Universidad Diego Portales
Escuela de Informática &
Telecomunicaciones

Estructuras de Datos & Análisis de Algoritmos

Laboratorio 2: Pilas y Colas

Autores:
Cinthya Fuentealba Bravo
Ignacia Reyes Ojeda

Profesor:
Cristian Llull Torres

30 de Septiembre de 2025

Índice

1. Introducción

En sistemas distribuidos y de alta concurrencia, como servidores web, plataformas de IoT o sistemas operativos, es fundamental procesar una gran cantidad de eventos o tareas de manera eficiente y confiable. Dado que las solicitudes entrantes no pueden ser atendidas de inmediato, se deben almacenar para ser gestionadas de forma ordenada. Este comportamiento es el principio FIFO (First-In, First-Out), perfectamente modelado por una Cola (*Queue*).

Por otro lado, la ejecución de una tareas complejas a menudo implican una secuencia de operaciones o llamadas a funciones anidadas. Para gestionar el flujo de ejecución y manejar posibles errores, el sistema debe recordar el orden en que dichas operaciones se han realizado. Este es un ejemplo clásico del principio LIFO (Last-In, First-Out), modelado por una Pila (*Stack*), la cual es crucial para la recursividad y el seguimiento de estados.

En este laboratorio se trabajó con la implementación propia de estas estructuras de datos (**PilaEnlazada** y **ColaEnlazada**) utilizando nodos enlazados en `Java`. Además, se diseñó una clase **Alerta** que simula un sistema de llamadas recursivas con cierta probabilidad de fallo, permitiendo observar de manera práctica cómo interactúan las colas, las pilas y la recursividad en el manejo de procesos.

De esta forma, el laboratorio no sólo refuerza la teoría de estructuras dinámicas, sino que también muestra su aplicación en la simulación de sistemas concurrentes.

Este informe requiere del uso de un programa informático creado en IntelliJ IDEA, este se caracteriza por ser un editor, compilador y depurador. El código informático realizado esta realizado en lenguaje Java. Por ende, dentro del informe se detallan la realización del código que busca dar solución al problema propuesto, también sobre los componentes de este código y su funcionamiento. En el repositorio de GitHub se encuentra el código Java y el archivo LATEX para su análisis.

https://github.com/Cinthya982012/Laboratorio2$_EDA$

2. Implementación

Para comenzar a dar solución al problema planteado, y teniendo en cuenta las consideraciones que se deben tener en la simulación, se inicia la creación del código informático, para ello lo primero realizado en este laboratorio es la creación de la siguiente clase:

2.1. Clase Nodo

- `Nodo<t>siguiente:` apunta al tope de la pila.
- `T valor:`

A continuación se puede observar los atributos implementados:

```
//CLASE NODO
class AnalizadorDeNotas {

    //============================================
    //ATRIBUTOS NODO
    class Nodo<T>{
    private T valor;
    private Nodo<T> siguiente;

    //============================================
```

Esta clase esta compuesta por tres constructores principalmente, estos son:

- `public Nodo()`: corresponde a un constructor vacio.
- `public Nodo(T valor)`: .
- `public Nodo(T valor, Nodo<T>siguiente)`:

```
//CLASE NODO
    //============================================
    //CONSTRUCTOR 1
    public Nodo(){
        valor = null;
        siguiente = null;
    }
    //============================================
```

```
//================================================
    //CONSTRUCTOR 2
   public Nodo(T valor){
        this.valor = valor;
        this.siguiente = null;
    }
    //============================================================
```

```
//================================================
    //CONSTRUCTOR 3
   public Nodo(T valor, Nodo<T> siguiente){
        this.valor = valor;
        this.siguiente = siguiente;
    }
    //=========================================
```

También se realizan los *setters* y *getters* correspondientes de los atributos, los cuales permiten acceder y modificar elementos fuera de la clase. Esto se puede visualizar en la siguiente representación:

```
 //=========================================
    //SETTERS Y GETTERS
    public T getValor(){return valor;
    }
    public Nodo<T> getSiguiente(){return siguiente;
    }

    public void setValor(T valor){this.valor = valor;
    }
    public void setSiguiente(Nodo<T> siguiente){this.siguiente =
        siguiente;
    }
}

//================================================
```

