Trabajo Integrador

- **Título del trabajo:** Estructuras de Datos Avanzadas: Implementación de Árboles Binarios con Listas

- **Alumnos:**
  Cintia Garcia - garcia.cintia1307mail.com
  Pablo de la Puente - pablo.delapuente@gmail.com

- **Materia:** Programación I
- **Profesor/a**: Julieta Trapé
- **Fecha de Entrega**: 09/06/2025

Índice

---

1. Introducción

El presente trabajo aborda el estudio y desarrollo de estructuras de datos avanzadas, focalizándose en los árboles binarios. Este tema fue seleccionado debido a la relevancia que tienen los árboles en la organización eficiente de grandes volúmenes de datos, así como en la optimización de procesos de búsqueda y ordenamiento, fundamentales en el área de la programación y ciencias de la computación.

Si bien existen implementaciones más complejas de árboles mediante el uso de clases y nodos enlazados, se optó por desarrollar el proyecto utilizando listas en Python para representar la estructura del árbol. Esta decisión facilitó la comprensión del funcionamiento básico de los árboles y permitió enfocarse en los conceptos esenciales

de inserción, recorrido y búsqueda, a la vez que se mantenía una implementación práctica y funcional.

El objetivo principal del trabajo es diseñar y desarrollar un programa que gestione patentes de autos mediante un árbol binario de búsqueda, demostrando cómo esta estructura permite realizar búsquedas rápidas y eficientes. Asimismo, se busca que el programa sea dinámico, sencillo y escalable, brindando una herramienta práctica que facilite la gestión de información ordenada y evite duplicaciones. De esta forma, se contribuye al aprendizaje y aplicación de estructuras de datos avanzadas en un contexto real y significativo.

---

2. Marco Teórico

Cuando pensamos en estructuras imaginamos algo que tiene una base, un cuerpo y tiende a expandirse. Esta idea se refleja en las estructuras de datos jerarquicas conocidas

como árboles binarios (Wikipedia, Árbol binario.). pensamos en un árbol, visualizamos su raíz, su tronco y su copa. La copa posee hojas y ramas, podemos distinguir distintas

partes, pero al detenernos vemos que todo se encuentra conectado. Para que se desarrolle la copa primero debe tener ramas, y esas ramas están unidas a un tronco. El tronco no existe por sí solo, sino que creció gracias a la raíz, y esa raíz para ser fuerte y darle crecimiento al árbol debe estar sólida, nutrida y haber sido cuidada con agua y nutrientes.

Explicando esto de forma sencilla, podemos imaginar un poco lo que sucede en programación cuando hablamos de estructuras: árboles binarios. De la misma forma que crece un árbol en la tierra, se desarrolla un programa.

Para construir un programa con estructuras de árboles, así como un árbol crece de manera estructurada, un árbol binario sigue un orden jerárquico y organizado: comienza con una raíz, que es el punto de partida, y cada nodo puede tener dos hijos como máximo, uno a la izquierda y otro a la derecha. El tronco representa la lógica del código, mientras que las ramas representan los distintos caminos que el programa puede recorrer. Siguiendo esta estructura, el programa avanza hasta alcanzar su resultado.

Como programadores, nuestra función es impulsar su crecimiento y mantenimiento, tal como cuidamos un árbol para que crezca correctamente.

¿Qué es un Árbol Binario?

Según la Wikipedia (https://es.wikipedia.org/wiki/%C3%81rbol_binario):

Un árbol binario es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo no apunta a ningún nodo (es decir, tiene una referencia null), se lo llama nodo externo. Si tiene al menos un hijo, se lo llama nodo interno.

Los árboles binarios son esenciales para representar decisiones, expresiones matemáticas, algoritmos de búsqueda, estructuras organizativas y más.

