Módulo 4 – Árboles y árboles binarios de búsqueda

Curso de Introducción a Estructuras de Datos y Algoritmos

21 de agosto de 2025

Contents

1 Módulo 4 – Árboles: Estructurando la Jerarquía

1.1 Motivación: De lo Lineal a lo Jerárquico

En los módulos anteriores, exploramos estructuras de datos **lineales** como listas, pilas y colas. Son como perlas en un hilo: cada elemento tiene un ”siguiente” y, a veces, un ”anterior”. Este modelo es perfecto para secuencias, pero el mundo real rara vez es tan simple.

Pensemos en:

- Un **organigrama de empresa**: un director tiene varios gerentes a su cargo, y cada gerente tiene sus propios equipos.
- Un **sistema de archivos**: una carpeta contiene otras carpetas y archivos.
- La **taxonomía biológica**: Reino, Filo, Clase, Orden...

Intentar modelar estas relaciones con una lista sería torpe y antinatural. Perderíamos la información esencial de la **jerarquía**. Aquí es donde los **árboles** entran en juego. No son solo una estructura de datos más, sino un salto conceptual que nos permite representar la **profundidad, la anidación y las relaciones padre-hijo** que definen los sistemas complejos.

Los árboles nos ofrecen un lenguaje para organizar la información de una manera que refleja su estructura inherente, permitiendo operaciones increíblemente eficientes que serían imposibles en un modelo lineal.

1.2 Conceptos Fundamentales de los Árboles

Un **árbol** es una colección de **nodos** conectados por **aristas** de forma jerárquica. A diferencia de los grafos (que veremos más adelante), un árbol no puede contener ciclos y siempre hay un único camino entre dos nodos cualesquiera.

1.2.1 Anatomía de un Árbol

Para hablar de árboles, necesitamos un vocabulario común.

- **Raíz (Root)**: El nodo origen de todo, el único que no tiene padre. Es el ancestro común de todos los demás nodos.
- **Nodo (Node)**: Cada uno de los elementos del árbol. Contiene un dato y punteros a sus hijos.
- **Arista (Edge)**: La conexión entre un nodo padre y un nodo hijo.
- **Hijo (Child)**: Un nodo que desciende directamente de otro.
- **Padre (Parent)**: El nodo del que desciende directamente otro.
- **Hoja (Leaf)**: Un nodo que no tiene hijos. Son los ”finales” del árbol.
- **Nodo Interno**: Cualquier nodo que no es ni raíz ni hoja (aunque a veces se incluye la raíz si tiene hijos).
- **Profundidad (Depth)**: La longitud del camino (número de aristas) desde la raíz hasta un nodo específico. La profundidad de la raíz es 0.
- **Altura (Height)**: La longitud del camino más largo desde un nodo hasta su hoja más lejana. La **altura del árbol** es la altura de su nodo raíz.
- **Subárbol (Subtree)**: Un nodo y todos sus descendientes forman un subárbol, que es un árbol válido en sí mismo.

1.3 El Árbol Binario: Una Simplificación Poderosa

Aunque un nodo en un árbol general puede tener cualquier número de hijos, una de las variantes más estudiadas y utilizadas es el **árbol binario**, donde cada nodo tiene, como máximo, **dos hijos**: un hijo izquierdo y un hijo derecho.

Esta restricción no es una limitación, sino una especialización que simplifica enormemente los algoritmos y permite estructuras de datos muy eficientes.

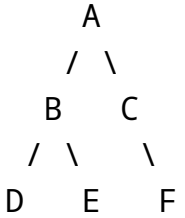
1.3.1 Tipos de Árboles Binarios

No todos los árboles binarios son iguales. Su forma (o *topología*) tiene un gran impacto en su eficiencia.

- **Árbol Binario Lleno**: Cada nodo tiene 0 o 2 hijos. No hay nodos con un solo hijo.
- **Árbol Binario Completo**: Todos los niveles están completamente llenos, excepto posiblemente el último, que se llena de izquierda a derecha. Esta estructura es ideal para ser almacenada en un array de forma compacta.
- **Árbol Binario Perfecto**: Un árbol lleno donde todas las hojas están en el mismo nivel. Representa la máxima "densidad" de nodos para una altura dada. Un árbol perfecto de altura h tiene exactamente $2^{h+1} - 1$ nodos.
- **Árbol Degenerado o Sesgado**: Cada nodo padre tiene un solo hijo. Se comporta exactamente como una **lista enlazada**, perdiendo todas las ventajas de un árbol. Es el peor caso para un árbol de búsqueda.