2.2. Clase Pila Enlazada

- `private Nodo<T>tope:` apunta al tope de la pila.
- `private int size:`

A continuación se puede observar los atributos implementados:

```
//CLASE PILA ENLAZADA
class PilaEnlazada {

    //=============================================
    //ATRIBUTOS PILA ENLAZADA
    private Nodo<T>tope;
    private int size;

    //=============================================
```

Además de los *atributos* creados, esta clase posee métodos, estos son los siguientes:

- `public void push(T valor)`: agrega un elemento al tope de la pila.

```
//================================================
//METODO QUE AGREGA UN ELEMENTO AL TOPE DE LA PILA
```

```
      public void push(T valor){
        Nodo<T>nuevo = new Nodo<>(valor, tope);
        tope = nuevo;
        size++;
    }
//=================================================
```

- `public T pop()`: Elimina y retorna el elemento del tope de la pila. Retorna null si la pila esta vacia.

```
//=============================================================
//METODO QUE ELIMINA EL TOPE Y LO RETORNA
     public T pop(){
        if(isEmpty()==true){return null;}

        T valor = tope.getValor();
        tope = tope.getSiguiente();
        size--;
        return valor;
    }
//=========================================================
```

- `public T peek()`: Retorna el elemento del tope de la pila sin eliminarlo.

```
//=================================================
//METODO QUE RETORNA EL TOPE
     public T peek(){
        if(isEmpty()==true){return null;}
        return tope.getValor();
    }
```

- `public boolean isEmpty()`: Retorna true si la pila no tiene elementos.

```
//=================================================
//METODO QUE RETORNA TRUE SI LA PILA ESTA VACIA
     public boolean isEmpty(){
     return tope == null;
     }

//=================================================
```

- `public String toString()`: Retorna true si la pila no contiene elementos.

```
    //=================================================
//METODO QUE RETORNA TRUE SI LA PILA ESTA VACIA
     public boolean isEmpty(){
     return tope == null;
     }
 public String toString(){
        String string =" ";
        Nodo<T>actual = tope; //Comienza desde el inicio
        while(actual!=null){ //Recorre la pila hasta el final
            string += actual.getSiguiente(); //Suma el valor +
               un espacio
```

```
            actual = actual.getSiguiente();//Devuelve el string
        }
        return string;
    }
//=================================================
```

- `public int size()`: Retorna el tamaño de la pila.

```
//=================================================
//METODO QUE RETORNA EL TAMA O DE LA PILA
     public int size(){
     return size;
     }
        }
//=================================================
```

- `public void vaciar()`: Elimina todos los elementos de la cola.

```
//=================================================
//METODO QUE VACIA LA PILA
     public void vaciar(){
        while(!pila.isEmpty()){
        pila.pop();
        }
    }
//=================================================
```

2.3. Clase Cola Enlazada

- `Nodo<T>inicio`: Apunta al nodo del inicio de la cola.
- `Nodo<T>fin`: Apunta al nodo del final de la cola.
- `int size`:

A continuación se puede observar los atributos implementados:

```
//CLASE COLA ENLAZADA
 private Nodo<T>inicio;
    private Nodo<T>fin;
    private int size;
    //=============================================
```

Además de los *atributos* creados, esta clase posee métodos, estos son los siguientes:

- `public boolean add(T valor)`: Agrega un elemento al final de la cola. Retorna true en caso de lograrlo.

```
//=================================================
//METODO QUE
       public boolean add(T valor){
        Nodo<T>nuevo = new Nodo<>(valor);//Creaci n de un
           nodo nuevo
```

```
5
6           if(inicio==null){//Si la cola esta vacia
7               inicio = nuevo;
8               fin = nuevo;
9           }
10          else{//Si ya existen elementos
11              fin.setSiguiente(nuevo);//El ultimo nodo apunta al
                    nuevo
12              fin = nuevo;
13          }
14          size++;//Aumentar el tama o
15          return true;//Devolver true (Proceso exitoso)
16      }
17  //=================================================
```