Recorridos en un Árbol Binario

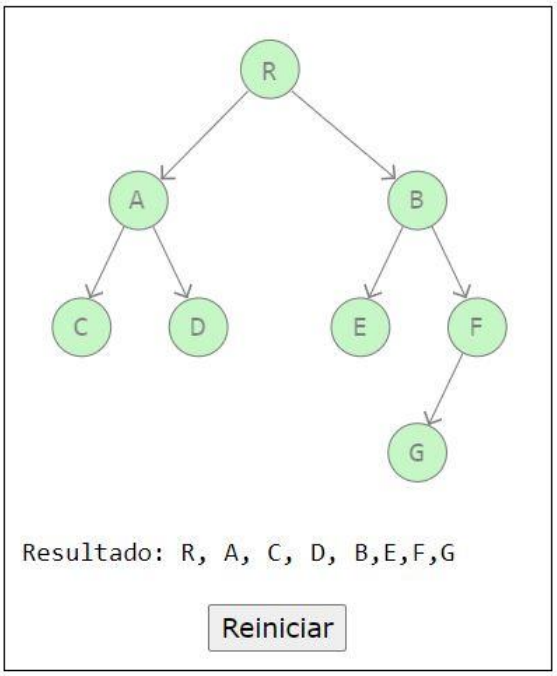
Los recorridos en profundidad son formas específicas de visitar los nodos de un árbol binario. Hay tres formas principales:

Preorden (Pre-order):

Se visita primero el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.

Orden de visita: Raíz → Izquierda → Derecha

Uso: copiado de árboles, expresión en notación prefija.

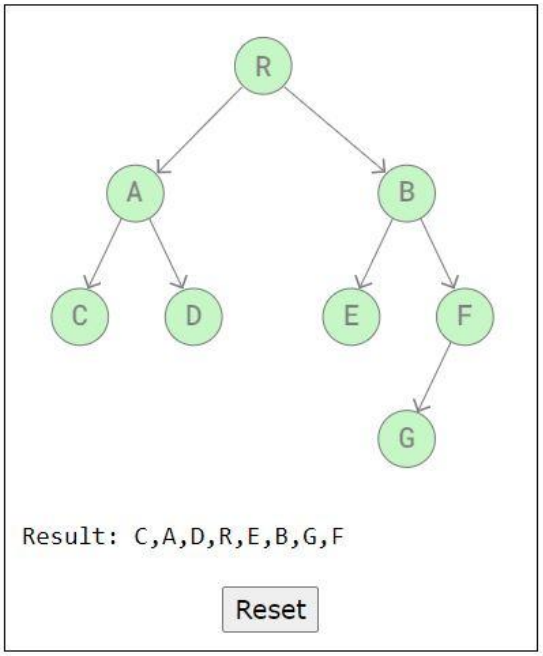


Inorden (In-order):

Se recorre primero el subárbol izquierdo, luego se visita la raíz, y finalmente el subárbol derecho.

Orden de visita: Izquierda → Raíz → Derecha

Uso: Ideal para árboles binarios de búsqueda (BST), ya que devuelve los elementos en orden ascendente.

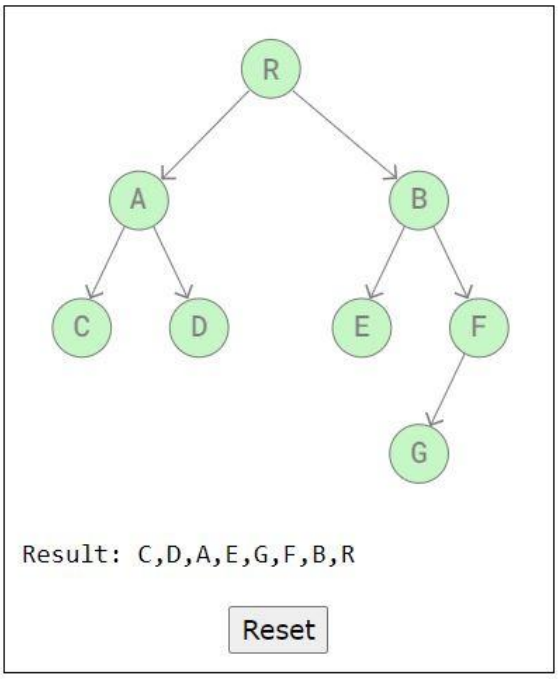


Postorden (Post-order):

Se recorre primero el subárbol izquierdo, luego el derecho, y finalmente se visita la raíz.

