Implementación de recorridos en árboles binarios

Taller 9 — Recorridos de Árboles

Wilson Palma
Miguel Armas
Ana Panamito
Cristhian Dávila
Steven Jumbo

2026-01-22

Resumen

Implementación en Java de los algoritmos de recorrido de árboles binarios: preorden, inorden, postorden y por niveles (BFS). El proyecto incluye la carga del árbol desde un archivo en formato indexado (index;value;left;right), validaciones de consistencia de los datos, impresión visual del árbol en consola y el cálculo de métricas estructurales como tamaño y altura. La ejecución se realiza desde IntelliJ IDEA y la salida se registra en el archivo logs/salida_en_consola.txt como evidencia del funcionamiento.

Resumen ejecutivo

Se implementó una aplicación Java que carga un árbol binario desde archivo (formato indexado o lista simple), construye la estructura `BinaryTree`, ejecuta los recorridos preorden, inorden, postorden y por niveles (BFS), muestra una representación visual en consola y calcula métricas básicas (tamaño, altura). Se valida el funcionamiento con el archivo `data/arbol_10_nodos.txt` y la salida se registra en `logs/salida_en_consola.txt`.

Objetivos

- Implementar recorridos clásicos en un árbol binario: preorden, inorden, postorden y por niveles.
- Permitir carga desde archivo en formato indexado `index;value;left;right`.
- Mostrar la estructura del árbol en consola y las métricas principales.
- Proveer un programa reproducible desde IntelliJ y desde terminal.
- Entregar evidencias de ejecución y un informe técnico.

Estructura del repositorio (relevante)

```
/ (raíz)
├─ data/
│  └─ arbol_10_nodos.txt
├─ logs/
│  └─ salida_en_consola.txt
├─ src/
│  └─ main/
│     └─ java/
│        └─ ed/u3/
│           ├─ App.java
```

```
    │           ├─ core/BinaryTree.java
    │           ├─ model/TreeNode.java
    │           └─ util/{TreeBuilder.java, TreePrinter.java}
    └─ README.md
```

Formato de entrada usado

- Formato indexado soportado por `TreeBuilder.fromIndexedFile`:
  Cada línea (tras una primera línea opcional con N) con el patrón: `index;value;left;right`
- `index`: entero identificador del nodo.
- `value`: entero (valor del nodo).
- `left`: índice del hijo izquierdo o `-1`.
- `right`: índice del hijo derecho o `-1`.
- La raíz se determina como el `index` que **no aparece** en ningún campo `left` o `right`.

Metodología

1. Lectura del archivo `data/arbol_10_nodos.txt` con `Files.readAllLines`.
2. Detección de formato (líneas que contengan `;` → indexado).
3. Parseo y validación por línea:
   - Comprobación de 4 campos por registro.
   - Validación de enteros en `index` y `value`.
   - Detección de índices duplicados.
   - Verificación que índices referenciados como hijos existan.
4. Construcción de nodos `TreeNode<Integer>` en un `Map<index, node>`.
5. Asignación de referencias `left` y `right`.
6. Identificación de raíz (índice no presente en `childIndices`).
7. Construcción de `BinaryTree<Integer>` con la raíz y ejecución de algoritmos:
   - Preorden, Inorden, Postorden: implementación recursiva.
   - Por niveles: implementación iterativa con `Queue`.
8. Cálculo de métricas: `size()` (recursivo) y `height()` (recursivo, altura definida como -1 para árbol vacío).
9. Salida en consola y registro en `logs/salida_en_consola.txt`.

Resultados (salida de consola registrada)

```
Working dir: C:\Users\HP Elite Book\Documents\GitHub\Recorrido-de-
Grafos-APE\Recorrido-de-arboles
Intentando leer archivo: C:\Users\HP Elite
Book\Documents\GitHub\Recorrido-de-Grafos-APE\Recorrido-de-
arboles\data\arbol_10_nodos.txt
Líneas leídas: 11
1: 10
2: 0;50;1;2
3: 1;30;3;4
4: 2;70;5;6
5: 3;20;-1;7
6: 4;40;-1;-1
7: 5;60;8;-1
8: 6;80;-1;9
9: 7;25;-1;-1
```

```
10: 8;55;-1;-1
Formato detectado: indexado (usar fromIndexedFile).

=== Árbol (visual) ===
│            ┌── 90
│        ┌── 80
│    ┌── 70
│    │   └── 60
│    │       └── 55
└── 50
  │   ┌── 40
  └── 30
      │   ┌── 25
      └── 20

=== Recorridos ===
Preorden: [50, 30, 20, 25, 40, 70, 60, 55, 80, 90]
Inorden:  [20, 25, 30, 40, 50, 55, 60, 70, 80, 90]
Postorden:[25, 20, 40, 30, 55, 60, 90, 80, 70, 50]
Por niveles: [50, 30, 70, 20, 40, 60, 80, 25, 55, 90]

=== Métricas ===
Tamaño: 10
Altura: 3
```

