Tarea programada dos

Primera Etapa

Documentación externa

Alexa Duarte Roldán

B42312

Ciencias de la Computación e Informática

Universidad de Costa Rica

II Semestre

-2016-

**2. Etapa 1: 5 %**

En la primera etapa de esta tarea programada el estudiante se enfocara´ en:

1. Crear la infraestructura para toda la tarea.
2. Implementar los modelos Lista, Cola y Pila utilizando diversas estructuras de datos.
3. Implementar varios algoritmos de ordenamiento para su aplicación en listas.

**Requerimientos**

Cada estudiante debería implementar una librería en Java, implementando cada uno de los modelos requeridos mediante las estructuras de datos especificadas. Para una correcta implementación, todo el código referente a sus modelos y estructuras de datos debe estar aislado de cualquier elemento de prueba o de interfaz de usuario. En este caso, sus clases para pruebas deben utilizar las clases implementadas para sus modelos, y debe obtener la información necesaria para el despliegue a partir de los operadores que ofrece cada modelo. Para esto, las clases relativas a los modelos y estructuras de datos deben estar en el paquete ucr.ac.cr.ecci.ci1221.util.collections. El código de sus pruebas debe estar en ucr.ac.cr.ecci.ci1221.tests. Si desarrollara una interfaz de usuario, esta debe estar en el paquete ucr.ac.cr.ecci.ci1221.ui.

Para esta etapa su librería debe tener los siguientes elementos:

1. Implementar los modelos Lista, Cola y Pila utilizando las siguientes estructuras de datos:
2. List
3. ArrayListLinkedList
4. DoubleLinkedList
5. Stack
6. LinkedListStack
7. ArrayStack
8. InvertedArrayStack
9. Queue
10. LinkedListQueue
11. ArrayQueue
12. CircularArrayQueue

Cada modelo debe estar en un paquete separado bajo el paquete ucr.ac.cr.ecci.ci1221.util.collections, según el nombre del modelo: list, stack, queue.

Una clase SearchAlgorithms que le permite al usuario buscar un elemento en una lista utilizando búsqueda lineal o búsqueda binaria. Estos deben ser implementados mediante métodos estáticos y de forma genérica

Una clase SortingAlgorithms que le permite al usuario ordenar de forma ascendente una lista de elementos utilizando los siguientes algoritmos, con una complejidad espacial constante o O(n) en el caso de mergeSort:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. QuickSort (escogencia del pivote libre)
5. MergeSort (top-down)

Estos deben ser implementados mediante métodos estáticos y de forma genérica.

Iterar sobre los elementos de cualquier de los modelos.

Los clases implementadas para los modelos requeridos deberán implementar las interfaces pro- vistas por el profesor del curso. Cada uno de los operadores especificados deberán ser implemen- tados. Si una estructura tiene un operador faltante, esta no se revisara´.

Implementar al menos un caso de prueba completo para cada uno de las estructuras de da- tos implementadas en las que se corrobore el correcto funcionamiento de todos los operadores implementados.

**Entrega**

La etapa 1 deberá ser entregada a más tardar el 30 de setiembre de 2016. La documentación externa deberá ser entregada a más tardar al inicio de la clase de ese día (3:05pm a más tardar). El archivo comprimido deberá subirse a Mediación Virtual a más tardar a las 12pm del mismo día.

1. Errores y correcciones de la etapa anterior.
2. Diseño de clases.
3. Análisis teórico espacial y temporal de las estructuras de datos implementadas.

|  |  |  |  |
| --- | --- | --- | --- |
| Método | ArrayList | LinkedList | DoubleLinkedList |
| get(int) : E | O(1)  Ya que con la posición accede de inmediato al elemento. |  |  |
| Next(int) : int | O(1)  Con solo la posición devuelve el elemento en la posición + 1. |  |  |
| Previous(int) : int | O(1)  Con solo la posición devuelve el elemento en la posición - 1. |  |  |
| Set(int, E) : E | O(1)  Con solo la posición inserta el elemento donde corresponde. |  |  |
| Add(int, E) | O(n)  Se deben correr los elementos en las posiciones.  Espacial O(n).  Si se debe hacer crecer el array. |  |  |
| Add(E) : boolean | O(1).  Se ingresa elemento en la última posición.  Espacial O(n).  Si se debe hacer crecer el array. |  |  |
| Remove(int) : E | O(n)  Deben correrse los elementos después de ese elemento eliminado. |  |  |
| Iterator() : Iterator<E> | O( |  |  |

