**ANÁLISIS DE PROYECTO 4:**

En este proyecto se desarrolló una **simulación de atención médica en una sala de emergencias**, utilizando estructuras de datos eficientes para gestionar la prioridad de los pacientes según su gravedad. El objetivo fue garantizar que los pacientes más críticos sean atendidos antes que los moderados o leves, respetando además el orden de llegada en caso de empate de prioridades. Este sistema combina conceptos de **programación orientada a objetos** y estructuras de datos como **montículos binarios (heaps)** para lograr una atención rápida y eficiente.

**Estructura del Proyecto:**

El programa se compone de las siguientes clases principales:

**Paciente**

Define la entidad que será procesada por el sistema. Cada paciente tiene:

* Nombre y apellido.
* Nivel de riesgo (1 = crítico, 2 = moderado, 3 = bajo).
* Descripción del motivo de consulta.
* Orden de llegada (para desempatar pacientes con igual riesgo).

La clase implementa los métodos de comparación necesarios para que los pacientes puedan ser ordenados según su prioridad en una estructura heap.

**MonticuloBinario**

Es la implementación de un **min-heap** personalizado que permite insertar y extraer elementos en orden de mínima prioridad. Internamente se representa como una lista y garantiza una complejidad de O(log n) en inserciones y extracciones.

**ColaDePrioridad**

Encapsula al MonticuloBinario y lo adapta para modelar una cola de prioridad médica. Al insertar pacientes, los organiza según su nivel de riesgo, y en caso de empate, por orden de llegada. Provee métodos para:

* Insertar un nuevo paciente.
* Atender (extraer) al paciente más urgente.
* Ver el siguiente en la cola.
* Mostrar el estado actual de la sala.

**Simulación Principal**

Se implementa un ciclo de simulación que representa una guardia médica en tiempo real:

* En cada paso del ciclo, se genera un nuevo paciente con un riesgo aleatorio (1 a 3).
* Se muestra el estado de la cola.
* Se atiende al paciente más urgente (o el que llego antes en determinados casos de misma urgencia).
* Se imprime el número de pacientes aún en espera.

Este comportamiento simula de forma realista el funcionamiento de una sala de emergencias saturada.

**Pruebas Unitarias**

Se desarrollaron múltiples pruebas automatizadas con el módulo unittest, organizadas en la clase TestColaDePrioridad, que verifica los siguientes casos:

* **Inserción ordenada**: Verifica que los pacientes se ordenan correctamente al insertarlos según su nivel de riesgo y orden de llegada.
* **Atención correcta**: Se asegura de que el paciente más urgente siempre sea el primero en ser atendido.
* **Desempate por orden de llegada**: Comprueba que, ante igual riesgo, se respeta la llegada cronológica.
* **Cola vacía**: Verifica que no se produzcan errores si se intenta atender sin pacientes.

Estas pruebas confirmaron el correcto funcionamiento de la lógica de prioridad, la robustez de la simulación, y la integridad de la estructura heap implementada.

**Resultados de la Simulación**

Se simulan 20 ciclos de atención, donde en cada ciclo:

* Se genera un nuevo paciente con un nivel de riesgo asignado aleatoriamente según una distribución predefinida.
* El paciente es encolado en una estructura de cola de prioridad.
* En el 50% de los ciclos se atiende (desencola) al paciente con mayor prioridad (riesgo más alto y llegada más temprana).

Se utilizaron estructuras de datos eficientes:

* La **cola de prioridad** está implementada sobre un **montículo binario** (heap mínimo), lo cual garantiza que el acceso y eliminación del elemento de mayor prioridad se realiza en tiempo logarítmico.
* La simulación imprime el estado interno de la cola en cada iteración para facilitar el seguimiento del sistema.