- `public T poll()`: Elimina y retorna el primer nodo de la cola. Retorna null si la cola esta vacia.

```
1   //=============================================================
2   //METODO QUE ELIMINA EL INICIO Y LO RETORNA
3       public T poll(){
4           if(inicio==null){return null;}//Si la cola esta vacia.
                No hay nodo inicial, no hay nada por quitar.
5           T valor = inicio.getValor();//Guarda el valor del nodo
                inicial
6           inicio = inicio.getSiguiente();//Avanza el puntero de
                inicio al siguiente nodo
7           size--;//Disminuye el tama o de la cola
8           return valor;//Retorna el valor que se quito
9       }
10
11  //=============================================================
```

- `public T peek()`: Retorna el primer nodo de la cola sin eliminar.

```
1   //=============================================
2   //METODO QUE RETORNA EL INICIAL
3       public T peek(){
4           if(isEmpty()==true){return null;}
5           return tope.getValor();
6       }
```

- `public boolean isEmpty()`: Retorna true si la cola no tiene elementos.

```
1   //=================================================
2   //METODO QUE RETORNA TRUE SI LA COLA ESTA VACIA
3       public boolean isEmpty(){
4           if(inicio==null){return true;}//Si la cola esta vacia
                retorna true
5           else{return false;}//Si la cola no esta vacia retorna
                false
6       }
7
```

```
8 //================================================
```

2.4. Clase Alerta

- `String nombre:`
- `double probFallo:`
- `int limiteAlertas:`
- `Random rng:`

A continuación se puede observar los atributos implementados:

```
1 //CLASE ALARMA
2  private String nombre;
3     private double probFallo;
4     private int limiteAlertas;
5     private Random rng;
6     //================================================
```

También se realizan los *setters* y *getters*. Esto se puede visualizar en la siguiente representación:

```
1  //================================================
2     //SETTERS Y GETTERS
3      public String getNombre(){
4      return nombre;
5      }
6     String setNombre(){
7     return nombre;
8     }
9
10 //================================================
```

Esta clase esta compuesta por dos constructores, estos son:

- `public Alerta(String nombre, double probFallo):`
- `public Alerta(String nombre, double probFallo, Random rng):`

```
1 //CLASE ALERTA
2     //================================================
3     //CONSTRUCTOR 1
4     public Alerta(String nombre, double probFallo){
5         this.nombre = nombre;
6         this.probFallo = probFallo;
7     }
8
9     //================================================
```

```
1 //================================================
2     //CONSTRUCTOR 2
```

```
3   public Alerta(String nombre, double probFallo, Random rng){
4         this.nombre = nombre;
5         this.probFallo = probFallo;
6         this.rng = rng;
7     }
8     //=============================================================================
```

Además de los *atributos*, *setters*, *getters* y *constructores* creados, esta clase posee métodos, estos son los siguientes:

- `public boolean procesarLlamada(PilaEnlazada<Alerta>pila, int limite)`: Retorna true si se alcanzo el limite de la cola, si no, genera un numero aleatorio y comprueba su funcionamiento, si falla retornando false o de caso contrario crea una nueva alerta.

```
1  //=================================================
2  //METODO QUE COMPRUEBA FUNCIONAMIENTO
3   public boolean procesarLlamada(PilaEnlazada<Alerta>pila, int
       limite){
4           //Stack<Alerta>pilaN= new Stack<>();//Pila nueva
5           /**Caso Base (Alcanzar la profundidad maxima)*/
6           if(pila.size()>=limite){return true;}//Si el tama o
               de la pila llego al limite, ya se alcanzo la
               profundidad maxima y retorna true
7
8           /**Caso 1: Posible falla aleatoria*/
9           double fallo=rng.nextDouble();//Se genera un n mero
               aleatorio entre 0 y 1
10          if(fallo <this.probFallo){return false;}//Si el fallo
               es menor que la probabilidad de fallo. La alerta
               falla y retorna false.
11
12          /**Caso 2: Recursivo (generar la siguiente alerta y
               continuar)*/
13          String idNueva = "Alerta Nivel" + pila.size();
14          Alerta siguiente = new Alerta(idNueva, probFallo,
               rng); //Se crea una nueva alerta
15          pila.push(siguiente);//Se mete a la pila
16          return siguiente.procesarLlamada(pila, limite);//Se
               llama recursivamente el metodo procesarLlamado
17      }
18
19 //=================================================
```

