INF2010

Contrôle périodique 2

Mon nom de famille est : _____Bourai_____

Mon prénom est : _____Sami_____

Mon matricule est : _____2041659_____

Directives :

- Le contrôle périodique 2 prend la forme d'un devoir à faire chez soi ;
- Vous avez jusqu'au 29 novembre pour compléter le travail ;
- L'examen est noté sur un total de 10 points et compte pour 15% de la note finale du cours ;
- Ne posez pas de question concernant l'examen. En cas de doute sur le sens d'une question, énoncez clairement toute supposition que vous faites.
- Ne communiquez avec personne au sujet de l'examen. Cet examen doit être réalisé de manière individuelle.
- Quand vous aurez fini, remettez un PDF comportant vos réponses aux questions ;
- La remise s'effectue sur Moodle ;
- Les remises par courriel ne sont pas acceptées ;
- La remise doit se faire au plus tard le 29 novembre à 23h59.

Réécrivez ci-après la phrase « J'affirme sur mon honneur avoir fait cet examen sans l'aide de personne » :

J'affirme sur mon honneur avoir fait cet examen sans l'aide de personne

---

Bonne chance!

Pour cet examen, il vous est demandé de compléter certaines fonctionnalités d'un code Java fourni dont les sources sont disponibles dans le fichier compressé `intra2_a2020.zip`. Le fichier compressé contient les fichiers suivants :

Fichier	Description
`Intra2A2020.java`	`Classe contenant la fonction principale main()`
Autres fichiers `*.java`	Diverses classes déjà implémentée
Fichier texte `ods.txt`	Dictionnaire officiel du Scrabble utilisé au CP1

L'examen se subdivise en trois parties. Vous devrez changer la valeur de certains booléens dans la fonction principale `main()` pour les exécuter. Pour que le code compile, vous devez modifier la ligne indiquée ci-après dans le fichier `Util.java` avec vos informations personnelles.

```
public class Util {

   private static final int MON_MATRICULE = ; // <= A COMPLETER
```

En exécutant le programme, vous devriez voir s'afficher un texte similaire à ceci (les lettres affichées dépendent du matricule ; ici, nous avons utilisé 1625144) :

```
Voici les 7 lettres tirées [E, S, Z, O, L, S, M]
```

Note : Si votre matricule produit une peu de combinaisons possibles (3 et moins), ajoutez-y 1 ou 2 pour remédier à la situation (inscrivez 2011510 à la place de 2011509 par exemple).

Pensez à indiquer le changement ci après :

Matricule utilisé : _____

Partie 1 :

Dans cette partie, on désire reprendre le problème du CP1 et déterminer la meilleure combinaison d'une pioche du jeu de Scrabble au moyen d'un monceau.

1) **(0.5 point)** Donnez l'affichage résultant de l'ajout de votre matricule au fichier `Util.java`. Le résultat de l'affichage devrait être aberrant. Pouvez-vous expliquer en quoi et pourquoi ?

Affichage :

```
C:\Users\Lenovo\.jdks\openjdk-15\bin\java.exe "-javaagent:C:\Program Files\JetBrains\I
Voici les 7 lettres tir◆es [V, O, H, R, Z, K, F]
Meilleur choix trouv◆ par tri
Mot:    ROKH    Valeur: 16
Meilleur choix trouv◆ par le monceau
Mot:    OR  Valeur: 2
```

Discussion :

En fait, quand on utilise le quickSort() pour trier les combinaisons obtenues, on obtient la meilleure valeur 16 pour le mot QUE. Ce qui est normal, puisque selon l'algorithme de ce dernier les valeurs sont triées de façon ascendante. Donc de la valeur la plus petite (OR Valeur :2) à la valeur la plus grande (ROKH Valeur :16) et ainsi ce tri nous retourne la dernière valeur du tableau qui correspond au meilleur choix.

Quant au monceau, ce dernier est de type minimal. Donc la racine (root) contient la plus petite valeur et plus on descend dans l'arbre plus on trouve de plus grandes valeurs. Donc les éléments sont triés de façon descendante. Or, lorsqu'on fait appel à la fonction deleteRoot(), cette dernière nous retoume minItem trouver par findMin(). Lorsque nous allons dans findMin(), celle-ci nous retourne le root donc la plus petite valeur (array[1]). Nous contrairement à cela, nous cherchons la meilleure valeur. Voilà donc pourquoi c'est aberrant.

2) **(1 point)** Afin de corriger l'affichage obtenu en (1), vous devez modifier la méthode `percolateDown` dans la classe `BinaryHeap`. Reproduisez ci-après votre implémentation. Reproduisez l'affichage ainsi obtenu pour démontrer le bon fonctionnement de votre code.

