POLYTECHNIQUE MONTRÉAL

Questionnaire examen final

INF2010

Sigle du cours

Q1	
Q2	
Q3	
Q4	
Q5	
Total	

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Sigle et titre du cours		*Groupe*	*Trimestre*
INF2010 – Structures de données et algorithmes		**Tous**	**20151**
Professeur		***Local***	***Téléphone***
Ettore Merlo, responsable / Tarek Ould Bachir, chargé		**-**	**-**
Jour	***Date***	***Durée***	***Heures***
Mercredi	**22 avril 2014**	**2h30**	**9h30-12h00**

Documentation	*Calculatrice*	
☒ Aucune	☐ Aucune	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**
☐ Toute	☐ Toutes	
☒ Voir directives particulières	☒ Non programmable	

Directives particulières

- Un cahier supplémentaire vous sera remis. Servez-vous de ce cahier comme brouillon. Toutes vos réponses doivent être faites sur le questionnaire. Le cahier supplémentaire n'est pas à remettre à la fin de l'examen.

Important

Cet examen contient **5** questions sur un total de **29** pages **(excluant cette page)**

La pondération de cet examen est de **40** **%**

Vous devez répondre sur : ☒ le questionnaire ☐ le cahier ☐ les deux

Vous devez remettre le questionnaire : ☒ oui ☐ non

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduite.

Question 1 : Files de priorité (30 points)

Pour cette question, vous devez vous référer au code Java de l'Annexe 1.

On y trouve deux classes implémentant une file de priorité décrite par l'interface :

```
public interface PriorityQueue<AnyType> {
    int size();
    void clear();
    boolean isEmpty( );
    boolean contains(AnyType x);
    AnyType peek() throws NoSuchElementException;
    AnyType remove() throws NoSuchElementException;
    boolean add(AnyType x, double priority);
    void updatePriority(AnyType x, double priority);
    AnyType getMax() throws NoSuchElementException;
}
```

La première classe s'appelle `HeapPriorityQueue`. Elle utilise un monceau tel que vu en classe. Un tableau contient tous les enregistrements. L'élément le plus prioritaire (plus petite valeur de priorité) se trouve au début.

La seconde classe s'appelle `ListePriorityQueue`. Elle fonctionne un peu comme la file idéale vue en classe. Un tableau contient tous les enregistrements. L'élément le plus prioritaire (plus petite valeur de priorité) se trouve à la fin.

Dans les deux cas, l'entrée 0 du tableau n'est pas utilisée. De plus une table de hachage est utilisée pour connaître la position de tous les éléments. Cette table de hachage est utilisée pour modifier la priorité d'un élément.

La modification de la priorité d'un élément se fait en ramenant l'élément modifié à la clase la plus prioritaire (début ou fin). On procède ensuite à son retrait puis à sa réinsertion avec la nouvelle priorité.

a) (8 pts) Pour chacune des fonctions définies dans l'interface ci-haut et énumérées ci-après, indiquez la complexité asymptotique (en cas moyen) de la méthode en fonction de n, le nombre d'éléments présents. On supposera une distribution uniforme des priorités.

	`HeapPriorityQueue`	`ListPriorityQueue`
`remove()`		
`getMax()`		
`add(…)`		
`updatePriority(…)`		

Le programme suivant est exécuté :

```
 1   public static void main(String[] args) {
 2
 3        HeapPriorityQueue<String> hpq = new HeapPriorityQueue<String>();
 4        ListPriorityQueue<String> lpq = new ListPriorityQueue<String>();
 5        int[] priorities = {4, 5, 6, 1, 2, 5, 3, 4, 2, 6, 3, 4, 6, 2, 5};
 6
 7        for(int i=0; i<priorities.length; i++){
 8            String item = new String("v_" + (i+1));
 9            System.out.println("insert "+item+" avec priorité "+priorities[i]);
10            hpq.add(item, priorities[i]);
11            lpq.add(item, priorities[i]);
12        }
13
14
15        System.out.println("modifie priorité de v_13 avec priorité 1");
16        hpq.updatePriority("v_13", 1);
17        lpq.updatePriority("v_13", 1);
18
19        System.out.println("\nFile de priorité de type monceau");
20
21        while(!hpq.isEmpty())
22            System.out.print(hpq.remove() + ", ");
23
24        System.out.println();
25
26        System.out.println("\nFile de priorité de type liste chaînée");
27
28        while(!lpq.isEmpty())
29            System.out.print(lpq.remove() + ", ");
30
31        System.out.println();
32    }
```

Les lignes 7 à 17 produisent l'affichage suivant :

```
insert v_1 avec priorité 4
insert v_2 avec priorité 5
insert v_3 avec priorité 6
insert v_4 avec priorité 1
insert v_5 avec priorité 2
insert v_6 avec priorité 5
insert v_7 avec priorité 3
insert v_8 avec priorité 4
insert v_9 avec priorité 2
insert v_10 avec priorité 6
insert v_11 avec priorité 3
insert v_12 avec priorité 4
insert v_13 avec priorité 6
insert v_14 avec priorité 2
insert v_15 avec priorité 5
modifie priorité de v_13 avec priorité 1
```

b) (11 pts) Sachant que `remove()` retourne l'élément le plus prioritaire de la file après l'avoir retiré, donnez ci-après l'affichage résultant de l'exécution des lignes 19 à 22.

c) (11 pts) Sachant que `remove()` retourne l'élément le plus prioritaire de la file après l'avoir retiré, donnez ci-après l'affichage résultant de l'exécution des lignes 26 à 29.

