Marco Teórico: Implementación de un Árbol Binario con Listas Anidadas en Python

1. Introducción a las Estructuras de Datos

Las estructuras de datos son un concepto fundamental en informática y permiten organizar, gestionar y almacenar información de manera eficiente. Dentro de las estructuras de datos más comunes se encuentran:

- **Listas**: Colecciones ordenadas de elementos accesibles por índice.
- **Diccionarios**: Pares clave-valor para búsquedas rápidas.
- **Pilas y Colas**: Manejo de datos bajo reglas específicas (LIFO y FIFO).
- **Árboles**: Modelos jerárquicos que permiten estructurar datos de manera eficiente.

¿Por qué un Árbol Binario?

Los árboles binarios son útiles cuando se requiere **organizar información en una estructura jerárquica**, como un catálogo de productos o una organización de categorías.

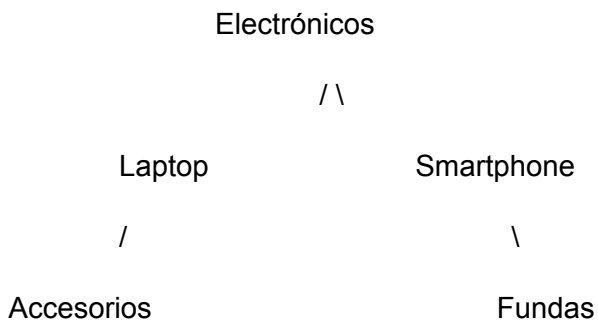
2. Concepto de Árbol Binario

Un **árbol binario** es una estructura de datos donde cada nodo tiene **como máximo dos hijos**: izquierdo y derecho. Se representa comúnmente como una colección de elementos que cumplen las siguientes propiedades:

Cada nodo almacena datos (en este caso, nombre del producto, stock y precio).
Los nodos pueden tener hijos, formando una jerarquía.
Se pueden realizar búsquedas eficientes si el árbol está ordenado adecuadamente.

Ejemplo de un árbol binario en un sistema de inventario

Imagina que estás administrando una tienda y quieres organizar los productos de forma jerárquica:



Aquí, la categoría "Electrónicos" contiene dos productos principales: Laptop y Smartphone. A su vez, estos productos pueden tener subcategorías, como Accesorios para Laptop y Fundas para Smartphone.

Fuente referencial:

- "Data Structures and Algorithm Analysis in Python" – Mark Allen Weiss

- "Introduction to Algorithms" – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

3. Implementación con Listas Anidadas en Python

Tu código utiliza **listas anidadas** en lugar de estructuras avanzadas como clases u objetos. Este método permite representar un árbol binario de manera simple:

def binary_tree(product_name, stock, price): return [product_name, stock, price, [], []] # Nodo con hijos vacíos

Cada nodo es una lista donde:

- **Posición 0:** Nombre del producto.
- **Posición 1:** Stock disponible.
- **Posición 2:** Precio.
- **Posición 3:** Hijo izquierdo.
- **Posición 4:** Hijo derecho.

Fuente referencial:

- "Algorithms and Data Structures" – Niklaus Wirth

4. Limitaciones del Código

Si bien la implementación con listas anidadas cumple su propósito básico, **tiene varias limitaciones**:

4.1. Acceso Ineficiente a los Datos

Para encontrar un producto específico, el código debe **recorrer manualmente el árbol**, lo cual es **poco eficiente** en estructuras grandes.

Solución:
Implementar un **Árbol de Búsqueda Binario (BST)**, donde los nodos se organizan de manera ordenada según el precio o stock.

4.2. Falta de Función de Eliminación

No hay una función para **eliminar productos del inventario**, lo que significa que una vez insertado un producto, **permanece en la estructura sin posibilidad de actualización**.

Solución:
Crear una función que elimine nodos y reorganice el árbol adecuadamente.

4.3. No hay persistencia de datos

Cuando el programa se cierra, **todos los datos desaparecen**. Esto es un problema grave en un sistema de inventario.

Solución:
Guardar los datos en **archivos JSON o bases de datos** para mantenerlos accesibles en futuras ejecuciones.

4.4. No hay validación de datos

Actualmente, el código permite insertar cualquier valor, incluso si:

El stock es negativo.
El precio es incorrecto.
Un producto ya existe en el árbol.

Solución:
Agregar restricciones que validen la entrada antes de insertar datos.

5. Mejoras y Escalabilidad

Para que tu código sea más robusto y escalable en aplicaciones reales, se pueden implementar varias mejoras:

5.1. Uso de Clases en lugar de Listas

El código sería más **estructurado y mantenible** si usamos **clases y objetos**:

class Nodo: def __init__(self, nombre, stock, precio): self.nombre = nombre self.stock = stock self.precio = precio self.izquierdo = None self.derecho = None

Fuente referencial:

- "Data Structures and Algorithms in Python" – Michael T. Goodrich

5.2. Implementación de un Árbol de Búsqueda Binario (BST)

En un BST, los nodos se organizan de manera ordenada:

Electrónicos / \ Laptop Smartphone / \ / \ Accesorios Mouse Funda Auriculares

Esto permite realizar búsquedas y actualizaciones **mucho más rápido**.