Implémentation de `percolateDown` :

```
  private void percolateDown( int hole ){
     int child;
     AnyType tmp = array[ hole ];

     for( ; hole * 2 <= currentSize; hole = child ) {
        child = hole * 2;
        if( child != currentSize &&
            array[ child + 1 ].compareTo( array[ child ] ) > 0 )
           child++;
        if( array[ child ].compareTo( tmp ) > 0 )
           array[ hole ] = array[ child ];
        else
           break;
     }
     array[ hole ] = tmp;
  }
}
```

Affichage :

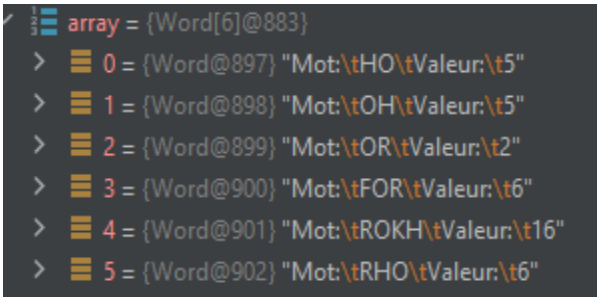
```
Voici les 7 lettres tirées [V, O, H, R, Z, K, F]
Meilleur choix trouvé par tri
Mot:    ROKH    Valeur: 16
Meilleur choix trouvé par le monceau
Mot:    ROKH    Valeur: 16

Process finished with exit code 0
```

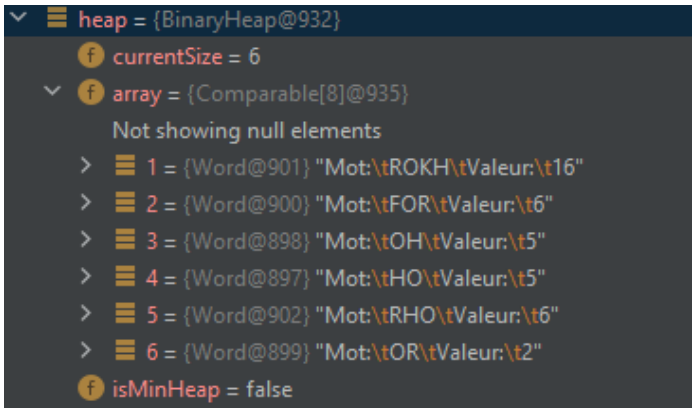
3) **(1 point)** Comparez les résultats obtenus par tri et par monceau selon les éléments suivants : Sont-ils identiques ou différents ? Quelle raison explique la ressemblance ou la dissemblance ? Quelle méthode est plus performante et pourquoi ?

Les résultats sont : Identiques _____Oui_____; Différents _____Non_____

Discussion sur la ressemblance ou la dissemblance :

Les résultats que j'obtiens sont identiques puisque j'ai qu'un seul mot dont la valeur est 16 et ce dernier est placé à la dernière case du tableau qui récupère les combinaison (array[]). Dans le cas du heap comme c'est un max heap le root correspond à la valeur la plus grande. Telles que dans les images ci-dessous.





Meilleure option : Tri `quickSort` _____; Monceau _____Oui_____

Justification du choix :

La meilleure option étant le tri par monceau puisque celle-ci en pire cas représente une complexité de O(nlog(n)). Quant au quickSort(), lui il peut atteindre une complexité en pire cas de $O(n^2)$.

Partie 2 :

On désire utiliser un arbre AVL pour manipuler les combinaisons possibles. Pour ce faire, la classe `AvlTree` du livre de Weiss a été empruntée et modifiée.

4) **(0.5 point)** Mettez le booléen `partie2` à **`true`** comme suit, l'affichage du programme est modifié. Reproduisez l'affichage obtenu :

```
public static void main(String args[]) {

   boolean partie2 = true; // <= Mettre à true pour la partie 2
```

```
C:\Users\Lenovo\.jdks\openjdk-15\bin\java.exe
Nombre d'elements inseres: 6

Arbre AVL obtenu:

|__Mot: HO  Valeur: 5
   |__Mot:  OR  Valeur: 2
   |__Mot:  FOR Valeur: 6
      |__Mot:   ROKH    Valeur: 16
```

5) **(0.5 point)** Donnez en fonction de *n*, le nombre de nœuds dans l'AVL, la complexité asymptotique en meilleur cas de la méthode publique `printFanycTree()`. Justifiez votre réponse.

