



UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

Laboratorio n°2

Autores: Kevin Cornejo
Javiera Valenzuela

Profesor:
Marcos Fantoval

26-04-2025

Índice

1. Introducción	2
2. Equipos y materiales	2
3. Actividades Realizadas	3
4. Implementacion	3
4.1. Clase Voto	3
4.2. Clase Candidato	4
4.3. Clase Votante	5
4.4. Clase UrnaElectoral - Atributos	6
5. Clase UrnaElectoral - Métodos Clave	7
6. Casos de Prueba	10
7. Experimentación y Resultados	11
7.1. Calculo del espacio requerido para almacenar los votos	11
7.2. Análisis de Complejidad	11
7.3. Observaciones:	11
7.4. Propuesta de mejora	11
8. Conclusión	12

1. Introducción

En la Facultad de Ingeniería y Ciencias de la Universidad Diego Portales se llevarán a cabo las elecciones para presidente del Centro de Alumnos de la Escuela de Informática y Telecomunicaciones y como en sistemas que manejen este tipo de información, es importante que todo funcione de manera segura y ordenada. En este laboratorio se desarrolló un sistema de votaciones en Java llamado ".Electo", pensado para administrar candidatos, votantes y los resultados de una elección. Para realizarlo se utilizaron estructuras de datos como listas enlazadas, pilas y colas que permiten manejar de forma eficiente la información.

El sistema fue diseñado no solo para registrar votos de forma correcta, sino también para prevenir situaciones como el doble voto, gestionar votos reportados y que los resultados sean fáciles de consultar. La idea principal fue acercarse a cómo funcionan los procesos de elección reales y ponerse en todos los casos posibles que podrían ocurrir en una votación.

El código se encuentra en el siguiente enlace:

<https://github.com/StudentKevinC/Laboratorio-2>

2. Equipos y materiales

En el desarrollo del proyecto se utilizaron diversas herramientas y equipos. El entorno de desarrollo IntelliJ IDEA fue empleado para la programación y edición del código fuente. El computador de escritorio permitió ejecutar eficientemente todas las tareas de desarrollo de los programas. Finalmente, GitHub se utilizó como plataforma de repositorio.

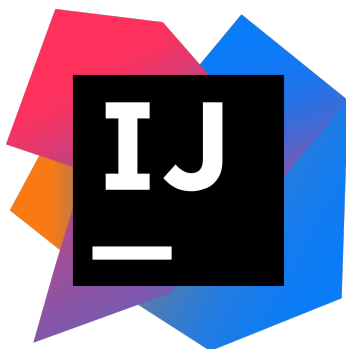


Figura 1: IntelliJ IDEA



Figura 2: Pc



Figura 3: GitHub

3. Actividades Realizadas

Actividades Realizadas

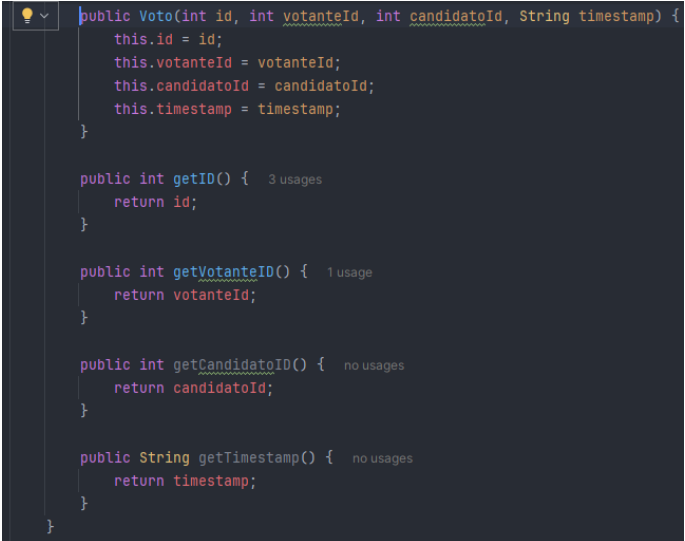
Durante el desarrollo del laboratorio se realizaron diversas actividades esenciales para construir el sistema de votación *Electo*. Se utilizó el entorno de desarrollo IntelliJ IDEA para programar las clases requeridas, permitiendo una gestión eficiente del proyecto y la correcta estructuración del código. Todas las actividades de control de versiones, respaldo de avances y colaboración se llevaron a cabo mediante la plataforma GitHub, lo que aseguró un historial completo de los cambios y facilitó el trabajo organizado. Finalmente, las tareas de programación, prueba y documentación fueron realizadas en un computador de escritorio, cuyo hardware permitió ejecutar de manera fluida las múltiples pruebas necesarias para validar el correcto funcionamiento del sistema, incluyendo la gestión de listas enlazadas, pilas y colas conforme a los requerimientos del laboratorio.

