TRABAJO PRÁCTICO INTEGRADOR

“ESTRUCTURA DE DATOS:
ÁRBOLES BINARIOS”

ALUMNOS:
ALBRIGI MARIANELA
BONANNO NICOLÁS

TECNICATURA UNIVERSITARIA EN PROGRAMACION
UNIVERSIDAD TECNOLOGICA NACIONAL

PROGRAMACIÓN 1

Docente Titular
Cinthia Rigoni

Docente Tutor
Oscar Londero

9 de JUNIO de 2025

TABLA DE CONTENIDO

ESTRUCTURA DE DATOS: ÁRBOLES

En el presente trabajo práctico se decidió abordar el tema de árboles como estructura de datos, dado que estos son considerados un elemento clave en la programación y en la informática en general.

La propia naturaleza de su organización jerárquica agiliza operaciones fundamentales como búsqueda, inserción y borrado, y permite modelar naturalmente dominios tan variados como sistemas de archivos, compiladores, motores de bases de datos o algoritmos de inteligencia artificial. Aunque estos usos pueden parecer muy técnicos o alejados, las estructuras de árboles también están presentes en tareas cotidianas y aplicaciones de uso diario, como la organización de carpetas en el Explorador de Windows o la gestión de elementos y decisiones en videojuegos. Esto demuestra lo esencial y versátil que resulta entender su funcionamiento, tanto para proyectos complejos como para herramientas de uso cotidiano.

El trabajo se enfoca en presentar los conceptos principales para comprender qué es una estructura de árbol, conocer sus elementos, propiedades y los diferentes tipos de estructuras de árbol. En particular, se profundiza en una estructura especial: los árboles binarios. Se analizan sus características distintivas y se exploran a través de un caso práctico, mediante el cual se reafirmarán los conceptos teóricos previamente expuestos. Para ello, se recreará una estructura de árbol binario usando listas anidadas en Python, lo que permitirá integrar contenidos clave de la materia, como la manipulación de listas y la recursividad. De este modo, se establece una conexión directa entre teoría y práctica.

DEFINICIONES INICIALES: estructuras en Programación

Antes de abordar directamente el tema de este trabajo práctico, se considera necesario hacer un breve repaso de ciertos conceptos y definiciones ya trabajados a lo largo de la materia Programación I. Esto servirá para comprender mejor una nueva categoría de estructura de datos: la estructura tipo árbol.

Una ***estructura de datos*** es la forma en la que se almacenan o se estructuran los datos. La forma en la que se organizan los datos influye en la forma en que estos se relacionan entre sí, el espacio que ocupan en la memoria y la eficiencia al agregarlos o buscarlos. Cada estructura tiene ventajas y desventajas, por eso es importante conocerlas y elegir la más adecuada para cada situación o necesidad.

ESTRUCTURAS DE DATOS LINEALES VS NO LINEALES

- Una ***estructura de datos lineal*** almacena los elementos en orden secuencial, donde cada elemento tiene su antecesor y sucesor, exceptuando el primero y el último. Para acceder a un elemento, es necesario recorrer la estructura desde el inicio, elemento por elemento, de forma secuencial hasta encontrar el elemento buscado. Ejemplos: listas, tuplas o arrays, etc.
- La ***estructura de datos no lineales***, en cambio, almacenan datos en un orden jerárquico. Los elementos pueden tener múltiples conexiones entre sí. A diferencia de la estructura lineal, se puede acceder a un cualquier elemento sin tener que recorrer la estructura desde principio. Algunos ejemplos incluyen árboles o grafos.

ESTRUCTURAS DE DATOS ESTÁTICAS VS NO ESTÁTICAS

- En una ***estructura de datos estática,*** su tamaño se define al momento de crearla y no puede modificarse mientras el programa está en ejecución; esto hace que la ubicación en memoria de los elementos también esté definida antes de la ejecución y, por lo tanto, sea más rápido acceder a sus elementos. Ejemplos: arrays, bibliotecas, etc.
- En la ***estructura de datos dinámica*** no hay que definir de antemano el tamaño, por lo que puede cambiar durante la ejecución del programa. Esto permite la adición o eliminación de elementos según sea necesario. Ejemplo: listas, diccionarios, etc.

De acuerdo con lo descripto anteriormente, se define a ***una estructura de tipo árbol*** como una estructura no lineal y dinámica. Eso significa que la estructuración de los elementos no es secuencial sino jerárquica y que su tamaño puede variar. Su estructura permite recorrerla de diferentes formas y además realizar operaciones básicas como agregar, eliminar y localizar elementos específicos.

LA ESTRUCTURA DE ÁRBOL: UN TIPO DE GRAFO

Para empezar a entender los elementos que componen un árbol, es necesario saber qué es un grafo, ya que un árbol es un tipo de grafo. Este término proviene de la **teoría de grafos,** una rama de las matemáticas y la informática, que estudia cómo se relacionan los elementos dentro de un grafo. Un grafo es una estructura formada por **nodos** (vértices) y las conexiones entre ellos, que se llaman **aristas** (ramas)**.**

En función del tema de este trabajo práctico, se considerarán algunos conceptos de esta teoría:

- **Grafos**: una colección de nodos y aristas que están interconectados.
- **Nodos (Vértices)**: representan una entidad u objeto dentro del grafo, es decir, dentro del sistema que se pretende modelar. Pueden representar una ciudad en un mapa, un usuario en una red social o puntos en un conjunto numérico.
- **Aristas**: son las conexiones entre los nodos. Las aristas pueden ser:
    - **Direccionales (unidireccionales):** significa que la relación entre nodos tiene un solo sentido. Por ejemplo, existen dos nodos que representan a dos usuarios en una red social. Si solo de uno está siguiendo al otro, significa que la relación es direccional o unidireccional.
    - **No direccionales (bidireccionales):** es aquel en el que las conexiones no tienen dirección. Es decir, la relación es bidireccional, va en ambos sentidos.
      Tomando el ejemplo anterior, la dirección se convierte en bidireccional cuando los usuarios se siguen mutuamente.

  El tipo de aristas determina el tipo de grafo:

    - **Grafo dirigido:** cuando todas las aristas son unidireccionales.

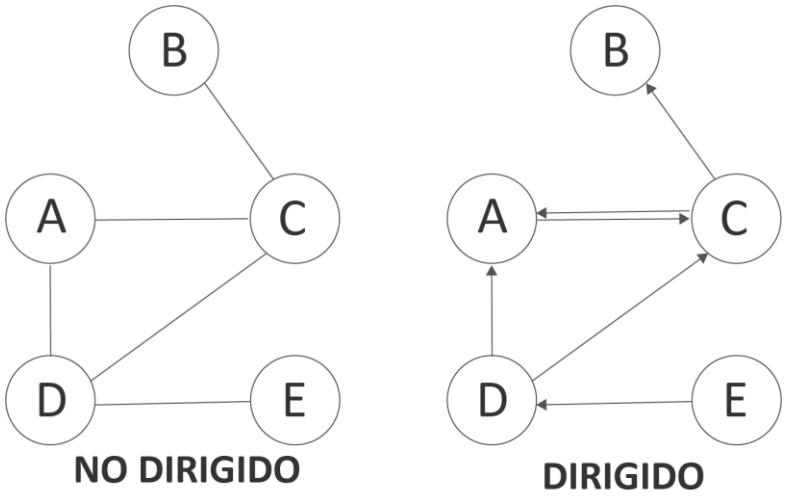
- **Grafo no dirigido:** cuando todas las aristas son bidireccionales.
- **Grafo mixto:** cuando hay una combinación de aristas.

- **Grado de un nodo:** es la cantidad de aristas que están conectadas a un nodo.
  - En un **grafo dirigido**, se distinguen dos cosas:

    - Grado de entrada (in-degree): cuántas aristas llegan al nodo.

    - Grado de salida (out-degree): cuántas aristas salen del nodo.

  - En un **grafo no dirigido**: el grado es simplemente **cuántas conexiones tiene el nodo** (sin importar sentido, porque no hay dirección).



Además de la clasificación ***grafos dirigidos, no dirigidos o mixtos,*** existen otras muchas formas de clasificar los grafos, pero, a continuación, solo se abordarán específicamente aquellas que permiten comprender mejor qué es un árbol como estructura de datos. En este sentido, podemos distinguir grafos según:

La presencia de ciclos:

- **Cíclicos:** cuando contiene al menos un ciclo, es decir, se trata de un camino cerrado donde el último nodo está conectado al primer nodo, creando una secuencia de aristas que regresa al punto de partida.
- **No cíclicos:** es aquel que no tiene ningún ciclo. Es decir, no es posible comenzar en un nodo y regresar a él siguiendo una secuencia de aristas sin repetir nodos.

La conectividad entre nodos:

- **Conexo:** todos los nodos están conectados de alguna forma, es decir, hay al menos un camino entre cualquier par de nodos.
- **No conexo:** hay nodos o grupos de nodos que no tienen conexión con otros.