6. Diseño de los casos de los experimentos.

Para cada uno de las estructuras de datos se harán las siguientes pruebas:

* Pruebas con 6 órdenes de magnitud donde cada una:

1. Añadira.
2. Añadirá en una posición.
3. Hara get de un elemento en una posición especifica.
4. Tomara el next de una posición.
5. Tomara el previous de una posición.
6. Hara set de un elemento en una posición.
7. Eliminara un elemento de una posición
8. Iterara en la lista para mostrara después de cada cambio.

7. Análisis temporal de los algoritmos y estructuras de datos implementados utilizando casos de prueba en al menos 6 órdenes de magnitud.

8. Problemas encontrados durante la resolución de la tarea.

9. Código fuente.

package ucr.ac.cr.ecci.ci1221.util.collections.list;  
  
import ucr.ac.cr.ecci.ci1221.util.collections.Collection;  
  
import java.util.Iterator;  
  
*/\*\*  
 \* Interface for implementing the indexed List model. Do not modify.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the list.  
 \*  
 \** ***@author*** *Rodrigo A. Bartels  
 \*  
 \** ***@see*** *Collection  
 \*/*public interface List<E> extends Collection{  
  
 */\*\*  
 \* Returns the element at the specified position in this list.  
 \*  
 \** ***@param*** *index index of the element to return  
 \** ***@return*** *the element at the specified position in this list  
 \** ***@throws*** *IndexOutOfBoundsException if the index is out of range  
 \* (<tt>index &lt; 0 || index &gt;= size()</tt>)  
 \*/* E get(int index);  
  
 */\*\*  
 \* Returns the next element in the list based on the given position.  
 \*  
 \** ***@param*** *index the index used to calculate the next element.  
 \** ***@return*** *the element in the next position, null if the  
 \* index is the last element in the list.  
 \*/* E next(int index);  
  
 */\*\*  
 \* Returns the previous element in the list based on the given position.  
 \*  
 \** ***@param*** *index the index used to calculate the previous element.  
 \** ***@return*** *the element in the previous position, null if the  
 \* index is the first element in the list.  
 \*/* E previous(int index);  
  
 */\*\*  
 \* Replaces the element at the specified position in this list with the  
 \* specified element.  
 \*  
 \** ***@param*** *index index of the element to replace  
 \** ***@param*** *element element to be stored at the specified position  
 \** ***@return*** *the element previously at the specified position  
 \** ***@throws*** *UnsupportedOperationException if the <tt>set</tt> operation  
 \* is not supported by this list  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \** ***@throws*** *NullPointerException if the specified element is null and  
 \* this list does not permit null elements  
 \** ***@throws*** *IllegalArgumentException if some property of the specified  
 \* element prevents it from being added to this list  
 \** ***@throws*** *IndexOutOfBoundsException if the index is out of range  
 \* (<tt>index &lt; 0 || index &gt;= size()</tt>)  
 \*/* E set(int index, E element);  
  
 */\*\*  
 \* Inserts the specified element at the specified position in this list.  
 \* Shifts the element currently at that position  
 \* (if any) and any subsequent elements to the right (adds one to their  
 \* indices).  
 \*  
 \** ***@param*** *index index at which the specified element is to be inserted  
 \** ***@param*** *element element to be inserted  
 \** ***@throws*** *UnsupportedOperationException if the <tt>add</tt> operation  
 \* is not supported by this list  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \** ***@throws*** *NullPointerException if the specified element is null and  
 \* this list does not permit null elements  
 \** ***@throws*** *IllegalArgumentException if some property of the specified  
 \* element prevents it from being added to this list  
 \** ***@throws*** *IndexOutOfBoundsException if the index is out of range  
 \* (<tt>index &lt; 0 || index &gt; size()</tt>)  
 \*/* void add(int index, E element);  
  