Orden de visita: Izquierda → Derecha → Raíz

Uso: eliminación de árboles, evaluación postfija de expresiones.



Implementación con Arrays

Una forma alternativa de representar árboles binarios es mediante arrays (vectores), lo cual puede ser útil cuando el árbol no se modifica frecuentemente.
¿Por qué?

- Ventajas: mayor eficiencia en lectura, mejor uso de la memoria gracias a la localidad de caché (cache locality).

- Desventajas: insertar o eliminar nodos puede ser más costoso, porque requiere reorganizar posiciones en el array.

Localidad de caché: como los arrays están almacenados de forma contigua en memoria, los procesadores pueden acceder más rápidamente al siguiente elemento sin necesidad

de buscarlo, porque ya está cargado en caché.
En árboles que se modifican con frecuencia (inserciones/eliminaciones), conviene usar la implementación clásica con punteros a los nodos izquierdo y derecho.

Árboles Binarios de Búsqueda (BST) – Parte del enfoque DSA

Los Binary Search Trees (BST) son árboles binarios donde los nodos están organizados según una relación de orden:

- El hijo izquierdo contiene valores menores que la raíz.
- El hijo derecho contiene valores mayores que la raíz.

Este orden permite realizar búsquedas, inserciones y eliminaciones de forma eficiente, con una complejidad promedio de O(log n) (en árboles balanceados). Son una pieza clave en estructuras de datos y algoritmos (DSA) porque permiten almacenar y buscar información de forma muy optimizada.
Existen variantes más avanzadas como:

- Árboles AVL: se mantienen balanceados automáticamente.
- Árboles Rojo-Negro: garantizan equilibrio a través de reglas de color.
- Splay Trees, Treaps, y más.

Tipos de Árboles Binarios:

- Árbol Binario de Búsqueda (BST): mantiene los nodos ordenados para búsquedas eficientes.
- Árbol de Fibonacci: usado en programación avanzada.
- Árbol Balanceado: diferencia de altura mínima entre subárboles.
- Árbol Completo: todos los niveles están llenos, excepto tal vez el último.
- Árbol Lleno: cada nodo tiene 0 o 2 hijos.
- Árbol Perfecto: todos los nodos internos tienen dos hijos y todas las hojas están al mismo nivel.

W3Schools. *Binary Tree Traversal*.
https://www.w3schools.com/dsa/dsa_data_binarytrees.php

---

3. Caso Práctico

Al desarrollar nuestro trabajo, buscamos una solución que sea dinámica, sencilla y práctica, evitando duplicaciones y optimizando la gestión de información. En Python desarrollamos un programa de PATENTES DE AUTOS (cada patente es única).
El diseño y desarrollo del mismo se enfocó en lograr una búsqueda rápida y eficiente, manteniendo un orden estructurado y permitiendo expandirse en caso de ser necesario.
Pensemos que tenemos muchas patentes de autos mezcladas, todas diferentes. En lugar de recorrer una lista completa para encontrar una específica, nuestro trabajo permite ir directamente al lugar indicado.
Las patentes de autos en nuestro sistema poseen números y letras, las cuales se acomodan alfabéticamente. La raíz de nuestro árbol es la primera patente ingresada. Luego se bifurca en dos nodos: las patentes menores (alfabéticamente) se colocan a la izquierda y las patentes mayores se colocan a la derecha.
El programa permite visualizar el árbol en consola, con una representación jerárquica que indica claramente la relación entre los nodos.

```
==============================
    D--> KAS568
D--> HD786NB
FGF456
    D--> CWD834
I--> AH456SA
    I--> AB342BH
==============================
Patente:
```

Además, se puede realizar el recorrido clásico del árbol: inorden, preorden y postorden, mostrando las patentes en el orden correspondiente.