Interpretación de resultados

- **Estructura del árbol:** raíz con valor `50` (índice 0). Hijos y subárboles están correctamente enlazados según índices del archivo.
- **Preorden (raíz-izq-der):** `[50, 30, 20, 25, 40, 70, 60, 55, 80, 90]` → coincide con recorrido esperado del árbol construido.
- **Inorden (izq-raíz-der):** `[20, 25, 30, 40, 50, 55, 60, 70, 80, 90]` → corresponde a recorrido que ordena los valores en orden ascendente para este árbol (es un BST en valores).
- **Postorden (izq-der-raíz):** `[25, 20, 40, 30, 55, 60, 90, 80, 70, 50]` → válido y consistente con la estructura.
- **Por niveles (BFS):** `[50, 30, 70, 20, 40, 60, 80, 25, 55, 90]` → recorrido por amplitud esperado.
- **Métricas:**
  - `Tamaño = 10` nodos (confirmado).
  - `Altura = 3` (definición usada: altura del árbol en niveles con raíz = 0; si se usa convención distinta, ajustar).

Complejidad algorítmica (resumen)

- Recorridos pre/in/post (recursivos): **O(n)** tiempo, **O(h)** espacio en pila (h = altura).
- Recorrido por niveles (BFS con cola): **O(n)** tiempo, **O(w)** espacio en cola (w = máxima anchura).
- Construcción desde archivo indexado: **O(n)** tiempo y espacio, asumiendo validaciones y creación de nodos.

Validaciones y manejo de errores

- Se detectan y reportan:

- Líneas con menos de 4 campos.
- Índices duplicados.
- Valores no enteros en `index` o `value`.
- Referencias a hijos inexistentes (se lanza `IOException` con mensaje claro).
- Advertencia si el conteo declarado en primera línea no coincide con registros leídos.
- En caso de archivo vacío, se devuelve un `BinaryTree` vacío.

Recomendaciones y mejoras futuras

- Generalizar el parser con `Function<String,T>` para soportar tipos distintos a `Integer`.
- Encapsular `TreeNode` (atributos `private`) y exponer getters/setters si se requiere control adicional.
- Añadir pruebas unitarias (JUnit) automatizadas que cubran:
- Carga correcta del archivo indexado.
- Recorridos con resultados esperados.
- Casos malformados y comportamiento ante errores.
- Incluir un `pom.xml` o `build.gradle` para integración y ejecución reproducible.
- Registrar salidas con un logger (por ejemplo `log4j` o `java.util.logging`) en lugar de `System.out` para mayor control sobre `logs/`.

Instrucciones de ejecución (resumen)

- Desde IntelliJ:
  - Abrir la carpeta raíz del proyecto.
  - Configurar SDK (JDK 11+).
  - Crear Run Configuration con main `ed.u3.App`, working dir = raíz del proyecto y argumento opcional `data/arbol_10_nodos.txt`.
  - Ejecutar y revisar la consola; la salida también se guarda en `logs/salida_en_consola.txt`.
- Desde terminal: `bash  javac -d out src/main/java/ed/u3/**/*.java  java -cp out ed.u3.App data/arbol_10_nodos.txt`

Anexos

- Archivo de entrada usado: `data/arbol_10_nodos.txt` (ejemplo incluido en el repositorio).
- Registro de salida: `logs/salida_en_consola.txt`.

Conclusión

La implementación cumple los objetivos: carga confiable desde archivo indexado, recorridos correctos y salidas verificables. Las validaciones incluidas evitan referencias inválidas y permiten identificar y corregir errores de formato en los datos de entrada. Se proponen mejoras orientadas a generalización de tipos y automatización de pruebas.