Premier rapport - Semaine 2

Groupe 2.2

Vendredi 25 septembre 2014

Question 1 : Thomas Celant

Le type abstrait de données est un ensemble de données, contenant les méthodes et opérations permettant de les utiliser. La file ou la pile sont des types courants de TAD rencontrés. Il est plus avantageux de décrire un TAD sur une interface, car il est par la suite plus facile de modifier l'implémentation de celui-ci. Cela rend donc aussi le TAD moins dépendant de son implémentation.

Question 2 : Charles Jacquet

Premièrement, une liste simplement chaînée est une liste composée de noeuds, dans laquelle chaque noeud contient des données et la référence du noeud suivant (lié uniquement dans un sens). Le premier noeud est appelé la tête (head) et le dernier élément est appelé la queue (tail). La liste est référencée par un élément gardant la référence de la tête de la liste. La référence au noeud suivant (appelé next) de la queue est null.
Une pile est une liste dans laquelle on ne peut ajouter et supprimer des éléments qu'à la queue de la liste par des opérations appelées respectivement push et pop.
Implémentation des deux fonctions :

- Push :
  Pour ajouter un élément, il faut parcourir la liste jusqu'à ce que l'élément de référence au prochain noeud soit null (la queue). Mettre la référence de ce noeud vers le nouveau noeud juste créé avec les données reçues et dont la référence au prochain noeud (next) est null.
  Lorsque la liste est vide. Il suffit de créer le noeud (toujours avec le next = null) et de mettre la référence de début de liste vers ce noeud.
  La complexité de cette fonction est de $\theta(n)$
- Pop :
  Pour supprimer un élément, il faut parcourir la liste jusqu'à ce que la référence contenue dans le noeud suivant soit null (next.next est null). Dans ce cas, il faut renvoyer les données de l'élément suivant et mettre la référence du noeud actuel (this.next) à null. Dans le cas où il n'y a qu'un seul élément (la tête = la queue) il faut simplement renvoyer les données de la tête et ensuite la supprimer.
  La complexité de cette fonction est aussi de $\theta(n)$

On se rend compte que ce n'est pas très efficace car dans les deux cas, que ce soit pour supprimer ou ajouter un élément, il faut parcourir toute la liste ($\theta(n)$). Une liste doublement chaînée pourrait être beaucoup plus efficace si nous gardions une référence vers la queue. Dans ce cas, la complexité serait en $\theta(1)$.

Question 3 : Zacharie Kerger

En consultant la documentation relative à la pile sur l'API de Java, on constate que la classe Stack étend la classe Vector avec 5 opérations supplémentaires afin de permettre à un vecteur d'adopter le comportement d'une pile.

- Des méthodes push et pop pour ajouter et retirer des éléments de la pile
- Une méthode peek afin d'obtenir le premier élément de la pile sans le retirer
- Une méthode empty qui vérifie si la pile est vide ou non
- Une méthode search utilisée pour chercher un objet dans la pile et savoir à quelle distance il se trouve par rapport au premier élément

Cette classe ne peut pas convenir comme implémentation de l'interface Stack décrite dans DSAJ-5 parce qu'elle ne contient pas de méthode size et que la signature de la méthode push n'est pas la même que celle présente dans l'interface.

Question 4 : Aurian de Potter

```
public class DNodeStack<E> implements Stack<E>
{
  protected DNode<E> top;
  protected int size;

  // Contruction d'une pile vide
  public DNodeStack()
  {
    top = null;
    size = 0;
  }

  public int size()
  {
    return size;
  }

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

  // Rajoute un element dans la pile
  public void push(E elem)
  {
    DNode<E> v = new DNode<E>(elem, top, null);
    top = v;
    v.getNext().setPrev(v);
    size++;
  }

  // Renvoie le premiere element de la pile
  public E top() throws EmptyStackException
  {
    if (isEmpty()) throw new EmptyStackException("Stack is empty.");
    return top.getElement();
  }

  // Enleve le premier element de la pile
  public E pop() throws EmptyStackException
  {
    if (isEmpty()) throw new EmptyStackException("Stack is empty.");
    E temp = top.getElement();
    top = top.getNext();
    top.setPrev(null);
    size--;
    return temp;
  }

  // Affiche le pile sous forme de string
  public String toString()
  {
    String out = "[";
    DNode<E> v = top;
    if(v == null) return (out + "]");
    while(v.getNext() != null)
    {
      out += (v.getElement().toString() + ",");
      v = v.getNext();
    }
    out += (v.getElement().toString() + "]");
    return out;
  }
}
```

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

```
  private DNode<E> prev;

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

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

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

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

  public DNode<E> getPrev()
  {
    return prev;
  }

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

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

  public void setPrev(DNode<E> prev)
  {
    this.prev = prev;
  }
}
```

```
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.
   * @return top element in the stack.
   * @exception EmptyStackException if the stack is empty.
   */
  public E top()
      throws EmptyStackException;
  /**
   * Insert an element at the top of the stack.
   * @param element to be inserted.
   */
  public void push (E element);
  /**
```

```
    * Remove the top element from the stack.
    * @return element removed.
    * @exception EmptyStackException if the stack is empty.
    */
   public E pop()
       throws EmptyStackException;
}
```

```
@SuppressWarnings("serial")
public class EmptyStackException extends RuntimeException
{
  public EmptyStackException(String err)
  {
    super(err);
  }
}
```

