rotation à gauche, simple rotation à droite, de double rotation à gauche et de double rotation à droite. (2 points)
2. Donnez les signatures de ses quatre procédures et l'algorithme des deux procédures de simple et double rotation à droite. (4 points)
3. Démontrez que l'algorithme de simple rotation à droite utilisé avec un arbre binaire de recherche conserve ses caractéristiques. (1 point)

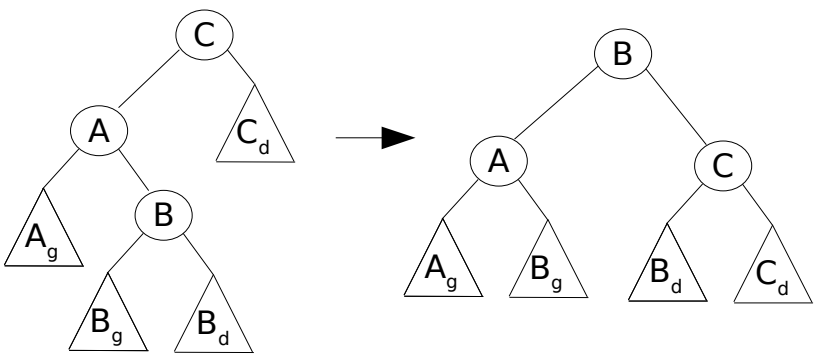
**Solution proposée:**

1. Voir le cours (exemples de ce qui est demandé)

Simple rotation à droite




Double rotation à droite



2.

— **procédure** simpleRotationADroite **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsGauche(a))

— **procédure** simpleRotationAGauche **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsDroit(a))

— **procédure** doubleRotationADroite **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsGauche(a))
et non estVide(obtenirFilsDroit(obtenirFilsGauche(a)))

— **procédure** doubleRotationAGauche **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsDroit(a))
et non estVide(obtenirFilsGauche(obtenirFilsDroit(a)))

**procédure** simpleRotationADroite **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsGauche(a))

**Déclaration** fG : ArbreBinaire

**debut**
fG ← obtenirFilsGauche(a)
fixerFilsGauche(a,obtenirFilsDroit(fg))
fixerFilsDroit(fg,a)
a ← fG
**fin**

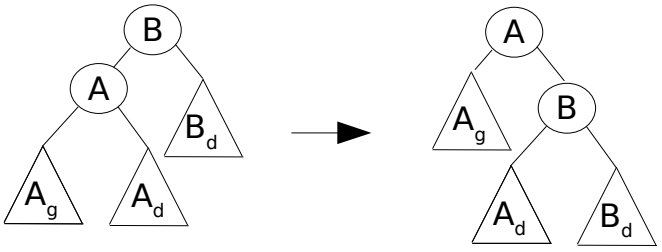
**procédure** doubleRotationADroite **(E/S** a : ArbreBinaire**)**

⌊**précondition(s)** non estVide(a) et non estVide(obtenirFilsGauche(a))
et non estVide(obtenirFilsDroit(obtenirFilsGauche(a)))

**Déclaration** fG : ArbreBinaire

**debut**
fG ← obtenirFilsGauche(a)
simpleRotationAGauche(fG)
fixerFilsGauche(a,fG)
simpleRotationADroite(a)
**fin**

3. Si l'arbre de gauche de la figure suivante est un ABR




alors on a :

(a) $A < B$

(b) $\forall x \in A_g, x < A, x < B$

(c) $\forall x \in A_d, x > A, x < B$

(d) $\forall x \in B_d, x > A, x > B$

Vérifions maintenant que ces propriétés nous permettent d'affirmer que l'arbre de droite est un ABR. Dans cet arbre :