 */\*\*  
 \* Appends the specified element to the end of this list.  
 \*  
 \** ***@param*** *e element to be appended to this list  
 \** ***@return*** *<tt>true</tt> (as specified by {****@link*** *java.util.Collection#add})  
 \** ***@throws*** *UnsupportedOperationException if the <tt>add</tt> operation  
 \* is not supported by this list  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \** ***@throws*** *NullPointerException if the specified element is null and this  
 \* list does not permit null elements  
 \** ***@throws*** *IllegalArgumentException if some property of this element  
 \* prevents it from being added to this list  
 \*/* boolean add(E e);  
  
 */\*\*  
 \* Removes the element at the specified position in this list.  
 \* Shifts any subsequent elements to the left (subtracts one  
 \* from their indices). Returns the element that was removed from the  
 \* list.  
 \*  
 \** ***@param*** *index the index of the element to be removed  
 \** ***@return*** *the element previously at the specified position  
 \** ***@throws*** *UnsupportedOperationException if the <tt>remove</tt> operation  
 \* is not supported by this list  
 \** ***@throws*** *IndexOutOfBoundsException if the index is out of range  
 \* (<tt>index &lt; 0 || index &gt;= size()</tt>)  
 \*/* E remove(int index);  
  
 */\*\*  
 \* Returns an iterator over elements of type {****@code*** *T}.  
 \*  
 \** ***@return*** *an Iterator.  
 \*/* Iterator<E> iterator();  
}

package ucr.ac.cr.ecci.ci1221.util.collections.list;  
  
import java.util.Iterator;  
  
*/\*\*  
 \* Array based implementation of the {****@link*** *List} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the list.  
 \** ***@author*** *Alexa Duarte  
 \*/*public class ArrayList<E> implements List<E> {  
 private E[] array;  
 private int size = 10;  
 private int N = 0;  
  
 public ArrayList() {  
 array = (E[]) new Object[size];  
 }  
  
 @Override  
 public E get(int index) {  
 if (index <= array.length) {  
 return array[index - 1];  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public E next(int index) {  
 if (index < N && index < array.length) {  
 return array[index];  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public E previous(int index) {  
 if (index - 1 >= 1 && index <= N) {  
 return array[index - 1];  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public E set(int index, E element) {  
 E prevElement = array[index - 1];  
  
 if (index <= array.length) {  
 array[index - 1] = element;  
 }  
 return prevElement;  
 }  
  
 @Override  
 public void add(int index, E element) {  
  
 if (index <= N + 1) {  
  
 if (index < array.length && N < array.length) {  
 if (array[index - 1] != null) {  
 for (int i = N; i > index - 1; --i) {  
 array[i] = array[i - 1];  
 }  
 }  
 } else {  
 resizeArray(index);  
 }  
  
 array[index - 1] = element;  
 ++N;  
 }else {  
 throw new IndexOutOfBoundsException();  
 }  
 }  
  
 */\*\*  
 \* Resize the array.  
 \*  
 \** ***@param*** *index the index of element to be add.  
 \*/* void resizeArray(int index) {  
 size = size+ 10;  
 E[] newArray = (E[]) new Object[size];  
 int i = 0;  
 int j = i;  
 while (i < N) {  
 if (i != index - 1) {  
 newArray[j] = array[i];  
 } else {  
 j = j + 1;  
 newArray[j] = array[i];  
 }  
 ++i;  
 ++j;  
 }  
 array = newArray;  
 }  
  
 @Override  
 public boolean add(E e) {  
 if (N <= array.length) {  
 array[N] = e;  
 ++N;  
 return true;  
 } else {  
 return false;  
 }  
 }  
  
 @Override  
 public E remove(int index) {  
 if (!isEmpty()) {  
 int position = index - 1;  
 E oldElement = array[position];  
  
 while (index <= N) {  
 array[position] = array[index];  
 position = index;  
 ++index;  
 }  
 array[position] = null;  
 --N;  
 return oldElement;  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
 @Override  
 public void clear() {  
 for(int i = 0; i < N; ++i)  
 {  
 array[i] = null;  
 }  
 N = 0;  
 }  
  