Teniendo en cuenta lo descripto anteriormente, es posible determinar que un ***árbol*** es un tipo de grafo acíclico, lo cual significa que no contiene ciclos ni bucles y conexo, lo que indica que hay un camino entre cualquier par de nodos en la estructura.

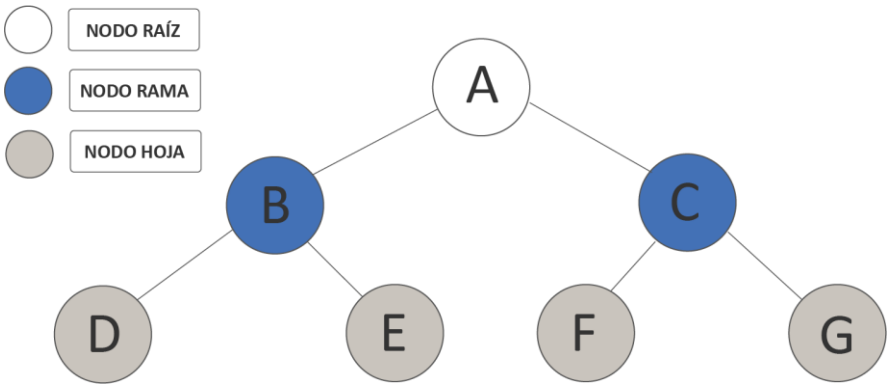
ELEMENTOS DE LA ESTRUCTURA DE ÁRBOL

Un árbol este compuesto por:

1) **nodos** (vértices) conectados entre sí y que almacenan la información.
2) **aristas** (ramas) que conectan o establecen las relaciones entre nodos.

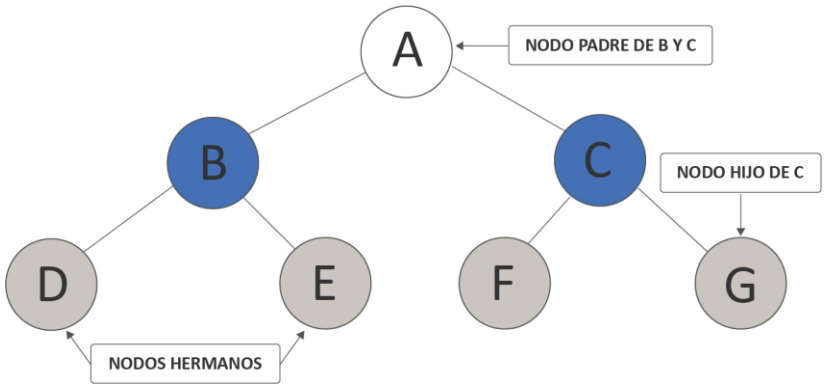
Según su posición los nodos pueden ser:

- **Nodo Raíz (root):** es el primer nodo de una estructura y es único, ya que un árbol puede tener solo un nodo raíz.
- **Nodo Rama:** Es cualquier nodo que tenga un padre y al menos un hijo, este nodo no es terminal, sigue propagando la jerarquía.
- **Nodo Hoja:** Es aquel nodo que se denomina nodo terminal del árbol, no tiene hijos y se encuentra en los extremos de la estructura.



A su vez, en su relación con otros nodos, se clasifican en:

- **Nodo Padre:** son todos aquellos nodos que tiene al menos un hijo. Cada nodo, a excepción del nodo raíz, tiene al menos un nodo padre. Es decir, un nodo padre es el antecesor inmediato de uno o más nodos.
- **Nodo Hijo:** es un nodo que depende directamente de otro nodo superior, al cual llamamos nodo padre. Es decir, es un nodo que nace de un padre y a su vez puede tener nodos hijos y convertirse en padre. Todo nodo, excepto el nodo raíz, es hijo de algún otro nodo.
- **Nodo Hermano:** es el caso de dos o más nodos son hermanos que comparten el mismo nodo padre. Esto significa que están al mismo nivel jerárquico y provienen del mismo nodo superior. Es una relación horizontal dentro del mismo conjunto de hijos.



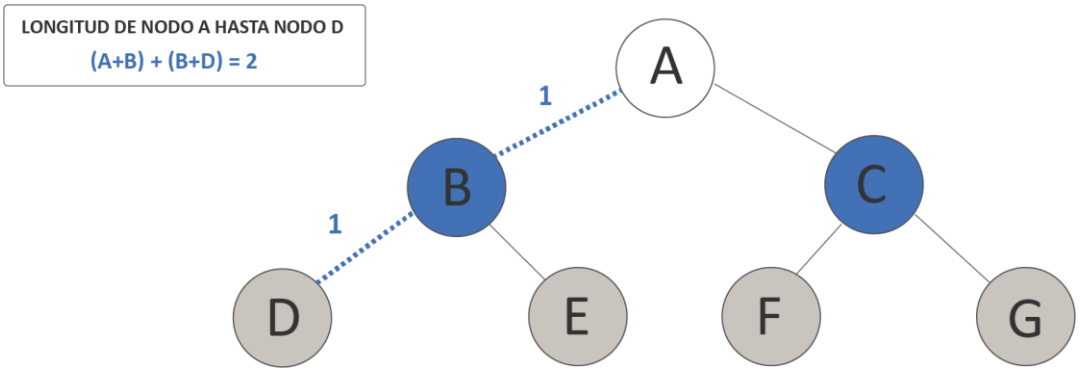
OTRAS PROPIEDADES DE LOS ÁRBOLES

Cada árbol tiene características que determinan su estructura, su forma y su comportamiento. A través de estas, se puede diferenciar un tipo de árbol de otro, entender cómo se relacionan los nodos y aplicar algoritmos según las necesidades del problema.

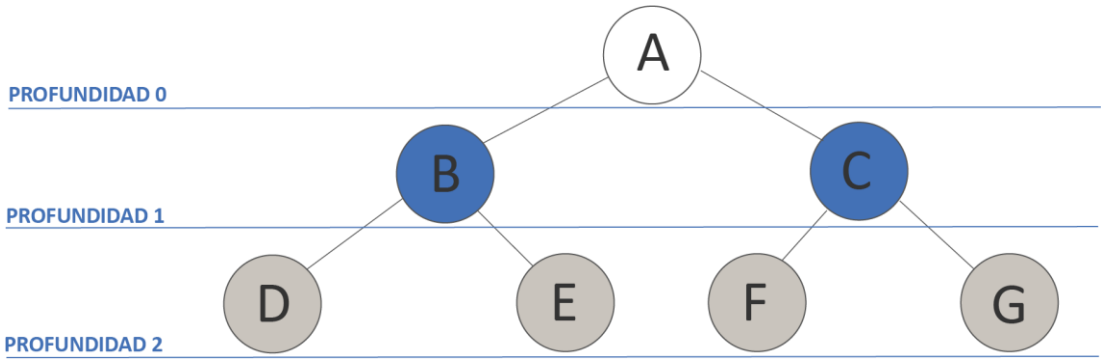
1. **Longitud:** se refiere al número total de aristas (o conexiones) entre dos nodos a lo largo de un camino específico. Normalmente, este camino se traza desde un nodo de nivel superior hacia un nodo descendiente. La longitud es usada, por ejemplo, para calcular distancias relativas dentro del árbol o para comparar rutas en recorridos. Comprender

la longitud ayuda a definir y analizar algoritmos de recorrido, como búsqueda por profundidad o por anchura.

Ejemplo: En un árbol donde A es la raíz y D es un descendiente con el camino A→B→C→D, la longitud de la ruta entre A y D es **3** (porque hay 3 aristas en ese camino).



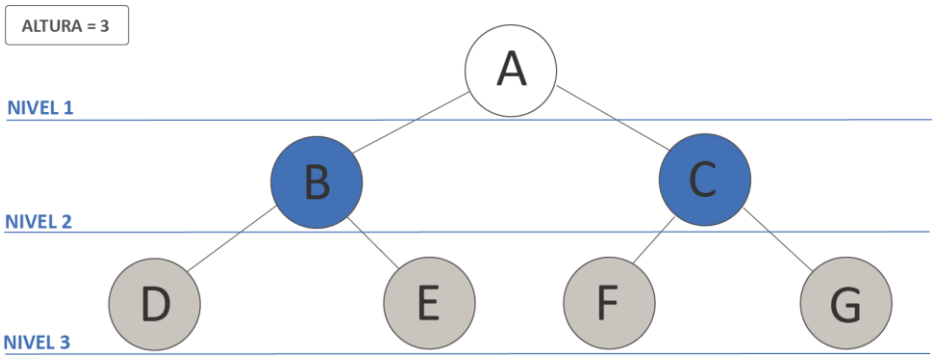
2. **Profundidad:** es la cantidad de aristas que existen desde el nodo raíz hasta un nodo x elegido. Por lo que, describe la cantidad de pasos que deben hacerse para llegar allí. Podría interpretarse como un valor que indica cuan abajo se encuentra un nodo dentro del árbol, tomando como referencia que ***el nodo raíz está en el nivel tiene profundidad cero***, ya que es el punto de partida.



