Trabajo Práctico Integrador – Programación I

Tema: Árboles binarios con listas anidadas
Alumno: Nicolás Mónaco
Comisión: 17
Docente: Julieta Trape
Tutor: David Lopez
Fecha: Junio 2025

---

1. Introducción

En este trabajo se implementa una estructura de árbol binario utilizando **listas anidadas en Python**. El objetivo es representar un **sistema de inventario para una ferretería**, permitiendo almacenar productos y su información asociada (nombre, stock y precio) en una **jerarquía lógica**. El árbol se construye dinámicamente a partir del ingreso de datos por parte del usuario. Se incluyen funciones para insertar elementos, visualizar la estructura y recorrer el árbol mediante distintos tipos de recorrido: **preorden, inorden y postorden**.

---

2. Objetivos

- Aplicar conocimientos sobre listas y estructuras de datos no lineales.
- Construir un árbol binario sin utilizar programación orientada a objetos.
- Permitir el ingreso dinámico de productos a través de la consola.
- Implementar funciones para inserción, visualización y recorrido del árbol.
- Simular una jerarquía organizada de productos de ferretería.

---

3. Marco teórico

Un **árbol binario** es una estructura de datos jerárquica en la que cada nodo tiene como máximo dos hijos: **izquierdo y derecho**. Esta estructura es eficiente para organizar, insertar y buscar información.

En este trabajo, cada nodo se representa como una lista con cinco elementos:

python

CopiarEditar

```
[nombre, stock, precio, hijo_izquierdo, hijo_derecho]
```

La construcción y recorrido del árbol se realiza mediante **funciones y estructuras condicionales**, sin clases ni objetos. La representación con listas anidadas permite mantener la jerarquía de forma clara y flexible.

---

4. Desarrollo

Se desarrolló un programa en Python con las siguientes funciones:

python

CopiarEditar

```
def crear_nodo(nombre, stock, precio):
    return [nombre, stock, precio, [], []]
```

Crea un nuevo nodo con nombre, stock y precio, e inicializa sus hijos como listas vacías.

python

CopiarEditar

```
def insertar_izquierda(raiz, nombre, stock, precio):
    t = raiz[3]
    if t:
        raiz[3] = [nombre, stock, precio, t, []]
    else:
        raiz[3] = [nombre, stock, precio, [], []]
```

Inserta un nuevo nodo a la izquierda. Si ya hay un hijo izquierdo, lo reubica como hijo del nuevo nodo.

python

CopiarEditar

```
def insertar_derecha(raiz, nombre, stock, precio):

    t = raiz[4]

    if t:

        raiz[4] = [nombre, stock, precio, [], t]

    else:

        raiz[4] = [nombre, stock, precio, [], []]
```

Funciona igual que la función anterior, pero para el hijo derecho.

python

CopiarEditar

```
def mostrar_arbol(nodo, nivel=0):

    if nodo:

        print("  " * nivel + f"-{nodo[0]} (stock: {nodo[1]}, precio: 
${nodo[2]})")

        mostrar_arbol(nodo[3], nivel + 1)

        mostrar_arbol(nodo[4], nivel + 1)
```

Muestra el árbol en consola, con indentación según la profundidad.

Recorridos:

python

CopiarEditar

```
def recorrido_preorden(nodo):
```

```
    if nodo:
        print(nodo[0])
        recorrido_preorden(nodo[3])
        recorrido_preorden(nodo[4])
```

python

CopiarEditar

```
def recorrido_inorden(nodo):
    if nodo:
        recorrido_inorden(nodo[3])
        print(nodo[0])
        recorrido_inorden(nodo[4])
```

python

CopiarEditar

```
def recorrido_postorden(nodo):
    if nodo:
        recorrido_postorden(nodo[3])
        recorrido_postorden(nodo[4])
        print(nodo[0])
```

Ingreso de datos:
El ingreso se realiza en un bucle `while True`, donde se solicita al usuario el nombre del producto, el stock, el precio y la posición deseada (izquierda o derecha respecto a la raíz). El bucle se interrumpe si el usuario indica que desea finalizar.