4. Implementacion

La implementación del sistema *Electo* se llevó a cabo siguiendo una estructura orientada a objetos, basada en el diseño planteado en el diagrama UML. Se desarrollaron las siguientes clases principales:

4.1. Clase Voto

La clase *Voto* representa cada voto emitido durante el proceso de votación. Incluye los atributos necesarios para identificar de forma única el voto, registrar el votante, el candidato seleccionado y el momento en que se realizó la votación.

A screenshot of the IntelliJ IDEA code editor showing the implementation of the `Voto` class. The code is written in Java and includes a constructor and four getter methods. The constructor initializes the `id`, `votanteId`, `candidatoId`, and `timestamp` attributes. The getters are `getID()`, `getVotanteID()`, `getCandidatoID()`, and `getTimestamp()`. The IDE's syntax highlighting and code completion are visible.

```
public Voto(int id, int votanteId, int candidatoId, String timestamp) {  
    this.id = id;  
    this.votanteId = votanteId;  
    this.candidatoId = candidatoId;  
    this.timestamp = timestamp;  
}  
  
public int getID() { 3 usages  
    return id;  
}  
  
public int getVotanteID() { 1 usage  
    return votanteId;  
}  
  
public int getCandidatoID() { no usages  
    return candidatoId;  
}  
  
public String getTimestamp() { no usages  
    return timestamp;  
}
```

Figura 4: Clase Voto

Métodos implementados:

- `Voto(int id, int votanteId, int candidatoId, String timestamp)`: Constructor que inicializa los valores del voto con los parámetros entregados.
- `int getID()`: Retorna el identificador único del voto.
- `int getVotanteID()`: Retorna el identificador del votante que emitió el voto.
- `int getCandidatoID()`: Retorna el identificador del candidato votado.
- `String getTimestamp()`: Retorna la hora en que se emitió el voto en formato "hh:mm:ss".

No se implementaron métodos `setters`, dado que una vez creado el voto, sus atributos no deberían ser modificados para garantizar la integridad de los datos.

4.2. Clase Candidato

La clase `Candidato` modela a los candidatos que participan en el proceso de votación. Cada candidato posee atributos que lo identifican y una estructura que almacena los votos que recibe.

```
public static class Candidato { 15 usages
    private int id; 2 usages
    private String nombre; 2 usages
    private String partido; 2 usages
    private Queue<Voto> votosRecibidos; 4 usages

    public Candidato(int id, String nombre, String partido) {
        this.id = id;
        this.nombre = nombre;
        this.partido = partido;
        this.votosRecibidos = new LinkedList<>();
    }

    public void agregarVoto(Voto voto) { 1 usage
        this.votosRecibidos.offer(voto);
    }

    public void eliminarVoto(Voto voto) { 1 usage
        this.votosRecibidos.remove(voto);
    }

    public int getID() { 4 usages
        return id;
    }

    public String getNombre() { 1 usage
        return nombre;
    }

    public String getPartido() { no usages
        return partido;
    }

    public Queue<Voto> getVotosRecibidos() { 3 usages
        return votosRecibidos;
    }
}
```

Figura 5: Clase Candidato

Métodos implementados:

- `Candidato(int id, String nombre, String partido)`: Constructor que inicializa el identificador, nombre, partido político y crea una cola vacía para almacenar los votos recibidos.
- `void agregarVoto(Voto voto)`: Añade un nuevo voto a la cola de votos recibidos.
- `void eliminarVoto(Voto voto)`: Elimina un voto específico de la cola de votos recibidos (por ejemplo, en caso de votos reportados).
- `int getID()`: Retorna el identificador único del candidato.
- `String getNombre()`: Retorna el nombre del candidato.
- `String getPartido()`: Retorna el nombre del partido político del candidato.
- `Queue<Voto>getVotosRecibidos()`: Retorna la cola de votos asociados al candidato.

La clase hace uso de una `Queue` (cola) para almacenar los votos, lo cual es apropiado dado que los votos se insertan en orden de llegada y pueden ser eliminados si es necesario (por ejemplo, en caso de fraude).

