Algorithmique et structures de données : Mission 1

Ivan Ahad - Jérôme Bertaux - Rodolphe Cambier - Guillaume Coutisse
Baptiste Degryse - Wojciech Grynczel - Joachim De Droogh - Thomas Grimée - Benoît Ickx

24 septembre 2014

1. **Définissez ce qu'est un type abstrait de données (TAD)? (Groupe)**
   Modèle mathématique de structure de données qui spécifie le type de données enregistrées, les opérations supportées par ces données et le type des paramètres de ces opérations. Il s'agit d'une sorte de contrat, où on dit ce que la structure doit être capable de faire, mais on ne dit pas comment. Exemple de TAD : arbres, piles, files, listes.

   En java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?
   Il est préférable d'utiliser une interface, car une interface est juste une liste de méthodes, et pas d'attributs, contrairement à la classe qui peut être implémentée et instanciée. Les TAD étant des structures, des « contrats », ils correspondent donc mieux à l'interface.

2. **Comment faire pour implémenter une pile par une liste simplement chaînée où les opérations push et pop se font en fin de liste ? (Groupe)**
   Il faut parcourir toute la liste dont chaque node contient un élément et une référence vers le node suivant, et au dernier node, on en rajoute un nouveau contenant l'élément à ajouter.

   Cette solution est-elle efficace ? Argumentez.
   La solution n'est pas efficace car pour faire push ou pop il faut toujours parcourir toute la liste. Ainsi, au plus la liste est grande au plus cela prendra du temps pour ajouter un élément. Lorsque les opérations de push et pop se font en fin de liste, la complexité temporelle est en O(n). Une meilleure solution serait d'utiliser les opérations push et pop en début de liste. Cela permettrait donc de ne pas devoir parcourir toute la pile pour effectuer les opérations. Dès lors, la complexité temporelle serait dans ce cas en O(1), c'est à dire en temps constant, la taille de la liste n'importerait donc pas.

3. **En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe java.util.Stack. (Ivan Ahad & Rodolphe Cambier)**

   Dans l'API de Java, l'implémentation de la pile se fait avec 5 méthodes. Les méthodes push et pop permettent de placer un élément au sommet de la pile ou de l'enlever, la méthode peek permet de jeter un coup d'œil sur l'élément se trouvant au sommet de la liste, la méthode empty permet de savoir si oui ou non la pile est vide, et enfin la méthode search permet de chercher un élément et nous donne la distance de l'élément recherché par rapport au sommet de la pile. Les piles utilisent le principe de last-in-first-out(LIFO). Ainsi, le dernier élément placé au sommet de la pile sera le premier à être enlevé du sommet de celle-ci.

   Cette classe peut-elle convenir comme implémentation de l'interface Stack 4 décrite dans DSAJ-5 ? Pourquoi ?

   L'implémentation ne convient pas car les interfaces ne sont pas similaires dû au fait que le nom des méthodes ne sont pas les mêmes. Dans DSAJ-5, les méthodes peek et empty s'appellent respectivement top et isEmpty. Cela poserait donc bien problème lors de leur implémentation. Cependant, si les noms des méthodes étaient les mêmes, la classe décrite dans l'API de Java conviendrait pour l'implémentation de l'interface décrite dans le livre DSAJ-5. A la seule différence que la méthode push décrite dans l'API de Java retourne l'élément placé au sommet de la pile tandis que cette même méthode décrite dans DSAJ-5 ne retourne rien.

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. (Wojciech Grynczel & Benoît Ickx)**

```
package s2;
//code et specifications ecrites par Baptiste Degryse + modif par Benoit Ickx
public class DNode<E> {
  protected E elem;
  protected DNode<E> prev, next;
  /**
  * cree un noeud
  * @param item : l'element contenu par le noeud
  * @param next : le noeud suivant
  * @param prev : le noeud precedent
  */
  public DNode(E item, DNode<E> next, DNode<E> prev){
    elem = item;
    this.next=next;
    this.prev=prev;
  }

  /**
  * retourne l'element du noeud
  * @return element du noeud
  */
  public E getElem(){
    return elem;
  }

  /**
  * retourne le noeud precedent
  * @return noeud precedant
  */
  public DNode<E> getPrev(){
    return prev;
  }

  /**
  * change le noeud precedent
  * @param prev le nouveau noeud precedent
  */
  public void setPrev(DNode<E> prev){
    if(prev != null){
      if(this.prev != null){
        this.prev.next = prev;
        prev.prev = this.prev;
        }
      prev.next = this;
      this.prev = prev;
    }
  }

  /**
  * retourne le noeud suivant
  * @return noeud suivant
  */
  public DNode<E> getNext(){
    return next;
  }