Question 2 : Programmation dynamique (20 points)

On désire trouver le parenthésage idéal pour multiplier les matrices A_1 à A_5 permettant de minimiser le nombre de multiplications (scalaires) à effectuer. Les matrices sont dimensionnées comme suit :

$A_1 : 2 \times 3$; $A_2 : 3 \times 2$; $A_3 : 2 \times 1$; $A_4 : 1 \times 3$; $A_5 : 3 \times 2$

Considérez les tables **m** et **s** obtenue par l'exécution de l'algorithme dynamique vu en cours.

m	**1**	**2**	**3**	**4**	**5**
1	**0**	**12**			
2		**0**	**6**	**15**	**18**
3			**0**	**6**	**10**
4				**0**	**6**
5					**0**

s	**1**	**2**	**3**	**4**	**5**
1		**1**			
2			**2**	**3**	**3**
3				**3**	**3**
4					**4**
5					

Compléter cette table pour répondre aux questions suivantes :

Rappel : $m[i, j] = \min\{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\}$ pour $k = i$ à $j-1$, sachant que la matrice A_i a une dimension $p_{i-1} \times p_i$.

a) (5 pts) Donnez le parenthésage optimal pour multiplier A_1 à A_3. Donnez son coût.

Parenthésage optimal: A_1 A_2 A_3

Coût: _____

b) (5 pts) Donnez le parenthésage optimal pour multiplier A_1 à A_4. Donnez son coût.

Parenthésage optimal: A_1 A_2 A_3 A_4

Coût: _____

c) (10 pts) Donnez le parenthésage optimal pour multiplier A_1 à A_5. Donnez son coût.

Parenthésage optimal: A_1 A_2 A_3 A_4 A_5

Coût: _____

Question 3 : Ordre topologique (10 points)

NOTE IMPORTANTE : Les questions 3 à 5 font référence au code Java de l'Annexe 2. Pour les questions 4 et 5, ce code faire intervenir la file de priorité ListPriorityQueue discutée à la question 1 dont l'implémentation est donnée à l'Annexe 1.

La classe `Graph` de l'Annexe 2 permet d'implémenter un graph dirigé ou non dirigé. L'option est donnée à la construction du graphe via un booléen : `public Graph(boolean isDirected)`. Le constructeur par défaut met le booléen à vrai : `public Graph(){ this(true); }`

Le code qui suit crée un graphe dirigé et valué pour lequel on désire trouver un ordre topologique.

```
 1    public static void main(String[] args) {
 2        // On crée un graphe dirigé
 3        Graph graph = new Graph();
 4
 5        graph.addVirtex("a");
 6        graph.addVirtex("b");
 7        graph.addVirtex("c");
 8        graph.addVirtex("d");
 9        graph.addVirtex("e");
10        graph.addVirtex("f");
11        graph.addVirtex("g");
12
13        graph.addEdge("a", "b", 1.0);
14        graph.addEdge("a", "c", 1.0);
15        graph.addEdge("a", "d", 3.0);
16        graph.addEdge("a", "g", 5.0);
17        graph.addEdge("b", "d", 2.0);
18        graph.addEdge("b", "e", 1.0);
19        graph.addEdge("c", "b", 1.0);
20        graph.addEdge("c", "d", 1.0);
21        graph.addEdge("c", "e", 3.0);
22        graph.addEdge("c", "f", 3.0);
23        graph.addEdge("d", "g", 2.0);
24        graph.addEdge("e", "g", 3.0);
25        graph.addEdge("f", "d", 2.0);
26        graph.addEdge("f", "e", 1.0);
27
28        System.out.println( "Graph dirigé: " );
29        System.out.println( graph );
30
31        System.out.println( "Son ordre topologique: " );
32        System.out.println( graph.printTopologicalOrder() + "\n");
33    }
```

Les lignes 28 et 29 produisent l'affichage suivant :

```
Graph dirigé:
a: b, 1.0; c, 1.0; d, 3.0; g, 5.0;
b: d, 2.0; e, 1.0;
c: b, 1.0; d, 1.0; e, 3.0; f, 3.0;
d: g, 2.0;
e: g, 3.0;
f: d, 2.0; e, 1.0;
g:
```

a) (8 pts) Donnez le résultat de l'affichage résultant de l'exécution des lignes 31 et 32. Vous pouvez vous aider du tableau suivant (le remplissage du tableau n'est pas noté).