1.3.2 El Propósito de los Recorridos

Recorrer un árbol significa visitar cada uno de sus nodos en un orden específico. Este orden no es arbitrario; cada tipo de recorrido tiene un propósito fundamental. Dado este árbol de ejemplo:



1. **Preorden (Raíz -> Izquierda -> Derecha)**: `A, B, D, E, C, F`
    - **Propósito**: Se usa para **copiar o serializar un árbol**. Al procesar la raíz primero, podemos recrear la estructura de forma unívoca. También se utiliza en árboles de expresión para obtener la notación prefija (Notación Polaca).
2. **Inorden (Izquierda -> Raíz -> Derecha)**: `D, B, E, A, C, F`
    - **Propósito**: Es la joya de la corona en los **Árboles Binarios de Búsqueda**

(BST). Un recorrido inorden de un BST siempre devuelve sus elementos **en orden ascendente**. Es la forma más natural de "aplanar" un BST en una secuencia ordenada.

3. **Postorden (Izquierda -> Derecha -> Raíz)**: `D, E, B, F, C, A`
    - **Propósito**: Su uso principal es para **eliminar un árbol de la memoria**. Se asegura de que los hijos de un nodo sean eliminados antes que el propio nodo, evitando dejar punteros huérfanos. También se usa para evaluar árboles de expresión (notación postfija o RPN).

1.4 Árboles Binarios de Búsqueda (BST)

Un BST impone una regla fundamental sobre un árbol binario: la **propiedad de orden del BST**.

Para cualquier nodo `N`:

- Todos los valores en el subárbol izquierdo de `N` deben ser **menores** que el valor de `N`.
- Todos los valores en el subárbol derecho de `N` deben ser **mayores** que el valor de `N`.

1.4.1 La Gran Idea: Dividir para Conquistar

Esta simple regla es la que le da al BST su poder. Cada vez que comparamos una clave con un nodo, podemos **descartar la mitad del árbol restante**. Si la clave que buscamos es menor que el nodo actual, sabemos con certeza que *no* puede estar en el subárbol derecho, y viceversa.

Este es el mismo principio de la **búsqueda binaria** en un array ordenado, pero aplicado a una estructura de datos dinámica que permite inserciones y eliminaciones eficientes. Mientras el árbol esté razonablemente **balanceado**, las operaciones principales (búsqueda, inserción, eliminación) tienen una complejidad temporal de $O(\log n)$.

1.4.2 Operaciones Detalladas

1.4.2.1 Inserción y Búsqueda Las operaciones de búsqueda e inserción son recursivas y elegantes. Se desciende por el árbol, tomando una decisión en cada nivel (ir a la izquierda o a la derecha) hasta que se encuentra el elemento o se llega a un punto nulo (`null`), donde se insertaría el nuevo nodo.

1.4.2.2 Análisis Detallado de la Eliminación

La eliminación es la operación más compleja porque debemos preservar la propiedad del BST después de quitar un nodo.

1. **Caso 1: El nodo a eliminar es una hoja**.
   - **Solución**: Es el caso más simple. Simplemente se elimina el nodo y se actualiza el puntero del padre a `null`.
2. **Caso 2: El nodo a eliminar tiene un solo hijo**.
   - **Solución**: Se ”salta” el nodo. El padre del nodo eliminado pasa a apuntar directamente al único hijo de este.
3. **Caso 3: El nodo a eliminar tiene dos hijos**.
   - **El Problema**: No podemos simplemente eliminarlo, ya que dejaríamos dos subárboles ”huérfanos”.
   - **La Solución Elegante**: a. No eliminamos el nodo físicamente. En su lugar, buscamos un sustituto que mantenga la propiedad del BST. b. Este sustituto puede ser:
     - El **sucesor inorden**: El nodo más pequeño en el subárbol derecho.
     - O el **predecesor inorden**: El nodo más grande en el subárbol izquierdo. c. Copiamos el valor del sucesor (o predecesor) al nodo que queremos ”eliminar”. d. Ahora, el problema se reduce a eliminar el nodo sucesor (o predecesor) de su ubicación original, lo cual es garantizado que será un caso más simple (Caso 1 o Caso 2).

1.5 El Talón de Aquiles de los BST: El Desequilibrio

La magia de $O(\log n)$ solo funciona si la altura del árbol (h) es cercana a $\log n$. ¿Qué pasa si insertamos elementos ya ordenados (ej. 10, 20, 30, 40, 50) en un BST?