3. **Nivel:** el nivel de un nodo representa su posición jerárquica dentro del árbol. Aunque muchas veces se usa como sinónimo de profundidad, el nivel es una etiqueta jerárquica, más que una medida de distancia. En un árbol la raíz representa el nivel 1, aumenta en uno con cada nivel descendente. Por ejemplo, si al final de una estructura le adherimos

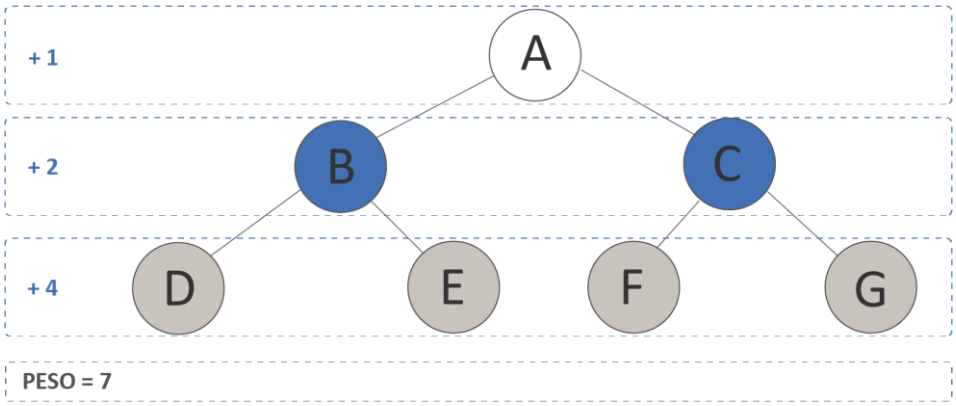
un nodo hijo a un nodo que antes no tenia, ese nodo rama pasa a ser nodo padre. Con la adjudicación de un nodo hijo, la estructura "tiene una generación más", es decir, un nivel más. Por lo que cada generación nueva, genera que la altura de la estructura sea un nivel más superior. ***Altura*** es entonces el número máximo de niveles de un árbol.

Con respecto a su importancia podemos decir que la profundidad es esencial para calcular la eficiencia de recorridos y para entender estructuras como montículos, árboles AVL y árboles binarios balanceados.



4. **Peso del árbol**

El peso (o tamaño) de un árbol es la cantidad total de nodos que contiene. Es una propiedad global que refleja la magnitud completa de la estructura. Este factor es importante porque nos da una idea del tamaño del árbol y el tamaño en memoria que nos puede ocupar en tiempo de ejecución. El peso permite dimensionar el árbol y calcular la complejidad espacial.



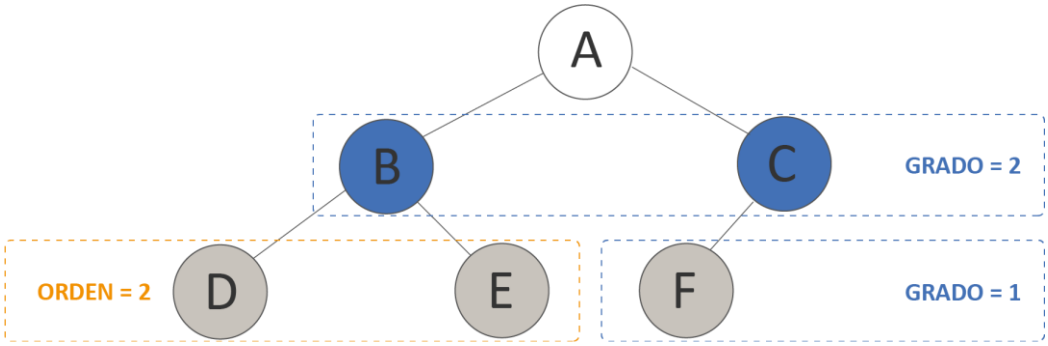
En el caso de los árboles con estructura ponderada, es decir, donde las aristas (conexiones entre nodos) tienen peso o costo (valor numérico asignado), el peso surge de la suma de esos valores que fueron asignados a cada arista. Este peso resultante puede representar distancia entre nodos, tiempo de recorrido, costo económico, cantidad de datos transferidos, etc. Con esta información se podría comparar árboles y elegir el más eficiente, aplicar algoritmos de optimización, etc.

5. **Grado:** El grado de un nodo es el número de hijos directos que posee. El grado del árbol es el valor máximo de grado entre todos sus nodos. Esta propiedad refleja el nivel de "ramificación" de cada nodo. El grado sirve para diferenciar tipos de árboles:

    - Árbol binario → grado máximo 2
    - Árbol n-ario → grado máximo *n*

    En el caso de árboles binarios, donde cada nodo puede tener al menos un nodo hijo, pero no más de dos, el grado de un nodo sirve para verificar si la estructura es válida.

    Asimismo, el grado de un nodo podría ser utilizado en algoritmos que ayuden a determinar cuántas opciones o caminos hay en un recorrido.



6. **Orden:** El orden de un árbol es el número máximo de hijos que puede tener un nodo, según las reglas de su estructura. A diferencia del grado, el orden no describe la situación real de un nodo, sino la capacidad teórica permitida para cada nodo. El orden es una cualidad que se establece antes de construir el árbol.

El orden define las restricciones estructurales del árbol y permite clasificar diferentes tipos:

- Árbol binario (orden 2)
- Árbol ternario (orden 3)
- Árbol B (orden n)

RESUMEN:

PROPIEDAD	QUÉ MIDE	EJEMPLO
Longitud	N° de aristas entre dos nodos	A → B → C = 2
Profundidad	N° de aristas desde la raíz a un nodo	Nodo en nivel 3 → profundidad 3
Nivel	Posición del nodo en la jerarquía	Raíz = nivel 1
Altura	Máxima distancia del nodo a una hoja	Raíz a hoja más lejana = altura
Peso	Total de nodos en el árbol	7 nodos → peso 7
Grado	N° de hijos de un nodo	Nodo con 2 hijos → grado 2
Orden	Máximo n° de hijos que puede tener un nodo	Árbol binario → orden 2

¿QUÉ SON LOS ARBOLES BINARIOS?

Los árboles binarios son un tipo de estructura de árbol con características que los diferencian de otras formas de organizar los datos. Por un lado, son de grado 2: esto quiere decir que cada nodo puede tener hasta dos hijos, también llamados subárboles. También podría no tener ninguno, y en ese caso se lo denomina nodo hoja, como se explicó anteriormente.

A su vez, cada nodo (excepto la raíz) está conectado únicamente a un nodo padre, lo que garantiza que la estructura no tenga ciclos y se mantenga jerárquica.

Cada nodo del árbol tiene tres partes:

- Un **valor**.

- Un puntero (o referencia) al **hijo izquierdo**.
- Un puntero al **hijo derecho**.

Existen ciertas variantes, como los árboles binarios de búsqueda, que aplican reglas más estrictas: por ejemplo, el hijo izquierdo suele tener un valor menor que su nodo padre, y el hijo derecho, un valor mayor. Este tipo de reglas permiten organizar la información para facilitar búsquedas rápidas.

OPERACIONES CON ESTRUCTURA DE ÁRBOLES BINARIOS

Al igual que en otras estructuras de árbol, en los binarios se pueden realizar distintas operaciones, que son clave para el manejo de los datos:

1. **Inserción**

Consiste en agregar un nodo nuevo respetando el orden del árbol. Por ejemplo, en un árbol de búsqueda binaria (que se explicará su definición más adelante) si el valor es menor que el del nodo actual, se va hacia la izquierda; si es mayor, hacia la derecha. Se repite este proceso hasta encontrar una posición libre.

Una de las ventajas de este tipo de inserción es que, si el árbol está balanceado, la operación se hace rápidamente, sin necesidad de revisar todos los nodos. Sin embargo, muchas inserciones seguidas y no controladas pueden volver al árbol desbalanceado (más cargado de un lado que del otro), casi como una lista. Por eso existen también existen otros árboles binarios como los AVL o los Rojo-Negro, que se encargan de mantener el equilibrio automáticamente después de cada inserción.

2. **Búsqueda**

Se recorre el árbol siguiendo la lógica del orden para encontrar un valor determinado. Como los datos están organizados jerárquicamente, no es necesario recorrer todos los nodos, lo que hace que esta operación sea mucho más eficiente.

La forma en la que esta organizada la información dentro del árbol va a determinar el tipo de búsqueda que realizará y cuan eficiente será.

En los árboles de búsqueda binaria los nodos están ordenados de tal manera que el subárbol izquierdo contiene valores menores al nodo actual y el derecho, mayores. Esto permite que, al

buscar un valor, no sea necesario recorrer todo el árbol, sino que se pueda ir "descartando mitades". El resultado es una búsqueda mucho más rápida.

En árboles que no tienen un orden definido, como algunos árboles generales o binarios no ordenados, la búsqueda no puede aprovechar ninguna lógica de orden.

En ese caso, la búsqueda no resulta tan eficiente como la anterior, ya que se tiene que recorrer nodo por nodo, usando algún tipo de recorrido como el inorden, preorden o postorden.