 @Override  
 public Iterator<E> iterator() {  
 return new ArrayListIterator();  
 }  
  
 private class ArrayListIterator implements Iterator<E>{  
 private int currentP = -1;  
  
 public boolean hasNext()  
 {  
 return N >= 0 && currentP < N - 1;  
 }  
  
 public E next() {  
 if (N > 0) {  
 return array[++currentP];  
 }  
 else {  
 return null;  
 }  
 }  
 }  
}

package ucr.ac.cr.ecci.ci1221.util.collections.list;  
  
import java.util.Iterator;  
  
*/\*\*  
 \* Doubly Linked pointer based implementation of the {****@link*** *List} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the list.  
 \** ***@author*** *Student Name  
 \*/*public class DoubleLinkedList<E> implements List<E> {  
 private Node head;  
 private Node tail;  
 private int nElements = 0;  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public E get(int index) {  
 if(!isEmpty() && index <= nElements) {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next() != null && cont != index - 1) {  
 node = node.next();  
 ++cont;  
 }  
 return node.getElement();  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public E next(int index) {  
 if(!isEmpty()) {  
 Node node = head;  
 int cont = 1;  
  
 while (node != null && cont != index)  
 {  
 node = node.next();  
 ++cont;  
 }  
 if(node != null) {  
 return node.getElement();  
 }  
 else  
 {  
 return null;  
 }  
 }  
 else  
 {  
 return null;  
 }  
 }  
  
 @Override  
 public E previous(int index) {  
 if (!isEmpty())  
 {  
 Node node = head;  
 int cont = 1;  
 while (node.next() != null && cont != index)  
 {  
 node = node.next();  
 ++cont;  
 }  
  
 if (node != head)  
 {  
 return node.prev().getElement();  
 }  
 else{  
 return null;  
 }  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public E set(int index, E element) {  
 if (!isEmpty()) {  
 Node node = head;  
 int cont = 1;  
 E oldElement;  
 while (node.next() != null && cont != index) {  
 node = node.next();  
 ++cont;  
 }  
 oldElement = node.getElement();  
 node.setElement(element);  
 return oldElement;  
 }else {  
 return null;  
 }  
 }  
  
 @Override  
 public void add(int index, E element) {  
 if (index <= nElements + 1) {  
 Node newNode = new Node(element);  
  
 if (!isEmpty() && index != 1) {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next() != null && cont != index - 1) {  
 node = node.next();  
 ++cont;  
 }  
 newNode.setPrev(node.prev());  
 newNode.setNext(node);  
 node.prev().setNext(newNode);  
 node.setPrev(newNode);  
 } else {  
 if(head != null) {  
 newNode.setNext(head.next());  
 newNode.setPrev(head);  
 }  
 head = newNode;  
 }  
 ++nElements;  
 }  
 else{  
 System.*out*.print("Invalid position");  
 }  
 }  
  
  
 @Override  
 public boolean add(E e) {  
 Node node = new Node(e);  
 if(!isEmpty()) {  
 Node tmp = head;  
  
 while (tmp.next() != null) {  
 tmp = tmp.next();  
 }  
  
 tmp.setNext(node);  
 node.setPrev(tmp);  
 }  
 else  
 {  
 head = node;  
 }  
 ++nElements;  
 return false;  
 }  
  
 @Override  
 public E remove(int index) {  
 if(!isEmpty() && index <= nElements) {  
 E oldElement;  
  
 if (index - 1 != 0) {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next() != null && cont != index - 1) {  
 node = node.next();  
 ++cont;  
 }  
 oldElement = node.getElement();  
 node.prev().setNext(node.next());  
 node.next().setPrev(node.prev());  
 --nElements;  
  
 return oldElement;  
 }  
 else {  
 oldElement = head.getElement();  
 head = head.next();  
 --nElements;  
 return oldElement;  
 }  
  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return nElements;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return nElements == 0;  
 }  
  
