Árboles con Orientación a Objetos en Python – Programación I

• **Alumnos:**

Herrera Gonzalez Franco Esteban

fh745543@gmail.com - francoesteban.herrera@tupad.utn.edu.ar

Gonzalez Rivero Anthony José – agonzr7@gmail.com

• **Materia:** Programación I

• **Profesor/a:** Ariel Enferrel

• **Fecha de Entrega:** 9/6/2025

Índice

INTRODUCCIÓN

En el presente trabajo se aborda la estructura de datos árbol en el contexto del lenguaje Python, motivado por su importancia fundamental en la informática y su amplia aplicabilidad. Los árboles son considerados una de las estructuras de datos más utilizadas, aunque también de las más complejas, debido a que organizan la información de forma jerárquica en lugar de lineal. A diferencia de arreglos o listas donde los datos se disponen secuencialmente, un árbol almacena sus elementos (*nodos*) en niveles, simulando una jerarquía de relaciones padre-hijo. Esta característica permite modelar problemas de forma más natural cuando existe una relación jerárquica entre los datos, por ejemplo, en la representación de organizaciones, taxonomías o estructuras de ficheros.

La elección de investigar sobre árboles se fundamenta en que ofrecen ventajas notables en ciertas operaciones. En particular, muchos tipos de árboles (como los binarios de búsqueda) permiten realizar búsquedas, inserciones y eliminaciones de manera rápida y eficiente, típicamente en tiempo logarítmico respecto al número de elementos. En aplicaciones de software, aparecen en numerosos contextos: algoritmos de decisión, análisis sintáctico de lenguajes de programación, inteligencia artificial, entre otros.

Esta investigación tiene como objetivo presentar una *fundamentación teórica* clara sobre la estructura de datos árbol, describiendo sus definiciones formales, terminología básica, variantes y particularidades de su implementación en Python. Asimismo, se desarrolla un *caso práctico* que consiste en la implementación de un árbol binario de búsqueda en Python para resolver un problema concreto, incluyendo el código fuente comentado, las decisiones de diseño adoptadas y la validación de su correcto funcionamiento.

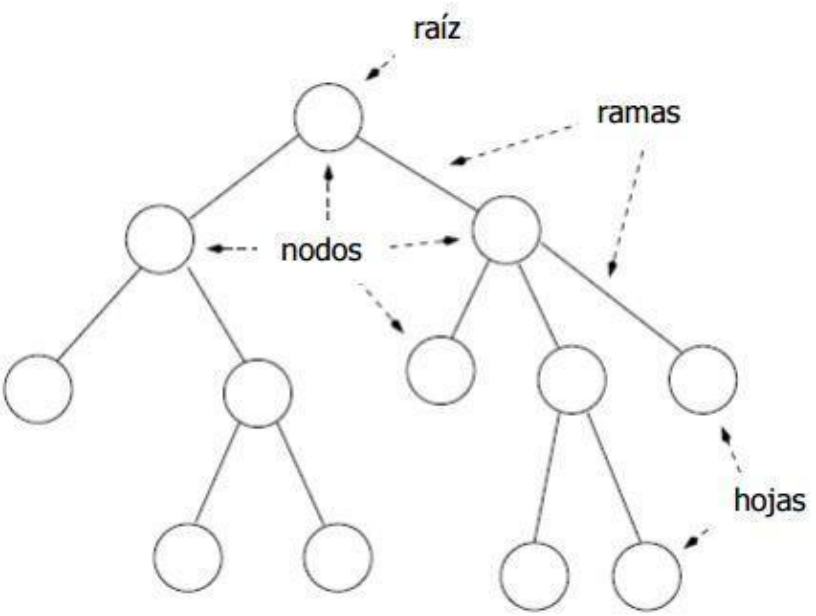
MARCO TEÓRICO

Definición y Conceptos Básicos.