- `public String mostrarAlerta()`: Retorna el nombre con la probabilidad de fallo.

```
1  //=============================================================
2  //METODO QUE RETORNA EL NOMBRE
3       public String mostrarAlerta(){
4           return nombre + "(Probabilidad de Fallo= " + probFallo
               + ")";
```

```
    }
//=========================================================
```

2.5. Clase Main

Esta clase se encuentra formada por los siguientes atributos y métodos:

```
    //=========================================================
//CLASE MAIN
public class Main {
    //ATRIBUTOS CLASE MAINRNG para Reproducibilidad
    static long semilla=42L;//Semilla
    static double probfallo=0.10;//Probabilidad de fallo
    static int numAlertas=5;//Numero de alertas que se tienen
    static int limiteRecursion=10;//Limite de hasta donde se
       puede llegar

    //METODOS CLASE MAIN
    public static void generarAlerta(int numeAlertas){}
    public static void iniciarSimulacion(){}

    public static void main(String[] args) {//Lectura de los
       parametros por linea
        //Semilla
        if(args.length>=1){semilla=Long.parseLong(args[0]);}
        //Probabilidad de fallo
        if(args.length>=2){probfallo=Double.parseDouble(args[1]);}
        //Numero de alertas
        if(args.length>=3){numAlertas=Integer.parseInt(args[2]);}
        //Limite recursion
        if(args.length>=4){limiteRecursion=Integer.parseInt(args[3]);}

        Random rng=new Random(semilla);//RNG se encuentra
           inicializado con semilla
```

También la clase Main contiene estructuras principales usadas como lo son una cola enlazada y una pila enlazada, estas son inicializadas. Un generador de numero de alertas con la misma probabilidad y que los va añadiendo la alarma a la cola que tiene la misma cantidad de probabilidad. Además, se inicializan los contadores de casos de éxitos y fallos, y la inicialización para la medición del tiempo de la simulación. Se muestran a continuación:

```
     /**Estructuras de datos principales*/
        ColaEnlazada<Alerta>cola=new ColaEnlazada<>();//Cola
           (FIFO)
        PilaEnlazada<Alerta>pila=new PilaEnlazada<>();//Pila
           (LIFO)

        for(int i=0; i < numAlertas; i++){//Genera numero de
           alertas con misma probabilidad
```

```
6               cola.add(new Alerta("Alerta"+i, probfallo,
                    rng));//Las a ade a una cola (misma
                    probabilidad)
7           }
8
9           /**Contadores*/
10          int exitos=0;//Numero de alertas que son exitosas
11          int fallo=0;//Numero de alertas que fallan
12          long tiempo=System.nanoTime();//Mide la duracion total
```

También se crea un bucle en el cual si la cola no esta vacía, la va procesando y vaciando, esto se encuentra en un ciclo *while*. En este bucle toma las alertas en orden (FIFO), y las va añadiendo a una pila de llamadas. Dentro de este ciclo se crea una flujo recursivo de alerta, para poder encontrar la profundidad máxima, si fue un éxito, el contador de éxito se suma, en caso contrario se sima el contador de fallos, y se muestran por pantalla. Esto se muestra a continuación:

```
1       /**La cola no esta vacia(la va vaciando)*/
2           while(!cola.isEmpty()){
3               Alerta a1 = cola.poll();//Toma la siguiente alerta
                    (FIFO)
4               System.out.println(">> Procesando a " +
                    a1.getNombre());
5               pila.push(a1);//Empuja la alerta a la pila de
                    llamadas
6
7               /**Se ejecuta un flujo recursivo de alerta , hasta
                    encontrar la profundidad m xima*/
8               boolean ok= a1.procesarLlamada(pila,
                    limiteRecursion);
9               if(ok){//Indica exito
10                  exitos++;
11                  System.out.println(">> Exito: Alcanzo el
                        l mite" + limiteRecursion);
12              }
13              else{//Indica fallo
14                  fallo++;
15                  System.out.println(">> Fallo: Se interrumpio");
16                  System.out.println("Pofundidad: " +
                        pila.size());
17              }
18
19              pila.vaciar();//Limpia la pila antes de ir a la
                    siguiente alerta
20              System.out.println("-----");
21          }
```