 @Override  
 public Iterator<E> iterator() {  
 return new DoubleLinkedListIterator();  
 }  
  
 @Override  
 public void clear() {  
 if(!isEmpty()) {  
 if (nElements != 1) {  
 Node node = head;  
  
 while (node != null) {  
 node.clearElement();  
 node = node.next();  
 --nElements;  
 }  
 } else {  
 head.clearElement();  
 --nElements;  
 }  
  
 }  
 }  
  
 private class Node{  
 private E element;  
 private Node next;  
 private Node prev;  
  
 public Node(E element) {  
 this.element = element;  
 this.next = null;  
 this.prev = null;  
 }  
  
 public Node(Node node)  
 {  
 this.next = node;  
 }  
  
 public void clearElement() {  
 this.element = null;  
 }  
  
 public void setNext(Node node)  
 {  
 this.next = node;  
 }  
  
 public void setPrev(Node node)  
 {  
 this.prev = node;  
 }  
  
 public void setElement(E element)  
 {  
 this.element = element;  
 }  
  
 public Node next()  
 {  
 return this.next;  
 }  
  
 public Node prev()  
 {  
 return this.prev;  
 }  
  
 public E getElement()  
 {  
 return this.element;  
 }  
 }  
  
 private class DoubleLinkedListIterator implements Iterator<E>{  
 private Node currentNode = new Node(head);  
  
 @Override  
 public boolean hasNext() {  
 if(!isEmpty()) {  
 return currentNode.next() != null;  
 }  
 else {  
 return false;  
 }  
 }  
  
 @Override  
 public E next() {  
 if(!isEmpty()) {  
 currentNode = currentNode.next();  
 return currentNode.getElement();  
 }  
 else {  
 return null;  
 }  
 }  
 }  
}

package ucr.ac.cr.ecci.ci1221.util.collections.list;  
  
import java.util.Iterator;  
  
*/\*\*  
 \* Linked pointer based implementation of the {****@link*** *List} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the list.  
 \** ***@author*** *Student Name  
 \*/*public class LinkedList<E> implements List<E> {  
 private Node head;  
 private int nElements = 0;  
  
 public LinkedList() {  
 this.head = null;  
 }  
  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public E get(int index) {  
 if(!isEmpty() && index <= nElements) {  
 if(index - 1 != 0)  
 {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next != null && cont != index - 1) {  
 node = node.next;  
 ++cont;  
 }  
  
 return node.element;  
 }  
 else  
 {  
 return head.element;  
 }  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public E next(int index) {  
 if (index <= nElements && !(isEmpty())) {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next != null && cont != index - 1) {  
 node = node.next;  
 ++cont;  
 }  
 return node.element;  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public E previous(int index) {  
 if (index <= nElements && !(isEmpty())) {  
 Node node = head;  
 int cont = 1;  
  
 while (node.next != null && cont != index - 1) {  
 node = node.next;  
 ++cont;  
 }  
 return node.element;  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public E set(int index, E element) {  
  
 Node node = head;  
 int cont = 0;  
 E oldElement = null;  
  
 while (node.next != null && cont != index - 1) {  
 node = node.next;  
 ++cont;  
 }  
  
 if (node != null) {  
 oldElement = node.element;  
 node.element = (element);  
 }  
  
 return oldElement;  
 }  
  
 @Override  
 public void add(int index, E element) {  
 if (index <= nElements + 1) {  
 Node newNode = new Node(element);  
  
 if (!isEmpty()) {  
 Node node = head;  
 int cont = 0;  
 while (node.next != null && cont != index - 1) {  
 node = node.next;  
 ++cont;  
 }  
 newNode.next = node;  
  
 node = head;  
 cont = 0;  
 while (node.next != null && cont != (index - 1) - 1) {  
 node = node.next;  
 ++cont;  
 }  
 node.next = newNode;  
  
 } else {  
 head = newNode;  
 }  
 ++nElements;  
 }  
 }  
  
  
  