Nœud	1	2	3	4	5	6	7
a							
b							
c							
d							
e							
f							
g							
Entrée							
Sortie							

Affichage obtenu :

b) (2 pts) Donnez l'ordre trouvé (débuter la numérotation à 1) :

Nœud	a	b	c	d	e	f	g
Ordre :							

Question 4 : Plus court chemin d'un graphe valué (20 points)

NOTE IMPORTANTE : Les questions 3 à 5 font référence au code Java de l'Annexe 2. Pour les questions 4 et 5, ce code faire intervenir la file de priorité ListPriorityQueue discutée à la question 1 dont l'implémentation est donnée à l'Annexe 1. Il est fortement suggéré d'avoir complété la question 1 pour cette question-ci.

La classe `Graph` de l'Annexe 2 permet d'implémenter un graph dirigé ou non dirigé. L'option est donnée à la construction du graphe via un booléen : `public Graph(boolean isDirected)`. Le constructeur par défaut met le booléen à vrai : `public Graph(){ this(true); }`

Le code qui suit crée un graphe dirigé et valué pour lequel on désire trouver tous les chemins partant du sommet « a ». Ce graphe est identique à celui de la question 3.

```
 1    public static void main(String[] args) {
 2        // On crée un graphe dirigé
 3        Graph graph = new Graph();
 4
 5        graph.addVirtex("a");
 6        graph.addVirtex("b");
 7        graph.addVirtex("c");
 8        graph.addVirtex("d");
 9        graph.addVirtex("e");
10        graph.addVirtex("f");
11        graph.addVirtex("g");
12
13        graph.addEdge("a", "b", 1.0);
14        graph.addEdge("a", "c", 1.0);
15        graph.addEdge("a", "d", 3.0);
16        graph.addEdge("a", "g", 5.0);
17        graph.addEdge("b", "d", 2.0);
18        graph.addEdge("b", "e", 1.0);
19        graph.addEdge("c", "b", 1.0);
20        graph.addEdge("c", "d", 1.0);
21        graph.addEdge("c", "e", 3.0);
22        graph.addEdge("c", "f", 3.0);
23        graph.addEdge("d", "g", 2.0);
24        graph.addEdge("e", "g", 3.0);
25        graph.addEdge("f", "d", 2.0);
26        graph.addEdge("f", "e", 1.0);
27
28        System.out.println( "Graph dirigé: " );
29        System.out.println( graph );
30
31        System.out.println( "Les chemins depuis \"a\": " );
32        System.out.println( graph.printPrintPathsFrom("a") + "\n");
33    }
```

Les lignes 28 et 29 produisent l'affichage suivant :

```
Graph dirigé:
a: b, 1.0; c, 1.0; d, 3.0; g, 5.0;
b: d, 2.0; e, 1.0;
c: b, 1.0; d, 1.0; e, 3.0; f, 3.0;
d: g, 2.0;
e: g, 3.0;
f: d, 2.0; e, 1.0;
g:
```

a) (10 pts) Donnez le résultat de l'affichage résultant de l'exécution des lignes 31 et 32. Vous pouvez vous aider du tableau suivant (le remplissage du tableau n'est pas noté).

Nœud	Connu	Dist min.	Parent
a		0,	
b		∞,	
c		∞,	
d		∞,	
e		∞,	
f		∞,	
g		∞,	

Affichage obtenu :

b) (4 pts) Détaillez chacun des chemins les plus courts trouvés :

Destination	Le plus court chemin	Distance parcourue
b	a → _____ → b	
c	a → _____ → c	
d	a → _____ → d	
e	a → _____ → e	
f	a → _____ → f	
g	a → _____ → g	

c) (6 pts) Dans l'exécution de l'Algorithme de Dijkstra par la fonction `printPathsFrom(…)`, la file de prioritaire utilisée pour traiter les sommets est `ListPriorityQueue<Virtex>`. Aurait-il été préférable d'utiliser `HeapPriorityQueue<Virtex>` ? Justifiez clairement mais brièvement votre réponse.

Note : On fait référence ici à la ligne :

```
// Execute Djikstra
PriorityQueue<Virtex> q = new ListPriorityQueue<Virtex>();
```

Question 5 : Arbre sous-tendant minimum (20 points)

NOTE IMPORTANTE : Les questions 3 à 5 font référence au code Java de l'Annexe 2. Pour les questions 4 et 5, ce code faire intervenir la file de priorité ListPriorityQueue discutée à la question 1 dont l'implémentation est donnée à l'Annexe 1. Il est fortement suggéré d'avoir complété la question 1 pour cette question-ci.

