UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN

STRUCTURES DE DONNÉES ET ALGORITHMES (SINF1121)

Mission 1 - Réponses aux questions

– 25 septembre 2013 –

TRAVAIL DU GROUPE 9 :

GÉGO Anthony	28581100	MOYAU Arnold	xxxxxx00
GENA Xavier	xxxxxx00	NAVEAU Adrien	xxxxxx00
JOVENEAU Quentin	xxxxxx00	PAYEN Marlon	xxxxxx00
LIBIOULLE Thibault	60271100		

1 Question 1

Définissez ce qu'est un type abstrait de données (TAD). En Java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?

Un type abstrait de données est un modèle mathématique d'une structure de données spécifiant le type d'informations stockées, les opérations supportées sur ces dernières, ainsi que les types de paramètres des opérations. Un TAD spécifie ce que chaque opération fait, mais pas comment elle le fait.

De ce fait, en Java, un TAD peut être décrit au moyen d'une interface, puisqu'elle ne consiste qu'en une liste de déclarations de méthodes sans implémentation.

2 Question 2

Comment faire pour implémenter une pile par une liste simplement chaînée où les opérations push et pop se font en fin de liste ? Cette solution est-elle efficace ? Argumentez.

L'essentiel est de maintenir une référence vers l'objet en fin de liste. Chaque nœud possède alors une référence vers l'élément contenu ainsi que vers le nœud qui le précède.

L'opération push se réalisera, quelque soit la taille de la pile, en un nombre constant d'opérations. Il suffit en effet que de lié le nouveau nœud à l'actuel dernier, et de modifier la référence pointant vers le dernier nœud vers le nouveau. Un raisonnement similaire peut être appliqué pour l'opération pop qui consiste à enlever le dernier nœud et modifier la référence du dernier nœud vers le précédent.

Au vu de ce raisonnement, nous pouvons conclure que la solution sera tout aussi efficace que si l'opération était effectuée en début de liste.

3 Question 3

En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe java.util.Stack. Cette classe peut-elle convenir comme implémentation de l'interface Stack décrite dans DSAJ-5 ? Pourquoi ?

La classe java.util.Stack étend la classe Vector à laquelle sont ajoutées les opérations nécessaires pour être utilisable comme pile. Un Vector est un tableau qui est redimensionné dynamiquement en fonction des besoins. Si, d'un point de vue, la complexité spatiale peut être réduite car il n'est pas nécessaire de retenir les références vers les noeuds suivants, il ne faut cependant pas négliger les opérations nécessaire pour la réallocation du tableau à chaque redimensionnement.

Puisque l'interface est complètement indépendante de l'implémentation et que les méthodes push et pop sont définies, la classe peut ainsi servir d'implémentation de l'interface Stack décrite dans le livre de référence.

4 Question 4

Proposez une implémentation de la classe DNodeStack. Il s'agit d'une classe similaire à NodeStack (décrite dans DSAJ-5) qui propose une implémentation générique d'une pile. Votre classe DNodeStack doit utiliser une implémentation en liste doublement chaînée générique. Elle reposera donc sur une classe DNode<E> (similaire à Node<E>, décrite dans DSAJ-5) que vous devez définir. Ajoutez, dans la classe DNodeStack, une méthode public String toString() qui renvoie une chaîne de caractères représentant le contenu de la pile. Commentez votre code.

```
1
/**
3  * Decrivez votre classe DNode ici.
 *
5  * @author (votre nom)
 * @version (un numero de version ou une date)
```

```
 */
public class DNode<E>
{
    // Instance variables:
    private E element;
    private DNode<E> next;
    private DNode<E> previous;

    /* Creates a node with null references to its element and next node. */
    public DNode() {
       this(null, null, null);
    }

    /* Creates a node with the given element and next node. */
    public DNode(E e, DNode<E> n, DNode<E> p) {
        element = e;
        next = n;
        previous = p;
    }

    // Accessor methods:
    public E getElement() {
        return element;
    }

    public DNode<E> getNext()
    {
        return next;
    }

    public DNode<E> getPrevious()
    {
        return previous;
    }

    // Modifier methods:
    public void setElement(E newElem) {
        element = newElem;
    }

    public void setNext(DNode<E> newNext) {
        next = newNext;
    }

    public void setPrevious(DNode<E> newPrevious) {
        previous = newPrevious;
    }
}
```

Listing 1 – DNode.java

```
/**
* Interface for a stack: a collection of objects that are inserted
* and removed according to the last-in first-out principle. This
* interface includes the main methods of java.utiI.Stack.
*
* @author Roberto Tamassia
* @author Michael Goodrich
* @see EmptyStackException
*/
public interface Stack<E> {

    /**
    * Return the number of elements in the stack.
    * @return number of elements in the stack.
    */
    public int size();

    /**
    * Return whether the stack is empty.
    * @return true if the stack is empty, false otherwise.
    */
    public boolean isEmpty();

    /**
    * Inspect the element at the top of the stack ..
```

```
26      * @return top element in the stack.
        * @exception EmptyStackException if the stack is empty.
28      */
        public E top() throws EmptyStackException;
30
        /**
32      * Insert an element at the top of the stack.
        * @param element to be inserted.
34      */
        public void push (E element);
36
        /**
38      * Remove the top element from the stack.
        * @return element removed.
40      * @exception EmptyStackExceptionif the stack is empty.
        */
42      public E pop() throws EmptyStackException;
    }
```