---

5. Ejemplo de ejecución

Árbol generado con tres productos:

less

CopiarEditar

```
-Martillo (stock: 12, precio: $850)
  -Clavos (stock: 100, precio: $50)
  -Destornillador (stock: 30, precio: $500)
```

Recorrido preorden:

nginx

CopiarEditar

```
Martillo
Clavos
Destornillador
```

Recorrido inorden:

nginx

CopiarEditar

```
Clavos
Martillo
Destornillador
```

Recorrido postorden:

nginx

CopiarEditar

```
Clavos

Destornillador

Martillo
```

---

6. Conclusión

Este trabajo permitió **comprender y aplicar el funcionamiento de los árboles binarios** utilizando listas anidadas. Esta estructura fue adecuada para representar jerarquías, como las que puede tener el inventario de una ferretería.

Se reforzaron conceptos clave como:

- Funciones recursivas
- Estructuras condicionales
- Entrada dinámica de datos
- Recorridos clásicos de árboles

La implementación sin POO facilitó el enfoque funcional y la comprensión detallada del flujo del programa.

Reflexión personal

Este trabajo no fue solo una práctica de programación, sino también un desafío personal. En la primera versión del proyecto, el uso del árbol binario no estaba completamente justificado. Gracias a la oportunidad de mejorarlo, pude **reformular la estructura**, incorporando **categorías y subcategorías**, lo que le dio mucho más sentido al uso de un árbol binario.

Además, aprendí a **editar videos**, grabar explicaciones claras y estructurar mejor mis ideas. Aunque me llevó **mucho tiempo y esfuerzo**, me siento conforme con el resultado, porque no solo entregué un proyecto más lógico, sino que además **incorporé nuevas habilidades técnicas y comunicacionales**.

Este proyecto me ayudó a entender de manera profunda la utilidad de estructuras como los árboles, y cómo pueden aplicarse en situaciones reales.

---

7. Anexo: Código fuente

python

CopiarEditar

```
def crear_nodo(nombre, stock, precio):
    return [nombre, stock, precio, [], []]

def insertar_izquierda(raiz, nombre, stock, precio):
    t = raiz[3]
    if t:
        raiz[3] = [nombre, stock, precio, t, []]
    else:
        raiz[3] = [nombre, stock, precio, [], []]

def insertar_derecha(raiz, nombre, stock, precio):
    t = raiz[4]
    if t:
        raiz[4] = [nombre, stock, precio, [], t]
    else:
        raiz[4] = [nombre, stock, precio, [], []]

def mostrar_arbol(nodo, nivel=0):
    if nodo:
        print("   " * nivel + f"-{nodo[0]} (stock: {nodo[1]}, precio:
${nodo[2]})")
```

```
        mostrar_arbol(nodo[3], nivel + 1)
        mostrar_arbol(nodo[4], nivel + 1)

def recorrido_preorden(nodo):
    if nodo:
        print(nodo[0])
        recorrido_preorden(nodo[3])
        recorrido_preorden(nodo[4])

def recorrido_inorden(nodo):
    if nodo:
        recorrido_inorden(nodo[3])
        print(nodo[0])
        recorrido_inorden(nodo[4])

def recorrido_postorden(nodo):
    if nodo:
        recorrido_postorden(nodo[3])
        recorrido_postorden(nodo[4])
        print(nodo[0])

# Ejemplo de uso
raiz = crear_nodo("Martillo", 12, 850)
insertar_izquierda(raiz, "Clavos", 100, 50)
```

```
insertar_derecha(raiz, "Destornillador", 30, 500)


print("Árbol generado:")

mostrar_arbol(raiz)


print("\nRecorrido preorden:")

recorrido_preorden(raiz)


print("\nRecorrido inorden:")

recorrido_inorden(raiz)


print("\nRecorrido postorden:")

recorrido_postorden(raiz)
```

---

8. Mejoras posibles

- Reescribir el programa usando **programación orientada a objetos** para mejorar modularidad y legibilidad.
- Agregar funciones para **eliminar** o **modificar** productos existentes.
- Implementar búsquedas por nombre u otras características.
- Crear una **interfaz gráfica** para facilitar la interacción.
- Validar datos ingresados (por ejemplo, que el stock y el precio sean positivos).
- Ampliar a **árboles no binarios** para jerarquías más complejas.
- Implementar funciones para **guardar y cargar** el árbol desde archivos.