Finalmente en esta clase se coloca una función para la medición de la simulación en milisegundos. De esta manera:

```
/**Tiempo total de la simulaci n en milisegundos*/
    long dt = System.nanoTime()-tiempo;
    System.out.printf(Locale.US, "Resumen -> exitos=%d,
       fallos=%d, tiempoMs=%.3f%n", exitos, fallo, dt/1e6);
```

3. Estrategias de implementación de la estructura de datos

3.1. Implementación

- **Estrategias de implementación para `PilaEnlazada` y `ColaEnlazada`, con énfasis en la gestión de memoria (nodos y redimensión del arreglo).**
  - Se implementó una clase `Nodo<T>` que almacena un valor genérico y una referencia al siguiente nodo.
  - La clase `PilaEnlazada<T>` gestiona la memoria enlazando nuevos nodos al tope en cada operación `push`. Al ejecutar `pop`, simplemente se avanza el puntero del tope, liberando el nodo anterior para recolección de basura. Esto permite una gestión eficiente sin necesidad de redimensionar estructuras de arreglo.
  - La clase `ColaEnlazada<T>` utiliza referencias a `inicio` y `fin`. Cada operación `add` agrega un nuevo nodo al final y actualiza el puntero `fin`. La operación `poll` avanza el puntero de `inicio`, descartando el nodo removido.
  - Ambas estructuras, al ser enlazadas, crecen dinámicamente sin necesidad de redimensionar, optimizando el uso de memoria en comparación con un arreglo.
- **La lógica de la función `procesarLlamada`, explicando cómo la recursividad y la pila interactúan para manejar el flujo de ejecución y los fallos.**
  - La función `procesarLlamada` en la clase `Alerta` implementa la lógica recursiva.
  - **Caso base:** si el tamaño de la pila alcanza el límite definido, la función retorna éxito.
  - **Caso de fallo:** se genera un número aleatorio. Si es menor que la probabilidad de fallo, la función retorna inmediatamente indicando error.

- **Caso recursivo:** si no ocurre fallo, se crea una nueva alerta, se apila en la `PilaEnlazada` y se llama nuevamente al método `procesarLlamada`.
- La pila modela la **profundidad de llamadas anidadas**: cada alerta nueva se agrega al tope, y al terminar la ejecución, se vacía para procesar la siguiente alerta de la cola.

4. Experimentación y resultados

4.1. Dificultades

Existió dificultad en el manejo correcto de índices entre `cantEstudiantes` y `cantEvaluaciones`, así como también confusiones o simplemente distraciones.

La implementación de la clase `AnalizadorDeNotas`, no entregó los resultados esperados principalmente por por el diseño y forma de la utilización de los diferentes métodos usados en `main`.

El mayor problema reflejado en la implementación es la creación incorrecta de generar RUT para cada uno de los estudiantes. Como el RUT es invalido, permite que el método `calcularPromedioEstudiante` tome el RUT como un parámetro erróneo y retorna -1, y mensajes de que el RUT no existe.

Otro de los problemas presentados es en el `Contructor 2` ya que debe recibir un arreglo lleno de parámetros y lo que hace es terminar copiando un arreglo vació lleno de ceros.

Deben existir otros errores y detalles que corregir en el código informático, pero los mencionados anteriormente fueron los más destacables.

5. Conclusión

El laboratorio realizado no obtuvo los resultados esperados, ya que existen errores en el código informático, los cuales permiten una buena ejecución de el pero malos resultados mostrados por pantalla.

En conclusión los cálculos descritos como promedios y varianzas, están correctamente planteados pero el sistema no funciona de una manera correcta.

Uno de los principales errores es la creación errónea de los RUT de los estudiantes, en el `Constructor 2`lo cual lleva a invalidar el RUT ingresado y marcarlo como inexistente y por lo tanto a resultados inconsistentes.

Para trabajos futuros se debe de realizar ajustes mas puntuales en el `Main`, validar correctamente los parámetros, corregir errores que no permitan resultados esperados, mejorar mensajes de salida, entre otros. Con ello en próximos trabajos se debería obtener resultados coherentes y permitir la resolución correcta del problema planteado.