- $B$ est la racine du fils droit de $A$, or d'après 1, $A < B$, donc cette partie de l'arbre respecte les conditions d'un ABR.
- $A_g$ est fils gauche de de $A$, or d'après 2, $\forall x \in A_g, x < A$, donc cette partie de l'arbre respecte les conditions d'un ABR.
- $A_d$ est dans le fils droit de $A$, or d'après 2, $\forall x \in A_d, x > A$, donc cette partie de l'arbre respecte les conditions d'un ABR.
- $A_d$ est le fils gauche de $B$, or d'après 2, $\forall x \in A_d, x < B$, donc cette partie de l'arbre respecte les conditions d'un ABR.
- $B_d$ est dans le fils droit de $A$, or d'après 3, $\forall x \in B_d, x > A$, donc cette partie de l'arbre respecte les conditions d'un ABR.
- $B_d$ est le fils droit de $B$, or d'après 3, $\forall x \in B_d, x > B$, donc cette partie de l'arbre respecte les conditions d'un ABR.

Toutes les composantes de l'arbre de droite respectent bien les conditions d'un ABR, donc l'arbre de droite est un ABR.

L'algorithme de simple rotation à droite conserve donc bien les caractéristiques d'un ABR.

# 2 Graphes (13 points)

Bien que l'on ne vous demande pas d'analyse descendante dans les questions suivantes, vous veillerez à bien décomposer votre problème (chaque suite d'instructions formant « un tout » doit devenir une fonction ou une procédure).

## 2.1 Arcs d'un graphe non orienté (4 points)

En utilisant les fonctions et procédures proposées en annexe, donnez l'algorithme de la fonction suivante qui retourne tous les arcs (un arc est une liste composée de deux sommets) d'un graphe non orienté :

- **fonction** obtenirArcs (g : Graphe) **:** Liste<Liste<Sommet>>

**Solution proposée:**

**fonction** arc (s1, s2 : Sommet) **:** Liste<Sommet>
  **Déclaration** a : Liste<Sommet>
**debut**
  a ← liste()
  inserer(a,s1)
  inserer(a,s2)
  **retourner** a
**fin**

**fonction** arcsEgaux (a1,a2 : Liste<Sommet>) **: Booleen**
  ⌊**précondition(s)** $longueur(a1) = 2$ et $longueur(a2) = 2$
**debut**
  **retourner** (obtenirElement(a1,1)=obtenirElement(a2,1) et obtenirElement(a1,2)=obtenirElement(a2,2))
  ou (obtenirElement(a1,2)=obtenirElement(a2,1) et obtenirElement(a1,1)=obtenirElement(a2,2))
**fin**

**fonction** arcPresent (l : Liste<Liste<Sommet>>, a : Liste<Sommet>) **: Booleen**

⌊**précondition(s)** $\forall i \in 1..longueur(l), longueur(obtenirElement(l,i)) = 2$ et $longueur(a) = 2$

**Déclaration** trouve : **Booleen**
i : **NaturelNonNul**
temp : Liste<Sommet>

**debut**
trouve ← FAUX
i ← 1
**tant que** non trouve et i≤longueur(l) **faire**
temp ← obtenirElement(l,i)
**si** arcsEgaux(temp,a) **alors**
trouve ← VRAI
**sinon**
i ← i+1
**finsi**
**fintantque**
**retourner** trouve
**fin**

**fonction** obtenirArcs (g : Graphe) **:** Liste<Liste<Sommet>>

**Déclaration** res : Liste<Liste<Sommet>>
s1,s2 : Sommet
a : Liste<Sommet>

**debut**
res ← liste()
**pour chaque** s1 **de** obtenirSommets(g)
**pour chaque** s2 **de** obtenirSommetsAdjacents(g,s1)
a ← arc(s1,s2)
**si** non arcPresent(res,a) **alors**
inserer(res,a)
**finsi**
**finpour**
**finpour**
**retourner** res
**fin**

## 2.2 Dijkstra (9 points)

Nous avons vu dans le cours sur les graphes l'algorithme de Dijkstra suivant :

**procédure** dijkstra (**E** g : Graphe<Sommet,,**ReelPositif**>, s : Sommet,**S** arbreRecouvrant : Arbre<Sommet>, cout : Dictionnaire<Sommet,**ReelPositif**>)