En resumen, la estructura no solo organiza los datos, sino que también determina cómo se los puede buscar. Si el árbol está bien diseñado y ordenado, la búsqueda va a ser más ágil y va a consumir menos recursos.

3. **Recorridos**

Esta operación consiste en visitar todos los nodos del árbol en cierto orden. Dependiendo del recorrido que se elija, se puede obtener diferente información o utilizarla con distintos fines, como mostrar los datos en cierto orden, copiarlos o buscar un valor.

Tipos de recorridos en árboles binarios:

Recorrido por profundidad:

- **Preorden:** es útil si queremos copiar un árbol o evaluarlo en ciertas estructuras como expresiones matemáticas.
- **Inorden:** en los árboles binarios de búsqueda, este recorrido devuelve los elementos ordenados de menor a mayor, lo que lo hace muy útil para mostrar los datos en orden.
- **Postorden: e**ste recorrido se suele usar cuando se necesita borrar o liberar memoria de abajo hacia arriba, o para calcular resultados acumulativos.

Recorrido por anchura: recorren los nodos por nivel, empezando desde la raíz y bajando capa por capa, de izquierda a derecha. Es útil para representar estructuras como jerarquías o ver qué tan “profundo” está un elemento. También puede servir para encontrar el nodo más cercano a la raíz que cumpla una condición.

Además de estas, hay otras operaciones importantes como:

Eliminación

Eliminar un nodo es una de las operaciones más complejas, ya que hay que hacerlo sin romper la estructura del árbol.

Lo primero a tener en cuenta es que el proceso varía dependiendo del tipo de nodo que se quiera eliminar:

- **Si el nodo no tiene hijos** (es una hoja): entonces se puede eliminar directamente, sin complicaciones.
- **Si el nodo tiene un solo hijo** (izquierdo o derecho): se reemplaza el nodo que se elimina por su hijo.
- **Si el nodo tiene dos hijos**: esta es la situación más compleja. Lo que se debe hacer es buscar el nodo más pequeño del subárbol derecho (sucesor inorden) o buscar el nodo más grande del izquierdo (predecedor inorden), se copia ese valor en el nodo que se quiere eliminar, y luego se elimina ese otro nodo (que sí tendría solo uno o ningún hijo).

Altura del árbol

Aunque ya se explicó en párrafos anteriores esta propiedad, vale la pena recordar que la altura se determina por la cantidad de niveles o el número máximo de pasos desde la raíz hasta una hoja. Esta característica ayuda a evaluar qué tan balanceado o profundo está el árbol.

La altura de un árbol está directamente relacionada con el rendimiento de ciertas operaciones: a mayor altura, mayor será el tiempo que puede tardar una búsqueda o inserción, porque hay más niveles que recorrer.

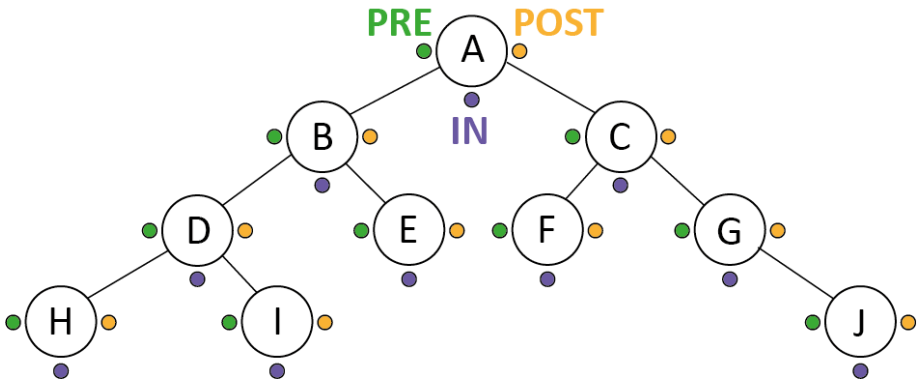
Por eso es tan importante mantener los árboles balanceados, ya que su altura se mantiene lo más baja posible, garantizando un mejor rendimiento. De hecho, muchos árboles (como los AVL) tienen mecanismos automáticos para mantener su altura controlada luego de inserciones o eliminaciones.

TIPOS DE RECORRIDO: PRE-IN-POST orden

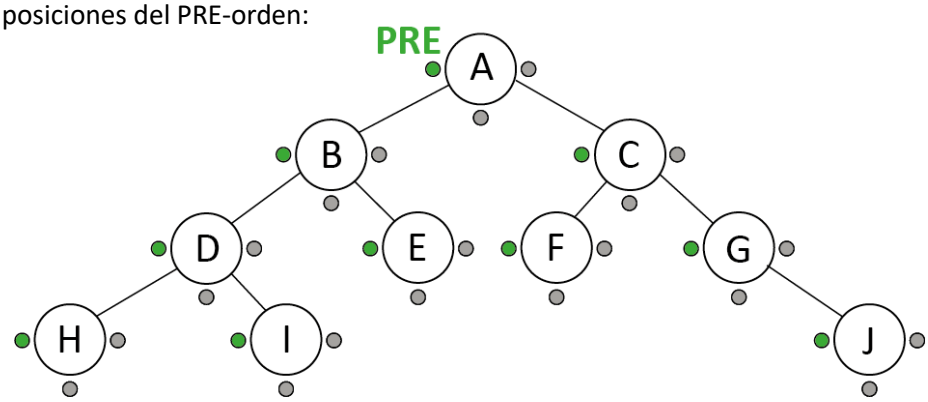
Tal como se definió anteriormente, las estructuras de árbol organizan sus datos de forma no lineal, por lo que la forma de recorrerlos difiere de la forma secuencial de las estructuras lineales

como las listas. Teniendo en cuenta esto, se pueden recorrer a los árboles binarios en tres sentidos: Pre-orden, In-orden y Post-orden. A continuación, se explicará cómo identificar y reconocer cada recorrido:

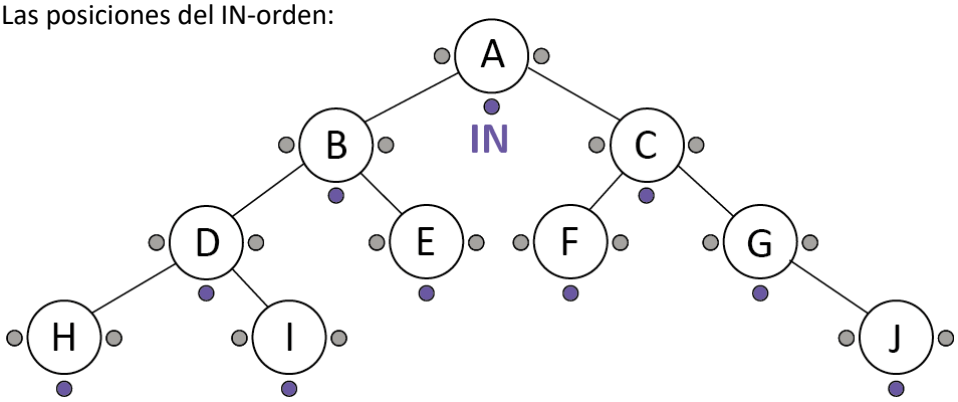
- Primero, en torno a cada nodo, se identifica a los tres recorridos como posiciones con colores diferentes.



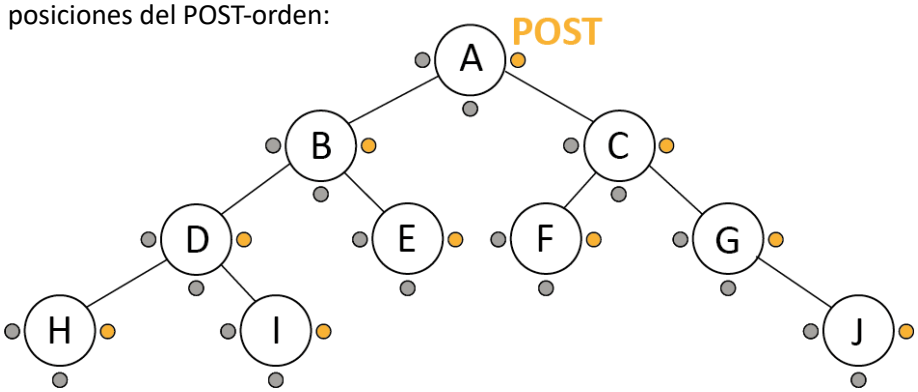
- Las posiciones del PRE-orden:



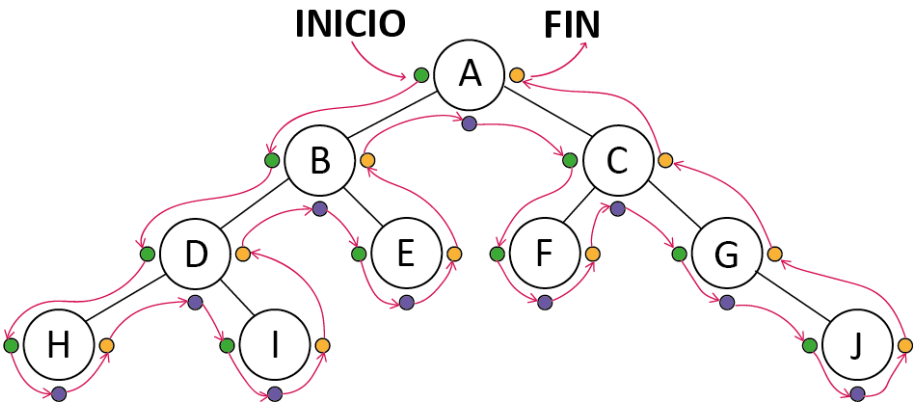
- Las posiciones del IN-orden:



- Las posiciones del POST-orden:



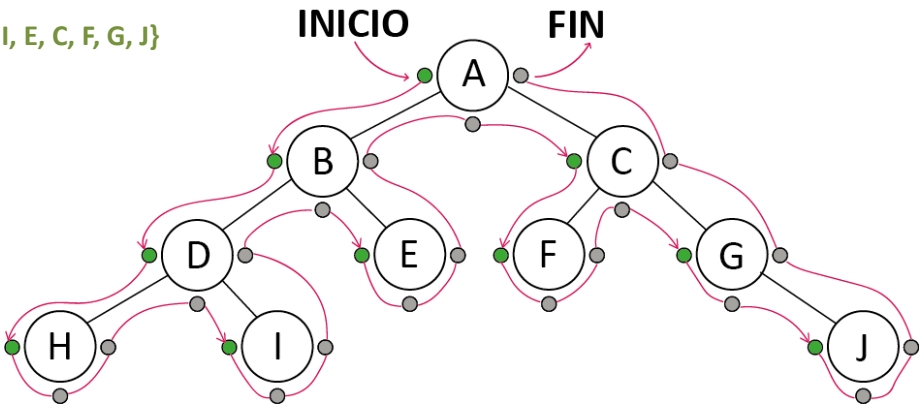
Independientemente de cuál de los tres algoritmos se ejecute, el recorrido del árbol siempre sigue el mismo itinerario. En realidad, el recorrido es el mismo, lo que varía es el orden en el que ingresan los nodos/vértices en la secuencia elegida (pre, in o post), ya que dependiendo de la elegida, se activan o no ciertos vértices.



Para entenderlo solo en necesario seguir a donde se dirigen las puntas de la flecha:

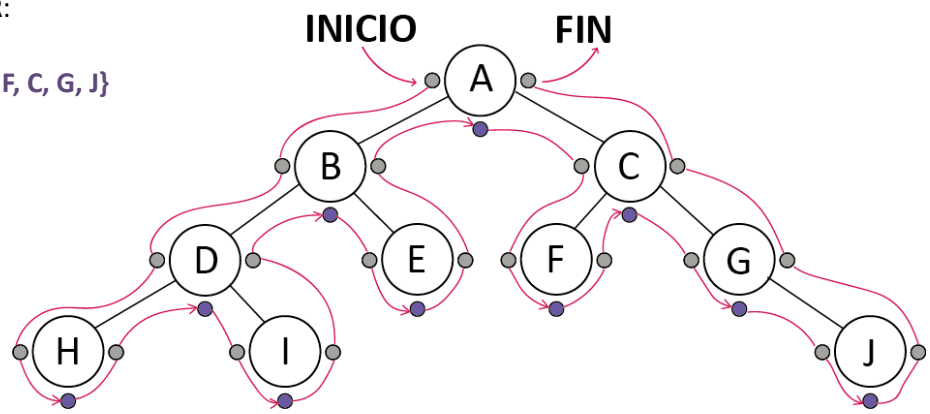
Ejemplo PRE-ORDER:

PRE {A, B, D, H, I, E, C, F, G, J}



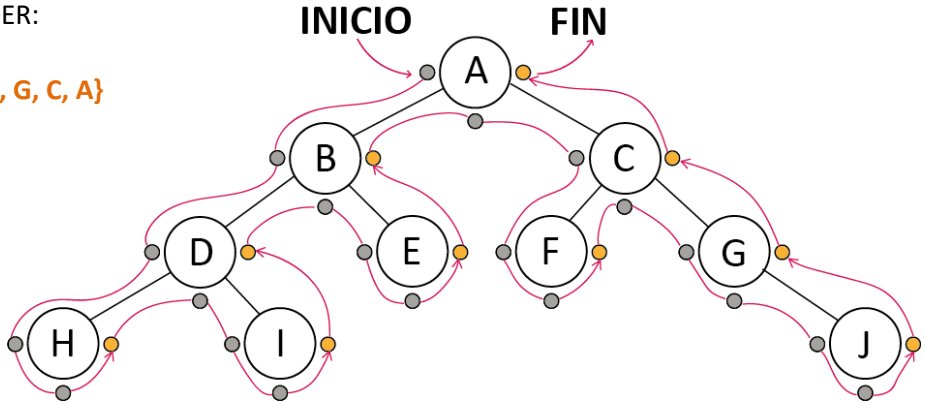
Ejemplo IN-ORDER:

IN {H, D, I, B, E, A, F, C, G, J}



Ejemplo POST-ORDER:

IN {H, I, D, E, B, F, J, G, C, A}



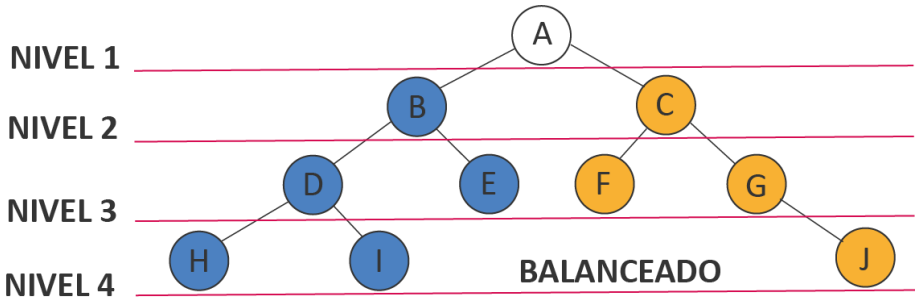
CLASIFICACIÓN SEGÚN SU ESTRUCTURA

De acuerdo con su estructura, los árboles binarios se pueden clasificar como:

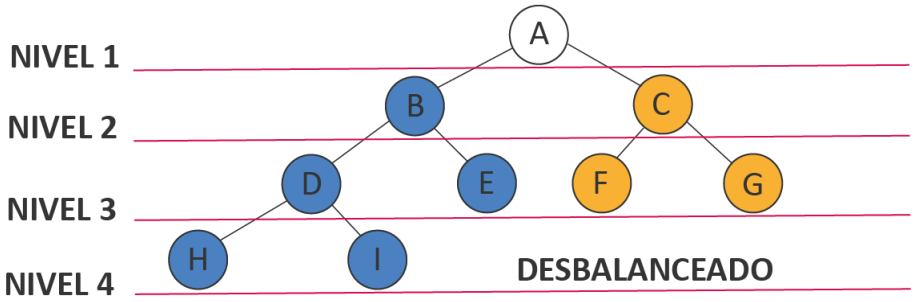
- **Balanceados o desbalanceados**: en este caso se debe tener en cuenta el factor de equilibro. Este factor resulta de la diferencia entre la altura del subárbol derecho y el izquierdo.

  Entonces: EQUILIBRIO = HD-HI

  De esta manera, un árbol balanceado es un árbol en el que los subárboles izquierdo y derecho de cada nodo tienen alturas lo más parecidas posible. Es decir, la diferencia de altura no supera cierto límite (generalmente 1).

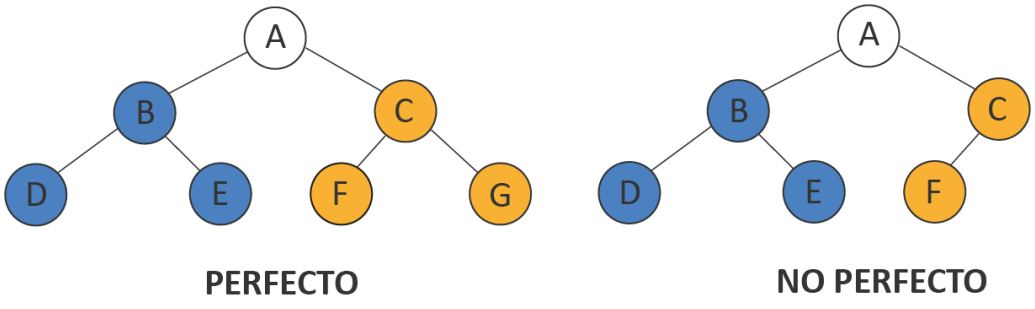


En el caso de los árboles desbalanceados, la cantidad de niveles hacia la izquierda no es la misma que hacia la derecha. Esto se puede observar cuando de uno de los subárboles crece mucho más que el otro. Respecto del factor de equilibro, si este es menor a -1 significa que el árbol se encuentra desbalanceado en el subárbol izquierdo, si el factor es mayor a 1, quiere decir que el árbol se encuentra desbalanceado en el subárbol derecho.