Un árbol, en términos de estructura de datos, se define como una colección no lineal de elementos interconectados que simula una jerarquía. A diferencia de las estructuras lineales como los arreglos, donde cada elemento tiene un sucesor y un predecesor, en un árbol cada elemento o nodo puede enlazar a múltiples nodos en niveles inferiores. Formalmente, un árbol se compone de un nodo distinguido llamado raíz y cero o más subárboles asociados a él; cada subárbol a su vez es un árbol con su propio nodo raíz. Esta definición es recursiva: cualquier nodo puede considerarse raíz de un subárbol que forma parte del árbol mayor.

Cada nodo en un árbol puede contener un valor o dato, y referencias (enlaces) hacia otros nodos. Si un nodo A tiene enlaces hacia nodos B, C, etc., entonces B, C, etc. Son llamados hijos de A, y A es el padre de dichos nodos. Un nodo puede tener múltiples hijos, pero un solo padre (excepto el nodo raíz, que no tiene padre). Los nodos sin ningún hijo se denominan hojas o nodos terminales, mientras que los nodos con al menos un hijo se llaman nodos internos o ramas. En la figura siguiente se ilustra la estructura general de un árbol y sus tipos de nodos:

Figura 1: Representación gráfica de un árbol genérico.



Nota. Representación gráfica de un árbol genérico con su nodo raíz (en la cima), nodos internos (ramas) y nodos hoja (extremos). Cada nodo contiene un valor (letras en el diagrama) y las conexiones representan relaciones padre-hijo. Por Ing. Fredy Román, (https://estructurasite.wordpress.com/arbol/).

Además de la relación jerárquica, en los árboles se definen métricas importantes. La profundidad (o nivel) de un nodo es la distancia (número de aristas o enlaces) desde la raíz hasta dicho nodo. Por convenio, la raíz tiene profundidad 0. La altura de un árbol es la longitud del camino más largo desde la raíz hasta alguna hoja; equivale al máximo nivel presente en el árbol. Por ejemplo, un árbol vacío tiene altura 0, un árbol con solo la raíz tiene altura 0, y así sucesivamente. La altura es un factor crítico para el rendimiento de muchas operaciones, pues determina cuán profundo puede ser el árbol en el peor caso. Otro concepto es el grado de un nodo, que indica el número de hijos que posee. El grado de un árbol completo se define como el máximo grado entre todos sus nodos. Cuando se establece un número fijo máximo de hijos por nodo, a ese valor se le denomina orden del árbol. Por ejemplo, en un árbol binario el orden es 2, ya que cada nodo puede tener como mucho dos hijos.

Clasificación de los árboles.

Existen múltiples tipos y clasificaciones de árboles, atendiendo a sus propiedades y usos:

- Árbol general (o multicamino): Es la forma más genérica de árbol, donde no hay un límite predefinido en la cantidad de hijos que puede tener un nodo. Este tipo de estructura es útil para modelar jerarquías sin restricciones, por ejemplo, la estructura de directorios de un sistema de archivos. En la práctica, para almacenamiento de discos externos, suelen emplearse árboles multicamino con orden alto, como los árboles B y B+, optimizados para minimizar accesos a disco. De hecho, mientras los árboles binarios se idearon para trabajar en memoria principal, los árboles multicamino (de orden mayor a 2) fueron

diseñados para manejar eficientemente grandes volúmenes de datos en sistemas de archivos y bases de datos.