4.3. Clase Votante

La clase `Votante` representa a un participante del proceso de votación. Se encarga de almacenar la información básica de cada votante y controlar si ya ha emitido su voto o no.

```
public static class Votante { 6 usages
    private int id; 2 usages
    private String nombre; 2 usages
    private boolean yaVoto; 3 usages

    public Votante(int id, String nombre) {
        this.id = id;
        this.nombre = nombre;
        this.yaVoto = false;
    }

    public void marcarComoVotado() { 1 usage
        this.yaVoto = true;
    }

    public boolean getYaVoto() { 1 usage
        return this.yaVoto;
    }

    public int getID() { 2 usages
        return id;
    }

    public String getNombre() { 1 usage
        return nombre;
    }
}
```

Figura 6: Clase Votante

Métodos implementados:

- `Votante(int id, String nombre)`: Constructor que inicializa el identificador del votante, su nombre, y establece el estado de votación como falso inicialmente (no ha votado).
- `void marcarComoVotado()`: Método que cambia el estado del votante a `true` después de emitir su voto, indicando que ya ha votado.
- `boolean getYaVoto()`: Retorna un valor booleano que indica si el votante ya ha votado.
- `int getID()`: Retorna el identificador único del votante.
- `String getNombre()`: Retorna el nombre del votante.

La clase `Votante` no administra votos directamente. Su única responsabilidad es reflejar el estado del proceso de votación del individuo. La gestión de los votos es realizada por la clase `UrnaElectoral`.

4.4. Clase UrnaElectoral - Atributos

La clase `UrnaElectoral` es la encargada de gestionar el proceso de votación. Sus atributos permiten almacenar, organizar y controlar tanto los candidatos como los votos emitidos y reportados.

```
public static class UrnaElectoral { 2 usages
    private LinkedList<Candidato> listaCandidatos;
    private Stack<Voto> historialVotos; 2 usages
    private Queue<Voto> votosReportados; 2 usages
    int idCounter; 3 usages

    public UrnaElectoral() { 1 usage
        listaCandidatos = new LinkedList<>();
        historialVotos = new Stack<>();
        votosReportados = new LinkedList<>();
        idCounter = 1;
    }
}
```

Figura 7: Clase Urna Electoral

Atributos implementados:

- `LinkedList<Candidato>listaCandidatos`: Lista enlazada que almacena todos los candidatos disponibles en la elección.
- `Stack<Voto>historialVotos`: Pila que guarda el historial de todos los votos emitidos, almacenándolos en orden cronológico inverso.

-
- `Queue<Voto>votosReportados`: Cola utilizada para registrar los votos que han sido anulados o reportados por alguna irregularidad.
 - `int idCounter`: Contador que se incrementa automáticamente para asignar un identificador único a cada voto nuevo generado.

5. Clase UrnaElectoral - Métodos Clave

La clase `UrnaElectoral` contiene los métodos esenciales para administrar el flujo del proceso de votación, desde la verificación del votante hasta el registro y reporte de votos.

Métodos clave implementados:

- `UrnaElectoral()`: Constructor que inicializa la lista de candidatos, el historial de votos y la cola de votos reportados. También inicia el contador de identificadores de votos en uno.

```
public UrnaElectoral() { 1 usage
    listaCandidatos = new LinkedList<>();
    historialVotos = new Stack<>();
    votosReportados = new LinkedList<>();
    idCounter = 1;
}
```

Figura 8: Urna Electoral

- `Candidato buscarCandidato(int id)`: Busca un candidato en la lista enlazada utilizando su identificador único. Retorna el candidato encontrado o `null` si no existe.

```
public Candidato buscarCandidato(int id) {
    for (Candidato c : listaCandidatos) {
        if (c.getID() == id) {
            return c;
        }
    }
    return null;
}
```

Figura 9: Buscar Candidato

- `boolean verificarVotante(Votante votante)`: Verifica si un votante ya ha emitido su voto, retornando `true` si ya votó y `false` en caso contrario.

```

public boolean verificarVotante(Votante votante) {
    return votante.getYaVoto();
}