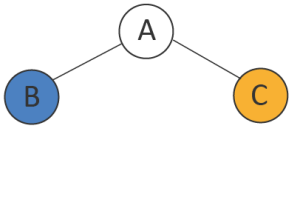
- **Perfecto o Imperfecto:**

  Se lo considera un árbol perfecto cuando todos los nodos tienen exactamente dos hijos y todos los niveles están completamente llenos.

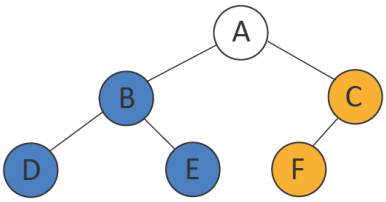


- **Completos o No completos:**

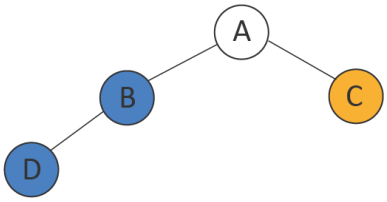
  Un árbol binario completo es un árbol cuyos nodos padre están llenos y los nodos hoja en el último nivel ocupan las posiciones más a la izquierda.



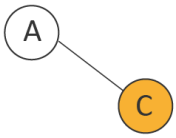
COMPLETO



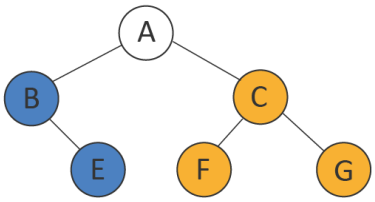
COMPLETO



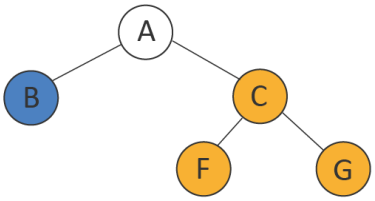
COMPLETO



NO COMPLETO



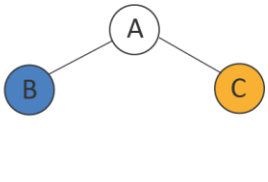
NO COMPLETO



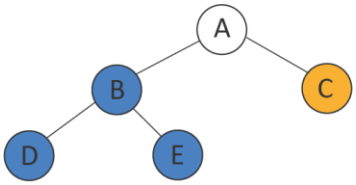
NO COMPLETO

- **Llenos o no llenos:**

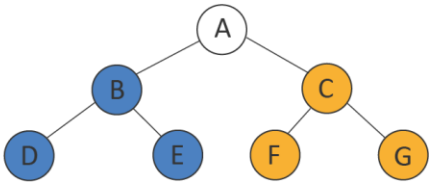
  Se considera que un árbol es lleno cuando los nodos tienen cero o dos hijos. No existen nodos que tengan un solo hijo.



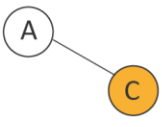
LLENO



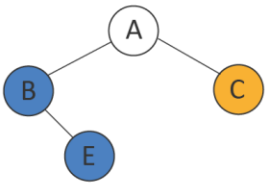
LLENO



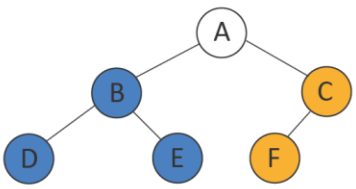
LLENO



NO LLENO



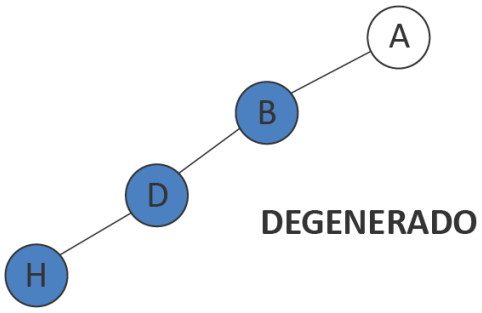
NO LLENO



NO LLENO

- **Degenerado**

Un árbol binario es degenerado cuando hay un solo nodo hoja y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado tiene una forma lineal y su equivalente es una lista enlazada. Todo árbol degenerado es desbalanceado, pero no al revés, ya que, en el caso de este último, los nodos pueden tener más dos hijos.



TIPOS DE ÁRBOLES BINARIOS

Anteriormente se clasificó a los árboles binarios por su forma o estructura, a continuación se los clasifica según las reglas que siguen para funcionar correctamente:

- **ÁRBOL DE BÚSQUEDA BINARIA (BST)**

Su principal función es facilitar el ordenamiento y la búsqueda rápida de elementos, como por ejemplo encontrar una palabra en un diccionario digital.

Una característica clave de este tipo de árbol binario es que, en cada nodo, los valores del subárbol izquierdo son menores que el valor del nodo padre, mientras que los valores del subárbol derecho son mayores.

Por otro lado, la búsqueda binaria es un algoritmo que trabaja dividiendo una lista ordenada por la mitad, comparando el valor que buscamos con el elemento del medio para saber en qué mitad continuar la búsqueda. Así se descarta la mitad donde el elemento no puede estar, hasta encontrarlo o confirmar que no está. Este método es mucho más rápido que recorrer la lista completa, porque su complejidad es O(log n), mientras que la búsqueda secuencial tiene O(n).

Los BST se usan mucho en diccionarios digitales y en otros programas donde se necesita buscar y ordenar información rápido y eficiente.

- **ÁRBOL AVL**

En este caso, todo árbol es de tipo AVL si la diferencia de altura entre subárboles izquierdo y derecho no puede ser mayor a 1. Si se rompe esa regla al insertar o eliminar, se rebalancea automáticamente con rotaciones. De esta manera, se puede decir que est tipo de árbol busca mantener el árbol equilibrado para que las operaciones sean rápidas.

Este equilibrio evita que el árbol se vuelva muy alto y delgado, lo que haría que buscar o insertar elementos sea lento. Gracias a esto, los árboles AVL mantienen una velocidad de búsqueda, inserción y eliminación de datos muy eficiente, con una complejidad en O(log n).

- **ÁRBOLES ROJO-NEGRO**

Los árboles rojo-negro también son una variante de árbol binario de búsqueda balanceado, pero con reglas diferentes para mantener el equilibrio. Cada nodo se pinta de rojo o negro, y el árbol sigue ciertas reglas, por ejemplo: un nodo rojo no puede tener un hijo rojo y todos los caminos desde la raíz hasta las hojas deben tener la misma cantidad de nodos negros. Así, el árbol intenta no desbalancearse demasiado. Estas reglas hacen que el árbol no esté perfectamente balanceado como un AVL, pero son suficientes para que las operaciones sigan siendo rápidas. Además, la estructura es un poco más flexible que la de un AVL, por lo que es muy usada en sistemas donde hay muchas inserciones y eliminaciones. La complejidad también es O(log n) para buscar, insertar y borrar.

LA IMPORTANCIA DE LA ESTRUCTURA DE ÁRBOL

Luego de conocer las definiciones más importantes, sus propiedades y características y también los tipos de árboles, es posible profundizar en las ventajas de este tipo de estructura. Como se mencionó antes, los elementos se organizan de manera jerárquica, lo que facilita que operaciones como agregar, eliminar y buscar se realicen más rápido, ya que se puede localizar un elemento sin tener que recorrer todos los datos almacenados. Por eso, se pueden destacar tres ventajas principales: orden y organización, acceso ágil a la información y eficiencia.

Es posible encontrar múltiples ejemplos de esta estructura en programación. Un ejemplo que se puede ver a diario es el Explorador de Windows, este representa visualmente un árbol de directorios. Las carpetas (directorios) están organizadas de forma jerárquica. Por un lado, la

carpeta principal (Disco C) sería la raíz del árbol, la cual contiene al resto de las carpetas que, a su vez, pueden contener subcarpetas y así sucesivamente.

Esta estructura hace que sea más fácil navegar para los usuarios, ya que pueden expandir o contraer ramas del árbol para ver solo lo que necesitan.

Respecto de formas más específicas de árbol, La eficacia, flexibilidad y facilidad de implementación que proporcionan los Árboles Binarios los convierten en un recurso importante no sólo en Informática, sino también en muchos otros campos técnicos. El árbol binario se utiliza en muchos programas y algoritmos para optimizar tareas y mejorar la eficiencia. Algunos de sus usos más comunes son:

- **Sistemas de archivos:** Como en el ejemplo del Explorador de Windows, la organización jerárquica de carpetas y archivos se basa en una estructura de árbol.
- **Representación de expresiones:** Los árboles binarios se usan para representar expresiones matemáticas o lógicas, donde los nodos son operadores y las hojas operandos. Esto es clave para compiladores y calculadoras.
- **Inteligencia artificial en videojuegos:** Los árboles binarios ayudan a gestionar la lógica de decisiones en personajes y enemigos, facilitando comportamientos complejos.
- **Compresión de datos:** Las tablas de Huffman, un tipo de árbol binario, se usan para comprimir información de manera eficiente, reduciendo el espacio necesario para almacenar datos.