 @Override  
 public boolean add(E e) {  
 Node newNode = new Node(e);  
 Node node = head;  
  
 while (node.next != null)  
 {  
 node = node.next;  
 }  
 node.next = newNode;  
 ++nElements;  
 return true;  
 }  
  
 @Override  
 public E remove(int index) {  
 if(!isEmpty() && index <= nElements) {  
 E oldElement;  
  
 if(index - 1 != 0 ) {  
 Node node = head;  
 int cont = 0;  
  
 while (node.next != null && (index - 1) - 1 != cont) {  
 node = node.next;  
 ++cont;  
 }  
 oldElement = node.next.element;  
 node.next = node.next.next;  
 }  
 else {  
 oldElement = head.element;  
 head = head.next;  
 }  
 --nElements;  
 return oldElement;  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return nElements;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return nElements == 0;  
 }  
  
 @Override  
 public Iterator<E> iterator() {  
 return new LinkedListIterator();  
 }  
  
 @Override  
 public void clear() {  
 if (!isEmpty()) {  
 if(nElements != 1) {  
 Node node = head;  
  
 while (node != null) {  
 node.element = null;  
 node = node.next;  
 }  
 }else{  
 head.element = null;  
 }  
 nElements = 0;  
 }  
 }  
  
 private class Node{  
 private E element;  
 private Node next;  
  
 public Node( E element)  
 {  
 this.element = element;  
 this.next = null;  
 }  
 public Node(Node node)  
 {  
 this.next = node;  
 }  
 }  
  
 private class LinkedListIterator implements Iterator<E>  
 {  
 private Node currentNode = new Node(head);  
  
 @Override  
 public boolean hasNext() {  
 if(!isEmpty()) {  
 return currentNode.next != null;  
 }  
 else {  
 return false;  
 }  
 }  
  
 @Override  
 public E next() {  
 if(!isEmpty()) {  
 currentNode = currentNode.next;  
 return currentNode.element;  
 }  
 else {  
 return null;  
 }  
 }  
 }  
}

Queue

package ucr.ac.cr.ecci.ci1221.util.collections.queue;  
  
import ucr.ac.cr.ecci.ci1221.util.collections.Collection;  
  
*/\*\*  
 \* Interface for all the implementations of the Queue model. Do not modify.  
 \*  
 \** ***@author*** *Rodrigo A. Bartels  
 \*/*public interface Queue<E> extends Collection{  
  
 */\*\*  
 \* Inserts the specified element into this queue.  
 \*  
 \** ***@param*** *element the element to add  
 \*/* void enqueue(E element);  
  
 */\*\*  
 \* Retrieves and removes the head of this queue,  
 \* or returns {****@code*** *null} if this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if this queue is empty  
 \*/* E dequeue();  
  
 */\*\*  
 \* Retrieves, but does not remove, the head of this queue,  
 \* or returns {****@code*** *null} if this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if this queue is empty  
 \*/* E peek();  
}

package ucr.ac.cr.ecci.ci1221.util.collections.queue;  
  
*/\*\*  
 \* Array based implementation of the {****@link*** *Queue} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the queue.  
 \** ***@author*** *Student Name  
 \*/*public class ArrayQueue<E> implements Queue<E> {  
 private E [] array;  
 private int size = 10;  
 private int N = 0;  
  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* public ArrayQueue()  
 {  
 array = (E[]) new Object[size];  
 }  
  
 @Override  
 public void enqueue(E element) {  
 if(N >= array.length) {  
 resizeArray();  
 }  
 array[N] = element;  
 ++N;  
 }  
  
 public void print()  
 {  
 for(int i = 0; i < N; ++i)  
 {  
 System.*out*.print(array[i] + " ");  
 }  
 }  
  
 private void resizeArray()  
 {  
 size = size +10;  
 E[] newArray = (E[]) new Object[size];  
  
 for(int i = 0; i < N; ++i)  
 {  
 newArray[i] = array[i];  
 }  
 array = newArray;  
 }  
  