Toda la lógica se apoya en funciones recursivas, tanto para la inserción de nodos como para los recorridos y el cálculo de profundidad del árbol.

```
def insertar(raiz, patente):
    if raiz is None:      # Si el árbol está vacío, se crea un nodo nuevo
        return crear_nodo(patente)
    if patente == raiz[0]:  # Si la patente ya existe, no se agrega
        print(f"La patente {patente} ya existe!")
        return raiz
    elif patente < raiz[0]:  # Si la patente es menor, va a la izquierda
        raiz[1] = insertar(raiz[1], patente)  # Llamada recursiva a la izquierda
    else:  # Si la patente es mayor, va a la derecha
        raiz[2] = insertar(raiz[2], patente)  # Llamada recursiva a la derecha
```

También incluye un menú interactivo, que permite al usuario ejecutar estas funciones de manera sencilla y amigable.

```
def main():
    raiz = None  # El árbol empieza vacío
    while True:
        print("\n====== MENU ======")
        print("1. Agregar patentes")
        print("2. Mostrar árbol")
        print("3. Ver profundidad del árbol")
        print("4. Recorrido INORDEN")
        print("5. Recorrido PREORDEN")
        print("6. Recorrido POSTORDEN")
        print("7. Buscar patente")
        print("8. Salir")
```

4. Metodología Utilizada

El desarrollo del trabajo se llevó a cabo siguiendo una metodología progresiva, basada en etapas claramente definidas que nos permitieron avanzar de manera ordenada.

Investigación previa:
Iniciamos con una etapa de exploración teórica sobre árboles binarios, centrándonos en su funcionamiento y utilidad dentro de la programación. Consultamos fuentes confiables como Wikipedia y sitios especializados como W3Schools para comprender en profundidad cómo se construyen, recorren y visualizan este tipo de estructuras, y qué ventajas ofrecen en la organización de datos.

Diseño y prueba del código:
A partir de esa base teórica, diseñamos el programa en Python y elegimos representar la estructura utilizando listas.
El código se organizó en módulos, incorporando funciones recursivas tanto para insertar patentes como para realizar los recorridos típicos.
Durante la etapa de prueba, ejecutamos múltiples simulaciones con distintos datos para comprobar el correcto ordenamiento de las patentes y la visualización jerárquica en consola. También utilizamos la plataforma online-python.com para realizar pruebas rápidas y compartir avances de forma remota.

Herramientas y recursos utilizados:
El proyecto fue desarrollado utilizando Visual Studio Code como entorno principal. Todo el código se escribió en Python puro, sin librerías externas.
El control de versiones y el trabajo colaborativo se gestionaron mediante GitHub, donde también almacenamos el material complementario del proyecto.
Además, generamos un video explicativo utilizando OBS Studio para la grabación de pantalla, y herramientas como grabadoras de voz para registrar el audio. Las imágenes ilustrativas de la presentación fueron creadas con Gemini.
Para la organización de archivos y respaldo de documentos compartidos, usamos Google

Drive como sistema de almacenamiento en la nube.

Trabajo colaborativo:
El trabajo fue realizado en conjunto, tanto en el diseño, desarrollo del código como en la documentación. Las tareas se distribuyeron equitativamente: uno de nosotros se centró en el menú interactivo y la lógica del árbol, mientras el otro se encargó de los recorridos y la presentación final del proyecto. La coordinación se sostuvo mediante reuniones virtuales, uso compartido del repositorio Git y herramientas colaborativas como Google Drive.

---

5. Resultados Obtenidos

El desarrollo del caso práctico nos permitió implementar con éxito un sistema de gestión y búsqueda de patentes de autos utilizando un árbol binario como estructura principal.

Funcionamiento general:
El programa logró cumplir con todos los objetivos planteados: insertar patentes de forma ordenada, visualizarlas jerárquicamente en consola y recorrer el árbol en los tres modos clásicos (inorden, preorden y postorden). También se logró integrar un menú interactivo que guía al usuario de manera sencilla a través de las diferentes funciones del sistema.

Casos de prueba:
Probamos el sistema con distintos conjuntos de datos (mezclas de patentes alfanuméricas en diferente orden) para verificar que el árbol se construyera correctamente y que los recorridos devolvieran los resultados esperados.
Los datos se cargaron manualmente para validar tanto la lógica de inserción como la representación estructural.

Errores corregidos:

Durante el desarrollo se presentaron algunos errores en la visualización jerárquica y en la comparación de patentes alfabéticas, especialmente al tratar letras mayúsculas/minúsculas.