- Árbol ordenado o no ordenado: Si a los hijos de cada nodo se les asigna un orden específico (es decir, si importa la posición relativa de cada hijo, por ejemplo "hijo izquierdo" vs "hijo derecho"), se dice que el árbol es ordenado. En caso contrario, cuando solo importa el conjunto de hijos, pero no su posición, se tiene un árbol no ordenado. Por defecto, en la literatura se asume que un árbol está ordenado a menos que se indique lo contrario.
- Árbol binario: Es un caso particular de árbol ordenado de orden 2, en el cual cada nodo puede tener a lo sumo dos hijos, típicamente designados como hijo izquierdo e hijo derecho. Los árboles binarios son muy importantes tanto teórica como prácticamente, ya que muchos otros tipos de árboles derivan de ellos. Dentro de esta categoría existen definiciones especiales: un árbol binario completo es aquel donde todos los niveles, excepto posiblemente el último, están llenos de nodos, y los nodos del último nivel se encuentran lo más a la izquierda posible; un árbol binario lleno es aquel en que cada nodo interno tiene exactamente dos hijos y todas las hojas están al mismo nivel; un árbol degenerado es un árbol binario donde cada nodo tiene solo un hijo, convirtiéndose efectivamente en una lista enlazada, etc. Estas clasificaciones impactan la forma y eficiencia del árbol.
- Árbol binario de búsqueda (ABB): Se trata de un árbol binario ordenado que mantiene sus nodos dispuestos de acuerdo con un criterio de orden específico, típicamente comparando claves. Formalmente, un árbol binario de búsqueda cumple la propiedad de orden BST “Binary Search Tree”: para cada nodo, todos los valores en su subárbol izquierdo son menores que el valor del nodo, y todos los valores en su subárbol derecho son mayores. Esta propiedad recursiva debe cumplirse en todos los nodos del árbol. Como consecuencia, el recorrido

inorden de un ABB produce la secuencia de valores ordenada de menor a mayor. Los ABB permiten búsquedas, inserciones y eliminaciones de elementos en tiempo promedio O(log n), lo que mejora significativamente el tiempo O(n) de una búsqueda lineal en una lista. Sin embargo, en el peor caso, por ejemplo, si el árbol se degenera en una estructura tipo lista enlazada, las operaciones pueden degradarse a O(n). Para mitigar esto existen árboles de búsqueda balanceados, como los árboles AVL o árboles rojo-negro, que garantizan alturas mínimas ajustando el árbol tras cada inserción/eliminación. Un árbol AVL, por ejemplo, es un ABB auto-balanceado que mantiene la diferencia de alturas entre subárboles de cualquier nodo como máximo en 1.

- Otras variantes: Además de los anteriores, el universo de árboles incluye estructuras especializadas como los árboles de decisión utilizados en inteligencia artificial y análisis predictivo, trie o árboles de prefijos optimizados para almacenar cadenas y facilitar búsquedas por prefijo, usados en aplicaciones como autocompletado, árboles de expresiones que representan la estructura sintáctica de expresiones matemáticas o lógicas, entre otros. Cada variante adapta la estructura básica del árbol a necesidades concretas, modificando las propiedades, el criterio de ordenamiento o las operaciones disponibles.

Operaciones y recorridos en árboles.

Las operaciones fundamentales sobre árboles incluyen: inserción de nuevos nodos, búsqueda de un nodo con determinado valor, y en algunos casos la eliminación de nodos. La implementación de estas operaciones depende del tipo de árbol. En un árbol binario de búsqueda, por ejemplo, la inserción consiste en ubicar el lugar apropiado para el nuevo valor comparando recursivamente: si el valor es menor que el nodo actual se dirige al subárbol izquierdo, si es mayor al subárbol derecho, hasta encontrar un nodo nulo donde insertar el nuevo nodo. De forma similar, la búsqueda recorre el árbol

comparando el valor buscado con el nodo actual y decidiendo ir a la izquierda o derecha según la propiedad BST.

Estas operaciones son típicamente implementadas de forma recursiva, aprovechando la naturaleza recursiva de la estructura de árbol. Alternativamente, pueden implementarse de manera iterativa con bucles, manteniendo un puntero al nodo actual. Una de las características más potentes de los árboles es la posibilidad de recorrer sus nodos en distintos órdenes. Existen recorridos profundos clásicos definidos recursivamente:

- Preorden, visitar primero el nodo actual, luego recorrer recursivamente el subárbol izquierdo y después el derecho.
- Inorden recorrer izquierdo, visitar nodo, recorrer derecho.
- Postorden recorrer izquierdo, recorrer derecho, visitar nodo.