El resultado es un **árbol degenerado**: una larga cadena de hijos derechos. La altura del árbol se convierte en n, y la búsqueda se degrada a una búsqueda lineal con complejidad $O(n)$, perdiendo toda su ventaja.

1.5.1 La Solución: Árboles Autobalanceados

Para resolver el problema del crecimiento desmesurado de la altura en los **árboles binarios de búsqueda (ABB)**, se inventaron los **árboles autobalanceados**. La idea central es sencilla: tras cada operación de inserción o eliminación, el árbol **verifica si alguna rama ha quedado demasiado “pesada”** en comparación con la otra, y en caso afirmativo **aplica transformaciones locales** para restaurar un equilibrio razonable. Estas

transformaciones se conocen como **rotaciones** (simples o dobles), y permiten reestructurar el árbol sin perder el orden de los elementos.

1.5.1.1 Árboles AVL

- **Condición de equilibrio**: en todo nodo, la altura de los subárboles izquierdo y derecho difiere como máximo en 1.
- **Mantenimiento**: cada nodo suele almacenar su altura o un *factor de equilibrio* (altura izquierda – altura derecha).
- **Corrección**:
  - Si tras una inserción/eliminación el factor de equilibrio sale de {-1, 0, +1}, se aplica una rotación.
  - Hay cuatro casos clásicos: **rotación simple a la derecha (LL)**, **rotación simple a la izquierda (RR)**, **rotación doble izquierda-derecha (LR)** y **rotación doble derecha-izquierda (RL)**.
- **Ventaja**: búsquedas extremadamente rápidas, ya que la altura del árbol es casi óptima.
- **Desventaja**: inserciones y eliminaciones más costosas porque requieren recalcular alturas y, a menudo, rotar.
- **Uso típico**: sistemas en los que la **lectura/búsqueda** es muchísimo más frecuente que las modificaciones (por ejemplo, índices de bases de datos muy consultados).

1.5.1.2 Árboles Rojo-Negro

- **Idea central**: cada nodo tiene un color (rojo o negro) y se cumplen unas propiedades de color que limitan cuánto puede desequilibrarse el árbol.
- **Propiedades clave**:
  1. La raíz siempre es negra.
  2. Ningún nodo rojo puede tener un hijo rojo.
  3. Todo camino desde un nodo hasta una hoja nula contiene el mismo número de nodos negros.
- **Altura garantizada**: se demuestra que el camino más largo no puede ser más del doble que el más corto altura $O(\log n)$.
- **Corrección**: cuando una operación viola las propiedades de color, se arregla con una combinación de **cambios de color** y **rotaciones**.
- **Ventaja**: menos rotaciones en promedio que los AVL operaciones de inserción y borrado más rápidas.

- **Desventaja**: búsquedas ligeramente más lentas que en AVL, porque el equilibrio no es tan perfecto.
- **Uso típico**: estructuras estándar de bibliotecas (como `std::map`, `std::set` en C++, o `TreeMap` en Java) y sistemas donde se necesita un buen equilibrio entre inserciones y búsquedas.

1.5.1.3 Resumen Ambos árboles usan el mismo "truco": **rotaciones locales** para mantener el árbol con altura logarítmica.

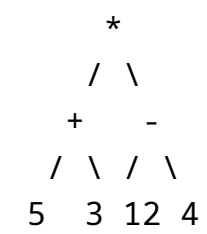
La diferencia está en el grado de perfeccionismo:

- AVL = equilibrio estricto, ideal si buscas rapidez de acceso.
- Rojo-Negro = equilibrio flexible, ideal en entornos con muchas modificaciones.

1.6 Caso de Estudio: Árboles de Expresión Aritmética

Una aplicación clásica de los árboles binarios es la representación de expresiones matemáticas.

La expresión `(5 + 3) * (12 - 4)` puede ser representada por el siguiente árbol:



- Los **nodos internos** son operadores (`*`, `+`, `-`).
- Las **hojas** son los operandos (los números).

¿Qué ocurre si recorremos este árbol?

- **Recorrido Preorden**: `* + 5 3 - 12 4` (Notación Prefija).
- **Recorrido Inorden**: `5 + 3 * 12 - 4` (Notación Infija - necesita paréntesis para ser correcta).
- **Recorrido Postorden**: `5 3 + 12 4 - *` (Notación Postfija o RPN).