CASO PRÁCTICO

Descripción del problema

En el contexto de una empresa, se necesita desarrollar un programa que permita gestionar una agenda jerárquica para el envío de correos electrónicos. La información debe ser transmitida primero a los jefes de área y, luego, a sus respectivos empleados. Para ello, se requiere modelar la estructura organizativa mediante un árbol donde cada nodo representa a un miembro del equipo. Así, se puede recorrer la estructura y enviar correos en el orden jerárquico adecuado.

Decisiones de diseño

Para organizar la información jerárquica de la agenda y facilitar la rápida búsqueda y manipulación de datos, se optó por implementar un árbol binario de búsqueda (BST), debido a que ofrece:

- Jerarquía y orden: Aunque se podría modelar la estructura de la agenda con otros tipos de árboles, un BST mantiene los datos ordenados (por ejemplo, por nombre), lo que facilita encontrar rápidamente un contacto específico.
- Eficiencia en operaciones: Los BST permiten búsquedas, inserciones y eliminaciones eficientes (mediante el predecerdor inorden), algo fundamental cuando la información se actualiza o consulta con frecuencia.

Además, para resolver el problema planteado, no solo fue clave la estructura de los datos, sino también la elección del recorrido. Se seleccionó el recorrido en preorden (primero el nodo actual y luego sus hijos) para garantizar que el jefe reciba el mensaje antes que sus empleados.

Codigo fuente

El código implementa una agenda utilizando una estructura de árbol binario para almacenar nombres en orden alfabético, y permite agregar, buscar y eliminar contactos. Al comenzar, se inicializa la agenda como “None”, es decir, vacía. Luego se define la función agregar, que permite insertar nombres en la estructura del árbol de forma ordenada: si la agenda está vacía, crea un nuevo nodo con el nombre; si el nombre es menor al nodo actual, lo inserta recursivamente a la izquierda; si es mayor, a la derecha; si ya existe, no se agrega. A continuación, se inicia un bucle que solicita al usuario nombres para cargar: si se ingresa "0", el ciclo se detiene; si se deja en

blanco o se ingresa un número, muestra un mensaje de error y vuelve a pedir el nombre. Los nombres válidos se insertan en la agenda mediante la función agregar. Finalizada esta carga, se muestra la estructura resultante del árbol.

Para facilitar la visualización de la agenda en forma de árbol, se incorpora la función mostrar_agenda, que recorre el árbol y lo imprime con una representación jerárquica estructurada. Esta función recibe como parámetros la agenda, un prefijo que se va concatenando para reflejar la profundidad del nodo, y una bandera booleana que indica si el nodo actual es izquierdo. Si el nodo no es None, la función imprime el contenido del nodo precedido por símbolos gráficos (├── o └──) que muestran la orientación en el árbol.

AGREGAR NOMBRES

```python
agenda = None                                   #Lista vacia de la agenda.


def agregar(agenda, nombre):                    #Función que agrega nombres a la agenda.
    nombre=nombre.capitalize()                  #Variable donde se guarda el nombre ingresado, acompañado del metodo .capitalice() para almacenar el nombre con su primer letra mayuscula
    if agenda is None:                          #Condicional if que verifica si la lista esta vacia, si es True retorna la variable nombre.
        return [nombre, None, None]             #[dato, dato izq, dato der].
    if nombre < agenda[0]:                      #Condicional que verifica si el siguiente nombre ingresado es menor al primer indice de la agenda[0].
        agenda[1] = agregar(agenda[1], nombre)  #Llama a la funcion recursiva, desde la variable del indice 1 de la agenda[1] (Agrega los nombres a su lado izquierdo).
    elif nombre > agenda [0]:                   #Condicional que verifica si el siguiente nombre ingresado es mayor que el primer indice de la agenda[0].
        agenda[2] = agregar(agenda[2], nombre)  #Llama a la funcion recursiva, desde la variable del indice 1 de la agenda[2] (Agrega los nombres a su lado derecho).
    return agenda                               #No se inserta si es duplicado.


while True:                                     #Bucle While para verificar el nombre ingresado.
    nombre=(input("Ingrese un nombre o cero para finalizar"))   #Solicitda un nombre al usuario por pantalla a traves de un input.
    nombre=nombre.strip()                       #Almacena el nombre en una variable y mediante el metodo .strip() elimina cualquier espacio en blanco en el str del dato ingresado.
    if nombre == "0":                           #Se evalua la informacion del usuario si esta es "0",sale del programa.
        break
    elif nombre == "":                          #Si el usuario no ingresa nada, se le muestra un mensaje de error y se solicita nuevamente la información.
        print("Error: No ha ingresado ningún nombre. Intente nuevamente.")
        continue
    elif nombre.isdigit():                      #De la misma manera si el usuario ingresa un digito, se el mostrara un mensaje de error.
        print("Error: Ha ingresado un número. Intente nuevamente.")
        continue
    else:
        agenda = agregar(agenda, nombre)        #Si es correcto se llama a la funcion agregar nombre, para almacenar la información.

print(agenda)
```

Python

MOSTRAR AGENDA (ESTRUCTURA DE ARBOL)

```python
def mostrar_agenda(agenda, prefijo="", es_izquierda=True):      #Función para mostrar la estructura del arbol de forma grafica.
    if agenda is not None:                                      #Mostrar por pantalla un prefijo que va acumulando espacios y lineas para crear sangrias y conectar elementos.
        print(prefijo + ("├── " if es_izquierda else "└── ") + agenda[0])   #y un parametro "es_izquierda" que indica si debe utilizar un conector u otro.
        if agenda[1] is not None or agenda[2] is not None:
            mostrar_agenda(agenda[1], prefijo + ("│   " if es_izquierda else "    "), True)     #Llamado a la recursividad para subarbol izquierdo.
            mostrar_agenda(agenda[2], prefijo + ("│   " if es_izquierda else "    "), False)    #Llamado a la recursividad para subarbol derecho.

mostrar_agenda(agenda)
```

Python

Luego, se define la función buscar_nombre, que busca un nombre recorriendo el árbol: si el nodo es None, retorna False; si coincide con el nombre buscado, retorna True; si el nombre es menor, busca a la izquierda; y si es mayor, a la derecha. Posteriormente se definen dos funciones más: extraer_sucesor (que se utiliza en el caso especial que el nodo a eliminar tenga dos hijos) y borrar_nombre, que elimina un nombre de la agenda respetando las reglas del árbol binario: si el nodo no tiene hijos, se elimina directamente; si tiene un solo hijo, se reemplaza por ese hijo; y si tiene dos hijos, se combina la función borrar_nombre y extraer_sucesor para encontrar al sucesor. Se utiliza la regla del predecesor inorden de los árboles de búsqueda binaria, la cual

determina que se debe buscar el nodo con valor máximo del subárbol izquierdo. Este será el que reemplazará al contacto/nombre eliminado.

FUNCIÓN BUSCAR NOMBRE

```python
def buscar_nombre(agenda, nombre):                  #Función que recorre la agenda para buscar un nombre determinado.
    if agenda is None:                              #Si la agenda esta vacia, retorna False.
        return False
    elif nombre == agenda[0]:                       #Si el nombre es igual al primer indice de la agenda, se retorna True.
        return True
    elif nombre < agenda[0]:                        #Sino se busca el nombre hacia la izquierda de la estructura.
        return buscar_nombre(agenda[1], nombre)
    else:                                           #Y si es mayor se busca el nombre hacia la derecha.
        return buscar_nombre(agenda[2], nombre)
```

[52] ✓ 0.0s Python

FUNCION BORRAR NOMBRE

```python
def extraer_predecesor(agenda):                     #Función para buscar al nodo predecesor, cuando borremos algun nombre de la agenda.
    lista_predecesor = agenda                       #Se asigna nueva variable a la lista actual.
    if lista_predecesor[2] is not None:             #Se verifica si tiene un hijo derecho.
        lista_predecesor = lista_predecesor[2]      #Si es True, se le asigna ese nodo a lista_predecesor.
    return lista_predecesor[0]                      #Sino se retorna el nodo almacenado en lista_predecesor[0], que seria el nombre del supuesto predecesor.

def borrar_nombre(agenda, nombre):                  #Función para borrar nombres de la agenda.
    if agenda is None:                              #Verifica si la agenda esta vacia, de ser asi devuelve "None".
        return None

    if nombre == agenda[0]:                         #Se encuentra el nodo con el nombre que queremos borrar.
    # Si el nodo no tiene hijos
        if agenda[1] is None and agenda[2] is None: #Si no tiene hijos elimina y retorna "None" al padre.
            return None
    # Si tiene hijo a la derecha
        elif agenda[1] is None:                     #Remplaza el nodo con su hijo derecho.
            return agenda[2]
    # si tiene hijo a la izquierda
        elif agenda[2] is None:                     #Remplaza el nodo con su hijo izquierdo.
            return agenda[1]
    # Si tene hijos en ambos lados
        else:
            nodo_predecesor = extraer_predecesor(agenda[1])     #Manda a llamar a la funcion etxraer_predecesor.
            agenda[0] = nodo_predecesor                         #Remplaza el dato actual por el predecesor.
            agenda[1] = borrar_nombre(agenda[1], nodo_predecesor)   #Se elimina el nodo que contenia originalmente el valor nodo_predecesor.
            return agenda                                       #Se retorna la agenda actualizada.

    elif nombre < agenda[0]:                        #Si el nombre es menor al altual busca y elimina recursivamente hacia la izquierda.
        agenda[1] = borrar_nombre(agenda[1], nombre)
    else:                                           #Si es mayor busca y elimina recursivamente hacia la derecha.
        agenda[2] = borrar_nombre(agenda[2], nombre)

    return agenda                                   #Retorna la agenda modificada.
```