Listing 2 – Stack.java

```
1   /**
     * Decrivez votre classe DNodeStack ici.
3    *
     * @author (votre nom)
5    * @version (un numero de version ou une date)
     */
7   public class DNodeStack<E> implements Stack<E>
    {
9       protected DNode<E> top; // reference to the head node
        protected DNode<E> bottom;
11      protected int size; // number of elements in the stack
        public DNodeStack()
13      { // constructs an empty stack
            top = null;
15          bottom = null;
            size =0 ;
17      }

19      public int size() { return size; }

21      public boolean isEmpty()
        {
23          if (top == null) return true;
            return false;
25      }

27      public void push(E elem)
        {
29          DNode<E> v = new DNode<E>(elem, top, null); // create and, link-in a new node
            if(top != null) top.setPrevious(v);
31          else bottom = v;
            top = v;
33          size++;
        }
35
        public E top() throws EmptyStackException
37      {
            if (isEmpty())
39              throw new EmptyStackException("!!Stack is empty. !!");
            return top.getElement();
41      }

43      public E pop() throws EmptyStackException
        {
45          if(isEmpty())
                throw new EmptyStackException("!!Stack is empty. !!");
47          E temp = top.getElement();
            top = top.getNext(); // link-out the former top node slze--; return temp;
49          size--;
            return temp;
51      }

53      public String toString()
        {
55          DNode<E> current = top;
```

```
        String result = "Contents : ";
        while(current != null)
        {
           //String prev = (current.getPrevious() != null) ? current.getPrevious().
               toString() : "null";
           //String next = (current.getNext() != null) ? current.getNext().toString() :"
               null";
           result +=  current.getElement().toString() + ((current.getNext() != null) ? "
               -->" : "");
           current = current.getNext();
        }
        return result;
    }

}
```

Listing 3 – DNodeStack.java

```

/**
 * Decrivez votre classe EmptyStackException ici.
 *
 * @author (votre nom)
 * @version (un numero de version ou une date)
 */
public class EmptyStackException extends Exception
{
    /**
     * Constructeur d'objets de classe EmptyStackException
     */
    public EmptyStackException()
    {
    }

    public EmptyStackException(String msg)
    {
    }
}
```

Listing 4 – EmptyStackException.java

5 Question 5

Complétez l'interface Queue (décrite dans DSAJ-5) contenant une interface pour le type abstrait file en ajoutant des préconditions et postconditions pour chacune des méthodes. Votre spécification correspond-elle à une programmation défensive ?

```
public interface Queue<E> {

    /**
    * Returns the number of elements in the queue.
    * @return number of elements in the queue.
    * @pre -
    * @post size() >= 0
    */
    public int size();

    /**
    * Returns whether the queue is empty.
    * @return true if the queue is empty, false otherwise.
    * @pre -
    * @post isEmpty() = true or false
    */
    public boolean isEmpty();

    /**
    * Inspects the element at the front of the queue.
    * @return element at the front of the queue.
    * @exception EmptyQueueException if the queue is empty.
    * @pre - (exception thrown if the queue is empty)
    * @post front() = the front element
    */
```

```
26      public E front() throws EmptyQueueException;

28      /**
        * Inserts an element at the rear of the queue.
30      * @param element new element to be inserted.
        * @pre element != null
32      * @post element E is enqueued
        */
34      public void enqueue (E element);

36      /**
        * Removes the element at the front of the queue.
38      * @return element removed.
        * @exception EmptyQueueException if the queue is empty.
40      * @pre - (exception thrown if queue is empty)
        * @post dequeue() = the front element, and element is dequeued
42      */
        public E dequeue() throws EmptyQueueException;
44  }
```

Listing 5 – Queue.java

```
    public class EmptyQueueException extends Exception
2   {
        public EmptyQueueException()
4       {
            // initialisation des variables d'instanc
6       }
    }
```

Listing 6 – EmptyQueueException.java

Ajouter des préconditions à une méthode signifie que cette dernière est prévue pour fonctionner correctement si et seulement si les préconditions sont respectées. Dès lors, à partir du moment où l'on néglige de vérifier l'entrée, on ne peut pas parler de programmation défensive. En effet, spécifier des préconditions n'empêche en rien à l'utilisateur de passer des paramètres n'y répondant pas.