Para **evaluar la expresión**, realizamos un recorrido postorden. Cuando visitamos un nodo operador, aplicamos la operación a los resultados de haber visitado sus hijos izquierdo y derecho.

1.6.1 Uso en calculadoras y contexto histórico

En las **calculadoras clásicas de los años 60–70**, el gran problema era **cómo evaluar expresiones complejas con recursos muy limitados**: poca memoria, procesadores lentos y sin capacidad de manejar paréntesis ni reglas complicadas de precedencia de operadores.

Hasta entonces, las calculadoras de mesa (y muchas electrónicas primitivas) funcionaban casi como sumadoras: metías un número, dabas a `+`, luego otro número, y así sucesivamente. Resolver algo como `(5 + 3) * (12 - 4)` requería hacerlo "a mano" en varios pasos, porque la máquina no sabía **respetar prioridades ni agrupar operaciones**.

El hito llegó cuando se introdujo el uso de **árboles de expresión → notación postfija (RPN, Reverse Polish Notation)**:

1. En lugar de tener que analizar paréntesis y precedencias, todo se reducía a **una lista lineal de instrucciones fáciles de ejecutar con una pila**.
2. Las calculadoras HP (como la mítica HP-35 en 1972) se apoyaron en este sistema: tú introducías números y operaciones en RPN, y la máquina los evaluaba directamente con una pila interna.
3. Esto eliminaba la necesidad de un analizador complejo, ahorraba memoria y chips, y además permitía al usuario **encadenar cálculos mucho más complejos** sin volverse loco con paréntesis.

Por qué fue un hito:

- Permitió que calculadoras relativamente "baratas" y con hardware limitado pudieran resolver expresiones complejas.
- Introdujo un modelo (RPN basado en árboles de expresión) que luego se generalizó en lenguajes de programación, compiladores y procesadores.
- Dio a HP y a otras marcas pioneras una ventaja brutal en el mercado, porque sus máquinas podían hacer "matemáticas serias" en el bolsillo.

En resumen: el uso de árboles de expresión y su traducción a notación postfija **transformó una limitación tecnológica en una solución elegante** que marcó la diferencia entre una calculadora que solo sumaba y una que ya parecía un pequeño ordenador.

1.7 Más Allá de los Binarios: Un Vistazo al Ecosistema de Árboles

Aunque los BST son fundamentales, no son el final del camino. Dependiendo del problema, se usan otras variantes:

- **Árboles B / B+**: Son árboles de búsqueda no binarios (un nodo puede tener muchos hijos). Son la base de datos de los **índices de las bases de datos** y los **sistemas de archivos**. Minimizan las lecturas de disco al ser anchos y poco profundos.
- **Tries (Árboles de Prefijos)**: Estructuras especializadas para almacenar y buscar cadenas de texto. Son la base de las funciones de **autocompletado** de los buscadores y editores de texto.
- **Heaps (Montículos)**: Son árboles binarios con una propiedad de orden diferente (el padre siempre es mayor/menor que sus hijos), usados para implementar **colas de prioridad**.

1.8 Ejercicios Ampliados

1.8.0.1 Ejercicios Teóricos y de Diseño

1. Dibuja el BST resultante de insertar los números `50, 25, 75, 10, 30, 60, 80, 5, 15, 65, 85`. ¿Cuál es la altura del árbol?
2. Sobre el árbol anterior, muestra los pasos para eliminar el nodo `25`. Luego, muestra los pasos para eliminar el nodo `50`.
3. ¿Por qué un recorrido inorden en un BST produce una secuencia ordenada? Explica cómo la propiedad del BST lo garantiza.
4. Describe una situación en la que un árbol Rojo-Negro sería preferible a un árbol AVL, y viceversa.
5. Implementa en pseudocódigo la función `findMin()` y `findMax()` en un BST de forma iterativa y recursiva.

1.8.0.2 Ejercicios Prácticos de Programación

6. Escribe una función que determine si un árbol binario dado es un BST válido.
7. Implementa una función que convierta un array ordenado en un BST perfectamente balanceado.
8. Escribe un programa que construya un árbol de expresión a partir de una cadena en notación postfija y luego lo evalúe.

1.9 Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms*. MIT Press.
- Weiss, M. A. *Data Structures and Algorithm Analysis*. Pearson.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. *Data Structures and Algorithms in Java*. Wiley.

- Sedgewick, R., & Wayne, K. *Algorithms.* Addison-Wesley.