	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): Edgar Tista García

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 3

No de Práctica(s): 8

Integrante(s): Román Ramos María Fernanda

No. de lista o brigada:

Semestre: 2026-1

Fecha de entrega: 25/10/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Objetivos

- Objetivo: El estudiante conocerá e identificará las características de la estructura no-lineal árbol
- Objetivo de clase: El alumno analizará las implementaciones proporcionadas y conocerá la forma en la que se pueden implementar estas estructuras de datos

**Nota: Para el desarrollo de la práctica usaré las herramientas:*

- *Visual Studio Code: para crear e implementar los programas en Java.*
- *Overleaf: para crear este documento en LaTeX del reporte.*

2. Desarrollo

2.1. Implementación de un árbol Binario

2.1.1. Métodos en ArbolBinario.java

- **ArbolBinario()**: Constructor, inicializa la raíz del árbol en null, porque el árbol está vacio
- **insertar(Nodo padre, int valor, *boolean esIzquierdo*)**: inserta un nuevo **nodo** en el árbol:
  - si padre es null: inserta el valor como raíz si el árbol está vacío, pero si ya tiene raíz, manda error.
  - si padre no es null: busca el nodo padre. Si *esIzquierdo* es *true*, asigna el nuevo nodo como su hijo izquierdo, verificando que no tenga uno ya. si *esIzquierdo* es *false*, lo asigna ocmo hijo derecho, verificando que no tenga uno ya
- **buscar(int valor)**: hace una búsqueda BFS para encontrar un nodo con el valor, usa una queue para recorrer el árbol nivel por nivel, regresa el nodo si lo encuentra o null si no existe

- **eliminar(int valor)**: elimina un nodo con el **valor** dado, reemplaza el nodo a eliminar con el nodo más profundo y a la derecha (el último nodo en el recorrido bfs), y luego elimina ese nodo más profundo
  - busca el nodo a eliminar (keyNode) y también el último nodo recorrido (temp qeu será el más profundo y a la derecha)
  - si encuwntra el nodo a eliminar, sobrescribe su valor con el valor del nodo más profundo
  - llama a eliminarNodoMasProfundo(temp) para borrar el nodo que fue movido
- **eliminarNodoMasProfundo(Nodo delNode)**: recorre el árbol para encontrar la referencia al nodo delNode y establecer la referencia de su padre (ya sea hijo izquierdo o derecho) a null, eliminando el nodo
- **preorden(Nodo nodo):** hace el recorrido Preorden (raíz, izquierda, derecha) de forma recursiva a partir del nodo dado
- **inorden(Nodo nodo)**: hace el recorrido Inorden (izquierda, raíz, derecha) de forma recursiva a partir del nodo dado
- **posorden(Nodo nodo)**: hace el recorrido Posorden (izquierda, derecha, raíz) de forma recursiva a partir del nodo dado-
- **imprimirArbol()**: muestra el árbol haciendo un recorrido BFS, imprime el valor de cada nodo y la referencia a sus hijos izquierdo y derecho

2.1.2. Pruebas de PruebaBinario.java

La función **eliminar(int valor)** en *ArbolBinario.java* implementa una estrategia específica para la eliminación en un árbol binario no de búsqueda:

1. identificar el nodo: busca el nodo a eliminar con el valor especificado y con un recorrido BFS identifica el último nodo en ser visitado (temp) uqe es el nodo más profundo la derecha
2. si se encuentra el nodo, se pone el valor del nodo más profundo (temp.valor) al nodo a eliminar (keyNode.valor = valorTemp), el nodo a eliminar mantiene su posición y sus hijos, pero se actualiza su valor

3. eliminación del nodo mas profundo: se llama a la función eliminarNodoMasProfundo(temp) para quitar el nodo temp (el que estaba más profundo la derecha)

Caso de eliminación del nodo 8:

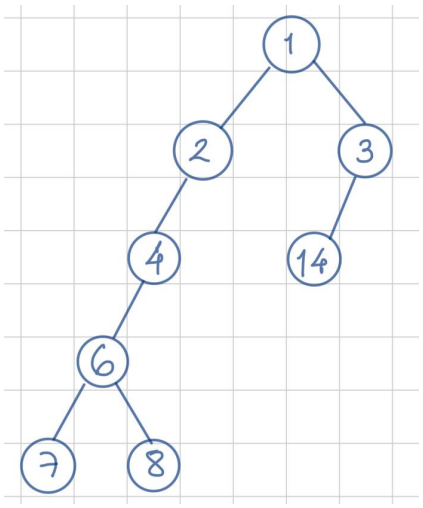

Figura 1: Árbol que se construye inicialmente en el programa PruebaBinario

- nodo a eliminar (keyNode): nodo 8
- nodo más profundo (temp): nodo 8
- el valor del nodo más profundo (8) se copia al keyNode (8), lo cual no cambia el valor, luego, el nodo más profundo (8) es eliminado de su posición

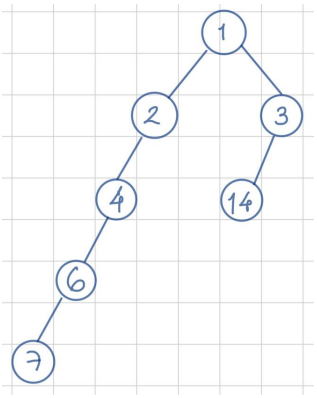

Figura 2: Árbol después de eliminar el 8

Caso de eliminación del nodo 1:

- nodo a eliminar (keyNode): nodo 1 (raiz)
- nodo más profundo (temp): nodo 7
- el valor del nodo más profundo (7) se copia al keyNode (1), luego, el nodo más profundo (7) es eliminado de su posición

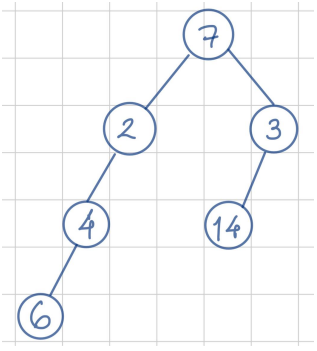

Figura 3: Árbol después de eliminar el 1

2.1.3. PruebaBinario2

2.1.4. Ventajas y Desventajas de la Implementación de Árbol Binario

Ventajas:

- se puede construir **cualquier** tipo dw árbol binario como los que vimos en clase (balanceado, completo, full, degenerado, etc), ya que el orden de los valores ”no importa”

Desventajas:

- el método buscar hace un recorrido BFS, que tiene una complejidad de $O(n)$ en el peor caso, ya que debe visitar todos los nodos hasta encontrar el valor, ya que si un árbol tiene **n** numero de noros, se tiene que revisar os **n** nodos.

2.1.5. Impresión de recorridos

- **Preorden (Raíz → Izquierdo → Derecho):**
  1. procesa el nodo actual (la raíz)
  2. llama al método recursivamente para el subárbol izquierdo
  3. llama al método recursivamente para el subárbol derecho
- **Inorden (Izquierdo → Raíz → Derecho):** *prioriza el recorrido en el subárbol izquierdo antes de visitar la raíz.
  1. llama al método recursivamente para el subárbol izquierdo
  2. procesa el nodo actual (la raíz)
  3. llama al método recursivamente para el subárbol derecho
- **postorden (Izquierdo → Derecho → Raíz)** *visita y procesa el nodo actual después de haber recorridp los subárboles
  1. llama al método recursivamente para el subárbol izquierdo
  2. llama al método recursivamente para el subárbol derecho
  3. procesa el nodo actual (la raíz)

2.2. Árbol binario de búsqueda

2.2.1. Diferencias en los métodos

- **insertar(int valor):**
  es recursivo y no reuqiere un nodo padre ni si es hijo izquierdo o derecho, la posición se decide con: si **valor** < nodo.**valor** va a la izquierda, si **valor** > nodo.**valor** va a la derecha
- **buscar(int valor):**
  es recursivo y no requiere un recorrido BFS, la búsqueda se va automáticamente hacia la izquierda o derecha según el valor a buscar si es mayor o menor
- **eliminar(int valor):**
  es recursivo y tiene tres casos: 0, 1 o 2 hijos. Para 2 hijos, reemplaza el nodo eliminado por el valor mínimo del subárbol derecho