⌊**précondition(s)** $sommetPresent(g, s)$

**Déclaration** l : Liste<Liste<Sommet>>
c : **ReelPositif**
sommetDeA, sommetAAjouter : Sommet

**debut**
arbreRecouvrant ← racine(s,liste())

```
cout ← dictionnaire()
ajouter(cout,s,0)
l ← arcsEntreArbreEtGraphe(arbreRecouvrant,g)
tant que non estVide(l) faire
   sommetLePlusProche(g,l,cout,sommetDeA,sommetAAjouter,c)
   ajouter(cout,
           sommetAAjouter,
           obtenirValeur(cout,sommetDeA)+c
   )
   ajouterCommeFils(arbreRecouvrant,sommetDeA,sommetAAjouter)
   l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)
fintantque
fin
```

Tel que :

— **fonction** arcsEntreArbreEtGraphe **(**a : Arbre<Sommet>, g : Graphe<Sommet,,**ReelPositif**>**)** **:** Liste<Liste<Sommet>>
retourne la liste des arcs (liste de deux sommets) dont le premier sommet appartient à *a* et le second sommet appartient à *g* et n'appartient pas à *a*

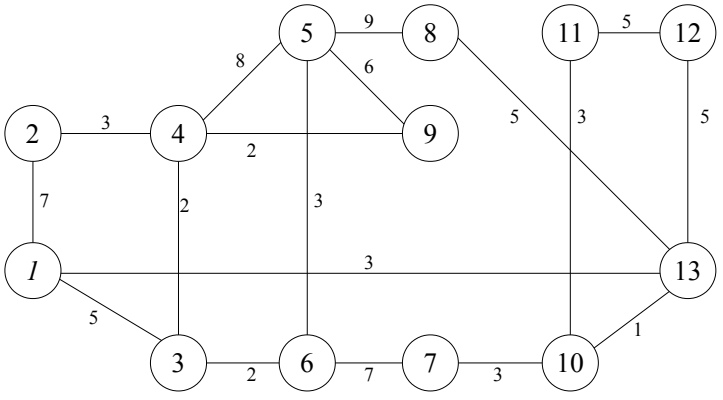
— **procédure** sommetLePlusProche **(E** g : Graphe<Sommet>, arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet, **ReelPositif**>,**S** sommeDeA, sommetAAjouter : Sommet, cout-Supplementaire : **ReelPositif)**

⌊**précondition(s)** $non(estVide(arcs))$
et $\forall i \in 1..longueur(arcs), longueur(obtenirElement(arcs, i)) = 2$

détermine, parmi les *arcs*, celui dont le sommet *sommetAAjouter* du graphe est le plus proche (au sens du dictionnaire de *cout*) des sommets de *a* (en identifiant *sommetDeA*)

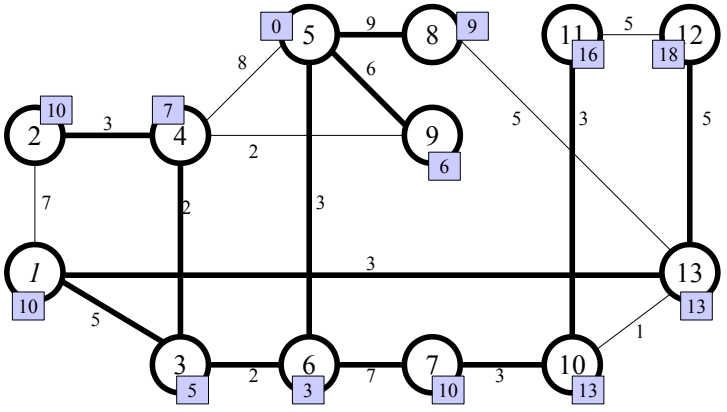
— **procédure** ajouterCommeFils **(E/S** a : Arbre<Sommet>, **E** sommetPere, sommetFils : Sommet**)**
ajoute un nouveau noeud, à l'arbre *a*, contenant *sommetFils* qui sera fils du noeud contenant *sommetPere*

