Tema 8- Implementación de Pila, Cola y Lista con Punto de Interés

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tema 8- Implementación de Pila, Cola y Lista con Punto de Interés

Índice general:

Objetivos y Bibliografía

- Desarrollar las implementaciones más eficientes de las Estructuras de Datos lineales Pila, Cola y Lista con Punto de Interés.
  - La clase ArrayPila como implementación de la interfaz Pila.
  - La clase ArrayCola como implementación de la interfaz Cola.
  - La clase LEGListaConPI como implementación de la interfaz ListaConPI.
- Bibliografía Principal:
  - Capítulo 15 del libro de M.A. Weiss: "Estructuras de Datos en Java". Adisson-Wesley, 2000.

Representación Eficaz de una EDA Lineal

- Las implementaciones de Pila, Cola y Lista deben permitir que sus operaciones básicas se ejecuten en **tiempo constante**.
  - Tiempo constante: Independiente del número de elementos de la EDA.
- Así, los recorridos y búsquedas simples tendrán coste temporal acotado por el número de elementos de la EDA:
  - Lineal (o proporcional) con el número de elementos de la EDA.
- La implementación de los métodos depende de la representación de la colección de datos:
  - Representación contigua: Array
  - Representación enlazada: Lista Enlazada Genérica

Claves para implementar una EDA lineal

1. Inicialmente, representamos los datos de la EDA sobre un array de CAPACIDAD_POR_DEFECTO componentes.
2. Si la EDA requiere alguna forma de acceso especial (por ejemplo FIFO o LIFO), definir nuevos atributos que los representen para implementar el acceso eficazmente.
3. Analizar el coste de cada una de las operaciones de la interfaz implementada.
   - Si alguna operación requiere desplazamiento de datos en la representación interna, se descarta la representación como un array.
4. Ensayar una implementación del interfaz con una representación basada en Lista Enlazada.

Implementación de Pila: La Clase ArrayPila

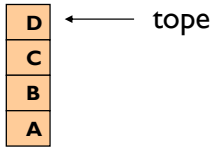
- Una *Pila* es una colección homogénea de datos que sólo se puede gestionar accediendo secuencialmente al dato que ocupa el Punto de Interés, siguiendo un criterio LIFO (**L**ast **I**n **F**irst **O**ut), esto es, accediendo al dato que ocupa el *tope* de la pila, es decir, el último que se insertó.

```
public interface Pila<E> {
        void apilar(E x);
        E desapilar();
        E tope();
        boolean esVacia();
    }
```

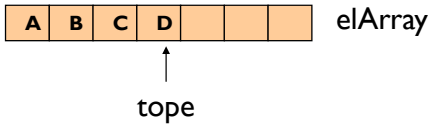
- La clase **ArrayPila** implementa el interfaz Pila.
  - Atributos principales:
    - Un array para almacenar los elementos de la Pila.
    - Una capacidad inicial para el vector.
    - Un marcador al tope de la pila (la posición en el vector del último elemento insertado), inicialmente valdrá -1.

Esquema de Implementación de Pila

- Pila en la que se han insertado, por este órden, los elementos A, B, C y D
- Visión desde el punto de vista del modelo:



- Punto de vista de la implementación:



Clase ArrayPila (1/3)

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
public class ArrayPila<E> implements Pila<E> {
    protected E elArray[];
    protected int tope;
    protected static final int CAPACIDAD_POR_DEFECTO = 200;

  @SuppressWarnings("unchecked")
  public ArrayPila () {
        elArray= (E[]) new Object[CAPACIDAD_POR_DEFECTO];
        tope = -1;
   }
  public void apilar(E x) {
        if ( tope + 1 == elArray.length) duplicarArray();
        tope++;  elArray[tope] = x;
   }
```

¿Qué ocurre si se trata de apilar un nuevo elemento y elArray está completo?

Clase ArrayPila (2/3)

```
public E desapilar() {
  E elUltimo = elArray[tope];
  tope--;
  return elUltimo;
}

public E tope() {
  return elArray[tope];
}

public boolean esVacia() {
   return ( tope == -1 );
}
```

¿Qué ocurre si se invoca a desapilar y elArray está vacío?

Clase ArrayPila (3/3)

```
public String toString() {
        String res = "";
        for (int i = tope; i >= 0; i--) res += elArray[i] +"\n";
        return res;
    }
@SuppressWarnings("unchecked")
private void duplicarArray() {
        E nuevoArray[] = (E[]) new Object[elArray.length*2];
        for ( int i = 0; i <= tope; i++ ) nuevoArray[i] = elArray[i];
        elArray = nuevoArray;
    }}
```

• El método toString() muestra los elementos en orden inverso al que fueron insertados en la Pila, es decir, el orden en el que serían desapilados.

Sobre la Implementación de Pila

- La especificación de Pila NO indica el número máximo de elementos que puede albergar la EDA.
  - La implementación debe gestionarlo adecuadamente, de manera transparente al usuario (método duplicarArray).
- Alternativamente se podría permitir que el usuario indique el máximo número de elementos de la Pila.
  - Sin embargo, la estrategia original permite utilizar la Pila con un número de elementos no conocido a priori.