En árboles binarios de búsqueda, el recorrido inorden produce los valores ordenados crecientemente por la propiedad de orden. También existe el recorrido en anchura o por niveles, que visita nodos nivel por nivel desde la raíz. La elección del recorrido depende de la aplicación: por ejemplo, en evaluación de expresiones aritméticas representadas en un árbol, un recorrido postorden permite calcular el resultado evaluando primero las subexpresiones.

Representación e implementación en Python

En lenguajes de programación de bajo nivel o fuertemente tipados (C, C++, Java, etc.), un árbol se construye típicamente definiendo una estructura o clase de nodo con campos para el dato y referencias a sus. Python, al ser un lenguaje dinámico, no proporciona en su biblioteca estándar una clase Árbol predefinida; sin embargo, ofrece la flexibilidad para implementar árboles de manera sencilla usando clases personalizadas o incluso tipos integrados como diccionarios o listas anidadas. La forma más clara y orientada a objetos es definir una clase Nodo que almacene el valor y una lista de hijos para un árbol general o referencias explícitas izquierdo y derecho para un árbol binario. Adicionalmente, suele definirse una clase contenedora para el árbol en sí.

Figura 2: Ejemplo de Árbol en Python con clase definida Nodo.

```python
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierdo = None    # Referencia al hijo izquierdo
        self.derecho   = None    # Referencia al hijo derecho
```

Esta clase permite crear nodos individuales. Con ella, un árbol binario se construye enlazando nodos: un nodo raíz contiene referencias a sus hijos, y estos a su vez a sus respectivos hijos, recursivamente. Sobre esta base, se implementan operaciones; por ejemplo, para insertar un valor en un ABB, se compara con el valor del nodo actual y se desciende recursivamente a la izquierda o derecha según corresponda hasta encontrar una posición vacía donde crear un nuevo nodo. A continuación, usualmente se implementan métodos de búsqueda y recorridos inorden, preorden, etc; de forma recursiva.

Vale la pena mencionar que existen librerías Python de terceros que facilitan el trabajo con árboles, como anytree o treelib, que proporcionan clases de nodos genéricos y métodos de recorrido; incluso el módulo estándar collections puede servir para implementar árboles mediante estructuras como diccionarios anidados. No obstante, para fines pedagógicos y de comprensión es valioso construir la estructura manualmente. En la siguiente sección se presenta un caso práctico completo de un árbol binario de búsqueda implementado en Python.

CASO PRÁCTICO

Implementación de un Árbol Binario de Búsqueda

Para ilustrar la aplicación de la teoría anterior, se desarrolla un caso práctico que consiste en implementar un árbol binario de búsqueda (ABB) en Python y aplicarlo a un problema concreto. El problema planteado es el siguiente:

En un “Ecom & Tech” tenemos que almacenar de forma eficiente un listado de productos que poseen distintas cantidades, precios y nombres que se asocian a códigos únicos identificatorios. Cada producto debe poder ser consultado y modificado, además, constantemente se solicitan nuevos productos para estar a la vanguardia del mercado.

Descripción del problema.

- Se requiere una estructura de datos que almacene de forma dinámica un conjunto de productos.

- Estos productos poseen datos asociados (código único identificatorio, nombre, precio y cantidad).
- El listado de productos crece constantemente.

Entradas y salidas:

Entradas:

- Funcionalidad requerida por el usuario.
- Datos relacionados al producto y funcionalidad requerida.

Salidas:

- Mensajes indicativos para realizar las acciones permitidas por el sistema junto a las limitaciones correspondientes.
- Mensajes de información, éxito y error según el caso.

Funcionalidades

El programa debe contar con las siguiente funcionalidades:

- Menú de opciones.
- Creación de nuevos productos con sus datos asociados.
- Búsqueda de productos específicos de manera eficiente.
- Modificación de productos existentes específicos de manera eficiente.
- Finalización del programa.