1. Donnez l'arbre obtenu pour le graphe suivant sachant que le sommet de départ est le sommet 5. (2 points)




2. Donnez l'algorithme de la fonction *arcsEntreArbreEtGraphe*. (2 points)
3. Donnez l'algorithme de la procédure *sommetLePlusProche*. (2 points)
4. Donnez l'algorithme de la procédure *ajouterCommeFils*. (3 points)

**Solution proposée:**




**fonction** estPresent (a : Arbre<Sommet>, s : Sommet) **: Booleen**
**debut**
  **si** estVide(a) **alors**
    **retourner** FAUX
  **sinon**
    **si** obtenirElement(a)=s **alors**
      **retourner** VRAI
    **sinon**
      trouve ← FAUX
      i ← 1
      **tant que** i≤longueur(obtenirFils(a) et non trouve **faire**
        trouve ← estPresent(obtenirElement(obtenirFils(a),i))
        i ← i+1
      **fintantque**
      **retourner** trouve
    **finsi**
  **finsi**
**fin**
**fonction** arcsEntreArbreEtGraphe (a : Arbre<Sommet>, g : Graphe<Sommet,,**ReelPositif**>) : Liste<Liste<Sommet>>
**debut**
  res ← liste()
  **pour chaque** arc **de** obtenirArcs(g)
    **si** estPresent(a, obtenirElement(arc,1)) et non estPresent(a, obtenirElement(arc,2))
    ou estPresent(a, obtenirElement(arc,2)) et non estPresent(a, obtenirElement(arc,1)) **alors**
      inserer(res,arc)
    **finsi**
  **finpour**
  **retourner** res
**fin**

**procédure** sommetLePlusProche (**E** g : Graphe<Sommet>, arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet, **ReelPositif**>,**S** sommeDeA, sommetAAjouter : Sommet, coutSupplementaire : **ReelPositif**)

⌊**précondition(s)** $non(estVide(arcs))$
et $\forall i \in 1..longueur(arcs), longueur(obtenirElement(arcs, i)) = 2$

**Déclaration** arc : Liste<Sommet>
min : **ReelPositif**

```
debut
   arc ← obtenirElement(arcs,1)
   coutSupplementaire ← obtenirValeur(cout,obtenirElement(arc,1)) + obtenirValeur(g, obtenirElement(arc,1), obtenirElement(arc,2))
   pour chaque a de arcs
      temp ← obtenirValeur(cout,obtenirElement(a,1))+obtenirValeur(g,obtenirElement(a,1), obtenirElement(a,2))
      si temp<coutSupplementaire alors
         arc ← a
         coutSupplementaire ← temp
      finsi
   finpour
   sommeDeA ← obtenirElement(arc,1)
   sommetAAjouter ← obtenirElement(arc,2)
fin

procédure remplacer (E/S l : Liste, E i : NaturelNonNul, e : Element)
   ⌊précondition(s)  i≤longueur(l)
debut
   supprimer(l,i)
   inserer(l,i,e)
fin
procédure ajouterCommeFils (E/S a : Arbre<Sommet>, E sommetPere, sommetFils : Sommet)
debut
   si non estVide(a) alors
      si obtenirElement(a) = sommetPere alors
         fils ← obtenirFils(a)
         inserer(fils,1,racine(sommetFils,liste()))
         fixerFils(a, fils)
      sinon
         fils ← obtenirFils(a)
         pour i ←1 à longueur(fils) faire
            temp ← obtenirElement(fils,i)
            ajouterCommeFils(temp,sommetPere,sommetFils)
            remplacer(fils,i,temp)
         finpour
         fixerFils(a,fils)
      finsi
   finsi
fin
```

# Annexe

Voici la conception préliminaire de quelques TAD.