La classe `Graph` de l'Annexe 2 permet d'implémenter un graph dirigé ou non dirigé. L'option est donnée à la construction du graphe via un booléen : `public Graph(boolean isDirected)`. Le constructeur par défaut met le booléen à vrai : `public Graph(){ this(true); }`

Le code qui suit crée un graphe dirigé et valué pour lequel on désire trouver tous les chemins partant du sommet « a ». Ce graphe est identique à celui de la question 3.

```
 1    public static void main(String[] args) {
 2        // On crée un graphe non dirigé
 3        Graph graph = new Graph(false);
 4
 5        graph.addVirtex("A");
 6        graph.addVirtex("B");
 7        graph.addVirtex("C");
 8        graph.addVirtex("D");
 9        graph.addVirtex("E");
10
11        graph.addEdge("A", "B", 2.0);
12        graph.addEdge("A", "C", 1.0);
13        graph.addEdge("B", "E", 2.0);
14        graph.addEdge("C", "E", 3.0);
15        graph.addEdge("D", "B", 2.0);
16        graph.addEdge("D", "C", 1.0);
17
18        System.out.println( "Graph non dirigé: " );
19        System.out.println( graph );
20
21        System.out.println( graph.printPrimMinSpanningThree() + "\n");
22
23        System.out.println( graph.printKruskalMinSpanningThree() + "\n");
24    }
```

Les lignes 18 et 19 produisent l'affichage suivant :

```
Graph non dirigé:
A: B, 2.0; C, 1.0;
B: A, 2.0; E, 2.0; D, 2.0;
C: A, 1.0; E, 3.0; D, 1.0;
D: B, 2.0; C, 1.0;
E: B, 2.0; C, 3.0;
```

a) (10 pts) Donnez le résultat de l'affichage résultant de l'exécution de la ligne 21. Vous pouvez vous aider du tableau suivant (le remplissage du tableau n'est pas noté).

Nœud	Connu	Dist min.	Parent
A		0,	
B		∞,	
C		∞,	
D		∞,	
E		∞,	

Affichage obtenu :

b) (10 pts) Donnez le résultat de l'affichage résultant de l'exécution de la ligne 23. Vous pouvez vous aider du tableau suivant (le remplissage du tableau n'est pas noté).

Ordre des arêtes dans la file de priorité

Arête	Poids	Retenue?

Affichage obtenu :

Annexe 1

// INTERFACE PriorityQueue:

```
import java.util.NoSuchElementException;

public interface PriorityQueue<AnyType> {

    int size();
    void clear();
    boolean isEmpty( );
    boolean contains(AnyType x);
    AnyType peek() throws NoSuchElementException;
    AnyType remove() throws NoSuchElementException;
    boolean add(AnyType x, double priority);
    void updatePriority(AnyType x, double priority);
    AnyType getMax() throws NoSuchElementException;
}
```

// CLASSE PQEntry:

```
public class PQEntry<T> {

    T value;
    double priority;

    PQEntry(T value, double priority) {
        this.value = value;
        this.priority = priority;
    }

    void setKey(double priority) {
        this.priority = priority; }
    }
```