Le meilleur cas étant celui ou par exemple l'arbre est vide alors on sort directement de la fonction et la complexité est de O(1).

```
if( isEmpty( ) )
   return "Empty tree";
```

6) **(1 point)** L'affichage obtenu en (4) devrait indiquer le nombre d'éléments insérés dans l'arbre AVL suivi de l'affichage stylisé de l'arbre AVL obtenu. Comparez le nombre d'éléments insérés dans l'arbre AVL et le nombre d'éléments contenus dans l'arbre. Expliquez les différences s'il y en a, ou l'absence de différence si vous n'en voyez aucune.

Le nombre total de valeur est 6, mais notre arbre en contient que 4. Ceci est dû au fait que les duplications ne sont pas ajoutées. Ceci se passe dans le else de la méthode insert().

```
else
   ;  // Duplicate; do nothing
```

7) **(1point)** Quelle modification faut-il apporter à la classe `Word` pour garantir que le nombre d'éléments contenus dans l'arbre soit identique au nombre d'éléments insérés ?

Pour ajouter les doublons on retourne un -1 dans compareTo(Word w) et ainsi les jouter à chaque fois à gauches.

```
@Override
public int compareTo(Word w) {
   if ( value-w.value==0 )
      return -1;
   return value - w.value;
}
```

8) **(1point)** La classe AvlTree comporte un membre statique appelé ALLOWED_IMBALANCE fixé à 1. Sachant que la hauteur maximale (h_{max}) d'un AVL est $O(\log(n))$ lorsque ALLOWED_IMBALANCE est 1, pouvez-vous affirmer que h_{max} est $O(\log(n))$ lorsque ALLOWED_IMBALANCE fixé à 2 ? Justifiez votre réponse.

Non, h_{max} n'est pas $O(\log(n))$ lorsque ALLOWED_IMBALANCE fixé à 2.

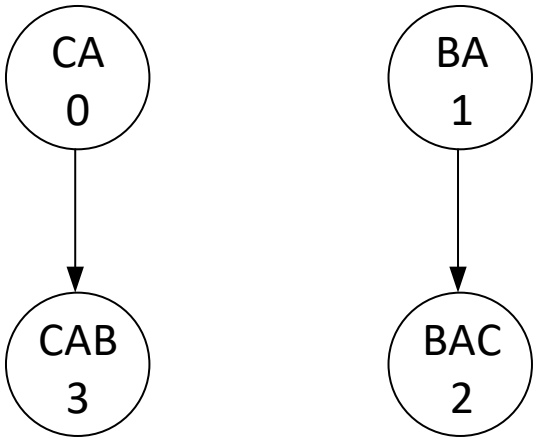
Ceci peut se justifier en calculant la hauteur max quand on a un débalancement de plus de 2, par exemple dans un graphe de n=3.

Log (3)=1.5 or on est supposé obtenir 2 ce qui justifie que hmax change lorsque ALLOWED_IMBALANCE fixé à 2.

Partie 3 :

On désire construire un graphe dirigé à partir des combinaisons possibles trouvées précédemment. Les sommets de ce graphe sont associés à chaque combinaison admissible. Un arc relie deux sommets si le sommet de départ est une sous-séquence du sommet d'arrivée, au sens où nous l'avons vu dans le cours sur les PLSC. Par exemple, si les lettres tirées étaient `[A, B, C]`, nous aurions 4 combinaisons possibles et le graphe suivant :

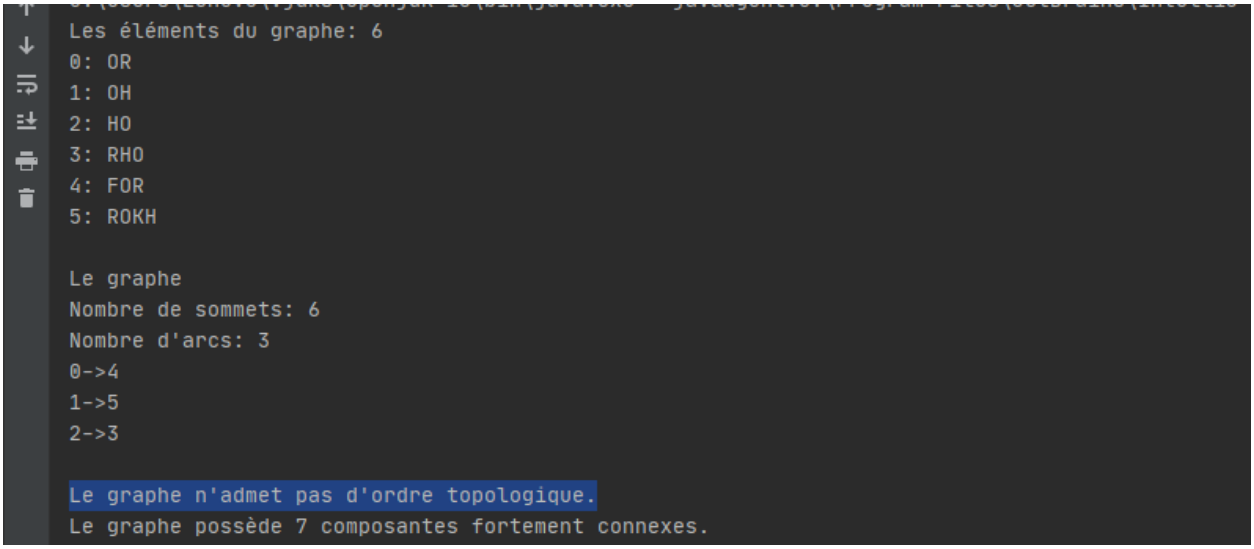
```
Voici les lettres tirées [A, B, C]
Les éléments du graphe: 4
0: CA
1: BA
2: BAC
3: CAB

Le graphe
Nombre de sommets: 4
Nombre d'arcs: 2
0->3
1->2
```