**Complejidad Algorítmica**

La eficiencia del sistema se sustenta en la implementación del **montículo binario**, que es utilizado como estructura base para la cola de prioridad. A continuación se detalla el análisis de complejidad para las operaciones clave:

| Operación | Complejidad Temporal | Detalles |
| --- | --- | --- |
| insertar(k) | O(log n) | Se inserta al final y se reacomoda con infiltArriba. |
| eliminarMin() | O(log n) | Se reubica el último nodo en la raíz y se aplica infiltAbajo. |
| encolar(paciente) | O(log n) | Equivale a insertar(k) en el montículo. |
| desencolar() | O(log n) | Equivale a eliminarMin() en el montículo. |
| Iteración por la cola | O(n) | Se recorre una copia de la lista interna del montículo (sin modificarlo). |

Donde n representa la cantidad de pacientes en la cola.

Además, la comparación entre pacientes (\_\_lt\_\_) se realiza en tiempo constante **O(1)**, ya que compara atributos enteros (riesgo y orden de llegada).

**Escalabilidad y Estabilidad**

El diseño basado en montículo permite escalar la simulación a cientos o miles de pacientes sin un impacto significativo en la eficiencia del sistema. La combinación de prioridad (riesgo) y orden de llegada permite un desempate estable y justo entre pacientes con el mismo nivel de criticidad.

**Robustez y Correctitud**

Las pruebas unitarias implementadas validan los siguientes casos:

* Inserción y eliminación en cola vacía.
* Mantenimiento del orden de prioridad entre múltiples pacientes.
* Desempate correcto por orden de llegada en casos de igual prioridad.
* Casos mixtos de múltiples niveles de riesgo.

Esto asegura que la cola de prioridad funciona correctamente bajo distintos escenarios.

**Conclusión**

Este proyecto permitió aplicar conceptos clave de estructuras de datos (montículos, colas de prioridad), junto con técnicas de simulación y testeo automático. Se demostró cómo una implementación eficiente puede reproducir una situación real (una guardia médica) priorizando correctamente según criterios establecidos. Además, el sistema pasó exitosamente todas las pruebas, lo que valida tanto la lógica del simulador como la estructura de datos subyacente.

**Implementación de Árbol AVL para Registro de Temperaturas**

**Objetivo General**

Este módulo implementa un **árbol AVL autobalanceado** para almacenar, consultar y manipular registros de **temperatura diaria** a través de objetos de tipo datetime.date. La estructura garantiza eficiencia logarítmica en operaciones clave: inserción, búsqueda y eliminación.

**Estructura del Código**

**Clase nodoAVL**

Representa un nodo individual del árbol AVL, que contiene:

* fecha: Clave del nodo (datetime.date), que identifica unívocamente cada registro.
* temperatura: Valor asociado a la fecha.
* altura: Altura del nodo (para el balance AVL).
* izquierda y derecha: Punteros a subárboles hijos.

**Validaciones**:

* Lanza un ValueError si fecha no es del tipo datetime.date.

**Clase AVL**

Encapsula toda la lógica del árbol AVL.

**Métodos clave:**

|  |
| --- |
| *insertar(fecha: datetime.date, temperatura: float)* |

* Inserta un nuevo nodo.
* Si la fecha ya existe, **actualiza** la temperatura.
* Realiza rebalanceo con rotaciones simples y dobles según sea necesario.

**Búsqueda:**

|  |
| --- |
| *buscar(fecha: datetime.date) → float | None* |

* Retorna la temperatura asociada a la fecha, o None si no existe.

**Eliminación:**

|  |
| --- |
| eliminar(fecha: datetime.date) |

* Elimina el nodo con la fecha dada.
* Si tiene dos hijos, se reemplaza con el sucesor in-order.
* Rebalancea tras la eliminación.

|  |
| --- |
| *obtener\_rangos(fecha\_min, fecha\_max) → List[float]* |
| *obtener\_datos\_en\_rango\_ordenados(fecha\_min, fecha\_max) → List[Tuple[fecha, temperatura]]* |

* Obtienen temperaturas dentro de un rango de fechas (sin y con ordenamiento).

|  |
| --- |
| *contar\_nodos() → int* |

* Devuelve el total de registros en el árbol.

|  |
| --- |
| *obtener\_valor\_minimo(nodo\_actual) → nodoAVL* |

* Devuelve el nodo con la fecha más temprana en el subárbol.