// CLASSE HeapPriorityQueue

```
import java.util.Hashtable;
import java.util.NoSuchElementException;

public class HeapPriorityQueue<AnyType> implements PriorityQueue<AnyType>{

    private static final int DEFAULT_CAPACITY = 20;

    private int currentSize;
    private PQEntry<AnyType>[] items;
    Hashtable<AnyType, Integer> indexMap;

    public HeapPriorityQueue(){ initialize(); }

    public AnyType peek() throws NoSuchElementException{
        if( isEmpty( ) ) throw new NoSuchElementException("Priority Queue is empty");
        return items[1].value;
    }
```

```
    public AnyType remove() throws NoSuchElementException {
        if( isEmpty( ) )
            throw new NoSuchElementException("Priority Queue is empty");

        AnyType minItem = items[ 1 ].value;
        indexMap.remove(minItem);

        items[ 1 ] = items[ currentSize-- ];
        percolateDown( 1 );

        return minItem;
    }
```

```
    public boolean add(AnyType x, double priority)
    {
        if( x == null ) return false;
        if( contains(x) ) return false;

        PQEntry<AnyType> newElem = new PQEntry<AnyType>(x, priority);

        if( currentSize == items.length - 1 )
            enlargeArray( items.length * 2 + 1 );

        // Percolate up
        int hole = ++currentSize;
        for( items[ 0 ] = newElem; priority < items[ hole / 2 ].priority; hole /= 2 )
        {
            items[ hole ] = items[ hole / 2 ];

            // update Map
            indexMap.replace(items[ hole ].value, hole);
        }

        // insert new element
        items[ hole ] = newElem;

        // update Map
        indexMap.put(x, hole);

        return true;
    }
```

```
    public int size(){
        return currentSize;
    }

    public void clear(){
        initialize();
    }

    public boolean isEmpty( ){
        return currentSize == 0;
    }
```

```
    public void updatePriority(AnyType x, double priority){
        if( x == null ) return;
        if( !indexMap.containsKey(x) ) return;
        int index = indexMap.get(x);
        updatePriority(index, priority);
    }
```

```
    private void updatePriority(int index, double priority){
        if(index <1 || index > currentSize) return;

        // retrouver l'élément à l'indice index
        PQEntry<AnyType> element = items[index];

        // noter sa priorité actuelle
        double oldPriority = element.priority;

        // si nouvelle priorité identique, quitter
        if(oldPriority == priority) return;

        // sinon, actualiser
        element.setKey(priority);

        // Si un seul élément est présent, on a fini
        if(currentSize == 1) return;

        // Sinon, remonter l'élément à la racine et le retirer
        for(int hole = index; hole != 1; hole /= 2 )
        {
            items[ hole ] = items[ hole / 2 ];
            // update Map
            indexMap.replace(items[ hole ].value, hole);
        }
        items[1] = element;
        remove();

        // insérer à la position désirée
        add(element.value, priority);
    }
```

```
    public AnyType getMax() throws NoSuchElementException {

        if( currentSize == 0 )
            throw new NoSuchElementException("Priority Queue is empty");

        int maxIndex = currentSize/2+1;
        double maxPriority = items[maxIndex].priority;

        for(int i=maxIndex; i<=currentSize; i++)
            if(items[i].priority>maxPriority){
                maxPriority = items[i].priority;
                maxIndex = i;
            }

        return items[maxIndex].value;
    }
```

```
    public boolean contains(AnyType x){
        return indexMap.containsKey(x);
    }

    @SuppressWarnings("unchecked")
    private void enlargeArray( int newSize )
    {
        if(newSize <= (currentSize+1)) return;

        PQEntry<AnyType> [] old = items;
        items = (PQEntry<AnyType>[]) new PQEntry[ newSize ];

        for( int i = 0; i <= old.length; i++ )
            items[ i ] = old[ i ];
    }

    @SuppressWarnings("unchecked")
    private void initialize(){
        currentSize = 0;
        items = (PQEntry<AnyType>[]) new PQEntry[ DEFAULT_CAPACITY ];
        indexMap = new Hashtable<AnyType, Integer>();
    }

    private void percolateDown( int hole ){

        int child;
        PQEntry<AnyType> tmp = items[ hole ];

        for( ; hole * 2 <= currentSize; hole = child ) {
            child = hole * 2;
            if( child != currentSize &&
                items[ child + 1 ].priority < items[ child ].priority )
                child++;

            if( items[ child ].priority < tmp.priority ){
                items[ hole ] = items[ child ];