```

Figura 10: Verificar Votante

- `boolean registrarVoto(Votante votante, int candidatoID)`: Registra un nuevo voto. Verifica si el votante ya ha votado, crea un nuevo objeto `Voto`, lo agrega a la cola de votos del candidato y al historial de votos. Además, marca al votante como `ya votó` y actualiza el contador de IDs.

```

public boolean registrarVoto(Votante votante, int candidatoID) { 3 usages
    if (verificarVotante(votante)) {
        System.out.println("El votante " + votante.getNombre() + " ya ha votado.");
        Candidato candidato = buscarCandidato(candidatoID);
        if (candidato != null) {
            for (Voto v : candidato.getVotosRecibidos()) {
                if (v.getVotanteID() == votante.getID()) {
                    reportarVoto(candidato, v.getID());
                }
            }
        }
        return false;
    }

    Candidato candidato = buscarCandidato(candidatoID);

    if (candidato == null) {
        System.out.println("Candidato no encontrado.");
        return false;
    }

    String timestamp = java.time.LocalDateTime.now().toString();
    timestamp = timestamp.substring(0, 8);

    Voto voto = new Voto(idCounter, votante.getID(), candidatoID, timestamp);

    candidato.agregarVoto(voto);
    historialVotos.push(voto);
    votante.marcarComoVotado();
    idCounter++;

    return true;
}

```

Figura 11: Registrar Voto

- `boolean reportarVoto(Candidato candidato, int idVoto)`: Permite reportar un voto sospechoso o duplicado. El voto es removido de la cola de votos del candidato y añadido a la cola de votos reportados.

```

public boolean reportarVoto(Candidato candidato, int idVoto) {
    System.out.println("Reportando voto..." + idVoto);
    Queue<Voto> votos = candidato.getVotosRecibidos();

    for (Voto v : votos) {
        if (v.getID() == idVoto) {
            votosReportados.offer(v);
            candidato.eliminarVoto(v);
            System.out.println("Voto reportado: " + v.getID());
            return true;
        }
    }

    return false;
}

```

Figura 12: Reportar Voto

-
- `Map<String, Integer> obtenerResultados()`: Genera un mapa que asocia a cada candidato con la cantidad total de votos recibidos, permitiendo obtener rápidamente los resultados de la votación.

```
public Map<String, Integer> obtenerResultados() { no usages
    Map<String, Integer> resultados = new HashMap<>();
    for (Candidato c : listaCandidatos) {
        resultados.put(c.getNombre(), c.getVotosRecibidos().size());
    }
    return resultados;
}
```

Figura 13: Map

- `void agregarCandidato(Candidato candidato)`: Añade un nuevo candidato a la lista de candidatos disponible en la elección.

```
public void agregarCandidato(Candidato candidato) {
    listaCandidatos.add(candidato);
}
}
```

Figura 14: Agregar Candidato

Cada uno de estos métodos permite mantener la integridad del proceso electoral, evitando votaciones duplicadas, registrando adecuadamente cada voto y posibilitando el control de votos irregulares.

6. Casos de Prueba

Con el fin de validar el correcto funcionamiento del sistema de votaciones **Electo**, se realizaron los siguientes casos de prueba, utilizando diferentes escenarios de votación:

- **Caso 1: Registro de voto válido**

El votante Kevin Cornejo emitió un voto válido seleccionando a la candidata Javiera Valenzuela. El sistema permitió el registro del voto, actualizando el historial de votos y marcando al votante como "ya votó".

- **Caso 2: Registro de segundo voto del mismo votante**

El votante Kevin Cornejo intentó emitir un segundo voto seleccionando a Ignacio Ríos. El sistema detectó que ya había votado previamente y procedió a reportar el intento de duplicación, moviendo el voto a la cola de votos reportados.

- **Caso 3: Registro de voto de otro votante**

La votante Javiera Valenzuela emitió un voto válido seleccionando al candidato Kevin Cornejo. El sistema registró correctamente el voto y actualizó el historial y el estado de la votante.

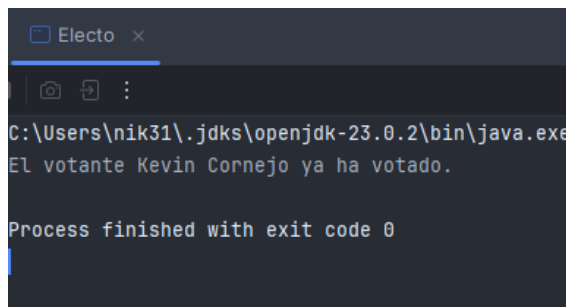
- **Caso 4: Votación por candidato no existente**

Se intentó registrar un voto hacia un candidato con un identificador no existente. El sistema detectó la ausencia del candidato y rechazó la operación, sin afectar el historial ni la cola de votos.

- **Caso 5: Obtención de resultados**

Tras realizar las votaciones válidas, se ejecutó la función `obtenerResultados()`, obteniendo un mapa con el conteo correcto de votos para cada candidato registrado.