Código comentado:

```
class Producto:   #permite crear nuevos productos (objetos) con datos dados por el usuario
    def __init__(self, codigo, nombre, cantidad, precio):
        self.nombre = nombre    #se asigna un nombre al producto
        self.cantidad = cantidad    #se asigna una cantidad al producto
        self.precio = precio    #se asigna un precio al producto
        self.codigo = codigo    #se asigna un codigo al producto
        self.izquierda = None   #se crea una referencia para un futuro nodo hijo izquierdo
        self.derecha = None     #se crea una referencia para un futuro nodo hijo derecho
```

```
class ArbolBinario:     #permite crear un arbol binario para almacenar productos (nodos - objetos)
    def __init__(self):
        self.referencia = None  #se crea la referencia al primer nodo del arbol(raiz)

    def insertar(self, codigo, nombre, cantidad, precio):   #se reciben los datos dados por el usuario
        self.referencia = self.insertar_recursivo(self.referencia, codigo, nombre, cantidad, precio)    #se asigna el valor return de la funcion a la referencia.

    def insertar_recursivo(self, nodo, codigo, nombre, cantidad, precio):   #recibe los datos
        if nodo is None:        #si la referencia esta vacia
            return Producto(codigo, nombre, cantidad, precio)   #se crea un nuevo producto
        if codigo < nodo.codigo:        #si ya existe un nodo padre se añadiran nodos hijo izquierdos o derechos
            nodo.izquierda = self.insertar_recursivo(nodo.izquierda, codigo, nombre, cantidad, precio)  #llama recursivamente
        elif codigo > nodo.codigo:
            nodo.derecha = self.insertar_recursivo(nodo.derecha, codigo, nombre, cantidad, precio)      #llama recursivamente
        return nodo     #return nodo con hijo asignado

    def buscar(self, codigo):
        return self.buscar_recursivo(self.referencia, codigo) #se envia el nodo raiz y el codigo

    def buscar_recursivo(self, nodo, codigo):   #busca el nodo a partir del codigo dado
```

```
        if nodo is None:        #se llego a un nodo hoja o el nodo raiz esta vacio (el nodo buscado no existe)
            return None
        if codigo == nodo.codigo:   #se compara el codigo del nodo actual y el codigo del nodo buscado
            return nodo             #devuelve el nodo buscado si es encontrado
        elif codigo < nodo.codigo:      #recorre el arbol hacia el hijo izq del nodo actual
            return self.buscar_recursivo(nodo.izquierda, codigo)
        else:                           #recorre el arbol hacia el hijo der del nodo actual
            return self.buscar_recursivo(nodo.derecha, codigo)

    def modificar(self, prod, nombre, cantidad, precio):#permite modificar nodos existentes
            prod.nombre = nombre        #asigna el nuevo nombre dado
            prod.cantidad = cantidad    #asigna la nueva cantidad dada
            prod.precio = precio        #asigna el nuevo precio dado
            return True


def mostrar_menu():     #funcion que permite imprimir el menu de opciones
    print("\n----TP-Integrador---- MENÚ ----")
    print("1. Agregar producto")
    print("2. Buscar producto")
    print("3. Modificar producto")
    print("4. Salir")
    print("--------------------------------")
```

```
arbol = ArbolBinario()

# Se crean productos de ejemplo

arbol.insertar(10, "computadora escritorio", 8, 1200) # (codigo de producto, "nombre", cantidad, precio)

arbol.insertar(11, "tv smart vision 60p 4k", 22, 1000)

arbol.insertar(15, "disco SSD 2tb", 18, 2200)



estado_programa = True

while estado_programa:          #estructura repetitiva que permite siempre volver al menu principal hasta que el usuario finalice el programa

    mostrar_menu() #se llama a la funcion mostrar_menu