                // update Map
                indexMap.replace(items[ hole ].value, hole);
            }
            else
                break;
        }

        items[ hole ] = tmp;
        // update Map
        indexMap.replace(tmp.value, hole);
    }
```

```
    public String toString(){
        String output = "";
        for(int i=1; i>=currentSize; i++)
            output += items[i].value + ", ";
        return output;
    }
}
// CLASSE ListePriorityQueue
```

```
import java.util.Hashtable;
import java.util.NoSuchElementException;

public class ListPriorityQueue<AnyType> implements PriorityQueue<AnyType>{

    private static final int DEFAULT_CAPACITY = 20;

    private int currentSize;
    private PQEntry<AnyType>[] items;
    Hashtable<AnyType, Integer> indexTable;

    public ListPriorityQueue(){
        initialize();
    }

    @SuppressWarnings("unchecked")
    private void initialize(){
        currentSize = 0;
        items = (PQEntry<AnyType>[]) new PQEntry[ DEFAULT_CAPACITY ];
        indexTable = new Hashtable<AnyType, Integer>();
    }

    public AnyType peek() throws NoSuchElementException{
        if( isEmpty( ) )
            throw new NoSuchElementException("Priority Queue is empty");

        return items[currentSize].value;
    }
```

```
    public AnyType remove() throws NoSuchElementException {
        if( isEmpty( ) )
            throw new NoSuchElementException("Priority Queue is empty");

        AnyType minItem = items[ currentSize ].value;
        indexTable.remove(minItem);

        currentSize--;

        return minItem;
    }
```

```
    public AnyType getMax() throws NoSuchElementException {
        if( currentSize == 0 )
            throw new NoSuchElementException("Priority Queue is empty");
        return items[1].value;
    }
```

```
    public int size(){ return currentSize; }

    public void clear(){
        initialize();
    }

    public boolean isEmpty( ){
        return currentSize == 0;
```

```
    }

    public boolean contains(AnyType x){
        return indexTable.containsKey(x);
    }
```

```
    public boolean add(AnyType x, double priority)
    {
        if( x == null ) return false;
        if( contains(x) ) return false;

        PQEntry<AnyType> newElem = new PQEntry<AnyType>(x, priority);

        if( currentSize == items.length - 1 )
            enlargeArray( items.length * 2 + 1 );

        // parcourir depuis la fin (plus prioritaire == valeur min de priority)
        // et insérer où approprié
        int index = ++currentSize;

        for(; index > 1 && items[index-1].priority < priority; index--){
            items[index] = items[index-1];
            indexTable.replace(items[index].value, index);
        }

        // positionner et mette à jour la table de hash
        items[index] = newElem;
        indexTable.put(newElem.value, index);

        return true;
    }
```

```
    public void updatePriority(AnyType x, double priority){
        if( x == null ) return;
        if( !indexTable.containsKey(x) ) return;
        int index = indexTable.get(x);
        updatePriority(index, priority);
    }
```

```
    private void updatePriority(int index, double priority){

        if(index <1 || index > currentSize) return;

        // retrouver l'élément à l'indice index
        PQEntry<AnyType> element = items[index];

        // noter sa priorité actuelle
        double oldPriority = element.priority;

        // si nouvelle priorité identique, quitter
        if(oldPriority == priority) return;


        // sinon, actualiser
        element.setKey(priority);
```

```
        // Si un seul élément est présent, on a fini
        if(currentSize == 1) return;

        // Sinon, mettre l'élément à la fin et le retirer
        for(int i=index+1; i <= currentSize; i++){
            items[ i-1 ] = items[ i ];
            // mettre à jour la table de dispersement
            indexTable.replace(items[ i-1 ].value, i-1);
        }

        items[currentSize] = element;
        remove();

        // insérer à la position désirée
        add(element.value, priority);
    }

    @SuppressWarnings("unchecked")
    private void enlargeArray( int newSize )
    {
        if(newSize <= (currentSize+1)) return;

        PQEntry<AnyType> [] old = items;
        items = (PQEntry<AnyType>[]) new PQEntry[ newSize ];

        for( int i = 0; i < old.length; i++ )
            items[ i ] = old[ i ];
    }

    public String toString(){
        String output = "";

        for(int i=currentSize; i>0; i--)
            output += items[i].value + ", ";

        return output;
    }
}
```