```
public ArrayPila(int n){
    elArray= (E[]) new Object[n];
    tope = -1;
}
```

¿Qué coste tienen las operaciones implementadas?

Implementación de Cola: La Clase ArrayCola

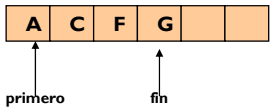
- Una *Cola* es una colección homogenea de elementos que solo permite acceder secuencialmente al dato que ocupa el punto de interés siguiendo un criterio FIFO (**F**irst **I**n **F**irst **O**ut), es decir, el primer elemento que fue insertado es el primero en ser atendido.

```
public interface Cola<E> {
        void encolar(E x);
        E desencolar();
        E primero()
        boolean esVacia();
    }
```

- La clase **ArrayCola** implementa el interfaz Cola.
  - Atributos principales:
    - Un array para almacenar los elementos de la Cola (Object).
    - Una capacidad inicial para el vector.
    - Un marcador al elemento *primero* de la Cola
    - Un marcador al *último* elemento de la Cola (*fin*)

Esquema de Implementación de Cola (1/2)

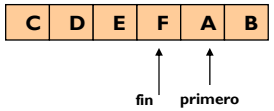
- Problemas:
  - Si se fija la posición de primero (Como en Pila) → Operación desapilar() con coste lineal por mantener contigüidad.
  - La posición más alta del vector limita la posición del último dato de la Cola.
- Solución:
  - Consideramos el array como una estructura circular.
    - No existe el final de la estructura.
    - La posición siguiente a la última es la primera.
- Implementación usando un vector e índices al principio y al final de la Cola



Esquema de Implementación de Cola (2/2)

- Dos estados del vector representando a **la misma** Cola.

- Mantenemos el tamaño de la Cola para diferenciar entre Cola vacía y Cola llena (primero es el siguiente elemento a fin).



- La circularidad se implementa mediante el método *incrementa* que nos permitirá recorrer la estructura elemento a elemento.
  - Cuando el valor del parámetro es el máximo, el siguiente vale 0, volviendo al inicio de la estructura.

La Clase ArrayCola (1/4)

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
public class ArrayCola<E> implements Cola<E> {
    protected E elArray[];
    protected int fin, primero, tallaActual;
    protected static final int CAPACIDAD_POR_DEFECTO = 200;

   @SuppressWarnings("unchecked")
    public ArrayCola(){
        elArray = (E[]) new Object[CAPACIDAD_POR_DEFECTO];
        tallaActual = 0;
        primero = 0;
        fin = -1;
    }
```


La Clase ArrayCola (2/4)

```
public void encolar(E x){
        if ( tallaActual == elArray.length ) duplicarArray();
        fin = incrementa(fin);
        elArray[fin] = x;
        tallaActual++;
    }
public E desencolar(){
        E elPrimero = elArray[primero];
        primero = incrementa(primero);
        tallaActual--;
        return elPrimero;
    }
```

La Clase ArrayCola (3/4)

```
public E primero() {
        return elArray[primero];
    }
public boolean esVacia(){
        return ( tallaActual == 0 );
    }
private int incrementa(int indice) {
        if ( ++indice == elArray.length) indice = 0;
        return indice;
    }
```


La Clase ArrayCola (4/4)

¿Cómo se implementaría el método toString?

```
public String toString() {
        String res = "";
        int aux = primero;
        for ( int i = 0; i < tallaActual; i++, aux = incrementa(aux) )
            res += elArray[aux] + " ";
        return res;
    }
@SuppressWarnings("unchecked")
private void duplicarArray() {
        E nuevo[] = (E[]) new Object[elArray.length*2];
        for (int i = 0; i < tallaActual; i++ , primero = incrementa(primero) )
            nuevo[i] = elArray[primero];
        elArray = nuevo;
        primero = 0;
        fin = tallaActual - 1;
    }}
```


Implementación de Lista Con Punto de Interés: La Clase LEGListaConPI

- Una Lista con Punto de Interés es una colección homogénea de datos que solo se puede manipular accediendo secuencialmente al dato que ocupa el punto de interés.