Estos casos de prueba permitieron verificar que el sistema maneja adecuadamente la creación de votos, evita la votación duplicada, administra los votos reportados y entrega resultados consistentes.



```
Electo x
C:\Users\nik31\.jdk\openjdk-23.0.2\bin\java.exe
El votante Kevin Cornejo ya ha votado.
Process finished with exit code 0
```

Figura 15: Ejecucion del codigo

7. Experimentación y Resultados

7.1. Calculo del espacio requerido para almacenar los votos

Suponiendo que cada voto utiliza 64 Bytes, y el código admite hasta 10 millones de votantes, el espacio total a ocupar estaría dado por el numero de botos multiplicado por el tamaño de cada voto.

Por lo que se tendría $64 \text{ Bytes} * 1.000.000 = 64.000.000 \text{ Bytes}$ o lo que es igual 640 MB.

7.2. Análisis de Complejidad

Método	Función	Complejidad
registrarVoto	- Buscar candidato en listaCandidato (ciclo for).	$O(n)$
	- Agregar voto a votosRecibidos.	$O(1)$
	- Actualizar estado de votante.	$O(1)$
	Total:	$O(n)$
obtenerResultados	- Iterar sobre listaCandidatos (ciclo for)	$O(n)$
reportarVoto	- Buscar voto en votosRecibidos (ciclo fo)	$O(n)$
	- Agregar el voto a votosReportados	$O(1)$
	- Eliminar voto de la cola de votosRecibidos	$O(1)$
	Total:	$O(n)$

Cuadro 1: Tabla notacion Big-O

7.3. Observaciones:

- El sistema presenta una eficiencia aceptable para un número moderado de candidatos y votantes.
- Las operaciones más costosas son las búsquedas secuenciales tanto de candidatos como de votos en las colas, pero son manejables en contextos de tamaño pequeño o mediano.
- El uso de estructuras como `LinkedList`, `Queue` y `Stack` permite una gestión de memoria eficiente y una correcta organización de los datos en base al flujo natural de la votación.

7.4. Propuesta de mejora

¿Cómo modificarías el sistema para soportar votaciones en múltiples facultades?

Agregar una nueva clase `Facultad`, que contenga su propia instancia de `UrnaElectoral`, lista de candidatos y votantes. De esta forma, cada facultad gestiona su propia elección de manera independiente

Conclusión de Experimentación y Resultados: El sistema *Electo* demuestra un buen rendimiento para procesos electorales de tamaño reducido o intermedio. Para escenarios con gran volumen de datos (por ejemplo, elecciones nacionales), sería recomendable optimizar las estructuras de búsqueda utilizando listas ordenadas, árboles o tablas hash para reducir la complejidad de búsqueda a $\mathcal{O}(\log n)$ o $\mathcal{O}(1)$.

8. Conclusión

El desarrollo de este laboratorio fue una muy buena oportunidad para aplicar lo aprendido sobre estructuras de datos y trabajar con listas enlazadas, pilas y colas nos permitió entender mejor cómo estas estructuras son útiles en situaciones reales donde se necesita manejar mucha información. Logramos hacer un sistema de votaciones que no solo administra de manera ordenada a los votantes y candidatos, sino que también mantiene seguros los datos y evita errores como el doble voto. Como ventajas de utilizar listas enlazadas (LinkedList) sobre arreglos, tenemos que en listas enlazadas resulta más fácil agregar nuevos candidatos o eliminar candidatos ya existentes, también al momento de agregar nuevos votos es más práctico usando listas puesto que este solo se enlaza a un nuevo nodo.

Algunas desventajas de utilizar las listas enlazadas, se encuentra al momento de querer buscar algún candidato por su ID, se vuelve lento ya que para buscarlo se debe recorrer toda la lista, en cambio si se utilizaran arreglos se obtendría el ID del candidato de manera directa. Además, el manejo de los votos también tiene un problema al estar en colas o pilas por qué no es posible acceder directamente a un voto específico mediante su índice.

En general, permitió implementar de manera práctica conceptos de estructuras de datos como las listas enlazadas, pilas y colas en un entorno diferente a las clases. Durante el proceso, se pudo observar cómo el uso de estas estructuras de datos influye directamente en la eficiencia del manejo de la información, especialmente en operaciones como la validación de votantes y el registro de votos. En resumen, a pesar de las dificultades del laboratorio se logró implementar de manera correcta lo solicitado y nos ayudó a aprender más sobre cómo funcionan y las diferencias de estructuras de datos en Java.