```

```
  /**
  * change le noeud suivant
  * @param next le nouveaux noeud suivant
  */
  public void setNext(DNode<E> next){
    if(next != null){
      if(this.next != null){
        this.next.prev = next;
        next.next = this.next;
      }
      next.prev = this;
      this.next = next;
    }
  }
}
```

Listing 1: DNode.java

```
package s2;
//code et specifications ecrit par Baptiste Degryse
  public class DNodeStack<E>{
  protected DNode<E> head=null;
  protected int size=0;

  /**
  * cree une nouvelle pile vide
  */
  public DNodeStack(){
  }

  /**
  * cree une nouvelle pile avec elem au dessus de la pile
  * @param elem : le premier element de la pile
  */
  public DNodeStack(E elem){
    head=new DNode<E>(elem,null,null);
    size=1;
  }

  /**
  * rajoute un element sur le sommet de la pile
  * @param elem : element ajoute au dessus de la pile
  */
  public void push(E elem){
    DNode<E> node=new DNode<E>(elem,head,null);
    if(head!=null)
      head.setPrev(node);
    head=node;
    size++;
  }

  /**
  * retire et renvoie l'element au sommet de la pile. Renvoie null en cas de pile
    vide.
  * @return element du sommet de la pile.
  */
  public E pop(){
    if(size==0)
      return null;
    E elem=head.getElem();
    head=head.getNext();
    if(head!=null)
      head.setPrev(null);
    size--;
    return elem;
  }

  /**
  * retourne true si la pile est vide, false sinon
  * @return true si empty et false sinon
  */
  public boolean empty(){
    return size==0;
```

```
55    }
56
57    /**
58    * regarde l'element du sommet de la pile, sans le retirer
59    * @return element au sommet de la pile
60    */
61    public E peek(){
62      if(head!=null)
63        return head.getElem();
64      return null;
65    }
66
67    /**
68    * returne l'indice de l'objet recherche. Retourne -1 si l'objet n'a pas été trouv
        é dans la pile.
69    * @param o : objet recherche
70    * @return indice de l'objet, -1 si il n'est pas dans la pile
71    */
72    public int search(Object o){
73      int notFound=-1;
74      if(head==null)
75        return notFound;
76      DNode<E> node=head;
77      for(int i=0;node!=null;i++){
78        if(node.getElem().equals(o))
79          return i;
80        node=node.getNext();
81      }
82      return notFound;
83    }
84
85    /**
86    * retourne la pile sous forme de String
87    * @return String representant la pile
88    */
89    public String toString(){
90      if(size==0)
91        return "";
92      StringBuffer buf=new StringBuffer("[ ");
93      for(DNode<E> node=head;node!=null; node=node.getNext()){
94        buf.append(node.getElem().toString() +" - ");
95      }
96      //buf.append("> BOTTOM");
97      return buf.toString();
98    }
99  }
```

Listing 2: DNodeStack.java

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 ? (Joachim De Droogh & Jérome Bertaux)**

   Le fait que la queue doit être instanciée est une précondition à appliquer de manière générale.

	Précondition	Postcondition
int size()	-	-
boolean isEmpty()	-	-
void enqueue (E Element)	- S'il y a déjà un élément dans la queue, il faut que le nouvel élément inséré soit du même type, - Gérer le throws de l'exception, - Erreur si la taille est limitée et que la queue est déjà pleine	-
E first()	-	-
E dequeue()	-	-

   Ces spécifications correspondent à une programmation défensive, car les conditions ajoutées permettent d'éviter les erreurs, qui pourraient mettre le système dans un état insistant.
   Remarque : la liste des méthodes est issue du DSAJ version 6.

```
1   public interface Queue<E> {
2     /** Returns the number of elements in the queue. */
3     int size();
4     /** Tests whether the queue is empty. */
5     boolean isEmpty();
6     /** Inserts an element at the rear of the queue. */
7     void enqueue(E e);
8     /** Returns, but does not remove, the first element of the queue (null if
    empty). */
9     E first();
10    /** Removes and returns the first element of the queue (null if empty). */
11    E dequeue();
12  }
13
```

Listing 3: Interface Queue DSAJ version 6 page 220 :