    opcion = input("Seleccione una opción: ") #se solicita la opcion al usuario


        #se ejecuta la opcion elegida a partir de de una estructura if

    if opcion == "1":       #opción: crear producto

        try:
#maneja errores en el ingreso de datos

            codigo = int(input("Ingrese código del producto (número entero): "))        #solicta el codigo del producto

            prod = arbol.buscar(codigo)
#busca si el código ingresado existe en el arbol

            if prod:
#si el código pertenece a otro producto

                print(f"\nEl codigo ingresado ya pertenece a un producto existente: {prod.nombre}")

            else:
#si el código esta disponible

                nombre = input("Ingrese nombre del producto: ")
#solicta el nombre del producto

                cantidad = int(input("Ingrese cantidad del producto (número entero): "))    #solicta la cantidad del producto
```

```
                precio = float(input("Ingrese precio del
producto(sin '$'): "))              #solicta el precio del producto
                arbol.insertar(codigo, nombre, cantidad, precio)
#envia todos los datos ingresados la funcion insertar
                print("\nProducto agregado con éxito: ")
                prod = arbol.buscar(codigo)
#muestra el producto creado a partir de la funcion buscar
                print(f"Código: {prod.codigo}")
                print(f"Nombre: {prod.nombre}")
                print(f"Cantidad: {prod.cantidad}")
                print(f"Precio: ${prod.precio}")
        except ValueError:
#mensaje en caso de error
            print("Ingrese datos validos.")


    elif opcion == "2":     #opción: buscar producto
        try:
#maneja errores en el ingreso de datos
            codigo = int(input("Ingrese código del producto a
buscar: "))   #solicita el codigo asociado al producto buscado
            prod = arbol.buscar(codigo)
#llama a la funcion buscar enviando el codigo
            if prod:
#muestra los atributos si existe el producto
                print(f"\nProducto encontrado:")
                print(f"Código: {prod.codigo}")
                print(f"Nombre: {prod.nombre}")
                print(f"Cantidad: {prod.cantidad}")
                print(f"Precio: ${prod.precio}")
            else:
#cuando el producto no es encontrado
                print("Producto no encontrado.")
        except ValueError:
#mensaje en caso de error
```

```
            print("Ingrese datos validos.")


    elif opcion == "3":     #opción: modificar producto
        try:
#maneja errores en el ingreso de datos
            codigo = int(input("Ingrese código del producto a
modificar: "))    #solicita el codigo asociado al producto a
modificar
            prod = arbol.buscar(codigo)
#identifica la existencia del producto y sus atributos
            if prod:
#si existe solicita los nuevos atributos (excepto el código de
producto)
                nombre = input(f"Nuevo nombre (actual:
{prod.nombre}): ")
                cantidad = int(input(f"Nueva cantidad (actual:
{prod.cantidad}): "))
                precio = input(f"Nuevo precio (actual:
{prod.precio}): ")
                arbol.modificar(prod, nombre, cantidad, precio)
#opcion de enviar el "prod" y no el "codigo"
                print("Producto modificado con exito.")
            else:
#en caso de que el producto no exista
                print("Producto no encontrado.")
        except ValueError:
#mensaje en caso de error
            print("Ingrese datos validos.")


    elif opcion == "4":     #opción: salir
        print("Saliendo del programa...")
        estado_programa = False      #finaliza el ciclo while y
termina el programa
    else:
```

```
        print("ingrese una de las opciones disponibles.")    #cuando
el usuario ingresa valores invalidos
```

