Taller 4 – Conjunto implementado sobre Árbol Binario de Búsqueda

En este taller se implementa la estructura de datos **Conjunto**, utilizando internamente un **Árbol Binario de Búsqueda (ABB)**. La idea es poder guardar elementos sin repetidos, de manera ordenada, y con la posibilidad de agregarlos, eliminarlos y consultarlos de forma eficiente.
Para poder ordenar los elementos, el tipo genérico `T` que se guarda en el conjunto debe implementar la interfaz `Comparable<T>`. Esto permite que podamos comparar dos elementos y decidir si uno es “menor”, “igual” o “mayor” que el otro.

Estructura interna

La clase principal es `ABB<T>`, que implementa la interfaz `Conjunto<T>`. Internamente, cada elemento está dentro de un **nodo** (`Nodo`), que es un objeto con:

- el **valor** que guarda el nodo,
- una referencia a su **hijo izquierdo** (`izq`),
- una referencia a su **hijo derecho** (`der`),
- y una referencia a su **padre** (`padre`).

El árbol comienza en un nodo especial llamado **raíz** (`_raiz`), y cada vez que insertamos un elemento nuevo, lo ubicamos siguiendo las reglas del árbol binario de búsqueda:

- si el elemento es menor que el nodo actual, avanzamos a la izquierda;
- si es mayor, avanzamos a la derecha;
- si es igual, no se inserta (porque en un conjunto no puede haber repetidos).

También se lleva un contador `_cardinal` con la cantidad de elementos que hay en el conjunto.

Principales operaciones

- **`cardinal()`**: devuelve la cantidad de elementos.

- `minimo()` y `maximo()`: recorren el árbol hasta llegar al elemento más chico (rama más a la izquierda) o más grande (rama más a la derecha).

- `pertenece(elem)`: busca un elemento en el árbol siguiendo el criterio de orden.

- `insertar(elem)`: si el árbol está vacío, el nuevo elemento pasa a ser la raíz. Si no, se busca dónde colocarlo comparando valores, y se lo enlaza en la posición correcta.

- `eliminar(elem)`: ubica el elemento y lo borra manejando distintos casos: que sea una hoja, que tenga un solo hijo, o que tenga dos hijos (en este último caso, se reemplaza por el siguiente en orden).

- `toString()`: devuelve una representación como texto de todos los elementos del conjunto, en orden ascendente y separados por comas, encerrados entre llaves.

El iterador y las listas

En programación, una **lista** es una estructura que guarda elementos uno tras otro, y a la que podemos recorrer desde el primero hasta el último, o incluso en cualquier posición intermedia. Un **iterador** es un objeto que nos permite recorrer una colección de elementos **uno por uno**, sin que tengamos que preocuparnos por cómo están guardados internamente.

En este taller, el iterador (`ABB_Iterador`) recorre el árbol **en orden** (*in-order traversal*), es decir:

1. Visita todo el subárbol izquierdo.

2. Visita el nodo actual.

3. Visita el subárbol derecho.

Para lograrlo, usa una **pila** (`Stack<Nodo>`) que va guardando los nodos por visitar. Al iniciar el iterador, se apilan todos los nodos que están a la izquierda de la raíz. Cada vez que pedimos el siguiente elemento, se desapila un nodo, y si este tiene hijo derecho, se apilan todos los nodos a la izquierda de ese hijo.

Este diseño permite que podamos recorrer el conjunto como si fuera una **lista ordenada**, pero sin necesidad de guardar realmente una lista aparte. El iterador se encarga de “traducir” la estructura del árbol en una secuencia lineal de elementos que podemos consumir de a uno.

En el método `toString()`, por ejemplo, se crea un iterador y se van pidiendo elementos hasta que ya no haya más, construyendo así la cadena que representa al conjunto.