Facultad de Ingeniería

Algoritmos y Estructuras de datos

Informe Trabajo Práctico N° 2

Integrantes: Obaid Aixa, Caraballo Mateo

Actividad 1

En esta simulación reemplazamos la lista simple por una **cola de prioridad** para que siempre se atienda primero al paciente más crítico, sin importar el orden de llegada.
La prioridad se define como una tupla (riesgo, fecha_llegada), donde un riesgo más bajo indica mayor urgencia. Si dos pacientes tienen el mismo nivel de riesgo, se prioriza al que llegó antes.
La estructura se implementó como un **heap** manual, manteniendo la eficiencia en las operaciones de inserción y extracción. Aunque fue diseñada para pacientes, es **genérica y reutilizable** en otros contextos.
Con esta solución logramos un sistema **rápido y ordenado**, que respeta el criterio médico de triaje y se mantiene eficiente incluso con muchos pacientes en espera.

Actividad 2

Implementamos la clase Temperaturas_DB utilizando un árbol AVL para almacenar temperaturas asociadas a fechas, lo que permite mantener los datos ordenados y acceder a ellos de forma eficiente. La estructura balanceada del árbol garantiza buen rendimiento en operaciones de inserción, búsqueda y eliminación. El sistema permite guardar temperaturas, consultarlas por fecha o por rango, obtener valores extremos y eliminar registros. También se puede consultar la cantidad total de muestras almacenadas. Se realizaron pruebas que verifican el correcto funcionamiento de cada operación y aseguran que el árbol se mantenga balanceado y responda adecuadamente a las consultas.

Tabla de Complejidades Big-O

Método	**Complejidad**	**Explicación breve**
guardar_temperatura	O(log n)	Inserción en árbol AVL balanceado.
devolver_temperatura	O(log n)	Búsqueda en árbol AVL.
borrar_temperatura	O(log n)	Eliminación y rebalanceo en AVL.
max_temp_rango	O(k + log n)	Búsqueda en rango (k elementos en rango).
min_temp_rango	O(k + log n)	Similar a max_temp_rango.
temp_extremos_rango	O(k + log n)	Recorrido para mínimos y máximos en rango.
devolver_temperaturas	O(k + log n)	Devuelve listado ordenado de temperaturas.
cantidad_muestras	O(1)	Simple retorno del contador.

Donde n es la cantidad total de nodos y k la cantidad de nodos dentro del rango consultado.

Actividad 3

En esta actividad teníamos que encontrar la forma de repartir un mensaje a todas las aldeas sin pasar dos veces por la misma. La condición era que cada aldea solo podía enviar mensajes a sus aldeas vecinas. Para resolver esto, usamos el algoritmo de Prim, que nos permite armar un árbol de expansión mínima (o AEM), es decir, una estructura que conecta todos los nodos con el menor costo posible.

La cátedra nos compartió un archivo con los nombres de las aldeas y sus conexiones vecinas, y a partir de eso armamos nuestro grafo. Para la implementación, reutilizamos una clase que ya teníamos (MonticuloBinario) y desarrollamos dos clases nuevas: Vertice y Grafo, que son claves para poder usar Prim.

La clase Vertice representa cada aldea, guarda con qué otras aldeas está conectada, el peso (o distancia) de esas conexiones, y también la distancia mínima que tiene hasta el momento, junto con el vértice desde el que se llegó (su predecesor). Toda esta info es necesaria para que Prim pueda ir eligiendo siempre la mejor opción.

La clase Grafo, por su parte, se encarga de organizar todos los vértices y aristas. Tiene métodos para agregar vértices, crear aristas entre ellos y también para cargar todos los datos directamente desde un archivo, lo cual nos facilitó bastante el trabajo.

Prim empieza desde un nodo que elegimos como inicial (en nuestro caso, la aldea "Peligros"). A todos los vértices les asigna una distancia infinita al principio, menos al de inicio que arranca con 0. Después, usando el MonticuloBinario, va armando el árbol seleccionando siempre el vértice con menor distancia y actualizando las conexiones de sus vecinos. Esto hace que podamos repartir el mensaje a todas las aldeas de la forma más eficiente posible, sin repetir caminos ni perder tiempo.