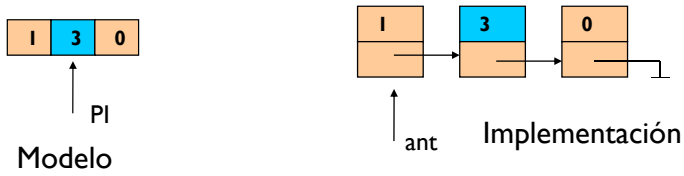
```
public interface ListaConPI<E> {
        void insertar(E x);
        void eliminar();
        void inicio();
        void fin();
        void siguiente();
        E recuperar();
        boolean esFin();
        boolean esVacia();
    }
```

- La clase LEGListaConPI implementa el interfaz ListaConPI.
  - Atributos principales:
    - Una Lista Enlazada representada por una referencia al primer objeto NodoLEG<E>.
    - Referencia al último nodo de la lista (eficiencia por la operación fin()).

Detalles de la Implementación LEGListaConPI

¿Por qué se utiliza esta estrategia?

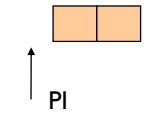
- El punto de interés se representa como una referencia al objeto NodoLEG<E> **anterior** al que debe ser accedido.



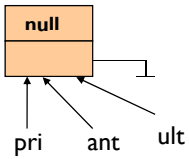
- Caso especial: El primer elemento NO tiene un elemento anterior.
  - Simulado mediante un nodo ficticio anterior al primer nodo.
  - Esta solución simplifica los métodos de inserción y borrado.
  - La referencia al primer nodo NUNCA será modificada.

Detalles de la Implementación LEGListaConPI

- Lista Vacía:

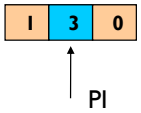


Modelo

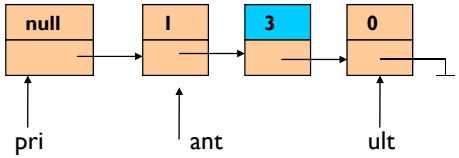


Implementación

- Lista con Elementos:



Modelo



Implementación

La Clase LEGListaConPI (1/3)

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
public class LEGListaConPI<E> implements ListaConPI<E> {
    protected NodoLEG<E> pri, ant, ult;

    public LEGListaConPI(){
        pri = ult = ant = new NodoLEG<E>(null);
    }
    public  void inicio(){ ant = pri; }
    public  void fin(){ ant = ult;}
    public  void siguiente(){ ant = ant.siguiente; }
    public  boolean esFin(){ return (ant == ult); }
    public  boolean esVacia(){ return (pri == ult); }
```


La Clase LEGListaConPI (2/3)

```
    public E recuperar(){
        return ant.siguiente.dato;
    }
    public void insertar(E x) {
        NodoLEG<E> nuevo = new NodoLEG<E>(x);
        nuevo.siguiente = ant.siguiente;
        ant.siguiente = nuevo;
        if ( ant == ult) ult = nuevo;
        ant = ant.siguiente;
    }
    public  void eliminar(){
        if ( ant.siguiente == ult ) ult = ant;
        ant.siguiente = ant.siguiente.siguiente;
    }
```

¿Qué ocurre si se invoca el método recuperar y el P.I. está situado tras el último elemento?

La Clase LEGListaConPI (3/3)

```
    public String toString() {
        String res;
        NodoLEG<E> aux = pri.siguiente;
        while ( aux != null ) {
            res += " " + aux.dato.toString();
            aux = aux.siguiente;
        }
        res += "\n";
        return res;
    }
}// Fin de la clase LEGListaConPI
```

¿Cómo se implementaría el método toString?

Implementaciones Alternativas de Pila y Cola

- Es posible implementar los modelos de Pila y de Cola aprovechando la funcionalidad ya existente en Java.
- En Java Platform SE 6 aparece la interfaz *Deque* (double ended queue) que permite:
  - Inserción y borrado de elementos tanto al principio como al final de la estructura.
  - NO permite acceso indexado a la estructura (solo en ambos extremos).
- Permite utilizar la estructura para acceso FIFO (como una Cola) y para acceso LIFO (como una Pila).
- La clase *ArrayDeque* proporciona la implementación de la interfaz, utilizando un array como mecanismo de almacenamiento.
  - Operaciones con coste constante amortizado.

La clase ArrayDequePila

```
public class ArrayDequePila<E> extends ArrayDeque<E> implements
    Pila<E>
{
 public ArrayDequePila() { super();}

 public void apilar(E x) { push(x);}
 public E desapilar() { return pop();}
 public E tope() { return peek();}
 public boolean esVacia() { return (size() == 0);}
}
```


La Clase ArrayDequeCola

```
public class ArrayDequeCola<E> extends ArrayDeque<E> implements
    Cola<E>
{
    public ArrayDequeCola() { super();}
    public void encolar(E x) {addLast(x);}
    public E desencolar() { return pollFirst();}
    public E primero() { return peekFirst();}
    public boolean esVacia() {
        return (size() == 0);
    }
}
```