METODOLOGÍA UTILIZADA

- Se realizó una búsqueda de información sobre la forma de implementación de árboles en Python, consultando principalmente videos de youtube y material de la cátedra.
- Se definió un problema a resolver de mutuo interés.
- Se comprendió el problema y se identificó el objetivo de la solución.
- Se identificaron las entradas y salidas que deberían existir.
- Se especificaron las distintas funcionalidades que debería tener el código.
- Se comenzó con la creación del código, funcionalidad por funcionalidad.
- Se realizó la conjunción de todas las funcionalidades y se creó una interfaz apta para realizar las pruebas pertinentes.
- Se ejecutaron distintas pruebas para verificar la correcta funcionalidad del programa.
- Se encontraron errores y se corrigieron modificando el código.
- Con el código funcionando correctamente se realizó una revisión general para su posible optimización.
- Se identificaron y ejecutaron las optimizaciones encontradas.
- Se realizaron pruebas finales para garantizar el correcto funcionamiento.
- Se realizó la documentación.

RESULTADOS OBTENIDOS

Link al repositorio de Github:
https://github.com/frann-ayb/Trabajo-Pr-ctico_Integrador_Programaci-n

Error detectados y solución Aplicada

Durante las primeras pruebas del programa, se identificó un comportamiento incorrecto al intentar añadir un nuevo producto utilizando un código ya asignado a otro producto existente. En estos casos, el programa continuaba su ejecución sin interrumpirse ni generar errores críticos, pero mostraba un mensaje de éxito al usuario, aunque el producto no era añadido correctamente, generando confusión y dando lugar la posibilidad de datos inconsistentes.

Este inconveniente fue solucionado mediante la implementación de una verificación previa que valida si el código ingresado ya está presente en el árbol. En caso afirmativo, el sistema informa al usuario y no permite la inserción, garantizando así la unicidad de los códigos de producto y reforzando la integridad de la estructura de datos.

```
if opcion == "1":       #opción: crear producto

        try:
#maneja errores en el ingreso de datos

            codigo = int(input("Ingrese código del producto (número
entero): "))         #solicta el codigo del producto

            prod = arbol.buscar(codigo)
#busca si el código ingresado existe en el arbol

            if prod:
#si el código pertenece a otro producto

                print(f"\nEl codigo ingresado ya pertenece a un
producto existente: {prod.nombre}")
```

Mejora en el control del flujo del programa

En versiones iniciales y para la facilidad del desarrollo, el menú principal del sistema se ejecutaba mediante un bucle `while True:`, interrumpido

únicamente a través de la instrucción `break` al seleccionar la opción de salida. Si bien era funcional, este enfoque dificulta la lectura del código y limita la claridad del control de flujo.

Con el objetivo de mejorar la comprensión y el mantenimiento del programa, se optó por implementar una variable de control denominada:

```
estado_programa = True
```

El ciclo principal pasó a estructurarse como:

```
while estado_programa:
```

La condición de finalización se gestionó asignando:

```
elif opcion == "4":     #opción: salir

        print("Saliendo del programa...")

        estado_programa = False      #finaliza el ciclo while y
termina el programa
```

Este cambio permite una gestión más explícita del estado del programa, facilitando su seguimiento, escalabilidad y depuración.

Implementaciones

Por otro lado se ejecutó la correcta construcción e implementación del Árbol Binario de Búsqueda, se lograron integrar satisfactoriamente una serie de funcionalidades clave orientadas a mejorar la experiencia del usuario y optimizar el desempeño del sistema. Estas incorporaciones permitieron validar tanto la utilidad práctica del programa como su capacidad para adaptarse a distintos escenarios de uso:

- Menú para el usuario: Se incorporó un menú simple y ordenado que guía al usuario durante el uso del programa, facilitando el acceso a las funciones principales sin dificultades.
- Búsquedas eficientes**:** Gracias a la aplicación de la estructura del ABB, encontrar un producto por su código es un proceso ágil, sin necesidad

de recorrer toda la lista. Esto hace que el sistema se mantenga eficiente incluso a medida que crece la cantidad de datos.

- Modificar información: El sistema permite modificar los datos de productos ya cargados (nombre, cantidad y precio) de manera sencilla, manteniendo la estructura general del árbol sin alteraciones.
- Pruebas con datos reales**:** Se realizaron pruebas tanto con entradas válidas como inválidas para asegurarse de que el sistema respondiera correctamente ante distintos escenarios.