2.2.2. Eliminaciones

Código donde agrego más nodos y hago las eliminaciones:

```
abb.insertar(valor:50);
abb.insertar(valor:30);
abb.insertar(valor:70);
abb.insertar(valor:20);
abb.insertar(valor:40);
abb.insertar(valor:60);
abb.insertar(valor:80);
System.out.println(x:"arbol Inicial:");
abb.imprimirArbol();
// nuevps nodos
abb.insertar(valor:10); // hijo izq de 20
abb.insertar(valor:65); // hijo izq de 70
abb.insertar(valor:55); // hijo izq de 60
abb.insertar(valor:45); // hijo der de 40

System.out.println(x:"arbol después de añadir más nodos:");
abb.imprimirArbol();

System.out.println(x:"\n--- eliminar 70 ---");
abb.eliminar(valor:70);
abb.imprimirArbol();

System.out.println(x:"\n--- eliminar 10 ---");
abb.eliminar(valor:10);
abb.imprimirArbol();
```

Figura 4: Códico de eliminaciones

Representación de cada parte:

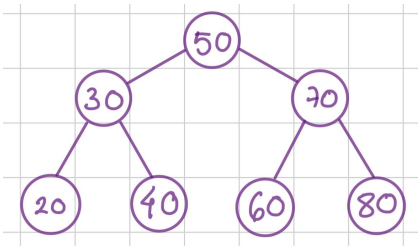

Figura 5: Árbol inicial

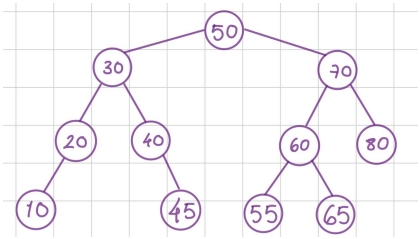

Figura 6: Arbol después de añadir más nodos

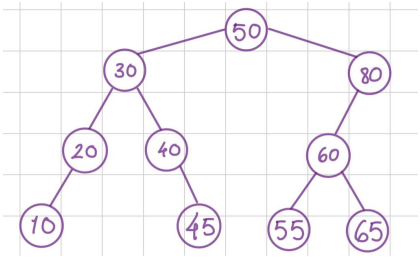

Figura 7: eliminar 70

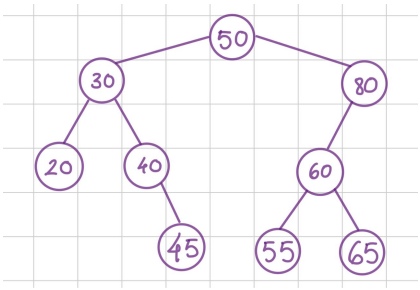

Figura 8: eliminar 10

Explicación de la Eliminación

1. **caso 1: nodo Hoja**
   - **si** *nodo.hIzquierdo* == *null* **y** *nodo.hDerecho* == *null*.
   - **s**e devuelve *null*, el puntero de su padre "apunta" a *null*, eliminando el nodo
   - **Ejemplo:** al eliminar *10* el nodo *20* ya no "apuntará" a *10*.
2. **caso 2: Nodo con 1 hijo**
   - **si** *nodo.hIzquierdo* == *null* **o** *nodo.hDerecho* == *null*.
   - se devuelve el puntero al único hijo (izquierdo o derecho). Esto hace que el padre del nodo eliminado"salte" este nodo y apunte directamente a su nieto
   - **ejemplo:** Si se hubiera eliminado *20* (que solo tiene un hijo: *10*), el nodo *30* apuntaría directamente a *10* en lugar de a *20*.
3. **caso 3: Nodo con 2 hijos**
   - **si** *nodo.hIzquierdo*! = *null* **y** *nodo.hDerecho*! = *null*.
   - **E**l nodo eliminado es reemplazado por el valor del nodo con el valor más pequeño en el subárbol derecho (obtenido con *valorMinimo(nodo.hDerecho)*), se copia este valor al nodo eliminado, y el mpas pequeño del subárbol derecho es eliminado recursivamente de su posición original.
   - **ejemplo:** Al eliminar *70* el nodo con el valor más pequeño en el subárbol derecho es *70*, se copia a *70*. Luego, el nodo de (*70*) es eliminado recursivamente (caso 1 o 2),el nodo a eliminar se reemplaza con el valor más a la izquierda de su subárbol derecho

2.2.3. Casos de uso

Árbol Binario General:

- se puede usar cuando en la relación entre nodos no o importa el orden de los valores
- puede representar expresiones aritméticas donde no importa el orden de las operaciones

Árbol Binario de Búsqueda:

- Es mejor para casos de búsqueda, ya que será $O(logn)$ por su lógica, que revisa si es mayor o menor y dependiendo del resultado, se irá a un subarbol o a otro comparando las claves, es decir, será una implementación de Busqueda Binaria.