[53] ✓ 0.0s Python

Finalmente, se permite al usuario ingresar un nombre para consultar en la agenda. Si el nombre existe, se pregunta si desea eliminarlo; si el usuario responde que sí, se lo elimina y se informa la acción. Si el nombre no existe, se consulta si desea agregarlo; si se acepta, se inserta utilizando la función agregar y se muestra un mensaje confirmando la operación. Al concluir, se imprime nuevamente la agenda, mostrando su estructura actualizada.

```python
nombre_ingresado = input("Ingrese el nombre que desea buscar: ").capitalize()

# Se evalua si esta en la lista o no
if buscar_nombre(agenda, nombre_ingresado):                 #Se busca el contacto llamando a la funcion buscar nombre
    consulta_borrado=input("¿Desea borrar el nombre? Ingrese si o no: ")    #Si el contacto está entonces se pregunta si el usuario desea eliminar el contacto
    consulta_borrado=consulta_borrado.lower()
    if consulta_borrado=="si":                              #Se evalúa si la respuesta es si
        agenda=borrar_nombre(agenda,nombre_ingresado)
        print(f"{nombre_ingresado} ha sido eliminado de la agenda.")    #Se notifica al usuario que el contacto fue eliminado
else:
    consulta_agregado=input("¿Desea agregar el contacto? Ingrese si o no: ")    #Si el contacto no está en la agenda entonces se pregunta si el usuario desea eliminar el contacto
    consulta_agregado=consulta_agregado.lower()
    if consulta_agregado=="si":                             #Se evalúa si la respuesta es si
        agenda=agregar(agenda,nombre_ingresado)
        print(f"{nombre_ingresado} ha sido agregado a la agenda.")      #Se notifica al usuario que el contacto fue agregado


print(agenda)
```

[ ] Python

Validación del funcionamiento

Para garantizar que la solución implementada cumple con los objetivos planteados y funciona correctamente, se realizaron varias pruebas que validan las operaciones principales sobre la agenda representada como un árbol de búsqueda binaria (BST):

- Inserción de nombres: Se comprobó que al ingresar distintos nombres, estos se almacenan respetando el orden del BST, ubicándose a la izquierda si son menores y a la derecha si son mayores al nodo actual. Además, se verificó que no se permiten duplicados, asegurando la integridad de los datos.
- Búsqueda de nombres: Se validó la correcta localización de nombres dentro de la agenda, confirmando que la función retorna True cuando el contacto existe y False en caso contrario. La búsqueda se realiza de forma eficiente, recorriendo únicamente los subárboles necesarios, lo que demuestra la correcta implementación del BST.
- Eliminación de nombres: Se probaron distintos escenarios para la eliminación: nodos hoja, nodos con un hijo y nodos con dos hijos. En cada caso, la estructura se actualizó manteniendo las propiedades del árbol, y el nodo eliminado desapareció correctamente de la agenda. También se validó que intentar eliminar un contacto inexistente no genera errores ni altera la estructura.
- Interacción con el usuario: El programa responde adecuadamente a las decisiones del usuario respecto a agregar o eliminar contactos. Se confirmó que la capitalización automática de nombres y la sensibilidad a mayúsculas/minúsculas funcionan correctamente, mejorando la experiencia y precisión en la búsqueda.

Se realizaron pruebas con múltiples operaciones consecutivas de inserción, búsqueda y eliminación para asegurar que la estructura mantiene su orden y funcionalidad sin fallas. Además, se manejaron entradas inválidas o vacías o números (diferentes a cero) sin que el programa se interrumpiera inesperadamente.

En conjunto, estas validaciones demuestran que la agenda jerárquica implementada cumple con el propósito de organizar y gestionar contactos eficientemente, facilitando la búsqueda y manipulación de datos según lo requerido en el problema planteado.

METODOLOGÍA UTILIZADA

Para el desarrollo del presente trabajo se siguieron los siguientes pasos:

1. **Estructuración inicial de la investigación:**

   Como primer acercamiento al tema, se utilizó una herramienta de inteligencia artificial (ChatGPT) para obtener una visión general y definir posibles títulos o ejes temáticos. Esta instancia permitió estructurar tanto la investigación como el desarrollo posterior del trabajo práctico.

2. **División de tareas:**

   El equipo distribuyó las distintas secciones del trabajo entre sus dos integrantes. Cada persona investigó su parte de manera autónoma, aunque manteniendo una visión integral del tema. Este enfoque favoreció la retroalimentación y coherencia entre las distintas partes.

3. **Proceso de investigación:**

   Cada integrante consultó diversas fuentes, incluyendo videos en YouTube, sitios web especializados, material de la cátedra y herramientas de IA como ChatGPT. Esta última fue utilizada principalmente para reforzar conceptos, aclarar dudas técnicas y enriquecer la redacción con vocabulario más preciso.

4. **Diseño, pruebas y revisión:**

   Una vez desarrolladas las partes, se unificó el contenido y se procedió a una revisión general. Se corrigieron errores, se reorganizó el texto para evitar redundancias y se buscó mantener una estructura clara y coherente.

5. **Aplicación de formato y recursos visuales:**

   Finalmente, se incorporaron elementos visuales (como cuadros comparativos e ilustraciones) para mejorar la presentación del trabajo y facilitar la comprensión de los conceptos desarrollados.

CONCLUSIÓN

A lo largo del desarrollo de este trabajo práctico se logró combinar de manera clara tanto los aspectos teóricos como los prácticos del tema sobre Arboles en Python. En la primera parte del trabajo se desarrollaron contenidos fundamentales para entender cómo funcionan estas estructuras como por ejemplo qué es un nodo, cómo se organizan los datos jerárquicamente, y qué significan conceptos como niveles, profundidad o recorrido, etc. Esta base teórica fue clave para poder comprender por qué se usan los árboles, cuáles son sus ventajas frente a otras formas de organizar datos, y cómo pueden adaptarse a distintos problemas. Estos conceptos no solo permitieron entender el porqué de cada decisión técnica posterior, sino que además consolidó el conocimiento previo adquirido a lo largo de la materia.

Desde el punto de vista práctico, la implementación de una agenda en Python utilizando un árbol binario de búsqueda (ABB) permitió aplicar los contenidos teóricos en un entorno funcional. A través de la manipulación de listas anidadas y el uso de funciones recursivas, se resolvieron operaciones clave como la inserción, búsqueda, eliminación y visualización estructurada de datos.

Desde un aspecto formativo, este trabajo representa una instancia valiosa destacando la importancia de seleccionar estructuras de datos adecuadas para resolver problemas específicos con eficiencia y claridad.

BIBLIOGRAFÍA

Páginas web:

Videos:

- Fazt. (2021, 26 de marzo). *Estructuras de Datos en Python | Curso Práctico* [Video]. YouTube. https://www.youtube.com/watch?v=G1VS5FbtMS4
- Dalto, C. (2022, 17 de mayo). *Estructuras de datos en Python - Parte 1 | Curso completo* [Video]. YouTube. https://www.youtube.com/watch?v=Jo2euX89Oz8
- Develoteca. (2018, 17 de octubre). *Curso de estructuras de datos | Árboles binarios* [Video]. YouTube. https://www.youtube.com/watch?v=r-rJePaU9wI&t=85s

Webs y documentos

- DataCamp. (s.f.). *Guía de estructuras de datos en Python*. DataCamp. https://www.datacamp.com/es/tutorial/data-structures-guide-python
- Cairo, A., & Guardati, M. (s.f.). *Estructuras de datos*. [Archivo PDF]. *(Agrega editorial o institución si aplica)*.
- KeepCoding. (2023). *Árboles binarios en programación: qué son y cómo funcionan*. KeepCoding Blog. https://keepcoding.io/blog/arboles-binarios-en-programacion/
- Diego Coder. (2022, 19 de abril). *Grafos y árboles en Python*. Medium. https://medium.com/@diego.coder/grafos-y-%C3%A1rboles-en-python-2d165dfc8bbd