## Arbre

- **fonction** arbre () **:** Arbre
- **fonction** racine (e : Element, fils : Liste<Arbre>) **:** Arbre
- **fonction** estVide (unArbre : Arbre) **: Booleen**

— **fonction** obtenirElement (unArbre : Arbre) **:** Element
  ⌊**précondition(s)** non estVide(unArbre)
— **fonction** obtenirFils (unArbre : Arbre) **:** Liste<Arbre>
  ⌊**précondition(s)** non estVide(unArbre)
— **procédure** fixerFils (**E/S** unArbre : Arbre, **E** fils : Liste<Arbre>)
  ⌊**précondition(s)** non estVide(unArbre)

## ArbreBinaire

— **fonction** arbreBinaire () **:** ArbreBinaire
— **fonction** estVide (unArbre : ArbreBinaire) **: Booleen**
— **procédure** insérer (**E/S** unArbre : ArbreBinaire, **E** element : Element)
— **procédure** supprimer (**E/S** unArbre : ArbreBinaire, **E** element : Element)
— **fonction** estPrésent (unArbre : ArbreBinaire, element : Element) **: Booleen**
— **fonction** obtenirElement (unArbre : ArbreBinaire) **:** Element
  ⌊**précondition(s)** non estVide(unArbre)
— **fonction** obtenirFilsGauche (unArbre : ArbreBinaire) **:** ArbreBinaire
  ⌊**précondition(s)** non estVide(unArbre)
— **fonction** obtenirFilsDroit (unArbre : ArbreBinaire) **:** ArbreBinaire
  ⌊**précondition(s)** non estVide(unArbre)
— **procédure** fixerFilsGauche (**E/S** unArbre : ArbreBinaire, **E** fils : ArbreBinaire)
  ⌊**précondition(s)** non estVide(unArbre)
— **procédure** fixerFilsDroit (**E/S** unArbre : ArbreBinaire, **E** fils : ArbreBinaire)
  ⌊**précondition(s)** non estVide(unArbre)

## 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>

## Graphe

— **fonction** graphe () **:** Graphe
— **procédure** ajouterSommet (**E/S** g : Graphe, **E** s : Sommet)
  ⌊**précondition(s)** non sommetPresent(g,s)
— **procédure** ajouterArc (**E/S** g : Graphe, **E** s1, s2 : Sommet)
  ⌊**précondition(s)** sommetPresent(g,s1) et sommetPresent(g,s2)
  et non arcPresent(g,s1,s2)
— **fonction** sommetPresent (g : Graphe, s : Sommet) **: Booleen**

— **fonction** arcPresent (g : Graphe, s1, s2 : Sommet) **: Booleen**
— **procédure** supprimerSommet (**E/S** g : Graphe, **E** s : Sommet)
  ⌊**précondition(s)** sommetPresent(g,s)
— **procédure** supprimerArc (**E/S** g : Graphe, **E** s1, s2 : Sommet)
  ⌊**précondition(s)** arcPresent(g,s1,s2)
— **fonction** obtenirSommets (g : Graphe) **:** Liste<Sommet>
— **fonction** obtenirSommetsAdjacents (g : Graphe, s : Sommet) **:** Liste<Sommet>
  ⌊**précondition(s)** sommetPresent(g,s)
— **fonction** possedeEtiquette (g : Graphe, s : Sommet) **: Booleen**
  ⌊**précondition(s)** sommetPresent(g,s)
— **fonction** obtenirEtiquette (g : Graphe, s : Sommet) **:** Etiquette
  ⌊**précondition(s)** sommetPresent(g,s)
— **procédure** fixerEtiquette (**E/S** g : Graphe, **E** s : Sommet, e : Etiquette)
  ⌊**précondition(s)** sommetPresent(g,s)
— **fonction** possedeValeur (g : Graphe, s1, s2 : Sommet) **: Booleen**
  ⌊**précondition(s)** arcPresent(g,s1,s2)
— **fonction** obtenirValeur (g : Graphe, s1, s2 : Sommet) **:** Valeur
  ⌊**précondition(s)** arcPresent(g,s1, s2)
— **procédure** fixerValeur (**E/S** g : Graphe, **E** s1, s2 : Sommet, v : Valeur)
  ⌊**précondition(s)** arcPresent(g,s1,s2)

## 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**