POLYTECHNIQUE MONTRÉAL

Corrigé examen final

Q1	
Q2	
Q3	
Q4	
Q5	
Total	

INF2010

Sigle du cours

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()`	O(log(n))	O(1)
`getMax()`	O(n)	O(1)
`add(…)`	O(1)	O(n)
`updatePriority(…)`	O(log(n))	O(n)

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.

```
File de priorité de type monceau
v_4, v_13, v_5, v_9, v_14, v_11, v_7, v_8, v_1, v_12, v_2, v_6, v_15, v_3, v_10,
```

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.

```
File de priorité de type liste chaînée
v_13, v_4, v_14, v_9, v_5, v_11, v_7, v_12, v_8, v_1, v_15, v_6, v_2, v_10, v_3,
```

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**	**12**	**18**	**22**
2		**0**	**6**	**15**	**18**
3			**0**	**6**	**10**
4				**0**	**6**
5					**0**

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

Compléter cette table pour répondre aux questions suivantes :

<u>Rappel</u> : $m[i, j] = \min\{m[i, k] + m[k+1, j] + p_{i-1}p_k,p_j \}$ pour $k = i$ à $j-1$, sachant que la matrice A_i a une dimension p_{i-1} x 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: 12

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:18

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: 22

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.

```
public static void main(String[] args) {
    // On crée un graphe dirigé
    Graph graph = new Graph();

    graph.addVirtex("a");
    graph.addVirtex("b");
    graph.addVirtex("c");
    graph.addVirtex("d");
    graph.addVirtex("e");
    graph.addVirtex("f");
    graph.addVirtex("g");

    graph.addEdge("a", "b", 1.0);
    graph.addEdge("a", "c", 1.0);
    graph.addEdge("a", "d", 3.0);
    graph.addEdge("a", "g", 5.0);
    graph.addEdge("b", "d", 2.0);
    graph.addEdge("b", "e", 1.0);
    graph.addEdge("c", "b", 1.0);
    graph.addEdge("c", "d", 1.0);
    graph.addEdge("c", "e", 3.0);
    graph.addEdge("c", "f", 3.0);
    graph.addEdge("d", "g", 2.0);
    graph.addEdge("e", "g", 3.0);
    graph.addEdge("f", "d", 2.0);
    graph.addEdge("f", "e", 1.0);

    System.out.println( "Graph dirigé: " );
    System.out.println( graph );

    System.out.println( "Son ordre topologique: " );
    System.out.println( graph.printTopologicalOrder() + "\n");
}
```

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 :

```
Son ordre topologique:
a, c, b, f, d, e, g,
```

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

Nœud	a	b	c	d	e	f	g
Ordre :	1	3	2	5	6	4	7

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.

```java
public static void main(String[] args) {
    // On crée un graphe dirigé
    Graph graph = new Graph();

    graph.addVirtex("a");
    graph.addVirtex("b");
    graph.addVirtex("c");
    graph.addVirtex("d");
    graph.addVirtex("e");
    graph.addVirtex("f");
    graph.addVirtex("g");

    graph.addEdge("a", "b", 1.0);
    graph.addEdge("a", "c", 1.0);
    graph.addEdge("a", "d", 3.0);
    graph.addEdge("a", "g", 5.0);
    graph.addEdge("b", "d", 2.0);
    graph.addEdge("b", "e", 1.0);
    graph.addEdge("c", "b", 1.0);
    graph.addEdge("c", "d", 1.0);
    graph.addEdge("c", "e", 3.0);
    graph.addEdge("c", "f", 3.0);
    graph.addEdge("d", "g", 2.0);
    graph.addEdge("e", "g", 3.0);
    graph.addEdge("f", "d", 2.0);
    graph.addEdge("f", "e", 1.0);

    System.out.println( "Graph dirigé: " );
    System.out.println( graph );

    System.out.println( "Les chemins depuis \"a\": " );
    System.out.println( graph.printPrintPathsFrom("a") + "\n");
}
```

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 :

```
Les chemins depuis "a":
a : 0.0
a->b : 1.0
a->c : 1.0
a->c->d : 2.0
a->b->e : 2.0
a->c->f : 4.0
a->c->d->g : 4.0
```

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

Destination	Le plus court chemin	Distance parcourue
b	a → b	1
c	a → c	1
d	a → c → d	2
e	a → b → e	2
f	a → c → f	4
g	a → c → d → g	4

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>();
```

Trois méthodes sont principalement appelées dans ce code. `remove()`, `add(…)` et `updatePriority(…)` dont voici les complexités :

	HeapPriorityQueue	ListPriorityQueue
remove()	O(log(n))	O(1)
add(…)	O(1)	O(n)
updatePriority(…)	O(log(n))	O(n)

`remove()` est appelée O(|V|) fois, tandis que `add(…)` et `updatePriority(…)` sont appelées O(|E|) fois. Le graphe étant peu dense, O(|E|) est proportionnel à O(|V|). Il apparaît donc clairement que `HeapPriorityQueue` est un meilleur choix puisqu'on obtiendrait un complexité O(|V|log(|V|)) au lieu de O(|V|^2).

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

<u>NOTE IMPORTANTE :</u> 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 :

```
Coût = 6.0
A: B, 2.0; C, 1.0;
B: A, 2.0; E, 2.0;
C: A, 1.0; D, 1.0;
D: C, 1.0;
E: B, 2.0;
```

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 :

```
Coût = 6.0
A: C, 1.0;
B: E, 2.0; D, 2.0;
C: A, 1.0; D, 1.0;
D: B, 2.0; C, 1.0;
E: B, 2.0;
```