2.2.4. Funcionamiento de Pre,in y pos-orden y busqueda

```
Nodo encontrado = abb.buscar(valor:40);
System.out.println("\nBúsqueda de 40: " + (encontrado != null ? "encontrado" : "no Encontrado"));
Nodo noEncontrado = abb.buscar(valor:99);
System.out.println("Búsqueda de 99: " + (noEncontrado != null ? "encontrado" : "no Encontrado"));

System.out.println(x:"\n--recorrido inorden:");
abb.inorden(abb.raiz);
System.out.println(x:"\n--recorrido preorden:");
abb.preorden(abb.raiz);
System.out.println(x:"\n--recorrido posorden:");
abb.posorden(abb.raiz);
System.out.println(x:"\n");
```

Figura 9: Código

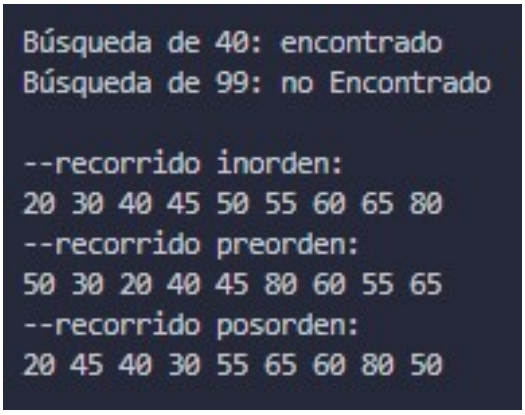

Figura 10: Resultado

***En busquedas:** hice una búsqueda par aun valor que se encuentra y otro que no, en cada caso concatené en las salidas un ternario en lugar d eun if para ver si encontrado y noEncontrado estaban en nulll o con el valor que correspondia si si se encuentra.
***En recorridos:**
se puede ver como en el recorrido **inorden** efectivamente se imprimern en orden ascendente los valores yq que revisa los menores (subuarbol izquierdo), la raiz y los mayores (subuarbol derecho).
en **preorden**: El resultado muestra los nodos en el orden en que fueron agregados al árbol
en **Postorden** muestra las hojas primero, luego intermedios y al final la raíz

2.3. Menú de usuario

En el código MenuArboles.java implementé los métodos de ArbolBinario y ArbolBinarioBusqueda con switch-case como siempre se hace para este tipo de menús, pero en lugar de ponderlo todo en el mismo switch lo dividí en varias funciones para facilitarme el manejo de los errores que surgían (que fueron muchos).

3. Conclusiones

La páctica me fué útil para ver cómo se pueden implementar las estructuras no lineales, así que se cumplieron los objetivos de la práctica. Logré entender la difrencia en las operaciones y la lógica de los Árboles Binarios Generales y Árboles Binarios de Búsqueda, también me fué útil ir haciendo las operaciones de los programas a mano, para ver cómo era que funcionaba cada una y logré aclarar dudas de los funcionamientos, que ahora me parecen mucho más sencillos o más intuitivos. También me pareció interesante y útil ver las ventajas y desventajas de cada uno, que por ejemplo para busqueda es mejor Árbol Binario de Búsqueda y para aplicaciones "más sencillas" creoq eu es mas conveniente o más fácil aplicar Árbol Binario, ya que aquí no importa el orden de los nodos y las operaciones son más sencillas.
Por otra parte, aunque era opcional (y fué en lo que más tardé y sí fué más difícil), sí quise hacer la parte del menú porque en prácticas anteriores de esta materia y de Programación Orientada a Objetos me han sido muy útiles ya que justamente debo fijarme qué datos se necesitan para hacer cada operación del menú, como organizarlo y sobre todo, que la aplicación sea "facil" de usar, igualmente mientras hago cada opción voy haciendome una idea de cuál es la "respuesta" o salida que se espera.

4. Referencias