Question 5 : Lemaire Jerome

Dans cette question, il était demandé de compléter l'interface se trouvant à la page 220 du DSAJ-6 en ajoutant les préconditions et les postconditions.
Voici l'interface complétée :

```

public interface Queue<E e> {
  @pre : La file n'est pas modifiee pendant que la methode est appliquee.
  @post : Retourne le nombre d'elements presents dans la file  si la file est vide alors la
    methode retourne null
  @exception :

    int size();


  @pre : La file n'est pas modifiee pendant que la methode est appliquee.
  @post : Retourne true si la file est vide sinon false
  @exception :

    boolean isempty();

  @pre : this n'est pas modifiee pendant l'ajout de l'element e.L'element e est du meme type
    que la liste.
  @post :Ajout l'element e a l'arriere de la file
  @exception : Si e n'est pas un element du type attendu alors une  InvalidElementException
    est lancee.

    void enqueue (E e)  throws InvalidElementException;

  @pre : La file n'est pas modifiee durant l'application de la methode.
  @post : Retourne le premier element de la file sans le retirer et retourne null si la file
    est vide
  @exception :

    E first();


  @pre : La file n'est pas modifiee durant l'application de la methode.
  @post : Retourne le premier element de la file et le retire. Retourne null  si la file est
    vide
  @exception :

    E dequeue();
```

Dans cette interface, les spécifications n'ont pas besoin d'être défensives car les méthodes n'autorisent pas l'utilisateur à modifier directement la pile sauf pour la méthode enqueue. La méthode enqueue doit donc vérifier si les préconditions sont respectées et lancer une exception en cas de non respect de celles-ci. Mais est-ce judicieux sachant qu'en java, une telle erreur lance automatiquement une exception ?

Question 6 : Arnaud Dethise

Soient deux files, q_1 et q_2.
push(A) : ajouter A dans la file non-vide (q_1).
pop() : transférer tous les éléments de la file non-vide (q_1) dans la file vide (q_2), à l'exception du dernier élément. Retourner le dernier élément. q_2 est maintenant la file non-vide, q_1 est la file vide.

Ainsi, les opérations suivantes :

```
push(1)
push(2)
push(3)
a = pop()
b = pop()
```

se traduisent en utilisant des files par :

```
q1.enqueue(1)                 %   q1: [1]          q2: []
q1.enqueue(2)                 %   q1: [2,1]        q2: []
q1.enqueue(3)                 %   q1: [3,2,1]      q2: []
q2.enqueue(p1.dequeue())      %   q1: [3,2]        q2: [1]
q2.enqueue(p1.dequeue())      %   q1: [3]          q2: [2,1]
a = q1.dequeue()              %   q1: []           q2: [2,1]
q1.enqueue(p2.dequeue())      %   q1: [1]          q2: [2]
b = q2.dequeue()              %   q1: [1]          q2: []
```

La complexité de la fonction push est alors de $\Theta(1)$ et celle de pop est de $\Theta(N)$ où N est le nombre d'élément dans la pile. Il est possible d'implémenter la pile au moyen de files de telle sorte que les complexités de push et pop soient inversées.

Cette solution est donc moins efficace que les implémentations normales d'une pile, dont la complexité est en temps constant pour push() et pop().

Nous avons également discuté en séance de la possibilité d'implémenter une file à partir de deux piles. La solution, pour laquelle la fonction push a une complexité constante $\Theta(1)$ et la fonction pop a une complexité moyenne $\Theta(1)$, est indiquée ci-dessous.

```
pile A
pile B

function enqueue(n) :
  A.push(n)

function dequeue() :
  if B.isEmpty() :
    while not A.isEmpty() :
      B.push(A.pop())
  return B.pop()
```

Question 7 : Gil De Grove, Romain Henneton

Fonction d'écriture :

```

public static int WriteInFile(String filePath, String toWrite)
  {

    Path file = Paths.get(filePath);
    if (! file.toFile().exists())
    {
      try {
        Files.createFile(file);
      } catch (IOException e) {
```

```
        e.printStackTrace();
      }

    }
    if (! file.toFile().canWrite()){
      return -1;
    }

    try {
      Files.write(file ,toWrite.getBytes("US-ASCII"));
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return 0;

  }
```

Fonction de lecture :

```
public static List<String> ReadFromFile(String filePath)
  {
    List<String> text = new ArrayList<String>();
    Path file = Paths.get(filePath);
    if(! file.toFile().exists())
    {
      return text;
    }

    try {
      text = Files.readAllLines(file , Charset.forName("US_ASCII"));
    } catch (IOException e) {
      e.printStackTrace();
    }
    return text;
  }

```