Annexe 2

```
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Queue;

import priorityqueue.*; // CODE DE L'ANNEXE 1

public class Graph {

    private enum GraphType {DIRECTED, UNDIRECTED};
    private final GraphType graphType;

    private static final int DEFAULT_CAPACITY = 20;

    int currentVSize, currentESize;
    Virtex[] V; Edge[] E;
    Hashtable<String, Integer> virtexIndexTable;
    Hashtable<String, Integer> edgeIndexTable; // Utile pour arbres sous-tendants min
```

```
    private class Virtex{

        private String name;
        private ArrayList<Edge> neighbours;
        private ArrayList<String> incident;

        public Virtex(String name){
            this.name = name;
            neighbours = new ArrayList<Edge>();
            incident = new ArrayList<String>();
        }

        public boolean addEdge(Edge e){
            if(e == null) return false;
            neighbours.add(e); return true;
        }

        public boolean addIncident(String src) {
            if(src == null || src.equals("")) return false;
            incident.add(src); return true;
        }

        public ArrayList<Edge> getNeighbours(){ return neighbours; }

        public String getName() { return name; }

        public int getInDegree() {
            return incident.size();
        }
    }
```

```
    private class Edge {
        double cost;
        String end1, end2;

        public Edge(String end1, String end2, double cost){
            this.end1 = end1;
            this.end2 = end2;
            this.cost = cost;
        }

        public String getEnd1(){ return end1; }

        public String getEnd2(){ return end2; }

        public double getCost(){ return cost; }

        // permet d'avoir un ID unique, peu important le sens end1 end2
        // si le graph n'est pas dirigé (utilisé pour la table de hash)
        public String id(){
            if( graphType == GraphType.UNDIRECTED &&
                    end1.hashCode() > end2.hashCode())
                return (end2+end1);
            else
                return (end1+end2);
        }
    }
```

```
    /**
     * Constructeurs
     */

    public Graph(){
        this(true);
    }

    public Graph(boolean isDirected){

        graphType = (isDirected ? GraphType.DIRECTED : GraphType.UNDIRECTED);

        currentVSize = 0;
        currentESize = 0;
        V = new Virtex[DEFAULT_CAPACITY];
        E = new Edge[DEFAULT_CAPACITY];
        virtexIndexTable = new Hashtable<String, Integer>();
        edgeIndexTable = new Hashtable<String, Integer>();
    }

    /**
     * Fonctions de construction du graphe
     */

    public boolean addVirtex(String vName){

        if(vName == null || vName.equals("")) return false;
        if(contains(vName)) return false;
```

```
        if( currentVSize == V.length)
            enlargeVArray(2*(currentVSize+1));

        virtexIndexTable.put(vName, currentVSize);
        V[currentVSize++] = new Virtex(vName);
        return true;
    }

    public boolean addEdge(String end1, String end2, double cost){

        if(end1 == null || end1.equals("")) return false;
        if(end2 == null || end2.equals("")) return false;
        if(!contains(end1)) return false;
        if(!contains(end2)) return false;

        Edge e1 = new Edge(end1, end2, cost);
        Edge e2 = new Edge(end2, end1, cost);
        Virtex v1 = V[virtexIndexTable.get(end1)];
        Virtex v2 = V[virtexIndexTable.get(end2)];

        v1.addEdge(e1);

        if(graphType == GraphType.UNDIRECTED)
            v2.addEdge(e2);
        else
            v2.addIncident(v1.getName());

        if( currentESize == E.length)
            enlargeEArray(2*(currentESize+1));

        edgeIndexTable.put(e1.id(),  currentESize);
        E[currentESize++] = e1;

        return true;
    }

    public boolean isEmpty(){
        return (currentVSize == 0);
    }

    public boolean contains(String vName){
        return virtexIndexTable.containsKey(vName);
    }

    private void enlargeVArray( int newSize )
    {
        if(newSize < currentVSize) return;

        Virtex[] oldV = V;
        V = new Virtex[ newSize ];

        for( int i = 0; i < oldV.length; i++ )
            V[ i ] = oldV[ i ];
    }
```

```
/**
 * Ordre topologique (graphe dirigé)
 */

public String printTopologicalOrder(){

    if(isEmpty()) return "Graph is empty";
    if(graphType == GraphType.UNDIRECTED) return "Undirected graph";

    String output = "";

    Queue<Virtex> q = (Queue<Virtex>) new LinkedList<Virtex>();
    int[] indegree = new int[currentVSize];

    for(int i=0; i<currentVSize; i++){
        // prendre le sommet
        Virtex v = V[i];

        // enregistrer son indegree
        indegree[i] = v.getInDegree();

        // ajouter à la file si indegree == 0
        if(indegree[i] == 0) q.add(v);
    }

    // counter permet de détecter un cycle
    int counter = 0;

    // boucle principale de l'algorithme
    while( !q.isEmpty() ){

        Virtex v = q.poll();
        counter++;
        output += v.getName() + ", ";

        for(Edge e : v.getNeighbours()){

            String dst = e.getEnd2();
            int index = virtexIndexTable.get(dst);

            if( --indegree[index] == 0 )
                q.add( V[index] );
        }
    }

    if( counter != currentVSize )
        return "Graph has a cycle";

    return output;

}
```

```
/**
 * Chemin le plus court (Dijkstra)
 */

public String printPathsFrom(String vSrcName){

    if(isEmpty()) return "Graph is empty";
    if(!contains(vSrcName)) return vSrcName + " is not in the graph";

    // Execute Djikstra
    PriorityQueue<Virtex> q = new ListPriorityQueue<Virtex>();
    double[] distances = new double[currentVSize];
    int[] sources      = new int[currentVSize];
    boolean[] known    = new boolean[currentVSize];

    for(int i=0; i<currentVSize; i++){
        distances[i] = Double.POSITIVE_INFINITY;
        sources[i]   = -1;
        known[i]     = false;
    }

    int srcIndex = virtexIndexTable.get(vSrcName);
    distances[srcIndex] = 0.0;
    q.add(V[srcIndex], distances[srcIndex]);

    // boucle principale de l'algorithme
    while( !q.isEmpty() ){

        Virtex v = q.remove();
        int vIndex    = virtexIndexTable.get(v.getName());
        double vCost  = distances[vIndex];
        known[vIndex] = true;

        for(Edge e : v.getNeighbours()){

            String dst = e.getEnd2();
            double cost = e.getCost();
            int index = virtexIndexTable.get(dst);

            if(!known[index] && (vCost + cost < distances[index])){