6 Question 6

> *Comment faire pour implémenter le type abstrait de données File à l'aide de deux piles ? Décrivez en particulier le fonctionnement des méthodes enqueue et dequeue dans ce cas. A titre d'exemple, précisez l'état de chacune des deux piles après avoir inséré les entiers 1 2 3 à partir d'une file initialement vide. Décrivez ce qu'il se passe ensuite lorsque l'on effectue l'opération dequeue. Quelle est la complexité temporelle de ces méthodes si l'on suppose que chaque opération pop et push s'exécute en temps constant. Cette implémentation d'une file est-elle efficace par rapport aux autres implémentations présentées dans DSAJ-5 ?*

Le but de cette première manipulation est que le premier élément insérer (**enqueue**) soit le premier à sortir (**dequeue**), on appelle ce principe FIFO. Prenons deux piles que nous nommons *entrée* et *sortie*, lorsque nous utilisons la méthode **enqueue**, nous faisons un **push** dans *entrée*. Ensuite, si nous utilisons **dequeue** :

- Nous devons vérifier si *sortie* est vide. Si oui, on utilise la méthode **push** de *sortie* avec comme argument le résultat du **pop** sur *entrée* ;
- On utilise la méthode **pop** sur *sortie* et on retourne le résultat.

L'action réalisée lorsque *sortie* est vide a pour but d'inverser le sens de la pile *entrée*. Comme cela, le premier élément insérer dans *entrée* sera le dernier élément dans *sortie* et donc le premier à sortir. Un exemple est donné dans la figure 1.

Au niveau de la complexité temporelle, celle de **enqueue** reste $O(1)$, puisqu'on utilise juste un **push**. Par contre si *sortie* est vide, **dequeue** a une complexité de $2n + 1$ opération puisqu'on opère n **push** sur *sortie*, n **pop** sur *entrée* et 1 **pop** sur *sortie*(ce cas est le pire cas possible). On peut simplifier en disant que cette complexité temporelle est $O(n)$.

Par rapport à l'implémetation proposée dans DSAJ-5, cette implémentation n'est pas efficace puisque la complexité de **dequeue** est $O(n)$ dans ce cas et $O(1)$ dans le livre.

Étape	*entrée*	*sortie*	Résultat
Départ	3 2 1	(vide)	/
Dequeue	(vide)	2 3	1

FIGURE 1 – Exemple de file implémentée avec deux piles

7 Question 7

Comment faire en Java pour lire des données textuelles depuis un fichier et pour écrire des résultats dans un fichier ASCII ? Écrivez en Java une méthode générique, c'est-à-dire aussi indépendante que possible de son utilisation dans un contexte particulier, de lecture depuis un fichier texte. Faites de même pour l'écriture dans un fichier ASCII.

```
1  import java.io.BufferedReader;
   import java.io.BufferedWriter;
3  import java.io.File;
   import java.io.FileReader;
5  import java.io.FileWriter;
   import java.io.IOException;
7  import java.io.PrintWriter;

9
   public class FileRW {
11
     private String content;
13   private String pathFile;
     private File file;
15

17   public FileRW (String path){
       content = "";
19     pathFile = path;
       file = new File(pathFile);
21   }

23   public void readFile(){
       String lineTMP = "";
25     try {
         BufferedReader reader = new BufferedReader(new FileReader(file));
27       while ((lineTMP = reader.readLine()) != null){
           content = content + lineTMP + "\n";
29       }
       } catch (IOException e) {
31       System.out.println("Error while reading file!");
         e.printStackTrace();
33     }
     }
35
   public void writeFile(String toWrite){
37   try {
       PrintWriter writer = new PrintWriter (new BufferedWriter (new FileWriter(file)));
39     writer.println(toWrite);
       writer.close();
41   } catch (IOException e) {
       System.out.println("Error while writing file!");
43     e.printStackTrace();
     }
45 }

47 public String toString(){
     return content;
49 }

51 }
```

Listing 7 – FileRW.java

8 Question 8

Comment faire en Java pour passer des arguments à un programme ? Soyez précis. Donnez un exemple illustrant l'intérêt de cette opération.

Quand on écrit notre programme, nous passons par une classe « main » pour instancier nos classes et réaliser le traitement adéquat. Nous pouvons attribuer à cette classe « main » un tableau composé des arguments de notre choix sous le format « String ». Si nous voulons utiliser un autre type, nous pouvons passer par des parseurs.

Ces arguments seront précisés après la compilation, c'est-à-dire lors de l'exécution du programme. Pour cela, via l'interprète de commande, nous écrivons l'instruction « java », suivi le nom du fichier contenant la classe « main » ainsi que le(s) argument(s) que vous voulez passer au programme.

Exemple

```
class Carre
{
    float cote;

    public Carre(float cote)
    {
            this.cote = cote;
    }

    public float perimetre()
    {
        return(this.cote * 4);
    }

    public float aire()
    {
        return(this.cote * this.cote);
    }

    public String toString()
    {
        return("Perimetre : "+  perimetre() + "/nAire : " + aire());
    }
}
```

Listing 8 – Carre.java

```
public class TestCarre
{
    public static void main(String args[])
    {
        Carre c = new Carre(Float.parseFloat(args[0])) ;

        System.out.println(c.toString());
    }
}
```

Listing 9 – TestCarre.java

```
javac TestCarre.java
```

Listing 10 – Compilation

```
java TestCarre 3.5
```

Listing 11 – Exécution (3.5 correspond à l'argument que l'on passe au programme)