

# STRUCTURES DE DONNÉES ET ALGORITHMES (SINF1121)

## Mission 1 - Réponses aux questions

– 25 septembre 2013 –

### TRAVAIL DU GROUPE 9 :

GÉGO Anthony	28581100
GENA Xavier	xxxxxx00
JOVENEAU Quentin	xxxxxx00
LIBIOULLE Thibault	xxxxxx00

MOYAU Arnold	xxxxxx00
NAVEAU Adrien	xxxxxx00
PAYEN Marlon	xxxxxx00

## 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 lier 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 nœuds suivants, il ne faut cependant pas négliger les opérations nécessaires 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 /**
2  * Décrivez votre classe DNode ici.
3  *
4  * @author (votre nom)
5  * @version (un numero de version ou une date)
```

```

7  */
public class DNode<E>
9  {
    // Instance variables:
11 private E element;
    private DNode<E> next;
13 private DNode<E> previous;

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

19     /* Creates a node with the given element and next node. */
21     public DNode(E e, DNode<E> n, DNode<E> p) {
        element = e;
23         next = n;
        previous = p;
25     }

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

31     public DNode<E> getNext()
33     {
        return next;
35     }

37     public DNode<E> getPrevious()
39     {
        return previous;
    }

41     // Modifier methods:
43     public void setElement(E newElem) {
        element = newElem;
45     }

47     public void setNext(DNode<E> newNext) {
        next = newNext;
49     }

51     public void setPrevious(DNode<E> newPrevious) {
        previous = newPrevious;
53     }
}

```

Listing 1 – DNode.java

```

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

12     /**
    * Return the number of elements in the stack.
14     * @return number of elements in the stack.
    */
    public int size();

18     /**
    * Return whether the stack is empty.
20     * @return true if the stack is empty, false otherwise.
    */
22     public boolean isEmpty();

24     /**
    * Inspect the element at the top of the stack ..

```

```

26      * @return top element in the stack.
27      * @exception EmptyStackException if the stack is empty.
28      */
29      public E top() throws EmptyStackException;
30
31      /**
32      * Insert an element at the top of the stack.
33      * @param element to be inserted.
34      */
35      public void push (E element);
36
37      /**
38      * Remove the top element from the stack.
39      * @return element removed.
40      * @exception EmptyStackException if the stack is empty.
41      */
42      public E pop() throws EmptyStackException;
43  }

```

Listing 2 – Stack.java

```

1  /**
2   * Décrivez votre classe DNodeStack ici.
3   *
4   * @author (votre nom)
5   * @version (un numero de version ou une date)
6   */
7  public class DNodeStack<E> implements Stack<E>
8  {
9      protected DNode<E> top; // reference to the head node
10     protected DNode<E> bottom;
11     protected int size; // number of elements in the stack
12     public DNodeStack()
13     { // constructs an empty stack
14         top = null;
15         bottom = null;
16         size = 0 ;
17     }
18
19     public int size() { return size; }
20
21     public boolean isEmpty()
22     {
23         if (top == null) return true;
24         return false;
25     }
26
27     public void push(E elem)
28     {
29         DNode<E> v = new DNode<E>(elem, top, null); // create and, link-in a new node
30         if(top != null) top.setPrevious(v);
31         else bottom = v;
32         top = v;
33         size++;
34     }
35
36     public E top() throws EmptyStackException
37     {
38         if (isEmpty())
39             throw new EmptyStackException("!!Stack is empty. !!");
40         return top.getElement();
41     }
42
43     public E pop() throws EmptyStackException
44     {
45         if(isEmpty())
46             throw new EmptyStackException("!!Stack is empty. !!");
47         E temp = top.getElement();
48         top = top.getNext(); // link-out the former top node size--; return temp;
49         size--;
50         return temp;
51     }
52
53     public String toString()
54     {
55         DNode<E> current = top;

```

```

57     String result = "Contents : ";
    while(current != null)
    {
59         //String prev = (current.getPrevious() != null) ? current.getPrevious().
            toString() : "null";
        //String next = (current.getNext() != null) ? current.getNext().toString() : "
            null";
61         result += current.getElement().toString() + ((current.getNext() != null) ? "
            -->" : "");
        current = current.getNext();
63     }
    return result;
65 }
67 }

```

Listing 3 – DNodeStack.java

```

1  /**
3   * Décrivez votre classe EmptyStackException ici.
4   *
5   * @author (votre nom)
6   * @version (un numéro de version ou une date)
7   */
8  public class EmptyStackException extends Exception
9  {
10     /**
11      * Constructeur d'objets de classe EmptyStackException
12      */
13     public EmptyStackException()
14     {
15     }
16
17     public EmptyStackException(String msg)
18     {
19     }
20 }

```

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 ?

```

1  public interface Queue<E> {
2
3     /**
4      * Returns the number of elements in the queue.
5      * @return number of elements in the queue.
6      * @pre -
7      * @post size() >= 0
8      */
9     public int size();
10
11     /**
12      * Returns whether the queue is empty.
13      * @return true if the queue is empty, false otherwise.
14      * @pre -
15      * @post isEmpty() = true or false
16      */
17     public boolean isEmpty();
18
19     /**
20      * Inspects the element at the front of the queue.
21      * @return element at the front of the queue.
22      * @exception EmptyQueueException if the queue is empty.
23      * @pre - (exception thrown if the queue is empty)
24      * @post front() = the front element
25      */
26 }

```

```

26     public E front() throws EmptyQueueException;
28
29     /**
30      * Inserts an element at the rear of the queue.
31      * @param element new element to be inserted.
32      * @pre element != null
33      * @post element E is enqueued
34      */
35     public void enqueue (E element);
36
37     /**
38      * Removes the element at the front of the queue.
39      * @return element removed.
40      * @exception EmptyQueueException if the queue is empty.
41      * @pre - (exception thrown if queue is empty)
42      * @post dequeue() = the front element, and element is dequeued
43      */
44     public E dequeue() throws EmptyQueueException;

```

Listing 5 – Queue.java

```

public class EmptyQueueException extends Exception
2 {
3     public EmptyQueueException()
4     {
5         // initialisation des variables d'instanc
6     }
7 }

```

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.