Fuente referencial:

- "Introduction to the Design and Analysis of Algorithms" – Anany Levitin

5.3. Integración con Bases de Datos

Si el sistema de inventario crece, almacenar datos en archivos no será suficiente. Es recomendable **usar bases de datos SQL o NoSQL** para gestionar productos de manera segura.

Solución:
Utilizar SQLite o MongoDB para gestionar el árbol de productos de forma persistente.

6. Conclusión

El código actual ofrece una **implementación básica de un árbol binario**, útil para organizar productos en categorías. Sin embargo, **su escalabilidad y eficiencia son limitadas**, por lo que es recomendable mejorar aspectos como:

Uso de clases y objetos para mayor flexibilidad.
Implementación de búsqueda eficiente con BST.
Persistencia de datos en archivos o bases de datos.
Funciones adicionales para eliminación y validación de productos.

CASO PRÁCTICO:

```
# Definimos una función para crear un "nodo" del árbol binario.
# Un nodo representa un producto y almacena su nombre, cantidad en stock y precio.
# Además, contiene dos espacios vacíos para los hijos izquierdo y derecho del árbol.
def binary_tree(product_name, stock, price):
    return [product_name, stock, price, [], []]  # Lista con el nombre del producto, stock y precio, más dos listas vacías (hijos)

# Función para insertar un nuevo producto en el lado IZQUIERDO del árbol.
def insert_left(root, product_name, stock, price):
    t = root[3]  # Obtenemos el hijo izquierdo actual.
    if t:  # Si ya existe un hijo izquierdo, lo desplazamos hacia abajo y agregamos el nuevo producto arriba.
        root[3] = [product_name, stock, price, t, []]
    else:  # Si el espacio está vacío, simplemente agregamos el nuevo producto.
        root[3] = [product_name, stock, price, [], []]
    return root

# Función para insertar un nuevo producto en el lado DERECHO del árbol.
def insert_right(root, product_name, stock, price):
    t = root[4]  # Obtenemos el hijo derecho actual.
    if t:  # Si ya existe un hijo derecho, lo desplazamos hacia abajo y agregamos el nuevo producto arriba.
        root[4] = [product_name, stock, price, [], t]
    else:  # Si el espacio está vacío, simplemente agregamos el nuevo producto.
        root[4] = [product_name, stock, price, [], []]
    return root
```

```
# Función para obtener la cantidad en stock de un producto dado.
def get_stock(root):
    return root[1]  # Devolvemos el valor del stock almacenado en la posición 1 de la lista.

# Función para actualizar el stock de un producto específico.
def update_stock(root, new_stock):
    root[1] = new_stock  # Modificamos el valor del stock con el nuevo número.

# Función para obtener el precio de un producto.
def get_price(root):
    return root[2]  # Devolvemos el precio almacenado en la posición 2 de la lista.

# Función para actualizar el precio de un producto.
def update_price(root, new_price):
    root[2] = new_price  # Modificamos el precio con el nuevo valor.

# Función para obtener el hijo izquierdo de un nodo del árbol.
def get_left_child(root):
    return root[3] if root[3] else None  # Retorna el hijo izquierdo si existe, sino retorna "None" (vacío).

# Función para obtener el hijo derecho de un nodo del árbol.
def get_right_child(root):
    return root[4] if root[4] else None  # Retorna el hijo derecho si existe, sino retorna "None".

# ---------- EJEMPLO DE USO ----------

# Creamos la "categoría principal" del árbol: Electrónicos (sin stock ni precio inicial).
almacen = binary_tree("Electrónicos", 0, 0)

# Insertamos dos productos dentro de la categoría principal:
insert_left(almacen, "Laptop", 10, 800)  # Agrega una Laptop al lado izquierdo.
insert_right(almacen, "Smartphone", 15, 600)  # Agrega un Smartphone al lado derecho.

# Agregamos más productos dentro de los existentes.
insert_left(get_left_child(almacen), "Accesorios Laptop", 20, 50)  # Accesorios para laptop en el lado izquierdo de Laptop.
insert_right(get_right_child(almacen), "Fundas Smartphone", 25, 20)  # Fundas para smartphone en el lado derecho de Smartphone.

# Mostramos la cantidad de stock de un producto específico.
```

```
print(f"Stock de Laptop: {get_stock(get_left_child(almacen))}")  # Devuelve "10"

# Actualizamos el stock de Laptop y lo volvemos a mostrar.
update_stock(get_left_child(almacen), 8)  # Modificamos el stock de la Laptop a 8 unidades.
print(f"Stock actualizado de Laptop: {get_stock(get_left_child(almacen))}")  # Devuelve "8"
```