                sources[index] = vIndex;
                double newDistance = vCost + cost;

                if(distances[index] == Double.POSITIVE_INFINITY)
                    q.add(V[index], newDistance);
                else if(distances[index] > newDistance)
                    q.updatePriority(V[index], newDistance);

                distances[index] = newDistance;
            }
        }
    }
```

```
        // print paths
        String output = "";

        for(int i=0; i<currentVSize; i++){
            output += printPath(i, sources);
            output += " : " + distances[i] + "\n";
        }

        return output;
    }
```

```
    /**
     * Fonction pour Dijkstra
     */
    private String printPath(int index, int[] sources){

        String output = V[index].getName();

        if( sources[index] == -1 )
            return output;
        else
            return printPath(sources[index], sources) + "->" + output;
    }
```

```
    /**
     * Arbres sous-tendants minimum (graphe non-dirigé)
     */

    public String printPrimMinSpanningThree(){

        if(isEmpty()) return "Graph is empty";
        if(graphType == GraphType.DIRECTED) return "Directed graph";

        // Exécute Prim
        PriorityQueue<Virtex> q = new ListPriorityQueue<Virtex>();
        double[] costs  = new double[currentVSize];
        int[] sources   = new int[currentVSize];
        boolean[] edges = new boolean[currentESize];
        boolean[] known = new boolean[currentVSize];

        for(int i=0; i<currentVSize; i++){
            costs[i]   = Double.POSITIVE_INFINITY;
            sources[i] = -1;
            known[i]   = false;
        }

        for(int i=0; i<currentESize; i++){
            edges[i] = false;
        }

        // premier noeud choisi arbitrairement
        costs[0] = 0.0;
        q.add(V[0], costs[0]);
```

```
        // boucle principale de l'algorithme
        while( !q.isEmpty() ){

            Virtex v = q.remove();
            int vIndex = virtexIndexTable.get(v.getName());
            known[vIndex] = true;

            for(Edge e : v.getNeighbours()){

                String neighbour = e.getEnd2();
                double cost = e.getCost();
                int index = virtexIndexTable.get(neighbour);

                if(!known[index] && (cost < costs[index])){

                    sources[index] = vIndex;

                    if(costs[index] == Double.POSITIVE_INFINITY)
                        q.add(V[index], cost);
                    else
                        q.updatePriority(V[index], cost);

                    costs[index] = cost;
                }
            }
        }

        for(int i=0; i<currentVSize; i++)
            if( sources[i] != -1){
                Edge e = new Edge(V[sources[i]].getName(), V[i].getName(), costs[i]);
                edges[edgeIndexTable.get(e.id())] = true;
            }

        // génère l'arbre sous forme de graphe
        Graph g = generateTree(edges);

        // calculer le coût
        double totalCost = cost(edges);

        return "Coût = " + totalCost + "\n" + g.toString();
    }
```

```
    public String printKruskalMinSpanningThree(){

        if(isEmpty()) return "Graph is empty";
        if(graphType == GraphType.DIRECTED) return "Directed graph";

        // Exécute Kruskal
        PriorityQueue<Edge> edgePQ = new ListPriorityQueue<Edge>();
        boolean[] edges = new boolean[currentESize];
        int[] sets = new int[currentVSize];

        for(int i=0; i<currentVSize; i++)
            sets[i] = -1;
```

```
        for(int i=0; i<currentESize; i++){
            edgePQ.add(E[i], E[i].getCost());
            edges[i] = false;
        }

        while(!edgePQ.isEmpty()){

            Edge e = edgePQ.remove();

            int v1 = virtexIndexTable.get(e.getEnd1());
            int v2 = virtexIndexTable.get(e.getEnd2());

            if( union(v1, v2, sets) )
                edges[edgeIndexTable.get(e.id())] = true;
        }

        // génère l'arbre sous forme de graphe
        Graph g = generateTree(edges);

        // calculer le coût
        double totalCost = cost(edges);

        return "Coût = " + totalCost + "\n" + g.toString();
    }
```

```
    /**
     * Fonctions pour manipuler les ensembles disjoints
     */

    /**
     * @param item : élémement à identifier
     * @param sets : éléments présents (-1 indique un identifiant)
     * @return indice de l'identifiant
     */
    private int find(int item, int[] sets){
        if( sets[item] == -1 ) return item;
        return find(sets[item], sets);
    }

    /**
     * @param set1 : élément du premier ensemble
     * @param set2 : élément du second ensemble
     * @param sets : éléments présents (-1 indique un identifiant)
     * @return vrai si ensembles ont été unis (si set1 et set2
     *                              appartiennent à des ensembles disjoints)
     */
    private boolean union(int set1, int set2, int[] sets){

        int id1 = find(set1, sets);
        int id2 = find(set2, sets);

        if(id1 != id2){ sets[id2] = id1; return true; }

        return false;
    }
```

```
/**
 *  Fonctions utilisées par arbre sous-tendant minimum
 */

// génère un arbre en éliminant les arcs signalés par false
private Graph generateTree(boolean[] edges){

    // créer le graphe non dirigé qui repréentera l'arbre
    Graph g = new Graph(false);

    // créer les noeuds
    for(int i=0; i<currentVSize; i++)
        g.addVirtex(V[i].getName());

    // ajouter les arcs
    for(int i=0; i<currentESize; i++)
        if( edges[i] )
            g.addEdge(E[i].getEnd1(), E[i].getEnd2(), E[i].getCost());

    return g;
}

// calcule le cout de l'un arbre obtenu en éliminant les arcs signalés par false
private double cost(boolean[] edges){

    double totalCost = 0.0;

    for(int i=0; i<currentESize; i++)
        if( edges[i] )
            totalCost += E[i].getCost();

    return totalCost;
}

/**
 * Fonctions utiles
 */
```

```
public String toString(){
    String output = "";

    for(int i=0; i<currentVSize; i++){

        Virtex v = V[i];
        output += v.getName() + ": ";

        ArrayList<Edge> neighbours = v.getNeighbours();
        for(Edge e: neighbours){
            output += e.getEnd2() + ", " + e.getCost() + "; ";
        }

        output += "\n";
    }

    return output;
}
```

```
    private void enlargeEArray( int newSize )
    {
        if(newSize < currentVSize) return;

        Virtex[] oldV = V;
        V = new Virtex[ newSize ];

        for( int i = 0; i < oldV.length; i++ )
            V[ i ] = oldV[ i ];
    }
}
```