Además, fue necesario tener en cuenta que las patentes argentinas pueden responder a dos formatos distintos: el formato viejo (tres letras y tres números, como "ABC123") y el formato nuevo (dos letras, tres números y dos letras, como "AB123CD"). Para evitar errores y garantizar la correcta comparación entre ambos tipos, se implementó un proceso de normalización de patentes antes de insertarlas en el árbol.

También se depuraron errores menores en el flujo del menú y en algunas llamadas recursivas que no contemplaban correctamente la condición base.

Enlace al Repositorio de GIThub:

https://github.com/JohnTheMano/arboles_binarios.git

---

6. Conclusiones

Este trabajo nos dejó mucho aprendizaje: pudimos entender a fondo cómo funcionan los árboles binarios y su utilidad para organizar datos de forma eficiente. Aplicarlo en Python y ver cómo mejora la búsqueda nos motivó a seguir profundizando en estructuras de datos.

El tema que trabajamos es muy útil no solo para programación académica sino también para proyectos reales que requieren manejar grandes volúmenes de datos con rapidez.

Durante el desarrollo, enfrentamos desafíos en la implementación de la recursividad y la visualización del árbol, pero los superamos con investigación y pruebas continuas.

También pensamos en extender el proyecto para manejar otro tipo de datos o incluir interfaces gráficas.

En definitiva, esta experiencia nos fortaleció como programadores, nos enseñó la importancia de la organización y documentación, y nos preparó para futuros desafíos en desarrollo y análisis de estructuras complejas.

---

7. Bibliografía

Python Software Foundation. (2024). Documentación de Python 3. Recuperado el 8 de junio de 2025, de https://docs.python.org/3/

Sweigart, A. (2019). Automatiza tareas aburridas con Python. No Starch Press.

Wikipedia. (2025). Árbol binario. Recuperado el 6 de junio de 2025, de https://es.wikipedia.org/wiki/Árbol_binario

W3Schools. (2025). Recorridos en árboles binarios. Recuperado el 6 de junio de 2025, de https://www.w3schools.com/dsa/dsa_data_binarytrees.php

UTN - Facultad Regional San Nicolás.

---

8. Anexos

```
==============================
    D--> ZRT867
D--> RTS345
    I--> POM564
        I--> OBS456
KAS568
    D--> AG654BG
I--> AD768BF
    I--> AA453BF
==============================
Patente:
```

```
====== MENU ======
1. Agregar patentes
2. Mostrar árbol
3. Ver profundidad del árbol
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
3
Profundidad del árbol: 5

Presioná Enter para volver al menú...
```

```
====== MENU ======
1. Agregar patentes
2. Mostrar árbol
3. Ver profundidad del árbol
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
4
Recorrido INORDEN: ['AD756BG', 'KAS564', 'OPS546', 'OPS590', 'OPZ555', 'ORS243']

Presioná Enter para volver al menú...
```

```
====== MENU ======
1. Agregar patentes
2. Mostrar árbol
3. Ver profundidad del árbol
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
5
Recorrido PREORDEN: ['KAS564', 'AD756BG', 'OPS546', 'ORS243', 'OPZ555', 'OPS590']

Presioná Enter para volver al menú...
```

```
====== MENU ======
1. Agregar patentes
2. Mostrar árbol
3. Ver profundidad del árbol
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
6
Recorrido POSTORDEN: ['AD756BG', 'OPS590', 'OPZ555', 'ORS243', 'OPS546', 'KAS564']

Presioná Enter para volver al menú...
```

```
====== MENU ======
1. Agregar patentes
2. Mostrar árbol
3. Ver profundidad del árbol
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
7
Ingresá la patente a buscar:
ad756bg
La patente AD756BG SÍ está en el árbol.

Presioná Enter para volver al menú...
```

```
4. Recorrido INORDEN
5. Recorrido PREORDEN
6. Recorrido POSTORDEN
7. Buscar patente
8. Salir
Elegí una opción:
2

Árbol binario de búsqueda:
    D--> ORS243
        I--> OPZ555
            I--> OPS590
D--> OPS546
KAS564
I--> AD756BG

Presioná Enter para volver al menú...
```