On voit alors que le sommet 0 pointe vers 3 puisque CA est une sous-séquence de CAB.

Note : Rappelons que sous-séquence ne signifie pas préfixe. Par exemple, CA est aussi une sous-séquence de INCRIMINA.

9) **(1 point)** Mettez le booléen `partie3` à **`true`**. L'affichage résultant de l'exécution du programme indique-t-il que le graphe ainsi obtenu admet un ordre topologique ? Combiens de composantes fortement connexes le graphe obtenu admet-il selon l'affichage ? Ces résultats vous semblent-t-ils corrects ? Justifiez votre réponse.



Le graphe admet un ordre topologique selon l'affichage : OUI _____; NON__XX_____

Discutez le résultat :

Dans la classe CFC, `hasOrder,le boolean, est initialisé à false donc c'est logique que cela affiche qu'il n'y a aucun ordre. Or,` cela est faux, le graphe admet un ordre topologique tel que l'affichage le démontre 0 -> 4, 1->5 et 2->3. Donc il n'y a aucun cycle. Ce qui fait qu'il y a un ordre topologique.

Le graphe admet __7__ composantes fortement connexes selon l'affichage.

Discutez le résultat :

Non, le graphe ne possède pas 7 composantes fortement connexes. Il est formé de 6 composantes, comment peut-t-il contenir plus de composantes fortement connexes que de sommets qui le forme, soit 6? Donc cela est faux aussi.

10) **(1 point)** La méthode `findTopologicalOrder()` de la classe `TopologicalOrder` est incomplète, en cela qu'elle devrait mettre `hasOrder` à vrai si le graphe admet un ordre topologique et elle ne le fait pas. Proposez ci-après une modification à la méthode `findTopologicalOrder()` pour que la situation soit corrigée.

```
// Completer
if ( counter != G.V() ) {   // si counter!= nombre de vertex alors il y a un cycle.
   hasOrder =false;            // donc pas d'ordre topologique
   //throw new UnderflowException(); // larguer l'exception

} ////// sinon
else {
   hasOrder = true;   //// il y a un ordre
}
```

```
Intra2A2020
Nombre d'arcs: 3
0->4
1->5
2->3

Le graphe admet un ordre topologique.
Le graphe possède 7 composantes fortement connexes.

Process finished with exit code 0
```

11) **(1 point)** L'implémentation de la classe `CFC` est erronée, en cela qu'elle ne retourne pas le bon nombre de composantes fortement connexes. Proposez une modification simple à la classe `CFC` qui permette de remédier à la situation.

```
public CFC(DirectedGraph G){

   if(G == null)
      throw new InvalidParameterException();

   dfsMarked = new boolean[G.V()];
   marked    = new boolean[G.V()];
   id        = new int[G.V()];
   count     = 0;          // initialisation du compteur a 0 plutot qu'a 1;
   dfo       = new LinkedList<Integer>();

   findCfc(G);

}
```

```
Nombre d'arcs: 3
0->4
1->5
2->3

Le graphe admet un ordre topologique.
Le graphe possède 6 composantes fortement connexes.
```

12) **(0.5 point)** Donnez la complexité asymptotique en pire cas de la méthode `findCfc` de la classe `CFC`.

E = arcs et V= sommets

La complexité en pire cas est O (|E| + |V|)

Lorsqu'on retrace tout ce qui est utilisé dans `findCfc()` on remarque qu'elle fait appelle à la fonction `findOrder()`, sans aller plus en profondeur cette dernière est de O (|E|+|V|). À cela s'ajoute la boucle for qui va jusqu'à G.V() nombre de sommets. Donc, comme la complexité revient à la plus grande des d'entre les deux alors c'est O (|E| + |V|) soit celle de findOrder().