 @Override  
 public E dequeue() {  
 if(!isEmpty()) {  
 E oldElement = array[0];  
 for (int i = 0; i < N; ++i) {  
 array[i] = array[i + 1];  
 }  
 --N;  
 return oldElement;  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public E peek() {  
 if(!isEmpty())  
 {  
 return array[0];  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
 @Override  
 public void clear() {  
 for(int i = 0; i<N; ++i){  
 array[i] = null;  
 }  
 }  
}

package ucr.ac.cr.ecci.ci1221.util.collections.queue;  
  
*/\*\*  
 \* Linked based implementation of the {****@link*** *Queue} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the queue.  
 \** ***@author*** *Student Name  
 \*/*public class LinkedListQueue<E> implements Queue<E> {  
 private Node head = null;  
 private int nElements = 0;  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public void enqueue(E element) {  
 if(!isEmpty())  
 {  
 Node node = head;  
  
 while (node.next() != null)  
 {  
 node = node.next();  
 }  
  
 node.setNext(new Node(element));  
 }  
 else  
 {  
 head = new Node(element);  
 }  
 ++nElements;  
 }  
  
 @Override  
 public E dequeue() {  
 if (!isEmpty()) {  
 E oldElement = head.getElement();  
 head = head.next();  
 --nElements;  
 return oldElement;  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public E peek() {  
 if(!isEmpty())  
 {  
 return head.getElement();  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return nElements;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return nElements == 0;  
 }  
  
 @Override  
 public void clear() {  
 if(!isEmpty())  
 {  
 Node node = head;  
  
 while(node != null)  
 {  
 node.setElement(null);  
 --nElements;  
 }  
 }  
 }  
  
 public void print(){  
 Node node = head;  
 while(node != null)  
 {  
 System.*out*.print(node.getElement()+ " ");  
 node = node.next();  
 }  
 }  
 private class Node{  
 private E element;  
 private Node next;  
  
 public Node(E element)  
 {  
 this.element = element;  
 this.next = next;  
 }  
  
 public void setNext(Node next){  
 this.next = next;  
 }  
  
 public Node next() {  
 return this.next;  
 }  
  
 public void setElement(E element)  
 {  
 this.element = element;  
 }  
 public E getElement()  
 {  
 return this.element;  
 }  
 }  
}

Stack

package ucr.ac.cr.ecci.ci1221.util.collections.stack;  
  
import ucr.ac.cr.ecci.ci1221.util.collections.Collection;  
  
*/\*\*  
 \* Interface for all the implementations of the Stack model. The <code>Stack</code> interface  
 \* represents a last-in-first-out (LIFO) stack of objects. Do not modify.  
 \*  
 \** ***@author*** *Rodrigo A. Bartels  
 \*/*public interface Stack<E> extends Collection{  
  
 */\*\*  
 \* Pushes an item onto the top of this stack.  
 \*  
 \** ***@param*** *element the item to be pushed onto this stack.  
 \** ***@return*** *the <code>item</code> argument.  
 \*/* void push(E element);  
  
 */\*\*  
 \* Removes the object at the top of this stack and returns that  
 \* object as the value of this function.  
 \*  
 \** ***@return*** *The object at the top of this stack or null if this stack is empty.  
 \*/* E pop();  
  
 */\*\*  
 \* Looks at the object at the top of this stack without removing it  
 \*  
 \*  
 \* from the stack.  
 \*  
 \** ***@return*** *the object at the top of this stack or null if this stack is empty.  
 \*/* E peek();  
}

package ucr.ac.cr.ecci.ci1221.util.collections.stack;  
  
*/\*\*  
 \* Array based implementation of the {****@link*** *Stack} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the stack.  
 \** ***@author*** *Rodrigo A. Bartels  
 \*/*public class ArrayStack<E> implements Stack<E>{  
 private int size = 10;  
 private E[] array = (E[]) new Object[size];  
 private int N = 0;  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public void push(E element) {  
 if(N < array.length) {  
 array[N] = element;  
  
 }  
 else{  
 resizeArray();  
 }  
 ++N;  
 }  
  
 private void resizeArray()  
 {  
 size =+10;  
 E[] newArray = (E[]) new Object[size];  
  
 for(int i = 0; i< N; ++i) {  
 newArray[i] = array[i];  
 }  
 array = newArray;  
 }  
  