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

   Pour réaliser une pile à l'aide de deux files, nous devons trouver un moyen de retirer et enlever le 1er élément de la « pile » à l'aide des fonctions « enqueue » et « dequeue ». Pour se faire nous allons donc utiliser les deux files, la première est vide et la seconde contient les éléments mis dans la pile, on considère (via un boolean) que la pile est par exemple la seconde file. Les opérations « isempty » et « size » sont dont les même et doivent juste être appelé sur la bonne file (ce qu'on sait faire facilement via le boolean). Pour la méthode « pop », nous utilisons la fonction « dequeue », celle-ci permet de retirer et de retourner le 1er élément de la liste (qui se trouve être notre sommet de la pile) et de la retourner. Pour réaliser la méthode « push » nous allons avoir besoin des deux files. En effet la fonction « enqueue » rajoute un élément à la fin de la file, or nous voulons qu'il le rajoute au début de celle-ci (donc au sommet). Nous allons donc rajouter cet élément dans la file vide et recopier le contenu de l'autre liste via les fonctions « dequeue » et « enqueue » à la suite du nouvel élément. Il ne reste qu'à changer le boolean pour dire que notre pile est l'autre file.

Exemple :

```
File  A                File  B                Pile(file B)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
|   |   |   |   |   |  |   |   |   |   |   |  |   |   |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
```

```
push(1) : A.enqueue(1);
```

```
File  A                File  B                Pile(file A)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
| 1 |   |   |   |   |  |   |   |   |   |   |  | 1 |   |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
```

```
push(2) : B.enqueue(2) ; B.enqueue(A.pop()) ;
```

```
File  A                File  B                Pile(file A)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
| 1 |   |   |   |   |  | 2 |   |   |   |   |  | 1 |   |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+

->

File  A                File  B                Pile(file B)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
|   |   |   |   |   |  | 2 | 1 |   |   |   |  | 2 | 1 |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
```

```
push(3) : A.enqueue(3) ; A.enqueue(B.pop()) ;  A.enqueue(B.pop()) ;
```

```
File  A                File  B                Pile(file B)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
| 3 |   |   |   |   |  | 2 | 1 |   |   |   |  | 2 | 1 |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+

->

File  A                File  B                Pile(file B)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
| 3 | 2 |   |   |   |  | 1 |   |   |   |   |  | 1 |   |   |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+

->

File  A                File  B                Pile(file A)
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
| 3 | 2 | 1 |   |   |  |   |   |   |   |   |  | 3 | 2 | 1 |   |   |
+---+---+---+---+---+  +---+---+---+---+---+  +---+---+---+---+---+
```

On peut donc facilement voir que la complexité temporelle de « pop » est O(1) car il ne dépend pas de la taille des files, il se contente juste de retourner et retirer le 1er élément de la liste. Pour « push », la complexité temporelle est O(n) car il dépend de la taille de la seconde file qui doit être vidée et mise dans la 1er, il va donc effectuer l'opération de retirer et mettre un élément de la seconde dans la 1er file n fois pour les n éléments. Enfin, vu que la complexité du push est plus élevée, l'implémentation est moins efficace que celle proposée dans le bouquin.

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. (Baptiste Degryse & Thomas Grimée)**

```java
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;

/**
*
* @author Thomas Grimée
*
*/
public class ReadWrite {

  /**
  * Creer un ArrayList<String> contenant toutes les lignes du fichier
  * fileName.
  *
  * @param fileName
  *            Contient le path du fichier qui doit etre lu.
  * @return Un ArrayList<String> dont chaque element est une ligne du fichier
  *         fileName.
  * @throws IOException
  *             Si une erreur I/O intervient.
  */
  public static ArrayList<String> mRead(String fileName) throws IOException {
    ArrayList<String> out = new ArrayList<String>();
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    String line;
    while ((line = reader.readLine()) != null) {
      out.add(line);
    }
    reader.close();
    return out;
  }

  /**
  * Ecrit en ASCII dans le fichier fileName tous les String contenu dans
  * lineFile. Si le fichier fileName n'existe pas, il est cree.
  *
  * @param fileName
  *            Contient le path du fichier qui doit etre ecrit.
  * @param lineFile
  *            Contient les lignes a ecrire dans le fichier fileName.
  * @throws FileNotFoundException
  *             Si fileName ne peut etre cree, ouvert, ou que c'est un
  *             repertoire.
  * @throws UnsupportedEncodingException
  *             Si ASCII n'est pas supporte
  */
  public static void mWrite(String fileName, ArrayList<String> lineFile)
  throws FileNotFoundException, UnsupportedEncodingException {
    PrintWriter w = new PrintWriter(fileName, "ASCII");
    for (int i = 0; i < lineFile.size(); i++) {
      w.println(lineFile.get(i));
    }
    w.close();
  }
}
```