Posibles mejoras:

- Incorporar una funcionalidad para eliminar productos añadidos anteriormente.
- Incorporar un sistema de verificación para evitar la duplicidad de nombres
- Incorporar una funcionalidad para garantizar el ingreso de nodos de forma balanceada sobre la totalidad de datos.
- Incorporar un sistema de reducción de cantidades por unidades para integrar el proceso de ventas.
- Incorporar un sistema de alertas al llegar a puntos de pedido asignados por el usuario.

CONCLUSIONES

La implementación de un Árbol Binario de Búsqueda (ABB) en Python permitió transformar conceptos teóricos en una solución práctica y funcional para la gestión estructurada de productos. Al desarrollar un sistema capaz de almacenar, buscar y modificar productos identificados por un código único y atributos asociados como nombre, cantidad y precio, se evidenció la eficiencia del ABB como estructura dinámica y ordenada.

Desde la etapa de investigación hasta la validación del programa, el proceso no solo facilitó una comprensión profunda del comportamiento recursivo y jerárquico de los árboles binarios, sino que también demostró su aplicabilidad directa en contextos reales, como el comercio y la administración de inventarios. Este trabajo integró teoría y práctica de forma coherente, resultando en una herramienta sólida, clara y adaptable.

BIBLIOGRAFÍA

Web:

“Estructuras de Datos II”

https://estructurasite.wordpress.com/

Blancarte, O. (2014, 22 de agosto). *Estructura de datos – Árboles*. Blog Oscar Blancarte – Arquitectura de Software. https://www.oscarblancarteblog.com/2014/08/22/estructura-de-datos-arboles/

Youtube:

“¿Qué son y cómo funcionan los árboles? | Ejemplo de implementación” (hasta el minuto 10:15): https://www.youtube.com/watch?v=tBaOQeyXYqg&t=256s

Cátedra Virtual:

“Datos Complejos”

https://tup.sied.utn.edu.ar/course/view.php?id=12&section=67

Udemy:

“Universidad Python - Cero a Experto, Sección 11: Clases y Objetos en Python, Sección 12: Programación Orientada a Objetos (POO) en Python”

https://www.udemy.com/course/universidad-python-desde-cero-hasta-experto-django-flask-rest-web/?couponCode=KEEPLEARNING

ANEXOS

Menú de opciones:

```
----TP-Integrador---- MENÚ ----
1. Agregar producto
2. Buscar producto
3. Modificar producto
4. Salir
--------------------------------
Seleccione una opción:
```

Búsqueda de producto por su código:

```
--------------------------------
Seleccione una opción: 2
Ingrese código del producto a buscar: 11

Producto encontrado:
Código: 11
Nombre: tv smart vision 60p 4k
Cantidad: 22
Precio: $1000
```

Modificación de producto:

```
--------------------------------
Seleccione una opción: 3
Ingrese código del producto a modificar: 11
Nuevo nombre (actual: tv smart vision 60p 4k): lavarropas
Nueva cantidad (actual: 22): 18
Nuevo precio (actual: 1000): 500
Producto modificado con exito.
```

Agregar producto (se evita la duplicidad):

```
--------------------------------
Seleccione una opción: 1
Ingrese código del producto (número entero): 11

El codigo ingresado ya pertenece a un producto existente: lavarropas
```

Agregar producto:

```
--------------------------------
Seleccione una opción: 1
Ingrese código del producto (número entero): 123
Ingrese nombre del producto: dron 4k
Ingrese cantidad del producto (número entero): 23
Ingrese precio del producto(sin '$'): 1200

Producto agregado con éxito:
Código: 123
Nombre: dron 4k
Cantidad: 23
Precio: $1200.0
```

Finalizar programa:

```
--------------------------------
Seleccione una opción: 4
Saliendo del programa...
```