 @Override  
 public E pop() {  
 if(!isEmpty()) {  
 E element = array[N-1];  
 array[N-1] = null;  
 --N;  
 return element;  
 }  
 else{  
 return null;  
 }  
 }  
  
 @Override  
 public E peek() {  
 if(!isEmpty()) {  
 return array[N - 1];  
 }  
 else{  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
 @Override  
 public void clear() {  
 for(int i = 0; i < N; ++i)  
 {  
 array[i] = null;  
 }  
 N = 0;  
 }  
  
 public void print()  
 {  
 for(int i = 0; i < N; ++i)  
 {  
 System.*out*.print(array[i]+ " ");  
 }  
 }  
}

package ucr.ac.cr.ecci.ci1221.util.collections.stack;  
  
*/\*\*  
 \* Array based implementation of the {****@link*** *Stack} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the stack.  
 \** ***@author*** *Rodrigo A. Bartels  
 \*/*public class InvertedArrayStack<E> implements Stack<E> {  
 private E[] array = (E[]) new Object[10];  
 private int N = 0;  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public void push(E element) {  
 if(!isEmpty()) {  
 if (N <= array.length) {  
 for (int i = N - 1; i >= 0; --i) {  
 array[i + 1] = array[i];  
 }  
 } else {  
 resizeArray();  
 }  
 }  
  
 ++N;  
 array[0] = element;  
 }  
  
 private void resizeArray()  
 {  
 E[] newArray = (E[]) new Object[array.length + 10];  
 for(int i = 1; i < N; ++i)  
 {  
 newArray[i] = array[i-1];  
 }  
 array = newArray;  
 }  
 @Override  
 public E pop() {  
 if(!isEmpty()) {  
 E element = array[0];  
  
 for (int i = 0; i < N; ++i) {  
 array[i] = array[i + 1];  
 }  
 --N;  
 return element;  
 }  
 else {  
 return null;  
 }  
  
 }  
  
 @Override  
 public E peek() {  
 if(!isEmpty()) {  
 return array[0];  
 }  
 else {  
 return null;  
 }  
 }  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
 @Override  
 public void clear() {  
 for(int i = 0; i < N; ++i)  
 {  
 array[i] = null;  
 }  
 N = 0;  
 }  
  
 public void print() {  
 for (int i = 0; i < N; ++i) {  
 System.*out*.print(array[i] + " ");  
 }  
 }  
}

package ucr.ac.cr.ecci.ci1221.util.collections.stack;  
  
*/\*\*  
 \* Array based implementation of the {****@link*** *Stack} model.  
 \*  
 \** ***@****TODO Complete operators implementations, javadoc and any missing code.  
 \*  
 \** ***@param*** <*E*> *the type of elements in the stack.  
 \** ***@author*** *Rodrigo A. Bartels  
 \*/*public class LinkedListStack<E> implements Stack<E> {  
 private Node head = null;  
 private int N = 0;  
 */\*\*  
 \** ***@****TODO Add missing attributes and private methods and classes.  
 \*/* @Override  
 public void push(E element) {  
 head = new Node(element, head);  
 ++N;  
 }  
  
 @Override  
 public E pop() {  
 if(!isEmpty()) {  
 E element = head.element;  
 head = head.next;  
 return element;  
 }  
 else{  
 return null;  
 }  
 }  
  
 @Override  
 public E peek() {  
 return head.element;  
 }  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
 @Override  
 public void clear() {  
 if(!isEmpty()) {  
 Node node = head;  
  
 while (node != null) {  
 node = node.next;  
 }  
 N = 0;  
 }  
 }  
  
 public void print()  
 {  
 if(!isEmpty()) {  
 Node node = head;  
 while (node != null) {  
 System.*out*.print(node.element);  
 node = node.next;  
 }  
 }  
 }  
  
 private class Node  
 {  
 Node next;  
 E element;  
  
 public Node(E element, Node next)  
 {  
 this.element = element;  
 this.next = next;  
 }  
 }  
}