**Rotaciones para Balanceo**

El rebalanceo automático se logra con:

* **Rotación simple a derecha (rot\_derecha)**
* **Rotación simple a izquierda (rot\_izquierda)**
* **Rotación doble izquierda-derecha (rot\_izq\_der)**
* **Rotación doble derecha-izquierda (rot\_der\_izq)**

Estas operaciones mantienen el árbol AVL balanceado tras inserciones o eliminaciones.

**Conclusión**

Esta implementación de un **árbol AVL para registro de temperaturas por fecha** es robusta, eficiente y adaptable. Proporciona una base sólida para desarrollos que requieren acceso rápido y ordenado a datos temporales, como aplicaciones meteorológicas, estadísticas históricas o gestión de sensores de IoT.

**sistema de envío de noticias entre aldeas**

Este proyecto implementa un sistema de optimización para el envío de noticias entre aldeas utilizando el **algoritmo de Prim** para encontrar un **árbol de expansión mínima (MST)**. El grafo de aldeas se construye desde un archivo de texto, y las conexiones (aristas) representan las distancias entre las aldeas. La solución hace uso de estructuras personalizadas como un **montículo binario**, una **cola de prioridad** y un **grafo no dirigido**.

**Estructura de módulos y clases**

*📂 modules/vertice.py → Vertice*

* Representa un nodo del grafo.
* Contiene un diccionario conectadoA que mapea vecinos a sus ponderaciones.
* Métodos clave:
  + *agregarVecino*(): Añade un vecino con peso.
  + *obtenerConexiones*(): Devuelve los vecinos.
  + *obtenerPonderacion*(): Devuelve el peso hacia un vecino dado.

*📂 modules/grafo.py → Grafo*

* Representa un grafo no dirigido.
* Usa un diccionario listaVertices con claves únicas de nodos.
* Conexiones bidireccionales entre nodos (*agregarArista*() crea ida y vuelta).
* Soporta iteración y uso de *len*().

*📂* *modules/monticulo.py → MonticuloBinario*

* Implementa un montículo binario mínimo usando una lista.
* Métodos clave:
  + *insertar*(): Añade un nuevo elemento manteniendo el orden.
  + *eliminarMin*(): Extrae el mínimo y reestructura el montículo.
  + *construirMonticulo*(): Convierte una lista no ordenada en un montículo.

*📂 modules/cola\_de\_prioridad.py → ColaDePrioridad*

* Encapsula un *MonticuloBinario* y gestiona la prioridad con claves.
* Utiliza *itertools.count*() para resolver empates por orden de llegada.
* Permite iteración directa sobre elementos encolados.

**Flujo principal de ejecución (main.py)**

1. **Cargar grafo desde archivo**:
   * El archivo "data/aldeas.txt" contiene líneas con el formato: Aldea1, Aldea2, Distancia.
   * Cada línea válida crea dos conexiones bidireccionales.
2. **Aplicar algoritmo de Prim**:
   * Se parte desde la aldea "Peligros".
   * Se usa una ColaDePrioridad para manejar las aristas salientes.
   * Se va construyendo el MST evitando ciclos y minimizando la distancia total.
3. **Construcción del árbol de envíos**:
   * Se arma una estructura hijos\_de[aldea] para mostrar a quién retransmite la noticia cada nodo.
   * Se imprime información sobre:
     + Quién envía a quién.
     + A quién se le reenvía.
     + Aldeas en orden alfabético.
     + La distancia total recorrida.

**Datos esperados de entrada**

Archivo: data/aldeas.txt  
Formato por línea (sin encabezado):

|  |
| --- |
| Peligros, Armilla, 10  Armilla, Ogíjares, 5  ... |

**Conclusión**

Este proyecto ofrece una implementación funcional y didáctica del algoritmo de Prim, destacando por:

* Su uso de estructuras personalizadas para aprender algoritmos de grafos.
* Su claridad modular.
* Su foco en un caso práctico: la difusión de noticias entre aldeas.

Su estructura lo convierte en un excelente ejemplo educativo y puede ser fácilmente extendido para casos más complejos de redes o logística.